Specification and Verification of Pipelining in the ARM2 RISC Microprocessor

James K. Huggins
Kettering University, Flint, Michigan
and
David Van Campenhout
University of Michigan, Ann Arbor, Michigan

Gurevich Abstract State Machines (ASMs) provide a sound mathematical basis for the specification and verification of systems. An application of the ASM methodology to the verification of a pipelined microprocessor (an ARM2 implementation) is described. Both the sequential execution model and final pipelined model are formalized using ASMs. A series of intermediate models are introduced that gradually expose the complications of pipelining. The first intermediate model is proven equivalent to the sequential model in the absence of structural, control, and data hazards. In the following steps, these simplifying assumptions are lifted one by one, and the original proof is refined to establish the equivalence of each intermediate model with the sequential model, leading ultimately to a full proof of equivalence of the sequential and pipelined models.

Categories and Subject Descriptors: B.5.2 [**Hardware**]: Register transfer level implementation—*Design Aids*; C.1.1 [**Computer Systems Organization**]: Processor Architectures—*Single Data Stream Architectures*; C.0 [**Computer Systems Organization**]: General—*Systems specification methodology*

General Terms: Verification, Design

Additional Key Words and Phrases: formal verification, design verification, pipelined processors, pipelining, abstract state machines, ARM processor

1. INTRODUCTION

The Gurevich Abstract State Machine (ASM) methodology, formerly known as the evolving algebra methodology and first proposed in [Gurevich 1988], is a simple yet powerful methodology for specifying and verifying software and hardware systems. ASMs have been applied to a wide variety of software and hardware systems: pro-

A version of this paper was presented in [Huggins and Van Campenhout 1997] at the 1997 IEEE International High Level Design Validation and Test Workshop, Oakland, California, November 14-15, 1997.
J. K. Huggins was partially supported by ONR grant N00014-94-1-1182 and NSF grant CCR-95-04375. D. Van Campenhout was partially supported by SRC contract 95-DJ-338 and NSF grant MIP-9404632.

gramming languages, distributed protocols, architectures, and so on. See [Börger and Huggins 1998; Huggins 1998] for numerous examples.

In this paper we apply the ASM methodology to the verification of a pipelined implementation of the ARM2 microprocessor (hereafter ARM). The ARM is an early commercial RISC microprocessor [VLSI Technology 1990; Furber 1989]. Key features of this processor include a load/store architecture, a 32-bit datapath, conditional execution of every instruction, and a small but powerful instruction set.

Our starting point is a register transfer level description of the pipelined implementation and a textual description of the instruction set architecture (sequential model). We formalize both the sequential model and the pipelined implementation using ASMs. A series of intermediate models are introduced that gradually expose the complications of pipelining. The first intermediate model is proven equivalent to the sequential model in the absence of structural, control, and data hazards. In the following steps, these simplifying assumptions are lifted one by one, and the original proof is refined to establish the equivalence of each intermediate model with the sequential model.

The rest of the paper is organized as follows. Section 2 gives a self-contained introduction to sequential ASMs; the definitions given there are sufficient to understand this paper. Section 3 introduces the ARM microprocessor in greater detail. Section 4 presents the ASM of a non-pipelined version of the ARM processor. Section 5 describes a pipelined version that ignores the possible problems with branch and data dependency; this simple pipelined processor is proved to be equivalent to the non-pipelined version in an appropriate sense. Section 6 introduces further intermediate models that lead to the final pipelined version. Section 7 discusses the result and compares with other work.

2. ABSTRACT STATE MACHINES

The ASM thesis is that any software or hardware system can be modeled at its natural abstraction level by an abstract state machine. Based upon this thesis, members of the ASM community have sought to develop a methodology based upon mathematics which would allow such systems to be modeled naturally; that is, described at their natural abstraction levels. See [Börger and Huggins 1998; Huggins 1998] for a number of examples of ASMs applied to various real-world systems.

Sequential ASMs (under their former name, evolving algebras) are described in [Gurevich 1993]; a more detailed description of ASMs (including distributed characteristics) is given in [Gurevich 1995a]. We present here only those features of sequential ASMs necessary to understand this paper. Those already familiar with ASMs may wish to skip ahead to the next section.

2.1 States

The states of an ASM are structures in the sense of first-order logic, except that relations are treated as Boolean-valued functions.

A *vocabulary* is a finite collection of function names, each with a fixed arity. Every ASM vocabulary contains the following *logic symbols*: nullary function names *true*, *false*, *undef*, the equality sign, (the names of) the usual Boolean operations, and

a unary function name *Bool*. Some function symbols (such as *Bool*) are tagged as *relations*.

A *state* S of vocabulary Υ is a non-empty set X (the *superuniverse* of S), together with interpretations of all function symbols in Υ over X (the *basic functions* of S). A function symbol f of arity r is interpreted as an r-ary operation over X; if $r = 0$, f is interpreted as an element of X. The interpretations of the function symbols *true*, *false*, and *undef* are distinct, and are operated upon by the Boolean operations in the usual way.

Let f be a relation symbol of arity r. We require that (the interpretation of) f is *true* or *false* for every r-tuple of elements of S. If f is unary, it can be viewed as a *universe*: the set of elements a for which $f(a)$ evaluates to *true*. For example, *Bool* is a universe consisting of the two elements (named) *true* and *false*.

Let f be an r-ary basic function and $U_0, \ldots, U_r$ be universes. We say that f has *type* $U_1 \times \ldots \times U_r \rightarrow U_0$ in a given state if $f(\bar{x})$ is in the universe U_0 for every $\bar{x} \in U_1 \times \ldots \times U_r$, and $f(\bar{x})$ has the value *undef* otherwise.

