```
#|
 Copyright (C) 1993 by Matt Wilding.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matt Wilding 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 Matt Wilding 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.
|#
```

EVENT: Start with the library "`piton`" using the compiled version.

```
;; NIM Piton proof
;; Matt Wilding 4-15-92

;; modified 7-92 to work on Piton library

;; This script takes 10 hours to run on a 64 meg Sparc2

;; This work is described in Technical Report #78.  A presentation
;; about this work was made at the CLI research review in April 92.

#|
Nim is a game played with piles of matches.  Two opponents alternate
taking at least one match from exactly one pile until there are no
matches left.  The player who takes the last match loses.

Piton is an assembly-level language with a formal semantics and a
verified compiler.  Piton is described in CLI tech report #22.  One of
the machines to which Piton is targeted is FM9001, a microprocessor
that has a formal semantics and that has been fabricated.
```

This proof script leads NQTHM to a proof that a Piton program that "plays" Nim does so optimally. Informally, this means

a) A Piton program (see function CM-PROG) when run on the Piton interpreter and given as input a reasonable Nim state yields a new Nim state equal to what is calculated by function COMPUTER-MOVE. (See event COMPUTER-MOVE-IMPLEMENTED.)

b) COMPUTER-MOVE generates valid moves. That is, it removes at least one match from exactly one pile. (VALID-MOVEP-COMPUTER-MOVE)

c) Depth-first search of the state space of all possible moves is used to define what is meant by optimal Nim play. Exhaustive search is used to find if there is a strategy for a Nim player to ensures eventual victory from the current Nim state. An optimal strategy transforms any state for which there exists such a winning strategy into a state from which exhaustive search can find no winning strategy.

Exhaustive search of all possible moves from a NIM state is formalized in the function WSP. The optimalilty of the strategy COMPUTER-MOVE is proved in the event COMPUTER-MOVE-WORKS.

d) The FM9001 Piton compiler correctness theorem assumes that the Piton state that is to be run contains valid Piton code, fit into an FM9001's memory, and use constants of word size 32. A Piton state (used in the Indiana test described below) was constructed that contains the Nim program, and is proved to meet the compiler correctness constraints (CM-PROG-FM9001-LOADABLE)

The algorithm used by the program is non-obvious and very efficient, avoiding the need to search. (I invented the programming trick only to subsequently discover that it has been known since the beginning of the century.) See tech report #78 for a description.

(Constant bounds on the number of Piton instructions executed while running this program have been proved using PC-NQTHM. These events are not included in this NQTHM-processable script, but the theorem is included in comments later in this file for completeness.)

This script was developed using only those events from the Piton library necessary to define the interpreter and the naturals library. In July 92 it was modified somewhat to "fit" onto the Piton library that sits on top of the FM9001 library. The immediate motivation was

```
to make it easier to include in the upcoming NQTHM-1992 proveall
release being put together by Boyer.  The Piton library contains the
events of the FM9001 library.  The FM9001 library contains an older
version of the naturals library (though not explicitly with a
note-lib).  Thus, this script requires only the Piton library.

In May 92, in consultation with researchers at Indiana University, I
compiled the Nim Piton program for the FM9001 and sent the image to
Indiana.  They ran the image and generated an optimal NIM move on a
fabricated FM9001 they had wired up.  The intial image and part of the
resulting image is included in a comment at the end of this script.  A
non-trivial program compiled using a reasonably-complex compiler for a
small microprocessor worked without any conventional testing done on
any of the components, and everyone was pleased but not surprised.

|#
```

EVENT: Set the status of the events in the theory addition. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled. Name this event 'addition-on'.

EVENT: Set the status of the events in the theory multiplication. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled. Name this event 'multiplication-on'.

EVENT: Set the status of the events in the theory remainders. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled. Name this event 'remainders-on'.

EVENT: Set the status of the events in the theory quotients. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled. Name this event 'quotients-on'.

EVENT: Set the status of the events in the theory exponentiation. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled. Name this event 'exponentiation-on'.

EVENT: Set the status of the events in the theory logs. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled.

Name this event 'logs-on'.

EVENT: Set the status of the events in the theory gcds. The status of each event is to be set as follows. Anything not otherwise mentioned is to be enabled. Name this event 'gcds-on'.

DEFINITION: $\text{clock-plus}\,(x,\ y) = (x + y)$

THEOREM: p-add1
$\text{p}\,(p0,\ 1 + n) = \text{p}\,(\text{p-step}\,(p0),\ n)$

THEOREM: p-0
$(n \simeq \mathtt{0}) \rightarrow (\text{p}\,(p0,\ n) = p0)$

THEOREM: clock-plus-function
$\text{p}\,(p0,\ \text{clock-plus}\,(x,\ y)) = \text{p}\,(\text{p}\,(p0,\ x),\ y)$

EVENT: Disable p-add1.

THEOREM: clock-plus-add1
$\text{p}\,(p0,\ \text{clock-plus}\,(1 + x,\ y)) = \text{p}\,(p0,\ 1 + \text{clock-plus}\,(x,\ y))$

EVENT: Disable clock-plus.

THEOREM: clock-plus-0
$(x \simeq \mathtt{0}) \rightarrow (\text{clock-plus}\,(x,\ y) = \text{fix}\,(y))$

THEOREM: fix-clock-plus
$\text{fix}\,(\text{clock-plus}\,(x,\ y)) = \text{clock-plus}\,(x,\ y)$

THEOREM: p-step1-opener
$\text{p-step1}\,(\text{cons}\,(opcode,\ operands),\ p)$
$= \ \ \textbf{if}\ \text{p-ins-okp}\,(\text{cons}\,(opcode,\ operands),\ p)$
$\quad\ \ \textbf{then}\ \text{p-ins-step}\,(\text{cons}\,(opcode,\ operands),\ p)$
$\quad\ \ \textbf{else}\ \text{p-halt}\,(p,\ \text{x-y-error-msg}\,(\texttt{'p},\ opcode))\ \textbf{endif}$

EVENT: Disable p-step1.

THEOREM: p-opener
$(\text{p}\,(s,\ \mathtt{0}) = s)$
$\land\ \ (\text{p}\,(\text{p-state}\,(pc,\ ctrl,\ temp,\ prog,\ data,\ max\text{-}ctrl,\ max\text{-}temp,\ word\text{-}size,\ psw),$
$\qquad 1 + n)$
$\qquad = \ \ \text{p}\,(\text{p-step}\,(\text{p-state}\,(pc,$

$ctrl$,
$temp$,
$prog$,
$data$,
$max\text{-}ctrl$,
$max\text{-}temp$,
$word\text{-}size$,
psw)),
n))

EVENT: Disable p.

DEFINITION:
at-least-morep ($base$, $delta$, $value$) = ($value \not< (base + delta)$)

THEOREM: at-least-morep-normalize
(at-least-morep (1 + $base$, $delta$, $value$)
= at-least-morep ($base$, 1 + $delta$, $value$))
∧ (at-least-morep ($base$, 1 + $delta$, 1 + $value$)
= at-least-morep ($base$, $delta$, $value$))

THEOREM: at-least-morep-linear
(at-least-morep ($base$, $d1$, $value$) ∧ ($d1 \not< d2$))
→ at-least-morep ($base$, $d2$, $value$)

THEOREM: lessp-as-at-least-morep
at-least-morep ($base$, $delta$, $value$)
→ ((($value < x$) = (¬ at-least-morep (x, `0`, $value$)))
∧ (($x < value$) = at-least-morep (x, `1`, $value$)))

EVENT: Disable at-least-morep.

DEFINITION:
nat-to-bv (nat, $size$)
= **if** $size \simeq$ `0` **then nil**
elseif $nat <$ exp (`2`, $size - 1$) **then** cons (`0`, nat-to-bv (nat, $size - 1$))
else cons (`1`, nat-to-bv ($nat -$ exp (`2`, $size - 1$), $size - 1$)) **endif**

DEFINITION:
nat-to-bv-state ($state$, $size$)
= **if** listp ($state$)
then cons (nat-to-bv (car ($state$), $size$), nat-to-bv-state (cdr ($state$), $size$))
else nil endif

```
;; a more elegant way to write this program would be to use the
;; bit-vector of all 0's initially, then xor all the elements of
;; the array.  But we don't want a pointer to memory to
;; ever have an improper value, so we write things this way
```

DEFINITION:
XOR-BVS-PROGRAM
=
```
'(xor-bvs
   (vecs-addr numvecs)
   nil
   (push-local vecs-addr)
   (fetch)
   (push-local numvecs)
   (sub1-nat)
   (pop-local numvecs)
   (dl loop nil (push-local numvecs))
   (test-nat-and-jump zero done)
   (push-local numvecs)
   (sub1-nat)
   (pop-local numvecs)
   (push-local vecs-addr)
   (push-constant (nat 1))
   (add-addr)
   (set-local vecs-addr)
   (fetch)
   (xor-bitv)
   (jump loop)
   (dl done nil (ret)))
```

DEFINITION:
bit-vectors-piton (*array*, *size*)
= **if** listp (*array*)
then listp (car (*array*))
$\land$ (caar (*array*) = `'bitv`)
$\land$ bit-vectorp (cadar (*array*), *size*)
$\land$ (cddar (*array*) = **nil**)
$\land$ bit-vectors-piton (cdr (*array*), *size*)
else *array* = **nil endif**

DEFINITION: array (*name*, *segment*) = cdr (assoc (*name*, *segment*))

```
;; vecs is name of state
```

DEFINITION:

xor-bvs-input-conditionp ($p0$)
= ((car (top (p-temp-stk ($p0$))) = `'nat`)
  ∧ (car (top (cdr (p-temp-stk ($p0$)))) = `'addr`)
  ∧ (cdadr (top (cdr (p-temp-stk ($p0$)))) = `0`)
  ∧ listp (cadr (top (cdr (p-temp-stk ($p0$)))))
  ∧ (cddr (top (p-temp-stk ($p0$))) = **nil**)
  ∧ (cddr (top (cdr (p-temp-stk ($p0$)))) = **nil**)
  ∧ definedp (caadr (top (cdr (p-temp-stk ($p0$)))), p-data-segment ($p0$))
  ∧ bit-vectors-piton (array (caadr (top (cdr (p-temp-stk ($p0$)))),
                                          p-data-segment ($p0$)),
                              p-word-size ($p0$))
  ∧ (cadr (top (p-temp-stk ($p0$)))
     = length (array (caadr (top (cdr (p-temp-stk ($p0$)))),
                              p-data-segment ($p0$))))
  ∧ at-least-morep (p-ctrl-stk-size (p-ctrl-stk ($p0$)),
                   `4`,
                   p-max-ctrl-stk-size ($p0$))
  ∧ at-least-morep (length (p-temp-stk ($p0$)),
                   `2`,
                   p-max-temp-stk-size ($p0$))
  ∧ (untag (top (p-temp-stk ($p0$))) $\not\simeq$ `0`)
  ∧ (untag (top (p-temp-stk ($p0$))) < exp (`2`, p-word-size ($p0$)))
  ∧ listp (p-ctrl-stk ($p0$)))

```
; time to run loop
```

DEFINITION:
xor-bvs-clock-loop (*numvecs*)
= **if** *numvecs* $\simeq$ `0` **then** `3`
  else `12` + xor-bvs-clock-loop (*numvecs* − 1) **endif**

```
; time to run xor-bvs, including call and ret
```

DEFINITION:
xor-bvs-clock (*numvecs*) = (`6` + xor-bvs-clock-loop (*numvecs* − 1))

DEFINITION:
xor-bvs-array (*current*, *array*, n, *array-size*)
= **if** $n \simeq$ `0` **then** *current*
  else xor-bvs-array (xor-bitv (*current*,
                                   untag (get (*array-size* − n, *array*))),
                         array,
                         $n - 1$,
                         array-size) **endif**

THEOREM: lessp-1-exp
$(1 < \exp(a,\ b)) = ((1 < a) \wedge (b \not\simeq 0))$

THEOREM: bit-vectors-piton-means
$(\text{bit-vectors-piton}(state,\ size) \wedge (p < \text{length}(state)))$
$\rightarrow\ ((\text{car}(\text{get}(p,\ state)) = \texttt{'bitv})$
$\quad \wedge\ \text{listp}(\text{get}(p,\ state))$
$\quad \wedge\ \text{bit-vectorp}(\text{cadr}(\text{get}(p,\ state)),\ size)$
$\quad \wedge\ (\text{cddr}(\text{get}(p,\ state)) = \textbf{nil}))$

DEFINITION:
xor-bvs-loop-correctness-general-induct (i, *current*, n, s, *data-segment*)
= **if** $i \simeq 0$ **then t**
else xor-bvs-loop-correctness-general-induct ($i - 1$,
xor-bitv (*current*,
cadr (get ($n - i$,
array (s,
data-segment)))),
n,
s,
data-segment) **endif**

```
;; in Piton library
;(prove-lemma bit-vectorp-xor-bitv (rewrite)
;      (implies
;       (and
;        (bit-vectorp x size)
;        (bit-vectorp y size))
;       (bit-vectorp (xor-bitv x y) size)))
```

EVENT: Enable bit-vectorp-xor-bitv.

THEOREM: xor-bvs-loop-correctness-general
$((\text{length}(\text{array}(s,\ data\text{-}segment)) < \exp(2,\ word\text{-}size))$
$\wedge\ (word\text{-}size \not\simeq 0)$
$\wedge\ \text{listp}(ctrl\text{-}stk)$
$\wedge\ \text{bit-vectors-piton}(\text{array}(s,\ data\text{-}segment),\ word\text{-}size)$
$\wedge\ \text{at-least-morep}(\text{length}(temp\text{-}stk),\ 3,\ max\text{-}temp\text{-}stk\text{-}size)$
$\wedge\ (\text{definition}(\texttt{'xor-bvs},\ prog\text{-}segment) = \text{XOR-BVS-PROGRAM})$
$\wedge\ \text{definedp}(s,\ data\text{-}segment)$
$\wedge\ (i \in \mathbf{N})$
$\wedge\ (i < n)$
$\wedge\ \text{bit-vectorp}(current,\ word\text{-}size)$

$\wedge$ (n = length (array (s, *data-segment*))))
$\rightarrow$ (p (p-state (`'(pc (xor-bvs . 5))`,
cons (list (list (cons (`'vecs-addr`,
list (`'addr`, cons (s, $(n - i) - 1$))),
cons (`'numvecs`, list (`'nat`, i))),
ret-pc),
ctrl-stk),
cons (list (`'bitv`, *current*), *temp-stk*),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
xor-bvs-clock-loop (i))
= p-state (*ret-pc*,
ctrl-stk,
cons (list (`'bitv`,
xor-bvs-array (*current*,
array (s, *data-segment*),
i,
n)),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

THEOREM: difference-x-sub1-x-better
$(x - (x - 1))$
= **if** `0` $< x$ **then** `1`
else `0` **endif**

THEOREM: xor-bvs-loop-correctness
((length (array (s, *data-segment*)) < exp (`2`, *word-size*))
$\wedge$ (*word-size* $\not\simeq$ `0`)
$\wedge$ listp (*ctrl-stk*)
$\wedge$ bit-vectors-piton (array (s, *data-segment*), *word-size*)
$\wedge$ at-least-morep (length (*temp-stk*), `3`, *max-temp-stk-size*)
$\wedge$ (definition (`'xor-bvs`, *prog-segment*) = XOR-BVS-PROGRAM)
$\wedge$ definedp (s, *data-segment*)
$\wedge$ (`0` $< n$)

$\wedge$ bit-vectorp (*current*, *word-size*)
$\wedge$ (n = length (array (s, *data-segment*))))
$\rightarrow$ (p (p-state (`'(pc (xor-bvs . 5))`,
cons (list (list (cons (`'vecs-addr`, list (`'addr`, cons (s, `0`))),
cons (`'numvecs`, list (`'nat`, $n - 1$))),
ret-pc),
ctrl-stk),
cons (list (`'bitv`, *current*), *temp-stk*),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
xor-bvs-clock-loop ($n - 1$))
= p-state (*ret-pc*,
ctrl-stk,
cons (list (`'bitv`,
xor-bvs-array (*current*,
array (s, *data-segment*),
$n - 1$,
n)),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

THEOREM: exp-0
($x \simeq$ `0`)
$\rightarrow$ ((exp (x, y)
= **if** $y \simeq$ `0` **then** `1`
else `0` **endif**)
$\wedge$ (exp (y, x) = `1`))

THEOREM: bit-vectors-piton-means-more
(listp (x) $\wedge$ bit-vectors-piton (x, *size*))
$\rightarrow$ (list (`'bitv`, cadar (x)) = car (x))

```
;; xor-bvs of an array of at least one bit vector
```

DEFINITION:
xor-bvs (*array*)

$=$ **if** listp ($array$) **then** xor-bitv (car ($array$), xor-bvs (cdr ($array$)))
else nil endif

DEFINITION:
untag-array ($array$)
$=$ **if** listp ($array$) **then** cons (untag (car ($array$)), untag-array (cdr ($array$)))
else nil endif

THEOREM: bit-vectorp-get
bit-vectors-piton ($array$, $size$)
$\rightarrow$ (bit-vectorp (untag (get (n, $array$)), $size$) $=$ ($n <$ length ($array$)))

```
;(prove-lemma difference-sub1-arg2 (rewrite)
;      (equal
;       (difference a (sub1 n))
;       (if (zerop n) (fix a)
; (if (lessp a n) 0 (add1 (difference a n))))))
```

EVENT: Enable difference-sub1-arg2.

THEOREM: xor-bitv-commutative
(length (a) $=$ length (b)) $\rightarrow$ (xor-bitv (a, b) $=$ xor-bitv (b, a))

THEOREM: xor-bitv-commutative2
(length (a) $=$ length (b))
$\rightarrow$ (xor-bitv (a, xor-bitv (b, c)) $=$ xor-bitv (b, xor-bitv (a, c)))

THEOREM: xor-bitv-associative
(length (a) $=$ length (b))
$\rightarrow$ (xor-bitv (xor-bitv (a, b), c) $=$ xor-bitv (a, xor-bitv (b, c)))

THEOREM: length-from-bit-vectorp
bit-vectorp (x, s) $\rightarrow$ (length (x) $=$ fix (s))

THEOREM: length-xor-bitv
length (xor-bitv (a, b)) $=$ length (a)

THEOREM: length-cadr-get-bit-vectors-piton
(bit-vectors-piton (x, l) $\wedge$ ($i <$ length (x)))
$\rightarrow$ (length (cadr (get (i, x))) $=$ fix (l))

THEOREM: equal-xor-bitv-x-x
(bit-vectorp (b, length (a)) $\wedge$ bit-vectorp (c, length (a)))
$\rightarrow$ ((xor-bitv (a, b) $=$ xor-bitv (a, c)) $=$ (b $=$ c))

DEFINITION:
bit-vectorp-induct ($size$, a, b)
$=$ **if** $size \simeq$ `0` **then t**
else bit-vectorp-induct ($size - 1$, cdr (a), cdr (b)) **endif**

THEOREM: bit-vectorp-xor-bitv2
bit-vectorp (xor-bitv (a, b), $size$) = (length (a) = fix ($size$))

DEFINITION:
bit-vectorsp (bvs, $size$)
$=$ **if** listp (bvs)
then bit-vectorp (car (bvs), $size$) $\wedge$ bit-vectorsp (cdr (bvs), $size$)
else bvs = **nil endif**

THEOREM: length-xor-bvs
bit-vectorsp (bvs, length (car (bvs)))
$\rightarrow$ (length (xor-bvs (bvs)) = length (car (bvs)))

THEOREM: bit-vectorsp-untag
bit-vectors-piton (x, s) $\rightarrow$ bit-vectorsp (untag-array (x), s)

THEOREM: bit-vectorsp-cdr-untag
bit-vectors-piton (cdr (x), s) $\rightarrow$ bit-vectorsp (cdr (untag-array (x)), s)

```
;; actually part of npiton-defs, but not supporter of p
;;
;(DEFN NTHCDR
;      (N L)
;      (IF (ZEROP N)
;          L
;          (NTHCDR (SUB1 N) (CDR L))))
```

EVENT: Enable nthcdr.

THEOREM: bit-vectorsp-nthcdr
(bit-vectorsp (x, s) $\wedge$ ($n <$ length (x))) $\rightarrow$ bit-vectorsp (nthcdr (n, x), s)

THEOREM: bit-vectorp-xor-bvs
(bit-vectorsp (x, $size$) $\wedge$ listp (x)) $\rightarrow$ bit-vectorp (xor-bvs (x), $size$)

THEOREM: length-untag-array
length (untag-array (x)) = length (x)

```
;(prove-lemma listp-nthcdr (rewrite)
;      (equal
;       (listp (nthcdr n x))
;       (lessp n (length x)))
;      ((enable nthcdr)))
```

EVENT: Enable listp-nthcdr.

THEOREM: nthcdr-open
$(n < \text{length}(x)) \rightarrow (\text{nthcdr}(n, x) = \text{cons}(\text{get}(n, x), \text{nthcdr}(1 + n, x)))$

THEOREM: get-untag-array
$(n < \text{length}(x)) \rightarrow (\text{get}(n, \text{untag-array}(x)) = \text{cadr}(\text{get}(n, x)))$

THEOREM: equal-xor-bitv-x-x-special
$(\text{bit-vectorp}(b, \text{length}(a)) \wedge \text{bit-vectorp}(\text{xor-bitv}(z, c), \text{length}(a)))$
$\rightarrow \quad ((\text{xor-bitv}(a, b) = \text{xor-bitv}(z, \text{xor-bitv}(a, c)))$
$\quad = \quad (b = \text{xor-bitv}(z, c)))$

DEFINITION:
fix-bit (b)
$=$ **if** $b = \mathtt{0}$ **then** $\mathtt{0}$
else $\mathtt{1}$ **endif**

THEOREM: xor-bitv-0
$(\text{xor-bit}(x, \mathtt{0}) = \text{fix-bit}(x)) \wedge (\text{xor-bit}(\mathtt{0}, x) = \text{fix-bit}(x))$

THEOREM: xor-bitv-nlistp
$(\neg \text{listp}(c)) \rightarrow (\text{xor-bitv}(a, \text{xor-bitv}(b, c)) = \text{xor-bitv}(a, b))$

THEOREM: xor-bitv-nlistp2
$(\text{bit-vectorp}(a, b) \wedge (\neg \text{listp}(c))) \rightarrow (\text{xor-bitv}(a, c) = a)$

THEOREM: xor-bvs-array-rewrite
$(\text{bit-vectors-piton}(\mathit{array}, \text{length}(\mathit{current}))$
$\wedge \quad \text{bit-vectorp}(\mathit{current}, \text{length}(\mathit{current}))$
$\wedge \quad (n < \text{length}(\mathit{array}))$
$\wedge \quad (\text{length}(\mathit{array}) = \mathit{as}))$
$\rightarrow \quad (\text{xor-bvs-array}(\mathit{current}, \mathit{array}, n, \mathit{as})$
$\quad = \quad \text{xor-bitv}(\mathit{current}, \text{xor-bvs}(\text{nthcdr}(\mathit{as} - n, \text{untag-array}(\mathit{array})))))$

```
;; in npiton-defs but not a supporter of p
;(PROVE-LEMMA EQUAL-LENGTH-0
;              (REWRITE)
;              (EQUAL (EQUAL (LENGTH X) '0)
;                     (NLISTP X)))
```

EVENT: Enable equal-length-0.

THEOREM: correctness-of-xor-bvs-helper
$((\mathit{p0} = \text{p-state}(\mathit{pc},$

ctrl-stk,
append (list (tag (`'nat`, *numvecs*), tag (`'addr`, cons (*state*, `0`))),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

$\land$ (p-current-instruction (*p0*) = `'(call xor-bvs)`)
$\land$ (definition (`'xor-bvs`, *prog-segment*) = XOR-BVS-PROGRAM)
$\land$ xor-bvs-input-conditionp (*p0*))
$\rightarrow$ (p (*p0*, xor-bvs-clock (*numvecs*))
= p-state (add1-addr (*pc*),
ctrl-stk,
cons (list (`'bitv`,
xor-bvs-array (untag (car (array (*state*,
data-segment))),
array (*state*, *data-segment*),
numvecs $-$ 1,
numvecs)),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

THEOREM: length-cadar-bvs
(bit-vectors-piton (x, s) $\land$ listp (x)) $\rightarrow$ (length (cadar (x)) = fix (s))

THEOREM: bit-vectorp-from-bit-vectors-piton
bit-vectors-piton (x, s)
$\rightarrow$ ((bit-vectorp (cadar (x), s) = listp (x))
$\land$ (bit-vectorp (cadr (get (n, x)), s) = (n < length (x))))

THEOREM: nthcdr-1
nthcdr (`1`, a) = cdr (a)

THEOREM: listp-untag-array
listp (untag-array (x)) = listp (x)

THEOREM: xor-bvs-input-conditionp-means-xor-bvs-hack
(xor-bvs-input-conditionp (p-state (*pc*,

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{ctrl\text{-}stk}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{cons}\,(\text{list}\,(\texttt{'nat},\ \mathit{numvecs}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{cons}\,(\text{list}\,(\texttt{'addr},\ \text{cons}\,(\mathit{state},\ \texttt{0})), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{temp\text{-}stk})), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{prog\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{data\text{-}segment}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{max\text{-}ctrl\text{-}stk\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{max\text{-}temp\text{-}stk\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{word\text{-}size}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{'run})) \\
\land \quad (\texttt{0} < \mathit{word\text{-}size})) \\
\rightarrow \quad (\text{xor-bvs-array}\,(\text{untag}\,(\text{car}\,(\text{array}\,(\mathit{state},\ \mathit{data\text{-}segment}))), \\
\qquad\qquad\qquad\qquad\quad \text{array}\,(\mathit{state},\ \mathit{data\text{-}segment}), \\
\qquad\qquad\qquad\qquad\quad \mathit{numvecs} - 1, \\
\qquad\qquad\qquad\qquad\quad \mathit{numvecs}) \\
\qquad = \quad \text{xor-bvs}\,(\text{untag-array}\,(\text{array}\,(\mathit{state},\ \mathit{data\text{-}segment}))))
\end{array}
$$

THEOREM: correctness-of-xor-bvs

$$
\begin{array}{l}
((\mathit{p0} = \text{p-state}\,(\mathit{pc}, \\
\qquad\qquad\qquad \mathit{ctrl\text{-}stk}, \\
\qquad\qquad\qquad \text{cons}\,(n,\ \text{cons}\,(s,\ \mathit{temp\text{-}stk})), \\
\qquad\qquad\qquad \mathit{prog\text{-}segment}, \\
\qquad\qquad\qquad \mathit{data\text{-}segment}, \\
\qquad\qquad\qquad \mathit{max\text{-}ctrl\text{-}stk\text{-}size}, \\
\qquad\qquad\qquad \mathit{max\text{-}temp\text{-}stk\text{-}size}, \\
\qquad\qquad\qquad \mathit{word\text{-}size}, \\
\qquad\qquad\qquad \texttt{'run})) \\
\land \quad (\texttt{0} < \mathit{word\text{-}size}) \\
\land \quad (\text{p-current-instruction}\,(\mathit{p0}) = \texttt{'(call xor-bvs)}) \\
\land \quad (\text{definition}\,(\texttt{'xor-bvs},\ \mathit{prog\text{-}segment}) = \textsc{xor-bvs-program}) \\
\land \quad \text{xor-bvs-input-conditionp}\,(\mathit{p0})) \\
\rightarrow \quad (\text{p}\,(\text{p-state}\,(\mathit{pc}, \\
\qquad\qquad\qquad\quad \mathit{ctrl\text{-}stk}, \\
\qquad\qquad\qquad\quad \text{cons}\,(n,\ \text{cons}\,(s,\ \mathit{temp\text{-}stk})), \\
\qquad\qquad\qquad\quad \mathit{prog\text{-}segment}, \\
\qquad\qquad\qquad\quad \mathit{data\text{-}segment}, \\
\qquad\qquad\qquad\quad \mathit{max\text{-}ctrl\text{-}stk\text{-}size}, \\
\qquad\qquad\qquad\quad \mathit{max\text{-}temp\text{-}stk\text{-}size}, \\
\qquad\qquad\qquad\quad \mathit{word\text{-}size}, \\
\qquad\qquad\qquad\quad \texttt{'run}), \\
\qquad\quad \text{xor-bvs-clock}\,(\text{cadr}\,(n))) \\
\qquad = \quad \text{p-state}\,(\text{add1-addr}\,(\mathit{pc}), \\
\qquad\qquad\qquad\qquad \mathit{ctrl\text{-}stk}, \\
\qquad\qquad\qquad\qquad \text{cons}\,(\text{list}\,(\texttt{'bitv},
\end{array}
$$

xor-bvs (untag-array (array (caadr (s),
data-segment)))),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

DEFINITION:
EXAMPLE-XOR-BVS-P-STATE
= p-state (`'(pc (main . 0))`,
`'((nil (pc (main . 0))))`,
nil,
list (`'(main nil nil`
`(push-constant (addr (arr . 0)))`
`(push-constant (nat 3))`
`(call xor-bvs)`
`(ret))`,
XOR-BVS-PROGRAM),
`'((arr`
`(bitv (0 1 0 1 1 0 0 1))`
`(bitv (0 0 0 0 0 0 0 1))`
`(bitv (0 1 1 0 1 0 0 1)))`),
`10`,
`8`,
`8`,
`'run`)

```
;;;; push-1-vector
;; Piton currently does not provide any mechanism for creating a
;; bit vector except as an operation on other bit vectors.  Until
;; this apparent flaw is fixed, we'll write our program as a
;; function of the word size.
```

DEFINITION:
one-bit-vector (*wordsize*)
= **if** *wordsize* $<$ `2` **then** list (`1`)
else cons (`0`, one-bit-vector (*wordsize* $-$ 1)) **endif**

DEFINITION:
push-1-vector-program (*wordsize*)
= list (`'push-1-vector`,

nil,
nil,
list (`'push-constant`, list (`'bitv`, one-bit-vector (*wordsize*))),
list (`'ret`))

DEFINITION:
EXAMPLE-PUSH-1-VECTOR-STATE
= p-state (`'(pc (main . 0))`,
`'((nil (pc (main . 0))))`,
nil,
list (`'(main nil nil (call push-1-vector) (ret))`,
push-1-vector-program (8)),
nil,
`10`,
`8`,
`8`,
`'run`)

DEFINITION:
push-1-vector-input-conditionp (*p0*)
= ((p-max-ctrl-stk-size (*p0*)
$\not<$ (2 + p-ctrl-stk-size (p-ctrl-stk (*p0*))))
$\wedge$ (p-max-temp-stk-size (*p0*) $\not<$ (1 + length (p-temp-stk (*p0*))))
$\wedge$ listp (p-ctrl-stk (*p0*)))

```
;(prove-lemma length-append (rewrite)
;      (equal
;       (length (append a b))
;       (plus (length a) (length b))))
```

EVENT: Enable length-append.

THEOREM: equal-assoc-cons
(assoc (k, a) = cons (x, y))
$\rightarrow$ ((car (assoc (k, a)) = x) $\wedge$ (cdr (assoc (k, a)) = y))

THEOREM: correctness-of-push-1-vector
((*p0* = p-state (*pc*,
ctrl-stk,
temp-stk,
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,

$word\text{-}size$,
'run))
∧ (p-current-instruction ($p0$) = '(call push-1-vector))
∧ (definition ('push-1-vector, $prog\text{-}segment$)
= push-1-vector-program ($word\text{-}size$))
∧ push-1-vector-input-conditionp ($p0$))
→ (p ($p0$, 3)
= p-state (add1-addr (pc),
$ctrl\text{-}stk$,
cons (list ('bitv, one-bit-vector ($word\text{-}size$)), $temp\text{-}stk$),
$prog\text{-}segment$,
$data\text{-}segment$,
$max\text{-}ctrl\text{-}stk\text{-}size$,
$max\text{-}temp\text{-}stk\text{-}size$,
$word\text{-}size$,
'run))

```
;;;; nat-to-bv
```

