Specification and Verification of Pipelining in the ARM2 RISC Microprocessor *

James K. Huggins†
Kettering University
Flint, Michigan

David Van Campenhout‡
University of Michigan
Ann Arbor, Michigan

August 14, 1998

Abstract

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.

1 Introduction

The Gurevich Abstract State Machine (ASM) methodology, formerly known as the evolving algebra methodology and first proposed in [4], 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: programming languages, distributed protocols, architectures, and so on. See [1, 10] 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 [16, 3]. 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. We begin with a self-contained introduction to sequential ASMs in section 2; the definitions given there are sufficient to understand this paper. Section 3 introduces the ARM microprocessor in greater detail. In section 4 we present the ASM of a non-pipelined version of

*A version of this paper was presented in [11] 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.

the ARM processor. Section 5 describes a pipelined version, ignoring 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. In section 6, further intermediate models are introduced that lead to the final pipelined version, which is given in Appendix A. 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 [1, 10] for a number of examples of ASMs applied to various real-world systems.

Sequential ASMs (under their former name, evolving algebras) are described in [5]; a more detailed description of ASMs (including distributed characteristics) is given in [6]. 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

if g **then** R_0 **else** R_1 **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.

3.1 ARM instruction set architecture

In this section we give an overview of the implemented ARM instruction set. These instructions fall into four categories:

ALU instructions perform a logical operation (such as bitwise-or) or an arithmetic operation (such as subtraction) on two operands, storing the result in the register file. The first operand is always (the contents of) a register. The second operand can either be an immediate value, encoded in the instruction word, or a

register. In either case, the second operand can be subjected to a shift before being used. Up to five types of shifts are supported, and the number of bits by which the operand is to be shifted can either be encoded in the instruction word or can be specified by a register. The result of applying an ALU operation is stored in the register file, with an optional update to the status flags. Compare instructions which do not store a result in the register file but always affect the status flags are also classified as ALU instructions.

Single data transfer instructions copy the contents of a register to a specified memory location or vice versa. Depending on the transfer direction the instruction is a load (from memory) or a store (to memory). The address of the memory location can be computed in several ways. An offset is added or subtracted from the value of a base register, which is a specified general-purpose register. The offset is either directly specified in the instruction word or obtained by shifting a directly specified quantity by a number of bits (also contained within the instruction word). The actual address used for the data transfer can be either the base address or the modified base address as described above. Furthermore, the register containing the base address may be updated with the computed address if desired.

Multiple data transfer instructions copy the contents of an arbitrary subset of registers to contiguous locations in memory, or vice versa. The location with the lowest address corresponds to the register with the smallest register number. Four different ways to compute the address of the lowest location involved are provided. The address of the lowest location in memory is a function of the content of the base register and the number of registers involved in the transfer. Again, the base register can be updated if desired.

Branch instructions allow the linear flow of the program execution to be interrupted. After execution of a branch instruction, the execution is continued with the instruction at a computed address. The address is specified by an offset relative to the program counter. For branch-with-link instructions, the address of the instruction (sequentially) following the branch instruction can be saved in the link register.

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 15 (R15) plays a special role in the ARM, as it serves as the program counter (PC). As this register is 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 the status bits of the processor. This is possible because the address space of the ARM2 can be addressed with 26 bits.

3.2 Hardware implementation

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 AR, Din, Dout, Aop and Bop are not part of the ISA.

The processor 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

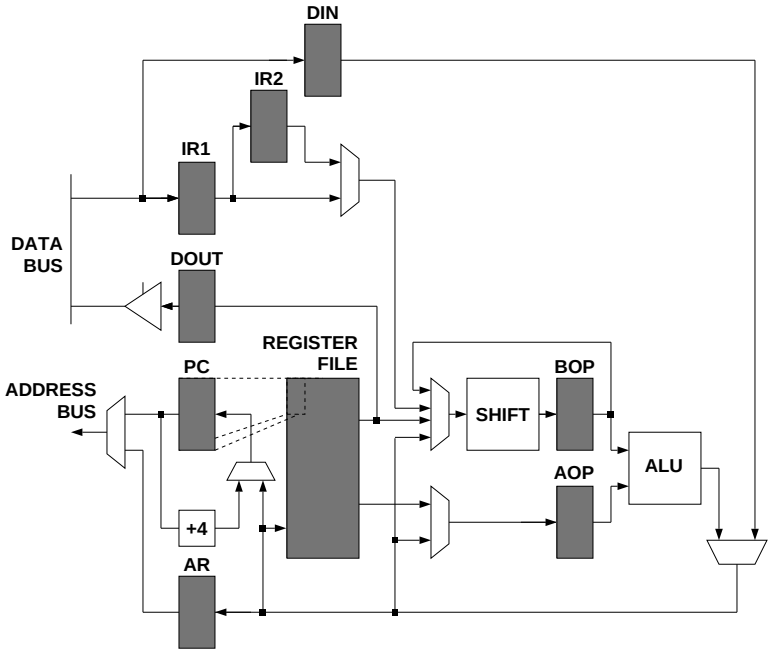

Figure 1: Block diagram of ARM datapath.

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) [16]. 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 cannot be found 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.

4.1 Some universes and functions

The ARM processor manipulates values from an abstract universe of *words*, of which the universe of *addresses* is a subset. Words are composed of four *bytes*, each of which is a value in the range $\{0, \ldots, 255\}$. Functions *Word: bytes*$^4 \rightarrow$ *words* and *ByteExtract: words* $\times$ *bytes* $\rightarrow$ *bytes* are used to translate between a word and its constituent bytes. The unary function *Memory: addresses* $\rightarrow$ *bytes* represents the memory of the computer system where the ARM processor stores and retrieves various values.

The ARM processor's register file is represented by a universe of *registers* and a function *Contents: registers* $\rightarrow$ *words*. A distinguished element *PC: registers* is the ARM's program counter, storing the address of the currently-executing program instruction. A distinguished element *LinkReg: registers* indicates the register to be used for "branch-with-link" operations, where the address of the instruction following the branch instruction is stored when a branch is performed. This feature can be used to implement subroutine calls: the calling routine leaves the return address in the link register to be used when the subroutine terminates.

The ARM processor performs instructions from a specified instruction set; the universe of *instructions* (a subset of the universe of *words*) represents this set. The nullary function *Instr: instructions* indicates the current instruction being executed.

$\mathcal{E}_1$ takes three steps, or *stages*, to execute each instruction: the fetch stage (in which the instruction is loaded from memory), the decode stage (in which the loaded instruction is decoded to determine the operands needed for the instruction), and the execute stage (in which the instruction is actually performed). A universe of *stages* contains elements *fetch*, *decode*, and *execute* to represent these steps. The nullary function *Stage: stages* indicates the current stage of execution of a given instruction.

Every instruction in the ARM instruction set is conditionally executed, depending upon a set of status flags. Universes of *bits* and *flaglists* are used to represent this information; flaglists are lists of bits. The nullary function *Status: flaglists* represents the current status flags of the ARM processor; various static functions such as *Carry: flaglists* $\rightarrow$ *bits* are used to extract information from this list (in this case, whether or not the last ALU instruction generated a carry bit).

To abbreviate some rules, we make use of a function *IfThenElse: Bool* $\times$ *words* $\times$ *words* $\rightarrow$ *words* which serves as a conditional expression. That is:

$$IfThenElse(true,term1,term2) = term1$$
$$IfThenElse(false,term1,term2) = term2$$

4.2 Fetch rule

The program for $\mathcal{E}_1$ is a block of transition rules. We present each of these rules individually in the following sections.

In the fetch stage, the ARM reads the instruction stored in memory at the address currently indicated by the program counter (PC) register. The transition rule in Figure 2 shows this activity.

FetchOK abbreviates an expression indicating that the fetch rule should fire; *FetchInstr* abbreviates an expression indicating the instruction to be fetched from memory. At this point it may seem simpler not to use these abbreviations and to incorporate their definitions in the transition rule above. In later sections, we will redefine these abbreviations with more complicated expressions; using these abbreviations now facilitates this revision.

4.3 Decode rule

The ARM instructions can be classified into the following categories:

- Instructions which do nothing (nops)
- Instructions which perform an arithmetic or logic operation, using the processor's arithmetic logic unit (ALU).

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))

Figure 2: Fetch rule.

- Instructions which cause execution to branch to a different location in memory
- Instructions which cause one byte or word to be transferred between the register file and memory
- Instructions which cause several words to be transferred between the register file and memory

The unary functions *Nop, Aluinstr, BranchInstr, SingleTransferInstr, MultipleTransferInstr: instructions* $\rightarrow$ *Bool* indicate whether or not a given instruction is in the corresponding category.

The decode stage retrieves the operands necessary to perform the instruction. The operands are temporarily stored in nullary functions. The functions *Aop, Bop: words* contain the two operands to be manipulated during the execute stage e.g. the values to be added by the ALU). The function *DestReg: registers* indicates (when appropriate) the register where the result of the instruction is to be stored.

The value of the first operand, *Aop*, is always the value stored in a specified register; *AopReg: instructions* $\rightarrow$ *registers* indicates this register. Obtaining the value of the second operand *Bop* is more involved.

Bop could be obtained immediately from the instruction itself (i.e. a constant), or could be obtained from a register. The function *ImmBop: instructions* $\rightarrow$ *Bool* indicates whether *Bop* should be obtained immediately from the instruction; the function *ImmediateVal: instructions* $\rightarrow$ *words* gives that immediate value. If *Bop* should be obtained from a register, the function *BopReg: instructions* $\rightarrow$ *registers* indicates that register.

