Design Verification Using Reverse Engineering

by

Jonathan D. Hauke

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science and Engineering

COMPUTER SCIENCE AND ENGINEERING
The University of Michigan
May 1999

Master's Committee:

Professor John P. Hayes, Chairman
Professor Trevor Mudge
Professor Karem Sakallah

ABSTRACT

Modern processors are very difficult to design and require advanced design verification methods to ensure that they function correctly. While simulation-based verification is the primary method used by industry, formal methods are gaining acceptance although they are limited to relatively small designs. Equivalence checking, for example, is useful for verifying the equivalence between two levels of a design hierarchy, but is not applicable above the widely used register-transfer level (RTL). For this reason, new design tools are necessary to verify that an RTL description of an implementation matches its original instruction set architecture (ISA), which specifies the processor components and instructions visible to a programmer.

This Master's thesis proposes a design verification method, Reverse Engineering Verification (REVE), based on analysis of an implementation's data flow. REVE takes a high-level RTL implementation of a new processor design and extracts (reverse engineers) specification information from the implementation's internal data paths. The reverse-engineered information is then matched with the original ISA specification to verify functional correctness of the implementation.

TABLE OF CONTENTS

APPENDICES

CHAPTER 1
INTRODUCTION

1.1 Industry Trends

Although general-purpose computers have existed for over 50 years, it is only in the last 10 years that the personal computer has become a part of everyday life. Because of the computer's widespread acceptance, the field of computer engineering is growing rapidly. The brain of a computer lies in its central processing unit, or CPU. This device, often simply called a *processor*, coordinates the communication between components inside the computer, and is responsible for executing the computer's software. In the standard Von Neumann model of a computer, the processor interacts with a memory system used for storing program instructions and data.

Today's processors are very complex and require large teams of people to design them. Figure 1.1 shows typical abstraction levels encountered in the design of a new processor. The design process usually proceeds from top to bottom, as more and more details of the design are developed. The first design step is to determine what the processor should be capable of doing, that is, create a specification for its behavior. This specification, typically called an *instruction set architecture* (ISA), refers to the programmer-visible instruction set and the processor

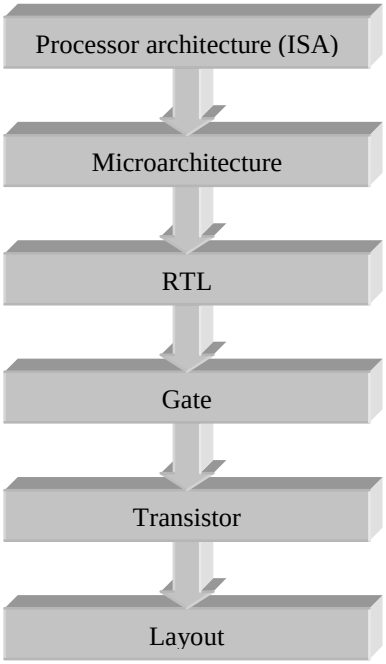


Figure 1.1: Levels of abstraction in the design of a processor

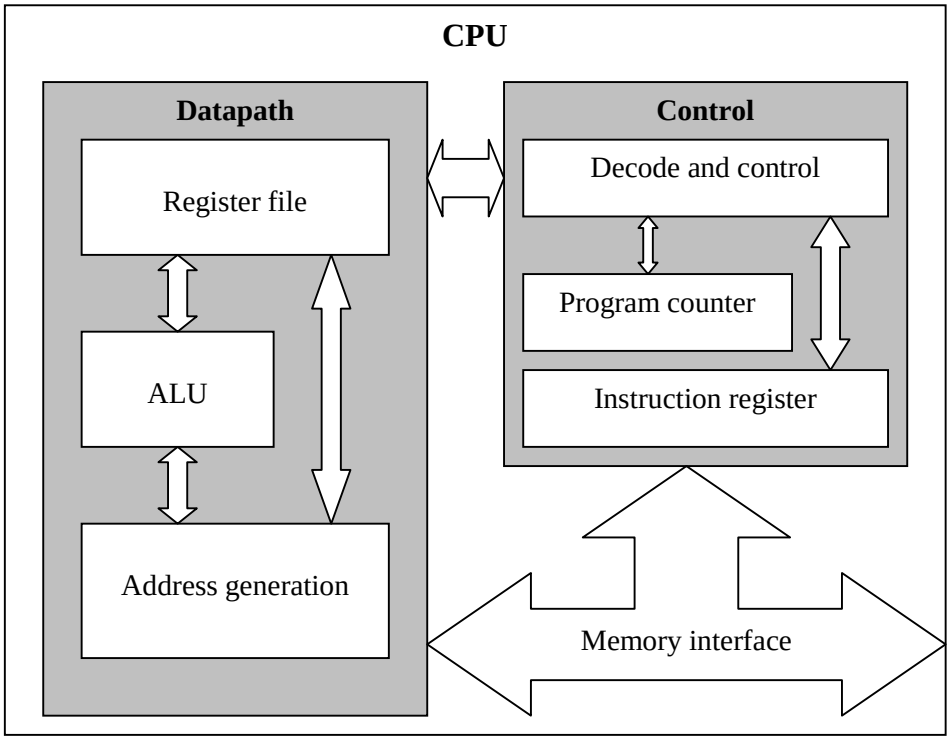


Figure 1.2: A typical processor (CPU) at the microarchitecture level

components seen by those instructions [22]. An implementation, which is a complete and manufacturable design of the processor, is created from this specification. The first implementation step is to design the processor's microarchitecture, that is, a high-level view of the design. Software tools such as simulators aid this process by measuring which microarchitecture is best for the task at hand. Figure 1.2 shows a typical CPU at the microarchitecture level. The Control block decodes the current instructions in the instruction register and ensures correct instruction execution by controlling the actions in the Datapath. Next, a register-transfer level (RTL) description is created, describing the processor's behavior in terms of the transfer of information between internal registers. The RTL description is usually written in a hardware description language (HDL) such as Verilog or VHDL. RTL designs are then converted to the gate-level, with the aid of tools such as logic synthesizers. For example, synthesis of the RTL add statement $Y=A+B$ would generate a gate-level implementation of an adder. This gate-level design is also usually written in a HDL.

Next, the gate-level design is converted to a corresponding transistor-level circuit. At this level, electrical issues such as capacitance and resistance affect the processor's design. Once the processor is specified at the transistor level, it is automatically translated into a layout view that describes the physical structure of the silicon chip on which a processor is implemented, and shows layers of materials such as metal and polysilicon. Finally, this layout design is sent to a fabrication facility where the chip is manufactured. After a prototype chip is built, it is returned to the design house where it undergoes further testing, and perhaps, costly redesign. Only after a chip passes all its tests is it mass-produced for eventual use in individual computers.

Integrated circuit transistor densities and complexities have increased at an amazing rate, necessitating the extensive use of automatic design programs known as computer aided design (CAD) tools to aid the designers. Synthesis CAD tools, for example, automatically convert a

design from a high to a low level, such as from the RTL level to the gate level. In addition, since today's processors are so complex, it is impossible to apply every possible combination of input signals for testing purposes. Therefore, CAD tools are required to generate a small set of tests to exercise a large area of the chip. Another important CAD application is design verification, that is, making sure that each design level correctly implements the original ISA specification. This topic is the focus of our thesis research.

1.2 Design Verification Methods

Verification is the task of making sure that a design's implementation matches its specification and so is free from design errors (bugs). In the case of processor design, the specification is the ISA, and the implementation is a complete description of the design at any lower level of the design hierarchy. In order to ensure that a processor is correct, its behavior must be verified at each level of the implementation process. Verification is of growing importance to processor design, as demonstrated by the Intel Pentium processor bug, which cost the manufacturer $475 million to replace defective chips [1]. This bug was a small, low-level error in a division lookup table and was not caught by simulation, since computed errors produced by the bug rarely occurred, and so were hard-to-detect "corner" cases.

