```
#|
 Copyright (C) 1994 by Computational Logic, Inc.  All Rights Reserved.

 You may copy and distribute verbatim copies of this Nqthm-1992 event script as
 you receive it, in any medium, including embedding it verbatim in derivative
 works, provided that you conspicuously and appropriately publish on each copy
 a valid copyright notice "Copyright (C) 1994 by Computational Logic, Inc.  All
 Rights Reserved."

 NO WARRANTY

 Computational Logic, Inc. PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT
 IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Computational Logic, Inc. BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.
|#

; FM9001 Piton

; J Strother Moore
; October, 1991

; History

; This file contains the complete script of the events establishing
; that Piton is correctly implemented on top of the FM9001.  The
; original correctness proof for Piton was done for the FM8502 and is
; reported accurately and completely in CLI Tech Report 22.  That
; report is still basically accurate.  FM9001 differs from FM8502
; primarily in that instructions are in a slightly different format
; and the instruction set is slightly different.  However, when Hunt
; and Brock developed FM9001 from FM8502 they explicitly considered
; Piton's use of the FM8502 instruction set.  For each FM8502
; instruction-instance used by the Piton compiler, they included an
; FM9001 instruction that could provide the same functionality.  Thus,
```

```
; to port the Piton implementation from FM8502 to FM9001 it was only
; necessary to change the code linkers, essentially from
; LINK-INSTR-WORD on down.

; However, porting the proof from FM8502 to FM9001 was harder than
; suggested by the instruction set changes alone.  FM9001 uses a
; different formal representation of memory -- a binary tree instead
; of a linear list -- and uses a different formalization of bit
; vectors -- lists of Booleans instead of the BITV shell.  But most
; dramatically, the FM8502 proof used an experimental version of NQTHM
; that was supposed to become the released version.  That experimental
; NQTHM, called XNQTHM, provided "books".  Indeed, it was the FM8502
; Piton effort that caused books to be invented in the first place.
; With books, the XNQTHM user could develop a large body of lemmas in
; one session and then, in the course of later sessions, extract
; selected results to build upon.  Thus, books were incremental as
; opposed to the monolithic "core dump" view of state saving provided
; by NQTHM's lib files.  The FM8502 Piton proofs used four different
; (major) books: one to handle the p->r diagram, one for r->i, one for
; i->m (where we also handled m=FM8502), and then one in which the
; main correctness result was derived by extracting the key theorems
; from the three preceding books.  In addition, the original
; development devoted a book each to the definitions of the various
; machines (p, r, i, and m) and to the implementation of the compiler
; and link-assembler so they could be loaded and run in isolation from
; the correctness proof.  (This was in fact necessary since, for
; example, both the p->r book and the r->i book both needed access to
; the definition of the r machine and the implementation.)  Finally, I
; redid the FM8502 correctness proof in book form so that Piton could
; build on the definition of FM8502 without the clutter of its
; correctness proof or gate-level implementation.

; In the summer of 1989, Boyer and I began the process of "blessing"
; XNQTHM in preparation for its release as the new NQTHM.  However,
; after several months of trying to combine macros and books, we
; decided to start afresh.  We thereby embarked on the NTHM Project,
; where books are central to the whole design.  That is another story.
; However, one impact of the decision not to release XNQTHM was the
; need to reproduce the Piton proofs in a publicly available NQTHM.  I
; decided to wait until FM9001 was done and then to do the new Piton
; in NQTHM (rather than XNQTHM).

; This has had a major detrimental effect on the readability of the
; proof script.  Books allowed a clean hierarchical decomposition of
```

```
; the various layers of the problem.  This was lost when I
; (essentially) just concatenated the event lists from all of the
; Piton books into one massive script.  Of course, I had to start this
; proof effort in the library file created for the FM9001 correctness
; proof, the only NQTHM file containing "the" correct definition of
; FM9001.  To achieve the effect of isolation provided by books, each
; major section is separated from the others by thousands of DISABLE
; events which effectively erase the rule base developed thus far.  In
; addition, some names had to be changed because of clashes which were
; allowed under the book mechanism.  In short, the "new" proof of
; Piton is, in many ways, a step backward from the original one.  This
; was particularly frustrating for me since NTHM's books (which are
; implemented and seem to work) provide even greater functionality
; than XNQTHM's books.  The ideal thing to have done would have been
; for everybody else to take ten years' vacation while Boyer and I
; perfected and released NTHM, and then to develop the new FM9001 stack
; in NTHM.

; I ported the first of Piton's 3 commutative diagrams to the FM9001
; in May, 1991.  At that time, the FM9001 correctness proof was not
; completed.  I "borrowed" a lib file from Bishop that contained the
; majority of the proof and did the bottom-most diagram in that lib.
; Actually, I introduced a fourth layer to Piton: the link from M to
; FM9001 (the corresponding level in the FM8502 proof was an equality)
; has become a full-fledged commutative diagram that handles the
; memory.  See M->FM9001, which is now the fourth application in LOAD,
; and ONE-WAY-CORRESPONDENCE-M-FM9001.  In one week in May, I
; successfully proved that the M machine was implemented on FM9001.  I
; then waited until my decks were clear (October, 1991) to port the
; rest of the proof.  It has taken roughly 2 weeks to do the rest.

; A major new proof burden was introduced by the new memory representation.
; See the events surrounding PLAUSIBLE-DATA-LINK-TABLEP.  The basic problem
; was that in order to display the FM9001 memory to recover the M memory (i.e.,
; to get the "up" arrow in the bottom-most diagram), it was necessary to know
; that no address exceeded the size the of FM9001 memory.  This in turn
; required showing that the size of the final memory was the same as that
; for the initial memory.  This in turn forced us to prove the analogous
; fact all the way up, i.e., for M, I, R, and P.  At the higher levels, of
; course, what we proved was that the machines preserved the "signatures"
; of the program and data segments.  While this work could have been
; rather naturally done at the time we were dealing with the respective
; machines in their component diagrams, we did not recognize the need for the
; theorems until we had redone all of the old diagrams and were trying to
```

```
; combine everything at the very end.  Thus, you will see at the end of this
; script some hideous lemmas about the r and p machines where we tediously
; enable things that were "naturally" enabled earlier in the script.

; The book structure of the original proof can be seen by searching
; for comments of the form "; Book:".

; November, 1991: After the first successful port to FM9001 a new
; patch was installed: The loader was changed to allow us to specify
; the location at which the Piton data segment was laid down in
; memory.  This was done so that part of the data segment could be
; memory mapped to allow us to do some output from Piton.  This was
; achieved by swapping the positions of the prog segment and the data
; segment (in the original Piton work, the prog segment was laid out
; first, followed by the data segment and then the system data).  Now
; the data segment is laid out first, then the program segment, and
; then the system data segment.  Furthermore, all of this is laid out
; starting at a given location, generally named the LOAD-ADDR, and the
; contents of memory below that point is specified by an oracle,
; generally named BOOT-LST.

; This change has had drastic effects rippling through the Piton
; proof.

; This takes roughly 8 hours on a Sparc 2 in AKCL and generates 30Mb
; of .proof output and a .lib file of about 20Mb.
```

EVENT: Start with the library "fm9001-replay" using the compiled version.

EVENT: Set the status of all events. The status of each event is to be set as follows. Each 'boot-strap' satellite is to be returned to the initial condition. Each 'add-shell' satellite is to be enabled. enabled. Anything not otherwise mentioned is to be disabled. Name this event 'close-data-base-1'.

```
; Book:  Implementation.events

; We first define the implementation of Piton.  It is odd that we
; start there rather than with the semantics of the P machine.
; Logically and historically we started with the semantics.  But we
; are building the proof from the bottom up and in the beginning all
; we have is FM9001 and the implementation and we will build the m
; machine to explain the link part of the implementation, the i
```

```
; machine to build the code generator part, etc.
```

DEFINITION:
incr $(c,\ x)$
$=$ **if** $x \simeq$ **nil then nil**
else cons (xor $(c,$ car $(x))$, incr $(c \wedge$ car (x), cdr $(x)))$ **endif**

DEFINITION:
bitn $(x,\ n)$
$=$ **if** $n \simeq \texttt{0}$ **then f**
elseif $n = \texttt{1}$ **then** car (x)
else bitn (cdr (x), $n - 1)$ **endif**

EVENT: Add the shell *p-state*, with recognizer function symbol *p-statep* and 9 accessors: *p-pc*, with type restriction (none-of) and default value zero; *p-ctrl-stk*, with type restriction (none-of) and default value zero; *p-temp-stk*, with type restriction (none-of) and default value zero; *p-prog-segment*, with type restriction (none-of) and default value zero; *p-data-segment*, with type restriction (none-of) and default value zero; *p-max-ctrl-stk-size*, with type restriction (none-of) and default value zero; *p-max-temp-stk-size*, with type restriction (none-of) and default value zero; *p-word-size*, with type restriction (none-of) and default value zero; *p-psw*, with type restriction (none-of) and default value zero.

DEFINITION:
errorp $(psw) = ((psw \neq \texttt{'run}) \wedge (psw \neq \texttt{'halt}))$

DEFINITION: definition $(name,\ alist) =$ assoc $(name,\ alist)$

DEFINITION:
strip-cdrs $(alist)$
$=$ **if** $alist \simeq$ **nil then nil**
else cons (cdar $(alist)$, strip-cdrs (cdr $(alist)))$ **endif**

DEFINITION: name $(d) =$ car (d)

DEFINITION: formal-vars $(d) =$ cadr (d)

DEFINITION: temp-var-dcls $(d) =$ caddr (d)

DEFINITION: program-body $(d) =$ cdddr (d)

DEFINITION:
local-vars $(d) =$ append (formal-vars (d), strip-cars (temp-var-dcls $(d)))$

DEFINITION: adp-name $(adp) =$ car (adp)

DEFINITION: adp-offset $(adp) = \text{cdr}\,(adp)$

DEFINITION:
sub-adp $(adp,\ n) = \text{cons}\,(\text{adp-name}\,(adp),\ \text{adp-offset}\,(adp) - n)$

DEFINITION:
get $(n,\ lst)$
$=$ **if** $n \simeq \texttt{0}$ **then** car (lst)
else get $(n - 1,\ \text{cdr}\,(lst))$ **endif**

DEFINITION: tag $(type,\ obj) = \text{list}\,(type,\ obj)$

DEFINITION: type $(const) = \text{car}\,(const)$

DEFINITION: untag $(const) = \text{cadr}\,(const)$

DEFINITION: area-name $(x) = \text{adp-name}\,(\text{untag}\,(x))$

DEFINITION:
sub-addr $(addr,\ n) = \text{tag}\,(\text{type}\,(addr),\ \text{sub-adp}\,(\text{untag}\,(addr),\ n))$

DEFINITION: top $(stk) = \text{car}\,(stk)$

DEFINITION: pop $(stk) = \text{cdr}\,(stk)$

DEFINITION: dl $(lab,\ comment,\ ins) = \text{list}\,(\texttt{'dl},\ lab,\ comment,\ ins)$

DEFINITION: labelledp $(x) = (\text{car}\,(x) = \texttt{'dl})$

DEFINITION:
unlabel (x)
$=$ **if** labelledp (x) **then** cadddr (x)
else x **endif**

DEFINITION:
find-label $(x,\ lst)$
$=$ **if** $lst \simeq$ **nil** **then** $\texttt{0}$
elseif labelledp $(\text{car}\,(lst)) \land (x = \text{cadar}\,(lst))$ **then** $\texttt{0}$
else $1 +$ find-label $(x,\ \text{cdr}\,(lst))$ **endif**

DEFINITION:
pc $(lab,\ program)$
$=$ tag $(\texttt{'pc},\ \text{cons}\,(\text{name}\,(program),\ \text{find-label}\,(lab,\ \text{program-body}\,(program))))$

DEFINITION: bindings $(frame) = \text{car}\,(frame)$

DEFINITION: ret-pc $(frame) = \text{cadr}\,(frame)$

DEFINITION:
p-frame-size (*frame*) = (2 + length (bindings (*frame*)))

DEFINITION:
p-ctrl-stk-size (*ctrl-stk*)
= **if** *ctrl-stk* $\simeq$ **nil** **then** 0
  else p-frame-size (top (*ctrl-stk*))
      + p-ctrl-stk-size (cdr (*ctrl-stk*)) **endif**

DEFINITION:
icode-call (*ins*, *pcn*, *program*)
= list (**'(cpush_*)**,
      tag (**'pc**, cons (name (*program*), 1 + *pcn*)),
      '(jump_*),
      tag (**'pc**, cons (cadr (*ins*), **'(prelude)**)))

DEFINITION:
generate-prelude1 (*temp-var-dcls*)
= **if** *temp-var-dcls* $\simeq$ **nil** **then nil**
  else cons (**'(cpush_*)**,
        cons (cadar (*temp-var-dcls*),
          generate-prelude1 (cdr (*temp-var-dcls*)))) **endif**

DEFINITION:
generate-prelude2 (*formal-vars*)
= **if** *formal-vars* $\simeq$ **nil** **then nil**
  else cons (**'(cpush_<tsp>+)**,
        generate-prelude2 (cdr (*formal-vars*))) **endif**

DEFINITION:
generate-prelude (*program*)
= append (cons (dl (cons (name (*program*), **'(prelude)**),
            '(prelude),
            '(cpush_cfp)),
          '((move_cfp_csp))),
      append (generate-prelude1 (reverse (temp-var-dcls (*program*))),
            generate-prelude2 (formal-vars (*program*))))

DEFINITION:
find-position-of-var (*var*, *lst*)
= **if** *lst* $\simeq$ **nil** **then** 0
  elseif *var* = car (*lst*) **then** 0
  else 1 + find-position-of-var (*var*, cdr (*lst*)) **endif**

DEFINITION:
offset-from-csp (*var*, *program*)
= find-position-of-var (*var*, local-vars (*program*))

DEFINITION:
```
generate-postlude (program)
=   cons (dl (cons (name (program), length (program-body (program))),
              '(postlude),
              '(move_csp_cfp)),
          '((cpop_cfp) (cpop_pc)))
```

DEFINITION:
```
icode-ret (ins, pcn, program)
=   list ('(jump_*),
          tag ('pc, cons (name (program), length (program-body (program)))))
```

DEFINITION:
```
icode-locn (ins, pcn, program)
=   cons ('(move_x_*),
          cons (tag ('nat, offset-from-csp (cadr (ins), program)),
                '((add_x{n}_csp)
                  (move_x_<x{s}>)
                  (add_x{n}_csp)
                  (tpush_<x{s}>))))
```

DEFINITION:
```
icode-push-constant (ins, pcn, program)
=   list ('(tpush_*),
          if cadr (ins) = 'pc then tag ('pc, cons (name (program), 1 + pcn))
          elseif cadr (ins) ≃ nil then pc (cadr (ins), program)
          else cadr (ins) endif)
```

DEFINITION:
```
icode-push-local (ins, pcn, program)
=   cons ('(move_x_*),
          cons (tag ('nat, offset-from-csp (cadr (ins), program)),
                '((add_x{n}_csp) (tpush_<x{s}>))))
```

DEFINITION:
```
icode-push-global (ins, pcn, program)
=   cons ('(move_x_*),
          cons (tag ('addr, cons (cadr (ins), 0)), '((tpush_<x{a}>))))
```

DEFINITION:
```
icode-push-ctrl-stk-free-size (ins, pcn, program)
=   '((move_x_*)
      (sys-addr (full-ctrl-stk-addr . 0))
      (move_x_<x{s}>)
      (tpush_csp)
      (sub_<tsp>{s}_x{s}))
```

DEFINITION:
icode-push-temp-stk-free-size (*ins*, *pcn*, *program*)
= `'((move_x_*)`
`(sys-addr (full-temp-stk-addr . 0))`
`(move_x_<x{s}>)`
`(tpush_tsp)`
`(sub_<tsp>{s}_x{s}))`

DEFINITION:
icode-push-temp-stk-index (*ins*, *pcn*, *program*)
= cons (`'(move_y_tsp)`,
cons (`'(move_x_*)`,
cons (`'(sys-addr (empty-temp-stk-addr . 0))`,
cons (`'(move_x_<x{s}>)`,
cons (`'(sub_<z>_x{s}_y{s})`,
cons (`'(tpush_x)`,
cons (`'(move_x_*)`,
cons (tag (`'nat`, 1 + cadr (*ins*)),
`'((sub_<tsp>{n}_x{n}))`))))))))

DEFINITION:
icode-jump-if-temp-stk-full (*ins*, *pcn*, *program*)
= cons (`'(move_x_tsp)`,
cons (`'(move_y_*)`,
cons (`'(sys-addr (full-temp-stk-addr . 0))`,
cons (`'(move_y_<y{s}>)`,
cons (`'(sub_<z>_x{s}_y{s})`,
cons (`'(move_x_*)`,
cons (pc (cadr (*ins*), *program*),
`'((jump-z_x))`)))))))

DEFINITION:
icode-jump-if-temp-stk-empty (*ins*, *pcn*, *program*)
= cons (`'(move_y_tsp)`,
cons (`'(move_x_*)`,
cons (`'(sys-addr (empty-temp-stk-addr . 0))`,
cons (`'(move_x_<x{s}>)`,
cons (`'(sub_<z>_x{s}_y{s})`,
cons (`'(move_x_*)`,
cons (pc (cadr (*ins*), *program*),
`'((jump-z_x))`)))))))

DEFINITION: icode-pop (*ins*, *pcn*, *program*) = `'((tpop_x))`

DEFINITION:

icode-pop* (*ins*, *pcn*, *program*)
= list ('(add_tsp_*{n}), tag ('nat, cadr (*ins*)))

DEFINITION:
icode-popn (*ins*, *pcn*, *program*) = '((tpop_x) (add_tsp_x{n}))

DEFINITION:
icode-pop-local (*ins*, *pcn*, *program*)
= cons ('(move_x_*),
cons (tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'((add_x{n}_csp) (tpop_<x{s}>))))

DEFINITION:
icode-pop-global (*ins*, *pcn*, *program*)
= cons ('(move_x_*),
cons (tag ('addr, cons (cadr (*ins*), 0)), '((tpop_<x{a}>))))

DEFINITION:
icode-pop-locn (*ins*, *pcn*, *program*)
= cons ('(move_x_*),
cons (tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'((add_x{n}_csp)
(move_x_<x{s}>)
(add_x{n}_csp)
(tpop_<x{s}>))))

DEFINITION:
icode-pop-call (*ins*, *pcn*, *program*)
= cons ('(tpop_x),
cons ('(cpush_*),
cons (tag ('pc, cons (name (*program*), 1 + *pcn*)),
'((jump_x{subr})))))

DEFINITION:
icode-fetch-temp-stk (*ins*, *pcn*, *program*)
= '((tpop_y)
(incr_y_y{n})
(move_x_*)
(sys-addr (empty-temp-stk-addr . 0))
(move_x_<x{s}>)
(sub_x{s}_y{n})
(tpush_<x{s}>))

DEFINITION:
icode-deposit-temp-stk (*ins*, *pcn*, *program*)

= '((tpop_y)
(incr_y_y{n})
(move_x_*)
(sys-addr (empty-temp-stk-addr . 0))
(move_x_<x{s}>)
(sub_x{s}_y{n})
(tpop_<x{s}>))

DEFINITION:
icode-jump (*ins*, *pcn*, *program*) = list ('(jump_*), pc (cadr (*ins*), *program*))

DEFINITION:
jump_*-lst (*lst*, *program*)
= **if** *lst* $\simeq$ **nil then nil**
else cons ('(jump_*),
cons (pc (car (*lst*), *program*),
jump_*-lst (cdr (*lst*), *program*))) **endif**

DEFINITION:
icode-jump-case (*ins*, *pcn*, *program*)
= append ('((tpop_x) (add_x_x{n}) (add_pc_x{n})),
jump_*-lst (cdr (*ins*), *program*))

DEFINITION:
icode-pushj (*ins*, *pcn*, *program*)
= list ('(tpush_*),
tag ('pc, cons (name (*program*), 1 + *pcn*)),
'(jump_*),
pc (cadr (*ins*), *program*))

DEFINITION: icode-popj (*ins*, *pcn*, *program*) = '((tpop_pc))

DEFINITION:
icode-set-local (*ins*, *pcn*, *program*)
= cons ('(move_x_*),
cons (tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'((add_x{n}_csp) (move_<x{s}>_<tsp>))))

DEFINITION:
icode-set-global (*ins*, *pcn*, *program*)
= cons ('(move_x_*),
cons (tag ('addr, cons (cadr (*ins*), 0)),
'((move_<x{a}>_<tsp>))))

DEFINITION:

```
icode-test-nat-and-jump (ins, pcn, program)
=   if cadr (ins) = 'zero
    then cons ('(tpop{n}_<z>_y),
                cons ('(move_x_*),
                     cons (pc (caddr (ins), program), '((jump-z_x)))))
    else cons ('(tpop{n}_<z>_y),
               cons ('(move_x_*),
                    cons (pc (caddr (ins), program), '((jump-nz_x))))) endif
```

Definition:

```
icode-test-int-and-jump (ins, pcn, program)
=   case on car (cdr (ins)):
    case = zero
    then cons ('(tpop{i}_<zn>_y),
                cons ('(move_x_*),
                     cons (pc (caddr (ins), program), '((jump-z_x)))))
    case = not-zero
     then cons ('(tpop{i}_<zn>_y),
                 cons ('(move_x_*),
                      cons (pc (caddr (ins), program), '((jump-nz_x)))))
    case = neg
     then cons ('(tpop{i}_<zn>_y),
                 cons ('(move_x_*),
                      cons (pc (caddr (ins), program), '((jump-n_x)))))
    case = not-neg
     then cons ('(tpop{i}_<zn>_y),
                 cons ('(move_x_*),
                      cons (pc (caddr (ins), program), '((jump-nn_x)))))
    case = pos
     then list ('(tpop{i}_<zn>_y),
                '(move_x_*),
                tag ('pc, cons (name (program), 1 + pcn)),
                '(jump-n_x),
                '(jump-z_x),
                '(jump_*),
                pc (caddr (ins), program))
    otherwise cons ('(tpop{i}_<zn>_y),
                     cons ('(move_x_*),
                          cons (pc (caddr (ins), program),
                                '((jump-n_x) (jump-z_x))))) endcase
```

Definition:

```
icode-test-bool-and-jump (ins, pcn, program)
=   if cadr (ins) = 't
```

then cons ('(tpop{b}_<z>_y),
cons ('(move_x_*),
cons (pc (caddr (*ins*), *program*), '((jump-nz_x)))))
else cons ('(tpop{b}_<z>_y),
cons ('(move_x_*),
cons (pc (caddr (*ins*), *program*), '((jump-z_x))))) **endif**

DEFINITION:
icode-test-bitv-and-jump (*ins*, *pcn*, *program*)
= **if** cadr (*ins*) = 'all-zero
then cons ('(tpop{v}_<z>_y),
cons ('(move_x_*),
cons (pc (caddr (*ins*), *program*), '((jump-z_x)))))
else cons ('(tpop{v}_<z>_y),
cons ('(move_x_*),
cons (pc (caddr (*ins*), *program*), '((jump-nz_x))))) **endif**

DEFINITION: icode-no-op (*ins*, *pcn*, *program*) = '((move_x_x))

DEFINITION:
icode-add-addr (*ins*, *pcn*, *program*)
= '((tpop_x) (add_<tsp>{a}_x{n}))

DEFINITION:
icode-sub-addr (*ins*, *pcn*, *program*)
= '((tpop_x) (sub_<tsp>{a}_x{n}))

DEFINITION:
icode-eq (*ins*, *pcn*, *program*)
= '((tpop_x)
(xor_<z>_<tsp>_x)
(xor_<tsp>_<tsp>)
(move-z_<tsp>_*)
(bool t))

DEFINITION:
icode-lt-addr (*ins*, *pcn*, *program*)
= '((tpop_x)
(sub_<c>_<tsp>{a}_x{a})
(xor_<tsp>_<tsp>)
(move-c_<tsp>_*)
(bool t))

DEFINITION:
icode-fetch (*ins*, *pcn*, *program*) = '((tpop_x) (tpush_<x{a}>))

DEFINITION:
icode-deposit (*ins*, *pcn*, *program*) =
```
'((tpop_x) (tpop_<x{a}>))
```

DEFINITION:
icode-add-int (*ins*, *pcn*, *program*)
```
= '((tpop_x) (add_<tsp>{i}_x{i}))
```

DEFINITION:
icode-add-int-with-carry (*ins*, *pcn*, *program*)
```
= '((tpop_x)
    (tpop_y)
    (asr_<c>_<tsp>_<tsp>{b})
    (addc_<v>_x{i}_y{i})
    (move-v_<tsp>_*)
    (bool t)
    (tpush_x))
```

DEFINITION:
icode-add1-int (*ins*, *pcn*, *program*) =
```
'((incr_<tsp>_<tsp>{i}))
```

DEFINITION:
icode-sub-int (*ins*, *pcn*, *program*)
```
= '((tpop_x) (sub_<tsp>{i}_x{i}))
```

DEFINITION:
icode-sub-int-with-carry (*ins*, *pcn*, *program*)
```
= '((tpop_y)
    (tpop_x)
    (asr_<c>_<tsp>_<tsp>{b})
    (subb_<v>_x{i}_y{i})
    (move-v_<tsp>_*)
    (bool t)
    (tpush_x))
```

DEFINITION:
icode-sub1-int (*ins*, *pcn*, *program*) =
```
'((decr_<tsp>_<tsp>{i}))
```

DEFINITION:
icode-neg-int (*ins*, *pcn*, *program*) =
```
'((neg_<tsp>_<tsp>{i}))
```

DEFINITION:
icode-lt-int (*ins*, *pcn*, *program*)
```
= '((tpop_x)
    (sub_<nv>_<tsp>{i}_x{i})
    (move_<tsp>_*)
```

```
    (bool f)
    (move-v_<tsp>_*)
    (bool t)
    (move_x_*)
    (bool f)
    (move-n_x_*)
    (bool t)
    (xor_<tsp>{b}_x{b}))
```

DEFINITION:
icode-int-to-nat (*ins*, *pcn*, *program*) = '((int-to-nat))

DEFINITION:
icode-add-nat (*ins*, *pcn*, *program*)
= '((tpop_x) (add_<tsp>{n}_x{n}))

DEFINITION:
icode-add-nat-with-carry (*ins*, *pcn*, *program*)

```
=  '((tpop_x)
    (tpop_y)
    (asr_<c>_<tsp>_<tsp>{b})
    (addc_<c>_x{n}_y{n})
    (move-c_<tsp>_*)
    (bool t)
    (tpush_x))
```

DEFINITION:
icode-add1-nat (*ins*, *pcn*, *program*) = '((incr_<tsp>_<tsp>{n}))

DEFINITION:
icode-sub-nat (*ins*, *pcn*, *program*)
= '((tpop_x) (sub_<tsp>{n}_x{n}))

DEFINITION:
icode-sub-nat-with-carry (*ins*, *pcn*, *program*)

```
=  '((tpop_y)
    (tpop_x)
    (asr_<c>_<tsp>_<tsp>{b})
    (subb_<c>_x{n}_y{n})
    (move-c_<tsp>_*)
    (bool t)
    (tpush_x))
```

DEFINITION:
icode-sub1-nat (*ins*, *pcn*, *program*) = '((decr_<tsp>_<tsp>{n}))

Definition:
icode-lt-nat (*ins*, *pcn*, *program*)

```
=  '((tpop_x)
     (sub_<c>_<tsp>{n}_x{n})
     (xor_<tsp>_<tsp>)
     (move-c_<tsp>_*)
     (bool t))
```

Definition:
icode-mult2-nat (*ins*, *pcn*, *program*) = `'((add_<tsp>_<tsp>{n}))`

Definition:
icode-mult2-nat-with-carry-out (*ins*, *pcn*, *program*)

```
=  '((tpop_x)
     (add_<c>_x_x{n})
     (tpush_*)
     (bool f)
     (move-c_<tsp>_*)
     (bool t)
     (tpush_x))
```

Definition:
icode-div2-nat (*ins*, *pcn*, *program*)

```
=  '((tpop_<c>_x)
     (lsr_<c>_x_x{n})
     (tpush_x)
     (tpush_*)
     (nat 0)
     (move-c_<tsp>_*)
     (nat 1))
```

Definition:
icode-or-bitv (*ins*, *pcn*, *program*) = `'((tpop_x) (or_<tsp>{v}_x{v}))`

Definition:
icode-and-bitv (*ins*, *pcn*, *program*)
= `'((tpop_x) (and_<tsp>{v}_x{v}))`

Definition:
icode-not-bitv (*ins*, *pcn*, *program*) = `'((not_<tsp>_<tsp>{v}))`

Definition:
icode-xor-bitv (*ins*, *pcn*, *program*)
= `'((tpop_x) (xor_<tsp>{v}_x{v}))`

DEFINITION:
icode-rsh-bitv (*ins*, *pcn*, *program*) = '((lsr_<tsp>_<tsp>{v}))

DEFINITION:
icode-lsh-bitv (*ins*, *pcn*, *program*) = '((add_<tsp>_<tsp>{v}))

DEFINITION:
icode-or-bool (*ins*, *pcn*, *program*) = '((tpop_x) (or_<tsp>{b}_x{b}))

DEFINITION:
icode-and-bool (*ins*, *pcn*, *program*)
= '((tpop_x) (and_<tsp>{b}_x{b}))

DEFINITION:
icode-not-bool (*ins*, *pcn*, *program*)
= '((xor_<tsp>{b}_*{b}) (bool t))

DEFINITION:
icode1 (*ins*, *pcn*, *prog*)
= **case on** car (*ins*):
case = *call*
then icode-call (*ins*, *pcn*, *prog*)
case = *ret*
then icode-ret (*ins*, *pcn*, *prog*)
case = *locn*
then icode-locn (*ins*, *pcn*, *prog*)
case = *push-constant*
then icode-push-constant (*ins*, *pcn*, *prog*)
case = *push-local*
then icode-push-local (*ins*, *pcn*, *prog*)
case = *push-global*
then icode-push-global (*ins*, *pcn*, *prog*)
case = *push-ctrl-stk-free-size*
then icode-push-ctrl-stk-free-size (*ins*, *pcn*, *prog*)
case = *push-temp-stk-free-size*
then icode-push-temp-stk-free-size (*ins*, *pcn*, *prog*)
case = *push-temp-stk-index*
then icode-push-temp-stk-index (*ins*, *pcn*, *prog*)
case = *jump-if-temp-stk-full*
then icode-jump-if-temp-stk-full (*ins*, *pcn*, *prog*)
case = *jump-if-temp-stk-empty*
then icode-jump-if-temp-stk-empty (*ins*, *pcn*, *prog*)
case = *pop*
then icode-pop (*ins*, *pcn*, *prog*)
case = *pop**

then icode-pop* (*ins*, *pcn*, *prog*)
case = *popn*
then icode-popn (*ins*, *pcn*, *prog*)
case = *pop-local*
then icode-pop-local (*ins*, *pcn*, *prog*)
case = *pop-global*
then icode-pop-global (*ins*, *pcn*, *prog*)
case = *pop-locn*
then icode-pop-locn (*ins*, *pcn*, *prog*)
case = *pop-call*
then icode-pop-call (*ins*, *pcn*, *prog*)
case = *fetch-temp-stk*
then icode-fetch-temp-stk (*ins*, *pcn*, *prog*)
case = *deposit-temp-stk*
then icode-deposit-temp-stk (*ins*, *pcn*, *prog*)
case = *jump*
then icode-jump (*ins*, *pcn*, *prog*)
case = *jump-case*
then icode-jump-case (*ins*, *pcn*, *prog*)
case = *pushj*
then icode-pushj (*ins*, *pcn*, *prog*)
case = *popj*
then icode-popj (*ins*, *pcn*, *prog*)
case = *set-local*
then icode-set-local (*ins*, *pcn*, *prog*)
case = *set-global*
then icode-set-global (*ins*, *pcn*, *prog*)
case = *test-nat-and-jump*
then icode-test-nat-and-jump (*ins*, *pcn*, *prog*)
case = *test-int-and-jump*
then icode-test-int-and-jump (*ins*, *pcn*, *prog*)
case = *test-bool-and-jump*
then icode-test-bool-and-jump (*ins*, *pcn*, *prog*)
case = *test-bitv-and-jump*
then icode-test-bitv-and-jump (*ins*, *pcn*, *prog*)
case = *no-op*
then icode-no-op (*ins*, *pcn*, *prog*)
case = *add-addr*
then icode-add-addr (*ins*, *pcn*, *prog*)
case = *sub-addr*
then icode-sub-addr (*ins*, *pcn*, *prog*)
case = *eq*
then icode-eq (*ins*, *pcn*, *prog*)
case = *lt-addr*

then icode-lt-addr (*ins*, *pcn*, *prog*)
case = *fetch*
then icode-fetch (*ins*, *pcn*, *prog*)
case = *deposit*
then icode-deposit (*ins*, *pcn*, *prog*)
case = *add-int*
then icode-add-int (*ins*, *pcn*, *prog*)
case = *add-int-with-carry*
then icode-add-int-with-carry (*ins*, *pcn*, *prog*)
case = *add1-int*
then icode-add1-int (*ins*, *pcn*, *prog*)
case = *sub-int*
then icode-sub-int (*ins*, *pcn*, *prog*)
case = *sub-int-with-carry*
then icode-sub-int-with-carry (*ins*, *pcn*, *prog*)
case = *sub1-int*
then icode-sub1-int (*ins*, *pcn*, *prog*)
case = *neg-int*
then icode-neg-int (*ins*, *pcn*, *prog*)
case = *lt-int*
then icode-lt-int (*ins*, *pcn*, *prog*)
case = *int-to-nat*
then icode-int-to-nat (*ins*, *pcn*, *prog*)
case = *add-nat*
then icode-add-nat (*ins*, *pcn*, *prog*)
case = *add-nat-with-carry*
then icode-add-nat-with-carry (*ins*, *pcn*, *prog*)
case = *add1-nat*
then icode-add1-nat (*ins*, *pcn*, *prog*)
case = *sub-nat*
then icode-sub-nat (*ins*, *pcn*, *prog*)
case = *sub-nat-with-carry*
then icode-sub-nat-with-carry (*ins*, *pcn*, *prog*)
case = *sub1-nat*
then icode-sub1-nat (*ins*, *pcn*, *prog*)
case = *lt-nat*
then icode-lt-nat (*ins*, *pcn*, *prog*)
case = *mult2-nat*
then icode-mult2-nat (*ins*, *pcn*, *prog*)
case = *mult2-nat-with-carry-out*
then icode-mult2-nat-with-carry-out (*ins*, *pcn*, *prog*)
case = *div2-nat*
then icode-div2-nat (*ins*, *pcn*, *prog*)
case = *or-bitv*

then icode-or-bitv $(ins,\ pcn,\ prog)$
case $=$ *and-bitv*
then icode-and-bitv $(ins,\ pcn,\ prog)$
case $=$ *not-bitv*
then icode-not-bitv $(ins,\ pcn,\ prog)$
case $=$ *xor-bitv*
then icode-xor-bitv $(ins,\ pcn,\ prog)$
case $=$ *rsh-bitv*
then icode-rsh-bitv $(ins,\ pcn,\ prog)$
case $=$ *lsh-bitv*
then icode-lsh-bitv $(ins,\ pcn,\ prog)$
case $=$ *or-bool*
then icode-or-bool $(ins,\ pcn,\ prog)$
case $=$ *and-bool*
then icode-and-bool $(ins,\ pcn,\ prog)$
case $=$ *not-bool*
then icode-not-bool $(ins,\ pcn,\ prog)$
otherwise `'((error))` **endcase**

DEFINITION:
dl-block $(lab,\ comment,\ block) = $ cons (dl $(lab,\ comment,$ car $(block))$, cdr $(block))$

DEFINITION:
icode $(ins,\ pcn,\ program)$
$=$ dl-block (cons (name $(program),\ pcn),\ ins,$ icode1 (unlabel $(ins),\ pcn,\ program))$

DEFINITION:
icompile-program-body $(lst,\ pcn,\ program)$
$=$ **if** $lst \simeq$ **nil then nil**
else append (icode (car $(lst),\ pcn,\ program)$,
icompile-program-body (cdr $(lst),\ 1 + pcn,\ program)$) **endif**

DEFINITION:
icompile-program $(program)$
$=$ cons (name $(program)$,
append (generate-prelude $(program)$,
append (icompile-program-body (program-body $(program)$,
`0`,
$program)$,
generate-postlude $(program)$)))

DEFINITION:
icompile $(programs)$
$=$ **if** $programs \simeq$ **nil then nil**
else cons (icompile-program (car $(programs)$),
icompile (cdr $(programs)$)) **endif**

DEFINITION:
segment-length (*segment*)
= **if** *segment* $\simeq$ **nil** **then** 0
else length (cdar (*segment*)) + segment-length (cdr (*segment*)) **endif**

DEFINITION:
total-p-system-size (*p*, *load-addr*)
= (*load-addr*
+ segment-length (p-data-segment (*p*))
+ segment-length (icompile (p-prog-segment (*p*)))
+ (1 + p-max-ctrl-stk-size (*p*))
+ (1 + p-max-temp-stk-size (*p*))
+ 3)

DEFINITION:
p-loadablep (*p*, *load-addr*)
= (total-p-system-size (*p*, *load-addr*) < exp (2, p-word-size (*p*)))

EVENT: Add the shell *r-state*, with recognizer function symbol *r-statep* and 15 accessors: *r-pc*, with type restriction (none-of) and default value zero; *r-cfp*, with type restriction (none-of) and default value zero; *r-csp*, with type restriction (none-of) and default value zero; *r-tsp*, with type restriction (none-of) and default value zero; *r-x*, with type restriction (none-of) and default value zero; *r-y*, with type restriction (none-of) and default value zero; *r-c-flg*, with type restriction (none-of) and default value zero; *r-v-flg*, with type restriction (none-of) and default value zero; *r-n-flg*, with type restriction (none-of) and default value zero; *r-z-flg*, with type restriction (none-of) and default value zero; *r-prog-segment*, with type restriction (none-of) and default value zero; *r-usr-data-segment*, with type restriction (none-of) and default value zero; *r-sys-data-segment*, with type restriction (none-of) and default value zero; *r-word-size*, with type restriction (none-of) and default value zero; *r-psw*, with type restriction (none-of) and default value zero.

DEFINITION:
nat-0s (*n*)
= **if** *n* $\simeq$ 0 **then nil**
else cons (tag (`'nat`, 0), nat-0s (*n* − 1)) **endif**

DEFINITION:
p->r_temp-stk (*temp-stk*, *max-temp-stk-size*)
= cons (`'tstk`,
append (nat-0s (*max-temp-stk-size* − length (*temp-stk*)),
append (*temp-stk*, list (tag (`'nat`, 0)))))

DEFINITION:
p->r_csp (*stk*, *max*)
= tag ('**sys-addr**, cons ('**cstk**, *max* − p-ctrl-stk-size (*stk*)))

DEFINITION:
p->r_cfp (*stk*, *max*) = sub-addr (p->r_csp (pop (*stk*), *max*), 2)

DEFINITION:
p->r_p-frame (*pframe*, *stk*, *max*)
= append (strip-cdrs (bindings (*pframe*)),
list (p->r_cfp (*stk*, *max*), ret-pc (*pframe*)))

DEFINITION:
p->r_ctrl-stk1 (*stk*, *max*)
= **if** *stk* ≃ **nil** **then** **nil**
else append (p->r_p-frame (top (*stk*), pop (*stk*), *max*),
p->r_ctrl-stk1 (pop (*stk*), *max*)) **endif**

DEFINITION:
p->r_ctrl-stk (*stk*, *max*)
= cons ('**cstk**,
append (nat-0s (*max* − p-ctrl-stk-size (*stk*)),
append (p->r_ctrl-stk1 (*stk*, *max*), list (tag ('**nat**, 0)))))

DEFINITION:
p->r_sys-data-segment (*ctrl-stk*, *max-ctrl-stk-size*, *temp-stk*, *max-temp-stk-size*)
= list (p->r_ctrl-stk (*ctrl-stk*, *max-ctrl-stk-size*),
p->r_temp-stk (*temp-stk*, *max-temp-stk-size*),
list ('**full-ctrl-stk-addr**, tag ('**sys-addr**, '**(cstk . 0)**)),
list ('**full-temp-stk-addr**, tag ('**sys-addr**, '**(tstk . 0)**)),
list ('**empty-temp-stk-addr**,
tag ('**sys-addr**, cons ('**tstk**, *max-temp-stk-size*))))

DEFINITION:
p->r_tsp (*stk*, *max*) = tag ('**sys-addr**, cons ('**tstk**, *max* − length (*stk*)))

DEFINITION:
p->r (*p*)
= r-state (p-pc (*p*),
p->r_cfp (p-ctrl-stk (*p*), p-max-ctrl-stk-size (*p*)),
p->r_csp (p-ctrl-stk (*p*), p-max-ctrl-stk-size (*p*)),
p->r_tsp (p-temp-stk (*p*), p-max-temp-stk-size (*p*)),
'**(nat 0)**,
'**(nat 0)**,
'**(bool f)**,

```
'(bool f),
'(bool f),
'(bool f),
p-prog-segment (p),
p-data-segment (p),
p->r_sys-data-segment (p-ctrl-stk (p),
                       p-max-ctrl-stk-size (p),
                       p-temp-stk (p),
                       p-max-temp-stk-size (p)),
p-word-size (p),
p-psw (p))
```

EVENT: Add the shell *i-state*, with recognizer function symbol *i-statep* and 15 accessors: *i-pc*, with type restriction (none-of) and default value zero; *i-cfp*, with type restriction (none-of) and default value zero; *i-csp*, with type restriction (none-of) and default value zero; *i-tsp*, with type restriction (none-of) and default value zero; *i-x*, with type restriction (none-of) and default value zero; *i-y*, with type restriction (none-of) and default value zero; *i-c-flg*, with type restriction (none-of) and default value zero; *i-v-flg*, with type restriction (none-of) and default value zero; *i-n-flg*, with type restriction (none-of) and default value zero; *i-z-flg*, with type restriction (none-of) and default value zero; *i-prog-segment*, with type restriction (none-of) and default value zero; *i-usr-data-segment*, with type restriction (none-of) and default value zero; *i-sys-data-segment*, with type restriction (none-of) and default value zero; *i-word-size*, with type restriction (none-of) and default value zero; *i-psw*, with type restriction (none-of) and default value zero.

DEFINITION:

```
r->i_pc (pc, programs)
=  tag ('ipc,
        cons (area-name (pc),
              find-label (untag (pc),
                          cdr (icompile-program (definition (area-name (pc),
                                                             programs))))))
```

DEFINITION:

```
r->i_psw (psw)
=  if psw = 'halt then 'run
   else psw endif
```

DEFINITION:

```
r->i (r)
=  i-state (r->i_pc (r-pc (r), r-prog-segment (r)),
            r-cfp (r),
```

r-csp (*r*),
r-tsp (*r*),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
icompile (r-prog-segment (*r*)),
r-usr-data-segment (*r*),
r-sys-data-segment (*r*),
r-word-size (*r*),
r->i_psw (r-psw (*r*)))

EVENT: Add the shell *m-state*, with recognizer function symbol *m-statep* and 6 accessors: *m-regs*, with type restriction (none-of) and default value zero; *m-c-flg*, with type restriction (none-of) and default value zero; *m-v-flg*, with type restriction (none-of) and default value zero; *m-n-flg*, with type restriction (none-of) and default value zero; *m-z-flg*, with type restriction (none-of) and default value zero; *m-mem*, with type restriction (none-of) and default value zero.

DEFINITION:
link-table-for-segment (*segment*, *addr0*)
= **if** *segment* $\simeq$ **nil then nil**
else cons (cons (caar (*segment*), *addr0*),
link-table-for-segment (cdr (*segment*),
addr0 + length (cdar (*segment*)))) **endif**

DEFINITION:
link-table-for-labels (*lst*, *addr0*)
= **if** *lst* $\simeq$ **nil then nil**
elseif labelledp (car (*lst*))
then cons (cons (cadar (*lst*), *addr0*),
link-table-for-labels (cdr (*lst*), 1 + *addr0*))
else link-table-for-labels (cdr (*lst*), 1 + *addr0*) **endif**

DEFINITION:
link-table-for-prog-labels (*segment*, *addr0*)
= **if** *segment* $\simeq$ **nil then nil**
else cons (cons (caar (*segment*),
link-table-for-labels (cdar (*segment*), *addr0*)),
link-table-for-prog-labels (cdr (*segment*),
addr0 + length (cdar (*segment*)))) **endif**

DEFINITION:
i-link-tables $(i,\ \mathit{load\text{-}addr})$
$=$ list (link-table-for-segment (i-prog-segment (i),
$\mathit{load\text{-}addr}$ + segment-length (i-usr-data-segment (i)))),
link-table-for-prog-labels (i-prog-segment (i),
$\mathit{load\text{-}addr}$ + segment-length (i-usr-data-segment (i)))),
link-table-for-segment (i-usr-data-segment (i), $\mathit{load\text{-}addr}$),
link-table-for-segment (i-sys-data-segment (i),
$\mathit{load\text{-}addr}$
+ segment-length (i-prog-segment (i))
+ segment-length (i-usr-data-segment (i))))

DEFINITION: prog-links ($\mathit{link\text{-}tables}$) = car ($\mathit{link\text{-}tables}$)

DEFINITION: prog-label-tables ($\mathit{link\text{-}tables}$) = cadr ($\mathit{link\text{-}tables}$)

DEFINITION: usr-data-links ($\mathit{link\text{-}tables}$) = caddr ($\mathit{link\text{-}tables}$)

DEFINITION: sys-data-links ($\mathit{link\text{-}tables}$) = cadddr ($\mathit{link\text{-}tables}$)

DEFINITION:
label-links (label, $\mathit{prog\text{-}label\text{-}tables}$)
$=$ cdr (assoc (adp-name (label), $\mathit{prog\text{-}label\text{-}tables}$))

DEFINITION:
base-address (name, $\mathit{link\text{-}table}$) = cdr (assoc ($\mathit{name}$, $\mathit{link\text{-}table}$))

DEFINITION:
find-containing-area-name (n, $\mathit{link\text{-}table}$)
$=$ **if** $\mathit{link\text{-}table} \simeq$ **nil then** 0
elseif cdr ($\mathit{link\text{-}table}$) $\simeq$ **nil then** caar ($\mathit{link\text{-}table}$)
elseif ($n \not<$ cdar ($\mathit{link\text{-}table}$)) $\wedge$ ($n <$ cdadr ($\mathit{link\text{-}table}$))
then caar ($\mathit{link\text{-}table}$)
else find-containing-area-name (n, cdr ($\mathit{link\text{-}table}$)) **endif**

DEFINITION:
invert-base-address (n, $\mathit{link\text{-}table}$)
$=$ find-containing-area-name (n, $\mathit{link\text{-}table}$)

DEFINITION:
label-address (label, $\mathit{prog\text{-}label\text{-}tables}$)
$=$ base-address (label, label-links (label, $\mathit{prog\text{-}label\text{-}tables}$))

DEFINITION:
assoc-cdrp (n, alist)
$=$ **if** $\mathit{alist} \simeq$ **nil then f**
elseif $n =$ cdar (alist) **then t**
else assoc-cdrp (n, cdr (alist)) **endif**

DEFINITION:
find-containing-label-table (n, *label-tables*)
= **if** *label-tables* $\simeq$ **nil** **then** **f**
elseif assoc-cdrp (n, cdar (*label-tables*)) **then** cdar (*label-tables*)
else find-containing-label-table (n, cdr (*label-tables*)) **endif**

DEFINITION:
invert-label-address (n, *prog-label-tables*)
= invert-base-address (n,
find-containing-label-table (n, *prog-label-tables*))

DEFINITION:
absolute-address (*adp*, *link-table*)
= (base-address (adp-name (*adp*), *link-table*) + adp-offset (*adp*))

DEFINITION:
invert-absolute-address (n, *link-table*)
= cons (find-containing-area-name (n, *link-table*),
n − base-address (find-containing-area-name (n, *link-table*),
link-table))

DEFINITION:
bitv-to-v (*lst*, *word-size*)
= **if** *word-size* $\simeq$ `0` **then** **nil**
else append (bitv-to-v (cdr (*lst*), *word-size* − 1),
list (**if** car (*lst*) = `0` **then** **f**
else **t** **endif**)) **endif**

DEFINITION:
v-to-bitv (v)
= **if** v $\simeq$ **nil** **then** **nil**
else append (v-to-bitv (cdr (v)),
list (**if** car (v) **then** `1`
else `0` **endif**)) **endif**

DEFINITION:
bool-to-v (b, *word-size*)
= **if** b = `'f` **then** nat-to-v (`0`, *word-size*)
else nat-to-v (`1`, *word-size*) **endif**

DEFINITION:
v-to-bool (v)
= **if** car (v) **then** `'t`
else `'f` **endif**

Definition:
addr-to-v $(adp,\ usr\text{-}data\text{-}links,\ word\text{-}size)$
$=$ nat-to-v (absolute-address $(adp,\ usr\text{-}data\text{-}links),\ word\text{-}size)$

Definition:
v-to-addr $(v,\ usr\text{-}data\text{-}links)$
$=$ invert-absolute-address (v-to-nat $(v),\ usr\text{-}data\text{-}links)$

Definition:
subr-to-v $(subr,\ prog\text{-}links,\ word\text{-}size)$
$=$ nat-to-v (base-address $(subr,\ prog\text{-}links),\ word\text{-}size)$

Definition:
v-to-subr $(v,\ prog\text{-}links)$ $=$ invert-base-address (v-to-nat $(v),\ prog\text{-}links)$

Definition:
sys-addr-to-v $(adp,\ sys\text{-}data\text{-}links,\ word\text{-}size)$
$=$ nat-to-v (absolute-address $(adp,\ sys\text{-}data\text{-}links),\ word\text{-}size)$

Definition:
v-to-sys-addr $(v,\ sys\text{-}data\text{-}links)$
$=$ invert-absolute-address (v-to-nat $(v),\ sys\text{-}data\text{-}links)$

Definition:
label-to-v $(ilab,\ prog\text{-}label\text{-}tables,\ word\text{-}size)$
$=$ nat-to-v (label-address $(ilab,\ prog\text{-}label\text{-}tables),\ word\text{-}size)$

Definition:
v-to-label $(v,\ prog\text{-}label\text{-}tables)$
$=$ invert-label-address (v-to-nat $(v),\ prog\text{-}label\text{-}tables)$

Definition:
ipc-to-v $(pcpp,\ prog\text{-}links,\ word\text{-}size)$
$=$ nat-to-v (absolute-address $(pcpp,\ prog\text{-}links),\ word\text{-}size)$

Definition:
link-data-word $(x,\ link\text{-}tables,\ word\text{-}size)$
$=$ **case on** type (x):
case $=$ *nat*
then nat-to-v (untag $(x),\ word\text{-}size)$
case $=$ *int*
then int-to-v (untag $(x),\ word\text{-}size)$
case $=$ *bitv*
then bitv-to-v (untag $(x),\ word\text{-}size)$
case $=$ *bool*
then bool-to-v (untag $(x),\ word\text{-}size)$

case = *addr*
 then addr-to-v (untag (x), usr-data-links (*link-tables*), *word-size*)
case = *subr*
 then subr-to-v (untag (x), prog-links (*link-tables*), *word-size*)
case = *sys-addr*
 then sys-addr-to-v (untag (x), sys-data-links (*link-tables*), *word-size*)
case = *pc*
 then label-to-v (untag (x), prog-label-tables (*link-tables*), *word-size*)
case = *ipc*
 then ipc-to-v (untag (x), prog-links (*link-tables*), *word-size*)
otherwise nat-to-v (`0`, *word-size*) **endcase**

Definition:
unlink-data-word (*type*, *v*, *link-tables*)
= **case on** *type*:
case = *nat*
then tag (`'nat`, v-to-nat (v))
case = *int*
 then tag (`'int`, v-to-int (v))
case = *bitv*
 then tag (`'bitv`, v-to-bitv (v))
case = *bool*
 then tag (`'bool`, v-to-bool (v))
case = *addr*
 then tag (`'addr`, v-to-addr (v, usr-data-links (*link-tables*)))
case = *subr*
 then tag (`'subr`, v-to-subr (v, prog-links (*link-tables*)))
case = *sys-addr*
 then tag (`'sys-addr`, v-to-sys-addr (v, sys-data-links (*link-tables*)))
case = *pc*
 then tag (`'pc`, v-to-label (v, prog-label-tables (*link-tables*)))
otherwise `'(unrecognized i-level type)` **endcase**

Definition:
link-instruction-alist
=
```
'((add_<c>_x_x{n} (add (c) x x))
  (add_<tsp>_<tsp>{v} (add nil (tsp) (tsp)))
  (add_<tsp>_<tsp>{n} (add nil (tsp) (tsp)))
  (add_<tsp>{a}_x{n} (add nil (tsp) x))
  (add_tsp_*{n} (add nil tsp (pc 1)))
  (add_tsp_x{n} (add nil tsp x))
  (add_<tsp>{i}_x{i} (add nil (tsp) x))
  (add_<tsp>{n}_x{n} (add nil (tsp) x))
  (add_pc_x{n} (add nil pc x))
```

```
(add_x_x{n} (add nil x x))
(add_x{n}_csp (add nil x csp))
(addc_<c>_x{n}_y{n} (addc (c) x y))
(addc_<v>_x{i}_y{i} (addc (v) x y))
(and_<tsp>{v}_x{v} (and nil (tsp) x))
(and_<tsp>{b}_x{b} (and nil (tsp) x))
(asr_<c>_<tsp>_<tsp>{b} (asr (c) (tsp) (tsp)))
(cpop_cfp (move nil cfp (csp 1)))
(cpop_pc (move nil pc (csp 1)))
(cpush_* (move nil (-1 csp) (pc 1)))
(cpush_<tsp>+ (move nil (-1 csp) (tsp 1)))
(cpush_cfp (move nil (-1 csp) cfp))
(decr_<tsp>_<tsp>{i} (decr nil (tsp) (tsp)))
(decr_<tsp>_<tsp>{n} (decr nil (tsp) (tsp)))
(incr_<tsp>_<tsp>{i} (incr nil (tsp) (tsp)))
(incr_<tsp>_<tsp>{n} (incr nil (tsp) (tsp)))
(incr_y_y{n} (incr nil y y))
(int-to-nat (move nil x x))
(jump-n_x (move-n nil pc x))
(jump-nn_x (move-nn nil pc x))
(jump-nz_x (move-nz nil pc x))
(jump-z_x (move-z nil pc x))
(jump_* (move nil pc (pc)))
(jump_x{subr} (move nil pc x))
(lsr_<c>_x_x{n} (lsr (c) x x))
(lsr_<tsp>_<tsp>{v} (lsr nil (tsp) (tsp)))
(move-c_<tsp>_* (move-c nil (tsp) (pc 1)))
(move-v_<tsp>_* (move-v nil (tsp) (pc 1)))
(move-z_<tsp>_* (move-z nil (tsp) (pc 1)))
(move-n_x_* (move-n nil x (pc 1)))
(move_<tsp>_* (move nil (tsp) (pc 1)))
(move_<x{a}>_<tsp> (move nil (x) (tsp)))
(move_<x{s}>_<tsp> (move nil (x) (tsp)))
(move_cfp_csp (move nil cfp csp))
(move_csp_cfp (move nil csp cfp))
(move_x_* (move nil x (pc 1)))
(move_x_<x{s}> (move nil x (x)))
(move_x_tsp (move nil x tsp))
(move_x_x (move nil x x))
(move_y_* (move nil y (pc 1)))
(move_y_<y{s}> (move nil y (y)))
(move_y_tsp (move nil y tsp))
(neg_<tsp>_<tsp>{i} (neg nil (tsp) (tsp)))
(not_<tsp>_<tsp>{v} (not nil (tsp) (tsp)))
```

```
(or_<tsp>{v}_x{v} (or nil (tsp) x))
(or_<tsp>{b}_x{b} (or nil (tsp) x))
(sub_<c>_<tsp>{a}_x{a} (sub (c) (tsp) x))
(sub_<c>_<tsp>{n}_x{n} (sub (c) (tsp) x))
(sub_<nv>_<tsp>{i}_x{i} (sub (n v) (tsp) x))
(sub_<tsp>{a}_x{n} (sub nil (tsp) x))
(sub_x{s}_y{n} (sub nil x y))
(sub_<tsp>{i}_x{i} (sub nil (tsp) x))
(sub_<tsp>{n}_x{n} (sub nil (tsp) x))
(sub_<tsp>{s}_x{s} (sub nil (tsp) x))
(sub_<z>_x{s}_y{s} (sub (z) x y))
(subb_<c>_x{n}_y{n} (subb (c) x y))
(subb_<v>_x{i}_y{i} (subb (v) x y))
(tpop_<c>_x (move (c) x (tsp 1)))
(tpop_<x{a}> (move nil (x) (tsp 1)))
(tpop_<x{s}> (move nil (x) (tsp 1)))
(tpop_pc (move nil pc (tsp 1)))
(tpop_x (move nil x (tsp 1)))
(tpop_y (move nil y (tsp 1)))
(tpop{v}_<z>_y (move (z) y (tsp 1)))
(tpop{b}_<z>_y (move (z) y (tsp 1)))
(tpop{i}_<zn>_y (move (z n) y (tsp 1)))
(tpop{n}_<z>_y (move (z) y (tsp 1)))
(tpush_* (move nil (-1 tsp) (pc 1)))
(tpush_<x{a}> (move nil (-1 tsp) (x)))
(tpush_<x{s}> (move nil (-1 tsp) (x)))
(tpush_csp (move nil (-1 tsp) csp))
(tpush_tsp (move nil (-1 tsp) tsp))
(tpush_x (move nil (-1 tsp) x))
(xor_<tsp>_<tsp> (xor nil (tsp) (tsp)))
(xor_<tsp>{v}_x{v} (xor nil (tsp) x))
(xor_<tsp>{b}_*{b} (xor nil (tsp) (pc 1)))
(xor_<tsp>{b}_x{b} (xor nil (tsp) x))
(xor_<z>_<tsp>_x (xor (z) (tsp) x)))
```

DEFINITION:

$$\begin{aligned}
&\text{pack-instruction}\,(\textit{op},\ \textit{move-bits},\ \textit{cvnz},\ \textit{mode-b},\ \textit{reg-b},\ \textit{mode-a},\ \textit{reg-a},\ \textit{word-size})\\
= \quad &\text{nat-to-v}\,((\textit{op} * \exp(2, 24))\\
&\qquad\qquad + \quad (\textit{move-bits} * \exp(2, 20))\\
&\qquad\qquad + \quad (\textit{cvnz} * \exp(2, 16))\\
&\qquad\qquad + \quad (\textit{mode-b} * \exp(2, 14))\\
&\qquad\qquad + \quad (\textit{reg-b} * \exp(2, 10))\\
&\qquad\qquad + \quad (\textit{mode-a} * \exp(2, 4))\\
&\qquad\qquad + \quad \textit{reg-a},
\end{aligned}$$

word-size)

DEFINITION:
extract-op (*opcode*)
= cadr (assoc (*opcode*,
```
'((incr 1)
  (addc 2)
  (add 3)
  (neg 4)
  (decr 5)
  (subb 6)
  (sub 7)
  (ror 8)
  (asr 9)
  (lsr 10)
  (xor 11)
  (or 12)
  (and 13)
  (not 14)
  (move 15)
  (move-nc 15)
  (move-c 15)
  (move-nv 15)
  (move-v 15)
  (move-nz 15)
  (move-z 15)
  (move-nn 15)
  (move-n 15))))
```

DEFINITION:
extract-move-bits (*opcode*)
= **case on** *opcode*:
case = *move*
then 14
case = *move-nc*
then 0
case = *move-c*
then 1
case = *move-nv*
then 2
case = *move-v*
then 3
case = *move-nz*
then 6

```
     case = move-z
      then 7
     case = move-nn
      then 4
     case = move-n
      then 5
     otherwise 14 endcase
```

DEFINITION:
extract-mode (*reg-spec*)

```
=    if litatom (reg-spec)  then 0
     elseif cdr (reg-spec) = nil  then 1
     elseif car (reg-spec) = -1  then 2
     else 3 endif
```

DEFINITION:
extract-cvnz (*flg-names*)

```
=    ((if 'c ∈ flg-names  then 1
       else 0 endif
        *   exp (2, 3))
       +   (if 'v ∈ flg-names  then 1
             else 0 endif
             *   exp (2, 2))
       +   (if 'n ∈ flg-names  then 1
             else 0 endif
             *   exp (2, 1))
       +   (if 'z ∈ flg-names  then 1
             else 0 endif
             *   exp (2, 0)))
```

DEFINITION:
extract-reg1 (*reg-spec*)

```
=    if litatom (reg-spec)  then reg-spec
     elseif cdr (reg-spec) = nil  then car (reg-spec)
     elseif car (reg-spec) = -1  then cadr (reg-spec)
     else car (reg-spec) endif
```

DEFINITION:
extract-reg (*reg-spec*)

```
=    cadr (assoc (extract-reg1 (reg-spec),
                  '((pc 15)
                    (cfp 1)
                    (csp 2)
                    (tsp 3)
                    (x 4)
                    (y 5))))
```

DEFINITION:
mci (*ins*, *word-size*)
= pack-instruction (extract-op (car (*ins*)),
extract-move-bits (car (*ins*)),
extract-cvnz (cadr (*ins*)),
extract-mode (caddr (*ins*)),
extract-reg (caddr (*ins*)),
extract-mode (cadddr (*ins*)),
extract-reg (cadddr (*ins*)),
word-size)

DEFINITION: icode-instructionp (*ins*) = (cdr (*ins*) = **nil**)

DEFINITION:
link-instr-word (*ins*, *word-size*)
= mci (cadr (assoc (car (*ins*), LINK-INSTRUCTION-ALIST)), *word-size*)

DEFINITION:
link-word (x, *link-tables*, *word-size*)
= **if** icode-instructionp (x) **then** link-instr-word (x, *word-size*)
else link-data-word (x, *link-tables*, *word-size*) **endif**

DEFINITION:
link-area (*lst*, *link-tables*, *word-size*)
= **if** *lst* $\simeq$ **nil** **then nil**
else cons (link-word (unlabel (car (*lst*)), *link-tables*, *word-size*),
link-area (cdr (*lst*), *link-tables*, *word-size*)) **endif**

DEFINITION:
link-segment (*segment*, *link-tables*, *word-size*)
= **if** *segment* $\simeq$ **nil** **then nil**
else append (link-area (cdar (*segment*), *link-tables*, *word-size*),
link-segment (cdr (*segment*), *link-tables*, *word-size*)) **endif**

DEFINITION:
boot-code (*lst*, n, *word-size*)
= **if** $n \simeq 0$ **then nil**
else cons (nat-to-v (car (*lst*), *word-size*),
boot-code (cdr (*lst*), $n - 1$, *word-size*)) **endif**

DEFINITION:
link-mem (*boot-lst*,
load-addr,
usr-data-segment,
prog-segment,

sys-data-segment,
link-tables,
word-size)
= append (boot-code (*boot-lst*, *load-addr*, *word-size*),
append (link-segment (*usr-data-segment*, *link-tables*, *word-size*),
append (link-segment (*prog-segment*, *link-tables*, *word-size*),
link-segment (*sys-data-segment*,
link-tables,
word-size))))

DEFINITION:
bool-to-logical (b)
= **if** b = `'f` **then f**
else t endif

DEFINITION:
i->m (i, *boot-lst*, *load-addr*)
= **let** *tables* **be** i-link-tables (i, *load-addr*),
w **be** i-word-size (i)
in
m-state (list (link-word (`'(nat 0)`, *tables*, w),
link-word (i-cfp (i), *tables*, w),
link-word (i-csp (i), *tables*, w),
link-word (i-tsp (i), *tables*, w),
link-word (i-x (i), *tables*, w),
link-word (i-y (i), *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (`'(nat 0)`, *tables*, w),
link-word (i-pc (i), *tables*, w)),
bool-to-logical (untag (i-c-flg (i))),
bool-to-logical (untag (i-v-flg (i))),
bool-to-logical (untag (i-n-flg (i))),
bool-to-logical (untag (i-z-flg (i))),
link-mem (*boot-lst*,
load-addr,
i-usr-data-segment (i),
i-prog-segment (i),

i-sys-data-segment (i),
tables,
w)) **endlet**

THEOREM: my-lessp-quotient
$((size \in \mathbf{N}) \land (size \neq 0)) \rightarrow ((size \div 2) < size)$

DEFINITION:
ram-tree (*lst*, *size*)
= **if** *size* $\simeq$ 0 **then** stub (nat-to-v (0, 32))
elseif *lst* $\simeq$ **nil** **then** stub (nat-to-v (0, 32))
elseif *size* = 1 **then** ram (car (*lst*))
else cons (ram-tree (firstn (*size* $\div$ 2, *lst*), *size* $\div$ 2),
ram-tree (restn (*size* $\div$ 2, *lst*), *size* $\div$ 2)) **endif**

DEFINITION:
fm-state (*regs*, c, v, n, z, *mem*)
= list (list (ram-tree (*regs*, 16), list (z, n, v, c)), ram-tree (*mem*, exp (2, 32)))

DEFINITION:
m->fm9001 (m)
= list (list (ram-tree (m-regs (m), 16),
list (m-z-flg (m), m-n-flg (m), m-v-flg (m), m-c-flg (m))),
ram-tree (m-mem (m), exp (2, 32)))

DEFINITION:
load (p, *boot-lst*, *load-addr*)
= m->fm9001 (i->m (r->i (p->r (p)), *boot-lst*, *load-addr*))

DEFINITION:
link-tables (p, *load-addr*) = i-link-tables (r->i (p->r (p)), *load-addr*)

DEFINITION:
type-lst (*lst*)
= **if** *lst* $\simeq$ **nil** **then nil**
else cons (type (car (*lst*)), type-lst (cdr (*lst*))) **endif**

DEFINITION:
area-type-specification (*area*) = cons (car (*area*), type-lst (cdr (*area*)))

DEFINITION:
type-specification (*segment*)
= **if** *segment* $\simeq$ **nil** **then nil**
else cons (area-type-specification (car (*segment*)),
type-specification (cdr (*segment*))) **endif**

Definition:

```
display-fm9001-array (type-lst, n, fm-mem, link-tables)
=  if type-lst ≃ nil then nil
   else cons (unlink-data-word (car (type-lst),
                                read-mem (nat-to-v (n, 32), fm-mem),
                                link-tables),
              display-fm9001-array (cdr (type-lst),
                                    1 + n,
                                    fm-mem,
                                    link-tables)) endif
```

Definition:

```
display-fm9001-data-area (area-type-spec, fm-mem, link-tables)
=  cons (car (area-type-spec),
         display-fm9001-array (cdr (area-type-spec),
                               base-address (car (area-type-spec),
                                             usr-data-links (link-tables)),
                               fm-mem,
                               link-tables))
```

Definition:

```
display-fm9001-data-segment1 (type-spec, fm-mem, link-tables)
=  if type-spec ≃ nil then nil
   else cons (display-fm9001-data-area (car (type-spec),
                                        fm-mem,
                                        link-tables),
              display-fm9001-data-segment1 (cdr (type-spec),
                                            fm-mem,
                                            link-tables)) endif
```

Definition:

```
display-fm9001-data-segment (fm-state, type-spec, link-tables)
=  display-fm9001-data-segment1 (type-spec, cadr (fm-state), link-tables)
```

```
; Book:  m.events.  Now we define the m machine.
```

Definition:

```
v-nth1 (v-n, lst)
=  if v-to-nat (v-n) < length (lst) then nth (v-to-nat (v-n), lst)
   else nat-to-v (0, 32) endif
```

Definition:

```
current-instruction (regs, mem) = v-nth1 (v-nth1 (nat-to-v (15, 4), regs), mem)
```

Definition:
m-store-resultp$\,(\mathit{store\text{-}cc},\ c,\ v,\ n,\ z)$
$=$ **case on** *store-cc*:
case $=$ *1*false$\,(\mathit{*1*false},\ \mathit{*1*false},\ \mathit{*1*false})$
then $\neg\ c$
case $=$ *1*true$\,(\mathit{*1*false},\ \mathit{*1*false},\ \mathit{*1*false})$
then c
case $=$ *1*false$\,(\mathit{*1*true},\ \mathit{*1*false},\ \mathit{*1*false})$
then $\neg\ v$
case $=$ *1*true$\,(\mathit{*1*true},\ \mathit{*1*false},\ \mathit{*1*false})$
then v
case $=$ *1*false$\,(\mathit{*1*false},\ \mathit{*1*true},\ \mathit{*1*false})$
then $\neg\ n$
case $=$ *1*true$\,(\mathit{*1*false},\ \mathit{*1*true},\ \mathit{*1*false})$
then n
case $=$ *1*false$\,(\mathit{*1*true},\ \mathit{*1*true},\ \mathit{*1*false})$
then $\neg\ z$
case $=$ *1*true$\,(\mathit{*1*true},\ \mathit{*1*true},\ \mathit{*1*false})$
then z
case $=$ *1*false$\,(\mathit{*1*false},\ \mathit{*1*false},\ \mathit{*1*true})$
then $(\neg\ c) \wedge (\neg\ z)$
case $=$ *1*true$\,(\mathit{*1*false},\ \mathit{*1*false},\ \mathit{*1*true})$
then $c \vee z$
case $=$ *1*false$\,(\mathit{*1*true},\ \mathit{*1*false},\ \mathit{*1*true})$
then $(n \wedge v) \vee ((\neg\ n) \wedge (\neg\ v))$
case $=$ *1*true$\,(\mathit{*1*true},\ \mathit{*1*false},\ \mathit{*1*true})$
then $(n \wedge (\neg\ v)) \vee ((\neg\ n) \wedge v)$
case $=$ *1*false$\,(\mathit{*1*false},\ \mathit{*1*true},\ \mathit{*1*true})$
then $(n \wedge v \wedge (\neg\ z)) \vee ((\neg\ n) \wedge (\neg\ v) \wedge (\neg\ z))$
case $=$ *1*true$\,(\mathit{*1*false},\ \mathit{*1*true},\ \mathit{*1*true})$
then $z \vee (n \wedge (\neg\ v)) \vee ((\neg\ n) \wedge v)$
case $=$ *1*false$\,(\mathit{*1*true},\ \mathit{*1*true},\ \mathit{*1*true})$
then t
otherwise f endcase

Theorem: m-store-resultp-is-store-resultp
m-store-resultp$\,(\mathit{store\text{-}cc},\ c,\ v,\ n,\ z)$ = store-resultp$\,(\mathit{store\text{-}cc},\ \mathrm{list}\,(z,\ n,\ v,\ c))$

Event: Disable m-store-resultp.

Event: Disable store-resultp.

Definition:

m-alu-operation (*regs*, *c*, *v*, *n*, *z*, *mem*, *ins*, *operand-a*, *operand-b*, *b-address*)
= m-state (**if** m-store-resultp (store-cc (*ins*), *c*, *v*, *n*, *z*)
    ∧ reg-direct-p (mode-b (*ins*))
  then update-v-nth (rn-b (*ins*),
      regs,
      bv (v-alu (*c*, *operand-a*, *operand-b*, op-code (*ins*))))
  else *regs* **endif**,
  b-if (c-set (set-flags (*ins*)),
    c (v-alu (*c*, *operand-a*, *operand-b*, op-code (*ins*))),
    c),
  b-if (v-set (set-flags (*ins*)),
    v (v-alu (*c*, *operand-a*, *operand-b*, op-code (*ins*))),
    v),
  b-if (n-set (set-flags (*ins*)),
    n (v-alu (*c*, *operand-a*, *operand-b*, op-code (*ins*))),
    n),
  b-if (z-set (set-flags (*ins*)),
    zb (v-alu (*c*, *operand-a*, *operand-b*, op-code (*ins*))),
    z),
  if m-store-resultp (store-cc (*ins*), *c*, *v*, *n*, *z*)
    ∧ (¬ reg-direct-p (mode-b (*ins*)))
  then update-v-nth (*b-address*,
      mem,
      bv (v-alu (*c*, *operand-a*, *operand-b*, op-code (*ins*))))
  else *mem* **endif**)

DEFINITION:
read-mem1-ram-tree-hint (*raddr*, *lst*)
= **if** *raddr* $\simeq$ **nil** **then** **t**
  elseif car (*raddr*)
  then read-mem1-ram-tree-hint (cdr (*raddr*),
      restn (exp (2, length (cdr (*raddr*))), *lst*))
  else read-mem1-ram-tree-hint (cdr (*raddr*),
      firstn (exp (2, length (cdr (*raddr*))),
        lst)) **endif**

DEFINITION:
rev (*x*)
= **if** *x* $\simeq$ **nil** **then** **nil**
  else append (rev (cdr (*x*)), list (car (*x*))) **endif**

EVENT: Enable append.

THEOREM: rev1-is-rev
rev1 (*x*, *a*) = append (rev (*x*), *a*)

THEOREM: append-nil
properp $(x) \rightarrow (\text{append}(x, \textbf{nil}) = x)$

THEOREM: properp-rev
properp $(\text{rev}(x))$

THEOREM: read-mem1-rev-ram-tree-lemma1
stubp $(\text{ram-tree}(lst, n)) \rightarrow (\text{stub-guts}(\text{ram-tree}(lst, n)) = \text{nat-to-v}(0, 32))$

EVENT: Enable plus-0.

EVENT: Enable plus-add1-sub1.

EVENT: Enable plus-add1.

EVENT: Enable associativity-of-plus.

EVENT: Enable commutativity-of-plus.

EVENT: Enable times-commutes.

EVENT: Enable times-add1-again.

EVENT: Enable times-1.

EVENT: Enable times-distributes-over-plus.

EVENT: Enable associativity-of-times.

THEOREM: my-v-to-nat-append
v-to-nat $(\text{append}(a, b))$
$= \quad (\text{v-to-nat}(a) + (\exp(2, \text{length}(a)) * \text{v-to-nat}(b)))$

EVENT: Enable plus.

EVENT: Enable difference.

THEOREM: difference-add1-add1-x-2
$((1 + (1 + x)) - 2) = \text{fix}(x)$

EVENT: Enable quotient-plus-x-x-2.

EVENT: Enable length-firstn.

EVENT: Enable length-restn.

THEOREM: my-lessp-v-to-nat-exp
$(n = \text{length}(v)) \rightarrow (\text{v-to-nat}(v) < \exp(2,\ n))$

THEOREM: length-rev
$\text{length}(\text{rev}(x)) = \text{length}(x)$

THEOREM: nth-firstn
$(i < n) \rightarrow (\text{nth}(i,\ \text{firstn}(n,\ lst)) = \text{nth}(i,\ lst))$

THEOREM: lessp-difference
$(v < (l - e))$
$= \quad \textbf{if } e < l \textbf{ then } (e + v) < l$
$\qquad \textbf{else f endif}$

THEOREM: nth-restn-plus
$(i < \text{length}(lst)) \rightarrow (\text{nth}(n,\ \text{restn}(i,\ lst)) = \text{nth}(i + n,\ lst))$

THEOREM: read-mem1-rev-ram-tree
$(\exp(2, \text{length}(raddr)) \not< \text{length}(lst))$
$\rightarrow \quad (\text{v-nth1}(\text{rev}(raddr),\ lst)$
$\qquad = \quad \text{read-mem1}(raddr,\ \text{ram-tree}(lst,\ \exp(2,\ \text{length}(raddr)))))$

THEOREM: rev-append
$\text{rev}(\text{append}(a,\ b)) = \text{append}(\text{rev}(b),\ \text{rev}(a))$

THEOREM: v-to-nat-rev-rev
$\text{v-to-nat}(\text{rev}(\text{rev}(addr))) = \text{v-to-nat}(addr)$

THEOREM: read-mem-ram-tree
$((e = \exp(2, \text{length}(addr))) \wedge (e \not< \text{length}(lst)))$
$\rightarrow \quad (\text{v-nth1}(addr,\ lst) = \text{read-mem}(addr,\ \text{ram-tree}(lst,\ e)))$

EVENT: Disable v-nth1.

EVENT: Disable read-mem.

THEOREM: restn-update-nth-plus
$\text{restn}(e,\ \text{update-nth}(v + e,\ l,\ val)) = \text{update-nth}(v,\ \text{restn}(e,\ l),\ val)$

THEOREM: firstn-update-nth-plus
$\text{firstn}\,(e,\ \text{update-nth}\,(v + e,\ l,\ \mathit{val})) = \text{firstn}\,(e,\ l)$

THEOREM: restn-update-nth
$(v < e) \rightarrow (\text{restn}\,(e,\ \text{update-nth}\,(v,\ l,\ \mathit{val})) = \text{restn}\,(e,\ l))$

THEOREM: firstn-update-nth
$(v < e)$
$\rightarrow\quad (\text{firstn}\,(e,\ \text{update-nth}\,(v,\ l,\ \mathit{val})) = \text{update-nth}\,(v,\ \text{firstn}\,(e,\ l),\ \mathit{val}))$

THEOREM: write-mem1-rev-ram-tree
$(\exp\,(2,\ \text{length}\,(\mathit{raddr})) \not< \text{length}\,(\mathit{lst}))$
$\rightarrow\quad (\text{ram-tree}\,(\text{update-v-nth}\,(\text{rev}\,(\mathit{raddr}),\ \mathit{lst},\ \mathit{val}),\ \exp\,(2,\ \text{length}\,(\mathit{raddr})))$
$\qquad = \quad \text{write-mem1}\,(\mathit{raddr},\ \text{ram-tree}\,(\mathit{lst},\ \exp\,(2,\ \text{length}\,(\mathit{raddr}))),\ \mathit{val}))$

THEOREM: write-mem-ram-tree
$((e = \exp\,(2,\ \text{length}\,(\mathit{addr}))) \wedge (e \not< \text{length}\,(\mathit{lst})))$
$\rightarrow\quad (\text{ram-tree}\,(\text{update-v-nth}\,(\mathit{addr},\ \mathit{lst},\ \mathit{val}),\ e)$
$\qquad = \quad \text{write-mem}\,(\mathit{addr},\ \text{ram-tree}\,(\mathit{lst},\ e),\ \mathit{val}))$

EVENT: Disable update-v-nth.

EVENT: Disable write-mem.

THEOREM: length-rn-b
$\text{length}\,(\text{rn-b}\,(\mathit{ins})) = 4$

THEOREM: m-alu-operation-is-fm9001-alu-operation
$((\text{length}\,(\mathit{b\text{-}address}) = 32)$
$\wedge\quad (\text{length}\,(\mathit{regs}) = 16)$
$\wedge\quad (\exp\,(2,\ 32) \not< \text{length}\,(\mathit{mem})))$
$\rightarrow\quad (\text{m->fm9001}\,(\text{m-alu-operation}\,(\mathit{regs},$
$c,$
$v,$
$n,$
$z,$
$\mathit{mem},$
$\mathit{ins},$
$\mathit{operand\text{-}a},$
$\mathit{operand\text{-}b},$
$\mathit{b\text{-}address}))$
$\qquad = \quad \text{fm9001-alu-operation}\,(\text{ram-tree}\,(\mathit{regs},\ 16),$
$\text{list}\,(z,\ n,\ v,\ c),$
$\text{ram-tree}\,(\mathit{mem},\ \exp\,(2,\ 32)),$

$\qquad\qquad\qquad\qquad\qquad\qquad$ *ins*,
$\qquad\qquad\qquad\qquad\qquad\qquad$ *operand-a*,
$\qquad\qquad\qquad\qquad\qquad\qquad$ *operand-b*,
$\qquad\qquad\qquad\qquad\qquad\qquad$ *b-address*))

DEFINITION:
bvp-listp (*lst*)
= **if** *lst* $\simeq$ **nil** **then** *lst* = **nil**
  else bvp (car (*lst*))
    $\wedge$ (length (car (*lst*)) = 32)
    $\wedge$ bvp-listp (cdr (*lst*)) **endif**

DEFINITION:
proper-m-statep (*m*)
= (m-statep (*m*)
  $\wedge$ bvp-listp (m-regs (*m*))
  $\wedge$ bvp-listp (m-mem (*m*))
  $\wedge$ (length (m-regs (*m*)) = 16)
  $\wedge$ (exp (2, 32) $\not<$ length (m-mem (*m*))))

THEOREM: length-nth
(bvp-listp (*lst*) $\wedge$ (*n* < length (*lst*)))
$\rightarrow$ (bvp (nth (*n*, *lst*)) $\wedge$ (length (nth (*n*, *lst*)) = 32))

THEOREM: length-v-nth1
bvp-listp (*lst*)
$\rightarrow$ (bvp (v-nth1 (*addr*, *lst*)) $\wedge$ (length (v-nth1 (*addr*, *lst*)) = 32))

THEOREM: length-read-mem
(bvp-listp (*lst*) $\wedge$ (*k* = exp (2, length (*addr*))) $\wedge$ (*k* $\not<$ length (*lst*)))
$\rightarrow$ (bvp (read-mem (*addr*, ram-tree (*lst*, *k*)))
  $\wedge$ (length (read-mem (*addr*, ram-tree (*lst*, *k*))) = 32))

THEOREM: bvp-listp-update-nth
(bvp (*val*) $\wedge$ (length (*val*) = 32) $\wedge$ bvp-listp (*lst*))
$\rightarrow$ bvp-listp (update-nth (*n*, *lst*, *val*))

THEOREM: bvp-listp-update-v-nth
(bvp (*val*) $\wedge$ (length (*val*) = 32) $\wedge$ bvp-listp (*lst*))
$\rightarrow$ bvp-listp (update-v-nth (*addr*, *lst*, *val*))

EVENT: Enable bvp-bv-v-alu.

THEOREM: length-bv-v-alu
((length (*a*) = length (*b*)) $\wedge$ listp (*a*))
$\rightarrow$ (length (bv (v-alu (*c*, *a*, *b*, *op*))) = length (*a*))

THEOREM: length-update-nth
length (update-nth (i, lst, val)) = length (lst)

THEOREM: length-update-v-nth
length (update-v-nth (v, lst, val)) = length (lst)

THEOREM: m-alu-operation-preserves-hyps
(bvp-listp (mem)
 $\land$ bvp-listp ($regs$)
 $\land$ bvp ($operand\text{-}a$)
 $\land$ (length ($operand\text{-}a$) = 32)
 $\land$ (length ($operand\text{-}b$) = 32)
 $\land$ listp ($operand\text{-}a$)
 $\land$ (length ($b\text{-}address$) = 32)
 $\land$ (length ($regs$) = 16)
 $\land$ (exp (2, 32) $\not<$ length (mem)))
$\rightarrow$ proper-m-statep (m-alu-operation ($regs$,
 c,
 v,
 n,
 z,
 mem,
 ins,
 $operand\text{-}a$,
 $operand\text{-}b$,
 $b\text{-}address$))

EVENT: Disable m-alu-operation.

EVENT: Disable fm9001-alu-operation.

DEFINITION:
m-operand-b ($regs$, c, v, n, z, mem, ins, $operand\text{-}a$)
= m-alu-operation (**if** pre-dec-p (mode-b (ins))
 then update-v-nth (rn-b (ins),
 $regs$,
 v-dec (v-nth1 (rn-b (ins), $regs$)))
 elseif post-inc-p (mode-b (ins))
 then update-v-nth (rn-b (ins),
 $regs$,
 v-inc (v-nth1 (rn-b (ins), $regs$)))
 else $regs$ **endif**,
 c,
 v,

n,
z,
mem,
ins,
$operand\text{-}a$,
if reg-direct-p (mode-b (ins))
then v-nth1 (rn-b (ins), $regs$)
else v-nth1 (**if** pre-dec-p (mode-b (ins))
then v-dec (v-nth1 (rn-b (ins), $regs$))
else v-nth1 (rn-b (ins), $regs$) **endif**,
mem) **endif**,
if pre-dec-p (mode-b (ins))
then v-dec (v-nth1 (rn-b (ins), $regs$))
else v-nth1 (rn-b (ins), $regs$) **endif**)

EVENT: Enable length-v-not.

THEOREM: length-v-adder
length (v-adder (c, a, b)) = (1 + length (a))

EVENT: Enable length-nat-to-v.

THEOREM: length-v-dec
length (v-dec (v)) = length (v)

THEOREM: m-operand-b-is-fm9001-operand-b
(bvp-listp ($regs$) $\wedge$ (length ($regs$) = 16) $\wedge$ (exp (2, 32) $\not<$ length (mem)))
$\rightarrow$ (m->fm9001 (m-operand-b ($regs$, c, v, n, z, mem, ins, $operand\text{-}a$))
= fm9001-operand-b (ram-tree ($regs$, 16),
list (z, n, v, c),
ram-tree (mem, exp (2, 32)),
ins,
$operand\text{-}a$))

THEOREM: length-v-inc
length (v-inc (v)) = length (v)

THEOREM: bvp-firstn-v-adder
(n = length (a)) $\rightarrow$ bvp (firstn (n, v-adder (c, a, b)))

THEOREM: bvp-v-dec
bvp (v-dec (v))

THEOREM: bvp-v-inc
bvp (v-inc (v))

THEOREM: bvp-listp-if
(bvp-listp (b) $\land$ bvp-listp (c))
$\rightarrow$ bvp-listp (**if** a **then** b
else c **endif**)

EVENT: Enable length-if.

THEOREM: listp-operand-a
(length (*operand-a*) $=$ 32) $\rightarrow$ listp (*operand-a*)

THEOREM: m-operand-b-preserves-hyps
(bvp-listp (*regs*)
$\land$ bvp-listp (*mem*)
$\land$ bvp (*operand-a*)
$\land$ (length (*operand-a*) $=$ 32)
$\land$ (length (*regs*) $=$ 16)
$\land$ (exp (2, 32) $\not<$ length (*mem*)))
$\rightarrow$ proper-m-statep (m-operand-b (*regs*, c, v, n, z, *mem*, *ins*, *operand-a*))

EVENT: Disable m-operand-b.

EVENT: Disable fm9001-operand-b.

DEFINITION:
m-operand-a (*regs*, c, v, n, z, *mem*, *ins*)
$=$ m-operand-b (**if** a-immediate-p (*ins*) **then** *regs*
elseif pre-dec-p (mode-a (*ins*))
then update-v-nth (rn-a (*ins*),
regs,
v-dec (v-nth1 (rn-a (*ins*), *regs*)))
elseif post-inc-p (mode-a (*ins*))
then update-v-nth (rn-a (*ins*),
regs,
v-inc (v-nth1 (rn-a (*ins*), *regs*)))
else *regs* **endif**,
c,
v,
n,
z,
mem,
ins,
if a-immediate-p (*ins*)
then sign-extend (a-immediate (*ins*), 32)

elseif reg-direct-p (mode-a (*ins*))
then v-nth1 (rn-a (*ins*), *regs*)
elseif pre-dec-p (mode-a (*ins*))
then v-nth1 (v-dec (v-nth1 (rn-a (*ins*), *regs*)), *mem*)
else v-nth1 (v-nth1 (rn-a (*ins*), *regs*), *mem*) **endif**)

THEOREM: length-rn-a
length (rn-a (*ins*)) = 4

THEOREM: m-operand-a-is-fm9001-operand-a
(bvp-listp (*regs*) ∧ (length (*regs*) = 16) ∧ (exp (2, 32) $\not<$ length (*mem*)))
→ (m->fm9001 (m-operand-a (*regs*, *c*, *v*, *n*, *z*, *mem*, *ins*))
= fm9001-operand-a (ram-tree (*regs*, 16),
list (*z*, *n*, *v*, *c*),
ram-tree (*mem*, exp (2, 32)),
ins))

THEOREM: m-operand-a-preserves-hyps
(bvp-listp (*regs*)
∧ bvp-listp (*mem*)
∧ (length (*regs*) = 16)
∧ (exp (2, 32) $\not<$ length (*mem*)))
→ proper-m-statep (m-operand-a (*regs*, *c*, *v*, *n*, *z*, *mem*, *ins*))

EVENT: Disable m-operand-a.

EVENT: Disable fm9001-operand-a.

DEFINITION:
m-step1 (*current-instruction*, *m*)
= m-operand-a (update-v-nth (nat-to-v (15, 4),
m-regs (*m*),
v-inc (v-nth1 (nat-to-v (15, 4), m-regs (*m*)))),
m-c-flg (*m*),
m-v-flg (*m*),
m-n-flg (*m*),
m-z-flg (*m*),
m-mem (*m*),
current-instruction)

DEFINITION:
m-step (*m*) = m-step1 (current-instruction (m-regs (*m*), m-mem (*m*)), *m*)

THEOREM: m-step-is-fm9001-step

$(\text{bvp-listp}\,(\text{m-regs}\,(m))$
$\ \wedge\ \ (\text{length}\,(\text{m-regs}\,(m)) = \texttt{16})$
$\ \wedge\ \ (\text{exp}\,(\texttt{2},\ \texttt{32}) \not< \text{length}\,(\text{m-mem}\,(m))))$
$\to\ \ (\text{m->fm9001}\,(\text{m-step}\,(m)) = \text{fm9001-step}\,(\text{m->fm9001}\,(m),\ \text{nat-to-v}\,(\texttt{15},\ \texttt{4})))$

THEOREM: m-step-preserves-hyps
$(\text{bvp-listp}\,(\text{m-regs}\,(m))$
$\ \wedge\ \ \text{bvp-listp}\,(\text{m-mem}\,(m))$
$\ \wedge\ \ (\text{length}\,(\text{m-regs}\,(m)) = \texttt{16})$
$\ \wedge\ \ (\text{exp}\,(\texttt{2},\ \texttt{32}) \not< \text{length}\,(\text{m-mem}\,(m))))$
$\to\ \ \text{proper-m-statep}\,(\text{m-step}\,(m))$

EVENT: Disable m-step.

EVENT: Disable m->fm9001.

EVENT: Disable fm9001-step.

DEFINITION:
$\text{m}\,(m,\ n)$
$=\ \ \textbf{if}\ n \simeq \texttt{0}\ \textbf{then}\ m$
$\quad\ \textbf{else}\ \text{m}\,(\text{m-step}\,(m),\ n - 1)\ \textbf{endif}$

THEOREM: one-way-correspondence-m-fm9001
$\text{proper-m-statep}\,(m) \to (\text{m->fm9001}\,(\text{m}\,(m,\ n)) = \text{fm9001}\,(\text{m->fm9001}\,(m),\ n))$

```
; Book:  i.events.  Now we move up to the i machine.
```

EVENT: Set the status of all events. The status of each event is to be set as follows. Each 'boot-strap' satellite is to be returned to the initial condition. Each 'add-shell' satellite is to be enabled. enabled. Anything not otherwise mentioned is to be disabled. Name this event 'close-data-base-2'.

DEFINITION:
$\text{put-assoc}\,(val,\ name,\ alist)$
$=\ \ \textbf{if}\ alist \simeq \textbf{nil}\ \textbf{then}\ alist$
$\quad\ \textbf{elseif}\ name = \text{caar}\,(alist)\ \textbf{then}\ \text{cons}\,(\text{cons}\,(name,\ val),\ \text{cdr}\,(alist))$
$\quad\ \textbf{else}\ \text{cons}\,(\text{car}\,(alist),\ \text{put-assoc}\,(val,\ name,\ \text{cdr}\,(alist)))\ \textbf{endif}$

DEFINITION:
$\text{definedp}\,(name,\ alist)$
$=\ \ \textbf{if}\ alist \simeq \textbf{nil}\ \textbf{then}\ \textbf{f}$
$\quad\ \textbf{elseif}\ name = \text{caar}\,(alist)\ \textbf{then}\ \textbf{t}$
$\quad\ \textbf{else}\ \text{definedp}\,(name,\ \text{cdr}\,(alist))\ \textbf{endif}$

```
; DEFINIENS, below, used to be named VALUE but that name is now used
; in FM9001.
```

DEFINITION: definiens $(name,\ alist) = \text{cdr}(\text{definition}(name,\ alist))$

DEFINITION:
put-value $(val,\ name,\ alist) = \text{put-assoc}(val,\ name,\ alist)$

DEFINITION:
adpp $(x,\ segment)$

$$
\begin{aligned}
= \quad & (\text{listp}(x) \\
& \quad \land \quad (\text{adp-offset}(x) \in \mathbf{N}) \\
& \quad \land \quad \text{definedp}(\text{adp-name}(x),\ segment) \\
& \quad \land \quad (\text{adp-offset}(x) < \text{length}(\text{definiens}(\text{adp-name}(x),\ segment))))
\end{aligned}
$$

DEFINITION:
add-adp $(adp,\ n) = \text{cons}(\text{adp-name}(adp),\ \text{adp-offset}(adp) + n)$

DEFINITION: add1-adp $(adp) = \text{add-adp}(adp,\ \mathtt{1})$

DEFINITION: sub1-adp $(adp) = \text{sub-adp}(adp,\ \mathtt{1})$

DEFINITION:
put $(val,\ n,\ lst)$

$$
\begin{aligned}
= \quad & \textbf{if } n \simeq \mathtt{0} \\
& \textbf{then if } \text{listp}(lst) \ \textbf{ then } \text{cons}(val,\ \text{cdr}(lst)) \\
& \qquad\quad \textbf{else } \text{list}(val) \ \textbf{endif} \\
& \textbf{else } \text{cons}(\text{car}(lst),\ \text{put}(val,\ n - 1,\ \text{cdr}(lst))) \ \textbf{endif}
\end{aligned}
$$

DEFINITION:
fetch-adp $(adp,\ segment)$

$$= \quad \text{get}(\text{adp-offset}(adp),\ \text{definiens}(\text{adp-name}(adp),\ segment))$$

DEFINITION:
deposit-adp $(val,\ adp,\ segment)$

$$
\begin{aligned}
= \quad \text{put-value}(&\text{put}(val,\ \text{adp-offset}(adp),\ \text{definiens}(\text{adp-name}(adp),\ segment)), \\
& \text{adp-name}(adp), \\
& segment)
\end{aligned}
$$

DEFINITION: addressp $(x,\ segment) = \text{adpp}(\text{untag}(x),\ segment)$

DEFINITION: offset $(x) = \text{adp-offset}(\text{untag}(x))$

DEFINITION:
add-addr $(addr,\ n) = \text{tag}(\text{type}(addr),\ \text{add-adp}(\text{untag}(addr),\ n))$

DEFINITION: add1-addr $(addr)$ = add-addr $(addr, \mathtt{1})$

DEFINITION: sub1-addr $(addr)$ = sub-addr $(addr, \mathtt{1})$

DEFINITION:
fetch $(addr, segment)$ = fetch-adp (untag $(addr)$, $segment$)

DEFINITION:
deposit $(val, addr, segment)$ = deposit-adp (val, untag $(addr)$, $segment$)

DEFINITION: booleanp $(x) = ((x = \mathtt{'t}) \vee (x = \mathtt{'f}))$

DEFINITION:
bool (x)
= tag ('bool,
  if x **then** 't
  else 'f **endif**)

DEFINITION:
or-bool (x, y)
= **if** x = 'f **then** y
  else 't **endif**

DEFINITION:
and-bool (x, y)
= **if** x = 'f **then** 'f
  else y **endif**

DEFINITION:
xor-bool (x, y)
= **if** x = 'f **then** y
  elseif y = 'f **then** 't
  else 'f **endif**

DEFINITION:
small-naturalp $(i, \textit{word-size}) = ((i \in \mathbf{N}) \wedge (i < \exp(2, \textit{word-size})))$

DEFINITION:
bool-to-nat $(\textit{flg})$
= **if** $\textit{flg}$ = 'f **then** 0
  else 1 **endif**

DEFINITION:
fix-small-natural $(n, \textit{word-size}) = (n \ \mathbf{mod} \ \exp(2, \textit{word-size}))$

Definition:
small-integerp $(i,\ \mathit{word\text{-}size})$
$= \quad (\text{integerp}\,(i)$
$\quad\quad \wedge \quad (\neg\ \text{ilessp}\,(i,\ -\ \exp\,(2,\ \mathit{word\text{-}size}\ -\ 1)))$
$\quad\quad \wedge \quad \text{ilessp}\,(i,\ \exp\,(2,\ \mathit{word\text{-}size}\ -\ 1)))$

Definition:
inegate (i)
$=$ **if** negativep (i) **then** negative-guts (i)
elseif $i \simeq 0$ **then** 0
else $-\ i$ **endif**

Definition:
fix-small-integer $(i,\ \mathit{word\text{-}size})$
$=$ **if** small-integerp $(i,\ \mathit{word\text{-}size})$ **then** i
elseif negativep (i) **then** iplus $(i,\ \exp\,(2,\ \mathit{word\text{-}size}))$
else iplus $(i,\ -\ \exp\,(2,\ \mathit{word\text{-}size}))$ **endif**

Definition: bitp $(x) = ((x = 0) \vee (x = 1))$

Definition:
bit-vectorp $(x,\ n)$
$=$ **if** $x \simeq$ **nil** **then** $(x =$ **nil**$) \wedge (n \simeq 0)$
else $(n \not\simeq 0) \wedge$ bitp (car (x)) $\wedge$ bit-vectorp (cdr (x), $n - 1$) **endif**

Definition:
or-bit $(\mathit{bit1},\ \mathit{bit2})$
$=$ **if** $\mathit{bit1} = 0$
then if $\mathit{bit2} = 0$ **then** 0
else 1 **endif**
else 1 **endif**

Definition:
not-bit (bit)
$=$ **if** $\mathit{bit} = 0$ **then** 1
else 0 **endif**

Definition:
and-bit $(\mathit{bit1},\ \mathit{bit2})$
$=$ **if** $\mathit{bit1} = 0$ **then** 0
elseif $\mathit{bit2} = 0$ **then** 0
else 1 **endif**

Definition:
xor-bit $(\mathit{bit1},\ \mathit{bit2})$

$=$ **if** $bit1 = \mathtt{0}$
then if $bit2 = \mathtt{0}$ **then** $\mathtt{0}$
else $\mathtt{1}$ **endif**
elseif $bit2 = \mathtt{0}$ **then** $\mathtt{1}$
else $\mathtt{0}$ **endif**

DEFINITION:
or-bitv (a, b)
$=$ **if** $a \simeq$ **nil** **then nil**
else cons (or-bit (car (a), car (b)), or-bitv (cdr (a), cdr (b))) **endif**

DEFINITION:
not-bitv (a)
$=$ **if** $a \simeq$ **nil** **then nil**
else cons (not-bit (car (a)), not-bitv (cdr (a))) **endif**

DEFINITION:
and-bitv (a, b)
$=$ **if** $a \simeq$ **nil** **then nil**
else cons (and-bit (car (a), car (b)), and-bitv (cdr (a), cdr (b))) **endif**

DEFINITION:
xor-bitv (a, b)
$=$ **if** $a \simeq$ **nil** **then nil**
else cons (xor-bit (car (a), car (b)), xor-bitv (cdr (a), cdr (b))) **endif**

DEFINITION:
all-but-last (a)
$=$ **if** $a \simeq$ **nil** **then nil**
elseif cdr $(a) \simeq$ **nil** **then nil**
else cons (car (a), all-but-last (cdr (a))) **endif**

DEFINITION: rsh-bitv (a) = cons ($\mathtt{0}$, all-but-last (a))

DEFINITION: lsh-bitv (a) = append (cdr (a), `'(0)`)

DEFINITION:
all-zero-bitvp (a)
$=$ **if** listp (a) **then** (car $(a) = \mathtt{0}$) $\land$ all-zero-bitvp (cdr (a))
else t endif

DEFINITION:
find-labelp (x, lst)
$=$ **if** $lst \simeq$ **nil** **then f**
elseif labelledp (car (lst)) $\land$ (x = cadar (lst)) **then t**
else find-labelp (x, cdr (lst)) **endif**

DEFINITION:
x-y-error-msg $(x,\ y)$
$=$ pack (append (unpack (`'illegal-`),
append (unpack (y), cdr (unpack (`'g-instruction`)))))

DEFINITION:
cstkp $(x,\ segment)$
$=$ ((type (x) $=$ `'sys-addr`)
$\land$ addressp $(x,\ segment)$
$\land$ (area-name (x) $=$ `'cstk`)
$\land$ (cddr (x) $=$ **nil**))

DEFINITION:
tstkp $(x,\ segment)$
$=$ ((type (x) $=$ `'sys-addr`)
$\land$ addressp $(x,\ segment)$
$\land$ (area-name (x) $=$ `'tstk`)
$\land$ (cddr (x) $=$ **nil**))

DEFINITION: push-stk $(sys\text{-}addr)$ $=$ sub1-addr $(sys\text{-}addr)$

DEFINITION: pop-stk $(sys\text{-}addr)$ $=$ add1-addr $(sys\text{-}addr)$

DEFINITION:
stk-length $(addr,\ segment)$
$=$ ((length (definiens (area-name $(addr)$, $segment$)) $-$ offset $(addr)$) $-$ 1)

DEFINITION:
empty-stkp $(addr,\ segment)$ $=$ (stk-length $(addr,\ segment)$ $\simeq$ `0`)

DEFINITION: free-stk-size $(addr)$ $=$ offset $(addr)$

DEFINITION:
icode-labelp $(lab,\ segment)$
$=$ (definedp (adp-name (lab), $segment$)
$\land$ find-labelp (lab, definiens (adp-name (lab), $segment$)))

DEFINITION:
i-objectp $(x,\ i)$
$=$ (listp (x)
$\land$ (cddr (x) $=$ **nil**)
$\land$ **case on** type (x):
case $=$ *nat*
then small-naturalp (untag (x), i-word-size (i))
case $=$ *int*
then small-integerp (untag (x), i-word-size (i))

case $=$ *bitv*
then bit-vectorp (untag (x), i-word-size (i))
case $=$ *bool*
then booleanp (untag (x))
case $=$ *addr*
then adpp (untag (x), i-usr-data-segment (i))
case $=$ *pc*
then icode-labelp (untag (x), i-prog-segment (i))
case $=$ *subr*
then adpp (cons (untag (x), `0`), i-prog-segment (i))
case $=$ *sys-addr*
then adpp (untag (x), i-sys-data-segment (i))
case $=$ *ipc*
then adpp (untag (x), i-prog-segment (i))
otherwise f endcase)

DEFINITION:
i-objectp-type (*type*, x, i) $=$ ((*type* $=$ type (x)) $\land$ i-objectp (x, i))

DEFINITION:
i-usr-data-typep (*type*)
$=$ (*type* $\in$ `'(nat int bitv bool addr subr pc)`)

DEFINITION:
i-usr-data-objectp (x, i) $=$ (i-objectp (x, i) $\land$ i-usr-data-typep (type (x)))

DEFINITION:
total-i-system-size (i, *load-addr*)
$=$ (*load-addr*
$+$ segment-length (i-usr-data-segment (i))
$+$ segment-length (i-prog-segment (i))
$+$ segment-length (i-sys-data-segment (i)))

DEFINITION:
i-loadablep (i, *load-addr*)
$=$ (total-i-system-size (i, *load-addr*) $<$ exp (`2`, i-word-size (i)))

DEFINITION:
i-state-okp (i, *load-addr*)
$=$ (i-statep (i)
$\land$ i-objectp-type (`'ipc`, i-pc (i), i)
$\land$ cstkp (i-cfp (i), i-sys-data-segment (i))
$\land$ cstkp (i-csp (i), i-sys-data-segment (i))
$\land$ tstkp (i-tsp (i), i-sys-data-segment (i))
$\land$ i-objectp-type (`'bool`, i-c-flg (i), i)

$\wedge$ i-objectp-type (`'bool`, i-v-flg (i), i)
$\wedge$ i-objectp-type (`'bool`, i-n-flg (i), i)
$\wedge$ i-objectp-type (`'bool`, i-z-flg (i), i)
$\wedge$ i-loadablep (i, *load-addr*))

DEFINITION:
all-i-usr-data-objectps (lst, i)
= **if** $lst \simeq$ **nil** **then** $lst =$ **nil**
else i-usr-data-objectp (car (lst), i)
$\wedge$ all-i-usr-data-objectps (cdr (lst), i) **endif**

DEFINITION:
proper-i-usr-data-area ($area$, i)
= (litatom (car ($area$))
$\wedge$ listp (cdr ($area$))
$\wedge$ all-i-usr-data-objectps (cdr ($area$), i))

DEFINITION:
proper-i-usr-data-segmentp (*data-segment*, i)
= **if** *data-segment* $\simeq$ **nil** **then** *data-segment* = **nil**
else proper-i-usr-data-area (car (*data-segment*), i)
$\wedge$ ($\neg$ definedp (caar (*data-segment*), cdr (*data-segment*)))
$\wedge$ proper-i-usr-data-segmentp (cdr (*data-segment*), i) **endif**

DEFINITION: add1-i-pc (i) = add1-addr (i-pc (i))

DEFINITION:
add1-i-pcp (i) = addressp (add1-i-pc (i), i-prog-segment (i))

DEFINITION: add2-i-pc (i) = add1-addr (add1-addr (i-pc (i)))

DEFINITION:
add2-i-pcp (i) = addressp (add2-i-pc (i), i-prog-segment (i))

DEFINITION:
i-current-instruction (i) = unlabel (fetch (i-pc (i), i-prog-segment (i)))

DEFINITION:
i-nextword (i) = unlabel (fetch (add1-i-pc (i), i-prog-segment (i)))

DEFINITION:
ipc (*lab*, *program*)
= tag (`'ipc`, cons (name (*program*), find-label (*lab*, cdr (*program*))))

DEFINITION:
i-halt (*i*, *psw*)
= i-state (i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
i-tsp (*i*),
i-x (*i*),
i-y (*i*),
i-c-flg (*i*),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
i-sys-data-segment (*i*),
i-word-size (*i*),
psw)

DEFINITION:
xor-xxx (*obj1*, *obj2*, *i*, *load-addr*)
= xor-bitv (v-to-bitv (link-data-word (*obj1*,
i-link-tables (*i*, *load-addr*),
i-word-size (*i*))),
v-to-bitv (link-data-word (*obj2*,
i-link-tables (*i*, *load-addr*),
i-word-size (*i*))))

DEFINITION:
i-add_<c>_x_x{n}-okp (*i*)
= (add1-i-pcp (*i*) ∧ i-objectp-type (**'nat**, i-x (*i*), *i*))

DEFINITION:
i-add_<c>_x_x{n}-step (*i*)
= i-state (add1-i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
i-tsp (*i*),
tag (**'nat**,
fix-small-natural (2 ∗ untag (i-x (*i*)), i-word-size (*i*))),
i-y (*i*),
bool (¬ small-naturalp (2 ∗ untag (i-x (*i*)), i-word-size (*i*))),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),

```
                 i-usr-data-segment (i),
                 i-sys-data-segment (i),
                 i-word-size (i),
                 'run)
```

DEFINITION:

```
i-add_<tsp>_<tsp>{v}-okp (i)
=   (add1-i-pcp (i)
      ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
      ∧   i-objectp-type ('bitv,
                          fetch (i-tsp (i), i-sys-data-segment (i)),
                          i))
```

DEFINITION:

```
i-add_<tsp>_<tsp>{v}-step (i)
=   i-state (add1-i-pc (i),
                 i-cfp (i),
                 i-csp (i),
                 i-tsp (i),
                 i-x (i),
                 i-y (i),
                 i-c-flg (i),
                 i-v-flg (i),
                 i-n-flg (i),
                 i-z-flg (i),
                 i-prog-segment (i),
                 i-usr-data-segment (i),
                 deposit (tag ('bitv,
                               lsh-bitv (untag (fetch (i-tsp (i),
                                                       i-sys-data-segment (i))))),
                          i-tsp (i),
                          i-sys-data-segment (i)),
                 i-word-size (i),
                 'run)
```

DEFINITION:

```
i-add_<tsp>_<tsp>{n}-okp (i)
=   (add1-i-pcp (i)
      ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
      ∧   i-objectp-type ('nat, fetch (i-tsp (i), i-sys-data-segment (i)), i)
      ∧   small-naturalp (2 * untag (fetch (i-tsp (i),
                                             i-sys-data-segment (i))),
                          i-word-size (i)))
```

DEFINITION:

```
i-add_<tsp>_<tsp>{n}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             deposit (tag ('nat,
                           2 * untag (fetch (i-tsp (i), i-sys-data-segment (i)))),
                      i-tsp (i),
                      i-sys-data-segment (i)),
             i-word-size (i),
             'run)
```

DEFINITION:

```
i-add_<tsp>{a}_x{n}-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('nat, i-x (i), i)
     ∧  i-objectp-type ('addr,
                        fetch (i-tsp (i), i-sys-data-segment (i)),
                        i)
     ∧  i-objectp-type ('addr,
                        add-addr (fetch (i-tsp (i), i-sys-data-segment (i)),
                                  untag (i-x (i))),
                        i))
```

DEFINITION:

```
i-add_<tsp>{a}_x{n}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
```

i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (add-addr (fetch (i-tsp (i), i-sys-data-segment (i)),
untag (i-x (i))),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-add_tsp_*{n}-okp (i)
= (add2-i-pcp (i)
$\wedge$ i-objectp-type (**'nat**, i-nextword (i), i)
$\wedge$ addressp (add-addr (i-tsp (i), untag (i-nextword (i))),
i-sys-data-segment (i)))

DEFINITION:
i-add_tsp_*{n}-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
i-csp (i),
add-addr (i-tsp (i), untag (i-nextword (i))),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-add_tsp_x{n}-okp (i)
= (add1-i-pcp (i)
$\wedge$ i-objectp-type (**'nat**, i-x (i), i)
$\wedge$ addressp (add-addr (i-tsp (i), untag (i-x (i))),
i-sys-data-segment (i)))

DEFINITION:
i-add_tsp_x{n}-step (i)
= i-state (add1-i-pc (i),

i-cfp (i),
i-csp (i),
add-addr (i-tsp (i), untag (i-x (i))),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-add_<tsp>{i}_x{i}-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (**'int**, i-x (i), i)
∧ i-objectp-type (**'int**, fetch (i-tsp (i), i-sys-data-segment (i)), i)
∧ small-integerp (iplus (untag (fetch (i-tsp (i),
i-sys-data-segment (i))),
untag (i-x (i))),
i-word-size (i)))

DEFINITION:
i-add_<tsp>{i}_x{i}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (**'int**,
iplus (untag (fetch (i-tsp (i), i-sys-data-segment (i))),
untag (i-x (i)))),
i-tsp (i),

i-sys-data-segment (*i*)),
i-word-size (*i*),
'run)

DEFINITION:
i-add_<tsp>{n}_x{n}-okp (*i*)
= (add1-i-pcp (*i*)
∧ (¬ empty-stkp (i-tsp (*i*), i-sys-data-segment (*i*)))
∧ i-objectp-type (**'nat**, i-x (*i*), *i*)
∧ i-objectp-type (**'nat**, fetch (i-tsp (*i*), i-sys-data-segment (*i*)), *i*)
∧ small-naturalp (untag (fetch (i-tsp (*i*), i-sys-data-segment (*i*)))
\+ untag (i-x (*i*)),
i-word-size (*i*)))

DEFINITION:
i-add_<tsp>{n}_x{n}-step (*i*)
= i-state (add1-i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
i-tsp (*i*),
i-x (*i*),
i-y (*i*),
i-c-flg (*i*),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
deposit (tag (**'nat**,
untag (fetch (i-tsp (*i*), i-sys-data-segment (*i*)))
\+ untag (i-x (*i*))),
i-tsp (*i*),
i-sys-data-segment (*i*)),
i-word-size (*i*),
'run)

DEFINITION:
i-add_pc_x{n}-okp (*i*)
= (i-objectp-type (**'nat**, i-x (*i*), *i*)
∧ addressp (add-addr (i-pc (*i*), 1 + untag (i-x (*i*))),
i-prog-segment (*i*)))

DEFINITION:
i-add_pc_x{n}-step (*i*)
= i-state (add-addr (i-pc (*i*), 1 + untag (i-x (*i*))),

i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-add_x_x{n}-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (**'nat**, i-x (i), i)
∧ small-naturalp (2 $*$ untag (i-x (i)), i-word-size (i)))

DEFINITION:
i-add_x_x{n}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
tag (**'nat**, 2 $*$ untag (i-x (i))),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-add_x{n}_csp-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (**'nat**, i-x (i), i)
∧ addressp (add-addr (i-csp (i), untag (i-x (i))),
i-sys-data-segment (i)))

DEFINITION:

```
i-add_x{n}_csp-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             add-addr (i-csp (i), untag (i-x (i))),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             i-sys-data-segment (i),
             i-word-size (i),
             'run)
```

DEFINITION:

```
i-addc_<c>_x{n}_y{n}-okp (i)
=   (add1-i-pcp (i)
      ∧  i-objectp-type ('nat, i-x (i), i)
      ∧  i-objectp-type ('nat, i-y (i), i))
```

DEFINITION:

```
i-addc_<c>_x{n}_y{n}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             tag ('nat,
                  fix-small-natural (bool-to-nat (untag (i-c-flg (i)))
                                     +  untag (i-y (i))
                                     +  untag (i-x (i)),
                                     i-word-size (i))),
             i-y (i),
             bool (¬ small-naturalp (bool-to-nat (untag (i-c-flg (i)))
                                     +  untag (i-y (i))
                                     +  untag (i-x (i)),
                                     i-word-size (i))),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
```

i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-addc_<v>_x{i}_y{i}-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (**'int**, i-x (i), i)
∧ i-objectp-type (**'int**, i-y (i), i))

DEFINITION:
i-addc_<v>_x{i}_y{i}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
tag (**'int**,
fix-small-integer (iplus (bool-to-nat (untag (i-c-flg (i))),
iplus (untag (i-y (i)), untag (i-x (i)))),
i-word-size (i))),
i-y (i),
i-c-flg (i),
bool (¬ small-integerp (iplus (bool-to-nat (untag (i-c-flg (i))),
iplus (untag (i-y (i)),
untag (i-x (i)))),
i-word-size (i))),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-and_<tsp>{v}_x{v}-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (**'bitv**, i-x (i), i)
∧ i-objectp-type (**'bitv**,
fetch (i-tsp (i), i-sys-data-segment (i)),
i))

DEFINITION:

```
i-and_<tsp>{v}_x{v}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             deposit (tag ('bitv,
                           and-bitv (untag (fetch (i-tsp (i),
                                                   i-sys-data-segment (i))),
                                     untag (i-x (i)))),
                      i-tsp (i),
                      i-sys-data-segment (i)),
             i-word-size (i),
             'run)
```

DEFINITION:

```
i-and_<tsp>{b}_x{b}-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('bool, i-x (i), i)
     ∧  i-objectp-type ('bool,
                        fetch (i-tsp (i), i-sys-data-segment (i)),
                        i))
```

DEFINITION:

```
i-and_<tsp>{b}_x{b}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
```

```
            i-usr-data-segment (i),
            deposit (tag ('bool,
                          and-bool (untag (fetch (i-tsp (i),
                                                  i-sys-data-segment (i))),
                                    untag (i-x (i)))),
                     i-tsp (i),
                     i-sys-data-segment (i)),
            i-word-size (i),
            'run)
```

DEFINITION:

```
i-asr_<c>_<tsp>_<tsp>{b}-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('bool,
                        fetch (i-tsp (i), i-sys-data-segment (i)),
                        i))
```

DEFINITION:

```
i-asr_<c>_<tsp>_<tsp>{b}-step (i)
=   i-state (add1-i-pc (i),
            i-cfp (i),
            i-csp (i),
            i-tsp (i),
            i-x (i),
            i-y (i),
            fetch (i-tsp (i), i-sys-data-segment (i)),
            i-v-flg (i),
            i-n-flg (i),
            i-z-flg (i),
            i-prog-segment (i),
            i-usr-data-segment (i),
            deposit (tag ('bool, 'f), i-tsp (i), i-sys-data-segment (i)),
            i-word-size (i),
            'run)
```

DEFINITION:

```
i-cpop_cfp-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-csp (i), i-sys-data-segment (i)))
     ∧  cstkp (fetch (i-csp (i), i-sys-data-segment (i)),
               i-sys-data-segment (i)))
```

DEFINITION:

```
i-cpop_cfp-step (i)
```

= i-state (add1-i-pc (i),
fetch (i-csp (i), i-sys-data-segment (i)),
pop-stk (i-csp (i)),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-cpop_pc-okp (i)
= (($\neg$ empty-stkp (i-csp (i), i-sys-data-segment (i)))
$\wedge$ i-objectp-type (**'pc**, fetch (i-csp (i), i-sys-data-segment (i)), i))

DEFINITION:
i-cpop_pc-step (i)
= i-state (ipc (untag (fetch (i-csp (i), i-sys-data-segment (i))),
definition (area-name (fetch (i-csp (i), i-sys-data-segment (i))),
i-prog-segment (i))),
i-cfp (i),
pop-stk (i-csp (i)),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-cpush_*-okp (i)
= (add2-i-pcp (i)
$\wedge$ i-objectp (i-nextword (i), i)

∧ (free-stk-size (i-csp (i)) $\not\simeq$ 0))

Definition:
i-cpush_*-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
push-stk (i-csp (i)),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (i-nextword (i),
push-stk (i-csp (i)),
i-sys-data-segment (i)),
i-word-size (i),
'run)

Definition:
i-cpush_<tsp>+-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ (free-stk-size (i-csp (i)) $\not\simeq$ 0))

Definition:
i-cpush_<tsp>+-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
push-stk (i-csp (i)),
pop-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (fetch (i-tsp (i), i-sys-data-segment (i)),
push-stk (i-csp (i)),
i-sys-data-segment (i)),

i-word-size (i),
'**run**)

Definition:
i-cpush_cfp-okp (i) = (add1-i-pcp (i) $\land$ (free-stk-size (i-csp (i)) $\not\simeq$ 0))

Definition:
i-cpush_cfp-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
push-stk (i-csp (i)),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (i-cfp (i), push-stk (i-csp (i)), i-sys-data-segment (i)),
i-word-size (i),
'**run**)

Definition:
i-decr_<tsp>_<tsp>{i}-okp (i)
= (add1-i-pcp (i)
$\land$ ($\neg$ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
$\land$ i-objectp-type ('**int**, fetch (i-tsp (i), i-sys-data-segment (i)), i)
$\land$ small-integerp (idifference (untag (fetch (i-tsp (i),
i-sys-data-segment (i))),
1),
i-word-size (i)))

Definition:
i-decr_<tsp>_<tsp>{i}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),

i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (**'int**,
idifference (untag (fetch (i-tsp (i),
i-sys-data-segment (i))),
`1`)),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-decr_<tsp>_<tsp>{n}-okp (i)
= (add1-i-pcp (i)
$\land$ ($\neg$ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
$\land$ i-objectp-type (**'nat**, fetch (i-tsp (i), i-sys-data-segment (i)), i)
$\land$ (untag (fetch (i-tsp (i), i-sys-data-segment (i))) $\not\simeq$ `0`))

DEFINITION:
i-decr_<tsp>_<tsp>{n}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (**'nat**,
untag (fetch (i-tsp (i), i-sys-data-segment (i))) $-$ 1),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-incr_<tsp>_<tsp>{i}-okp (i)
= (add1-i-pcp (i)
$\land$ ($\neg$ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
$\land$ i-objectp-type (**'int**, fetch (i-tsp (i), i-sys-data-segment (i)), i)

```
     ∧  small-integerp (iplus (untag (fetch (i-tsp (i),
                                             i-sys-data-segment (i))),
                               1),
                         i-word-size (i)))
```

DEFINITION:

```
i-incr_<tsp>_<tsp>{i}-step (i)
=  i-state (add1-i-pc (i),
            i-cfp (i),
            i-csp (i),
            i-tsp (i),
            i-x (i),
            i-y (i),
            i-c-flg (i),
            i-v-flg (i),
            i-n-flg (i),
            i-z-flg (i),
            i-prog-segment (i),
            i-usr-data-segment (i),
            deposit (tag ('int,
                          iplus (untag (fetch (i-tsp (i), i-sys-data-segment (i))),
                                 1)),
                     i-tsp (i),
                     i-sys-data-segment (i)),
            i-word-size (i),
            'run)
```

DEFINITION:

```
i-incr_<tsp>_<tsp>{n}-okp (i)
=  (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('nat, fetch (i-tsp (i), i-sys-data-segment (i)), i)
     ∧  small-naturalp (1 + untag (fetch (i-tsp (i), i-sys-data-segment (i))),
                        i-word-size (i)))
```

DEFINITION:

```
i-incr_<tsp>_<tsp>{n}-step (i)
=  i-state (add1-i-pc (i),
            i-cfp (i),
            i-csp (i),
            i-tsp (i),
            i-x (i),
            i-y (i),
            i-c-flg (i),
            i-v-flg (i),
```

i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (**'nat**,
1 + untag (fetch (i-tsp (i), i-sys-data-segment (i)))),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-incr_y_y{n}-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (**'nat**, i-y (i), i)
∧ small-naturalp (1 + untag (i-y (i)), i-word-size (i)))

DEFINITION:
i-incr_y_y{n}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
tag (**'nat**, 1 + untag (i-y (i))),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-int-to-nat-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (**'int**, fetch (i-tsp (i), i-sys-data-segment (i)), i)
∧ (¬ negativep (untag (fetch (i-tsp (i), i-sys-data-segment (i))))))

DEFINITION:
i-int-to-nat-step (i)
= i-state (add1-i-pc (i),

```
               i-cfp (i),
               i-csp (i),
               i-tsp (i),
               i-x (i),
               i-y (i),
               i-c-flg (i),
               i-v-flg (i),
               i-n-flg (i),
               i-z-flg (i),
               i-prog-segment (i),
               i-usr-data-segment (i),
               deposit (tag ('nat,
                             untag (fetch (i-tsp (i), i-sys-data-segment (i)))),
                        i-tsp (i),
                        i-sys-data-segment (i)),
               i-word-size (i),
               'run)
```

DEFINITION:

```
i-jump-n_x-okp (i)
=    if untag (i-n-flg (i)) = 'f  then add1-i-pcp (i)
     else i-objectp-type ('pc, i-x (i), i) endif
```

DEFINITION:

```
i-jump-n_x-step (i)
=    i-state (if untag (i-n-flg (i)) = 'f  then add1-i-pc (i)
               else ipc (untag (i-x (i)),
                          definition (area-name (i-x (i)), i-prog-segment (i))) endif,
               i-cfp (i),
               i-csp (i),
               i-tsp (i),
               i-x (i),
               i-y (i),
               i-c-flg (i),
               i-v-flg (i),
               i-n-flg (i),
               i-z-flg (i),
               i-prog-segment (i),
               i-usr-data-segment (i),
               i-sys-data-segment (i),
               i-word-size (i),
               'run)
```

DEFINITION:

```
i-jump-nn_x-okp (i)
```

= **if** untag (i-n-flg (i)) = `'f` **then** i-objectp-type (`'pc`, i-x (i), i)
  else add1-i-pcp (i) **endif**

DEFINITION:
i-jump-nn_x-step (i)
= i-state (**if** untag (i-n-flg (i)) = `'f`
    then ipc (untag (i-x (i)),
        definition (area-name (i-x (i)), i-prog-segment (i)))
    else add1-i-pc (i) **endif**,
    i-cfp (i),
    i-csp (i),
    i-tsp (i),
    i-x (i),
    i-y (i),
    i-c-flg (i),
    i-v-flg (i),
    i-n-flg (i),
    i-z-flg (i),
    i-prog-segment (i),
    i-usr-data-segment (i),
    i-sys-data-segment (i),
    i-word-size (i),
    `'run`)

DEFINITION:
i-jump-nz_x-okp (i)
= **if** untag (i-z-flg (i)) = `'f` **then** i-objectp-type (`'pc`, i-x (i), i)
  else add1-i-pcp (i) **endif**

DEFINITION:
i-jump-nz_x-step (i)
= i-state (**if** untag (i-z-flg (i)) = `'f`
    then ipc (untag (i-x (i)),
        definition (area-name (i-x (i)), i-prog-segment (i)))
    else add1-i-pc (i) **endif**,
    i-cfp (i),
    i-csp (i),
    i-tsp (i),
    i-x (i),
    i-y (i),
    i-c-flg (i),
    i-v-flg (i),
    i-n-flg (i),
    i-z-flg (i),
    i-prog-segment (i),

i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-jump-z_x-okp (i)
= **if** untag (i-z-flg (i)) = **'f** **then** add1-i-pcp (i)
else i-objectp-type (**'pc**, i-x (i), i) **endif**

DEFINITION:
i-jump-z_x-step (i)
= i-state (**if** untag (i-z-flg (i)) = **'f** **then** add1-i-pc (i)
else ipc (untag (i-x (i)),
definition (area-name (i-x (i)), i-prog-segment (i))) **endif**,
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-jump_*-okp (i)
= (add1-i-pcp (i) $\land$ i-objectp-type (**'pc**, i-nextword (i), i))

DEFINITION:
i-jump_*-step (i)
= i-state (ipc (untag (i-nextword (i)),
definition (area-name (i-nextword (i)), i-prog-segment (i))),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),

i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
`'run`)

DEFINITION:
i-jump_x{subr}-okp (i) = i-objectp-type (`'subr`, i-x (i), i)

DEFINITION:
i-jump_x{subr}-step (i)
= i-state (tag (`'ipc`, cons (untag (i-x (i)), `0`)),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
`'run`)

DEFINITION:
i-lsr_<c>_x_x{n}-okp (i)
= (add1-i-pcp (i) $\wedge$ i-objectp-type (`'nat`, i-x (i), i))

DEFINITION:
i-lsr_<c>_x_x{n}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
tag (`'nat`, untag (i-x (i)) $\div$ `2`),
i-y (i),
bool ((untag (i-x (i)) **mod** `2`) = `1`),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),

```
                 i-prog-segment (i),
                 i-usr-data-segment (i),
                 i-sys-data-segment (i),
                 i-word-size (i),
                 'run)
```

DEFINITION:

```
i-lsr_<tsp>_<tsp>{v}-okp (i)
=    (add1-i-pcp (i)
      ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
      ∧   i-objectp-type ('bitv,
                          fetch (i-tsp (i), i-sys-data-segment (i)),
                          i))
```

DEFINITION:

```
i-lsr_<tsp>_<tsp>{v}-step (i)
=    i-state (add1-i-pc (i),
              i-cfp (i),
              i-csp (i),
              i-tsp (i),
              i-x (i),
              i-y (i),
              i-c-flg (i),
              i-v-flg (i),
              i-n-flg (i),
              i-z-flg (i),
              i-prog-segment (i),
              i-usr-data-segment (i),
              deposit (tag ('bitv,
                            rsh-bitv (untag (fetch (i-tsp (i),
                                                    i-sys-data-segment (i))))),
                       i-tsp (i),
                       i-sys-data-segment (i)),
              i-word-size (i),
              'run)
```

DEFINITION:

```
i-move-c_<tsp>_*-okp (i)
=    (add2-i-pcp (i)
      ∧   i-objectp (i-nextword (i), i)
      ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i))))
```

DEFINITION:

```
i-move-c_<tsp>_*-step (i)
=    i-state (add2-i-pc (i),
```

i-cfp (*i*),
i-csp (*i*),
i-tsp (*i*),
i-x (*i*),
i-y (*i*),
i-c-flg (*i*),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
if untag (i-c-flg (*i*)) = `'f` **then** i-sys-data-segment (*i*)
else deposit (i-nextword (*i*),
i-tsp (*i*),
i-sys-data-segment (*i*)) **endif**,
i-word-size (*i*),
'run)

Definition:
i-move-v_<tsp>_*-okp (*i*)
= (add2-i-pcp (*i*)
∧ i-objectp (i-nextword (*i*), *i*)
∧ (¬ empty-stkp (i-tsp (*i*), i-sys-data-segment (*i*))))

Definition:
i-move-v_<tsp>_*-step (*i*)
= i-state (add2-i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
i-tsp (*i*),
i-x (*i*),
i-y (*i*),
i-c-flg (*i*),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
if untag (i-v-flg (*i*)) = `'f` **then** i-sys-data-segment (*i*)
else deposit (i-nextword (*i*),
i-tsp (*i*),
i-sys-data-segment (*i*)) **endif**,
i-word-size (*i*),
'run)

Definition:

```
i-move-z_<tsp>_*-okp (i)
=   (add2-i-pcp (i)
     ∧  i-objectp (i-nextword (i), i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i))))
```

Definition:

```
i-move-z_<tsp>_*-step (i)
=   i-state (add2-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             if untag (i-z-flg (i)) = 'f then i-sys-data-segment (i)
             else deposit (i-nextword (i),
                           i-tsp (i),
                           i-sys-data-segment (i)) endif,
             i-word-size (i),
             'run)
```

Definition:

```
i-move-n_x_*-okp (i) = (add2-i-pcp (i) ∧ i-objectp (i-nextword (i), i))
```

Definition:

```
i-move-n_x_*-step (i)
=   i-state (add2-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             if untag (i-n-flg (i)) = 'f then i-x (i)
             else i-nextword (i) endif,
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
```

i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-move_<tsp>_*-okp (i)
= (add2-i-pcp (i)
∧ i-objectp (i-nextword (i), i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i))))

DEFINITION:
i-move_<tsp>_*-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (i-nextword (i), i-tsp (i), i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-move_<x{a}>_<tsp>-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (**'addr**, i-x (i), i))

DEFINITION:
i-move_<x{a}>_<tsp>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),

i-z-flg (i),
i-prog-segment (i),
deposit (fetch (i-tsp (i), i-sys-data-segment (i)),
i-x (i),
i-usr-data-segment (i)),
i-sys-data-segment (i),
i-word-size (i),
'**run**)

DEFINITION:
i-move_<x{s}>_<tsp>-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type ('**sys-addr**, i-x (i), i))

DEFINITION:
i-move_<x{s}>_<tsp>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (fetch (i-tsp (i), i-sys-data-segment (i)),
i-x (i),
i-sys-data-segment (i)),
i-word-size (i),
'**run**)

DEFINITION: i-move_cfp_csp-okp (i) = add1-i-pcp (i)

DEFINITION:
i-move_cfp_csp-step (i)
= i-state (add1-i-pc (i),
i-csp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),

i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION: i-move_csp_cfp-okp (i) = add1-i-pcp (i)

DEFINITION:
i-move_csp_cfp-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-cfp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION: i-move_x_*-okp (i) = add2-i-pcp (i)

DEFINITION:
i-move_x_*-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-nextword (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),

i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
`'run`)

DEFINITION:
i-move_x_<x{s}>-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (`'sys-addr`, i-x (i), i)
∧ i-objectp (fetch (i-x (i), i-sys-data-segment (i)), i))

DEFINITION:
i-move_x_<x{s}>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
fetch (i-x (i), i-sys-data-segment (i)),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
`'run`)

DEFINITION: i-move_x_tsp-okp (i) = add1-i-pcp (i)

DEFINITION:
i-move_x_tsp-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-tsp (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),

i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION: i-move_x_x-okp (i) = add1-i-pcp (i)

DEFINITION:
i-move_x_x-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION: i-move_y_*-okp (i) = add2-i-pcp (i)

DEFINITION:
i-move_y_*-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-nextword (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:

i-move_y_<y{s}>-okp (i)
= (add1-i-pcp (i)
  $\wedge$ i-objectp-type (**'sys-addr**, i-y (i), i)
  $\wedge$ i-objectp (fetch (i-y (i), i-sys-data-segment (i)), i))

DEFINITION:
i-move_y_<y{s}>-step (i)
= i-state (add1-i-pc (i),
 i-cfp (i),
 i-csp (i),
 i-tsp (i),
 i-x (i),
 fetch (i-y (i), i-sys-data-segment (i)),
 i-c-flg (i),
 i-v-flg (i),
 i-n-flg (i),
 i-z-flg (i),
 i-prog-segment (i),
 i-usr-data-segment (i),
 i-sys-data-segment (i),
 i-word-size (i),
 'run)

DEFINITION: i-move_y_tsp-okp (i) = add1-i-pcp (i)

DEFINITION:
i-move_y_tsp-step (i)
= i-state (add1-i-pc (i),
 i-cfp (i),
 i-csp (i),
 i-tsp (i),
 i-x (i),
 i-tsp (i),
 i-c-flg (i),
 i-v-flg (i),
 i-n-flg (i),
 i-z-flg (i),
 i-prog-segment (i),
 i-usr-data-segment (i),
 i-sys-data-segment (i),
 i-word-size (i),
 'run)

DEFINITION:
i-neg_<tsp>_<tsp>{i}-okp (i)

```
=    (add1-i-pcp (i)
       ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
       ∧   i-objectp-type ('int, fetch (i-tsp (i), i-sys-data-segment (i)), i)
       ∧   small-integerp (inegate (untag (fetch (i-tsp (i),
                                                 i-sys-data-segment (i)))),
                           i-word-size (i)))
```

DEFINITION:

```
i-neg_<tsp>_<tsp>{i}-step (i)
=    i-state (add1-i-pc (i),
              i-cfp (i),
              i-csp (i),
              i-tsp (i),
              i-x (i),
              i-y (i),
              i-c-flg (i),
              i-v-flg (i),
              i-n-flg (i),
              i-z-flg (i),
              i-prog-segment (i),
              i-usr-data-segment (i),
              deposit (tag ('int,
                            inegate (untag (fetch (i-tsp (i),
                                                   i-sys-data-segment (i))))),
                       i-tsp (i),
                       i-sys-data-segment (i)),
              i-word-size (i),
              'run)
```

DEFINITION:

```
i-not_<tsp>_<tsp>{v}-okp (i)
=    (add1-i-pcp (i)
       ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
       ∧   i-objectp-type ('bitv,
                           fetch (i-tsp (i), i-sys-data-segment (i)),
                           i))
```

DEFINITION:

```
i-not_<tsp>_<tsp>{v}-step (i)
=    i-state (add1-i-pc (i),
              i-cfp (i),
              i-csp (i),
              i-tsp (i),
              i-x (i),
              i-y (i),
```

```
                i-c-flg (i),
                i-v-flg (i),
                i-n-flg (i),
                i-z-flg (i),
                i-prog-segment (i),
                i-usr-data-segment (i),
                deposit (tag ('bitv,
                              not-bitv (untag (fetch (i-tsp (i),
                                                      i-sys-data-segment (i))))),
                         i-tsp (i),
                         i-sys-data-segment (i)),
                i-word-size (i),
                'run)
```

DEFINITION:

```
i-or_<tsp>{v}_x{v}-okp (i)
=    (add1-i-pcp (i)
      ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
      ∧  i-objectp-type ('bitv, i-x (i), i)
      ∧  i-objectp-type ('bitv,
                          fetch (i-tsp (i), i-sys-data-segment (i)),
                          i))
```

DEFINITION:

```
i-or_<tsp>{v}_x{v}-step (i)
=    i-state (add1-i-pc (i),
                i-cfp (i),
                i-csp (i),
                i-tsp (i),
                i-x (i),
                i-y (i),
                i-c-flg (i),
                i-v-flg (i),
                i-n-flg (i),
                i-z-flg (i),
                i-prog-segment (i),
                i-usr-data-segment (i),
                deposit (tag ('bitv,
                              or-bitv (untag (fetch (i-tsp (i),
                                                     i-sys-data-segment (i))),
                                       untag (i-x (i)))),
                         i-tsp (i),
                         i-sys-data-segment (i)),
                i-word-size (i),
                'run)
```

DEFINITION:

```
i-or_<tsp>{b}_x{b}-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('bool, i-x (i), i)
     ∧  i-objectp-type ('bool,
                        fetch (i-tsp (i), i-sys-data-segment (i)),
                        i))
```

DEFINITION:

```
i-or_<tsp>{b}_x{b}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             deposit (tag ('bool,
                           or-bool (untag (fetch (i-tsp (i),
                                                  i-sys-data-segment (i))),
                                    untag (i-x (i)))),
                      i-tsp (i),
                      i-sys-data-segment (i)),
             i-word-size (i),
             'run)
```

DEFINITION:

```
i-sub_<c>_<tsp>{a}_x{a}-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('addr, i-x (i), i)
     ∧  i-objectp-type ('addr,
                        fetch (i-tsp (i), i-sys-data-segment (i)),
                        i)
     ∧  (area-name (fetch (i-tsp (i), i-sys-data-segment (i)))
          =   area-name (i-x (i))))
```

DEFINITION:

```
i-sub_<c>_<tsp>{a}_x{a}-step (i)
```

```
=  i-state (add1-i-pc (i),
           i-cfp (i),
           i-csp (i),
           i-tsp (i),
           i-x (i),
           i-y (i),
           bool (offset (fetch (i-tsp (i), i-sys-data-segment (i)))
                 <  offset (i-x (i))),
           i-v-flg (i),
           i-n-flg (i),
           i-z-flg (i),
           i-prog-segment (i),
           i-usr-data-segment (i),
           deposit (tag ('nat,
                         if offset (fetch (i-tsp (i), i-sys-data-segment (i)))
                            <  offset (i-x (i))
                         then exp (2, i-word-size (i))
                              -  (offset (i-x (i))
                                  -  offset (fetch (i-tsp (i),
                                                   i-sys-data-segment (i))))
                         else offset (fetch (i-tsp (i),
                                             i-sys-data-segment (i)))
                              -  offset (i-x (i)) endif),
                    i-tsp (i),
                    i-sys-data-segment (i)),
           i-word-size (i),
           'run)
```

DEFINITION:

```
i-sub_<c>_<tsp>{n}_x{n}-okp (i)
=  (add1-i-pcp (i)
   ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
   ∧  i-objectp-type ('nat, i-x (i), i)
   ∧  i-objectp-type ('nat, fetch (i-tsp (i), i-sys-data-segment (i)), i))
```

DEFINITION:

```
i-sub_<c>_<tsp>{n}_x{n}-step (i)
=  i-state (add1-i-pc (i),
           i-cfp (i),
           i-csp (i),
           i-tsp (i),
           i-x (i),
           i-y (i),
           bool (untag (fetch (i-tsp (i), i-sys-data-segment (i)))
```

```
            <    untag (i-x (i))),
       i-v-flg (i),
       i-n-flg (i),
       i-z-flg (i),
       i-prog-segment (i),
       i-usr-data-segment (i),
       deposit (tag ('nat,
                     if untag (fetch (i-tsp (i), i-sys-data-segment (i)))
                        <    untag (i-x (i))
                     then exp (2, i-word-size (i))
                             -    (untag (i-x (i))
                                      -    untag (fetch (i-tsp (i),
                                                         i-sys-data-segment (i))))
                     else untag (fetch (i-tsp (i),
                                        i-sys-data-segment (i)))
                            -    untag (i-x (i)) endif),
                i-tsp (i),
                i-sys-data-segment (i)),
       i-word-size (i),
       'run)
```

DEFINITION:

```
i-sub_<nv>_<tsp>{i}_x{i}-okp (i)
=   (add1-i-pcp (i)
     ∧   i-objectp-type ('int, fetch (i-tsp (i), i-sys-data-segment (i)), i)
     ∧   i-objectp-type ('int, i-x (i), i))
```

DEFINITION:

```
i-sub_<nv>_<tsp>{i}_x{i}-step (i)
=   i-state (add1-i-pc (i),
       i-cfp (i),
       i-csp (i),
       i-tsp (i),
       i-x (i),
       i-y (i),
       i-c-flg (i),
       bool (¬ small-integerp (idifference (untag (fetch (i-tsp (i),
                                                          i-sys-data-segment (i))),
                                            untag (i-x (i))),
                               i-word-size (i))),
       bool (negativep (fix-small-integer (idifference (untag (fetch (i-tsp (i),
                                                                      i-sys-data-segment (i))),
                                                        untag (i-x (i))),
                                           i-word-size (i)))),
```

```
                i-z-flg (i),
                i-prog-segment (i),
                i-usr-data-segment (i),
                deposit (tag ('int,
                              fix-small-integer (idifference (untag (fetch (i-tsp (i),
                                                                            i-sys-data-segment (i))),
                                                             untag (i-x (i))),
                                                 i-word-size (i))),
                         i-tsp (i),
                         i-sys-data-segment (i)),
                i-word-size (i),
                'run)
```

DEFINITION:

```
i-sub_<tsp>{a}_x{n}-okp (i)
=   (add1-i-pcp (i)
     ∧  (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧  i-objectp-type ('nat, i-x (i), i)
     ∧  i-objectp-type ('addr,
                        fetch (i-tsp (i), i-sys-data-segment (i)),
                        i)
     ∧  (offset (fetch (i-tsp (i), i-sys-data-segment (i)))
           ≮  untag (i-x (i))))
```

DEFINITION:

```
i-sub_<tsp>{a}_x{n}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             deposit (sub-addr (fetch (i-tsp (i), i-sys-data-segment (i)),
                                untag (i-x (i))),
                      i-tsp (i),
                      i-sys-data-segment (i)),
             i-word-size (i),
             'run)
```

DEFINITION:
i-sub_x{s}_y{n}-okp (i)
= (add1-i-pcp (i)
  ∧ i-objectp-type (**'sys-addr**, i-x (i), i)
  ∧ i-objectp-type (**'nat**, i-y (i), i)
  ∧ (offset (i-x (i)) $\not<$ untag (i-y (i))))

DEFINITION:
i-sub_x{s}_y{n}-step (i)
= i-state (add1-i-pc (i),
    i-cfp (i),
    i-csp (i),
    i-tsp (i),
    sub-addr (i-x (i), untag (i-y (i))),
    i-y (i),
    i-c-flg (i),
    i-v-flg (i),
    i-n-flg (i),
    i-z-flg (i),
    i-prog-segment (i),
    i-usr-data-segment (i),
    i-sys-data-segment (i),
    i-word-size (i),
    'run)

DEFINITION:
i-sub_<tsp>{i}_x{i}-okp (i)
= (add1-i-pcp (i)
  ∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
  ∧ i-objectp-type (**'int**, i-x (i), i)
  ∧ i-objectp-type (**'int**, fetch (i-tsp (i), i-sys-data-segment (i)), i)
  ∧ small-integerp (idifference (untag (fetch (i-tsp (i),
                i-sys-data-segment (i))),
            untag (i-x (i))),
        i-word-size (i)))

DEFINITION:
i-sub_<tsp>{i}_x{i}-step (i)
= i-state (add1-i-pc (i),
    i-cfp (i),
    i-csp (i),
    i-tsp (i),
    i-x (i),
    i-y (i),
    i-c-flg (i),

```
                i-v-flg (i),
                i-n-flg (i),
                i-z-flg (i),
                i-prog-segment (i),
                i-usr-data-segment (i),
                deposit (tag ('int,
                              idifference (untag (fetch (i-tsp (i),
                                                         i-sys-data-segment (i))),
                                           untag (i-x (i)))),
                         i-tsp (i),
                         i-sys-data-segment (i)),
                i-word-size (i),
                'run)
```

DEFINITION:

```
i-sub_<tsp>{n}_x{n}-okp (i)
=   (add1-i-pcp (i)
     ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧   i-objectp-type ('nat, i-x (i), i)
     ∧   i-objectp-type ('nat, fetch (i-tsp (i), i-sys-data-segment (i)), i)
     ∧   (untag (fetch (i-tsp (i), i-sys-data-segment (i)))
          ≮   untag (i-x (i))))
```

DEFINITION:

```
i-sub_<tsp>{n}_x{n}-step (i)
=   i-state (add1-i-pc (i),
                i-cfp (i),
                i-csp (i),
                i-tsp (i),
                i-x (i),
                i-y (i),
                i-c-flg (i),
                i-v-flg (i),
                i-n-flg (i),
                i-z-flg (i),
                i-prog-segment (i),
                i-usr-data-segment (i),
                deposit (tag ('nat,
                              untag (fetch (i-tsp (i), i-sys-data-segment (i)))
                              -   untag (i-x (i))),
                         i-tsp (i),
                         i-sys-data-segment (i)),
                i-word-size (i),
                'run)
```

DEFINITION:
i-sub_<tsp>{s}_x{s}-okp (i)
= (add1-i-pcp (i)
$\wedge$ ($\neg$ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
$\wedge$ i-objectp-type (**'sys-addr**, i-x (i), i)
$\wedge$ i-objectp-type (**'sys-addr**,
fetch (i-tsp (i), i-sys-data-segment (i)),
i)
$\wedge$ (area-name (fetch (i-tsp (i), i-sys-data-segment (i)))
= area-name (i-x (i)))
$\wedge$ (offset (fetch (i-tsp (i), i-sys-data-segment (i)))
$\not<$ offset (i-x (i))))

DEFINITION:
i-sub_<tsp>{s}_x{s}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (**'nat**,
offset (fetch (i-tsp (i), i-sys-data-segment (i)))
$-$ offset (i-x (i))),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-sub_<z>_x{s}_y{s}-okp (i)
= (add1-i-pcp (i)
$\wedge$ i-objectp-type (**'sys-addr**, i-x (i), i)
$\wedge$ i-objectp-type (**'sys-addr**, i-y (i), i)
$\wedge$ (area-name (i-y (i)) = area-name (i-x (i)))
$\wedge$ (offset (i-x (i)) $\not<$ offset (i-y (i))))

DEFINITION:
i-sub_<z>_x{s}_y{s}-step (i)

= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
tag (**'nat**, offset (i-x (i)) − offset (i-y (i))),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
bool (offset (i-x (i)) = offset (i-y (i))),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-subb_<c>_x{n}_y{n}-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (**'nat**, i-x (i), i)
∧ i-objectp-type (**'nat**, i-y (i), i))

DEFINITION:
i-subb_<c>_x{n}_y{n}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
tag (**'nat**,
if untag (i-x (i))
< (untag (i-y (i))
+ bool-to-nat (untag (i-c-flg (i))))
then exp (2, i-word-size (i))
− ((untag (i-y (i))
+ bool-to-nat (untag (i-c-flg (i))))
− untag (i-x (i)))
else untag (i-x (i))
− (untag (i-y (i))
+ bool-to-nat (untag (i-c-flg (i)))) **endif**),
i-y (i),
bool (untag (i-x (i))
< (untag (i-y (i)) + bool-to-nat (untag (i-c-flg (i))))),
i-v-flg (i),
i-n-flg (i),

i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-subb_<v>_x{i}_y{i}-okp (i)
= (add1-i-pcp (i)
∧ i-objectp-type (**'int**, i-x (i), i)
∧ i-objectp-type (**'int**, i-y (i), i))

DEFINITION:
i-subb_<v>_x{i}_y{i}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
tag (**'int**,
fix-small-integer (idifference (untag (i-x (i)),
iplus (untag (i-y (i)),
bool-to-nat (untag (i-c-flg (i))))),
i-word-size (i))),
i-y (i),
i-c-flg (i),
bool (¬ small-integerp (idifference (untag (i-x (i)),
iplus (untag (i-y (i)),
bool-to-nat (untag (i-c-flg (i))))),
i-word-size (i))),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-tpop_<c>_x-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp (fetch (i-tsp (i), i-sys-data-segment (i)), i))

DEFINITION:

i-tpop_<c>_x-step (*i*)
= i-state (add1-i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
pop-stk (i-tsp (*i*)),
fetch (i-tsp (*i*), i-sys-data-segment (*i*)),
i-y (*i*),
tag (`'bool`, `'f`),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
i-sys-data-segment (*i*),
i-word-size (*i*),
`'run`)

DEFINITION:
i-tpop_<x{a}>-okp (*i*)
= (add1-i-pcp (*i*)
∧ (¬ empty-stkp (i-tsp (*i*), i-sys-data-segment (*i*)))
∧ i-objectp-type (`'addr`, i-x (*i*), *i*))

DEFINITION:
i-tpop_<x{a}>-step (*i*)
= i-state (add1-i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
pop-stk (i-tsp (*i*)),
i-x (*i*),
i-y (*i*),
i-c-flg (*i*),
i-v-flg (*i*),
i-n-flg (*i*),
i-z-flg (*i*),
i-prog-segment (*i*),
deposit (fetch (i-tsp (*i*), i-sys-data-segment (*i*)),
i-x (*i*),
i-usr-data-segment (*i*)),
i-sys-data-segment (*i*),
i-word-size (*i*),
`'run`)

DEFINITION:
i-tpop_<x{s}>-okp (*i*)

= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (`'sys-addr`, i-x (i), i))

DEFINITION:
i-tpop_<x{s}>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (fetch (i-tsp (i), i-sys-data-segment (i)),
i-x (i),
i-sys-data-segment (i)),
i-word-size (i),
`'run`)

DEFINITION:
i-tpop_pc-okp (i)
= ((¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (`'pc`, fetch (i-tsp (i), i-sys-data-segment (i)), i))

DEFINITION:
i-tpop_pc-step (i)
= i-state (ipc (untag (fetch (i-tsp (i), i-sys-data-segment (i))),
definition (area-name (fetch (i-tsp (i), i-sys-data-segment (i))),
i-prog-segment (i))),
i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),

i-sys-data-segment (i),
i-word-size (i),
'run)

Definition:
i-tpop_x-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp (fetch (i-tsp (i), i-sys-data-segment (i)), i))

Definition:
i-tpop_x-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
fetch (i-tsp (i), i-sys-data-segment (i)),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

Definition:
i-tpop_y-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp (fetch (i-tsp (i), i-sys-data-segment (i)), i))

Definition:
i-tpop_y-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
i-x (i),
fetch (i-tsp (i), i-sys-data-segment (i)),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),

i-z-flg (*i*),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
i-sys-data-segment (*i*),
i-word-size (*i*),
'run)

DEFINITION:
i-tpop{v}_<z>_y-okp (*i*)
= (add1-i-pcp (*i*)
∧ (¬ empty-stkp (i-tsp (*i*), i-sys-data-segment (*i*)))
∧ i-objectp-type (**'bitv**,
fetch (i-tsp (*i*), i-sys-data-segment (*i*)),
i))

DEFINITION:
i-tpop{v}_<z>_y-step (*i*)
= i-state (add1-i-pc (*i*),
i-cfp (*i*),
i-csp (*i*),
pop-stk (i-tsp (*i*)),
i-x (*i*),
fetch (i-tsp (*i*), i-sys-data-segment (*i*)),
i-c-flg (*i*),
i-v-flg (*i*),
i-n-flg (*i*),
bool (all-zero-bitvp (untag (fetch (i-tsp (*i*),
i-sys-data-segment (*i*))))),
i-prog-segment (*i*),
i-usr-data-segment (*i*),
i-sys-data-segment (*i*),
i-word-size (*i*),
'run)

DEFINITION:
i-tpop{b}_<z>_y-okp (*i*)
= (add1-i-pcp (*i*)
∧ (¬ empty-stkp (i-tsp (*i*), i-sys-data-segment (*i*)))
∧ i-objectp-type (**'bool**,
fetch (i-tsp (*i*), i-sys-data-segment (*i*)),
i))

DEFINITION:
i-tpop{b}_<z>_y-step (*i*)
= i-state (add1-i-pc (*i*),

i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
i-x (i),
fetch (i-tsp (i), i-sys-data-segment (i)),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
bool (untag (fetch (i-tsp (i), i-sys-data-segment (i))) = **'f**),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-tpop{i}_<zn>_y-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (**'int**, fetch (i-tsp (i), i-sys-data-segment (i)), i))

DEFINITION:
i-tpop{i}_<zn>_y-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
i-x (i),
fetch (i-tsp (i), i-sys-data-segment (i)),
i-c-flg (i),
i-v-flg (i),
bool (negativep (untag (fetch (i-tsp (i), i-sys-data-segment (i))))),
bool (untag (fetch (i-tsp (i), i-sys-data-segment (i))) = 0),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-tpop{n}_<z>_y-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (**'nat**, fetch (i-tsp (i), i-sys-data-segment (i)), i))

DEFINITION:
i-tpop{n}_<z>_y-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
pop-stk (i-tsp (i)),
i-x (i),
fetch (i-tsp (i), i-sys-data-segment (i)),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
bool (untag (fetch (i-tsp (i), i-sys-data-segment (i))) = 0),
i-prog-segment (i),
i-usr-data-segment (i),
i-sys-data-segment (i),
i-word-size (i),
'run)

DEFINITION:
i-tpush_*-okp (i)
= (add2-i-pcp (i)
∧ i-objectp (i-nextword (i), i)
∧ (free-stk-size (i-tsp (i)) $\not\simeq$ 0))

DEFINITION:
i-tpush_*-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
i-csp (i),
push-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (i-nextword (i),
push-stk (i-tsp (i)),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:

i-tpush_<x{a}>-okp (i)
= (add1-i-pcp (i)
$\wedge$ (free-stk-size (i-tsp (i)) $\not\simeq$ `0`)
$\wedge$ i-objectp-type (**'addr**, i-x (i), i)
$\wedge$ i-objectp (fetch (i-x (i), i-usr-data-segment (i)), i))

DEFINITION:
i-tpush_<x{a}>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
push-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (fetch (i-x (i), i-usr-data-segment (i)),
push-stk (i-tsp (i)),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-tpush_<x{s}>-okp (i)
= (add1-i-pcp (i)
$\wedge$ (free-stk-size (i-tsp (i)) $\not\simeq$ `0`)
$\wedge$ i-objectp-type (**'sys-addr**, i-x (i), i))

DEFINITION:
i-tpush_<x{s}>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
push-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),

i-usr-data-segment (i),
deposit (fetch (i-x (i), i-sys-data-segment (i)),
push-stk (i-tsp (i)),
i-sys-data-segment (i)),
i-word-size (i),
'**run**)

DEFINITION:
i-tpush_csp-okp (i) = (add1-i-pcp (i) $\land$ (free-stk-size (i-tsp (i)) $\not\simeq$ 0))

DEFINITION:
i-tpush_csp-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
push-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (i-csp (i), push-stk (i-tsp (i)), i-sys-data-segment (i)),
i-word-size (i),
'**run**)

DEFINITION:
i-tpush_tsp-okp (i) = (add1-i-pcp (i) $\land$ (free-stk-size (i-tsp (i)) $\not\simeq$ 0))

DEFINITION:
i-tpush_tsp-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
push-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),

deposit (i-tsp (i), push-stk (i-tsp (i)), i-sys-data-segment (i)),
i-word-size (i),
'**run**)

DEFINITION:
i-tpush_x-okp (i)
= (add1-i-pcp (i)
∧ (free-stk-size (i-tsp (i)) $\not\simeq$ 0)
∧ i-objectp (i-x (i), i))

DEFINITION:
i-tpush_x-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
push-stk (i-tsp (i)),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (i-x (i), push-stk (i-tsp (i)), i-sys-data-segment (i)),
i-word-size (i),
'**run**)

DEFINITION:
i-xor_<tsp>_<tsp>-okp (i)
= (add1-i-pcp (i) ∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i))))

DEFINITION:
i-xor_<tsp>_<tsp>-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),

```
                  i-usr-data-segment (i),
                  deposit (tag ('bool, 'f), i-tsp (i), i-sys-data-segment (i)),
                  i-word-size (i),
                  'run)
```

DEFINITION:

```
i-xor_<tsp>{v}_x{v}-okp (i)
=   (add1-i-pcp (i)
     ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧   i-objectp-type ('bitv, i-x (i), i)
     ∧   i-objectp-type ('bitv,
                         fetch (i-tsp (i), i-sys-data-segment (i)),
                         i))
```

DEFINITION:

```
i-xor_<tsp>{v}_x{v}-step (i)
=   i-state (add1-i-pc (i),
             i-cfp (i),
             i-csp (i),
             i-tsp (i),
             i-x (i),
             i-y (i),
             i-c-flg (i),
             i-v-flg (i),
             i-n-flg (i),
             i-z-flg (i),
             i-prog-segment (i),
             i-usr-data-segment (i),
             deposit (tag ('bitv,
                           xor-bitv (untag (fetch (i-tsp (i),
                                                   i-sys-data-segment (i))),
                                     untag (i-x (i)))),
                      i-tsp (i),
                      i-sys-data-segment (i)),
             i-word-size (i),
             'run)
```

DEFINITION:

```
i-xor_<tsp>{b}_*{b}-okp (i)
=   (add2-i-pcp (i)
     ∧   (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
     ∧   i-objectp-type ('bool,
                         fetch (i-tsp (i), i-sys-data-segment (i)),
                         i)
     ∧   i-objectp-type ('bool, i-nextword (i), i))
```

DEFINITION:
i-xor_<tsp>{b}_*{b}-step (i)
= i-state (add2-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),
i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (`'bool`,
xor-bool (untag (fetch (i-tsp (i),
i-sys-data-segment (i))),
untag (i-nextword (i)))),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
`'run`)

DEFINITION:
i-xor_<tsp>{b}_x{b}-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-objectp-type (`'bool`,
fetch (i-tsp (i), i-sys-data-segment (i)),
i)
∧ i-objectp-type (`'bool`, i-x (i), i))

DEFINITION:
i-xor_<tsp>{b}_x{b}-step (i)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
i-z-flg (i),

i-prog-segment (i),
i-usr-data-segment (i),
deposit (tag (**'bool**,
xor-bool (untag (fetch (i-tsp (i),
i-sys-data-segment (i))),
untag (i-x (i)))),
i-tsp (i),
i-sys-data-segment (i)),
i-word-size (i),
'run)

DEFINITION:
i-xor_<z>_<tsp>_x-okp (i)
= (add1-i-pcp (i)
∧ (¬ empty-stkp (i-tsp (i), i-sys-data-segment (i)))
∧ i-usr-data-objectp (i-x (i), i)
∧ i-usr-data-objectp (fetch (i-tsp (i), i-sys-data-segment (i)), i)
∧ (type (i-x (i)) = type (fetch (i-tsp (i), i-sys-data-segment (i)))))

```
; Note: This step function is different from all the others in that it
; takes a second argument!  The problem is that to xor any two
; addresses we must know where they are loaded to compute the
; resultant bit vector.  In the days before we added load-addr, the
; address 0 was built into this definition and this stepper was like
; all the others.  But now we have to supply it.  We could have made
; it part of the I state, but that would have required changing all
; the steppers to construct the new sized state and might have
; required other changes (e.g., in proving that it is unchanged?).  So
; we have just added it to this one stepper, which will be handled
; specially in by i-ins-step.  Of course, we have to add load-addr as
; an argument to the I machine itself.  Sigh.
```

DEFINITION:
i-xor_<z>_<tsp>_x-step (i, *load-addr*)
= i-state (add1-i-pc (i),
i-cfp (i),
i-csp (i),
i-tsp (i),
i-x (i),
i-y (i),
i-c-flg (i),
i-v-flg (i),
i-n-flg (i),
bool (fetch (i-tsp (i), i-sys-data-segment (i)) = i-x (i)),

```
        i-prog-segment (i),
        i-usr-data-segment (i),
        deposit (tag ('bitv,
                      xor-xxx (fetch (i-tsp (i), i-sys-data-segment (i)),
                               i-x (i),
                               i,
                               load-addr)),
              i-tsp (i),
              i-sys-data-segment (i)),
        i-word-size (i),
        'run)
```

DEFINITION:

```
i-ins-okp (ins, i)
=   ((cdr (ins) = nil)
     ∧  case on car (ins):
        case = add_<c>_x_x{n}
        then i-add_<c>_x_x{n}-okp (i)
        case = add_<tsp>_<tsp>{v}
         then i-add_<tsp>_<tsp>{v}-okp (i)
        case = add_<tsp>_<tsp>{n}
         then i-add_<tsp>_<tsp>{n}-okp (i)
        case = add_<tsp>{a}_x{n}
         then i-add_<tsp>{a}_x{n}-okp (i)
        case = add_tsp_*{n}
         then i-add_tsp_*{n}-okp (i)
        case = add_tsp_x{n}
         then i-add_tsp_x{n}-okp (i)
        case = add_<tsp>{i}_x{i}
         then i-add_<tsp>{i}_x{i}-okp (i)
        case = add_<tsp>{n}_x{n}
         then i-add_<tsp>{n}_x{n}-okp (i)
        case = add_pc_x{n}
         then i-add_pc_x{n}-okp (i)
        case = add_x_x{n}
         then i-add_x_x{n}-okp (i)
        case = add_x{n}_csp
         then i-add_x{n}_csp-okp (i)
        case = addc_<c>_x{n}_y{n}
         then i-addc_<c>_x{n}_y{n}-okp (i)
        case = addc_<v>_x{i}_y{i}
         then i-addc_<v>_x{i}_y{i}-okp (i)
        case = and_<tsp>{v}_x{v}
         then i-and_<tsp>{v}_x{v}-okp (i)
```

case = *and_<tsp>{b}_x{b}*
then i-and_<tsp>{b}_x{b}-okp (i)
case = *asr_<c>_<tsp>_<tsp>{b}*
then i-asr_<c>_<tsp>_<tsp>{b}-okp (i)
case = *cpop_cfp*
then i-cpop_cfp-okp (i)
case = *cpop_pc*
then i-cpop_pc-okp (i)
case = *cpush_**
then i-cpush_*-okp (i)
case = *cpush_<tsp>+*
then i-cpush_<tsp>+-okp (i)
case = *cpush_cfp*
then i-cpush_cfp-okp (i)
case = *decr_<tsp>_<tsp>{i}*
then i-decr_<tsp>_<tsp>{i}-okp (i)
case = *decr_<tsp>_<tsp>{n}*
then i-decr_<tsp>_<tsp>{n}-okp (i)
case = *incr_<tsp>_<tsp>{i}*
then i-incr_<tsp>_<tsp>{i}-okp (i)
case = *incr_<tsp>_<tsp>{n}*
then i-incr_<tsp>_<tsp>{n}-okp (i)
case = *incr_y_y{n}*
then i-incr_y_y{n}-okp (i)
case = *int-to-nat*
then i-int-to-nat-okp (i)
case = *jump-n_x*
then i-jump-n_x-okp (i)
case = *jump-nn_x*
then i-jump-nn_x-okp (i)
case = *jump-nz_x*
then i-jump-nz_x-okp (i)
case = *jump-z_x*
then i-jump-z_x-okp (i)
case = *jump_**
then i-jump_*-okp (i)
case = *jump_x{subr}*
then i-jump_x{subr}-okp (i)
case = *lsr_<c>_x_x{n}*
then i-lsr_<c>_x_x{n}-okp (i)
case = *lsr_<tsp>_<tsp>{v}*
then i-lsr_<tsp>_<tsp>{v}-okp (i)
case = *move-c_<tsp>_**
then i-move-c_<tsp>_*-okp (i)

```
case = move-v_<tsp>_*
  then i-move-v_<tsp>_*-okp (i)
case = move-z_<tsp>_*
  then i-move-z_<tsp>_*-okp (i)
case = move-n_x_*
  then i-move-n_x_*-okp (i)
case = move_<tsp>_*
  then i-move_<tsp>_*-okp (i)
case = move_<x{a}>_<tsp>
  then i-move_<x{a}>_<tsp>-okp (i)
case = move_<x{s}>_<tsp>
  then i-move_<x{s}>_<tsp>-okp (i)
case = move_cfp_csp
  then i-move_cfp_csp-okp (i)
case = move_csp_cfp
  then i-move_csp_cfp-okp (i)
case = move_x_*
  then i-move_x_*-okp (i)
case = move_x_<x{s}>
  then i-move_x_<x{s}>-okp (i)
case = move_x_tsp
  then i-move_x_tsp-okp (i)
case = move_x_x
  then i-move_x_x-okp (i)
case = move_y_*
  then i-move_y_*-okp (i)
case = move_y_<y{s}>
  then i-move_y_<y{s}>-okp (i)
case = move_y_tsp
  then i-move_y_tsp-okp (i)
case = neg_<tsp>_<tsp>{i}
  then i-neg_<tsp>_<tsp>{i}-okp (i)
case = not_<tsp>_<tsp>{v}
  then i-not_<tsp>_<tsp>{v}-okp (i)
case = or_<tsp>{v}_x{v}
  then i-or_<tsp>{v}_x{v}-okp (i)
case = or_<tsp>{b}_x{b}
  then i-or_<tsp>{b}_x{b}-okp (i)
case = sub_<c>_<tsp>{a}_x{a}
  then i-sub_<c>_<tsp>{a}_x{a}-okp (i)
case = sub_<c>_<tsp>{n}_x{n}
  then i-sub_<c>_<tsp>{n}_x{n}-okp (i)
case = sub_<nv>_<tsp>{i}_x{i}
  then i-sub_<nv>_<tsp>{i}_x{i}-okp (i)
```

case = *sub_<tsp>{a}_x{n}*
then i-sub_<tsp>{a}_x{n}-okp (i)
case = *sub_x{s}_y{n}*
then i-sub_x{s}_y{n}-okp (i)
case = *sub_<tsp>{i}_x{i}*
then i-sub_<tsp>{i}_x{i}-okp (i)
case = *sub_<tsp>{n}_x{n}*
then i-sub_<tsp>{n}_x{n}-okp (i)
case = *sub_<tsp>{s}_x{s}*
then i-sub_<tsp>{s}_x{s}-okp (i)
case = *sub_<z>_x{s}_y{s}*
then i-sub_<z>_x{s}_y{s}-okp (i)
case = *subb_<c>_x{n}_y{n}*
then i-subb_<c>_x{n}_y{n}-okp (i)
case = *subb_<v>_x{i}_y{i}*
then i-subb_<v>_x{i}_y{i}-okp (i)
case = *tpop_<c>_x*
then i-tpop_<c>_x-okp (i)
case = *tpop_<x{a}>*
then i-tpop_<x{a}>-okp (i)
case = *tpop_<x{s}>*
then i-tpop_<x{s}>-okp (i)
case = *tpop_pc*
then i-tpop_pc-okp (i)
case = *tpop_x*
then i-tpop_x-okp (i)
case = *tpop_y*
then i-tpop_y-okp (i)
case = *tpop{v}_<z>_y*
then i-tpop{v}_<z>_y-okp (i)
case = *tpop{b}_<z>_y*
then i-tpop{b}_<z>_y-okp (i)
case = *tpop{i}_<zn>_y*
then i-tpop{i}_<zn>_y-okp (i)
case = *tpop{n}_<z>_y*
then i-tpop{n}_<z>_y-okp (i)
case = *tpush_**
then i-tpush_*-okp (i)
case = *tpush_<x{a}>*
then i-tpush_<x{a}>-okp (i)
case = *tpush_<x{s}>*
then i-tpush_<x{s}>-okp (i)
case = *tpush_csp*
then i-tpush_csp-okp (i)

case = *tpush_tsp*
then i-tpush_tsp-okp (*i*)
case = *tpush_x*
then i-tpush_x-okp (*i*)
case = *xor_<tsp>_<tsp>*
then i-xor_<tsp>_<tsp>-okp (*i*)
case = *xor_<tsp>{v}_x{v}*
then i-xor_<tsp>{v}_x{v}-okp (*i*)
case = *xor_<tsp>{b}_*{b}*
then i-xor_<tsp>{b}_*{b}-okp (*i*)
case = *xor_<tsp>{b}_x{b}*
then i-xor_<tsp>{b}_x{b}-okp (*i*)
case = *xor_<z>_<tsp>_x*
then i-xor_<z>_<tsp>_x-okp (*i*)
otherwise f endcase)

DEFINITION:
i-ins-step (*ins*, *i*, *load-addr*)
= **case on** car (*ins*):
case = *add_<c>_x_x{n}*
then i-add_<c>_x_x{n}-step (*i*)
case = *add_<tsp>_<tsp>{v}*
then i-add_<tsp>_<tsp>{v}-step (*i*)
case = *add_<tsp>_<tsp>{n}*
then i-add_<tsp>_<tsp>{n}-step (*i*)
case = *add_<tsp>{a}_x{n}*
then i-add_<tsp>{a}_x{n}-step (*i*)
case = *add_tsp_*{n}*
then i-add_tsp_*{n}-step (*i*)
case = *add_tsp_x{n}*
then i-add_tsp_x{n}-step (*i*)
case = *add_<tsp>{i}_x{i}*
then i-add_<tsp>{i}_x{i}-step (*i*)
case = *add_<tsp>{n}_x{n}*
then i-add_<tsp>{n}_x{n}-step (*i*)
case = *add_pc_x{n}*
then i-add_pc_x{n}-step (*i*)
case = *add_x_x{n}*
then i-add_x_x{n}-step (*i*)
case = *add_x{n}_csp*
then i-add_x{n}_csp-step (*i*)
case = *addc_<c>_x{n}_y{n}*
then i-addc_<c>_x{n}_y{n}-step (*i*)
case = *addc_<v>_x{i}_y{i}*

then i-addc_<v>_x{i}_y{i}-step (i)
case = *and_<tsp>{v}_x{v}*
then i-and_<tsp>{v}_x{v}-step (i)
case = *and_<tsp>{b}_x{b}*
then i-and_<tsp>{b}_x{b}-step (i)
case = *asr_<c>_<tsp>_<tsp>{b}*
then i-asr_<c>_<tsp>_<tsp>{b}-step (i)
case = *cpop_cfp*
then i-cpop_cfp-step (i)
case = *cpop_pc*
then i-cpop_pc-step (i)
case = *cpush_**
then i-cpush_*-step (i)
case = *cpush_<tsp>+*
then i-cpush_<tsp>+-step (i)
case = *cpush_cfp*
then i-cpush_cfp-step (i)
case = *decr_<tsp>_<tsp>{i}*
then i-decr_<tsp>_<tsp>{i}-step (i)
case = *decr_<tsp>_<tsp>{n}*
then i-decr_<tsp>_<tsp>{n}-step (i)
case = *incr_<tsp>_<tsp>{i}*
then i-incr_<tsp>_<tsp>{i}-step (i)
case = *incr_<tsp>_<tsp>{n}*
then i-incr_<tsp>_<tsp>{n}-step (i)
case = *incr_y_y{n}*
then i-incr_y_y{n}-step (i)
case = *int-to-nat*
then i-int-to-nat-step (i)
case = *jump-n_x*
then i-jump-n_x-step (i)
case = *jump-nn_x*
then i-jump-nn_x-step (i)
case = *jump-nz_x*
then i-jump-nz_x-step (i)
case = *jump-z_x*
then i-jump-z_x-step (i)
case = *jump_**
then i-jump_*-step (i)
case = *jump_x{subr}*
then i-jump_x{subr}-step (i)
case = *lsr_<c>_x_x{n}*
then i-lsr_<c>_x_x{n}-step (i)
case = *lsr_<tsp>_<tsp>{v}*

then i-lsr_<tsp>_<tsp>{v}-step (i)
case = move-c_<tsp>_*
then i-move-c_<tsp>_*-step (i)
case = move-v_<tsp>_*
then i-move-v_<tsp>_*-step (i)
case = move-z_<tsp>_*
then i-move-z_<tsp>_*-step (i)
case = move-n_x_*
then i-move-n_x_*-step (i)
case = move_<tsp>_*
then i-move_<tsp>_*-step (i)
case = move_<x{a}>_<tsp>
then i-move_<x{a}>_<tsp>-step (i)
case = move_<x{s}>_<tsp>
then i-move_<x{s}>_<tsp>-step (i)
case = move_cfp_csp
then i-move_cfp_csp-step (i)
case = move_csp_cfp
then i-move_csp_cfp-step (i)
case = move_x_*
then i-move_x_*-step (i)
case = move_x_<x{s}>
then i-move_x_<x{s}>-step (i)
case = move_x_tsp
then i-move_x_tsp-step (i)
case = move_x_x
then i-move_x_x-step (i)
case = move_y_*
then i-move_y_*-step (i)
case = move_y_<y{s}>
then i-move_y_<y{s}>-step (i)
case = move_y_tsp
then i-move_y_tsp-step (i)
case = neg_<tsp>_<tsp>{i}
then i-neg_<tsp>_<tsp>{i}-step (i)
case = not_<tsp>_<tsp>{v}
then i-not_<tsp>_<tsp>{v}-step (i)
case = or_<tsp>{v}_x{v}
then i-or_<tsp>{v}_x{v}-step (i)
case = or_<tsp>{b}_x{b}
then i-or_<tsp>{b}_x{b}-step (i)
case = sub_<c>_<tsp>{a}_x{a}
then i-sub_<c>_<tsp>{a}_x{a}-step (i)
case = sub_<c>_<tsp>{n}_x{n}

then i-sub_<c>_<tsp>{n}_x{n}-step (*i*)
case = *sub_<nv>_<tsp>{i}_x{i}*
then i-sub_<nv>_<tsp>{i}_x{i}-step (*i*)
case = *sub_<tsp>{a}_x{n}*
then i-sub_<tsp>{a}_x{n}-step (*i*)
case = *sub_x{s}_y{n}*
then i-sub_x{s}_y{n}-step (*i*)
case = *sub_<tsp>{i}_x{i}*
then i-sub_<tsp>{i}_x{i}-step (*i*)
case = *sub_<tsp>{n}_x{n}*
then i-sub_<tsp>{n}_x{n}-step (*i*)
case = *sub_<tsp>{s}_x{s}*
then i-sub_<tsp>{s}_x{s}-step (*i*)
case = *sub_<z>_x{s}_y{s}*
then i-sub_<z>_x{s}_y{s}-step (*i*)
case = *subb_<c>_x{n}_y{n}*
then i-subb_<c>_x{n}_y{n}-step (*i*)
case = *subb_<v>_x{i}_y{i}*
then i-subb_<v>_x{i}_y{i}-step (*i*)
case = *tpop_<c>_x*
then i-tpop_<c>_x-step (*i*)
case = *tpop_<x{a}>*
then i-tpop_<x{a}>-step (*i*)
case = *tpop_<x{s}>*
then i-tpop_<x{s}>-step (*i*)
case = *tpop_pc*
then i-tpop_pc-step (*i*)
case = *tpop_x*
then i-tpop_x-step (*i*)
case = *tpop_y*
then i-tpop_y-step (*i*)
case = *tpop{v}_<z>_y*
then i-tpop{v}_<z>_y-step (*i*)
case = *tpop{b}_<z>_y*
then i-tpop{b}_<z>_y-step (*i*)
case = *tpop{i}_<zn>_y*
then i-tpop{i}_<zn>_y-step (*i*)
case = *tpop{n}_<z>_y*
then i-tpop{n}_<z>_y-step (*i*)
case = *tpush_**
then i-tpush_*-step (*i*)
case = *tpush_<x{a}>*
then i-tpush_<x{a}>-step (*i*)
case = *tpush_<x{s}>*

then i-tpush_<x{s}>-step (*i*)
case = *tpush_csp*
then i-tpush_csp-step (*i*)
case = *tpush_tsp*
then i-tpush_tsp-step (*i*)
case = *tpush_x*
then i-tpush_x-step (*i*)
case = *xor_<tsp>_<tsp>*
then i-xor_<tsp>_<tsp>-step (*i*)
case = *xor_<tsp>{v}_x{v}*
then i-xor_<tsp>{v}_x{v}-step (*i*)
case = *xor_<tsp>{b}_*{b}*
then i-xor_<tsp>{b}_*{b}-step (*i*)
case = *xor_<tsp>{b}_x{b}*
then i-xor_<tsp>{b}_x{b}-step (*i*)
case = *xor_<z>_<tsp>_x*
then i-xor_<z>_<tsp>_x-step (*i*, *load-addr*)
otherwise i-halt (*i*, `'run`) **endcase**

DEFINITION:
i-step1 (*ins*, *i*, *load-addr*)
= **if** i-state-okp (*i*, *load-addr*)
then if i-ins-okp (*ins*, *i*) **then** i-ins-step (*ins*, *i*, *load-addr*)
else i-halt (*i*, x-y-error-msg (`'i`, car (*ins*))) **endif**
else i-halt (*i*, `'state-not-okp`) **endif**

DEFINITION:
i-step (*i*, *load-addr*)
= **if** i-psw (*i*) = `'run`
then i-step1 (i-current-instruction (*i*), *i*, *load-addr*)
else *i* **endif**

DEFINITION:
i (*i*, *n*, *load-addr*)
= **if** $n \simeq$ `0` **then** *i*
else i (i-step (*i*, *load-addr*), $n - 1$, *load-addr*) **endif**

DEFINITION:
list-of-nats (*lst*)
= **if** *lst* $\simeq$ **nil then nil**
else cons (v-to-nat (car (*lst*)), list-of-nats (cdr (*lst*))) **endif**

```
; Book: i-m.events.  We now develop the proof that the i machine is
; implemented on the m machine.
```

```
; I first develop the necessary rules about the FM9001 alu.

; Each of the alu theorems is about the bv-alu-cv function.
; That function returns a triple.  We are most interested in
; the bv and c components of the triple.  We are occasionally
; interested in the v component, but only for those instructions
; that we use that set the v flag.

; For each opcode op, each component, cmp, in {bv c v}
; and type, t, in {bitv nat tc} we have a theorem called
; alu-thm_t-cmp-op.  For example, we have alu-thm_BITV-BV-MOVE,
; which gives the BITV interpretation of the BV component of
; the alu on a MOVE opcode.  We omit some combinations because
; they do not make much sense.  Occasionally we have to prove
; conflicting rules -- rules that rewrite the same lhs.  This
; happens when the opcode has a natural interpretation for more
; than one type.  When it happens we note it.
```

EVENT: Enable v-buf-works.

