Applying Formal Verification to a Commercial Microprocessor*

To be presented at the 1995 IFIP International Conference on Computer Hardware Description Languages, Chiba, Japan, August 1995

Mandayam K. Srivas

Computer Science Laboratory
SRI International
Menlo Park, CA 94025 USA
Tel: +1 (415) 859-6136
Fax: +1 (415) 859-2844
e-mail: srivas@csl.sri.com

Steven P. Miller

Collins Commercial Avionics
Rockwell International
Cedar Rapids, IA 52498 USA
Tel: +1 (319) 395-8008
Fax: +1 (319) 395-4068
e-mail: spmiller@cca.rockwell.com

Abstract— Formal verification using interactive proof-checkers has been used successfully to verify a wide variety of moderate-sized hardware designs. The industry is beginning to look at formal verification as an alternative to simulation for obtaining higher assurance than is currently possible. However, many questions remain regarding its use in practice: Can these techniques scale up to industrial systems, where are they likely to be useful, and how should industry go about incorporating them into practice? This paper describes a project recently undertaken by SRI International and Collins Commercial Avionics, a division of Rockwell International to explore some of these questions. The project formally specified in SRI's PVS language a Rockwell proprietary pipelined microprocessor (the AAMP5, built using almost half a million transistors) at both the instruction-set and register-transfer levels and used the PVS theorem prover to show the microcode correctly implemented the instruction-level specification for a representative subset of instructions. The key results of the project were the development of a practical methodology for microprocessor verification in industrial settings and the discovery of both actual and seeded errors.

I. Introduction

It is becoming increasingly evident within the VLSI design industry that the complexity of many current hardware designs is outstripping the capability of traditional simulation-based tools to adequately verify them. This situation was well-illustrated by the recent floating point bug discovered in Intel's Pentium processor. The industry is beginning to look at formal verification as a technological alternative to simulation for obtaining higher assurance than is currently possible [6].

Recently, SRI International and Collins Commercial Avionics, a division of Rockwell International, undertook a project to explore how formal techniques for specification and verification could be introduced into an industrial process. The project, sponsored by the Systems Validation Branch of NASA Langley and Collins Commercial Avionics, consisted of specifying in the PVS language [10] a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register-transfer levels and using the PVS interactive proof-checker [10] to show that the microcode correctly implemented the specified behavior for a representative subset of instructions. PVS (Prototype Verification System) is an environment for specification and verification that has been developed at SRI International's Computer Science Laboratory. The distinguishing characteristic of PVS when compared to other provers is that it supports a highly expressive specification language with a very effective interactive theorem prover in which most of the low-level proof steps are automated using decision procedures.

The main goal of the project was two-fold: First, to investigate the feasibility of formally specifying and verifying a complex commercial microprocessor that was not expressly designed for formal verification. Second, to explore effective ways to transfer the technology to an industrial setting. The choice of the AAMP5 satisfied the first goal since the AAMP5 was not designed for formal verification, but to provide a more than threefold performance improvement while remaining object-code-compatible with the earlier AAMP2, which is used in numerous avionics applications, including the Boeing 737, 747, 757, and 767.

To satisfy the technology transfer objective, we had to develop a suitable verification methodology and a formal infrastructure to make the technology usable by practicing engineers. This infrastructure includes techniques for decomposing the microprocessor verification problem into a set of verification conditions that the engineers can formulate and strategies to automate the proof of the verification conditions. The development of the infrastructure was one of the key accomplishments of the project. Most of the infrastructure and methodology are general enough

*This work was supported by NASA Langley Research Center under contract NAS1-20334 and NAS1-19704.

to be reused for other microprocessors, certainly in the verification of another member of the AAMP family. This methodology was used to formally specify the entire microarchitecture and more than half of the instruction set and to verify a core set of eleven AAMP5 instructions representative of several instruction classes. However, the methodology and the formal machinery developed are adequate to cover most of the remaining AAMP5 instructions. Although PVS was the vehicle of the experiment, the methodology is applicable to other sufficiently powerful theorem provers.

Another key result of the project was the discovery of both actual and seeded errors. Two actual microcode errors were discovered during development of the formal specification, illustrating the value of simply creating a precise specification. Both were specific to the AAMP5 and were corrected before first fabrication. Two additional errors seeded by Collins in the microcode were systematically uncovered by SRI, who knew that bugs had been seeded, but not their location or identity, while doing correctness proofs. One of these was an actual error that had been discovered by Collins after first fabrication but left in the microcode provided to SRI. The other error was designed to be unlikely to be detected by walk-throughs, testing, or simulation. In [9], we gave an overview of the AAMP5 project emphasizing the technology transfer process with its administrative and managerial aspects. In this paper, we describe the technical approach used in verifying the AAMP5.

II. Pipelined Microprocessor Correctness

Theorem proveing approach has been used to verify a number of microprocessor designs since Hunt applied it to verify the FM8501 processor [8]. (For a more comprehensive bibliography of microprocessor verification, see [13].) However, the AAMP5 is significantly more complex, at both the macro and micro-architecture levels, than any other processor for which formal verification has been attempted; it has a large, complex instruction set, multiple data types and addressing modes, and a microcoded, pipelined implementation. Of these, the pipeline and autonomous instruction and data fetching present special challenges. One measure of the complexity of a processor is the size of its implementation. In the case of the AAMP5, this is some 500,000 transistors.

