Formal Verification of an Avionics Microprocessor

Mandayam K. Srivas
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Steven P. Miller
Collins Commercial Avionics
Rockwell International
Cedar Rapids, IA 52498 USA

SRI International Computer Science Laboratory
Technical Report CSL-95-04

June 1995

Abstract

Formal specification combined with mechanical verification is a promising approach for achieving the extremely high levels of assurance required of safety-critical digital systems. However, many questions remain regarding their 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 report discusses a project undertaken to answer some of these questions, the formal verification of the AAMP5 microprocessor. This project consisted of formally specifying in the PVS language a Rockwell proprietary microprocessor at both the instruction-set and register-transfer levels and using the PVS theorem prover to show that the microcode correctly implemented the instruction-level specification for a representative subset of instructions. Notable aspects of this project include the use of a formal specification language by practicing hardware and software engineers, the integration of traditional inspections with formal specifications, and the use of a mechanical theorem prover to verify a portion of a commercial, pipelined microprocessor that was not explicitly designed for formal verification.

Contents

List of Figures

Chapter 1

Introduction

Software and digital hardware are increasingly being used in situations where failure could be life threatening, such as aircraft, nuclear power plants, weapon systems, and medical instrumentation. Several authors have demonstrated the infeasibility of showing that such systems meet ultra-high reliability requirements through testing alone [BF93, LS93]. Formal methods are a promising approach for increasing our confidence in digital systems, but many questions remain as to how it can be used effectively in an industrial setting.

This report describes a project, formal verification of the microcode in the AAMP5 microprocessor, conducted to explore how formal techniques for specification and verification could be introduced into an industrial process. Sponsored by the Systems Validation Branch of NASA Langley and Collins Commercial Avionics, a division of Rockwell International, it was conducted by Collins and the Computer Science Laboratory at SRI International. The project consisted of specifying in the PVS language developed by SRI [OSR93] a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register-transfer levels and using the PVS theorem prover [ORS92, SOR93] to show that the microcode correctly implemented the specified behavior for a representative subset of instructions.

The central result of this project was to demonstrate the feasibility of formally specifying a commercial microprocessor and the use of mechanical proofs of correctness to verify microcode. This result is particularly significant 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. As a consequence, the AAMP5 is one of the most complex microprocessors to which formal methods have been applied.

Besides demonstrating the verification of a subset of AAMP5 microcode, an equally important accomplishment of the project was the development of a methodology that can be used by practicing engineers to apply formal verification technology to a complex microprocessor design. This 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.

This methodology was used to formally verify a core set of eleven AAMP5 instructions representative of several instruction classes. Although the number of instructions verified is small, the methodology and the formal machinery developed are adequate to cover most of the remaining AAMP5 microcode. The success of this project has lead to a sequel in which the same methodology is being reused to verify another member of the AAMP family.

Another key result 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. Two additional errors seeded by Collins in the microcode were systematically uncovered by SRI 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.

Several other results emerged during the project, including the ease with which practicing engineers became comfortable with PVS, the need for libraries of general-purpose theories, the usefulness of formal specification in revealing errors, the natural fit between formal specification and inspections, the difficulty of selecting the best style of specification for a new problem domain, the high level of assurance provided by proofs of correctness, and the need to engineer proof strategies for reuse.

Organization of the Report

This report is organized as follows. Chapter 2 provides general background, describing the participants in the project, the history of the AAMP family of microprocessors, the PVS specification language, and a brief survey of related work. Chapter 3 discusses the goals, organization, and history of the project. Chapter 4 describes the AAMP5 instruction set (macro) architecture, the PVS specification of the macroarchitecture, and how the specification was produced and validated. Chapter 5 provides a similar discussion of the AAMP5 register transfer (micro) architecture and its formal specification. Chapter 6 describes the formal verification effort, including an overview of the general approach to microprocessor verification and the impact of the AAMP5's pipelining and asynchronous memory interface on this model, a detailed discussion of our approach, and a summary of the errors found and the scope of the verification performed. Chapter 7 presents conclusions and lessons learned.

Chapter 2

Background

2.1 NASA Langley, SRI International, and Rockwell Collins

NASA Langley's research program in formal methods [But91] was established to bring formal methods technology to a sufficiently mature level for use by the United States aerospace industry. Besides the inhouse development of a formally verified reliable computing platform RCP [DBC90], NASA has sponsored a variety of demonstration projects to apply formal methods to critical subsystems of real aerospace computer systems.

The Computer Science Laboratory of SRI International has been involved in the development and application of formal methods for more than twenty years. The formal verification systems EHDM and the more advanced PVS were both developed at SRI. Both EHDM and PVS have been used to perform several verifications of significant difficulty, most notably in the field of fault-tolerant architectures and hardware designs. Recently, SRI has been actively involved in investigating ways to transfer formal verification technology to industry.

Collins Commercial Avionics is a division of Rockwell International and one of the largest suppliers of communications and avionics systems for commercial transport and general aviation aircraft. Collins' interest in formal methods dates from 1991 when it participated in the MCC Formal Methods Transition Study [GBG+91]. As a result of this study, Collins initiated several small pilot projects to explore the use of formal methods, with verification of the AAMP5 microcode being the latest and most ambitious in the series.

2.2 AAMP Family of Microprocessors

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 [Roc90, BKM+82]. The AAMP architecture is specifically designed for use with block-structured, high-level languages such as Ada in real-time embedded applications. It is based on a stack architecture and provides hardware support for many features normally provided by the compiler run-time environment, such as procedure state saving, parameter passage, return linkage, and reentrancy. The AAMP also simplifies the real-time executive by implementing in hardware such functions as interrupt handling, task state saving, and context switching. Use of internal registers holding the top few elements of the stack provides the AAMP family with performance that rivals or exceeds that of most commercially available 16-bit microprocessors.

The original CAPS architecture, a multiboard minicomputer, was developed in 1972 and was quickly followed by the CAPS-2 through CAPS-10. In 1981, the original AAMP consolidated all CAPS functions except memory on a single integrated circuit. It was followed by the AAMP2, AAMP3, and AAMP5. Members of the CAPS/AAMP family have been used in an impressive variety of products as shown in Table 2.1.

The AAMP5 was designed as an object-code-compatible replacement for the earlier AAMP2 [Roc90], with advanced implementation techniques such as pipelining providing a more than threefold performance improvement. The AAMP5 is designed for use in critical applications such as avionics displays, but is not intended for use in ultra-critical systems such as autoland or fly-by-wire.

2.3 PVS

PVS (Prototype Verification System) [SOR93] is an environment for specification and verification that has been developed at SRI International's

Table 2.1: Applications of the CAPS/AAMP Family

CAPS-4	1974	Global Positioning System, General Development Model (GPS GDM)
CAPS-6	1977	Boeing 757, 767 Autopilot Flight Director System (AFDS) Lockheed L-1011 Active Control System (ACS) Lockheed L-1011 Digital Flight Control System (DFCS) NASA Fault Tolerant Multiprocessor (FTMP)
CAPS-8	1979	Boeing 757, 767 Electronic Flight Instrumentation System (EFIS) Boeing 757, 767 Engine Instrumentation/Crew Alerting System (EICAS)
CAPS-7	1979	Navstar Global Positioning System (GPS) Boeing 747-400 Integrated Display System (IDS)
CAPS-10	1979	Boeing 747-400 Central Maintenance Computer (CMC) Boeing 737-300 Electronic Flight Instrumentation System (EFIS)
AAMP1	1981	Boeing 737-300 Engine Instrumentation/Crew Alerting System (EICAS) Air Transport Traffic Collision Avoidance System (TCAS)
AAMP2	1987	Air Transport TCAS Vertical Speed Indicator (TVI) Boeing 777 Flight Control Backdrive Commercial GPS: Navcore I, Navcore II, Navcore V
AAMP3	1992	Boeing 777 Standby Instruments
AAMP5	1993	Global Positioning Systems, Upgrade for AAMP2

Computer Science Laboratory. In comparison to other widely used verification systems, such as HOL [GM93] and the Boyer-Moore prover [BM79], the distinguishing characteristic of PVS is that it supports both a highly expressive specification language and a very effective interactive theorem prover in which most of the low-level proof steps are automated. The system consists of a specification language, a parser, a typechecker, and an interactive proof checker. The PVS specification language is based on higher-order logic with a richly expressive type system so that a number of semantic errors in specification can be caught by the typechecker. The PVS prover consists of a powerful collection of inference steps that can be used to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD-based boolean simplification.

2.4 Historical Perspective and the Scale of the Challenge

Microprogram verification has much in common with processor verification, in that both relate the programmer's view of a processor to its hardware implementation. A number of microprocessor designs have been formally verified [BB94, Hun94, CJB78, Coo86, SGGH94, SB90, Win90]. 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, compared with some tens of thousands in previous formally verified designs and 3.1 million in an Intel Pentium [Int93].

Microcode verification is not new: it was pioneered by Bill Carter [LCB74] at IBM in the 1970's and applied to elements of NASA's Standard Spaceborne Computer [LCB74]; in the 1980's a group at the Aerospace Corporation verified microcode for an implementation of the C/30 switching computer using a verification system called SDVS [Coo86]; and a group at Inmos in the UK established correctness across two levels of description (in Occam) of the microcode for the T800 floating-point unit using mechanized transformations. Similarly, several groups have performed automated verification of non-microcoded processors, of which Warren Hunt's FM8501 [Hun94] and subsequent FM9000 [HB92] are among the most substantial. The problems of pipeline correctness were also studied previously by Srivas and Bickford [SB90], by Saxe and Garland [SGGH94], Burch and Dill [BD94], and Windley and Coe [WC94]. A very simple microcoded processor design developed by Mike Gordon called "Tamarack" serves as something of a benchmark for microprogram verification and was considered quite a challenge not so long ago [Joy88]. PVS is able to verify the microcode of Tamarack and Saxe's pipelined processor completely automatically in about five minutes [CRSS94].

Chapter 3

Project Goals and Organization

Formal verification of the AAMP5 microcode was selected for this project for a number of reasons. Both Collins and SRI wanted to explore the usefulness of formal verification on an example that was large enough to provide realistic insight, yet small enough to be completed at reasonable cost. Verification of the AAMP5 microcode fit these criteria well. While the AAMP5 was one of the most complex microprocessors Collins had built, its requirements were well understood since it was to be object-code-compatible with the earlier AAMP2. This allowed the formal methods team to concentrate on formal specification and verification rather than on designing a new product. Also, much of the complexity of an AAMP microprocessor resides in the microcode, and past experience has shown that this is one of the most difficult parts of the microprocessor to get right. Success with formal verification in other significant projects suggested that this technology might be ready for application to an industrial microprocessor.

Because of the importance of the AAMP5 to Collins' product line, the formal specification and verification of the AAMP5 were performed as a shadow project and did not replace any of the normal design and verification activities performed on a new microprocessor. This parallel approach also allowed us to relax some of the steps that would be required on a production project and focus instead on the application of formal methods. To fit the scope of the project to the time available, a core set of 13 instructions, each representative of a class of AAMP instructions, was identified to be specified and verified by SRI. An additional set of 11 instructions was identified to be specified and verified by Collins as time permitted. Even so, it was necessary to specify the entire AAMP5 architecture and develop the infrastructure needed to verify the entire instruction set since the core set contained at least one member from each of the major instruction classes.

Staffing for the project was provided by SRI and Collins, with funding provided by NASA and Collins. SRI provided one formal methods expert full time for the project's duration while Collins provided approximately one full-time equivalent split among several engineers. A project plan was developed at the start of the project identifying goals, strategies, tasks, and schedules. This plan was updated periodically and level of effort was recorded by both SRI and Collins personnel on each task. A summary of the level of effort is presented in Table 3.1 (page 12).

As shown in Table 3.1, relatively little time was spent on training the Collins' engineers in PVS. The small amount of structured training needed was one of the surprises of the project. Early on, SRI conducted a one-week course on the use of PVS and formal specifications at the Collins Cedar Rapids facility for the five engineers who would be involved with the project. The course consisted of five half-day lectures with related lab exercises in the afternoon. No additional formal training seemed necessary. When new team members joined the project, they were provided access to the PVS documentation and trained by inclusion in review of the PVS specifications. The most effective form of education seemed to be hands-on development with frequent peer review.

Aside from overall management and education, the project split naturally into three phases: specification of the macroarchitecture (Chapter 4), specification of the microarchitecture (Chapter 5), and proofs of correctness of the microcode (Chapter 6). The basic process followed in the first two phases was that Collins would provide design specifications to SRI, SRI would provide first drafts of PVS specifications to Collins, and Collins would informally review these specifications and return comments to SRI for revision. At some point, the Collins team would take the specifications, prepare them for formal inspections [Fag86], conduct the inspections, correct the defects found, and send the revised specifications back to SRI. This approach was chosen both to validate the correctness of the specifications and to ensure that Collins personnel became actively involved in developing the PVS specifications. A similar process was followed for performing proofs of correctness of the microcode, with SRI providing the first examples and strategies that Collins would use on similar instructions.

To reduce the potential for missing errors in the microcode due to errors in the PVS specifications, independent teams were assigned to different portions of the project. While all early drafts of the specifications were produced by SRI, different individuals at Collins were assigned to review and revise the macroarchitecture and microarchitecture specifications. Different teams were also used to inspect the macroarchitecture and the microarchitecture. The microcode itself was produced by a member of the original AAMP5 team without any knowledge of the formal specifications and translated into PVS by yet another individual. As a result, the process of proving the microcode correct often revealed errors in the specifications, but once a proof was completed, confidence in the correctness of the associated microcode was high.

Table 3.1: Level of Effort

Task	Performed	Start	Stop	Hours
Project Management				
Planning & Monitoring	Collins	Jan 93	Aug 94	123
Education				
PVS Course	Collins	Feb 93	Feb 93	125
	SRI	Feb 93	Feb 93	68
Specification of the Macroarchitecture (2,550 Lines of PVS)				
Initial Development	Collins	Mar 93	May 93	172
	SRI	Mar 93	May 93	360
Revision & Extension	Collins	May 93	Sept 93	289
	SRI	May 93	Sept 93	120
Inspection	Collins	Sept 93	Feb 94	96
Resolve Inspection Issues	Collins	Feb 94	May 94	64
Revision to Support Proofs	Collins	Mar 94	Aug 94	54
Specification of the Microarchitecture (2,679 Lines of PVS)				
Initial Development	Collins	May 93	Feb 94	137
	SRI	May 93	Feb 94	520
Revision	Collins	Feb 94	Aug 94	160
	SRI	Feb 94	Aug 94	120
Inspection	Collins	Mar 94	Aug 94	83
Resolve Inspection Issues	Collins	Mar 94	Aug 94	66
Translate Microcode to PVS	Collins	Jun 94	Aug 94	21
Revision to Support Proofs	Collins	Jun 94	Aug 94	12
Proofs of Correctness				
Development of Correctness Criteria	SRI	Mar 94	Jun 94	320
Developing Proof Infrastructure	SRI	May 94	Aug 94	240
Verification of Core Instructions	SRI	Jun 94	Aug 94	240

Chapter 4

The Macroarchitecture: The Programmer's View of the AAMP5

4.1 Overview of the AAMP Macroarchitecture

Important features of the AAMP macroarchitecture include its organization of memory, process stack, internal registers that affect its observable behavior, instruction set, and support for multi-tasking and error handling. These are discussed in the following sections. A more detailed discussion can be found in [BKM+82].

4.1.1 Organization of Memory

The AAMP provides separate address spaces for *code memory* and *data memory*. While not required by the AAMP architecture, code memory is typically implemented in ROM. Both code and data memory are segmented, with code memory organized into 512 *code environments*, each containing 64K 8-bit bytes of code. Data memory is organized into 256 *data environments*, each containing 64K 16-bit words of data. Actual memory addresses are formed by concatenating a 9-bit code environment pointer (CENV) with a 16-bit program counter for instruction addresses, or an 8-bit data environment pointer (DENV) with a 16-bit offset for data addresses. The processor also provides data-transfer status lines which can be used by an external memory management unit for protection against improper accesses to memory, allowing code fetches to be distinguished from data accesses and executive task accesses to be distinguished from user task accesses.

4.1.2 Process Stack

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, illustrated in Figure 4.1. At the top of the process stack is the *accumulator stack* used for manipulation of instruction operands and pointers. Directly below the accumulator stack is the *stack mark* of the current procedure. The stack mark contains the information needed to restore the calling procedure upon return from the current procedure, to access local variables within the calling procedure, and to locate the current procedure's header and executable code.

Below the stack mark is the current procedure's *local environment* consisting of its local variables and any parameters passed from the calling procedure. A procedure's local environment, stack mark, and accumulator stack form a *stack frame*. Beneath the current procedure's stack frame is the frame of its calling procedure, and so on. Note that the stack grows downward towards decreasing memory addresses.

4.1.3 Internal Registers

Many of the internal registers maintained by the AAMP are used to define the process stack. The DENV (data environment) register contains the pointer to the data environment of the current active task. The TOS (top of stack) register points to the topmost word in the process stack and the SKLM (stack limit) register points to the lower limit beyond which the stack is not allowed to grow. The LENV (local environment) register points to the local environment of the current procedure and is used in addressing local variables. The CENV (code environment) and PC (program counter) registers point to the code environment and address of the next instruction to be executed. Finally, the AAMP maintains two boolean flags, a USER flag to indicate whether the processor is in user or executive mode, and the INTE flag indicating whether or not the maskable interrupt is enabled (see Section 4.1.5).

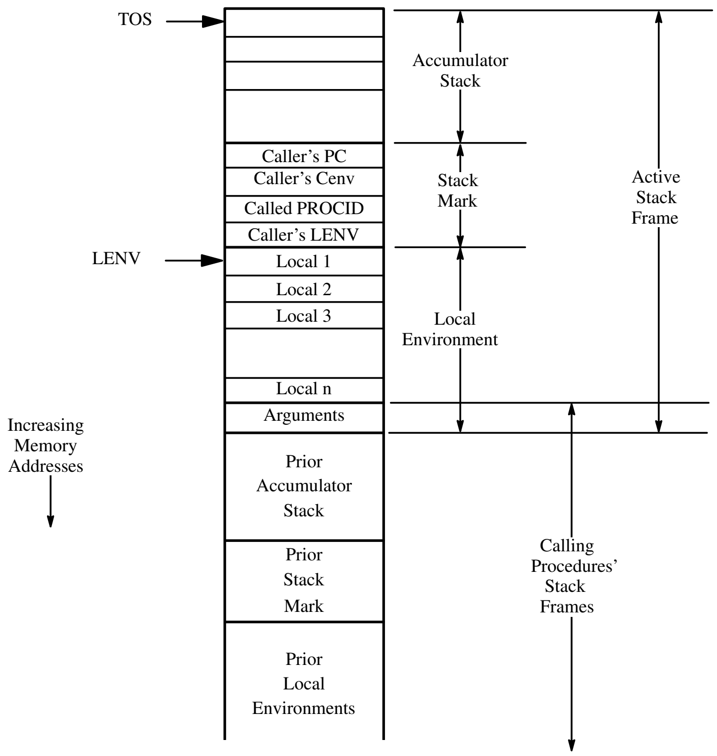


Figure 4.1: The Process Stack

4.1.4 Instruction Set and Memory Data Types

The AAMP instruction set 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, although most are only 8 bits long, 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. The floating-point formats have an 8-bit, excess-128 exponent and, respectively, 24 and 40 bits of fractional mantissa. Addition, subtraction, multiplication, division, and type conversions are provided for all of the arithmetic types. Computational exceptions, such as arithmetic overflow and divide-by-zero, are detected and handled automatically.