THEOREM: alu-thm_bitv-bv-move-15
$(\text{bvp}\,(a) \land \text{bvp}\,(b) \land (\text{length}\,(a) = \text{length}\,(b)) \land \text{boolp}\,(c))$
$\rightarrow \quad (\text{bv}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{t},\ \mathbf{t},\ \mathbf{t},\ \mathbf{t}))) = a)$

THEOREM: alu-thm_bitv-c-move-15
$(\text{bvp}\,(a) \land \text{bvp}\,(b) \land (\text{length}\,(a) = \text{length}\,(b)) \land \text{boolp}\,(c))$
$\rightarrow \quad (\text{c}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{t},\ \mathbf{t},\ \mathbf{t},\ \mathbf{t}))) = \mathbf{f})$

THEOREM: alu-thm_bitv-bv-not
$(\text{bvp}\,(a) \land \text{bvp}\,(b) \land (\text{length}\,(a) = \text{length}\,(b)) \land \text{boolp}\,(c))$
$\rightarrow \quad (\text{bv}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{f},\ \mathbf{t},\ \mathbf{t},\ \mathbf{t}))) = \text{v-not}\,(a))$

THEOREM: alu-thm_bitv-c-not
$(\text{bvp}\,(a) \land \text{bvp}\,(b) \land (\text{length}\,(a) = \text{length}\,(b)) \land \text{boolp}\,(c))$
$\rightarrow \quad (\text{c}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{f},\ \mathbf{t},\ \mathbf{t},\ \mathbf{t}))) = \mathbf{f})$

THEOREM: alu-thm_bitv-bv-and
$(\text{bvp}\,(a) \land \text{bvp}\,(b) \land (\text{length}\,(a) = \text{length}\,(b)) \land \text{boolp}\,(c))$
$\rightarrow \quad (\text{bv}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{t},\ \mathbf{f},\ \mathbf{t},\ \mathbf{t}))) = \text{v-and}\,(a,\ b))$

THEOREM: alu-thm_bitv-c-and
$(\text{bvp}\,(a) \land \text{bvp}\,(b) \land (\text{length}\,(a) = \text{length}\,(b)) \land \text{boolp}\,(c))$
$\rightarrow \quad (\text{c}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{t},\ \mathbf{f},\ \mathbf{t},\ \mathbf{t}))) = \mathbf{f})$

THEOREM: alu-thm_bitv-bv-or
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{f},\ \mathbf{t},\ \mathbf{t}))) = \text{v-or}(a,\ b))$

THEOREM: alu-thm_bitv-c-or
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{f},\ \mathbf{t},\ \mathbf{t}))) = \mathbf{f})$

THEOREM: alu-thm_bitv-bv-xor
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{t},\ \mathbf{f},\ \mathbf{t}))) = \text{v-xor}(a,\ b))$

THEOREM: alu-thm_bitv-c-xor
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{t},\ \mathbf{f},\ \mathbf{t}))) = \mathbf{f})$

```
; Conflict:  We will have both a BITV and a NAT interpretation of the
; BV and C produced by the LSR instruction.  Here is the BITV case:
```

THEOREM: alu-thm_bitv-bv-lsr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{t},\ \mathbf{f},\ \mathbf{t}))) = \text{v-lsr}(a))$

```
; Note that we have an additional hypothesis below, (listp a).
```

THEOREM: alu-thm_bitv-c-lsr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{listp}(a) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{t},\ \mathbf{f},\ \mathbf{t}))) = \text{bitn}(a,\ 1))$

```
; Conflict:  We will have both a BITV and a TC interpretation of the BV
; produced by the ASR instruction.
```

THEOREM: alu-thm_bitv-bv-asr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{f},\ \mathbf{f},\ \mathbf{t}))) = \text{v-asr}(a))$

```
; Note that we have an additional hypothesis below, (listp a).
```

THEOREM: alu-thm_bitv-c-asr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{listp}(a) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{f},\ \mathbf{f},\ \mathbf{t}))) = \text{bitn}(a,\ 1))$

THEOREM: alu-thm_bitv-bv-ror
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{f},\ \mathbf{f},\ \mathbf{t}))) = \text{v-ror}(a,\ c))$

```
; To prove the next one we need another piece of help.
```

THEOREM: boolp-truep
$\text{boolp}(c) \rightarrow (\text{truep}(c) = c)$

THEOREM: alu-thm_bitv-c-ror
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{f},\ \mathbf{f},\ \mathbf{t})))$
$= \quad \textbf{if } \text{length}(a) \simeq 0 \textbf{ then } c$
$\quad \textbf{else } \text{bitn}(a,\ 1) \textbf{ endif})$

```
; Here are the natural number interpretations of the bv and c flag of the
; alu.

; Our first step is to get the natural interpretation lemma as a rewrite
; rule and shut down v-alu:
```

EVENT: Disable v-alu.

THEOREM: v-alu-correct-nat-rewriter
$\text{bv2p}(a,\ b) \rightarrow (\text{v-alu}(c,\ a,\ b,\ op) = \text{v-alu-nat}(c,\ a,\ b,\ op))$

```
; The original (FM8502) Piton proofs were based on alu interpretation
; lemmas proved for FM8502.  When FM9001 was done, Matt Kaufmann was given
; a free hand to formulate the alu interpretation lemmas for it.  His
; algebraic expressions were different than those used for FM8502, even
; though they were equivalent.  Rather than redesign the Piton proof to
; use Matt's formulation, we take the approach of proving the old-style
; algebraic formulations from Matt's.  That is what we do now.

; This gets exceptionally tedious for the twos-complement stuff.  I essentially
; just use brute force to wade through this stuff.  The general scheme was
; was to attack the top-level theorems with a fixed set of rules enabled,
; generating certain goals (all phrased in integer terms) and then attack
; each of those goals as a separate lemma with the integer stuff enabled.
; Those integer lemmas are killers, taking hundreds or thousands of seconds
; each.  But it was better than thinking...
```

THEOREM: alu-thm_nat-bv-lsr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f}, \mathbf{t}, \mathbf{f}, \mathbf{t})))$
$\quad = \quad \text{nat-to-v}(\text{v-to-nat}(a) \div 2, \text{length}(a)))$

THEOREM: alu-thm_nat-c-lsr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f}, \mathbf{t}, \mathbf{f}, \mathbf{t}))) = ((\text{v-to-nat}(a) \ \mathbf{mod}\ 2) \not\simeq 0))$

EVENT: Enable plus-0.

THEOREM: pathological-difference
$(a \not< b) \rightarrow ((b - a) = 0)$

THEOREM: difference-difference-plus
$(((b + e) - a) - e) = (b - a)$

EVENT: Enable lessp-v-to-nat-exp.

THEOREM: remainder-difference-hack
$(b < e) \rightarrow (((b - a) \ \mathbf{mod}\ e) = (b - a))$

THEOREM: difference-difference-hack
$(a \not< b) \rightarrow ((e - (a - b)) = ((b + e) - a))$

```
; Conflict:  We will have both a NAT and a TC interpretation of the BV
; produced by the SUB instruction.
```

THEOREM: alu-thm_nat-bv-sub
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t}, \mathbf{t}, \mathbf{t}, \mathbf{f})))$
$\quad = \quad \mathbf{if}\ \text{v-to-nat}(b) \not< \text{v-to-nat}(a)$
$\quad\quad \mathbf{then}\ \text{nat-to-v}(\text{v-to-nat}(b) - \text{v-to-nat}(a), \text{length}(a))$
$\quad\quad \mathbf{else}\ \text{nat-to-v}(\text{exp}(2, \text{length}(a))$
$\quad\quad\quad\quad - \quad (\text{v-to-nat}(a) - \text{v-to-nat}(b)),$
$\quad\quad\quad\quad \text{length}(a))\ \mathbf{endif})$

THEOREM: alu-thm_nat-c-sub
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t}, \mathbf{t}, \mathbf{t}, \mathbf{f}))) = (\text{v-to-nat}(b) < \text{v-to-nat}(a)))$

DEFINITION:
carry (c)
$= \quad \mathbf{if}\ c\ \mathbf{then}\ 1$
$\quad\ \mathbf{else}\ 0\ \mathbf{endif}$

```
; Conflict:  We will have both a NAT and a TC interpretation of the the
; BV produced by the SUBB instruction.
```

THEOREM: alu-thm_nat-bv-subb
(bvp (a) $\wedge$ bvp (b) $\wedge$ (length (a) = length (b)) $\wedge$ boolp (c))
$\rightarrow$ (bv (v-alu (c, a, b, list (**f**, **t**, **t**, **f**)))
= **if** v-to-nat (b) < (v-to-nat (a) + carry (c))
then nat-to-v (exp (2, length (a))
− ((v-to-nat (a) + carry (c))
− v-to-nat (b)),
length (a))
else nat-to-v (v-to-nat (b)
− (v-to-nat (a) + carry (c)),
length (a)) **endif**)

THEOREM: alu-thm_nat-c-subb
(bvp (a) $\wedge$ bvp (b) $\wedge$ (length (a) = length (b)) $\wedge$ boolp (c))
$\rightarrow$ (c (v-alu (c, a, b, list (**f**, **t**, **t**, **f**)))
= (v-to-nat (b) < (v-to-nat (a) + carry (c))))

EVENT: Disable pathological-difference.

EVENT: Disable difference-difference-plus.

EVENT: Disable lessp-v-to-nat-exp.

EVENT: Disable remainder-difference-hack.

EVENT: Disable difference-difference-hack.

EVENT: Enable length-nat-to-v.

EVENT: Enable commutativity-of-plus.

EVENT: Enable associativity-of-plus.

EVENT: Enable plus-add1.

EVENT: Enable plus-0.

EVENT: Enable times-commutes.

EVENT: Enable associativity-of-times.

EVENT: Enable times-distributes-over-plus.

EVENT: Enable times-add1-again.

EVENT: Enable times-1.

THEOREM: plus-commutes2
$(x + (y + z)) = (y + (x + z))$

THEOREM: times-commutes2
$(x * (y * z)) = (y * (x * z))$

THEOREM: remainder-lessp
$((x \textbf{ mod } y) < y) = (y \not\simeq \mathsf{0})$

EVENT: Enable remainder-quotient-elim.

THEOREM: remainder-quotient-rewrite
$((x \in \mathbf{N}) \land (y \not\simeq \mathsf{0})) \rightarrow (((x \textbf{ mod } y) + (y * (x \div y))) = x)$

THEOREM: v-to-nat-nat-to-v
$(n < \exp(\mathsf{2}, \mathit{len})) \rightarrow (\text{v-to-nat}(\text{nat-to-v}(n, \mathit{len})) = \text{fix}(n))$

THEOREM: lessp-0-exp
$(i \not\simeq \mathsf{0}) \rightarrow (\mathsf{0} < \exp(i, j))$

THEOREM: difference-plus
$((a + b) - b) = \text{fix}(a)$

```
; The following lemma is a way to get linear arithmetic to address the
; question "is a less than b" when opening up (remainder a b).
```

THEOREM: remainder-opener
$(a < b) \rightarrow ((a \textbf{ mod } b) = \text{fix}(a))$

EVENT: Enable lessp-v-to-nat-exp.

THEOREM: lessp-v-to-nat-exp-gen
$\text{v-to-nat}(a) < \exp(2, \text{length}(a))$

THEOREM: remainder-hack2
$((\text{v-to-nat}(v) - 1) \ \mathbf{mod} \ \exp(2, \text{length}(v))) = (\text{v-to-nat}(v) - 1)$

EVENT: Enable difference-x-1.

```
; Conflict:  We will have both a NAT and a TC interpretation of the the
; BV produced by the DECR instruction.
```

THEOREM: alu-thm_nat-bv-decr
$$\begin{array}{l} (\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c)) \\ \to \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{f},\ \mathbf{t},\ \mathbf{f}))) \\ \qquad = \quad \mathbf{if}\ \text{v-to-nat}(a) \simeq 0 \\ \qquad\qquad \mathbf{then}\ \text{nat-to-v}(\exp(2, \text{length}(a)) - 1, \text{length}(a)) \\ \qquad\qquad \mathbf{else}\ \text{nat-to-v}(\text{v-to-nat}(a) - 1, \text{length}(a))\ \mathbf{endif}) \end{array}$$

THEOREM: alu-thm_nat-c-decr
$$\begin{array}{l} (\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c)) \\ \to \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{f},\ \mathbf{t},\ \mathbf{f}))) = (\text{v-to-nat}(a) \simeq 0)) \end{array}$$

```
; Conflict:  We will have both a NAT and a TC interpretation of the the
; BV produced by the ADD instruction.
```

THEOREM: alu-thm_nat-bv-add
$$\begin{array}{l} (\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c)) \\ \to \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{t},\ \mathbf{f},\ \mathbf{f}))) \\ \qquad = \quad \mathbf{if}\ (\text{v-to-nat}(a) + \text{v-to-nat}(b)) < \exp(2, \text{length}(a)) \\ \qquad\qquad \mathbf{then}\ \text{nat-to-v}(\text{v-to-nat}(a) + \text{v-to-nat}(b), \text{length}(a)) \\ \qquad\qquad \mathbf{else}\ \text{nat-to-v}((\text{v-to-nat}(a) + \text{v-to-nat}(b)) \\ \qquad\qquad\qquad\qquad \mathbf{mod}\quad \exp(2, \text{length}(a)), \\ \qquad\qquad\qquad\qquad \text{length}(a))\ \mathbf{endif}) \end{array}$$

THEOREM: alu-thm_nat-c-add
$$\begin{array}{l} (\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c)) \\ \to \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{t},\ \mathbf{f},\ \mathbf{f}))) \\ \qquad = \quad ((\text{v-to-nat}(a) + \text{v-to-nat}(b)) \not< \exp(2, \text{length}(a)))) \end{array}$$

```
; Conflict:  We will have both a NAT and a TC interpretation of the the
; BV produced by the ADDC instruction.
```

THEOREM: alu-thm_nat-bv-addc
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{t},\ \mathbf{f},\ \mathbf{f})))$
$\quad = \quad \textbf{if}\ (\text{v-to-nat}(a) + \text{v-to-nat}(b) + \text{carry}(c))$
$\qquad < \quad \exp(2,\ \text{length}(a))$
$\qquad \textbf{then}\ \text{nat-to-v}(\text{v-to-nat}(a) + \text{v-to-nat}(b) + \text{carry}(c),$
$\qquad\qquad \text{length}(a))$
$\qquad \textbf{else}\ \text{nat-to-v}((\text{v-to-nat}(a) + \text{v-to-nat}(b) + \text{carry}(c))$
$\qquad\qquad \textbf{mod} \quad \exp(2,\ \text{length}(a)),$
$\qquad\qquad \text{length}(a))\ \textbf{endif})$

THEOREM: alu-thm_nat-c-addc
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{f},\ \mathbf{t},\ \mathbf{f},\ \mathbf{f})))$
$\quad = \quad ((\text{v-to-nat}(a) + \text{v-to-nat}(b) + \text{carry}(c))$
$\qquad \not< \quad \exp(2,\ \text{length}(a))))$

```
; Conflict:  We will have both a NAT and a TC interpretation of the the
; BV produced by the INCR instruction.
```

THEOREM: alu-thm_nat-bv-incr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{bv}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{f},\ \mathbf{f},\ \mathbf{f})))$
$\quad = \quad \textbf{if}\ (1 + \text{v-to-nat}(a)) < \exp(2,\ \text{length}(a))$
$\qquad \textbf{then}\ \text{nat-to-v}(1 + \text{v-to-nat}(a),\ \text{length}(a))$
$\qquad \textbf{else}\ \text{nat-to-v}(0,\ \text{length}(a))\ \textbf{endif})$

THEOREM: alu-thm_nat-c-incr
$(\text{bvp}(a) \land \text{bvp}(b) \land (\text{length}(a) = \text{length}(b)) \land \text{boolp}(c))$
$\rightarrow \quad (\text{c}(\text{v-alu}(c,\ a,\ b,\ \text{list}(\mathbf{t},\ \mathbf{f},\ \mathbf{f},\ \mathbf{f})))$
$\quad = \quad ((1 + \text{v-to-nat}(a)) \not< \exp(2,\ \text{length}(a))))$

```
; Conflict:  Each of the TC interpretations conflicts with some other rule.

; Warning: all of these tc guys had (hands-off if) hints!
```

EVENT: Disable v-alu-correct-nat-rewriter.

THEOREM: v-alu-correct-int-rewriter
$(\text{bv2p}(a,\ b) \land \text{listp}(a)) \rightarrow (\text{v-alu}(c,\ a,\ b,\ op) = \text{v-alu-int}(c,\ a,\ b,\ op))$

DEFINITION:
$\text{mod2}(x)$
$= \quad \textbf{if}\ \text{negativep}(x)\ \ \textbf{then}\ -((1 + \text{negative-guts}(x)) \div 2)$
$\qquad \textbf{else}\ x \div 2\ \textbf{endif}$

EVENT: Enable quotient-plus-x-x-2.

```
; A lemma quotient-add1-plus-x-x-2 exists in fm9001.events, but it did
; not originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-quotient-add1-plus-x-x-2
$((1 + (x + x)) \div 2) = \text{fix}(x)$

THEOREM: idiv-is-mod2
idiv $(x, 2)$
$=$ **if** integerp (x) **then** mod2 (x)
else 0 **endif**

EVENT: Disable idiv.

```
; A lemma integerp-v-to-int exists in fm9001.events, but it did not
; originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-integerp-v-to-int
integerp (v-to-int (v))

THEOREM: alu-thm_tc-bv-asr
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (bv (v-alu $(c,\ a,\ b,$ list (**t**, **f**, **f**, **t**)))
$=$ int-to-v (mod2 (v-to-int (a)), length (a)))

DEFINITION:
tc-in-rangep $(x,\ n)$
$=$ **if** $n \simeq 0$ **then f**
elseif negativep (x) **then** exp $(2,\ n - 1) \not< $ negative-guts (x)
else $x <$ exp $(2,\ n - 1)$ **endif**

DEFINITION: tc-add $(x,\ y)$ = iplus $(x,\ y)$

DEFINITION: tc-minus (x) = ineg (x)

THEOREM: iplus-0
$\text{iplus}\,(x,\ 0) = \text{fix-int}\,(x)$

THEOREM: equal-length-0
$(\text{length}\,(x) = 0) = (x \simeq \textbf{nil})$

THEOREM: negativep-ineg
$((x \in \mathbf{N}) \land (x \neq 0)) \rightarrow \text{negativep}\,(\text{ineg}\,(x))$

THEOREM: not-equal-exp-0
$(x \not\simeq 0) \rightarrow (\exp\,(x,\ y) \neq 0)$

THEOREM: ineg-exp-2-x
$\text{ineg}\,(\exp\,(2,\ x)) = (-\ \exp\,(2,\ x))$

THEOREM: diff-diff
$$
\begin{array}{l}
((a \in \mathbf{N}) \land (b \in \mathbf{N})) \\
\rightarrow \quad ((a - (a - b)) \\
\qquad = \quad \textbf{if}\ a < b\ \ \textbf{then}\ a \\
\qquad\qquad \textbf{else}\ b\ \textbf{endif})
\end{array}
$$

THEOREM: alu-thm_tc-bv-sub-lemma
$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \land \quad \text{listp}\,(a) \\
\ \land \quad \text{bvp}\,(b) \\
\ \land \quad \text{listp}\,(b) \\
\ \land \quad (\text{length}\,(a) = \text{length}\,(b)) \\
\ \land \quad (\neg\ \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(b),\ \text{ineg}\,(\text{v-to-int}\,(a))))) \\
\ \land \quad (\text{iplus}\,(\text{v-to-int}\,(b),\ \text{ineg}\,(\text{v-to-int}\,(a))) \not< \exp\,(2,\ \text{length}\,(a)\ - 1))) \\
\rightarrow \quad (\text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(b), \\
\qquad\qquad\qquad\qquad \text{iplus}\,(-\ \exp\,(2,\ \text{length}\,(a)),\ \text{ineg}\,(\text{v-to-int}\,(a)))), \\
\qquad\qquad\qquad \text{length}\,(a)) \\
\qquad = \quad \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(b),\ \text{ineg}\,(\text{v-to-int}\,(a))),\ \text{length}\,(a)))
\end{array}
$$

```
; Note: This was proved with the lower case a's replaced by b's, but I
; have now assumed it as shown below (except that the concluding
; equality was reversed).  Under that assumption, I can prove the
; tc-bv-sub lemma I need.  However, I now want to reverse the
; direction of the concluding equality so as not to take the chance
; that it loops.
```

THEOREM: alu-thm_tc-bv-sub-lemma2
$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \land \quad \text{listp}\,(a)
\end{array}
$$

$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (iplus (v-to-int (b), ineg (v-to-int (a))))
$\wedge$ (exp (2, length (a) $-$ 1)
  $<$ negative-guts (iplus (v-to-int (b), ineg (v-to-int (a))))))
$\rightarrow$ (int-to-v (iplus (v-to-int (b),
      iplus (exp (2, length (a)), ineg (v-to-int (a)))),
    length (a))
  = int-to-v (iplus (v-to-int (b), ineg (v-to-int (a))), length (a)))

THEOREM: alu-thm_tc-bv-sub
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (bv (v-alu (c, a, b, list (**t**, **t**, **t**, **f**)))
  = int-to-v (**if** tc-in-rangep (tc-add (v-to-int (b),
        tc-minus (v-to-int (a))),
      length (a))
    then tc-add (v-to-int (b), tc-minus (v-to-int (a)))
    elseif negativep (tc-add (v-to-int (b),
        tc-minus (v-to-int (a))))
    then tc-add (v-to-int (b),
      tc-add (exp (2, length (a)),
        tc-minus (v-to-int (a))))
    else tc-add (v-to-int (b),
      tc-add ($-$ exp (2, length (a)),
        tc-minus (v-to-int (a)))) **endif**,
    length (a)))

THEOREM: alu-thm_tc-v-sub
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (v (v-alu (c, a, b, list (**t**, **t**, **t**, **f**)))
  = ($\neg$ tc-in-rangep (tc-add (v-to-int (b), tc-minus (v-to-int (a))),
      length (a))))

THEOREM: diff-plus-cancellation-hack2

$((a + b + c + d) - (b + c)) = (a + d)$

THEOREM: diff-plus-sub1-hack1
$(a < e) \rightarrow (((b + (e - 1)) - a) = (b + ((e - a) - 1)))$

THEOREM: alu-thm_tc-bv-subb-lemma1

$$
\begin{array}{ll}
(\text{bvp}\,(a) & \\
\land & \text{listp}\,(a) \\
\land & \text{bvp}\,(b) \\
\land & \text{listp}\,(b) \\
\land & (\text{length}\,(a) = \text{length}\,(b)) \\
\land & (\neg\ \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(b),\ \text{ineg}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ 1))))) \\
\land & (\text{iplus}\,(\text{v-to-int}\,(b),\ \text{ineg}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ 1))) \\
& \quad \not< \quad \text{exp}\,(2,\ \text{length}\,(a) - 1))) \\
\rightarrow & (\text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(b), \\
& \qquad\qquad \text{iplus}\,(-\ \text{exp}\,(2,\ \text{length}\,(a)), \\
& \qquad\qquad\qquad \text{ineg}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ 1)))), \\
& \qquad \text{length}\,(a)) \\
& \quad = \quad \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(b),\ \text{ineg}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ 1))), \\
& \qquad\qquad \text{length}\,(a)))
\end{array}
$$

THEOREM: diff-plus-sub1-hack2

$$
\begin{array}{ll}
((e \not\simeq 0) \land (a < e)) & \\
\rightarrow & (((b + ((e - a) - 1)) - e) = ((b + (e - 1)) - (a + e)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-subb-lemma2-lemma1

$$
\begin{array}{ll}
(\text{bvp}\,(a) \land (\text{length}\,(a) \neq 0)) & \\
\rightarrow & (\text{nat-to-v}\,((\text{v-to-nat}\,(b) \\
& \qquad + \quad (((\text{exp}\,(2,\ \text{length}\,(a) - 1) + \text{exp}\,(2,\ \text{length}\,(a) - 1)) \\
& \qquad\qquad - \quad \text{v-to-nat}\,(a)) - 1)) \\
& \qquad - \quad (\text{exp}\,(2,\ \text{length}\,(a) - 1) + \text{exp}\,(2,\ \text{length}\,(a) - 1)), \\
& \qquad \text{length}\,(a)) \\
& \quad = \quad \text{nat-to-v}\,((\text{v-to-nat}\,(b) \\
& \qquad\qquad + \quad ((\text{exp}\,(2,\ \text{length}\,(a) - 1) \\
& \qquad\qquad\qquad + \quad \text{exp}\,(2,\ \text{length}\,(a) - 1)) - 1)) \\
& \qquad\qquad - \quad (\text{v-to-nat}\,(a) \\
& \qquad\qquad\qquad + \quad \text{exp}\,(2,\ \text{length}\,(a) - 1) \\
& \qquad\qquad\qquad + \quad \text{exp}\,(2,\ \text{length}\,(a) - 1)), \\
& \qquad\qquad \text{length}\,(a)))
\end{array}
$$

EVENT: Disable diff-plus-sub1-hack2.

THEOREM: alu-thm_tc-bv-subb-lemma2
$(\text{bvp}\,(a)$

$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (iplus (v-to-int (b), ineg (iplus (v-to-int (a), 1))))
$\wedge$ (exp (2, length (a) $-$ 1)
  $<$ negative-guts (iplus (v-to-int (b), ineg (iplus (v-to-int (a), 1))))))
$\rightarrow$ (int-to-v (iplus (v-to-int (b),
      iplus (exp (2, length (a)), ineg (iplus (v-to-int (a), 1)))),
    length (a))
  $=$ int-to-v (iplus (v-to-int (b), ineg (iplus (v-to-int (a), 1))),
    length (a)))

THEOREM: alu-thm_tc-bv-subb
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (bv (v-alu (c, a, b, list (**f**, **t**, **t**, **f**)))
  $=$ int-to-v (**if** tc-in-rangep (tc-add (v-to-int (b),
      tc-minus (tc-add (v-to-int (a),
        carry (c)))),
      length (a))
    then tc-add (v-to-int (b),
      tc-minus (tc-add (v-to-int (a), carry (c))))
    elseif negativep (tc-add (v-to-int (b),
      tc-minus (tc-add (v-to-int (a),
        carry (c)))))
    then tc-add (v-to-int (b),
      tc-add (exp (2, length (a)),
        tc-minus (tc-add (v-to-int (a),
          carry (c)))))
    else tc-add (v-to-int (b),
      tc-add ($-$ exp (2, length (a)),
        tc-minus (tc-add (v-to-int (a),
          carry (c))))) **endif**,
    length (a)))

THEOREM: alu-thm_tc-v-subb
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)

$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (v (v-alu (c, a, b, list (**f**, **t**, **t**, **f**)))
= ($\neg$ tc-in-rangep (tc-add (v-to-int (b),
tc-minus (tc-add (v-to-int (a), carry (c)))),
length (a))))

THEOREM: v-to-int-int-to-v-0
v-to-int (int-to-v (`0`, n)) = `0`

```
; A lemma length-int-to-v exists in fm9001.events, but it did not
; originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-length-int-to-v
length (int-to-v (x, n)) = fix (n)

THEOREM: alu-thm_tc-bv-decr-lemma1
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ ($\neg$ negativep (iplus (v-to-int (a), `-1`)))
$\wedge$ (iplus (v-to-int (a), `-1`) $\not<$ exp (2, length (a) $-$ 1)))
$\rightarrow$ (int-to-v (iplus (v-to-int (a), `-1`), length (a))
= int-to-v (iplus (v-to-int (a), iplus (`-1`, $-$ exp (2, length (a)))),
length (a)))

THEOREM: alu-thm_tc-bv-decr-lemma2
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (iplus (v-to-int (a), `-1`))
$\wedge$ (exp (2, length (a) $-$ 1) $<$ negative-guts (iplus (v-to-int (a), `-1`))))
$\rightarrow$ (int-to-v (iplus (v-to-int (a), `-1`), length (a))
= int-to-v (iplus (v-to-int (a), iplus (`-1`, exp (2, length (a)))),
length (a)))

THEOREM: alu-thm_tc-bv-decr
(bvp (a)

$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (bv (v-alu $(c,\ a,\ b,$ list $(\mathbf{t},\ \mathbf{f},\ \mathbf{t},\ \mathbf{f})))$
= int-to-v (**if** tc-in-rangep (tc-add (v-to-int (a), `-1`), length (a))
then tc-add (v-to-int (a), `-1`)
elseif negativep (tc-add (v-to-int (a), `-1`))
then tc-add (v-to-int (a), tc-add (`-1`, exp (`2`, length (a))))
else tc-add (v-to-int (a),
tc-add (`-1`, $-$ exp (`2`, length (a)))) **endif**,
length (a)))

THEOREM: iplus-0-2
iplus (`0`, x) = fix-int (x)

THEOREM: alu-thm_tc-bv-neg-lemma1
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ ($\neg$ negativep (ineg (v-to-int (a))))
$\wedge$ (ineg (v-to-int (a)) $\not<$ exp (`2`, length (a) $-$ 1))
$\wedge$ ($\neg$ integerp (ineg (v-to-int (a)))))
$\rightarrow$ (int-to-v (`0`, length (a))
= int-to-v (iplus ($-$ exp (`2`, length (a)), ineg (v-to-int (a))),
length (a)))

THEOREM: alu-thm_tc-bv-neg-lemma2
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ ($\neg$ negativep (ineg (v-to-int (a))))
$\wedge$ (ineg (v-to-int (a)) $\not<$ exp (`2`, length (a) $-$ 1))
$\wedge$ integerp (ineg (v-to-int (a))))
$\rightarrow$ (int-to-v (ineg (v-to-int (a)), length (a))
= int-to-v (iplus ($-$ exp (`2`, length (a)), ineg (v-to-int (a))),
length (a)))

THEOREM: alu-thm_tc-bv-neg-lemma3
(bvp (a)

$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (ineg (v-to-int (a)))
$\wedge$ (exp (2, length $(a) - 1$) $<$ negative-guts (ineg (v-to-int (a))))
$\wedge$ ($\neg$ integerp (ineg (v-to-int (a)))))
$\rightarrow$ (int-to-v (0, length (a))
= int-to-v (iplus (exp (2, length (a)), ineg (v-to-int (a))), length (a)))

THEOREM: alu-thm_tc-bv-neg-lemma4
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (ineg (v-to-int (a)))
$\wedge$ (exp (2, length $(a) - 1$) $<$ negative-guts (ineg (v-to-int (a))))
$\wedge$ integerp (ineg (v-to-int (a))))
$\rightarrow$ (int-to-v (ineg (v-to-int (a)), length (a))
= int-to-v (iplus (exp (2, length (a)), ineg (v-to-int (a))), length (a)))

THEOREM: alu-thm_tc-bv-neg-lemma5
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ ($\neg$ negativep (ineg (v-to-int (a))))
$\wedge$ (ineg (v-to-int (a)) $<$ exp (2, length $(a) - 1$))
$\wedge$ ($\neg$ integerp (ineg (v-to-int (a)))))
$\rightarrow$ (int-to-v (0, length (a)) = int-to-v (ineg (v-to-int (a)), length (a)))

THEOREM: alu-thm_tc-bv-neg-lemma6
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (ineg (v-to-int (a)))
$\wedge$ (exp (2, length $(a) - 1$) $\not<$ negative-guts (ineg (v-to-int (a))))
$\wedge$ ($\neg$ integerp (ineg (v-to-int (a)))))
$\rightarrow$ (int-to-v (0, length (a)) = int-to-v (ineg (v-to-int (a)), length (a)))

THEOREM: alu-thm_tc-bv-neg

$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \wedge\quad \text{listp}\,(a) \\
\ \wedge\quad \text{bvp}\,(b) \\
\ \wedge\quad \text{listp}\,(b) \\
\ \wedge\quad (\text{length}\,(a) = \text{length}\,(b)) \\
\ \wedge\quad \text{boolp}\,(c)) \\
\rightarrow\quad (\text{bv}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{f},\ \mathbf{f},\ \mathbf{t},\ \mathbf{f}))) \\
\qquad = \quad \text{int-to-v}\,(\mathbf{if}\ \text{tc-in-rangep}\,(\text{tc-minus}\,(\text{v-to-int}\,(a)),\ \text{length}\,(a)) \\
\qquad\qquad\qquad\quad \mathbf{then}\ \text{tc-minus}\,(\text{v-to-int}\,(a)) \\
\qquad\qquad\qquad\quad \mathbf{elseif}\ \text{negativep}\,(\text{tc-minus}\,(\text{v-to-int}\,(a))) \\
\qquad\qquad\qquad\quad \mathbf{then}\ \text{tc-add}\,(\exp\,(2,\ \text{length}\,(a)),\ \text{tc-minus}\,(\text{v-to-int}\,(a))) \\
\qquad\qquad\qquad\quad \mathbf{else}\ \text{tc-add}\,(-\ \exp\,(2,\ \text{length}\,(a)), \\
\qquad\qquad\qquad\qquad\qquad\quad \text{tc-minus}\,(\text{v-to-int}\,(a)))\ \mathbf{endif}, \\
\qquad\qquad\qquad\quad \text{length}\,(a)))
\end{array}
$$

THEOREM: diff-plus-plus-plus-cancellation

$$((a + (b + e)) - e) = (a + b)$$

THEOREM: alu-thm_tc-bv-add-lemma1

$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \wedge\quad \text{listp}\,(a) \\
\ \wedge\quad \text{bvp}\,(b) \\
\ \wedge\quad \text{listp}\,(b) \\
\ \wedge\quad (\text{length}\,(a) = \text{length}\,(b)) \\
\ \wedge\quad (\neg\ \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)))) \\
\ \wedge\quad (\text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)) \not< \exp\,(2,\ \text{length}\,(a) - 1))) \\
\rightarrow\quad (\text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)),\ \text{length}\,(a)) \\
\qquad = \quad \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad \text{iplus}\,(\text{v-to-int}\,(b),\ -\ \exp\,(2,\ \text{length}\,(a)))), \\
\qquad\qquad\qquad\quad \text{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-add-lemma2

$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \wedge\quad \text{listp}\,(a) \\
\ \wedge\quad \text{bvp}\,(b) \\
\ \wedge\quad \text{listp}\,(b) \\
\ \wedge\quad (\text{length}\,(a) = \text{length}\,(b)) \\
\ \wedge\quad \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))) \\
\ \wedge\quad (\exp\,(2,\ \text{length}\,(a) - 1) \\
\qquad < \quad \text{negative-guts}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))))) \\
\rightarrow\quad (\text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)),\ \text{length}\,(a)) \\
\qquad = \quad \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad \text{iplus}\,(\text{v-to-int}\,(b),\ \exp\,(2,\ \text{length}\,(a)))), \\
\qquad\qquad\qquad\quad \text{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-add

$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \wedge\ \ \text{listp}\,(a) \\
\ \wedge\ \ \text{bvp}\,(b) \\
\ \wedge\ \ \text{listp}\,(b) \\
\ \wedge\ \ (\text{length}\,(a) = \text{length}\,(b)) \\
\ \wedge\ \ \text{boolp}\,(c)) \\
\rightarrow\ (\text{bv}\,(\text{v-alu}\,(c,\ a,\ b,\ \text{list}\,(\mathbf{t},\ \mathbf{t},\ \mathbf{f},\ \mathbf{f}))) \\
\qquad = \ \text{int-to-v}\,(\textbf{if}\ \text{tc-in-rangep}\,(\text{tc-add}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{length}\,(a)) \\
\qquad\qquad\qquad\qquad \textbf{then}\ \text{tc-add}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)) \\
\qquad\qquad\qquad\qquad \textbf{elseif}\ \text{negativep}\,(\text{tc-add}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))) \\
\qquad\qquad\qquad\qquad \textbf{then}\ \text{tc-add}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\text{v-to-int}\,(b),\ \text{exp}\,(2,\ \text{length}\,(a)))) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \text{tc-add}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\text{v-to-int}\,(b), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad - \ \text{exp}\,(2,\ \text{length}\,(a)))) \ \textbf{endif}, \\
\qquad\qquad\qquad\qquad \text{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-addc-lemma1

$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \wedge\ \ \text{listp}\,(a) \\
\ \wedge\ \ \text{bvp}\,(b) \\
\ \wedge\ \ \text{listp}\,(b) \\
\ \wedge\ \ (\text{length}\,(a) = \text{length}\,(b)) \\
\ \wedge\ \ (\neg\ \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1)))) \\
\ \wedge\ \ (\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1)) \not< \text{exp}\,(2,\ \text{length}\,(a)\ -\ 1))) \\
\rightarrow\ (\text{int-to-v}\,(\text{iplus}\,(1,\ \text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))),\ \text{length}\,(a)) \\
\qquad = \ \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad \text{iplus}\,(\text{v-to-int}\,(b), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{iplus}\,(1,\ -\ \text{exp}\,(2,\ \text{length}\,(a))))), \\
\qquad\qquad\qquad\qquad \text{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-addc-lemma2

$$
\begin{array}{l}
(\text{bvp}\,(a) \\
\ \wedge\ \ \text{listp}\,(a) \\
\ \wedge\ \ \text{bvp}\,(b) \\
\ \wedge\ \ \text{listp}\,(b) \\
\ \wedge\ \ (\text{length}\,(a) = \text{length}\,(b)) \\
\ \wedge\ \ \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1))) \\
\ \wedge\ \ (\text{exp}\,(2,\ \text{length}\,(a)\ -\ 1) \\
\qquad < \ \text{negative-guts}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1))))) \\
\rightarrow\ (\text{int-to-v}\,(\text{iplus}\,(1,\ \text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))),\ \text{length}\,(a)) \\
\qquad = \ \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a),
\end{array}
$$

$$\text{iplus}\,(\text{v-to-int}\,(b),\ \text{iplus}\,(1,\ \exp(2,\ \text{length}\,(a))))),$$
$$\text{length}\,(a)))$$

```
; The following three lemmas are killers as far as performance goes.
; I prove them, use them to get the lemmas I need, and then
; disable them.
```

THEOREM: alu-thm_tc-bv-addc-lemma3-killer1

$$\begin{aligned}
&((a \in \mathbf{N}) \\
&\ \wedge\ \ (a \not< ((e + e) - b)) \\
&\ \wedge\ \ (1 \not< ((e + e) - b)) \\
&\ \wedge\ \ (b < (e + e))) \\
&\rightarrow\ \ ((1 + ((a + b) - (e + e))) = a)
\end{aligned}$$

THEOREM: alu-thm_tc-bv-addc-lemma3-killer2

$$\begin{aligned}
&((1 \not< ((e + e) - b)) \wedge (b < (e + e))) \\
&\rightarrow\ \ (((e + e) - b) = 1)
\end{aligned}$$

THEOREM: alu-thm_tc-bv-addc-lemma3-killer3

$$(\text{v-to-nat}\,(a) < 1) \rightarrow (\text{v-to-nat}\,(a) = 0)$$

THEOREM: alu-thm_tc-bv-addc-lemma3-lemma1

$$\begin{aligned}
&((\text{v-to-nat}\,(b) < (\exp(2,\ \text{length}\,(a) - 1) + \exp(2,\ \text{length}\,(a) - 1))) \\
&\ \wedge\ \ (1 \not< ((\exp(2,\ \text{length}\,(a) - 1) + \exp(2,\ \text{length}\,(a) - 1)) \\
&\qquad\qquad -\ \ \text{v-to-nat}\,(b))) \\
&\ \wedge\ \ (\text{v-to-nat}\,(a) \\
&\qquad \not<\ \ ((\exp(2,\ \text{length}\,(a) - 1) + \exp(2,\ \text{length}\,(a) - 1)) \\
&\qquad\qquad -\ \ \text{v-to-nat}\,(b)))) \\
&\rightarrow\ \ ((\text{nat-to-v}\,(1 + ((\text{v-to-nat}\,(a) + \text{v-to-nat}\,(b)) \\
&\qquad\qquad\qquad\qquad -\ \ (\exp(2,\ \text{length}\,(a) - 1) \\
&\qquad\qquad\qquad\qquad\qquad +\ \ \exp(2,\ \text{length}\,(a) - 1))), \\
&\qquad\qquad\qquad \text{length}\,(a)) \\
&\qquad\ =\ \ \text{nat-to-v}\,(\text{v-to-nat}\,(a),\ \text{length}\,(a))) \\
&\qquad =\ \ \mathbf{t})
\end{aligned}$$

THEOREM: alu-thm_tc-bv-addc-lemma3-lemma2

$$\begin{aligned}
&((1 \not< ((\exp(2,\ \text{length}\,(a) - 1) + \exp(2,\ \text{length}\,(a) - 1)) - \text{v-to-nat}\,(b))) \\
&\ \wedge\ \ (\text{v-to-nat}\,(b) < (\exp(2,\ \text{length}\,(a) - 1) + \exp(2,\ \text{length}\,(a) - 1))) \\
&\ \wedge\ \ (\text{v-to-nat}\,(a) \\
&\qquad <\ \ ((\exp(2,\ \text{length}\,(a) - 1) + \exp(2,\ \text{length}\,(a) - 1)) \\
&\qquad\qquad -\ \ \text{v-to-nat}\,(b)))) \\
&\rightarrow\ \ ((\text{nat-to-v}\,(0,\ \text{length}\,(a)) = \text{nat-to-v}\,(\text{v-to-nat}\,(a),\ \text{length}\,(a))) = \mathbf{t})
\end{aligned}$$

EVENT: Disable alu-thm_tc-bv-addc-lemma3-killer1.

EVENT: Disable alu-thm_tc-bv-addc-lemma3-killer2.

EVENT: Disable alu-thm_tc-bv-addc-lemma3-killer3.

THEOREM: alu-thm_tc-bv-addc-lemma3

$$
\begin{array}{ll}
(\text{bvp}\,(a) & \\
\wedge & \text{listp}\,(a) \\
\wedge & \text{bvp}\,(b) \\
\wedge & \text{listp}\,(b) \\
\wedge & (\text{length}\,(a) = \text{length}\,(b)) \\
\wedge & (\neg\ \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1)))) \\
\wedge & (\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1)) < \exp\,(2,\ \text{length}\,(a) - 1))) \\
\rightarrow & (\text{int-to-v}\,(\text{iplus}\,(1,\ \text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))),\ \text{length}\,(a)) \\
& \quad = \quad \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1)),\ \text{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-addc-lemma4-lemma1

$$
\begin{array}{ll}
((\text{v-to-nat}\,(b) < (\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1))) & \\
\wedge & (\text{v-to-nat}\,(a) < (\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1))) \\
\wedge & ((\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1)) \\
& \quad \not< \quad (((\exp\,(2,\ \text{length}\,(a) - 1) \\
& \qquad\qquad + \quad \exp\,(2,\ \text{length}\,(a) - 1) \\
& \qquad\qquad + \quad \exp\,(2,\ \text{length}\,(a) - 1) \\
& \qquad\qquad + \quad \exp\,(2,\ \text{length}\,(a) - 1)) \\
& \qquad\quad - \quad \text{v-to-nat}\,(a)) \\
& \qquad - \quad \text{v-to-nat}\,(b)))) \\
\rightarrow & (((\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1)) \\
& \quad - \quad (((\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1)) \\
& \qquad\quad - \quad \text{v-to-nat}\,(a)) \\
& \qquad + \quad (((\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1)) \\
& \qquad\qquad - \quad \text{v-to-nat}\,(b)) - 1))) \\
& \quad = \quad (1 + ((\text{v-to-nat}\,(a) + \text{v-to-nat}\,(b)) \\
& \qquad\qquad - \quad (\exp\,(2,\ \text{length}\,(a) - 1) + \exp\,(2,\ \text{length}\,(a) - 1)))))
\end{array}
$$

THEOREM: alu-thm_tc-bv-addc-lemma4

$$
\begin{array}{ll}
(\text{bvp}\,(a) & \\
\wedge & \text{listp}\,(a) \\
\wedge & \text{bvp}\,(b) \\
\wedge & \text{listp}\,(b) \\
\wedge & (\text{length}\,(a) = \text{length}\,(b)) \\
\wedge & \text{negativep}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1))) \\
\wedge & (\exp\,(2,\ \text{length}\,(a) - 1) \\
& \quad \not< \quad \text{negative-guts}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1))))) \\
\rightarrow & (\text{int-to-v}\,(\text{iplus}\,(1,\ \text{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))),\ \text{length}\,(a)) \\
& \quad = \quad \text{int-to-v}\,(\text{iplus}\,(\text{v-to-int}\,(a),\ \text{iplus}\,(\text{v-to-int}\,(b),\ 1)),\ \text{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-bv-addc

$$
\begin{array}{l}
(\mathrm{bvp}\,(a) \\
\;\land\;\; \mathrm{listp}\,(a) \\
\;\land\;\; \mathrm{bvp}\,(b) \\
\;\land\;\; \mathrm{listp}\,(b) \\
\;\land\;\; (\mathrm{length}\,(a) = \mathrm{length}\,(b)) \\
\;\land\;\; \mathrm{boolp}\,(c)) \\
\to\;\; (\mathrm{bv}\,(\text{v-alu}\,(c,\ a,\ b,\ \mathrm{list}\,(\mathbf{f},\ \mathbf{t},\ \mathbf{f},\ \mathbf{f}))) \\
\qquad = \;\; \text{int-to-v}\,(\mathbf{if}\ \text{tc-in-rangep}\,(\text{tc-add}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\text{v-to-int}\,(b),\ \mathrm{carry}\,(c))), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{length}\,(a)) \\
\qquad\qquad\qquad \mathbf{then}\ \text{tc-add}\,(\text{v-to-int}\,(a),\ \text{tc-add}\,(\text{v-to-int}\,(b),\ \mathrm{carry}\,(c))) \\
\qquad\qquad\qquad \mathbf{elseif}\ \mathrm{negativep}\,(\text{tc-add}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\text{v-to-int}\,(b),\ \mathrm{carry}\,(c)))) \\
\qquad\qquad\qquad \mathbf{then}\ \text{tc-add}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\text{v-to-int}\,(b), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\mathrm{carry}\,(c),\ \exp\,(2,\ \mathrm{length}\,(a))))) \\
\qquad\qquad\qquad \mathbf{else}\ \text{tc-add}\,(\text{v-to-int}\,(a), \\
\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\text{v-to-int}\,(b), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{tc-add}\,(\mathrm{carry}\,(c), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad - \exp\,(2,\ \mathrm{length}\,(a))))) \ \mathbf{endif}, \\
\qquad\qquad\qquad \mathrm{length}\,(a)))
\end{array}
$$

THEOREM: alu-thm_tc-v-addc-lemma1

$$
\begin{array}{l}
(\mathrm{bvp}\,(a) \\
\;\land\;\; \mathrm{listp}\,(a) \\
\;\land\;\; \mathrm{bvp}\,(b) \\
\;\land\;\; \mathrm{listp}\,(b) \\
\;\land\;\; (\mathrm{length}\,(a) = \mathrm{length}\,(b)) \\
\;\land\;\; (\neg\ \mathrm{negativep}\,(\mathrm{iplus}\,(\text{v-to-int}\,(a),\ \mathrm{iplus}\,(\text{v-to-int}\,(b),\ 1)))) \\
\;\land\;\; (\mathrm{iplus}\,(\text{v-to-int}\,(a),\ \mathrm{iplus}\,(\text{v-to-int}\,(b),\ 1)) \not< \exp\,(2,\ \mathrm{length}\,(a) - 1)) \\
\;\land\;\; (\neg\ \mathrm{negativep}\,(\mathrm{iplus}\,(1,\ \mathrm{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)))))) \\
\to\;\; ((\mathrm{iplus}\,(1,\ \mathrm{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b))) < \exp\,(2,\ \mathrm{length}\,(a) - 1)) \\
\qquad = \;\; \mathbf{f})
\end{array}
$$

THEOREM: alu-thm_tc-v-addc-lemma2

$$
\begin{array}{l}
(\mathrm{bvp}\,(a) \\
\;\land\;\; \mathrm{listp}\,(a) \\
\;\land\;\; \mathrm{bvp}\,(b) \\
\;\land\;\; \mathrm{listp}\,(b) \\
\;\land\;\; (\mathrm{length}\,(a) = \mathrm{length}\,(b)) \\
\;\land\;\; (\neg\ \mathrm{negativep}\,(\mathrm{iplus}\,(\text{v-to-int}\,(a),\ \mathrm{iplus}\,(\text{v-to-int}\,(b),\ 1)))) \\
\;\land\;\; (\mathrm{iplus}\,(\text{v-to-int}\,(a),\ \mathrm{iplus}\,(\text{v-to-int}\,(b),\ 1)) \not< \exp\,(2,\ \mathrm{length}\,(a) - 1)) \\
\;\land\;\; \mathrm{negativep}\,(\mathrm{iplus}\,(1,\ \mathrm{iplus}\,(\text{v-to-int}\,(a),\ \text{v-to-int}\,(b)))))
\end{array}
$$

$\rightarrow$ ((exp (2, length (a) $-$ 1)
$<$ negative-guts (iplus (1, iplus (v-to-int (a), v-to-int (b)))))
$=$ **t**)

THEOREM: alu-thm_tc-v-addc-lemma3
(bvp (a)
$\land$ listp (a)
$\land$ bvp (b)
$\land$ listp (b)
$\land$ (length (a) $=$ length (b))
$\land$ negativep (iplus (v-to-int (a), iplus (v-to-int (b), 1)))
$\land$ (exp (2, length (a) $-$ 1)
$<$ negative-guts (iplus (v-to-int (a), iplus (v-to-int (b), 1))))
$\land$ ($\neg$ negativep (iplus (1, iplus (v-to-int (a), v-to-int (b))))))
$\rightarrow$ ((iplus (1, iplus (v-to-int (a), v-to-int (b))) $<$ exp (2, length (a) $-$ 1))
$=$ **f**)

THEOREM: alu-thm_tc-v-addc-lemma4
(bvp (a)
$\land$ listp (a)
$\land$ bvp (b)
$\land$ listp (b)
$\land$ (length (a) $=$ length (b))
$\land$ negativep (iplus (v-to-int (a), iplus (v-to-int (b), 1)))
$\land$ (exp (2, length (a) $-$ 1)
$<$ negative-guts (iplus (v-to-int (a), iplus (v-to-int (b), 1))))
$\land$ negativep (iplus (1, iplus (v-to-int (a), v-to-int (b)))))
$\rightarrow$ ((exp (2, length (a) $-$ 1)
$<$ negative-guts (iplus (1, iplus (v-to-int (a), v-to-int (b)))))
$=$ **t**)

THEOREM: alu-thm_tc-v-addc-lemma5
(bvp (a)
$\land$ listp (a)
$\land$ bvp (b)
$\land$ listp (b)
$\land$ (length (a) $=$ length (b))
$\land$ ($\neg$ negativep (iplus (v-to-int (a), iplus (v-to-int (b), 1))))
$\land$ (iplus (v-to-int (a), iplus (v-to-int (b), 1)) $<$ exp (2, length (a) $-$ 1))
$\land$ ($\neg$ negativep (iplus (1, iplus (v-to-int (a), v-to-int (b))))))
$\rightarrow$ ((iplus (1, iplus (v-to-int (a), v-to-int (b))) $<$ exp (2, length (a) $-$ 1))
$=$ **t**)

THEOREM: alu-thm_tc-v-addc-lemma6
(bvp (a)

$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ ($\neg$ negativep (iplus (v-to-int (a), iplus (v-to-int (b), 1))))
$\wedge$ (iplus (v-to-int (a), iplus (v-to-int (b), 1)) $<$ exp (2, length (a) $-$ 1))
$\wedge$ negativep (iplus (1, iplus (v-to-int (a), v-to-int (b))))
$\rightarrow$ ((exp (2, length (a) $-$ 1)
$<$ negative-guts (iplus (1, iplus (v-to-int (a), v-to-int (b)))))
= **f**)

THEOREM: alu-thm_tc-v-addc-lemma7
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (iplus (v-to-int (a), iplus (v-to-int (b), 1)))
$\wedge$ (exp (2, length (a) $-$ 1)
$\not<$ negative-guts (iplus (v-to-int (a), iplus (v-to-int (b), 1))))
$\wedge$ ($\neg$ negativep (iplus (1, iplus (v-to-int (a), v-to-int (b)))))
$\rightarrow$ ((iplus (1, iplus (v-to-int (a), v-to-int (b))) $<$ exp (2, length (a) $-$ 1))
= **t**)

THEOREM: alu-thm_tc-v-addc-lemma8
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ negativep (iplus (v-to-int (a), iplus (v-to-int (b), 1)))
$\wedge$ (exp (2, length (a) $-$ 1)
$\not<$ negative-guts (iplus (v-to-int (a), iplus (v-to-int (b), 1))))
$\wedge$ negativep (iplus (1, iplus (v-to-int (a), v-to-int (b))))
$\rightarrow$ ((exp (2, length (a) $-$ 1)
$<$ negative-guts (iplus (1, iplus (v-to-int (a), v-to-int (b)))))
= **f**)

THEOREM: alu-thm_tc-v-addc
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))

$\rightarrow$ (v (v-alu (c, a, b, list (**f**, **t**, **f**, **f**)))
= ($\neg$ tc-in-rangep (tc-add (v-to-int (a),
tc-add (v-to-int (b), carry (c))),
length (a))))

THEOREM: alu-thm_tc-bv-incr-lemma1
(bvp (a)
$\wedge$ listp (a)
$\wedge$ ($\neg$ negativep (iplus (v-to-int (a), 1)))
$\wedge$ (iplus (v-to-int (a), 1) $\not<$ exp (2, length (a) $-$ 1)))
$\rightarrow$ (int-to-v (iplus (v-to-int (a), iplus (1, $-$ exp (2, length (a)))), length (a))
= int-to-v (iplus (1, v-to-int (a)), length (a)))

THEOREM: alu-thm_tc-bv-incr-lemma2
(bvp (a)
$\wedge$ listp (a)
$\wedge$ negativep (iplus (v-to-int (a), 1))
$\wedge$ (exp (2, length (a) $-$ 1) $<$ negative-guts (iplus (v-to-int (a), 1))))
$\rightarrow$ (int-to-v (iplus (v-to-int (a), iplus (1, exp (2, length (a)))), length (a))
= int-to-v (iplus (1, v-to-int (a)), length (a)))

THEOREM: alu-thm_tc-bv-incr
(bvp (a)
$\wedge$ listp (a)
$\wedge$ bvp (b)
$\wedge$ listp (b)
$\wedge$ (length (a) = length (b))
$\wedge$ boolp (c))
$\rightarrow$ (bv (v-alu (c, a, b, list (**t**, **f**, **f**, **f**)))
= int-to-v (**if** tc-in-rangep (tc-add (v-to-int (a), 1), length (a))
then tc-add (v-to-int (a), 1)
elseif negativep (tc-add (v-to-int (a), 1))
then tc-add (v-to-int (a), tc-add (1, exp (2, length (a))))
else tc-add (v-to-int (a),
tc-add (1, $-$ exp (2, length (a)))) **endif**,
length (a)))

```
; That completes the alu interpretation lemmas.  Now we disable everything we did
; in this section other than the old-style interpretation lemmas (i.e., we make the
; data base behave as though we had done an add-axiom for each of the interpretation
; lemmas).  The reason for this is that the first time we rebuilt Piton on FM9001 we
; skipped the proofs of these boring facts so we could get on with the interesting
; part and now, having proved them, we have to make sure the lemmas we added don't
; mess up future proofs.
```

EVENT: Disable theory equal-length-0, negativep-ineg, not-equal-exp-0, ineg-exp-2-x, diff-diff, alu-thm_tc-bv-sub-lemma, alu-thm_tc-bv-sub-lemma2, diff-plus-cancellation-hack2, diff-plus-sub1-hack1, alu-thm_tc-bv-subb-lemma1, diff-plus-sub1-hack2, alu-thm_tc-bv-subb-lemma2-lemma1, diff-plus-sub1-hack2-off, alu-thm_tc-bv-subb-lemma2, v-to-int-int-to-v-0, my-length-int-to-v, alu-thm_tc-bv-decr-lemma1, alu-thm_tc-bv-decr-lemma2, iplus-0-2, alu-thm_tc-bv-neg-lemma1, alu-thm_tc-bv-neg-lemma2, alu-thm_tc-bv-neg-lemma3, alu-thm_tc-bv-neg-lemma4, alu-thm_tc-bv-neg-lemma5, alu-thm_tc-bv-neg-lemma6, diff-plus-plus-plus-cancellation, alu-thm_tc-bv-add-lemma1, alu-thm_tc-bv-add-lemma2, alu-thm_tc-bv-addc-lemma1, alu-thm_tc-bv-addc-lemma2, alu-thm_tc-bv-addc-lemma3-killer1, alu-thm_tc-bv-addc-lemma3-killer2, alu-thm_tc-bv-addc-lemma3-killer3, alu-thm_tc-bv-addc-lemma3-lemma1, alu-thm_tc-bv-addc-lemma3-lemma2, alu-thm_tc-bv-addc-lemma3-killer1-off, alu-thm_tc-bv-addc-lemma3-killer2-off, alu-thm_tc-bv-addc-lemma3-killer3-off, alu-thm_tc-bv-addc-lemma3, alu-thm_tc-bv-addc-lemma4-lemma1, alu-thm_tc-bv-addc-lemma4, alu-thm_tc-v-addc-lemma1, alu-thm_tc-v-addc-lemma2, alu-thm_tc-v-addc-lemma3, alu-thm_tc-v-addc-lemma4, alu-thm_tc-v-addc-lemma5, alu-thm_tc-v-addc-lemma6, alu-thm_tc-v-addc-lemma7, alu-thm_tc-v-addc-lemma8, alu-thm_tc-bv-incr-lemma1, alu-thm_tc-bv-incr-lemma2, bv, v, c, v-alu-correct-int-rewriter, unlabel, i-usr-data-typep.

```
; This completes our rule base for the alu.  We should now be able
; to reduce any bv-alu-cv expression that arises.

; This completes the preliminary pass from m down to fm9001.
```

```
; I now prove a few useful equivalences between
; concepts at the m level and concepts at the i level.
```

THEOREM: nth-is-get
$\text{nth}\,(n,\ l) = \text{get}\,(n,\ l)$

THEOREM: update-nth-is-put
$(n < \text{length}\,(l)) \rightarrow (\text{update-nth}\,(n,\ l,\ v) = \text{put}\,(v,\ n,\ l))$

```
; Goal:  v-to-nat-inverts-nat-to-v

; Subgoal: remainder-quotient-elim

; First we build up the classic proof of remainder-quotient-elim, doing
; a lot of arithmetic along the way.
```

EVENT: Enable difference-elim.

THEOREM: difference-elim-rewrite
$((x \in \mathbf{N}) \wedge (x \not< y)) \rightarrow (((x - y) + y) = x)$

THEOREM: v-to-nat-inverts-nat-to-v
$\text{small-naturalp}\,(n,\ \textit{word-size}) \rightarrow (\text{v-to-nat}\,(\text{nat-to-v}\,(n,\ \textit{word-size})) = n)$

```
; QED:  v-to-nat-inverts-nat-to-v


; Next I will speed up the handling of i-objectp-type for
; the case where the type is a quoted constant.
```

THEOREM: i-objectp-type-simplifier
$$
\begin{aligned}
&(\text{i-objectp-type}\,(\texttt{'nat},\ x,\ i)\\
&\quad = \quad ((\text{type}\,(x) = \texttt{'nat})\\
&\qquad\quad \wedge \quad (\text{cddr}\,(x) = \mathbf{nil})\\
&\qquad\quad \wedge \quad \text{small-naturalp}\,(\text{untag}\,(x),\ \text{i-word-size}\,(i))))\\
&\wedge \quad (\text{i-objectp-type}\,(\texttt{'int},\ x,\ i)\\
&\qquad = \quad ((\text{type}\,(x) = \texttt{'int})\\
&\qquad\qquad \wedge \quad (\text{cddr}\,(x) = \mathbf{nil})\\
&\qquad\qquad \wedge \quad \text{small-integerp}\,(\text{untag}\,(x),\ \text{i-word-size}\,(i))))\\
&\wedge \quad (\text{i-objectp-type}\,(\texttt{'bitv},\ x,\ i)\\
&\qquad = \quad ((\text{type}\,(x) = \texttt{'bitv})
\end{aligned}
$$

$\wedge$ (cddr (x) = **nil**)
$\wedge$ bit-vectorp (untag (x), i-word-size (i))))
$\wedge$ (i-objectp-type (`'bool`, x, i)
= ((type (x) = `'bool`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ booleanp (untag (x))))
$\wedge$ (i-objectp-type (`'addr`, x, i)
= ((type (x) = `'addr`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ adpp (untag (x), i-usr-data-segment (i))))
$\wedge$ (i-objectp-type (`'pc`, x, i)
= ((type (x) = `'pc`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ icode-labelp (untag (x), i-prog-segment (i))))
$\wedge$ (i-objectp-type (`'subr`, x, i)
= ((type (x) = `'subr`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ adpp (cons (untag (x), `0`), i-prog-segment (i))))
$\wedge$ (i-objectp-type (`'sys-addr`, x, i)
= ((type (x) = `'sys-addr`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ adpp (untag (x), i-sys-data-segment (i))))
$\wedge$ (i-objectp-type (`'ipc`, x, i)
= ((type (x) = `'ipc`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ adpp (untag (x), i-prog-segment (i))))

EVENT: Disable i-objectp-type.

```
; I will disable type, tag and untag for sanity's sake,
; but prove the obvious relationships first.
```

THEOREM: type-tag
type (tag (x, y)) = x

THEOREM: untag-tag
untag (tag (x, y)) = y

EVENT: Disable type.

EVENT: Disable tag.

EVENT: Disable untag.

```
; And I will do a similar thing for the adp handlers
```

THEOREM: adp-name-cons
adp-name (cons ($name$, $offset$)) = $name$

THEOREM: adp-offset-cons
adp-offset (cons ($name$, $offset$)) = $offset$

THEOREM: adp-name-add-adp
adp-name (add-adp (adp, n)) = adp-name (adp)

THEOREM: adp-offset-add-adp
adp-offset (add-adp (adp, n)) = (adp-offset (adp) + n)

THEOREM: adp-name-sub-adp
adp-name (sub-adp (adp, n)) = adp-name (adp)

THEOREM: adp-offset-sub-adp
adp-offset (sub-adp (adp, n)) = (adp-offset (adp) − n)

EVENT: Disable sub-adp.

EVENT: Disable add-adp.

EVENT: Disable adp-name.

EVENT: Disable adp-offset.

```
; The function BOOLEANP is a killer because it
; causes everything to split.  I will disable it
; and prove what I need about it as necessary.
```

EVENT: Disable booleanp.

EVENT: Disable bool-to-logical.

```
; I am now going to partition the conjunctions of i-state-okp
; into two pots.  The first pot will put into the function
; defined below, i-state-types-okp, which will remain disabled.
; I will extract stuff from this hyp by back chaining when
; necessary.
```

Definition:
i-state-types-okp (i)
= (i-statep (i)
∧ (type (i-pc (i)) = `'ipc`)
∧ (cddr (i-pc (i)) = **nil**)
∧ listp (untag (i-pc (i)))
∧ definedp (adp-name (untag (i-pc (i))), i-prog-segment (i))
∧ (type (i-cfp (i)) = `'sys-addr`)
∧ listp (untag (i-cfp (i)))
∧ (adp-name (untag (i-cfp (i))) = `'cstk`)
∧ (cddr (i-cfp (i)) = **nil**)
∧ definedp (`'cstk`, i-sys-data-segment (i))
∧ (type (i-csp (i)) = `'sys-addr`)
∧ listp (untag (i-csp (i)))
∧ (adp-name (untag (i-csp (i))) = `'cstk`)
∧ (cddr (i-csp (i)) = **nil**)
∧ (type (i-tsp (i)) = `'sys-addr`)
∧ listp (untag (i-tsp (i)))
∧ (adp-name (untag (i-tsp (i))) = `'tstk`)
∧ (cddr (i-tsp (i)) = **nil**)
∧ definedp (`'tstk`, i-sys-data-segment (i))
∧ (type (i-c-flg (i)) = `'bool`)
∧ (cddr (i-c-flg (i)) = **nil**)
∧ booleanp (untag (i-c-flg (i)))
∧ (type (i-v-flg (i)) = `'bool`)
∧ (cddr (i-v-flg (i)) = **nil**)
∧ booleanp (untag (i-v-flg (i)))
∧ (type (i-n-flg (i)) = `'bool`)
∧ (cddr (i-n-flg (i)) = **nil**)
∧ booleanp (untag (i-n-flg (i)))
∧ (type (i-z-flg (i)) = `'bool`)
∧ (cddr (i-z-flg (i)) = **nil**)
∧ booleanp (untag (i-z-flg (i))))

```
; The second pot will be introduced explicitly into any clause
; containing i-state-okp.  These hyps will all be useful to
; linear and I don't want to hide them.

; Here is the lemma that does the restructuring.
```

Theorem: i-state-okp-restructuring
i-state-okp (i, *load-addr*)
= (i-state-types-okp (i)

$\land$ (adp-offset (untag (i-pc (i))) $\in$ **N**)
$\land$ (adp-offset (untag (i-pc (i)))
  $<$ length (cdr (assoc (adp-name (untag (i-pc (i))),
                i-prog-segment (i)))))
$\land$ (adp-offset (untag (i-cfp (i))) $\in$ **N**)
$\land$ (adp-offset (untag (i-cfp (i)))
  $<$ length (cdr (assoc (`'cstk`, i-sys-data-segment (i)))))
$\land$ (adp-offset (untag (i-csp (i))) $\in$ **N**)
$\land$ (adp-offset (untag (i-csp (i)))
  $<$ length (cdr (assoc (`'cstk`, i-sys-data-segment (i)))))
$\land$ (adp-offset (untag (i-tsp (i))) $\in$ **N**)
$\land$ (adp-offset (untag (i-tsp (i)))
  $<$ length (cdr (assoc (`'tstk`, i-sys-data-segment (i)))))
$\land$ ((*load-addr*
  $+$ segment-length (i-prog-segment (i))
  $+$ segment-length (i-usr-data-segment (i))
  $+$ segment-length (i-sys-data-segment (i)))
  $<$ exp (2, i-word-size (i))))

EVENT: Disable i-state-okp.

```
; And here is the lemma that lets me back chain into i-state-types-okp.
```

THEOREM: i-state-types-okp-properties
i-state-types-okp (i)
$\rightarrow$ (i-statep (i)
$\land$ (type (i-pc (i)) $=$ `'ipc`)
$\land$ (cddr (i-pc (i)) $=$ **nil**)
$\land$ (listp (untag (i-pc (i))) $=$ **t**)
$\land$ definedp (adp-name (untag (i-pc (i))), i-prog-segment (i))
$\land$ (type (i-cfp (i)) $=$ `'sys-addr`)
$\land$ (listp (untag (i-cfp (i))) $=$ **t**)
$\land$ (adp-name (untag (i-cfp (i))) $=$ `'cstk`)
$\land$ (cddr (i-cfp (i)) $=$ **nil**)
$\land$ definedp (`'cstk`, i-sys-data-segment (i))
$\land$ (type (i-csp (i)) $=$ `'sys-addr`)
$\land$ (listp (untag (i-csp (i))) $=$ **t**)
$\land$ (adp-name (untag (i-csp (i))) $=$ `'cstk`)
$\land$ (cddr (i-csp (i)) $=$ **nil**)
$\land$ (type (i-tsp (i)) $=$ `'sys-addr`)
$\land$ (listp (untag (i-tsp (i))) $=$ **t**)
$\land$ (adp-name (untag (i-tsp (i))) $=$ `'tstk`)
$\land$ (cddr (i-tsp (i)) $=$ **nil**)

$\land$ definedp ('**tstk**, i-sys-data-segment (i))
$\land$ (type (i-c-flg (i)) = '**bool**)
$\land$ (cddr (i-c-flg (i)) = **nil**)
$\land$ booleanp (untag (i-c-flg (i)))
$\land$ (type (i-v-flg (i)) = '**bool**)
$\land$ (cddr (i-v-flg (i)) = **nil**)
$\land$ booleanp (untag (i-v-flg (i)))
$\land$ (type (i-n-flg (i)) = '**bool**)
$\land$ (cddr (i-n-flg (i)) = **nil**)
$\land$ booleanp (untag (i-n-flg (i)))
$\land$ (type (i-z-flg (i)) = '**bool**)
$\land$ (cddr (i-z-flg (i)) = **nil**)
$\land$ booleanp (untag (i-z-flg (i))))

EVENT: Disable i-state-types-okp.

```
; Some accellerators for psw handling...
```

THEOREM: not-equal-x-y-error-msg-run
x-y-error-msg (x, y) $\neq$ '**run**

EVENT: Disable x-y-error-msg.

THEOREM: i-psw-i-halt
i-psw (i-halt (i, psw)) = psw

EVENT: Disable i-halt.

```
; Disabling some messy functions.

; I will disable a bunch of functions that cause large case
; splits.  These functions will have to be explicitly enabled
; to do proofs about their values.
```

THEOREM: link-instr-word-expander
link-instr-word (cons (*opcode*, **nil**), *word-size*)
= mci (cadr (assoc (*opcode*, LINK-INSTRUCTION-ALIST)), *word-size*)

EVENT: Disable link-instr-word.

THEOREM: link-data-word-expander

litatom (type (x))
$\rightarrow$ (link-data-word (x, *link-tables*, *word-size*)
$=$ **case on** type (x):
case $=$ *nat*
then nat-to-v (untag (x), *word-size*)
case $=$ *int*
then int-to-v (untag (x), *word-size*)
case $=$ *bitv*
then bitv-to-v (untag (x), *word-size*)
case $=$ *bool*
then bool-to-v (untag (x), *word-size*)
case $=$ *addr*
then addr-to-v (untag (x),
usr-data-links (*link-tables*),
word-size)
case $=$ *subr*
then subr-to-v (untag (x), prog-links (*link-tables*), *word-size*)
case $=$ *sys-addr*
then sys-addr-to-v (untag (x),
sys-data-links (*link-tables*),
word-size)
case $=$ *pc*
then label-to-v (untag (x),
prog-label-tables (*link-tables*),
word-size)
case $=$ *ipc*
then ipc-to-v (untag (x), prog-links (*link-tables*), *word-size*)
otherwise nat-to-v (0, *word-size*) **endcase**)

THEOREM: size-incr
length (incr (c, a)) $=$ length (a)

THEOREM: size-compl
length (v-not (a)) $=$ length (a)

THEOREM: size-bitv-to-v
length (bitv-to-v (a, *word-size*)) $=$ fix (*word-size*)

THEOREM: size-link-data-word
length (link-data-word (x, *link-tables*, *word-size*)) $=$ fix (*word-size*)

EVENT: Disable link-data-word.

EVENT: Disable *1*exp.

THEOREM: size-link-instr-word
$\text{length}\,(\text{link-instr-word}\,(\mathit{ins},\ \mathit{word\text{-}size})) = \text{fix}\,(\mathit{word\text{-}size})$

THEOREM: size-link-word
$\text{length}\,(\text{link-word}\,(x,\ \mathit{link\text{-}tables},\ \mathit{word\text{-}size})) = \text{fix}\,(\mathit{word\text{-}size})$

THEOREM: link-word-expander

$$
\begin{array}{l}
(\text{icode-instructionp}\,(x) \\
\quad \rightarrow \quad (\text{link-word}\,(x,\ \mathit{link\text{-}tables},\ \mathit{word\text{-}size}) = \text{link-instr-word}\,(x,\ \mathit{word\text{-}size}))) \\
\land \quad ((\neg\ \text{icode-instructionp}\,(x)) \\
\qquad \rightarrow \quad (\text{link-word}\,(x,\ \mathit{link\text{-}tables},\ \mathit{word\text{-}size}) \\
\qquad\qquad = \quad \text{link-data-word}\,(x,\ \mathit{link\text{-}tables},\ \mathit{word\text{-}size})))
\end{array}
$$

EVENT: Disable link-word.

```
; I am going to keep the i-link-tables function turned off and
; enable it as needed.  It expands into something I don't like
; reading.  Instead I'll prove the rewrite rules that let the
; four accessors work on it.
```

THEOREM: prog-links-i-link-tables

$$
\begin{array}{l}
\text{prog-links}\,(\text{i-link-tables}\,(i,\ \mathit{load\text{-}addr})) \\
= \quad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))
\end{array}
$$

THEOREM: prog-label-tables-i-link-tables

$$
\begin{array}{l}
\text{prog-label-tables}\,(\text{i-link-tables}\,(i,\ \mathit{load\text{-}addr})) \\
= \quad \text{link-table-for-prog-labels}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))
\end{array}
$$

THEOREM: usr-data-links-i-link-tables

$$
\begin{array}{l}
\text{usr-data-links}\,(\text{i-link-tables}\,(i,\ \mathit{load\text{-}addr})) \\
= \quad \text{link-table-for-segment}\,(\text{i-usr-data-segment}\,(i),\ \mathit{load\text{-}addr})
\end{array}
$$

THEOREM: sys-data-links-i-link-tables

$$
\begin{array}{l}
\text{sys-data-links}\,(\text{i-link-tables}\,(i,\ \mathit{load\text{-}addr})) \\
= \quad \text{link-table-for-segment}\,(\text{i-sys-data-segment}\,(i), \\
\qquad\qquad \mathit{load\text{-}addr} \\
\qquad\qquad + \quad \text{segment-length}\,(\text{i-prog-segment}\,(i)) \\
\qquad\qquad + \quad \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))
\end{array}
$$

EVENT: Disable i-link-tables.

EVENT: Disable prog-links.

EVENT: Disable prog-label-tables.

EVENT: Disable usr-data-links.

EVENT: Disable sys-data-links.

```
; Goal:  The fundamental theorems about Link Tables
```

THEOREM: assoc-append
$\text{assoc}\,(x,\ \text{append}\,(\mathit{alist1},\ \mathit{alist2}))$
$= \ \ \textbf{if}\ \text{definedp}\,(x,\ \mathit{alist1})\ \ \textbf{then}\ \text{assoc}\,(x,\ \mathit{alist1})$
$\quad\ \ \textbf{else}\ \text{assoc}\,(x,\ \mathit{alist2})\ \textbf{endif}$

THEOREM: get-append-1
$(n < \text{length}\,(a)) \rightarrow (\text{get}\,(n,\ \text{append}\,(a,\ b)) = \text{get}\,(n,\ a))$

THEOREM: get-append-2
$(n = \text{length}\,(a)) \rightarrow (\text{get}\,(n + k,\ \text{append}\,(a,\ b)) = \text{get}\,(k,\ b))$

```
; Because we have swapped the positions of the data and prog
; segments we have to add the following version of get-append-2.
; Unknownst to me, the original proof exploited the fact that
; "i-prog-segment" comes lexicographically before "i-usr-data-segment"
; so that when the segment-lengths of those two are commuted into
; order the order agreed with that used by link-mem, namely,
; the prog-segment comes first.
```

THEOREM: get-append-3
$((\mathit{n1} = \text{length}\,(\mathit{a1})) \land (\mathit{n2} = \text{length}\,(\mathit{a2})))$
$\rightarrow \ \ (\text{get}\,(\mathit{n2} + \mathit{n1} + k,\ \text{append}\,(\mathit{a1},\ \text{append}\,(\mathit{a2},\ b))) = \text{get}\,(k,\ b))$

THEOREM: length-link-area-is-length
$\text{length}\,(\text{link-area}\,(\mathit{lst},\ \mathit{table},\ \mathit{word\text{-}size})) = \text{length}\,(\mathit{lst})$

EVENT: Enable associativity-of-append.

THEOREM: get-link-area
$(n < \text{length}\,(\mathit{area}))$
$\rightarrow \ \ (\text{get}\,(n,\ \text{link-area}\,(\mathit{area},\ \mathit{table},\ \mathit{word\text{-}size}))$
$\qquad = \ \ \text{link-word}\,(\text{unlabel}\,(\text{get}\,(n,\ \mathit{area})),\ \mathit{table},\ \mathit{word\text{-}size}))$

DEFINITION:
link-table-entry (*name*, *segment*)
= **if** *segment* $\simeq$ **nil** **then** 0
elseif *name* = caar (*segment*) **then** 0
else length (cdar (*segment*))
+ link-table-entry (*name*, cdr (*segment*)) **endif**

DEFINITION:
label-table-entry (*lab*, *segment*)
= find-label (*lab*, cdr (assoc (car (*lab*), *segment*)))

THEOREM: assoc-link-table-for-segment
(definedp (*name*, *segment*) $\wedge$ (*addr0* $\in \mathbf{N}$))
$\rightarrow$ (cdr (assoc (*name*, link-table-for-segment (*segment*, *addr0*)))
= (*addr0* + link-table-entry (*name*, *segment*)))

THEOREM: lessp-absolute-address-segment-length-generalized
(definedp (*name*, *segment*) $\wedge$ (*offset* < length (cdr (assoc (*name*, *segment*)))))
$\rightarrow$ ((link-table-entry (*name*, *segment*) + *offset*)
< segment-length (*segment*))

```
; In anticipation of the various uses of this fact, we prove four
; instances...
```

THEOREM: lessp-absolute-address-segment-length-adp-name
(definedp (adp-name (*adp*), *segment*)
$\wedge$ (adp-offset (*adp*) < length (cdr (assoc (adp-name (*adp*), *segment*)))))
$\rightarrow$ ((link-table-entry (adp-name (*adp*), *segment*) + adp-offset (*adp*))
< segment-length (*segment*))

THEOREM: lessp-absolute-address-segment-length-cfp
(definedp (`'cstk`, i-sys-data-segment (*i*))
$\wedge$ (adp-offset (untag (i-cfp (*i*)))
< length (cdr (assoc (`'cstk`, i-sys-data-segment (*i*))))))
$\rightarrow$ ((link-table-entry (`'cstk`, i-sys-data-segment (*i*))
+ adp-offset (untag (i-cfp (*i*))))
< segment-length (i-sys-data-segment (*i*)))

THEOREM: lessp-absolute-address-segment-length-csp
(definedp (`'cstk`, i-sys-data-segment (*i*))
$\wedge$ (adp-offset (untag (i-csp (*i*)))
< length (cdr (assoc (`'cstk`, i-sys-data-segment (*i*))))))
$\rightarrow$ ((link-table-entry (`'cstk`, i-sys-data-segment (*i*))
+ adp-offset (untag (i-csp (*i*))))
< segment-length (i-sys-data-segment (*i*)))

THEOREM: lessp-absolute-address-segment-length-tsp
(definedp ('**tstk**, i-sys-data-segment (i))
 ∧ (adp-offset (untag (i-tsp (i)))
   < length (cdr (assoc ('**tstk**, i-sys-data-segment (i))))))
→ ((link-table-entry ('**tstk**, i-sys-data-segment (i))
   + adp-offset (untag (i-tsp (i))))
   < segment-length (i-sys-data-segment (i)))

EVENT: Enable length-append.

THEOREM: length-link-segment-is-segment-length
length (link-segment (*segment*, *table*, *word-size*)) = segment-length (*segment*)

EVENT: Disable commutativity-of-plus.

THEOREM: get-link-table-entry-link-segment
(definedp (adp-name (*adp*), *segment*)
 ∧ (adp-offset (*adp*) < length (cdr (assoc (adp-name (*adp*), *segment*)))))
→ (get (link-table-entry (adp-name (*adp*), *segment*) + adp-offset (*adp*),
     link-segment (*segment*, *table*, *word-size*))
   = link-word (unlabel (get (adp-offset (*adp*),
                 cdr (assoc (adp-name (*adp*), *segment*)))),
       table,
       word-size))

```
; The above lemma states the fundamental relationship between
; linking and fetching.

; At this point we have enough machinery to reduce any legal
; current-instruction fetch at the m level to a link-word of the
; i level.

; We can speed things up in the subsequent proofs if we package
; this knowledge up.  We do it by proving a fact about
; current-instruction.  We still need the foregoing machinery for
; those occasions when the m machine fetches data from program memory.
; But after this we will disable current-instruction and at least
; short circuit all this for the top level fetch for each instruction.


; We first make it possible to get and put past the boot-code in
; linked memory.  By doing it this way (proving length-boot-code to introduce
; fix and this disabling it when we are done) we prevent all future
; splits on whether load-addr is numeric.
```

THEOREM: length-boot-code
$\text{length}\,(\text{boot-code}\,(\mathit{lst},\ n,\ \mathit{word\text{-}size})) = \text{fix}\,(n)$

THEOREM: get-past-boot-code
$\text{get}\,(\mathit{load\text{-}addr} + n, \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),\ a))$
$= \quad \text{get}\,(n,\ a)$

```
; When the address in question has been incremented or decremented,
; as happens when it is a stack pointer, we need:
```

THEOREM: get-past-boot-code-add1
$\text{get}\,(1 + (\mathit{load\text{-}addr} + n), \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),\ a))$
$= \quad \text{get}\,(1 + n,\ a)$

THEOREM: get-past-boot-code-sub1
$(n \not\simeq 0)$
$\to \quad (\text{get}\,((\mathit{load\text{-}addr} + n) - 1,$
$\qquad \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),\ a))$
$\quad = \quad \text{get}\,(n - 1,\ a))$

THEOREM: put-past-boot-code
$\text{put}\,(\mathit{val},\ \mathit{load\text{-}addr} + n, \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),\ a))$
$= \quad \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}), \text{put}\,(\mathit{val},\ n,\ a))$

THEOREM: put-past-boot-code-add1
$\text{put}\,(\mathit{val},$
$\quad 1 + (\mathit{load\text{-}addr} + n),$
$\quad \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),\ a))$
$= \quad \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}), \text{put}\,(\mathit{val},\ 1 + n,\ a))$

THEOREM: put-past-boot-code-sub1
$(n \not\simeq 0)$
$\to \quad (\text{put}\,(\mathit{val},$
$\qquad (\mathit{load\text{-}addr} + n) - 1,$
$\qquad \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),\ a))$
$\quad = \quad \text{append}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size}),$
$\qquad\qquad \text{put}\,(\mathit{val},\ n - 1,\ a)))$

THEOREM: length-boot-code-linear
$(\mathit{load\text{-}addr} \not< \text{length}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size})))$
$\land \quad (\text{length}\,(\text{boot-code}\,(\mathit{boot\text{-}lst},\ \mathit{load\text{-}addr},\ \mathit{word\text{-}size})) \not< \mathit{load\text{-}addr})$

EVENT: Disable length-boot-code.

THEOREM: current-instruction-i->m

```
(i-state-types-okp (i)
 ∧   (adp-offset (untag (i-pc (i))) ∈ N)
 ∧   (adp-offset (untag (i-pc (i)))
       <   length (cdr (assoc (adp-name (untag (i-pc (i))), i-prog-segment (i)))))
 ∧   ((load-addr
        +   segment-length (i-prog-segment (i))
        +   segment-length (i-usr-data-segment (i))
        +   segment-length (i-sys-data-segment (i)))
       <   exp (2, 32))
 ∧   (i-word-size (i) = 32))
→   (current-instruction (list (r0,
                                r1,
                                r2,
                                r3,
                                r4,
                                r5,
                                r6,
                                r7,
                                r8,
                                r9,
                                r10,
                                r11,
                                r12,
                                r13,
                                r14,
                                nat-to-v (load-addr
                                          +   segment-length (i-usr-data-segment (i))
                                          +   link-table-entry (adp-name (untag (i-pc (i))),
                                                                i-prog-segment (i))
                                          +   adp-offset (untag (i-pc (i))),
                                          32)),
                             append (boot-code (boot-lst, load-addr, 32),
                                     append (link-segment (i-usr-data-segment (i),
                                                           i-link-tables (i,
                                                                          load-addr),
                                                           32),
                                             append (link-segment (i-prog-segment (i),
                                                                   i-link-tables (i,
                                                                                  load-addr),
                                                                   32),
                                                     data-mem))))
     =   link-word (unlabel (fetch (i-pc (i), i-prog-segment (i))),
                    i-link-tables (i, load-addr),
```

32))

Event: Disable current-instruction.

```
; The following section of this file documents about a half-day's
; work.

; I am headed for a simple theorem, namely that the i-psw of
; i-ins-step is 'run.  This follows immediately by opening up
; the -step fn for each i-level opcode and observing that the
; i-psw in each is 'run.  But there is much case analysis.
; To help out, I will disable -- in fact, hands-off -- every
; function used by any of the -step fns.  In addition, I provide
; the following accelerators:
```

Theorem: equal-i-psw-if-1
(i-psw (**if** *t1* **then** *b1*
else *b2* **endif**)
= 'run)
= **if** *t1* **then** i-psw (*b1*) = 'run
else i-psw (*b2*) = 'run **endif**

Theorem: equal-i-psw-if-2
(i-psw (**if** *t1* **then** *b1*
elseif *t2* **then** *b2*
else *b3* **endif**)
= 'run)
= **if** *t1* **then** i-psw (*b1*) = 'run
elseif *t2* **then** i-psw (*b2*) = 'run
else i-psw (*b3*) = 'run **endif**

Theorem: equal-i-psw-i-ins-step-run
i-psw (i-ins-step (*ins*, *i*, *load-addr*)) = 'run

```
; Note: The hint above was mechanically generated; it is the list of
; all function symbols occurring in the body of any i-xxx-step fn, minus
; i-psw.  We clearly do not wish to keep hands off of (i-psw (i-state ...)).

; During the morning I have proved this theorem many times.  Here are
; some observations.  I don't remember what happens if you try it without
; any preparation, but either it explodes into many cases or it gets
; stack overflow.  I first included just a (hands-off i-state) under
; the mistaken belief that it would prevent all the (i-state ...)'s from
```

```
; being rewritten.  But it does not prevent the args to the i-state being
; rewritten so I disabled all the fns listed above.  However, that still
; permitted rewrite rules to be applied to them and so I did a hands-off
; on them instead.

; More interesting is the use of the two lemmas, equal-i-psw-if-1 and -2.
; The first I call the "single stripper" and the second the "double stripper."

; With just the single stripper around you get a stack overflow.  That is
; because rewrite recurses several times for every application of the
; lemma and there are so many opcodes.  My first attempt to solve this
; was to prove a backchaining version of the single stripper.  It said
; "if the psw of b and c are both 'run then the psw of (if a b c) is too."
; That causes fewer recursions in the rewriter and permitted the proof
; to go through, but cost 8538 seconds!

; Then I hit upon the idea of the double stripper, which cuts in half the
; number of applications -- and this recursions -- necessary.  If the
; double stripper is available but the single stripper is not, the
; proof goes through but generates 30 cases and takes 354 seconds.

; Analysis of this showed that the problem is that some of our -step
; fns generate terms of the form (if & (i-state ...) (i-state ...)) and
; the double stripper could not be applied to these.  They get elevated
; to explicit cases.

; If both strippers are present, the proof goes through without any
; visible cases and takes 46 seconds.

; End of aside.

; In the one-way correspondence theorems I have the hypothesis
; that (i-step i load-addr) does not produce an error and then I make a
; statement about (i->m (i-step i load-addr) boot-lst load-addr).  The two occurrences of
; i-step bother me and the following theorem lets me replace the
; first with a conjunction of predicates.
```

THEOREM: i-psw-i-step-run
(i-psw (i-step (i, *load-addr*)) = **'run**)
= ((i-psw (i) = **'run**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ i-ins-okp (i-current-instruction (i), i))

THEOREM: i-link-tables-i-state

$$
\begin{array}{l}
((\text{link-table-for-segment}\,(\mathit{i\text{-}usr\text{-}data\text{-}segment},\ \mathit{load\text{-}addr}) \\
\quad = \quad \text{link-table-for-segment}\,(\text{i-usr-data-segment}\,(i),\ \mathit{load\text{-}addr})) \\
\ \wedge \quad (\text{segment-length}\,(\mathit{i\text{-}usr\text{-}data\text{-}segment}) \\
\qquad = \quad \text{segment-length}\,(\text{i-usr-data-segment}\,(i))) \\
\ \wedge \quad (\text{link-table-for-segment}\,(\mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{load\text{-}addr} \\
\qquad\qquad\qquad\qquad\qquad\qquad + \quad \text{segment-length}\,(\text{i-prog-segment}\,(i)) \\
\qquad\qquad\qquad\qquad\qquad\qquad + \quad \text{segment-length}\,(\text{i-usr-data-segment}\,(i))) \\
\qquad = \quad \text{link-table-for-segment}\,(\text{i-sys-data-segment}\,(i), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{load\text{-}addr} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad + \quad \text{segment-length}\,(\text{i-prog-segment}\,(i)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad + \quad \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))))) \\
\rightarrow \quad (\text{i-link-tables}\,(\text{i-state}\,(\mathit{pc}, \\
\qquad\qquad\qquad\qquad \mathit{cfp}, \\
\qquad\qquad\qquad\qquad \mathit{csp}, \\
\qquad\qquad\qquad\qquad \mathit{tsp}, \\
\qquad\qquad\qquad\qquad x, \\
\qquad\qquad\qquad\qquad y, \\
\qquad\qquad\qquad\qquad \mathit{c\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{v\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{n\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{z\text{-}flg}, \\
\qquad\qquad\qquad\qquad \text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{word\text{-}size}, \\
\qquad\qquad\qquad\qquad \mathit{psw}), \\
\qquad\qquad\qquad \mathit{load\text{-}addr}) \\
\qquad = \quad \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}))
\end{array}
$$

```
; Now I will embark on the proofs of each of one-way correspondence
; step theorems, starting with what I think is the simplest: the
; no-op case.
```

EVENT: Enable plus-add1.

THEOREM: get-add1-cons
$\text{get}\,(1 + n,\ \text{cons}\,(x,\ y)) = \text{get}\,(n,\ y)$

THEOREM: get-0-cons
$\text{get}\,(0,\ \text{cons}\,(x,\ y)) = x$

EVENT: Enable length-cons.

THEOREM: length-reg-file
$\text{length}\,(\text{list}\,(r0,\ r1,\ r2,\ r3,\ r4,\ r5,\ r6,\ r7,\ r8,\ r9,\ r10,\ r11,\ r12,\ r13,\ r14,\ r15)) = 16$

THEOREM: put-add1-cons
$\text{put}\,(val,\ 1 + n,\ \text{cons}\,(x,\ y)) = \text{cons}\,(x,\ \text{put}\,(val,\ n,\ y))$

THEOREM: put-zero-cons
$\text{put}\,(val,\ 0,\ \text{cons}\,(x,\ y)) = \text{cons}\,(val,\ y)$

THEOREM: boolp-bool-to-logical
$\text{boolp}\,(\text{bool-to-logical}\,(b))$

THEOREM: i-state-types-okp-property2
$$
\begin{array}{l}
\text{i-state-types-okp}\,(i) \\
\to\ \ ((\neg\ \text{icode-instructionp}\,(\text{i-pc}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-cfp}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-csp}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-tsp}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-c-flg}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-v-flg}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-n-flg}\,(i))) \\
\qquad \land\ \ (\neg\ \text{icode-instructionp}\,(\text{i-z-flg}\,(i))))
\end{array}
$$

THEOREM: not-icode-instructionp-tag
$\neg\ \text{icode-instructionp}\,(\text{tag}\,(type,\ obj))$

EVENT: Disable icode-instructionp.

THEOREM: length-put
$(n < \text{length}\,(lst)) \to (\text{length}\,(\text{put}\,(v,\ n,\ lst)) = \text{length}\,(lst))$

THEOREM: nat-to-v-equivalence
$$
\begin{array}{l}
((n \in \mathbf{N}) \\
\ \land\ \ (m \in \mathbf{N}) \\
\ \land\ \ (n < \exp\,(2,\ word\text{-}size)) \\
\ \land\ \ (m < \exp\,(2,\ word\text{-}size))) \\
\to\ \ ((\text{nat-to-v}\,(n,\ word\text{-}size) = \text{nat-to-v}\,(m,\ word\text{-}size)) = (n = m))
\end{array}
$$

```
; --- Here I take time out in my development to define a few
; functions used to make slides for my Piton talk.
```

DEFINITION:
list-of-bvs (lst)
$$
\begin{array}{l}
=\ \ \textbf{if}\ lst \simeq \textbf{nil}\ \ \textbf{then}\ \textbf{nil} \\
\qquad \textbf{else}\ \text{cons}\,(\text{nat-to-v}\,(\text{car}\,(lst),\ 32),\ \text{list-of-bvs}\,(\text{cdr}\,(lst)))\ \textbf{endif}
\end{array}
$$

DEFINITION:
n->m (n)
= m-state (list-of-bvs (car (n)),
cadr (n),
caddr (n),
cadddr (n),
caddddr (n),
list-of-bvs (cadddddr (n)))

DEFINITION:
m->n (m)
= list (list-of-nats (m-regs (m)),
m-c-flg (m),
m-v-flg (m),
m-n-flg (m),
m-z-flg (m),
list-of-nats (m-mem (m)))

DEFINITION: n-step (n) = m->n (m-step (n->m (n)))

```
; --- end of aside

; Recall



;(prove-lemma get-link-table-entry-link-segment (rewrite)
;  (implies (and (definedp (adp-name adp) segment)
;                (lessp (adp-offset adp)
;                       (length (cdr (assoc (adp-name adp) segment)))))
;           (equal (get (plus (link-table-entry (adp-name adp) segment)
;                             (adp-offset adp))
;                       (link-segment segment table word-size))
;                  (link-word
;                    (unlabel
;                      (get (adp-offset adp)
;                           (cdr (assoc (adp-name adp) segment))))
;                    table
;                    word-size))))
;

; from above.  Analogous lemmas for the data segment and the
; stack segment are needed.  In addition, we need to be able to handle
; the arithmetically equivalent cases that arise from opening plus
; or incrementing and decrementing addresses.  We'll do that first
```

```
; and then turn to the other segments and the add1/sub1 cases for them.
```

THEOREM: plus-zerop
$(y \notin \mathbf{N}) \rightarrow ((x + y) = \mathrm{fix}\,(x))$

THEOREM: get-link-table-entry-link-segment-add1
$$
\begin{array}{ll}
& (\mathrm{definedp}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment}) \\
\wedge & ((1 + \mathrm{adp\text{-}offset}\,(\mathit{adp})) < \mathrm{length}\,(\mathrm{cdr}\,(\mathrm{assoc}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment}))))) \\
\rightarrow & (\mathrm{get}\,(1 + (\mathrm{link\text{-}table\text{-}entry}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment}) + \mathrm{adp\text{-}offset}\,(\mathit{adp})), \\
& \quad \mathrm{link\text{-}segment}\,(\mathit{segment},\ \mathit{table},\ \mathit{word\text{-}size})) \\
& = \mathrm{link\text{-}word}\,(\mathrm{unlabel}\,(\mathrm{get}\,(1 + \mathrm{adp\text{-}offset}\,(\mathit{adp}), \\
& \qquad\qquad \mathrm{cdr}\,(\mathrm{assoc}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment})))), \\
& \qquad \mathit{table}, \\
& \qquad \mathit{word\text{-}size}))
\end{array}
$$

THEOREM: plus-sub1
$(y \not\simeq 0) \rightarrow ((x + (y - 1)) = ((x + y) - 1))$

```
;  The above lemma causes infinite looping if plus opens and commutes.
;  So I'll disable it asap.
```

THEOREM: get-link-table-entry-link-segment-sub1
$$
\begin{array}{ll}
& (\mathrm{definedp}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment}) \\
\wedge & ((\mathrm{adp\text{-}offset}\,(\mathit{adp}) - 1) \not\simeq 0) \\
\wedge & (\mathrm{adp\text{-}offset}\,(\mathit{adp}) < \mathrm{length}\,(\mathrm{cdr}\,(\mathrm{assoc}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment}))))) \\
\rightarrow & (\mathrm{get}\,((\mathrm{link\text{-}table\text{-}entry}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment}) + \mathrm{adp\text{-}offset}\,(\mathit{adp})) - 1, \\
& \quad \mathrm{link\text{-}segment}\,(\mathit{segment},\ \mathit{table},\ \mathit{word\text{-}size})) \\
& = \mathrm{link\text{-}word}\,(\mathrm{unlabel}\,(\mathrm{get}\,(\mathrm{adp\text{-}offset}\,(\mathit{adp}) - 1, \\
& \qquad\qquad \mathrm{cdr}\,(\mathrm{assoc}\,(\mathrm{adp\text{-}name}\,(\mathit{adp}),\ \mathit{segment})))), \\
& \qquad \mathit{table}, \\
& \qquad \mathit{word\text{-}size}))
\end{array}
$$

EVENT: Disable plus-sub1.

```
; Before we work on the two data segments we do some
; similar trivial arithmetic for the add1/sub1 cases of
; the previously proved:

;
;(prove-lemma lessp-absolute-address-segment-length-adp-name (rewrite)
;  (implies (and (definedp (adp-name adp) segment)
;                (lessp (adp-offset adp)
;                       (length (cdr (assoc (adp-name adp) segment)))))
```

```
;           (lessp (plus (link-table-entry (adp-name adp) segment)
;                        (adp-offset adp))
;                  (segment-length segment)))
;  ((use (lessp-absolute-address-segment-length-generalized
;          (name (adp-name adp))
;          (offset (adp-offset adp))))))
;
```

THEOREM: lessp-absolute-address-segment-length-adp-name-add1
(definedp (adp-name (adp), $segment$)
 ∧ (adp-offset (adp) < (length (cdr (assoc (adp-name (adp), $segment$))) − 1)))
→ ((1 + (link-table-entry (adp-name (adp), $segment$) + adp-offset (adp)))
 < segment-length ($segment$))

```
;  We will undoubtedly need related facts about 'cstk and 'tsk, similar
; to lessp-absolute-address-segment-length-cfp, etc., above.  But I
; don't know the exact form we'll need so I'll wait...  Here however
; are the related lemmas in their pure form
```

```
;(prove-lemma  (rewrite)
;  (implies (and (definedp 'cstk (i-sys-data-segment i))
;                (lessp (adp-offset (untag (i-cfp i)))
;                       (length (cdr (assoc 'cstk
;                                           (i-sys-data-segment i))))))
;           (lessp (plus (link-table-entry 'cstk
;                                          (i-sys-data-segment i))
;                        (adp-offset (untag (i-cfp i))))
;                  (segment-length (i-sys-data-segment i))))
;  ((use (lessp-absolute-address-segment-length-generalized
;          (name 'cstk)
;          (offset (adp-offset (untag (i-cfp i))))
;          (segment (i-sys-data-segment i))))))
;
;(prove-lemma lessp-absolute-address-segment-length-csp (rewrite)
;  (implies (and (definedp 'cstk (i-sys-data-segment i))
;                (lessp (adp-offset (untag (i-csp i)))
;                       (length (cdr (assoc 'cstk
;                                           (i-sys-data-segment i))))))
;           (lessp (plus (link-table-entry 'cstk
;                                          (i-sys-data-segment i))
```

```
;                       (adp-offset (untag (i-csp i))))
;                 (segment-length (i-sys-data-segment i))))
;  ((use (lessp-absolute-address-segment-length-generalized
;         (name 'cstk)
;         (offset (adp-offset (untag (i-csp i))))
;         (segment (i-sys-data-segment i))))))
;
;(prove-lemma lessp-absolute-address-segment-length-tsp (rewrite)
;  (implies (and (definedp 'tstk (i-sys-data-segment i))
;                (lessp (adp-offset (untag (i-tsp i)))
;                       (length (cdr (assoc 'tstk
;                                           (i-sys-data-segment i))))))
;          (lessp (plus (link-table-entry 'tstk
;                                          (i-sys-data-segment i))
;                       (adp-offset (untag (i-tsp i))))
;                 (segment-length (i-sys-data-segment i))))
;  ((use (lessp-absolute-address-segment-length-generalized
;         (name 'tstk)
;         (offset (adp-offset (untag (i-tsp i))))
;         (segment (i-sys-data-segment i))))))
```

```
; The previously proved:

;
;(prove-lemma get-append-2 (rewrite)
;  (implies (equal n (length a))
;          (equal (get (plus n k) (append a b))
;                 (get k b))))
;
; also needs to be expanded to the add1/sub1 cases:
```

THEOREM: get-append-2-add1
$(n = \text{length}\,(a)) \rightarrow (\text{get}\,(1 + (n + k), \text{append}\,(a,\, b)) = \text{get}\,(1 + k,\, b))$

THEOREM: get-append-2-sub1
$((n = \text{length}\,(a)) \wedge (k \not\simeq \mathsf{0}))$
$\rightarrow \quad (\text{get}\,((n + k) - 1, \text{append}\,(a,\, b)) = \text{get}\,(k - 1,\, b))$

THEOREM: get-append-3-add1
$((n1 = \text{length}\,(a1)) \wedge (n2 = \text{length}\,(a2)))$

$\rightarrow$ (get $(1 + (n2 + n1 + k)$, append $(a1$, append $(a2, b))) =$ get $(1 + k, b))$

THEOREM: get-append-3-sub1
$((n1 = \text{length}(a1)) \wedge (n2 = \text{length}(a2)) \wedge (k \not\simeq 0))$
$\rightarrow$ (get $((n2 + n1 + k) - 1$, append $(a1$, append $(a2, b))) =$ get $(k - 1, b))$

```
; Now it turns out we never increment or decrement addresses into the
; usr-data segment.  So we don't need lemmas for that.  But we do both
; operations on sys-data addresses when pushing and popping.  There
; is no difference between the usr-data and the sys-data accessing
; but it will turn out that we know that (adp-name (untag (i-csp i)))
; is 'cstk so we have to phrase the lemma explicitly for 'cstk and another
; one for tstk.

; Now for the sys-data segment, we just use the result above,
; but acknowledge that (adp-name adp) will be either 'cstk or 'tstk
; in all of our uses...
```

THEOREM: get-link-table-entry-link-segment-cstk
(definedp (`'cstk`, *segment*)
$\wedge$ (adp-offset (*adp*) $<$ length (cdr (assoc (`'cstk`, *segment*)))))
$\rightarrow$ (get (link-table-entry (`'cstk`, *segment*) + adp-offset (*adp*),
link-segment (*segment*, *table*, *word-size*))
= link-word (unlabel (get (adp-offset (*adp*),
cdr (assoc (`'cstk`, *segment*)))),
table,
word-size))

THEOREM: get-link-table-entry-link-segment-tstk
(definedp (`'tstk`, *segment*)
$\wedge$ (adp-offset (*adp*) $<$ length (cdr (assoc (`'tstk`, *segment*)))))
$\rightarrow$ (get (link-table-entry (`'tstk`, *segment*) + adp-offset (*adp*),
link-segment (*segment*, *table*, *word-size*))
= link-word (unlabel (get (adp-offset (*adp*),
cdr (assoc (`'tstk`, *segment*)))),
table,
word-size))

```
; and now, for both cstk and tstk, we get the pushed and popped
; versions.
```

THEOREM: get-link-table-entry-link-segment-cstk-add1
(definedp (`'cstk`, *segment*)

$$
\begin{array}{ll}
\wedge & (\text{adp-offset}\,(adp) < (\text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ segment))) - 1))) \\
\rightarrow & (\text{get}\,(1 + (\text{link-table-entry}\,(\texttt{'cstk},\ segment) + \text{adp-offset}\,(adp)), \\
& \quad \text{link-segment}\,(segment,\ table,\ word\text{-}size)) \\
& = \ \text{link-word}\,(\text{unlabel}\,(\text{get}\,(1 + \text{adp-offset}\,(adp), \\
& \qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ segment)))), \\
& \qquad\qquad table, \\
& \qquad\qquad word\text{-}size))
\end{array}
$$

THEOREM: get-link-table-entry-link-segment-cstk-sub1

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{definedp}\,(\texttt{'cstk},\ segment)} \\
\wedge & (\text{adp-offset}\,(adp) \not\simeq \texttt{0}) \\
\wedge & (\text{adp-offset}\,(adp) < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ segment))))) \\
\rightarrow & (\text{get}\,((\text{link-table-entry}\,(\texttt{'cstk},\ segment) + \text{adp-offset}\,(adp)) - 1, \\
& \quad \text{link-segment}\,(segment,\ table,\ word\text{-}size)) \\
& = \ \text{link-word}\,(\text{unlabel}\,(\text{get}\,(\text{adp-offset}\,(adp) - 1, \\
& \qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ segment)))), \\
& \qquad\qquad table, \\
& \qquad\qquad word\text{-}size))
\end{array}
$$

THEOREM: get-link-table-entry-link-segment-tstk-add1

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{definedp}\,(\texttt{'tstk},\ segment)} \\
\wedge & (\text{adp-offset}\,(adp) < (\text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ segment))) - 1))) \\
\rightarrow & (\text{get}\,(1 + (\text{link-table-entry}\,(\texttt{'tstk},\ segment) + \text{adp-offset}\,(adp)), \\
& \quad \text{link-segment}\,(segment,\ table,\ word\text{-}size)) \\
& = \ \text{link-word}\,(\text{unlabel}\,(\text{get}\,(1 + \text{adp-offset}\,(adp), \\
& \qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ segment)))), \\
& \qquad\qquad table, \\
& \qquad\qquad word\text{-}size))
\end{array}
$$

THEOREM: get-link-table-entry-link-segment-tstk-sub1

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{definedp}\,(\texttt{'tstk},\ segment)} \\
\wedge & (\text{adp-offset}\,(adp) \not\simeq \texttt{0}) \\
\wedge & (\text{adp-offset}\,(adp) < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ segment))))) \\
\rightarrow & (\text{get}\,((\text{link-table-entry}\,(\texttt{'tstk},\ segment) + \text{adp-offset}\,(adp)) - 1, \\
& \quad \text{link-segment}\,(segment,\ table,\ word\text{-}size)) \\
& = \ \text{link-word}\,(\text{unlabel}\,(\text{get}\,(\text{adp-offset}\,(adp) - 1, \\
& \qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ segment)))), \\
& \qquad\qquad table, \\
& \qquad\qquad word\text{-}size))
\end{array}
$$

```
; We now need for put all of the stuff we have developed for get...
```

THEOREM: put-append-1

$$(n < \text{length}\,(a)) \rightarrow (\text{put}\,(val,\ n,\ \text{append}\,(a,\ b)) = \text{append}\,(\text{put}\,(val,\ n,\ a),\ b))$$

THEOREM: put-append-2
$(n = \text{length}(a))$
$\rightarrow \quad (\text{put}(val,\ n + k,\ \text{append}(a,\ b)) = \text{append}(a,\ \text{put}(val,\ k,\ b)))$

THEOREM: put-append-3
$((n1 = \text{length}(a1)) \land (n2 = \text{length}(a2)))$
$\rightarrow \quad (\text{put}(val,\ n2 + n1 + k,\ \text{append}(a1,\ \text{append}(a2,\ b)))$
$\quad\quad = \quad \text{append}(a1,\ \text{append}(a2,\ \text{put}(val,\ k,\ b))))$

THEOREM: put-append-2-add1
$(n = \text{length}(a))$
$\rightarrow \quad (\text{put}(val,\ 1 + (n + k),\ \text{append}(a,\ b)) = \text{append}(a,\ \text{put}(val,\ 1 + k,\ b)))$

THEOREM: put-append-2-sub1
$((n = \text{length}(a)) \land (k \not\simeq 0))$
$\rightarrow \quad (\text{put}(val,\ (n + k) - 1,\ \text{append}(a,\ b)) = \text{append}(a,\ \text{put}(val,\ k - 1,\ b)))$

THEOREM: put-append-3-add1
$((n1 = \text{length}(a1)) \land (n2 = \text{length}(a2)))$
$\rightarrow \quad (\text{put}(val,\ 1 + (n2 + n1 + k),\ \text{append}(a1,\ \text{append}(a2,\ b)))$
$\quad\quad = \quad \text{append}(a1,\ \text{append}(a2,\ \text{put}(val,\ 1 + k,\ b))))$

THEOREM: put-append-3-sub1
$((n1 = \text{length}(a1)) \land (n2 = \text{length}(a2)) \land (k \not\simeq 0))$
$\rightarrow \quad (\text{put}(val,\ (n2 + n1 + k) - 1,\ \text{append}(a1,\ \text{append}(a2,\ b)))$
$\quad\quad = \quad \text{append}(a1,\ \text{append}(a2,\ \text{put}(val,\ k - 1,\ b))))$

THEOREM: put-link-area
$(n < \text{length}(lst))$
$\rightarrow \quad (\text{link-area}(\text{put}(val,\ n,\ lst),\ tables,\ word\text{-}size)$
$\quad\quad = \quad \text{put}(\text{link-word}(\text{unlabel}(val),\ tables,\ word\text{-}size),$
$\quad\quad\quad\quad n,$
$\quad\quad\quad\quad \text{link-area}(lst,\ tables,\ word\text{-}size)))$

```
; Now we do for put and link-segment what we did for get and link-segment:
```

THEOREM: put-link-segment
$(\text{definedp}(\text{adp-name}(adp),\ segment)$
$\land \quad (\text{adp-offset}(adp) < \text{length}(\text{cdr}(\text{assoc}(\text{adp-name}(adp),\ segment)))))$
$\rightarrow \quad (\text{link-segment}(\text{put-assoc}(\text{put}(val,$
$\quad\quad\quad\quad\quad\quad \text{adp-offset}(adp),$
$\quad\quad\quad\quad\quad\quad \text{cdr}(\text{assoc}(\text{adp-name}(adp),\ segment))),$
$\quad\quad\quad\quad\quad \text{adp-name}(adp),$
$\quad\quad\quad\quad\quad segment),$

$$
\begin{array}{l}
\qquad\qquad\qquad \textit{tables}, \\
\qquad\qquad\qquad \textit{word-size}) \\
\quad = \quad \text{put}\,(\text{link-word}\,(\text{unlabel}\,(\textit{val}),\ \textit{tables},\ \textit{word-size}), \\
\qquad\qquad\quad \text{link-table-entry}\,(\text{adp-name}\,(\textit{adp}),\ \textit{segment}) \\
\qquad\qquad\quad +\quad \text{adp-offset}\,(\textit{adp}), \\
\qquad\qquad\quad \text{link-segment}\,(\textit{segment},\ \textit{tables},\ \textit{word-size})))
\end{array}
$$

```
; And now we prove the cstk and tstk versions of the above, with
; the add1/sub1 variants.  These six lemmas, all just variations on
; the basic one above, should explain all of our deposits into the
; cstk and tstk areas.
```

THEOREM: put-link-segment-cstk

$$
\begin{array}{l}
(\text{definedp}\,(\texttt{'cstk},\ \textit{segment}) \\
\ \wedge\quad (\textit{stk-ptr} \in \mathbf{N}) \\
\ \wedge\quad (\textit{stk-ptr} < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment}))))) \\
\to\quad (\text{link-segment}\,(\text{put-assoc}\,(\text{put}\,(\textit{val},\ \textit{stk-ptr},\ \text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment}))), \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{'cstk}, \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{segment}), \\
\qquad\qquad\qquad \textit{tables}, \\
\qquad\qquad\qquad \textit{word-size}) \\
\quad = \quad \text{put}\,(\text{link-word}\,(\text{unlabel}\,(\textit{val}),\ \textit{tables},\ \textit{word-size}), \\
\qquad\qquad\quad \text{link-table-entry}\,(\texttt{'cstk},\ \textit{segment}) + \textit{stk-ptr}, \\
\qquad\qquad\quad \text{link-segment}\,(\textit{segment},\ \textit{tables},\ \textit{word-size})))
\end{array}
$$

THEOREM: put-link-segment-tstk

$$
\begin{array}{l}
(\text{definedp}\,(\texttt{'tstk},\ \textit{segment}) \\
\ \wedge\quad (\textit{stk-ptr} \in \mathbf{N}) \\
\ \wedge\quad (\textit{stk-ptr} < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ \textit{segment}))))) \\
\to\quad (\text{link-segment}\,(\text{put-assoc}\,(\text{put}\,(\textit{val},\ \textit{stk-ptr},\ \text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ \textit{segment}))), \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{'tstk}, \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{segment}), \\
\qquad\qquad\qquad \textit{tables}, \\
\qquad\qquad\qquad \textit{word-size}) \\
\quad = \quad \text{put}\,(\text{link-word}\,(\text{unlabel}\,(\textit{val}),\ \textit{tables},\ \textit{word-size}), \\
\qquad\qquad\quad \text{link-table-entry}\,(\texttt{'tstk},\ \textit{segment}) + \textit{stk-ptr}, \\
\qquad\qquad\quad \text{link-segment}\,(\textit{segment},\ \textit{tables},\ \textit{word-size})))
\end{array}
$$

THEOREM: put-link-segment-cstk-add1

$$
\begin{array}{l}
(\text{definedp}\,(\texttt{'cstk},\ \textit{segment}) \\
\ \wedge\quad ((1 + \textit{stk-ptr}) < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment}))))) \\
\to\quad (\text{link-segment}\,(\text{put-assoc}\,(\text{put}\,(\textit{val}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad 1 + \textit{stk-ptr}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment}))),
\end{array}
$$

'**cstk**,
segment),
tables,
word-size)
= put (link-word (unlabel (*val*), *tables*, *word-size*),
1 + (link-table-entry ('**cstk**, *segment*) + *stk-ptr*),
link-segment (*segment*, *tables*, *word-size*)))

THEOREM: put-link-segment-cstk-sub1
(definedp ('**cstk**, *segment*)
∧ (*stk-ptr* $\not\simeq$ 0)
∧ (*stk-ptr* < length (cdr (assoc ('**cstk**, *segment*)))))
→ (link-segment (put-assoc (put (*val*,
stk-ptr − 1,
cdr (assoc ('**cstk**, *segment*))),
'**cstk**,
segment),
tables,
word-size)
= put (link-word (unlabel (*val*), *tables*, *word-size*),
(link-table-entry ('**cstk**, *segment*) + *stk-ptr*) − 1,
link-segment (*segment*, *tables*, *word-size*)))

THEOREM: put-link-segment-tstk-add1
(definedp ('**tstk**, *segment*)
∧ ((1 + *stk-ptr*) < length (cdr (assoc ('**tstk**, *segment*)))))
→ (link-segment (put-assoc (put (*val*,
1 + *stk-ptr*,
cdr (assoc ('**tstk**, *segment*))),
'**tstk**,
segment),
tables,
word-size)
= put (link-word (unlabel (*val*), *tables*, *word-size*),
1 + (link-table-entry ('**tstk**, *segment*) + *stk-ptr*),
link-segment (*segment*, *tables*, *word-size*)))

THEOREM: put-link-segment-tstk-sub1
(definedp ('**tstk**, *segment*)
∧ (*stk-ptr* $\not\simeq$ 0)
∧ (*stk-ptr* < length (cdr (assoc ('**tstk**, *segment*)))))
→ (link-segment (put-assoc (put (*val*,
stk-ptr − 1,
cdr (assoc ('**tstk**, *segment*))),
'**tstk**,

$segment$),
$tables$,
$word\text{-}size$)
$=$ put (link-word (unlabel (val), $tables$, $word\text{-}size$),
(link-table-entry (`'tstk`, $segment$) + $stk\text{-}ptr$) − 1,
link-segment ($segment$, $tables$, $word\text{-}size$)))

THEOREM: definedp-put-assoc
definedp ($name$, put-assoc (val, $name2$, $segment$)) = definedp ($name$, $segment$)

THEOREM: link-table-entry-put-assoc
(length (val) = length (cdr (assoc ($name2$, $segment$))))
→ (link-table-entry ($name$, put-assoc (val, $name2$, $segment$))
= link-table-entry ($name$, $segment$))

THEOREM: link-table-for-segment-put-assoc
(length (val) = length (cdr (assoc ($name2$, $segment$))))
→ (link-table-for-segment (put-assoc (val, $name2$, $segment$), $addr$)
= link-table-for-segment ($segment$, $addr$))

THEOREM: equal-append
(append (a, b) = append (a, c)) = (b = c)

```
; We also need...
```

EVENT: Disable i-objectp.

THEOREM: unlabel-i-objectp
i-objectp (x, i) → (unlabel (x) = x)

THEOREM: unlabel-tagged-object
(type (x) ≠ `'dl`) → (unlabel (x) = x)

THEOREM: link-area-put-unlabel
i-objectp (unlabel (val), i)
→ (link-area (put (unlabel (val), n, lst), $tables$, $word\text{-}size$)
= link-area (put (val, n, lst), $tables$, $word\text{-}size$))

THEOREM: link-segment-put-unlabel
i-objectp (unlabel (val), i)
→ (link-segment (put-assoc (put (unlabel (val),
$offset$,
cdr (assoc ($name$, $segment$))),

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad name, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad segment), \\
\qquad\qquad\qquad\qquad\qquad tables, \\
\qquad\qquad\qquad\qquad\qquad word\text{-}size) \\
= \quad \text{link-segment}\,(\text{put-assoc}\,(\text{put}\,(val, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad offset, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(name,\ segment))), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad name, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad segment), \\
\qquad\qquad\qquad\qquad\qquad tables, \\
\qquad\qquad\qquad\qquad\qquad word\text{-}size))
\end{array}
$$

THEOREM: difference-1
$(x - \mathsf{1}) = (x - 1)$

EVENT: Disable int-to-v.

THEOREM: tc-add-is-iplus
$\text{tc-add}\,(x,\ y) = \text{iplus}\,(x,\ y)$

EVENT: Disable add.

THEOREM: tc-in-rangep-is-small-integerp
$(\text{integerp}\,(x) \land (word\text{-}size \not\simeq \mathsf{0}))$
$\rightarrow \quad (\text{tc-in-rangep}\,(x,\ word\text{-}size) = \text{small-integerp}\,(x,\ word\text{-}size))$

EVENT: Disable tc-in-rangep.

EVENT: Enable integerp-iplus.

THEOREM: small-integerp-implies-integerp
$\text{small-integerp}\,(x,\ word\text{-}size) \rightarrow \text{integerp}\,(x)$

THEOREM: bitn-of-positive-is-f
$(i < \exp\,(\mathsf{2},\ word\text{-}size - 1))$
$\rightarrow \quad (\text{bitn}\,(\text{nat-to-v}\,(i,\ word\text{-}size),\ word\text{-}size) = \mathbf{f})$

THEOREM: lessp-exp-sub1-exp
$(word\text{-}size \not\simeq \mathsf{0}) \rightarrow (\exp\,(\mathsf{2},\ word\text{-}size - 1) < \exp\,(\mathsf{2},\ word\text{-}size))$

THEOREM: incr-f
$\text{bvp}\,(b) \rightarrow (\text{incr}\,(\mathbf{f},\ b) = b)$

THEOREM: compl-compl
$\text{bvp}(b) \rightarrow (\text{v-not}(\text{v-not}(b)) = b)$

THEOREM: v-to-int-inverts-int-to-v-lemma1
$$
\begin{array}{ll}
((x \in \mathbf{N}) & \\
\ \wedge & (x \neq 0) \\
\ \wedge & (\exp(2,\ \textit{word-size} - 1) \not< x) \\
\ \wedge & (\textit{word-size} \neq 0) \\
\ \wedge & (\textit{word-size} \in \mathbf{N})) \\
\rightarrow & (\text{v-to-nat}(\text{incr}(\mathbf{t},\ \text{v-not}(\text{incr}(\mathbf{t},\ \text{v-not}(\text{nat-to-v}(x,\ \textit{word-size})))))) = x)
\end{array}
$$

THEOREM: bitn-nat-to-v-0
$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{bitn}(\text{nat-to-v}(0,\ k),\ i)} \\
= & \textbf{if } i \simeq 0 \textbf{ then f} \\
 & \textbf{elseif } k < i \textbf{ then } 0 \\
 & \textbf{else f endif}
\end{array}
$$

```
; In fm8502 days we proved
; (prove-lemma not-zerop-size-implies-not-equal-bv-btm (rewrite)
;   (implies (not (zerop (length x)))
;            (not (equal x (btm)))))
; but I'm going with equal-length-0 now:
```

EVENT: Enable equal-length-0.

THEOREM: bitn-compl
$((n \not\simeq 0) \wedge (\text{length}(b) \not< n)) \rightarrow (\text{bitn}(\text{v-not}(b),\ n) = (\neg\ \text{bitn}(b,\ n)))$

THEOREM: times-non-numberp
$(x \notin \mathbf{N}) \rightarrow ((y * x) = 0)$

EVENT: Enable difference-plus-cancellation.

THEOREM: remainder-times
$((x * y) \textbf{ mod } x) = 0$

THEOREM: remainder-exp-2-2
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\exp(2,\ x) \textbf{ mod } 2)} \\
= & \textbf{if } x \simeq 0 \textbf{ then } 1 \\
 & \textbf{else } 0 \textbf{ endif}
\end{array}
$$

THEOREM: my-quotient-times
$(x \not\simeq 0) \rightarrow (((x * y) \div x) = \text{fix}(y))$

THEOREM: quotient-exp-2-2
$(x \not\simeq 0) \rightarrow ((\exp(2,\ x) \div 2) = \exp(2,\ x - 1))$

THEOREM: bitn-exp-2-sub1-word-size
$(\mathit{word\text{-}size} \not\simeq 0)$
$\rightarrow \quad (\text{bitn}(\text{nat-to-v}(\exp(2,\ \mathit{word\text{-}size} - 1),\ \mathit{word\text{-}size}),\ \mathit{word\text{-}size}) = \mathbf{t})$

THEOREM: bitn-of-negatives-is-t-lemma
$((\mathit{word\text{-}size} \not\simeq 0) \land (x \in \mathbf{N}) \land (\exp(2,\ \mathit{word\text{-}size} - 1) \not< x))$
$\rightarrow \quad (\text{bitn}(\text{nat-to-v}(x,\ \mathit{word\text{-}size}),\ \mathit{word\text{-}size})$
$\quad = \quad (x = \exp(2,\ \mathit{word\text{-}size} - 1)))$

THEOREM: bitn-of-negatives-is-t-lemma-hyp
$((\mathit{word\text{-}size} \not\simeq 0) \land (\exp(2,\ \mathit{word\text{-}size}) \not< i))$
$\rightarrow \quad (\exp(2,\ \mathit{word\text{-}size} - 1) \not< (i \div 2))$

THEOREM: incr-f-is-noop
$\text{bitn}(\text{incr}(\mathbf{f},\ v),\ n)$
$= \quad$ **if** $\text{length}(v) < n$ **then** 0
elseif $\text{bitn}(v,\ n)$ **then t**
else f endif

EVENT: Disable plus-commutes2.

THEOREM: bitn-cons
$\text{bitn}(\text{cons}(b,\ v),\ i)$
$= \quad$ **if** $i \simeq 0$ **then f**
elseif $i = 1$ **then** b
else $\text{bitn}(v,\ i - 1)$ **endif**

THEOREM: bitn-of-negatives-is-t
$((i \not\simeq 0) \land (\mathit{word\text{-}size} \not\simeq 0) \land (\exp(2,\ \mathit{word\text{-}size} - 1) \not< i))$
$\rightarrow \quad (\text{bitn}(\text{incr}(\mathbf{t},\ \text{v-not}(\text{nat-to-v}(i,\ \mathit{word\text{-}size}))),\ \mathit{word\text{-}size}) = \mathbf{t})$

EVENT: Disable incr-f-is-noop.

EVENT: Disable bitn-cons.

EVENT: Disable bitn-of-negatives-is-t-lemma.

EVENT: Disable bitn-of-negatives-is-t-lemma-hyp.

THEOREM: v-to-int-inverts-int-to-v
$(\text{small-integerp}(i,\ \textit{word-size}) \land (\textit{word-size} \not\simeq 0))$
$\rightarrow\quad (\text{v-to-int}(\text{int-to-v}(i,\ \textit{word-size})) = i)$

THEOREM: v-to-int-inverts-link-word
$((\text{type}(x) = \texttt{'int})$
$\land\quad (\text{cddr}(x) = \textbf{nil})$
$\land\quad \text{small-integerp}(\text{untag}(x),\ \textit{word-size})$
$\land\quad (\textit{word-size} \not\simeq 0))$
$\rightarrow\quad (\text{v-to-int}(\text{link-word}(x,\ \textit{tables},\ \textit{word-size})) = \text{untag}(x))$

THEOREM: not-equal-link-word-btm
$\text{listp}(\text{link-word}(x,\ \textit{tables},\ 32))$

EVENT: Enable commutativity-of-iplus.

```
; Initially, my owc step theorems were stated in terms of i-ins-okp
; and i-step.  But now I use the -okp and -step function specific to
; the given current-instruction.  Frequently the attempt to prove a
; owc step theorem exposes bugs in the -okp or -step function for
; the instruction.  If the previously proved owc step theorems are
; stated in terms of the i-ins-okp and i-step those previously proved
; theorems must be proved again when the -okp and -step functions for
; the current instruction are redefined.

; Here I have inserted a new theorem, m-step1-opener, that play the
; role that m-step1-expander did in fm8502.
```

EVENT: Enable remainder-plus-x-x-2.

THEOREM: remainder-add1-plus-x-x-2
$((1 + (x + x)) \textbf{ mod } 2) = 1$

THEOREM: nat-to-v-inverts-v-to-nat
$\text{bvp}(v) \rightarrow (\text{nat-to-v}(\text{v-to-nat}(v),\ \text{length}(v)) = v)$

THEOREM: backchainer-trick
$(\text{bvp}(v) \land (\text{length}(v) \not< \textit{word-size}) \land (\text{v-to-nat}(v) = n))$
$\rightarrow\quad ((\text{firstn}(\textit{word-size},\ v) = \text{nat-to-v}(n,\ \textit{word-size})) = \textbf{t})$

THEOREM: v-adder-t-0-is-nat-to-v-add1
$((pc \in \mathbf{N}) \land (\textit{word-size} \not\simeq 0) \land ((1 + pc) < \exp(2,\ \textit{word-size})))$
$\rightarrow\quad (\text{firstn}(\textit{word-size},$
$\qquad\qquad \text{v-adder}(\textbf{t},\ \text{nat-to-v}(pc,\ \textit{word-size}),\ \text{nat-to-v}(0,\ \textit{word-size})))$
$\qquad = \quad \text{nat-to-v}(1 + pc,\ \textit{word-size}))$

THEOREM: v-adder-f-1-is-nat-to-v-sub1
$((pc \in \mathbf{N}) \land (pc \neq 0) \land (\textit{word-size} \not\simeq 0) \land (pc < \exp(2, \textit{word-size})))$
$\rightarrow$ (firstn (*word-size*,
v-adder (**f**,
v-not (nat-to-v (0, *word-size*)),
nat-to-v (pc, *word-size*)))
$=$ nat-to-v ($pc - 1$, *word-size*))

EVENT: Disable backchainer-trick.

EVENT: Disable nat-to-v-inverts-v-to-nat.

EVENT: Disable remainder-add1-plus-x-x-2.

EVENT: Disable remainder-plus-x-x-2.

THEOREM: v-inc-nat-to-v
$((i \in \mathbf{N}) \land (\textit{word-size} \not\simeq 0) \land ((1 + i) < \exp(2, \textit{word-size})))$
$\rightarrow$ (v-inc (nat-to-v (i, *word-size*)) = nat-to-v ($1 + i$, *word-size*))

EVENT: Disable v-inc.

EVENT: Enable bvp-nat-to-v.

THEOREM: v-dec-nat-to-v
$((i < \exp(2, \textit{word-size})) \land (\textit{word-size} \not\simeq 0))$
$\rightarrow$ (v-dec (nat-to-v (i, *word-size*))
$=$ **if** $i \simeq 0$ **then** nat-to-v (exp (2, *word-size*) $- 1$, *word-size*)
else nat-to-v ($i - 1$, *word-size*) **endif**)

EVENT: Disable v-dec.

THEOREM: m-step1-opener
$((pc \in \mathbf{N}) \land (\textit{word-size} \not\simeq 0) \land ((1 + pc) < \exp(2, \textit{word-size})))$
$\rightarrow$ (m-step1 (cons (*ins0*, *ins1-31*),
m-state (list (*r0*,
r1,
r2,
r3,
r4,
r5,

$r6$,
$r7$,
$r8$,
$r9$,
$r10$,
$r11$,
$r12$,
$r13$,
$r14$,
nat-to-v (pc, *word-size*)),
c,
v,
n,
z,
mem))
= m-operand-a (list ($r0$,
$r1$,
$r2$,
$r3$,
$r4$,
$r5$,
$r6$,
$r7$,
$r8$,
$r9$,
$r10$,
$r11$,
$r12$,
$r13$,
$r14$,
nat-to-v ($1 + pc$, *word-size*)),
c,
v,
n,
z,
mem,
cons (*ins0*, *ins1-31*)))

EVENT: Disable m-step1.

THEOREM: bvp-bitv-to-v
bvp (bitv-to-v (bv, *word-size*))

```
; A lemma bvp-int-to-v exists in fm9001.events, but it did not
```

```
; originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-bvp-int-to-v
bvp (int-to-v (i, *word-size*))

THEOREM: bvp-link-word
bvp (link-word (x, *link-tables*, *word-size*))

THEOREM: i-m-one-way-correspondence-step-move_x_x
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = 32)
$\land$ i-move_x_x-okp (i)
$\land$ (i-current-instruction (i) = `'(move_x_x)`))
$\rightarrow$ (i->m (i-move_x_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

```
; This next segment has been added during the fm9001 port, to speed up
; the processing of the new fns introduced there.

; I had this around in the fm9001 effort before backing up and
; disabling v-inc and v-dec...

;(prove-lemma v-adder-t-0-is-nat-to-v-add1-32 (rewrite)
;  (implies (and (numberp pc)
;                (lessp (add1 pc) (exp 2 32)))
;           (equal (firstn 32
;                          (v-adder t
;                                   (nat-to-v pc 32)
;                                   (LIST F F F F F F F F F F F F F F F F
;                                         F F F F F F F F F F F F F F F F)))
;                  (nat-to-v (add1 pc) 32)))
;  ((disable v-adder-t-0-is-nat-to-v-add1)
;   (use (v-adder-t-0-is-nat-to-v-add1 (word-size 32)))))

; The following theorem delays the opening of m-operand-a until the
; reg file has been put into normal form.  Note that this means we
; list all 16 regs and the pc is (NAT-TO-V & 32). Use of the free var
; regs in the hyp is arranged to save the repeated rewriting of it in
; the rhs of the concl.  The conclusion is just what (ppsd
; 'm-operand-a) prints, except for the list-expr in the regs slot on
; the lhs.
```

THEOREM: m-operand-a-opener

$$
\begin{array}{l}
(\mathit{regs} = \text{list}\,(\mathit{r0}, \mathit{r1}, \mathit{r2}, \mathit{r3}, \mathit{r4}, \mathit{r5}, \mathit{r6}, \mathit{r7}, \mathit{r8}, \mathit{r9}, \mathit{r10}, \mathit{r11}, \mathit{r12}, \mathit{r13}, \mathit{r14}, \text{nat-to-v}\,(\mathit{pc}, 32))) \\
\rightarrow \quad (\text{m-operand-a}\,(\text{list}\,(\mathit{r0}, \mathit{r1}, \mathit{r2}, \mathit{r3}, \mathit{r4}, \mathit{r5}, \mathit{r6}, \mathit{r7}, \mathit{r8}, \mathit{r9}, \mathit{r10}, \mathit{r11}, \mathit{r12}, \mathit{r13}, \mathit{r14}, \text{nat-to-v}\,(\mathit{pc}, 32)), \\
\qquad\qquad c, v, n, z, \mathit{mem}, \mathit{ins}) \\
\qquad = \quad \textbf{if}\ \text{a-immediate-p}\,(\mathit{ins}) \\
\qquad\qquad \textbf{then}\ \text{m-operand-b}\,(\mathit{regs}, c, v, n,
\end{array}
$$

```
                    z,
                    mem,
                    ins,
                    sign-extend (a-immediate (ins), 32))
elseif pre-dec-p (mode-a (ins))
then if reg-direct-p (mode-a (ins))
      then m-operand-b (update-v-nth (rn-a (ins),
                                     regs,
                                     v-dec (v-nth1 (rn-a (ins),
                                                    regs))),
                        c,
                        v,
                        n,
                        z,
                        mem,
                        ins,
                        v-nth1 (rn-a (ins), regs))
      else m-operand-b (update-v-nth (rn-a (ins),
                                     regs,
                                     v-dec (v-nth1 (rn-a (ins),
                                                    regs))),
                        c,
                        v,
                        n,
                        z,
                        mem,
                        ins,
                        v-nth1 (v-dec (v-nth1 (rn-a (ins),
                                               regs)),
                                mem)) endif
elseif post-inc-p (mode-a (ins))
then if reg-direct-p (mode-a (ins))
      then m-operand-b (update-v-nth (rn-a (ins),
                                     regs,
                                     v-inc (v-nth1 (rn-a (ins),
                                                    regs))),
                        c,
                        v,
                        n,
                        z,
                        mem,
                        ins,
                        v-nth1 (rn-a (ins), regs))
      else m-operand-b (update-v-nth (rn-a (ins),
```

```
                                            regs,
                                            v-inc (v-nth1 (rn-a (ins),
                                                                  regs))),
                                    c,
                                    v,
                                    n,
                                    z,
                                    mem,
                                    ins,
                                    v-nth1 (v-nth1 (rn-a (ins), regs), mem)) endif
      elseif reg-direct-p (mode-a (ins))
      then m-operand-b (regs, c, v, n, z, mem, ins, v-nth1 (rn-a (ins), regs))
      else m-operand-b (regs,
                        c,
                        v,
                        n,
                        z,
                        mem,
                        ins,
                        v-nth1 (v-nth1 (rn-a (ins), regs), mem)) endif)
```

EVENT: Disable m-operand-a.

```
; Now we repeat the exercise for m-operand-b.
```

```
THEOREM: m-operand-b-opener
 (regs = list (r0,
               r1,
               r2,
               r3,
               r4,
               r5,
               r6,
               r7,
               r8,
               r9,
               r10,
               r11,
               r12,
               r13,
               r14,
               nat-to-v (pc, 32)))
 →   (m-operand-b (list (r0,
```

```
                    r1,
                    r2,
                    r3,
                    r4,
                    r5,
                    r6,
                    r7,
                    r8,
                    r9,
                    r10,
                    r11,
                    r12,
                    r13,
                    r14,
                    nat-to-v (pc, 32)),
                c,
                v,
                n,
                z,
                mem,
                ins,
                operand-a)
=   if pre-dec-p (mode-b (ins))
    then if reg-direct-p (mode-b (ins))
         then m-alu-operation (update-v-nth (rn-b (ins),
                                             regs,
                                             v-dec (v-nth1 (rn-b (ins),
                                                            regs))),
                               c,
                               v,
                               n,
                               z,
                               mem,
                               ins,
                               operand-a,
                               v-nth1 (rn-b (ins), regs),
                               v-dec (v-nth1 (rn-b (ins), regs)))
         else m-alu-operation (update-v-nth (rn-b (ins),
                                             regs,
                                             v-dec (v-nth1 (rn-b (ins),
                                                            regs))),
                               c,
                               v,
                               n,
```

```
                                z,
                                mem,
                                ins,
                                operand-a,
                                v-nth1 (v-dec (v-nth1 (rn-b (ins),
                                                        regs)),
                                        mem),
                                v-dec (v-nth1 (rn-b (ins), regs))) endif
elseif post-inc-p (mode-b (ins))
then if reg-direct-p (mode-b (ins))
     then m-alu-operation (update-v-nth (rn-b (ins),
                                         regs,
                                         v-inc (v-nth1 (rn-b (ins),
                                                        regs))),
                           c,
                           v,
                           n,
                           z,
                           mem,
                           ins,
                           operand-a,
                           v-nth1 (rn-b (ins), regs),
                           v-nth1 (rn-b (ins), regs))
     else m-alu-operation (update-v-nth (rn-b (ins),
                                         regs,
                                         v-inc (v-nth1 (rn-b (ins),
                                                        regs))),
                           c,
                           v,
                           n,
                           z,
                           mem,
                           ins,
                           operand-a,
                           v-nth1 (v-nth1 (rn-b (ins), regs),
                                   mem),
                           v-nth1 (rn-b (ins), regs)) endif
elseif reg-direct-p (mode-b (ins))
then m-alu-operation (regs,
                      c,
                      v,
                      n,
                      z,
                      mem,
```

ins,
operand-a,
v-nth1 (rn-b (*ins*), *regs*),
v-nth1 (rn-b (*ins*), *regs*))
else m-alu-operation (*regs*,
c,
v,
n,
z,
mem,
ins,
operand-a,
v-nth1 (v-nth1 (rn-b (*ins*), *regs*), *mem*),
v-nth1 (rn-b (*ins*), *regs*)) **endif**)

EVENT: Disable m-operand-b.

```
; In watching expansions we have found that the following fact comes up
; and causes a split, which we wish to avoid.  Note that we have
; generalized (I-SYS-DATA-SEGMENT I) to I-SYS-DATA-SEGMENT so we can
; induct here.
```

THEOREM: non-zero-total-i-system-size
(adp-offset (untag (i-tsp (*i*)))
< length (cdr (assoc ('tstk, *i-sys-data-segment*))))
→ ((*load-addr*
+ segment-length (i-usr-data-segment (*i*))
+ segment-length (i-prog-segment (*i*))
+ segment-length (*i-sys-data-segment*))
≠ 0)

```
; And again for m-alu-operation...except this time we also eliminate the
; B-IF's in m-alu-operation (in favor of plain IFs, which we hypothesize
; have boolp args).
```

EVENT: Disable boolp.

```
; I am not sure if these two should be shut down or not.  But I will
; shut them down and we'll see...
```

EVENT: Disable zb.

EVENT: Disable n.

```
; To eliminate b-if without explosion, we have to be careful.
```

THEOREM: b-if-is-if

(boolp (a) $\wedge$ boolp (b))

$\rightarrow$ (b-if $(c,\ a,\ b)$

$\quad = \ $ **if** c **then** a

$\quad\quad$ **else** b **endif**)

EVENT: Disable b-if.

THEOREM: m-alu-operation-opener

(($regs$ = list ($r0$, $r1$, $r2$, $r3$, $r4$, $r5$, $r6$, $r7$, $r8$, $r9$, $r10$, $r11$, $r12$, $r13$, $r14$, pc))

$\wedge$ boolp (c)

$\wedge$ boolp (v)

$\wedge$ boolp (n)

$\wedge$ boolp (z)

$\wedge$ bvp ($operand\text{-}a$)

$\wedge$ (length ($operand\text{-}a$) $\neq$ 0))

$\rightarrow$ (m-alu-operation (list ($r0$,
$r1$,
$r2$,
$r3$,
$r4$,
$r5$,
$r6$,
$r7$,
$r8$,
$r9$,
$r10$,
$r11$,
$r12$,
$r13$,
$r14$,
pc),
c,
v,
n,
z,
mem,
ins,

```
                  operand-a,
                  operand-b,
                  b-address)
=   if m-store-resultp (store-cc (ins), c, v, n, z)
    then if reg-direct-p (mode-b (ins))
         then m-state (update-v-nth (rn-b (ins),
                                     regs,
                                     bv (v-alu (c,
                                                operand-a,
                                                operand-b,
                                                op-code (ins)))),
                       if c-set (set-flags (ins))
                       then c (v-alu (c,
                                      operand-a,
                                      operand-b,
                                      op-code (ins)))
                       else c endif,
                       if v-set (set-flags (ins))
                       then v (v-alu (c,
                                      operand-a,
                                      operand-b,
                                      op-code (ins)))
                       else v endif,
                       if n-set (set-flags (ins))
                       then n (v-alu (c,
                                      operand-a,
                                      operand-b,
                                      op-code (ins)))
                       else n endif,
                       if z-set (set-flags (ins))
                       then zb (v-alu (c,
                                       operand-a,
                                       operand-b,
                                       op-code (ins)))
                       else z endif,
                       mem)
         else m-state (regs,
                       if c-set (set-flags (ins))
                       then c (v-alu (c,
                                      operand-a,
                                      operand-b,
                                      op-code (ins)))
                       else c endif,
                       if v-set (set-flags (ins))
```

```
              then v (v-alu (c,
                             operand-a,
                             operand-b,
                             op-code (ins)))
              else v endif,
              if n-set (set-flags (ins))
              then n (v-alu (c,
                             operand-a,
                             operand-b,
                             op-code (ins)))
              else n endif,
              if z-set (set-flags (ins))
              then zb (v-alu (c,
                              operand-a,
                              operand-b,
                              op-code (ins)))
              else z endif,
              update-v-nth (b-address,
                            mem,
                            bv (v-alu (c,
                                       operand-a,
                                       operand-b,
                                       op-code (ins))))) endif
else m-state (regs,
          if c-set (set-flags (ins))
          then c (v-alu (c,
                         operand-a,
                         operand-b,
                         op-code (ins)))
          else c endif,
          if v-set (set-flags (ins))
          then v (v-alu (c,
                         operand-a,
                         operand-b,
                         op-code (ins)))
          else v endif,
          if n-set (set-flags (ins))
          then n (v-alu (c,
                         operand-a,
                         operand-b,
                         op-code (ins)))
          else n endif,
          if z-set (set-flags (ins))
          then zb (v-alu (c,
```

operand-a,
operand-b,
op-code (*ins*)))
else z **endif**,
mem) **endif**)

EVENT: Disable m-alu-operation.

THEOREM: i-m-one-way-correspondence-step-cpush_*
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-cpush_*-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(cpush_*)`))
$\rightarrow$ (i->m (i-cpush_*-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-add_<tsp>{i}_x{i}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-add_<tsp>{i}_x{i}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(add_<tsp>{i}_x{i})`))
$\rightarrow$ (i->m (i-add_<tsp>{i}_x{i}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-cpush_cfp
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-cpush_cfp-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(cpush_cfp)`))
$\rightarrow$ (i->m (i-cpush_cfp-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_cfp_csp
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-move_cfp_csp-okp (*i*)

∧ (i-current-instruction (i) = `'(move_cfp_csp)`))
→ (i->m (i-move_cfp_csp-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-cpush_<tsp>+
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-cpush_<tsp>+-okp (i)
∧ (i-current-instruction (i) = `'(cpush_<tsp>+)`))
→ (i->m (i-cpush_<tsp>+-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_csp_cfp
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-move_csp_cfp-okp (i)
∧ (i-current-instruction (i) = `'(move_csp_cfp)`))
→ (i->m (i-move_csp_cfp-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_x_tsp
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-move_x_tsp-okp (i)
∧ (i-current-instruction (i) = `'(move_x_tsp)`))
→ (i->m (i-move_x_tsp-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-cpop_cfp
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-cpop_cfp-okp (i)
∧ (i-current-instruction (i) = `'(cpop_cfp)`))
→ (i->m (i-cpop_cfp-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_x_*

$((\textit{load-addr} \in \mathbf{N})$
$\wedge$ i-state-okp $(i,\ \textit{load-addr})$
$\wedge$ (i-psw $(i) =$ `'run`)
$\wedge$ (i-word-size $(i) = 32$)
$\wedge$ i-move_x_*-okp (i)
$\wedge$ (i-current-instruction $(i) =$ `'(move_x_*)`))
$\rightarrow$ (i->m (i-move_x_*-step (i), *boot-lst*, *load-addr*)
$=$ m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_y_*
$((\textit{load-addr} \in \mathbf{N})$
$\wedge$ i-state-okp $(i,\ \textit{load-addr})$
$\wedge$ (i-psw $(i) =$ `'run`)
$\wedge$ (i-word-size $(i) = 32$)
$\wedge$ i-move_y_*-okp (i)
$\wedge$ (i-current-instruction $(i) =$ `'(move_y_*)`))
$\rightarrow$ (i->m (i-move_y_*-step (i), *boot-lst*, *load-addr*)
$=$ m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: icode-instructionp-data-word
(cddr $(x) =$ **nil**) $\rightarrow$ (icode-instructionp $(x) =$ **f**)

THEOREM: lessp-absolute-address-segment-length-rewrite
(definedp (*name*, *segment*) $\wedge$ (*offset* $<$ length (cdr (assoc (*name*, *segment*)))))
$\rightarrow$ ((link-table-entry (*name*, *segment*) $+$ *offset*)
$<$ segment-length (*segment*))

THEOREM: i-m-one-way-correspondence-step-add_x{n}_csp
$((\textit{load-addr} \in \mathbf{N})$
$\wedge$ i-state-okp $(i,\ \textit{load-addr})$
$\wedge$ (i-psw $(i) =$ `'run`)
$\wedge$ (i-word-size $(i) = 32$)
$\wedge$ i-add_x{n}_csp-okp (i)
$\wedge$ (i-current-instruction $(i) =$ `'(add_x{n}_csp)`))
$\rightarrow$ (i->m (i-add_x{n}_csp-step (i), *boot-lst*, *load-addr*)
$=$ m-step (i->m $(i$, *boot-lst*, *load-addr*)))

```
; To make the theorem below work, I had to change the -okp fn to
; check that the fetch of sys-addr i-x produced an i-objectp.
```

THEOREM: i-m-one-way-correspondence-step-move_x_<x{s}>
$((\textit{load-addr} \in \mathbf{N})$
$\wedge$ i-state-okp $(i,\ \textit{load-addr})$
$\wedge$ (i-psw $(i) =$ `'run`)

$\land$ (i-word-size (i) = `32`)
$\land$ i-move_x_<x{s}>-okp (i)
$\land$ (i-current-instruction (i) = **`'(move_x_<x{s}>)`**))
$\rightarrow$ (i->m (i-move_x_<x{s}>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_y_<y{s}>
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **`'run`**)
$\land$ (i-word-size (i) = `32`)
$\land$ i-move_y_<y{s}>-okp (i)
$\land$ (i-current-instruction (i) = **`'(move_y_<y{s}>)`**))
$\rightarrow$ (i->m (i-move_y_<y{s}>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_y_tsp
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **`'run`**)
$\land$ (i-word-size (i) = `32`)
$\land$ i-move_y_tsp-okp (i)
$\land$ (i-current-instruction (i) = **`'(move_y_tsp)`**))
$\rightarrow$ (i->m (i-move_y_tsp-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpush_csp
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **`'run`**)
$\land$ (i-word-size (i) = `32`)
$\land$ i-tpush_csp-okp (i)
$\land$ (i-current-instruction (i) = **`'(tpush_csp)`**))
$\rightarrow$ (i->m (i-tpush_csp-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpush_tsp
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **`'run`**)
$\land$ (i-word-size (i) = `32`)
$\land$ i-tpush_tsp-okp (i)
$\land$ (i-current-instruction (i) = **`'(tpush_tsp)`**))
$\rightarrow$ (i->m (i-tpush_tsp-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpush_x
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = 32)
$\land$ i-tpush_x-okp (i)
$\land$ (i-current-instruction (i) = `'(tpush_x)`))
$\rightarrow$ (i->m (i-tpush_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpush_<x{s}>
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = 32)
$\land$ i-tpush_<x{s}>-okp (i)
$\land$ (i-current-instruction (i) = `'(tpush_<x{s}>)`))
$\rightarrow$ (i->m (i-tpush_<x{s}>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpush_<x{a}>
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = 32)
$\land$ i-tpush_<x{a}>-okp (i)
$\land$ (i-current-instruction (i) = `'(tpush_<x{a}>)`))
$\rightarrow$ (i->m (i-tpush_<x{a}>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

```
; To make the theorem below work I had to change the -okp function
; to make sure that nextword was an i-objectp
```

THEOREM: i-m-one-way-correspondence-step-tpush_*
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = 32)
$\land$ i-tpush_*-okp (i)
$\land$ (i-current-instruction (i) = `'(tpush_*)`))
$\rightarrow$ (i->m (i-tpush_*-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

```
; To make the theorem below work I had to change the -okp function
```

```
; to make sure that the top of tsp was an i-objectp.
```

THEOREM: i-m-one-way-correspondence-step-tpop_x
(($load\text{-}addr \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-tpop_x-okp (i)
$\land$ (i-current-instruction (i) = `'(tpop_x)`))
$\rightarrow$ (i->m (i-tpop_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

```
; To make the theorem below work I had to change the -okp function
; to make sure that the top of tsp was an i-objectp.
```

THEOREM: i-m-one-way-correspondence-step-tpop_y
(($load\text{-}addr \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-tpop_y-okp (i)
$\land$ (i-current-instruction (i) = `'(tpop_y)`))
$\rightarrow$ (i->m (i-tpop_y-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpop_<x{s}>
(($load\text{-}addr \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-tpop_<x{s}>-okp (i)
$\land$ (i-current-instruction (i) = `'(tpop_<x{s}>)`))
$\rightarrow$ (i->m (i-tpop_<x{s}>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: segment-length-put-assoc
(length (cdr (assoc (*name*, *segment*))) = length (*val*))
$\rightarrow$ (segment-length (put-assoc (*val*, *name*, *segment*))
= segment-length (*segment*))

THEOREM: i-m-one-way-correspondence-step-tpop_<x{a}>
(($load\text{-}addr \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)

∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-tpop_<x{a}>-okp (*i*)
∧ (i-current-instruction (*i*) = `'(tpop_<x{a}>)`))
→ (i->m (i-tpop_<x{a}>-step (*i*), *boot-lst*, *load-addr*)
  = m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: car-assoc
definedp (*name*, *alist*) → (car (assoc (*name*, *alist*)) = *name*)

Theorem: assoc-link-table-for-prog-labels
((*addr0* ∈ **N**) ∧ definedp (*name*, *segment*))
→ (cdr (assoc (*name*, link-table-for-prog-labels (*segment*, *addr0*)))
  = link-table-for-labels (definiens (*name*, *segment*),
    addr0 + link-table-entry (*name*, *segment*)))

Theorem: assoc-link-table-for-labels
((*addr0* ∈ **N**) ∧ find-labelp (*label*, *lst*))
→ (cdr (assoc (*label*, link-table-for-labels (*lst*, *addr0*)))
  = (*addr0* + find-label (*label*, *lst*)))

Theorem: i-m-one-way-correspondence-step-jump_*
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-jump_*-okp (*i*)
∧ (i-current-instruction (*i*) = `'(jump_*)`))
→ (i->m (i-jump_*-step (*i*), *boot-lst*, *load-addr*)
  = m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: i-m-one-way-correspondence-step-cpop_pc
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-cpop_pc-okp (*i*)
∧ (i-current-instruction (*i*) = `'(cpop_pc)`))
→ (i->m (i-cpop_pc-step (*i*), *boot-lst*, *load-addr*)
  = m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: i-m-one-way-correspondence-step-move_<x{s}>_<tsp>
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)

∧ (i-word-size (i) = `32`)
∧ i-move_<x{s}>_<tsp>-okp (i)
∧ (i-current-instruction (i) = **'(move_<x{s}>_<tsp>)**))
→ (i->m (i-move_<x{s}>_<tsp>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_<x{a}>_<tsp>
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = **'run**)
∧ (i-word-size (i) = `32`)
∧ i-move_<x{a}>_<tsp>-okp (i)
∧ (i-current-instruction (i) = **'(move_<x{a}>_<tsp>)**))
→ (i->m (i-move_<x{a}>_<tsp>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-jump_x{subr}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = **'run**)
∧ (i-word-size (i) = `32`)
∧ i-jump_x{subr}-okp (i)
∧ (i-current-instruction (i) = **'(jump_x{subr})**))
→ (i->m (i-jump_x{subr}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: bool-to-logical-non-f
(x ≠ **'f**) → (bool-to-logical (x) = **t**)

THEOREM: i-m-one-way-correspondence-step-jump-n_x
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = **'run**)
∧ (i-word-size (i) = `32`)
∧ i-jump-n_x-okp (i)
∧ (i-current-instruction (i) = **'(jump-n_x)**))
→ (i->m (i-jump-n_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-jump-nn_x
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = **'run**)
∧ (i-word-size (i) = `32`)
∧ i-jump-nn_x-okp (i)

$\wedge$ (i-current-instruction (i) = `'(jump-nn_x)`))
$\rightarrow$ (i->m (i-jump-nn_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-jump-z_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp $(i$, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump-z_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump-z_x)`))
$\rightarrow$ (i->m (i-jump-z_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-jump-nz_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp $(i$, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump-nz_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump-nz_x)`))
$\rightarrow$ (i->m (i-jump-nz_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-tpop_pc
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp $(i$, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpop_pc-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpop_pc)`))
$\rightarrow$ (i->m (i-tpop_pc-step (i), *boot-lst*, *load-addr*)
= m-step (i->m $(i$, *boot-lst*, *load-addr*)))

```
; To make the theorem below work I had to change the -okp fn to
; insure that the top of tsp was an i-objectp.
```

THEOREM: i-m-one-way-correspondence-step-tpop_<c>_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp $(i$, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpop_<c>_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpop_<c>_x)`))

$\rightarrow$ (i->m (i-tpop_<c>_x-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

```
; To make the thm below work I had to redefine the -step function
; because it did not include an untag in the computation of the flag.

; A lemma v-nzerop-nat-to-v exists in fm9001.events, but it did not
; originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-v-nzerop-nat-to-v
(n < exp (2, *word-size*)) $\rightarrow$ (v-nzerop (nat-to-v (n, *word-size*)) = ($n \not\simeq 0$))

THEOREM: alu-thm_bitv-zb-move-15
(bvp (a) $\wedge$ bvp (b) $\wedge$ (length (a) = length (b)) $\wedge$ boolp (c))
$\rightarrow$ (zb (v-alu (c, a, b, list (**t**, **t**, **t**, **t**))) = ($\neg$ v-nzerop (a)))

THEOREM: i-m-one-way-correspondence-step-tpop{n}_<z>_y
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-tpop{n}_<z>_y-okp (i)
$\wedge$ (i-current-instruction (i) = **'(tpop{n}_<z>_y)**))
$\rightarrow$ (i->m (i-tpop{n}_<z>_y-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: size-int-to-v
length (int-to-v (i, *word-size*)) = fix (*word-size*)

```
; These seem irrelevant and I'm going to skip them for the moment...
; They may have v-nzerop counterparts...