2.2 Updates

The simplest change that can occur to a state is the change of an interpretation of a function at one particular tuple of arguments. We formalize this notion.

A *location* of a state S is a pair $\ell = (f, \bar{x})$, where f is an r-ary function name in the vocabulary of S and $\bar{x}$ is an r-tuple of elements of (the superuniverse of) S. (If f is nullary, ℓ is simply f.) An *update* α of a state S is a pair (ℓ, y), where ℓ is a location of S and y is an element of S. To *fire* α at S, put y into location ℓ; that is, if $\ell = (f, \bar{x})$, redefine S to interpret $f(\bar{x})$ as y and leave everything else unchanged.

2.3 Transition rules

We introduce rules for describing changes to states. At a given state S whose vocabulary includes that of a rule R, R gives rise to a set of updates; to execute R at S, fire all the updates in the corresponding update set. We suppose throughout that a state of discourse S has a sufficiently rich vocabulary.

An *update rule* R has the form

$$f(t_1, t_2, \ldots, t_n) := t_0$$

where f is an r-ary function name and each t_i is a term. (If $r = 0$, we write $f := t_0$ rather than $f() := t_0$.) The update set for R contains a single update (ℓ, y), where y is the value $(t_0)_S$ of t_0 at S, and $\ell = (f, (x_1, \ldots, x_r))$, where $x_i = (t_i)_S$. In other words, to execute R at S, set $f(x_1, \ldots, x_n)$ to y, where x_i is the value of t_i at S and y is the value of t_0 at S.

A *block rule* R is a sequence $R_1, \ldots, R_n$ of transition rules. To execute R at S, execute all the R_i at S simultaneously. That is, the update set of R at S is the union of the update sets of the R_i at S.

A *conditional rule* R has the form

$$\textbf{if } g \textbf{ then } R_0 \textbf{ else } R_1 \textbf{ endif}$$

where g (the *guard*) is a term and R_0, R_1 are rules. The meaning of R is the obvious one: if g evaluates to *true* in S, then the update set for R at S is the same as that for R_0 at S; otherwise, the update set for R at S is the same as that for R_1 at S.

A *parallel synchronous rule* (or *declaration rule*) R has the form

var v **ranges over** $c(v)$
$R_0(v)$
endvar

where v is a variable, $c(v)$ is a term involving variable v, and $R_0(v)$ is a rule with free variable v. To execute R in state S, execute simultaneously all rules $R(u)$, where u is an element of the superuniverse of S and $c(u)$ has the value *true* in S.

2.4 Programs and runs

A *program* Π is simply a transition rule (typically a block rule).

A *sequential run* ρ of program Π from an initial state S_0 is a sequence of states $S_0, S_1, \ldots$, where each S_{i+1} is obtained from S_i by executing program Π in state S_i.

A sequential ASM is thus given by a program and a collection of initial states; this determines a corresponding collection of runs of the ASM.

3. THE ARM MICROPROCESSOR

The ARM is a 32-bit microprocessor. In user mode, 16 general purpose registers (R0-R15) are visible to the programmer. Among those registers, R15 plays a special role as it serves as the program counter (PC). Moreover, R15 also contains the processor status register. The status register records certain events, such as overflow, that occur while executing an instruction. R14 also plays a special role; it is the register used to save the return address in branch-with-link instructions. The other registers (R0-R13) are truly interchangeable. The processor can also operate in three special modes, other than user mode, due to the occurrence of exceptions and interrupts. This feature is beyond the scope of the paper, as we focus on the effect of pipelining.

The implemented ARM instruction set architecture contains four types of instructions: (1) *ALU instructions* which perform operations on two operands (one operand may be shifted by a third operand), storing the result in the register file and the status flags; (2) *single data transfer instructions* which copy data between memory and the register file; (3) *multiple data transfer instructions* which copy data between memory and an arbitrary subset of the register file; and (4) *branch instructions* which alter program execution flow. The implementation of the ARM discussed in this paper does not incorporate the floating point instructions nor the multiply instructions.

An interesting feature of the instruction set is that the execution of every instruction is conditional. Every instruction contains a field which indicates under which conditions it is to be executed. If the condition is not met, the instruction is simply converted into a nop, i.e., it does not have any effect. The conditions refer to the condition register of the processor, which can be set by ALU instructions.

Register R15 serves as the program counter, but is also accessible like any other register in the register file. This has some interesting consequences. When R15 appears as the result register in an ALU instruction, or as the destination in single-load or multiple-load instructions, the sequential flow of the program is interrupted, and execution continues at the value stored into R15. Furthermore, R15 also stores

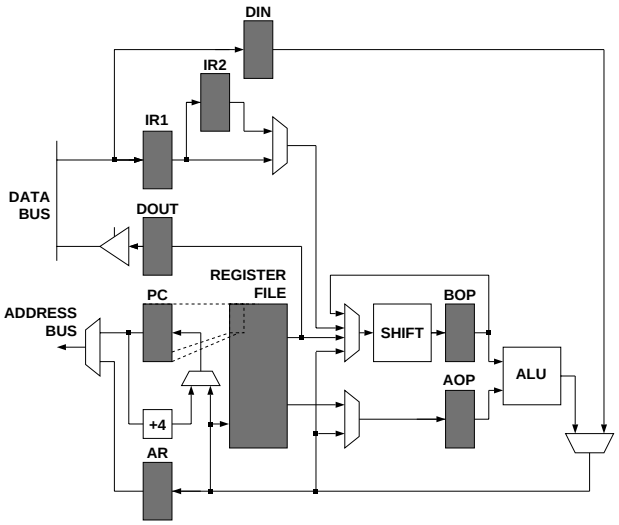

Fig. 1. Block diagram of ARM datapath.

the status bits of the processor. This is possible because the address space of the ARM2 can be addressed with 26 bits.