Additionally, the value of *Bop* (regardless of how it is obtained) may have a mathematical "shift" operation performed upon it before it is used. The universe of *shifts* represents the types of shifts which may be performed. *ShiftType: instructions* $\rightarrow$ *shifts* indicates the shift called for by a given instruction. The function *Shift: words* $\times$ *shifts* $\times$ *words* $\times$ *bits* $\rightarrow$ *words* is used to perform the shift. *Shift(val,shifttype,amt,carry)* performs a shift of type *shifttype* for *amt* bits, possibly using the bit *carry* to fill-in shifted positions. The related function *ShiftCarry: words* $\times$ *shifts* $\times$ *words* $\times$ *bits* $\rightarrow$ *bits* gives the corresponding carry bit for the specified operation; the nullary function *ShiftCarryOp: bits* is used to store this information for possible use by the execute rules.

The magnitude of the shift can be specified in two ways. The instruction itself may specify the shift amount; alternatively, the shift amount may be the contents of a specified register. The function *ImmShift: instructions* $\rightarrow$ *Bool* indicates whether the given instruction specifies an immediate shift amount; if so, *ImmShiftAmt: instructions* $\rightarrow$ *words* indicates that amount. Otherwise, *ShiftReg: instructions* $\rightarrow$ *registers* indicates the register containing the shift amount.

The function *DestOp: instructions* $\rightarrow$ *registers* indicates the destination register (if any) to be modified by a given instruction.

The transition rule for the decode stage is given in Figure 3.

```
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)))
```

Figure 3: Decode rule.

The observant reader may notice that in operations involving the PC register, the value retrieved from the register file is not *Contents(PC)* but rather *Contents(PC) + 8*. This is part of the ARM instruction set architecture; it anticipates certain aspects of the pipelined design which will be made apparent in later sections.

4.4 Execute rules

The ARM processor executes every instruction conditionally, depending upon the values of the status flags stored (in our model) in the nullary function *Status*. The function *CondCode: instructions* $\rightarrow$ *flaglists* indicates the conditions under which an instruction should be executed; the function *Satisfies: flaglists* $\times$ *flaglists* $\rightarrow$ *Bool* indicates whether the current status flags satisfy the corresponding condition.

4.4.1 Incrementing PC

The execute stage always modifies the PC register. Usually, the contents of the PC register is incremented by 4. The exception arises when PC is explicitly modified by an instruction (for example, a branch instruction). The function *WritesPC: instructions* $\rightarrow$ *Bool* indicates whether a given instruction explicitly attempts to write to the PC register.

Thus, in every execute stage, the PC register is incremented by 4 if the condition attached to the instruction fails, or if the instruction does not attempt to write to the PC register. The transition rule which performs this activity is shown in Figure 4.

4.4.2 Nop instructions

As seen in Figure 5 below, nop instructions have no effect on the system, other than resetting *Stage* to *fetch* (as will every other execute stage transition rule to follow).

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

Figure 4: Rule for incrementing PC register.

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

Figure 5: Nop rule.

4.4.3 ALU instructions

Mathematical operations involving the arithmetic and logic unit (ALU) of the ARM processor require four pieces of information: the operation to be performed, the two arguments for that operation, and the last carry bit set (which is part of the processor's status flag set). The universe of $ALUops$ represents the operations which may be performed by the ALU; the function $ALUOp\colon instructions \rightarrow ALUops$ indicates the ALU operation called for by a given instruction.

An ALU instruction may call for an update to the specified destination register and/or an update to the status flags. The functions $WriteResult\colon instructions \rightarrow Bool$ and $SetCondCode\colon instructions \rightarrow Bool$ indicate which of these actions are required.

The function $ALU\colon ALUops \times words \times words \times bits \rightarrow words$ performs the requested mathematical operation; *ALU(op, word1, word2, carry)* performs operation *op* on arguments *word1* and *word2*, possibly using the carry bit *carry*. A similar function $UpdateStatus\colon flaglists \times ALUops \times words \times words \times bits \rightarrow flaglists$ computes the status resulting from this instruction; *UpdateStatus(oldflags, op, word1, word2, carry)* produces the new status flag list obtained from *oldflags* by performing *op* on arguments *word1* and *word2*, possibly using the bit *carry* generated by the shifter during the decode stage.

The transition rule for ALU instructions is given in Figure 6.

4.4.4 Branch instructions

Branch instructions call for a change to the normal sequential flow of the program. The address for the next instruction to be executed is the sum of the current value of the PC register and an immediate offset (contained in the instruction word). To execute a branch, the current value of the PC register is placed in Aop and the desired offset is placed in Bop during the decode stage (that is, $AopReg(i) = PC$ for any branch instruction i); these values are then added and placed in the PC register. Branch-with-link instructions perform an additional action: they store the address of the instruction which would normally be performed next in a specific register called the *link register*. The function $BranchWithLinkInstr\colon instructions \rightarrow Bool$ indicates whether an instruction calls for that linking action; the static nullary function $LinkReg\colon registers$ indicates which register is used for this purpose.

```
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
```

Figure 6: ALU instruction rule.

The transition rule for branch instructions is shown in Figure 7.

```
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
```

Figure 7: Branch instruction rule.

4.4.5 Single-transfer instructions

Single-transfer instructions call for a transfer of a single byte or a single word between memory and the register file. The transfer may load data into a register from memory or store data from a register into memory; the function *LoadInstr: instructions* $\rightarrow$ *Bool* indicates which action is required for a given instruction. Also, the datum transferred may be a single byte or an entire four-byte word; the function *ByteTransferInstr: instructions* $\rightarrow$ *Bool* gives this information.

In the event that only a single byte is being loaded from memory, a static function *PadWord: bytes* $\rightarrow$ *words* converts the given byte into an equivalent 4-byte word.

The abbreviation *MemAddr* indicates the location in memory involved in this data transfer. *MemAddr* is an expression involving the base address and an offset value calculated during the decode stage and placed in *Aop* and *Bop*, respectively. Certain instructions (called *pre-indexed* instructions) call for using the base address without the offset value: *PreIndexed: instructions* $\rightarrow$ *Bool* indicates these instructions. Certain instructions call for the base address to be incremented by the offset value, while others call for the base

address to be decremented by that value; *IncrOp: instructions* $\rightarrow$ *Bool* indicates if a given instruction calls for incrementing the base address.

Certain instructions call for the base address plus the offset to be written back to the base register (the register from which the base address was retrieved). *WriteBack: instructions* $\rightarrow$ *Bool* indicates if a given instruction should perform this "write-back" operation; *BaseOp: instructions* $\rightarrow$ *registers* indicates the base register (note that if *WriteBack(i)* is true for an instruction i, then *BaseOp(i)* = *AopReg(i)*).

The ARM instruction set architecture specifies that *BaseOp(i)* = *PC* implies *WriteBack(i)* = *false*; that is, single-transfer instructions never call for write-back to the program counter *PC*.

The transition rule for single-transfer instructions is shown in Figure 8.

```
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)
```

Figure 8: Single-transfer rule.

4.4.6 Multiple-transfer instructions

Multiple-transfer instructions call for multiple words to be transferred between a subset of the register file and a sequence of consecutive locations in memory. Depending on the transfer direction, these instructions are either multiple-load instructions or multiple-store instructions. The registers to be used are indicated by a binary function *TransferReg: registers* $\times$ *instructions* $\rightarrow$ *Bool*; *TransferReg(r,i)* is true if instruction i calls for transfer to or from register r.

The relationship between the transfer registers and the memory locations involved in the operation is such that the numerical order of the register numbers is the same as the numerical order of the addresses. For example, an instruction which calls for loading registers 1, 3, and 6 from memory location 1000 would load the word beginning at location 1000 in register 1, the word beginning at location 1004 in register 3, and the word beginning at location 1008 in register 6. The function *NumPrevRegs: registers* $\times$ *instructions* $\rightarrow$ *integers* indicates the number of registers previous to a given register that must be transferred. That is, *NumPrevRegs(r,i)* indicates how many registers prior to register *r* must also be transferred.

As with single transfer instructions, multiple-transfer instructions have a "write-back" option. The *PC* register is prohibited from serving as a base register; hence, no write-back update can occur to the *PC* register. It is possible that write-back can be specified and the base register also occurs in the list of registers to be written to memory; the value which actually should be written to memory for that register is a little complicated to explain. If this register is the first register to be written to memory, the result is the value which resided originally in the register; otherwise, the result is the write-back value. (This definition reflects how the instruction will be implemented in the pipelined version; this convoluted definition will become clearer at that time.)

The transition rule for multiple-transfer instructions is shown in Figure 9.

```
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)
```

Figure 9: Multiple-transfer rules.

4.5 Definitions and discussion

Let Υ_V be the vocabulary containing the function names *Contents*, *Status*, and *Memory*; these "visible" functions constitute the ISA's view of the state of the processor.

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: *Stage = fetch* (respectively, *decode*, *execute*) in σ_j (respectively, σ_{j+1}, σ_{j+2}). We refer to the three states of C, respectively, as the *fetch*, *decode*, and *execute stages* of C.