Recently, mechanical verification has been applied to a number of pipelined processors [12,15,16]. But, there has not been much work in developing a general framework for microprocessor correctness. In [16], Windley developed such a framework using the notion of generic interpreters, but it is not applicable to pipelined processors. We derived the AAMP5 correctness criterion using the general framework developed by David Cyrluk [5] for proving correspondence between state machines in PVS which has been shown to be applicable to pipelined processors.

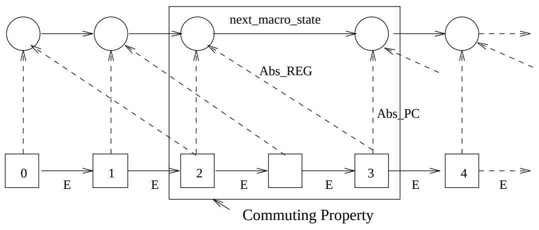

Fig. 1. Pipelined Microprocessor Correctness

Formal microprocessor verification typically involves specifying the processor as a machine that interprets the instructions in the instruction set at two levels—*macro* and *micro*. The macromachine gives a programmer's view of the processor, hiding internal details such as pipelining. The micromachine describes the processor at a lower level, such as the architecture or register-transfer levels.

The correctness criterion for a pipelined processor is expressed, as shown in Figure 1, as a desired correspondence between the execution traces of the macro and micromachines. In the figure, `E` and `next_macro_state` denote the micromachine and macromachine state transition functions, respectively. The objective of the correctness criterion is to ensure that the micromachine does not introduce any behaviors not allowed by the macromachine. Since the micromachine usually takes a number of cycles to execute an instruction that is viewed atomically at the macro level, the two machine traces are required to correspond only at selected points (called *visible* states) in the micromachine trace. For example, a possible definition of a visible state is one in which an instruction gets completed and a new instruction begins. A visible state is shown in Figure 1 with the number of the current instruction in that state. Since macrostate is an abstraction of microstate, it is necessary to define an *abstraction* function `ABS` (shown sparately as `Abs_PC` and `Abs_REG` in the figure for reasons that will be explained shortly) that returns the macrostate corresponding to the state of the micromachine at any given time.

Given the notions of `ABS` and visible state, the correctness criterion described by Figure 1 can be stated as follows[1]: "Every micromachine trace starting with an initial state (`S0`) that abstracts to a macrostate (`s0`) must be abstractable to the macromachine trace starting with the initial state `s0`." To prove such a correspondence between two infinite traces, it is sufficient to prove the *commuting* property that captures the correspondence between the traces for an arbitrary step of the macromachine, i.e., between two arbitrary, but successive visible states. This is stated formally below, where `ABS` and other functions that take a time argument t implicitly operate on the micromachine state at t. The reason that the property is

[1] We also need to ensure that processor resetting will always lead to an initial state in which the correspondence required by `ABS` holds.

stated using an existential quantifier is because the distance between two successive visible state can vary.

```
commuting_property: LEMMA (FORALL (t: time):
 visible_state(t) => EXISTS (tp: time | tp > t):
  stays_low(t+1, tp-1)(visible_state) &
    visible_state(tp) & ABS(tp) = next_macro_state(ABS(t))
```

The trace correspondence criterion described in the previous paragraph can also be used for a pipelined processor provided appropriate adjustments are made to deal with the consequences of pipelining on instruction execution. First, in a pipelined processor, a new instruction enters the pipeline before the results of the previous uncompleted instructions are at their destinations. This characteristic of pipelining can be handled by allowing `ABS` to peek into future microstates, possibly nonvisible, following the current visible state (eg., `Abs_REG` shown in Figure 1) to get correct values for some of the macrostate components. Typically, the values for all but the program counter must be obtained from the future state. We refer to the distance into the future from the current visible state where the information is to be obtained as the *latency* for the abstraction function.

Second, some pipelined processors, typically RISC processors, do not inhibit all kinds of pipeline hazards. For example, they allow instructions following a conditional branch to execute assuming the branch would not be taken leaving the burden of generating correct code to the compilers. One way to handle this is (as shown in [12]) either to restrict the set of programs for which the two machines are related or expose some of the effects of pipelining in the macromachine. Further adjustment would be needed if pipelining allows out-of-order instruction execution.

Since the AAMP5 was designed to be object-code compatible with its nonpipelined CISC-like predecessor, its pipeline does not use some of the aforementioned characteristics found in RISC processors. Hence, we were able to formalize the correctness of its pipeline by suitably adjusting the ABS definition. However, coming up with a suitable definition for ABS and the visible state and mechanizing the proof of the correctness criterion for the AAMP5 were challenging for several reasons. First, the AAMP5 is a stack machine in which the top few elements of process stack are "cached" in internal registers. This caching introduces a significant gap between the memory models at the two levels that is nontrivial to bridge.

Second, in the AAMP5, instruction fetches and data transfers happen autonomously with respect to the interpretation of instructions in the pipeline. This makes it hard to choose a reference point for characterizing the visible state since the distance between two consecutive visible states is indefinite and varying. If the definition of the visible state is not engineered carefully the abstraction can become complicated. Burch and Dill [2] have proposed the idea of using a sequence of NOPs to flush out the states of the desired internal registers to eliminate the need to explicitly define an abstraction function in some microprocessor verifications. This idea cannot applied for the AAMP5 because of the indefinite distance between visible states and also because of the nontrivial abstraction gap between the levels.