A block diagram of the datapath of our pipelined ARM implementation is shown in Figure 1. Not shown is the control section, which computes the signals that steer the datapath (such as select signals to multiplexers). Registers and the register file are colored gray, to indicate that they are clocked. The other units are constituted by combinational logic, such as multiplexers. The ISA's general purpose registers, R0-R15, are kept in the register file. The register file has two read ports and one write port. The PC is a special register, as it can be written both via the general write port of the register file and via a dedicated port used to increment the PC. The fact that the PC is also accessible through the register file is indicated in the figure by the dashed lines. The address sent to memory is either obtained from the PC, for instruction fetches, or from the address register AR, for data fetches during data transfer instructions. Data is transferred between the processor and memory via a bidirectional bus. During instruction fetches, any data transferred is stored in one of two instruction registers. For loads and stores, this data is transferred via the registers Din and Dout, respectively. Note that the registers IR1, IR2, AR, Din, Dout, Aop and Bop are not part of the ISA.

The ARM has a three-stage pipeline. This means that while the current instruction is in its execute stage, the next instruction is being decoded, and the instruction which follows the instruction being decoded is being fetched from memory. During the instruction fetch stage, the instruction at the address indicated by the PC is obtained from memory. During the decode stage of an instruction, the processor sets up the operands the instruction requires for its execution. For ALU instructions, typically two registers are read from the register file. One of them has a shift applied to its contents. During the execute stage, both operands Aop and

Bop are combined in the arithmetic and logic unit (ALU). The result is stored in the register file.

Often these three stages can take place in parallel, so that the throughput of the processor is one instruction per clock cycle. However, some instructions require more than one clock cycle for their execution. This causes stalls to occur. Only during the first cycle of an instruction's execute stage will the processor fetch a new instruction from memory. Only during the last cycle of an instruction's execute stage will the processor decode the next instruction. Registers IR1 and IR2 are used to store instructions waiting to be executed.

4. $\mathcal{E}_1$: SEQUENTIAL ARM MODEL

The sequential model of the ARM processes one instruction at a time. We formalize the sequential model in an ASM called $\mathcal{E}_1$. The ASM was derived from the textual instruction set architecture (ISA) [VLSI Technology 1990]. Anticipating the pipelined implementation, the processing of each instruction is divided into three stages: fetch, decode, and execute. We also make use of certain intermediate registers that are not in the ISA description. The ASM can be seen as an interpreter for the ARM instruction set. $\mathcal{E}_1$ processes instructions sequentially; that is, $\mathcal{E}_1$ completes its execution of a given ARM instruction before beginning to execute the next instruction in sequence.

The rules defining $\mathcal{E}_1$, as well as the universes (Table I) and functions (Table II) occurring in $\mathcal{E}_1$, are given in the Appendix. Notice that all rules fire simultaneously.

4.1 Definitions and discussion

Let $\rho =< \sigma_1, \sigma_2, \ldots >$ be a run of $\mathcal{E}_1$. An *execution cycle* (or simply a *cycle*) C of a run ρ of $\mathcal{E}_1$ is a subsequence $< \sigma_j, \sigma_{j+1}, \sigma_{j+2} >$ such that an instruction i is fetched in σ_j, decoded in σ_{j+1}, and executed in σ_{j+2}; i is the instruction *performed* by C. We refer to the three states of C, respectively, as the *fetch*, *decode*, and *execute stages* of C.

A cycle C is is *meaningful* if the instruction i performed by C is not a nop instruction. The *significant updates* of a meaningful cycle C are the updates to the memory, status bits, and register file performed in the execute stage of C, except for any update to $Contents(PC)$. Clearly each run $\rho =< \sigma_1, \sigma_2, \ldots >$ gives rise to a unique sequence of meaningful cycles $< C_1, C_2, \ldots >$.

Every instruction i has a corresponding set of *input locations*; these are the locations whose values, when i is executed, are directly used in the execution of i. For a given instruction i, there are at most four input locations (depending upon the instruction): two operand registers, a shift register, and the carry bit (stored in the status bits).

For simplicity of exposition, assume that the instructions of every ARM program are stored in consecutive words in memory. (This is not strictly necessary but makes the following explanations simpler.) We say that a program is *self-modification free* if the set of memory locations modified by the program is distinct from the memory locations containing program instructions. This eliminates the possibility of an instruction modifying the code being executed. Throughout this paper we will consider only programs which are self-modification free.

We can thus, without loss of generality, characterize the program being executed by the processor as a sequence of instructions $I =< i_0, i_1, \ldots >$ where instruction i_j is stored in memory location $b + 4j$ for some base address b. We say that such a program is *branch conflict free* if for every instruction i_j which may update the PC register, instructions i_{j+1} and i_{j+2} are nop instructions, and no other nop instructions appear in the program.

We say that two consecutive instructions i_j, i_{j+1} have a *data dependency* if one of the following conditions hold:

—Instruction i_j potentially writes to a register (other than PC) which serves as an operand to instruction i_{j+1}.

—Instruction i_j potentially updates the status condition flags (in particular, the carry bit) and instruction i_{j+1} is an ALU instruction (which may use the carry bit).

A program is *data dependency free* if every pair of consecutive instructions does not have a data dependency.

5. $\mathcal{E}_2$: FIRST PIPELINED MODEL

In this section, we introduce an intermediate model $\mathcal{E}_2$, which is a pipelined version of $\mathcal{E}_1$. We outline a proof of equivalence of $\mathcal{E}_1$ and $\mathcal{E}_2$; full details are given in [Huggins and Van Campenhout 1998].