We say that instruction i is *performed* by C if *FetchInstr* $= i$ holds in the fetch stage of C and *Instr* $= i$ holds in the decode and execute stages of C; C is a *meaningful cycle* if i is not a nop instruction. The *significant updates* of a meaningful cycle C are the updates to functions in Υ_V 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): *Contents(AopReg(i))*, *Contents(BopReg(i))*, *Contents(ShiftReg(i))*, and *Status*. (Technically, every instruction is dependent upon *Status*, since every instruction is conditionally executed. But only certain ALU instructions use the carry flag stored in *Status* as an operand; it is these instructions for which we consider *Status* to be an input location.)

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 such that *WritesPC(i_j)* is true, 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 present a proof of equivalence of $\mathcal{E}_1$ and $\mathcal{E}_2$.

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.

To reflect the pipeline of $\mathcal{E}_2$, we add new functions *DecodeInstr, ExecuteInstr* which will hold the instructions being decoded and executed, respectively. We change the rule *Fetch* by substituting *DecodeInstr* for *Instr*; we also change the rule *Decode* to read from *DecodeInstr* instead of *Instr*. Similarly, we add the update

ExecuteInstr := DecodeInstr to rule *Decode*, and change all of the execute rules to read from *ExecuteInstr* instead of *Instr*.

Of course, we wish the fetch, decode, and execute rules to execute at every step, so we remove all references to the function *Stage* from all rules and redefine the abbreviations *FetchOK*, *DecodeOK*, and *ExecuteOK* to the constant value *true*.

Notice that the contents of a register used in a particular instruction are read during its decode stage. If the register in question is the *PC* register (which holds the address of the instruction being executed), the value of that register should be incremented by 8 before the value is used. In our pipelined model, however, the contents of the *PC* register are usually incremented by 4 at every step; thus, the value of the *PC* register seen by the decode stage is actually 4 greater than the address of the instruction being decoded. Consequently, we only need to increment the value seen in the *PC* register by 4 during the decode stage. We thus redefine the abbreviation *Contents'(x)*, used only during the decode stage, to:

IfThenElse(x≠PC,Contents(x),Contents(PC)+4)

In the initial state of $\mathcal{E}_2$, *DecodeInstr* and *ExecuteInstr* are both nop instructions (i.e., *Nop(DecodeInstr) = Nop(ExecuteInstr) = true*).

5.2 Proof of equivalence

We now proceed to prove 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*, *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; C is a *meaningful cycle* if i is not a nop instruction. As before, each run $\rho =< \sigma_1, \sigma_2, \ldots >$ gives rise to a unique sequence of meaningful cycles $< C_1, C_2, \ldots >$.

The *significant updates* of a meaningful cycle C which performs instruction i are all the updates to functions in Υ_V performed in the execute stage of C, except for any update to *Contents(PC)*.

In this section, let s_1 and s_1' be initial states of $\mathcal{E}_1$, and $\mathcal{E}_2$, respectively, such that $s_1|\Upsilon_V = s_1'|\Upsilon_V$ (that is, the reducts of s_1 and s_2 to the common vocabulary Υ_V are identical). 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 *Contents(PC)* (and thus on the value of *FetchInstr*) in their fetch stages.
- C and C' agree on the values of all input locations (with respect to *Instr* and *DecodeInstr*) in their decode stages.
- C and C' agree on the values of *Memory*, *Status*, and *Contents* (with the exception of *Contents(PC)*) in their execute stages.

This notion of correspondence is illustrated in Figure 10.

Lemma 1 *(Consecutive Instruction Lemma) Fix a state s of $\mathcal{E}_2$. Let i_e, i_d, and i_f be the values of ExecuteInstr, DecodeInstr, and FetchInstr, respectively. Let a be the value of Contents(PC), i.e., i_f=MemoryWord(a). If i_e is not a nop instruction, then i_e=MemoryWord(a − 8) and i_d=MemoryWord(a − 4). Further, if i_d is not a nop instruction, then i_d=MemoryWord(a − 4).*

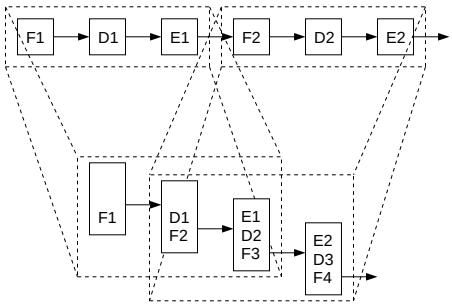

Figure 10: Corresponding cycles in $\mathcal{E}_1$ and $\mathcal{E}_2$.

Proof. By induction over states. The invariant condition is trivially true in the initial state, since i_e and i_d are both nop instructions.

Consider an arbitrary state in which the desired condition holds. In every state, *FetchInstr* is moved to *DecodeInstr* and *DecodeInstr* is moved to *ExecuteInstr*. Usually *Contents(PC)* is incremented by 4, which maintains the invariant. The exception occurs when *WritesPC(i_e)* and *Satisfies(Status,CondCode(i_e))* are true in s; in this case, *Contents(PC)* is updated to an arbitrary value. By the invariant condition, i_d and i_f are the two instructions which follow i_e in memory; since the program stored in memory is branch-conflict free, these are both nop instructions. Thus, i_e and i_d will be nop instructions in the successor state of s, maintaining the invariant. □

Lemma 2 *(Nop Pipe Lemma) Fix a state s of $\mathcal{E}_2$. Let i_e, i_d, and i_f be the values of* ExecuteInstr*,* Decode-Instr*, and* FetchInstr*, respectively. If WritesPC(i_e) is true, then i_d and i_f are nop instructions. Further, if WritesPC(i_d) is true, then i_f is a nop instruction.*

Proof. Since *WritesPC(i_e)* is true, i_e is not a nop instruction. By the Consecutive Instruction Lemma the instructions i_d and i_f are stored in memory immediately after i_e; since the program being executed is branch-conflict free and *WritesPC(i_e)* is true, these instructions are nops. The case for i_d is similar. □

Lemma 3 *(Update Lemma) Suppose execution cycles C and C' correspond. Then the significant updates of C and C' are identical.*

Proof. Let $C =< s_1, s_2, s_3 >$ and $C' =< s'_1, s'_2, s'_3 >$, and consider the three corresponding stages of C and C'.

- **Fetch.** Since C and C' correspond, s_1 and s'_1 agree on the value of *Contents(PC)*, and thus the value of *Instr* in s_2 and the value of *DecodeInstr* in s'_2 are identical (recall that memory locations containing instructions are never changed, since the program is self-modification free). Thus, the instruction performed by C and C' is identical.

- **Decode.** Since the instruction performed by C and C' is identical, and C and C' correspond, the values of the input locations for *Instr* (or *DecodeInstr*) are identical. The updates performed to *Aop*, *Bop*, *DestReg*, and *ShiftCarryOp* in either s_2 or s'_2 depend only upon the instruction being performed, the input locations for the instruction, and various static functions; thus, the values of these four functions will be identical in s_3 and s'_3. Additionally, the value of *ExecuteInstr* in s'_3 will match the value of *Instr* in s_3.

- **Execute.** Since C and C' correspond, the values of *Memory* and *Status* agree in s_3 and s'_3. The updates to *Memory*, *Status*, and *Contents* performed in s_3 and s'_3 (with the possible exception of *Contents(PC)*) depend upon the values of *Memory* and *Status* (identical from correspondence), various static functions, and the functions updated in the decode stage (which we have already shown to be identical). Thus, the significant updates performed are identical. □

Corollary 1 *(Update Corollary) Let C and C' be corresponding cycles. Let s and s' be the states immediately following the execute stages of C and C', respectively. Then s and s' agree with respect to* Memory, Status, *and* Contents *(except for* Contents(PC)*).*

Proof. By the Update Lemma, the execute stages agree with respect to *Memory*, *Status*, and *Contents* (except for *Contents(PC)*) and generate the same updates to these functions. □

Lemma 4 *(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* Contents(PC).

Proof. By supposition, since C_j and C'_j agree in their fetch stages with respect to *Contents(PC)*, C_j and C'_j perform the same instruction i. There are several cases.

- *WritesPC(i)* is true, and *Satisfies(Status,CondCode(i))* holds in the execute stage of C_j (and by correspondence, in C'_j). Then the value of *Contents(PC)* in the fetch stage of C_{j+1} is determined by the updates executed in the execute stage of C_j.

  Notice that by the Nop Pipe Lemma, two nop instructions will follow i in the pipeline for $\mathcal{E}_2$. Since nop instructions do not appear in significant execution cycles, the next significant instruction cycle C'_{j+1} will not begin until the state following the execute stage of C'_j (when a non-nop instruction will appear in the pipeline). Thus, the value of *Contents(PC)* in the fetch stage of C'_{j+1} is determined by the updates executed in the execute stage of C'_j.

  An argument similar to that given in the Update Lemma shows that the updates to *Contents(PC)* performed in the execute stages of C_j and C'_j are identical.

- *WritesPC(i)* is true, but *Satisfies(Status,CondCode(i))* does not hold in the execute stage of C_j (and by correspondence, in C'_j). Then *Contents(PC)* will be incremented by 4 in the execute stage of C_j. $\mathcal{E}_1$ will then proceed through two non-meaningful execution cycles (since i is always followed by two nop instructions), incrementing *Contents(PC)* at the end of each cycle. Thus, the value of *Contents(PC)* in the fetch stage of C_{j+1} is 12 greater than its value in the fetch stage of C_j.

