A Proved Application with Simple Real-Time Properties

Matthew Wilding

Technical Report 78 October, 1992

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: wilding@cli.com

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government

Abstract

This report describes the proof of an application on the CLI short stack described in [2]. The application generates moves in a game of Nim. We describe Nim and formalize the notion of optimal play. We develop a specification that also includes some simple real-time constraints. A Piton program that implements the specification is presented, along with an Nqthm-checked correctness proof. We make some observations about proving Piton programs correct, and describe the use of the fabricated FM9001 to run the compiled program.

1 Introduction

Improvement in computer hardware has led to computer control of power plants, aircraft, pacemakers, chemical processes, and many other things. Reliability is particularly important in these kinds of applications in order to ensure safety, but hardware improvement has also led to software so complex that it is unpredictable and therefore undependable. The application of software engineering practices such as rigorous testing can improve matters somewhat, but the correctness of a computer system can not be guaranteed this way.

Correct operation of these sorts of control programs require not just *functional correctness* — that the program produce the correct answer. A correct control program must operate in real-time, so it must produce answers in a timely fashion.

One approach to developing dependable software is to formalize a problem specification in a logic, formalize the semantics of a computer language in the logic, and prove that a particular program written in the computer language meets the requirements of the problem specification. Extra reliability can be achieved by checking the proof mechanically. This report describes a small application proved to have certain desirable properties. Some of these properties involve a simple characterization of the program's timeliness.

We discuss briefly some related work in Section 2. In Section 3 we develop a specification for a program that plays Nim, a mathematical game that has been played for centuries. Section 4 describes an algorithm and implementation of this program, and a correctness proof is discussed in Section 5. Appendix A lists the program, and Appendix B lists the input to a mechanical theorem prover that constitutes a mechanical proof of the correctness theorem.

2 Background

Nqthm is the name of both a logic and an associated theorem-proving system that are documented in [4]. A large number of mathematical theorems from many disparate domains have been proved using Nqthm. One of these domains is computer systems verification.

```
                  (n Piton interpreter steps)
                           P(P0, n)
              P0 ----------------------------> Pn
              ||                               /\
              ||                               ||
  LOAD(P0)    ||                               ||  DISPLAY(Mk)
              ||                               ||
              \/                               ||
              M0 ----------------------------> Mk
                      FM9001-DESIGN(M0, k)
                 (k FM9001 interpreter steps)
```

Figure 1: The Piton correctness proof.

[8] describes the implementation of a compiler for the language Piton and the associated mechanically-checked correctness theorem. A formal semantics for Piton, a formal description of the FM9001 microprocessor, and the Piton compiler are introduced as Nqthm functions. The compiler correctness theorem relates the data values after running a Piton program (using Piton's formal semantics) to the data values computed by the FM9001 running a compiled Piton program.

The semantics of Piton is described using an interpreter function. A Piton state consists of elements that comprise a programmer's model of how Piton executes: a list of Piton programs to run, a user-addressable stack, a current instruction pointer, a subroutine calling stack, a data area, the word size, stack size limits, and a program status flag. The interpreter function **P** takes as arguments a Piton state and the number of Piton instructions to run and returns the Piton state resulting from the computation.

The semantics of FM9001 is also described using an interpreter function. An FM9001 state consists of elements that comprise a programmer's model for the FM9001 design: values for the register file, the condition flags, and the memory. The interpreter function **FM9001-DESIGN** takes as arguments an FM9001 state and the number of FM9001 instructions to run and returns the FM9001 state resulting from the computation.

The Piton correctness theorem is suggested by Figure 1. **LOAD** is the Piton compiler, and **DISPLAY** is a function that extracts a Piton data segment from an FM9001 image. Roughly speaking, the theorem states that the data segment calculated by a Piton program running on the Piton interpreter and the data segment calculated by running a compiled Piton program on the FM9001 are identical.

An interpreter function serves as a precise specification for the expected behavior of a system component. This general approach to system verification is described fully in [2]. Interpreter functions can be very complex: Piton has 71 instructions and some high-level features, and the definition of **P** in the Nqthm logic requires about 50 pages. Two other related projects have used the interpreter approach. [11] describes a verified compiler for the Pascal-like language Micro-Gypsy that generates Piton code. [6] and [1] describe a microprocessor design that is proved to satisfy the behavior formalized by the FM9001 specification. Since these proofs use the same interpreter functions as in the Piton proof, the correctness theorems of these three projects can be "stacked" to derive a single correctness theorem that relates Micro-Gypsy semantics to a compiled Micro-Gypsy image running on an FM9001. Taken together these projects are known as the *CLI short stack*.

Recent work on fabricating and making usable the FM9001 has progressed. It is possible to run a Piton program on an actual FM9001 using a specially-constructed test board. (See [1] for a description of this effort, including details of three trivial but currently unverified changes made to the Piton compiler to make the compiled Piton code usable on the FM9001 test board.)

Real-time programs are programs that have real, "clock on the wall", timing requirements. In this project we prove that the number of Piton instructions a program executes is within a specified range. This is a natural sort of proof to accomplish given the interpreter-based approach used to formalize programming languages in [2]. We will call this kind of constraint a simple real-time property of the program since, by knowing the bounds of the Piton instruction execution times, we can in principle derive execution time bounds from bounds on the execution times of the individual instructions. Nevertheless, this sort of program and specification is unlike what is commonly thought of as "real-time" program in at least two important ways.

- Correctness is not cast directly in terms of time. Bounds on the number of executed instructions can be related to time, although we do not do this. The timing behavior of the underlying machine that executes the program is ignored.

- Real-time programs are complicated, requiring timely reaction to inputs received unpredictably. The example program discussed later in this report does not receive input other than the parameters with which it is called, and it has a relatively simple functional specification.

With these caveats, we will use the expression "simple real-time property" to characterize a constraint on the number of instructions a program will execute.

Figure 2: An Example Nim Game

3 A specification for a good Nim program

3.1 Good Nim Play

Nim is one of the oldest and one of the most engaging mathematical games [5]. The game is played with piles of stones and two players who alternate turns. On his turn a player removes at least one stone from exactly one pile. The player who removes the final stone loses. Figure 2 shows an example game.

A Nim state consists of a list of numbers that represents the number of stones in each of the piles. A *strategy* is a function that maps non-empty Nim states to Nim states in a way consistent with the notion of a legal Nim move. A *winning strategy* is a strategy that guarantees for a particular NIM state that the player who is about to take a turn will win. An *optimal move* for a particular non-empty Nim state is the first step in some winning strategy if one exists, or any legal move otherwise.

Sometimes there is no winning strategy. For example, if a player has to take a turn and there are two piles left each with two stones, there will be a winning strategy for his opponent on the next move regardless of the move he makes, and thus there is no winning strategy for the player and all legal moves are optimal. Sometimes there is a move a player can make that will cause his opponent in his next turn to have no winning strategy. For example, if the player is left

Figure 3: State-space search for an optimal move

with two piles with 4 and 2 stones respectively, he can remove 2 stones from the 4 pile and leave his opponent in the 2 piles of 2 situation. Thus, there is a winning strategy for the current player which begins with a move that ends in a state from which there is no winning strategy.

For any non-empty Nim state, there either is one stone left, or there is a winning strategy for the next player, or there will a winning strategy for the opponent on his next move. We can therefore search for an optimal move from any Nim state. Figure 3 depicts the search tree of the search for an optimal move from a state. The nodes in the tree in bold type are *losing states* from which there is no winning strategy.

We formalize this search idea using the Nqthm function WSP. (WSP state t) returns false if state is a losing state, and an optimal move otherwise.

```
;; 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.

(defn wsp (state flag)
  (if flag
      (if (or (all-zero-bitvp state) (not (nat-listp-simple state)))
          state
        (wsp (all-valid-moves state) f))
    (if (listp state)
        (if (not (wsp (car state) t))
            (car state)
          (wsp (cdr state) f))
      f)))
```

Nqthm provides a facility for executing functions on constants and a trace facility. Figure 4 is a trace of the calculation of an optimal move from the same state whose search tree was presented in Figure 3. The result is '(1 1 1), which is an optimal move as expected.

3.2 A Nim-Playing Program Specification

Having formalized the notion of an optimal move in Nim, we can construct a specification for a program that plays Nim optimally and efficiently.

The specification uses the following five undefined functions.

- `CM-PROG` : the Piton program
- `COMPUTER-MOVE-CLOCK` : the number of piton instructions the program will execute
- `NIM-PITON-CTRL-STK-REQUIREMENT` : an upper bound on the amount of control stack space
- `NIM-PITON-TEMP-STK-REQUIREMENT` : an upper bound on the amount of temporary stack space
- `COMPUTER-MOVE` : the algorithm by which moves are computed

In the following subsections we constrain these functions in a way that specifies the program.

```
*(wsp '(1 2 1) t)
 1> (<<WSP>> (LIST '(1 2 1) T))
  2> (<<WSP>> (LIST '((0 2 1) (1 1 1) (1 0 1) (1 2 0)) F))
   3> (<<WSP>> (LIST '(0 2 1) T))
    4> (<<WSP>> (LIST '((0 1 1) (0 0 1) (0 2 0)) F))
     5> (<<WSP>> (LIST '(0 1 1) T))
      6> (<<WSP>> (LIST '((0 0 1) (0 1 0)) F))
       7> (<<WSP>> (LIST '(0 0 1) T))
        8> (<<WSP>> (LIST '((0 0 0)) F))
         9> (<<WSP>> (LIST '(0 0 0) T))
         <9 (<<WSP>> '(0 0 0))
         9> (<<WSP>> (LIST NIL F))
         <9 (<<WSP>> F)
        <8 (<<WSP>> F)
       <7 (<<WSP>> F)
      <6 (<<WSP>> '(0 0 1))
     <5 (<<WSP>> '(0 0 1))
     5> (<<WSP>> (LIST '((0 0 1) (0 2 0)) F))
      6> (<<WSP>> (LIST '(0 0 1) T))
       7> (<<WSP>> (LIST '((0 0 0)) F))
        8> (<<WSP>> (LIST '(0 0 0) T))
        <8 (<<WSP>> '(0 0 0))
        8> (<<WSP>> (LIST NIL F))
        <8 (<<WSP>> F)
       <7 (<<WSP>> F)
      <6 (<<WSP>> F)
     <5 (<<WSP>> '(0 0 1))
    <4 (<<WSP>> '(0 0 1))
   <3 (<<WSP>> '(0 0 1))
   3> (<<WSP>> (LIST '((1 1 1) (1 0 1) (1 2 0)) F))
    4> (<<WSP>> (LIST '(1 1 1) T))
     5> (<<WSP>> (LIST '((0 1 1) (1 0 1) (1 1 0)) F))
      6> (<<WSP>> (LIST '(0 1 1) T))
       7> (<<WSP>> (LIST '((0 0 1) (0 1 0)) F))
        8> (<<WSP>> (LIST '(0 0 1) T))
         9> (<<WSP>> (LIST '((0 0 0)) F))
          10> (<<WSP>> (LIST '(0 0 0) T))
          <10 (<<WSP>> '(0 0 0))
          10> (<<WSP>> (LIST NIL F))
          <10 (<<WSP>> F)
         <9 (<<WSP>> F)
        <8 (<<WSP>> F)
       <7 (<<WSP>> '(0 0 1))
      <6 (<<WSP>> '(0 0 1))
      6> (<<WSP>> (LIST '((1 0 1) (1 1 0)) F))
       7> (<<WSP>> (LIST '(1 0 1) T))
        8> (<<WSP>> (LIST '((0 0 1) (1 0 0)) F))
         9> (<<WSP>> (LIST '(0 0 1) T))
          10> (<<WSP>> (LIST '((0 0 0)) F))
          11> (<<WSP>> (LIST '(0 0 0) T))
          <11 (<<WSP>> '(0 0 0))
          11> (<<WSP>> (LIST NIL F))
          <11 (<<WSP>> F)
          <10 (<<WSP>> F)
         <9 (<<WSP>> F)
        <8 (<<WSP>> '(0 0 1))
       <7 (<<WSP>> '(0 0 1))
       7> (<<WSP>> (LIST '((1 1 0)) F))
        8> (<<WSP>> (LIST '(1 1 0) T))
         9> (<<WSP>> (LIST '((0 1 0) (1 0 0)) F))
          10> (<<WSP>> (LIST '(0 1 0) T))
          11> (<<WSP>> (LIST '((0 0 0)) F))
          12> (<<WSP>> (LIST '(0 0 0) T))
          <12 (<<WSP>> '(0 0 0))
          12> (<<WSP>> (LIST NIL F))
          <12 (<<WSP>> F)
          <11 (<<WSP>> F)
          <10 (<<WSP>> F)
         <9 (<<WSP>> '(0 1 0))
        <8 (<<WSP>> '(0 1 0))
        8> (<<WSP>> (LIST NIL F))
        <8 (<<WSP>> F)
       <7 (<<WSP>> F)
      <6 (<<WSP>> F)
     <5 (<<WSP>> F)
    <4 (<<WSP>> F)
   <3 (<<WSP>> '(1 1 1))
  <2 (<<WSP>> '(1 1 1))
 <1 (<<WSP>> '(1 1 1))
'(1 1 1)
*
```

Figure 4: WSP trace

3.2.1 Algorithm Legality

We require that the function `COMPUTER-MOVE` returns valid Nim moves.

```
(implies
 (good-non-empty-nim-statep state ws)
 (valid-movep state (computer-move state ws)))
```

3.2.2 Algorithm Optimality

We require that the function `COMPUTER-MOVE` return an optimal move from non-losing Nim states

```
(implies
 (and
  (good-non-empty-nim-statep state ws)
  (wsp state t))
 (not (wsp (computer-move state ws) t)))
```

3.2.3 Algorithm Implementation

We require that a program that produces the same result as `COMPUTER-MOVE` be implemented in Piton. We represent the Nim state by an array of naturals and a length that is passed to the program as parameters. We call the Piton program `computer-move`[1].

When the Piton subroutine computer-move in a list of programs returned by the function `CM-PROG`[2] is executed using the Piton interpreter on a "reasonable" Piton state for `COMPUTER-MOVE-CLOCK` "ticks" the resulting state has the program counter incremented by 1, the program status word set to 'run, and the naturals array representing the Nim state replaced by a new array with the same value as that calculated by `COMPUTER-MOVE`. (See [8] for a full description of the Piton interpreter `P` and the significance of the program status word and program counter.)

---

[1] We use upper case for Nqthm event names such as lemma names and function definition names and lower case for the name of Piton subroutine names. So, `COMPUTER-MOVE` is an Nqthm function and `computer-move` is the name of a Piton subroutine.

[2] It is disappointing that `CM-PROG`, the function that returns the Piton program that implements this specification, is a function of one argument rather than a constant. We wish to use bit vectors in our program and also write Piton programs that are machine independant. Unfortunately, because of an oversight in the Piton language design, there is no way to use bit-vectors without knowledge of the word size. The only subprogram that uses the word size in the implementation is push-1-vector, which is a one-line program that pushes a vector onto the stack. Perhaps future versions of Piton will have an instruction that pushes a 1-vector onto the stack.

```
(implies
 (and
  (equal p0 (p-state pc ctrl-stk
              (cons wa (cons np (cons s temp-stk)))
              (append (cm-prog word-size) prog-segment)
              data-segment max-ctrl-stk-size max-temp-stk-size
              word-size 'run))
  (equal (p-current-instruction p0) '(call computer-move))
  (computer-move-implemented-input-conditionp p0))
 (let ((result
         (p p0
           (computer-move-clock
             (untag-array (array (car (untag s)) data-segment))
             word-size))))
   (and
      (equal (p-pc result) (add1-addr pc))
      (equal (p-psw result) 'run)
      (equal
        (untag-array
          (array (car (untag s)) (p-data-segment result)))
        (computer-move
          (untag-array (array (car (untag s)) data-segment))
          word-size)))))
```

`COMPUTER-MOVE-IMPLEMENTED-INPUT-CONDITIONP` above identifies "reasonable" Piton states from which we expect our program to work properly. The conditions that must be met are:

- The control stack is non-empty.
- The word-size is at least 8 bits.
- At least `NIM-PITON-CTRL-STK-REQUIREMENTS` bytes are available on the control stack.
- At least `NIM-PITON-TEMP-STK-REQUIREMENTS` bytes are available on the temporary stack
- A naturals array address is first on the temporary stack. (This array is used to represent the Nim state.)
- The length of the naturals array is second on the temporary stack.
- An array address is third on the temp stack. (This array is used as a work area for the program.)
- The array of naturals whose address is first on the stack contains at least one non-zero element.

Precise Nqthm definitions are contained in Appendix B.

3.2.4 Fast Response

We require that the program return a calculated move within a window of time. The program must execute between 10,000 and 20,000 Piton instructions. We assume the word size is at most of length 32, and the Nim state has no more than 6 piles.

```
(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)))
```

Note that this part of the specification eliminates several possible implementations. One is the blind search implementation suggested by the definition of `WSP`, since the first level of the search tree has as many as $6 * 2^{32}$ nodes, and there are as many as $6 * 2^{32}$ levels to the tree.

3.2.5 Realistic memory use

We require a very modest use of stack space. (Note that these function are contained in `COMPUTER-MOVE-IMPLEMENTED-INPUT-CONDITIONP` above.)

```
(lessp (plus (nim-piton-ctrl-stk-requirement)
             (nim-piton-temp-stk-requirement))
       1000)
```

This part of the specification eliminates, for example, a table-driven implementation since there are 2^{177} distinct states.

3.2.6 FM9001 Loadability

We require that the program work on an FM9001 and that it meet the requirements of the compiler correctness proof of [8]. This requires among other things that the compiled Piton programs fit into the FM9001 address space and that the Piton programs be well-formed. For some P0 for which

```
(COMPUTER-MOVE-IMPLEMENTED-INPUT-CONDITIONP P0)
```

holds, we require that

```
(and (proper-p-statep p0)
     (p-loadablep p0 0)
     (equal (p-word-size p0) 32))
```

This part of the specification allows us to apply the compiler correctness proof. It also eliminates some possible implementation approaches that we don't want. A program that uses alternation to solve the Nim problem by cases will be so large that it will not fit into the FM9001 address space, and will therefore not meet the requirements of this part of the specification.

4 The Nim Implementation

In this section we develop an algorithm for efficient calculation of optimal moves, and present a Piton program that implements this algorithm. In Section 5 we discuss the mechanical proof that this implementation meets the specification developed in Section 3.

Since a formal specification has been developed for this program as well as a mechanical proof that the program meets the specification, a reader interested only in the behavior of the Nim software would best skip this section. Note that as a part of the correctness proof, for example, each of the hundreds of instructions in the program has been proved to execute on arguments of the expected Piton type, each loop is proved to terminate, the stacks are proved never to be exhausted, etc. In contrast to conventional program development efforts where the program source code is the only place where an absolutely dependable description of the behavior of the system can be found, in this effort the specification in Section 3 is dependable because the correctness proof outlined in Section 5 guarantees that it is a correct description of the program's behavior.

We present the program because we hope to demonstrate how verified software for the short stack can be developed and to make more concrete the description of the development effort.

4.1 An efficient Nim algorithm

In Section 3 we defined `WSP`. Recall that `(WSP state t)` is false if no winning strategy exists for state, and non-false otherwise.

Let `(BIGP state)` = number of piles with at least 2 stones.

Let the bit-vector representation of a number be the conventional low-order-bit-first base 2 representation for some large number of bits.

Let `(XOR-BVS state)` = bitwise exclusive-or of the bit-vector representations of the number of stones in each pile.

Let `(GREEN-STATEP state)` = `(BIGP state)`$>0 \leftrightarrow$ `(XOR-BVS state)`$\neq$`0-vector`.

Theorem: `(GREEN-STATEP state)` $\leftrightarrow$ `(WSP state t)`.

This remarkable property was rediscovered at Computational Logic, but has in fact been known at least since its publication in 1901 [3]. The most obvious

Figure 5: State-space search for an optimal move with green states starred

proof uses an induction on the search tree. The mechanical proof was achieved as a part of this effort since it is a needed lemma in the proof of program correctness that we will discuss later. (See **WSP-GREEN-STATE** in appendix B.)

Figure 5 is the search tree of Figure 3 except that stars are added to nodes where **(GREEN-STATEP state)** is non-false. Note that, as guaranteed by the theorem above, the non-bold nodes from which there is a winning strategy are exactly the nodes that are marked with stars.

We can exploit this theorem in the following algorithm that computes optimal moves efficiently.

> If **(BIGP state)** < 2 and there are an even number of non-empty piles, remove all the stones from a largest pile.
>
> If **(BIGP state)** < 2 and there are an odd number of non-empty piles, remove all but one stone from a largest pile.
>
> If **(BIGP state)** > 1, find a pile whose binary representation has a 1-bit in the position of the highest 1-bit in **(XOR-BVS state)**, and remove enough stones so that the new pile's binary representation is the **XOR** of the binary representations of the other piles.

From the previous theorem one can prove that this algorithm efficiently generates optimal moves when a winning strategy exists. A mechanical proof of this was constructed. (See **COMPUTER-MOVE-WORKS** in appendix B.)

```
SUBROUTINE NAT-TO-BV
   (VALUE) (CURRENT-BIT TEMP)
        CALL PUSH-1-VECTOR
        POP-LOCAL CURRENT-BIT
        CALL PUSH-1-VECTOR
        RSH-BITV
 LOOP:  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
 LAB:   PUSH-LOCAL CURRENT-BIT
        LSH-BITV
        POP-LOCAL CURRENT-BIT
        JUMP LOOP
 DONE:  RET
```

```
SUBROUTINE NAT-TO-BV-LIST
   (NAT-LIST BV-LIST LENGTH) (I 0)
 LOOP:  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
 DONE:  RET
```

Figure 6: Two example Piton subroutines

4.2 The Piton Nim program

A Piton program that implements this algorithm has been coded. It appears as the Nqthm definition `CM-PROG` in appendix B. A version with all the subsidiary definitions expanded and with some cosmetic syntactic changes intended to enhance readability appears in Appendix A. Figure 6 lists two of the routines in that appendix.

5 The Nqthm correctness proof

5.1 Different types of theorems in the proof

The proof that the Piton program meets the specification uses the default arithmetic library [9] and the Piton interpreter definitions [8]. Most of the lemmas in the proof script fall into one of the following four categories or are designed specifically to support a lemma in one of the categories.

- Many lemmas relate the behavior of Piton loops and subroutines when interpreted by the Piton interpreter to an Nqthm function. Typically, a special Nqthm function that calculates a result in precisely the same manner as the Piton program in question is defined. A clock function that computes the number of Piton instructions the loop or subroutine will execute is defined. A correctness lemma for a loop or subroutine states that the Piton interpreter running the loop or subroutine for the number of instructions computed by the clock function yields the same result as the Nqthm function.

  We call proofs of this kind of lemma *code correctness proofs*.

- Some lemmas demonstrate the equivalence of Nqthm functions that mimic Piton programs to Nqthm functions defined more naturally that are easier to reason about.

  We call proofs of this kind of lemma *specification proofs*.

- Some lemmas are used to prove optimality of `COMPUTER-MOVE`. That is, that the algorithm outlined in Section 4 works.

  We call proofs of this kind of lemma *algorithm proofs*.

- Some lemmas establish bounds on the clock functions.

  We call proofs of this kind of lemma *timing proofs*.

The timing proofs were done using PC-Nqthm [7], the interactive enhancement to Nqthm. All other proofs require only Nqthm. The algorithm proofs and time bound proofs are fairly standard mechanical proofs of a type done often before, so we will not discuss them in detail. We instead focus on the Piton-related lemmas in the proof.

5.2 An example subroutine proof

For each loop and subroutine, we characterize precisely the conditions under which it is supposed to work, and the effect it will have when executed. As an example, we focus on the correctness proof of the Piton subroutine `nat-to-bv` in figure 6.

Like most Nqthm prove-lemma commands, `CORRECTNESS-OF-NAT-TO-BV` has two important properties. First, since it is accepted by the Nqthm prover, we believe that it represents mathematical truth. Second, it is an instruction to the prover about how to prove future theorems. By constructing the exact form of the theorem mindful of the theorem's interpretation in later proof efforts, we add to the prover's ability to reason about programs.

The prove-lemma command (modified slightly for readability) is:

```
(prove-lemma correctness-of-nat-to-bv (rewrite)
             (implies
              (and
               (equal 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))
               (equal (p-current-instruction p0) '(call nat-to-bv))
               (nat-to-bv-input-conditionp p0))
              (equal
               (p p0 (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))))
```

The function `NAT-TO-BV-INPUT-CONDITIONP` contains additional preconditions that this subprogram requires in order to run correctly. There must be enough stack space to do the calculation, the value at the top of the stack must be a natural number representable on the machine, and the program segment must have the needed programs loaded. `NAT-TO-BV-CLOCK` is a function that computes the number of instructions the Piton subroutine `nat-to-bv` executes when called.

This lemma is useful because it equates the behavior of a Piton subprogram as defined by the Piton interpreter to an Nqthm term that does not involve the interpreter. By applying this lemma we are able to reason about this program without regard to the semantics of Piton. This makes proofs achievable, since as a practical matter proofs involving Piton programs running on the very complicated Piton interpreter are much more complex than proofs about Nqthm functions that compute similar results.

The lemma is stored in Nqthm as a replacement rule, and has been constructed so that later proofs can apply this lemma automatically during proofs. The subroutine `nat-to-bv-list` contains several kinds of Piton instructions, and the proof of the correctness of `nat-to-bv-list` depends on the semantics of these instructions as defined in the Piton interpreter. For example, `PUSH-LOCAL` is defined in the Piton interpreter and the theorem prover uses the definitions that describe the effect of executing a `PUSH-LOCAL` instruction automatically. Similarly, the instruction `CALL NAT-TO-BV` in the subprogram is reasoned about by Nqthm automatically using `CORRECTNESS-OF-NAT-TO-BV`.

Once a carefully-constructed correctness theorem about a subroutine is added to the database of proved lemmas, a call to that subroutine in a Piton program is reasoned about just as easily as any built-in Piton instruction.

5.3 Proof Statistics

The correctness proof of the specification in Section 3 required about 3 man-months, and consists of an approximately 800 event 220K events file listed in appendix B.[3] Approximately 40% of the man-hours were spent on code correctness proofs, 30% on specification proofs, 20% on algorithm proofs, and 10% on timing proofs.

6 Conclusions

Mechanical verification of programs in this manner is time-consuming and difficult. Nevertheless, and quite remarkably, the experience of building the Nim program suggests that development time scales linearly with program length. Once a Piton subroutine has been proved correct, a call to this subroutine can be reasoned about as easily as any basic Piton statement.

A modest but non-trivial application has been constructed for use on the verified CLI short stack. Its functional correctness has been verified using Nqthm. Mechanically-checked proofs of bounds on the number of executed instructions have been constructed. The formalization of a programming language with an interpreter as in [8] is particularly well-suited to proving program timing properties.

An FM9001 has been fabricated by LSI logic, and has run a compiled version of the Nim program. Some FM9001 code was written that allows the Piton Nim program to be run interactively. See [1] for details.

The FM9001 microprocessor, the Piton compiler, and the Nim program were never tested in a conventional manner. Even so, they each worked the first time and we would have been surprised if they had not.