;(prove-lemma v-to-nat-incr-t (rewrite)
;  (equal (v-to-nat (incr t b))
;         (if (all-onesp b)
;             0
;             (add1 (v-to-nat b)))))

;(prove-lemma all-onesp-compl (rewrite)
;  (equal (all-onesp (v-not b))
;         (equal (v-to-nat b) 0)))
```

THEOREM: zerop-v-to-nat-int-to-v
$((\textit{word-size} \not\simeq 0) \land \text{small-integerp}(i,\ \textit{word-size}))$
$\rightarrow \quad ((\text{v-to-nat}(\text{int-to-v}(i,\ \textit{word-size})) = 0) = (i = 0))$

```
; A lemma v-nzerop-int-to-v exists in fm9001.events, but it did not
; originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-v-nzerop-int-to-v
$((\textit{word-size} \not\simeq 0) \land \text{small-integerp}(i,\ \textit{word-size}))$
$\rightarrow \quad (\text{v-nzerop}(\text{int-to-v}(i,\ \textit{word-size})) = (i \neq 0))$

THEOREM: v-negp-nat-to-v
$((w \not\simeq 0) \land (n < \exp(2,\ w)))$
$\rightarrow \quad (\text{v-negp}(\text{nat-to-v}(n,\ w)) = (n \not< \exp(2,\ w - 1)))$

THEOREM: v-negp-int-to-v
$((\textit{word-size} \not\simeq 0) \land \text{small-integerp}(i,\ \textit{word-size}))$
$\rightarrow \quad (\text{v-negp}(\text{int-to-v}(i,\ \textit{word-size})) = \text{negativep}(i))$

EVENT: Disable v-negp-nat-to-v.

```
; I had to change the -step fn below because it failed to set y!
```

THEOREM: i-m-one-way-correspondence-step-tpop{i}_<zn>_y
$((\textit{load-addr} \in \mathbf{N})$
$\land \quad \text{i-state-okp}(i,\ \textit{load-addr})$
$\land \quad (\text{i-psw}(i) = \texttt{'run})$
$\land \quad (\text{i-word-size}(i) = 32)$
$\land \quad \text{i-tpop\{i\}\_<zn>\_y-okp}(i)$
$\land \quad (\text{i-current-instruction}(i) = \texttt{'(tpop\{i\}\_<zn>\_y)}))$
$\rightarrow \quad (\text{i->m}(\text{i-tpop\{i\}\_<zn>\_y-step}(i),\ \textit{boot-lst},\ \textit{load-addr})$
$\quad = \quad \text{m-step}(\text{i->m}(i,\ \textit{boot-lst},\ \textit{load-addr})))$

THEOREM: i-m-one-way-correspondence-step-tpop{b}_<z>_y
$((\textit{load-addr} \in \mathbf{N})$
$\land \quad \text{i-state-okp}(i,\ \textit{load-addr})$
$\land \quad (\text{i-psw}(i) = \texttt{'run})$
$\land \quad (\text{i-word-size}(i) = 32)$
$\land \quad \text{i-tpop\{b\}\_<z>\_y-okp}(i)$
$\land \quad (\text{i-current-instruction}(i) = \texttt{'(tpop\{b\}\_<z>\_y)}))$
$\rightarrow \quad (\text{i->m}(\text{i-tpop\{b\}\_<z>\_y-step}(i),\ \textit{boot-lst},\ \textit{load-addr})$
$\quad = \quad \text{m-step}(\text{i->m}(i,\ \textit{boot-lst},\ \textit{load-addr})))$

THEOREM: all-zero-bitvp-not-v-nzerop-bitv-to-v
all-zero-bitvp (b) = ($\neg$ v-nzerop (bitv-to-v (b, length (b))))

THEOREM: length-bit-vectorp
bit-vectorp (b, *word-size*) $\rightarrow$ (length (b) = fix (*word-size*))

THEOREM: i-m-one-way-correspondence-step-tpop{v}_<z>_y
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpop{v}_<z>_y-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpop{v}_<z>_y)`))
$\rightarrow$ (i->m (i-tpop{v}_<z>_y-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

```
; For the next owc lemma it would be useful to have the
; commutativity of plus, but providing it messes up our
; rewrite rules.  We have implicitly assumed we had total
; knowledge of the shape of plus-nests.  The following rule,
; which is elegant in its own right, elevates the two
; plus expressions to the point where linear arithmetic
; can deal with them.
```

THEOREM: put-equal
(put (*val1*, n, *lst*) = put (*val2*, n, *lst*)) = (*val1* = *val2*)

THEOREM: i-m-one-way-correspondence-step-add_<tsp>{a}_x{n}
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-add_<tsp>{a}_x{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_<tsp>{a}_x{n})`))
$\rightarrow$ (i->m (i-add_<tsp>{a}_x{n}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-add_tsp_*{n}
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-add_tsp_*{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_tsp_*{n})`))
$\rightarrow$ (i->m (i-add_tsp_*{n}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

Theorem: i-m-one-way-correspondence-step-add_tsp_x{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ (i-word-size (*i*) = `32`)
$\wedge$ i-add_tsp_x{n}-okp (*i*)
$\wedge$ (i-current-instruction (*i*) = `'(add_tsp_x{n})`))
$\rightarrow$ (i->m (i-add_tsp_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: i-m-one-way-correspondence-step-sub_<tsp>{a}_x{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ (i-word-size (*i*) = `32`)
$\wedge$ i-sub_<tsp>{a}_x{n}-okp (*i*)
$\wedge$ (i-current-instruction (*i*) = `'(sub_<tsp>{a}_x{n})`))
$\rightarrow$ (i->m (i-sub_<tsp>{a}_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: i-m-one-way-correspondence-step-sub_x{s}_y{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ (i-word-size (*i*) = `32`)
$\wedge$ i-sub_x{s}_y{n}-okp (*i*)
$\wedge$ (i-current-instruction (*i*) = `'(sub_x{s}_y{n})`))
$\rightarrow$ (i->m (i-sub_x{s}_y{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: all-zero-bitvp-xor-bitv
(bit-vectorp (a, *word-size*) $\wedge$ bit-vectorp (b, *word-size*))
$\rightarrow$ (all-zero-bitvp (xor-bitv (a, b)) = (a = b))

Theorem: equal-append-generalized
(properp (a) $\wedge$ properp (b) $\wedge$ (length (a) = length (b)))
$\rightarrow$ ((append (a, x) = append (b, y)) = ((a = b) $\wedge$ (x = y)))

Theorem: properp-v-to-bitv
properp (v-to-bitv (x))

Theorem: length-v-to-bitv
length (v-to-bitv (a)) = length (a)

Theorem: equal-v-to-bitv
(bvp (x) $\wedge$ bvp (y) $\wedge$ (length (x) = length (y)))
$\rightarrow$ ((v-to-bitv (x) = v-to-bitv (y)) = (x = y))

THEOREM: bit-vectorp-append-singleton
$(\text{bitp}(bit) \land \text{properp}(a))$
$\rightarrow \quad (\text{bit-vectorp}(\text{append}(a, \text{list}(bit)), n)$
$\quad = \quad \textbf{if } n \simeq 0 \textbf{ then f}$
$\quad\quad \textbf{else } \text{bit-vectorp}(a, n-1) \textbf{ endif})$

DEFINITION:
$\text{bit-vectorp-v-to-bitv-hint}(a, \textit{word-size})$
$= \quad \textbf{if } (\textit{word-size} \simeq 0) \lor (\neg \text{bvp}(a)) \lor (a \simeq \textbf{nil}) \textbf{ then t}$
$\quad \textbf{else } \text{bit-vectorp-v-to-bitv-hint}(\text{cdr}(a), \textit{word-size} - 1) \textbf{ endif}$

THEOREM: bit-vectorp-v-to-bitv-lemma
$\text{bit-vectorp}(\text{v-to-bitv}(a), \text{length}(a))$

THEOREM: bit-vectorp-v-to-bitv
$\text{bit-vectorp}(\text{v-to-bitv}(a), \textit{word-size}) = (\text{length}(a) = \text{fix}(\textit{word-size}))$

THEOREM: bit-vectorp-v-to-bitv-link-data-word
$\text{bit-vectorp}(\text{v-to-bitv}(\text{link-data-word}(x, \textit{tables}, \textit{word-size})), \textit{word-size})$

THEOREM: equal-int-to-v
$((\textit{word-size} \not\simeq 0)$
$\land \quad \text{small-integerp}(i, \textit{word-size})$
$\land \quad \text{small-integerp}(j, \textit{word-size}))$
$\rightarrow \quad ((\text{int-to-v}(i, \textit{word-size}) = \text{int-to-v}(j, \textit{word-size})) = (i = j))$

THEOREM: v-to-bitv-inverts-bitv-to-v
$\text{bit-vectorp}(x, \textit{word-size}) \rightarrow (\text{v-to-bitv}(\text{bitv-to-v}(x, \textit{word-size})) = x)$

THEOREM: equal-bitv-to-v
$(\text{bit-vectorp}(x, \textit{word-size}) \land \text{bit-vectorp}(y, \textit{word-size}))$
$\rightarrow \quad ((\text{bitv-to-v}(x, \textit{word-size}) = \text{bitv-to-v}(y, \textit{word-size})) = (x = y))$

THEOREM: v-to-bool-inverts-bool-to-v
$((\textit{word-size} \not\simeq 0) \land \text{booleanp}(x))$
$\rightarrow \quad (\text{v-to-bool}(\text{bool-to-v}(x, \textit{word-size})) = x)$

THEOREM: equal-bool-to-v
$((\textit{word-size} \not\simeq 0) \land \text{booleanp}(x) \land \text{booleanp}(y))$
$\rightarrow \quad ((\text{bool-to-v}(x, \textit{word-size}) = \text{bool-to-v}(y, \textit{word-size})) = (x = y))$

```
; We now turn to the harder cases of linking, those involving
; the link tables.

; Three of the linkers, namely addr-to-v, sys-addr-to-v, and
; ipc-to-v, use the primitive function absolute-address to
; compute their values.  We therefore first establish that
; absolute-address can be inverted on the link table actually used.
```

THEOREM: cons-adp-name-adp-offset
listp (adp) $\rightarrow$ (cons (adp-name (adp), adp-offset (adp)) = adp)

THEOREM: listp-link-table-for-segment
listp (link-table-for-segment ($segment$, $addr0$)) = listp ($segment$)

THEOREM: cdar-link-table-for-segment
listp ($segment$) $\rightarrow$ (cdar (link-table-for-segment ($segment$, $addr0$)) = $addr0$)

THEOREM: find-containing-area-name-link-table-for-segment
adpp (adp, $segment$)
$\rightarrow$ (find-containing-area-name (cdr (assoc (adp-name (adp),
link-table-for-segment ($segment$,
$addr0$)))
+ adp-offset (adp),
link-table-for-segment ($segment$, $addr0$))
= adp-name (adp))

THEOREM: invert-absolute-address-inverts-absolute-address
adpp (adp, $segment$)
$\rightarrow$ (invert-absolute-address (absolute-address (adp,
link-table-for-segment ($segment$,
$addr0$)),
link-table-for-segment ($segment$, $addr0$))
= adp)

```
; Now I prove the invertibility results and the equality results for
; each of the three linkers that use absolute-address:  ipc-to-v,
; addr-to-v, and sys-addr-to-v.

; Each linker is used with its own link table.  In fact, if we could
; completely characterize the link table properties we could avoid
; having different theorems for these three linkers and just have a
; general one.  Rather than do that I just prove the appropriate
; result for the link table used by each linker.

; In fact we prove three lemmas for each linker.  The first establishes
; that the absolute address computed for a given link table is small
; enough.  The second is the invertibility result for the link table.
; The third is the equality result.

; Here is the series for prog-segment addresses.

; The following lemma would not be necessary if absolute-address could be
; enabled in the lemma below this.  But the invertibility of absolute-address
```

```
; above requires that the fn be disabled.

; The (PLUS LOAD-ADDR ...) term below is just (i-loadablep i load-addr).
; But it is open because it contains the free-var load-addr and will be
; the term we look for to bind that var.
```

THEOREM: lessp-absolute-address-ipc

$$
\begin{array}{l}
(\text{adpp}\,(adp,\ \text{i-prog-segment}\,(i)) \\
\ \wedge\ ((load\text{-}addr \\
\qquad +\ \text{segment-length}\,(\text{i-usr-data-segment}\,(i)) \\
\qquad +\ \text{segment-length}\,(\text{i-prog-segment}\,(i)) \\
\qquad +\ \text{segment-length}\,(\text{i-sys-data-segment}\,(i))) \\
\qquad <\ \exp\,(2,\ \text{i-word-size}\,(i)))) \\
\to\ (\text{absolute-address}\,(adp, \\
\qquad\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad load\text{-}addr + \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))) \\
\qquad <\ \exp\,(2,\ \text{i-word-size}\,(i)))
\end{array}
$$

```
; To prove the invertibility result for ipcs and keep it exactly
; symmetric with addrs and sys-addrs, it is necessary to define
; the inverter.  The inverter is not used in the main theorem.
```

DEFINITION:

$$
\begin{array}{l}
\text{v-to-ipc}\,(bv,\ prog\text{-}links) \\
=\ \text{invert-absolute-address}\,(\text{v-to-nat}\,(bv),\ prog\text{-}links)
\end{array}
$$

THEOREM: v-to-ipc-inverts-ipc-to-v

$$
\begin{array}{l}
(\text{adpp}\,(adp,\ \text{i-prog-segment}\,(i)) \wedge \text{i-loadablep}\,(i,\ load\text{-}addr)) \\
\to\ (\text{v-to-ipc}\,(\text{ipc-to-v}\,(adp, \\
\qquad\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad load\text{-}addr + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))), \\
\qquad\qquad \text{i-word-size}\,(i)), \\
\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad load\text{-}addr + \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))) \\
\qquad =\ adp)
\end{array}
$$

THEOREM: equal-ipc-to-v

$$
\begin{array}{l}
(\text{adpp}\,(adp1,\ \text{i-prog-segment}\,(i)) \\
\ \wedge\ \text{adpp}\,(adp2,\ \text{i-prog-segment}\,(i)) \\
\ \wedge\ \text{i-loadablep}\,(i,\ load\text{-}addr)) \\
\to\ ((\text{ipc-to-v}\,(adp1,
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\text{link-table-for-segment}\,(\text{i-prog-segment}\,(i), \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))), \\
\quad\quad\text{i-word-size}\,(i)) \\
= \quad \text{ipc-to-v}\,(\mathit{adp2}, \\
\quad\quad\quad\text{link-table-for-segment}\,(\text{i-prog-segment}\,(i), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))), \\
\quad\quad\quad\text{i-word-size}\,(i))) \\
= \quad (\mathit{adp1} = \mathit{adp2}))
\end{array}
$$

```
; So now we repeat the exercise for ADDRs.  That is, we use the usr-data
; segment and its link table...
```

THEOREM: lessp-absolute-address-usr-data

$$
\begin{array}{l}
(\text{adpp}\,(\mathit{adp}, \text{i-usr-data-segment}\,(i)) \\
\land \quad ((\mathit{load\text{-}addr} \\
\quad\quad + \quad \text{segment-length}\,(\text{i-usr-data-segment}\,(i)) \\
\quad\quad + \quad \text{segment-length}\,(\text{i-prog-segment}\,(i)) \\
\quad\quad + \quad \text{segment-length}\,(\text{i-sys-data-segment}\,(i))) \\
\quad\quad < \quad \exp\,(2, \text{i-word-size}\,(i))) \\
\land \quad (\mathit{load\text{-}addr} \in \mathbf{N})) \\
\rightarrow \quad (\text{absolute-address}\,(\mathit{adp}, \\
\quad\quad\quad\quad\quad \text{link-table-for-segment}\,(\text{i-usr-data-segment}\,(i), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{load\text{-}addr})) \\
\quad\quad < \quad \exp\,(2, \text{i-word-size}\,(i)))
\end{array}
$$

THEOREM: v-to-addr-inverts-addr-to-v

$$
\begin{array}{l}
((\mathit{load\text{-}addr} \in \mathbf{N}) \\
\land \quad \text{adpp}\,(\mathit{adp}, \text{i-usr-data-segment}\,(i)) \\
\land \quad \text{i-loadablep}\,(i, \mathit{load\text{-}addr})) \\
\rightarrow \quad (\text{v-to-addr}\,(\text{addr-to-v}\,(\mathit{adp}, \\
\quad\quad\quad\quad\quad\quad \text{link-table-for-segment}\,(\text{i-usr-data-segment}\,(i), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{load\text{-}addr}), \\
\quad\quad\quad\quad\quad\quad \text{i-word-size}\,(i)), \\
\quad\quad\quad\quad \text{link-table-for-segment}\,(\text{i-usr-data-segment}\,(i), \mathit{load\text{-}addr})) \\
\quad\quad = \quad \mathit{adp})
\end{array}
$$

THEOREM: equal-addr-to-v

$$
\begin{array}{l}
((\mathit{load\text{-}addr} \in \mathbf{N}) \\
\land \quad \text{adpp}\,(\mathit{adp1}, \text{i-usr-data-segment}\,(i)) \\
\land \quad \text{adpp}\,(\mathit{adp2}, \text{i-usr-data-segment}\,(i)) \\
\land \quad \text{i-loadablep}\,(i, \mathit{load\text{-}addr})) \\
\rightarrow \quad ((\text{addr-to-v}\,(\mathit{adp1}, \\
\quad\quad\quad\quad \text{link-table-for-segment}\,(\text{i-usr-data-segment}\,(i), \mathit{load\text{-}addr}), \\
\quad\quad\quad\quad \text{i-word-size}\,(i))
\end{array}
$$

$=$ addr-to-v (*adp2*,
link-table-for-segment (i-usr-data-segment (i),
load-addr),
i-word-size (i)))
$=$ (*adp1* $=$ *adp2*))

```
; And finally we do the SYS-ADDR type...
```

THEOREM: lessp-absolute-address-sys-data
(adpp (*adp*, i-sys-data-segment (i))
$\land$ ((*load-addr*
$+$ segment-length (i-usr-data-segment (i))
$+$ segment-length (i-prog-segment (i))
$+$ segment-length (i-sys-data-segment (i)))
$<$ exp (2, i-word-size (i))))
$\rightarrow$ (absolute-address (*adp*,
link-table-for-segment (i-sys-data-segment (i),
load-addr
$+$ segment-length (i-usr-data-segment (i))
$+$ segment-length (i-prog-segment (i))))
$<$ exp (2, i-word-size (i)))

THEOREM: v-to-sys-addr-inverts-sys-addr-to-v
(adpp (*adp*, i-sys-data-segment (i)) $\land$ i-loadablep (i, *load-addr*))
$\rightarrow$ (v-to-sys-addr (sys-addr-to-v (*adp*,
link-table-for-segment (i-sys-data-segment (i),
load-addr
$+$ segment-length (i-usr-data-segment (i))
$+$ segment-length (i-prog-segment (i))),
i-word-size (i)),
link-table-for-segment (i-sys-data-segment (i),
load-addr
$+$ segment-length (i-usr-data-segment (i))
$+$ segment-length (i-prog-segment (i))))
$=$ *adp*)

THEOREM: equal-sys-addr-to-v
(adpp (*adp1*, i-sys-data-segment (i))
$\land$ adpp (*adp2*, i-sys-data-segment (i))
$\land$ i-loadablep (i, *load-addr*))
$\rightarrow$ ((sys-addr-to-v (*adp1*,
link-table-for-segment (i-sys-data-segment (i),
load-addr
$+$ segment-length (i-prog-segment (i))

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad + \ \text{segment-length}\,(\text{i-usr-data-segment}\,(i))),\\
&\qquad \text{i-word-size}\,(i))\\
= \ & \text{sys-addr-to-v}\,(\mathit{adp2},\\
&\quad \text{link-table-for-segment}\,(\text{i-sys-data-segment}\,(i),\\
&\qquad\qquad \mathit{load\text{-}addr}\\
&\qquad\qquad + \ \text{segment-length}\,(\text{i-prog-segment}\,(i))\\
&\qquad\qquad + \ \text{segment-length}\,(\text{i-usr-data-segment}\,(i))),\\
&\quad \text{i-word-size}\,(i)))\\
= \ & (\mathit{adp1} = \mathit{adp2}))
\end{aligned}
$$

```
; Now I turn to the SUBR case.  It uses base-address instead of
; absolute-address.  The results we need could probably be
; obtained by creative instantiation of what we have.  But I
; prefer simply to prove them cold in perfect analogy with
; what we have done already.
```

THEOREM: lessp-0-length
$\text{listp}\,(a) \rightarrow (0 < \text{length}\,(a))$

THEOREM: invert-base-address-inverts-base-address-on-subrs

$$
\begin{aligned}
&\text{adpp}\,(\text{cons}\,(\mathit{name},\ 0),\ \mathit{segment})\\
\rightarrow \ & (\text{invert-base-address}\,(\text{base-address}\,(\mathit{name},\\
&\qquad\qquad \text{link-table-for-segment}\,(\mathit{segment},\\
&\qquad\qquad\qquad \mathit{addr0})),\\
&\qquad \text{link-table-for-segment}\,(\mathit{segment},\ \mathit{addr0}))\\
&= \ \mathit{name})
\end{aligned}
$$

THEOREM: lessp-base-address-link-table-for-segment

$$
\begin{aligned}
&(\text{adpp}\,(\text{cons}\,(\mathit{name},\ 0),\ \text{i-prog-segment}\,(i))\\
\land \ & ((\mathit{load\text{-}addr}\\
&\quad + \ \text{segment-length}\,(\text{i-usr-data-segment}\,(i))\\
&\quad + \ \text{segment-length}\,(\text{i-prog-segment}\,(i))\\
&\quad + \ \text{segment-length}\,(\text{i-sys-data-segment}\,(i)))\\
&\ < \ \exp\,(2,\ \text{i-word-size}\,(i))))\\
\rightarrow \ & (\text{base-address}\,(\mathit{name},\\
&\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i),\\
&\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))))\\
&\ < \ \exp\,(2,\ \text{i-word-size}\,(i)))
\end{aligned}
$$

THEOREM: numberp-base-address-link-table-for-segment

$$
\begin{aligned}
&(\mathit{addr0} \in \mathbf{N})\\
\rightarrow \ & (\text{base-address}\,(\mathit{name},\ \text{link-table-for-segment}\,(\mathit{segment},\ \mathit{addr0})) \in \mathbf{N})
\end{aligned}
$$

THEOREM: v-to-subr-inverts-subr-to-v

$$
\begin{array}{l}
(\text{adpp}\,(\text{cons}\,(\mathit{subr},\ \mathtt{0}),\ \text{i-prog-segment}\,(i)) \land \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr}))\\
\to\quad (\text{v-to-subr}\,(\text{subr-to-v}\,(\mathit{subr},\\
\qquad\qquad\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i),\\
\qquad\qquad\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))),\\
\qquad\qquad\qquad \text{i-word-size}\,(i)),\\
\qquad\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i),\\
\qquad\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))))\\
\qquad = \quad \mathit{subr})
\end{array}
$$

THEOREM: equal-subr-to-v

$$
\begin{array}{l}
(\text{adpp}\,(\text{cons}\,(\mathit{subr1},\ \mathtt{0}),\ \text{i-prog-segment}\,(i))\\
\land\quad \text{adpp}\,(\text{cons}\,(\mathit{subr2},\ \mathtt{0}),\ \text{i-prog-segment}\,(i))\\
\land\quad \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr}))\\
\to\quad ((\text{subr-to-v}\,(\mathit{subr1},\\
\qquad\qquad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i),\\
\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))),\\
\qquad\qquad \text{i-word-size}\,(i))\\
\qquad = \quad \text{subr-to-v}\,(\mathit{subr2},\\
\qquad\qquad\quad \text{link-table-for-segment}\,(\text{i-prog-segment}\,(i),\\
\qquad\qquad\qquad\quad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))),\\
\qquad\qquad\quad \text{i-word-size}\,(i)))\\
\qquad = \quad (\mathit{subr1} = \mathit{subr2}))
\end{array}
$$

THEOREM: lessp-find-label-length

$$
\text{find-labelp}\,(\mathit{lab},\ \mathit{lst}) \to (\text{find-label}\,(\mathit{lab},\ \mathit{lst}) < \text{length}\,(\mathit{lst}))
$$

THEOREM: lessp-base-address-link-table-for-labels

$$
\begin{array}{l}
(\text{icode-labelp}\,(\mathit{lab},\ \text{i-prog-segment}\,(i)) \land \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr}))\\
\to\quad (\text{base-address}\,(\mathit{lab},\\
\qquad\qquad \text{link-table-for-labels}\,(\text{cdr}\,(\text{assoc}\,(\text{adp-name}\,(\mathit{lab}),\\
\qquad\qquad\qquad\qquad \text{i-prog-segment}\,(i))),\\
\qquad\qquad\qquad \mathit{load\text{-}addr}\\
\qquad\qquad\qquad + \quad \text{segment-length}\,(\text{i-usr-data-segment}\,(i))\\
\qquad\qquad\qquad + \quad \text{link-table-entry}\,(\text{adp-name}\,(\mathit{lab}),\\
\qquad\qquad\qquad\qquad\qquad \text{i-prog-segment}\,(i))))\\
\qquad < \quad \exp\,(\mathtt{2},\ \text{i-word-size}\,(i)))
\end{array}
$$

THEOREM: numberp-base-address-link-table-for-labels

$$
(\mathit{addr0} \in \mathbf{N}) \to (\text{base-address}\,(\mathit{name},\ \text{link-table-for-labels}\,(\mathit{lst},\ \mathit{addr0})) \in \mathbf{N})
$$

THEOREM: assoc-cdrp-link-table-for-labels-t

$$
\begin{array}{l}
((\mathit{addr0} \in \mathbf{N}) \land \text{find-labelp}\,(\mathit{lab},\ \mathit{lst}))\\
\to\quad \text{assoc-cdrp}\,(\mathit{addr0} + \text{find-label}\,(\mathit{lab},\ \mathit{lst}),\\
\qquad\qquad \text{link-table-for-labels}\,(\mathit{lst},\ \mathit{addr0}))
\end{array}
$$

THEOREM: assoc-cdrp-link-table-for-labels-f

$(n \not< (\mathit{addr0} + \text{length}(\mathit{lst})))$
$\rightarrow (\neg \text{assoc-cdrp}(n, \text{link-table-for-labels}(\mathit{lst}, \mathit{addr0})))$

```
;The following lemma is a generalization of what we need.  We will
; instantiate it with addr0 = 0
```

THEOREM: find-containing-label-table-base-address-generalized

$$
\begin{array}{l}
((\mathit{addr0} \in \mathbf{N}) \\
\wedge\ \text{definedp}(\text{adp-name}(\mathit{lab}), \mathit{segment}) \\
\wedge\ \text{find-labelp}(\mathit{lab}, \text{cdr}(\text{assoc}(\text{adp-name}(\mathit{lab}), \mathit{segment})))) \\
\rightarrow\ (\text{find-containing-label-table}(\text{base-address}(\mathit{lab}, \text{link-table-for-labels}(\text{cdr}(\text{assoc}(\text{adp-name}(\mathit{lab}), \mathit{segment})), \mathit{addr0} + \text{link-table-entry}(\text{adp-name}(\mathit{lab}), \mathit{segment}))), \\
\qquad \text{link-table-for-prog-labels}(\mathit{segment}, \mathit{addr0})) \\
\qquad = \text{link-table-for-labels}(\text{cdr}(\text{assoc}(\text{adp-name}(\mathit{lab}), \mathit{segment})), \mathit{addr0} + \text{link-table-entry}(\text{adp-name}(\mathit{lab}), \mathit{segment})))
\end{array}
$$

THEOREM: find-containing-label-table-base-address

$$
\begin{array}{l}
(\text{definedp}(\text{adp-name}(\mathit{lab}), \mathit{segment}) \\
\wedge\ \text{find-labelp}(\mathit{lab}, \text{cdr}(\text{assoc}(\text{adp-name}(\mathit{lab}), \mathit{segment})))) \\
\rightarrow\ (\text{find-containing-label-table}(\text{base-address}(\mathit{lab}, \text{link-table-for-labels}(\text{cdr}(\text{assoc}(\text{adp-name}(\mathit{lab}), \mathit{segment})), \mathit{load\text{-}addr} + \text{segment-length}(\text{i-usr-data-segment}(i)) + \text{link-table-entry}(\text{adp-name}(\mathit{lab}), \mathit{segment}))), \\
\qquad \text{link-table-for-prog-labels}(\mathit{segment}, \mathit{load\text{-}addr} + \text{segment-length}(\text{i-usr-data-segment}(i)))) \\
\qquad = \text{link-table-for-labels}(\text{cdr}(\text{assoc}(\text{adp-name}(\mathit{lab}), \mathit{segment})), \mathit{load\text{-}addr} + \text{segment-length}(\text{i-usr-data-segment}(i)) + \text{link-table-entry}(\text{adp-name}(\mathit{lab}), \mathit{segment})))
\end{array}
$$

THEOREM: lessp-cdar-link-table-for-labels

$\text{listp}(\text{link-table-for-labels}(\mathit{lst}, \mathit{addr0}))$
$\rightarrow (\text{cdar}(\text{link-table-for-labels}(\mathit{lst}, \mathit{addr0})) \not< \mathit{addr0})$

THEOREM: lessp-find-label-cdar-link-table-for-labels

$$
\begin{array}{l}
\text{find-labelp}\,(\mathit{lab},\ \mathit{lst}) \\
\rightarrow\quad ((\mathit{addr0} + \text{find-label}\,(\mathit{lab},\ \mathit{lst})) \\
\qquad \not< \quad \text{cdar}\,(\text{link-table-for-labels}\,(\mathit{lst},\ \mathit{addr0})))
\end{array}
$$

THEOREM: find-labelp-implies-listp-link-table-for-labels

$$
\text{find-labelp}\,(\mathit{lab},\ \mathit{lst}) \rightarrow \text{listp}\,(\text{link-table-for-labels}\,(\mathit{lst},\ \mathit{addr0}))
$$

THEOREM: invert-base-address-inverts-base-address-on-labels

$$
\begin{array}{l}
((\mathit{addr0} \in \mathbf{N}) \wedge \text{find-labelp}\,(\mathit{lab},\ \mathit{lst})) \\
\rightarrow\quad (\text{invert-base-address}\,(\text{base-address}\,(\mathit{lab}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{link-table-for-labels}\,(\mathit{lst},\ \mathit{addr0})), \\
\qquad\qquad\qquad\qquad \text{link-table-for-labels}\,(\mathit{lst},\ \mathit{addr0})) \\
\qquad = \quad \mathit{lab})
\end{array}
$$

THEOREM: v-to-label-inverts-label-to-v

$$
\begin{array}{l}
(\text{icode-labelp}\,(\mathit{lab},\ \text{i-prog-segment}\,(i)) \wedge \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr})) \\
\rightarrow\quad (\text{v-to-label}\,(\text{label-to-v}\,(\mathit{lab}, \\
\qquad\qquad\qquad \text{link-table-for-prog-labels}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))), \\
\qquad\qquad\qquad \text{i-word-size}\,(i)), \\
\qquad\qquad \text{link-table-for-prog-labels}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i)))) \\
\qquad = \quad \mathit{lab})
\end{array}
$$

THEOREM: equal-label-to-v

$$
\begin{array}{l}
(\text{icode-labelp}\,(\mathit{lab1},\ \text{i-prog-segment}\,(i)) \\
\ \wedge\quad \text{icode-labelp}\,(\mathit{lab2},\ \text{i-prog-segment}\,(i)) \\
\ \wedge\quad \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr})) \\
\rightarrow\quad ((\text{label-to-v}\,(\mathit{lab1}, \\
\qquad\qquad \text{link-table-for-prog-labels}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))), \\
\qquad\qquad \text{i-word-size}\,(i)) \\
\qquad = \quad \text{label-to-v}\,(\mathit{lab2}, \\
\qquad\qquad\qquad \text{link-table-for-prog-labels}\,(\text{i-prog-segment}\,(i), \\
\qquad\qquad\qquad\qquad \mathit{load\text{-}addr} + \text{segment-length}\,(\text{i-usr-data-segment}\,(i))), \\
\qquad\qquad\qquad \text{i-word-size}\,(i))) \\
\qquad = \quad (\mathit{lab1} = \mathit{lab2}))
\end{array}
$$

```
; The following little lemma would not be necessary if type and untag
; were opened to car and cadr and we were willing to suffer elims.
```

THEOREM: equal-types-and-untags

$$
\begin{array}{l}
((\text{type}\,(x) = \text{type}\,(y)) \\
\ \wedge\quad (\text{cddr}\,(x) = \mathbf{nil})
\end{array}
$$

$\wedge \quad (\mathrm{cddr}\,(y) = \mathbf{nil})$
$\wedge \quad (x \neq y))$
$\rightarrow \quad (\mathrm{untag}\,(x) \neq \mathrm{untag}\,(y))$

THEOREM: equal-link-data-word
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge \quad (\text{i-word-size}\,(i) \not\simeq \mathtt{0})$
$\wedge \quad \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr})$
$\wedge \quad \text{i-objectp}\,(x,\ i)$
$\wedge \quad \text{i-objectp-type}\,(\text{type}\,(x),\ y,\ i))$
$\rightarrow \quad ((\text{link-data-word}\,(x,\ \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}),\ \text{i-word-size}\,(i))$
$\qquad = \quad \text{link-data-word}\,(y,\ \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}),\ \text{i-word-size}\,(i)))$
$\qquad = \quad (x = y))$

```
; Note:  small-naturalp is not disabled above simply because I
; didn't provide what might be called equal-nat-to-v but rely
; instead on nat-to-v-equivalence, which has small-naturalp
; expanded in its hyps.
```

THEOREM: bvp-link-data-word
$\text{bvp}\,(\text{link-data-word}\,(x,\ \mathit{table},\ \mathit{word\text{-}size}))$

THEOREM: xor-xxx-is-equal
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge \quad (\text{i-word-size}\,(i) \not\simeq \mathtt{0})$
$\wedge \quad \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr})$
$\wedge \quad \text{i-objectp}\,(x,\ i)$
$\wedge \quad \text{i-objectp-type}\,(\text{type}\,(x),\ y,\ i))$
$\rightarrow \quad (\text{all-zero-bitvp}\,(\text{xor-xxx}\,(x,\ y,\ i,\ \mathit{load\text{-}addr})) = (x = y))$

THEOREM: v-to-nat-v-xor-xx
$\text{v-to-nat}\,(\text{v-xor}\,(x,\ x)) = \mathtt{0}$

THEOREM: v-xor-is-equal
$(\text{bvp}\,(a) \wedge \text{bvp}\,(b) \wedge (\text{length}\,(a) = \text{length}\,(b)))$
$\rightarrow \quad ((\text{v-to-nat}\,(\text{v-xor}\,(a,\ b)) = \mathtt{0}) = (a = b))$

THEOREM: equal-link-word-lemma1
$\text{i-objectp}\,(x,\ i) \rightarrow (\neg\ \text{icode-instructionp}\,(x))$

THEOREM: equal-link-word-lemma2
$\text{i-objectp-type}\,(\mathit{type},\ x,\ i) \rightarrow (\neg\ \text{icode-instructionp}\,(x))$

THEOREM: equal-link-word
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\land \quad (\mathit{word\text{-}size} = \text{i-word-size}\,(i))$
$\land \quad (\text{i-word-size}\,(i) \not\simeq 0)$
$\land \quad \text{i-loadablep}\,(i,\ \mathit{load\text{-}addr})$
$\land \quad \text{i-objectp}\,(x,\ i)$
$\land \quad \text{i-objectp-type}\,(\text{type}\,(x),\ y,\ i))$
$\to \quad ((\text{link-word}\,(x,\ \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}),\ \mathit{word\text{-}size})$
$\qquad = \quad \text{link-word}\,(y,\ \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}),\ \mathit{word\text{-}size}))$
$\quad = \quad (x = y))$

THEOREM: v-xor-append
$(\text{bvp}\,(x) \land \text{bvp}\,(y) \land (\text{length}\,(a) = \text{length}\,(b)))$
$\to \quad (\text{v-xor}\,(\text{append}\,(a,\ x),\ \text{append}\,(b,\ y)) = \text{append}\,(\text{v-xor}\,(a,\ b),\ \text{v-xor}\,(x,\ y)))$

THEOREM: bitv-to-v-xor-bitv
$(\text{length}\,(a) = \text{length}\,(b))$
$\to \quad (\text{bitv-to-v}\,(\text{xor-bitv}\,(a,\ b),\ \text{length}\,(a))$
$\quad = \quad \text{v-xor}\,(\text{bitv-to-v}\,(a,\ \text{length}\,(a)),\ \text{bitv-to-v}\,(b,\ \text{length}\,(a))))$

THEOREM: bitv-to-v-inverts-v-to-bitv
$(\text{bvp}\,(a) \land (\mathit{word\text{-}size} = \text{length}\,(a)))$
$\to \quad (\text{bitv-to-v}\,(\text{v-to-bitv}\,(a),\ \mathit{word\text{-}size}) = a)$

THEOREM: bitv-to-v-xor-xxx-is-v-xor-link-word
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\land \quad (\mathit{word\text{-}size} = \text{i-word-size}\,(i))$
$\land \quad (\text{i-word-size}\,(i) \in \mathbf{N})$
$\land \quad \text{i-objectp}\,(x,\ i)$
$\land \quad \text{i-objectp-type}\,(\text{type}\,(x),\ y,\ i))$
$\to \quad ((\text{bitv-to-v}\,(\text{xor-xxx}\,(x,\ y,\ i,\ \mathit{load\text{-}addr}),\ \mathit{word\text{-}size})$
$\qquad = \quad \text{v-xor}\,(\text{link-word}\,(x,\ \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}),\ \mathit{word\text{-}size}),$
$\qquad\qquad\qquad \text{link-word}\,(y,\ \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}),\ \mathit{word\text{-}size})))$
$\quad = \quad \mathbf{t})$

```
; The complications are neverending!  Ok.  The next problem is
; that the two machines use xor in opposite orders.  That is,
; the xor done by one is the commuted version of the one done
; by the other.  So
```

THEOREM: commutativity-of-xor-bitv
$(\text{length}\,(x) = \text{length}\,(y)) \to (\text{xor-bitv}\,(x,\ y) = \text{xor-bitv}\,(y,\ x))$

THEOREM: commutativity-of-xor-xxx
$\text{xor-xxx}\,(x,\ y,\ i,\ \mathit{load\text{-}addr}) = \text{xor-xxx}\,(y,\ x,\ i,\ \mathit{load\text{-}addr})$

THEOREM: alu-thm_bitv-zb-xor
(bvp (a) $\wedge$ bvp (b) $\wedge$ (length (a) = length (b)) $\wedge$ boolp (c))
$\rightarrow$ (zb (v-alu $(c,\ a,\ b,$ list (**t**, **t**, **f**, **t**))) = ($\neg$ v-nzerop (v-xor $(a,\ b)$)))

```
; There is a certain amount of xor versus equal work above.  Unfortunately,
; I think it was all made irrelevant when FM9001 was introduced with its
; kind of bit vectors.
```

THEOREM: v-nzerop-v-xor
(bvp (a) $\wedge$ bvp (b) $\wedge$ (length (a) = length (b)))
$\rightarrow$ (v-nzerop (v-xor $(a,\ b)$) = $(a \neq b)$)

THEOREM: i-m-one-way-correspondence-step-xor_<z>_<tsp>_x
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp $(i,$ *load-addr*$)$
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-xor_<z>_<tsp>_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(xor_<z>_<tsp>_x)`))
$\rightarrow$ (i->m (i-xor_<z>_<tsp>_x-step $(i,$ *load-addr*$)$, *boot-lst*, *load-addr*)
= m-step (i->m $(i,$ *boot-lst*, *load-addr*$)$))

```
; The next owc uses xor to zero the top of stack.  I need
```

THEOREM: v-xor-x-x-is-0
v-xor $(x,\ x)$ = nat-to-v (0, length (x))

```
; This rule will clash with others, like (v-to-nat (v-xor x x)) = 0,
; but I don't think I'll need the other rules anymore.  Famous last
; words.  This will return to haunt me.
```

THEOREM: i-m-one-way-correspondence-step-xor_<tsp>_<tsp>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp $(i,$ *load-addr*$)$
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-xor_<tsp>_<tsp>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(xor_<tsp>_<tsp>)`))
$\rightarrow$ (i->m (i-xor_<tsp>_<tsp>-step (i), *boot-lst*, *load-addr*)
= m-step (i->m $(i,$ *boot-lst*, *load-addr*$)$))

```
; To make the next theorem work I had to change the -okp fn to
; insure that the nextword was an i-objectp
```

THEOREM: i-m-one-way-correspondence-step-move-z_<tsp>_*
(($load\text{-}addr \in \mathbf{N}$)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = 'run)
∧ (i-word-size (i) = 32)
∧ i-move-z_<tsp>_*-okp (i)
∧ (i-current-instruction (i) = '(move-z_<tsp>_*)))
→ (i->m (i-move-z_<tsp>_*-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move-n_x_*
(($load\text{-}addr \in \mathbf{N}$)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = 'run)
∧ (i-word-size (i) = 32)
∧ i-move-n_x_*-okp (i)
∧ (i-current-instruction (i) = '(move-n_x_*)))
→ (i->m (i-move-n_x_*-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move_<tsp>_*
(($load\text{-}addr \in \mathbf{N}$)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = 'run)
∧ (i-word-size (i) = 32)
∧ i-move_<tsp>_*-okp (i)
∧ (i-current-instruction (i) = '(move_<tsp>_*)))
→ (i->m (i-move_<tsp>_*-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: listp-nat-to-v
listp (nat-to-v (n, *word-size*)) = (*word-size* $\not\simeq$ 0)

THEOREM: listp-int-to-v
listp (int-to-v (i, *word-size*)) = (*word-size* $\not\simeq$ 0)

THEOREM: i-m-one-way-correspondence-step-incr_<tsp>_<tsp>{i}
(($load\text{-}addr \in \mathbf{N}$)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = 'run)
∧ (i-word-size (i) = 32)
∧ i-incr_<tsp>_<tsp>{i}-okp (i)
∧ (i-current-instruction (i) = '(incr_<tsp>_<tsp>{i})))
→ (i->m (i-incr_<tsp>_<tsp>{i}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-sub_<tsp>{i}_x{i}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = 32)
$\land$ i-sub_<tsp>{i}_x{i}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(sub_<tsp>{i}_x{i})`))
$\rightarrow$ (i->m (i-sub_<tsp>{i}_x{i}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-decr_<tsp>_<tsp>{i}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = 32)
$\land$ i-decr_<tsp>_<tsp>{i}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(decr_<tsp>_<tsp>{i})`))
$\rightarrow$ (i->m (i-decr_<tsp>_<tsp>{i}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-neg_<tsp>_<tsp>{i}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = 32)
$\land$ i-neg_<tsp>_<tsp>{i}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(neg_<tsp>_<tsp>{i})`))
$\rightarrow$ (i->m (i-neg_<tsp>_<tsp>{i}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-asr_<c>_<tsp>_<tsp>{b}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = 32)
$\land$ i-asr_<c>_<tsp>_<tsp>{b}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(asr_<c>_<tsp>_<tsp>{b})`))
$\rightarrow$ (i->m (i-asr_<c>_<tsp>_<tsp>{b}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: plus-1
$(1 + x) = (1 + x)$

THEOREM: not-zerop-exp-2-32
0 < exp (2, 32)

THEOREM: i-m-one-way-correspondence-step-addc_<c>_x{n}_y{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-addc_<c>_x{n}_y{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(addc_<c>_x{n}_y{n})`))
→ (i->m (i-addc_<c>_x{n}_y{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-add_<tsp>{n}_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-add_<tsp>{n}_x{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(add_<tsp>{n}_x{n})`))
→ (i->m (i-add_<tsp>{n}_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-incr_<tsp>_<tsp>{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-incr_<tsp>_<tsp>{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(incr_<tsp>_<tsp>{n})`))
→ (i->m (i-incr_<tsp>_<tsp>{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-incr_y_y{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-incr_y_y{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(incr_y_y{n})`))
→ (i->m (i-incr_y_y{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

```
; The following lemma, put-link-segment-no-op, states that it is a
; no-op to put into the link-segment of a segment a value that is
; the link-word of what is already at the corresponding position
; in the segment.  This lemma is what enables us to convert from
; one data type to another, e.g., from INT to NAT, by doing an
```

```
; explicit push in the i-level machine but a no-op in the m-level.
; To justify such an act all we need to know is that the link-word
; of what is being pushed is the link-word of what was already
; there.
```

THEOREM: put-link-area-no-op
$((\mathit{offset} \in \mathbf{N}) \land (\mathit{offset} < \text{length}\,(v)))$
$\rightarrow$ (put (link-word (unlabel (get (*offset*, *v*)), *link-tables*, *word-size*),
offset,
link-area (*v*, *link-tables*, *word-size*))
= link-area (*v*, *link-tables*, *word-size*))

THEOREM: put-link-segment-no-op
(definedp (*name*, *segment*)
$\land$ (*offset* $\in \mathbf{N}$)
$\land$ (*offset* < length (cdr (assoc (*name*, *segment*))))
$\land$ (*val* = link-word (unlabel (get (*offset*, cdr (assoc (*name*, *segment*)))),
link-tables,
word-size)))
$\rightarrow$ ((put (*val*,
link-table-entry (*name*, *segment*) + *offset*,
link-segment (*segment*, *link-tables*, *word-size*))
= link-segment (*segment*, *link-tables*, *word-size*))
= **t**)

```
; For the INT-TO-NAT case the justification is then:
```

THEOREM: int-to-v-is-nat-to-v-on-nats
($n \in \mathbf{N}$) $\rightarrow$ (int-to-v (n, *word-size*) = nat-to-v (n, *word-size*))

THEOREM: i-m-one-way-correspondence-step-int-to-nat
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-int-to-nat-okp (i)
$\land$ (i-current-instruction (i) = `'(int-to-nat)`))
$\rightarrow$ (i->m (i-int-to-nat-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

EVENT: Disable int-to-v-is-nat-to-v-on-nats.

THEOREM: i-m-one-way-correspondence-step-decr_<tsp>_<tsp>{n}

((*load-addr* $\in$ **N**)
 $\wedge$ i-state-okp (i, *load-addr*)
 $\wedge$ (i-psw (i) = `'run`)
 $\wedge$ (i-word-size (i) = 32)
 $\wedge$ i-decr_<tsp>_<tsp>{n}-okp (i)
 $\wedge$ (i-current-instruction (i) = `'(decr_<tsp>_<tsp>{n})`))
$\rightarrow$ (i->m (i-decr_<tsp>_<tsp>{n}-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

EVENT: Enable times-2.

THEOREM: i-m-one-way-correspondence-step-add_<c>_x_x{n}
((*load-addr* $\in$ **N**)
 $\wedge$ i-state-okp (i, *load-addr*)
 $\wedge$ (i-psw (i) = `'run`)
 $\wedge$ (i-word-size (i) = 32)
 $\wedge$ i-add_<c>_x_x{n}-okp (i)
 $\wedge$ (i-current-instruction (i) = `'(add_<c>_x_x{n})`))
$\rightarrow$ (i->m (i-add_<c>_x_x{n}-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-add_<tsp>_<tsp>{n}
((*load-addr* $\in$ **N**)
 $\wedge$ i-state-okp (i, *load-addr*)
 $\wedge$ (i-psw (i) = `'run`)
 $\wedge$ (i-word-size (i) = 32)
 $\wedge$ i-add_<tsp>_<tsp>{n}-okp (i)
 $\wedge$ (i-current-instruction (i) = `'(add_<tsp>_<tsp>{n})`))
$\rightarrow$ (i->m (i-add_<tsp>_<tsp>{n}-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: remainder-by-2
$((x \bmod 2) \neq 0) \rightarrow ((x \bmod 2) = 1)$

THEOREM: i-m-one-way-correspondence-step-lsr_<c>_x_x{n}
((*load-addr* $\in$ **N**)
 $\wedge$ i-state-okp (i, *load-addr*)
 $\wedge$ (i-psw (i) = `'run`)
 $\wedge$ (i-word-size (i) = 32)
 $\wedge$ i-lsr_<c>_x_x{n}-okp (i)
 $\wedge$ (i-current-instruction (i) = `'(lsr_<c>_x_x{n})`))
$\rightarrow$ (i->m (i-lsr_<c>_x_x{n}-step (i), *boot-lst*, *load-addr*)
 = m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-subb_<c>_x{n}_y{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-subb_<c>_x{n}_y{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(subb_<c>_x{n}_y{n})`))
→ (i->m (i-subb_<c>_x{n}_y{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

```
; I redefined the -step fn below because it had the wrong parity on the
; c-flg!
```

THEOREM: i-m-one-way-correspondence-step-sub_<c>_<tsp>{n}_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<c>_<tsp>{n}_x{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(sub_<c>_<tsp>{n}_x{n})`))
→ (i->m (i-sub_<c>_<tsp>{n}_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-sub_<tsp>{n}_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<tsp>{n}_x{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(sub_<tsp>{n}_x{n})`))
→ (i->m (i-sub_<tsp>{n}_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

```
; The use hint below is necessary here because we have the hypothesis
; that both the top of tsp and x are addresses into the same usr-data
; area.  The rewriter replaces references to the adp-name of the top
; of tsp with references to the adp-name of x because it is simpler.
; But that prevents the firing of our rewrite rules for establishing
; that addresses link to small-naturalps.
```

THEOREM: i-m-one-way-correspondence-step-sub_<c>_<tsp>{a}_x{a}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)

∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<c>_<tsp>{a}_x{a}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<c>_<tsp>{a}_x{a})))
→ (i->m (i-sub_<c>_<tsp>{a}_x{a}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-sub_<tsp>{s}_x{s}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<tsp>{s}_x{s}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<tsp>{s}_x{s})))
→ (i->m (i-sub_<tsp>{s}_x{s}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: v-nzerop-nat-to-v-bridge
((*firstn* = nat-to-v (*n*, *word-size*)) ∧ (*n* < exp (2, *word-size*)))
→ (v-nzerop (*firstn*) = (*n* $\not\simeq$ 0))

THEOREM: equal-difference-zero
(($x - y$) = 0) = ($y \not< x$)

THEOREM: equal-v-to-nat-v-to-nat
(bvp (*a*) ∧ bvp (*b*) ∧ (length (*a*) = length (*b*)))
→ ((v-to-nat (*a*) = v-to-nat (*b*)) = (*a* = *b*))

EVENT: Enable my-lessp-v-to-nat-exp.

THEOREM: alu-thm_bitv-zb-sub
(bvp (*a*) ∧ bvp (*b*) ∧ (length (*a*) = length (*b*)) ∧ boolp (*c*))
→ (zb (v-alu (*c*, *a*, *b*, list (**t**, **t**, **t**, **f**))) = (*a* = *b*))

THEOREM: i-m-one-way-correspondence-step-sub_<z>_x{s}_y{s}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<z>_x{s}_y{s}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<z>_x{s}_y{s})))
→ (i->m (i-sub_<z>_x{s}_y{s}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

```
; To make the next thm work I changed the -okp fn to insure
; that nextword is an i-objectp.  I also changed the link instruction
; alist.
```

THEOREM: i-m-one-way-correspondence-step-move-c_<tsp>_*
$((\textit{load-addr} \in \mathbf{N})$
$\land$ i-state-okp $(i,\ \textit{load-addr})$
$\land$ (i-psw $(i) =$ `'run`)
$\land$ (i-word-size $(i) = 32$)
$\land$ i-move-c_<tsp>_*-okp (i)
$\land$ (i-current-instruction $(i) =$ `'(move-c_<tsp>_*)`))
$\to$ (i->m (i-move-c_<tsp>_*-step (i), *boot-lst*, *load-addr*)
$=$ m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-move-v_<tsp>_*
$((\textit{load-addr} \in \mathbf{N})$
$\land$ i-state-okp $(i,\ \textit{load-addr})$
$\land$ (i-psw $(i) =$ `'run`)
$\land$ (i-word-size $(i) = 32$)
$\land$ i-move-v_<tsp>_*-okp (i)
$\land$ (i-current-instruction $(i) =$ `'(move-v_<tsp>_*)`))
$\to$ (i->m (i-move-v_<tsp>_*-step (i), *boot-lst*, *load-addr*)
$=$ m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: v-or-append
((length $(a) =$ length $(b)) \land$ bvp $(x) \land$ bvp (y))
$\to$ (v-or (append $(a,\ x)$, append $(b,\ y)) =$ append (v-or $(a,\ b)$, v-or $(x,\ y)$))

THEOREM: v-or-bitv-to-v
(bit-vectorp $(x$, *word-size*$) \land$ bit-vectorp $(y$, *word-size*))
$\to$ (v-or (bitv-to-v $(x$, *word-size*), bitv-to-v $(y$, *word-size*))
$=$ bitv-to-v (or-bitv $(x,\ y)$, *word-size*))

THEOREM: commutativity-of-or-bitv
(length $(x) =$ length $(y)) \to$ (or-bitv $(x,\ y) =$ or-bitv $(y,\ x)$)

THEOREM: i-m-one-way-correspondence-step-or_<tsp>{v}_x{v}
$((\textit{load-addr} \in \mathbf{N})$
$\land$ i-state-okp $(i,\ \textit{load-addr})$
$\land$ (i-psw $(i) =$ `'run`)
$\land$ (i-word-size $(i) = 32$)
$\land$ i-or_<tsp>{v}_x{v}-okp (i)
$\land$ (i-current-instruction $(i) =$ `'(or_<tsp>{v}_x{v})`))
$\to$ (i->m (i-or_<tsp>{v}_x{v}-step (i), *boot-lst*, *load-addr*)
$=$ m-step (i->m $(i$, *boot-lst*, *load-addr*)))

THEOREM: v-and-append
((length (a) = length (b)) ∧ bvp (x) ∧ bvp (y))
→ (v-and (append (a, x), append (b, y)) = append (v-and (a, b), v-and (x, y)))

THEOREM: v-and-bitv-to-v
(bit-vectorp (x, *word-size*) ∧ bit-vectorp (y, *word-size*))
→ (v-and (bitv-to-v (x, *word-size*), bitv-to-v (y, *word-size*))
= bitv-to-v (and-bitv (x, y), *word-size*))

THEOREM: commutativity-of-and-bitv
(length (x) = length (y)) → (and-bitv (x, y) = and-bitv (y, x))

THEOREM: i-m-one-way-correspondence-step-and_<tsp>{v}_x{v}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-and_<tsp>{v}_x{v}-okp (i)
∧ (i-current-instruction (i) = `'(and_<tsp>{v}_x{v})`))
→ (i->m (i-and_<tsp>{v}_x{v}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: v-not-append
bvp (x) → (v-not (append (a, x)) = append (v-not (a), v-not (x)))

THEOREM: v-not-bitv-to-v
bit-vectorp (x, *word-size*)
→ (v-not (bitv-to-v (x, *word-size*)) = bitv-to-v (not-bitv (x), *word-size*))

THEOREM: i-m-one-way-correspondence-step-not_<tsp>_<tsp>{v}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-not_<tsp>_<tsp>{v}-okp (i)
∧ (i-current-instruction (i) = `'(not_<tsp>_<tsp>{v})`))
→ (i->m (i-not_<tsp>_<tsp>{v}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: v-xor-bitv-to-v
(bit-vectorp (x, *word-size*) ∧ bit-vectorp (y, *word-size*))
→ (v-xor (bitv-to-v (x, *word-size*), bitv-to-v (y, *word-size*))
= bitv-to-v (xor-bitv (x, y), *word-size*))

THEOREM: i-m-one-way-correspondence-step-xor_<tsp>{v}_x{v}

((*load-addr* ∈ **N**)
 ∧ i-state-okp (*i*, *load-addr*)
 ∧ (i-psw (*i*) = `'run`)
 ∧ (i-word-size (*i*) = `32`)
 ∧ i-xor_<tsp>{v}_x{v}-okp (*i*)
 ∧ (i-current-instruction (*i*) = `'(xor_<tsp>{v}_x{v})`))
→ (i->m (i-xor_<tsp>{v}_x{v}-step (*i*), *boot-lst*, *load-addr*)
  = m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: cdr-bitv-to-v
(listp (*x*) ∧ bit-vectorp (*x*, *word-size*))
→ (cdr (bitv-to-v (*x*, *word-size*))
  = bitv-to-v (all-but-last (*x*), *word-size* − 1))

THEOREM: listp-append
listp (append (*a*, *b*)) = (listp (*a*) ∨ listp (*b*))

THEOREM: cdr-append
cdr (append (*a*, *b*))
= **if** listp (*a*) **then** append (cdr (*a*), *b*)
  else cdr (*b*) **endif**

THEOREM: v-buf-append
v-buf (append (*a*, *b*)) = append (v-buf (*a*), v-buf (*b*))

THEOREM: v-lsr-bitv-to-v
bit-vectorp (*x*, *word-size*)
→ (v-lsr (bitv-to-v (*x*, *word-size*)) = bitv-to-v (rsh-bitv (*x*), *word-size*))

```
; The following two disables are motivated merely by the fact that the
; named lemmas were proved for the fm9001 effort and hence might get
; in the way.
```

EVENT: Disable v-buf-append.

EVENT: Disable cdr-append.

THEOREM: i-m-one-way-correspondence-step-lsr_<tsp>_<tsp>{v}
((*load-addr* ∈ **N**)
 ∧ i-state-okp (*i*, *load-addr*)
 ∧ (i-psw (*i*) = `'run`)
 ∧ (i-word-size (*i*) = `32`)
 ∧ i-lsr_<tsp>_<tsp>{v}-okp (*i*)

∧ (i-current-instruction (i) = '(lsr_<tsp>_<tsp>{v}))
→ (i->m (i-lsr_<tsp>_<tsp>{v}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

Theorem: lsh-lemma-1
(bit-vectorp (x, *word-size*) ∧ (*word-size* $\not\simeq$ 0))
→ (bitv-to-v (append (cdr (x), '(0)), *word-size*)
= cons (**f**, bitv-to-v (cdr (x), *word-size* − 1)))

```
; Four of the next six lemmas use (plus x x y y) where (plus i i) could be
; used and then instantiated with (plus x y) and rearranged.  But
; they are, as stated, exactly what is needed and they go through
; without help.
```

Theorem: remainder-add1-xxyy-2
$((1 + (x + x + y + y))$ **mod** $2) = 1$

Theorem: quotient-add1-xxyy-2
$((1 + (x + x + y + y)) \div 2) = (x + y)$

Theorem: remainder-xx-2
$((x + x)$ **mod** $2) = 0$

Theorem: quotient-xx-2
$((x + x) \div 2) = \mathrm{fix}\,(x)$

Theorem: remainder-xxyy-2
$((x + x + y + y)$ **mod** $2) = 0$

Theorem: quotient-xxyy-2
$((x + x + y + y) \div 2) = (x + y)$

Theorem: lsh-lemma-2
nat-to-v (z + exp (2, *word-size*), *word-size*) = nat-to-v (z, *word-size*)

Theorem: lsh-lemma-3
nat-to-v (z + exp (2, *word-size*) + u, *word-size*)
= nat-to-v (z + u, *word-size*)

Theorem: lsh-lemma-4
nat-to-v (z + (exp (2, *word-size*) ∗ v), *word-size*)
= nat-to-v (z, *word-size*)

Event: Disable times-commutes.

THEOREM: lsh-lemma-5
($x \in \mathbf{N}$)
$\rightarrow$ (nat-to-v (x **mod** exp (2, *word-size*), *word-size*)
= nat-to-v (x, *word-size*))

THEOREM: lsh-lemma-6
(*word-size* $\not\simeq$ 0)
$\rightarrow$ (nat-to-v ($x + x$, *word-size*) = cons (**f**, nat-to-v (x, *word-size* $- 1$)))

THEOREM: remainder-add1-xx-2
((1 + ($x + x$)) **mod** 2) = 1

THEOREM: quotient-add1-xx-2
((1 + ($x + x$)) $\div$ 2) = fix (x)

THEOREM: lsh-lemma-7
(bvp (x) $\wedge$ (k = length (x)))
$\rightarrow$ (nat-to-v (v-to-nat (append (x, y)), k) = x)

EVENT: Enable bvp-append.

THEOREM: i-m-one-way-correspondence-step-add_<tsp>_<tsp>{v}
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-add_<tsp>_<tsp>{v}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_<tsp>_<tsp>{v})`))
$\rightarrow$ (i->m (i-add_<tsp>_<tsp>{v}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-or_<tsp>{b}_x{b}
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-or_<tsp>{b}_x{b}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(or_<tsp>{b}_x{b})`))
$\rightarrow$ (i->m (i-or_<tsp>{b}_x{b}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-and_<tsp>{b}_x{b}
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)

∧ (i-word-size (*i*) = 32)
∧ i-and_<tsp>{b}_x{b}-okp (*i*)
∧ (i-current-instruction (*i*) = '(and_<tsp>{b}_x{b})))
→ (i->m (i-and_<tsp>{b}_x{b}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-xor_<tsp>{b}_*{b}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-xor_<tsp>{b}_*{b}-okp (*i*)
∧ (i-current-instruction (*i*) = '(xor_<tsp>{b}_*{b})))
→ (i->m (i-xor_<tsp>{b}_*{b}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step-xor_<tsp>{b}_x{b}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-xor_<tsp>{b}_x{b}-okp (*i*)
∧ (i-current-instruction (*i*) = '(xor_<tsp>{b}_x{b})))
→ (i->m (i-xor_<tsp>{b}_x{b}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

```
; The following lemma is just enough arithmetic to get the add_pc_x{n}
; lemma.  As noted before, I've adopted the strategy of not automatically
; rewriting arithmetic expressions.  I could perhaps get the proof with
; associativity and commutativity available, but I don't want to even
; bother trying.
```

THEOREM: commutativity-of-plus-hack
(*load-addr*
\+ segment-length (i-usr-data-segment (*i*))
\+ link-table-entry (adp-name (untag (i-pc (*i*))), i-prog-segment (*i*))
\+ adp-offset (untag (i-pc (*i*)))
\+ untag (i-x (*i*)))
= (untag (i-x (*i*))
\+ *load-addr*
\+ segment-length (i-usr-data-segment (*i*))
\+ link-table-entry (adp-name (untag (i-pc (*i*))), i-prog-segment (*i*))
\+ adp-offset (untag (i-pc (*i*))))

Event: Disable equal-length-0.

Event: Disable remainder-opener.

Theorem: i-m-one-way-correspondence-step-add_pc_x{n}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-add_pc_x{n}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(add_pc_x{n})`))
$\rightarrow$ (i->m (i-add_pc_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: i-m-one-way-correspondence-step-add_x_x{n}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-add_x_x{n}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(add_x_x{n})`))
$\rightarrow$ (i->m (i-add_x_x{n}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Event: Enable associativity-of-iplus.

Event: Enable commutativity2-of-iplus.

Theorem: i-m-one-way-correspondence-step-addc_<v>_x{i}_y{i}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (*i*, *load-addr*)
$\land$ (i-psw (*i*) = `'run`)
$\land$ (i-word-size (*i*) = `32`)
$\land$ i-addc_<v>_x{i}_y{i}-okp (*i*)
$\land$ (i-current-instruction (*i*) = `'(addc_<v>_x{i}_y{i})`))
$\rightarrow$ (i->m (i-addc_<v>_x{i}_y{i}-step (*i*), *boot-lst*, *load-addr*)
= m-step (i->m (*i*, *boot-lst*, *load-addr*)))

Theorem: tc-minus-is-inegate
tc-minus (x) = inegate (x)

Event: Disable tc-minus.

THEOREM: integerp-inegate
integerp (inegate (x))

THEOREM: ineg-is-inegate
ineg (x) = inegate (x)

EVENT: Disable ineg.

THEOREM: i-m-one-way-correspondence-step-subb_<v>_x{i}_y{i}
(($load\text{-}addr \in \mathbf{N}$)
 ∧ i-state-okp (i, $load\text{-}addr$)
 ∧ (i-psw (i) = 'run)
 ∧ (i-word-size (i) = 32)
 ∧ i-subb_<v>_x{i}_y{i}-okp (i)
 ∧ (i-current-instruction (i) = '(subb_<v>_x{i}_y{i})))
→ (i->m (i-subb_<v>_x{i}_y{i}-step (i), $boot\text{-}lst$, $load\text{-}addr$)
 = m-step (i->m (i, $boot\text{-}lst$, $load\text{-}addr$)))

```
; To relieve one of the hypotheses above we need the following lemma.
; Intuitively it says that (inegate x) is a small-integerp if x is.
; That's not true, because x might be the most negative integer, which
; has no small opposite.  But in the case in question we know that
; (inegate x) + y is negative, so things work out to tell us that  x isn't
; the most negative.
```

THEOREM: small-integerp-inegate
(($word\text{-}size \not\simeq 0$)
 ∧ small-integerp (x, $word\text{-}size$)
 ∧ negativep (iplus (inegate (x), y))
 ∧ small-integerp (y, $word\text{-}size$))
→ small-integerp (inegate (x), $word\text{-}size$)

THEOREM: small-integerp-fix-small-integer-case-1
(small-integerp (x, $word\text{-}size$)
 ∧ small-integerp (y, $word\text{-}size$)
 ∧ ($word\text{-}size \not\simeq 0$)
 ∧ (¬ small-integerp (iplus (inegate (x), y), $word\text{-}size$))
 ∧ negativep (iplus (inegate (x), y)))
→ small-integerp (iplus (exp (2, $word\text{-}size$), iplus (inegate (x), y)), $word\text{-}size$)

THEOREM: small-integerp-fix-small-integer-case-2
(small-integerp (x, $word\text{-}size$)
 ∧ small-integerp (y, $word\text{-}size$)
 ∧ ($word\text{-}size \not\simeq 0$)

$\wedge$ ($\neg$ small-integerp (iplus (inegate (x), y), *word-size*))
$\wedge$ ($\neg$ negativep (iplus (inegate (x), y))))
$\rightarrow$ small-integerp (iplus ($-$ exp (2, *word-size*), iplus (inegate (x), y)),
word-size)

THEOREM: i-m-one-way-correspondence-step-sub_<nv>_<tsp>{i}_x{i}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-sub_<nv>_<tsp>{i}_x{i}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(sub_<nv>_<tsp>{i}_x{i})`))
$\rightarrow$ (i->m (i-sub_<nv>_<tsp>{i}_x{i}-step (i), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-current-instruction-packer-lemma
((car (*ins*) = pack (*opcode*)) $\wedge$ (cdr (*ins*) = **nil**))
$\rightarrow$ (*ins* = list (pack (*opcode*)))

THEOREM: i-current-instruction-packer
((car (i-current-instruction (i)) = pack (*opcode*))
$\wedge$ (cdr (i-current-instruction (i)) = **nil**))
$\rightarrow$ (i-current-instruction (i) = list (pack (*opcode*)))

THEOREM: i-m-one-way-correspondence-ins-okp-step
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-ins-okp (i-current-instruction (i), i))
$\rightarrow$ (i->m (i-ins-step (i-current-instruction (i), i, *load-addr*),
boot-lst,
load-addr)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-m-one-way-correspondence-step
((*load-addr* $\in$ **N**)
$\wedge$ (i-psw (i-step (i, *load-addr*)) = `'run`)
$\wedge$ (i-word-size (i) = 32))
$\rightarrow$ (i->m (i-step (i, *load-addr*), *boot-lst*, *load-addr*)
= m-step (i->m (i, *boot-lst*, *load-addr*)))

THEOREM: i-word-size-i-halt
i-word-size (i-halt (i, *psw*)) = i-word-size (i)

```
; The following two lemmas and then i-word-size-i-ins-step follow exactly
; the same strategy described above for i-psw-i-ins-step.
```

THEOREM: equal-i-word-size-if-1
(i-word-size (**if** *t1* **then** *b1*
else *b2* **endif**)
= *w*)
= **if** *t1* **then** i-word-size (*b1*) = *w*
else i-word-size (*b2*) = *w* **endif**

THEOREM: equal-i-word-size-if-2
(i-word-size (**if** *t1* **then** *b1*
elseif *t2* **then** *b2*
else *b3* **endif**)
= *w*)
= **if** *t1* **then** i-word-size (*b1*) = *w*
elseif *t2* **then** i-word-size (*b2*) = *w*
else i-word-size (*b3*) = *w* **endif**

THEOREM: i-word-size-i-ins-step
i-word-size (i-ins-step (*ins*, *i*, *load-addr*)) = i-word-size (*i*)

```
; The hint above is mechanically generated as in i-psw case mentioned
; above, except that it does not include i-word-size.
```

THEOREM: i-word-size-i-step
i-word-size (i-step (*i*, *load-addr*)) = i-word-size (*i*)

THEOREM: i-psw-i-run
(i-psw (i (*i*, *n*, *load-addr*)) = `'run`) → (i-psw (*i*) = `'run`)

THEOREM: i-m-one-way-correspondence
((*load-addr* $\in$ **N**)
∧ (i-psw (i (*i*, *n*, *load-addr*)) = `'run`)
∧ (i-word-size (*i*) = 32))
→ (i->m (i (*i*, *n*, *load-addr*), *boot-lst*, *load-addr*)
= m (i->m (*i*, *boot-lst*, *load-addr*), *n*))

```
;  We now prove that i-state-okp is preserved by i-step.  We do
;  it in a very predictable way:  we prove it for each stepper.
;  First a few lemmas...
```

THEOREM: assoc-put-assoc
definedp (*name2*, *alist*)
$\rightarrow$ (assoc (*name1*, put-assoc (*val*, *name2*, *alist*))
= **if** *name1* = *name2* **then** cons (*name1*, *val*)
else assoc (*name1*, *alist*) **endif**)

THEOREM: cddr-tag
cddr (tag (*type*, *obj*)) = **nil**

```
;  Now comes the litany of instructions.

;  After the first two, the hints for all of these preservation lemmas
;  are derived mechanically from the hint for the corresponding
;  one-way-correspondence lemma above by simply adding to the old hint
;  the disabling of the restructuring lemma and the enabling of
;  i-state-okp.
```

THEOREM: i-state-okp-i-move_x_x-step
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ i-move_x_x-okp (*i*))
$\rightarrow$ i-state-okp (i-move_x_x-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-cpush_*-step
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ i-cpush_*-okp (*i*))
$\rightarrow$ i-state-okp (i-cpush_*-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-add_<tsp>{i}_x{i}-step
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ i-add_<tsp>{i}_x{i}-okp (*i*))
$\rightarrow$ i-state-okp (i-add_<tsp>{i}_x{i}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-move_cfp_csp-step
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (*i*, *load-addr*)
$\wedge$ (i-psw (*i*) = `'run`)
$\wedge$ i-move_cfp_csp-okp (*i*))
$\rightarrow$ i-state-okp (i-move_cfp_csp-step (*i*), *load-addr*)

Theorem: i-state-okp-i-cpush_<tsp>+-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-cpush_<tsp>+-okp (i))
$\rightarrow$ i-state-okp (i-cpush_<tsp>+-step (i), *load-addr*)

Theorem: i-state-okp-i-move_csp_cfp-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-move_csp_cfp-okp (i))
$\rightarrow$ i-state-okp (i-move_csp_cfp-step (i), *load-addr*)

Theorem: i-state-okp-i-move_x_tsp-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-move_x_tsp-okp (i))
$\rightarrow$ i-state-okp (i-move_x_tsp-step (i), *load-addr*)

Theorem: i-state-okp-i-cpop_cfp-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-cpop_cfp-okp (i))
$\rightarrow$ i-state-okp (i-cpop_cfp-step (i), *load-addr*)

Theorem: i-state-okp-i-move_x_*-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-move_x_*-okp (i))
$\rightarrow$ i-state-okp (i-move_x_*-step (i), *load-addr*)

Theorem: i-state-okp-i-move_y_*-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-move_y_*-okp (i))
$\rightarrow$ i-state-okp (i-move_y_*-step (i), *load-addr*)

Theorem: i-state-okp-i-add_x{n}_csp-step
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)

$\land$ (i-psw (i) = 'run)
$\land$ i-add_x{n}_csp-okp (i))
$\rightarrow$ i-state-okp (i-add_x{n}_csp-step (i), *load-addr*)

THEOREM: i-state-okp-i-move_x_<x{s}>-step
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = 'run)
$\land$ i-move_x_<x{s}>-okp (i))
$\rightarrow$ i-state-okp (i-move_x_<x{s}>-step (i), *load-addr*)

THEOREM: i-state-okp-i-move_y_<y{s}>-step
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = 'run)
$\land$ i-move_y_<y{s}>-okp (i))
$\rightarrow$ i-state-okp (i-move_y_<y{s}>-step (i), *load-addr*)

THEOREM: i-state-okp-i-move_y_tsp-step
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = 'run)
$\land$ i-move_y_tsp-okp (i))
$\rightarrow$ i-state-okp (i-move_y_tsp-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpush_csp-step
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = 'run)
$\land$ i-tpush_csp-okp (i))
$\rightarrow$ i-state-okp (i-tpush_csp-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpush_tsp-step
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = 'run)
$\land$ i-tpush_tsp-okp (i))
$\rightarrow$ i-state-okp (i-tpush_tsp-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpush_x-step
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = 'run)
$\land$ i-tpush_x-okp (i))
$\rightarrow$ i-state-okp (i-tpush_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpush_<x{s}>-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-tpush_<x{s}>-okp (i))
$\rightarrow$ i-state-okp (i-tpush_<x{s}>-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpush_<x{a}>-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-tpush_<x{a}>-okp (i))
$\rightarrow$ i-state-okp (i-tpush_<x{a}>-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpush_*-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-tpush_*-okp (i))
$\rightarrow$ i-state-okp (i-tpush_*-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop_x-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-tpop_x-okp (i))
$\rightarrow$ i-state-okp (i-tpop_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop_y-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-tpop_y-okp (i))
$\rightarrow$ i-state-okp (i-tpop_y-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop_<x{s}>-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-tpop_<x{s}>-okp (i))
$\rightarrow$ i-state-okp (i-tpop_<x{s}>-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop_<x{a}>-step
$((load\text{-}addr \in \mathbf{N})$
$\wedge$ i-state-okp (i, *load-addr*)

∧ (i-psw (i) = `'run`)
∧ i-tpop_<x{a}>-okp (i))
→ i-state-okp (i-tpop_<x{a}>-step (i), *load-addr*)

THEOREM: i-state-okp-i-jump_*-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-jump_*-okp (i))
→ i-state-okp (i-jump_*-step (i), *load-addr*)

THEOREM: i-state-okp-i-cpop_pc-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-cpop_pc-okp (i))
→ i-state-okp (i-cpop_pc-step (i), *load-addr*)

THEOREM: i-state-okp-i-move_<x{s}>_<tsp>-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-move_<x{s}>_<tsp>-okp (i))
→ i-state-okp (i-move_<x{s}>_<tsp>-step (i), *load-addr*)

THEOREM: i-state-okp-i-move_<x{a}>_<tsp>-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-move_<x{a}>_<tsp>-okp (i))
→ i-state-okp (i-move_<x{a}>_<tsp>-step (i), *load-addr*)

THEOREM: i-state-okp-i-jump_x{subr}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-jump_x{subr}-okp (i))
→ i-state-okp (i-jump_x{subr}-step (i), *load-addr*)

THEOREM: i-state-okp-i-jump-n_x-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-jump-n_x-okp (i))
→ i-state-okp (i-jump-n_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-jump-nn_x-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-jump-nn_x-okp (i))
$\rightarrow$ i-state-okp (i-jump-nn_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-jump-z_x-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-jump-z_x-okp (i))
$\rightarrow$ i-state-okp (i-jump-z_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-jump-nz_x-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-jump-nz_x-okp (i))
$\rightarrow$ i-state-okp (i-jump-nz_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop_pc-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-tpop_pc-okp (i))
$\rightarrow$ i-state-okp (i-tpop_pc-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop_<c>_x-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-tpop_<c>_x-okp (i))
$\rightarrow$ i-state-okp (i-tpop_<c>_x-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop{n}_<z>_y-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = **'run**)
$\land$ i-tpop{n}_<z>_y-okp (i))
$\rightarrow$ i-state-okp (i-tpop{n}_<z>_y-step (i), *load-addr*)

THEOREM: i-state-okp-i-tpop{i}_<zn>_y-step
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)

∧ (i-psw (*i*) = 'run)
∧ i-tpop{i}_<zn>_y-okp (*i*))
→ i-state-okp (i-tpop{i}_<zn>_y-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-tpop{b}_<z>_y-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ i-tpop{b}_<z>_y-okp (*i*))
→ i-state-okp (i-tpop{b}_<z>_y-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-tpop{v}_<z>_y-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ i-tpop{v}_<z>_y-okp (*i*))
→ i-state-okp (i-tpop{v}_<z>_y-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-add_<tsp>{a}_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ i-add_<tsp>{a}_x{n}-okp (*i*))
→ i-state-okp (i-add_<tsp>{a}_x{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-add_tsp_*{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ i-add_tsp_*{n}-okp (*i*))
→ i-state-okp (i-add_tsp_*{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-add_tsp_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ i-add_tsp_x{n}-okp (*i*))
→ i-state-okp (i-add_tsp_x{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-sub_<tsp>{a}_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ i-sub_<tsp>{a}_x{n}-okp (*i*))
→ i-state-okp (i-sub_<tsp>{a}_x{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-sub_x{s}_y{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = **'run**)
∧ i-sub_x{s}_y{n}-okp (*i*))
→ i-state-okp (i-sub_x{s}_y{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-xor_<tsp>_<tsp>-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = **'run**)
∧ i-xor_<tsp>_<tsp>-okp (*i*))
→ i-state-okp (i-xor_<tsp>_<tsp>-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-move-z_<tsp>_*-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = **'run**)
∧ i-move-z_<tsp>_*-okp (*i*))
→ i-state-okp (i-move-z_<tsp>_*-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-move-n_x_*-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = **'run**)
∧ i-move-n_x_*-okp (*i*))
→ i-state-okp (i-move-n_x_*-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-move_<tsp>_*-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = **'run**)
∧ i-move_<tsp>_*-okp (*i*))
→ i-state-okp (i-move_<tsp>_*-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-incr_<tsp>_<tsp>{i}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = **'run**)
∧ i-incr_<tsp>_<tsp>{i}-okp (*i*))
→ i-state-okp (i-incr_<tsp>_<tsp>{i}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-xor_<z>_<tsp>_x-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)

∧ (i-psw (i) = '**run**)
∧ i-xor_<z>_<tsp>_x-okp (i))
→ i-state-okp (i-xor_<z>_<tsp>_x-step (i, *load-addr*), *load-addr*)

THEOREM: i-state-okp-i-sub_<tsp>{i}_x{i}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = '**run**)
∧ i-sub_<tsp>{i}_x{i}-okp (i))
→ i-state-okp (i-sub_<tsp>{i}_x{i}-step (i), *load-addr*)

THEOREM: i-state-okp-i-decr_<tsp>_<tsp>{i}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = '**run**)
∧ i-decr_<tsp>_<tsp>{i}-okp (i))
→ i-state-okp (i-decr_<tsp>_<tsp>{i}-step (i), *load-addr*)

THEOREM: i-state-okp-i-neg_<tsp>_<tsp>{i}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = '**run**)
∧ i-neg_<tsp>_<tsp>{i}-okp (i))
→ i-state-okp (i-neg_<tsp>_<tsp>{i}-step (i), *load-addr*)

THEOREM: i-state-okp-i-asr_<c>_<tsp>_<tsp>{b}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = '**run**)
∧ i-asr_<c>_<tsp>_<tsp>{b}-okp (i))
→ i-state-okp (i-asr_<c>_<tsp>_<tsp>{b}-step (i), *load-addr*)

THEOREM: i-state-okp-i-addc_<c>_x{n}_y{n}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = '**run**)
∧ i-addc_<c>_x{n}_y{n}-okp (i))
→ i-state-okp (i-addc_<c>_x{n}_y{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-add_<tsp>{n}_x{n}-step
((*load-addr* $\in$ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = '**run**)
∧ i-add_<tsp>{n}_x{n}-okp (i))
→ i-state-okp (i-add_<tsp>{n}_x{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-incr_<tsp>_<tsp>{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ i-incr_<tsp>_<tsp>{n}-okp (*i*))
→ i-state-okp (i-incr_<tsp>_<tsp>{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-incr_y_y{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ i-incr_y_y{n}-okp (*i*))
→ i-state-okp (i-incr_y_y{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-int-to-nat-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ i-int-to-nat-okp (*i*))
→ i-state-okp (i-int-to-nat-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-decr_<tsp>_<tsp>{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ i-decr_<tsp>_<tsp>{n}-okp (*i*))
→ i-state-okp (i-decr_<tsp>_<tsp>{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-add_<c>_x_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ i-add_<c>_x_x{n}-okp (*i*))
→ i-state-okp (i-add_<c>_x_x{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-add_<tsp>_<tsp>{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ i-add_<tsp>_<tsp>{n}-okp (*i*))
→ i-state-okp (i-add_<tsp>_<tsp>{n}-step (*i*), *load-addr*)

THEOREM: i-state-okp-i-lsr_<c>_x_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)

∧ (i-psw (i) = `'run`)
∧ i-lsr_<c>_x_x{n}-okp (i))
→ i-state-okp (i-lsr_<c>_x_x{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-subb_<c>_x{n}_y{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-subb_<c>_x{n}_y{n}-okp (i))
→ i-state-okp (i-subb_<c>_x{n}_y{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-sub_<c>_<tsp>{n}_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-sub_<c>_<tsp>{n}_x{n}-okp (i))
→ i-state-okp (i-sub_<c>_<tsp>{n}_x{n}-step (i), *load-addr*)

```
; I got to here, but ran out of CONS space and can't save my lib.
```

THEOREM: i-state-okp-i-sub_<nv>_<tsp>{i}_x{i}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-sub_<nv>_<tsp>{i}_x{i}-okp (i))
→ i-state-okp (i-sub_<nv>_<tsp>{i}_x{i}-step (i), *load-addr*)

THEOREM: i-state-okp-i-sub_<tsp>{n}_x{n}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-sub_<tsp>{n}_x{n}-okp (i))
→ i-state-okp (i-sub_<tsp>{n}_x{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-sub_<c>_<tsp>{a}_x{a}-step
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ i-sub_<c>_<tsp>{a}_x{a}-okp (i))
→ i-state-okp (i-sub_<c>_<tsp>{a}_x{a}-step (i), *load-addr*)

THEOREM: i-state-okp-i-sub_<tsp>{s}_x{s}-step
((*load-addr* ∈ **N**)

$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-sub_<tsp>{s}_x{s}-okp (i))
$\rightarrow$ i-state-okp (i-sub_<tsp>{s}_x{s}-step (i), *load-addr*)

Theorem: i-state-okp-i-move-c_<tsp>_*-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-move-c_<tsp>_*-okp (i))
$\rightarrow$ i-state-okp (i-move-c_<tsp>_*-step (i), *load-addr*)

Theorem: i-state-okp-i-move-v_<tsp>_*-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-move-v_<tsp>_*-okp (i))
$\rightarrow$ i-state-okp (i-move-v_<tsp>_*-step (i), *load-addr*)

Theorem: i-state-okp-i-or_<tsp>{v}_x{v}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-or_<tsp>{v}_x{v}-okp (i))
$\rightarrow$ i-state-okp (i-or_<tsp>{v}_x{v}-step (i), *load-addr*)

Theorem: i-state-okp-i-and_<tsp>{v}_x{v}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-and_<tsp>{v}_x{v}-okp (i))
$\rightarrow$ i-state-okp (i-and_<tsp>{v}_x{v}-step (i), *load-addr*)

Theorem: i-state-okp-i-not_<tsp>_<tsp>{v}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-not_<tsp>_<tsp>{v}-okp (i))
$\rightarrow$ i-state-okp (i-not_<tsp>_<tsp>{v}-step (i), *load-addr*)

Theorem: i-state-okp-i-xor_<tsp>{v}_x{v}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-xor_<tsp>{v}_x{v}-okp (i))
$\rightarrow$ i-state-okp (i-xor_<tsp>{v}_x{v}-step (i), *load-addr*)

THEOREM: i-state-okp-i-lsr_<tsp>_<tsp>{v}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-lsr_<tsp>_<tsp>{v}-okp (i))
$\rightarrow$ i-state-okp (i-lsr_<tsp>_<tsp>{v}-step (i), *load-addr*)

THEOREM: i-state-okp-i-add_<tsp>_<tsp>{v}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-add_<tsp>_<tsp>{v}-okp (i))
$\rightarrow$ i-state-okp (i-add_<tsp>_<tsp>{v}-step (i), *load-addr*)

THEOREM: i-state-okp-i-or_<tsp>{b}_x{b}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-or_<tsp>{b}_x{b}-okp (i))
$\rightarrow$ i-state-okp (i-or_<tsp>{b}_x{b}-step (i), *load-addr*)

THEOREM: i-state-okp-i-and_<tsp>{b}_x{b}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-and_<tsp>{b}_x{b}-okp (i))
$\rightarrow$ i-state-okp (i-and_<tsp>{b}_x{b}-step (i), *load-addr*)

THEOREM: i-state-okp-i-xor_<tsp>{b}_*{b}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-xor_<tsp>{b}_*{b}-okp (i))
$\rightarrow$ i-state-okp (i-xor_<tsp>{b}_*{b}-step (i), *load-addr*)

THEOREM: i-state-okp-i-xor_<tsp>{b}_x{b}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = **'run**)
$\wedge$ i-xor_<tsp>{b}_x{b}-okp (i))
$\rightarrow$ i-state-okp (i-xor_<tsp>{b}_x{b}-step (i), *load-addr*)

THEOREM: i-state-okp-i-add_pc_x{n}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)

$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-add_pc_x{n}-okp (i))
$\rightarrow$ i-state-okp (i-add_pc_x{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-add_x_x{n}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-add_x_x{n}-okp (i))
$\rightarrow$ i-state-okp (i-add_x_x{n}-step (i), *load-addr*)

THEOREM: i-state-okp-i-addc_<v>_x{i}_y{i}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-addc_<v>_x{i}_y{i}-okp (i))
$\rightarrow$ i-state-okp (i-addc_<v>_x{i}_y{i}-step (i), *load-addr*)

THEOREM: i-state-okp-i-subb_<v>_x{i}_y{i}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-subb_<v>_x{i}_y{i}-okp (i))
$\rightarrow$ i-state-okp (i-subb_<v>_x{i}_y{i}-step (i), *load-addr*)

THEOREM: i-state-okp-i-cpush_cfp-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-cpush_cfp-okp (i))
$\rightarrow$ i-state-okp (i-cpush_cfp-step (i), *load-addr*)

THEOREM: i-state-okp-i-sub_<z>_x{s}_y{s}-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-sub_<z>_x{s}_y{s}-okp (i))
$\rightarrow$ i-state-okp (i-sub_<z>_x{s}_y{s}-step (i), *load-addr*)

THEOREM: i-state-okp-i-ins-step
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = 'run)
$\wedge$ i-ins-okp (*ins*, i))
$\rightarrow$ i-state-okp (i-ins-step (*ins*, i, *load-addr*), *load-addr*)

THEOREM: i-state-okp-i-halt
$\text{i-state-okp}\,(\text{i-halt}\,(i,\ psw),\ load\text{-}addr) = \text{i-state-okp}\,(i,\ load\text{-}addr)$

THEOREM: i-state-okp-i-step
$((load\text{-}addr \in \mathbf{N}) \land \text{i-state-okp}\,(i,\ load\text{-}addr))$
$\to \quad \text{i-state-okp}\,(\text{i-step}\,(i,\ load\text{-}addr),\ load\text{-}addr)$

```
; The link tables in display-data-segment play two roles.  One is
; to let us get at that section of memory containing a certain
; area.  The other is to map the objects in that section up.
; I wish to distinguish these two roles because I need to do some
; inductions in which one of the tables is fixed and the other
; varies.  In addition, I need to separate the process of fetching
; the objects from the process of mapping them up.  So I will define
; the functions that do these separate jobs and prove their equivalence
; to display-data-segment.
```

DEFINITION:
get-array (n, $m\text{-}addr$, $m\text{-}mem$)
= **if** $n \simeq 0$ **then nil**
else cons (get ($m\text{-}addr$, $m\text{-}mem$), get-array ($n - 1$, $1 + m\text{-}addr$, $m\text{-}mem$)) **endif**

DEFINITION:
unlink-data-word-array ($type\text{-}lst$, $obj\text{-}lst$, $link\text{-}tables$)
= **if** $type\text{-}lst \simeq$ **nil then nil**
else cons (unlink-data-word (car ($type\text{-}lst$),
car ($obj\text{-}lst$),
$link\text{-}tables$),
unlink-data-word-array (cdr ($type\text{-}lst$),
cdr ($obj\text{-}lst$),
$link\text{-}tables$)) **endif**

DEFINITION:
get-areas ($type\text{-}spec$, $usr\text{-}data\text{-}links$, $m\text{-}mem$)
= **if** $type\text{-}spec \simeq$ **nil then nil**
else cons (cons (caar ($type\text{-}spec$),
get-array (length (cdar ($type\text{-}spec$)),
base-address (caar ($type\text{-}spec$),
$usr\text{-}data\text{-}links$),
$m\text{-}mem$)),
get-areas (cdr ($type\text{-}spec$), $usr\text{-}data\text{-}links$, $m\text{-}mem$)) **endif**

DEFINITION:
unlink-areas ($type\text{-}spec$, $segment$, $link\text{-}tables$)

```
=   if type-spec ≃ nil then nil
    else cons (cons (caar (type-spec),
                     unlink-data-word-array (cdar (type-spec),
                                             cdar (segment),
                                             link-tables)),
               unlink-areas (cdr (type-spec), cdr (segment), link-tables)) endif
```

DEFINITION:

```
display-array (type-lst, n, m-mem, link-tables)
=   if type-lst ≃ nil then nil
    else cons (unlink-data-word (car (type-lst),
                                 get (n, m-mem),
                                 link-tables),
               display-array (cdr (type-lst), 1 + n, m-mem, link-tables)) endif
```

DEFINITION:

```
display-data-area (area-type-spec, m-mem, link-tables)
=   cons (car (area-type-spec),
          display-array (cdr (area-type-spec),
                         base-address (car (area-type-spec),
                                       usr-data-links (link-tables)),
                         m-mem,
                         link-tables))
```

DEFINITION:

```
display-data-segment (type-spec, m-mem, link-tables)
=   if type-spec ≃ nil then nil
    else cons (display-data-area (car (type-spec), m-mem, link-tables),
               display-data-segment (cdr (type-spec), m-mem, link-tables)) endif
```

DEFINITION:

```
display-m-data-segment (m, type-spec, link-tables)
=   display-data-segment (type-spec, m-mem (m), link-tables)
```

THEOREM: display-array-decomposition

```
display-array (type-lst, m-addr, m-mem, link-tables)
=   unlink-data-word-array (type-lst,
                            get-array (length (type-lst), m-addr, m-mem),
                            link-tables)
```

THEOREM: display-data-segment-decomposition

```
display-data-segment (type-spec, m-mem, i-link-tables (i, load-addr))
=   unlink-areas (type-spec,
                  get-areas (type-spec,
                             link-table-for-segment (i-usr-data-segment (i),
```

$$\begin{aligned}&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{load-addr}),\\ &\qquad\qquad\qquad\qquad\qquad \textit{m-mem}),\\ &\qquad\qquad\qquad \text{i-link-tables}\,(i,\ \textit{load-addr}))\end{aligned}$$

```
; The key steps to proving that display-data-segment inverts is to
; show that (a) get-areas retrieves the linked areas and (b)
; unlink-areas inverts link-areas.  We begin with the get-segment
; part.

; Link-areas, mentioned above, is like link-segment, only it
; preserves the areas instead of flattening them.
```

DEFINITION:
link-areas (*segment*, *link-tables*, *word-size*)
$$\begin{aligned}=\ \ & \textbf{if}\ \textit{segment} \simeq \textbf{nil}\ \ \textbf{then nil}\\ & \textbf{else}\ \text{cons}\,(\text{cons}\,(\text{caar}\,(\textit{segment}),\\ &\qquad\qquad\qquad\quad \text{link-area}\,(\text{cdar}\,(\textit{segment}),\ \textit{link-tables},\ \textit{word-size})),\\ &\qquad\quad\ \ \text{link-areas}\,(\text{cdr}\,(\textit{segment}),\ \textit{link-tables},\ \textit{word-size}))\ \textbf{endif}\end{aligned}$$

```
; To show that get-areas on a linked memory gives link-areas, it is
; necessary to know that get-array gets a link-area.
```

DEFINITION:
cdr-n (n, *lst*)
$$\begin{aligned}=\ \ & \textbf{if}\ n \simeq \mathsf{0}\ \ \textbf{then}\ \textit{lst}\\ & \textbf{else}\ \text{cdr-n}\,(n-1,\ \text{cdr}\,(\textit{lst}))\ \textbf{endif}\end{aligned}$$

EVENT: Enable equal-length-0.

THEOREM: lessp-link-table-entry
$$\begin{aligned}&(\text{definedp}\,(\textit{name},\ \textit{segment}) \land \text{listp}\,(\text{cdr}\,(\text{assoc}\,(\textit{name},\ \textit{segment}))))\\ \to\ \ & (\text{link-table-entry}\,(\textit{name},\ \textit{segment}) < \text{segment-length}\,(\textit{segment}))\end{aligned}$$

THEOREM: get-link-table-entry-link-segment-generalized
$$\begin{aligned}&(\text{definedp}\,(\textit{name},\ \textit{segment}) \land (i < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\textit{name},\ \textit{segment})))))\\ \to\ \ & (\text{get}\,(i + \text{link-table-entry}\,(\textit{name},\ \textit{segment}),\\ &\qquad\ \text{link-segment}\,(\textit{segment},\ \textit{table},\ \textit{word-size}))\\ &\quad =\ \text{link-word}\,(\text{unlabel}\,(\text{get}\,(i,\ \text{cdr}\,(\text{assoc}\,(\textit{name},\ \textit{segment})))),\\ &\qquad\qquad\qquad \textit{table},\\ &\qquad\qquad\qquad \textit{word-size}))\end{aligned}$$

THEOREM: plus-equal-0
$$((x + y) = \mathsf{0}) = ((x \simeq \mathsf{0}) \land (y \simeq \mathsf{0}))$$

THEOREM: car-cdr-n-is-get
car (cdr-n (n, *lst*)) = get (n, *lst*)

THEOREM: cdr-cdr-n
cdr (cdr-n (n, *lst*)) = cdr-n (n, cdr (*lst*))

```
; This lemma was excrutiatingly hard for me to get.
; The reason, I think, is that I've just come back to
; the im level proofs after 3 months of working elsewhere
; and I no longer have the required mastery of the rules.

; This theorem was hard to invent too.  The main reason is
; that I had to get a sufficiently general version to permit
; inductive proof.  The trick was the introduction of i.
```

DEFINITION:
first-n (n, x)
= **if** $n \simeq 0$ **then nil**
else cons (car (x), first-n ($n - 1$, cdr (x))) **endif**

THEOREM: get-array-link-mem
(definedp (*name*, *usr-data-segment*)
∧ (length (cdr (assoc (*name*, *usr-data-segment*))) $\not<$ (i + n))
∧ ($i \in \mathbf{N}$)
∧ ($n \in \mathbf{N}$))
→ (get-array (n,
load-addr + (i + link-table-entry (*name*,
usr-data-segment)),
append (boot-code (*boot-lst*, *load-addr*, *word-size*),
append (link-segment (*usr-data-segment*,
tables,
word-size),
rest)))
= link-area (first-n (n,
cdr-n (i, cdr (assoc (*name*, *usr-data-segment*)))),
tables,
word-size))

```
; Now I instantiate the above theorem to get the rule I need.

; The following lemma has a link-area around a familiar identity
; just so I can avoid bringing up a properp hypothesis.
```

THEOREM: first-n-length-embedded
link-area (first-n (length (lst), lst), $tables$, $word\text{-}size$)
= link-area (lst, $tables$, $word\text{-}size$)

THEOREM: get-array-link-mem-instance
(definedp ($name$, $usr\text{-}data\text{-}segment$)
∧ (n = length (cdr (assoc ($name$, $usr\text{-}data\text{-}segment$)))))
→ (get-array (n,
$load\text{-}addr$ + link-table-entry ($name$, $usr\text{-}data\text{-}segment$),
append (boot-code ($boot\text{-}lst$, $load\text{-}addr$, $word\text{-}size$),
append (link-segment ($usr\text{-}data\text{-}segment$,
$tables$,
$word\text{-}size$),
$rest$)))
= link-area (cdr (assoc ($name$, $usr\text{-}data\text{-}segment$)), $tables$, $word\text{-}size$))

```
; To formulate a suitably general version of the get-areas
; theorem I need the concept that one segment is an extension
; of another.  By that I mean that everything that occurs in
; the one is identically defined in the other.
; Notice that this concept is not reflexive
; unless we know the segment contains no duplicate area names.
; This whole concept is fairly subtle in its interaction with
; the proof of the get-areas lemma.  The first three lemmas
; below are the crucial properties of subsegmentp for that
; proof.  Reflexivity is needed only for the final use of
; the get-areas theorem.
```

DEFINITION:
subsegmentp ($segment1$, $segment2$)
= **if** $segment1 \simeq$ **nil then t**
else definedp (caar ($segment1$), $segment2$)
∧ (assoc (caar ($segment1$), $segment2$) = car ($segment1$))
∧ subsegmentp (cdr ($segment1$), $segment2$) **endif**

THEOREM: subsegmentp-cdr
(listp (x) ∧ subsegmentp (x, y)) → subsegmentp (cdr (x), y)

THEOREM: subsegmentp-implies-definedp
(listp ($segment$) ∧ subsegmentp ($segment$, $usr\text{-}data\text{-}segment$))
→ definedp (caar ($segment$), $usr\text{-}data\text{-}segment$)

THEOREM: subsegmentp-implies-equal-assoc
(listp ($segment$) ∧ subsegmentp ($segment$, $usr\text{-}data\text{-}segment$))
→ (assoc (caar ($segment$), $usr\text{-}data\text{-}segment$) = car ($segment$))

THEOREM: subsegmentp-reflexive-lemma
$(\text{subsegmentp}\,(x,\ \text{cdr}\,(y)) \wedge (\neg\ \text{definedp}\,(\text{caar}\,(y),\ x))) \rightarrow \text{subsegmentp}\,(x,\ y)$

THEOREM: subsegmentp-reflexive
$\text{proper-i-usr-data-segmentp}\,(\textit{segment},\ i) \rightarrow \text{subsegmentp}\,(\textit{segment},\ \textit{segment})$

THEOREM: length-type-lst
$\text{length}\,(\text{type-lst}\,(\textit{area})) = \text{length}\,(\textit{area})$

```
; The following theorem is key step (a) above, when the theorem is
; instantiated with segment replaced by usr-data-segment.
```

THEOREM: get-areas-link-mem

$$
\begin{array}{l}
((\textit{load-addr} \in \mathbf{N}) \wedge \text{subsegmentp}\,(\textit{segment},\ \textit{usr-data-segment})) \\
\rightarrow \quad (\text{get-areas}\,(\text{type-specification}\,(\textit{segment}), \\
\qquad\qquad\qquad \text{link-table-for-segment}\,(\textit{usr-data-segment},\ \textit{load-addr}), \\
\qquad\qquad\qquad \text{append}\,(\text{boot-code}\,(\textit{boot-lst},\ \textit{load-addr},\ \textit{word-size}), \\
\qquad\qquad\qquad\qquad \text{append}\,(\text{link-segment}\,(\textit{usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{tables}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{word-size}), \\
\qquad\qquad\qquad\qquad\qquad \textit{rest}))) \\
\qquad = \quad \text{link-areas}\,(\textit{segment},\ \textit{tables},\ \textit{word-size}))
\end{array}
$$

THEOREM: not-icode-instructionp
$\text{i-objectp}\,(x,\ i) \rightarrow (\neg\ \text{icode-instructionp}\,(x))$

```
; We now have to prove that if i-state-okp holds then i-word-size is
; non-0.  This is necessary because otherwise booleans don't invert!
```

THEOREM: cstk-implies-not-zerop-segment-length

$$
\begin{array}{l}
(n < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment})))) \\
\rightarrow \quad (\text{segment-length}\,(\textit{segment}) \not\simeq \texttt{0})
\end{array}
$$

THEOREM: i-loadablep-and-cstk-implies-not-zerop-i-word-size

$$
\begin{array}{l}
(\text{i-loadablep}\,(i,\ \textit{load-addr}) \\
\ \wedge \quad (\text{adp-offset}\,(\text{untag}\,(\text{i-cfp}\,(i))) \\
\qquad < \quad \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \text{i-sys-data-segment}\,(i)))))) \\
\rightarrow \quad ((\text{i-word-size}\,(i) \in \mathbf{N}) \wedge (\texttt{0} < \text{i-word-size}\,(i)))
\end{array}
$$

THEOREM: car-cdr-elim-accelerator

$$
\begin{array}{l}
(\text{listp}\,(x) \wedge (\text{cddr}\,(x) = \mathbf{nil}) \wedge (\textit{type} = \text{car}\,(x))) \\
\rightarrow \quad (\text{list}\,(\textit{type},\ \text{cadr}\,(x)) = x)
\end{array}
$$

```
; Here is the key invertibility lemma.

; The list of disabled functions, after icode-instructionp, in the
; hint below was mechanically generated.  It contains all the
; functions used to link and unlink individual data types.  The list
; could be shortened.
```

THEOREM: unlink-data-word-link-data-word
$((\mathit{load\text{-}addr} \in \mathbf{N}) \wedge \text{i-usr-data-objectp}\,(x,\ i) \wedge \text{i-state-okp}\,(i,\ \mathit{load\text{-}addr}))$
$\rightarrow$ (unlink-data-word (type (x),
link-data-word (x,
i-link-tables (i, *load-addr*),
i-word-size (i)),
i-link-tables (i, *load-addr*))
$=$ x)

THEOREM: unlink-data-word-array-link-area
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge$ all-i-usr-data-objectps (*array*, i)
$\wedge$ i-state-okp (i, *load-addr*))
$\rightarrow$ (unlink-data-word-array (type-lst (*array*),
link-area (*array*,
i-link-tables (i, *load-addr*),
i-word-size (i)),
i-link-tables (i, *load-addr*))
$=$ *array*)

THEOREM: unlink-areas-link-areas
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge$ proper-i-usr-data-segmentp (*segment*, i)
$\wedge$ i-state-okp (i, *load-addr*))
$\rightarrow$ (unlink-areas (type-specification (*segment*),
link-areas (*segment*,
i-link-tables (i, *load-addr*),
i-word-size (i)),
i-link-tables (i, *load-addr*))
$=$ *segment*)

THEOREM: display-m-data-segment-inverts-i->m
$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge$ proper-i-usr-data-segmentp (i-usr-data-segment (i), i)
$\wedge$ i-state-okp (i, *load-addr*))
$\rightarrow$ (display-m-data-segment (i->m (i, *boot-lst*, *load-addr*),
type-specification (i-usr-data-segment (i)),

i-link-tables (*i*, *load-addr*))
= i-usr-data-segment (*i*))

THEOREM: i-link-tables-step-move_x_x
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-move_x_x-okp (*i*)
∧ (i-current-instruction (*i*) = `'(move_x_x)`))
→ (i-link-tables (i-move_x_x-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-cpush_*
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-cpush_*-okp (*i*)
∧ (i-current-instruction (*i*) = `'(cpush_*)`))
→ (i-link-tables (i-cpush_*-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-add_<tsp>{i}_x{i}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-add_<tsp>{i}_x{i}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(add_<tsp>{i}_x{i})`))
→ (i-link-tables (i-add_<tsp>{i}_x{i}-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-cpush_cfp
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = 32)
∧ i-cpush_cfp-okp (*i*)
∧ (i-current-instruction (*i*) = `'(cpush_cfp)`))
→ (i-link-tables (i-cpush_cfp-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-move_cfp_csp
((*load-addr* ∈ **N**)

$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-move_cfp_csp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_cfp_csp)`))
$\rightarrow$ (i-link-tables (i-move_cfp_csp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-cpush_<tsp>+
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-cpush_<tsp>+-okp (i)
$\wedge$ (i-current-instruction (i) = `'(cpush_<tsp>+)`))
$\rightarrow$ (i-link-tables (i-cpush_<tsp>+-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_csp_cfp
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-move_csp_cfp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_csp_cfp)`))
$\rightarrow$ (i-link-tables (i-move_csp_cfp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_x_tsp
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-move_x_tsp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_x_tsp)`))
$\rightarrow$ (i-link-tables (i-move_x_tsp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-cpop_cfp
(($\mathit{load\text{-}addr} \in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-cpop_cfp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(cpop_cfp)`))

$\rightarrow$ (i-link-tables (i-cpop_cfp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_x_*
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_x_*-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_x_*)`))
$\rightarrow$ (i-link-tables (i-move_x_*-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_y_*
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_y_*-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_y_*)`))
$\rightarrow$ (i-link-tables (i-move_y_*-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_x{n}_csp
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-add_x{n}_csp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_x{n}_csp)`))
$\rightarrow$ (i-link-tables (i-add_x{n}_csp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_x_<x{s}>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_x_<x{s}>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_x_<x{s}>)`))
$\rightarrow$ (i-link-tables (i-move_x_<x{s}>-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_y_<y{s}>
((*load-addr* $\in$ **N**)

$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_y_<y{s}>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_y_<y{s}>)`))
$\rightarrow$ (i-link-tables (i-move_y_<y{s}>-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_y_tsp
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_y_tsp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_y_tsp)`))
$\rightarrow$ (i-link-tables (i-move_y_tsp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpush_csp
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpush_csp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpush_csp)`))
$\rightarrow$ (i-link-tables (i-tpush_csp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpush_tsp
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpush_tsp-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpush_tsp)`))
$\rightarrow$ (i-link-tables (i-tpush_tsp-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpush_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpush_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpush_x)`))

$\rightarrow$ (i-link-tables (i-tpush_x-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpush_<x{s}>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpush_<x{s}>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpush_<x{s}>)`))
$\rightarrow$ (i-link-tables (i-tpush_<x{s}>-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpush_<x{a}>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpush_<x{a}>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpush_<x{a}>)`))
$\rightarrow$ (i-link-tables (i-tpush_<x{a}>-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpush_*
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpush_*-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpush_*)`))
$\rightarrow$ (i-link-tables (i-tpush_*-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop_x
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpop_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpop_x)`))
$\rightarrow$ (i-link-tables (i-tpop_x-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop_y
((*load-addr* $\in$ **N**)

$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) $=$ `'run`)
$\wedge$ (i-word-size (i) $=$ `32`)
$\wedge$ i-tpop_y-okp (i)
$\wedge$ (i-current-instruction (i) $=$ `'(tpop_y)`))
$\rightarrow$ (i-link-tables (i-tpop_y-step (i), *load-addr*)
$=$ i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop_<x{s}>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) $=$ `'run`)
$\wedge$ (i-word-size (i) $=$ `32`)
$\wedge$ i-tpop_<x{s}>-okp (i)
$\wedge$ (i-current-instruction (i) $=$ `'(tpop_<x{s}>)`))
$\rightarrow$ (i-link-tables (i-tpop_<x{s}>-step (i), *load-addr*)
$=$ i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop_<x{a}>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) $=$ `'run`)
$\wedge$ (i-word-size (i) $=$ `32`)
$\wedge$ i-tpop_<x{a}>-okp (i)
$\wedge$ (i-current-instruction (i) $=$ `'(tpop_<x{a}>)`))
$\rightarrow$ (i-link-tables (i-tpop_<x{a}>-step (i), *load-addr*)
$=$ i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-jump_*
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) $=$ `'run`)
$\wedge$ (i-word-size (i) $=$ `32`)
$\wedge$ i-jump_*-okp (i)
$\wedge$ (i-current-instruction (i) $=$ `'(jump_*)`))
$\rightarrow$ (i-link-tables (i-jump_*-step (i), *load-addr*)
$=$ i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-cpop_pc
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) $=$ `'run`)
$\wedge$ (i-word-size (i) $=$ `32`)
$\wedge$ i-cpop_pc-okp (i)
$\wedge$ (i-current-instruction (i) $=$ `'(cpop_pc)`))

$\rightarrow$ (i-link-tables (i-cpop_pc-step (i), *load-addr*)
    = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_<x{s}>_<tsp>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_<x{s}>_<tsp>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_<x{s}>_<tsp>)`))
$\rightarrow$ (i-link-tables (i-move_<x{s}>_<tsp>-step (i), *load-addr*)
    = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_<x{a}>_<tsp>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move_<x{a}>_<tsp>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move_<x{a}>_<tsp>)`))
$\rightarrow$ (i-link-tables (i-move_<x{a}>_<tsp>-step (i), *load-addr*)
    = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-jump_x{subr}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump_x{subr}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump_x{subr})`))
$\rightarrow$ (i-link-tables (i-jump_x{subr}-step (i), *load-addr*)
    = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-jump-n_x
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump-n_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump-n_x)`))
$\rightarrow$ (i-link-tables (i-jump-n_x-step (i), *load-addr*)
    = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-jump-nn_x
((*load-addr* $\in$ **N**)

$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump-nn_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump-nn_x)`))
$\rightarrow$ (i-link-tables (i-jump-nn_x-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-jump-z_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump-z_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump-z_x)`))
$\rightarrow$ (i-link-tables (i-jump-z_x-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-jump-nz_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-jump-nz_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(jump-nz_x)`))
$\rightarrow$ (i-link-tables (i-jump-nz_x-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop_pc
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpop_pc-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpop_pc)`))
$\rightarrow$ (i-link-tables (i-tpop_pc-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop_<c>_x
((*load-addr* $\in \mathbf{N}$)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-tpop_<c>_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(tpop_<c>_x)`))

→ (i-link-tables (i-tpop_<c>_x-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop{n}_<z>_y
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-tpop{n}_<z>_y-okp (i)
∧ (i-current-instruction (i) = `'(tpop{n}_<z>_y)`))
→ (i-link-tables (i-tpop{n}_<z>_y-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop{i}_<zn>_y
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-tpop{i}_<zn>_y-okp (i)
∧ (i-current-instruction (i) = `'(tpop{i}_<zn>_y)`))
→ (i-link-tables (i-tpop{i}_<zn>_y-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop{b}_<z>_y
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-tpop{b}_<z>_y-okp (i)
∧ (i-current-instruction (i) = `'(tpop{b}_<z>_y)`))
→ (i-link-tables (i-tpop{b}_<z>_y-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-tpop{v}_<z>_y
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-tpop{v}_<z>_y-okp (i)
∧ (i-current-instruction (i) = `'(tpop{v}_<z>_y)`))
→ (i-link-tables (i-tpop{v}_<z>_y-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_<tsp>{a}_x{n}
((*load-addr* ∈ **N**)

$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-add_<tsp>{a}_x{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_<tsp>{a}_x{n})`))
$\rightarrow$ (i-link-tables (i-add_<tsp>{a}_x{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_tsp_*{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-add_tsp_*{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_tsp_*{n})`))
$\rightarrow$ (i-link-tables (i-add_tsp_*{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_tsp_x{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-add_tsp_x{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_tsp_x{n})`))
$\rightarrow$ (i-link-tables (i-add_tsp_x{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-sub_<tsp>{a}_x{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-sub_<tsp>{a}_x{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(sub_<tsp>{a}_x{n})`))
$\rightarrow$ (i-link-tables (i-sub_<tsp>{a}_x{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-sub_x{s}_y{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = 32)
$\wedge$ i-sub_x{s}_y{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(sub_x{s}_y{n})`))

$\rightarrow$ (i-link-tables (i-sub_x{s}_y{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-xor_<z>_<tsp>_x
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-xor_<z>_<tsp>_x-okp (i)
$\wedge$ (i-current-instruction (i) = `'(xor_<z>_<tsp>_x)`))
$\rightarrow$ (i-link-tables (i-xor_<z>_<tsp>_x-step (i, *load-addr*), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-xor_<tsp>_<tsp>
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-xor_<tsp>_<tsp>-okp (i)
$\wedge$ (i-current-instruction (i) = `'(xor_<tsp>_<tsp>)`))
$\rightarrow$ (i-link-tables (i-xor_<tsp>_<tsp>-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move-z_<tsp>_*
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move-z_<tsp>_*-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move-z_<tsp>_*)`))
$\rightarrow$ (i-link-tables (i-move-z_<tsp>_*-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move-n_x_*
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-move-n_x_*-okp (i)
$\wedge$ (i-current-instruction (i) = `'(move-n_x_*)`))
$\rightarrow$ (i-link-tables (i-move-n_x_*-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move_<tsp>_*
((*load-addr* $\in$ **N**)

∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-move_<tsp>_*-okp (i)
∧ (i-current-instruction (i) = `'(move_<tsp>_*)`))
→ (i-link-tables (i-move_<tsp>_*-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-incr_<tsp>_<tsp>{i}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-incr_<tsp>_<tsp>{i}-okp (i)
∧ (i-current-instruction (i) = `'(incr_<tsp>_<tsp>{i})`))
→ (i-link-tables (i-incr_<tsp>_<tsp>{i}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-sub_<tsp>{i}_x{i}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-sub_<tsp>{i}_x{i}-okp (i)
∧ (i-current-instruction (i) = `'(sub_<tsp>{i}_x{i})`))
→ (i-link-tables (i-sub_<tsp>{i}_x{i}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-decr_<tsp>_<tsp>{i}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-decr_<tsp>_<tsp>{i}-okp (i)
∧ (i-current-instruction (i) = `'(decr_<tsp>_<tsp>{i})`))
→ (i-link-tables (i-decr_<tsp>_<tsp>{i}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-neg_<tsp>_<tsp>{i}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = 32)
∧ i-neg_<tsp>_<tsp>{i}-okp (i)
∧ (i-current-instruction (i) = `'(neg_<tsp>_<tsp>{i})`))

$\rightarrow$ (i-link-tables (i-neg_<tsp>_<tsp>{i}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-asr_<c>_<tsp>_<tsp>{b}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-asr_<c>_<tsp>_<tsp>{b}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(asr_<c>_<tsp>_<tsp>{b})`))
$\rightarrow$ (i-link-tables (i-asr_<c>_<tsp>_<tsp>{b}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-addc_<c>_x{n}_y{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-addc_<c>_x{n}_y{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(addc_<c>_x{n}_y{n})`))
$\rightarrow$ (i-link-tables (i-addc_<c>_x{n}_y{n}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-add_<tsp>{n}_x{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-add_<tsp>{n}_x{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(add_<tsp>{n}_x{n})`))
$\rightarrow$ (i-link-tables (i-add_<tsp>{n}_x{n}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-incr_<tsp>_<tsp>{n}
((*load-addr* $\in$ **N**)
$\wedge$ i-state-okp (i, *load-addr*)
$\wedge$ (i-psw (i) = `'run`)
$\wedge$ (i-word-size (i) = `32`)
$\wedge$ i-incr_<tsp>_<tsp>{n}-okp (i)
$\wedge$ (i-current-instruction (i) = `'(incr_<tsp>_<tsp>{n})`))
$\rightarrow$ (i-link-tables (i-incr_<tsp>_<tsp>{n}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-incr_y_y{n}
((*load-addr* $\in$ **N**)

$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-incr_y_y{n}-okp (i)
$\land$ (i-current-instruction (i) = `'(incr_y_y{n})`))
$\rightarrow$ (i-link-tables (i-incr_y_y{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-int-to-nat
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-int-to-nat-okp (i)
$\land$ (i-current-instruction (i) = `'(int-to-nat)`))
$\rightarrow$ (i-link-tables (i-int-to-nat-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-decr_<tsp>_<tsp>{n}
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-decr_<tsp>_<tsp>{n}-okp (i)
$\land$ (i-current-instruction (i) = `'(decr_<tsp>_<tsp>{n})`))
$\rightarrow$ (i-link-tables (i-decr_<tsp>_<tsp>{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-add_<c>_x_x{n}
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-add_<c>_x_x{n}-okp (i)
$\land$ (i-current-instruction (i) = `'(add_<c>_x_x{n})`))
$\rightarrow$ (i-link-tables (i-add_<c>_x_x{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

Theorem: i-link-tables-step-add_<tsp>_<tsp>{n}
((*load-addr* $\in \mathbf{N}$)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-add_<tsp>_<tsp>{n}-okp (i)
$\land$ (i-current-instruction (i) = `'(add_<tsp>_<tsp>{n})`))

→ (i-link-tables (i-add_<tsp>_<tsp>{n}-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-lsr_<c>_x_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-lsr_<c>_x_x{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(lsr_<c>_x_x{n})`))
→ (i-link-tables (i-lsr_<c>_x_x{n}-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-subb_<c>_x{n}_y{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-subb_<c>_x{n}_y{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(subb_<c>_x{n}_y{n})`))
→ (i-link-tables (i-subb_<c>_x{n}_y{n}-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-sub_<c>_<tsp>{n}_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-sub_<c>_<tsp>{n}_x{n}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(sub_<c>_<tsp>{n}_x{n})`))
→ (i-link-tables (i-sub_<c>_<tsp>{n}_x{n}-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-sub_<nv>_<tsp>{i}_x{i}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = `'run`)
∧ (i-word-size (*i*) = `32`)
∧ i-sub_<nv>_<tsp>{i}_x{i}-okp (*i*)
∧ (i-current-instruction (*i*) = `'(sub_<nv>_<tsp>{i}_x{i})`))
→ (i-link-tables (i-sub_<nv>_<tsp>{i}_x{i}-step (*i*), *load-addr*)
= i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-sub_<tsp>{n}_x{n}
((*load-addr* ∈ **N**)

∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<tsp>{n}_x{n}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<tsp>{n}_x{n})))
→ (i-link-tables (i-sub_<tsp>{n}_x{n}-step (*i*), *load-addr*)
  = i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-sub_<c>_<tsp>{a}_x{a}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<c>_<tsp>{a}_x{a}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<c>_<tsp>{a}_x{a})))
→ (i-link-tables (i-sub_<c>_<tsp>{a}_x{a}-step (*i*), *load-addr*)
  = i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-sub_<tsp>{s}_x{s}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<tsp>{s}_x{s}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<tsp>{s}_x{s})))
→ (i-link-tables (i-sub_<tsp>{s}_x{s}-step (*i*), *load-addr*)
  = i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-sub_<z>_x{s}_y{s}
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-sub_<z>_x{s}_y{s}-okp (*i*)
∧ (i-current-instruction (*i*) = '(sub_<z>_x{s}_y{s})))
→ (i-link-tables (i-sub_<z>_x{s}_y{s}-step (*i*), *load-addr*)
  = i-link-tables (*i*, *load-addr*))

THEOREM: i-link-tables-step-move-c_<tsp>_*
((*load-addr* ∈ **N**)
∧ i-state-okp (*i*, *load-addr*)
∧ (i-psw (*i*) = 'run)
∧ (i-word-size (*i*) = 32)
∧ i-move-c_<tsp>_*-okp (*i*)
∧ (i-current-instruction (*i*) = '(move-c_<tsp>_*)))

→ (i-link-tables (i-move-c_<tsp>_*-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-move-v_<tsp>_*
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-move-v_<tsp>_*-okp (i)
∧ (i-current-instruction (i) = `'(move-v_<tsp>_*)`))
→ (i-link-tables (i-move-v_<tsp>_*-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-or_<tsp>{v}_x{v}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-or_<tsp>{v}_x{v}-okp (i)
∧ (i-current-instruction (i) = `'(or_<tsp>{v}_x{v})`))
→ (i-link-tables (i-or_<tsp>{v}_x{v}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-and_<tsp>{v}_x{v}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-and_<tsp>{v}_x{v}-okp (i)
∧ (i-current-instruction (i) = `'(and_<tsp>{v}_x{v})`))
→ (i-link-tables (i-and_<tsp>{v}_x{v}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-not_<tsp>_<tsp>{v}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-not_<tsp>_<tsp>{v}-okp (i)
∧ (i-current-instruction (i) = `'(not_<tsp>_<tsp>{v})`))
→ (i-link-tables (i-not_<tsp>_<tsp>{v}-step (i), *load-addr*)
  = i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-xor_<tsp>{v}_x{v}
((*load-addr* ∈ **N**)

$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-xor_<tsp>{v}_x{v}-okp (i)
$\land$ (i-current-instruction (i) = `'(xor_<tsp>{v}_x{v})`))
$\to$ (i-link-tables (i-xor_<tsp>{v}_x{v}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-lsr_<tsp>_<tsp>{v}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-lsr_<tsp>_<tsp>{v}-okp (i)
$\land$ (i-current-instruction (i) = `'(lsr_<tsp>_<tsp>{v})`))
$\to$ (i-link-tables (i-lsr_<tsp>_<tsp>{v}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_<tsp>_<tsp>{v}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-add_<tsp>_<tsp>{v}-okp (i)
$\land$ (i-current-instruction (i) = `'(add_<tsp>_<tsp>{v})`))
$\to$ (i-link-tables (i-add_<tsp>_<tsp>{v}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-or_<tsp>{b}_x{b}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-or_<tsp>{b}_x{b}-okp (i)
$\land$ (i-current-instruction (i) = `'(or_<tsp>{b}_x{b})`))
$\to$ (i-link-tables (i-or_<tsp>{b}_x{b}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-and_<tsp>{b}_x{b}
((*load-addr* $\in$ **N**)
$\land$ i-state-okp (i, *load-addr*)
$\land$ (i-psw (i) = `'run`)
$\land$ (i-word-size (i) = `32`)
$\land$ i-and_<tsp>{b}_x{b}-okp (i)
$\land$ (i-current-instruction (i) = `'(and_<tsp>{b}_x{b})`))

→ (i-link-tables (i-and_<tsp>{b}_x{b}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-xor_<tsp>{b}_*{b}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-xor_<tsp>{b}_*{b}-okp (i)
∧ (i-current-instruction (i) = `'(xor_<tsp>{b}_*{b})`))
→ (i-link-tables (i-xor_<tsp>{b}_*{b}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-xor_<tsp>{b}_x{b}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-xor_<tsp>{b}_x{b}-okp (i)
∧ (i-current-instruction (i) = `'(xor_<tsp>{b}_x{b})`))
→ (i-link-tables (i-xor_<tsp>{b}_x{b}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_pc_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-add_pc_x{n}-okp (i)
∧ (i-current-instruction (i) = `'(add_pc_x{n})`))
→ (i-link-tables (i-add_pc_x{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-add_x_x{n}
((*load-addr* ∈ **N**)
∧ i-state-okp (i, *load-addr*)
∧ (i-psw (i) = `'run`)
∧ (i-word-size (i) = `32`)
∧ i-add_x_x{n}-okp (i)
∧ (i-current-instruction (i) = `'(add_x_x{n})`))
→ (i-link-tables (i-add_x_x{n}-step (i), *load-addr*)
= i-link-tables (i, *load-addr*))

THEOREM: i-link-tables-step-addc_<v>_x{i}_y{i}
((*load-addr* ∈ **N**)

$\wedge\quad \text{i-state-okp}\,(i,\ \textit{load-addr})$
$\wedge\quad (\text{i-psw}\,(i) = \texttt{'run})$
$\wedge\quad (\text{i-word-size}\,(i) = \texttt{32})$
$\wedge\quad \text{i-addc\_<v>\_x\{i\}\_y\{i\}-okp}\,(i)$
$\wedge\quad (\text{i-current-instruction}\,(i) = \texttt{'(addc\_<v>\_x\{i\}\_y\{i\}))})$
$\rightarrow\quad (\text{i-link-tables}\,(\text{i-addc\_<v>\_x\{i\}\_y\{i\}-step}\,(i),\ \textit{load-addr})$
$\qquad = \quad \text{i-link-tables}\,(i,\ \textit{load-addr}))$

THEOREM: i-link-tables-step-subb_<v>_x{i}_y{i}
$((\textit{load-addr} \in \mathbf{N})$
$\wedge\quad \text{i-state-okp}\,(i,\ \textit{load-addr})$
$\wedge\quad (\text{i-psw}\,(i) = \texttt{'run})$
$\wedge\quad (\text{i-word-size}\,(i) = \texttt{32})$
$\wedge\quad \text{i-subb\_<v>\_x\{i\}\_y\{i\}-okp}\,(i)$
$\wedge\quad (\text{i-current-instruction}\,(i) = \texttt{'(subb\_<v>\_x\{i\}\_y\{i\}))})$
$\rightarrow\quad (\text{i-link-tables}\,(\text{i-subb\_<v>\_x\{i\}\_y\{i\}-step}\,(i),\ \textit{load-addr})$
$\qquad = \quad \text{i-link-tables}\,(i,\ \textit{load-addr}))$

THEOREM: i-link-tables-ins-okp-step
$((\textit{load-addr} \in \mathbf{N})$
$\wedge\quad \text{i-state-okp}\,(i,\ \textit{load-addr})$
$\wedge\quad (\text{i-psw}\,(i) = \texttt{'run})$
$\wedge\quad (\text{i-word-size}\,(i) = \texttt{32})$
$\wedge\quad \text{i-ins-okp}\,(\text{i-current-instruction}\,(i),\ i))$
$\rightarrow\quad (\text{i-link-tables}\,(\text{i-ins-step}\,(\text{i-current-instruction}\,(i),\ i,\ \textit{load-addr}),$
$\qquad\qquad \textit{load-addr})$
$\qquad = \quad \text{i-link-tables}\,(i,\ \textit{load-addr}))$

THEOREM: i-link-tables-step
$((\textit{load-addr} \in \mathbf{N})$
$\wedge\quad (\text{i-psw}\,(\text{i-step}\,(i,\ \textit{load-addr})) = \texttt{'run})$
$\wedge\quad (\text{i-word-size}\,(i) = \texttt{32}))$
$\rightarrow\quad (\text{i-link-tables}\,(\text{i-step}\,(i,\ \textit{load-addr}),\ \textit{load-addr})$
$\qquad = \quad \text{i-link-tables}\,(i,\ \textit{load-addr}))$

THEOREM: i-link-tables-i
$((\textit{load-addr} \in \mathbf{N})$
$\wedge\quad (\text{i-psw}\,(\text{i}\,(i,\ n,\ \textit{load-addr})) = \texttt{'run})$
$\wedge\quad (\text{i-word-size}\,(i) = \texttt{32}))$
$\rightarrow\quad (\text{i-link-tables}\,(\text{i}\,(i,\ n,\ \textit{load-addr}),\ \textit{load-addr})$
$\qquad = \quad \text{i-link-tables}\,(i,\ \textit{load-addr}))$

THEOREM: read-mem-ram-tree-is-get
$((\exp\,(\texttt{2},\ \textit{word-size}) \not< \text{length}\,(\textit{lst})) \wedge (n < \text{length}\,(\textit{lst})))$
$\rightarrow\quad (\text{read-mem}\,(\text{nat-to-v}\,(n,\ \textit{word-size}),\ \text{ram-tree}\,(\textit{lst},\ \exp\,(\texttt{2},\ \textit{word-size})))$
$\qquad = \quad \text{get}\,(n,\ \textit{lst}))$

THEOREM: one-way-correspondence-display-array
(bvp-listp (mem)
 $\land$ (exp (2, 32) $\not<$ length (mem))
 $\land$ (length (mem) $\not<$ (n + length ($type\text{-}lst$))))
$\rightarrow$ (display-fm9001-array ($type\text{-}lst$, n, ram-tree (mem, exp (2, 32)), $link\text{-}tables$)
 = display-array ($type\text{-}lst$, n, mem, $link\text{-}tables$))

DEFINITION:
plausible-data-link-tablep ($type\text{-}spec$, n, $usr\text{-}data\text{-}link\text{-}table$)
= **if** $type\text{-}spec \simeq$ **nil then t**
 else ($n \not<$ (cdr (assoc (caar ($type\text{-}spec$), $usr\text{-}data\text{-}link\text{-}table$))
 + length (cdar ($type\text{-}spec$))))
 $\land$ plausible-data-link-tablep (cdr ($type\text{-}spec$),
 n,
 $usr\text{-}data\text{-}link\text{-}table$) **endif**

THEOREM: one-way-correspondence-display-fm9001-data-segment1
(bvp-listp (mem)
 $\land$ (exp (2, 32) $\not<$ length (mem))
 $\land$ plausible-data-link-tablep ($type\text{-}spec$,
 length (mem),
 usr-data-links ($link\text{-}tables$)))
$\rightarrow$ (display-fm9001-data-segment1 ($type\text{-}spec$,
 ram-tree (mem, exp (2, 32)),
 $link\text{-}tables$)
 = display-data-segment ($type\text{-}spec$, mem, $link\text{-}tables$))

THEOREM: display-fm9001-data-segment-m->fm9001
(proper-m-statep (m)
 $\land$ plausible-data-link-tablep ($type\text{-}spec$,
 length (m-mem (m)),
 usr-data-links ($link\text{-}tables$)))
$\rightarrow$ (display-fm9001-data-segment (m->fm9001 (m), $type\text{-}spec$, $link\text{-}tables$)
 = display-m-data-segment (m, $type\text{-}spec$, $link\text{-}tables$))

```
; We will have to return to the question of a plausible data link table once we have
; finished the other levels.  The problem is that we have to show that the
; actual data link table generated is plausible and we will do that with
; arguments about the various levels preserving the signatures of the
; the various segments.  But these concepts are not available yet and so
; we complete our work at the m v. fm9001 level by just leaving plausibility
; for later.

; Book:  r.events.  Now we introduce the r machine.
```

EVENT: Set the status of all events. The status of each event is to be set as follows. Each 'boot-strap' satellite is to be returned to the initial condition. Each 'add-shell' satellite is to be enabled. enabled. Anything not otherwise mentioned is to be disabled. Name this event 'close-data-base-3'.

```
; We now build r-i.events from its subbooks.
```

DEFINITION:
strip-cadrs (*alist*)
= **if** *alist* $\simeq$ **nil then nil**
else cons (cadar (*alist*), strip-cadrs (cdr (*alist*))) **endif**

DEFINITION:
pcpp (x, *segment*)
= (listp (x)
$\wedge$ (adp-offset (x) $\in$ **N**)
$\wedge$ definedp (adp-name (x), *segment*)
$\wedge$ (adp-offset (x)
$<$ length (program-body (definition (adp-name (x), *segment*)))))

DEFINITION:
not-bool (x)
= **if** x = `'f` **then** `'t`
else `'f` **endif**

DEFINITION:
p-test-natp (*flg*, x)
= **if** *flg* = `'zero` **then** $x = 0$
else $x \neq 0$ **endif**

DEFINITION:
p-test-intp (*flg*, x)
= **case on** *flg*:
case = *zero*
then $x = 0$
case = *not-zero*
then $x \neq 0$
case = *neg*
then negativep (x)
case = *not-neg*
then $\neg$ negativep (x)
case = *pos*
then ($x \in$ **N**) $\wedge$ ($x \neq 0$)
otherwise ($x = 0$) $\vee$ negativep (x) **endcase**

Definition:
p-test-boolp (flg, x)
= **if** flg = `'t` **then** x = `'t`
else x = `'f` **endif**

Definition:
p-test-bitvp (flg, x)
= **if** flg = `'all-zero` **then** all-zero-bitvp (x)
else ¬ all-zero-bitvp (x) **endif**

Definition:
PITON-OPCODES

```
=  '(call ret locn push-constant push-local push-global
     push-ctrl-stk-free-size push-temp-stk-free-size
     push-temp-stk-index jump-if-temp-stk-full
     jump-if-temp-stk-empty pop pop* popn pop-local
     pop-global pop-locn pop-call fetch-temp-stk
     deposit-temp-stk jump jump-case pushj popj set-local
     set-global test-nat-and-jump test-int-and-jump
     test-bool-and-jump test-bitv-and-jump no-op add-addr
     sub-addr eq lt-addr fetch deposit add-int
     add-int-with-carry add1-int sub-int sub-int-with-carry
     sub1-int neg-int lt-int int-to-nat add-nat
     add-nat-with-carry add1-nat sub-nat sub-nat-with-carry
     sub1-nat lt-nat mult2-nat mult2-nat-with-carry-out
     div2-nat or-bitv and-bitv not-bitv xor-bitv rsh-bitv
     lsh-bitv or-bool and-bool not-bool)
```

Definition:
legal-labelp (ins) = (labelledp (ins) → litatom (cadr (ins)))

Definition:
r-objectp (x, r)
= (listp (x)
∧ (cddr (x) = **nil**)
∧ **case on** type (x):
case = nat
then small-naturalp (untag (x), r-word-size (r))
case = int
then small-integerp (untag (x), r-word-size (r))
case = $bitv$
then bit-vectorp (untag (x), r-word-size (r))
case = $bool$
then booleanp (untag (x))
case = $addr$

then adpp (untag (x), r-usr-data-segment (r))
case = *pc*
then pcpp (untag (x), r-prog-segment (r))
case = *subr*
then definedp (untag (x), r-prog-segment (r))
case = *sys-addr*
then adpp (untag (x), r-sys-data-segment (r))
otherwise **f** **endcase**)

DEFINITION:
r-objectp-type (*type*, x, r) = ((type (x) = *type*) $\wedge$ r-objectp (x, r))

DEFINITION:
all-r-objectps (*lst*, r)
= **if** *lst* $\simeq$ **nil** **then** *lst* = **nil**
else r-objectp (car (*lst*), r) $\wedge$ all-r-objectps (cdr (*lst*), r) **endif**

DEFINITION:
r-usr-data-typep (*type*)
= (*type* $\in$ `'(nat int bitv bool addr subr pc)`)

DEFINITION:
r-usr-data-objectp (x, r) = (r-objectp (x, r) $\wedge$ r-usr-data-typep (type (x)))

DEFINITION: push-stk-n (n, *sys-addr*) = sub-addr (*sys-addr*, n)

DEFINITION: pop-stk-n (n, *sys-addr*) = add-addr (*sys-addr*, n)

DEFINITION:
fetch-n (n, *addr*, *segment*)
= **if** $n \simeq$ `0` **then** **nil**
else cons (fetch (*addr*, *segment*),
fetch-n ($n - 1$, pop-stk (*addr*), *segment*)) **endif**

DEFINITION:
deposit-lst (*lst*, *addr*, *segment*)
= **if** *lst* $\simeq$ **nil** **then** *segment*
else deposit-lst (cdr (*lst*),
push-stk (*addr*),
deposit (car (*lst*), *addr*, *segment*)) **endif**

DEFINITION:
total-r-system-size (r, *load-addr*)
= (*load-addr*
+ segment-length (r-usr-data-segment (r))
+ segment-length (icompile (r-prog-segment (r)))
+ segment-length (r-sys-data-segment (r)))

DEFINITION:
r-loadablep $(r,\ load\text{-}addr)$
$=\quad (\text{total-r-system-size}\,(r,\ load\text{-}addr)\ <\ \exp\,(2,\ \text{r-word-size}\,(r)))$

DEFINITION:
r-instructionp (ins)
$=\quad ((\text{car}\,(ins) \in \text{PITON-OPCODES})$
$\quad\quad \wedge \quad ((\text{car}\,(ins) = \texttt{'push-constant})$
$\quad\quad\quad\quad \rightarrow \quad (\neg\ \text{labelledp}\,(\text{cadr}\,(ins)))))$

DEFINITION:
r-program-bodyp (lst)
$=\quad$ **if** $lst \simeq$ **nil** **then** $lst =$ **nil**
else legal-labelp $(\text{car}\,(lst))$
$\quad\quad \wedge \quad \text{r-instructionp}\,(\text{unlabel}\,(\text{car}\,(lst)))$
$\quad\quad \wedge \quad \text{r-program-bodyp}\,(\text{cdr}\,(lst))$ **endif**

DEFINITION:
r-temp-var-dclsp $(temp\text{-}var\text{-}dcls)$
$=\quad$ **if** listp $(temp\text{-}var\text{-}dcls)$
then $(\neg\ \text{labelledp}\,(\text{cadar}\,(temp\text{-}var\text{-}dcls)))$
$\quad\quad \wedge \quad \text{r-temp-var-dclsp}\,(\text{cdr}\,(temp\text{-}var\text{-}dcls))$
else t endif

DEFINITION:
r-programp $(prog)$
$=\quad (\text{litatom}\,(\text{name}\,(prog))$
$\quad\quad \wedge \quad \text{r-temp-var-dclsp}\,(\text{temp-var-dcls}\,(prog))$
$\quad\quad \wedge \quad \text{listp}\,(\text{program-body}\,(prog))$
$\quad\quad \wedge \quad \text{r-program-bodyp}\,(\text{program-body}\,(prog)))$

DEFINITION:
r-prog-segmentp $(segment)$
$=\quad$ **if** $segment \simeq$ **nil** **then** $segment =$ **nil**
else r-programp $(\text{car}\,(segment))$
$\quad\quad \wedge \quad \text{r-prog-segmentp}\,(\text{cdr}\,(segment))$ **endif**

DEFINITION:
all-r-usr-data-objectps $(lst,\ r)$
$=\quad$ **if** $lst \simeq$ **nil** **then** $lst =$ **nil**
else r-usr-data-objectp $(\text{car}\,(lst),\ r)$
$\quad\quad \wedge \quad \text{all-r-usr-data-objectps}\,(\text{cdr}\,(lst),\ r)$ **endif**

DEFINITION:
proper-r-usr-data-area $(area,\ r)$

$=$ (litatom (car (*area*))
    $\wedge$ listp (cdr (*area*))
    $\wedge$ all-r-usr-data-objectps (cdr (*area*), r))

DEFINITION:
proper-r-usr-data-segmentp (*data-segment*, r)
$=$ **if** *data-segment* $\simeq$ **nil** **then** *data-segment* $=$ **nil**
  else proper-r-usr-data-area (car (*data-segment*), r)
    $\wedge$ ($\neg$ definedp (caar (*data-segment*), cdr (*data-segment*)))
    $\wedge$ proper-r-usr-data-segmentp (cdr (*data-segment*), r) **endif**

DEFINITION:
proper-r-statep (r, *load-addr*)
$=$ (r-statep (r)
    $\wedge$ r-objectp-type (`'pc`, r-pc (r), r)
    $\wedge$ cstkp (r-cfp (r), r-sys-data-segment (r))
    $\wedge$ cstkp (r-csp (r), r-sys-data-segment (r))
    $\wedge$ tstkp (r-tsp (r), r-sys-data-segment (r))
    $\wedge$ (definition (`'full-ctrl-stk-addr`, r-sys-data-segment (r))
      $=$ list (`'full-ctrl-stk-addr`,
          tag (`'sys-addr`, `'(cstk . 0)`)))
    $\wedge$ (definition (`'full-temp-stk-addr`, r-sys-data-segment (r))
      $=$ list (`'full-temp-stk-addr`,
          tag (`'sys-addr`, `'(tstk . 0)`)))
    $\wedge$ (definition (`'empty-temp-stk-addr`, r-sys-data-segment (r))
      $=$ list (`'empty-temp-stk-addr`,
          tag (`'sys-addr`,
            cons (`'tstk`,
              length (definiens (`'tstk`,
                  r-sys-data-segment (r))) $-$ 1))))
    $\wedge$ r-objectp-type (`'bool`, r-c-flg (r), r)
    $\wedge$ r-objectp-type (`'bool`, r-v-flg (r), r)
    $\wedge$ r-objectp-type (`'bool`, r-n-flg (r), r)
    $\wedge$ r-objectp-type (`'bool`, r-z-flg (r), r)
    $\wedge$ r-prog-segmentp (r-prog-segment (r))
    $\wedge$ proper-r-usr-data-segmentp (r-usr-data-segment (r), r)
    $\wedge$ r-loadablep (r, *load-addr*))

EVENT: Enable pathological-difference.

DEFINITION:
stk-equal (*sp*, *segment1*, *segment2*)
$=$ **if** empty-stkp (*sp*, *segment1*) **then t**
  else (fetch (*sp*, *segment1*) $=$ fetch (*sp*, *segment2*))
    $\wedge$ stk-equal (pop-stk (*sp*), *segment1*, *segment2*) **endif**

DEFINITION: signature (*area*) = cons (car (*area*), length (*area*))

DEFINITION:
same-signature (*segment1*, *segment2*)
= **if** listp (*segment1*)
then listp (*segment2*)
∧ (signature (car (*segment1*)) = signature (car (*segment2*)))
∧ same-signature (cdr (*segment1*), cdr (*segment2*))
else *segment2* ≃ **nil endif**

DEFINITION:
r-equal (*r1*, *r2*)
= (r-statep (*r1*)
∧ r-statep (*r2*)
∧ (r-pc (*r1*) = r-pc (*r2*))
∧ (r-cfp (*r1*) = r-cfp (*r2*))
∧ (r-csp (*r1*) = r-csp (*r2*))
∧ (r-tsp (*r1*) = r-tsp (*r2*))
∧ (r-prog-segment (*r1*) = r-prog-segment (*r2*))
∧ (r-usr-data-segment (*r1*) = r-usr-data-segment (*r2*))
∧ stk-equal (r-csp (*r1*),
r-sys-data-segment (*r1*),
r-sys-data-segment (*r2*))
∧ stk-equal (r-tsp (*r1*),
r-sys-data-segment (*r1*),
r-sys-data-segment (*r2*))
∧ same-signature (r-sys-data-segment (*r1*), r-sys-data-segment (*r2*))
∧ (r-word-size (*r1*) = r-word-size (*r2*))
∧ (r-psw (*r1*) = r-psw (*r2*)))

DEFINITION: add1-r-pc (*r*) = add1-addr (r-pc (*r*))

DEFINITION:
add1-r-pcp (*r*) = pcpp (untag (add1-addr (r-pc (*r*))), r-prog-segment (*r*))

DEFINITION:
r-current-program (*r*) = definition (area-name (r-pc (*r*)), r-prog-segment (*r*))

DEFINITION:
r-current-instruction (*r*)
= unlabel (get (offset (r-pc (*r*)), program-body (r-current-program (*r*))))

DEFINITION:
r-halt (*r*, *psw*)
= r-state (r-pc (*r*),

r-cfp (*r*),
r-csp (*r*),
r-tsp (*r*),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
r-sys-data-segment (*r*),
r-word-size (*r*),
psw)

DEFINITION:
r-call-okp (*ins*, *r*)
= (definedp (cadr (*ins*), r-prog-segment (*r*))
∧ (free-stk-size (r-csp (*r*))
≮ (1 + (1 + length (local-vars (definition (cadr (*ins*),
r-prog-segment (*r*)))))))
∧ (stk-length (r-tsp (*r*), r-sys-data-segment (*r*))
≮ length (formal-vars (definition (cadr (*ins*),
r-prog-segment (*r*)))))
∧ add1-r-pcp (*r*)
∧ all-r-objectps (strip-cadrs (temp-var-dcls (definition (cadr (*ins*),
r-prog-segment (*r*)))),
r))

DEFINITION:
r-call-step (*ins*, *r*)
= r-state (tag (`'pc`, cons (cadr (*ins*), `0`)),
push-stk-n (`2`, r-csp (*r*)),
push-stk-n (`2`
+ length (formal-vars (definition (cadr (*ins*),
r-prog-segment (*r*))))
+ length (temp-var-dcls (definition (cadr (*ins*),
r-prog-segment (*r*)))),
r-csp (*r*)),
pop-stk-n (length (formal-vars (definition (cadr (*ins*),
r-prog-segment (*r*)))),
r-tsp (*r*)),
r-x (*r*),
r-y (*r*),

```
          r-c-flg (r),
          r-v-flg (r),
          r-n-flg (r),
          r-z-flg (r),
          r-prog-segment (r),
          r-usr-data-segment (r),
          deposit-lst (append (list (add1-addr (r-pc (r)), r-cfp (r)),
                           append (rev (strip-cadrs (temp-var-dcls (definition (cadr (ins),
                                                                      r-prog-segment (r))))),
                                fetch-n (length (formal-vars (definition (cadr (ins),
                                                                       r-prog-segment (r)))),
                                        r-tsp (r),
                                        r-sys-data-segment (r)))),
                      push-stk (r-csp (r)),
                      r-sys-data-segment (r)),
          r-word-size (r),
          'run)
```

DEFINITION:

```
r-call-step-clock (ins, r)
=   (2
     +  2
     +  length (temp-var-dcls (definition (cadr (ins), r-prog-segment (r))))
     +  length (formal-vars (definition (cadr (ins), r-prog-segment (r)))))
```

DEFINITION:

```
r-ret-okp (ins, r)
=   ((¬ empty-stkp (r-cfp (r), r-sys-data-segment (r)))
     ∧  (¬ empty-stkp (pop-stk (r-cfp (r)), r-sys-data-segment (r)))
     ∧  r-objectp-type ('pc,
                        fetch (pop-stk (r-cfp (r)), r-sys-data-segment (r)),
                        r)
     ∧  (area-name (r-cfp (r)) = 'cstk)
     ∧  (area-name (r-csp (r)) = 'cstk)
     ∧  (offset (r-cfp (r)) ≮ offset (r-csp (r)))
     ∧  cstkp (fetch (r-cfp (r), r-sys-data-segment (r)),
               r-sys-data-segment (r)))
```

DEFINITION:

```
r-ret-step (ins, r)
=   if empty-stkp (pop-stk (pop-stk (r-cfp (r))), r-sys-data-segment (r))
    then r-halt (r, 'halt)
    else r-state (fetch (pop-stk (r-cfp (r)), r-sys-data-segment (r)),
                  fetch (r-cfp (r), r-sys-data-segment (r)),
                  pop-stk (pop-stk (r-cfp (r))),
```

```
                      r-tsp (r),
                      r-x (r),
                      r-y (r),
                      r-c-flg (r),
                      r-v-flg (r),
                      r-n-flg (r),
                      r-z-flg (r),
                      r-prog-segment (r),
                      r-usr-data-segment (r),
                      r-sys-data-segment (r),
                      r-word-size (r),
                      'run) endif
```

DEFINITION:

```
r-ret-step-clock (ins, r)
=  if empty-stkp (pop-stk (pop-stk (r-cfp (r))), r-sys-data-segment (r))
   then 0
   else 1 + 3 endif
```

DEFINITION:

```
r-locn-okp (ins, r)
=  (add1-r-pcp (r)
    ∧  r-objectp-type ('sys-addr,
                       add-addr (r-csp (r),
                                 offset-from-csp (cadr (ins),
                                                  r-current-program (r))),
                       r)
    ∧  r-objectp-type ('nat,
                       fetch (add-addr (r-csp (r),
                                        offset-from-csp (cadr (ins),
                                                         r-current-program (r))),
                              r-sys-data-segment (r)),
                       r)
    ∧  r-objectp-type ('sys-addr,
                       add-addr (r-csp (r),
                                 untag (fetch (add-addr (r-csp (r),
                                                         offset-from-csp (cadr (ins),
                                                                          r-current-program (r))),
                                               r-sys-data-segment (r)))),
                       r)
    ∧  (¬ empty-stkp (add-addr (r-csp (r),
                                 offset-from-csp (cadr (ins),
                                                  r-current-program (r))),
                       r-sys-data-segment (r)))
```

```
    ∧  (¬ empty-stkp (add-addr (r-csp (r),
                                untag (fetch (add-addr (r-csp (r),
                                                        offset-from-csp (cadr (ins),
                                                                         r-current-program (r))),
                                              r-sys-data-segment (r)))),
                      r-sys-data-segment (r)))
    ∧  (free-stk-size (r-tsp (r)) ≄ 0))
```

Definition:

```
r-locn-step (ins, r)
=  r-state (add1-r-pc (r),
            r-cfp (r),
            r-csp (r),
            push-stk (r-tsp (r)),
            add-addr (r-csp (r),
                      untag (fetch (add-addr (r-csp (r),
                                              offset-from-csp (cadr (ins),
                                                               r-current-program (r))),
                                    r-sys-data-segment (r)))),
            r-y (r),
            r-c-flg (r),
            r-v-flg (r),
            r-n-flg (r),
            r-z-flg (r),
            r-prog-segment (r),
            r-usr-data-segment (r),
            deposit (fetch (add-addr (r-csp (r),
                                      untag (fetch (add-addr (r-csp (r),
                                                              offset-from-csp (cadr (ins),
                                                                               r-current-program (r))),
                                                    r-sys-data-segment (r)))),
                            r-sys-data-segment (r)),
                     push-stk (r-tsp (r)),
                     r-sys-data-segment (r)),
            r-word-size (r),
            'run)
```

Definition: r-locn-step-clock (*ins*, *r*) = 5

Definition:

```
r-push-constant-okp (ins, r)
=  (add1-r-pcp (r)
    ∧  (r-objectp (cadr (ins), r)
        ∨  (cadr (ins) = 'pc)
        ∨  ((cadr (ins) ≃ nil)
```

$\wedge$ find-labelp (cadr (*ins*),
program-body (r-current-program (*r*)))))
$\wedge$ (free-stk-size (r-tsp (*r*)) $\not\simeq$ `0`))

DEFINITION:
r-push-constant-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
r-cfp (*r*),
r-csp (*r*),
push-stk (r-tsp (*r*)),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
deposit (**if** cadr (*ins*) = **'pc** **then** add1-r-pc (*r*)
elseif cadr (*ins*) $\simeq$ **nil**
then pc (cadr (*ins*), r-current-program (*r*))
else cadr (*ins*) **endif**,
push-stk (r-tsp (*r*)),
r-sys-data-segment (*r*)),
r-word-size (*r*),
'run)

DEFINITION: r-push-constant-step-clock (*ins*, *r*) = `1`

DEFINITION:
r-push-local-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
$\wedge$ r-objectp-type (**'sys-addr**,
add-addr (r-csp (*r*),
offset-from-csp (cadr (*ins*),
r-current-program (*r*))),
r)
$\wedge$ ($\neg$ empty-stkp (add-addr (r-csp (*r*),
offset-from-csp (cadr (*ins*),
r-current-program (*r*))),
r-sys-data-segment (*r*)))
$\wedge$ (free-stk-size (r-tsp (*r*)) $\not\simeq$ `0`))

DEFINITION:
r-push-local-step (*ins*, *r*)

$=$ r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
push-stk (r-tsp (r)),
add-addr (r-csp (r),
offset-from-csp (cadr (ins), r-current-program (r))),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (fetch (add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r-sys-data-segment (r)),
push-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-push-local-step-clock (ins, r) $=$ `3`

DEFINITION:
r-push-global-okp (ins, r)
$=$ (add1-r-pcp (r)
$\wedge$ definedp (cadr (ins), r-usr-data-segment (r))
$\wedge$ listp (definiens (cadr (ins), r-usr-data-segment (r)))
$\wedge$ (free-stk-size (r-tsp (r)) $\not\simeq$ `0`))

DEFINITION:
r-push-global-step (ins, r)
$=$ r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
push-stk (r-tsp (r)),
tag (**'addr**, cons (cadr (ins), `0`)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),

deposit (fetch (tag (`'addr`, cons (cadr (ins), `0`)),
r-usr-data-segment (r)),
push-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
`'run`)

DEFINITION: r-push-global-step-clock (ins, r) = `2`

DEFINITION:
r-push-ctrl-stk-free-size-okp (ins, r)
= (add1-r-pcp (r) $\wedge$ (free-stk-size (r-tsp (r)) $\not\simeq$ `0`))

DEFINITION:
r-push-ctrl-stk-free-size-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
push-stk (r-tsp (r)),
fetch (`'(sys-addr (full-ctrl-stk-addr . 0))`,
r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (`'nat`, free-stk-size (r-csp (r))),
push-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
`'run`)

DEFINITION: r-push-ctrl-stk-free-size-step-clock (ins, r) = `4`

DEFINITION:
r-push-temp-stk-free-size-okp (ins, r)
= (add1-r-pcp (r) $\wedge$ (free-stk-size (r-tsp (r)) $\not\simeq$ `0`))

DEFINITION:
r-push-temp-stk-free-size-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),

push-stk (r-tsp (r)),
fetch (`'(sys-addr (full-temp-stk-addr . 0))`,
r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (`'nat`, free-stk-size (r-tsp (r))),
push-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
`'run`)

DEFINITION: r-push-temp-stk-free-size-step-clock (ins, r) = `4`

DEFINITION:
r-push-temp-stk-index-okp (ins, r)
= (add1-r-pcp (r)
∧ (free-stk-size (r-tsp (r)) $\not\simeq$ `0`)
∧ (cadr (ins) $\in \mathbf{N}$)
∧ (cadr (ins) $<$ stk-length (r-tsp (r), r-sys-data-segment (r))))

DEFINITION:
r-push-temp-stk-index-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
push-stk (r-tsp (r)),
tag (`'nat`, 1 + cadr (ins)),
r-tsp (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
`'(bool f)`,
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (`'nat`,
(stk-length (r-tsp (r), r-sys-data-segment (r)) $-$ 1)
$-$ cadr (ins)),
push-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),

'run)

DEFINITION: r-push-temp-stk-index-step-clock (*ins*, *r*) = 7

DEFINITION:
r-jump-if-temp-stk-full-okp (*ins*, *r*)
= **if** free-stk-size (r-tsp (*r*)) ≃ 0
 then find-labelp (cadr (*ins*), program-body (r-current-program (*r*)))
 else add1-r-pcp (*r*) **endif**

DEFINITION:
r-jump-if-temp-stk-full-step (*ins*, *r*)
= r-state (**if** free-stk-size (r-tsp (*r*)) ≃ 0
 then pc (cadr (*ins*), r-current-program (*r*))
 else add1-r-pc (*r*) **endif**,
 r-cfp (*r*),
 r-csp (*r*),
 r-tsp (*r*),
 pc (cadr (*ins*), r-current-program (*r*)),
 '(sys-addr (tstk . 0)),
 r-c-flg (*r*),
 r-v-flg (*r*),
 r-n-flg (*r*),
 bool (free-stk-size (r-tsp (*r*)) ≃ 0),
 r-prog-segment (*r*),
 r-usr-data-segment (*r*),
 r-sys-data-segment (*r*),
 r-word-size (*r*),
 'run)

DEFINITION: r-jump-if-temp-stk-full-step-clock (*ins*, *r*) = 6

DEFINITION:
r-jump-if-temp-stk-empty-okp (*ins*, *r*)
= **if** empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*))
 then find-labelp (cadr (*ins*), program-body (r-current-program (*r*)))
 else add1-r-pcp (*r*) **endif**

DEFINITION:
r-jump-if-temp-stk-empty-step (*ins*, *r*)
= r-state (**if** empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*))
 then pc (cadr (*ins*), r-current-program (*r*))
 else add1-r-pc (*r*) **endif**,
 r-cfp (*r*),
 r-csp (*r*),

r-tsp (r),
pc (cadr (ins), r-current-program (r)),
r-tsp (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
bool (empty-stkp (r-tsp (r), r-sys-data-segment (r))),
r-prog-segment (r),
r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-jump-if-temp-stk-empty-step-clock $(ins,\ r) = 6$

DEFINITION:
r-pop-okp $(ins,\ r)$
= (add1-r-pcp (r)
∧ r-objectp (fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r))))

DEFINITION:
r-pop-step $(ins,\ r)$
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-pop-step-clock $(ins,\ r) = 1$

DEFINITION:
r-pop*-okp $(ins,\ r)$
= (add1-r-pcp (r)
∧ (cadr $(ins) \in \mathbf{N}$)
∧ r-objectp-type (**'sys-addr**, pop-stk-n (cadr (ins), r-tsp (r)), r))

DEFINITION:
r-pop*-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk-n (cadr (ins), r-tsp (r)),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-pop*-step-clock (ins, r) = 1

DEFINITION:
r-popn-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ r-objectp-type (**'sys-addr**,
pop-stk-n (1 + untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-tsp (r)),
r))

DEFINITION:
r-popn-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk-n (1 + untag (fetch (r-tsp (r), r-sys-data-segment (r))),
r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),

r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-popn-step-clock (ins, r) = 2

DEFINITION:
r-pop-local-okp (ins, r)
= (add1-r-pcp (r)
∧ r-objectp-type (**'sys-addr**,
add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r)
∧ (¬ empty-stkp (add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r-sys-data-segment (r)))
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r))))

DEFINITION:
r-pop-local-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
add-addr (r-csp (r),
offset-from-csp (cadr (ins), r-current-program (r))),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (fetch (r-tsp (r), r-sys-data-segment (r)),
add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-pop-local-step-clock (ins, r) = 3

DEFINITION:
r-pop-global-okp (ins, r)
= (add1-r-pcp (r)
  ∧ definedp (cadr (ins), r-usr-data-segment (r))
  ∧ listp (definiens (cadr (ins), r-usr-data-segment (r)))
  ∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
  ∧ r-usr-data-objectp (fetch (r-tsp (r), r-sys-data-segment (r)), r))

DEFINITION:
r-pop-global-step (ins, r)
= r-state (add1-r-pc (r),
    r-cfp (r),
    r-csp (r),
    pop-stk (r-tsp (r)),
    tag (**'addr**, cons (cadr (ins), 0)),
    r-y (r),
    r-c-flg (r),
    r-v-flg (r),
    r-n-flg (r),
    r-z-flg (r),
    r-prog-segment (r),
    deposit (fetch (r-tsp (r), r-sys-data-segment (r)),
      tag (**'addr**, cons (cadr (ins), 0)),
      r-usr-data-segment (r)),
    r-sys-data-segment (r),
    r-word-size (r),
    'run)

DEFINITION: r-pop-global-step-clock (ins, r) = 2

DEFINITION:
r-pop-locn-okp (ins, r)
= (add1-r-pcp (r)
  ∧ r-objectp-type (**'sys-addr**,
      add-addr (r-csp (r),
        offset-from-csp (cadr (ins),
          r-current-program (r))),
      r)
  ∧ r-objectp-type (**'nat**,
      fetch (add-addr (r-csp (r),
          offset-from-csp (cadr (ins),
            r-current-program (r))),
        r-sys-data-segment (r)),
      r)
  ∧ r-objectp-type (**'sys-addr**,

```
                    add-addr (r-csp (r),
                              untag (fetch (add-addr (r-csp (r),
                                                      offset-from-csp (cadr (ins),
                                                                       r-current-program (r))),
                                            r-sys-data-segment (r)))),
                    r)
    ∧  (¬ empty-stkp (add-addr (r-csp (r),
                                offset-from-csp (cadr (ins),
                                                 r-current-program (r))),
                      r-sys-data-segment (r)))
    ∧  (¬ empty-stkp (add-addr (r-csp (r),
                                untag (fetch (add-addr (r-csp (r),
                                                        offset-from-csp (cadr (ins),
                                                                         r-current-program (r))),
                                              r-sys-data-segment (r)))),
                      r-sys-data-segment (r)))
    ∧  (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r))))
```

DEFINITION:

```
r-pop-locn-step (ins, r)
=  r-state (add1-r-pc (r),
            r-cfp (r),
            r-csp (r),
            pop-stk (r-tsp (r)),
            add-addr (r-csp (r),
                      untag (fetch (add-addr (r-csp (r),
                                              offset-from-csp (cadr (ins),
                                                               r-current-program (r))),
                                    r-sys-data-segment (r)))),
            r-y (r),
            r-c-flg (r),
            r-v-flg (r),
            r-n-flg (r),
            r-z-flg (r),
            r-prog-segment (r),
            r-usr-data-segment (r),
            deposit (fetch (r-tsp (r), r-sys-data-segment (r)),
                     add-addr (r-csp (r),
                               untag (fetch (add-addr (r-csp (r),
                                                       offset-from-csp (cadr (ins),
                                                                        r-current-program (r))),
                                             r-sys-data-segment (r)))),
                     r-sys-data-segment (r)),
            r-word-size (r),
```

'**run**)

DEFINITION: r-pop-locn-step-clock $(ins,\ r) = 5$

DEFINITION:
r-pop-call-okp $(ins,\ r)$
= ((¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
  ∧ r-objectp-type ('**subr**,
      fetch (r-tsp (r), r-sys-data-segment (r)),
      r)
  ∧ r-call-okp (list ('**call**,
      untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
    r-state (r-pc (r),
      r-cfp (r),
      r-csp (r),
      pop-stk (r-tsp (r)),
      fetch (r-tsp (r), r-sys-data-segment (r)),
      r-y (r),
      r-c-flg (r),
      r-v-flg (r),
      r-n-flg (r),
      r-z-flg (r),
      r-prog-segment (r),
      r-usr-data-segment (r),
      r-sys-data-segment (r),
      r-word-size (r),
      '**run**)))

DEFINITION:
r-pop-call-step $(ins,\ r)$
= r-call-step (list ('**call**, untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
    r-state (r-pc (r),
      r-cfp (r),
      r-csp (r),
      pop-stk (r-tsp (r)),
      fetch (r-tsp (r), r-sys-data-segment (r)),
      r-y (r),
      r-c-flg (r),
      r-v-flg (r),
      r-n-flg (r),
      r-z-flg (r),
      r-prog-segment (r),
      r-usr-data-segment (r),
      r-sys-data-segment (r),
      r-word-size (r),

'run))

DEFINITION:
r-pop-call-step-clock (*ins*, *r*)
= (3
 + 2
 + length (temp-var-dcls (definition (untag (fetch (r-tsp (*r*),
 r-sys-data-segment (*r*))),
 r-prog-segment (*r*))))
 + length (formal-vars (definition (untag (fetch (r-tsp (*r*),
 r-sys-data-segment (*r*))),
 r-prog-segment (*r*)))))

DEFINITION:
r-fetch-temp-stk-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
 ∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
 ∧ r-objectp-type ('**nat**, fetch (r-tsp (*r*), r-sys-data-segment (*r*)), *r*)
 ∧ (untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))
 < stk-length (r-tsp (*r*), r-sys-data-segment (*r*))))

DEFINITION:
r-fetch-temp-stk-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
 r-cfp (*r*),
 r-csp (*r*),
 r-tsp (*r*),
 sub-addr (tag ('**sys-addr**,
 cons ('**tstk**,
 length (definiens ('**tstk**,
 r-sys-data-segment (*r*))) − 1)),
 1 + untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))),
 tag ('**nat**, 1 + untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))),
 r-c-flg (*r*),
 r-v-flg (*r*),
 r-n-flg (*r*),
 r-z-flg (*r*),
 r-prog-segment (*r*),
 r-usr-data-segment (*r*),
 deposit (fetch (sub-addr (tag ('**sys-addr**,
 cons ('**tstk**,
 length (definiens ('**tstk**,
 r-sys-data-segment (*r*))) − 1)),
 1 + untag (fetch (r-tsp (*r*),
 r-sys-data-segment (*r*)))),

r-sys-data-segment (r)),
r-tsp (r),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-fetch-temp-stk-step-clock (ins, r) = 6

DEFINITION:
r-deposit-temp-stk-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ (untag (fetch (r-tsp (r), r-sys-data-segment (r)))
< stk-length (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))
∧ r-usr-data-objectp (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r))

DEFINITION:
r-deposit-temp-stk-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (pop-stk (r-tsp (r))),
sub-addr (tag (**'sys-addr**,
cons (**'tstk**,
length (definiens (**'tstk**,
r-sys-data-segment (r))) − 1)),
1 + untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
tag (**'nat**, 1 + untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
sub-addr (tag (**'sys-addr**,
cons (**'tstk**,
length (definiens (**'tstk**,
r-sys-data-segment (r))) − 1)),
1 + untag (fetch (r-tsp (r),

r-sys-data-segment (*r*)))),
r-sys-data-segment (*r*)),
r-word-size (*r*),
`'run`)

DEFINITION: r-deposit-temp-stk-step-clock (*ins*, *r*) = `6`

DEFINITION:
r-jump-okp (*ins*, *r*)
= find-labelp (cadr (*ins*), program-body (r-current-program (*r*)))

DEFINITION:
r-jump-step (*ins*, *r*)
= r-state (pc (cadr (*ins*), r-current-program (*r*)),
r-cfp (*r*),
r-csp (*r*),
r-tsp (*r*),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
r-sys-data-segment (*r*),
r-word-size (*r*),
`'run`)

DEFINITION: r-jump-step-clock (*ins*, *r*) = `1`

DEFINITION:
r-jump-case-okp (*ins*, *r*)
= ((¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
∧ r-objectp-type (`'nat`, fetch (r-tsp (*r*), r-sys-data-segment (*r*)), *r*)
∧ (untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))
< length (cdr (*ins*)))
∧ find-labelp (get (untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*))),
cdr (*ins*)),
program-body (r-current-program (*r*))))

DEFINITION:
r-jump-case-step (*ins*, *r*)
= r-state (pc (get (untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*))), cdr (*ins*)),
r-current-program (*r*)),

r-cfp (*r*),
r-csp (*r*),
pop-stk (r-tsp (*r*)),
tag (**'nat**, 2 * untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
r-sys-data-segment (*r*),
r-word-size (*r*),
'run)

DEFINITION: r-jump-case-step-clock (*ins*, *r*) = 4

DEFINITION:
r-pushj-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
∧ find-labelp (cadr (*ins*), program-body (r-current-program (*r*)))
∧ (free-stk-size (r-tsp (*r*)) $\not\simeq$ 0))

DEFINITION:
r-pushj-step (*ins*, *r*)
= r-state (pc (cadr (*ins*), r-current-program (*r*)),
r-cfp (*r*),
r-csp (*r*),
push-stk (r-tsp (*r*)),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
deposit (add1-r-pc (*r*), push-stk (r-tsp (*r*)), r-sys-data-segment (*r*)),
r-word-size (*r*),
'run)

DEFINITION: r-pushj-step-clock (*ins*, *r*) = 2

DEFINITION:
r-popj-okp (*ins*, *r*)

= (r-objectp-type ('**pc**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r))))

DEFINITION:
r-popj-step (ins, r)
= r-state (fetch (r-tsp (r), r-sys-data-segment (r)),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
'**run**)

DEFINITION: r-popj-step-clock (ins, r) = 1

DEFINITION:
r-set-local-okp (ins, r)
= (add1-r-pcp (r)
∧ r-objectp-type ('**sys-addr**,
add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r)
∧ (¬ empty-stkp (add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r-sys-data-segment (r)))
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r))))

DEFINITION:
r-set-local-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
r-tsp (r),
add-addr (r-csp (r),
offset-from-csp (cadr (ins), r-current-program (r))),

r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (fetch (r-tsp (r), r-sys-data-segment (r)),
add-addr (r-csp (r),
offset-from-csp (cadr (ins),
r-current-program (r))),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-set-local-step-clock (ins, r) = 3

DEFINITION:
r-set-global-okp (ins, r)
= (add1-r-pcp (r)
∧ definedp (cadr (ins), r-usr-data-segment (r))
∧ listp (definiens (cadr (ins), r-usr-data-segment (r)))
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-usr-data-objectp (fetch (r-tsp (r), r-sys-data-segment (r)), r))

DEFINITION:
r-set-global-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
r-tsp (r),
tag (**'addr**, cons (cadr (ins), 0)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
deposit (fetch (r-tsp (r), r-sys-data-segment (r)),
tag (**'addr**, cons (cadr (ins), 0)),
r-usr-data-segment (r)),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-set-global-step-clock (*ins*, *r*) = 2

DEFINITION:
r-test-and-jump-okp (*ins*, *type*, *test*, *r*)
= ((¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
  ∧ r-objectp-type (*type*, fetch (r-tsp (*r*), r-sys-data-segment (*r*)), *r*)
  ∧ **if** *test*
    then find-labelp (caddr (*ins*),
                program-body (r-current-program (*r*)))
    else add1-r-pcp (*r*) **endif**)

DEFINITION:
r-test-and-jump-step (*test*, *lab*, *x*, *y*, *c*, *v*, *n*, *z*, *r*)
= **if** *test*
  then r-state (pc (*lab*, r-current-program (*r*)),
          r-cfp (*r*),
          r-csp (*r*),
          pop-stk (r-tsp (*r*)),
          x,
          y,
          c,
          v,
          n,
          z,
          r-prog-segment (*r*),
          r-usr-data-segment (*r*),
          r-sys-data-segment (*r*),
          r-word-size (*r*),
          'run)
  else r-state (add1-r-pc (*r*),
          r-cfp (*r*),
          r-csp (*r*),
          pop-stk (r-tsp (*r*)),
          x,
          y,
          c,
          v,
          n,
          z,
          r-prog-segment (*r*),
          r-usr-data-segment (*r*),
          r-sys-data-segment (*r*),
          r-word-size (*r*),
          'run) **endif**

DEFINITION:
```
r-test-nat-and-jump-okp (ins, r)
=    r-test-and-jump-okp (ins,
                          'nat,
                          p-test-natp (cadr (ins),
                                       untag (fetch (r-tsp (r),
                                                     r-sys-data-segment (r)))),
                          r)
```

DEFINITION:
```
r-test-nat-and-jump-step (ins, r)
=    r-test-and-jump-step (p-test-natp (cadr (ins),
                                        untag (fetch (r-tsp (r),
                                                      r-sys-data-segment (r)))),
                           caddr (ins),
                           pc (caddr (ins), r-current-program (r)),
                           fetch (r-tsp (r), r-sys-data-segment (r)),
                           r-c-flg (r),
                           r-v-flg (r),
                           r-n-flg (r),
                           bool (untag (fetch (r-tsp (r), r-sys-data-segment (r)))
                                 =    0),
                           r)
```

DEFINITION: r-test-nat-and-jump-step-clock (*ins*, *r*) = 3

DEFINITION:
```
r-test-int-and-jump-okp (ins, r)
=    r-test-and-jump-okp (ins,
                          'int,
                          p-test-intp (cadr (ins),
                                       untag (fetch (r-tsp (r),
                                                     r-sys-data-segment (r)))),
                          r)
```

DEFINITION:
```
r-test-int-and-jump-step (ins, r)
=    r-test-and-jump-step (p-test-intp (cadr (ins),
                                        untag (fetch (r-tsp (r),
                                                      r-sys-data-segment (r)))),
                           caddr (ins),
                           if cadr (ins) = 'pos then add1-addr (r-pc (r))
                           else pc (caddr (ins), r-current-program (r)) endif,
                           fetch (r-tsp (r), r-sys-data-segment (r)),
                           r-c-flg (r),
```

```
                          r-v-flg (r),
                          bool (negativep (untag (fetch (r-tsp (r),
                                                        r-sys-data-segment (r))))),
                          bool (untag (fetch (r-tsp (r), r-sys-data-segment (r)))
                               =   0),
                          r)
```

DEFINITION:

```
r-test-int-and-jump-step-clock (ins, r)
=   case on car (cdr (ins)):
    case = zero
    then 3
    case = not-zero
     then 3
    case = neg
     then 3
    case = not-neg
     then 3
    case = pos
     then 2 + if negativep (untag (fetch (r-tsp (r), r-sys-data-segment (r))))
              then 1
              elseif untag (fetch (r-tsp (r), r-sys-data-segment (r))) ≃ 0
              then 2
              else 3 endif
    otherwise if negativep (untag (fetch (r-tsp (r), r-sys-data-segment (r))))
              then 3
              else 4 endif endcase
```

DEFINITION:

```
r-test-bool-and-jump-okp (ins, r)
=   r-test-and-jump-okp (ins,
                         'bool,
                         p-test-boolp (cadr (ins),
                                       untag (fetch (r-tsp (r),
                                                     r-sys-data-segment (r)))),
                         r)
```

DEFINITION:

```
r-test-bool-and-jump-step (ins, r)
=   r-test-and-jump-step (p-test-boolp (cadr (ins),
                                        untag (fetch (r-tsp (r),
                                                      r-sys-data-segment (r)))),
                          caddr (ins),
                          pc (caddr (ins), r-current-program (r)),
                          fetch (r-tsp (r), r-sys-data-segment (r)),
```

```
                        r-c-flg (r),
                        r-v-flg (r),
                        r-n-flg (r),
                        tag ('bool,
                             not-bool (untag (fetch (r-tsp (r),
                                                     r-sys-data-segment (r))))),
                        r)
```

DEFINITION: r-test-bool-and-jump-step-clock (*ins*, *r*) = 3

DEFINITION:

```
r-test-bitv-and-jump-okp (ins, r)
=   r-test-and-jump-okp (ins,
                         'bitv,
                         p-test-bitvp (cadr (ins),
                                       untag (fetch (r-tsp (r),
                                                     r-sys-data-segment (r)))),
                         r)
```

DEFINITION:

```
r-test-bitv-and-jump-step (ins, r)
=   r-test-and-jump-step (p-test-bitvp (cadr (ins),
                                        untag (fetch (r-tsp (r),
                                                      r-sys-data-segment (r)))),
                          caddr (ins),
                          pc (caddr (ins), r-current-program (r)),
                          fetch (r-tsp (r), r-sys-data-segment (r)),
                          r-c-flg (r),
                          r-v-flg (r),
                          r-n-flg (r),
                          bool (all-zero-bitvp (untag (fetch (r-tsp (r),
                                                              r-sys-data-segment (r))))),
                          r)
```

DEFINITION: r-test-bitv-and-jump-step-clock (*ins*, *r*) = 3

DEFINITION: r-no-op-okp (*ins*, *r*) = add1-r-pcp (*r*)

DEFINITION:

```
r-no-op-step (ins, r)
=   r-state (add1-r-pc (r),
             r-cfp (r),
             r-csp (r),
             r-tsp (r),
             r-x (r),
```

r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-no-op-step-clock (*ins*, r) = 1

DEFINITION:
r-add-addr-okp (*ins*, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ r-objectp-type (**'addr**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
∧ r-objectp-type (**'addr**,
add-addr (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
untag (fetch (r-tsp (r),
r-sys-data-segment (r)))),
r))

DEFINITION:
r-add-addr-step (*ins*, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (add-addr (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)),

untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-add-addr-step-clock (ins, r) = 2

DEFINITION:
r-sub-addr-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ r-objectp-type (**'addr**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
∧ (offset (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
$\not<$ untag (fetch (r-tsp (r), r-sys-data-segment (r)))))

DEFINITION:
r-sub-addr-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (sub-addr (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-sub-addr-step-clock (ins, r) = 2

DEFINITION:

r-eq-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
  ∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
  ∧ (¬ empty-stkp (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
  ∧ r-usr-data-objectp (fetch (pop-stk (r-tsp (*r*)),
                                    r-sys-data-segment (*r*)),
                             r)
  ∧ r-usr-data-objectp (fetch (r-tsp (*r*), r-sys-data-segment (*r*)), *r*)
  ∧ (type (fetch (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
      = type (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))))

DEFINITION:
r-eq-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
    r-cfp (*r*),
    r-csp (*r*),
    pop-stk (r-tsp (*r*)),
    fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
    r-y (*r*),
    r-c-flg (*r*),
    r-v-flg (*r*),
    r-n-flg (*r*),
    bool (untag (fetch (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
        = untag (fetch (r-tsp (*r*), r-sys-data-segment (*r*)))),
    r-prog-segment (*r*),
    r-usr-data-segment (*r*),
    deposit (bool (untag (fetch (pop-stk (r-tsp (*r*)),
                                  r-sys-data-segment (*r*)))
                  = untag (fetch (r-tsp (*r*),
                                  r-sys-data-segment (*r*)))),
        pop-stk (r-tsp (*r*)),
        r-sys-data-segment (*r*)),
    r-word-size (*r*),
    'run)

DEFINITION: r-eq-step-clock (*ins*, *r*) = 4

DEFINITION:
r-lt-addr-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
  ∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
  ∧ (¬ empty-stkp (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
  ∧ r-objectp-type (**'addr**,
                    fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
                    r)

$\wedge$ r-objectp-type ('**addr**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
$\wedge$ (area-name (fetch (r-tsp (r), r-sys-data-segment (r)))
= area-name (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))))

Definition:
r-lt-addr-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
bool (offset (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
< offset (fetch (r-tsp (r), r-sys-data-segment (r)))),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (bool (offset (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))
< offset (fetch (r-tsp (r),
r-sys-data-segment (r)))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'**run**)

Definition: r-lt-addr-step-clock (ins, r) = `4`

Definition:
r-fetch-okp (ins, r)
= (add1-r-pcp (r)
$\wedge$ ($\neg$ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
$\wedge$ r-objectp-type ('**addr**,
fetch (r-tsp (r), r-sys-data-segment (r)),
r))

Definition:
r-fetch-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),

r-csp (r),
r-tsp (r),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (fetch (fetch (r-tsp (r), r-sys-data-segment (r)),
r-usr-data-segment (r)),
r-tsp (r),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-fetch-step-clock (ins, r) = 2

DEFINITION:
r-deposit-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ r-objectp-type (**'addr**,
fetch (r-tsp (r), r-sys-data-segment (r)),
r)
∧ r-usr-data-objectp (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r))

DEFINITION:
r-deposit-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (pop-stk (r-tsp (r))),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
deposit (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),

fetch (r-tsp (r), r-sys-data-segment (r)),
r-usr-data-segment (r)),
r-sys-data-segment (r),
r-word-size (r),
'run)

DEFINITION: r-deposit-step-clock (ins, r) = 2

DEFINITION:
r-add-int-okp (ins, r)
= (add1-r-pcp (r)
$\wedge$ ($\neg$ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
$\wedge$ ($\neg$ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
$\wedge$ r-objectp-type (**'int**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
$\wedge$ r-objectp-type (**'int**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
$\wedge$ small-integerp (iplus (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r))),
untag (fetch (r-tsp (r),
r-sys-data-segment (r)))),
r-word-size (r)))

DEFINITION:
r-add-int-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (**'int**,
iplus (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r))),
untag (fetch (r-tsp (r), r-sys-data-segment (r))))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-add-int-step-clock (ins, r) = 2

DEFINITION:

```
r-add-int-with-carry-okp (ins, r)
=   (add1-r-pcp (r)
      ∧  (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
      ∧  (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
      ∧  (¬ empty-stkp (pop-stk (pop-stk (r-tsp (r))),
                        r-sys-data-segment (r)))
      ∧  r-objectp-type ('int, fetch (r-tsp (r), r-sys-data-segment (r)), r)
      ∧  r-objectp-type ('int,
                        fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
                        r)
      ∧  r-objectp-type ('bool,
                        fetch (pop-stk (pop-stk (r-tsp (r))),
                               r-sys-data-segment (r)),
                        r))
```

DEFINITION:

```
r-add-int-with-carry-step (ins, r)
=   r-state (add1-r-pc (r),
             r-cfp (r),
             r-csp (r),
             pop-stk (r-tsp (r)),
             tag ('int,
                  fix-small-integer (iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                        r-sys-data-segment (r)))),
                                            iplus (untag (fetch (pop-stk (r-tsp (r)),
                                                                 r-sys-data-segment (r))),
                                                   untag (fetch (r-tsp (r),
                                                                 r-sys-data-segment (r))))),
                                     r-word-size (r))),
             fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
             fetch (pop-stk (pop-stk (r-tsp (r))), r-sys-data-segment (r)),
             bool (¬ small-integerp (iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                        r-sys-data-segment (r)))),
                                            iplus (untag (fetch (pop-stk (r-tsp (r)),
                                                                 r-sys-data-segment (r))),
                                                   untag (fetch (r-tsp (r),
                                                                 r-sys-data-segment (r))))),
                                     r-word-size (r))),
             r-n-flg (r),
             r-z-flg (r),
             r-prog-segment (r),
```

```
              r-usr-data-segment (r),
              deposit (tag ('int,
                          fix-small-integer (iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                               r-sys-data-segment (r)))),
                                                     iplus (untag (fetch (pop-stk (r-tsp (r)),
                                                                          r-sys-data-segment (r))),
                                                            untag (fetch (r-tsp (r),
                                                                          r-sys-data-segment (r))))),
                                              r-word-size (r))),
                   pop-stk (r-tsp (r)),
                   deposit (bool (¬ small-integerp (iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                                      r-sys-data-segment (r)))),
                                                            iplus (untag (fetch (pop-stk (r-tsp (r)),
                                                                                 r-sys-data-segment (r))),
                                                                   untag (fetch (r-tsp (r),
                                                                                 r-sys-data-segment (r))))),
                                                     r-word-size (r))),
                            pop-stk (pop-stk (r-tsp (r))),
                            r-sys-data-segment (r))),
              r-word-size (r),
              'run)
```

