# Formal Verification of an ARM processor

**Vishnu A. Patankar**
Department of ECE
Carnegie Mellon University
Pittsburgh, PA 15213
Email: vishnup@ece.cmu.edu

**Alok Jain**
Cadence Design Systems
Noida Export Processing Zone
Noida, India 201305
Email: alokj@cadence.com

**Randal E. Bryant**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Email: randy.bryant@cs.cmu.edu

## Abstract

This paper presents a detailed description of the application of a formal verification methodology to an ARM processor. The processor, a hybrid between the ARM7 and the StrongARM processors, uses features such as a 5-stage instruction pipeline, predicated execution, forwarding logic and multi-cycle instructions. The instruction set of the processor was defined as a set of abstract assertions. An implementation mapping was used to relate the abstract states in these assertions to detailed circuit states in the gate-level implementation of the processor. Symbolic Trajectory Evaluation was used to verify that the circuit fulfills each abstract assertion under the implementation mapping. The verification was done concurrently with the design implementation of the processor. Our verification did uncover 4 bugs that were reported back to the designer in a timely manner.

## 1. Introduction

Processor evolution has had the effect of increasing performance by the use of design techniques such as pipelining and parallelism. The complexities resulting from such techniques manifest as interactions between operations and contention for resources. Simulation has been used in industry to validate such designs. In order to claim that simulation exercises all possible behaviors of the design, we would require to exhaustively simulate the design. Exhaustive simulation is prohibitively expensive in time and space. Hence, industry relies on simulating a limited number of patterns which exercise a small fraction of the circuit. But such a validation procedure could lead to undetected bugs. This weakness leads us to think about formal verification. Formal verification uses a set of languages, tools and techniques to mathematically reason about the hardware system.

Our verification methodology, based on Symbolic Trajectory Evaluation, is able to verify an RTL, gate or switch-level implementation of a processor against its instruction set architecture. This paper focuses on the application of this methodology to verify the ARM core. This processor has many complicated features such as forwarding logic, instruction pipeline, pipeline interlock, multiple cycle instructions and conditionally executed instructions. A practical consideration in choosing this processor was that the ARM designers were available at CMU to provide descriptions of the design.

A high-level overview of our methodology and some of the related work is presented in Section 2. Section 3 discusses the architecture and implementation details of the ARM. An example of instruction execution and timing in the pipeline is detailed in this section. The steps required by our methodology to verify the

ARM are detailed in Section 4. The abstract assertion and the implementation mapping for a representative immediate bitwise-OR instruction is detailed in this section. The bugs discovered are presented in Section 5.

## 2. Verification Methodology

Our verification methodology can be used to show that an implementation correctly fulfills an abstract specification of the desired system behavior. The abstract specification defines the instruction set architecture of the processor. The specification is a set of *abstract assertions* defining the effect of each instruction on the user-visible state elements. The verification process has to bridge a wide gap between the detailed processor implementation and the abstract specification. To bridge this gap, the verification process requires some additional mapping information. The *implementation mapping* relates the abstract specification to the complex temporal and spatial behavior of the pipelined implementation. In effect, the mapping exposes the micro-architecture of the processor. The implementation mapping is a nondeterministic mapping defined in terms of state diagrams. As an example, an instruction might stall in a pipeline stage waiting to obtain the necessary resources. The order and timing in which these resources are granted vary, leading to nondeterministic behavior. Our methodology will verify the implementation under all possible order and timing.

The abstract specification and the implementation mapping are used to generate the *trajectory specification*. The trajectory specification consists of a set of *trajectory assertions*. Each abstract assertion gets mapped into a trajectory assertion. A modified form of symbolic simulation called Symbolic Trajectory Evaluation (STE) [1] is used to verify the set of trajectory assertions on the implementation.

The reader is referred to [2][3] for a more detailed description of our verification methodology.

### 2.1 Related Work

Beatty[4] laid down the foundation of our methodology for formal verification of processors. He used the methodology to verify the Hector microprocessor. A switch-level implementation of the processor was verified against its instruction set architecture. Hector was a simple non-pipelined processor whereas the ARM has a 5-stage pipeline.

Nelson[5] used our methodology to verify parts of a PowerPC implementation called the Cobra-Lite processor. This was a *post-facto* verification that was done after the Cobra-Lite processor had been designed and fabricated. Cobra-Lite is implemented as a set of interconnected functional units - the fixed point unit or integer unit (FXU), the Load Store Unit (LSU), the Branch Processing Unit (BPU) and the floating point unit (FPU). The FXU is responsible for executing all fixed point instructions other than loads and stores and it processes instructions in three stages - dispatch, decode and execute. Since the Cobra-Lite verification was effected by verifying individual functional units, complete processor verification would constitute modeling protocols on the interface signals between the unit under verification (UUV) and units that interact with the UUV and later reasoning that - since individual UUV’s work correctly - the overall system works correctly. In contrast to this, the entire ARM core was verified as a single unit without decomposing the verification task into smaller subtasks. The ARM verification was done concurrently with the design implementation leading to considerable overlap of the two phases. Another differentiating factor is that we used the Efficient Memory Modeling (EMM)[11] technique to reduce the system memory demands of the verification task.