DEFINITION:
NAT-TO-BV-PROGRAM

```
=  '(nat-to-bv
     (value)
     ((current-bit (nat 0)) (temp (nat 0)))
     (call push-1-vector)
     (pop-local current-bit)
     (call push-1-vector)
     (rsh-bitv)
     (dl loop nil (push-local value))
     (test-nat-and-jump zero done)
     (push-local value)
     (div2-nat)
     (pop-local temp)
     (pop-local value)
     (push-local temp)
     (test-nat-and-jump zero lab)
     (push-local current-bit)
     (xor-bitv)
     (dl lab nil (push-local current-bit))
     (lsh-bitv)
     (pop-local current-bit)
     (jump loop)
     (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-NAT-TO-BV-STATE
= p-state ('(pc (main . 0)),
'((nil (pc (main . 0)))),
nil,
list ('(main nil nil
(push-constant (nat 86))
(call nat-to-bv)
(ret)),
push-1-vector-program (8),
NAT-TO-BV-PROGRAM),
nil,
10,
8,
8,
'run)

DEFINITION:
zero-bit-vector (*size*)
= **if** *size* $\simeq$ 0 **then nil**
else cons (0, zero-bit-vector (*size* $-$ 1)) **endif**

DEFINITION:
nat-to-bv2-helper (*value*, *current-bit*, *bit-vector*)
= **if** *value* $\simeq$ 0 **then** *bit-vector*
else nat-to-bv2-helper (*value* $\div$ 2,
append (cdr (*current-bit*), list (0)),
if (*value* **mod** 2) = 1
then xor-bitv (*current-bit*, *bit-vector*)
else *bit-vector* **endif**) **endif**

DEFINITION:
nat-to-bv2 (*value*, *size*)
= nat-to-bv2-helper (*value*, one-bit-vector (*size*), zero-bit-vector (*size*))

DEFINITION:
nat-to-bv-loop-clock (*value*)
= **if** *value* $\simeq$ 0 **then** 3
else if (*value* **mod** 2) = 0 **then** 12
else 14 **endif**
$+$ nat-to-bv-loop-clock (*value* $\div$ 2) **endif**

DEFINITION:
correctness-of-nat-to-bv-general-induct (*value*, *cb*, *temp*, *bv*)
= **if** *value* $\simeq$ 0 **then t**

else correctness-of-nat-to-bv-general-induct (*value* $\div$ 2,
append (cdr (*cb*),
list (0)),
list ('nat,
value **mod** 2),
if (*value* **mod** 2)
= 0
then *bv*
else xor-bitv (*bv*,
cb) **endif**) **endif**

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

THEOREM: lessp-exp-simple
$((x < y) \wedge (z \not\simeq 0)) \rightarrow (x < \exp(y,\ z))$

THEOREM: lessp-1
$(x < 1) = (x \simeq 0)$

THEOREM: lessp-remainder-x-exp-x
$((x \textbf{ mod } y) < \exp(y,\ z))$
$= \quad (((z \simeq 0) \wedge ((x \textbf{ mod } y) = 0)) \vee ((z \not\simeq 0) \wedge (y \not\simeq 0)))$

THEOREM: bit-vectorp-append
bit-vectorp (x, *x-size*)
$\rightarrow$ (bit-vectorp (append (x, y), *size*)
= (bit-vectorp (y, *size* $-$ *x-size*) $\wedge$ (*size* $\not<$ *x-size*)))

THEOREM: correctness-of-nat-to-bv-general
(listp (*ctrl-stk*)
$\wedge$ at-least-morep (length (*temp-stk*), 3, *max-temp-stk-size*)
$\wedge$ (definition ('nat-to-bv, *prog-segment*) = NAT-TO-BV-PROGRAM)
$\wedge$ (*value* $\in \mathbf{N}$)
$\wedge$ (0 < *word-size*)
$\wedge$ (*value* < exp (2, *word-size*))
$\wedge$ bit-vectorp (*bv*, *word-size*)
$\wedge$ bit-vectorp (*cb*, *word-size*))
$\rightarrow$ (p (p-state ('(pc (nat-to-bv . 4)),
cons (list (list (cons ('value, list ('nat, *value*)),
cons ('current-bit, list ('bitv, *cb*)),
cons ('temp, *temp*)),
ret-pc),
ctrl-stk),
cons (list ('bitv, *bv*), *temp-stk*),

$$
\begin{array}{l}
\quad\quad\quad\quad \textit{prog-segment}, \\
\quad\quad\quad\quad \textit{data-segment}, \\
\quad\quad\quad\quad \textit{max-ctrl-stk-size}, \\
\quad\quad\quad\quad \textit{max-temp-stk-size}, \\
\quad\quad\quad\quad \textit{word-size}, \\
\quad\quad\quad\quad \texttt{'run}), \\
\quad \text{nat-to-bv-loop-clock}\,(\textit{value})) \\
= \quad \text{p-state}\,(\textit{ret-pc}, \\
\quad\quad\quad\quad \textit{ctrl-stk}, \\
\quad\quad\quad\quad \text{cons}\,(\text{list}\,(\texttt{'bitv}, \text{nat-to-bv2-helper}\,(\textit{value},\ \textit{cb},\ \textit{bv})), \\
\quad\quad\quad\quad\quad\quad \textit{temp-stk}), \\
\quad\quad\quad\quad \textit{prog-segment}, \\
\quad\quad\quad\quad \textit{data-segment}, \\
\quad\quad\quad\quad \textit{max-ctrl-stk-size}, \\
\quad\quad\quad\quad \textit{max-temp-stk-size}, \\
\quad\quad\quad\quad \textit{word-size}, \\
\quad\quad\quad\quad \texttt{'run}))
\end{array}
$$

DEFINITION:
nat-to-bv-clock $(\textit{value}) = (\texttt{9} + \text{nat-to-bv-loop-clock}\,(\textit{value}))$

DEFINITION:
nat-to-bv-input-conditionp $(\textit{p0})$

$$
\begin{array}{ll}
= & ((\text{cadr}\,(\text{top}\,(\text{p-temp-stk}\,(\textit{p0}))) < \exp\,(\texttt{2}, \text{p-word-size}\,(\textit{p0}))) \\
& \wedge\ (\text{cadr}\,(\text{top}\,(\text{p-temp-stk}\,(\textit{p0}))) \in \mathbf{N}) \\
& \wedge\ \text{at-least-morep}\,(\text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(\textit{p0})), \\
& \quad\quad\quad\quad \texttt{7}, \\
& \quad\quad\quad\quad \text{p-max-ctrl-stk-size}\,(\textit{p0})) \\
& \wedge\ (\texttt{0} < \text{p-word-size}\,(\textit{p0})) \\
& \wedge\ \text{at-least-morep}\,(\text{length}\,(\text{p-temp-stk}\,(\textit{p0})), \\
& \quad\quad\quad\quad \texttt{2}, \\
& \quad\quad\quad\quad \text{p-max-temp-stk-size}\,(\textit{p0})) \\
& \wedge\ \text{listp}\,(\text{p-ctrl-stk}\,(\textit{p0})))
\end{array}
$$

THEOREM: bit-vectorp-one-bit-vector
bit-vectorp (one-bit-vector $(s),\ s) = (\texttt{0} < s)$

THEOREM: bit-vectorp-zero-bit-vector
bit-vectorp (zero-bit-vector $(s),\ s)$

THEOREM: all-but-last-one-bit-vector
all-but-last (one-bit-vector $(s)) =$ zero-bit-vector $(s - 1)$

THEOREM: correctness-of-nat-to-bv-helper
$((\textit{p0} = \text{p-state}\,(\textit{pc},$

$ctrl\text{-}stk$,
cons (list (`'nat`, $value$), $temp\text{-}stk$),
$prog\text{-}segment$,
$data\text{-}segment$,
$max\text{-}ctrl\text{-}stk\text{-}size$,
$max\text{-}temp\text{-}stk\text{-}size$,
$word\text{-}size$,
`'run`))
$\wedge$ (p-current-instruction ($p0$) = `'(call nat-to-bv)`)
$\wedge$ (definition (`'nat-to-bv`, $prog\text{-}segment$) = NAT-TO-BV-PROGRAM)
$\wedge$ (definition (`'push-1-vector`, $prog\text{-}segment$)
= push-1-vector-program ($word\text{-}size$))
$\wedge$ nat-to-bv-input-conditionp ($p0$))
$\rightarrow$ (p ($p0$, nat-to-bv-clock ($value$))
= p-state (add1-addr (pc),
$ctrl\text{-}stk$,
cons (list (`'bitv`, nat-to-bv2 ($value$, $word\text{-}size$)),
$temp\text{-}stk$),
$prog\text{-}segment$,
$data\text{-}segment$,
$max\text{-}ctrl\text{-}stk\text{-}size$,
$max\text{-}temp\text{-}stk\text{-}size$,
$word\text{-}size$,
`'run`))

DEFINITION:
bv-to-nat (bv)
= **if** listp (bv)
then (fix-bit (car (bv)) $*$ exp (2, length (cdr (bv))))
$+$ bv-to-nat (cdr (bv))
else 0 **endif**

THEOREM: length-nat-to-bv
length (nat-to-bv (nat, $size$)) = fix ($size$)

THEOREM: bv-to-nat-nat-to-bv
bv-to-nat (nat-to-bv (nat, $size$))
= **if** $nat < $ exp (2, $size$) **then** fix (nat)
else exp (2, $size$) $-$ 1 **endif**

EVENT: Enable exp.

THEOREM: nat-to-bv-simple
($x \simeq 0$)
$\rightarrow$ ((nat-to-bv (x, y) = zero-bit-vector (y)) $\wedge$ (nat-to-bv (y, x) = **nil**))

THEOREM: nat-to-bv-bv-to-nat
bit-vectorp $(bv,\ size) \rightarrow (\text{nat-to-bv}(\text{bv-to-nat}(bv),\ size) = bv)$

THEOREM: bv-to-nat-append
bv-to-nat (append $(x,\ y)$)
$= \quad (\text{bv-to-nat}(y) + (\exp(2,\ \text{length}(y)) * \text{bv-to-nat}(x)))$

THEOREM: lessp-plus-hacks
$(((a + (b + (c + ((d * e) + (d * e))))) < (e + e))$
$= \quad ((d \simeq 0) \wedge ((a + (b + c)) < (e + e))))$
$\wedge \quad (((a + (b + (c + (e + ((d * g) + h))))) < g)$
$\quad = \quad ((d \simeq 0) \wedge ((a + (b + (c + (e + h)))) < g)))$

THEOREM: bv-to-nat-xor-bitv
(bit-vectorp $(x,\ size)$
$\wedge$ bit-vectorp $(y,\ size)$
$\wedge$ (and-bitv $(x,\ y)$ = zero-bit-vector $(size)$))
$\rightarrow$ (bv-to-nat (xor-bitv $(x,\ y)$) = (bv-to-nat (x) + bv-to-nat (y)))

THEOREM: lessp-bv-to-nat-exp-2
bit-vectorp $(x,\ size) \rightarrow ((\text{bv-to-nat}(x) < \exp(2,\ size)) = \mathbf{t})$

THEOREM: equal-nat-to-bv
(bit-vectorp $(bv,\ size) \wedge (y < \exp(2,\ size))$)
$\rightarrow \quad (((\text{nat-to-bv}(y,\ size) = bv) = (\text{bv-to-nat}(bv) = \text{fix}(y)))$
$\quad \wedge \quad ((bv = \text{nat-to-bv}(y,\ size)) = (\text{fix}(y) = \text{bv-to-nat}(bv))))$

DEFINITION:
least-bit-higher-than-high-bit $(x,\ y)$
= **if** listp (x)
then (length (x) = length (y))
$\wedge$ **if** car $(y) \neq 0$ **then** all-zero-bitvp (x)
else least-bit-higher-than-high-bit (cdr (x), cdr (y)) **endif**
else $y \simeq$ **nil** **endif**

THEOREM: least-bit-higher-means-and-0
(bit-vectorp $(x,\ size)$
$\wedge$ bit-vectorp $(y,\ size)$
$\wedge$ least-bit-higher-than-high-bit $(x,\ y)$)
$\rightarrow$ ((and-bitv $(x,\ y)$ = zero-bit-vector $(size)$)
$\wedge$ (and-bitv $(y,\ x)$ = zero-bit-vector $(size)$))

THEOREM: all-zero-bitvp-zero-bit-vector
all-zero-bitvp (zero-bit-vector $(size)$)

THEOREM: bv-to-nat-one-bit-vector
bv-to-nat (one-bit-vector $(size)$) = 1

```
;; renamed from firstn to avoid conflict with similar but different
;; definition in piton library
```

DEFINITION:
make-list-from $(n,\ \mathit{list})$
$=$ **if** $n \simeq 0$ **then nil**
else cons (car (list), make-list-from ($n - 1$, cdr (list))) **endif**

THEOREM: length-make-list-from
length (make-list-from (n, list)) $=$ fix (n)

THEOREM: make-list-from-1
make-list-from (1, x) $=$ list (car (x))

THEOREM: listp-make-list-from
listp (make-list-from (n, x)) $=$ ($n \not\simeq 0$)

DEFINITION:
double-cdr-induct (x, z)
$=$ **if** listp (x) **then** double-cdr-induct (cdr (x), cdr (z))
else t endif

THEOREM: all-zero-bitvp-append
all-zero-bitvp (append (x, y)) $=$ (all-zero-bitvp (x) $\wedge$ all-zero-bitvp (y))

THEOREM: least-bit-higher-than-high-bit-append-0s
(all-zero-bitvp (y) $\wedge$ (length (z) $=$ length (append (x, y))))
$\rightarrow$ (least-bit-higher-than-high-bit (append (x, y), z)
$=$ least-bit-higher-than-high-bit (x, make-list-from (length (x), z)))

THEOREM: equal-xor-bitv-x-y-1
(xor-bitv (a, b) $=$ b) $=$ (bit-vectorp (b, length (a)) $\wedge$ all-zero-bitvp (a))

THEOREM: equal-xor-bitv-x-y-2
(xor-bitv (a, b) $=$ a)
$=$ (all-zero-bitvp (make-list-from (length (a), b))
$\wedge$ bit-vectorp (a, length (a)))

THEOREM: least-bit-higher-than-high-bit-simple
all-zero-bitvp (x)
$\rightarrow$ (least-bit-higher-than-high-bit (x, y) $=$ (length (x) $=$ length (y)))

THEOREM: make-list-from-is-all-but-last
(length (x) $=$ ($1 + n$)) $\rightarrow$ (make-list-from (n, x) $=$ all-but-last (x))

THEOREM: least-bit-higher-all-but-last
least-bit-higher-than-high-bit $(a,\ b)$
$\rightarrow$ (least-bit-higher-than-high-bit $(a$, cons $(0$, all-but-last $(b)))$
$=$ listp $(a))$

DEFINITION:
at-most-one-bit-on (bv)
$=$ **if** listp (bv)
then if car $(bv) = 0$ **then** at-most-one-bit-on (cdr (bv))
else all-zero-bitvp (cdr (bv)) **endif**
else t endif

```
;
;(prove-lemma length-all-but-last (rewrite)
;      (equal
;       (length (all-but-last x))
;       (if (listp x) (sub1 (length x)) 0)))
```

EVENT: Enable length-all-but-last.

THEOREM: listp-xor-bitv
listp (xor-bitv $(a,\ b)$) $=$ listp (a)

THEOREM: all-zero-bitvp-means-at-most-one-bit-on
all-zero-bitvp $(x) \rightarrow$ at-most-one-bit-on (x)

THEOREM: at-most-one-bit-on-append
at-most-one-bit-on (append $(a,\ b)$)
$=$ ((all-zero-bitvp $(a) \wedge$ at-most-one-bit-on (b))
$\vee$ (at-most-one-bit-on $(a) \wedge$ all-zero-bitvp (b)))

DEFINITION:
fix-bitv $(list)$
$=$ **if** listp $(list)$
then cons (**if** car $(list) = 0$ **then** 0
else 1 **endif**,
fix-bitv (cdr $(list)$))
else nil endif

THEOREM: xor-bitv-nlistp3
$(\neg$ listp $(x))$
$\rightarrow$ ((xor-bitv $(x,\ y) =$ **nil**) $\wedge$ (xor-bitv $(y,\ x) =$ fix-bitv (y)))

EVENT: Disable xor-bitv-nlistp.

EVENT: Disable xor-bitv-nlistp2.

THEOREM: fix-bitv-all-but-last
$\text{fix-bitv}(\text{all-but-last}(x)) = \text{all-but-last}(\text{fix-bitv}(x))$

THEOREM: all-but-last-xor-bitv
$$\begin{array}{ll} & \text{all-but-last}(\text{xor-bitv}(a,\ b)) \\ = & \textbf{if } \text{length}(b) < \text{length}(a) \ \textbf{ then } \text{xor-bitv}(\text{all-but-last}(a),\ b) \\ & \textbf{else } \text{xor-bitv}(\text{all-but-last}(a),\ \text{all-but-last}(b)) \ \textbf{endif} \end{array}$$

THEOREM: least-bit-higher-cons-xor-bitv-hack
$$\begin{array}{ll} & (\text{least-bit-higher-than-high-bit}(\text{cdr}(x),\ a) \\ \wedge & \text{least-bit-higher-than-high-bit}(\text{cdr}(x),\ b)) \\ \rightarrow & (\text{least-bit-higher-than-high-bit}(x,\ \text{cons}(\mathtt{0},\ \text{xor-bitv}(a,\ b))) \\ & \quad = \quad \text{listp}(x)) \end{array}$$

THEOREM: least-bit-higher-cdr-all-but-last
$$\begin{array}{ll} & \text{least-bit-higher-than-high-bit}(a,\ b) \\ \rightarrow & \text{least-bit-higher-than-high-bit}(\text{cdr}(a),\ \text{all-but-last}(b)) \end{array}$$

THEOREM: least-bit-higher-x-all-but-last-x
$$\begin{array}{ll} & \text{at-most-one-bit-on}(x) \\ \rightarrow & \text{least-bit-higher-than-high-bit}(\text{cdr}(x),\ \text{all-but-last}(x)) \end{array}$$

THEOREM: nat-to-bv-equiv-helper
$$\begin{array}{ll} & (\text{bit-vectorp}(cb,\ \mathit{size}) \\ \wedge & \text{bit-vectorp}(bv,\ \mathit{size}) \\ \wedge & (\mathtt{0} < \mathit{size}) \\ \wedge & ((\text{bv-to-nat}(bv) + (\text{bv-to-nat}(cb) * \mathit{value})) < \exp(\mathtt{2},\ \mathit{size})) \\ \wedge & \text{at-most-one-bit-on}(cb) \\ \wedge & \text{least-bit-higher-than-high-bit}(cb,\ bv)) \\ \rightarrow & (\text{nat-to-bv2-helper}(\mathit{value},\ cb,\ bv) \\ & \quad = \quad \text{nat-to-bv}((\text{bv-to-nat}(cb) * \mathit{value}) + \text{bv-to-nat}(bv),\ \mathit{size})) \end{array}$$

THEOREM: at-most-one-bit-on-one-bit-vector
$\text{at-most-one-bit-on}(\text{one-bit-vector}(\mathit{size}))$

THEOREM: bv-to-nat-all-zero-bitvp
$\text{all-zero-bitvp}(x) \rightarrow (\text{bv-to-nat}(x) = \mathtt{0})$

THEOREM: least-bit-higher-than-high-bit-simple2
$$\begin{array}{ll} & \text{all-zero-bitvp}(x) \\ \rightarrow & (\text{least-bit-higher-than-high-bit}(y,\ x) = (\text{length}(x) = \text{length}(y))) \end{array}$$

THEOREM: length-zero-bit-vector
$\text{length}(\text{zero-bit-vector}(\mathit{size})) = \text{fix}(\mathit{size})$

THEOREM: length-one-bit-vector
length (one-bit-vector (*size*))
= **if** *size* $\simeq$ `0` **then** `1`
else *size* **endif**

THEOREM: nat-to-bv-equivalence
((*value* < exp (`2`, *size*)) ∧ (`0` < *size*))
→ (nat-to-bv2 (*value*, *size*) = nat-to-bv (*value*, *size*))

THEOREM: correctness-of-nat-to-bv
((*p0* = p-state (*pc*,
ctrl-stk,
cons (*v*, *temp-stk*),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))
∧ (car (*v*) = `'nat`)
∧ (cddr (*v*) = **nil**)
∧ (p-current-instruction (*p0*) = `'(call nat-to-bv)`)
∧ (definition (`'nat-to-bv`, *prog-segment*) = NAT-TO-BV-PROGRAM)
∧ (definition (`'push-1-vector`, *prog-segment*)
= push-1-vector-program (*word-size*))
∧ (*num* = cadr (*v*))
∧ nat-to-bv-input-conditionp (*p0*))
→ (p (p-state (*pc*,
ctrl-stk,
cons (*v*, *temp-stk*),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
nat-to-bv-clock (*num*))
= p-state (add1-addr (*pc*),
ctrl-stk,
cons (list (`'bitv`, nat-to-bv (cadr (*v*), *word-size*)),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,

max-temp-stk-size,
word-size,
`'run`))

```
;;;; bv-to-nat
```

DEFINITION:
BV-TO-NAT-PROGRAM
= `'(bv-to-nat`

```
    (bv)
    ((current-bit (nat 0)) (current-2power (nat 0)))
    (push-constant (nat 1))
    (pop-local current-2power)
    (call push-1-vector)
    (pop-local current-bit)
    (push-constant (nat 0))
    (dl loop nil (push-local bv))
    (push-local current-bit)
    (and-bitv)
    (test-bitv-and-jump all-zero lab)
    (push-local current-2power)
    (add-nat)
    (dl lab nil (push-local current-bit))
    (lsh-bitv)
    (set-local current-bit)
    (test-bitv-and-jump all-zero done)
    (push-local current-2power)
    (mult2-nat)
    (pop-local current-2power)
    (jump loop)
    (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-BV-TO-NAT-STATE
= p-state (`'(pc (main . 0))`,
`'((nil (pc (main . 0))))`,
nil,
list (`'(main nil nil`

```
      (push-constant (bitv (1 0 1 1 0 0 0 1)))
      (call bv-to-nat)
      (ret))
```

, push-1-vector-program (8),
BV-TO-NAT-PROGRAM),
nil,

```
10,
8,
8,
'run)
```

DEFINITION:
trailing-zeros-helper ($list$, acc)
$=$ **if** listp ($list$)
then trailing-zeros-helper (cdr ($list$),
if car ($list$) $=$ 0 **then** $1 + acc$
else 0 **endif**)
else fix (acc) **endif**

DEFINITION:
trailing-zeros ($list$) $=$ trailing-zeros-helper ($list$, 0)

THEOREM: trailing-zeros-of-all-zero-bitvp
all-zero-bitvp (bv) $\rightarrow$ (trailing-zeros-helper (bv, n) $=$ (length (bv) $+$ n))

THEOREM: non-zero-means-acc-irrelevant-spec
($\neg$ all-zero-bitvp (bv))
$\rightarrow$ (trailing-zeros-helper (bv, $1 + x$) $=$ trailing-zeros-helper (bv, x))

THEOREM: non-zero-means-acc-irrelevant
(($\neg$ all-zero-bitvp (bv)) $\wedge$ ($x \neq 0$))
$\rightarrow$ (trailing-zeros-helper (bv, x) $=$ trailing-zeros-helper (bv, 0))

THEOREM: trailing-zeros-helper-append
trailing-zeros-helper (append (x, y), acc)
$=$ **if** all-zero-bitvp (y) **then** trailing-zeros-helper (x, acc)
$+$ length (y)
else trailing-zeros-helper (y, acc) **endif**

THEOREM: trailing-zeros-append
trailing-zeros (append (x, y))
$=$ **if** all-zero-bitvp (y) **then** trailing-zeros (x) $+$ length (y)
else trailing-zeros (y) **endif**

THEOREM: lessp-trailing-zeros-helper
($n \not< (acc +$ length (x)))
$\rightarrow$ (($n <$ trailing-zeros-helper (x, acc)) $=$ **f**)

DEFINITION:
last ($list$)
$=$ **if** listp ($list$)
then if listp (cdr ($list$)) **then** last (cdr ($list$))
else car ($list$) **endif**
else 0 **endif**

THEOREM: equal-trailing-zeros-helper-0
(trailing-zeros-helper $(x,\ acc) = 0)$
$=\quad (((acc \simeq 0) \wedge (x \simeq \textbf{nil})) \vee (\text{last}\,(x) \neq 0))$

THEOREM: lessp-length-cdr-trailing
$((\text{length}\,(\text{cdr}\,(x)) < \text{trailing-zeros-helper}\,(x,\ acc)) \wedge \text{listp}\,(x))$
$\rightarrow\quad (\text{all-zero-bitvp}\,(x) \wedge \text{all-zero-bitvp}\,(\text{cdr}\,(x)))$

THEOREM: not-all-zero-bitvp-cdr-means
$(\neg\ \text{all-zero-bitvp}\,(\text{cdr}\,(x)))$
$\rightarrow\quad (\text{trailing-zeros-helper}\,(x,\ acc) = \text{trailing-zeros-helper}\,(\text{cdr}\,(x),\ 0))$

DEFINITION:
bv-to-nat2-helper (*bv*, *cb*, *current-2power*)
= **if** all-zero-bitvp (*cb*) **then** 0
  else if all-zero-bitvp (and-bitv (*bv*, *cb*)) **then** 0
    else *current-2power* **endif**
    + bv-to-nat2-helper (*bv*,
        append (cdr (*cb*), list (0)),
        2 $*$ *current-2power*) **endif**

DEFINITION:
bv-to-nat2 (*bv*) = bv-to-nat2-helper (*bv*, one-bit-vector (length (*bv*)), 1)

THEOREM: lessp-length-trailing-zeros-hack
$(acc \simeq 0) \rightarrow (\text{length}\,(x) \not< \text{trailing-zeros-helper}\,(x,\ acc))$

DEFINITION:
bv-to-nat-loop-clock (*cb*, *bv*)
= **if** all-zero-bitvp (cdr (*cb*))
  then if (car (*cb*) = 0) $\vee$ (car (*bv*) = 0) **then** 9
    else 11 **endif**
  else if all-zero-bitvp (and-bitv (*cb*, *bv*)) **then** 12
    else 14 **endif**
    + bv-to-nat-loop-clock (append (cdr (*cb*), '(0)), *bv*) **endif**

DEFINITION:
bv-to-nat-induct (*value*, *bv*, *cb*, *current-2power*)
= **if** all-zero-bitvp (*cb*) **then** 0
  else bv-to-nat-induct (**if** all-zero-bitvp (and-bitv (*bv*, *cb*)) **then** 0
      else *current-2power* **endif**
      + *value*,
      bv,
      append (cdr (*cb*), list (0)),
      2 $*$ *current-2power*) **endif**

DEFINITION:
double-cdr-with-sub1-induct $(x,\ y,\ n)$
$=$ **if** listp (x) **then** double-cdr-with-sub1-induct (cdr (x), cdr (y), $n - 1$)
else t endif

THEOREM: bit-vectorp-and-bitv-better
bit-vectorp (and-bitv $(x,\ y)$, $size$) $=$ (length (x) $=$ fix $(size)$)

THEOREM: lessp-bv-to-nat-exp
bit-vectorp $(x,\ size)$ $\rightarrow$ (bv-to-nat (x) $<$ exp $(2,\ size)$)

THEOREM: equal-bv-to-nat-0
bit-vectorp $(x,\ size)$ $\rightarrow$ ((bv-to-nat (x) $=$ 0) $=$ all-zero-bitvp (x))

```
;(prove-lemma commutativity-of-and-bitv (rewrite)
;      (implies
;       (equal (length x) (length y))
;       (equal (and-bitv x y) (and-bitv y x))))
```

EVENT: Enable commutativity-of-and-bitv.

THEOREM: commutativity2-of-and-bitv
(length (x) $=$ length (y))
$\rightarrow$ (and-bitv (x, and-bitv $(y,\ z)$) $=$ and-bitv (y, and-bitv $(x,\ z)$))

THEOREM: associativity-of-and-bitv
(length (x) $=$ length (y))
$\rightarrow$ (and-bitv (and-bitv $(x,\ y)$, z) $=$ and-bitv (x, and-bitv $(y,\ z)$))

THEOREM: all-zero-bitvp-and-bitv
all-zero-bitvp (x)
$\rightarrow$ (all-zero-bitvp (and-bitv $(x,\ y)$) $\wedge$ all-zero-bitvp (and-bitv $(y,\ x)$))

THEOREM: bv-to-nat-loop-clock-open
all-zero-bitvp (x) $\rightarrow$ (bv-to-nat-loop-clock $(x,\ y)$ $=$ 9)

THEOREM: bv-to-nat2-helper-hack
(all-zero-bitvp (z) $\wedge$ (length (d) $=$ length (z)))
$\rightarrow$ (bv-to-nat2-helper (cons $(1,\ d)$, cons $(1,\ z)$, v) $=$ fix (v))

THEOREM: bv-to-nat2-helper-hack2
(all-zero-bitvp (z) $\wedge$ (length (d) $=$ length (z)))
$\rightarrow$ (bv-to-nat2-helper (cons $(0,\ d)$, cons $(1,\ z)$, v) $=$ 0)

THEOREM: correctness-of-bv-to-nat-general

```
(listp (ctrl-stk)
 ∧  at-least-morep (length (temp-stk), 3, max-temp-stk-size)
 ∧  (definition ('bv-to-nat, prog-segment) = BV-TO-NAT-PROGRAM)
 ∧  (c2p ∈ N)
 ∧  (0 < word-size)
 ∧  at-most-one-bit-on (cb)
 ∧  (c2p = bv-to-nat (cb))
 ∧  bit-vectorp (bv, word-size)
 ∧  bit-vectorp (cb, word-size)
 ∧  (value ∈ N)
 ∧  (value < c2p))
→   (p (p-state ('(pc (bv-to-nat . 5)),
                 cons (list (list (cons ('bv, list ('bitv, bv)),
                                   cons ('current-bit, list ('bitv, cb)),
                                   cons ('current-2power, list ('nat, c2p))),
                             ret-pc),
                       ctrl-stk),
                 cons (list ('nat, value), temp-stk),
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run),
        bv-to-nat-loop-clock (cb, bv))
     =  p-state (ret-pc,
                 ctrl-stk,
                 cons (list ('nat,
                             bv-to-nat2-helper (bv, cb, c2p) + value),
                       temp-stk),
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run))
```

DEFINITION:

```
bv-to-nat-clock (word-size, bv)
=  (8 + bv-to-nat-loop-clock (one-bit-vector (word-size), bv))
```

DEFINITION:

```
bv-to-nat-input-conditionp (p0)
```

```
=   ((car (top (p-temp-stk (p0))) = 'bitv)
     ∧  (cddr (top (p-temp-stk (p0))) = nil)
     ∧  bit-vectorp (cadr (top (p-temp-stk (p0))), p-word-size (p0))
     ∧  at-least-morep (p-ctrl-stk-size (p-ctrl-stk (p0)),
                        7,
                        p-max-ctrl-stk-size (p0))
     ∧  (0 < p-word-size (p0))
     ∧  at-least-morep (length (p-temp-stk (p0)),
                        2,
                        p-max-temp-stk-size (p0))
     ∧  listp (p-ctrl-stk (p0)))
```

THEOREM: correctness-of-bv-to-nat-helper

```
((p0 = p-state (pc,
                ctrl-stk,
                cons (list ('bitv, bv), temp-stk),
                prog-segment,
                data-segment,
                max-ctrl-stk-size,
                max-temp-stk-size,
                word-size,
                'run))
 ∧  (p-current-instruction (p0) = '(call bv-to-nat))
 ∧  (definition ('bv-to-nat, prog-segment) = BV-TO-NAT-PROGRAM)
 ∧  (definition ('push-1-vector, prog-segment)
      =  push-1-vector-program (word-size))
 ∧  bv-to-nat-input-conditionp (p0))
→   (p (p0, bv-to-nat-clock (word-size, bv))
     =  p-state (add1-addr (pc),
                 ctrl-stk,
                 cons (list ('nat, bv-to-nat2 (bv)), temp-stk),
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run))
```

THEOREM: bit-vectorp-append-better

```
bit-vectorp (x, length (x))
→   (bit-vectorp (append (x, y), size)
     =  (bit-vectorp (y, length (y))
         ∧  (fix (size) = (length (x) + length (y)))))
```

THEOREM: bit-vectorp-hack

bit-vectorp $(x,\ a + \text{length}(\text{cdr}(x)))$
$= \quad (\textbf{if } \text{listp}(x) \textbf{ then } a = 1$
$\quad \textbf{else } a \simeq 0 \textbf{ endif}$
$\quad \land \quad \text{bit-vectorp}(x, \text{length}(x)))$

Theorem: bv-to-nat-one-bit
$((\neg\ \text{all-zero-bitvp}(x)) \land \text{at-most-one-bit-on}(x) \land \text{bit-vectorp}(x,\ \mathit{size}))$
$\rightarrow \quad (\text{bv-to-nat}(x) = \exp(2, \text{trailing-zeros}(x)))$

Theorem: lessp-sub1-plus-hack
$((x - 1) < (y + x)) = ((x \not\simeq 0) \lor (y \not\simeq 0))$

Theorem: quotient-plus-hack
$((a + b + b) \div 2) = ((a \div 2) + b)$

Theorem: bv-to-nat-all-but-last
bit-vectorp $(x,\ \mathit{size})$
$\rightarrow \quad (\text{bv-to-nat}(\text{all-but-last}(x)) = (\text{bv-to-nat}(x) \div 2))$

Theorem: make-list-from-cons
make-list-from $(n,\ \text{cons}(a,\ b))$
$= \quad \textbf{if } n \simeq 0 \textbf{ then nil}$
$\quad \textbf{else } \text{cons}(a, \text{make-list-from}(n - 1,\ b)) \textbf{ endif}$

Theorem: equal-plus-times-hack
$((a + (a * b) + (a * c)) = (a * d))$
$= \quad ((a \simeq 0) \lor ((1 + b + c) = d))$

```
;(defn nth (n list)
;  (if (zerop n)
;      (car list)
;    (nth (sub1 n) (cdr list))))
```

Event: Enable nth.

Theorem: last-make-list-from
last $(\text{make-list-from}(n,\ x))$
$= \quad \textbf{if } n \simeq 0 \textbf{ then } 0$
$\quad \textbf{else } \text{nth}(n - 1,\ x) \textbf{ endif}$

Theorem: make-list-from-nlistp
$(x \simeq \textbf{nil}) \rightarrow (\text{make-list-from}(n,\ x) = \text{zero-bit-vector}(n))$

Theorem: nth-nlistp
$(x \simeq \textbf{nil}) \rightarrow (\text{nth}(n,\ x) = 0)$

THEOREM: bit-vectorp-make-list-from
bit-vectorp $(x,\ size) \rightarrow$ bit-vectorp (make-list-from $(n,\ x),\ n)$

THEOREM: equal-bv-to-nat-0-2
bit-vectorp $(x,$ length $(x)) \rightarrow (($bv-to-nat $(x) = \mathtt{0}) =$ all-zero-bitvp $(x))$

THEOREM: bv-to-nat-make-list-from-from-sub1-make-list-from
(bv-to-nat (make-list-from $(n - 1,\ v)) = \mathtt{0})$
$\rightarrow$ (bv-to-nat (make-list-from $(n,\ v))$
$=$ **if** $n \simeq \mathtt{0}$ **then** $\mathtt{0}$
else fix-bit (nth $(n - 1,\ v))$ **endif**)

THEOREM: plus-bv-to-nat-make-list-from
(bv-to-nat (make-list-from $(z - 1,\ v))$ + bv-to-nat (make-list-from $(z - 1,\ v)))$
$=$ (bv-to-nat (make-list-from $(z,\ v))$
$-$ fix-bit (last (make-list-from $(z,\ v))))$

THEOREM: equal-bv-to-nat-1
bit-vectorp $(x,$ length $(x))$
$\rightarrow$ ((bv-to-nat $(x) = \mathtt{1})$
$=$ (all-zero-bitvp (all-but-last $(x)) \land$ (last $(x) = \mathtt{1})))$

THEOREM: lessp-1-hack
$(\mathtt{1} < a) = ((a \not\simeq \mathtt{0}) \land (a \neq \mathtt{1}))$

THEOREM: not-equal-nth-0-means
((nth $(n,\ v) \neq \mathtt{0}) \land (n < s))$
$\rightarrow$ ($\neg$ all-zero-bitvp (make-list-from $(s,\ v)))$

THEOREM: all-zero-bitvp-all-but-last-means
(all-zero-bitvp (all-but-last $(x))$
$\land$ (length $(x) =$ length $(y))$
$\land$ $(\mathtt{0} <$ trailing-zeros-helper $(y,\ acc)))$
$\rightarrow$ all-zero-bitvp (and-bitv $(x,\ y))$

THEOREM: all-zero-bitvp-all-but-last-means-spec
(all-zero-bitvp (all-but-last $(x))$
$\land$ (length $(x) =$ length $(y))$
$\land$ $(\mathtt{0} <$ trailing-zeros-helper $(y,\ \mathtt{0})))$
$\rightarrow$ all-zero-bitvp (and-bitv $(x,\ y))$

THEOREM: all-zero-means-to-and-bitv
all-zero-bitvp (x)
$\rightarrow$ ((and-bitv $(x,\ y) =$ zero-bit-vector (length $(x)))$
$\land$ (and-bitv $(y,\ x) =$ zero-bit-vector (length $(y))))$

THEOREM: trailing-zeros-nth-proof
$((n = ((\text{length}(x) - \text{trailing-zeros-helper}(x,\ acc)) - 1))$
$\land \quad (\neg\ \text{all-zero-bitvp}(x)))$
$\rightarrow \quad (\text{nth}(n,\ x) \neq \mathtt{0})$

THEOREM: trailing-zeros-nth-spec
$((n = ((\text{length}(x) - \text{trailing-zeros-helper}(x,\ \mathtt{0})) - 1))$
$\land \quad (\neg\ \text{all-zero-bitvp}(x)))$
$\rightarrow \quad (\text{nth}(n,\ x) \neq \mathtt{0})$

THEOREM: and-bitv-special
$((\text{nth}(n,\ w) = \mathtt{0})$
$\land \quad (\text{nth}(n,\ x) \neq \mathtt{0})$
$\land \quad (\text{length}(x) = \text{length}(x))$
$\land \quad \text{at-most-one-bit-on}(x))$
$\rightarrow \quad (\text{and-bitv}(w,\ x) = \text{zero-bit-vector}(\text{length}(w)))$

THEOREM: equal-trailing-zeros-length-spec
$(\text{trailing-zeros-helper}(x,\ acc) = (acc + \text{length}(x)))$
$= \quad \text{all-zero-bitvp}(x)$

THEOREM: equal-trailing-zeros-length
$(acc \simeq \mathtt{0})$
$\rightarrow \quad ((\text{trailing-zeros-helper}(x,\ acc) = \text{length}(x)) = \text{all-zero-bitvp}(x))$

THEOREM: bit-vectorp-trailing-zeros
$\text{bit-vectorp}(x,\ \text{trailing-zeros-helper}(x,\ acc))$
$= \quad (\text{all-zero-bitvp}(x) \land \text{bit-vectorp}(x,\ \text{length}(x)) \land (acc \simeq \mathtt{0}))$

THEOREM: quotient-exp-hack
$(b \not\simeq \mathtt{0})$
$\rightarrow \quad (((x + \exp(b,\ y)) \div b)$
$\quad = \quad \textbf{if}\ y \simeq \mathtt{0}\ \textbf{then}\ (1 + x) \div b$
$\quad\quad \textbf{else}\ \exp(b,\ y - 1) + (x \div b)\ \textbf{endif})$

```
;(defn properp (list)
;  (if (listp list)
;      (properp (cdr list))
;    (equal list nil)))
```

THEOREM: bit-vectorp-means-properp
$\text{bit-vectorp}(x,\ \text{length}(x)) \rightarrow \text{properp}(x)$

THEOREM: make-list-from-simplify
$((n = \text{length}(x)) \land \text{properp}(x)) \rightarrow (\text{make-list-from}(n,\ x) = x)$

THEOREM: equal-add1-plus-hack

$$(((1 + (a + b)) = (c + (d + a))) = ((1 + b) = (c + d)))$$
$$\land \quad (((1 + (a + b)) = (c + a)) = ((1 + b) = \mathrm{fix}\,(c)))$$
$$\land \quad (((1 + a) = (b + a)) = (\mathtt{1} = \mathrm{fix}\,(b)))$$

THEOREM: plus-quotient-bv-to-nat

$$((\text{bv-to-nat}\,(x) \div \mathtt{2}) + (\text{bv-to-nat}\,(x) \div \mathtt{2}))$$
$$= \quad (\text{bv-to-nat}\,(x) - \text{fix-bit}\,(\mathrm{last}\,(x)))$$

THEOREM: equal-plus-times-hack2

$$(((a * b) + (a * c)) = (a * d))$$
$$= \quad ((a \simeq \mathtt{0}) \lor ((b + c) = \mathrm{fix}\,(d)))$$

THEOREM: and-bitv-special-special

$$((\mathrm{last}\,(w) = \mathtt{0})$$
$$\land \quad (\mathrm{last}\,(x) \neq \mathtt{0})$$
$$\land \quad (\mathrm{length}\,(x) = \mathrm{length}\,(w))$$
$$\land \quad \text{at-most-one-bit-on}\,(x))$$
$$\rightarrow \quad (\text{and-bitv}\,(w,\ x) = \text{zero-bit-vector}\,(\mathrm{length}\,(w)))$$

THEOREM: and-bitv-special-3

$$((\mathrm{length}\,(x) = \mathrm{length}\,(y)) \land (\mathrm{last}\,(x) \neq \mathtt{0}) \land (\mathrm{last}\,(y) \neq \mathtt{0}))$$
$$\rightarrow \quad (\neg\ \text{all-zero-bitvp}\,(\text{and-bitv}\,(x,\ y)))$$

THEOREM: and-bitv-special-4

$$((\mathrm{nth}\,(n,\ w) \neq \mathtt{0}) \land (\mathrm{nth}\,(n,\ x) \neq \mathtt{0}) \land (\mathrm{length}\,(x) = \mathrm{length}\,(x)))$$
$$\rightarrow \quad (\neg\ \text{all-zero-bitvp}\,(\text{and-bitv}\,(w,\ x)))$$

THEOREM: and-bitv-special-5

$$((\mathrm{last}\,(w) = \mathtt{0})$$
$$\land \quad (\mathrm{last}\,(x) \neq \mathtt{0})$$
$$\land \quad ((1 + \mathrm{length}\,(x)) = \mathrm{length}\,(w))$$
$$\land \quad \text{at-most-one-bit-on}\,(x))$$
$$\rightarrow \quad (\text{and-bitv}\,(w,\ \mathrm{cons}\,(\mathtt{0},\ x)) = \text{zero-bit-vector}\,(\mathrm{length}\,(w)))$$

THEOREM: and-bitv-special-6

$$((\mathrm{last}\,(w) \neq \mathtt{0}) \land (\mathrm{last}\,(x) \neq \mathtt{0}) \land ((1 + \mathrm{length}\,(x)) = \mathrm{length}\,(w)))$$
$$\rightarrow \quad (\neg\ \text{all-zero-bitvp}\,(\text{and-bitv}\,(w,\ \mathrm{cons}\,(\mathtt{0},\ x))))$$

THEOREM: not-last-0-means-not-all-0

$$(\mathrm{last}\,(x) \neq \mathtt{0}) \rightarrow (\neg\ \text{all-zero-bitvp}\,(x))$$

THEOREM: bit-vectorp-plus-length-hack

$$(\text{bit-vectorp}\,(x,\ z + \mathrm{length}\,(x)) = (\text{bit-vectorp}\,(x,\ \mathrm{length}\,(x)) \land (z \simeq \mathtt{0})))$$
$$\land \quad (\text{bit-vectorp}\,(x,\ (1 + z) + \mathrm{length}\,(\mathrm{cdr}\,(x)))$$
$$\qquad = \quad (\text{bit-vectorp}\,(x,\ \mathrm{length}\,(x)) \land \mathrm{listp}\,(x) \land (z \simeq \mathtt{0})))$$