[3]This does not include the events of the Piton compiler or arithmetic library. It does not include the time to accomplish an earlier proof related to Nim [10]. It does not include the time taken to prepare a report and a talk on this project.

A Nim-playing Piton program listing

This appendix contains a listing of the Nim Piton program. Appendix B contains the "official" version about which the correctness proofs have been accomplished in function `CM-PROG`. The listing in this appendix has been changed into a more-standard syntax for readability.

```
SUBROUTINE XOR-BVS (VECS-ADDR NUMVECS)
        PUSH-LOCAL VECS-ADDR
        FETCH
        PUSH-LOCAL NUMVECS
        SUB1-NAT
        POP-LOCAL NUMVECS
  LOOP: 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
  DONE: RET

SUBROUTINE PUSH-1-VECTOR NIL
        PUSH-CONSTANT (BITV (0 0 0 0 0 0 0 1))
        RET

SUBROUTINE NAT-TO-BV (VALUE) (CURRENT-BIT TEMP)
        CALL PUSH-1-VECTOR
        POP-LOCAL CURRENT-BIT
        CALL PUSH-1-VECTOR
        RSH-BITV
  LOOP: 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
  LAB:  PUSH-LOCAL CURRENT-BIT
        LSH-BITV
        POP-LOCAL CURRENT-BIT
        JUMP LOOP
  DONE: RET

SUBROUTINE BV-TO-NAT
          (BV) (CURRENT-BIT CURRENT-2POWER)
        PUSH-CONSTANT (NAT 1)
        POP-LOCAL CURRENT-2POWER
        CALL PUSH-1-VECTOR
        POP-LOCAL CURRENT-BIT
        PUSH-CONSTANT (NAT 0)
  LOOP: PUSH-LOCAL BV
        PUSH-LOCAL CURRENT-BIT
        AND-BITV
        TEST-BITV-AND-JUMP ALL-ZERO LAB
        PUSH-LOCAL CURRENT-2POWER
```

```
        ADD-NAT
  LAB:  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
  DONE: RET

SUBROUTINE NUMBER-WITH-AT-LEAST
          (NUMS-ADDR NUMNUMS MIN) (I)
        PUSH-CONSTANT (NAT 0)
        SET-LOCAL I
  LOOP: PUSH-LOCAL NUMS-ADDR
        FETCH
        PUSH-LOCAL MIN
        LT-NAT
        TEST-BOOL-AND-JUMP T LAB
        ADD1-NAT
  LAB:  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
  DONE: RET

SUBROUTINE HIGHEST-BIT (BV) (CB)
        CALL PUSH-1-VECTOR
        SET-LOCAL CB
        RSH-BITV
  LOOP: 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
  LAB:  PUSH-LOCAL CB
        LSH-BITV
        POP-LOCAL CB
        JUMP LOOP
  DONE: RET

SUBROUTINE MATCH-AND-XOR
          (VECS NUMVECS MATCH XOR-VECTOR) (I)
        PUSH-CONSTANT (NAT 0)
        POP-LOCAL I
  LOOP: 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
  FOUND: PUSH-LOCAL VECS
        FETCH
        PUSH-LOCAL XOR-VECTOR
        XOR-BITV
        PUSH-LOCAL VECS
        DEPOSIT
  DONE:  RET

SUBROUTINE REPLACE-VALUE (LIST OLDVAL NEWVAL)
  LOOP:  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
  DONE:  PUSH-LOCAL NEWVAL
        PUSH-LOCAL LIST
        DEPOSIT RET

SUBROUTINE NAT-TO-BV-LIST
          (NAT-LIST BV-LIST LENGTH) (I 0)
  LOOP:  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
  DONE:  RET

SUBROUTINE BV-TO-NAT-LIST
          (BV-LIST NAT-LIST LENGTH) (I 0)
  LOOP:  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
  DONE: RET

SUBROUTINE MAX-NAT (NAT-LIST LENGTH) (I 0 J)
        PUSH-CONSTANT (NAT 0)
  LOOP: 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
  LAB:  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
  DONE: RET

SUBROUTINE SMART-MOVE
          (STATE NUMPILES WORK-AREA) (I)
        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
  LAB:  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

SUBROUTINE DELAY (TIME)
  LAB:  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
  DONE: RET

SUBROUTINE COMPUTER-MOVE
          (STATE NUMPILES WORK-AREA) (I)
        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
  LAB:  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
```

B Nim Correctness proof

These events constitute a proof of the correctness of the Nim program except for some events regarding timing bounds that were proved separately using PC-Nqthm[7] and are not included here.

```
(proveall "nim"
'(

;; 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.

In August 92, it was run on the CLI FM9001.

```
|#
(note-lib "/artist-local/src/nqthm-1992/examples/fm9001-piton/piton" t)

(set-status addition-on addition ((otherwise enable)))
(set-status multiplication-on multiplication ((otherwise enable)))
(set-status remainders-on remainders ((otherwise enable)))
(set-status quotients-on quotients ((otherwise enable)))
(set-status exponentiation-on exponentiation ((otherwise enable)))
(set-status logs-on logs ((otherwise enable)))
(set-status gcds-on gcds ((otherwise enable)))

(defn clock-plus (x y)
  (plus x y))

(prove-lemma p-add1 (rewrite)
             (equal
              (p p0 (add1 n))
              (p (p-step p0) n))
             ((disable p-step)))

(prove-lemma p-0 (rewrite)
             (implies
              (zerop n)
              (equal (p p0 n) p0)))

(prove-lemma clock-plus-function (rewrite)
             (equal
              (p p0 (clock-plus x y))
              (p (p p0 x) y))
             ((induct (p p0 x))
              (disable p p-step)))

(disable p-add1)

(prove-lemma clock-plus-add1 (rewrite)
             (equal
              (p p0 (clock-plus (add1 x) y))
              (p p0 (add1 (clock-plus x y)))))

(disable clock-plus)

(prove-lemma clock-plus-0 (rewrite)
             (implies
              (zerop x)
             (equal
              (clock-plus x y)
              (fix y)))
             ((enable clock-plus)))

(prove-lemma fix-clock-plus (rewrite)
             (equal
              (fix (clock-plus x y))
              (clock-plus x y))
             ((enable clock-plus)))

(prove-lemma p-step1-opener (rewrite)
  (equal (p-step1 (cons opcode operands) p)
         (if (p-ins-okp (cons opcode operands) p)
             (p-ins-step (cons opcode operands) p)
             (p-halt p (x-y-error-msg 'p opcode))))

  ((disable p-ins-okp p-ins-step)))

(disable p-step1)

(prove-lemma p-opener (rewrite)
  (and (equal (p s 0) s)
       (equal (p (p-state pc ctrl temp prog data max-ctrl max-temp word-size
                          psw)
                 (add1 n))
              (p (p-step (p-state pc ctrl temp prog data max-ctrl max-temp
                                  word-size psw))
                 n)))
  ((disable p-step)))

(disable p)

(defn at-least-morep (base delta value)
  (not (lessp value (plus base delta))))
```

```
(prove-lemma at-least-morep-normalize (rewrite)
             (and
              (equal
               (at-least-morep (add1 base) delta value)
               (at-least-morep base (add1 delta) value))
              (equal
               (at-least-morep base (add1 delta) (add1 value))
               (at-least-morep base delta value))))

(prove-lemma at-least-morep-linear (rewrite)
             (implies
              (and
               (at-least-morep base d1 value)
               (not (lessp d1 d2)))
              (at-least-morep base d2 value)))

(prove-lemma lessp-as-at-least-morep (rewrite)
             (implies
              (at-least-morep base delta value)
              (and
               (equal
                (lessp value x)
                (not (at-least-morep x 0 value)))
               (equal
                (lessp x value)
                (at-least-morep x 1 value)))))

(disable at-least-morep)

(defn nat-to-bv (nat size)
  (if (zerop size)
      nil
    (if (lessp nat (exp 2 (sub1 size)))
        (cons 0 (nat-to-bv nat (sub1 size)))
      (cons 1 (nat-to-bv (difference nat (exp 2 (sub1 size)))
                         (sub1 size))))))

(defn nat-to-bv-state (state size)
  (if (listp state)
      (cons (nat-to-bv (car state) size)
            (nat-to-bv-state (cdr state) size))
    nil))

;; 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

(defn xor-bvs-program nil
  '(xor-bvs (vecs-addr numvecs)
            nil
            (push-local vecs-addr)
            (fetch)
            (push-local numvecs)
            (sub1-nat)
            (pop-local numvecs)
      (dl loop ()
            (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 ()
            (ret))))

(defn bit-vectors-piton (array size)
  (if (listp array)
      (and
       (listp (car array))
       (equal (caar array) 'bitv)
       (bit-vectorp (cadar array) size)
       (equal (cddar array) nil)
       (bit-vectors-piton (cdr array) size))
    (equal array nil)))

(defn array (name segment)
  (cdr (assoc name segment)))

;; vecs is name of state
(defn xor-bvs-input-conditionp (p0)
```

```
  (and
   (equal (car (top (p-temp-stk p0))) 'nat)
   (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
   (equal (cdadr (top (cdr (p-temp-stk p0)))) 0)
   (listp (cadr (top (cdr (p-temp-stk p0)))))
   (equal (cddr (top (p-temp-stk p0))) nil)
   (equal (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))
   (equal (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))
   (not (zerop (untag (top (p-temp-stk p0)))))
   (lessp (untag (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (listp (p-ctrl-stk p0))))

; time to run loop
(defn xor-bvs-clock-loop (numvecs)
  (if (zerop numvecs)
      3
    (plus 12 (xor-bvs-clock-loop (sub1 numvecs)))))

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

(defn xor-bvs-clock (numvecs)
  (plus 6 (xor-bvs-clock-loop (sub1 numvecs))))

(defn xor-bvs-array (current array n array-size)
  (if (zerop n)
      current
    (xor-bvs-array
     (xor-bitv current (untag (get (difference array-size n) array)))
     array (sub1 n) array-size)))

(prove-lemma lessp-1-exp (rewrite)
             (equal
              (lessp 1 (exp a b))
              (and (lessp 1 a) (not (zerop b))))
             ((enable exp)))

(prove-lemma bit-vectors-piton-means (rewrite)
             (implies
              (and
               (bit-vectors-piton state size)
               (lessp p (length state)))
              (and
               (equal (car (get p state)) 'bitv)
               (listp (get p state))
               (bit-vectorp (cadr (get p state)) size)
               (equal (cddr (get p state)) nil))))

(defn xor-bvs-loop-correctness-general-induct (i current n s data-segment)
  (if (zerop i) t
    (xor-bvs-loop-correctness-general-induct
     (sub1 i)
     (xor-bitv
      current
      (cadr (get (difference n i) (array s data-segment))))
     n s data-segment)))


;; 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)))

(enable bit-vectorp-xor-bitv)

(prove-lemma xor-bvs-loop-correctness-general nil
             (implies
              (and
               (lessp (length (array s data-segment))
                      (exp 2 word-size))
               (not (zerop word-size))
               (listp ctrl-stk)
               (bit-vectors-piton (array s data-segment) word-size)
               (at-least-morep (length temp-stk) 3 max-temp-stk-size)
               (equal (definition 'xor-bvs prog-segment)
```

```
                 (xor-bvs-program))
            (definedp s data-segment)
            (numberp i)
            (lessp i n)
            (bit-vectorp current word-size)
            (equal n (length (array s data-segment))))
           (equal
            (p (p-state '(pc (xor-bvs . 5))
                        (cons (list
                               (list
                                (cons 'vecs-addr
                                      (list 'addr
                                            (cons
                                             s
                                             (sub1
                                              (difference n i)))))
                                (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)))
          ((induct (xor-bvs-loop-correctness-general-induct
                    i current n s data-segment))))

(prove-lemma difference-x-sub1-x-better (rewrite)
          (equal (difference x (sub1 x))
                 (if (lessp 0 x) 1 0)))

(prove-lemma xor-bvs-loop-correctness nil
          (implies
           (and
            (lessp (length (array s data-segment))
                   (exp 2 word-size))
            (not (zerop word-size))
            (listp ctrl-stk)
            (bit-vectors-piton (array s data-segment) word-size)
            (at-least-morep (length temp-stk) 3 max-temp-stk-size)
            (equal (definition 'xor-bvs prog-segment)
                   (xor-bvs-program))
            (definedp s data-segment)
            (lessp 0 n)
            (bit-vectorp current word-size)
            (equal n (length (array s data-segment))))
           (equal
            (p (p-state '(pc (xor-bvs . 5))
                        (cons (list
                               (list
                                (cons 'vecs-addr
                                      (list 'addr (cons s 0)))
                                (cons 'numvecs
                                      (list 'nat (sub1 n))))
                               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 (sub1 n)))
            (p-state ret-pc
                     ctrl-stk
                     (cons (list 'bitv (xor-bvs-array
                                        current
                                        (array s data-segment)
                                        (sub1 n) n))
```

```
                        temp-stk)
                  prog-segment
                  data-segment
                  max-ctrl-stk-size
                  max-temp-stk-size
                  word-size
                  'run)))
          ((use (xor-bvs-loop-correctness-general (i (sub1 n))))))

(prove-lemma exp-0 (rewrite)
          (implies
           (zerop x)
           (and (equal (exp x y) (if (zerop y) 1 0))
               (equal (exp y x) 1)))
          ((enable exp)))

(prove-lemma bit-vectors-piton-means-more (rewrite)
          (implies
           (and
            (listp x)
            (bit-vectors-piton x size))
           (equal
            (LIST 'BITV (CADAR x))
            (car x))))

;; xor-bvs of an array of at least one bit vector
(defn xor-bvs (array)
  (if (listp array)
     (xor-bitv (car array) (xor-bvs (cdr array)))
    nil))


(defn untag-array (array)
  (if (listp array)
     (cons (untag (car array))
          (untag-array (cdr array)))
    nil))

(prove-lemma bit-vectorp-get (rewrite)
          (implies
           (bit-vectors-piton array size)
           (equal
            (bit-vectorp (untag (get n array)) size)
            (lessp 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))))))
(enable difference-sub1-arg2)

(prove-lemma xor-bitv-commutative (rewrite)
          (implies
           (equal (length a) (length b))
           (equal (xor-bitv a b) (xor-bitv b a))))

(prove-lemma xor-bitv-commutative2 (rewrite)
          (implies
           (equal (length a) (length b))
           (equal (xor-bitv a (xor-bitv b c))
                 (xor-bitv b (xor-bitv a c)))))

(prove-lemma xor-bitv-associative (rewrite)
          (implies
           (equal (length a) (length b))
           (equal (xor-bitv (xor-bitv a b) c)
                 (xor-bitv a (xor-bitv b c)))))


(prove-lemma length-from-bit-vectorp (rewrite)
          (implies
           (bit-vectorp x s)
           (equal (length x) (fix s))))

(prove-lemma length-xor-bitv (rewrite)
          (equal
           (length (xor-bitv a b))
           (length a)))

(prove-lemma length-cadr-get-bit-vectors-piton (rewrite)
          (implies
           (and
            (bit-vectors-piton x l)
            (lessp i (length x)))
           (equal (length (cadr (get i x))) (fix l))))
```

```
(prove-lemma equal-xor-bitv-x-x (rewrite)
            (implies
             (and
              (bit-vectorp b (length a))
              (bit-vectorp c (length a)))
             (equal
              (equal (xor-bitv a b) (xor-bitv a c))
              (equal b c))))

(defn bit-vectorp-induct (size a b)
  (if (zerop size) t
    (bit-vectorp-induct (sub1 size) (cdr a) (cdr b))))

(prove-lemma bit-vectorp-xor-bitv2 (rewrite)
            (equal
             (bit-vectorp (xor-bitv a b) size)
             (equal (length a) (fix size)))
            ((induct (bit-vectorp-induct size a b))))


(defn bit-vectorsp (bvs size)
  (if (listp bvs)
      (and
       (bit-vectorp (car bvs) size)
       (bit-vectorsp (cdr bvs) size))
    (equal bvs nil)))

(prove-lemma length-xor-bvs (rewrite)
            (implies
             (bit-vectorsp bvs (length (car bvs)))
             (equal (length (xor-bvs bvs)) (length (car bvs)))))

(prove-lemma bit-vectorsp-untag (rewrite)
            (implies
             (bit-vectors-piton x s)
             (bit-vectorsp (untag-array x) s)))

(prove-lemma bit-vectorsp-cdr-untag (rewrite)
            (implies
             (bit-vectors-piton (cdr x) s)
             (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))))
(enable nthcdr)

(prove-lemma bit-vectorsp-nthcdr (rewrite)
            (implies
             (and
              (bit-vectorsp x s)
              (lessp n (length x)))
             (bit-vectorsp (nthcdr n x) s))
            ((enable nthcdr)))

(prove-lemma bit-vectorp-xor-bvs (rewrite)
            (implies
             (and
              (bit-vectorsp x size)
              (listp x))
             (bit-vectorp (xor-bvs x) size)))

(prove-lemma length-untag-array (rewrite)
            (equal (length (untag-array x)) (length x)))

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

(prove-lemma nthcdr-open (rewrite)
            (implies
             (lessp n (length x))
             (equal
              (nthcdr n x)
              (cons (get n x) (nthcdr (add1 n) x))))
            ((enable nthcdr)))

(prove-lemma get-untag-array (rewrite)
            (implies
             (lessp n (length x))
```

```
            (equal
             (get n (untag-array x))
             (cadr (get n x)))))


(prove-lemma equal-xor-bitv-x-x-special (rewrite)
            (implies
             (and
              (bit-vectorp b (length a))
              (bit-vectorp (xor-bitv z c) (length a)))
             (equal
              (equal (xor-bitv a b) (xor-bitv z (xor-bitv a c)))
              (equal b (xor-bitv z c)))))

(defn fix-bit (b)
  (if (equal b 0) 0 1))

(prove-lemma xor-bitv-0 (rewrite)
            (and
             (equal (xor-bit x 0) (fix-bit x))
             (equal (xor-bit 0 x) (fix-bit x))))

(prove-lemma xor-bitv-nlistp (rewrite)
            (implies
             (not (listp c))
             (equal (xor-bitv a (xor-bitv b c))
                   (xor-bitv a b))))

(prove-lemma xor-bitv-nlistp2 (rewrite)
            (implies
             (and
              (bit-vectorp a b)
              (not (listp c)))
             (equal (xor-bitv a c) a)))

(prove-lemma xor-bvs-array-rewrite (rewrite)
            (implies
             (and
              (bit-vectors-piton array (length current))
              (bit-vectorp current (length current))
              (lessp n (length array))
              (equal (length array) as))
            (equal
             (xor-bvs-array current array n as)
             (xor-bitv current
                      (xor-bvs
                       (nthcdr (difference as n)
                              (untag-array array))))))
            ((induct (xor-bvs-array current array n as))
             (enable nthcdr)))

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

(prove-lemma correctness-of-xor-bvs-helper
 (rewrite)
 (implies
  (and (equal p0
             (p-state 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))
      (equal (p-current-instruction p0)
             '(call xor-bvs))
      (equal (definition 'xor-bvs prog-segment)
             (xor-bvs-program))
      (xor-bvs-input-conditionp p0))
  (equal
   (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)
                                    (sub1 numvecs)
                                    numvecs))
                temp-stk)
           prog-segment data-segment max-ctrl-stk-size max-temp-stk-size
           word-size 'run)))
 ((use
   (xor-bvs-loop-correctness (current (untag (car (array state data-segment))))
```

```
                      (s state)
                      (n numvecs)
                      (ret-pc (add-addr pc 1))))))

(prove-lemma length-cadar-bvs (rewrite)
           (implies
            (and
             (bit-vectors-piton x s)
             (listp x))
            (equal (length (cadar x)) (fix s))))

(prove-lemma bit-vectorp-from-bit-vectors-piton (rewrite)
           (implies
            (bit-vectors-piton x s)
            (and
             (equal
              (bit-vectorp (cadar x) s)
              (listp x))
             (equal
              (bit-vectorp (cadr (get n x)) s)
              (lessp n (length x))))))

(prove-lemma nthcdr-1 (rewrite)
           (equal
            (nthcdr 1 a)
            (cdr a))
           ((enable nthcdr)))

(prove-lemma listp-untag-array (rewrite)
           (equal
            (listp (untag-array x))
            (listp x)))

(prove-lemma xor-bvs-input-conditionp-means-xor-bvs-hack (rewrite)
           (implies
            (and
             (xor-bvs-input-conditionp
                      (p-state
                      pc
                      ctrl-stk
                      (cons (list 'nat numvecs)
                           (cons (list 'addr (cons state 0))
                                temp-stk))
                      prog-segment
                      data-segment
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run))
             (lessp 0 word-size))
            (equal
             (xor-bvs-array
              (untag
               (car (array state data-segment)))
              (array state data-segment)
              (sub1 numvecs) numvecs)
             (xor-bvs (untag-array (array state data-segment)))))
           ((enable nthcdr)))

(prove-lemma correctness-of-xor-bvs (rewrite)
           (implies
            (and
             (equal 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))
             (lessp 0 word-size)
             (equal (p-current-instruction p0) '(call xor-bvs))
             (equal (definition 'xor-bvs prog-segment)
                   (xor-bvs-program))
             (xor-bvs-input-conditionp p0))
            (equal
             (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)
            (xor-bvs-clock (cadr n)))
          (p-state (add1-addr pc)
                 ctrl-stk
                 (cons (list 'bitv
                           (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)))
        ((disable xor-bvs-clock)
         (use (correctness-of-xor-bvs-helper
              (state (caadr s)) (numvecs (cadr n)))
             (xor-bvs-input-conditionp-means-xor-bvs-hack
              (state (caadr s)) (numvecs (cadr n))))))

(defn 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.

(defn one-bit-vector (wordsize)
  (if (lessp wordsize 2)
      (list 1)
    (cons 0 (one-bit-vector (sub1 wordsize)))))

(defn push-1-vector-program (wordsize)
  (list 'push-1-vector nil nil
       (list 'push-constant (list 'bitv (one-bit-vector wordsize)))
       (list 'ret)))


(defn 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))

(defn push-1-vector-input-conditionp (p0)
  (and
   (not (lessp (p-max-ctrl-stk-size p0)
             (plus 2 (p-ctrl-stk-size (p-ctrl-stk p0)))))
   (not (lessp (p-max-temp-stk-size p0)
             (plus 1 (length (p-temp-stk p0)))))
   (listp (p-ctrl-stk p0))))

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

(prove-lemma equal-assoc-cons (rewrite)
            (implies
             (equal (assoc k a) (cons x y))
             (and
```

```
            (equal (car (assoc k a)) x)
            (equal (cdr (assoc k a)) y))))

(prove-lemma correctness-of-push-1-vector (rewrite)
           (implies
            (and
             (equal p0 (p-state
                        pc
                        ctrl-stk
                        temp-stk
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run))
             (equal (p-current-instruction p0)
                    '(call push-1-vector))
             (equal (definition 'push-1-vector prog-segment)
                    (push-1-vector-program word-size))
             (push-1-vector-input-conditionp p0))
            (equal
             (p p0 3)
             (p-state (add1-addr pc)
                      ctrl-stk
                      (cons (list 'bitv (one-bit-vector word-size))
                            temp-stk)
                      prog-segment
                      data-segment
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run))))

;;;; nat-to-bv

(defn nat-to-bv-program nil
  '(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 ()
            (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 ()
            (push-local current-bit))
            (lsh-bitv)
            (pop-local current-bit)
            (jump loop)
        (dl done ()
            (ret))))

(defn 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))

(defn zero-bit-vector (size)
  (if (zerop size) nil
    (cons 0 (zero-bit-vector (sub1 size)))))


(defn nat-to-bv2-helper (value current-bit bit-vector)
  (if (zerop value)
      bit-vector
    (nat-to-bv2-helper (quotient value 2)
```

```
                    (append (cdr current-bit) (list 0))
                    (if (equal (remainder value 2) 1)
                        (xor-bitv current-bit bit-vector)
                      bit-vector))))

(defn nat-to-bv2 (value size)
  (nat-to-bv2-helper value (one-bit-vector size)
                  (zero-bit-vector size)))

(defn nat-to-bv-loop-clock (value)
  (if (zerop value)
      3
    (plus
     (if (equal (remainder value 2) 0) 12 14)
     (nat-to-bv-loop-clock (quotient value 2)))))

(defn correctness-of-nat-to-bv-general-induct (value cb temp bv)
  (if (zerop value) t
    (correctness-of-nat-to-bv-general-induct
     (quotient value 2)
     (append (cdr cb) (list 0))
     (list 'nat (remainder value 2))
     (if (equal (remainder value 2) 0) bv (xor-bitv bv cb)))))

(prove-lemma lessp-remainder-simple (rewrite)
             (implies
              (not (zerop y))
              (lessp (remainder x y) y)))

(prove-lemma lessp-exp-simple (rewrite)
             (implies
              (and
               (lessp x y)
               (not (zerop z)))
              (lessp x (exp y z)))
             ((enable exp)))

(prove-lemma lessp-1 (rewrite)
             (equal
              (lessp x 1)
              (zerop x)))

(prove-lemma lessp-remainder-x-exp-x (rewrite)
             (equal
              (lessp (remainder x y) (exp y z))
              (or
               (and
                (zerop z)
                (equal (remainder x y) 0))
               (and
                (not (zerop z))
                (not (zerop y)))))
             ((enable exp)))


(prove-lemma bit-vectorp-append (rewrite)
             (implies
              (bit-vectorp x x-size)
              (equal
               (bit-vectorp (append x y) size)
               (and
                (bit-vectorp y (difference size x-size))
                (not (lessp size x-size))))))

(prove-lemma correctness-of-nat-to-bv-general (rewrite)
             (implies
              (and
               (listp ctrl-stk)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'nat-to-bv prog-segment)
                      (nat-to-bv-program))
               (numberp value)
               (lessp 0 word-size)
               (lessp value (exp 2 word-size))
               (bit-vectorp bv word-size)
               (bit-vectorp cb word-size))
              (equal
               (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)
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                 (nat-to-bv-loop-clock value))
              (p-state ret-pc
                      ctrl-stk
                      (cons (list 'bitv (nat-to-bv2-helper
                                          value cb bv))
                            temp-stk)
                      prog-segment
                      data-segment
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run)))
             ((induct (correctness-of-nat-to-bv-general-induct
                      value cb temp bv))))

(defn nat-to-bv-clock (value)
  (plus 9 (nat-to-bv-loop-clock value)))

(defn nat-to-bv-input-conditionp (p0)
  (and
   (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (numberp (cadr (top (p-temp-stk p0))))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                7 (p-max-ctrl-stk-size p0))
   (lessp 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))))

(prove-lemma bit-vectorp-one-bit-vector (rewrite)
             (equal
              (bit-vectorp (one-bit-vector s) s)
              (lessp 0 s)))

(prove-lemma bit-vectorp-zero-bit-vector (rewrite)
             (bit-vectorp (zero-bit-vector s) s))

(prove-lemma all-but-last-one-bit-vector (rewrite)
             (equal
              (all-but-last (one-bit-vector s))
              (zero-bit-vector (sub1 s))))

