A Mechanically Verified Language Implementation

J Strother Moore

Technical Report 30 September, 1988

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author 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 paper briefly describes a programming language, its implementation on a microprocessor via a compiler and link-assembler, and the mechanically checked proof of the correctness of the implementation. The programming language, called Piton, is a high-level assembly language designed for verified applications and as the target language for high-level language compilers. It provides execute-only programs, recursive subroutine call and return, stack based parameter passing, local variables, global variables and arrays, a user-visible stack for intermediate results, and seven abstract data types including integers, data addresses, program addresses and subroutine names. Piton is formally specified by an interpreter written for it in the computational logic of Boyer and Moore. Piton has been implemented on the FM8502, a general purpose microprocessor whose gate-level design has been mechanically proved to implement its machine code interpreter. The FM8502 implementation of Piton is via a function in the Boyer-Moore logic which maps a Piton initial state into an FM8502 binary core image. The compiler and link-assembler are all defined as functions in the logic. The implementation requires approximately 36K bytes and 1,400 lines of prettyprinted source code in the Pure Lisp-like syntax of the logic. The implementation has been mechanically proved correct. In particular, if a Piton state can be run to completion without error, then the final values of all the global data structures can be ascertained from an inspection of an FM8502 core image obtained by running the core image produced by the compiler and link-assembler. Thus, verified Piton programs running on FM8502 can be thought of as having been verified down to the gate level.

1. Introduction

The correctness of mechanically verified software typically rests on several informal assumptions about the correctness of the underlying subsystems. A correctly specified and accurately verified high-level language program may still exhibit incorrect behavior because of bugs in the compiler, assembler, linker, loader, runtime support software or the hardware. But, at least down to the hardware, these other components of the system are programs and hence are mathematical objects whose properties are subject to formal proof. It is the goal of *system verification* to specify and formally verify all of the components of a system down to the hardware level.

For our purposes it is convenient to define as software those components of a system accurately modelled by mathematical logic (as opposed to physics). Even so there is still a question of where to draw the line between software and hardware. What is the lowest level accurately modelled by mathematical logic? The machine code? The microcode? The register-transfer level? The layout and timing diagrams? Models of individual gates? A line must be drawn. The correctness of any physical system ultimately rests on statistically described assumptions about the properties of physical devices.

But because the software is accurately modelled by mathematical logic, and because mathematical proof is the ultimate arbiter of mathematical truth, the most reliable software systems will be those in which all the software components have been proved correct.

The line has been drawn in a variety of places by various verification projects. The vast majority of verification work (as measured by funding levels) addresses ‘‘design proofs,’’ an activity in which the line is drawn somewhere *above* the highest level executable code in the system. Some verification work addresses ‘‘code proofs,’’ where traditionally the line has been drawn at the definition of a high-level programming language like Gypsy [6, 7, 8], Pascal [19], Fortran [2], and others [20, 5, 13, 17, 4]. There has been some work on compiler verification, notably the work of Polak [15] in which a compiler for a Pascal subset is verified. Finally, there has been some recent work closer to the bottom of the system stack. For example, Gordon [9] and Hunt [10] draw the line essentially at the register-transfer level and offer mechanically certified designs for digital hardware.

But the research underlying the construction of the first fully verified system must address more than just the verification of the individual components. The components of a system are built on top of each other. To verify a system one must be able to *stack* verified components. That is, one must be able to use what was proved about one level to construct the proof of the next. This paper reports on the successful implementation and proof of a relatively high level language on top of a microprocessor for which we have a verified gate-level design. By choosing such a lower-level machine we forced the language implementation to face issues (such as link-assembling) not generally faced in previously reported work on verified implementations.

This paper is a brief summary of a much longer report ([12]) in which we

- formally define an assembly level programming language,
- demonstrate its utility for programming and for verification purposes,
- implement it on the FM8502 microprocessor via a compiler and link-assembler written as functions in the computational logic of Boyer and Moore [1, 3],
- state formally the correctness of the implementation, and
- describe a mechanically checked proof of the correctness of the implementation.

We believe that this is the most complete and comprehensive language implementation proof to date.

In this paper we briefly touch on each of the issues above. Space prevents a thorough treatment. However, the curious reader is urged to see [12] where the details are spelled out both formally and informally. The purpose of the present paper is to give the reader enough information to determine the accuracy of our claim that we have verified a compiler and link-assembler— or more precisely a ‘‘cross-compiler’’ and ‘‘cross link-assembler’’ since in our case those programs are not executed by the target machine.

In the next section we informally describe Piton and its formalization. We illustrate its power by writing a big number addition routine in it. In subsequent sections we sketch the FM8502, state and explain the correctness result for FM8502 Piton, describe the compiler and link-assembler, and sketch the proof of correctness. It is our intention that there is nothing particularly new or startling about Piton, FM8502 or the compiler and link-assembler. We hope these components of our system are conventional and exhibit many similarities with other languages, processors, and implementations. What *is* novel here is that we have mechanically proved the correctness of a nontrivial language implementation. The very familiarity of the implementation techniques discussed in this paper is central to our argument that we have verified a realistic implementation of a realistic assembly level language.

2. Piton

Piton is a high level stack based assembly language. Piton programs operate on an abstract state that includes a user-visible push down stack and global arrays and variables. Piton subroutines may have local variables and may be recursive. Piton supports seven different atomic data types: integers, natural numbers, Booleans, bit vectors, data addresses, program addresses, and subroutine names. Variables and arrays may contain arbitrary atomic objects. Perhaps the most significant departure from conventional assembly-level languages is the deliberate decision that Piton programs cannot overwrite program space. This makes the verification of Piton programs more tractible.

Figure 1 lists the Piton instructions. The language as currently defined provides 65 instructions. We do not give the semantics of the instructions and offer the figure only as an indication of the complexity of the language. The annotated example programs below may help the reader guess the meaning of some instructions. The curious reader should see [12]. The language includes such instructions as **`POP-CALL`**, which pops a subroutine name off the stack and calls it, and **`PUSHJ`** and **`POPJ`** (‘‘Push Jump’’ and ‘‘Pop Jump’’) which enable direct manipulation of the Piton program counter within the current subroutine.

Consider the **`ADD-NAT`** instruction. When **`ADD-NAT`** is executed, there must be two natural numbers on top of the stack and their sum must be representable in the word size of the Piton computation. If this *precondition* is met, the *effect* of the **`ADD-NAT`** instruction is to pop the two naturals off the stack, push their sum, and increment the Piton program counter by one. Each instruction in Figure 1 is characterized by a precondition and an effect. For example, the precondition of a **`POP-CALL`** includes the restriction that the object on top of the stack is a subroutine name. The precondition on **`POPJ`** is that object on top of the stack is a legal program counter in the current subroutine. If the precondition of the instruction to be executed is false then we say the computation is *erroneous*.

Most Piton instructions include among their preconditions requirements on the types of their operands. But there are no Piton primitives for determining the type of an object. The Piton programmer must *know* the type of each data object manipulated.

The semantics of Piton programs is defined with respect to a nine-component *Piton state* or *p-state* that contains the system of programs, certain control information, the push down stack, the global data, and various resource limits (e.g., the word size). Part of the control information is the *program status word* or *psw* which is normally **`RUN`** but is **`HALT`** when the computation has terminated normally and is some error

Figure 1: Piton Instructions

Control	Integers	Natural Numbers
CALL	**ADD-INT**	**ADD-NAT**
JUMP	**ADD-INT-WITH-CARRY**	**ADD-NAT-WITH-CARRY**
JUMP-CASE	**ADD1-INT**	**ADD1-NAT**
NO-OP	**EQ**	**DIV2-NAT**
RET	**INT-TO-NAT**	**EQ**
TEST-BITV-AND-JUMP	**LT-INT**	**LT-NAT**
TEST-BOOL-AND-JUMP	**NEG-INT**	**MULT2-NAT**
TEST-INT-AND-JUMP	**SUB-INT**	**MULT2-NAT-WITH-CARRY-OUT**
TEST-NAT-AND-JUMP	**SUB-INT-WITH-CARRY**	**SUB-NAT**
	SUB1-INT	**SUB-NAT-WITH-CARRY**
		SUB1-NAT

Variables	Booleans	Bit Vectors
LOCN	**AND-BOOL**	**AND-BITV**
POP-GLOBAL	**EQ**	**EQ**
POP-LOCAL	**NOT-BOOL**	**LSH-BITV**
POP-LOCN	**OR-BOOL**	**NOT-BITV**
PUSH-GLOBAL		**OR-BITV**
PUSH-LOCAL		**RSH-BITV**
SET-GLOBAL		**XOR-BITV**
SET-LOCAL		

Stack	Data Addresses	Subroutines
DEPOSIT-TEMP-STK	**ADD-ADDR**	**EQ**
FETCH-TEMP-STK	**DEPOSIT**	**POP-CALL**
POP	**EQ**	
POP*	**FETCH**	
POPN	**LT-ADDR**	Program Addresses
PUSH-CONSTANT	**SUB-ADDR**	
PUSH-TEMP-STK-INDEX		**EQ**
		POPJ
Resources		**PUSHJ**
JUMP-IF-TEMP-STK-EMPTY		
JUMP-IF-TEMP-STK-FULL		
PUSH-CTRL-STK-FREE-SIZE		
PUSH-TEMP-STK-FREE-SIZE		

message if a precondition has been violated. We illustrate a p-state in Figure 3.