In the following sections, we elaborate these points and describe the main technical issues that emerged in formalizing and verifying the correctness criterion of AAMP5. Although we discuss all these issues in the context of the AAMP5, the techniques presented are applicable to other designs of similar complexity that use similar features. Sections III and IV describe the AAMP5 microprocessor and its formal specification. Section V describes the formalization of the correctness of the AAMP5 pipeline and the approach we used to mechanize the proof of correctness. Our conclusions are presented in the last section.

III. The AAMP5 Microprocessor: An Informal Description

The Advanced Architecture Microprocessor (AAMP) consists of a Rockwell proprietary family of microprocessors based on the Collins Adaptive Processor System (CAPS) originally developed in 1972 [1]. The AAMP architecture is specifically designed for use with block-structured, high-level languages such as Ada in real-time embedded applications. Although it is based on a stack architecture, the AAMP5 design uses pipelining and other modern RISC-like features to provide a performance that rivals or exceeds that of most commercially available 16-bit microprocessors.

A. The Macroarchitecture

The AAMP5 was designed as an object-code-compatible replacement for the earlier AAMP2 [1], with advanced implementation techniques such as pipelining providing a more than threefold performance improvement. The main macroarchitecture features that make its specification and verification challenging are its support for multi-tasking and error handling, process stack, and the variety of its CISC-like instructions. We briefly discuss these features below. A more detailed discussion can be found in [1].

The AAMP provides separate address spaces for *code memory* and *data memory*. The process stack is central to the AAMP macroarchitecture, implementing in hardware many of the features needed to support high-level block structured languages and multi-tasking. Each task maintains a single process stack in the task's data environment. Many of the internal registers maintained by the AAMP are used to define the process stack. For example, the TOS (top of stack) register points to the topmost word in the process stack.

The AAMP instruction set (209 instructions) is large and CISC-like. It closely resembles the intermediate-level output of many compilers, directly supporting high-level language constructs such as procedure calls and returns. The instructions vary in length from 8 to 56 bits yielding improved throughput and code density. The instruction set supports a variety of data types, including 16- and 32-bit integer, 16- and 32-bit fractional, 32- and 48-bit floating-point, and 16-bit logical variables and floating-point formats, which are not considered in the verification.

B. The Microarchitecture

The internal microarchitecture of the AAMP5 employs four major, semiautonomous, functional units as shown in the block diagram of Figure 2.

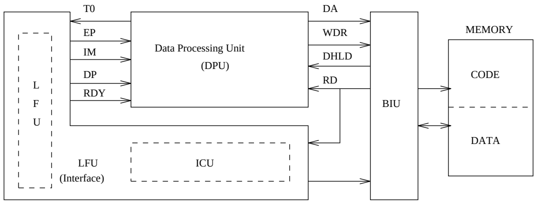

Fig. 2. AAMP5 Block Diagram

The Data Processing Unit (DPU), where the microcode resides, provides the data manipulation and processing functions to execute the AAMP5 instruction set. Instruction fetching is performed by the Lookahead Fetch Unit (LFU) by maintaining a 4-byte-long instruction queue that it endeavors to keep full. In addition to prefetching into the instruction queue, the LFU parses the instruction stream, assembles immediate operands (IM), and provides the DPU with a microprogram entry point (EP) translated from the instruction opcode. The LFU also passes the program counter (DP) associated with the parsed instruction. The AAMP5 includes 1024 bytes of direct-mapped instruction cache (ICU). Because the operational details of ICU are hidden from the DPU, the LFU and the ICU are shown in Figure 2 enclosed inside a shell with which the DPU interacts directly. The Bus Interface Unit (BIU) performs external fetchs for the LFU, as well as data reads and writes initiated by the DPU.

The DPU provides the data manipulation and processing functions required to execute the AAMP5 instruction set. A block diagram of the DPU is shown in Figure 3, which also includes the details of the LFU. The DPU has two main parts: the datapath and the microcontroller.

C. The Datapath

The datapath consists of the components that perform the required data manipulation for instruction execution.

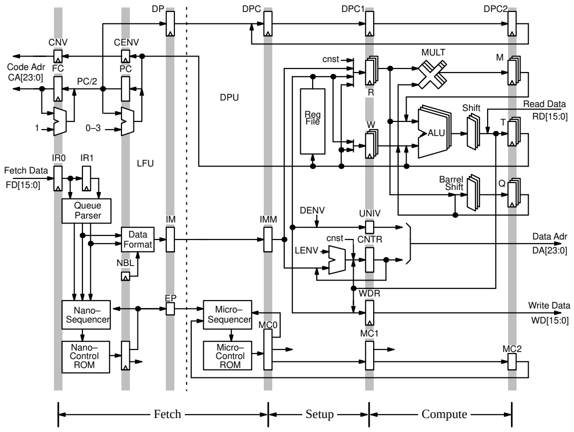

Fig. 3. The LFU and DPU Datapath

It includes a 10-word multi-port register file, a 48-bit ALU, shifters, multiplication support, and several registers. Some of the registers (R and W) are used to hold the operands of the ALU and some are used to hold the results (T, Q, and M triplets) after ALU and shift operations.