(prove-lemma correctness-of-nat-to-bv-helper nil
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons (list 'nat value) temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0) '(call nat-to-bv))
               (equal (definition 'nat-to-bv prog-segment)
                      (nat-to-bv-program))
               (equal (definition 'push-1-vector prog-segment)
                      (push-1-vector-program word-size))
               (nat-to-bv-input-conditionp p0))
              (equal
               (p p0 (nat-to-bv-clock value))
               (p-state (add1-addr pc)
                      ctrl-stk
                      (cons (list 'bitv
                                  (nat-to-bv2 value word-size))
                            temp-stk)
                      prog-segment
                      data-segment
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run))))

(defn bv-to-nat (bv)
  (if (listp bv)
      (plus (times (fix-bit (car bv)) (exp 2 (length (cdr bv))))
            (bv-to-nat (cdr bv)))
```

```
   0))

(prove-lemma length-nat-to-bv (rewrite)
          (equal (length (nat-to-bv nat size)) (fix size)))


(prove-lemma bv-to-nat-nat-to-bv (rewrite)
           (equal (bv-to-nat (nat-to-bv nat size))
                  (if (lessp nat (exp 2 size))
                      (fix nat)
                    (sub1 (exp 2 size))))
           ((enable exp)))

(enable exp)

(prove-lemma nat-to-bv-simple (rewrite)
          (implies
           (zerop x)
           (and
            (equal (nat-to-bv x y) (zero-bit-vector y))
            (equal (nat-to-bv y x) nil))))

(prove-lemma nat-to-bv-bv-to-nat (rewrite)
          (implies
           (bit-vectorp bv size)
           (equal (nat-to-bv (bv-to-nat bv) size) bv)))

(prove-lemma bv-to-nat-append (rewrite)
          (equal
           (bv-to-nat (append x y))
           (plus
            (bv-to-nat y)
            (times (exp 2 (length y)) (bv-to-nat x)))))

(prove-lemma lessp-plus-hacks (rewrite)
          (and
          (equal
           (lessp
            (plus a (plus b (plus c (plus (times d e) (times d e)))))
            (plus e e))
           (and
            (zerop d)
            (lessp (plus a (plus b c)) (plus e e))))
          (equal
           (lessp
            (plus a (plus b (plus c (plus e (plus (times d g) h)))))
            g)
           (and
            (zerop d)
            (lessp (plus a (plus b (plus c (plus e h)))) g)))))

(prove-lemma bv-to-nat-xor-bitv (rewrite)
          (implies
           (and
            (bit-vectorp x size)
            (bit-vectorp y size)
            (equal (and-bitv x y) (zero-bit-vector size)))
           (equal
            (bv-to-nat (xor-bitv x y))
            (plus (bv-to-nat x) (bv-to-nat y)))))

(prove-lemma lessp-bv-to-nat-exp-2 (rewrite)
          (implies
           (bit-vectorp x size)
           (equal (lessp (bv-to-nat x) (exp 2 size)) t)))

(prove-lemma equal-nat-to-bv (rewrite)
          (implies
           (and
            (bit-vectorp bv size)
            (lessp y (exp 2 size)))
           (and
            (equal
             (equal (nat-to-bv y size) bv)
             (equal (bv-to-nat bv) (fix y)))
            (equal
             (equal bv (nat-to-bv y size))
             (equal (fix y) (bv-to-nat bv))))))

(defn least-bit-higher-than-high-bit (x y)
  (if (listp x)
      (and
       (equal (length x) (length y))
       (if (not (equal (car y) 0))
           (all-zero-bitvp x)
         (least-bit-higher-than-high-bit (cdr x) (cdr y))))
    (nlistp y)))
```

```
(prove-lemma least-bit-higher-means-and-0 (rewrite)
            (implies
             (and
              (bit-vectorp x size)
              (bit-vectorp y size)
              (least-bit-higher-than-high-bit x y))
             (and
              (equal (and-bitv x y) (zero-bit-vector size))
              (equal (and-bitv y x) (zero-bit-vector size)))))

(prove-lemma all-zero-bitvp-zero-bit-vector (rewrite)
            (all-zero-bitvp (zero-bit-vector size)))

(prove-lemma bv-to-nat-one-bit-vector (rewrite)
            (equal (bv-to-nat (one-bit-vector size)) 1))

;; renamed from firstn to avoid conflict with similar but different
;; definition in piton library
(defn make-list-from (n list)
  (if (zerop n)
      nil
    (cons (car list) (make-list-from (sub1 n) (cdr list)))))

(prove-lemma length-make-list-from (rewrite)
            (equal (length (make-list-from n list)) (fix n)))

(prove-lemma make-list-from-1 (rewrite)
            (equal
             (make-list-from 1 x)
             (list (car x))))

(prove-lemma listp-make-list-from (rewrite)
            (equal
             (listp (make-list-from n x))
             (not (zerop n))))

(defn double-cdr-induct (x z)
  (if (listp x)
      (double-cdr-induct (cdr x) (cdr z))
    t))


(prove-lemma all-zero-bitvp-append (rewrite)
            (equal
             (all-zero-bitvp (append x y))
             (and
              (all-zero-bitvp x)
              (all-zero-bitvp y))))


(prove-lemma least-bit-higher-than-high-bit-append-0s (rewrite)
            (implies
             (and
              (all-zero-bitvp y)
              (equal (length z) (length (append x y))))
            (equal
             (least-bit-higher-than-high-bit (append x y) z)
             (least-bit-higher-than-high-bit
              x (make-list-from (length x) z))))
            ((induct (double-cdr-induct x z))))

(prove-lemma equal-xor-bitv-x-y-1 (rewrite)
            (equal
             (equal (xor-bitv a b) b)
             (and
              (bit-vectorp b (length a))
              (all-zero-bitvp a))))

(prove-lemma equal-xor-bitv-x-y-2 (rewrite)
            (equal
             (equal (xor-bitv a b) a)
             (and
              (all-zero-bitvp (make-list-from (length a) b))
              (bit-vectorp a (length a)))))

(prove-lemma least-bit-higher-than-high-bit-simple (rewrite)
            (implies
             (all-zero-bitvp x)
             (equal
              (least-bit-higher-than-high-bit x y)
              (equal (length x) (length y)))))

(prove-lemma make-list-from-is-all-but-last (rewrite)
            (implies
             (equal (length x) (add1 n))
             (equal
```

```
              (make-list-from n x)
              (all-but-last x))))

(prove-lemma least-bit-higher-all-but-last (rewrite)
             (implies
              (least-bit-higher-than-high-bit a b)
              (equal
               (least-bit-higher-than-high-bit
                a (cons 0 (all-but-last b)))
               (listp a))))

(defn at-most-one-bit-on (bv)
  (if (listp bv)
      (if (equal (car bv) 0)
          (at-most-one-bit-on (cdr bv))
        (all-zero-bitvp (cdr bv)))
    t))

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

(prove-lemma listp-xor-bitv (rewrite)
             (equal
              (listp (xor-bitv a b))
              (listp a)))

(prove-lemma all-zero-bitvp-means-at-most-one-bit-on (rewrite)
             (implies
              (all-zero-bitvp x)
              (at-most-one-bit-on x)))

(prove-lemma at-most-one-bit-on-append (rewrite)
             (equal
              (at-most-one-bit-on (append a b))
              (or
               (and
                (all-zero-bitvp a)
                (at-most-one-bit-on b))
               (and
                (at-most-one-bit-on a)
                (all-zero-bitvp b)))))

(defn fix-bitv (list)
  (if (listp list)
      (cons (if (equal (car list) 0) 0 1) (fix-bitv (cdr list)))
    nil))

(prove-lemma xor-bitv-nlistp3 (rewrite)
             (implies
              (not (listp x))
              (and
               (equal (xor-bitv x y) nil)
               (equal (xor-bitv y x) (fix-bitv y)))))

(disable xor-bitv-nlistp)
(disable xor-bitv-nlistp2)

(prove-lemma fix-bitv-all-but-last (rewrite)
             (equal
              (fix-bitv (all-but-last x))
              (all-but-last (fix-bitv x))))

(prove-lemma all-but-last-xor-bitv (rewrite)
             (equal
              (all-but-last (xor-bitv a b))
              (if (lessp (length b) (length a))
                  (xor-bitv (all-but-last a) b)
                (xor-bitv (all-but-last a) (all-but-last b)))))

(prove-lemma least-bit-higher-cons-xor-bitv-hack (rewrite)
             (implies
              (and
               (least-bit-higher-than-high-bit (cdr x) a)
               (least-bit-higher-than-high-bit (cdr x) b))
              (equal
               (least-bit-higher-than-high-bit
                x (cons 0 (xor-bitv a b)))
               (listp x))))

(prove-lemma least-bit-higher-cdr-all-but-last (rewrite)
             (implies
              (least-bit-higher-than-high-bit a b)
              (least-bit-higher-than-high-bit (cdr a) (all-but-last b))))
```

```
(prove-lemma least-bit-higher-x-all-but-last-x (rewrite)
            (implies
             (at-most-one-bit-on x)
             (least-bit-higher-than-high-bit
              (cdr x) (all-but-last x))))

(prove-lemma nat-to-bv-equiv-helper nil
            (implies
             (and
              (bit-vectorp cb size)
              (bit-vectorp bv size)
              (lessp 0 size)
              (lessp (plus (bv-to-nat bv)
                           (times (bv-to-nat cb) value))
                     (exp 2 size))
              (at-most-one-bit-on cb)
              (least-bit-higher-than-high-bit cb bv))
            (equal
             (nat-to-bv2-helper value cb bv)
             (nat-to-bv (plus (times (bv-to-nat cb) value)
                              (bv-to-nat bv))
                        size)))
            ((induct (nat-to-bv2-helper value cb bv))))

(prove-lemma at-most-one-bit-on-one-bit-vector (rewrite)
            (at-most-one-bit-on (one-bit-vector size)))

(prove-lemma bv-to-nat-all-zero-bitvp (rewrite)
            (implies
             (all-zero-bitvp x)
             (equal (bv-to-nat x) 0)))


(prove-lemma least-bit-higher-than-high-bit-simple2 (rewrite)
            (implies
             (all-zero-bitvp x)
             (equal
              (least-bit-higher-than-high-bit y x)
              (equal (length x) (length y)))))

(prove-lemma length-zero-bit-vector (rewrite)
            (equal (length (zero-bit-vector size)) (fix size)))

(prove-lemma length-one-bit-vector (rewrite)
            (equal (length (one-bit-vector size))
                   (if (zerop size) 1 size)))

(prove-lemma nat-to-bv-equivalence (rewrite)
            (implies
             (and
              (lessp value (exp 2 size))
              (lessp 0 size))
            (equal
             (nat-to-bv2 value size)
             (nat-to-bv value size)))
            ((use (nat-to-bv-equiv-helper
                   (cb (one-bit-vector size))
                   (bv (zero-bit-vector size))))))

(prove-lemma correctness-of-nat-to-bv (rewrite)
            (implies
             (and
              (equal 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))
              (equal (car v) 'nat)
              (equal (cddr v) nil)
              (equal (p-current-instruction p0) '(call nat-to-bv))
              (equal (definition 'nat-to-bv prog-segment)
                     (nat-to-bv-program))
              (equal (definition 'push-1-vector prog-segment)
                     (push-1-vector-program word-size))
              (equal num (cadr v))
              (nat-to-bv-input-conditionp p0))
             (equal
              (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)))
            ((disable-theory t)
             (enable nat-to-bv-input-conditionp p-state
                    p-word-size-p-state p-temp-stk-p-state top)
             (enable-theory ground-zero)
             (use (correctness-of-nat-to-bv-helper (value (cadr v)))
                  (nat-to-bv-equivalence (size word-size)
                                         (value (cadr v))))))

;;;; bv-to-nat

(defn bv-to-nat-program nil
  '(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 ()
             (push-local bv))
             (push-local current-bit)
             (and-bitv)
             (test-bitv-and-jump all-zero lab)
             (push-local current-2power)
             (add-nat)
          (dl lab ()
             (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 ()
             (ret))))

(defn 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))

(defn trailing-zeros-helper (list acc)
  (if (listp list)
      (trailing-zeros-helper
       (cdr list) (if (equal (car list) 0) (add1 acc) 0))
    (fix acc)))

(defn trailing-zeros (list)
  (trailing-zeros-helper list 0))

(prove-lemma trailing-zeros-of-all-zero-bitvp (rewrite)
            (implies
             (all-zero-bitvp bv)
             (equal (trailing-zeros-helper bv n)
                    (plus (length bv) n))))

(prove-lemma non-zero-means-acc-irrelevant-spec (rewrite)
```

```
         (implies
          (not (all-zero-bitvp bv))
          (equal (trailing-zeros-helper bv (add1 x))
                 (trailing-zeros-helper bv x))))

(prove-lemma non-zero-means-acc-irrelevant (rewrite)
         (implies
          (and
           (not (all-zero-bitvp bv))
           (not (equal x 0)))
          (equal (trailing-zeros-helper bv x)
                 (trailing-zeros-helper bv 0))))

(prove-lemma trailing-zeros-helper-append (rewrite)
         (equal
          (trailing-zeros-helper (append x y) acc)
          (if (all-zero-bitvp y)
              (plus (trailing-zeros-helper x acc) (length y))
            (trailing-zeros-helper y acc))))

(prove-lemma trailing-zeros-append (rewrite)
         (equal
          (trailing-zeros (append x y))
          (if (all-zero-bitvp y)
              (plus (trailing-zeros x) (length y))
            (trailing-zeros y))))

(prove-lemma lessp-trailing-zeros-helper (rewrite)
         (implies
          (not (lessp n (plus acc (length x))))
          (equal (lessp n (trailing-zeros-helper x acc)) f)))

(defn last (list)
  (if (listp list)
      (if (listp (cdr list))
          (last (cdr list))
        (car list))
    0))

(prove-lemma equal-trailing-zeros-helper-0 (rewrite)
         (equal
          (equal (trailing-zeros-helper x acc) 0)
          (or (and (zerop acc) (nlistp x))
              (not (equal (last x) 0)))))

(prove-lemma lessp-length-cdr-trailing (rewrite)
         (implies
          (and
           (lessp (length (cdr x)) (trailing-zeros-helper x acc))
           (listp x))
          (and
           (all-zero-bitvp x)
           (all-zero-bitvp (cdr x)))))

(prove-lemma not-all-zero-bitvp-cdr-means (rewrite)
         (implies
          (not (all-zero-bitvp (cdr x)))
          (equal
           (trailing-zeros-helper x acc)
           (trailing-zeros-helper (cdr x) 0))))

(defn bv-to-nat2-helper (bv cb current-2power)
  (if (all-zero-bitvp cb)
      0
    (plus
     (if (all-zero-bitvp (and-bitv bv cb)) 0 current-2power)
     (bv-to-nat2-helper
      bv (append (cdr cb) (list 0)) (times 2 current-2power))))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn bv-to-nat2 (bv)
  (bv-to-nat2-helper bv (one-bit-vector (length bv)) 1))

(prove-lemma lessp-length-trailing-zeros-hack (rewrite)
         (implies
          (zerop acc)
          (not (lessp (length x) (trailing-zeros-helper x acc)))))

(defn bv-to-nat-loop-clock (cb bv)
  (if (all-zero-bitvp (cdr cb))
      (if (or (equal (car cb) 0) (equal (car bv) 0)) 9 11)
    (plus (if (all-zero-bitvp (and-bitv cb bv)) 12 14)
          (bv-to-nat-loop-clock (append (cdr cb) '(0)) bv)))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn bv-to-nat-induct (value bv cb current-2power)
  (if (all-zero-bitvp cb)
```

```
     0
    (bv-to-nat-induct
     (plus (if (all-zero-bitvp (and-bitv bv cb)) 0 current-2power)
           value)
     bv (append (cdr cb) (list 0)) (times 2 current-2power)))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn double-cdr-with-sub1-induct (x y n)
  (if (listp x)
     (double-cdr-with-sub1-induct (cdr x) (cdr y) (sub1 n))
    t))

(prove-lemma bit-vectorp-and-bitv-better (rewrite)
             (equal
              (bit-vectorp (and-bitv x y) size)
              (equal (length x) (fix size)))
             ((induct (double-cdr-with-sub1-induct x y size))))

(prove-lemma lessp-bv-to-nat-exp (rewrite)
             (implies
              (bit-vectorp x size)
              (lessp (bv-to-nat x) (exp 2 size))))

(prove-lemma equal-bv-to-nat-0 (rewrite)
             (implies
              (bit-vectorp x size)
             (equal
              (equal (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))))
(enable commutativity-of-and-bitv)

(prove-lemma commutativity2-of-and-bitv (rewrite)
             (implies
              (equal (length x) (length y))
              (equal (and-bitv x (and-bitv y z))
                     (and-bitv y (and-bitv x z)))))

(prove-lemma associativity-of-and-bitv (rewrite)
             (implies
              (equal (length x) (length y))
             (equal (and-bitv (and-bitv x y) z)
                    (and-bitv x (and-bitv y z)))))

(prove-lemma all-zero-bitvp-and-bitv (rewrite)
             (implies
              (all-zero-bitvp x)
              (and
               (all-zero-bitvp (and-bitv x y))
               (all-zero-bitvp (and-bitv y x)))))

(prove-lemma bv-to-nat-loop-clock-open (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (bv-to-nat-loop-clock x y) 9)))

(prove-lemma bv-to-nat2-helper-hack (rewrite)
             (implies
              (and
               (all-zero-bitvp z)
               (equal (length d) (length z)))
              (equal (bv-to-nat2-helper (cons 1 d) (cons 1 z) v)
                     (fix v)))
             ((expand (bv-to-nat2-helper (cons 1 d) (cons 1 z) v))))

(prove-lemma bv-to-nat2-helper-hack2 (rewrite)
             (implies
              (and
               (all-zero-bitvp z)
               (equal (length d) (length z)))
              (equal (bv-to-nat2-helper (cons 0 d) (cons 1 z) v) 0))
             ((expand (bv-to-nat2-helper (cons 0 d) (cons 1 z) v))))

(prove-lemma correctness-of-bv-to-nat-general (rewrite)
             (implies
              (and
               (listp ctrl-stk)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'bv-to-nat prog-segment)
                      (bv-to-nat-program))
               (numberp c2p)
               (lessp 0 word-size)
```

```
             (at-most-one-bit-on cb)
             (equal c2p (bv-to-nat cb))
             (bit-vectorp bv word-size)
             (bit-vectorp cb word-size)
             (numberp value)
             (lessp value c2p))
            (equal
             (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
                             (plus (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)))
            ((induct (bv-to-nat-induct value bv cb c2p))))


(defn bv-to-nat-clock (word-size bv)
  (plus 8 (bv-to-nat-loop-clock (one-bit-vector word-size) bv)))

(defn bv-to-nat-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'bitv)
   (equal (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))
   (lessp 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))))

(prove-lemma correctness-of-bv-to-nat-helper nil
            (implies
             (and
              (equal 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))
              (equal (p-current-instruction p0) '(call bv-to-nat))
              (equal (definition 'bv-to-nat prog-segment)
                     (bv-to-nat-program))
              (equal (definition 'push-1-vector prog-segment)
                     (push-1-vector-program word-size))
              (bv-to-nat-input-conditionp p0))
             (equal
              (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))))
```

```
(prove-lemma bit-vectorp-append-better (rewrite)
           (implies
            (bit-vectorp x (length x))
            (equal (bit-vectorp (append x y) size)
                   (and
                    (bit-vectorp y (length y))
                    (equal (fix size)
                           (plus (length x) (length y))))))
            ((induct (make-list-from size x))))

(prove-lemma bit-vectorp-hack (rewrite)
           (equal
            (bit-vectorp x (plus a (length (cdr x))))
            (and
             (if (listp x) (equal a 1) (zerop a))
             (bit-vectorp x (length x))))
           ((use (length-from-bit-vectorp
                  (x x) (s (plus a (length (cdr x))))))
            (disable length-from-bit-vectorp)))

(prove-lemma bv-to-nat-one-bit (rewrite)
           (implies
            (and
             (not (all-zero-bitvp x))
             (at-most-one-bit-on x)
             (bit-vectorp x size))
            (equal (bv-to-nat x) (exp 2 (trailing-zeros x)))))

(prove-lemma lessp-sub1-plus-hack (rewrite)
           (equal
            (lessp (sub1 x) (plus y x))
            (or (not (zerop x)) (not (zerop y)))))

(prove-lemma quotient-plus-hack (rewrite)
           (equal
            (quotient (plus a b b) 2)
            (plus (quotient a 2) b)))

(prove-lemma bv-to-nat-all-but-last (rewrite)
           (implies
            (bit-vectorp x size)
           (equal
            (bv-to-nat (all-but-last x))
            (quotient (bv-to-nat x) 2))))

(prove-lemma make-list-from-cons (rewrite)
           (equal
            (make-list-from n (cons a b))
            (if (zerop n) nil (cons a (make-list-from (sub1 n) b)))))

(prove-lemma equal-plus-times-hack (rewrite)
           (equal
            (equal (plus a (times a b) (times a c)) (times a d))
            (or (zerop a) (equal (plus 1 b c) d)))
           ((use (equal-times-arg1 (a a) (x (plus 1 b c)) (y d)))))

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

(prove-lemma last-make-list-from (rewrite)
           (equal (last (make-list-from n x))
                  (if (zerop n) 0 (nth (sub1 n) x))))

(prove-lemma make-list-from-nlistp (rewrite)
           (implies
            (nlistp x)
            (equal (make-list-from n x) (zero-bit-vector n))))

(prove-lemma nth-nlistp (rewrite)
           (implies
            (nlistp x)
            (equal (nth n x) 0)))

(prove-lemma bit-vectorp-make-list-from (rewrite)
           (implies
            (bit-vectorp x size)
            (bit-vectorp (make-list-from n x) n)))

(prove-lemma equal-bv-to-nat-0-2 (rewrite)
           (implies
            (bit-vectorp x (length x))
            (equal
             (equal (bv-to-nat x) 0)
             (all-zero-bitvp x))))
```

```
(prove-lemma bv-to-nat-make-list-from-from-sub1-make-list-from (rewrite)
            (implies
             (equal (bv-to-nat (make-list-from (sub1 n) v)) 0)
             (equal (bv-to-nat (make-list-from n v))
                    (if (zerop n) 0 (fix-bit (nth (sub1 n) v))))))

(prove-lemma plus-bv-to-nat-make-list-from (rewrite)
            (equal
             (plus (bv-to-nat (make-list-from (sub1 z) v))
                   (bv-to-nat (make-list-from (sub1 z) v)))
             (difference (bv-to-nat (make-list-from z v))
                         (fix-bit (last (make-list-from z v)))))
            ((induct (make-list-from z v))))

(prove-lemma equal-bv-to-nat-1 (rewrite)
            (implies
             (bit-vectorp x (length x))
            (equal
             (equal (bv-to-nat x) 1)
             (and
              (all-zero-bitvp (all-but-last x))
              (equal (last x) 1))))
            ((induct (all-but-last x))))

(prove-lemma lessp-1-hack (rewrite)
            (equal
             (lessp 1 a)
             (and
              (not (zerop a))
              (not (equal a 1)))))

(prove-lemma not-equal-nth-0-means (rewrite)
            (implies
             (and
              (not (equal (nth n v) 0))
              (lessp n s))
             (not (all-zero-bitvp (make-list-from s v)))))

(prove-lemma all-zero-bitvp-all-but-last-means nil
            (implies
             (and
              (all-zero-bitvp (all-but-last x))
              (equal (length x) (length y))
              (lessp 0 (trailing-zeros-helper y acc)))
             (all-zero-bitvp (and-bitv x y))))

(prove-lemma all-zero-bitvp-all-but-last-means-spec (rewrite)
            (implies
             (and
              (all-zero-bitvp (all-but-last x))
              (equal (length x) (length y))
              (lessp 0 (trailing-zeros-helper y 0)))
             (all-zero-bitvp (and-bitv x y)))
            ((use (all-zero-bitvp-all-but-last-means (acc 0)))))

(prove-lemma all-zero-means-to-and-bitv (rewrite)
            (implies
             (all-zero-bitvp x)
             (and
              (equal (and-bitv x y) (zero-bit-vector (length x)))
              (equal (and-bitv y x) (zero-bit-vector (length y))))))

(prove-lemma trailing-zeros-nth-proof nil
            (implies
             (and
              (equal n
                     (sub1 (difference (length x)
                                       (trailing-zeros-helper x acc))))
              (not (all-zero-bitvp x)))
             (not (equal (nth n x) 0))))


(prove-lemma trailing-zeros-nth-spec (rewrite)
            (implies
             (and
              (equal n
                     (sub1 (difference (length x)
                                       (trailing-zeros-helper x 0))))
              (not (all-zero-bitvp x)))
             (not (equal (nth n x) 0)))
            ((use (trailing-zeros-nth-proof (acc 0)))))

(prove-lemma and-bitv-special (rewrite)
            (implies
             (and
              (equal (nth n w) 0)
```

```
        (not (equal (nth n x) 0))
        (equal (length x) (length x))
        (at-most-one-bit-on x))
       (equal (and-bitv w x) (zero-bit-vector (length w)))))


(prove-lemma equal-trailing-zeros-length-spec nil
       (equal
        (equal (trailing-zeros-helper x acc) (plus acc (length x)))
        (all-zero-bitvp x)))

(prove-lemma equal-trailing-zeros-length (rewrite)
       (implies
        (zerop acc)
       (equal
        (equal (trailing-zeros-helper x acc) (length x))
        (all-zero-bitvp x)))
       ((use (equal-trailing-zeros-length-spec))))



(prove-lemma bit-vectorp-trailing-zeros (rewrite)
       (equal
        (bit-vectorp x (trailing-zeros-helper x acc))
        (and
         (all-zero-bitvp x)
         (bit-vectorp x (length x))
         (zerop acc)))
       ((use (equal-trailing-zeros-length-spec)
             (length-from-bit-vectorp
              (s (trailing-zeros-helper x acc))))
        (disable length-from-bit-vectorp)))


(prove-lemma quotient-exp-hack (rewrite)
       (implies
        (not (zerop b))
       (equal (quotient (plus x (exp b y)) b)
              (if (zerop y)
                  (quotient (add1 x) b)
                (plus (exp b (sub1 y)) (quotient x b))))))

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

(prove-lemma bit-vectorp-means-properp (rewrite)
       (implies
        (bit-vectorp x (length x))
        (properp x)))

(prove-lemma make-list-from-simplify (rewrite)
       (implies
        (and
         (equal n (length x))
         (properp x))
        (equal (make-list-from n x) x)))

(prove-lemma equal-add1-plus-hack (rewrite)
       (and
        (equal
         (equal (add1 (plus a b)) (plus c (plus d a)))
         (equal (add1 b) (plus c d)))
        (equal
         (equal (add1 (plus a b)) (plus c a))
         (equal (add1 b) (fix c)))
        (equal
         (equal (add1 a) (plus b a))
         (equal 1 (fix b)))))

(prove-lemma plus-quotient-bv-to-nat (rewrite)
       (equal
        (plus (quotient (bv-to-nat x) 2)
              (quotient (bv-to-nat x) 2))
        (difference (bv-to-nat x) (fix-bit (last x)))))

(prove-lemma equal-plus-times-hack2 (rewrite)
       (equal
        (equal (plus (times a b) (times a c)) (times a d))
        (or (zerop a)
            (equal (plus b c) (fix d))))
       ((use (equal-times-arg1 (a a) (x (plus b c)) (y d)))))

