To appear in the Proceedings of the Workshop on Industrial–Strength Formal Specification Techniques (WIFT'95), Boca Raton, Florida, April 5–8, 1995.

Formal Verification of the AAMP5 Microprocessor [1]

A Case Study in the Industrial Use of Formal Methods

Steven P. Miller
Collins Commercial Avionics
Rockwell International
Cedar Rapids, IA 52498 USA
spmiller@pobox.cca.rockwell.com

Mandayam Srivas
Computer Science Laboratory
SRI International
Menlo Park, CA 94025 USA
srivas@csl.sri.com

Abstract

This paper describes the experiences of Collins Commercial Avionics and SRI International in formally specifying and verifying the microcode for the AAMP5 microprocessor with the PVS verification system. This project was conducted to determine if an industrial microprocessor designed for use in real–time embedded systems could be formally specified at the instruction set and register transfer levels and if formal proofs could be used to prove the microcode correct. The paper provides a brief technical overview, but its emphasis is on the lessons learned in using PVS for an example of this size and the implications for using formal methods in an industrial setting.

Keywords: Formal Methods, Formal Specification, Formal Verification, Microprocessor Verification, Microcode Verification, Hardware Verification, High Integrity Systems, Safety Critical Systems, PVS

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 [9,19]. Formal methods are a promising approach for increasing our confidence in digital systems, but many questions remain on how it can be used effectively in an industrial setting.

This paper 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 by Collins Commercial Avionics, a division of Rockwell International, it was conducted by Collins and the SRI International Computer Science Laboratory. The project consisted of specifying in the PVS language developed by SRI [22] a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register–transfer levels and using the PVS theorem prover to show the microcode correctly implemented the specified behavior for a representative subset of instructions.

While this paper includes a brief technical overview (see [28,29] for a detailed technical discussion), its emphasis is on the lessons learned in using PVS for an example of this size and the implications for using formal methods in an industrial setting. 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 is particularly significant since the AAMP5 was not designed for formal verification, but to provide a more than three fold performance improvement, by pipelining instruction execution, 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.

Another key result was the discovery of both actual and seeded errors. Two actual microcode errors were discovered and corrected during development of the formal specification, illustrating the value of simply creating a precise specification. Two seeded errors were systematically uncovered while doing correctness proofs. One of these was an actual error that had been discovered after first fabrication but left in the microcode provided to SRI. The other error was designed to be unlikely to be detected by walkthroughs, testing, or simulation.

Several other results emerged during the project, including the ease with which practicing engineers became

[1] This work was supported by the National Aeronautics and Space Administration, Langley Research Center, under contracts NAS1–18969 and NAS1–19704.

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.

2 Background

NASA Langley's research program in formal methods [8] 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) [13], it 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 [23] and hardware designs [12]. 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 [17]. 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.

2.1 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 [1,4]. The AAMP architecture is specifically designed for use with block–structured, high level languages like 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. Use of an internal stack cache 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 processors. To support real–time embedded systems, the AAMP architecture also provides many functions normally provided by the real–time executive, such as interrupt handling, task state saving, and context switching.

The AAMP instruction set is large (208 instructions), CISC–like, and closely resembles the intermediate output of many compilers. Instructions are of variable length, although most are only one byte long, resulting in improved throughput and code density. The instruction set supports arithmetic operations on 16–bit and 32–bit integer and fractional data types, as well as 32–bit and 48–bit floating point data. The AAMP family also provides built–in error checking. Computational exceptions, such as arithmetic overflow, divide–by–zero, and stack overflow are automatically detected and handled.

The original CAPS architecture, a multi–board 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. 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. Members of the CAPS/AAMP family have been used in an impressive variety of products including the Lockheed L–1011 Digital Flight Control System (DFCS) and Active Control System (ACS), the Boeing 757 and 767 Autopilot Flight Director System (AFDS), the Boeing 747–400 Integrated Display System (IDS) and Central Maintenance Computer (CMC), the Boeing 757, 767, and 737–300 Electronic Flight Instrumentation System (EFIS) and Engine Instrumentation/Crew Alerting System (EICAS), and Navstar Global Positioning System (GPS) receivers.

2.2 PVS

PVS (Prototype Verification System) [22] is an environment for specification and verification that has been developed at the SRI International Computer Science Laboratory. In comparison to other widely used verification systems, such as HOL [14] and the Boyer–Moore prover [5], the distinguishing characteristic of PVS is that it supports a highly expressive specification language with a very effective interactive theorem prover in which most of the low level proof steps are automated. 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 include, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD–based Boolean simplification.

