Specification and Verification of Pipelining in the ARM2 RISC Microprocessor[*]

James K. Huggins[†]and David Van Campenhout[‡]
EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA

December 19, 1996

Abstract

We specify the ARM2 RISC microprocessor using the Gurevich Abstract State Machine methodology, and prove the correctness of its pipelining techniques.

The Gurevich Abstract State Machine (ASM) methodology, formerly known as the evolving algebra or ealgebra methodology, first proposed by Yuri Gurevich in [5], 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 [2, 8] for numerous examples.

The ARM2 [1, 4] is one of the early commercial RISC microprocessors. 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.

In this paper, we specify the ARM2 microprocessor and prove the correctness of its pipelining techniques. We begin with a self-contained introduction to sequential ASMs in section 1; the definitions given there are sufficient for understanding of this paper. Section 2 introduces the ARM2 (hereafter ARM) microprocessor in greater detail. We then present a non-pipelined version of the ARM processor in section 3; this version establishes the standard to which we will compare the pipelined versions.

Following [3], we alter the ASM presented in section 3 to operate with a simple, three-stage pipeline, ignoring the possible problems with branch and data dependency which may arise; this simple pipelined processor is presented in section 4 and proved to be equivalent to the non-pipelined version in an appropriate sense.

In section 5, we alter the ASM to reflect the hardware limitations of the external memory system of the ARM2. In section 6, we alter the ASM to correctly handle branch-conflicts. In section 7, we alter the ASM to correctly handle data-dependencies. In section 8, we alter the ASM to reflect the hardware restrictions of the register file. For each of these revised machines, we prove that the revised machine is equivalent to the original ASM described in section 3. The final pipelined version of the ARM2 processor is given in Appendix A.

Acknowledgements. Trevor Mudge originally suggested this project to us.

1 Gurevich Abstract State Machines

The ASM thesis is that any software or hardware system can be modeled at its natural abstraction level by a Gurevich abstract state machine. Based upon this thesis, members of the ASM community have sought to

[*]University of Michigan EECS Department Technical Report CSE-TR-321-96.

[†]huggins@eecs.umich.edu. Partially supported by ONR grant N00014-94-1-1182 and NSF grant CCR-95-04375.

[‡]davidvc@eecs.umich.edu. Partially supported by SRC contract 95-DJ-338 and NSF grant MIP-9404632.

develop a methodology based upon mathematics which would allow such systems to be modeled naturally; that is, described at their natural abstraction levels. See [2, 8] for a number of examples of ASMs applied to various real-world systems.

Sequential ASMs (under their former name, evolving algebras) are described in [6]; a more detailed description of ASMs (including distributed characteristics) is given in [7]. 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.

1.1 States

The states of a 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 (for convenience) 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.

1.2 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.

1.3 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 instruction* 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.

1.4 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.

2 The ARM2 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.

2.1 ARM2 Instruction Set Architecture

In this section we give an overview of the ARM2 instruction set architecture (ISA). The 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 sequential 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 sequential 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 (*i.e.* 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 ARM2 discussed in this paper does not implement 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.

2.2 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 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 phase, the next instruction is being decoded, and the instruction which following the instruction being decoded is being fetched from memory. During the instruction fetch phase, the instruction at the address indicated by the PC is obtained from memory. During the decode phase of an instruction, the processor sets up the operands the instruction requires for its execution. For data processing instructions, typically two registers are read from the registerfile. One of them has a shift applied to its contents. During the execute phase, 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 phases 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 phase will the processor