The ALU and the Reg File (register file) are two of the most complex units in the DPU. The ALU supports a large number of logical and 2's complement and 1's complement arithmetic operations on 48-bit bit-vectors. Besides the two bit-vector operands, the ALU accepts a carry-in input and produces a carry-out. The ALU can, if necessary, bypass the operand registers (R and W) and get its operands directly from the result registers (T, Q, and M).

The Reg File is a complex multi-port register file that can be read (and written) simultaneously from (into) up to three registers relative to a read (write) index controlled by the microcode. It contains bypass logic to use the write value instead of the contents of a register during a read if there is a read-write clash. Six designated registers (STK0 through STK5) in the register file (along with the T registers) are used to implement the stack cache as described in section E. The remaining registers in the register file are used as pointers to the code and data memory such as the top-of-stack and data environment pointer (DENV).

D. The Microcontroller

The microcontroller generates the signals that control the movement of data within the datapath as well as the next microinstruction to be read from the ROM. It contains the microprogram store, the microsequencer, which computes the next microaddress, and the microinstruction control registers (MC0, MC1, and MC2). AAMP5 instructions are executed in a 3-stage—*fetch*, *setup*, *compute*—pipeline, if one includes the activities that take place prior to an instruction moves into the DPU from the LFU in the fetch stage. The pipeline and its stages are described in more detail in Section A. The activities performed in the pipeline stages are controlled by a microinstruction as it moves from MC0 to MC2. The program counter DP is also pipelined using a sequence of registers so that DPC, DPC1, and DPC2 contain the program counters corresponding to the instructions in MC0, MC1, and MC2, respectively.

E. The Stack Cache and Stack Adjustment

Conceptually, the process stack consists of an area of data memory, a stack limit (SKLM) register, and a *logical* top-of-stack. register. In actuality, the DPU holds up to nine words of the top of the process stack in a set of internal registers called the *stack cache*, with the rest being stored in memory. The T register-triplet always holds the latest data item, which can be one, two, or three words long, pushed on to the top of the stack. The rest of the stack cache elements are stored in the register file. The DPU maintains the *physical* top-of-stack (TOS) register marking the boundary between the portion of the stack implemented in data memory and the portion implemented in the cache and the stack cache occupancy information. Note that the logical top-of-stack is the sum of TOS and the stack cache occupancy.

The occupancy state of the stack cache that would result if an instruction were completed is maintained in a pair of registers (not shown in Figure 3) for each of the AAMP5 instructions being executed in MC0 (SV and TV) and MC1 (SV1 and TV1). When an instruction to be executed requires more operands than are available in the cache, the processor automatically reads additional locations from memory to the stack cache. Similarly, when an instruction will place more operands on to the stack than there are empty spaces in the stack cache, the processor automatically writes the cache register contents to memory. The process of maintaining the proper number of operands in the stack cache is called a *stack adjustment*.

IV. Formal Specification of the AAMP5

This section describes some of the interesting technical issues that emerged during the development of the formal specifications of the AAMP5. Although we identified a core set of 13 reprsentative instructions to mechanically verify, we wanted to model the microarchitecture in its entirety and as much of the macroarchitecture as possible so that verification could be later extended to a larger set. The macroarchitecture specification specified 108 of the AAMP's 209 instructions including the executive service functions. The macroarchitecture specification consisted of 2550 lines of PVS. The microarchitecture specification consisted of 2679 lines of PVS. This does not include the specification of bit-vectors used in the specification that evolved into a reusable library of 2030 lines. The bit-vector library was primarily developed by Rick Butler at NASA.

A. Macroarchitecture Specification

Our approach to specifying the AAMP5 macroarchitecture is similar to the one used in most earlier microprocessor verification. The state of the macromachine is modeled as a record that includes external memory and the internal state that affects its observable behavior, such as the internal registers defining the process stack and the program counter. The AAMP5 instructions are specified by defining a next-state function that characterizes the effect of executing the "current" instruction pointed to by the program counter. The behavior of all instructions in the absence of external interrupts can be specified in this fashion.[2] Two issues, one general and the other specific to AAMP5, that arose in the development of macroarchitecture specification are discussed below.

B. Constructive vs. Descriptive Specifications

One of the early choices facing us in specifying the macro-architecture was whether to specify the `next_macro_state` function *constructively*, i.e. by explicitly defining how the result of the function is to be constructed, or *descriptively*, i.e., by stating a set of properties (axioms) that the function is to satisfy. The main advantage of a constructive style of specification is that the PVS language mechanisms will ensure that the function is not only well-defined but total. A disadvantage is that it is difficult to leave parts of the specification deliberately underspecified. A descriptive style is naturally suited for underspecification, but makes it possible to introduce inconsistent axioms. We initially chose the constructive style since (besides the reasons given above) many of the AAMP instructions are similar enough that they could be compactly specified using a constructive style. However, while doing the correctness proofs it became clear that a more descriptive style of specification in which the change in state was defined more directly would be helpful as an intermediate step in the proof. Fortunately, these could be stated as lemmas that could be proven from the original specification, preserving our investment in the original specification. An example of the descriptive style of specification is presented below for the ADD instruction for the case when addition does not overflow. The other cases is characterized similarly by