THEOREM: last-append
last (append (x, y))
$=$ **if** listp (y) **then** last (y)
else last (x) **endif**

THEOREM: bv-to-nat2-helper-bv-to-nat
(at-most-one-bit-on (cb)
$\land$ bit-vectorp (cb, $size$)
$\land$ bit-vectorp (bv, $size$)
$\land$ ($c2$ = bv-to-nat (cb)))
$\rightarrow$ (bv-to-nat2-helper (bv, cb, $c2$)
$=$ bv-to-nat (append (make-list-from (length (bv)
$-$ trailing-zeros (cb),
bv),
zero-bit-vector (trailing-zeros (cb)))))

THEOREM: bv-to-nat2-helper-bv-to-nat-better
(at-most-one-bit-on (cb)
$\land$ bit-vectorp (cb, length (cb))
$\land$ bit-vectorp (bv, length (cb))
$\land$ ($c2$ = bv-to-nat (cb)))
$\rightarrow$ (bv-to-nat2-helper (bv, cb, $c2$)
$=$ bv-to-nat (append (make-list-from (length (bv)
$-$ trailing-zeros (cb),
bv),
zero-bit-vector (trailing-zeros (cb)))))

THEOREM: nlistp-bv-to-nat2
($\neg$ listp (x)) $\rightarrow$ (bv-to-nat2-helper (x, a, b) = 0)

THEOREM: bv-to-nat2-bv-to-nat-helper
($x \simeq$ **nil**) $\rightarrow$ (bv-to-nat2 (x) = bv-to-nat (x))

THEOREM: bit-vectorp-one-bit-vector-rewrite
bit-vectorp (one-bit-vector (s), n)
$=$ **if** $s \simeq 0$ **then** $n = 1$
else fix (s) = fix (n) **endif**

THEOREM: trailing-zeros-helper-one-bit-vector
trailing-zeros-helper (one-bit-vector (n), acc) = 0

THEOREM: bv-to-nat2-bv-to-nat
bit-vectorp (x, length (x)) $\rightarrow$ (bv-to-nat2 (x) = bv-to-nat (x))

THEOREM: bv-length-weaker
bit-vectorp (x, s) $\rightarrow$ bit-vectorp (x, length (x))

THEOREM: correctness-of-bv-to-nat

```
((p0 = p-state (pc,
                ctrl-stk,
                cons (b, temp-stk),
                prog-segment,
                data-segment,
                max-ctrl-stk-size,
                max-temp-stk-size,
                word-size,
                'run))
 ∧  (p-current-instruction (p0) = '(call bv-to-nat))
 ∧  (definition ('bv-to-nat, prog-segment) = BV-TO-NAT-PROGRAM)
 ∧  (definition ('push-1-vector, prog-segment)
       =  push-1-vector-program (word-size))
 ∧  (bv = cadr (b))
 ∧  bv-to-nat-input-conditionp (p0))
→   (p (p-state (pc,
                 ctrl-stk,
                 cons (b, temp-stk),
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run),
        bv-to-nat-clock (word-size, bv))
      =  p-state (add1-addr (pc),
                  ctrl-stk,
                  cons (list ('nat, bv-to-nat (bv)), temp-stk),
                  prog-segment,
                  data-segment,
                  max-ctrl-stk-size,
                  max-temp-stk-size,
                  word-size,
                  'run))
```

```
;;;; number-with-at-least
```

DEFINITION:
NUMBER-WITH-AT-LEAST-PROGRAM

```
=  '(number-with-at-least
     (nums-addr numnums min)
```

```
((i (nat 0)))
(push-constant (nat 0))
(set-local i)
(dl loop nil (push-local nums-addr))
(fetch)
(push-local min)
(lt-nat)
(test-bool-and-jump t lab)
(add1-nat)
(dl lab nil (push-local numnums))
(push-local i)
(add1-nat)
(set-local i)
(sub-nat)
(test-nat-and-jump zero done)
(push-local nums-addr)
(push-constant (nat 1))
(add-addr)
(pop-local nums-addr)
(jump loop)
(dl done nil (ret)))
```

DEFINITION:
EXAMPLE-NUMBER-WITH-AT-LEAST-STATE

```
=  p-state ('(pc (main . 0)),
           '((nil (pc (main . 0)))),
           nil,
           list ('(main nil nil
                 (push-constant (addr (nums . 0)))
                 (push-constant (nat 5))
                 (push-constant (nat 3))
                 (call number-with-at-least)
                 (ret)),
               NUMBER-WITH-AT-LEAST-PROGRAM),
           '((nums
              (nat 3)
              (nat 8)
              (nat 9)
              (nat 2)
              (nat 100))),
           10,
           8,
           8,
           'run)
```

DEFINITION:
number-with-at-least ($numlist$, min)
$=$ **if** listp ($numlist$)
then if car ($numlist$) $<$ min
then number-with-at-least (cdr ($numlist$), min)
else $1 +$ number-with-at-least (cdr ($numlist$), min) **endif**
else `0` **endif**

DEFINITION:
nat-list-piton ($array$, $word\text{-}size$)
$=$ **if** listp ($array$)
then listp (car ($array$))
$\wedge$ (caar ($array$) $=$ `'nat`)
$\wedge$ (cadar ($array$) $\in \mathbf{N}$)
$\wedge$ (cddar ($array$) $=$ **nil**)
$\wedge$ (cadar ($array$) $<$ exp (`2`, $word\text{-}size$))
$\wedge$ nat-list-piton (cdr ($array$), $word\text{-}size$)
else $array$ $=$ **nil endif**

DEFINITION:
number-with-at-least-general-induct (i, $current$, n, s, min, $data\text{-}segment$)
$=$ **if** $i < n$
then number-with-at-least-general-induct ($1 + i$,
if cadr (get (i,
array (s,
$data\text{-}segment$)))
$<$ min **then** $current$
else $1 +$ $current$ **endif**,
n,
s,
min,
$data\text{-}segment$)
else **t** **endif**

DEFINITION:
number-with-at-least-clock-loop (i, min, $array$)
$=$ **if** $i \not<$ length ($array$) **then** `0`
else if cadr (get (i, $array$)) $<$ min **then** `0`
else `1` **endif**
$+$ **if** $(1 + i) =$ length ($array$) **then** `12`
else `16` $+$ number-with-at-least-clock-loop ($1 + i$,
min,
$array$) **endif endif**

THEOREM: equal-difference-1
$((x - y) = 1) = (x = (1 + y))$

THEOREM: nat-list-piton-means
$(\text{nat-list-piton}\,(state,\ size) \land (p < \text{length}\,(state)))$
$\rightarrow\ ((\text{car}\,(\text{get}\,(p,\ state)) = \texttt{'nat})$
$\quad \land\ \text{listp}\,(\text{get}\,(p,\ state))$
$\quad \land\ (\text{cadr}\,(\text{get}\,(p,\ state)) \in \mathbf{N})$
$\quad \land\ (\text{cadr}\,(\text{get}\,(p,\ state)) < \exp\,(2,\ size))$
$\quad \land\ (\text{cddr}\,(\text{get}\,(p,\ state)) = \mathbf{nil}))$

THEOREM: number-with-at-least-nlistp
$(\neg\ \text{listp}\,(x)) \rightarrow (\text{number-with-at-least}\,(x,\ min) = 0)$

THEOREM: equal-add1-length
$((1 + x) = \text{length}\,(y)) = (\text{listp}\,(y) \land (\text{fix}\,(x) = \text{length}\,(\text{cdr}\,(y))))$

EVENT: Disable number-with-at-least-clock-loop.

THEOREM: length-cdr-untag-array
$\text{length}\,(\text{cdr}\,(\text{untag-array}\,(x))) = \text{length}\,(\text{cdr}\,(x))$

THEOREM: number-with-at-least-correctness-general
$((\text{length}\,(\text{array}\,(s,\ data\text{-}segment)) < \exp\,(2,\ word\text{-}size))$
$\land\ (word\text{-}size \not\simeq 0)$
$\land\ \text{listp}\,(ctrl\text{-}stk)$
$\land\ \text{nat-list-piton}\,(\text{array}\,(s,\ data\text{-}segment),\ word\text{-}size)$
$\land\ \text{at-least-morep}\,(\text{length}\,(temp\text{-}stk),\ 3,\ max\text{-}temp\text{-}stk\text{-}size)$
$\land\ (\text{definition}\,(\texttt{'number-with-at-least},\ prog\text{-}segment)$
$\quad = \ \text{NUMBER-WITH-AT-LEAST-PROGRAM})$
$\land\ \text{definedp}\,(s,\ data\text{-}segment)$
$\land\ (min < \exp\,(2,\ word\text{-}size))$
$\land\ (min \in \mathbf{N})$
$\land\ (i < n)$
$\land\ (i \in \mathbf{N})$
$\land\ (n < \exp\,(2,\ word\text{-}size))$
$\land\ (current \in \mathbf{N})$
$\land\ (i \not< current)$
$\land\ (n = \text{length}\,(\text{array}\,(s,\ data\text{-}segment))))$
$\rightarrow\ (\text{p}\,(\text{p-state}\,(\texttt{'(pc (number-with-at-least . 2))},$
$\qquad \text{cons}\,(\text{list}\,(\text{list}\,(\text{cons}\,(\texttt{'nums-addr},\ \text{list}\,(\texttt{'addr},\ \text{cons}\,(s,\ i))),$
$\qquad\qquad \text{cons}\,(\texttt{'numnums},\ \text{list}\,(\texttt{'nat},\ n)),$
$\qquad\qquad \text{cons}\,(\texttt{'min},\ \text{list}\,(\texttt{'nat},\ min)),$
$\qquad\qquad \text{cons}\,(\texttt{'i},\ \text{list}\,(\texttt{'nat},\ i))),$
$\qquad\qquad ret\text{-}pc),$
$\qquad\quad ctrl\text{-}stk),$
$\qquad \text{cons}\,(\text{list}\,(\texttt{'nat},\ current),\ temp\text{-}stk),$

$$
\begin{array}{l}
\quad\quad\quad\quad \mathit{prog\text{-}segment}, \\
\quad\quad\quad\quad \mathit{data\text{-}segment}, \\
\quad\quad\quad\quad \mathit{max\text{-}ctrl\text{-}stk\text{-}size}, \\
\quad\quad\quad\quad \mathit{max\text{-}temp\text{-}stk\text{-}size}, \\
\quad\quad\quad\quad \mathit{word\text{-}size}, \\
\quad\quad\quad\quad \texttt{'run}), \\
\quad \text{number-with-at-least-clock-loop}\,(i,\ \mathit{min},\ \text{array}\,(s,\ \mathit{data\text{-}segment}))) \\
= \quad \text{p-state}\,(\mathit{ret\text{-}pc}, \\
\quad\quad\quad\quad \mathit{ctrl\text{-}stk}, \\
\quad\quad\quad\quad \text{cons}\,(\text{list}\,(\texttt{'nat}, \\
\quad\quad\quad\quad\quad\quad \mathit{current} + \text{number-with-at-least}\,(\text{nthcdr}\,(i, \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{untag-array}\,(\text{array}\,(s, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{data\text{-}segment}))), \\
\quad\quad\quad\quad\quad\quad\quad \mathit{min})), \\
\quad\quad\quad\quad\quad \mathit{temp\text{-}stk}), \\
\quad\quad\quad\quad \mathit{prog\text{-}segment}, \\
\quad\quad\quad\quad \mathit{data\text{-}segment}, \\
\quad\quad\quad\quad \mathit{max\text{-}ctrl\text{-}stk\text{-}size}, \\
\quad\quad\quad\quad \mathit{max\text{-}temp\text{-}stk\text{-}size}, \\
\quad\quad\quad\quad \mathit{word\text{-}size}, \\
\quad\quad\quad\quad \texttt{'run}))
\end{array}
$$

DEFINITION:

$$
\begin{array}{l}
\text{number-with-at-least-clock}\,(\mathit{min},\ \mathit{array}) \\
= \quad (\texttt{3} + \text{number-with-at-least-clock-loop}\,(\texttt{0},\ \mathit{min},\ \mathit{array}))
\end{array}
$$

DEFINITION:

$$
\begin{array}{l}
\text{number-with-at-least-input-conditionp}\,(p0) \\
= \quad ((\text{car}\,(\text{car}\,(\text{p-temp-stk}\,(p0))) = \texttt{'nat}) \\
\quad\quad \land \quad (\text{cddr}\,(\text{car}\,(\text{p-temp-stk}\,(p0))) = \textbf{nil}) \\
\quad\quad \land \quad (\text{car}\,(\text{cadr}\,(\text{p-temp-stk}\,(p0))) = \texttt{'nat}) \\
\quad\quad \land \quad (\text{cddr}\,(\text{cadr}\,(\text{p-temp-stk}\,(p0))) = \textbf{nil}) \\
\quad\quad \land \quad (\text{car}\,(\text{caddr}\,(\text{p-temp-stk}\,(p0))) = \texttt{'addr}) \\
\quad\quad \land \quad (\text{cdadr}\,(\text{caddr}\,(\text{p-temp-stk}\,(p0))) = \texttt{0}) \\
\quad\quad \land \quad (\text{cddr}\,(\text{caddr}\,(\text{p-temp-stk}\,(p0))) = \textbf{nil}) \\
\quad\quad \land \quad \text{listp}\,(\text{cadr}\,(\text{caddr}\,(\text{p-temp-stk}\,(p0)))) \\
\quad\quad \land \quad (\text{length}\,(\text{array}\,(\text{car}\,(\text{cadr}\,(\text{caddr}\,(\text{p-temp-stk}\,(p0)))), \\
\quad\quad\quad\quad\quad\quad\quad \text{p-data-segment}\,(p0))) \\
\quad\quad\quad\quad < \quad \exp\,(\texttt{2},\ \text{p-word-size}\,(p0))) \\
\quad\quad \land \quad (\text{p-word-size}\,(p0) \not\simeq \texttt{0}) \\
\quad\quad \land \quad \text{listp}\,(\text{p-ctrl-stk}\,(p0)) \\
\quad\quad \land \quad \text{nat-list-piton}\,(\text{array}\,(\text{car}\,(\text{cadr}\,(\text{caddr}\,(\text{p-temp-stk}\,(p0)))), \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{p-data-segment}\,(p0)), \\
\quad\quad\quad\quad\quad\quad \text{p-word-size}\,(p0))
\end{array}
$$

$\land$ at-least-morep (p-ctrl-stk-size (p-ctrl-stk ($p0$)),
6,
p-max-ctrl-stk-size ($p0$))
$\land$ at-least-morep (length (p-temp-stk ($p0$)),
0,
p-max-temp-stk-size ($p0$))
$\land$ (definition (**'number-with-at-least**, p-prog-segment ($p0$))
= NUMBER-WITH-AT-LEAST-PROGRAM)
$\land$ definedp (car (cadr (caddr (p-temp-stk ($p0$)))), p-data-segment ($p0$))
$\land$ (cadr (car (p-temp-stk ($p0$))) $<$ exp (2, p-word-size ($p0$)))
$\land$ (cadr (car (p-temp-stk ($p0$))) $\in \mathbf{N}$)
$\land$ (0 $<$ cadr (cadr (p-temp-stk ($p0$))))
$\land$ (cadr (cadr (p-temp-stk ($p0$)))
= length (array (car (cadr (caddr (p-temp-stk ($p0$)))),
p-data-segment ($p0$)))))

THEOREM: correctness-of-number-with-at-least
(($p0$ = p-state (pc,
ctrl-stk,
cons (m, cons (n, cons (s, *temp-stk*))),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
'run))
$\land$ (p-current-instruction ($p0$) = **'(call number-with-at-least)**)
$\land$ number-with-at-least-input-conditionp ($p0$)
$\land$ ($minc$ = cadr (m))
$\land$ ($arrayc$ = array (car (cadr (s)), *data-segment*)))
$\rightarrow$ (p (p-state (pc,
ctrl-stk,
cons (m, cons (n, cons (s, *temp-stk*))),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
'run),
number-with-at-least-clock ($minc$, $arrayc$))
= p-state (add1-addr (pc),
ctrl-stk,
cons (list (**'nat**,
number-with-at-least (untag-array (array (car (cadr (s)),

data-segment)),
cadr (m))),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
'run))

```
;; highest-bit
```

DEFINITION:
HIGHEST-BIT-PROGRAM

```
=  '(highest-bit
     (bv)
     ((cb (nat 0)))
     (call push-1-vector)
     (set-local cb)
     (rsh-bitv)
     (dl loop nil (push-local cb))
     (test-bitv-and-jump all-zero done)
     (push-local bv)
     (push-local cb)
     (and-bitv)
     (test-bitv-and-jump all-zero lab)
     (pop)
     (push-local cb)
     (dl lab nil (push-local cb))
     (lsh-bitv)
     (pop-local cb)
     (jump loop)
     (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-HIGHEST-BIT-STATE

```
=  p-state ('(pc (main . 0)),
            '((nil (pc (main . 0)))),
            nil,
            list ('(main nil nil
                   (push-constant (bitv (0 0 0 0 0 0 0 0)))
                   (call highest-bit)
                   (push-constant (bitv (0 0 1 1 0 1 0 0)))
```

```
        (call highest-bit)
        (push-constant (bitv (1 0 1 1 0 1 0 0)))
        (call highest-bit)
        (ret)),
      HIGHEST-BIT-PROGRAM,
      push-1-vector-program (8)),
    nil,
    10,
    8,
    8,
    'run)
```

DEFINITION:
highest-bit (x)
= **if** listp (x)
  then if car (x) = 0 **then** cons (0, highest-bit (cdr (x)))
    else cons (1, zero-bit-vector (length (cdr (x)))) **endif**
  else nil endif

THEOREM: listp-highest-bit
listp (highest-bit (x)) = listp (x)

THEOREM: length-highest-bit
length (highest-bit (x)) = length (x)

DEFINITION:
highest-bit-loop-clock (cb, bv)
= **if** all-zero-bitvp (cb) **then** 3
  else 10
    + **if** all-zero-bitvp (and-bitv (cb, bv)) **then** 0
      else 2 **endif**
    + highest-bit-loop-clock (append (cdr (cb), '(0)), bv) **endif**

DEFINITION:
highest-bit-induct ($current$, bv, cb)
= **if** all-zero-bitvp (cb) **then t**
  else highest-bit-induct (**if** all-zero-bitvp (and-bitv (cb, bv))
        then $current$
        else cb **endif**,
        bv,
        append (cdr (cb), '(0))) **endif**

THEOREM: bit-vectorp-simple-not
(length (x) $\neq$ fix (y)) $\rightarrow$ ($\neg$ bit-vectorp (x, y))

THEOREM: at-most-one-bit-on-cdr
$\text{at-most-one-bit-on}\,(x) \rightarrow \text{at-most-one-bit-on}\,(\text{cdr}\,(x))$

THEOREM: equal-one-bit-vector
$$
\begin{aligned}
& (x = \text{one-bit-vector}\,(z)) \\
= \quad & (((z \simeq \texttt{0}) \land (x = \texttt{'(1)})) \\
& \quad \lor \quad (\text{bit-vectorp}\,(x,\ z) \\
& \qquad\quad \land \quad \text{all-zero-bitvp}\,(\text{all-but-last}\,(x)) \\
& \qquad\quad \land \quad (\text{last}\,(x) \neq \texttt{0})))
\end{aligned}
$$

THEOREM: at-most-one-bit-is-all-zeros
$$
\begin{aligned}
& (\text{last}\,(x) \neq \texttt{0}) \\
\rightarrow \quad & (\text{at-most-one-bit-on}\,(x) = \text{all-zero-bitvp}\,(\text{all-but-last}\,(x)))
\end{aligned}
$$

DEFINITION:
$$
\begin{aligned}
& \text{highest-bit2-helper}\,(\mathit{current},\ \mathit{cb},\ \mathit{bv}) \\
= \quad & \textbf{if } \text{all-zero-bitvp}\,(\mathit{cb}) \ \textbf{ then } \mathit{current} \\
& \textbf{else } \text{highest-bit2-helper}\,(\textbf{if } \text{all-zero-bitvp}\,(\text{and-bitv}\,(\mathit{bv},\ \mathit{cb})) \\
& \qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathit{current} \\
& \qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{cb} \textbf{ endif}, \\
& \qquad\qquad\qquad\qquad\qquad\quad \text{append}\,(\text{cdr}\,(\mathit{cb}),\ \texttt{'(0)}), \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathit{bv}) \textbf{ endif}
\end{aligned}
$$

THEOREM: equal-x-zero-bit-vector
$(x = \text{zero-bit-vector}\,(y)) = (\text{all-zero-bitvp}\,(x) \land \text{bit-vectorp}\,(x,\ y))$

THEOREM: all-zero-bitvp-all-but-last-simple
$\text{all-zero-bitvp}\,(x) \rightarrow \text{all-zero-bitvp}\,(\text{all-but-last}\,(x))$

THEOREM: equal-trailing-zeros-acc-irrelevant
$$
\begin{aligned}
& (\text{trailing-zeros-helper}\,(x,\ \mathit{acc1}) = \text{trailing-zeros-helper}\,(x,\ \mathit{acc2})) \\
= \quad & ((\text{fix}\,(\mathit{acc1}) = \text{fix}\,(\mathit{acc2})) \lor (\neg\ \text{all-zero-bitvp}\,(x)))
\end{aligned}
$$

THEOREM: lessp-trailing-zeros
$(\neg\ \text{all-zero-bitvp}\,(z)) \rightarrow (\text{trailing-zeros-helper}\,(z,\ \mathit{acc}) < \text{length}\,(z))$

THEOREM: nthcdr-append
$$
\begin{aligned}
& \text{nthcdr}\,(n,\ \text{append}\,(x,\ y)) \\
= \quad & \textbf{if } \text{length}\,(x) < n \ \textbf{ then } \text{nthcdr}\,(n - \text{length}\,(x),\ y) \\
& \textbf{else } \text{append}\,(\text{nthcdr}\,(n,\ x),\ y) \textbf{ endif}
\end{aligned}
$$

THEOREM: listp-zero-bit-vector
$\text{listp}\,(\text{zero-bit-vector}\,(x)) = (x \not\simeq \texttt{0})$

THEOREM: and-bitv-append
$$
\begin{aligned}
& \text{and-bitv}\,(\text{append}\,(x,\ y),\ z) \\
= \quad & \text{append}\,(\text{and-bitv}\,(x,\ \text{make-list-from}\,(\text{length}\,(x),\ z)), \\
& \qquad\qquad \text{and-bitv}\,(y,\ \text{nthcdr}\,(\text{length}\,(x),\ z)))
\end{aligned}
$$

THEOREM: and-bitv-append2
$(\text{length}(\text{append}(x,\ y)) = \text{length}(z))$
$\rightarrow\quad (\text{and-bitv}(z,\ \text{append}(x,\ y))$
$\quad = \quad \text{append}(\text{and-bitv}(x,\ \text{make-list-from}(\text{length}(x),\ z)),$
$\qquad\qquad \text{and-bitv}(y,\ \text{nthcdr}(\text{length}(x),\ z))))$

THEOREM: all-zero-bitvp-and-bitv-append
$\text{all-zero-bitvp}(\text{and-bitv}(z,\ \text{append}(x,\ y)))$
$= \quad (\text{all-zero-bitvp}(\text{and-bitv}(\text{make-list-from}(\text{length}(x),\ z),\ x))$
$\quad \wedge \quad \text{all-zero-bitvp}(\text{and-bitv}(\text{nthcdr}(\text{length}(x),\ z),\ y)))$

THEOREM: length-cdr-zero-bit-vector
$\text{length}(\text{cdr}(\text{zero-bit-vector}(x))) = (x - 1)$

THEOREM: length-nthcdr
$\text{length}(\text{nthcdr}(n,\ x)) = (\text{length}(x) - n)$

EVENT: Disable and-bitv-special.

EVENT: Disable and-bitv-special-special.

EVENT: Disable and-bitv-special-3.

EVENT: Disable and-bitv-special-4.

EVENT: Disable and-bitv-special-5.

EVENT: Disable and-bitv-special-6.

THEOREM: bit-vectorp-zero-bit-vector-better
$\text{bit-vectorp}(\text{zero-bit-vector}(x),\ \mathit{size}) = (\text{fix}(x) = \text{fix}(\mathit{size}))$

THEOREM: highest-bit-correctness-general
$((\mathit{word\text{-}size} \not\simeq \mathtt{0})$
$\wedge\quad \text{at-most-one-bit-on}(\mathit{cb})$
$\wedge\quad \text{listp}(\mathit{ctrl\text{-}stk})$
$\wedge\quad \text{at-least-morep}(\text{length}(\mathit{temp\text{-}stk}),\ \mathtt{3},\ \mathit{max\text{-}temp\text{-}stk\text{-}size})$
$\wedge\quad (\text{definition}(\mathtt{'highest\text{-}bit},\ \mathit{prog\text{-}segment}) = \text{HIGHEST-BIT-PROGRAM})$
$\wedge\quad \text{bit-vectorp}(\mathit{bv},\ \mathit{word\text{-}size})$
$\wedge\quad \text{bit-vectorp}(\mathit{cb},\ \mathit{word\text{-}size})$
$\wedge\quad \text{bit-vectorp}(\mathit{current},\ \mathit{word\text{-}size}))$
$\rightarrow\quad (\text{p}(\text{p-state}(\mathtt{'(pc\ (highest\text{-}bit\ .\ 3))},$

$$
\begin{array}{l}
\quad\quad \mathrm{cons}\,(\mathrm{list}\,(\mathrm{list}\,(\mathrm{cons}\,(\texttt{'bv},\ \mathrm{list}\,(\texttt{'bitv},\ bv)),\\
\quad\quad\quad\quad\quad\quad \mathrm{cons}\,(\texttt{'cb},\ \mathrm{list}\,(\texttt{'bitv},\ cb))),\\
\quad\quad\quad\quad\quad \mathit{ret\text{-}pc}),\\
\quad\quad\quad\quad \mathit{ctrl\text{-}stk}),\\
\quad\quad \mathrm{cons}\,(\mathrm{list}\,(\texttt{'bitv},\ \mathit{current}),\ \mathit{temp\text{-}stk}),\\
\quad\quad \mathit{prog\text{-}segment},\\
\quad\quad \mathit{data\text{-}segment},\\
\quad\quad \mathit{max\text{-}ctrl\text{-}stk\text{-}size},\\
\quad\quad \mathit{max\text{-}temp\text{-}stk\text{-}size},\\
\quad\quad \mathit{word\text{-}size},\\
\quad\quad \texttt{'run}),\\
\quad \mathrm{highest\text{-}bit\text{-}loop\text{-}clock}\,(cb,\ bv))\\
= \quad \mathrm{p\text{-}state}\,(\mathit{ret\text{-}pc},\\
\quad\quad\quad \mathit{ctrl\text{-}stk},\\
\quad\quad\quad \mathrm{cons}\,(\mathrm{list}\,(\texttt{'bitv},\ \mathrm{highest\text{-}bit2\text{-}helper}\,(\mathit{current},\ cb,\ bv)),\\
\quad\quad\quad\quad\quad \mathit{temp\text{-}stk}),\\
\quad\quad\quad \mathit{prog\text{-}segment},\\
\quad\quad\quad \mathit{data\text{-}segment},\\
\quad\quad\quad \mathit{max\text{-}ctrl\text{-}stk\text{-}size},\\
\quad\quad\quad \mathit{max\text{-}temp\text{-}stk\text{-}size},\\
\quad\quad\quad \mathit{word\text{-}size},\\
\quad\quad\quad \texttt{'run}))
\end{array}
$$

THEOREM: make-list-from-simple
$(n \simeq \texttt{0}) \rightarrow (\mathrm{make\text{-}list\text{-}from}\,(n,\ x) = \mathbf{nil})$

THEOREM: not-all-zero-bitvp-make-list-from
$(\mathrm{all\text{-}zero\text{-}bitvp}\,(\mathrm{make\text{-}list\text{-}from}\,(n1,\ x)) \land (n1 \not< n2))$
$\rightarrow \quad \mathrm{all\text{-}zero\text{-}bitvp}\,(\mathrm{make\text{-}list\text{-}from}\,(n2,\ x))$

EVENT: Disable nthcdr-open.

```
;(prove-lemma listp-append (rewrite)
;      (equal
;       (listp (append x y))
;       (or (listp x) (listp y))))
```

EVENT: Enable listp-append.

THEOREM: highest-bit2-helper-cons-1
$((\neg\ \mathrm{all\text{-}zero\text{-}bitvp}\,(cb))$
$\land \quad \mathrm{at\text{-}most\text{-}one\text{-}bit\text{-}on}\,(cb)$
$\land \quad (\mathrm{length}\,(cb) = \mathrm{length}\,(bv))$
$\land \quad \mathrm{bit\text{-}vectorp}\,(cb,\ \mathrm{length}\,(cb))$

$\wedge$ (car (bv) = 1))
$\rightarrow$ (highest-bit2-helper ($current$, cb, bv)
  = cons (1, zero-bit-vector (length (cdr (bv)))))

Theorem: bit-vectorp-append-cdr-hack
bit-vectorp (x, n) $\rightarrow$ (bit-vectorp (append (cdr (x), '(0)), n) = listp (x))

Definition:
triple-cdr-with-sub1-induct (x, y, z, n)
= **if** $n \simeq 0$ **then t**
else triple-cdr-with-sub1-induct (cdr (x), cdr (y), cdr (z), $n - 1$) **endif**

Theorem: car-append-better
car (append (x, y))
= **if** listp (x) **then** car (x)
else car (y) **endif**

Theorem: open-highest-when-and-not-0
(($\neg$ all-zero-bitvp (and-bitv (x, y))) $\wedge$ (length (x) = length (y)))
$\rightarrow$ (highest-bit2-helper (c, x, y)
  = highest-bit2-helper (x, append (cdr (x), '(0)), y))

```
;(prove-lemma cdr-append (rewrite)
;      (equal
;       (cdr (append x y))
;       (if (listp x)
;   (append (cdr x) y)
; (cdr y))))
```

Event: Enable cdr-append.

Theorem: highest-bit2-helper-cons-0
(($\neg$ all-zero-bitvp (cb))
$\wedge$ at-most-one-bit-on (cb)
$\wedge$ bit-vectorp (cb, $size$)
$\wedge$ bit-vectorp (bv, $size$)
$\wedge$ bit-vectorp ($current$, $size$)
$\wedge$ (car (bv) = 0)
$\wedge$ (car ($current$) = 0))
$\rightarrow$ (highest-bit2-helper ($current$, cb, bv)
  = cons (0, highest-bit2-helper (cdr ($current$), cdr (cb), cdr (bv))))

Theorem: highest-bit2-helper-cons-0-rewrite
(($\neg$ all-zero-bitvp (cb))
$\wedge$ at-most-one-bit-on (cb)

$\wedge$ bit-vectorp (cb, length (cb))
$\wedge$ bit-vectorp (bv, length (cb))
$\wedge$ bit-vectorp ($current$, length (cb))
$\wedge$ (car (bv) = `0`)
$\wedge$ (car ($current$) = `0`))
$\rightarrow$ (highest-bit2-helper ($current$, cb, bv)
  = cons (`0`, highest-bit2-helper (cdr ($current$), cdr (cb), cdr (bv))))

THEOREM: append-zeros-0
(all-zero-bitvp (x) $\wedge$ properp (x)) $\rightarrow$ (append (x, `'(0)`) = cons (`0`, x))

THEOREM: highest-bit2-helper-cons-helper
(bit-vectorp (bv, $size$)
$\wedge$ bit-vectorp (cb, $size$)
$\wedge$ at-most-one-bit-on (cb)
$\wedge$ at-most-one-bit-on ($current$)
$\wedge$ ($\neg$ all-zero-bitvp (cb))
$\wedge$ (all-zero-bitvp ($current$)
  $\vee$ (trailing-zeros ($current$) $<$ trailing-zeros (cb)))
$\wedge$ bit-vectorp ($current$, $size$)
$\wedge$ ($size \not\simeq$ `0`))
$\rightarrow$ (highest-bit2-helper ($current$, cb, bv)
  = **if** car (bv) = `0`
    then cons (`0`,
        highest-bit2-helper (cdr ($current$), cdr (cb), cdr (bv)))
    else cons (`1`, zero-bit-vector ($size$ − 1)) **endif**)

THEOREM: highest-bit2-helper-cons-helper-rewrite
(bit-vectorp (bv, length (bv))
$\wedge$ bit-vectorp (cb, length (bv))
$\wedge$ at-most-one-bit-on (cb)
$\wedge$ at-most-one-bit-on ($current$)
$\wedge$ ($\neg$ all-zero-bitvp (cb))
$\wedge$ (all-zero-bitvp ($current$)
  $\vee$ (trailing-zeros ($current$) $<$ trailing-zeros (cb)))
$\wedge$ bit-vectorp ($current$, length (bv))
$\wedge$ listp (bv))
$\rightarrow$ (highest-bit2-helper ($current$, cb, bv)
  = **if** car (bv) = `0`
    then cons (`0`,
        highest-bit2-helper (cdr ($current$), cdr (cb), cdr (bv)))
    else cons (`1`, zero-bit-vector (length (bv) − 1)) **endif**)

THEOREM: bit-vectorp-cdr-from-free
bit-vectorp (x, n) $\rightarrow$ (bit-vectorp (cdr (x), s) = ((1 + s) = n))

THEOREM: all-zero-bitvp-one-bit-vector
$\neg$ all-zero-bitvp (one-bit-vector (x))

THEOREM: trailing-zeros-one-bit-vector
trailing-zeros (one-bit-vector (n)) $=$ 0

THEOREM: cdr-zero-one-bit-vector
(cdr (one-bit-vector (x))
 $=$ **if** $x < 2$ **then nil**
  else one-bit-vector ($x - 1$) **endif**)
$\wedge$ (cdr (zero-bit-vector (x))
  $=$ **if** $x \simeq 0$ **then** 0
   else zero-bit-vector ($x - 1$) **endif**)

THEOREM: highest-bit2-helper-highest-bit
(bit-vectorp (bv, *word-size*) $\wedge$ (*word-size* $\not\simeq$ 0))
$\rightarrow$ (highest-bit2-helper (zero-bit-vector (*word-size*),
  one-bit-vector (*word-size*),
  bv)
 $=$ highest-bit (bv))

DEFINITION:
highest-bit-input-conditionp ($p0$)
$=$ ((car (top (p-temp-stk ($p0$))) $=$ 'bitv)
 $\wedge$ (cddr (top (p-temp-stk ($p0$))) $=$ **nil**)
 $\wedge$ (p-word-size ($p0$) $\not\simeq$ 0)
 $\wedge$ listp (p-ctrl-stk ($p0$))
 $\wedge$ at-least-morep (p-ctrl-stk-size (p-ctrl-stk ($p0$)),
  6,
  p-max-ctrl-stk-size ($p0$))
 $\wedge$ at-least-morep (length (p-temp-stk ($p0$)),
  2,
  p-max-temp-stk-size ($p0$))
 $\wedge$ (definition ('highest-bit, p-prog-segment ($p0$))
  $=$ HIGHEST-BIT-PROGRAM)
 $\wedge$ (definition ('push-1-vector, p-prog-segment ($p0$))
  $=$ push-1-vector-program (p-word-size ($p0$)))
 $\wedge$ bit-vectorp (cadr (top (p-temp-stk ($p0$))), p-word-size ($p0$)))

DEFINITION:
highest-bit-clock (bv)
$=$ (6 $+$ highest-bit-loop-clock (one-bit-vector (length (bv)), bv))

THEOREM: cons-0-zero-bit-vector
cons (0, zero-bit-vector (x)) $=$ zero-bit-vector ($1 + x$)

EVENT: Disable cons-0-zero-bit-vector.

THEOREM: correctness-of-highest-bit

$$
\begin{aligned}
&((p0 = \text{p-state}\,(pc,\\
&\qquad\qquad ctrl\text{-}stk,\\
&\qquad\qquad \text{cons}\,(b,\ temp\text{-}stk),\\
&\qquad\qquad prog\text{-}segment,\\
&\qquad\qquad data\text{-}segment,\\
&\qquad\qquad max\text{-}ctrl\text{-}stk\text{-}size,\\
&\qquad\qquad max\text{-}temp\text{-}stk\text{-}size,\\
&\qquad\qquad word\text{-}size,\\
&\qquad\qquad \texttt{'run}))\\
&\ \land\ (\text{p-current-instruction}\,(p0) = \texttt{'(call highest-bit)})\\
&\ \land\ (bc = \text{cadr}\,(b))\\
&\ \land\ \text{highest-bit-input-conditionp}\,(p0))\\
&\to\ (\text{p}\,(\text{p-state}\,(pc,\\
&\qquad\qquad ctrl\text{-}stk,\\
&\qquad\qquad \text{cons}\,(b,\ temp\text{-}stk),\\
&\qquad\qquad prog\text{-}segment,\\
&\qquad\qquad data\text{-}segment,\\
&\qquad\qquad max\text{-}ctrl\text{-}stk\text{-}size,\\
&\qquad\qquad max\text{-}temp\text{-}stk\text{-}size,\\
&\qquad\qquad word\text{-}size,\\
&\qquad\qquad \texttt{'run}),\\
&\qquad \text{highest-bit-clock}\,(bc))\\
&\quad = \ \text{p-state}\,(\text{add1-addr}\,(pc),\\
&\qquad\qquad ctrl\text{-}stk,\\
&\qquad\qquad \text{cons}\,(\text{list}\,(\texttt{'bitv},\ \text{highest-bit}\,(\text{cadr}\,(b))),\ temp\text{-}stk),\\
&\qquad\qquad prog\text{-}segment,\\
&\qquad\qquad data\text{-}segment,\\
&\qquad\qquad max\text{-}ctrl\text{-}stk\text{-}size,\\
&\qquad\qquad max\text{-}temp\text{-}stk\text{-}size,\\
&\qquad\qquad word\text{-}size,\\
&\qquad\qquad \texttt{'run}))
\end{aligned}
$$