2.3 Historical Perspective/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 [2,3,10,11,15,16,18,25,26,27,30,31,33]. 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 [24].

Microcode verification is not new: it was pioneered by Bill Carter at IBM in the 1970's and applied to elements of NASA's Standard Spaceborne Computer [10]; in the 1980's a group at the Aerospace Corporation verified the microcode for an implementation of the C/30 switching computer using a verification system called SDVS [11]; 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 [2]. Similarly, several groups have performed automated verification of (non–microcoded) processors, of which Warren Hunt's FM8501 [15] (and subsequent FM9000 [16]) are among the most substantial. The problems of pipeline correctness have also been studied previously by Srivas and Bickford [26,27], Saxe and Garland [25], Burch and Dill [7], and Windley and Coe [33]. 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 [30]. PVS is able to verify the microcode of Tamarack completely automatically in about five minutes [12].

2.4 Overview of the Technical Approach

The verification of a microprocessor normally involves specifying the processor as a machine that executes instructions at two levels – the macro level and the micro level – and then proving a desired correctness condition that relates the behavior of the processor at these two levels. The macro level specification of AAMP5 describes the externally observable effect of executing an instruction on the state visible to an assembly language programmer. The micro level specification describes the AAMP5 at the register–transfer level, defining the effect of executing an arbitrary micro–instruction on the movement of data between the registers and other components in the AAMP5 design. Verifying the correctness of an instruction consists of defining an appropriate *Abstraction* function between these levels (Figure 1) and showing that the sequence of micro–instructions f_1, f_2, ..., f_n making up each machine instruction F causes a corresponding change in the micro–state s_1 as F does to the macro–state S_1, i.e., that $F(Abstraction(s_1)) = Abstraction(f_n(...(f_2(f_1(s_1)))...))$. This basic notion of correctness must be supplemented with a few additional assurances, such as a demonstration that the machine reaches a valid initial state after power–up and that each instruction eventually terminates. Further refinements are also necessary to deal with the internal pipelining of the AAMP5. These details are discussed more fully in Sections 4, 5, and 6.

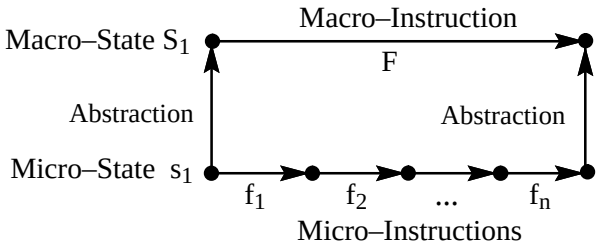

Figure 1 – Overview of the Technical Approach

It is important to note that the microcode verification performed here, unlike other commercial microcode verification efforts [2,32], does not rely on an abstract semantics for the interpretation of the micro–instructions. We show that the actual register–transfer level hardware interpretation of the microcode correctly implements the macro–instructions. The verification [2] of the microcode of the T800 floating point unit showed that the microcode satisfied its specification using a high level operational model for the micro–instructions. Yuan Yu's work [32] mechanically checked the correctness of a set of MC68020 object–code programs using a formal model of the MC68020 instruction set.

3 Project Organization and History

This project was selected by Collins and SRI 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 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 had shown that this is one of the most difficult parts of the microprocessor to get right. Success with formal verification in other projects suggested that this technology might be ready for application to an industrial microprocessor.

Due to the importance of the AAMP5 to Collins, the formal specification and verification of the AAMP5 was performed as a shadow project and did not replace any of the normal design and verification activities performed on a new microprocessor. This also allowed us to relax some of the steps that would be required on a production project and focus 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, were identified to be specified and verified by SRI. An additional set of 11 instructions were 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 verify the entire instruction set since the core set contained at least one member from each instruction class.

A summary of the level of effort is presented in Table I. As shown there, 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 Collins' Cedar Rapids facility for the five engineers that would be involved with the project. These consisted of half–day lectures with related lab exercises in the afternoon. No additional formal training was felt necessary. When new

Table I – 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 Macro–Architecture (2,550 Lines of PVS in 48 Theories)					
	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 Micro–Architecture (2,679 Lines of PVS in 20 Theories)					
	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

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 macro–architecture, specification of the micro–architecture, and proofs of correctness of the microcode. Each of these phases are discussed in detail in Sections 4, 5, and 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 inspections, 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 macro–architecture and micro–architecture specifications. Different teams were also used to inspect the macro–architecture and the micro–architecture. The microcode itself was produced by a member of the original AAMP5 team without any knowledge of the formal specifications and translated into the PVS specification language 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.