Srivas[6] used PVS to verify the AAMP5 microprocessor. The AAMP5 has a large complex instruction set, multiple data types and addressing modes and a microcoded, pipelined implementation. Srivas decomposed the verification problem into three sub-problems: 1. A part that reasons about stalling behavior, 2. A part that reasons about individual instructions in the absence of stalling, 3. A part that combines the first two parts.

Sawada[7] used the ACL2 theorem prover to verify an out-of-order pipelined processor. The processor includes out-of-order execution and speculative instruction fetch. Sawada used a table to store an execution trace of instructions representing states in the implementation. The table representation helps to easily define various pipeline properties such as the absence of WAW-hazards. In a sense, the table captures the past history of the processor. In Symbolic Trajectory Evaluation, the initial state of the circuit is set to the most general state that reflects all possible past histories for the processor.

## 3. ARM Processor Architecture

The ARM CPU core is a 32-bit RISC processor macro-cell upon which the current generation of ARM processors is based. It has 32-bit data and address buses. It has a single 32-bit external data interface through which both instructions and data pass during execution. It includes 15 general purpose registers. A 5-stage pipeline is employed to speed the execution of instructions. Because branches cause the sequential flow of instructions to be interrupted, it is usual to employ the ARM’s conditional execution facility when possible. The ability of every instruction to be conditionally executed increases the chance that the program address references will run sequentially thereby allowing the memory sub-system to make predictions about the next address required. Non-sequential addresses are held for two cycles.

The implementation we used was a hybrid between the ARM7 [8] and StrongARM [9] cores. The memory interface was derived from the ARM7. The pipeline structure was derived from the StrongARM core. Figure 1 shows the core functional blocks and the pipeline organization that corresponds to the description in the next two sections.

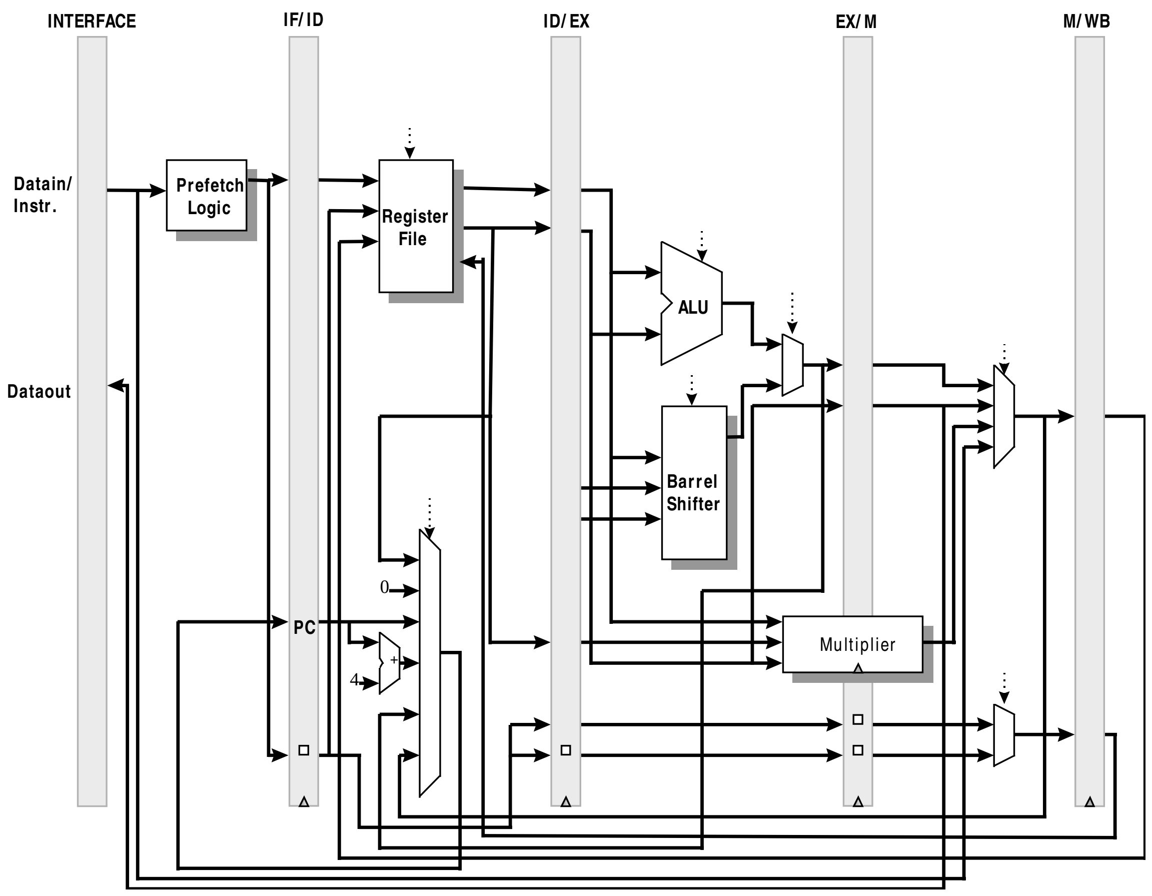

**Figure 1. ARM pipeline and main functional blocks**

## 3.1 CPU Core Functional Blocks

This section briefly explains some of the major functional blocks in the ARM such as the Register File, the Barrel Shifter, the ALU, the Booth's multiplier and the Control Logic.