Simulation, which is the process of applying random or specific instruction sequences to an implementation in progress and monitoring the output to ensure correctness with respect to the specification (ISA) [2], is the most common verification method. Entire processor designs at the RTL level and higher can be simulated, and smaller modules can be simulated at lower design levels. Because simulation is very general and has been around for a long time, it is considered a reliable verification method.

However, as processors grow in size and complexity, the number of possible input sequences grows, and it is impossible to exercise every one of them during simulation. The range of values of a processor's internal storage elements is its state space. As state spaces grow with new processors, simulation-based methods are increasingly inadequate for fully verifying a design. For example, a small design with 300 state elements (flip-flops) has approximately 10^{80} states, a number larger than the number of protons in the universe [2]. Simulating each of these states is obviously infeasible during a processor's design cycle.

Although simulation-based methods are the primary verification techniques used in industry, formal verification has been a hot topic in the last few years. However, it has experienced limited success because formal methods are hard to learn and time-consuming to apply. In addition, many formal methods are limited by the size of the circuits they can handle, or are only applicable to very special cases, such as verifying small state machines. However, as the size and complexity of processors continues to increase, verification methods that can go beyond simulation are necessary. Because of this, formal methods are typically used in conjunction with simulation to verify a processor [3, 4].

Equivalence checking, one of the first formal methods to gain acceptance, compares circuit designs developed at different phases of the design cycle, e.g., designs at the gate and transistor levels. In addition, it can verify the equivalence of two alternative designs at the same level of hierarchy. Theoretically, if divided into small enough pieces, the implementation of a processor's design level can be checked against the design in the next higher level, up to the RTL description of the machine. Circuit complexity is the main limitation on equivalence checking. Mainly for this reason, equivalence checking is limited to verifying small pieces of

combinational logic, and not an entire processor. Binary decision diagrams (BDDs), for representing functional behavior, are often used in equivalence checking [2]. Unfortunately, efficient BDDs do not exist for some components such as multipliers, which frequently appear in processor designs.

Model checking, a second well-known formal method, attempts to verify specific properties about a machine [5]. This method excels at verifying small state machines, but is also limited in the size of models it can handle. As the number of state bits in a machine grows linearly, the state space being analyzed grows exponentially. For example, adding one more state variable to a design doubles its state space. Because of this exponential growth, model checking runs into state space explosion problems and cannot verify the large state machines used for control purposes in a processor. These size problems can be reduced by abstracting away datapath details [6], but remain a problem for full processor verification. In addition, a model checker only verifies specific properties given to it by a designer, and is less useful for completely verifying that a processor's implementation satisfies its ISA.

A third type of formal verification is theorem proving, which is a method of verifying an implementation by proving theorems about its correctness [2]. Although this method is theoretically complete, it is extremely limited in the circuit sizes it can handle. In addition, because it is so complex, theorem proving has to be guided by hand and frequently requires a theorem proving expert to conduct the verification. Although several toy processors have been verified with theorem provers [7,8], theorem proving is usually far too complex and tedious for practical use.

Symbolic trajectory evaluation (STE), a method that is gaining some acceptance in industry, symbolically simulates a design [2]. While ordinary simulation only considers the binary signals 0 and 1 when verifying an implementation, STE handles variables and don't-care conditions as well. This allows the designer to verify the logic functionality of a module without exhaustively simulating it. STE's main limitation is that it requires a specification to be written in a custom language such as FL [9]. This extra specification step creates more work for the designer, and can itself introduce specification errors. In addition, STE methods frequently use BDDs as their storage representation and so suffer from the limitations of BDDs noted earlier.

Although the above formal methods are the most common, new verification methods are actively being researched, especially for special design features such as pipelining. A pipeline divides a processor into various sequential stages, and an instruction flows through a pipeline from stage to stage until it completes. This allows multiple instructions to execute in a processor at the same time, in a manner analogous to production flow on an assembly line [21], thereby increasing the instruction throughput of a machine. Levitt and Olukotun [10] demonstrate how a pipelined processor can be verified by "unpipelining" it, or iteratively merging the two deepest stages of the pipeline. After each merging operation, a check is performed to see whether the new merged model is equivalent to the old unmerged model. High-level knowledge about the design is essential to their methodology. Burch and Dill use quantifier-free first-order logic in conjunction with symbolic simulation to verify the equivalence of next-state functions in the specification and implementation of a processor [11]. Ho et al. [12] use state space exploration techniques to verify a high-level view of the control logic in terms of architectural components such as the program counter and instruction cache. Shen and Arvind [13] use term rewriting systems to model ISAs and verify high-level implementations at the microarchitecture level, but their method is primarily limited to comparing two high-level specifications. Finally, Van Campenhout et al. [14] use test generation methodology for physical fault testing in conjunction

with synthetic design error models to verify pipelined designs. Error models of this type have yet to gain widespread acceptance.

1.3 Thesis Problem

Equivalence checking is the most popular formal method in industry, and can partially verify a processor, if it is partitioned into small enough blocks. However, in order to provide full functional verification for the processor, an RTL model of the implementation must be verified against the original specification (ISA) of the processor. Several methods attempt to solve this problem, but are limited either by the circuit sizes they can handle [12], or by their ability to verify complex processors [13] allowing errors in the RTL model to remain undetected. Our goal is to derive a method for verifying a full RTL implementation that is not limited by the BDD or state space explosion problems occurring in other methods. Therefore, we developed a technique using reverse-engineering principles to reconstruct high-level specification information from a processor's data paths. Reverse engineering is the process of creating a specification from an implementation, and is the antithesis of normal design flow. It is commonly used for computer software tasks such as reconstructing program specifications and software maintenance [16].

The interconnections between the internal datapath and control components shown in Figure 1.2 are defined by a set of data paths that support the flow of data through the processor during instruction execution. The existence of specific data paths, such as from the register file to the program counter, indicates processor support for instruction types such as branches. Because of this, we can reverse engineer high-level information about supported instruction types, and compare this against the original ISA specification to verify the correctness of the implementation.

This thesis introduces our reverse engineering verification system (REVE). Chapter 2 describes the algorithm with which REVE verifies a processor. We illustrate REVE by using it to verify a simple unpipelined implementation of the hypothetical LC-2 processor. In Chapter 3, we discuss pipelining verification and extend REVE to support pipelined designs. In addition, we apply REVE to a pipelined implementation of the LC-2. Chapter 4 presents a case study of applying REVE to a commercial processor, the ARM 7. We discuss the verification results from this case study in detail, including the detected bugs. We state our conclusions in Chapter 5 and briefly discuss how REVE can be extended to handle more complex architectural features such as superscalar design.

CHAPTER 2
PROPOSED VERIFICATION METHODOLOGY (REVE)

Since existing methods are limited in their ability to solve high-level verification problems, our goal is to derive a method capable of completely verifying a design at the RTL level. This chapter describes our reverse-engineering verification method and demonstrates its usefulness by applying it to a small example.

2.1 Method Overview

In order to verify that an implementation is correct, we must compare what the designer wants to create with what is actually created. In the case of processor design, the designer wants to implement an ISA, that is, a set of predefined instructions, system registers, and associated rules of operation. In order to determine what is actually created, we reverse engineer the implementation by effectively stepping through the design process backwards. Figure 2.1 shows a high level view of our proposed approach which we call the Reverse Engineering VErification (REVE) system. A knowledge base is necessary to store a predefined set of rules about the