The original AAMP had 153 instructions. The AAMP5 has 209, which can be divided into several classes, as shown in Table 4.1. Of these, 72 are Reference or Assign instructions that move data between the top of the process stack and data memory. The Logical, Arithmetic, Relational, Type Conversion, Shift, Rotate, and Field classes account for another 81 instructions, each of which performs a prescribed operation on the top few elements of the process stack and pushes the result back onto the stack. An additional 20 instructions deal with program control, such as branch, loop, call, and return instructions. The remaining 31 instructions are used for block data memory transfers, pushing literals on the stack, locating operands in the stack, and miscellaneous functions.

Table 4.1: Instruction Classes

Instruction Class		**Number of Instructions**
Stack Management		8
Literal Data		8
Reference Data		36
Assign Data		36
Mutual Exclusion Synchronization		1
Operand Location		5
Logical		8
Arithmetic		43
Relational		11
Type Conversion		9
Shift, Rotate, and Field		10
Program Control	Branch	7
	Call	5
	Case	1
	Exception	1
	Halt	1
	Loop	2
	Return	2
	Trap	1
Block Data		12
Miscellaneous		2

4.1.5 Multi-Tasking and Error Handling

The AAMP stack architecture is designed for real-time multi-tasking applications, wherein the processor is time-shared among two or more concurrent tasks. Each task maintains its own process stack in its assigned data environment. Most multitasking operations, including context switching, are performed automatically by the AAMP. Since many errors generate a context switch to executive mode, the multi-tasking and error-handling features of the AAMP are closely intertwined.

Scheduling of user tasks is performed by an executive task assigned to code environment 0 and data environment 0. During execution, the executive selects one of several user tasks to be activated or resumed, then executes a RETURN instruction from its outer procedure. This causes the AAMP to transfer control to the selected user task. Normal execution of a user task can be suspended by an exception, an external interrupt, a trap, or an outer procedure return.

Exceptions are erroneous events, such as arithmetic overflow, that are primarily relevant to the currently executing user task. These are handled by the AAMP within the context of the user task, i.e., a context switch to the executive is not initiated. When an exception is detected, the exception error code is placed on the task's process stack and the AAMP calls the appropriate exception handler.

External hardware interrupts and traps differ from exceptions in that they are handled in executive mode and initiate a transfer of control to the executive, which schedules the appropriate interrupt or trap handler. External interrupts may arrive at any time during an instruction execution, and are recognized on master clock boundaries. Four interrupts are supported. Reset (RST) and memory transfer errors (XER) are processed immediately. The maskable interrupt (IRQ) and the nonmaskable interrupt (NMI) are held pending completion of the current instruction.

Traps are handled in a manner very similar to interrupts, initiating a context switch to the executive, which schedules the appropriate trap handler. Traps come in two varieties. A "hardware" trap is generated when the AAMP detects an error that must be processed in executive mode, such as stack overflow. "Software" traps are generated when the user process executes a TRAP instruction to request an executive service such as masking interrupts or transfering control to another user task. A user task may also suspend its execution by executing an outer procedure return. An outer procedure return from a user task is handled in the same way as a trap.

Finally, normal execution of the executive can be suspended by detection of an unrecoverable error, such as overflow of the executive stack, while in executive mode. Such errors initiate a context switch to the executive error handler. If the executive error handler is undefined or an unrecoverable error is detected while in the executive error handler, the AAMP is forced into an idle loop pending a reset.

4.2 The Macroarchitecture Specification

The macroarchitecture specification formalizes an assembly-level programmer's view of the AAMP and its instruction set, hiding most of the internal state and pipelining of the processor. The PVS specification of the AAMP models the processor as a state machine: the state of the macromachine includes external memory and the internal state that affects its observable behavior, such as the internal registers defining the process stack; the next state function specifies the effect of executing the "current" instruction pointed to by the program counter.

An overview of the *import chain* for the macroarchitecture specification is shown in Figure 4.2. In PVS, a theory gains access to another theory's definitions and axioms by *importing* that theory. Each box in the figure represents a theory in the specification. Importation of a theory is depicted by an arrow from the importing theory to the imported theory.

At the topmost level is the `normal_macro_machine` theory that defines the behavior of the AAMP in the absence of interrupts and unrecoverable errors such as stack overflow (handling of these is discussed in Section 4.3). The `next_macro_state` function for each instruction class is specified in an `update` theory for that class. Some instructions, such as the `branch` instructions, are simple enough that they can be defined directly in terms of changes to the macromachine state defined in the `macro_state` theory. More complex instructions require additional definitions provided in supporting theories. For example, the `assign` (assignment) and `ref` (reference) instructions import the `addressing` theory that determines the source or target data memory address based on the addressing mode of the current instruction. Instructions that can raise exceptions require the definitions found in the `exceptions` theory. CALL, RETURN, and instructions that

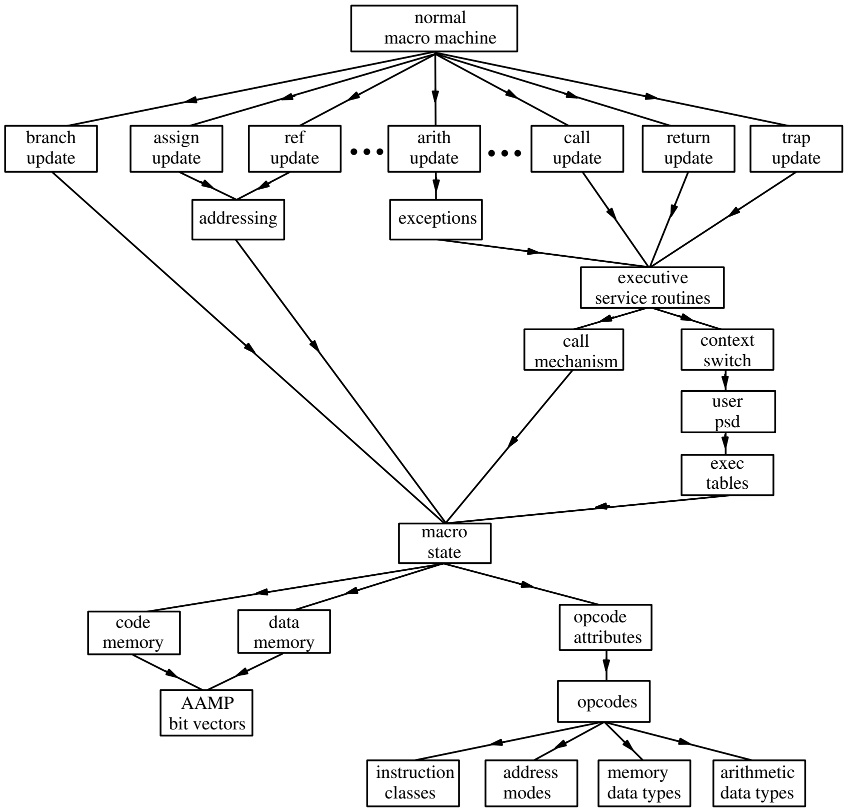


Figure 4.2: Macroarchitecture Specification Hierarchy

require executive services cause the most complex changes to the macrostate and import a family of theories defining the executive services provided by the AAMP.

The `macro_state` theory defines the state of the macromachine and functions useful in manipulating that state. This includes the internal registers that affect the observable behavior of the AAMP and instantiations of memory defined in the `code_memory` and `data_memory` theories. The `AAMP_bit_vectors` theory imports the `bit_vectors` library, adding to it the specific bit vectors found in the AAMP such as words and bytes. To simplify the import chain, the `macro_state` theory also imports the enumeration of the instructions, or `opcodes`, of the AAMP and their attributes. Representative samples of this hierarchy are discussed in the following sections, beginning with the theories lowest in the hierarchy. It should be noted that the PVS theories shown in this report have been edited to remove material not relevant to the text.

4.2.1 Attributes of AAMP Instructions

The `opcodes_attributes` theory is actually partitioned into a family of theories, one for each instruction class. The attributes of each instruction are specified as functions over the uninterpreted type `opcodes`. Specific values for each instruction are given in a theory similar to that shown in Figure 4.3. This theory defines two of the reference instructions, REFS and REFDL, as constants of type `opcodes` and defines the attributes of instruction class, memory data type, address mode, length of the instruction and immediate data in code memory, and the number of words of address information expected on the top of the stack prior to execution.

Care must be taken when using this form of specification to ensure that inconsistent axioms are not introduced. For example, it would be relatively easy to insert the wrong opcode in an axiom and inadvertently declare that opcode to have two different values for the same attribute. Originally, the opcodes category was declared as a PVS enumerated type and the attributes were specified as case analysis over the constants of the type. This form of specification ensured that PVS would detect such inconsistencies. However, early versions of PVS did not provide efficient support for large enumerated types. As the size of the macro and microarchitecture specifications grew, the definition of opcodes as enumerated types became cumbersome enough to justify switching to the declarative style shown in Figure 4.3. More recent versions of PVS have resolved this issue and provide efficient support for large enumerated types.

It is also worth noting that the preferred form of specification for the instruction attributes would be one or more "tables", with PVS internally generating the axioms describing the tables and checking to ensure their consistency. Such a construct has been added to the latest version of PVS.

```
% Defines the attributes of the reference instructions.

opcodes_attributes_reference: THEORY

BEGIN

  IMPORTING opcodes

  %===================================================================
  % REFS - Reference Absolute - Single Word
  %===================================================================
   REFS: opcodes

   REFS_class      : AXIOM instruction_class_of(REFS)      = reference
   REFS_mdt        : AXIOM memory_data_type_of(REFS)       = single
   REFS_amode      : AXIOM address_mode_of(REFS)           = absolute
   REFS_length     : AXIOM length_of_instruction(REFS)     = 1
   REFS_nstkwds    : AXIOM number_of_stack_words(REFS)     = 1

  %===================================================================
  % REFDL - Reference Local Environment - Double Word
  %===================================================================
   REFDL: opcodes

   REFDL_class     : AXIOM instruction_class_of(REFDL)     = reference
   REFDL_mdt       : AXIOM memory_data_type_of(REFDL)      = double
   REFDL_amode     : AXIOM address_mode_of(REFDL)          = local
   REFDL_length    : AXIOM length_of_instruction(REFDL)    = 1
   REFDL_nstkwds   : AXIOM number_of_stack_words(REFDL)    = 0
     ...
end opcodes_attributes_reference
```

Figure 4.3: PVS Specification of Reference Instructions Attributes

4.2.2 Data and Code Memory

The PVS specification of data memory is shown in Figure 4.4. AAMP data memory is organized into 256 possible data environments, each containing 64K 16-bit words of data. Data memory itself is defined as a function from 8-bit data environment pointers to data environments, where each data environment is another function from 16-bit data environment addresses to 16-bit words of memory. Standard bit vectors used in the AAMP, such as 16-bit words, are defined in the imported `AAMP_bit_vectors` theory, which also defines operations over bit vectors such as concatenation and extraction of bits. The specification of data memory also introduces *unspecified* constants

```
% The data_memory theory defines the AAMP's view of data memory.
% Data memory is organized into 256 possible data environments,
% each addressed by an 8-bit data environment pointer.
% Each data environment contains 64K words of data,
% each addressed by a 16-bit data environment relative address.

data_memory: THEORY
BEGIN

  IMPORTING AAMP_bit_vectors

  % A data environment address is a 16-bit word.
  data_env_addr: TYPE = word

  % An unspecified constant data environment address
  unspecified_data_env_addr: data_env_addr

  % A data environment is a function from data environment addresses
  % to words of memory.
  data_env: TYPE = [data_env_addr -> word]

  % A data environment pointer is a bit vector of size 8.
  data_env_ptr: TYPE = bvec[8]

  % An unspecified constant data environment pointer.
   unspecified_data_env_ptr: data_env_ptr

  % Data memory is defined as a function from data environment pointers
  % to data environments.
  data_memory: TYPE = [data_env_ptr -> data_env]

  % Converts a word to a data environment pointer.
  word2denv(wd: word): data_env_ptr = wd^(7,0)

  % Converts a data environment pointer to a word.
  denv2word(denv: data_env_ptr): word = zero_extend[8](16)(denv)

END data_memory
```

Figure 4.4: PVS Specification of AAMP Data Memory

of type data environment pointer and data environment address used in later theories.

The specification of code memory is similar except that code memory is organized into 512 possible code environments, each containing 64K 8-bit bytes.

4.2.3 Macroarchitecture State

The macrostate defines the minimal view of the AAMP's state an application programmer must understand to write assembly code. Choosing the best representation is important since its form heavily influences the rest of the specification. Several models were considered, each with its own advantages and disadvantages. In the end, we settled on the simple structure shown in Figure 4.5 (page 24), largely because this seemed to best reflect the view of the AAMP designers.

```
macro_state: THEORY

BEGIN

  IMPORTING opcodes_attributes, code_memory, data_memory

  % The macro state of the AAMP consists of code memory, data memory,
  % the user/executive mode indicator, the interrupt enabled indicator,
  % the cenv, pc, denv, sklm, lenv, and tos registers.

  macro_state: TYPE = [# cmem  : code_memory,
                         dmem  : data_memory,
                         user  : bool,
                         inte  : bool,
                         cenv  : word,
                         pc    : word,
                         denv  : word,
                         sklm  : word,
                         lenv  : word,
                         tos   : word #]
```

Figure 4.5: The AAMP5 Macrostate (continues)

```
  % Pushes a word on the process stack. If the stack is full,
  % tos is decremented but the word is not written.
  push(wd:word, st:macro_state):macro_state =
    IF tos(st) > sklm(st)   THEN
       st WITH [(dmem)(word2denv(denv(st)))(tos(st)-1) := wd,
                (tos)  := tos(st) - 1 ]
     ELSE
       st WITH [(tos)  := tos(st) - 1 ]
     ENDIF
                          ...
END macro_state
```

Figure 4.5: The AAMP5 Macrostate

The macrostate is defined as a record type with several fields. Code memory and data memory are defined separately, since this is the conceptual model presented to a programmer. While it is possible for an external memory unit to implement code and data memory in the same address space, and even to overlap code and data memory, code memory is normally implemented in ROM and is physically distinct from data memory.

The remaining items in the macrostate define the CENV, PC, DENV, SKLM, LENV registers and the USER and INTE flags discussed in Section 4.1.3. The macrostate theory also defines a number of auxiliary functions that manipulate the macrostate to make later theories more readable. For example, `push` pushes a word on the current process stack.

4.2.4 The Next State Function

The next state function (`next_macro_state`) takes an arbitrary macrostate and returns the macrostate that would result after the current instruction is executed. The next state function is defined for each instruction class in an `update` theory for that class. For example, the next state function for reference instructions is defined in the `ref_update` theory and for arithmetic instructions in the `arith_update` theory.

A portion of the `next_macro_state` function for the REF (reference) instructions is shown in Figure 4.6 (on page 25). REF instructions copy words from data memory to the top of the accumulator stack. The base and offset of the source data is provided by the `data_address_base`

```
% The ref_update theory computes the next macro state after a REF
% instruction.  The new macro state is returned with the addressing
% arguments popped from the stack, the data values pushed on the
% accumulator stack, and the program counter incremented to point to
% the next instruction.

ref_update : THEORY

BEGIN

  IMPORTING addressing

  % Subtype of macro state in which current instruction is a REF.
  ref_state : TYPE =
    {st:macro_state|instruction_class_of(current_opcode(st)) = reference}

  % Source word address base and offset.
  base  (st:macro_state)  : data_env_ptr  = data_address_base(st)
  offset(st:macro_state)  : data_env_addr = data_address_offset(st)

  % Number of total arguments and address arguments on the stack.
  nstackwords(st:macro_state): nat =
    number_of_stack_words(current_opcode(st))
  naddrwords (st:macro_state): nat =
     number_of_address_words(address_mode_of(current_opcode(st)))

  % Returns the next macro state on a REF instruction.
  next_macro_state(st: ref_state): macro_state =

     CASES memory_data_type_of(current_opcode(st)) OF

        % Move a single word from memory to the stack.
        single: push(data_memory_ref(st, base(st), offset(st)),
                  multipop(st, nstackwords(st))),

        % Move two words from memory to the stack.
        double: push(data_memory_ref(st, base(st), offset(st)),
                  push(data_memory_ref(st, base(st), offset(st) + 1),
                    multipop(st, nstackwords(st)))),
                           ...
    ENDCASES
      WITH [(pc) := next_pc(st)]

END ref_update
```

Figure 4.6: PVS Specification of Reference Instructions

and `data_address_offset` functions defined in the imported `addressing` theory. The auxiliary functions of `push`, `multipop`, and `data_memory_ref` (defined in the imported `macro_state` theory) are also used to define the change to the macrostate. Note that the argument to the `next_macro_state` function is of type `ref_state`. This subtype is defined immediately following the `IMPORTING` clause. Its use causes PVS to generate a *type correctness condition*, or TCC, requiring that the instruction be a reference instruction. This TCC can be discharged with the PVS theorem prover as an additional check on the consistency of the specification.

```
arith_update: THEORY

BEGIN

  IMPORTING exceptions

  % Subtype of macro state in which current instruction is an
  % arithmetic instruction.
  arith_state: TYPE =
  {st: macro_state|instruction_class_of(current_opcode(st)) = arithmetic}

  % Returns the exception number associated with an instruction.
  % Zero is used to indicate the absence of an exception.
  exception_number(st:arith_state): below[29] =
    LET nstkwds = number_of_stack_words(current_opcode(st)),
        args    = top_elements(st,nstkwds)
     IN IF current_opcode(st) = ADD THEN
             IF overflow(args(0), args(1))
                THEN 7 ELSE 0 ENDIF
       ELSIF current_opcode(st) = ADDD THEN
             IF overflow(args(1) o args(0), args(3) o args(2))
                THEN 8 ELSE 0 ENDIF
       ELSE 0 ENDIF
```

Figure 4.7: PVS Specification of Arithmetic Instructions (continues)

```
    % Returns the normal result obtained in the absence of an exception.
    % The arguments to the instruction are popped from the stack, the
    % results are pushed on the stack and the program counter advanced to
    % the next instruction.
    normal_result(st: arith_state): macro_state =
      LET nstkwds = number_of_stack_words(current_opcode(st)),
          args    = top_elements(st, nstkwds),
          popped  = multipop(st, nstkwds)
      IN IF current_opcode(st) = ADD THEN
            push(args(0) + args(1), popped)
         ELSIF current_opcode(st) = ADDD THEN
            LET result = (args(1) o args(0)) + (args(3) o args(2))
            IN push(result^(31, 16), push(result^(15, 0), popped))
         ELSE st  ENDIF WITH [(pc) := next_pc(st)]

    % Returns the next macro state for an arithmetic instruction.
    next_macro_state(st: arith_state): macro_state =
      IF exception_number(st) = 0 THEN normal_result(st) ELSE
         exception_macro_state(normal_result(st),exception_number(st))
      ENDIF

  END arith_update
```

Figure 4.7: PVS Specification of Arithmetic Instructions

A more complex example of the `next_macro_state` function is shown for the arithmetic instructions in Figure 4.7 (on page 27). Arithmetic instructions perform a specific operation, such as addition or subtraction, on the top few elements of the accumulator stack and push the results onto the stack after consuming the operands. Two functions are used in the definition of the `next_macro_state` function for the arithmetic operations. The `exception_number` function determines if the instruction will result in an exception given the current macrostate. The `normal_result` function defines the change in state returned by the `next_macro_state` function when an exception does not occur. If an exception does occur, this state and the exception number are given as parameters to the `exception_macro_state` function, defined in the `exceptions` theory, that pushes the exception number on top of the (erroneous) result and invokes the exception handler.