5.1 Constructing $\mathcal{E}_2$ from $\mathcal{E}_1$

In the sequential model, $\mathcal{E}_1$, an instruction is processed in three steps. Ignoring structural, data, and control hazards, a pipelined version can be derived by overlapping the processing of three consecutive instructions. Let i_j, i_{j+1}, i_{j+2} be consecutive ARM instructions in an ARM program. If i_j, i_{j+1}, and i_{j+2} are "independent" (in an appropriate sense), we can decode instruction i_{j+1} at the same time as we are executing instruction i_j. Further, we can fetch instruction i_{j+2} at the same time as these other two actions.

In $\mathcal{E}_1$, the instruction being executed is held in a single function *Instr*. To reflect the pipeline of $\mathcal{E}_2$, we add new functions *FetchInstr* [1], *DecodeInstr*, and *ExecuteInstr* which hold the instructions being fetched, decoded, and executed, respectively. We also modify the decode rule and execute rules to read from their respective registers and to move instructions through the pipeline simultaneously.

5.2 Proof of equivalence

We now outline our proof that $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, in an appropriate sense to be defined here. We fix an ARM-program Π and show that $\mathcal{E}_1$ and $\mathcal{E}_2$ generate the same sequence of "significant updates" when executing Π. Throughout this section, we assume that the ARM program being executed by $\mathcal{E}_1$ and $\mathcal{E}_2$ is both branch conflict and data dependency free.

First, some definitions. Let $\rho =< \sigma_1, \sigma_2, \ldots >$ be a run of $\mathcal{E}_2$. An *execution cycle* (or simply a *cycle*) C of a run ρ of $\mathcal{E}_2$ is any three element subsequence $< \sigma_j, \sigma_{j+1}, \sigma_{j+2} >$. We refer to the three states of C, respectively, as the *fetch*,

[1] Technically, an abbreviation for a term involving other functions.

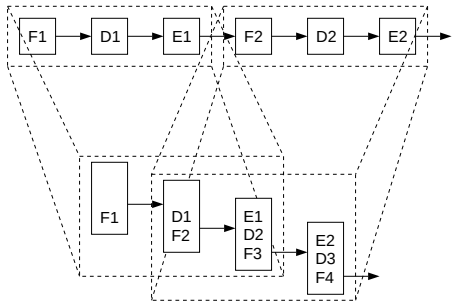

Fig. 2. Corresponding cycles in $\mathcal{E}_1$ and $\mathcal{E}_2$.

decode, and *execute* stages of C. We say that instruction i is *performed* during C if i is the value of *FetchInstr* (respectively, *DecodeInstr*, *ExecuteInstr*) during the fetch (respectively, decode, execute) stage of C. The notions of meaningful instruction cycle and significant updates are unchanged.

We assert that the initial states of $\mathcal{E}_1$ and $\mathcal{E}_2$ agree with respect to their common functions, i.e., external memory, the register file, and the status bits.

Let $\rho =< s_1, s_2, \ldots >$ be a run of $\mathcal{E}_1$ with corresponding sequence of meaningful cycles $< C_1, C_2, \ldots >$. Let $\rho' =< s'_1, s'_2, \ldots >$ be a run of $\mathcal{E}_2$ with corresponding sequence of meaningful cycles $< C'_1, C'_2, \ldots >$.

We say that execution cycles C and C' of $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, *correspond* if:

—C and C' agree on the value of PC (and thus fetch the same instruction) in their fetch stages.

—C and C' agree on the values of all input locations (with respect to the instruction just fetched) in their decode stages.

—C and C' agree on the values of the memory, status bits, and the register file (with the possible exception of PC) in their execute stages.

This notion of correspondence is illustrated in Figure 2. Notice that a given ASM state of $\mathcal{E}_1$ is a member of at most one instruction cycle of $\mathcal{E}_1$, while an ASM state of $\mathcal{E}_2$ may be a member of up to three instruction cycles of $\mathcal{E}_2$.

The heart of the proof of correctness lies in the following three lemmas:

LEMMA 1. *(Update Lemma) Suppose execution cycles C and C' correspond. Then the significant updates of C and C' are identical.*

LEMMA 2. *(PC Lemma) Suppose C_j and C'_j correspond. Then the fetch stages of C_{j+1} and C'_{j+1} agree with respect to the value of the PC register.*

LEMMA 3. *(Correspondence Lemma) For every $j \geq 1$, C_j and C'_j correspond.*

The first two lemmas are proven by a straightforward analysis of the updates generated by a given execution cycle. The last lemma is proven by a straightforward induction over the number of instruction cycles already executed.

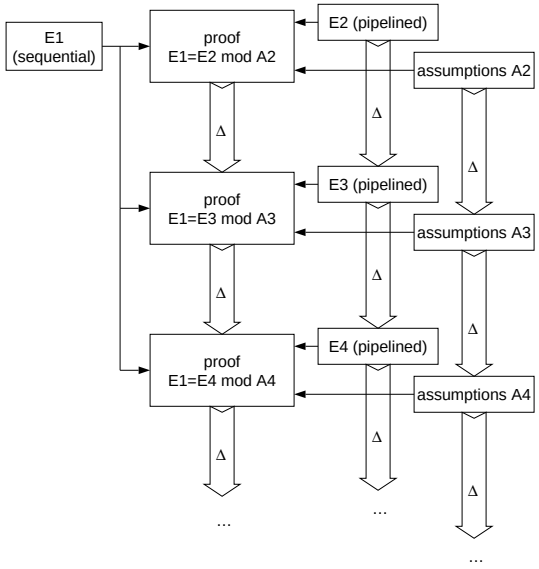

Fig. 3. Refinement in verification.