4 Specification of the Macro−Architecture

The macro–architecture specification of the AAMP family defines each instruction as a state transition function over

```
% The macro state theory defines the state of internal registers in the AAMP, code memory,
% and data memory. Basic operations based on the combined state of these registers
% and memory are also defined here.
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 flag, the interrupt enabled flag, 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 #]

    % Returns the current code environment.
    current_code_env(st: macro_state) : code_env = cmem(st)(word2cenv(cenv(st)))

    % Returns the current instruction byte + n.
    nth_instr_byte (st: macro_state, n: nat) : byte = current_code_env(st)(pc(st) + n)
                                    • • •
END macro_state
```

Figure 2 – PVS Specification of the AAMP Macro–Architecture State

the state of the microprocessor and external memory. Figure 2 shows a portion of the PVS specification of the *macro state*, consisting of code memory, data memory, and several internal flags and registers. This is precisely the view of the AAMP state that an application programmer must understand to write assembly code.

Figure 3 shows a portion of the *ref update* theory that defined how the AAMP REF (reference) instructions update the *macro state*. REF instructions copy words from data memory to the top of the accumulator stack. The current macro state is provided as a parameter to the *next macro state* function. The base and offset of the source data is provided by the *data address base* 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 macro state.

4.1 Initial Development

The macro–architecture specification of the AAMP was developed through gradual refinement by SRI and Collins and progressed through 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 [1]. 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. 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).

There was one design choice made in implementing the AAMP5 where completion of the formal specification beforehand would have been beneficial. The AAMP5 does not signal a stack overflow until the top–of–stack exceeds

```
% 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 stack, and the program counter incremented to point to
% the next instruction.
ref_update : THEORY

BEGIN

    IMPORTING addressing, executive_service_routines

    % Address base and offset of data source.
    base  (st:macro_state)     : data_env_ptr     =  data_address_base(st)
    offset (st: macro_state)   : data_env_addr    = data_address_offset(st)

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

      CASES memory_data_type_of(current_opcode(st)) OF
                        • • •

        % 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, number_of_addr_words(current_opcode(st))))),
                        • • •
      ENDCASES WITH [(pc) := next_pc(st)]

END ref_update
```

Figure 3 – Specification of the REF Instructions

both the stack limit and the additional capacity of the stack cache, providing the application program a few more words of space beyond the stack limit. Since the exact number of extra words varies depending on the instruction and the state of the stack cache, this was impossible to model without bringing all the details of the stack cache into the macro–architecture specification. Ultimately, the formal specification was written as though stack overflow was signaled when the stack limit was exceeded and adjustments were made to the correctness conditions to complete the proofs. If the formal specification had been developed before the detailed design, stack overflow would probably have been signaled when the stack limit was exceeded, better hiding the stack cache from the application programmer.

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 a 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 that the Collins team was best situated to facilitate this.

Over the next five months, the roles of SRI and Collins on the macro–architecture 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 macro–architecture specification consisted of 2,550 lines of PVS organized into 48 theories, not including the bit vectors library mentioned earlier.

Most surprising during this phase was the discovery of two errors in the AAMP5 microcode even though it had already been reviewed and partially tested. 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 would have been discovered during Ada validation testing, the validation suite probably 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.

4.3 Inspection of the Macro Specification

Formal inspection of the macro–architecture was felt to be essential, both to validate the correctness of the specification and to familiarize more engineers at Collins with PVS. Checklists were drawn up for use in the inspections based on earlier checklists used in inspecting VDM specifications, the RAISE Method Manual [6], 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. Major defects were defined as those that affected correctness. As a rule of thumb, a defect was classified as minor if two reasonable people could disagree on whether it was a defect. 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. Total time spent in preparation by all participants, time spent in inspection, and number of defects found are shown in Table II.

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 rate increased during inspections nine through eleven since these were of simple tables).

Table II – Macro–Architecture 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**

This rate is probably 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 for inspection. As shown in Table I, 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 clarity could dominate an inspection if a theory was not well organized. On the few occasions that unprepared theories were submitted for 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, 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 II and III by the lower number of defects recorded in the later inspections.

4.4 Revisions to Support Proofs

As work began on proving the microcode correct, it became necessary to make several changes to the macro–architecture specification to support the proofs. Some of these were due to limitations in the version of PVS then available. The combined macro and micro–architecture specifications were the largest PVS specification completed to date and optimizations were needed in how PVS handled certain constructs. Since these improvements would take time, the short–term solution was to revise the specifications to use other PVS constructs.