```
;; match-and-xor
```

DEFINITION:
MATCH-AND-XOR-PROGRAM

```
=  '(match-and-xor
     (vecs numvecs match xor-vector)
     ((i (nat 0)))
     (push-constant (nat 0))
```

```
      (pop-local i)
      (dl loop nil (push-local vecs))
      (fetch)
      (push-local match)
      (and-bitv)
      (test-bitv-and-jump not-all-zeros found)
      (push-local i)
      (add1-nat)
      (set-local i)
      (push-local numvecs)
      (lt-nat)
      (test-bool-and-jump f done)
      (push-local vecs)
      (push-constant (nat 1))
      (add-addr)
      (pop-local vecs)
      (jump loop)
      (dl found nil (push-local vecs))
      (fetch)
      (push-local xor-vector)
      (xor-bitv)
      (push-local vecs)
      (deposit)
      (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-MATCH-AND-XOR-P-STATE

```
=  p-state ('(pc (main . 0)),
            '((nil (pc (main . 0)))),
            nil,
            list ('(main nil nil
                     (push-constant (addr (arr . 0)))
                     (push-constant (nat 6))
                     (push-constant (bitv (0 0 0 1 0 0 0 0)))
                     (push-constant (bitv (1 1 1 1 1 1 1 1)))
                     (call match-and-xor)
                     (ret)),
                  MATCH-AND-XOR-PROGRAM),
            '((arr
               (bitv (0 1 0 0 1 0 0 1))
               (bitv (0 0 0 0 0 0 0 1))
               (bitv (0 0 0 1 0 0 0 1))
               (bitv (0 0 0 0 0 0 0 1))
               (bitv (0 0 0 1 0 0 0 1))
```

```
            (bitv (0 1 1 0 1 0 0 1)))),
        10,
        8,
        8,
        'run)
```

DEFINITION:
match-and-xor (*bvs*, *match*, *xor-vector*)
= **if** listp (*bvs*)
  then if all-zero-bitvp (and-bitv (car (*bvs*), *match*))
    then cons (car (*bvs*), match-and-xor (cdr (*bvs*), *match*, *xor-vector*))
    else cons (xor-bitv (car (*bvs*), *xor-vector*), cdr (*bvs*)) **endif**
  else nil endif

DEFINITION:
match-and-xor-loop-clock (*bvs*, *match*)
= **if** listp (*bvs*)
  then if all-zero-bitvp (and-bitv (car (*bvs*), *match*))
    then if listp (cdr (*bvs*))
      then `16` + match-and-xor-loop-clock (cdr (*bvs*), *match*)
      else `12` **endif**
    else `12` **endif**
  else `0` **endif**

DEFINITION:
tag-array (*tag*, *array*)
= **if** listp (*array*)
  then cons (tag (*tag*, car (*array*)), tag-array (*tag*, cdr (*array*)))
  else nil endif

DEFINITION:
match-and-xor-general-induct (*numvecs*, *i*)
= **if** $i < numvecs$ **then** match-and-xor-general-induct (*numvecs*, $1 + i$)
  else t endif

THEOREM: tag-array-untag-array
bit-vectors-piton (x, length (untag (car (x))))
$\rightarrow$ (tag-array (`'bitv`, untag-array (x)) $= x$)

THEOREM: bit-vectors-piton-free-means
bit-vectors-piton (x, *size*)
$\rightarrow$ (bit-vectors-piton (x, length (cadr (car (x))))
  $\wedge$ (bit-vectors-piton (cdr (x), length (cadr (cadr (x)))) = listp (x)))

THEOREM: listp-cdr-nthcdr
listp (cdr (nthcdr (i, x))) = ($i <$ length (cdr (x)))

THEOREM: nthcdr-untag-array
$\text{nthcdr}\,(i,\ \text{untag-array}\,(x))$
$= \ \ \textbf{if}\ i < \text{length}\,(x)\ \ \textbf{then}\ \text{untag-array}\,(\text{nthcdr}\,(i,\ x))$
$\quad \textbf{elseif}\ \text{fix}\,(i) = \text{length}\,(x)\ \ \textbf{then nil}$
$\quad \textbf{else}\ \texttt{0}\ \textbf{endif}$

THEOREM: put-length-cdr
$\text{properp}\,(x)$
$\rightarrow \ \ (\text{put}\,(val,\ \text{length}\,(\text{cdr}\,(x)),\ x) = \text{append}\,(\text{all-but-last}\,(x),\ \text{list}\,(val)))$

THEOREM: nthcdr-length-cdr
$\text{properp}\,(x)$
$\rightarrow \ \ (\text{nthcdr}\,(\text{length}\,(\text{cdr}\,(x)),\ x)$
$\quad = \ \ \textbf{if}\ \text{listp}\,(x)\ \ \textbf{then}\ \text{list}\,(\text{last}\,(x))$
$\quad\quad \textbf{else nil endif})$

THEOREM: get-length-cdr
$\text{get}\,(\text{length}\,(\text{cdr}\,(x)),\ x)$
$= \ \ \textbf{if}\ \text{listp}\,(x)\ \ \textbf{then}\ \text{last}\,(x)$
$\quad \textbf{else}\ \texttt{0}\ \textbf{endif}$

THEOREM: append-all-but-last-last
$\text{properp}\,(x)$
$\rightarrow \ \ (\text{append}\,(\text{all-but-last}\,(x),\ \text{list}\,(\text{last}\,(x)))$
$\quad = \ \ \textbf{if}\ x \simeq \textbf{nil}\ \ \textbf{then}\ \text{list}\,(\text{last}\,(x))$
$\quad\quad \textbf{else}\ x\ \textbf{endif})$

THEOREM: listp-cdr-untag-array
$\text{listp}\,(\text{cdr}\,(\text{untag-array}\,(x))) = \text{listp}\,(\text{cdr}\,(x))$

THEOREM: bit-vectorp-last
$\text{bit-vectors-piton}\,(bvs,\ s)$
$\rightarrow \ \ (\text{bit-vectorp}\,(\text{cadr}\,(\text{last}\,(bvs)),\ s) = \text{listp}\,(bvs))$

THEOREM: bit-vectors-piton-means-properp
$\text{bit-vectors-piton}\,(x,\ s) \rightarrow \text{properp}\,(x)$

THEOREM: bit-vectors-piton-means-last
$(\text{bit-vectors-piton}\,(state,\ size) \land \text{listp}\,(state))$
$\rightarrow \ \ ((\text{car}\,(\text{last}\,(state)) = \texttt{'bitv})$
$\quad \land \ \ \text{listp}\,(\text{last}\,(state))$
$\quad \land \ \ \text{bit-vectorp}\,(\text{cadr}\,(\text{last}\,(state)),\ size)$
$\quad \land \ \ (\text{cddr}\,(\text{last}\,(state)) = \textbf{nil})$
$\quad \land \ \ (\text{length}\,(\text{cadr}\,(\text{last}\,(state))) = \text{fix}\,(size)))$

THEOREM: list-bitv-cadr-bitvp
$((\text{car}\,(x) = \texttt{'bitv}) \land (\text{cddr}\,(x) = \textbf{nil}))$
$\rightarrow \quad (\text{list}\,(\texttt{'bitv}, \text{cadr}\,(x)) = x)$

THEOREM: equal-x-put-assoc-x
$(x = \text{put-assoc}\,(val,\ name,\ x))$
$= \quad ((\neg\ \text{definedp}\,(name,\ x)) \lor (\text{assoc}\,(name,\ x) = \text{cons}\,(name,\ val)))$

THEOREM: cons-n-assoc-n
$\text{listp}\,(\text{assoc}\,(n,\ d))$
$\rightarrow \quad (\text{cons}\,(n, \text{cdr}\,(\text{assoc}\,(n,\ d)))$
$\qquad = \quad \textbf{if } \text{definedp}\,(n,\ d) \ \textbf{ then } \text{assoc}\,(n,\ d)$
$\qquad\qquad \textbf{else } \text{cons}\,(n, \texttt{0}) \textbf{ endif})$

THEOREM: cons-n-cadr-list-assoc-n
$(\text{cddr}\,(\text{assoc}\,(n,\ d)) = \textbf{nil})$
$\rightarrow \quad (\text{list}\,(n, \text{cadr}\,(\text{assoc}\,(n,\ d)))$
$\qquad = \quad \textbf{if } \text{definedp}\,(n,\ d) \ \textbf{ then } \text{assoc}\,(n,\ d)$
$\qquad\qquad \textbf{else } \text{cons}\,(n, \texttt{0}) \textbf{ endif})$

THEOREM: cons-n-assoc-n-hack
$\text{listp}\,(\text{cdr}\,(\text{assoc}\,(n,\ d)))$
$\rightarrow \quad (\text{cons}\,(n, \text{cdr}\,(\text{assoc}\,(n,\ d)))$
$\qquad = \quad \textbf{if } \text{definedp}\,(n,\ d) \ \textbf{ then } \text{assoc}\,(n,\ d)$
$\qquad\qquad \textbf{else } \text{cons}\,(n, \texttt{0}) \textbf{ endif})$

THEOREM: car-untag-array
$\text{car}\,(\text{untag-array}\,(x)) = \text{untag}\,(\text{car}\,(x))$

THEOREM: car-nthcdr
$\text{car}\,(\text{nthcdr}\,(i,\ x)) = \text{get}\,(i,\ x)$

THEOREM: tag-array-cdr-untag-array-hack
$\text{bit-vectors-piton}\,(x, \text{length}\,(\text{untag}\,(\text{car}\,(x))))$
$\rightarrow \quad (\text{tag-array}\,(\texttt{'bitv}, \text{cdr}\,(\text{untag-array}\,(x)))$
$\qquad = \quad \textbf{if } \text{listp}\,(x) \ \textbf{ then } \text{cdr}\,(x)$
$\qquad\qquad \textbf{else nil endif})$

THEOREM: bit-vectors-piton-means-car
$(\text{bit-vectors-piton}\,(state,\ size) \land \text{listp}\,(state))$
$\rightarrow \quad ((\text{caar}\,(state) = \texttt{'bitv})$
$\qquad \land \quad \text{listp}\,(\text{car}\,(state))$
$\qquad \land \quad \text{bit-vectorp}\,(\text{cadr}\,(\text{car}\,(state)),\ size)$
$\qquad \land \quad (\text{cddr}\,(\text{car}\,(state)) = \textbf{nil})$
$\qquad \land \quad (\text{length}\,(\text{cadr}\,(\text{car}\,(state))) = \text{fix}\,(size)))$

THEOREM: equal-put-assoc
$(\text{put-assoc}\,(v1,\ n,\ s) = \text{put-assoc}\,(v2,\ n,\ s))$
$= \quad ((\neg\ \text{definedp}\,(n,\ s)) \lor (v1 = v2))$

THEOREM: get-nlistp-better
$(i \not< \text{length}\,(x)) \rightarrow (\text{get}\,(i,\ x) = \mathtt{0})$

THEOREM: equal-append-zero-bit-vector-zero-bit-vector
$(\text{append}\,(\text{zero-bit-vector}\,(n1),\ x) = \text{append}\,(\text{zero-bit-vector}\,(n2),\ y))$
$= \quad$ **if** $n1 < n2$ **then** $x = \text{append}\,(\text{zero-bit-vector}\,(n2 - n1),\ y)$
else $y = \text{append}\,(\text{zero-bit-vector}\,(n1 - n2),\ x)$ **endif**

THEOREM: append-make-list-from-cons-get-hack
$\text{append}\,(\text{make-list-from}\,(i,\ x),\ \text{cons}\,(\text{get}\,(i,\ x),\ y))$
$= \quad \text{append}\,(\text{make-list-from}\,(1 + i,\ x),\ y)$

THEOREM: cdr-untag-array-nthcdr
$(i < \text{length}\,(x))$
$\rightarrow \quad (\text{cdr}\,(\text{untag-array}\,(\text{nthcdr}\,(i,\ x))) = \text{untag-array}\,(\text{nthcdr}\,(i,\ \text{cdr}\,(x))))$

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

THEOREM: lessp-0-length-better
$(x \simeq \mathtt{0}) \rightarrow ((x < \text{length}\,(y)) = \text{listp}\,(y))$

THEOREM: bit-vectors-piton-means-get-cdr
$\text{bit-vectors-piton}\,(\mathit{state},\ \mathit{size})$
$\rightarrow \quad ((\text{car}\,(\text{get}\,(i,\ \text{cdr}\,(\mathit{state})))$
$= \quad$ **if** $i < \text{length}\,(\text{cdr}\,(\mathit{state}))$ **then** `'bitv`
else `0` **endif**)
$\land \quad (\text{listp}\,(\text{get}\,(i,\ \text{cdr}\,(\mathit{state}))) = (i < \text{length}\,(\text{cdr}\,(\mathit{state}))))$
$\land \quad (\text{bit-vectorp}\,(\text{cadr}\,(\text{get}\,(i,\ \text{cdr}\,(\mathit{state}))),\ \mathit{size})$
$= \quad (i < \text{length}\,(\text{cdr}\,(\mathit{state}))))$
$\land \quad (\text{cddr}\,(\text{get}\,(i,\ \text{cdr}\,(\mathit{state})))$
$= \quad$ **if** $i < \text{length}\,(\text{cdr}\,(\mathit{state}))$ **then nil**
else `0` **endif**)
$\land \quad (\text{length}\,(\text{cadr}\,(\text{get}\,(i,\ \text{cdr}\,(\mathit{state}))))$
$= \quad$ **if** $i < \text{length}\,(\text{cdr}\,(\mathit{state}))$ **then** $\text{fix}\,(\mathit{size})$
else `0` **endif**))

THEOREM: bit-vectors-piton-nthcdr
$\text{bit-vectors-piton}\,(x,\ s1)$
$\rightarrow \quad ((\text{bit-vectors-piton}\,(\text{nthcdr}\,(i,\ x),\ s)$
$= \quad (((i < \text{length}\,(x)) \land (\text{fix}\,(s1) = \text{fix}\,(s)))$

$\lor \quad (\text{fix}\,(i) = \text{length}\,(x))))$
$\land \quad (\text{bit-vectors-piton}\,(\text{nthcdr}\,(i,\ \text{cdr}\,(x)),\ s)$
$= \quad ((i < \text{length}\,(x))$
$\land \quad ((\text{fix}\,(s) = \text{fix}\,(s1))$
$\lor \quad (\text{fix}\,(i) = \text{length}\,(\text{cdr}\,(x)))))))$

THEOREM: tag-array-untag-array-nthcdr-cddr-hack
$((i < \text{length}\,(x)) \land \text{bit-vectors-piton}\,(x,\ s))$
$\rightarrow \quad (\text{tag-array}\,(\texttt{'bitv},\ \text{untag-array}\,(\text{nthcdr}\,(i,\ \text{cdr}\,(x))))$
$= \quad \text{nthcdr}\,(i,\ \text{cdr}\,(x)))$

THEOREM: append-make-list-from-put-hack
$(i < \text{length}\,(x))$
$\rightarrow \quad (\text{append}\,(\text{make-list-from}\,(i,\ x),\ \text{cons}\,(v,\ \text{nthcdr}\,(i,\ \text{cdr}\,(x)))) = \text{put}\,(v,\ i,\ x))$

EVENT: Enable lessp-sub1-x-x.

EVENT: Enable lessp-x-x.

THEOREM: correctness-of-match-and-xor-general
(listp (*ctrl-stk*)
∧ at-least-morep (p-ctrl-stk-size (*ctrl-stk*), `7`, *max-ctrl-stk-size*)
∧ at-least-morep (length (*temp-stk*), `4`, *max-temp-stk-size*)
∧ (definition (`'match-and-xor`, *prog-segment*)
= MATCH-AND-XOR-PROGRAM)
∧ bit-vectorp (*match*, *word-size*)
∧ bit-vectorp (*xor-vector*, *word-size*)
∧ (*numvecs* = length (array (*vecs*, *data-segment*)))
∧ bit-vectors-piton (array (*vecs*, *data-segment*), *word-size*)
∧ definedp (*vecs*, *data-segment*)
∧ $(i \in \mathbf{N})$
∧ $(i < numvecs)$
∧ (length (array (*vecs*, *data-segment*)) < exp (`2`, *word-size*)))
→ (p (p-state (`'(pc (match-and-xor . 2))`,
cons (list (list (cons (`'vecs`, list (`'addr`, cons (*vecs*, *i*))),
cons (`'numvecs`, list (`'nat`, *numvecs*)),
cons (`'match`, list (`'bitv`, *match*)),
cons (`'xor-vector`,
list (`'bitv`, *xor-vector*)),
cons (`'i`, list (`'nat`, *i*))),
ret-pc),
ctrl-stk),
temp-stk,
prog-segment,

data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
match-and-xor-loop-clock (nthcdr (i,
untag-array (array (*vecs*,
data-segment))),
match))

= p-state (*ret-pc*,
ctrl-stk,
temp-stk,
prog-segment,
put-assoc (append (make-list-from (i,
array (*vecs*,
data-segment)),
tag-array (`'bitv`,
match-and-xor (untag-array (nthcdr (i,
array (*vecs*,
data-segment))),
match,
xor-vector))),
vecs,
data-segment),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

DEFINITION:
match-and-xor-clock (*bvs*, *match*)
= (`3` + match-and-xor-loop-clock (*bvs*, *match*))

DEFINITION:
match-and-xor-input-conditionp (*p0*)
= ((car (top (p-temp-stk (*p0*))) = `'bitv`)
∧ (cddr (top (p-temp-stk (*p0*))) = **nil**)
∧ (car (top (cdr (p-temp-stk (*p0*)))) = `'bitv`)
∧ (cddr (top (cdr (p-temp-stk (*p0*)))) = **nil**)
∧ (car (top (cddr (p-temp-stk (*p0*)))) = `'nat`)
∧ (cddr (top (cddr (p-temp-stk (*p0*)))) = **nil**)
∧ (car (top (cdddr (p-temp-stk (*p0*)))) = `'addr`)
∧ (cdr (cadr (top (cdddr (p-temp-stk (*p0*))))) = `0`)
∧ listp (cadr (top (cdddr (p-temp-stk (*p0*))))))

$\wedge$ (cddr (top (cdddr (p-temp-stk ($p0$)))) = **nil**)
$\wedge$ (cadr (top (cddr (p-temp-stk ($p0$)))) $\not\simeq$ `0`)
$\wedge$ listp (p-ctrl-stk ($p0$))
$\wedge$ at-least-morep (p-ctrl-stk-size (p-ctrl-stk ($p0$)),
`7`,
p-max-ctrl-stk-size ($p0$))
$\wedge$ at-least-morep (length (p-temp-stk ($p0$)),
`0`,
p-max-temp-stk-size ($p0$))
$\wedge$ (definition (`'match-and-xor`, p-prog-segment ($p0$))
= MATCH-AND-XOR-PROGRAM)
$\wedge$ bit-vectorp (cadr (top (cdr (p-temp-stk ($p0$)))), p-word-size ($p0$))
$\wedge$ bit-vectorp (cadr (top (p-temp-stk ($p0$))), p-word-size ($p0$))
$\wedge$ (cadr (top (cddr (p-temp-stk ($p0$))))
= length (array (car (cadr (top (cdddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$))))
$\wedge$ definedp (car (cadr (top (cdddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$))
$\wedge$ (length (array (car (cadr (top (cdddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$)))
$<$ exp (`2`, p-word-size ($p0$)))
$\wedge$ bit-vectors-piton (array (car (cadr (top (cdddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$)),
p-word-size ($p0$)))

THEOREM: correctness-of-match-and-xor
(($p0$ = p-state (pc,
ctrl-stk,
cons (x, cons (m, cons (n, cons (v, *temp-stk*)))),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))
$\wedge$ (p-current-instruction ($p0$) = `'(call match-and-xor)`)
$\wedge$ match-and-xor-input-conditionp ($p0$)
$\wedge$ (*match* = cadr (m))
$\wedge$ (*vecs-to-match* = untag-array (array (caadr (v), *data-segment*)))
$\rightarrow$ (p (p-state (pc,
ctrl-stk,
cons (x, cons (m, cons (n, cons (v, *temp-stk*)))),
prog-segment,
data-segment,

max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
match-and-xor-clock (*vecs-to-match*, *match*))
= p-state (add1-addr (*pc*),
ctrl-stk,
temp-stk,
prog-segment,
put-assoc (tag-array (`'bitv`,
match-and-xor (untag-array (array (caadr (v),
data-segment)),
cadr (m),
cadr (x))),
caadr (v),
data-segment),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

```
;;; nat-to-bv-list
```

DEFINITION:
NAT-TO-BV-LIST-PROGRAM

```
=  '(nat-to-bv-list
     (nat-list bv-list length)
     ((i (nat 0)))
     (dl loop nil (push-local nat-list))
     (fetch)
     (call nat-to-bv)
     (push-local bv-list)
     (deposit)
     (push-local i)
     (add1-nat)
     (set-local i)
     (push-local length)
     (eq)
     (test-bool-and-jump t done)
     (push-local nat-list)
     (push-constant (nat 1))
     (add-addr)
     (pop-local nat-list)
```

```
      (push-local bv-list)
      (push-constant (nat 1))
      (add-addr)
      (pop-local bv-list)
      (jump loop)
      (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-NAT-TO-BV-LIST-P-STATE

```
=  p-state ('(pc (main . 0)),
           '((nil (pc (main . 0)))),
           nil,
           list ('(main nil nil
                  (push-constant (addr (arr . 0)))
                  (push-constant (addr (arr2 . 0)))
                  (push-constant (nat 6))
                  (call nat-to-bv-list)
                  (ret)),
                NAT-TO-BV-LIST-PROGRAM,
                NAT-TO-BV-PROGRAM,
                push-1-vector-program (8)),
           '((arr
               (nat 3)
               (nat 5)
               (nat 8)
               (nat 0)
               (nat 23)
               (nat 9))
             (arr2 nil nil nil nil nil nil)),
           100,
           80,
           8,
           'run)
```

DEFINITION:
nat-to-bv-list (*nat-list*, *size*)
= **if** listp (*nat-list*)
  then cons (nat-to-bv (car (*nat-list*), *size*),
             nat-to-bv-list (cdr (*nat-list*), *size*))
  else nil endif

DEFINITION:
nat-to-bv-list-loop-clock (i, *nats*)
= **if** $i <$ length (*nats*)
  then clock-plus ('2,

```
              clock-plus (nat-to-bv-clock (untag (get (i, nats))),
                          if i < length (cdr (nats))
                          then clock-plus ('17,
                                           nat-to-bv-list-loop-clock (1 + i,
                                                                      nats))
                          else '9 endif))
    else '0 endif
```

DEFINITION:

```
nat-to-bv-list-loop-induct (i, nats, bvs-name, word-size, data-segment)
=   if (i < length (nats)) ∧ (i < length (cdr (nats)))
    then nat-to-bv-list-loop-induct (1 + i,
                                     nats,
                                     bvs-name,
                                     word-size,
                                     put-assoc (append (make-list-from (i,
                                                                        cdr (assoc (bvs-name,
                                                                                    data-segment))),
                                                        cons (list ('bitv,
                                                                    nat-to-bv (untag (get (i,
                                                                                           nats)),
                                                                               word-size)),
                                                              nthcdr (1 + i,
                                                                      cdr (assoc (bvs-name,
                                                                                  data-segment))))),
                                                bvs-name,
                                                data-segment))
    else t endif
```

THEOREM: nat-list-piton-means-car

```
 (nat-list-piton (state, size) ∧ listp (state))
 →   ((caar (state) = 'nat)
      ∧  listp (car (state))
      ∧  (cadr (car (state)) ∈ N)
      ∧  (cadr (car (state)) < exp (2, size))
      ∧  ((cadr (car (state)) < exp (2, size)) = t)
      ∧  (cddr (car (state)) = nil))
```

DEFINITION:

```
array-pitonp (array, length)
=   if listp (array)
    then (length ≄ 0) ∧ array-pitonp (cdr (array), length − 1)
    else (array = nil) ∧ (length ≃ 0) endif
```

THEOREM: nat-list-piton-means-last

nat-list-piton (*state*, *size*)
$\rightarrow$ ((car (last (*state*))
  = **if** listp (*state*) **then** `'nat`
  else `0` **endif**)
 $\wedge$ (listp (last (*state*)) = listp (*state*))
 $\wedge$ (cadr (last (*state*)) $\in$ **N**)
 $\wedge$ ((cadr (last (*state*)) < exp (`2`, *size*)) = **t**)
 $\wedge$ (cddr (last (*state*))
  = **if** listp (*state*) **then nil**
  else `0` **endif**))

THEOREM: nat-list-piton-means-last-cdr
(nat-list-piton (*state*, *size*) $\wedge$ listp (*state*))
$\rightarrow$ ((car (last (cdr (*state*)))
  = **if** listp (cdr (*state*)) **then** `'nat`
  else `0` **endif**)
 $\wedge$ (listp (last (cdr (*state*))) = listp (cdr (*state*)))
 $\wedge$ (cadr (last (cdr (*state*))) $\in$ **N**)
 $\wedge$ ((cadr (last (cdr (*state*))) < exp (`2`, *size*)) = **t**)
 $\wedge$ (cddr (last (cdr (*state*)))
  = **if** listp (cdr (*state*)) **then nil**
  else `0` **endif**))

```
;(prove-lemma definedp-put-assoc (rewrite)
;      (equal
;       (definedp n (put-assoc v n1 a))
;       (definedp n a)))
```

EVENT: Enable definedp-put-assoc.

THEOREM: assoc-put-assoc-better
assoc (*n1*, put-assoc (*v*, *n2*, *a*))
= **if** definedp (*n1*, *a*)
 then if *n1* = *n2* **then** cons (*n1*, *v*)
  else assoc (*n1*, *a*) **endif**
 else f endif

EVENT: Disable nat-to-bv-list-loop-clock.

THEOREM: get-add1
(n < `4`) $\rightarrow$ (get ($1 + n$, x) = get (n, cdr (x)))

THEOREM: get-0-better
($n \simeq$ `0`) $\rightarrow$ (get (n, x) = car (x))

THEOREM: length-cdr-tag-array
$\text{length}\,(\text{cdr}\,(\text{tag-array}\,(l,\ x))) = \text{length}\,(\text{cdr}\,(x))$

THEOREM: length-nat-to-bv-list
$\text{length}\,(\text{nat-to-bv-list}\,(x,\ s)) = \text{length}\,(x)$

THEOREM: length-cdr-nat-to-bv-list
$\text{length}\,(\text{cdr}\,(\text{nat-to-bv-list}\,(x,\ s))) = \text{length}\,(\text{cdr}\,(x))$

THEOREM: length-tag-array
$\text{length}\,(\text{tag-array}\,(l,\ x)) = \text{length}\,(x)$

THEOREM: listp-tag-array
$\text{listp}\,(\text{tag-array}\,(l,\ x)) = \text{listp}\,(x)$

THEOREM: listp-nat-to-bv-list
$\text{listp}\,(\text{nat-to-bv-list}\,(x,\ s)) = \text{listp}\,(x)$

THEOREM: array-pitonp-tag-array
$\text{array-pitonp}\,(\text{tag-array}\,(l,\ x),\ \mathit{length}) = (\text{length}\,(x) = \text{fix}\,(\mathit{length}))$

THEOREM: array-pitonp-append
$\text{array-pitonp}\,(\text{append}\,(x,\ y),\ \mathit{size})$
$= \quad ((\mathit{size} \not< \text{length}\,(x)) \land \text{array-pitonp}\,(y,\ \mathit{size} - \text{length}\,(x)))$

THEOREM: nat-list-piton-means-cadr
$(\text{nat-list-piton}\,(\mathit{state},\ \mathit{size}) \land \text{listp}\,(\text{cdr}\,(\mathit{state})))$
$\rightarrow \quad ((\text{caadr}\,(\mathit{state}) = \texttt{'nat})$
$\quad \land \quad \text{listp}\,(\text{cadr}\,(\mathit{state}))$
$\quad \land \quad (\text{cadr}\,(\text{cadr}\,(\mathit{state})) \in \mathbf{N})$
$\quad \land \quad (\text{cadr}\,(\text{cadr}\,(\mathit{state})) < \text{exp}\,(2,\ \mathit{size}))$
$\quad \land \quad ((\text{cadr}\,(\text{cadr}\,(\mathit{state})) < \text{exp}\,(2,\ \mathit{size})) = \mathbf{t})$
$\quad \land \quad (\text{cddr}\,(\text{cadr}\,(\mathit{state})) = \mathbf{nil}))$

THEOREM: array-pitonp-add1
$\text{array-pitonp}\,(x,\ 1 + n) = (\text{listp}\,(x) \land \text{array-pitonp}\,(\text{cdr}\,(x),\ n))$

```
;(prove-lemma put-assoc-put-assoc (rewrite)
;       (equal
;        (put-assoc v n (put-assoc v2 n a))
;        (put-assoc v n a)))
```

EVENT: Enable put-assoc-put-assoc.

THEOREM: length-cdr-nlistp
$(\neg \text{ listp}(\text{cdr}(x)))$
$\rightarrow \quad (\text{length}(x) = \textbf{if } \text{listp}(x) \textbf{ then } 1 \textbf{ else } 0 \textbf{ endif})$

THEOREM: equal-nil-cdr-tag-array-hack
$(\textbf{nil} = \text{cdr}(\text{tag-array}(l, x))) = (\text{length}(x) = 1)$

THEOREM: length-from-array-pitonp
$\text{array-pitonp}(x, s) \rightarrow (\text{length}(x) = \text{fix}(s))$

EVENT: Disable nat-to-bv-clock.

EVENT: Disable nat-to-bv-list-loop-clock.

THEOREM: nat-list-piton-means-get
$\text{nat-list-piton}(state, size)$
$\rightarrow \quad ((\text{car}(\text{get}(p, state)) = \textbf{if } p < \text{length}(state) \textbf{ then } \texttt{'nat} \textbf{ else } 0 \textbf{ endif})$
$\wedge \quad (\text{listp}(\text{get}(p, state)) = (p < \text{length}(state)))$
$\wedge \quad (\text{cadr}(\text{get}(p, state)) \in \mathbf{N})$
$\wedge \quad ((\text{cadr}(\text{get}(p, state)) < \exp(2, size)) = \mathbf{t})$
$\wedge \quad (\text{cddr}(\text{get}(p, state)) = \textbf{if } p < \text{length}(state) \textbf{ then nil else } 0 \textbf{ endif}))$

THEOREM: nat-list-piton-means-get-cdr
$(\text{nat-list-piton}(state, size) \wedge \text{listp}(state))$
$\rightarrow \quad ((\text{car}(\text{get}(p, \text{cdr}(state))) = \textbf{if } p < \text{length}(\text{cdr}(state)) \textbf{ then } \texttt{'nat} \textbf{ else } 0 \textbf{ endif})$
$\wedge \quad (\text{listp}(\text{get}(p, \text{cdr}(state))) = (p < \text{length}(\text{cdr}(state))))$
$\wedge \quad (\text{cadr}(\text{get}(p, \text{cdr}(state))) \in \mathbf{N})$
$\wedge \quad ((\text{cadr}(\text{get}(p, \text{cdr}(state))) < \exp(2, size)) = \mathbf{t})$
$\wedge \quad (\text{cddr}(\text{get}(p, \text{cdr}(state))) = \textbf{if } p < \text{length}(\text{cdr}(state)) \textbf{ then nil else } 0 \textbf{ endif}))$

EVENT: Disable nat-list-piton-means.

THEOREM: length-put-better

length (put (v, n, a))
$=$ **if** $n <$ length (a) **then** length (a)
else $1 + n$ **endif**

THEOREM: cons-car-x-put-append-make-list-from-hack
cons (car (put (v, x, b)), append (make-list-from (x, cdr (put (v, x, b))), y))
$=$ append (make-list-from (x, b), cons (v, y))

THEOREM: array-pitonp-put
(array-pitonp (a, l) $\land$ (fix (l) $=$ fix (*length*)))
$\rightarrow$ (array-pitonp (put (v, n, a), *length*) $=$ ($n <$ length (a)))

THEOREM: array-pitonp-means-properp
(array-pitonp (x, s) $\rightarrow$ properp (x))
$\land$ (array-pitonp (cdr (x), s) $\rightarrow$ properp (x))

THEOREM: put-length-cdr-general
(properp (x) $\land$ (length (x) $=$ length (y)))
$\rightarrow$ (put (*val*, length (cdr (y)), x) $=$ append (all-but-last (x), list (*val*)))

THEOREM: nthcdr-x-cdr-put-x
properp (a)
$\rightarrow$ (nthcdr (x, cdr (put (*val*, x, a)))
$=$ **if** $x <$ length (a) **then** nthcdr (x, cdr (a))
else nil endif)

THEOREM: equal-append-a-append-a
(append (a, b) $=$ append (a, c)) $=$ ($b = c$)

```
;The correctness lemma of nat-to-bv-list could be more general -
;the proof assumes the data areas are distinct, though the program
;works when they're not.  I did it this way because I had assumed
;I'd need distinct arrays in NIM, and designed the proof accordingly.
;This is a weakness I should correct in this proof as it will lead
;to sloppy use of memory in the program, but it takes so long
;to do these proofs I'll wait.
```

THEOREM: correctness-of-nat-to-bv-list-general
(listp (*ctrl-stk*)
$\land$ at-least-morep (p-ctrl-stk-size (*ctrl-stk*), `13`, *max-ctrl-stk-size*)
$\land$ at-least-morep (length (*temp-stk*), `3`, *max-temp-stk-size*)
$\land$ (definition (`'nat-to-bv-list`, *prog-segment*)
$=$ NAT-TO-BV-LIST-PROGRAM)

$\land$ (definition (`'nat-to-bv`, *prog-segment*) = NAT-TO-BV-PROGRAM)
$\land$ (definition (`'push-1-vector`, *prog-segment*)
  = push-1-vector-program (*word-size*))
$\land$ (*length* = length (array (*nats*, *data-segment*)))
$\land$ array-pitonp (array (*bvs*, *data-segment*), *length*)
$\land$ nat-list-piton (array (*nats*, *data-segment*), *word-size*)
$\land$ definedp (*nats*, *data-segment*)
$\land$ definedp (*bvs*, *data-segment*)
$\land$ (*word-size* $\not\simeq$ `0`)
$\land$ ($i \in \mathbf{N}$)
$\land$ ($i <$ *length*)
$\land$ (*length* $<$ exp (`2`, *word-size*))
$\land$ (*nats* $\neq$ *bvs*)
$\land$ (*natlist* = array (*nats*, *data-segment*)))
$\rightarrow$ (p (p-state (`'(pc (nat-to-bv-list . 0))`,
      cons (list (list (cons (`'nat-list`,
            list (`'addr`, cons (*nats*, *i*))),
          cons (`'bv-list`, list (`'addr`, cons (*bvs*, *i*))),
          cons (`'length`, list (`'nat`, *length*)),
          cons (`'i`, list (`'nat`, *i*))),
        ret-pc),
      ctrl-stk),
      temp-stk,
      prog-segment,
      data-segment,
      max-ctrl-stk-size,
      max-temp-stk-size,
      word-size,
      `'run`),
  nat-to-bv-list-loop-clock (*i*, *natlist*))
  = p-state (*ret-pc*,
      ctrl-stk,
      temp-stk,
      prog-segment,
      put-assoc (append (make-list-from (*i*,
            array (*bvs*,
              data-segment)),
          tag-array (`'bitv`,
            nat-to-bv-list (untag-array (nthcdr (*i*,
                array (*nats*,
                  data-segment))),
              word-size))),
        bvs,
        data-segment),

max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

DEFINITION:
nat-to-bv-list-clock (*natlist*)
= clock-plus (`1`, nat-to-bv-list-loop-clock (`0`, *natlist*))

DEFINITION:
nat-to-bv-list-input-conditionp (*p0*)
= (listp (p-ctrl-stk (*p0*))
∧ at-least-morep (p-ctrl-stk-size (p-ctrl-stk (*p0*)),
`13`,
p-max-ctrl-stk-size (*p0*))
∧ at-least-morep (length (p-temp-stk (*p0*)),
`0`,
p-max-temp-stk-size (*p0*))
∧ (definition (`'nat-to-bv-list`, p-prog-segment (*p0*))
= NAT-TO-BV-LIST-PROGRAM)
∧ (definition (`'nat-to-bv`, p-prog-segment (*p0*))
= NAT-TO-BV-PROGRAM)
∧ (definition (`'push-1-vector`, p-prog-segment (*p0*))
= push-1-vector-program (p-word-size (*p0*)))
∧ (cadr (top (p-temp-stk (*p0*)))
= length (array (caadr (top (cddr (p-temp-stk (*p0*)))),
p-data-segment (*p0*))))
∧ array-pitonp (array (caadr (top (cdr (p-temp-stk (*p0*)))),
p-data-segment (*p0*)),
cadr (top (p-temp-stk (*p0*))))
∧ nat-list-piton (array (caadr (top (cddr (p-temp-stk (*p0*)))),
p-data-segment (*p0*)),
p-word-size (*p0*))
∧ definedp (caadr (top (cddr (p-temp-stk (*p0*)))), p-data-segment (*p0*))
∧ definedp (caadr (top (cdr (p-temp-stk (*p0*)))), p-data-segment (*p0*))
∧ (p-current-instruction (*p0*) = `'(call nat-to-bv-list)`)
∧ (p-word-size (*p0*) $\not\simeq$ `0`)
∧ (caadr (top (cddr (p-temp-stk (*p0*))))
≠ caadr (top (cdr (p-temp-stk (*p0*)))))
∧ (car (top (p-temp-stk (*p0*))) = `'nat`)
∧ (cadr (top (p-temp-stk (*p0*))) $\not\simeq$ `0`)
∧ (cadr (top (p-temp-stk (*p0*))) < exp (`2`, p-word-size (*p0*)))
∧ (cddr (top (p-temp-stk (*p0*))) = **nil**)
∧ (car (top (cdr (p-temp-stk (*p0*)))) = `'addr`)

$\wedge$ listp (cadr (top (cdr (p-temp-stk ($p0$)))))
$\wedge$ (cdadr (top (cdr (p-temp-stk ($p0$)))) = `0`)
$\wedge$ (cddr (top (cdr (p-temp-stk ($p0$)))) = **nil**)
$\wedge$ (car (top (cddr (p-temp-stk ($p0$)))) = `'addr`)
$\wedge$ listp (cadr (top (cddr (p-temp-stk ($p0$)))))
$\wedge$ (cdadr (top (cddr (p-temp-stk ($p0$)))) = `0`)
$\wedge$ (cddr (top (cddr (p-temp-stk ($p0$)))) = **nil**))

THEOREM: correctness-of-nat-to-bv-list
(($p0$ = p-state (pc,
ctrl-stk,
cons (l, cons (b, cons (n, *temp-stk*))),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))
$\wedge$ nat-to-bv-list-input-conditionp ($p0$)
$\wedge$ (*natlist* = array (caadr (n), *data-segment*)))
$\rightarrow$ (p (p-state (pc,
ctrl-stk,
cons (l, cons (b, cons (n, *temp-stk*))),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
nat-to-bv-list-clock (*natlist*))
= p-state (add1-addr (pc),
ctrl-stk,
temp-stk,
prog-segment,
put-assoc (tag-array (`'bitv`,
nat-to-bv-list (untag-array (array (caadr (n),
data-segment)),
word-size)),
caadr (b),
data-segment),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

Event: Disable nat-to-bv-list-program.

Event: Disable match-and-xor-program.

Event: Disable highest-bit-program.

Event: Disable number-with-at-least-program.

Event: Disable bv-to-nat-program.

Event: Disable nat-to-bv-program.

Event: Disable push-1-vector-program.

Event: Disable xor-bvs-program.