THEOREM 1. *(Equivalence Theorem) The sequence of update sets* $< U_1, U_2, \ldots >$ *and* $< U_1', U_2', \ldots >$ *produced by* $< C_1, C_2, \ldots >$ *and* $< C_1', C_2', \ldots >$*, respectively, are identical.*

The theorem is an immediate consequence of the Update Lemma and Correspondence Lemma.

6. FURTHER REVISIONS

In the remainder of the verification process, the simple pipelined model $\mathcal{E}_2$ is gradually refined. In each revision another complication of pipelining the sequential model is exposed, and the simplifying assumptions on valid programs are relaxed. A new proof is constructed that establishes the equivalence of the new pipelined model to the sequential model. A key strength of this approach is that during the construction of the new proof, one has to concentrate only on the particular aspect introduced in the new model. Most of the proof is simply inherited from the previous version. The process of refinement is illustrated in Figure 3; full details on each of the refined models and refined proofs may be found in [Huggins and Van Campenhout 1998].

6.1 $\mathcal{E}_3$: Memory system constraints

The intermediate model $\mathcal{E}_3$ reflects the structural constraints of the memory interface. In the ARM implementation, only one word may be transferred between memory and the register file in a given clock cycle. This poses the following two constraints:

M1. Only one word or byte may be transferred per clock cycle (step).

M2. The address used in a data transfer instruction must be available at the beginning of the clock cycle.

Constraint M1 is violated in several ways by $\mathcal{E}_2$. The multiple transfer instructions allow an arbitrary subset of registers to be transferred to or from memory in a single clock cycle. In $\mathcal{E}_3$, these rules are refined so that these transfers are serialized. Notice also that the instruction fetch stage accesses the memory to fetch a new instruction; thus, our serialization must be careful not to conflict with the fetching of the next instruction from memory.

Constraint M2 requires several modifications to $\mathcal{E}_2$ as well. Data transfer instructions require the generation of an address to be used during the transfer. In the ARM implementation, this address is generated using the ALU. Consequently, the actual data transfer has to take place one cycle after the address is generated.

The ARM implementation complies with these constraints by "stalling" the pipeline during the execution of data transfer instructions. This means the remaining instructions in the pipeline are delayed from advancing through the pipeline until the data transfer instruction has finished executing. The stalling mechanism is incorporated in the intermediate model $\mathcal{E}_3$ by modifying some of the rules so that instructions do not proceed through the pipeline until the last step of a data transfer instruction is being performed. After revising the Update Lemma to ensure that the serialization works correctly, the rest of the correctness proof is borrowed from that of $\mathcal{E}_2$.

6.2 $\mathcal{E}_4$: Branches

In the next intermediate model $\mathcal{E}_4$, the simplifying assumption that branch instructions are to be followed by two nop instructions is lifted. The model reveals the squashing mechanism on taken branches.

The problem which we must now solve is as follows. Recall that in our pipelined model, while we are executing an instruction i_j, we are also decoding the next instruction i_{j+1} and fetching the even later instruction i_{j+2}. However, if i_j is a branch instruction and its condition codes are satisfied, the instruction which should be executed following i_j may not be i_{j+1}, but some completely different instruction i_k. Thus, the instructions which are currently in the pipeline must be discarded, and the pipeline must be allowed to fill with instructions starting with i_k.

The revised model $\mathcal{E}_4$ contains new rules which are used to refill the pipeline (and discard any old values from the pipeline) when an explicit update to the PC register is made. The proof of correctness requires a straightforward re-verification of the PC Lemma, to ensure that the sequence of meaningful instructions executed is still identical.

6.3 $\mathcal{E}_5$: Data dependencies

$\mathcal{E}_5$ exposes the forwarding paths in the pipeline which deal with data hazards. Recall that a data dependency occurs if there are two consecutive instructions i_j, i_{j+1} such that i_{j+1} uses the contents of a register r as input while i_j modifies register r. In our pipelined model, this is problematic because i_j writes to register r (in its execute stage) at the same time as i_{j+1} reads from register r (in its decode stage). A similar problem results with the status flags; i_j may update the status

register during its execute stage while i_{j+1} is reading the status register to perform a shift during its decode stage.

The ARM implementation includes forwarding paths that make the value being written by instruction i_j directly available to the decode stage of i_{j+1}, thus bypassing the register file. These forwarding paths are reflected in $\mathcal{E}_5$ by redefining the decode rules. Constructing the new proof of correctness involves a revision of the Update Lemma, to ensure that the correct values are in fact available at each stage.

6.4 $\mathcal{E}_6$: Register file restrictions

In the final model, $\mathcal{E}_6$, the structural constraint that the register file has only two read ports and one write port is introduced. This means that at most two registers may be read and at most one register may be written during any step of $\mathcal{E}_6$. (The special register PC, which is read and/or written at virtually every step of the ARM, is exempted from this requirement.) This affects certain types of ALU instructions: namely, those instructions which require a shift where the shift amount is specified by a register. Such instructions require three registers: two for the operands, and one to indicate the amount of the shift.

In the ARM implementation, such an instruction i stalls the pipeline for one clock cycle. During i's decode stage, the ARM uses its two read ports to calculate the shifted operand (using one register for the pre-shifted value and another for the amount of the shift), while ignoring the second operand. During the first step of i's execute stage, the proper value for the second operand is loaded; the required ALU operation is then performed during the second step of i's execute stage.