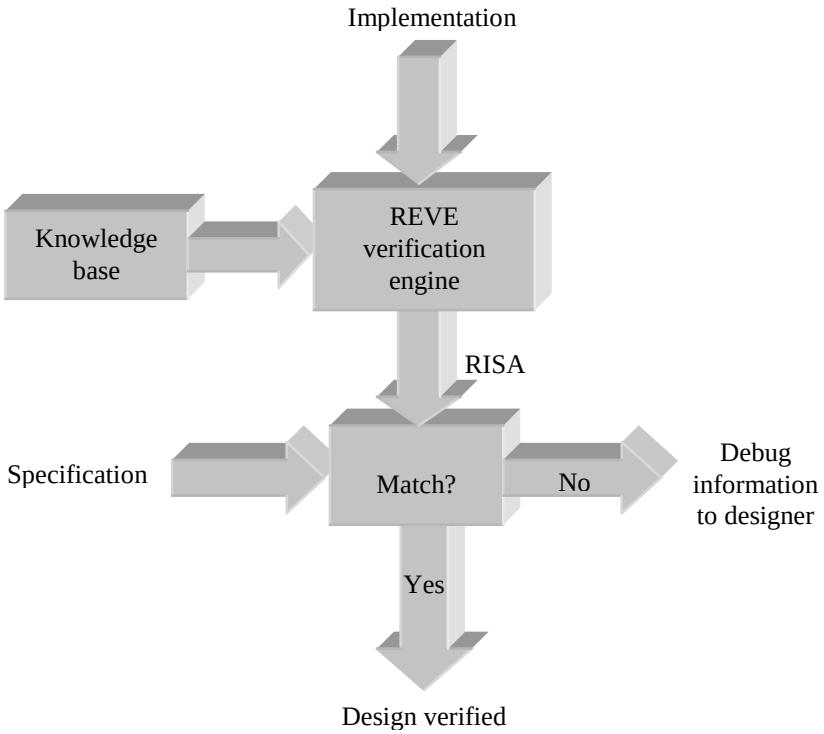


Figure 2.1: The proposed reverse-engineering verification system (REVE)

processor architectures of interest. The knowledge base interacts with the REVE verification engine, which also receives the designer's implementation as its input. This engine reverse engineers the implementation to obtain a reverse-engineered ISA (RISA). The RISA is then compared against the original ISA that the designer wants to implement. If the ISA and RISA match, then the design has been verified. If they do not match, then we have found a bug in the design, and debugging information is returned to the designer.

The underlying concept behind REVE is that comprehensive information about a processor's ISA can be extracted from its datapath interconnections, such as those shown in Figure 1.2. Datapath links between components such as the register file, the ALU, and the program counter, determine how architectural state, that is, the set of state-holding components defined in the ISA, is updated. Each instruction in an implemented instruction set has a specific data path (or set of data paths) that it traverses while in execution. Since normal design procedures derive data paths from an ISA specification, we use reverse engineering to do the opposite: create an ISA from the data path implications. This RISA contains a description of all instructions the implemented design is capable of executing. In fact, depending on the implementation, the RISA may include more instructions than the original ISA. For example, in an ISA with 32-bit opcodes, there are 2^{32} possible instructions. Typically an ISA leaves some of these instructions undefined, either to minimize internal hardware, or to leave room for future ISA expansion. Since the designer is only required to correctly implement the currently specified ISA, he can build the implementation to do anything when it encounters an undefined instruction. The REVE system is capable of reverse engineering *all* possible instructions in a machine, and so will identify both defined and undefined instructions. Therefore, it is possible for the RISA to be larger than, or a superset of, the original ISA. In order to prove that an implementation is correct, it is only necessary to ensure that the original ISA is a subset of the RISA.

In addition, as processor families are redesigned, legacy hardware may exist which serves no useful purpose in the current CPU. Instead of redesigning an entire processor, however, it may be easier to leave this hardware in place. Since this extra hardware may produce outputs that are not defined by the current ISA, this is a second way in which a RISA can be larger than its associated ISA.

2.1.1 Reverse Engineering

The REVE method is based on reverse engineering, which is defined as follows.

> "The act of creating a set of specifications for a piece of hardware by someone other than the original designers, primarily based upon analyzing and dimensioning a specimen or collection of specimens."[15]

The normal design process starts with a specification, usually the ISA, and builds an implementation to satisfy it. Our goal is to start with an implementation and use reverse-engineering principles to reconstruct a satisfying ISA. This resulting RISA is compared against the original ISA to ensure correctness of the implementation.

In the field of computer science, reverse engineering is frequently used for software tasks such as reconstructing lost program specifications and software maintenance [16]. Although reverse engineering also exists in hardware applications, it is primarily confined to low-level tasks, such as converting designs from the transistor to the gate level [17]. One exception to this is the successful reverse engineering of the ISCAS-85 benchmark suite by Hansen et al. [18].

This set of circuits used for benchmarking test generation programs is distributed as a set of gate-level netlists. By employing several ad hoc methods, Hansen et al. were able to reverse-engineer RTL specifications from these netlists. Apart from this example, however, reverse engineering has not been used for higher-level hardware analysis, and has not been used at all in the field of design verification.

The REVE method follows a similar approach to that of [18], as both utilize a knowledge base, implicit or explicit, to generate specifications for lower-level implementations. While the work by Hansen et al. primarily reverse engineers between the gate and the RTL level, the REVE method is used between the RTL level and the ISA. Their method utilizes several ad hoc techniques to step through the reverse-engineering process. The REVE method also uses a similar set of steps that are tailored to common processor implementations and defined in Section 2.1.3.

In order to minimize bias concerning what the designer intended, we intentionally limit information about the original specification from entering REVE. However, in order to obtain useful results and properly match components between the ISA and the RISA, we must establish certain rules. For example, REVE is targeted at classic Von Neumann style processors, which are a general model of today's CPUs having a single main memory used for both instruction and data storage. In order to aid REVE, basic knowledge about Von Neumann style processors is built into the verification system. This knowledge is separate from the verification engine, and is treated as a knowledge base, enabling the verification of different types of processors by substituting an appropriate knowledge base for each one. Although this knowledge base is central to REVE, we still wish to limit its required information, since ideally we want to verify a machine without predefined design constraints. The following section explains the rules in a sample knowledge base for our basic processor model.

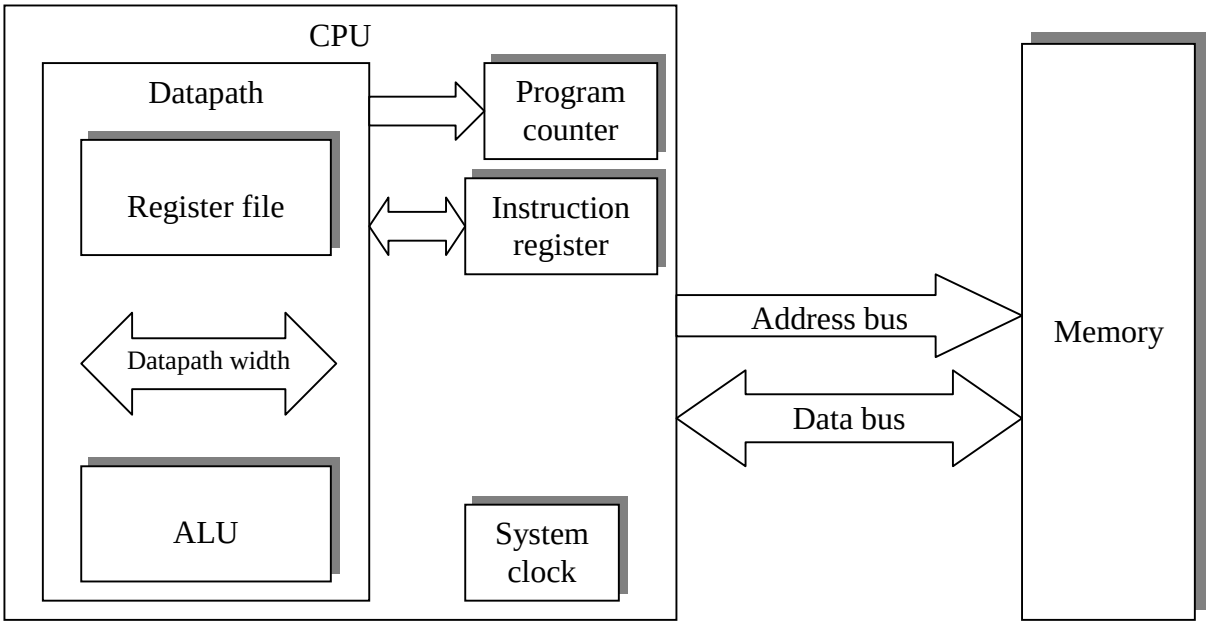


Figure 2.2: Basic processor knowledge base KB0

2.1.2 Knowledge Base

Figure 2.2 shows the basic knowledge base KB0 for a simple processor. The items within KB0 represent standard components in today's processors, and are summarized below.