Similar update theories are defined for each instruction class. All the update theories are imported into the `normal_macro_machine` theory which defines the overall behavior of the AAMP in the absence of interrupts and unrecoverable user errors. The `normal_macro_machine` theory is shown in below.

```
% The normal macro machine theory defines the behavior of the AAMP in the
  % absence of interupts and  unrecoverable user errors. It computes the
  % next macro state of the microprocessor based on its current state.

  normal_macro_machine: THEORY
  BEGIN

    IMPORTING
       arith_update,       assign_update,  block_update,   branch_update,
       call_update,        halt_update,    return_update,  trap_update,
       literal_update,     misc_update,    mutex_update,   ref_update,
       relational_update,  shift_update

    next_macro_state(st: macro_state): macro_state =
      CASES instruction_class_of(current_opcode(st)) OF
          arithmetic  : arith_update.next_macro_state(st),
          assign      : assign_update.next_macro_state(st),
          block       : block_update.next_macro_state(st),
          control     : CASES control_class_of(current_opcode(st)) OF
                           branch  : branch_update.next_macro_state(st),
                           call    : call_update.next_macro_state(st),
                           halt    : halt_update.next_macro_state(st),
                           return  : return_update.next_macro_state(st),
                           trap    : trap_update.next_macro_state(st)
                        ENDCASES,
          literal     : literal_update.next_macro_state(st),
          misc        : misc_update.next_macro_state(st),
          mutex       : mutex_update.next_macro_state(st),
          reference   : ref_update.next_macro_state(st),
          relational  : relational_update.next_macro_state(st),
          shift       : shift_update.next_macro_state(st)
      ENDCASES

  END normal_macro_machine
```

4.2.5 Constructive vs. Descriptive Specifications

PVS allows a function to be specified *constructively* by explicitly defining how the result of the function is to be constructed, or *descriptively* by stating a set of properties (axioms) that the function is to satisfy. For example, the `mod` function that returns the remainder when a natural number is divided by another can be specified constructively by giving a recursive definition for it, or descriptively by stating several number theoretic properties about it.

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 disdvantage 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. In addition, descriptive specifications are sometimes more difficult to understand to an uninitiated reader than constructive specifications.

One of the early choices facing us was whether to specify the `next_macro_state` function in a constructive or a descriptive style. In addition to the reasons given above, many of the AAMP instructions are similar enough (for example, a REFS and a REFSL differ only in the addressing mode) that they could be compactly specified using a constructive style. Finally, much of the most complex behavior of the AAMP was already specified using a procedural style that could be easily translated into a constructive specification. For all of these reasons, the constructive style was initially chosen for the `next_macro_state` function.

One result of this choice was to make the effort required to specify a single instruction quite high since most of the infrastructure for an entire class of instructions was required to complete the first instruction in that class. A benevolent side-effect was that this made it easy to specify far more instructions (108) than the 13 instructions in the original core set.

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 for portions of the REFDL and ADD instructions shown below. (see Section 4.2.6 for a discussion of the `not_stack_cache_address` predicate).

```
%-----------------------------------------------------------------------
  % REFDL - Reference Local Environment, Double Word (no stack overflow).
  % The two words located at LENV+F are pushed onto the accumulator stack,
  % where LENV is the current local environment and F is the least
  % significant four bits of the current instruction byte.
  %-----------------------------------------------------------------------
  REFDL_lemma_1: LEMMA
    LET F   = current_code_env(st)(pc(st))^(3,0),
        ALS = lenv(st) + bv2nat(F),
        AMS = lenv(st) + bv2nat(F) + 1,
        XLS = current_data_env(st)(ALS),
        XMS = current_data_env(st)(AMS)
    IN current_opcode(st) = REFDL & tos(st) > sklm(st)+ 1 &
       not_stack_cache_address(st, denv(st)) (ALS) &
       not_stack_cache_address(st, denv(st)) (AMS)  =>
         normal_macro_machine.next_macro_state(st) =
           st WITH [(dmem)(word2denv(denv(st)))(tos(st)-1) := XMS]
              WITH [(dmem)(word2denv(denv(st)))(tos(st)-2) := XLS,
                    (pc)  := pc(st)  + 1,
                    (tos) := tos(st) - 2]

  %-----------------------------------------------------------------------
  % ADD - Two's Complement Add, Single Word (no exception).
  %       The two words on the top of the stack are added. The resulting
  %       sum replaces the two original values on top of the stack.
  %-----------------------------------------------------------------------
  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]
```

As these lemmas were created, it became evident that they were in many ways a preferable style of specification. They were more readable, simpler to validate, and were closer to what a user wanted to know in the first place. They also made it possible to specify a small portion of the `next_macro_state function`, i.e., to specify one instruction or part of an instruction at a time. Using this style would have made specifying the core set of 13 instructions much simpler. However, doing so also would have made it easier to introduce inconsistencies in the specification, e.g., specifying two different next states for the same current state. The standard technique for showing that a set of axioms is consistent is to prove the existence of a model satisfying those axioms, which is exactly what was done in proving that the constructive specification satisfied each lemma. The same effect could also be achieved by proving that the microarchitecture and microcode satisfy the macro lemmas. One of the more important lessons learned during this project was to more carefully consider the trade-offs between these two styles of specification. The declarative style of specification is better-suited for reasoning with complex instruction sets.

4.2.6 The Stack Cache Abstraction

As a stack machine, the AAMP performs all data computations and manipulations on operands that have been pushed onto the accumulator stack. To improve efficiency, the top few words of the stack are actually maintained in internal registers referred to as the *stack cache*. Consistency between the stack cache registers and external memory is maintained through *stack adjustments* that read additional operands into the registers or write operands out to memory prior to the execution of each instruction. Studies have shown that for over 90 percent of the instructions executed in a typical embedded application, no stack adjustments are required. As a result, this encachement technique allows the AAMP to provide performance that rivals or exceeds that of most commercially available 16-bit microprocessors.

One of the most interesting issues pertaining to the use of abstraction in specification arose in modeling the process stack. Conceptually, the process stack consists of an area of data memory, a stack limit (SKLM) register, and a *logical* top-of-stack (TOS) register. In actuality, as shown in Figure 4.8(a) (and described in Section 5.1.3), the process stack is implemented as an area of data memory, a stack limit register, a stack cache that can hold up to nine of the topmost words of the stack, and a *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. The logical TOS is the sum of the physical TOS and the contents of two registers (SV and TV) hidden from the macro level that define the number of words held in the stack cache. 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, 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

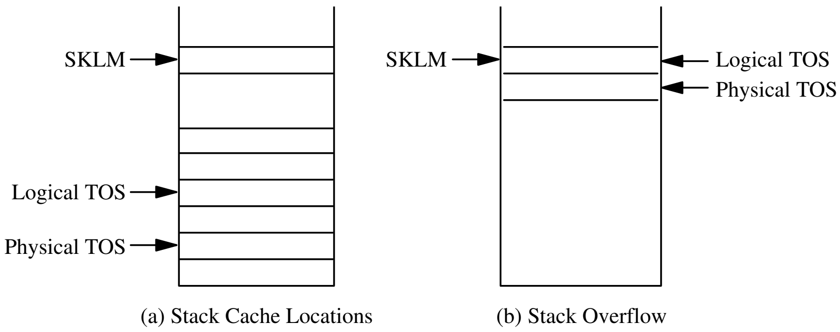


Figure 4.8: Effects of Stack Cache Abstraction

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 writers 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. For example, in a situation shown in Figure 4.8(b), where the logical TOS is at its limit (SKLM) although the physical TOS is not, the AAMP5 microcode will not signal an overflow when an attempt is made to push another word on the process stack. This is also not a problem in practice since a well behaved program should never cause a stack overflow, much less depend on one occuring precisely when the stack limit is reached. In addition, avionics applications routinely demonstrate through analysis that stack overflow does not occur.

However, 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 as shown below.

```
% A function that characterizes those addresses currently overlaid by
% the stack cache.

  not_stack_cache_address(st:macro_state, base:data_env_ptr)
                                      (offset:data_env_addr): bool

% Stores a word at a specific address. Note that the type of offsets
% is restricted to the set of words that are not overlaid by stack cache.

data_memory_assign(st     : macro_state,
                   base   : data_env_ptr,
                   offset : (not_stack_cache_address(st, base)),
                   wd     : word): macro_state =
    st WITH [(dmem)(base)(offset) := wd]

% Returns the word pointed to by a base and offset.

data_memory_ref(st     : macro_state,
                base   : data_env_ptr,
                offset : (not_stack_cache_address(st, base))): word =
    dmem(st)(base)(offset)
```

The functions `data_memory_ref` and `data_memory_assign` are used exclusively for direct data memory accesses at a given base (data environment pointer) and offset. The predicate `not_stack_cache_address` is used to define a subtype *dependent* [1] on `st` and `base` to restrict the offset to locations that are not overlaid by the stack cache. Since all direct memory references are defined in terms of these functions, the effect of reading or writing to the vicinity of the stack cache is left unspecified. This was made explicit when writing the descriptive lemmas (Section 4.2.5) by incorporating the predicate directly into the antecedent of the lemmas for the REF and ASSIGN instructions. This reflects the view of the AAMP designers and documents a contraint on use of the AAMP processor. It also relaxes the specification that must be satisfied by the microarchitecture.

The exact definition of the `not_stack_cache_address` predicate is left unspecified in the macroarchitecture since it varies with the specific AAMP microprocessor. One approach would be to be overly conservative and define it at the maximum number of words the stack cache can hold (nine for the AAMP5) beneath the logical TOS. Another approach is to define it as an abstraction of the micromachine state that contains information about the

[1] Dependent subtypes are an important PVS feature allowing a type component to depend on earlier components.

stack cache as shown below. Such a specification was used in the proofs of correctness relating the micro and macro levels.

```
not_stack_cache_address(ABS(micro_state), base)(offset) =
  (base /= denv(ABS(micro_state)))  OR  offset >= TOS(micro_state)  OR
  (offset < TOS(macro_state) - (SV(macro_state) + TV(macro_state))
```

Using the physical TOS rather than the logical TOS to detect stack overflow was a more difficult problem to resolve. After discussion, the designers of the AAMP decided they preferred to hide the stack cache as much as possible from the application programmer and that future versions of the AAMP would probably use the logical TOS rather than the physical TOS to signal stack overflow. This, combined with the use of `data_memory_ref` and `data_memory_assign` above, make it possible to restrict the effect of the stack cache on the macroarchitecture specification to introduction of the `not_stack_cache_address` predicate.

Of course, the actual microcode for the AAMP5 uses the physical TOS rather than the logical TOS to signal stack overflow. 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 micro machine. In the AAMP5 a stack overflow can occur only during a procedure call or at the beginning of an instruction during the execution of a special-purpose micro subroutine that performs stack adjustment. (Section 5.1.3). The stack overflow correctness condition can be proved as a property of the stack-adjustment micro-routine independent of the instruction being verified.

The issues raised by the stack cache are an excellent illustration of the need for formal, or at least precise, specifications. While the need for speed makes it unlikely that the AAMP will be changed to prevent direct memory accesses in the region of the stack cache, explicitly capturing this constraint provides important documentation for the users of the AAMP, particularly the compiler writers. Using the logical TOS rather than the physical TOS to signal stack overflow is a relatively minor change that may well be incorporated into future AAMPs—this issue had simply never arose since it has such a benign effect. Both issues were brought forward by the requirement to create an abstract, yet precise, specification of the AAMP macroarchitecture.

4.3 Interrupts and Unrecoverable User Errors

The AAMP5 provides four hardware interrupts, reset (RST), bus transfer error (XER), non-maskable interrupt (NMI), and maskable interrupt (IRQ). 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. An external interrupt can arrive and be acted upon by the processor not just at macroinstruction boundaries but at any point within a macroinstruction execution cycle. For this reason, a complete specification of all aspects of the behavior of the AAMP5 in response to an external interrupt is possible only in a model, eg., micromachine, where time is represented at the clock cycle level. This topic will be discussed in more detail in Section 6.6.

4.4 Development of the Macroarchitecture Specification

This section describes how the macroarchitecture specification was developed and the techniques used for its validation.

4.4.1 Initial Development

The macroarchitecture specification of the AAMP was developed through gradual refinement by SRI and Collins and resulted in several iterations, each incorporating increasing amounts of detail. Since the AAMP5 was to be object-code compatible with the earlier AAMP2, this work was based on the AAMP2 Reference Manual [Roc90]. Each iteration was reviewed via informal walkthroughs by Collins and the comments returned to SRI. This phase lasted approximately three months, took 532 man hours to complete, and resulted in a first draft of the specification consisting of 1,595 lines of PVS organized into 25 theories. Several issues emerged during this period.

As the specification grew in size, an ever increasing portion of it was devoted to defining the properties of bit vectors, i.e., sequences of bits such as words of memory and internal registers. Ultimately, these theories evolved into a reusable library of 2,030 lines of PVS organized into 31 theories. The bit-vector library was developed at NASA Langley Research Center by Rick

Butler and Paul Miner. The bit-vector library is specified as a general-purpose library parameterized with respect to the size of the bit-vectors. Availability of this library at the start of the project would have greatly shortened this phase.

Large parts of the specification were simply tables of attributes of the various AAMP instructions. While the PVS representation of this information was readable, a PVS construct explicitly designed to support the expression of tabular data would have improved their clarity. Such a construct has been added to the latest version of PVS.

4.4.2 Revision and Extension

Once SRI and Collins were satisfied with the overall structure of the specification, its completion was taken over by Collins. This was done for several reasons, the most pragmatic being to allow SRI to move on to the specification of the micro-architecture. Ownership by Collins also encouraged transfer of the formal methods technology. There was also growing concern whether the AAMP domain experts, who were not skilled in PVS, would be able and willing to read the PVS specification. It was felt the Collins team was best situated to facilitate this.

Over the next five months, the roles of SRI and Collins on the macroarchitecture were reversed, with Collins revising and extending the specifications and SRI providing informal review. More of the executive service functions were specified and the number of instructions specified was increased to 108 of the AAMP's 209 instructions. NASA Langley also took over completion and validation of the bit vector theories. To make the specifications more accessible to the AAMP domain experts, considerable effort was invested in improving their readability by choosing more meaningful names, adding general comments, adding comments tracing the specifications back to the AAMP2 Reference Manual, and ensuring that all functions were written as clearly as possible. Approximately 409 man hours were invested in this effort. At its conclusion, the macroarchitecture specification consisted of 2,550 lines of PVS organized into 48 theories, not including the bit vectors library discussed earlier.

4.4.3 Inspection

Formal inspection [Fag86] of the macroarchitecture was felt to be essential, both to validate the correctness of the specification and to familiarize more engineers at Collins with PVS. In preparation for these inspections, an overview of the specifications was presented in four reviews of two hours each. At the end of these sessions, the engineers were divided into two teams, one that would review the macro-level specifications and one that would review the micro-level specifications. Checklists were drawn up for use in the inspections based on earlier checklists used in inspecting VDM [Jon90] specifications, the RAISE Method Manual [BG90], and checklists used for code inspections at Collins.

Eleven inspections were held of the macro level specification covering thirty-one of the most important theories. Inspectors were required to review the designated theories ahead of time, using the checklists as guides, and record all potential defects encountered. Defects were classified as trivial, minor, and major. Trivial defects were defined as those that did not affect correctness and for which an obvious solution existed, such as spelling errors. Minor defects included those that might affect clarity or maintenance but did not affect correctness. As a rule of thumb, a defect was classified as minor if two reasonable people could disagree on whether it was a defect. Major defects were defined as those that affected correctness and obviously should be corrected. Despite their name, most of the major defects were very limited in scope and could be corrected in a few minutes. Some of the errors were misunderstandings by SRI, some were errors in the original AAMP documentation, and some had been inserted by Collins during the revisions. During the inspections, each inspector presented the minor and major defects they had found. While consensus of the team was required for a defect to be officially recorded, the majority of issues raised were recorded as defects. Later, each defect recorded was corrected. Total time spent in preparation by all participants, time spent in inspection, and number of defects found are shown in Table 4.2.

During the first inspection, team members were still uncomfortable with PVS, as indicated by the number of hours spent in preparation. This apprehension dissipated quickly and the inspectors settled down to a rate of approximately 150 lines of PVS and comments per hour of preparation time (the inspection rate increased during inspections nine through eleven since these were of simple tables). This rate is probably somewhat high since the inspectors were well aware that this was a shadow project. On an actual project, more preparation time would have been required. Even so, 53 minor (style) defects and 28 major (substantive) defects were discovered in specifications that had been carefully prepared prior to inspection. As shown in Table 3.1, approximately 96 hours were spent conducting the inspections and 64 hours were spent correcting the defects found.

The ease with which the inspectors became comfortable with PVS was one of the main surprises of the project. A similar result was observed with a different team on the inspections of the micro-architecture. Much of this was due to the time that was spent preparing the theories for inspection. While the purpose of the inspections was to validate the accuracy of the formal specifications, issues of style and clarify could dominate an inspection if a theory was not well organized. On the few occasions that unprepared theories were submitted to inspection, the result was quick rejection by the inspection team. Clear organization, standard naming conventions, and meaningful comments were essential.

Also surprising was the extent to which formal specifications and inspections complemented each other. The inspections were improved by the use of a formal notation, greatly reducing the amount of debate over whether an issue really was a defect or a personal preference. In turn, the inspections served as a useful vehicle for education and arriving at consensus on the most effective styles of specification. This is reflected in Tables 4.2 and 5.1 by the lower number of defects recorded in the later inspections.

Table 4.2: Macroarchitecture Inspection Results

Inspection #	# PVS Theories	Lines of PVS	Inspectors	Preparation (hours)	Inspection (hours)	Minor Defects	Major Defects
1	3	203	3	12.0	0.8	6	1
2	3	281	3	3.0	0.8	6	6
3	4	216	3	4.0	0.8	12	3
4	2	116	3	4.3	1.0	3	5
5	2	195	3	3.5	1.0	5	6
6	3	149	3	3.0	0.5	3	2
7	4	147	3	2.7	0.4	6	2
8	3	135	3	3.5	0.6	4	3
9	3	332	3	1.5	0.5	5	0
10	2	204	3	1.2	0.4	1	0
11	2	079	3	0.9	0.3	2	0
Total	**31**	**2057**		**39.6**	**6.3**	**53**	**28**

Chapter 5

The Microarchitecture: The Pipelined View of the AAMP5

The internal microarchitecture of the AAMP5 employs four major, semi-autonomous, functional units as shown in the block diagram of Figure 5.1.