[2] Although formal analysis of the interrupt behavior of the AAMP5 was not included in this effort, the basic framework developed is adequate to allow it to be easily added. Details on how the AAMP5 interrupts can be specified and verified can be found in [13].

other lemmas. In the constructive style all the cases were embedded inside a function definition in an algorithmic style.

```
ADD_lemma_1: LEMMA
  LET X = current_data_env(st)(tos(st)+1),
      Y = current_data_env(st)(tos(st))
  IN current_opcode(st) = ADD &
      arith_update.exception_number(st)=0  =>
        normal_macro_machine.next_macro_state(st) =
         st WITH [(dmem)(word2denv(denv(st)))(tos(st)+1)
                        := X + Y,
                   (pc)  := pc(st)  + 1,
                   (tos) := tos(st) + 1]
```

B.1 The Stack Cache Abstraction

One of the most interesting issues pertaining to the use of abstraction in specification arose in modeling the process stack. Conceptually, it is desirable to hide the presence of the stack cache so that an application programmer does not have to be concerned with it. In actuality, effect of the stack cache is visible at the macro level in two ways.

For performance reasons, the AAMP does not check memory accesses to determine if the word being referenced lies in the vicinity of the stack cache. As a result, REF and ASSIGN instructions that access the region of memory between the physical TOS and the logical TOS may obtain values different from those held in the stack cache. In practice this does not pose a problem since well behaved applications do not directly access the process stack. Since applications seldom write assembly code for the AAMP (recall that it is designed for use with high order, block structured languages such as Ada), this is primarily a concern for the compiler writers. The second manifestation of the stack cache arises because the AAMP signals an overflow when the physical TOS exceeds the stack limit, rather than when the logical TOS exceeds the stack limit. In effect, the existence of the stack cache allows a few more words to be written to the process stack than should strictly be allowed. While benign, these two manifestations of the stack cache made it difficult to write a precise definition of the AAMP5's behavior without bringing the details of the stack cache into the macro level specification. To avoid this, two adjustments were made to the specification of the macroarchitecture. To reflect the constraint that memory accesses should not be made to the vicinity of the stack cache, we left the effect of such accesses unspecified. We proved the correspondence between the two levels of specification only when the instructions obeyed this constraint. Using the physical TOS rather than the logical TOS to detect stack overflow was a more difficult problem to resolve. This is dealt with in the proofs by (1) showing the correspondence between the two levels only as long as there is no stack overflow and (2) by proving a correctness statement about the behavior of the stack overflow condition as a separate property of the micromachine.

C. Microarchitecture Specification

The two main technical issues that had to be addressed in specifying the microarchitecture were: (1) the level of detail at which the different parts of the microarchitecture are to be specified and (2) the style of specification to be used to model the design.

One of our objectives was to to keep the abstraction level in the microarchitecture specification as close to that used in the official (unpublished) microarchitecture document as possible. The microarchitecture document describes the design approximately at the register-transfer level, with some of the components, such as ALU and the register file, being described as black boxes. The DPU, in which most of the data transformations initiated by the microcode occurs, is specified at the register-transfer level. Since the internal details of LFU and BIU are not needed to verify the correctness of the microcode, the LFU and the BIU are specified more abstractly by formalizing the expected behavior of the interface between those blocks and the DPU. The LFU and BIU provide a set of services for the DPU that can be formalized as a set of "rely guarantee assertions" involving the interface signals. For example, one of the interface assertions specified guarantees that whenever RDY output of the LFU is true the LFU outputs EP and IM have the appropriate information pertaining to the instruction associated with the program counter DP. The interface assertions also include liveness assumptions about the behavior of the LFU and the BIU. For example, the LFU, if not ready with an instruction, is guaranteed to be eventually ready with a new instruction, provided the DPU obeys a certain protocol.

Formalizing the interface specification was one of the most time-consuming parts of specifying the microarchitecture since it had to be reverse engineered from the LFU/BIU design details and by consulting the design engineers. Nevertheless, abstracting the LFU and BIU provided a natural way of decomposing the verification task. The task of verifying that the LFU/BIU satisfies our interface specification is better-suited for finite-state enumeration tools, such as model-checkers, than theorem provers. To perform such a verification, it is necessary abstract appropriate parts of the LFU-BIU-DPU system to construct a finite-state machine for the model-checker. We have explored [11] this problem for a very simplified model of the AAMP5 using a prototype enhancement to PVS that provides the ability to invoke a model-checker as a decision procedure within PVS.

The DPU is constructed by combining the specifications of the datapath and the microcontroller at the register-transfer level. In most applications of formal logic for the verification of register-transfer-level hardware, the design is modeled as a finite state machine with an implicit clock. The state of the machine consists of the states of the sequential components, such as registers, in the design. The next state function defines the new state value at the next cycle of the implicit clock. The exact style used to specify the state machine implemented by the design can vary. We used a *functional* style of specification where the inputs and outputs of every component are modeled as *signals*, where a signal is a function that maps time, i.e., number of cycles of the implicit clock, to a value of the appropriate type. Every signal that is an output of a component is specified as a function of the signals appearing at the inputs to the component. The signal definitions implicitly specify the connectivity between the components. We decided to use a functional style because proofs based on functional specifications tend to be more automatic than those involving *predicative* style specifications, popularized by Gordon [7]. For a more detailed look at a comparison of the two styles and their impact on proofs see [14].