Other changes were more substantive. The original style of specification shown in Figure 3 was *constructive* in that it defined a specific function for the next macro state function. It also used a number of auxiliary functions that made the specifications compact and easy to read, but required the user to look up each function to truly understand the resulting change in state. This style also made it difficult to specify a single instruction since most of the structure for an entire class of instructions was required in order to complete the first instruction in that class. In doing the proofs, it became clear that a more *descriptive* style of specification, in which the change in state was more directly defined, would be needed as an intermediate step. Fortunately, these could be stated as lemmas that could be proven from the original specification, preserving our investment in inspections. An example is presented in Figure 4 for the REFDL instruction. As these lemmas were created, it became evident that in many ways they were 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. Specifying each instruction as one or more axioms in this style would have made specifying the core set of 13 instructions much simpler. However, this would also have made it easier to introduce inconsistencies in the specification, e.g., specifying two different next states for the same current

```
% 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 pointer and F is the least
%          significant four bits of the current instruction byte.
REFDL_lemma: LEMMA
    LET    F      =  current_code_env(st)(pc(st)^(3,0)),
           XLS    = current_data_env(st)(lenv(st) + (bv2nat(F) )),
           XMS    = current_data_env(st)(lenv(st) + (bv2nat(F) + 1))
    IN current_opcode(st) = REFDL & tos(st) > sklm(st) + 1 IMPLIES
        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]
```

Figure 4 – Descriptive Style of Specification

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 the constructive specification satisfied each lemma. One of the lessons learned during this project was to more carefully consider the trade–offs between these two styles of specification.

5 Specification of the Micro–Architecture

The micro–level specification describes the AAMP5 design at the register–transfer level, i.e., it specifies the effect of executing an arbitrary micro–instruction on the movement of data between the registers and other components in the AAMP5 design. As shown in Figure 5, the AAMP5 design consists of four semi–autonomous functional units. The DPU uses the LFU/ICU pair to fetch instructions from memory and the BIU to transfer data to and from memory. The microcode for an instruction resides inside a ROM in the DPU. In addition to controlling the movement of data within the DPU, the microcode also generates DPU requests for the other units. Since our goal was to verify the microcode as interpreted by the DPU, we modeled the DPU design (shown in greater detail in Figure 6) at the register–transfer level, but we abstracted the behavior of the LFU and BIU by specifying their responses to the various requests generated by the DPU rather than their detailed design.

The DPU is specified as a finite state machine. The state of the machine is modeled as a set of signals, i.e., values on wires that vary over time. A signal may denote an external input to the machine, e.g., an interrupt, or an

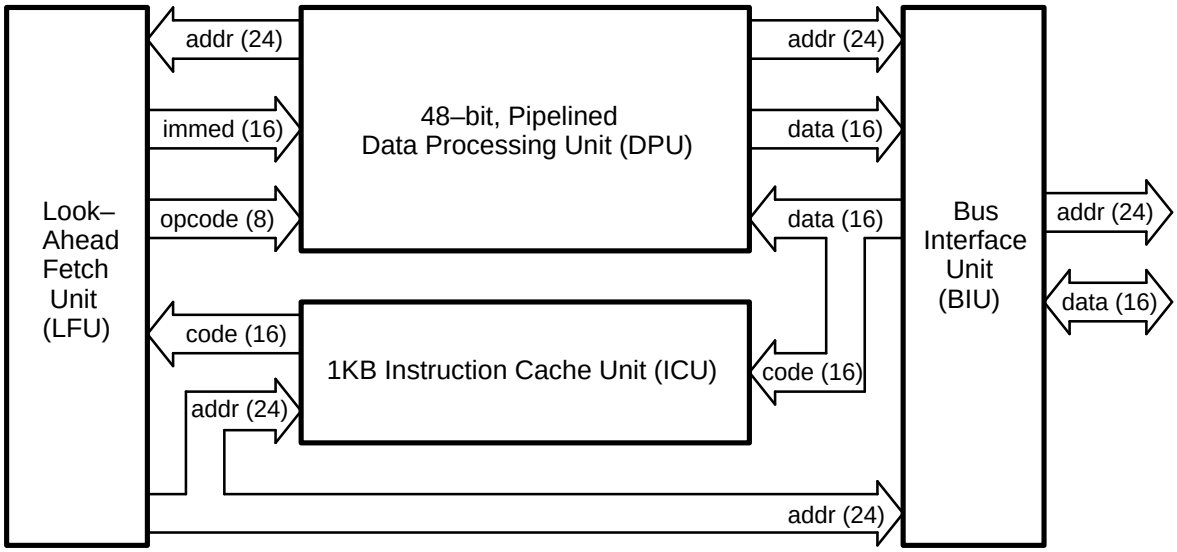

Figure 5 – Block Diagram of the AAMP5 Microarchitecture

output of a state–sensitive component of the design, e.g., a register, memory, register file, etc. Time is a discrete quantity measured in cycles of the master clock. The specification of the machine defines the value at time t+1 of every signal that denotes an output of a component in terms of the values of the inputs to the component at time t.