(prove-lemma and-bitv-special-special (rewrite)
       (implies
        (and
```

```
              (equal (last w) 0)
              (not (equal (last x) 0))
              (equal (length x) (length w))
              (at-most-one-bit-on x))
             (equal (and-bitv w x) (zero-bit-vector (length w)))))

(prove-lemma and-bitv-special-3 (rewrite)
            (implies
             (and
              (equal (length x) (length y))
              (not (equal (last x) 0))
              (not (equal (last y) 0)))
             (not (all-zero-bitvp (and-bitv x y)))))

(prove-lemma and-bitv-special-4 (rewrite)
            (implies
             (and
              (not (equal (nth n w) 0))
              (not (equal (nth n x) 0))
              (equal (length x) (length x)))
              (not (all-zero-bitvp (and-bitv w x)))))

(prove-lemma and-bitv-special-5 (rewrite)
            (implies
             (and
              (equal (last w) 0)
              (not (equal (last x) 0))
              (equal (add1 (length x)) (length w))
              (at-most-one-bit-on x))
             (equal (and-bitv w (cons 0 x))
                    (zero-bit-vector (length w)))))

(prove-lemma and-bitv-special-6 (rewrite)
            (implies
             (and
              (not (equal (last w) 0))
              (not (equal (last x) 0))
              (equal (add1 (length x)) (length w)))
             (not (all-zero-bitvp (and-bitv w (cons 0 x))))))

(prove-lemma not-last-0-means-not-all-0 (rewrite)
            (implies
             (not (equal (last x) 0))
             (not (all-zero-bitvp x))))

(prove-lemma bit-vectorp-plus-length-hack (rewrite)
            (and
            (equal
             (bit-vectorp x (plus z (length x)))
             (and
              (bit-vectorp x (length x))
              (zerop z)))
            (equal
             (bit-vectorp x (plus (add1 z) (length (cdr x))))
             (and
              (bit-vectorp x (length x))
              (listp x)
              (zerop z)))))

(prove-lemma last-append (rewrite)
            (equal
             (last (append x y))
             (if (listp y) (last y) (last x))))

(prove-lemma bv-to-nat2-helper-bv-to-nat (rewrite)
            (implies
             (and
              (at-most-one-bit-on cb)
              (bit-vectorp cb size)
              (bit-vectorp bv size)
              (equal c2 (bv-to-nat cb)))
             (equal
              (bv-to-nat2-helper bv cb c2)
              (bv-to-nat (append
                         (make-list-from (difference (length bv)
                                                     (trailing-zeros cb))
                                 bv)
                         (zero-bit-vector (trailing-zeros cb)))))))

(prove-lemma bv-to-nat2-helper-bv-to-nat-better (rewrite)
            (implies
             (and
              (at-most-one-bit-on cb)
              (bit-vectorp cb (length cb))
              (bit-vectorp bv (length cb))
              (equal c2 (bv-to-nat cb)))
             (equal
```

```
            (bv-to-nat2-helper bv cb c2)
            (bv-to-nat (append
                       (make-list-from (difference (length bv)
                                                   (trailing-zeros cb))
                              bv)
                       (zero-bit-vector (trailing-zeros cb)))))))

(prove-lemma nlistp-bv-to-nat2 (rewrite)
           (implies
            (not (listp x))
            (equal (bv-to-nat2-helper x a b) 0)))

(prove-lemma bv-to-nat2-bv-to-nat-helper nil
           (implies (nlistp x)
                   (equal (bv-to-nat2 x) (bv-to-nat x))))

(prove-lemma bit-vectorp-one-bit-vector-rewrite (rewrite)
           (equal
            (bit-vectorp (one-bit-vector s) n)
            (if (zerop s) (equal n 1) (equal (fix s) (fix n))))
           ((induct (lessp s n))))

(prove-lemma trailing-zeros-helper-one-bit-vector (rewrite)
           (equal
            (trailing-zeros-helper (one-bit-vector n) acc)
            0))

(prove-lemma bv-to-nat2-bv-to-nat (rewrite)
           (implies
            (bit-vectorp x (length x))
            (equal (bv-to-nat2 x) (bv-to-nat x)))
           ((use (bv-to-nat2-bv-to-nat-helper))))

(prove-lemma bv-length-weaker (rewrite)
           (implies
            (bit-vectorp x s)
            (bit-vectorp x (length x))))