  As in the previous case, two nop instructions will follow i in the pipeline for $\mathcal{E}_2$, and the value of *Contents(PC)* in the fetch stage of C'_{j+1} is determined by the updates executed in the execute stage of C'_j. Since *Satisfies(Status,CondCode(i'))* does not hold in that stage, *Contents(PC)* will be incremented by 4; by the Consecutive Instruction Lemma, the value of *Contents(PC)* at this time is 8 more than the address of i (which is the value of *Contents(PC)* in the fetch stage of C_j). Thus, the value of *Contents(PC)* in the fetch stage of C_{j+1} is 12 greater than its value in the fetch stage of C'_{j+1}.

- *WritesPC(i)* is false. Then *Contents(PC)* is incremented by 4 in the execute stage of C_j.

  Consider the fetch stage s of C'_j. Since C_j and C'_j are meaningful instruction cycles, the instruction i being performed in C_j and C'_j is not a nop instruction. The Nop Pipe Lemma implies that *WritesPC(ExecuteInstr)* is false in s; thus, *Contents(PC)* will be incremented by 4 in s. The Nop Pipe Lemma also implies that *WritesPC(DecodeInstr)* is false in s. Since the program being executed is branch-conflict free, and lies in continuous memory, the instruction immediately following i is not a nop instruction. Thus, C_{j+1} will begin in the state immediately following the fetch stage of C_j; thus, the value of *Contents(PC)* in the fetch stage of C_{j+1} is determined by the update to *Contents(PC)* in the fetch stage of C_j; as discussed above, *Contents(PC)* is incremented by 4 in that state. □

Lemma 5 *(Correspondence Lemma) For every $j \geq 1$, C_j and C'_j correspond.*

Proof. By induction over j.

Consider $C_1 =< s_1, s_2, s_3 >$ and $C'_1 =< s'_1, s'_2, s'_3 >$. By construction, s_1 and s'_1 agree with respect to *Contents*, *Memory*, and *Status*. Notice that $\mathcal{E}_1$ performs no updates to these functions in s_1, and $\mathcal{E}_2$ only increments *Contents(PC)* by 4; taking into account the differing definitions of *Contents'*, we know that s_2 and s'_2 agree with respect to *Contents'*, *Memory*, and *Status*, and thus agree over all input locations. Neither s_2 nor s'_2 update these functions (again, with the exception of *Contents(PC)*, so s_3 and s'_3 agree over these functions. Thus C_1 and C'_1 correspond.

For the inductive step, assume that C_k and C'_k coincide for every $k \leq j$, and consider the three stages of C_{j+1} and C'_{j+1}.

- **Fetch.** We must show that *Contents(PC)* has the same value in the fetch stages of C_{j+1} and C'_{j+1}; this is the PC Lemma above.

- **Decode.** Let s and s' be the decode stages of C_{j+1} and C'_{j+1}, respectively. We must show that all input locations of s and s' agree. Since the fetch stages of C_{j+1} and C'_{j+1} agree on the value of *Contents(PC)*, we know that the value of *Instr* and *DecodeInstr* in s and s' agree, and thus the set of input locations in C_{j+1} and C'_{j+1} are identical.

  Notice that s', the decode stage of C'_{j+1}, may also be the execute stage of C'_j (if *ExecuteInstr* is not a nop instruction) or occur strictly after the execute stage of C'_j (if *ExecuteInstr* is a nop instruction). We consider each case in turn.

  - **Case 1.** Suppose *ExecuteInstr* is a nop instruction; that is, the execute stage of C'_j occurs strictly before s'. By the Update Corollary, the state r which follows the execute stage of C'_j agrees with the fetch stage of C_{j+1} with respect to *Status*, *Memory*, and *Contents* (except for *Contents(PC)*). Since C'_j is the last significant cycle before C_{j+1}, s' agrees with r with respect to these locations (notice that r and s may be the same state). Further, since $\mathcal{E}_1$ does not modify any of these locations in its fetch stages, s agrees with r (and s') with respect to these locations. We must still show that s and s' agree with respect to *Contents'(PC)*; see below.
  - **Case 2.** Suppose s' is both the execute stage of C'_j and the decode stage of C_{j+1}. Then the most we can assert is that, with respect to the input locations of C_j except for *Contents(PC)*, s' agrees with the state following C_{j-1}. If the instruction i being performed relies on updates performed by C_j, the input locations of s and s' may not have the same values. But our data-dependency assertion specifically disallows this condition, so s and s' will agree with respect to these locations.

  It remains to show that s and s' agree with respect to *Contents'(PC)*. Since i is not a nop instruction, by the Nop Pipe Lemma, the instruction i' in *ExecuteInstr* when i was in *FetchInstr* did not satisfy *WritesPC(i')*. Thus, *Contents(PC)* was incremented by 4 in the fetch stage of C'_{j+1}, and thus its value in s' is 4 greater than its value in s. But the definition of *Contents'* in $\mathcal{E}_2$ reports a value 4 less for *Contents(PC)*; thus, s and s' agree with respect to *Contents'(PC)*.

- **Execute.** The correspondence of the execute stages is easy to observe; by the Update Corollary, the states which follow the execute stages of C_j and C'_j, agree with respect to *Status*, *Memory*, and *Contents* (except for *Contents(PC)*). No other updates to these locations occur between these states and the execute stages of C_{j+1} and C'_{j+1}. □

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.*

Proof. A simple induction using the Correspondence Lemma and the Update Lemma. □

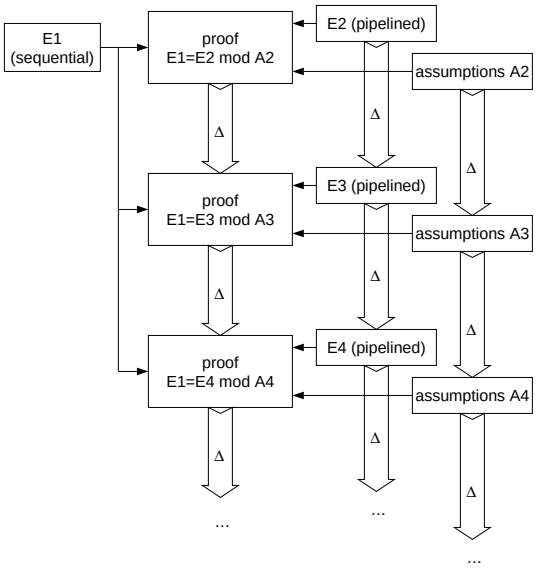

Figure 11: Refinement in Verification.

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 11.

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

The intermediate model, $\mathcal{E}_3$, reflects the structural constraints of the memory interface. In the 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 memory operation 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. Memory transfer instructions require the generation of an address to be used during the transfer. In the implementation of the ARM, this address is generated using the ALU. Consequently, the actual memory transfer has to take place one cycle after the address is generated.

Both of these constraints force us to take several steps to execute ARM instructions involving memory, once the given instruction has been fetched and decoded. Consequently, the remaining instructions in the

pipeline must be "stalled"; that is, delayed from advancing through the pipeline until the multiple-step memory instruction has finished executing.

To accomplish this, we introduce a new universe of *modes* and a new nullary function *ExecuteMode: modes*. *ExecuteMode* indicates which step of a multi-step execution sequence is currently being performed; usually, *ExecuteMode* has the value *first-step* (including in the initial state). For multi-step instructions, *ExecuteMode* will take on other values as the execution of that instruction proceeds.

The fetch and decode rules clearly will need to be modified so that if an instruction being executed takes more than one step, the fetch and decode rules execute only once while the multi-step instruction is executing. The ARM executes the fetch rule during the first step of a multi-step instruction, while it executes the decode rule during the last step of a multi-step instruction (i.e. just before the instruction being decoded will be executed). We describe the changes this requires to the program for $\mathcal{E}_2$ in greater detail.

Fetch. We redefine the abbreviation *FetchOK* to be equivalent to the expression "*ExecuteMode = first-step*"; this will ensure that the fetch rule only executes once for each instruction being executed.

Since the decode rule will not execute until the last step of a multi-step instruction, the fetch rule cannot simply put the instruction retrieved from memory into *DecodeInstr*, since that could possibly displace the instruction waiting to be decoded. Consequently, we introduce two new nullary functions, *FetchedInstr, PrevFetchedInstr: instructions*, which will store the instruction most-recently and next-most-recently fetched from memory, respectively. In the block diagram Figure 1, this is embodied by the two registers IR1 and IR2. In the initial state, *FetchedInstr* and *PrevFetchedInstr* contain nop instructions.

The new fetch rule is shown in Figure 12.

Rule: Fetch
if *FetchOK* **then**
    FetchedInstr := FetchInstr
    PrevFetchedInstr := FetchedInstr
endif

Figure 12: New fetch rules.

Decode. As stated above, the ARM delays the decoding of an instruction until the execute rules are performing the last step of the required instruction sequence. Consequently, we introduce a new binary function *LastStep: instructions* $\times$ *modes* $\rightarrow$ *Bool*, which indicates for a given instruction i and mode m, whether mode m is the last step in executing instruction i. For example, for any single-step instruction i, *LastStep(i,first-step)* is true.

We then redefine the abbreviation *DecodeOK* to be equivalent to:

not *Satisfied(Status,CondCode(ExecuteInstr))*
    or *LastStep(ExecuteInstr,ExecuteMode)*

That is, the instruction currently in the execute stage is in its last step if the instruction's condition is not satisfied (and thus the instruction will not be performed at all), or if *LastStep* explicitly indicates that it is in its last step.

We remove the function *DecodeInstr* and instead make *DecodeInstr* an abbreviation for:

IfThenElse(ExecuteMode=first-step, FetchedInstr,PrevFetchedInstr)

That is, if the decode rule is executing during the first step of the execute cycle (as it did in $\mathcal{E}_2$), the instruction to be decoded is simply the instruction which was most-recently fetched from memory, i.e.