- **Program Counter**: The program counter (PC) determines the processor's program flow. It is a register pointing to the address of the next instruction to be fetched from memory. In order to verify instructions that change program flow, the PC must be identified.

- **Instruction Register**: The instruction register (IR) holds the opcode of the currently executing instruction. Instructions are stored in memory, and are fetched based on the address stored in the PC. Since our goal is to reverse engineer all possible instructions, it is necessary to identify where these instructions are stored in the processor.

- **Register File**: In order to avoid costly memory-to-memory operations, most processors have a storage area in the CPU called a register file. While our method will work without an implemented register file (by reverse engineering all memory-to-memory operations), if one exists, it needs to be identified.

- **Data Bus**: In order for externally stored programs to interact with a CPU's internal elements, a path to main memory needs to be established. The path by which instructions and data travel between the processor and its memory is called the data bus. The data bus is an interface between the processor and memory, and must be identified by REVE.

- **Address Bus**: An address bus transmits a memory address, which points to the memory location that the data bus accesses. Since the address bus is also an interface between the processor and memory, it must also be identified by REVE.

- **System Clock**: This is necessary for later control analysis where we must have cycle-accurate information about system behavior. Since the system clock determines a cycle used to sequence the processor, REVE must be able to identify the clock signal.

The above components define a processor's architectural state, and in the REVE knowledge base, form a set of design ground rules that are built into the verification system. A processor's instruction set is based on the above components, and since the REVE system matches the RISA with the original ISA, these architectural components must match as well. Figure 2.3 illustrates the matching process and shows a case where the ISA and RISA, both of which are HDL descriptions, match. This example shows an ISA fragment for a jump to subroutine instruction with a conditional link. Bits 15:12, 10, and 9 of the instruction register are predefined opcode bits. If bit 11 of the instruction register is 1, then the address of the next instruction is loaded into register 7 of the register file. In all cases, the program counter's contents become the concatenation of bits 15 through 9 of the address of the next instruction, and bits 8 through 0 of the instruction register. Figure 2.4 considers the same ISA fragment where the ISA and RISA do not match. In this example, the implementation uses the address of the *current* instruction, instead of the *next* instruction for the register link and concatenation. This

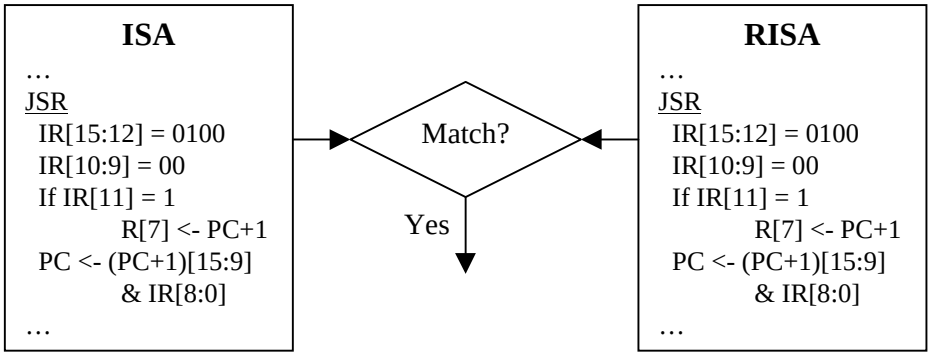


Figure 2.3: Matching fragments of an ISA and RISA

bug creates a match failure, and the unmatched portions of the two instructions are sent to the designer. In both cases, architectural components such as the PC and IR are necessary for comparing the two instruction sets. Since essentially all instruction-set processors include these five architectural components, adding them to the knowledge base does not greatly diminish REVE's verification scope.

In addition to the previous components, two other pieces of information shown in Figure 2.2 and listed below are added to the knowledge base.

- **ALU(s)**: A processor contains at least one internal processing element for arithmetic and logical operations, and most instruction sets have a large class of data-processing instructions. Since these instructions interact with the ALUs during execution, it is helpful to identify any ALUs present.

- **Datapath width**: Since the REVE approach uses data paths for reverse engineering, it is necessary to separate internal data and control paths. Knowing the standard datapath width aids in this separation, since most data paths are of this width, or a fraction of this width.

Figure 2.5 summarizes the knowledge contents of KB0.

A typical processor supports four basic instruction types: data-processing, branch, load, and store. Data-processing instructions modify either the processor's internal state (register file) or external state (memory). Branch instructions modify the PC in order to change the program flow. Load instructions modify the processor's internal state, while stores modify the external

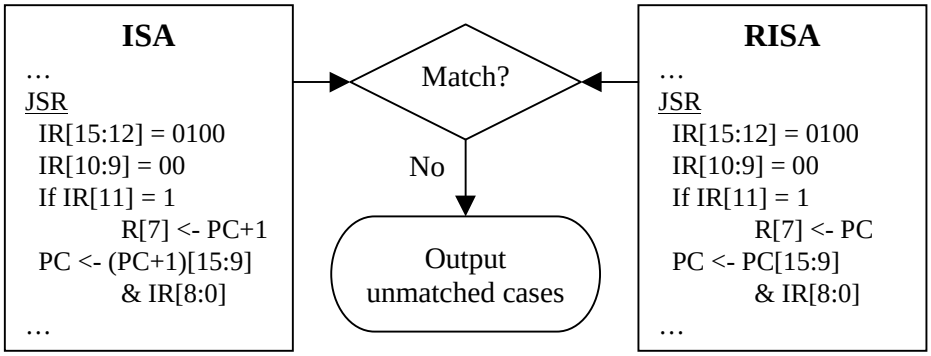


Figure 2.4: Unmatched fragments of an ISA and RISA

Knowledge Base Entry	Function
Program counter	Points to address of next instruction
Instruction register	Holds opcode of current instruction
Register file	The CPU's internal storage area
Data bus	Path by which data travels between processor and memory
Address bus	Transmits address of memory location accessed by the data bus
System clock signal	Defines a basic RTL step or clock cycle
ALU	Internal data processing element
Datapath width	Width of common internal datapaths

Figure 2.5: Information in knowledge base KB0

state. REVE assumes that every instruction in an ISA is either one of these four types, or is a combination of two or more types.

Additional information may be necessary to handle advanced properties such as pipelining or superscalar design. We examine pipelining in detail in Chapter 3.

2.1.3 Processor Verification Algorithm

This section describes the basic procedures of the REVE verification system and briefly discusses CAD tool implementation. Although REVE can be extended to lower design levels, we concentrate on implementations at the RTL level, since our original goal was to solve high-level verification problems. Figure 2.6 shows a flowchart of REVE's major steps.

The first step is to identify the components specified in the knowledge base. It is frequently easy to identify major components such as the register file or the program counter when analyzing a design, and a CAD tool could quickly traverse a design using ad hoc methods like HDL string matching for identification purposes. Since all succeeding verification steps use these components, however, misidentification at this point can cause erroneous results to ripple through the analysis, resulting in an incorrect RISA. Therefore, a manual check that these components are identified correctly may be desirable at this point.

The next step (step 2 of Figure 2.6) is to identify all data paths in the implementation by looking for paths whose widths are equal to the standard datapath width. A CAD tool can easily determine this from the original HDL code of the implementation, since it simply needs to search through modules looking for buses or registers of size equal to the datapath width.

Figure 2.7 shows a representative implementation fragment with data paths between architectural components. Each data path is associated with one or more of the four basic instruction types identified in Section 2.1.2. These instruction types modify architectural components as shown in Figure 2.8. In order to trace the data paths for each instruction type, we start at the (final) component modified by the instruction. For example, with branch instructions, we trace all data paths that modify the program counter. Each path is traced until an architectural state-holding component identified by the knowledge base is reached. In Figure 2.7, several data paths are traced for the branch instruction type. The first data path is from the register file to the program counter via input 0 of Mux1. The remaining data paths reach the program counter via the ALU and input 1 of Mux1.