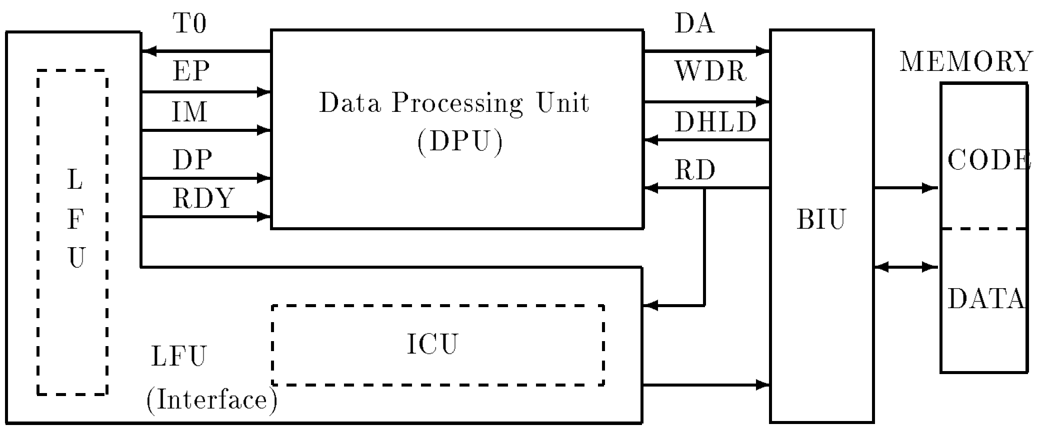


Figure 5.1: AAMP5 Block Diagram

The Data Processing Unit (DPU) provides the data manipulation and processing functions to execute the AAMP5 instruction set. Instructions are executed in a series of smaller operations in a pipeline, so that portions of multiple instructions can execute simultaneously. Included in the DPU are an ALU, combinational multiplication facilities, barrel shifter, multiport register file, and address computation hardware, all under microprogram control.

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. Each cache "hit" provides two bytes of code to the LFU in a single clock cycle. During a "miss," two bytes are fetched from external memory and provided simultaneously to the Instruction Cache Unit (ICU) storage and to the LFU for queuing. Because the operational details of ICU are hidden from the DPU, the LFU and the ICU are shown in Figure 5.1 enclosed inside a shell with which the DPU interacts directly. We have highlighted only those signals that are significant for our specification.

The Bus Interface Unit (BIU) performs external fetchs for the LFU, as well as data reads and writes initiated by the DPU. It supports 24-bits of word address, a 16-bit data bus, and several control and status signals.

5.1 The Data Processing Unit

The DPU provides the data manipulation and processing functions required to execute the AAMP5 instruction set. A block diagram of the DPU that includes most of the details of the DPU is shown in Figure 5.2, which also includes some of the details of the LFU. The DPU has two main parts: the datapath and the microcontroller.

5.1.1 The Datapath

The datapath consists of the components that perform the required data manipulation for instruction execution. It includes a 10-word multi-port

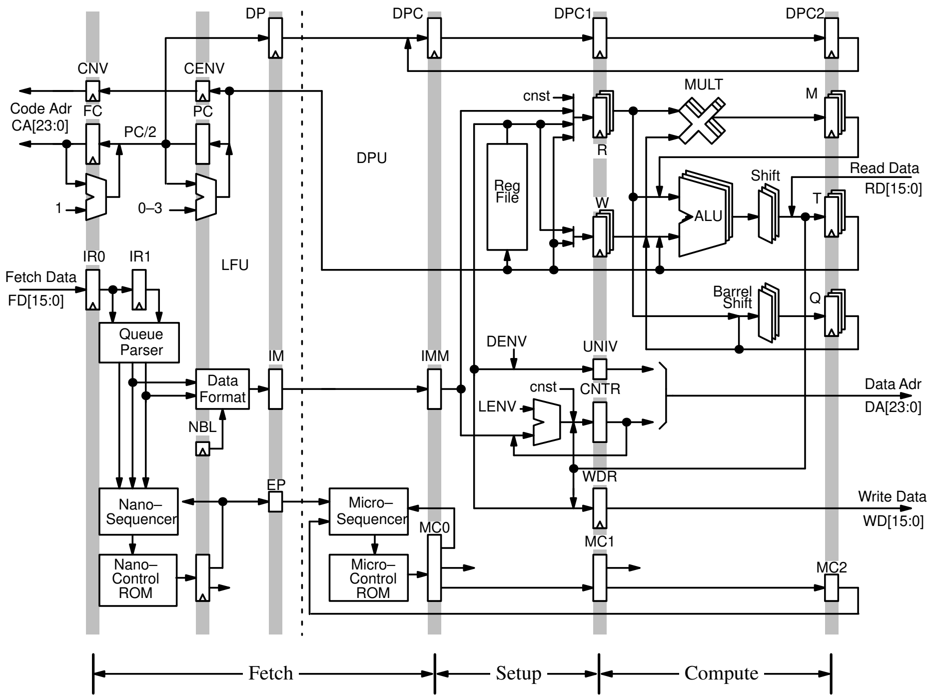


Figure 5.2: The LFU and DPU Datapath

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 5.1.3. 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).

5.1.2 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).

Once an instruction moves into MC0, it is interpreted in a two-stage pipeline The signals for controlling the operations in the first stage are generated from MC0. The operations performed in the first (*setup*) data-path stage typically include setting up the operands for the ALU operation as well as determining the next microaddress. A subset of the fields of the microinstruction in MC0 is passed onto MC1, which controls operations in the second data-path stage. The second (*compute*) stage typically involves ALU computation, register write-back and memory operations. Finally, the jump-address field of MC1 is passed onto MC2 as the target microaddress of a potential delayed jump, typically used to implement exceptions such as arithmetic overflow. 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 then the microinstruction at the jump address is fetched and loaded into MC0. 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.

5.1.3 The Stack Cache and Stack Adjustment

As mentioned in section 4.2.6, the DPU holds up to nine words of the top of the process stack in its internal registers. 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. In other words, if the latest data item pushed is a triple-word item, then it is held in T0, T1, and T2, with T0 holding the least significant part. If the data item occupies two words, then they are held in T0 and T1, with T2 being "empty." The rest of the stack cache elements are stored in the register file with the stack growing from STK0 up to STK5; i.e., STK0 is at the bottom.

The occupancy state of the stack cache that would result if an instruction were completed is maintained in a pair of registers for each of the AAMP5 instructions being executed in MC0 (SV and TV) and MC1 (SV1 and TV1). The circuit that maintains the SV-TV registers is shown in Figure 5.3. TV contains the number of words that would be occupied in the T register triplet and SV contains the number of registers in the register file that are used for the stack cache. 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.

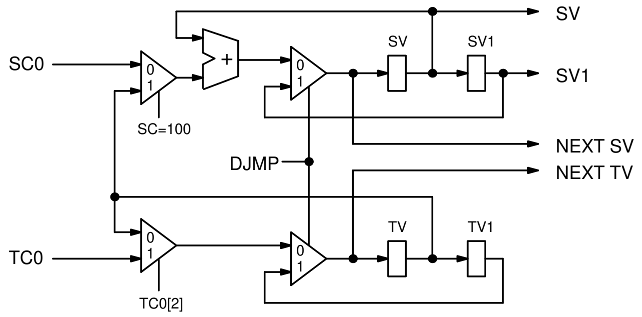


Figure 5.3: SV and TV Logic

The process of maintaining the proper number of operands in the stack cache is called a *stack adjustment.* Prior to loading into MC0 the first microinstruction of an instruction, the DPU compares the stack cache occupancy state corresponding to MC0 against the *entry condition code* of the instruction. This code, which is stored for every instruction in a read-only-memory, gives the stack cache requirements for the instruction. If the stack cache occupancy is not acceptable, the DPU automatically executes microcoded push or pop stack adjustments, causing a minimum number of 16-bit words to be written to or read from the active process stack area in memory to satisfy the entry condition of the instruction.

5.2 Formal Specification of the Microarchitecture

Since our goal was to verify the microcode residing in the DPU, the formal specification of the microarchitecture abstracts the internal details of all of the blocks of the AAMP5 except the DPU. The DPU is specified at the same level of detail given in the informal microarchitecture document [Roc93]. The rest of the AAMP5 is specified by formalizing the behavior of the other blocks expected by the DPU. The DPU directly interacts only with the LFU and the BIU. The ICU is maintained and managed entirely by the LFU and its effect on the DPU is observed only via the LFU interface. Hence, the DPU interface specification is divided into two parts: the DPU-LFU part and the DPU-BIU part.

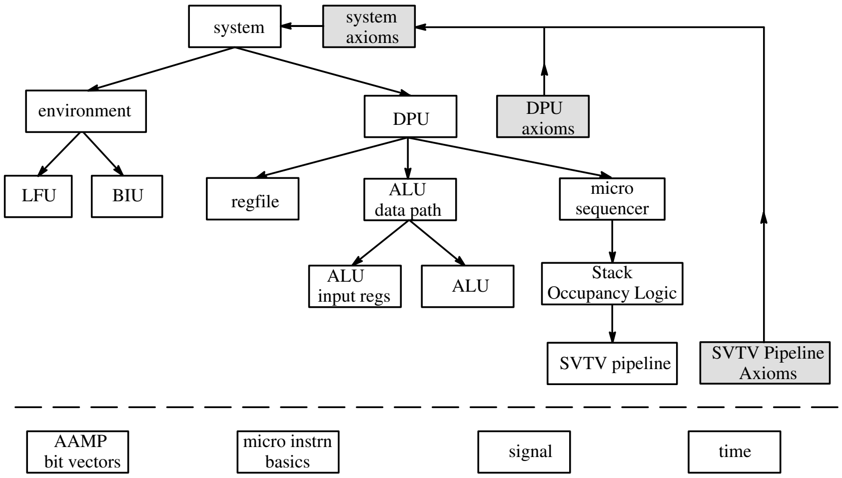


Figure 5.4: Microarchitecture Specification Hierarchy

An overview of the theory hierarchy comprising the microarchitecture specification is shown in Figure 5.4. Again, an arrow drawn from theory A to B indicates that A imports B. The hierarchy closely parallels the structural decomposition of the DPU into smaller circuit blocks in the AAMP5 microarchitecture document [Roc93]. Note that the ALU and the regfile are treated as black boxes in our specification. Every node in the graph shown in Figure 5.4 contains the PVS theories associated with a block of the circuit of the DPU. As described in the next section, each circuit block is specified by a pair of theories: one that declares the signals generated in the circuit block and one containing the axioms that constrain the signals (shown in Figure 5.4 as shaded boxes for some of the circuit blocks).

The theory-pair at the root, `system` and `system_axioms`, denotes the specification for the entire system. `System` is constructed from `environment`, which combines the DPU-BIU and DPU-LFU interface specifications, and `DPU`. The set of basic theories imported by all the theories in the hierarchy are shown below the dashed line in Figure 5.4. Note that `system` is imported by all the axioms theories thereby permitting signals generated in one circuit block at an arbitrary level to be easily used in another block. Separating the signal declarations from the axioms constraining their values made it possible to use signals globally without complicating the name structure of the signals. In the following sections, we describe the formal specifications of the DPU, the DPU-LFU interface, and the DPU-BIU interface in more detail.

5.2.1 DPU Specification

The `DPU` is constructed by combining the specifications of the datapath, namely `regfile`, `ALU_data_path`, and the microcontroller. The micropipeline part, i.e., MC0, MC1, and MC2 registers along with the abort logic, of the microcontroller is specified in the `DPU` theory. The parts of the microcontroller determining the next microaddress and the stack occupancy logic are specified in `microsequencer`, `StackOccupancyLogic` and `SVTVpipeline`. The `ALU_data_path` is decomposed into specifications of the input registers (`ALU_input_regs`) of the ALU and the ALU itself (`ALU`).

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. For the AAMP5, we used a *functional* style of specification.

In this style, 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.

This style should be contrasted with the *predicative* style [Gor85] that is commonly used in most HOL [GM93] applications. In the predicative style, every hardware component is specified as a predicate relating the input and output signals of the component and a design is specified as a conjunction of the component predicates, with signals on the internal wires used to connect the components hidden by existential quantification. We decided to use a functional style because proofs based on functional specifications tend to be more automatic than those involving predicative specifications. A proof of correctness for a predicative style of specification usually involves executing a few additional steps at the start of the proof to transform the predicative specification into an equivalent functional style. For a more detailed look at the two styles and their impact on proofs see [SSR95].

```
SVTVpipeline: THEORY

  BEGIN
  IMPORTING micro_instrn_basics, AAMP_bit_vectors, signal

  % Extract the required control fields from MC0
  SC0: signal[SC_field]
  TC0: signal[TC_field]

  SV, SV1, TV, TV1: signal[bvec[3]]

  NEXT_SV: signal[bvec[3]]
  NEXT_TV: signal[bvec[3]]

  END SVTVpipeline
```

Figure 5.5: Signals for the SV-TV Pipeline

As an illustration, a complete specification in PVS of the circuit block (shown in Figure 5.3) that updates the pipelined stack pointer registers SV, TV, SV1, and TV1 is given in Figures 5.5 and 5.6. The specification of a circuit block is divided into two theories: a *signals* theory and an *axioms* theory. The signals theory (Figure 5.5) declares all the signals used to represent the state of the circuit. For the SV-TV circuit, these consist of the inputs (the control signals SC0 and TC0 originating from the microinstruction register MC0) and the outputs at the various registers used in the circuit. We also include NEXT_SV and NEXT_TV as signals since they are also used as outputs of the circuit.

The types of all the signals in the design, except the one used for microinstructions, are bit-vectors of the actual size used in the design. An AAMP5 microinstruction is 88 bits wide organized into 27 different fields. In our specification, a microinstruction is represented symbolically as a record type with enumerated types for the microinstruction fields. The enumerated type associated with a microinstruction field contains a distinct literal for every bit-pattern value possible for the field. The microinstruction record type, the default values for the fields, and related functions are specified in the theory `micro_instrn_basics`, which is one of the basic theories imported by the `SV_TV_pipeline` theory.

The axioms theory (Figure 5.6) for the circuit defines the values of the declared signals. Every signal is specified by means of an axiom that constrains the way the values of the signals change over time. For example, the axiom TVax specifies the value of the signal TV to be equal to that of NEXT_TV delayed by one cycle unless the updating of the signal is disabled when data hold (DHLD) is asserted to be high.

The axioms and the signals theory import a number of other theories besides `micro_instrn_basics` described above. `AAMP_bit_vectors` defines the different kinds of bit-vectors used in the specification. `System` is the top-level theory that imports all the signals declared in different parts of the circuit so that a signal generated in one circuit block can easily be used as an input in another block. For example, the signals DJMP and MC0 used here are generated in another part of the circuit defined by the `DPU` theory. For the sake of brevity, certain parts of a circuit, usually those denoting a combinational block, are abstracted in a single signal specification. For example, the axiom NEXT_SVax combines the entire combinational logic in the circuit shown in Figure 5.3 that generates the NEXT_SV signal.

5.2.2 DPU-LFU Interface

The DPU and the LFU undertake a "hand-shake" protocol to accomplish the proper transfer of information between them. The LFU independently prefetches instructions into a queue, decodes the instruction in the front of the queue, and presents the decoded information to the DPU via the signals EP, DP, and IM described below. The LFU signals the availability of a new instruction on its output lines by making RDY true. Whenever RDY is true, the LFU ensures that the following conditions hold:

```
SVTVpipeline_axioms: THEORY

  BEGIN
  IMPORTING system
  IMPORTING micro_instrn_basics, signal, AAMP_bit_vectors

  t: VAR time

  % Extract the required control fields from MC0
  SC0ax: AXIOM SC0(t) = SC(MC0(t))
  TC0ax: AXIOM TC0(t) = TC(MC0(t))

  NEXT_SVax: AXIOM NEXT_SV(t) =
    (IF DJMP(t) THEN SV1(t)
       ELSE CASES SC(MC0(t)) OF
         NO_SV : SV(t),
         SPUSH1: SV(t) + 1,
         SPUSH2: SV(t) + 2,
         SPUSH3: SV(t) + 3,
         SPOP1 : SV(t) - 1,
         SPOP2 : SV(t) - 2,
         SPOP3 : SV(t) - 3,
         SPUSHT: SV(t) + bv2nat(TV(t))
       ENDCASES
     ENDIF)

  SVax: AXIOM SV(t+1) = IF DHLD(t) THEN SV(t) ELSE NEXT_SV(t) ENDIF

  SV1ax: AXIOM SV1(t+1) = IF DHLD(t) THEN SV1(t) ELSE SV(t) ENDIF

  NEXT_TVax: AXIOM NEXT_TV(t) =
     IF DJMP(t) THEN TV1(t)
     ELSIF TV_TO_0?(TC(MC0(t))) THEN nat2bv[3](0)
     ELSIF TV_TO_1?(TC(MC0(t))) THEN nat2bv[3](1)
     ELSIF TV_TO_2?(TC(MC0(t))) THEN nat2bv[3](2)
     ELSIF TV_TO_3?(TC(MC0(t))) THEN nat2bv[3](3)
     ELSE TV(t) ENDIF

  TVax: AXIOM TV(t+1) = IF DHLD(t) THEN TV(t) ELSE NEXT_TV(t) ENDIF

  TV1ax: AXIOM TV1(t+1) = IF DHLD(t) THEN TV1(t) ELSE TV(t) ENDIF

  END SVTVpipeline_axioms
```

Figure 5.6: Specification of the SV-TV Pipeline

1. EP has the entry-point (starting microaddress) of the microcode of the instruction associated with the value of DP.

2. IM has the first unit of immediate data, if any, of the instruction associated with the value of DP.

DP is an offset (actually one greater than the real offset since it's "delayed" by a byte for technical reasons) into the code environment from which the instruction is fetched. The real address of the instruction is formed as a concatenation of the code environment pointer in CENV, which resides in the LFU (but can be read as a register in the register file), and DP.

Once RDY becomes true, the LFU idles, i.e., does not change its outputs, although it may be filling up the instruction queue. The idling state continues until the DPU expresses its intent to consume, or sends a request to the LFU to discard the current instruction and start fetching from a different target address. The DPU controls the operation of the LFU by means of the NC field of the microinstruction in MC0. A value of CLR_RDY for NC means the DPU is ready to consume a new instruction and will cause the LFU to assemble information about the instruction following DP and present it the next time RDY becomes true. However, if the DPU instructs the LFU to retarget its fetching (NC is NANO_SKIP, for instance) then the new target address (presented by the DPU in T0) gets transferred into the LFU. After that, the LFU presents the instruction at the new target address the next time RDY becomes true. For the protocol to work correctly, the microcode in the DPU must obey the following conditions:

1. The DPU must consume a new piece of information from the LFU only when RDY is true. That is, if the DPU is ready to issue a CLR_RDY signal to the LFU when the LFU is not ready with the required information, the DPU must wait until RDY is true.