```
;; bv-to-nat-list
```

Definition:
bv-to-nat-list-program

```
=  '(bv-to-nat-list
      (bv-list nat-list length)
      ((i (nat 0)))
      (dl loop nil (push-local bv-list))
      (fetch)
      (call bv-to-nat)
      (push-local nat-list)
      (deposit)
      (push-local i)
      (add1-nat)
      (set-local i)
      (push-local length)
      (eq)
      (test-bool-and-jump t done)
      (push-local nat-list)
      (push-constant (nat 1))
      (add-addr)
      (pop-local nat-list)
      (push-local bv-list)
```

```
      (push-constant (nat 1))
      (add-addr)
      (pop-local bv-list)
      (jump loop)
      (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-BV-TO-NAT-LIST-P-STATE
```
=  p-state ('(pc (main . 0)),
            '((nil (pc (main . 0)))),
            nil,
            list ('(main nil nil
                     (push-constant (addr (arr . 0)))
                     (push-constant (addr (arr2 . 0)))
                     (push-constant (nat 6))
                     (call bv-to-nat-list)
                     (ret)),
                  BV-TO-NAT-LIST-PROGRAM,
                  BV-TO-NAT-PROGRAM,
                  push-1-vector-program (8)),
            '((arr
               (bitv (0 1 0 1 0 1 0 0))
               (bitv (1 1 1 1 1 1 1 1))
               (bitv (0 1 0 1 0 1 0 0))
               (bitv (0 1 0 1 0 1 0 0))
               (bitv (0 0 0 0 0 0 0 0))
               (bitv (0 1 0 1 0 1 0 0)))
              (arr2 nil nil nil nil nil nil)),
            100,
            80,
            8,
            'run)
```

DEFINITION:
bv-to-nat-list-input-conditionp ($p0$)
```
=  (listp (p-ctrl-stk (p0))
    ∧  at-least-morep (p-ctrl-stk-size (p-ctrl-stk (p0)),
                       13,
                       p-max-ctrl-stk-size (p0))
    ∧  at-least-morep (length (p-temp-stk (p0)),
                       0,
                       p-max-temp-stk-size (p0))
    ∧  (definition ('bv-to-nat-list, p-prog-segment (p0))
         =  BV-TO-NAT-LIST-PROGRAM)
```

```
∧   (definition ('bv-to-nat, p-prog-segment (p0))
     =   BV-TO-NAT-PROGRAM)
∧   (definition ('push-1-vector, p-prog-segment (p0))
     =   push-1-vector-program (p-word-size (p0)))
∧   (cadr (top (p-temp-stk (p0)))
     =   length (array (caadr (top (cddr (p-temp-stk (p0)))),
                          p-data-segment (p0))))
∧   array-pitonp (array (caadr (top (cdr (p-temp-stk (p0)))),
                          p-data-segment (p0)),
                   cadr (top (p-temp-stk (p0))))
∧   bit-vectors-piton (array (caadr (top (cddr (p-temp-stk (p0)))),
                               p-data-segment (p0)),
                        p-word-size (p0))
∧   definedp (caadr (top (cddr (p-temp-stk (p0)))), p-data-segment (p0))
∧   definedp (caadr (top (cdr (p-temp-stk (p0)))), p-data-segment (p0))
∧   (p-current-instruction (p0) = '(call bv-to-nat-list))
∧   (p-word-size (p0) ≄ 0)
∧   (caadr (top (cddr (p-temp-stk (p0))))
     ≠   caadr (top (cdr (p-temp-stk (p0)))))
∧   (car (top (p-temp-stk (p0))) = 'nat)
∧   (cadr (top (p-temp-stk (p0))) ≄ 0)
∧   (cadr (top (p-temp-stk (p0))) < exp (2, p-word-size (p0)))
∧   (cddr (top (p-temp-stk (p0))) = nil)
∧   (car (top (cdr (p-temp-stk (p0)))) = 'addr)
∧   listp (cadr (top (cdr (p-temp-stk (p0)))))
∧   (cdadr (top (cdr (p-temp-stk (p0)))) = 0)
∧   (cddr (top (cdr (p-temp-stk (p0)))) = nil)
∧   (car (top (cddr (p-temp-stk (p0)))) = 'addr)
∧   listp (cadr (top (cddr (p-temp-stk (p0)))))
∧   (cdadr (top (cddr (p-temp-stk (p0)))) = 0)
∧   (cddr (top (cddr (p-temp-stk (p0)))) = nil))
```

DEFINITION:

```
bv-to-nat-list-loop-clock (wordsize, i, bvs)
=   if i < length (bvs)
    then clock-plus ('2,
                     clock-plus (bv-to-nat-clock (wordsize, untag (get (i, bvs))),
                                 if i < length (cdr (bvs))
                                 then clock-plus ('17,
                                                  bv-to-nat-list-loop-clock (wordsize,
                                                                             1 + i,
                                                                             bvs))
                                 else '9 endif))
    else '0 endif
```

DEFINITION:
bv-to-nat-list (*bv-list*)
= **if** listp (*bv-list*)
then cons (bv-to-nat (car (*bv-list*)), bv-to-nat-list (cdr (*bv-list*)))
else nil endif

DEFINITION:
bv-to-nat-list-loop-induct (*i*, *bvs*, *nats-name*, *data-segment*)
= **if** $(i <$ length (*bvs*)$) \land (i <$ length (cdr (*bvs*)))
then bv-to-nat-list-loop-induct (1 + *i*,
bvs,
nats-name,
put-assoc (append (make-list-from (*i*,
cdr (assoc (*nats-name*,
data-segment))),
cons (list (`'nat`,
bv-to-nat (untag (get (*i*,
bvs)))),
nthcdr (1 + *i*,
cdr (assoc (*nats-name*,
data-segment))))),
nats-name,
data-segment))
else t endif

EVENT: Disable bv-to-nat-clock.

THEOREM: correctness-of-bv-to-nat-list-general
(listp (*ctrl-stk*)
$\land$ at-least-morep (p-ctrl-stk-size (*ctrl-stk*), `13`, *max-ctrl-stk-size*)
$\land$ at-least-morep (length (*temp-stk*), `3`, *max-temp-stk-size*)
$\land$ (definition (`'bv-to-nat-list`, *prog-segment*)
= BV-TO-NAT-LIST-PROGRAM)
$\land$ (definition (`'bv-to-nat`, *prog-segment*) = BV-TO-NAT-PROGRAM)
$\land$ (definition (`'push-1-vector`, *prog-segment*)
= push-1-vector-program (*word-size*))
$\land$ (*length* = length (array (*bvs*, *data-segment*)))
$\land$ bit-vectors-piton (array (*bvs*, *data-segment*), *word-size*)
$\land$ array-pitonp (array (*nats*, *data-segment*), *length*)
$\land$ definedp (*nats*, *data-segment*)
$\land$ definedp (*bvs*, *data-segment*)
$\land$ (*word-size* $\not\simeq$ `0`)
$\land$ $(i \in \mathbf{N})$
$\land$ $(i < length)$

```
∧   (length < exp (2, word-size))
∧   (nats ≠ bvs)
∧   (bvlist = array (bvs, data-segment)))
→   (p (p-state ('(pc (bv-to-nat-list . 0)),
                 cons (list (list (cons ('bv-list, list ('addr, cons (bvs, i))),
                                   cons ('nat-list,
                                         list ('addr, cons (nats, i))),
                                   cons ('length, list ('nat, length)),
                                   cons ('i, list ('nat, i))),
                             ret-pc),
                       ctrl-stk),
                 temp-stk,
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run),
        bv-to-nat-list-loop-clock (word-size, i, bvlist))
     =  p-state (ret-pc,
                 ctrl-stk,
                 temp-stk,
                 prog-segment,
                 put-assoc (append (make-list-from (i,
                                                    array (nats,
                                                           data-segment)),
                                    tag-array ('nat,
                                               bv-to-nat-list (untag-array (nthcdr (i,
                                                                                    array (bvs,
                                                                                           data-segment)))))),
                            nats,
                            data-segment),
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run))
```

Definition:
bv-to-nat-list-clock (*word-size*, *natlist*)
= clock-plus (1, bv-to-nat-list-loop-clock (*word-size*, 0, *natlist*))

Event: Disable bv-to-nat-list-loop-clock.

Event: Disable bv-to-nat-list-program.

THEOREM: correctness-of-bv-to-nat-list
$((p0 = \text{p-state}(pc,$
$\quad ctrl\text{-}stk,$
$\quad \text{cons}(l, \text{cons}(n, \text{cons}(b, temp\text{-}stk))),$
$\quad prog\text{-}segment,$
$\quad data\text{-}segment,$
$\quad max\text{-}ctrl\text{-}stk\text{-}size,$
$\quad max\text{-}temp\text{-}stk\text{-}size,$
$\quad word\text{-}size,$
$\quad \texttt{'run}))$
$\land \quad \text{bv-to-nat-list-input-conditionp}(p0)$
$\land \quad (bvlist = \text{array}(\text{caadr}(b), data\text{-}segment)))$
$\rightarrow \quad (\text{p}(\text{p-state}(pc,$
$\quad ctrl\text{-}stk,$
$\quad \text{cons}(l, \text{cons}(n, \text{cons}(b, temp\text{-}stk))),$
$\quad prog\text{-}segment,$
$\quad data\text{-}segment,$
$\quad max\text{-}ctrl\text{-}stk\text{-}size,$
$\quad max\text{-}temp\text{-}stk\text{-}size,$
$\quad word\text{-}size,$
$\quad \texttt{'run}),$
$\quad \text{bv-to-nat-list-clock}(word\text{-}size, bvlist))$
$\quad = \quad \text{p-state}(\text{add1-addr}(pc),$
$\quad ctrl\text{-}stk,$
$\quad temp\text{-}stk,$
$\quad prog\text{-}segment,$
$\quad \text{put-assoc}(\text{tag-array}(\texttt{'nat},$
$\quad \text{bv-to-nat-list}(\text{untag-array}(\text{array}(\text{caadr}(b),$
$\quad data\text{-}segment)))),$
$\quad \text{caadr}(n),$
$\quad data\text{-}segment),$
$\quad max\text{-}ctrl\text{-}stk\text{-}size,$
$\quad max\text{-}temp\text{-}stk\text{-}size,$
$\quad word\text{-}size,$
$\quad \texttt{'run}))$

```
;; max-nat
```

DEFINITION:
MAX-NAT-PROGRAM
```
=  '(max-nat
     (nat-list length)
```

```
((i (nat 0)) (j (nat 0)))
(push-constant (nat 0))
(dl loop nil (set-local j))
(push-local j)
(push-local nat-list)
(fetch)
(set-local j)
(lt-nat)
(test-bool-and-jump f lab)
(pop)
(push-local j)
(dl lab nil (push-local i))
(add1-nat)
(set-local i)
(push-local length)
(eq)
(test-bool-and-jump t done)
(push-local nat-list)
(push-constant (nat 1))
(add-addr)
(pop-local nat-list)
(jump loop)
(dl done nil (ret)))
```

DEFINITION:
EXAMPLE-MAX-NAT-P-STATE

```
=  p-state ('(pc (main . 0)),
            '((nil (pc (main . 0)))),
            nil,
            list ('(main nil nil
                     (push-constant (addr (arr . 0)))
                     (push-constant (nat 6))
                     (call max-nat)
                     (ret)),
                  MAX-NAT-PROGRAM),
            '((arr
                (nat 3)
                (nat 10)
                (nat 3)
                (nat 6)
                (nat 9)
                (nat 0))
               (arr2 nil nil nil nil nil nil)),
            100,
```

```
            80,
            8,
            'run)
```

DEFINITION:
max-list-helper (*val*, *x*)
```
=   if listp (x)
    then if val < car (x)  then max-list-helper (car (x), cdr (x))
         else max-list-helper (val, cdr (x)) endif
    else fix (val) endif
```

DEFINITION:
max-list (*x*)
```
=   if listp (x)
    then if max-list (cdr (x)) < car (x)  then car (x)
         else max-list (cdr (x)) endif
    else 0 endif
```

THEOREM: max-list-helper-max-list
```
 max-list-helper (val, x)
 =   if max-list (x) < val  then val
     else max-list (x) endif
```

THEOREM: max-list-helper-max-list-0
max-list-helper (0, *x*) = max-list (*x*)

EVENT: Disable max-list-helper-max-list.

DEFINITION:
max-nat-loop-clock (*val*, *i*, *nats*)
```
=   if i < length (nats)
    then if val < untag (get (i, nats))
         then if i < length (cdr (nats))
              then clock-plus (20,
                               max-nat-loop-clock (untag (get (i, nats)),
                                                   1 + i,
                                                   nats))
              else 16 endif
         elseif i < length (cdr (nats))
         then clock-plus (18, max-nat-loop-clock (val, 1 + i, nats))
         else 14 endif
    else 0 endif
```

DEFINITION:
max-nat-loop-induct (*i*, *j*, *x*, *natlist*, *length*)

$=$ **if** $i <$ *length*
then max-nat-loop-induct (1 + i,
get (i, *natlist*),
if untag (get (i, *natlist*)) $<$ untag (x)
then list (`'nat`, untag (x))
else list (`'nat`, untag (get (i, *natlist*))) **endif**,
natlist,
length)
else t endif

THEOREM: list-nat-from-assoc-nat-list-piton-hack
nat-list-piton (nl, s)
$\rightarrow$ ((($z <$ length (cdr (nl)))
$\rightarrow$ (list (`'nat`, cadr (get (z, cdr (nl)))) = get (z, cdr (nl))))
$\wedge$ (listp (nl)
$\rightarrow$ ((list (`'nat`, cadr (last (nl))) = last (nl))
$\wedge$ (list (`'nat`, cadr (car (nl))) = car (nl)))))

THEOREM: correctness-of-max-nat-general
(listp (*ctrl-stk*)
$\wedge$ at-least-morep (p-ctrl-stk-size (*ctrl-stk*), `4`, *max-ctrl-stk-size*)
$\wedge$ at-least-morep (length (*temp-stk*), `4`, *max-temp-stk-size*)
$\wedge$ (definition (`'max-nat`, *prog-segment*) = MAX-NAT-PROGRAM)
$\wedge$ (*length* = length (array (*nats*, *data-segment*)))
$\wedge$ nat-list-piton (array (*nats*, *data-segment*), *word-size*)
$\wedge$ definedp (*nats*, *data-segment*)
$\wedge$ (*word-size* $\not\simeq$ `0`)
$\wedge$ ($i \in \mathbf{N}$)
$\wedge$ ($i <$ *length*)
$\wedge$ (untag (x) $\in \mathbf{N}$)
$\wedge$ (untag (x) $<$ exp (`2`, *word-size*))
$\wedge$ (car (x) = `'nat`)
$\wedge$ (cddr (x) = **nil**)
$\wedge$ (*length* $<$ exp (`2`, *word-size*))
$\wedge$ (*natlist* = array (*nats*, *data-segment*)))
$\rightarrow$ (p (p-state (`'(pc (max-nat . 1))`,
cons (list (list (cons (`'nat-list`,
list (`'addr`, cons (*nats*, i))),
cons (`'length`, list (`'nat`, *length*)),
cons (`'i`, list (`'nat`, i)),
cons (`'j`, j)),
ret-pc),
ctrl-stk),
cons (x, *temp-stk*),

```
                  prog-segment,
                  data-segment,
                  max-ctrl-stk-size,
                  max-temp-stk-size,
                  word-size,
                  'run),
        max-nat-loop-clock (untag (x), i, array (nats, data-segment)))
   =   p-state (ret-pc,
                ctrl-stk,
                cons (list ('nat,
                            max-list-helper (untag (x),
                                             untag-array (nthcdr (i,
                                                                  array (nats,
                                                                         data-segment))))),
                      temp-stk),
                prog-segment,
                data-segment,
                max-ctrl-stk-size,
                max-temp-stk-size,
                word-size,
                'run))
```

Definition:
max-nat-clock (*natlist*) = clock-plus (2, max-nat-loop-clock (0, 0, *natlist*))

Definition:

```
max-nat-input-conditionp (p0)
=   (listp (p-ctrl-stk (p0))
     ∧  at-least-morep (p-ctrl-stk-size (p-ctrl-stk (p0)),
                        6,
                        p-max-ctrl-stk-size (p0))
     ∧  at-least-morep (length (p-temp-stk (p0)),
                        2,
                        p-max-temp-stk-size (p0))
     ∧  (definition ('max-nat, p-prog-segment (p0))
         =  MAX-NAT-PROGRAM)
     ∧  (untag (top (p-temp-stk (p0)))
         =  length (array (car (untag (top (cdr (p-temp-stk (p0))))),
                           p-data-segment (p0))))
     ∧  nat-list-piton (array (car (untag (top (cdr (p-temp-stk (p0))))),
                               p-data-segment (p0)),
                        p-word-size (p0))
     ∧  definedp (car (untag (top (cdr (p-temp-stk (p0))))),
                  p-data-segment (p0))
```

$\land$ (p-word-size (*p0*) $\not\simeq$ `0`)
$\land$ (untag (top (p-temp-stk (*p0*))) $\not\simeq$ `0`)
$\land$ (car (top (p-temp-stk (*p0*))) = `'nat`)
$\land$ (car (top (cdr (p-temp-stk (*p0*)))) = `'addr`)
$\land$ listp (untag (top (cdr (p-temp-stk (*p0*)))))
$\land$ (untag (top (p-temp-stk (*p0*))) < exp (`2`, p-word-size (*p0*)))
$\land$ (cddr (top (cdr (p-temp-stk (*p0*)))) = **nil**)
$\land$ (cddr (top (p-temp-stk (*p0*))) = **nil**)
$\land$ (cdr (untag (top (cdr (p-temp-stk (*p0*))))) = `0`))

THEOREM: correctness-of-max-nat
((*p0* = p-state (*pc*,
ctrl-stk,
cons (*n*, cons (*s*, *temp-stk*)),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))
$\land$ (p-current-instruction (*p0*) = `'(call max-nat)`)
$\land$ max-nat-input-conditionp (*p0*)
$\land$ (*natlist* = array (car (untag (*s*)), *data-segment*)))
$\rightarrow$ (p (p-state (*pc*,
ctrl-stk,
cons (*n*, cons (*s*, *temp-stk*)),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
max-nat-clock (*natlist*))
= p-state (add1-addr (*pc*),
ctrl-stk,
cons (list (`'nat`, max-list (untag-array (*natlist*))),
temp-stk),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

EVENT: Disable max-nat-program.

```
;; replace-value

;; replace the first occurrence of oldval by newval in list
;; of naturals (assumes oldval occurs in list)
```

DEFINITION:
REPLACE-VALUE-PROGRAM

```
=  '(replace-value
     (list oldval newval)
     nil
     (dl loop nil (push-local list))
     (fetch)
     (push-local oldval)
     (eq)
     (test-bool-and-jump t done)
     (push-local list)
     (push-constant (nat 1))
     (add-addr)
     (pop-local list)
     (jump loop)
     (dl done nil (push-local newval))
     (push-local list)
     (deposit)
     (ret))
```

DEFINITION:
EXAMPLE-REPLACE-VALUE-P-STATE

```
=  p-state ('(pc (main . 0)),
           '((nil (pc (main . 0)))),
           nil,
           list ('(main nil nil
                 (push-constant (addr (arr . 0)))
                 (push-constant (nat 3))
                 (push-constant (nat 4))
                 (call replace-value)
                 (ret)),
               REPLACE-VALUE-PROGRAM),
           '((arr
              (nat 9)
              (nat 10)
              (nat 3)
              (nat 6)
              (nat 9)
```

```
        (nat 0))),
    100,
    80,
    8,
    'run)
```

DEFINITION:
replace-value-loop-clock (i, *list*, *oldvalue*)
= **if** $i <$ length (*list*)
  then if get (i, *list*) $=$ *oldvalue* **then** `9`
    else clock-plus (`10`,
      replace-value-loop-clock ($1 + i$,
        list,
        oldvalue)) **endif**
  else `0` **endif**

DEFINITION:
replace-value-loop-induct (i, *list*, *list-addr*)
= **if** $i <$ length (*list*)
  then replace-value-loop-induct ($1 + i$, *list*, add1-addr (*list-addr*))
  else t endif

DEFINITION:
replace-value (*list*, *oldvalue*, *newvalue*)
= **if** listp (*list*)
  then if car (*list*) $=$ *oldvalue* **then** cons (*newvalue*, cdr (*list*))
    else cons (car (*list*),
      replace-value (cdr (*list*), *oldvalue*, *newvalue*)) **endif**
  else nil endif

THEOREM: member-nthcdr-means
$(x \in \text{nthcdr}(i,\ y)) \rightarrow (x \in y)$

THEOREM: member-of-natlist-means
(nat-list-piton (y, s) $\land$ ($x \in y$))
$\rightarrow$ (listp (x)
  $\land$ (car (x) $=$ `'nat`)
  $\land$ (cadr (x) $\in \mathbf{N}$)
  $\land$ (cadr (x) $<$ exp (`2`, s))
  $\land$ (cddr (x) $=$ **nil**))

THEOREM: replace-value-nthcdr-open
((get (x, a) $\neq$ *old*) $\land$ ($x <$ length (a)))
$\rightarrow$ (replace-value (nthcdr (x, a), *old*, *new*)
  $=$ cons (get (x, a), replace-value (nthcdr (x, cdr (a)), *old*, *new*)))

THEOREM: list-nat-cadr-get-hack
$(\text{nat-list-piton}\,(y,\ s) \land (x < \text{length}\,(y)))$
$\rightarrow\quad (\text{list}\,(\texttt{'nat},\ \text{cadr}\,(\text{get}\,(x,\ y))) = \text{get}\,(x,\ y))$

THEOREM: member-nthcdr-simplify
$((i < \text{length}\,(x)) \land (a \notin \text{nthcdr}\,(i,\ \text{cdr}\,(x))))$
$\rightarrow\quad ((a \in \text{nthcdr}\,(i,\ x)) = (a = \text{get}\,(i,\ x)))$

THEOREM: append-make-list-from-cons-cdr
$(i < \text{length}\,(y))$
$\rightarrow\quad (\text{append}\,(\text{make-list-from}\,(i,\ y),\ \text{cons}\,(v,\ \text{cdr}\,(\text{nthcdr}\,(i,\ y)))) = \text{put}\,(v,\ i,\ y))$

THEOREM: correctness-of-replace-value-general
(listp (*ctrl-stk*)
∧ at-least-morep (length (*temp-stk*), `2`, *max-temp-stk-size*)
∧ (definition (`'replace-value`, *prog-segment*)
  = REPLACE-VALUE-PROGRAM)
∧ nat-list-piton (array (*list*, *data-segment*), *word-size*)
∧ definedp (*list*, *data-segment*)
∧ (*word-size* $\not\simeq$ `0`)
∧ ($i \in \mathbf{N}$)
∧ (*vallist* = array (*list*, *data-segment*))
∧ (car (*list-addr*) = `'addr`)
∧ (cddr (*list-addr*) = **nil**)
∧ listp (untag (*list-addr*))
∧ (car (untag (*list-addr*)) = *list*)
∧ (cdr (untag (*list-addr*)) = *i*)
∧ (car (*newvalue*) = `'nat`)
∧ (cddr (*newvalue*) = **nil**)
∧ (car (*oldvalue*) = `'nat`)
∧ (cddr (*oldvalue*) = **nil**)
∧ (cadr (*oldvalue*) ∈ **N**)
∧ (cadr (*oldvalue*) < exp (`2`, *word-size*))
∧ (untag (*newvalue*) ∈ **N**)
∧ (untag (*newvalue*) < exp (`2`, *word-size*))
∧ (*oldvalue* ∈ nthcdr (*i*, *vallist*)))
→ (p (p-state (`'(pc (replace-value . 0))`,
      cons (list (list (cons (`'list`, *list-addr*),
            cons (`'oldval`, *oldvalue*),
            cons (`'newval`, *newvalue*)),
          ret-pc),
        ctrl-stk),
      temp-stk,
      prog-segment,
      data-segment,

max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
replace-value-loop-clock (i, *vallist*, *oldvalue*))
= p-state (*ret-pc*,
ctrl-stk,
temp-stk,
prog-segment,
put-assoc (append (make-list-from (i,
array (*list*,
data-segment)),
replace-value (nthcdr (i,
array (*list*,
data-segment)),
oldvalue,
newvalue)),
list,
data-segment),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

DEFINITION:
replace-value-clock (*list*, *ov*)
= clock-plus (`1`, replace-value-loop-clock (`0`, *list*, *ov*))

DEFINITION:
replace-value-input-conditionp (*p0*)
= (listp (p-ctrl-stk (*p0*))
∧ at-least-morep (length (p-temp-stk (*p0*)),
`0`,
p-max-temp-stk-size (*p0*))
∧ at-least-morep (p-ctrl-stk-size (p-ctrl-stk (*p0*)),
`5`,
p-max-ctrl-stk-size (*p0*))
∧ (definition (`'replace-value`, p-prog-segment (*p0*))
= REPLACE-VALUE-PROGRAM)
∧ nat-list-piton (array (car (untag (top (cddr (p-temp-stk (*p0*))))),
p-data-segment (*p0*)),
p-word-size (*p0*))
∧ definedp (car (untag (top (cddr (p-temp-stk (*p0*))))),
p-data-segment (*p0*))

$$
\begin{array}{ll}
\land & (\text{p-word-size}\,(p0) \not\simeq \texttt{0}) \\
\land & (\text{car}\,(\text{top}\,(\text{cddr}\,(\text{p-temp-stk}\,(p0)))) = \texttt{'addr}) \\
\land & (\text{cddr}\,(\text{top}\,(\text{cddr}\,(\text{p-temp-stk}\,(p0)))) = \textbf{nil}) \\
\land & \text{listp}\,(\text{untag}\,(\text{top}\,(\text{cddr}\,(\text{p-temp-stk}\,(p0))))) \\
\land & (\text{cdr}\,(\text{untag}\,(\text{top}\,(\text{cddr}\,(\text{p-temp-stk}\,(p0))))) = \texttt{0}) \\
\land & (\text{car}\,(\text{top}\,(\text{p-temp-stk}\,(p0))) = \texttt{'nat}) \\
\land & (\text{cddr}\,(\text{top}\,(\text{p-temp-stk}\,(p0))) = \textbf{nil}) \\
\land & (\text{untag}\,(\text{top}\,(\text{p-temp-stk}\,(p0))) \in \mathbf{N}) \\
\land & (\text{untag}\,(\text{top}\,(\text{p-temp-stk}\,(p0))) < \text{exp}\,(\texttt{2},\ \text{p-word-size}\,(p0))) \\
\land & (\text{top}\,(\text{cdr}\,(\text{p-temp-stk}\,(p0))) \\
& \quad \in \quad \text{array}\,(\text{car}\,(\text{untag}\,(\text{top}\,(\text{cddr}\,(\text{p-temp-stk}\,(p0))))), \\
& \qquad\qquad\qquad \text{p-data-segment}\,(p0))))
\end{array}
$$

THEOREM: correctness-of-replace-value

$$
\begin{array}{ll}
\multicolumn{2}{l}{((p0 = \text{p-state}\,(pc,} \\
& \qquad\qquad ctrl\text{-}stk, \\
& \qquad\qquad \text{cons}\,(nv,\ \text{cons}\,(ov,\ \text{cons}\,(nats,\ temp\text{-}stk))), \\
& \qquad\qquad prog\text{-}segment, \\
& \qquad\qquad data\text{-}segment, \\
& \qquad\qquad max\text{-}ctrl\text{-}stk\text{-}size, \\
& \qquad\qquad max\text{-}temp\text{-}stk\text{-}size, \\
& \qquad\qquad word\text{-}size, \\
& \qquad\qquad \texttt{'run})) \\
\land & (\text{p-current-instruction}\,(p0) = \texttt{'(call replace-value)}) \\
\land & \text{replace-value-input-conditionp}\,(p0) \\
\land & (natlist = \text{array}\,(\text{car}\,(\text{untag}\,(nats)),\ data\text{-}segment)) \\
\land & (ov = ov2)) \\
\rightarrow & (\text{p}\,(\text{p-state}\,(pc, \\
& \qquad\qquad ctrl\text{-}stk, \\
& \qquad\qquad \text{cons}\,(nv,\ \text{cons}\,(ov,\ \text{cons}\,(nats,\ temp\text{-}stk))), \\
& \qquad\qquad prog\text{-}segment, \\
& \qquad\qquad data\text{-}segment, \\
& \qquad\qquad max\text{-}ctrl\text{-}stk\text{-}size, \\
& \qquad\qquad max\text{-}temp\text{-}stk\text{-}size, \\
& \qquad\qquad word\text{-}size, \\
& \qquad\qquad \texttt{'run}), \\
& \quad \text{replace-value-clock}\,(natlist,\ ov2)) \\
& = \quad \text{p-state}\,(\text{add1-addr}\,(pc), \\
& \qquad\qquad ctrl\text{-}stk, \\
& \qquad\qquad temp\text{-}stk, \\
& \qquad\qquad prog\text{-}segment, \\
& \qquad\qquad \text{put-assoc}\,(\text{replace-value}\,(natlist,\ ov,\ nv), \\
& \qquad\qquad\qquad \text{caadr}\,(nats), \\
& \qquad\qquad\qquad data\text{-}segment),
\end{array}
$$

max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

EVENT: Disable replace-value-program.

```
;; smart-move
```

DEFINITION:
SMART-MOVE-PROGRAM

```
= '(smart-move
      (state numpiles work-area)
      ((i (nat 0)))
      (push-local state)
      (push-local numpiles)
      (push-constant (nat 2))
      (call number-with-at-least)
      (push-constant (nat 2))
      (lt-nat)
      (test-bool-and-jump t lab)
      (push-local state)
      (push-local work-area)
      (push-local numpiles)
      (call nat-to-bv-list)
      (push-local work-area)
      (push-local numpiles)
      (push-local work-area)
      (push-local numpiles)
      (call xor-bvs)
      (set-local i)
      (call highest-bit)
      (push-local i)
      (call match-and-xor)
      (push-local work-area)
      (push-local state)
      (push-local numpiles)
      (call bv-to-nat-list)
      (ret)
      (dl lab nil (push-local state))
      (push-local state)
      (push-local numpiles)
      (call max-nat)
```

```
(push-local state)
(push-local numpiles)
(push-constant (nat 1))
(call number-with-at-least)
(div2-nat)
(pop-local i)
(pop)
(push-local i)
(call replace-value)
(ret))
```

DEFINITION:
EXAMPLE-SMART-MOVE-P-STATE
= p-state ('(pc (main . 0)),
'((nil (pc (main . 0)))),
nil,
list ('(main nil nil

```
(push-constant (addr (arr . 0)))
(push-constant (nat 4))
(push-constant (addr (arr5 . 0)))
(call smart-move)
(push-constant (addr (arr2 . 0)))
(push-constant (nat 4))
(push-constant (addr (arr5 . 0)))
(call smart-move)
(push-constant (addr (arr3 . 0)))
(push-constant (nat 4))
(push-constant (addr (arr5 . 0)))
(call smart-move)
(push-constant (addr (arr4 . 0)))
(push-constant (nat 4))
(push-constant (addr (arr5 . 0)))
(call smart-move)
(ret)),
```

REPLACE-VALUE-PROGRAM,
MAX-NAT-PROGRAM,
BV-TO-NAT-LIST-PROGRAM,
NAT-TO-BV-LIST-PROGRAM,
MATCH-AND-XOR-PROGRAM,
HIGHEST-BIT-PROGRAM,
NUMBER-WITH-AT-LEAST-PROGRAM,
BV-TO-NAT-PROGRAM,
NAT-TO-BV-PROGRAM,
push-1-vector-program (8),

XOR-BVS-PROGRAM,
SMART-MOVE-PROGRAM),

```
'((arr (nat 3) (nat 4) (nat 2) (nat 1))
  (arr2 (nat 1) (nat 1) (nat 1) (nat 9))
  (arr3 (nat 1) (nat 1) (nat 0) (nat 9))
  (arr4 (nat 0) (nat 0) (nat 0) (nat 0))
  (arr5 (nat 3) (nat 4) (nat 2) (nat 1))),
100,
80,
8,
'run)
```

DEFINITION:
smart-move (*state*, *wordsize*)
= **if** number-with-at-least (*state*, 2) $<$ 2
then replace-value (*state*,
max-list (*state*),
number-with-at-least (*state*, 1) **mod** 2)
else bv-to-nat-list (match-and-xor (nat-to-bv-list (*state*, *wordsize*),
highest-bit (xor-bvs (nat-to-bv-list (*state*,
wordsize))),
xor-bvs (nat-to-bv-list (*state*,
wordsize)))) **endif**

```
(defn smart-move-clock (state wordsize)
  (clock-plus 4
   (clock-plus (number-with-at-least-clock 2 (tag-array 'nat state))
    (clock-plus 3
     (if (lessp (number-with-at-least state 2) 2)
 (clock-plus 3
  (clock-plus (max-nat-clock (tag-array 'nat state))
   (clock-plus 3
    (clock-plus (number-with-at-least-clock
 1 (tag-array 'nat state))
     (clock-plus 4
      (clock-plus (replace-value-clock (tag-array 'nat state)
       (list 'nat (max-list state)))
       1))))))
       (clock-plus 3
(clock-plus (nat-to-bv-list-clock (tag-array 'nat state))
   (clock-plus 4
  (clock-plus (xor-bvs-clock (length state))
```

```
   (clock-plus 1
    (clock-plus (highest-bit-clock
 (xor-bvs (nat-to-bv-list state wordsize)))
     (clock-plus 1
      (clock-plus (match-and-xor-clock
   (nat-to-bv-list state wordsize)
   (highest-bit
    (xor-bvs (nat-to-bv-list state wordsize))))
       (clock-plus 3
(clock-plus (bv-to-nat-list-clock
     wordsize
     (tag-array
      'bitv
      (match-and-xor
       (nat-to-bv-list state wordsize)
       (highest-bit
(xor-bvs (nat-to-bv-list state wordsize)))
       (xor-bvs (nat-to-bv-list state wordsize)))))
    1)))))))))))))))
```

EVENT: Disable replace-value-clock.

EVENT: Disable max-nat-clock.

EVENT: Disable bv-to-nat-list-clock.

EVENT: Disable nat-to-bv-list-clock.

EVENT: Disable match-and-xor-clock.

EVENT: Disable highest-bit-clock.

EVENT: Disable number-with-at-least-clock.

EVENT: Disable bv-to-nat-clock.

EVENT: Disable nat-to-bv-clock.

EVENT: Disable xor-bvs-clock.

EVENT: Disable replace-value-program.

EVENT: Disable max-nat-program.

EVENT: Disable bv-to-nat-list-program.

EVENT: Disable nat-to-bv-list-program.

EVENT: Disable match-and-xor-program.

EVENT: Disable highest-bit-program.

EVENT: Disable number-with-at-least-program.

EVENT: Disable bv-to-nat-program.

EVENT: Disable nat-to-bv-program.

EVENT: Disable push-1-vector-program.

EVENT: Disable *1*xor-bvs-program.

EVENT: Disable *1*replace-value-program.

EVENT: Disable *1*max-nat-program.

EVENT: Disable *1*bv-to-nat-list-program.

EVENT: Disable *1*nat-to-bv-list-program.

EVENT: Disable *1*match-and-xor-program.

EVENT: Disable *1*highest-bit-program.

EVENT: Disable *1*number-with-at-least-program.

EVENT: Disable *1*bv-to-nat-program.

EVENT: Disable *1*nat-to-bv-program.

EVENT: Disable *1*push-1-vector-program.

EVENT: Disable *1*xor-bvs-program.