(prove-lemma correctness-of-bv-to-nat (rewrite)
           (implies
            (and
             (equal 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))
             (equal (p-current-instruction p0) '(call bv-to-nat))
             (equal (definition 'bv-to-nat prog-segment)
                   (bv-to-nat-program))
             (equal (definition 'push-1-vector prog-segment)
                   (push-1-vector-program word-size))
             (equal bv (cadr b))
             (bv-to-nat-input-conditionp p0))
            (equal
             (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)))
           ((use (correctness-of-bv-to-nat-helper)
                (bv-to-nat2-bv-to-nat (x bv)))
            (disable-theory t)
            (enable bv-to-nat-input-conditionp p-state
                   p-word-size-p-state bv-length-weaker
                   top p-temp-stk-p-state)
            (enable-theory ground-zero)))
```

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

(defn number-with-at-least-program nil
  '(number-with-at-least (nums-addr numnums min) ((i (nat 0)))
          (push-constant (nat 0))
          (set-local i)
     (dl loop ()
          (push-local nums-addr))
          (fetch)
          (push-local min)
          (lt-nat)
          (test-bool-and-jump t lab)
          (add1-nat)
      (dl lab ()
          (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 ()
          (ret))))

(defn 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))

(defn number-with-at-least (numlist min)
  (if (listp numlist)
      (if (lessp (car numlist) min)
          (number-with-at-least (cdr numlist) min)
        (add1 (number-with-at-least (cdr numlist) min)))
    0))

(defn nat-list-piton (array word-size)
  (if (listp array)
      (and
       (listp (car array))
       (equal (caar array) 'nat)
       (numberp (cadar array))
       (equal (cddar array) nil)
       (lessp (cadar array) (exp 2 word-size))
       (nat-list-piton (cdr array) word-size))
    (equal array nil)))

(defn number-with-at-least-general-induct (i current n s min data-segment)
  (if (lessp i n)
      (number-with-at-least-general-induct
       (add1 i)
       (if (lessp (cadr (get i (array s data-segment))) min)
           current (add1 current))
       n s min data-segment)
    t)
  ((lessp (difference n i))))

(defn number-with-at-least-clock-loop (i min array)
  (if (not (lessp i (length array))) 0
    (plus
     (if (lessp (cadr (get i array)) min) 0 1)
     (if (equal (add1 i) (length array))
         12
       (plus 16 (number-with-at-least-clock-loop (add1 i) min array)))))
  ((lessp (difference (length array) i))))

(prove-lemma equal-difference-1 (rewrite)
             (equal
              (equal (difference x y) 1)
              (equal x (add1 y))))
```

```
(prove-lemma nat-list-piton-means (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (lessp p (length state)))
              (and
               (equal (car (get p state)) 'nat)
               (listp (get p state))
               (numberp (cadr (get p state)))
               (lessp (cadr (get p state)) (exp 2 size))
               (equal (cddr (get p state)) nil))))

(prove-lemma number-with-at-least-nlistp (rewrite)
             (implies
              (not (listp x))
              (equal (number-with-at-least x min) 0)))

(prove-lemma equal-add1-length (rewrite)
             (equal
              (equal (add1 x) (length y))
              (and
               (listp y)
               (equal (fix x) (length (cdr y))))))

(disable number-with-at-least-clock-loop)


(prove-lemma length-cdr-untag-array (rewrite)
             (equal
              (length (cdr (untag-array x)))
              (length (cdr x))))

(prove-lemma number-with-at-least-correctness-general nil
             (implies
              (and
               (lessp (length (array s data-segment))
                      (exp 2 word-size))
               (not (zerop word-size))
               (listp ctrl-stk)
               (nat-list-piton (array s data-segment) word-size)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'number-with-at-least prog-segment)
                      (number-with-at-least-program))
               (definedp s data-segment)
               (lessp min (exp 2 word-size))
               (numberp min)
               (lessp i n)
               (numberp i)
               (lessp n (exp 2 word-size))
               (numberp current)
               (not (lessp i current))
               (equal n (length (array s data-segment))))
              (equal
               (p (p-state '(pc (number-with-at-least . 2))
                           (cons (list
                                  (list
                                   (cons 'nums-addr
                                         (list 'addr (cons s i)))
                                   (cons 'numnums (list 'nat n))
                                   (cons 'min (list 'nat min))
                                   (cons 'i (list 'nat i)))
                                  ret-pc)
                                 ctrl-stk)
                           (cons (list 'nat current)
                                 temp-stk)
                           prog-segment
                           data-segment
                           max-ctrl-stk-size
                           max-temp-stk-size
                           word-size
                           'run)
                  (number-with-at-least-clock-loop
                   i min (array s data-segment)))
               (p-state ret-pc
                        ctrl-stk
                        (cons
                         (list 'nat
                               (plus current
                               (number-with-at-least
                                (nthcdr i (untag-array
                                           (array s data-segment)))
                                min)))
                         temp-stk)
                        prog-segment
```

```
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)))
           ((induct (number-with-at-least-general-induct
                     i current n s min data-segment))
            (expand (NUMBER-WITH-AT-LEAST-CLOCK-LOOP
                     I MIN (CDR (ASSOC S DATA-SEGMENT))))))

(defn number-with-at-least-clock (min array)
  (plus 3 (number-with-at-least-clock-loop 0 min array)))

(defn number-with-at-least-input-conditionp (p0)
  (and
   (equal (car (car (p-temp-stk p0))) 'nat)
   (equal (cddr (car (p-temp-stk p0))) nil)
   (equal (car (cadr (p-temp-stk p0))) 'nat)
   (equal (cddr (cadr (p-temp-stk p0))) nil)
   (equal (car (caddr (p-temp-stk p0))) 'addr)
   (equal (cdadr (caddr (p-temp-stk p0))) 0)
   (equal (cddr (caddr (p-temp-stk p0))) nil)
   (listp (cadr (caddr (p-temp-stk p0))))
   (lessp (length (array (car (cadr (caddr (p-temp-stk p0))))
                  (p-data-segment p0)))
          (exp 2 (p-word-size p0)))
   (not (zerop (p-word-size p0)))
   (listp (p-ctrl-stk p0))
   (nat-list-piton (array (car (cadr (caddr (p-temp-stk p0))))
                   (p-data-segment p0))
                (p-word-size 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))
                0 (p-max-temp-stk-size p0))
   (equal (definition 'number-with-at-least (p-prog-segment p0))
          (number-with-at-least-program))
   (definedp (car (cadr (caddr (p-temp-stk p0)))) (p-data-segment p0))
   (lessp (cadr (car (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (numberp (cadr (car (p-temp-stk p0))))
   (lessp 0 (cadr (cadr (p-temp-stk p0))))
   (equal (cadr (cadr (p-temp-stk p0)))
          (length (array (car (cadr (caddr (p-temp-stk p0))))
                  (p-data-segment p0))))))

(prove-lemma correctness-of-number-with-at-least (rewrite)
           (implies
            (and
             (equal 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))
             (equal (p-current-instruction p0)
                    '(call number-with-at-least))
             (number-with-at-least-input-conditionp p0)
             (equal minc (cadr m))
             (equal arrayc (array (car (cadr s)) data-segment)))
            (equal
             (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)))
           ((use (number-with-at-least-correctness-general
                 (i 0) (current 0) (s (car (cadr s)))
                 (ret-pc (add1-addr pc))
                 (n (cadr n)) (min (cadr m))))
            (disable number-with-at-least-clock-loop)))

;; highest-bit


(defn highest-bit-program nil
  '(highest-bit (bv) ((cb (nat 0)))
          (call push-1-vector)
          (set-local cb)
          (rsh-bitv)
     (dl loop ()
          (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 ()
          (push-local cb))
          (lsh-bitv)
          (pop-local cb)
          (jump loop)
       (dl done ()
          (ret))))

(defn 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))

(defn highest-bit (x)
  (if (listp x)
      (if (equal (car x) 0)
          (cons 0 (highest-bit (cdr x)))
        (cons 1 (zero-bit-vector (length (cdr x)))))
    nil))

(prove-lemma listp-highest-bit (rewrite)
          (equal (listp (highest-bit x)) (listp x)))

(prove-lemma length-highest-bit (rewrite)
          (equal (length (highest-bit x)) (length x)))

(defn highest-bit-loop-clock (cb bv)
  (if (all-zero-bitvp cb)
      3
    (plus
     10 (if (all-zero-bitvp (and-bitv cb bv)) 0 2)
     (highest-bit-loop-clock (append (cdr cb) '(0)) bv)))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn highest-bit-induct (current bv cb)
  (if (all-zero-bitvp cb)
      t
    (highest-bit-induct
     (if (all-zero-bitvp (and-bitv cb bv)) current cb)
     bv
     (append (cdr cb) '(0))))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(prove-lemma bit-vectorp-simple-not (rewrite)
          (implies
           (not (equal (length x) (fix y)))
```

```
             (not (bit-vectorp x y))))

(prove-lemma at-most-one-bit-on-cdr (rewrite)
            (implies
             (at-most-one-bit-on x)
             (at-most-one-bit-on (cdr x))))


(prove-lemma equal-one-bit-vector (rewrite)
            (equal
             (equal x (one-bit-vector z))
             (or
              (and
               (zerop z)
               (equal x '(1)))
              (and
               (bit-vectorp x z)
               (all-zero-bitvp (all-but-last x))
               (not (equal (last x) 0))))))

(prove-lemma at-most-one-bit-is-all-zeros (rewrite)
            (implies
             (not (equal (last x) 0))
             (equal
              (at-most-one-bit-on x)
              (all-zero-bitvp (all-but-last x)))))

(defn highest-bit2-helper (current cb bv)
  (if (all-zero-bitvp cb)
      current
    (highest-bit2-helper
     (if (all-zero-bitvp (and-bitv bv cb)) current cb)
     (append (cdr cb) '(0))
     bv))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(prove-lemma equal-x-zero-bit-vector (rewrite)
            (equal
             (equal x (zero-bit-vector y))
             (and
              (all-zero-bitvp x)
              (bit-vectorp x y))))

(prove-lemma all-zero-bitvp-all-but-last-simple (rewrite)
            (implies
             (all-zero-bitvp x)
             (all-zero-bitvp (all-but-last x))))

(prove-lemma equal-trailing-zeros-acc-irrelevant (rewrite)
            (equal
             (equal (trailing-zeros-helper x acc1)
                    (trailing-zeros-helper x acc2))
             (or
              (equal (fix acc1) (fix acc2))
              (not (all-zero-bitvp x)))))

(prove-lemma lessp-trailing-zeros (rewrite)
            (implies
             (not (all-zero-bitvp z))
             (lessp (trailing-zeros-helper z acc)
                    (length z))))

(prove-lemma nthcdr-append (rewrite)
            (equal (nthcdr n (append x y))
                   (if (lessp (length x) n)
                       (nthcdr (difference n (length x)) y)
                     (append (nthcdr n x) y))))

(prove-lemma listp-zero-bit-vector (rewrite)
            (equal
             (listp (zero-bit-vector x))
             (not (zerop x))))

(prove-lemma and-bitv-append (rewrite)
            (equal
             (and-bitv (append x y) z)
             (append
              (and-bitv x (make-list-from (length x) z))
              (and-bitv y (nthcdr (length x) z))))
            ((induct (double-cdr-induct x z))))

(prove-lemma and-bitv-append2 (rewrite)
            (implies
             (equal (length (append x y)) (length z))
            (equal
             (and-bitv z (append x y))
             (append
```

```
            (and-bitv x (make-list-from (length x) z))
            (and-bitv y (nthcdr (length x) z)))))
          ((induct (double-cdr-induct x z))))

(prove-lemma all-zero-bitvp-and-bitv-append (rewrite)
          (equal
           (all-zero-bitvp (and-bitv z (append x y)))
           (and
            (all-zero-bitvp (and-bitv (make-list-from (length x) z) x))
            (all-zero-bitvp (and-bitv (nthcdr (length x) z) y))))
          ((induct (double-cdr-induct x z))))

(prove-lemma length-cdr-zero-bit-vector (rewrite)
          (equal
           (length (cdr (zero-bit-vector x)))
           (sub1 x)))

(prove-lemma length-nthcdr (rewrite)
          (equal
           (length (nthcdr n x))
           (difference (length x) n)))

(disable and-bitv-special)
(disable and-bitv-special-special)
(disable and-bitv-special-3)
(disable and-bitv-special-4)
(disable and-bitv-special-5)
(disable and-bitv-special-6)


(prove-lemma bit-vectorp-zero-bit-vector-better (rewrite)
          (equal
           (bit-vectorp (zero-bit-vector x) size)
           (equal (fix x) (fix size))))

(prove-lemma highest-bit-correctness-general nil
          (implies
           (and
            (not (zerop word-size))
            (at-most-one-bit-on cb)
            (listp ctrl-stk)
            (at-least-morep (length temp-stk)
                            3 max-temp-stk-size)
            (equal (definition 'highest-bit prog-segment)
                   (highest-bit-program))
            (bit-vectorp bv word-size)
            (bit-vectorp cb word-size)
            (bit-vectorp current word-size))
           (equal
            (p (p-state '(pc (highest-bit . 3))
                        (cons (list
                               (list
                                (cons 'bv (list 'bitv bv))
                                (cons 'cb (list 'bitv cb)))
                               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)
               (highest-bit-loop-clock cb bv))
            (p-state ret-pc
                     ctrl-stk
                     (cons
                      (list 'bitv
                            (highest-bit2-helper current cb bv))
                      temp-stk)
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)))
          ((induct (highest-bit-induct current bv cb))))

(prove-lemma make-list-from-simple (rewrite)
          (implies
           (zerop n)
           (equal (make-list-from n x) nil)))

(prove-lemma not-all-zero-bitvp-make-list-from (rewrite)
          (implies
           (and
            (all-zero-bitvp (make-list-from n1 x))
```

```
            (not (lessp n1 n2)))
           (all-zero-bitvp (make-list-from n2 x))))

(disable nthcdr-open)

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

(prove-lemma highest-bit2-helper-cons-1 (rewrite)
            (implies
             (and
              (not (all-zero-bitvp cb))
              (at-most-one-bit-on cb)
              (equal (length cb) (length bv))
              (bit-vectorp cb (length cb))
              (equal (car bv) 1))
            (equal
             (highest-bit2-helper current cb bv)
             (cons 1 (zero-bit-vector (length (cdr bv)))))))

(prove-lemma bit-vectorp-append-cdr-hack (rewrite)
            (implies
             (bit-vectorp x n)
             (equal (bit-vectorp (append (cdr x) '(0)) n)
                    (listp x))))

(defn triple-cdr-with-sub1-induct (x y z n)
  (if (zerop n) t
    (triple-cdr-with-sub1-induct (cdr x) (cdr y) (cdr z) (sub1 n))))


(prove-lemma car-append-better (rewrite)
            (equal
             (car (append x y))
             (if (listp x)
                 (car x)
               (car y))))

(prove-lemma open-highest-when-and-not-0 (rewrite)
            (implies
             (and
              (not (all-zero-bitvp (and-bitv x y)))
              (equal (length x) (length y)))
             (equal (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))))
(enable cdr-append)

(prove-lemma highest-bit2-helper-cons-0 (rewrite)
            (implies
             (and
              (not (all-zero-bitvp cb))
              (at-most-one-bit-on cb)
              (bit-vectorp cb size)
              (bit-vectorp bv size)
              (bit-vectorp current size)
              (equal (car bv) 0)
              (equal (car current) 0))
            (equal
             (highest-bit2-helper current cb bv)
             (cons 0 (highest-bit2-helper
                      (cdr current) (cdr cb) (cdr bv)))))
            ((expand (highest-bit2-helper z x v))))

(prove-lemma highest-bit2-helper-cons-0-rewrite (rewrite)
            (implies
             (and
              (not (all-zero-bitvp cb))
              (at-most-one-bit-on cb)
              (bit-vectorp cb (length cb))
              (bit-vectorp bv (length cb))
              (bit-vectorp current (length cb))
              (equal (car bv) 0)
              (equal (car current) 0))
            (equal
             (highest-bit2-helper current cb bv)
             (cons 0 (highest-bit2-helper
                      (cdr current) (cdr cb) (cdr bv)))))
```

```
          ((use (highest-bit2-helper-cons-0 (size (length cb))))))

(prove-lemma append-zeros-0 (rewrite)
          (implies
           (and
            (all-zero-bitvp x)
            (properp x))
           (equal (append x '(0)) (cons 0 x))))

(prove-lemma highest-bit2-helper-cons-helper nil
          (implies
           (and
            (bit-vectorp bv size)
            (bit-vectorp cb size)
            (at-most-one-bit-on cb)
            (at-most-one-bit-on current)
            (not (all-zero-bitvp cb))
            (or
             (all-zero-bitvp current)
             (lessp (trailing-zeros current) (trailing-zeros cb)))
            (bit-vectorp current size)
            (not (zerop size)))
            (equal
             (highest-bit2-helper current cb bv)
             (if (equal (car bv) 0)
                 (cons 0 (highest-bit2-helper (cdr current)
                                              (cdr cb) (cdr bv)))
               (cons 1 (zero-bit-vector (sub1 size))))))
          ((induct (triple-cdr-with-sub1-induct
                   bv cb current size))))

(prove-lemma highest-bit2-helper-cons-helper-rewrite (rewrite)
          (implies
           (and
            (bit-vectorp bv (length bv))
            (bit-vectorp cb (length bv))
            (at-most-one-bit-on cb)
            (at-most-one-bit-on current)
            (not (all-zero-bitvp cb))
            (or
             (all-zero-bitvp current)
             (lessp (trailing-zeros current) (trailing-zeros cb)))
            (bit-vectorp current (length bv))
            (listp bv))
            (equal
             (highest-bit2-helper current cb bv)
             (if (equal (car bv) 0)
                 (cons 0 (highest-bit2-helper (cdr current)
                                              (cdr cb) (cdr bv)))
               (cons 1 (zero-bit-vector (sub1 (length bv)))))))
          ((use (highest-bit2-helper-cons-helper (size (length bv))))
           (disable-theory t)
           (enable equal-length-0)
           (enable-theory ground-zero)))

(prove-lemma bit-vectorp-cdr-from-free (rewrite)
          (implies
           (bit-vectorp x n)
           (equal (bit-vectorp (cdr x) s)
                  (equal (add1 s) n))))

(prove-lemma all-zero-bitvp-one-bit-vector (rewrite)
          (not (all-zero-bitvp (one-bit-vector x))))

(prove-lemma trailing-zeros-one-bit-vector (rewrite)
          (equal (trailing-zeros (one-bit-vector n)) 0))

(prove-lemma cdr-zero-one-bit-vector (rewrite)
          (and
           (equal (cdr (one-bit-vector x))
                  (if (lessp x 2) nil (one-bit-vector (sub1 x))))
           (equal (cdr (zero-bit-vector x))
                  (if (zerop x) 0 (zero-bit-vector (sub1 x))))))

(prove-lemma highest-bit2-helper-highest-bit (rewrite)
          (implies
           (and
            (bit-vectorp bv word-size)
            (not (zerop word-size)))
          (equal
           (highest-bit2-helper (zero-bit-vector word-size)
                                (one-bit-vector word-size) bv)
           (highest-bit bv)))
          ((induct (bit-vectorp bv word-size))
           (disable-theory t)
           (enable-theory ground-zero naturals)
           (enable highest-bit2-helper-cons-helper-rewrite
```

```
                    bv-length-weaker LENGTH-FROM-BIT-VECTORP
                    at-most-one-bit-on-one-bit-vector
                    *1*zero-bit-vector *1*one-bit-vector
                    *1*highest-bit2-helper bitp
                    bit-vectorp cdr-zero-one-bit-vector
                    ALL-ZERO-BITVP-ZERO-BIT-VECTOR
                    bit-vectorp-simple-not length highest-bit
                    TRAILING-ZEROS-OF-ALL-ZERO-BITVP
                    bit-vectorp-zero-bit-vector-better
                    length-zero-bit-vector
                    trailing-zeros-one-bit-vector
                    all-zero-bitvp-one-bit-vector
                    ALL-ZERO-BITVP-MEANS-AT-MOST-ONE-BIT-ON
                    bit-vectorp-one-bit-vector-rewrite)))

(defn highest-bit-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'bitv)
   (equal (cddr (top (p-temp-stk p0))) nil)
   (not (zerop (p-word-size 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))
   (equal (definition 'highest-bit (p-prog-segment p0))
        (highest-bit-program))
   (equal (definition 'push-1-vector (p-prog-segment p0))
        (push-1-vector-program (p-word-size p0)))
   (bit-vectorp (cadr (top (p-temp-stk p0))) (p-word-size p0))))

(defn highest-bit-clock (bv)
  (plus 6 (highest-bit-loop-clock (one-bit-vector (length bv)) bv)))

(prove-lemma cons-0-zero-bit-vector (rewrite)
           (equal
            (cons 0 (zero-bit-vector x))
            (zero-bit-vector (add1 x))))

(disable cons-0-zero-bit-vector)

(prove-lemma correctness-of-highest-bit (rewrite)
           (implies
            (and
             (equal 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))
             (equal (p-current-instruction p0)
                   '(call highest-bit))
             (equal bc (cadr b))
             (highest-bit-input-conditionp p0))
            (equal
             (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)
               (highest-bit-clock bc))
             (p-state (add1-addr pc)
                    ctrl-stk
                    (cons (list 'bitv (highest-bit (cadr b)))
                         temp-stk)
                    prog-segment
                    data-segment
                    max-ctrl-stk-size
                    max-temp-stk-size
                    word-size
                    'run)))
           ((use (highest-bit-correctness-general
                 (ret-pc (add1-addr pc))
                 (current (zero-bit-vector word-size))
                 (cb (one-bit-vector word-size))
                 (bv (cadr b))))
            (expand (highest-bit-clock (cadr b))))
```

```
          (enable cons-0-zero-bit-vector)
          (disable highest-bit-loop-clock zero-bit-vector)))
```

```
;; match-and-xor

(defn match-and-xor-program ()
  '(match-and-xor (vecs numvecs match xor-vector) ((i (nat 0)))
                (push-constant (nat 0))
                (pop-local i)
          (dl loop ()
                (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 ()
                (push-local vecs))
                (fetch)
                (push-local xor-vector)
                (xor-bitv)
                (push-local vecs)
                (deposit)
           (dl done ()
                (ret))))


(defn 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))

(defn match-and-xor (bvs match xor-vector)
  (if (listp bvs)
      (if (all-zero-bitvp (and-bitv (car bvs) match))
          (cons (car bvs) (match-and-xor (cdr bvs) match xor-vector))
        (cons (xor-bitv (car bvs) xor-vector) (cdr bvs)))
    nil))

(defn match-and-xor-loop-clock (bvs match)
  (if (listp bvs)
      (if (all-zero-bitvp (and-bitv (car bvs) match))
          (if (listp (cdr bvs))
            (plus 16 (match-and-xor-loop-clock (cdr bvs) match))
            12)
        12)
    0))

(defn tag-array (tag array)
  (if (listp array)
      (cons (tag tag (car array))
            (tag-array tag (cdr array)))
    nil))
```

```
(defn match-and-xor-general-induct (numvecs i)
  (if (lessp i numvecs)
      (match-and-xor-general-induct numvecs (add1 i))
    t)
  ((lessp (difference numvecs i))))

(prove-lemma tag-array-untag-array (rewrite)
  (implies
   (bit-vectors-piton x (length (untag (car x))))
   (equal (tag-array 'bitv (untag-array x)) x)))

(prove-lemma bit-vectors-piton-free-means (rewrite)
             (implies
              (bit-vectors-piton x size)
              (and
               (bit-vectors-piton x (length (cadr (car x))))
               (equal
                (bit-vectors-piton (cdr x) (length (cadr (cadr x))))
                (listp x)))))

(prove-lemma listp-cdr-nthcdr (rewrite)
             (equal
              (listp (cdr (nthcdr i x)))
              (lessp i (length (cdr x)))))

(prove-lemma nthcdr-untag-array (rewrite)
              (equal
               (nthcdr i (untag-array x))
               (if (lessp i (length x))
                   (untag-array (nthcdr i x))
                 (if (equal (fix i) (length x)) nil 0))))

(prove-lemma put-length-cdr (rewrite)
             (implies
              (properp x)
             (equal
              (put val (length (cdr x)) x)
              (append (all-but-last x) (list val)))))

(prove-lemma nthcdr-length-cdr (rewrite)
             (implies
              (properp x)
             (equal
              (nthcdr (length (cdr x)) x)
              (if (listp x) (list (last x)) nil))))

(prove-lemma get-length-cdr (rewrite)
             (equal
              (get (length (cdr x)) x)
              (if (listp x) (last x) 0)))

(prove-lemma append-all-but-last-last (rewrite)
             (implies
              (properp x)
              (equal
               (append (all-but-last x) (list (last x)))
               (if (nlistp x) (list (last x)) x))))

(prove-lemma listp-cdr-untag-array (rewrite)
             (equal
              (listp (cdr (untag-array x)))
              (listp (cdr x))))

(prove-lemma bit-vectorp-last (rewrite)
             (implies
              (bit-vectors-piton bvs s)
              (equal (bit-vectorp (cadr (last bvs)) s) (listp bvs))))

(prove-lemma bit-vectors-piton-means-properp (rewrite)
             (implies
              (bit-vectors-piton x s)
              (properp x)))

(prove-lemma bit-vectors-piton-means-last (rewrite)
             (implies
              (and
               (bit-vectors-piton state size)
               (listp state))
              (and
               (equal (car (last state)) 'bitv)
               (listp (last state))
               (bit-vectorp (cadr (last state)) size)
               (equal (cddr (last state)) nil)
               (equal (length (cadr (last state))) (fix size))))
             ((disable bit-vectorp-last)))
```

```
(prove-lemma list-bitv-cadr-bitvp (rewrite)
           (implies
            (and
             (equal (car x) 'bitv)
             (equal (cddr x) nil))
            (equal (list 'bitv (cadr x)) x)))

(prove-lemma equal-x-put-assoc-x (rewrite)
           (equal
            (equal x (put-assoc val name  x))
            (or
             (not (definedp name x))
             (equal (assoc name x) (cons name val)))))

(prove-lemma cons-n-assoc-n (rewrite)
           (implies
            (listp (assoc n d))
            (equal (cons n (cdr (assoc n d)))
                  (if (definedp n d)
                      (assoc n d)
                    (cons n 0)))))

(prove-lemma cons-n-cadr-list-assoc-n (rewrite)
           (implies
             (equal (cddr (assoc n d)) nil)
            (equal (list n (cadr (assoc n d)))
                  (if (definedp n d)
                      (assoc n d)
                    (cons n 0)))))

(prove-lemma cons-n-assoc-n-hack (rewrite)
        (implies
        (listp (cdr (assoc n d)))
        (equal (cons n (cdr (assoc n d)))
            (if (definedp n d)
                (assoc n d)
                 (cons n 0)))))

(prove-lemma car-untag-array (rewrite)
           (equal
            (car (untag-array x))
            (untag (car x))))

(prove-lemma car-nthcdr (rewrite)
           (equal
            (car (nthcdr i x))
            (get i x)))


(prove-lemma tag-array-cdr-untag-array-hack (rewrite)
           (implies
            (bit-vectors-piton x (length (untag (car x))))
           (equal
            (tag-array 'bitv (cdr (untag-array x)))
            (if (listp x) (cdr x) nil))))

(prove-lemma bit-vectors-piton-means-car (rewrite)
           (implies
            (and
             (bit-vectors-piton state size)
             (listp state))
            (and
             (equal (caar state) 'bitv)
             (listp (car state))
             (bit-vectorp (cadr (car state)) size)
             (equal (cddr (car state)) nil)
             (equal (length (cadr (car state))) (fix size)))))

(prove-lemma equal-put-assoc (rewrite)
           (equal
            (equal (put-assoc v1 n s) (put-assoc v2 n s))
            (or
             (not (definedp n s))
             (equal v1 v2))))

(prove-lemma get-nlistp-better (rewrite)
           (implies
            (not (lessp i (length x)))
            (equal (get i x) 0)))

(prove-lemma equal-append-zero-bit-vector-zero-bit-vector (rewrite)
           (equal
            (equal (append (zero-bit-vector n1) x)
                  (append (zero-bit-vector n2) y))
            (if (lessp n1 n2)
                (equal
                 x (append (zero-bit-vector (difference n2 n1)) y))
```

```
              (equal
               y (append (zero-bit-vector (difference n1 n2)) x)))))

(prove-lemma append-make-list-from-cons-get-hack (rewrite)
            (equal
             (append (make-list-from i x) (cons (get i x) y))
             (append (make-list-from (add1 i) x) y)))

(prove-lemma cdr-untag-array-nthcdr (rewrite)
            (implies
             (lessp i (length x))
             (equal
              (cdr (untag-array (nthcdr i x)))
              (untag-array (nthcdr i (cdr x))))))

(prove-lemma equal-nil-nthcdr-length (rewrite)
            (equal
             (equal nil (nthcdr (length x) x))
             (properp x)))

(prove-lemma lessp-0-length-better (rewrite)
            (implies
             (zerop x)
             (equal (lessp x (length y)) (listp y))))

(prove-lemma bit-vectors-piton-means-get-cdr (rewrite)
            (implies
             (bit-vectors-piton state size)
             (and
              (equal (car (get i (cdr state)))
                     (if (lessp i (length (cdr state))) 'bitv 0))
              (equal (listp (get i (cdr state)))
                     (lessp i (length (cdr state))))
              (equal (bit-vectorp (cadr (get i (cdr state))) size)
                     (lessp i (length (cdr state))))
              (equal (cddr (get i (cdr state)))
                     (if (lessp i (length (cdr state))) nil 0))
              (equal (length (cadr (get i (cdr state))))
                     (if (lessp i (length (cdr state))) (fix size) 0))))
            ((induct (get i state))))

(prove-lemma bit-vectors-piton-nthcdr (rewrite)
            (implies
             (bit-vectors-piton x s1)
             (and
              (equal (bit-vectors-piton (nthcdr i x) s)
                     (or
                      (and
                       (lessp i (length x))
                       (equal (fix s1) (fix s)))
                      (equal (fix i) (length x))))
              (equal (bit-vectors-piton (nthcdr i (cdr x)) s)
                     (and
                      (lessp i (length x))
                      (or (equal (fix s) (fix s1))
                          (equal (fix i) (length (cdr x)))))))))

(prove-lemma tag-array-untag-array-nthcdr-cddr-hack (rewrite)
            (implies
             (and
              (lessp i (length x))
              (bit-vectors-piton x s))
             (equal
              (tag-array 'bitv (untag-array (nthcdr i (cdr x))))
              (nthcdr i (cdr x))))
            ((use (tag-array-untag-array (x (nthcdr i (cdr x)))))))

(prove-lemma append-make-list-from-put-hack (rewrite)
            (implies
             (lessp i (length x))
            (equal
             (append
              (make-list-from i x)
              (cons v (nthcdr i (cdr x))))
             (put v i x))))

(enable lessp-sub1-x-x)
(enable lessp-x-x)

(prove-lemma correctness-of-match-and-xor-general nil
            (implies
             (and
                 (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)
```

```
              (equal (definition 'match-and-xor prog-segment) (match-and-xor-program))
              (bit-vectorp match word-size)
              (bit-vectorp xor-vector word-size)
              (equal numvecs (length (array vecs data-segment)))
              (bit-vectors-piton (array vecs data-segment) word-size)
              (definedp vecs data-segment)
              (numberp i)
              (lessp i numvecs)
              (lessp (length (array vecs data-segment)) (exp 2 word-size)))
            (equal
             (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)))
           ((induct (match-and-xor-general-induct numvecs i))
            (disable bit-vectors-piton)
            (expand
             (match-and-xor-loop-clock
              (untag-array (nthcdr i (cdr (assoc vecs data-segment))))
              match)
             (match-and-xor-loop-clock
              (untag-array (nthcdr (length (cddr (assoc vecs data-segment)))
                                   (cdr (assoc vecs data-segment))))
              match))))

(defn match-and-xor-clock (bvs match)
  (plus 3 (match-and-xor-loop-clock bvs match)))


(defn match-and-xor-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'bitv)
   (equal (cddr (top (p-temp-stk p0))) nil)
   (equal (car (top (cdr (p-temp-stk p0)))) 'bitv)
   (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
   (equal (car (top (cddr (p-temp-stk p0)))) 'nat)
   (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
   (equal (car (top (cdddr (p-temp-stk p0)))) 'addr)
   (equal (cdr (cadr (top (cdddr (p-temp-stk p0))))) 0)
   (listp (cadr (top (cdddr (p-temp-stk p0)))))
   (equal (cddr (top (cdddr (p-temp-stk p0)))) nil)
   (not (zerop (cadr (top (cddr (p-temp-stk p0))))))
   (listp (p-ctrl-stk p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   7 (p-max-ctrl-stk-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   0 (p-max-temp-stk-size p0))
   (equal (definition 'match-and-xor (p-prog-segment p0))
          (match-and-xor-program))
   (bit-vectorp (cadr (top (cdr (p-temp-stk p0)))) (p-word-size p0))
   (bit-vectorp (cadr (top (p-temp-stk p0))) (p-word-size p0))
   (equal (cadr (top (cddr (p-temp-stk p0))))
          (length (array (car (cadr (top (cdddr (p-temp-stk p0)))))
                         (p-data-segment p0))))
   (definedp (car (cadr (top (cdddr (p-temp-stk p0)))))
```

```
    (p-data-segment p0))
  (lessp (length (array (car (cadr (top (cdddr (p-temp-stk p0)))))
                       (p-data-segment p0)))
         (exp 2 (p-word-size p0)))
  (bit-vectors-piton
   (array (car (cadr (top (cdddr (p-temp-stk p0)))))
          (p-data-segment p0))
   (p-word-size p0))))

(prove-lemma correctness-of-match-and-xor (rewrite)
            (implies
             (and
              (equal 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))
              (equal (p-current-instruction p0) '(call match-and-xor))
              (match-and-xor-input-conditionp p0)
              (equal match (cadr m))
              (equal vecs-to-match
                     (untag-array (array (caadr v) data-segment))))
             (equal
              (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)))
            ((disable match-and-xor-loop-clock)
             (use (correctness-of-match-and-xor-general
                   (ret-pc (add1-addr pc)) (i 0)
                   (xor-vector (cadr x)) (match (cadr m))
                   (numvecs (cadr n)) (vecs (caadr v))))))

;;; nat-to-bv-list

(defn nat-to-bv-list-program ()
  '(nat-to-bv-list (nat-list bv-list length) ((i (nat 0)))
             (dl loop ()
                   (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 () (ret))))
```

```
(defn 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))

(defn nat-to-bv-list (nat-list size)
  (if (listp nat-list)
      (cons
       (nat-to-bv (car nat-list) size)
       (nat-to-bv-list (cdr nat-list) size))
    nil))

(defn nat-to-bv-list-loop-clock (i nats)
       (if (lessp i (length nats))
           (clock-plus '2
               (clock-plus (nat-to-bv-clock (untag (get i nats)))
                   (if (lessp i (length (cdr nats)))
                       (clock-plus '17
                           (nat-to-bv-list-loop-clock (add1 i) nats))
                       '9)))
           '0)
       ((lessp (difference (length nats) i))))

(defn nat-to-bv-list-loop-induct (i nats bvs-name word-size data-segment)
  (if (and (lessp i (length nats))
           (lessp i (length (cdr nats))))
      (nat-to-bv-list-loop-induct
       (add1 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 (add1 i) (cdr (assoc bvs-name data-segment)))))
                  bvs-name data-segment))
    t)
  ((lessp (difference (length nats) i))))


(prove-lemma nat-list-piton-means-car (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (listp state))
              (and
               (equal (caar state) 'nat)
               (listp (car state))
               (numberp (cadr (car state)))
               (lessp (cadr (car state)) (exp 2 size))
               (equal (lessp (cadr (car state)) (exp 2 size)) t)
               (equal (cddr (car state)) nil))))

(defn array-pitonp (array length)
  (if (listp array)
      (and
       (not (zerop length))
       (array-pitonp (cdr array) (sub1 length)))
    (and
     (equal array nil)
     (zerop length))))


(prove-lemma nat-list-piton-means-last (rewrite)
             (implies
              (nat-list-piton state size)
              (and
               (equal (car (last state)) (if (listp state) 'nat 0))
```

```
            (equal (listp (last state)) (listp state))
            (numberp (cadr (last state)))
            (equal (lessp (cadr (last state)) (exp 2 size)) t)
            (equal (cddr (last state)) (if (listp state) nil 0)))))

(prove-lemma nat-list-piton-means-last-cdr (rewrite)
          (implies
           (and
            (nat-list-piton state size)
            (listp state))
           (and
            (equal (car (last (cdr state)))
                   (if (listp (cdr state)) 'nat 0))
            (equal (listp (last (cdr state))) (listp (cdr state)))
            (numberp (cadr (last (cdr state))))
            (equal (lessp (cadr (last (cdr state))) (exp 2 size)) t)
            (equal (cddr (last (cdr state)))
                   (if (listp (cdr state)) nil 0)))))

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

(prove-lemma assoc-put-assoc-better (rewrite)
          (equal (assoc n1 (put-assoc v n2 a))
                 (if (definedp n1 a)
                     (if (equal n1 n2)
                         (cons n1 v)
                       (assoc n1 a))
                   f)))

(disable nat-to-bv-list-loop-clock)

(prove-lemma get-add1 (rewrite)
          (implies
           (lessp n 4)
           (equal (get (add1 n) x) (get n (cdr x)))))

(prove-lemma get-0-better (rewrite)
          (implies
           (zerop n)
           (equal (get n x) (car x))))

(prove-lemma length-cdr-tag-array (rewrite)
          (equal
           (length (cdr (tag-array l x)))
           (length (cdr x))))

(prove-lemma length-nat-to-bv-list (rewrite)
          (equal
           (length (nat-to-bv-list x s))
           (length x)))

(prove-lemma length-cdr-nat-to-bv-list (rewrite)
          (equal
           (length (cdr (nat-to-bv-list x s)))
           (length (cdr x))))

(prove-lemma length-tag-array (rewrite)
          (equal
           (length (tag-array l x))
           (length x)))

(prove-lemma listp-tag-array (rewrite)
          (equal
           (listp (tag-array l x))
           (listp x)))

(prove-lemma listp-nat-to-bv-list (rewrite)
          (equal
           (listp (nat-to-bv-list x s))
           (listp x)))

(prove-lemma array-pitonp-tag-array (rewrite)
          (equal
           (array-pitonp (tag-array l x) length)
           (equal (length x) (fix length)))
          ((induct (array-pitonp x length))))

(prove-lemma array-pitonp-append (rewrite)
          (equal
           (array-pitonp (append x y) size)
           (and
            (not (lessp size (length x)))
            (array-pitonp y (difference size (length x))))))
```

```
          ((induct (get size x))))

(prove-lemma nat-list-piton-means-cadr (rewrite)
          (implies
           (and
            (nat-list-piton state size)
            (listp (cdr state)))
           (and
            (equal (caadr state) 'nat)
            (listp (cadr state))
            (numberp (cadr (cadr state)))
            (lessp (cadr (cadr state)) (exp 2 size))
            (equal (lessp (cadr (cadr state)) (exp 2 size)) t)
            (equal (cddr (cadr state)) nil))))

(prove-lemma array-pitonp-add1 (rewrite)
          (equal
           (array-pitonp x (add1 n))
           (and
            (listp x)
            (array-pitonp (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)))
(enable put-assoc-put-assoc)

(prove-lemma length-cdr-nlistp (rewrite)
          (implies
           (not (listp (cdr x)))
           (equal (length x) (if (listp x) 1 0))))

(prove-lemma equal-nil-cdr-tag-array-hack (rewrite)
          (equal
           (equal nil (cdr (tag-array l x)))
           (equal (length x) 1)))

(prove-lemma length-from-array-pitonp (rewrite)
          (implies
           (array-pitonp x s)
           (equal (length x) (fix s))))

(disable nat-to-bv-clock)

(disable nat-to-bv-list-loop-clock)

(prove-lemma nat-list-piton-means-get (rewrite)
          (implies
           (nat-list-piton state size)
           (and
            (equal (car (get p state))
                   (if (lessp p (length state)) 'nat 0))
            (equal (listp (get p state)) (lessp p (length state)))
            (numberp (cadr (get p state)))
            (equal (lessp (cadr (get p state)) (exp 2 size)) t)
            (equal (cddr (get p state))
                   (if (lessp p (length state)) nil 0)))))

(prove-lemma nat-list-piton-means-get-cdr (rewrite)
          (implies
           (and
            (nat-list-piton state size)
            (listp state))
           (and
            (equal (car (get p (cdr state)) )
                   (if (lessp p (length (cdr state))) 'nat 0))
            (equal (listp (get p (cdr state))) (lessp p (length (cdr state))))
            (numberp (cadr (get p (cdr state))))
            (equal (lessp (cadr (get p (cdr state))) (exp 2 size)) t)
            (equal (cddr (get p (cdr state)))
                   (if (lessp p (length (cdr state))) nil 0)))))

(disable nat-list-piton-means)

(prove-lemma length-put-better (rewrite)
          (equal (length (put v n a))
                 (if (lessp n (length a)) (length a) (add1 n))))

(prove-lemma cons-car-x-put-append-make-list-from-hack (rewrite)
          (equal
           (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))))

(prove-lemma array-pitonp-put (rewrite)
          (implies
```

```
            (and
             (array-pitonp a l)
             (equal (fix l) (fix length)))
            (equal
             (array-pitonp (put v n a) length)
             (lessp n (length a)))))


(prove-lemma array-pitonp-means-properp (rewrite)
           (and
            (implies (array-pitonp x s) (properp x))
            (implies (array-pitonp (cdr x) s) (properp x))))

(prove-lemma put-length-cdr-general (rewrite)
           (implies
            (and
             (properp x)
             (equal (length x) (length y)))
           (equal
            (put val (length (cdr y)) x)
            (append (all-but-last x) (list val)))))

(prove-lemma nthcdr-x-cdr-put-x (rewrite)
           (implies
            (properp a)
           (equal
            (nthcdr x (cdr (put val x a)))
            (if (lessp x (length a))
                (nthcdr x (cdr a))
              nil))))

(prove-lemma equal-append-a-append-a (rewrite)
           (equal
            (equal (append a b) (append a c))
            (equal 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.


(prove-lemma correctness-of-nat-to-bv-list-general nil
           (implies
            (and
                (listp ctrl-stk)
                (at-least-morep (p-ctrl-stk-size ctrl-stk)
                                13 max-ctrl-stk-size)
                (at-least-morep (length temp-stk)
                                3 max-temp-stk-size)
                (equal (definition 'nat-to-bv-list prog-segment)
                       (nat-to-bv-list-program))
                (equal (definition 'nat-to-bv prog-segment)
                       (nat-to-bv-program))
                (equal (definition 'push-1-vector prog-segment)
                       (push-1-vector-program word-size))
                (equal length (length (array nats data-segment)))
                (array-pitonp (array bvs data-segment) length)
                (nat-list-piton (array nats data-segment) word-size)
                (definedp nats data-segment)
                (definedp bvs data-segment)
                (not (zerop word-size))
                (numberp i)
                (lessp i length)
                (lessp length (exp 2 word-size))
                (not (equal nats bvs))
                (equal natlist (array nats data-segment)))
            (equal
             (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)))
       ((induct (nat-to-bv-list-loop-induct
               i natlist bvs word-size data-segment))
        (disable nat-list-piton)
        (expand (ARRAY-PITONP (CDR (ASSOC BVS DATA-SEGMENT)) 1)
              (UNTAG-ARRAY (CDR (ASSOC NATS DATA-SEGMENT)))
              (UNTAG-ARRAY (CDR (ASSOC BVS DATA-SEGMENT)))
              (NAT-TO-BV-LIST-LOOP-CLOCK 0
               (CDR (ASSOC NATS DATA-SEGMENT)))
              (nat-to-bv-list-loop-clock i natlist)
              (NAT-TO-BV-LIST-LOOP-CLOCK i
               (CDR (ASSOC NATS DATA-SEGMENT))))))

(defn nat-to-bv-list-clock (natlist)
  (clock-plus 1 (nat-to-bv-list-loop-clock 0 natlist)))

(defn nat-to-bv-list-input-conditionp (p0)
        (and
            (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))
            (equal (definition 'nat-to-bv-list (p-prog-segment p0))
                  (nat-to-bv-list-program))
            (equal (definition 'nat-to-bv (p-prog-segment p0))
                  (nat-to-bv-program))
            (equal (definition 'push-1-vector (p-prog-segment p0))
                  (push-1-vector-program (p-word-size p0)))
              (equal (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))
            (equal (p-current-instruction p0)
                  '(call nat-to-bv-list))
            (not (zerop (p-word-size p0)))
            (not (equal (caadr (top (cddr (p-temp-stk p0)))) (caadr (top (cdr (p-temp-stk p0))))))
            (equal (car (top (p-temp-stk p0))) 'nat)
            (not (zerop (cadr (top (p-temp-stk p0)))))
            (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
            (equal (cddr (top (p-temp-stk p0))) nil)
            (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
            (listp (cadr (top (cdr (p-temp-stk p0)))))
            (equal (cdadr (top (cdr (p-temp-stk p0)))) 0)
            (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
            (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
            (listp (cadr (top (cddr (p-temp-stk p0)))))
            (equal (cdadr (top (cddr (p-temp-stk p0)))) 0)
            (equal (cddr (top (cddr (p-temp-stk p0)))) nil)))


(prove-lemma correctness-of-nat-to-bv-list (rewrite)
       (implies
        (and
         (equal 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))
```

```
         (nat-to-bv-list-input-conditionp p0)
         (equal natlist (array (caadr n) data-segment)))
        (equal
         (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)))
        ((use (correctness-of-nat-to-bv-list-general
               (i 0)
               (ret-pc (LIST (CAR PC) (CONS (CAADR PC) (ADD1 (CDADR PC)))))
               (nats (caadr n))
               (bvs (caadr b))
               (length (cadr l))))))

(disable nat-to-bv-list-program)
(disable match-and-xor-program)
(disable highest-bit-program)
(disable number-with-at-least-program)
(disable bv-to-nat-program)
(disable nat-to-bv-program)
(disable push-1-vector-program)
(disable xor-bvs-program)



;; bv-to-nat-list

(defn bv-to-nat-list-program ()
  '(bv-to-nat-list (bv-list nat-list length) ((i (nat 0)))
        (dl loop ()
             (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 () (ret))))

(defn 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))


(defn bv-to-nat-list-input-conditionp (p0)
          (and
             (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))
             (equal (definition 'bv-to-nat-list (p-prog-segment p0))
                   (bv-to-nat-list-program))
             (equal (definition 'bv-to-nat (p-prog-segment p0))
                   (bv-to-nat-program))
             (equal (definition 'push-1-vector (p-prog-segment p0))
                   (push-1-vector-program (p-word-size p0)))
               (equal (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))
             (equal (p-current-instruction p0)
                   '(call bv-to-nat-list))
             (not (zerop (p-word-size p0)))
             (not (equal (caadr (top (cddr (p-temp-stk p0)))) (caadr (top (cdr (p-temp-stk p0))))))
             (equal (car (top (p-temp-stk p0))) 'nat)
             (not (zerop (cadr (top (p-temp-stk p0)))))
             (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
             (equal (cddr (top (p-temp-stk p0))) nil)
             (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
             (listp (cadr (top (cdr (p-temp-stk p0)))))
             (equal (cdadr (top (cdr (p-temp-stk p0)))) 0)
             (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
             (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
             (listp (cadr (top (cddr (p-temp-stk p0)))))
             (equal (cdadr (top (cddr (p-temp-stk p0)))) 0)
             (equal (cddr (top (cddr (p-temp-stk p0)))) nil)))

(defn bv-to-nat-list-loop-clock (wordsize i bvs)
      (if (lessp i (length bvs))
         (clock-plus '2
            (clock-plus (bv-to-nat-clock wordsize (untag (get i bvs)))
               (if (lessp i (length (cdr bvs)))
                  (clock-plus '17
                     (bv-to-nat-list-loop-clock
                      wordsize (add1 i) bvs))
                  '9)))
         '0)
      ((lessp (difference (length bvs) i))))

(defn bv-to-nat-list (bv-list)
  (if (listp bv-list)
     (cons
      (bv-to-nat (car bv-list))
      (bv-to-nat-list (cdr bv-list)))
    nil))

(defn bv-to-nat-list-loop-induct (i bvs nats-name data-segment)
  (if (and (lessp i (length bvs))
         (lessp i (length (cdr bvs))))
     (bv-to-nat-list-loop-induct
      (add1 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 (add1 i) (cdr (assoc nats-name data-segment)))))
              nats-name data-segment))
    t)
  ((lessp (difference (length bvs) i))))

(disable bv-to-nat-clock)
```

```
(prove-lemma correctness-of-bv-to-nat-list-general nil
           (implies
            (and
               (listp ctrl-stk)
               (at-least-morep (p-ctrl-stk-size ctrl-stk)
                              13 max-ctrl-stk-size)
               (at-least-morep (length temp-stk)
                              3 max-temp-stk-size)
               (equal (definition 'bv-to-nat-list prog-segment)
                     (bv-to-nat-list-program))
               (equal (definition 'bv-to-nat prog-segment)
                     (bv-to-nat-program))
               (equal (definition 'push-1-vector prog-segment)
                     (push-1-vector-program word-size))
               (equal length (length (array bvs data-segment)))
               (bit-vectors-piton (array bvs data-segment)
                                 word-size)
               (array-pitonp (array nats data-segment) length)
               (definedp nats data-segment)
               (definedp bvs data-segment)
               (not (zerop word-size))
               (numberp i)
               (lessp i length)
               (lessp length (exp 2 word-size))
               (not (equal nats bvs))
               (equal bvlist (array bvs data-segment)))
            (equal
             (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)))
           ((induct (bv-to-nat-list-loop-induct
                   i bvlist nats data-segment))
            (disable bit-vectors-piton)
            (expand (ARRAY-PITONP (CDR (ASSOC BVS DATA-SEGMENT)) 1)
                   (UNTAG-ARRAY (CDR (ASSOC NATS DATA-SEGMENT)))
                   (UNTAG-ARRAY (CDR (ASSOC BVS DATA-SEGMENT)))
                   (BV-TO-NAT-LIST-LOOP-CLOCK wordsize 0
                    (CDR (ASSOC NATS DATA-SEGMENT)))
                   (bv-to-nat-list-loop-clock wordsize i bvlist)
                   (BV-TO-NAT-LIST-LOOP-CLOCK wordsize i
                    (CDR (ASSOC NATS DATA-SEGMENT))))))

(defn bv-to-nat-list-clock (word-size natlist)
  (clock-plus 1 (bv-to-nat-list-loop-clock word-size 0 natlist)))

(disable bv-to-nat-list-loop-clock)

(disable bv-to-nat-list-program)

(prove-lemma correctness-of-bv-to-nat-list (rewrite)
           (implies
            (and
```

```
          (equal p0
                 (p-state pc
                    ctrl-stk
                    (cons l (cons n (cons b temp-stk)))
                    prog-segment
                    data-segment
                    max-ctrl-stk-size
                    max-temp-stk-size
                    word-size
                    'run))
          (bv-to-nat-list-input-conditionp p0)
          (equal bvlist (array (caadr b) data-segment)))
         (equal
          (p
           (p-state pc
                    ctrl-stk
                    (cons l (cons n (cons b temp-stk)))
                    prog-segment
                    data-segment
                    max-ctrl-stk-size
                    max-temp-stk-size
                    word-size
                    'run)
           (bv-to-nat-list-clock word-size bvlist))
          (p-state
           (add1-addr pc)
           ctrl-stk
           temp-stk
           prog-segment
           (put-assoc
            (tag-array 'nat
                       (bv-to-nat-list
                        (untag-array
                         (array (caadr b) data-segment))))
            (caadr n) data-segment)
           max-ctrl-stk-size
           max-temp-stk-size
           word-size
           'run)))
         ((use (correctness-of-bv-to-nat-list-general
                (i 0)
                (ret-pc (LIST (CAR PC) (CONS (CAADR PC) (ADD1 (CDADR PC)))))
                (nats (caadr n))
                (bvs (caadr b))
                (length (cadr l))))))

;; max-nat


(defn max-nat-program ()
  '(max-nat (nat-list length)  ((i (nat 0)) (j (nat 0)))
                (push-constant (nat 0))
         (dl loop ()
                (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 ()
                (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 () (ret))))


(defn 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))

(defn max-list-helper (val x)
  (if (listp x)
      (if (lessp val (car x))
          (max-list-helper (car x) (cdr x))
        (max-list-helper val (cdr x)))
    (fix val)))

(defn max-list (x)
  (if (listp x)
      (if (lessp (max-list (cdr x)) (car x))
          (car x)
        (max-list (cdr x)))
    0))

(prove-lemma max-list-helper-max-list (rewrite)
             (equal (max-list-helper val x)
                    (if (lessp (max-list x) val)
                        val
                      (max-list x))))

(prove-lemma max-list-helper-max-list-0 (rewrite)
             (equal (max-list-helper 0 x)
                    (max-list x)))

(disable max-list-helper-max-list)

(defn max-nat-loop-clock (val i nats)
  (if (lessp i (length nats))
       (if (lessp val (untag (get i nats)))
           (if (lessp i (length (cdr nats)))
               (clock-plus 20
                (max-nat-loop-clock
                 (untag (get i nats)) (add1 i) nats))
             16)
         (if (lessp i (length (cdr nats)))
             (clock-plus 18 (max-nat-loop-clock val (add1 i) nats))
           14))
    0)
  ((lessp (difference (length nats) i))))

(defn max-nat-loop-induct (i j x natlist length)
  (if (lessp i length)
      (max-nat-loop-induct
       (add1 i) (get i natlist)
       (if (lessp (untag (get i natlist)) (untag x))
           (list 'nat (untag x)) (list 'nat (untag (get i natlist))))
       natlist length)
    t)
  ((lessp (difference length i))))


(prove-lemma list-nat-from-assoc-nat-list-piton-hack (rewrite)
             (implies
              (nat-list-piton nl s)
              (and
               (implies
                (lessp z (length (cdr nl)))
                (equal (list 'nat (cadr (get z (cdr nl))))
                       (get z (cdr nl))))
               (implies
                (listp nl)
                (and
                 (equal
                  (list 'nat (cadr (last nl)))
                  (last nl))
                (equal
                 (list 'nat (cadr (car nl)))
                 (car nl))))))
             ((induct (get z nl))))

(prove-lemma correctness-of-max-nat-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (p-ctrl-stk-size ctrl-stk)
                                  4 max-ctrl-stk-size)
                  (at-least-morep (length temp-stk)
                                  4 max-temp-stk-size)
                  (equal (definition 'max-nat prog-segment)
                         (max-nat-program))
```

```
              (equal length (length (array nats data-segment)))
              (nat-list-piton (array nats data-segment) word-size)
              (definedp nats data-segment)
              (not (zerop word-size))
              (numberp i)
              (lessp i length)
              (numberp (untag x))
              (lessp (untag x) (exp 2 word-size))
              (equal (car x) 'nat)
              (equal (cddr x) nil)
              (lessp length (exp 2 word-size))
              (equal natlist (array nats data-segment)))
            (equal
             (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)))
            ((expand (MAX-NAT-LOOP-CLOCK
                      (cadr X) I (CDR (ASSOC NATS DATA-SEGMENT))))
             (induct (max-nat-loop-induct i j x natlist length))))

(defn max-nat-clock (natlist)
  (clock-plus 2 (max-nat-loop-clock 0 0 natlist)))

(defn max-nat-input-conditionp (p0)
  (and
   (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))
   (equal (definition 'max-nat (p-prog-segment p0))
          (max-nat-program))
   (equal (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))
  (not (zerop (p-word-size p0)))
  (not (zerop (untag (top (p-temp-stk p0)))))
  (equal (car (top (p-temp-stk p0))) 'nat)
  (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
  (listp (untag (top (cdr (p-temp-stk p0)))))
  (lessp (untag (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
  (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
  (equal (cddr (top (p-temp-stk p0))) nil)
  (equal (cdr (untag (top (cdr (p-temp-stk p0))))) 0)))

(prove-lemma correctness-of-max-nat (rewrite)
             (implies
              (and
               (equal 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))
          (equal (p-current-instruction p0)
                 '(call max-nat))
          (max-nat-input-conditionp p0)
          (equal natlist (array (car (untag s)) data-segment)))
         (equal
          (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)))
         ((use (correctness-of-max-nat-general
                (i 0) (j '(nat 0)) (x (list 'nat 0))
                (nats (caadr s))
               (ret-pc (add1-addr pc))
               (length (cadr n))))))

(disable max-nat-program)

;; replace-value

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

(defn replace-value-program ()
  '(replace-value (list oldval newval) ()
         (dl loop ()
             (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 ()
             (push-local newval))
             (push-local list)
             (deposit)
             (ret)))

(defn 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))

(defn replace-value-loop-clock (i list oldvalue)
  (if (lessp i (length list))
      (if (equal (get i list) oldvalue)
```

```
        9
       (clock-plus
        10 (replace-value-loop-clock (add1 i) list oldvalue)))
   0)
  ((lessp (difference (length list) i))))

(defn replace-value-loop-induct (i list list-addr)
  (if (lessp i (length list))
      (replace-value-loop-induct (add1 i) list (add1-addr list-addr))
    t)
  ((lessp (difference (length list) i))))

(defn replace-value (list oldvalue newvalue)
  (if (listp list)
      (if (equal (car list) oldvalue)
          (cons newvalue (cdr list))
        (cons (car list) (replace-value (cdr list) oldvalue newvalue)))
    nil))

(prove-lemma member-nthcdr-means (rewrite)
             (implies
              (member x (nthcdr i y))
              (member x y)))

(prove-lemma member-of-natlist-means (rewrite)
             (implies
              (and
               (nat-list-piton y s)
               (member x y))
              (and
               (listp x)
               (equal (car x) 'nat)
               (numberp (cadr x))
               (lessp (cadr x) (exp 2 s))
               (equal (cddr x) nil))))

(prove-lemma replace-value-nthcdr-open (rewrite)
             (implies
              (and
               (not (equal (get x a) old))
               (lessp x (length a)))
              (equal
               (replace-value (nthcdr x a) old new)
               (cons (get x a)
                     (replace-value (nthcdr x (cdr a)) old new)))))

(prove-lemma list-nat-cadr-get-hack (rewrite)
             (implies
              (and
               (nat-list-piton y s)
               (lessp x (length y)))
              (equal
               (list 'nat (cadr (get x y)))
               (get x y))))

(prove-lemma member-nthcdr-simplify (rewrite)
             (implies
              (and
               (lessp i (length x))
               (not (member a (nthcdr i (cdr x)))))
              (equal
               (member a (nthcdr i x))
               (equal a (get i x)))))

(PROVE-LEMMA APPEND-MAKE-LIST-FROM-CONS-CDR
             (REWRITE)
             (IMPLIES (LESSP I (LENGTH Y))
                      (EQUAL (APPEND (MAKE-LIST-FROM I Y)
                                     (CONS V (CDR (NTHCDR I Y))))
                             (PUT V I Y))))

(prove-lemma correctness-of-replace-value-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (length temp-stk)
                                  2 max-temp-stk-size)
                  (equal (definition 'replace-value prog-segment)
                         (replace-value-program))
                  (nat-list-piton (array list data-segment) word-size)
                  (definedp list data-segment)
                  (not (zerop word-size))
                  (numberp i)
                  (equal vallist (array list data-segment))
                  (equal (car list-addr) 'addr)
                  (equal (cddr list-addr) nil)
                  (listp (untag list-addr))
```

```
                (equal (car (untag list-addr)) list)
                (equal (cdr (untag list-addr)) i)
                (equal (car newvalue) 'nat)
                (equal (cddr newvalue) nil)
                (equal (car oldvalue) 'nat)
                (equal (cddr oldvalue) nil)
                (numberp (cadr oldvalue))
                (lessp (cadr oldvalue) (exp 2 word-size))
                (numberp (untag newvalue))
                (lessp (untag newvalue) (exp 2 word-size))
                (member oldvalue (nthcdr i vallist)))
            (equal
             (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)))
           ((expand
             (REPLACE-VALUE-LOOP-CLOCK
              (CDADR LIST-ADDR)
              (CDR (ASSOC (CAADR LIST-ADDR)
                          DATA-SEGMENT))
              OLDVALUE))
             (induct (replace-value-loop-induct i vallist list-addr))
             (disable member-of-natlist-means)))

(defn replace-value-clock (list ov)
  (clock-plus 1 (replace-value-loop-clock 0 list ov)))

(defn replace-value-input-conditionp (p0)
            (and
                (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))
                (equal (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))
                (not (zerop (p-word-size p0)))
                (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
                (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
                (listp (untag (top (cddr (p-temp-stk p0)))))
                (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
                (equal (car (top (p-temp-stk p0))) 'nat)
                (equal (cddr (top (p-temp-stk p0))) nil)
                (numberp (untag (top (p-temp-stk p0))))
                (lessp (untag (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
                (member (top (cdr (p-temp-stk p0)))
                        (array (car (untag (top (cddr (p-temp-stk p0)))))
                               (p-data-segment p0)))))

(prove-lemma correctness-of-replace-value (rewrite)
           (implies
            (and
```

```
             (equal p0 (p-state
                        pc
                        ctrl-stk
                        (cons nv (cons ov (cons nats temp-stk)))
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run))
             (equal (p-current-instruction p0)
                    '(call replace-value))
             (replace-value-input-conditionp p0)
             (equal natlist
                    (array (car (untag nats)) data-segment))
             (equal ov ov2))
            (equal
             (p (p-state
                        pc
                        ctrl-stk
                        (cons nv (cons ov (cons nats temp-stk)))
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)
                (replace-value-clock natlist ov2))
             (p-state
              (add1-addr pc)
              ctrl-stk
              temp-stk
              prog-segment
              (put-assoc
               (replace-value natlist ov nv)
               (caadr nats) data-segment)
              max-ctrl-stk-size
              max-temp-stk-size
              word-size
              'run)))
           ((disable replace-value-loop-clock
                     nat-list-piton-means-car)
            (use (correctness-of-replace-value-general
                  (newvalue nv) (oldvalue ov) (list-addr nats)
                  (i 0) (vallist natlist) (list (caadr nats))
                  (ret-pc (add1-addr pc))))))

(disable replace-value-program)

;; smart-move

(defn 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 ()
             (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)))


(defn 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))

(defn smart-move (state wordsize)
  (if (lessp (number-with-at-least state 2) 2)
      (replace-value
       state (max-list state)
       (remainder (number-with-at-least state 1) 2))
    (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))))))

(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)))))))))))))))

(disable replace-value-clock)
(disable max-nat-clock)
(disable bv-to-nat-list-clock)
(disable nat-to-bv-list-clock)
(disable match-and-xor-clock)
(disable highest-bit-clock)
(disable number-with-at-least-clock)
(disable bv-to-nat-clock)
(disable nat-to-bv-clock)
(disable xor-bvs-clock)
(disable replace-value-program)
(disable max-nat-program)
(disable bv-to-nat-list-program)
(disable nat-to-bv-list-program)
(disable match-and-xor-program)
(disable highest-bit-program)
(disable number-with-at-least-program)
(disable bv-to-nat-program)
(disable nat-to-bv-program)
(disable push-1-vector-program)
(disable *1*xor-bvs-program)
(disable *1*replace-value-program)
(disable *1*max-nat-program)
(disable *1*bv-to-nat-list-program)
(disable *1*nat-to-bv-list-program)
(disable *1*match-and-xor-program)
(disable *1*highest-bit-program)
(disable *1*number-with-at-least-program)
(disable *1*bv-to-nat-program)
(disable *1*nat-to-bv-program)
(disable *1*push-1-vector-program)
(disable *1*xor-bvs-program)

(defn nat-listp (list size)
  (if (listp list)
      (and
       (numberp (car list))
       (lessp (car list) (exp 2 size))
       (nat-listp (cdr list) size))
    (equal list nil)))

(prove-lemma bv-to-nat-list-nat-to-bv-list (rewrite)
             (implies
              (nat-listp x size)
              (equal
               (bv-to-nat-list (nat-to-bv-list x size))
               x)))

(prove-lemma bit-vectorsp-nat-to-bv-list (rewrite)
             (bit-vectorsp (nat-to-bv-list x size) size))

(prove-lemma nat-to-bv-list-bv-to-nat-list (rewrite)
             (implies
              (bit-vectorsp x size)
              (equal
               (nat-to-bv-list (bv-to-nat-list x) size)
               x)))

(prove-lemma bit-vectorsp-match-and-xor (rewrite)
             (implies
              (bit-vectorsp x size)
              (bit-vectorsp (match-and-xor x y z) size)))

(prove-lemma equal-sub1-add1 (rewrite)
             (and
              (equal
               (equal (sub1 x) (add1 y))
               (equal x (add1 (add1 y))))
              (equal
               (equal (sub1 x) 0)
               (or (zerop x) (equal x 1)))))

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