V. Verifying the AAMP5

This section describes the verification of the AAMP5. We first informally describe the AAMP5 pipeline and then formalize its correctness criterion. Then, we describe the mechanization of the proof of correctness.

A. The AAMP5 Pipeline

The reader is advised to refer to Figure 3 in section C while reading this section. The AAMP5 executes its instructions in a three-stage pipeline: *fetch*, *setup*, and *compute*. The fetch stage consists of the DPU activities that occur prior to the entry of the first microinstruction of an AAMP5 instruction into MC0. This includes reading the microinstruction at EP from the ROM and checking if the instruction requires a stack adjustment, and if so, performing the stack adjustment. The setup stage consists of the activities controlled while the microinstruction is in MC0, which includes computing the new stack cache occupancy state and reading the register file to determine the sources for the ALU input registers R and W. The compute stage consists of the activities determined by the part of the microinstruction that moves from MC0 to MC1. It includes computing the results of the ALU, checking if there will be delayed jump and initiating writing of the ALU results. If a delayed jump is detected (as a function of the ALU outputs) in the second stage, then both MC0 and MC1 are cleared to abort the operations normally performed by the MC0 and MC1 microinstructions and the jump-address field of MC1 is passed onto MC2, which will be used as the target microaddress of a potential delayed jump. A delayed jump is typically used to implement exceptions such as arithmetic overflow. Note that the register MC2 is used only for delayed jump execution and does not add a stage under normal instruction execution.

A.1 Normal Pipeline Operation

The DPU logic is designed so that the DPC register always contains the program counter value associated with the macroinstruction that is being interpreted in MC0. A similar correspondence holds between DPC1 and MC1 as well as DPC2 and MC2. We use the macroinstruction associated with DPC as the reference point and refer to it as the *current* instruction. Under *normal* pipeline operation, i.e., when AAMP5 instructions are being completed at the rate of one per cycle, the DPU performs the setup stage of the current instruction along with the compute stage of the previous instruction and the fetch stage of the next instruction.

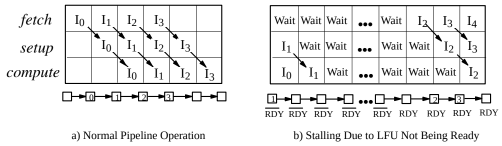

Fig. 4. Pipeline Execution Traces

The execution trace of the micromachine under normal pipeline operation is shown in Figure 4(a). A state (square) in the execution trace (shown at the bottom of Figure 4) with a number `n` in it denotes the visible state where instruction `n` is the current instruction. The column above a state transition arrow shows the instructions in the pipeline and the stages that are being performed in a given cycle.

A.2 Pipeline Stalling and Delayed Jumps

One of the challenges posed by the AAPM5 from the point of view of verification is that there are a number of conditions under which the normal operation of the pipeline can be interrupted. Verifications under these conditions has to be handled in special ways. When the AAMP5 pipeline is at a visible state, the pipeline operates normally (i.e., the distance to the next visible state is one cycle) only if the following conditions hold: (1) The current instruction, i.e. the one in the setup stage in the pipeline, is implemented by a single microinstruction. (2) The previous instruction, i.e., the one in the compute stage, (a) does not cause a delayed jump (nextDJMP, the value assigned to DJMP in the next cycle, is false), and (b) will not not cause a data hold (DHLD is false). (3) The LFU is ready with the next instruction (RDY is true). (4) The next instruction will not need a stack adjustment (SADJcondn is false).

If any of the conditions listed do not hold the pipeline is considered to be *stalled* or *extended* and the distance to the next visible state will be more than one cycle. For example, Figure 4(b) shows the trace when the LFU is not ready with the next instruction. In this case, a jump is made to a special microroutine (MAP_WAIT) that implements a busy wait loop which is exited only when RDY becomes true again. The distance to the next visible state can be arbitrarily large depending on when RDY becomes true again.

A.3 The AAMP5 Pipeline Correctness

The correctness of the AAMP5 pipeline is formalized by appropriately instantiating the general commuting property formalized in Section II. For this, we first need to define `ABS` and `visible_state` for the AAMP5.

The main issue to be considered in characterizing the visible state is which of the macroinstructions in the pipeline and what point of its execution is to be used as the reference point to relate the microstate to the macrostate. In principle, one can use any of the instructions in the pipeline, but the choice influences how complex the definition of the abstraction function will be. In most pipeline verifications, the instruction in the fetch stage is used as a reference point. This choice, however, is not suitable in the present case because variations in the latency of the pipeline due to pipeline stalling will be embedded in the definition of the abstraction function. For automating proofs, it is advisable to choose the reference point so that abstraction function need only look a fixed distance into the future.