2. Every issue of a request to change the normal sequence of instruction fetching, such as NANO_SKIP, NANO_CALL, etc., must be followed by a CLR_RDY. For example, if a NANO_SKIP is issued before the results of a previous request are consumed, the second request may override the first one.

Our LFU-DPU interface specification states only those assumptions we made about the behavior of the LFU in our proofs. It is not a complete

```
LFU: THEORY
  BEGIN
  IMPORTING micro_instrn_basics, AAMP_bit_vectors, signal
  IMPORTING opcodes, temporalops, MEMORY, system, EPROM_basics

  t, t1, t2: VAR time

  NCO(t): NC_field = NC(MCO(t))
  normal_fetching: signal[bool]

  % Invariant constraints on the outputs when LFU is RDY
  RDYinv: AXIOM
     RDY(t) =>
       LET opcode = opcode_at(CENV(t), DP(t), CODE_MEMORY(t))
       IN  EP(t) = EP_ROM(opcode) &
            (has_imm_data(opcode) => IM(t) = first_unit_of_imm_data(t))

  % CLR_RDY behavior specification
  CLR_RDY_no_wait: AXIOM
     RDY(t) & CLR_RDY?(NCO(t)) =>
       normal_fetching(t+1) &
        DP(t+1) = IF RDY(t+1) THEN DP(t)+length_of_instruction(opcode)
                  ELSE DP(t) ENDIF

  CLR_RDY_wait: AXIOM
     NOT RDY(t) & normal_fetching(t) & CLR_RDY?(NCO(t)) =>
        (EXISTS (t1 | t1 > t):
            stays_same(NCO(t))(t, t1)  =>
              stays_same(RDY)(t, t1-1) & RDY(t1) &
                 normal_fetching(t1) &
                   DP(t1) = DP(t)+length_of_instruction(opcode) )

   % SKIP behavior specification
   no_change_in_flow(t): bool = ((NCO(t) = CLR_RDY) OR (NCO(t) = NO_NANO))

   NANO_SKIPax: AXIOM
     NANO_SKIP?(NCO(t1)) =>
       NOT RDY(t1+1) & NOT(normal_fetching(t1+1)) &
         (EXISTS (t2 | t2 > t1+1):
             stays_high(no_change_in_flow)(t1+1, t2-1)) =>
                stays_same(RDY)(t1+1, t2-1) & RDY(t2) &
                  stays_same(normal_fetching)(t1+1, t2) & DP(t2) = TO(t1) )

     % Idling behavior of LFU
     LFUidles(t1, t2): bool =
      stays_same(EP)(t1, t2)
        & stays_same(IM)(t1, t2) & stays_same(RDY)(t1, t2) &
          stays_same(DP)(t1, t2) & stays_same(CENV)(t1, t2)

    NO_NANOax: AXIOM
      RDY(t1) & stays_equal_to(NCO, NO_NANO)(t1, t2) => LFUidles(t1, t2)

  END LFU
```

Figure 5.7: DPU-LFU Interface Specification

specification of the LFU interface. Section 6.7 discusses a possible approach for validating these assumptions. Figure 5.7 shows part of the LFU interface specification.

CODE_MEMORY, declared in the theory MEMORY, denotes the portion of the external memory where the code is stored. Although the AAMP5 microarchitecture does not require external memory to be separated into mutually exclusive code and data areas, our specification uses two distinct memory objects, CODE_MEMORY and DATA_MEMORY. Both of these memories map 24-bit addresses to 16-bit words. All instructions are assumed to be fetched from CODE_MEMORY and all data read/writes are assumed to only affect DATA_MEMORY. That is, we assume that the CODE_MEMORY state remains constant. This simplification restricts the validity of our proofs of correctness of microcode to programs that are not self-modifying.

The first axiom (`RDYinv`) states a condition on the LFU outputs that the DPU can rely on whenever LFU is ready with a decoded instruction. It states that whenever RDY is true, EP and IM have the appropriate information pertaining to the instruction associated with the program counter DP.

The remaining axioms specify the expected responses to some of the different instruction-fetching requests originating from the DPU. The axioms beginning with the prefix CLR_RDY specify the possible LFU behaviors in response to a request (CLR_RDY) by the DPU to consume an instruction from the LFU. If the LFU is ready with an instruction when the DPU issues a CLR_RDY, which is the case covered by the first axiom, the LFU goes into a state `normal_fetching` where it presents instructions sequentially. Note that the DP is updated only if the next instruction can be presented in the following cycle (otherwise, the invariant `RDYinv` will not be preserved). If the LFU is not ready with an instruction when NC is CLR_RDY, the case covered by the second axiom in the set, we assume that the LFU will eventually present the DPU with a new instruction, provided DPU holds the NC0 line stable with CLR_RDY.

The behavior specified by the second axiom applies only when the LFU is in a `normal_fetching` state. We do not need an analogous axiom for the case when LFU is not in a `normal_fetching` state because the LFU can get into such a state only after a DPU request, such as NANO_SKIP, that causes a change in the flow of instruction fetching. The requirement that RDY must eventually become true in such a state is covered by the axioms constructed for those other DPU requests, as we have shown for NANO_SKIP in Figure 5.7. In addition to ensuring that RDY eventually becomes true, the NANO_SKIP axiom updates DP to the target address (T0) and makes the `normal_fetching` signal false. It guarantees the outcome only if the DPU does not issue another request for a change of instruction flow. That is, the DPU must keep the NC signal input to the LFU either NO_NANO or CLR_RDY until RDY becomes true for proper action.

The axiom `NO_NANOax` states that the LFU idles, i.e., the values of EP, IM, DP and CENV stay the same, once RDY becomes true as long as NC0 stays NO_NANO.

5.2.3 DPU-BIU Interface

```
BIU : THEORY

  BEGIN
    IMPORTING system
    IMPORTING micro_instrn_basics, AAMP_bit_vectors, signal
    IMPORTING MEMORY, temporalops

  t, t1, t2: VAR time

  data_mem_unchanged: AXIOM
     NOT WRITE(t+1) => DATA_MEMORY(t+2) = DATA_MEMORY(t+1)

  read_complete_delayed: AXIOM
      READ(t1) & DHLD(t1) =>
         EXISTS (t2 | t2 > t1):
             NOT DHLD(t2) & stays_high(DHLD)(t1, t2-1) &
                RD(t2) = DATA_MEMORY(t1)(DA(t1)) &
                  DATA_MEMORY(t2) = DATA_MEMORY(t1)

  write_complete_delayed: AXIOM
    WRITE(t1) & DHLD(t1) =>
      (EXISTS (t2 | t2 > t1): NOT DHLD(t2) &
              stays_high(DHLD)(t1, t2-1) &
                   DATA_MEMORY(t2) =
                      DATA_MEMORY(t1) WITH [(DA(t1)) := WDR(t1)] )

  END BIU
```

Figure 5.8: DPU-BIU Interface Specification

The DPU interacts with the BIU (Bus Interface Unit) to accomplish data reads and writes to external memory. In addition to the logic needed to accomplish data transfers to the external memory, the BIU also contains logic to arbitrate between the memory requests from the DPU and the LFU. The BIU is specified similar to the LFU by stating a set of assumptions made about the behavior of the interaction between the BIU and the DPU. Memory access requests (READ and WRITE) by the DPU are controlled by the DB field of the microinstructions in MC0 and MC1 as well as the abort logic in the DPU. Figure 5.8 shows a portion of the text of the theory containing the DPU-BIU interface specification. The 24-bit data address for the memory operation is output on the line DA. WDR contains the data for a memory write and RD is used to store the read data from the memory. The BIU indicates the completion of memory operation by means of the data hold (DHLD) signal. Once a memory access request is made by the DPU, DHLD is held true until the memory operation is complete.

The first axiom states that DATA_MEMORY remains unchanged unless there is a pending write request from the DPU. The second axiom asserts the guaranteed completion of every read request, while the third states a similar requirement for proper completion of a write request.

5.3 Development of the Microarchitecture Specification

This section describes how the microarchitecture specification was developed and the techniques used for its validation.

5.3.1 Initial Development

Development of the microarchitecture specification mirrored that of the macroarchitecture. As indicated in Table 3.1, initial development of the microarchitecture specification by SRI took approximately 657 man hours over 10 months. The specification closely followed the block structure of the microarchitecture, usually with one theory per component. Surprisingly, the specification of the microarchitecture without the PVS version of the microcode was only slightly larger than the specification of the macroarchitecture, an indication how much of the complexity of the AAMP5 is contained within the microcode. Once decisions about the data types to be used to model signals and states are made, it should be possible to automatically derive a PVS specification of a hardware design from its description in a more traditional hardware description language.

Completion of the microarchitecture specification took longer than the specification of the macroarchitecture for a number of reasons. The AAMP5 microarchitecture document, unlike the AAMP2 Reference Manual, is targeted for an audience that is familiar with the basic architecture of the AAMP processor family. As a result, SRI had to spend considerable time becoming familiar with the design. In particular, the interactions between the DPU and its environment were difficult to specify. Although the design of the LFU and the BIU were well documented, the interface conditions that the DPU has to obey to ensure proper LFU and BIU services were not explicitly stated. This information had to be extracted through reverse engineering of the detailed designs and extensive discussions with the Collins staff. The time spent on both these efforts could have been substantially reduced if formal specification activity were integrated earlier and more closely into the conventional design cycle.

5.3.2 Revision

To make the specifications acceptable to the AAMP5 designers for inspection, the initial microarchitecture specifications developed by SRI were informally reviewed by three Collins engineers familiar with both PVS and the AAMP5 and revised as was done for the macroarchitecture. Since the initial specifications covered the entire microarchitecture, there was no need for Collins to extend the specification. Most of the changes consisted of modifying names to reflect local conventions, adding comments tracing back to the design documents, and improving the clarity of the specifications. Even so, this was a sizeable effort, requiring 280 hours spread over seven months. When completed, the microarchitecture specification consisted of 2,679 lines of PVS and comments organized into 20 theories.

Revisions were also made later to the microarchitecture specifications to facilitate proofs, but these were more technical in nature and not as significant as the ones made to the macroarchitecture specification. Most involved trading the use of an advanced and expressive construct of the specification language for a more basic construct to improve the efficiency of proofs.

5.3.3 Inspection

Formal inspections of the microarchitecture were conducted just as for the macroarchitecture. To maximize the independence between the macro and micro specifications, only one participant from the macroarchitecture inspections was included in the microarchitecture inspection team. Ten inspections were held, including two reinspections, covering 15 of the most important theories. The results are shown in Table 5.1.

Again, the inspectors quickly adapted to PVS, reaching an average inspection rate of approximately 290 lines of PVS and comments per hour. Interestingly enough, the designers of the AAMP5, who were the least familiar with the PVS language, found the specifications the simplest to read, consistently turning in the most major defects and the lowest preparation times. This was a direct result of their detailed knowledge of the AAMP5 and the close correspondence between the AAMP5 design and the specifications. As with the macroarchitecture specification, more preparation time would have been required on an actual project. Sixty-four minor (style) defects and 19 major (substantive) defects were discovered. As shown in Table 3.1, 83 hours were invested in conducting the inspections and 66 hours were spent correcting the defects found.

Table 5.1: Microarchitecture Inspection Results

Inspection #	**# PVS Theories**	**Lines of PVS**	**Inspectors**	**Preparation (hours)**	**Inspection (hours)**	**Minor Defects**	**Major Defects**
1	3	357	5	8.3	1.5	15	6
2	2	173	5	3.9	1.0	11	2
3	1	146	4	3.2	0.4	7	1
4	1	146	5	3.3	0.8	6	2
5	1	152	5	4.2	0.3	3	4
6*	1	160	4	1.3	0.5	1	0
7	1	272	4	3.6	0.6	10	1
8*	1	423	5	4.6	0.8	4	2
9	3	197	4	2.0	0.5	6	1
10	3	256	4	1.6	0.5	1	0
Total	**17**	**2282**		**36.0**	**6.9**	**64**	**19**

* Reinspection of previously inspected theory

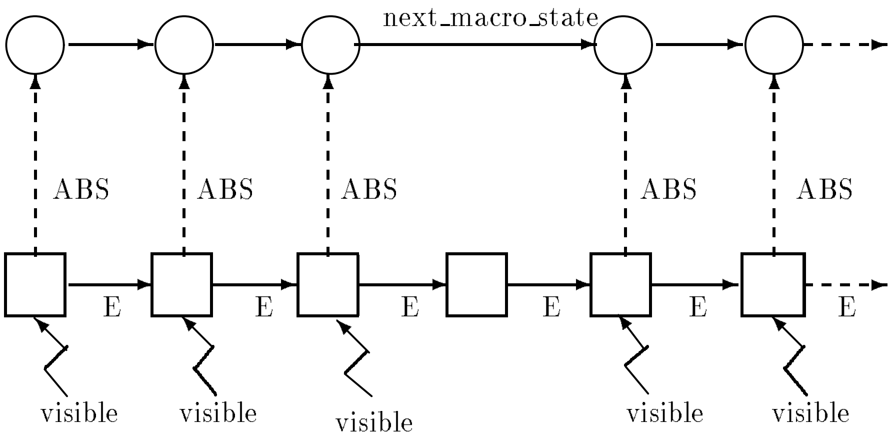
Chapter 6

Formal Verification of AAMP5

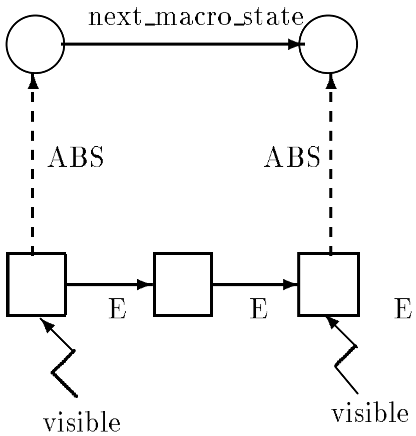
In this chapter, we first describe the general correctness criterion that is commonly used for microprocessor verification. We then describe the impact of pipelining and asynchronous memory interaction on the correctness criterion and the proof of correctness. That is followed by a description of the AAMP5 pipeline operation and how the general microprocessor correctness criterion is applied to the AAMP5 pipeline. We then describe how we mechanized the verification of the AAMP5 and some of the errors discovered during the mechanization of the proof. We end the chapter with a discussion of the scope of the formal verification that has been undertaken so far.

6.1 General Microprocessor Correctness

In most mechanical verifications of microprocessors, the general correctness criterion used is based on establishing a correspondence, as shown in Figure 6.1(a), between the execution traces of two state machines that the specifications at the macro and micro levels denote. In the figure, E and `next_macro_state` denote the micromachine and macromachine state transition functions, respectively, and the circles and squares denote the macro and micromachine states, respectively. The objective of the correctness criterion is to ensure that the micromachine does not introduce any behaviors not allowed by the macromachine. The macromachine uses two kinds of abstraction to hide details of the micromachine:



(a) Infinite Trace Correspondence



(b) Commutes Property

Figure 6.1: General Microprocessor Correctness

- *Representation* abstraction: Not every component of the micromachine state is visible at the macro level. Even the visible part of the micromachine state can take on quite a different form at the macro level. For example, the process stack, which is viewed as residing entirely in the data memory in the macromachine, is actually split between the memory and internal registers in the micromachine.

- *Temporal* abstraction: The macro and the micromachines do not run at the same speed. For example, while every instruction in the macromachine is viewed as being executed atomically in one unit of time, the same instruction may take a number of clock cycles to complete in the micromachine. The difference in speed means that the micromachine trace may have "intermediate" states for which there are no corresponding macromachine states.

To deal with representation abstraction it is necessary to define an *abstraction* function (ABS) that returns the macrostate corresponding to the state of the micromachine at any given time. This function must be surjective to ensure that if the macromachine starts at some initial state, the micromachine is able to start at a corresponding state. One way to deal with the consequences of temporal abstraction is to characterize the set of microstates that have corresponding macrostates as *visible* states. For example, a possible definition of a visible state is one in which an instruction gets completed and a new instruction begins. In Figure 6.1(a), the visible states in a trace are those in which an arrow labeled ABS is drawn from the micromachine trace to the macromachine trace.

Given the notions of ABS and visible state, the correctness criterion described by Figure 6.1(a) can be stated as follows: "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 formally stated below and depicted graphically in Figure 6.1(b). The commuting property captures the correspondence between the traces for one step of the macromachine, i.e., between two successive visible states. The formalization of the commuting diagram has two additional preconditions to handle the consequences, described in section 4.2.6, 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. `No_logical_stack_overflow(t)` ensures that the logical process stack would not overflow as a result of executing the current instruction.

```
        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))
```

David Cyrluk has developed a general framework [Cyr93] for proving correspondence between state machines in PVS and has shown its application for pipelined microprocessors. The characterization of microprocessor correctness described here is similar to the visible state approach given there [Cyr93]. Windley [Win90] also develops a general framework for microprocessor correctness, although it is not applicable to pipelined processors.

6.2 Impact of Pipelining and Asynchronous Memory Interface

The basic idea behind pipelining is to increase the instruction execution rate of a processor by executing different stages of more than one instruction within the same clock cycle. Thus, although execution of an instruction is still spread over multiple clock cycles, it is possible to complete up to one instruction every cycle.

The general correctness criterion shown in Figure 6.1 is still applicable for pipelined processors with some adjustment as long as instructions get completed in the same order as they enter the pipeline. An adjustment is needed to handle the fact that when a new instruction enters the pipeline, the results of the previous uncompleted instructions may not yet be at their destinations. The definition of the abstraction function ABS may have to peek to a future microstate, possibly nonvisible, following the current visible state (as shown in Figure 6.2) 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.

In principle, it should be possible to define the abstraction function as a function of the current visible state alone because the information necessary to compute the result of the to-be-completed instructions is stored in various hidden registers in the micromachine. In the MiniCayuga verification [SB90], which is one of the earliest efforts in the mechanical verification of a pipelined processor, the abstraction function was defined in this fashion. However, it is easier and conceptually clearer to define the abstraction in a "distributed fashion" using the future visible or nonvisible states, which is the approach taken for the AAMP5. The pipelined verifications described in [Cyr93], [SGGH94], [TK93], and most recently [WC94], also define the

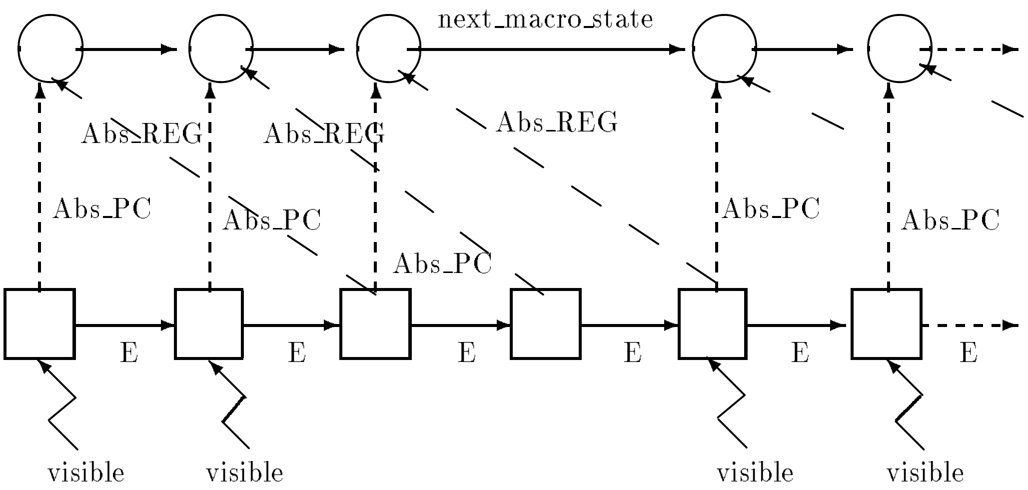


Figure 6.2: Impact of Pipelining on Abstraction

abstraction function in a distributed fashion. Burch and Dill use a slightly different approach in [BD94] to handle the time skew between the two levels. They "run" the micromachine longer by an appropriate number of cycles by streaming in NOP instructions before relating the states of the macro and micromachines.