**Register File and other Registers:** The ARM CPU core has a total of 16 registers comprising 15 general purpose 32-bit registers. The implementation of the register file has two read ports and one write port. Register R15 is the Program Counter. Because the PC is accessible to programmers, it can be included in standard instructions, and as a base for load and store instructions. This permits the easy generation of position independent code.

A further register, the Current Program Status Register (CPSR) is also accessible to programmers. This register stores the condition code flags. The condition codes flags are *Negative/Less* than (**N**), *Zero* (**Z**), *Carry/Borrow/Extend* (**C**) and *Overflow* (**O**). These flags may be changed as a result of arithmetic, logical and comparison operations in the CPU and which may be tested by all instructions to determine whether execution is to take place. Some of the other 28 bits of the 32-bit CPSR can be used for storing the CPU mode bits in future implementations of the ARM[10]. The current implementation of the ARM operates in the User Mode only.

**The Barrel Shifter:** The 32-bit Barrel Shifter implements shift/rotate logic of its input by any amount to produce an output within a fixed period. It has associated logic to allow values to be arithmetic shifted (preserving the sign-bit) or rotated through the carry bit (to give a 33-bit shift register).

**The ALU:** The ALU performs all arithmetic, logical and comparison operations on two input operands. There is a carry look-ahead within each 4-bit ALU block. A second level carry look-ahead option provides 16-bit carry look-ahead capability increasing the speed of the ALU at the expense of area.

**The Multiplier:** We did not verify the multiplier due to the exponential memory complexity associated with representing multipliers using BDDs[13]. Section 6 mentions as future work some word-level techniques that could be used to verify the multiplier.

## 3.2 Pipelining

The ARM uses a 5 stage instruction pipeline - Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (M) and Write-back (WB).

The ARM completes an instruction every clock cycle under most circumstances. The instruction set allows instructions to execute conditionally, since spending a single cycle not executing a conditional instruction is clearly quicker than a 3 cycle pipeline refill.

The pipeline has features like data forwarding and stalling to achieve maximal concurrency in instruction execution. Data forwarding can occur from **EX** ⇒ **ID** and **M** ⇒ **ID**. Since the **PC** (shown in the **IF/ID** pipe register) can be the target of a particular instruction, three other forwarding paths exist i.e. **EX** ⇒ **PC**, **M** ⇒ **PC** and **ID** ⇒ **PC**. The **ID** ⇒ **PC** forwarding path is activated when an instruction like **MOVAL PC ← R10, LSL 0** is executed (the **AL** mnemonic encodes that the instruction is always executed, and **LSL 0** implies that **R10** is moved as is - consequently the **EX** stage is not necessary for shifting). An instruction stalls in a particular stage when an earlier instruction in the pipeline takes multiple cycles or it has not got its operands.

The critical path of the processor is the **ID** stage since this is where many control decisions are made. Most instructions normally spend a single cycle in each stage. The cases in which the instruction spends more time in a particular stage are:

1. The instruction is waiting for the result of a previous instruction i.e. a data dependency.
2. The instruction inherently takes more than one cycle e.g. a multiply instruction which depending on its arguments, may spend up to 3 cycles in the multiplier.
3. The instruction is doing a memory access and this takes more than 1 cycle.

If, as a result of an instruction spending more than one cycle in a pipeline stage, the next pipeline stage becomes empty, then the processor will place a null instruction in this next pipeline stage. Once a null instruction is in the pipeline, it will spend one cycle in each remaining pipeline stage unless the pipeline is stalled. If an instruction completes its current instruction before the next pipeline stage is available, it will stall in the current pipeline stage. This will normally stall all previous pipeline stages. If, however, a previous pipeline stage is executing a multi-cycle operation, then that stage will not stall until the multi-cycle operation completes. In some instructions like long multiplies, the processor fetches and decodes the instructions as a single instruction but the **ID** stage passes multiple instructions in the **E**, **M** and **WB** stages. Also, in the case of a load instruction, since the off chip memory places the restriction that the address should be held for 2 cycles, the **ID** stage splits this load instruction into two. Splitting of the load instruction serves another purpose - base write-back (if the write-back bit is asserted) is done when the first copy of the instruction is in the **WB** stage and the actual data is loaded when the second copy of the instruction is in the **M** stage. Incidentally, this obviates the need to have two write ports in the register file despite the fact that a load instruction could potentially update two registers.