The semantics is given via a formally defined interpreter expressed as a recursive function in the Boyer-Moore logic. The function, named **P**, takes two arguments, a p-state, **p**, and a natural number **n**. The value of **P** applied to **p** and **n**, which we write in the Lisp-like prefix syntax **(P p n)**, is the state obtained by starting at **p** and executing **n** Piton instructions (or until the psw is no longer **RUN**).

Below we exhibit a simple Piton program called **MAIN**. **MAIN** is written as a list constant in the Boyer-Moore logic. The two **NIL**s indicate that the program has no formal parameters and no temporary variables. The program has six instructions.

```
(MAIN NIL NIL
      (PUSH-CONSTANT (ADDR (BNA . 0)))
      (PUSH-CONSTANT (ADDR (BNB . 0)))
      (PUSH-GLOBAL N)
      (CALL BIG-ADD)
      (POP-GLOBAL C)
      (RET))
```

The first instruction pushes the tagged data **ADDR**ess constant **(BNA . 0)** onto the stack. This object addresses the 0^{th} position of the global array named **BNA**. The second instruction pushes the data address **(BNB . 0)**. The third instruction pushes the value of the global variable **N**. The fourth instruction calls the user-defined Piton subroutine **BIG-ADD**. **BIG-ADD** is shown in Figure 2. The call of **BIG-ADD** will pop three objects off the stack (to use as actuals) and will return one object by pushing it onto the stack. The fifth instruction of **MAIN** pops **BIG-ADD**'s answer and stores it into the global variable **C**. The last instruction returns to the caller of **MAIN**, or if **MAIN** was the top-level entry into the Piton system (as it will be in our example), halts the Piton machine.

Figure 2: A Piton Program for Big Number Addition

```
(BIG-ADD    (A B N)                          ; Formal parameters
            NIL                              ; Temporary variables
                                             ; Body
            (PUSH-CONSTANT (BOOL F))         ; Push the input carry flag for
                                             ; the first ADD-NAT-WITH-CARRY
            (PUSH-LOCAL A)                   ; Push the address A

   (DL LOOP ()                               ; This is the top level loop.
                                             ; Every time we get here the carry
                                             ; flag from the last addition and
                                             ; the current value of A will be
                                             ; on the stack.
            (FETCH))                         ; Fetch next digit from A
            (PUSH-LOCAL B)                   ; Push the address B
            (FETCH)                          ; Fetch next digit from B
            (ADD-NAT-WITH-CARRY)             ; Add the two digits and flag
            (PUSH-LOCAL A)                   ; Deposit the sum digit in A
            (DEPOSIT)                        ; (but leave carry flag)
            (PUSH-LOCAL N)                   ; Decrement N by 1
            (SUB1-NAT)
            (SET-LOCAL N)                    ; (but leave N on the stack)
            (TEST-NAT-AND-JUMP ZERO DONE)    ; If N=0, go to DONE
            (PUSH-LOCAL B)                   ; Increment B by 1
            (PUSH-CONSTANT (NAT 1))
            (ADD-ADDR)
            (POP-LOCAL B)
            (PUSH-LOCAL A)                   ; Increment A by 1
            (PUSH-CONSTANT (NAT 1))
            (ADD-ADDR)
            (SET-LOCAL A)                    ; (but leave A on the stack)
            (JUMP LOOP)                      ; goto LOOP
   (DL DONE ()
            (RET)))                          ; Exit.
```

Figure 2 shows a Piton program for adding together two base 2^w big numbers, where w is the word size of the Piton computation. When called, the formal parameters **A** and **B** must be bound to data addresses and **N** must be bound to a natural number n. The two addresses should point to the beginning of two ‘‘big number arrays’’ of length n. A ‘‘big number array’’ is an application-specific concept, here defined to mean a Piton array of natural numbers. The intended interpretation of such an array is that it represents the natural number whose base 2^w representation is the sequence of ‘‘digits’’ in the array, with the least significant digit at position 0. **BIG-ADD** computes the sum of the two big numbers. It destructively overwrites the first array and leaves on the stack a Boolean flag indicating whether the sum ‘‘carried out’’ of the input array.

Suppose we wished to use **MAIN** to solve the big number version of

$$\begin{array}{r l} & 786{,}433{,}689{,}351{,}873{,}913{,}098{,}236{,}738 \\ + & \underline{141{,}915{,}430{,}937{,}733{,}100{,}148{,}932{,}872} \\ & ? \end{array}$$

In base 2^{32} (which is 4,294,967,296) these two naturals can be represented by the following big numbers of length 4:

```
'((NAT 246838082) (NAT 3116233281) (NAT 42632655) (NAT 0))
```

and

```
'((NAT 3579363592) (NAT 3979696680) (NAT 7693250) (NAT 0)).
```

A suitable Piton initial state for adding together these two big numbers is shown in Figure 3. The elipses stand for the text previously shown in the definitions of **MAIN** and **BIG-ADD**.

Figure 3: An Initial Piton State for Big Number Addition

```
(P-STATE '(PC (MAIN . 0))                          ; (1) program counter
         '((NIL (PC (MAIN . 0))))                  ; (2) control stack
         NIL                                       ; (3) temporary stack

         '((MAIN NIL NIL                           ; (4) program segment
                 (PUSH-CONSTANT (ADDR (BNA . 0)))
                 ...
                 (RET))
           (BIG-ADD (A B N) NIL
                 (PUSH-CONSTANT (BOOL F))
                 ...
                 (JUMP LOOP)
             (DL DONE NIL (RET))))

         '((BNA  (NAT 246838082)                   ; (5) data segment
                 (NAT 3116233281)
                 (NAT 42632655)
                 (NAT 0))
           (BNB  (NAT 3579363592)
                 (NAT 3979696680)
                 (NAT 7693250)
                 (NAT 0))
           (N    (NAT 4))
           (C    (NAT 0)))

         10                                        ; (6) max ctrl stk size
         8                                         ; (7) max temp stk size
         32                                        ; (8) word size
         'RUN)                                     ; (9) psw
```

The nine fields of the p-state in Figure 3 are enumerated and named in the comments of the figure. We discuss each field in turn. Field (1) is the *program counter*. Note that it is a tagged address. The **PC** tag indicates that it is an address into the program segment. The pair **(MAIN . 0)** is an address, pointing to the 0^{th} instruction in the **MAIN** program. Field (2) is the *control stack*. In this example it contains only one frame and so is of the form **'((bindings return-pc))**. Since the current program counter is in **MAIN**, the single frame on the control stack describes the invocation of **MAIN**. Since **MAIN** has no local variables, the frame has the empty list, **`NIL`**, as the local variable bindings. Since there is only one frame on the stack, it describes the top-level entry into Piton and hence the return program counter is completely irrelevant. If control is ever ‘‘returned’’ from this invocation of **`MAIN`** the Piton machine will halt rather than ‘‘return control’’ outside of Piton. However, despite the fact that the initial return program counter is irrelevant we insist that it be a legal program counter and so in this example we chose **`(PC (MAIN . 0))`**. Field (3) is the *temporary stack*. In this example it is empty. Field (4) is the *program segment*. It contains two programs, **`MAIN`** and **`BIG-ADD`**. Field (5) is the *data segment*. It contains four ‘‘global arrays’’ named, respectively, **`BNA`**, **`BNB`**, **`N`** and **`C`**. **`BNA`** and **`BNB`** are both arrays of length four. **`N`** and **`C`** are each arrays of length one. We think of **`N`** and **`C`** simply as global variables. The **`BNA`** array contains the first of the two big numbers we wish to add, namely **`((NAT 246838082) (NAT 3116233281) (NAT 42632655) (NAT 0))`**. The **`BNB`** array contains the second big number. **`N`** contains the (tagged) length of the two arrays. **`C`** contains the (tagged) natural number 0; the initial value of **`C`** is irrelevant in this system of programs. Fields (6)-(8) are, respectively, the *maximum control stack size*, 10, the *maximum temporary stack size*, 8, and the *word size*, 32. The stack sizes declared in this example are unusually small but sufficient for the computation described. Finally, field (9) is the *psw* **`RUN`**.

Let $\mathbf{p_0}$ be the p-state in Figure 3. If one steps this p-state forward 76 (or more) times one arrives at a p-state in which the final psw is **`HALT`**, the final value of **`BNA`** is **`'((NAT 3826201674) (NAT 2800962665) (NAT 50325906) (NAT 0))`**, and the final value of **`C`** is **`(BOOL F)`**. A little arithmetic will confirm that **`BNA`** contains the correct big number sum and that no carry out occurs.