DEFINITION: r-add-int-with-carry-step-clock (ins, r) = 6

DEFINITION:

```
r-add1-int-okp (ins, r)
=   (add1-r-pcp (r)
      ∧   (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
      ∧   r-objectp-type ('int, fetch (r-tsp (r), r-sys-data-segment (r)), r)
      ∧   small-integerp (iplus (1,
                                  untag (fetch (r-tsp (r),
                                                r-sys-data-segment (r)))),
                          r-word-size (r)))
```

DEFINITION:

```
r-add1-int-step (ins, r)
=   r-state (add1-r-pc (r),
              r-cfp (r),
              r-csp (r),
              r-tsp (r),
              r-x (r),
              r-y (r),
              r-c-flg (r),
              r-v-flg (r),
              r-n-flg (r),
```

```
                r-z-flg (r),
                r-prog-segment (r),
                r-usr-data-segment (r),
                deposit (tag ('int,
                                 iplus (1,
                                        untag (fetch (r-tsp (r), r-sys-data-segment (r))))),
                           r-tsp (r),
                           r-sys-data-segment (r)),
                r-word-size (r),
                'run)
```

Definition: r-add1-int-step-clock (*ins*, *r*) = 1

Definition:

```
r-sub-int-okp (ins, r)
=    (add1-r-pcp (r)
      ∧  (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
      ∧  (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
      ∧  r-objectp-type ('int, fetch (r-tsp (r), r-sys-data-segment (r)), r)
      ∧  r-objectp-type ('int,
                          fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
                          r)
      ∧  small-integerp (idifference (untag (fetch (pop-stk (r-tsp (r)),
                                                     r-sys-data-segment (r))),
                                        untag (fetch (r-tsp (r),
                                                     r-sys-data-segment (r)))),
                          r-word-size (r)))
```

Definition:

```
r-sub-int-step (ins, r)
=    r-state (add1-r-pc (r),
                r-cfp (r),
                r-csp (r),
                pop-stk (r-tsp (r)),
                fetch (r-tsp (r), r-sys-data-segment (r)),
                r-y (r),
                r-c-flg (r),
                r-v-flg (r),
                r-n-flg (r),
                r-z-flg (r),
                r-prog-segment (r),
                r-usr-data-segment (r),
                deposit (tag ('int,
                                 idifference (untag (fetch (pop-stk (r-tsp (r)),
                                                           r-sys-data-segment (r))),
```

untag (fetch (r-tsp (r),
r-sys-data-segment (r))))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
`'run`)

DEFINITION: r-sub-int-step-clock (ins, r) = `2`

DEFINITION:
r-sub-int-with-carry-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))
∧ r-objectp-type (`'int`, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ r-objectp-type (`'int`,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
∧ r-objectp-type (`'bool`,
fetch (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)),
r))

DEFINITION:
r-sub-int-with-carry-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
tag (`'int`,
fix-small-integer (idifference (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r))),
iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))),
untag (fetch (r-tsp (r),
r-sys-data-segment (r))))),
r-word-size (r))),
fetch (r-tsp (r), r-sys-data-segment (r)),
fetch (pop-stk (pop-stk (r-tsp (r))), r-sys-data-segment (r)),
bool (¬ small-integerp (idifference (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r))),
iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))),

```
                                                untag (fetch (r-tsp (r),
                                                                 r-sys-data-segment (r))))),
                                    r-word-size (r))),
            r-n-flg (r),
            r-z-flg (r),
            r-prog-segment (r),
            r-usr-data-segment (r),
            deposit (tag ('int,
                          fix-small-integer (idifference (untag (fetch (pop-stk (r-tsp (r)),
                                                                        r-sys-data-segment (r))),
                                                          iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                                            r-sys-data-segment (r)))),
                                                                 untag (fetch (r-tsp (r),
                                                                               r-sys-data-segment (r))))),
                                             r-word-size (r))),
                     pop-stk (r-tsp (r)),
                     deposit (bool (¬ small-integerp (idifference (untag (fetch (pop-stk (r-tsp (r)),
                                                                                 r-sys-data-segment (r))),
                                                                  iplus (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                                                    r-sys-data-segment (r)))),
                                                                         untag (fetch (r-tsp (r),
                                                                                       r-sys-data-segment (r))))),
                                                     r-word-size (r))),
                              pop-stk (pop-stk (r-tsp (r))),
                              r-sys-data-segment (r))),
            r-word-size (r),
            'run)
```

DEFINITION: r-sub-int-with-carry-step-clock (*ins*, *r*) = 6

DEFINITION:

```
r-sub1-int-okp (ins, r)
=   (add1-r-pcp (r)
     ∧   (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
     ∧   r-objectp-type ('int, fetch (r-tsp (r), r-sys-data-segment (r)), r)
     ∧   small-integerp (idifference (untag (fetch (r-tsp (r),
                                                    r-sys-data-segment (r))),
                                     1),
                         r-word-size (r)))
```

DEFINITION:

```
r-sub1-int-step (ins, r)
=   r-state (add1-r-pc (r),
            r-cfp (r),
            r-csp (r),
```

r-tsp (r),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (`'int`,
idifference (untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
`1`)),
r-tsp (r),
r-sys-data-segment (r)),
r-word-size (r),
`'run`)

DEFINITION: r-sub1-int-step-clock $(ins,\ r) = 1$

DEFINITION:
r-neg-int-okp $(ins,\ r)$
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-objectp-type (`'int`, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ small-integerp (inegate (untag (fetch (r-tsp (r),
r-sys-data-segment (r)))),
r-word-size (r)))

DEFINITION:
r-neg-int-step $(ins,\ r)$
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
r-tsp (r),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (`'int`,
inegate (untag (fetch (r-tsp (r),

r-sys-data-segment (*r*))))),
r-tsp (*r*),
r-sys-data-segment (*r*)),
r-word-size (*r*),
`'run`)

DEFINITION: r-neg-int-step-clock (*ins*, *r*) = `1`

DEFINITION:
r-lt-int-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
∧ (¬ empty-stkp (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
∧ r-objectp-type (`'int`, fetch (r-tsp (*r*), r-sys-data-segment (*r*)), *r*)
∧ r-objectp-type (`'int`,
fetch (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)),
r))

DEFINITION:
r-lt-int-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
r-cfp (*r*),
r-csp (*r*),
pop-stk (r-tsp (*r*)),
bool (negativep (fix-small-integer (idifference (untag (fetch (pop-stk (r-tsp (*r*)),
r-sys-data-segment (*r*))),
untag (fetch (r-tsp (*r*),
r-sys-data-segment (*r*)))),
r-word-size (*r*)))),
r-y (*r*),
r-c-flg (*r*),
bool (¬ small-integerp (idifference (untag (fetch (pop-stk (r-tsp (*r*)),
r-sys-data-segment (*r*))),
untag (fetch (r-tsp (*r*),
r-sys-data-segment (*r*)))),
r-word-size (*r*))),
bool (negativep (fix-small-integer (idifference (untag (fetch (pop-stk (r-tsp (*r*)),
r-sys-data-segment (*r*))),
untag (fetch (r-tsp (*r*),
r-sys-data-segment (*r*)))),
r-word-size (*r*)))),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
deposit (bool (ilessp (untag (fetch (pop-stk (r-tsp (*r*)),

r-sys-data-segment (r))),
untag (fetch (r-tsp (r),
r-sys-data-segment (r))))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-lt-int-step-clock (*ins*, r) = 7

DEFINITION:
r-int-to-nat-okp (*ins*, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-objectp-type (**'int**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ (¬ negativep (untag (fetch (r-tsp (r), r-sys-data-segment (r))))))

DEFINITION:
r-int-to-nat-step (*ins*, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
r-tsp (r),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (**'nat**,
untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
r-tsp (r),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-int-to-nat-step-clock (*ins*, r) = 1

DEFINITION:
r-add-nat-okp (*ins*, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))

$\wedge$ r-objectp-type ('**nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
$\wedge$ r-objectp-type ('**nat**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
$\wedge$ small-naturalp (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))
$+$ untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r)))

DEFINITION:
r-add-nat-step (ins, r)
$=$ r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag ('**nat**,
untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))
$+$ untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'**run**)

DEFINITION: r-add-nat-step-clock (ins, r) $=$ 2

DEFINITION:
r-add-nat-with-carry-okp (ins, r)
$=$ (add1-r-pcp (r)
$\wedge$ ($\neg$ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
$\wedge$ ($\neg$ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
$\wedge$ ($\neg$ empty-stkp (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))
$\wedge$ r-objectp-type ('**nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
$\wedge$ r-objectp-type ('**nat**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),

```
                         r)
     ∧   r-objectp-type ('bool,
                         fetch (pop-stk (pop-stk (r-tsp (r))),
                                r-sys-data-segment (r)),
                         r))
```

DEFINITION:

```
r-add-nat-with-carry-step (ins, r)
=   r-state (add1-r-pc (r),
             r-cfp (r),
             r-csp (r),
             pop-stk (r-tsp (r)),
             tag ('nat,
                  fix-small-natural (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                r-sys-data-segment (r))))
                                     +   untag (fetch (pop-stk (r-tsp (r)),
                                                       r-sys-data-segment (r)))
                                     +   untag (fetch (r-tsp (r),
                                                       r-sys-data-segment (r))),
                                     r-word-size (r))),
             fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
             bool (¬ small-naturalp (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                r-sys-data-segment (r))))
                                     +   untag (fetch (pop-stk (r-tsp (r)),
                                                       r-sys-data-segment (r)))
                                     +   untag (fetch (r-tsp (r),
                                                       r-sys-data-segment (r))),
                                     r-word-size (r))),
             r-v-flg (r),
             r-n-flg (r),
             r-z-flg (r),
             r-prog-segment (r),
             r-usr-data-segment (r),
             deposit (tag ('nat,
                           fix-small-natural (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                         r-sys-data-segment (r))))
                                              +   untag (fetch (pop-stk (r-tsp (r)),
                                                                r-sys-data-segment (r)))
                                              +   untag (fetch (r-tsp (r),
                                                                r-sys-data-segment (r))),
                                              r-word-size (r))),
                      pop-stk (r-tsp (r)),
                      deposit (bool (¬ small-naturalp (bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                                                  r-sys-data-segment (r))))
```

```
                                             +   untag (fetch (pop-stk (r-tsp (r)),
                                                               r-sys-data-segment (r)))
                                             +   untag (fetch (r-tsp (r),
                                                               r-sys-data-segment (r))),
                                             r-word-size (r))),
                          pop-stk (pop-stk (r-tsp (r))),
                          r-sys-data-segment (r))),
          r-word-size (r),
          'run)
```

Definition: r-add-nat-with-carry-step-clock (ins, r) = 6

Definition:

```
r-add1-nat-okp (ins, r)
=   (add1-r-pcp (r)
     ∧  (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
     ∧  r-objectp-type ('nat, fetch (r-tsp (r), r-sys-data-segment (r)), r)
     ∧  small-naturalp (1 + untag (fetch (r-tsp (r), r-sys-data-segment (r))),
                        r-word-size (r)))
```

Definition:

```
r-add1-nat-step (ins, r)
=   r-state (add1-r-pc (r),
          r-cfp (r),
          r-csp (r),
          r-tsp (r),
          r-x (r),
          r-y (r),
          r-c-flg (r),
          r-v-flg (r),
          r-n-flg (r),
          r-z-flg (r),
          r-prog-segment (r),
          r-usr-data-segment (r),
          deposit (tag ('nat,
                        1 + untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
                   r-tsp (r),
                   r-sys-data-segment (r)),
          r-word-size (r),
          'run)
```

Definition: r-add1-nat-step-clock (ins, r) = 1

Definition:

r-sub1-nat-okp (ins, r)

$$
\begin{aligned}
= \quad & (\text{add1-r-pcp}\,(r) \\
& \quad \wedge \quad (\neg\ \text{empty-stkp}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r))) \\
& \quad \wedge \quad \text{r-objectp-type}\,(\texttt{'nat},\ \text{fetch}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r)),\ r) \\
& \quad \wedge \quad (\text{untag}\,(\text{fetch}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r))) \not\simeq \texttt{0}))
\end{aligned}
$$

DEFINITION:
r-sub1-nat-step $(ins,\ r)$

$$
\begin{aligned}
= \quad & \text{r-state}\,(\text{add1-r-pc}\,(r), \\
& \qquad \text{r-cfp}\,(r), \\
& \qquad \text{r-csp}\,(r), \\
& \qquad \text{r-tsp}\,(r), \\
& \qquad \text{r-x}\,(r), \\
& \qquad \text{r-y}\,(r), \\
& \qquad \text{r-c-flg}\,(r), \\
& \qquad \text{r-v-flg}\,(r), \\
& \qquad \text{r-n-flg}\,(r), \\
& \qquad \text{r-z-flg}\,(r), \\
& \qquad \text{r-prog-segment}\,(r), \\
& \qquad \text{r-usr-data-segment}\,(r), \\
& \qquad \text{deposit}\,(\text{tag}\,(\texttt{'nat}, \\
& \qquad\qquad\qquad \text{untag}\,(\text{fetch}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r))) - 1), \\
& \qquad\qquad \text{r-tsp}\,(r), \\
& \qquad\qquad \text{r-sys-data-segment}\,(r)), \\
& \qquad \text{r-word-size}\,(r), \\
& \qquad \texttt{'run})
\end{aligned}
$$

DEFINITION: r-sub1-nat-step-clock $(ins,\ r) = \texttt{1}$

DEFINITION:
r-sub-nat-okp $(ins,\ r)$

$$
\begin{aligned}
= \quad & (\text{add1-r-pcp}\,(r) \\
& \quad \wedge \quad (\neg\ \text{empty-stkp}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r))) \\
& \quad \wedge \quad (\neg\ \text{empty-stkp}\,(\text{pop-stk}\,(\text{r-tsp}\,(r)),\ \text{r-sys-data-segment}\,(r))) \\
& \quad \wedge \quad \text{r-objectp-type}\,(\texttt{'nat},\ \text{fetch}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r)),\ r) \\
& \quad \wedge \quad \text{r-objectp-type}\,(\texttt{'nat}, \\
& \qquad\qquad\qquad \text{fetch}\,(\text{pop-stk}\,(\text{r-tsp}\,(r)),\ \text{r-sys-data-segment}\,(r)), \\
& \qquad\qquad\qquad r) \\
& \quad \wedge \quad (\text{untag}\,(\text{fetch}\,(\text{pop-stk}\,(\text{r-tsp}\,(r)),\ \text{r-sys-data-segment}\,(r))) \\
& \qquad\quad \not< \quad \text{untag}\,(\text{fetch}\,(\text{r-tsp}\,(r),\ \text{r-sys-data-segment}\,(r)))))
\end{aligned}
$$

DEFINITION:
r-sub-nat-step $(ins,\ r)$

$$
\begin{aligned}
= \quad & \text{r-state}\,(\text{add1-r-pc}\,(r), \\
& \qquad \text{r-cfp}\,(r), \\
& \qquad \text{r-csp}\,(r),
\end{aligned}
$$

pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (**'nat**,
untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))
− untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-sub-nat-step-clock (ins, r) = 2

DEFINITION:
r-sub-nat-with-carry-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))
∧ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ r-objectp-type (**'nat**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r)
∧ r-objectp-type (**'bool**,
fetch (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)),
r))

DEFINITION:
r-sub-nat-with-carry-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
tag (**'nat**,
if untag (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
< (untag (fetch (r-tsp (r), r-sys-data-segment (r)))

```
             +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                         r-sys-data-segment (r)))))
    then exp (2, r-word-size (r))
           -  ((untag (fetch (r-tsp (r), r-sys-data-segment (r)))
                +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                              r-sys-data-segment (r)))))
               -  untag (fetch (pop-stk (r-tsp (r)),
                                r-sys-data-segment (r))))
    else untag (fetch (pop-stk (r-tsp (r)),
                       r-sys-data-segment (r)))
          -  (untag (fetch (r-tsp (r),
                            r-sys-data-segment (r)))
              +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                            r-sys-data-segment (r))))) endif),
fetch (r-tsp (r), r-sys-data-segment (r)),
bool (untag (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
      <  (untag (fetch (r-tsp (r), r-sys-data-segment (r)))
          +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                        r-sys-data-segment (r)))))),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag ('nat,
              if untag (fetch (pop-stk (r-tsp (r)),
                               r-sys-data-segment (r)))
                 <  (untag (fetch (r-tsp (r),
                                   r-sys-data-segment (r)))
                     +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                   r-sys-data-segment (r)))))
              then exp (2, r-word-size (r))
                     -  ((untag (fetch (r-tsp (r),
                                        r-sys-data-segment (r)))
                          +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                        r-sys-data-segment (r)))))
                         -  untag (fetch (pop-stk (r-tsp (r)),
                                          r-sys-data-segment (r))))
              else untag (fetch (pop-stk (r-tsp (r)),
                                 r-sys-data-segment (r)))
                    -  (untag (fetch (r-tsp (r),
                                      r-sys-data-segment (r)))
                        +  bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
                                                      r-sys-data-segment (r))))) endif),
```

pop-stk (r-tsp (r)),
deposit (bool (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))
$<$ (untag (fetch (r-tsp (r),
r-sys-data-segment (r)))
$+$ bool-to-nat (untag (fetch (pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r)))))),
pop-stk (pop-stk (r-tsp (r))),
r-sys-data-segment (r))),
r-word-size (r),
'run)

DEFINITION: r-sub-nat-with-carry-step-clock (ins, r) $=$ 6

DEFINITION:
r-lt-nat-okp (ins, r)
$=$ (add1-r-pcp (r)
$\land$ ($\neg$ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
$\land$ ($\neg$ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
$\land$ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
$\land$ r-objectp-type (**'nat**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r))

DEFINITION:
r-lt-nat-step (ins, r)
$=$ r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
bool (untag (fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
$<$ untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (bool (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r)))
$<$ untag (fetch (r-tsp (r),
r-sys-data-segment (r)))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),

r-word-size (r),
'**run**)

DEFINITION: r-lt-nat-step-clock (ins, r) = `4`

DEFINITION:
r-mult2-nat-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-objectp-type ('**nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ small-naturalp (`2` ∗ untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r)))

DEFINITION:
r-mult2-nat-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
r-tsp (r),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag ('**nat**,
`2` ∗ untag (fetch (r-tsp (r), r-sys-data-segment (r)))),
r-tsp (r),
r-sys-data-segment (r)),
r-word-size (r),
'**run**)

DEFINITION: r-mult2-nat-step-clock (ins, r) = `1`

DEFINITION:
r-mult2-nat-with-carry-out-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-objectp-type ('**nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ (free-stk-size (r-tsp (r)) $\not\simeq$ `0`))

DEFINITION:

r-mult2-nat-with-carry-out-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
push-stk (r-tsp (r)),
tag (**'nat**,
fix-small-natural (2 ∗ untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r))),
r-y (r),
bool (¬ small-naturalp (2 ∗ untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r))),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (**'nat**,
fix-small-natural (2 ∗ untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r))),
push-stk (r-tsp (r)),
deposit (bool (¬ small-naturalp (2 ∗ untag (fetch (r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r))),
r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r),
'run)

DEFINITION: r-mult2-nat-with-carry-out-step-clock (ins, r) = 5

DEFINITION:
r-div2-nat-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ r-objectp-type (**'nat**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
∧ (free-stk-size (r-tsp (r)) $\not\simeq$ 0))

DEFINITION:
r-div2-nat-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),

push-stk (r-tsp (r)),
tag (**'nat**, untag (fetch (r-tsp (r), r-sys-data-segment (r))) $\div$ 2),
r-y (r),
bool ((untag (fetch (r-tsp (r), r-sys-data-segment (r))) **mod** 2) = 1),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (**'nat**, untag (fetch (r-tsp (r), r-sys-data-segment (r))) **mod** 2),
push-stk (r-tsp (r)),
deposit (tag (**'nat**, untag (fetch (r-tsp (r), r-sys-data-segment (r))) $\div$ 2),
r-tsp (r),
r-sys-data-segment (r))),
r-word-size (r),
'run)

DEFINITION: r-div2-nat-step-clock (ins, r) = 5

DEFINITION:
r-or-bitv-okp (ins, r)
= (add1-r-pcp (r)
$\wedge$ ($\neg$ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
$\wedge$ ($\neg$ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
$\wedge$ r-objectp-type (**'bitv**, fetch (r-tsp (r), r-sys-data-segment (r)), r)
$\wedge$ r-objectp-type (**'bitv**, fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)), r))

DEFINITION:
r-or-bitv-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),

r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
deposit (tag (**'bitv**,
or-bitv (untag (fetch (pop-stk (r-tsp (*r*)),
r-sys-data-segment (*r*))),
untag (fetch (r-tsp (*r*),
r-sys-data-segment (*r*))))),
pop-stk (r-tsp (*r*)),
r-sys-data-segment (*r*)),
r-word-size (*r*),
'run)

DEFINITION: r-or-bitv-step-clock (*ins*, *r*) = 2

DEFINITION:
r-and-bitv-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
∧ (¬ empty-stkp (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
∧ r-objectp-type (**'bitv**,
fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
r)
∧ r-objectp-type (**'bitv**,
fetch (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)),
r))

DEFINITION:
r-and-bitv-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
r-cfp (*r*),
r-csp (*r*),
pop-stk (r-tsp (*r*)),
fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),

```
            deposit (tag ('bitv,
                         and-bitv (untag (fetch (pop-stk (r-tsp (r)),
                                                r-sys-data-segment (r))),
                                   untag (fetch (r-tsp (r),
                                                r-sys-data-segment (r))))),
                    pop-stk (r-tsp (r)),
                    r-sys-data-segment (r)),
            r-word-size (r),
            'run)
```

DEFINITION: r-and-bitv-step-clock (*ins*, *r*) = 2

DEFINITION:

```
r-not-bitv-okp (ins, r)
=   (add1-r-pcp (r)
     ∧  (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
     ∧  r-objectp-type ('bitv,
                        fetch (r-tsp (r), r-sys-data-segment (r)),
                        r))
```

DEFINITION:

```
r-not-bitv-step (ins, r)
=   r-state (add1-r-pc (r),
            r-cfp (r),
            r-csp (r),
            r-tsp (r),
            r-x (r),
            r-y (r),
            r-c-flg (r),
            r-v-flg (r),
            r-n-flg (r),
            r-z-flg (r),
            r-prog-segment (r),
            r-usr-data-segment (r),
            deposit (tag ('bitv,
                         not-bitv (untag (fetch (r-tsp (r),
                                                r-sys-data-segment (r))))),
                    r-tsp (r),
                    r-sys-data-segment (r)),
            r-word-size (r),
            'run)
```

DEFINITION: r-not-bitv-step-clock (*ins*, *r*) = 1

DEFINITION:

r-xor-bitv-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
  ∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
  ∧ (¬ empty-stkp (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)))
  ∧ r-objectp-type (**'bitv**,
      fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
      r)
  ∧ r-objectp-type (**'bitv**,
      fetch (pop-stk (r-tsp (*r*)), r-sys-data-segment (*r*)),
      r))

DEFINITION:
r-xor-bitv-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
    r-cfp (*r*),
    r-csp (*r*),
    pop-stk (r-tsp (*r*)),
    fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
    r-y (*r*),
    r-c-flg (*r*),
    r-v-flg (*r*),
    r-n-flg (*r*),
    r-z-flg (*r*),
    r-prog-segment (*r*),
    r-usr-data-segment (*r*),
    deposit (tag (**'bitv**,
        xor-bitv (untag (fetch (pop-stk (r-tsp (*r*)),
              r-sys-data-segment (*r*))),
          untag (fetch (r-tsp (*r*),
              r-sys-data-segment (*r*))))),
      pop-stk (r-tsp (*r*)),
      r-sys-data-segment (*r*)),
    r-word-size (*r*),
    'run)

DEFINITION: r-xor-bitv-step-clock (*ins*, *r*) = 2

DEFINITION:
r-rsh-bitv-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
  ∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
  ∧ r-objectp-type (**'bitv**,
      fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
      r))

DEFINITION:
r-rsh-bitv-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
r-cfp (*r*),
r-csp (*r*),
r-tsp (*r*),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),
r-usr-data-segment (*r*),
deposit (tag (**'bitv**,
rsh-bitv (untag (fetch (r-tsp (*r*),
r-sys-data-segment (*r*))))),
r-tsp (*r*),
r-sys-data-segment (*r*)),
r-word-size (*r*),
'run)

DEFINITION: r-rsh-bitv-step-clock (*ins*, *r*) = 1

DEFINITION:
r-lsh-bitv-okp (*ins*, *r*)
= (add1-r-pcp (*r*)
∧ (¬ empty-stkp (r-tsp (*r*), r-sys-data-segment (*r*)))
∧ r-objectp-type (**'bitv**,
fetch (r-tsp (*r*), r-sys-data-segment (*r*)),
r))

DEFINITION:
r-lsh-bitv-step (*ins*, *r*)
= r-state (add1-r-pc (*r*),
r-cfp (*r*),
r-csp (*r*),
r-tsp (*r*),
r-x (*r*),
r-y (*r*),
r-c-flg (*r*),
r-v-flg (*r*),
r-n-flg (*r*),
r-z-flg (*r*),
r-prog-segment (*r*),

```
            r-usr-data-segment (r),
            deposit (tag ('bitv,
                         lsh-bitv (untag (fetch (r-tsp (r),
                                                 r-sys-data-segment (r))))),
                    r-tsp (r),
                    r-sys-data-segment (r)),
            r-word-size (r),
            'run)
```

DEFINITION: r-lsh-bitv-step-clock (*ins*, *r*) = 1

DEFINITION:

```
r-or-bool-okp (ins, r)
=   (add1-r-pcp (r)
     ∧  (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
     ∧  (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
     ∧  r-objectp-type ('bool,
                        fetch (r-tsp (r), r-sys-data-segment (r)),
                        r)
     ∧  r-objectp-type ('bool,
                        fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
                        r))
```

DEFINITION:

```
r-or-bool-step (ins, r)
=   r-state (add1-r-pc (r),
            r-cfp (r),
            r-csp (r),
            pop-stk (r-tsp (r)),
            fetch (r-tsp (r), r-sys-data-segment (r)),
            r-y (r),
            r-c-flg (r),
            r-v-flg (r),
            r-n-flg (r),
            r-z-flg (r),
            r-prog-segment (r),
            r-usr-data-segment (r),
            deposit (tag ('bool,
                         or-bool (untag (fetch (pop-stk (r-tsp (r)),
                                                r-sys-data-segment (r))),
                                  untag (fetch (r-tsp (r),
                                                r-sys-data-segment (r))))),
                    pop-stk (r-tsp (r)),
                    r-sys-data-segment (r)),
            r-word-size (r),
```

'**run**)

DEFINITION: r-or-bool-step-clock (ins, r) = 2

DEFINITION:
r-and-bool-okp (ins, r)
= (add1-r-pcp (r)
∧ (¬ empty-stkp (r-tsp (r), r-sys-data-segment (r)))
∧ (¬ empty-stkp (pop-stk (r-tsp (r)), r-sys-data-segment (r)))
∧ r-objectp-type ('**bool**,
fetch (r-tsp (r), r-sys-data-segment (r)),
r)
∧ r-objectp-type ('**bool**,
fetch (pop-stk (r-tsp (r)), r-sys-data-segment (r)),
r))

DEFINITION:
r-and-bool-step (ins, r)
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
pop-stk (r-tsp (r)),
fetch (r-tsp (r), r-sys-data-segment (r)),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag ('**bool**,
and-bool (untag (fetch (pop-stk (r-tsp (r)),
r-sys-data-segment (r))),
untag (fetch (r-tsp (r),
r-sys-data-segment (r))))),
pop-stk (r-tsp (r)),
r-sys-data-segment (r)),
r-word-size (r),
'**run**)

DEFINITION: r-and-bool-step-clock (ins, r) = 2

DEFINITION:
r-not-bool-okp (ins, r)
= (add1-r-pcp (r)

$\wedge$ $(\neg$ empty-stkp (r-tsp (r), r-sys-data-segment $(r)))$
$\wedge$ r-objectp-type (`'bool`,
fetch (r-tsp (r), r-sys-data-segment (r)),
r))

DEFINITION:
r-not-bool-step $(ins,\ r)$
= r-state (add1-r-pc (r),
r-cfp (r),
r-csp (r),
r-tsp (r),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
r-prog-segment (r),
r-usr-data-segment (r),
deposit (tag (`'bool`,
not-bool (untag (fetch (r-tsp (r),
r-sys-data-segment (r)))))),
r-tsp (r),
r-sys-data-segment (r)),
r-word-size (r),
'run)

DEFINITION: r-not-bool-step-clock $(ins,\ r) = 1$

DEFINITION:
r-ins-okp $(ins,\ r)$
= **case on** car (ins):
case = *call*
then r-call-okp $(ins,\ r)$
case = *ret*
then r-ret-okp $(ins,\ r)$
case = *locn*
then r-locn-okp $(ins,\ r)$
case = *push-constant*
then r-push-constant-okp $(ins,\ r)$
case = *push-local*
then r-push-local-okp $(ins,\ r)$
case = *push-global*
then r-push-global-okp $(ins,\ r)$
case = *push-ctrl-stk-free-size*

then r-push-ctrl-stk-free-size-okp (ins, r)
case = *push-temp-stk-free-size*
then r-push-temp-stk-free-size-okp (ins, r)
case = *push-temp-stk-index*
then r-push-temp-stk-index-okp (ins, r)
case = *jump-if-temp-stk-full*
then r-jump-if-temp-stk-full-okp (ins, r)
case = *jump-if-temp-stk-empty*
then r-jump-if-temp-stk-empty-okp (ins, r)
case = *pop*
then r-pop-okp (ins, r)
case = *pop**
then r-pop*-okp (ins, r)
case = *popn*
then r-popn-okp (ins, r)
case = *pop-local*
then r-pop-local-okp (ins, r)
case = *pop-global*
then r-pop-global-okp (ins, r)
case = *pop-locn*
then r-pop-locn-okp (ins, r)
case = *pop-call*
then r-pop-call-okp (ins, r)
case = *fetch-temp-stk*
then r-fetch-temp-stk-okp (ins, r)
case = *deposit-temp-stk*
then r-deposit-temp-stk-okp (ins, r)
case = *jump*
then r-jump-okp (ins, r)
case = *jump-case*
then r-jump-case-okp (ins, r)
case = *pushj*
then r-pushj-okp (ins, r)
case = *popj*
then r-popj-okp (ins, r)
case = *set-local*
then r-set-local-okp (ins, r)
case = *set-global*
then r-set-global-okp (ins, r)
case = *test-nat-and-jump*
then r-test-nat-and-jump-okp (ins, r)
case = *test-int-and-jump*
then r-test-int-and-jump-okp (ins, r)
case = *test-bool-and-jump*

then r-test-bool-and-jump-okp (ins, r)
case = *test-bitv-and-jump*
then r-test-bitv-and-jump-okp (ins, r)
case = *no-op*
then r-no-op-okp (ins, r)
case = *add-addr*
then r-add-addr-okp (ins, r)
case = *sub-addr*
then r-sub-addr-okp (ins, r)
case = *eq*
then r-eq-okp (ins, r)
case = *lt-addr*
then r-lt-addr-okp (ins, r)
case = *fetch*
then r-fetch-okp (ins, r)
case = *deposit*
then r-deposit-okp (ins, r)
case = *add-int*
then r-add-int-okp (ins, r)
case = *add-int-with-carry*
then r-add-int-with-carry-okp (ins, r)
case = *add1-int*
then r-add1-int-okp (ins, r)
case = *sub-int*
then r-sub-int-okp (ins, r)
case = *sub-int-with-carry*
then r-sub-int-with-carry-okp (ins, r)
case = *sub1-int*
then r-sub1-int-okp (ins, r)
case = *neg-int*
then r-neg-int-okp (ins, r)
case = *lt-int*
then r-lt-int-okp (ins, r)
case = *int-to-nat*
then r-int-to-nat-okp (ins, r)
case = *add-nat*
then r-add-nat-okp (ins, r)
case = *add-nat-with-carry*
then r-add-nat-with-carry-okp (ins, r)
case = *add1-nat*
then r-add1-nat-okp (ins, r)
case = *sub-nat*
then r-sub-nat-okp (ins, r)
case = *sub-nat-with-carry*

then r-sub-nat-with-carry-okp (*ins*, *r*)
case = *sub1-nat*
then r-sub1-nat-okp (*ins*, *r*)
case = *lt-nat*
then r-lt-nat-okp (*ins*, *r*)
case = *mult2-nat*
then r-mult2-nat-okp (*ins*, *r*)
case = *mult2-nat-with-carry-out*
then r-mult2-nat-with-carry-out-okp (*ins*, *r*)
case = *div2-nat*
then r-div2-nat-okp (*ins*, *r*)
case = *or-bitv*
then r-or-bitv-okp (*ins*, *r*)
case = *and-bitv*
then r-and-bitv-okp (*ins*, *r*)
case = *not-bitv*
then r-not-bitv-okp (*ins*, *r*)
case = *xor-bitv*
then r-xor-bitv-okp (*ins*, *r*)
case = *rsh-bitv*
then r-rsh-bitv-okp (*ins*, *r*)
case = *lsh-bitv*
then r-lsh-bitv-okp (*ins*, *r*)
case = *or-bool*
then r-or-bool-okp (*ins*, *r*)
case = *and-bool*
then r-and-bool-okp (*ins*, *r*)
case = *not-bool*
then r-not-bool-okp (*ins*, *r*)
otherwise f endcase

DEFINITION:
r-ins-step (*ins*, *r*)
= **case on** car (*ins*):
case = *call*
then r-call-step (*ins*, *r*)
case = *ret*
then r-ret-step (*ins*, *r*)
case = *locn*
then r-locn-step (*ins*, *r*)
case = *push-constant*
then r-push-constant-step (*ins*, *r*)
case = *push-local*
then r-push-local-step (*ins*, *r*)

case = *push-global*
then r-push-global-step (ins, r)
case = *push-ctrl-stk-free-size*
then r-push-ctrl-stk-free-size-step (ins, r)
case = *push-temp-stk-free-size*
then r-push-temp-stk-free-size-step (ins, r)
case = *push-temp-stk-index*
then r-push-temp-stk-index-step (ins, r)
case = *jump-if-temp-stk-full*
then r-jump-if-temp-stk-full-step (ins, r)
case = *jump-if-temp-stk-empty*
then r-jump-if-temp-stk-empty-step (ins, r)
case = *pop*
then r-pop-step (ins, r)
case = *pop**
then r-pop*-step (ins, r)
case = *popn*
then r-popn-step (ins, r)
case = *pop-local*
then r-pop-local-step (ins, r)
case = *pop-global*
then r-pop-global-step (ins, r)
case = *pop-locn*
then r-pop-locn-step (ins, r)
case = *pop-call*
then r-pop-call-step (ins, r)
case = *fetch-temp-stk*
then r-fetch-temp-stk-step (ins, r)
case = *deposit-temp-stk*
then r-deposit-temp-stk-step (ins, r)
case = *jump*
then r-jump-step (ins, r)
case = *jump-case*
then r-jump-case-step (ins, r)
case = *pushj*
then r-pushj-step (ins, r)
case = *popj*
then r-popj-step (ins, r)
case = *set-local*
then r-set-local-step (ins, r)
case = *set-global*
then r-set-global-step (ins, r)
case = *test-nat-and-jump*
then r-test-nat-and-jump-step (ins, r)

case = *test-int-and-jump*
 then r-test-int-and-jump-step (*ins*, *r*)
case = *test-bool-and-jump*
 then r-test-bool-and-jump-step (*ins*, *r*)
case = *test-bitv-and-jump*
 then r-test-bitv-and-jump-step (*ins*, *r*)
case = *no-op*
 then r-no-op-step (*ins*, *r*)
case = *add-addr*
 then r-add-addr-step (*ins*, *r*)
case = *sub-addr*
 then r-sub-addr-step (*ins*, *r*)
case = *eq*
 then r-eq-step (*ins*, *r*)
case = *lt-addr*
 then r-lt-addr-step (*ins*, *r*)
case = *fetch*
 then r-fetch-step (*ins*, *r*)
case = *deposit*
 then r-deposit-step (*ins*, *r*)
case = *add-int*
 then r-add-int-step (*ins*, *r*)
case = *add-int-with-carry*
 then r-add-int-with-carry-step (*ins*, *r*)
case = *add1-int*
 then r-add1-int-step (*ins*, *r*)
case = *sub-int*
 then r-sub-int-step (*ins*, *r*)
case = *sub-int-with-carry*
 then r-sub-int-with-carry-step (*ins*, *r*)
case = *sub1-int*
 then r-sub1-int-step (*ins*, *r*)
case = *neg-int*
 then r-neg-int-step (*ins*, *r*)
case = *lt-int*
 then r-lt-int-step (*ins*, *r*)
case = *int-to-nat*
 then r-int-to-nat-step (*ins*, *r*)
case = *add-nat*
 then r-add-nat-step (*ins*, *r*)
case = *add-nat-with-carry*
 then r-add-nat-with-carry-step (*ins*, *r*)
case = *add1-nat*
 then r-add1-nat-step (*ins*, *r*)

case = *sub-nat*
then r-sub-nat-step (*ins*, *r*)
case = *sub-nat-with-carry*
then r-sub-nat-with-carry-step (*ins*, *r*)
case = *sub1-nat*
then r-sub1-nat-step (*ins*, *r*)
case = *lt-nat*
then r-lt-nat-step (*ins*, *r*)
case = *mult2-nat*
then r-mult2-nat-step (*ins*, *r*)
case = *mult2-nat-with-carry-out*
then r-mult2-nat-with-carry-out-step (*ins*, *r*)
case = *div2-nat*
then r-div2-nat-step (*ins*, *r*)
case = *or-bitv*
then r-or-bitv-step (*ins*, *r*)
case = *and-bitv*
then r-and-bitv-step (*ins*, *r*)
case = *not-bitv*
then r-not-bitv-step (*ins*, *r*)
case = *xor-bitv*
then r-xor-bitv-step (*ins*, *r*)
case = *rsh-bitv*
then r-rsh-bitv-step (*ins*, *r*)
case = *lsh-bitv*
then r-lsh-bitv-step (*ins*, *r*)
case = *or-bool*
then r-or-bool-step (*ins*, *r*)
case = *and-bool*
then r-and-bool-step (*ins*, *r*)
case = *not-bool*
then r-not-bool-step (*ins*, *r*)
otherwise r-halt (*r*, **'run**) **endcase**

DEFINITION:
r-step1 (*ins*, *r*)
= **if** r-ins-okp (*ins*, *r*) **then** r-ins-step (*ins*, *r*)
else r-halt (*r*, x-y-error-msg (**'r**, car (*ins*))) **endif**

DEFINITION:
r-step (*r*)
= **if** r-psw (*r*) = **'run** **then** r-step1 (r-current-instruction (*r*), *r*)
else *r* **endif**

DEFINITION:

r (r, n)
= **if** $n \simeq 0$ **then** r
else r (r-step (r), $n - 1$) **endif**

DEFINITION:
r-ins-step-clock (ins, r)
= **case on** car (ins):
case = *call*
then r-call-step-clock (ins, r)
case = *ret*
then r-ret-step-clock (ins, r)
case = *locn*
then r-locn-step-clock (ins, r)
case = *push-constant*
then r-push-constant-step-clock (ins, r)
case = *push-local*
then r-push-local-step-clock (ins, r)
case = *push-global*
then r-push-global-step-clock (ins, r)
case = *push-ctrl-stk-free-size*
then r-push-ctrl-stk-free-size-step-clock (ins, r)
case = *push-temp-stk-free-size*
then r-push-temp-stk-free-size-step-clock (ins, r)
case = *push-temp-stk-index*
then r-push-temp-stk-index-step-clock (ins, r)
case = *jump-if-temp-stk-full*
then r-jump-if-temp-stk-full-step-clock (ins, r)
case = *jump-if-temp-stk-empty*
then r-jump-if-temp-stk-empty-step-clock (ins, r)
case = *pop*
then r-pop-step-clock (ins, r)
case = *pop**
then r-pop*-step-clock (ins, r)
case = *popn*
then r-popn-step-clock (ins, r)
case = *pop-local*
then r-pop-local-step-clock (ins, r)
case = *pop-global*
then r-pop-global-step-clock (ins, r)
case = *pop-locn*
then r-pop-locn-step-clock (ins, r)
case = *pop-call*
then r-pop-call-step-clock (ins, r)
case = *fetch-temp-stk*

then r-fetch-temp-stk-step-clock $(ins,\ r)$
case $=$ *deposit-temp-stk*
then r-deposit-temp-stk-step-clock $(ins,\ r)$
case $=$ *jump*
then r-jump-step-clock $(ins,\ r)$
case $=$ *jump-case*
then r-jump-case-step-clock $(ins,\ r)$
case $=$ *pushj*
then r-pushj-step-clock $(ins,\ r)$
case $=$ *popj*
then r-popj-step-clock $(ins,\ r)$
case $=$ *set-local*
then r-set-local-step-clock $(ins,\ r)$
case $=$ *set-global*
then r-set-global-step-clock $(ins,\ r)$
case $=$ *test-nat-and-jump*
then r-test-nat-and-jump-step-clock $(ins,\ r)$
case $=$ *test-int-and-jump*
then r-test-int-and-jump-step-clock $(ins,\ r)$
case $=$ *test-bool-and-jump*
then r-test-bool-and-jump-step-clock $(ins,\ r)$
case $=$ *test-bitv-and-jump*
then r-test-bitv-and-jump-step-clock $(ins,\ r)$
case $=$ *no-op*
then r-no-op-step-clock $(ins,\ r)$
case $=$ *add-addr*
then r-add-addr-step-clock $(ins,\ r)$
case $=$ *sub-addr*
then r-sub-addr-step-clock $(ins,\ r)$
case $=$ *eq*
then r-eq-step-clock $(ins,\ r)$
case $=$ *lt-addr*
then r-lt-addr-step-clock $(ins,\ r)$
case $=$ *fetch*
then r-fetch-step-clock $(ins,\ r)$
case $=$ *deposit*
then r-deposit-step-clock $(ins,\ r)$
case $=$ *add-int*
then r-add-int-step-clock $(ins,\ r)$
case $=$ *add-int-with-carry*
then r-add-int-with-carry-step-clock $(ins,\ r)$
case $=$ *add1-int*
then r-add1-int-step-clock $(ins,\ r)$
case $=$ *sub-int*

then r-sub-int-step-clock (ins, r)
case = *sub-int-with-carry*
then r-sub-int-with-carry-step-clock (ins, r)
case = *sub1-int*
then r-sub1-int-step-clock (ins, r)
case = *neg-int*
then r-neg-int-step-clock (ins, r)
case = *lt-int*
then r-lt-int-step-clock (ins, r)
case = *int-to-nat*
then r-int-to-nat-step-clock (ins, r)
case = *add-nat*
then r-add-nat-step-clock (ins, r)
case = *add-nat-with-carry*
then r-add-nat-with-carry-step-clock (ins, r)
case = *add1-nat*
then r-add1-nat-step-clock (ins, r)
case = *sub-nat*
then r-sub-nat-step-clock (ins, r)
case = *sub-nat-with-carry*
then r-sub-nat-with-carry-step-clock (ins, r)
case = *sub1-nat*
then r-sub1-nat-step-clock (ins, r)
case = *lt-nat*
then r-lt-nat-step-clock (ins, r)
case = *mult2-nat*
then r-mult2-nat-step-clock (ins, r)
case = *mult2-nat-with-carry-out*
then r-mult2-nat-with-carry-out-step-clock (ins, r)
case = *div2-nat*
then r-div2-nat-step-clock (ins, r)
case = *or-bitv*
then r-or-bitv-step-clock (ins, r)
case = *and-bitv*
then r-and-bitv-step-clock (ins, r)
case = *not-bitv*
then r-not-bitv-step-clock (ins, r)
case = *xor-bitv*
then r-xor-bitv-step-clock (ins, r)
case = *rsh-bitv*
then r-rsh-bitv-step-clock (ins, r)
case = *lsh-bitv*
then r-lsh-bitv-step-clock (ins, r)
case = *or-bool*

then r-or-bool-step-clock (ins, r)
case = *and-bool*
then r-and-bool-step-clock (ins, r)
case = *not-bool*
then r-not-bool-step-clock (ins, r)
otherwise 0 **endcase**

DEFINITION:
r-step1-clock (ins, r)
= **if** r-ins-okp (ins, r) **then** r-ins-step-clock (ins, r)
else 0 **endif**

DEFINITION:
r-step-clock (r)
= **if** r-psw (r) = `'run` **then** r-step1-clock (r-current-instruction (r), r)
else 0 **endif**

DEFINITION:
clock (r, n)
= **if** $n \simeq 0$ **then** 0
else r-step-clock (r) + clock (r-step (r), $n - 1$) **endif**

```
; Book: r-i.events.  In this section we prove that the r machine
; is correctly implemented on the i machine.
```

EVENT: Disable unlabel.

EVENT: Disable r-usr-data-typep.

EVENT: Disable r-instructionp.

EVENT: Disable i-usr-data-typep.

THEOREM: r-instructionp-observation
r-instructionp (unlabel (ins)) $\rightarrow$ (car (unlabel (ins)) $\neq$ `'dcl`)

THEOREM: r-push-constant-observation
(r-instructionp (unlabel (ins))
$\land$ (car (unlabel (ins)) = `'push-constant`))
$\rightarrow$ (caadr (unlabel (ins)) $\neq$ `'dl`)

```
; This section is stolen from im.events.
```

EVENT: Enable i-objectp-type-simplifier.

EVENT: Disable i-objectp-type.

```
; I will disable type, tag and untag for sanity's sake,
; but prove the obvious relationships first.
```

EVENT: Enable type-tag.

EVENT: Enable untag-tag.

EVENT: Disable type.

EVENT: Disable tag.

EVENT: Disable untag.

```
; And I will do a similar thing for the adp handlers
```

EVENT: Enable adp-name-cons.

EVENT: Enable adp-offset-cons.

EVENT: Enable adp-name-add-adp.

EVENT: Enable adp-offset-add-adp.

EVENT: Enable adp-name-sub-adp.

EVENT: Enable adp-offset-sub-adp.

EVENT: Disable sub-adp.

EVENT: Disable add-adp.

EVENT: Disable adp-name.

EVENT: Disable adp-offset.

```
; The function BOOLEANP is a killer because it
; causes everything to split.  I will disable it
; and prove what I need about it as necessary.
```

EVENT: Disable booleanp.

EVENT: Disable bool-to-logical.

```
; Some accellerators for psw handling...
```

EVENT: Enable not-equal-x-y-error-msg-run.

THEOREM: not-equal-x-y-error-msg-halt
x-y-error-msg $(x, y) \neq$ `'halt`

EVENT: Disable x-y-error-msg.

EVENT: Enable i-psw-i-halt.

EVENT: Disable i-halt.

```
; Now I repeat the i-objectp-type exercise for r-objectp-type:
```

THEOREM: r-objectp-type-simplifier
(r-objectp-type (`'nat`, x, r)
 = ((type (x) = `'nat`)
  $\land$ (cddr (x) = **nil**)
  $\land$ small-naturalp (untag (x), r-word-size (r))))
$\land$ (r-objectp-type (`'int`, x, r)
 = ((type (x) = `'int`)
  $\land$ (cddr (x) = **nil**)
  $\land$ small-integerp (untag (x), r-word-size (r))))
$\land$ (r-objectp-type (`'bitv`, x, r)
 = ((type (x) = `'bitv`)

$\wedge$ (cddr (x) = **nil**)
$\wedge$ bit-vectorp (untag (x), r-word-size (r))))

$\wedge$ (r-objectp-type (`'bool`, x, r)
= ((type (x) = `'bool`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ booleanp (untag (x))))

$\wedge$ (r-objectp-type (`'addr`, x, r)
= ((type (x) = `'addr`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ adpp (untag (x), r-usr-data-segment (r))))

$\wedge$ (r-objectp-type (`'pc`, x, r)
= ((type (x) = `'pc`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ pcpp (untag (x), r-prog-segment (r))))

$\wedge$ (r-objectp-type (`'subr`, x, r)
= ((type (x) = `'subr`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ definedp (untag (x), r-prog-segment (r))))

$\wedge$ (r-objectp-type (`'sys-addr`, x, r)
= ((type (x) = `'sys-addr`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ adpp (untag (x), r-sys-data-segment (r))))

EVENT: Disable r-objectp-type.

```
; The notion of r-usr-data-objectp is important in the definition
; of proper r states.  It is defined as a conjunction of
; r-objectp and r-usr-data-typep.  We do not allow either of those
; to expand in our proofs, except under careful control because they
; case case splits.
```

EVENT: Disable r-objectp.

EVENT: Disable r-usr-data-typep.

```
; I disable icode1 and will enable it when it is needed.
```

EVENT: Disable icode1.

```
; We now prove a lemma that is basically equivalent to
```

```
; (implies (proper-r-statep r load-addr) body-of-proper-r-statep)

; except that the body has been simplified so that it is
; in normal form.  The point is to permit me
; to keep proper-r-statep disabled but to allow each of
; its conjuncts to be proved by backchaining into the
; (proper-r-statep r load-addr) hypothesis that governs every main
; theorem.  I have also tossed in few other facts that
; follow from those conjuncts.

; The following embarrassing rule is needed to cause the
; rewriter to replace the (assoc & &) by the nearly
; explicit albeit larger list expression.
```

THEOREM: equality-stuffer
(assoc (*name*, *alist*) = list (*name*, tag (*type*, *obj*)))
→ (cdr (assoc (*name*, *alist*)) = list (tag (*type*, *obj*)))

THEOREM: listp-assoc-implies-definedp
listp (assoc (*name*, *alist*)) → definedp (*name*, *alist*)

THEOREM: proper-r-statep-properties
proper-r-statep (r, *load-addr*)
→ (r-statep (r)
 ∧ (type (r-pc (r)) = `'pc`)
 ∧ (cddr (r-pc (r)) = **nil**)
 ∧ listp (untag (r-pc (r)))
 ∧ definedp (adp-name (untag (r-pc (r))), r-prog-segment (r))
 ∧ (adp-offset (untag (r-pc (r))) $\in$ **N**)
 ∧ (adp-offset (untag (r-pc (r)))
  < length (program-body (assoc (adp-name (untag (r-pc (r))),
   r-prog-segment (r)))))
 ∧ (type (r-cfp (r)) = `'sys-addr`)
 ∧ listp (untag (r-cfp (r)))
 ∧ (adp-name (untag (r-cfp (r))) = `'cstk`)
 ∧ (cddr (r-cfp (r)) = **nil**)
 ∧ adpp (untag (r-cfp (r)), r-sys-data-segment (r))
 ∧ definedp (`'cstk`, r-sys-data-segment (r))
 ∧ (adp-offset (untag (r-cfp (r))) $\in$ **N**)
 ∧ (adp-offset (untag (r-cfp (r)))
  < length (cdr (assoc (`'cstk`, r-sys-data-segment (r)))))
 ∧ listp (cdr (assoc (`'cstk`, r-sys-data-segment (r))))
 ∧ (type (r-csp (r)) = `'sys-addr`)

```
∧  listp (untag (r-csp (r)))
∧  (adp-name (untag (r-csp (r))) = 'cstk)
∧  (cddr (r-csp (r)) = nil)
∧  adpp (untag (r-csp (r)), r-sys-data-segment (r))
∧  (adp-offset (untag (r-csp (r))) ∈ N)
∧  (adp-offset (untag (r-csp (r)))
     <  length (cdr (assoc ('cstk, r-sys-data-segment (r)))))
∧  (type (r-tsp (r)) = 'sys-addr)
∧  listp (untag (r-tsp (r)))
∧  (adp-name (untag (r-tsp (r))) = 'tstk)
∧  (cddr (r-tsp (r)) = nil)
∧  adpp (untag (r-tsp (r)), r-sys-data-segment (r))
∧  definedp ('tstk, r-sys-data-segment (r))
∧  (adp-offset (untag (r-tsp (r))) ∈ N)
∧  (adp-offset (untag (r-tsp (r)))
     <  length (cdr (assoc ('tstk, r-sys-data-segment (r)))))
∧  listp (cdr (assoc ('tstk, r-sys-data-segment (r))))
∧  (assoc ('full-ctrl-stk-addr, r-sys-data-segment (r))
     =  '(full-ctrl-stk-addr
           (sys-addr (cstk . 0))))
∧  (assoc ('full-temp-stk-addr, r-sys-data-segment (r))
     =  '(full-temp-stk-addr
           (sys-addr (tstk . 0))))
∧  (assoc ('empty-temp-stk-addr, r-sys-data-segment (r))
     =  list ('empty-temp-stk-addr,
              tag ('sys-addr,
                   cons ('tstk,
                         length (cdr (assoc ('tstk,
                                             r-sys-data-segment (r)))) − 1))))
∧  adpp ('(full-ctrl-stk-addr . 0), r-sys-data-segment (r))
∧  adpp ('(full-temp-stk-addr . 0), r-sys-data-segment (r))
∧  adpp ('(empty-temp-stk-addr . 0), r-sys-data-segment (r))
∧  (fetch-adp ('(full-ctrl-stk-addr . 0),
               r-sys-data-segment (r))
     =  '(sys-addr (cstk . 0)))
∧  (fetch-adp ('(full-temp-stk-addr . 0),
               r-sys-data-segment (r))
     =  '(sys-addr (tstk . 0)))
∧  (fetch-adp ('(empty-temp-stk-addr . 0),
               r-sys-data-segment (r))
     =  tag ('sys-addr,
             cons ('tstk,
                   length (cdr (assoc ('tstk,
                                       r-sys-data-segment (r)))) − 1)))
```

$\wedge$ (type (r-c-flg (r)) = '`bool`)
$\wedge$ (cddr (r-c-flg (r)) = **nil**)
$\wedge$ booleanp (untag (r-c-flg (r)))
$\wedge$ (type (r-v-flg (r)) = '`bool`)
$\wedge$ (cddr (r-v-flg (r)) = **nil**)
$\wedge$ booleanp (untag (r-v-flg (r)))
$\wedge$ (type (r-n-flg (r)) = '`bool`)
$\wedge$ (cddr (r-n-flg (r)) = **nil**)
$\wedge$ booleanp (untag (r-n-flg (r)))
$\wedge$ (type (r-z-flg (r)) = '`bool`)
$\wedge$ (cddr (r-z-flg (r)) = **nil**)
$\wedge$ booleanp (untag (r-z-flg (r)))
$\wedge$ r-prog-segmentp (r-prog-segment (r))
$\wedge$ proper-r-usr-data-segmentp (r-usr-data-segment (r), r)
$\wedge$ ((*load-addr*
  + segment-length (icompile (r-prog-segment (r)))
  + segment-length (r-usr-data-segment (r))
  + segment-length (r-sys-data-segment (r)))
 < exp (2, r-word-size (r))))

Event: Disable listp-assoc-implies-definedp.

Event: Disable equality-stuffer.

```
; Having proved that, we will
```

Event: Disable proper-r-statep.

```
; I now begin the study of find-label and pc mapping
```

Theorem: find-labelp-is-find-label
find-labelp (lab, lst) = (find-label (lab, lst) < length (lst))

Theorem: find-label-append
find-label (lab, append (a, b))
= **if** find-labelp (lab, a) **then** find-label (lab, a)
  else length (a) + find-label (lab, b) **endif**

Event: Enable difference-x-x.

Event: Enable length-append.

```
; The following odd rule is needed because we sometimes treat
; a tagged object, e.g., the constant following a move_x_*,
; as an instruction and ask if it is labelled.
```

THEOREM: car-tag
car (tag (*type*, *obj*)) = *type*

THEOREM: find-label-jump_*-lst
find-label (*lab*, jump_*-lst (*lst*, *program*)) = length (jump_*-lst (*lst*, *program*))

THEOREM: find-label-if
find-label (*lab*,
            if *a* **then** *b*
            else *c* **endif**)
= **if** *a* **then** find-label (*lab*, *b*)
  else find-label (*lab*, *c*) **endif**

THEOREM: length-if{jsm}
length (**if** *a* **then** *b*
        else *c* **endif**)
= **if** *a* **then** length (*b*)
  else length (*c*) **endif**

```
; The icode1 fn is too big for the theorem prover's rewriter to
; explore on the standard akcl stack.   So I cut the function
; into two parts:
```

DEFINITION:
icode12 (*ins*, *pcn*, *prog*)
= **case on** car (*ins*):
  case = *eq*
  then icode-eq (*ins*, *pcn*, *prog*)
  case = *lt-addr*
   then icode-lt-addr (*ins*, *pcn*, *prog*)
  case = *fetch*
   then icode-fetch (*ins*, *pcn*, *prog*)
  case = *deposit*
   then icode-deposit (*ins*, *pcn*, *prog*)
  case = *add-int*
   then icode-add-int (*ins*, *pcn*, *prog*)
  case = *add-int-with-carry*
   then icode-add-int-with-carry (*ins*, *pcn*, *prog*)
  case = *add1-int*

then icode-add1-int (*ins*, *pcn*, *prog*)
case = *sub-int*
then icode-sub-int (*ins*, *pcn*, *prog*)
case = *sub-int-with-carry*
then icode-sub-int-with-carry (*ins*, *pcn*, *prog*)
case = *sub1-int*
then icode-sub1-int (*ins*, *pcn*, *prog*)
case = *neg-int*
then icode-neg-int (*ins*, *pcn*, *prog*)
case = *lt-int*
then icode-lt-int (*ins*, *pcn*, *prog*)
case = *int-to-nat*
then icode-int-to-nat (*ins*, *pcn*, *prog*)
case = *add-nat*
then icode-add-nat (*ins*, *pcn*, *prog*)
case = *add-nat-with-carry*
then icode-add-nat-with-carry (*ins*, *pcn*, *prog*)
case = *add1-nat*
then icode-add1-nat (*ins*, *pcn*, *prog*)
case = *sub-nat*
then icode-sub-nat (*ins*, *pcn*, *prog*)
case = *sub-nat-with-carry*
then icode-sub-nat-with-carry (*ins*, *pcn*, *prog*)
case = *sub1-nat*
then icode-sub1-nat (*ins*, *pcn*, *prog*)
case = *lt-nat*
then icode-lt-nat (*ins*, *pcn*, *prog*)
case = *mult2-nat*
then icode-mult2-nat (*ins*, *pcn*, *prog*)
case = *mult2-nat-with-carry-out*
then icode-mult2-nat-with-carry-out (*ins*, *pcn*, *prog*)
case = *div2-nat*
then icode-div2-nat (*ins*, *pcn*, *prog*)
case = *or-bitv*
then icode-or-bitv (*ins*, *pcn*, *prog*)
case = *and-bitv*
then icode-and-bitv (*ins*, *pcn*, *prog*)
case = *not-bitv*
then icode-not-bitv (*ins*, *pcn*, *prog*)
case = *xor-bitv*
then icode-xor-bitv (*ins*, *pcn*, *prog*)
case = *rsh-bitv*
then icode-rsh-bitv (*ins*, *pcn*, *prog*)
case = *lsh-bitv*

then icode-lsh-bitv (*ins*, *pcn*, *prog*)
case = *or-bool*
then icode-or-bool (*ins*, *pcn*, *prog*)
case = *and-bool*
then icode-and-bool (*ins*, *pcn*, *prog*)
case = *not-bool*
then icode-not-bool (*ins*, *pcn*, *prog*)
otherwise `'((error))` **endcase**

DEFINITION:
icode11 (*ins*, *pcn*, *prog*)
= **case on** car (*ins*):
case = *call*
then icode-call (*ins*, *pcn*, *prog*)
case = *ret*
then icode-ret (*ins*, *pcn*, *prog*)
case = *locn*
then icode-locn (*ins*, *pcn*, *prog*)
case = *push-constant*
then icode-push-constant (*ins*, *pcn*, *prog*)
case = *push-local*
then icode-push-local (*ins*, *pcn*, *prog*)
case = *push-global*
then icode-push-global (*ins*, *pcn*, *prog*)
case = *push-ctrl-stk-free-size*
then icode-push-ctrl-stk-free-size (*ins*, *pcn*, *prog*)
case = *push-temp-stk-free-size*
then icode-push-temp-stk-free-size (*ins*, *pcn*, *prog*)
case = *push-temp-stk-index*
then icode-push-temp-stk-index (*ins*, *pcn*, *prog*)
case = *jump-if-temp-stk-full*
then icode-jump-if-temp-stk-full (*ins*, *pcn*, *prog*)
case = *jump-if-temp-stk-empty*
then icode-jump-if-temp-stk-empty (*ins*, *pcn*, *prog*)
case = *pop*
then icode-pop (*ins*, *pcn*, *prog*)
case = *pop**
then icode-pop* (*ins*, *pcn*, *prog*)
case = *popn*
then icode-popn (*ins*, *pcn*, *prog*)
case = *pop-local*
then icode-pop-local (*ins*, *pcn*, *prog*)
case = *pop-global*
then icode-pop-global (*ins*, *pcn*, *prog*)

case = *pop-locn*
then icode-pop-locn (*ins*, *pcn*, *prog*)
case = *pop-call*
then icode-pop-call (*ins*, *pcn*, *prog*)
case = *fetch-temp-stk*
then icode-fetch-temp-stk (*ins*, *pcn*, *prog*)
case = *deposit-temp-stk*
then icode-deposit-temp-stk (*ins*, *pcn*, *prog*)
case = *jump*
then icode-jump (*ins*, *pcn*, *prog*)
case = *jump-case*
then icode-jump-case (*ins*, *pcn*, *prog*)
case = *pushj*
then icode-pushj (*ins*, *pcn*, *prog*)
case = *popj*
then icode-popj (*ins*, *pcn*, *prog*)
case = *set-local*
then icode-set-local (*ins*, *pcn*, *prog*)
case = *set-global*
then icode-set-global (*ins*, *pcn*, *prog*)
case = *test-nat-and-jump*
then icode-test-nat-and-jump (*ins*, *pcn*, *prog*)
case = *test-int-and-jump*
then icode-test-int-and-jump (*ins*, *pcn*, *prog*)
case = *test-bool-and-jump*
then icode-test-bool-and-jump (*ins*, *pcn*, *prog*)
case = *test-bitv-and-jump*
then icode-test-bitv-and-jump (*ins*, *pcn*, *prog*)
case = *no-op*
then icode-no-op (*ins*, *pcn*, *prog*)
case = *add-addr*
then icode-add-addr (*ins*, *pcn*, *prog*)
case = *sub-addr*
then icode-sub-addr (*ins*, *pcn*, *prog*)
otherwise icode12 (*ins*, *pcn*, *prog*) **endcase**

THEOREM: icode1-is-icode11
icode1 (*ins*, *pcn*, *prog*) = icode11 (*ins*, *pcn*, *prog*)

EVENT: Disable icode11.

EVENT: Disable icode12.

THEOREM: find-label-icode12

r-instructionp (unlabel (ins))
$\rightarrow$ (find-label (lab, icode12 (unlabel (ins), pcn, $program$))
= length (icode12 (unlabel (ins), pcn, $program$)))

THEOREM: find-label-icode1
r-instructionp (unlabel (ins))
$\rightarrow$ (find-label (lab, icode1 (unlabel (ins), pcn, $program$))
= length (icode1 (unlabel (ins), pcn, $program$)))

EVENT: Disable icode1-is-icode11.

THEOREM: find-label-failure-extends-to-cdr
(find-label (lab, lst) = length (lst))
$\rightarrow$ (find-label (lab, cdr (lst)) = length (cdr (lst)))

THEOREM: find-label-icode
r-instructionp (unlabel (ins))
$\rightarrow$ (find-label (lab, icode (ins, pcn, $program$))
= **if** lab = cons (name ($program$), pcn) **then** 0
else length (icode (ins, pcn, $program$)) **endif**)

THEOREM: find-label-icompile-program-body-lemma1
(listp (lab) $\wedge$ (car (lab) = car ($program$)) $\wedge$ (cdr (lab) = pcn))
$\rightarrow$ (find-label (lab, icompile-program-body ($body$, pcn, $program$)) = 0)

EVENT: Enable equal-length-0.

THEOREM: listp-first-n
listp (first-n (n, x)) = ($n \not\simeq 0$)

THEOREM: length-first-n
length (first-n (n, x)) = fix (n)

THEOREM: zerop-difference
(($x - y$) = 0) = ($y \not< x$)

THEOREM: car-first-n
car (first-n (n, x))
= **if** $n \simeq 0$ **then** 0
else car (x) **endif**

THEOREM: find-label-icompile-program-body-lemma2
($pcn < n$)
$\rightarrow$ (first-n (($n - 1$) $-$ pcn, cdr (b)) = cdr (first-n ($n - pcn$, b)))

```
; The following hackish lemma is dangerous because it folds up
; difference.  I'll disable it as soon as it has done its bit.
; The term (cdr lab) could be replaced by n and the four hyps could
; be replaced by (lessp pcn n), but things are this way so it
; finds immediate use in the next proof.
```

THEOREM: find-label-icompile-program-body-lemma3
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{listp}\,(lab)} \\
\land & (\text{cdr}\,(lab) \in \mathbf{N}) \\
\land & (pcn \in \mathbf{N}) \\
\land & (\text{cdr}\,(lab) \not< pcn) \\
\land & (lab \neq \text{cons}\,(\text{car}\,(lab),\ pcn))) \\
\rightarrow & (\text{first-n}\,((\text{cdr}\,(lab) - 1) - pcn,\ \text{cdr}\,(b)) \\
 & \quad = \quad \text{cdr}\,(\text{first-n}\,(\text{cdr}\,(lab) - pcn,\ b)))
\end{array}
$$

```
; In the useful statement of the relationship below, lab is an pcpp,
; namely an untagged r-level pc.  But for the proof itself lab is
; a cons pair.  So we have to enable adp-name and adp-offset.
```

THEOREM: find-label-icompile-program-body
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{listp}\,(lab)} \\
\land & (\text{adp-name}\,(lab) = \text{name}\,(program)) \\
\land & (\text{adp-offset}\,(lab) \in \mathbf{N}) \\
\land & (pcn \in \mathbf{N}) \\
\land & (\text{adp-offset}\,(lab) \not< pcn) \\
\land & ((pcn + \text{length}\,(body)) \not< \text{adp-offset}\,(lab)) \\
\land & \text{r-program-bodyp}\,(body)) \\
\rightarrow & (\text{find-label}\,(lab,\ \text{icompile-program-body}\,(body,\ pcn,\ program)) \\
 & \quad = \quad \text{length}\,(\text{icompile-program-body}\,(\text{first-n}\,(\text{adp-offset}\,(lab) - pcn,\ body),\ pcn,\ program)))
\end{array}
$$

EVENT: Disable find-label-icompile-program-body-lemma3.

THEOREM: assoc-of-append
$$\text{append}\,(\text{append}\,(a,\ b),\ c) = \text{append}\,(a,\ \text{append}\,(b,\ c))$$

```
; !!! Here we enable plus-add1 (which is a conjunct that hits an add1
; in either side of a plus) to replace a lemma that hit (plus x (add1 y))
; only.
```

EVENT: Enable plus-add1.

THEOREM: icompile-program-body-append

$$
\begin{aligned}
&(pcn \in \mathbf{N})\\
\rightarrow\quad &(\text{icompile-program-body}(\text{append}(body1,\ body2),\ pcn,\ program)\\
&\quad = \text{append}(\text{icompile-program-body}(body1,\ pcn,\ program),\\
&\qquad\qquad \text{icompile-program-body}(body2,\\
&\qquad\qquad\qquad \text{length}(body1) + pcn,\\
&\qquad\qquad\qquad program)))
\end{aligned}
$$

```
; !!! Plus-0 is more powerful than the rule we're replacing,
; (plus x 0) = (fix x).
```

EVENT: Enable plus-0.

THEOREM: plus-difference

$$
\begin{aligned}
&((x - y) + y)\\
= \quad &\textbf{if } x < y \textbf{ then } \text{fix}(y)\\
&\textbf{else } \text{fix}(x) \textbf{ endif}
\end{aligned}
$$

```
; The next two lemmas are of questionable value in indiscriminant
; use so they will be disabled as soon as they've been used where needed.
```

THEOREM: difference-add1

$$(x \not< y) \rightarrow (((1 + x) - y) = (1 + (x - y)))$$

THEOREM: open-first-n-at-high-end

$$\text{first-n}(1 + n,\ x) = \text{append}(\text{first-n}(n,\ x),\ \text{list}(\text{get}(n,\ x)))$$

THEOREM: find-label-add-adp-1-icompile-program-body

$$
\begin{aligned}
&(\text{listp}(r\text{-}pc)\\
\land\quad &(\text{name}(program) = \text{adp-name}(r\text{-}pc))\\
\land\quad &(\text{adp-offset}(r\text{-}pc) \in \mathbf{N})\\
\land\quad &(pcn \in \mathbf{N})\\
\land\quad &(\text{adp-offset}(r\text{-}pc) \not< pcn)\\
\land\quad &(\text{adp-offset}(r\text{-}pc) < (pcn + \text{length}(body)))\\
\land\quad &\text{r-program-bodyp}(body))\\
\rightarrow\quad &(\text{find-label}(\text{add-adp}(r\text{-}pc,\ \mathtt{1}),\ \text{icompile-program-body}(body,\ pcn,\ program))\\
&\quad = (\text{find-label}(r\text{-}pc,\ \text{icompile-program-body}(body,\ pcn,\ program))\\
&\qquad + \text{length}(\text{icode}(\text{get}(\text{adp-offset}(r\text{-}pc) - pcn,\ body),\\
&\qquad\qquad \text{adp-offset}(r\text{-}pc),\\
&\qquad\qquad program))))
\end{aligned}
$$

EVENT: Disable difference-add1.

EVENT: Disable open-first-n-at-high-end.

```
; We are next headed for the result that no labels occur in the prelude
; (at least, no labels of the form (cons name n) where n is numeric).

; However, we will first
```

EVENT: Disable name.

EVENT: Disable formal-vars.

EVENT: Disable temp-var-dcls.

EVENT: Disable program-body.

```
; This probably could be done much earlier.
```

EVENT: Enable length-rev.

EVENT: Enable associativity-of-plus.

THEOREM: find-label-generate-prelude1
r-temp-var-dclsp (*var-dcls*)
$\rightarrow$ (find-label (*lab*, generate-prelude1 (*var-dcls*))
$=$ length (generate-prelude1 (*var-dcls*)))

THEOREM: not-find-label-generate-prelude2
find-label (*lab*, generate-prelude2 (*formal-vars*))
$=$ length (generate-prelude2 (*formal-vars*))

THEOREM: r-temp-var-dclsp-append
r-temp-var-dclsp (append (*var-dcls1*, *var-dcls2*))
$=$ (r-temp-var-dclsp (*var-dcls1*) $\wedge$ r-temp-var-dclsp (*var-dcls2*))

THEOREM: r-temp-var-dclsp-rev
r-temp-var-dclsp (rev (*var-dcls*)) $=$ r-temp-var-dclsp (*var-dcls*)

```
; The following is necessary because generate-prelude was defined
; with REVERSE (which is REV1) rather than with REV.
```

THEOREM: reverse-is-rev
reverse $(x) = \text{rev}(x)$

EVENT: Disable reverse.

THEOREM: find-label-generate-prelude
$(\text{r-temp-var-dclsp}(\text{temp-var-dcls}(prog)) \land (\text{adp-offset}(lab) \in \mathbf{N}))$
$\rightarrow \quad (\text{find-label}(lab,\ \text{generate-prelude}(prog))$
$\quad\quad = \quad \text{length}(\text{generate-prelude}(prog)))$

EVENT: Disable generate-prelude.

EVENT: Disable generate-postlude.

```
; Now we begin the study of labels in the icompiled body
```

THEOREM: member-assoc
$\text{definedp}(x,\ a) \rightarrow (\text{assoc}(x,\ a) \in a)$

THEOREM: difference-x-0
$(x - 0) = \text{fix}(x)$

EVENT: Enable car-assoc.

EVENT: Enable length-cons.

THEOREM: listp-if
listp (**if** a **then** b
else c **endif**)
= **if** a **then** listp (b)
else listp (c) **endif**

```
; Note that the following lemma is stored as a type-prescription and
; hence the dependency of subsequent facts on this may not be apparent.
```

THEOREM: listp-icode12
$\text{listp}(\text{icode12}(ins,\ pcn,\ prog))$

THEOREM: listp-icode1
listp (icode1 (*ins*, *pcn*, *prog*))

THEOREM: listp-icompile-program-body
listp (icompile-program-body (*body*, *pcn*, *prog*)) = listp (*body*)

DEFINITION:
induct-hint1 (*n*, *body*, *pcn*)
= **if** $n \simeq 0$ **then t**
elseif *body* $\simeq$ **nil then t**
else induct-hint1 ($n - 1$, cdr (*body*), $1 + pcn$) **endif**

THEOREM: weak-monotonicity-of-icompile-program-body
$((n <$ length (*body*)$) \wedge (pcn \in \mathbf{N}))$
$\rightarrow$ (length (icompile-program-body (*body*, *pcn*, *prog*))
$\not<$ (length (icompile-program-body (first-n (*n*, *body*), *pcn*, *prog*))
$+$ length (icode (get (*n*, *body*), $n + pcn$, *prog*))))

```
; The next major goal is R->I_PC-TAG-ADD-ADP-1, which has the hypothesis
; that R is a proper-r-statep.  We need to be able to recover various
; things from that.  We get those lemmas now.  The trick to the lemmas
; below is that we don't want free vars in them.  So I prove the
; interesting fact as a lemma of class nil and then instantiate
; it to components of the proper-r-statep r.  In the "instantiated"
; version I also backchain through the proper-r-statep-observations
; to get the single hypothesis (proper-r-statep r load-addr).
```

THEOREM: r-program-bodyp-program-body-lemma
(r-prog-segmentp (*prog-segment*) $\wedge$ (*prog* $\in$ *prog-segment*))
$\rightarrow$ r-program-bodyp (program-body (*prog*))

THEOREM: r-program-bodyp-program-body
proper-r-statep (*r*, *load-addr*)
$\rightarrow$ r-program-bodyp (program-body (assoc (adp-name (untag (r-pc (*r*))),
r-prog-segment (*r*))))

THEOREM: r-temp-var-dclsp-temp-var-dcls-lemma
(r-prog-segmentp (*prog-segment*) $\wedge$ (*prog* $\in$ *prog-segment*))
$\rightarrow$ r-temp-var-dclsp (temp-var-dcls (*prog*))

THEOREM: r-temp-var-dclsp-temp-var-dcls
proper-r-statep (*r*, *load-addr*)
$\rightarrow$ r-temp-var-dclsp (temp-var-dcls (assoc (adp-name (untag (r-pc (*r*))),
r-prog-segment (*r*))))

```
; Now we begin our attack directly on the lemma that says
; you can map down an incremented r-pc by mapping down the
; the r-pc and then incrementing by the length of icode.
```

THEOREM: equal-tag
$(\text{tag}(t1,\ obj1) = \text{tag}(t2,\ obj2)) = ((t1 = t2) \wedge (obj1 = obj2))$

THEOREM: add-adp-cons
$\text{add-adp}(\text{cons}(name,\ n),\ m) = \text{cons}(name,\ n + m)$

THEOREM: name-assoc
$\text{definedp}(name,\ alist) \rightarrow (\text{name}(\text{assoc}(name,\ alist)) = name)$

THEOREM: find-label-generate-postlude
$\text{find-label}(lab,\ \text{generate-postlude}(prog))$
$= \ \textbf{if}\ lab = \text{cons}(\text{name}(prog),\ \text{length}(\text{program-body}(prog)))\ \textbf{then}\ \mathtt{0}$
$\quad \textbf{else}\ \text{length}(\text{generate-postlude}(prog))\ \textbf{endif}$

THEOREM: equal-add-adp-cons-adp-name
$(\text{add-adp}(x,\ n) = \text{cons}(\text{adp-name}(x),\ y)) = ((\text{adp-offset}(x) + n) = y)$

THEOREM: weak-monotonicity-of-icompile-program-body-0
$((n < \text{length}(body)) \wedge (n \in \mathbf{N}))$
$\rightarrow \quad (\text{length}(\text{icompile-program-body}(body,\ \mathtt{0},\ prog))$
$\quad \not< \quad (\text{length}(\text{icompile-program-body}(\text{first-n}(n,\ body),\ \mathtt{0},\ prog))$
$\quad\quad + \quad \text{length}(\text{icode}(\text{get}(n,\ body),\ n,\ prog))))$

THEOREM: leq-means-lessp-is-not-equal
$((b \not< (a + i)) \wedge (b \in \mathbf{N}))$
$\rightarrow \quad (((a + i) < b) = ((a + i) \neq b))$

THEOREM: equal-plus-cancellation1
$(x = (x + y)) = ((x \in \mathbf{N}) \wedge (y \simeq \mathtt{0}))$

THEOREM: last-instruction
$((n < \text{length}(body)) \wedge (n \in \mathbf{N}) \wedge (pcn \in \mathbf{N}))$
$\rightarrow \quad (((\text{length}(\text{icompile-program-body}(\text{first-n}(n,\ body),\ pcn,\ prog))$
$\quad\quad + \quad \text{length}(\text{icode}(\text{get}(n,\ body),\ n + pcn,\ prog)))$
$\quad\quad = \quad \text{length}(\text{icompile-program-body}(body,\ pcn,\ prog)))$
$\quad = \quad (n = (\text{length}(body) - 1)))$

THEOREM: last-instruction-0
$((n < \text{length}(body)) \wedge (n \in \mathbf{N}))$
$\rightarrow \quad (((\text{length}(\text{icompile-program-body}(\text{first-n}(n,\ body),\ \mathtt{0},\ prog))$
$\quad\quad + \quad \text{length}(\text{icode}(\text{get}(n,\ body),\ n,\ prog)))$
$\quad\quad = \quad \text{length}(\text{icompile-program-body}(body,\ \mathtt{0},\ prog)))$
$\quad = \quad (n = (\text{length}(body) - 1)))$

THEOREM: equal-plus-cancellation2
$((x + y) = (x + z)) = (\text{fix}\,(y) = \text{fix}\,(z))$

THEOREM: r->i_pc-tag-add-adp-1-lemma1
$$\begin{array}{l}
(((1 + n) = \text{length}\,(\mathit{body})) \land (n \in \mathbf{N}) \land (\mathit{pcn} \in \mathbf{N})) \\
\rightarrow \quad ((\text{length}\,(\text{icompile-program-body}\,(\mathit{body},\ \mathit{pcn},\ \mathit{prog})) \\
\qquad\quad = \quad (\text{length}\,(\text{icompile-program-body}\,(\text{first-n}\,(n,\ \mathit{body}),\ \mathit{pcn},\ \mathit{prog})) \\
\qquad\qquad\quad + \quad \text{length}\,(\text{icode}\,(\text{get}\,(n,\ \mathit{body}),\ n + \mathit{pcn},\ \mathit{prog})))) \\
\qquad = \quad \mathbf{t})
\end{array}$$

THEOREM: r->i_pc-tag-add-adp-1-lemma1-0
$$\begin{array}{l}
(((1 + n) = \text{length}\,(\mathit{body})) \land (n \in \mathbf{N})) \\
\rightarrow \quad ((\text{length}\,(\text{icompile-program-body}\,(\mathit{body},\ \mathtt{0},\ \mathit{prog})) \\
\qquad\quad = \quad (\text{length}\,(\text{icompile-program-body}\,(\text{first-n}\,(n,\ \mathit{body}),\ \mathtt{0},\ \mathit{prog})) \\
\qquad\qquad\quad + \quad \text{length}\,(\text{icode}\,(\text{get}\,(n,\ \mathit{body}),\ n,\ \mathit{prog})))) \\
\qquad = \quad \mathbf{t})
\end{array}$$

THEOREM: r->i_pc-tag-add-adp-1-lemma2
$$\begin{array}{l}
((n \in \mathbf{N}) \land (\mathit{pcn} \in \mathbf{N})) \\
\rightarrow \quad ((\text{length}\,(\text{icompile-program-body}\,(\mathit{body},\ \mathit{pcn},\ \mathit{prog})) \\
\qquad\quad = \quad \text{length}\,(\text{icompile-program-body}\,(\text{first-n}\,(n,\ \mathit{body}),\ \mathit{pcn},\ \mathit{prog}))) \\
\qquad = \quad (n = \text{length}\,(\mathit{body})))
\end{array}$$

THEOREM: r->i_pc-tag-add-adp-1
$$\begin{array}{l}
\text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr}) \\
\rightarrow \quad (\text{r->i_pc}\,(\text{tag}\,(\mathtt{'pc},\ \text{add-adp}\,(\text{untag}\,(\text{r-pc}\,(r)),\ \mathtt{1})),\ \text{r-prog-segment}\,(r)) \\
\qquad = \quad \text{add-addr}\,(\text{r->i_pc}\,(\text{r-pc}\,(r),\ \text{r-prog-segment}\,(r)), \\
\qquad\qquad\qquad\qquad \text{length}\,(\text{icode}\,(\text{get}\,(\text{offset}\,(\text{r-pc}\,(r)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{program-body}\,(\text{r-current-program}\,(r))), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{offset}\,(\text{r-pc}\,(r)), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{definition}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{r-prog-segment}\,(r))))))
\end{array}$$

THEOREM: assoc-icompile
$$\begin{array}{l}
\text{definedp}\,(\mathit{name},\ \mathit{progs}) \\
\rightarrow \quad (\text{assoc}\,(\mathit{name},\ \text{icompile}\,(\mathit{progs})) \\
\qquad = \quad \text{icompile-program}\,(\text{definition}\,(\mathit{name},\ \mathit{progs})))
\end{array}$$

THEOREM: get-car-cdr-n
$\text{get}\,(n,\ x) = \text{car}\,(\text{cdr-n}\,(n,\ x))$

THEOREM: cdr-n-plus
$\text{cdr-n}\,(m + n,\ x) = \text{cdr-n}\,(n,\ \text{cdr-n}\,(m,\ x))$

THEOREM: cdr-n-append
$(\text{length}\,(a) \not< n) \rightarrow (\text{cdr-n}\,(n,\ \text{append}\,(a,\ b)) = \text{append}\,(\text{cdr-n}\,(n,\ a),\ b))$

THEOREM: listp-cdr-n
$\text{listp}(\text{cdr-n}(n,\ x)) = (n < \text{length}(x))$

THEOREM: not-lessp-length-find-label
$\text{length}(x) \not< \text{find-label}(lab,\ x)$

```
; I used to prove this but I'm not sure I'll use it so it has been skipped
;(prove-lemma first-n-cdr-n (rewrite)
;  (implies (lessp n (length x))
;           (equal (append (first-n n x) (cdr-n n x)) x)))
```

THEOREM: car-append
$\text{listp}(a) \rightarrow (\text{car}(\text{append}(a,\ b)) = \text{car}(a))$

```
; here is my new version of what is shown below...
```

THEOREM: cdr-n-length-icompile-program-body-first-n

$$
\begin{aligned}
&((\text{length}(body) \not< n) \land (n \in \mathbf{N}) \land (pcn \in \mathbf{N}))\\
\rightarrow\quad &(\text{cdr-n}(\text{length}(\text{icompile-program-body}(\text{first-n}(n,\ body),\ pcn,\ prog)),\\
&\qquad\quad \text{icompile-program-body}(body,\ pcn,\ prog))\\
&\quad = \text{icompile-program-body}(\text{cdr-n}(n,\ body),\ n + pcn,\ prog))
\end{aligned}
$$

THEOREM: icompile-program-body-cdr-n-expander

$$
\begin{aligned}
&\text{proper-r-statep}(r,\ load\text{-}addr)\\
\rightarrow\quad &(\text{icompile-program-body}(\text{cdr-n}(\text{adp-offset}(\text{untag}(\text{r-pc}(r))),\\
&\qquad \text{program-body}(\text{assoc}(\text{adp-name}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad \text{r-prog-segment}(r)))),\\
&\qquad \text{adp-offset}(\text{untag}(\text{r-pc}(r))),\\
&\qquad \text{assoc}(\text{adp-name}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad \text{r-prog-segment}(r)))\\
&\quad = \text{append}(\text{icode}(\text{car}(\text{cdr-n}(\text{adp-offset}(\text{untag}(\text{r-pc}(r))),\\
&\qquad \text{program-body}(\text{assoc}(\text{adp-name}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad \text{r-prog-segment}(r))))),\\
&\qquad \text{adp-offset}(\text{untag}(\text{r-pc}(r))),\\
&\qquad \text{assoc}(\text{adp-name}(\text{untag}(\text{r-pc}(r))),\ \text{r-prog-segment}(r))),\\
&\qquad \text{icompile-program-body}(\text{cdr}(\text{cdr-n}(\text{adp-offset}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad \text{program-body}(\text{assoc}(\text{adp-name}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad\qquad \text{r-prog-segment}(r))))),\\
&\qquad\qquad 1 + \text{adp-offset}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad \text{assoc}(\text{adp-name}(\text{untag}(\text{r-pc}(r))),\\
&\qquad\qquad\qquad \text{r-prog-segment}(r)))))
\end{aligned}
$$

THEOREM: fetch-adp-add-adp-untag-r->i
(proper-r-statep (r, *load-addr*)
$\wedge$ ($n <$ length (icode (get (offset (r-pc (r)),
program-body (definition (adp-name (untag (r-pc (r))),
r-prog-segment (r)))),
offset (r-pc (r)),
definition (adp-name (untag (r-pc (r))),
r-prog-segment (r))))))
$\rightarrow$ (fetch-adp (add-adp (untag (r->i_pc (r-pc (r), r-prog-segment (r))), n),
icompile (r-prog-segment (r)))
= get (n,
icode (get (offset (r-pc (r)),
program-body (definition (area-name (r-pc (r)),
r-prog-segment (r)))),
offset (r-pc (r)),
definition (area-name (r-pc (r)), r-prog-segment (r)))))

```
; This is really a crucial result because it tells us how to get the i-level
; instruction corresponding to a given r level one.  In the future I prefer
; to see get rather than cdr-n, so now that cdr-n has done its job I'm going
; to disable to rewrite rule that introduces it.
```

EVENT: Disable get-car-cdr-n.

```
; In order for the above lemma to be used, R->I_PC must be disabled.
; We need the following about it:
```

THEOREM: adp-name-untag-r->i_pc
adp-name (untag (r->i_pc (r-pc (r), r-prog-segment (r))))
= adp-name (untag (r-pc (r)))

EVENT: Disable r->i_pc.

THEOREM: definedp-icompile
definedp (*name*, icompile (*list*)) = definedp (*name*, *list*)

THEOREM: length-generate-prelude1
length (generate-prelude1 (*temp-var-dcls*)) = (length (*temp-var-dcls*) $*$ 2)

THEOREM: length-generate-prelude2
length (generate-prelude2 (*formal-vars*)) = length (*formal-vars*)

THEOREM: length-generate-prelude
$$\text{length}\,(\text{generate-prelude}\,(\mathit{program})) = (2 + (\text{length}\,(\text{temp-var-dcls}\,(\mathit{program})) * 2) + \text{length}\,(\text{formal-vars}\,(\mathit{program})))$$

THEOREM: length-generate-postlude
$$\text{length}\,(\text{generate-postlude}\,(\mathit{prog})) = 3$$

THEOREM: find-label-cdr-icompile-program
$$\begin{aligned}
&(((1 + \text{adp-offset}\,(\mathit{lab})) < \text{length}\,(\text{program-body}\,(\mathit{prog})))\\
&\ \wedge\ \text{r-temp-var-dclsp}\,(\text{temp-var-dcls}\,(\mathit{prog}))\\
&\ \wedge\ (\text{adp-offset}\,(\mathit{lab}) \in \mathbf{N})\\
&\ \wedge\ \text{r-program-bodyp}\,(\text{program-body}\,(\mathit{prog}))\\
&\ \wedge\ (\text{adp-name}\,(\mathit{lab}) = \text{name}\,(\mathit{prog}))\\
&\ \wedge\ \text{listp}\,(\mathit{lab}))\\
&\to\ ((1 + \text{find-label}\,(\mathit{lab},\ \text{cdr}\,(\text{icompile-program}\,(\mathit{prog}))))\\
&\qquad < \text{length}\,(\text{cdr}\,(\text{icompile-program}\,(\mathit{prog}))))
\end{aligned}$$

EVENT: Disable length-generate-prelude.

THEOREM: addp-add-adp-untag-r->i_pc
$$\begin{aligned}
&(\text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})\\
&\ \wedge\ (\text{length}\,(\text{icode}\,(\text{get}\,(\text{offset}\,(\text{r-pc}\,(r)),\\
&\qquad\qquad \text{program-body}\,(\text{definition}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))),\\
&\qquad\qquad\qquad \text{r-prog-segment}\,(r)))),\\
&\qquad \text{offset}\,(\text{r-pc}\,(r)),\\
&\qquad \text{definition}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))),\ \text{r-prog-segment}\,(r))))\\
&\qquad \not< n))\\
&\to\ \text{adpp}\,(\text{add-adp}\,(\text{untag}\,(\text{r->i_pc}\,(\text{r-pc}\,(r),\ \text{r-prog-segment}\,(r))),\ n),\\
&\qquad \text{icompile}\,(\text{r-prog-segment}\,(r)))
\end{aligned}$$

```
; The following disable is necessary because adpp is used in the concl above.
; This could have been done earlier, when we disabled the other adp fns.
; If it is moved, however, we must remember to enable it for the proof above.
```

EVENT: Disable adpp.

```
; The following lemma establishes that every label of the form
; (cons name n) where name is less than the length of the program
; occurs in the icompiled program.  It follows from our
; FIND-LABEL-ICOMPILE-PROGRAM-BODY but we wish to have that
```

```
; rewrite rule disabled so as not to rewrite FIND-LABEL
; expressions in the future.
```

THEOREM: lessp-find-label-icompile-program

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{listp}\,(lab)} \\
\land & (\text{adp-name}\,(lab) = \text{name}\,(program)) \\
\land & (\text{adp-offset}\,(lab) \in \mathbf{N}) \\
\land & (\text{length}\,(\text{program-body}\,(program)) \not< \text{adp-offset}\,(lab)) \\
\land & \text{r-programp}\,(program)) \\
\rightarrow & (\text{find-label}\,(lab,\ \text{cdr}\,(\text{icompile-program}\,(program))) \\
& \quad < \quad \text{length}\,(\text{cdr}\,(\text{icompile-program}\,(program))))
\end{array}
$$

```
; To relieve the hypothesis about r-programp we will need:
```

THEOREM: r-prog-segmentp-contains-r-programp

$(\text{r-prog-segmentp}\,(segment) \land (prog \in segment)) \rightarrow \text{r-programp}\,(prog)$

THEOREM: proper-r-statep-implies-r-programp

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{proper-r-statep}\,(r,\ load\text{-}addr) \land \text{definedp}\,(name,\ \text{r-prog-segment}\,(r)))} \\
\rightarrow & \text{r-programp}\,(\text{assoc}\,(name,\ \text{r-prog-segment}\,(r)))
\end{array}
$$

EVENT: Disable r-programp.

```
; At this point I think we have completely dealt with all issues concerning
; FIND-LABEL and we will disable the lemmas surrounding it.
```

EVENT: Disable find-label-generate-postlude.

EVENT: Disable find-label-generate-prelude.

EVENT: Disable not-find-label-generate-prelude2.

EVENT: Disable find-label-generate-prelude1.

EVENT: Disable find-label-add-adp-1-icompile-program-body.

EVENT: Disable find-label-icompile-program-body.

EVENT: Disable find-label-icompile-program-body-lemma1.

EVENT: Disable find-label-icode.

EVENT: Disable find-label-failure-extends-to-cdr.

EVENT: Disable find-label-icode1.

EVENT: Disable find-label-jump_*-lst.

EVENT: Disable find-label-append.

```
; We have also exhausted our patience with icompile-program and will
; shut it down.
```

THEOREM: name-cons
name (cons (x, y)) = x

THEOREM: name-icompile-program
name (icompile-program (*program*)) = name (*program*)

EVENT: Disable icompile-program.

```
; Now in order for these lemmas mentioning (icode (get & &) ...) to be used
; we have to force icode to expand when we know what the instr opcode is.
; That is what the following theorems do:
```

THEOREM: icode-opener
(car (unlabel (*ins*)) = pack (*xxx*))
$\rightarrow$ (icode (*ins*, *pcn*, *prog*)
= dl-block (cons (name (*prog*), *pcn*),
ins,
case on car (unlabel (*ins*)):
case = *call*
then icode-call (unlabel (*ins*), *pcn*, *prog*)
case = *ret*
then icode-ret (unlabel (*ins*), *pcn*, *prog*)
case = *locn*
then icode-locn (unlabel (*ins*), *pcn*, *prog*)
case = *push-constant*
then icode-push-constant (unlabel (*ins*), *pcn*, *prog*)
case = *push-local*

then icode-push-local (unlabel (*ins*), *pcn*, *prog*)
case = *push-global*
then icode-push-global (unlabel (*ins*), *pcn*, *prog*)
case = *push-ctrl-stk-free-size*
then icode-push-ctrl-stk-free-size (unlabel (*ins*),
pcn,
prog)
case = *push-temp-stk-free-size*
then icode-push-temp-stk-free-size (unlabel (*ins*),
pcn,
prog)
case = *push-temp-stk-index*
then icode-push-temp-stk-index (unlabel (*ins*),
pcn,
prog)
case = *jump-if-temp-stk-full*
then icode-jump-if-temp-stk-full (unlabel (*ins*),
pcn,
prog)
case = *jump-if-temp-stk-empty*
then icode-jump-if-temp-stk-empty (unlabel (*ins*),
pcn,
prog)
case = *pop*
then icode-pop (unlabel (*ins*), *pcn*, *prog*)
case = *pop**
then icode-pop* (unlabel (*ins*), *pcn*, *prog*)
case = *popn*
then icode-popn (unlabel (*ins*), *pcn*, *prog*)
case = *pop-local*
then icode-pop-local (unlabel (*ins*), *pcn*, *prog*)
case = *pop-global*
then icode-pop-global (unlabel (*ins*), *pcn*, *prog*)
case = *pop-locn*
then icode-pop-locn (unlabel (*ins*), *pcn*, *prog*)
case = *pop-call*
then icode-pop-call (unlabel (*ins*), *pcn*, *prog*)
case = *fetch-temp-stk*
then icode-fetch-temp-stk (unlabel (*ins*), *pcn*, *prog*)
case = *deposit-temp-stk*
then icode-deposit-temp-stk (unlabel (*ins*), *pcn*, *prog*)
case = *jump*
then icode-jump (unlabel (*ins*), *pcn*, *prog*)
case = *jump-case*

then icode-jump-case (unlabel (*ins*), *pcn*, *prog*)
case = *pushj*
then icode-pushj (unlabel (*ins*), *pcn*, *prog*)
case = *popj*
then icode-popj (unlabel (*ins*), *pcn*, *prog*)
case = *set-local*
then icode-set-local (unlabel (*ins*), *pcn*, *prog*)
case = *set-global*
then icode-set-global (unlabel (*ins*), *pcn*, *prog*)
case = *test-nat-and-jump*
then icode-test-nat-and-jump (unlabel (*ins*), *pcn*, *prog*)
case = *test-int-and-jump*
then icode-test-int-and-jump (unlabel (*ins*), *pcn*, *prog*)
case = *test-bool-and-jump*
then icode-test-bool-and-jump (unlabel (*ins*),
pcn,
prog)
case = *test-bitv-and-jump*
then icode-test-bitv-and-jump (unlabel (*ins*),
pcn,
prog)
case = *no-op*
then icode-no-op (unlabel (*ins*), *pcn*, *prog*)
case = *add-addr*
then icode-add-addr (unlabel (*ins*), *pcn*, *prog*)
case = *sub-addr*
then icode-sub-addr (unlabel (*ins*), *pcn*, *prog*)
case = *eq*
then icode-eq (unlabel (*ins*), *pcn*, *prog*)
case = *lt-addr*
then icode-lt-addr (unlabel (*ins*), *pcn*, *prog*)
case = *fetch*
then icode-fetch (unlabel (*ins*), *pcn*, *prog*)
case = *deposit*
then icode-deposit (unlabel (*ins*), *pcn*, *prog*)
case = *add-int*
then icode-add-int (unlabel (*ins*), *pcn*, *prog*)
case = *add-int-with-carry*
then icode-add-int-with-carry (unlabel (*ins*),
pcn,
prog)
case = *add1-int*
then icode-add1-int (unlabel (*ins*), *pcn*, *prog*)
case = *sub-int*

then icode-sub-int (unlabel (*ins*), *pcn*, *prog*)
case = *sub-int-with-carry*
then icode-sub-int-with-carry (unlabel (*ins*),
pcn,
prog)
case = *sub1-int*
then icode-sub1-int (unlabel (*ins*), *pcn*, *prog*)
case = *neg-int*
then icode-neg-int (unlabel (*ins*), *pcn*, *prog*)
case = *lt-int*
then icode-lt-int (unlabel (*ins*), *pcn*, *prog*)
case = *int-to-nat*
then icode-int-to-nat (unlabel (*ins*), *pcn*, *prog*)
case = *add-nat*
then icode-add-nat (unlabel (*ins*), *pcn*, *prog*)
case = *add-nat-with-carry*
then icode-add-nat-with-carry (unlabel (*ins*),
pcn,
prog)
case = *add1-nat*
then icode-add1-nat (unlabel (*ins*), *pcn*, *prog*)
case = *sub-nat*
then icode-sub-nat (unlabel (*ins*), *pcn*, *prog*)
case = *sub-nat-with-carry*
then icode-sub-nat-with-carry (unlabel (*ins*),
pcn,
prog)
case = *sub1-nat*
then icode-sub1-nat (unlabel (*ins*), *pcn*, *prog*)
case = *lt-nat*
then icode-lt-nat (unlabel (*ins*), *pcn*, *prog*)
case = *mult2-nat*
then icode-mult2-nat (unlabel (*ins*), *pcn*, *prog*)
case = *mult2-nat-with-carry-out*
then icode-mult2-nat-with-carry-out (unlabel (*ins*),
pcn,
prog)
case = *div2-nat*
then icode-div2-nat (unlabel (*ins*), *pcn*, *prog*)
case = *or-bitv*
then icode-or-bitv (unlabel (*ins*), *pcn*, *prog*)
case = *and-bitv*
then icode-and-bitv (unlabel (*ins*), *pcn*, *prog*)
case = *not-bitv*

then icode-not-bitv (unlabel (*ins*), *pcn*, *prog*)
case = *xor-bitv*
then icode-xor-bitv (unlabel (*ins*), *pcn*, *prog*)
case = *rsh-bitv*
then icode-rsh-bitv (unlabel (*ins*), *pcn*, *prog*)
case = *lsh-bitv*
then icode-lsh-bitv (unlabel (*ins*), *pcn*, *prog*)
case = *or-bool*
then icode-or-bool (unlabel (*ins*), *pcn*, *prog*)
case = *and-bool*
then icode-and-bool (unlabel (*ins*), *pcn*, *prog*)
case = *not-bool*
then icode-not-bool (unlabel (*ins*), *pcn*, *prog*)
otherwise `'((error))` **endcase**))

EVENT: Disable icode.

THEOREM: fetch-adp-add-adp-untag-r->i-0
proper-r-statep (r, *load-addr*)
$\rightarrow$ (fetch-adp (untag (r->i_pc (r-pc (r), r-prog-segment (r))),
icompile (r-prog-segment (r)))
= car (icode (get (offset (r-pc (r)),
program-body (definition (area-name (r-pc (r)),
r-prog-segment (r)))),
offset (r-pc (r)),
definition (area-name (r-pc (r)), r-prog-segment (r)))))

```
; I also want to disable deposit-adp.  But I need to prove the
; fundamental relationships between fetch and deposit first.
```

THEOREM: assoc-put-assoc{gen}
assoc (*u1*, put-assoc (*val*, *u2*, *segment*))
= **if** *u1* = *u2*
then if definedp (*u1*, *segment*) **then** cons (*u1*, *val*)
else f endif
else assoc (*u1*, *segment*) **endif**

THEOREM: get-nlistp
($x \simeq$ **nil**) $\rightarrow$ (get (n, x) = 0)

THEOREM: get-put
(($n \in \mathbf{N}$) $\wedge$ ($k \in \mathbf{N}$))
$\rightarrow$ (get (n, put (*val*, k, *lst*))
= **if** $n = k$ **then** *val*
else get (n, *lst*) **endif**)

THEOREM: put-assoc-put-assoc
put-assoc (*val1*, *name*, put-assoc (*val2*, *name*, *alist*))
= put-assoc (*val1*, *name*, *alist*)

THEOREM: put-put
put (*val1*, *n*, put (*val2*, *n*, *lst*)) = put (*val1*, *n*, *lst*)

THEOREM: adpp-put-assoc
(length (*val*) = length (definiens (*name*, *alist*)))
→ (adpp (*x*, put-assoc (*val*, *name*, *alist*)) = adpp (*x*, *alist*))

EVENT: Enable length-put.

THEOREM: adpp-deposit-adp
adpp (*adp2*, *segment*)
→ (adpp (*adp1*, deposit-adp (*val*, *adp2*, *segment*)) = adpp (*adp1*, *segment*))

THEOREM: fetch-adp-deposit-adp
(adpp (*adp1*, *segment*) ∧ adpp (*adp2*, *segment*))
→ (fetch-adp (*adp1*, deposit-adp (*val*, *adp2*, *segment*))
= **if** *adp1* = *adp2* **then** *val*
else fetch-adp (*adp1*, *segment*) **endif**)

THEOREM: deposit-adp-deposit-adp
deposit-adp (*val1*, *adp*, deposit-adp (*val2*, *adp*, *segment*))
= deposit-adp (*val1*, *adp*, *segment*)

```
; That concludes our first pass at fetch-adp and deposit-adp lemmas.
```

EVENT: Disable deposit-adp.

EVENT: Disable fetch-adp.

THEOREM: unlabel-dl
unlabel (dl (*a*, *b*, *c*)) = *c*

EVENT: Disable dl.

THEOREM: type-r->i_pc
type (r->i_pc (*r-pc*, *r-prog-segment*)) = `'ipc`

THEOREM: cddr-r->i_pc
cddr (r->i_pc (*r-pc*, *r-prog-segment*)) = **nil**

THEOREM: addp-add-adp-untag-r->i_pc-0
proper-r-statep (r, *load-addr*)
$\rightarrow$ adpp (untag (r->i_pc (r-pc (r), r-prog-segment (r))),
icompile (r-prog-segment (r)))

THEOREM: add-adp-add-adp
add-adp (add-adp (adp, i), j) = add-adp (adp, $i + j$)

```
; We are now headed into the block of lemmas that are responsible
; for expanding (i * 3 load-addr) for various 3's.  Very roughly speaking,
; we expand (I * 3 load-addr) to (I-STEP (I-STEP (I-STEP *))).  But there
; are various problems, all more or less having to do with the
; fact that we wish to simplify that nest of I-STEPs from the
; inside out, thus symbolically executing the i-level code.
; So imagine that * above is just any intermediate state produced
; in this process.

; The first problem is we call the "I-STATE-OKP problem."
; The sequence of function calls from I is

; I -> I-STEP -> I-STEP1 -> I-STATE-OKP, I-INS-OKP & I-INS-STEP.

; We will need to prove (I-STATE-OKP * load-addr).  But we have from the
; im level proofs:
```

EVENT: Enable i-state-okp-i-ins-step.

EVENT: Disable i-ins-step.

EVENT: Disable i-ins-okp.

```
#|
(include-book i-m-book "i-m" t nil)

; Now we disable all of the just imported events except for
; i-state-okp-i-ins-step and we also disable i-ins-step and i-ins-okp.
; This inclusion also introduces many function symbols.  Most we
; keep disabled.  But we used PROPERP in a theorem far below (without
; really knowing when it came into the chronology) and then discovered
; when a proof failed that it was introduced here and was disabled.
; We need it enabled.  One way to look at this inclusion is that we
; looked at "i-m" and saw two things we need, i-state-okp-i-ins-step
```

```
; and properp.

(status i-m-book-status
        (set-difference (union (@ *e* add-adp-add-adp)
                               '(i-state-okp-i-ins-step properp))
                        '(i-ins-step i-ins-okp)))

|#
; Thus, on every unfolding of I we get for free the I-STATE-OKP
; if we have it initially (which we do).  But the invariant
; lemma above can't be used if * has been simplified out to
; an I-STATE shell.  So our first move is to replace (I * n load-addr)
; by (I* * n load-addr), where I* is like I but does not have the I-STATE-OKP
; check at all.  We prove that I is I* if the initial state is OKP.
```

DEFINITION:
i* (i, n, *load-addr*)
= **if** $n \simeq 0$ **then** i
elseif i-psw (i) = **'run**
then if i-ins-okp (i-current-instruction (i), i)
then i* (i-ins-step (i-current-instruction (i), i, *load-addr*),
$n - 1$,
load-addr)
else i-halt (i,
x-y-error-msg (**'i**,
car (i-current-instruction (i)))) **endif**
else i **endif**

```
; A little lemma to deal with the fact that I* halts on the first non-RUN
; psw while I chugs onward.
```

THEOREM: i-id-on-non-run
(i-psw (i) $\neq$ **'run**) $\rightarrow$ (i (i, n, *load-addr*) = i)

```
; The solution to the I-STATE-OKP problem:
```

THEOREM: i-state-okp-implies-i-is-i*
((*load-addr* $\in$ **N**) $\wedge$ i-state-okp (i, *load-addr*))
$\rightarrow$ (i (i, n, *load-addr*) = i* (i, n, *load-addr*))

EVENT: Disable i.

```
; So now (I s n load-addr) stays in the problem until we can show that s is
; okp and then it changes to I*.

; There is only one s for which we need to prove i-state-okp, and
; that is the opened up version of (r->i r):
```

THEOREM: proper-r-statep-implies-i-state-okp-r->i-opened
proper-r-statep (r, *load-addr*)
$\rightarrow$ i-state-okp (i-state (r->i_pc (r-pc (r), r-prog-segment (r)),
r-cfp (r),
r-csp (r),
r-tsp (r),
r-x (r),
r-y (r),
r-c-flg (r),
r-v-flg (r),
r-n-flg (r),
r-z-flg (r),
icompile (r-prog-segment (r)),
r-usr-data-segment (r),
r-sys-data-segment (r),
r-word-size (r),
`'run`),
load-addr)

EVENT: Disable i-state-okp.

```
; Aside:  We will want the elegant version later so we prove it now:
```

THEOREM: proper-r-statep-implies-i-state-okp-r->i
(proper-r-statep (r, *load-addr*) $\land$ ($\neg$ errorp (r-psw (r))))
$\rightarrow$ i-state-okp (r->i (r), *load-addr*)

```
; Next we want to expand (I* s (ADD1 n) load-addr).  But we want this to
; happen only if s has been reduced to an explicit state.  That way
; we avoid a proliferation of case splits.
```

THEOREM: i*-opener
(*ins* = i-current-instruction (i-state (*i-pc*,
i-cfp,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}))) \\
\to \quad (\mathrm{i}^{*}\,(\text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}), \\
\qquad\quad 1 + n, \\
\qquad\quad \mathit{load\text{-}addr}) \\
\quad = \quad \textbf{if } \mathit{i\text{-}psw} = \texttt{'run} \\
\qquad\quad\; \textbf{then if } \text{i-ins-okp}\,(\mathit{ins}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{i\text{-}usr\text{-}data\text{-}segment},
\end{array}
$$

```
                                i-sys-data-segment,
                                i-word-size,
                                i-psw))
        then i* (i-ins-step (ins,
                             i-state (i-pc,
                                      i-cfp,
                                      i-csp,
                                      i-tsp,
                                      i-x,
                                      i-y,
                                      i-c-flg,
                                      i-v-flg,
                                      i-n-flg,
                                      i-z-flg,
                                      i-prog-segment,
                                      i-usr-data-segment,
                                      i-sys-data-segment,
                                      i-word-size,
                                      i-psw),
                             load-addr),
                 n,
                 load-addr)
        else i-state (i-pc,
                      i-cfp,
                      i-csp,
                      i-tsp,
                      i-x,
                      i-y,
                      i-c-flg,
                      i-v-flg,
                      i-n-flg,
                      i-z-flg,
                      i-prog-segment,
                      i-usr-data-segment,
                      i-sys-data-segment,
                      i-word-size,
                      x-y-error-msg ('i, car (ins))) endif
else i-state (i-pc,
              i-cfp,
              i-csp,
              i-tsp,
              i-x,
              i-y,
              i-c-flg,
```

i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw) **endif**)

THEOREM: i*-done
i* (*i*, 0, *load-addr*) = *i*

EVENT: Disable i*.

```
; So now the only way (i* i n load-addr) will open is if n is 0 or n is an ADD1
; and i is an explicit state.

; Next we consider the problem of opening up I-INS-OKP and I-INS-STEP.
; We wish to open these fns only when the instruction is known and
; the state is explicit.

; Because the i machine has instructions that are all constants --
; there are no operands (unlike the r-level ICODE-OPENER case) --
; we just prove a set of unconditional equalities
```

THEOREM: i-ins-okp-opener
(i-ins-okp ('(add_<c>_x_x{n}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-add_<c>_x_x{n}-okp (i-state (*i-pc*,

i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(add_<tsp>_<tsp>{v})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-add_<tsp>_<tsp>{v}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,

$$
\begin{array}{ll}
& \qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}))) \\
\wedge & (\text{i-ins-okp}\,(\texttt{'(add\_}{<}\texttt{tsp}{>}\texttt{\_}{<}\texttt{tsp}{>}\{\texttt{n}\}\texttt{)}, \\
& \quad \text{i-state}\,(\mathit{i\text{-}pc}, \\
& \qquad \mathit{i\text{-}cfp}, \\
& \qquad \mathit{i\text{-}csp}, \\
& \qquad \mathit{i\text{-}tsp}, \\
& \qquad \mathit{i\text{-}x}, \\
& \qquad \mathit{i\text{-}y}, \\
& \qquad \mathit{i\text{-}c\text{-}flg}, \\
& \qquad \mathit{i\text{-}v\text{-}flg}, \\
& \qquad \mathit{i\text{-}n\text{-}flg}, \\
& \qquad \mathit{i\text{-}z\text{-}flg}, \\
& \qquad \mathit{i\text{-}prog\text{-}segment}, \\
& \qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
& \qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
& \qquad \mathit{i\text{-}word\text{-}size}, \\
& \qquad \mathit{i\text{-}psw})) \\
& = \ \text{i-add\_}{<}\text{tsp}{>}\text{\_}{<}\text{tsp}{>}\{\text{n}\}\text{-okp}\,(\text{i-state}\,(\mathit{i\text{-}pc}, \\
& \qquad\qquad \mathit{i\text{-}cfp}, \\
& \qquad\qquad \mathit{i\text{-}csp}, \\
& \qquad\qquad \mathit{i\text{-}tsp}, \\
& \qquad\qquad \mathit{i\text{-}x}, \\
& \qquad\qquad \mathit{i\text{-}y}, \\
& \qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
& \qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
& \qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
& \qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
& \qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
& \qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
& \qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
& \qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
& \qquad\qquad \mathit{i\text{-}psw}))) \\
\wedge & (\text{i-ins-okp}\,(\texttt{'(add\_}{<}\texttt{tsp}{>}\{\texttt{a}\}\texttt{\_x}\{\texttt{n}\}\texttt{)}, \\
& \quad \text{i-state}\,(\mathit{i\text{-}pc}, \\
& \qquad \mathit{i\text{-}cfp}, \\
& \qquad \mathit{i\text{-}csp}, \\
& \qquad \mathit{i\text{-}tsp}, \\
& \qquad \mathit{i\text{-}x}, \\
& \qquad \mathit{i\text{-}y}, \\
& \qquad \mathit{i\text{-}c\text{-}flg}, \\
& \qquad \mathit{i\text{-}v\text{-}flg}, \\
& \qquad \mathit{i\text{-}n\text{-}flg}, \\
& \qquad \mathit{i\text{-}z\text{-}flg}, \\
& \qquad \mathit{i\text{-}prog\text{-}segment},
\end{array}
$$

$$
\begin{array}{lll}
 & & \quad\quad\quad\quad\quad\quad \textit{i-usr-data-segment}, \\
 & & \quad\quad\quad\quad\quad\quad \textit{i-sys-data-segment}, \\
 & & \quad\quad\quad\quad\quad\quad \textit{i-word-size}, \\
 & & \quad\quad\quad\quad\quad\quad \textit{i-psw})) \\
 & = & \text{i-add\_<tsp>\{a\}\_x\{n\}-okp}\,(\text{i-state}\,(\textit{i-pc}, \textit{i-cfp}, \textit{i-csp}, \textit{i-tsp}, \textit{i-x}, \textit{i-y}, \textit{i-c-flg}, \textit{i-v-flg}, \textit{i-n-flg}, \textit{i-z-flg}, \\
 & & \quad \textit{i-prog-segment}, \textit{i-usr-data-segment}, \textit{i-sys-data-segment}, \textit{i-word-size}, \textit{i-psw}))) \\
\wedge & & (\text{i-ins-okp}\,(\texttt{'(add\_tsp\_*}\{\texttt{n}\}\texttt{)}, \\
 & & \quad \text{i-state}\,(\textit{i-pc}, \textit{i-cfp}, \textit{i-csp}, \textit{i-tsp}, \textit{i-x}, \textit{i-y}, \textit{i-c-flg}, \textit{i-v-flg}, \textit{i-n-flg}, \textit{i-z-flg}, \\
 & & \quad\quad \textit{i-prog-segment}, \textit{i-usr-data-segment}, \textit{i-sys-data-segment}, \textit{i-word-size}, \textit{i-psw})) \\
 & = & \text{i-add\_tsp\_*\{n\}-okp}\,(\text{i-state}\,(\textit{i-pc}, \textit{i-cfp}, \textit{i-csp}, \textit{i-tsp}, \textit{i-x}, \textit{i-y}, \textit{i-c-flg}, \textit{i-v-flg}, \textit{i-n-flg},
\end{array}
$$

```
                                    i-z-flg,
                                    i-prog-segment,
                                    i-usr-data-segment,
                                    i-sys-data-segment,
                                    i-word-size,
                                    i-psw)))
∧   (i-ins-okp ('(add_tsp_x{n}),
           i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw))
     =   i-add_tsp_x{n}-okp (i-state (i-pc,
                                    i-cfp,
                                    i-csp,
                                    i-tsp,
                                    i-x,
                                    i-y,
                                    i-c-flg,
                                    i-v-flg,
                                    i-n-flg,
                                    i-z-flg,
                                    i-prog-segment,
                                    i-usr-data-segment,
                                    i-sys-data-segment,
                                    i-word-size,
                                    i-psw)))
∧   (i-ins-okp ('(add_<tsp>{i}_x{i}),
           i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
```

$$
\begin{array}{lll}
 & & \qquad\qquad \textit{i-c-flg}, \\
 & & \qquad\qquad \textit{i-v-flg}, \\
 & & \qquad\qquad \textit{i-n-flg}, \\
 & & \qquad\qquad \textit{i-z-flg}, \\
 & & \qquad\qquad \textit{i-prog-segment}, \\
 & & \qquad\qquad \textit{i-usr-data-segment}, \\
 & & \qquad\qquad \textit{i-sys-data-segment}, \\
 & & \qquad\qquad \textit{i-word-size}, \\
 & & \qquad\qquad \textit{i-psw})) \\
 & = & \text{i-add\_<tsp>\{i\}\_x\{i\}-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\wedge & \multicolumn{2}{l}{(\text{i-ins-okp}\,(\texttt{'(add\_<tsp>\{n\}\_x\{n\})},} \\
 & & \quad \text{i-state}\,(\textit{i-pc}, \\
 & & \qquad\qquad \textit{i-cfp}, \\
 & & \qquad\qquad \textit{i-csp}, \\
 & & \qquad\qquad \textit{i-tsp}, \\
 & & \qquad\qquad \textit{i-x}, \\
 & & \qquad\qquad \textit{i-y}, \\
 & & \qquad\qquad \textit{i-c-flg}, \\
 & & \qquad\qquad \textit{i-v-flg}, \\
 & & \qquad\qquad \textit{i-n-flg}, \\
 & & \qquad\qquad \textit{i-z-flg}, \\
 & & \qquad\qquad \textit{i-prog-segment}, \\
 & & \qquad\qquad \textit{i-usr-data-segment}, \\
 & & \qquad\qquad \textit{i-sys-data-segment}, \\
 & & \qquad\qquad \textit{i-word-size}, \\
 & & \qquad\qquad \textit{i-psw})) \\
 & = & \text{i-add\_<tsp>\{n\}\_x\{n\}-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp},
\end{array}
$$

$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\wedge$ (i-ins-okp (`'(add_pc_x{n})`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-add_pc_x{n}-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\wedge$ (i-ins-okp (`'(add_x_x{n})`,
i-state ($i\text{-}pc$,

i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-add_x_x{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(add_x{n}_csp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,

i-psw))
= i-add_x{n}_csp-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(addc_<c>_x{n}_y{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-addc_<c>_x{n}_y{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,

```
                                                   i-sys-data-segment,
                                                   i-word-size,
                                                   i-psw)))
∧   (i-ins-okp ('(addc_<v>_x{i}_y{i}),
                i-state (i-pc,
                         i-cfp,
                         i-csp,
                         i-tsp,
                         i-x,
                         i-y,
                         i-c-flg,
                         i-v-flg,
                         i-n-flg,
                         i-z-flg,
                         i-prog-segment,
                         i-usr-data-segment,
                         i-sys-data-segment,
                         i-word-size,
                         i-psw))
     =   i-addc_<v>_x{i}_y{i}-okp (i-state (i-pc,
                                            i-cfp,
                                            i-csp,
                                            i-tsp,
                                            i-x,
                                            i-y,
                                            i-c-flg,
                                            i-v-flg,
                                            i-n-flg,
                                            i-z-flg,
                                            i-prog-segment,
                                            i-usr-data-segment,
                                            i-sys-data-segment,
                                            i-word-size,
                                            i-psw)))
∧   (i-ins-okp ('(and_<tsp>{v}_x{v}),
                i-state (i-pc,
                         i-cfp,
                         i-csp,
                         i-tsp,
                         i-x,
                         i-y,
                         i-c-flg,
                         i-v-flg,
                         i-n-flg,
```

```
                 i-z-flg,
                 i-prog-segment,
                 i-usr-data-segment,
                 i-sys-data-segment,
                 i-word-size,
                 i-psw))
   =   i-and_<tsp>{v}_x{v}-okp (i-state (i-pc,
                                         i-cfp,
                                         i-csp,
                                         i-tsp,
                                         i-x,
                                         i-y,
                                         i-c-flg,
                                         i-v-flg,
                                         i-n-flg,
                                         i-z-flg,
                                         i-prog-segment,
                                         i-usr-data-segment,
                                         i-sys-data-segment,
                                         i-word-size,
                                         i-psw)))
∧   (i-ins-okp ('(and_<tsp>{b}_x{b}),
            i-state (i-pc,
                     i-cfp,
                     i-csp,
                     i-tsp,
                     i-x,
                     i-y,
                     i-c-flg,
                     i-v-flg,
                     i-n-flg,
                     i-z-flg,
                     i-prog-segment,
                     i-usr-data-segment,
                     i-sys-data-segment,
                     i-word-size,
                     i-psw))
   =   i-and_<tsp>{b}_x{b}-okp (i-state (i-pc,
                                         i-cfp,
                                         i-csp,
                                         i-tsp,
                                         i-x,
                                         i-y,
                                         i-c-flg,
```

i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(asr_<c>_<tsp>_<tsp>{b})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-asr_<c>_<tsp>_<tsp>{b}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(cpop_cfp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,

$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-cpop_cfp-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-okp (`'(cpop_pc)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-cpop_pc-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,

i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(cpush_*)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-cpush_*-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp ('(cpush_<tsp>+),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-cpush_<tsp>+-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp ('(cpush_cfp),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,

i-sys-data-segment,
i-word-size,
i-psw))
= i-cpush_cfp-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(decr_<tsp>_<tsp>{i})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-decr_<tsp>_<tsp>{i}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,

i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp ('**(decr_<tsp>_<tsp>{n})**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-decr_<tsp>_<tsp>{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp ('**(incr_<tsp>_<tsp>{i})**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,

i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-incr_<tsp>_<tsp>{i}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(incr_<tsp>_<tsp>{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-incr_<tsp>_<tsp>{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-okp (`'(incr_y_y{n})`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-incr_y_y{n}-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-okp (`'(int-to-nat)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}word\text{-}size, \\
\qquad\qquad\qquad\qquad i\text{-}psw)) \\
\quad = \quad \text{i-int-to-nat-okp}\,(\text{i-state}\,(i\text{-}pc, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}word\text{-}size, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}psw))) \\
\land \quad (\text{i-ins-okp}\,(\texttt{'(jump-n\_x)}, \\
\qquad\qquad\quad \text{i-state}\,(i\text{-}pc, \\
\qquad\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}word\text{-}size, \\
\qquad\qquad\qquad\qquad i\text{-}psw))
\end{array}
$$

$$
\begin{array}{ll}
= & \text{i-jump-n\_x-okp}\,(\text{i-state}\,(\mathit{i\text{-}pc}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}psw}))) \\
\wedge & (\text{i-ins-okp}\,(\texttt{'(jump-nn\_x)}, \\
 & \qquad \text{i-state}\,(\mathit{i\text{-}pc}, \\
 & \qquad\qquad \mathit{i\text{-}cfp}, \\
 & \qquad\qquad \mathit{i\text{-}csp}, \\
 & \qquad\qquad \mathit{i\text{-}tsp}, \\
 & \qquad\qquad \mathit{i\text{-}x}, \\
 & \qquad\qquad \mathit{i\text{-}y}, \\
 & \qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
 & \qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
 & \qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
 & \qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
 & \qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
 & \qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
 & \qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
 & \qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
 & \qquad\qquad \mathit{i\text{-}psw})) \\
= & \text{i-jump-nn\_x-okp}\,(\text{i-state}\,(\mathit{i\text{-}pc}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
 & \qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment},
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-okp}\,(\texttt{'(jump-nz\_x)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw})) \\
\qquad = \quad \text{i-jump-nz\_x-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-okp}\,(\texttt{'(jump-z\_x)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg},
\end{array}
$$

$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-jump-z_x-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\wedge$ (i-ins-okp (`'(jump_*)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-jump_*-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-okp}\,(\texttt{'(jump\_x}\{\texttt{subr}\}\texttt{)}, \\
\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad \textit{i-psw})) \\
\quad = \quad \text{i-jump\_x}\{\text{subr}\}\text{-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-okp}\,(\texttt{'(lsr\_<c>\_x\_x}\{\texttt{n}\}\texttt{)}, \\
\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad \textit{i-x},
\end{array}
$$

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-lsr_<c>_x_x{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(lsr_<tsp>_<tsp>{v})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-lsr_<tsp>_<tsp>{v}-okp (i-state (*i-pc*,
i-cfp,
i-csp,

$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\land$ (i-ins-okp (`'(move-c_<tsp>_*)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-move-c_<tsp>_*-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\land$ (i-ins-okp (`'(move-v_<tsp>_*)`,

i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-move-v_<tsp>_*-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(move-z_<tsp>_*)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,

i-word-size,
i-psw))

= i-move-z_<tsp>_*-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(move-n_x_*)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-move-n_x_*-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,

i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp ('(move_<tsp>_*),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-move_<tsp>_*-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp ('(move_<x{a}>_<tsp>),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,

i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-move_<x{a}>_<tsp>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(move_<x{s}>_<tsp>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-move_<x{s}>_<tsp>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw})))\\
\land \quad (\text{i-ins-okp}\,(\texttt{'(move\_cfp\_csp)},\\
\qquad\qquad \text{i-state}\,(\textit{i-pc},\\
\qquad\qquad\qquad \textit{i-cfp},\\
\qquad\qquad\qquad \textit{i-csp},\\
\qquad\qquad\qquad \textit{i-tsp},\\
\qquad\qquad\qquad \textit{i-x},\\
\qquad\qquad\qquad \textit{i-y},\\
\qquad\qquad\qquad \textit{i-c-flg},\\
\qquad\qquad\qquad \textit{i-v-flg},\\
\qquad\qquad\qquad \textit{i-n-flg},\\
\qquad\qquad\qquad \textit{i-z-flg},\\
\qquad\qquad\qquad \textit{i-prog-segment},\\
\qquad\qquad\qquad \textit{i-usr-data-segment},\\
\qquad\qquad\qquad \textit{i-sys-data-segment},\\
\qquad\qquad\qquad \textit{i-word-size},\\
\qquad\qquad\qquad \textit{i-psw}))\\
\quad = \quad \text{i-move\_cfp\_csp-okp}\,(\text{i-state}\,(\textit{i-pc},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-cfp},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-csp},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-tsp},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-x},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-y},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-word-size},\\
\qquad\qquad\qquad\qquad\qquad \textit{i-psw})))\\
\land \quad (\text{i-ins-okp}\,(\texttt{'(move\_csp\_cfp)},\\
\qquad\qquad \text{i-state}\,(\textit{i-pc},\\
\qquad\qquad\qquad \textit{i-cfp},\\
\qquad\qquad\qquad \textit{i-csp},
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw})) \\
\quad = \quad \text{i-move\_csp\_cfp-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-okp}\,(\texttt{'(move\_x\_*)}, \\
\qquad\qquad\quad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw})) \\
\quad = \quad \text{i-move\_x\_*-okp}\,(\text{i-state}\,(\textit{i-pc},
\end{array}
$$

i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(move_x_<x{s}>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-move_x_<x{s}>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,

$i\text{-}psw)))$

$\wedge$ (i-ins-okp (`'(move_x_tsp)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw))$

= i-move_x_tsp-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw)))$

$\wedge$ (i-ins-okp (`'(move_x_x)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,

```
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw))
    =  i-move_x_x-okp (i-state (i-pc,
                                i-cfp,
                                i-csp,
                                i-tsp,
                                i-x,
                                i-y,
                                i-c-flg,
                                i-v-flg,
                                i-n-flg,
                                i-z-flg,
                                i-prog-segment,
                                i-usr-data-segment,
                                i-sys-data-segment,
                                i-word-size,
                                i-psw)))
∧  (i-ins-okp ('(move_y_*),
              i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw))
    =  i-move_y_*-okp (i-state (i-pc,
                                i-cfp,
                                i-csp,
                                i-tsp,
                                i-x,
                                i-y,
                                i-c-flg,
                                i-v-flg,
                                i-n-flg,
```

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(move_y_<y{s}>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-move_y_<y{s}>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(move_y_tsp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,

i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-move_y_tsp-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
$\wedge$ (i-ins-okp (`'(neg_<tsp>_<tsp>{i})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-neg_<tsp>_<tsp>{i}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,

i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(not_<tsp>_<tsp>{v})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-not_<tsp>_<tsp>{v}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(or_<tsp>{v}_x{v})`,
i-state (*i-pc*,

i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-or_<tsp>{v}_x{v}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(or_<tsp>{b}_x{b})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,

i-psw))

= i-or_<tsp>{b}_x{b}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(sub_<c>_<tsp>{a}_x{a})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-sub_<c>_<tsp>{a}_x{a}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,

i-sys-data-segment,
i-word-size,
i-psw)))

$\wedge$ (i-ins-okp ('(sub_<c>_<tsp>{n}_x{n}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-sub_<c>_<tsp>{n}_x{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

$\wedge$ (i-ins-okp ('(sub_<nv>_<tsp>{i}_x{i}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-sub_<nv>_<tsp>{i}_x{i}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(sub_<tsp>{a}_x{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-sub_<tsp>{a}_x{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,

```
                                          i-v-flg,
                                          i-n-flg,
                                          i-z-flg,
                                          i-prog-segment,
                                          i-usr-data-segment,
                                          i-sys-data-segment,
                                          i-word-size,
                                          i-psw)))
∧  (i-ins-okp ('(sub_x{s}_y{n}),
          i-state (i-pc,
                   i-cfp,
                   i-csp,
                   i-tsp,
                   i-x,
                   i-y,
                   i-c-flg,
                   i-v-flg,
                   i-n-flg,
                   i-z-flg,
                   i-prog-segment,
                   i-usr-data-segment,
                   i-sys-data-segment,
                   i-word-size,
                   i-psw))
    =  i-sub_x{s}_y{n}-okp (i-state (i-pc,
                                     i-cfp,
                                     i-csp,
                                     i-tsp,
                                     i-x,
                                     i-y,
                                     i-c-flg,
                                     i-v-flg,
                                     i-n-flg,
                                     i-z-flg,
                                     i-prog-segment,
                                     i-usr-data-segment,
                                     i-sys-data-segment,
                                     i-word-size,
                                     i-psw)))
∧  (i-ins-okp ('(sub_<tsp>{i}_x{i}),
          i-state (i-pc,
                   i-cfp,
                   i-csp,
                   i-tsp,
```

i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-sub_<tsp>{i}_x{i}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(sub_<tsp>{n}_x{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-sub_<tsp>{n}_x{n}-okp (i-state (*i-pc*,
i-cfp,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad \text{(i-ins-okp}\,(\texttt{'(sub\_}<\texttt{tsp}>\{\texttt{s}\}\texttt{\_x}\{\texttt{s}\}\texttt{)}, \\
\qquad\qquad\quad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\quad \textit{i-cfp}, \\
\qquad\qquad\qquad\quad \textit{i-csp}, \\
\qquad\qquad\qquad\quad \textit{i-tsp}, \\
\qquad\qquad\qquad\quad \textit{i-x}, \\
\qquad\qquad\qquad\quad \textit{i-y}, \\
\qquad\qquad\qquad\quad \textit{i-c-flg}, \\
\qquad\qquad\qquad\quad \textit{i-v-flg}, \\
\qquad\qquad\qquad\quad \textit{i-n-flg}, \\
\qquad\qquad\qquad\quad \textit{i-z-flg}, \\
\qquad\qquad\qquad\quad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\quad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\quad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\quad \textit{i-word-size}, \\
\qquad\qquad\qquad\quad \textit{i-psw})) \\
\quad = \quad \text{i-sub\_}<\text{tsp}>\{\text{s}\}\text{\_x}\{\text{s}\}\text{-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw})))
\end{array}
$$

∧ (i-ins-okp ('(sub_<z>_x{s}_y{s}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-sub_<z>_x{s}_y{s}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp ('(subb_<c>_x{n}_y{n}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,

i-sys-data-segment,
i-word-size,
i-psw))
= i-subb_<c>_x{n}_y{n}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(subb_<v>_x{i}_y{i})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-subb_<v>_x{i}_y{i}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-psw}))) \\
\wedge \quad (\text{i-ins-okp}\,(\texttt{'(tpop\_<c>\_x)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-psw})) \\
\qquad = \quad \text{i-tpop\_<c>\_x-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\wedge \quad (\text{i-ins-okp}\,(\texttt{'(tpop\_<x\{a\}>)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\quad \textit{i-c-flg},
\end{array}
$$

i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-tpop_<x{a}>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(tpop_<x{s}>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-tpop_<x{s}>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

$\wedge$ (i-ins-okp (`'(tpop_pc)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

$=$ i-tpop_pc-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

$\wedge$ (i-ins-okp (`'(tpop_x)`,
i-state (*i-pc*,
i-cfp,

$$
\begin{array}{lll}
 & & \textit{i-csp}, \\
 & & \textit{i-tsp}, \\
 & & \textit{i-x}, \\
 & & \textit{i-y}, \\
 & & \textit{i-c-flg}, \\
 & & \textit{i-v-flg}, \\
 & & \textit{i-n-flg}, \\
 & & \textit{i-z-flg}, \\
 & & \textit{i-prog-segment}, \\
 & & \textit{i-usr-data-segment}, \\
 & & \textit{i-sys-data-segment}, \\
 & & \textit{i-word-size}, \\
 & & \textit{i-psw})) \\
 & = & \text{i-tpop\_x-okp}\,(\text{i-state}\,(\textit{i-pc}, \\
 & & \qquad \textit{i-cfp}, \\
 & & \qquad \textit{i-csp}, \\
 & & \qquad \textit{i-tsp}, \\
 & & \qquad \textit{i-x}, \\
 & & \qquad \textit{i-y}, \\
 & & \qquad \textit{i-c-flg}, \\
 & & \qquad \textit{i-v-flg}, \\
 & & \qquad \textit{i-n-flg}, \\
 & & \qquad \textit{i-z-flg}, \\
 & & \qquad \textit{i-prog-segment}, \\
 & & \qquad \textit{i-usr-data-segment}, \\
 & & \qquad \textit{i-sys-data-segment}, \\
 & & \qquad \textit{i-word-size}, \\
 & & \qquad \textit{i-psw}))) \\
\land & & (\text{i-ins-okp}\,(\texttt{'(tpop\_y)}, \\
 & & \quad \text{i-state}\,(\textit{i-pc}, \\
 & & \qquad \textit{i-cfp}, \\
 & & \qquad \textit{i-csp}, \\
 & & \qquad \textit{i-tsp}, \\
 & & \qquad \textit{i-x}, \\
 & & \qquad \textit{i-y}, \\
 & & \qquad \textit{i-c-flg}, \\
 & & \qquad \textit{i-v-flg}, \\
 & & \qquad \textit{i-n-flg}, \\
 & & \qquad \textit{i-z-flg}, \\
 & & \qquad \textit{i-prog-segment}, \\
 & & \qquad \textit{i-usr-data-segment}, \\
 & & \qquad \textit{i-sys-data-segment}, \\
 & & \qquad \textit{i-word-size}, \\
 & & \qquad \textit{i-psw}))
\end{array}
$$

$$
\begin{array}{lll}
= & \text{i-tpop\_y-okp}\,(\text{i-state}\,(& i\text{-}pc, \\
 & & i\text{-}cfp, \\
 & & i\text{-}csp, \\
 & & i\text{-}tsp, \\
 & & i\text{-}x, \\
 & & i\text{-}y, \\
 & & i\text{-}c\text{-}flg, \\
 & & i\text{-}v\text{-}flg, \\
 & & i\text{-}n\text{-}flg, \\
 & & i\text{-}z\text{-}flg, \\
 & & i\text{-}prog\text{-}segment, \\
 & & i\text{-}usr\text{-}data\text{-}segment, \\
 & & i\text{-}sys\text{-}data\text{-}segment, \\
 & & i\text{-}word\text{-}size, \\
 & & i\text{-}psw)))
\end{array}
$$

$$
\begin{array}{lll}
\wedge & \multicolumn{2}{l}{(\text{i-ins-okp}\,(\texttt{'(tpop\{v\}\_<z>\_y)},} \\
 & \quad\quad \text{i-state}\,(& i\text{-}pc, \\
 & & i\text{-}cfp, \\
 & & i\text{-}csp, \\
 & & i\text{-}tsp, \\
 & & i\text{-}x, \\
 & & i\text{-}y, \\
 & & i\text{-}c\text{-}flg, \\
 & & i\text{-}v\text{-}flg, \\
 & & i\text{-}n\text{-}flg, \\
 & & i\text{-}z\text{-}flg, \\
 & & i\text{-}prog\text{-}segment, \\
 & & i\text{-}usr\text{-}data\text{-}segment, \\
 & & i\text{-}sys\text{-}data\text{-}segment, \\
 & & i\text{-}word\text{-}size, \\
 & & i\text{-}psw))
\end{array}
$$

$$
\begin{array}{lll}
= & \text{i-tpop\{v\}\_<z>\_y-okp}\,(\text{i-state}\,(& i\text{-}pc, \\
 & & i\text{-}cfp, \\
 & & i\text{-}csp, \\
 & & i\text{-}tsp, \\
 & & i\text{-}x, \\
 & & i\text{-}y, \\
 & & i\text{-}c\text{-}flg, \\
 & & i\text{-}v\text{-}flg, \\
 & & i\text{-}n\text{-}flg, \\
 & & i\text{-}z\text{-}flg, \\
 & & i\text{-}prog\text{-}segment, \\
 & & i\text{-}usr\text{-}data\text{-}segment, \\
 & & i\text{-}sys\text{-}data\text{-}segment,
\end{array}
$$

i-word-size,
i-psw)))
∧ (i-ins-okp (`'(tpop{b}_<z>_y)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-tpop{b}_<z>_y-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(tpop{i}_<zn>_y)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,

i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-tpop{i}_<zn>_y-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(tpop{n}_<z>_y)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-tpop{n}_<z>_y-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,

$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\land$ (i-ins-okp (`'(tpush_*)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-tpush_*-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\land$ (i-ins-okp (`'(tpush_<x{a}>)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-tpush_<x{a}>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(tpush_<x{s}>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-tpush_<x{s}>-okp (i-state (*i-pc*,
i-cfp,
i-csp,

$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-okp (`'(tpush_csp)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$))

= i-tpush_csp-okp (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-okp (`'(tpush_tsp)`,

i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-tpush_tsp-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (`'(tpush_x)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,

i-word-size,
i-psw))
= i-tpush_x-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-okp (**'(xor_<tsp>_<tsp>)**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))
= i-xor_<tsp>_<tsp>-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,

i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(xor_<tsp>{v}_x{v})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-xor_<tsp>{v}_x{v}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(xor_<tsp>{b}_*{b})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,

i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-xor_<tsp>{b}_*{b}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(xor_<tsp>{b}_x{b})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-xor_<tsp>{b}_x{b}-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,

i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-okp (`'(xor_<z>_<tsp>_x)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw))

= i-xor_<z>_<tsp>_x-okp (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

THEOREM: i-ins-step-opener
(i-ins-step (`'(add_<c>_x_x{n})`,
i-state (*i-pc*,

$$
\begin{aligned}
&\qquad\qquad\qquad \textit{i-cfp}, \\
&\qquad\qquad\qquad \textit{i-csp}, \\
&\qquad\qquad\qquad \textit{i-tsp}, \\
&\qquad\qquad\qquad \textit{i-x}, \\
&\qquad\qquad\qquad \textit{i-y}, \\
&\qquad\qquad\qquad \textit{i-c-flg}, \\
&\qquad\qquad\qquad \textit{i-v-flg}, \\
&\qquad\qquad\qquad \textit{i-n-flg}, \\
&\qquad\qquad\qquad \textit{i-z-flg}, \\
&\qquad\qquad\qquad \textit{i-prog-segment}, \\
&\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
&\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
&\qquad\qquad\qquad \textit{i-word-size}, \\
&\qquad\qquad\qquad \textit{i-psw}), \\
&\qquad\qquad \textit{load-addr}) \\
= \quad & \text{i-add\_$<$c$>$\_x\_x\{n\}-step}\,(\text{i-state}\,(\textit{i-pc}, \\
&\qquad \textit{i-cfp}, \\
&\qquad \textit{i-csp}, \\
&\qquad \textit{i-tsp}, \\
&\qquad \textit{i-x}, \\
&\qquad \textit{i-y}, \\
&\qquad \textit{i-c-flg}, \\
&\qquad \textit{i-v-flg}, \\
&\qquad \textit{i-n-flg}, \\
&\qquad \textit{i-z-flg}, \\
&\qquad \textit{i-prog-segment}, \\
&\qquad \textit{i-usr-data-segment}, \\
&\qquad \textit{i-sys-data-segment}, \\
&\qquad \textit{i-word-size}, \\
&\qquad \textit{i-psw})) \\
\wedge \quad & (\text{i-ins-step}\,(\texttt{'(add\_$<$tsp$>$\_$<$tsp$>$\{v\})}, \\
&\qquad \text{i-state}\,(\textit{i-pc}, \\
&\qquad\qquad \textit{i-cfp}, \\
&\qquad\qquad \textit{i-csp}, \\
&\qquad\qquad \textit{i-tsp}, \\
&\qquad\qquad \textit{i-x}, \\
&\qquad\qquad \textit{i-y}, \\
&\qquad\qquad \textit{i-c-flg}, \\
&\qquad\qquad \textit{i-v-flg}, \\
&\qquad\qquad \textit{i-n-flg}, \\
&\qquad\qquad \textit{i-z-flg}, \\
&\qquad\qquad \textit{i-prog-segment}, \\
&\qquad\qquad \textit{i-usr-data-segment}, \\
&\qquad\qquad \textit{i-sys-data-segment},
\end{aligned}
$$

i-word-size,
i-psw),
load-addr)
= i-add_<tsp>_<tsp>{v}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(add_<tsp>_<tsp>{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-add_<tsp>_<tsp>{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

```
                                             i-z-flg,
                                             i-prog-segment,
                                             i-usr-data-segment,
                                             i-sys-data-segment,
                                             i-word-size,
                                             i-psw)))
∧   (i-ins-step ('(add_<tsp>{a}_x{n}),
                i-state (i-pc,
                         i-cfp,
                         i-csp,
                         i-tsp,
                         i-x,
                         i-y,
                         i-c-flg,
                         i-v-flg,
                         i-n-flg,
                         i-z-flg,
                         i-prog-segment,
                         i-usr-data-segment,
                         i-sys-data-segment,
                         i-word-size,
                         i-psw),
                load-addr)
     =   i-add_<tsp>{a}_x{n}-step (i-state (i-pc,
                                            i-cfp,
                                            i-csp,
                                            i-tsp,
                                            i-x,
                                            i-y,
                                            i-c-flg,
                                            i-v-flg,
                                            i-n-flg,
                                            i-z-flg,
                                            i-prog-segment,
                                            i-usr-data-segment,
                                            i-sys-data-segment,
                                            i-word-size,
                                            i-psw)))
∧   (i-ins-step ('(add_tsp_*{n}),
                i-state (i-pc,
                         i-cfp,
                         i-csp,
                         i-tsp,
                         i-x,
```

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-add_tsp_*{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(add_tsp_x{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-add_tsp_x{n}-step (i-state (*i-pc*,

```
                                    i-cfp,
                                    i-csp,
                                    i-tsp,
                                    i-x,
                                    i-y,
                                    i-c-flg,
                                    i-v-flg,
                                    i-n-flg,
                                    i-z-flg,
                                    i-prog-segment,
                                    i-usr-data-segment,
                                    i-sys-data-segment,
                                    i-word-size,
                                    i-psw)))
∧   (i-ins-step ('(add_<tsp>{i}_x{i}),
             i-state (i-pc,
                      i-cfp,
                      i-csp,
                      i-tsp,
                      i-x,
                      i-y,
                      i-c-flg,
                      i-v-flg,
                      i-n-flg,
                      i-z-flg,
                      i-prog-segment,
                      i-usr-data-segment,
                      i-sys-data-segment,
                      i-word-size,
                      i-psw),
             load-addr)
  =   i-add_<tsp>{i}_x{i}-step (i-state (i-pc,
                                         i-cfp,
                                         i-csp,
                                         i-tsp,
                                         i-x,
                                         i-y,
                                         i-c-flg,
                                         i-v-flg,
                                         i-n-flg,
                                         i-z-flg,
                                         i-prog-segment,
                                         i-usr-data-segment,
                                         i-sys-data-segment,
```

```
                                    i-word-size,
                                    i-psw)))
∧  (i-ins-step ('(add_<tsp>{n}_x{n}),
               i-state (i-pc,
                        i-cfp,
                        i-csp,
                        i-tsp,
                        i-x,
                        i-y,
                        i-c-flg,
                        i-v-flg,
                        i-n-flg,
                        i-z-flg,
                        i-prog-segment,
                        i-usr-data-segment,
                        i-sys-data-segment,
                        i-word-size,
                        i-psw),
               load-addr)
    =  i-add_<tsp>{n}_x{n}-step (i-state (i-pc,
                                          i-cfp,
                                          i-csp,
                                          i-tsp,
                                          i-x,
                                          i-y,
                                          i-c-flg,
                                          i-v-flg,
                                          i-n-flg,
                                          i-z-flg,
                                          i-prog-segment,
                                          i-usr-data-segment,
                                          i-sys-data-segment,
                                          i-word-size,
                                          i-psw)))
∧  (i-ins-step ('(add_pc_x{n}),
               i-state (i-pc,
                        i-cfp,
                        i-csp,
                        i-tsp,
                        i-x,
                        i-y,
                        i-c-flg,
                        i-v-flg,
                        i-n-flg,
```

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-add_pc_x{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(add_x_x{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-add_x_x{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(add_x{n}_csp)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)

= i-add_x{n}_csp-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(addc_<c>_x{n}_y{n})`,
i-state ($i\text{-}pc$,

$$
\begin{array}{l}
\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment, \\
\qquad\qquad\qquad i\text{-}word\text{-}size, \\
\qquad\qquad\qquad i\text{-}psw), \\
\qquad\qquad load\text{-}addr) \\
\quad = \quad \text{i-addc\_<c>\_x\{n\}\_y\{n\}-step}\,(\text{i-state}\,(i\text{-}pc, i\text{-}cfp, i\text{-}csp, i\text{-}tsp, i\text{-}x, i\text{-}y, i\text{-}c\text{-}flg, i\text{-}v\text{-}flg, i\text{-}n\text{-}flg, i\text{-}z\text{-}flg, \\
\qquad\qquad i\text{-}prog\text{-}segment, i\text{-}usr\text{-}data\text{-}segment, i\text{-}sys\text{-}data\text{-}segment, i\text{-}word\text{-}size, i\text{-}psw))) \\
\wedge \quad (\text{i-ins-step}\,(\text{'}\mathtt{(addc\_<v>\_x\{i\}\_y\{i\})}, \\
\qquad\qquad \text{i-state}\,(i\text{-}pc, \\
\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment,
\end{array}
$$

i-word-size,
i-psw),
load-addr)
= i-addc_<v>_x{i}_y{i}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(and_<tsp>{v}_x{v})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-and_<tsp>{v}_x{v}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (`'(and_<tsp>{b}_x{b})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-and_<tsp>{b}_x{b}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (`'(asr_<c>_<tsp>_<tsp>{b})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-asr_<c>_<tsp>_<tsp>{b}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(cpop_cfp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-cpop_cfp-step (i-state (*i-pc*,

$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\wedge$ (i-ins-step (`'(cpop_pc)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)

= i-cpop_pc-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,

$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\wedge$ (i-ins-step (`'(cpush_*)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-cpush_*-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

$\wedge$ (i-ins-step (`'(cpush_<tsp>+)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-cpush_<tsp>+-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(cpush_cfp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-cpush_cfp-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(decr_<tsp>_<tsp>{i})`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)

= i-decr_<tsp>_<tsp>{i}-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(decr_<tsp>_<tsp>{n})`,
i-state ($i\text{-}pc$,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-psw}), \\
\qquad\qquad\qquad \textit{load-addr}) \\
\quad = \quad \text{i-decr\_<tsp>\_<tsp>\{n\}-step}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-psw}))) \\
\wedge \quad (\text{i-ins-step}\,(\texttt{'(incr\_<tsp>\_<tsp>\{i\})}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment},
\end{array}
$$

i-word-size,
i-psw),
load-addr)
= i-incr_<tsp>_<tsp>{i}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(incr_<tsp>_<tsp>{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-incr_<tsp>_<tsp>{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(incr_y_y{n})`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-incr_y_y{n}-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(int-to-nat)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-int-to-nat-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
$\land$ (i-ins-step (`'(jump-n_x)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-jump-n_x-step (i-state (*i-pc*,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}word\text{-}size, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}psw))) \\
\wedge \quad (\text{i-ins-step}\,(\texttt{'(jump-nn\_x)}, \\
\qquad\qquad\quad \text{i-state}\,(i\text{-}pc, \\
\qquad\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad i\text{-}word\text{-}size, \\
\qquad\qquad\qquad\qquad i\text{-}psw), \\
\qquad\qquad\quad load\text{-}addr) \\
\quad = \quad \text{i-jump-nn\_x-step}\,(\text{i-state}\,(i\text{-}pc, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}cfp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}csp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}tsp, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}x, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}y, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}c\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}v\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}n\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}z\text{-}flg, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}prog\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}usr\text{-}data\text{-}segment, \\
\qquad\qquad\qquad\qquad\qquad\qquad i\text{-}sys\text{-}data\text{-}segment,
\end{array}
$$

i-word-size,
i-psw)))

$\wedge$ (i-ins-step ('(jump-nz_x),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-jump-nz_x-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

$\wedge$ (i-ins-step ('(jump-z_x),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-jump-z_x-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))
$\wedge$ (i-ins-step (`'(jump_*)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-jump_*-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (`'(jump_x{subr})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)

= i-jump_x{subr}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (`'(lsr_<c>_x_x{n})`,
i-state (*i-pc*,

```
                     i-cfp,
                     i-csp,
                     i-tsp,
                     i-x,
                     i-y,
                     i-c-flg,
                     i-v-flg,
                     i-n-flg,
                     i-z-flg,
                     i-prog-segment,
                     i-usr-data-segment,
                     i-sys-data-segment,
                     i-word-size,
                     i-psw),
                load-addr)
   =   i-lsr_<c>_x_x{n}-step (i-state (i-pc,
                                       i-cfp,
                                       i-csp,
                                       i-tsp,
                                       i-x,
                                       i-y,
                                       i-c-flg,
                                       i-v-flg,
                                       i-n-flg,
                                       i-z-flg,
                                       i-prog-segment,
                                       i-usr-data-segment,
                                       i-sys-data-segment,
                                       i-word-size,
                                       i-psw)))
∧  (i-ins-step ('(lsr_<tsp>_<tsp>{v}),
                i-state (i-pc,
                     i-cfp,
                     i-csp,
                     i-tsp,
                     i-x,
                     i-y,
                     i-c-flg,
                     i-v-flg,
                     i-n-flg,
                     i-z-flg,
                     i-prog-segment,
                     i-usr-data-segment,
                     i-sys-data-segment,
```

i-word-size,
i-psw),
load-addr)
= i-lsr_<tsp>_<tsp>{v}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(move-c_<tsp>_*)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move-c_<tsp>_*-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (**'(move-v_<tsp>_*)**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)

= i-move-v_<tsp>_*-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (**'(move-z_<tsp>_*)**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move-z_<tsp>_*-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(move-n_x_*)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move-n_x_*-step (i-state (*i-pc*,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size},\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw})))\\
\land \quad (\text{i-ins-step}\,(\texttt{'(move\_<tsp>\_*)},\\
\qquad\qquad \text{i-state}\,(\textit{i-pc},\\
\qquad\qquad\qquad \textit{i-cfp},\\
\qquad\qquad\qquad \textit{i-csp},\\
\qquad\qquad\qquad \textit{i-tsp},\\
\qquad\qquad\qquad \textit{i-x},\\
\qquad\qquad\qquad \textit{i-y},\\
\qquad\qquad\qquad \textit{i-c-flg},\\
\qquad\qquad\qquad \textit{i-v-flg},\\
\qquad\qquad\qquad \textit{i-n-flg},\\
\qquad\qquad\qquad \textit{i-z-flg},\\
\qquad\qquad\qquad \textit{i-prog-segment},\\
\qquad\qquad\qquad \textit{i-usr-data-segment},\\
\qquad\qquad\qquad \textit{i-sys-data-segment},\\
\qquad\qquad\qquad \textit{i-word-size},\\
\qquad\qquad\qquad \textit{i-psw}),\\
\qquad\qquad \textit{load-addr})\\
\quad = \quad \text{i-move\_<tsp>\_*-step}\,(\text{i-state}\,(\textit{i-pc},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-cfp},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-csp},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-tsp},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-x},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-y},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-c-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-v-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-n-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-z-flg},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-prog-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-usr-data-segment},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{i-sys-data-segment},
\end{array}
$$

i-word-size,
i-psw)))

∧ (i-ins-step ('`(move_<x{a}>_<tsp>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)

= i-move_<x{a}>_<tsp>-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step ('`(move_<x{s}>_<tsp>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move_<x{s}>_<tsp>-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(move_cfp_csp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move_cfp_csp-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(move_csp_cfp)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)

= i-move_csp_cfp-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(move_x_*)`,
i-state ($i\text{-}pc$,

```
                      i-cfp,
                      i-csp,
                      i-tsp,
                      i-x,
                      i-y,
                      i-c-flg,
                      i-v-flg,
                      i-n-flg,
                      i-z-flg,
                      i-prog-segment,
                      i-usr-data-segment,
                      i-sys-data-segment,
                      i-word-size,
                      i-psw),
               load-addr)
    =  i-move_x_*-step (i-state (i-pc,
                                 i-cfp,
                                 i-csp,
                                 i-tsp,
                                 i-x,
                                 i-y,
                                 i-c-flg,
                                 i-v-flg,
                                 i-n-flg,
                                 i-z-flg,
                                 i-prog-segment,
                                 i-usr-data-segment,
                                 i-sys-data-segment,
                                 i-word-size,
                                 i-psw)))
∧  (i-ins-step ('(move_x_<x{s}>),
               i-state (i-pc,
                      i-cfp,
                      i-csp,
                      i-tsp,
                      i-x,
                      i-y,
                      i-c-flg,
                      i-v-flg,
                      i-n-flg,
                      i-z-flg,
                      i-prog-segment,
                      i-usr-data-segment,
                      i-sys-data-segment,
```

i-word-size,
i-psw),
load-addr)
= i-move_x_<x{s}>-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(move_x_tsp)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move_x_tsp-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (`'(move_x_x)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)

= i-move_x_x-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (`'(move_y_*)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move_y_*-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step ('(move_y_<y{s}>),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-move_y_<y{s}>-step (i-state (*i-pc*,

$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step (`'(move_y_tsp)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)

= i-move_y_tsp-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,

i-word-size,
i-psw)))

∧ (i-ins-step ('**(neg_<tsp>_<tsp>{i})**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)

= i-neg_<tsp>_<tsp>{i}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step ('**(not_<tsp>_<tsp>{v})**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-not_<tsp>_<tsp>{v}-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))
∧ (i-ins-step (`'(or_<tsp>{v}_x{v})`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-or_<tsp>{v}_x{v}-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,

```
                                          i-y,
                                          i-c-flg,
                                          i-v-flg,
                                          i-n-flg,
                                          i-z-flg,
                                          i-prog-segment,
                                          i-usr-data-segment,
                                          i-sys-data-segment,
                                          i-word-size,
                                          i-psw)))
∧  (i-ins-step ('(or_<tsp>{b}_x{b}),
           i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw),
           load-addr)
   =  i-or_<tsp>{b}_x{b}-step (i-state (i-pc,
                                          i-cfp,
                                          i-csp,
                                          i-tsp,
                                          i-x,
                                          i-y,
                                          i-c-flg,
                                          i-v-flg,
                                          i-n-flg,
                                          i-z-flg,
                                          i-prog-segment,
                                          i-usr-data-segment,
                                          i-sys-data-segment,
                                          i-word-size,
                                          i-psw)))
∧  (i-ins-step ('(sub_<c>_<tsp>{a}_x{a}),
           i-state (i-pc,
```

```
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw),
              load-addr)
   =  i-sub_<c>_<tsp>{a}_x{a}-step (i-state (i-pc,
                                             i-cfp,
                                             i-csp,
                                             i-tsp,
                                             i-x,
                                             i-y,
                                             i-c-flg,
                                             i-v-flg,
                                             i-n-flg,
                                             i-z-flg,
                                             i-prog-segment,
                                             i-usr-data-segment,
                                             i-sys-data-segment,
                                             i-word-size,
                                             i-psw)))
∧  (i-ins-step ('(sub_<c>_<tsp>{n}_x{n}),
              i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
```

i-word-size,
i-psw),
load-addr)
= i-sub_<c>_<tsp>{n}_x{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(sub_<nv>_<tsp>{i}_x{i})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-sub_<nv>_<tsp>{i}_x{i}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

```
                                                 i-z-flg,
                                                 i-prog-segment,
                                                 i-usr-data-segment,
                                                 i-sys-data-segment,
                                                 i-word-size,
                                                 i-psw)))
∧  (i-ins-step ('(sub_<tsp>{a}_x{n}),
             i-state (i-pc,
                      i-cfp,
                      i-csp,
                      i-tsp,
                      i-x,
                      i-y,
                      i-c-flg,
                      i-v-flg,
                      i-n-flg,
                      i-z-flg,
                      i-prog-segment,
                      i-usr-data-segment,
                      i-sys-data-segment,
                      i-word-size,
                      i-psw),
             load-addr)
    =  i-sub_<tsp>{a}_x{n}-step (i-state (i-pc,
                                           i-cfp,
                                           i-csp,
                                           i-tsp,
                                           i-x,
                                           i-y,
                                           i-c-flg,
                                           i-v-flg,
                                           i-n-flg,
                                           i-z-flg,
                                           i-prog-segment,
                                           i-usr-data-segment,
                                           i-sys-data-segment,
                                           i-word-size,
                                           i-psw)))
∧  (i-ins-step ('(sub_x{s}_y{n}),
             i-state (i-pc,
                      i-cfp,
                      i-csp,
                      i-tsp,
                      i-x,
```

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-sub_x{s}_y{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
$\wedge$ (i-ins-step (`'(sub_<tsp>{i}_x{i})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-sub_<tsp>{i}_x{i}-step (i-state (*i-pc*,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}))) \\
\land \quad (\text{i-ins-step}\,(\texttt{'(sub\_<tsp>\{n\}\_x\{n\})}, \\
\qquad\qquad\quad \text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}), \\
\qquad\qquad\quad \mathit{load\text{-}addr}) \\
\quad = \quad \text{i-sub\_<tsp>\{n\}\_x\{n\}-step}\,(\text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment},
\end{array}
$$

i-word-size,
i-psw)))
∧ (i-ins-step ('(sub_<tsp>{s}_x{s}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-sub_<tsp>{s}_x{s}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step ('(sub_<z>_x{s}_y{s}),
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-sub_<z>_x{s}_y{s}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(subb_<c>_x{n}_y{n})`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-subb_<c>_x{n}_y{n}-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

```
                                        i-y,
                                        i-c-flg,
                                        i-v-flg,
                                        i-n-flg,
                                        i-z-flg,
                                        i-prog-segment,
                                        i-usr-data-segment,
                                        i-sys-data-segment,
                                        i-word-size,
                                        i-psw)))
∧  (i-ins-step ('(subb_<v>_x{i}_y{i}),
           i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw),
           load-addr)
    =  i-subb_<v>_x{i}_y{i}-step (i-state (i-pc,
                                       i-cfp,
                                       i-csp,
                                       i-tsp,
                                       i-x,
                                       i-y,
                                       i-c-flg,
                                       i-v-flg,
                                       i-n-flg,
                                       i-z-flg,
                                       i-prog-segment,
                                       i-usr-data-segment,
                                       i-sys-data-segment,
                                       i-word-size,
                                       i-psw)))
∧  (i-ins-step ('(tpop_<c>_x),
           i-state (i-pc,
```

$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)
= i-tpop_<c>_x-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))
$\wedge$ (i-ins-step (`'(tpop_<x{a}>)`,
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,

```
                  i-word-size,
                  i-psw),
              load-addr)
   =   i-tpop_<x{a}>-step (i-state (i-pc,
                                    i-cfp,
                                    i-csp,
                                    i-tsp,
                                    i-x,
                                    i-y,
                                    i-c-flg,
                                    i-v-flg,
                                    i-n-flg,
                                    i-z-flg,
                                    i-prog-segment,
                                    i-usr-data-segment,
                                    i-sys-data-segment,
                                    i-word-size,
                                    i-psw)))
∧   (i-ins-step ('(tpop_<x{s}>),
              i-state (i-pc,
                  i-cfp,
                  i-csp,
                  i-tsp,
                  i-x,
                  i-y,
                  i-c-flg,
                  i-v-flg,
                  i-n-flg,
                  i-z-flg,
                  i-prog-segment,
                  i-usr-data-segment,
                  i-sys-data-segment,
                  i-word-size,
                  i-psw),
              load-addr)
   =   i-tpop_<x{s}>-step (i-state (i-pc,
                                    i-cfp,
                                    i-csp,
                                    i-tsp,
                                    i-x,
                                    i-y,
                                    i-c-flg,
                                    i-v-flg,
                                    i-n-flg,
```

$i\text{-}z\text{-}flg,$
$i\text{-}prog\text{-}segment,$
$i\text{-}usr\text{-}data\text{-}segment,$
$i\text{-}sys\text{-}data\text{-}segment,$
$i\text{-}word\text{-}size,$
$i\text{-}psw)))$

$\wedge$ (i-ins-step (`'(tpop_pc)`,
i-state ($i\text{-}pc,$
$i\text{-}cfp,$
$i\text{-}csp,$
$i\text{-}tsp,$
$i\text{-}x,$
$i\text{-}y,$
$i\text{-}c\text{-}flg,$
$i\text{-}v\text{-}flg,$
$i\text{-}n\text{-}flg,$
$i\text{-}z\text{-}flg,$
$i\text{-}prog\text{-}segment,$
$i\text{-}usr\text{-}data\text{-}segment,$
$i\text{-}sys\text{-}data\text{-}segment,$
$i\text{-}word\text{-}size,$
$i\text{-}psw),$
$load\text{-}addr)$
= i-tpop_pc-step (i-state ($i\text{-}pc,$
$i\text{-}cfp,$
$i\text{-}csp,$
$i\text{-}tsp,$
$i\text{-}x,$
$i\text{-}y,$
$i\text{-}c\text{-}flg,$
$i\text{-}v\text{-}flg,$
$i\text{-}n\text{-}flg,$
$i\text{-}z\text{-}flg,$
$i\text{-}prog\text{-}segment,$
$i\text{-}usr\text{-}data\text{-}segment,$
$i\text{-}sys\text{-}data\text{-}segment,$
$i\text{-}word\text{-}size,$
$i\text{-}psw)))$

$\wedge$ (i-ins-step (`'(tpop_x)`,
i-state ($i\text{-}pc,$
$i\text{-}cfp,$
$i\text{-}csp,$
$i\text{-}tsp,$
$i\text{-}x,$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw}), \\
\qquad\qquad\qquad \textit{load-addr}) \\
\quad = \quad \text{i-tpop\_x-step}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-step}\,(\texttt{'(tpop\_y)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw}), \\
\qquad\qquad\qquad \textit{load-addr}) \\
\quad = \quad \text{i-tpop\_y-step}\,(\text{i-state}\,(\textit{i-pc},
\end{array}
$$

```
                                  i-cfp,
                                  i-csp,
                                  i-tsp,
                                  i-x,
                                  i-y,
                                  i-c-flg,
                                  i-v-flg,
                                  i-n-flg,
                                  i-z-flg,
                                  i-prog-segment,
                                  i-usr-data-segment,
                                  i-sys-data-segment,
                                  i-word-size,
                                  i-psw)))
∧  (i-ins-step ('(tpop{v}_<z>_y),
               i-state (i-pc,
                        i-cfp,
                        i-csp,
                        i-tsp,
                        i-x,
                        i-y,
                        i-c-flg,
                        i-v-flg,
                        i-n-flg,
                        i-z-flg,
                        i-prog-segment,
                        i-usr-data-segment,
                        i-sys-data-segment,
                        i-word-size,
                        i-psw),
               load-addr)
    =  i-tpop{v}_<z>_y-step (i-state (i-pc,
                                      i-cfp,
                                      i-csp,
                                      i-tsp,
                                      i-x,
                                      i-y,
                                      i-c-flg,
                                      i-v-flg,
                                      i-n-flg,
                                      i-z-flg,
                                      i-prog-segment,
                                      i-usr-data-segment,
                                      i-sys-data-segment,
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}))) \\
\land \quad (\text{i-ins-step}\,(\texttt{'(tpop\{b\}\_<z>\_y)}, \\
\qquad\qquad \text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad \mathit{i\text{-}psw}), \\
\qquad\qquad \mathit{load\text{-}addr}) \\
\qquad = \quad \text{i-tpop\{b\}\_<z>\_y-step}\,(\text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}z\text{-}flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}usr\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}sys\text{-}data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}word\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{i\text{-}psw}))) \\
\land \quad (\text{i-ins-step}\,(\texttt{'(tpop\{i\}\_<zn>\_y)}, \\
\qquad\qquad \text{i-state}\,(\mathit{i\text{-}pc}, \\
\qquad\qquad\qquad \mathit{i\text{-}cfp}, \\
\qquad\qquad\qquad \mathit{i\text{-}csp}, \\
\qquad\qquad\qquad \mathit{i\text{-}tsp}, \\
\qquad\qquad\qquad \mathit{i\text{-}x}, \\
\qquad\qquad\qquad \mathit{i\text{-}y}, \\
\qquad\qquad\qquad \mathit{i\text{-}c\text{-}flg}, \\
\qquad\qquad\qquad \mathit{i\text{-}v\text{-}flg}, \\
\qquad\qquad\qquad \mathit{i\text{-}n\text{-}flg},
\end{array}
$$

i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-tpop{i}_<zn>_y-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (**'(tpop{n}_<z>_y)**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-tpop{n}_<z>_y-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (**'(tpush_*)**,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-tpush_*-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))

∧ (i-ins-step (**'(tpush_<x{a}>)**,
i-state (*i-pc*,

i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-tpush_<x{a}>-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(tpush_<x{s}>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,

```
                i-word-size,
                i-psw),
            load-addr)
    =   i-tpush_<x{s}>-step (i-state (i-pc,
                                      i-cfp,
                                      i-csp,
                                      i-tsp,
                                      i-x,
                                      i-y,
                                      i-c-flg,
                                      i-v-flg,
                                      i-n-flg,
                                      i-z-flg,
                                      i-prog-segment,
                                      i-usr-data-segment,
                                      i-sys-data-segment,
                                      i-word-size,
                                      i-psw)))
∧   (i-ins-step ('(tpush_csp),
            i-state (i-pc,
                i-cfp,
                i-csp,
                i-tsp,
                i-x,
                i-y,
                i-c-flg,
                i-v-flg,
                i-n-flg,
                i-z-flg,
                i-prog-segment,
                i-usr-data-segment,
                i-sys-data-segment,
                i-word-size,
                i-psw),
            load-addr)
    =   i-tpush_csp-step (i-state (i-pc,
                                   i-cfp,
                                   i-csp,
                                   i-tsp,
                                   i-x,
                                   i-y,
                                   i-c-flg,
                                   i-v-flg,
                                   i-n-flg,
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-step}\,(\texttt{'(tpush\_tsp)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad \textit{i-psw}), \\
\qquad\qquad\qquad \textit{load-addr}) \\
\quad = \quad \text{i-tpush\_tsp-step}\,(\text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-x}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-y}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-prog-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-word-size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{i-psw}))) \\
\land \quad (\text{i-ins-step}\,(\texttt{'(tpush\_x)}, \\
\qquad\qquad\qquad \text{i-state}\,(\textit{i-pc}, \\
\qquad\qquad\qquad\qquad \textit{i-cfp}, \\
\qquad\qquad\qquad\qquad \textit{i-csp}, \\
\qquad\qquad\qquad\qquad \textit{i-tsp}, \\
\qquad\qquad\qquad\qquad \textit{i-x},
\end{array}
$$

i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-tpush_x-step (i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw)))
∧ (i-ins-step (`'(xor_<tsp>_<tsp>)`,
i-state (*i-pc*,
i-cfp,
i-csp,
i-tsp,
i-x,
i-y,
i-c-flg,
i-v-flg,
i-n-flg,
i-z-flg,
i-prog-segment,
i-usr-data-segment,
i-sys-data-segment,
i-word-size,
i-psw),
load-addr)
= i-xor_<tsp>_<tsp>-step (i-state (*i-pc*,

```
                                        i-cfp,
                                        i-csp,
                                        i-tsp,
                                        i-x,
                                        i-y,
                                        i-c-flg,
                                        i-v-flg,
                                        i-n-flg,
                                        i-z-flg,
                                        i-prog-segment,
                                        i-usr-data-segment,
                                        i-sys-data-segment,
                                        i-word-size,
                                        i-psw)))
∧   (i-ins-step ('(xor_<tsp>{v}_x{v}),
                i-state (i-pc,
                         i-cfp,
                         i-csp,
                         i-tsp,
                         i-x,
                         i-y,
                         i-c-flg,
                         i-v-flg,
                         i-n-flg,
                         i-z-flg,
                         i-prog-segment,
                         i-usr-data-segment,
                         i-sys-data-segment,
                         i-word-size,
                         i-psw),
                load-addr)
  =   i-xor_<tsp>{v}_x{v}-step (i-state (i-pc,
                                         i-cfp,
                                         i-csp,
                                         i-tsp,
                                         i-x,
                                         i-y,
                                         i-c-flg,
                                         i-v-flg,
                                         i-n-flg,
                                         i-z-flg,
                                         i-prog-segment,
                                         i-usr-data-segment,
                                         i-sys-data-segment,
```

$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step ('(xor_<tsp>{b}_*{b}),
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$)

= i-xor_<tsp>{b}_*{b}-step (i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$)))

∧ (i-ins-step ('(xor_<tsp>{b}_x{b}),
i-state ($i\text{-}pc$,
$i\text{-}cfp$,
$i\text{-}csp$,
$i\text{-}tsp$,
$i\text{-}x$,
$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,

```
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw),
               load-addr)
    =   i-xor_<tsp>{b}_x{b}-step (i-state (i-pc,
                                           i-cfp,
                                           i-csp,
                                           i-tsp,
                                           i-x,
                                           i-y,
                                           i-c-flg,
                                           i-v-flg,
                                           i-n-flg,
                                           i-z-flg,
                                           i-prog-segment,
                                           i-usr-data-segment,
                                           i-sys-data-segment,
                                           i-word-size,
                                           i-psw)))
∧   (i-ins-step ('(xor_<z>_<tsp>_x),
               i-state (i-pc,
                    i-cfp,
                    i-csp,
                    i-tsp,
                    i-x,
                    i-y,
                    i-c-flg,
                    i-v-flg,
                    i-n-flg,
                    i-z-flg,
                    i-prog-segment,
                    i-usr-data-segment,
                    i-sys-data-segment,
                    i-word-size,
                    i-psw),
               load-addr)
    =   i-xor_<z>_<tsp>_x-step (i-state (i-pc,
                                         i-cfp,
                                         i-csp,
                                         i-tsp,
                                         i-x,
```

$i\text{-}y$,
$i\text{-}c\text{-}flg$,
$i\text{-}v\text{-}flg$,
$i\text{-}n\text{-}flg$,
$i\text{-}z\text{-}flg$,
$i\text{-}prog\text{-}segment$,
$i\text{-}usr\text{-}data\text{-}segment$,
$i\text{-}sys\text{-}data\text{-}segment$,
$i\text{-}word\text{-}size$,
$i\text{-}psw$),
$load\text{-}addr$))

```
; Empty-stkp introduces difference, which slows things down a lot.
; We avoid opening it by proving:
```

THEOREM: powerful-length-put
length (put (val, n, lst))
$=$ **if** $n <$ length (lst) **then** length (lst)
else $1 + n$ **endif**

THEOREM: empty-stkp-deposit-adp
($\neg$ empty-stkp (sp, $segment$))
$\rightarrow$ ($\neg$ empty-stkp (sp, deposit-adp (val, adp, $segment$)))

```
; Under what conditions is a pushed stack empty?  What does empty
; mean?  It means there is no room left.  When is there no room?  When
; the stack pointer is one less than the length of the stack area.
; A push moves the stack pointer down -- provided the pointer is not
; zero (i.e., provided the stack is not already full).  So, summarizing,
; a pushed stack pointer is non-empty provided it wasn't full before
; the push.  In addition, it had better be a legal stack pointer
; initially or else the push could just bring it down onto the
; upper limit!
```

THEOREM: not-empty-stkp-push
(adpp (untag (sp), $segment$) $\wedge$ (adp-offset (untag (sp)) $\not\simeq$ `0`))
$\rightarrow$ ($\neg$ empty-stkp (tag (`'sys-addr`, sub-adp (untag (sp), $1 + n$)), $segment$))

EVENT: Disable empty-stkp.

```
; The following lemma temporarily opens up adpp by backchaining into
; its defn.
```

THEOREM: adpp-backchainer
(listp (x)
$\land$ (adp-offset $(x) \in \mathbf{N}$)
$\land$ definedp (adp-name (x), *segment*)
$\land$ (adp-offset (x) < length (definiens (adp-name (x), *segment*))))
$\rightarrow$ adpp $(x$, *segment*)

```
; We need to
```

EVENT: Disable i-objectp.

```
; we will deduce what we have to from it.
```

THEOREM: unlabel-tag
unlabel (tag (*type*, *obj*))
= **if** *type* = `'dl` **then** `0`
else tag (*type*, *obj*) **endif**

EVENT: Enable cddr-tag.

```
; I am now heading for the lemma that states that if x is an r-objectp
; in r then it is an i-objectp in an appropriately corresponding i state.
```

THEOREM: same-signature-implies-adpp
same-signature (*segment1*, *segment2*)
$\rightarrow$ (adpp $(x$, *segment1*) = adpp $(x$, *segment2*))

THEOREM: r-program-bodyp-program-body-any
(proper-r-statep $(r$, *load-addr*) $\land$ definedp (*name*, r-prog-segment (r)))
$\rightarrow$ r-program-bodyp (program-body (assoc (*name*, r-prog-segment (r))))

THEOREM: listp-cdr-icompile-program
listp (cdr (icompile-program (*program*)))

```
; So here is the r-objectp -> i-objectp lemma.
```

```
; The proper-r-statep below is required to show that the labels show up
; properly in the i-level code.
```

THEOREM: r-objectp-implies-i-objectp
(proper-r-statep $(r$, *load-addr*)

$\wedge$ r-objectp (x, r)
$\wedge$ same-signature (r-sys-data-segment (r), *i-sys-data-segment*)
$\wedge$ same-signature (r-usr-data-segment (r), *i-usr-data-segment*))
$\rightarrow$ i-objectp (x,
    i-state (*i-pc*,
        i-cfp,
        i-csp,
        i-tsp,
        i-x,
        i-y,
        i-c-flg,
        i-v-flg,
        i-n-flg,
        i-z-flg,
        icompile (r-prog-segment (r)),
        i-usr-data-segment,
        i-sys-data-segment,
        r-word-size (r),
        i-psw))

THEOREM: same-signature-reflexive
same-signature (*segment*, *segment*)

THEOREM: same-signature-put-assoc
(litatom (*name*)
$\wedge$ definedp (*name*, *segment2*)
$\wedge$ (length (definiens (*name*, *segment2*)) = length (*val*)))
$\rightarrow$ (same-signature (*segment1*, put-assoc (*val*, *name*, *segment2*))
  = same-signature (*segment1*, *segment2*))

```
; The litatom hyp in the same-signature lemma will be relieved, I think,
; by:
```

DEFINITION:
proper-alistp (*alist*)
= **if** *alist* $\simeq$ **nil then** *alist* = **nil**
  else listp (car (*alist*))
    $\wedge$ litatom (caar (*alist*))
    $\wedge$ proper-alistp (cdr (*alist*)) **endif**

THEOREM: litatom-adp-name
(definedp (adp-name (*adp*), *segment*) $\wedge$ proper-alistp (*segment*))
$\rightarrow$ litatom (adp-name (*adp*))

```
; The lemma above has nothing to do with adp-name -- it could be generalized
; to an arbitrary name.  But it has been so instantiated to be useful.

; To establish the proper-alistp above we'll need something like the following
; two lemmas.  The first dives past all the modifications to the
; usr-data-segment and the second (or third) backchains into the
; proper-r-statep.
```

THEOREM: proper-alistp-put-assoc
proper-alistp (put-assoc (*val*, *name*, *alist*)) = proper-alistp (*alist*)

THEOREM: proper-r-usr-data-segmentp-implies-proper-alistp
proper-r-usr-data-segmentp (*segment*, r) $\rightarrow$ proper-alistp (*segment*)

THEOREM: proper-alistp-r-usr-data-segment
proper-r-statep (r, *load-addr*) $\rightarrow$ proper-alistp (r-usr-data-segment (r))

```
; The r-objectp hyp of the i-objectp lemma will often be relieved by the
; next lemma when fetching from the r-usr-data-segment.
```

THEOREM: r-objectp-get-all-r-usr-data-objectps
((n < length (*lst*)) $\wedge$ all-r-usr-data-objectps (*lst*, r))
$\rightarrow$ r-objectp (get (n, *lst*), r)

THEOREM: all-r-usr-data-objectps-cdr-assoc
(proper-r-usr-data-segmentp (*segment*, r) $\wedge$ definedp (*name*, *segment*))
$\rightarrow$ all-r-usr-data-objectps (cdr (assoc (*name*, *segment*)), r)

THEOREM: r-objectp-fetch-adp-r-usr-data-segment
(proper-r-statep (r, *load-addr*) $\wedge$ adpp (*adp*, r-usr-data-segment (r)))
$\rightarrow$ r-objectp (fetch-adp (*adp*, r-usr-data-segment (r)), r)

THEOREM: r-objectp-opener
((type (x) = `'nat`)
 $\rightarrow$ (r-objectp (x, r)
  = ((cddr (x) = **nil**)
   $\wedge$ small-naturalp (untag (x), r-word-size (r)))))
$\wedge$ ((type (x) = `'int`)
 $\rightarrow$ (r-objectp (x, r)
  = ((cddr (x) = **nil**)
   $\wedge$ small-integerp (untag (x), r-word-size (r)))))
$\wedge$ ((type (x) = `'bitv`)
 $\rightarrow$ (r-objectp (x, r)

$= \quad ((\text{cddr}\,(x) = \textbf{nil})$
$\quad\quad \wedge \quad \text{bit-vectorp}\,(\text{untag}\,(x),\ \text{r-word-size}\,(r)))))$

$\wedge \quad ((\text{type}\,(x) = \texttt{'bool})$
$\quad \rightarrow \quad (\text{r-objectp}\,(x,\ r)$
$\quad\quad = \quad ((\text{cddr}\,(x) = \textbf{nil}) \wedge \text{booleanp}\,(\text{untag}\,(x)))))$

$\wedge \quad ((\text{type}\,(x) = \texttt{'addr})$
$\quad \rightarrow \quad (\text{r-objectp}\,(x,\ r)$
$\quad\quad = \quad ((\text{cddr}\,(x) = \textbf{nil})$
$\quad\quad\quad \wedge \quad \text{adpp}\,(\text{untag}\,(x),\ \text{r-usr-data-segment}\,(r)))))$

$\wedge \quad ((\text{type}\,(x) = \texttt{'pc})$
$\quad \rightarrow \quad (\text{r-objectp}\,(x,\ r)$
$\quad\quad = \quad ((\text{cddr}\,(x) = \textbf{nil})$
$\quad\quad\quad \wedge \quad \text{pcpp}\,(\text{untag}\,(x),\ \text{r-prog-segment}\,(r)))))$

$\wedge \quad ((\text{type}\,(x) = \texttt{'subr})$
$\quad \rightarrow \quad (\text{r-objectp}\,(x,\ r)$
$\quad\quad = \quad ((\text{cddr}\,(x) = \textbf{nil})$
$\quad\quad\quad \wedge \quad \text{definedp}\,(\text{untag}\,(x),\ \text{r-prog-segment}\,(r)))))$

$\wedge \quad ((\text{type}\,(x) = \texttt{'sys-addr})$
$\quad \rightarrow \quad (\text{r-objectp}\,(x,\ r)$
$\quad\quad = \quad ((\text{cddr}\,(x) = \textbf{nil})$
$\quad\quad\quad \wedge \quad \text{adpp}\,(\text{untag}\,(x),\ \text{r-sys-data-segment}\,(r)))))$

THEOREM: i-objectp-tag
$(\text{i-objectp}\,(\text{tag}\,(\texttt{'nat},\ x),\ i) = \text{small-naturalp}\,(x,\ \text{i-word-size}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'int},\ x),\ i) = \text{small-integerp}\,(x,\ \text{i-word-size}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'bitv},\ x),\ i) = \text{bit-vectorp}\,(x,\ \text{i-word-size}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'bool},\ x),\ i) = \text{booleanp}\,(x))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'addr},\ x),\ i) = \text{adpp}\,(x,\ \text{i-usr-data-segment}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'pc},\ x),\ i) = \text{icode-labelp}\,(x,\ \text{i-prog-segment}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'subr},\ x),\ i) = \text{adpp}\,(\text{cons}\,(x,\ \texttt{0}),\ \text{i-prog-segment}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'sys-addr},\ x),\ i) = \text{adpp}\,(x,\ \text{i-sys-data-segment}\,(i)))$
$\wedge \quad (\text{i-objectp}\,(\text{tag}\,(\texttt{'ipc},\ x),\ i) = \text{adpp}\,(x,\ \text{i-prog-segment}\,(i)))$

```
; Now we begin the one way correspondence proofs for each instruction.
```

THEOREM: no-op-one-way-correspondence-r-i
$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'no-op})$
$\wedge \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})$
$\wedge \quad (\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge \quad \text{r-no-op-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow \quad (\text{r->i}\,(\text{r-no-op-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\quad\quad \text{r-no-op-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$

$load\text{-}addr$))

THEOREM: push-global-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'push-global})$
$\land \quad \text{proper-r-statep}(r,\ load\text{-}addr)$
$\land \quad (load\text{-}addr \in \mathbf{N})$
$\land \quad \text{r-push-global-okp}(\text{r-current-instruction}(r),\ r))$
$\to \quad (\text{r->i}(\text{r-push-global-step}(\text{r-current-instruction}(r),\ r))$
$\quad = \quad \text{i}(\text{r->i}(r),$
$\qquad \text{r-push-global-step-clock}(\text{r-current-instruction}(r),\ r),$
$\qquad load\text{-}addr))$

THEOREM: lessp-find-position-of-var
$\text{length}(x) \not< \text{find-position-of-var}(name,\ x)$

THEOREM: lessp-length-icompile-program
$(prog \in progs)$
$\to \quad (\text{segment-length}(\text{icompile}(progs))$
$\quad \not< \quad \text{length}(\text{cdr}(\text{icompile-program}(prog))))$

THEOREM: times-add1{gen}
$(x * (1 + y)) = (x + (x * y))$

THEOREM: length-strip-cars
$\text{length}(\text{strip-cars}(x)) = \text{length}(x)$

THEOREM: lessp-length-local-vars
$\text{length}(\text{local-vars}(d)) < \text{length}(\text{generate-prelude}(d))$

THEOREM: lessp-find-position-of-var-exp
$(\text{proper-r-statep}(r,\ load\text{-}addr) \land \text{definedp}(name,\ \text{r-prog-segment}(r)))$
$\to \quad (\text{find-position-of-var}(var,$
$\qquad \text{append}(\text{formal-vars}(\text{assoc}(name,\ \text{r-prog-segment}(r))),$
$\qquad\quad \text{strip-cars}(\text{temp-var-dcls}(\text{assoc}(name,$
$\qquad\qquad \text{r-prog-segment}(r))))))$
$\quad < \quad \exp(\mathbf{2},\ \text{r-word-size}(r)))$

THEOREM: push-local-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'push-local})$
$\land \quad \text{proper-r-statep}(r,\ load\text{-}addr)$
$\land \quad (load\text{-}addr \in \mathbf{N})$
$\land \quad \text{r-push-local-okp}(\text{r-current-instruction}(r),\ r))$
$\to \quad (\text{r->i}(\text{r-push-local-step}(\text{r-current-instruction}(r),\ r))$
$\quad = \quad \text{i}(\text{r->i}(r),$
$\qquad \text{r-push-local-step-clock}(\text{r-current-instruction}(r),\ r),$
$\qquad load\text{-}addr))$

THEOREM: add-addp-1-folded
cons (adp-name (untag (r-pc (r))), $1 +$ adp-offset (untag (r-pc (r))))
$=$ add-adp (untag (r-pc (r)), `1`)

THEOREM: unlabel-non-dl
(car (x) $\neq$ `'dl`) $\rightarrow$ (unlabel (x) $= x$)

THEOREM: r-program-bodyp-contains-r-instructionps
(r-program-bodyp (*lst*) $\wedge$ (*ins* $\in$ *lst*)) $\rightarrow$ r-instructionp (unlabel (*ins*))

THEOREM: r-prog-segmentp-contains-r-program-bodyps
(r-prog-segmentp (*segment*) $\wedge$ (*prog* $\in$ *segment*))
$\rightarrow$ r-program-bodyp (program-body (*prog*))

THEOREM: member-get
($n <$ length (*lst*)) $\rightarrow$ (get (n, *lst*) $\in$ *lst*)

THEOREM: proper-r-statep-implies-r-instructionp
(proper-r-statep (r, *load-addr*)
$\wedge$ definedp (*name*, r-prog-segment (r))
$\wedge$ ($n <$ length (program-body (assoc (*name*, r-prog-segment (r))))))
$\rightarrow$ r-instructionp (unlabel (get (n,
program-body (assoc (*name*, r-prog-segment (r))))))

THEOREM: pcpp-cons
(definedp (*name*, r-prog-segment (r))
$\wedge$ ($n \in \mathbf{N}$)
$\wedge$ ($n <$ length (program-body (assoc (*name*, r-prog-segment (r))))))
$\rightarrow$ pcpp (cons (*name*, n), r-prog-segment (r))

THEOREM: not-r-objectp
($\neg$ listp (x)) $\rightarrow$ ($\neg$ r-objectp (x, r))

THEOREM: push-constant-one-way-correspondence-r-i
((r-psw (r) $=$ `'run`)
$\wedge$ (car (r-current-instruction (r)) $=$ `'push-constant`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ (*load-addr* $\in \mathbf{N}$)
$\wedge$ r-push-constant-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-push-constant-step (r-current-instruction (r), r))
$=$ i (r->i (r),
r-push-constant-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: i-objectp-cons-sys-addr
i-objectp (list (`'sys-addr`, *adp*), i) $=$ adpp (*adp*, i-sys-data-segment (i))

THEOREM: sub-adp-sub-adp
sub-adp (sub-adp (adp, i), j) = sub-adp (adp, i + j)

THEOREM: equal-sub-adp-cons-adp-name
(sub-adp (x, n) = cons (adp-name (x), y)) = ((adp-offset (x) − n) = y)

THEOREM: length-cdr-assoc-deposit-adp
adpp (adp, $segment$)
→ (length (cdr (assoc ($name$, deposit-adp (val, adp, $segment$))))
  = length (cdr (assoc ($name$, $segment$))))

THEOREM: push-ctrl-stk-free-size-one-way-correspondence-r-i
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'push-ctrl-stk-free-size`)
∧ proper-r-statep (r, $load\text{-}addr$)
∧ ($load\text{-}addr \in \mathbf{N}$)
∧ r-push-ctrl-stk-free-size-okp (r-current-instruction (r), r))
→ (r->i (r-push-ctrl-stk-free-size-step (r-current-instruction (r), r))
  = i (r->i (r),
    r-push-ctrl-stk-free-size-step-clock (r-current-instruction (r),
    r),
    $load\text{-}addr$))

THEOREM: push-temp-stk-free-size-one-way-correspondence-r-i
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'push-temp-stk-free-size`)
∧ proper-r-statep (r, $load\text{-}addr$)
∧ ($load\text{-}addr \in \mathbf{N}$)
∧ r-push-temp-stk-free-size-okp (r-current-instruction (r), r))
→ (r->i (r-push-temp-stk-free-size-step (r-current-instruction (r), r))
  = i (r->i (r),
    r-push-temp-stk-free-size-step-clock (r-current-instruction (r),
    r),
    $load\text{-}addr$))

THEOREM: sub1-difference
$((x - y) - 1) = ((x - 1) - y)$

THEOREM: lessp-length-assoc-segment-length
segment-length ($segment$) $\not<$ length (cdr (assoc ($name$, $segment$)))

THEOREM: lessp-length-assoc-tstk-exp-2-word-size
proper-r-statep (r, $load\text{-}addr$)
→ (length (cdr (assoc (`'tstk`, r-sys-data-segment (r))))
  < exp (2, r-word-size (r)))

THEOREM: push-temp-stk-index-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'push-temp-stk-index**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-push-temp-stk-index-okp (r-current-instruction (r), r))
→ (r->i (r-push-temp-stk-index-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-push-temp-stk-index-step-clock (r-current-instruction (r), r),
 load-addr))

EVENT: Disable sub1-difference.