$\mathcal{E}_6$ incorporates rules which introduce a one-step stall (similar to that needed for data transfer rules treated in $\mathcal{E}_3$) in order to handle the above situation. The proof of correctness requires a small change to the Update Lemma (similar to that performed in verifying $\mathcal{E}_3$) to verify that the above stall still generates the same significant updates.

7. DISCUSSION

Automated formal verification for microprocessors has become an intensely researched area. The spectrum of methods [McFarland 1993] can be characterized as follows. On the one end, there are highly automated methods whose power is limited in the type of properties that can be expressed and handled. These methods also tend to fail on real-life sized systems unless an highly abstracted system model is used, which most often must be derived manually. On the opposite end, there are methods that require substantial user assistance, but offer richer expressiveness and facilitate hierarchical analysis.

Levitt and Olukotun [Levitt and Olukotun 1996] proposed a methodology for verifying pipelined processors. The datapath functional units are assumed to be correct and are represented by uninterpreted functions. The methodology iteratively merges the two deepest pipeline stages until the sequential model is obtained. In each iteration, the equivalence between the old pipeline and the new pipeline is proved. As in our work, the proof is by induction on the number of execution cycles. The induction hypothesis is derived automatically and is checked automatically with a validity checker [Jones, Dill, and Burch 1995]. Once a proper description

of the pipelined and sequential model in terms of uninterpreted functions has been written, the method is highly automatic.

Börger and Mazzanti [Börger and Mazzanti 1997] applied the ASM methodology for the first time to microprocessor verification. They proved the correctness of a pipelined version of the DLX processor [Hennessy and Patterson 1990] with respect to the sequential specification. The overall structure of the approach is similar to our work, although the architecture and the micro-architecture (pipeline) of the DLX and the ARM differ significantly.

One of the benefits of the ASM approach is its ability to treat a given system at multiple layers of abstraction. Notice that each successive layer of abstraction in our specification of the ARM deals chiefly with one particular problem (e.g. structural constraints, control hazards) in isolation, allowing the reader to understand which components of the ARM specification ensure the correctness of pipelining in the presence of those problems. To the extent that problems such as data dependency and control dependency are independent, we have treated the solutions and verified the correctness of those solutions independently as well. We argue that it may be easier to understand a proof of correctness presented in this stepwise fashion; notice the first pipelined model presented is the "ideal" pipeline, which can often be obscured in the final model which contains all the "fixes" which must be applied to the ideal model.

The selection of the proper levels of abstraction is a creative act, which varies from application to application. One can always tailor an ASM to a higher or a lower level of abstraction, depending on one's interests. For example, we could have constructed a truly sequential model $\mathcal{E}_0$ in which an ARM instruction is performed in exactly one step, rather than the three steps taken by $\mathcal{E}_1$, and then proven the equivalence of $\mathcal{E}_0$ and $\mathcal{E}_1$. Similarly, we could have revised the final model $\mathcal{E}_6$ to create models of the ARM at even lower levels of abstraction. Our models were chosen in order to effectively demonstrate the difference between pipelined and non-pipelined versions of the same microprocessor.

While our specification and verification was based on a completed hardware design for the processor, the same techniques could easily (and more beneficially) be applied during the design process. We constructed our levels of abstraction through trial and error, creating new models as needed and revising previous models when errors became apparent. If applied during the design process, the natural levels of abstraction used by the designers could also be used for this specification and verification task, leading to rapid generation and more accurate models of the system being developed and verified.

One does not need to have the complete model in hand in order to begin the process of verification. Notice that even in the first proof, the assumptions which must be made to complete the proof (namely, that the program to be executed is data dependency, branch dependency, and self-modification free) are a useful discovery by themselves; these

assumptions point out the problems which must be treated by successive models (or else the assumptions must be guaranteed to hold by some other means). In addition, these techniques could be applied to more detailed models of the processor, if desired.

Our proof techniques are hand-oriented, intended for consumption by a human audience, and as such have been subjected to the traditional scrutiny given to handwritten proofs. Our proofs do not require sophisticated tools such as higher-order or temporal logics; in most cases, induction is the only tool which is required.

Machine-verified proofs are certainly of great value; it may be more difficult, however, for readers to understand a machine-oriented proof than a human-oriented proof (since a different audience is addressed by such a proof) [Gurevich 1995b]. There is no automated proof assistant specifically designed for ASMs; however, ASM specifications have been used as the starting point for mechanical verification using proof assistants such as KIV [Schellhorn and Ahrendt 1997], PVS [Zimmerman and Gaul 1997], and SMV [Winter 1997]. In fact, when using certain interactive proof assistants such as PVS, it is often important to have a clear outline of the proof before beginning verification [Hooman 1996]; a hand-oriented specification and proof such as that presented here more than fulfills that need.

Conclusion

We applied the ASM methodology to the verification of a pipelined microprocessor, the ARM. The processor was modeled at multiple levels of abstraction that bridge the gap between the sequential view of the machine and the detailed pipelined implementation. With this basis, we proved the correctness of the ARM's pipelining techniques by proving each model equivalent to the sequential model, re-using the proofs of correctness of higher levels while proving the correctness of lower levels.

ACKNOWLEDGMENTS

We thank Trevor Mudge for originally suggesting the project to us and supporting its development. We also thank Egon Börger, Stefan Küng and several anonymous referees who made suggestions on earlier drafts of this paper.

References

Börger, E. and Huggins, J. K. 1998. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS 64*, (Feb.), 105–127.

Börger, E. and Mazzanti, S. 1997. A practical method for rigorously controllable hardware design. In *ZUM'97: The Z Formal Specification Notation: 10th International Conference of Z Users*, J. Bowen, M. Hinchey, and D. Till, Eds. (1997). Springer Lecture Notes in Computer Science 1212, 151–187.