It is possible to specify formally what it means for the above programs to be correct. That is, it is possible to write down a formula about the Piton interpreter, **`P`**, and the list constants above defining **`MAIN`** and **`BIG-ADD`**, that may be paraphrased as follows: if (in some initial Piton state) **`BNA`** and **`BNB`** are big numbers (base 2^{32}) of length $0 < n < 2^{32}$ and **`N`** is a global variable whose value is n then after running the **`MAIN/BIG-ADD`** system,

- the value of the **`BNA`** array is the low order n digits of the correct big number sum,
- the value of the global variable **`C`** is the Boolean flag indicating whether the n+1st digit is 0 or 1,
- no other data structure or global variable is altered,
- no errors arise during the computation, and
- a certain (constructively given) number of instructions are executed.

Furthermore, the formula alluded to above is a theorem and can be proved. Our mechanical theorem prover has checked the proof.

In this paper we are not interested in programming in Piton or proving such programs correct. The foregoing is offered merely as evidence that Piton is a useful programming language with a formal semantics. We are interested in implementing Piton on a lower-level machine. We therefore proceed to describe the host machine.

3. FM8502

The FM8502 is a 32-bit general purpose microprocessor with word addressing. The machine has eight general purpose 32-bit wide registers, numbered 0 through 7. Register 0 serves as the program counter.

The machine has four 1-bit condition code registers, designated ‘‘carry,’’ ‘‘zero,’’ ‘‘overflow,’’ and ‘‘negative.’’

The alu supports the standard operations on 32-bit wide vectors interpreted as twos complement integers, natural numbers and simple bit vectors. It takes as input an opcode, two 32-bit wide operands, and the contents of the four condition code registers and yields as output a 32-bit wide bit vector and four new condition codes.

The instruction word contains a 5-bit *opcode* field, specifying some operation to be performed; an interrupt-enabled bit (currently unused by the processor); a 4-bit *condition code mask*; a 5-bit *operand b* field, logically divided into a 3-bit field specifying a register number and a 2-bit field specifying an address mode; and a 5-bit *operand a* field, logically divided into a 3-bit field specifying a register number and a 2-bit field specifying an address mode.

The FM8502 fetch-execute cycle is as follows:

1. Register 0 is used as the program counter. Its contents is treated as a memory address. The value of that address is fetched and decoded into an opcode, condition code mask, and operands b and a, as described above. Register 0 is incremented by one.
2. The pre-decrement computation (defined below) is performed on operand a.
3. The effective address (defined below), e_2, specified by a is determined;
4. The contents, x_2, of e_2 is fetched;
5. The pre-decrement computation is performed on b;
6. The effective address, e_1, specified by b is determined;
7. The contents, x_1, of e_1, is fetched;
8. The operation indicated by opcode is performed by the alu on x_1 and x_2 and the contents of the four condition code registers, yielding an output bit vector, y, and four condition codes;
9. The four condition codes are stored into their respective condition code registers provided the corresponding bit in the condition code mask is set;
10. The output bit vector, y, is stored into address e_1;
11. The post-increment computation (defined below) is performed on a.
12. The post-increment computation is performed on b.

The four address modes are: register, register-indirect, register-indirect with pre-decrement, and register-indirect with post-increment.

The *pre-decrement computation* on a register number and an address mode decrements the contents of the register if the address mode is ‘‘register-indirect with pre-decrement’’ and does nothing otherwise.

The *effective address* specified by a register number and an address mode is the indicated register if the address mode is ‘‘register’’ and is the contents of the indicated register otherwise.

The *post-increment computation* on a register number and an address mode increments the contents of the register if the address mode is ‘‘register-indirect with post-increment’’ and does nothing otherwise.

The opcodes can be divided into two groups: arithmetic/logical opcodes and conditional move opcodes. The arithmetic/logical opcodes are unconditional move, increment, add with carry, add, negate, decrement, subtract with borrow, subtract, rotate right through carry, arithmetic shift right, logical shift right, exclusive or, or, and, and not. The eight conditional move opcodes permit the second operand to be moved into the address specified by the first conditionally on the setting of any single condition code register. A jump or conditional jump can be coded as a move or conditional move into register 0. Left shifting can be performed by adding a quantity to itself.

It should be noted that the only quantities stored in the FM8502's memory and registers are 32-bit wide bit vectors. The FM8502 alu operates on such bit vectors. The machine does not contain integers or natural numbers, despite the fact that it has instructions with mnemonics like ''add'' and ''subtract.'' However, Hunt proves that the bit vectors produced by the alu are correct with respect to the various expected interpretations of the input vectors. For example, the output of the add instruction, if interpreted as a natural number, is the natural number sum of the natural interpretations (in binary notation) of the input, with suitable accounting for the input and output carry flags. However, the same output can also be interpreted as the integer sum of the integer interpretation of the inputs (in twos complement notation), with suitable accounting for the input carry and the output overflow and negative flags.

FM8502 is formalized in the function **FM8502**. The state of the FM8502 machine is a six-tuple called an *m-state* which contains the register file, the four condition code bits, and the memory. The function **FM8502** takes two arguments, an m-state and a natural number n and returns the m-state obtained by executing n machine instructions. FM8502 does not halt or cause errors—every bit vector in its memory can be interpreted as a legitimate instruction.

There is a register transfer level design of FM8502 that has been proved correct. The interested reader should see Warren Hunt's report [10] on ''FM8501.'' Hunt produced and verified FM8502 in an analogous fashion by modifying the FM8501 design to (a) provided word size 32 and (b) implement individual control of the condition code settings. See [12] for the formal definition of FM8502.

4. The Correctness of Piton on FM8502

The implementation of Piton on FM8502 is via what might be called a ''cross-loader'' written as a function, **LOAD**, in the logic. The function takes a Piton state and generates an FM8502 state or ''core image.''

The correctness theorem is a formalization of the following classic commutative diagram:

```
               n  Piton steps
        p0 --------------------> pn
        |                        |
        |                        |
   LOAD |                        | DISPLAY-M-DATA-SEGMENT
        |                        |
        |                        |
        m0 --------------------> mk
               k  FM8502 steps
```

The basic idea is that one has some initial Piton state, $\mathbf{p_0}$, and one wishes to obtain the state produced by running Piton forward **n** steps. However, depending on one's point of view, the abstract Piton machine does not ''really'' exist or is too expensive or inefficient to use. The correctness theorem tells us there is another way: map the Piton state ''down'' to an FM8502 state, run FM8502, and then map the resulting state back ''up.'' However, the situation is much more subtle than suggested by the diagram.

The correctness theorem may be paraphrased as follows. Suppose $\mathbf{p_0}$ is some proper (syntactically well-formed) Piton state that is loadable on FM8502 and has word size 32. Let $\mathbf{p_n}$ be the state obtained by running the Piton machine **n** steps on $\mathbf{p_0}$. Suppose $\mathbf{p_n}$ is not erroneous. Suppose furthermore that the ''type specification'' (see below) for the data segment of $\mathbf{p_n}$ is **ts**. Then it is possible to obtain the data segment of $\mathbf{p_n}$ via FM8502 as follows:

- *Down*. Let the initial FM8502 state, $\mathbf{m_0}$, be **(LOAD** $\mathbf{p_0}$**)**.

- *Across*. Obtain a final FM8502 state, $\mathbf{m_k}$, by running FM8502 **k** steps on $\mathbf{m_0}$, where **k** is a number obtained from $\mathbf{p_0}$ and **n** by the constructive function **FM8502-CLOCK**.
- *Up*. Apply the constructive function **DISPLAY-M-DATA-SEGMENT** to (a) the final FM8502 state, $\mathbf{m_k}$, (b) the final type specification, **ts**, and (c) the link tables (computed by the constructive function **LINK-TABLES** from $\mathbf{p_0}$).

This is formalized as

Theorem. FM8502 Piton is Correct

```
(IMPLIES (AND (PROPER-P-STATEP P0)
              (P-LOADABLEP P0)
              (EQUAL (P-WORD-SIZE P0) 32)
              (EQUAL PN (P P0 N))
              (NOT (ERRORP (P-PSW PN)))
              (EQUAL TS (TYPE-SPECIFICATION (P-DATA-SEGMENT PN))))
         (EQUAL (P-DATA-SEGMENT PN)
                (DISPLAY-M-DATA-SEGMENT
                  (FM8502 (LOAD P0) (FM8502-CLOCK P0 N))
                  TS
                  (LINK-TABLES P0)))).
```

We believe this theorem is merely a formalization of what is usually meant by the informal remark that a programming language is implemented correctly on given hardware base.

We comment on three subtle aspects of this formula. First, unlike the classical commuting diagram, we do not map the final FM8502 state ‘‘up’’ to a full Piton state—there is not enough information in the lower state to recover the final upper state completely. We show how to recover only the global data (arrays and global variables). Second, we concern ourselves only with non-erroneous computations, i.e., non-erroneous final states. Third, we require knowledge of the final ‘‘type specification,’’ i.e., to recover the global data in the final Piton state we must be told the data type of every object in the final Piton state.