5.1 Initial Development

Development of the micro–architecture specification mirrored that of the macro–architecture. As indicated in Table I, initial development of the micro–architecture specification by SRI took approximately 657 man hours over 10 months. The specification closely followed the block structure of the micro–architecture, usually with one theory per component. Surprisingly, the specification of the micro–architecture without the PVS version of the microcode was only slightly larger than the specification of the macro–architecture, an indication how much of the complexity of the AAMP5 is contained within the microcode and how structured hardware designs tend to be. 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 micro–architecture specification took longer than the specification of the macro–architecture for a number of reasons. The AAMP5 micro–architecture 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 be substantially reduced if formal specification activity were integrated earlier and more closely into the conventional design cycle.

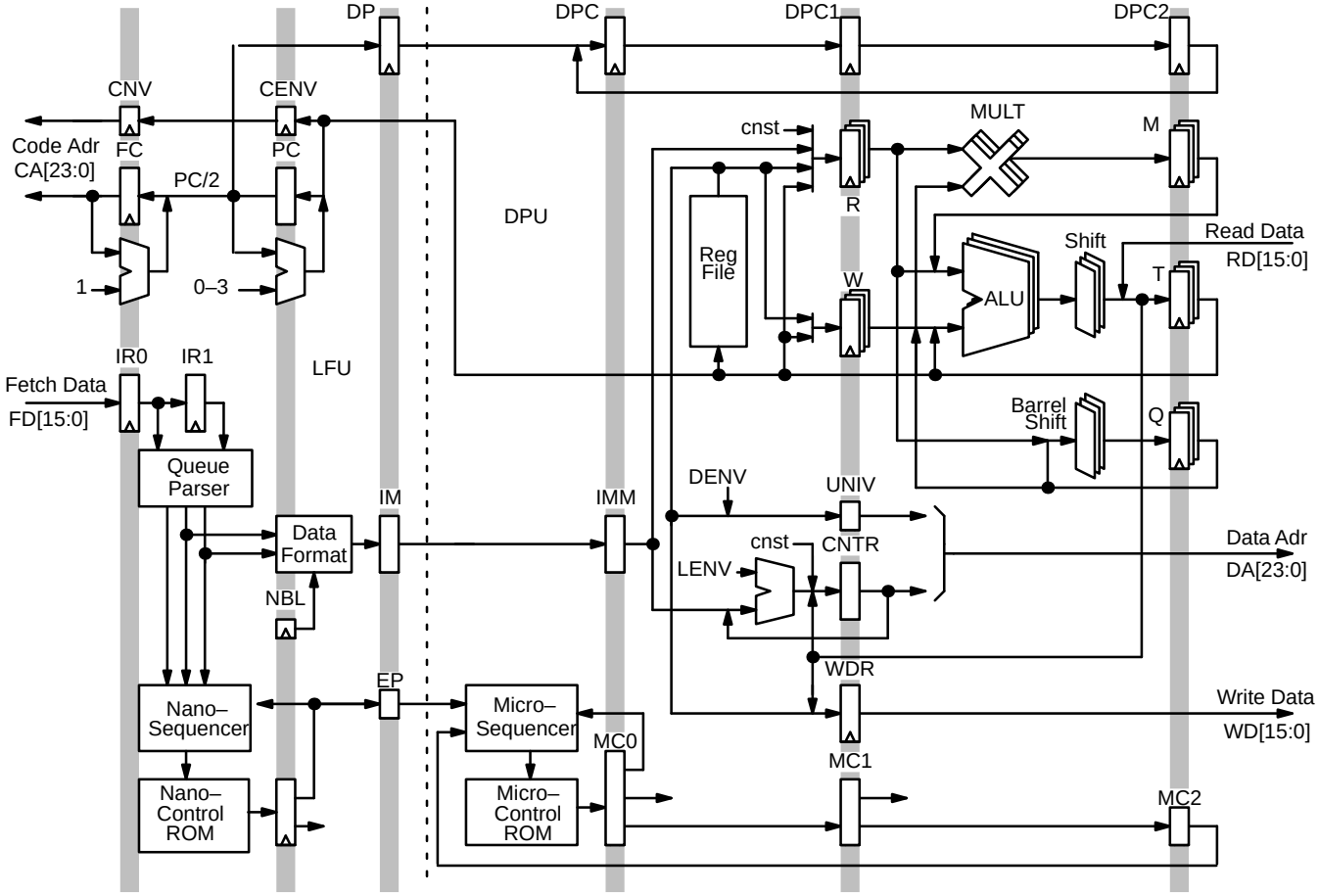

Figure 6 – AAMP5 Microarchitecture

5.2 Revision

To make the specifications acceptable to the AAMP5 designers for inspection, the initial micro–architecture 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 macro–architecture. Since the initial specifications covered the entire micro–architecture, there was no need for Collins to extend the specification as was done for the macro–architecture. 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 micro–architecture specification consisted of 2,679 lines of PVS and comments organized into 20 theories.

Revisions were also made later to the micro–architecture specifications to facilitate proofs, but these were more technical in nature and not as significant as the ones made to the macro–architecture 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 Inspection

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

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 macro–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 I, 83 hours were invested in conducting the inspections and 66 hours were spent correcting the defects found.

5.4 Translation of the Microcode to PVS

Translation of AAMP5 microcode to PVS was performed by hand for this project, even though the translation was straightforward enough that it could easily be automated. Hand translation of the microcode required approximately 1/2 hour per AAMP instruction.

6 Proofs of Correctness

In addition to verifying a core set of thirteen instructions, a key goal for the verification task was to facilitate the transfer of the verification technology to Collins so they would be able to verify instructions outside the core set. To support this, SRI developed the infrastructure necessary not only to verify the entire AAMP5 instruction set, but also to do it as automatically as possible.

Table III – Micro–Architecture 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