So, we define a visible state `visible_state` as one in which the first microinstruction of a new AAMP5 instruction is in MC0 and is guaranteed to advance to MC1 in the next cycle. That is, the left-over effects of the previous instruction are guaranteed to be completed in the next cycle. Under this characterization of the visible state, the latency for the ABS function is exactly one cycle. During normal pipeline operation, the distance between consecutive visible states is exactly one cycle. The abstraction function `ABS` acts like a projection function on the microstate except for the stack-structured memory. A formalization of the commuting property for the AAMP5 is given below. For more details see [13].

```
commuting_property: LEMMA
 visible_state(t) & proper_instrns_in_pipe(t)
                  & no_logical_stack_overflow(t)
  => EXISTS (tp: time | tp > t):
      stays_low(t+1, tp-1)(visible_state) &
        visible_state(tp)
          & ABS(tp) = next_macro_state(ABS(t))
```

The commuting property has two additional preconditions when compared to the general commuting diagram of Section II to handle the consequences, described in section B.1, of the hiding of the stack cache from the macro level. `Proper_instrns_in_pipe(t)` ensures that the memory address operands (if any) of every macroinstruction in the pipeline are not within the area overlaid by the stack cache; `next_macro_state` is left undefined if this predicate were false. `No_logical_stack_overflow(t)` ensures that the logical process stack would not overflow as a result of executing the current instruction.

B. Mechanizing the Proof of Correctness

A common strategy used to prove the commuting property is to start from an arbitrary visible state at the lower left-hand corner of the diamond, traverse the two possible paths to the top right-hand corner of the box representing the commuting property in Figure 1, then check whether the two states resulting from the traversals are equivalent. Traversing the top path, which entails applying the abstraction function followed by "running" the macromachine one step, results in the *expected* state. Traversing the bottom path, which consists of running the micromachine a multiple number of cycles followed by an application of the abstraction function, gives the *actual* state.

In a proof, running the micro and the macromachines can be accomplished by *rewriting*, which consists of unfolding every occurrence of a defined function with its definition until all the functions have been simplified to primitive expressions. Checking whether the expected and actual states are equivalent is essentially a problem of checking the equivalence of boolean or bit-vector expressions since the state of most of the machine components is modeled as either boolean or bit-vector types. As discussed in section A, the number of cycles that the micromachine needs to be run depends on the type of current instruction and whether one or more of the stalling conditions apply. Hence, a proof typically consists of a case analysis on these conditions.

Our main objective in deciding on a suitable approach for mechanization of the proof was two-fold. First, the verification method had to be automatic enough so that the Collins engineers could perform proofs of instructions after proofs of a few representative instructions were demonstrated to them. Second, the verification method should permit incremental and partial verification of instructions. The importance of automation cannot be overemphasized since it is a pre-requisite for transfer of this technology to practice. We also wanted the ability to perform partial verification since that is a good way of getting early feedback from the verification process.

One way to facilitate automation, is to transform the verification problem into one of proving a set of properties each of which relates a pair of states of the micromachine that are a fixed and finite distance apart in time. As shown in [4], for register-transfer level designs specified in the functional style described in Section C such "finite distance" properties can be automatically proved using a *core hardware strategy*. The hardware strategy consists of repeatedly performing the following sequence of proof steps: (1) rewriting the property using the design specification, (2) lifting the boolean expression in if-then-else structures to the top-level, and (3) simplification using PVS's BDD and other descision procedures. The special-purpose decision procedures for microprocessors proposed in [2, 3] are efficient methods for proving such properties under certain restrictions.

To accomplish such a transformation of the verification problem, we decomposed the proof of the commuting property into two parts: one that reasons exclusively about the stalling behavior of the pipeline and another that reasons about the instruction correctness in the absence of stalling. This decomposition was natural since pipeline stalling is implemented by a set of special-purpose microroutines that are invoked as a subroutine during an instruction execution. The correctness of pipeline stalling was characterized by a set of formulas called *general verification conditions* since they are common to all instructions. The correctness of instructions in the absence of stalling were formalized by a separate set called *instruction-specific* verification conditions. The commuting property is proved by composing the two sets of verification conditions.

B.1 General Verification Conditions

A general verification condition states a property of the following form: "if the pipeline is in a stalling state then it will eventually progress to an appropriate non-stalling state without affecting the externally visible parts of the micromachine state." Since the general verification conditions are not in the form of a property that relates states that are a finite distance apart, their proof is not as automatic as that of the instruction-specific verification conditions. The proof involves induction on the length of the time taken by the memory to respond or the size of the stack adjustment needed. However, we only needed to formulate about half a dozen general verification conditions to characterize the three stalling situations discussed in Section A.2. Since the general verification conditions are common to all instructions, their proof needed to be done once and for all by verification experts at SRI and the Collins engineers did not need to worry about them.

B.2 Instruction-Specific Verification Conditions

In formulating the instruction-specific verification conditions, it was necessary to apply two additional decomposition steps for pragmatic reasons. The first decomposition was performed to handle the fact that even in the absence of pipeline stalling the length of execution of an instruction, although definite, can vary depending on the type of the current instruction and whether an instruction causes an exception or user-recoverable error. One way to handle the variation in the length of the execution time for different instructions is to define an "oracle" function that gives, a priori, the expected execution time for all instructions under different situations. Another possible approach is to partition the proof by hand into different instructions and conditions according to the execution times. We took the second approach since it was more amenable to incremental and partial verification.