For the sake of exposition, the Figure 2 shows a trace of the flow of two consecutive load instructions in the pipeline. Assume that L1 uses the data loaded by L0 and I2 is independent of L0 and L1 and that base write-back is specified for L0. Some micro-achitectural features of the processor worth noting from the execution trace are:

1. L1 splits into two in the **ID** stage. This is because the memory system places the restriction that the address should be held for two cycles.
2. L0 does not advance into the **M** stage in cycle 4. The reason for this is that instruction I2 is being fetched at this time and both address and data share a bus in the ARM.
3. L1 does not advance into **EX** until L0 exits the **M** stage in cycle 6. This is because L1's effective address cannot be calculated since it depends on the data forwarded by L0 in cycle 6.
4. Two copies of L0 are in the **WB** stage - when the first copy of L0 is in **WB** the base is written back to the register file and when the second copy of L0 is in the **WB** stage loaded data is written back to the register file.

| Cycle | IF | ID | EX | M | WB |
|---|---|---|---|---|---|
| **1** | L0 | | | | |
| **2** | L1 | L0 | | | |
| **3** | L1 | L0 | L0 | | |
| **4** | I2 | L1 | L0 | - | |
| **5** | I2 | L1 | - | L0 | |
| **6** | I2 | L1 | - | L0 | L0 |
| **7** | I2 | L1 | L1 | - | L0 |
| **8** | I2 | I2 | L1 | - | - |
| **9** | I2 | I2 | I2 | L1 | - |

**Figure 2. Example trace of an interesting case**

## 4. ARM Processor Verification

This section applies our methodology to verify the ARM processor.

## 4.1 Abstract Specification

The first step is to define the instruction set of the ARM as a set of abstract assertion in a Hardware Specification Language. The exact syntax and associated formal semantics of this language is described in a companion paper submitted to this conference[17]. For the purposes of this paper, an abstract assertion is of the form: *P* **`LEADSTO`** *Q*, where *P* serves as the precondition and *Q* as the postcondition. *P* and *Q* are conjunction of clauses where each clause is an assignment to an abstract state element. As an example the abstract assertion for immediate bitwise-OR instruction is as follows:

```
(op IS OR)and(RA IS ra)and(RT IS rt)
and(Imm IS imm)and(Reg[ra] IS dataA)
  LEADSTO
(Reg[rt] IS dataA | imm)
```

The first two lines constitute the precondition of the abstract assertion. The clause (`op IS OR`) specifies that the opcode must be that for the immediate bitwise-OR instruction. The clauses (`RA IS ra`) and (`RT IS rt`) specify that the source and destination register identifiers maybe some symbolic values. The clause (`Imm IS imm`) says that the immediate source is the symbolic value `imm.` The clause (`Reg[ra] IS dataA`) specifies that the content of `Reg[ra]` is the symbolic value `dataA`. The last line specifies that in the postcondition, the content of the target register will contain the bitwise-OR of the immediate data and the register data.

## 4.2 Implementation Mapping

The next step is to define the implementation mapping. The implementation mapping has to relate the high-level information flow to a transfer of logic values on actual signals in the circuit. Our intention is to verify the instruction under verification (IUV) under every possible sequence of leading instructions. One possible way to represent all leading instruction streams, is to issue two completely symbolic instructions before fetching the IUV into the **IF** stage of the ARM pipeline since, potentially, the longest span of the forwarding is two instructions away from the IUV. The problem with this approach is that the symbolic computation required for the leading instructions is prohibitively large. Hence, we capture all possible leading instruction streams by exposing and asserting some of the internal state elements in the ARM pipeline. We get savings in symbolic computation because the IUV interacts with only a limited number of internal state elements but we still capture every possible sequence of leading instructions.

The implementation mapping consists of a *main machine* and a set of *map machines*. The main machine defines the flow of control of a generic instruction. The map machines define a mapping for each abstract state element in the abstract specification. The main machine and the map machines are modeled as *control graphs*. Control graphs are state diagrams with the capability of synchronization at specific time points. A control graph has two sets of vertices: **1.** State vertices that represent some non-zero duration of time and **2.** Event vertices that represent instantaneous time points. A control graph has a *source*, an event vertex with no incoming edges, and a *sink*, an event vertex with no outgoing edges. Nondeterminism is modeled as multiple outgoing edges from a vertex.