FetchedInstr. If not, the fetch rule has executed in the meantime, and the instruction to be performed resides in *PrevFetchInstr* instead.

We redefine the expression *Contents'(x)* to be equivalent to:

IfThenElse(x≠PC,Contents(x), IfThenElse(ExecuteMode=first-step,
Contents(PC)+4,Contents(PC)))

As with the definition of *DecodeInstr*, this reflects the fact that the execute rule which increments PC by 4 always fires in the first step of any multi-step instruction. Thus, if the contents of PC are to be accessed after that first step during the decode stage, the access should take this into account.

We can now comment on the odd requirement in the ARM that the value of the PC register for any instruction is 8 greater than the address of that instruction. Notice that by the time an instruction which uses PC as an operand is permitted to decode the instruction, PC may have been incremented by 4 once or twice. If PC has been incremented twice by 4, the ARM can access the appropriate value by reading PC directly. If PC has been incremented once by 4, the ARM will need to take the value of PC and increment it again by 4; this is accomplished by the same combinational logic used to increment PC by 4 during each execute stage.

Execute. We redefine the abbreviation *ExecuteOK* to be equivalent to the expression "*ExecuteMode = first-step*", reflecting the idea that all instructions begin executing in the first step of each stage. Additionally, we add the update "*ExecuteMode := first-step*" to the *ExecuteALU* and *ExecuteBranch* rules (which will take only one step to perform in $\mathcal{E}_3$).

The new rules for single-load instructions are shown in Figure 13. The execute stage of a single-load instruction takes three steps. The first step calculates the address to be loaded and places that address in a special location *AddrReg: words*. The second step loads the specified value from memory into an intermediate location *DataIn: words*. The third step places that value into the desired register. Additionally, if write-back to the base register is required, that write occurs during the second step.

```
Rule: SingleLoad
if SingleLoadInstr(ExecuteInstr) then
    if ExecuteMode=first-step and Satisfies(Status,CondCode(ExecuteInstr)) then
        AddrReg := MemAddr
        ExecuteMode := load-read-memory
        Bop := Offset
    elseif ExecuteMode = load-read-memory then
        if ByteTransferInstr(ExecuteInstr) then
            DataIn := PadWord(Memory(AddrReg))
        else DataIn := MemoryWord(AddrReg)
        endif
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        ExecuteMode := load-write-register
    elseif ExecuteMode = load-write-register
        Contents(DestReg) := DataIn
        ExecuteMode := first-step
    endif
endif
```

Figure 13: Single-load rules.

The rules for single-store instructions, shown in Figure 14, follow a similar pattern. The execute stage of a single-store instruction takes two steps to execute. The first step calculates the address to which the datum should be stored, as well as places the value to be stored in a nullary function *DataOut*. The second step performs the transfer to memory, as well as any required write-back operation.

```
Rule: SingleStore
if SingleStoreInstr(ExecuteInstr) then
    if ExecuteMode=first-step and Satisfies(Status,CondCode(ExecuteInstr))then
        AddrReg := MemAddr
        DataOut := Contents'(DestReg)
        ExecuteMode := store
        Bop := Offset
    elseif ExecuteMode = store then
        if ByteTransferInstr(ExecuteInstr) then
            Memory(AddrReg) := DataOut
        else AssignWord(AddrReg, DataOut)
        endif
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        ExecuteMode := first-step
    endif
endif
```

Figure 14: Single-store rules.

Rules for multiple-load instructions, shown in Figure 15, are similar to those for single-load instructions. In a given step, the ARM calculates the address of the memory location to be used to read from memory in the following step, storing the address in the special location *AddrReg*. Functions *FirstTransferReg: instructions* $\rightarrow$ *registers* and *NextTransferReg: instructions* $\times$ *registers* $\rightarrow$ *registers* give the list of registers to be loaded by this instruction; *LastTransferReg: instructions* $\rightarrow$ *registers* gives the last register in this list.

The reader may notice that if write-back is specified for a multiple-load instruction, and the register to be written back is also in the list of registers to be loaded, that register may be written twice during the execution of the instruction. The semantics of the multiple-load instruction specify that the value loaded from memory should be the value stored in the register; it is easy to see that indeed this value is the last value assigned to this register.

The rules for multiple-store instructions are given in Figure 16; they operate in a similar fashion to the rules for multiple-load instructions.

The reader may recall one peculiar requirement for multiple-store instructions. Suppose that the list of registers to be written to memory includes the base register. Then the value written to memory for the base register is the write-back value if and only if the base register is not the first register being written to memory; otherwise the value contained in the base register before the write-back is used. The rules above implement this requirement rather cleanly. Observe that the value being written back to the register file is written during the second step of the execute cycle, while the first value to be written to memory is read during the previous step. Thus, the write back value will only be present in the register file for the second and succeeding writes to memory.

```
Rule: MultipleLoad
if MultipleLoadInstr(ExecuteInstr) then
    if ExecuteMode=first-step and Satisfies(Status,CondCode(ExecuteInstr)) then
        AddrReg := ALU("+",Aop,Bop,0)
        Bop := Offset
        ExecuteMode := multi-load-init
    elseif ExecuteMode = multi-load-init then
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        DataIn := MemoryWord(AddrReg)
        AddrReg := AddrReg + 4
        TransferReg := FirstTransferReg(ExecuteInstr)
        ExecuteMode := multi-load-loop
    elseif ExecuteMode = multi-load-loop then
        Contents(TransferReg) := DataIn
        if TransferReg = LastTransferReg(ExecuteInstr) then
            TransferReg := undef
            ExecuteMode := first-step
        else
            DataIn := MemoryWord(AddrReg)
            AddrReg := AddrReg + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
        endif
    endif
endif
```

Figure 15: Multiple-load instructions.

```
Rule: MultipleStore
if MultipleStoreInstr(ExecuteInstr) then
    if ExecuteMode = first-step and Satisfies(Status,CondCode(ExecuteInstr)) then
        AddrReg := ALU("+",Aop,Bop,0)
        TransferReg := FirstTransferReg(ExecuteInstr)
        DataOut := Contents'(FirstTransferReg(ExecuteInstr))
        Bop := Offset
        ExecuteMode := multi-store-init
    elseif ExecuteMode = multi-store-init then
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        AssignWord(AddrReg, DataOut)
        if TransferReg = LastTransferReg(ExecuteInstr) then
            ExecuteMode := first-step
        else
            AddrReg := AddrReg + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
            DataOut := Contents'(NextTransferReg(ExecuteInstr, TransferReg))
            ExecuteMode := multi-store-loop
        endif
    elseif ExecuteMode = multi-store-loop then
        AssignWord(AddrReg, DataOut)
        if TransferReg = LastTransferReg(ExecuteInstr) then
            ExecuteMode := first-step
        else
            AddrReg := AddrReg + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
            DataOut := Contents'(NextTransferReg(ExecuteInstr, TransferReg))
        endif
    endif endif
```

Figure 16: Multiple-store rules.

6.1.1 Proof of correctness

In this section, we prove the correctness of $\mathcal{E}_3$ with respect to the original ASM model $\mathcal{E}_1$. The proof that $\mathcal{E}_1$ and $\mathcal{E}_3$ produce the same sequence of significant updates is almost identical to that given in the last section for $\mathcal{E}_1$ and $\mathcal{E}_2$. We point out here only the places where the proof differs, due (naturally) to the changes introduced in $\mathcal{E}_3$.

First, some definitions. Let $\rho =< \sigma_1, \sigma_2, \ldots >$ be a run of $\mathcal{E}_3$. An *execution cycle* (or simply a *cycle*) C of a run of ρ of $\mathcal{E}_3$ is any maximal subsequence $< \sigma_i, \ldots, \sigma_{j-1}, \sigma_j, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_{\ell-1} >$ such that *ExecuteMode = first-step* is true in $\sigma_i, \sigma_j, \sigma_k, \sigma_\ell$ but in no other states in that interval. We refer to the three subsequences $\sigma_i, \ldots, \sigma_{j-1}$, $\sigma_j, \ldots, \sigma_{k-1}$, and $\sigma_k, \ldots, \sigma_{\ell-1}$, respectively, as the *fetch*, *decode*, and *execute* stages of C. Notice that every stage may have as little as one state.

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; C is a *meaningful cycle* if i is not a nop instruction. It is easy to check that the instruction i performed during a cycle C is well-defined.

The *significant updates* of a meaningful cycle C which performs instruction i are all the updates to functions in Υ_V performed in the execute stage of C, except for any update to *Contents(PC)*. In the case this set is inconsistent (that is, contains multiple updates to the same location), only the last update applied to a particular location is included in the set. Each run $\rho =< \sigma_1, \sigma_2, \ldots >$ gives rise to a unique sequence of cycles $< C_1, C_2, \ldots >$.

As before, we assume that the program being executed by both $\mathcal{E}_1$ and $\mathcal{E}_3$ is data-dependency and branch-conflict free.

In this section, let s_1 and s_1' be initial states of $\mathcal{E}_1$, and $\mathcal{E}_3$, respectively, such that $s_1|\Upsilon_V = s_1'|\Upsilon_V$. Let $\rho =< s_1, s_2, \ldots >$ be a run of $\mathcal{E}_1$ with corresponding sequence of execution cycles $< C_1, C_2, \ldots >$. Let $\rho' =< s_1', s_2', \ldots >$ be a run of $\mathcal{E}_3$ with corresponding sequence of execution cycles $< C_1', C_2', \ldots >$.