DEFINITION:
nat-listp (*list*, *size*)
= **if** listp (*list*)
then (car (*list*) $\in$ **N**)
$\wedge$ (car (*list*) $<$ exp (2, *size*))
$\wedge$ nat-listp (cdr (*list*), *size*)
else *list* = **nil endif**

THEOREM: bv-to-nat-list-nat-to-bv-list
nat-listp (x, *size*) $\rightarrow$ (bv-to-nat-list (nat-to-bv-list (x, *size*)) = x)

THEOREM: bit-vectorsp-nat-to-bv-list
bit-vectorsp (nat-to-bv-list (x, *size*), *size*)

THEOREM: nat-to-bv-list-bv-to-nat-list
bit-vectorsp (x, *size*) $\rightarrow$ (nat-to-bv-list (bv-to-nat-list (x), *size*) = x)

THEOREM: bit-vectorsp-match-and-xor
bit-vectorsp (x, *size*) $\rightarrow$ bit-vectorsp (match-and-xor (x, y, z), *size*)

THEOREM: equal-sub1-add1
$(((x - 1) = (1 + y)) = (x = (1 + (1 + y))))$
$\wedge \quad (((x - 1) = 0) = ((x \simeq 0) \vee (x = 1)))$

```
;; part of more recent naturals library that's missing from Piton library
```

THEOREM: equal-times-x-x
$(((x * y) = x) = (((x \in \mathbf{N}) \wedge (y = 1)) \vee (x = 0)))$
$\wedge \quad (((y * x) = x) = (((x \in \mathbf{N}) \wedge (y = 1)) \vee (x = 0)))$

THEOREM: equal-exp-x-y-x
$(\exp(x, y) = x)$
$= \quad ((x = 1) \vee ((x = 0) \wedge (y \not\simeq 0)) \vee ((x \in \mathbf{N}) \wedge (y = 1)))$

THEOREM: lessp-number-with-at-least
length (x) $\not<$ number-with-at-least (x, *min*)

THEOREM: bit-vectors-piton-tag-array
bit-vectorsp $(x,\ size) \rightarrow$ bit-vectors-piton (tag-array (`'bitv`, x), $size$)

THEOREM: tag-array-untag-array-of-nat-list-piton
nat-list-piton $(x,\ size) \rightarrow$ (tag-array (`'nat`, untag-array $(x)) = x)$

THEOREM: tag-array-untag-array-of-bit-vectors-piton
bit-vectors-piton $(x,\ size) \rightarrow$ (tag-array (`'bitv`, untag-array $(x)) = x)$

THEOREM: bit-vectorp-xor-bvs-nat-to-bv-list
bit-vectorp (xor-bvs (nat-to-bv-list $(x,\ s)$), s) = (listp $(x) \vee (s \simeq 0)$)

THEOREM: listp-cdr-assoc-hack-from-free
((length (cdr (assoc $(x,\ y)$))) = $free$) $\wedge$ ($free \not\simeq 0$))
$\rightarrow$ listp (cdr (assoc $(x,\ y)$))

THEOREM: bit-vectorp-highest-bit
bit-vectorp $(x,\ s) \rightarrow$ bit-vectorp (highest-bit (x), s)

THEOREM: length-match-and-xor
length (match-and-xor ($list$, m, v)) = length ($list$)

THEOREM: array-pitonp-from-nat-list-piton
nat-list-piton $(x,\ s)$
$\rightarrow$ (array-pitonp (x, $length$) = (fix ($length$) = length (x)))

THEOREM: untag-array-tag-array-of-bit-vectorsp
bit-vectorsp $(x,\ size) \rightarrow$ (untag-array (tag-array $(l,\ x)) = x$)

THEOREM: untag-array-tag-array-of-nat-to-bv-list
untag-array (tag-array (l, nat-to-bv-list $(x,\ size)$)) = nat-to-bv-list $(x,\ size)$

THEOREM: untag-array-tag-array-of-match-and-xor-hack
untag-array (tag-array (l, match-and-xor (nat-to-bv-list $(x,\ s)$, y, z)))
= match-and-xor (nat-to-bv-list $(x,\ s)$, y, z)

THEOREM: member-list-max-list
nat-list-piton $(x,\ s)$
$\rightarrow$ ((list (`'nat`, max-list (untag-array (x))) $\in x$) = listp (x))

THEOREM: tag-array-replace-value-untag-array
nat-list-piton $(x,\ s)$
$\rightarrow$ (tag-array (`'nat`, replace-value (untag-array (x), y, z))
= replace-value (x, list (`'nat`, y), list (`'nat`, z)))

DEFINITION:

smart-move-input-conditionp ($p0$)

$=$ (listp (p-ctrl-stk ($p0$))

$\wedge$ at-least-morep (length (p-temp-stk ($p0$)),
3,
p-max-temp-stk-size ($p0$))

$\wedge$ at-least-morep (p-ctrl-stk-size (p-ctrl-stk ($p0$)),
19,
p-max-ctrl-stk-size ($p0$))

$\wedge$ ($1 <$ p-word-size ($p0$))

$\wedge$ (cddr (top (p-temp-stk ($p0$))) $=$ **nil**)

$\wedge$ (cddr (top (cdr (p-temp-stk ($p0$)))) $=$ **nil**)

$\wedge$ (cddr (top (cddr (p-temp-stk ($p0$)))) $=$ **nil**)

$\wedge$ (car (top (p-temp-stk ($p0$))) $=$ `'addr`)

$\wedge$ (car (top (cdr (p-temp-stk ($p0$)))) $=$ `'nat`)

$\wedge$ (untag (top (cdr (p-temp-stk ($p0$)))) $<$ exp (2, p-word-size ($p0$)))

$\wedge$ (untag (top (cdr (p-temp-stk ($p0$)))) $\not\simeq$ 0)

$\wedge$ (car (top (cddr (p-temp-stk ($p0$)))) $=$ `'addr`)

$\wedge$ listp (untag (top (p-temp-stk ($p0$))))

$\wedge$ listp (untag (top (cddr (p-temp-stk ($p0$)))))

$\wedge$ (cdr (untag (top (p-temp-stk ($p0$)))) $=$ 0)

$\wedge$ (cdr (untag (top (cddr (p-temp-stk ($p0$))))) $=$ 0)

$\wedge$ definedp (car (untag (top (p-temp-stk ($p0$)))), p-data-segment ($p0$))

$\wedge$ definedp (car (untag (top (cddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$))

$\wedge$ (car (untag (top (p-temp-stk ($p0$))))
$\neq$ car (untag (top (cddr (p-temp-stk ($p0$))))))

$\wedge$ nat-list-piton (array (car (untag (top (cddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$)),
p-word-size ($p0$))

$\wedge$ array-pitonp (array (car (untag (top (p-temp-stk ($p0$)))),
p-data-segment ($p0$)),
untag (top (cdr (p-temp-stk ($p0$)))))

$\wedge$ (length (array (car (untag (top (cddr (p-temp-stk ($p0$))))),
p-data-segment ($p0$)))
$=$ untag (top (cdr (p-temp-stk ($p0$)))))

$\wedge$ (assoc (`'smart-move`, p-prog-segment ($p0$))
$=$ SMART-MOVE-PROGRAM)

$\wedge$ (assoc (`'replace-value`, p-prog-segment ($p0$))
$=$ REPLACE-VALUE-PROGRAM)

$\wedge$ (assoc (`'max-nat`, p-prog-segment ($p0$)) $=$ MAX-NAT-PROGRAM)

$\wedge$ (assoc (`'bv-to-nat-list`, p-prog-segment ($p0$))
$=$ BV-TO-NAT-LIST-PROGRAM)

$\wedge$ (assoc (`'nat-to-bv-list`, p-prog-segment ($p0$))

```
        =   NAT-TO-BV-LIST-PROGRAM)
   ∧   (assoc ('match-and-xor, p-prog-segment (p0))
        =   MATCH-AND-XOR-PROGRAM)
   ∧   (assoc ('highest-bit, p-prog-segment (p0))
        =   HIGHEST-BIT-PROGRAM)
   ∧   (assoc ('number-with-at-least, p-prog-segment (p0))
        =   NUMBER-WITH-AT-LEAST-PROGRAM)
   ∧   (assoc ('bv-to-nat, p-prog-segment (p0))
        =   BV-TO-NAT-PROGRAM)
   ∧   (assoc ('nat-to-bv, p-prog-segment (p0))
        =   NAT-TO-BV-PROGRAM)
   ∧   (assoc ('push-1-vector, p-prog-segment (p0))
        =   push-1-vector-program (p-word-size (p0)))
   ∧   (assoc ('xor-bvs, p-prog-segment (p0)) = XOR-BVS-PROGRAM))
```

THEOREM: correctness-of-smart-move

```
((p0 = p-state (pc,
                ctrl-stk,
                cons (wa, cons (np, cons (s, temp-stk))),
                prog-segment,
                data-segment,
                max-ctrl-stk-size,
                max-temp-stk-size,
                word-size,
                'run))
 ∧  (p-current-instruction (p0) = '(call smart-move))
 ∧  (state = untag-array (array (car (untag (s)), data-segment)))
 ∧  (word-size = word-size2)
 ∧  smart-move-input-conditionp (p0))
 →  (p (p-state (pc,
                 ctrl-stk,
                 cons (wa, cons (np, cons (s, temp-stk))),
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run),
        smart-move-clock (state, word-size2))
     =  p-state (add1-addr (pc),
                 ctrl-stk,
                 temp-stk,
                 prog-segment,
                 put-assoc (tag-array ('nat,
```

smart-move (*state*, *word-size*)),
car (untag (*s*)),
if number-with-at-least (*state*, `2`) < `2`
then *data-segment*
else put-assoc (tag-array (`'bitv`,
nat-to-bv-list (smart-move (*state*,
word-size),
word-size)),
car (untag (*wa*)),
data-segment) **endif**),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

EVENT: Disable smart-move-clock.

EVENT: Disable smart-move-program.

EVENT: Disable *1*smart-move-program.

```
;; delay
```

DEFINITION:
DELAY-PROGRAM

```
=  '(delay
      (time)
      nil
      (dl lab nil (push-local time))
      (sub1-nat)
      (set-local time)
      (no-op)
      (no-op)
      (no-op)
      (no-op)
      (test-nat-and-jump zero done)
      (no-op)
      (jump lab)
      (dl done nil (ret)))
```

DEFINITION:
EXAMPLE-DELAY-P-STATE

```
=   p-state ('(pc (main . 0)),
            '((nil (pc (main . 0)))),
            nil,
            list ('(main nil nil
                    (push-constant (nat 4))
                    (call delay)
                    (ret)),
                  DELAY-PROGRAM),
            nil,
            100,
            80,
            8,
            'run)
```

DEFINITION:
delay-loop-clock (*time*)
= **if** *time* $<$ `2` **then** `9`
  else `10` $+$ delay-loop-clock (*time* $-$ 1) **endif**

THEOREM: correctness-of-delay-general

```
 (listp (ctrl-stk)
  ∧  (definition ('delay, prog-segment) = DELAY-PROGRAM)
  ∧  at-least-morep (length (temp-stk), 1, max-temp-stk-size)
  ∧  (0 < time)
  ∧  (time < exp (2, word-size)))
 →  (p (p-state ('(pc (delay . 0)),
                 cons (list (list (cons ('time, list ('nat, time))), ret-pc),
                       ctrl-stk),
                 temp-stk,
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run),
        delay-loop-clock (time))
     =  p-state (ret-pc,
                 ctrl-stk,
                 temp-stk,
                 prog-segment,
                 data-segment,
                 max-ctrl-stk-size,
                 max-temp-stk-size,
                 word-size,
                 'run))
```

DEFINITION:
delay-input-conditionp ($p0$)
$=$ (listp (p-ctrl-stk ($p0$))
∧ (definition (`'delay`, p-prog-segment ($p0$)) $=$ DELAY-PROGRAM)
∧ (car (top (p-temp-stk ($p0$))) $=$ `'nat`)
∧ at-least-morep (length (p-temp-stk ($p0$)),
`0`,
p-max-temp-stk-size ($p0$))
∧ at-least-morep (p-ctrl-stk-size (p-ctrl-stk ($p0$)),
`3`,
p-max-ctrl-stk-size ($p0$))
∧ (`0` $<$ cadr (top (p-temp-stk ($p0$))))
∧ (cadr (top (p-temp-stk ($p0$))) $<$ exp (`2`, p-word-size ($p0$)))
∧ (cddr (top (p-temp-stk ($p0$))) $=$ **nil**))

DEFINITION: delay-clock ($time$) $=$ (1 + delay-loop-clock ($time$))

THEOREM: correctness-of-delay
(($p0$ $=$ p-state (pc,
$ctrl\text{-}stk$,
cons (n, $temp\text{-}stk$),
$prog\text{-}segment$,
$data\text{-}segment$,
$max\text{-}ctrl\text{-}stk\text{-}size$,
$max\text{-}temp\text{-}stk\text{-}size$,
$word\text{-}size$,
`'run`))
∧ (p-current-instruction ($p0$) $=$ `'(call delay)`)
∧ delay-input-conditionp ($p0$)
∧ ($time$ $=$ cadr (n)))
→ (p (p-state (pc,
$ctrl\text{-}stk$,
cons (n, $temp\text{-}stk$),
$prog\text{-}segment$,
$data\text{-}segment$,
$max\text{-}ctrl\text{-}stk\text{-}size$,
$max\text{-}temp\text{-}stk\text{-}size$,
$word\text{-}size$,
`'run`),
delay-clock ($time$))
$=$ p-state (add1-addr (pc),
$ctrl\text{-}stk$,
$temp\text{-}stk$,
$prog\text{-}segment$,

data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
'run))

```
;; computer-move
```

Definition:
COMPUTER-MOVE-PROGRAM

```
=  '(computer-move
      (state numpiles work-area)
      ((i (nat 0)))
      (push-constant (nat 250))
      (call delay)
      (push-constant (nat 250))
      (call delay)
      (push-constant (nat 250))
      (call delay)
      (push-constant (nat 250))
      (call delay)
      (push-local state)
      (push-local numpiles)
      (push-constant (nat 2))
      (call number-with-at-least)
      (test-nat-and-jump zero lab)
      (push-local state)
      (push-local work-area)
      (push-local numpiles)
      (call nat-to-bv-list)
      (push-local work-area)
      (push-local numpiles)
      (call xor-bvs)
      (test-bitv-and-jump all-zero lab)
      (push-local state)
      (push-local numpiles)
      (push-local work-area)
      (call smart-move)
      (ret)
      (dl lab nil (push-local state))
      (push-local state)
      (push-local numpiles)
      (call max-nat)
```

```
      (pop-local i)
      (push-local i)
      (push-local i)
      (sub1-nat)
      (call replace-value)
      (ret))
```

DEFINITION:
EXAMPLE-COMPUTER-MOVE-P-STATE
= p-state ('(pc (main . 0)),
          '((nil (pc (main . 0)))),
          nil,
          list ('(main nil nil
```
                (push-constant (addr (arr . 0)))
                (push-constant (nat 4))
                (push-constant (addr (arr5 . 0)))
                (call computer-move)
                (push-constant (addr (arr2 . 0)))
                (push-constant (nat 4))
                (push-constant (addr (arr5 . 0)))
                (call computer-move)
                (push-constant (addr (arr3 . 0)))
                (push-constant (nat 4))
                (push-constant (addr (arr5 . 0)))
                (call computer-move)
                (push-constant (addr (arr4 . 0)))
                (push-constant (nat 4))
                (push-constant (addr (arr5 . 0)))
                (call computer-move)
                (ret)),
```
                COMPUTER-MOVE-PROGRAM,
                DELAY-PROGRAM,
                REPLACE-VALUE-PROGRAM,
                MAX-NAT-PROGRAM,
                BV-TO-NAT-LIST-PROGRAM,
                NAT-TO-BV-LIST-PROGRAM,
                MATCH-AND-XOR-PROGRAM,
                HIGHEST-BIT-PROGRAM,
                NUMBER-WITH-AT-LEAST-PROGRAM,
                BV-TO-NAT-PROGRAM,
                NAT-TO-BV-PROGRAM,
                push-1-vector-program (8),
                XOR-BVS-PROGRAM,
                SMART-MOVE-PROGRAM),

```
'((arr (nat 3) (nat 4) (nat 2) (nat 1))
  (arr2 (nat 1) (nat 1) (nat 1) (nat 0))
  (arr3 (nat 1) (nat 1) (nat 0) (nat 9))
  (arr4 (nat 7) (nat 4) (nat 2) (nat 2))
  (arr5 (nat 3) (nat 4) (nat 2) (nat 1))),
100,
80,
8,
'run)
```

DEFINITION:
computer-move (*state*, *wordsize*)
= **if** (number-with-at-least (*state*, 2) = 0)
  ∨ all-zero-bitvp (xor-bvs (nat-to-bv-list (*state*, *wordsize*)))
  then replace-value (*state*, max-list (*state*), max-list (*state*) − 1)
  else smart-move (*state*, *wordsize*) **endif**

EVENT: Disable delay-clock.

```
(defn computer-move-clock (state wordsize)
  (clock-plus
   2
   (clock-plus (delay-clock 250)
       (clock-plus 1 (clock-plus (delay-clock 250)
       (clock-plus 1 (clock-plus (delay-clock 250)
       (clock-plus 1 (clock-plus (delay-clock 250)
  (clock-plus
   3
   (clock-plus
    (number-with-at-least-clock 2 (tag-array 'nat state))
    (clock-plus
     1
    (if (equal (number-with-at-least state 2) 0)
(clock-plus
    3
    (clock-plus
     (max-nat-clock (tag-array 'nat state))
     (clock-plus
      4
      (clock-plus
       (replace-value-clock
(tag-array 'nat state)
```

```
(list 'nat (max-list state)))
       1))))
  (clock-plus
   3
   (clock-plus
    (nat-to-bv-list-clock (tag-array 'nat state))
    (clock-plus
     2
     (clock-plus
      (xor-bvs-clock (length state))
      (clock-plus
       1
       (if (all-zero-bitvp (xor-bvs (nat-to-bv-list state wordsize)))
   (clock-plus
    3
    (clock-plus
     (max-nat-clock (tag-array 'nat state))
     (clock-plus
      4
      (clock-plus
       (replace-value-clock
(tag-array 'nat state)
(list 'nat (max-list state)))
       1))))
 (clock-plus
  3
  (clock-plus
   (smart-move-clock state wordsize)
   1))))))))))))))))))))
```

Definition:

computer-move-input-conditionp $(p0)$

$$
\begin{aligned}
= \quad & ((\neg \text{ all-zero-bitvp}(\text{untag-array}(\text{array}(\text{car}(\text{untag}(\text{top}(\text{cddr}(\text{p-temp-stk}(p0))))), \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}(p0))))) \\
& \quad \land \quad \text{listp}(\text{p-ctrl-stk}(p0)) \\
& \quad \land \quad \text{at-least-morep}(\text{length}(\text{p-temp-stk}(p0)), \\
& \qquad\qquad\qquad\qquad \mathtt{3}, \\
& \qquad\qquad\qquad\qquad \text{p-max-temp-stk-size}(p0)) \\
& \quad \land \quad \text{at-least-morep}(\text{p-ctrl-stk-size}(\text{p-ctrl-stk}(p0)), \\
& \qquad\qquad\qquad\qquad \mathtt{25}, \\
& \qquad\qquad\qquad\qquad \text{p-max-ctrl-stk-size}(p0)) \\
& \quad \land \quad (\mathtt{7} < \text{p-word-size}(p0)) \\
& \quad \land \quad (\text{p-word-size}(p0) \not\simeq \mathtt{0}) \\
& \quad \land \quad (\text{p-word-size}(p0) \neq \mathtt{1})
\end{aligned}
$$

$\land$ (cddr (top (p-temp-stk ($p0$))) = **nil**)
$\land$ (cddr (top (cdr (p-temp-stk ($p0$)))) = **nil**)
$\land$ (cddr (top (cddr (p-temp-stk ($p0$)))) = **nil**)
$\land$ (car (top (p-temp-stk ($p0$))) = `'addr`)
$\land$ (car (top (cdr (p-temp-stk ($p0$)))) = `'nat`)
$\land$ (untag (top (cdr (p-temp-stk ($p0$)))) $<$ exp (`2`, p-word-size ($p0$)))
$\land$ (untag (top (cdr (p-temp-stk ($p0$)))) $\not\simeq$ `0`)
$\land$ (car (top (cddr (p-temp-stk ($p0$)))) = `'addr`)
$\land$ listp (untag (top (p-temp-stk ($p0$))))
$\land$ listp (untag (top (cddr (p-temp-stk ($p0$)))))
$\land$ (cdr (untag (top (p-temp-stk ($p0$)))) = `0`)
$\land$ (cdr (untag (top (cddr (p-temp-stk ($p0$))))) = `0`)
$\land$ definedp (car (untag (top (p-temp-stk ($p0$)))), p-data-segment ($p0$))
$\land$ definedp (car (untag (top (cddr (p-temp-stk ($p0$))))),
    p-data-segment ($p0$))
$\land$ (car (untag (top (p-temp-stk ($p0$))))
    $\neq$ car (untag (top (cddr (p-temp-stk ($p0$))))))
$\land$ nat-list-piton (array (car (untag (top (cddr (p-temp-stk ($p0$))))),
    p-data-segment ($p0$)),
    p-word-size ($p0$))
$\land$ array-pitonp (array (car (untag (top (p-temp-stk ($p0$)))),
    p-data-segment ($p0$)),
    untag (top (cdr (p-temp-stk ($p0$)))))
$\land$ (length (array (car (untag (top (cddr (p-temp-stk ($p0$))))),
    p-data-segment ($p0$)))
    = untag (top (cdr (p-temp-stk ($p0$)))))
$\land$ (assoc (`'delay`, p-prog-segment ($p0$)) = DELAY-PROGRAM)
$\land$ (assoc (`'computer-move`, p-prog-segment ($p0$))
    = COMPUTER-MOVE-PROGRAM)
$\land$ (assoc (`'smart-move`, p-prog-segment ($p0$))
    = SMART-MOVE-PROGRAM)
$\land$ (assoc (`'replace-value`, p-prog-segment ($p0$))
    = REPLACE-VALUE-PROGRAM)
$\land$ (assoc (`'max-nat`, p-prog-segment ($p0$)) = MAX-NAT-PROGRAM)
$\land$ (assoc (`'bv-to-nat-list`, p-prog-segment ($p0$))
    = BV-TO-NAT-LIST-PROGRAM)
$\land$ (assoc (`'nat-to-bv-list`, p-prog-segment ($p0$))
    = NAT-TO-BV-LIST-PROGRAM)
$\land$ (assoc (`'match-and-xor`, p-prog-segment ($p0$))
    = MATCH-AND-XOR-PROGRAM)
$\land$ (assoc (`'highest-bit`, p-prog-segment ($p0$))
    = HIGHEST-BIT-PROGRAM)
$\land$ (assoc (`'number-with-at-least`, p-prog-segment ($p0$))
    = NUMBER-WITH-AT-LEAST-PROGRAM)

$\wedge$ (assoc (`'bv-to-nat`, p-prog-segment ($p0$))
  = BV-TO-NAT-PROGRAM)
$\wedge$ (assoc (`'nat-to-bv`, p-prog-segment ($p0$))
  = NAT-TO-BV-PROGRAM)
$\wedge$ (assoc (`'push-1-vector`, p-prog-segment ($p0$))
  = push-1-vector-program (p-word-size ($p0$)))
$\wedge$ (assoc (`'xor-bvs`, p-prog-segment ($p0$)) = XOR-BVS-PROGRAM))

THEOREM: numberp-max-list
max-list (x) $\in$ **N**

THEOREM: max-0-means
nat-list-piton (x, s)
$\rightarrow$ ((max-list (untag-array (x)) = `0`) = all-zero-bitvp (untag-array (x)))

THEOREM: max-list-not-too-big
(nat-list-piton (x, s) $\wedge$ ($s \not\simeq$ `0`))
$\rightarrow$ ((max-list (untag-array (x)) < exp (`2`, s))
  $\wedge$ (((max-list (untag-array (x)) $-$ 1) < exp (`2`, s)) = **t**))

EVENT: Disable delay-clock.

EVENT: Disable *1*delay-clock.

THEOREM: lessp-exp-2-8-hack
((`7` < *free*) $\wedge$ (x = *free*) $\wedge$ (*val* < `256`))
$\rightarrow$ ((*val* < exp (`2`, x)) = **t**)

THEOREM: correctness-of-computer-move
(($p0$ = p-state (*pc*,
  ctrl-stk,
  cons (*wa*, cons (*np*, cons (s, *temp-stk*))),
  prog-segment,
  data-segment,
  max-ctrl-stk-size,
  max-temp-stk-size,
  word-size,
  `'run`))
$\wedge$ (p-current-instruction ($p0$) = `'(call computer-move)`)
$\wedge$ (*state* = untag-array (array (car (untag (s)), *data-segment*)))
$\wedge$ (*word-size* = *word-size2*)
$\wedge$ computer-move-input-conditionp ($p0$))
$\rightarrow$ (p (p-state (*pc*,
  ctrl-stk,

cons (*wa*, cons (*np*, cons (*s*, *temp-stk*))),
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`),
computer-move-clock (*state*, *word-size2*))
= p-state (add1-addr (*pc*),
ctrl-stk,
temp-stk,
prog-segment,
put-assoc (tag-array (`'nat`,
computer-move (*state*, *word-size*)),
car (untag (*s*)),
if number-with-at-least (*state*, `2`) = `0`
then *data-segment*
else put-assoc (**if** all-zero-bitvp (xor-bvs (nat-to-bv-list (*state*,
word-size)))
∨ (number-with-at-least (*state*,
`2`)
< `2`)
then tag-array (`'bitv`,
nat-to-bv-list (*state*,
word-size))
else tag-array (`'bitv`,
nat-to-bv-list (computer-move (*state*,
word-size),
word-size)) **endif**,
car (untag (*wa*)),
data-segment) **endif**),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
`'run`))

```
;;;;
;;;;
;;;;
;;;;
;;;; Having proved the behavior of the programs, we now introduce
;;;; the spec to which we've been writing code.
;;;;
;;;;
```

```
;;;;
```

DEFINITION:
sum (*list*)
= **if** listp (*list*) **then** car (*list*) + sum (cdr (*list*))
  else 0 **endif**

THEOREM: sum-append
sum (append (x, y)) = (sum (x) + sum (y))

```
;; returns a list of states that are valid moves from
```

DEFINITION:
all-valid-moves-helper (*old*, *val*, *origval*, *new*)
= **if** *val* $\simeq$ 0
  then if *new* $\simeq$ **nil** **then nil**
    else all-valid-moves-helper (append (*old*, list (*origval*)),
      car (*new*),
      car (*new*),
      cdr (*new*)) **endif**
  else cons (append (*old*, cons (*val* $-$ 1, *new*)),
    all-valid-moves-helper (*old*, *val* $-$ 1, *origval*, *new*)) **endif**

DEFINITION:
all-valid-moves (x) = all-valid-moves-helper (**nil**, car (x), car (x), cdr (x))

DEFINITION:
max-sum (*list*)
= **if** listp (*list*)
  then if sum (car (*list*)) < max-sum (cdr (*list*)) **then** max-sum (cdr (*list*))
    else sum (car (*list*)) **endif**
  else 0 **endif**

THEOREM: nat-listp-append
properp (x)
$\rightarrow$ (nat-listp (append (x, y), *size*)
  = (nat-listp (x, *size*) $\wedge$ nat-listp (y, *size*)))

```
;(prove-lemma properp-append (rewrite)
;      (equal
;       (properp (append x y))
;       (properp y)))
```

EVENT: Enable properp-append.

DEFINITION:
nat-listp-simple (*list*)
$=$ **if** listp (*list*) **then** (car (*list*) $\in$ **N**) $\wedge$ nat-listp-simple (cdr (*list*))
else *list* $=$ **nil endif**

THEOREM: nat-listp-simple-append
properp (x)
$\rightarrow$ (nat-listp-simple (append (x, y))
$=$ (nat-listp-simple (x) $\wedge$ nat-listp-simple (y)))

THEOREM: lessp-max-sum-helper
(($c \not< temp$)
$\wedge$ properp (x)
$\wedge$ nat-listp-simple (append (x, cons (c, y)))
$\wedge$ ($\neg$ all-zero-bitvp (append (x, cons (c, y)))))
$\rightarrow$ (max-sum (all-valid-moves-helper (x, $temp$, c, y))
$<$ sum (append (x, cons (c, y))))

THEOREM: lessp-max-sum-all-valid-moves
(nat-listp-simple (s) $\wedge$ ($\neg$ all-zero-bitvp (s)))
$\rightarrow$ ((max-sum (all-valid-moves (s)) $<$ sum (s)) $=$ **t**)

THEOREM: lessp-sum-max-sum
max-sum (x) $\not<$ sum (car (x))

EVENT: Disable all-valid-moves.

DEFINITION:
wsp-measure (*state*, *flag*)
$=$ cons (**if** *flag* **then** $1 +$ sum (*state*)
else $1 +$ max-sum (*state*) **endif**,
if *flag* **then** 0
else length (*state*) **endif**)

```
;; wsp searchs for a successor to  the current state on
;; a path to a guaranteed win.
;; if flag
;;      state is a nim state - return state if all zeros.
;;      Return a successor state not wsp if one exists, f otherwise
;; if not flag
;;      state is a list of states - return member of list if it is
;;      not wsp, f is no such member.
```

DEFINITION:
wsp ($state$, $flag$)
= **if** $flag$
then if all-zero-bitvp ($state$) $\vee$ ($\neg$ nat-listp-simple ($state$))
then $state$
else wsp (all-valid-moves ($state$), **f**) **endif**
elseif listp ($state$)
then if $\neg$ wsp (car ($state$), **t**) **then** car ($state$)
else wsp (cdr ($state$), **f**) **endif**
else f endif

DEFINITION:
green-statep ($state$, $wordsize$)
= ((number-with-at-least ($state$, 2) $\simeq$ 0)
= all-zero-bitvp (xor-bvs (nat-to-bv-list ($state$, $wordsize$))))

DEFINITION:
non-green-in-list ($list$, $wordsize$)
= **if** listp ($list$)
then ($\neg$ green-statep (car ($list$), $wordsize$))
$\vee$ non-green-in-list (cdr ($list$), $wordsize$)
else f endif

THEOREM: nat-listp-means-nat-listp-simple
nat-listp (x, s) $\rightarrow$ nat-listp-simple (x)

THEOREM: xor-bitv-zero-bit-vector
all-zero-bitvp (x)
$\rightarrow$ (xor-bitv (x, y) = make-list-from (length (x), fix-bitv (y)))

THEOREM: fix-bitv-xor-bitv
fix-bitv (xor-bitv (x, y)) = xor-bitv (x, y)

THEOREM: fix-bitv-and-bitv
fix-bitv (and-bitv (x, y)) = and-bitv (x, y)

THEOREM: fix-bitv-xor-bvs
fix-bitv (xor-bvs (x)) = xor-bvs (x)

THEOREM: all-zero-bitvp-make-list-from-simple
all-zero-bitvp (x) $\rightarrow$ all-zero-bitvp (make-list-from (n, x))

THEOREM: all-zero-bitvp-xor-bvs-nat-to-bv-list-zeros
all-zero-bitvp (x) $\rightarrow$ all-zero-bitvp (xor-bvs (nat-to-bv-list (x, s)))

THEOREM: number-with-at-least-of-all-zeros
all-zero-bitvp (x)
$\rightarrow$ (number-with-at-least $(x,\ m)$
$=$ **if** $m \simeq 0$ **then** length (x)
else 0 **endif**)

DEFINITION:
nat-listp-listp (*list*, *wordsize*)
$=$ **if** listp (*list*)
then nat-listp (car (*list*), *wordsize*)
$\wedge$ nat-listp-listp (cdr (*list*), *wordsize*)
else t endif

THEOREM: nat-listp-listp-all-valid-moves-helper
(nat-listp $(a,\ s)$
$\wedge$ $(b < \exp(2,\ s))$
$\wedge$ $(c < \exp(2,\ s))$
$\wedge$ $(b \in \mathbf{N})$
$\wedge$ $(c \in \mathbf{N})$
$\wedge$ nat-listp $(d,\ s)$
$\wedge$ properp (a))
$\rightarrow$ nat-listp-listp (all-valid-moves-helper $(a,\ b,\ c,\ d)$, s)

THEOREM: nat-listp-listp-all-valid-moves
nat-listp $(a,\ s)$ $\rightarrow$ nat-listp-listp (all-valid-moves (a), s)

THEOREM: listp-all-valid-move-helper
nat-listp-simple (d)
$\rightarrow$ (listp (all-valid-moves-helper $(a,\ b,\ c,\ d)$)
$=$ (($\neg$ all-zero-bitvp (d)) $\vee$ $(b \not\simeq 0)$))

THEOREM: listp-all-valid-move
nat-listp-simple (x)
$\rightarrow$ (listp (all-valid-moves (x)) $=$ ($\neg$ all-zero-bitvp (x)))

THEOREM: number-with-at-least-append
number-with-at-least (append $(x,\ y)$, m)
$=$ (number-with-at-least $(x,\ m)$ $+$ number-with-at-least $(y,\ m)$)

THEOREM: length-xor-bvs2
bit-vectorsp $(x,\ s)$
$\rightarrow$ (length (xor-bvs (x))
$=$ **if** listp (x) **then** fix (s)
else 0 **endif**)

THEOREM: xor-bitv-xor-bvs-hack
bit-vectorsp (z, length (y))
$\rightarrow$ (xor-bitv (xor-bitv (y, xor-bvs (z)), x)
= **if** listp (z) **then** xor-bitv (y, xor-bitv (xor-bvs (z), x))
else xor-bitv (y, x) **endif**)

THEOREM: xor-bitv-fix-bitv
(xor-bitv (fix-bitv (x), y) = xor-bitv (x, y))
$\wedge$ (xor-bitv (x, fix-bitv (y)) = xor-bitv (x, y))

THEOREM: xor-bvs-append
(bit-vectorsp (x, s) $\wedge$ bit-vectorsp (y, s) $\wedge$ ($s \in \mathbf{N}$))
$\rightarrow$ (xor-bvs (append (x, y))
= **if** listp (x) **then** xor-bitv (xor-bvs (x), xor-bvs (y))
else xor-bvs (y) **endif**)

THEOREM: xor-bvs-append-hack
bit-vectorsp (y, ws)
$\rightarrow$ (xor-bvs (append (nat-to-bv-list (a, ws), y))
= **if** listp (a)
then xor-bitv (xor-bvs (nat-to-bv-list (a, ws)), xor-bvs (y))
else xor-bvs (y) **endif**)

THEOREM: nat-to-bv-list-append
nat-to-bv-list (append (x, y), s)
= append (nat-to-bv-list (x, s), nat-to-bv-list (y, s))

THEOREM: bit-vectorp-nat-to-bv
bit-vectorp (nat-to-bv (x, s), $s2$) = (fix (s) = fix ($s2$))

THEOREM: fix-bitv-nat-to-bv
fix-bitv (nat-to-bv (x, s)) = nat-to-bv (x, s)

THEOREM: fix-bitv-zero-bit-vector
fix-bitv (zero-bit-vector (x)) = zero-bit-vector (x)

THEOREM: all-zero-bitvp-nat-to-bv
all-zero-bitvp (nat-to-bv (x, s)) = (($x \simeq 0$) $\vee$ ($s \simeq 0$))

THEOREM: all-zero-bitvp-xor-bitv-better
all-zero-bitvp (xor-bitv (x, y))
= (fix-bitv (x) = make-list-from (length (x), fix-bitv (y)))

THEOREM: length-xor-bvs-nat-to-bv-list
length (xor-bvs (nat-to-bv-list (x, s)))
= **if** listp (x) **then** fix (s)
else 0 **endif**

THEOREM: fix-bitv-make-list-from
fix-bitv (make-list-from (s, x)) = make-list-from (s, fix-bitv (x))

DEFINITION:
double-sub1-cdr ($n1$, $n2$, l)
= **if** ($n1 \simeq 0$) $\vee$ ($n2 \simeq 0$) **then t**
else double-sub1-cdr ($n1 - 1$, $n2 - 1$, cdr (l)) **endif**

THEOREM: make-list-from-make-list-from
make-list-from ($s1$, make-list-from ($s2$, x))
= **if** $s2 < s1$
then append (make-list-from ($s2$, x), zero-bit-vector ($s1 - s2$))
else make-list-from ($s1$, x) **endif**

```
;(prove-lemma associativity-of-append (rewrite)
;      (equal
;       (append (append a b) c)
;       (append a (append b c))))
```

EVENT: Enable associativity-of-append.

```
;(prove-lemma append-cons (rewrite)
;      (equal
;       (append (cons a b) c)
;       (cons a (append b c))))
```

EVENT: Enable append-cons.