THEOREM: pop-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'pop**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-pop-okp (r-current-instruction (r), r))
→ (r->i (r-pop-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-pop-step-clock (r-current-instruction (r), r),
 load-addr))

```
; r-pop*-okp just checks that tsp + n is a stack address, but the
; i machine needs to know n is a small-naturalp.  The following
; lemma establishes that.
```

THEOREM: adpp-implies-lessp-exp-2-word-size
(proper-r-statep (r, *load-addr*) ∧ adpp (add-adp (a, n), r-sys-data-segment (r)))
→ ((n < exp (2, r-word-size (r))) = **t**)

THEOREM: pop*-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'pop***)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-pop*-okp (r-current-instruction (r), r))
→ (r->i (r-pop*-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-pop*-step-clock (r-current-instruction (r), r),
 load-addr))

THEOREM: plus-add1-first
$((1 + i) + j) = (1 + (i + j))$

THEOREM: popn-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'popn**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* ∈ **N**)
 ∧ r-popn-okp (r-current-instruction (r), r))
→ (r->i (r-popn-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-popn-step-clock (r-current-instruction (r), r),
 load-addr))

THEOREM: pop-local-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'pop-local**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* ∈ **N**)
 ∧ r-pop-local-okp (r-current-instruction (r), r))
→ (r->i (r-pop-local-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-pop-local-step-clock (r-current-instruction (r), r),
 load-addr))

THEOREM: pop-global-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'pop-global**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* ∈ **N**)
 ∧ r-pop-global-okp (r-current-instruction (r), r))
→ (r->i (r-pop-global-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-pop-global-step-clock (r-current-instruction (r), r),
 load-addr))

EVENT: Enable get-add1-cons.

THEOREM: get-0
get (0, *lst*) = car (*lst*)

THEOREM: set-local-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'set-local**)

$\wedge$ proper-r-statep (*r*, *load-addr*)
$\wedge$ (*load-addr* $\in$ **N**)
$\wedge$ r-set-local-okp (r-current-instruction (*r*), *r*))
$\rightarrow$ (r->i (r-set-local-step (r-current-instruction (*r*), *r*))
  = i (r->i (*r*),
      r-set-local-step-clock (r-current-instruction (*r*), *r*),
      load-addr))

THEOREM: set-global-one-way-correspondence-r-i
((r-psw (*r*) = `'run`)
$\wedge$ (car (r-current-instruction (*r*)) = `'set-global`)
$\wedge$ proper-r-statep (*r*, *load-addr*)
$\wedge$ (*load-addr* $\in$ **N**)
$\wedge$ r-set-global-okp (r-current-instruction (*r*), *r*))
$\rightarrow$ (r->i (r-set-global-step (r-current-instruction (*r*), *r*))
  = i (r->i (*r*),
      r-set-global-step-clock (r-current-instruction (*r*), *r*),
      load-addr))

```
; The following lemma really just opens up r->i_pc in the
; special case that arises for the jump instruction.
```

THEOREM: r->i_pc-opener-in-jump-case
r->i_pc (tag (`'pc`,
      cons (*name*,
         find-label (*lab*, program-body (assoc (*name*, r-prog-segment (*r*)))))),
    r-prog-segment (*r*))
= tag (`'ipc`,
    cons (*name*,
       find-label (cons (*name*,
              find-label (*lab*,
                  program-body (assoc (*name*,
                          r-prog-segment (*r*))))),
           cdr (icompile-program (definition (*name*,
                          r-prog-segment (*r*)))))))

THEOREM: jump-one-way-correspondence-r-i
((r-psw (*r*) = `'run`)
$\wedge$ (car (r-current-instruction (*r*)) = `'jump`)
$\wedge$ proper-r-statep (*r*, *load-addr*)
$\wedge$ (*load-addr* $\in$ **N**)
$\wedge$ r-jump-okp (r-current-instruction (*r*), *r*))
$\rightarrow$ (r->i (r-jump-step (r-current-instruction (*r*), *r*))
  = i (r->i (*r*),

```
                r-jump-step-clock (r-current-instruction (r), r),
                load-addr))
```

THEOREM: pushj-one-way-correspondence-r-i

```
((r-psw (r) = 'run)
 ∧  (car (r-current-instruction (r)) = 'pushj)
 ∧  proper-r-statep (r, load-addr)
 ∧  (load-addr ∈ N)
 ∧  r-pushj-okp (r-current-instruction (r), r))
→  (r->i (r-pushj-step (r-current-instruction (r), r))
      =  i (r->i (r),
             r-pushj-step-clock (r-current-instruction (r), r),
             load-addr))
```

THEOREM: r->i_pc-opener-in-popj-case

```
r->i_pc (fetch-adp (untag (r-tsp (r)), r-sys-data-segment (r)), r-prog-segment (r))
=  tag ('ipc,
        cons (area-name (fetch-adp (untag (r-tsp (r)), r-sys-data-segment (r))),
              find-label (untag (fetch-adp (untag (r-tsp (r)),
                                            r-sys-data-segment (r))),
                          cdr (icompile-program (definition (area-name (fetch-adp (untag (r-tsp (r)),
                                                                                   r-sys-data-segment (r))),
                                                            r-prog-segment (r)))))))
```

THEOREM: popj-one-way-correspondence-r-i

```
((r-psw (r) = 'run)
 ∧  (car (r-current-instruction (r)) = 'popj)
 ∧  proper-r-statep (r, load-addr)
 ∧  (load-addr ∈ N)
 ∧  r-popj-okp (r-current-instruction (r), r))
→  (r->i (r-popj-step (r-current-instruction (r), r))
      =  i (r->i (r),
             r-popj-step-clock (r-current-instruction (r), r),
             load-addr))
```

THEOREM: add-addr-one-way-correspondence-r-i

```
((r-psw (r) = 'run)
 ∧  (car (r-current-instruction (r)) = 'add-addr)
 ∧  proper-r-statep (r, load-addr)
 ∧  (load-addr ∈ N)
 ∧  r-add-addr-okp (r-current-instruction (r), r))
→  (r->i (r-add-addr-step (r-current-instruction (r), r))
      =  i (r->i (r),
             r-add-addr-step-clock (r-current-instruction (r), r),
             load-addr))
```

THEOREM: sub-addr-one-way-correspondence-r-i
$((\text{r-psw}\,(r) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'sub-addr})$
$\land \quad \text{proper-r-statep}\,(r,\ \textit{load-addr})$
$\land \quad (\textit{load-addr} \in \mathbf{N})$
$\land \quad \text{r-sub-addr-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\to \quad (\text{r->i}\,(\text{r-sub-addr-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad \text{r-sub-addr-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad \textit{load-addr}))$

```
; In all of the xor instructions we should be careful not to
; expand the xors.  This also holds for the other boolean
; instructions.  It especially holds for any instruction involving
; the hideous:
```

EVENT: Disable xor-xxx.

THEOREM: i-usr-data-typep-is-r-usr-data-typep
$\text{i-usr-data-typep}\,(x) = \text{r-usr-data-typep}\,(x)$

THEOREM: listp-cdr-assoc-deposit-adp
$\text{adpp}\,(\textit{adp},\ \textit{segment})$
$\to \quad (\text{listp}\,(\text{cdr}\,(\text{assoc}\,(\textit{name},\ \text{deposit-adp}\,(\textit{val},\ \textit{adp},\ \textit{segment}))))$
$\quad = \quad \text{listp}\,(\text{cdr}\,(\text{assoc}\,(\textit{name},\ \textit{segment}))))$

THEOREM: i-objectp-bool
$\text{i-objectp}\,(\text{bool}\,(x),\ s)$

THEOREM: type-bool
$\text{type}\,(\text{bool}\,(x)) = \texttt{'bool}$

THEOREM: cddr-bool
$\text{cddr}\,(\text{bool}\,(x)) = \mathbf{nil}$

THEOREM: booleanp-untag-bool
$\text{booleanp}\,(\text{untag}\,(\text{bool}\,(x)))$

THEOREM: equal-untag-bool-t
$(\text{untag}\,(\text{bool}\,(x)) = \texttt{'t})$
$= \quad \mathbf{if}\ x\ \mathbf{then\ t}$
$\qquad \mathbf{else\ f\ endif}$

THEOREM: equal-untag-bool-f
$(\text{untag}\,(\text{bool}\,(x)) = \texttt{'f}) = (\neg\ x)$

EVENT: Disable bool.

```
; Despite the disabling of bool above, we still sometimes
; see '(bool t) because we write it explicitly in some
; definitions and assembly code.
```

THEOREM: i-objectp-quote-bool
i-objectp (`'(bool t)`, s) $\land$ i-objectp (`'(bool f)`, s)

```
; The following lemma has nothing to do with fetch-adp
; but is so instantiated to control its consideration.

; We first prove the general version.
```

THEOREM: r-objectps-of-same-type-equal-iff-untags-equal
(r-objectp (x, r) $\land$ r-objectp (y, r) $\land$ (type (x) $=$ type (y)))
$\rightarrow$ ((untag (x) $=$ untag (y)) $=$ (x $=$ y))

```
; and then instantiate it:
```

THEOREM: r-objectps-of-same-type-equal-iff-untags-equal-rewrite
(r-objectp (fetch-adp (*adp1*, *segment1*), r)
$\land$ r-objectp (fetch-adp (*adp2*, *segment2*), r)
$\land$ (type (fetch-adp (*adp1*, *segment1*)) $=$ type (fetch-adp (*adp2*, *segment2*))))
$\rightarrow$ ((untag (fetch-adp (*adp1*, *segment1*)) $=$ untag (fetch-adp (*adp2*, *segment2*)))
$=$ (fetch-adp (*adp1*, *segment1*) $=$ fetch-adp (*adp2*, *segment2*)))

THEOREM: eq-one-way-correspondence-r-i
((r-psw (r) $=$ `'run`)
$\land$ (car (r-current-instruction (r)) $=$ `'eq`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ (*load-addr* $\in$ **N**)
$\land$ r-eq-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-eq-step (r-current-instruction (r), r))
$=$ i (r->i (r),
r-eq-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: lt-addr-one-way-correspondence-r-i
((r-psw (r) $=$ `'run`)
$\land$ (car (r-current-instruction (r)) $=$ `'lt-addr`)
$\land$ proper-r-statep (r, *load-addr*)

$\wedge$ $(\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge$ r-lt-addr-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-lt-addr-step (r-current-instruction (r), r))
    = i (r->i (r),
        r-lt-addr-step-clock (r-current-instruction (r), r),
        $\mathit{load\text{-}addr}$))

```
; I might need a more general version of the following lemma.
; But right now I need this and the more general versions I have
; thought of are messy because they introduce cases on the
; relative magnitudes of the increment, decrement, and initial
; offset.

; The hypothesis below is weaker than (adpp adp segment) because
; that stronger hyp has the free variable segment in it.
```

THEOREM: sub1-adp-add1-adp
(listp (adp) $\wedge$ (adp-offset (adp) $\in \mathbf{N}$))
$\rightarrow$ (sub-adp (add-adp $(\mathit{adp}, 1)$, 1) = adp)

```
; The following lemma has free vars in the hyp that cannot be
; relieved in the actual application needed, so I use lemma class
; nil and instantiate it,  backchaining on into the proper-r-statep.
```

THEOREM: tag-untag
(r-objectp (x, r) $\wedge$ (type (x) = type)) $\rightarrow$ (tag (type, untag (x)) = x)

THEOREM: tag-untag-r-tsp
proper-r-statep (r, $\mathit{load\text{-}addr}$)
$\rightarrow$ (tag (**'sys-addr**, untag (r-tsp (r))) = r-tsp (r))

THEOREM: fetch-one-way-correspondence-r-i
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'fetch**)
$\wedge$ proper-r-statep (r, $\mathit{load\text{-}addr}$)
$\wedge$ $(\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge$ r-fetch-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-fetch-step (r-current-instruction (r), r))
    = i (r->i (r),
        r-fetch-step-clock (r-current-instruction (r), r),
        $\mathit{load\text{-}addr}$))

THEOREM: deposit-one-way-correspondence-r-i

((r-psw (r) = '**run**)
 ∧ (car (r-current-instruction (r)) = '**deposit**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-deposit-okp (r-current-instruction (r), r))
→ (r->i (r-deposit-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-deposit-step-clock (r-current-instruction (r), r),
 load-addr))

THEOREM: fetch-temp-stk-one-way-correspondence-r-i
((r-psw (r) = '**run**)
 ∧ (car (r-current-instruction (r)) = '**fetch-temp-stk**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-fetch-temp-stk-okp (r-current-instruction (r), r))
→ (r->i (r-fetch-temp-stk-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-fetch-temp-stk-step-clock (r-current-instruction (r), r),
 load-addr))

THEOREM: deposit-temp-stk-one-way-correspondence-r-i
((r-psw (r) = '**run**)
 ∧ (car (r-current-instruction (r)) = '**deposit-temp-stk**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-deposit-temp-stk-okp (r-current-instruction (r), r))
→ (r->i (r-deposit-temp-stk-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-deposit-temp-stk-step-clock (r-current-instruction (r), r),
 load-addr))

```
; Here is the second of the promised sub-adp-add-adp lemmas.
```

THEOREM: sub1-adp-add2-adp
(listp (*adp*) ∧ (adp-offset (*adp*) $\in$ **N**))
→ (sub-adp (add-adp (*adp*, 2), 1) = add-adp (*adp*, 1))

```
; The following simple identity needs the hypothesis that x is an integerp.
; But since we have that it is actually a small-integerp, I'll use that
; and save the trouble of backchaining through small-integerp implies
; integerp.
```

THEOREM: iplus-0{special}
small-integerp (x, *word-size*) $\rightarrow$ (iplus (0, x) = x)

EVENT: Enable commutativity-of-iplus.

THEOREM: small-integerp-fix-small-integer-plus-case
(small-integerp (x, *word-size*) $\wedge$ small-integerp (y, *word-size*))
$\rightarrow$ small-integerp (fix-small-integer (iplus (bool-to-nat (c), iplus (x, y)),
word-size),
word-size)

THEOREM: small-integerp-fix-small-integer-difference-case
(small-integerp (x, *word-size*) $\wedge$ small-integerp (y, *word-size*))
$\rightarrow$ small-integerp (fix-small-integer (idifference (x,
iplus (y, bool-to-nat (c))),
word-size),
word-size)

EVENT: Disable fix-small-integer.

EVENT: Disable bool-to-nat.

EVENT: Disable iplus.

EVENT: Disable idifference.

EVENT: Disable ilessp.

EVENT: Disable integerp.

EVENT: Disable small-integerp.

THEOREM: add-int-one-way-correspondence-r-i
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'add-int**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ (*load-addr* $\in \mathbf{N}$)
$\wedge$ r-add-int-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-add-int-step (r-current-instruction (r), r))
= i (r->i (r),

r-add-int-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: add-int-with-carry-one-way-correspondence-r-i
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'add-int-with-carry`)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-add-int-with-carry-okp (r-current-instruction (r), r))
→ (r->i (r-add-int-with-carry-step (r-current-instruction (r), r))
= i (r->i (r),
r-add-int-with-carry-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: add1-int-one-way-correspondence-r-i
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'add1-int`)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-add1-int-okp (r-current-instruction (r), r))
→ (r->i (r-add1-int-step (r-current-instruction (r), r))
= i (r->i (r),
r-add1-int-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: sub-int-one-way-correspondence-r-i
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'sub-int`)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-sub-int-okp (r-current-instruction (r), r))
→ (r->i (r-sub-int-step (r-current-instruction (r), r))
= i (r->i (r),
r-sub-int-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: sub-int-with-carry-one-way-correspondence-r-i
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'sub-int-with-carry`)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-sub-int-with-carry-okp (r-current-instruction (r), r))
→ (r->i (r-sub-int-with-carry-step (r-current-instruction (r), r))
= i (r->i (r),
r-sub-int-with-carry-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: sub1-int-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'sub1-int**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* ∈ **N**)
 ∧ r-sub1-int-okp (r-current-instruction (r), r))
→ (r->i (r-sub1-int-step (r-current-instruction (r), r))
   = i (r->i (r),
     r-sub1-int-step-clock (r-current-instruction (r), r),
     load-addr))

THEOREM: neg-int-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'neg-int**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* ∈ **N**)
 ∧ r-neg-int-okp (r-current-instruction (r), r))
→ (r->i (r-neg-int-step (r-current-instruction (r), r))
   = i (r->i (r),
     r-neg-int-step-clock (r-current-instruction (r), r),
     load-addr))

EVENT: Disable xor-bool.

THEOREM: not-empty-stkp-implies-adpp-pop
(adpp (untag (*sp*), *segment*) ∧ (¬ empty-stkp (*sp*, *segment*)))
→ adpp (add-adp (untag (*sp*), 1), *segment*)

```
; We are about to go into a really silly little section.  We need to
; know that the word-size is not 0.  To get that, we
; use R-LOADABLEP and the fact that the sys-data-segment is
; non-empty.
```

THEOREM: r-loadablep-and-cstk-implies-not-zerop-r-word-size
(proper-r-statep (r, *load-addr*)
 ∧ (adp-offset (untag (r-cfp (r)))
   < length (cdr (assoc (**'cstk**, r-sys-data-segment (r))))))
→ (r-word-size (r) $\not\simeq$ 0)

THEOREM: proper-r-statep-implies-r-loadablep-and-cstk
proper-r-statep (r, *load-addr*)
→ (r-loadablep (r, *load-addr*)
   ∧ (adp-offset (untag (r-cfp (r)))
     < length (cdr (assoc (**'cstk**, r-sys-data-segment (r))))))

THEOREM: proper-r-statep-implies-lessp-0-r-word-size
proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\rightarrow$ ((r-word-size $(r) \in \mathbf{N}) \land (0 <$ r-word-size $(r)))$

```
; We now prove the key fact:  (ilessp x y) is equivalent to
; the xor of the v-flg and the n-flg.  Because of case splits
; caused by the paths through the icode-lt-int code, we need to
; prove this fact in 4 different versions, as though xor had
; opened up:
```

THEOREM: ilessp-is-xor-case1
(small-integerp $(x,\ \mathit{word\text{-}size})$
$\land$ small-integerp $(y,\ \mathit{word\text{-}size})$
$\land$ $(\mathit{word\text{-}size} \not\simeq 0)$
$\land$ small-integerp (idifference $(x,\ y)$, $\mathit{word\text{-}size}$)
$\land$ negativep (fix-small-integer (idifference $(x,\ y)$, $\mathit{word\text{-}size}$)))
$\rightarrow$ ilessp $(x,\ y)$

THEOREM: ilessp-is-xor-case2
(small-integerp $(x,\ \mathit{word\text{-}size})$
$\land$ small-integerp $(y,\ \mathit{word\text{-}size})$
$\land$ $(\mathit{word\text{-}size} \not\simeq 0)$
$\land$ $(\neg$ small-integerp (idifference $(x,\ y)$, $\mathit{word\text{-}size}$))
$\land$ negativep (fix-small-integer (idifference $(x,\ y)$, $\mathit{word\text{-}size}$)))
$\rightarrow$ $(\neg$ ilessp $(x,\ y))$

THEOREM: ilessp-is-xor-case3
(small-integerp $(x,\ \mathit{word\text{-}size})$
$\land$ small-integerp $(y,\ \mathit{word\text{-}size})$
$\land$ $(\mathit{word\text{-}size} \not\simeq 0)$
$\land$ small-integerp (idifference $(x,\ y)$, $\mathit{word\text{-}size}$)
$\land$ $(\neg$ negativep (fix-small-integer (idifference $(x,\ y)$, $\mathit{word\text{-}size}$))))
$\rightarrow$ $(\neg$ ilessp $(x,\ y))$

THEOREM: ilessp-is-xor-case4
(small-integerp $(x,\ \mathit{word\text{-}size})$
$\land$ small-integerp $(y,\ \mathit{word\text{-}size})$
$\land$ $(\mathit{word\text{-}size} \not\simeq 0)$
$\land$ $(\neg$ small-integerp (idifference $(x,\ y)$, $\mathit{word\text{-}size}$))
$\land$ $(\neg$ negativep (fix-small-integer (idifference $(x,\ y)$, $\mathit{word\text{-}size}$))))
$\rightarrow$ ilessp $(x,\ y)$

THEOREM: lt-int-one-way-correspondence-r-i
((r-psw (r) = **'run**)

$\land$ (car (r-current-instruction (r)) = `'lt-int`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ (*load-addr* $\in \mathbf{N}$)
$\land$ r-lt-int-okp (r-current-instruction (r), r))
$\to$ (r->i (r-lt-int-step (r-current-instruction (r), r))
= i (r->i (r),
r-lt-int-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: int-to-nat-one-way-correspondence-r-i
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'int-to-nat`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ (*load-addr* $\in \mathbf{N}$)
$\land$ r-int-to-nat-okp (r-current-instruction (r), r))
$\to$ (r->i (r-int-to-nat-step (r-current-instruction (r), r))
= i (r->i (r),
r-int-to-nat-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: add-nat-one-way-correspondence-r-i
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'add-nat`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ (*load-addr* $\in \mathbf{N}$)
$\land$ r-add-nat-okp (r-current-instruction (r), r))
$\to$ (r->i (r-add-nat-step (r-current-instruction (r), r))
= i (r->i (r),
r-add-nat-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: lessp-remainder{special}
$(y \not\simeq 0) \to ((x \textbf{ mod } y) < y)$

THEOREM: add-nat-with-carry-one-way-correspondence-r-i
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'add-nat-with-carry`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ (*load-addr* $\in \mathbf{N}$)
$\land$ r-add-nat-with-carry-okp (r-current-instruction (r), r))
$\to$ (r->i (r-add-nat-with-carry-step (r-current-instruction (r), r))
= i (r->i (r),
r-add-nat-with-carry-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: add1-nat-one-way-correspondence-r-i
((r-psw (r) = 'run)
∧ (car (r-current-instruction (r)) = 'add1-nat)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-add1-nat-okp (r-current-instruction (r), r))
→ (r->i (r-add1-nat-step (r-current-instruction (r), r))
= i (r->i (r),
r-add1-nat-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: sub-nat-one-way-correspondence-r-i
((r-psw (r) = 'run)
∧ (car (r-current-instruction (r)) = 'sub-nat)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-sub-nat-okp (r-current-instruction (r), r))
→ (r->i (r-sub-nat-step (r-current-instruction (r), r))
= i (r->i (r),
r-sub-nat-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: sub-nat-with-carry-one-way-correspondence-r-i
((r-psw (r) = 'run)
∧ (car (r-current-instruction (r)) = 'sub-nat-with-carry)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-sub-nat-with-carry-okp (r-current-instruction (r), r))
→ (r->i (r-sub-nat-with-carry-step (r-current-instruction (r), r))
= i (r->i (r),
r-sub-nat-with-carry-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: sub1-nat-one-way-correspondence-r-i
((r-psw (r) = 'run)
∧ (car (r-current-instruction (r)) = 'sub1-nat)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in$ **N**)
∧ r-sub1-nat-okp (r-current-instruction (r), r))
→ (r->i (r-sub1-nat-step (r-current-instruction (r), r))
= i (r->i (r),
r-sub1-nat-step-clock (r-current-instruction (r), r),
load-addr))

THEOREM: lt-nat-one-way-correspondence-r-i

$((\text{r-psw}(r) = \texttt{'run})$
$\wedge$ $(\text{car}(\text{r-current-instruction}(r)) = \texttt{'lt-nat})$
$\wedge$ $\text{proper-r-statep}(r,\ \textit{load-addr})$
$\wedge$ $(\textit{load-addr} \in \mathbf{N})$
$\wedge$ $\text{r-lt-nat-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow$ $(\text{r->i}(\text{r-lt-nat-step}(\text{r-current-instruction}(r),\ r))$
$= \ \text{i}(\text{r->i}(r),$
$\text{r-lt-nat-step-clock}(\text{r-current-instruction}(r),\ r),$
$\textit{load-addr}))$

THEOREM: mult2-nat-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\wedge$ $(\text{car}(\text{r-current-instruction}(r)) = \texttt{'mult2-nat})$
$\wedge$ $\text{proper-r-statep}(r,\ \textit{load-addr})$
$\wedge$ $(\textit{load-addr} \in \mathbf{N})$
$\wedge$ $\text{r-mult2-nat-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow$ $(\text{r->i}(\text{r-mult2-nat-step}(\text{r-current-instruction}(r),\ r))$
$= \ \text{i}(\text{r->i}(r),$
$\text{r-mult2-nat-step-clock}(\text{r-current-instruction}(r),\ r),$
$\textit{load-addr}))$

THEOREM: mult2-nat-with-carry-out-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\wedge$ $(\text{car}(\text{r-current-instruction}(r)) = \texttt{'mult2-nat-with-carry-out})$
$\wedge$ $\text{proper-r-statep}(r,\ \textit{load-addr})$
$\wedge$ $(\textit{load-addr} \in \mathbf{N})$
$\wedge$ $\text{r-mult2-nat-with-carry-out-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow$ $(\text{r->i}(\text{r-mult2-nat-with-carry-out-step}(\text{r-current-instruction}(r),\ r))$
$= \ \text{i}(\text{r->i}(r),$
$\text{r-mult2-nat-with-carry-out-step-clock}(\text{r-current-instruction}(r),$
$r),$
$\textit{load-addr}))$

THEOREM: lessp-quotient{special}
$x \not< (x \div y)$

```
; The following lemma is just an instance of the much prettier fact
; that (remainder x 2) is 0 or 1.  But I couldn't think of a way to
; state that fact without causing a lot of backchaining.  So I'll
; just do it this way.
```

THEOREM: remainder-by-2-not-1-lemma
$((\text{untag}(\text{fetch-adp}(\text{untag}(\text{r-tsp}(r)),\ \text{r-sys-data-segment}(r)))\ \mathbf{mod}\ 2) \neq 1)$
$\rightarrow$ $((\text{untag}(\text{fetch-adp}(\text{untag}(\text{r-tsp}(r)),\ \text{r-sys-data-segment}(r)))\ \mathbf{mod}\ 2)$
$= \ 0)$

THEOREM: exp-2-at-least-2
$(i \not\simeq 0) \rightarrow (1 < \exp(2,\ i))$

THEOREM: i-objectp-nat-0
$(0 < \text{i-word-size}(i)) \rightarrow \text{i-objectp}(\texttt{'(nat 0)},\ i)$

THEOREM: i-objectp-nat-1
$(0 < \text{i-word-size}(i)) \rightarrow \text{i-objectp}(\texttt{'(nat 1)},\ i)$

; Thus ends the silly little section.

THEOREM: div2-nat-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\wedge\ (\text{car}(\text{r-current-instruction}(r)) = \texttt{'div2-nat})$
$\wedge\ \text{proper-r-statep}(r,\ \textit{load-addr})$
$\wedge\ (\textit{load-addr} \in \mathbf{N})$
$\wedge\ \text{r-div2-nat-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ (\text{r->i}(\text{r-div2-nat-step}(\text{r-current-instruction}(r),\ r))$
$=\ \text{i}(\text{r->i}(r),$
$\text{r-div2-nat-step-clock}(\text{r-current-instruction}(r),\ r),$
$\textit{load-addr}))$

THEOREM: or-bitv-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\wedge\ (\text{car}(\text{r-current-instruction}(r)) = \texttt{'or-bitv})$
$\wedge\ \text{proper-r-statep}(r,\ \textit{load-addr})$
$\wedge\ (\textit{load-addr} \in \mathbf{N})$
$\wedge\ \text{r-or-bitv-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ (\text{r->i}(\text{r-or-bitv-step}(\text{r-current-instruction}(r),\ r))$
$=\ \text{i}(\text{r->i}(r),$
$\text{r-or-bitv-step-clock}(\text{r-current-instruction}(r),\ r),$
$\textit{load-addr}))$

THEOREM: and-bitv-one-way-correspondence-r-i
$((\text{r-psw}(r) = \texttt{'run})$
$\wedge\ (\text{car}(\text{r-current-instruction}(r)) = \texttt{'and-bitv})$
$\wedge\ \text{proper-r-statep}(r,\ \textit{load-addr})$
$\wedge\ (\textit{load-addr} \in \mathbf{N})$
$\wedge\ \text{r-and-bitv-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ (\text{r->i}(\text{r-and-bitv-step}(\text{r-current-instruction}(r),\ r))$
$=\ \text{i}(\text{r->i}(r),$
$\text{r-and-bitv-step-clock}(\text{r-current-instruction}(r),\ r),$
$\textit{load-addr}))$

THEOREM: not-bitv-one-way-correspondence-r-i

$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge\quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'not-bitv})$
$\wedge\quad \text{proper-r-statep}\,(r,\ \textit{load-addr})$
$\wedge\quad (\textit{load-addr} \in \mathbf{N})$
$\wedge\quad \text{r-not-bitv-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow\quad (\text{r->i}\,(\text{r-not-bitv-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad\qquad \text{r-not-bitv-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad\qquad \textit{load-addr}))$

THEOREM: xor-bitv-one-way-correspondence-r-i

$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge\quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'xor-bitv})$
$\wedge\quad \text{proper-r-statep}\,(r,\ \textit{load-addr})$
$\wedge\quad (\textit{load-addr} \in \mathbf{N})$
$\wedge\quad \text{r-xor-bitv-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow\quad (\text{r->i}\,(\text{r-xor-bitv-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad\qquad \text{r-xor-bitv-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad\qquad \textit{load-addr}))$

THEOREM: rsh-bitv-one-way-correspondence-r-i

$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge\quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'rsh-bitv})$
$\wedge\quad \text{proper-r-statep}\,(r,\ \textit{load-addr})$
$\wedge\quad (\textit{load-addr} \in \mathbf{N})$
$\wedge\quad \text{r-rsh-bitv-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow\quad (\text{r->i}\,(\text{r-rsh-bitv-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad\qquad \text{r-rsh-bitv-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad\qquad \textit{load-addr}))$

THEOREM: lsh-bitv-one-way-correspondence-r-i

$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge\quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'lsh-bitv})$
$\wedge\quad \text{proper-r-statep}\,(r,\ \textit{load-addr})$
$\wedge\quad (\textit{load-addr} \in \mathbf{N})$
$\wedge\quad \text{r-lsh-bitv-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow\quad (\text{r->i}\,(\text{r-lsh-bitv-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad\qquad \text{r-lsh-bitv-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad\qquad \textit{load-addr}))$

EVENT: Disable or-bool.

EVENT: Disable and-bool.

EVENT: Disable not-bool.

THEOREM: or-bool-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'or-bool**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-or-bool-okp (r-current-instruction (r), r))
→ (r->i (r-or-bool-step (r-current-instruction (r), r))
  = i (r->i (r),
   r-or-bool-step-clock (r-current-instruction (r), r),
   load-addr))

THEOREM: and-bool-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'and-bool**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-and-bool-okp (r-current-instruction (r), r))
→ (r->i (r-and-bool-step (r-current-instruction (r), r))
  = i (r->i (r),
   r-and-bool-step-clock (r-current-instruction (r), r),
   load-addr))

THEOREM: xor-bool-t-is-not-bool
xor-bool (x, **'t**) = not-bool (x)

THEOREM: not-bool-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'not-bool**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in$ **N**)
 ∧ r-not-bool-okp (r-current-instruction (r), r))
→ (r->i (r-not-bool-step (r-current-instruction (r), r))
  = i (r->i (r),
   r-not-bool-step-clock (r-current-instruction (r), r),
   load-addr))

THEOREM: jump-if-temp-stk-full-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'jump-if-temp-stk-full**)
 ∧ proper-r-statep (r, *load-addr*)

$\wedge$ ($load\text{-}addr \in \mathbf{N}$)
$\wedge$ r-jump-if-temp-stk-full-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-jump-if-temp-stk-full-step (r-current-instruction (r), r))
= i (r->i (r),
r-jump-if-temp-stk-full-step-clock (r-current-instruction (r),
r),
$load\text{-}addr$))

THEOREM: jump-if-temp-stk-empty-one-way-correspondence-r-i
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'jump-if-temp-stk-empty`)
$\wedge$ proper-r-statep (r, $load\text{-}addr$)
$\wedge$ ($load\text{-}addr \in \mathbf{N}$)
$\wedge$ r-jump-if-temp-stk-empty-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-jump-if-temp-stk-empty-step (r-current-instruction (r), r))
= i (r->i (r),
r-jump-if-temp-stk-empty-step-clock (r-current-instruction (r),
r),
$load\text{-}addr$))

THEOREM: test-nat-and-jump-one-way-correspondence-r-i
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'test-nat-and-jump`)
$\wedge$ proper-r-statep (r, $load\text{-}addr$)
$\wedge$ ($load\text{-}addr \in \mathbf{N}$)
$\wedge$ r-test-nat-and-jump-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-test-nat-and-jump-step (r-current-instruction (r), r))
= i (r->i (r),
r-test-nat-and-jump-step-clock (r-current-instruction (r), r),
$load\text{-}addr$))

```
; The following lemma recognizes a certain opened up
; r->i_pc and folds it back together.  This is needed
; in test-int-and-jump.
```

THEOREM: r->i_pc-folder-test-int-and-jump-case
cons (adp-name (untag (r-pc (r))),
find-label (add-adp (untag (r-pc (r)), `1`),
cdr (icompile-program (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))))
= untag (r->i_pc (add1-addr (r-pc (r)), r-prog-segment (r)))

THEOREM: small-integerp-subtypes
(small-integerp (untag (obj), $word\text{-}size$) $\wedge$ (untag (obj) $\notin \mathbf{N}$))
$\rightarrow$ negativep (untag (obj))

THEOREM: test-int-and-jump-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'test-int-and-jump**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ $(\mathit{load\text{-}addr} \in \mathbf{N})$
 ∧ r-test-int-and-jump-okp (r-current-instruction $(r),\ r)$)
→ (r->i (r-test-int-and-jump-step (r-current-instruction $(r),\ r)$)
 = i (r->i (r),
 r-test-int-and-jump-step-clock (r-current-instruction $(r),\ r)$,
 load-addr))

```
; The following lemma makes the arbitrary convention that we'll represent
; all comparisions on Booleans as comparisons with 'F.
```

THEOREM: booleanp-subtypes
booleanp (untag (obj)) → ((untag (obj) = **'t**) = (untag (obj) ≠ **'f**))

```
; Now we'll pay for that decision:
```

THEOREM: not-bool-of-non-f
(untag (obj) ≠ **'f**) → (not-bool (untag (obj)) = **'f**)

THEOREM: test-bool-and-jump-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'test-bool-and-jump**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ $(\mathit{load\text{-}addr} \in \mathbf{N})$
 ∧ r-test-bool-and-jump-okp (r-current-instruction $(r),\ r)$)
→ (r->i (r-test-bool-and-jump-step (r-current-instruction $(r),\ r)$)
 = i (r->i (r),
 r-test-bool-and-jump-step-clock (r-current-instruction $(r),\ r)$,
 load-addr))

THEOREM: test-bitv-and-jump-one-way-correspondence-r-i
((r-psw (r) = **'run**)
 ∧ (car (r-current-instruction (r)) = **'test-bitv-and-jump**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ $(\mathit{load\text{-}addr} \in \mathbf{N})$
 ∧ r-test-bitv-and-jump-okp (r-current-instruction $(r),\ r)$)
→ (r->i (r-test-bitv-and-jump-step (r-current-instruction $(r),\ r)$)
 = i (r->i (r),
 r-test-bitv-and-jump-step-clock (r-current-instruction $(r),\ r)$,
 load-addr))

THEOREM: pop-locn-one-way-correspondence-r-i
$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'pop-locn})$
$\wedge \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})$
$\wedge \quad (\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge \quad \text{r-pop-locn-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow \quad (\text{r->i}\,(\text{r-pop-locn-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad \text{r-pop-locn-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad \mathit{load\text{-}addr}))$

THEOREM: locn-one-way-correspondence-r-i
$((\text{r-psw}\,(r) = \texttt{'run})$
$\wedge \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'locn})$
$\wedge \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})$
$\wedge \quad (\mathit{load\text{-}addr} \in \mathbf{N})$
$\wedge \quad \text{r-locn-okp}\,(\text{r-current-instruction}\,(r),\ r))$
$\rightarrow \quad (\text{r->i}\,(\text{r-locn-step}\,(\text{r-current-instruction}\,(r),\ r))$
$\quad = \quad \text{i}\,(\text{r->i}\,(r),$
$\qquad \text{r-locn-step-clock}\,(\text{r-current-instruction}\,(r),\ r),$
$\qquad \mathit{load\text{-}addr}))$

```
; The code generated for the jump-case instruction pops a number, n,
; off the stack and doubles it, as part of the computation of the
; index into a jump table.  We must prove that 2n is representable.
; We do it by observing that 2n is less than the number of labels
; supplied in the jump-case, that the number of labels is dominated
; by the length of the icode, that the length of the icode for an instruction
; is dominated by the length of the icompile-program, that the length
; of the program is dominated by the segment-length, and that the
; segment length is representable.
```

THEOREM: length-jump_*-lst
$\text{length}\,(\text{jump\_*-lst}\,(\mathit{labs},\ \mathit{program})) = (2 * \text{length}\,(\mathit{labs}))$

THEOREM: jump-case-instruction-is-much-shorter-than-icode
$(\text{car}\,(\text{unlabel}\,(\mathit{ins})) = \texttt{'jump-case})$
$\rightarrow \quad ((2 * \text{length}\,(\text{cdr}\,(\text{unlabel}\,(\mathit{ins})))) < \text{length}\,(\text{icode}\,(\mathit{ins},\ \mathit{pcn},\ \mathit{body})))$

THEOREM: icode-is-shorter-than-icompile-program
$((n < \text{length}\,(\text{program-body}\,(\mathit{prog}))) \wedge (n \in \mathbf{N}))$
$\rightarrow \quad (\text{length}\,(\text{icode}\,(\text{get}\,(n,\ \text{program-body}\,(\mathit{prog})),\ n,\ \mathit{prog}))$
$\quad < \quad \text{length}\,(\text{icompile-program}\,(\mathit{prog})))$

```
; In the following theorem, the (untag (fetch-adp ...)) could
; be generalized to the variable case-number.
```

THEOREM: jump-case-number-is-very-small
(proper-r-statep (r, *load-addr*)
 ∧ (car (unlabel (get (adp-offset (untag (r-pc (r))),
 program-body (assoc (adp-name (untag (r-pc (r))),
 r-prog-segment (r))))))
 = `'jump-case`)
 ∧ (untag (fetch-adp (untag (r-tsp (r)), r-sys-data-segment (r)))
 < length (cdr (unlabel (get (adp-offset (untag (r-pc (r))),
 program-body (assoc (adp-name (untag (r-pc (r))),
 r-prog-segment (r))))))))
→ ((2 $*$ untag (fetch-adp (untag (r-tsp (r)), r-sys-data-segment (r))))
 < exp (2, r-word-size (r)))

THEOREM: times-0
$(z \simeq 0) \rightarrow ((x * z) = 0)$

THEOREM: get-2n-jump_*-lst
(n < length (*labs*))
→ (get (2 $*$ n, jump_*-lst (*labs*, *prog*)) = `'(jump_*)`)

THEOREM: get-add1-2n-jump_*-lst
(n < length (*labs*))
→ (get (1 + (2 $*$ n), jump_*-lst (*labs*, *prog*)) = pc (get (n, *labs*), *prog*))

THEOREM: jump-case-one-way-correspondence-r-i
((r-psw (r) = `'run`)
 ∧ (car (r-current-instruction (r)) = `'jump-case`)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ (*load-addr* $\in \mathbf{N}$)
 ∧ r-jump-case-okp (r-current-instruction (r), r))
→ (r->i (r-jump-case-step (r-current-instruction (r), r))
 = i (r->i (r),
 r-jump-case-step-clock (r-current-instruction (r), r),
 load-addr))

THEOREM: find-label-prelude
(*name* = name (*program*))
→ (find-label (cons (*name*, `'(prelude)`), cdr (icompile-program (*program*)))
 = 0)

THEOREM: fetch-adp-prelude-0

$\text{definedp}\,(\mathit{name},\ \mathit{prog\text{-}segment})$
$\rightarrow\quad (\text{fetch-adp}\,(\text{cons}\,(\mathit{name},\ \mathtt{0}),\ \text{icompile}\,(\mathit{prog\text{-}segment}))$
$\quad = \quad \text{dl}\,(\text{cons}\,(\mathit{name},\ \mathtt{'(prelude)}),\ \mathtt{'(prelude)},\ \mathtt{'(cpush\_cfp)}))$

THEOREM: fetch-adp-prelude-1
$\text{definedp}\,(\mathit{name},\ \mathit{prog\text{-}segment})$
$\rightarrow\quad (\text{fetch-adp}\,(\text{cons}\,(\mathit{name},\ \mathtt{1}),\ \text{icompile}\,(\mathit{prog\text{-}segment}))$
$\quad = \quad \mathtt{'(move\_cfp\_csp)})$

THEOREM: adpp-prelude
$(\text{definedp}\,(\mathit{name},\ \mathit{prog\text{-}segment})$
$\wedge\quad (n \in \mathbf{N})$
$\wedge\quad (\text{length}\,(\text{generate-prelude}\,(\text{assoc}\,(\mathit{name},\ \mathit{prog\text{-}segment}))) \not< n))$
$\rightarrow\quad \text{adpp}\,(\text{cons}\,(\mathit{name},\ n),\ \text{icompile}\,(\mathit{prog\text{-}segment}))$

EVENT: Enable length-generate-prelude.

THEOREM: i*-plus
$\text{i*}\,(s,\ i + j,\ \mathit{load\text{-}addr}) = \text{i*}\,(\text{i*}\,(s,\ i,\ \mathit{load\text{-}addr}),\ j,\ \mathit{load\text{-}addr})$

```
; The following theorem is just an instance of the i*-plus theorem above.
; However, I don't want the i*-plus theorem as a rewrite rule because I
; don't want the i* expression rewritten until we have done all the work
; we can.
```

THEOREM: i*-plus-prelude1-prelude2
$\text{i*}\,(s,\ \text{length}\,(\text{temp-var-dcls}\,(\mathit{prog})) + \text{length}\,(\text{formal-vars}\,(\mathit{prog})),\ \mathit{load\text{-}addr})$
$= \quad \text{i*}\,(\text{i*}\,(s,\ \text{length}\,(\text{temp-var-dcls}\,(\mathit{prog})),\ \mathit{load\text{-}addr}),$
$\qquad \text{length}\,(\text{formal-vars}\,(\mathit{prog})),$
$\qquad \mathit{load\text{-}addr})$

```
; We now wish to package up the execution of prelude1.  See i*-prelude1
; below.  To prove it, we must prove the more general i*-prelude1-generalized

; We start by establishing what fetch-adp returns on both even and odd
; addresses into the prelude1.  The main theorems of this section are
; fetch-adp-prelude1-even and fetch-adp-prelude1-odd.
```

THEOREM: sub-adp-0
$\text{adpp}\,(x,\ \mathit{segment}) \rightarrow (\text{sub-adp}\,(x,\ \mathtt{0}) = x)$

THEOREM: tag-type-untag
$((\text{type}\,(x) = \mathit{type}) \wedge (\text{cddr}\,(x) = \mathbf{nil})) \rightarrow (\text{tag}\,(\mathit{type},\ \text{untag}\,(x)) = x)$

THEOREM: adpp-sub-adp
$\text{adpp}(x,\ \textit{segment}) \rightarrow \text{adpp}(\text{sub-adp}(x,\ \mathtt{1}),\ \textit{segment})$

THEOREM: fetch-adp-prelude1-even-lemma
$(n < \text{length}(\textit{rev-temp-var-dcls}))$
$\rightarrow \quad (\text{car}(\text{cdr-n}(n + n,\ \text{generate-prelude1}(\textit{rev-temp-var-dcls})))$
$\qquad = \quad \mathtt{'(cpush\_*)})$

THEOREM: fetch-adp-prelude1-even
$(\text{definedp}(\textit{name},\ \textit{prog-segment})$
$\ \wedge \quad (n < \text{length}(\text{temp-var-dcls}(\text{assoc}(\textit{name},\ \textit{prog-segment})))))$
$\rightarrow \quad (\text{fetch-adp}(\text{cons}(\textit{name},\ 1 + (1 + (n + n))),\ \text{icompile}(\textit{prog-segment}))$
$\qquad = \quad \mathtt{'(cpush\_*)})$

THEOREM: listp-rev
$\text{listp}(\text{rev}(x)) = \text{listp}(x)$

THEOREM: get-append
$(n < \text{length}(a)) \rightarrow (\text{get}(n,\ \text{append}(a,\ b)) = \text{get}(n,\ a))$

THEOREM: fetch-adp-prelude1-odd-lemma
$(n < \text{length}(\textit{rev-temp-var-dcls}))$
$\rightarrow \quad (\text{get}(1 + (n + n),\ \text{generate-prelude1}(\textit{rev-temp-var-dcls}))$
$\qquad = \quad \text{cadr}(\text{get}(n,\ \textit{rev-temp-var-dcls})))$

THEOREM: fetch-adp-prelude1-odd
$(\text{definedp}(\textit{name},\ \textit{prog-segment})$
$\ \wedge \quad (n < \text{length}(\text{temp-var-dcls}(\text{assoc}(\textit{name},\ \textit{prog-segment})))))$
$\rightarrow \quad (\text{fetch-adp}(\text{cons}(\textit{name},\ 1 + (1 + (1 + (n + n)))),$
$\qquad\qquad\qquad \text{icompile}(\textit{prog-segment}))$
$\qquad = \quad \text{cadr}(\text{get}(n,\ \text{rev}(\text{temp-var-dcls}(\text{assoc}(\textit{name},\ \textit{prog-segment}))))))$

```
; We need establish that what the CPUSH_* pushes is an i-objectp
; in the current i-state.  We know that everything r-call-step
; pushes is an r-objectp in the initial r state.  We need to make
; the connections.  The key here is our previously proved lemma
; that r-objectp-implies-i-objectp.  We first establish that what
; is pushed at the i-level is an r-object.
```

THEOREM: adpp-implies-numberp-adp-offset
$\text{adpp}(x,\ \textit{segment}) \rightarrow (\text{adp-offset}(x) \in \mathbf{N})$

THEOREM: all-r-objectps-implies-r-objectp-get
$((n < \text{length}(\textit{var-dcls})) \wedge \text{all-r-objectps}(\text{strip-cadrs}(\textit{var-dcls}),\ r))$
$\rightarrow \quad \text{r-objectp}(\text{cadr}(\text{get}(n,\ \textit{var-dcls})),\ r)$

```
; Unfortunately, what the icompiler lays down is not elements of
; (strip-cadrs var-dcls) but elements of its reverse.  So we need
; to prove that the reverse doesn't affect the argument above.
```

THEOREM: strip-cadrs-append
strip-cadrs (append (a, b)) $=$ append (strip-cadrs (a), strip-cadrs (b))

THEOREM: all-r-objectps-append
properp (a)
$\rightarrow$ (all-r-objectps (append (a, b), r)
$=$ (all-r-objectps (a, r) $\wedge$ all-r-objectps (b, r)))

THEOREM: properp-strip-cadrs
properp (strip-cadrs (x))

THEOREM: all-r-objectps-strip-cadrs-rev
all-r-objectps (strip-cadrs (rev (x)), r) $=$ all-r-objectps (strip-cadrs (x), r)

```
; The work above is sufficient to relieve the r-objectp hyp
; in r-objectp-implies-i-objectp.

; We also have to establish that the signature of the r state
; is the same as that of the i state.  This follows from:
```

THEOREM: same-signature-deposit-adp
(litatom (adp-name (*adp*)) $\wedge$ adpp (*adp*, *segment2*))
$\rightarrow$ (same-signature (*segment1*, deposit-adp (*val*, *adp*, *segment2*))
$=$ same-signature (*segment1*, *segment2*))

```
; Now once the i machine has fetched the element it UNLABELs it.
; We need to show that it is not a DL.  That follows from the
; fact that we have a proper-r-statep which thus contains
; a r-programp which thus contains r-temp-var-dclsp,
; none of which is a DL.  Again we have to worry about the
; reverse.
```

EVENT: Enable member-append.

THEOREM: member-rev
($x \in$ rev (l)) $=$ ($x \in l$)

THEOREM: unlabel-cadr-get-rev-temp-var-dcls-lemma
((*var-dcl* $\in$ *temp-var-dcls*) $\wedge$ r-temp-var-dclsp (*temp-var-dcls*))
$\rightarrow$ (unlabel (cadr (*var-dcl*)) $=$ cadr (*var-dcl*))

THEOREM: unconditional-get-append
$$\begin{aligned}&\text{get}(n, \text{append}(a, b))\\ =\ &\textbf{if } n < \text{length}(a) \ \textbf{then } \text{get}(n, a)\\ &\textbf{else } \text{get}(n - \text{length}(a), b) \ \textbf{endif}\end{aligned}$$

THEOREM: get-rev
$$(n < \text{length}(l)) \rightarrow (\text{get}(n, \text{rev}(l)) = \text{get}((\text{length}(l) - n) - 1, l))$$

EVENT: Disable unconditional-get-append.

THEOREM: member-get-rev
$$(n < \text{length}(l)) \rightarrow (\text{get}(n, \text{rev}(l)) \in l)$$

THEOREM: unlabel-cadr-get-rev-temp-var-dcls
$$\begin{aligned}&(\text{proper-r-statep}(r, \textit{load-addr})\\ \wedge\ &\text{definedp}(\textit{name}, \text{r-prog-segment}(r))\\ \wedge\ &(n < \text{length}(\text{temp-var-dcls}(\text{assoc}(\textit{name}, \text{r-prog-segment}(r))))))\\ \rightarrow\ &(\text{unlabel}(\text{cadr}(\text{get}(n, \text{rev}(\text{temp-var-dcls}(\text{assoc}(\textit{name}, \text{r-prog-segment}(r)))))))\\ &\quad = \ \text{cadr}(\text{get}(n, \text{rev}(\text{temp-var-dcls}(\text{assoc}(\textit{name}, \text{r-prog-segment}(r)))))))\end{aligned}$$

```
; In the prelude1 proof we have difference disabled and the following lemma
; comes in at just the right place and reduces the pc in the induction
; conclusion to that in the induction hypothesis.
```

THEOREM: add4-plus-difference-difference
$$\begin{aligned}&(n \not< (1 + z))\\ \rightarrow\ &((1 + (1 + (1 + (1 + ((n - (1 + z)) + (n - (1 + z)))))))\\ &\quad = \ (1 + (1 + ((n - z) + (n - z)))))\end{aligned}$$

```
; Finally, we need to show that depositing one element and then the
; rest is the same as depositing them all.
```

THEOREM: deposit-lst-append
$$\begin{aligned}&\text{adpp}(\text{untag}(\textit{sp}), \textit{segment})\\ \rightarrow\ &(\text{deposit-lst}(\text{append}(a, b), \textit{sp}, \textit{segment})\\ &\quad = \ \text{deposit-lst}(b,\\ &\qquad\qquad \text{push-stk-n}(\text{length}(a), \textit{sp}),\\ &\qquad\qquad \text{deposit-lst}(a, \textit{sp}, \textit{segment})))\end{aligned}$$

THEOREM: first-n-opener-right-end
$$\begin{aligned}&(z < \text{length}(\textit{lst}))\\ \rightarrow\ &(\text{first-n}(1 + z, \textit{lst}) = \text{append}(\text{first-n}(z, \textit{lst}), \text{list}(\text{get}(z, \textit{lst}))))\end{aligned}$$

EVENT: Enable properp-rev.

THEOREM: append-right-id
properp$(a) \rightarrow (\text{append}(a, \textbf{nil}) = a)$

EVENT: Enable rev-append.

THEOREM: get-difference-sub1-length-rev

$$
\begin{array}{l}
(z < \text{length}(\textit{temp-var-dcls})) \\
\rightarrow \quad (\text{get}((\text{length}(\textit{temp-var-dcls}) - 1) - z, \text{rev}(\textit{temp-var-dcls})) \\
\qquad = \quad \text{get}(z, \textit{temp-var-dcls}))
\end{array}
$$

```
; The basic idea of the execution of i*-prelude1-generalized is that
; we start in the middle of the prelude (with n pushes left to go)
; and execute those n.  We then describe what happens to the stack.
; The lemma below is the key observation.  To understand it, it is
; really necessary to see the prelude1 theorem first.
```

THEOREM: deposit-lst-equivalence

$$
\begin{array}{l}
(\text{adpp}(\text{sub-adp}(\text{untag}(\textit{csp}), 1), \textit{sys-data-segment}) \\
\ \wedge \quad (z \in \mathbf{N}) \\
\ \wedge \quad (z < \text{length}(\textit{temp-var-dcls}))) \\
\rightarrow \quad (\text{deposit-lst}(\text{rev}(\text{strip-cadrs}(\text{first-n}(z, \textit{temp-var-dcls}))), \\
\qquad\qquad \text{tag}(\texttt{'sys-addr}, \text{sub-adp}(\text{untag}(\textit{csp}), 2)), \\
\qquad\qquad \text{deposit-adp}(\text{cadr}(\text{get}(\text{length}(\textit{temp-var-dcls}) - (1 + z), \\
\qquad\qquad\qquad\qquad \text{rev}(\textit{temp-var-dcls}))), \\
\qquad\qquad\qquad \text{sub-adp}(\text{untag}(\textit{csp}), 1), \\
\qquad\qquad\qquad \textit{sys-data-segment})) \\
\qquad = \quad \text{deposit-lst}(\text{rev}(\text{strip-cadrs}(\text{first-n}(1 + z, \textit{temp-var-dcls}))), \\
\qquad\qquad\qquad \text{tag}(\texttt{'sys-addr}, \text{sub-adp}(\text{untag}(\textit{csp}), 1)), \\
\qquad\qquad\qquad \textit{sys-data-segment}))
\end{array}
$$

```
; i*-prelude1-generalized requires a complicated induction.  The
; idea is to execute one clock tick and say what happens to
; csp (it is pushed by one) and sys-data-segment (one of the
; temp-var-dcls' initial values is deposited).
```

DEFINITION:
hint-prelude1$(n, \textit{csp}, \textit{sys-data-segment}, \textit{name}, r)$

$$
\begin{array}{l}
= \quad \textbf{if } n \simeq 0 \textbf{ then t} \\
\quad\ \ \textbf{else } \text{hint-prelude1}(n - 1, \\
\qquad\qquad \text{push-stk}(\textit{csp}), \\
\qquad\qquad \text{deposit}(\text{cadr}(\text{get}(\text{length}(\text{temp-var-dcls}(\text{assoc}(\textit{name}, \\
\qquad\qquad\qquad\qquad \text{r-prog-segment}(r))))
\end{array}
$$

$-\quad n$,
rev (temp-var-dcls (assoc (*name*,
r-prog-segment (r)))))),
push-stk (*csp*),
sys-data-segment),
name,
r) **endif**

```
; Crucial to this proof is that we elim (SUB1 N).  This introduces
; an (I* s (ADD1 Z) load-addr) where we had an (I* s N load-addr).  Then I* can be
; opened.  But the elimination on (SUB1 N) is not the first elimination
; available -- we can also elim on CDR and on the accessors to the
; R-STATEs.  Thus, I disable those elim lemmas.  I also keep first-n
; and difference closed up, manipulating them completely with the
; lemmas above.
```

THEOREM: i*-prelude1-generalized
(proper-r-statep (r, *load-addr*)
$\wedge$ all-r-objectps (strip-cadrs (temp-var-dcls (assoc (*name*,
r-prog-segment (r)))),
r)
$\wedge$ definedp (*name*, r-prog-segment (r))
$\wedge$ same-signature (r-sys-data-segment (r), *sys-data-segment*)
$\wedge$ ($n \in \mathbf{N}$)
$\wedge$ (length (temp-var-dcls (assoc (*name*, r-prog-segment (r)))) $\not< n$)
$\wedge$ cstkp (*csp*, *sys-data-segment*)
$\wedge$ (adp-offset (untag (*csp*)) $\not< n$))
$\rightarrow$ (i* (i-state (tag (’`ipc`,
cons (*name*,
$1 + (1 + (2 *$ (length (temp-var-dcls (assoc (*name*,
r-prog-segment (r))))
$-\quad n$))))),
cfp,
csp,
tsp,
x,
y,
c-flg,
v-flg,
n-flg,
z-flg,
icompile (r-prog-segment (r)),
r-usr-data-segment (r),

```
                sys-data-segment,
                r-word-size (r),
                'run),
          n,
          load-addr)
    =   i-state (tag ('ipc,
                      cons (name,
                            1 + (1 + (2 * length (temp-var-dcls (assoc (name,
                                                                         r-prog-segment (r)))))))),
                 cfp,
                 push-stk-n (n, csp),
                 tsp,
                 x,
                 y,
                 c-flg,
                 v-flg,
                 n-flg,
                 z-flg,
                 icompile (r-prog-segment (r)),
                 r-usr-data-segment (r),
                 deposit-lst (rev (strip-cadrs (first-n (n,
                                                         temp-var-dcls (assoc (name,
                                                                               r-prog-segment (r)))))),
                              push-stk (csp),
                              sys-data-segment),
                 r-word-size (r),
                 'run))
```

```
; The above theorem tells us what happens if we execute the last n
; instructions in prelude1.  Now we instantiate n with the number
; of executable instructions in prelude1 and deduce what happens when
; we execute the prelude1.

; To prove the instance we need the trivial:
```

EVENT: Disable first-n-opener-right-end.

THEOREM: first-n-length
strip-cadrs (first-n (length (*lst*), *lst*)) = strip-cadrs (*lst*)

```
; The above theorem is essentially just a first-n property and has
; almost nothing to do with strip-cadrs:  (first-n (length lst) lst)
; is lst.  That is not true if lst is not proper.  Rather than set up
```

```
; the lemmas necessary to get that the temp-var-dcls are proper, I'll
; just observe that in our application we strip-cadrs the result and
; that does the same thing whether lst is proper or not.
```

THEOREM: i*-prelude1
(proper-r-statep (*r*, *load-addr*)
 ∧ all-r-objectps (strip-cadrs (temp-var-dcls (assoc (*name*,
 r-prog-segment (*r*)))),
 r)
 ∧ definedp (*name*, r-prog-segment (*r*))
 ∧ same-signature (r-sys-data-segment (*r*), *sys-data-segment*)
 ∧ cstkp (*csp*, *sys-data-segment*)
 ∧ (adp-offset (untag (*csp*))
 $\not<$ length (temp-var-dcls (assoc (*name*, r-prog-segment (*r*))))))
→ (i* (i-state (tag ('`ipc`, cons (*name*, `2`)),
 cfp,
 csp,
 tsp,
 x,
 y,
 c-flg,
 v-flg,
 n-flg,
 z-flg,
 icompile (r-prog-segment (*r*)),
 r-usr-data-segment (*r*),
 sys-data-segment,
 r-word-size (*r*),
 '**run**),
 length (temp-var-dcls (assoc (*name*, r-prog-segment (*r*)))),
 load-addr)
 = i-state (tag ('`ipc`,
 cons (*name*,
 $1 + (1 + (2 *$ length (temp-var-dcls (assoc (*name*,
 r-prog-segment (*r*)))))))),
 cfp,
 push-stk-n (length (temp-var-dcls (assoc (*name*,
 r-prog-segment (*r*)))),
 csp),
 tsp,
 x,
 y,
 c-flg,

v-flg,
n-flg,
z-flg,
icompile (r-prog-segment (r)),
r-usr-data-segment (r),
deposit-lst (rev (strip-cadrs (temp-var-dcls (assoc (*name*,
r-prog-segment (r)))))),
push-stk (*csp*),
sys-data-segment),
r-word-size (r),
'run))

```
; That completes our work on prelude1.

; We now turn to prelude2.

; For some reason the proof of the call one-way-correspondence theorem
; raises the possibility that the sum of the lengths of the temp-var-dcls
; and the formal-vars is 0.  Rather than prevent this case split, I just
; deal with it.  The following lemma reduces that case to the case that
; there are not temp or formal variables.
```

EVENT: Enable equal-plus-0.

```
; In the case where both formals and temp-var-dcls is empty, we
; immediately get to the end of the prelude and the only remaining
; problem is dealing with the two pc expressions.  On the r level
; we have (r->i_pc (tag 'pc (cons name 0)) (r-prog-segment r))
; while at the i level we have
; (tag 'ipc (cons name length-prelude)).  We will rewrite the former
; to the latter.  We have all the necessary machinery to do this.
; We do have to appeal to the fact that the temp-var-dcls of the
; program are well-formed, e.g., r-temp-var-dclsp, if the
; program is in the program segment of a proper r state.  That was
; proved earlier, but for an instance in which name was derived from
; the current pc.  We need it for the subroutine name now.  So we
; just instantiate the necessary lemma.
```

THEOREM: r->i_pc-prelude
(proper-r-statep (r, *load-addr*) $\land$ definedp (*name*, r-prog-segment (r)))
$\rightarrow$ (r->i_pc (tag ('pc, cons (*name*, 0)), r-prog-segment (r))
= tag ('ipc,

$\text{cons}\,(\mathit{name},$
$\quad\text{length}\,(\text{generate-prelude}\,(\text{assoc}\,(\mathit{name},\ \text{r-prog-segment}\,(r)))))))$

```
; In this pathological case we also have to know:
```

THEOREM: add-adp-0-untag-r-tsp
$(\text{listp}\,(x) \land (\text{adp-offset}\,(x) \in \mathbf{N})) \rightarrow (\text{add-adp}\,(x,\ 0) = x)$

```
; We might as well get the sub-adp analogue:
```

THEOREM: sub-adp-0-untag-r-tsp
$(\text{listp}\,(x) \land (\text{adp-offset}\,(x) \in \mathbf{N})) \rightarrow (\text{sub-adp}\,(x,\ 0) = x)$

```
; Embarassing anomoly:  The LENGTH-GENERATE-PRELUDE lemma introduces a term
; of the form (TIMES x 2) which is replaced by (PLUS x x) by the
; TIMES-ADD1{gen} lemma.  But I*-PRELUDE introduces (TIMES 2 x) which is
; left alone.  I could reformulate LENGTH-GENERATE-PRELUDE.  But
; instead I will just prove (TIMES 2 X) = (PLUS X X) and keep
; TIMES out of the problem henceforth.
```

EVENT: Enable times-2.

```
; We start our study of prelude2 just like that of prelude1,
; by determining what instruction you get when you fetch from the prelude2.
```

THEOREM: fetch-adp-prelude2-lemma
$(n < \text{length}\,(\mathit{formal\text{-}vars}))$
$\rightarrow \quad (\text{get}\,(n,\ \text{generate-prelude2}\,(\mathit{formal\text{-}vars})) = {}'(\texttt{cpush\_<tsp>+}))$

THEOREM: difference-plus-k-k-length
$(((k + k) = \text{length}\,(a)) \land (n \in \mathbf{N}))$
$\rightarrow \quad (((k + k + n) - \text{length}\,(a)) = n)$

THEOREM: get-append-bridge
$(((k + k) = \text{length}\,(a)) \land (n \in \mathbf{N}))$
$\rightarrow \quad (\text{get}\,(k + (k + n),\ \text{append}\,(a,\ b)) = \text{get}\,(n,\ b))$

THEOREM: fetch-adp-prelude2
$(\text{definedp}\,(\mathit{name},\ \mathit{prog\text{-}segment})$
$\land \quad (n \in \mathbf{N})$
$\land \quad (n < \text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\mathit{name},\ \mathit{prog\text{-}segment}))))$

$\rightarrow$ (fetch-adp (cons (*name*,
1 + (1 + (length (temp-var-dcls (assoc (*name*,
prog-segment)))
+ length (temp-var-dcls (assoc (*name*,
prog-segment)))
+ *n*))),
icompile (*prog-segment*))
= `'(cpush_<tsp>+)`)

EVENT: Disable get-append-bridge.

```
; We specify the induction for prelude2 with:
```

DEFINITION:
hint-prelude2 (*n*, *csp*, *tsp*, *sys-data-segment*, *name*, *r*)
= **if** $n \simeq$ `0` **then t**
else hint-prelude2 ($n - 1$,
push-stk (*csp*),
pop-stk (*tsp*),
deposit (fetch-adp (untag (*tsp*),
sys-data-segment),
push-stk (*csp*),
sys-data-segment),
name,
r) **endif**

```
; Just as in the prelude1 case, we have disabled difference and use
; tweezers to manipulate such expressions.  This lemma shows that
; the pc in the induction conclusion is equal to that in the induction
; hypothesis.
```

THEOREM: add1-add1-plus-length-length-difference-length-add1
(length (formal-vars (assoc (*name*, r-prog-segment (*r*)))) $\not<$ (1 + *z*))
$\rightarrow$ ((1 + (1 + (1 + (length (temp-var-dcls (assoc (*name*, r-prog-segment (*r*))))
+ length (temp-var-dcls (assoc (*name*,
r-prog-segment (*r*))))
+ (length (formal-vars (assoc (*name*,
r-prog-segment (*r*))))
− (1 + *z*))))))
= (1 + (1 + (length (temp-var-dcls (assoc (*name*, r-prog-segment (*r*))))
+ length (temp-var-dcls (assoc (*name*,
r-prog-segment (*r*))))

$$+ \quad (\text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\mathit{name}, \text{r-prog-segment}\,(r))))$$
$$- \quad z)))))$$

```
; We also have to show that if you fetch n things from the tstk all
; at once that it is equivalent to fetching them interspersed with
; writes to the cstk.  In particular:
```

THEOREM: fetch-n-tstk-deposit-adp-cstk

$$
\begin{aligned}
&(\text{adpp}\,(\mathit{csp\text{-}adp},\ \mathit{sys\text{-}data\text{-}segment}) \\
&\ \wedge\ (\text{adp-name}\,(\mathit{csp\text{-}adp}) = \texttt{'cstk}) \\
&\ \wedge\ \text{tstkp}\,(\mathit{tsp},\ \mathit{sys\text{-}data\text{-}segment}) \\
&\ \wedge\ (\text{stk-length}\,(\mathit{tsp},\ \mathit{sys\text{-}data\text{-}segment}) \not< z)) \\
&\to\ (\text{fetch-n}\,(z,\ \mathit{tsp},\ \text{deposit-adp}\,(\mathit{val},\ \mathit{csp\text{-}adp},\ \mathit{sys\text{-}data\text{-}segment})) \\
&\qquad = \ \text{fetch-n}\,(z,\ \mathit{tsp},\ \mathit{sys\text{-}data\text{-}segment}))
\end{aligned}
$$

```
; So here is the generalized version of i*-prelude2.
```

THEOREM: i*-prelude2-generalized

$$
\begin{aligned}
&(\text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr}) \\
&\ \wedge\ \text{definedp}\,(\mathit{name},\ \text{r-prog-segment}\,(r)) \\
&\ \wedge\ \text{cstkp}\,(\mathit{csp},\ \mathit{sys\text{-}data\text{-}segment}) \\
&\ \wedge\ \text{tstkp}\,(\mathit{tsp},\ \mathit{sys\text{-}data\text{-}segment}) \\
&\ \wedge\ (\text{adp-offset}\,(\text{untag}\,(\mathit{csp})) \not< n) \\
&\ \wedge\ (\text{stk-length}\,(\mathit{tsp},\ \mathit{sys\text{-}data\text{-}segment}) \not< n) \\
&\ \wedge\ (n \in \mathbf{N}) \\
&\ \wedge\ (\text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\mathit{name},\ \text{r-prog-segment}\,(r)))) \not< n)) \\
&\to\ (\text{i*}\,(\text{i-state}\,(\text{tag}\,(\texttt{'ipc}, \\
&\qquad\qquad \text{cons}\,(\mathit{name}, \\
&\qquad\qquad\quad 1 + (1 + (\text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\mathit{name}, \text{r-prog-segment}\,(r)))) \\
&\qquad\qquad\qquad + \ \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\mathit{name}, \text{r-prog-segment}\,(r)))) \\
&\qquad\qquad\qquad + \ (\text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\mathit{name}, \text{r-prog-segment}\,(r)))) \\
&\qquad\qquad\qquad\quad - \ n))))), \\
&\qquad\quad \mathit{cfp}, \\
&\qquad\quad \mathit{csp}, \\
&\qquad\quad \mathit{tsp}, \\
&\qquad\quad x, \\
&\qquad\quad y, \\
&\qquad\quad \mathit{c\text{-}flg}, \\
&\qquad\quad \mathit{v\text{-}flg},
\end{aligned}
$$

$n\text{-}flg$,
$z\text{-}flg$,
icompile (r-prog-segment (r)),
r-usr-data-segment (r),
$sys\text{-}data\text{-}segment$,
r-word-size (r),
`'run`),
n,
$load\text{-}addr$)
= i-state (tag (`'ipc`,
cons ($name$,
1 + (1 + (length (temp-var-dcls (assoc ($name$,
r-prog-segment (r))))
+ length (temp-var-dcls (assoc ($name$,
r-prog-segment (r))))
+ length (formal-vars (assoc ($name$,
r-prog-segment (r)))))))),
cfp,
push-stk-n (n, csp),
pop-stk-n (n, tsp),
x,
y,
$c\text{-}flg$,
$v\text{-}flg$,
$n\text{-}flg$,
$z\text{-}flg$,
icompile (r-prog-segment (r)),
r-usr-data-segment (r),
deposit-lst (fetch-n (n, tsp, $sys\text{-}data\text{-}segment$),
push-stk (csp),
$sys\text{-}data\text{-}segment$),
r-word-size (r),
`'run`))

THEOREM: i*-prelude2
(proper-r-statep (r, $load\text{-}addr$)
$\wedge$ definedp ($name$, r-prog-segment (r))
$\wedge$ cstkp (csp, $sys\text{-}data\text{-}segment$)
$\wedge$ tstkp (tsp, $sys\text{-}data\text{-}segment$)
$\wedge$ (adp-offset (untag (csp))
$\not<$ length (formal-vars (assoc ($name$, r-prog-segment (r)))))
$\wedge$ (stk-length (tsp, $sys\text{-}data\text{-}segment$)
$\not<$ length (formal-vars (assoc ($name$, r-prog-segment (r))))))
$\rightarrow$ (i* (i-state (tag (`'ipc`,

```
        cons (name,
              1 + (1 + (length (temp-var-dcls (assoc (name,
                                                      r-prog-segment (r))))
                        +    length (temp-var-dcls (assoc (name,
                                                           r-prog-segment (r)))))))),
    cfp,
    csp,
    tsp,
    x,
    y,
    c-flg,
    v-flg,
    n-flg,
    z-flg,
    icompile (r-prog-segment (r)),
    r-usr-data-segment (r),
    sys-data-segment,
    r-word-size (r),
    'run),
length (formal-vars (assoc (name, r-prog-segment (r)))),
load-addr)
=   i-state (tag ('ipc,
        cons (name,
              1 + (1 + (length (temp-var-dcls (assoc (name,
                                                      r-prog-segment (r))))
                        +    length (temp-var-dcls (assoc (name,
                                                           r-prog-segment (r))))
                        +    length (formal-vars (assoc (name,
                                                         r-prog-segment (r)))))))),
    cfp,
    push-stk-n (length (formal-vars (assoc (name,
                                            r-prog-segment (r)))),
                csp),
    pop-stk-n (length (formal-vars (assoc (name,
                                           r-prog-segment (r)))),
               tsp),
    x,
    y,
    c-flg,
    v-flg,
    n-flg,
    z-flg,
    icompile (r-prog-segment (r)),
    r-usr-data-segment (r),
```

deposit-lst (fetch-n (length (formal-vars (assoc (*name*,
r-prog-segment (r)))),
tsp,
sys-data-segment),
push-stk (*csp*),
sys-data-segment),
r-word-size (r),
'run))

EVENT: Disable adpp-sub-adp.

THEOREM: adpp-sub-adp-generalize
adpp (x, *segment*) $\rightarrow$ adpp (sub-adp (x, n), *segment*)

THEOREM: length-strip-cadrs
length (strip-cadrs (x)) = length (x)

THEOREM: definedp-put-assoc{special}
definedp (*name1*, *segment*) $\rightarrow$ definedp (*name1*, put-assoc (*val*, *name2*, *segment*))

```
; We have some theorems like these, but they rewrite
; (adpp ... (deposit-adp ... segment)) to (adpp ... segment)
; under hyps about the legality of the deposit address.  I have
; since gotten tired of proving the legality of the addresses and
; wish to observe that even if the address is illegal the worst
; case is that the area written into becomes longer.  So the
; legal addresses after the deposit include the legal addresses
; before.
```

THEOREM: adpp-deposit-adp-generalized
adpp (*adp1*, *segment*) $\rightarrow$ adpp (*adp1*, deposit-adp (*val*, *adp2*, *segment*))

THEOREM: adpp-deposit-lst
adpp (*adp1*, *segment*) $\rightarrow$ adpp (*adp1*, deposit-lst (*lst*, *adp2*, *segment*))

```
; The following crappy lemma is used in relieving the fifth hypothesis
; of the i*-prelude2 lemma.  This undoubtedly could be generalized
; but I just got tired of finding elegant solutions.  This fact is
; immediate from linear arithmetic; unfortunately, it involves a little
; case analysis and is not caught during back chaining.
```

THEOREM: i*-prelude2-hyp-5-reliever
(((adp-offset (untag (r-csp (r))) $-$ 1) $\neq$ 0)

$$
\begin{aligned}
\land\quad & (((\text{adp-offset}\,(\text{untag}\,(\text{r-csp}\,(r))) - 1) - 1) \\
& \quad \not< \quad (\text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\text{cadr}\,(\text{unlabel}\,(\text{get}\,(\text{adp-offset}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad \text{program-body}\,(\text{assoc}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad\qquad \text{r-prog-segment}\,(r)))))), \\
& \qquad\qquad \text{r-prog-segment}\,(r)))) \\
& \qquad + \quad \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{cadr}\,(\text{unlabel}\,(\text{get}\,(\text{adp-offset}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad \text{program-body}\,(\text{assoc}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad\qquad \text{r-prog-segment}\,(r)))))), \\
& \qquad\qquad \text{r-prog-segment}\,(r))))))) \\
\to\quad & (((((\text{adp-offset}\,(\text{untag}\,(\text{r-csp}\,(r))) - 1) - 1) \\
& \quad - \quad \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{cadr}\,(\text{unlabel}\,(\text{get}\,(\text{adp-offset}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad \text{program-body}\,(\text{assoc}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad\qquad \text{r-prog-segment}\,(r)))))), \\
& \qquad\qquad \text{r-prog-segment}\,(r))))) \\
& \quad < \quad \text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\text{cadr}\,(\text{unlabel}\,(\text{get}\,(\text{adp-offset}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad \text{program-body}\,(\text{assoc}\,(\text{adp-name}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
& \qquad\qquad\qquad\qquad \text{r-prog-segment}\,(r)))))), \\
& \qquad\qquad \text{r-prog-segment}\,(r))))) \\
& \quad = \quad \mathbf{f})
\end{aligned}
$$

```
; The sixth hypothesis of i*-prelude2 also requires work.  The problem
; is that by the time we get around to asking the stk-length of tstk for
; prelude2 we have done some deposits into sys-data-segment and we must know
; that we haven't changed its length.  We have already proved that
; deposit-adp doesn't change the length, but now we must deal with
; deposit-lst.  In the case in point, we know the deposits have all
; gone into the cstk and we are interested in the length of the tstk.
; Rather than argue that no deposit-lst changes the length  of its
; area -- a theorem that requires the additional hypothesis that the
; length of the list scanned is sufficiently short -- I will prove
; this one:
```

THEOREM: length-cdr-assoc-deposit-lst

$$
\begin{aligned}
& (name \neq \text{adp-name}\,(\text{untag}\,(adp))) \\
\to\quad & (\text{length}\,(\text{cdr}\,(\text{assoc}\,(name,\ \text{deposit-lst}\,(lst,\ adp,\ segment)))) \\
& \quad = \quad \text{length}\,(\text{cdr}\,(\text{assoc}\,(name,\ segment))))
\end{aligned}
$$

```
; Now we finally turn to the arithmetic battles suggested by the fact
; that we don't know plus is commutative.  I don't want to add that
; rule; I don't want to even take the chance that it'll mess things
; up.  Instead I will prove each instance I need.
```

THEOREM: commutativity-of-plus-application

$$
\begin{array}{l}
(\text{sub-adp}\,(\text{untag}\,(\text{r-csp}\,(r)),\\
\qquad\quad 1 + (1 + (\text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\mathit{name},\ \text{r-prog-segment}\,(r))))\\
\qquad\qquad\qquad + \quad \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\mathit{name},\ \text{r-prog-segment}\,(r)))))))\\
\ = \quad \text{sub-adp}\,(\text{untag}\,(\text{r-csp}\,(r)),\\
\qquad\qquad 1 + (1 + (\text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\mathit{name},\ \text{r-prog-segment}\,(r))))\\
\qquad\qquad\qquad + \quad \text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\mathit{name},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{r-prog-segment}\,(r))))))))\\
= \quad \mathbf{t}
\end{array}
$$

```
; And again, we have to consider the fact that the r level machine
; fetches all the args from tsp and then deposits them in csp
; while the i level machine does the fetches and deposits in an
; interleaved fashion.  We exploit, once again, the fact that
; we are fetching from one area and depositing into another.
```

THEOREM: assoc-deposit-lst

$$
\begin{array}{l}
(\mathit{name} \neq \text{adp-name}\,(\text{untag}\,(\mathit{csp})))\\
\rightarrow \quad (\text{assoc}\,(\mathit{name},\ \text{deposit-lst}\,(\mathit{lst},\ \mathit{csp},\ \mathit{segment})) = \text{assoc}\,(\mathit{name},\ \mathit{segment}))
\end{array}
$$

THEOREM: fetch-n-tstk-deposit-lst-cstk

$$
\begin{array}{l}
(\text{adp-name}\,(\text{untag}\,(\mathit{tsp})) \neq \text{adp-name}\,(\text{untag}\,(\mathit{csp})))\\
\rightarrow \quad (\text{fetch-n}\,(n,\ \mathit{tsp},\ \text{deposit-lst}\,(\mathit{lst},\ \mathit{csp},\ \mathit{segment}))\\
\qquad = \quad \text{fetch-n}\,(n,\ \mathit{tsp},\ \mathit{segment}))
\end{array}
$$

THEOREM: call-one-way-correspondence-r-i

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run})\\
\ \wedge \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'call})\\
\ \wedge \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})\\
\ \wedge \quad (\mathit{load\text{-}addr} \in \mathbf{N})\\
\ \wedge \quad \text{r-call-okp}\,(\text{r-current-instruction}\,(r),\ r))\\
\rightarrow \quad (\text{r->i}\,(\text{r-call-step}\,(\text{r-current-instruction}\,(r),\ r))\\
\qquad = \quad \text{i}\,(\text{r->i}\,(r),\\
\qquad\qquad\quad \text{r-call-step-clock}\,(\text{r-current-instruction}\,(r),\ r),\\
\qquad\qquad\quad \mathit{load\text{-}addr}))
\end{array}
$$

```
; We now turn to the pop-call instruction, which has basically
; the same proof as that just done.  There are a few
; minor differences.  Pop-call starts by doing a pop and
; requires that the tstk be non-empty:
```

THEOREM: proper-r-state-implies-legal-r-tsp

$$
\begin{array}{l}
\text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})\\
\rightarrow \quad (\text{adp-offset}\,(\text{untag}\,(\text{r-tsp}\,(r)))\\
\qquad < \quad \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ \text{r-sys-data-segment}\,(r)))))
\end{array}
$$

THEOREM: assoc-tstk
proper-r-statep (r, *load-addr*)
$\rightarrow$ listp (cdr (assoc (`'tstk`, r-sys-data-segment (r))))

```
; When r-pop-call-okp checks that the call is ok, it uses
; r-call-okp on a faked up r state.  r-call-okp checks, among
; other things, that all the temp-var-dcl values are r-objects
; in that faked up state.  But the 2nd hypothesis of i*-prelude1
; requires that they all be objects in r.  The following lemma
; establishes that:
```

THEOREM: same-signature-implies-r-objectp
(r-objectp (x, $r2$)
$\wedge$ (r-prog-segment ($r1$) = r-prog-segment ($r2$))
$\wedge$ same-signature (r-usr-data-segment ($r1$), r-usr-data-segment ($r2$))
$\wedge$ same-signature (r-sys-data-segment ($r1$), r-sys-data-segment ($r2$))
$\wedge$ (r-word-size ($r1$) = r-word-size ($r2$)))
$\rightarrow$ r-objectp (x, $r1$)

THEOREM: same-signature-implies-all-r-objectps
(all-r-objectps (*lst*, $r2$)
$\wedge$ (r-prog-segment ($r1$) = r-prog-segment ($r2$))
$\wedge$ same-signature (r-usr-data-segment ($r1$), r-usr-data-segment ($r2$))
$\wedge$ same-signature (r-sys-data-segment ($r1$), r-sys-data-segment ($r2$))
$\wedge$ (r-word-size ($r1$) = r-word-size ($r2$)))
$\rightarrow$ all-r-objectps (*lst*, $r1$)

EVENT: Disable same-signature-implies-r-objectp.

```
; So now we can do pop-call.
```

THEOREM: pop-call-one-way-correspondence-r-i
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'pop-call`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ (*load-addr* $\in \mathbf{N}$)
$\wedge$ r-pop-call-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-pop-call-step (r-current-instruction (r), r))
= i (r->i (r),
r-pop-call-step-clock (r-current-instruction (r), r),
load-addr))

```
; So, now we turn to the ret instruction.  We begin with the study
; of the postlude.  First, you've got to find it.
```

THEOREM: first-n-length-properp
properp $(lst) \rightarrow$ (first-n (length (lst), lst) $=$ lst)

THEOREM: r-program-bodyp-implies-properp
r-program-bodyp $(body) \rightarrow$ properp $(body)$

THEOREM: properp-program-body
proper-r-statep (r, $load\text{-}addr$)
$\rightarrow$ properp (program-body (assoc (adp-name (untag (r-pc (r))), r-prog-segment (r))))

THEOREM: find-label-postlude
proper-r-statep (r, $load\text{-}addr$)
$\rightarrow$ (find-label (cons (adp-name (untag (r-pc (r))),
length (program-body (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))),
cdr (icompile-program (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r)))))
$=$ (length (generate-prelude (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))
$+$ length (icompile-program-body (program-body (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))),
0,
assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))))

```
; Then you've got to know what you get when you fetch from it:
```

THEOREM: fetch-adp-postlude0
proper-r-statep (r, $load\text{-}addr$)
$\rightarrow$ (fetch-adp (cons (adp-name (untag (r-pc (r))),
length (generate-prelude (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))
$+$ length (icompile-program-body (program-body (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))),
0,
assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))),
icompile (r-prog-segment (r)))
$=$ dl (cons (adp-name (untag (r-pc (r))),
length (program-body (assoc (adp-name (untag (r-pc (r))),

r-prog-segment $(r)))))$,
'(postlude),
'(move_csp_cfp)))

THEOREM: difference-add1-plus-i-j-i
$((1 + (i + j)) - i) = (1 + j)$

THEOREM: difference-add1-i-i
$((1 + i) - i) = 1$

THEOREM: fetch-adp-postlude1
proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\rightarrow$ (fetch-adp (cons (adp-name (untag (r-pc (r))),
1 + (length (generate-prelude (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r)))))
+ length (icompile-program-body (program-body (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r)))),
0,
assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r))))))),
icompile (r-prog-segment (r))))
= '(cpop_cfp))

EVENT: Disable difference-add1-plus-i-j-i.

EVENT: Disable difference-add1-i-i.

THEOREM: difference-add2-plus-i-j-i
$((1 + (1 + (i + j))) - i) = (1 + (1 + j))$

THEOREM: difference-add2-i-i
$((1 + (1 + i)) - i) = 2$

THEOREM: fetch-adp-postlude2
proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\rightarrow$ (fetch-adp (cons (adp-name (untag (r-pc (r))),
1 + (1 + (length (generate-prelude (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r)))))
+ length (icompile-program-body (program-body (assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r)))),
0,
assoc (adp-name (untag (r-pc (r))),
r-prog-segment (r)))))))),
icompile (r-prog-segment (r))))
= '(cpop_pc))

EVENT: Disable difference-add2-plus-i-j-i.

EVENT: Disable difference-add2-i-i.

EVENT: Disable length-generate-prelude.

THEOREM: length-icompile-program
length (cdr (icompile-program (*prog*)))
= (length (generate-prelude (*prog*))
+ length (icompile-program-body (program-body (*prog*), 0, *prog*))
+ 3)

```
; As usual in the cases involving jump we have to open up r->i_pc in
; the internal application (not in the fetch-execute cycle for this
; instruction).
```

THEOREM: r->i_pc-folder-ret-case
tag ('ipc,
cons (adp-name (untag (fetch-adp (add-adp (untag (r-cfp (r)), 1),
r-sys-data-segment (r)))),
find-label (untag (fetch-adp (add-adp (untag (r-cfp (r)), 1),
r-sys-data-segment (r))),
cdr (icompile-program (assoc (adp-name (untag (fetch-adp (add-adp (untag (r-cfp (r)),
1),
r-sys-data-segment (r)))),
r-prog-segment (r)))))))
= r->i_pc (fetch-adp (add-adp (untag (r-cfp (r)), 1), r-sys-data-segment (r)),
r-prog-segment (r))

THEOREM: ret-one-way-correspondence-r-i
((r-psw (r) = 'run)
∧ (car (r-current-instruction (r)) = 'ret)
∧ proper-r-statep (r, *load-addr*)
∧ (*load-addr* $\in \mathbf{N}$)
∧ r-ret-okp (r-current-instruction (r), r))
→ (r->i (r-ret-step (r-current-instruction (r), r))
= i (r->i (r),
r-ret-step-clock (r-current-instruction (r), r),
load-addr))

```
; We now begin the proof of the invariance of proper-r-statep
; under the r-step function.  The theorem is needed to relieve the
```

```
; proper-r-statep hypothesis in our induction hypothesis for the
; owc theorem.

; The proof proceeds exactly as you would expect, one lemma for
; each r-level instruction.

; First, however, we prove a key fact about proper-r-statep.
; The idea is that if r is proper and r' is obtained from r by
; modifying certain of the fields, e.g., pushing on r-tsp,
; what must we establish to show that r' is proper?  We
; lay all this out in the proper-r-statep-r-state lemma below.
```

THEOREM: equal-lessp-rewriter

$((x < y) = z)$
$= \quad (((x < y) \rightarrow (\mathbf{t} = z)) \land ((x \not< y) \rightarrow (\mathbf{f} = z)))$

THEOREM: same-signature-segment-length

same-signature (*segment1*, *segment2*)
$\rightarrow$ (segment-length (*segment2*) = segment-length (*segment1*))

THEOREM: same-signature-implies-r-objectp-rewriter

(proper-r-statep (r, *load-addr*)
$\land$ same-signature (r-sys-data-segment (r), *r-sys-data-segment*)
$\land$ same-signature (r-usr-data-segment (r), *r-usr-data-segment*))
$\rightarrow$ (r-objectp (x,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,
r-y,
r-c-flg,
r-v-flg,
r-n-flg,
r-z-flg,
r-prog-segment (r),
r-usr-data-segment,
r-sys-data-segment,
r-word-size (r),
r-psw))
= r-objectp (x, r))

THEOREM: same-signature-implies-all-r-usr-data-objectps

(proper-r-statep (r, *load-addr*)

$\wedge$ same-signature (r-sys-data-segment (r), *r-sys-data-segment*)
$\wedge$ same-signature (r-usr-data-segment (r), *r-usr-data-segment*))
$\rightarrow$ (all-r-usr-data-objectps (*lst*,
    r-state (*r-pc*,
        r-cfp,
        r-csp,
        r-tsp,
        r-x,
        r-y,
        r-c-flg,
        r-v-flg,
        r-n-flg,
        r-z-flg,
        r-prog-segment (r),
        r-usr-data-segment,
        r-sys-data-segment,
        r-word-size (r),
        r-psw))
  = all-r-usr-data-objectps (*lst*, r))

THEOREM: same-signature-implies-proper-r-usr-data-segmentp
(proper-r-statep (r, *load-addr*)
$\wedge$ same-signature (r-sys-data-segment (r), *r-sys-data-segment*)
$\wedge$ same-signature (r-usr-data-segment (r), *r-usr-data-segment*))
$\rightarrow$ (proper-r-usr-data-segmentp (*lst*,
    r-state (*r-pc*,
        r-cfp,
        r-csp,
        r-tsp,
        r-x,
        r-y,
        r-c-flg,
        r-v-flg,
        r-n-flg,
        r-z-flg,
        r-prog-segment (r),
        r-usr-data-segment,
        r-sys-data-segment,
        r-word-size (r),
        r-psw))
  = proper-r-usr-data-segmentp (*lst*, r))

```
; Here is the main theorem of this section.  This is really a
; beautiful result because it says that proper-r-statep can
```

```
; be opened up on an explicit r-state, r', that is weakly related
; to another r-state, r,  (as delivered by the
; step functions) provided  r was proper and
; has the same signatures for the sys and usr data segments.
; These hypotheses should be easy to get, given our existing
; work.

; But when we "open" proper-r-statep on r' we don't actually use
; r' but r in the various checks.  For example, we require that
; the new r-tsp be a tstkp in r, not in r' as the defn would
; require.  They are equivalent, of course, but by reducing
; the conditions to facts about r we render most of them obvious.
; The reason for this is that a typical step will change a few of
; the fields and leave the rest untouched.  Those that are untouched
; will be proper by the properness of r.

; The hint below, to expand (proper-r-statep r' load-addr), is ugly.  At first
; I just enabled proper-r-statep.  But that is suicide.  The reason is
; that we have an enormous number of rules that back chain to proper-r-statep
; and if it is enabled, those rules kill us.
```

THEOREM: proper-r-statep-r-state

$$
\begin{array}{l}
(\text{proper-r-statep}\,(r,\ \textit{load-addr}) \\
\ \ \land\ \ \text{same-signature}\,(\text{r-sys-data-segment}\,(r),\ \textit{r-sys-data-segment}) \\
\ \ \land\ \ \text{same-signature}\,(\text{r-usr-data-segment}\,(r),\ \textit{r-usr-data-segment})) \\
\to\ \ (\text{proper-r-statep}\,(\text{r-state}\,(\textit{r-pc}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-cfp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-csp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-tsp}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-x}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-y}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-c-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-v-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-n-flg}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-z-flg}, \\
\qquad\qquad\qquad\qquad\qquad \text{r-prog-segment}\,(r), \\
\qquad\qquad\qquad\qquad\qquad \textit{r-usr-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \textit{r-sys-data-segment}, \\
\qquad\qquad\qquad\qquad\qquad \text{r-word-size}\,(r), \\
\qquad\qquad\qquad\qquad\qquad \textit{r-psw}), \\
\qquad\qquad\qquad\qquad \textit{load-addr}) \\
\qquad =\ \ (\text{r-objectp-type}\,(\texttt{'pc},\ \textit{r-pc},\ r) \\
\qquad\qquad \land\ \ \text{cstkp}\,(\textit{r-cfp},\ \text{r-sys-data-segment}\,(r))
\end{array}
$$

$\land$ cstkp (*r-csp*, r-sys-data-segment (*r*))
$\land$ tstkp (*r-tsp*, r-sys-data-segment (*r*))
$\land$ (definition (`'full-ctrl-stk-addr`,
r-sys-data-segment)
= list (`'full-ctrl-stk-addr`,
tag (`'sys-addr`, `'(cstk . 0)`)))
$\land$ (definition (`'full-temp-stk-addr`,
r-sys-data-segment)
= list (`'full-temp-stk-addr`,
tag (`'sys-addr`, `'(tstk . 0)`)))
$\land$ (definition (`'empty-temp-stk-addr`,
r-sys-data-segment)
= list (`'empty-temp-stk-addr`,
tag (`'sys-addr`,
cons (`'tstk`,
length (definiens (`'tstk`,
r-sys-data-segment)) $-$ 1))))
$\land$ r-objectp-type (`'bool`, *r-c-flg*, *r*)
$\land$ r-objectp-type (`'bool`, *r-v-flg*, *r*)
$\land$ r-objectp-type (`'bool`, *r-n-flg*, *r*)
$\land$ r-objectp-type (`'bool`, *r-z-flg*, *r*)
$\land$ proper-r-usr-data-segmentp (*r-usr-data-segment*, *r*)))

EVENT: Disable same-signature-segment-length.

EVENT: Disable equal-lessp-rewriter.

THEOREM: no-op-preserves-proper-r-statep
((r-psw (*r*) = `'run`)
$\land$ proper-r-statep (*r*, *load-addr*)
$\land$ r-no-op-okp (r-current-instruction (*r*), *r*))
$\rightarrow$ proper-r-statep (r-no-op-step (r-current-instruction (*r*), *r*), *load-addr*)

```
; This lemma is similar to fetch-adp-deposit-adp, but we are
; interested in the entire area assoc finds, not just the
; contents of a given location.
```

THEOREM: assoc-deposit-adp
(*name* $\neq$ adp-name (*adp*))
$\rightarrow$ (assoc (*name*, deposit-adp (*val*, *adp*, *segment*)) = assoc (*name*, *segment*))

THEOREM: push-global-proper-r-statep
((r-psw (*r*) = `'run`)

$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-global-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-push-global-step (r-current-instruction (r), r), *load-addr*)

THEOREM: push-local-proper-r-statep
((r-psw (r) = `'run`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-local-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-push-local-step (r-current-instruction (r), r), *load-addr*)

THEOREM: push-constant-proper-r-statep
((r-psw (r) = `'run`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-constant-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-push-constant-step (r-current-instruction (r), r), *load-addr*)

THEOREM: push-ctrl-stk-free-size-proper-r-statep
((r-psw (r) = `'run`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-ctrl-stk-free-size-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-push-ctrl-stk-free-size-step (r-current-instruction (r), r), *load-addr*)

THEOREM: push-temp-stk-free-size-proper-r-statep
((r-psw (r) = `'run`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-temp-stk-free-size-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-push-temp-stk-free-size-step (r-current-instruction (r), r), *load-addr*)

THEOREM: push-temp-stk-index-proper-r-statep
((r-psw (r) = `'run`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-temp-stk-index-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-push-temp-stk-index-step (r-current-instruction (r), r), *load-addr*)

THEOREM: pop-proper-r-statep
((r-psw (r) = `'run`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-pop-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-pop-step (r-current-instruction (r), r), *load-addr*)

THEOREM: pop*-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ r-pop*-okp (r-current-instruction (r), r))
 → proper-r-statep (r-pop*-step (r-current-instruction (r), r), *load-addr*)

THEOREM: popn-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ r-popn-okp (r-current-instruction (r), r))
 → proper-r-statep (r-popn-step (r-current-instruction (r), r), *load-addr*)

THEOREM: pop-local-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ r-pop-local-okp (r-current-instruction (r), r))
 → proper-r-statep (r-pop-local-step (r-current-instruction (r), r),
 load-addr)

```
; This lemma is just like litatom-adp-name above except
; is about (cadr (unlabel x)) instead of (adp-name adp).
; As noted earlier, it actually has nothing to do with
; cadr or adp-name; it is an instance of a more general
; (but unusable) rule about proper-alists and litatom.
```

THEOREM: litatom-cadr-unlabel
(definedp (cadr (unlabel (x)), *segment*) ∧ proper-alistp (*segment*))
 → litatom (cadr (unlabel (x)))

```
; Here is another fundamental fact:
```

THEOREM: all-r-usr-data-objectps-put
((n < length (*lst*))
 ∧ r-usr-data-objectp (x, r)
 ∧ all-r-usr-data-objectps (*lst*, r))
 → all-r-usr-data-objectps (put (x, n, *lst*), r)

THEOREM: not-definedp-put-assoc
(¬ definedp (*name1*, *alist*))
 → (¬ definedp (*name1*, put-assoc (*val*, *name2*, *alist*)))

THEOREM: proper-r-usr-data-segmentp-deposit-adp
(adpp (*adp*, *segment*)
 ∧ proper-r-usr-data-segmentp (*segment*, r)
 ∧ r-usr-data-objectp (*val*, r))
 → proper-r-usr-data-segmentp (deposit-adp (*val*, *adp*, *segment*), r)

THEOREM: pop-global-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-pop-global-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-pop-global-step}(\text{r-current-instruction}(r),\ r),$
$\qquad\qquad \mathit{load\text{-}addr})$

THEOREM: set-local-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-set-local-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-set-local-step}(\text{r-current-instruction}(r),\ r),$
$\qquad\qquad \mathit{load\text{-}addr})$

THEOREM: set-global-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-set-global-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-set-global-step}(\text{r-current-instruction}(r),\ r),$
$\qquad\qquad \mathit{load\text{-}addr})$

THEOREM: jump-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-jump-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-jump-step}(\text{r-current-instruction}(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: pushj-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-pushj-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-pushj-step}(\text{r-current-instruction}(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: popj-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-popj-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-popj-step}(\text{r-current-instruction}(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: add-addr-proper-r-statep
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ \ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \ \text{r-add-addr-okp}(\text{r-current-instruction}(r),\ r))$
$\rightarrow\ \ \text{proper-r-statep}(\text{r-add-addr-step}(\text{r-current-instruction}(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: sub-addr-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep (r, *load-addr*)
 $\land$ r-sub-addr-okp (r-current-instruction (r), r))
 $\rightarrow$ proper-r-statep (r-sub-addr-step (r-current-instruction (r), r), *load-addr*)

THEOREM: eq-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep (r, *load-addr*)
 $\land$ r-eq-okp (r-current-instruction (r), r))
 $\rightarrow$ proper-r-statep (r-eq-step (r-current-instruction (r), r), *load-addr*)

THEOREM: lt-addr-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep (r, *load-addr*)
 $\land$ r-lt-addr-okp (r-current-instruction (r), r))
 $\rightarrow$ proper-r-statep (r-lt-addr-step (r-current-instruction (r), r), *load-addr*)

THEOREM: fetch-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep (r, *load-addr*)
 $\land$ r-fetch-okp (r-current-instruction (r), r))
 $\rightarrow$ proper-r-statep (r-fetch-step (r-current-instruction (r), r), *load-addr*)

THEOREM: litatom-adp-name-instance
(adpp (untag (fetch-adp (untag (r-tsp (r)), r-sys-data-segment (r))),
 r-usr-data-segment (r))
 $\land$ proper-alistp (r-usr-data-segment (r)))
 $\rightarrow$ litatom (adp-name (untag (fetch-adp (untag (r-tsp (r)),
 r-sys-data-segment (r)))))

THEOREM: not-zerop-sub1-difference
$(((x - y) - 1) = \mathtt{0}) = (y \not< (x - 1))$

THEOREM: not-empty-stkp-implies-adpp-add-adp
(adpp (*adp*, *segment*)
 $\land$ ($\neg$ empty-stkp (tag (**'sys-addr**, add-adp (*adp*, n)), *segment*))
 $\land$ ($n \in \mathbf{N}$))
 $\rightarrow$ adpp (add-adp (*adp*, $1 + n$), *segment*)

THEOREM: deposit-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep (r, *load-addr*)
 $\land$ r-deposit-okp (r-current-instruction (r), r))
 $\rightarrow$ proper-r-statep (r-deposit-step (r-current-instruction (r), r), *load-addr*)

THEOREM: fetch-temp-stk-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-fetch-temp-stk-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-fetch-temp-stk-step (r-current-instruction $(r),\ r)$,
load-addr)

THEOREM: deposit-temp-stk-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-deposit-temp-stk-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-deposit-temp-stk-step (r-current-instruction $(r),\ r)$,
load-addr)

THEOREM: add-int-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-add-int-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-add-int-step (r-current-instruction $(r),\ r)$, *load-addr*)

THEOREM: add-int-with-carry-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-add-int-with-carry-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-add-int-with-carry-step (r-current-instruction $(r),\ r)$,
load-addr)

THEOREM: add1-int-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-add1-int-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-add1-int-step (r-current-instruction $(r),\ r)$, *load-addr*)

THEOREM: sub-int-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-sub-int-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-sub-int-step (r-current-instruction $(r),\ r)$, *load-addr*)

THEOREM: sub-int-with-carry-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-sub-int-with-carry-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ proper-r-statep (r-sub-int-with-carry-step (r-current-instruction $(r),\ r)$,
load-addr)

THEOREM: sub1-int-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-sub1-int-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-sub1-int-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: neg-int-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-neg-int-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-neg-int-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: lt-int-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-lt-int-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-lt-int-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: int-to-nat-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-int-to-nat-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-int-to-nat-step (r-current-instruction $(r),\ r),$
 $\mathit{load\text{-}addr})$

THEOREM: add-nat-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-add-nat-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-add-nat-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: add-nat-with-carry-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-add-nat-with-carry-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-add-nat-with-carry-step (r-current-instruction $(r),\ r),$
 $\mathit{load\text{-}addr})$

THEOREM: add1-nat-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-add1-nat-okp (r-current-instruction $(r),\ r)$)
 $\rightarrow$ proper-r-statep (r-add1-nat-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: sub-nat-proper-r-statep

((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-sub-nat-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-sub-nat-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: sub-nat-with-carry-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-sub-nat-with-carry-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-sub-nat-with-carry-step (r-current-instruction $(r),\ r),$
 $\mathit{load\text{-}addr})$

THEOREM: sub1-nat-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-sub1-nat-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-sub1-nat-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: lt-nat-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-lt-nat-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-lt-nat-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: mult2-nat-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-mult2-nat-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-mult2-nat-step (r-current-instruction $(r),\ r),$
 $\mathit{load\text{-}addr})$

THEOREM: mult2-nat-with-carry-out-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-mult2-nat-with-carry-out-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-mult2-nat-with-carry-out-step (r-current-instruction $(r),$
 $r),$
 $\mathit{load\text{-}addr})$

THEOREM: div2-nat-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-div2-nat-okp (r-current-instruction $(r),\ r)$)
 → proper-r-statep (r-div2-nat-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr})$

THEOREM: or-bitv-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-or-bitv-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-or-bitv-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: and-bitv-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-and-bitv-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-and-bitv-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: not-bitv-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-not-bitv-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-not-bitv-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: xor-bitv-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-xor-bitv-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-xor-bitv-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: rsh-bitv-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-rsh-bitv-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-rsh-bitv-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: lsh-bitv-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-lsh-bitv-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-lsh-bitv-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: or-bool-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-or-bool-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-or-bool-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: and-bool-proper-r-statep
((r-psw (r) = **'run**)
 ∧ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 ∧ r-and-bool-okp (r-current-instruction $(r),\ r$))
→ proper-r-statep (r-and-bool-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: not-bool-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-not-bool-okp (r-current-instruction $(r),\ r$))
 $\rightarrow$ proper-r-statep (r-not-bool-step (r-current-instruction $(r),\ r),\ \mathit{load\text{-}addr}$)

THEOREM: jump-if-temp-stk-full-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-jump-if-temp-stk-full-okp (r-current-instruction $(r),\ r$))
 $\rightarrow$ proper-r-statep (r-jump-if-temp-stk-full-step (r-current-instruction (r),
 r),
 $\mathit{load\text{-}addr}$)

THEOREM: jump-if-temp-stk-empty-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-jump-if-temp-stk-empty-okp (r-current-instruction $(r),\ r$))
 $\rightarrow$ proper-r-statep (r-jump-if-temp-stk-empty-step (r-current-instruction (r),
 r),
 $\mathit{load\text{-}addr}$)

THEOREM: test-nat-and-jump-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-test-nat-and-jump-okp (r-current-instruction $(r),\ r$))
 $\rightarrow$ proper-r-statep (r-test-nat-and-jump-step (r-current-instruction $(r),\ r$),
 $\mathit{load\text{-}addr}$)

THEOREM: test-int-and-jump-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-test-int-and-jump-okp (r-current-instruction $(r),\ r$))
 $\rightarrow$ proper-r-statep (r-test-int-and-jump-step (r-current-instruction $(r),\ r$),
 $\mathit{load\text{-}addr}$)

THEOREM: test-bool-and-jump-proper-r-statep
((r-psw (r) = **'run**)
 $\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
 $\land$ r-test-bool-and-jump-okp (r-current-instruction $(r),\ r$))
 $\rightarrow$ proper-r-statep (r-test-bool-and-jump-step (r-current-instruction $(r),\ r$),
 $\mathit{load\text{-}addr}$)

THEOREM: test-bitv-and-jump-proper-r-statep
((r-psw (r) = **'run**)

$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-test-bitv-and-jump-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-test-bitv-and-jump-step (r-current-instruction (r), r), *load-addr*)

THEOREM: pop-locn-proper-r-statep
((r-psw (r) = **'run**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-pop-locn-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-pop-locn-step (r-current-instruction (r), r), *load-addr*)

THEOREM: locn-proper-r-statep
((r-psw (r) = **'run**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-locn-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-locn-step (r-current-instruction (r), r), *load-addr*)

THEOREM: jump-case-proper-r-statep
((r-psw (r) = **'run**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-jump-case-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-jump-case-step (r-current-instruction (r), r), *load-addr*)

THEOREM: same-signature-deposit-lst
(adpp (untag (*sp*), *segment2*)
$\wedge$ litatom (adp-name (untag (*sp*)))
$\wedge$ ((1 + adp-offset (untag (*sp*))) $\not<$ length (*lst*)))
$\rightarrow$ (same-signature (*segment1*, deposit-lst (*lst*, *sp*, *segment2*))
= same-signature (*segment1*, *segment2*))

THEOREM: length-fetch-n
length (fetch-n (n, *sp*, *segment*)) = fix (n)

```
; In the call case, the same-signature-deposit-lst lemma is used twice
; in establishing the sys-data-segment same-signature hyp of the
; proper-r-statep-r-state theorem.  The first time is to strip off
; the deposit-lst across the formals and the second is to strip off
; the deposit-lst across the temp-var-dcls.  Both times we have to
; prove the third hyp of same-signature-deposit-lst lemma, the one
; that says there is enough room on the stack to push everything in
; the list.  The problem, as always, is that DIFFERENCE is messing
; things up.  We need a lemma for each occasion.
```

THEOREM: same-signature-deposit-lst-hyp-3-reliever-1
$(c \not< (y + x))$
$\rightarrow$ $((\textbf{if } c = 0 \textbf{ then } 0$
$\textbf{else } (c - 1) - x \textbf{ endif}$
$< (y - 1))$
$= \textbf{f})$

THEOREM: same-signature-deposit-lst-hyp-3-reliever-2
$(c \not< (y + x))$
$\rightarrow$ $((\textbf{if } c = 0 \textbf{ then } 0$
$\textbf{else } c - 1 \textbf{ endif}$
$< (x - 1))$
$= \textbf{f})$

```
; Historical Note:  Until I got to the call case, I was unaware that
; it has been assumed that program bodies are always listps.  Otherwise,
; setting the pc to 0 on call would be illegal.  I had to add the listp
; requirement to r-programp to get call to go through.
```

THEOREM: listp-program-body-generalized
(r-prog-segmentp (*segment*) $\land$ definedp (*name*, *segment*))
$\rightarrow$ listp (program-body (assoc (*name*, *segment*)))

THEOREM: listp-program-body
(proper-r-statep (r, *load-addr*) $\land$ definedp (*name*, r-prog-segment (r)))
$\rightarrow$ listp (program-body (assoc (*name*, r-prog-segment (r))))

THEOREM: call-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-call-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-call-step (r-current-instruction (r), r), *load-addr*)

THEOREM: pop-call-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-pop-call-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-pop-call-step (r-current-instruction (r), r), *load-addr*)

THEOREM: ret-proper-r-statep
((r-psw (r) = **'run**)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-ret-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-ret-step (r-current-instruction (r), r), *load-addr*)

```
; I will now disable all the individual okp, step, and clock functions.
```

EVENT: Disable r-call-okp.

EVENT: Disable r-call-step.

EVENT: Disable r-call-step-clock.

EVENT: Disable r-ret-okp.

EVENT: Disable r-ret-step.

EVENT: Disable r-ret-step-clock.

EVENT: Disable r-locn-okp.

EVENT: Disable r-locn-step.

EVENT: Disable r-locn-step-clock.

EVENT: Disable r-push-constant-okp.

EVENT: Disable r-push-constant-step.

EVENT: Disable r-push-constant-step-clock.

EVENT: Disable r-push-local-okp.

EVENT: Disable r-push-local-step.

EVENT: Disable r-push-local-step-clock.

EVENT: Disable r-push-global-okp.

EVENT: Disable r-push-global-step.

EVENT: Disable r-push-global-step-clock.

EVENT: Disable r-push-ctrl-stk-free-size-okp.

EVENT: Disable r-push-ctrl-stk-free-size-step.

EVENT: Disable r-push-ctrl-stk-free-size-step-clock.

EVENT: Disable r-push-temp-stk-free-size-okp.

EVENT: Disable r-push-temp-stk-free-size-step.

EVENT: Disable r-push-temp-stk-free-size-step-clock.

EVENT: Disable r-push-temp-stk-index-okp.

EVENT: Disable r-push-temp-stk-index-step.

EVENT: Disable r-push-temp-stk-index-step-clock.

EVENT: Disable r-jump-if-temp-stk-full-okp.

EVENT: Disable r-jump-if-temp-stk-full-step.

EVENT: Disable r-jump-if-temp-stk-full-step-clock.

EVENT: Disable r-jump-if-temp-stk-empty-okp.

EVENT: Disable r-jump-if-temp-stk-empty-step.

EVENT: Disable r-jump-if-temp-stk-empty-step-clock.

EVENT: Disable r-pop-okp.

EVENT: Disable r-pop-step.

EVENT: Disable r-pop-step-clock.

EVENT: Disable r-pop*-okp.

EVENT: Disable r-pop*-step.

EVENT: Disable r-pop*-step-clock.

EVENT: Disable r-popn-okp.

EVENT: Disable r-popn-step.

EVENT: Disable r-popn-step-clock.

EVENT: Disable r-pop-local-okp.

EVENT: Disable r-pop-local-step.

EVENT: Disable r-pop-local-step-clock.

EVENT: Disable r-pop-global-okp.

EVENT: Disable r-pop-global-step.

EVENT: Disable r-pop-global-step-clock.

EVENT: Disable r-pop-locn-okp.

EVENT: Disable r-pop-locn-step.

EVENT: Disable r-pop-locn-step-clock.

EVENT: Disable r-pop-call-okp.

EVENT: Disable r-pop-call-step.

EVENT: Disable r-pop-call-step-clock.

EVENT: Disable r-fetch-temp-stk-okp.

EVENT: Disable r-fetch-temp-stk-step.

EVENT: Disable r-fetch-temp-stk-step-clock.

EVENT: Disable r-deposit-temp-stk-okp.

EVENT: Disable r-deposit-temp-stk-step.

EVENT: Disable r-deposit-temp-stk-step-clock.

EVENT: Disable r-jump-okp.

EVENT: Disable r-jump-step.

EVENT: Disable r-jump-step-clock.

EVENT: Disable r-jump-case-okp.

EVENT: Disable r-jump-case-step.

EVENT: Disable r-jump-case-step-clock.

EVENT: Disable r-pushj-okp.

EVENT: Disable r-pushj-step.

EVENT: Disable r-pushj-step-clock.

EVENT: Disable r-popj-okp.

EVENT: Disable r-popj-step.

EVENT: Disable r-popj-step-clock.

EVENT: Disable r-set-local-okp.

EVENT: Disable r-set-local-step.

EVENT: Disable r-set-local-step-clock.

EVENT: Disable r-set-global-okp.

EVENT: Disable r-set-global-step.

EVENT: Disable r-set-global-step-clock.

EVENT: Disable r-test-nat-and-jump-okp.

EVENT: Disable r-test-nat-and-jump-step.

EVENT: Disable r-test-nat-and-jump-step-clock.

EVENT: Disable r-test-int-and-jump-okp.

EVENT: Disable r-test-int-and-jump-step.

EVENT: Disable r-test-int-and-jump-step-clock.

EVENT: Disable r-test-bool-and-jump-okp.

EVENT: Disable r-test-bool-and-jump-step.

EVENT: Disable r-test-bool-and-jump-step-clock.

EVENT: Disable r-test-bitv-and-jump-okp.

EVENT: Disable r-test-bitv-and-jump-step.

EVENT: Disable r-test-bitv-and-jump-step-clock.

EVENT: Disable r-no-op-okp.

EVENT: Disable r-no-op-step.

EVENT: Disable r-no-op-step-clock.

EVENT: Disable r-add-addr-okp.

EVENT: Disable r-add-addr-step.

EVENT: Disable r-add-addr-step-clock.

EVENT: Disable r-sub-addr-okp.

EVENT: Disable r-sub-addr-step.

EVENT: Disable r-sub-addr-step-clock.

EVENT: Disable r-eq-okp.

EVENT: Disable r-eq-step.

EVENT: Disable r-eq-step-clock.

EVENT: Disable r-lt-addr-okp.

EVENT: Disable r-lt-addr-step.

EVENT: Disable r-lt-addr-step-clock.

EVENT: Disable r-fetch-okp.

EVENT: Disable r-fetch-step.

EVENT: Disable r-fetch-step-clock.

EVENT: Disable r-deposit-okp.

EVENT: Disable r-deposit-step.

EVENT: Disable r-deposit-step-clock.

EVENT: Disable r-add-int-okp.

EVENT: Disable r-add-int-step.

EVENT: Disable r-add-int-step-clock.

EVENT: Disable r-add-int-with-carry-okp.

EVENT: Disable r-add-int-with-carry-step.

EVENT: Disable r-add-int-with-carry-step-clock.

EVENT: Disable r-add1-int-okp.

EVENT: Disable r-add1-int-step.

EVENT: Disable r-add1-int-step-clock.

EVENT: Disable r-sub-int-okp.

EVENT: Disable r-sub-int-step.

EVENT: Disable r-sub-int-step-clock.

EVENT: Disable r-sub-int-with-carry-okp.

EVENT: Disable r-sub-int-with-carry-step.

Event: Disable r-sub-int-with-carry-step-clock.

Event: Disable r-sub1-int-okp.

Event: Disable r-sub1-int-step.

Event: Disable r-sub1-int-step-clock.

Event: Disable r-neg-int-okp.

Event: Disable r-neg-int-step.

Event: Disable r-neg-int-step-clock.

Event: Disable r-lt-int-okp.

Event: Disable r-lt-int-step.

Event: Disable r-lt-int-step-clock.

Event: Disable r-int-to-nat-okp.

Event: Disable r-int-to-nat-step.

Event: Disable r-int-to-nat-step-clock.

Event: Disable r-add-nat-okp.

Event: Disable r-add-nat-step.

Event: Disable r-add-nat-step-clock.

Event: Disable r-add-nat-with-carry-okp.

Event: Disable r-add-nat-with-carry-step.

EVENT: Disable r-add-nat-with-carry-step-clock.

EVENT: Disable r-add1-nat-okp.

EVENT: Disable r-add1-nat-step.

EVENT: Disable r-add1-nat-step-clock.

EVENT: Disable r-sub-nat-okp.

EVENT: Disable r-sub-nat-step.

EVENT: Disable r-sub-nat-step-clock.

EVENT: Disable r-sub-nat-with-carry-okp.

EVENT: Disable r-sub-nat-with-carry-step.

EVENT: Disable r-sub-nat-with-carry-step-clock.

EVENT: Disable r-sub1-nat-okp.

EVENT: Disable r-sub1-nat-step.

EVENT: Disable r-sub1-nat-step-clock.

EVENT: Disable r-lt-nat-okp.

EVENT: Disable r-lt-nat-step.

EVENT: Disable r-lt-nat-step-clock.

EVENT: Disable r-mult2-nat-okp.

EVENT: Disable r-mult2-nat-step.

Event: Disable r-mult2-nat-step-clock.

Event: Disable r-mult2-nat-with-carry-out-okp.

Event: Disable r-mult2-nat-with-carry-out-step.

Event: Disable r-mult2-nat-with-carry-out-step-clock.

Event: Disable r-div2-nat-okp.

Event: Disable r-div2-nat-step.

Event: Disable r-div2-nat-step-clock.

Event: Disable r-or-bitv-okp.

Event: Disable r-or-bitv-step.

Event: Disable r-or-bitv-step-clock.

Event: Disable r-and-bitv-okp.

Event: Disable r-and-bitv-step.

Event: Disable r-and-bitv-step-clock.

Event: Disable r-not-bitv-okp.

Event: Disable r-not-bitv-step.

Event: Disable r-not-bitv-step-clock.

Event: Disable r-xor-bitv-okp.

Event: Disable r-xor-bitv-step.

EVENT: Disable r-xor-bitv-step-clock.

EVENT: Disable r-rsh-bitv-okp.

EVENT: Disable r-rsh-bitv-step.

EVENT: Disable r-rsh-bitv-step-clock.

EVENT: Disable r-lsh-bitv-okp.

EVENT: Disable r-lsh-bitv-step.

EVENT: Disable r-lsh-bitv-step-clock.

EVENT: Disable r-or-bool-okp.

EVENT: Disable r-or-bool-step.

EVENT: Disable r-or-bool-step-clock.

EVENT: Disable r-and-bool-okp.

EVENT: Disable r-and-bool-step.

EVENT: Disable r-and-bool-step-clock.

EVENT: Disable r-not-bool-okp.

EVENT: Disable r-not-bool-step.

EVENT: Disable r-not-bool-step-clock.

```
; I now proceed to package all this up into two lemmas
; about r-step.  The first is that it preserves proper-r-statep.
; The second is the one-way-correspondence theorem for it.
```

THEOREM: proper-r-statep-r-halt
proper-r-statep (r, *load-addr*) $\rightarrow$ proper-r-statep (r-halt (r, *psw*), *load-addr*)

```
; I am trying to control the case explosion produced by r-step.
; It has two sources, r-ins-okp and r-ins-step.  I have arbitrarily
; decided to disable r-ins-okp and let r-ins-step drive the case
; expansion.  To get away with disabling r-ins-okp I need to be
; able to establish all of the individual instruction okps from
; (r-ins-okp ins r).  Then, when the previously proved lemmas
; for particular instructions need a particular okp, they will
; find it here:
```

THEOREM: r-ins-okp-backchainer
r-ins-okp (*ins*, r)
$\rightarrow$ (((car (*ins*) = `'call`) $\rightarrow$ r-call-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'ret`) $\rightarrow$ r-ret-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'locn`) $\rightarrow$ r-locn-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'push-constant`)
$\rightarrow$ r-push-constant-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'push-local`) $\rightarrow$ r-push-local-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'push-global`) $\rightarrow$ r-push-global-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'push-ctrl-stk-free-size`)
$\rightarrow$ r-push-ctrl-stk-free-size-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'push-temp-stk-free-size`)
$\rightarrow$ r-push-temp-stk-free-size-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'push-temp-stk-index`)
$\rightarrow$ r-push-temp-stk-index-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'jump-if-temp-stk-full`)
$\rightarrow$ r-jump-if-temp-stk-full-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'jump-if-temp-stk-empty`)
$\rightarrow$ r-jump-if-temp-stk-empty-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'pop`) $\rightarrow$ r-pop-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'pop*`) $\rightarrow$ r-pop*-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'popn`) $\rightarrow$ r-popn-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'pop-local`) $\rightarrow$ r-pop-local-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'pop-global`) $\rightarrow$ r-pop-global-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'pop-locn`) $\rightarrow$ r-pop-locn-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'pop-call`) $\rightarrow$ r-pop-call-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'fetch-temp-stk`)
$\rightarrow$ r-fetch-temp-stk-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'deposit-temp-stk`)
$\rightarrow$ r-deposit-temp-stk-okp (*ins*, r))
$\wedge$ ((car (*ins*) = `'jump`) $\rightarrow$ r-jump-okp (*ins*, r))

$\wedge$ ((car (ins) = '`jump-case`) $\rightarrow$ r-jump-case-okp (ins, r))
$\wedge$ ((car (ins) = '`pushj`) $\rightarrow$ r-pushj-okp (ins, r))
$\wedge$ ((car (ins) = '`popj`) $\rightarrow$ r-popj-okp (ins, r))
$\wedge$ ((car (ins) = '`set-local`) $\rightarrow$ r-set-local-okp (ins, r))
$\wedge$ ((car (ins) = '`set-global`) $\rightarrow$ r-set-global-okp (ins, r))
$\wedge$ ((car (ins) = '`test-nat-and-jump`)
  $\rightarrow$ r-test-nat-and-jump-okp (ins, r))
$\wedge$ ((car (ins) = '`test-int-and-jump`)
  $\rightarrow$ r-test-int-and-jump-okp (ins, r))
$\wedge$ ((car (ins) = '`test-bool-and-jump`)
  $\rightarrow$ r-test-bool-and-jump-okp (ins, r))
$\wedge$ ((car (ins) = '`test-bitv-and-jump`)
  $\rightarrow$ r-test-bitv-and-jump-okp (ins, r))
$\wedge$ ((car (ins) = '`no-op`) $\rightarrow$ r-no-op-okp (ins, r))
$\wedge$ ((car (ins) = '`add-addr`) $\rightarrow$ r-add-addr-okp (ins, r))
$\wedge$ ((car (ins) = '`sub-addr`) $\rightarrow$ r-sub-addr-okp (ins, r))
$\wedge$ ((car (ins) = '`eq`) $\rightarrow$ r-eq-okp (ins, r))
$\wedge$ ((car (ins) = '`lt-addr`) $\rightarrow$ r-lt-addr-okp (ins, r))
$\wedge$ ((car (ins) = '`fetch`) $\rightarrow$ r-fetch-okp (ins, r))
$\wedge$ ((car (ins) = '`deposit`) $\rightarrow$ r-deposit-okp (ins, r))
$\wedge$ ((car (ins) = '`add-int`) $\rightarrow$ r-add-int-okp (ins, r))
$\wedge$ ((car (ins) = '`add-int-with-carry`)
  $\rightarrow$ r-add-int-with-carry-okp (ins, r))
$\wedge$ ((car (ins) = '`add1-int`) $\rightarrow$ r-add1-int-okp (ins, r))
$\wedge$ ((car (ins) = '`sub-int`) $\rightarrow$ r-sub-int-okp (ins, r))
$\wedge$ ((car (ins) = '`sub-int-with-carry`)
  $\rightarrow$ r-sub-int-with-carry-okp (ins, r))
$\wedge$ ((car (ins) = '`sub1-int`) $\rightarrow$ r-sub1-int-okp (ins, r))
$\wedge$ ((car (ins) = '`neg-int`) $\rightarrow$ r-neg-int-okp (ins, r))
$\wedge$ ((car (ins) = '`lt-int`) $\rightarrow$ r-lt-int-okp (ins, r))
$\wedge$ ((car (ins) = '`int-to-nat`) $\rightarrow$ r-int-to-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`add-nat`) $\rightarrow$ r-add-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`add-nat-with-carry`)
  $\rightarrow$ r-add-nat-with-carry-okp (ins, r))
$\wedge$ ((car (ins) = '`add1-nat`) $\rightarrow$ r-add1-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`sub-nat`) $\rightarrow$ r-sub-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`sub-nat-with-carry`)
  $\rightarrow$ r-sub-nat-with-carry-okp (ins, r))
$\wedge$ ((car (ins) = '`sub1-nat`) $\rightarrow$ r-sub1-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`lt-nat`) $\rightarrow$ r-lt-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`mult2-nat`) $\rightarrow$ r-mult2-nat-okp (ins, r))
$\wedge$ ((car (ins) = '`mult2-nat-with-carry-out`)
  $\rightarrow$ r-mult2-nat-with-carry-out-okp (ins, r))
$\wedge$ ((car (ins) = '`div2-nat`) $\rightarrow$ r-div2-nat-okp (ins, r))

$\wedge$ ((car (ins) = 'or-bitv) $\rightarrow$ r-or-bitv-okp (ins, r))
$\wedge$ ((car (ins) = 'and-bitv) $\rightarrow$ r-and-bitv-okp (ins, r))
$\wedge$ ((car (ins) = 'not-bitv) $\rightarrow$ r-not-bitv-okp (ins, r))
$\wedge$ ((car (ins) = 'xor-bitv) $\rightarrow$ r-xor-bitv-okp (ins, r))
$\wedge$ ((car (ins) = 'rsh-bitv) $\rightarrow$ r-rsh-bitv-okp (ins, r))
$\wedge$ ((car (ins) = 'lsh-bitv) $\rightarrow$ r-lsh-bitv-okp (ins, r))
$\wedge$ ((car (ins) = 'or-bool) $\rightarrow$ r-or-bool-okp (ins, r))
$\wedge$ ((car (ins) = 'and-bool) $\rightarrow$ r-and-bool-okp (ins, r))
$\wedge$ ((car (ins) = 'not-bool) $\rightarrow$ r-not-bool-okp (ins, r)))

EVENT: Disable r-ins-okp.

```
; The proof strategy I have in mind requires that the machine be able
; to walk all the way down the IF tree inside of a single r-ins-step.
; Unfortunately, we run out of stack space before we get there.  The
; solution is to define new versions of r-ins-step and
; r-ins-step-clock that are composed of two functions, each of which
; handles half of the instructions.
```

DEFINITION:
r-ins-step2 (ins, r)
= **case on** car (ins):
case = *eq*
then r-eq-step (ins, r)
case = *lt-addr*
then r-lt-addr-step (ins, r)
case = *fetch*
then r-fetch-step (ins, r)
case = *deposit*
then r-deposit-step (ins, r)
case = *add-int*
then r-add-int-step (ins, r)
case = *add-int-with-carry*
then r-add-int-with-carry-step (ins, r)
case = *add1-int*
then r-add1-int-step (ins, r)
case = *sub-int*
then r-sub-int-step (ins, r)
case = *sub-int-with-carry*
then r-sub-int-with-carry-step (ins, r)
case = *sub1-int*
then r-sub1-int-step (ins, r)
case = *neg-int*

then r-neg-int-step (*ins*, *r*)
case = *lt-int*
then r-lt-int-step (*ins*, *r*)
case = *int-to-nat*
then r-int-to-nat-step (*ins*, *r*)
case = *add-nat*
then r-add-nat-step (*ins*, *r*)
case = *add-nat-with-carry*
then r-add-nat-with-carry-step (*ins*, *r*)
case = *add1-nat*
then r-add1-nat-step (*ins*, *r*)
case = *sub-nat*
then r-sub-nat-step (*ins*, *r*)
case = *sub-nat-with-carry*
then r-sub-nat-with-carry-step (*ins*, *r*)
case = *sub1-nat*
then r-sub1-nat-step (*ins*, *r*)
case = *lt-nat*
then r-lt-nat-step (*ins*, *r*)
case = *mult2-nat*
then r-mult2-nat-step (*ins*, *r*)
case = *mult2-nat-with-carry-out*
then r-mult2-nat-with-carry-out-step (*ins*, *r*)
case = *div2-nat*
then r-div2-nat-step (*ins*, *r*)
case = *or-bitv*
then r-or-bitv-step (*ins*, *r*)
case = *and-bitv*
then r-and-bitv-step (*ins*, *r*)
case = *not-bitv*
then r-not-bitv-step (*ins*, *r*)
case = *xor-bitv*
then r-xor-bitv-step (*ins*, *r*)
case = *rsh-bitv*
then r-rsh-bitv-step (*ins*, *r*)
case = *lsh-bitv*
then r-lsh-bitv-step (*ins*, *r*)
case = *or-bool*
then r-or-bool-step (*ins*, *r*)
case = *and-bool*
then r-and-bool-step (*ins*, *r*)
case = *not-bool*
then r-not-bool-step (*ins*, *r*)
otherwise r-halt (*r*, `'run`) **endcase**

DEFINITION:
r-ins-step1 (*ins*, *r*)
= **case on** car (*ins*):
case = *call*
then r-call-step (*ins*, *r*)
case = *ret*
then r-ret-step (*ins*, *r*)
case = *locn*
then r-locn-step (*ins*, *r*)
case = *push-constant*
then r-push-constant-step (*ins*, *r*)
case = *push-local*
then r-push-local-step (*ins*, *r*)
case = *push-global*
then r-push-global-step (*ins*, *r*)
case = *push-ctrl-stk-free-size*
then r-push-ctrl-stk-free-size-step (*ins*, *r*)
case = *push-temp-stk-free-size*
then r-push-temp-stk-free-size-step (*ins*, *r*)
case = *push-temp-stk-index*
then r-push-temp-stk-index-step (*ins*, *r*)
case = *jump-if-temp-stk-full*
then r-jump-if-temp-stk-full-step (*ins*, *r*)
case = *jump-if-temp-stk-empty*
then r-jump-if-temp-stk-empty-step (*ins*, *r*)
case = *pop*
then r-pop-step (*ins*, *r*)
case = *pop**
then r-pop*-step (*ins*, *r*)
case = *popn*
then r-popn-step (*ins*, *r*)
case = *pop-local*
then r-pop-local-step (*ins*, *r*)
case = *pop-global*
then r-pop-global-step (*ins*, *r*)
case = *pop-locn*
then r-pop-locn-step (*ins*, *r*)
case = *pop-call*
then r-pop-call-step (*ins*, *r*)
case = *fetch-temp-stk*
then r-fetch-temp-stk-step (*ins*, *r*)
case = *deposit-temp-stk*
then r-deposit-temp-stk-step (*ins*, *r*)
case = *jump*

then r-jump-step (*ins*, *r*)
case = *jump-case*
then r-jump-case-step (*ins*, *r*)
case = *pushj*
then r-pushj-step (*ins*, *r*)
case = *popj*
then r-popj-step (*ins*, *r*)
case = *set-local*
then r-set-local-step (*ins*, *r*)
case = *set-global*
then r-set-global-step (*ins*, *r*)
case = *test-nat-and-jump*
then r-test-nat-and-jump-step (*ins*, *r*)
case = *test-int-and-jump*
then r-test-int-and-jump-step (*ins*, *r*)
case = *test-bool-and-jump*
then r-test-bool-and-jump-step (*ins*, *r*)
case = *test-bitv-and-jump*
then r-test-bitv-and-jump-step (*ins*, *r*)
case = *no-op*
then r-no-op-step (*ins*, *r*)
case = *add-addr*
then r-add-addr-step (*ins*, *r*)
case = *sub-addr*
then r-sub-addr-step (*ins*, *r*)
otherwise r-ins-step2 (*ins*, *r*) **endcase**

DEFINITION:
r-ins-step-clock2 (*ins*, *r*)
= **case on** car (*ins*):
case = *eq*
then r-eq-step-clock (*ins*, *r*)
case = *lt-addr*
then r-lt-addr-step-clock (*ins*, *r*)
case = *fetch*
then r-fetch-step-clock (*ins*, *r*)
case = *deposit*
then r-deposit-step-clock (*ins*, *r*)
case = *add-int*
then r-add-int-step-clock (*ins*, *r*)
case = *add-int-with-carry*
then r-add-int-with-carry-step-clock (*ins*, *r*)
case = *add1-int*
then r-add1-int-step-clock (*ins*, *r*)

case = *sub-int*
 then r-sub-int-step-clock (*ins*, *r*)
case = *sub-int-with-carry*
 then r-sub-int-with-carry-step-clock (*ins*, *r*)
case = *sub1-int*
 then r-sub1-int-step-clock (*ins*, *r*)
case = *neg-int*
 then r-neg-int-step-clock (*ins*, *r*)
case = *lt-int*
 then r-lt-int-step-clock (*ins*, *r*)
case = *int-to-nat*
 then r-int-to-nat-step-clock (*ins*, *r*)
case = *add-nat*
 then r-add-nat-step-clock (*ins*, *r*)
case = *add-nat-with-carry*
 then r-add-nat-with-carry-step-clock (*ins*, *r*)
case = *add1-nat*
 then r-add1-nat-step-clock (*ins*, *r*)
case = *sub-nat*
 then r-sub-nat-step-clock (*ins*, *r*)
case = *sub-nat-with-carry*
 then r-sub-nat-with-carry-step-clock (*ins*, *r*)
case = *sub1-nat*
 then r-sub1-nat-step-clock (*ins*, *r*)
case = *lt-nat*
 then r-lt-nat-step-clock (*ins*, *r*)
case = *mult2-nat*
 then r-mult2-nat-step-clock (*ins*, *r*)
case = *mult2-nat-with-carry-out*
 then r-mult2-nat-with-carry-out-step-clock (*ins*, *r*)
case = *div2-nat*
 then r-div2-nat-step-clock (*ins*, *r*)
case = *or-bitv*
 then r-or-bitv-step-clock (*ins*, *r*)
case = *and-bitv*
 then r-and-bitv-step-clock (*ins*, *r*)
case = *not-bitv*
 then r-not-bitv-step-clock (*ins*, *r*)
case = *xor-bitv*
 then r-xor-bitv-step-clock (*ins*, *r*)
case = *rsh-bitv*
 then r-rsh-bitv-step-clock (*ins*, *r*)
case = *lsh-bitv*
 then r-lsh-bitv-step-clock (*ins*, *r*)

case = *or-bool*
  then r-or-bool-step-clock (*ins*, *r*)
case = *and-bool*
  then r-and-bool-step-clock (*ins*, *r*)
case = *not-bool*
  then r-not-bool-step-clock (*ins*, *r*)
otherwise 0 **endcase**

DEFINITION:
r-ins-step-clock1 (*ins*, *r*)
= **case on** car (*ins*):
case = *call*
then r-call-step-clock (*ins*, *r*)
case = *ret*
  then r-ret-step-clock (*ins*, *r*)
case = *locn*
  then r-locn-step-clock (*ins*, *r*)
case = *push-constant*
  then r-push-constant-step-clock (*ins*, *r*)
case = *push-local*
  then r-push-local-step-clock (*ins*, *r*)
case = *push-global*
  then r-push-global-step-clock (*ins*, *r*)
case = *push-ctrl-stk-free-size*
  then r-push-ctrl-stk-free-size-step-clock (*ins*, *r*)
case = *push-temp-stk-free-size*
  then r-push-temp-stk-free-size-step-clock (*ins*, *r*)
case = *push-temp-stk-index*
  then r-push-temp-stk-index-step-clock (*ins*, *r*)
case = *jump-if-temp-stk-full*
  then r-jump-if-temp-stk-full-step-clock (*ins*, *r*)
case = *jump-if-temp-stk-empty*
  then r-jump-if-temp-stk-empty-step-clock (*ins*, *r*)
case = *pop*
  then r-pop-step-clock (*ins*, *r*)
case = *pop**
  then r-pop*-step-clock (*ins*, *r*)
case = *popn*
  then r-popn-step-clock (*ins*, *r*)
case = *pop-local*
  then r-pop-local-step-clock (*ins*, *r*)
case = *pop-global*
  then r-pop-global-step-clock (*ins*, *r*)
case = *pop-locn*

then r-pop-locn-step-clock (ins, r)
case = *pop-call*
then r-pop-call-step-clock (ins, r)
case = *fetch-temp-stk*
then r-fetch-temp-stk-step-clock (ins, r)
case = *deposit-temp-stk*
then r-deposit-temp-stk-step-clock (ins, r)
case = *jump*
then r-jump-step-clock (ins, r)
case = *jump-case*
then r-jump-case-step-clock (ins, r)
case = *pushj*
then r-pushj-step-clock (ins, r)
case = *popj*
then r-popj-step-clock (ins, r)
case = *set-local*
then r-set-local-step-clock (ins, r)
case = *set-global*
then r-set-global-step-clock (ins, r)
case = *test-nat-and-jump*
then r-test-nat-and-jump-step-clock (ins, r)
case = *test-int-and-jump*
then r-test-int-and-jump-step-clock (ins, r)
case = *test-bool-and-jump*
then r-test-bool-and-jump-step-clock (ins, r)
case = *test-bitv-and-jump*
then r-test-bitv-and-jump-step-clock (ins, r)
case = *no-op*
then r-no-op-step-clock (ins, r)
case = *add-addr*
then r-add-addr-step-clock (ins, r)
case = *sub-addr*
then r-sub-addr-step-clock (ins, r)
otherwise r-ins-step-clock2 (ins, r) **endcase**

`; We now prove these functions are equivalent to what we normally use:`

THEOREM: r-ins-step-is-r-ins-step1
r-ins-step (ins, r) = r-ins-step1 (ins, r)

THEOREM: r-ins-step-clock-is-r-ins-step-clock1
r-ins-step-clock (ins, r) = r-ins-step-clock1 (ins, r)

EVENT: Disable r-ins-step.

EVENT: Disable r-ins-step-clock.

```
; Now we will prove that r-step preserves proper-r-statep.

; Consider (proper-r-statep (r-step r) load-addr).  The r-step will expand to ifs.
; We'll bring the ifs out of proper-r-statep with:
```

THEOREM: proper-r-statep-if
proper-r-statep (**if** a **then** b
else c **endif**,
load-addr)
= **if** a **then** proper-r-statep (b, *load-addr*)
else proper-r-statep (c, *load-addr*) **endif**

```
; That will elevate the test a, which will be (r-ins-okp ins r) the first
; time and then particular instruction opcodes subsequently.  Under each
; particular assumption we'll have to prove (proper-r-statep b load-addr) where
; b will be the corresponding r-xxx-step.  Our just proved backchainer
; and our previously proved lemmas will hit each.  However, because we
; can't explore the full depth of r-ins-step, we will first do this for
; r-ins-step2 and then for r-step (which will expand into r-ins-step and
; hence into r-ins-step1, where it will then do the first half of the nest
; and then appeal to our r-ins-step2 work).
```

EVENT: Disable r-current-instruction.

THEOREM: proper-r-statep-r-ins-step2
(proper-r-statep (r, *load-addr*)
$\wedge$ (r-psw (r) = **'run**)
$\wedge$ r-ins-okp (r-current-instruction (r), r))
$\rightarrow$ proper-r-statep (r-ins-step2 (r-current-instruction (r), r), *load-addr*)

EVENT: Disable r-ins-step2.

THEOREM: proper-r-statep-r-step
proper-r-statep (r, *load-addr*) $\rightarrow$ proper-r-statep (r-step (r), *load-addr*)

```
; Note:  This was easier before I added Bill Young's requested 7 instructions.
; It was those 7 that put me beyond the depth of the stack!

; We repeat the exercise now, for one-way-correspondence.
```

```
; First we draw out the ifs.
```

THEOREM: r->i-if
r->i (**if** a **then** b
else c **endif**)
$=$ **if** a **then** r->i (b)
else r->i (c) **endif**

THEOREM: i-if
i (r,
if a **then** b
else c **endif**,
load-addr)
$=$ **if** a **then** i (r, b, *load-addr*)
else i (r, c, *load-addr*) **endif**

THEOREM: equal-if-if
(**if** a **then** b
else c **endif**
$=$ **if** a **then** d
else e **endif**)
$=$ **if** a **then** $b = d$
else $c = e$ **endif**

```
; Note how clever the above strategy is.  Without this lemma, which extracts
; the branches of two isomorphic if nests simultaneously from the arms of
; the equal, we would cross multiply the tests.

; Then we arrange for the error and termination cases.
```

THEOREM: i-0
i (i, 0, *load-addr*) $= i$

THEOREM: r-halt-run
(r-statep (r) $\wedge$ (r-psw (r) $=$ **'run**)) $\rightarrow$ (r-halt (r, **'run**) $= r$)

```
; Now we are really shooting for one-way-correspondence-r-i-step, below,
; but it has two occurrences of (r-step r) in it and I don't want to deal
; with them both at once.  So I'll first prove a mildly simplified version
; and then prove that if the (r-step r) is non-erroneous then the r-ins-okp
; below holds.
```

THEOREM: one-way-correspondence-r-i-step2-simplified
(proper-r-statep (r, *load-addr*)
 $\land$ (*load-addr* $\in$ **N**)
 $\land$ (r-psw (r) = **'run**)
 $\land$ r-ins-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-ins-step2 (r-current-instruction (r), r))
 = i (r->i (r),
 r-ins-step-clock2 (r-current-instruction (r), r),
 load-addr))

EVENT: Disable r-ins-step-clock2.