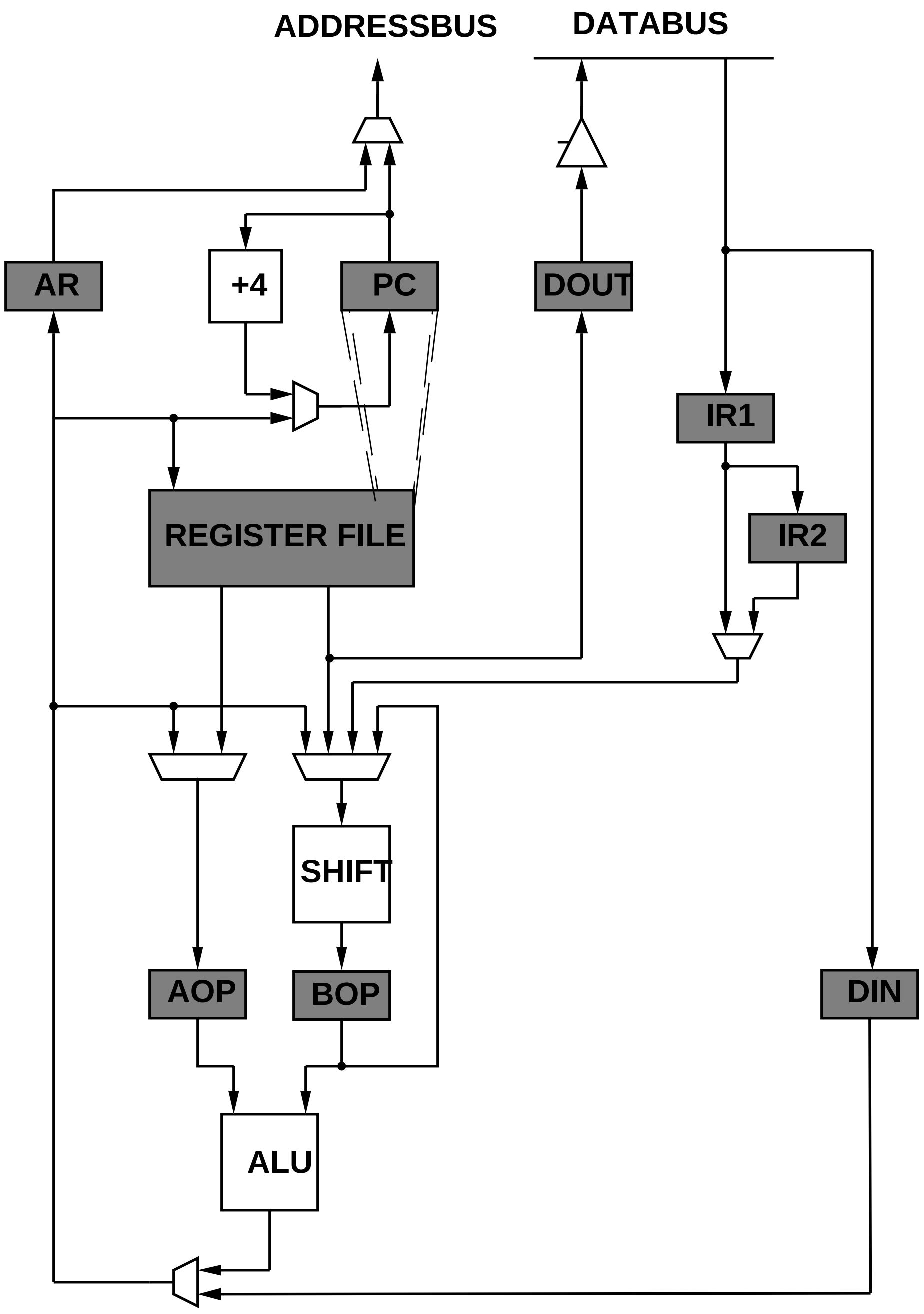

Figure 1: Block diagram of ARM datapath.

fetch a new instruction from memory. Only during the last cycle of an instruction's execute phase will the processor decode the next instruction.

3 $\mathcal{E}_1$: Sequential ARM

Here, we present $\mathcal{E}_1$, our first ASM for the ARM. The ASM can be seen as an interpreter for the ARM instruction set. $\mathcal{E}_1$ processes commands sequentially; that is, $\mathcal{E}_1$ completes its execution of a given ARM instruction before beginning to execute the next instruction in sequence.

3.1 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 is 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 operands needed for the instruction are loaded from the register file), and the execute stage (in which the instruction is actually performed). A universe of *stages* contains elements *fetch*, *decode*, and *execute* to represent these stages. 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:

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

3.2 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 phase, 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.

```
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.

3.3 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).
- 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 phase 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 phase (*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. 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 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 transition rule for the decode phase 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 architecture which will be made apparent in later sections.

3.4 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.

3.4.1 Incrementing PC

The execute phase 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 phase, 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.

```
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.

3.4.2 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: ExecuteNop
if ExecuteOK and Nop(Instr) then Stage := fetch endif
```

Figure 5: Nop rule.

3.4.3 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: 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: instructions* $\rightarrow$ *Bool* and *SetCondCode: instructions* $\rightarrow$ *Bool* indicate which of these actions are required.

The function *ALU: 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: 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 phase.

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

3.4.4 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*; 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:*

```
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.

instructions $\rightarrow$ *Bool* indicates whether an instruction calls for that linking action; the static nullary function *LinkReg: registers* indicates which register is used for this purpose.