The user of FM8502 Piton cannot determine whether an error occurred by inspection of the final core image. Nor can he determine the data type of the Piton object allocated any given memory location. How then can the correctness theorem be used? The answer is that it can be applied to Piton systems that have been *proved* to be non-erroneous and to produce final answers of an indicated type. For example, suppose we use FM8502 to run a loadable core image produced from the **MAIN/BIG-ADD** system. Suppose furthermore that the input conditions on **MAIN/BIG-ADD** are satisfied, i.e., **BNA** and **BNB** are big numbers of length $0<\mathbf{N}<2^{32}$. Then the correctness result for the **MAIN/BIG-ADD** system insures that the final state is non-erroneous and contains a big number (i.e., an array of naturals) in **BNA** and a carry out (i.e., a Boolean) in **C** and does not change the other global data areas. Thus, we can combine the correctness of **MAIN/BIG-ADD** with the correctness of FM8502 Piton to conclude that our interpretation of the final FM8502 core image is the correct big number sum.

5. The Implementation

In this section we sketch the implementation of Piton on FM8502. We first give an example of an FM8502 core image produced by our **LOAD** function. The intention is simply to reassure the reader that we do get down to the bit level eventually. Then we describe the implementation itself in three parts. First we describe how we use use the resources of the FM8502, namely a linearly addressed memory and registers, to represent the stacks of the Piton machine. Second, we show how we translate Piton instructions into FM8502 instructions. We call this *compiling* the Piton programs. In our implementation, the compiler does not produce binary machine code; instead it produces what we call ‘‘i-code,’’ an assembly code for FM8502 containing symbolic program names, labels, and abstract Piton objects. Third, we explain our *link-assembler* which macroexpands i-code instructions into assembly language and then assembles them into their machine code counterparts, constructs various *link tables*, and then uses the link tables to replace symbolic Piton objects by their bit vector counterparts.

5.1 An Example

Recall the program **BIG-ADD** and the initial state illustrated in Figure 3 on page 5.

The result of **LOAD**ing this state is the FM8502 m-state shown in Figure 4.[1] There is absolutely no expectation on our part that this core dump is understandable. We only expect it to impress upon the reader that **LOAD** does indeed produce a core image, despite our early fascination with abstractions such as ''resource representation.''

5.2 Use of the FM8502 Resources

Our implementation of Piton partitions the memory of FM8502 into three parts. The first holds the binary programs and is called the *program segment*. The second holds the data segment of the Piton machine and is called the *user data segment* of the FM8502 memory. The third holds the control and temporary stacks and three words of data used to implement the stack resource instructions. This segment of the FM8502 memory is called the *system data segment*. In addition, the implementation uses six of the eight registers of FM8502 and the four condition code flags.

5.2-A The Registers

We give symbolic names to six of the FM8502's registers. The actual register numbers allocated are 0-5, in the order listed.

- *pc* (program counter).
- *cfp* (control stack frame pointer): determines the extent of the topmost frame on the control stack.
- *csp* (control stack pointer): addresses the top of the control stack.
- *tsp* (temporary stack pointer): addresses the top of the temporary stack.
- *x* and *y*: used as *temporaries* in the machine code implementing Piton instructions: no assumptions are made about the values of these registers when (the machine code for) a Piton instruction begins execution and no promises are offered about the values at the conclusion of execution.

5.2-B The Condition Codes

Recall the four condition code registers of FM8502: carry, overflow, negative, and zero. These can be set by any instruction according to the operation performed by the alu during that instruction. The condition codes are tested by the conditional move instructions.

The implementation of certain Piton instructions, namely the ''test and jump'' family and the ''arithmetic with carry'' family, load and test the condition code registers of FM8502. However, these registers are also treated as temporaries in the machine code.

---

[1] In this paper we use the standard notation for bit vectors, with the least significant digit to the right and the entire string of binary digits prefixed by the letter **B**. In Hunt's formalization of FM8501—to which we actually map—bit vectors are shells constructed from **T** and **F** by the function **BITV** and the empty bit vector **(BTM)**. The least significant bit is the outermost in the nest. For example, the formal term denoted by **B011** is **(BITV T (BITV T (BITV F (BTM))))**. Readers familiar with our logic will recognize **'B011** as a **LITATOM**. To display the bit vectors in this report we simply defined the function in the logic that converts a **BITV** shell object into the appropriate literal atom.

Figure 4: The FM8502 Core Image for Big Number Addition

```
(M-STATE '(B00000000000000000000000000000010 B00000000000000000000000000110011
           B00000000000000000000000000110011 B00000000000000000000000000111110
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000)
         F F F F
         '(B00000000000011111000001001000001 B00000000000011111000000000100010
           B00000000000011111000001001111000 B00000000000000000000000001100001
           B00000000000011111000001001111000 B00000000000000000000000001100101
           B00000000000011111000000010011000 B00000000000000000000000001101001
           B00000000000011111000001001101100 B00000000000011111000001001011000
           B00000000000000000000000000001101 B00000000000011111000000000001000
           B00000000000000000000000000010101 B00000000000011111000000010011000
           B00000000000000000000000001101010 B00000000000011111000000110011011
           B00000000000011111000000000001000 B00000000000000000000000000010010
           B00000000000011111000000001000001 B00000000000011111000000000111010
           B00000000000011111000000000011010 B00000000000011111000001001000001
           B00000000000011111000000000100010 B00000000000011111000001001011011
           B00000000000011111000001001011011 B00000000000011111000001001011011
           B00000000000011111000001001111000 B00000000000000000000000000000000
           B00000000000011111000000010011000 B00000000000000000000000000000000
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000000010011011 B00000000000011111000001001101100
           B00000000000011111000000010011000 B00000000000000000000000000000001
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000000010011011 B00000000000011111000001001101100
           B00000000000011111000000010011011 B00000000000011111000000010111011
           B00000000000010010010000101101011 B00000000000000100010000010000101
           B00000000000000011000000101111000 B00000000000000000000000000000001
           B00000000000011111000001001100100 B00000000000011111000000010011000
           B00000000000000000000000000000000 B00000000000000110000000010000010
           B00000000000011111000001001101100 B00000000000011111000000010011011
           B00000000000011111000000110011011 B00000000000011111000000010011000
           B00000000000000000000000000000010 B00000000000000110000000010000010
           B00000000000011111000001001101100 B00000000000000101000000101101011
           B00000000000011111000000010011000 B00000000000000000000000000000010
           B00000000000000110000000010000010 B00000000000011111000000110001011
           B00000000000011111000010010111011 B00000000000011111000000010011000
           B00000000000000000000000001011100 B00000000000000101100000000000100
           B00000000000011111000000010011000 B00000000000000000000000000000001
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000001001111000 B00000000000000000000000000000001
           B00000000000011111000000010011011 B00000000000000110000000101100100
           B00000000000011111000000010011000 B00000000000000000000000000000001
           B00000000000000110000000010000010 B00000000000011111000000110011011
           B00000000000011111000000010011000 B00000000000000000000000000000000
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000001001111000 B00000000000000000000000000000001
           B00000000000011111000000010011011 B00000000000000110000000101100100
           B00000000000011111000000010011000 B00000000000000000000000000000000
           B00000000000000110000000010000010 B00000000000011111000000110001011
           B00000000000011111000000000001000 B00000000000000000000000000100000
           B00000000000011111000000000001000 B00000000000000000000000001011110
           B00000000000011111000000001000001 B00000000000011111000000000111010
           B00000000000011111000000000011010 B00001110101101100111001101000010
           B10111001101111011111001001000001 B00000010100010101000010111001111
           B00000000000000000000000000000000 B11010101010110001100000100001000
           B11101101001101010101101000101000 B00000000111010101100011110000010
           B00000000000000000000000000000000 B00000000000000000000000000000100
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000110011
           B00000000000000000000000000000010 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000001101011
           B00000000000000000000000001110110 B00000000000000000000000000111110))
```

5.2-C The Program Segment

The program segment of the FM8502 memory is logically divided into separate areas. Each area corresponds to a Piton program. The contents of such an area is the binary machine code obtained by compiling the corresponding Piton program and then link-assembling it. We are more specific later.

5.2-D User Data Segment

The user data segment of the FM8502 machine is isomorphic to the data segment of the Piton machine. The data segment is allocated in the FM8502 memory space immediately above the program segment. Each array in the data segment is allocated a block of words as long as the array. The arrays are laid out successively, each immediately adjacent to its neighbors.

5.2-E The System Data Segment

Immediately above the data segment we allocate the system data segment. This segment contains five ‘‘areas,’’ comparable to the areas (arrays) of the user data segment. In the *control stack area* we represent the Piton control stack, using, in addition, the **`cfp`** and **`csp`** registers. In the *temporary stack area* we represent the Piton temporary stack, using, in addition, the **`tsp`** register. The other three system data areas contain resource bounds used in the implementation of such instructions as **`JUMP-IF-TEMP-STK-EMPTY`**.

The amount of space allocated to each stack is one greater than the maximum stack size for that stack, as specified in the Piton state.

Because of the addressing modes in FM8502 it was decided that the stacks should grow ‘‘downwards’’ (i.e., toward absolute address 0). When a push is performed, the stack pointer is decremented and then the operand is deposited into the memory location indicated by the new stack pointer; when a pop is performed, the contents of the indicated memory location is fetched and then the pointer is incremented. When a stack is empty its stack pointer addresses the high word in its area; the contents of this word is never read or written and hence the allocation of the word is actually a waste of one word.