In Figure 6.2, the distance between two consecutive visible states can vary for a number of different reasons, although a pipelined design strives to keep the distance to one cycle as often as possible. For example, an instruction requiring a memory access may take longer than a register-to-register instruction, or an instruction that signals an exception may take a few extra cycles for completion. The distance can usually be expressed as a function of the class of the new instruction entering the pipeline in the visible state. For the AAMP5, however, the distance can be arbitrary in some situations because the DPU relies on two semi-autonomous units (the LFU and the BIU) for instruction and data fetches. While we do rely on having the LFU and the BIU guarantee correct services, we cannot make any assumptions about the time taken for those services (other than that they are finite) for two reasons. First, the internal details of the LFU and BIU have been abstracted in the micromachine specification. Second, the time of service is determined by factors that are not completely under the control of the DPU.

6.3 The AAMP5 Pipeline

The reader is advised to refer to Figure 5.2 in section 5.1 while reading this section. The AAMP5 executes its instructions in a three-stage pipeline where we identify the stages with the names: *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. The setup stage consists of the activities controlled while the microinstruction is in MC0. The compute stage consists of the activities determined by the part of the microinstruction that moves from MC0 to MC1. Note that the register MC2 is only used for delayed jump execution and does not add a stage under normal instruction execution.

The different components of an AAMP5 instruction enter the DPU through the following input signals: DP, which is the program counter value associated with the instruction entering the DPU; EP, which gives the entry point, i.e., the beginning microaddress, of the microcode of the instruction; and IM, which is the first unit of immediate data, if any, of the instruction. A summary of the activities that take place in each of the three pipeline stages is given below.

Fetch Stage:

- Read the microinstruction at EP from ROM.
- Read the entry condition code for the instruction from SROM (not shown in Figure 5.2).
- Perform any stack adjustments required.

Setup Stage:

- Compute the stack cache occupancy state (NEXT_SV and NEXT_TV) that would result if the instruction were performed.
- Read the register file to determine the sources for the ALU input registers R and W.
- Compute memory sources and destination addresses (needed only for memory access instructions).

Compute Stage:

- Compute the results of the ALU.

- Check whether there is going to be a *delayed jump*, i.e., an exception, such as an arithmetic overflow, that arises as an outcome of the ALU computation.
- Initiate writing into the stack cache.
- Initiate memory read/write (needed only for memory access instructions).

6.3.1 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.

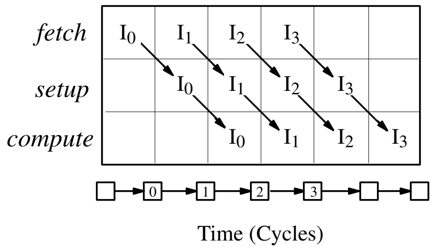


Figure 6.3: Normal Pipeline Operation

The normal pipeline operation along with the execution trace of the micromachine is shown in Figure 6.3. A state (square) in the execution trace (shown at the bottom of Figure 6.3) 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.

We define a 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. A formal characterization of a visible state is given below. `Visible_state_with(op)(t)` defines the condition that must be satisfied for a micromachine state to be visible with the opcode of the current instruction being `op`. The first condition in the conjunction ensures that MC0 contains the first microinstruction of the microcode corresponding to `op`. The remaining conditions ensure that the previous instruction will complete in the next cycle: DHLD being false ensures that the previous instruction will not be held due to data memory transfer; `nextDJMP(t)` (which is the value of the DJMP register one cycle later) being false ensures the previous instruction will not cause a delayed jump; `entry_cond_met` ensures that the entry conditions of the instruction in MC0 are met by the stack occupancy state determined by the SV1 and TV1 registers.

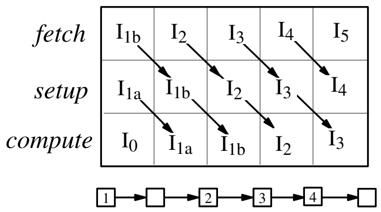
```
visible_state_with(op: opcodes)(t): bool =
  MC0(t) = ROM(zero_extend[8](11)(EP_ROM(op)(t))) &
     & NOT(DHLD(t)) & NOT(nextDJMP(t)) & entry_cond_met(op)(t)

visible_state(t): bool =
    EXISTS (op: opcodes): visible_state_with(op)(t)
```

6.3.2 Pipeline Stalling and Delayed Jumps

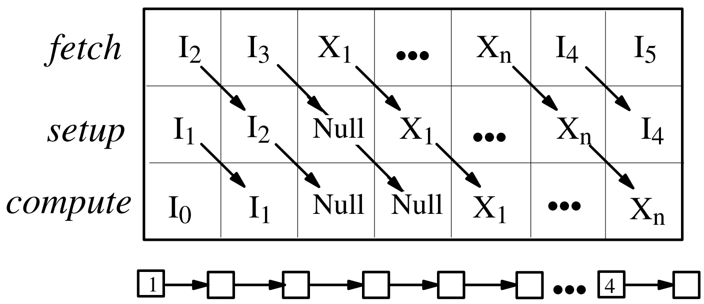
Given that the AAMP5 pipeline is at a visible state, the pipeline operates normally (i.e., the distance to the next visible state is one cycle) only if the following conditions hold:

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 distance to the next visible state will be more than one cycle and its value will depend on which of the conditions in the list are violated. We classify these situations in which the next visible state is more than one cycle away from the present one as follows: The behavior resulting from violations of conditions 2(b), 3, or 4 is referred to as pipeline *stalling*; in these situations, the micromachine performs a computation that has no visible effect on the macrostate. Violations of the other two cases (1, 2(a)) are referred to as pipeline *extension*. In a pipeline extension, the micromachine components that affect the macrostate are updated incrementally in two or more cycles. In all these situations, the compute stage of the previous incomplete instruction is always completed in the following cycle. Some of the execution traces for pipeline extensions are shown in Figure 6.4 and those for stalls are shown in Figure 6.5. While both figures show the traces when the respective situations arise alone, in practice any combination is possible.



(a) Multi–instruction microcode trace



(b) Execution trace for a delayed jump

Figure 6.4: Execution Traces for Pipeline Extension

Figure 6.4(a) shows the trace for the case where the size of the microcode is 2. Note that, in general, the distance to the next visible state in this case is equal to the size of the microcode for the instruction. The stages associated with the two microinstructions of the current instruction are labeled (1-a) and (1-b), in order.

Figure 6.4(b) shows the trace when the current instruction causes a delayed jump. The delayed jump is detected only after the current instruction (I1) is in its compute stage. Once a delayed jump is detected, the MC0 and MC1 registers are cleared to a NULL microinstruction, which behaves like a NOP. Any memory operation initiated in the compute stage is aborted when a delayed jump is detected. The next visible state is reached only after the preprocessing required to set up the exception handler is completed. In Figure 6.4(b), the microinstructions that process the delayed jump are labeled X1 through Xn. I4 denotes the first instruction of the exception handler.

Figure 6.5(a) shows the trace when the pipeline has to be stalled while the DPU waits (that is, the DHLD input to the DPU is true) for completion of a data memory access initiated by the current instruction. In this case, after the compute stage of the previous instruction in the pipeline is completed, the micromachine state is held constant until the next visible state, when DHLD becomes false again. The distance to the next visible state is $n + 1$, where n is the duration for which DHLD is held true, which can be arbitrarily long.

Figure 6.5(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. As in the case of waiting for DHLD, the distance to the next visible state can be arbitrarily large depending on when RDY becomes true again.

Finally, Figure 6.5(c) shows the trace when the entry of the next instruction has to be delayed for a stack adjustment. In this case, the hardware causes a jump to a special microroutine that moves the appropriate number of elements between the stack cache and memory. Here, the distance to the next visible state is a function of the number of elements that need to be moved. In Figure 6.5(c), SADJi denotes microinstruction `i` of the stack adjustment routine.

6.4 Our Approach to Verification

To prove the general correctness criterion for the AAMP5, shown in Figure 6.1(b), we have to first define the abstraction function. The components

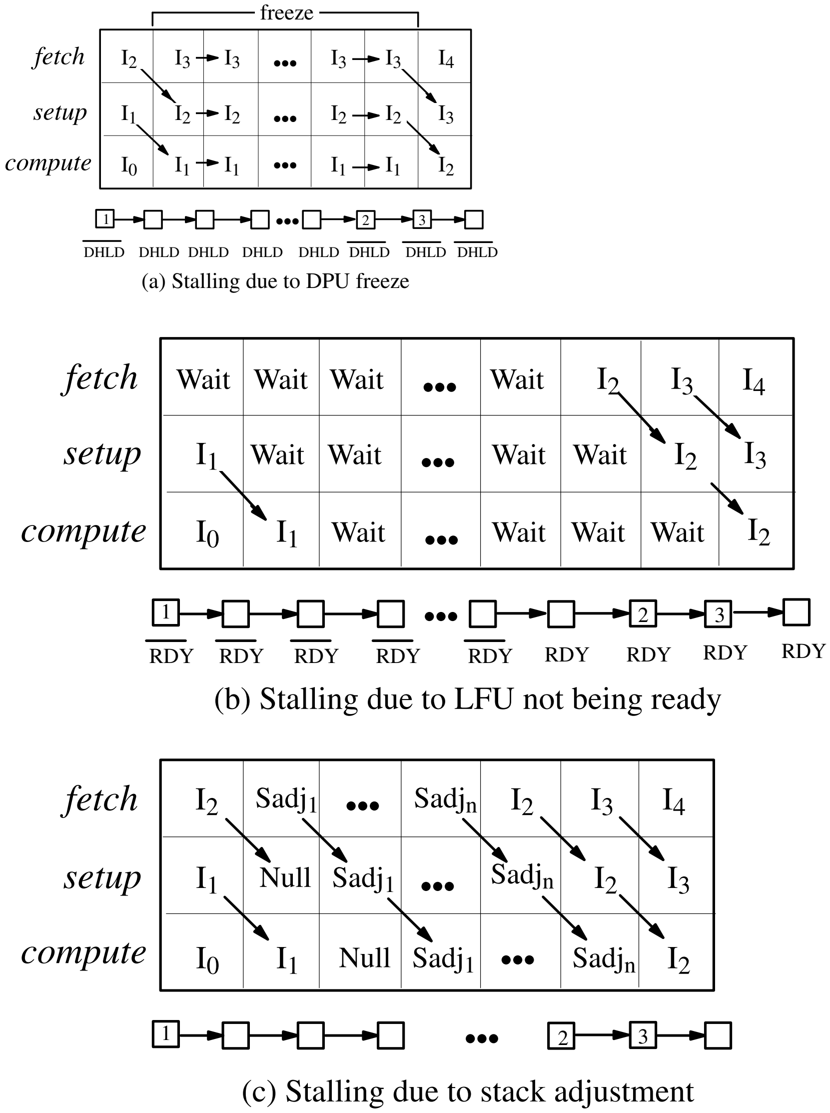


Figure 6.5: Execution Traces for Pipeline Stalling

of the micromachine state that are relevant in determining the corresponding macrostate are the following:

1. The special registers (inside the register file) DENV, CENV, and SKLM. These map to their corresponding counterparts in the macrostate.

2. The special register TOS in teh register file, and the registers TV1 and SV1, which are part of the stack occupancy logic shown in Figure 5.3. These define the value of TOS in the macrostate.

3. DATA_MEMORY and CODE_MEMORY

4. The DPC register. This is mapped to the macrostate program counter.

5. The registers T0, T1, T2, and the STKi registers in the register file, where `i` ranges from 0 through SV1. These determine the contents of the macrostate data memory that are overlaid by the stack cache.

A formal definition of the abstraction function for AAMP5 is shown below. The abstraction is divided into a separate function for each of the components in the macrostate. Given a time `t`, an abstraction function returns a component of the macrostate corresponding to the microstate at `t`. Every abstraction function except the one associated with the program counter actually constructs the macrostate component from the micromachine state one cycle later to account for the pipeline latency. The definition given below is applicable to only those situations where the process stack at the macro level does not cause an overflow. As was explained in section 4.2.6, the correctness of the microcode in the presence of stack overflow is shown as a separate property.

```
ABSpc(t): word = DPC(t)

stack_cache_size(t): nat = bv2nat(SV1(t)) + bv2nat(TV1(t))

ABStos(t): word = TOS(t+1) - stack_cache_size(t+1)

ABSsklm(t): word = SKLM(t+1)

ABSdenv(t): word = DENV(t+1)

ABSlenv(t): word = LENV(t+1)

ABScenv(t): word = CENV(t+1)

i: VAR nat
ith_top_of_scache: [time -> [nat -> word]]
ith_top_of_scache: AXIOM i < stack_cache_size(t)  =>
 ith_top_of_scache(t)(i) =
  IF i < bv2nat(TV1(t)) THEN IF i = 0    THEN T0(t)
                             ELSIF i = 1 THEN T1(t)
                             ELSE T2(t) ENDIF
  ELSE REG(t)(stack_cache_size(t)-i-1) ENDIF

ABSdmem(t)(dptr: data_env_ptr)(daddr: word): word =
   IF bv2nat(daddr) < bv2nat(TOS(t+1)) &
        bv2nat(daddr) >= bv2nat(TOS(t+1)) - stack_cache_size(t+1)
      THEN LET offset = stack_cache_size(t+1)
                             - (bv2nat(TOS(t+1)) - bv2nat(daddr))
           IN ith_top_of_scache(t+1)(offset)
      ELSE DATA_MEMORY(t+1)(dptr o daddr) ENDIF
```

Every macrostate component, except TOS and `dmem`, has a direct correspondence with a micromachine component. Since the elements of the stack cache are viewed as being in the data memory at the macro level, the macro TOS is smaller than the micro TOS by the size of the stack cache, i.e., (SV1 + TV1). The locations of `dmem` correspond to the locations in DATA_MEMORY except for those that range from the micromachine (TOS - 1) to (TOS - (SV1+TV1)).

6.4.1 The Verification Conditions

A common strategy used to prove the commuting property of Figure 6.1(b) is to start from an arbitrary visible state at the lower left-hand corner of the commuting rectangle, traverse the two possible paths to the top right-hand corner of the rectangle, 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 6.3, 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.

We used a bottom-up incremental approach to mechanizing the verification of the correctness criterion. In this approach, we formulate a set of properties, called *verification conditions*, about the execution traces of the micromachine that can originate from a visible state under restricted scenarios. The verification conditions are formulated so that the commuting property can be proved by combining the verification conditions using higher-level proof techniques, such as induction and case-splitting, without having to reason with the AAMP5 microcode. The formulation of the verification conditions reflects the initial case analysis that must be performed in constructing a proof of the commuting property to determine the number of cycles the micromachine must be run in the proof.

The main reason for taking the bottom-up approach was that the macro specification, which was primarily written at Rockwell, was still evolving when the verification task was begun at SRI. Also the amount of rewriting that would have to be done in a top-down undecomposed approach would strain the capacity of the PVS rewriter. This increased demand on the rewriter arises in part from the constructive style of specification that the macro specification was initially written in, as described in section 4.2.5. The constructive style increased the depth of the hierarchy in the definition of the next state function of the macromachine.

The decision to take a bottom-up approach, while chosen for technical reasons, had several beneficial consequences. The proofs of the verification conditions can be constructed fairly automatically because the crucial case-splitting that determines the number of cycles on the micromachine has already been done. The Rockwell engineers related much more readily to the verification conditions than to the general correctness criterion. But, most important, the approach provided a convenient method for partial verification of a complex and big design, such as the AAMP5. It was the quickest and, perhaps, the only way we could have obtained useful feedback from the verification exercise within the time allotted for the project.

6.4.1.1 Instruction-Specific Verification Conditions

The verification conditions are organized into a number of sets, with a separate set for each specific instruction and a general set of that consists of properties common to all instructions. In this section, we discuss the instruction-specific verification conditions using the ADD instruction for illustration. The set of general verification conditions in discussed in Section 6.4.1.2.

The verification conditions[1] for the ADD instruction, assuming there will be no exception, is shown in Figure 6.6. The exception case has a separate set of verification conditions. A number of observations are in order regarding the verification conditions.

The verification conditions state the effect on the state of each of the micromachine components relevant to the macrostate by the execution of an instruction, in this case ADD. In Figure 6.6, the predicate `visible_state_with(ADD)(t)` (defined in Section 6.3.1) is included as a precondition to the verification conditions because we are interested in analyzing properties of execution traces that start from a visible state where the instruction of interest is the current instruction. 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.

The other precondition, `SysInv`, in the verification conditions consists of a set of conditions that are expected to hold in every visible state of the

[1] The PVS text of the verification conditions and other formalizations shown in this report are a slightly edited version of the PVS theories actually used in the proofs.

```
% Verification conditions for ADD in the absence of exception
ADD_correctness: THEORY

  BEGIN

  IMPORTING global_assumptions, invariants, reg_file_theorems,
            next_instrn_correctness_general, ADD_correctness_setup,
            bv_rules, more_bv_rules

 t, t1, t2: VAR time

SVTV_correctness: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) =>
     bv2nat(SV1(t+2)) = bv2nat(SV1(t+1))-1 &
           bv2nat(TV1(t+2)) = bv2nat(TV1(t+1))

T0_correctness: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) =>
      T0(t+2) = (sign_extend[16](48)(ith_top_of_scache(t+1)(1))
                  + sign_extend[16](48)(ith_top_of_scache(t+1)(0)))^(15,0)

i: VAR below[10]
REG_correctness: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) =>
                 REG(t + 2)(i) = REG(t + 1)(i)

DATA_MEMORY_correctness: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) =>
     DATA_MEMORY(t+2) = DATA_MEMORY(t+1)

next_macro_instrn_entry: LEMMA
   visible_state_with(ADD)(t) & SysInv(t) & NOT Soverflow(t + 1) =>
     (EXISTS (tp: time | tp > t):
        DPC(tp) = DPC(t) + length_of_instruction(opcode_at_DPC(t)) &
           visible_state_with(next_opcode_at_DPC(t))(tp) & SysInv(tp) )

  END ADD_correctness
```

Figure 6.6: Verification Conditions for ADD

micromachine. `SysInv`, whose definition is shown below, is also included as a postcondition in the `next_macro_instrn_entry` verification condition to ensure that it is preserved at the next visible state.

```
% DJMP and SADJ flip-flops can't be high at instruction entry point
% for, if they were, then MC0 would be null_MC0, which is not the case
% at an instruction entry point.
  DJMPinv_ep(t): bool = NOT DJMP(t)
  SADJinv_ep(t): bool = NOT SADJ(t)

  % Constraints on stack occupancy registers SV and TV
  SVinv(t): bool = bv2nat[3](SV(t)) <= 6
  TVinv(t): bool = bv2nat[3](TV(t)) <= 3

  SysInv(t): bool =
    SADJinv_ep(t)& DJMPinv_ep(t) & SVinv(t) & TVinv(t)
```

`SVTV_correctness` states that SV1 should be decreased by one and TV1 should remain unchanged, which reflects the fact that ADD, by popping two elements and pushing one element, decreases the net size of the process stack by one. Since the number of microinstructions executed for ADD is one, in the absence of exception, all the results (except for the next instruction moving in) would be at their destinations two cycles from the initial visible state. Thus, every verification condition, except `next_macro_instrn_entry`, relates the state of a component at `t+2` with those at `t+1`. The reason for using `t+1` instead of `t` is to adjust for the one cycle latency for the abstraction function.