```
(PROVE-LEMMA EQUAL-TIMES-X-X
          (REWRITE)
          (AND (EQUAL (EQUAL (TIMES X Y) X)
                   (OR (AND (NUMBERP X) (EQUAL Y 1))
                      (EQUAL X 0)))
              (EQUAL (EQUAL (TIMES Y X) X)
                   (OR (AND (NUMBERP X) (EQUAL Y 1))
                      (EQUAL X 0))))
          ((INDUCT (TIMES Y X))))



(prove-lemma equal-exp-x-y-x (rewrite)
          (equal
           (equal (exp x y) x)
           (or
            (equal x 1)
            (and
             (equal x 0)
             (not (zerop y)))
            (and
             (numberp x)
             (equal y 1)))))

(prove-lemma lessp-number-with-at-least (rewrite)
          (not (lessp (length x) (number-with-at-least x min))))

(prove-lemma bit-vectors-piton-tag-array (rewrite)
          (implies
           (bit-vectorsp x size)
           (bit-vectors-piton (tag-array 'bitv x) size)))

(prove-lemma tag-array-untag-array-of-nat-list-piton (rewrite)
          (implies
           (nat-list-piton x size)
           (equal (tag-array 'nat (untag-array x)) x)))

(prove-lemma tag-array-untag-array-of-bit-vectors-piton (rewrite)
          (implies
           (bit-vectors-piton x size)
           (equal (tag-array 'bitv (untag-array x)) x)))

(prove-lemma bit-vectorp-xor-bvs-nat-to-bv-list (rewrite)
          (equal
           (bit-vectorp (xor-bvs (nat-to-bv-list x s)) s)
           (or
            (listp x)
            (zerop s))))

(prove-lemma listp-cdr-assoc-hack-from-free (rewrite)
          (implies
           (and
            (equal (length (cdr (assoc x y))) free)
            (not (zerop free)))
           (listp (cdr (assoc x y)))))

(prove-lemma bit-vectorp-highest-bit (rewrite)
          (implies
           (bit-vectorp x s)
           (bit-vectorp (highest-bit x) s)))

(prove-lemma length-match-and-xor (rewrite)
          (equal
           (length (match-and-xor list m v))
           (length list)))

(prove-lemma array-pitonp-from-nat-list-piton (rewrite)
          (implies
           (nat-list-piton x s)
           (equal
            (array-pitonp x length)
            (equal (fix length) (length x)))))

(prove-lemma untag-array-tag-array-of-bit-vectorsp (rewrite)
          (implies
           (bit-vectorsp x size)
           (equal (untag-array (tag-array l x)) x)))

(prove-lemma untag-array-tag-array-of-nat-to-bv-list (rewrite)
          (equal
           (untag-array (tag-array l (nat-to-bv-list x size)))
           (nat-to-bv-list x size)))

(prove-lemma untag-array-tag-array-of-match-and-xor-hack (rewrite)
          (equal
           (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)))

(prove-lemma member-list-max-list (rewrite)
        (implies
         (nat-list-piton x s)
         (equal
          (member (list 'nat (max-list (untag-array x))) x)
          (listp x))))

(prove-lemma tag-array-replace-value-untag-array (rewrite)
        (implies
         (nat-list-piton x s)
         (equal
          (tag-array 'nat (replace-value (untag-array x) y z))
          (replace-value x (list 'nat y) (list 'nat z)))))


(defn smart-move-input-conditionp (p0)
 (and
  (listp (p-ctrl-stk p0))
  (at-least-morep (length (p-temp-stk p0))
                  3 (p-max-temp-stk-size p0))
  (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                  19 (p-max-ctrl-stk-size p0))
  (lessp 1 (p-word-size p0))
  (equal (cddr (top (p-temp-stk p0))) nil)
  (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
  (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
  (equal (car (top (p-temp-stk p0))) 'addr)
  (equal (car (top (cdr (p-temp-stk p0)))) 'nat)
  (lessp (untag (top (cdr (p-temp-stk p0)))) (exp 2 (p-word-size p0)))
  (not (zerop (untag (top (cdr (p-temp-stk p0))))))
  (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
  (listp (untag (top (p-temp-stk p0))))
  (listp (untag (top (cddr (p-temp-stk p0)))))
  (equal (cdr (untag (top (p-temp-stk p0)))) 0)
  (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
  (definedp (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
  (definedp (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
  (not (equal (car (untag (top (p-temp-stk p0)))) (car (untag (top (cddr (p-temp-stk p0)))))))
  (nat-list-piton (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
                  (p-word-size p0))
  (array-pitonp (array (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
                (untag (top (cdr (p-temp-stk p0)))))
  (equal (length (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0)))
         (untag (top (cdr (p-temp-stk p0)))))
  (equal (assoc 'smart-move (p-prog-segment p0))
         (smart-move-program))
  (equal (assoc 'replace-value (p-prog-segment p0))
         (replace-value-program))
  (equal (assoc 'max-nat (p-prog-segment p0))
         (max-nat-program))
  (equal (assoc 'bv-to-nat-list (p-prog-segment p0))
         (bv-to-nat-list-program))
  (equal (assoc 'nat-to-bv-list (p-prog-segment p0))
         (nat-to-bv-list-program))
  (equal (assoc 'match-and-xor (p-prog-segment p0))
         (match-and-xor-program))
  (equal (assoc 'highest-bit (p-prog-segment p0))
         (highest-bit-program))
  (equal (assoc 'number-with-at-least (p-prog-segment p0))
         (number-with-at-least-program))
  (equal (assoc 'bv-to-nat (p-prog-segment p0))
         (bv-to-nat-program))
  (equal (assoc 'nat-to-bv (p-prog-segment p0))
         (nat-to-bv-program))
  (equal (assoc 'push-1-vector (p-prog-segment p0))
         (push-1-vector-program (p-word-size p0)))
  (equal (assoc 'xor-bvs (p-prog-segment p0))
         (xor-bvs-program))))

(prove-lemma correctness-of-smart-move (rewrite)
        (implies
         (and
          (equal 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))
          (equal (p-current-instruction p0)
                 '(call smart-move))
```

```
            (equal state
               (untag-array (array (car (untag s)) data-segment)))
            (equal word-size word-size2)
            (smart-move-input-conditionp p0))
                  (equal
            (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 (lessp (number-with-at-least state 2) 2)
                  data-segment
                (put-assoc
                 (tag-array 'bitv
                  (nat-to-bv-list (smart-move state word-size)
                                  word-size))
                 (car (untag wa)) data-segment)))
             max-ctrl-stk-size
             max-temp-stk-size
             word-size
             'run))))

(disable smart-move-clock)
(disable smart-move-program)
(disable *1*smart-move-program)

;; delay

(defn delay-program ()
  '(delay (time) ()
         (dl lab ()
             (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 () (ret))))


(defn 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))


(defn delay-loop-clock (time)
  (if (lessp time 2) 9 (plus 10 (delay-loop-clock (sub1 time)))))

(prove-lemma correctness-of-delay-general nil
           (implies
            (and
               (listp ctrl-stk)
               (equal (definition 'delay prog-segment)
                      (delay-program))
               (at-least-morep (length temp-stk) 1 max-temp-stk-size)
               (lessp 0 time)
               (lessp time (exp 2 word-size)))
            (equal
```

```
             (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)))
           ((induct (times time y))))

(defn delay-input-conditionp (p0)
  (and
   (listp (p-ctrl-stk p0))
   (equal (definition 'delay (p-prog-segment p0)) (delay-program))
   (equal (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))
   (lessp 0 (cadr (top (p-temp-stk p0))))
   (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (equal (cddr (top (p-temp-stk p0))) nil)))

(defn delay-clock (time)
  (add1 (delay-loop-clock time)))

(prove-lemma correctness-of-delay (rewrite)
           (implies
            (and
             (equal p0 (p-state
                        pc
                        ctrl-stk
                        (cons n temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run))
             (equal (p-current-instruction p0) '(call delay))
             (delay-input-conditionp p0)
             (equal time (cadr n)))
                   (equal
             (p (p-state
                        pc
                        ctrl-stk
                        (cons n temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)
                (delay-clock time))
             (p-state
              (add1-addr pc)
              ctrl-stk
              temp-stk
              prog-segment
              data-segment
              max-ctrl-stk-size
              max-temp-stk-size
              word-size
              'run)))
           ((use (correctness-of-delay-general
                  (time (cadr n))
                  (ret-pc (add1-addr pc))))))

;; computer-move

(defn 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 ()
             (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)))

(defn 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))

(defn computer-move (state wordsize)
  (if (or (equal (number-with-at-least state 2) 0)
```

```
        (all-zero-bitvp (xor-bvs (nat-to-bv-list state wordsize))))
    (replace-value state (max-list state) (sub1 (max-list state)))
  (smart-move state wordsize)))

(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))))))))))))))))))))

(defn computer-move-input-conditionp (p0)
 (and
  (not (all-zero-bitvp
       (untag-array
        (array (car (untag (top (cddr (p-temp-stk p0)))))
               (p-data-segment p0)))))
  (listp (p-ctrl-stk p0))
  (at-least-morep (length (p-temp-stk p0))
                  3 (p-max-temp-stk-size p0))
  (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                  25 (p-max-ctrl-stk-size p0))
  (lessp 7 (p-word-size p0))
  (not (zerop (p-word-size p0)))
  (not (equal (p-word-size p0) 1))
  (equal (cddr (top (p-temp-stk p0))) nil)
  (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
  (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
  (equal (car (top (p-temp-stk p0))) 'addr)
  (equal (car (top (cdr (p-temp-stk p0)))) 'nat)
  (lessp (untag (top (cdr (p-temp-stk p0)))) (exp 2 (p-word-size p0)))
  (not (zerop (untag (top (cdr (p-temp-stk p0))))))
  (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
  (listp (untag (top (p-temp-stk p0))))
  (listp (untag (top (cddr (p-temp-stk p0)))))
  (equal (cdr (untag (top (p-temp-stk p0)))) 0)
  (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
  (definedp (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
  (definedp (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
```

```
  (not (equal (car (untag (top (p-temp-stk p0)))) (car (untag (top (cddr (p-temp-stk p0)))))))
  (nat-list-piton (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
                  (p-word-size p0))
  (array-pitonp (array (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
                (untag (top (cdr (p-temp-stk p0)))))
  (equal (length (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0)))
        (untag (top (cdr (p-temp-stk p0)))))
  (equal (assoc 'delay (p-prog-segment p0))
        (delay-program))
  (equal (assoc 'computer-move (p-prog-segment p0))
        (computer-move-program))
  (equal (assoc 'smart-move (p-prog-segment p0))
        (smart-move-program))
  (equal (assoc 'replace-value (p-prog-segment p0))
        (replace-value-program))
  (equal (assoc 'max-nat (p-prog-segment p0))
        (max-nat-program))
  (equal (assoc 'bv-to-nat-list (p-prog-segment p0))
        (bv-to-nat-list-program))
  (equal (assoc 'nat-to-bv-list (p-prog-segment p0))
        (nat-to-bv-list-program))
  (equal (assoc 'match-and-xor (p-prog-segment p0))
        (match-and-xor-program))
  (equal (assoc 'highest-bit (p-prog-segment p0))
        (highest-bit-program))
  (equal (assoc 'number-with-at-least (p-prog-segment p0))
        (number-with-at-least-program))
  (equal (assoc 'bv-to-nat (p-prog-segment p0))
        (bv-to-nat-program))
  (equal (assoc 'nat-to-bv (p-prog-segment p0))
        (nat-to-bv-program))
  (equal (assoc 'push-1-vector (p-prog-segment p0))
        (push-1-vector-program (p-word-size p0)))
  (equal (assoc 'xor-bvs (p-prog-segment p0))
        (xor-bvs-program))))

(prove-lemma numberp-max-list (rewrite)
           (numberp (max-list x)))

(prove-lemma max-0-means (rewrite)
           (implies
            (nat-list-piton x s)
            (equal
             (equal (max-list (untag-array x)) 0)
             (all-zero-bitvp (untag-array x)))))

(prove-lemma max-list-not-too-big (rewrite)
           (implies
            (and
             (nat-list-piton x s)
             (not (zerop s)))
            (and
             (lessp (max-list (untag-array x)) (exp 2 s))
             (equal (lessp (sub1 (max-list (untag-array x))) (exp 2 s)) t))))