## Main Machine

The main machine for the ARM is shown in Figure 3. The vertices labelled **IF**, **ID**, **EX**, **M** and **WB** are state vertices that represent the five pipeline stages in the ARM. The rest of the vertices in the figure are event vertices. Essentially, an instruction can spend 1-4 cycles in **IF**, 1-4 cycles in **ID**, 1-2 cycles in **EX**, 1-2 cycles in **M** and 1-2 cycles in **WB**. An instruction can stay in a particular stage for a nondeterministic number of cycles - for example the vertices **$IF_1$**, **$IF_2$**, **$IF_3$**, **$IF_4$** represent that an instruction can stay in the **IF** stage for 1 or 2 or 3 or 4 cycles. The reason for this nondeterminism is that there might be preceding instructions in later stages that cause the current instruction to stall. For instance, in the example in section 2.2, L0 spends 1 cycle in **IF** (**$IF_1$**), L1 spends 2 cycles in **IF** (**$IF_1$**, **$IF_2$**) and I2 spends 4 cycles in **IF** (**$IF_1$**, **$IF_2$**, **$IF_3$**, **$IF_4$**). Essentially, the main machine has to capture the behaviors of all instructions in various stages of the processor. As the nextmarker shows, a successive instruction can be started at event vertex **Fetched**, thus overlapping the **ID** stage of the current instruction with the **IF** stage of the successive instruction.

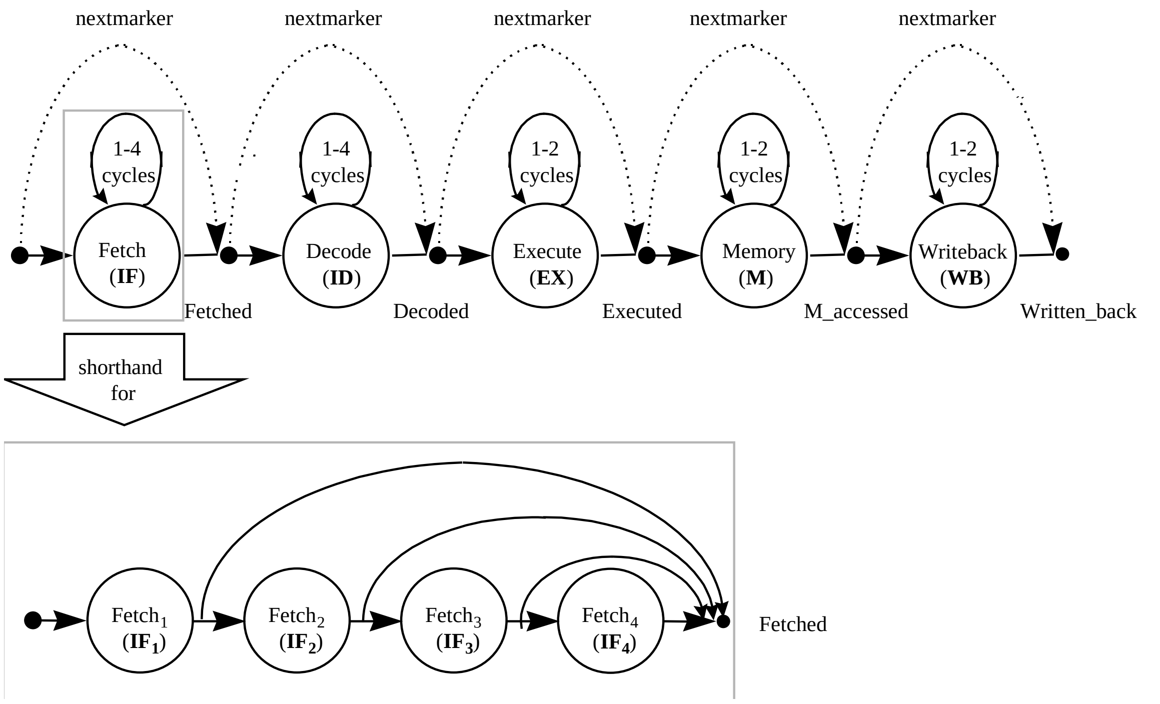

**Figure 3. Main machine for the ARM**

## IF Stage Map Machines

The map machines relate abstract clauses to assignments on detailed circuit states in the implementation. The map machine for the abstract clause (`op IS OR`) sets the opcode for the bitwise-OR instruction in the

**IF/ID** pipe register during the **IF** stage of the main machine. Bits 21-24 of the pipe register are set to 1100 to reflect a bitwise-OR operation as shown in Figure 4. The map machines for the register addresses (`RA` **`IS`** `ra`) and (`RT` **`IS`** `rt`) set bits 16-19 and 12-15 respectively of the pipe register. The map machine for the immediate field (`Imm` **`IS`** `imm`) sets the lower order bits 0-7 of the pipe.

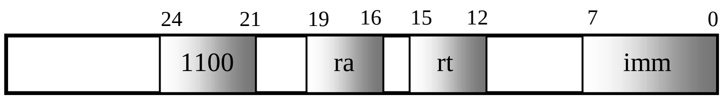

**Figure 4. IF/ID Pipe Register**

**ID and EX Stage Map Machines**

The decision to fetch the source operands from the register file or forward the data from one of two stages (**EX** or **M**) is made in the **ID** stage. Forwarding is dependent on two previous instructions. The three instructions involved in the decision are the previous instruction in the **ID** stage, the previous-to-previous instruction in the **EX** stage and the IUV in the **IF** stage. Eventually, the previous instruction will move into the **EX** stage and the previous-to-previous instruction will move into the **M** stage and forward the data to the IUV in the **ID** stage.