5.2-F The Temporary Stack

Consider the following Piton temporary stack containing four Piton data objects:

$$
\begin{array}{l}
(obj_1 \\
\ obj_2 \\
\ obj_3 \\
\ obj_4).
\end{array}
$$

obj_1 is the topmost element of the stack. Suppose the maximum temporary stack size is 6.

At the FM8502 level, this temporary stack is represented by a block of 7 words in the system data segment, together with the **`tsp`** register. Suppose, for concreteness, that the temporary stack data area is allocated beginning at absolute address 1000. In Figure 5 we show how the above stack is represented. Of course, the contents of the memory locations are not the Piton objects themselves but bit vectors representing them. We deal with this problem later. A push on the stack above would decrement **`tsp`** and write into address **`1001`**. A pop would fetch from the current **`tsp`** and then increment **`tsp`** to **`1003`**. If the stack is popped four times it is empty and **`tsp`** would be **`1006`**. Because we allocated an extra word to the temporary stack area, the empty stack pointer is still an address into the temporary stack area. Because we will never pop the empty stack, the contents of the extra word is irrelevant.

Figure 5: An FM8502 Temporary Stack

memory		registers	
address	contents	name	contents
1006	*unused*	**tsp**	**1002**
1005	obj_4		
1004	obj_3		
1003	obj_2		
1002	obj_1		
1001	*inactive*		
1000	*inactive*		

5.2-G The Control Stack

The Piton control stack is a stack of frames, each frame containing the bindings for the local variables of the current subroutine and the return program counter. The bindings are in the form of an association list in which each local variable of the invoked subroutine is paired with its current value. At the FM8502 level, each frame contains the values of the local variables (but not the names) and the return program counter; the machine code references the values by position within the frame. Because the FM8502 frames are just blocks of words laid out consecutively in the control stack area, each frame contains an additional word that points to the ‘‘beginning’’ of the previous frame.

We explain the exact layout of the control stack by example. Below we show a Piton control stack containing three frames.

```
(LIST
  (LIST '((X . valx)          ;Frame 1 - current
          (Y . valy)
          (Z . valz))
        ret-pc1)
  (LIST '((A . vala)          ;Frame 2
          (B . valb))
        ret-pc2)
  (LIST '((U . valu)          ;Frame 3
          (V . valv))
        ret-pc3))
```

Suppose the maximum control stack size is 16. Then the stack above is represented as a block of 17 consecutive words in the system data segment, together with two registers, the **csp** register and the **cfp** register.

For the sake of concreteness, suppose the control stack area begins at address 5000. Then the relevant portion of the FM8502 state is shown in Figure 6.

Observe that by incrementing **csp** by i one obtains the address of the value of the i^{th} local variable of the current frame. Observe also that by following the chain of addresses starting in the **cfp** register one can identify each frame of the control stack.

5.2-H Stack Resource Limits

In addition to the two stack areas, the system data segment contains three words used to implement the stack resource instructions. These three words each comprise a named area within the system data segment.

Figure 6: An FM8502 Control Stack

memory address	memory contents	registers name	registers contents
5016	*unused*	**cfp**	**5006**
5015	*ret-pc*$_3$	**csp**	**5003**
5014	**5014**		
5013	*val*$_v$		
5012	*val*$_u$		
5011	*ret-pc*$_2$		
5010	**5014**		
5009	*val*$_b$		
5008	*val*$_a$		
5007	*ret-pc*$_1$		
5006	**5010**		
5005	*val*$_z$		
5004	*val*$_y$		
5003	*val*$_x$		
5002	*inactive*		
5001	*inactive*		
5000	*inactive*		

5.3 Compiling

The compiler scans the program segment of the Piton state and pairs each program name with the assembly code for that program. Observe that compilation of each program is independent of the other programs and of the other components of the Piton state. It is the link phase that worries about references between programs and data.

The assembly code for a Piton program is logically divided into three parts. The first part, called the *prelude*, is executed as part of subroutine **CALL**. The prelude builds the new control stack frame for the invocation and removes the actuals from the temporary stack. The second part, called the *body*, is the translation of the successive instructions in the body of the Piton program. The third part, called the *postlude*, is executed as part of the **RET** instruction and pops the top frame off the control stack, returning control to the indicated pc.

We exhibit a simple compilation below. Consider the following silly Piton program:

```
(DEMO (X Y Z)                   ; Formals X, Y, and Z
      ((A (INT -1))             ; Temporaries A and I
       (I (NAT 2)))
      (PUSH-LOCAL Y)            ; pc 0
      (PUSH-CONSTANT (NAT 4))   ; pc 1
      (ADD-NAT)                 ; pc 2
      (RET))                    ; pc 3
```

The program, named **DEMO**, has three formals and two locals. The locals are initialized to the integer -1 and the natural 2 upon entry. The program adds 4 to the value of the second formal and leaves the result on the stack.

The output of the compiler on this program is shown in Figure 7. The output is in a symbolic, annotated version of FM8502 machine code, called *i-code*. I-code is similar to assembly code in that instructions are written in symbolic form and translate 1:1 into FM8502 machine code. We have printed the compiler's output in a four column format to make its structure clearer. The first column consists entirely of labels.

The second column contains i-code instructions and data.[2] The third column contains the Piton source code used as input to the compiler. The fourth column consists of manually inserted comments that enumerate the successive values of the i-code program counter.

Figure 7: Compiler Output for DEMO

```
label               i-code                  Piton                      pc
                    instruction             instruction

(DEMO
 (DL (DEMO PRELUDE)                         (PRELUDE)
                    (CPUSH_CFP))                                       ; 0
                    (MOVE_CFP_CSP)                                     ; 1
                    (CPUSH_*)                                          ; 2
                    (NAT 2)                                            ; 3
                    (CPUSH_*)                                          ; 4
                    (INT -1)                                           ; 5
                    (CPUSH_<TSP>+)                                     ; 6
                    (CPUSH_<TSP>+)                                     ; 7
                    (CPUSH_<TSP>+)                                     ; 8
 (DL (DEMO . 0)                             (PUSH-LOCAL Y)
                    (MOVE_X_*))                                        ; 9
                    (NAT 1)                                            ;10
                    (ADD_X{N}_CSP)                                     ;11
                    (TPUSH_<X{S}>)                                     ;12
 (DL (DEMO . 1)                             (PUSH-CONSTANT (NAT 4))
                    (TPUSH_*))                                         ;13
                    (NAT 4)                                            ;14
 (DL (DEMO . 2)                             (ADD-NAT)
                    (TPOP_X))                                          ;15
                    (ADD_<TSP>{N}_X{N})                                ;16
 (DL (DEMO . 3)                             (RET)
                    (JUMP_*))                                          ;17
                    (PC (DEMO . 4))                                    ;18
 (DL (DEMO . 4)                             (POSTLUDE)
                    (MOVE_CSP_CFP))                                    ;19
                    (CPOP_CFP)                                         ;20
                    (CPOP_PC))                                         ;21
```

The prelude for the code in Figure 7 consists of instructions **0** through **8**. The prelude is labelled by the label **(DEMO PRELUDE)**, a list object in the logic and guaranteed to be unique among the labels used by the assembler. All labels, of course, are eventually removed by the linker and merely serve as unique entries in the link table.

The prelude builds a new frame on the control stack. The construction of the frame begins before the prelude is entered, when the **CALL** instruction is executed. **CALL** pushes the return program counter onto the control stack and jumps to the prelude of the called subroutine. Recall that the first word in the new frame is the return program counter (which has just been pushed). The prelude builds the rest of the frame by pushing more words onto the control stack.

Below we display the prelude for **DEMO**.

```
                    (CPUSH_CFP)                                        ; 0
```

[2] FM8502 machine code instructions can take ‘‘immediate’’ data from the next word in the instruction stream using the post-increment addressing mode with the program counter register. In our i-code, such double-word instructions have names ending in a ‘‘*****.’’

```
(MOVE_CFP_CSP)                                ; 1
(CPUSH_*)                                     ; 2
(NAT 2)                                       ; 3
(CPUSH_*)                                     ; 4
(INT -1)                                      ; 5
(CPUSH_<TSP>+)                                ; 6
(CPUSH_<TSP>+)                                ; 7
(CPUSH_<TSP>+)                                ; 8
```

At instruction **0** the current frame pointer register, **cfp**, is pushed onto the control stack. That sets the old cfp word of the frame. At instruction **1** the current **csp**, which now points to the old cfp word just pushed, is moved into the **cfp** register. Thus, the **cfp** register points to the old cfp word of the frame under construction. The instruction at **2** is a double-word instruction that pushes the natural number **2** onto the control stack. This is the initial value of the temporary variable **I**. At instruction **4** the initial value of the temporary variable **A** is pushed. The last three instructions of the prelude, **6**-**8**, each pop one thing off the temporary stack and push the result onto the control stack. These instructions move the actuals into the new frame. This completes the construction of the frame. Note that the **csp** register points to the top of the stack and the **cfp** register points to the old cfp word as required.

The first executable Piton instruction in the program, **(PUSH-LOCAL Y)**, is compiled as instructions **9**-**12**. That block of code is labelled in the assembly language by **(DEMO . 0)** which happens to be the Piton program address of the Piton instruction for which the code was generated. Each program address at the Piton level is defined as a label in the i-code. It is via this identification of Piton program address objects with i-code labels that the linker is able to replace each **PC** object by its absolute address.