THEOREM: properp-xor-bvs
properp (xor-bvs (x))

THEOREM: properp-xor-bitv
properp (xor-bitv (x, y))

DEFINITION:
member-number-with-at-least (x, min)
= **if** listp (x)
then if number-with-at-least (car (x), min) $\not\simeq 0$ **then** car (x)
else member-number-with-at-least (cdr (x), min) **endif**
else f endif

THEOREM: xor-bvs-nat-to-bv-list-zerop-ws
($ws \simeq 0$) $\rightarrow$ (xor-bvs (nat-to-bv-list (x, ws)) = **nil**)

DEFINITION:
nat-listp-listp-simple (x)
$=$ **if** listp (x)
then nat-listp-simple (car (x)) $\wedge$ nat-listp-listp-simple (cdr (x))
else $x =$ **nil** **endif**

THEOREM: non-green-in-list-zerop-ws
$((ws \simeq 0) \wedge$ nat-listp-listp-simple $(x))$
$\rightarrow$ (non-green-in-list $(x,\ ws) \leftrightarrow$ member-number-with-at-least $(x,\ 2)$)

THEOREM: nat-listp-listp-simple-means-properp
nat-listp-listp-simple $(x) \rightarrow$ properp (x)

THEOREM: nat-listp-simple-means-properp
nat-listp-simple $(x) \rightarrow$ properp (x)

THEOREM: make-list-from-xor-bvs-nat-to-bv-list
(fix $(ws) =$ fix (s))
$\rightarrow$ (make-list-from $(ws,$ xor-bvs (nat-to-bv-list $(x,\ s)$))
$=$ **if** listp (x) **then** xor-bvs (nat-to-bv-list $(x,\ s)$)
else zero-bit-vector (ws) **endif**)

THEOREM: last-xor-bitv
last (xor-bitv $(x,\ y)$)
$=$ **if** listp (x) **then** xor-bit (last (x), nth (length $(x) - 1,\ y$))
else 0 **endif**

THEOREM: last-one-bit-vector
last (one-bit-vector (x)) $= 1$

THEOREM: nth-as-last
$((1 + n) =$ length $(x)) \rightarrow$ (nth $(n,\ x) =$ last (x))

THEOREM: listp-nat-to-bv
listp (nat-to-bv $(x,\ s)$) $= (s \not\simeq 0)$

```
;(prove-lemma remainder-plus-x-x-2 (rewrite)
;      (equal (remainder (plus x x) 2) 0))
```

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

THEOREM: last-nat-to-bv
last (nat-to-bv $(x,\ s)$)
$=$ **if** $(s \simeq 0) \vee ((x <$ exp $(2,\ s)) \wedge ((x$ **mod** $2) = 0))$ **then** 0
else 1 **endif**

THEOREM: fix-bitv-one-bit-vector
fix-bitv (one-bit-vector (x)) = one-bit-vector (x)

THEOREM: nat-to-bv-1
nat-to-bv (1, x)
= **if** $x \simeq 0$ **then nil**
else one-bit-vector (x) **endif**

THEOREM: last-zero-bit-vector
last (zero-bit-vector (x)) = 0

THEOREM: nat-to-bv-2
nat-to-bv (x, 2)
= **if** $x \simeq 0$ **then** list (0, 0)
elseif $x = 1$ **then** list (0, 1)
elseif $x = 2$ **then** list (1, 0)
else list (1, 1) **endif**

THEOREM: all-zero-bitvp-all-but-last-nat-to-bv
all-zero-bitvp (all-but-last (nat-to-bv (x, s))) = (($x < 2$) $\vee$ ($s < 2$))

THEOREM: xor-bvs-of-list-of-0s-and-1s
(number-with-at-least (c, 2) $\simeq$ 0)
$\rightarrow$ (xor-bvs (nat-to-bv-list (c, ws))
= **if** ($c \simeq$ **nil**) $\vee$ ($ws \simeq 0$) **then nil**
elseif (sum (c) **mod** 2) = 0 **then** zero-bit-vector (ws)
else one-bit-vector (ws) **endif**)

THEOREM: equal-nat-to-bv-nlistp
($x \simeq$ **nil**) $\rightarrow$ ((x = nat-to-bv (y, s)) = ((x = **nil**) $\wedge$ ($s \simeq 0$)))

THEOREM: different-lengths-means-different-hack
(fix ($s1$) $\neq$ fix ($s2$))
$\rightarrow$ (length (nat-to-bv (x, $s1$)) $\neq$ length (nat-to-bv (y, $s2$)))

THEOREM: different-lengths-means-different
(fix ($s1$) $\neq$ fix ($s2$)) $\rightarrow$ (nat-to-bv (x, $s1$) $\neq$ nat-to-bv (y, $s2$))

DEFINITION:
nat-to-bv-induct (x, y, $s1$, $s2$)
= **if** $s1 \simeq 0$ **then t**
else nat-to-bv-induct (**if** $x <$ exp (2, $s1 - 1$) **then** x
else $x -$ exp (2, $s1 - 1$) **endif**,
if $y <$ exp (2, $s2 - 1$) **then** y
else $y -$ exp (2, $s2 - 1$) **endif**,
$s1 - 1$,
$s2 - 1$) **endif**

DEFINITION:
all-ones-vector (x)
$=$ **if** $x \simeq 0$ **then nil**
else cons $(1,$ all-ones-vector $(x - 1))$ **endif**

THEOREM: not-lessp-exp-means-all-ones
$(x \not< (\exp(2, s) - 1)) \rightarrow (\text{nat-to-bv}(x, s) = \text{all-ones-vector}(s))$

THEOREM: lessp-sub1-plus-sub1-hack
$(y \not\simeq 0) \rightarrow ((((x + y) - 1) < ((y - 1) + z)) = (x < z))$

THEOREM: equal-cons-zero-bit-vector-nat-to-bv
$(\text{cons}(0, \text{zero-bit-vector}(x)) = \text{nat-to-bv}(a, b))$
$= \quad (((1 + x) = \text{fix}(b)) \wedge (a \simeq 0))$

THEOREM: equal-all-ones-vector-all-ones-vector
$(\text{all-ones-vector}(x) = \text{all-ones-vector}(y)) = (\text{fix}(x) = \text{fix}(y))$

THEOREM: equal-all-ones-vector-cons
$(\text{all-ones-vector}(x) = \text{cons}(a, b))$
$= \quad ((x \not\simeq 0) \wedge (a = 1) \wedge (\text{all-ones-vector}(x - 1) = b))$

THEOREM: different-lengths-obvious
$(x = y) \rightarrow (\text{length}(x) = \text{length}(y))$

THEOREM: equal-all-ones-vector-nlistp
$(x \simeq \mathbf{nil}) \rightarrow ((\text{all-ones-vector}(y) = x) = ((x = \mathbf{nil}) \wedge (y \simeq 0)))$

THEOREM: length-all-ones-vector
$\text{length}(\text{all-ones-vector}(x)) = \text{fix}(x)$

THEOREM: different-lengths-hack
$(\text{fix}(x) \neq \text{fix}(y)) \rightarrow (\text{all-ones-vector}(x) \neq \text{nat-to-bv}(a, y))$

THEOREM: lessp-difference-arg1
$(x \not< y) \rightarrow (((x - y) < z) = (x < (y + z)))$

THEOREM: equal-difference
$(x \not< y) \rightarrow (((x - y) = z) = ((\text{fix}(x) = (y + z)) \wedge (z \in \mathbf{N})))$

THEOREM: equal-exp
$(\text{fix}(a) = \text{fix}(b))$
$\rightarrow \quad ((\exp(a, c) = \exp(b, d))$
$\quad = \quad ((a = 1)$
$\quad\quad \vee \quad ((a \simeq 0) \wedge ((c \simeq 0) = (d \simeq 0)))$
$\quad\quad \vee \quad (\text{fix}(c) = \text{fix}(d))))$

THEOREM: equal-all-ones-nat-to-bv
$(\text{all-ones-vector}(x) = \text{nat-to-bv}(a,\ b))$
$= \quad ((\text{fix}(x) = \text{fix}(b)) \wedge (a \not< (\exp(2,\ b) - 1)))$

THEOREM: equal-nat-to-bv-nat-to-bv
$(\text{nat-to-bv}(x,\ s1) = \text{nat-to-bv}(y,\ s2))$
$= \quad ((\text{fix}(s1) = \text{fix}(s2))$
$\quad \wedge \quad ((\text{fix}(x) = \text{fix}(y))$
$\quad\quad \vee \quad ((x \not< (\exp(2,\ s1) - 1)) \wedge (y \not< (\exp(2,\ s1) - 1)))))$

THEOREM: listp-xor-bvs
$\text{listp}(\text{xor-bvs}(x)) = \text{listp}(\text{car}(x))$

THEOREM: car-nat-to-bv-list
$\text{car}(\text{nat-to-bv-list}(x,\ s))$
$= \quad$ **if** $\text{listp}(x)$ **then** $\text{nat-to-bv}(\text{car}(x),\ s)$
else 0 **endif**

```
;; from later version of naturals that is used in this proof
```

THEOREM: quotient-difference
$((x - y) \div z)$
$= \quad$ **if** $(x$ **mod** $z) < (y$ **mod** $z)$ **then** $((x \div z) - (y \div z)) - 1$
else $(x \div z) - (y \div z)$ **endif**

EVENT: Enable difference-x-sub1-x.

THEOREM: all-but-last-nat-to-bv
$\text{all-but-last}(\text{nat-to-bv}(x,\ s))$
$= \quad$ **if** $s \simeq 0$ **then nil**
else $\text{nat-to-bv}(x \div 2,\ s - 1)$ **endif**

THEOREM: equal-last-xor-bvs-1
$(\text{last}(\text{xor-bvs}(x)) = 1) = (\text{last}(\text{xor-bvs}(x)) \neq 0)$

THEOREM: not-green-state-means
$((\neg\ \text{green-statep}(\text{append}(a,\ \text{cons}(b,\ c)),\ ws))$
$\wedge \quad (d < \exp(2,\ ws))$
$\wedge \quad (b < d))$
$\rightarrow \quad (\text{green-statep}(\text{append}(a,\ \text{cons}(d,\ c)),\ ws)$
$\quad = \quad ((ws \not\simeq 0)$
$\quad\quad \vee \quad (\text{number-with-at-least}(\text{append}(a,\ \text{cons}(d,\ c)),\ 2) \simeq 0)))$

THEOREM: green-in-list-all-valid-moves-helper
$(\text{nat-listp}(a,\ ws)$

$\wedge$ nat-listp $(d,\ ws)$
$\wedge$ $(c \in \mathbf{N})$
$\wedge$ $(c < \exp(\texttt{2},\ ws))$
$\wedge$ properp (a)
$\wedge$ non-green-in-list (all-valid-moves-helper $(a,\ b,\ c,\ d),\ ws)$
$\wedge$ $(c \not< b)$
$\wedge$ $(b \in \mathbf{N}))$
$\rightarrow$ green-statep (append $(a$, cons $(c,\ d)),\ ws)$

Theorem: green-in-list-all-valid-moves-means
(nat-listp $(x,\ ws)$ $\wedge$ non-green-in-list (all-valid-moves $(x),\ ws)$)
$\rightarrow$ green-statep $(x,\ ws)$

Definition:
valid-movep $(s1,\ s2)$
$=$ **if** listp $(s1)$ $\wedge$ listp $(s2)$
then ((car $(s2)$ $<$ car $(s1)$)
$\wedge$ (car $(s2)$ $\in \mathbf{N}$)
$\wedge$ (cdr $(s1)$ $=$ cdr $(s2)$)))
$\vee$ ((car $(s1)$ $=$ car $(s2)$) $\wedge$ valid-movep (cdr $(s1)$, cdr $(s2)$)))
else f endif

Definition:
match-member $(m,\ list)$
$=$ **if** listp $(list)$
then if $\neg$ all-zero-bitvp (and-bitv (car $(list)$, m)) **then** car $(list)$
else match-member (m, cdr $(list)$) **endif**
else f endif

Theorem: xor-bvs-match-and-xor
bit-vectorsp ($list$, length $(value)$)
$\rightarrow$ (xor-bvs (match-and-xor $(list,\ match,\ value)$)
$=$ **if** match-member $(match,\ list)$
then xor-bitv ($value$, xor-bvs $(list)$)
else xor-bvs $(list)$ **endif**)

Definition:
remove-highest-bits (x)
$=$ **if** listp (x) **then** cons (cdar (x), remove-highest-bits (cdr (x)))
else nil endif

Theorem: car-remove-highest-bits
car (remove-highest-bits (x)) $=$ cdr (car (x))

Theorem: equal-car-highest-bit-1
(car (highest-bit (x)) $=$ $\texttt{1}$)
$=$ (highest-bit (x) $=$ cons ($\texttt{1}$, zero-bit-vector (length (x) $-$ 1)))

THEOREM: car-xor-bitv
$\text{car}(\text{xor-bitv}(x, y))$
$= \ \textbf{if } \text{listp}(x) \ \textbf{then } \text{xor-bit}(\text{car}(x), \text{car}(y))$
$\quad \textbf{else } \mathtt{0} \ \textbf{endif}$

THEOREM: match-member-cons-0
$\text{match-member}(\text{cons}(\mathtt{0}, x), y) \leftrightarrow \text{match-member}(x, \text{remove-highest-bits}(y))$

THEOREM: match-member-cons
$(\text{bit-vectorsp}(v, \text{length}(\text{cons}(a, b))) \land (\text{car}(\text{xor-bvs}(v)) \neq \mathtt{0}))$
$\rightarrow \ (\text{match-member}(\text{cons}(a, b), v)$
$\quad \leftrightarrow \ ((a \neq \mathtt{0}) \lor \text{match-member}(b, \text{remove-highest-bits}(v))))$

THEOREM: equal-highest-bit-cons-1
$(\text{highest-bit}(x) = \text{cons}(\mathtt{1}, y))$
$= \ ((\text{car}(x) \neq \mathtt{0}) \land (y = \text{zero-bit-vector}(\text{length}(x) - 1)))$

THEOREM: length-cdr-xor-bitv
$\text{length}(\text{cdr}(\text{xor-bitv}(x, y))) = \text{length}(\text{cdr}(x))$

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

THEOREM: length-cdr-xor-bvs
$\text{bit-vectorsp}(x, s)$
$\rightarrow \ (\text{length}(\text{cdr}(\text{xor-bvs}(x)))$
$\quad = \ \textbf{if } \text{listp}(x) \ \textbf{then } s - 1$
$\quad\quad \textbf{else } \mathtt{0} \ \textbf{endif})$

DEFINITION:
$\text{highest-bits-induct}(x, s)$
$= \ \textbf{if } \text{listp}(x)$
$\quad \textbf{then if } \text{listp}(\text{car}(x))$
$\quad\quad \textbf{then if } \text{car}(\text{highest-bit}(\text{xor-bvs}(x))) = \mathtt{1} \ \textbf{then t}$
$\quad\quad\quad \textbf{else } \text{highest-bits-induct}(\text{remove-highest-bits}(x), s - 1) \ \textbf{endif}$
$\quad\quad \textbf{else t endif}$
$\quad \textbf{else t endif}$

THEOREM: bit-vectorsp-remove-highest-bits
$\text{bit-vectorsp}(x, 1 + s) \rightarrow \text{bit-vectorsp}(\text{remove-highest-bits}(x), s)$

THEOREM: xor-bvs-remove-highest-bits
$\text{bit-vectorsp}(x, s)$
$\rightarrow \ (\text{xor-bvs}(\text{remove-highest-bits}(x))$
$\quad = \ \textbf{if } \text{listp}(x) \land (s \not\simeq \mathtt{0}) \ \textbf{then } \text{cdr}(\text{xor-bvs}(x))$
$\quad\quad \textbf{else nil endif})$

THEOREM: match-member-highest-bit-xor-bvs
bit-vectorsp $(x,\ s)$
$\rightarrow$ (match-member (highest-bit (xor-bvs (x)), x)
$\leftrightarrow$ ($\neg$ all-zero-bitvp (xor-bvs (x))))

THEOREM: match-member-highest-bit-xor-bvs-rewrite
bit-vectorsp (x, length (xor-bvs (x)))
$\rightarrow$ (match-member (highest-bit (xor-bvs (x)), x)
$\leftrightarrow$ ($\neg$ all-zero-bitvp (xor-bvs (x))))

THEOREM: bit-vectorsp-nat-to-bv-list-better
bit-vectorsp (nat-to-bv-list (x, *size*), *size2*)
$=$ (($x \simeq$ **nil**) $\vee$ (fix (*size*) $=$ fix (*size2*)))

THEOREM: match-member-at-least-min-means
(match-member (y, nat-to-bv-list (x, *ws*))
$\wedge$ (bv-to-nat (match-member (y, nat-to-bv-list (x, *ws*))) $\not<$ *min*))
$\rightarrow$ (number-with-at-least (x, *min*) $\neq$ 0)

THEOREM: bit-vectorp-highest-bit-xor-bvs
bit-vectorp (highest-bit (xor-bvs (x)), *ws*) $=$ bit-vectorp (xor-bvs (x), *ws*)

THEOREM: number-with-at-least-match-and-xor
(nat-listp (x, *ws*) $\wedge$ bit-vectorp (y, *ws*) $\wedge$ bit-vectorp (z, *ws*))
$\rightarrow$ (number-with-at-least (bv-to-nat-list (match-and-xor (nat-to-bv-list (x,
ws),
y,
z)),
min)
$=$ **if** match-member (y, nat-to-bv-list (x, *ws*))
then (number-with-at-least (x, *min*)
$+$ **if** bv-to-nat (xor-bitv (z,
match-member (y,
nat-to-bv-list (x,
ws))))
$<$ *min* **then** 0
else 1 **endif**)
$-$ **if** bv-to-nat (match-member (y, nat-to-bv-list (x, *ws*)))
$<$ *min* **then** 0
else 1 **endif**
else number-with-at-least (x, *min*) **endif**)

THEOREM: number-with-at-least-replace-value
number-with-at-least (replace-value (x, e, v), *min*)
$=$ **if** $e \in x$

then (number-with-at-least $(x,\ min)$
$\quad + \quad$ **if** $v < min$ **then** $\mathtt{0}$
$\quad\quad$ **else** $\mathtt{1}$ **endif**)
$\quad - \quad$ **if** $e < min$ **then** $\mathtt{0}$
$\quad\quad$ **else** $\mathtt{1}$ **endif**
else number-with-at-least $(x,\ min)$ **endif**

THEOREM: max-list-means-number-0
$((\text{max-list}\,(x) = n) \wedge (n < m)) \rightarrow (\text{number-with-at-least}\,(x,\ m) = \mathtt{0})$

THEOREM: member-max-list
$\text{nat-listp-simple}\,(x) \rightarrow ((\text{max-list}\,(x) \in x) = \text{listp}\,(x))$

THEOREM: listp-replace-value
$\text{listp}\,(\text{replace-value}\,(x,\ e,\ v)) = \text{listp}\,(x)$

THEOREM: member-means-lessp-sum
$(e \in x) \rightarrow (\text{sum}\,(x) \not< e)$

THEOREM: sum-replace-value
sum (replace-value $(x,\ e,\ v)$)
$=$ **if** $e \in x$ **then** $(\text{sum}\,(x) + v) - e$
else sum (x) **endif**

THEOREM: remainder-difference-2
$((x - y)$ **mod** $2)$
$=$ **if** $x < y$ **then** $\mathtt{0}$
elseif $(x$ **mod** $2) = (y$ **mod** $2)$ **then** $\mathtt{0}$
else $\mathtt{1}$ **endif**

THEOREM: lessp-max-list
$\text{sum}\,(x) \not< \text{max-list}\,(x)$

THEOREM: remainder-plus-remainder
$((((x \textbf{ mod } y) + z) \textbf{ mod } y) = ((x + z) \textbf{ mod } y))$
$\wedge \quad (((z + (x \textbf{ mod } y)) \textbf{ mod } y) = ((x + z) \textbf{ mod } y))$

THEOREM: remainder-plus-remainder2
$(((z + (a + (x \textbf{ mod } y))) \textbf{ mod } y) = ((z + (a + x)) \textbf{ mod } y))$
$\wedge \quad (((z + ((x \textbf{ mod } y) + a)) \textbf{ mod } y) = ((z + (a + x)) \textbf{ mod } y))$

THEOREM: lessp-max-list-from-number-with-at-least
$((\text{number-with-at-least}\,(x,\ m) = \mathtt{0}) \wedge (m \not\simeq \mathtt{0})) \rightarrow (\text{max-list}\,(x) < m)$

THEOREM: number-with-at-least-as-sum
$(\text{number-with-at-least}\,(x,\ \mathtt{2}) \simeq \mathtt{0}) \rightarrow (\text{number-with-at-least}\,(x,\ \mathtt{1}) = \text{sum}\,(x))$

THEOREM: equal-remainder-add1-2
$(((1 + x) \textbf{ mod } 2) = ((1 + y) \textbf{ mod } 2)) = ((x \textbf{ mod } 2) = (y \textbf{ mod } 2))$

THEOREM: remainder-plus-sum-number-hack
$(\text{number-with-at-least}\,(x,\, 2) = 1)$
$\rightarrow \quad (((\text{sum}\,(x) + \text{number-with-at-least}\,(x,\, 1)) \textbf{ mod } 2)$
$\qquad = \quad ((1 + \text{max-list}\,(x)) \textbf{ mod } 2))$

THEOREM: equal-remainder-add1
$(((1 + x) \textbf{ mod } y) = (x \textbf{ mod } y)) = (y = 1)$

THEOREM: max-0-means-sum-0
$(\text{sum}\,(x) = 0) = (\text{max-list}\,(x) = 0)$

THEOREM: max-0-means-all-zero-bitvp
$((\text{max-list}\,(x) = 0) \land \text{nat-listp-simple}\,(x)) \rightarrow \text{all-zero-bitvp}\,(x)$

THEOREM: remainder-add1-2
$((((1 + x) \textbf{ mod } 2) = 0) = ((x \textbf{ mod } 2) = 1))$
$\land \quad ((((1 + x) \textbf{ mod } 2) = 1) = ((x \textbf{ mod } 2) = 0))$

THEOREM: remainder-sub1-2
$(x \not\simeq 0)$
$\rightarrow \quad (((((x - 1) \textbf{ mod } 2) = 0) = ((x \textbf{ mod } 2) = 1))$
$\qquad \land \quad ((((x - 1) \textbf{ mod } 2) = 1) = ((x \textbf{ mod } 2) = 0)))$

THEOREM: equal-x-remainder-sub1-x
$(((x - 1) \textbf{ mod } y) = x) = (x = 0)$

THEOREM: computer-move-makes-non-green
$(\text{green-statep}\,(x,\, ws)$
$\land \quad \text{nat-listp}\,(x,\, ws)$
$\land \quad \text{listp}\,(x)$
$\land \quad (ws \not\simeq 0)$
$\land \quad (\neg\ \text{all-zero-bitvp}\,(x)))$
$\rightarrow \quad (\neg\ \text{green-statep}\,(\text{computer-move}\,(x,\, ws),\, ws))$

THEOREM: nat-listp-simplify
$(ws \simeq 0) \rightarrow (\text{nat-listp}\,(x,\, ws) = (\text{properp}\,(x) \land \text{all-zero-bitvp}\,(x)))$

THEOREM: computer-move-makes-non-green-rewrite
$(\text{green-statep}\,(x,\, ws) \land \text{nat-listp}\,(x,\, ws) \land (\neg\ \text{all-zero-bitvp}\,(x)))$
$\rightarrow \quad (\neg\ \text{green-statep}\,(\text{computer-move}\,(x,\, ws),\, ws))$

DEFINITION:
$\text{make-properp}\,(x)$
$= \quad \textbf{if } \text{listp}\,(x) \textbf{ then } \text{cons}\,(\text{car}\,(x),\, \text{make-properp}\,(\text{cdr}\,(x)))$
$\qquad \textbf{else nil endif}$

THEOREM: properp-make-properp
$\text{properp}(x) \rightarrow (\text{make-properp}(x) = x)$

THEOREM: replace-value-simplify
$(x \notin y) \rightarrow (\text{replace-value}(y, x, z) = \text{make-properp}(y))$

THEOREM: member-make-properp
$(x \in \text{make-properp}(y)) = (x \in y)$

THEOREM: valid-movep-x-x
$\neg\ \text{valid-movep}(x, x)$

THEOREM: valid-movep-replace-value
$\text{properp}(x)$
$\rightarrow \quad (\text{valid-movep}(x, \text{replace-value}(x, y, z))$
$\qquad = \quad ((y \in x) \wedge (z < y) \wedge (z \in \mathbf{N})))$

THEOREM: number-with-at-least-max-list
$((\text{number-with-at-least}(x, m) = v) \wedge (\mathtt{0} < v)) \rightarrow (\text{max-list}(x) \not< m)$

THEOREM: valid-movep-match-and-xor
$(\text{nat-listp}(x, ws) \wedge \text{bit-vectorp}(y, ws) \wedge \text{bit-vectorp}(z, ws))$
$\rightarrow \quad (\text{valid-movep}(x,$
$\qquad\qquad\qquad \text{bv-to-nat-list}(\text{match-and-xor}(\text{nat-to-bv-list}(x, ws), y, z)))$
$\qquad = \quad (\text{match-member}(y, \text{nat-to-bv-list}(x, ws))$
$\qquad\qquad \wedge \quad (\text{bv-to-nat}(\text{xor-bitv}(z,$
$\qquad\qquad\qquad\qquad\qquad \text{match-member}(y,$
$\qquad\qquad\qquad\qquad\qquad\qquad \text{nat-to-bv-list}(x, ws))))$
$\qquad\qquad\qquad < \quad \text{bv-to-nat}(\text{match-member}(y, \text{nat-to-bv-list}(x, ws))))))$

DEFINITION:
$\text{lessp-bv}(x, y)$
$= \quad \textbf{if } \text{listp}(x) \wedge \text{listp}(y)$
$\qquad \textbf{then if } \text{fix-bit}(\text{car}(x)) = \text{fix-bit}(\text{car}(y))$
$\qquad\qquad \textbf{then } \text{lessp-bv}(\text{cdr}(x), \text{cdr}(y))$
$\qquad\qquad \textbf{else } \text{car}(x) = \mathtt{0} \textbf{ endif}$
$\qquad \textbf{else f endif}$

THEOREM: lessp-bv-to-nat
$\text{bv-to-nat}(x) < \exp(\mathtt{2}, \text{length}(x))$

THEOREM: lessp-as-lessp-bv
$(\text{length}(x) = \text{length}(y))$
$\rightarrow \quad ((\text{bv-to-nat}(x) < \text{bv-to-nat}(y)) = \text{lessp-bv}(x, y))$

THEOREM: fix-bitv-highest-bit
$\text{fix-bitv}(\text{highest-bit}(x)) = \text{highest-bit}(x)$

THEOREM: properp-highest-bit
properp (highest-bit (x))

THEOREM: bit-vectorp-fix-bitv
bit-vectorp (fix-bitv (x), s) = (length (x) = fix (s))

THEOREM: lessp-bv-xor-bitv
(length (x) = length (y))
$\rightarrow$ (lessp-bv (xor-bitv (x, y), y)
= ($\neg$ all-zero-bitvp (and-bitv (y, highest-bit (x)))))

THEOREM: length-match-member-nat-to-bv-list
length (match-member (a, nat-to-bv-list (x, ws)))
= **if** match-member (a, nat-to-bv-list (x, ws)) **then** fix (ws)
else 0 **endif**

THEOREM: bit-vectorsp-remove-highest-bits2
bit-vectorsp (x, $s1$)
$\rightarrow$ (bit-vectorsp (remove-highest-bits (x), $s2$)
= ((($1 + s2$) = $s1$) $\vee$ ($\neg$ listp (x))))

THEOREM: match-member-high-bit-xor-bvs-helper
bit-vectorsp (x, ws)
$\rightarrow$ (match-member (highest-bit (xor-bvs (x)), x)
$\leftrightarrow$ ($\neg$ all-zero-bitvp (xor-bvs (x))))

THEOREM: match-member-high-bit-xor-bvs
match-member (highest-bit (xor-bvs (nat-to-bv-list (y, ws))), nat-to-bv-list (y, ws))
$\leftrightarrow$ ($\neg$ all-zero-bitvp (xor-bvs (nat-to-bv-list (y, ws))))

THEOREM: length-match-member
match-member (a, nat-to-bv-list (x, ws))
$\rightarrow$ (length (match-member (a, nat-to-bv-list (x, ws))) = fix (ws))

THEOREM: all-zero-bitvp-and-match-member
bit-vectorsp (b, length (a))
$\rightarrow$ (all-zero-bitvp (and-bitv (a, match-member (a, b)))
= ($\neg$ match-member (a, b)))

THEOREM: valid-movep-computer-move-helper
(nat-listp (x, ws) $\wedge$ ($\neg$ all-zero-bitvp (x)) $\wedge$ ($ws \not\simeq 0$) $\wedge$ listp (x))
$\rightarrow$ valid-movep (x, computer-move (x, ws))

`;;; PART OF SPECIFICATION`

THEOREM: valid-movep-computer-move
(nat-listp (x, ws) $\wedge$ ($\neg$ all-zero-bitvp (x)))
$\rightarrow$ valid-movep (x, computer-move (x, ws))

THEOREM: nthcdr-cdr
$\text{nthcdr}\,(n,\ \text{cdr}\,(x)) = \text{cdr}\,(\text{nthcdr}\,(n,\ x))$

THEOREM: make-list-from-append
$\text{make-list-from}\,(n,\ \text{append}\,(a,\ b))$
$= \ \ \textbf{if}\ \text{length}\,(a) < n\ \ \textbf{then}\ \text{append}\,(a,\ \text{make-list-from}\,(n - \text{length}\,(a),\ b))$
$\quad\ \ \textbf{else}\ \text{make-list-from}\,(n,\ a)\ \textbf{endif}$

THEOREM: make-list-from-simplify-better
$(n = \text{length}\,(x)) \rightarrow (\text{make-list-from}\,(n,\ x) = \text{make-properp}\,(x))$

```
;(prove-lemma length-cons (rewrite)
;      (equal (length (cons a b)) (add1 (length b))))
```

EVENT: Enable length-cons.

THEOREM: cdr-nthcdr-cons
$\text{cdr}\,(\text{nthcdr}\,(n,\ \text{cons}\,(a,\ b))) = \text{nthcdr}\,(n,\ b)$

THEOREM: equal-append-2
$(x = \text{append}\,(a,\ b))$
$= \ \ ((\text{length}\,(x) \not< \text{length}\,(a))$
$\quad\ \ \wedge\ \ (\text{make-list-from}\,(\text{length}\,(a),\ x) = \text{make-properp}\,(a))$
$\quad\ \ \wedge\ \ (\text{nthcdr}\,(\text{length}\,(a),\ x) = b))$

THEOREM: member-cons-all-valid-moves-helper1
$\text{listp}\,(a)$
$\rightarrow\ \ ((\text{cons}\,(x,\ y) \in \text{all-valid-moves-helper}\,(a,\ b,\ c,\ d))$
$\quad\ \ =\ \ ((x = \text{car}\,(a))$
$\quad\quad\ \ \wedge\ \ (y \in \text{all-valid-moves-helper}\,(\text{cdr}\,(a),\ b,\ c,\ d))))$

THEOREM: member-all-valid-moves-means-prefix
$\text{properp}\,(a)$
$\rightarrow\ \ ((x \in \text{all-valid-moves-helper}\,(a,\ b,\ c,\ d))$
$\quad\ \ \rightarrow\ \ (\text{make-list-from}\,(\text{length}\,(a),\ x) = a))$

THEOREM: equal-nthcdr-cons
$(n \not< \text{length}\,(x)) \rightarrow (\text{nthcdr}\,(n,\ x) \neq \text{cons}\,(a,\ b))$

THEOREM: lessp-length-simple-member-all-valid-moves
$(\text{length}\,(a) \not< \text{length}\,(x)) \rightarrow (x \notin \text{all-valid-moves-helper}\,(a,\ b,\ c,\ d))$

THEOREM: nth-cons
$\text{nth}\,(n,\ \text{cons}\,(a,\ b))$
$= \ \ \textbf{if}\ n \simeq \texttt{0}\ \ \textbf{then}\ a$
$\quad\ \ \textbf{else}\ \text{nth}\,(n - 1,\ b)\ \textbf{endif}$

THEOREM: nth-1
$\text{nth}\,(1,\ x) = \text{cadr}\,(x)$

THEOREM: get-as-nth
$\text{get}\,(n,\ x) = \text{nth}\,(n,\ x)$

THEOREM: equal-cons-make-properp
$$\begin{array}{l} (\text{cons}\,(a,\ b) = \text{make-properp}\,(x)) \\ = \quad (\text{listp}\,(x) \land (a = \text{car}\,(x)) \land (b = \text{make-properp}\,(\text{cdr}\,(x)))) \end{array}$$

THEOREM: nthcdr-cons-make-list-from-hack
$$\begin{array}{l} \text{nthcdr}\,(n,\ \text{cons}\,(a,\ \text{make-list-from}\,(n,\ z))) \\ = \quad \textbf{if } n \simeq \mathtt{0} \ \textbf{ then } \text{list}\,(a) \\ \qquad \textbf{else } \text{list}\,(\text{nth}\,(n-1,\ z))\ \textbf{endif} \end{array}$$

THEOREM: lessp-sub1-as-equal
$(a < b) \rightarrow ((a < (b - 1)) = ((1 + a) \neq b))$

THEOREM: equal-nthcdr-cons-better
$$\begin{array}{l} (\text{nthcdr}\,(n,\ x) = \text{cons}\,(a,\ b)) \\ = \quad ((n < \text{length}\,(x)) \land (\text{nth}\,(n,\ x) = a) \land (\text{nthcdr}\,(1 + n,\ x) = b)) \end{array}$$

THEOREM: member-all-valid-moves-helper
$$\begin{array}{l} (\text{nat-listp-simple}\,(a) \land \text{nat-listp-simple}\,(d) \land (b \in \mathbf{N}) \land (c \in \mathbf{N})) \\ \rightarrow \quad ((x \in \text{all-valid-moves-helper}\,(a,\ b,\ c,\ d)) \\ \qquad = \quad ((\text{make-list-from}\,(\text{length}\,(a),\ x) = \text{make-properp}\,(a)) \\ \qquad\qquad \land \quad (((\text{nth}\,(\text{length}\,(a),\ x) < b) \\ \qquad\qquad\qquad \land \quad (\text{nth}\,(\text{length}\,(a),\ x) \in \mathbf{N}) \\ \qquad\qquad\qquad \land \quad (\text{nthcdr}\,(1 + \text{length}\,(a),\ x) = d)) \\ \qquad\qquad\quad \lor \quad ((\text{nth}\,(\text{length}\,(a),\ x) = c) \\ \qquad\qquad\qquad \land \quad \text{valid-movep}\,(d, \\ \qquad\qquad\qquad\qquad\qquad \text{cdr}\,(\text{nthcdr}\,(\text{length}\,(a),\ x))))))) \end{array}$$

THEOREM: member-all-valid-moves
$\text{nat-listp-simple}\,(x) \rightarrow ((y \in \text{all-valid-moves}\,(x)) = \text{valid-movep}\,(x,\ y))$

THEOREM: valid-movep-and-makes-nongreen-means
$((x \in y) \land (\neg\ \text{green-statep}\,(x,\ ws))) \rightarrow \text{non-green-in-list}\,(y,\ ws)$

THEOREM: green-means-non-green-in-valid-moves
$$\begin{array}{l} (\text{nat-listp}\,(s,\ ws) \land (\neg\ \text{all-zero-bitvp}\,(s)) \land \text{green-statep}\,(s,\ ws)) \\ \rightarrow \quad \text{non-green-in-list}\,(\text{all-valid-moves}\,(s),\ ws) \end{array}$$

THEOREM: green-in-list-all-valid-moves
$$\begin{array}{l} (\text{nat-listp}\,(s,\ ws) \land (\neg\ \text{all-zero-bitvp}\,(s))) \\ \rightarrow \quad (\text{non-green-in-list}\,(\text{all-valid-moves}\,(s),\ ws) \leftrightarrow \text{green-statep}\,(s,\ ws)) \end{array}$$

THEOREM: sum-when-all-zero
all-zero-bitvp $(x) \rightarrow (\text{sum}(x) = 0)$

THEOREM: green-statep-all-zero-bitvp
all-zero-bitvp $(x) \rightarrow$ green-statep (x, s)

THEOREM: wsp-green-state-proof

$$
\begin{array}{l}
(((\mathit{flag} \wedge \text{nat-listp}(s, \mathit{wordsize})) \\
\quad \vee \quad ((\neg \mathit{flag}) \wedge \text{nat-listp-listp}(s, \mathit{wordsize}))) \\
\ \wedge \quad \text{listp}(s)) \\
\rightarrow \quad (\text{wsp}(s, \mathit{flag}) \\
\qquad \leftrightarrow \quad \textbf{if } \mathit{flag} \ \textbf{ then } \text{green-statep}(s, \mathit{wordsize}) \\
\qquad\qquad \textbf{else } \text{non-green-in-list}(s, \mathit{wordsize}) \ \textbf{endif})
\end{array}
$$

THEOREM: wsp-green-state
nat-listp $(s, \mathit{wordsize}) \rightarrow (\text{wsp}(s, \mathbf{t}) \leftrightarrow \text{green-statep}(s, \mathit{wordsize}))$

THEOREM: nat-listp-replace-value
$(\text{nat-listp}(s, \mathit{ws}) \wedge (\mathit{new} < \exp(2, \mathit{ws})) \wedge (\mathit{new} \in \mathbf{N}))$
$\rightarrow \quad \text{nat-listp}(\text{replace-value}(s, y, \mathit{new}), \mathit{ws})$

THEOREM: nat-listp-bv-to-nat-list
bit-vectorsp $(x, s) \rightarrow$ nat-listp (bv-to-nat-list (x), s)

THEOREM: nat-listp-smart-move
$(\text{nat-listp}(s, \mathit{ws}) \wedge (\mathit{ws} \not\simeq 0)) \rightarrow \text{nat-listp}(\text{smart-move}(s, \mathit{ws}), \mathit{ws})$

THEOREM: all-zero-bitvp-max-list
all-zero-bitvp $(s) \rightarrow (\text{max-list}(s) = 0)$

THEOREM: replace-value-x-x
properp $(x) \rightarrow (\text{replace-value}(x, y, y) = x)$

THEOREM: smart-move-small-ws
$(\text{nat-listp}(s, \mathit{ws}) \wedge (\mathit{ws} \simeq 0)) \rightarrow (\text{smart-move}(s, \mathit{ws}) = s)$

THEOREM: lessp-max-list-from-nat-listp
nat-listp $(s, \mathit{ws}) \rightarrow (\text{max-list}(s) < \exp(2, \mathit{ws}))$

THEOREM: nat-listp-computer-move
nat-listp $(s, \mathit{ws}) \rightarrow$ nat-listp (computer-move (s, ws), ws)

`;; PART OF SPECIFICATION`

THEOREM: computer-move-works
$(\text{nat-listp}(\mathit{state}, \mathit{ws}) \wedge (\neg \text{ all-zero-bitvp}(\mathit{state})) \wedge \text{wsp}(\mathit{state}, \mathbf{t}))$
$\rightarrow \quad (\neg \text{ wsp}(\text{computer-move}(\mathit{state}, \mathit{ws}), \mathbf{t}))$

DEFINITION: NIM-PITON-CTRL-STK-REQUIREMENT = 25

DEFINITION: NIM-PITON-TEMP-STK-REQUIREMENT = 3

DEFINITION:
computer-move-implemented-input-conditionp ($p0$)

```
=   (listp (p-ctrl-stk (p0))
     ∧  (7 < p-word-size (p0))
     ∧  (1 < p-word-size (p0))
     ∧  at-least-morep (length (p-temp-stk (p0)),
                        NIM-PITON-TEMP-STK-REQUIREMENT,
                        p-max-temp-stk-size (p0))
     ∧  at-least-morep (p-ctrl-stk-size (p-ctrl-stk (p0)),
                        NIM-PITON-CTRL-STK-REQUIREMENT,
                        p-max-ctrl-stk-size (p0))
     ∧  (car (top (cddr (p-temp-stk (p0)))) = 'addr)
     ∧  (cddr (top (cddr (p-temp-stk (p0)))) = nil)
     ∧  listp (untag (top (cddr (p-temp-stk (p0)))))
     ∧  (cdr (untag (top (cddr (p-temp-stk (p0))))) = 0)
     ∧  definedp (car (untag (top (cddr (p-temp-stk (p0))))),
                  p-data-segment (p0))
     ∧  nat-list-piton (array (car (untag (top (cddr (p-temp-stk (p0))))),
                               p-data-segment (p0)),
                        p-word-size (p0))
     ∧  (car (top (cdr (p-temp-stk (p0)))) = 'nat)
     ∧  (cddr (top (cdr (p-temp-stk (p0)))) = nil)
     ∧  (length (array (car (untag (top (cddr (p-temp-stk (p0))))),
                        p-data-segment (p0)))
         =  untag (top (cdr (p-temp-stk (p0)))))
     ∧  (untag (top (cdr (p-temp-stk (p0)))) < exp (2, p-word-size (p0)))
     ∧  (untag (top (cdr (p-temp-stk (p0)))) ≄ 0)
     ∧  (car (top (p-temp-stk (p0))) = 'addr)
     ∧  (cddr (top (p-temp-stk (p0))) = nil)
     ∧  listp (untag (top (p-temp-stk (p0))))
     ∧  (cdr (untag (top (p-temp-stk (p0)))) = 0)
     ∧  definedp (car (untag (top (p-temp-stk (p0)))), p-data-segment (p0))
     ∧  array-pitonp (array (car (untag (top (p-temp-stk (p0)))),
                             p-data-segment (p0)),
                      untag (top (cdr (p-temp-stk (p0)))))
     ∧  (car (untag (top (p-temp-stk (p0))))
         ≠  car (untag (top (cddr (p-temp-stk (p0))))))
     ∧  (¬ all-zero-bitvp (untag-array (array (car (untag (top (cddr (p-temp-stk (p0))))),
                                               p-data-segment (p0))))))
```

```
;; cm-prog is the Nim program.  It may be disappointing to see that it is
```

```
;; a function of one argument rather than a constant, as programs ought to
;; be.  This is because we wish to use bit vectors in our program, and
;; because of a weakness in the Piton design there is no way to push
;; a bit-vector on the stack without knowing the word-size.  The only
;; subprogram that uses the word-size is push-1-vector, which is a
;; one-line program that pushes a one-vector onto the stack.
```

DEFINITION:
cm-prog (*word-size*)
= list (XOR-BVS-PROGRAM,
    push-1-vector-program (*word-size*),
    NAT-TO-BV-PROGRAM,
    BV-TO-NAT-PROGRAM,
    NUMBER-WITH-AT-LEAST-PROGRAM,
    HIGHEST-BIT-PROGRAM,
    MATCH-AND-XOR-PROGRAM,
    NAT-TO-BV-LIST-PROGRAM,
    BV-TO-NAT-LIST-PROGRAM,
    MAX-NAT-PROGRAM,
    REPLACE-VALUE-PROGRAM,
    SMART-MOVE-PROGRAM,
    DELAY-PROGRAM,
    COMPUTER-MOVE-PROGRAM)

EVENT: Disable computer-move-program.