The map machine for the abstract clause (`Reg[ra]` **`IS`** `dataA`) is shown in Figure 5. The control graph is aligned with the **ID** stage of the main machine. The node assignments in the upper half of the state vertex are asserting signals in the implementation based on the following criteria:

1. If the address `ra` is the same as the target of the previous instruction `prev_rt` then the hold register in **EX** stage is asserted to the value `dataA`.
2. Else if the address `ra` is the same as the target of the previous-to-previous instruction then the hold register in the **M** stage is asserted to the value `dataA`.
3. Else the register file modelled as an EMM stores the data at the address location `ra`.

The abstract clause (`Reg[rt]` **`IS`** `dataA|imm`) appears in the postcondition. The map machine for this clause is automatically derived during the trajectory generation phase. The derived map machine is shown in Figure 5. The map machine is shifted by the nextmarker so that it gets aligned with the **EX** stage of the main machine. The lower half of the state vertex defines the desired response from the processor. The desired response is that the hold register in the **EX** stage should be assigned the bitwise-OR of `dataA` and the immediate operand.

This section has presented a somewhat simplified view of forwarding in the ARM. A few more internal states had to be exposed to set up all the necessary conditions for data forwarding.

.

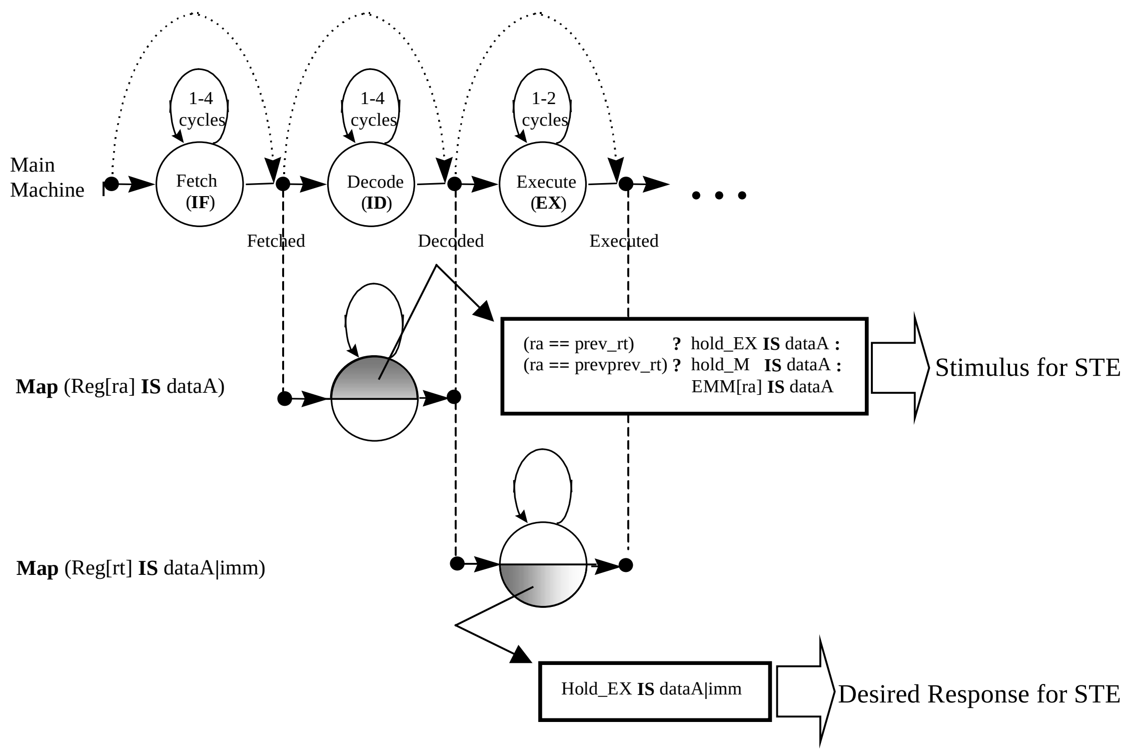

**Figure 5. Map machines for the ID and EX stages.**

### 4.3 Complexity Management

Several steps were undertaken to manage the complexity of the verification task:

**Efficient Memory Model:** Simulation models for memory arrays normally explicitly represent each memory bit. Symbolic simulation requires a symbolic variable for each bit of memory. A bit-level symbolic model checker would require the next-state function for each memory bit. This could prove to be prohibitive for large embedded memory arrays. In our verification, we replaced the Register File with a behavioral model called the Efficient Memory Model (EMM)[11], where the number of variables used to characterize the initial state of the memory is proportional to the number of memory accesses. Researchers have shown that It has been shown that EMM significantly outperforms the transistor level memory model when verifying simple pipelined datapaths[11].