THEOREM: one-way-correspondence-r-i-step-simplified
(proper-r-statep (r, *load-addr*)
 $\land$ (*load-addr* $\in$ **N**)
 $\land$ r-ins-okp (r-current-instruction (r), r))
$\rightarrow$ (r->i (r-step (r)) = i (r->i (r), r-step-clock (r), *load-addr*))

THEOREM: r->i-r-halt
(r-psw (r) = **'run**) $\rightarrow$ (r->i (r-halt (r, **'halt**)) = r->i (r))

THEOREM: r-psw-r-halt
r-psw (r-halt (r, *psw*)) = *psw*

```
; The double occurrence of r-step in the thm below makes its direct proof
; very explosive.  So I proved the simplified version above, from which this
; version is easy.
```

THEOREM: one-way-correspondence-r-i-step
(proper-r-statep (r, *load-addr*)
 $\land$ (*load-addr* $\in$ **N**)
 $\land$ ($\neg$ errorp (r-psw (r-step (r)))))
$\rightarrow$ (r->i (r-step (r)) = i (r->i (r), r-step-clock (r), *load-addr*))

THEOREM: i-plus
i (i, $x + y$, *load-addr*) = i (i (i, x, *load-addr*), y, *load-addr*)

THEOREM: once-errorp-always-errorp-step
($\neg$ errorp (r-psw (r-step (r)))) $\rightarrow$ ($\neg$ errorp (r-psw (r)))

THEOREM: once-errorp-always-errorp
($\neg$ errorp (r-psw (r (r, n)))) $\rightarrow$ ($\neg$ errorp (r-psw (r)))

THEOREM: one-way-correspondence-r-i
(proper-r-statep (r, *load-addr*)
 $\wedge$ (*load-addr* $\in$ **N**)
 $\wedge$ ($\neg$ errorp (r-psw (r (r, n)))))
$\rightarrow$ (r->i (r (r, n)) = i (r->i (r), clock (r, n), *load-addr*))

THEOREM: proper-r-statep-r
proper-r-statep (r, *load-addr*) $\rightarrow$ proper-r-statep (r (r, n), *load-addr*)

THEOREM: all-r-usr-data-objectps-implies-all-i-usr-data-objectps
(proper-r-statep (r, *load-addr*) $\wedge$ all-r-usr-data-objectps (*lst*, r))
$\rightarrow$ all-i-usr-data-objectps (*lst*, r->i (r))

THEOREM: proper-r-usr-data-segmentp-implies-proper-i-usr-data-segmentp
(proper-r-statep (r, *load-addr*) $\wedge$ proper-r-usr-data-segmentp (*segment*, r))
$\rightarrow$ proper-i-usr-data-segmentp (*segment*, r->i (r))

THEOREM: proper-r-statep-implies-proper-i-user-data-segmentp
proper-r-statep (r, *load-addr*)
$\rightarrow$ proper-i-usr-data-segmentp (r-usr-data-segment (r), r->i (r))

```
; Book: p.events.  Finally, we define the p machine itself, the semantics
; of Piton.
```

EVENT: Set the status of all events. The status of each event is to be set as follows. Each 'boot-strap' satellite is to be returned to the initial condition. Each 'add-shell' satellite is to be enabled. enabled. Anything not otherwise mentioned is to be disabled. Name this event 'close-data-base-4'.

```
; We now build p-r.events from its subbooks.
```

DEFINITION:
p-halt (p, *psw*)
= p-state (p-pc (p),
  p-ctrl-stk (p),
  p-temp-stk (p),
  p-prog-segment (p),
  p-data-segment (p),
  p-max-ctrl-stk-size (p),
  p-max-temp-stk-size (p),
  p-word-size (p),
  psw)

DEFINITION: rget (n, lst) = get ((length (lst) − n) − 1, lst)

DEFINITION:
rput (val, n, lst) = put $(val$, (length (lst) − n) − 1, $lst)$

DEFINITION: push (x, stk) = cons (x, stk)

DEFINITION:
popn (n, x)
= **if** $n \simeq$ 0 **then** x
else popn (n − 1, cdr (x)) **endif**

DEFINITION: top1 (stk) = top (pop (stk))

DEFINITION: top2 (stk) = top (pop (pop (stk)))

DEFINITION:
all-find-labelp $(lab\text{-}lst, lst)$
= **if** $lab\text{-}lst \simeq$ **nil** **then** **t**
else find-labelp (car $(lab\text{-}lst)$, lst)
$\wedge$ all-find-labelp (cdr $(lab\text{-}lst)$, lst) **endif**

DEFINITION:
p-objectp (x, p)
= (listp (x)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ **case on** type (x):
case = *nat*
then small-naturalp (untag (x), p-word-size (p))
case = *int*
then small-integerp (untag (x), p-word-size (p))
case = *bitv*
then bit-vectorp (untag (x), p-word-size (p))
case = *bool*
then booleanp (untag (x))
case = *addr*
then adpp (untag (x), p-data-segment (p))
case = *pc*
then pcpp (untag (x), p-prog-segment (p))
case = *subr*
then definedp (untag (x), p-prog-segment (p))
otherwise **f** **endcase**)

DEFINITION:
p-objectp-type $(type, x, p)$ = ((type (x) = $type$) $\wedge$ p-objectp (x, p))

DEFINITION:
all-p-objectps (lst, p)
$=$ **if** $lst \simeq$ **nil** **then** $lst =$ **nil**
else p-objectp (car (lst), p) $\wedge$ all-p-objectps (cdr (lst), p) **endif**

DEFINITION: add1-p-pc (p) $=$ add1-addr (p-pc (p))

DEFINITION:
p-current-program (p) $=$ definition (area-name (p-pc (p)), p-prog-segment (p))

DEFINITION:
p-current-instruction (p)
$=$ unlabel (get (offset (p-pc (p)), program-body (p-current-program (p))))

DEFINITION: p-frame ($bindings$, $ret\text{-}pc$) $=$ list ($bindings$, $ret\text{-}pc$)

DEFINITION:
local-var-value (var, $ctrl\text{-}stk$) $=$ definiens (var, bindings (top ($ctrl\text{-}stk$)))

DEFINITION:
set-local-var-value (val, var, $ctrl\text{-}stk$)
$=$ push (p-frame (put-value (val, var, bindings (top ($ctrl\text{-}stk$))),
ret-pc (top ($ctrl\text{-}stk$))),
pop ($ctrl\text{-}stk$))

DEFINITION:
pair-formal-vars-with-actuals ($formal\text{-}vars$, $temp\text{-}stk$)
$=$ pairlist ($formal\text{-}vars$, rev (first-n (length ($formal\text{-}vars$), $temp\text{-}stk$)))

DEFINITION:
pair-temps-with-initial-values ($temp\text{-}var\text{-}dcls$)
$=$ **if** $temp\text{-}var\text{-}dcls \simeq$ **nil** **then** **nil**
else cons (cons (caar ($temp\text{-}var\text{-}dcls$), cadar ($temp\text{-}var\text{-}dcls$)),
pair-temps-with-initial-values (cdr ($temp\text{-}var\text{-}dcls$))) **endif**

DEFINITION:
make-p-call-frame ($formal\text{-}vars$, $temp\text{-}stk$, $temp\text{-}var\text{-}dcls$, $ret\text{-}pc$)
$=$ p-frame (append (pair-formal-vars-with-actuals ($formal\text{-}vars$, $temp\text{-}stk$),
pair-temps-with-initial-values ($temp\text{-}var\text{-}dcls$)),
$ret\text{-}pc$)

DEFINITION:
p-call-okp (ins, p)
$=$ ((p-max-ctrl-stk-size (p)
$\not<$ p-ctrl-stk-size (push (make-p-call-frame (formal-vars (definition (cadr (ins),
p-prog-segment (p))),

```
                                                    p-temp-stk (p),
                                                    temp-var-dcls (definition (cadr (ins),
                                                                                p-prog-segment (p))),
                                                    add1-addr (p-pc (p))),
                                       p-ctrl-stk (p))))
     ∧   (length (p-temp-stk (p))
           ≮   length (formal-vars (definition (cadr (ins),
                                                p-prog-segment (p))))))
```

DEFINITION:

```
p-call-step (ins, p)
=   p-state (tag ('pc, cons (cadr (ins), 0)),
            push (make-p-call-frame (formal-vars (definition (cadr (ins),
                                                              p-prog-segment (p))),
                                     p-temp-stk (p),
                                     temp-var-dcls (definition (cadr (ins),
                                                                p-prog-segment (p))),
                                     add1-addr (p-pc (p))),
                  p-ctrl-stk (p)),
            popn (length (formal-vars (definition (cadr (ins),
                                                   p-prog-segment (p)))),
                  p-temp-stk (p)),
            p-prog-segment (p),
            p-data-segment (p),
            p-max-ctrl-stk-size (p),
            p-max-temp-stk-size (p),
            p-word-size (p),
            'run)
```

DEFINITION: p-ret-okp (*ins*, *p*) = **t**

DEFINITION:

```
p-ret-step (ins, p)
=   if listp (pop (p-ctrl-stk (p)))
    then p-state (ret-pc (top (p-ctrl-stk (p))),
                  pop (p-ctrl-stk (p)),
                  p-temp-stk (p),
                  p-prog-segment (p),
                  p-data-segment (p),
                  p-max-ctrl-stk-size (p),
                  p-max-temp-stk-size (p),
                  p-word-size (p),
                  'run)
    else p-halt (p, 'halt) endif
```

DEFINITION:
p-locn-okp (ins, p)
= (p-objectp-type (**'nat**, local-var-value (cadr (ins), p-ctrl-stk (p)), p)
  ∧ (untag (local-var-value (cadr (ins), p-ctrl-stk (p)))
    < length (bindings (top (p-ctrl-stk (p)))))
  ∧ (length (p-temp-stk (p)) < p-max-temp-stk-size (p)))

DEFINITION:
p-locn-step (ins, p)
= p-state (add1-p-pc (p),
    p-ctrl-stk (p),
    push (cdr (get (untag (local-var-value (cadr (ins), p-ctrl-stk (p))),
        bindings (top (p-ctrl-stk (p))))),
      p-temp-stk (p)),
    p-prog-segment (p),
    p-data-segment (p),
    p-max-ctrl-stk-size (p),
    p-max-temp-stk-size (p),
    p-word-size (p),
    'run)

DEFINITION:
p-push-constant-okp (ins, p)
= (length (p-temp-stk (p)) < p-max-temp-stk-size (p))

DEFINITION:
unabbreviate-constant (c, p)
= **if** c = **'pc** **then** add1-p-pc (p)
  elseif $c \simeq$ **nil** **then** pc (c, p-current-program (p))
  else c **endif**

DEFINITION:
p-push-constant-step (ins, p)
= p-state (add1-p-pc (p),
    p-ctrl-stk (p),
    push (unabbreviate-constant (cadr (ins), p), p-temp-stk (p)),
    p-prog-segment (p),
    p-data-segment (p),
    p-max-ctrl-stk-size (p),
    p-max-temp-stk-size (p),
    p-word-size (p),
    'run)

DEFINITION:
p-push-local-okp (ins, p)
= (length (p-temp-stk (p)) < p-max-temp-stk-size (p))

DEFINITION:
p-push-local-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
  p-ctrl-stk (*p*),
  push (local-var-value (cadr (*ins*), p-ctrl-stk (*p*)), p-temp-stk (*p*)),
  p-prog-segment (*p*),
  p-data-segment (*p*),
  p-max-ctrl-stk-size (*p*),
  p-max-temp-stk-size (*p*),
  p-word-size (*p*),
  'run)

DEFINITION:
p-push-global-okp (*ins*, *p*)
= (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

DEFINITION:
p-push-global-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
  p-ctrl-stk (*p*),
  push (fetch (tag (**'addr**, cons (cadr (*ins*), 0)), p-data-segment (*p*)),
    p-temp-stk (*p*)),
  p-prog-segment (*p*),
  p-data-segment (*p*),
  p-max-ctrl-stk-size (*p*),
  p-max-temp-stk-size (*p*),
  p-word-size (*p*),
  'run)

DEFINITION:
p-push-ctrl-stk-free-size-okp (*ins*, *p*)
= (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

DEFINITION:
p-push-ctrl-stk-free-size-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
  p-ctrl-stk (*p*),
  push (tag (**'nat**,
      p-max-ctrl-stk-size (*p*)
      − p-ctrl-stk-size (p-ctrl-stk (*p*))),
    p-temp-stk (*p*)),
  p-prog-segment (*p*),
  p-data-segment (*p*),
  p-max-ctrl-stk-size (*p*),
  p-max-temp-stk-size (*p*),

p-word-size (p),
'**run**)

DEFINITION:
p-push-temp-stk-free-size-okp $(ins,\ p)$
= (length (p-temp-stk (p)) $<$ p-max-temp-stk-size (p))

DEFINITION:
p-push-temp-stk-free-size-step $(ins,\ p)$
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag ('**nat**,
p-max-temp-stk-size (p) $-$ length (p-temp-stk (p))),
p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'**run**)

DEFINITION:
p-push-temp-stk-index-okp $(ins,\ p)$
= ((length (p-temp-stk (p)) $<$ p-max-temp-stk-size (p))
$\wedge$ (cadr (ins) $<$ length (p-temp-stk (p))))

DEFINITION:
p-push-temp-stk-index-step $(ins,\ p)$
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag ('**nat**, (length (p-temp-stk (p)) $-$ cadr (ins)) $-$ 1),
p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'**run**)

DEFINITION: p-jump-if-temp-stk-full-okp $(ins,\ p)$ = **t**

DEFINITION:
p-jump-if-temp-stk-full-step $(ins,\ p)$
= p-state (**if** length (p-temp-stk (p)) = p-max-temp-stk-size (p)
then pc (cadr (ins), p-current-program (p))

else add1-p-pc (p) **endif**,
p-ctrl-stk (p),
p-temp-stk (p),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'**run**)

DEFINITION: p-jump-if-temp-stk-empty-okp (ins, p) = **t**

DEFINITION:
p-jump-if-temp-stk-empty-step (ins, p)
= p-state (**if** length (p-temp-stk (p)) $\simeq$ 0
then pc (cadr (ins), p-current-program (p))
else add1-p-pc (p) **endif**,
p-ctrl-stk (p),
p-temp-stk (p),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'**run**)

DEFINITION: p-pop-okp (ins, p) = listp (p-temp-stk (p))

DEFINITION:
p-pop-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'**run**)

DEFINITION:
p-pop*-okp (ins, p) = (length (p-temp-stk (p)) $\not<$ cadr (ins))

DEFINITION:
p-pop*-step (ins, p)

$$
\begin{aligned}
= \quad & \text{p-state}\,(\text{add1-p-pc}\,(p),\\
& \qquad \text{p-ctrl-stk}\,(p),\\
& \qquad \text{popn}\,(\text{cadr}\,(ins),\ \text{p-temp-stk}\,(p)),\\
& \qquad \text{p-prog-segment}\,(p),\\
& \qquad \text{p-data-segment}\,(p),\\
& \qquad \text{p-max-ctrl-stk-size}\,(p),\\
& \qquad \text{p-max-temp-stk-size}\,(p),\\
& \qquad \text{p-word-size}\,(p),\\
& \qquad \texttt{'run})
\end{aligned}
$$

DEFINITION:
p-popn-okp $(ins,\ p)$

$$
\begin{aligned}
= \quad & (\text{listp}\,(\text{p-temp-stk}\,(p))\\
& \ \wedge \quad \text{p-objectp-type}\,(\texttt{'nat},\ \text{top}\,(\text{p-temp-stk}\,(p)),\ p)\\
& \ \wedge \quad (\text{length}\,(\text{p-temp-stk}\,(p)) \not< (1 + \text{untag}\,(\text{top}\,(\text{p-temp-stk}\,(p))))))
\end{aligned}
$$

DEFINITION:
p-popn-step $(ins,\ p)$

$$
\begin{aligned}
= \quad & \text{p-state}\,(\text{add1-p-pc}\,(p),\\
& \qquad \text{p-ctrl-stk}\,(p),\\
& \qquad \text{popn}\,(\text{untag}\,(\text{top}\,(\text{p-temp-stk}\,(p))),\ \text{pop}\,(\text{p-temp-stk}\,(p))),\\
& \qquad \text{p-prog-segment}\,(p),\\
& \qquad \text{p-data-segment}\,(p),\\
& \qquad \text{p-max-ctrl-stk-size}\,(p),\\
& \qquad \text{p-max-temp-stk-size}\,(p),\\
& \qquad \text{p-word-size}\,(p),\\
& \qquad \texttt{'run})
\end{aligned}
$$

DEFINITION: p-pop-local-okp $(ins,\ p) = \text{listp}\,(\text{p-temp-stk}\,(p))$

DEFINITION:
p-pop-local-step $(ins,\ p)$

$$
\begin{aligned}
= \quad & \text{p-state}\,(\text{add1-p-pc}\,(p),\\
& \qquad \text{set-local-var-value}\,(\text{top}\,(\text{p-temp-stk}\,(p)),\\
& \qquad \qquad \text{cadr}\,(ins),\\
& \qquad \qquad \text{p-ctrl-stk}\,(p)),\\
& \qquad \text{pop}\,(\text{p-temp-stk}\,(p)),\\
& \qquad \text{p-prog-segment}\,(p),\\
& \qquad \text{p-data-segment}\,(p),\\
& \qquad \text{p-max-ctrl-stk-size}\,(p),\\
& \qquad \text{p-max-temp-stk-size}\,(p),\\
& \qquad \text{p-word-size}\,(p),\\
& \qquad \texttt{'run})
\end{aligned}
$$

DEFINITION: p-pop-global-okp $(ins,\ p) = \text{listp}\,(\text{p-temp-stk}\,(p))$

DEFINITION:
p-pop-global-step (*ins*, *p*)

```
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             pop (p-temp-stk (p)),
             p-prog-segment (p),
             deposit (top (p-temp-stk (p)),
                      tag ('addr, cons (cadr (ins), 0)),
                      p-data-segment (p)),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run)
```

DEFINITION:
p-pop-locn-okp (*ins*, *p*)

```
=   (p-objectp-type ('nat, local-var-value (cadr (ins), p-ctrl-stk (p)), p)
     ∧  (untag (local-var-value (cadr (ins), p-ctrl-stk (p)))
          <  length (bindings (top (p-ctrl-stk (p)))))
     ∧  listp (p-temp-stk (p)))
```

DEFINITION:
put-value-indirect (*val*, *n*, *lst*)

```
=   if listp (lst)
    then if n ≃ 0 then cons (cons (caar (lst), val), cdr (lst))
         else cons (car (lst), put-value-indirect (val, n − 1, cdr (lst))) endif
    else lst endif
```

DEFINITION:
set-local-var-indirect (*val*, *index*, *ctrl-stk*)

```
=   push (p-frame (put-value-indirect (val, index, bindings (top (ctrl-stk))),
                   ret-pc (top (ctrl-stk))),
          pop (ctrl-stk))
```

DEFINITION:
p-pop-locn-step (*ins*, *p*)

```
=   p-state (add1-p-pc (p),
             set-local-var-indirect (top (p-temp-stk (p)),
                                     untag (local-var-value (cadr (ins),
                                                             p-ctrl-stk (p))),
                                     p-ctrl-stk (p)),
             pop (p-temp-stk (p)),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
```

p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-pop-call-okp (ins, p)
= (listp (p-temp-stk (p))
$\land$ p-objectp-type (**'subr**, top (p-temp-stk (p)), p)
$\land$ p-call-okp (list (**'call**, untag (top (p-temp-stk (p)))),
p-state (p-pc (p),
p-ctrl-stk (p),
pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)))

DEFINITION:
p-pop-call-step (ins, p)
= p-call-step (list (**'call**, untag (top (p-temp-stk (p)))),
p-state (p-pc (p),
p-ctrl-stk (p),
pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run))

DEFINITION:
p-fetch-temp-stk-okp (ins, p)
= (listp (p-temp-stk (p))
$\land$ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
$\land$ (untag (top (p-temp-stk (p))) $<$ length (p-temp-stk (p))))

DEFINITION:
p-fetch-temp-stk-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (rget (untag (top (p-temp-stk (p))), p-temp-stk (p)),
pop (p-temp-stk (p))),
p-prog-segment (p),

p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

Definition:
p-deposit-temp-stk-okp (ins, p)
= (listp (p-temp-stk (p))
∧ listp (pop (p-temp-stk (p)))
∧ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
∧ (untag (top (p-temp-stk (p)))
< length (pop (pop (p-temp-stk (p))))))

Definition:
p-deposit-temp-stk-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
rput (top1 (p-temp-stk (p)),
untag (top (p-temp-stk (p))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

Definition: p-jump-okp (ins, p) = **t**

Definition:
p-jump-step (ins, p)
= p-state (pc (cadr (ins), p-current-program (p)),
p-ctrl-stk (p),
p-temp-stk (p),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

Definition:
p-jump-case-okp (ins, p)
= (listp (p-temp-stk (p))

$\quad\wedge\quad$ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
$\quad\wedge\quad$ (untag (top (p-temp-stk (p))) $<$ length (cdr (ins))))

DEFINITION:
p-jump-case-step (ins, p)
= p-state (pc (get (untag (top (p-temp-stk (p))), cdr (ins)),
p-current-program (p)),
p-ctrl-stk (p),
pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-pushj-okp (ins, p) = (length (p-temp-stk (p)) $<$ p-max-temp-stk-size (p))

DEFINITION:
p-pushj-step (ins, p)
= p-state (pc (cadr (ins), p-current-program (p)),
p-ctrl-stk (p),
push (add1-p-pc (p), p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-popj-okp (ins, p)
= (listp (p-temp-stk (p))
$\quad\wedge\quad$ p-objectp-type (**'pc**, top (p-temp-stk (p)), p)
$\quad\wedge\quad$ (area-name (top (p-temp-stk (p))) = area-name (p-pc (p))))

DEFINITION:
p-popj-step (ins, p)
= p-state (top (p-temp-stk (p)),
p-ctrl-stk (p),
pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),

p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION: p-set-local-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:
p-set-local-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
set-local-var-value (top (p-temp-stk (*p*)),
cadr (*ins*),
p-ctrl-stk (*p*)),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION: p-set-global-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:
p-set-global-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
deposit (top (p-temp-stk (*p*)),
tag (**'addr**, cons (cadr (*ins*), 0)),
p-data-segment (*p*)),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
p-test-and-jump-okp (*ins*, *type*, *test*, *p*)
= (listp (p-temp-stk (*p*)) $\land$ p-objectp-type (*type*, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-test-and-jump-step (*test*, *lab*, *p*)
= **if** *test*
then p-state (pc (*lab*, p-current-program (*p*)),
p-ctrl-stk (*p*),

pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)
else p-state (add1-p-pc (p),
p-ctrl-stk (p),
pop (p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`) **endif**

Definition:
p-test-nat-and-jump-okp (ins, p)
= p-test-and-jump-okp (ins,
`'nat`,
p-test-natp (cadr (ins), untag (top (p-temp-stk (p)))),
p)

Definition:
p-test-nat-and-jump-step (ins, p)
= p-test-and-jump-step (p-test-natp (cadr (ins), untag (top (p-temp-stk (p)))),
caddr (ins),
p)

Definition:
p-test-int-and-jump-okp (ins, p)
= p-test-and-jump-okp (ins,
`'int`,
p-test-intp (cadr (ins), untag (top (p-temp-stk (p)))),
p)

Definition:
p-test-int-and-jump-step (ins, p)
= p-test-and-jump-step (p-test-intp (cadr (ins), untag (top (p-temp-stk (p)))),
caddr (ins),
p)

Definition:
p-test-bool-and-jump-okp (ins, p)

$=\quad$ p-test-and-jump-okp $(ins,$
$\qquad$ **'bool**,
$\qquad$ p-test-boolp (cadr (ins), untag (top (p-temp-stk (p)))),
$\qquad p)$

DEFINITION:
p-test-bool-and-jump-step $(ins,\ p)$
$=\quad$ p-test-and-jump-step (p-test-boolp (cadr (ins), untag (top (p-temp-stk (p)))),
$\qquad$ caddr (ins),
$\qquad p)$

DEFINITION:
p-test-bitv-and-jump-okp $(ins,\ p)$
$=\quad$ p-test-and-jump-okp $(ins,$
$\qquad$ **'bitv**,
$\qquad$ p-test-bitvp (cadr (ins), untag (top (p-temp-stk (p)))),
$\qquad p)$

DEFINITION:
p-test-bitv-and-jump-step $(ins,\ p)$
$=\quad$ p-test-and-jump-step (p-test-bitvp (cadr (ins), untag (top (p-temp-stk (p)))),
$\qquad$ caddr (ins),
$\qquad p)$

DEFINITION: p-no-op-okp $(ins,\ p) =$ **t**

DEFINITION:
p-no-op-step $(ins,\ p)$
$=\quad$ p-state (add1-p-pc (p),
$\qquad$ p-ctrl-stk (p),
$\qquad$ p-temp-stk (p),
$\qquad$ p-prog-segment (p),
$\qquad$ p-data-segment (p),
$\qquad$ p-max-ctrl-stk-size (p),
$\qquad$ p-max-temp-stk-size (p),
$\qquad$ p-word-size (p),
$\qquad$ **'run**)

DEFINITION:
p-add-addr-okp $(ins,\ p)$
$=\quad$ (listp (p-temp-stk (p))
$\quad\land\quad$ listp (pop (p-temp-stk (p)))
$\quad\land\quad$ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
$\quad\land\quad$ p-objectp-type (**'addr**, top1 (p-temp-stk (p)), p)
$\quad\land\quad$ p-objectp-type (**'addr**,

add-addr (top1 (p-temp-stk (p)),
    untag (top (p-temp-stk (p)))),
p))

DEFINITION:
p-add-addr-step (ins, p)
= p-state (add1-p-pc (p),
    p-ctrl-stk (p),
    push (add-addr (top1 (p-temp-stk (p)), untag (top (p-temp-stk (p)))),
        pop (pop (p-temp-stk (p)))),
    p-prog-segment (p),
    p-data-segment (p),
    p-max-ctrl-stk-size (p),
    p-max-temp-stk-size (p),
    p-word-size (p),
    `'run`)

DEFINITION:
p-sub-addr-okp (ins, p)
= (listp (p-temp-stk (p))
    $\wedge$ listp (pop (p-temp-stk (p)))
    $\wedge$ p-objectp-type (`'nat`, top (p-temp-stk (p)), p)
    $\wedge$ p-objectp-type (`'addr`, top1 (p-temp-stk (p)), p)
    $\wedge$ (offset (top1 (p-temp-stk (p))) $\not<$ untag (top (p-temp-stk (p)))))

DEFINITION:
p-sub-addr-step (ins, p)
= p-state (add1-p-pc (p),
    p-ctrl-stk (p),
    push (sub-addr (top1 (p-temp-stk (p)), untag (top (p-temp-stk (p)))),
        pop (pop (p-temp-stk (p)))),
    p-prog-segment (p),
    p-data-segment (p),
    p-max-ctrl-stk-size (p),
    p-max-temp-stk-size (p),
    p-word-size (p),
    `'run`)

DEFINITION:
p-eq-okp (ins, p)
= (listp (p-temp-stk (p))
    $\wedge$ listp (pop (p-temp-stk (p)))
    $\wedge$ (type (top (p-temp-stk (p))) = type (top1 (p-temp-stk (p)))))

DEFINITION:

p-eq-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (bool (untag (top1 (p-temp-stk (*p*)))
= untag (top (p-temp-stk (*p*)))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
p-lt-addr-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type (`'addr`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'addr`, top1 (p-temp-stk (*p*)), *p*)
∧ (area-name (top (p-temp-stk (*p*)))
= area-name (top1 (p-temp-stk (*p*)))))

DEFINITION:
p-lt-addr-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (bool (offset (top1 (p-temp-stk (*p*)))
< offset (top (p-temp-stk (*p*)))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
p-fetch-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'addr`, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-fetch-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),

```
            push (fetch (top (p-temp-stk (p)), p-data-segment (p)),
                  pop (p-temp-stk (p))),
            p-prog-segment (p),
            p-data-segment (p),
            p-max-ctrl-stk-size (p),
            p-max-temp-stk-size (p),
            p-word-size (p),
            'run)
```

DEFINITION:

```
p-deposit-okp (ins, p)
=   (listp (p-temp-stk (p))
     ∧  listp (pop (p-temp-stk (p)))
     ∧  p-objectp-type ('addr, top (p-temp-stk (p)), p))
```

DEFINITION:

```
p-deposit-step (ins, p)
=   p-state (add1-p-pc (p),
            p-ctrl-stk (p),
            pop (pop (p-temp-stk (p))),
            p-prog-segment (p),
            deposit (top1 (p-temp-stk (p)),
                     top (p-temp-stk (p)),
                     p-data-segment (p)),
            p-max-ctrl-stk-size (p),
            p-max-temp-stk-size (p),
            p-word-size (p),
            'run)
```

DEFINITION:

```
p-add-int-okp (ins, p)
=   (listp (p-temp-stk (p))
     ∧  listp (pop (p-temp-stk (p)))
     ∧  p-objectp-type ('int, top (p-temp-stk (p)), p)
     ∧  p-objectp-type ('int, top1 (p-temp-stk (p)), p)
     ∧  small-integerp (iplus (untag (top1 (p-temp-stk (p))),
                               untag (top (p-temp-stk (p)))),
                        p-word-size (p)))
```

DEFINITION:

```
p-add-int-step (ins, p)
=   p-state (add1-p-pc (p),
            p-ctrl-stk (p),
            push (tag ('int,
                       iplus (untag (top1 (p-temp-stk (p))),
```

untag (top (p-temp-stk (p))))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-add-int-with-carry-okp (ins, p)
= (listp (p-temp-stk (p))
∧ listp (pop (p-temp-stk (p)))
∧ listp (pop (pop (p-temp-stk (p))))
∧ p-objectp-type (**'int**, top (p-temp-stk (p)), p)
∧ p-objectp-type (**'int**, top1 (p-temp-stk (p)), p)
∧ p-objectp-type (**'bool**, top2 (p-temp-stk (p)), p))

DEFINITION:
p-add-int-with-carry-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (**'int**,
fix-small-integer (iplus (bool-to-nat (untag (top2 (p-temp-stk (p)))),
iplus (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p))))),
p-word-size (p))),
push (bool (¬ small-integerp (iplus (bool-to-nat (untag (top2 (p-temp-stk (p)))),
iplus (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p))))),
p-word-size (p))),
pop (pop (pop (p-temp-stk (p)))))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-add1-int-okp (ins, p)
= (listp (p-temp-stk (p))
∧ p-objectp-type (**'int**, top (p-temp-stk (p)), p)
∧ small-integerp (iplus (1, untag (top (p-temp-stk (p)))),
p-word-size (p)))

DEFINITION:
p-add1-int-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'int`, iplus (`1`, untag (top (p-temp-stk (p))))),
pop (p-temp-stk (p))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-sub-int-okp (ins, p)
= (listp (p-temp-stk (p))
$\land$ listp (pop (p-temp-stk (p)))
$\land$ p-objectp-type (`'int`, top (p-temp-stk (p)), p)
$\land$ p-objectp-type (`'int`, top1 (p-temp-stk (p)), p)
$\land$ small-integerp (idifference (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p)))),
p-word-size (p)))

DEFINITION:
p-sub-int-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'int`,
idifference (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p))))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-sub-int-with-carry-okp (ins, p)
= (listp (p-temp-stk (p))
$\land$ listp (pop (p-temp-stk (p)))
$\land$ listp (pop (pop (p-temp-stk (p))))
$\land$ p-objectp-type (`'int`, top (p-temp-stk (p)), p)
$\land$ p-objectp-type (`'int`, top1 (p-temp-stk (p)), p)

```
    ∧  p-objectp-type ('bool, top2 (p-temp-stk (p)), p))
```

DEFINITION:

```
p-sub-int-with-carry-step (ins, p)
=  p-state (add1-p-pc (p),
            p-ctrl-stk (p),
            push (tag ('int,
                       fix-small-integer (idifference (untag (top1 (p-temp-stk (p))),
                                                      iplus (untag (top (p-temp-stk (p))),
                                                             bool-to-nat (untag (top2 (p-temp-stk (p)))))),
                                          p-word-size (p))),
                  push (bool (¬ small-integerp (idifference (untag (top1 (p-temp-stk (p))),
                                                             iplus (untag (top (p-temp-stk (p))),
                                                                    bool-to-nat (untag (top2 (p-temp-stk (p)))))),
                                                p-word-size (p))),
                        pop (pop (pop (p-temp-stk (p)))))),
            p-prog-segment (p),
            p-data-segment (p),
            p-max-ctrl-stk-size (p),
            p-max-temp-stk-size (p),
            p-word-size (p),
            'run)
```

DEFINITION:

```
p-sub1-int-okp (ins, p)
=  (listp (p-temp-stk (p))
    ∧  p-objectp-type ('int, top (p-temp-stk (p)), p)
    ∧  small-integerp (idifference (untag (top (p-temp-stk (p))), 1),
                       p-word-size (p)))
```

DEFINITION:

```
p-sub1-int-step (ins, p)
=  p-state (add1-p-pc (p),
            p-ctrl-stk (p),
            push (tag ('int, idifference (untag (top (p-temp-stk (p))), 1)),
                  pop (p-temp-stk (p))),
            p-prog-segment (p),
            p-data-segment (p),
            p-max-ctrl-stk-size (p),
            p-max-temp-stk-size (p),
            p-word-size (p),
            'run)
```

DEFINITION:

```
p-neg-int-okp (ins, p)
```

$=$ (listp (p-temp-stk (p))
∧ p-objectp-type (`'int`, top (p-temp-stk (p)), p)
∧ small-integerp (inegate (untag (top (p-temp-stk (p)))),
p-word-size (p)))

DEFINITION:
p-neg-int-step (ins, p)
$=$ p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'int`, inegate (untag (top (p-temp-stk (p))))),
pop (p-temp-stk (p))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-lt-int-okp (ins, p)
$=$ (listp (p-temp-stk (p))
∧ listp (pop (p-temp-stk (p)))
∧ p-objectp-type (`'int`, top (p-temp-stk (p)), p)
∧ p-objectp-type (`'int`, top1 (p-temp-stk (p)), p))

DEFINITION:
p-lt-int-step (ins, p)
$=$ p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (bool (ilessp (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p))))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-int-to-nat-okp (ins, p)
$=$ (listp (p-temp-stk (p))
∧ p-objectp-type (`'int`, top (p-temp-stk (p)), p)
∧ (¬ negativep (untag (top (p-temp-stk (p))))))

DEFINITION:
p-int-to-nat-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'nat`, untag (top (p-temp-stk (*p*)))), pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
p-add-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type (`'nat`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'nat`, top1 (p-temp-stk (*p*)), *p*)
∧ small-naturalp (untag (top1 (p-temp-stk (*p*)))
+ untag (top (p-temp-stk (*p*))),
p-word-size (*p*)))

DEFINITION:
p-add-nat-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'nat`,
untag (top1 (p-temp-stk (*p*)))
+ untag (top (p-temp-stk (*p*)))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
p-add-nat-with-carry-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ listp (pop (pop (p-temp-stk (*p*))))
∧ p-objectp-type (`'nat`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'nat`, top1 (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'bool`, top2 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-add-nat-with-carry-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (**'nat**,
fix-small-natural (bool-to-nat (untag (top2 (p-temp-stk (p))))
+ untag (top1 (p-temp-stk (p)))
+ untag (top (p-temp-stk (p))),
p-word-size (p))),
push (bool ($\neg$ small-naturalp (bool-to-nat (untag (top2 (p-temp-stk (p))))
+ untag (top1 (p-temp-stk (p)))
+ untag (top (p-temp-stk (p))),
p-word-size (p))),
pop (pop (pop (p-temp-stk (p)))))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-add1-nat-okp (ins, p)
= (listp (p-temp-stk (p))
$\land$ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
$\land$ small-naturalp (1 + untag (top (p-temp-stk (p))), p-word-size (p)))

DEFINITION:
p-add1-nat-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (**'nat**, 1 + untag (top (p-temp-stk (p)))),
pop (p-temp-stk (p))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-sub-nat-okp (ins, p)
= (listp (p-temp-stk (p))
$\land$ listp (pop (p-temp-stk (p)))
$\land$ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)

$\wedge$ p-objectp-type (**'nat**, top1 (p-temp-stk (p)), p)
$\wedge$ (untag (top1 (p-temp-stk (p))) $\not<$ untag (top (p-temp-stk (p)))))

Definition:
p-sub-nat-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (**'nat**,
untag (top1 (p-temp-stk (p)))
− untag (top (p-temp-stk (p)))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

Definition:
p-sub-nat-with-carry-okp (ins, p)
= (listp (p-temp-stk (p))
$\wedge$ listp (pop (p-temp-stk (p)))
$\wedge$ listp (pop (pop (p-temp-stk (p))))
$\wedge$ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
$\wedge$ p-objectp-type (**'nat**, top1 (p-temp-stk (p)), p)
$\wedge$ p-objectp-type (**'bool**, top2 (p-temp-stk (p)), p))

Definition:
p-sub-nat-with-carry-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (**'nat**,
if untag (top1 (p-temp-stk (p)))
< (untag (top (p-temp-stk (p)))
+ bool-to-nat (untag (top2 (p-temp-stk (p)))))
then exp (2, p-word-size (p))
− ((untag (top (p-temp-stk (p)))
+ bool-to-nat (untag (top2 (p-temp-stk (p)))))
− untag (top1 (p-temp-stk (p))))
else untag (top1 (p-temp-stk (p)))
− (untag (top (p-temp-stk (p)))
+ bool-to-nat (untag (top2 (p-temp-stk (p))))) **endif**),
push (bool (untag (top1 (p-temp-stk (p)))
< (untag (top (p-temp-stk (p)))
+ bool-to-nat (untag (top2 (p-temp-stk (p)))))),

pop (pop (pop (p-temp-stk (p)))))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-sub1-nat-okp (ins, p)
= (listp (p-temp-stk (p))
∧ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
∧ (untag (top (p-temp-stk (p))) $\not\simeq$ 0))

DEFINITION:
p-sub1-nat-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (**'nat**, untag (top (p-temp-stk (p))) $-$ 1),
pop (p-temp-stk (p))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
p-lt-nat-okp (ins, p)
= (listp (p-temp-stk (p))
∧ listp (pop (p-temp-stk (p)))
∧ p-objectp-type (**'nat**, top (p-temp-stk (p)), p)
∧ p-objectp-type (**'nat**, top1 (p-temp-stk (p)), p))

DEFINITION:
p-lt-nat-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (bool (untag (top1 (p-temp-stk (p)))
$<$ untag (top (p-temp-stk (p))))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),

p-word-size (*p*),
'**run**)

DEFINITION:
p-mult2-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('**nat**, top (p-temp-stk (*p*)), *p*)
  ∧ small-naturalp (2 ∗ untag (top (p-temp-stk (*p*))),
      p-word-size (*p*)))

DEFINITION:
p-mult2-nat-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
    p-ctrl-stk (*p*),
    push (tag ('**nat**, 2 ∗ untag (top (p-temp-stk (*p*)))),
      pop (p-temp-stk (*p*))),
    p-prog-segment (*p*),
    p-data-segment (*p*),
    p-max-ctrl-stk-size (*p*),
    p-max-temp-stk-size (*p*),
    p-word-size (*p*),
    '**run**)

DEFINITION:
p-mult2-nat-with-carry-out-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('**nat**, top (p-temp-stk (*p*)), *p*)
  ∧ (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*)))

DEFINITION:
p-mult2-nat-with-carry-out-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
    p-ctrl-stk (*p*),
    push (tag ('**nat**,
        fix-small-natural (2 ∗ untag (top (p-temp-stk (*p*))),
          p-word-size (*p*))),
      push (bool (¬ small-naturalp (2 ∗ untag (top (p-temp-stk (*p*))),
          p-word-size (*p*))),
        pop (p-temp-stk (*p*)))),
    p-prog-segment (*p*),
    p-data-segment (*p*),
    p-max-ctrl-stk-size (*p*),
    p-max-temp-stk-size (*p*),
    p-word-size (*p*),
    '**run**)

DEFINITION:
p-div2-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (p))
∧ p-objectp-type (`'nat`, top (p-temp-stk (p)), p)
∧ (length (p-temp-stk (p)) < p-max-temp-stk-size (p)))

DEFINITION:
p-div2-nat-step (*ins*, *p*)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'nat`, untag (top (p-temp-stk (p))) **mod** 2),
push (tag (`'nat`, untag (top (p-temp-stk (p))) ÷ 2),
pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-or-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (p))
∧ listp (pop (p-temp-stk (p)))
∧ p-objectp-type (`'bitv`, top (p-temp-stk (p)), p)
∧ p-objectp-type (`'bitv`, top1 (p-temp-stk (p)), p))

DEFINITION:
p-or-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'bitv`,
or-bitv (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p))))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-and-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (p))

$\wedge$ listp (pop (p-temp-stk (p)))
$\wedge$ p-objectp-type (`'bitv`, top (p-temp-stk (p)), p)
$\wedge$ p-objectp-type (`'bitv`, top1 (p-temp-stk (p)), p))

DEFINITION:
p-and-bitv-step (*ins*, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'bitv`,
and-bitv (untag (top1 (p-temp-stk (p))),
untag (top (p-temp-stk (p))))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-not-bitv-okp (*ins*, p)
= (listp (p-temp-stk (p))
$\wedge$ p-objectp-type (`'bitv`, top (p-temp-stk (p)), p))

DEFINITION:
p-not-bitv-step (*ins*, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'bitv`, not-bitv (untag (top (p-temp-stk (p))))),
pop (p-temp-stk (p))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-xor-bitv-okp (*ins*, p)
= (listp (p-temp-stk (p))
$\wedge$ listp (pop (p-temp-stk (p)))
$\wedge$ p-objectp-type (`'bitv`, top (p-temp-stk (p)), p)
$\wedge$ p-objectp-type (`'bitv`, top1 (p-temp-stk (p)), p))

DEFINITION:

p-xor-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'bitv`,
xor-bitv (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*))))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
p-rsh-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'bitv`, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-rsh-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'bitv`, rsh-bitv (untag (top (p-temp-stk (*p*))))),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
p-lsh-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'bitv`, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-lsh-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'bitv`, lsh-bitv (untag (top (p-temp-stk (*p*))))),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),

```
                p-max-ctrl-stk-size (p),
                p-max-temp-stk-size (p),
                p-word-size (p),
                'run)
```

DEFINITION:

```
p-or-bool-okp (ins, p)
=   (listp (p-temp-stk (p))
      ∧  listp (pop (p-temp-stk (p)))
      ∧  p-objectp-type ('bool, top (p-temp-stk (p)), p)
      ∧  p-objectp-type ('bool, top1 (p-temp-stk (p)), p))
```

DEFINITION:

```
p-or-bool-step (ins, p)
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('bool,
                        or-bool (untag (top1 (p-temp-stk (p))),
                                 untag (top (p-temp-stk (p))))),
                   pop (pop (p-temp-stk (p)))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run)
```

DEFINITION:

```
p-and-bool-okp (ins, p)
=   (listp (p-temp-stk (p))
      ∧  listp (pop (p-temp-stk (p)))
      ∧  p-objectp-type ('bool, top (p-temp-stk (p)), p)
      ∧  p-objectp-type ('bool, top1 (p-temp-stk (p)), p))
```

DEFINITION:

```
p-and-bool-step (ins, p)
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('bool,
                        and-bool (untag (top1 (p-temp-stk (p))),
                                  untag (top (p-temp-stk (p))))),
                   pop (pop (p-temp-stk (p)))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
```

p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-not-bool-okp (ins, p)
= (listp (p-temp-stk (p))
∧ p-objectp-type (`'bool`, top (p-temp-stk (p)), p))

DEFINITION:
p-not-bool-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag (`'bool`, not-bool (untag (top (p-temp-stk (p))))),
pop (p-temp-stk (p))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
p-ins-okp (ins, p)
= **case on** car (ins):
case = *call*
then p-call-okp (ins, p)
case = *ret*
then p-ret-okp (ins, p)
case = *locn*
then p-locn-okp (ins, p)
case = *push-constant*
then p-push-constant-okp (ins, p)
case = *push-local*
then p-push-local-okp (ins, p)
case = *push-global*
then p-push-global-okp (ins, p)
case = *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-okp (ins, p)
case = *push-temp-stk-free-size*
then p-push-temp-stk-free-size-okp (ins, p)
case = *push-temp-stk-index*
then p-push-temp-stk-index-okp (ins, p)
case = *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-okp (ins, p)

case = *jump-if-temp-stk-empty*
 then p-jump-if-temp-stk-empty-okp (ins, p)
case = *pop*
 then p-pop-okp (ins, p)
case = *pop**
 then p-pop*-okp (ins, p)
case = *popn*
 then p-popn-okp (ins, p)
case = *pop-local*
 then p-pop-local-okp (ins, p)
case = *pop-global*
 then p-pop-global-okp (ins, p)
case = *pop-locn*
 then p-pop-locn-okp (ins, p)
case = *pop-call*
 then p-pop-call-okp (ins, p)
case = *fetch-temp-stk*
 then p-fetch-temp-stk-okp (ins, p)
case = *deposit-temp-stk*
 then p-deposit-temp-stk-okp (ins, p)
case = *jump*
 then p-jump-okp (ins, p)
case = *jump-case*
 then p-jump-case-okp (ins, p)
case = *pushj*
 then p-pushj-okp (ins, p)
case = *popj*
 then p-popj-okp (ins, p)
case = *set-local*
 then p-set-local-okp (ins, p)
case = *set-global*
 then p-set-global-okp (ins, p)
case = *test-nat-and-jump*
 then p-test-nat-and-jump-okp (ins, p)
case = *test-int-and-jump*
 then p-test-int-and-jump-okp (ins, p)
case = *test-bool-and-jump*
 then p-test-bool-and-jump-okp (ins, p)
case = *test-bitv-and-jump*
 then p-test-bitv-and-jump-okp (ins, p)
case = *no-op*
 then p-no-op-okp (ins, p)
case = *add-addr*
 then p-add-addr-okp (ins, p)

case = *sub-addr*
then p-sub-addr-okp (*ins*, *p*)
case = *eq*
then p-eq-okp (*ins*, *p*)
case = *lt-addr*
then p-lt-addr-okp (*ins*, *p*)
case = *fetch*
then p-fetch-okp (*ins*, *p*)
case = *deposit*
then p-deposit-okp (*ins*, *p*)
case = *add-int*
then p-add-int-okp (*ins*, *p*)
case = *add-int-with-carry*
then p-add-int-with-carry-okp (*ins*, *p*)
case = *add1-int*
then p-add1-int-okp (*ins*, *p*)
case = *sub-int*
then p-sub-int-okp (*ins*, *p*)
case = *sub-int-with-carry*
then p-sub-int-with-carry-okp (*ins*, *p*)
case = *sub1-int*
then p-sub1-int-okp (*ins*, *p*)
case = *neg-int*
then p-neg-int-okp (*ins*, *p*)
case = *lt-int*
then p-lt-int-okp (*ins*, *p*)
case = *int-to-nat*
then p-int-to-nat-okp (*ins*, *p*)
case = *add-nat*
then p-add-nat-okp (*ins*, *p*)
case = *add-nat-with-carry*
then p-add-nat-with-carry-okp (*ins*, *p*)
case = *add1-nat*
then p-add1-nat-okp (*ins*, *p*)
case = *sub-nat*
then p-sub-nat-okp (*ins*, *p*)
case = *sub-nat-with-carry*
then p-sub-nat-with-carry-okp (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-okp (*ins*, *p*)
case = *lt-nat*
then p-lt-nat-okp (*ins*, *p*)
case = *mult2-nat*
then p-mult2-nat-okp (*ins*, *p*)

case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-okp (*ins*, *p*)
case = *div2-nat*
then p-div2-nat-okp (*ins*, *p*)
case = *or-bitv*
then p-or-bitv-okp (*ins*, *p*)
case = *and-bitv*
then p-and-bitv-okp (*ins*, *p*)
case = *not-bitv*
then p-not-bitv-okp (*ins*, *p*)
case = *xor-bitv*
then p-xor-bitv-okp (*ins*, *p*)
case = *rsh-bitv*
then p-rsh-bitv-okp (*ins*, *p*)
case = *lsh-bitv*
then p-lsh-bitv-okp (*ins*, *p*)
case = *or-bool*
then p-or-bool-okp (*ins*, *p*)
case = *and-bool*
then p-and-bool-okp (*ins*, *p*)
case = *not-bool*
then p-not-bool-okp (*ins*, *p*)
otherwise f endcase

DEFINITION:
p-ins-step (*ins*, *p*)
= **case on** car (*ins*):
case = *call*
then p-call-step (*ins*, *p*)
case = *ret*
then p-ret-step (*ins*, *p*)
case = *locn*
then p-locn-step (*ins*, *p*)
case = *push-constant*
then p-push-constant-step (*ins*, *p*)
case = *push-local*
then p-push-local-step (*ins*, *p*)
case = *push-global*
then p-push-global-step (*ins*, *p*)
case = *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-step (*ins*, *p*)
case = *push-temp-stk-free-size*
then p-push-temp-stk-free-size-step (*ins*, *p*)
case = *push-temp-stk-index*

then p-push-temp-stk-index-step (ins, p)
case = *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-step (ins, p)
case = *jump-if-temp-stk-empty*
then p-jump-if-temp-stk-empty-step (ins, p)
case = *pop*
then p-pop-step (ins, p)
case = *pop**
then p-pop*-step (ins, p)
case = *popn*
then p-popn-step (ins, p)
case = *pop-local*
then p-pop-local-step (ins, p)
case = *pop-global*
then p-pop-global-step (ins, p)
case = *pop-locn*
then p-pop-locn-step (ins, p)
case = *pop-call*
then p-pop-call-step (ins, p)
case = *fetch-temp-stk*
then p-fetch-temp-stk-step (ins, p)
case = *deposit-temp-stk*
then p-deposit-temp-stk-step (ins, p)
case = *jump*
then p-jump-step (ins, p)
case = *jump-case*
then p-jump-case-step (ins, p)
case = *pushj*
then p-pushj-step (ins, p)
case = *popj*
then p-popj-step (ins, p)
case = *set-local*
then p-set-local-step (ins, p)
case = *set-global*
then p-set-global-step (ins, p)
case = *test-nat-and-jump*
then p-test-nat-and-jump-step (ins, p)
case = *test-int-and-jump*
then p-test-int-and-jump-step (ins, p)
case = *test-bool-and-jump*
then p-test-bool-and-jump-step (ins, p)
case = *test-bitv-and-jump*
then p-test-bitv-and-jump-step (ins, p)
case = *no-op*

then p-no-op-step $(ins,\ p)$
case $=$ *add-addr*
then p-add-addr-step $(ins,\ p)$
case $=$ *sub-addr*
then p-sub-addr-step $(ins,\ p)$
case $=$ *eq*
then p-eq-step $(ins,\ p)$
case $=$ *lt-addr*
then p-lt-addr-step $(ins,\ p)$
case $=$ *fetch*
then p-fetch-step $(ins,\ p)$
case $=$ *deposit*
then p-deposit-step $(ins,\ p)$
case $=$ *add-int*
then p-add-int-step $(ins,\ p)$
case $=$ *add-int-with-carry*
then p-add-int-with-carry-step $(ins,\ p)$
case $=$ *add1-int*
then p-add1-int-step $(ins,\ p)$
case $=$ *sub-int*
then p-sub-int-step $(ins,\ p)$
case $=$ *sub-int-with-carry*
then p-sub-int-with-carry-step $(ins,\ p)$
case $=$ *sub1-int*
then p-sub1-int-step $(ins,\ p)$
case $=$ *neg-int*
then p-neg-int-step $(ins,\ p)$
case $=$ *lt-int*
then p-lt-int-step $(ins,\ p)$
case $=$ *int-to-nat*
then p-int-to-nat-step $(ins,\ p)$
case $=$ *add-nat*
then p-add-nat-step $(ins,\ p)$
case $=$ *add-nat-with-carry*
then p-add-nat-with-carry-step $(ins,\ p)$
case $=$ *add1-nat*
then p-add1-nat-step $(ins,\ p)$
case $=$ *sub-nat*
then p-sub-nat-step $(ins,\ p)$
case $=$ *sub-nat-with-carry*
then p-sub-nat-with-carry-step $(ins,\ p)$
case $=$ *sub1-nat*
then p-sub1-nat-step $(ins,\ p)$
case $=$ *lt-nat*

then p-lt-nat-step (ins, p)
case = *mult2-nat*
then p-mult2-nat-step (ins, p)
case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-step (ins, p)
case = *div2-nat*
then p-div2-nat-step (ins, p)
case = *or-bitv*
then p-or-bitv-step (ins, p)
case = *and-bitv*
then p-and-bitv-step (ins, p)
case = *not-bitv*
then p-not-bitv-step (ins, p)
case = *xor-bitv*
then p-xor-bitv-step (ins, p)
case = *rsh-bitv*
then p-rsh-bitv-step (ins, p)
case = *lsh-bitv*
then p-lsh-bitv-step (ins, p)
case = *or-bool*
then p-or-bool-step (ins, p)
case = *and-bool*
then p-and-bool-step (ins, p)
case = *not-bool*
then p-not-bool-step (ins, p)
otherwise p-halt (p, `'run`) **endcase**

DEFINITION:
p-step1 (ins, p)
= **if** p-ins-okp (ins, p) **then** p-ins-step (ins, p)
else p-halt (p, x-y-error-msg (`'p`, car (ins))) **endif**

DEFINITION:
p-step (p)
= **if** p-psw (p) = `'run` **then** p-step1 (p-current-instruction (p), p)
else p **endif**

DEFINITION:
p (p, n)
= **if** $n \simeq$ `0` **then** p
else p (p-step (p), $n - 1$) **endif**

DEFINITION:
proper-p-call-instructionp (ins, $name$, p)
= ((length (ins) = `2`) $\land$ definedp (cadr (ins), p-prog-segment (p)))

Definition:
proper-p-ret-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = \texttt{1})$

Definition:
proper-p-locn-instructionp $(ins,\ name,\ p)$

$$
\begin{aligned}
= \quad & ((\text{length}\,(ins) = \texttt{2}) \\
& \wedge \quad (\text{cadr}\,(ins) \in \text{local-vars}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))
\end{aligned}
$$

Definition:
proper-p-push-constant-instructionp $(ins,\ name,\ p)$

$$
\begin{aligned}
= \quad & ((\text{length}\,(ins) = \texttt{2}) \\
& \wedge \quad (\text{p-objectp}\,(\text{cadr}\,(ins),\ p) \\
& \qquad \vee \quad (\text{cadr}\,(ins) = \texttt{'pc}) \\
& \qquad \vee \quad \text{find-labelp}\,(\text{cadr}\,(ins), \\
& \qquad\qquad\qquad \text{program-body}\,(\text{definition}\,(name, \\
& \qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))))
\end{aligned}
$$

Definition:
proper-p-push-local-instructionp $(ins,\ name,\ p)$

$$
\begin{aligned}
= \quad & ((\text{length}\,(ins) = \texttt{2}) \\
& \wedge \quad (\text{cadr}\,(ins) \in \text{local-vars}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))
\end{aligned}
$$

Definition:
proper-p-push-global-instructionp $(ins,\ name,\ p)$

$$= \quad ((\text{length}\,(ins) = \texttt{2}) \wedge \text{definedp}\,(\text{cadr}\,(ins),\ \text{p-data-segment}\,(p)))$$

Definition:
proper-p-push-ctrl-stk-free-size-instructionp $(ins,\ name,\ p)$

$$= \quad (\text{length}\,(ins) = \texttt{1})$$

Definition:
proper-p-push-temp-stk-free-size-instructionp $(ins,\ name,\ p)$

$$= \quad (\text{length}\,(ins) = \texttt{1})$$

Definition:
proper-p-push-temp-stk-index-instructionp $(ins,\ name,\ p)$

$$= \quad ((\text{length}\,(ins) = \texttt{2}) \wedge \text{small-naturalp}\,(\text{cadr}\,(ins),\ \text{p-word-size}\,(p)))$$

Definition:
proper-p-jump-if-temp-stk-full-instructionp $(ins,\ name,\ p)$

$$
\begin{aligned}
= \quad & ((\text{length}\,(ins) = \texttt{2}) \\
& \wedge \quad \text{find-labelp}\,(\text{cadr}\,(ins), \\
& \qquad\qquad \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))
\end{aligned}
$$

DEFINITION:
proper-p-jump-if-temp-stk-empty-instructionp (ins, $name$, p)
= ((length (ins) = 2)
∧ find-labelp (cadr (ins),
program-body (definition ($name$, p-prog-segment (p)))))

DEFINITION:
proper-p-pop-instructionp (ins, $name$, p) = (length (ins) = 1)

DEFINITION:
proper-p-pop*-instructionp (ins, $name$, p)
= ((length (ins) = 2) ∧ small-naturalp (cadr (ins), p-word-size (p)))

DEFINITION:
proper-p-popn-instructionp (ins, $name$, p) = (length (ins) = 1)

DEFINITION:
proper-p-pop-local-instructionp (ins, $name$, p)
= ((length (ins) = 2)
∧ (cadr (ins) ∈ local-vars (definition ($name$, p-prog-segment (p)))))

DEFINITION:
proper-p-pop-global-instructionp (ins, $name$, p)
= ((length (ins) = 2) ∧ definedp (cadr (ins), p-data-segment (p)))

DEFINITION:
proper-p-pop-locn-instructionp (ins, $name$, p)
= ((length (ins) = 2)
∧ (cadr (ins) ∈ local-vars (definition ($name$, p-prog-segment (p)))))

DEFINITION:
proper-p-pop-call-instructionp (ins, $name$, p) = (length (ins) = 1)

DEFINITION:
proper-p-fetch-temp-stk-instructionp (ins, $name$, p) = (length (ins) = 1)

DEFINITION:
proper-p-deposit-temp-stk-instructionp (ins, $name$, p) = (length (ins) = 1)

DEFINITION:
proper-p-jump-instructionp (ins, $name$, p)
= ((length (ins) = 2)
∧ find-labelp (cadr (ins),
program-body (definition ($name$, p-prog-segment (p)))))

DEFINITION:
proper-p-jump-case-instructionp $(ins,\ name,\ p)$
$$= \ (\text{listp}\,(\text{cdr}\,(ins)) \\ \quad \wedge \ \text{all-find-labelp}\,(\text{cdr}\,(ins),\ \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

DEFINITION:
proper-p-pushj-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 2) \\ \quad \wedge \ \text{find-labelp}\,(\text{cadr}\,(ins),\ \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

DEFINITION:
proper-p-popj-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

DEFINITION:
proper-p-set-local-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 2) \\ \quad \wedge \ (\text{cadr}\,(ins) \in \text{local-vars}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

DEFINITION:
proper-p-set-global-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 2) \wedge \text{definedp}\,(\text{cadr}\,(ins),\ \text{p-data-segment}\,(p)))$$

DEFINITION:
proper-p-test-nat-and-jump-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 3) \\ \quad \wedge \ \text{find-labelp}\,(\text{caddr}\,(ins),\ \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

DEFINITION:
proper-p-test-int-and-jump-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 3) \\ \quad \wedge \ \text{find-labelp}\,(\text{caddr}\,(ins),\ \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

DEFINITION:
proper-p-test-bool-and-jump-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 3) \\ \quad \wedge \ \text{find-labelp}\,(\text{caddr}\,(ins),\ \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

DEFINITION:
proper-p-test-bitv-and-jump-instructionp $(ins,\ name,\ p)$
$$= \ ((\text{length}\,(ins) = 3) \\ \quad \wedge \ \text{find-labelp}\,(\text{caddr}\,(ins),\ \text{program-body}\,(\text{definition}\,(name,\ \text{p-prog-segment}\,(p)))))$$

Definition:
proper-p-no-op-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-add-addr-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub-addr-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-eq-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-lt-addr-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-fetch-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-deposit-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-add-int-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-add-int-with-carry-instructionp $(ins, \ name, \ p)$
$= \quad (\text{length}\,(ins) = 1)$

Definition:
proper-p-add1-int-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub-int-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub-int-with-carry-instructionp $(ins, \ name, \ p)$
$= \quad (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub1-int-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-neg-int-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-lt-int-instructionp $(ins, \ name, \ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-int-to-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-add-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-add-nat-with-carry-instructionp $(ins,\ name,\ p)$
$= \quad (\text{length}\,(ins) = 1)$

Definition:
proper-p-add1-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub-nat-with-carry-instructionp $(ins,\ name,\ p)$
$= \quad (\text{length}\,(ins) = 1)$

Definition:
proper-p-sub1-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-lt-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-mult2-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-mult2-nat-with-carry-out-instructionp $(ins,\ name,\ p)$
$= \quad (\text{length}\,(ins) = 1)$

Definition:
proper-p-div2-nat-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-or-bitv-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-and-bitv-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-not-bitv-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-xor-bitv-instructionp $(ins,\ name,\ p) = (\text{length}\,(ins) = 1)$

Definition:
proper-p-rsh-bitv-instructionp (ins, $name$, p) = (length (ins) = 1)

Definition:
proper-p-lsh-bitv-instructionp (ins, $name$, p) = (length (ins) = 1)

Definition:
proper-p-or-bool-instructionp (ins, $name$, p) = (length (ins) = 1)

Definition:
proper-p-and-bool-instructionp (ins, $name$, p) = (length (ins) = 1)

Definition:
proper-p-not-bool-instructionp (ins, $name$, p) = (length (ins) = 1)

Definition:
proper-p-instructionp (ins, $name$, p)
= (properp (ins)
  ∧ **case on** car (ins):
    case = *call*
    then proper-p-call-instructionp (ins, $name$, p)
    case = *ret*
    then proper-p-ret-instructionp (ins, $name$, p)
    case = *locn*
    then proper-p-locn-instructionp (ins, $name$, p)
    case = *push-constant*
    then proper-p-push-constant-instructionp (ins, $name$, p)
    case = *push-local*
    then proper-p-push-local-instructionp (ins, $name$, p)
    case = *push-global*
    then proper-p-push-global-instructionp (ins, $name$, p)
    case = *push-ctrl-stk-free-size*
    then proper-p-push-ctrl-stk-free-size-instructionp (ins,
    $name$,
    p)
    case = *push-temp-stk-free-size*
    then proper-p-push-temp-stk-free-size-instructionp (ins,
    $name$,
    p)
    case = *push-temp-stk-index*
    then proper-p-push-temp-stk-index-instructionp (ins, $name$, p)
    case = *jump-if-temp-stk-full*
    then proper-p-jump-if-temp-stk-full-instructionp (ins, $name$, p)
    case = *jump-if-temp-stk-empty*
    then proper-p-jump-if-temp-stk-empty-instructionp (ins,

name,
p)
case = *pop*
then proper-p-pop-instructionp (*ins*, *name*, *p*)
case = *pop**
then proper-p-pop*-instructionp (*ins*, *name*, *p*)
case = *popn*
then proper-p-popn-instructionp (*ins*, *name*, *p*)
case = *pop-local*
then proper-p-pop-local-instructionp (*ins*, *name*, *p*)
case = *pop-global*
then proper-p-pop-global-instructionp (*ins*, *name*, *p*)
case = *pop-locn*
then proper-p-pop-locn-instructionp (*ins*, *name*, *p*)
case = *pop-call*
then proper-p-pop-call-instructionp (*ins*, *name*, *p*)
case = *fetch-temp-stk*
then proper-p-fetch-temp-stk-instructionp (*ins*, *name*, *p*)
case = *deposit-temp-stk*
then proper-p-deposit-temp-stk-instructionp (*ins*, *name*, *p*)
case = *jump*
then proper-p-jump-instructionp (*ins*, *name*, *p*)
case = *jump-case*
then proper-p-jump-case-instructionp (*ins*, *name*, *p*)
case = *pushj*
then proper-p-pushj-instructionp (*ins*, *name*, *p*)
case = *popj*
then proper-p-popj-instructionp (*ins*, *name*, *p*)
case = *set-local*
then proper-p-set-local-instructionp (*ins*, *name*, *p*)
case = *set-global*
then proper-p-set-global-instructionp (*ins*, *name*, *p*)
case = *test-nat-and-jump*
then proper-p-test-nat-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-int-and-jump*
then proper-p-test-int-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-bool-and-jump*
then proper-p-test-bool-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-bitv-and-jump*
then proper-p-test-bitv-and-jump-instructionp (*ins*, *name*, *p*)
case = *no-op*
then proper-p-no-op-instructionp (*ins*, *name*, *p*)
case = *add-addr*
then proper-p-add-addr-instructionp (*ins*, *name*, *p*)

case = *sub-addr*
then proper-p-sub-addr-instructionp (*ins*, *name*, *p*)
case = *eq*
then proper-p-eq-instructionp (*ins*, *name*, *p*)
case = *lt-addr*
then proper-p-lt-addr-instructionp (*ins*, *name*, *p*)
case = *fetch*
then proper-p-fetch-instructionp (*ins*, *name*, *p*)
case = *deposit*
then proper-p-deposit-instructionp (*ins*, *name*, *p*)
case = *add-int*
then proper-p-add-int-instructionp (*ins*, *name*, *p*)
case = *add-int-with-carry*
then proper-p-add-int-with-carry-instructionp (*ins*, *name*, *p*)
case = *add1-int*
then proper-p-add1-int-instructionp (*ins*, *name*, *p*)
case = *sub-int*
then proper-p-sub-int-instructionp (*ins*, *name*, *p*)
case = *sub-int-with-carry*
then proper-p-sub-int-with-carry-instructionp (*ins*, *name*, *p*)
case = *sub1-int*
then proper-p-sub1-int-instructionp (*ins*, *name*, *p*)
case = *neg-int*
then proper-p-neg-int-instructionp (*ins*, *name*, *p*)
case = *lt-int*
then proper-p-lt-int-instructionp (*ins*, *name*, *p*)
case = *int-to-nat*
then proper-p-int-to-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat*
then proper-p-add-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat-with-carry*
then proper-p-add-nat-with-carry-instructionp (*ins*, *name*, *p*)
case = *add1-nat*
then proper-p-add1-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat*
then proper-p-sub-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat-with-carry*
then proper-p-sub-nat-with-carry-instructionp (*ins*, *name*, *p*)
case = *sub1-nat*
then proper-p-sub1-nat-instructionp (*ins*, *name*, *p*)
case = *lt-nat*
then proper-p-lt-nat-instructionp (*ins*, *name*, *p*)
case = *mult2-nat*
then proper-p-mult2-nat-instructionp (*ins*, *name*, *p*)

case = *mult2-nat-with-carry-out*
then proper-p-mult2-nat-with-carry-out-instructionp (*ins*, *name*, *p*)
case = *div2-nat*
then proper-p-div2-nat-instructionp (*ins*, *name*, *p*)
case = *or-bitv*
then proper-p-or-bitv-instructionp (*ins*, *name*, *p*)
case = *and-bitv*
then proper-p-and-bitv-instructionp (*ins*, *name*, *p*)
case = *not-bitv*
then proper-p-not-bitv-instructionp (*ins*, *name*, *p*)
case = *xor-bitv*
then proper-p-xor-bitv-instructionp (*ins*, *name*, *p*)
case = *rsh-bitv*
then proper-p-rsh-bitv-instructionp (*ins*, *name*, *p*)
case = *lsh-bitv*
then proper-p-lsh-bitv-instructionp (*ins*, *name*, *p*)
case = *or-bool*
then proper-p-or-bool-instructionp (*ins*, *name*, *p*)
case = *and-bool*
then proper-p-and-bool-instructionp (*ins*, *name*, *p*)
case = *not-bool*
then proper-p-not-bool-instructionp (*ins*, *name*, *p*)
otherwise f endcase)

DEFINITION:
proper-labeled-p-instructionsp (*lst*, *name*, *p*)
= **if** *lst* $\simeq$ **nil then** *lst* = **nil**
else legal-labelp (car (*lst*))
$\land$ proper-p-instructionp (unlabel (car (*lst*)), *name*, *p*)
$\land$ proper-labeled-p-instructionsp (cdr (*lst*), *name*, *p*) **endif**

DEFINITION:
fall-off-proofp (*lst*)
= (car (unlabel (get (length (*lst*) $-$ 1, *lst*)))
$\in$ `'(ret jump jump-case popj)`)

DEFINITION:
proper-p-program-bodyp (*lst*, *name*, *p*)
= (listp (*lst*)
$\land$ proper-labeled-p-instructionsp (*lst*, *name*, *p*)
$\land$ fall-off-proofp (*lst*))

DEFINITION:

all-litatoms(lst)
$=$ **if** $lst \simeq$ **nil** **then** $lst =$ **nil**
else litatom (car (lst)) $\wedge$ all-litatoms (cdr (lst)) **endif**

DEFINITION:
proper-p-temp-var-dclsp $(temp\text{-}var\text{-}dcls,\ p)$
$=$ **if** $temp\text{-}var\text{-}dcls \simeq$ **nil** **then** **t**
else litatom (caar $(temp\text{-}var\text{-}dcls)$)
$\wedge$ p-objectp (cadar $(temp\text{-}var\text{-}dcls)$, p)
$\wedge$ proper-p-temp-var-dclsp (cdr $(temp\text{-}var\text{-}dcls)$, p) **endif**

DEFINITION:
proper-p-programp $(prog,\ p)$
$=$ (litatom (name $(prog)$)
$\wedge$ all-litatoms (formal-vars $(prog)$)
$\wedge$ proper-p-temp-var-dclsp (temp-var-dcls $(prog)$, p)
$\wedge$ proper-p-program-bodyp (program-body $(prog)$, name $(prog)$, p))

DEFINITION:
proper-p-prog-segmentp $(segment,\ p)$
$=$ **if** $segment \simeq$ **nil** **then** $segment =$ **nil**
else proper-p-programp (car $(segment)$, p)
$\wedge$ proper-p-prog-segmentp (cdr $(segment)$, p) **endif**

DEFINITION:
proper-p-alistp $(alist,\ p)$
$=$ **if** $alist \simeq$ **nil** **then** $alist =$ **nil**
else listp (car $(alist)$)
$\wedge$ litatom (caar $(alist)$)
$\wedge$ p-objectp (cdar $(alist)$, p)
$\wedge$ proper-p-alistp (cdr $(alist)$, p) **endif**

DEFINITION:
proper-p-framep $(frame,\ name,\ p)$
$=$ (listp $(frame)$
$\wedge$ listp (cdr $(frame)$)
$\wedge$ (cddr $(frame) =$ **nil**)
$\wedge$ proper-p-alistp (bindings $(frame)$, p)
$\wedge$ (strip-cars (bindings $(frame)$)
$=$ local-vars (definition $(name$, p-prog-segment (p))))
$\wedge$ p-objectp-type (`'pc`, ret-pc $(frame)$, p))

DEFINITION:
proper-p-ctrl-stkp $(ctrl\text{-}stk,\ name,\ p)$
$=$ **if** $ctrl\text{-}stk \simeq$ **nil** **then** $ctrl\text{-}stk =$ **nil**

else proper-p-framep (top ($ctrl\text{-}stk$), $name$, p)
$\wedge$ proper-p-ctrl-stkp (pop ($ctrl\text{-}stk$),
area-name (ret-pc (top ($ctrl\text{-}stk$))),
p) **endif**

DEFINITION:
proper-p-temp-stkp ($temp\text{-}stk$, p)
= **if** $temp\text{-}stk \simeq$ **nil then** $temp\text{-}stk =$ **nil**
else p-objectp (top ($temp\text{-}stk$), p)
$\wedge$ proper-p-temp-stkp (pop ($temp\text{-}stk$), p) **endif**

DEFINITION:
proper-p-area ($area$, p)
= (litatom (car ($area$))
$\wedge$ listp (cdr ($area$))
$\wedge$ all-p-objectps (cdr ($area$), p))

DEFINITION:
proper-p-data-segmentp ($data\text{-}segment$, p)
= **if** $data\text{-}segment \simeq$ **nil then** $data\text{-}segment =$ **nil**
else proper-p-area (car ($data\text{-}segment$), p)
$\wedge$ ($\neg$ definedp (caar ($data\text{-}segment$), cdr ($data\text{-}segment$)))
$\wedge$ proper-p-data-segmentp (cdr ($data\text{-}segment$), p) **endif**

DEFINITION:
proper-p-statep (p)
= (p-statep (p)
$\wedge$ p-objectp-type (`'pc`, p-pc (p), p)
$\wedge$ listp (p-ctrl-stk (p))
$\wedge$ proper-p-framep (top (p-ctrl-stk (p)), area-name (p-pc (p)), p)
$\wedge$ proper-p-ctrl-stkp (pop (p-ctrl-stk (p)),
area-name (ret-pc (top (p-ctrl-stk (p)))),
p)
$\wedge$ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p)))
$\wedge$ proper-p-temp-stkp (p-temp-stk (p), p)
$\wedge$ (p-max-temp-stk-size (p) $\not<$ length (p-temp-stk (p)))
$\wedge$ proper-p-prog-segmentp (p-prog-segment (p), p)
$\wedge$ proper-p-data-segmentp (p-data-segment (p), p)
$\wedge$ (p-max-ctrl-stk-size (p) $\in \mathbf{N}$)
$\wedge$ (p-max-temp-stk-size (p) $\in \mathbf{N}$)
$\wedge$ (p-word-size (p) $\in \mathbf{N}$)
$\wedge$ (p-max-ctrl-stk-size (p) $<$ exp (2, p-word-size (p)))
$\wedge$ (p-max-temp-stk-size (p) $<$ exp (2, p-word-size (p)))
$\wedge$ (0 $<$ p-word-size (p)))

```
; Book: p-r.events.  In this section we prove that the p machine
; is implemented on the r machine.
```

EVENT: Disable unlabel.

EVENT: Disable r-usr-data-typep.

EVENT: Disable r-instructionp.

```
; I will disable type, tag and untag for sanity's sake,
; but prove the obvious relationships first.
```

EVENT: Enable type-tag.

EVENT: Enable untag-tag.

THEOREM: equal-tags
$(\text{tag}\,(\mathit{type},\ x) = \text{tag}\,(\mathit{type},\ y)) = (x\ =\ y)$

EVENT: Enable cddr-tag.

EVENT: Disable type.

EVENT: Disable tag.

EVENT: Disable untag.

```
; The function BOOLEANP is a killer because it
; causes everything to split.  I will disable it
; and prove what I need about it as necessary.
```

EVENT: Disable booleanp.

```
; Error messages are irrelevant as long as they are not 'run or 'halt.
```

EVENT: Enable not-equal-x-y-error-msg-run.

EVENT: Enable not-equal-x-y-error-msg-halt.

EVENT: Disable x-y-error-msg.

EVENT: Enable r-psw-r-halt.

EVENT: Disable r-halt.

```
; Now I arrange for r-objectp to open when we know the type:
```

EVENT: Enable r-objectp-opener.

EVENT: Disable r-objectp.

EVENT: Disable r-usr-data-typep.

```
; We repeat the same exercise for p-objectp:
```

THEOREM: p-objectp-opener

$$
\begin{array}{l}
((\text{type}\,(x) = \texttt{'nat}) \\
\quad \rightarrow \quad (\text{p-objectp}\,(x,\ p) \\
\qquad\quad = \quad ((\text{cddr}\,(x) = \textbf{nil}) \\
\qquad\qquad\quad \wedge \quad \text{small-naturalp}\,(\text{untag}\,(x),\ \text{p-word-size}\,(p))))) \\
\wedge \quad ((\text{type}\,(x) = \texttt{'int}) \\
\qquad \rightarrow \quad (\text{p-objectp}\,(x,\ p) \\
\qquad\qquad = \quad ((\text{cddr}\,(x) = \textbf{nil}) \\
\qquad\qquad\qquad \wedge \quad \text{small-integerp}\,(\text{untag}\,(x),\ \text{p-word-size}\,(p))))) \\
\wedge \quad ((\text{type}\,(x) = \texttt{'bitv}) \\
\qquad \rightarrow \quad (\text{p-objectp}\,(x,\ p) \\
\qquad\qquad = \quad ((\text{cddr}\,(x) = \textbf{nil}) \\
\qquad\qquad\qquad \wedge \quad \text{bit-vectorp}\,(\text{untag}\,(x),\ \text{p-word-size}\,(p))))) \\
\wedge \quad ((\text{type}\,(x) = \texttt{'bool}) \\
\qquad \rightarrow \quad (\text{p-objectp}\,(x,\ p) \\
\qquad\qquad = \quad ((\text{cddr}\,(x) = \textbf{nil}) \wedge \text{booleanp}\,(\text{untag}\,(x))))) \\
\wedge \quad ((\text{type}\,(x) = \texttt{'addr}) \\
\qquad \rightarrow \quad (\text{p-objectp}\,(x,\ p)
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad = \quad ((\mathrm{cddr}\,(x) = \mathbf{nil}) \\
\quad\quad\quad\quad\quad \wedge \quad \mathrm{adpp}\,(\mathrm{untag}\,(x),\ \text{p-data-segment}\,(p))))) \\
\wedge \quad ((\mathrm{type}\,(x) = \texttt{'pc}) \\
\quad \rightarrow \quad (\text{p-objectp}\,(x,\ p) \\
\quad\quad\quad = \quad ((\mathrm{cddr}\,(x) = \mathbf{nil}) \\
\quad\quad\quad\quad\quad \wedge \quad \mathrm{pcpp}\,(\mathrm{untag}\,(x),\ \text{p-prog-segment}\,(p))))) \\
\wedge \quad ((\mathrm{type}\,(x) = \texttt{'subr}) \\
\quad \rightarrow \quad (\text{p-objectp}\,(x,\ p) \\
\quad\quad\quad = \quad ((\mathrm{cddr}\,(x) = \mathbf{nil}) \\
\quad\quad\quad\quad\quad \wedge \quad \mathrm{definedp}\,(\mathrm{untag}\,(x),\ \text{p-prog-segment}\,(p)))))
\end{array}
$$

EVENT: Disable p-objectp.

EVENT: Disable name.

EVENT: Disable formal-vars.

EVENT: Disable temp-var-dcls.

EVENT: Disable program-body.

THEOREM: bindings-p-frame
bindings (p-frame ($alist$, pc)) = $alist$

THEOREM: ret-pc-p-frame
ret-pc (p-frame ($alist$, pc)) = pc

EVENT: Disable bindings.

EVENT: Disable ret-pc.

EVENT: Disable p-frame.

EVENT: Enable associativity-of-plus.

EVENT: Enable commutativity-of-plus.

EVENT: Enable plus-add1.

EVENT: Enable commutativity2-of-plus.

```
; difference is used a lot in the stack pointer mappings and
; it is opened too often.  I want to control the expansion of
; difference carefully.
```

Event: Disable difference.

```
; The icompiler is involved in the characterization of loadablep
; but is actually irrelevant at this level.
; So we
```

Event: Disable icompile.

```
; to save the trouble of repeatedly considering expanding it.

; We need to get from (car (p-current-instruction p)) = 'no-op and
; (proper-p-statep p) the fact that
; (proper-p-no-op-instructionp (p-current-instruction p) name p).
; We will do it in two steps.  First we prove that proper-p-statep
; implies proper-p-instructionp for the current instruction.
; Then we'll arrange for proper-p-statep to expand and introduce
; the implied proper-p-instructionp too.
```

Event: Disable proper-p-instructionp.

Event: Disable legal-labelp.

Theorem: proper-labeled-p-instructionsp-implies-labelp-and-instructionp
(proper-labeled-p-instructionsp (lst, $name$, p) $\land$ ($x \in lst$))
$\rightarrow$ (legal-labelp (x) $\land$ proper-p-instructionp (unlabel (x), $name$, p))

Theorem: proper-p-prog-segmentp-implies-proper-p-programp
(proper-p-prog-segmentp ($segment$, p) $\land$ ($prog \in segment$))
$\rightarrow$ proper-p-programp ($prog$, p)

Event: Enable member-assoc.

Event: Enable name-assoc.

Event: Enable member-get.

THEOREM: proper-p-statep-implies-proper-p-instructionp
(proper-p-prog-segmentp (p-prog-segment (p), p)
∧ definedp (car (untag (p-pc (p))), p-prog-segment (p))
∧ (cdr (untag (p-pc (p)))
< length (program-body (assoc (car (untag (p-pc (p))),
p-prog-segment (p))))))
→ proper-p-instructionp (p-current-instruction (p), car (untag (p-pc (p))), p)

```
; Now I restructure proper-p-statep into two parts, those that
; have to do with arithmetic and those that do not.  The parts
; that do not are arithmetic, I confine to proper-p-statep1,
; which I keep closed and backchain into as needed.  The other
; parts I keep out in the open.  In addition, I throw in the
; proper-p-instructionp proved above, essentially forward
; chaining through the lemma.
```

EVENT: Disable p-loadablep.

DEFINITION:
proper-p-statep1 (p)
= (p-statep (p)
∧ (type (p-pc (p)) = `'pc`)
∧ listp (p-pc (p))
∧ (cddr (p-pc (p)) = **nil**)
∧ listp (untag (p-pc (p)))
∧ definedp (car (untag (p-pc (p))), p-prog-segment (p))
∧ listp (p-ctrl-stk (p))
∧ listp (car (p-ctrl-stk (p)))
∧ listp (cdr (car (p-ctrl-stk (p))))
∧ (cddr (car (p-ctrl-stk (p))) = **nil**)
∧ proper-p-alistp (bindings (car (p-ctrl-stk (p))), p)
∧ (strip-cars (bindings (car (p-ctrl-stk (p))))
= append (formal-vars (assoc (car (untag (p-pc (p))),
p-prog-segment (p))),
strip-cars (temp-var-dcls (assoc (car (untag (p-pc (p))),
p-prog-segment (p))))))
∧ (type (ret-pc (car (p-ctrl-stk (p)))) = `'pc`)
∧ listp (ret-pc (car (p-ctrl-stk (p))))
∧ (cddr (ret-pc (car (p-ctrl-stk (p)))) = **nil**)
∧ listp (untag (ret-pc (car (p-ctrl-stk (p)))))
∧ definedp (car (untag (ret-pc (car (p-ctrl-stk (p))))),
p-prog-segment (p))
∧ proper-p-ctrl-stkp (cdr (p-ctrl-stk (p)),

car (untag (ret-pc (car (p-ctrl-stk (p))))),
p)
∧ proper-p-temp-stkp (p-temp-stk (p), p)
∧ proper-p-prog-segmentp (p-prog-segment (p), p)
∧ proper-p-data-segmentp (p-data-segment (p), p))

THEOREM: proper-p-statep-restructuring
proper-p-statep (p)
= (proper-p-statep1 (p)
∧ (cdr (untag (p-pc (p))) $\in$ **N**)
∧ (cdr (untag (p-pc (p)))
< length (program-body (assoc (car (untag (p-pc (p))),
p-prog-segment (p)))))
∧ (cdr (untag (ret-pc (car (p-ctrl-stk (p))))) $\in$ **N**)
∧ (cdr (untag (ret-pc (car (p-ctrl-stk (p)))))
< length (program-body (assoc (car (untag (ret-pc (car (p-ctrl-stk (p))))),
p-prog-segment (p)))))
∧ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p)))
∧ (p-max-temp-stk-size (p) $\not<$ length (p-temp-stk (p)))
∧ (p-max-ctrl-stk-size (p) $\in$ **N**)
∧ (p-max-temp-stk-size (p) $\in$ **N**)
∧ (p-word-size (p) $\in$ **N**)
∧ (p-max-ctrl-stk-size (p) < exp (2, p-word-size (p)))
∧ (p-max-temp-stk-size (p) < exp (2, p-word-size (p)))
∧ (0 < p-word-size (p))
∧ proper-p-instructionp (p-current-instruction (p),
car (untag (p-pc (p))),
p))

THEOREM: proper-p-statep1-properties
proper-p-statep1 (p)
→ (p-statep (p)
∧ (type (p-pc (p)) = `'pc`)
∧ listp (p-pc (p))
∧ (cddr (p-pc (p)) = **nil**)
∧ listp (untag (p-pc (p)))
∧ definedp (car (untag (p-pc (p))), p-prog-segment (p))
∧ listp (p-ctrl-stk (p))
∧ listp (car (p-ctrl-stk (p)))
∧ listp (cdr (car (p-ctrl-stk (p))))
∧ (cddr (car (p-ctrl-stk (p))) = **nil**)
∧ proper-p-alistp (bindings (car (p-ctrl-stk (p))), p)
∧ (strip-cars (bindings (car (p-ctrl-stk (p))))
= append (formal-vars (assoc (car (untag (p-pc (p))),

p-prog-segment (p))),
strip-cars (temp-var-dcls (assoc (car (untag (p-pc (p))),
p-prog-segment (p))))))
∧ (type (ret-pc (car (p-ctrl-stk (p)))) = `'pc`)
∧ listp (ret-pc (car (p-ctrl-stk (p))))
∧ (cddr (ret-pc (car (p-ctrl-stk (p)))) = **nil**)
∧ listp (untag (ret-pc (car (p-ctrl-stk (p)))))
∧ definedp (car (untag (ret-pc (car (p-ctrl-stk (p))))),
p-prog-segment (p))
∧ proper-p-ctrl-stkp (cdr (p-ctrl-stk (p)),
car (untag (ret-pc (car (p-ctrl-stk (p))))),
p)
∧ proper-p-temp-stkp (p-temp-stk (p), p)
∧ proper-p-prog-segmentp (p-prog-segment (p), p)
∧ proper-p-data-segmentp (p-data-segment (p), p))

EVENT: Disable proper-p-statep.

EVENT: Disable proper-p-statep1.

EVENT: Disable p-current-instruction.

```
; Now I provide a simple delayed opener for proper-p-instructionp:
```

THEOREM: proper-p-instructionp-opener
(car (*ins*) = pack (*xxx*))
→ (proper-p-instructionp (*ins*, *name*, *p*)
= (properp (*ins*)
∧ **case on** car (*ins*):
case = *call*
then proper-p-call-instructionp (*ins*, *name*, *p*)
case = *ret*
then proper-p-ret-instructionp (*ins*, *name*, *p*)
case = *locn*
then proper-p-locn-instructionp (*ins*, *name*, *p*)
case = *push-constant*
then proper-p-push-constant-instructionp (*ins*, *name*, *p*)
case = *push-local*
then proper-p-push-local-instructionp (*ins*, *name*, *p*)
case = *push-global*
then proper-p-push-global-instructionp (*ins*, *name*, *p*)
case = *push-ctrl-stk-free-size*

then proper-p-push-ctrl-stk-free-size-instructionp (*ins*,
name,
p)
case = *push-temp-stk-free-size*
then proper-p-push-temp-stk-free-size-instructionp (*ins*,
name,
p)
case = *push-temp-stk-index*
then proper-p-push-temp-stk-index-instructionp (*ins*,
name,
p)
case = *jump-if-temp-stk-full*
then proper-p-jump-if-temp-stk-full-instructionp (*ins*,
name,
p)
case = *jump-if-temp-stk-empty*
then proper-p-jump-if-temp-stk-empty-instructionp (*ins*,
name,
p)
case = *pop*
then proper-p-pop-instructionp (*ins*, *name*, *p*)
case = *pop**
then proper-p-pop*-instructionp (*ins*, *name*, *p*)
case = *popn*
then proper-p-popn-instructionp (*ins*, *name*, *p*)
case = *pop-local*
then proper-p-pop-local-instructionp (*ins*, *name*, *p*)
case = *pop-global*
then proper-p-pop-global-instructionp (*ins*, *name*, *p*)
case = *pop-locn*
then proper-p-pop-locn-instructionp (*ins*, *name*, *p*)
case = *pop-call*
then proper-p-pop-call-instructionp (*ins*, *name*, *p*)
case = *fetch-temp-stk*
then proper-p-fetch-temp-stk-instructionp (*ins*,
name,
p)
case = *deposit-temp-stk*
then proper-p-deposit-temp-stk-instructionp (*ins*,
name,
p)
case = *jump*
then proper-p-jump-instructionp (*ins*, *name*, *p*)
case = *jump-case*

then proper-p-jump-case-instructionp (ins, $name$, p)
case = $pushj$
then proper-p-pushj-instructionp (ins, $name$, p)
case = $popj$
then proper-p-popj-instructionp (ins, $name$, p)
case = $set\text{-}local$
then proper-p-set-local-instructionp (ins, $name$, p)
case = $set\text{-}global$
then proper-p-set-global-instructionp (ins, $name$, p)
case = $test\text{-}nat\text{-}and\text{-}jump$
then proper-p-test-nat-and-jump-instructionp (ins,
$name$,
p)
case = $test\text{-}int\text{-}and\text{-}jump$
then proper-p-test-int-and-jump-instructionp (ins,
$name$,
p)
case = $test\text{-}bool\text{-}and\text{-}jump$
then proper-p-test-bool-and-jump-instructionp (ins,
$name$,
p)
case = $test\text{-}bitv\text{-}and\text{-}jump$
then proper-p-test-bitv-and-jump-instructionp (ins,
$name$,
p)
case = $no\text{-}op$
then proper-p-no-op-instructionp (ins, $name$, p)
case = $add\text{-}addr$
then proper-p-add-addr-instructionp (ins, $name$, p)
case = $sub\text{-}addr$
then proper-p-sub-addr-instructionp (ins, $name$, p)
case = eq
then proper-p-eq-instructionp (ins, $name$, p)
case = $lt\text{-}addr$
then proper-p-lt-addr-instructionp (ins, $name$, p)
case = $fetch$
then proper-p-fetch-instructionp (ins, $name$, p)
case = $deposit$
then proper-p-deposit-instructionp (ins, $name$, p)
case = $add\text{-}int$
then proper-p-add-int-instructionp (ins, $name$, p)
case = $add\text{-}int\text{-}with\text{-}carry$
then proper-p-add-int-with-carry-instructionp (ins,
$name$,

p)
case = *add1-int*
then proper-p-add1-int-instructionp (ins, $name$, p)
case = *sub-int*
then proper-p-sub-int-instructionp (ins, $name$, p)
case = *sub-int-with-carry*
then proper-p-sub-int-with-carry-instructionp (ins,
$name$,
p)
case = *sub1-int*
then proper-p-sub1-int-instructionp (ins, $name$, p)
case = *neg-int*
then proper-p-neg-int-instructionp (ins, $name$, p)
case = *lt-int*
then proper-p-lt-int-instructionp (ins, $name$, p)
case = *int-to-nat*
then proper-p-int-to-nat-instructionp (ins, $name$, p)
case = *add-nat*
then proper-p-add-nat-instructionp (ins, $name$, p)
case = *add-nat-with-carry*
then proper-p-add-nat-with-carry-instructionp (ins,
$name$,
p)
case = *add1-nat*
then proper-p-add1-nat-instructionp (ins, $name$, p)
case = *sub-nat*
then proper-p-sub-nat-instructionp (ins, $name$, p)
case = *sub-nat-with-carry*
then proper-p-sub-nat-with-carry-instructionp (ins,
$name$,
p)
case = *sub1-nat*
then proper-p-sub1-nat-instructionp (ins, $name$, p)
case = *lt-nat*
then proper-p-lt-nat-instructionp (ins, $name$, p)
case = *mult2-nat*
then proper-p-mult2-nat-instructionp (ins, $name$, p)
case = *mult2-nat-with-carry-out*
then proper-p-mult2-nat-with-carry-out-instructionp (ins,
$name$,
p)
case = *div2-nat*
then proper-p-div2-nat-instructionp (ins, $name$, p)
case = *or-bitv*

then proper-p-or-bitv-instructionp (*ins*, *name*, *p*)
case = *and-bitv*
then proper-p-and-bitv-instructionp (*ins*, *name*, *p*)
case = *not-bitv*
then proper-p-not-bitv-instructionp (*ins*, *name*, *p*)
case = *xor-bitv*
then proper-p-xor-bitv-instructionp (*ins*, *name*, *p*)
case = *rsh-bitv*
then proper-p-rsh-bitv-instructionp (*ins*, *name*, *p*)
case = *lsh-bitv*
then proper-p-lsh-bitv-instructionp (*ins*, *name*, *p*)
case = *or-bool*
then proper-p-or-bool-instructionp (*ins*, *name*, *p*)
case = *and-bool*
then proper-p-and-bool-instructionp (*ins*, *name*, *p*)
case = *not-bool*
then proper-p-not-bool-instructionp (*ins*, *name*, *p*)
otherwise f endcase))

```
; The following lemma tells us that if the current instruction is
; not one of the fall-off barriers then it is possible to increment
; the pc.
```

THEOREM: proper-p-statep-implies-add1-p-pcp
((car (p-current-instruction (*p*)) = pack (*xxx*))
$\land$ proper-p-statep1 (*p*)
$\land$ (cdr (untag (p-pc (*p*)))
$<$ length (program-body (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*)))))
$\land$ (car (p-current-instruction (*p*))
$\notin$ `'(ret jump jump-case popj)`))
$\rightarrow$ (cdr (untag (p-pc (*p*)))
$<$ (length (program-body (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*)))) $-$ 1))

```
; We also know that the r-current-instruction of the mapped down state
; is the p-current-instruction.
```

THEOREM: r-current-instruction-p->r
r-current-instruction (p->r (*p*)) = p-current-instruction (*p*)

EVENT: Disable r-current-instruction.

```
; Henceforth, in my lemmas building up to the owc theorem, I will use
; (p-current-instruction p) even on the r side of the lemmas because the
; above theorem gives the equality I would otherwise need.  Furthermore,
; with the above theorem around, (r-current-instruction (p->r p)) would
; be Knuth-Bendixed away anyway.

; The major difficulty of this proof is in controlling the expansion
; of the sys-data-segment and tracking the isomorphism between the
; p and r level stacks -- that is ALL there is to this proof!

; We will keep p->r_sys-data-segment disabled because it generates
; a big list I don't like to see.

; We begin by showing what you get when you assoc for each of
; the 5 components of the sys-data-segment.
```

THEOREM: assoc-cstk-p->r_sys-data-segment
assoc (**'cstk**,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))
= p->r_ctrl-stk (*ctrl-stk*, *max-ctrl-stk-size*)

THEOREM: assoc-tstk-p->r_sys-data-segment
assoc (**'tstk**,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))
= p->r_temp-stk (*temp-stk*, *max-temp-stk-size*)

THEOREM: assoc-full-ctrl-stk-addr-p->r_sys-data-segment
assoc (**'full-ctrl-stk-addr**,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))
= list (**'full-ctrl-stk-addr**, tag (**'sys-addr**, **'(cstk . 0)**))

THEOREM: assoc-full-temp-stk-addr-p->r_sys-data-segment
assoc (**'full-temp-stk-addr**,
p->r_sys-data-segment (*ctrl-stk*,

max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))
= list ('full-temp-stk-addr, tag ('sys-addr, '(tstk . 0)))

THEOREM: assoc-empty-temp-stk-addr-p->r_sys-data-segment
assoc ('empty-temp-stk-addr,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))
= list ('empty-temp-stk-addr,
tag ('sys-addr, cons ('tstk, *max-temp-stk-size*)))

`; Next we establish that those we put into are defined:`

THEOREM: definedp-tstk-p->r_sys-data-segment
definedp ('tstk,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))

THEOREM: definedp-cstk-p->r_sys-data-segment
definedp ('cstk,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))

`; Next we establish the lengths of the mapped down cstk and tstk:`

THEOREM: length-strip-cdrs
length (strip-cdrs (x)) = length (x)

EVENT: Enable length-append.

THEOREM: length-p->r_ctrl-stk1
length (p->r_ctrl-stk1 (*stk*, *max*)) = p-ctrl-stk-size (*stk*)

THEOREM: length-nat-0s
length (nat-0s (n)) = fix (n)

EVENT: Enable lessp-difference=0.

THEOREM: length-cdr-p->r_ctrl-stk
$(\mathit{max\text{-}ctrl\text{-}stk\text{-}size} \not< \text{p-ctrl-stk-size}\,(\mathit{ctrl\text{-}stk}))$
$\rightarrow \quad (\text{length}\,(\text{cdr}\,(\text{p->r\_ctrl-stk}\,(\mathit{ctrl\text{-}stk},\ \mathit{max\text{-}ctrl\text{-}stk\text{-}size})))$
$\quad = \quad (1 + \mathit{max\text{-}ctrl\text{-}stk\text{-}size}))$

THEOREM: length-cdr-p->r_temp-stk
$(\mathit{max\text{-}temp\text{-}stk\text{-}size} \not< \text{length}\,(\mathit{temp\text{-}stk}))$
$\rightarrow \quad (\text{length}\,(\text{cdr}\,(\text{p->r\_temp-stk}\,(\mathit{temp\text{-}stk},\ \mathit{max\text{-}temp\text{-}stk\text{-}size})))$
$\quad = \quad (1 + \mathit{max\text{-}temp\text{-}stk\text{-}size}))$

```
; Now we show what happens when you access and change the mapped down stacks.
; My plan is this.  I will leave p->r_ctrl-stk and p->r_temp-stk disabled.
; I will explain how to get an "arbitrary" element out of both.
; I will explain how to put at an "arbitrary" location in p->r_ctrl-stk --
; by showing that the appropriate put-assoc-put combination on p->r_ctrl-stk
; is equal to a p->r_ctrl-stk of an appropriate put.  But I will not discuss
; how to put into p->r_temp-stk.  The problem has to do with popping the
; temp stack and the fact that we often pop several times and then put.
; It is not possible to say what the corresponding p->r_temp-stk would be
; because of the dead junk at the top.  This problem is dealt with by
; stk-equal and so we will address putting into the p->r_temp-stk when
; we deal with stk-equal.

; We start with accessing the temp-stk, which is the simplest operation.
```

EVENT: Enable unconditional-get-append.

THEOREM: difference-difference-difference
$((i \not< j) \wedge (i \not< k)) \rightarrow (((i - j) - (i - k)) = (k - j))$

```
; The following theorem shows how to get at an "arbitrary"
; address in a mapped down stack, where by "arbitrary" I
; mean any address in the live part.
```

THEOREM: get-cdr-p->r_temp-stk
$((\mathit{max\text{-}temp\text{-}stk\text{-}size} \not< \text{length}\,(\mathit{temp\text{-}stk}))$
$\wedge \quad (i < \mathit{max\text{-}temp\text{-}stk\text{-}size})$
$\wedge \quad (i \not< (\mathit{max\text{-}temp\text{-}stk\text{-}size} - \text{length}\,(\mathit{temp\text{-}stk}))))$
$\rightarrow \quad (\text{get}\,(i,\ \text{cdr}\,(\text{p->r\_temp-stk}\,(\mathit{temp\text{-}stk},\ \mathit{max\text{-}temp\text{-}stk\text{-}size})))$
$\quad = \quad \text{get}\,(i - (\mathit{max\text{-}temp\text{-}stk\text{-}size} - \text{length}\,(\mathit{temp\text{-}stk})),\ \mathit{temp\text{-}stk}))$

```
; Most often we will be getting at the top of the temp stk,
; which is i = (difference max-temp-stk-size (length temp-stk))
```

```
; (just look at p->r_tsp).  In that case, we will need:
```

EVENT: Enable difference-x-x.

```
; When we get at top1 i will be
;    (difference max-temp-stk-size (length (cdr temp-stk)))
; and the (difference max-temp-stk-size (length temp-stk))
; above will become
;    (difference max-temp-stk-size (add1 (length (cdr temp-stk))))
; which will expand to
;    (sub1 (difference max-temp-stk-size (length (cdr temp-stk))))
; by
```

THEOREM: difference-add1{special}
$(x - (1 + y)) = ((x - y) - 1)$

```
; So we will need:
```

THEOREM: difference-x-sub1-x
$(x \not\simeq 0) \rightarrow ((x - (x - 1)) = 1)$

```
; Similarly for top2:
```

THEOREM: difference-x-sub2-x
$((x \not\simeq 0) \wedge ((x - 1) \not\simeq 0)) \rightarrow ((x - ((x - 1) - 1)) = 2)$

```
; We then eliminate the gets in these cases because the p machine
; uses top1 and top2, not get-1 and get-2:
```

EVENT: Enable get-0.

THEOREM: get-1
$\text{get}(1, \mathit{lst}) = \text{cadr}(\mathit{lst})$

THEOREM: get-2
$\text{get}(2, \mathit{lst}) = \text{caddr}(\mathit{lst})$

```
; Now I turn to the control stack.  Again, we need to be able to
; get and put into it.  However, we only entertain those operations
; on local variable positions of the top frame.
```

```
; We begin with get

; The key idea, of course, is that getting at a specified position
; in the strip-cdrs of an alist is the same as assoc.
```

THEOREM: get-find-position-of-var
(proper-p-alistp (*alist*, *p*) $\wedge$ (*var* $\in$ strip-cars (*alist*)))
$\rightarrow$ (get (find-position-of-var (*var*, strip-cars (*alist*)),
append (strip-cdrs (*alist*), *stk*))
= cdr (assoc (*var*, *alist*)))

```
; Now we have to work a little to establish the correspondence between
; the list we find the position in and the list we assoc into.
```

THEOREM: member-implies-lessp-find-position-of-var
($x \in lst$) $\rightarrow$ (find-position-of-var (x, *lst*) $<$ length (*lst*))

THEOREM: equal-strip-cars-implies-equal-lengths
(strip-cars (x) $=$ y) $\rightarrow$ (length (x) $=$ length (y))

EVENT: Enable length-strip-cars.

THEOREM: length-bindings
proper-p-statep1 (p)
$\rightarrow$ (length (bindings (car (p-ctrl-stk (p))))
= (length (formal-vars (assoc (car (untag (p-pc (p))),
p-prog-segment (p))))
+ length (temp-var-dcls (assoc (car (untag (p-pc (p))),
p-prog-segment (p))))))

THEOREM: difference-plus{special}
$((x + y) - x) =$ fix (y)

THEOREM: p->r_ctrl-stk1-opener
proper-p-statep1 (p)
$\rightarrow$ (p->r_ctrl-stk1 (p-ctrl-stk (p), p-max-ctrl-stk-size (p))
= append (p->r_p-frame (top (p-ctrl-stk (p)),
pop (p-ctrl-stk (p)),
p-max-ctrl-stk-size (p)),
p->r_ctrl-stk1 (pop (p-ctrl-stk (p)),
p-max-ctrl-stk-size (p))))

```
; And here is the main lemma about getting at a local variable
; slot in the mapped down control stack:
```

THEOREM: get-cdr-p->r_ctrl-stk-1

$$
\begin{array}{l}
(\text{proper-p-statep1}\,(p) \\
\;\wedge\;\; (\text{p-max-ctrl-stk-size}\,(p) \not< \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p))) \\
\;\wedge\;\; (\mathit{var} \in \text{append}\,(\text{formal-vars}\,(\text{assoc}\,(\text{car}\,(\text{untag}\,(\text{p-pc}\,(p))), \\
\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))), \\
\qquad\qquad\quad \text{strip-cars}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{car}\,(\text{untag}\,(\text{p-pc}\,(p))), \\
\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))))) \\
\to\;\; (\text{get}\,((\text{p-max-ctrl-stk-size}\,(p) - \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p))) \\
\qquad\quad + \;\; \text{find-position-of-var}\,(\mathit{var}, \\
\qquad\qquad\qquad \text{append}\,(\text{formal-vars}\,(\text{assoc}\,(\text{car}\,(\text{untag}\,(\text{p-pc}\,(p))), \\
\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))), \\
\qquad\qquad\qquad\quad \text{strip-cars}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{car}\,(\text{untag}\,(\text{p-pc}\,(p))), \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))))), \\
\qquad\quad \text{cdr}\,(\text{p->r\_ctrl-stk}\,(\text{p-ctrl-stk}\,(p),\ \text{p-max-ctrl-stk-size}\,(p)))) \\
\qquad = \;\; \text{cdr}\,(\text{assoc}\,(\mathit{var},\ \text{bindings}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))))
\end{array}
$$

```
; Now we turn to put.
```

THEOREM: put-find-position-of-var

$$
\begin{array}{l}
(\text{proper-p-alistp}\,(\mathit{alist},\ p) \wedge (\mathit{var} \in \text{strip-cars}\,(\mathit{alist}))) \\
\to\;\; (\text{put}\,(\mathit{val}, \\
\qquad\quad \text{find-position-of-var}\,(\mathit{var},\ \text{strip-cars}\,(\mathit{alist})), \\
\qquad\quad \text{append}\,(\text{strip-cdrs}\,(\mathit{alist}),\ \mathit{stk})) \\
\qquad = \;\; \text{append}\,(\text{strip-cdrs}\,(\text{put-assoc}\,(\mathit{val},\ \mathit{var},\ \mathit{alist})),\ \mathit{stk}))
\end{array}
$$

EVENT: Enable equal-length-0.

THEOREM: put-plus-append

$$
\begin{array}{l}
(i = \text{length}\,(a)) \\
\to\;\; (\text{put}\,(\mathit{val},\ i + j,\ \text{append}\,(a,\ b)) = \text{append}\,(a,\ \text{put}\,(\mathit{val},\ j,\ b)))
\end{array}
$$

THEOREM: length-put-assoc

$$\text{length}\,(\text{put-assoc}\,(\mathit{val},\ \mathit{name},\ \mathit{alist})) = \text{length}\,(\mathit{alist})$$

THEOREM: p-ctrl-stk-size-cons-p-frame-put-assoc

$$
\begin{array}{l}
\text{listp}\,(\mathit{stk}) \\
\to\;\; (\text{p-ctrl-stk-size}\,(\text{cons}\,(\text{p-frame}\,(\text{put-assoc}\,(\mathit{val},\ \mathit{var},\ \text{bindings}\,(\text{car}\,(\mathit{stk}))), \\
\qquad\qquad\qquad\qquad \text{ret-pc}\,(\text{car}\,(\mathit{stk}))), \\
\qquad\qquad\qquad \text{cdr}\,(\mathit{stk}))) \\
\qquad = \;\; \text{p-ctrl-stk-size}\,(\mathit{stk}))
\end{array}
$$

EVENT: Enable associativity-of-append.

THEOREM: put-cdr-p->r_ctrl-stk-1

```
(proper-p-statep1 (p)
 ∧   (p-max-ctrl-stk-size (p) ≮ p-ctrl-stk-size (p-ctrl-stk (p)))
 ∧   (var ∈ append (formal-vars (assoc (car (untag (p-pc (p))),
                                         p-prog-segment (p))),
                    strip-cars (temp-var-dcls (assoc (car (untag (p-pc (p))),
                                                      p-prog-segment (p)))))))
→   (put (val,
          (p-max-ctrl-stk-size (p) − p-ctrl-stk-size (p-ctrl-stk (p)))
          +   find-position-of-var (var,
                                    append (formal-vars (assoc (car (untag (p-pc (p))),
                                                                p-prog-segment (p))),
                                            strip-cars (temp-var-dcls (assoc (car (untag (p-pc (p))),
                                                                             p-prog-segment (p)))))),
          cdr (p->r_ctrl-stk (p-ctrl-stk (p), p-max-ctrl-stk-size (p))))
     =   cdr (p->r_ctrl-stk (cons (p-frame (put-assoc (val,
                                                        var,
                                                        bindings (car (p-ctrl-stk (p)))),
                                            ret-pc (car (p-ctrl-stk (p)))),
                                   cdr (p-ctrl-stk (p))),
                             p-max-ctrl-stk-size (p))))
```

```
; Recall that we are keeping p->r_sys-data-segment closed.  So we have
; to package up the put-assoc that the r machine does on its sys-data-segment:
```

THEOREM: car-p->r_ctrl-stk

car (p->r_ctrl-stk (*temp-stk*, *max*)) = `'cstk`

THEOREM: car-p->r_temp-stk

car (p->r_temp-stk (*temp-stk*, *max*)) = `'tstk`

THEOREM: put-assoc-cstk-p->r_sys-data-segment-local

```
put-assoc (cdr (p->r_ctrl-stk (ctrl-stk1, max-ctrl-stk-size1)),
           'cstk,
           p->r_sys-data-segment (ctrl-stk2,
                                  max-ctrl-stk-size2,
                                  temp-stk2,
                                  max-temp-stk-size2))
=   p->r_sys-data-segment (ctrl-stk1,
                           max-ctrl-stk-size1,
                           temp-stk2,
                           max-temp-stk-size2)
```

```
; Note:  The suffix "local" on the name above is present because we
; have a more elaborate version of this same idea later for call.

; Finally, we shut down the mappers:
```

EVENT: Disable p->r_ctrl-stk.

EVENT: Disable p->r_temp-stk.

EVENT: Disable p->r_sys-data-segment.

```
; Now we address the stk-equal problem.  Since we have p->r_sys-data-segment
; disabled, we need theorems that tell us when one such application is
; stk-equal to another or to a put-assoc that changes on of the components.

; The following function is the standard induction hint for theorems about
; stk-equal that use a non-variable term for the stack pointer.  E.g.,
; if (stk-equal sp ...) could be proved by pop-to-empty-stkp induction
; on sp, then (stk-equal (tag 'sys-addr (cons 'cstk n)) ...) should be
; proved by (stk-equal-hint-1 n ...) induction on n.
```

DEFINITION:
stk-equal-hint-1 (n, max)
= **if** ($max - n$) $\simeq$ 0 **then t**
else stk-equal-hint-1 ($1 + n$, max) **endif**

```
; Because of the restrictions on the form of terms used as induction
; hints, we need a second form of this hint function.  (The problem has
; to do with the prohibition of non-variable terms in induction hint
; args, even for args in constant slots.)
```

EVENT: Enable difference-x-0.

DEFINITION:
stk-equal-hint-2 (n, $ctrl\text{-}stk$, $max\text{-}ctrl\text{-}stk\text{-}size$, $temp\text{-}stk$, $max\text{-}temp\text{-}stk\text{-}size$)
= **if** ((length (cdr (assoc ('cstk,
p->r_sys-data-segment ($ctrl\text{-}stk$,
$max\text{-}ctrl\text{-}stk\text{-}size$,
$temp\text{-}stk$,
$max\text{-}temp\text{-}stk\text{-}size$))))

$- \quad n) - 1) \simeq 0$ **then** **t**
else stk-equal-hint-2 (n + 1,
ctrl-stk,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size) **endif**

```
; By the way, normally in the theorems we have to prove formulas of the
; form (stk-equal sp segment1 segment2) have the following ancestry and:
; form.  The sp is some semi-explicit pointer like
; (tag 'sys-addr (cons 'cstk n)) as produced by p->r_csp.  Segment1 is
; a call of p->r_sys-data-segment on the result of stepping the p machine.
; Segment2 is the result of stepping the r machine on the original
; p->r_sys-data-segment.

; If the stack pointer of interest is in the cstk area, then it doesn't
; matter what happens to the tstk area at the r level.  The following theorem
; formalizes that for the case where the r-level does nothing, e.g., no-op.
```

THEOREM: stk-equal-reflexive-cstk
stk-equal (tag (`'sys-addr`, cons (`'cstk`, n)),
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk1,
max-temp-stk-size1),
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk2,
max-temp-stk-size2))

```
; The next theorem formalizes it (by reducing it to the no-op case) when the
; r-level just hits the tstk:
```

THEOREM: assoc-put-assoc-1
($name1 \neq name2$)
$\rightarrow$ (assoc (*name1*, put-assoc (*val*, *name2*, *alist*)) = assoc (*name1*, *alist*))

DEFINITION:
stk-equal-hint-3 (n, *segment*)
= **if** ((length (cdr (assoc (`'cstk`, *segment*))) $- n) - 1) \simeq 0$ **then** **t**
else stk-equal-hint-3 (n + 1, *segment*) **endif**

THEOREM: stk-equal-cstk-put-assoc-tstk

stk-equal (tag ('**sys-addr**, cons ('**cstk**, *n*)),
segment1,
put-assoc (*area*, '**tstk**, *segment2*))
= stk-equal (tag ('**sys-addr**, cons ('**cstk**, *n*)), *segment1*, *segment2*)

```
; We now do the same thing for the case where the r-level machine doesn't
; change the cstk.  This is very common, of course.
```

DEFINITION:
stk-equal-hint-4 (*n*, *ctrl-stk*, *max-ctrl-stk-size*, *temp-stk*, *max-temp-stk-size*)
= **if** ((length (cdr (assoc ('**tstk**,
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size))))
− *n*) − 1) $\simeq$ 0 **then t**
else stk-equal-hint-4 (*n* + 1,
ctrl-stk,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size) **endif**

THEOREM: stk-equal-reflexive-tstk
stk-equal (tag ('**sys-addr**, cons ('**tstk**, *n*)),
p->r_sys-data-segment (*ctrl-stk1*,
max-ctrl-stk-size1,
temp-stk,
max-temp-stk-size),
p->r_sys-data-segment (*ctrl-stk2*,
max-ctrl-stk-size2,
temp-stk,
max-temp-stk-size))

```
; This completely handles the stk-equal work for the cstk (except on
; CALL and RET, which we be done specially when they arise).  Now I
; turn to tstk.

; Consider the pattern of pushes and pops used by our instructions.
; For each we need to know that the high level operation corresponds to
; the low level one.  The patterns we use are

;   pattern     typical instruction

;  push            push-global
```

```
;  pop             pop-global
;  pop 2           deposit
;  pop 1 push 1    add1-nat
;  pop 2 push 1    add-addr
;  pop 3 push 2    add-nat-with-carry
;  pop 1 push 2    mult2-nat-with-carry-out

; Our approach to stk-equal questions of the form
; (stk-equal (tag 'sys-addr (cons 'tstk n))
;            (p->r_sys-data-segment ... p-level-temp-stk ...)
;            r-level-temp-stk)
; will be to reduce them to (equal p-level-temp-stk &), where & is the
; "live part" of the r-level-temp-stk.  We don't define the function
; live-part but rather just use the combination of cdr-n and all-but-last
; defined below.
```

THEOREM: get-equal-car-cdr-n
$\text{get}(n,\ \textit{lst}) = \text{car}(\text{cdr-n}(n,\ \textit{lst}))$

THEOREM: cdr-n-nil
$((n \not< \text{length}(\textit{lst})) \land \text{properp}(\textit{lst}))$
$\rightarrow \quad (\text{cdr-n}(n,\ \textit{lst})$
$\qquad = \quad \textbf{if } \text{fix}(n) = \text{length}(\textit{lst}) \textbf{ then nil}$
$\qquad\qquad \textbf{else } 0 \textbf{ endif})$

THEOREM: length-all-but-last
$\text{length}(\text{all-but-last}(x)) = (\text{length}(x) - 1)$

THEOREM: properp-all-but-last
$\text{properp}(\text{all-but-last}(x))$

THEOREM: stk-equal-is-equal-cdr-n-all-but-last-lemma1
$(\text{cdr-n}(n,\ x) = \text{cdr-n}(n,\ y))$
$\rightarrow \quad ((\text{cdr-n}(n,\ \text{cdr}(x)) = \text{cdr-n}(n,\ \text{cdr}(y))) = \textbf{t})$

THEOREM: stk-equal-is-equal-cdr-n-all-but-last-lemma2
$((n < (\text{length}(\textit{lst1}) - 1))$
$\land \quad (n \in \mathbf{N})$
$\land \quad (\text{cdr-n}(n,\ \text{cdr}(\text{all-but-last}(\textit{lst1})))$
$\qquad = \quad \text{cdr-n}(n,\ \text{cdr}(\text{all-but-last}(\textit{lst2}))))$
$\land \quad (\text{length}(\textit{lst1}) = \text{length}(\textit{lst2}))$
$\land \quad (\text{cdr-n}(n,\ \text{all-but-last}(\textit{lst1})) \neq \text{cdr-n}(n,\ \text{all-but-last}(\textit{lst2}))))$
$\rightarrow \quad (\text{car}(\text{cdr-n}(n,\ \textit{lst1})) \neq \text{car}(\text{cdr-n}(n,\ \textit{lst2})))$

THEOREM: stk-equal-is-equal-cdr-n-all-but-last-lemma3

$$((n < (\text{length}(\mathit{lst1}) - 1)) \land (n \in \mathbf{N}) \land (\text{length}(\mathit{lst1}) = \text{length}(\mathit{lst2})) \land (\text{cdr-n}(n, \text{all-but-last}(\mathit{lst1})) = \text{cdr-n}(n, \text{all-but-last}(\mathit{lst2})))) \rightarrow ((\text{car}(\text{cdr-n}(n, \mathit{lst1})) = \text{car}(\text{cdr-n}(n, \mathit{lst2}))) = \mathbf{t})$$

THEOREM: equal-sub1-difference-0

$$(((x - y) - 1) = 0) = (y \not< (x - 1))$$

```
; This is the key theorem.  We do not, in general, want to eliminate
; stk-equal, so it is proved with lemma class nil.  We'll prove a
; certain instance of it later for use as a rewrite rule.

; Note that as proved below it applies as well to the cstk as to tstk.
```

THEOREM: stk-equal-is-equal-cdr-n-all-but-last

$$((\text{cdr}(\text{untag}(\mathit{sp})) \in \mathbf{N}) \land (\text{length}(\text{cdr}(\text{assoc}(\text{car}(\text{untag}(\mathit{sp})), \mathit{segment1}))) = \text{length}(\text{cdr}(\text{assoc}(\text{car}(\text{untag}(\mathit{sp})), \mathit{segment2}))))) \rightarrow (\text{stk-equal}(\mathit{sp}, \mathit{segment1}, \mathit{segment2}) = (\text{cdr-n}(\text{cdr}(\text{untag}(\mathit{sp})), \text{all-but-last}(\text{cdr}(\text{assoc}(\text{car}(\text{untag}(\mathit{sp})), \mathit{segment1})))) = \text{cdr-n}(\text{cdr}(\text{untag}(\mathit{sp})), \text{all-but-last}(\text{cdr}(\text{assoc}(\text{car}(\text{untag}(\mathit{sp})), \mathit{segment2}))))))$$

EVENT: Disable equal-sub1-difference-0.

EVENT: Disable get-equal-car-cdr-n.

EVENT: Disable cdr-n-nil.

EVENT: Disable stk-equal-is-equal-cdr-n-all-but-last-lemma1.

EVENT: Disable stk-equal-is-equal-cdr-n-all-but-last-lemma2.

EVENT: Disable stk-equal-is-equal-cdr-n-all-but-last-lemma3.

THEOREM: assoc-put-assoc-2

$$\text{definedp}(\mathit{name}, \mathit{alist}) \rightarrow (\text{assoc}(\mathit{name}, \text{put-assoc}(\mathit{val}, \mathit{name}, \mathit{alist})) = \text{cons}(\mathit{name}, \mathit{val}))$$

THEOREM: all-but-last-append
listp $(b) \rightarrow$ (all-but-last (append $(a,\ b)$) = append (a, all-but-last (b)))

THEOREM: length-cdr-n
length (cdr-n $(n,\ lst)$) = (length (lst) $-$ n)

EVENT: Enable append-right-id.

THEOREM: cdr-n-append{special}
(n = length (a)) $\rightarrow$ (cdr-n (n, append $(a,\ b)$) = b)

```
; Here, finally, is the rewrite rule that reduce stk-equal
; questions on tstk to consideration of the "live part" of the
; r-level stack:
```

THEOREM: stk-equal-tstk-p->r_sys-data-segment
(properp (*temp-stk*)
 $\wedge$ (n = (*max-temp-stk-size* $-$ length (*temp-stk*)))
 $\wedge$ (*max-temp-stk-size* $\not<$ length (*temp-stk*))
 $\wedge$ ((1 + *max-temp-stk-size*) = length (cdr (assoc (`'tstk`, *segment*)))))
$\rightarrow$ (stk-equal (tag (`'sys-addr`, cons (`'tstk`, n)),
 p->r_sys-data-segment (*ctrl-stk*,
 max-ctrl-stk-size,
 temp-stk,
 max-temp-stk-size),
 segment)
 = (*temp-stk* = cdr-n (*max-temp-stk-size* $-$ length (*temp-stk*),
 all-but-last (cdr (assoc (`'tstk`, *segment*))))))

EVENT: Disable stk-equal.

EVENT: Disable empty-stkp.

```
; To use the theorem above we need to relieve the hypotheses.

; The typical form of instantiations for temp-stk in the lemma
; above is either (p-temp-stk p) or else some collection of
; conses and cdrs of it.  We need to know that it is proper.
; The first theorem below establishes that the initial temp
; stack is proper.
```

THEOREM: proper-p-temp-stkp-implies-properp
proper-p-temp-stkp $(stk,\ p) \rightarrow$ properp (stk)

THEOREM: properp-p-temp-stkp
$\text{proper-p-statep1}(p) \rightarrow \text{properp}(\text{p-temp-stk}(p))$

```
; And this one establishes that the cdr of it is proper.  This is
; what we'll need when we pop.
```

THEOREM: properp-implies-properp-cdr
$(\text{listp}(x) \land \text{properp}(x)) \rightarrow \text{properp}(\text{cdr}(x))$

```
; When we push we'll cons onto the stack and that cons will be eaten
; by properp itself.

; The only other problematic hypothesis for the lemma above is that
; concerning the length of the new tstk area.  The typical instantiation
; of segment above will a put-assoc.  When we assoc for 'tstk in it we
; will get a (put val n previous-area), where n is some legal address in
; the previous tstk area.  We need to know the length of the resulting
; area.  It is, of course, unchanged:
```

EVENT: Enable length-put.

```
; That handles most of the problems I can anticipate on relieving the
; hyps.  Now we consider simplifying the rhs of the lemma under the
; typical instantiations of segment.  We are interested in cdr-n and
; all-but-last lemmas on the result of (assoc 'tstk segment).  As noted,
; the assoc will turn into a put under most instantiations of segment.

; In the case that it is a put, we will use the following lemma to
; drive the all-but-last through the put.  If there are multiple puts,
; this lemma will drive it all the way down.
```

THEOREM: all-but-last-put
$(n < (\text{length}(lst) - 1))$
$\rightarrow \quad (\text{all-but-last}(\text{put}(val,\ n,\ lst)) = \text{put}(val,\ n,\ \text{all-but-last}(lst)))$

```
; Eventually it will come to rest against the original tsk and we'll
; open it up:
```

THEOREM: all-but-last-cdr-p->r_temp-stk
$\text{properp}(temp\text{-}stk)$
$\rightarrow \quad (\text{all-but-last}(\text{cdr}(\text{p->r\_temp-stk}(temp\text{-}stk,\ max\text{-}temp\text{-}stk\text{-}size)))$
$\quad = \quad \text{append}(\text{nat-0s}(max\text{-}temp\text{-}stk\text{-}size - \text{length}(temp\text{-}stk)),$
$\quad\quad temp\text{-}stk))$

```
; to remove the (NAT 0) at the end, kicking out an append
; which will let the PUTs start to resolve themselves.

; Now if the put is an simple push the following lemma will take care
; of it:
```

THEOREM: nat-0s-opener-at-right-end
nat-0s (n)
$=$ **if** $n \simeq 0$ **then nil**
else append (nat-0s $(n - 1)$, list (`'(nat 0)`)) **endif**

THEOREM: nat-0s-opener-at-right-end-on-add1
nat-0s $(1 + n)$ = append (nat-0s (n), list (`'(nat 0)`))

THEOREM: put-append
$((n = \text{length}(a)) \land \text{listp}(b))$
$\rightarrow$ $(\text{put}(\mathit{val}, n, \text{append}(a, b)) = \text{append}(a, \text{cons}(\mathit{val}, \text{cdr}(b))))$

```
; Whew!  That was tough wasn't it?  Anyway, here is the key fact
; about a put that implements a push.
```

THEOREM: put-append-1{special}
$((n \not\simeq 0) \land (k = (n - 1)))$
$\rightarrow$ $(\text{put}(\mathit{val}, k, \text{append}(\text{nat-0s}(n), \mathit{lst})) = \text{append}(\text{nat-0s}(k), \text{cons}(\mathit{val}, \mathit{lst})))$

```
; The following is needed to simplify the cdr-n in the case where
; we have done a pop.
```

THEOREM: cdr-n-cdr
cdr-n $(n, \text{cdr}(\mathit{lst}))$ = cdr (cdr-n (n, lst))

```
; There are probably more facts about cdr-n than I have proved.  I will
; prove the rest as needed to get the owc proofs through.

; Now we begin considering the same-signature requirements of r-equal.
```

THEOREM: same-signature-reflexive{special}
$((\mathit{max\text{-}ctrl\text{-}stk\text{-}size} \not< \text{p-ctrl-stk-size}(\mathit{ctrl\text{-}stk1}))$
$\land$ $(\mathit{max\text{-}ctrl\text{-}stk\text{-}size} \not< \text{p-ctrl-stk-size}(\mathit{ctrl\text{-}stk2}))$
$\land$ $(\mathit{max\text{-}temp\text{-}stk\text{-}size} \not< \text{length}(\mathit{temp\text{-}stk1}))$
$\land$ $(\mathit{max\text{-}temp\text{-}stk\text{-}size} \not< \text{length}(\mathit{temp\text{-}stk2})))$
$\rightarrow$ same-signature (p->r_sys-data-segment $(\mathit{ctrl\text{-}stk1},$

$$\begin{array}{ll} & \textit{max-ctrl-stk-size}, \\ & \textit{temp-stk1}, \\ & \textit{max-temp-stk-size}), \\ \text{p->r\_sys-data-segment}\,(& \textit{ctrl-stk2}, \\ & \textit{max-ctrl-stk-size}, \\ & \textit{temp-stk2}, \\ & \textit{max-temp-stk-size})) \end{array}$$

THEOREM: same-signature-put-assoc-1
(definedp (*name*, *segment1*)
$\wedge$ litatom (*name*)
$\wedge$ (length (*val*) = length (definiens (*name*, *segment1*))))
$\rightarrow$ (same-signature (put-assoc (*val*, *name*, *segment1*), *segment2*)
= same-signature (*segment1*, *segment2*))

THEOREM: same-signature-put-assoc-2
(definedp (*name*, *segment2*)
$\wedge$ litatom (*name*)
$\wedge$ (length (*val*) = length (definiens (*name*, *segment2*))))
$\rightarrow$ (same-signature (*segment1*, put-assoc (*val*, *name*, *segment2*))
= same-signature (*segment1*, *segment2*))

EVENT: Disable same-signature.

```
; Now I want to disable get and put so that they are only manipulated by
; the lemmas we have proved.
```

EVENT: Disable get.

EVENT: Disable put.

```
; I have decided to do the owc proofs at this level a little differently.
; First I will prove that
; (p-xxx-okp p) -> (requal (p->r (p-xxx-step p)) (r-xxx-step (p->r p)))
; Then I will prove
; (p-xxx-okp p) -> (r-xxx-okp (p->r p)).

; Then I will put them together to get
; (p->xxx-okp p) -> (r-equal (p->r (p-xxx-step p)) (r-step (p->r p))).

; We now begin the owc proofs.  I have selected several theorems for my
; initial test section and will work on the lemma development to make the
```

```
; proofs of these work.  Then I'll turn it loose on all the others.
; CALL and RET are left for last.  I have made sure to include one theorem
; for each pattern of tstk use mentioned above, plus local variable
; pushing and popping.  I begin, as always, with no-op, the simplest.
```

THEOREM: no-op-one-way-correspondence-p-r
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'no-op)
∧ proper-p-statep (p)
∧ p-no-op-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-no-op-step (p-current-instruction (p), p)),
r-no-op-step (p-current-instruction (p), p->r (p)))

THEOREM: push-global-one-way-correspondence-p-r
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'push-global)
∧ proper-p-statep (p)
∧ p-push-global-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-push-global-step (p-current-instruction (p), p)),
r-push-global-step (p-current-instruction (p), p->r (p)))

THEOREM: lessp-length-cdr
length (x) $\not<$ length (cdr (x))

```
; To relieve the 2nd hypothesis of get-cdr-p->r_temp-stk, the one
; that says that i is less than max-temp-stk-size, we need to know,
; in the case of a simple pop, that:
```

THEOREM: lessp-difference-x-y-x
(($max \not<$ length ($temp\text{-}stk$)) ∧ listp ($temp\text{-}stk$))
→ ((max − length ($temp\text{-}stk$)) $< max$)

```
; Note:  The rule above uses max as a free-var, accidentally.  I
; don't know if this will prevent its use later...
```

THEOREM: pop-global-one-way-correspondence-p-r
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'pop-global)
∧ proper-p-statep (p)
∧ p-pop-global-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-pop-global-step (p-current-instruction (p), p)),
r-pop-global-step (p-current-instruction (p), p->r (p)))

THEOREM: push-local-one-way-correspondence-p-r
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'push-local})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-push-local-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to\quad \text{r-equal}\,(\text{p->r}\,(\text{p-push-local-step}\,(\text{p-current-instruction}\,(p),\ p)),$
$\qquad\qquad \text{r-push-local-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p)))$

THEOREM: pop-local-one-way-correspondence-p-r
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'pop-local})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-pop-local-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to\quad \text{r-equal}\,(\text{p->r}\,(\text{p-pop-local-step}\,(\text{p-current-instruction}\,(p),\ p)),$
$\qquad\qquad \text{r-pop-local-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p)))$

THEOREM: difference-add1-x-x
$((1 + x) - x) = \texttt{1}$

THEOREM: cdr-n-append-pop2
$((n \not\simeq \texttt{0}) \land ((n - 1) \not\simeq \texttt{0}))$
$\to\quad (\text{cdr-n}\,(n,\ \text{append}\,(\text{nat-0s}\,((n - 1) - 1),\ stk)) = \text{cddr}\,(stk))$

THEOREM: deposit-one-way-correspondence-p-r
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'deposit})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-deposit-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to\quad \text{r-equal}\,(\text{p->r}\,(\text{p-deposit-step}\,(\text{p-current-instruction}\,(p),\ p)),$
$\qquad\qquad \text{r-deposit-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p)))$

```
; The following lemmas are needed because lessp opens up when it
; on (lessp (p-max-temp-stk-size p) (length (p-temp-stk p)))
; because length opens up.
```

THEOREM: not-equal-p-max-temp-stk-size-0
$(\text{listp}\,(\text{p-temp-stk}\,(p)) \land (\text{p-max-temp-stk-size}\,(p) \not< \text{length}\,(\text{p-temp-stk}\,(p))))$
$\to\quad (\text{p-max-temp-stk-size}\,(p) \neq \texttt{0})$

THEOREM: not-lessp-sub1-p-max-temp-stk-size
$(\text{p-max-temp-stk-size}\,(p) \not< \text{length}\,(\text{p-temp-stk}\,(p)))$
$\to\quad ((\text{p-max-temp-stk-size}\,(p) - 1) \not< \text{length}\,(\text{cdr}\,(\text{p-temp-stk}\,(p))))$

THEOREM: add1-nat-one-way-correspondence-p-r

$$((\text{p-psw}(p) = \texttt{'run})$$
$$\land\ (\text{car}(\text{p-current-instruction}(p)) = \texttt{'add1-nat})$$
$$\land\ \text{proper-p-statep}(p)$$
$$\land\ \text{p-add1-nat-okp}(\text{p-current-instruction}(p),\ p))$$
$$\rightarrow\ \text{r-equal}(\text{p->r}(\text{p-add1-nat-step}(\text{p-current-instruction}(p),\ p)),$$
$$\text{r-add1-nat-step}(\text{p-current-instruction}(p),\ \text{p->r}(p)))$$

```
; The following lemma explains a push after two pops.
```

THEOREM: put-append-2{special}
$$((n \not\simeq \mathtt{0}) \land \text{listp}(\text{cdr}(stk)))$$
$$\rightarrow\ (\text{put}(val,\ n,\ \text{append}(\text{nat-0s}(n-1),\ stk))$$
$$=\ \text{append}(\text{nat-0s}(n-1),\ \text{cons}(\text{car}(stk),\ \text{cons}(val,\ \text{cddr}(stk)))))$$

THEOREM: cdr-n-append-2
$$(n \not\simeq \mathtt{0}) \rightarrow (\text{cdr-n}(n,\ \text{append}(\text{nat-0s}(n-1),\ stk)) = \text{cdr}(stk))$$

THEOREM: add-addr-one-way-correspondence-p-r
$$((\text{p-psw}(p) = \texttt{'run})$$
$$\land\ (\text{car}(\text{p-current-instruction}(p)) = \texttt{'add-addr})$$
$$\land\ \text{proper-p-statep}(p)$$
$$\land\ \text{p-add-addr-okp}(\text{p-current-instruction}(p),\ p))$$
$$\rightarrow\ \text{r-equal}(\text{p->r}(\text{p-add-addr-step}(\text{p-current-instruction}(p),\ p)),$$
$$\text{r-add-addr-step}(\text{p-current-instruction}(p),\ \text{p->r}(p)))$$

```
; Because add-nat-with-carry writes twice to the tstk, we need:
```

EVENT: Enable put-assoc-put-assoc.

```
; Both the p and the r machine use fix-small-natural and small-naturalp
; to operate on nats, so I will
```

EVENT: Disable fix-small-natural.

EVENT: Disable small-naturalp.

EVENT: Disable bool-to-nat.

EVENT: Disable bool.

```
; The following lemma explains a push after three pops.
```

THEOREM: put-append-nat-0s-sub1-sub1
$(((n - 1) \not\simeq 0) \land \text{listp}(\text{cddr}(stk)))$
$\rightarrow \quad (\text{put}(val,\ n,\ \text{append}(\text{nat-0s}((n - 1) - 1),\ stk))$
$\quad = \quad \text{append}(\text{nat-0s}((n - 1) - 1),$
$\qquad \text{cons}(\text{car}(stk),\ \text{cons}(\text{cadr}(stk),\ \text{cons}(val,\ \text{cdddr}(stk))))))$

THEOREM: add-nat-with-carry-one-way-correspondence-p-r
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'add-nat-with-carry})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-add-nat-with-carry-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{r-equal}(\text{p->r}(\text{p-add-nat-with-carry-step}(\text{p-current-instruction}(p),\ p)),$
$\qquad \text{r-add-nat-with-carry-step}(\text{p-current-instruction}(p),\ \text{p->r}(p)))$

```
; As above, length opens up on (cons val (cdr temp-stk)), and lessp
; opens on that and we have to accomodate them.
```

THEOREM: not-equal-sub1-p-max-temp-stk-size-0
$(\text{listp}(\text{p-temp-stk}(p)) \land (\text{length}(\text{p-temp-stk}(p)) < \text{p-max-temp-stk-size}(p)))$
$\rightarrow \quad ((\text{p-max-temp-stk-size}(p) - 1) \neq 0)$

THEOREM: not-lessp-sub1-sub1-p-max-temp-stk-size
$(\text{length}(\text{p-temp-stk}(p)) < \text{p-max-temp-stk-size}(p))$
$\rightarrow \quad ((((\text{p-max-temp-stk-size}(p) - 1) - 1) < \text{length}(\text{cdr}(\text{p-temp-stk}(p))))$
$\quad = \quad \mathbf{f})$

THEOREM: mult2-nat-with-carry-out-one-way-correspondence-p-r
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'mult2-nat-with-carry-out})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-mult2-nat-with-carry-out-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{r-equal}(\text{p->r}(\text{p-mult2-nat-with-carry-out-step}(\text{p-current-instruction}(p),$
$\qquad p)),$
$\qquad \text{r-mult2-nat-with-carry-out-step}(\text{p-current-instruction}(p),$
$\qquad \text{p->r}(p)))$

THEOREM: difference-sub1
$((x - 1) - y) = ((x - y) - 1)$

THEOREM: fetch-temp-stk-one-way-correspondence-p-r

((p-psw (p) = '`run`)
 ∧ (car (p-current-instruction (p)) = '`fetch-temp-stk`)
 ∧ proper-p-statep (p)
 ∧ p-fetch-temp-stk-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-fetch-temp-stk-step (p-current-instruction (p), p)),
  r-fetch-temp-stk-step (p-current-instruction (p), p->r (p)))

THEOREM: properp-put
properp (*lst*) → properp (put (*val*, n, *lst*))

THEOREM: cdr-n-put
(($n \not< k$) ∧ ($n <$ length (*lst*)))
→ (cdr-n (k, put (*val*, n, *lst*)) = put (*val*, $n - k$, cdr-n (k, *lst*)))

THEOREM: deposit-temp-stk-one-way-correspondence-p-r
((p-psw (p) = '`run`)
 ∧ (car (p-current-instruction (p)) = '`deposit-temp-stk`)
 ∧ proper-p-statep (p)
 ∧ p-deposit-temp-stk-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-deposit-temp-stk-step (p-current-instruction (p), p)),
  r-deposit-temp-stk-step (p-current-instruction (p), p->r (p)))

EVENT: Disable cdr-n-put.

EVENT: Disable difference-sub1.

```
; The pop* instruction shows a new twist:  popping many things off tstk.
; The p machine uses popn but we have a lot of machinery sitting around
; for cdr-n, which is an equivalent function:
```

THEOREM: popn-is-cdr-n
popn (n, *lst*) = cdr-n (n, *lst*)

```
; We now turn to stk-equal on the tstk. Recall
; stk-equal-tstk-p->r_sys-data-segment:  it has two variables of importance,
; temp-stk, the p-level version of the stack, and segment, the r-level
; sys-data-segment. In the call case, temp-stk will be a popn (now cdr-n)
; and segment is untouched.  If we can relieve the hyps of the lemma,
; we will have to prove that the popn is equal to a certain
; cdrn-all-but-last expression.
```

THEOREM: properp-cdr-n
(length (*lst*) $\not<$ n) → (properp (cdr-n (n, *lst*)) = properp (*lst*))

```
; That relieves the hyps.  The all-but-last disappears under
; the hands of our previous lemmas, taking the terminal '(NAT 0)
; with it.  We are left with proving the equivalence of two CDR-N
; expressions.  The key is getting the n in one of them into a
; normal form to allow cdr-n-plus-append, below, to apply.
```

THEOREM: difference-x-difference
$((\mathit{len} \not< n) \land (\mathit{max} \not< \mathit{len}))$
$\rightarrow \quad ((\mathit{max} - (\mathit{len} - n)) = (n + (\mathit{max} - \mathit{len})))$

THEOREM: cdr-n-plus-append
$(i = \text{length}\,(a)) \rightarrow (\text{cdr-n}\,(i + j,\ \text{append}\,(a,\ b)) = \text{cdr-n}\,(j,\ b))$

THEOREM: cdr-n-plus-append-commuted
$(i = \text{length}\,(a)) \rightarrow (\text{cdr-n}\,(j + i,\ \text{append}\,(a,\ b)) = \text{cdr-n}\,(j,\ b))$

THEOREM: pop*-one-way-correspondence-p-r
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'pop*})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-pop*-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow \quad \text{r-equal}\,(\text{p->r}\,(\text{p-pop*-step}\,(\text{p-current-instruction}\,(p),\ p)),$
$\qquad\qquad \text{r-pop*-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p)))$

THEOREM: length-cdr-cdr-n
$\text{length}\,(\text{cdr}\,(\text{cdr-n}\,(n,\ \mathit{lst}))) = (\text{length}\,(\text{cdr}\,(\mathit{lst})) - n)$

THEOREM: properp-cdr-cdr-n
$(\text{length}\,(\mathit{lst}) \not< (1 + n)) \rightarrow (\text{properp}\,(\text{cdr}\,(\text{cdr-n}\,(n,\ \mathit{lst}))) = \text{properp}\,(\mathit{lst}))$

THEOREM: cdr-n-plus-sub1-append-commuted
$((i \not\simeq \texttt{0}) \land (j \in \mathbf{N}) \land ((i - 1) = \text{length}\,(a)))$
$\rightarrow \quad (\text{cdr-n}\,(j + i,\ \text{append}\,(a,\ b)) = \text{cdr}\,(\text{cdr-n}\,(j,\ b)))$

THEOREM: popn-one-way-correspondence-p-r
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'popn})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-popn-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow \quad \text{r-equal}\,(\text{p->r}\,(\text{p-popn-step}\,(\text{p-current-instruction}\,(p),\ p)),$
$\qquad\qquad \text{r-popn-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p)))$

```
; This ends the test section.
```

THEOREM: set-local-one-way-correspondence-p-r
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'set-local`)
∧ proper-p-statep (p)
∧ p-set-local-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-set-local-step (p-current-instruction (p), p)),
r-set-local-step (p-current-instruction (p), p->r (p)))

THEOREM: set-global-one-way-correspondence-p-r
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'set-global`)
∧ proper-p-statep (p)
∧ p-set-global-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-set-global-step (p-current-instruction (p), p)),
r-set-global-step (p-current-instruction (p), p->r (p)))

THEOREM: push-constant-one-way-correspondence-p-r
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'push-constant`)
∧ proper-p-statep (p)
∧ p-push-constant-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-push-constant-step (p-current-instruction (p), p)),
r-push-constant-step (p-current-instruction (p), p->r (p)))

THEOREM: push-ctrl-stk-free-size-one-way-correspondence-p-r
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'push-ctrl-stk-free-size`)
∧ proper-p-statep (p)
∧ p-push-ctrl-stk-free-size-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-push-ctrl-stk-free-size-step (p-current-instruction (p),
p)),
r-push-ctrl-stk-free-size-step (p-current-instruction (p),
p->r (p)))

THEOREM: push-temp-stk-free-size-one-way-correspondence-p-r
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'push-temp-stk-free-size`)
∧ proper-p-statep (p)
∧ p-push-temp-stk-free-size-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-push-temp-stk-free-size-step (p-current-instruction (p),
p)),
r-push-temp-stk-free-size-step (p-current-instruction (p),
p->r (p)))

THEOREM: push-temp-stk-index-one-way-correspondence-p-r

((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'push-temp-stk-index**)
 ∧ proper-p-statep (p)
 ∧ p-push-temp-stk-index-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-push-temp-stk-index-step (p-current-instruction (p), p)),
  r-push-temp-stk-index-step (p-current-instruction (p), p->r (p)))

THEOREM: pop-one-way-correspondence-p-r
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'pop**)
 ∧ proper-p-statep (p)
 ∧ p-pop-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-pop-step (p-current-instruction (p), p)),
  r-pop-step (p-current-instruction (p), p->r (p)))

THEOREM: jump-one-way-correspondence-p-r
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'jump**)
 ∧ proper-p-statep (p)
 ∧ p-jump-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-jump-step (p-current-instruction (p), p)),
  r-jump-step (p-current-instruction (p), p->r (p)))

THEOREM: pushj-one-way-correspondence-p-r
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'pushj**)
 ∧ proper-p-statep (p)
 ∧ p-pushj-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-pushj-step (p-current-instruction (p), p)),
  r-pushj-step (p-current-instruction (p), p->r (p)))

THEOREM: popj-one-way-correspondence-p-r
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'popj**)
 ∧ proper-p-statep (p)
 ∧ p-popj-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-popj-step (p-current-instruction (p), p)),
  r-popj-step (p-current-instruction (p), p->r (p)))

THEOREM: sub-addr-one-way-correspondence-p-r
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'sub-addr**)
 ∧ proper-p-statep (p)
 ∧ p-sub-addr-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-sub-addr-step (p-current-instruction (p), p)),
  r-sub-addr-step (p-current-instruction (p), p->r (p)))

THEOREM: eq-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'eq`)
$\land$ proper-p-statep (p)
$\land$ p-eq-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-eq-step (p-current-instruction (p), p)),
r-eq-step (p-current-instruction (p), p->r (p)))

THEOREM: lt-addr-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'lt-addr`)
$\land$ proper-p-statep (p)
$\land$ p-lt-addr-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-lt-addr-step (p-current-instruction (p), p)),
r-lt-addr-step (p-current-instruction (p), p->r (p)))

THEOREM: fetch-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'fetch`)
$\land$ proper-p-statep (p)
$\land$ p-fetch-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-fetch-step (p-current-instruction (p), p)),
r-fetch-step (p-current-instruction (p), p->r (p)))

EVENT: Disable small-integerp.

EVENT: Disable iplus.

EVENT: Disable fix-small-integer.

EVENT: Disable idifference.

EVENT: Disable inegate.

EVENT: Disable ilessp.

THEOREM: add-int-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add-int`)
$\land$ proper-p-statep (p)
$\land$ p-add-int-okp (p-current-instruction (p), p))

$\rightarrow$ r-equal (p->r (p-add-int-step (p-current-instruction (p), p)),
r-add-int-step (p-current-instruction (p), p->r (p)))

THEOREM: add-int-with-carry-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add-int-with-carry`)
$\land$ proper-p-statep (p)
$\land$ p-add-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-add-int-with-carry-step (p-current-instruction (p), p)),
r-add-int-with-carry-step (p-current-instruction (p), p->r (p)))

THEOREM: add1-int-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add1-int`)
$\land$ proper-p-statep (p)
$\land$ p-add1-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-add1-int-step (p-current-instruction (p), p)),
r-add1-int-step (p-current-instruction (p), p->r (p)))

THEOREM: sub-int-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'sub-int`)
$\land$ proper-p-statep (p)
$\land$ p-sub-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-sub-int-step (p-current-instruction (p), p)),
r-sub-int-step (p-current-instruction (p), p->r (p)))

EVENT: Enable commutativity-of-iplus.

THEOREM: sub-int-with-carry-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'sub-int-with-carry`)
$\land$ proper-p-statep (p)
$\land$ p-sub-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-sub-int-with-carry-step (p-current-instruction (p), p)),
r-sub-int-with-carry-step (p-current-instruction (p), p->r (p)))

THEOREM: sub1-int-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'sub1-int`)
$\land$ proper-p-statep (p)
$\land$ p-sub1-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-sub1-int-step (p-current-instruction (p), p)),
r-sub1-int-step (p-current-instruction (p), p->r (p)))

THEOREM: neg-int-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'neg-int`)
 ∧ proper-p-statep (p)
 ∧ p-neg-int-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-neg-int-step (p-current-instruction (p), p)),
      r-neg-int-step (p-current-instruction (p), p->r (p)))

THEOREM: lt-int-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'lt-int`)
 ∧ proper-p-statep (p)
 ∧ p-lt-int-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-lt-int-step (p-current-instruction (p), p)),
      r-lt-int-step (p-current-instruction (p), p->r (p)))

THEOREM: int-to-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'int-to-nat`)
 ∧ proper-p-statep (p)
 ∧ p-int-to-nat-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-int-to-nat-step (p-current-instruction (p), p)),
      r-int-to-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: add-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'add-nat`)
 ∧ proper-p-statep (p)
 ∧ p-add-nat-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-add-nat-step (p-current-instruction (p), p)),
      r-add-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: sub-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'sub-nat`)
 ∧ proper-p-statep (p)
 ∧ p-sub-nat-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-sub-nat-step (p-current-instruction (p), p)),
      r-sub-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: sub-nat-with-carry-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'sub-nat-with-carry`)
 ∧ proper-p-statep (p)
 ∧ p-sub-nat-with-carry-okp (p-current-instruction (p), p))

$\rightarrow$ r-equal (p->r (p-sub-nat-with-carry-step (p-current-instruction (p), p)),
r-sub-nat-with-carry-step (p-current-instruction (p), p->r (p)))

THEOREM: sub1-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'sub1-nat`)
$\land$ proper-p-statep (p)
$\land$ p-sub1-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-sub1-nat-step (p-current-instruction (p), p)),
r-sub1-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: lt-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'lt-nat`)
$\land$ proper-p-statep (p)
$\land$ p-lt-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-lt-nat-step (p-current-instruction (p), p)),
r-lt-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: mult2-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'mult2-nat`)
$\land$ proper-p-statep (p)
$\land$ p-mult2-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-mult2-nat-step (p-current-instruction (p), p)),
r-mult2-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: div2-nat-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'div2-nat`)
$\land$ proper-p-statep (p)
$\land$ p-div2-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-div2-nat-step (p-current-instruction (p), p)),
r-div2-nat-step (p-current-instruction (p), p->r (p)))

THEOREM: or-bitv-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'or-bitv`)
$\land$ proper-p-statep (p)
$\land$ p-or-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-or-bitv-step (p-current-instruction (p), p)),
r-or-bitv-step (p-current-instruction (p), p->r (p)))

THEOREM: and-bitv-one-way-correspondence-p-r
((p-psw (p) = `'run`)

$\land$ (car (p-current-instruction (p)) = `'and-bitv`)
$\land$ proper-p-statep (p)
$\land$ p-and-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-and-bitv-step (p-current-instruction (p), p)),
r-and-bitv-step (p-current-instruction (p), p->r (p)))

THEOREM: not-bitv-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'not-bitv`)
$\land$ proper-p-statep (p)
$\land$ p-not-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-not-bitv-step (p-current-instruction (p), p)),
r-not-bitv-step (p-current-instruction (p), p->r (p)))

THEOREM: xor-bitv-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'xor-bitv`)
$\land$ proper-p-statep (p)
$\land$ p-xor-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-xor-bitv-step (p-current-instruction (p), p)),
r-xor-bitv-step (p-current-instruction (p), p->r (p)))

THEOREM: rsh-bitv-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'rsh-bitv`)
$\land$ proper-p-statep (p)
$\land$ p-rsh-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-rsh-bitv-step (p-current-instruction (p), p)),
r-rsh-bitv-step (p-current-instruction (p), p->r (p)))

THEOREM: lsh-bitv-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'lsh-bitv`)
$\land$ proper-p-statep (p)
$\land$ p-lsh-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-lsh-bitv-step (p-current-instruction (p), p)),
r-lsh-bitv-step (p-current-instruction (p), p->r (p)))

EVENT: Disable and-bool.

EVENT: Disable or-bool.

EVENT: Disable not-bool.

THEOREM: or-bool-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'or-bool`)
 ∧ proper-p-statep (p)
 ∧ p-or-bool-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-or-bool-step (p-current-instruction (p), p)),
 r-or-bool-step (p-current-instruction (p), p->r (p)))

THEOREM: and-bool-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'and-bool`)
 ∧ proper-p-statep (p)
 ∧ p-and-bool-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-and-bool-step (p-current-instruction (p), p)),
 r-and-bool-step (p-current-instruction (p), p->r (p)))

THEOREM: not-bool-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'not-bool`)
 ∧ proper-p-statep (p)
 ∧ p-not-bool-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-not-bool-step (p-current-instruction (p), p)),
 r-not-bool-step (p-current-instruction (p), p->r (p)))

```
; In the next owc, we happen to compute a tstk stack pointer that
; is constant.  In particular, we generate (tag 'sys-addr (cons 'tstk n))
; for n = 0.  Thus, the tag computes and we get a constant that does
; not match a tag expression.  I therefore have to prove:
```

THEOREM: stk-equal-reflexive-tstk-instance
stk-equal (`'(sys-addr (tstk . 0))`,
 p->r_sys-data-segment (*ctrl-stk1*,
 max-ctrl-stk-size1,
 temp-stk,
 max-temp-stk-size),
 p->r_sys-data-segment (*ctrl-stk2*,
 max-ctrl-stk-size2,
 temp-stk,
 max-temp-stk-size))

THEOREM: jump-if-temp-stk-full-one-way-correspondence-p-r
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'jump-if-temp-stk-full`)
 ∧ proper-p-statep (p)

∧ p-jump-if-temp-stk-full-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-jump-if-temp-stk-full-step (p-current-instruction (p), p)),
r-jump-if-temp-stk-full-step (p-current-instruction (p), p->r (p)))

```
; The following owc depends upon the definition of empty-stkp.  Recall
; that it has been disabled.  This is the only one so far that needs
; to know what it is.  Of course, the test made by the r-level
; jump-if-temp-stk-empty crucially depends on empty-stkp being
; consistently defined with the arithmetic done at the r-level.
```

THEOREM: jump-if-temp-stk-empty-one-way-correspondence-p-r
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'jump-if-temp-stk-empty)
∧ proper-p-statep (p)
∧ p-jump-if-temp-stk-empty-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-jump-if-temp-stk-empty-step (p-current-instruction (p), p)),
r-jump-if-temp-stk-empty-step (p-current-instruction (p),
p->r (p)))

```
; All of the test instructions at the r level use the same
; predicate the p level does to determine whether the test
; is satisfied.
```

EVENT: Disable p-test-natp.

EVENT: Disable p-test-intp.

EVENT: Disable p-test-bitvp.

EVENT: Disable p-test-boolp.

THEOREM: test-nat-and-jump-one-way-correspondence-p-r
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'test-nat-and-jump)
∧ proper-p-statep (p)
∧ p-test-nat-and-jump-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-test-nat-and-jump-step (p-current-instruction (p), p)),
r-test-nat-and-jump-step (p-current-instruction (p), p->r (p)))

THEOREM: test-int-and-jump-one-way-correspondence-p-r
((p-psw (p) = 'run)

$\land$ (car (p-current-instruction (p)) = `'test-int-and-jump`)
$\land$ proper-p-statep (p)
$\land$ p-test-int-and-jump-okp (p-current-instruction (p), p))
$\to$ r-equal (p->r (p-test-int-and-jump-step (p-current-instruction (p), p)),
r-test-int-and-jump-step (p-current-instruction (p), p->r (p)))

THEOREM: test-bool-and-jump-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'test-bool-and-jump`)
$\land$ proper-p-statep (p)
$\land$ p-test-bool-and-jump-okp (p-current-instruction (p), p))
$\to$ r-equal (p->r (p-test-bool-and-jump-step (p-current-instruction (p), p)),
r-test-bool-and-jump-step (p-current-instruction (p), p->r (p)))

THEOREM: test-bitv-and-jump-one-way-correspondence-p-r
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'test-bitv-and-jump`)
$\land$ proper-p-statep (p)
$\land$ p-test-bitv-and-jump-okp (p-current-instruction (p), p))
$\to$ r-equal (p->r (p-test-bitv-and-jump-step (p-current-instruction (p), p)),
r-test-bitv-and-jump-step (p-current-instruction (p), p->r (p)))

THEOREM: length-put-value-indirect
length (put-value-indirect (val, n, $alist$)) = length ($alist$)

THEOREM: put-is-put-value-indirect
(i < length ($alist$))
$\to$ (put (val, i, append (strip-cdrs ($alist$), stk))
= append (strip-cdrs (put-value-indirect (val, i, $alist$)), stk))

THEOREM: put-plus-append-commuted
(i = length (a))
$\to$ (put (val, j + i, append (a, b)) = append (a, put (val, j, b)))

THEOREM: put-cdr-p->r_ctrl-stk-2
(proper-p-statep1 (p)
$\land$ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p)))
$\land$ (i < length (bindings (car (p-ctrl-stk (p))))))
$\to$ (put (val,
(p-max-ctrl-stk-size (p) $-$ p-ctrl-stk-size (p-ctrl-stk (p)))
+ i,
cdr (p->r_ctrl-stk (p-ctrl-stk (p), p-max-ctrl-stk-size (p))))
= cdr (p->r_ctrl-stk (cons (p-frame (put-value-indirect (val,
i,
bindings (car (p-ctrl-stk (p)))),

ret-pc (car (p-ctrl-stk (p)))),
cdr (p-ctrl-stk (p))),
p-max-ctrl-stk-size (p))))

EVENT: Disable put-plus-append-commuted.

THEOREM: p-ctrl-stk-size-put-value-indirect
listp (*ctrl-stk*)
$\rightarrow$ (p-ctrl-stk-size (cons (p-frame (put-value-indirect (*val*,
i,
bindings (car (*ctrl-stk*))),
ret-pc (car (*ctrl-stk*))),
cdr (*ctrl-stk*)))
= p-ctrl-stk-size (*ctrl-stk*))

```
; The semantics of the pop-locn instruction, at the p level,
; introduces the term (cdr (p-ctrl-stk p)), which permits
; (p-ctrl-stk-size (p-ctrl-stk p)) to expand.  When that happens
; the lemma above (at least) is no longer applicable.  Perhaps
; the problem could be avoided by proving a bridged version of that
; single result.  But rather than do that, I will simply disable
; p-ctrl-stk-size in the owc proof.
```

THEOREM: pop-locn-one-way-correspondence-p-r
((p-psw (p) = 'run)
$\land$ (car (p-current-instruction (p)) = 'pop-locn)
$\land$ proper-p-statep (p)
$\land$ p-pop-locn-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-pop-locn-step (p-current-instruction (p), p)),
r-pop-locn-step (p-current-instruction (p), p->r (p)))

THEOREM: difference-plus-x-y-y
$((x + y) - y) = \text{fix}(x)$

THEOREM: get-strip-cdrs
get (i, strip-cdrs (*alist*)) = cdr (get (i, *alist*))

THEOREM: get-non-numberp
($i \notin \mathbf{N}$) $\rightarrow$ (get (i, x) = car (x))

THEOREM: get-cdr-p->r_ctrl-stk-2
(proper-p-statep1 (p)
$\land$ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p)))
$\land$ (i < length (bindings (car (p-ctrl-stk (p))))))

$\rightarrow$ (get ((p-max-ctrl-stk-size (p) $-$ p-ctrl-stk-size (p-ctrl-stk (p)))
  $+$ i,
  cdr (p->r_ctrl-stk (p-ctrl-stk (p), p-max-ctrl-stk-size (p))))
 $=$ cdr (get (i, bindings (car (p-ctrl-stk (p))))))

EVENT: Disable get-non-numberp.

```
; I now need a more general version of get-find-position-of-var.
; This could be moved up to make the proof of the other version
; faster.
```

THEOREM: get-find-position-of-var-generalized
(var $\in$ strip-cars ($alist$))
$\rightarrow$ (get (find-position-of-var (var, strip-cars ($alist$)), $alist$)
 $=$ assoc (var, $alist$))

THEOREM: get-find-position-of-var-bindings
(proper-p-statep1 (p)
$\wedge$ (var $\in$ append (formal-vars (assoc (car (untag (p-pc (p))),
  p-prog-segment (p))),
  strip-cars (temp-var-dcls (assoc (car (untag (p-pc (p))),
  p-prog-segment (p)))))))
$\rightarrow$ (get (find-position-of-var (var,
  append (formal-vars (assoc (car (untag (p-pc (p))),
  p-prog-segment (p))),
  strip-cars (temp-var-dcls (assoc (car (untag (p-pc (p))),
  p-prog-segment (p)))))),
  bindings (car (p-ctrl-stk (p))))
 $=$ assoc (var, bindings (car (p-ctrl-stk (p)))))

THEOREM: locn-one-way-correspondence-p-r
((p-psw (p) $=$ `'run`)
$\wedge$ (car (p-current-instruction (p)) $=$ `'locn`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-locn-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-locn-step (p-current-instruction (p), p)),
  r-locn-step (p-current-instruction (p), p->r (p)))

THEOREM: jump-case-one-way-correspondence-p-r
((p-psw (p) $=$ `'run`)
$\wedge$ (car (p-current-instruction (p)) $=$ `'jump-case`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-jump-case-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-jump-case-step (p-current-instruction (p), p)),
  r-jump-case-step (p-current-instruction (p), p->r (p)))

```
; We are headed into the CALL case.  I am going to keep
; p-ctrl-stk-size disabled and open it explicitly on the newly
; pushed stack.
```

THEOREM: p-ctrl-stk-size-cons-p-frame
p-ctrl-stk-size (cons (*p-frame*, *stk*))
= (p-frame-size (*p-frame*) + p-ctrl-stk-size (*stk*))

```
; Proceeding with the proof of the owc, I first address the equality
; of the two ctrl stack sizes.  It suffices to know.
```

THEOREM: length-pairlist
length (pairlist (a, b)) = length (a)

THEOREM: length-pair-temps-with-initial-values
length (pair-temps-with-initial-values (*alist*)) = length (*alist*)

```
; The following speeds things up slightly.
```

EVENT: Enable plus-0.

```
; Next we establish that the lengths of the final tstks are the same
; at both levels.  This follows from what we already have about popn
; being cdr-n and the length of cdr-n.

; Now we proceed to the stk-equal question on the cstk.  This is the
; crux of the call proof.  By allowing p->r_ctrl-stk to open we
; turn it into an append of nat-0s, etc., and enable our previously
; implemented lemmas converting puts to pushes to work.

; That explains the first two puts that the r machine does on call,
; one to push the ret-pc, the other to push the old cfp.

; Then r does a fetch-n to get the formals and a deposit-lst to push
; the formals and locals.  We deal with that now.

; We establish that the values the r machine fetches from its
; temp stk are the same as the values the p machine fetches.
```

DEFINITION:
fetch-n-p->r_sys-data-segment-generalized-hint (n, i)
= **if** $n \simeq 0$ **then t**
else fetch-n-p->r_sys-data-segment-generalized-hint ($n - 1$, $1 + i$) **endif**

THEOREM: cdr-n-append-1
$(\text{length}(a) \not< n) \rightarrow (\text{cdr-n}(n, \text{append}(a, b)) = \text{append}(\text{cdr-n}(n, a), b))$

THEOREM: car-append{general}
$$\begin{array}{l} \text{car}(\text{append}(a, b)) \\ = \quad \textbf{if } \text{listp}(a) \ \textbf{then } \text{car}(a) \\ \qquad \textbf{else } \text{car}(b) \ \textbf{endif} \end{array}$$

EVENT: Enable listp-cdr-n.

THEOREM: fetch-n-p->r_sys-data-segment-generalized
$$\begin{array}{l} (\text{proper-p-statep1}(p) \wedge (i \in \mathbf{N}) \wedge (\text{length}(\text{cdr-n}(i, \text{p-temp-stk}(p))) \not< n)) \\ \rightarrow \quad (\text{fetch-n}(n, \\ \qquad\qquad \text{tag}(\texttt{'sys-addr}, \\ \qquad\qquad\quad \text{cons}(\texttt{'tstk}, \\ \qquad\qquad\qquad i + (\text{p-max-temp-stk-size}(p) \\ \qquad\qquad\qquad\quad - \ \text{length}(\text{p-temp-stk}(p))))), \\ \qquad\qquad \text{p->r\_sys-data-segment}(\text{p-ctrl-stk}(p), \\ \qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}(p), \\ \qquad\qquad\qquad\qquad \text{p-temp-stk}(p), \\ \qquad\qquad\qquad\qquad \text{p-max-temp-stk-size}(p))) \\ \qquad = \quad \text{first-n}(n, \text{cdr-n}(i, \text{p-temp-stk}(p)))) \end{array}$$

THEOREM: fetch-n-p->r_sys-data-segment-call
$$\begin{array}{l} (\text{proper-p-statep1}(p) \wedge (\text{length}(\text{p-temp-stk}(p)) \not< n)) \\ \rightarrow \quad (\text{fetch-n}(n, \\ \qquad\qquad \text{tag}(\texttt{'sys-addr}, \\ \qquad\qquad\quad \text{cons}(\texttt{'tstk}, \\ \qquad\qquad\qquad \text{p-max-temp-stk-size}(p) - \text{length}(\text{p-temp-stk}(p)))), \\ \qquad\qquad \text{p->r\_sys-data-segment}(\text{p-ctrl-stk}(p), \\ \qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}(p), \\ \qquad\qquad\qquad\qquad \text{p-temp-stk}(p), \\ \qquad\qquad\qquad\qquad \text{p-max-temp-stk-size}(p))) \\ \qquad = \quad \text{first-n}(n, \text{p-temp-stk}(p))) \end{array}$$

```
; So next the r machine does a deposit-lst on cstk.  Since deposit-lst
; operates on a segment, not just a single area, we have to give it
; a segment below.  The segment we give it is one obtained by doing
; a put-assoc to any other segment.  The value put by that put-assoc
; represents the cstk on which the deposits occur.  That cstk, in our
; application, happens to be the one on which the ret-pc and cfp were
; pushed.
```

DEFINITION:

deposit-lst-put-assoc-append-hint $(n,\ lst,\ stk)$
$=$ **if** $lst \simeq$ **nil then t**
else deposit-lst-put-assoc-append-hint $(n - 1,$
cdr (lst),
cons (car $(lst),\ stk)$) **endif**

THEOREM: deposit-lst-put-assoc-append
(definedp (`'cstk`, *segment*) $\wedge$ $(n \not< \text{length}(lst))$ $\wedge$ $(n \in \mathbf{N})$)
$\rightarrow$ (deposit-lst (lst,
tag (`'sys-addr`, cons (`'cstk`, $n - 1$)),
put-assoc (append (nat-0s (n), stk), `'cstk`, *segment*))
$=$ put-assoc (append (nat-0s (n $-$ length (lst)),
append (rev (lst), stk)),
`'cstk`,
segment))

```
; Note:  Each iteration we do a put.  We want our put-append-1 lemma to
; resolve it.  We don't want nat-0s expanding on us!

; To relieve the hypotheses we have to know the classic theorems:
```

EVENT: Enable length-first-n.

EVENT: Enable length-strip-cadrs.

```
; Having applied the lemma it remains to simplify the resulting rhs.  This
; requires more golden oldies:
```

THEOREM: rev-rev
properp $(x) \rightarrow (\text{rev}(\text{rev}(x)) = x)$

EVENT: Enable properp-strip-cadrs.

THEOREM: append-nil{special}
append (**nil**, x) $= x$

```
; The rule above isn't crucial of course but it speeds things
; up a bit.

; Having simplified the rhs maximally, we observe that it is
; equivalent to using p->r_sys-data-segment on a newly pushed
; p-frame.
```

```
; Before we can prove that we have to know the following classic theorems:
```

THEOREM: strip-cdrs-append
strip-cdrs (append $(a,\ b)$) = append (strip-cdrs (a), strip-cdrs (b))

THEOREM: strip-cdrs-pairlist
((length (a) = length (b)) $\wedge$ properp (b))
$\rightarrow$ (strip-cdrs (pairlist $(a,\ b)$) = b)

THEOREM: strip-cdrs-pair-temps-with-initial-values
strip-cdrs (pair-temps-with-initial-values (*alist*)) = strip-cadrs (*alist*)

THEOREM: difference-difference
$((x - y) - z) = (x - (y + z))$

EVENT: Enable properp-rev.

EVENT: Enable rev-append.

EVENT: Enable length-rev.

```
; Here then is the theorem that folds the PUT-ASSOC of the r-level
; cstk into a p->r_sys-data-segment.

; Note:  At one time I arranged this lemma to fold up into a
; p->r_ctrl-stk, with the intention of relying on
; put-assoc-cstk-p->r_sys-dat-segment-local to fold that
; up into the p->r_sys-data-segment.  That strategy did not
; work in the proof of the call owc because there I have
; p->r_ctrl-stk enabled and as soon as it was folded up
; it unfolded again.
```

THEOREM: put-assoc-cstk-p->r_sys-data-segment-call
(((p-max-ctrl-stk-size (p) $-$ 1) $-$ 1)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (p))
$+$ length (formal-vars (assoc (*name*, p-prog-segment (p))))
$+$ length (temp-var-dcls (assoc (*name*, p-prog-segment (p))))))
$\rightarrow$ (put-assoc (append (nat-0s ((((p-max-ctrl-stk-size (p)
$-$ p-ctrl-stk-size (p-ctrl-stk (p)))$-$ 1) $-$ 1)
$-$ (length (formal-vars (assoc (*name*,
p-prog-segment (p)))))

```
                   +  length (temp-var-dcls (assoc (name,
                                                   p-prog-segment (p)))))),
             append (rev (first-n (length (formal-vars (assoc (name,
                                                       p-prog-segment (p)))),
                          p-temp-stk (p))),
                  append (strip-cadrs (temp-var-dcls (assoc (name,
                                                             p-prog-segment (p)))),
                       cons (tag ('sys-addr,
                                 cons ('cstk,
                                      ((p-max-ctrl-stk-size (p)
                                        -  p-ctrl-stk-size (cdr (p-ctrl-stk (p))))- 1) - 1)),
                            cons (tag ('pc,
                                      cons (car (untag (p-pc (p))),
                                           1 + cdr (untag (p-pc (p))))),
                                 append (p->r_ctrl-stk1 (p-ctrl-stk (p),
                                                         p-max-ctrl-stk-size (p)),
                                         '((nat 0)))))))),
      'cstk,
      p->r_sys-data-segment (ctrl-stk,
                             p-max-ctrl-stk-size (p),
                             temp-stk,
                             max-temp-stk-size))
=  p->r_sys-data-segment (cons (p-frame (append (pairlist (formal-vars (assoc (name,
                                                                  p-prog-segment (p))),
                                                  rev (first-n (length (formal-vars (assoc (name,
                                                                             p-prog-segment (p)))),
                                                               p-temp-stk (p)))),
                                           pair-temps-with-initial-values (temp-var-dcls (assoc (name,
                                                                             p-prog-segment (p))))),
                                      tag ('pc,
                                           cons (car (untag (p-pc (p))),
                                                1 + cdr (untag (p-pc (p)))))),
                                p-ctrl-stk (p)),
                          p-max-ctrl-stk-size (p),
                          temp-stk,
                          max-temp-stk-size))
```

EVENT: Disable difference-difference.

```
; The lemma above takes care of the stk-equal question for cstk, given
; reflexivity.
```

THEOREM: call-one-way-correspondence-p-r

$$
\begin{array}{l}
((\text{p-psw}\,(p) = \texttt{'run})\\
\ \wedge\ \ (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'call})\\
\ \wedge\ \ \text{proper-p-statep}\,(p)\\
\ \wedge\ \ \text{p-call-okp}\,(\text{p-current-instruction}\,(p),\ p))\\
\to\ \ \text{r-equal}\,(\text{p->r}\,(\text{p-call-step}\,(\text{p-current-instruction}\,(p),\ p)),\\
\qquad\qquad \text{r-call-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p)))
\end{array}
$$

```
; pop-call does an additional pop, which causes us a little trouble.

; The following three lemmas are necessary to let us relieve the hypotheses
; of the DEPOSIT-LST-PUT-ASSOC-APPEND lemma.
```

THEOREM: not-equal-sub1-difference-p-max-ctrl-stk-size-etc

$$
\begin{array}{l}
(\text{proper-p-statep1}\,(p)\\
\ \wedge\ \ (((\text{p-max-ctrl-stk-size}\,(p) - 1) - 1)\\
\qquad \not< \ \ (\text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p))\\
\qquad\qquad + \ \ \text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\text{untag}\,(\text{car}\,(\text{p-temp-stk}\,(p))),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))\\
\qquad\qquad + \ \ \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{untag}\,(\text{car}\,(\text{p-temp-stk}\,(p))),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p)))))))\\
\to\ \ (((\text{p-max-ctrl-stk-size}\,(p) - \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p))) - 1)\\
\qquad \neq\ \ \mathtt{0})
\end{array}
$$

EVENT: Enable length-fetch-n.

THEOREM: not-lessp-sub1-sub1-difference-p-max-ctrl-stk-size-etc

$$
\begin{array}{l}
(\text{proper-p-statep1}\,(p)\\
\ \wedge\ \ (((\text{p-max-ctrl-stk-size}\,(p) - 1) - 1)\\
\qquad \not< \ \ (\text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p))\\
\qquad\qquad + \ \ \text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\text{untag}\,(\text{car}\,(\text{p-temp-stk}\,(p))),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))\\
\qquad\qquad + \ \ \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{untag}\,(\text{car}\,(\text{p-temp-stk}\,(p))),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p)))))))\\
\to\ \ (((((\text{p-max-ctrl-stk-size}\,(p) - \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p))) - 1) - 1)\\
\qquad < \ \ (\text{length}\,(\text{formal-vars}\,(\text{assoc}\,(\text{untag}\,(\text{car}\,(\text{p-temp-stk}\,(p))),\\
\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))\\
\qquad\qquad + \ \ \text{length}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\text{untag}\,(\text{car}\,(\text{p-temp-stk}\,(p))),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p))))))\\
\qquad =\ \ \mathbf{f})
\end{array}
$$

THEOREM: fetch-n-p->r_sys-data-segment-pop-call

$$
\begin{array}{l}
(\text{proper-p-statep1}\,(p)\\
\ \wedge\ \ (\text{p-max-temp-stk-size}\,(p) \not< \text{length}\,(\text{p-temp-stk}\,(p)))
\end{array}
$$

```
 ∧   (length (cdr (p-temp-stk (p))) ≮ n))
→   (fetch-n (n,
               tag ('sys-addr,
                    cons ('tstk,
                          p-max-temp-stk-size (p)
                          −   length (cdr (p-temp-stk (p))))),
               p->r_sys-data-segment (p-ctrl-stk (p),
                                      p-max-ctrl-stk-size (p),
                                      p-temp-stk (p),
                                      p-max-temp-stk-size (p)))
     =   first-n (n, cdr (p-temp-stk (p))))
```

THEOREM: put-assoc-cstk-p->r_sys-data-segment-pop-call

```
(((p-max-ctrl-stk-size (p) − 1) − 1)
 ≮   (p-ctrl-stk-size (p-ctrl-stk (p))
      +   length (formal-vars (assoc (name, p-prog-segment (p))))
      +   length (temp-var-dcls (assoc (name, p-prog-segment (p))))))
→   (put-assoc (append (nat-0s ((((p-max-ctrl-stk-size (p)
                                  −   p-ctrl-stk-size (p-ctrl-stk (p)))
                                 −   (length (formal-vars (assoc (name,
                                                                  p-prog-segment (p))))
                                      +   length (temp-var-dcls (assoc (name,
                                                                         p-prog-segment (p))))))− 1) − 1),
                       append (rev (first-n (length (formal-vars (assoc (name,
                                                                         p-prog-segment (p)))),
                                             cdr (p-temp-stk (p)))),
                               append (strip-cadrs (temp-var-dcls (assoc (name,
                                                                          p-prog-segment (p)))),
                                       cons (tag ('sys-addr,
                                                  cons ('cstk,
                                                        ((p-max-ctrl-stk-size (p)
                                                          −   p-ctrl-stk-size (cdr (p-ctrl-stk (p))))− 1) − 1)),
                                             cons (tag ('pc,
                                                        cons (car (untag (p-pc (p))),
                                                              1 + cdr (untag (p-pc (p))))),
                                                   append (p->r_ctrl-stk1 (p-ctrl-stk (p),
                                                                           p-max-ctrl-stk-size (p)),
                                                           '((nat 0)))))))),
               'cstk,
               p->r_sys-data-segment (ctrl-stk,
                                      p-max-ctrl-stk-size (p),
                                      temp-stk,
                                      max-temp-stk-size))
     =   p->r_sys-data-segment (cons (p-frame (append (pairlist (formal-vars (assoc (name,
```

p-prog-segment (p))),
rev (first-n (length (formal-vars (assoc ($name$,
p-prog-segment (p)))),
cdr (p-temp-stk (p))))),
pair-temps-with-initial-values (temp-var-dcls (assoc ($name$,
p-prog-segment (p))))),
tag (**'pc**,
cons (car (untag (p-pc (p))),
1 + cdr (untag (p-pc (p)))))),
p-ctrl-stk (p)),
p-max-ctrl-stk-size (p),
temp-stk,
max-temp-stk-size))

THEOREM: cdr-n-plus-append-pop-call
($len < max$)
$\rightarrow$ (cdr-n (($max - len$) + n, append (nat-0s (($max - len$) − 1), stk))
= cdr (cdr-n (n, stk)))

```
; pop-call has all the disable/enable hints of call; in addition,
; it disables nat-0s.  The problem is that because of the extra
; pop there is permission to expand nat-0s one more and lemmas
; fail to fire, it also enables difference-sub1 to let the extra
; pop come out where I expect it.
```

THEOREM: pop-call-one-way-correspondence-p-r
((p-psw (p) = **'run**)
$\land$ (car (p-current-instruction (p)) = **'pop-call**)
$\land$ proper-p-statep (p)
$\land$ p-pop-call-okp (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-pop-call-step (p-current-instruction (p), p)),
r-pop-call-step (p-current-instruction (p), p->r (p)))

```
; I am now headed for ret.  Because the ret command sometimes halts,
; and we're trying to prove r-equals of such states, we need to know
; what r-halt does.  I suppose I could have allowed it to expand.
```

THEOREM: components-of-r-halt
(r-pc (r-halt (r, psw)) = r-pc (r))
$\land$ (r-cfp (r-halt (r, psw)) = r-cfp (r))
$\land$ (r-csp (r-halt (r, psw)) = r-csp (r))
$\land$ (r-tsp (r-halt (r, psw)) = r-tsp (r))

$\land$ (r-x (r-halt (r, psw)) = r-x (r))
$\land$ (r-y (r-halt (r, psw)) = r-y (r))
$\land$ (r-c-flg (r-halt (r, psw)) = r-c-flg (r))
$\land$ (r-v-flg (r-halt (r, psw)) = r-v-flg (r))
$\land$ (r-n-flg (r-halt (r, psw)) = r-n-flg (r))
$\land$ (r-z-flg (r-halt (r, psw)) = r-z-flg (r))
$\land$ (r-prog-segment (r-halt (r, psw)) = r-prog-segment (r))
$\land$ (r-usr-data-segment (r-halt (r, psw)) = r-usr-data-segment (r))
$\land$ (r-sys-data-segment (r-halt (r, psw)) = r-sys-data-segment (r))
$\land$ (r-word-size (r-halt (r, psw)) = r-word-size (r))

```
; The p level machine cdrs the ctrl stack.  The r level, of course,
; doesn't, but it does set the csp and cfp by computation on the
; about-to-be popped top frame.  We need to know that the values
; it retrieves for ret-pc and the new cfp are the same as those the
; p machine uses.

; The values of cfp and ret-pc are fetched from an address based
; on the current cfp, which involves the size of the stack from
; before the top frame was pushed.  But the lemmas we have for
; explaining how to get from a stack are based on offsets from the
; the top of the stack, not negative offsets from somewhere in the
; middle.  The following lemma takes care of this by converting the
; previously computed cfp from a "somewhere in the middle" address
; to an offset from the current top.

; Warning:  The following lemma folds up p-ctrl-stk-size.  It must be not
; enabled unless that function is disabled.
```

THEOREM: difference-p-ctrl-stk-size-folder
(listp (p-ctrl-stk (p))
$\land$ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p))))
$\rightarrow$ ((p-max-ctrl-stk-size (p) $-$ p-ctrl-stk-size (cdr (p-ctrl-stk (p))))
= (1 + ((p-max-ctrl-stk-size (p)
$-$ p-ctrl-stk-size (p-ctrl-stk (p)))
$+$ length (bindings (top (p-ctrl-stk (p))))
$+$ 1)))

```
; Note: The rather cruious ordering of the addends above makes it perfect
; fodder for get-append, provided the system doesn't rearrange the
; plus nests.  Note that the classic properties of plus are disabled
; in the next lemma.

; So here are the two lemmas that tell us the r machine fetches the same
```

```
; ret-pc and cfp that the p machine does.
```

THEOREM: ret-pc-in-correct-position
$$
\begin{array}{ll}
& (\text{listp}\,(\text{p-ctrl-stk}\,(p)) \\
\wedge & (\text{p-max-ctrl-stk-size}\,(p) \not< \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p)))) \\
\rightarrow & (\text{get}\,((\text{p-max-ctrl-stk-size}\,(p) - \text{p-ctrl-stk-size}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)))) - 1, \\
& \quad \text{cdr}\,(\text{p->r\_ctrl-stk}\,(\text{p-ctrl-stk}\,(p), \text{p-max-ctrl-stk-size}\,(p)))) \\
& = \quad \text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))
\end{array}
$$

THEOREM: cfp-in-correct-position
$$
\begin{array}{ll}
& (\text{listp}\,(\text{p-ctrl-stk}\,(p)) \\
\wedge & (\text{p-max-ctrl-stk-size}\,(p) \not< \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p)))) \\
\rightarrow & (\text{get}\,(((\text{p-max-ctrl-stk-size}\,(p) - \text{p-ctrl-stk-size}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)))) - 1) - 1, \\
& \quad \text{cdr}\,(\text{p->r\_ctrl-stk}\,(\text{p-ctrl-stk}\,(p), \text{p-max-ctrl-stk-size}\,(p)))) \\
& = \quad \text{p->r\_cfp}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)), \text{p-max-ctrl-stk-size}\,(p)))
\end{array}
$$

```
; And now we shut down the lemma that folds p-ctrl-stk-size.
```

EVENT: Disable difference-p-ctrl-stk-size-folder.

```
; The next pair of lemmas help deal with the arithmetic surrounding
; the hypothesis that the ctrl-stk has more than one frame on it.
```

THEOREM: lessp-p-ctrl-stk-size
$$\text{listp}\,(stk) \rightarrow (0 < \text{p-ctrl-stk-size}\,(stk))$$

THEOREM: zerop-difference-p-ctrl-stk-size
$$
\begin{array}{ll}
& (\text{listp}\,(\text{p-ctrl-stk}\,(p)) \\
\wedge & (\text{p-max-ctrl-stk-size}\,(p) \not< \text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p)))) \\
\rightarrow & ((((((1 + \text{p-max-ctrl-stk-size}\,(p)) \\
& \quad - \quad ((\text{p-max-ctrl-stk-size}\,(p) \\
& \quad\quad - \quad \text{p-ctrl-stk-size}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)))) - 1)) - 1) - 1) \\
& \quad = \quad 0) \\
& = \quad (\text{cdr}\,(\text{p-ctrl-stk}\,(p)) \simeq \textbf{nil}))
\end{array}
$$

```
; We now consider the crux of the matter, which is proving the
; stk-equality of the ctrl stks after the ret.  Recall that until
; now we have handled stk-equal questions in two different ways:
; For cstk we arranged for the r-level operation to fold itself into
; a mapped down image of the corresponding p level operation and then
; appealed simply to the reflexivity of stk-equal.  For tstk, on the
; other hand, we adopted the much more general strategy of showing
```

```
; the equality of the live part of the two stacks.  The stk-equal
; question on tstk was converted to an equality question on cdr-n
; and all-but-last by the lemma stk-equal-tstk-p->r_sys-data-segment.

; To handle ret we will adopt the tstk strategy for cstk.
; We prove a cstk version of stk-equal-tstk-p->r_sys-data-segment.

; If you look below you will see it.  It is necessary to know
; what cdr-n and all-but-last do on p->r_ctrl-stk.
```

EVENT: Enable properp-append.