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.

3.4.5 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 word 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 phase 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, which is also the register to which the write-back value should be written. The ARM instruction set architecture specifies that $BaseOp(i) = PC \Rightarrow WriteBack(i) = false$; 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) := Contents'(DestReg)
            else AssignWord(MemAddr,Contents'(DestReg))
            endif
        endif
        if WriteBack(Instr) then Contents(BaseOp(Instr)) := Aop + Offset
        endif
    endif
    Stage := fetch
endif

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.

3.4.6 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

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.

3.5 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 phases of C, respectively, as the *fetch*, *decode*, and *execute phases* 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 phase 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 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.

4 $\mathcal{E}_2$: Simple Pipeline Model

In this section, we revise our previous ASM $\mathcal{E}_1$ to incorporate a simple pipelined model. The result is our revised ASM $\mathcal{E}_2$.

4.1 Constructing $\mathcal{E}_2$ From $\mathcal{E}_1$

The basic idea is as follows. Observe that in $\mathcal{E}_1$, every ARM instruction is executed in three steps; an execution of the Fetch rule, an execution of the Decode rule, and an execution of one of the Execute rules. The execution of one ARM instruction must be completed before execution of the next ARM instruction can occur. Suppose that two consecutive instructions in memory are independent of one another (that is, performing the first instruction has no effect on if or how the second instruction is performed). We can improve the speed of the system by allowing the second instruction to begin executing while the first instruction is completing its execution.

Since the ARM uses a three-step execution cycle, we can implement a three-stage pipeline. 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 the sense described above), 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 incorporate an instruction pipeline into $\mathcal{E}_2$, we add new distinguished elements *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 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 phase. 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 phase 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 phase. We thus redefine the abbreviation *Contents'(x)*, used only during the decode phase, 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*).

4.2 Proof of Equivalence

We now proceed to prove that $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, in an appropriate sense. 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 $s_1|\Upsilon_V = s'_1|\Upsilon_V$. 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 phases.
- C and C' agree on the values of all input locations (with respect to *Instr* and *DecodeInstr*) in their decode phases.
- C and C' agree on the values of *Memory*, *Status*, and *Contents* (with the exception of *Contents(PC)*) in their execute phases.

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)*; that is,* 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($a-4$).

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. QED.

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, DecodeInstr, *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. QED.

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 phases 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 phase (which we have already shown to be identical). Thus, the significant updates performed are identical. QED.

Corollary 1 *(Update Corollary) Let C and C' be corresponding cycles. Let s and s' be the states immediately following the execute phases 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 phases agree with respect to *Memory*, *Status*, and *Contents* (except for *Contents(PC)* and generate the same updates to these functions. QED.

Lemma 4 *(PC Lemma) Suppose C_j and C'_j correspond. Then the fetch phases 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 phases 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 phase of C_j (and by correspondence, in C'_j). Then the value of *Contents(PC)* in the fetch phase of C_{j+1} is determined by the updates executed in the execute phase 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 phase of C'_j (when a non-nop instruction will appear in the pipeline). Thus, the value of *Contents(PC)* in the fetch phase of C'_{j+1} is determined by the updates executed in the execute phase of C'_j.

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

- *WritesPC(i)* is true, but *Satisfies(Status, CondCode(i))* does not hold in the execute phase of C_j (and by correspondence, in C'_j). Then *Contents(PC)* will be incremented by 4 in the execute phase 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 phase of C_{j+1} is 12 greater than its value in the fetch phase 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 phase of C'_{j+1} is determined by the updates executed in the execute phase of C'_j. Since *Satisfies(Status, CondCode(i'))* does not hold in that phase, *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 phase of C_j). Thus, the value of *Contents(PC)* in the fetch phase of C_{j+1} is 12 greater than its value in the fetch phase of C'_{j+1}.

- *WritesPC(i)* is false. Then *Contents(PC)* is incremented by 4 in the execute phase 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 phase of C_j; thus, the value of *Contents(PC)* in the fetch phase of C_{j+1} is determined by the update to *Contents(PC)* in the fetch phase of C_j; as discussed above, *Contents(PC)* is incremented by 4 in that state. QED.

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 phases of C_{j+1} and C'_{j+1}.