**Separate Functional Unit Verification:** Some of the functional units like the PC, the ALU and the barrel-shifter were independently and separately verified. This reduced the memory requirements during the instruction verification phase which could now focus on verifying the control. The PC in the ARM pipeline can be updated in 6 different ways. This was verified by asserting certain internal and off-chip signals to symbolic values and checking if the PC is updated correctly in the next cycle. The ALU has 16 modes of operation which were all independently verified. The barrel shifter performs four types of shifts. Logical Shift Left (LSL), Logical Shift Right (LSR), Arithmetic Shift Right (ASR), Rotate Right and Rotate Right

Extended (actually a 33-bit rotation with the carry also taking part). If the shift amount is more than 32, the ARM shifts by *amount* **mod** 32.

### 4.4. Trajectory Generation

Section 4.2 gave a flavor of the map machines that were specified for the ARM processor. A total of 10 map machines were required for the processor. The abstract assertion for the immediate bitwise-OR operation and the implementation mapping were used to automatically generate the trajectory assertion. The trajectory assertion corresponds to the composition of the 10 map machines defined in the implementation mapping. Composition amounts to taking the cross-product of these aligned map machines under restrictions placed by the synchronization function.

### 4.5 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation was used to verify the immediate bitwise-OR trajectory assertion on a gate-level model of the ARM processor. The node assignments in the upper half of the state vertices define the stimulus for the simulator. The node assignments in the lower half of the state vertex define the desired response and state transitions.

The verification process uncovered a few bugs that are discussed in detail in the next section.

## 5. Bugs Uncovered

McMillan[12] noted that *the measure of success in integrating formal hardware verification methodologies is not the ability to provide formal guarantees of correctness, but the ability to detect design errors in a timely manner, as the design evolves*.

We discovered four bugs in the ARM core. Three of them were corner cases that resulted from designer oversight. Section 5.1 gives a background that helps to bring these bugs into context.

### 5.1 Background

All data processing instructions in the ARM accept one or more registers as their operands and always return the result to a register, optionally setting the condition code flags according to the result. The first source operand of a data processing instruction is (except for MOV and MVN) is always a register and is known in syntax definitions as `Rn`. Any register may be specified, including the PC (R15). The second operand (the only operand of  MOV and MVN) may either be a register `Rm` that is optionally shifted before use, or an 8-bit immediate constant optionally rotated before use. The shifted register forms allow one of the following types of multi-bit shifts:

LSL - Logical Shift Left

LSR - Logical Shift Right

ASR - Arithmetic Shift Right

ROR - Rotate Right

In each case, the number of bits to shift by is supplied by either as a constant or by another register. One further shift type is available - Rotate Right Extended (RRX) which performs a single bit rotation of the operand through the Carry Flag.

In an LSL operation, the contents of the register `Rm` are moved by the number of bits specified by the shift amount to more significant bit positions. The least significant bits thus revealed are filled with zeros and the most significant bits are discarded except that the least significant discarded bit becomes the shifter carry output (which may later set the Carry Flag in the CPSR). An LSL with a shift amount of 0 is treated as a special case - the shifter carry output is simply the old value of the Carry Flag and the contents of the operand register `Rm` are passed through unshifted.

The LSR, ASR and ROR operations behave like the LSL operation but for the shift direction and shift by 0. Since LSR by 0, ASR by 0 and ROR by 0 would duplicate the effect of LSL by 0, the ARM avoids this redundancy by doing the following. LSR by 0 is reserved and is used to encode LSR by 32, ASR by 0 is reserved and is used to encode ASR by 32 and ROR by 0 is reserved and is used to encode RRX. LSR by 32 yields a result of 0 but makes the shifter carry output become bit-31 of the source register. ASR by 32 duplicates the sign bit (bit-31) of the source register throughout the result (i.e. in 2's complement the result is a -1 or a 0) and the shifter carry output also takes the values of bit-31. ROR by 0 encodes the special case which performs RRX - the contents of the 33-bit shift register formed by concatenating the Carry Flag and `Rm` is rotated by a single bit to less significant bit positions and the new shifter carry output becomes the original bit-0.

### 5.2 The Bugs

Three of the bugs were shift-class bugs. The expected (specified) behavior of LSR by 0, ASR by 0 and ROR by 0 were unimplemented in the ARM processor design. On talking with the designer about these bugs, it became clear that this was due to oversight of what could be termed as the typical corner cases. These bugs went unnoticed when the ARM model was tested by the Dhrystone simulation benchmarks.

The fourth - a datapath-class bug - related to the conditional execution feature in the ARM. In the specification, condition code 1001 in an instruction represents that the instruction is executed if the **C** flag is cleared ***or*** the **Z** flag is set (unsigned lower or same). Instead, the implementation misinterpreted this condition code to represent the case that the instruction is executed if **C** flag is cleared ***and*** the **Z** flag is set.

We were able to detect design errors in a timely manner as the design evolved and provide valuable feedback to the designers.