Furber, S. B. 1989. *VLSI RISC architecture and organization.* M. Dekker, New York.

Gurevich, Y. 1988. Logic and the challenge of computer science. In *Current Trends in Theoretical Computer Science*, E. Börger, Ed., 1–57. Computer Science Press.

Gurevich, Y. 1993. Evolving algebras: An attempt to discover semantics. In *Current Trends in Theoretical Computer Science*, G. Rozenberg and A. Salomaa, Eds., 266–292. World Scientific. (First published in *Bulletin of EATCS 43* (Feb.), 264–284.).

Gurevich, Y. 1995a. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, E. Börger, Ed., 9–36. Oxford University Press.

Gurevich, Y. 1995b. Platonism, constructivism, and computer proofs vs. proofs by hand. *Bulletin of EATCS 47*, (Oct.), 145–166.

Hennessy, J. and Patterson, D. 1990. *Computer Architecture: A quantitative Approach.* Morgan Kaufman Publishers, San Mateo, Calif. Revised second edition, 1996.

Hooman, J. 1996. Using PVS for an assertional verification of the RPC-memory specification problem. In *Formal Systems Specification; The RPC-Memory Specification Case Study*, 275–304. Springer Lecture Notes in Computer Science 1169.

Huggins, J. K., Ed. 1998. *Abstract State Machine Home Page*. EECS Department, University of Michigan, http://www.eecs.umich.edu/gasm/.

Huggins, J. K. and Van Campenhout, D. 1997. Specification and verification of pipelining in the ARM2 RISC microprocessor. In *Digest IEEE Int. High Level Design Validation and Test Workshop* (1997). 186–193.

Huggins, J. K. and Van Campenhout, D. 1998. Specification and verification of pipelining in the ARM2 RISC microprocessor. Technical Report CSE-TR-371-98, University of Michigan EECS Department.

Jones, R., Dill, D., and Burch, J. 1995. Efficient validity checking for processor verification. In *Proc. IEEE International Conference on Computer Aided Design* (Nov. 1995). 2–6.

Levitt, J. and Olukotun, K. 1996. Scalable formal verification methodology for pipelined microprocessors. In *Proc. 33rd ACM/IEEE Design Automation Conference* (1996). 558–563.

McFarland, M. C. 1993. Formal verification of sequential hardware. a tutorial. *IEEE Transactions on Computer-Aided Design 12*, (May), 633–654.

Schellhorn, G. and Ahrendt, W. 1997. Reasoning about abstract state machines: The WAM case study. *Journal of Universal Computer Science 3*, 4, 377–413.

VLSI Technology, I. 1990. *Acorn RISC machine (ARM) family data manual*. Englewood Clifs, N.J.: Prentice Hall.

Winter, K. 1997. Model checking for abstract state machines. *Journal of Universal Computer Science 3*, 5, 689–701.

Zimmerman, W. and Gaul, T. 1997. On the construction of correct compiler back-ends: An ASM approach. *Journal of Universal Computer Science 3*, 5, 504–567.

APPENDIX A: SEQUENTIAL MODEL $\mathcal{E}_1$

Rule: Fetch
if *FetchOK* **then**
    Instr := FetchInstr
    Stage := decode
endif

where
    FetchOK abbreviates Stage=Fetch
    FetchInstr abbreviates MemoryWord(Contents(PC))
    MemoryWord(x) abbreviates
        Word(Memory(x),Memory(x+1),Memory(x+2),Memory(x+3))

Rule: ExecutePC
if *ExecuteOK* **then**
    if not *Satisfies(Status,CondCode(Instr))* **or not** *WritesPC(Instr)* **then**
        Contents(PC) := Contents(PC) + 4
    endif
endif

where ExecuteOK abbreviates Stage=Execute

Rule: ExecuteNop
if *ExecuteOK* **and** *Nop(Instr)* **then** *Stage := fetch* **endif**

Rule: Decode
if *DecodeOK* **then**
    Stage := execute
    if not *Nop(Instr)* **then**
        DestReg := DestOp(Instr)
        Aop := Contents'(AopReg(Instr))
        Bop := Shift(SourceVal,ShiftType(Instr),ShiftAmt,Carry(Status))
        ShiftCarryOp := ShiftCarry(SourceVal,ShiftType(Instr),ShiftAmt,Carry(Status))
    endif
endif

where
    DecodeOK abbreviates Stage=decode
    Contents'(x) abbreviates IfThenElse(x ≠ PC,Contents(x),Contents(PC)+8)
    SourceVal abbreviates
        IfThenElse(ImmBop(Instr),ImmediateVal(Instr),Contents'(BopReg(Instr)))
    ShiftAmt abbreviates
        IfThenElse(ImmShift(Instr),ImmShiftAmt(Instr),Contents'(ShiftReg(Instr)))

Rule: ExecuteALU
if *ExecuteOK* **and** *AluInstr(Instr)* **then**
    if *Satisfies(Status,CondCode(Instr))* **then**
        if *WriteResult(Instr)* **then**
            Contents(DestReg) := ALU(ALUop(Instr), Aop, Bop, Carry(Status))
        endif
        if *SetCondCode(Instr)* **then**
            Status := UpdateStatus(Status,ALUop(Instr),Aop,Bop,ShiftCarryOp)
        endif
    endif
    Stage := fetch
endif