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

- **Decode.** Let s and s' be the decode phases of C_{j+1} and C'_{j+1}, respectively. We must show that all input locations of s and s' agree. Since the fetch phases 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 phase of C'_{j+1}, may also be the execute phase of C'_j (if *ExecuteInstr* is not a nop instruction) or occur strictly after the execute phase 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 phase of C'_j occurs strictly before s'. By the Update Corollary, the state r which follows the execute phase of C'_j agrees with the fetch phase 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 phases, 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 phase of C'_j and the decode phase 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 phase 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 phases is easy to observe; by the Update Corollary, the states which follow the execute phases 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 phases of C_{j+1} and C'_{j+1}. QED.

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, is identical.*

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

5 $\mathcal{E}_3$: Coping With The Memory System

We now modify our model of the ARM processor to reflect some important hardware constraints pertaining to the memory system.

The (external) memory system requires one cycle to transfer a datum between memory and the register file. At the beginning of a load cycle, the address is given to the memory system; the desired datum is placed on the data bus by the end of the same cycle. Similarly, at the beginning of a store cycle, the address and datum are given to the memory system; the desired datum is written to the specified memory location by the end of the same cycle. This poses the following two constraints:

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

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 phase 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.

The new fetch rule is shown in Figure 10.

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

Figure 10: 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* × *modes* → *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 phase 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 re-define 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 phase, 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 phase. 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 11. 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
        Contents(AddrReg) := MemAddr
        ExecuteMode := load-read-memory
        Bop := Offset
    elseif ExecuteMode = load-read-memory then
        if ByteTransferInstr(ExecuteInstr) then
            DataIn := PadWord(Memory(Contents(AddrReg)))
        else DataIn := MemoryWord(Contents(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 11: Single-load rules.

The rules for single-store instructions, shown in Figure 12, follow a similar pattern. 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.

Rules for multiple-load instructions, shown in Figure 13, 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 register *AddrReg*. Functions *FirstTransferReg: instructions* $\rightarrow$ *registers* and *NextTransferReg: instructions* $\times$ *registers* $\rightarrow$ *registers* give the list of registers to be loaded by this instruction.

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

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

Figure 12: Single-store rules.

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 14; 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.

5.1 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* phases of C. Notice that every phase 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

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

Figure 13: Multiple-load instructions.

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

Figure 14: Multiple-store rules.

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

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 phases.
- C and C' agree on the values of all input locations (with respect to *Instr* and *DecodeInstr*) in the first state of their decode phases.
- 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 phases.

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 phases of C and C'.

- **Fetch.** In the fetch phase 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 phase 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 phases 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 phase of C' has only one state, the claim is obvious. If the decode phase 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 phase. 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 phase. 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 phase; the new definition for *Contents'* accounts for this discrepancy. Otherwise, this would mean that the instruction whose execute phase coincides with this decode phase 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 phases 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 phase of C' has only one state; in this case, the rules performed in the execute phase 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 *Contents(AddrReg)*, copying *Memory(Contents(AddrReg))* or *MemoryWord(Contents(AddrReg))* to *DataIn*, and copying *DataIn* to *Contents(DestReg)*. Since *DestReg* does not change during the execute phase 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 phase 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 *Contents(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 phase 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 subleties. 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. QED.

6 $\mathcal{E}_4$: Branches

We now modify our model of the ARM processor to correctly handle branch conflicts. Recall that an ARM program is branch-conflict free if every instruction which explicitly modifies the *PC* register is followed by two nop instructions. Up until this point we have assumed that every program was branch-conflict free; we now remove this assumption.

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 15. Notice that there are two new modes used: *refill1* and *refill2*, which reflect steps in the computation where the pipeline will be refilled.

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:

```
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
```

Figure 15: Revised branch instruction rules.

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

In this manner, every instruction which 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.

6.1 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.* QED.

Lemma 8 *(PC Lemma) Suppose C_j and C'_j correspond. Then the fetch phases 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 phases 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 phase of C_j (and thus in C'_j). Thus, *Contents(PC)* will be modified in the execute phase of C_j and C'_j.

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

  In $\mathcal{E}_4$, the execute phase 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 phase of C'_j. Thus, r_1 and r_2 are the fetch and decode phases 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 phases 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 phase 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 phases, as desired.

7 $\mathcal{E}_5$: Data dependencies

In $\mathcal{E}_5$, we now remove the assumption that the ARM programs being executed have no data-dependencies.

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 phase) at the same time as i_{j+1} reads from register r (in its decode phase). A similar problem results with the status flags; i_j may update the status register during its execute phase while i_{j+1} is reading the status register to perform a shift during its decode phase.