The instructions for **(PUSH-LOCAL Y)** are

```
(MOVE_X_*)                                    ; 9
(NAT 1)                                       ;10
(ADD_X{N}_CSP)                                ;11
(TPUSH_<X{S}>)                                ;12
```

The basic idea is to fetch a certain element from the top frame and push it onto the temporary stack. The element is the one in the slot for the local variable **Y**, which is in position **1** of the locals. Recall that the locals are numbered from **0** and **X** is thus the **0**th local of the program; the value of **X** is thus at the address indicated by **csp**. The value of **Y** is at the address one greater than **csp**. The code above may be explained as follows: (**9**) move the index, **1**, of the required local variable into the **x** register. (**11**) add the contents of the **csp** register to **x** and store the result in **x**; this is the address of the appropriate slot in the control stack. (**12**) fetch indirect through the **x** register and push the result onto the temporary stack.

In our symbolic code, we use angle brackets, e.g., **<X>**, around a register to indicate register-indirect addressing mode. We use set braces, e.g., **{S}**, to indicate the type of object in the register. Thus, the instruction **ADD_X{N}_CSP** means ‘‘add the contents of **csp** to *the natural number in* **x**’’ and **TPUSH_<X{S}>** means ‘‘indirect through *the system data address in* **x**.’’ We discuss the data type annotations later.

The next instruction in the Piton program is **(PUSH-CONSTANT (NAT 4))** and the code generated is

```
(TPUSH_*)                                     ;13
(NAT 4)                                       ;14
```

This code pushes the natural number 4 onto the temporary stack.

The next Piton instruction is **(ADD-NAT)**, which is supposed to pop two naturals off the temporary stack, add them together, and push the result. The generated code is

```
(TPOP_X)                                      ;15
(ADD_<TSP>{N}_X{N})                           ;16
```

The code pops one thing off the temporary stack into **x**. Then it adds the natural in **x** to the natural fetched indirect through **tsp** (the top item on the stack), and deposits the result indirect through **tsp** (back onto the top of the stack).

The last executable instruction in the Piton code is the return instruction. It compiles into

```
(JUMP_*)                                   ;17
(PC (DEMO . 4))                            ;18
```

This is just an unconditional jump to the label **(DEMO . 4)**, which is where the postlude is located.[3]

The postlude is

```
(MOVE_CSP_CFP)                             ;19
(CPOP_CFP)                                 ;20
(CPOP_PC))                                 ;21
```

The postlude must remove the top frame from the stack, restore the **cfp** register to the value it had at the time of the **CALL**, and restore the program counter to the return pc. The first instruction moves the contents of the **cfp** register into **csp**. This effectively pops all the bindings of this frame and makes the top of the stack be the old cfp word. The next instruction pops the control stack into the **cfp** register, restoring **cfp** and exposing the return pc at the top of the stack. The last instruction pops the control stack into the **pc**, completing the return and removing the last vestige of the now popped frame.

Now consider the following segment of a Piton program **MAIN** that calls **DEMO** on the address of the 25^{th} element of the array **DELTA1**, the natural number 17, and the Boolean value **T**. Suppose the **CALL** instruction is located at program address **(MAIN . 3)**.

```
(PUSH-CONSTANT (ADDR (DELTA1 . 25)))
(PUSH-CONSTANT (NAT 17))
(PUSH-CONSTANT (BOOL T))
(CALL DEMO)
```

The i-code generated for this segment is shown below. We have stripped out the label definitions.

```
(TPUSH_*)                ;Push first actual on temp
(ADDR (DELTA1 . 25))
(TPUSH_*)                ;Push second actual on temp
(NAT 17)
(TPUSH_*)                ;Push third actual on temp
(BOOL T)
(CPUSH_*)                ;Push return pc on ctrl
(PC (MAIN . 4))
(JUMP_*)                 ;Jump to (DEMO PRELUDE)
(PC (DEMO PRELUDE))
```

All addresses, both program and data, are represented in the i-code by *extended data objects*—i.e., either Piton data objects, system data address objects (see below), or i-code labels tagged **PC**. This is true regardless of how the address originated. For example, **(DELTA . 25)** was originally a data object in the source program. The reference to **DEMO** occurred in the **CALL** instruction and has been transformed into the i-code data object **(DEMO PRELUDE)** of type **PC**. The return pc **(MAIN . 4)** above was only implicit in the source program. The linker's job is to replace all these symbolic address objects by absolute addresses.

This brief sketch fails to illustrate many interesting features of our compiler such as the handling of labels,

---

[3]Our compiler has much room for improvement. The instruction at **17** jumps to the next executable instruction, and so could be eliminated. We do not do any such optimizations.

the test and jump instructions, and the implementation of operations that are not primitive on FM8502. The reader is urged to see [12].

5.4 The Link-Assembler

Traditionally, compilers produce relocatable assembly code, which is then turned into relocatable machine code by an ‘‘assembler’’, and then into absolute machine code by a ‘‘linker.’’ In addition, assemblers and linkers must traditionally consider the user’s data declarations and initialization too. We do not follow this paradigm rigidly but the basic concepts are still present in our ‘‘link-assembler.’’

By the time the link-assembler is invoked the first two phases of **`LOAD`** have been carried out: the resource representation phase has produced symbolic descriptions of the registers and the system data segment; the compiler has produced the i-code version of the program segment. The user data segment is symbolically described by the Piton data segment. The job of our ‘‘link-assembler’’ is to replace the symbolic instructions and data objects by concrete bit vectors.

To do so, the link-assembler first builds a collection of ‘‘link tables’’ which indicate where each program, label, system data area and user data area are located in absolute terms. Note that by the time we build the link tables, the three segments of the FM8502 memory are symbolically described by the i-code program segment, the Piton data segment, and the system data segment. All three of these symbolic descriptions have the same form: each is a list of pairs consisting of the name of the program or area and an array listing the contents of the area. Each element of the array is either an (optionally labelled) i-code instruction or an extended data object. Each element can be mapped to a single word in FM8502. Thus, the number of words to be allocated to each area is just the length of the associated array. The absolute location of each name can be determined by summing the lengths of the areas preceding the definition of the name. The absolute location of each label can be similarly determined by counting the number of items in each i-code program preceding the label definition.

Once the link tables have been created, the link-assembler scans each of the three memory segments in turn. Each i-code instruction is replaced by the corresponding FM8502 machine code instruction, by a function which can be thought of as the basic component of an assembler. Each data object is replaced by the corresponding bit vector, using the link tables as appropriate.

We present the link-assembler in three subsections. First, we describe how individual i-code instructions are assembled into FM8502 machine code instructions. Then we discuss how we generate the link tables. Finally, we describe how we transform each of the data objects.

5.4-A The Instruction Assembler

The instruction assembler converts a single i-code instruction into an FM8502 machine code instruction, i.e., a 32-bit wide bit vector. The conversion is done in two steps and explicitly involves an assembly language for FM8502. In essence, each i-code instruction is taken as a ‘‘pseudo-instruction’’ that is mapped first into an assembly instruction and then into a bit vector.

In Figure 8 we illustrate the mapping of i-code instructions to assembly language. There are a total of 87 i-code opcodes. But some distinct i-code opcodes map to the same assembly language instruction. For example, both **`ADD_<TSP>{I}_X{I}`** and **`ADD_<TSP>{N}_X{N}`** map to **`(ADD () (TSP) X)`**. As is suggested by the annotations ‘‘**`{I}`**’’ and ‘‘**`{N}`**’’ in the opcode names, the first instruction deals with integers and the second deals with naturals. As manifested by the fact that they both map to a single instruction, no such distinction exists at the concrete level of FM8502. Why then do we have two different i-code instructions? The answer is that by having two different i-code instructions we can factor our correctness proof into a proof of the compiler and a proof of the link-assembler. That is, we can define the semantics of the two different i-code instructions differently, so that one operates on and produces integers while the other operates on and produces naturals. We can prove the compiler correct with respect to that semantics for i-code. We can then show that under the mapping provided by link-assembling the two operations can be carried out by the same FM8502 instruction. Such factoring would be harder (indeed, impossible in light of issues raised in [12]) if i-code did not distinguish the different data types.