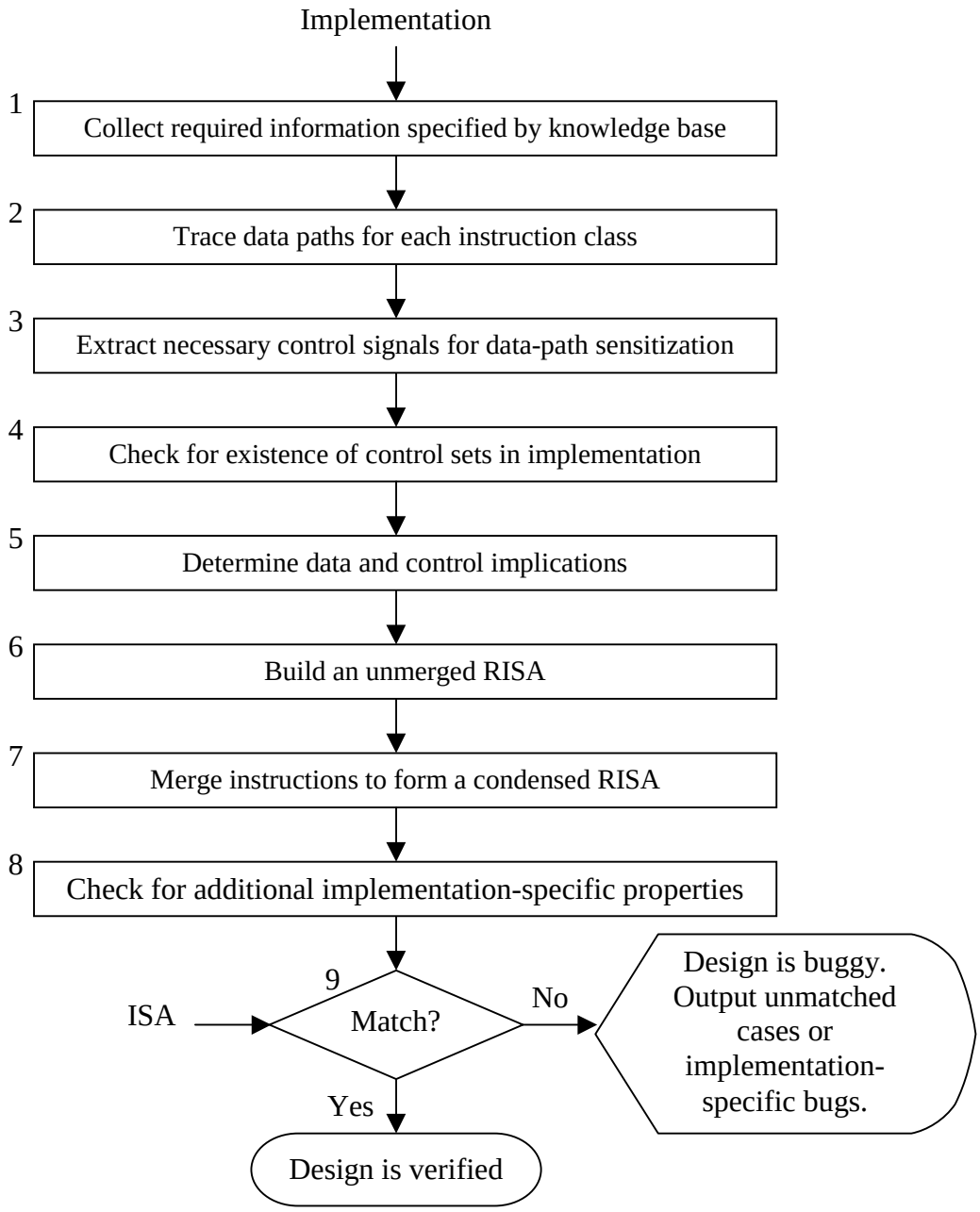


Figure 2.6: Flowchart of the REVE algorithm

Many instruction sets have a large class of data-processing instructions that use the ALU. These instructions typically load their results into the register file, or store them directly to memory. Therefore, in order to separate data-processing instructions from load and store instructions, all load or store data paths that pass through the ALU are detected under the data-processing instruction type.

Once we trace paths that are equal to the standard data path in width, we look for paths that have submultiples of this width. Although this search may erroneously identify control paths as data paths, it is necessary to take an overly cautious approach and include all detected

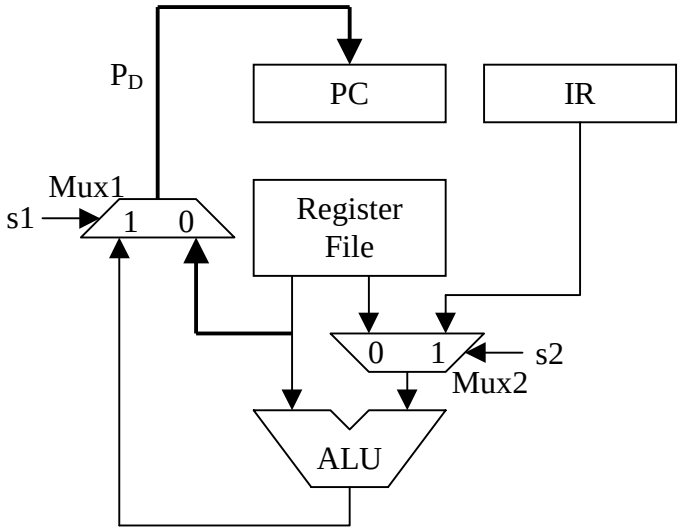


Figure 2.7: Implementation fragment of a simple processor

paths in the analysis. If we discover later that one of the extracted paths is actually a control path, we can discard any reverse-engineered information associated with it.

After identifying the data paths for the various instruction types, step 3 determines the necessary control signals in order to sensitize each one, which involves opening any multiplexers, registers, or latches necessary for data to flow on the path. For example, in Figure 2.7, the path P_D between the register file and the program counter that passes through input 0 of Mux1 is sensitized by setting the multiplexer select signal s1 to 0. If a data path passes through a clocked register, then sensitization of that data path may require control signals to be set during more than one cycle. For example, if a data path between the register file and program counter contains an additional register as shown in Figure 2.9, proper sensitization for P_D requires that s1 be 0 one cycle before the program counter's load signal is turned on. The control analysis in step 3 may identify more than one set of control signals that exercises each data path, e.g., if the logic functions generating these control signals have don't care conditions. It is important to keep track of all of these control sets, since each set, combined with its associated data path, is treated separately in succeeding verification steps.

In step 4, we check if the control sets identified in step 3 exist in the implementation, requiring the generation of logic equations for each control signal, as shown in Figure 2.10. These equations frequently already appear in the implementation, but if not, can be reverse engineered by the methods described in [18]. Sometimes the control equations are given in a tabular form. In order to check that all necessary control signals exist, we look for intersections between the equations for control signals in each reverse-engineered control set. If we discover

Instruction type	**Component modified**	**Sample action**
Data-processing	Register file or data bus	$R_k \leftarrow f(R_i, (R_j \text{ or IR}))$
Branch	Program counter	$PC \leftarrow f(R_i, ALU)$
Load	Register file	$R_k \leftarrow f(\text{Data bus})$
Store	Data bus	$\text{Data bus} \leftarrow f(R_i)$

Figure 2.8: Basic instruction types with sample operations

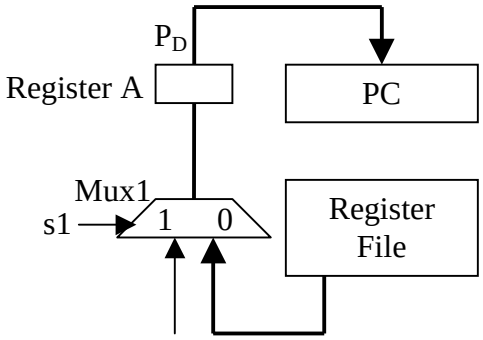


Figure 2.9: Data path with register

that any control sets do not exist in the implementation, they are removed from our analysis. Similarly, if no control sets exist for some data path, then the data path is not sensitized and can also be removed from analysis.