For the remainder of this section, we will speak of the stages of $\mathcal{E}_1$ as if they were composed of multiple states (as in $\mathcal{E}_3$), even though a stage of a cycle of $\mathcal{E}_1$ only has one state. Thus, the first state, or the last state, of a stage of $\mathcal{E}_1$ is the same as the unique state of that stage.

We say that execution cycles C and C' of $\mathcal{E}_1$ and $\mathcal{E}_3$, respectively, *correspond* if:

- C and C' agree on the value of *FetchInstr* in the first state of their fetch stages.
- C and C' agree on the values of all input locations (with respect to *Instr* and *DecodeInstr*) in the first state of their decode stages.
- C and C' agree on the values of *Memory*, *Status*, and *Contents* (with the exception of *Contents(PC)*) in the first state of their execute stages.

The proof of equivalence for $\mathcal{E}_1$ and $\mathcal{E}_3$ is identical to that for $\mathcal{E}_1$ and $\mathcal{E}_2$, except for the following lemma:

Lemma 6 *(Update Lemma) Suppose execution cycles C and C' correspond. Then the significant updates of C and C' are identical.*

Proof. We consider the three corresponding stages of C and C'.

- **Fetch.** In the fetch stage of C, a new instruction, *MemoryWord(Contents(PC))*, is loaded into *DecodeInstr*. Since C and C' correspond, *Contents(PC)* has the same value in the first state of the fetch stage of C', in which *MemoryWord(Contents(PC))* is loaded into *FetchedInstr*. It is easy to show that this value is the instruction being performed by C'; thus, C and C' agree over all stages with respect to the current instruction being performed.
- **Decode.** Since the instruction i performed by C and C' is identical, and C and C' correspond, the first state of the decode stages of C and C' agree with respect to all values which are used to perform updates to *Aop*, *Bop*, *DestReg*, and *ShiftCarryOp*: namely, the instruction i being performed, the input

locations for i, and various static functions. We claim that C and C' make the same updates to *Aop*, *Bop*, *DestReg*, and *ShiftCarryOp*.

If the decode stage of C' has only one state, the claim is obvious. If the decode stage of C' has more than one state, $\mathcal{E}_3$ does not perform the updates to these functions until the last state of the decode stage. The only way the claim could be violated is if one of the input locations for i were modified between the beginning and end of the decode stage. It might be the case that the location in question is *Contents(PC)*, which is usually incremented by 4 in the first state of every stage; the new definition for *Contents'* accounts for this discrepancy. Otherwise, this would mean that the instruction whose execute stage coincides with this decode stage modifies one of i's input locations; this would violate our assumption that the program being executed is data-dependency free. So the claim holds.

- **Execute.** Since C and C' correspond, the first state of the execute stages of C and C' agree with respect to *Memory*, *Status*, and the intermediate locations *Aop*, *Bop*, *DestReg*, and *ShiftCarryOp*. We claim that C and C' produce the same significant updates. This is obvious if the execute stage of C' has only one state; in this case, the rules performed in the execute stage of C and C' are identical. There are several other cases.

  - **Case 1: Single-Load.** In one step, C assigns *Memory(MemAddr)* or *MemoryWord(MemAddr)* to *Contents(DestReg)*. C' takes three steps to perform this operation: copying *MemAddr* to *AddrReg*, copying *Memory(AddrReg)* or *MemoryWord(AddrReg)* to *DataIn*, and copying *DataIn* to *Contents(DestReg)*. Since *DestReg* does not change during the execute stage of C', these updates yield the same effect.

    C may also update *Contents(BaseOp(ExecuteInstr))*; in this case, an identical update occurs in the second step of the execute stage of C'.

  - **Case 2: Single-Store.** In one step, C executes either *Memory(MemAddr) := Contents'(DestReg)* or *AssignWord(MemAddr, Contents'(DestReg))*. C' takes two steps to perform this operation: copying *Contents'(DestReg)* to *DataOut* and *MemAddr* to *AddrReg*, then performing either *Memory(Contents(AddrReg)) := DataOut* or *AssignWord(Contents(AddrReg), DataOut)*. Clearly these updates have the same effect.

    C may also update *Contents(BaseOp(ExecuteInstr))*; in this case, an identical update occurs in the second step of the execute stage of C'.

  - **Case 3: Multiple-Load.** In one step, C performs a number of updates to various registers through the *Contents* function, based upon several consecutive locations in memory. Additionally, if *WriteBack(i)* holds, *Contents(BaseOp(i))* is also modified.

    It is easy to see that C' performs the same updates as C in an iterative fashion, proceeding sequentially through the consecutive locations in memory and the list of registers to be written. If required, the same update to *Contents(BaseOp(i))* is also performed. Notice that *Contents(BaseOp(i))* might be updated twice; once if *WriteBack(i)* holds and once if *BaseOp(i)* is in the list of registers to be loaded. C requires that only the latter update be performed; in C', this update is performed after the former update, so the effect is the same.

  - **Case 4: Multiple-Store.** Again, C' performs the same updates to memory as C in an iterative fashion. There are two subtleties. Notice that if *PC* appears in the list of registers to be stored, it will have been incremented by 4 by the time it is ready to be stored in memory; the new definition of *Contents'* handles this discrepancy. Notice also that if *WriteBack(i)* holds and *BaseOp(i)* appears in the list of registers to be copied to memory, the value copied to memory will be the "write-back" value if *BaseOp(i)* is not the first in the list of registers; this is consistent with the definition of *WriteVal* in $\mathcal{E}_2$.

All cases have been considered and the proof is complete. □

6.2 $\mathcal{E}_4$: Branches

In revision $\mathcal{E}_4$, 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 rules for branch instructions are shown in Figure 17. Notice that there are two new modes used: *refill1* and *refill2*, which reflect steps in the computation where the pipeline will be refilled.

```
Rule: ExecuteBranch
if ExecuteMode = first-step and BranchInstr(ExecuteInstr) then
    if Satisfies(Status,CondCode(ExecuteInstr)) then
        Contents(PC) := ALU("+", Aop, Bop, 0)
        if BranchWithLinkInstr(ExecuteInstr) then
            Aop := Contents(PC)
            Bop := 4
        endif
        ExecuteMode := refill1
    endif
endif
if ExecuteMode = refill1 then
    ExecuteMode := refill2
    Contents(PC) := Contents(PC) + 4
    if BranchWithLinkInstr(ExecuteInstr) then
        Contents(LinkReg) := ALU("-",Aop,Bop,0)
    endif endif
if ExecuteMode = refill2 then
    ExecuteMode := first-step
    Contents(PC) := Contents(PC) + 4
endif
```

Figure 17: Revised branch instruction rules.

Notice that instructions which explicitly write to a register (such as load instructions) may write to the PC register; as with branch statements, we use this refilling technique. Consequently, we replace every update of the form "*ExecuteMode := first-step*" appearing in every rule (except for *ExecutePC*) to the following:

```
if WritesPC(ExecuteInstr) then ExecuteMode := refill1
else ExecuteMode := first-step
endif
```

In this manner, every instruction which explicitely writes to the PC register will proceed through modes *refill1* and *refill2* before beginning the next execution cycle.

We now need to ensure that the fetch and execute rules in fact will perform as required during modes *refill1* and *refill2* in order to refill the instruction pipeline. We redefine the abbreviation *FetchOK* to:

ExecuteMode = first-step **or** *ExecuteMode = refill1* **or** *ExecuteMode = refill2*

Thus, the fetch rule will load instructions from memory during the refill modes as well as at the beginning of each phase. The function *LastExecutionStep* will ensure that the decode rules are executed during mode *refill2*.

We also redefine the abbreviation *DecodeInstr* to:

IfThenElse(ExecuteMode=first-step **or** *ExecuteMode=refill2, FetchedInstr,PrevFetchedInstr)*

This ensures that *DecodeInstr* will retrieve the correct instruction during the refill modes.

6.2.1 Proof of correctness

In this section we prove the equivalence of $\mathcal{E}_1$ and $\mathcal{E}_4$, by describing the changes necessary to the proof of equivalence of $\mathcal{E}_1$ and $\mathcal{E}_3$. Notice that the proof of equivalence of $\mathcal{E}_1$ and $\mathcal{E}_3$ assumes that the program being executed by the ARM is branch-conflict free; we make no such assumptions in this proof of equivalence.

In the initial state of $\mathcal{E}_4$, *ExecuteMode = refill1*. The definitions of execution cycle, phase, instructions performed during a phase, etc. are the same in $\mathcal{E}_4$ as in $\mathcal{E}_3$.

In this section, let s_1 and s_1' be initial states of $\mathcal{E}_1$, and $\mathcal{E}_4$, respectively, such that $s_1|\Upsilon_V = s_1'|\Upsilon_V$. Let $\rho =< s_1, s_2, \ldots >$ be a run of $\mathcal{E}_1$ with corresponding sequence of execution cycles $< C_1, C_2, \ldots >$. Let $\rho' =< s_1', s_2', \ldots >$ be a run of $\mathcal{E}_4$ with corresponding sequence of execution cycles $< C_1', C_2', \ldots >$.