Figure 8: Sample of the Map from I-code to Assembly Code

i-code	assembly code
`ADD_<TSP>{I}_X{I}`	`(ADD () (TSP) X)`
`ADD_<TSP>{N}_X{N}`	`(ADD () (TSP) X)`
`ADD_X{N}_CSP`	`(ADD () X CSP)`
`ADDC_<C>_X{N}_Y{N}`	`(ADDC (C) X Y)`
`ADDC_<V>_X{I}_Y{I}`	`(ADDC (V) X Y)`
`CPOP_CFP`	`(MOVE () CFP (CSP +1))`
`CPOP_PC`	`(MOVE () PC (CSP +1))`
`CPUSH_*`	`(MOVE () (-1 CSP) (PC +1))`
`CPUSH_<TSP>+`	`(MOVE () (-1 CSP) (TSP +1))`
`CPUSH_CFP`	`(MOVE () (-1 CSP) CFP)`
`JUMP_*`	`(MOVE () PC (PC))`
`MOVE_CFP_CSP`	`(MOVE () CFP CSP)`
`MOVE_CSP_CFP`	`(MOVE () CSP CFP)`
`MOVE_X_*`	`(MOVE () X (PC +1))`
`TPOP_X`	`(MOVE () X (TSP +1))`
`TPOP{I}_<ZN>_Y`	`(MOVE (Z N) Y (TSP +1))`
`TPUSH_*`	`(MOVE () (-1 TSP) (PC +1))`
`TPUSH_X`	`(MOVE () (-1 TSP) X)`
`TPUSH_<X{S}>`	`(MOVE () (-1 TSP) (X))`

There are only 69 distinct assembly language instructions used in the i-code mapping table sampled above. These could have been converted into the corresponding bit vectors by hand and listed in the table. However, it was less error-prone to implement a general purpose assembler for FM8502.

The structure of the assembly language should be pretty obvious from Figure 8. Consider the i-code instruction **`(TPUSH_X)`**. This instruction is mapped by the i-code table into the assembly instruction **`(MOVE () (-1 TSP) X)`**. This in turn is assembled into the natural number 1016420, which when converted to a bit vector in standard 32-bit binary notation is

```
B00000000000011111000001001100100.
```

Decoding this vector as an FM8502 instruction yields

```
opcode:           1111
move:                 1
i-bit:                 0
cvnz:                   0000
mode-b:                     10
reg-b:                        011
mode-a:                          00
reg-a:                             100
```

That is, the instruction is an unconditional (**`cvnz: B0000`**) move (**`opcode/move: B11111`**). Operand-a, the source of the move, is register 4 (**`B100`**) in register-direct mode (**`B00`**). Operand-b, the destination of the move, is register 3 (**`B011`**), in register-indirect with pre-decrement mode (**`B10`**). The effect of the instruction is to (a) increment the program counter, register 0, by 1; (b) fetch the contents, **`x`**, of register 4; (c) decrement the contents of register 3 by 1 and store the result in 3; (d) deposit **`x`** at the address contained in register 3. Observe that if register 3 contains a stack pointer and one uses the conventions that the stack pointer points to the topmost element of the stack and stacks grow downward, then this instruction pushes **`x`** onto the stack pointed to by register 3.

Observe that **`ADD_<TSP>{N}_X{N}`**, used in the compilation of the Piton **`ADD-NAT`** instruction, assembles into **`(ADD () (TSP) X)`**. No type checking is done and no provision for detecting carry out is made. Such checks are unnecessary since we are obliged by the correctness result to implement Piton only for non-erroneous computations.

5.4-B The Link Tables

We use four link tables. The first, called the *program link table*, maps each program name to the absolute location of the beginning of the program. The second table, called the *label tables*, maps each each program name to its ‘‘label table.’’ The *label table* for a program name maps the i-code labels to the absolute position of the definition of the label in the i-code program. The third table, called the *user data link table*, maps each global data area name to the absolute location of the beginning of the associated array. The fourth table, called the *system data link table*, maps each of the five system data area names to the absolute location of the beginning of the associated area.

5.4-C The Linker

Eight types of data objects are encountered by the linker: the seven Piton data types, tagged with the words **`NAT`**, **`INT`**, **`BITV`**, **`BOOL`**, **`ADDR`**, **`PC`** and **`SUBR`**, and an implementation-internal type tagged **`SYS-ADDR`**, explained below. The linker uses the tag word to determine the type of the object and then maps it into a bit vector accordingly. We use standard binary representation for the naturals and twos complement representation for the intengers. Bit vectors are mapped identically. **`(BOOL F)`** is mapped to the same thing as **`(NAT 0)`** and **`(BOOL T)`** is mapped to the same thing as **`(NAT 1)`**.

An object of the form **`(ADDR (name . n))`** is mapped first to a natural number and then to the bit vector representing that natural in binary notation. To map such a data address to a natural number we look in the data link table to get the base address of **`name`** and add **`n`** to it.

An object of the form **`(PC (name . x))`** is mapped first to a natural number and then to the corresponding bit vector. The natural number is obtained by finding the label table for **`name`** in the label tables and then looking up **`(name . x)`** in that table.

Objects of type **`SUBR`** and **`SYS-ADDR`** are mapped in the obvious way.

5.5 The Formal Definition of LOAD

The definition of **`LOAD`** is

Definition.
```
(LOAD P)
   =
(I->M (R->I (P->R P))).
```

Note that **`LOAD`** is the composition of three functions. These functions correspond exactly to our three-step description of the implementation.

The first step, **`P->R`**, is to use the resources of FM8502 to represent system data in the Piton state to be loaded. The result of **`P->R`** is what we call an *r-state*. An r-state is a modified p-state in which the four components relating to the system data, namely the control and temporary stacks and their two size limits, are eliminated and eleven new components are introduced: the system data segment, the six registers, and the four condition code flags. **P->R** creates the r-state corresponding to a given p-state by copying all but the system data components. It initializes the system data segment of the new r-state by setting up arrays of the appropriate size for the two stacks, copying the stack elements into the high end of the arrays (threading the pointers appropriately for the control stack frames), and setting the registers to point to the tops of the stacks. Because we must write down stack addresses (in the registers and threads) but do not know the absolute location of the stacks yet, we have to introduce a new internal data type that is analogous to the tagged data addresses of Piton but refer to system data. For example, a reference to the 25th position in the control stack is written as the tagged object **(SYS-ADDR (CSTK . 25))**. The actual address will be provided by the link-assembler when we know where the control stack is actually located.[4]

The second step, **R->I**, transforms an r-state into an *i-state*, which has exactly the same components as an r-state except that the program segment now contains i-code programs instead of Piton programs. The new program segment is obtained by compiling each of the programs in the r-state.

The third step, **I->M**, transforms an i-state into the final m-state. This is the link-assembly step. The FM8502 registers and flags are all explicit (but symbolic) in the i-state. The FM8502 memory is obtained by concatenating the three segments together after replacing the symbolic objects by their bit vector representatives.

6. The Proof

We have mechanically proved the correctness theorem. The proof is decomposed into three main lemmas, one for each step of **LOAD**. That is, we prove the correctness of the resource representation, then we prove the correctness of the compiler, and then we prove the correctness of the link-assembler. The final theorem is then obtained by composition.

To state the correctness of the individual steps it is necessary to define formally the semantics of the intermediate objects they produce. Thus, we define two new interpreters, the **R** machine and the **I** machine. The **R** machine interprets r-states and can be thought of as a modification of the Piton **P** machine in which stacks are represented with registers and arrays. When the **P** machine sees a **PUSH-CONSTANT** instruction and pushes an item onto the temporary stack, the **R** machine sees a **PUSH-CONSTANT** instruction, decrements the **tsp** register and writes an item to the newly addressed location in the system data segment.

The **I** machine takes this one step further and interprets i-code programs instead of Piton. The **I** machine is very similar to FM8502 except that its programs are represented symbolically (e.g., a typical i-code instruction might be **(TPUSH_X)**) and its data is also symbolic (e.g., a typical address might be **(SYS-ADDR (CSTK . 25))** and a typical natural might be **(NAT 3)**). All of its operations check that the operands are legal (e.g., one cannot write to a program address) and produce tagged objects of the correct type (e.g., the **ADD_<TSP>{I}_X{I}** instruction produces an integer by summing two integers while the **ADD_<TSP>{N}_X{N}** instruction produces a natural by summing two naturals).

Given these two new machines we can state the sense in which each of the three maps, **P->R**, **R->I**, and **I->M**, is correct. Consider **P->R**. We would like it to be the case that running **P** and then mapping down

---

[4]Conventional implementations do not allow nontrivial initial states. Had we followed the convention, **P->R** would be trivial—it would lay out the standard initial control stack and an empty temporary stack. However, to prove the correctness theorem it was necessary to define the more general **P->R**. Having developed that map, we decided to provide it to the FM8502 Piton user rather than force him to start from standardized initial states.

with **P->R** produces the same r-state as mapping down and running **R**, assuming the initial state is well-formed and no errors occur. That is, we would like r_1 and r_2 in the diagram below to be equal.

```
                n P-steps

      P ------------------> *
      |                     |
      |                     |
 P->R |                     | P->R
      |                     |
      |                     * r1
      *-------------------> * r2

                n R-steps
```

This is not the case. The reason is that the **R** machine has resources that are hidden from the **P** machine. For example, consider that portion of the control stack array not participating in the current stack—the part of the array beyond the current top of stack. Call that part of the array the *hidden stack segment*. If the Piton code executed pops the control stack, then in r_1 the hidden stack segment contains whatever initial value **P->R** puts into the array, while in r_2 it contains whatever garbage was on the stack when the pop occurred. This problem is more pervasive than just the stacks. The flags and the temporary registers are left in different states by the two runs. However, we can define an equivalence relation, **R-EQUAL**, on r-states and prove that (a) two r-equivalent states produce r-equivalent states when run with **R**, and (b) r_1 and r_2 are r-equivalent. It turns out that the hardest problem we had to deal with in the entire Piton implementation proof was the use of hidden resources by the lower level machine.