When a data path is used in more than one cycle by an instruction, we must ensure that control signals necessary for data path sensitization are turned on in the correct cycles. This requires an examination of the next-state equations generated in the previous step for each control signal in order to locate all reachable states. For example, if we determine that control signal A must be turned on in cycle P, and control signal B must be on in cycle P – 1, we first check to see whether control signal A is ever turned on. If we find that A is turned on in a control state S, we use the next-state equations to generate the set of states S' which can lead to S. Then, we examine this set S' to see whether B is ever turned on. If B is turned on in S', then our data path is sensitizable. However, if B is not turned on in S', we can eliminate this data path from consideration. Although this analysis can become complex, it is systematic and can be easily handled by a CAD tool.

The preceding control analysis narrows the current list of data paths to a set of sensitizable ones. Next, in step 5, we determine data and control implications resulting from the control-signal assignments made in the previous step. Frequently, a control signal only turns on in response to other control signals. In the data path between the register file and the program counter (Figure 2.11), we may discover that sensitization of P_D also results in the propagation of certain bits, in this case, bits 8:6 of the instruction register IR to the control inputs of the register file. This propagation is *implied* by the control assignments in step 4. Both data and control paths may be implied in this way, and information from such implications is used in building the RISA.

The next operation (step 6) is to create the RISA. After the previous control analysis, all retained data paths correspond to one of the four basic instruction types. Each data path represents one instruction, and the data and control implications from step 5 define each instruction's functionality by identifying all flows of data through the processor with a given set of control signals. These data and control implications relate each instruction to components such as the instruction register (where the opcode and operand bits reside), and allow us to

```
assign #1 ex_fwd_a_1 =
    ((ir_ex[‘BaseR]==ir_me[‘DR]) & (ex_me_will_write)) ? 1’b1: 1’b0;
```

Figure 2.10: Logic equation for a control signal in Verilog

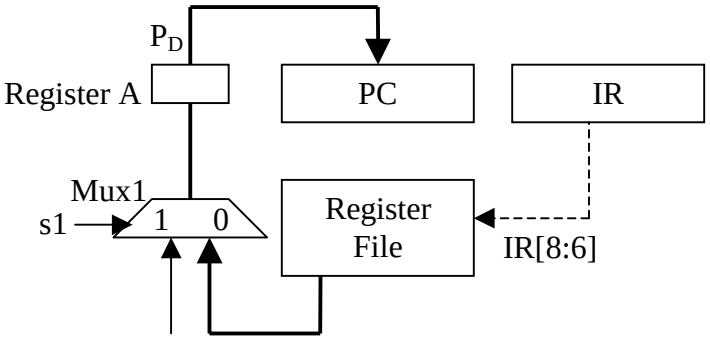


Figure 2.11: Control implications from reverse engineering

generate each instruction's exact functionality.

Once we have compiled a list of reverse-engineered instructions, we merge similar ones together in step 7 to form the final RISA. For example, data-processing instructions frequently use one bit of the IR to denote whether an operand is a register name or an immediate value. In the unmerged RISA, each of these cases (register and immediate) is a separate instruction. Since the two instructions have the same overall functionality in a typical ISA such as that of [23], they are treated as a single instruction with one bit designating the addressing mode.

As a second example, suppose the unmerged RISA contains the two instructions shown in Figure 2.12. These two instructions correspond to a single branch-and-link instruction, which loads the program counter into the register file and branches to a new memory location. The link part of the instruction is detected under the load instruction class, while the branch is detected under the branch instruction class. Since the opcode bits appearing in the RISA are identical for these two instructions, they are merged to form a single instruction that conditionally links based on bit 11 of the instruction register. After the merging step, we have the final RISA. Step 8 is now used to verify any special features of the implementation such as pipelining and superscalar design.

Finally, step 9 compares the RISA to the original ISA. It does so by storing both ISAs in a standard template form, as shown in Figure 2.13. Each template entry contains the assigned opcode bits, and the instruction functionality. A sample template entry for the branch-and-link instruction previously described is also shown. This functional description contains entries of the form:

output ← input1 (function) input2.

If the original ISA is a subset of the RISA, then the implementation is assumed to be correct. If the ISA is not a subset of the RISA, then ISA instructions not included in this subset are not implemented correctly. In order to aid the designer in tracking these bugs, REVE outputs two

	Opcode Bits					
Instruction type	**15**	**14**	**13**	**12**	**11**	**Function**
Branch	0	1	0	0	X	PC ← R(IR[8:6])
Load	0	1	0	0	1	R(IR[5:3]) ← PC +1

Figure 2.12: Instruction candidates for RISA merging

Instruction	Assigned opcode bits	Functionality
Branch and load	0100 xxxx xxxx xxxx	If IR[11] == 1 R(IR[5:3]) ← PC + 1 PC ← R(IR[8:6])

Figure 2.13: Sample template for storing ISA instructions

pieces of information: the ISA instructions not included in the RISA, and any unmatched instructions in the RISA. The RISA contains an implemented algorithm for each instruction, and the designer can compare this against the original ISA's algorithms to determine where the implementation is incorrect. We feel that this is more valuable than simply reporting a test vector for bug-tracking purposes. A designer can use a test vector it is used to determine the difference between what is implemented, and what was meant to be implemented. Since REVE reports instruction algorithms in the functionality portion of each template entry for the RISA (what is implemented) and the ISA (what is intended), we have eliminated one step of the debugging process.

Figure 2.14 summarizes the steps taken by REVE and defines the input and output data that should be included in a CAD tool. Although we have not implemented REVE as a CAD tool, we have considered CAD issues during our analysis, and believe that each of REVE's steps can be implemented in software. Steps 1 through 4 search either the HDL

Input

Name of top-level module
ISA specification (in a predefined template)

Steps

1. Traverse all modules of the implementation until all components described by the knowledge base KB0 are identified.
   - Prompt user asking whether these assignments are correct. If not, the user should define acceptable module names for each component.
2. Reverse engineer data paths by tracing each path in the implementation.
3. Determine appropriate control signals for exercising each data path.
4. Check whether each data path is exercised by its corresponding control signals. If a data path is exercised by more than one set of control signals, separate these cases. Eliminate any control sets that are never exercised.
5. For remaining data path and control set pairs, determine implications of prior control assignments.
6. For each remaining data path, derive an associated instruction based on propagation of instruction register values throughout the data path.
7. Merge associated instructions to form a condensed RISA.
8. Check for additional implementation-specific properties.
9. Check whether the original ISA is a subset of the RISA.

Output

Verification decision
List of instructions in ISA that are not implemented correctly
List of unmatched instructions in RISA

Figure 2.14: The REVE algorithm

specification, or equivalent internal representations such as linked lists to store data paths, or a tree structure to store module hierarchy. These searches can be implemented by a number of well-known search algorithms [19]. The remaining steps involve propagation of signals throughout a datapath to determine data and control implications, and pattern matching to merge the RISA. These can also be tackled by directly using the HDL, or by converting the HDL to another representation.

2.1.4 Hierarchical Reverse Engineering

As previously stated, REVE targets processor designs at a high level (RTL) where design verification tools do not presently exist. The underlying reverse engineering in REVE, however, is not limited to RTL analysis. CAD tools already exist which reverse engineer between transistor and gate level designs [17], and as Hansen et al. [18] demonstrated on smaller datapath modules, reverse engineering can be applied between the gate and RTL levels of design. Their reverse engineering was done with a collection of ad hoc methods using an implied knowledge base. In order to identify components such as adders, decoders, and multiplexers, designs were compared against standard designs in IC databooks and textbooks. In cases where simple pattern matching between an implemented design and textbook designs was not successful, various other techniques were used. These ranged from simple procedures such as identifying shared names between components to more complex techniques for logic function construction and matching.