The second decomposition involved separating, in the proof of the commuting property, the step of applying the abstraction function and executing the macromachine from the step of symbolically executing the micromachine. This decomposition was done to keep the complexity of the automatic rewriting to be done in a single proof manageable. For example, shown below is one of the instruction-specific verification conditions for the ADD instruction under the condition that addition does not overflow. The overflow situation has a separate set of verification conditions, as does every situation that requires a distinct number of cycles to complete the ADD instruction execution. In the absence of an exception, the latency of ADD is 2 cycles. The formula expresses constraints on the T0 register, which holds the top element of the process stack. It is important to note that the expected value for the state of a component is expressed only in terms of the initial values of the states of the macro-level relevant components. Thus, although the verification conditions do not relate the microstate to the macrostate, they are, in effect, making assertions about the intended changes to the macrostate.

```
T0_correctness: LEMMA
 visible_state_with(ADD)(t) =>
  T0(t+2)=(sign_extend[16](48)(ith_top_of_scache(t+1)(1))
```

In our methodology, the responsibility of the design engineer in the verification phase would be to formulate and prove the instruction-specific verification conditions for the instructions in the instruction set. Our core hardware strategy is applicable to the instruction-specific verification conditions since they are in the form of a "finite distance" property. Even when the hardware strategy is unable to automatically prove a verification condition, it reduces the proof to one or more goals of proving equality on two bit-vector expressions.

VI. Conclusions

The central goal of the AAMP5 project was to explore the feasibility of formally specifying and verifying a complex commercial microprocessor and develop a microprocessor verification methodology that could be used by practicing engineers who are not formal verification experts. Based on the experience we have gained so far, we do not see our verification approach and methodology any technical obstacles to extending either the specification of the macro-architecture or the correctness proofs for the whole of AAMP5. In the following, we discuss some of the other lessons learned on this project and their implications for the industrial use of formal methods.

Cost: The cost of developing and validating the macro and micro-architecture specifications and developing the proofs of correctness were significant, but many of these expenses have to be attributed to the exploratory nature of the project. Reuse of specifications, such as the bit vector theories, proof strategies, and expertise should greatly reduce these costs in the future. A large portion of the time spent on the correctness proofs was invested in the development of reusable proof strategies rather than just proving the correctness of the core set of instructions. Even so, we feel that more effective and efficient automation of lower level proofs involving rewriting and bit-vectors are needed to make formal verification cost-effective. For example, some of the decompositions that we used to facilitate our proofs would not have been necessary if the hardware strategy could be implemented more efficiently. Toward this end, it would be interesting to explore integrating special purpose decision procedures, such as [2], into PVS.

Benefits: While it is not hard to convince engineers that formal methods subject a design to a more thorough analysis than traditional methods can, these benefits must also be evaluated relative to their cost. Until technological advances reduce the cost of formal verification significantly below the present level, it will be necessary to apply formal verification prudently. One way to maximize the benefits is to apply it to well-understood problem domains, such as microprocessor designs and perform partial verificationy by "looking into dark corners."

Two errors were found while constructing the macroarchitecture specification. Both errors were found while trying to formally specify the behavior of the AAMP under unusual circumstances that were not clearly specified in the AAMP2 Reference Manual, prompting a team member to examine the microcode in the AAMP5. Both errors were unique to the AAMP5 and corrected before first fabrication. The proof of correctness phase revealed two errors in the microcode. Both of these errors were known to the AAMP5 designers but were left in the microcode supplied to SRI for verification without divulging any information about the errors. While the second error was deliberately planted by Collins engineers in the microcode, the first one had escaped the the microcode traditional walk-throughs and simulations. More details are given in [13].

Technology transfer: Despite their name, formal methods provide remarkably little methodology to guide their use in a new setting. Given this, it seems prudent to plan for costs to be high the first time around and to expect most of the benefits to appear on subsequent projects of a similar nature. It is our belief that the groundwork performed on this project will greatly lower the cost of specifying and verifying another member of the AAMP family. To test this hypothesis we have begun a new project to use the methodology developed to verify another member of the AAMP family in its entirety.

References

[1] D. Best, C. Kress, N. Mykris, J. Russel, and W. Smith. An advanced-architecture cmos/sos microprocessor. *IEEE Micro*, pages 11—26, August 1982.

[2] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.

[3] D. Cyrluk and P. Narendran. Ground temporal logic—a logic for hardware verification. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 247–259, Stanford, CA, June 1994. Springer-Verlag.

[4] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[5] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[6] James Glanz. Mathematical logic flushes out the bugs in chip designs. *Science*, 267:332–333, January 20, 1995.

[7] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, September 1985.

[8] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1994.

[9] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[10] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[11] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, Lecture Notes in Computer Science, Liege, Belgium, June 1995. Springer-Verlag. To appear.

[12] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[13] Mandayam Srivas and Steven P. Miller. Formal verification of a commercial microprocessor. Technical Report SRI-CSL-95-4, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1995.

[14] Mandayam Srivas, Natarajan Shankar, and Sreeranga Rajan. Hardware verification using PVS: A tutorial. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995. Forthcoming.

[15] S. Tahar and R. Kumar. Implementing a methodology for formally verifying risc processors in hol. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, number 780 in Lecture Notes in Computer Science, pages 281–294, Vancouver, Canada, August 1993. Springer-Verlag.

[16] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, CA, June 1990.