THEOREM: properp-p->r_ctrl-stk1
properp (p->r_ctrl-stk1 (*ctrl-stk*, *max-ctrl-stk-size*))

THEOREM: all-but-last-cdr-p->r_ctrl-stk
all-but-last (cdr (p->r_ctrl-stk (*ctrl-stk*, *max-ctrl-stk-size*)))
= append (nat-0s (*max-ctrl-stk-size* − p-ctrl-stk-size (*ctrl-stk*)),
p->r_ctrl-stk1 (*ctrl-stk*, *max-ctrl-stk-size*))

```
; Here then is the lemma.  The hints were formed by analogy with
; the hints for the tstk version.
```

THEOREM: stk-equal-cstk-p->r_sys-data-segment
(properp (*ctrl-stk*)
∧ (n = (*max-ctrl-stk-size* − p-ctrl-stk-size (*ctrl-stk*)))
∧ (*max-ctrl-stk-size* $\not<$ p-ctrl-stk-size (*ctrl-stk*))
∧ ((1 + *max-ctrl-stk-size*) = length (cdr (assoc (`'cstk`, *segment*)))))
→ (stk-equal (tag (`'sys-addr`, cons (`'cstk`, n)),
p->r_sys-data-segment (*ctrl-stk*,
max-ctrl-stk-size,
temp-stk,
max-temp-stk-size),
segment)
= (p->r_ctrl-stk1 (*ctrl-stk*, *max-ctrl-stk-size*)
= cdr-n (*max-ctrl-stk-size* − p-ctrl-stk-size (*ctrl-stk*),
all-but-last (cdr (assoc (`'cstk`, *segment*))))))

```
; To apply the lemma we to be able to relieve the hypothesis that
; the new ctrl-stk produced by the p machine is proper.
```

THEOREM: proper-p-ctrl-stkp-implies-properp
proper-p-ctrl-stkp (*ctrl-stk*, *name*, p) → properp (*ctrl-stk*)

THEOREM: properp-p-ctrl-stkp
$$\text{proper-p-statep1}\,(p) \rightarrow \text{properp}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)))$$

```
; Once we've done that we rewrite the stk-equal to the cdr-n of
; the all-but-last of the mapped down stack.  The all-but-last
; theorem above takes care of that function.  The following cdr-n
; lemma shows that the cdr-n is exactly enough to pop off the
; top frame.
```

THEOREM: difference-plus-plus-plus-cancellation
$$((x + (y + (z + n))) - n) = (y + (z + x))$$

```
; The odd reordering of the addends above makes the result perfect
; fodder for cdr-n-plus-append.
```

THEOREM: cdr-n-append-ret
$$\begin{array}{ll} ((\mathit{max} = (2 + \text{length}\,(\mathit{part1}) + \text{length}\,(\mathit{part2}) + \text{length}\,(\mathit{tail}))) & \\ \ \wedge \quad (\mathit{len} = \text{length}\,(\mathit{tail}))) & \\ \rightarrow \quad (\text{cdr-n}\,(\mathit{max} - \mathit{len}, \text{append}\,(\mathit{part1}, \text{append}\,(\mathit{part2}, \text{cons}\,(\mathit{e1}, \text{cons}\,(\mathit{e2}, \mathit{tail}))))) & \\ \qquad = \quad \mathit{tail}) & \end{array}$$

```
; And that is that.
```

THEOREM: ret-one-way-correspondence-p-r
$$\begin{array}{ll} ((\text{p-psw}\,(p) = \texttt{'run}) & \\ \ \wedge \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'ret}) & \\ \ \wedge \quad \text{proper-p-statep}\,(p) & \\ \ \wedge \quad \text{p-ret-okp}\,(\text{p-current-instruction}\,(p),\ p)) & \\ \rightarrow \quad \text{r-equal}\,(\text{p->r}\,(\text{p-ret-step}\,(\text{p-current-instruction}\,(p),\ p)), & \\ \qquad\qquad \text{r-ret-step}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))) & \end{array}$$

```
; We now attack the invariance of proper-p-statep.  The proof is
; as you would expect, by consideration of each of the cases.

; We first prove a beautiful theorem.  Suppose p is a proper p state
; and p' is derived from p, having certain components in common with it
; (e.g., the program segment).  Then the question "is p' proper?"
; can be reduced the some questions about its components and p.
; That may sound like just opening up proper-p-statep, but it's not.
; Opening up would reduce the question to some questions about its
; components and p'.  The nice thing about reducing it to p is that
; we will generally have many hypotheses about things being
```

```
; proper p objects.

; The amount of work we have to do to prove this lemma is an indication
; of how much it saves us, over merely opening up proper-p-statep.


; Whenever I have a same-signature as a hyp of a rule with concl
; lhs = rhs the same-signature will determine the direction of
; the rule.  In particular, (same-signature segment1 segment2)
; in a hyp means I will rewrite segment1 terms to segment2 terms.
```

EVENT: Enable same-signature-segment-length.

THEOREM: same-signature-implies-equal-p-loadablep
same-signature (*new-data-segment*, p-data-segment (p))
→ (p-loadablep (p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw),
load-addr)
= p-loadablep (p, *load-addr*))

THEOREM: same-signature-implies-equal-lengths
same-signature (*segment1*, *segment2*)
→ (length (cdr (assoc (*name*, *segment1*)))
= length (cdr (assoc (*name*, *segment2*))))

THEOREM: same-signature-implies-equal-definedp
same-signature (*segment1*, *segment2*)
→ (definedp (*name*, *segment1*) = definedp (*name*, *segment2*))

THEOREM: same-signature-implies-equal-p-objectp
same-signature (*new-data-segment*, p-data-segment (p))
→ (p-objectp (x,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-data-segment},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-temp-stk-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-word-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-psw}))$
$\qquad = \quad \text{p-objectp}\,(x,\ p))$

THEOREM: same-signature-implies-equal-all-p-objectps
same-signature (*new-data-segment*, p-data-segment (p))
$\rightarrow \quad (\text{all-p-objectps}\,(\textit{lst},$
$\qquad\qquad\qquad\qquad\qquad \text{p-state}\,(\textit{new-pc},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-ctrl-stk},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-temp-stk},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-data-segment},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-temp-stk-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-word-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-psw}))$
$\qquad = \quad \text{all-p-objectps}\,(\textit{lst},\ p))$

THEOREM: same-signature-implies-equal-proper-p-alistp
same-signature (*new-data-segment*, p-data-segment (p))
$\rightarrow \quad (\text{proper-p-alistp}\,(\textit{alist},$
$\qquad\qquad\qquad\qquad\qquad \text{p-state}\,(\textit{new-pc},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-ctrl-stk},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-temp-stk},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-data-segment},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-temp-stk-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-word-size}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-psw}))$
$\qquad = \quad \text{proper-p-alistp}\,(\textit{alist},\ p))$

THEOREM: same-signature-implies-equal-proper-p-framep
same-signature (*new-data-segment*, p-data-segment (p))
$\rightarrow \quad (\text{proper-p-framep}\,(\textit{frame},$
$\qquad\qquad\qquad\qquad\qquad \textit{name},$
$\qquad\qquad\qquad\qquad\qquad \text{p-state}\,(\textit{new-pc},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-ctrl-stk},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-temp-stk},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-data-segment},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}\,(p),$

p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-framep (*frame*, *name*, p))

EVENT: Disable proper-p-framep.

THEOREM: same-signature-implies-equal-proper-p-ctrl-stkp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-ctrl-stkp (*stk*,
name,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-ctrl-stkp (*stk*, *name*, p))

THEOREM: same-signature-implies-equal-proper-p-temp-stkp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-temp-stkp (*stk*,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-temp-stkp (*stk*, p))

THEOREM: same-signature-implies-equal-proper-p-temp-var-dclsp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-temp-var-dclsp (*lst*,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,

p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-temp-var-dclsp (*lst*, p))

THEOREM: same-signature-implies-equal-proper-p-instructionp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-instructionp (*ins*,
name,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-instructionp (*ins*, *name*, p))

THEOREM: same-signature-implies-equal-proper-labeled-p-instructionsp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-labeled-p-instructionsp (*lst*,
name,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-labeled-p-instructionsp (*lst*, *name*, p))

THEOREM: same-signature-implies-equal-proper-p-program-bodyp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-program-bodyp (*lst*,
name,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,

p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-program-bodyp (*lst*, *name*, *p*))

EVENT: Disable proper-p-program-bodyp.

THEOREM: same-signature-implies-equal-proper-p-programp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-programp (*prog*,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-programp (*prog*, *p*))

EVENT: Disable proper-p-programp.

THEOREM: same-signature-implies-equal-proper-p-prog-segmentp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-prog-segmentp (*lst*,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= proper-p-prog-segmentp (*lst*, *p*))

THEOREM: same-signature-implies-equal-proper-p-data-segmentp
same-signature (*new-data-segment*, p-data-segment (p))
→ (proper-p-data-segmentp (*lst*,
p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,

p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))

$=$ proper-p-data-segmentp $(lst,\ p)$)

```
; We are about to prove the theorem that is our main tool for
; the invariance proofs.  However, there is one major trick ahead.
; Here is a typical step theorem we will have to prove:

;(prove-lemma xxx-preserves-proper-p-statep (rewrite)
; (implies (and (equal (p-psw p) 'run)
;               (equal (car (p-current-instruction p))
;                      'xxx)
;               (proper-p-statep p)
;               (p-xxx-okp (p-current-instruction p) p))
;          (proper-p-statep
;           (p-xxx-step (p-current-instruction p) p))))

; Recall proper-p-state-restructuring.  It is an unconditional
; rewrite that replaces the hypothesized (proper-p-statep p)
; by a conjunction of (proper-p-statep1 p) and some linear
; conditions and a gratuitous but extremely useful
; (proper-p-instructionp (p-current-instruction p) p).
; We depend on that restructuring to let us access many of the
; lemmas already proved.

; However, that same restructuring will hit our concluding
; (proper-p-statep (p-xxx-step ...)) and force us to prove
; different things than we wish.  We are going to avoid this by
; a silly syntactic trick.  The step theorems we prove will
; be of the form:

;(prove-lemma xxx-preserves-proper-p-statep (rewrite)
; (implies (and (equal (p-psw p) 'run)
;               (proper-p-statep p)
;               (p-xxx-okp (p-current-instruction p) p))
;          (ur-proper-p-statep
;           (p-xxx-step (p-current-instruction p) p))))

; where ur-proper-p-statep is in fact proper-p-statep but
; is disabled.
```

DEFINITION: ur-proper-p-statep (p) = proper-p-statep (p)

```
; The lemma towards which we have been working, namely the
; lemma that rewrites (proper-p-statep p') to a conjunction
; of facts about p, will actually rewrite (ur-proper-p-statep p'):
```

THEOREM: ur-proper-p-statep-p-state
(proper-p-statep (p) $\wedge$ same-signature (*new-data-segment*, p-data-segment (p)))
$\rightarrow$ (ur-proper-p-statep (p-state (*new-pc*,
new-ctrl-stk,
new-temp-stk,
p-prog-segment (p),
new-data-segment,
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
new-psw))
= (p-objectp-type ('pc, *new-pc*, p)
$\wedge$ listp (*new-ctrl-stk*)
$\wedge$ proper-p-framep (top (*new-ctrl-stk*), area-name (*new-pc*), p)
$\wedge$ proper-p-ctrl-stkp (pop (*new-ctrl-stk*),
area-name (ret-pc (top (*new-ctrl-stk*))),
p)
$\wedge$ (p-max-ctrl-stk-size (p)
$\not<$ p-ctrl-stk-size (*new-ctrl-stk*))
$\wedge$ proper-p-temp-stkp (*new-temp-stk*, p)
$\wedge$ (p-max-temp-stk-size (p) $\not<$ length (*new-temp-stk*))
$\wedge$ proper-p-data-segmentp (*new-data-segment*, p)))

EVENT: Disable ur-proper-p-statep.

```
; When we are all done we will reveal that ur-proper-p-statep
; is proper-p-statep and get the invariance theorem.

; To apply the main lemma above we need to establish the
; same-signature hypothesis.  We will prove the necessary
; lemmas as needed.  For the no-op case it is:
```

EVENT: Enable same-signature-reflexive.

```
; We also have to be able to establish (proper-p-statep p)
; in a context in which it has been restructured.  This is
; a new problem -- in our previous proofs we always went after
; (proper-p-statep1 p) or explicit inequalities.  Anyway, to
; get (proper-p-statep p) it is sufficient to know:
```

THEOREM: listp-p-current-instruction
$(\text{car}\,(\text{p-current-instruction}\,(p)) = \text{pack}\,(xxx))$
$\rightarrow \quad \text{listp}\,(\text{p-current-instruction}\,(p))$

EVENT: Enable proper-p-framep.

THEOREM: no-op-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'no-op})$
$\wedge \quad \text{proper-p-statep}\,(p)$
$\wedge \quad \text{p-no-op-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}\,(\text{p-no-op-step}\,(\text{p-current-instruction}\,(p),\ p))$

```
; To prove push-global we need to know that the object pushed is
; a p-objectp.  The object pushed is fetched from the data segment.
; So we have to prove the general fact that anything fetched from
; the data segment is a p-objectp.  This, of course, will be used
; in many of the subsequent proofs.
```

THEOREM: all-p-objectps-implies-p-objectp
$(\text{all-p-objectps}\,(lst,\ p) \wedge (x \in lst)) \rightarrow \text{p-objectp}\,(x,\ p)$

THEOREM: proper-p-data-segmentp-implies-all-p-objectps
$(\text{proper-p-data-segmentp}\,(segment,\ p) \wedge \text{definedp}\,(name,\ segment))$
$\rightarrow \quad \text{all-p-objectps}\,(\text{cdr}\,(\text{assoc}\,(name,\ segment)),\ p)$

THEOREM: p-objectp-get-p-data-segment
$(\text{proper-p-data-segmentp}\,(segment,\ p)$
$\wedge \quad \text{definedp}\,(name,\ segment)$
$\wedge \quad (n < \text{length}\,(\text{cdr}\,(\text{assoc}\,(name,\ segment)))))$
$\rightarrow \quad \text{p-objectp}\,(\text{get}\,(n,\ \text{cdr}\,(\text{assoc}\,(name,\ segment))),\ p)$

```
; To access global vars we use a naked car of the global data
; area, knowing that it is a listp -- i.e., there is not
; legal address check.  In addition, we don't get via the
; function get and so the lemma above is not quite what
; we need.  We take care of these details here.
```

THEOREM: definedp-p-data-segment-implies-listp-generalized
(proper-p-data-segmentp ($segment$, p) $\land$ definedp ($name$, $segment$))
$\rightarrow$ listp (cdr (assoc ($name$, $segment$)))

THEOREM: definedp-p-data-segment-implies-listp
(proper-p-statep1 (p) $\land$ definedp ($name$, p-data-segment (p)))
$\rightarrow$ listp (cdr (assoc ($name$, p-data-segment (p))))

THEOREM: p-objectp-cadr-assoc-p-data-segment
(proper-p-statep1 (p) $\land$ definedp ($name$, p-data-segment (p)))
$\rightarrow$ p-objectp (car (cdr (assoc ($name$, p-data-segment (p)))), p)

```
; So that sets us up for:
```

THEOREM: push-global-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'push-global`)
$\land$ proper-p-statep (p)
$\land$ p-push-global-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-push-global-step (p-current-instruction (p), p))

```
; To do pop-global we need first to relieve the hypotheses on our
; ur-proper-p-statep-p-state lemma, namely the same-signature
; hypothesis on the data segment.  The new data segment in
; question is obtained by put-assoc and put.  So we prove:
```

THEOREM: same-signature-put-assoc-generalized
(proper-p-data-segmentp ($segment$, p)
$\land$ definedp ($name$, $segment$)
$\land$ (length (val) = length (cdr (assoc ($name$, $segment$)))))
$\rightarrow$ same-signature (put-assoc (val, $name$, $segment$), $segment$)

```
; The proper-p-data-segmentp hyp above is necessary only to assure us that
; the alist does not have any nlistp members, so a less sophisticated
; (and state free) predicate could have been used.  But since I had this
; one I used it.  Unfortunately, it introduces a free variable, p, so
; I eliminate it and make it a rewrite rule by choosing the only
; instantiation I ever need:
```

THEOREM: same-signature-put-assoc{special}
(proper-p-statep1 (p)
$\land$ definedp ($name$, p-data-segment (p))
$\land$ (length (val) = length (cdr (assoc ($name$, p-data-segment (p))))))
$\rightarrow$ same-signature (put-assoc (val, $name$, p-data-segment (p)),
p-data-segment (p))

```
; Since pop-global pops the temp stack, we have to prove that the new
; one is proper:
```

THEOREM: proper-p-temp-stk-cdr
(listp (*stk*) $\land$ proper-p-temp-stkp (*stk*, *p*))
$\rightarrow$ proper-p-temp-stkp (cdr (*stk*), *p*)

```
; We wouldn't need the theorem above if proper-p-statep were opened in the
; hypothesis because the (proper-p-temp-stkp (p-temp-stk p) p) would open
; given our interest in (cdr (p-temp-stk p)).

; Finally, since pop-global does a put into the data segment, we need to
; to prove that proper-p-data-segmentp is preserved by the put-assoc and
; put.
```

THEOREM: all-p-objectps-put
(p-objectp (*val*, *p*) $\land$ ($n <$ length (*lst*)) $\land$ all-p-objectps (*lst*, *p*))
$\rightarrow$ all-p-objectps (put (*val*, *n*, *lst*), *p*)

EVENT: Enable definedp-put-assoc.

THEOREM: proper-p-data-segmentp-put-assoc
(proper-p-data-segmentp (*segment*, *p*)
$\land$ definedp (*name*, *segment*)
$\land$ listp (*lst*)
$\land$ all-p-objectps (*lst*, *p*))
$\rightarrow$ proper-p-data-segmentp (put-assoc (*lst*, *name*, *segment*), *p*)

```
; The lemma above does the job, but note that we need to know that
; the thing we put into the data area is a p-objectp.  What is that
; thing?  It's the top of the temp stack.  So we need to prove:
```

THEOREM: p-objectp-top-p-temp-stk
(proper-p-temp-stkp (*temp-stk*, *p*) $\land$ listp (*temp-stk*))
$\rightarrow$ p-objectp (car (*temp-stk*), *p*)

```
; We also get the similar results for top1 and top2:
```

THEOREM: p-objectp-top1-p-temp-stk
(proper-p-temp-stkp (*temp-stk*, *p*) $\land$ listp (cdr (*temp-stk*)))
$\rightarrow$ p-objectp (cadr (*temp-stk*), *p*)

THEOREM: p-objectp-top2-p-temp-stk
(proper-p-temp-stkp ($temp\text{-}stk$, p) $\land$ listp (cddr ($temp\text{-}stk$)))
$\rightarrow$ p-objectp (caddr ($temp\text{-}stk$), p)

```
; For CALL I will need a more general lemma about the first-n
; of the temp-stk, but I'll wait till I see the need.

; The provision of the three lemmas above interacts in a detrimental
; way with something else in our proofs.  Consider p-deposit-okp,
; for example.  It requires that the top of the temp-stk by a
; p-objectp of type 'addr.  When we assume that -okp we have been
; getting the p-objectp in our assumptions and the p-objectp has then
; been opening up to reveal important things like that the car of the
; object is definedp in the data-segment or, in the case of the arithmetic
; instructions, the object is a numberp.  But with the provision of the
; three p-objectp lemmas above, whenever we hypothesis that something on
; the stack is a p-objectp of a given type the hypothesis is rewritten
; simply to the fact that it is of the right type -- because the p-objectp
; is obvious by the thing merely being on the stack.  This in fact
; suggests an improvement we could make in the defns of all the -okp
; fns:  don't check p-objectp of things on the stack, just check their
; types.  I'll save that for another day.

; Meanwhile, we have a problem:  lots of rules have definedp and numberp
; hyps that will no longer be relieved because the three lemmas above
; have simplified away the p-objectps that used to open up.  What to do?

; I am taking a really simple way out, exploiting the fact that all the
; -okp fns that check for the type of an object do so with
; p-objectp-type.  I will disable it after proving the rule below, which
; not only expands p-objectp-type but also the resulting p-objectp.
; This trick just slips the p-objectp past the three lemmas above.
```

THEOREM: p-objectp-type-opener
(p-objectp-type ('**nat**, x, p)
= ((type (x) = '**nat**)
$\land$ (cddr (x) = **nil**)
$\land$ small-naturalp (untag (x), p-word-size (p))))
$\land$ (p-objectp-type ('**int**, x, p)
= ((type (x) = '**int**)
$\land$ (cddr (x) = **nil**)
$\land$ small-integerp (untag (x), p-word-size (p))))
$\land$ (p-objectp-type ('**bitv**, x, p)

$= \quad ((\text{type}(x) = \texttt{'bitv})$
$\quad \wedge \quad (\text{cddr}(x) = \textbf{nil})$
$\quad \wedge \quad \text{bit-vectorp}(\text{untag}(x),\ \text{p-word-size}(p))))$
$\wedge \quad (\text{p-objectp-type}(\texttt{'bool},\ x,\ p)$
$= \quad ((\text{type}(x) = \texttt{'bool})$
$\quad \wedge \quad (\text{cddr}(x) = \textbf{nil})$
$\quad \wedge \quad \text{booleanp}(\text{untag}(x))))$
$\wedge \quad (\text{p-objectp-type}(\texttt{'addr},\ x,\ p)$
$= \quad ((\text{type}(x) = \texttt{'addr})$
$\quad \wedge \quad (\text{cddr}(x) = \textbf{nil})$
$\quad \wedge \quad \text{adpp}(\text{untag}(x),\ \text{p-data-segment}(p))))$
$\wedge \quad (\text{p-objectp-type}(\texttt{'pc},\ x,\ p)$
$= \quad ((\text{type}(x) = \texttt{'pc})$
$\quad \wedge \quad (\text{cddr}(x) = \textbf{nil})$
$\quad \wedge \quad \text{pcpp}(\text{untag}(x),\ \text{p-prog-segment}(p))))$
$\wedge \quad (\text{p-objectp-type}(\texttt{'subr},\ x,\ p)$
$= \quad ((\text{type}(x) = \texttt{'subr})$
$\quad \wedge \quad (\text{cddr}(x) = \textbf{nil})$
$\quad \wedge \quad \text{definedp}(\text{untag}(x),\ \text{p-prog-segment}(p))))$

Event: Disable p-objectp-type.

```
; Confession:  I feel a little dirty for this.  If I ever remove
; the p-objectp tests in the -okp predicates, this will return to
; bite me.  By the way, this problem first arose when I did
; the deposit preservation lemma below.  I put the solution up
; here to keep it in proximity with the lemmas that caused the
; trouble.
```

Theorem: pop-global-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\wedge \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'pop-global})$
$\wedge \quad \text{proper-p-statep}(p)$
$\wedge \quad \text{p-pop-global-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-pop-global-step}(\text{p-current-instruction}(p),\ p))$

```
; We need similar machinery to do local variable manipulation.
; In particular, we need to know that any var value extracted
; from the top frame of the ctrl stack is a p-objectp and that
; depositing any p-objectp there leaves the stack proper.

; Here's extraction:
```

THEOREM: p-objectp-cdr-assoc-proper-p-alistp
(proper-p-alistp ($alist$, p) $\wedge$ ($var \in$ strip-cars ($alist$)))
$\rightarrow$ p-objectp (cdr (assoc (var, $alist$)), p)

```
; So we get:
```

THEOREM: push-local-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'push-local`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-push-local-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-push-local-step (p-current-instruction (p), p))

```
; And now we deal with depositing.  The p machine puts a
; new frame on the stack, replacing the old one.  We need
; to prove that the new frame is proper.  There are many
; parts.  The new frame is built with P-FRAME and the
; "shape" requirements follow immediately from its defn:
```

THEOREM: p-frame-properties
listp (cdr (p-frame ($alist$, $ret\text{-}pc$))) $\wedge$ (cddr (p-frame ($alist$, $ret\text{-}pc$)) = **nil**)

```
; The requirement that the new bindings be a proper-p-alistp
; is established by:
```

THEOREM: proper-p-alistp-put-assoc
(proper-p-alistp ($alist$, p)
$\wedge$ ($var \in$ strip-cars ($alist$))
$\wedge$ p-objectp (val, p))
$\rightarrow$ proper-p-alistp (put-assoc (val, var, $alist$), p)

```
; The requirement that the strip-cars of the bindings be
; the local-vars of the current subroutine follows from the
; observation that the strip-cars of the new bindings is the
; same as for the old:
```

THEOREM: strip-cars-put-assoc
strip-cars (put-assoc (val, var, $alist$)) = strip-cars ($alist$)

```
; That completes the proof that the new frame is proper.  We
; also have to prove that the new ctrl stack isn't too big.
; This follows immediately from the fact that the new stack is
```

```
; the same size as the old, but that simple observation doesn't
; help us unless we disable some arithmetic stuff, because the
; sizes are computed and cause some linear arithmetic splits.
; Rather than disable I'll just handle the splits.  It suffices
; to know:
```

THEOREM: not-equal-p-ctrl-stk-size-1
proper-p-statep1 $(p) \rightarrow ((\text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(p)) - 1) \neq \texttt{0})$

```
; That completes the entire pop-local exercise.
```

THEOREM: pop-local-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'pop-local})$
$\land$ proper-p-statep (p)
$\land$ p-pop-local-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-pop-local-step (p-current-instruction (p), p))

THEOREM: deposit-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'deposit})$
$\land$ proper-p-statep (p)
$\land$ p-deposit-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-deposit-step (p-current-instruction (p), p))

THEOREM: add1-nat-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'add1-nat})$
$\land$ proper-p-statep (p)
$\land$ p-add1-nat-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-add1-nat-step (p-current-instruction (p), p))

THEOREM: add-addr-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'add-addr})$
$\land$ proper-p-statep (p)
$\land$ p-add-addr-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-add-addr-step (p-current-instruction (p), p))

```
; I still need some facts about lessp length and cdr!
; I hope this will take care of it:
```

THEOREM: not-lessp-length-cdr-backchainer
$(max \not< \text{length}\,(lst)) \rightarrow ((max < \text{length}\,(\text{cdr}\,(lst))) = \mathbf{f})$

```
; For add-nat-with-carry we need to prove that our computed
; result is a p-objectp.  It suffices to know:
```

THEOREM: small-naturalp-fix-small-natural
small-naturalp (fix-small-natural (i, n), n)

```
; These kind of lemmas will be required each time we shift to
; consideration of a new data type.
```

THEOREM: add-nat-with-carry-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'add-nat-with-carry`)
 ∧ proper-p-statep (p)
 ∧ p-add-nat-with-carry-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-add-nat-with-carry-step (p-current-instruction (p), p))

THEOREM: mult2-nat-with-carry-out-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'mult2-nat-with-carry-out`)
 ∧ proper-p-statep (p)
 ∧ p-mult2-nat-with-carry-out-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-mult2-nat-with-carry-out-step (p-current-instruction (p), p))

THEOREM: p-objectp-get-temp-stk
(proper-p-temp-stkp (*temp-stk*, p) ∧ ($n <$ length (*temp-stk*)))
→ p-objectp (get (n, *temp-stk*), p)

THEOREM: fetch-temp-stk-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'fetch-temp-stk`)
 ∧ proper-p-statep (p)
 ∧ p-fetch-temp-stk-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-fetch-temp-stk-step (p-current-instruction (p), p))

THEOREM: proper-p-temp-stkp-put
(proper-p-temp-stkp (*temp-stk*, p)
 ∧ p-objectp (*val*, p)
 ∧ ($n <$ length (*temp-stk*)))
→ proper-p-temp-stkp (put (*val*, n, *temp-stk*), p)

THEOREM: deposit-temp-stk-preserves-proper-p-statep

$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'deposit-temp-stk})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-deposit-temp-stk-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-deposit-temp-stk-step}(\text{p-current-instruction}(p),\ p))$

```
; For POP* we clearly need:
```

THEOREM: proper-p-temp-stkp-cdr-n
$(\text{proper-p-temp-stkp}(stk,\ p) \land (\text{length}(stk) \not< n))$
$\rightarrow \quad \text{proper-p-temp-stkp}(\text{cdr-n}(n,\ stk),\ p)$

THEOREM: pop*-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'pop*})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-pop*-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-pop*-step}(\text{p-current-instruction}(p),\ p))$

THEOREM: popn-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'popn})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-popn-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-popn-step}(\text{p-current-instruction}(p),\ p))$

```
; For what it's worth, this concludes the so-called "test section"
; of instructions identified as interesting in the case of the
; one-way-correspondence proof.
```

THEOREM: set-local-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'set-local})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-set-local-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-set-local-step}(\text{p-current-instruction}(p),\ p))$

THEOREM: set-global-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'set-global})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-set-global-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-set-global-step}(\text{p-current-instruction}(p),\ p))$

```
; To get push-constant we have to know that the case analysis done
; to unabbreviate the constant is the same as that done in the
; syntax check, proper-p-push-constant-instructionp.  It suffices
; to know:
```

THEOREM: nlistp-implies-not-p-objectp
$(x \simeq \textbf{nil}) \rightarrow (\text{p-objectp}\,(x,\ p) = \textbf{f})$

THEOREM: labels-are-nlistp-generalized
$(\text{proper-labeled-p-instructionsp}\,(body,\ name,\ p) \land \text{listp}\,(lab))$
$\rightarrow\quad (\neg\ \text{find-labelp}\,(lab,\ body))$

THEOREM: proper-p-prog-segmentp-implies-proper-labled-p-instructionsp
$(\text{proper-p-prog-segmentp}\,(segment,\ p) \land \text{definedp}\,(name,\ segment))$
$\rightarrow\quad \text{proper-labeled-p-instructionsp}\,(\text{program-body}\,(\text{assoc}\,(name,\ segment)),\ name,\ p)$

THEOREM: labels-are-nlistp
$(\text{proper-p-statep1}\,(p) \land \text{definedp}\,(name,\ \text{p-prog-segment}\,(p)) \land \text{listp}\,(lab))$
$\rightarrow\quad (\neg\ \text{find-labelp}\,(lab,\ \text{program-body}\,(\text{assoc}\,(name,\ \text{p-prog-segment}\,(p)))))$

```
; The following is needed to prove that when a "label" is pushed
; the object is a p-objectp.
```

THEOREM: lessp-find-label
$\text{find-labelp}\,(lab,\ lst) \rightarrow (\text{find-label}\,(lab,\ lst) < \text{length}\,(lst))$

THEOREM: push-constant-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'push-constant})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-push-constant-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow\quad \text{ur-proper-p-statep}\,(\text{p-push-constant-step}\,(\text{p-current-instruction}\,(p),\ p))$

```
; To get push-ctrl-stk-free-size we need to know that the number
; pushed is a small-naturalp.  Despite its name the lemma below
; actually shows that any number less than p-max-ctrl-stk-size is
; ok.
```

THEOREM: proper-p-statep-implies-small-naturalp-p-max-ctrl-stk-size
$(\text{p-max-ctrl-stk-size}\,(p) < \exp\,(2,\ \text{p-word-size}\,(p)))$
$\rightarrow\quad \text{small-naturalp}\,(\text{p-max-ctrl-stk-size}\,(p) - n,\ \text{p-word-size}\,(p))$

THEOREM: push-ctrl-stk-free-size-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'push-ctrl-stk-free-size})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-push-ctrl-stk-free-size-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-push-ctrl-stk-free-size-step}(\text{p-current-instruction}(p),$
$p))$

```
; We prove an analogous lemma for p-max-temp-stk-size:
```

THEOREM: proper-p-statep-implies-small-naturalp-p-max-temp-stk-size
$(\text{p-max-temp-stk-size}(p) < \exp(2, \text{p-word-size}(p)))$
$\rightarrow \quad \text{small-naturalp}(\text{p-max-temp-stk-size}(p) - n, \text{p-word-size}(p))$

THEOREM: push-temp-stk-free-size-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'push-temp-stk-free-size})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-push-temp-stk-free-size-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-push-temp-stk-free-size-step}(\text{p-current-instruction}(p),$
$p))$

```
; And we prove the analogous lemma for the index pushed by push-temp-stk-index.
```

THEOREM: proper-p-statep-implies-small-naturalp-length-p-temp-stk
$((\text{p-max-temp-stk-size}(p) < \exp(2, \text{p-word-size}(p)))$
$\land \quad (\text{length}(\text{p-temp-stk}(p)) < \text{p-max-temp-stk-size}(p)))$
$\rightarrow \quad \text{small-naturalp}((\text{length}(\text{p-temp-stk}(p)) - n) - 1, \text{p-word-size}(p))$

THEOREM: push-temp-stk-index-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'push-temp-stk-index})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-push-temp-stk-index-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-push-temp-stk-index-step}(\text{p-current-instruction}(p),$
$p))$

THEOREM: pop-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'pop})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-pop-okp}(\text{p-current-instruction}(p),\ p))$
$\rightarrow \quad \text{ur-proper-p-statep}(\text{p-pop-step}(\text{p-current-instruction}(p),\ p))$

THEOREM: jump-preserves-proper-p-statep
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'jump)
 ∧ proper-p-statep (p)
 ∧ p-jump-okp (p-current-instruction (p), p))
 → ur-proper-p-statep (p-jump-step (p-current-instruction (p), p))

THEOREM: pushj-preserves-proper-p-statep
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'pushj)
 ∧ proper-p-statep (p)
 ∧ p-pushj-okp (p-current-instruction (p), p))
 → ur-proper-p-statep (p-pushj-step (p-current-instruction (p), p))

THEOREM: popj-preserves-proper-p-statep
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'popj)
 ∧ proper-p-statep (p)
 ∧ p-popj-okp (p-current-instruction (p), p))
 → ur-proper-p-statep (p-popj-step (p-current-instruction (p), p))

THEOREM: sub-addr-preserves-proper-p-statep
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'sub-addr)
 ∧ proper-p-statep (p)
 ∧ p-sub-addr-okp (p-current-instruction (p), p))
 → ur-proper-p-statep (p-sub-addr-step (p-current-instruction (p), p))

```
; Since bool is disabled we need:
```

THEOREM: p-objectp-bool
p-objectp (bool (x), p)

THEOREM: eq-preserves-proper-p-statep
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'eq)
 ∧ proper-p-statep (p)
 ∧ p-eq-okp (p-current-instruction (p), p))
 → ur-proper-p-statep (p-eq-step (p-current-instruction (p), p))

THEOREM: lt-addr-preserves-proper-p-statep
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'lt-addr)
 ∧ proper-p-statep (p)
 ∧ p-lt-addr-okp (p-current-instruction (p), p))
 → ur-proper-p-statep (p-lt-addr-step (p-current-instruction (p), p))

THEOREM: fetch-preserves-proper-p-statep
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'fetch)
∧ proper-p-statep (p)
∧ p-fetch-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-fetch-step (p-current-instruction (p), p))

THEOREM: add-int-preserves-proper-p-statep
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'add-int)
∧ proper-p-statep (p)
∧ p-add-int-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-add-int-step (p-current-instruction (p), p))

EVENT: Enable small-integerp-fix-small-integer-plus-case.

THEOREM: add-int-with-carry-preserves-proper-p-statep
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'add-int-with-carry)
∧ proper-p-statep (p)
∧ p-add-int-with-carry-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-add-int-with-carry-step (p-current-instruction (p),
p))

THEOREM: add1-int-preserves-proper-p-statep
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'add1-int)
∧ proper-p-statep (p)
∧ p-add1-int-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-add1-int-step (p-current-instruction (p), p))

THEOREM: sub-int-preserves-proper-p-statep
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'sub-int)
∧ proper-p-statep (p)
∧ p-sub-int-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-sub-int-step (p-current-instruction (p), p))

EVENT: Enable small-integerp-fix-small-integer-difference-case.

THEOREM: sub-int-with-carry-preserves-proper-p-statep
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'sub-int-with-carry)
∧ proper-p-statep (p)

$\wedge$ p-sub-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-sub-int-with-carry-step (p-current-instruction (p),
p))

THEOREM: sub1-int-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'sub1-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-sub1-int-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-sub1-int-step (p-current-instruction (p), p))

THEOREM: neg-int-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'neg-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-neg-int-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-neg-int-step (p-current-instruction (p), p))

THEOREM: lt-int-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'lt-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-lt-int-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-lt-int-step (p-current-instruction (p), p))

```
; For int-to-nat conversion we clearly need:
```

THEOREM: small-integerp-implies-small-naturalp
(small-integerp (x, *word-size*) $\wedge$ ($\neg$ negativep (x)))
$\rightarrow$ small-naturalp (x, *word-size*)

THEOREM: int-to-nat-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'int-to-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-int-to-nat-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-int-to-nat-step (p-current-instruction (p), p))

THEOREM: add-nat-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'add-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-add-nat-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-add-nat-step (p-current-instruction (p), p))

THEOREM: small-naturalp-difference
small-naturalp $(x,\ \textit{word-size}) \rightarrow$ small-naturalp $(x - y,\ \textit{word-size})$

THEOREM: sub-nat-preserves-proper-p-statep
((p-psw (p) = 'run)
$\wedge$ (car (p-current-instruction (p)) = 'sub-nat)
$\wedge$ proper-p-statep (p)
$\wedge$ p-sub-nat-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-sub-nat-step (p-current-instruction (p), p))

```
; sub-nat-with-carry flag returns one of two different
; differences depending on the args.  We have to show that
; both are small-naturalps.  The first is handled by
; small-natural-difference above.

; The second is:
```

THEOREM: small-naturalp-difference-exp-x
$(x \not\simeq 0) \rightarrow$ small-naturalp (exp $(2,\ \textit{word-size}) - x,\ \textit{word-size}$)

THEOREM: sub-nat-with-carry-preserves-proper-p-statep
((p-psw (p) = 'run)
$\wedge$ (car (p-current-instruction (p)) = 'sub-nat-with-carry)
$\wedge$ proper-p-statep (p)
$\wedge$ p-sub-nat-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-sub-nat-with-carry-step (p-current-instruction (p),
p))

THEOREM: small-naturalp-sub1
small-naturalp $(x,\ \textit{word-size}) \rightarrow$ small-naturalp $(x - 1,\ \textit{word-size})$

THEOREM: sub1-nat-preserves-proper-p-statep
((p-psw (p) = 'run)
$\wedge$ (car (p-current-instruction (p)) = 'sub1-nat)
$\wedge$ proper-p-statep (p)
$\wedge$ p-sub1-nat-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-sub1-nat-step (p-current-instruction (p), p))

THEOREM: lt-nat-preserves-proper-p-statep
((p-psw (p) = 'run)
$\wedge$ (car (p-current-instruction (p)) = 'lt-nat)
$\wedge$ proper-p-statep (p)
$\wedge$ p-lt-nat-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-lt-nat-step (p-current-instruction (p), p))

THEOREM: mult2-nat-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'mult2-nat})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-mult2-nat-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to\quad \text{ur-proper-p-statep}\,(\text{p-mult2-nat-step}\,(\text{p-current-instruction}\,(p),\ p))$

```
; For div2-nat we must establish that the results are small-naturalps.
```

THEOREM: not-lessp-remainder
$x \not< (x \ \mathbf{mod}\ y)$

EVENT: Enable not-lessp-quotient.

THEOREM: small-naturalp-remainder
$\text{small-naturalp}\,(x,\ \textit{word-size}) \to \text{small-naturalp}\,(x\ \mathbf{mod}\ \texttt{2},\ \textit{word-size})$

THEOREM: small-naturalp-quotient
$\text{small-naturalp}\,(x,\ \textit{word-size}) \to \text{small-naturalp}\,(x \div \texttt{2},\ \textit{word-size})$

THEOREM: div2-nat-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'div2-nat})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-div2-nat-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to\quad \text{ur-proper-p-statep}\,(\text{p-div2-nat-step}\,(\text{p-current-instruction}\,(p),\ p))$

```
; We must show that each bit vector operation yields a bit vector:
```

THEOREM: bit-vectorp-or-bitv
$(\text{bit-vectorp}\,(x,\ \textit{word-size}) \land \text{bit-vectorp}\,(y,\ \textit{word-size}))$
$\to\quad \text{bit-vectorp}\,(\text{or-bitv}\,(x,\ y),\ \textit{word-size})$

THEOREM: bit-vectorp-and-bitv
$(\text{bit-vectorp}\,(x,\ \textit{word-size}) \land \text{bit-vectorp}\,(y,\ \textit{word-size}))$
$\to\quad \text{bit-vectorp}\,(\text{and-bitv}\,(x,\ y),\ \textit{word-size})$

THEOREM: bit-vectorp-xor-bitv
$(\text{bit-vectorp}\,(x,\ \textit{word-size}) \land \text{bit-vectorp}\,(y,\ \textit{word-size}))$
$\to\quad \text{bit-vectorp}\,(\text{xor-bitv}\,(x,\ y),\ \textit{word-size})$

THEOREM: bit-vectorp-not-bitv
$\text{bit-vectorp}\,(x,\ \textit{word-size}) \to \text{bit-vectorp}\,(\text{not-bitv}\,(x),\ \textit{word-size})$

```
; It surprised me to see that we need, yet again, the fact that our
; machine has a non-zero word-size.
```

THEOREM: bit-vectorp-rsh-bitv
(bit-vectorp (x, *word-size*) $\wedge$ (*word-size* $\not\simeq$ 0))
$\rightarrow$ bit-vectorp (rsh-bitv (x), *word-size*)

EVENT: Disable rsh-bitv.

THEOREM: bit-vectorp-lsh-bitv
(bit-vectorp (x, *word-size*) $\wedge$ (*word-size* $\not\simeq$ 0))
$\rightarrow$ bit-vectorp (lsh-bitv (x), *word-size*)

EVENT: Disable lsh-bitv.

```
; To use the latter two rules we need:
```

THEOREM: p-loadablep-implies-not-zerop-p-word-size
p-loadablep (p, *load-addr*)
$\rightarrow$ ((p-word-size (p) $\in \mathbf{N}$) $\wedge$ (p-word-size (p) $\neq$ 0))

THEOREM: or-bitv-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'or-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-or-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-or-bitv-step (p-current-instruction (p), p))

THEOREM: and-bitv-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'and-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-and-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-and-bitv-step (p-current-instruction (p), p))

THEOREM: not-bitv-preserves-proper-p-statep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'not-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-not-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ ur-proper-p-statep (p-not-bitv-step (p-current-instruction (p), p))

THEOREM: xor-bitv-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'xor-bitv})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-xor-bitv-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to \quad \text{ur-proper-p-statep}\,(\text{p-xor-bitv-step}\,(\text{p-current-instruction}\,(p),\ p))$

THEOREM: rsh-bitv-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'rsh-bitv})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-rsh-bitv-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to \quad \text{ur-proper-p-statep}\,(\text{p-rsh-bitv-step}\,(\text{p-current-instruction}\,(p),\ p))$

THEOREM: lsh-bitv-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'lsh-bitv})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-lsh-bitv-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to \quad \text{ur-proper-p-statep}\,(\text{p-lsh-bitv-step}\,(\text{p-current-instruction}\,(p),\ p))$

```
; And we must do the analogous thing for the boolean type and its
; operators.
```

THEOREM: booleanp-or-bool
$(\text{booleanp}\,(x) \land \text{booleanp}\,(y)) \to \text{booleanp}\,(\text{or-bool}\,(x,\ y))$

THEOREM: booleanp-and-bool
$(\text{booleanp}\,(x) \land \text{booleanp}\,(y)) \to \text{booleanp}\,(\text{and-bool}\,(x,\ y))$

THEOREM: booleanp-not-bool
$\text{booleanp}\,(x) \to \text{booleanp}\,(\text{not-bool}\,(x))$

THEOREM: or-bool-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'or-bool})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-or-bool-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to \quad \text{ur-proper-p-statep}\,(\text{p-or-bool-step}\,(\text{p-current-instruction}\,(p),\ p))$

THEOREM: and-bool-preserves-proper-p-statep
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land \quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'and-bool})$
$\land \quad \text{proper-p-statep}\,(p)$
$\land \quad \text{p-and-bool-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to \quad \text{ur-proper-p-statep}\,(\text{p-and-bool-step}\,(\text{p-current-instruction}\,(p),\ p))$

THEOREM: not-bool-preserves-proper-p-statep
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**not-bool**)
 ∧ proper-p-statep (p)
 ∧ p-not-bool-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-not-bool-step (p-current-instruction (p), p))

THEOREM: jump-if-temp-stk-full-preserves-proper-p-statep
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**jump-if-temp-stk-full**)
 ∧ proper-p-statep (p)
 ∧ p-jump-if-temp-stk-full-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-jump-if-temp-stk-full-step (p-current-instruction (p),
p))

THEOREM: jump-if-temp-stk-empty-preserves-proper-p-statep
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**jump-if-temp-stk-empty**)
 ∧ proper-p-statep (p)
 ∧ p-jump-if-temp-stk-empty-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-jump-if-temp-stk-empty-step (p-current-instruction (p),
p))

THEOREM: test-nat-and-jump-preserves-proper-p-statep
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**test-nat-and-jump**)
 ∧ proper-p-statep (p)
 ∧ p-test-nat-and-jump-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-test-nat-and-jump-step (p-current-instruction (p),
p))

THEOREM: test-int-and-jump-preserves-proper-p-statep
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**test-int-and-jump**)
 ∧ proper-p-statep (p)
 ∧ p-test-int-and-jump-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-test-int-and-jump-step (p-current-instruction (p),
p))

THEOREM: test-bool-and-jump-preserves-proper-p-statep
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**test-bool-and-jump**)
 ∧ proper-p-statep (p)
 ∧ p-test-bool-and-jump-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-test-bool-and-jump-step (p-current-instruction (p),
p))

THEOREM: test-bitv-and-jump-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'test-bitv-and-jump`)
 ∧ proper-p-statep (p)
 ∧ p-test-bitv-and-jump-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-test-bitv-and-jump-step (p-current-instruction (p), p))

```
; Since pop-locn uses put-value-indirect to modify the control
; stack, we must know:
```

THEOREM: proper-p-alistp-put-value-indirect
(proper-p-alistp (*alist*, p) ∧ (n < length (*alist*)) ∧ p-objectp (*val*, p))
→ proper-p-alistp (put-value-indirect (*val*, n, *alist*), p)

THEOREM: strip-cars-put-value-indirect
strip-cars (put-value-indirect (*val*, n, *alist*)) = strip-cars (*alist*)

```
; Note:  below we prevent p-ctrl-stk-size from expanding -- either
; via its defn or the -cons-p-frame lemma proved during the call
; owc work.  This allows the lemma we proved during pop-locn
; owc to normalize the ctrl-stk-size of the new stack.
```

THEOREM: pop-locn-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'pop-locn`)
 ∧ proper-p-statep (p)
 ∧ p-pop-locn-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-pop-locn-step (p-current-instruction (p), p))

THEOREM: p-objectp-cdr-get-proper-p-alistp
(proper-p-alistp (*alist*, p) ∧ (n < length (*alist*)))
→ p-objectp (cdr (get (n, *alist*)), p)

THEOREM: locn-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'locn`)
 ∧ proper-p-statep (p)
 ∧ p-locn-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-locn-step (p-current-instruction (p), p))

```
; The legality of the pc computed by a jump-case depends on the
; all-find-labelp check:
```

THEOREM: all-find-labelp-implies-find-labelp-get
(all-find-labelp $(lst,\ body) \wedge (n <$ length $(lst)))$
$\rightarrow$ find-labelp (get $(n,\ lst),\ body)$

THEOREM: jump-case-preserves-proper-p-statep
((p-psw $(p) =$ `'run`)
$\wedge$ (car (p-current-instruction $(p)) =$ `'jump-case`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-jump-case-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ ur-proper-p-statep (p-jump-case-step (p-current-instruction $(p),\ p)$)

```
; So now we begin call.

; First we establish that the body of the called subroutine is a listp.
```

THEOREM: listp-program-body-generalized{special}
(proper-p-prog-segmentp $(segment,\ p) \wedge$ definedp $(name,\ segment))$
$\rightarrow$ listp (program-body (assoc $(name,\ segment)))$

THEOREM: listp-program-body{special}
(proper-p-statep1 $(p) \wedge$ definedp $(name,$ p-prog-segment $(p)))$
$\rightarrow$ listp (program-body (assoc $(name,$ p-prog-segment $(p))))$

```
; Next we have to know that the bindings in the new frame is proper.
; We take it in steps.  First, we divide it into the section of formals
; and the section of locals:
```

THEOREM: proper-p-alistp-append
properp (a)
$\rightarrow$ (proper-p-alistp (append $(a,\ b),\ p)$
$=$ (proper-p-alistp $(a,\ p) \wedge$ proper-p-alistp $(b,\ p)))$

```
; In the application in question, a above is a call of pairlist.  It is
; properp:

; A lemma properp-pairlist exists in fm9001.events, but it did not
; originally.  We accommodate this change by introducing the "my-"
; prefix below.
```

THEOREM: my-properp-pairlist
properp (pairlist $(a,\ b)$)

```
; So after the rule is applied, we have to prove two proper-p-alistps.
; The first is for the pairlist of the formals to the actuals.
; The second is for the initialization of the temp vars.

; The first is handled by:
```

THEOREM: proper-p-alistp-pairlist
(all-litatoms ($formals$)
 $\wedge$ all-p-objectps ($actuals$, p)
 $\wedge$ (length ($formals$) = length ($actuals$)))
$\rightarrow$ proper-p-alistp (pairlist ($formals$, $actuals$), p)

```
; where the hypotheses above are relieved as follows.  The all-litatoms
; of the formals is derived from proper-p-prog-segmentp:
```

THEOREM: all-litatoms-formal-vars-generalized
(proper-p-prog-segmentp ($segment$, p) $\wedge$ definedp ($name$, $segment$))
$\rightarrow$ all-litatoms (formal-vars (assoc ($name$, $segment$)))

THEOREM: all-litatoms-formal-vars
(proper-p-prog-segmentp (p-prog-segment (p), p)
 $\wedge$ definedp ($name$, p-prog-segment (p)))
$\rightarrow$ all-litatoms (formal-vars (assoc ($name$, p-prog-segment (p))))

```
; The all-p-objectps of the actuals is a little more interesting.  The
; actuals in question are the reverse of the first-n elements of temp-stk.
; We eliminate the reverse with:
```

THEOREM: all-p-objectps-append
properp (a)
$\rightarrow$ (all-p-objectps (append (a, b), p)
 = (all-p-objectps (a, p) $\wedge$ all-p-objectps (b, p)))

THEOREM: all-p-objectps-rev
properp (lst) $\rightarrow$ (all-p-objectps (rev (lst), p) = all-p-objectps (lst, p))

```
; where we know:
```

THEOREM: properp-first-n
properp (first-n (n, lst))

```
; Then we get that the first-n elements of the temp stack are p-objects
; by:
```

THEOREM: all-p-objectps-first-n
(proper-p-temp-stkp ($temp\text{-}stk$, p) $\wedge$ (length ($temp\text{-}stk$) $\not<$ n))
$\rightarrow$ all-p-objectps (first-n (n, $temp\text{-}stk$), p)

```
; The third hypothesis of the pairlist lemma above is that the lengths of
; the formals and actuals are the same.  That follows from what we've
; already proved.

; Now we move to the second proper-p-alistp task, namely, for the
; initialization of the temp vars.  That is handled by:
```

THEOREM: proper-p-prog-segmentp-implies-proper-p-temp-var-dclsp
(proper-p-prog-segmentp ($segment$, p) $\wedge$ definedp ($name$, $segment$))
$\rightarrow$ proper-p-temp-var-dclsp (temp-var-dcls (assoc ($name$, $segment$)), p)

THEOREM: proper-p-alistp-pair-temps-with-initial-values
proper-p-temp-var-dclsp ($var\text{-}dcls$, p)
$\rightarrow$ proper-p-alistp (pair-temps-with-initial-values ($var\text{-}dcls$), p)

```
; That takes care of the proper-p-alistp of the new bindings.

; Next we have to prove that the strip-cars of the new bindings
; is correct.  We start by distributing the strip-cars over the
; append
```

THEOREM: strip-cars-append
strip-cars (append (a, b)) = append (strip-cars (a), strip-cars (b))

```
; Of course, a and b are as discussed above.  So for the a side we need:
```

THEOREM: strip-cars-pairlist
properp ($formals$) $\rightarrow$ (strip-cars (pairlist ($formals$, $actuals$)) = $formals$)

```
; where we know
```

THEOREM: all-litatoms-implies-properp
all-litatoms (lst) $\rightarrow$ properp (lst)

```
; and we've already set up the machinery to show that the formals are
; all-litatoms.

; For the b side, its
```

THEOREM: strip-cars-pair-temps-with-initial-values
strip-cars (pair-temps-with-initial-values (lst)) = strip-cars (lst)

```
; That completes the work for the strip-cars.

; The next part of the proper-p-statep check is that the temp-stk
; is proper.  We did the work for that with POP*.

; And that is all we need for call.
```

THEOREM: call-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'call`)
 ∧ proper-p-statep (p)
 ∧ p-call-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-call-step (p-current-instruction (p), p))

```
; Note:  The hints above are identical to those for the call owc
; proof.  The entire list of preservation lemmas was generated
; mechanically, hints and all, from the owc list.

; The call work also takes care of pop-call.
```

THEOREM: pop-call-preserves-proper-p-statep
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'pop-call`)
 ∧ proper-p-statep (p)
 ∧ p-pop-call-okp (p-current-instruction (p), p))
→ ur-proper-p-statep (p-pop-call-step (p-current-instruction (p), p))

```
; The ret case splits into two parts:  a top level ret or a ret inside
; a subroutine.  In both cases the problem is to prove proper-p-framep
; of the newly exposed frame.  If we leave proper-p-framep enabled,
; the proof of the top-level return is trivial because the newly exposed
; frame IS the frame described by proper-p-statep1, where proper-p-framep
; is opened up.  But the proof of the interior return becomes complicated
; because proper-p-framep expands after we pop the stack.  We therefore
```

```
; disable proper-p-framep in our proof and first prove:

; In the lemma below we need a little more than proper-p-statep1,
; because the arithmetic part of proper-p-statep was not put into
; proper-p-statep1.
```

THEOREM: proper-p-statep1-implies-proper-p-framep

$$
\begin{array}{l}
(\text{proper-p-statep1}\,(p) \\
\;\wedge\quad (\text{cdr}\,(\text{untag}\,(\text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))) \in \mathbf{N}) \\
\;\wedge\quad (\text{cdr}\,(\text{untag}\,(\text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))) \\
\qquad\quad < \quad \text{length}\,(\text{program-body}\,(\text{assoc}\,(\text{car}\,(\text{untag}\,(\text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-prog-segment}\,(p)))))) \\
\to\quad \text{proper-p-framep}\,(\text{car}\,(\text{p-ctrl-stk}\,(p)),\ \text{car}\,(\text{untag}\,(\text{p-pc}\,(p))),\ p)
\end{array}
$$

```
; In the case of an interior return, it suffices to open up
; proper-p-ctrl-stkp to see that the newly exposed frame is ok
; and that the rest of the stack is proper.  But proper-p-ctrl-stkp
; is hidden in proper-p-statep1.  So I prove two backchaining
; lemmas:
```

THEOREM: proper-p-ctrl-stkp-implies-proper-p-framep

$$
\begin{array}{l}
(\text{proper-p-ctrl-stkp}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)), \\
\qquad\qquad\qquad\quad \text{car}\,(\text{untag}\,(\text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))), \\
\qquad\qquad\qquad\quad p) \\
\;\wedge\quad \text{listp}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)))) \\
\to\quad \text{proper-p-framep}\,(\text{cadr}\,(\text{p-ctrl-stk}\,(p)), \\
\qquad\qquad\qquad\qquad \text{car}\,(\text{untag}\,(\text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))), \\
\qquad\qquad\qquad\qquad p)
\end{array}
$$

THEOREM: proper-p-ctrl-stkp-implies-proper-p-ctrl-stkp

$$
\begin{array}{l}
(\text{proper-p-ctrl-stkp}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)), \\
\qquad\qquad\qquad\quad \text{car}\,(\text{untag}\,(\text{ret-pc}\,(\text{car}\,(\text{p-ctrl-stk}\,(p))))), \\
\qquad\qquad\qquad\quad p) \\
\;\wedge\quad \text{listp}\,(\text{cdr}\,(\text{p-ctrl-stk}\,(p)))) \\
\to\quad \text{proper-p-ctrl-stkp}\,(\text{cddr}\,(\text{p-ctrl-stk}\,(p)), \\
\qquad\qquad\qquad\qquad \text{car}\,(\text{untag}\,(\text{ret-pc}\,(\text{cadr}\,(\text{p-ctrl-stk}\,(p))))), \\
\qquad\qquad\qquad\qquad p)
\end{array}
$$

THEOREM: ret-preserves-proper-p-statep

$$
\begin{array}{l}
((\text{p-psw}\,(p) = \texttt{'run}) \\
\;\wedge\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'ret}) \\
\;\wedge\quad \text{proper-p-statep}\,(p) \\
\;\wedge\quad \text{p-ret-okp}\,(\text{p-current-instruction}\,(p),\ p)) \\
\to\quad \text{ur-proper-p-statep}\,(\text{p-ret-step}\,(\text{p-current-instruction}\,(p),\ p))
\end{array}
$$

```
; I had forgotten to handle the lemmas of the form:

; (p-xxx-okp p) -> (r-xxx-okp (p->r p)).

; So I'll do them now.  Once I tried doing them up
; there, where I had their -step correspondents, but
; I found there were several useful rules developed during
; the proper-p-state preservation lemmas.
```

THEOREM: p-no-op-okp-implies-r-no-op-okp
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'no-op)
 ∧ proper-p-statep (p)
 ∧ p-no-op-okp (p-current-instruction (p), p))
→ r-no-op-okp (p-current-instruction (p), p->r (p))

THEOREM: p-push-global-okp-implies-r-push-global-okp
((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'push-global)
 ∧ proper-p-statep (p)
 ∧ p-push-global-okp (p-current-instruction (p), p))
→ r-push-global-okp (p-current-instruction (p), p->r (p))

EVENT: Enable empty-stkp.

```
; We need to know that every p-objectp is an r-usr-data-objectp
```

THEOREM: p-objectp-is-r-usr-data-typep
p-objectp (x, p) → r-usr-data-typep (type (x))

```
; The lemma above isn't very useful as a rule because of the free
; var.  We take care of that for the instance we need right now:
```

THEOREM: r-usr-data-typep-top
(proper-p-statep1 (p) ∧ listp (p-temp-stk (p)))
→ r-usr-data-typep (type (car (p-temp-stk (p))))

THEOREM: r-usr-data-typep-top1
(proper-p-statep1 (p) ∧ listp (cdr (p-temp-stk (p))))
→ r-usr-data-typep (type (cadr (p-temp-stk (p))))

THEOREM: r-usr-data-typep-top2
(proper-p-statep1 (p) $\wedge$ listp (cddr (p-temp-stk (p))))
$\rightarrow$ r-usr-data-typep (type (caddr (p-temp-stk (p))))

```
; That was the easy part, about the type tag.  Now we do the
; nitty-gritty part.
```

THEOREM: p-objectp-implies-r-objectp
(p-objectp (x, p) $\wedge$ same-signature (*r-usr-data-segment*, p-data-segment (p)))
$\rightarrow$ r-objectp (x,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,
r-y,
r-c-flg,
r-v-flg,
r-n-flg,
r-z-flg,
p-prog-segment (p),
r-usr-data-segment,
r-sys-data-segment,
p-word-size (p),
r-psw))

THEOREM: p-pop-global-okp-implies-r-pop-global-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'pop-global`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-pop-global-okp (p-current-instruction (p), p))
$\rightarrow$ r-pop-global-okp (p-current-instruction (p), p->r (p))

```
; To get the transition for push-local we need to know a little more
; about the size of the control stack -- recall that r-push-local-okp
; checks that several incremented stack pointers are non-empty-stkp.
; The idea is to show that the position from which we are fetching
; -- which is computed by looking up the posiition of the the var -- is not
; at the (NAT 0) marking the end of the stack.
```

THEOREM: lessp-length-bindings
listp (*ctrl-stk*)
$\rightarrow$ (length (bindings (top (*ctrl-stk*))) $<$ p-ctrl-stk-size (*ctrl-stk*))

THEOREM: lessp-length-bindings-p-ctrl-stk
proper-p-statep1 (p)
$\rightarrow$ ((length (formal-vars (assoc (car (untag (p-pc (p))), p-prog-segment (p)))))
+ length (temp-var-dcls (assoc (car (untag (p-pc (p)))),
p-prog-segment (p))))))
$<$ p-ctrl-stk-size (p-ctrl-stk (p)))

THEOREM: p-push-local-okp-implies-r-push-local-okp
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'push-local`)
$\land$ proper-p-statep (p)
$\land$ p-push-local-okp (p-current-instruction (p), p))
$\rightarrow$ r-push-local-okp (p-current-instruction (p), p->r (p))

THEOREM: p-pop-local-okp-implies-r-pop-local-okp
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'pop-local`)
$\land$ proper-p-statep (p)
$\land$ p-pop-local-okp (p-current-instruction (p), p))
$\rightarrow$ r-pop-local-okp (p-current-instruction (p), p->r (p))

THEOREM: p-deposit-okp-implies-r-deposit-okp
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'deposit`)
$\land$ proper-p-statep (p)
$\land$ p-deposit-okp (p-current-instruction (p), p))
$\rightarrow$ r-deposit-okp (p-current-instruction (p), p->r (p))

THEOREM: p-add1-nat-okp-implies-r-add1-nat-okp
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add1-nat`)
$\land$ proper-p-statep (p)
$\land$ p-add1-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-add1-nat-okp (p-current-instruction (p), p->r (p))

THEOREM: p-add-addr-okp-implies-r-add-addr-okp
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add-addr`)
$\land$ proper-p-statep (p)
$\land$ p-add-addr-okp (p-current-instruction (p), p))
$\rightarrow$ r-add-addr-okp (p-current-instruction (p), p->r (p))

THEOREM: p-add-nat-with-carry-okp-implies-r-add-nat-with-carry-okp
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add-nat-with-carry`)

$\wedge$ proper-p-statep (p)
$\wedge$ p-add-nat-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ r-add-nat-with-carry-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-mult2-nat-with-carry-out-okp-implies-r-mult2-nat-with-carry-out-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'mult2-nat-with-carry-out`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-mult2-nat-with-carry-out-okp (p-current-instruction (p), p))
$\rightarrow$ r-mult2-nat-with-carry-out-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-fetch-temp-stk-okp-implies-r-fetch-temp-stk-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'fetch-temp-stk`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-fetch-temp-stk-okp (p-current-instruction (p), p))
$\rightarrow$ r-fetch-temp-stk-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-deposit-temp-stk-okp-implies-r-deposit-temp-stk-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'deposit-temp-stk`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-deposit-temp-stk-okp (p-current-instruction (p), p))
$\rightarrow$ r-deposit-temp-stk-okp (p-current-instruction (p), p->r (p)))

```
; To prove the next one we need to know that small-naturalp, as
; checked in proper-p-pop*-instructionp, implies numberp, as
; checked in r-pop*-okp.  We just enable small-naturalp
```

THEOREM: p-pop*-okp-implies-r-pop*-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'pop*`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-pop*-okp (p-current-instruction (p), p))
$\rightarrow$ r-pop*-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-popn-okp-implies-r-popn-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'popn`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-popn-okp (p-current-instruction (p), p))
$\rightarrow$ r-popn-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-set-local-okp-implies-r-set-local-okp

$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'set-local})$
$\wedge$ $\text{proper-p-statep}\,(p)$
$\wedge$ $\text{p-set-local-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow$ $\text{r-set-local-okp}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))$

THEOREM: p-set-global-okp-implies-r-set-global-okp
$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'set-global})$
$\wedge$ $\text{proper-p-statep}\,(p)$
$\wedge$ $\text{p-set-global-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow$ $\text{r-set-global-okp}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))$

THEOREM: p-push-constant-okp-implies-r-push-constant-okp
$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'push-constant})$
$\wedge$ $\text{proper-p-statep}\,(p)$
$\wedge$ $\text{p-push-constant-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow$ $\text{r-push-constant-okp}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))$

THEOREM: p-push-ctrl-stk-free-size-okp-implies-r-push-ctrl-stk-free-size-okp
$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'push-ctrl-stk-free-size})$
$\wedge$ $\text{proper-p-statep}\,(p)$
$\wedge$ $\text{p-push-ctrl-stk-free-size-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow$ $\text{r-push-ctrl-stk-free-size-okp}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))$

THEOREM: p-push-temp-stk-free-size-okp-implies-r-push-temp-stk-free-size-okp
$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'push-temp-stk-free-size})$
$\wedge$ $\text{proper-p-statep}\,(p)$
$\wedge$ $\text{p-push-temp-stk-free-size-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow$ $\text{r-push-temp-stk-free-size-okp}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))$

```
; Again, we enable small-naturalp to get the numberp check at the r-level.
```

THEOREM: p-push-temp-stk-index-okp-implies-r-push-temp-stk-index-okp
$((\text{p-psw}\,(p) = \texttt{'run})$
$\wedge$ $(\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'push-temp-stk-index})$
$\wedge$ $\text{proper-p-statep}\,(p)$
$\wedge$ $\text{p-push-temp-stk-index-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\rightarrow$ $\text{r-push-temp-stk-index-okp}\,(\text{p-current-instruction}\,(p),\ \text{p->r}\,(p))$

THEOREM: p-pop-okp-implies-r-pop-okp

((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'pop`)
 ∧ proper-p-statep (p)
 ∧ p-pop-okp (p-current-instruction (p), p))
→ r-pop-okp (p-current-instruction (p), p->r (p))

THEOREM: p-jump-okp-implies-r-jump-okp
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'jump`)
 ∧ proper-p-statep (p)
 ∧ p-jump-okp (p-current-instruction (p), p))
→ r-jump-okp (p-current-instruction (p), p->r (p))

THEOREM: p-pushj-okp-implies-r-pushj-okp
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'pushj`)
 ∧ proper-p-statep (p)
 ∧ p-pushj-okp (p-current-instruction (p), p))
→ r-pushj-okp (p-current-instruction (p), p->r (p))

THEOREM: p-popj-okp-implies-r-popj-okp
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'popj`)
 ∧ proper-p-statep (p)
 ∧ p-popj-okp (p-current-instruction (p), p))
→ r-popj-okp (p-current-instruction (p), p->r (p))

THEOREM: p-sub-addr-okp-implies-r-sub-addr-okp
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'sub-addr`)
 ∧ proper-p-statep (p)
 ∧ p-sub-addr-okp (p-current-instruction (p), p))
→ r-sub-addr-okp (p-current-instruction (p), p->r (p))

THEOREM: p-eq-okp-implies-r-eq-okp
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'eq`)
 ∧ proper-p-statep (p)
 ∧ p-eq-okp (p-current-instruction (p), p))
→ r-eq-okp (p-current-instruction (p), p->r (p))

THEOREM: p-lt-addr-okp-implies-r-lt-addr-okp
((p-psw (p) = `'run`)
 ∧ (car (p-current-instruction (p)) = `'lt-addr`)
 ∧ proper-p-statep (p)
 ∧ p-lt-addr-okp (p-current-instruction (p), p))
→ r-lt-addr-okp (p-current-instruction (p), p->r (p))

THEOREM: p-fetch-okp-implies-r-fetch-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'fetch`)
 $\land$ proper-p-statep (p)
 $\land$ p-fetch-okp (p-current-instruction (p), p))
$\rightarrow$ r-fetch-okp (p-current-instruction (p), p->r (p))

THEOREM: p-add-int-okp-implies-r-add-int-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'add-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-add-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-add-int-okp (p-current-instruction (p), p->r (p))

THEOREM: p-add-int-with-carry-okp-implies-r-add-int-with-carry-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'add-int-with-carry`)
 $\land$ proper-p-statep (p)
 $\land$ p-add-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ r-add-int-with-carry-okp (p-current-instruction (p), p->r (p))

THEOREM: p-add1-int-okp-implies-r-add1-int-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'add1-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-add1-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-add1-int-okp (p-current-instruction (p), p->r (p))

THEOREM: p-sub-int-okp-implies-r-sub-int-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-sub-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-sub-int-okp (p-current-instruction (p), p->r (p))

THEOREM: p-sub-int-with-carry-okp-implies-r-sub-int-with-carry-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub-int-with-carry`)
 $\land$ proper-p-statep (p)
 $\land$ p-sub-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ r-sub-int-with-carry-okp (p-current-instruction (p), p->r (p))

THEOREM: p-sub1-int-okp-implies-r-sub1-int-okp
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub1-int`)

$\wedge$ proper-p-statep (p)
$\wedge$ p-sub1-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-sub1-int-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-neg-int-okp-implies-r-neg-int-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'neg-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-neg-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-neg-int-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-lt-int-okp-implies-r-lt-int-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'lt-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-lt-int-okp (p-current-instruction (p), p))
$\rightarrow$ r-lt-int-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-int-to-nat-okp-implies-r-int-to-nat-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'int-to-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-int-to-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-int-to-nat-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-add-nat-okp-implies-r-add-nat-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'add-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-add-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-add-nat-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-sub-nat-okp-implies-r-sub-nat-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'sub-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-sub-nat-okp (p-current-instruction (p), p))
$\rightarrow$ r-sub-nat-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-sub-nat-with-carry-okp-implies-r-sub-nat-with-carry-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'sub-nat-with-carry`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-sub-nat-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ r-sub-nat-with-carry-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-sub1-nat-okp-implies-r-sub1-nat-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'sub1-nat**)
 ∧ proper-p-statep (p)
 ∧ p-sub1-nat-okp (p-current-instruction (p), p))
→ r-sub1-nat-okp (p-current-instruction (p), p->r (p))

THEOREM: p-lt-nat-okp-implies-r-lt-nat-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'lt-nat**)
 ∧ proper-p-statep (p)
 ∧ p-lt-nat-okp (p-current-instruction (p), p))
→ r-lt-nat-okp (p-current-instruction (p), p->r (p))

THEOREM: p-mult2-nat-okp-implies-r-mult2-nat-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'mult2-nat**)
 ∧ proper-p-statep (p)
 ∧ p-mult2-nat-okp (p-current-instruction (p), p))
→ r-mult2-nat-okp (p-current-instruction (p), p->r (p))

THEOREM: p-div2-nat-okp-implies-r-div2-nat-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'div2-nat**)
 ∧ proper-p-statep (p)
 ∧ p-div2-nat-okp (p-current-instruction (p), p))
→ r-div2-nat-okp (p-current-instruction (p), p->r (p))

THEOREM: p-or-bitv-okp-implies-r-or-bitv-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'or-bitv**)
 ∧ proper-p-statep (p)
 ∧ p-or-bitv-okp (p-current-instruction (p), p))
→ r-or-bitv-okp (p-current-instruction (p), p->r (p))

THEOREM: p-and-bitv-okp-implies-r-and-bitv-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'and-bitv**)
 ∧ proper-p-statep (p)
 ∧ p-and-bitv-okp (p-current-instruction (p), p))
→ r-and-bitv-okp (p-current-instruction (p), p->r (p))

THEOREM: p-not-bitv-okp-implies-r-not-bitv-okp
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'not-bitv**)

$\wedge$ proper-p-statep (p)
$\wedge$ p-not-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-not-bitv-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-xor-bitv-okp-implies-r-xor-bitv-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'xor-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-xor-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-xor-bitv-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-rsh-bitv-okp-implies-r-rsh-bitv-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'rsh-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-rsh-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-rsh-bitv-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-lsh-bitv-okp-implies-r-lsh-bitv-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'lsh-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-lsh-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ r-lsh-bitv-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-or-bool-okp-implies-r-or-bool-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'or-bool`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-or-bool-okp (p-current-instruction (p), p))
$\rightarrow$ r-or-bool-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-and-bool-okp-implies-r-and-bool-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'and-bool`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-and-bool-okp (p-current-instruction (p), p))
$\rightarrow$ r-and-bool-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-not-bool-okp-implies-r-not-bool-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'not-bool`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-not-bool-okp (p-current-instruction (p), p))
$\rightarrow$ r-not-bool-okp (p-current-instruction (p), p->r (p)))

THEOREM: p-jump-if-temp-stk-full-okp-implies-r-jump-if-temp-stk-full-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'jump-if-temp-stk-full**)
∧ proper-p-statep (p)
∧ p-jump-if-temp-stk-full-okp (p-current-instruction (p), p))
→ r-jump-if-temp-stk-full-okp (p-current-instruction (p), p->r (p))

THEOREM: p-jump-if-temp-stk-empty-okp-implies-r-jump-if-temp-stk-empty-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'jump-if-temp-stk-empty**)
∧ proper-p-statep (p)
∧ p-jump-if-temp-stk-empty-okp (p-current-instruction (p), p))
→ r-jump-if-temp-stk-empty-okp (p-current-instruction (p), p->r (p))

THEOREM: p-test-nat-and-jump-okp-implies-r-test-nat-and-jump-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'test-nat-and-jump**)
∧ proper-p-statep (p)
∧ p-test-nat-and-jump-okp (p-current-instruction (p), p))
→ r-test-nat-and-jump-okp (p-current-instruction (p), p->r (p))

THEOREM: p-test-int-and-jump-okp-implies-r-test-int-and-jump-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'test-int-and-jump**)
∧ proper-p-statep (p)
∧ p-test-int-and-jump-okp (p-current-instruction (p), p))
→ r-test-int-and-jump-okp (p-current-instruction (p), p->r (p))

THEOREM: p-test-bool-and-jump-okp-implies-r-test-bool-and-jump-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'test-bool-and-jump**)
∧ proper-p-statep (p)
∧ p-test-bool-and-jump-okp (p-current-instruction (p), p))
→ r-test-bool-and-jump-okp (p-current-instruction (p), p->r (p))

THEOREM: p-test-bitv-and-jump-okp-implies-r-test-bitv-and-jump-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'test-bitv-and-jump**)
∧ proper-p-statep (p)
∧ p-test-bitv-and-jump-okp (p-current-instruction (p), p))
→ r-test-bitv-and-jump-okp (p-current-instruction (p), p->r (p))

THEOREM: p-pop-locn-okp-implies-r-pop-locn-okp
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'pop-locn**)

∧ proper-p-statep (p)
∧ p-pop-locn-okp (p-current-instruction (p), p))
→ r-pop-locn-okp (p-current-instruction (p), p->r (p))

THEOREM: p-locn-okp-implies-r-locn-okp
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'locn`)
∧ proper-p-statep (p)
∧ p-locn-okp (p-current-instruction (p), p))
→ r-locn-okp (p-current-instruction (p), p->r (p))

THEOREM: p-jump-case-okp-implies-r-jump-case-okp
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'jump-case`)
∧ proper-p-statep (p)
∧ p-jump-case-okp (p-current-instruction (p), p))
→ r-jump-case-okp (p-current-instruction (p), p->r (p))

THEOREM: proper-p-temp-var-dclsp-implies-all-p-objectps
proper-p-temp-var-dclsp (*temp-var-dcls*, p)
→ all-p-objectps (strip-cadrs (*temp-var-dcls*), p)

THEOREM: all-p-objectps-implies-all-r-objectps
(all-p-objectps (*lst*, p)
∧ same-signature (*r-usr-data-segment*, p-data-segment (p)))
→ all-r-objectps (*lst*,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,
r-y,
r-c-flg,
r-v-flg,
r-n-flg,
r-z-flg,
p-prog-segment (p),
r-usr-data-segment,
r-sys-data-segment,
p-word-size (p),
r-psw))

THEOREM: p-call-okp-implies-r-call-okp
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'call`)

$\wedge$ proper-p-statep (p)
$\wedge$ p-call-okp (p-current-instruction (p), p))
$\rightarrow$ r-call-okp (p-current-instruction (p), p->r (p))

THEOREM: p-pop-call-okp-implies-r-pop-call-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'pop-call`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-pop-call-okp (p-current-instruction (p), p))
$\rightarrow$ r-pop-call-okp (p-current-instruction (p), p->r (p))

THEOREM: p-ret-okp-implies-r-ret-okp
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'ret`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-ret-okp (p-current-instruction (p), p))
$\rightarrow$ r-ret-okp (p-current-instruction (p), p->r (p))

```
; We now turn to the preservation of r-equal.  Our basic strategy is to
; disable r-equal after proving rewrite rules that let us convert
; terms involving r2 to terms involving r1 under the hyp that r2 is
; r-equal to r1.  Then we will force open the r-equal in the conclusion
; of our theorems by an r-equal-r-state lemma below.  We begin with
; the very basic r-equal properties:
```

THEOREM: r-equal-properties
r-equal ($r2$, $r1$)
$\rightarrow$ (r-statep ($r2$)
    $\wedge$ r-statep ($r1$)
    $\wedge$ (r-pc ($r2$) = r-pc ($r1$))
    $\wedge$ (r-cfp ($r2$) = r-cfp ($r1$))
    $\wedge$ (r-csp ($r2$) = r-csp ($r1$))
    $\wedge$ (r-tsp ($r2$) = r-tsp ($r1$))
    $\wedge$ (r-prog-segment ($r2$) = r-prog-segment ($r1$))
    $\wedge$ (r-usr-data-segment ($r2$) = r-usr-data-segment ($r1$))
    $\wedge$ stk-equal (r-csp ($r1$),
            r-sys-data-segment ($r2$),
            r-sys-data-segment ($r1$))
    $\wedge$ stk-equal (r-tsp ($r1$),
            r-sys-data-segment ($r2$),
            r-sys-data-segment ($r1$))
    $\wedge$ same-signature (r-sys-data-segment ($r2$), r-sys-data-segment ($r1$))
    $\wedge$ (r-word-size ($r2$) = r-word-size ($r1$))
    $\wedge$ (r-psw ($r2$) = r-psw ($r1$)))

THEOREM: r-equal-r-state

$$
\begin{array}{l}
\text{r-equal}\,(\text{r-state}\,(\mathit{r\text{-}pc2}, \\
\quad \mathit{r\text{-}cfp2}, \\
\quad \mathit{r\text{-}csp2}, \\
\quad \mathit{r\text{-}tsp2}, \\
\quad \mathit{r\text{-}x2}, \\
\quad \mathit{r\text{-}y2}, \\
\quad \mathit{r\text{-}c\text{-}flg2}, \\
\quad \mathit{r\text{-}v\text{-}flg2}, \\
\quad \mathit{r\text{-}n\text{-}flg2}, \\
\quad \mathit{r\text{-}z\text{-}flg2}, \\
\quad \mathit{r\text{-}prog\text{-}segment}, \\
\quad \mathit{r\text{-}usr\text{-}data\text{-}segment2}, \\
\quad \mathit{r\text{-}sys\text{-}data\text{-}segment2}, \\
\quad \mathit{r\text{-}word\text{-}size}, \\
\quad \mathit{r\text{-}psw}), \\
\quad \text{r-state}\,(\mathit{r\text{-}pc1}, \\
\quad \mathit{r\text{-}cfp1}, \\
\quad \mathit{r\text{-}csp1}, \\
\quad \mathit{r\text{-}tsp1}, \\
\quad \mathit{r\text{-}x1}, \\
\quad \mathit{r\text{-}y1}, \\
\quad \mathit{r\text{-}c\text{-}flg1}, \\
\quad \mathit{r\text{-}v\text{-}flg1}, \\
\quad \mathit{r\text{-}n\text{-}flg1}, \\
\quad \mathit{r\text{-}z\text{-}flg1}, \\
\quad \mathit{r\text{-}prog\text{-}segment}, \\
\quad \mathit{r\text{-}usr\text{-}data\text{-}segment1}, \\
\quad \mathit{r\text{-}sys\text{-}data\text{-}segment1}, \\
\quad \mathit{r\text{-}word\text{-}size}, \\
\quad \mathit{r\text{-}psw})) \\
= \quad ((\mathit{r\text{-}pc2} = \mathit{r\text{-}pc1}) \\
\quad \land \quad (\mathit{r\text{-}cfp2} = \mathit{r\text{-}cfp1}) \\
\quad \land \quad (\mathit{r\text{-}csp2} = \mathit{r\text{-}csp1}) \\
\quad \land \quad (\mathit{r\text{-}tsp2} = \mathit{r\text{-}tsp1}) \\
\quad \land \quad (\mathit{r\text{-}usr\text{-}data\text{-}segment2} = \mathit{r\text{-}usr\text{-}data\text{-}segment1}) \\
\quad \land \quad \text{stk-equal}\,(\mathit{r\text{-}csp1}, \mathit{r\text{-}sys\text{-}data\text{-}segment2}, \mathit{r\text{-}sys\text{-}data\text{-}segment1}) \\
\quad \land \quad \text{stk-equal}\,(\mathit{r\text{-}tsp1}, \mathit{r\text{-}sys\text{-}data\text{-}segment2}, \mathit{r\text{-}sys\text{-}data\text{-}segment1}) \\
\quad \land \quad \text{same-signature}\,(\mathit{r\text{-}sys\text{-}data\text{-}segment2}, \mathit{r\text{-}sys\text{-}data\text{-}segment1}))
\end{array}
$$

EVENT: Disable r-equal.

THEOREM: r-equal-implies-equal-r-current-instruction

r-equal ($r2$, $r1$)

$\rightarrow$ (r-current-instruction ($r2$) = r-current-instruction ($r1$))

```
; We adopt the same policy for handling proper-r-statep that we adopted
; at the r-i level:
```

EVENT: Enable proper-r-statep-properties.

```
; Note: Unfortunately, the above remark is wrong, as I discovered much
; later below where I have to prove
; proper-r-statep-properties{augmented}.  The two statements of
; proper-r-statep-properties (in r-i.events and p-r.events) are NOT
; the same!

; Having proved that, we will
```

EVENT: Disable proper-r-statep.

THEOREM: r-equal-implies-equal-add1-r-pcp
r-equal ($r2$, $r1$) $\rightarrow$ (add1-r-pcp ($r2$) = add1-r-pcp ($r1$))

EVENT: Disable add1-r-pcp.

```
; We need to enable empty-stkp.  The reason is that the r-equal in our
; hypothesis tells us that the tstk, say, in both r1 and r2 are stk-equal,
; but that implies the equality of the live parts only down to the empty-stk.
; It does not assure us that both stacks have the '(NAT 0) at the bottom.
; But all of our tests just insure that when we get at n in the stack,
; n is less than the length of the stack.  We now need to know that it is
; less than sub1 the length of the stack, so we're within the range checked
; by the stk-equal.  That assurance is given only by the empty-stkp.
```

EVENT: Enable empty-stkp.

THEOREM: r-equal-implies-equal-r-no-op-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'no-op`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-no-op-okp (r-current-instruction ($r1$), $r2$)
= r-no-op-okp (r-current-instruction ($r1$), $r1$))

THEOREM: no-op-preserves-r-equal

$$
\begin{aligned}
&((\text{r-psw}(r1) = \texttt{'run})\\
&\ \wedge\ \ (\text{car}(\text{r-current-instruction}(r1)) = \texttt{'no-op})\\
&\ \wedge\ \ \text{proper-r-statep}(r1,\ \textit{load-addr})\\
&\ \wedge\ \ \text{proper-r-statep}(r2,\ \textit{load-addr})\\
&\ \wedge\ \ \text{r-no-op-okp}(\text{r-current-instruction}(r1),\ r2)\\
&\ \wedge\ \ \text{r-equal}(r2,\ r1))\\
&\rightarrow\ \ \text{r-equal}(\text{r-no-op-step}(\text{r-current-instruction}(r1),\ r2),\\
&\qquad\qquad\quad \text{r-no-op-step}(\text{r-current-instruction}(r1),\ r1))
\end{aligned}
$$

THEOREM: r-equal-implies-equal-r-push-global-okp

$$
\begin{aligned}
&((\text{r-psw}(r1) = \texttt{'run})\\
&\ \wedge\ \ (\text{car}(\text{r-current-instruction}(r1)) = \texttt{'push-global})\\
&\ \wedge\ \ \text{proper-r-statep}(r1,\ \textit{load-addr})\\
&\ \wedge\ \ \text{proper-r-statep}(r2,\ \textit{load-addr})\\
&\ \wedge\ \ \text{r-equal}(r2,\ r1))\\
&\rightarrow\ \ (\text{r-push-global-okp}(\text{r-current-instruction}(r1),\ r2)\\
&\qquad =\ \ \text{r-push-global-okp}(\text{r-current-instruction}(r1),\ r1))
\end{aligned}
$$

```
; The following lemma is the key to obtaining the stk-equal on the cstk
; in cases where the cstk is not being modified.
```

THEOREM: stk-equal-cstk-put-assoc-tstk2

$$
\begin{aligned}
&(\text{car}(\text{untag}(\textit{csp})) = \texttt{'cstk})\\
&\rightarrow\ \ (\text{stk-equal}(\textit{csp},\\
&\qquad\qquad\quad \text{put-assoc}(\textit{val2},\ \texttt{'tstk},\ \textit{segment2}),\\
&\qquad\qquad\quad \text{put-assoc}(\textit{val1},\ \texttt{'tstk},\ \textit{segment1}))\\
&\qquad =\ \ \text{stk-equal}(\textit{csp},\ \textit{segment2},\ \textit{segment1}))
\end{aligned}
$$

```
; And now we turn to the key lemma for the stk-equal on the tstk.
; The lemma, stk-equal-put-assoc-push, below, is general enough to apply
; to the cstk as well, if it is being pushed.
```

THEOREM: get-put{special}

$$
\begin{aligned}
&((k \in \mathbf{N}) \wedge (n \in \mathbf{N}) \wedge (n < \text{length}(\textit{lst})))\\
&\rightarrow\ \ (\text{get}(k,\ \text{put}(\textit{val},\ n,\ \textit{lst}))\\
&\qquad =\ \ \textbf{if } k = n \textbf{ then } \textit{val}\\
&\qquad\qquad \textbf{else } \text{get}(k,\ \textit{lst}) \textbf{ endif})
\end{aligned}
$$

```
; The following lemma is the inductive part of the main theorem,
; named stk-equal-put-assoc-push, below.  The form of the main
; theorem is discussed there.
```

THEOREM: stk-equal-put-assoc-above-top

$$
\begin{array}{ll}
 & ((\mathit{name} = \mathrm{car}\,(\mathrm{untag}\,(\mathit{sp}))) \\
\land & \mathrm{definedp}\,(\mathit{name},\ \mathit{segment1}) \\
\land & \mathrm{definedp}\,(\mathit{name},\ \mathit{segment2}) \\
\land & (\mathrm{cdr}\,(\mathrm{untag}\,(\mathit{sp})) \in \mathbf{N}) \\
\land & (n \in \mathbf{N}) \\
\land & (n < \mathrm{cdr}\,(\mathrm{untag}\,(\mathit{sp}))) \\
\land & (\mathrm{length}\,(\mathit{area1}) = \mathrm{length}\,(\mathrm{cdr}\,(\mathrm{assoc}\,(\mathit{name},\ \mathit{segment1})))) \\
\land & (\mathrm{length}\,(\mathit{area2}) = \mathrm{length}\,(\mathrm{cdr}\,(\mathrm{assoc}\,(\mathit{name},\ \mathit{segment2})))) \\
\land & (n < (\mathrm{length}\,(\mathit{area1}) - 1)) \\
\land & (n < (\mathrm{length}\,(\mathit{area2}) - 1))) \\
\to & (\mathrm{stk\text{-}equal}\,(\mathit{sp}, \\
 & \qquad\qquad \mathrm{put\text{-}assoc}\,(\mathrm{put}\,(\mathit{val},\ n,\ \mathit{area2}),\ \mathit{name},\ \mathit{segment2}), \\
 & \qquad\qquad \mathrm{put\text{-}assoc}\,(\mathrm{put}\,(\mathit{val},\ n,\ \mathit{area1}),\ \mathit{name},\ \mathit{segment1})) \\
 & \quad = \quad \mathrm{stk\text{-}equal}\,(\mathit{sp}, \\
 & \qquad\qquad\qquad \mathrm{put\text{-}assoc}\,(\mathit{area2},\ \mathit{name},\ \mathit{segment2}), \\
 & \qquad\qquad\qquad \mathrm{put\text{-}assoc}\,(\mathit{area1},\ \mathit{name},\ \mathit{segment1})))
\end{array}
$$

THEOREM: equal-sub1-difference

$$(0 = ((\mathit{max} - n) - 1)) = (n \not< (\mathit{max} - 1))$$

```
; This is the main theorem for dealing with stk-equality on two
; stacks that have been pushed.  The lemma says, roughly, that if
; you are interested in stack equality from sp on down, and you
; just deposited equal things at sp, then pop and omit the deposits.

; But the lemma is more general that that suggests.  Consider
; the use of area1 and area2.  My first version of the lemma
; used (cdr (assoc name segmenti)) for areai.
; That limited the utility of the lemma to cases in which
; only one thing is pushed.  This version permits areai to be another
; put.  Observe that on the rhs of the lemma I omite the puts but I
; keep the put-assocs because I have to put the areai's into the
; segments.
```

THEOREM: stk-equal-put-assoc-push

$$
\begin{array}{ll}
 & (\mathrm{listp}\,(\mathrm{untag}\,(\mathit{sp})) \\
\land & (\mathit{name} = \mathrm{car}\,(\mathrm{untag}\,(\mathit{sp}))) \\
\land & \mathrm{definedp}\,(\mathit{name},\ \mathit{segment1}) \\
\land & \mathrm{definedp}\,(\mathit{name},\ \mathit{segment2}) \\
\land & (n = \mathrm{cdr}\,(\mathrm{untag}\,(\mathit{sp}))) \\
\land & (n \in \mathbf{N}) \\
\land & (\mathrm{length}\,(\mathit{area1}) = \mathrm{length}\,(\mathrm{cdr}\,(\mathrm{assoc}\,(\mathit{name},\ \mathit{segment1})))) \\
\land & (\mathrm{length}\,(\mathit{area2}) = \mathrm{length}\,(\mathrm{cdr}\,(\mathrm{assoc}\,(\mathit{name},\ \mathit{segment2}))))
\end{array}
$$

$\land$ (¬ empty-stkp (tag (`'sys-addr`, cons ($name$, n)), $segment1$))
$\land$ (¬ empty-stkp (tag (`'sys-addr`, cons ($name$, n)), $segment2$)))
$\to$ (stk-equal (sp,
put-assoc (put (val, n, $area2$), $name$, $segment2$),
put-assoc (put (val, n, $area1$), $name$, $segment1$))
$=$ stk-equal (pop-stk (sp),
put-assoc ($area2$, $name$, $segment2$),
put-assoc ($area1$, $name$, $segment1$)))

```
; Eventually, when all the puts are exhausted we will be putting
; the original area into the segment and the following rule will
; collapse it:
```

THEOREM: put-assoc-cdr-assoc
listp (assoc ($name$, $segment$))
$\to$ (put-assoc (cdr (assoc ($name$, $segment$)), $name$, $segment$) $=$ $segment$)

```
; However, to get the hypothesis above we need:
```

THEOREM: proper-r-statep-implies-tstkp
proper-r-statep (r, $load\text{-}addr$)
$\to$ (cdr (untag (r-tsp (r)))
$<$ length (cdr (assoc (`'tstk`, r-sys-data-segment (r)))))

THEOREM: listp-assoc-tstk
proper-r-statep (r, $load\text{-}addr$)
$\to$ listp (assoc (`'tstk`, r-sys-data-segment (r)))

```
; Finally, in the main theorem above we sometimes have to prove a
; non-empty-stkp of a pushed stack, which follows from the non-emptiness
; of the unpushed stack:
```

THEOREM: not-empty-stkp-push{special}
(¬ empty-stkp (tag (`'sys-addr`, cons ($name$, n)), $segment$))
$\to$ (¬ empty-stkp (tag (`'sys-addr`, cons ($name$, $n - 1$)), $segment$))

```
; The next lemma establishes that if we have a stk-equal at n
; we have it at all greater n.  This means that if we pop several
; things off the stack and then push one, the lemma above will take
; care of the push and the lemma below will take care of the pops.
```

THEOREM: stk-equal-extends-upwards-base-case
$((n \not< (\text{length}\,(\text{cdr}\,(\text{assoc}\,(\mathit{name},\ \mathit{segment2}))) - 1))$
$\land \quad (n \in \mathbf{N})$
$\land \quad (k \in \mathbf{N})$
$\land \quad (k \not< n))$
$\to \quad \text{stk-equal}\,(\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\mathit{name},\ k)),\ \mathit{segment2},\ \mathit{segment1})$

DEFINITION:
$\text{stk-equal-hint-5}\,(n,\ \mathit{name},\ \mathit{segment})$
$= \quad \textbf{if}\ ((\text{length}\,(\text{cdr}\,(\text{assoc}\,(\mathit{name},\ \mathit{segment}))) - n) - 1) \simeq \texttt{0}\ \ \textbf{then t}$
$\quad \textbf{else}\ \text{stk-equal-hint-5}\,(n + \texttt{1},\ \mathit{name},\ \mathit{segment})\ \textbf{endif}$

```
; The next lemma is really the key result.  However, as stated it is
; not useful as a rewrite rule because it has the free-var segment1
; picked up by the stk-equal but in our applications the stk-equal is
; enclosed inside the r-equal we keep disabled.  This pattern will be
; repeated many times in the lemma development here.
```

THEOREM: stk-equal-extends-upwards
$((n \in \mathbf{N})$
$\land \quad (k \in \mathbf{N})$
$\land \quad \text{stk-equal}\,(\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\mathit{name},\ n)),\ \mathit{segment2},\ \mathit{segment1})$
$\land \quad (k \not< n))$
$\to \quad \text{stk-equal}\,(\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\mathit{name},\ k)),\ \mathit{segment2},\ \mathit{segment1})$

EVENT: Disable stk-equal-extends-upwards-base-case.

THEOREM: proper-r-statep-implies-tag-untag
$((\text{type}\,(x) = \texttt{'sys-addr})$
$\land \quad \text{listp}\,(\text{untag}\,(x))$
$\land \quad (\text{car}\,(\text{untag}\,(x)) = \texttt{'tstk})$
$\land \quad (\text{cddr}\,(x) = \textbf{nil}))$
$\to \quad (\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\texttt{'tstk},\ \text{cdr}\,(\text{untag}\,(x)))) = x)$

```
; When this proper-r-statep-properties was proved in r-i.events it had some
; adp-name and other fns "closed".  But when it was proved in p-r.events I
; had expanded those names.  I did not notice this when I concatenated the two
; event lists and removed "duplicates"  -- i.e., I mistook the second occurrence
; of the name proper-r-statep-properties as a duplicate of the first when in fact
; the formulas were different by virtue of these non-rec fn expansions.  So here
; I prove the expanded conjuncts.
```

THEOREM: proper-r-statep-properties{augmented}
$$
\begin{array}{l}
\text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr}) \\
\rightarrow\ \ (\text{definedp}\,(\text{car}\,(\text{untag}\,(\text{r-pc}\,(r))),\ \text{r-prog-segment}\,(r)) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-pc}\,(r))) \in \mathbf{N}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-pc}\,(r))) \\
\quad\quad\quad\quad < \ \ \text{length}\,(\text{program-body}\,(\text{assoc}\,(\text{car}\,(\text{untag}\,(\text{r-pc}\,(r))), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{r-prog-segment}\,(r))))) \\
\quad\quad \land\ \ (\text{car}\,(\text{untag}\,(\text{r-cfp}\,(r))) = \texttt{'cstk}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-cfp}\,(r))) \in \mathbf{N}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-cfp}\,(r))) \\
\quad\quad\quad\quad < \ \ \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \text{r-sys-data-segment}\,(r))))) \\
\quad\quad \land\ \ (\text{car}\,(\text{untag}\,(\text{r-csp}\,(r))) = \texttt{'cstk}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-csp}\,(r))) \in \mathbf{N}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-csp}\,(r))) \\
\quad\quad\quad\quad < \ \ \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \text{r-sys-data-segment}\,(r))))) \\
\quad\quad \land\ \ (\text{car}\,(\text{untag}\,(\text{r-tsp}\,(r))) = \texttt{'tstk}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-tsp}\,(r))) \in \mathbf{N}) \\
\quad\quad \land\ \ (\text{cdr}\,(\text{untag}\,(\text{r-tsp}\,(r))) \\
\quad\quad\quad\quad < \ \ \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'tstk},\ \text{r-sys-data-segment}\,(r))))) \\
\quad\quad \land\ \ \text{definedp}\,(\texttt{'full-ctrl-stk-addr},\ \text{r-sys-data-segment}\,(r)) \\
\quad\quad \land\ \ \text{definedp}\,(\texttt{'full-temp-stk-addr},\ \text{r-sys-data-segment}\,(r)) \\
\quad\quad \land\ \ \text{definedp}\,(\texttt{'empty-temp-stk-addr},\ \text{r-sys-data-segment}\,(r)))
\end{array}
$$

THEOREM: r-equal-implies-stk-equal-extends-upwards-tstk
$$
\begin{array}{l}
(\text{r-equal}\,(\mathit{r2},\ \mathit{r1}) \\
\ \land\ \ \text{proper-r-statep}\,(\mathit{r1},\ \mathit{load\text{-}addr}) \\
\ \land\ \ \text{proper-r-statep}\,(\mathit{r2},\ \mathit{load\text{-}addr}) \\
\ \land\ \ (k \in \mathbf{N}) \\
\ \land\ \ (k \not< \text{cdr}\,(\text{untag}\,(\text{r-tsp}\,(\mathit{r1}))))) \\
\rightarrow\ \ \text{stk-equal}\,(\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\texttt{'tstk},\ k)), \\
\quad\quad\quad\quad\quad\quad \text{r-sys-data-segment}\,(\mathit{r2}), \\
\quad\quad\quad\quad\quad\quad \text{r-sys-data-segment}\,(\mathit{r1}))
\end{array}
$$

```
; So now we can do
```

THEOREM: push-global-preserves-r-equal
$$
\begin{array}{l}
((\text{r-psw}\,(\mathit{r1}) = \texttt{'run}) \\
\ \land\ \ (\text{car}\,(\text{r-current-instruction}\,(\mathit{r1})) = \texttt{'push-global}) \\
\ \land\ \ \text{proper-r-statep}\,(\mathit{r1},\ \mathit{load\text{-}addr}) \\
\ \land\ \ \text{proper-r-statep}\,(\mathit{r2},\ \mathit{load\text{-}addr}) \\
\ \land\ \ \text{r-push-global-okp}\,(\text{r-current-instruction}\,(\mathit{r1}),\ \mathit{r2}) \\
\ \land\ \ \text{r-equal}\,(\mathit{r2},\ \mathit{r1})) \\
\rightarrow\ \ \text{r-equal}\,(\text{r-push-global-step}\,(\text{r-current-instruction}\,(\mathit{r1}),\ \mathit{r2}), \\
\quad\quad\quad\quad\quad \text{r-push-global-step}\,(\text{r-current-instruction}\,(\mathit{r1}),\ \mathit{r1}))
\end{array}
$$

```
; We will keep empty-stkp disabled to cut down on the arithmetic
; used.  Of course, we need:
```

THEOREM: r-equal-implies-equal-length
r-equal ($r2$, $r1$)
$\rightarrow$ (length (cdr (assoc ($name$, r-sys-data-segment ($r2$))))
= length (cdr (assoc ($name$, r-sys-data-segment ($r1$)))))

THEOREM: r-equal-implies-equal-empty-stkp
r-equal ($r2$, $r1$)
$\rightarrow$ (empty-stkp (sp, r-sys-data-segment ($r2$))
= empty-stkp (sp, r-sys-data-segment ($r1$)))

```
; The next two lemmas establish that when we get from the active
; region of either the cstk or the tstk of two r-equal states
; we get the same result.  The key fact is the lemma immediately
; below, which is then instantiated for each stack and lifted up
; to r-equal.
```

THEOREM: stk-equal-implies-equal-get
(stk-equal (sp, $segment2$, $segment1$)
$\wedge$ ($k \in \mathbf{N}$)
$\wedge$ (cdr (untag (sp)) $\in \mathbf{N}$)
$\wedge$ ($k \not< $ cdr (untag (sp)))
$\wedge$ ($k <$ (length (cdr (assoc (car (untag (sp)), $segment2$))) $-$ 1)))
$\rightarrow$ (get (k, cdr (assoc (car (untag (sp)), $segment2$)))
= get (k, cdr (assoc (car (untag (sp)), $segment1$))))

```
; The instantiated versions of the lemma above requires that we know
; the stack pointer at which we are getting is non-empty.
```

THEOREM: r-equal-implies-equal-get-tstk
(r-equal ($r2$, $r1$)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ ($k \in \mathbf{N}$)
$\wedge$ ($\neg$ empty-stkp (tag (`'sys-addr`, cons (`'tstk`, k)),
r-sys-data-segment ($r1$)))
$\wedge$ ($k \not<$ cdr (untag (r-tsp ($r1$)))))
$\rightarrow$ (get (k, cdr (assoc (`'tstk`, r-sys-data-segment ($r2$))))
= get (k, cdr (assoc (`'tstk`, r-sys-data-segment ($r1$)))))

```
; We don't actually need the cstk version just yet, but I have proved
; it because when we get to local vars we will need it.
```

THEOREM: r-equal-implies-equal-get-cstk
(r-equal ($r2$, $r1$)
$\land$ proper-r-statep ($r1$, *load-addr*)
$\land$ ($k \in \mathbf{N}$)
$\land$ ($\neg$ empty-stkp (tag (`'sys-addr`, cons (`'cstk`, k)),
r-sys-data-segment ($r1$)))
$\land$ ($k \not< $ cdr (untag (r-csp ($r1$)))))
$\rightarrow$ (get (k, cdr (assoc (`'cstk`, r-sys-data-segment ($r2$))))
= get (k, cdr (assoc (`'cstk`, r-sys-data-segment ($r1$)))))

```
; We now disable empty-stkp.  We will undoubtedly have to prove some
; lemmas to establish the non-empty-stkp hypotheses above.
```

EVENT: Disable empty-stkp.

```
; In the case where we are getting from r-tsp we need:
```

```
; We need that the r-objects of one state are those of the other.
```

THEOREM: r-equal-implies-equal-r-objectp
r-equal ($r2$, $r1$) $\rightarrow$ (r-objectp (x, $r2$) = r-objectp (x, $r1$))

```
; We also get the rule for r-objectp-type and disable that function
; because here we do not need to know what it means, just that if
; it is true for r2 it is true for r1.
```

THEOREM: r-equal-implies-equal-r-objectp-type
r-equal ($r2$, $r1$)
$\rightarrow$ (r-objectp-type (*type*, x, $r2$) = r-objectp-type (*type*, x, $r1$))

EVENT: Disable r-objectp-type.

```
; And we'll get the rule for r-usr-data-objectp too.
```

THEOREM: r-equal-implies-equal-r-usr-data-objectp
r-equal ($r2$, $r1$) $\rightarrow$ (r-usr-data-objectp (x, $r2$) = r-usr-data-objectp (x, $r1$))

EVENT: Disable r-usr-data-objectp.

```
; Now we can also handle:
```

THEOREM: r-equal-implies-equal-r-pop-global-okp
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'pop-global**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-pop-global-okp (r-current-instruction ($r1$), $r2$)
  = r-pop-global-okp (r-current-instruction ($r1$), $r1$))

THEOREM: pop-global-preserves-r-equal
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'pop-global**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-pop-global-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
→ r-equal (r-pop-global-step (r-current-instruction ($r1$), $r2$),
  r-pop-global-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-push-local-okp
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'push-local**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-push-local-okp (r-current-instruction ($r1$), $r2$)
  = r-push-local-okp (r-current-instruction ($r1$), $r1$))

```
; To prove a stk-equal after a push we have to be able to establish
; the non-empty-stkp hypothesis for the pushed stack.  That follows
; from:
```

THEOREM: proper-r-statep-implies-non-empty-stkp-push
(proper-r-statep (r, *load-addr*) ∧ (free-stk-size (r-tsp (r)) $\not\simeq$ 0))
→ (¬ empty-stkp (tag (**'sys-addr**,
  cons (**'tstk**, cdr (untag (r-tsp (r))) − 1)),
  r-sys-data-segment (r)))

THEOREM: push-local-preserves-r-equal
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'push-local**)

$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-push-local-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-push-local-step (r-current-instruction (*r1*), *r2*),
r-push-local-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-pop-local-okp
((r-psw (*r1*) = `'run`)
$\wedge$ (car (r-current-instruction (*r1*)) = `'pop-local`)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-pop-local-okp (r-current-instruction (*r1*), *r2*)
= r-pop-local-okp (r-current-instruction (*r1*), *r1*))

```
; That we could get to here without further work is not too surprising
; considering the nature of the instructions we've learned to handle.
; That we stopped here is also not surprising: we haven't explained
; how to handle stk-equal on the cstk (except in the rare case of
; pushes onto the cstk).  What we need is the lemma that says if two
; cstks are stk-equal and you put the same thing at the same place in
; both then they are still stk-equal.  This is sort of like the
; stk-equal-put-assoc-push lemma above except (a) we don't require
; that the put be a push and because of that (b) we don't require that
; the stk-equal start its scan at the location of the push.  This
; operation, of sticking something into the middle of a stack, is of
; course only performed on the cstk; never on the tstk.
```

THEOREM: stk-equal-put-assoc-insertion
(listp (untag (*sp*))
$\wedge$ stk-equal (*sp*, *segment2*, *segment1*)
$\wedge$ (*name* = car (untag (*sp*)))
$\wedge$ definedp (*name*, *segment2*)
$\wedge$ definedp (*name*, *segment1*)
$\wedge$ (cdr (untag (*sp*)) $\in \mathbf{N}$)
$\wedge$ ($n \in \mathbf{N}$)
$\wedge$ ($n \not<$ cdr (untag (*sp*)))
$\wedge$ ($\neg$ empty-stkp (tag (`'sys-addr`, cons (*name*, *n*)), *segment2*))
$\wedge$ ($\neg$ empty-stkp (tag (`'sys-addr`, cons (*name*, *n*)), *segment1*))
$\wedge$ listp (assoc (*name*, *segment1*))
$\wedge$ listp (assoc (*name*, *segment2*)))
$\rightarrow$ stk-equal (*sp*,
put-assoc (put (*val*, *n*, cdr (assoc (*name*, *segment2*))),

$\qquad\qquad\qquad\qquad$ *name*,
$\qquad\qquad\qquad\qquad$ *segment2*),
$\qquad\qquad$ put-assoc (put (*val*, *n*, cdr (assoc (*name*, *segment1*))),
$\qquad\qquad\qquad\qquad$ *name*,
$\qquad\qquad\qquad\qquad$ *segment1*))

```
; The hypotheses above about (listp (assoc name segmenti)) will
; be relieved in our applications by:
```

THEOREM: proper-r-statep-implies-cstkp
proper-r-statep (r, *load-addr*)
$\rightarrow$ (cdr (untag (r-csp (r)))
$\quad <$ length (cdr (assoc (**'cstk**, r-sys-data-segment (r)))))

THEOREM: listp-assoc-cstk
proper-r-statep (r, *load-addr*)
$\rightarrow$ listp (assoc (**'cstk**, r-sys-data-segment (r)))

```
; The following rule is the dual of its namesake: it establishes a
; stk-equal on the tstk when the cstk has been changed.
```

THEOREM: stk-equal-tstk-put-assoc-cstk2
(car (untag (*tsp*)) = **'tstk**)
$\rightarrow$ (stk-equal (*tsp*,
$\qquad\qquad$ put-assoc (*val2*, **'cstk**, *segment2*),
$\qquad\qquad$ put-assoc (*val1*, **'cstk**, *segment1*))
$\quad =$ stk-equal (*tsp*, *segment2*, *segment1*))

```
; The following lemma is needed to relieve the hypothesis that we are
; putting into a legal position.  This would not be necessary if we
; had empty-stkp enabled.
```

THEOREM: empty-stkp-backchainer
($\neg$ empty-stkp (tag (*xxx*, cons (*name*, k)), *segment*))
$\rightarrow$ ((k < length (cdr (assoc (*name*, *segment*)))) = **t**)

THEOREM: pop-local-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'pop-local**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-pop-local-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-pop-local-step (r-current-instruction (*r1*), *r2*),
$\qquad\qquad$ r-pop-local-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-deposit-okp
((r-psw (*r1*) = **'run**)
 ∧ (car (r-current-instruction (*r1*)) = **'deposit**)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ (r-deposit-okp (r-current-instruction (*r1*), *r2*)
 = r-deposit-okp (r-current-instruction (*r1*), *r1*))

THEOREM: deposit-preserves-r-equal
((r-psw (*r1*) = **'run**)
 ∧ (car (r-current-instruction (*r1*)) = **'deposit**)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-deposit-okp (r-current-instruction (*r1*), *r2*)
 ∧ r-equal (*r2*, *r1*))
→ r-equal (r-deposit-step (r-current-instruction (*r1*), *r2*),
 r-deposit-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-add1-nat-okp
((r-psw (*r1*) = **'run**)
 ∧ (car (r-current-instruction (*r1*)) = **'add1-nat**)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ (r-add1-nat-okp (r-current-instruction (*r1*), *r2*)
 = r-add1-nat-okp (r-current-instruction (*r1*), *r1*))

THEOREM: add1-nat-preserves-r-equal
((r-psw (*r1*) = **'run**)
 ∧ (car (r-current-instruction (*r1*)) = **'add1-nat**)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-add1-nat-okp (r-current-instruction (*r1*), *r2*)
 ∧ r-equal (*r2*, *r1*))
→ r-equal (r-add1-nat-step (r-current-instruction (*r1*), *r2*),
 r-add1-nat-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-add-addr-okp
((r-psw (*r1*) = **'run**)
 ∧ (car (r-current-instruction (*r1*)) = **'add-addr**)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ (r-add-addr-okp (r-current-instruction (*r1*), *r2*)
 = r-add-addr-okp (r-current-instruction (*r1*), *r1*))

THEOREM: add-addr-preserves-r-equal
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'add-addr**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-add-addr-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))
→ r-equal (r-add-addr-step (r-current-instruction ($r1$), $r2$),
r-add-addr-step (r-current-instruction ($r1$), $r1$))

```
; The following lemma is a good example of the value of tuning
; the disable/enable strategy.  In an environment just like
; the current one except with empty-stkp enabled and the
; necessary rule that (sub1 (sub1 (difference x y))) = 0
; iff y is not lessp x-2, the following theorem took 67 seconds
; to prove and 118 seconds for io.  It now takes 3.8 to prove
; and .1 for io.  The reason is that when empty-stkp opens
; and is converted to an inequality, the linear arithmetic
; package repeatedly splits on various zerop possibilities.
```

THEOREM: r-equal-implies-equal-r-add-nat-with-carry-okp
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'add-nat-with-carry**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-equal ($r2$, $r1$))
→ (r-add-nat-with-carry-okp (r-current-instruction ($r1$), $r2$)
= r-add-nat-with-carry-okp (r-current-instruction ($r1$), $r1$))

THEOREM: add-nat-with-carry-preserves-r-equal
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'add-nat-with-carry**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-add-nat-with-carry-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))
→ r-equal (r-add-nat-with-carry-step (r-current-instruction ($r1$), $r2$),
r-add-nat-with-carry-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-mult2-nat-with-carry-out-okp
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$))
= **'mult2-nat-with-carry-out**)
∧ proper-r-statep ($r1$, *load-addr*)

$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-equal ($r2$, $r1$))
$\to$ (r-mult2-nat-with-carry-out-okp (r-current-instruction ($r1$), $r2$)
= r-mult2-nat-with-carry-out-okp (r-current-instruction ($r1$), $r1$))

THEOREM: mult2-nat-with-carry-out-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$))
= `'mult2-nat-with-carry-out`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-mult2-nat-with-carry-out-okp (r-current-instruction ($r1$), $r2$)
$\land$ r-equal ($r2$, $r1$))
$\to$ r-equal (r-mult2-nat-with-carry-out-step (r-current-instruction ($r1$), $r2$),
r-mult2-nat-with-carry-out-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-fetch-temp-stk-okp
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$)) = `'fetch-temp-stk`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-equal ($r2$, $r1$))
$\to$ (r-fetch-temp-stk-okp (r-current-instruction ($r1$), $r2$)
= r-fetch-temp-stk-okp (r-current-instruction ($r1$), $r1$))

```
; To relieve the hypothesis of R-EQUAL-IMPLIES-EQUAL-GET-TSTK we need:
```

THEOREM: not-empty-stkp-temp-stk-index
(n < (length (cdr (assoc (`'tstk`, r-sys-data-segment (r)))) − 1))
$\to$ ($\neg$ empty-stkp (tag (`'sys-addr`,
cons (`'tstk`,
((length (cdr (assoc (`'tstk`,
r-sys-data-segment (r)))) − 1)
− n) − 1)),
r-sys-data-segment (r)))

THEOREM: fetch-temp-stk-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$)) = `'fetch-temp-stk`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-fetch-temp-stk-okp (r-current-instruction ($r1$), $r2$)
$\land$ r-equal ($r2$, $r1$))
$\to$ r-equal (r-fetch-temp-stk-step (r-current-instruction ($r1$), $r2$),
r-fetch-temp-stk-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-deposit-temp-stk-okp
((r-psw ($r1$) = 'run)
∧ (car (r-current-instruction ($r1$)) = 'deposit-temp-stk)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-equal ($r2$, $r1$))
→ (r-deposit-temp-stk-okp (r-current-instruction ($r1$), $r2$)
= r-deposit-temp-stk-okp (r-current-instruction ($r1$), $r1$))

THEOREM: deposit-temp-stk-preserves-r-equal
((r-psw ($r1$) = 'run)
∧ (car (r-current-instruction ($r1$)) = 'deposit-temp-stk)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-deposit-temp-stk-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))
→ r-equal (r-deposit-temp-stk-step (r-current-instruction ($r1$), $r2$),
r-deposit-temp-stk-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-pop*-okp
((r-psw ($r1$) = 'run)
∧ (car (r-current-instruction ($r1$)) = 'pop*)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-equal ($r2$, $r1$))
→ (r-pop*-okp (r-current-instruction ($r1$), $r2$)
= r-pop*-okp (r-current-instruction ($r1$), $r1$))

THEOREM: pop*-preserves-r-equal
((r-psw ($r1$) = 'run)
∧ (car (r-current-instruction ($r1$)) = 'pop*)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-pop*-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))
→ r-equal (r-pop*-step (r-current-instruction ($r1$), $r2$),
r-pop*-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-popn-okp
((r-psw ($r1$) = 'run)
∧ (car (r-current-instruction ($r1$)) = 'popn)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-equal ($r2$, $r1$))
→ (r-popn-okp (r-current-instruction ($r1$), $r2$)
= r-popn-okp (r-current-instruction ($r1$), $r1$))

THEOREM: popn-preserves-r-equal
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'popn**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-popn-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))
→ r-equal (r-popn-step (r-current-instruction ($r1$), $r2$),
r-popn-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-set-local-okp
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'set-local**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-equal ($r2$, $r1$))
→ (r-set-local-okp (r-current-instruction ($r1$), $r2$)
= r-set-local-okp (r-current-instruction ($r1$), $r1$))

THEOREM: set-local-preserves-r-equal
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'set-local**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-set-local-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))
→ r-equal (r-set-local-step (r-current-instruction ($r1$), $r2$),
r-set-local-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-set-global-okp
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'set-global**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-equal ($r2$, $r1$))
→ (r-set-global-okp (r-current-instruction ($r1$), $r2$)
= r-set-global-okp (r-current-instruction ($r1$), $r1$))

THEOREM: set-global-preserves-r-equal
((r-psw ($r1$) = **'run**)
∧ (car (r-current-instruction ($r1$)) = **'set-global**)
∧ proper-r-statep ($r1$, *load-addr*)
∧ proper-r-statep ($r2$, *load-addr*)
∧ r-set-global-okp (r-current-instruction ($r1$), $r2$)
∧ r-equal ($r2$, $r1$))

→ r-equal (r-set-global-step (r-current-instruction (*r1*), *r2*),
r-set-global-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-push-constant-okp
((r-psw (*r1*) = 'run)
∧ (car (r-current-instruction (*r1*)) = 'push-constant)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-push-constant-okp (r-current-instruction (*r1*), *r2*)
= r-push-constant-okp (r-current-instruction (*r1*), *r1*))

THEOREM: push-constant-preserves-r-equal
((r-psw (*r1*) = 'run)
∧ (car (r-current-instruction (*r1*)) = 'push-constant)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-push-constant-okp (r-current-instruction (*r1*), *r2*)
∧ r-equal (*r2*, *r1*))
→ r-equal (r-push-constant-step (r-current-instruction (*r1*), *r2*),
r-push-constant-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-push-ctrl-stk-free-size-okp
((r-psw (*r1*) = 'run)
∧ (car (r-current-instruction (*r1*)) = 'push-ctrl-stk-free-size)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-push-ctrl-stk-free-size-okp (r-current-instruction (*r1*), *r2*)
= r-push-ctrl-stk-free-size-okp (r-current-instruction (*r1*), *r1*))

THEOREM: push-ctrl-stk-free-size-preserves-r-equal
((r-psw (*r1*) = 'run)
∧ (car (r-current-instruction (*r1*)) = 'push-ctrl-stk-free-size)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-push-ctrl-stk-free-size-okp (r-current-instruction (*r1*), *r2*)
∧ r-equal (*r2*, *r1*))
→ r-equal (r-push-ctrl-stk-free-size-step (r-current-instruction (*r1*), *r2*),
r-push-ctrl-stk-free-size-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-push-temp-stk-free-size-okp
((r-psw (*r1*) = 'run)
∧ (car (r-current-instruction (*r1*)) = 'push-temp-stk-free-size)
∧ proper-r-statep (*r1*, *load-addr*)

$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-push-temp-stk-free-size-okp (r-current-instruction (*r1*), *r2*)
  = r-push-temp-stk-free-size-okp (r-current-instruction (*r1*), *r1*))

THEOREM: push-temp-stk-free-size-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'push-temp-stk-free-size**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-push-temp-stk-free-size-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-push-temp-stk-free-size-step (r-current-instruction (*r1*), *r2*),
  r-push-temp-stk-free-size-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-push-temp-stk-index-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'push-temp-stk-index**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-push-temp-stk-index-okp (r-current-instruction (*r1*), *r2*)
  = r-push-temp-stk-index-okp (r-current-instruction (*r1*), *r1*))

THEOREM: push-temp-stk-index-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'push-temp-stk-index**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-push-temp-stk-index-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-push-temp-stk-index-step (r-current-instruction (*r1*), *r2*),
  r-push-temp-stk-index-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-pop-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'pop**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-pop-okp (r-current-instruction (*r1*), *r2*)
  = r-pop-okp (r-current-instruction (*r1*), *r1*))

THEOREM: pop-preserves-r-equal
((r-psw (*r1*) = **'run**)

$\wedge$ (car (r-current-instruction ($r1$)) = `'pop`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-pop-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-pop-step (r-current-instruction ($r1$), $r2$),
r-pop-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-jump-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'jump`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-jump-okp (r-current-instruction ($r1$), $r2$)
= r-jump-okp (r-current-instruction ($r1$), $r1$))

THEOREM: jump-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'jump`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-jump-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-jump-step (r-current-instruction ($r1$), $r2$),
r-jump-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-pushj-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'pushj`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-pushj-okp (r-current-instruction ($r1$), $r2$)
= r-pushj-okp (r-current-instruction ($r1$), $r1$))

THEOREM: pushj-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'pushj`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-pushj-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-pushj-step (r-current-instruction ($r1$), $r2$),
r-pushj-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-popj-okp
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'popj`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-popj-okp (r-current-instruction (*r1*), *r2*)
= r-popj-okp (r-current-instruction (*r1*), *r1*))

THEOREM: popj-preserves-r-equal
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'popj`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-popj-okp (r-current-instruction (*r1*), *r2*)
∧ r-equal (*r2*, *r1*))
→ r-equal (r-popj-step (r-current-instruction (*r1*), *r2*),
r-popj-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-sub-addr-okp
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'sub-addr`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-sub-addr-okp (r-current-instruction (*r1*), *r2*)
= r-sub-addr-okp (r-current-instruction (*r1*), *r1*))

THEOREM: sub-addr-preserves-r-equal
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'sub-addr`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-sub-addr-okp (r-current-instruction (*r1*), *r2*)
∧ r-equal (*r2*, *r1*))
→ r-equal (r-sub-addr-step (r-current-instruction (*r1*), *r2*),
r-sub-addr-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-eq-okp
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'eq`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-eq-okp (r-current-instruction (*r1*), *r2*)
= r-eq-okp (r-current-instruction (*r1*), *r1*))

THEOREM: eq-preserves-r-equal

$$
\begin{array}{ll}
((\text{r-psw}\,(r1) = \texttt{'run}) & \\
\ \wedge & (\text{car}\,(\text{r-current-instruction}\,(r1)) = \texttt{'eq}) \\
\ \wedge & \text{proper-r-statep}\,(r1,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{proper-r-statep}\,(r2,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{r-eq-okp}\,(\text{r-current-instruction}\,(r1),\ r2) \\
\ \wedge & \text{r-equal}\,(r2,\ r1)) \\
\to & \text{r-equal}\,(\text{r-eq-step}\,(\text{r-current-instruction}\,(r1),\ r2), \\
 & \qquad\quad \text{r-eq-step}\,(\text{r-current-instruction}\,(r1),\ r1))
\end{array}
$$

THEOREM: r-equal-implies-equal-r-lt-addr-okp

$$
\begin{array}{ll}
((\text{r-psw}\,(r1) = \texttt{'run}) & \\
\ \wedge & (\text{car}\,(\text{r-current-instruction}\,(r1)) = \texttt{'lt-addr}) \\
\ \wedge & \text{proper-r-statep}\,(r1,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{proper-r-statep}\,(r2,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{r-equal}\,(r2,\ r1)) \\
\to & (\text{r-lt-addr-okp}\,(\text{r-current-instruction}\,(r1),\ r2) \\
 & \ = \ \text{r-lt-addr-okp}\,(\text{r-current-instruction}\,(r1),\ r1))
\end{array}
$$

THEOREM: lt-addr-preserves-r-equal

$$
\begin{array}{ll}
((\text{r-psw}\,(r1) = \texttt{'run}) & \\
\ \wedge & (\text{car}\,(\text{r-current-instruction}\,(r1)) = \texttt{'lt-addr}) \\
\ \wedge & \text{proper-r-statep}\,(r1,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{proper-r-statep}\,(r2,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{r-lt-addr-okp}\,(\text{r-current-instruction}\,(r1),\ r2) \\
\ \wedge & \text{r-equal}\,(r2,\ r1)) \\
\to & \text{r-equal}\,(\text{r-lt-addr-step}\,(\text{r-current-instruction}\,(r1),\ r2), \\
 & \qquad\quad \text{r-lt-addr-step}\,(\text{r-current-instruction}\,(r1),\ r1))
\end{array}
$$

THEOREM: r-equal-implies-equal-r-fetch-okp

$$
\begin{array}{ll}
((\text{r-psw}\,(r1) = \texttt{'run}) & \\
\ \wedge & (\text{car}\,(\text{r-current-instruction}\,(r1)) = \texttt{'fetch}) \\
\ \wedge & \text{proper-r-statep}\,(r1,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{proper-r-statep}\,(r2,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{r-equal}\,(r2,\ r1)) \\
\to & (\text{r-fetch-okp}\,(\text{r-current-instruction}\,(r1),\ r2) \\
 & \ = \ \text{r-fetch-okp}\,(\text{r-current-instruction}\,(r1),\ r1))
\end{array}
$$

THEOREM: fetch-preserves-r-equal

$$
\begin{array}{ll}
((\text{r-psw}\,(r1) = \texttt{'run}) & \\
\ \wedge & (\text{car}\,(\text{r-current-instruction}\,(r1)) = \texttt{'fetch}) \\
\ \wedge & \text{proper-r-statep}\,(r1,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{proper-r-statep}\,(r2,\ \mathit{load\text{-}addr}) \\
\ \wedge & \text{r-fetch-okp}\,(\text{r-current-instruction}\,(r1),\ r2) \\
\ \wedge & \text{r-equal}\,(r2,\ r1))
\end{array}
$$

→ r-equal (r-fetch-step (r-current-instruction (*r1*), *r2*),
r-fetch-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-add-int-okp
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'add-int`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-add-int-okp (r-current-instruction (*r1*), *r2*)
= r-add-int-okp (r-current-instruction (*r1*), *r1*))

THEOREM: add-int-preserves-r-equal
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'add-int`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-add-int-okp (r-current-instruction (*r1*), *r2*)
∧ r-equal (*r2*, *r1*))
→ r-equal (r-add-int-step (r-current-instruction (*r1*), *r2*),
r-add-int-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-add-int-with-carry-okp
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'add-int-with-carry`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-equal (*r2*, *r1*))
→ (r-add-int-with-carry-okp (r-current-instruction (*r1*), *r2*)
= r-add-int-with-carry-okp (r-current-instruction (*r1*), *r1*))

THEOREM: add-int-with-carry-preserves-r-equal
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'add-int-with-carry`)
∧ proper-r-statep (*r1*, *load-addr*)
∧ proper-r-statep (*r2*, *load-addr*)
∧ r-add-int-with-carry-okp (r-current-instruction (*r1*), *r2*)
∧ r-equal (*r2*, *r1*))
→ r-equal (r-add-int-with-carry-step (r-current-instruction (*r1*), *r2*),
r-add-int-with-carry-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-add1-int-okp
((r-psw (*r1*) = `'run`)
∧ (car (r-current-instruction (*r1*)) = `'add1-int`)
∧ proper-r-statep (*r1*, *load-addr*)

$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-add1-int-okp (r-current-instruction (*r1*), *r2*)
= r-add1-int-okp (r-current-instruction (*r1*), *r1*))

THEOREM: add1-int-preserves-r-equal
((r-psw (*r1*) = `'run`)
$\wedge$ (car (r-current-instruction (*r1*)) = `'add1-int`)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-add1-int-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-add1-int-step (r-current-instruction (*r1*), *r2*),
r-add1-int-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-sub-int-okp
((r-psw (*r1*) = `'run`)
$\wedge$ (car (r-current-instruction (*r1*)) = `'sub-int`)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-sub-int-okp (r-current-instruction (*r1*), *r2*)
= r-sub-int-okp (r-current-instruction (*r1*), *r1*))

THEOREM: sub-int-preserves-r-equal
((r-psw (*r1*) = `'run`)
$\wedge$ (car (r-current-instruction (*r1*)) = `'sub-int`)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-sub-int-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-sub-int-step (r-current-instruction (*r1*), *r2*),
r-sub-int-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-sub-int-with-carry-okp
((r-psw (*r1*) = `'run`)
$\wedge$ (car (r-current-instruction (*r1*)) = `'sub-int-with-carry`)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-sub-int-with-carry-okp (r-current-instruction (*r1*), *r2*)
= r-sub-int-with-carry-okp (r-current-instruction (*r1*), *r1*))

THEOREM: sub-int-with-carry-preserves-r-equal
((r-psw (*r1*) = `'run`)

$\wedge$ (car (r-current-instruction ($r1$)) = `'sub-int-with-carry`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-sub-int-with-carry-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-sub-int-with-carry-step (r-current-instruction ($r1$), $r2$),
r-sub-int-with-carry-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-sub1-int-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'sub1-int`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-sub1-int-okp (r-current-instruction ($r1$), $r2$)
= r-sub1-int-okp (r-current-instruction ($r1$), $r1$))

THEOREM: sub1-int-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'sub1-int`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-sub1-int-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-sub1-int-step (r-current-instruction ($r1$), $r2$),
r-sub1-int-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-neg-int-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'neg-int`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-neg-int-okp (r-current-instruction ($r1$), $r2$)
= r-neg-int-okp (r-current-instruction ($r1$), $r1$))

THEOREM: neg-int-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'neg-int`)
$\wedge$ proper-r-statep ($r1$, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-neg-int-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-neg-int-step (r-current-instruction ($r1$), $r2$),
r-neg-int-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-lt-int-okp
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'lt-int**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
 → (r-lt-int-okp (r-current-instruction ($r1$), $r2$)
        = r-lt-int-okp (r-current-instruction ($r1$), $r1$))

THEOREM: lt-int-preserves-r-equal
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'lt-int**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-lt-int-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
 → r-equal (r-lt-int-step (r-current-instruction ($r1$), $r2$),
              r-lt-int-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-int-to-nat-okp
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'int-to-nat**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
 → (r-int-to-nat-okp (r-current-instruction ($r1$), $r2$)
        = r-int-to-nat-okp (r-current-instruction ($r1$), $r1$))

THEOREM: int-to-nat-preserves-r-equal
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'int-to-nat**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-int-to-nat-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
 → r-equal (r-int-to-nat-step (r-current-instruction ($r1$), $r2$),
              r-int-to-nat-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-add-nat-okp
((r-psw ($r1$) = **'run**)
 ∧ (car (r-current-instruction ($r1$)) = **'add-nat**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
 → (r-add-nat-okp (r-current-instruction ($r1$), $r2$)
        = r-add-nat-okp (r-current-instruction ($r1$), $r1$))

THEOREM: add-nat-preserves-r-equal
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'add-nat`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-add-nat-okp (r-current-instruction (*r1*), *r2*)
 ∧ r-equal (*r2*, *r1*))
→ r-equal (r-add-nat-step (r-current-instruction (*r1*), *r2*),
 r-add-nat-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-sub-nat-okp
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'sub-nat`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ (r-sub-nat-okp (r-current-instruction (*r1*), *r2*)
 = r-sub-nat-okp (r-current-instruction (*r1*), *r1*))

THEOREM: sub-nat-preserves-r-equal
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'sub-nat`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-sub-nat-okp (r-current-instruction (*r1*), *r2*)
 ∧ r-equal (*r2*, *r1*))
→ r-equal (r-sub-nat-step (r-current-instruction (*r1*), *r2*),
 r-sub-nat-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-sub-nat-with-carry-okp
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'sub-nat-with-carry`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ (r-sub-nat-with-carry-okp (r-current-instruction (*r1*), *r2*)
 = r-sub-nat-with-carry-okp (r-current-instruction (*r1*), *r1*))

THEOREM: sub-nat-with-carry-preserves-r-equal
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'sub-nat-with-carry`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-sub-nat-with-carry-okp (r-current-instruction (*r1*), *r2*)
 ∧ r-equal (*r2*, *r1*))

$\rightarrow$ r-equal (r-sub-nat-with-carry-step (r-current-instruction (*r1*), *r2*),
r-sub-nat-with-carry-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-sub1-nat-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'sub1-nat**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-sub1-nat-okp (r-current-instruction (*r1*), *r2*)
= r-sub1-nat-okp (r-current-instruction (*r1*), *r1*))

THEOREM: sub1-nat-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'sub1-nat**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-sub1-nat-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-sub1-nat-step (r-current-instruction (*r1*), *r2*),
r-sub1-nat-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-lt-nat-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'lt-nat**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-lt-nat-okp (r-current-instruction (*r1*), *r2*)
= r-lt-nat-okp (r-current-instruction (*r1*), *r1*))

THEOREM: lt-nat-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'lt-nat**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-lt-nat-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-lt-nat-step (r-current-instruction (*r1*), *r2*),
r-lt-nat-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-mult2-nat-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'mult2-nat**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)

$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-mult2-nat-okp (r-current-instruction ($r1$), $r2$)
$=$ r-mult2-nat-okp (r-current-instruction ($r1$), $r1$))

THEOREM: mult2-nat-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$)) = `'mult2-nat`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-mult2-nat-okp (r-current-instruction ($r1$), $r2$)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-mult2-nat-step (r-current-instruction ($r1$), $r2$),
r-mult2-nat-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-div2-nat-okp
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$)) = `'div2-nat`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-div2-nat-okp (r-current-instruction ($r1$), $r2$)
$=$ r-div2-nat-okp (r-current-instruction ($r1$), $r1$))

THEOREM: div2-nat-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$)) = `'div2-nat`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-div2-nat-okp (r-current-instruction ($r1$), $r2$)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-div2-nat-step (r-current-instruction ($r1$), $r2$),
r-div2-nat-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-or-bitv-okp
((r-psw ($r1$) = `'run`)
$\land$ (car (r-current-instruction ($r1$)) = `'or-bitv`)
$\land$ proper-r-statep ($r1$, $load\text{-}addr$)
$\land$ proper-r-statep ($r2$, $load\text{-}addr$)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-or-bitv-okp (r-current-instruction ($r1$), $r2$)
$=$ r-or-bitv-okp (r-current-instruction ($r1$), $r1$))

THEOREM: or-bitv-preserves-r-equal
((r-psw ($r1$) = `'run`)

$\wedge$ (car (r-current-instruction ($r1$)) = `'or-bitv`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-or-bitv-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-or-bitv-step (r-current-instruction ($r1$), $r2$),
r-or-bitv-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-and-bitv-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'and-bitv`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-and-bitv-okp (r-current-instruction ($r1$), $r2$)
= r-and-bitv-okp (r-current-instruction ($r1$), $r1$))

THEOREM: and-bitv-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'and-bitv`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-and-bitv-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-and-bitv-step (r-current-instruction ($r1$), $r2$),
r-and-bitv-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-not-bitv-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'not-bitv`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-not-bitv-okp (r-current-instruction ($r1$), $r2$)
= r-not-bitv-okp (r-current-instruction ($r1$), $r1$))

THEOREM: not-bitv-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'not-bitv`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-not-bitv-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-not-bitv-step (r-current-instruction ($r1$), $r2$),
r-not-bitv-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-xor-bitv-okp
((r-psw $(r1)$ = 'run)
 ∧ (car (r-current-instruction $(r1)$) = 'xor-bitv)
 ∧ proper-r-statep $(r1, \ load\text{-}addr)$
 ∧ proper-r-statep $(r2, \ load\text{-}addr)$
 ∧ r-equal $(r2, \ r1)$)
→ (r-xor-bitv-okp (r-current-instruction $(r1)$, $r2$)
 = r-xor-bitv-okp (r-current-instruction $(r1)$, $r1$))

THEOREM: xor-bitv-preserves-r-equal
((r-psw $(r1)$ = 'run)
 ∧ (car (r-current-instruction $(r1)$) = 'xor-bitv)
 ∧ proper-r-statep $(r1, \ load\text{-}addr)$
 ∧ proper-r-statep $(r2, \ load\text{-}addr)$
 ∧ r-xor-bitv-okp (r-current-instruction $(r1)$, $r2$)
 ∧ r-equal $(r2, \ r1)$)
→ r-equal (r-xor-bitv-step (r-current-instruction $(r1)$, $r2$),
 r-xor-bitv-step (r-current-instruction $(r1)$, $r1$))

THEOREM: r-equal-implies-equal-r-rsh-bitv-okp
((r-psw $(r1)$ = 'run)
 ∧ (car (r-current-instruction $(r1)$) = 'rsh-bitv)
 ∧ proper-r-statep $(r1, \ load\text{-}addr)$
 ∧ proper-r-statep $(r2, \ load\text{-}addr)$
 ∧ r-equal $(r2, \ r1)$)
→ (r-rsh-bitv-okp (r-current-instruction $(r1)$, $r2$)
 = r-rsh-bitv-okp (r-current-instruction $(r1)$, $r1$))

THEOREM: rsh-bitv-preserves-r-equal
((r-psw $(r1)$ = 'run)
 ∧ (car (r-current-instruction $(r1)$) = 'rsh-bitv)
 ∧ proper-r-statep $(r1, \ load\text{-}addr)$
 ∧ proper-r-statep $(r2, \ load\text{-}addr)$
 ∧ r-rsh-bitv-okp (r-current-instruction $(r1)$, $r2$)
 ∧ r-equal $(r2, \ r1)$)
→ r-equal (r-rsh-bitv-step (r-current-instruction $(r1)$, $r2$),
 r-rsh-bitv-step (r-current-instruction $(r1)$, $r1$))

THEOREM: r-equal-implies-equal-r-lsh-bitv-okp
((r-psw $(r1)$ = 'run)
 ∧ (car (r-current-instruction $(r1)$) = 'lsh-bitv)
 ∧ proper-r-statep $(r1, \ load\text{-}addr)$
 ∧ proper-r-statep $(r2, \ load\text{-}addr)$
 ∧ r-equal $(r2, \ r1)$)
→ (r-lsh-bitv-okp (r-current-instruction $(r1)$, $r2$)
 = r-lsh-bitv-okp (r-current-instruction $(r1)$, $r1$))

THEOREM: lsh-bitv-preserves-r-equal
((r-psw ($r1$) = '**run**)
 ∧ (car (r-current-instruction ($r1$)) = '**lsh-bitv**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-lsh-bitv-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
→ r-equal (r-lsh-bitv-step (r-current-instruction ($r1$), $r2$),
 r-lsh-bitv-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-or-bool-okp
((r-psw ($r1$) = '**run**)
 ∧ (car (r-current-instruction ($r1$)) = '**or-bool**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-or-bool-okp (r-current-instruction ($r1$), $r2$)
 = r-or-bool-okp (r-current-instruction ($r1$), $r1$))

THEOREM: or-bool-preserves-r-equal
((r-psw ($r1$) = '**run**)
 ∧ (car (r-current-instruction ($r1$)) = '**or-bool**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-or-bool-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
→ r-equal (r-or-bool-step (r-current-instruction ($r1$), $r2$),
 r-or-bool-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-and-bool-okp
((r-psw ($r1$) = '**run**)
 ∧ (car (r-current-instruction ($r1$)) = '**and-bool**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-and-bool-okp (r-current-instruction ($r1$), $r2$)
 = r-and-bool-okp (r-current-instruction ($r1$), $r1$))

THEOREM: and-bool-preserves-r-equal
((r-psw ($r1$) = '**run**)
 ∧ (car (r-current-instruction ($r1$)) = '**and-bool**)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-and-bool-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))

$\rightarrow$ r-equal (r-and-bool-step (r-current-instruction (*r1*), *r2*),
r-and-bool-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-not-bool-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'not-bool**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-not-bool-okp (r-current-instruction (*r1*), *r2*)
= r-not-bool-okp (r-current-instruction (*r1*), *r1*))

THEOREM: not-bool-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'not-bool**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-not-bool-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-not-bool-step (r-current-instruction (*r1*), *r2*),
r-not-bool-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-jump-if-temp-stk-full-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'jump-if-temp-stk-full**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-jump-if-temp-stk-full-okp (r-current-instruction (*r1*), *r2*)
= r-jump-if-temp-stk-full-okp (r-current-instruction (*r1*), *r1*))

THEOREM: jump-if-temp-stk-full-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'jump-if-temp-stk-full**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-jump-if-temp-stk-full-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-jump-if-temp-stk-full-step (r-current-instruction (*r1*), *r2*),
r-jump-if-temp-stk-full-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-jump-if-temp-stk-empty-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'jump-if-temp-stk-empty**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)

$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-jump-if-temp-stk-empty-okp (r-current-instruction ($r1$), $r2$)
= r-jump-if-temp-stk-empty-okp (r-current-instruction ($r1$), $r1$))

THEOREM: jump-if-temp-stk-empty-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'jump-if-temp-stk-empty`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-jump-if-temp-stk-empty-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-jump-if-temp-stk-empty-step (r-current-instruction ($r1$), $r2$),
r-jump-if-temp-stk-empty-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-test-nat-and-jump-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'test-nat-and-jump`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-test-nat-and-jump-okp (r-current-instruction ($r1$), $r2$)
= r-test-nat-and-jump-okp (r-current-instruction ($r1$), $r1$))

THEOREM: test-nat-and-jump-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'test-nat-and-jump`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-test-nat-and-jump-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-test-nat-and-jump-step (r-current-instruction ($r1$), $r2$),
r-test-nat-and-jump-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-test-int-and-jump-okp
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'test-int-and-jump`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-test-int-and-jump-okp (r-current-instruction ($r1$), $r2$)
= r-test-int-and-jump-okp (r-current-instruction ($r1$), $r1$))

THEOREM: test-int-and-jump-preserves-r-equal
((r-psw ($r1$) = `'run`)

$\wedge$ (car (r-current-instruction (*r1*)) = **'test-int-and-jump**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-test-int-and-jump-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-test-int-and-jump-step (r-current-instruction (*r1*), *r2*),
r-test-int-and-jump-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-test-bool-and-jump-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'test-bool-and-jump**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-test-bool-and-jump-okp (r-current-instruction (*r1*), *r2*)
= r-test-bool-and-jump-okp (r-current-instruction (*r1*), *r1*))

THEOREM: test-bool-and-jump-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'test-bool-and-jump**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-test-bool-and-jump-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-test-bool-and-jump-step (r-current-instruction (*r1*), *r2*),
r-test-bool-and-jump-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-test-bitv-and-jump-okp
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'test-bitv-and-jump**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ (r-test-bitv-and-jump-okp (r-current-instruction (*r1*), *r2*)
= r-test-bitv-and-jump-okp (r-current-instruction (*r1*), *r1*))

THEOREM: test-bitv-and-jump-preserves-r-equal
((r-psw (*r1*) = **'run**)
$\wedge$ (car (r-current-instruction (*r1*)) = **'test-bitv-and-jump**)
$\wedge$ proper-r-statep (*r1*, *load-addr*)
$\wedge$ proper-r-statep (*r2*, *load-addr*)
$\wedge$ r-test-bitv-and-jump-okp (r-current-instruction (*r1*), *r2*)
$\wedge$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-test-bitv-and-jump-step (r-current-instruction (*r1*), *r2*),
r-test-bitv-and-jump-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-equal-implies-equal-r-pop-locn-okp
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'pop-locn`)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-pop-locn-okp (r-current-instruction ($r1$), $r2$)
 = r-pop-locn-okp (r-current-instruction ($r1$), $r1$))

THEOREM: pop-locn-preserves-r-equal
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'pop-locn`)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-pop-locn-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
→ r-equal (r-pop-locn-step (r-current-instruction ($r1$), $r2$),
 r-pop-locn-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-locn-okp
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'locn`)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-locn-okp (r-current-instruction ($r1$), $r2$)
 = r-locn-okp (r-current-instruction ($r1$), $r1$))

THEOREM: locn-preserves-r-equal
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'locn`)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-locn-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
→ r-equal (r-locn-step (r-current-instruction ($r1$), $r2$),
 r-locn-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-jump-case-okp
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'jump-case`)
 ∧ proper-r-statep ($r1$, *load-addr*)
 ∧ proper-r-statep ($r2$, *load-addr*)
 ∧ r-equal ($r2$, $r1$))
→ (r-jump-case-okp (r-current-instruction ($r1$), $r2$)
 = r-jump-case-okp (r-current-instruction ($r1$), $r1$))

THEOREM: jump-case-preserves-r-equal
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'jump-case`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-jump-case-okp (r-current-instruction (*r1*), *r2*)
 ∧ r-equal (*r2*, *r1*))
→ r-equal (r-jump-case-step (r-current-instruction (*r1*), *r2*),
 r-jump-case-step (r-current-instruction (*r1*), *r1*))

```
; To do call, we need:
```

THEOREM: r-equal-implies-equal-all-r-objectps
r-equal (*r2*, *r1*) → (all-r-objectps (*lst*, *r2*) = all-r-objectps (*lst*, *r1*))

THEOREM: r-equal-implies-equal-r-call-okp
((r-psw (*r1*) = `'run`)
 ∧ (car (r-current-instruction (*r1*)) = `'call`)
 ∧ proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ (r-call-okp (r-current-instruction (*r1*), *r2*)
 = r-call-okp (r-current-instruction (*r1*), *r1*))

```
; Now we start working on call-step itself:

; There are two main problems.  The first is that the machine fetches the
; same actuals from each of the two states.  This is a generalization of
; r-equal-implies-equal-get-tstk.
```

DEFINITION:
stk-equal-fetch-n-hint (*n*, *sp*, *segment2*)
= **if** $n \simeq$ `0` **then t**
 elseif (length (cdr (assoc (car (untag (*sp*)), *segment2*))) − 1)
 < cdr (untag (*sp*)) **then t**
 else stk-equal-fetch-n-hint (*n* − 1, pop-stk (*sp*), *segment2*) **endif**

THEOREM: stk-equal-implies-equal-fetch-n
(listp (untag (*sp*))
 ∧ stk-equal (*sp*, *segment2*, *segment1*)
 ∧ (stk-length (*sp*, *segment2*) $\not<$ *n*))
→ (fetch-n (*n*, *sp*, *segment2*) = fetch-n (*n*, *sp*, *segment1*))

THEOREM: r-equal-implies-equal-fetch-n-call

$$
\begin{array}{l}
(\text{r-equal}\,(r2,\ r1) \\
\ \land\ \ \text{listp}\,(\text{untag}\,(\text{r-tsp}\,(r1))) \\
\ \land\ \ (\text{stk-length}\,(\text{r-tsp}\,(r1),\ \text{r-sys-data-segment}\,(r1)) \not< n)) \\
\to\ \ (\text{fetch-n}\,(n,\ \text{r-tsp}\,(r1),\ \text{r-sys-data-segment}\,(r2)) \\
\qquad =\ \ \text{fetch-n}\,(n,\ \text{r-tsp}\,(r1),\ \text{r-sys-data-segment}\,(r1)))
\end{array}
$$

```
; The second problem is proving that deposit-lst produces stk-equal
; stacks.

; There may be an easier way to do this, but I don't see it.  My
; plan is to replace deposit-lst by deposit-lst1, which runs down
; the list in the reverse order and does the put-assocs on the outside
; of its recursion.  This function, which we prove equivalent to
; deposit-lst, is easier for stk-equal to handle because each time
; it recurses it kicks out a put-assoc for the stk-equal to eat.
```

DEFINITION:

$$
\begin{array}{l}
\text{deposit-lst1}\,(lst,\ name,\ n,\ segment) \\
=\ \ \textbf{if}\ lst \simeq \textbf{nil}\ \ \textbf{then}\ segment \\
\quad\ \ \textbf{else}\ \text{put-assoc}\,(\text{put}\,(\text{car}\,(lst), \\
\qquad\qquad\qquad\qquad n, \\
\qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(name, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{deposit-lst1}\,(\text{cdr}\,(lst), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad name, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1 + n, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad segment)))), \\
\qquad\qquad\qquad name, \\
\qquad\qquad\qquad segment)\ \textbf{endif}
\end{array}
$$

```
; To prove deposit-lst is deposit-lst1 we need to make explicit the
; relation with append because the main theorem involves reverse:
```

THEOREM: deposit-lst1-append

$$
\begin{array}{l}
(n \in \mathbf{N}) \\
\to\ \ (\text{deposit-lst1}\,(\text{append}\,(a,\ \text{list}\,(x)),\ name,\ n,\ segment) \\
\qquad =\ \ \text{deposit-lst1}\,(a, \\
\qquad\qquad\qquad name, \\
\qquad\qquad\qquad n, \\
\qquad\qquad\qquad \text{put-assoc}\,(\text{put}\,(x, \\
\qquad\qquad\qquad\qquad\qquad n + \text{length}\,(a), \\
\qquad\qquad\qquad\qquad\qquad \text{cdr}\,(\text{assoc}\,(name,\ segment))), \\
\qquad\qquad\qquad\qquad name, \\
\qquad\qquad\qquad\qquad segment)))
\end{array}
$$

THEOREM: plus-difference{special}
$(x \not< y) \rightarrow ((y + (x - y)) = \text{fix}(x))$

```
; So here is the equivalence of the two versions of deposit-lst.
```

THEOREM: deposit-lst-equal-deposit-lst1
$$\begin{array}{l} ((\text{cdr}(\text{untag}(sp)) \in \mathbf{N}) \land (\text{cdr}(\text{untag}(sp)) \not< (\text{length}(lst) - 1))) \\ \rightarrow \quad (\text{deposit-lst}(lst, sp, segment) \\ \qquad = \quad \text{deposit-lst1}(\text{rev}(lst), \\ \qquad\qquad\qquad \text{car}(\text{untag}(sp)), \\ \qquad\qquad\qquad \text{cdr}(\text{untag}(sp)) - (\text{length}(lst) - 1), \\ \qquad\qquad\qquad segment)) \end{array}$$

```
; Now I am headed for the theorem that says that if you have
; two stk-equal segments then the result of doing deposit-lst is
; stk-equal.  In the proof deposit-lst will be replaced, as above,
; by deposit-lst1, and then I will use arithmetic, including
```

EVENT: Enable difference-elim.

EVENT: Enable difference-elim-rewrite.

```
; and
```

THEOREM: difference-sub1-plus-etc
$$\begin{array}{l} ((v \in \mathbf{N}) \land \text{listp}(lst)) \\ \rightarrow \quad ((((v + \text{length}(lst)) - 1) - (\text{length}(lst) - 1)) = v) \end{array}$$

```
; to reduce the stk-equal-deposit-lst-generalized theorem to an instance of
; stk-equal-deposit-lst1, below.  By the way, the order of the
; actuals, whether reversed or not, is unimportant here; i.e., the
; reverse introduced above is now simply generalized away.

; The proof of stk-equal-deposit-lst1 relies on stk-equal-put-assoc-push
; to eat each put-assoc kicked out by deposit-lst1.  We need to relieve
; the hypotheses of the -push lemma:
```

THEOREM: length-cdr-assoc-deposit-lst1
$$\begin{array}{l} (\text{definedp}(name, segment) \\ \land \quad (\text{length}(\text{cdr}(\text{assoc}(name, segment))) \not< (n + \text{length}(lst)))) \\ \rightarrow \quad (\text{length}(\text{cdr}(\text{assoc}(name, \text{deposit-lst1}(lst, name, n, segment)))) \\ \qquad = \quad \text{length}(\text{cdr}(\text{assoc}(name, segment)))) \end{array}$$

```
; In the main theorem to which we are heading, we have tag, type, and
; untag enabled.  We therefore need to state this lemma in terms of list
; rather than tag, and enable the accessors.  We explain below.
```

THEOREM: room-to-deposit-lst-implies-not-empty-stkp
$((n + \text{length}(\text{cdr}(\mathit{lst}))) < (\text{length}(\text{cdr}(\text{assoc}(\mathit{name}, \mathit{segment1}))) - 1))$
$\rightarrow \quad (\neg\ \text{empty-stkp}(\text{list}(\texttt{'sys-addr}, \text{cons}(\mathit{name}, n)), \mathit{segment1}))$

```
; After we relieve the hyps we need to simplify the rhs.  Recall that
; the rhs of the stk-equal-put-assoc-push lemma says that you continue
; the stk-equal at the next higher position after put-associng the
; name with the list into which you did the put.  In our case, the
; list into which we did the put is obtained by associng for
; the name in the result of the continued deposit-lst1.  In any case,
; the identity we need is this:
```

THEOREM: stk-equal-put-assoc-cdr-assoc
$$
\begin{array}{l}
((n \in \mathbf{N}) \wedge \text{definedp}(\mathit{name}, \mathit{old\text{-}segment1}) \wedge \text{definedp}(\mathit{name}, \mathit{old\text{-}segment2})) \\
\rightarrow \quad (\text{stk-equal}(\text{list}(\mathit{sys\text{-}addr}, \text{cons}(\mathit{name}, n)), \\
\qquad\qquad\qquad \text{put-assoc}(\text{cdr}(\text{assoc}(\mathit{name}, \mathit{segment1})), \mathit{name}, \mathit{old\text{-}segment1}), \\
\qquad\qquad\qquad \text{put-assoc}(\text{cdr}(\text{assoc}(\mathit{name}, \mathit{segment2})), \mathit{name}, \mathit{old\text{-}segment2})) \\
\qquad = \quad \text{stk-equal}(\text{list}(\mathit{sys\text{-}addr}, \text{cons}(\mathit{name}, n)), \mathit{segment1}, \mathit{segment2}))
\end{array}
$$

```
; This is the key result for stk-equal and deposit-lst1.  The lemma is
; a little unusual in having sys-addr as a variable.  Most of the time
; we've used the constant 'sys-addr.  The reason we used a variable
; this time is just that in the lemma we are headed for,
; stk-equal-deposit-lst-generalized, we don't bother to say that
; sp-init is a sys-addr address.
```

THEOREM: stk-equal-deposit-lst1
$$
\begin{array}{l}
((n \in \mathbf{N}) \\
\ \wedge \quad \text{definedp}(\mathit{name}, \mathit{segment2}) \\
\ \wedge \quad ((n + \text{length}(\mathit{lst})) < \text{length}(\text{cdr}(\text{assoc}(\mathit{name}, \mathit{segment2})))) \\
\ \wedge \quad \text{definedp}(\mathit{name}, \mathit{segment1}) \\
\ \wedge \quad ((n + \text{length}(\mathit{lst})) < \text{length}(\text{cdr}(\text{assoc}(\mathit{name}, \mathit{segment1})))) \\
\ \wedge \quad ((n + \text{length}(\mathit{lst})) \not< \text{length}(\mathit{lst})) \\
\ \wedge \quad \text{stk-equal}(\text{list}(\mathit{sys\text{-}addr}, \text{cons}(\mathit{name}, n + \text{length}(\mathit{lst}))), \\
\qquad\qquad\qquad \mathit{segment2}, \\
\qquad\qquad\qquad \mathit{segment1})) \\
\rightarrow \quad \text{stk-equal}(\text{list}(\mathit{sys\text{-}addr}, \text{cons}(\mathit{name}, n)), \\
\qquad\qquad\qquad \text{deposit-lst1}(\mathit{lst}, \mathit{name}, n, \mathit{segment2}), \\
\qquad\qquad\qquad \text{deposit-lst1}(\mathit{lst}, \mathit{name}, n, \mathit{segment1}))
\end{array}
$$

```
; Here is the main theorem about stk-equal and deposit-lst.  As noted, the
; proof involves an elim to get rid of the difference introduced when we
; trade deposit-lst for deposit-lst1.  That elim also introduces conses for
; sp-init, below, and so we need to enable type and untag so they compute on
; conses.  That is why the lemma development above used list instead of tag
; everywhere.  The lemma is of lemma class nil because we need to instantiate
; it later to derive the actual rewrite rule we need.
```

THEOREM: stk-equal-deposit-lst-generalized
$$
\begin{array}{ll}
\multicolumn{2}{l}{((\text{cddr}\,(\textit{sp-init}) = \mathbf{nil})} \\
\land & \text{addressp}\,(\textit{sp-init},\ \textit{segment2}) \\
\land & \text{addressp}\,(\textit{sp-init},\ \textit{segment1}) \\
\land & (\text{free-stk-size}\,(\textit{sp-init}) \not< \text{length}\,(\textit{lst})) \\
\land & \text{stk-equal}\,(\textit{sp-init},\ \textit{segment2},\ \textit{segment1})) \\
\rightarrow & \text{stk-equal}\,(\text{push-stk-n}\,(\text{length}\,(\textit{lst}),\ \textit{sp-init}), \\
& \qquad\qquad \text{deposit-lst}\,(\textit{lst},\ \text{push-stk}\,(\textit{sp-init}),\ \textit{segment2}), \\
& \qquad\qquad \text{deposit-lst}\,(\textit{lst},\ \text{push-stk}\,(\textit{sp-init}),\ \textit{segment1}))
\end{array}
$$

```
; We now disable the things we introduced just for this proof.
```

EVENT: Disable deposit-lst-equal-deposit-lst1.

EVENT: Disable difference-elim.

EVENT: Disable difference-sub1-plus-etc.

EVENT: Disable room-to-deposit-lst-implies-not-empty-stkp.

THEOREM: stk-equal-deposit-lst
$$
\begin{array}{ll}
\multicolumn{2}{l}{((\textit{n-final} = (\textit{n-init} - \text{length}\,(\textit{lst})))} \\
\land & \text{definedp}\,(\texttt{'cstk},\ \textit{segment2}) \\
\land & \text{definedp}\,(\texttt{'cstk},\ \textit{segment1}) \\
\land & (\textit{n-init} \in \mathbf{N}) \\
\land & (\textit{n-init} \not< \text{length}\,(\textit{lst})) \\
\land & (\textit{n-init} < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment2})))) \\
\land & (\textit{n-init} < \text{length}\,(\text{cdr}\,(\text{assoc}\,(\texttt{'cstk},\ \textit{segment1})))) \\
\land & \text{stk-equal}\,(\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\texttt{'cstk},\ \textit{n-init})),\ \textit{segment2},\ \textit{segment1})) \\
\rightarrow & \text{stk-equal}\,(\text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\texttt{'cstk},\ \textit{n-final})), \\
& \qquad\qquad \text{deposit-lst}\,(\textit{lst}, \\
& \qquad\qquad\qquad\qquad \text{tag}\,(\texttt{'sys-addr},\ \text{cons}\,(\texttt{'cstk},\ \textit{n-init} - 1)),
\end{array}
$$

$\qquad\qquad\qquad\qquad$ *segment2*),
$\qquad\qquad$ deposit-lst (*lst*,
$\qquad\qquad\qquad\qquad$ tag ('`sys-addr`, cons ('`cstk`, $n\text{-}init - 1$)),
$\qquad\qquad\qquad\qquad$ *segment1*))

```
; Returning now to the consideration of call itself and the stk-equality
; on the new cstks, the theorem above reduces (actually backchains) the
; problem to one of the stk-equality after the first pushes of the old fp
; and ret-pc.  The following lemma establishes that the stack on which
; we're pushing is non-empty and allows the famous stk-equal-put-assoc-push
; to handle those two pushes:
```

THEOREM: not-empty-stkp-push-csp
(proper-r-statep (r, *load-addr*) $\wedge$ ($n <$ cdr (untag (r-csp (r)))))
$\rightarrow$ ($\neg$ empty-stkp (tag ('`sys-addr`, cons ('`cstk`, n)),
$\qquad\qquad\qquad$ r-sys-data-segment (r)))

```
; We are finally left with the stk-equality of the two cstks below the
; new frame.  That is given to us, except that we have torn the csp
; apart with all the stuff above and need to put it back together:
```

THEOREM: proper-r-statep-implies-tag-untag-cstk
((type (x) = '`sys-addr`)
$\wedge$ (car (untag (x)) = '`cstk`)
$\wedge$ (cddr (x) = **nil**))
$\rightarrow$ (tag ('`sys-addr`, cons ('`cstk`, cdr (untag (x)))) = x)

```
; This completes the argument that the new cstks are stk-equal.

; We now turn to the stk-equality of the new tstks.

; The only observation we need to make is that the deposit-lst on the
; cstk does not affect the question of stk-equality on the tstk.  Then
; the knowledge that stk-equal extends upwards will handle things for
; us.

; The strategy is the same as before: replace deposit-lst with
; deposit-lst1 and appeal to the simple theorem about deposit-lst1:
```

THEOREM: stk-equal-tstk-deposit-lst1-cstk
stk-equal (tag ('`sys-addr`, cons ('`tstk`, n)),
$\qquad\qquad$ deposit-lst1 (*lst*, '`cstk`, k, *segment2*),
$\qquad\qquad$ deposit-lst1 (*lst*, '`cstk`, k, *segment1*))
= $\quad$ stk-equal (tag ('`sys-addr`, cons ('`tstk`, n)), *segment2*, *segment1*)

```
; So here is the result we needed for the stk-equality on the tstk:
```

THEOREM: stk-equal-tstk-deposit-lst-cstk
((car (untag (csp)) = 'cstk)
 ∧ (cdr (untag (csp)) ∈ **N**)
 ∧ (cdr (untag (csp)) $\not<$ (length (lst) − 1)))
→ (stk-equal (tag ('sys-addr, cons ('tstk, n)),
 deposit-lst (lst, csp, $segment2$),
 deposit-lst (lst, csp, $segment1$))
 = stk-equal (tag ('sys-addr, cons ('tstk, n)), $segment2$, $segment1$))

```
; The last thing we have to handle is the same-signature of the new
; sys-data-segments.  We use deposit-lst1 again.
```

THEOREM: same-signature-deposit-lst1
(litatom ($name$)
 ∧ definedp ($name$, $segment1$)
 ∧ definedp ($name$, $segment2$)
 ∧ (length (cdr (assoc ($name$, $segment1$))) $\not<$ (n + length (lst)))
 ∧ (length (cdr (assoc ($name$, $segment2$))) $\not<$ (n + length (lst))))
→ (same-signature (deposit-lst1 (lst, $name$, n, $segment2$),
 deposit-lst1 (lst, $name$, n, $segment1$))
 = same-signature ($segment2$, $segment1$))

THEOREM: same-signature-deposit-lst{special}
(litatom (car (untag (csp)))
 ∧ definedp (car (untag (csp)), $segment1$)
 ∧ definedp (car (untag (csp)), $segment2$)
 ∧ (cdr (untag (csp)) ∈ **N**)
 ∧ (cdr (untag (csp)) $\not<$ (length (lst) − 1))
 ∧ (cdr (untag (csp)) < length (cdr (assoc (car (untag (csp)), $segment1$))))
 ∧ (cdr (untag (csp)) < length (cdr (assoc (car (untag (csp)), $segment2$)))))
→ (same-signature (deposit-lst (lst, csp, $segment2$),
 deposit-lst (lst, csp, $segment1$))
 = same-signature ($segment2$, $segment1$))

```
; And that completes the r-equal work for call!
```

THEOREM: call-preserves-r-equal
((r-psw ($r1$) = 'run)
 ∧ (car (r-current-instruction ($r1$)) = 'call)
 ∧ proper-r-statep ($r1$, $load\text{-}addr$)

$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-call-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-call-step (r-current-instruction ($r1$), $r2$),
r-call-step (r-current-instruction ($r1$), $r1$))

```
; In order to reduce the pop-call case to call, we need to
; explain that the fake state created by pop-call does not
; affect the tests we make:
```

THEOREM: r-objectp-on-pop-call-state
((r-prog-segment (r) = *r-prog-segment*)
$\wedge$ (r-usr-data-segment (r) = *r-usr-data-segment*)
$\wedge$ (r-word-size (r) = *r-word-size*))
$\rightarrow$ (r-objectp (x,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,
r-y,
r-c-flg,
r-v-flg,
r-n-flg,
r-z-flg,
r-prog-segment,
r-usr-data-segment,
r-sys-data-segment (r),
r-word-size,
r-psw))
= r-objectp (x, r))

THEOREM: all-r-objectps-on-pop-call-state
((r-prog-segment (r) = *r-prog-segment*)
$\wedge$ (r-usr-data-segment (r) = *r-usr-data-segment*)
$\wedge$ (r-word-size (r) = *r-word-size*))
$\rightarrow$ (all-r-objectps (*lst*,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,
r-y,
r-c-flg,

$r\text{-}v\text{-}flg$,
$r\text{-}n\text{-}flg$,
$r\text{-}z\text{-}flg$,
$r\text{-}prog\text{-}segment$,
$r\text{-}usr\text{-}data\text{-}segment$,
r-sys-data-segment (r),
$r\text{-}word\text{-}size$,
$r\text{-}psw$))
= all-r-objectps (lst, r))

THEOREM: add1-r-pcp-on-pop-call-state
((r-prog-segment (r) = $r\text{-}prog\text{-}segment$) ∧ (r-pc ($r$) = $r\text{-}pc$))
→ (add1-r-pcp (r-state ($r\text{-}pc$,
$r\text{-}cfp$,
$r\text{-}csp$,
$r\text{-}tsp$,
$r\text{-}x$,
$r\text{-}y$,
$r\text{-}c\text{-}flg$,
$r\text{-}v\text{-}flg$,
$r\text{-}n\text{-}flg$,
$r\text{-}z\text{-}flg$,
$r\text{-}prog\text{-}segment$,
$r\text{-}usr\text{-}data\text{-}segment$,
r-sys-data-segment (r),
$r\text{-}word\text{-}size$,
$r\text{-}psw$))
= add1-r-pcp (r))

THEOREM: r-equal-implies-equal-r-pop-call-okp
((r-psw ($r1$) = `'run`)
∧ (car (r-current-instruction ($r1$)) = `'pop-call`)
∧ proper-r-statep ($r1$, $load\text{-}addr$)
∧ proper-r-statep ($r2$, $load\text{-}addr$)
∧ r-equal ($r2$, $r1$))
→ (r-pop-call-okp (r-current-instruction ($r1$), $r2$)
= r-pop-call-okp (r-current-instruction ($r1$), $r1$))

```
; Recall the lemma r-equal-implies-equal-fetch-n-call.  It rewrites
; (fetch-n n (r-tsp r1) (r-sys-data-segment r2)).  That will not
; do for pop-call because instead of (r-tsp r1) we are interested
; in (pop-stk (r-tsp r1)), with the pop-stk expanded.  So we
; prove a slightly different version:

; The proper-r-statep hypothesis added below is necessary to
```

```
; insure that (tag 'sys-addr (cons 'tstk (cdr (untag (r-tsp r1)))))
; is in fact (r-tsp r1).
```

THEOREM: r-equal-implies-equal-fetch-n-pop-call
(r-equal ($r2$, $r1$)
 ∧ proper-r-statep ($r1$, $load\text{-}addr$)
 ∧ listp (untag (r-tsp ($r1$)))
 ∧ (stk-length (tag (`'sys-addr`, cons (`'tstk`, 1 + cdr (untag (r-tsp ($r1$))))),
 r-sys-data-segment ($r1$))
 $\not<$ n))
→ (fetch-n (n,
 tag (`'sys-addr`, cons (`'tstk`, 1 + cdr (untag (r-tsp ($r1$))))),
 r-sys-data-segment ($r2$))
 = fetch-n (n,
 tag (`'sys-addr`,
 cons (`'tstk`, 1 + cdr (untag (r-tsp ($r1$))))),
 r-sys-data-segment ($r1$)))

THEOREM: pop-call-preserves-r-equal
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'pop-call`)
 ∧ proper-r-statep ($r1$, $load\text{-}addr$)
 ∧ proper-r-statep ($r2$, $load\text{-}addr$)
 ∧ r-pop-call-okp (r-current-instruction ($r1$), $r2$)
 ∧ r-equal ($r2$, $r1$))
→ r-equal (r-pop-call-step (r-current-instruction ($r1$), $r2$),
 r-pop-call-step (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-ret-okp
((r-psw ($r1$) = `'run`)
 ∧ (car (r-current-instruction ($r1$)) = `'ret`)
 ∧ proper-r-statep ($r1$, $load\text{-}addr$)
 ∧ proper-r-statep ($r2$, $load\text{-}addr$)
 ∧ r-equal ($r2$, $r1$))
→ (r-ret-okp (r-current-instruction ($r1$), $r2$)
 = r-ret-okp (r-current-instruction ($r1$), $r1$))

```
; Needless to say, on ret we need to know that the stk-equality
; on the cstk extends upwards too.  Previously we had proved it
; only for the tstk.
```

THEOREM: r-equal-implies-stk-equal-extends-upwards-cstk
(r-equal ($r2$, $r1$)

$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ ($k \in \mathbf{N}$)
$\wedge$ ($k \not< $ cdr (untag (r-csp ($r1$)))))
$\rightarrow$ stk-equal (tag (`'sys-addr`, cons (`'cstk`, k)),
r-sys-data-segment ($r2$),
r-sys-data-segment ($r1$))

```
; And we need to know about r-halt:
```

THEOREM: r-equal-implies-r-equal-r-halt
r-equal ($r2$, $r1$) $\rightarrow$ r-equal (r-halt ($r2$, $r\text{-}psw$), r-halt ($r1$, $r\text{-}psw$))

THEOREM: ret-preserves-r-equal
((r-psw ($r1$) = `'run`)
$\wedge$ (car (r-current-instruction ($r1$)) = `'ret`)
$\wedge$ proper-r-statep ($r1$, $load\text{-}addr$)
$\wedge$ proper-r-statep ($r2$, $load\text{-}addr$)
$\wedge$ r-ret-okp (r-current-instruction ($r1$), $r2$)
$\wedge$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-ret-step (r-current-instruction ($r1$), $r2$),
r-ret-step (r-current-instruction ($r1$), $r1$))

```
; We also have to prove that p-loadablep is preserved by p-step.
; We do it in the expected way, beginning with the individual
; instruction steppers.
```

THEOREM: no-op-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'no-op`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, $load\text{-}addr$)
$\wedge$ p-no-op-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-no-op-step (p-current-instruction (p), p), $load\text{-}addr$)

THEOREM: push-global-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'push-global`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, $load\text{-}addr$)
$\wedge$ p-push-global-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-push-global-step (p-current-instruction (p), p), $load\text{-}addr$)

THEOREM: pop-global-preserves-p-loadablep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'pop-global})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-loadablep}(p, \ \textit{load-addr})$
$\land \quad \text{p-pop-global-okp}(\text{p-current-instruction}(p), \ p))$
$\to \quad \text{p-loadablep}(\text{p-pop-global-step}(\text{p-current-instruction}(p), \ p), \ \textit{load-addr})$

THEOREM: push-local-preserves-p-loadablep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'push-local})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-loadablep}(p, \ \textit{load-addr})$
$\land \quad \text{p-push-local-okp}(\text{p-current-instruction}(p), \ p))$
$\to \quad \text{p-loadablep}(\text{p-push-local-step}(\text{p-current-instruction}(p), \ p), \ \textit{load-addr})$

THEOREM: pop-local-preserves-p-loadablep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'pop-local})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-loadablep}(p, \ \textit{load-addr})$
$\land \quad \text{p-pop-local-okp}(\text{p-current-instruction}(p), \ p))$
$\to \quad \text{p-loadablep}(\text{p-pop-local-step}(\text{p-current-instruction}(p), \ p), \ \textit{load-addr})$

THEOREM: deposit-preserves-p-loadablep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'deposit})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-loadablep}(p, \ \textit{load-addr})$
$\land \quad \text{p-deposit-okp}(\text{p-current-instruction}(p), \ p))$
$\to \quad \text{p-loadablep}(\text{p-deposit-step}(\text{p-current-instruction}(p), \ p), \ \textit{load-addr})$

THEOREM: add1-nat-preserves-p-loadablep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'add1-nat})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-loadablep}(p, \ \textit{load-addr})$
$\land \quad \text{p-add1-nat-okp}(\text{p-current-instruction}(p), \ p))$
$\to \quad \text{p-loadablep}(\text{p-add1-nat-step}(\text{p-current-instruction}(p), \ p), \ \textit{load-addr})$

THEOREM: add-addr-preserves-p-loadablep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad (\text{car}(\text{p-current-instruction}(p)) = \texttt{'add-addr})$
$\land \quad \text{proper-p-statep}(p)$
$\land \quad \text{p-loadablep}(p, \ \textit{load-addr})$

$\land$ p-add-addr-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-add-addr-step (p-current-instruction (p), p), *load-addr*)

THEOREM: add-nat-with-carry-preserves-p-loadablep
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'add-nat-with-carry`)
$\land$ proper-p-statep (p)
$\land$ p-loadablep (p, *load-addr*)
$\land$ p-add-nat-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-add-nat-with-carry-step (p-current-instruction (p), p),
load-addr)

THEOREM: mult2-nat-with-carry-out-preserves-p-loadablep
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'mult2-nat-with-carry-out`)
$\land$ proper-p-statep (p)
$\land$ p-loadablep (p, *load-addr*)
$\land$ p-mult2-nat-with-carry-out-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-mult2-nat-with-carry-out-step (p-current-instruction (p),
p),
load-addr)

THEOREM: fetch-temp-stk-preserves-p-loadablep
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'fetch-temp-stk`)
$\land$ proper-p-statep (p)
$\land$ p-loadablep (p, *load-addr*)
$\land$ p-fetch-temp-stk-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-fetch-temp-stk-step (p-current-instruction (p), p),
load-addr)

THEOREM: deposit-temp-stk-preserves-p-loadablep
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'deposit-temp-stk`)
$\land$ proper-p-statep (p)
$\land$ p-loadablep (p, *load-addr*)
$\land$ p-deposit-temp-stk-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-deposit-temp-stk-step (p-current-instruction (p), p),
load-addr)

THEOREM: pop*-preserves-p-loadablep
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'pop*`)
$\land$ proper-p-statep (p)
$\land$ p-loadablep (p, *load-addr*)

$\wedge$ p-pop*-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-pop*-step (p-current-instruction (p), p), *load-addr*)

THEOREM: popn-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'popn`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-popn-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-popn-step (p-current-instruction (p), p), *load-addr*)

THEOREM: set-local-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'set-local`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-set-local-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-set-local-step (p-current-instruction (p), p), *load-addr*)

THEOREM: set-global-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'set-global`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-set-global-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-set-global-step (p-current-instruction (p), p), *load-addr*)

THEOREM: push-constant-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'push-constant`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-push-constant-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-push-constant-step (p-current-instruction (p), p),
load-addr)

THEOREM: push-ctrl-stk-free-size-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'push-ctrl-stk-free-size`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-push-ctrl-stk-free-size-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-push-ctrl-stk-free-size-step (p-current-instruction (p), p),
load-addr)

THEOREM: push-temp-stk-free-size-preserves-p-loadablep
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'push-temp-stk-free-size`)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-push-temp-stk-free-size-okp (p-current-instruction (p), p))
→ p-loadablep (p-push-temp-stk-free-size-step (p-current-instruction (p), p),
load-addr)

THEOREM: push-temp-stk-index-preserves-p-loadablep
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'push-temp-stk-index`)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-push-temp-stk-index-okp (p-current-instruction (p), p))
→ p-loadablep (p-push-temp-stk-index-step (p-current-instruction (p), p),
load-addr)

THEOREM: pop-preserves-p-loadablep
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'pop`)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-pop-okp (p-current-instruction (p), p))
→ p-loadablep (p-pop-step (p-current-instruction (p), p), *load-addr*)

THEOREM: jump-preserves-p-loadablep
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'jump`)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-jump-okp (p-current-instruction (p), p))
→ p-loadablep (p-jump-step (p-current-instruction (p), p), *load-addr*)

THEOREM: pushj-preserves-p-loadablep
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'pushj`)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-pushj-okp (p-current-instruction (p), p))
→ p-loadablep (p-pushj-step (p-current-instruction (p), p), *load-addr*)

THEOREM: popj-preserves-p-loadablep
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'popj`)

$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep $(p,\ \textit{load-addr})$
$\wedge$ p-popj-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ p-loadablep (p-popj-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: sub-addr-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p))) = `'sub-addr`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep $(p,\ \textit{load-addr})$
$\wedge$ p-sub-addr-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ p-loadablep (p-sub-addr-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: eq-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p))) = `'eq`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep $(p,\ \textit{load-addr})$
$\wedge$ p-eq-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ p-loadablep (p-eq-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: lt-addr-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p))) = `'lt-addr`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep $(p,\ \textit{load-addr})$
$\wedge$ p-lt-addr-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ p-loadablep (p-lt-addr-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: fetch-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p))) = `'fetch`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep $(p,\ \textit{load-addr})$
$\wedge$ p-fetch-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ p-loadablep (p-fetch-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: add-int-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p))) = `'add-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep $(p,\ \textit{load-addr})$
$\wedge$ p-add-int-okp (p-current-instruction $(p),\ p)$)
$\rightarrow$ p-loadablep (p-add-int-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: add-int-with-carry-preserves-p-loadablep
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'add-int-with-carry**)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-add-int-with-carry-okp (p-current-instruction (p), p))
→ p-loadablep (p-add-int-with-carry-step (p-current-instruction (p), p), *load-addr*)

THEOREM: add1-int-preserves-p-loadablep
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'add1-int**)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-add1-int-okp (p-current-instruction (p), p))
→ p-loadablep (p-add1-int-step (p-current-instruction (p), p), *load-addr*)

THEOREM: sub-int-preserves-p-loadablep
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'sub-int**)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-sub-int-okp (p-current-instruction (p), p))
→ p-loadablep (p-sub-int-step (p-current-instruction (p), p), *load-addr*)

THEOREM: sub-int-with-carry-preserves-p-loadablep
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'sub-int-with-carry**)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-sub-int-with-carry-okp (p-current-instruction (p), p))
→ p-loadablep (p-sub-int-with-carry-step (p-current-instruction (p), p), *load-addr*)

THEOREM: sub1-int-preserves-p-loadablep
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'sub1-int**)
∧ proper-p-statep (p)
∧ p-loadablep $(p,\ \mathit{load\text{-}addr})$
∧ p-sub1-int-okp (p-current-instruction (p), p))
→ p-loadablep (p-sub1-int-step (p-current-instruction (p), p), *load-addr*)

THEOREM: neg-int-preserves-p-loadablep
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'neg-int**)

$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-neg-int-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-neg-int-step (p-current-instruction (p), p), *load-addr*)

THEOREM: lt-int-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'lt-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-lt-int-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-lt-int-step (p-current-instruction (p), p), *load-addr*)

THEOREM: int-to-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'int-to-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-int-to-nat-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-int-to-nat-step (p-current-instruction (p), p), *load-addr*)

THEOREM: add-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'add-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-add-nat-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-add-nat-step (p-current-instruction (p), p), *load-addr*)

THEOREM: sub-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'sub-nat`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-sub-nat-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-sub-nat-step (p-current-instruction (p), p), *load-addr*)

THEOREM: sub-nat-with-carry-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'sub-nat-with-carry`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-sub-nat-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-sub-nat-with-carry-step (p-current-instruction (p), p),
load-addr)

THEOREM: sub1-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub1-nat`)
 $\land$ proper-p-statep (p)
 $\land$ p-loadablep $(p,\ \textit{load-addr})$
 $\land$ p-sub1-nat-okp (p-current-instruction $(p),\ p$))
 $\rightarrow$ p-loadablep (p-sub1-nat-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: lt-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'lt-nat`)
 $\land$ proper-p-statep (p)
 $\land$ p-loadablep $(p,\ \textit{load-addr})$
 $\land$ p-lt-nat-okp (p-current-instruction $(p),\ p$))
 $\rightarrow$ p-loadablep (p-lt-nat-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: mult2-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'mult2-nat`)
 $\land$ proper-p-statep (p)
 $\land$ p-loadablep $(p,\ \textit{load-addr})$
 $\land$ p-mult2-nat-okp (p-current-instruction $(p),\ p$))
 $\rightarrow$ p-loadablep (p-mult2-nat-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: div2-nat-preserves-p-loadablep
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'div2-nat`)
 $\land$ proper-p-statep (p)
 $\land$ p-loadablep $(p,\ \textit{load-addr})$
 $\land$ p-div2-nat-okp (p-current-instruction $(p),\ p$))
 $\rightarrow$ p-loadablep (p-div2-nat-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: or-bitv-preserves-p-loadablep
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'or-bitv`)
 $\land$ proper-p-statep (p)
 $\land$ p-loadablep $(p,\ \textit{load-addr})$
 $\land$ p-or-bitv-okp (p-current-instruction $(p),\ p$))
 $\rightarrow$ p-loadablep (p-or-bitv-step (p-current-instruction $(p),\ p),\ \textit{load-addr})$

THEOREM: and-bitv-preserves-p-loadablep
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'and-bitv`)
 $\land$ proper-p-statep (p)
 $\land$ p-loadablep $(p,\ \textit{load-addr})$

$\wedge$ p-and-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-and-bitv-step (p-current-instruction (p), p), *load-addr*)

THEOREM: not-bitv-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'not-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-not-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-not-bitv-step (p-current-instruction (p), p), *load-addr*)

THEOREM: xor-bitv-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'xor-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-xor-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-xor-bitv-step (p-current-instruction (p), p), *load-addr*)

THEOREM: rsh-bitv-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'rsh-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-rsh-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-rsh-bitv-step (p-current-instruction (p), p), *load-addr*)

THEOREM: lsh-bitv-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'lsh-bitv`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-lsh-bitv-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-lsh-bitv-step (p-current-instruction (p), p), *load-addr*)

THEOREM: or-bool-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'or-bool`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-or-bool-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-or-bool-step (p-current-instruction (p), p), *load-addr*)

THEOREM: and-bool-preserves-p-loadablep
((p-psw (p) = `'run`)

$\wedge$ (car (p-current-instruction (p)) = `'and-bool`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-and-bool-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-and-bool-step (p-current-instruction (p), p), *load-addr*)

THEOREM: not-bool-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'not-bool`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-not-bool-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-not-bool-step (p-current-instruction (p), p), *load-addr*)

THEOREM: jump-if-temp-stk-full-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'jump-if-temp-stk-full`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-jump-if-temp-stk-full-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-jump-if-temp-stk-full-step (p-current-instruction (p), p),
load-addr)

THEOREM: jump-if-temp-stk-empty-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'jump-if-temp-stk-empty`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-jump-if-temp-stk-empty-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-jump-if-temp-stk-empty-step (p-current-instruction (p), p),
load-addr)

THEOREM: test-nat-and-jump-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'test-nat-and-jump`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-test-nat-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-test-nat-and-jump-step (p-current-instruction (p), p),
load-addr)

THEOREM: test-int-and-jump-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'test-int-and-jump`)
$\wedge$ proper-p-statep (p)

$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-test-int-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-test-int-and-jump-step (p-current-instruction (p), p),
load-addr)

THEOREM: test-bool-and-jump-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'test-bool-and-jump`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-test-bool-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-test-bool-and-jump-step (p-current-instruction (p), p),
load-addr)

THEOREM: test-bitv-and-jump-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'test-bitv-and-jump`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-test-bitv-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-test-bitv-and-jump-step (p-current-instruction (p), p),
load-addr)

THEOREM: pop-locn-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'pop-locn`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-pop-locn-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-pop-locn-step (p-current-instruction (p), p), *load-addr*)

THEOREM: locn-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'locn`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-locn-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-locn-step (p-current-instruction (p), p), *load-addr*)

THEOREM: jump-case-preserves-p-loadablep
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'jump-case`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ p-jump-case-okp (p-current-instruction (p), p))
$\rightarrow$ p-loadablep (p-jump-case-step (p-current-instruction (p), p), *load-addr*)

THEOREM: call-preserves-p-loadablep
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'call**)
 ∧ proper-p-statep (p)
 ∧ p-loadablep (p, *load-addr*)
 ∧ p-call-okp (p-current-instruction (p), p))
→ p-loadablep (p-call-step (p-current-instruction (p), p), *load-addr*)

THEOREM: pop-call-preserves-p-loadablep
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'pop-call**)
 ∧ proper-p-statep (p)
 ∧ p-loadablep (p, *load-addr*)
 ∧ p-pop-call-okp (p-current-instruction (p), p))
→ p-loadablep (p-pop-call-step (p-current-instruction (p), p), *load-addr*)

THEOREM: ret-preserves-p-loadablep
((p-psw (p) = **'run**)
 ∧ (car (p-current-instruction (p)) = **'ret**)
 ∧ proper-p-statep (p)
 ∧ p-loadablep (p, *load-addr*)
 ∧ p-ret-okp (p-current-instruction (p), p))
→ p-loadablep (p-ret-step (p-current-instruction (p), p), *load-addr*)

```
; We now turn our attention to putting all these lemmas together to
; get the p-step and r-step versions of each.
```

EVENT: Disable p-call-okp.