(disable delay-clock)
(disable *1*delay-clock)

(prove-lemma lessp-exp-2-8-hack (rewrite)
           (implies
            (and
             (lessp 7 free)
             (equal x free)
             (lessp val 256))
            (equal (lessp val (exp 2 x)) t)))

(prove-lemma correctness-of-computer-move (rewrite)
           (implies
            (and
             (equal 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))
             (equal (p-current-instruction p0)
                   '(call computer-move))
             (equal state
                 (untag-array (array (car (untag s)) data-segment)))
             (equal word-size word-size2)
             (computer-move-input-conditionp p0))
                  (equal
                   (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 (equal (number-with-at-least state 2) 0)
                data-segment
              (put-assoc
               (if (or (all-zero-bitvp
                        (xor-bvs (nat-to-bv-list state word-size)))
                       (lessp (number-with-at-least state 2) 2))
                   (tag-array 'bitv
                              (nat-to-bv-list state word-size))
                 (tag-array 'bitv
                            (nat-to-bv-list (computer-move state word-size)
                                            word-size)))
               (car (untag wa)) data-segment)))
           max-ctrl-stk-size
           max-temp-stk-size
           word-size
           'run)))
         ((disable smart-move)))

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

(defn sum (list)
  (if (listp list)
      (plus (car list) (sum (cdr list)))
    0))

(prove-lemma sum-append (rewrite)
             (equal
              (sum (append x y))
              (plus (sum x) (sum y))))

;; returns a list of states that are valid moves from
(defn all-valid-moves-helper (old val origval new)
  (if (zerop val)
      (if (nlistp new)
          nil
        (all-valid-moves-helper (append old (list origval))
                                (car new) (car new) (cdr new)))
    (cons (append old (cons (sub1 val) new))
          (all-valid-moves-helper old (sub1 val) origval new)))
  ((ord-lessp (cons (add1 (length new)) (fix val)))))

(defn all-valid-moves (x)
  (all-valid-moves-helper nil (car x) (car x) (cdr x)))

(defn max-sum (list)
  (if (listp list)
      (if (lessp (sum (car list))
                 (max-sum (cdr list)))
          (max-sum (cdr list))
        (sum (car list)))
    0))