As of this reporting, we have completed the verification of eleven instructions (some of which are outside the core set) ranging over several major instruction classes, spending about 4 to 5 staff months on the verification task. Out of this time, about a month was spent on actually verifying the instructions. The rest of the time was spent in developing the general infrastructure that is reusable for all instructions. Verifying an instruction from a new class that has not been tried before typically takes 4 to 5 days to complete since it may involve extending the infrastructure. Verifying additional instructions within the same class typically takes up to a day.

6.1 Development of Proof Infrastructure

Once an instruction moves into the DPU, the instruction is interpreted in a two–stage pipeline. The first stage is used to read the register file and setup the operands required for the ALU operation. In the second stage the results of the (combinatorial) ALU are written to their destinations. A third stage is invoked only if an instruction causes a *delayed branch* in the instruction execution based on the outcome of the ALU operation. For example, an ALU overflow during an ADD instruction will cause a jump to the exception handler. In case of a delayed jump the instructions in the first two stages are aborted.

To verify the correctness of an instruction, we prove that the micro machine satisfies the *commuting* property shown in Figure 1. This property states that if the DPU is at an *instruction starting point*, i.e., a state in which the DPU is beginning to execute the first stage of the first micro–instruction of the microcode for the instruction, then the current instruction (1) will eventually complete, (2) will have the effect as stipulated in the macro specification, and (3) the first micro–instruction of the next AAMP5 instruction will move into the DPU.

The box in Figure 1 illustrates the requirement that the actual effect of the instruction execution must correspond with the expected result specified by the *next macro state* function. The function *Abstraction* shown in Figure 1 maps the state of a micro–machine at any given time into an object that corresponds to the abstract view used at the macro level. In addition to proving the condition described by Figure 1, we have to show that the special resetting (startup) sequence designed for the processor will force the processor into the appropriate instruction starting point. All AAMP5 interrupts, of which reset is an instance, have to be proved in a similar fashion.

Although the commuting diagram condition shown in Figure 1 is generally applicable to all microprocessors, it has to be refined in a number of ways to fit the idiosyncrasies of a particular processor. In the case of AAMP5, the design aspects that make this refinement challenging are the pipelining of the micro–instructions and the fact that instruction fetching and data transfers are handled autonomously. An adjustment is needed to handle pipelining because when the execution of a new instruction begins, the results of the previous uncompleted instruction may not yet be in place at their destinations. We handle this by ‘‘skewing’’ the definition of the abstraction function over time by an amount that is a function of the depth of the pipeline. That is, in the definition of *Abstraction*, values for the states of the micro–machine components that will be set by the uncompleted instructions must be obtained from a future micro–state.