The correctness theorem for **R->I** states that running with **R** and mapping down via **R->I** produces exactly the same state as mapping down and running with **I**, provided the initial state is well-formed and no errors occur. The main issue here is dealing with the fact that the **I** machine must take several steps for each single step taken by the **R** machine and the code executed at the i-level is the code generated by the compiler on the **R** level instruction being executed. For example, when the **R** machine executes a **CALL** we must be able to show that the **I** machine executes the arbitrarily long sequence of i-code instructions into which **CALL** compiles.

Finally, the correctness theorem for **I->M** states that running with **I** and mapping down via **I->M** produces exactly the same state as mapping down and running **FM8502**, provided the initial state is well-formed and no errors occur. Here there are three issues. First, we must show that for each i-code instruction executed by **I** the **FM8502** executes an appropriate machine code instruction. Second, we must show that addressing is preserved, i.e., the contents of each symbolic address on the i-level is the image of the contents of the image of that address on the **FM8502** level. Third, the data transformation is correct, i.e., adding two naturals and then forming the binary representation is equivalent to forming the binary sum of the representations.

There are many more issues dealt with in the proofs of the three key results and in obtaining the main result from these. However, space prevents us from going into them here. See [12].

Below we enumerate several different subsystems within the entire Piton definition, implementation and proof. For each we show the number of shell definitions, function definitions, and lemmas proved ‘‘for’’ that subsystem. There are a total of 2,774 names involved in the entire proof. Every name has been assigned to exactly one of the subsystems in which it is used—though many names are used in many subsystems. The statistics below do not include the verification of the gate level design of FM8502.

	Shells	Definitions	Theorems
Statement of the Problem			
FM8502	3	78	
Piton	1	320	
Implementation			
Resource Phase	1	11	
Compiler	1	82	
Link-assembler		39	
Load		1	
Totals (Implementation)	(2)	(133)	
Correctness Theorem	__	24	
Totals	6	555	
Proof of Correctness			
Statement of the Problem	6	555	
R Machine		173	
I Machine		197	
M Machine		12	
Theorem 1 (**P->R**)		27	690
Theorem 2 (**R->I**)		74	245
Theorem 3 (**I->M**)		19	385
Theorem 4 (**FM8502-EQUAL-M**)			4
Theorems 5-10	__	23	364
Totals	6	1080	1688

A total of 9 man-months was spent on the Piton project. This includes the formal design of Piton as expressed in the definition of **P**, the development of the correctness theorem, the implementation, and the proof. Before embarking on the Piton project, the author spent 7 man-months on a feasibility study that involved the design, specification, implementation and proof of a 4 instruction ‘‘toy’’ version of Piton that did not include stacks or types and was implemented on a similarly stripped down machine code model. For more details, see [12].

7. Conclusion

Among the significant achievements of the Piton project are the following.

Piton truly provides abstract objects and a new programming language on top of a much lower level machine. Much has been written about this classic problem but the previous attempts to deal with it formally and mechanically have been incomplete. We have in mind specifically the work related to the SRI Hierarchical Design Methodology [16] and its use in the Provably Secure Operating System (PSOS) [14] and the Software Implemented Fault Tolerant (SIFT) operating system [11, 18]. While virtually all of the issues are correctly intuited, we personally find great joy in seeing their formalization and mechanization.

Piton was implemented on top of another verified system, namely FM8502. This represents the first time that two verified systems of such complexity were successfully ‘‘stacked.’’ The basic tenet of stacking is that one can assume only those capabilities one is willing and able to implement and verify. Our commitment to stacking has had several effects on the Piton project. The desire to implement Piton forced into its design such practical considerations as the finite resources of the host machine. The desire to use

Piton forced us to reflect the resource limits into the language itself—programs that cannot anticipate the imminent exhaustion of their resources are impractical.

The Piton implementation obtains efficiency by exploiting the fact that Piton programs are to be proved correct. In particular, the Piton semantics identifies ‘‘erroneous’’ computations and the Piton compiler is proved correct only for non-erroneous computations. By assuming the computation to be non-erroneous the compiler can generate more efficient code. But the only way to establish that a given computation is non-erroneous is to prove it from the formal semantics.

Developing the statement of the correctness result for the Piton implementation was very illuminating. Initially the idea was that the final state produced by a Piton computation could be alternatively produced by mapping the initial Piton state down to the FM8502, running the FM8502 to obtain a final state, and then mapping it up. The problem with this, aside from the issue of erroneous computations, is that we cannot map FM8502 states up to Piton states because not enough information is present. For example, we do not know how to interpret the FM8502 bit vectors in the data segment of the final FM8502 state: Are they natural numbers? Integers? Addresses? Because Piton syntax is untyped, it is impossible to determine the type of the result of a Piton program from a syntactic analysis of the program. Yet untyped languages are useful. They can be used because the user *knows* what the type of the result *will be* and interprets the final bit vectors accordingly. Our correctness result formalizes this notion of ‘‘foreknowledge’’ in the notion of type specification.

The proof of the correctness result was, of course, the hardest task and the place where we learned the most. The implementation involves several phases: resource representation, compilation and link-assembling. Each phase is essentially a form of translation from one programming language to another. The topmost language is Piton, whose formal definition is part of the statement of the problem. The bottom-most language is FM8502 machine code, whose formal definition is embodied in an imagined physical machine and which is the *only* language that is physically implemented. But the compiler produces assembly code and the assembler produces symbolically linked machine code. One need not know about, much less formally define, these intermediate languages to *state* the correctness result. One need not know about, much less formally define, these languages to *use* the FM8502 implementation of Piton. It is even possible to *implement* Piton on FM8502 without ever writing down the definitions of these languages. And yet to *prove* the implementation correct we had to define these intermediate languages formally. In this sense, the Piton machine is three layers above FM8502, each layer implementing a Piton abstraction on top of a more primitive machine. Perhaps the most useful part of the Piton experience was recognizing the need for these layers and learning what kinds of problems could best be handled at each layer. The most difficult problem handled was the use of hidden resources. Our **R** machine and the introduction of the **R-EQUAL** congruence relation was our formal solution to this problem.

In [12] we report the full details of the Piton semantics, proofs of Piton programs, FM8502, the compiler and link-assembler, and the proof of the correctness of FM8502 Piton. We also comment on the relative sizes of the various formal systems involved and the difficulty of producing mechanical proofs.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

2. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

3. R. S. Boyer and J S. Moore. A User's Manual for A Computational Logic. Tech. Rept. 18, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703, 1988.

4. Dan Craigen. *A Description of m-Verdi [Working Draft].* I. P. Sharp Associates, Ltd., 1986.

5. S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor and D. S. Wile. An Overview of AFFIRM: A Specification and Verification System. Information Processing 80, S. H. Lavington (Ed.), October, 1980, pp. 343-348. North Holland Publishing Company.

6. Donald I. Good. Mechanical Proofs about Computer Programs. In C. A. R. Hoare and J. C. Shepherdson, Ed., *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science., 1985, pp. 55-75.

7. D.I. Good, R.L. Akers, L.M. Smith. *Report on Gypsy 2.05 - January 1986.* Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703., 1986.

8. Michael K. Smith, Donald I. Good, Benedetto L. DiVito. *Using the Gypsy Methodology.* Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703., 1988. Revised January 1988.

9. Mike Gordon. Proving a Computer Correct. Tech. Rept. TR 42, University of Cambridge, Computer Laboratory, 1983.

10. W.A. Hunt, Jr. FM8501: A Verified Microprocessor. University of Texas at Austin, December, 1985. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

11. P.M. Melliar-Smith and R. Schwartz. Hierarchical Specification of the SIFT Fault-Tolerant Flight Control System. Tech. Rept. CSL-123, Computer Science Laboratory, SRI International, Menlo Park, Ca., 1981.

12. J S. Moore. Piton: A Verified Assembly Level Language. Tech. Rept. 22, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1988.

13. David R. Musser and David A. Cyrluk. *AFFIRM-85 Installation Guide and Reference Manual Update.* General Electric Corporate Research and Development, 1985.

14. P. G. Neumann, L. Robinson, K. Levitt, R. Boyer, A. Saxena. A Provably Secure Operating System. Tech. Rept. CSL-116, Computer Science Laboratory, SRI International, 1977.

15. W. Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

16. L. Robinson and K. Levitt. "Proof Techniques for Hierarchically Structured Programs". *Comm. ACM 20*, 4 (April 1977).

17. Mark Saaltink. The Verdi Logic [Working Draft]. I. P. Sharp Associates, Ltd., 1986.

18. D.F. Stanat, T.A. Thomas, and J.R. Dunham. Proceedings of a Formal Verification/Design Proof Peer Review. Tech. Rept. RTI/2094/13-01F, Research Triangle Institute, P.O. Box 12194, Research Triangle Park, N.C., 27709, 1984.

19. Stanford Verification Group. *Stanford Pascal Verifier User Manual.* Stanford University, 1979.

20. D. Thompson and W. Erikson. AFFIRM Reference Manual. USC Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, Ca. 90291, 1981.

Table of Contents

List of Figures