(prove-lemma nat-listp-append (rewrite)
             (implies
              (properp x)
             (equal
              (nat-listp (append x y) size)
              (and
               (nat-listp x size)
               (nat-listp y size)))))
```

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

(defn nat-listp-simple (list)
  (if (listp list)
      (and
       (numberp (car list))
       (nat-listp-simple (cdr list)))
    (equal list nil)))

(prove-lemma nat-listp-simple-append (rewrite)
             (implies
              (properp x)
             (equal
              (nat-listp-simple (append x y))
              (and (nat-listp-simple x) (nat-listp-simple y)))))

(prove-lemma lessp-max-sum-helper nil
             (implies
              (and
               (not (lessp c temp))
               (properp x)
               (nat-listp-simple (append x (cons c y)))
               (not (all-zero-bitvp (append x (cons c y)))))
              (lessp
               (max-sum (all-valid-moves-helper x temp c y))
               (sum (append x (cons c y))))))

(prove-lemma lessp-max-sum-all-valid-moves (rewrite)
             (implies
              (and
               (nat-listp-simple s)
               (not (all-zero-bitvp s)))
              (equal (lessp (max-sum (all-valid-moves s)) (sum s)) t))
             ((use (lessp-max-sum-helper (c (car s)) (temp (car s)) (x nil)
                                         (y (cdr s))))))

(prove-lemma lessp-sum-max-sum (rewrite)
             (not (lessp (max-sum x) (sum (car x)))))

(disable all-valid-moves)


(defn wsp-measure (state flag)
  (cons (if flag (add1 (sum state)) (add1 (max-sum state)))
        (if flag 0 (length state))))

;; 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.

(defn wsp (state flag)
  (if flag
      (if (or (all-zero-bitvp state) (not (nat-listp-simple state)))
          state
        (wsp (all-valid-moves state) f))
    (if (listp state)
        (if (not (wsp (car state) t))
            (car state)
          (wsp (cdr state) f))
      f))
  ((ord-lessp (wsp-measure state flag))))

(defn green-statep (state wordsize)
  (equal
   (zerop (number-with-at-least state 2))
   (all-zero-bitvp (xor-bvs (nat-to-bv-list state wordsize)))))

(defn non-green-in-list (list wordsize)
  (if (listp list)
      (or
       (not (green-statep (car list) wordsize))
       (non-green-in-list (cdr list) wordsize))
    f))

(prove-lemma nat-listp-means-nat-listp-simple (rewrite)
             (implies
              (nat-listp x s)
              (nat-listp-simple x)))
```

```
(prove-lemma xor-bitv-zero-bit-vector (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (xor-bitv x y)
                     (make-list-from (length x) (fix-bitv y)))))

(prove-lemma fix-bitv-xor-bitv (rewrite)
             (equal (fix-bitv (xor-bitv x y)) (xor-bitv x y)))

(prove-lemma fix-bitv-and-bitv (rewrite)
             (equal (fix-bitv (and-bitv x y)) (and-bitv x y)))

(prove-lemma fix-bitv-xor-bvs (rewrite)
             (equal (fix-bitv (xor-bvs x)) (xor-bvs x)))

(prove-lemma all-zero-bitvp-make-list-from-simple (rewrite)
             (implies
              (all-zero-bitvp x)
              (all-zero-bitvp (make-list-from n x))))

(prove-lemma all-zero-bitvp-xor-bvs-nat-to-bv-list-zeros (rewrite)
             (implies
              (all-zero-bitvp x)
              (all-zero-bitvp (xor-bvs (nat-to-bv-list x s)))))

(prove-lemma number-with-at-least-of-all-zeros (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (number-with-at-least x m)
                     (if (zerop m) (length x) 0))))

(defn nat-listp-listp (list wordsize)
  (if (listp list)
      (and
       (nat-listp (car list) wordsize)
       (nat-listp-listp (cdr list) wordsize))
    t))

(prove-lemma nat-listp-listp-all-valid-moves-helper (rewrite)
             (implies
              (and
               (nat-listp a s)
               (lessp b (exp 2 s))
               (lessp c (exp 2 s))
               (numberp b)
               (numberp c)
               (nat-listp d s)
               (properp a))
             (nat-listp-listp (all-valid-moves-helper a b c d) s)))

(prove-lemma nat-listp-listp-all-valid-moves (rewrite)
             (implies
               (nat-listp a s)
               (nat-listp-listp (all-valid-moves a) s))
             ((enable all-valid-moves)))

(prove-lemma listp-all-valid-move-helper (rewrite)
             (implies
              (nat-listp-simple d)
              (equal (listp (all-valid-moves-helper a b c d))
                     (or
                      (not (all-zero-bitvp d))
                      (not (zerop b))))))

(prove-lemma listp-all-valid-move (rewrite)
             (implies
              (nat-listp-simple x)
              (equal (listp (all-valid-moves x))
                     (not (all-zero-bitvp x))))
             ((enable all-valid-moves)))

(prove-lemma number-with-at-least-append (rewrite)
             (equal
              (number-with-at-least (append x y) m)
              (plus
               (number-with-at-least x m)
               (number-with-at-least y m))))

(prove-lemma length-xor-bvs2 (rewrite)
             (implies
              (bit-vectorsp x s)
              (equal (length (xor-bvs x))
                     (if (listp x) (fix s) 0))))

(prove-lemma xor-bitv-xor-bvs-hack (rewrite)
             (implies
```

```
             (bit-vectorsp z (length y))
            (equal
             (xor-bitv (xor-bitv y (xor-bvs z)) x)
             (if (listp z)
                 (xor-bitv y (xor-bitv (xor-bvs z) x))
               (xor-bitv y x)))))

(prove-lemma xor-bitv-fix-bitv (rewrite)
            (and
             (equal (xor-bitv (fix-bitv x) y) (xor-bitv x y))
             (equal (xor-bitv x (fix-bitv y)) (xor-bitv x y))))

(prove-lemma xor-bvs-append (rewrite)
            (implies
             (and
              (bit-vectorsp x s)
              (bit-vectorsp y s)
              (numberp s))
            (equal
             (xor-bvs (append x y))
             (if (listp x)
                 (xor-bitv (xor-bvs x) (xor-bvs y))
               (xor-bvs y)))))

(prove-lemma xor-bvs-append-hack (rewrite)
            (implies
             (bit-vectorsp y ws)
            (equal
             (xor-bvs (append (nat-to-bv-list a ws) y))
             (if (listp a)
                 (xor-bitv (xor-bvs (nat-to-bv-list a ws))
                           (xor-bvs y))
               (xor-bvs y))))
            ((use (xor-bvs-append
                   (x (nat-to-bv-list a ws))
                   (s ws)))))

(prove-lemma nat-to-bv-list-append (rewrite)
            (equal
             (nat-to-bv-list (append x y) s)
             (append
              (nat-to-bv-list x s)
              (nat-to-bv-list y s))))

(prove-lemma bit-vectorp-nat-to-bv (rewrite)
            (equal
             (bit-vectorp (nat-to-bv x s) s2)
             (equal (fix s) (fix s2))))

(prove-lemma fix-bitv-nat-to-bv (rewrite)
            (equal (fix-bitv (nat-to-bv x s)) (nat-to-bv x s)))

(prove-lemma fix-bitv-zero-bit-vector (rewrite)
            (equal
             (fix-bitv (zero-bit-vector x))
             (zero-bit-vector x)))

(prove-lemma all-zero-bitvp-nat-to-bv (rewrite)
            (equal
             (all-zero-bitvp (nat-to-bv x s))
             (or
              (zerop x)
              (zerop s))))

(prove-lemma all-zero-bitvp-xor-bitv-better (rewrite)
            (equal
             (all-zero-bitvp (xor-bitv x y))
             (equal
              (fix-bitv x)
              (make-list-from (length x) (fix-bitv y)))))

(prove-lemma length-xor-bvs-nat-to-bv-list (rewrite)
            (equal
             (length (xor-bvs (nat-to-bv-list x s)))
             (if (listp x) (fix s) 0)))

(prove-lemma fix-bitv-make-list-from (rewrite)
            (equal
             (fix-bitv (make-list-from s x))
             (make-list-from s (fix-bitv x))))


(defn double-sub1-cdr (n1 n2 l)
  (if (or (zerop n1) (zerop n2))
      t
    (double-sub1-cdr (sub1 n1) (sub1 n2) (cdr l))))
```

```
(prove-lemma make-list-from-make-list-from (rewrite)
             (equal
              (make-list-from s1 (make-list-from s2 x))
              (if (lessp s2 s1)
                  (append (make-list-from s2 x)
                          (zero-bit-vector (difference s1 s2)))
                (make-list-from s1 x))))

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

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

(prove-lemma properp-xor-bvs (rewrite)
             (properp (xor-bvs x)))

(prove-lemma properp-xor-bitv (rewrite)
             (properp (xor-bitv x y)))

(defn member-number-with-at-least (x min)
  (if (listp x)
      (if (not (zerop (number-with-at-least (car x) min)))
          (car x)
        (member-number-with-at-least (cdr x) min))
    f))

(prove-lemma xor-bvs-nat-to-bv-list-zerop-ws (rewrite)
             (implies
              (zerop ws)
              (equal (xor-bvs (nat-to-bv-list x ws)) nil)))

(defn nat-listp-listp-simple (x)
  (if (listp x)
      (and
       (nat-listp-simple (car x))
       (nat-listp-listp-simple (cdr x)))
    (equal x nil)))

(prove-lemma non-green-in-list-zerop-ws (rewrite)
             (implies
              (and
               (zerop ws)
               (nat-listp-listp-simple x))
              (iff
               (non-green-in-list x ws)
               (member-number-with-at-least x 2))))

(prove-lemma nat-listp-listp-simple-means-properp (rewrite)
             (implies
              (nat-listp-listp-simple x)
              (properp x)))


(prove-lemma nat-listp-simple-means-properp (rewrite)
             (implies
              (nat-listp-simple x)
              (properp x)))

(prove-lemma make-list-from-xor-bvs-nat-to-bv-list (rewrite)
             (implies
              (equal (fix ws) (fix s))
              (equal
               (make-list-from ws (xor-bvs (nat-to-bv-list x s)))
               (if (listp x)
                   (xor-bvs (nat-to-bv-list x s))
                 (zero-bit-vector ws)))))


(prove-lemma last-xor-bitv (rewrite)
             (equal
              (last (xor-bitv x y))
              (if (listp x)
                  (xor-bit (last x) (nth (sub1 (length x)) y))
                0)))

(prove-lemma last-one-bit-vector (rewrite)
             (equal (last (one-bit-vector x)) 1))

(prove-lemma nth-as-last (rewrite)
```

```
            (implies
             (equal (add1 n) (length x))
             (equal (nth n x) (last x))))

(prove-lemma listp-nat-to-bv (rewrite)
            (equal
             (listp (nat-to-bv x s))
             (not (zerop s))))

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

(prove-lemma last-nat-to-bv (rewrite)
            (equal
             (last (nat-to-bv x s))
             (if  (or (zerop s)
                      (and
                       (lessp x (exp 2 s))
                       (equal (remainder x 2) 0)))
                 0
               1)))

(prove-lemma fix-bitv-one-bit-vector (rewrite)
            (equal
             (fix-bitv (one-bit-vector x))
             (one-bit-vector x)))

(prove-lemma nat-to-bv-1 (rewrite)
            (equal
             (nat-to-bv 1 x)
             (if (zerop x) nil (one-bit-vector x))))

(prove-lemma last-zero-bit-vector (rewrite)
            (equal (last (zero-bit-vector x)) 0))

(prove-lemma nat-to-bv-2 (rewrite)
            (equal
             (nat-to-bv x 2)
             (if (zerop x)
                 (list 0 0)
               (if (equal x 1)
                   (list 0 1)
                 (if (equal x 2)
                     (list 1 0)
                   (list 1 1)))))
            ((expand (nat-to-bv x 2))))

(prove-lemma all-zero-bitvp-all-but-last-nat-to-bv (rewrite)
            (equal
             (all-zero-bitvp (all-but-last (nat-to-bv x s)))
             (or
              (lessp x 2)
              (lessp s 2))))

(prove-lemma xor-bvs-of-list-of-0s-and-1s (rewrite)
            (implies
             (zerop (number-with-at-least c 2))
             (equal
              (xor-bvs (nat-to-bv-list c ws))
              (if (or (nlistp c) (zerop ws)) nil
                (if (equal (remainder (sum c) 2) 0)
                    (zero-bit-vector ws)
                  (one-bit-vector ws))))))

(prove-lemma equal-nat-to-bv-nlistp (rewrite)
            (implies
             (nlistp x)
             (equal
              (equal x (nat-to-bv y s))
              (and (equal x nil) (zerop s)))))

(prove-lemma different-lengths-means-different-hack nil
            (implies
             (not (equal (fix s1) (fix s2)))
             (not (equal (length (nat-to-bv x s1))
                         (length (nat-to-bv y s2))))))

(prove-lemma different-lengths-means-different (rewrite)
            (implies
             (not (equal (fix s1) (fix s2)))
             (not (equal (nat-to-bv x s1) (nat-to-bv y s2))))
            ((use (different-lengths-means-different-hack))
             (disable-theory t)
             (enable-theory ground-zero)))

(defn nat-to-bv-induct (x y s1 s2)
```

```
  (if (zerop s1) t
    (nat-to-bv-induct
     (if (lessp x (exp 2 (sub1 s1))) x
       (difference x (exp 2 (sub1 s1))))
     (if (lessp y (exp 2 (sub1 s2))) y
       (difference y (exp 2 (sub1 s2)) ))
     (sub1 s1)
     (sub1 s2))))


(defn all-ones-vector (x)
  (if (zerop x)
      nil
    (cons 1 (all-ones-vector (sub1 x)))))

(prove-lemma not-lessp-exp-means-all-ones  (rewrite)
             (implies
              (not (lessp x (sub1 (exp 2 s))))
              (equal (nat-to-bv x s) (all-ones-vector s))))

(prove-lemma lessp-sub1-plus-sub1-hack (rewrite)
             (implies
              (not (zerop y))
             (equal
              (lessp (sub1 (plus x y))
                     (plus (sub1 y) z))
              (lessp x z))))

(prove-lemma equal-cons-zero-bit-vector-nat-to-bv (rewrite)
             (equal
              (equal (cons 0 (zero-bit-vector x)) (nat-to-bv a b))
              (and
               (equal (add1 x) (fix b))
               (zerop a))))

(prove-lemma equal-all-ones-vector-all-ones-vector (rewrite)
             (equal
              (equal (all-ones-vector x) (all-ones-vector y))
              (equal (fix x) (fix y)))
             ((induct (double-sub1-cdr x y l))))

(prove-lemma equal-all-ones-vector-cons (rewrite)
             (equal
              (equal (all-ones-vector x) (cons a b))
              (and
               (not (zerop x))
               (equal a 1)
               (equal (all-ones-vector (sub1 x)) b))))

(prove-lemma different-lengths-obvious nil
             (implies
              (equal x y)
              (equal (length x) (length y))))

(prove-lemma equal-all-ones-vector-nlistp (rewrite)
             (implies
              (nlistp x)
              (equal
               (equal (all-ones-vector y) x)
               (and
                (equal x nil)
                (zerop y)))))


(prove-lemma length-all-ones-vector (rewrite)
             (equal (length (all-ones-vector x)) (fix x)))

(prove-lemma different-lengths-hack (rewrite)
             (implies
              (not (equal (fix x) (fix y)))
              (not (equal (all-ones-vector x) (nat-to-bv a y))))
             ((use (different-lengths-obvious
                    (x (all-ones-vector x))
                    (y (nat-to-bv a y))))
              (disable equal-all-ones-vector-nlistp)))

(prove-lemma lessp-difference-arg1 (rewrite)
             (implies
              (not (lessp x y))
              (equal (lessp (difference x y) z)
                     (lessp x (plus y z)))))

(prove-lemma equal-difference (rewrite)
             (implies
              (not (lessp x y))
              (equal
               (equal (difference x y) z)
```

```
              (and
               (equal (fix x) (plus y z))
               (numberp z)))))

(prove-lemma equal-exp (rewrite)
             (implies
              (equal (fix a) (fix b))
              (equal
               (equal (exp a c) (exp b d))
               (or
                (equal a 1)
                (and (zerop a) (equal (zerop c) (zerop d)))
                (equal (fix c) (fix d)))))
             ((induct (double-sub1-cdr c d l))))

(prove-lemma equal-all-ones-nat-to-bv (rewrite)
             (equal
              (equal (all-ones-vector x) (nat-to-bv a b))
              (and
               (equal (fix x) (fix b))
               (not (lessp a (sub1 (exp 2 b))))))
             ((induct (nat-to-bv-induct q a x b))))

(prove-lemma equal-nat-to-bv-nat-to-bv (rewrite)
             (equal
              (equal (nat-to-bv x s1) (nat-to-bv y s2))
              (and
               (equal (fix s1) (fix s2))
               (or
                (equal (fix x) (fix y))
                (and (not (lessp x (sub1 (exp 2 s1))))
                     (not (lessp y (sub1 (exp 2 s1))))))))
             ((induct (nat-to-bv-induct x y s1 s2))))

(prove-lemma listp-xor-bvs (rewrite)
             (equal
              (listp (xor-bvs x))
              (listp (car x))))

(prove-lemma car-nat-to-bv-list (rewrite)
             (equal
              (car (nat-to-bv-list x s))
              (if (listp x)
                  (nat-to-bv (car x) s)
                0)))

;; from later version of naturals that is used in this proof
(PROVE-LEMMA QUOTIENT-DIFFERENCE
             (REWRITE)
             (EQUAL (QUOTIENT (DIFFERENCE X Y) Z)
                    (IF (LESSP (REMAINDER X Z)
                               (REMAINDER Y Z))
                        (SUB1 (DIFFERENCE (QUOTIENT X Z)
                                          (QUOTIENT Y Z)))
                        (DIFFERENCE (QUOTIENT X Z)
                                    (QUOTIENT Y Z))))
             ((DISABLE QUOTIENT-DIFFERENCE1 QUOTIENT-DIFFERENCE2
                       QUOTIENT-DIFFERENCE3)
              (INDUCT (QUOTIENT Y Z))))

(enable DIFFERENCE-X-SUB1-X)

(prove-lemma all-but-last-nat-to-bv (rewrite)
             (equal
              (all-but-last (nat-to-bv x s))
              (if (zerop s)
                  nil
                (nat-to-bv (quotient x 2) (sub1 s)))))

(prove-lemma equal-last-xor-bvs-1 (rewrite)
             (equal
              (equal (last (xor-bvs x)) 1)
              (not (equal (last (xor-bvs x)) 0))))

(prove-lemma not-green-state-means (rewrite)
             (implies
              (and
               (not (green-statep (append a (cons b c)) ws))
               (lessp d (exp 2 ws))
               (lessp b d))
              (equal
               (green-statep (append a (cons d c)) ws)
               (or (not (zerop ws))
                   (zerop (number-with-at-least
                           (append a (cons d c)) 2))))))

(prove-lemma green-in-list-all-valid-moves-helper nil
```

```
            (implies
             (and
              (nat-listp a ws)
              (nat-listp d ws)
              (numberp c)
              (lessp c (exp 2 ws))
              (properp a)
              (non-green-in-list
               (all-valid-moves-helper a b c d) ws)
              (not (lessp c b))
              (numberp b))
             (green-statep (append a (cons c d)) ws))
            ((disable green-statep)))

(prove-lemma green-in-list-all-valid-moves-means (rewrite)
            (implies
             (and
              (nat-listp x ws)
              (non-green-in-list (all-valid-moves x) ws))
             (green-statep x ws))
            ((disable green-statep)
             (enable all-valid-moves)
             (use (green-in-list-all-valid-moves-helper
                   (a nil) (b (car x)) (c (car x)) (d (cdr x))))))

(defn valid-movep (s1 s2)
  (if (and (listp s1) (listp s2))
      (or
       (and
        (lessp (car s2) (car s1))
        (numberp (car s2))
        (equal (cdr s1) (cdr s2)))
       (and
        (equal (car s1) (car s2))
        (valid-movep (cdr s1) (cdr s2))))
    f))

(defn match-member (m list)
  (if (listp list)
      (if (not (all-zero-bitvp (and-bitv (car list) m)))
          (car list)
        (match-member m (cdr list)))
    f))

(prove-lemma xor-bvs-match-and-xor (rewrite)
            (implies
             (bit-vectorsp list (length value))
            (equal
             (xor-bvs (match-and-xor list match value))
             (if (match-member match list)
                 (xor-bitv value (xor-bvs list))
               (xor-bvs list)))))

(defn remove-highest-bits (x)
  (if (listp x)
      (cons (cdar x) (remove-highest-bits (cdr x)))
    nil))

(prove-lemma car-remove-highest-bits (rewrite)
            (equal
             (car (remove-highest-bits x))
             (cdr (car x))))

(prove-lemma equal-car-highest-bit-1 (rewrite)
            (equal
             (equal (car (highest-bit x)) 1)
             (equal
              (highest-bit x)
              (cons 1 (zero-bit-vector (sub1 (length x)))))))

(prove-lemma car-xor-bitv (rewrite)
            (equal
             (car (xor-bitv x y))
             (if (listp x)
                 (xor-bit (car x) (car y))
               0)))

(prove-lemma match-member-cons-0 (rewrite)
            (iff
             (match-member (cons 0 x) y)
             (match-member x (remove-highest-bits y))))

(prove-lemma match-member-cons (rewrite)
            (implies
             (and
              (bit-vectorsp v (length (cons a b)))
              (not (equal (car (xor-bvs v)) 0)))
```

```
             (iff
              (match-member (cons a b) v)
              (or
               (not (equal a 0))
               (match-member b (remove-highest-bits v))))))

(prove-lemma equal-highest-bit-cons-1 (rewrite)
            (equal
             (equal (highest-bit x) (cons 1 y))
             (and
              (not (equal (car x) 0))
              (equal y (zero-bit-vector (sub1 (length x)))))))

(prove-lemma length-cdr-xor-bitv (rewrite)
            (equal
             (length (cdr (xor-bitv x y)))
             (length (cdr x))))

(prove-lemma length-fix-bitv (rewrite)
            (equal (length (fix-bitv x)) (length x)))

(prove-lemma length-cdr-xor-bvs (rewrite)
            (implies
             (bit-vectorsp x s)
            (equal
             (length (cdr (xor-bvs x)))
             (if (listp x) (sub1 s) 0))))

(defn highest-bits-induct (x s)
  (if (listp x)
     (if (listp (car x))
         (if (equal (car (highest-bit (xor-bvs x))) 1)
             t
           (highest-bits-induct (remove-highest-bits x) (sub1 s)))
       t)
    t)
  ((lessp (length (car x)))))

(prove-lemma bit-vectorsp-remove-highest-bits (rewrite)
            (implies
             (bit-vectorsp x (add1 s))
             (bit-vectorsp (remove-highest-bits x) s)))

(prove-lemma xor-bvs-remove-highest-bits (rewrite)
            (implies
             (bit-vectorsp x s)
            (equal
             (xor-bvs (remove-highest-bits x))
             (if (and (listp x) (not (zerop s)))
                 (cdr (xor-bvs x))
               nil))))

(prove-lemma match-member-highest-bit-xor-bvs nil
            (implies
             (bit-vectorsp x s)
             (iff
             (match-member (highest-bit (xor-bvs x)) x)
             (not (all-zero-bitvp (xor-bvs x)))))
            ((induct (highest-bits-induct x s))))

(prove-lemma match-member-highest-bit-xor-bvs-rewrite (rewrite)
            (implies
             (bit-vectorsp x (length (xor-bvs x)))
             (iff
             (match-member (highest-bit (xor-bvs x)) x)
             (not (all-zero-bitvp (xor-bvs x)))))
            ((use (match-member-highest-bit-xor-bvs
                   (s (length (xor-bvs x)))))))

(prove-lemma bit-vectorsp-nat-to-bv-list-better (rewrite)
            (equal
             (bit-vectorsp (nat-to-bv-list x size) size2)
             (or
              (nlistp x)
              (equal (fix size) (fix size2)))))

(prove-lemma match-member-at-least-min-means (rewrite)
            (implies
             (and
              (match-member y (nat-to-bv-list x ws))
              (not (lessp
                    (bv-to-nat
                     (match-member y (nat-to-bv-list x ws)))
                    min)))
             (not (equal (number-with-at-least x min) 0))))

(prove-lemma bit-vectorp-highest-bit-xor-bvs (rewrite)
```

```
          (equal
           (bit-vectorp (highest-bit (xor-bvs x)) ws)
           (bit-vectorp (xor-bvs x) ws)))

(prove-lemma number-with-at-least-match-and-xor (rewrite)
          (implies
           (and
            (nat-listp x ws)
            (bit-vectorp y ws)
            (bit-vectorp z ws))
           (equal
            (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))
               (difference
                (plus
                 (number-with-at-least x min)
                 (if (lessp
                      (bv-to-nat
                       (xor-bitv z
                        (match-member y (nat-to-bv-list x ws))))
                      min) 0 1))
                (if (lessp
                     (bv-to-nat
                      (match-member y (nat-to-bv-list x ws)))
                     min) 0 1))
             (number-with-at-least x min)))))

(prove-lemma number-with-at-least-replace-value (rewrite)
          (equal
           (number-with-at-least (replace-value x e v) min)
           (if (member e x)
               (difference
                (plus
                 (number-with-at-least x min)
                 (if (lessp v min) 0 1))
                (if (lessp e min) 0 1))
             (number-with-at-least x min))))

(prove-lemma max-list-means-number-0 (rewrite)
          (implies
           (and
            (equal (max-list x) n)
            (lessp n m))
           (equal (number-with-at-least x m) 0)))

(prove-lemma member-max-list (rewrite)
          (implies
           (nat-listp-simple x)
           (equal
            (member (max-list x) x)
            (listp x))))

(prove-lemma listp-replace-value (rewrite)
          (equal
           (listp (replace-value x e v))
           (listp x)))

(prove-lemma member-means-lessp-sum (rewrite)
          (implies
           (member e x)
           (not (lessp (sum x) e))))

(prove-lemma sum-replace-value (rewrite)
          (equal
           (sum (replace-value x e v))
           (if (member e x)
               (difference (plus (sum x) v) e)
             (sum x))))


(prove-lemma remainder-difference-2 (rewrite)
          (equal
           (remainder (difference x y) 2)
           (if (lessp x y) 0
             (if (equal (remainder x 2) (remainder y 2))
                 0
               1))))

(prove-lemma lessp-max-list (rewrite)
          (not (lessp (sum x) (max-list x))))

(prove-lemma remainder-plus-remainder (rewrite)
          (and
```

```
           (equal (remainder (plus (remainder x y) z) y)
                  (remainder (plus x z) y))
           (equal (remainder (plus z (remainder x y)) y)
                  (remainder (plus x z) y))))

(prove-lemma remainder-plus-remainder2 (rewrite)
          (and
           (equal (remainder (plus z (plus a (remainder x y))) y)
                  (remainder (plus z (plus a x)) y))
           (equal (remainder (plus z (plus (remainder x y) a)) y)
                  (remainder (plus z (plus a x)) y)))
          ((use (remainder-plus-remainder (z (plus z a))))
           (disable remainder-plus-remainder)))

(prove-lemma lessp-max-list-from-number-with-at-least (rewrite)
          (implies
           (and
            (equal (number-with-at-least x m) 0)
            (not (zerop m)))
           (lessp (max-list x) m)))

(prove-lemma number-with-at-least-as-sum (rewrite)
          (implies
           (zerop (number-with-at-least x 2))
           (equal
            (number-with-at-least x 1)
            (sum x))))

(prove-lemma equal-remainder-add1-2 (rewrite)
          (equal
           (equal (remainder (add1 x) 2) (remainder (add1 y) 2))
           (equal (remainder x 2) (remainder y 2))))

(prove-lemma remainder-plus-sum-number-hack (rewrite)
          (implies
           (equal (number-with-at-least x 2) 1)
           (equal
            (remainder (plus (sum x) (number-with-at-least x 1)) 2)
            (remainder (add1 (max-list x)) 2))))

(prove-lemma equal-remainder-add1 (rewrite)
          (equal
           (equal (remainder (add1 x) y) (remainder x y))
           (equal y 1)))

(prove-lemma max-0-means-sum-0 (rewrite)
          (equal
           (equal (sum x) 0)
           (equal (max-list x) 0)))

(prove-lemma max-0-means-all-zero-bitvp (rewrite)
          (implies
           (and
            (equal (max-list x) 0)
            (nat-listp-simple x))
           (all-zero-bitvp x)))

(prove-lemma remainder-add1-2 (rewrite)
          (and
           (equal
            (equal (remainder (add1 x) 2) 0)
            (equal (remainder x 2) 1))
           (equal
            (equal (remainder (add1 x) 2) 1)
            (equal (remainder x 2) 0))))

(prove-lemma remainder-sub1-2 (rewrite)
           (implies
            (not (zerop x))
            (and
             (equal
              (equal (remainder (sub1 x) 2) 0)
              (equal (remainder x 2) 1))
             (equal
              (equal (remainder (sub1 x) 2) 1)
              (equal (remainder x 2) 0)))))

(prove-lemma equal-x-remainder-sub1-x (rewrite)
          (equal
           (equal (remainder (sub1 x) y) x)
           (equal x 0)))

(prove-lemma computer-move-makes-non-green nil
          (implies
           (and
            (green-statep x ws)
            (nat-listp x ws)
```

```
              (listp x)
              (not (zerop ws))
              (not (all-zero-bitvp x)))
             (not (green-statep (computer-move x ws) ws)))
            ((disable nat-to-bv bv-to-nat lessp-number-with-at-least)))

(prove-lemma nat-listp-simplify (rewrite)
             (implies
              (zerop ws)
              (equal (nat-listp x ws)
                     (and
                      (properp x)
                      (all-zero-bitvp x)))))

(prove-lemma computer-move-makes-non-green-rewrite (rewrite)
             (implies
              (and
               (green-statep x ws)
               (nat-listp x ws)
               (not (all-zero-bitvp x)))
              (not (green-statep (computer-move x ws) ws)))
             ((use (computer-move-makes-non-green))
              (disable-theory t)
              (enable nat-listp all-zero-bitvp nat-listp-simplify)
              (enable-theory ground-zero)))

(defn make-properp (x)
  (if (listp x)
      (cons (car x) (make-properp (cdr x)))
    nil))

(prove-lemma properp-make-properp (rewrite)
             (implies
              (properp x)
              (equal (make-properp x) x)))

(prove-lemma replace-value-simplify (rewrite)
             (implies
              (not (member x y))
              (equal (replace-value y x z) (make-properp y))))

(prove-lemma member-make-properp (rewrite)
             (equal
              (member x (make-properp y))
              (member x y)))

(prove-lemma valid-movep-x-x (rewrite)
             (not (valid-movep x x)))

(prove-lemma valid-movep-replace-value (rewrite)
             (implies
              (properp x)
             (equal
              (valid-movep x (replace-value x y z))
              (and
               (member y x)
               (lessp z y)
               (numberp z)))))

(prove-lemma number-with-at-least-max-list (rewrite)
             (implies
              (and
               (equal (number-with-at-least x m) v)
               (lessp 0 v))
              (not (lessp (max-list x) m))))

(prove-lemma valid-movep-match-and-xor (rewrite)
             (implies
              (and
               (nat-listp x ws)
               (bit-vectorp y ws)
               (bit-vectorp z ws))
             (equal
              (valid-movep x
                 (bv-to-nat-list
                  (match-and-xor (nat-to-bv-list x ws) y z)))
              (and
               (match-member y (nat-to-bv-list x ws))
               (lessp
                (bv-to-nat (xor-bitv z
                           (match-member y (nat-to-bv-list x ws))))
                (bv-to-nat (match-member y (nat-to-bv-list x ws)))))))
             ((induct (length x))))

(defn lessp-bv (x y)
  (if (and (listp x) (listp y))
      (if (equal (fix-bit (car x)) (fix-bit (car y)))
```

```
         (lessp-bv (cdr x) (cdr y))
       (equal (car x) 0))
   f))

(prove-lemma lessp-bv-to-nat (rewrite)
            (lessp (bv-to-nat x) (exp 2 (length x))))

(prove-lemma lessp-as-lessp-bv (rewrite)
            (implies
             (equal (length x) (length y))
             (equal
              (lessp (bv-to-nat x) (bv-to-nat y))
              (lessp-bv x y))))

(prove-lemma fix-bitv-highest-bit (rewrite)
            (equal (fix-bitv (highest-bit x)) (highest-bit x)))

(prove-lemma properp-highest-bit (rewrite)
            (properp (highest-bit x)))

(prove-lemma bit-vectorp-fix-bitv (rewrite)
            (equal
             (bit-vectorp (fix-bitv x) s)
             (equal (length x) (fix s))))

(prove-lemma lessp-bv-xor-bitv (rewrite)
            (implies
             (equal (length x) (length y))
             (equal
              (lessp-bv (xor-bitv x y) y)
              (not (all-zero-bitvp (and-bitv y (highest-bit x)))))))

(prove-lemma length-match-member-nat-to-bv-list (rewrite)
            (equal
             (length (match-member a (nat-to-bv-list x ws)))
             (if (match-member a (nat-to-bv-list x ws))
                 (fix ws)
               0)))

(prove-lemma bit-vectorsp-remove-highest-bits2 (rewrite)
            (implies
             (bit-vectorsp x s1)
             (equal
              (bit-vectorsp (remove-highest-bits x) s2)
              (or
               (equal (add1 s2) s1)
               (not (listp x))))))

(prove-lemma match-member-high-bit-xor-bvs-helper nil
            (implies
             (bit-vectorsp x ws)
            (iff
             (match-member (highest-bit (xor-bvs x)) x)
             (not (all-zero-bitvp (xor-bvs x)))))
            ((induct (highest-bits-induct x ws))))

(prove-lemma match-member-high-bit-xor-bvs (rewrite)
            (iff
             (match-member
              (highest-bit (xor-bvs (nat-to-bv-list y ws)))
              (nat-to-bv-list y ws))
             (not (all-zero-bitvp (xor-bvs (nat-to-bv-list y ws)))))
            ((use (match-member-high-bit-xor-bvs-helper
                   (x (nat-to-bv-list y ws))))
             (disable-theory t)
             (enable-theory ground-zero)
             (enable bit-vectorsp-nat-to-bv-list-better)))

(prove-lemma length-match-member (rewrite)
            (implies
             (match-member a (nat-to-bv-list x ws))
             (equal
              (length (match-member a (nat-to-bv-list x ws)))
              (fix ws))))

(prove-lemma all-zero-bitvp-and-match-member (rewrite)
            (implies
             (bit-vectorsp b (length a))
            (equal
             (all-zero-bitvp (and-bitv a (match-member a b)))
             (not (match-member a b))))
            ((induct (match-member a b))))

(prove-lemma valid-movep-computer-move-helper nil
          (implies
           (and
            (nat-listp x ws)
```

```
           (not (all-zero-bitvp x))
           (not (zerop ws))
           (listp x))
          (valid-movep x (computer-move x ws)))
         ((disable all-zero-bitvp nat-to-bv bv-to-nat-list match-and-xor)))

;;; PART OF SPECIFICATION
(prove-lemma valid-movep-computer-move (rewrite)
         (implies
          (and
           (nat-listp x ws)
           (not (all-zero-bitvp x)))
          (valid-movep x (computer-move x ws)))
         ((use (valid-movep-computer-move-helper))
          (disable-theory t)
          (enable-theory ground-zero)
          (enable nat-listp-simplify all-zero-bitvp)))

(prove-lemma nthcdr-cdr (rewrite)
           (equal
            (nthcdr n (cdr x))
            (cdr (nthcdr n x))))

(prove-lemma make-list-from-append (rewrite)
           (equal
            (make-list-from n (append a b))
            (if (lessp (length a) n)
                (append a (make-list-from (difference n (length a)) b))
              (make-list-from n a))))

(prove-lemma make-list-from-simplify-better (rewrite)
           (implies
            (equal n (length x))
            (equal (make-list-from n x) (make-properp x))))

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

(prove-lemma cdr-nthcdr-cons (rewrite)
           (equal
            (cdr (nthcdr n (cons a b)))
            (nthcdr n b)))

(prove-lemma equal-append-2 (rewrite)
           (equal
            (equal x (append a b))
            (and
             (not (lessp (length x) (length a)))
             (equal (make-list-from (length a) x) (make-properp a))
             (equal (nthcdr (length a) x) b)))
           ((induct (double-cdr-induct x a))
            (disable length)))

(prove-lemma member-cons-all-valid-moves-helper1 (rewrite)
           (implies
            (listp a)
           (equal
            (member (cons x y) (all-valid-moves-helper a b c d))
            (and
             (equal x (car a))
             (member y (all-valid-moves-helper (cdr a) b c d))))))

(prove-lemma member-all-valid-moves-means-prefix (rewrite)
           (implies
            (properp a)
            (implies
             (member x (all-valid-moves-helper a b c d))
             (equal (make-list-from (length a) x) a))))

(prove-lemma equal-nthcdr-cons (rewrite)
           (implies
            (not (lessp n (length x)))
            (not (equal (nthcdr n x) (cons a b)))))

(prove-lemma lessp-length-simple-member-all-valid-moves (rewrite)
           (implies
            (not (lessp (length a) (length x)))
            (not (member x (all-valid-moves-helper a b c d)))))

(prove-lemma nth-cons (rewrite)
           (equal (nth n (cons a b))
                  (if (zerop n) a (nth (sub1 n) b))))

(prove-lemma nth-1 (rewrite)
           (equal (nth 1 x) (cadr x)))
```

```
(prove-lemma get-as-nth (rewrite)
            (equal (get n x) (nth n x)))

(prove-lemma equal-cons-make-properp (rewrite)
            (equal
             (equal (cons a b) (make-properp x))
             (and
              (listp x)
              (equal a (car x))
              (equal b (make-properp (cdr x))))))

(prove-lemma nthcdr-cons-make-list-from-hack (rewrite)
            (equal
             (nthcdr n (cons a (make-list-from n z)))
             (if (zerop n) (list a) (list (nth (sub1 n) z)))))

(prove-lemma lessp-sub1-as-equal (rewrite)
             (implies
              (lessp a b)
              (equal (lessp a (sub1 b)) (not (equal (add1 a) b)))))

(prove-lemma equal-nthcdr-cons-better (rewrite)
            (equal
             (equal (nthcdr n x) (cons a b))
             (and
              (lessp n (length x))
              (equal (nth n x) a)
              (equal (nthcdr (add1 n) x) b))))

(prove-lemma member-all-valid-moves-helper (rewrite)
            (implies
             (and
              (nat-listp-simple a)
              (nat-listp-simple d)
              (numberp b)
              (numberp c))
            (equal
             (member x (all-valid-moves-helper a b c d))
             (and
              (equal (make-list-from (length a) x) (make-properp a))
              (or
               (and
                (lessp (nth (length a) x) b)
                (numberp (nth (length a) x))
                (equal (nthcdr (add1 (length a)) x) d))
               (and
                (equal (nth (length a) x) c)
                (valid-movep d (cdr (nthcdr (length a) x)))))))))

(prove-lemma member-all-valid-moves (rewrite)
            (implies
             (nat-listp-simple x)
            (equal
             (member y (all-valid-moves x))
             (valid-movep x y)))
            ((enable all-valid-moves)))

(prove-lemma valid-movep-and-makes-nongreen-means (rewrite)
            (implies
             (and
              (member x y)
              (not (green-statep x ws)))
             (non-green-in-list y ws))
            ((disable green-statep)))

(prove-lemma green-means-non-green-in-valid-moves (rewrite)
            (implies
             (and
              (nat-listp s ws)
              (not (all-zero-bitvp s))
              (green-statep s ws))
             (non-green-in-list (all-valid-moves s) ws))
            ((use (valid-movep-and-makes-nongreen-means
                   (x (computer-move s ws)) (y (all-valid-moves s))))
             (disable green-statep  computer-move)))

(prove-lemma green-in-list-all-valid-moves (rewrite)
            (implies
             (and
              (nat-listp s ws)
              (not (all-zero-bitvp s)))
             (iff
              (non-green-in-list (all-valid-moves s) ws)
              (green-statep s ws)))
            ((disable green-statep)))

(prove-lemma sum-when-all-zero (rewrite)
```

```
            (implies
             (all-zero-bitvp x)
             (equal (sum x) 0)))

(prove-lemma green-statep-all-zero-bitvp (rewrite)
            (implies
             (all-zero-bitvp x)
             (green-statep x s)))

(prove-lemma wsp-green-state-proof nil
            (implies
             (and
              (or
               (and flag (nat-listp s wordsize))
               (and (not flag) (nat-listp-listp s wordsize)))
              (listp s))
             (iff
              (wsp s flag)
              (if flag
                  (green-statep s wordsize)
                (non-green-in-list s wordsize))))
            ((disable green-statep)))


(prove-lemma wsp-green-state (rewrite)
            (implies
             (nat-listp s wordsize)
             (iff
              (wsp s t)
              (green-statep s wordsize)))
            ((use (wsp-green-state-proof (flag t)))
             (disable green-statep)))

(prove-lemma nat-listp-replace-value (rewrite)
            (implies
             (and
              (nat-listp s ws)
              (lessp new (exp 2 ws))
              (numberp new))
             (nat-listp (replace-value s y new) ws)))

(prove-lemma nat-listp-bv-to-nat-list (rewrite)
            (implies
             (bit-vectorsp x s)
             (nat-listp (bv-to-nat-list x) s)))

(prove-lemma nat-listp-smart-move nil
            (implies
             (and
              (nat-listp s ws)
              (not (zerop ws)))
             (nat-listp (smart-move s ws) ws))
            ((disable-theory t)
             (enable-theory ground-zero naturals)
             (enable smart-move bit-vectorsp-nat-to-bv-list-better
                     nat-listp-replace-value
                     nat-listp-bv-to-nat-list
                     lessp-exp-simple lessp-remainder-x-exp-x)
             (use (bit-vectorsp-match-and-xor
                   (size ws)
                   (x (nat-to-bv-list s ws))
                   (y (highest-bit (xor-bvs (nat-to-bv-list s ws))))
                   (z (xor-bvs (nat-to-bv-list s ws)))))))

(prove-lemma all-zero-bitvp-max-list (rewrite)
            (implies
             (all-zero-bitvp s)
             (equal (max-list s) 0)))

(prove-lemma replace-value-x-x (rewrite)
            (implies
             (properp x)
             (equal (replace-value x y y) x)))

(prove-lemma smart-move-small-ws (rewrite)
            (implies
             (and
              (nat-listp s ws)
              (zerop ws))
             (equal (smart-move s ws) s)))

(prove-lemma lessp-max-list-from-nat-listp (rewrite)
            (implies
             (nat-listp s ws)
             (lessp (max-list s) (exp 2 ws))))
```

```
(prove-lemma nat-listp-computer-move (rewrite)
            (implies
             (nat-listp s ws)
             (nat-listp (computer-move s ws) ws))
            ((use (nat-listp-smart-move))
             (disable-theory t)
             (enable-theory ground-zero)
             (enable lessp-max-list-from-nat-listp
                    smart-move-small-ws computer-move
                    nat-listp-replace-value)))

;; PART OF SPECIFICATION
(prove-lemma computer-move-works (rewrite)
            (implies
             (and
              (nat-listp state ws)
              (not (all-zero-bitvp state))
              (wsp state t))
             (not (wsp (computer-move state ws) t)))
            ((disable computer-move green-statep
                     nat-listp-computer-move)
             (use (nat-listp-computer-move (s state)))))


(defn nim-piton-ctrl-stk-requirement ()
  25)

(defn nim-piton-temp-stk-requirement ()
  3)

(defn computer-move-implemented-input-conditionp (p0)
 (and
  (listp (p-ctrl-stk p0))
  (lessp 7 (p-word-size p0))
  (lessp 1 (p-word-size p0))  ; useful to prover, but subsumed

;; there is some room on the stacks

  (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))

;; the third thing on the stack is an address to the state array
  (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
  (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
  (listp (untag (top (cddr (p-temp-stk p0)))))
  (equal (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))

;; the second thing on the stack is the length of the state
  (equal (car (top (cdr (p-temp-stk p0)))) 'nat)
  (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
  (equal (length (array (car (untag (top (cddr (p-temp-stk p0)))))
                    (p-data-segment p0)))
        (untag (top (cdr (p-temp-stk p0)))))
  (lessp (untag (top (cdr (p-temp-stk p0)))) (exp 2 (p-word-size p0)))
  (not (zerop (untag (top (cdr (p-temp-stk p0))))))

;; the top thing on the stack is a pointer to an array
;; that is the same size as the state array but distinct
  (equal (car (top (p-temp-stk p0))) 'addr)
  (equal (cddr (top (p-temp-stk p0))) nil)
  (listp (untag (top (p-temp-stk p0))))
  (equal (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)))))
  (not (equal (car (untag (top (p-temp-stk p0))))
             (car (untag (top (cddr (p-temp-stk p0)))))))

;; at least one pile has one match
  (not (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.
(defn 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)))

(disable computer-move-program)
(disable *1*computer-move-program)

(prove-lemma car-xor-bvs-program (rewrite)
             (equal (car (xor-bvs-program)) 'xor-bvs)
             ((enable xor-bvs-program)))

(prove-lemma car-push-1-vector-program (rewrite)
             (equal (car (push-1-vector-program word-size))
                    'push-1-vector)
             ((enable push-1-vector-program)))

(prove-lemma car-nat-to-bv-program (rewrite)
             (equal (car (nat-to-bv-program)) 'nat-to-bv)
             ((enable nat-to-bv-program)))

(prove-lemma car-bv-to-nat-program (rewrite)
             (equal (car (bv-to-nat-program)) 'bv-to-nat)
             ((enable bv-to-nat-program)))

(prove-lemma car-number-with-at-least-program (rewrite)
             (equal (car (number-with-at-least-program))
                    'number-with-at-least)
             ((enable number-with-at-least-program)))

(prove-lemma car-highest-bit-program (rewrite)
             (equal (car (highest-bit-program)) 'highest-bit)
             ((enable highest-bit-program)))

(prove-lemma car-match-and-xor-program (rewrite)
             (equal (car (match-and-xor-program)) 'match-and-xor)
             ((enable match-and-xor-program)))

(prove-lemma car-nat-to-bv-list-program (rewrite)
             (equal (car (nat-to-bv-list-program))
                    'nat-to-bv-list)
             ((enable nat-to-bv-list-program)))

(prove-lemma car-bv-to-nat-list-program (rewrite)
             (equal (car (bv-to-nat-list-program)) 'bv-to-nat-list)
             ((enable bv-to-nat-list-program)))

(prove-lemma car-max-nat-program (rewrite)
             (equal (car (max-nat-program)) 'max-nat)
             ((enable max-nat-program)))

(prove-lemma car-replace-value-program (rewrite)
             (equal (car (replace-value-program)) 'replace-value)
             ((enable replace-value-program)))

(prove-lemma car-smart-move-program (rewrite)
             (equal (car (smart-move-program)) 'smart-move)
             ((enable smart-move-program)))

(prove-lemma car-delay-program (rewrite)
             (equal (car (delay-program)) 'delay))

(disable delay-program)

(prove-lemma car-computer-move-program (rewrite)
             (equal (car (computer-move-program)) 'computer-move)
             ((enable computer-move-program)))

(prove-lemma equal-untag-array-tag-array-x-x (rewrite)
             (equal
              (equal (untag-array (tag-array l x)) x)
              (properp x)))
```

```
(prove-lemma properp-replace-value (rewrite)
           (implies
            (properp x)
            (properp (replace-value x y z))))

(prove-lemma properp-bv-to-nat-list (rewrite)
           (properp (bv-to-nat-list x)))

(prove-lemma properp-nat-to-bv-list (rewrite)
           (properp (nat-to-bv-list x ws)))

(prove-lemma properp-computer-move (rewrite)
           (implies
            (properp x)
            (properp (computer-move x s)))
           ((disable-theory t)
            (enable-theory ground-zero)
            (enable properp-replace-value properp-bv-to-nat-list
                   smart-move computer-move)))

(prove-lemma properp-untag-array (rewrite)
           (properp (untag-array x)))

(prove-lemma properp-tag-array (rewrite)
           (properp (tag-array l x)))

(prove-lemma computer-move-implemented (rewrite)
           (implies
            (and
             (equal p0 (p-state
                       pc
                       ctrl-stk
                       (cons wa (cons np (cons s temp-stk)))
                       (append (cm-prog word-size) prog-segment)
                       data-segment
                       max-ctrl-stk-size
                       max-temp-stk-size
                       word-size
                       'run))
             (equal (p-current-instruction p0)
                   '(call computer-move))
             (computer-move-implemented-input-conditionp p0))
            (let ((result
                  (p p0 (computer-move-clock
                        (untag-array (array (car (untag s)) data-segment))
                        word-size))))
              (and
               (equal (p-pc result) (add1-addr pc))
               (equal (p-psw result) 'run)
               (equal (untag-array
                      (array (car (untag s)) (p-data-segment result)))
                     (computer-move
                      (untag-array (array (car (untag s)) data-segment))
                      word-size)))))
           ((disable computer-move computer-move-clock
                    p-current-instruction
                    lessp-max-list max-list
                    all-zero-bitvp sum member-of-natlist-means
                    lessp-sub1-x-y-crock all-zero-bitvp-max-list
                    max-list-not-too-big)))

#—
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)))

—#

(prove-lemma nim-piton-space-reasonable (rewrite)
           (not (lessp 1000 (plus (nim-piton-ctrl-stk-requirement)
                                (nim-piton-temp-stk-requirement)))))

;; bind up defns for presentation purposes
(defn good-non-empty-nim-statep (state ws)
  (and
```

```
  (nat-listp state ws)
  (not (all-zero-bitvp state))))

(prove-lemma valid-movep-computer-move-better (rewrite)
            (implies
             (good-non-empty-nim-statep state ws)
             (valid-movep state (computer-move state ws)))
            ((use (valid-movep-computer-move (x state)))
             (disable-theory t)
             (enable good-non-empty-nim-statep)
             (enable-theory ground-zero)))

(prove-lemma computer-move-works-better (rewrite)
            (implies
             (and
              (good-non-empty-nim-statep state ws)
              (wsp state t))
             (not (wsp (computer-move state ws) t)))
            ((use (computer-move-works))
             (disable-theory t)
             (enable good-non-empty-nim-statep)
             (enable-theory ground-zero)))

;;; An initial p-state to run the program on a particular NIM state, then
;;; enter an infinite loop.
(defn 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 () (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.)

(prove-lemma cm-prog-fm9001-loadable nil
            (let ((p0 (example2-computer-move-p-state)))
              (and (proper-p-statep p0)
                   (p-loadablep p0 0)
                   (equal (p-word-size p0) 32)))
            ((enable  XOR-BVS-PROGRAM PUSH-1-VECTOR-PROGRAM 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)))

 ))
```

References

[1] Albin, Hunt, and Wilding. Fm9001 fabrication (in preparation). Technical Report ??, CLI, 1992.

[2] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] Charles L. Bouton. Nim, a game with a complete mathematical theory. In *Annals of Mathematics*, volume 3, 1901-02.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[5] Martin Gardner. *Mathematical Puzzles and Diversions*. Simon and Schuster, New York, 1959.

[6] Warren A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989.

[7] Matt Kaufmann. A user's manual for an interactive enhancement to the boyer-moore theorem prover. Technical Report 60, Institute for Computing Science, University of Texas at Austin, Austin, Texas, August 1987.

[8] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):493–518, December 1989. Also published as CLI Technical Report 30.

[9] Matthew Wilding. Proving Matijasevich's lemma with a default arithmetic strategy. *Journal of Automated Reasoning*, 7(3), September 1991.

[10] Matthew Wilding. A verified nim strategy. Internal Note 249, Computational Logic, Inc., November 1991.

[11] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989. Also published as CLI Technical Report 30.

Acknowledgements: I thank J Moore and Bill Bevier for many very valuable suggestions related to this work. Bill Young made a careful reading of a draft of this report and his comments improved it considerably.