## 6. Conclusion

This paper has shown the applicability of our methodology for formal verification of an ARM core using Symbolic Trajectory Evaluation. The user specified the instruction set architecture as a set of abstract assertions. An implementation mapping captured the micro-architecture of the processor. The abstract specification and the implementation mapping were used to generate a set of trajectory assertions. The trajectory assertion captures all possible nondeterministic interactions that can arise in the implementation due to an instruction. Symbolic Trajectory Evaluator was used to verify the trajectory assertion on a gate-level implementation of the ARM processor. The verification process uncovered four bugs that were reported back to the designers.

## 6.1 Future Work

**Complexity of implementation mapping:** It has been our experience that the implementation mapping for the ARM can become quite complex. An area of focus for future work would be to simplify or automate the generation of the mapping information as much as possible. Jain[2] has suggested that annotated timing diagrams could serve as an alternative for expressing the mapping.

**Other instructions:** Current work has concentrated on verifying the bitwise-OR instruction in the ARM processor. The next step is to verify an instruction from each of the other two instruction classes - Data-transfer and Branch instructions. We did not verify instructions that involve multiplication due to the exponential memory complexity associated with representing multipliers using BDDs[13]. It would be interesting to use word-level techniques [14] to verify these instructions.

**Complexity of verification:** An important area of future would be to study various techniques to reduce the complexity of verification using STE. We used the Efficient Memory Modeling technique to reduce the system memory demands of the verification task. Currently, our tools use a simple minded user-defined variable ordering. Future work would involve exploring various automated techniques such as dynamic variable ordering [15] and group sifting [16].

**Counterexample facility:** In the case of a counterexample, STE spits out a sum-of-products form of the BDD that represents the check failure. Sometimes, it is difficult to reverse engineer the counterexample to trace a potential bug. In this direction, one could incorporate into STE automatic advice-generation for reverse engineering based on hints given by the user.

**Languages:** During the verification of the shifter, we had to emulate the divide operation by splitting the divide of an 8-bit quantity by 32 into 7 cases. It would be helpful if the mod, multiply and divide operations would be supported by the language.

## References

[1] R. E. Bryant, D. L. Beatty and C. J. H. Seger, “Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation,” 28th Design Automation Conference, pp. 397-402, June 1991.

[2] A. Jain, “Formal Hardware Verification by Symbolic Trajectory Evaluation,” Ph.D. Thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, August 1997.

[3] A. Jain, K. Nelson, and R. E. Bryant, “Verifying Nondeterministic Implementations of Deterministic Systems”, Lecture Notes in Computer Science, Formal Methods in Computer Aided Design, pp. 109-125, November 1996.

[4] D.L. Beatty, “A Methodology for Formal Hardware Verification with Application to Microprocessors,” Ph.D. Thesis, Technical Report CMU-CS-93-190, School of Computer Science, Carnegie Mellon University, August 1993

[5] K. Nelson, A. Jain and R. E. Bryant, “Formal Verification of a Superscalar Execution Unit,” 34th Design Automation Conference, pp. 161-166, June 1997.

[6] M. K. Srivas and S. P. Miller, “Applying Formal Verification to the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods,” Formal Methods in System Design, 8(2), pp. 153-188, March 1996.

[7] J. Sawada and W. A. Hunt Jr. , “Trace Table Based Approach for Pipelined Microprocessor Verification,” Computer Aided Verification, CAV-97, pp. 364-375, June 1997.

[8] ARM 7 Data Sheet, ARM DDI 0020C, Advanced RISC Machines Ltd. (ARM), 1994.

[9] ARM 8 Data Sheet, ARM DDI 0080C, Advanced RISC Machines Ltd. (ARM), 1996.

[10] A. van Someren and C. Atack, “The ARM RISC Chip. A Programmer’s Guide,” Addison-Wesley Publishing Company, 1995.

[11] M. Velev, R. E. Bryant and A. Jain, “Efficient Modeling of Memory Arrays in Symbolic Simulation,” Computer Aided Verification, CAV-97, pp. 388-399, June 1997.

[12] K. L. McMillan, “Fitting Formal Methods into the Design Cycle,” 31st Design Automation Conference, pp. 314-319, June 1994.

[13] R. E. Bryant, “On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication,” IEEE Transactions on Computers, 40(2):pp. 205-213, 1991.

[14] Y. A. Chen and R. E. Bryant, "ACV: An Arithmetic Circuit Verifier", Proceedings of International Conference of Computer-Aid Design, Nov. 1996, pp. 361-365.

[15] R. Rudell, “Dynamic Variable Ordering for Ordered Binary Decision Diagrams”, Proceedings of International Conference of Computer-Aid Design, Nov. 1993, pp. 42-47.

[16] S. Panda and F. Somenzi, “Who Are the Variables in Your Neighborhood”, Proceedings of International Conference of Computer-Aid Design, Nov. 95, pp. 74-77.

[17] A. Jain, “A Case for Hardware Specification Languages,” Submitted for acceptance to VLSI design conference, January 99.