The impact of autonomous instruction and data fetches is that the number of cycles required to complete an instruction, although finite, becomes indefinite. The DPU stalls, i.e., performs activities that do not have any externally visible effect, while waiting for the memory operations to complete. We handle this by decomposing the commuting condition proof into a set of *general verification conditions* that characterize the stalling behavior of the DPU and another set of *instruction specific verification conditions* that characterize the correctness of an instruction in the absence of stalling. The two sets of verification conditions are then combined to prove the correctness of an instruction. The proof of the general verification conditions and the combination proof only need to be performed once. The proofs of the general verification conditions rely on the specification of the DPU’s environment, namely the LFU and the BIU.

We have also developed a general methodology for formulating the verification conditions for each instruction. The general methodology can be customized to a particular AAMP5 instruction based on a set of attributes of the instruction such as its instruction class and the length of its microcode. We have developed a general proof strategy, i.e., a recipe involving a set of smaller proof steps, to prove the instruction specific verification conditions. The proof strategy uses *symbolic execution*, which is supported by PVS’s automatic rewriter, *case analysis*, which is supported by PVS’s BDD–based propositional simplifier, and *simplification* supported by PVS’s decision procedures. More details about this strategy, which is suitable for a variety of hardware verification applications, can be found in [12].

We estimate that about fifty percent of the infrastructure that has been developed could be reused. This includes general rules about bit–vector operations that are useful in automating the correctness proofs and the general hardware proof strategy. Most of the rest of the infrastructure should be reusable in the verification of another microprocessor belonging to the AAMP family.

6.2 Errors Discovered by the Proofs

The process of proving the microcode correct revealed several errors in the macro and micro–architecture specifications, even though they had been carefully

reviewed and inspected. More significant were errors discovered in the microcode itself. 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 proofs of correctness. To address this, two memory address calculation errors were deliberately inserted in the microcode without the knowledge of SRI. The first was designed to be unlikely to be detected by walkthroughs, testing, or simulation. The second was an actual error that had not been detected by traditional verification methods, but was found when running application code on an early fabrication of the AAMP5. Both errors were easily detected by SRI during the proof process, who also explained what corrections were needed. This served as a powerful demonstration of the value of formal verification.

7 Implications for Industry

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 [1] 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 does force 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 [21] 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 [21].

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

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

[2] Barrett, G., Formal Methods Applied to a Floating–Point Number System, *IEEE Transactions on Software Engineering*, Vol. 15, No. 5, pg. 611–621, May, 1989.

[3] Beatty, D. and R. Bryant, Formally Verifying a Microprocessor Using a Simulation Methodology, in *Proceedings of the 31st Design Automation Conference*, ACM, pg. 596–602, June, 1994.

[4] Best, D., C. Kress, N. Mykris, J. Russel, and W. Smith, *An Advanced–Architecture CMOS/SOS Microprocessor, IEEE Micro*, pg. 11–26, August, 1982.

[5] Boyer, R. and J. Moore, *A Computational Logic Handbook*, Academic Press, Inc.: San Diego, CA, 1988.

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

[7] Burch, J. and D. Dill, Automatic Verification of a Pipelined Microprocessor Control, *in Proceedings of Computer Aided Verification (CAV'94)*, LNCS 818, pg. 68–80, June 1994.

[8] *Butler, R.*, NASA Langley's Research Program in Formal Methods, in *Proceedings of the Sixth Annual Conference on Computer Assurance (COMPASS'91)*, pg. 157–162, Gaithersburg, MD, June 1991.

[9] Butler, R. and G. Finelli, The Infeasibility of Experimental Quantification of Life–Critical Software Reliability, *Software Engineering Notes*, Vol. 16, No.5, pg. 66–76, December 1991.

[10] Carter, W., W. Joyner, Jr., and D. Brand, Microprogram Verification Considered Necessary, in *Proceedings of the National Computer Conference*, AFIPS, pg. 657–664, Vol. 48, 1978.

[11] Cook, J., Verification of the C/30 Microcode Using the State Delta Verification System (SDVS), in *Proceedings of the 13th National Computer Security Conference*, National Institute of Standards and Technology/National Computer Security Center, pg. 20–31, Washington, D.C., Oct. 1990.

[12] Cyrluk, D., S. Rajan, N. Shankar, and M. Srivas, Effective Theorem Proving for Hardware Verification, in *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, R. Kumar and T. Kropf, Editors, pg. 287–305, Forschungszentrum Informatik an der Universit'at Karlsruhe, FZI Publication, 1994.