EVENT: Disable *1*computer-move-program.

THEOREM: car-xor-bvs-program
car (XOR-BVS-PROGRAM) = `'xor-bvs`

THEOREM: car-push-1-vector-program
car (push-1-vector-program (*word-size*)) = `'push-1-vector`

THEOREM: car-nat-to-bv-program
car (NAT-TO-BV-PROGRAM) = `'nat-to-bv`

THEOREM: car-bv-to-nat-program
car (BV-TO-NAT-PROGRAM) = `'bv-to-nat`

THEOREM: car-number-with-at-least-program
car (NUMBER-WITH-AT-LEAST-PROGRAM) = `'number-with-at-least`

THEOREM: car-highest-bit-program
car (HIGHEST-BIT-PROGRAM) = `'highest-bit`

THEOREM: car-match-and-xor-program
car (MATCH-AND-XOR-PROGRAM) = `'match-and-xor`

THEOREM: car-nat-to-bv-list-program
car (NAT-TO-BV-LIST-PROGRAM) = `'nat-to-bv-list`

THEOREM: car-bv-to-nat-list-program
car (BV-TO-NAT-LIST-PROGRAM) = `'bv-to-nat-list`

THEOREM: car-max-nat-program
car (MAX-NAT-PROGRAM) = `'max-nat`

THEOREM: car-replace-value-program
car (REPLACE-VALUE-PROGRAM) = `'replace-value`

THEOREM: car-smart-move-program
car (SMART-MOVE-PROGRAM) = `'smart-move`

THEOREM: car-delay-program
car (DELAY-PROGRAM) = `'delay`

EVENT: Disable delay-program.

THEOREM: car-computer-move-program
car (COMPUTER-MOVE-PROGRAM) = `'computer-move`

THEOREM: equal-untag-array-tag-array-x-x
(untag-array (tag-array (l, x)) = x) = properp (x)

THEOREM: properp-replace-value
properp (x) $\rightarrow$ properp (replace-value (x, y, z))

THEOREM: properp-bv-to-nat-list
properp (bv-to-nat-list (x))

THEOREM: properp-nat-to-bv-list
properp (nat-to-bv-list (x, ws))

THEOREM: properp-computer-move
properp (x) $\rightarrow$ properp (computer-move (x, s))

THEOREM: properp-untag-array
properp (untag-array (x))

THEOREM: properp-tag-array
properp (tag-array (l, x))

THEOREM: computer-move-implemented

$$
\begin{array}{l}
((p0 = \text{p-state}\,(pc, \\
\qquad\qquad \textit{ctrl-stk}, \\
\qquad\qquad \text{cons}\,(wa, \text{cons}\,(np, \text{cons}\,(s, \textit{temp-stk}))), \\
\qquad\qquad \text{append}\,(\text{cm-prog}\,(\textit{word-size}), \textit{prog-segment}), \\
\qquad\qquad \textit{data-segment}, \\
\qquad\qquad \textit{max-ctrl-stk-size}, \\
\qquad\qquad \textit{max-temp-stk-size}, \\
\qquad\qquad \textit{word-size}, \\
\qquad\qquad \texttt{'run})) \\
\;\wedge\;\; (\text{p-current-instruction}\,(p0) = \texttt{'(call computer-move)}) \\
\;\wedge\;\; \text{computer-move-implemented-input-conditionp}\,(p0)) \\
\rightarrow\;\; \textbf{let}\ \textit{result}\ \textbf{be}\ \text{p}\,(p0, \\
\qquad\qquad\qquad \text{computer-move-clock}\,(\text{untag-array}\,(\text{array}\,(\text{car}\,(\text{untag}\,(s)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{data-segment})), \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{word-size})) \\
\qquad \textbf{in} \\
\qquad (\text{p-pc}\,(\textit{result}) = \text{add1-addr}\,(pc)) \\
\qquad \wedge\;\; (\text{p-psw}\,(\textit{result}) = \texttt{'run}) \\
\qquad \wedge\;\; (\text{untag-array}\,(\text{array}\,(\text{car}\,(\text{untag}\,(s)), \text{p-data-segment}\,(\textit{result}))) \\
\qquad\qquad = \;\; \text{computer-move}\,(\text{untag-array}\,(\text{array}\,(\text{car}\,(\text{untag}\,(s)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{data-segment})), \\
\qquad\qquad\qquad\qquad\qquad \textit{word-size}))\ \textbf{endlet}
\end{array}
$$

```
#|
A proof of some constant bounds on the computer-move-clock function
was developed that makes slight use of the proof-checker enhancement of
NQTHM.  For completeness, here is the final theorem of that
digression, with no proof included so that this script is executable
in NQTHM without the enhancement

             (implies (and (nat-listp state ws)
                           (lessp 0 ws)
                           (not (lessp 32 ws))
                           (lessp 1 (length state))
                           (not (lessp 6 (length state))))
                      (and (lessp 10000
                                  (computer-move-clock state ws))
                           (lessp (computer-move-clock state ws)
                                  20000)))

|#
```

THEOREM: nim-piton-space-reasonable

$1000 \not< ($NIM-PITON-CTRL-STK-REQUIREMENT
$+$ NIM-PITON-TEMP-STK-REQUIREMENT)

```
;; bind up defns for presentation purposes
```

DEFINITION:
good-non-empty-nim-statep (*state*, *ws*)
= (nat-listp (*state*, *ws*) $\wedge$ ($\neg$ all-zero-bitvp (*state*)))

THEOREM: valid-movep-computer-move-better
good-non-empty-nim-statep (*state*, *ws*)
$\rightarrow$ valid-movep (*state*, computer-move (*state*, *ws*))

THEOREM: computer-move-works-better
(good-non-empty-nim-statep (*state*, *ws*) $\wedge$ wsp (*state*, **t**))
$\rightarrow$ ($\neg$ wsp (computer-move (*state*, *ws*), **t**))

```
;;; An initial p-state to run the program on a particular NIM state, then
;;; enter an infinite loop.
```

DEFINITION:
EXAMPLE2-COMPUTER-MOVE-P-STATE
```
=   p-state ('(pc (main . 0)),
             '((nil (pc (main . 0)))),
             nil,
             cons ('(main nil nil
                    (push-constant (addr (arr . 0)))
                    (push-constant (nat 4))
                    (push-constant (addr (arr5 . 0)))
                    (call computer-move)
                    (push-constant (nat 1))
                    (push-constant (addr (flag . 0)))
                    (deposit)
                    (dl loop nil (jump loop))
                    (ret)),
                  cm-prog (32)),
             '((arr (nat 15) (nat 4) (nat 7) (nat 1))
               (arr5 (nat 3) (nat 4) (nat 2) (nat 1))
               (flag (nat 0))),
             30,
             10,
             32,
             'run)
```

```
;;; Extra event that shows that the program is compilable and loadable
```

```
;;; onto FM9001, and that the correctness lemma for the Piton interpreter
;;; therefore holds.   (ref: J's e-mail of 10 April 92.)
```

THEOREM: cm-prog-fm9001-loadable
let *p0* **be** EXAMPLE2-COMPUTER-MOVE-P-STATE
in
proper-p-statep (*p0*)
∧ p-loadablep (*p0*, 0)
∧ (p-word-size (*p0*) = 32) **endlet**

```
;;; Some events written by J Moore that produce an FM9001 image
```

DEFINITION:
pretty-load1 (*p0*, *offset*) = i->m (r->i (p->r (*p0*)), **nil**, *offset*)

DEFINITION:
pretty-vector1 (*lst*)
= **if** *lst* ≃ **nil** **then** 0
elseif car (*lst*) **then** cons (49, pretty-vector1 (cdr (*lst*)))
else cons (48, pretty-vector1 (cdr (*lst*))) **endif**

DEFINITION:
pretty-vector (*lst*) = pack (cons (66, pretty-vector1 (*lst*)))

DEFINITION:
pretty-vector-lst (*lst*)
= **if** *lst* ≃ **nil** **then nil**
else cons (pretty-vector (car (*lst*)), pretty-vector-lst (cdr (*lst*))) **endif**

DEFINITION:
pretty-state (*m*)
= list ('fm9001-state,
pretty-vector-lst (m-regs (*m*)),
m-c-flg (*m*),
m-v-flg (*m*),
m-n-flg (*m*),
m-z-flg (*m*),
pretty-vector-lst (m-mem (*m*)))

DEFINITION:
pretty-load (*p*, *offset*) = pretty-state (pretty-load1 (*p*, *offset*))

```
;; Using pretty-load in r-loop, we produce the following image:

#|
```

```
(LIST 'FM9001-STATE
      '(B00000000000000000000000000000000 B01001010001000000000000000000000
        B01001010001000000000000000000000 B11111010001000000000000000000000
        B00000000000000000000000000000000 B00000000000000000000000000000000
        B00000000000000000000000000000000 B00000000000000000000000000000000
        B00000000000000000000000000000000 B00000000000000000000000000000000
        B00000000000000000000000000000000 B00000000000000000000000000000000
        B00000000000000000000000000000000 B00000000000000000000000000000000
        B00000000000000000000000000000000 B11010000000000000000000000000000)
      F F F F
      '(B11100000000000000000000000000000 B00100000000000000000000000000000
        B11100000000000000000000000000000 B10000000000000000000000000000000
        B11000000000000000000000000000000 B00100000000000000000000000000000
        B01000000000000000000000000000000 B10000000000000000000000000000000
        B00000000000000000000000000000000 B10000000000100010000011111110000
        B01000000001000000000011111110000 B11111100001100010000011111110000
        B00000000000000000000000000000000 B11111100001100010000011111110000
        B00100000000000000000000000000000 B11111100001100010000011111110000
        B00100000000000000000000000000000 B11111100000100010000011111110000
        B10101000000000000000000000000000 B11111000001111000000011111110000
        B10110101110000000000000000000000 B11111100001100010000011111110000
        B10000000000000000000000000000000 B11111100001100010000011111110000
        B00010000000000000000000000000000 B11001100000010000000011111110000
        B11001100000010100000011111110000 B11111000001111000000011111110000
        B11011000000000000000000000000000 B11111000001111000000011111110000
        B11111000000000000000000000000000 B10000000000100000000011111110000
        B01001100001000000000011111110000 B01001100001111000000011111110000
        B10000000000100010000011111110000 B01000000001000000000011111110000
        B11001100000100010000011111110000 B11001100000100010000011111110000
        B11111100000010000000011111110000 B00000000000000000000000000000000
        B01000000000010000000011110000000 B00101000001100010000011111110000
        B11001100000010000000011111110000 B00101000001100010000011111110000
        B11111100000010000000011111110000 B10000000000000000000000000000000
        B01000000000010000000011110000000 B00101000001100010000011111110000
        B11001000001100100000011110100000 B11111100000010000000011111110000
        B10000000000000000000000000000000 B01000000000010000000011110000000
        B11001100000010100000011111110000 B11111100000010000000011111110000
        B10000000000000000000000000000000 B01000000000010000000011110000000
        B00101000001100010000011111110000 B11001100001010001000011111110000
        B11111100000010000000011111110000 B00011010000000000000000000000000
        B00100000001111000000111011110000 B11111100000010000000011111110000
        B10000000000000000000000000000000 B01000000000010000000011110000000
        B00101000001100010000011111110000 B11001000001100100000011110100000
        B11111100000010000000011111110000 B10000000000000000000000000000000
```

B01000000000010000000011110000000 B11001100000010100000011111110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011110000000 B00101000001100010000011111110000
B11111100001100010000011111110000 B10000000000000000000000000000000
B11001100000010000000011111110000 B00100000001100100000011110000000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011110000000 B11001000000010100000011111110000
B11001100000010000000011111110000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00100000001100100000011111010000
B11111000001110000000011111110000 B10101100000000000000000000000000
B11111000001110000000011111110000 B01011010000000000000000000000000
B10000000000100000000011111110000 B01001100001000000000011111110000
B01001100001110000000011111110000 B10000000000100010000011111110000
B01000000001000000000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11111000001110000000011111110000
B11000110000000000000000000000000 B10000000000100000000011111110000
B01001100001000000000011111110000 B01001100001110000000011111110000
B10000000000100010000011111110000 B01000000001000000000011111110000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11001100000100010000011111110000 B11111100000100010000011111110000
B10001110000000000000000000000000 B11111000001110000000011111110000
B10111010000000000000000000000000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011110000000
B11001100000010100000011111110000 B11111100000100010000011111110000
B10011110000000000000000000000000 B11111000001110000000011111110000
B10111010000000000000000000000000 B11001000001100100000011101010000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011110000000 B00101000001100010000011111110000
B11001100001010001000011111110000 B11111100000010000000011111110000
B01110101000000000000000000000000 B00100000001110000000011011110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011110000000 B00101000001100010000011111110000
B11001100000010000001011111110000 B00100000000010000001011101010000
B00100000001100010000011111110000 B11111100001100010000011111110000
B00000000000000000000000000000000 B11111100001100100000100011110000
B10000000000000000000000000000000 B11111100000010000000011111110000
B01000000000000000000000000000000 B01000000000010000000011110000000
B11001100000010100000011111110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011110000000
B11001100000010100000011111110000 B11111100000010000000011111110000
B01000000000000000000000000000000 B01000000000010000000011110000000
B00101000001100010000011111110000 B11001100001010001000011111110000
B11111100000010000000011111110000 B11000101000000000000000000000000

B001000000011110000000111011110000 B111111000000100000000111111110000
B100000000000000000000000000000000 B010000000000100000000111110000000
B001010000011000100000111111110000 B110011000000100000000111111110000
B001000000011001000000111110100000 B111111000000100000000111111110000
B100000000000000000000000000000000 B010000000000100000000111110000000
B001010000011000100000111111110000 B110010000011001000000111110000000
B111111000000100000000111111110000 B100000000000000000000000000000000
B010000000000100000000111110000000 B110011000000101000000111111110000
B111110000011110000000111111110000 B010111000000000000000000000000000
B111110000011110000000111111110000 B000011010000000000000000000000000
B100000000001000000000111111110000 B010011000010000000000111111110000
B010011000011110000000111111110000 B100000000001000100000111111110000
B010000000010000000000111111110000 B111111000001000100000111111110000
B000000000000000000000000000000000 B111111000001000100000111111110000
B000000000000000000000000000000000 B110011000001000100000111111110000
B111111000011000100000111111110000 B100000000000000000000000000000000
B111111000000100000000111111110000 B010000000000000000000000000000000
B010000000000100000000111110000000 B110011000000101000000111111110000
B111111000001000100000111111110000 B001000110000000000000000000000000
B111110000011110000000111111110000 B101110100000000000000000000000000
B111111000000100000000111111110000 B100000000000000000000000000000000
B010000000000100000000111110000000 B110011000000101000000111111110000
B111111000011000100000111111110000 B000000000000000000000000000000000
B111111000000100000000111111110000 B000000000000000000000000000000000
B010000000000100000000111110000000 B001010000011000100000111111110000
B111111000000100000000111111110000 B100000000000000000000000000000000
B010000000000100000000111110000000 B001010000011000100000111111110000
B110011000000100000000111111110000 B001000000011001000000111101100000
B110011000010100010000111111110000 B111111000000100000000111111110000
B011110110000000000000000000000000 B001000000011110000000111011110000
B111111000000100000000111111110000 B010000000000000000000000000000000
B010000000000100000000111110000000 B001010000011000100000111111110000
B110011000000100000000111111110000 B001000000011001000000111110000000
B111111000000100000000111111110000 B100000000000000000000000000000000
B010000000000100000000111110000000 B001010000011000100000111111110000
B110010000011001000000111110000000 B111111000000100000000111111110000
B100000000000000000000000000000000 B010000000000100000000111110000000
B110010000000101000000111111110000 B110011000010100010000111111110000
B111111000000100000000111111110000 B011011110000000000000000000000000
B001000000011110000000111011110000 B111111000000100000000111111110000
B010000000000000000000000000000000 B010000000000100000000111110000000
B001010000011000100000111111110000 B110010000011001000000111110000000
B111111000000100000000111111110000 B010000000000000000000000000000000
B010000000000100000000111110000000 B110011000000101000000111111110000

```
B1111100000111100000000111111110000 B0101001100000000000000000000000000
B1111100000111100000000111111110000 B0001111100000000000000000000000000
B1000000000010000000000111111110000 B0100110000100000000000111111110000
B0100110000111100000000111111110000 B1000000000001000100000111111110000
B0100000000100000000000111111110000 B1111110000010001000000111111110000
B0000000000000000000000000000000000 B1100110000010001000000111111110000
B1100110000010001000000111111110000 B1100110000010001000000111111110000
B1111110000110001000000111111110000 B0000000000000000000000000000000000
B1111110000001000000000111111110000 B1100000000000000000000000000000000
B0100000000001000000000111110000000 B1100100000000101000000111111110000
B1111110000001000000000111111110000 B0000000000000000000000000000000000
B0100000000001000000000111110000000 B0010100000110001000000111111110000
B1100110000001000000000111111110000 B0010100000110001000000111111110000
B1111110000001000000000111111110000 B0100000000000000000000000000000000
B0100000000001000000000111110000000 B0010100000110001000000111111110000
B1100110000001000000000111111110000 B0010000000110010000101111111000000
B1100100000110010000000111110100000 B1111110000110010000001000111110000
B1000000000000000000000000000000000 B1100110000101000100000111111110000
B1111110000001000000000111111110000 B0011100010000000000000000000000000
B0010000000111100000000110111110000 B1100100000110010000000111100000000
B1111110000001000000000111111110000 B1000000000000000000000000000000000
B0100000000001000000000111110000000 B0010100000110001000000111111110000
B1111110000001000000000111111110000 B1100000000000000000000000000000000
B0100000000001000000000111110000000 B0010100000110001000000111111110000
B1100100000110010000000111100000000 B1111110000001000000000111111110000
B1100000000000000000000000000000000 B0100000000001000000000111110000000
B1100100000000101000000111111110000 B1100110000001000000000111111110000
B0010000000110010000000111111000000 B1100110000101000100000111111110000
B1111110000001000000000111111110000 B1011110010000000000000000000000000
B0010000000111100000000110111110000 B1111110000001000000000111111110000
B0000000000000000000000000000000000 B0100000000001000000000111110000000
B0010100000110001000000111111110000 B1111110000110001000000111111110000
B1000000000000000000000000000000000 B1100110000001000000000111111110000
B0010000000110010000000111110000000 B1111110000001000000000111111110000
B0000000000000000000000000000000000 B0100000000001000000000111110000000
B1100110000000101000000111111110000 B1111100000111100000000111111110000
B0001000010000000000000000000000000 B1111100000111100000000111111110000
B1111110010000000000000000000000000 B1000000000010000000000111111110000
B0100110000100000000000111111110000 B0100110000111100000000111111110000
B1000000000001000100000111111110000 B0100000000100000000000111111110000
B1111110000010001000000111111110000 B0000000000000000000000000000000000
B1100110000010001000000111111110000 B1111110000010001000000111111110000
B1101001010000000000000000000000000 B1111100000111100000000111111110000
B1011101000000000000000000000000000 B1111110000001000000000111111110000
```

B10000000000000000000000000000000
B11001000000010100000011111110000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001100001010001000011111110000
B01101110100000000000000000000000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001100000010000000011111110000
B11001100001010001000011111110000
B11010110100000000000000000000000
B11001100000010000000011111110000
B10000000000000000000000000000000
B00101000001100010000011111110000
B10000000000000000000000000000000
B00101000001100010000011111110000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11111000001110000000011111110000
B11111000001110000000011111110000
B10000000000100000000011111110000
B01001100001110000000011111110000
B01000000001000000000011111110000
B00000000000000000000000000000000
B11001100000100010000011111110000
B11001100000100010000011111110000
B00000000000000000000000000000000
B00100000000000000000000000000000
B11001100000010100000011111110000
B00000000000000000000000000000000
B00101000001100010000011111110000
B00101000001100010000011111110000
B01000000000000000000000000000000
B00101000001100010000011111110000
B00100000001100100000011110110000
B11111100000010000000011111110000
B00100000001111000000011011110000
B00100000000000000000000000000000
B00101000001100010000011111110000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001000001100100000011101010000
B10000000000000000000000000000000
B00101000001100010000011111110000
B11111100000010000000011111110000
B00100000001111000000011011110000
B00000000000000000000000000000000
B00101000001100010000011111110000
B10000000000000000000000000000000
B00101000001100010000011111110000
B00100000001100100000011110110000
B11111100000010000000011111110000
B00100000001111000000011011110000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001000001100100000011111000000
B10000000000000000000000000000000
B11001100000010100000011111110000
B00001010100000000000000000000000
B00011110100000000000000000000000
B01001100001000000000011111110000
B10000000000100010000011111110000
B11111100000100010000011111110000
B11001100000100010000011111110000
B11001100000100010000011111110000
B11111100001100010000011111110000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001100000010000000011111110000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001100000010000000011111110000
B11001100001010001000011111110000
B10111101100000000000000000000000
B11111100000010000000011111110000
B01000000000010000000011111000000
B11001000001100100000011110000000
B00100000000000000000000000000000
B11001000000010100000011111110000
B10000000000000000000000000000000

B01000000000010000000011111000000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00100000001100100001011111100000
B11001000001100100000011111010000 B11111100001100100000100011110000
B10000000000000000000000000000000 B11001100001010001000011111110000
B11111100000010000000011111110000 B11110011100000000000000000000000
B00100000001111000000111011110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11001100000010000000011111110000
B00100000001100100000011111000000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B11001100000010100000011111110000 B11111000001111000000011111110000
B10010001100000000000000000000000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B00101000001100010000011111110000 B11111100000010000000011111110000
B11000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B00100000001100100000011111010000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B11001100000010100000011111110000 B11111000001111000000011111110000
B10001011100000000000000000000000 B10000000000100000000011111110000
B01001100001000000000011111110000 B01001100001111000000011111110000
B10000000000100010000011111110000 B01000000001000000000011111110000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11001100000100010000011111110000 B11001100000100010000011111110000
B11001100000100010000011111110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B00101000001100010000011111110000 B11111100000100010000011111110000
B10100111100000000000000000000000 B11111000001111000000011111110000
B01100110000000000000000000000000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B11001100000010100000011111110000 B11111100000010000000011111110000
B11000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001000001100100000011110000000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B11001000000010100000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00100000001100101000011111010000
B11001000001100100000011111010000 B11111100001100100000111011110000

B10000000000000000000000000000000 B11001100001010001000011111110000
B11111100000010000000011111110000 B11011000010000000000000000000000
B00100000001111000000011011110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11001100000010000000011111110000
B00100000001100100000011111000000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B11001100000010100000011111110000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11001100000010000000011111110000
B00100000001100100000011111000000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B11001100000010100000011111110000 B11111000001111000000011111110000
B11011011100000000000000000000000 B11111000001111000000011111110000
B10111000010000000000000000000000 B10000000000100000000011111110000
B01001100001000000000011111110000 B01001100001111000000011111110000
B10000000000100010000011111110000 B01000000001000000000011111110000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11001100000100010000011111110000 B11001100000100010000011111110000
B11001100000100010000011111110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B00101000001100010000011111110000 B11111100000100010000011111110000
B10001100010000000000000000000000 B11111000001111000000011111110000
B11001101000000000000000000000000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B11001100000010100000011111110000 B11111100000010000000011111110000
B11000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001000001100100000011110000000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B11001000000010100000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00100000001100101000011111010000
B11001000001100100000011111010000 B11111100001100100000111011110000
B10000000000000000000000000000000 B11001100001010001000011111110000
B11111100000010000000011111110000 B11100110010000000000000000000000
B00100000001111000000011011110000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11001100000010000000011111110000

```
B00100000001100100000011111000000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B11001100000001010000001111111000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11001100000010000000011111110000
B00100000001100100000011111000000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B11001100000001010000001111111000 B11111000001111000000011111110000
B11100100010000000000000000000000 B11111000001111000000011111110000
B10010110010000000000000000000000 B10000000000100000000011111110000
B01001100001000000000011111110000 B01001100001111000000011111110000
B10000000000100010000011111110000 B01000000000100000000001111110000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11001100000100010000011111110000 B11001100000100010000011111110000
B11111100001100010000011111110000 B00000000000000000000000000000000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B11001000000001010000001111111000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00101000001100010000011111110000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B11001000000001010000001111111000
B11001100000010000000011111110000 B00100000001100100001011111100000
B11001000001100100000011111010000 B11111100001100100000100011110000
B10000000000000000000000000000000 B11001100001010001000011111110000
B11111100000010000000011111110000 B01101001010000000000000000000000
B00100000001111000000111011110000 B11001100000010000000011111110000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11001000001100100000011110000000 B11111100000010000000011111110000
B01000000000000000000000000000000 B01000000000010000000011111000000
B11001000000001010000001111111000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B00100000001100101000011111010000 B11001000001100100000011111010000
B11111100001100100000111011110000 B10000000000000000000000000000000
B11001100001010001000011111110000 B11111100000010000000011111110000
B01011101010000000000000000000000 B00100000001111000000011011110000
```

```
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100001100010000011111110000 B10000000000000000000000000000000
B11001100000010000000011111110000 B00100000001100100000011111000000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B11001100000010100000011111110000
B11111000001111000000011111110000 B01101110010000000000000000000000
B11111000001111000000011111110000 B00111101010000000000000000000000
B10000000000100000000011111110000 B01001100001000000000011111110000
B01001100001111000000011111110000 B10000000000100010000011111110000
B01000000001000000000011111110000 B11001100000100010000011111110000
B11001100000100010000011111110000 B11001100000100010000011111110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11001100000010000000011111110000 B00100000001100101000011111010000
B11001000001100100000011111010000 B11111100001100100000011101111000
B10000000000000000000000000000000 B11001100001010001000011111110000
B11111100000010000000011111110000 B10100111010000000000000000000000
B00100000001111000000011011110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
B10000000000000000000000000000000 B11001100000010000000011111110000
B00100000001100100000011111000000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B11001100000010100000011111110000 B11111000001111000000011111110000
B00100011010000000000000000000000 B11111100000010000000011111110000
B01000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11001100000010000000011111110000
B11001100000010100000011111110000 B11111000001111000000011111110000
B10001111010000000000000000000000 B10000000000100000000011111110000
B01001100001000000000011111110000 B01001100001111000000011111110000
B10000000000100010000011111110000 B01000000001000000000011111110000
B11111100000100010000011111110000 B00000000000000000000000000000000
B11001100000100010000011111110000 B11001100000100010000011111110000
B11001100000100010000011111110000 B11111100000010000000011111110000
B00000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100000010000000011111110000
B10000000000000000000000000000000 B01000000000010000000011111000000
B00101000001100010000011111110000 B11111100001100010000011111110000
```

B01000000000000000000000000000000 B11111100000100010000011111110000
B10010000110000000000000000000000 B11111000001111000000011111110000
B11011111000000000000000000000000 B11111100001100010000011111110000
B01000000000000000000000000000000 B11001100000010000000011111110000
B00100000001100100001011111100000 B11001000001100100000011111010000
B11111100001100100000100011110000 B10000000000000000000000000000000
B11001100001010001000011111110000 B11111100000010000000011111110000
B01011010110000000000000000000000 B00100000001111000000011011110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B00100100110000000000000000000000
B11111000001111000000011111110000 B00101011100000000000000000000000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B00011100110000000000000000000000
B11111000001111000000011111110000 B01000100000000000000000000000000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B11001000000010100000011111110000
B11111100000100010000011111110000 B00000010110000000000000000000000
B11111000001111000000011111110000 B01000010100000000000000000000000
B11111100000010000000011111110000 B11000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B00010010110000000000000000000000
B11111000001111000000011111110000 B11011111010000000000000000000000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B00011010110000000000000000000000
B11111000001111000000011111110000 B00000100010000000000000000000000
B11111000001111000000011111110000 B01110001110000000000000000000000
B11111100000010000000011111110000 B00000000000000000000000000000000

B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1111110000000100000000011111110000 B0000000000000000000000000000000000
B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1111110000000100000000011111110000 B1000000000000000000000000000000000
B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1111110000001000100000111111110000 B0101011011000000000000000000000000
B1111100000111100000000111111110000 B0011011001000000000000000000000000
B1111110000000100000000011111110000 B0000000000000000000000000000000000
B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1111110000000100000000011111110000 B1000000000000000000000000000000000
B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1111110000011000100000111111110000 B1000000000000000000000000000000000
B1111110000001000100000111111110000 B0001111011000000000000000000000000
B1111100000111100000000111111110000 B1101111110000000000000000000000000
B1100110000000100000010111111110000 B0010000000000100000010111010100000
B0010000000110001000001111111110000 B1111110000110001000001111111110000
B0000000000000000000000000000000000 B1111110000110010000010001111110000
B1000000000000000000000000000000000 B1111110000000100000000011111110000
B1100000000000000000000000000000000 B0100000000000100000000011111000000
B1100110000000101000000111111110000 B1100110000000100000000011111110000
B1111110000000100000000011111110000 B1100000000000000000000000000000000
B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1111110000001000100000111111110000 B0011000111000000000000000000000000
B1111100000111100000000111111110000 B1111110101000000000000000000000000
B1111100000111100000000111111110000 B0111000111000000000000000000000000
B1000000000010000000000111111110000 B0100110000100000000001111111110000
B0100110000111100000000111111110000 B1000000000010001000001111111110000
B0100000000100000000000111111110000 B1100110000010001000001111111110000
B1111110000000100000000011111110000 B0000000000000000000000000000000000
B0100000000000100000000011111000000 B0010100000110001000001111111110000
B1100100000110010000000111101000000 B1111110000000100000000011111110000
B0000000000000000000000000000000000 B0100000000000100000000011111000000
B1100100000000101000000111111110000 B0010000000000100000000011111110000
B0010000000000100000000011111110000 B0010000000000100000000011111110000
B0010000000000100000000011111110000 B1100110000101000100001111111110000
B1111110000000100000000011111110000 B0001010111000000000000000000000000
B0010000000111100000000111011110000 B0010000000000100000000011111110000
B1111100000111100000000111111110000 B0010100111000000000000000000000000
B1111100000111100000000111111110000 B0101010111000000000000000000000000
B1000000000010000000000111111110000 B0100110000100000000001111111110000
B0100110000111100000000111111110000 B1000000000010001000001111111110000
B0100000000100000000000111111110000 B1111110000010001000001111111110000
B0000000000000000000000000000000000 B1100110000010001000001111111110000
B1100110000010001000001111111110000 B1100110000010001000001111111110000

B11111100001100010000011111110000 B01011111000000000000000000000000
B11111100000100010000011111110000 B01011101110000000000000000000000
B11111000001111000000011111110000 B10001001110000000000000000000000
B11111100001100010000011111110000 B01011111000000000000000000000000
B11111100000100010000011111110000 B00000011100000000000000000000000
B11111000001111000000011111110000 B10001001110000000000000000000000
B11111100001100010000011111110000 B01011111000000000000000000000000
B11111100000100010000011111110000 B01100011100000000000000000000000
B11111000001111000000011111110000 B10001001110000000000000000000000
B11111100001100010000011111110000 B01011111000000000000000000000000
B11111100000100010000011111110000 B00110011100000000000000000000000
B11111000001111000000011111110000 B10001001110000000000000000000000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100001100010000011111110000 B01000000000000000000000000000000
B11111100000100010000011111110000 B01011011110000000000000000000000
B11111000001111000000011111110000 B11011111000000000000000000000000
B11001100001010001000011111110000 B11111100000010000000011111110000
B00001000001000000000000000000000 B00100000001110000000111011110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B01110111110000000000000000000000
B11111000001111000000011111110000 B00101011100000000000000000000000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B01011111110000000000000000000000
B11111000001111000000011111110000 B01000100000000000000000000000000
B11001100001010001000011111110000 B11111100000010000000011111110000
B00001000001000000000000000000000 B00100000001110000000111011110000
B11111100000010000000011111110000 B00000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B10000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000010000000011111110000 B01000000000000000000000000000000
B01000000000010000000011111000000 B00101000001100010000011111110000
B11111100000100010000011111110000 B01110000001000000000000000000000

```
B1111100000111100000000111111110000 B0010111101000000000000000000000000
B1111100000111100000000111111110000 B1100110000100000000000000000000000
B1111110000001000000000111111110000 B0000000000000000000000000000000000
B0100000000001000000000111111000000 B0010100000110001000000111111110000
B1111110000001000000000111111110000 B0000000000000000000000000000000000
B0100000000001000000000111111000000 B0010100000110001000000111111110000
B1111110000001000000000111111110000 B1000000000000000000000000000000000
B0100000000001000000000111111000000 B0010100000110001000000111111110000
B1111110000001000100000111111110000 B0000010000100000000000000000000000
B1111100000111100000000111111110000 B0011011001000000000000000000000000
B1111110000001000000000111111110000 B1100000000000000000000000000000000
B0100000000001000000000111111000000 B1100110000001010000000111111110000
B1111110000001000000000111111110000 B1100000000000000000000000000000000
B0100000000001000000000111111000000 B0010100000110001000000111111110000
B1111110000001000000000111111110000 B1100000000000000000000000000000000
B0100000000001000000000111111000000 B0010100000110001000000111111110000
B1100100000110010000000111101000000 B1111110000001000100000111111110000
B1000110000100000000000000000000000 B1111100000111100000000111111110000
B1111110101000000000000000000000000 B1111100000111100000000111111110000
B1100110000100000000000000000000000 B1000000000001000000000111111110000
B0100110000100000000000111111110000 B0100110000111100000000111111110000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0100101000100000000000000000000000 B1101000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0000000000000000000000000000000000 B0000000000000000000000000000000000
B0110110000100000000000000000000000 B1010101000100000000000000000000000
B1111101000100000000000000000000000))
```

```
|#

;;  This image above executed on a fabricated FM9001 in a test jig at Indiana
;;  University with the help of M. Esen Tuna, Bhaskar Bose, and Steve Johnson.
;;  The resulting state that begins with the following:

#|

(2  4  7  1
 2  4  7  1
 1  266373121  266339330  266374207
 0  266374207  4  266374207
 4  266373183  21  266353695
 941  266374207  1  266374207
 8  266342451  266358835  266353695
 27  266353695  31  266340353
 266339378  266353714  266373121  266339330
 266373171  266373171  266342463  0
 65015810  266374164  266342451  266374164
 266342463  1  65015810  266374164
 98585619  266342463  1  65015810
 266358835  266342463  1  65015810
 266374164  266409011  266342463  88
 259013636  266342463  1  65015810
 266374164  98585619  266342463  1
 65015810  266358835  266342463  0
....


Note that the Nim state beginning in location 0 of 15 4 7 1 has been
transformed into 2 4 7 1, which by proof we know is an optimal move.

|#
```

Index