The solution is to make the value being written by instruction i_j explicitly available to the decode phase 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 phase. We do this by redefining the abbreviation *Contents'(x)* to:

> *IfThenElse (x ≠ 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(Op(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 phase of C_j reflect the values stored in phases C_1 through C_{j-1}. The only possible problem arises when the decode phase of C_j coincides with the execute phase 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.

8 $\mathcal{E}_6$: Register File Restrictions

In our final ASM $\mathcal{E}_6$, we take into account certain hardware restrictions on the register file. In the implementation of the ARM, the register file has two read ports and one write port. 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 phase, 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 16.

```
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 16: Revised decode rule.

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

ExecuteMode = *first-step* **and not** *AluRegShift(ExecuteInstr)*

The new rules for ALU-register-shift operations are shown below in Figure 17.

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.

References

[1] Acorn RISC machine (ARM) family data manual, Englewood Clifs, N.J. : Prentice Hall, 1990.

[2] E. Börger, "Annotated Bibliography on Evolving Algebras", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 37–51.

[3] E. Börger and S. Mazzanti, "A correctness proof for pipelining in RISC architectures." DIMACS Technical Report 96-22, July 1996.

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

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

[6] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", *Current Trends in Theoretical Computer Science*, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292. (First published in Bull. EATCS 57 (1991), 264–284; an updated version appears in [8].)

[7] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 9–36.

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

if $ExecuteMode = alu\text{-}shift$ 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 $ExecuteMode := first\text{-}step$
    endif
endif
```

Figure 17: ALU-register-shift rules.

[8] J. Huggins, ed., "Gurevich Abstract State Machine Home Page", EECS Department, University of Michigan, `http://www.eecs.umich.edu/gasm/`.

A Appendix: Full ARM Model

```
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
        Contents(AddrReg) := MemAddr
        ExecuteMode := load-read-memory
        Bop := FinalOffset(ExecuteInstr)
    elseif ExecuteMode = load-read-memory then
        if ByteTransferInstr(ExecuteInstr) then
            DataIn := PadWord(Memory(Contents(AddrReg)))
        else DataIn := MemoryWord(Contents(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
```

```
Rule: SingleStore
if SingleStoreInstr(ExecuteInstr) then
    if ExecuteMode=first-step and Satisfies(Status,CondCode(ExecuteInstr))then
        Contents(AddrReg) := MemAddr
        DataOut := Contents'(DestReg)
        Bop := FinalOffset(ExecuteInstr)
        ExecuteMode := store
    elseif ExecuteMode = store then
        if ByteTransferInstr(ExecuteInstr) then
            Memory(Contents(AddrReg)) := DataOut
        else AssignWord(Contents(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
        Contents(AddrReg) := ALU("+",Aop,Bop,0)
        Bop := FinalOffset(ExecuteInstr)
        ExecuteMode := multi-load-init
    elseif ExecuteMode = multi-load-init then
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        DataIn := MemoryWord(Contents(AddrReg))
        Contents(AddrReg) := Contents(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(Contents(AddrReg))
            Contents(AddrReg) := Contents(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
        Contents(AddrReg) := ALU("+",Aop,Bop,0)
        TransferReg := FirstTransferReg(ExecuteInstr)
        DataOut := Contents'(FirstTransferReg(ExecuteInstr))
        Bop := FinalOffset(ExecuteInstr)
        ExecuteMode := multi-store-init
    elseif ExecuteMode = multi-store-init then
        if WriteBack(ExecuteInstr) then
            Contents(BaseOp(ExecuteInstr)) := ALU("+",Aop,Bop,0)
        endif
        AssignWord(Contents(AddrReg), DataOut)
        if TransferReg = LastTransferReg(ExecuteInstr) then
            if WritesPC(ExecuteInstr) then ExecuteMode := refill1
            else ExecuteMode := first-step
            endif
        else
            Contents(AddrReg) := Contents(AddrReg) + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
            DataOut := Contents'(NextTransferReg(ExecuteInstr, TransferReg))
            ExecuteMode := multi-store-loop
        endif
    elseif ExecuteMode = multi-store-loop then
        AssignWord(Contents(AddrReg), DataOut)
        if TransferReg = LastTransferReg(ExecuteInstr) then
            if WritesPC(ExecuteInstr) then ExecuteMode := refill1
            else ExecuteMode := first-step
            endif
        else
            Contents(AddrReg) := Contents(AddrReg) + 4
            TransferReg := NextTransferReg(ExecuteInstr, TransferReg)
            DataOut := Contents'(NextTransferReg(ExecuteInstr, TransferReg))
        endif
    endif
endif
```