The proof of equivalence of $\mathcal{E}_1$ and $\mathcal{E}_4$ is identical to that of $\mathcal{E}_1$ and $\mathcal{E}_3$, except for the following lemmas:

Lemma 7 *(Consecutive Instruction Lemma) Fix a state s of* $\mathcal{E}_4$*. Let* i_e*,* i_d*, and* i_f *be the values of* ExecuteInstr, DecodeInstr, *and* FetchInstr, *respectively. Then:*

- *If ExecuteMode = refill2, then* i_f *= MemoryWord(a) and* i_d *= MemoryWord(a − 4), where a = Contents(PC).*
- *If ExecuteMode* $\neq$ *refill2 and ExecuteMode* $\neq$ *refill1, then* i_f *= MemoryWord(a),* i_d *= MemoryWord(a− 4), and* i_e *= MemoryWord(a − 8), where a = Contents(PC) if ExecuteMode = first-step and a = Contents(PC) - 4 otherwise.*

Proof. By induction over states. The condition is trivially true in the initial state, since *ExecuteMode = refill1*.

In most states, *Contents(PC)* is incremented by 4, which can be seen to maintain the invariant. The exception occurs when *Contents(PC)* is updated to an arbitrary value; in this case, *ExecuteMode* is always updated to *refill1*. □

Lemma 8 *(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* Contents(PC).

Proof. By supposition, since C_j and C_j' agree in the first state of their fetch stages with respect to *Contents(PC)*, C_j and C_j' perform the same instruction i.

- *WritesPC(i)* is true, and *Satisfies(Status,CondCode(i))* holds in the first state of the execute stage of C_j (and thus in C_j'). Thus, *Contents(PC)* will be modified in the execute stage of C_j and C_j'.

  In $\mathcal{E}_1$, the next meaningful instruction cycle will begin with the state following the execute stage of C_j; consequently, the address of the instruction executed by that stage is determined by the update to *Contents(PC)* performed in the execute stage of C_j.

  In $\mathcal{E}_4$, the execute stage will have several states in which updates are made to *Contents* and *Memory* (including the update to *Contents(PC)*, followed by states r_1 and r_2, in which *ExecuteMode* has the value *refill1* and *refill2*, respectively. The instruction j which appears in *ExecuteInstr* in the first state

following r_2 will be loaded into *FetchedInstr* in r_1 and copied into *DecodeInstr* in r_2; the memory address used to find j is thus determined by the update to *Contents(PC)* performed in the execute stage of C'_j. Thus, r_1 and r_2 are the fetch and decode stages of C'_{j+1}.

An argument similar to that given in the Update Lemma shows that C_j and C'_j perform the same update to *Contents(PC)*; thus, the fetch stages of C_{j+1} and C'_{j+1} agree with respect to *Contents(PC)*, as desired.

- *WritesPC(i)* is not true, or *Satisfies(Status,CondCode(i))* does not hold in the execute stage of C_j (and C'_j). Then the instruction i' executed in C_{j+1} and C'_{j+1} is the next instruction in the pipeline, which is the instruction following i in memory. Thus $FetchInstr = i'$ in both C_{j+1} and C'_{j+1}.

In each case, C_{j+1} and C'_{j+1} agree with respect to *FetchInstr* in their fetch stages, as desired. □

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 solution is to make the value being written by instruction i_j explicitly available to the decode stage of i_{j+1}. The ARM performs this task by means of a forwarding path where the written value is conveyed immediately to the decode stage. We do this by redefining the abbreviation *Contents'(x)* to:

IfThenElse ($x \neq PC$,
  IfThenElse(WritingReg(ExecuteMode,ExecuteInstr,x),
    WriteData,Contents(x)),
    IfThenElse(ExecuteMode=first-step,Contents(PC)+4,Contents(PC)))

where *WriteData* abbreviates:

IfThenElse(ExecuteOK **and** *AluInstr(ExecuteInstr),*
  ALU(ALUOp(ExecuteInstr), Aop, Bop, Carry(Status)),
  IfThenElse(ExecuteMode=store,Aop+Offset,
  IfThenElse(ExecuteMode=load-write-register **or** *ExecuteMode=multi-load-loop,*
    DataIn, IfThenElse(ExecuteMode=multi-store-init,Aop+Bop,undef)))))

These abbreviations make use of a new function *WritingReg: modes* × *instructions* × *registers* → *Bool*, which indicates whether the specified register is being written by the instruction currently executing in the specified mode. If *WritingReg* returns *true*, the desired value is simply the value that is about to be written to that register; otherwise, the value is as before. Notice that *WriteData* only considers certain values for *ExecuteMode*; these are the only modes in which a change in the register file can occur during the last step of an execution cycle.

The problem with the status register is solved similarly. We replace the expression *Carry(Status)* in the decode rule by *Carry(STATUS)*, where *STATUS* is defined as:

IfThenElse (WritingStatus(Status, ExecuteMode, ExecuteInstr),
  UpdateStatus(Status, ALUop(ExecuteInstr), Aop, Bop, ShiftCarryOp), Status)

The function *WritingStatus* indicates whether the instruction in the execute stage is writing to the status register. If this is the case and the status is required during the decode stage, the required value being written is read directly.

The proof of correctness for $\mathcal{E}_1$ and $\mathcal{E}_5$ is the same as that for $\mathcal{E}_1$ and $\mathcal{E}_4$, with one slight change to the Update Lemma. Previously, our assumption that the program was data-dependency free was used only in one place: to assure us that values accessed from memory and registers during the decode stage of C_j reflect the values stored in stages C_1 through C_{j-1}. The only possible problem arises when the decode stage of C_j coincides with the execute stage of C_{j-1}, where the values being modified in C_{j-1} are used by the instruction executing in C_j. But in this case, the new definition of *Contents'* given above makes those changed values available to C_j, so no problem arises.

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.)

An examination of $\mathcal{E}_5$ will show that in almost every case, $\mathcal{E}_5$ reads from at most two registers and writes at most one register (other than *PC*) at each step. The one exception is for certain types of ALU instructions: namely, those instructions which require a shift where the shift amount is specified by a register. Such an instruction requires three registers; one for the value of *Aop*, one for the pre-shifted value of *Bop*, and one for the amount of the shift. We indicate such instructions by means of a static function *AluRegShift: instructions* $\rightarrow$ *Bool*.

The solution is to perform such an instruction i in two steps, thereby introducing a stall cycle. During i's decode stage, the ARM uses its two read ports to calculate *Bop*, while ignoring *Aop*. This causes a change to the decode rule, shown below in Figure 18.

```
Rule: Decode
if DecodeOK then
    ExecuteInstr := DecodeInstr
    if not Nop(DecodeInstr) then
        DestReg := DestOp(DecodeInstr)
        if not AluRegShift(DecodeInstr) then
            Aop := Contents'(AopReg(DecodeInstr))
        endif
        Bop := Shift(SourceVal,ShiftType(DecodeInstr),ShiftAmt,Carry(STATUS))
        ShiftCarryOp :=
            ShiftCarry(SourceVal,ShiftType(DecodeInstr),ShiftAmt,Carry(STATUS))
    endif
endif
```

Figure 18: Revised decode rule.

During the first step of i's execute stage, we load the proper value into *Aop*; the required ALU operation is then performed during the second step of i's execute stage. To distinguish this case from ALU instructions which do not require this extra step, we re-define the abbreviation *ExecuteOK* to:

$$\mathit{ExecuteMode} = \mathit{first\text{-}step} \textbf{ and not } \mathit{AluRegShift(ExecuteInstr)}$$

The new rules for ALU-register-shift operations are shown below in Figure 19.

```
if ExecuteMode = first-step and AluRegShift(ExecuteInstr) then
    if Satisfies(Status,CondCode(ExecuteInstr)) then
        Aop := Contents'(AopReg(ExecuteInstr))
        ExecuteMode := alu-shift
    endif
endif

if ExecuteMode = alu-shift then
    if WriteResult(ExecuteInstr) then
        Contents(DestReg) := ALU(Op(ExecuteInstr), Aop, Bop, Carry(Status))
    endif
    if SetCondCode(ExecuteInstr) then
        Status := UpdateStatus(Status,Op(ExecuteInstr),Aop,Bop,ShiftCarryOp)
    endif
    if WritesPC(ExecuteInstr) then ExecuteMode := refill1
    else ExecuteMode := first-step
    endif
endif
```

Figure 19: ALU-register-shift rules.

The proof of equivalence for $\mathcal{E}_1$ and $\mathcal{E}_6$ is similar to that for $\mathcal{E}_1$ and $\mathcal{E}_5$; a small change to the Update Lemma is needed to confirm that the new rules above perform the same updates to the register file.

The complete set of rules of $\mathcal{E}_6$ are given in Appendix 7. The universes that appear in $\mathcal{E}_6$ are listed in Table 1.

7 Discussion

Automated formal verification for microprocessors has become a popular area of research. The spectrum of methods [14] 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 abstracted system model is used, which 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 [13] 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 [12]. 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 [2] applied the ASM methodology for the first time to microprocessor verification. They proved the correctness of a pipelined version of the DLX processor [8] 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) [7]. 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 [15], PVS [18], and SMV [17]. 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 [9]; 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.