[13] Divito, B., R. Butler, and J. Caldwell, High Level Design Proof of a Reliable Computing Platform, in *Dependable Computing for Critical Applications – 2*, J. Meyer and R. Schlichting, Editors, pg. 279–306, Springer Verlag: Vienna, Austria, February 1991.

[14] Gordon, M. and T. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher–Order Logic*, Cambridge University Press: Cambridge, UK, 1993.

[15] Hunt Jr., W., FM8501: A Verified Microprocessor, *Lecture Notes in Artificial Intelligence*, Vol. 795, Springer–Verlag:Berlin, 1994.

[16] Hunt Jr., W. and B. Brock, A Formal HDL and its Use in the FM9001 Verification, in *Mechanized Reasoning and Hardware Design*, C. Hoare and M. Gordon, Editors, pg. 35–47, Prentice Hall International Series in Computer Science: Hemel Hempstead, UK, 1992.

[17] Gerhart, S., M. Bouler, K. Greene, D. Jamsek, T. Ralston, and D. Russinoff, *Formal Methods Transition Study Final Report*, MCC Report STP–FT–322–91, Microelectronics and Computer Technology Corporation, Austin, Texas, August 1991.

[18] Leeman, G., W. Carter, and A. Birman, Some Techniques for Microprogram Validation, in *Information Processing 74, Proc. IFIP Congress 1974*, North–Holland Publishing Co., pg. 76–80, 1974.

[19] Littlewood, B. and L. Strigini, Validation of Ultra–High Dependability for Software–based Systems, *Communications of the ACM*, Vol. 36, No. 11, pg. 69–80, November 1993.

[20] May, D., G. Barrett, and D. Shepherd, Designing Chips that Work, in *Mechanized Reasoning and Hardware Design*, C. Hoare and M. Gordon, Editors, pg. 3–19, Prentice Hall International: Hemel Hempstead, UK, 1992.

[21] McDermid, J., Formal Software Development Using Z, York Software Engineering Limited, a tutorial presented at the *Ninth Annual Conference on Computer Assurance (COMPASS'94)*, Gaithersburg, MD, June 27– July 1, 1994.

[22] Owre, S., J. Rushby, and N. Shankar, PVS: A Prototype Verification System, In Deepak Kapur, Editor, *11th International Conference on Automated Deduction, (CADE)*, pg. 748–752, Saratoga, NY, June 1992, Vol. 607 of Lecture Notes in Artificial Intelligence, Springer–Verlag.

[23] Owre, S., J. Rushby, N. Shankar, and F. von Henke, Formal Verification for Fault–Tolerant Architectures: Prolegomena to the Design of PVS, to appear in *IEEE Transactions on Software Engineering*, 1995.

[24] *Pentium Processor User's Manual, Volume 1: Pentium Processor Data Book*, Order Number 241428–001, Intel Corporation, 1993.

[25] Saxe J. and S. Garland, Using Transformations and Verification in Circuit Design, *Formal Methods in System Design*, Vol. 4, No. 1, pg. 181–210, 1994.

[26] Srivas, M., and M. Bickford, Formal Verification of a Pipelined Microprocessor, *IEEE Software*, Vol. 7, No. 5, pg. 52–64, September, 1990.

[27] Srivas, M. and M. Bickford, *Verification of the FtCayuga Fault–Tolerant Microprocessor System, Volume 1: A Case Study in Theorem Prover–Based Verification*, NASA Contractor Report 4381, July 1991.

[28] Srivas, M. and S. Miller, Formal Verification of the AAMP5: A Case Study in the Verification of a Commercial Microprocessor, to appear in *Applications of Formal Methods*, Michael G. Hinchey and Jonathan P. Bowen, Editors, Prentice–Hall International Series in Computer Science.

[29] Srivas, M. and S. Miller, *Formal Verification of an Avionics Microprocessor*, to be submitted as a NASA Contractor Report.

[30] Stavridou, V., Gordon's Computer: A Hardware Verification Case Study in OBJ*3*, *Formal Methods in System Design*, Vol. 4, No. 3, pg. 265–310, 1994.

[31] Windley, P., K. Levitt, and G. Cohen, *Formal Proof of the AVM–1 Microprocessor Using the Concept of Generic Interpreters*, NASA Contractor Report 187491, March 1991.

[32] Yu, Yuan, *Automated Proofs of Object Code for a Widely Used Microprocessor*, DEC/SRC Research Report 114, October 5, 1993.

[33] Windley, P. and M. Coe, A correctness model for pipelined microprocessors, in *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, R. Kumar and T. Kropf, Editors, pg. 35–54, Forschungszentrum Informatik an der Universit'at Karlsruhe, FZI Publication, 1994.