Although the REVE method appears to be most useful at higher levels of abstraction, it can also be applied in a hierarchical manner to multiple levels. If we incorporate lower levels of knowledge similar to [18], REVE can reverse engineer an ISA from a gate-level implementation. This could be particularly useful when a design is written in a modularized logic format rather than standard RTL. In this case, pieces of the processor are reverse engineered one module at a time. For example, a gate-level ALU module can be reverse engineered to determine the logic functions and modes that it supports, and this information can be stored in an algorithmic form for use in building the RISA.

2.2 Illustrative Example: LC-2

To demonstrate the feasibility of the REVE system, we will apply it to several implementations of a small processor, the LC-2 [20]. Although this processor is a “toy” example and is much less complex than most commercial processors, it does include instructions from each of the major instruction classes and has a classic Von Neumann architecture. This section describes the main features of this processor and shows how our methodology is easily applied to it.

2.2.1 Processor ISA

The LC-2 (Little Computer 2) is a small 16-bit processor designed at the University of Michigan for use in introductory computer engineering courses. It employs 16 instructions, which are summarized in Figure 2.15. A full description of each LC-2 instruction can be found in Appendix A. The instruction set includes four data-processing instructions (ADD, AND, NOT, NOP), four branch instructions (BR, JSR, JMP, RET), four load instructions (LD, LDI, LDR, LEA), three store instructions (ST, STI, STR), and one control instruction (TRAP). There are eight 16-bit registers, and memory addresses are 16 bits wide, allowing for 128KB of memory. The LC-2 supports register, immediate, direct, indirect, and base+index addressing modes. The direct addressing mode generates addresses by concatenating the top 7 bits of the PC and the bottom 9 bits of the IR. There are three condition codes in the LC-2 (negative N, zero Z, and positive P) that are modified by data-processing and load instructions, and are used by branch instructions to determine whether or not a branch is taken.

While most features in the LC-2 are straightforward, several peculiarities are worth mentioning. First, the ISA frequently refers to the PC (program counter) in its instruction descriptions. In the LC-2, all such references to PC designate the address of the *next* instruction to be executed. This creates some confusion, as it is normally assumed is that PC refers to the

Mnemonic name	**Instruction bits**															
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
*ADD	0001				DR			SR1			0	00		SR2		
*ADD	0001				DR			SR1			1	Imm5				
*AND	0101				DR			SR1			0	00		SR2		
*AND	0101				DR			SR1			1	Imm5				
BR	1000				N	Z	P	Page offset								
JSR	0100				L	00		Page offset								
JSRR	1100				L	00		BaseR			Index6					
*LD	0010				DR			Page offset								
*LDI	1010				DR			Page offset								
*LDR	0110				DR			BaseR			Index6					
*LEA	1110				DR			Page offset								
NOP	0000				000000000000											
*NOT	1001				DR			SR			111111					
RET	1101				000000000000											
ST	0011				SR			Page offset								
STI	1011				SR			Page offset								
STR	0111				SR			BaseR			Index6					
TRAP	1111				0000			Trapvect8								

DR: Destination register
SR: Source register
BaseR: Base register
* Modifies condition codes
N: Negative condition code
Z: Zero condition code
P: Positive condition code
Imm5: 5-bit immediate address
Index6: 6-bit index
Trapvect8: 8-bit trap vector offset

Figure 2.15: Summary of the LC-2's instructions and their formats

address of the instruction *currently* in execution. Therefore, since each address is 16-bits wide, a reference to "PC" in the LC-2 ISA is usually denoted by PC + 1. We found that this anomaly frequently caused bugs in student implementations of the LC-2.

An interesting instruction in the LC-2 is TRAP. This instruction is used for executing system calls based on a trap vector, which points to a trap vector table stored in the first 256 memory locations. The trap vector offset is encoded within the instruction, and proper operation dictates that the PC should switch to the address *specified by* the trap vector, and not to the trap vector itself.

Finally, while most LC-2 instructions are simple, LDI and STI are quite complicated. These instructions perform indirect accesses to memory and involve two memory lookups per instruction. Concatenating the first 7 bits of the program counter and the last 9 bits of the instruction register forms the first address. Data residing at this address are used as the address for a *second* memory lookup, which performs the actual load or store. Depending on the implementation of these instructions, this double access of memory can lead to loops in instruction execution. In our control analysis, these loops must be identified along with the loop control variable (typically a counter), in order to avoid tracing infinite loops.

2.2.2 Sample Implementations

The computer architecture course EECS470 taught at the University of Michigan in Fall 1998 required students to implement the LC-2 as a class project. Since each student built a separate implementation, we had several LC-2 designs at our disposal. The examples described in this section are of unpipelined models that the students built early in the semester. The design data received from the students consists of their processor's Verilog HDL code, implemented at the gate level. We intentionally did not request extra information about each student's design philosophy, in order to avoid forming preconceived notions about the designs. The block diagrams shown in Figure 2.16 were generated by us and not by the students. Implementations B, C, and D were designed in EECS470 and are implemented at the gate level. Implementation A is from an earlier student design project, and although the datapath is implemented at the gate level, the processor's control is implemented at the RTL level.

Before delving into a verification example, it is worthwhile to note some of the differences among the implementations. One difference is the origin of the first input to the Merge block. This block concatenates bits from the PC and IR used by the direct addressing mode. Several of the implementations take the first input from PC, and several take it from PC + 1. This is due to confusion about what the ISA means by the term PC, as noted earlier. A second difference between implementations concerns the output of the ZEXT8 block. This block performs zero-extension of the 8-bit trap vector to a full 16-bit address. The last 8 bits of a TRAP instruction (Figure 2.15) form a trap vector which points to a location L in the first 256 bytes of memory. As previously noted, a TRAP instruction should begin execution at the address L. We see that implementation C has the output of ZEXT8 going to the Address Bus (correct), while implementations B and D have it going directly to PC (incorrect).