`T0_correctness` states that the T0 register, which holds the top-of-stack, must contain the result of adding the top two elements of the current process stack. The function `ith_top_of_scache(t)(i)`, which returns the element at a given distance `i` from the top of the stack cache state at time `t`, is used to fetch the process stack elements. The required arguments for ADD are guaranteed to be resident in the stack cache because the entry condition for the instruction is satisfied in a visible state.

`REG_correctness` states that the register file must remain unchanged as a result of ADD. This condition ensures two requirements on the normal execution of ADD: (1) The contents of none of the special registers (SKLM, TOS, DENV, CENV) ought to change, and (2) the contents of the stack cache registers unused by the ADD instruction remains unchanged. `DATA_MEMORY_correctness` requires that the contents of the physical memory remain unchanged. The only time a stack instruction, such as ADD, can affect physical memory is as a result of a stack adjustment performed prior to the entry of the instruction. In our approach, the correctness of stack adjustment, which is described is section 6.4.1.2, is proved separately.

The `next_macro_instrn_entry` verification condition ensures that the next instruction, i.e., the first microinstruction of the next macroinstruction, moves into MC0 and the processor enters a visible state. Since the distance to the next visible state is indefinite depending on whether and how long the pipeline will be stalled (see section 6.3.2), this verification condition, unlike the rest of the verification conditions for ADD, has to be expressed as an eventuality property using an existential quantifier. To prove this verification condition, it is necessary to reason about the stalling behavior of the pipeline. Since the behavior of the pipeline during stalling is uniform over all instructions, we separated their formalization into the general verification conditions discussed in the next section. The general verification conditions are used as lemmas in proving `next_macro_instrn_entry`.

6.4.1.2 General Verification Conditions

The general (i.e., instruction-independent verification) verification conditions characterize the behavior of the micromachine when the AAMP5 pipeline stalls. The general verification conditions that characterize three possible ways in which the pipeline can stall are shown in Figure 6.7.

The `DPU_freeze_lemma` asserts that the state of every micromachine register, including those in the register file, remain unchanged as long as DHLD is true. The `LFU_induced_stalling` lemma states a property about the MAP_WAIT-loop microcode that the DPU executes while the LFU is not ready with the next instruction. It asserts that if (1) MC0 has the last microinstruction of an arbitrary macroinstruction (`MAP?(MC0(t))`) and (2) the LFU is not RDY, then the next time when RDY becomes true, the relevant information about the next macroinstruction will move into MC0 and DPC. It also asserts that during the time the DPU is waiting for RDY to become true, the state of every component (except DPC) relevant to the macrostate remains unchanged. This lemma also includes several ancillary conditions that were brought out during the proof process. For example, NOT `nextDJMP(t)` and NOT `nextDJMP(t+1)` are imposed as preconditions to the lemmas because the MAP_WAIT-loop will be entered only when a delayed jump does not occur. `NOT SADJcondn(t+i+1)` is a precondition to the lemma since the next macroinstruction will move into MC0 when the

```
  all_but_DPC_and_dmem_stays_same(t1, t2): bool =
    SV1(t1) = SV1(t2) & TV1(t1) = TV1(t2) &
      SKLM(t1) = SKLM(t2) & TOS(t1) = TOS(t2) & LENV(t1) = LENV(t2) &
        DENV(t1) = DENV(t2) & CENV(t1) = CENV(t2)

  all_but_data_mem_stays_same(t1, t2): bool =
    DPC(t1) = DPC(t2) & all_but_DPC_and_dmem_stays_same(t1, t2)

  % Stalling due to DHLD
  DPU_freeze_lemma: LEMMA
    stays_high(DHLD)(t1, t2) =>
      all_but_data_mem_stays_same(t1, t2+1)

 % Stalling due to LFU not being ready
 LFU_induced_stalling: LEMMA
  MAP?(MCO(t)) & NOT RDY(t) & stays_low(RDY)(t, t+i) & RDY(t+i+1)
   & NOT nextDJMP(t) & SysInv(t) & NOT nextDJMP(t+1) &
       NOT SADJcondn(t+i+1)=>
         MCO(t+i+2) = ROM(zero_extend[8](11)(EP(t+i+1))) &
            DPC(t+i+2) = DP(t+i+1) &
              all_but_DPC_and_dmem_stays_same(t+1, t+i+2) &
                NOT nextDJMP(t+i+1) & NOT nextDJMP(t+i+2) & SysInv(t+i+2)

  % Stalling due to stack adjustment
  stack_adjust_lemma: LEMMA
    MAP?(MCO(t)) & RDY(t) & SADJcondn(t) & NOT nextDJMP(t) &
      NOT stack_overflow_condn(t)
         => EXISTS (tp: time | tp > t):
               CENV(tp) = CENV(t) &
               DPC(tp) = DP(t) &
               MCO(tp) = ROM(zero_extend[8](11)(EP(t))) &
               entry_cond(next_opcode_at_DPC(t))(tp) &
               ABSdmem(tp) = ABSdmem(t) &
               NOT nextDJMP(tp) & SysInv(tp)

  % A general lemma about fetching next instruction
  guaranteed_fetch_of_next_instrn: LEMMA
    NOT nextDJMP(t) & NOT nextDJMP(t+1) & SysInv(t) &
          MAP?(MCO(t)) & NOT stack_overflow_condn(t)
      => (EXISTS (tp: time | tp > t):
            DPC(tp) = DPC(t) + length_of_instruction(opcode_at_DPC(t)) &
            MCO(tp) = ROM(zero_extend[8](11)
                                   (EP_ROM(next_opcode_at_DPC(t)))) &
            entry_cond(next_opcode_at_DPC(t))(tp) &
            SysInv(tp) & NOT nextDJMP(tp) )
```

Figure 6.7: General Verification Conditions

MAP_WAIT-loop is exited, i.e., at `t+i+2`, only if the next macroinstruction does not require a stack adjustment. The lemma also ensures that the invariants will be true at `t+i+2`.

The `stack_adjust_lemma` states a property about the behavior of the execution of the stack adjustment microcode. It asserts that if (1) MC0 has the last microinstruction of a macroinstruction (`MAP?(MC0(t))`) and (2) the LFU is RDY with the next instruction, and (3) that instruction needs a stack adjustment (`SADJcondn(t)`), then the micromachine will eventually reach the state in which the following conditions hold:

1. CENV and DPC point to the next instruction.
2. The first microinstruction of the next macroinstruction is back in MC0.
3. The entry condition (`entry_cond`) for the opcode associated with the macroinstruction is met.
4. The data memory as viewed by the macromachine (`ABSdmem`) is unchanged.

As before, this lemma also includes several preconditions that define the context in which the stack adjustment microroutine is entered. We have proved this lemma only in the absence of a stack overflow. (Section 6.7 discusses the scope of the verification work completed so far.) These lemmas can be combined to prove a general lemma `guaranteed_fetch_of_next_instrn` that guarantees the entrance of the first microinstruction of the next macroinstruction, whenever the DPU is at the end of the microcode of an arbitrary macroinstruction.

6.4.2 Bridging the Micro-Macro Gap

The following proof tasks have to be performed to establish the relationship characterized by the commuting diagram shown in Figure 6.1(b):

1. *Micro-leap*: Prove the existence of a common time, the next visible state point, at which all of the verification conditions associated with a particular instruction will hold simultaneously.

2. *Macro-elevate*: Use the abstraction function to prove that the expected values, guaranteed by the verification conditions, for the macro-relevant micromachine components indeed correspond to the macrostate conjectured by the `next_macro_state` function.

It is important to observe that neither of the above tasks require any reasoning about the AAMP5 microcode. The need for a Micro-leap proof arises primarily to relax the assumption that the DPU is never stalled for data memory accesses. A Micro-leap proof involves performing a case analysis on the number of times the DPU would have to be stalled before the next visible point is reached. Each of these cases can be discharged using the general verification condition for the appropriate stalling condition.

Macro-elevate involves applying the abstraction function to elevate the microstate to the macrostate as well as unwinding the `next_macro_state` function. The proof effort (rewriting) involved in applying the abstraction function is not particularly complex because the abstraction function definition, except for data memory, is straightforward for AAMP5. The effort required to rewrite the definition of `next_macro_state`, however, is quite complex. This complexity is partly due to the constructive style chosen for the macro specification, which introduced a significant number of intermediate definitional layers, and partly because the specification was not written in a fashion amenable to efficient rewriting. As a result, we decided to decompose the Macro-elevate step by first formulating a set of *macro lemmas* that characterized the value returned by `next_macro_state` for each of the restricted scenarios for which a separate set of verification conditions was set up. The macro lemma for the ADD instruction for the case where the instruction does not yield an exception is shown in Section 4.2.5 on page 4.2.5.

6.5 Mechanization of Proofs of Verification Conditions

In an interactive theorem prover, such as PVS, proof of a goal (i.e., a formula) is constructed by interactively (or automatically) invoking inference steps to simplify the given goal into simpler subgoals until all the subgoals are trivially true. At the highest level, the user directs the verification process by elaborating and modifying the specification, providing relevant lemmas, and backtracking on the fruitless paths in a proof attempt. Given our style of hardware specification, proof of a formula relating two states of a state machine that are a fixed distance apart usually follows a standard pattern consisting of a sequence of proof tasks shown below. Note that every verification condition in Figure 6.6 except the last one is a formula of this form.

Quantifier elimination: Eliminate (skolemize) the universally quantified variable (`t`) by *skolemization* and simplify the preconditions. Skolemization consists of replacing the universally quantified variable by a new constant symbol denoting an arbitrary value for that variable. This technique is a simple and general way to prove a property for all values in a set that a variable ranges over.

Unfolding definitions: Simplify selected expressions and defined function symbols in the goal by rewriting using definitions, axioms or previously established lemmas in the micromachine specification.

Case analysis: At the end of the unfolding step, the original goal will have been simplified to an equation on two nested `IF-THEN-ELSE` expressions, not necessarily identical, involving user-defined as well as primitive function symbols. The `IF-THEN-ELSE`s are introduced by the unfolding of the defined function symbols. To prove such an equation, it may be necessary to split the proof, based on selected boolean expressions in the current goal, and further simplify the resulting goals.

The complexity and the degree of automation of the proof required to perform the above tasks depend on the level and efficiency of the primitive proof steps supported by a theorem prover. If the primitive inference steps are powerful and efficient, then one can use more brute-force and automatic strategies to perform the unfolding and case-analysis steps. If not, it is necessary to exercise more intelligent manual control in the proof to keep the size and number of cases small. The primitive inference steps of PVS are implemented using powerful and efficient decision procedures for linear arithmetic, equality on uninterpreted function expressions, and boolean tautology checking. The primitive steps also use an efficient conditional rewriter that interacts with the decision procedures to simplify conditions. The following short PVS proof strategy, called the *core* strategy, can accomplish the above proof tasks:

```
1: ( then*
2:        (skosimp*)
3:        (auto-rewrite-micro-theories)
4:        (repeat (assert))
5:        (apply (then* (repeat (lift-if))
6:                      (bddsimp)
7:                      (assert))))
```

*Then** on Line 1 is a strategy constructor that composes a sequence of commands. The proof command on Line 2 performs the skolemization task. The command on Line 3 makes rewrite rules out of all the definitions and axioms in our specification, after which, the command on Line 4 rewrites all the expressions in the goal until no further simplification is possible. `Assert` is the PVS primitive command that performs rewriting as well as simplifications using arithmetic and equality decision procedures. In the case of our verification conditions, this rewriting step has the effect of reducing all expressions into ones that involve only values of signals in the initial state. The compound proof step on Lines 5 through 7 performs the case-analysis and further simplification task. The case analysis is performed by lifting all the `IF-THEN-ELSE` structures to the top and then simplifying the resulting expression propositionally (`bddsimp`). (*Apply* applies a compound proof step as an atomic step.)

We have used the core strategy (or slight variants of it) to prove completely automatically the correctness of several microprocessor designs that have served as informal hardware verification benchmarks for theorem provers. See Cyrluk et al., [CRSS94] for more details about application of this strategy for hardware verification. The core strategy was, however, not adequate to automate proofs of the AAMP5 verification conditions. The core strategy is usually successful in constructing a proof if the rewriting step simplifies the original goal into a relation on expressions involving either operations on primitive PVS types that the decision procedures are capable of handling or user-defined functions that can be left uninterpreted.

The AAMP5 specification, both at the micro and macro levels, uses a number of complex bit-vector operations, such as concatenation (`o`), subsetting (`^`), shifting, etc. All bit-vector operations are specified as parameterized functions that are defined recursively in the bit-vector library. The core strategy, when unsuccessful in proving a verification condition, simplifies the original goal into equations on bit-vector expressions. These expressions can be shown to be equivalent only by using properties about the bit-vector operations as lemmas. To automate the bit-vector equivalence checking step of the proof, we formulated a number of bit-vector lemmas in the form of rewrite rules. Most of these lemmas have been proved as theorems by Rick Butler using the bit-vector library. A subset of these rules can be used to first attempt to simplify the bit-vector expressions into a common normal form. If the rules are unsuccessful in normalizing the expressions into a common form, then another set of rules can be used to convert the bit-vector equivalence into an equivalence on natural numbers that can most likely be handled by the decision procedures.

A significant portion of our verification effort was devoted to formulating the bit-vector simplification rules. The rules formulated are parameterized with respect to the size of the bit-vectors involved, but are not complete enough to decide equivalence for all possible bit-vector expressions. They were able to successfully decide equivalence of expressions in most of our proofs. Even when they were unsuccessful, we had to perform only a few standard case-splits manually to complete the proof.

The core strategy can automate the proofs of only those verification conditions relating micromachine states that are a fixed distance apart. The general verification conditions do not fall into the above category. The general verification conditions are proved using induction on the length of the execution trace that the verification conditions constrain. For example, the `stack_adjust_lemma` has to be proved by induction on the number of stack cache elements that have to be moved to satisfy the entry condition. Once the proper induction scheme is set up, the core strategy can be used to complete the proofs of the different cases.

6.6 Verification of Interrupt Handling

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. In the following, we sketch a possible approach for extending this work to include the interrupt behavior of the AAMP5.

An external interrupt can arrive and be acted upon by the processor not just at macroinstruction boundaries, but at any point within a macroinstruction execution cycle. For this reason, a complete specification of all aspects of the behavior of the AAMP5 in response to an external interrupt is possible only in a model, such as the micromachine, where time is represented at the clock cycle level. As a specific example, we consider the response to a reset (RST) interrupt.

An RST interrupt forces the processor to go through a predefined procedure has the following requirements:

1. The interrupt is initiated by setting the reset input false. This is recognized by the AAMP5 in the same clock cycle when the reset input becomes false.
2. The machine then "idles" until the reset input is released, i.e., set true.
3. Once the reset is released, all other pending interrupts (except bus transfer error) are ignored.
4. The processor then executes a special microcode routine that initializes the processor and puts it in a state in which it is ready to start executing macroinstructions.

Each of the requirements in the above list can only be specified as a constraint on the behavior of the micromachine. The only aspect of the requirements that can be specified at the macro level is the effect that a reset must have on the macrostate. Hence, a specification of the effect of an interrupt is specified in two parts. The first part specifies the temporal aspects of the interrupt that includes details such as when an interrupt is recognized and masked. The second part specifies the intended cumulative effect of the external interrupt on the macrostate by providing a next-state-function similar to the next-state-function defined for normal instruction execution. A specification of the requirements on the reset interrupt is given below.

```
reset_initiate(t): bool
 = IF t = 0 THEN true ELSE (reset(t-1) & NOT(reset(t))) ENDIF

reset_release(t): bool
  = NOT(reset(t)) & reset(t+1)

idle_upon_reset:
     THEOREM (FORALL t1, t2: reset_initiate(t1) & stays_low(reset)(t1, t2)
                IMPLIES idles(t1, t2) )

reset_complete:
   THEOREM (FORALL t1, (t2 | t2 > t1), (t3 | t3 > t2):
       reset_initiate(t1) & stays_low(reset)(t1, t2) & reset_release(t3)
         IMPLIES (EXISTS (t4 | t4 > t3):
            (visible_state(t4) & ABS(t4) = reset_next_state(ABS(t3)) ) ) )
```

`Reset_initiate` and `reset_release` define the conditions for initiating and releasing the reset input. `Idle_upon_reset` specifies the requirement that the machine must idle from the time reset becomes false and as long as reset stays false. `Reset_complete` specifies the intended behavior of the machine after a reset is released. `ABS` is the abstraction function that extracts the macrostate, i.e., externally visible part, of the micromachine state at a given point in time. `Visible_state` characterizes the condition when the micromachine pipeline is executing the macroinstructions normally. `ABS` and `Visible_state` are defined in Section 6. These properties are similar in form to the verification conditions, shown in later Section 6, that are proved to establish the correctness of normal instruction execution. Although the interrupt properties have not been mechanically verified, the proof techniques describe in Section 6 should be directly applicable.

This illustrates how verification of the reset interrupt could be handled. A complete discussion of interrupt handling would specify the behavior of the other three interrupts (XER, NMI, IRQ) and how they interact. In particular, the order in which interrupts are processed and how they affect each other's behvaior must be dealt with. However, as illustrated for the reset interrupt, it should be possible to treat this as a separate case from the specification and verification of the AAMP5's normal behavior.

6.7 Scope of the Verification Performed

The goal of this project was multifold. Besides demonstrating the verification of a portion of AAMP5 microcode, we were interested in investigating how formal verification technology can be packaged so that practicing engineers could use it productively, with some expert help, for verifying a complex microprocessor design. Hence the contributions of this project extend beyond what was mechanically verified. In the following, we discuss the scope and some of the limitations of our verification effort.

1. We developed a methodology for incremental verification of a complex microprogrammed and pipelined microprocessor design. The methodology includes the following three components:

    - A style of specification that strikes a good compromise between the competing demands of efficient verifiability versus readability by practicing engineers.