Rule: ExecuteBranch
if *ExecuteOK* **and** *BranchInstr(Instr)* **then**
    if *Satisfies(Status,CondCode(Instr))* **then**
        Contents(PC) := ALU("+", Aop, Bop, 0)
        if *BranchWithLinkInstr(Instr)* **then**
            Contents(LinkReg) := Contents'(PC) - 4
        endif
    endif
    Stage := fetch
endif

```
Rule: ExecuteSingleTransfer
if ExecuteOK and SingleTransferInstr(Instr) then
    if Satisfies(Status,CondCode(Instr))then
        if LoadInstr(Instr) then
            if ByteTransferInstr(Instr) then
                Contents(DestReg) := PadWord(Memory(MemAddr))
            else Contents(DestReg) := MemoryWord(MemAddr)
            endif
        elseif StoreInstr(Instr) then
            if ByteTransferInstr(Instr) then
                Memory(MemAddr) := ByteExtract(Contents'(DestReg),0)
            else AssignWord(MemAddr,Contents'(DestReg))
            endif
        endif
        if WriteBack(Instr) then Contents(BaseOp(Instr)) := Aop + Offset
        endif
    endif
    Stage := fetch
endif

where
    AssignWord(l,v) abbreviates:
        Memory(l) := ByteExtract(v,0)          Memory(l+2) := ByteExtract(v,2)
        Memory(l+1) := ByteExtract(v,1)        Memory(l+3) := ByteExtract(v,3)
    MemAddr abbreviates IfThenElse(PreIndexed(Instr),Aop + Offset,Aop)
    Offset abbreviates IfThenElse(IncrOp(Instr),Bop,-Bop)


Rule: ExecuteMultipleTransfer
if ExecuteOK and MultipleTransferInstr(Instr) then
    if Satisfies(Status,CondCode(Instr)) then
        var r ranges over TransferReg(r,Instr)
            if LoadInstr(Instr) then
                Contents(r) := MemoryWord(Aop + Bop + 4*NumPrevRegs(r,Instr))
            else AssignWord(Aop + Bop + 4*NumPrevRegs(r,Instr),WriteVal)
            endif
        endvar
        if WriteBack(Instr) and not (LoadInstr(Instr)
            and TransferReg(BaseOp(Instr),Instr)) then
            Contents(BaseOp(Instr)) := WriteBackVal
        endif
    endif
    Stage := fetch
endif

where
    WriteVal abbreviates
        IfThenElse(r = BaseOp(Instr) and NumPrevRegs(r) ≥ 1 and WriteBack(Instr),
        WriteBackVal, Contents'(r))
    WriteBackVal abbreviates Aop + Bop + 4*NumRegs(Instr)
```

Universe	Description
stages	$\{fetch, decode, execute\}$
Bool	*true* or *false*
bits	$\{0, 1\}$
words	$\{0, \ldots, 2^{32} - 1\}$
instructions	valid ARM instructions
registers	{R0,...,R15}
flaglists	list of control flags
ALUops	opcodes for ALU instructions
shifts	types of shifts

Table I. Universes in $\mathcal{E}_1$

Function	Type
Stage	*stages*
Instr	*instructions*
LinkReg, DestReg	*registers*
Status	*flaglists*
Aop, Bop	*words*
ShiftCarryOp	*bits*
MemoryWord	$words \rightarrow words$
Contents, Contents'	$registers \rightarrow words$
ALU	$ALUops \times words \times words \times bits \rightarrow words$
Shift	$words \times shifts \times words \times bits \rightarrow words$
ShiftCarry	$words \times shifts \times words \times bits \rightarrow bits$
UpdateStatus	$flaglists \times ALUops \times words \times words \times bits \rightarrow flaglists$
AopReg, BopReg, ShiftReg, DestOp	$instructions \rightarrow registers$
ShiftType	$instructions \rightarrow shifts$
Carry	$flaglists \rightarrow bits$
ImmBop, ImmediateVal, ImmShiftAmnt	$instructions \rightarrow words$
ImmShift, AluInstr, Nop	$instructions \rightarrow Bool$
WritesResult, WritesPC	$instructions \rightarrow Bool$
BranchInstr, BranchWithLinkInstr	$instructions \rightarrow Bool$
CondCode	$instructions \rightarrow flaglists$
ALUop	$instructions \rightarrow ALUops$
Satisfies	$flaglists \times flaglists \rightarrow Bool$

Table II. Some functions in $\mathcal{E}_1$ and their types

REVIEWS AND RESPONSES

Review #1

This is an area of growing importance. The paper has been written carefully. Nevertheless, the paper is filled with details such that it is almost impossible to follow every detail. Readability of the paper severely suffers from this. Attempts should be made to improve readability and possibly shorten the paper.

Concerning the technical content, it should be mentioned that processor verification using ASM methodology has already been published by Börger and Mazzanti. The new aspects of this work are limited to the introduction of a step-wise proof procedure and to using a new example.

Also, the technique requires a human being to analyse the equations and to believe that the equations actually correspond to the real hardware.

Despite these limitations, I would recommend to accept the paper.

Response

We recognize that the detail and length of the original manuscript might pose a serious obstacle to the reader. Furthermore we realize that the original manuscript does not meet the maximum length requirement. Therefore we have omitted the technical detail of the intermediate models as well the proofs. We refer the interested reader to the original manuscript which we have made available in its entirety as a technical report [Huggins and Van Campenhout 1998].

We agree with the reviewer on the relationship of our work with that of Börger and Mazzanti, and have tried to clarify this in the discussion.

Review #2

This paper applies the ASM (Abstract State Machine) principles for the verification of the ARM microprocessor. The authors consider a pipelined version of the microprocessor - which is a practical case.

The authors describe the step-by-step verification process starting with the sequential description of the ARM down to the detailed pipelined implementation.

The paper is however very long! The authors need to reduce the paper size significantly. The details may either be provided as a reference to a thesis and/or to a WEB location

Response

See response to review #1.