EVENT: Disable p-call-step.

EVENT: Disable p-ret-okp.

EVENT: Disable p-ret-step.

EVENT: Disable p-locn-okp.

EVENT: Disable p-locn-step.

EVENT: Disable p-push-constant-okp.

EVENT: Disable p-push-constant-step.

EVENT: Disable p-push-local-okp.

EVENT: Disable p-push-local-step.

EVENT: Disable p-push-global-okp.

EVENT: Disable p-push-global-step.

EVENT: Disable p-push-ctrl-stk-free-size-okp.

EVENT: Disable p-push-ctrl-stk-free-size-step.

EVENT: Disable p-push-temp-stk-free-size-okp.

EVENT: Disable p-push-temp-stk-free-size-step.

EVENT: Disable p-push-temp-stk-index-okp.

EVENT: Disable p-push-temp-stk-index-step.

EVENT: Disable p-jump-if-temp-stk-full-okp.

EVENT: Disable p-jump-if-temp-stk-full-step.

EVENT: Disable p-jump-if-temp-stk-empty-okp.

EVENT: Disable p-jump-if-temp-stk-empty-step.

EVENT: Disable p-pop-okp.

EVENT: Disable p-pop-step.

EVENT: Disable p-pop*-okp.

EVENT: Disable p-pop*-step.

EVENT: Disable p-popn-okp.

EVENT: Disable p-popn-step.

EVENT: Disable p-pop-local-okp.

EVENT: Disable p-pop-local-step.

EVENT: Disable p-pop-global-okp.

EVENT: Disable p-pop-global-step.

EVENT: Disable p-pop-locn-okp.

EVENT: Disable p-pop-locn-step.

EVENT: Disable p-pop-call-okp.

EVENT: Disable p-pop-call-step.

EVENT: Disable p-fetch-temp-stk-okp.

EVENT: Disable p-fetch-temp-stk-step.

EVENT: Disable p-deposit-temp-stk-okp.

EVENT: Disable p-deposit-temp-stk-step.

EVENT: Disable p-jump-okp.

EVENT: Disable p-jump-step.

EVENT: Disable p-jump-case-okp.

EVENT: Disable p-jump-case-step.

EVENT: Disable p-pushj-okp.

EVENT: Disable p-pushj-step.

EVENT: Disable p-popj-okp.

EVENT: Disable p-popj-step.

EVENT: Disable p-set-local-okp.

EVENT: Disable p-set-local-step.

EVENT: Disable p-set-global-okp.

EVENT: Disable p-set-global-step.

EVENT: Disable p-test-nat-and-jump-okp.

EVENT: Disable p-test-nat-and-jump-step.

EVENT: Disable p-test-int-and-jump-okp.

EVENT: Disable p-test-int-and-jump-step.

EVENT: Disable p-test-bool-and-jump-okp.

EVENT: Disable p-test-bool-and-jump-step.

EVENT: Disable p-test-bitv-and-jump-okp.

EVENT: Disable p-test-bitv-and-jump-step.

EVENT: Disable p-no-op-okp.

EVENT: Disable p-no-op-step.

EVENT: Disable p-add-addr-okp.

EVENT: Disable p-add-addr-step.

EVENT: Disable p-sub-addr-okp.

EVENT: Disable p-sub-addr-step.

EVENT: Disable p-eq-okp.

EVENT: Disable p-eq-step.

EVENT: Disable p-lt-addr-okp.

EVENT: Disable p-lt-addr-step.

EVENT: Disable p-fetch-okp.

EVENT: Disable p-fetch-step.

EVENT: Disable p-deposit-okp.

EVENT: Disable p-deposit-step.

EVENT: Disable p-add-int-okp.

EVENT: Disable p-add-int-step.

EVENT: Disable p-add-int-with-carry-okp.

EVENT: Disable p-add-int-with-carry-step.

EVENT: Disable p-add1-int-okp.

Event: Disable p-add1-int-step.

Event: Disable p-sub-int-okp.

Event: Disable p-sub-int-step.

Event: Disable p-sub-int-with-carry-okp.

Event: Disable p-sub-int-with-carry-step.

Event: Disable p-sub1-int-okp.

Event: Disable p-sub1-int-step.

Event: Disable p-neg-int-okp.

Event: Disable p-neg-int-step.

Event: Disable p-lt-int-okp.

Event: Disable p-lt-int-step.

Event: Disable p-int-to-nat-okp.

Event: Disable p-int-to-nat-step.

Event: Disable p-add-nat-okp.

Event: Disable p-add-nat-step.

Event: Disable p-add-nat-with-carry-okp.

Event: Disable p-add-nat-with-carry-step.

Event: Disable p-add1-nat-okp.

EVENT: Disable p-add1-nat-step.

EVENT: Disable p-sub-nat-okp.

EVENT: Disable p-sub-nat-step.

EVENT: Disable p-sub-nat-with-carry-okp.

EVENT: Disable p-sub-nat-with-carry-step.

EVENT: Disable p-sub1-nat-okp.

EVENT: Disable p-sub1-nat-step.

EVENT: Disable p-lt-nat-okp.

EVENT: Disable p-lt-nat-step.

EVENT: Disable p-mult2-nat-okp.

EVENT: Disable p-mult2-nat-step.

EVENT: Disable p-mult2-nat-with-carry-out-okp.

EVENT: Disable p-mult2-nat-with-carry-out-step.

EVENT: Disable p-div2-nat-okp.

EVENT: Disable p-div2-nat-step.

EVENT: Disable p-or-bitv-okp.

EVENT: Disable p-or-bitv-step.

EVENT: Disable p-and-bitv-okp.

EVENT: Disable p-and-bitv-step.

EVENT: Disable p-not-bitv-okp.

EVENT: Disable p-not-bitv-step.

EVENT: Disable p-xor-bitv-okp.

EVENT: Disable p-xor-bitv-step.

EVENT: Disable p-rsh-bitv-okp.

EVENT: Disable p-rsh-bitv-step.

EVENT: Disable p-lsh-bitv-okp.

EVENT: Disable p-lsh-bitv-step.

EVENT: Disable p-or-bool-okp.

EVENT: Disable p-or-bool-step.

EVENT: Disable p-and-bool-okp.

EVENT: Disable p-and-bool-step.

EVENT: Disable p-not-bool-okp.

EVENT: Disable p-not-bool-step.

EVENT: Disable r-call-okp.

EVENT: Disable r-call-step.

EVENT: Disable r-ret-okp.

EVENT: Disable r-ret-step.

EVENT: Disable r-locn-okp.

EVENT: Disable r-locn-step.

EVENT: Disable r-push-constant-okp.

EVENT: Disable r-push-constant-step.

EVENT: Disable r-push-local-okp.

EVENT: Disable r-push-local-step.

EVENT: Disable r-push-global-okp.

EVENT: Disable r-push-global-step.

EVENT: Disable r-push-ctrl-stk-free-size-okp.

EVENT: Disable r-push-ctrl-stk-free-size-step.

EVENT: Disable r-push-temp-stk-free-size-okp.

EVENT: Disable r-push-temp-stk-free-size-step.

EVENT: Disable r-push-temp-stk-index-okp.

EVENT: Disable r-push-temp-stk-index-step.

EVENT: Disable r-jump-if-temp-stk-full-okp.

EVENT: Disable r-jump-if-temp-stk-full-step.

EVENT: Disable r-jump-if-temp-stk-empty-okp.

EVENT: Disable r-jump-if-temp-stk-empty-step.

EVENT: Disable r-pop-okp.

EVENT: Disable r-pop-step.

EVENT: Disable r-pop*-okp.

EVENT: Disable r-pop*-step.

EVENT: Disable r-popn-okp.

EVENT: Disable r-popn-step.

EVENT: Disable r-pop-local-okp.

EVENT: Disable r-pop-local-step.

EVENT: Disable r-pop-global-okp.

EVENT: Disable r-pop-global-step.

EVENT: Disable r-pop-locn-okp.

EVENT: Disable r-pop-locn-step.

EVENT: Disable r-pop-call-okp.

EVENT: Disable r-pop-call-step.

EVENT: Disable r-fetch-temp-stk-okp.

EVENT: Disable r-fetch-temp-stk-step.

EVENT: Disable r-deposit-temp-stk-okp.

EVENT: Disable r-deposit-temp-stk-step.

EVENT: Disable r-jump-okp.

EVENT: Disable r-jump-step.

EVENT: Disable r-jump-case-okp.

EVENT: Disable r-jump-case-step.

EVENT: Disable r-pushj-okp.

EVENT: Disable r-pushj-step.

EVENT: Disable r-popj-okp.

EVENT: Disable r-popj-step.

EVENT: Disable r-set-local-okp.

EVENT: Disable r-set-local-step.

EVENT: Disable r-set-global-okp.

EVENT: Disable r-set-global-step.

EVENT: Disable r-test-nat-and-jump-okp.

EVENT: Disable r-test-nat-and-jump-step.

EVENT: Disable r-test-int-and-jump-okp.

EVENT: Disable r-test-int-and-jump-step.

EVENT: Disable r-test-bool-and-jump-okp.

EVENT: Disable r-test-bool-and-jump-step.

EVENT: Disable r-test-bitv-and-jump-okp.

EVENT: Disable r-test-bitv-and-jump-step.

EVENT: Disable r-no-op-okp.

EVENT: Disable r-no-op-step.

EVENT: Disable r-add-addr-okp.

EVENT: Disable r-add-addr-step.

EVENT: Disable r-sub-addr-okp.

EVENT: Disable r-sub-addr-step.

EVENT: Disable r-eq-okp.

EVENT: Disable r-eq-step.

EVENT: Disable r-lt-addr-okp.

EVENT: Disable r-lt-addr-step.

EVENT: Disable r-fetch-okp.

EVENT: Disable r-fetch-step.

EVENT: Disable r-deposit-okp.

EVENT: Disable r-deposit-step.

EVENT: Disable r-add-int-okp.

EVENT: Disable r-add-int-step.

EVENT: Disable r-add-int-with-carry-okp.

EVENT: Disable r-add-int-with-carry-step.

EVENT: Disable r-add1-int-okp.

EVENT: Disable r-add1-int-step.

EVENT: Disable r-sub-int-okp.

EVENT: Disable r-sub-int-step.

EVENT: Disable r-sub-int-with-carry-okp.

EVENT: Disable r-sub-int-with-carry-step.

EVENT: Disable r-sub1-int-okp.

EVENT: Disable r-sub1-int-step.

EVENT: Disable r-neg-int-okp.

EVENT: Disable r-neg-int-step.

EVENT: Disable r-lt-int-okp.

EVENT: Disable r-lt-int-step.

EVENT: Disable r-int-to-nat-okp.

EVENT: Disable r-int-to-nat-step.

EVENT: Disable r-add-nat-okp.

EVENT: Disable r-add-nat-step.

EVENT: Disable r-add-nat-with-carry-okp.

EVENT: Disable r-add-nat-with-carry-step.

EVENT: Disable r-add1-nat-okp.

EVENT: Disable r-add1-nat-step.

EVENT: Disable r-sub-nat-okp.

EVENT: Disable r-sub-nat-step.

EVENT: Disable r-sub-nat-with-carry-okp.

EVENT: Disable r-sub-nat-with-carry-step.

EVENT: Disable r-sub1-nat-okp.

EVENT: Disable r-sub1-nat-step.

EVENT: Disable r-lt-nat-okp.

EVENT: Disable r-lt-nat-step.

EVENT: Disable r-mult2-nat-okp.

EVENT: Disable r-mult2-nat-step.

EVENT: Disable r-mult2-nat-with-carry-out-okp.

EVENT: Disable r-mult2-nat-with-carry-out-step.

EVENT: Disable r-div2-nat-okp.

EVENT: Disable r-div2-nat-step.

EVENT: Disable r-or-bitv-okp.

EVENT: Disable r-or-bitv-step.

EVENT: Disable r-and-bitv-okp.

EVENT: Disable r-and-bitv-step.

EVENT: Disable r-not-bitv-okp.

EVENT: Disable r-not-bitv-step.

EVENT: Disable r-xor-bitv-okp.

EVENT: Disable r-xor-bitv-step.

EVENT: Disable r-rsh-bitv-okp.

EVENT: Disable r-rsh-bitv-step.

EVENT: Disable r-lsh-bitv-okp.

EVENT: Disable r-lsh-bitv-step.

EVENT: Disable r-or-bool-okp.

EVENT: Disable r-or-bool-step.

EVENT: Disable r-and-bool-okp.

EVENT: Disable r-and-bool-step.

EVENT: Disable r-not-bool-okp.

EVENT: Disable r-not-bool-step.

```
; Our plan for putting everything together requires -- as every plan must
; I think -- enough Lisp stack space for the rewriter to completely explore
; the if-nest in r-ins-step and the other "big switch" functions.
; Unfortunately that is not the case.  So we artificially partition our
; functions into two nearly equal halves and deal with each half separately.
```

DEFINITION:
r-ins-okp2 (ins, r)
= **case on** car (ins):
case = *eq*
then r-eq-okp (ins, r)
case = *lt-addr*
then r-lt-addr-okp (ins, r)
case = *fetch*
then r-fetch-okp (ins, r)
case = *deposit*
then r-deposit-okp (ins, r)
case = *add-int*
then r-add-int-okp (ins, r)
case = *add-int-with-carry*
then r-add-int-with-carry-okp (ins, r)
case = *add1-int*
then r-add1-int-okp (ins, r)
case = *sub-int*
then r-sub-int-okp (ins, r)
case = *sub-int-with-carry*
then r-sub-int-with-carry-okp (ins, r)
case = *sub1-int*
then r-sub1-int-okp (ins, r)
case = *neg-int*
then r-neg-int-okp (ins, r)
case = *lt-int*
then r-lt-int-okp (ins, r)
case = *int-to-nat*
then r-int-to-nat-okp (ins, r)
case = *add-nat*
then r-add-nat-okp (ins, r)
case = *add-nat-with-carry*
then r-add-nat-with-carry-okp (ins, r)
case = *add1-nat*
then r-add1-nat-okp (ins, r)

case $=$ *sub-nat*
  then r-sub-nat-okp $(ins,\ r)$
case $=$ *sub-nat-with-carry*
  then r-sub-nat-with-carry-okp $(ins,\ r)$
case $=$ *sub1-nat*
  then r-sub1-nat-okp $(ins,\ r)$
case $=$ *lt-nat*
  then r-lt-nat-okp $(ins,\ r)$
case $=$ *mult2-nat*
  then r-mult2-nat-okp $(ins,\ r)$
case $=$ *mult2-nat-with-carry-out*
  then r-mult2-nat-with-carry-out-okp $(ins,\ r)$
case $=$ *div2-nat*
  then r-div2-nat-okp $(ins,\ r)$
case $=$ *or-bitv*
  then r-or-bitv-okp $(ins,\ r)$
case $=$ *and-bitv*
  then r-and-bitv-okp $(ins,\ r)$
case $=$ *not-bitv*
  then r-not-bitv-okp $(ins,\ r)$
case $=$ *xor-bitv*
  then r-xor-bitv-okp $(ins,\ r)$
case $=$ *rsh-bitv*
  then r-rsh-bitv-okp $(ins,\ r)$
case $=$ *lsh-bitv*
  then r-lsh-bitv-okp $(ins,\ r)$
case $=$ *or-bool*
  then r-or-bool-okp $(ins,\ r)$
case $=$ *and-bool*
  then r-and-bool-okp $(ins,\ r)$
case $=$ *not-bool*
  then r-not-bool-okp $(ins,\ r)$
otherwise f endcase

DEFINITION:
r-ins-okp1 $(ins,\ r)$
$=$ **case on** car (ins):
case $=$ *call*
then r-call-okp $(ins,\ r)$
case $=$ *ret*
  then r-ret-okp $(ins,\ r)$
case $=$ *locn*
  then r-locn-okp $(ins,\ r)$
case $=$ *push-constant*

then r-push-constant-okp $(ins,\ r)$
case = *push-local*
then r-push-local-okp $(ins,\ r)$
case = *push-global*
then r-push-global-okp $(ins,\ r)$
case = *push-ctrl-stk-free-size*
then r-push-ctrl-stk-free-size-okp $(ins,\ r)$
case = *push-temp-stk-free-size*
then r-push-temp-stk-free-size-okp $(ins,\ r)$
case = *push-temp-stk-index*
then r-push-temp-stk-index-okp $(ins,\ r)$
case = *jump-if-temp-stk-full*
then r-jump-if-temp-stk-full-okp $(ins,\ r)$
case = *jump-if-temp-stk-empty*
then r-jump-if-temp-stk-empty-okp $(ins,\ r)$
case = *pop*
then r-pop-okp $(ins,\ r)$
case = *pop**
then r-pop*-okp $(ins,\ r)$
case = *popn*
then r-popn-okp $(ins,\ r)$
case = *pop-local*
then r-pop-local-okp $(ins,\ r)$
case = *pop-global*
then r-pop-global-okp $(ins,\ r)$
case = *pop-locn*
then r-pop-locn-okp $(ins,\ r)$
case = *pop-call*
then r-pop-call-okp $(ins,\ r)$
case = *fetch-temp-stk*
then r-fetch-temp-stk-okp $(ins,\ r)$
case = *deposit-temp-stk*
then r-deposit-temp-stk-okp $(ins,\ r)$
case = *jump*
then r-jump-okp $(ins,\ r)$
case = *jump-case*
then r-jump-case-okp $(ins,\ r)$
case = *pushj*
then r-pushj-okp $(ins,\ r)$
case = *popj*
then r-popj-okp $(ins,\ r)$
case = *set-local*
then r-set-local-okp $(ins,\ r)$
case = *set-global*

then r-set-global-okp (*ins*, *r*)
case = *test-nat-and-jump*
then r-test-nat-and-jump-okp (*ins*, *r*)
case = *test-int-and-jump*
then r-test-int-and-jump-okp (*ins*, *r*)
case = *test-bool-and-jump*
then r-test-bool-and-jump-okp (*ins*, *r*)
case = *test-bitv-and-jump*
then r-test-bitv-and-jump-okp (*ins*, *r*)
case = *no-op*
then r-no-op-okp (*ins*, *r*)
case = *add-addr*
then r-add-addr-okp (*ins*, *r*)
case = *sub-addr*
then r-sub-addr-okp (*ins*, *r*)
otherwise r-ins-okp2 (*ins*, *r*) **endcase**

THEOREM: r-ins-okp-is-r-ins-okp1
r-ins-okp (*ins*, *r*) = r-ins-okp1 (*ins*, *r*)

EVENT: Disable r-ins-okp.

EVENT: Disable r-ins-okp-is-r-ins-okp1.

```
; We repeat the exercise for r-ins-step.
```

EVENT: Enable r-ins-step-is-r-ins-step1.

EVENT: Disable r-ins-step.

EVENT: Disable r-ins-step-is-r-ins-step1.

DEFINITION:
p-ins-okp2 (*ins*, *p*)
= **case on** car (*ins*):
case = *eq*
then p-eq-okp (*ins*, *p*)
case = *lt-addr*
then p-lt-addr-okp (*ins*, *p*)
case = *fetch*
then p-fetch-okp (*ins*, *p*)

case = *deposit*
 then p-deposit-okp (ins, p)
case = *add-int*
 then p-add-int-okp (ins, p)
case = *add-int-with-carry*
 then p-add-int-with-carry-okp (ins, p)
case = *add1-int*
 then p-add1-int-okp (ins, p)
case = *sub-int*
 then p-sub-int-okp (ins, p)
case = *sub-int-with-carry*
 then p-sub-int-with-carry-okp (ins, p)
case = *sub1-int*
 then p-sub1-int-okp (ins, p)
case = *neg-int*
 then p-neg-int-okp (ins, p)
case = *lt-int*
 then p-lt-int-okp (ins, p)
case = *int-to-nat*
 then p-int-to-nat-okp (ins, p)
case = *add-nat*
 then p-add-nat-okp (ins, p)
case = *add-nat-with-carry*
 then p-add-nat-with-carry-okp (ins, p)
case = *add1-nat*
 then p-add1-nat-okp (ins, p)
case = *sub-nat*
 then p-sub-nat-okp (ins, p)
case = *sub-nat-with-carry*
 then p-sub-nat-with-carry-okp (ins, p)
case = *sub1-nat*
 then p-sub1-nat-okp (ins, p)
case = *lt-nat*
 then p-lt-nat-okp (ins, p)
case = *mult2-nat*
 then p-mult2-nat-okp (ins, p)
case = *mult2-nat-with-carry-out*
 then p-mult2-nat-with-carry-out-okp (ins, p)
case = *div2-nat*
 then p-div2-nat-okp (ins, p)
case = *or-bitv*
 then p-or-bitv-okp (ins, p)
case = *and-bitv*
 then p-and-bitv-okp (ins, p)

case $=$ *not-bitv*
then p-not-bitv-okp $(ins,\ p)$
case $=$ *xor-bitv*
then p-xor-bitv-okp $(ins,\ p)$
case $=$ *rsh-bitv*
then p-rsh-bitv-okp $(ins,\ p)$
case $=$ *lsh-bitv*
then p-lsh-bitv-okp $(ins,\ p)$
case $=$ *or-bool*
then p-or-bool-okp $(ins,\ p)$
case $=$ *and-bool*
then p-and-bool-okp $(ins,\ p)$
case $=$ *not-bool*
then p-not-bool-okp $(ins,\ p)$
otherwise f endcase

DEFINITION:
p-ins-okp1 $(ins,\ p)$
$=$ **case on** car (ins):
case $=$ *call*
then p-call-okp $(ins,\ p)$
case $=$ *ret*
then p-ret-okp $(ins,\ p)$
case $=$ *locn*
then p-locn-okp $(ins,\ p)$
case $=$ *push-constant*
then p-push-constant-okp $(ins,\ p)$
case $=$ *push-local*
then p-push-local-okp $(ins,\ p)$
case $=$ *push-global*
then p-push-global-okp $(ins,\ p)$
case $=$ *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-okp $(ins,\ p)$
case $=$ *push-temp-stk-free-size*
then p-push-temp-stk-free-size-okp $(ins,\ p)$
case $=$ *push-temp-stk-index*
then p-push-temp-stk-index-okp $(ins,\ p)$
case $=$ *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-okp $(ins,\ p)$
case $=$ *jump-if-temp-stk-empty*
then p-jump-if-temp-stk-empty-okp $(ins,\ p)$
case $=$ *pop*
then p-pop-okp $(ins,\ p)$
case $=$ *pop**

then p-pop*-okp (ins, p)
case = *popn*
then p-popn-okp (ins, p)
case = *pop-local*
then p-pop-local-okp (ins, p)
case = *pop-global*
then p-pop-global-okp (ins, p)
case = *pop-locn*
then p-pop-locn-okp (ins, p)
case = *pop-call*
then p-pop-call-okp (ins, p)
case = *fetch-temp-stk*
then p-fetch-temp-stk-okp (ins, p)
case = *deposit-temp-stk*
then p-deposit-temp-stk-okp (ins, p)
case = *jump*
then p-jump-okp (ins, p)
case = *jump-case*
then p-jump-case-okp (ins, p)
case = *pushj*
then p-pushj-okp (ins, p)
case = *popj*
then p-popj-okp (ins, p)
case = *set-local*
then p-set-local-okp (ins, p)
case = *set-global*
then p-set-global-okp (ins, p)
case = *test-nat-and-jump*
then p-test-nat-and-jump-okp (ins, p)
case = *test-int-and-jump*
then p-test-int-and-jump-okp (ins, p)
case = *test-bool-and-jump*
then p-test-bool-and-jump-okp (ins, p)
case = *test-bitv-and-jump*
then p-test-bitv-and-jump-okp (ins, p)
case = *no-op*
then p-no-op-okp (ins, p)
case = *add-addr*
then p-add-addr-okp (ins, p)
case = *sub-addr*
then p-sub-addr-okp (ins, p)
otherwise p-ins-okp2 (ins, p) **endcase**

THEOREM: p-ins-okp-is-p-ins-okp1

p-ins-okp (ins, p) = p-ins-okp1 (ins, p)

EVENT: Disable p-ins-okp.

EVENT: Disable p-ins-okp-is-p-ins-okp1.

DEFINITION:
p-ins-step2 (ins, p)
= **case on** car (ins):
case = *eq*
then p-eq-step (ins, p)
case = *lt-addr*
then p-lt-addr-step (ins, p)
case = *fetch*
then p-fetch-step (ins, p)
case = *deposit*
then p-deposit-step (ins, p)
case = *add-int*
then p-add-int-step (ins, p)
case = *add-int-with-carry*
then p-add-int-with-carry-step (ins, p)
case = *add1-int*
then p-add1-int-step (ins, p)
case = *sub-int*
then p-sub-int-step (ins, p)
case = *sub-int-with-carry*
then p-sub-int-with-carry-step (ins, p)
case = *sub1-int*
then p-sub1-int-step (ins, p)
case = *neg-int*
then p-neg-int-step (ins, p)
case = *lt-int*
then p-lt-int-step (ins, p)
case = *int-to-nat*
then p-int-to-nat-step (ins, p)
case = *add-nat*
then p-add-nat-step (ins, p)
case = *add-nat-with-carry*
then p-add-nat-with-carry-step (ins, p)
case = *add1-nat*
then p-add1-nat-step (ins, p)
case = *sub-nat*
then p-sub-nat-step (ins, p)

case = *sub-nat-with-carry*
then p-sub-nat-with-carry-step (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-step (*ins*, *p*)
case = *lt-nat*
then p-lt-nat-step (*ins*, *p*)
case = *mult2-nat*
then p-mult2-nat-step (*ins*, *p*)
case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-step (*ins*, *p*)
case = *div2-nat*
then p-div2-nat-step (*ins*, *p*)
case = *or-bitv*
then p-or-bitv-step (*ins*, *p*)
case = *and-bitv*
then p-and-bitv-step (*ins*, *p*)
case = *not-bitv*
then p-not-bitv-step (*ins*, *p*)
case = *xor-bitv*
then p-xor-bitv-step (*ins*, *p*)
case = *rsh-bitv*
then p-rsh-bitv-step (*ins*, *p*)
case = *lsh-bitv*
then p-lsh-bitv-step (*ins*, *p*)
case = *or-bool*
then p-or-bool-step (*ins*, *p*)
case = *and-bool*
then p-and-bool-step (*ins*, *p*)
case = *not-bool*
then p-not-bool-step (*ins*, *p*)
otherwise p-halt (*p*, **'run**) **endcase**

DEFINITION:
p-ins-step1 (*ins*, *p*)
= **case on** car (*ins*):
case = *call*
then p-call-step (*ins*, *p*)
case = *ret*
then p-ret-step (*ins*, *p*)
case = *locn*
then p-locn-step (*ins*, *p*)
case = *push-constant*
then p-push-constant-step (*ins*, *p*)
case = *push-local*

then p-push-local-step (ins, p)
case = *push-global*
then p-push-global-step (ins, p)
case = *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-step (ins, p)
case = *push-temp-stk-free-size*
then p-push-temp-stk-free-size-step (ins, p)
case = *push-temp-stk-index*
then p-push-temp-stk-index-step (ins, p)
case = *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-step (ins, p)
case = *jump-if-temp-stk-empty*
then p-jump-if-temp-stk-empty-step (ins, p)
case = *pop*
then p-pop-step (ins, p)
case = *pop**
then p-pop*-step (ins, p)
case = *popn*
then p-popn-step (ins, p)
case = *pop-local*
then p-pop-local-step (ins, p)
case = *pop-global*
then p-pop-global-step (ins, p)
case = *pop-locn*
then p-pop-locn-step (ins, p)
case = *pop-call*
then p-pop-call-step (ins, p)
case = *fetch-temp-stk*
then p-fetch-temp-stk-step (ins, p)
case = *deposit-temp-stk*
then p-deposit-temp-stk-step (ins, p)
case = *jump*
then p-jump-step (ins, p)
case = *jump-case*
then p-jump-case-step (ins, p)
case = *pushj*
then p-pushj-step (ins, p)
case = *popj*
then p-popj-step (ins, p)
case = *set-local*
then p-set-local-step (ins, p)
case = *set-global*
then p-set-global-step (ins, p)
case = *test-nat-and-jump*

then p-test-nat-and-jump-step $(ins,\ p)$
case $=$ *test-int-and-jump*
then p-test-int-and-jump-step $(ins,\ p)$
case $=$ *test-bool-and-jump*
then p-test-bool-and-jump-step $(ins,\ p)$
case $=$ *test-bitv-and-jump*
then p-test-bitv-and-jump-step $(ins,\ p)$
case $=$ *no-op*
then p-no-op-step $(ins,\ p)$
case $=$ *add-addr*
then p-add-addr-step $(ins,\ p)$
case $=$ *sub-addr*
then p-sub-addr-step $(ins,\ p)$
otherwise p-ins-step2 $(ins,\ p)$ **endcase**

THEOREM: p-ins-step-is-p-ins-step1
p-ins-step $(ins,\ p)$ $=$ p-ins-step1 $(ins,\ p)$

EVENT: Disable p-ins-step.

EVENT: Disable p-ins-step-is-p-ins-step1.

```
; We disable p-ins-okp because the main owc theorem involves p-ins-okp
; and p-ins-step, and opening either is sufficient to drive the case
; analysis.  If both open, then we get cross-multiplied cases and
; there is a lot of silly propositional work to do.  So we arbitrarily
; choose to keep p-ins-okp disabled.  Now we could choose to rewrite
; it to p-xxx-okp when the car of ins is 'xxx, analogously to what we
; did for r-ins-okp above.  But the proof of the owc should be faster
; if we do this: keep p-ins-okp disabled and show that p-xxx-okp holds
; if the car of ins is 'xxx.  The reason is that when we fire the
; xxx-one-way-correspondence-p-r lemma we will want to get p-xxx-okp
; and we'll be governed by a p-ins-okp and a hyp about car ins.  But
; the hyp about car ins won't have been raised all the way to the top
; and so won't yet be available to tell the p-ins-okp which way to go.
; This is the same strategy used for r-ins-okp in the r-i level proof.
```

THEOREM: p-ins-okp2-backchainer
p-ins-okp2 $(ins,\ p)$
$\rightarrow$ $(((\mathrm{car}\,(ins) = \texttt{'eq}) \rightarrow \text{p-eq-okp}\,(ins,\ p))$
$\wedge$ $((\mathrm{car}\,(ins) = \texttt{'lt-addr}) \rightarrow \text{p-lt-addr-okp}\,(ins,\ p))$
$\wedge$ $((\mathrm{car}\,(ins) = \texttt{'fetch}) \rightarrow \text{p-fetch-okp}\,(ins,\ p))$
$\wedge$ $((\mathrm{car}\,(ins) = \texttt{'deposit}) \rightarrow \text{p-deposit-okp}\,(ins,\ p))$

$\wedge$ ((car (ins) = 'add-int) → p-add-int-okp (ins, p))
$\wedge$ ((car (ins) = 'add-int-with-carry)
→ p-add-int-with-carry-okp (ins, p))
$\wedge$ ((car (ins) = 'add1-int) → p-add1-int-okp (ins, p))
$\wedge$ ((car (ins) = 'sub-int) → p-sub-int-okp (ins, p))
$\wedge$ ((car (ins) = 'sub-int-with-carry)
→ p-sub-int-with-carry-okp (ins, p))
$\wedge$ ((car (ins) = 'sub1-int) → p-sub1-int-okp (ins, p))
$\wedge$ ((car (ins) = 'neg-int) → p-neg-int-okp (ins, p))
$\wedge$ ((car (ins) = 'lt-int) → p-lt-int-okp (ins, p))
$\wedge$ ((car (ins) = 'int-to-nat) → p-int-to-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'add-nat) → p-add-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'add-nat-with-carry)
→ p-add-nat-with-carry-okp (ins, p))
$\wedge$ ((car (ins) = 'add1-nat) → p-add1-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'sub-nat) → p-sub-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'sub-nat-with-carry)
→ p-sub-nat-with-carry-okp (ins, p))
$\wedge$ ((car (ins) = 'sub1-nat) → p-sub1-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'lt-nat) → p-lt-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'mult2-nat) → p-mult2-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'mult2-nat-with-carry-out)
→ p-mult2-nat-with-carry-out-okp (ins, p))
$\wedge$ ((car (ins) = 'div2-nat) → p-div2-nat-okp (ins, p))
$\wedge$ ((car (ins) = 'or-bitv) → p-or-bitv-okp (ins, p))
$\wedge$ ((car (ins) = 'and-bitv) → p-and-bitv-okp (ins, p))
$\wedge$ ((car (ins) = 'not-bitv) → p-not-bitv-okp (ins, p))
$\wedge$ ((car (ins) = 'xor-bitv) → p-xor-bitv-okp (ins, p))
$\wedge$ ((car (ins) = 'rsh-bitv) → p-rsh-bitv-okp (ins, p))
$\wedge$ ((car (ins) = 'lsh-bitv) → p-lsh-bitv-okp (ins, p))
$\wedge$ ((car (ins) = 'or-bool) → p-or-bool-okp (ins, p))
$\wedge$ ((car (ins) = 'and-bool) → p-and-bool-okp (ins, p))
$\wedge$ ((car (ins) = 'not-bool) → p-not-bool-okp (ins, p)))

EVENT: Disable p-ins-okp2.

THEOREM: p-ins-okp-backchainer
p-ins-okp (ins, p)
→ (((car (ins) = 'call) → p-call-okp (ins, p))
$\wedge$ ((car (ins) = 'ret) → p-ret-okp (ins, p))
$\wedge$ ((car (ins) = 'locn) → p-locn-okp (ins, p))
$\wedge$ ((car (ins) = 'push-constant)
→ p-push-constant-okp (ins, p))
$\wedge$ ((car (ins) = 'push-local) → p-push-local-okp (ins, p))

$\wedge$ ((car (*ins*) = `'push-global`) $\rightarrow$ p-push-global-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'push-ctrl-stk-free-size`)
$\rightarrow$ p-push-ctrl-stk-free-size-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'push-temp-stk-free-size`)
$\rightarrow$ p-push-temp-stk-free-size-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'push-temp-stk-index`)
$\rightarrow$ p-push-temp-stk-index-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'jump-if-temp-stk-full`)
$\rightarrow$ p-jump-if-temp-stk-full-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'jump-if-temp-stk-empty`)
$\rightarrow$ p-jump-if-temp-stk-empty-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pop`) $\rightarrow$ p-pop-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pop*`) $\rightarrow$ p-pop*-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'popn`) $\rightarrow$ p-popn-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pop-local`) $\rightarrow$ p-pop-local-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pop-global`) $\rightarrow$ p-pop-global-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pop-locn`) $\rightarrow$ p-pop-locn-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pop-call`) $\rightarrow$ p-pop-call-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'fetch-temp-stk`)
$\rightarrow$ p-fetch-temp-stk-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'deposit-temp-stk`)
$\rightarrow$ p-deposit-temp-stk-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'jump`) $\rightarrow$ p-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'jump-case`) $\rightarrow$ p-jump-case-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'pushj`) $\rightarrow$ p-pushj-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'popj`) $\rightarrow$ p-popj-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'set-local`) $\rightarrow$ p-set-local-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'set-global`) $\rightarrow$ p-set-global-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'test-nat-and-jump`)
$\rightarrow$ p-test-nat-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'test-int-and-jump`)
$\rightarrow$ p-test-int-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'test-bool-and-jump`)
$\rightarrow$ p-test-bool-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'test-bitv-and-jump`)
$\rightarrow$ p-test-bitv-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'no-op`) $\rightarrow$ p-no-op-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'add-addr`) $\rightarrow$ p-add-addr-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = `'sub-addr`) $\rightarrow$ p-sub-addr-okp (*ins*, *p*)))

```
; The only problem with these backchainers is that we don't know that the
; above cases are exhaustive.  That is, suppose the current instruction
; is none of the above.  Then p-ins-okp is false.  We need to know that.
; The most natural expression of this fact is that if the instruction
```

```
; is not call and not ret and, etc., then p-ins-okp is false.  But that
; is an inefficient way to hang it, because every time we see p-ins-okp
; we'll try to show the instruction is none of the above and most of the time
; it will be one of the ones listed half-way through the list.  So we'll
; hang it on one of the instructions!  It reads: if p-ins-okp is true
; and the instruction is not call, not ret, etc., then the instruction
; is 'not-bool, except we split it into two lemmas.
```

THEOREM: p-ins-okp-exhausted
(p-ins-okp (*ins*, *p*)
∧ (car (*ins*) ≠ `'call`)
∧ (car (*ins*) ≠ `'ret`)
∧ (car (*ins*) ≠ `'locn`)
∧ (car (*ins*) ≠ `'push-constant`)
∧ (car (*ins*) ≠ `'push-local`)
∧ (car (*ins*) ≠ `'push-global`)
∧ (car (*ins*) ≠ `'push-ctrl-stk-free-size`)
∧ (car (*ins*) ≠ `'push-temp-stk-free-size`)
∧ (car (*ins*) ≠ `'push-temp-stk-index`)
∧ (car (*ins*) ≠ `'jump-if-temp-stk-full`)
∧ (car (*ins*) ≠ `'jump-if-temp-stk-empty`)
∧ (car (*ins*) ≠ `'pop`)
∧ (car (*ins*) ≠ `'pop*`)
∧ (car (*ins*) ≠ `'popn`)
∧ (car (*ins*) ≠ `'pop-local`)
∧ (car (*ins*) ≠ `'pop-global`)
∧ (car (*ins*) ≠ `'pop-locn`)
∧ (car (*ins*) ≠ `'pop-call`)
∧ (car (*ins*) ≠ `'fetch-temp-stk`)
∧ (car (*ins*) ≠ `'deposit-temp-stk`)
∧ (car (*ins*) ≠ `'jump`)
∧ (car (*ins*) ≠ `'jump-case`)
∧ (car (*ins*) ≠ `'pushj`)
∧ (car (*ins*) ≠ `'popj`)
∧ (car (*ins*) ≠ `'set-local`)
∧ (car (*ins*) ≠ `'set-global`)
∧ (car (*ins*) ≠ `'test-nat-and-jump`)
∧ (car (*ins*) ≠ `'test-int-and-jump`)
∧ (car (*ins*) ≠ `'test-bool-and-jump`)
∧ (car (*ins*) ≠ `'test-bitv-and-jump`)
∧ (car (*ins*) ≠ `'no-op`)
∧ (car (*ins*) ≠ `'add-addr`)
∧ (car (*ins*) ≠ `'sub-addr`))

$\rightarrow$ p-ins-okp2 $(ins,\ p)$

THEOREM: p-ins-okp2-exhausted
(p-ins-okp2 $(ins,\ p)$
$\wedge$ (car $(ins) \neq$ `'eq`)
$\wedge$ (car $(ins) \neq$ `'lt-addr`)
$\wedge$ (car $(ins) \neq$ `'fetch`)
$\wedge$ (car $(ins) \neq$ `'deposit`)
$\wedge$ (car $(ins) \neq$ `'add-int`)
$\wedge$ (car $(ins) \neq$ `'add-int-with-carry`)
$\wedge$ (car $(ins) \neq$ `'add1-int`)
$\wedge$ (car $(ins) \neq$ `'sub-int`)
$\wedge$ (car $(ins) \neq$ `'sub-int-with-carry`)
$\wedge$ (car $(ins) \neq$ `'sub1-int`)
$\wedge$ (car $(ins) \neq$ `'neg-int`)
$\wedge$ (car $(ins) \neq$ `'lt-int`)
$\wedge$ (car $(ins) \neq$ `'int-to-nat`)
$\wedge$ (car $(ins) \neq$ `'add-nat`)
$\wedge$ (car $(ins) \neq$ `'add-nat-with-carry`)
$\wedge$ (car $(ins) \neq$ `'add1-nat`)
$\wedge$ (car $(ins) \neq$ `'sub-nat`)
$\wedge$ (car $(ins) \neq$ `'sub-nat-with-carry`)
$\wedge$ (car $(ins) \neq$ `'sub1-nat`)
$\wedge$ (car $(ins) \neq$ `'lt-nat`)
$\wedge$ (car $(ins) \neq$ `'mult2-nat`)
$\wedge$ (car $(ins) \neq$ `'mult2-nat-with-carry-out`)
$\wedge$ (car $(ins) \neq$ `'div2-nat`)
$\wedge$ (car $(ins) \neq$ `'or-bitv`)
$\wedge$ (car $(ins) \neq$ `'and-bitv`)
$\wedge$ (car $(ins) \neq$ `'not-bitv`)
$\wedge$ (car $(ins) \neq$ `'xor-bitv`)
$\wedge$ (car $(ins) \neq$ `'rsh-bitv`)
$\wedge$ (car $(ins) \neq$ `'lsh-bitv`)
$\wedge$ (car $(ins) \neq$ `'or-bool`)
$\wedge$ (car $(ins) \neq$ `'and-bool`))
$\rightarrow$ ((car $(ins) =$ `'not-bool`) $=$ **t**)

```
; We will package the owc lemmas for each instruction into two lemmas
; about p-ins-okp v. r-ins-okp and p-ins-step v. r-ins-step.  Then we will
; put those together to get a p-step v. r-step lemma.

; Look at p-ins-okp-implies-r-ins-okp below.  We will let the r-ins-okp
; drive the case analysis, by enabling it during the proof.  It will
; open into an if, carrying the (p->r p) down into the maw of each of
```

```
; the n r-xxx-okps, each  governed by a hyp on the car of the current
; instruction.  When (r-xxx-okp (p-current-instruction p) (p->r p)) is
; formed, it will fire the p-xxx-okp-implies-r-xxx-okp lemma.  That will
; back chain to p-xxx-okp, under the hyp about the current instruction
; being 'xxx, and that will backchain into p-ins-okp by the lemma above.
; Every single case will be proved by appeal to one of our lemmas and
; all that will be left is proving that the case analysis done by r-ins-okp
; exhausts p-ins-okp.  But we already have that.  So one rewrite will
; prove the theorem.

; Of course, we want the following disabled:
```

EVENT: Disable p->r.

EVENT: Disable proper-p-statep-restructuring.

```
; And we need to split the proof into two manageable sized steps.
```

THEOREM: p-r-ins-okp2
((p-psw (p) = `'run`)
 ∧ proper-p-statep (p)
 ∧ p-ins-okp2 (p-current-instruction (p), p))
 → r-ins-okp2 (p-current-instruction (p), p->r (p))

THEOREM: p-r-ins-okp
((p-psw (p) = `'run`)
 ∧ proper-p-statep (p)
 ∧ p-ins-okp (p-current-instruction (p), p))
 → r-ins-okp (p-current-instruction (p), p->r (p))

```
; Now we move on to the p v. r ins-step case.

; Now consider p-r-ins-step-owc below.  We will let both the p-ins-step
; and the r-ins-step expand.  They both go to ghastly if expressions.
; With the following lemma we will bring the p-ins-step's ifs out of p->r:
```

THEOREM: p->r-if
p->r (**if** a **then** b
  else c **endif**)
= **if** a **then** p->r (b)
  else p->r (c) **endif**

```
; So now the r-equal has two if nests as its args.  If we let the theorem
; prover handle things it would cross multiply them.  But there is a simpler
; way because the two nests have identical tests in the same order.  So
; we can use:
```

THEOREM: r-equal-if
r-equal (**if** *op* **then** *r1*
else *r2* **endif**,
if *op* **then** *r3*
else *r4* **endif**)
= **if** *op* **then** r-equal (*r1*, *r3*)
else r-equal (*r2*, *r4*) **endif**

```
; Since r2 and r4 are themselves ifs, this will just drive on down.  At the
; bottom we will need:
```

THEOREM: stk-equal-reflexive
stk-equal (*sp*, *segment*, *segment*)

THEOREM: r-equal-reflexive
r-statep (r) $\rightarrow$ r-equal (r, r)

```
; Each of the branches of this new if will be an r-equality between
; (p->r (p-xxx-step ...)) and (r-xxx-step ... (p->r p)), governed by
; the hyp that the instruction is 'xxx.  That will fire our individual
; one way correspondence lemmas and backchain to (p-xxx-okp ...), which
; we'll get, as above, from the p-ins-okp.

; So once again, under these rules, the following theorem rewrites to
; T in one pass:
```

THEOREM: p-r-ins-step2-owc
((p-psw (p) = **'run**)
$\wedge$ proper-p-statep (p)
$\wedge$ p-ins-okp2 (p-current-instruction (p), p))
$\rightarrow$ r-equal (p->r (p-ins-step2 (p-current-instruction (p), p)),
r-ins-step2 (p-current-instruction (p), p->r (p)))

EVENT: Disable p-ins-step2.

EVENT: Disable r-ins-step2.

THEOREM: p-r-ins-step-owc
((p-psw (p) = **'run**)
 ∧ proper-p-statep (p)
 ∧ p-ins-okp (p-current-instruction (p), p))
→ r-equal (p->r (p-ins-step (p-current-instruction (p), p)),
 r-ins-step (p-current-instruction (p), p->r (p)))

```
; We now package the p-r-ins-okp and p-r-ins-step pair into p-r-step:

; We need to know a few things about the psw to get through r-step properly.
```

THEOREM: r-psw-p->r
r-psw (p->r (p)) = p-psw (p)

THEOREM: step-one-way-correspondence-p-r
(proper-p-statep (p) ∧ (¬ errorp (p-psw (p-step (p)))))
→ r-equal (p->r (p-step (p)), r-step (p->r (p)))

```
; So that completes the packaging of the owc lemmas.  Now we move on to
; the packaging of the proper-p-statep preservation lemmas.

; Recall that we proved the lemmas about ur-proper-p-statep.  So we
; take it in two steps.  First we gather all the lemmas into a single
; stepper concluding with ur-proper-p-statep.  Then we expose the true
; identity of ur-proper-p-statep.  As above, we deal first with p-ins-step.

; The strategy is to let p-ins-step open, bring the ifs through
; ur-proper-p-statep, and then hit each branch with the preservation
; lemma for the governing case.
```

THEOREM: ur-proper-p-statep-if
ur-proper-p-statep (**if** a **then** b
 else c **endif**)
= **if** a **then** ur-proper-p-statep (b)
 else ur-proper-p-statep (c) **endif**

THEOREM: p-ins-step2-preserves-ur-proper-p-statep
((p-psw (p) = **'run**)
 ∧ p-ins-okp2 (p-current-instruction (p), p)
 ∧ proper-p-statep (p))
→ ur-proper-p-statep (p-ins-step2 (p-current-instruction (p), p))

THEOREM: p-ins-step-preserves-ur-proper-p-statep

$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad \text{p-ins-okp}(\text{p-current-instruction}(p),\ p)$
$\land \quad \text{proper-p-statep}(p))$
$\to \quad \text{ur-proper-p-statep}(\text{p-ins-step}(\text{p-current-instruction}(p),\ p))$

```
; Now we show the true identity of ur-proper-p-statep.
```

THEOREM: p-ins-step-preserves-proper-p-statep
$((\text{p-psw}(p) = \texttt{'run})$
$\land \quad \text{p-ins-okp}(\text{p-current-instruction}(p),\ p)$
$\land \quad \text{proper-p-statep}(p))$
$\to \quad \text{proper-p-statep}(\text{p-ins-step}(\text{p-current-instruction}(p),\ p))$

```
; And now we package it up to the p-step level:
```

THEOREM: p-step-preserves-proper-p-statep
$(\text{proper-p-statep}(p) \land (\neg\ \text{errorp}(\text{p-psw}(\text{p-step}(p)))))$
$\to \quad \text{proper-p-statep}(\text{p-step}(p))$

```
; And that completes the packaging of the proper-p-statep preservation.

; Finally, we prove the preservation of r-equality by r-step.

; This whole exercise would be simpler if we did two things.
; (1) in all of the lemmas of the form r-equal-implies-equal-r-xxx-okp
; replace the concluding (equal lhs rhs) with (equal (equal lhs rhs) t).
; (2) in all of the lemmas r-xxx-step-preserves-r-equal, replace
; the (r-xxx-okp ... r2) hyp by (r-xxx-okp ... r1).  Then my
; concern about r-equal-reflexivity can be forgotten.  !!! except
; in the r-step lemma below -- the r-ins-okp lemma below must still
; be (equal lhs rhs) and so reflexivity must be disabled when
; I use that lemma.


; We need a backchainer for r-ins-okp analogous to that used for p-ins-okp
; above:
```

THEOREM: r-ins-okp2-backchainer
$\text{r-ins-okp2}(ins,\ r)$
$\to \quad (((\text{car}(ins) = \texttt{'eq}) \to \text{r-eq-okp}(ins,\ r))$
$\quad\quad \land \quad ((\text{car}(ins) = \texttt{'lt-addr}) \to \text{r-lt-addr-okp}(ins,\ r))$
$\quad\quad \land \quad ((\text{car}(ins) = \texttt{'fetch}) \to \text{r-fetch-okp}(ins,\ r))$

$\wedge$ ((car (*ins*) = `'deposit`) $\rightarrow$ r-deposit-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'add-int`) $\rightarrow$ r-add-int-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'add-int-with-carry`)
$\rightarrow$ r-add-int-with-carry-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'add1-int`) $\rightarrow$ r-add1-int-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'sub-int`) $\rightarrow$ r-sub-int-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'sub-int-with-carry`)
$\rightarrow$ r-sub-int-with-carry-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'sub1-int`) $\rightarrow$ r-sub1-int-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'neg-int`) $\rightarrow$ r-neg-int-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'lt-int`) $\rightarrow$ r-lt-int-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'int-to-nat`) $\rightarrow$ r-int-to-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'add-nat`) $\rightarrow$ r-add-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'add-nat-with-carry`)
$\rightarrow$ r-add-nat-with-carry-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'add1-nat`) $\rightarrow$ r-add1-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'sub-nat`) $\rightarrow$ r-sub-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'sub-nat-with-carry`)
$\rightarrow$ r-sub-nat-with-carry-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'sub1-nat`) $\rightarrow$ r-sub1-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'lt-nat`) $\rightarrow$ r-lt-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'mult2-nat`) $\rightarrow$ r-mult2-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'mult2-nat-with-carry-out`)
$\rightarrow$ r-mult2-nat-with-carry-out-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'div2-nat`) $\rightarrow$ r-div2-nat-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'or-bitv`) $\rightarrow$ r-or-bitv-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'and-bitv`) $\rightarrow$ r-and-bitv-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'not-bitv`) $\rightarrow$ r-not-bitv-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'xor-bitv`) $\rightarrow$ r-xor-bitv-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'rsh-bitv`) $\rightarrow$ r-rsh-bitv-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'lsh-bitv`) $\rightarrow$ r-lsh-bitv-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'or-bool`) $\rightarrow$ r-or-bool-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'and-bool`) $\rightarrow$ r-and-bool-okp (*ins*, *r*))
$\wedge$ ((car (*ins*) = `'not-bool`) $\rightarrow$ r-not-bool-okp (*ins*, *r*)))

EVENT: Disable r-ins-okp2.

EVENT: Enable r-ins-okp-backchainer.

THEOREM: r-ins-okp-exhausted
(r-ins-okp (*ins*, *r*)
$\wedge$ (car (*ins*) $\neq$ `'call`)
$\wedge$ (car (*ins*) $\neq$ `'ret`)

$\wedge$ (car (ins) $\neq$ `'locn`)
$\wedge$ (car (ins) $\neq$ `'push-constant`)
$\wedge$ (car (ins) $\neq$ `'push-local`)
$\wedge$ (car (ins) $\neq$ `'push-global`)
$\wedge$ (car (ins) $\neq$ `'push-ctrl-stk-free-size`)
$\wedge$ (car (ins) $\neq$ `'push-temp-stk-free-size`)
$\wedge$ (car (ins) $\neq$ `'push-temp-stk-index`)
$\wedge$ (car (ins) $\neq$ `'jump-if-temp-stk-full`)
$\wedge$ (car (ins) $\neq$ `'jump-if-temp-stk-empty`)
$\wedge$ (car (ins) $\neq$ `'pop`)
$\wedge$ (car (ins) $\neq$ `'pop*`)
$\wedge$ (car (ins) $\neq$ `'popn`)
$\wedge$ (car (ins) $\neq$ `'pop-local`)
$\wedge$ (car (ins) $\neq$ `'pop-global`)
$\wedge$ (car (ins) $\neq$ `'pop-locn`)
$\wedge$ (car (ins) $\neq$ `'pop-call`)
$\wedge$ (car (ins) $\neq$ `'fetch-temp-stk`)
$\wedge$ (car (ins) $\neq$ `'deposit-temp-stk`)
$\wedge$ (car (ins) $\neq$ `'jump`)
$\wedge$ (car (ins) $\neq$ `'jump-case`)
$\wedge$ (car (ins) $\neq$ `'pushj`)
$\wedge$ (car (ins) $\neq$ `'popj`)
$\wedge$ (car (ins) $\neq$ `'set-local`)
$\wedge$ (car (ins) $\neq$ `'set-global`)
$\wedge$ (car (ins) $\neq$ `'test-nat-and-jump`)
$\wedge$ (car (ins) $\neq$ `'test-int-and-jump`)
$\wedge$ (car (ins) $\neq$ `'test-bool-and-jump`)
$\wedge$ (car (ins) $\neq$ `'test-bitv-and-jump`)
$\wedge$ (car (ins) $\neq$ `'no-op`)
$\wedge$ (car (ins) $\neq$ `'add-addr`)
$\wedge$ (car (ins) $\neq$ `'sub-addr`))
$\rightarrow$ r-ins-okp2 (ins, r)

```
; Now we'll prove that r-equal implies r-ins-okp is indifferent to
; which state it gets.

; The plan is to let r-ins-okp expand into ifs on both sides of the
; equal and then take them out with:
```

THEOREM: equal-if
(**if** a **then** $r1$
else $r2$ **endif**
= **if** a **then** $r3$

else $r4$ **endif**)
$=$ **if** a **then** $r1 = r3$
else $r2 = r4$ **endif**

```
; The proof requires some care.  If you leave r-equal-reflexive enabled
; we infinitely rewrite (r-xxx-okp ins r1) to (r-xxx-okp ins r1)
; because we know r1 is r-equal to itself.
```

THEOREM: r-equal-implies-equal-r-ins-okp2
((r-psw ($r1$) $=$ **'run**)
$\land$ proper-r-statep ($r1$, *load-addr*)
$\land$ proper-r-statep ($r2$, *load-addr*)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-ins-okp2 (r-current-instruction ($r1$), $r2$)
$=$ r-ins-okp2 (r-current-instruction ($r1$), $r1$))

THEOREM: r-equal-implies-equal-r-ins-okp
((r-psw ($r1$) $=$ **'run**)
$\land$ proper-r-statep ($r1$, *load-addr*)
$\land$ proper-r-statep ($r2$, *load-addr*)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ (r-ins-okp (r-current-instruction ($r1$), $r2$)
$=$ r-ins-okp (r-current-instruction ($r1$), $r1$))

```
; Now we do the r-ins-step.

; We'll let r-ins-step open on both sides of the r-equal.  The r-equal-if
; lemma will drive the r-equal through the matching if-nests and we'll
; appeal to our previously proved preservation lemmas.
```

THEOREM: r-ins-step2-preserves-r-equal
((r-psw ($r1$) $=$ **'run**)
$\land$ proper-r-statep ($r1$, *load-addr*)
$\land$ proper-r-statep ($r2$, *load-addr*)
$\land$ r-ins-okp2 (r-current-instruction ($r1$), $r1$)
$\land$ r-equal ($r2$, $r1$))
$\rightarrow$ r-equal (r-ins-step2 (r-current-instruction ($r1$), $r2$),
r-ins-step2 (r-current-instruction ($r1$), $r1$))

THEOREM: r-ins-step-preserves-r-equal
((r-psw ($r1$) $=$ **'run**)
$\land$ proper-r-statep ($r1$, *load-addr*)
$\land$ proper-r-statep ($r2$, *load-addr*)

$\land$ r-ins-okp (r-current-instruction (*r1*), *r1*)
$\land$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-ins-step (r-current-instruction (*r1*), *r2*),
r-ins-step (r-current-instruction (*r1*), *r1*))

THEOREM: r-step-preserves-r-equal
(proper-r-statep (*r1*, *load-addr*)
$\land$ proper-r-statep (*r2*, *load-addr*)
$\land$ r-equal (*r2*, *r1*))
$\rightarrow$ r-equal (r-step (*r2*), r-step (*r1*))

```
; Now we package up the p-loadablep results into a p-step.
; The approach is exactly the same as used above.
```

THEOREM: p-loadablep-if
p-loadablep (**if** *a* **then** *b*
else *c* **endif**,
load-addr)
= **if** *a* **then** p-loadablep (*b*, *load-addr*)
else p-loadablep (*c*, *load-addr*) **endif**

THEOREM: p-ins-step2-preserves-p-loadablep
((p-psw (*p*) = **'run**)
$\land$ p-ins-okp2 (p-current-instruction (*p*), *p*)
$\land$ proper-p-statep (*p*)
$\land$ p-loadablep (*p*, *load-addr*))
$\rightarrow$ p-loadablep (p-ins-step2 (p-current-instruction (*p*), *p*), *load-addr*)

THEOREM: p-ins-step-preserves-p-loadablep
((p-psw (*p*) = **'run**)
$\land$ p-ins-okp (p-current-instruction (*p*), *p*)
$\land$ proper-p-statep (*p*)
$\land$ p-loadablep (*p*, *load-addr*))
$\rightarrow$ p-loadablep (p-ins-step (p-current-instruction (*p*), *p*), *load-addr*)

THEOREM: p-step-preserves-p-loadablep
(proper-p-statep (*p*)
$\land$ p-loadablep (*p*, *load-addr*)
$\land$ ($\neg$ errorp (p-psw (p-step (*p*)))))
$\rightarrow$ p-loadablep (p-step (*p*), *load-addr*)

```
;  Now we grab an event from the r-i proofs:

#|
```

```
(include-book r-i-book "r-i" t nil)

;  The following status expression has the effect of enabling only
;  proper-r-statep-r-step among all the other things brought in by
;  the above inclusion.  It also disables r-step.

(status r-i-book-status
        (set-difference
         (union (@ *e* p-step-preserves-p-loadablep)
                '(proper-r-statep-r-step))
         '(r-step)))

|#
```

EVENT: Enable proper-r-statep-r-step.

EVENT: Disable r-step.

THEOREM: r-preserves-r-equal
(proper-r-statep (*r1*, *load-addr*)
 ∧ proper-r-statep (*r2*, *load-addr*)
 ∧ r-equal (*r2*, *r1*))
→ r-equal (r (*r2*, *n*), r (*r1*, *n*))

THEOREM: transitivity-of-same-signature
(same-signature (*segment1*, *segment2*) ∧ same-signature (*segment2*, *segment3*))
→ same-signature (*segment1*, *segment3*)

THEOREM: transitivity-of-stk-equal
(stk-equal (*sp*, *segment1*, *segment2*)
 ∧ stk-equal (*sp*, *segment2*, *segment3*)
 ∧ (length (cdr (assoc (car (untag (*sp*)), *segment2*)))
   = length (cdr (assoc (car (untag (*sp*)), *segment1*))))
 ∧ (length (cdr (assoc (car (untag (*sp*)), *segment3*)))
   = length (cdr (assoc (car (untag (*sp*)), *segment1*)))))
→ stk-equal (*sp*, *segment1*, *segment3*)

THEOREM: transitivity-of-r-equal
(r-equal (*r1*, *r2*) ∧ r-equal (*r2*, *r3*)) → r-equal (*r1*, *r3*)

```
; We are now headed toward the proof that proper-p-statep maps down via
; p->r to a proper-r-statep.
```

```
; Proper temp-var-dcls at the p level implies the corresponding thing
; at the r level:
```

THEOREM: proper-p-temp-var-dclsp-implies-r-temp-var-dclsp
proper-p-temp-var-dclsp (lst, p) $\rightarrow$ r-temp-var-dclsp (lst)

```
; The surprisingly hard result is that proper programs at the p level
; implies proper programs at the r level.  We start with two
; observations: anything occuring as a label in a properly labeled
; list of instructions is a litatom; and to be a proper instruction at
; the p level means (among many other things) that the car is a piton
; opcode and push-constants satisfy the
; proper-p-push-constant-instructionp predicate.
```

THEOREM: proper-labeled-p-instructionsp-implies-litatom-labels
(proper-labeled-p-instructionsp (lst, $name$, p) $\land$ find-labelp (lab, lst))
$\rightarrow$ litatom (lab)

EVENT: Disable proper-p-call-instructionp.

EVENT: Disable proper-p-ret-instructionp.

EVENT: Disable proper-p-locn-instructionp.

EVENT: Disable proper-p-push-constant-instructionp.

EVENT: Disable proper-p-push-local-instructionp.

EVENT: Disable proper-p-push-global-instructionp.

EVENT: Disable proper-p-push-ctrl-stk-free-size-instructionp.

EVENT: Disable proper-p-push-temp-stk-free-size-instructionp.

EVENT: Disable proper-p-push-temp-stk-index-instructionp.

EVENT: Disable proper-p-jump-if-temp-stk-full-instructionp.

EVENT: Disable proper-p-jump-if-temp-stk-empty-instructionp.

EVENT: Disable proper-p-pop-instructionp.

EVENT: Disable proper-p-pop*-instructionp.

EVENT: Disable proper-p-popn-instructionp.

EVENT: Disable proper-p-pop-local-instructionp.

EVENT: Disable proper-p-pop-global-instructionp.

EVENT: Disable proper-p-pop-locn-instructionp.

EVENT: Disable proper-p-pop-call-instructionp.

EVENT: Disable proper-p-fetch-temp-stk-instructionp.

EVENT: Disable proper-p-deposit-temp-stk-instructionp.

EVENT: Disable proper-p-jump-instructionp.

EVENT: Disable proper-p-jump-case-instructionp.

EVENT: Disable proper-p-pushj-instructionp.

EVENT: Disable proper-p-popj-instructionp.

EVENT: Disable proper-p-set-local-instructionp.

EVENT: Disable proper-p-set-global-instructionp.

EVENT: Disable proper-p-test-nat-and-jump-instructionp.

EVENT: Disable proper-p-test-int-and-jump-instructionp.

EVENT: Disable proper-p-test-bool-and-jump-instructionp.

EVENT: Disable proper-p-test-bitv-and-jump-instructionp.

EVENT: Disable proper-p-no-op-instructionp.

EVENT: Disable proper-p-add-addr-instructionp.

EVENT: Disable proper-p-sub-addr-instructionp.

EVENT: Disable proper-p-eq-instructionp.

EVENT: Disable proper-p-lt-addr-instructionp.

EVENT: Disable proper-p-fetch-instructionp.

EVENT: Disable proper-p-deposit-instructionp.

EVENT: Disable proper-p-add-int-instructionp.

EVENT: Disable proper-p-add-int-with-carry-instructionp.

EVENT: Disable proper-p-add1-int-instructionp.

EVENT: Disable proper-p-sub-int-instructionp.

EVENT: Disable proper-p-sub-int-with-carry-instructionp.

EVENT: Disable proper-p-sub1-int-instructionp.

EVENT: Disable proper-p-neg-int-instructionp.

EVENT: Disable proper-p-lt-int-instructionp.

EVENT: Disable proper-p-int-to-nat-instructionp.

EVENT: Disable proper-p-add-nat-instructionp.

EVENT: Disable proper-p-add-nat-with-carry-instructionp.

EVENT: Disable proper-p-add1-nat-instructionp.

EVENT: Disable proper-p-sub-nat-instructionp.

EVENT: Disable proper-p-sub-nat-with-carry-instructionp.

EVENT: Disable proper-p-sub1-nat-instructionp.

EVENT: Disable proper-p-lt-nat-instructionp.

EVENT: Disable proper-p-mult2-nat-instructionp.

EVENT: Disable proper-p-mult2-nat-with-carry-out-instructionp.

EVENT: Disable proper-p-div2-nat-instructionp.

EVENT: Disable proper-p-or-bitv-instructionp.

EVENT: Disable proper-p-and-bitv-instructionp.

EVENT: Disable proper-p-not-bitv-instructionp.

EVENT: Disable proper-p-xor-bitv-instructionp.

EVENT: Disable proper-p-rsh-bitv-instructionp.

EVENT: Disable proper-p-lsh-bitv-instructionp.

EVENT: Disable proper-p-or-bool-instructionp.

EVENT: Disable proper-p-and-bool-instructionp.

EVENT: Disable proper-p-not-bool-instructionp.

```
; We split proper-p-instructionp into two parts, as usual with our
; big switches.
```

DEFINITION:
proper-p-instructionp2 (*ins*, *name*, *p*)
= **case on** car (*ins*):
case = *eq*
then proper-p-eq-instructionp (*ins*, *name*, *p*)
case = *lt-addr*
then proper-p-lt-addr-instructionp (*ins*, *name*, *p*)
case = *fetch*
then proper-p-fetch-instructionp (*ins*, *name*, *p*)
case = *deposit*
then proper-p-deposit-instructionp (*ins*, *name*, *p*)
case = *add-int*
then proper-p-add-int-instructionp (*ins*, *name*, *p*)
case = *add-int-with-carry*
then proper-p-add-int-with-carry-instructionp (*ins*, *name*, *p*)
case = *add1-int*
then proper-p-add1-int-instructionp (*ins*, *name*, *p*)
case = *sub-int*
then proper-p-sub-int-instructionp (*ins*, *name*, *p*)
case = *sub-int-with-carry*
then proper-p-sub-int-with-carry-instructionp (*ins*, *name*, *p*)
case = *sub1-int*
then proper-p-sub1-int-instructionp (*ins*, *name*, *p*)
case = *neg-int*
then proper-p-neg-int-instructionp (*ins*, *name*, *p*)
case = *lt-int*
then proper-p-lt-int-instructionp (*ins*, *name*, *p*)
case = *int-to-nat*
then proper-p-int-to-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat*
then proper-p-add-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat-with-carry*
then proper-p-add-nat-with-carry-instructionp (*ins*, *name*, *p*)
case = *add1-nat*
then proper-p-add1-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat*
then proper-p-sub-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat-with-carry*

then proper-p-sub-nat-with-carry-instructionp (*ins*, *name*, *p*)
case = *sub1-nat*
then proper-p-sub1-nat-instructionp (*ins*, *name*, *p*)
case = *lt-nat*
then proper-p-lt-nat-instructionp (*ins*, *name*, *p*)
case = *mult2-nat*
then proper-p-mult2-nat-instructionp (*ins*, *name*, *p*)
case = *mult2-nat-with-carry-out*
then proper-p-mult2-nat-with-carry-out-instructionp (*ins*, *name*, *p*)
case = *div2-nat*
then proper-p-div2-nat-instructionp (*ins*, *name*, *p*)
case = *or-bitv*
then proper-p-or-bitv-instructionp (*ins*, *name*, *p*)
case = *and-bitv*
then proper-p-and-bitv-instructionp (*ins*, *name*, *p*)
case = *not-bitv*
then proper-p-not-bitv-instructionp (*ins*, *name*, *p*)
case = *xor-bitv*
then proper-p-xor-bitv-instructionp (*ins*, *name*, *p*)
case = *rsh-bitv*
then proper-p-rsh-bitv-instructionp (*ins*, *name*, *p*)
case = *lsh-bitv*
then proper-p-lsh-bitv-instructionp (*ins*, *name*, *p*)
case = *or-bool*
then proper-p-or-bool-instructionp (*ins*, *name*, *p*)
case = *and-bool*
then proper-p-and-bool-instructionp (*ins*, *name*, *p*)
case = *not-bool*
then proper-p-not-bool-instructionp (*ins*, *name*, *p*)
otherwise f endcase

DEFINITION:
proper-p-instructionp1 (*ins*, *name*, *p*)
= (properp (*ins*)
∧ **case on** car (*ins*):
case = *call*
then proper-p-call-instructionp (*ins*, *name*, *p*)
case = *ret*
then proper-p-ret-instructionp (*ins*, *name*, *p*)
case = *locn*
then proper-p-locn-instructionp (*ins*, *name*, *p*)
case = *push-constant*
then proper-p-push-constant-instructionp (*ins*, *name*, *p*)
case = *push-local*

then proper-p-push-local-instructionp (ins, $name$, p)
case = *push-global*
then proper-p-push-global-instructionp (ins, $name$, p)
case = *push-ctrl-stk-free-size*
then proper-p-push-ctrl-stk-free-size-instructionp (ins,
$name$,
p)
case = *push-temp-stk-free-size*
then proper-p-push-temp-stk-free-size-instructionp (ins,
$name$,
p)
case = *push-temp-stk-index*
then proper-p-push-temp-stk-index-instructionp (ins, $name$, p)
case = *jump-if-temp-stk-full*
then proper-p-jump-if-temp-stk-full-instructionp (ins, $name$, p)
case = *jump-if-temp-stk-empty*
then proper-p-jump-if-temp-stk-empty-instructionp (ins,
$name$,
p)
case = *pop*
then proper-p-pop-instructionp (ins, $name$, p)
case = *pop**
then proper-p-pop*-instructionp (ins, $name$, p)
case = *popn*
then proper-p-popn-instructionp (ins, $name$, p)
case = *pop-local*
then proper-p-pop-local-instructionp (ins, $name$, p)
case = *pop-global*
then proper-p-pop-global-instructionp (ins, $name$, p)
case = *pop-locn*
then proper-p-pop-locn-instructionp (ins, $name$, p)
case = *pop-call*
then proper-p-pop-call-instructionp (ins, $name$, p)
case = *fetch-temp-stk*
then proper-p-fetch-temp-stk-instructionp (ins, $name$, p)
case = *deposit-temp-stk*
then proper-p-deposit-temp-stk-instructionp (ins, $name$, p)
case = *jump*
then proper-p-jump-instructionp (ins, $name$, p)
case = *jump-case*
then proper-p-jump-case-instructionp (ins, $name$, p)
case = *pushj*
then proper-p-pushj-instructionp (ins, $name$, p)
case = *popj*

then proper-p-popj-instructionp (*ins*, *name*, *p*)
case = *set-local*
then proper-p-set-local-instructionp (*ins*, *name*, *p*)
case = *set-global*
then proper-p-set-global-instructionp (*ins*, *name*, *p*)
case = *test-nat-and-jump*
then proper-p-test-nat-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-int-and-jump*
then proper-p-test-int-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-bool-and-jump*
then proper-p-test-bool-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-bitv-and-jump*
then proper-p-test-bitv-and-jump-instructionp (*ins*, *name*, *p*)
case = *no-op*
then proper-p-no-op-instructionp (*ins*, *name*, *p*)
case = *add-addr*
then proper-p-add-addr-instructionp (*ins*, *name*, *p*)
case = *sub-addr*
then proper-p-sub-addr-instructionp (*ins*, *name*, *p*)
otherwise proper-p-instructionp2 (*ins*, *name*, *p*) **endcase**)

THEOREM: proper-p-instructionp-is-proper-p-instructionp1
proper-p-instructionp (*ins*, *name*, *p*) = proper-p-instructionp1 (*ins*, *name*, *p*)

EVENT: Disable proper-p-instructionp-is-proper-p-instructionp1.

```
; The following theorem is the proper-p-instructionp2 analogoue of the
; next theorem, except that piton-opcodes has been expanded since
; I am using this as a rewrite rule.
```

THEOREM: proper-p-instructionp2-implies-member-piton-opcodes
proper-p-instructionp2 (*ins*, *name*, *p*)
$\rightarrow$ ((car (*ins*)
$\in$ '(call ret locn push-constant push-local
push-global push-ctrl-stk-free-size
push-temp-stk-free-size push-temp-stk-index
jump-if-temp-stk-full jump-if-temp-stk-empty
pop pop* popn pop-local pop-global pop-locn
pop-call fetch-temp-stk deposit-temp-stk jump
jump-case pushj popj set-local set-global
test-nat-and-jump test-int-and-jump
test-bool-and-jump test-bitv-and-jump no-op
add-addr sub-addr eq lt-addr fetch deposit

```
add-int add-int-with-carry add1-int sub-int
sub-int-with-carry sub1-int neg-int lt-int
int-to-nat add-nat add-nat-with-carry add1-nat
sub-nat sub-nat-with-carry sub1-nat lt-nat
mult2-nat mult2-nat-with-carry-out div2-nat
or-bitv and-bitv not-bitv xor-bitv rsh-bitv
lsh-bitv or-bool and-bool not-bool))
```

$\land$ ((car (*ins*) = `'push-constant`)
$\rightarrow$ proper-p-push-constant-instructionp (*ins*, *name*, *p*)))

EVENT: Disable proper-p-instructionp2.

THEOREM: proper-p-instructionp-implies-member-piton-opcodes
proper-p-instructionp (*ins*, *name*, *p*)
$\rightarrow$ ((car (*ins*) $\in$ PITON-OPCODES)
$\land$ ((car (*ins*) = `'push-constant`)
$\rightarrow$ proper-p-push-constant-instructionp (*ins*, *name*, *p*)))

```
; The push-constant predicate checks that the constant is (possibly)
; a p-object.  The r level makes sure it is not a fake dl.  We need
; to know p-objects aren't dls:
```

THEOREM: p-objectp-implies-not-dl
p-objectp (x, p) $\rightarrow$ (car (x) $\neq$ `'dl`)

```
; Any program defined within a proper-p-prog-segmentp has a properly
; labelled body:
```

THEOREM: proper-p-prog-segmentp-contains-proper-labeled-p-instructionsp
(definedp (*name*, *segment*) $\land$ proper-p-prog-segmentp (*segment*, *p*))
$\rightarrow$ proper-labeled-p-instructionsp (program-body (assoc (*name*, *segment*)),
name,
p)

```
; So, if you have a proper list of instructions at the p level then
; it is proper at the r level, provided the name is defined within
; a proper-p-prog-segmentp.
```

THEOREM: proper-p-instructionp-implies-r-instructionp
(proper-p-instructionp (*ins*, *name*, *p*)
$\land$ definedp (*name*, p-prog-segment (*p*))
$\land$ proper-p-prog-segmentp (p-prog-segment (*p*), *p*))
$\rightarrow$ r-instructionp (*ins*)

```
; Consequently, a properly labelled sequence of instructions at the p
; level is a proper sequence at the r level -- this is just an induction
; applying the lemma above.
```

THEOREM: proper-labeled-p-instructionsp-implies-r-program-bodyp
(proper-labeled-p-instructionsp (lst, $name$, p)
$\land$ definedp ($name$, p-prog-segment (p))
$\land$ proper-p-prog-segmentp (p-prog-segment (p), p))
$\rightarrow$ r-program-bodyp (lst)

```
; So a proper p prog-segment is a proper r prog-segment.  But it is
; pretty tricky to state a general enough lemma.  The trouble is that we
; need to induct on the segment while holding the state and its prog-segment
; fixed.  We need the auxiliary concept that every name in one segment
; is defined in the state's:
```

DEFINITION:
all-names-definedp ($segment1$, $segment2$)
= **if** $segment1 \simeq$ **nil then t**
else definedp (name (car ($segment1$)), $segment2$)
$\land$ all-names-definedp (cdr ($segment1$), $segment2$) **endif**

```
; Now we can state the essence of the theorem:
```

THEOREM: proper-p-prog-segmentp-implies-r-prog-segmentp-generalized
(proper-p-prog-segmentp ($segment$, p)
$\land$ all-names-definedp ($segment$, p-prog-segment (p))
$\land$ proper-p-prog-segmentp (p-prog-segment (p), p))
$\rightarrow$ r-prog-segmentp ($segment$)

```
; But of course,
```

THEOREM: all-names-definedp-cdr
all-names-definedp (x, cdr (y)) $\rightarrow$ all-names-definedp (x, y)

THEOREM: all-names-definedp-reflexive
all-names-definedp ($segment$, $segment$)

```
; And so we can instantiate the lemma above,
; proper-p-prog-segmentp-implies-r-prog-segmentp-generalized, with
; segment replaced by (p-prog-segment p) and collapse the hyps to get:
```

THEOREM: proper-p-prog-segmentp-implies-r-prog-segmentp
proper-p-prog-segmentp (p-prog-segment (p), p)
$\rightarrow$ r-prog-segmentp (p-prog-segment (p))

```
; Whew!  The delicacy of that proof really surprised me.

; So, now we move on the next part of the proper-r-statep predicate,
; the consideration of the usr-data-segment.

; The core of course is just showing that p-objects are
; r-usr-data-objects.  Then we lift it up through several layers
; of recursion.
```

THEOREM: p-objectp-implies-r-usr-data-typep
p-objectp $(x, p) \rightarrow$ r-usr-data-typep (type (x))

THEOREM: p-objectp-implies-r-usr-data-objectp
(p-objectp $(x, p) \wedge$ same-signature (*r-usr-data-segment*, p-data-segment (p)))
$\rightarrow$ r-usr-data-objectp (x,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,
r-y,
r-c-flg,
r-v-flg,
r-n-flg,
r-z-flg,
p-prog-segment (p),
r-usr-data-segment,
r-sys-data-segment,
p-word-size (p),
r-psw))

THEOREM: all-p-objectps-implies-all-r-usr-data-objectps
(all-p-objectps (*lst*, p)
$\wedge$ same-signature (*r-usr-data-segment*, p-data-segment (p)))
$\rightarrow$ all-r-usr-data-objectps (*lst*,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
r-x,

$r\text{-}y$,
$r\text{-}c\text{-}flg$,
$r\text{-}v\text{-}flg$,
$r\text{-}n\text{-}flg$,
$r\text{-}z\text{-}flg$,
p-prog-segment (p),
r-usr-data-segment,
r-sys-data-segment,
p-word-size (p),
r-psw))

THEOREM: proper-p-data-segmentp-implies-proper-r-usr-data-segmentp
(proper-p-data-segmentp (*segment*, p)
$\land$ same-signature (*r-usr-data-segment*, p-data-segment (p)))
$\rightarrow$ proper-r-usr-data-segmentp (*segment*,
r-state (*r-pc*,
r-cfp,
r-csp,
r-tsp,
$r\text{-}x$,
$r\text{-}y$,
$r\text{-}c\text{-}flg$,
$r\text{-}v\text{-}flg$,
$r\text{-}n\text{-}flg$,
$r\text{-}z\text{-}flg$,
p-prog-segment (p),
r-usr-data-segment,
r-sys-data-segment,
p-word-size (p),
r-psw))

```
; That completes the data segment work.  Finally, we consider the question
; of loadability.

; The need for the second and third hypotheses surprised me. The size
; of the mapped down stacks is as expected only if the p level stacks
; are sufficiently small.
```

THEOREM: p-loadablep-implies-r-loadablep
(p-loadablep (p, *load-addr*)
$\land$ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p)))
$\land$ (p-max-temp-stk-size (p) $\not<$ length (p-temp-stk (p))))
$\rightarrow$ ((*load-addr*
$+$ segment-length (p-data-segment (p))

$+$ segment-length (icompile (p-prog-segment (p)))
$+$ segment-length (p->r_sys-data-segment (p-ctrl-stk (p),
p-max-ctrl-stk-size (p),
p-temp-stk (p),
p-max-temp-stk-size (p))))
$<$ exp (2, p-word-size (p)))

```
; So, here we are at what we wanted:  proper states map
; down from p to r.
```

THEOREM: proper-p-statep-implies-proper-r-statep
(proper-p-statep (p) $\wedge$ p-loadablep (p, *load-addr*))
$\rightarrow$ proper-r-statep (p->r (p), *load-addr*)

```
; So we can now complete the p-r level proof by just observing the
; usual things about the treatment of errors.
```

THEOREM: once-errorp-always-errorp-step{special}
($\neg$ errorp (p-psw (p-step (p)))) $\rightarrow$ ($\neg$ errorp (p-psw (p)))

THEOREM: once-errorp-always-errorp{special}
($\neg$ errorp (p-psw (p (p, n)))) $\rightarrow$ ($\neg$ errorp (p-psw (p)))

```
; The following lemma wasn't necessary until I added load-addr.  Its
; only purpose is to solve a free-var problem in r-preserves-r-equal.
; Essentially it is just that lemma except we've gone ahead and
; backchained up to (p-loadablep p load-addr) so that the
; instantiation of load-addr can be found in
; one-way-correspondence-p-r immediately below.
```

THEOREM: r-preserves-r-equal-bridge
(proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ proper-r-statep ($r2$, *load-addr*)
$\wedge$ r-equal ($r2$, r-step (p->r (p))))
$\rightarrow$ r-equal (r ($r2$, n), r (r-step (p->r (p)), n))

THEOREM: one-way-correspondence-p-r
(proper-p-statep (p)
$\wedge$ p-loadablep (p, *load-addr*)
$\wedge$ ($\neg$ errorp (p-psw (p (p, n)))))
$\rightarrow$ r-equal (p->r (p (p, n)), r (p->r (p), n))

THEOREM: p-preserves-proper-p-statep
(proper-p-statep $(p) \land (\neg$ errorp (p-psw (p $(p,\ n)))))$
$\rightarrow$ proper-p-statep (p $(p,\ n))$

```
; Book: prim.events (sort of).  This section contains an informal
; sketch of how all of the foregoing are put together into the final
; result.

; In the FM8502 script this informal sketch was followed by its
; formalization: we were virtually done at this point.  This section
; began as prim.events which just imported the names enabled below,
; proved a few little helpers, and then combined the results to get
; the map from P down to FM8502.  But when we tried to do that for
; FM9001 we learned that to get the map up from FM9001 to M we needed
; a constraint on the link tables having to do with the size of the
; physical memory.  (In the first effort to move Piton to FM9001 we
; proved the map down and just left the map up for this section,
; thinking it would be simple.  When we discovered the problem, we
; defined the constraint (PLAUSIBLE-DATA-LINK-TABLEP) in the section
; on m to FM9001 and we proved that it was sufficient to get the map
; up, but we did not show that the plausibility condition was
; satisfied.  We left that for here because its satisfaction is tied
; up with the notion of same-signature, which was not available at the
; time we defined plausibility.)

; So this section of the script begins the final assault on the proof
; but then has a long "digression" were we show how to get the
; plausibility condition.
```

EVENT: Set the status of all events. The status of each event is to be set as follows. Each 'boot-strap' satellite is to be returned to the initial condition. Each 'add-shell' satellite is to be enabled. Each '*1* function' is to be enabled. disabled. Anything not otherwise mentioned is to be disabled. Name this event 'close-data-base-5'.

EVENT: Enable theory (proper-p-statep-implies-proper-r-statep proper-r-statep-implies-proper-i-user-data-segmentp proper-r-statep-r proper-r-statep-r-step proper-r-statep-implies-i-state-okp-r-¿i display-fm9001-data-segment-m-¿fm9001 display-m-data-segment-inverts-i-¿m i-link-tables-i i-state-okp-i-ins-step)proper-p-statep-implies-proper-r-statep, proper-r-statep-implies-proper-i-user-data-segmentp, proper-r-statep-r, proper-r-statep-r-step, proper-r-statep-implies-i-state-okp-r->i, display-fm9001-data-segment-m->fm9001, display-m-data-segment-inverts-i->m, i-link-tables-i, i-state-okp-i-ins-step.

THEOREM: fm9001-m-owc
proper-m-statep$(m) \rightarrow$ (fm9001 (m->fm9001(m), n) = m->fm9001 (m (m, n)))

THEOREM: i-m-owc
$((\textit{load-addr} \in \mathbf{N})$
$\wedge$ (i-psw (i $(i, n, \textit{load-addr})$) = `'run`)
$\wedge$ (i-word-size(i) = 32))
$\rightarrow$ (m (i->m $(i, \textit{boot-lst}, \textit{load-addr})$, n)
= i->m (i $(i, n, \textit{load-addr})$, $\textit{boot-lst}$, $\textit{load-addr}$))

```
; (This one was oriented the other way.)
```

THEOREM: r-i-owc
(proper-r-statep $(r, \textit{load-addr})$
$\wedge$ $(\textit{load-addr} \in \mathbf{N})$
$\wedge$ ($\neg$ errorp (r-psw (r (r, n)))))
$\rightarrow$ (i (r->i(r), clock(r, n), $\textit{load-addr}$) = r->i (r (r, n))))

```
; (This one was oriented the other way.)
```

THEOREM: p-r-owc-corrolary
(proper-p-statep(p)
$\wedge$ p-loadablep$(p, \textit{load-addr})$
$\wedge$ ($\neg$ errorp (p-psw (p (p, n)))))
$\rightarrow$ ((r-psw (r (p->r(p), n)) = p-psw (p (p, n)))
$\wedge$ (r-usr-data-segment (r (p->r(p), n)) = p-data-segment (p (p, n)))))

```
; (This one was packaged up with R-EQUAL and now we are just interested
; in the equality of the psw and data segment components.)

; The three owc theorems have hypotheses about word-size, proper states,
; and psws.  The proper states hyps are all obtainable from the
; proper-p-statep in the top goal and the previously noted theorem
; that proper-p-statep implies proper-r-statep of p->r and the corresponding
; relation down to the i-level by r->i.  However, the lemma that we
; can get from proper r states to proper i states is not a REWRITE
; rule and so we package it up now:
```

THEOREM: proper-r-statep-implies-i-state-okp-r->i-rewrite
(proper-r-statep$(r, \textit{load-addr})$ $\wedge$ ($\neg$ errorp (r-psw(r))))
$\rightarrow$ i-state-okp (r->i(r), $\textit{load-addr}$)

```
; The main theorem contains the hypothesis that the word size at the p-level
; is 32.  But our i-m-owc requires that the word size at the i level be 32.
; Clearly we need to know that the word size is the same all the way down:
```

THEOREM: i-word-size-r->i
i-word-size (r->i (r)) = r-word-size (r)

THEOREM: r-word-size-p->r
r-word-size (p->r (p)) = p-word-size (p)

```
; Similarly, the main theorem gives us that the final p-psw is not an errorp.
; But the i-m-owc requires that the final i-psw is 'run.  The corrollary to
; the p-r-owc gives us that the r-psw is not an error.  So it suffices to
; observe
```

THEOREM: i-psw-r->i
($\neg$ errorp (r-psw (r))) $\rightarrow$ (i-psw (r->i (r)) = 'run)

```
; That completes the preliminaries.  We will use the three owcs without
; further attention to their hyps.  We now give the proof of the top goal.

; Note:  The following comments are from the FM8502 proof.  But all of this
; still holds except that we now have one more mapping function, M->FM9001,
; after i->m.

; Consider the display-m-data-segment expression in the top goal:

; (display-m-data-segment (fm8502 (i->m (r->i (p->r p0)) boot-lst load-addr) ...) ...)

; By the equivalence of m and fm8502, this becomes

; (display-m-data-segment (m (i->m (r->i (p->r p0)) boot-lst load-addr) ...) ...).

; Then, by i-m-owc, the i->m is brought outside the m to produce:

; (display-m-data-segment (i->m (i (r->i (p->r p0)) n load-addr) boot-lst load-addr) ...).

; Then, by r-i-owc, the r->i is brought outside the i:

; (display-m-data-segment (i->m (r->i (r (p->r p0) n)) boot-lst load-addr)
;                         (type-specification (p-data-segment (p p0 n)))
;                         (link-tables p0 load-addr))
```

```
; In addition, link-tables expands and we get:

; (display-m-data-segment (i->m (r->i (r (p->r p0) n)) boot-lst load-addr)
;                         (type-specification (p-data-segment (p p0 n)))
;                         (i-link-tables (r->i (p->r p0)) load-addr))

; We wish that would match the left-hand side of the conclusion of
; display-m-data-segment-inverts-i->m, said conclusion being:

; (equal (display-m-data-segment (i->m i boot-lst load-addr)
;                                (type-specification (i-usr-data-segment i))
;                                (i-link-tables i load-addr))
;        (i-usr-data-segment i)))

; But it does not because of the second and third args.

; We therefore build a bridged version of the lemma noted above.  It
; is:
```

THEOREM: display-m-data-segment-inverts-i->m-bridge

$$
\begin{array}{ll}
\multicolumn{2}{l}{((\mathit{load\text{-}addr} \in \mathbf{N})} \\
\land & \text{proper-i-usr-data-segmentp}\,(\text{i-usr-data-segment}\,(i),\ i) \\
\land & \text{i-state-okp}\,(i,\ \mathit{load\text{-}addr}) \\
\land & (\mathit{type\text{-}spec} = \text{type-specification}\,(\text{i-usr-data-segment}\,(i))) \\
\land & (\mathit{link\text{-}tables} = \text{i-link-tables}\,(i,\ \mathit{load\text{-}addr}))) \\
\rightarrow & (\text{display-m-data-segment}\,(\text{i->m}\,(i,\ \mathit{boot\text{-}lst},\ \mathit{load\text{-}addr}), \\
 & \qquad \mathit{type\text{-}spec}, \\
 & \qquad \mathit{link\text{-}tables}) \\
 & \quad = \ \text{i-usr-data-segment}\,(i))
\end{array}
$$

```
; Suppose we could relieve the four hyps.   Then the top goal would reduce
; to:

;  (equal (i-usr-data-segment (r->i (r (p->r p0) n)))
;         (p-data-segment (p p0 n))).

; But we know
```

THEOREM: i-usr-data-segment-r->i

i-usr-data-segment (r->i (r)) = r-usr-data-segment (r)

```
; So the top goal becomes:
```

```
;  (equal (r-usr-data-segment (r (p->r p0) n))
;         (p-data-segment (p p0 n))).

; which t by the corrollary of p-r-owc.

; So it only remains to relieve the hypotheses of the bridged display lemma
; above.  There are four:

; Hyp1. (proper-i-usr-data-segmentp (i-usr-data-segment i) i)
; Hyp2. (i-state-okp i load-addr)
; Hyp3. (equal type-spec (type-specification (i-usr-data-segment i)))
; Hyp4. (equal link-tables (i-link-tables i load-addr)))

; where i           is   (r->i (r (p->r p0) n))
;       type-spec   is   (type-specification (p-data-segment (p p0 n)))
;       link-tables is   (i-link-tables (r->i (p->r p0)) load-addr)

; We deal with them in turn.

; Hyp1.  The fully instantiated first hypothesis is:

;   (proper-i-usr-data-segmentp (i-usr-data-segment (r->i (r (p->r p0) n)))
;                               (r->i (r (p->r p0) n)))

; If we apply the previous rule for simplifying the i-usr-data-segment of
; r->i, we would get:

;   (proper-i-usr-data-segmentp (r-usr-data-segment (r (p->r p0) n))
;                               (r->i (r (p->r p0) n)))

; This would immediately match the imported theorem
; proper-r-statep-implies-proper-i-user-data-segmentp
; and we would only have to establish that (r (p->r p0) n) was a
; proper-r-statep.  This follows from the fact that p->r is a
; proper-r-statep (since p0 is a proper-p-statep) and r preserves
; proper-r-statep.  [Note:  we could have relied on p preserving
; proper-p-statep, but then we'd need to express the r term above
; as (p->r (p p0 n)).]

; But this simple argument doesn't work because of the presence of
; the p-r-owc corrollary.  In particular, (r-usr-data-segment (r (p->r p0) n))
; rewrites to (p-data-segment (p p0 n)) by the corrollary and we no
; longer get the match on proper-r-statep-implies-proper-i-user-data-segmentp.
```

```
; So we prove a bridged version of that rule:
```

THEOREM: proper-r-statep-implies-proper-i-user-data-segmentp-bridge
(proper-r-statep (r, *load-addr*) $\land$ (*segment* = r-usr-data-segment (r)))
$\rightarrow$ proper-i-usr-data-segmentp (*segment*, r->i (r))

```
; That completes hyp1.

; Hyp2.  (i-state-okp (r->i (r (p->r p0) n)) load-addr) follows from the fact that
; r->i produces okp i states from proper r states.

; Hyp3.  The third hypothesis to prove is that the equivalence of

;    (type-specification (p-data-segment (p p0 n)))

; and

;    (type-specification (i-usr-data-segment (r->i (r (p->r p0 n))))).

; The first is what occurs in the top goal; the second is what our lemma
; expects.  But the previously noted fact about the i-usr-data-segment
; of r->i reduces the second to:

;    (type-specification (r-usr-data-segment (r (p->r p0 n))))

; And the corrollary of p-r-owc reduces that to the first.

; Hyp4.  The last hypothesis is the equivalence of

;    (i-link-tables (r->i (p->r p0)) load-addr)

; and

;    (i-link-tables (r->i (r (p->r p0) n)) load-addr).

; Observe that what we are asked to show is that execution on the r machine
; preserves the i-link-tables (computed from mapped down r states).  But
; we know that the i machine preserves link tables.  And we know, from
; r-i-owc, that any mapped down r computation can be seen as an i computation
; on the mapped down states.  So the above requirement follows.  Here is
; a formal rendering of it:
```

THEOREM: i-link-tables-r->i

$((\mathit{load\text{-}addr} \in \mathbf{N})$
$\land \quad (\text{r-word-size}(\mathit{r0}) = 32)$
$\land \quad \text{proper-r-statep}(\mathit{r0},\ \mathit{load\text{-}addr})$
$\land \quad (\neg\ \text{errorp}(\text{r-psw}(\text{r}(\mathit{r0},\ n)))))$
$\rightarrow \quad (\text{i-link-tables}(\text{r->i}(\text{r}(\mathit{r0},\ n)),\ \mathit{load\text{-}addr})$
$\quad = \quad \text{i-link-tables}(\text{r->i}(\mathit{r0}),\ \mathit{load\text{-}addr}))$

```
; That completes the proof of FM8501 Piton.  We have to define the
; witness function, FM8502-CLOCK, but that is easy:
```

DEFINITION: $\text{fm9001-clock}(p,\ n) = \text{clock}(\text{p->r}(p),\ n)$

```
; Note:  In the FM8502 script we were in fact done!  Here we simply
; put top-goal.  But no longer.  We must relieve the plausibility
; constraint.

; Book: plausibility.  We now depart from the FM8502 proof and begin
; to work on deriving the plausibility condition from the fact that
; the initial p state is proper.  We ultimately have to establish that
; the intermediate machines preserve the word size of the state as
; well as the signatures of the program and data segments.
```

THEOREM: bvp-listp-append
$\text{bvp-listp}(a) \rightarrow (\text{bvp-listp}(\text{append}(a,\ b)) = \text{bvp-listp}(b))$

THEOREM: bvp-listp-link-area
$\text{bvp-listp}(\text{link-area}(\mathit{lst},\ \mathit{link\text{-}tables},\ 32))$

THEOREM: bvp-listp-link-segment
$\text{bvp-listp}(\text{link-segment}(\mathit{lst},\ \mathit{link\text{-}tables},\ 32))$

THEOREM: bvp-listp-boot-code
$\text{bvp-listp}(\text{boot-code}(\mathit{boot\text{-}lst},\ n,\ 32))$

THEOREM: proper-m-statep-i->m
$((\text{i-word-size}(i) = 32) \land \text{i-loadablep}(i,\ \mathit{load\text{-}addr}))$
$\rightarrow \quad \text{proper-m-statep}(\text{i->m}(i,\ \mathit{boot\text{-}lst},\ \mathit{load\text{-}addr}))$

THEOREM: i-loadable-r->i
$\text{proper-r-statep}(r,\ \mathit{load\text{-}addr}) \rightarrow \text{i-loadablep}(\text{r->i}(r),\ \mathit{load\text{-}addr})$

```
; I need to show that r preserves the word-size.  It surprises me that I haven't
; proved this at every level.  I apparently haven't proved this at any level!
```

THEOREM: r-word-size-r-step1
r-word-size (r-step1 (ins, r)) = r-word-size (r)

THEOREM: r-word-size-r-step
r-word-size (r-step (r)) = r-word-size (r)

THEOREM: r-word-size-r
r-word-size (r (r, n)) = r-word-size (r)

```
; I have confirmed that if we add the following PLAUSIBLE-TYPE-
; SPEC-AND-LINK-TABLESP instance as an axiom, we can prove top-goal.
; So the entire problem reduces to that described below.  Everything
; between here and the proof of consolidated-plausibility-argument was
; created over a 3 day period (Oct 9-11, 1991) in order to establish
; the plausibility hypothesis necessary to apply DISPLAY-FM9001-
; DATA-SEGMENT-M->FM9001.  That hypothesis is:

; (PLAUSIBLE-DATA-LINK-TABLEP
;         (TYPE-SPECIFICATION (P-DATA-SEGMENT (P P0 N)))
;         (LENGTH (M-MEM (I->M (R->I (R (P->R P0) N)) boot-lst LOAD-ADDR)))
;         (USR-DATA-LINKS (I-LINK-TABLES (R->I (P->R P0)) LOAD-ADDR)))
; under the assumptions
; (NOT (ERRORP (P-PSW (P P0 N))))
; (EQUAL (P-WORD-SIZE P0) 32)
; (P-LOADABLEP P0 LOAD-ADDR)
; (PROPER-P-STATEP P0)

; Plausibility insures that, for each area in the type-spec, the base address of
; the area plus the length of the area must be less than or equal to the length of
; the memory.

; We start by noting
```

THEOREM: same-signature-plausible-data-link-tablep
(same-signature ($seg1$, $seg2$)
$\wedge$ plausible-data-link-tablep (type-specification ($seg1$), n, $table$))
$\rightarrow$ plausible-data-link-tablep (type-specification ($seg2$), n, $table$)

THEOREM: same-signature-reflexive-1
same-signature ($segment$, $segment$)

THEOREM: proper-p-data-segmentp-proper-alistp
proper-p-data-segmentp (x, p) $\rightarrow$ proper-alistp (x)

THEOREM: proper-alistp-p-data-segment
proper-p-statep (p) $\rightarrow$ proper-alistp (p-data-segment (p))

```
; Our next goal is to prove that the p machine does not change the
; signature of the data segment.  Most instructions don't change
; the data segment at all and the corresponding lemma can be proved
; without using most of the hypotheses provided.  But we make all of the
; lemmas of the same form simply so we can use an Emacs macro to
; create them all.  Had these proofs been done at the appropriate
; stage in the proof of the p-r, we would not need to give the hints.
```

THEOREM: same-signature-p-ins-step-call
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'call`)
$\land$ proper-p-statep (p)
$\land$ p-call-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-call-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-ret
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'ret`)
$\land$ proper-p-statep (p)
$\land$ p-ret-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-ret-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-locn
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'locn`)
$\land$ proper-p-statep (p)
$\land$ p-locn-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-locn-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-push-constant
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'push-constant`)
$\land$ proper-p-statep (p)
$\land$ p-push-constant-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-push-constant-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-push-local
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'push-local)
∧ proper-p-statep (p)
∧ p-push-local-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-push-local-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-push-global
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'push-global)
∧ proper-p-statep (p)
∧ p-push-global-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-push-global-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-push-ctrl-stk-free-size
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'push-ctrl-stk-free-size)
∧ proper-p-statep (p)
∧ p-push-ctrl-stk-free-size-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-push-ctrl-stk-free-size-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-push-temp-stk-free-size
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'push-temp-stk-free-size)
∧ proper-p-statep (p)
∧ p-push-temp-stk-free-size-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-push-temp-stk-free-size-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-push-temp-stk-index
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'push-temp-stk-index)
∧ proper-p-statep (p)
∧ p-push-temp-stk-index-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-push-temp-stk-index-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-jump-if-temp-stk-full
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'jump-if-temp-stk-full**)
∧ proper-p-statep (p)
∧ p-jump-if-temp-stk-full-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-jump-if-temp-stk-full-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-jump-if-temp-stk-empty
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'jump-if-temp-stk-empty**)
∧ proper-p-statep (p)
∧ p-jump-if-temp-stk-empty-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-jump-if-temp-stk-empty-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-pop
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'pop**)
∧ proper-p-statep (p)
∧ p-pop-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-pop-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-pop*
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'pop***)
∧ proper-p-statep (p)
∧ p-pop*-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-pop*-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-popn
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'popn**)
∧ proper-p-statep (p)
∧ p-popn-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-popn-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-pop-local
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'pop-local**)

$\land$ proper-p-statep (p)
$\land$ p-pop-local-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-pop-local-step (p-current-instruction (p),
p)))

```
; So here is the first occasion on which we need to investigate the hyps to
; get the proof.  An important fact here is that since the state is proper
; the programs are all proper so that the pop-global instruction we are
; executing references a legal area name in the data section.  If it did not
; the resulting deposit would change the signature!  Similarly, we must know that
; the length of the area is sufficient.  We have to enable the rules that were
; freely available during the p-r correspondence work.
```

THEOREM: same-signature-p-ins-step-pop-global
((p-psw (p) = '**run**)
$\land$ (car (p-current-instruction (p)) = '**pop-global**)
$\land$ proper-p-statep (p)
$\land$ p-pop-global-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-pop-global-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-pop-locn
((p-psw (p) = '**run**)
$\land$ (car (p-current-instruction (p)) = '**pop-locn**)
$\land$ proper-p-statep (p)
$\land$ p-pop-locn-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-pop-locn-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-pop-call
((p-psw (p) = '**run**)
$\land$ (car (p-current-instruction (p)) = '**pop-call**)
$\land$ proper-p-statep (p)
$\land$ p-pop-call-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-pop-call-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-fetch-temp-stk
((p-psw (p) = '**run**)
$\land$ (car (p-current-instruction (p)) = '**fetch-temp-stk**)

∧ proper-p-statep (p)
∧ p-fetch-temp-stk-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-fetch-temp-stk-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-deposit-temp-stk
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'deposit-temp-stk`)
∧ proper-p-statep (p)
∧ p-deposit-temp-stk-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-deposit-temp-stk-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-jump
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'jump`)
∧ proper-p-statep (p)
∧ p-jump-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-jump-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-jump-case
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'jump-case`)
∧ proper-p-statep (p)
∧ p-jump-case-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-jump-case-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-pushj
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'pushj`)
∧ proper-p-statep (p)
∧ p-pushj-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-pushj-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-popj
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'popj`)
∧ proper-p-statep (p)
∧ p-popj-okp (p-current-instruction (p), p))

$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-popj-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-set-local
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'set-local`)
$\land$ proper-p-statep (p)
$\land$ p-set-local-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-set-local-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-set-global
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'set-global`)
$\land$ proper-p-statep (p)
$\land$ p-set-global-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-set-global-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-test-nat-and-jump
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'test-nat-and-jump`)
$\land$ proper-p-statep (p)
$\land$ p-test-nat-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-test-nat-and-jump-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-test-int-and-jump
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'test-int-and-jump`)
$\land$ proper-p-statep (p)
$\land$ p-test-int-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-test-int-and-jump-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-test-bool-and-jump
((p-psw (p) = `'run`)
$\land$ (car (p-current-instruction (p)) = `'test-bool-and-jump`)
$\land$ proper-p-statep (p)
$\land$ p-test-bool-and-jump-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),

p-data-segment (p-test-bool-and-jump-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-test-bitv-and-jump
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'test-bitv-and-jump`)
∧ proper-p-statep (p)
∧ p-test-bitv-and-jump-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-test-bitv-and-jump-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-no-op
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'no-op`)
∧ proper-p-statep (p)
∧ p-no-op-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-no-op-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-add-addr
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'add-addr`)
∧ proper-p-statep (p)
∧ p-add-addr-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-add-addr-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-sub-addr
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'sub-addr`)
∧ proper-p-statep (p)
∧ p-sub-addr-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-sub-addr-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-eq
((p-psw (p) = `'run`)
∧ (car (p-current-instruction (p)) = `'eq`)
∧ proper-p-statep (p)
∧ p-eq-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-eq-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-lt-addr
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'lt-addr`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-lt-addr-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-lt-addr-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-fetch
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'fetch`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-fetch-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-fetch-step (p-current-instruction (p), p)))

THEOREM: same-signature-p-ins-step-deposit
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'deposit`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-deposit-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-deposit-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-add-int
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'add-int`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-add-int-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-add-int-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-add-int-with-carry
((p-psw (p) = `'run`)
$\wedge$ (car (p-current-instruction (p)) = `'add-int-with-carry`)
$\wedge$ proper-p-statep (p)
$\wedge$ p-add-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
p-data-segment (p-add-int-with-carry-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-add1-int

((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'add1-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-add1-int-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
 p-data-segment (p-add1-int-step (p-current-instruction (p),
 p)))

THEOREM: same-signature-p-ins-step-sub-int
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-sub-int-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
 p-data-segment (p-sub-int-step (p-current-instruction (p),
 p)))

THEOREM: same-signature-p-ins-step-sub-int-with-carry
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub-int-with-carry`)
 $\land$ proper-p-statep (p)
 $\land$ p-sub-int-with-carry-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
 p-data-segment (p-sub-int-with-carry-step (p-current-instruction (p),
 p)))

THEOREM: same-signature-p-ins-step-sub1-int
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'sub1-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-sub1-int-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
 p-data-segment (p-sub1-int-step (p-current-instruction (p),
 p)))

THEOREM: same-signature-p-ins-step-neg-int
((p-psw (p) = `'run`)
 $\land$ (car (p-current-instruction (p)) = `'neg-int`)
 $\land$ proper-p-statep (p)
 $\land$ p-neg-int-okp (p-current-instruction (p), p))
$\rightarrow$ same-signature (p-data-segment (p),
 p-data-segment (p-neg-int-step (p-current-instruction (p),
 p)))

THEOREM: same-signature-p-ins-step-lt-int

((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'lt-int)
∧ proper-p-statep (p)
∧ p-lt-int-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-lt-int-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-int-to-nat
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'int-to-nat)
∧ proper-p-statep (p)
∧ p-int-to-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-int-to-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-add-nat
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'add-nat)
∧ proper-p-statep (p)
∧ p-add-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-add-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-add-nat-with-carry
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'add-nat-with-carry)
∧ proper-p-statep (p)
∧ p-add-nat-with-carry-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-add-nat-with-carry-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-add1-nat
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'add1-nat)
∧ proper-p-statep (p)
∧ p-add1-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-add1-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-sub-nat

((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'sub-nat**)
∧ proper-p-statep (p)
∧ p-sub-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-sub-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-sub-nat-with-carry
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'sub-nat-with-carry**)
∧ proper-p-statep (p)
∧ p-sub-nat-with-carry-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-sub-nat-with-carry-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-sub1-nat
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'sub1-nat**)
∧ proper-p-statep (p)
∧ p-sub1-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-sub1-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-lt-nat
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'lt-nat**)
∧ proper-p-statep (p)
∧ p-lt-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-lt-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-mult2-nat
((p-psw (p) = **'run**)
∧ (car (p-current-instruction (p)) = **'mult2-nat**)
∧ proper-p-statep (p)
∧ p-mult2-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-mult2-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-mult2-nat-with-carry-out

((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'mult2-nat-with-carry-out)
∧ proper-p-statep (p)
∧ p-mult2-nat-with-carry-out-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-mult2-nat-with-carry-out-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-div2-nat
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'div2-nat)
∧ proper-p-statep (p)
∧ p-div2-nat-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-div2-nat-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-or-bitv
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'or-bitv)
∧ proper-p-statep (p)
∧ p-or-bitv-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-or-bitv-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-and-bitv
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'and-bitv)
∧ proper-p-statep (p)
∧ p-and-bitv-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-and-bitv-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-not-bitv
((p-psw (p) = 'run)
∧ (car (p-current-instruction (p)) = 'not-bitv)
∧ proper-p-statep (p)
∧ p-not-bitv-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
p-data-segment (p-not-bitv-step (p-current-instruction (p),
p)))

THEOREM: same-signature-p-ins-step-xor-bitv

((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**xor-bitv**)
 ∧ proper-p-statep (p)
 ∧ p-xor-bitv-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
  p-data-segment (p-xor-bitv-step (p-current-instruction (p),
  p)))

THEOREM: same-signature-p-ins-step-rsh-bitv
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**rsh-bitv**)
 ∧ proper-p-statep (p)
 ∧ p-rsh-bitv-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
  p-data-segment (p-rsh-bitv-step (p-current-instruction (p),
  p)))

THEOREM: same-signature-p-ins-step-lsh-bitv
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**lsh-bitv**)
 ∧ proper-p-statep (p)
 ∧ p-lsh-bitv-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
  p-data-segment (p-lsh-bitv-step (p-current-instruction (p),
  p)))

THEOREM: same-signature-p-ins-step-or-bool
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**or-bool**)
 ∧ proper-p-statep (p)
 ∧ p-or-bool-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
  p-data-segment (p-or-bool-step (p-current-instruction (p),
  p)))

THEOREM: same-signature-p-ins-step-and-bool
((p-psw (p) = '**run**)
 ∧ (car (p-current-instruction (p)) = '**and-bool**)
 ∧ proper-p-statep (p)
 ∧ p-and-bool-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (p),
  p-data-segment (p-and-bool-step (p-current-instruction (p),
  p)))

THEOREM: same-signature-p-ins-step-not-bool

$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad (\text{car}\,(\text{p-current-instruction}\,(p)) = \texttt{'not-bool})$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad \text{p-not-bool-okp}\,(\text{p-current-instruction}\,(p),\ p))$
$\to\quad \text{same-signature}\,(\text{p-data-segment}\,(p),$
$\qquad\qquad \text{p-data-segment}\,(\text{p-not-bool-step}\,(\text{p-current-instruction}\,(p),$
$\qquad\qquad\qquad p)))$

```
; We now put these all together.
```

THEOREM: same-signature-p-ins-step
$((\text{p-psw}\,(p) = \texttt{'run})$
$\land\quad \text{p-ins-okp}\,(\text{p-current-instruction}\,(p),\ p)$
$\land\quad \text{proper-p-statep}\,(p))$
$\to\quad \text{same-signature}\,(\text{p-data-segment}\,(p),$
$\qquad\qquad \text{p-data-segment}\,(\text{p-ins-step}\,(\text{p-current-instruction}\,(p),\ p)))$

THEOREM: same-signature-p-step
$\text{proper-p-statep}\,(p)$
$\to\quad \text{same-signature}\,(\text{p-data-segment}\,(p),\ \text{p-data-segment}\,(\text{p-step}\,(p)))$

THEOREM: same-signature-p-transitivity-lemma
$((n \in \mathbf{N})$
$\land\quad (n \neq 0)$
$\land\quad \text{same-signature}\,(\text{p-data-segment}\,(\text{p-step}\,(p)),$
$\qquad\qquad \text{p-data-segment}\,(\text{p}\,(\text{p-step}\,(p),\ n - 1)))$
$\land\quad \text{proper-p-statep}\,(p)$
$\land\quad (\neg\ \text{errorp}\,(\text{p-psw}\,(\text{p}\,(\text{p-step}\,(p),\ n - 1)))))$
$\to\quad \text{same-signature}\,(\text{p-data-segment}\,(p),\ \text{p-data-segment}\,(\text{p}\,(\text{p-step}\,(p),\ n - 1)))$

THEOREM: same-signature-p
$(\text{proper-p-statep}\,(p) \land (\neg\ \text{errorp}\,(\text{p-psw}\,(\text{p}\,(p,\ n)))))$
$\to\quad \text{same-signature}\,(\text{p-data-segment}\,(p),\ \text{p-data-segment}\,(\text{p}\,(p,\ n)))$

```
; What we wish to prove is:
; (PLAUSIBLE-DATA-LINK-TABLEP
;   (TYPE-SPECIFICATION (P-DATA-SEGMENT (P P0 N)))
;   (LENGTH (M-MEM (I->M (R->I (R (P->R P0) N)) boot-lst LOAD-ADDR)))
;   (USR-DATA-LINKS (I-LINK-TABLES (R->I (P->R P0)) LOAD-ADDR)))

; By instantiating same-signature-plausible-data-link-tablep with
; (seg1 (p-data-segment p0))
; (seg2 (p-data-segment (p p0 n)))
; and applealing to same-signature-p, we can backchain from our goal to
```

```
; (PLAUSIBLE-DATA-LINK-TABLEP
;   (TYPE-SPECIFICATION (P-DATA-SEGMENT P0))
;   (LENGTH (M-MEM (I->M (R->I (R (P->R P0) N)) boot-lst LOAD-ADDR)))
;   (USR-DATA-LINKS (I-LINK-TABLES (R->I (P->R P0)) LOAD-ADDR)))

; Next I need to show that
; (length (m-mem (i->m (r->i (r pr n)) boot-lst load-addr)))
;  = (length (m-mem (i->m (r->i pr) boot-lst load-addr))).
; This follows from the fact that the r-prog-segment is unchanged by r and the
; signatures of both the usr-data-segment and sys-data-segment are unchanged.

; So now we undertake those proofs. (!)
```

THEOREM: same-signature-r-ins-step-call
((r-psw (r) = '**run**)
 ∧ (car (r-current-instruction (r)) = '**call**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ r-call-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-call-step (r-current-instruction (r), r))
  = r-prog-segment (r))
 ∧ same-signature (r-usr-data-segment (r),
   r-usr-data-segment (r-call-step (r-current-instruction (r),
   r)))
 ∧ same-signature (r-sys-data-segment (r),
   r-sys-data-segment (r-call-step (r-current-instruction (r),
   r))))

THEOREM: same-signature-r-ins-step-ret
((r-psw (r) = '**run**)
 ∧ (car (r-current-instruction (r)) = '**ret**)
 ∧ proper-r-statep (r, *load-addr*)
 ∧ r-ret-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-ret-step (r-current-instruction (r), r))
  = r-prog-segment (r))
 ∧ same-signature (r-usr-data-segment (r),
   r-usr-data-segment (r-ret-step (r-current-instruction (r),
   r)))
 ∧ same-signature (r-sys-data-segment (r),
   r-sys-data-segment (r-ret-step (r-current-instruction (r),
   r))))

THEOREM: same-signature-r-ins-step-locn
((r-psw (r) = '**run**)

$\wedge$ (car (r-current-instruction (r)) = `'locn`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-locn-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-locn-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-locn-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-locn-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-push-constant
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'push-constant`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-push-constant-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-push-constant-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-push-constant-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-push-constant-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-push-local
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'push-local`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-push-local-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-push-local-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-push-local-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-push-local-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-push-global
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'push-global`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-push-global-okp (r-current-instruction (r), r))

$\rightarrow$ ((r-prog-segment (r-push-global-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-push-global-step (r-current-instruction (r),
r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-push-global-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-push-ctrl-stk-free-size
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'push-ctrl-stk-free-size`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-ctrl-stk-free-size-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-push-ctrl-stk-free-size-step (r-current-instruction (r),
r))
= r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-push-ctrl-stk-free-size-step (r-current-instruction (r),
r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-push-ctrl-stk-free-size-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-push-temp-stk-free-size
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'push-temp-stk-free-size`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-temp-stk-free-size-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-push-temp-stk-free-size-step (r-current-instruction (r),
r))
= r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-push-temp-stk-free-size-step (r-current-instruction (r),
r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-push-temp-stk-free-size-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-push-temp-stk-index
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'push-temp-stk-index`)
$\land$ proper-r-statep (r, *load-addr*)
$\land$ r-push-temp-stk-index-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-push-temp-stk-index-step (r-current-instruction (r),

$r))$

$= \quad$ r-prog-segment $(r))$
$\land \quad$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-push-temp-stk-index-step (r-current-instruction (r),
$r)))$
$\land \quad$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-push-temp-stk-index-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-jump-if-temp-stk-full
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'jump-if-temp-stk-full`)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-jump-if-temp-stk-full-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ ((r-prog-segment (r-jump-if-temp-stk-full-step (r-current-instruction (r),
$r))$
$= \quad$ r-prog-segment $(r))$
$\land \quad$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-jump-if-temp-stk-full-step (r-current-instruction (r),
$r)))$
$\land \quad$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-jump-if-temp-stk-full-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-jump-if-temp-stk-empty
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'jump-if-temp-stk-empty`)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-jump-if-temp-stk-empty-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ ((r-prog-segment (r-jump-if-temp-stk-empty-step (r-current-instruction (r),
$r))$
$= \quad$ r-prog-segment $(r))$
$\land \quad$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-jump-if-temp-stk-empty-step (r-current-instruction (r),
$r)))$
$\land \quad$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-jump-if-temp-stk-empty-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-pop
((r-psw (r) = `'run`)
$\land$ (car (r-current-instruction (r)) = `'pop`)
$\land$ proper-r-statep $(r,\ \mathit{load\text{-}addr})$
$\land$ r-pop-okp (r-current-instruction $(r),\ r)$)
$\rightarrow$ ((r-prog-segment (r-pop-step (r-current-instruction $(r),\ r)$)

$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pop-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pop-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-pop*
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'pop***)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-pop*-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-pop*-step (r-current-instruction (r), r))
$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pop*-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pop*-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-popn
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'popn**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-popn-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-popn-step (r-current-instruction (r), r))
$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-popn-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-popn-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-pop-local
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'pop-local**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-pop-local-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-pop-local-step (r-current-instruction (r), r))
$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pop-local-step (r-current-instruction (r),

r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pop-local-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-pop-global
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'pop-global`)
∧ proper-r-statep (r, *load-addr*)
∧ r-pop-global-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-pop-global-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pop-global-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pop-global-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-pop-locn
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'pop-locn`)
∧ proper-r-statep (r, *load-addr*)
∧ r-pop-locn-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-pop-locn-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pop-locn-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pop-locn-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-pop-call
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'pop-call`)
∧ proper-r-statep (r, *load-addr*)
∧ r-pop-call-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-pop-call-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pop-call-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pop-call-step (r-current-instruction (r),

$r))))$

THEOREM: same-signature-r-ins-step-fetch-temp-stk
((r-psw(r) = **'run**)
∧ (car (r-current-instruction(r)) = **'fetch-temp-stk**)
∧ proper-r-statep$(r,\ load\text{-}addr)$
∧ r-fetch-temp-stk-okp (r-current-instruction(r), r))
→ ((r-prog-segment (r-fetch-temp-stk-step (r-current-instruction(r), r))
= r-prog-segment(r))
∧ same-signature (r-usr-data-segment(r),
r-usr-data-segment (r-fetch-temp-stk-step (r-current-instruction(r),
r)))
∧ same-signature (r-sys-data-segment(r),
r-sys-data-segment (r-fetch-temp-stk-step (r-current-instruction(r),
r))))

```
; I am having trouble getting same-signature-r-ins-step-DEPOSIT-TEMP-STK
; so I am proving a lemma just for it.  I'm sure there is one like this
; somewhere already here.  I am embarassed to confess that the following
; dumb helper is also beyond the abilities of the unassisted prover in its
; current disabled state!  I suppose I could find lemmas to make this go
; through too, but I'll just push it through by induction!
```

THEOREM: same-signature-r-ins-step-deposit-temp-stk-lemma-helper
$(n \not\simeq 0) \rightarrow ((((n - 1) - 1) - untag) < n)$

THEOREM: same-signature-r-ins-step-deposit-temp-stk-lemma
proper-r-statep$(r,\ load\text{-}addr)$
→ ((**if** (length (cdr (assoc (**'tstk**, r-sys-data-segment(r)))) − 1) = 0
then 0
else ((length (cdr (assoc (**'tstk**, r-sys-data-segment(r)))) − 1) − 1)
− untag (fetch (r-tsp(r), r-sys-data-segment(r))) **endif**
< length (cdr (assoc (**'tstk**, r-sys-data-segment(r)))))
= **t**)

THEOREM: same-signature-r-ins-step-deposit-temp-stk
((r-psw(r) = **'run**)
∧ (car (r-current-instruction(r)) = **'deposit-temp-stk**)
∧ proper-r-statep$(r,\ load\text{-}addr)$
∧ r-deposit-temp-stk-okp (r-current-instruction(r), r))
→ ((r-prog-segment (r-deposit-temp-stk-step (r-current-instruction(r), r))
= r-prog-segment(r))
∧ same-signature (r-usr-data-segment(r),
r-usr-data-segment (r-deposit-temp-stk-step (r-current-instruction(r),

$r)))$

$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-deposit-temp-stk-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-jump
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'jump`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-jump-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-jump-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-jump-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-jump-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-jump-case
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'jump-case`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-jump-case-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-jump-case-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-jump-case-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-jump-case-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-pushj
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'pushj`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-pushj-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-pushj-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-pushj-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-pushj-step (r-current-instruction (r),

$r))))$

THEOREM: same-signature-r-ins-step-popj
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ (\text{car}(\text{r-current-instruction}(r)) = \texttt{'popj})$
$\land\ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \text{r-popj-okp}(\text{r-current-instruction}(r),\ r))$
$\to\ ((\text{r-prog-segment}(\text{r-popj-step}(\text{r-current-instruction}(r),\ r))$
$= \text{r-prog-segment}(r))$
$\land\ \text{same-signature}(\text{r-usr-data-segment}(r),$
$\text{r-usr-data-segment}(\text{r-popj-step}(\text{r-current-instruction}(r),$
$r)))$
$\land\ \text{same-signature}(\text{r-sys-data-segment}(r),$
$\text{r-sys-data-segment}(\text{r-popj-step}(\text{r-current-instruction}(r),$
$r))))$

THEOREM: same-signature-r-ins-step-set-local
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ (\text{car}(\text{r-current-instruction}(r)) = \texttt{'set-local})$
$\land\ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \text{r-set-local-okp}(\text{r-current-instruction}(r),\ r))$
$\to\ ((\text{r-prog-segment}(\text{r-set-local-step}(\text{r-current-instruction}(r),\ r))$
$= \text{r-prog-segment}(r))$
$\land\ \text{same-signature}(\text{r-usr-data-segment}(r),$
$\text{r-usr-data-segment}(\text{r-set-local-step}(\text{r-current-instruction}(r),$
$r)))$
$\land\ \text{same-signature}(\text{r-sys-data-segment}(r),$
$\text{r-sys-data-segment}(\text{r-set-local-step}(\text{r-current-instruction}(r),$
$r))))$

THEOREM: same-signature-r-ins-step-set-global
$((\text{r-psw}(r) = \texttt{'run})$
$\land\ (\text{car}(\text{r-current-instruction}(r)) = \texttt{'set-global})$
$\land\ \text{proper-r-statep}(r,\ \mathit{load\text{-}addr})$
$\land\ \text{r-set-global-okp}(\text{r-current-instruction}(r),\ r))$
$\to\ ((\text{r-prog-segment}(\text{r-set-global-step}(\text{r-current-instruction}(r),\ r))$
$= \text{r-prog-segment}(r))$
$\land\ \text{same-signature}(\text{r-usr-data-segment}(r),$
$\text{r-usr-data-segment}(\text{r-set-global-step}(\text{r-current-instruction}(r),$
$r)))$
$\land\ \text{same-signature}(\text{r-sys-data-segment}(r),$
$\text{r-sys-data-segment}(\text{r-set-global-step}(\text{r-current-instruction}(r),$
$r))))$

THEOREM: same-signature-r-ins-step-test-nat-and-jump

$((\text{r-psw}(r) = \texttt{'run})$
$\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'test-nat-and-jump})$
$\land \quad \text{proper-r-statep}(r,\ \textit{load-addr})$
$\land \quad \text{r-test-nat-and-jump-okp}(\text{r-current-instruction}(r),\ r))$
$\to \quad ((\text{r-prog-segment}(\text{r-test-nat-and-jump-step}(\text{r-current-instruction}(r),\ r))$
$= \quad \text{r-prog-segment}(r))$
$\land \quad \text{same-signature}(\text{r-usr-data-segment}(r),$
$\text{r-usr-data-segment}(\text{r-test-nat-and-jump-step}(\text{r-current-instruction}(r),$
$r)))$
$\land \quad \text{same-signature}(\text{r-sys-data-segment}(r),$
$\text{r-sys-data-segment}(\text{r-test-nat-and-jump-step}(\text{r-current-instruction}(r),$
$r))))$

THEOREM: same-signature-r-ins-step-test-int-and-jump
$((\text{r-psw}(r) = \texttt{'run})$
$\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'test-int-and-jump})$
$\land \quad \text{proper-r-statep}(r,\ \textit{load-addr})$
$\land \quad \text{r-test-int-and-jump-okp}(\text{r-current-instruction}(r),\ r))$
$\to \quad ((\text{r-prog-segment}(\text{r-test-int-and-jump-step}(\text{r-current-instruction}(r),\ r))$
$= \quad \text{r-prog-segment}(r))$
$\land \quad \text{same-signature}(\text{r-usr-data-segment}(r),$
$\text{r-usr-data-segment}(\text{r-test-int-and-jump-step}(\text{r-current-instruction}(r),$
$r)))$
$\land \quad \text{same-signature}(\text{r-sys-data-segment}(r),$
$\text{r-sys-data-segment}(\text{r-test-int-and-jump-step}(\text{r-current-instruction}(r),$
$r))))$

THEOREM: same-signature-r-ins-step-test-bool-and-jump
$((\text{r-psw}(r) = \texttt{'run})$
$\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'test-bool-and-jump})$
$\land \quad \text{proper-r-statep}(r,\ \textit{load-addr})$
$\land \quad \text{r-test-bool-and-jump-okp}(\text{r-current-instruction}(r),\ r))$
$\to \quad ((\text{r-prog-segment}(\text{r-test-bool-and-jump-step}(\text{r-current-instruction}(r),\ r))$
$= \quad \text{r-prog-segment}(r))$
$\land \quad \text{same-signature}(\text{r-usr-data-segment}(r),$
$\text{r-usr-data-segment}(\text{r-test-bool-and-jump-step}(\text{r-current-instruction}(r),$
$r)))$
$\land \quad \text{same-signature}(\text{r-sys-data-segment}(r),$
$\text{r-sys-data-segment}(\text{r-test-bool-and-jump-step}(\text{r-current-instruction}(r),$
$r))))$

THEOREM: same-signature-r-ins-step-test-bitv-and-jump
$((\text{r-psw}(r) = \texttt{'run})$
$\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'test-bitv-and-jump})$
$\land \quad \text{proper-r-statep}(r,\ \textit{load-addr})$

$\wedge$ r-test-bitv-and-jump-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-test-bitv-and-jump-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-test-bitv-and-jump-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-test-bitv-and-jump-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-no-op
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'no-op`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-no-op-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-no-op-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-no-op-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-no-op-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-add-addr
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'add-addr`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-add-addr-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-add-addr-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-add-addr-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-add-addr-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-sub-addr
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'sub-addr`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-sub-addr-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-sub-addr-step (r-current-instruction (r), r))
= r-prog-segment (r))

∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-sub-addr-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-sub-addr-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-eq
((r-psw (r) = **'run**)
∧ (car (r-current-instruction (r)) = **'eq**)
∧ proper-r-statep (r, *load-addr*)
∧ r-eq-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-eq-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-eq-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-eq-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-lt-addr
((r-psw (r) = **'run**)
∧ (car (r-current-instruction (r)) = **'lt-addr**)
∧ proper-r-statep (r, *load-addr*)
∧ r-lt-addr-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-lt-addr-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-lt-addr-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-lt-addr-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-fetch
((r-psw (r) = **'run**)
∧ (car (r-current-instruction (r)) = **'fetch**)
∧ proper-r-statep (r, *load-addr*)
∧ r-fetch-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-fetch-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-fetch-step (r-current-instruction (r),
r)))

$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-fetch-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-deposit
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'deposit**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-deposit-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-deposit-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-deposit-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-deposit-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-add-int
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'add-int**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-add-int-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-add-int-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-add-int-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-add-int-step (r-current-instruction (r),
r))))

THEOREM: definedp-deposit-adp
definedp (*name*, deposit-adp (*val*, *adp*, *segment*)) = definedp (*name*, *segment*)

```
; We need a special lemma here because add-int-with-carry pushes two things.
; As usual, there is undoubtedly a set of lemmas already available for this
; if I only wanted to understand every detail of the old proofs.
```

THEOREM: adpp-add-adp-hack
(proper-r-statep (r, *load-addr*)
$\wedge$ ((1 + adp-offset (untag (r-tsp (r))))
< ((length (cdr (assoc (**'tstk**, r-sys-data-segment (r)))) − 1) − 1)))
$\rightarrow$ adpp (add-adp (untag (r-tsp (r)), 2), r-sys-data-segment (r))

THEOREM: same-signature-r-ins-step-add-int-with-carry
((r-psw (r) = **'run**)
$\land$ (car (r-current-instruction (r)) = **'add-int-with-carry**)
$\land$ proper-r-statep $(r,\ \textit{load-addr})$
$\land$ r-add-int-with-carry-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-add-int-with-carry-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-add-int-with-carry-step (r-current-instruction (r),
r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-add-int-with-carry-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-add1-int
((r-psw (r) = **'run**)
$\land$ (car (r-current-instruction (r)) = **'add1-int**)
$\land$ proper-r-statep $(r,\ \textit{load-addr})$
$\land$ r-add1-int-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-add1-int-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-add1-int-step (r-current-instruction (r),
r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-add1-int-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-sub-int
((r-psw (r) = **'run**)
$\land$ (car (r-current-instruction (r)) = **'sub-int**)
$\land$ proper-r-statep $(r,\ \textit{load-addr})$
$\land$ r-sub-int-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-sub-int-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-sub-int-step (r-current-instruction (r),
r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-sub-int-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-sub-int-with-carry
((r-psw (r) = **'run**)
$\land$ (car (r-current-instruction (r)) = **'sub-int-with-carry**)

$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-sub-int-with-carry-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-sub-int-with-carry-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-sub-int-with-carry-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-sub-int-with-carry-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-sub1-int
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'sub1-int`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-sub1-int-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-sub1-int-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-sub1-int-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-sub1-int-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-neg-int
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'neg-int`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-neg-int-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-neg-int-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-neg-int-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-neg-int-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-lt-int
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'lt-int`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-lt-int-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-lt-int-step (r-current-instruction (r), r))

$$
\begin{array}{l}
\quad\quad = \quad \text{r-prog-segment}\,(r))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-usr-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-usr-data-segment}\,(\text{r-lt-int-step}\,(\text{r-current-instruction}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r)))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-sys-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-sys-data-segment}\,(\text{r-lt-int-step}\,(\text{r-current-instruction}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-int-to-nat

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run})\\
\land \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'int-to-nat})\\
\land \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})\\
\land \quad \text{r-int-to-nat-okp}\,(\text{r-current-instruction}\,(r),\ r))\\
\rightarrow \quad ((\text{r-prog-segment}\,(\text{r-int-to-nat-step}\,(\text{r-current-instruction}\,(r),\ r))\\
\quad\quad = \quad \text{r-prog-segment}\,(r))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-usr-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-usr-data-segment}\,(\text{r-int-to-nat-step}\,(\text{r-current-instruction}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r)))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-sys-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-sys-data-segment}\,(\text{r-int-to-nat-step}\,(\text{r-current-instruction}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-add-nat

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run})\\
\land \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'add-nat})\\
\land \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})\\
\land \quad \text{r-add-nat-okp}\,(\text{r-current-instruction}\,(r),\ r))\\
\rightarrow \quad ((\text{r-prog-segment}\,(\text{r-add-nat-step}\,(\text{r-current-instruction}\,(r),\ r))\\
\quad\quad = \quad \text{r-prog-segment}\,(r))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-usr-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-usr-data-segment}\,(\text{r-add-nat-step}\,(\text{r-current-instruction}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r)))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-sys-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-sys-data-segment}\,(\text{r-add-nat-step}\,(\text{r-current-instruction}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-add-nat-with-carry

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run})\\
\land \quad (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'add-nat-with-carry})\\
\land \quad \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr})\\
\land \quad \text{r-add-nat-with-carry-okp}\,(\text{r-current-instruction}\,(r),\ r))\\
\rightarrow \quad ((\text{r-prog-segment}\,(\text{r-add-nat-with-carry-step}\,(\text{r-current-instruction}\,(r),\ r))\\
\quad\quad = \quad \text{r-prog-segment}\,(r))\\
\quad\quad \land \quad \text{same-signature}\,(\text{r-usr-data-segment}\,(r),\\
\quad\quad\quad\quad\quad\quad\quad \text{r-usr-data-segment}\,(\text{r-add-nat-with-carry-step}\,(\text{r-current-instruction}\,(r),
\end{array}
$$

r)))

∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-add-nat-with-carry-step (r-current-instruction (r),
r))))

Theorem: same-signature-r-ins-step-add1-nat
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'add1-nat`)
∧ proper-r-statep (r, *load-addr*)
∧ r-add1-nat-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-add1-nat-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-add1-nat-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-add1-nat-step (r-current-instruction (r),
r))))

Theorem: same-signature-r-ins-step-sub-nat
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'sub-nat`)
∧ proper-r-statep (r, *load-addr*)
∧ r-sub-nat-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-sub-nat-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-sub-nat-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-sub-nat-step (r-current-instruction (r),
r))))

Theorem: same-signature-r-ins-step-sub-nat-with-carry
((r-psw (r) = `'run`)
∧ (car (r-current-instruction (r)) = `'sub-nat-with-carry`)
∧ proper-r-statep (r, *load-addr*)
∧ r-sub-nat-with-carry-okp (r-current-instruction (r), r))
→ ((r-prog-segment (r-sub-nat-with-carry-step (r-current-instruction (r), r))
= r-prog-segment (r))
∧ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-sub-nat-with-carry-step (r-current-instruction (r),
r)))
∧ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-sub-nat-with-carry-step (r-current-instruction (r),

$r))))$

THEOREM: same-signature-r-ins-step-sub1-nat

$$
\begin{array}{l}
((\text{r-psw}(r) = \texttt{'run}) \\
\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'sub1-nat}) \\
\land \quad \text{proper-r-statep}(r,\ \textit{load-addr}) \\
\land \quad \text{r-sub1-nat-okp}(\text{r-current-instruction}(r),\ r)) \\
\to \quad ((\text{r-prog-segment}(\text{r-sub1-nat-step}(\text{r-current-instruction}(r),\ r)) \\
\qquad = \quad \text{r-prog-segment}(r)) \\
\qquad \land \quad \text{same-signature}(\text{r-usr-data-segment}(r), \\
\qquad\qquad\qquad \text{r-usr-data-segment}(\text{r-sub1-nat-step}(\text{r-current-instruction}(r), \\
\qquad\qquad\qquad\qquad r))) \\
\qquad \land \quad \text{same-signature}(\text{r-sys-data-segment}(r), \\
\qquad\qquad\qquad \text{r-sys-data-segment}(\text{r-sub1-nat-step}(\text{r-current-instruction}(r), \\
\qquad\qquad\qquad\qquad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-lt-nat

$$
\begin{array}{l}
((\text{r-psw}(r) = \texttt{'run}) \\
\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'lt-nat}) \\
\land \quad \text{proper-r-statep}(r,\ \textit{load-addr}) \\
\land \quad \text{r-lt-nat-okp}(\text{r-current-instruction}(r),\ r)) \\
\to \quad ((\text{r-prog-segment}(\text{r-lt-nat-step}(\text{r-current-instruction}(r),\ r)) \\
\qquad = \quad \text{r-prog-segment}(r)) \\
\qquad \land \quad \text{same-signature}(\text{r-usr-data-segment}(r), \\
\qquad\qquad\qquad \text{r-usr-data-segment}(\text{r-lt-nat-step}(\text{r-current-instruction}(r), \\
\qquad\qquad\qquad\qquad r))) \\
\qquad \land \quad \text{same-signature}(\text{r-sys-data-segment}(r), \\
\qquad\qquad\qquad \text{r-sys-data-segment}(\text{r-lt-nat-step}(\text{r-current-instruction}(r), \\
\qquad\qquad\qquad\qquad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-mult2-nat

$$
\begin{array}{l}
((\text{r-psw}(r) = \texttt{'run}) \\
\land \quad (\text{car}(\text{r-current-instruction}(r)) = \texttt{'mult2-nat}) \\
\land \quad \text{proper-r-statep}(r,\ \textit{load-addr}) \\
\land \quad \text{r-mult2-nat-okp}(\text{r-current-instruction}(r),\ r)) \\
\to \quad ((\text{r-prog-segment}(\text{r-mult2-nat-step}(\text{r-current-instruction}(r),\ r)) \\
\qquad = \quad \text{r-prog-segment}(r)) \\
\qquad \land \quad \text{same-signature}(\text{r-usr-data-segment}(r), \\
\qquad\qquad\qquad \text{r-usr-data-segment}(\text{r-mult2-nat-step}(\text{r-current-instruction}(r), \\
\qquad\qquad\qquad\qquad r))) \\
\qquad \land \quad \text{same-signature}(\text{r-sys-data-segment}(r), \\
\qquad\qquad\qquad \text{r-sys-data-segment}(\text{r-mult2-nat-step}(\text{r-current-instruction}(r), \\
\qquad\qquad\qquad\qquad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-mult2-nat-with-carry-out

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run}) \\
\ \land\ \ (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'mult2-nat-with-carry-out}) \\
\ \land\ \ \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr}) \\
\ \land\ \ \text{r-mult2-nat-with-carry-out-okp}\,(\text{r-current-instruction}\,(r),\ r)) \\
\to\ \ ((\text{r-prog-segment}\,(\text{r-mult2-nat-with-carry-out-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r)) \\
\qquad = \ \text{r-prog-segment}\,(r)) \\
\qquad \land\ \ \text{same-signature}\,(\text{r-usr-data-segment}\,(r), \\
\qquad\qquad\qquad\quad \text{r-usr-data-segment}\,(\text{r-mult2-nat-with-carry-out-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r))) \\
\qquad \land\ \ \text{same-signature}\,(\text{r-sys-data-segment}\,(r), \\
\qquad\qquad\qquad\quad \text{r-sys-data-segment}\,(\text{r-mult2-nat-with-carry-out-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-div2-nat

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run}) \\
\ \land\ \ (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'div2-nat}) \\
\ \land\ \ \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr}) \\
\ \land\ \ \text{r-div2-nat-okp}\,(\text{r-current-instruction}\,(r),\ r)) \\
\to\ \ ((\text{r-prog-segment}\,(\text{r-div2-nat-step}\,(\text{r-current-instruction}\,(r),\ r)) \\
\qquad = \ \text{r-prog-segment}\,(r)) \\
\qquad \land\ \ \text{same-signature}\,(\text{r-usr-data-segment}\,(r), \\
\qquad\qquad\qquad\quad \text{r-usr-data-segment}\,(\text{r-div2-nat-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r))) \\
\qquad \land\ \ \text{same-signature}\,(\text{r-sys-data-segment}\,(r), \\
\qquad\qquad\qquad\quad \text{r-sys-data-segment}\,(\text{r-div2-nat-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-or-bitv

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run}) \\
\ \land\ \ (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'or-bitv}) \\
\ \land\ \ \text{proper-r-statep}\,(r,\ \mathit{load\text{-}addr}) \\
\ \land\ \ \text{r-or-bitv-okp}\,(\text{r-current-instruction}\,(r),\ r)) \\
\to\ \ ((\text{r-prog-segment}\,(\text{r-or-bitv-step}\,(\text{r-current-instruction}\,(r),\ r)) \\
\qquad = \ \text{r-prog-segment}\,(r)) \\
\qquad \land\ \ \text{same-signature}\,(\text{r-usr-data-segment}\,(r), \\
\qquad\qquad\qquad\quad \text{r-usr-data-segment}\,(\text{r-or-bitv-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r))) \\
\qquad \land\ \ \text{same-signature}\,(\text{r-sys-data-segment}\,(r), \\
\qquad\qquad\qquad\quad \text{r-sys-data-segment}\,(\text{r-or-bitv-step}\,(\text{r-current-instruction}\,(r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r))))
\end{array}
$$

THEOREM: same-signature-r-ins-step-and-bitv

$$
\begin{array}{l}
((\text{r-psw}\,(r) = \texttt{'run}) \\
\ \land\ \ (\text{car}\,(\text{r-current-instruction}\,(r)) = \texttt{'and-bitv})
\end{array}
$$

$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-and-bitv-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-and-bitv-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-and-bitv-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-and-bitv-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-not-bitv
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'not-bitv`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-not-bitv-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-not-bitv-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-not-bitv-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-not-bitv-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-xor-bitv
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'xor-bitv`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-xor-bitv-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-xor-bitv-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-xor-bitv-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-xor-bitv-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-rsh-bitv
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'rsh-bitv`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-rsh-bitv-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-rsh-bitv-step (r-current-instruction (r), r))

$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-rsh-bitv-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-rsh-bitv-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-lsh-bitv
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'lsh-bitv**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-lsh-bitv-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-lsh-bitv-step (r-current-instruction (r), r))
$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-lsh-bitv-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-lsh-bitv-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-or-bool
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'or-bool**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-or-bool-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-or-bool-step (r-current-instruction (r), r))
$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-or-bool-step (r-current-instruction (r),
$r)))$
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-or-bool-step (r-current-instruction (r),
$r))))$

THEOREM: same-signature-r-ins-step-and-bool
((r-psw (r) = **'run**)
$\wedge$ (car (r-current-instruction (r)) = **'and-bool**)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-and-bool-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-and-bool-step (r-current-instruction (r), r))
$=$ r-prog-segment $(r))$
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-and-bool-step (r-current-instruction (r),

r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-and-bool-step (r-current-instruction (r),
r))))

THEOREM: same-signature-r-ins-step-not-bool
((r-psw (r) = `'run`)
$\wedge$ (car (r-current-instruction (r)) = `'not-bool`)
$\wedge$ proper-r-statep (r, *load-addr*)
$\wedge$ r-not-bool-okp (r-current-instruction (r), r))
$\rightarrow$ ((r-prog-segment (r-not-bool-step (r-current-instruction (r), r))
= r-prog-segment (r))
$\wedge$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-not-bool-step (r-current-instruction (r),
r)))
$\wedge$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-not-bool-step (r-current-instruction (r),
r))))

```
; Now we put these all together.  We'll do each segment individually
; because the case splitting is pretty bad here:
```

THEOREM: same-signature-r-prog-segment-r-ins-step
((r-psw (r) = `'run`)
$\wedge$ r-ins-okp (r-current-instruction (r), r)
$\wedge$ proper-r-statep (r, *load-addr*))
$\rightarrow$ (r-prog-segment (r-ins-step (r-current-instruction (r), r))
= r-prog-segment (r))

THEOREM: same-signature-r-usr-data-segment-r-ins-step
((r-psw (r) = `'run`)
$\wedge$ r-ins-okp (r-current-instruction (r), r)
$\wedge$ proper-r-statep (r, *load-addr*))
$\rightarrow$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-ins-step (r-current-instruction (r),
r)))

THEOREM: same-signature-r-sys-data-segment-r-ins-step
((r-psw (r) = `'run`)
$\wedge$ r-ins-okp (r-current-instruction (r), r)
$\wedge$ proper-r-statep (r, *load-addr*))
$\rightarrow$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-ins-step (r-current-instruction (r),
r)))

THEOREM: same-signature-r-step
proper-r-statep $(r,\ load\text{-}addr)$
$\rightarrow$ ((r-prog-segment (r-step (r)) = r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r-step (r)))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r-step (r))))

THEOREM: same-signature-r-transitivity-lemma1
(($n \in \mathbf{N}$)
$\land$ ($n \neq 0$)
$\land$ same-signature (r-usr-data-segment (r-step (r)),
r-usr-data-segment (r (r-step (r), $n - 1$)))
$\land$ proper-r-statep $(r,\ load\text{-}addr)$
$\land$ ($\neg$ errorp (r-psw (r (r-step (r), $n - 1$)))))
$\rightarrow$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r (r-step (r), $n - 1$)))

THEOREM: same-signature-r-transitivity-lemma2
(($n \in \mathbf{N}$)
$\land$ ($n \neq 0$)
$\land$ same-signature (r-sys-data-segment (r-step (r)),
r-sys-data-segment (r (r-step (r), $n - 1$)))
$\land$ proper-r-statep $(r,\ load\text{-}addr)$
$\land$ ($\neg$ errorp (r-psw (r (r-step (r), $n - 1$)))))
$\rightarrow$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r (r-step (r), $n - 1$)))

THEOREM: same-signature-r
(proper-r-statep $(r,\ load\text{-}addr)$ $\land$ ($\neg$ errorp (r-psw (r $(r,\ n)$))))
$\rightarrow$ ((r-prog-segment (r $(r,\ n)$) = r-prog-segment (r))
$\land$ same-signature (r-usr-data-segment (r),
r-usr-data-segment (r $(r,\ n)$))
$\land$ same-signature (r-sys-data-segment (r),
r-sys-data-segment (r $(r,\ n)$)))

```
; So now we can derive from this same-signature invariance the fact that
; the length of the mapped down memory is preserved by the r machine.

; The key lemma relating same-signature and link-mem...
```

THEOREM: same-signature-implies-length-link-mem
(same-signature $(usr\text{-}data\text{-}segment1,\ usr\text{-}data\text{-}segment2)$
$\land$ same-signature $(sys\text{-}data\text{-}segment1,\ sys\text{-}data\text{-}segment2)$)

$\rightarrow$ (length (link-mem (*boot-lst*,
load-addr,
usr-data-segment2,
prog-segment,
sys-data-segment2,
tables,
word-size))
= length (link-mem (*boot-lst*,
load-addr,
usr-data-segment1,
prog-segment,
sys-data-segment1,
tables,
word-size)))

```
; ... which we can use to show that the mapped down memory is the same size:
```

THEOREM: length-m-mem-r->i->m
(proper-r-statep (r, *load-addr*)
$\wedge$ (*load-addr* $\in$ **N**)
$\wedge$ ($\neg$ errorp (r-psw (r (r, n))))
$\wedge$ (r-word-size (r) = 32))
$\rightarrow$ (length (m-mem (i->m (r->i (r (r, n)), *boot-lst*, *load-addr*)))
= length (m-mem (i->m (r->i (r), *boot-lst*, *load-addr*))))

THEOREM: i-usr-data-segment-p->r->i
i-usr-data-segment (r->i (p->r (p))) = p-data-segment (p)

THEOREM: once-errorp-always-errorp{mapping}
(proper-p-statep (*p0*)
$\wedge$ p-loadablep (*p0*, *load-addr*)
$\wedge$ ($\neg$ errorp (p-psw (p (*p0*, n)))))
$\rightarrow$ ($\neg$ errorp (r-psw (r (p->r (*p0*), n))))

THEOREM: length-m-mem-p->r->i->m
length (m-mem (i->m (r->i (p->r (*p0*)), *boot-lst*, *load-addr*)))
= (*load-addr*
+ segment-length (p-data-segment (*p0*))
+ segment-length (icompile (p-prog-segment (*p0*)))
+ segment-length (p->r_sys-data-segment (p-ctrl-stk (*p0*),
p-max-ctrl-stk-size (*p0*),
p-temp-stk (*p0*),
p-max-temp-stk-size (*p0*))))

THEOREM: i-prog-segment-p->r->i
i-prog-segment (r->i (p->r ($p0$))) = icompile (p-prog-segment ($p0$))

THEOREM: consolidated-plausibility-argument-lemma
(proper-p-statep ($p0$)
∧ p-loadablep ($p0$, $load\text{-}addr$)
∧ ($load\text{-}addr \in \mathbf{N}$)
∧ (p-word-size ($p0$) = 32)
∧ (¬ errorp (p-psw (p ($p0$, n))))
∧ plausible-data-link-tablep (type-specification (p-data-segment ($p0$)),
$load\text{-}addr$
\+ segment-length (p-data-segment ($p0$))
\+ segment-length (icompile (p-prog-segment ($p0$)))
\+ segment-length (p->r_sys-data-segment (p-ctrl-stk ($p0$),
p-max-ctrl-stk-size ($p0$),
p-temp-stk ($p0$),
p-max-temp-stk-size ($p0$))),
link-table-for-segment (p-data-segment ($p0$),
$load\text{-}addr$)))
→ plausible-data-link-tablep (type-specification (p-data-segment (p ($p0$, n))),
length (m-mem (i->m (r->i (r (p->r ($p0$), n)),
$boot\text{-}lst$,
$load\text{-}addr$))),
usr-data-links (i-link-tables (r->i (p->r ($p0$)),
$load\text{-}addr$)))

```
; So to establish the desired plausibility condition we can backchain through
; the above and reduce the goal to:

; (plausible-data-link-tablep
;            (type-specification (p-data-segment p0))
;            (plus (segment-length (icompile (p-prog-segment p0)))
;                  (segment-length (p-data-segment p0))
;                  (segment-length
;                   (p->r_sys-data-segment (p-ctrl-stk p0)
;                                          (p-max-ctrl-stk-size p0)
;                                          (p-temp-stk p0)
;                                          (p-max-temp-stk-size p0))))
;            (link-table-for-segment (p-data-segment p0)
;                                    (segment-length (icompile (p-prog-segment p0)))))

; Which is in fact an instance of

; (plausible-data-link-tablep
```

```
;             (type-specification seg)
;             (plus i (segment-length seg) j)
;             (link-table-for-segment seg i))

; It is this more general plausibility condition that we now attack.
```

THEOREM: not-lessp-segment-length-plus-link-table-entry-length
segment-length $(segment)$
$\not<$ (link-table-entry $(name,\ segment)$ + length (cdr (assoc $(name,\ segment)$)))

THEOREM: lessp-absolute-address-segment-length-rewrite{weakened-instance}
(definedp $(name,\ segment)$ $\wedge$ (i = length (cdr (assoc $(name,\ segment)$))))
$\rightarrow$ (segment-length $(segment)$ $\not<$ (i + link-table-entry $(name,\ segment)$))

THEOREM: plausible-data-link-tablep-link-table-for-segment
(subsegmentp $(seg1,\ seg2)$
$\wedge$ $(i \in \mathbf{N})$
$\wedge$ ($max \not<$ (i + segment-length $(seg2)$)))
$\rightarrow$ plausible-data-link-tablep (type-specification $(seg1)$,
max,
link-table-for-segment $(seg2,\ i)$)

THEOREM: r-usr-data-segment-p->r
r-usr-data-segment (p->r (p)) = p-data-segment (p)

```
; Note:  The use and disable hints below were unnecessary
; before load-addr became an insidious free variable.
```

THEOREM: proper-p-statep-implies-proper-i-usr-data-segmentp
(proper-p-statep (p) $\wedge$ p-loadablep $(p,\ load\text{-}addr)$ $\wedge$ $(load\text{-}addr \in \mathbf{N})$)
$\rightarrow$ proper-i-usr-data-segmentp (p-data-segment (p), r->i (p->r (p)))

EVENT: Disable length-m-mem-p->r->i->m.

EVENT: Disable length-m-mem-r->i->m.

THEOREM: consolidated-plausibility-argument
(proper-p-statep $(p0)$
$\wedge$ p-loadablep $(p0,\ load\text{-}addr)$
$\wedge$ $(load\text{-}addr \in \mathbf{N})$
$\wedge$ ($\neg$ errorp (p-psw (p $(p0,\ n)$)))
$\wedge$ (p-word-size $(p0)$ = 32))

$$\rightarrow \quad \text{plausible-data-link-tablep}\,(\text{type-specification}\,(\text{p-data-segment}\,(\text{p}\,(p0,\ n))),\ \text{length}\,(\text{m-mem}\,(\text{i->m}\,(\text{r->i}\,(\text{r}\,(\text{p->r}\,(p0),\ n)),\ \mathit{boot\text{-}lst},\ \mathit{load\text{-}addr}))),\ \text{usr-data-links}\,(\text{i-link-tables}\,(\text{r->i}\,(\text{p->r}\,(p0)),\ \mathit{load\text{-}addr})))$$

```
; Book:  Top-goal.  Ok, so here, at long last, is the statement that
; Piton is implemented on FM9001!
```

THEOREM: top-goal

$$\begin{aligned}
&(\text{proper-p-statep}\,(p0)\\
&\ \wedge\ (\mathit{load\text{-}addr} \in \mathbf{N})\\
&\ \wedge\ \text{p-loadablep}\,(p0,\ \mathit{load\text{-}addr})\\
&\ \wedge\ (\text{p-word-size}\,(p0) = 32)\\
&\ \wedge\ (pn = \text{p}\,(p0,\ n))\\
&\ \wedge\ (\neg\ \text{errorp}\,(\text{p-psw}\,(pn)))\\
&\ \wedge\ (ts = \text{type-specification}\,(\text{p-data-segment}\,(pn))))\\
&\rightarrow\ (\text{p-data-segment}\,(pn)\\
&\quad = \text{display-fm9001-data-segment}\,(\text{fm9001}\,(\text{load}\,(p0,\ \mathit{boot\text{-}lst},\ \mathit{load\text{-}addr}),\ \text{fm9001-clock}\,(p0,\ n)),\ ts,\ \text{link-tables}\,(p0,\ \mathit{load\text{-}addr})))
\end{aligned}$$

EVENT: Set the status of all events. The status of each event is to be set as follows. Each 'boot-strap' satellite is to be returned to the initial condition. Each 'add-shell' satellite is to be enabled. enabled. Anything not otherwise mentioned is to be disabled. Name this event 'close-data-base-6'.

```
; The rest of this file is related to the implementation of the
; fm9001.  Since this version of Piton is built on an fm9001 events
; file that does not go down to that level, we omit the events below.

#|
(enable-theory (fm9001=fm9001-interpreter
                fm9001-statep
                map-down
                map-up-inverts-map-down
                fm9001-statep-implies-fm9001-state-structure
                fm9001-statep-implies-macrocycle-invariant-lemma1
                fm9001-statep-implies-macrocycle-invariant
                fm9001=chip-system-lemma1
                fm9001=chip-system))
```

```
; Book:  Piton to Dual-Eval.  We now wish to connect Piton to DUAL-EVAL, i.e.,
; to eliminate FM9001 and go straight down to the bottom.  The motivation is
; to make it easier for the user of Piton to connect his machine to the gate
; level, so that stacks can be built step-wise:  Each layer should finish by
; connecting to the bottom in one step.


; We now develop the lemmas leading to the theorem that maps Piton all the
; way down to the dual-eval level.

(defn ram-tree-hint (lst n)
  (cond ((zerop n) t)
        ((nlistp lst) t)
        (t (cons (ram-tree-hint (firstn (exp 2 (sub1 n)) lst)
                               (sub1 n))
                 (ram-tree-hint (restn (exp 2 (sub1 n)) lst)
                               (sub1 n))))))

(prove-lemma not-equal-times-2-x-1 (rewrite)
  (not (equal (times 2 x) 1))
  ((enable times-0)))

(prove-lemma bvp-listp-firstn (rewrite)
  (implies (bvp-listp lst)
           (bvp-listp (firstn n lst))))

(prove-lemma bvp-listp-restn (rewrite)
  (implies (bvp-listp lst)
           (bvp-listp (restn n lst))))

(prove-lemma memory-okp-ram-tree (rewrite)
  (implies (bvp-listp lst)
           (memory-okp n 32 (ram-tree lst (exp 2 n))))
  ((enable quotient-times-instance-temp)
   (induct (ram-tree-hint lst n))))

(prove-lemma memory-okp-ram-tree-bridge (rewrite)
  (implies (and (bvp-listp lst)
                (equal k (exp 2 n)))
           (memory-okp n 32 (ram-tree lst k))))

(disable memory-okp-ram-tree)
```

```
(prove-lemma bvp-listp-cons (rewrite)
  (equal (bvp-listp (cons bv lst))
         (and (bvp bv)
              (equal (length bv) 32)
              (bvp-listp lst))))

(prove-lemma all-ramp-mem-ram-tree (rewrite)
  (implies (not (lessp (length lst) (exp 2 n)))
           (all-ramp-mem n (ram-tree lst (exp 2 n))))
  ((enable quotient-times-instance-temp length-firstn length-restn  lessp-0-exp)
   (induct (ram-tree-hint lst n))))

(prove-lemma all-ramp-mem-ram-tree-bridge (rewrite)
  (implies (and (equal k (exp 2 n))
                (not (lessp (length lst) k)))
           (all-ramp-mem n (ram-tree lst k))))

(disable all-ramp-mem-ram-tree)

(prove-lemma fm9001-statep-load (rewrite)
  (implies (and (proper-p-statep p0)
                (p-loadablep p0 load-addr)
                (equal (p-word-size p0) 32))
           (fm9001-statep (load p0 boot-lst load-addr)))
  ((disable proper-p-statep p-loadablep link-word  p->r_cfp p->r_csp p->r_tsp
            p->r_sys-data-segment r->i_pc icompile r->i_psw i-link-tables
            all-ramp-mem memory-okp ram-tree)
   (enable *1*exp length-cons size-link-word bvp-link-word bvp-listp-append
           bvp-listp-boot-code
           bvp-listp-link-segment bool-to-logical-non-f)))

(disable fm9001=fm9001-interpreter)

(prove-lemma piton-to-dual-eval (rewrite)
  (implies (and (proper-p-statep p0)
                (p-loadablep p0 load-addr)
                (numberp load-addr)
                (equal (p-word-size p0) 32)
                (not (errorp (p-psw (p p0 n))))
                (equal ts (type-specification (p-data-segment (p p0 n))))
                (chip-system& netlist)
                (chip-system-operating-inputs-p inputs
                                                (total-microcycles (map-down (load p0 boot-lst load-addr))
                                                                   (map-up-inputs inputs)
```

```
                                                                      (fm9001-clock p0 n)))
                 (operating-inputs-p (map-up-inputs inputs)
                                     (total-microcycles (map-down (load p0 boot-lst load-addr))
                                                        (map-up-inputs inputs)
                                                        (fm9001-clock p0 n))))
            (equal (p-data-segment (p p0 n))
                   (display-fm9001-data-segment
                    (map-up
                     (simulate-dual-eval-2 'chip-system inputs
                                           (map-down (load p0 boot-lst load-addr))
                                           netlist
                                           (total-microcycles (map-down (load p0 boot-lst load-addr))
                                                              (map-up-inputs inputs)
                                                              (fm9001-clock p0 n))))
                    ts
                    (link-tables p0 load-addr))))
  ((enable fm9001=chip-system)
   (disable proper-p-statep p-loadablep errorp type-specification p chip-system&
            chip-system-operating-inputs-p total-microcycles map-down load map-up-inputs
            fm9001-clock operating-inputs-p display-fm9001-data-segment map-up
            simulate-dual-eval-2 link-tables)
   (use (top-goal (pn (p p0 n))))))

(set-status close-data-base-7 t ((boot-strap initial)
                                 (add-shell enable)
                                 ((defn *1*defn) enable)
                                 (otherwise disable)))

|#
```

EVENT: Make the library "`piton`" and compile it.

Index