- An approach to systematically decomposing the problem of verifying the general correctness criterion for a processor specified at two levels to a set of smaller more automatically provable, verification conditions.
- A proof strategy that can be used to verify the verification conditions completely automatically or that can be used within a higher-level proof technique, such as induction, to construct a proof of the verification conditions.

2. We formalized the verification conditions of a total of eleven instructions and verified them completely using the core strategy. Seven of these verified instructions are from the core set and are representative of several of the main classes—literal data, assign data, reference data, and branch—of AAMP5 instructions. The rest of the verified instructions are from a supplementary set that has instructions drawn from the same class as the core set. One of the instructions from the supplementary set (AND) was partly verified by one of the Rockwell staff members (Al Mass) working on the project.
3. We have developed the basic formal machinery required to specify the entire macroinstruction set architecture of AAMP5 and used it to specify over 100 of its instructions.
4. We have derived a set of macro lemmas from the macro specification that specify the instructions in the core set in a declarative style.

Of the accomplishments listed above, the one whose impact is, perhaps, the most enduring is the development of the methodology. Not only can a significant part of the methodology be reused in the verification of another member of the AAMP5 family of processors, but many of the general ideas are useful in verifying any complex microprocessor design.

Some of the limitations of the project are the following:

1. The verification conditions that were mechanically verified prove the correctness of instructions only in the absence of any external interrupts. The framework and the methodology developed here are capable of formalizing and verifying correctness of instructions in the presence of interrupts. We have formalized the correctness of the `reset` interrupt although it was not included in the report.

2. Not every AAMP5 instruction class had a representative among the instructions that were verified. For example, instructions, such as CALL, RETURN, belonging to procedure call class were not verified. One of the reasons the verification did not address stack overflow was because a stack overflow condition, in general, involves procedure calling. Arithmetic instructions implementing floating point calculations were not addressed by the verification.

3. The correspondence between the verification conditions and the macro lemmas was verified only for the ADD instruction. The strategies used to perform this proof task for ADD should be applicable for other instructions.

4. The proof of correctness is only valid for programs that are not self-modifying.

5. The correctness of our verification is predicated on the correctness of the interface specification, which has been inspected by Collins but has not been verified with respect to the LFU and BIU. Verification of the interface specification, which is expressed as properties of the protocol used by the DPU to interact with its environment, is best done using state enumeration verification tools, such as CTL model-checkers [Gup92] on an abstract global state machine of the DPU-BIU-LFU system. We have verified some the properties in the interface specification for a simplified model of the DPU-BIU-LFU interface using Murphi [MD93] and the recently implemented connection between PVS and a model-checker.

6.8 Errors Discovered by the Verification Effort

Analysis of a system by means of formal methods can reveal errors in two ways. First, the act of formalizing the system specification by itself forces a closer scrutiny of the design than would be performed using traditional methods. Second, mechanizing the proof of correctness has the effect of "testing" the design for all possible inputs. In the AAMP5 verification exercise, both the specification and the proof phases revealed errors in the microcode.

Two errors in the microcode 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. The first was a logic error that allowed the top of stack register (TOS) to wrap around a data environment instead of raising a stack overflow. To result in a failure, this error required the very unlikely combination of an unusual system configuration, an improperly sized stack, and a specific sequence of instructions. The second error was made precisely because the reference manual was unclear on how the AAMP should update the local environment register (LENV) when a procedure call caused a stack overflow. This was implemented by setting the LENV to its "overflow" value, while the correct behavior was to leave the LENV unchanged. While this error would have been discovered during Ada validation testing, the validation suite would not have been executed until after the first AAMP5 chips were in hand. Both errors were unique to the AAMP5 and corrected before first fabrication.

Errors were also found in the formal specifications through inspections as discussed in Sections 4.4.3 and 5.3.3. These revealed 28 correctness errors in the macro specification and 19 correctness errors in the micro specification. Even so, constructing proofs of correctness found several additional errors in our specifications. In the initial stages, most of the errors found were mistakes in our micromachine specification. Some of these were due to ambiguities or mistakes in the microarchitecture document and the rest were errors in our transcription of the design document into a formal specification.

One technique used to find errors in our specifications was to assign independent teams to different portions of the project. For example, different individuals at Collins were assigned to review and revise the macro and micro specifications. While this independence undoubtedly introduced some errors of communication, it also reduced the likelihood of coincident errors in the macro and micro specifications, greatly increasing our confidence in the specifications once the proof was actually completed.

More significant were errors discovered in the microcode itself by the proof process. Since only a small set of instructions were to be formally verified and because these instructions had already been verified by traditional methods, it was unlikely that any errors would be found through the correctness proofs. To address this, two memory calculation errors were deliberately inserted in the microcode without the knowledge of SRI. The motivation behind this was not only to check whether the proof process would reveal the errors but also to see whether it would provide enough information to show how the errors could be corrected.

One of the microcode errors was in the ASNDXI instruction, which moves data in the process stack to a location in memory. The other was in the REFBXU instruction, which loads data from memory to the top of process stack. The ASNDXI error was deliberately planted by Collins. The REFBXU error was an actual error that had not been detected by traditional verification methods such as walkthroughs and testing, but was found by Collins while running samples of application code on early prototypes of the AAMP5 chip. This error was left in the microcode delivered to SRI to determine if the proof process would find an error that was discovered late in the traditional verification process.

The two errors were similar in that they occurred in the calculation of the memory address and were hard to spot during microcode walkthroughs due to the pipelining of microinstructions. The address of an AAMP5 memory location is constructed as a concatenation of a pointer to a data environment and an offset into the data environment. The REFBXU instruction, for example, uses the top three elements—A, B, and I—of the process stack to determine the source data address. The least significant byte of B is used as the data environment pointer part of the address while (A+I/2) forms the offset. The error in the microcode had the effect of I being used for the data environment instead of B.

This error was not detected during simulation due to a limitation in the simulator's memory model. The memory model has since been enhanced, but even with this change, errors such as these could still be missed since a simulation run can only exercise a small fraction of the input space. One of the advantages of formal verification is that it has the effect of exhaustive testing since the proof performs symbolic reasoning for all possible inputs.

An error in the microcode manifests itself during a proof in the form of a verification condition reducing to a goal that is unprovable. The error in the REFBXU instruction, for instance, was detected during the proof of the verification condition that characterized the correctness of the T0 register. The unprovable proof goal that was produced at the end of the application of the proof strategy described in section 6.5 is shown below.

```
[-1]   nmod(bv2nat[3](SV(t0)) + bv2nat(TV(t0)) + 5, 6) >= 0
[-2]   nmod(bv2nat[3](SV(t0)) + bv2nat(TV(t0)) + 5, 6) < 6
[-3]   bv2nat[3](SV(t0)) <= 6
[-4]   bv2nat[3](TV(t0)) <= 3
[-5]   (bv2nat(SV(t0)) + bv2nat(TV(t0)) >= 3)
[-6]   (bv2nat(SV(t0)) + bv2nat(TV(t0)) <= 6)
[-7]   (bv2nat(TV(t0)) = 2)
[-8]   islow(REG(t0 + 1)(bv2nat(SV(t0)) + bv2nat(TV(t0)) - 3) ^ 0)
  |-------
[1]   nextDJMP(t0)
[2]   SADJ(t0)
[3]   DJMP(t0)
{4}   fill[8](0)
       ((DATA_MEMORY(t0 + 1)(REG(t0 + 1)(bv2nat[3](SV(t0)) - 1))^ (7, 0)
                     o
                     (T0(t0 + 1) ^ (15, 0)
                      +
                      (fill[1](0)
                       o REG(t0 + 1)(bv2nat[3](SV(t0)) - 1)
                            ^ (15, 1))))^(7,0))
      = fill[8](0) o
        ((DATA_MEMORY(t0 + 1)(T1(t0 + 1) ^ (7, 0)
                              o
                              (T0(t0 + 1) ^ (15, 0)
                               +
                               (fill[1](0)
                                o
                                REG(t0 + 1)(bv2nat(SV(t0)) - 1)
                                ^ (15, 1)))))^ (7, 0))
Rule?
```

A careful reading of the succedent labeled 4, which is the reduced form of the equation that compares the actual (on the left hand side) and the expected values (on the right hand side) for the T0 register, in the sequent shows that the equation is false. The two sides are identical except for the term appearing in the data environment part of the memory address. While the expected result is supposed to get this from the T1 register (`T1(t0 + 1)^(7, 0)`), the microcode actually uses the REG register instead, which contains the third element from the top-of-stack.

Chapter 7

Conclusions and Lessons Learned

This section discusses the lessons learned on this project and their implications for the industrial use of formal methods.

7.1 Feasibility of Formal Verification

The central result of this project was to demonstrate the technical feasibility of formally specifying the AAMP5 and the use of mechanical proofs of correctness to verify its microcode and micro-architecture. A much larger fraction of the AAMP instruction set was specified than originally planned, with 108 of the AAMP's 209 instructions completed. The portion completed is actually greater than this, since many of the instructions specified are representative of an entire family of instructions. This is notable since the AAMP has a large and complex instruction set, providing in hardware many of the features normally provided by the compiler's run-time environment and the real-time executive.

All of the micro-architecture needed for formal verification of the microcode was formally specified. Due to the style of specification chosen, translation of the microcode into PVS was a simple exercise that should be easy to automate.

At this time, eleven instructions have been proven correct in the absence of interrupts. Since these are representative of several major instruction classes, most of the low level proof strategies could be reused in verification of the remaining instructions. The existence of these strategies will also make it simpler to transfer this technology to Collins. We do not see any technical obstacles to extending either the specification of the macro-architecture or the correctness proofs.

7.2 Benefits of Formal Verification

Many benefits were obtained on this project through the use of formal specifications alone. Our experiences suggest that one of the most important benefits of formal specification is to precisely define the interface between users and developers, encouraging the development of a clean interface. For example, the difficulty of formally specifying when stack overflow is detected pointed out the need to better hide the stack cache from an application programmer. The process of completing a formal specification encourages the specifier to "look in the corners" and consider unusual cases and boundary conditions that are often sources of errors. To our surprise, this process alone uncovered two errors in the microcode that had not yet been discovered by traditional methods. Formal specification also pointed out several situations that the AAMP2 Reference Manual [Roc90] and the AAMP5 design documents left unspecified or stated unclearly. This seems to be a general deficiency of any English specification and not of the AAMP documentation.

The process of performing mechanical proofs has detected several errors in the formal macro and micro-specifications. More importantly, the correctness proofs systematically found two errors seeded in the microcode. Our belief is that construction of a proof forces a much more detailed review of the microcode than is achieved through traditional methods such as walkthroughs.

7.3 Cost of Formal Verification

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, proof strategies, and expertise should greatly reduce these costs in the future. Some portions of the specification, such as the bit vector theories, can be reused across a wide range of hardware applications. For microprocessors in the AAMP family, virtually the entire macro-architecture specification can be reused. Even for new development, the examples created in this project are rich enough to allow the designers to write similar specifications, eliminating the time spent by SRI in studying the AAMP5 and by Collins in reviewing and revising the formal specifications.

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. Also, much of the overhead of completing the proofs was due to inefficiencies in the implementation of PVS then available. Improvements to PVS incorporated as a direct result of this project rectify most of these problems. Even so, formal verification is likely to remain more expensive than traditional methods. This should not be surprising. Traditional methods rely on reviews, partial analyses, and testing a portion of the input space. Proofs of correctness are a rigorous form of analysis that verifies the design for all possible inputs. To provide the same level of assurance, traditional methods would be just as, if not more, expensive than formal methods.

7.4 Transferring Formal Methods to Industry

It is very difficult to inject new methodologies into an industrial setting since one of the ways industry remains competitive is to use tried and tested approaches within a well understood problem domain. Despite their name, formal methods provide remarkably little methodology to guide their use in a new setting. The difficulties in specifying and verifying a real-time executive are likely to be very different from those of verifying microcode. 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, a hypothesis we plan to demonstrate in the upcoming year.

We did not feel that it was particularly difficult for the engineers at Collins to learn to use either the PVS language or the theorem prover. In fact, it was much easier for them to apply formal methods than it was for the formal methods experts to become knowledgeable about the AAMP5.

The real problem was not how to use PVS, but how to build a precise mathematical model of our own microprocessor. Even so, widespread acceptance of a general purpose specification language such as PVS or Z [BMD92] by practicing engineers is likely to be an uphill battle. A more productive approach may be to develop specialized notations or models that fit a specific problem domain and that can automatically be translated into an underlying formalism such as PVS. This would allow the domain experts to work in a familiar and natural notation while a small group of formal methods experts (and tools) check their work for consistency and completeness.

In the near term, an important goal on future projects will be to get formal specification integrated into the early design effort. This will eliminate many of the costs of developing and validating the specifications, particularly if they can be used as the primary specification, not just as an add-on. Enhancements to PVS could facilitate this. In particular, integrating PVS with the standard document preparation system used at Collins would allow us to intersperse the formal specification with the text and diagram style used currently, i.e., the "specification as a document" concept promoted in Z and CaDiZ [BMD92].

Validation of formal specifications is essential to have confidence in the correctness proofs. We found inspections worked well with formal specifications, were quite inexpensive, and provided a natural vehicle for training. Maximizing the independence of the teams producing the specifications greatly increased our confidence in both the proofs and the specifications. When combined with proofs of correctness, this is a very powerful validation technique that should not be overlooked. Other forms of validation that could have been used more extensively in this project include early proof of the type correctness conditions generated by PVS and proving expected properties, or putative theorems, of the specification. Even our limited experience with proving putative theorems suggests that this is a useful validation technique.

Full proofs of microcode correctness are a very rigorous form of analysis, enabling one person to achieve a much higher level of confidence than can now be achieved by a team. Even so, there is very little in existing verification practices that would be eliminated by formal proofs. It may be possible to decrease the time spent on inspections, but some level of peer review will still be necessary to ensure good style, maintainability, and to check for issues not modeled in the specifications. It may be possible to replace some testing with proofs, but testing would not be eliminated since it provides an important check on the fidelity of the specifications and low level properties not modeled in the specification. In the specific case of the AAMP family, large libraries of simulations and diagnostics have been built up over the years. These cost very little to modify and execute, so it is unlikely that any testing would be eliminated on future AAMP projects.

Formal methods provide the means to improve, not replace, existing practices. Formal specification can play an important role by improving the precision and clarity of communication, particularly when the specification language closely matches the problem domain. Formal verification of selected properties can provide validation of the specifications that would be particularly valuable during early life-cycle activities such as requirements capture. Finally, formal proofs of correctness provide a rigorous analysis of the consistency between a specification and its design appropriate when extremely high levels of assurance are essential or when the complexity of interacting components is so great that analysis is the only adequate means of verification.

Acknowledgements

The authors thank Rick Butler of NASA Langley for his support and refinement of the bit vectors library, Sam Owre and Natarajan Shankar of SRI International for the development and maintenance of PVS, and Al Mass and Dave Greve of Rockwell International for their many hours on this project. We also thank John Rushby of SRI and John Gee, Dave Hardin, Doug Hiratzka, Ray Kamin, Charlie Kress, Norb Hemesath, Steve Maher, Jeff Russell, and Roger Shultz of Rockwell for their support and assistance.

References

[BB94] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st Design Automation Conference*, pages 596–602. Association for Computing Machinery, June 1994.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, pages 68–80, Stanford, CA, June 1994. Volume 818 of *Lecture Notes in Computer Science*, Springer-Verlag.

[BF93] R. Butler and G. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 16(5):66–76, January 1993.

[BG90] S. Brock and C. George. *The RAISE Method Manual.* Computer Resources International A/S, 1990.

[BKM+82] D. Best, C. Kress, N. Mykris, J. Russel, and W. Smith. An advanced-architecture cmos/sos microprocessor. *IEEE Micro*, pages 11—26, August 1982.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[BMD92] A. Burns, J. McDermid, and J. Dobson. On the meaning of safety and security. *Computer Journal*, 35(1):3–15, February 1992.

[But91] Ricky W. Butler. NASA Langley's research program in formal methods. In *COMPASS '91 (Proceedings of the Sixth Annual*

Conference on Computer Assurance), pages 157–162, Gaithersburg, MD, June 1991. IEEE Washington Section.

[CJB78] W. C. Carter, W. H. Joyner, Jr., and D. Brand. Microprogram verification considered necessary. In *National Computer Conference*, pages 657–664. Volume 48, AFIPS Conference Proceedings, 1978.

[Coo86] J. V. Cook. Final report for the C/30 microcode verification project. Technical Report ATR-86(6771)-3, Computer Science Laboratory, The Aerospace Corporation, El Segundo, CA, September 1986. Export-controlled.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Kumar and Kropf [KK94], pages 203–222.

[Cyr93] 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.

[DBC90] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. Formal design and verification of a reliable computing platform for real-time control. NASA Technical Memorandum 102716, NASA Langley Research Center, Hampton, VA, October 1990.

[Fag86] M. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744—751, July 1986.

[GBG+91] S. Gerhart, M. Bouler, K. Greene, D. Jamsek, T. Ralston, and D. Russinoff. Formal methods transition study final report. Technical Report STP-FT-322-91, Microelectronics and Computer Technology Corporation, Austin, Texas, August 1991.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[Gor85] 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.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in Systems Design*, 1(2/3):151–238, October 1992.

[HB92] Warren A. Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–47, Hemel Hempstead, UK, 1992. Prentice Hall International Series in Computer Science.

[Hun94] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1994.

[Int93] *Pentium Processor User's Manual, Volume I: Pentium Processor Data Book*. Intel Corporation, order number 241428 edition, 1993.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[Joy88] Jeffrey Joyce. Verification and implementation of a microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, Boston, MA, 1988.

[KK94] Ramayya Kumar and Thomas Kropf, editors. *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[LCB74] George B. Leeman, William C. Carter, and Alexander Birman. Some techniques for microprogram validation. In *Information Processing 74 (Proc. IFIP Congress 1974)*, pages 76–80. North-Holland Publishing Co, 1974.

[LS93] B. Littlewood and L. Strigini. Validation of ultra-high dependability of software-based systems. *CACM*, November 1993.

[MD93] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, June 1992. Volume 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in early 1995.

[Roc90] *AAMP2 Advanced Architecture Microprocessor II Reference Manual*. Rockwell International, Collins Commercial Avionics, Rockwell International Corporation, Cedar Rapids, Iowa 52498, February 1990.

[Roc93] *AAMP5 Microarchitecture (Unreleased Document)*. Rockwell International, Processor and Software Technology Department, Advanced Technology and Engineering, Collins Commercial Avionics, Rockwell International Corporation, Cedar Rapids, Iowa 52498, February 1993.

[SB90] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[SGGH94] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. *Formal Methods in System Design*, 4(1):181–210, 1994.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in early 1995.

[SSR95] 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.

[TK93] 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)*, pages 281–294, Vancouver, Canada, August 1993. Number 780 in Lecture Notes in Computer Science, Springer-Verlag.

[WC94] Phillip J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Kumar and Kropf [KK94], pages 33–51.

[Win90] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, CA, June 1990.