Acknowledgements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A: Final model $\mathcal{E}_6$

Universe	Description
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
modes	{*first-step*, *alu-shift*, *refill1*, *refill2*, *store*, *load-read-memory*, *load-write-register*, *multi-load-init*, *multi-load-loop*, *multi-store-init*, *multi-store-loop*}

Table 1: Universes in $\mathcal{E}_6$

Rule: Fetch
if *FetchOK* **then**
    FetchedInstr := FetchInstr
    PrevFetchedInstr := FetchedInstr
endif
where
    FetchOK abbreviates ExecuteMode = first-step **or**
        ExecuteMode = refill1 **or** *ExecuteMode = refill2*
    FetchInstr abbreviates MemoryWord(Contents(PC))
    MemoryWord(x) abbreviates
        Word(Memory(x),Memory(x+1),Memory(x+2),Memory(x+3))

```
Rule: Decode
if DecodeOK then
    ExecuteInstr := DecodeInstr
    if not Nop(DecodeInstr) then
        DestReg := DestOp(DecodeInstr)
        if not AluRegShift(DecodeInstr) then
            Aop := Contents'(AopReg(DecodeInstr))
        endif
        Bop := Shift(SourceVal,ShiftType(DecodeInstr),
            ShiftAmt,Carry(STATUS))
        ShiftCarryOp := ShiftCarry(SourceVal,
            ShiftType(DecodeInstr),ShiftAmt,Carry(STATUS))
    endif
endif
where
    DecodeOK abbreviates
        not Satisfied(Status,CondCode(ExecuteInstr)) or
            LastStep(ExecuteInstr,ExecuteMode)
    DecodeInstr abbreviates
        IfThenElse(ExecuteMode=first-step or
            ExecuteMode=refill2,FetchedInstr,PrevFetchedInstr)
    Contents'(x) abbreviates
        IfThenElse (x ≠ PC,
            IfThenElse(WritingReg(ExecuteMode,ExecuteInstr,x),
                WriteData,Contents(x)),
            IfThenElse(ExecuteMode=first-step,Contents(PC)+4,
                Contents(PC)))
    WriteData abbreviates
        IfThenElse(ExecuteOK and AluInstr(ExecuteInstr),
                ALU(Op(ExecuteInstr), Aop, Bop, Carry(Status)),
            IfThenElse(ExecuteOk and BranchInstr(ExecuteInstr),
                Contents(PC)-4,
            IfThenElse(ExecuteMode=store,Aop+Offset,
            IfThenElse(ExecuteMode=load-write-register or
                ExecuteMode=multi-load-loop,DataIn,
            IfThenElse(ExecuteMode=multi-store-init,Aop+Bop,undef)))))
    SourceVal abbreviates
        IfThenElse(ImmBop(DecodeInstr),
            ImmediateVal(DecodeInstr),Contents'(BopReg(DecodeInstr)))
    ShiftAmt abbreviates
        IfThenElse(ImmShift(DecodeInstr),
            ImmShiftAmt(DecodeInstr),Contents'(ShiftReg(DecodeInstr)))
    STATUS abbreviates
        IfThenElse (WritingStatus(Status, ExecuteMode, ExecuteInstr),
            UpdateStatus(Status, ALUop(ExecuteInstr),
                Aop, Bop, ShiftCarryOp),Status)
```

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

where ExecuteOK abbreviates ExecuteMode = first-step
```

```
Rule: ExecuteALU
if ExecuteMode = first-step and AluInstr(ExecuteInstr) and
    not AluRegShift(ExecuteInstr) then
    if Satisfies(Status,CondCode(ExecuteInstr)) then
        if WriteResult(ExecuteInstr) then
            Contents(DestReg) :=
                ALU(Op(ExecuteInstr), Aop, Bop, Carry(Status))
        endif
        if SetCondCode(ExecuteInstr) then
            Status := UpdateStatus(Status,Op(ExecuteInstr),
                Aop,Bop,ShiftCarryOp)
        endif
    endif
    if WritesPC(ExecuteInstr) then ExecuteMode := refill1
    else ExecuteMode := first-step
    endif
endif
```

```
Rule: ALU-RegisterShift
if ExecuteMode = first-step and AluRegShift(ExecuteInstr) then
    if Satisfies(Status,CondCode(ExecuteInstr)) then
        Aop := Contents'(AopReg(ExecuteInstr))
        ExecuteMode := alu-shift
    endif
endif

if ExecuteMode = alu-shift then
    if WriteResult(ExecuteInstr) then
        Contents(DestReg) :=
            ALU(Op(ExecuteInstr), Aop, Bop, Carry(Status))
    endif
    if SetCondCode(ExecuteInstr) then
        Status :=
            UpdateStatus(Status,Op(ExecuteInstr),Aop,Bop,ShiftCarryOp)
    endif
    if WritesPC(ExecuteInstr) then ExecuteMode := refill1
    else ExecuteMode := first-step
    endif
endif
```

```
Rule: ExecuteBranch
if ExecuteMode = first-step and BranchInstr(ExecuteInstr) then
    if Satisfies(Status,CondCode(ExecuteInstr)) then
        Contents(PC) := ALU("+", Aop, Bop, 0)
        if BranchWithLinkInstr(ExecuteInstr) then
            Aop := Contents(PC)
            Bop := 4
        endif
        ExecuteMode := refill1
    endif
endif
if ExecuteMode = refill1 then
    ExecuteMode := refill1
    Contents(PC) := Contents(PC) + 4
    if BranchWithLinkInstr(ExecuteInstr) then
        Contents(LinkReg) := ALU("-",Aop,Bop,0)
    endif endif
if ExecuteMode = refill2 then
    ExecuteMode := first-step
    Contents(PC) := Contents(PC) + 4
endif
```

Rule: SingleLoad
if *SingleLoadInstr(ExecuteInstr)* **then**
  if *ExecuteMode=first-step* **and** *Satisfies(Status,CondCode(ExecuteInstr))* **then**
    AddrReg := MemAddr
    ExecuteMode := load-read-memory
    Bop := Offset
  elseif *ExecuteMode = load-read-memory* **then**
    if *ByteTransferInstr(ExecuteInstr)* **then**
      DataIn := PadWord(Memory(AddrReg))
    else *DataIn := MemoryWord(AddrReg)*
    endif
    if *WriteBack(ExecuteInstr)* **then**
      Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
    endif
    ExecuteMode := load-write-register
  elseif *ExecuteMode = load-write-register*
    Contents(DestReg) := DataIn
    if *WritesPC(ExecuteInstr)* **then** *ExecuteMode := refill1*
    else *ExecuteMode := first-step*
    endif
  endif
endif

MemAddr abbreviates IfThenElse(PreIndexed(Instr),Aop + Offset,Aop)
Offset abbreviates IfThenElse(IncrOp(Instr),Bop,−Bop)

```
Rule: SingleStore
if SingleStoreInstr(ExecuteInstr) then
    if ExecuteMode=first-step and
        Satisfies(Status,CondCode(ExecuteInstr))then
        AddrReg := MemAddr
        DataOut := Contents'(DestReg)
        Bop := Offset
        ExecuteMode := store
    elseif ExecuteMode = store then
        if ByteTransferInstr(ExecuteInstr) then
            Memory(AddrReg) := DataOut
        else AssignWord(AddrReg, DataOut)
        endif
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        if WritesPC(ExecuteInstr) then ExecuteMode := refill1
        else ExecuteMode := first-step
        endif
    endif
endif
```

Rule: MultipleLoad
if *MultipleLoadInstr(ExecuteInstr)* **then**
  if *ExecuteMode=first-step* **and**
    Satisfies(Status,CondCode(ExecuteInstr)) **then**
    AddrReg := ALU("+",Aop,Bop,0)
    Bop := Offset
    ExecuteMode := multi-load-init
  elseif *ExecuteMode = multi-load-init* **then**
    if *WriteBack(ExecuteInstr)* **then**
      Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
    endif
    DataIn := MemoryWord(AddrReg)
    AddrReg := AddrReg + 4
    TransferReg := FirstTransferReg(ExecuteInstr)
    ExecuteMode := multi-load-loop
  elseif *ExecuteMode = multi-load-loop* **then**
    Contents(TransferReg) := DataIn
    if *TransferReg = LastTransferReg(ExecuteInstr)* **then**
      TransferReg := undef
      if *WritesPC(ExecuteInstr)* **then** *ExecuteMode := refill1*
      else *ExecuteMode := first-step*
      endif
    else
      DataIn := MemoryWord(AddrReg)
      AddrReg := AddrReg + 4
      TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
    endif
  endif
endif

```
Rule: MultipleStore
if MultipleStoreInstr(ExecuteInstr) then
    if ExecuteMode = first-step and
        Satisfies(Status,CondCode(ExecuteInstr)) then
        AddrReg := ALU("+",Aop,Bop,0)
        TransferReg := FirstTransferReg(ExecuteInstr)
        DataOut := Contents'(FirstTransferReg(ExecuteInstr))
        Bop := Offset
        ExecuteMode := multi-store-init
    elseif ExecuteMode = multi-store-init then
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        AssignWord(AddrReg, DataOut)
        if TransferReg = LastTransferReg(ExecuteInstr) then
            if WritesPC(ExecuteInstr) then ExecuteMode := refill1
            else ExecuteMode := first-step
            endif
        else
            AddrReg := AddrReg + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
            DataOut :=
                Contents'(NextTransferReg(ExecuteInstr, TransferReg))
            ExecuteMode := multi-store-loop
        endif
    elseif ExecuteMode = multi-store-loop then
        AssignWord(AddrReg, DataOut)
        if TransferReg = LastTransferReg(ExecuteInstr) then
            if WritesPC(ExecuteInstr) then ExecuteMode := refill1
            else ExecuteMode := first-step
            endif
        else
            AddrReg := AddrReg + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
            DataOut :=
                Contents'(NextTransferReg(ExecuteInstr, TransferReg))
        endif
    endif
endif
```