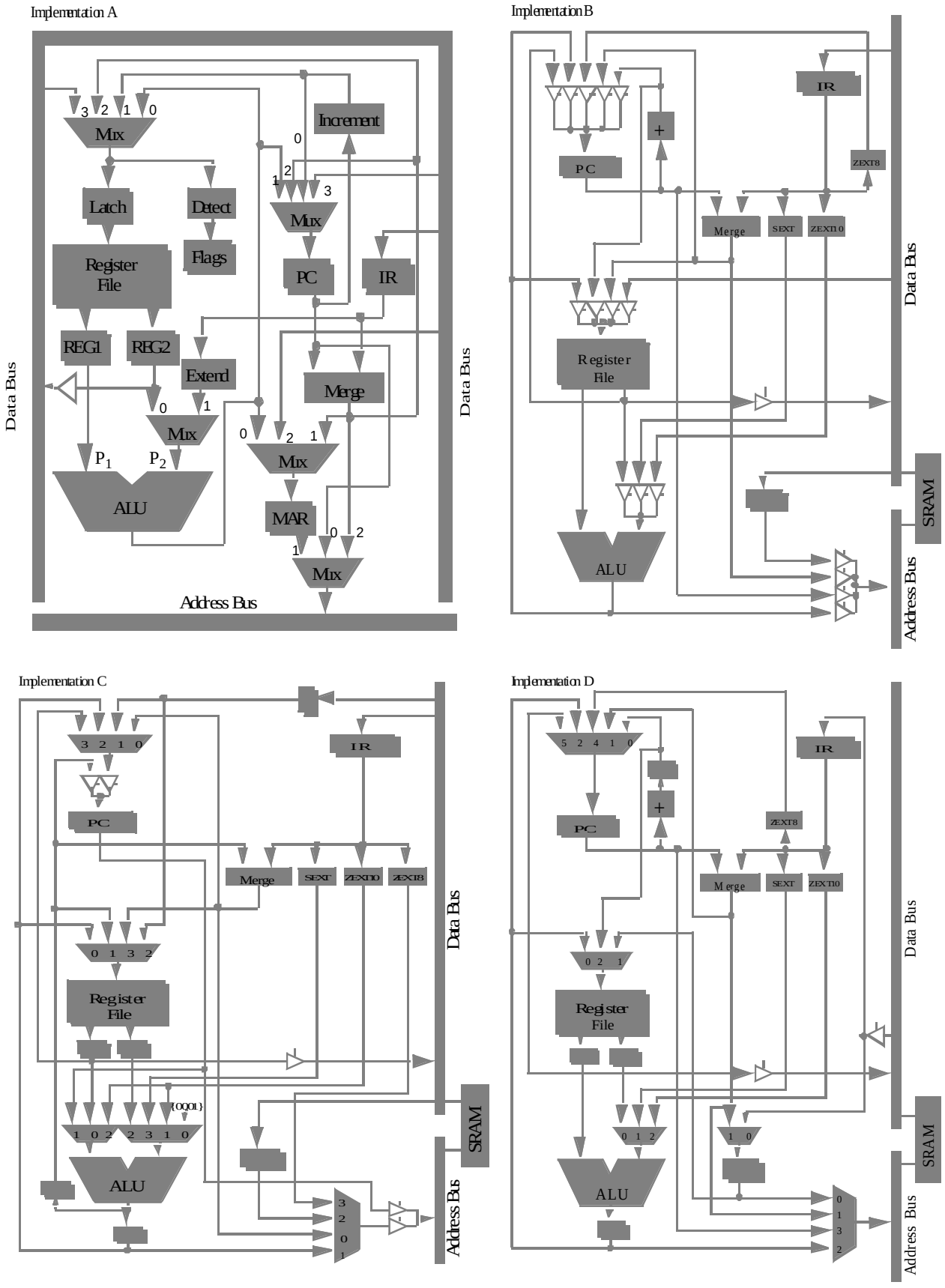


Figure 2.16: Four student implementations of the LC-2

2.2.3 Verification Results

This section demonstrates the REVE system by applying it to Implementation A of the LC-2, shown in Figure 2.16. As mentioned, the design is a gate-level implementation. Since our intent is to work at the RTL level, we hierarchically reverse engineered the datapath to the higher level by hand. Fortunately, since the datapath is designed in a modular format, this process was relatively simple. The full Verilog code for this implementation can be found in Appendix B. For brevity, this section only discusses control analysis of the data-processing instructions. A full analysis of all control steps is found in Appendix C.

REVE's first step is to identify important architectural components. Since we did not have a CAD tool to help us with this identification, we performed this analysis by hand; the details are found in Appendix C.

Step 2 reverse engineers the major data paths in the design. The four instruction classes previously identified were data-processing, branch, load, and store. Using the just-identified architectural components as starting and ending points for data-path analysis, and passing through minor components such as registers and multiplexers, we derived all data paths associated with each basic instruction class.

For data-processing instructions, we examined the inputs to the ALU of Implementation A in Figure 2.16. The left input always comes from REG1, which connects directly to the register file. Since the register file is an architectural component specified by KB0, we stop

a) Data-processing
1. Register file AND Register file
2. Register file AND IR
3. Register file (through first ALU input)
4. Register file (through second ALU input)
5. IR

b) Branch
1. PC
2. ALU
3. PC @ IR
4. Data bus with Address bus being:
   ALU, PC, Data bus, or PC @ IR

c) Load
1. ALU
2. PC
3. PC @ IR
4. Data bus with Address bus being:
   ALU, PC, Data bus, or PC @ IR

d) Store
1. Register file with Address bus being:
   ALU, PC, Data bus, or PC @ IR

Figure 2.17: Operand sources for each instruction type

tracing along P_1, once we reach it. The right ALU input comes from a 2-to-1 multiplexer. The first input to the multiplexer is from REG2, which connects directly to the register file. The second multiplexer input comes from the instruction register through the "Extend" block. Since both the register file and the instruction register are architectural components in KB0, we stop tracing along P_2. From this analysis, we discover five operand sources that can be applied to the ALU, as shown in Figure 2.17a.

Operand sources 3 through 5 are necessary because we have not analyzed the control of the data paths leading to the ALU. If this control specifies that a data path leading to each ALU input is always sensitized, then source combinations 3 through 5 are unnecessary. However, because the control may have don't-care conditions, we must specify these operand source combinations. For example, instructions that only operate on a register operand may assign an X to the control signal of the multiplexer feeding the right ALU input. If we do not include case 3 in our analysis, this don't-care condition will not be detected and the RISA will be incomplete.

The next instruction class is branch. Using the same technique, we look at the inputs to the PC, and see that there are four possible input sources, namely the PC, ALU, PC @ IR (program counter bits concatenated with instruction register bits), and the main data bus. Because we have a data bus input, we must also examine the input sources to the address bus, since the address bus transmits and address pointing to the memory location accessed by the data bus. The address bus has four possible inputs: ALU, PC, data bus, and PC @ IR, leading to the input sources to the PC shown in Figure 2.17b.

The third instruction class is load. Using the same approach, we look for input paths to the register file. It can be seen that these paths exist from the ALU, PC, PC @ IR, and data bus. From the previous address bus analysis, we obtain the input sources to the register file shown in Figure 2.17c.

The final instruction class is store. Again, using the same approach, we find that the only input to the data bus comes from the register file. Combining this with paths for the address bus, we obtain the input source combinations shown in Figure 2.17d.

Step 3 in the REVE algorithm extracts control signals necessary for data path sensitization. In the LC-2 implementation under consideration, control is separate from the datapath, and so is easily analyzed. For brevity, we analyze the control only for the data-processing class of instructions; a full analysis for the remaining three instruction classes can be found in Appendix C. Examples of essential control signals for data-path sensitization are multiplexer control signals, enable signals for registers, and write-enable signals to the memory and register file. All control signals referred to in this section can be found in the control portion of the Verilog description of the LC-2 in Appendix B.

Data (operand) sources	Signal name	Value	Activation cycle
1. Register file AND Register file	clear	1	
	load_reg1_bar	0	*i – 1*
	load_reg2_bar	0	*i – 1*
	sel_alu_mux	0	*i or i – 1*
	sel_rf_mux[1:0]	00	*i*
	clock_bar	1	*i*
	WE	1	*i*
2. Register file AND IR	clear	1	
	load_reg1_bar	0	*i – 1*
	load_ir_bar	0	*i – 1*
	sel_alu_mux	1	*i or i – 1*
	sel_rf_mux[1:0]	00	*i*
	clock_bar	1	*i*
	WE	1	*i*
3. Register file (left ALU input)	clear	1	
	load_reg1_bar	0	*i – 1*
	sel_rf_mux[1:0]	00	*i*
	clock_bar	1	*i*
	WE	1	*i*
4. Register File (right ALU input)	clear	1	
	load_reg2_bar	0	*i – 1*
	sel_alu_mux	0	*i or i – 1*
	sel_rf_mux[1:0]	00	*i*
	clock_bar	1	*i*
	WE	1	*i*
5. IR (right ALU input)	clear	1	
	load_reg2_bar	0	*i – 1*
	sel_alu_mux	0	*i or i – 1*
	sel_rf_mux[1:0]	00	*i*
	clock_bar	1	*i*
	WE	1	*i*

Figure 2.18: Control signals needed to sensitize the data paths of data-processing instructions

Data (operand) sources	Case	Condtion
1. Register file AND Register file	AND	IR[5] = 0
	ADD	IR[5] = 0
2. Register file AND IR	AND	IR[5] = 1
	ADD	IR[5] = 1
3. Register file (left ALU input)	AND	
	ADD	
	NOT	
4. Register File (right ALU input)	AND	IR[5] = 0
	ADD	IR[5] = 0
5. IR (right ALU input)	AND	IR[5] = 1
	ADD	IR[5] = 1

Figure 2.19: Cases where control sets exist for data-processing data paths

Figure 2.18 shows the necessary control signals for sensitization of each of the five data paths shown in Figure 2.17a. Since several data paths pass through registers, it is necessary to identify the proper cycle during which each control signal must be activated,
and the activation cycle column holds this information. In our notation, cycle *i – 1* refers to the cycle immediately preceding cycle *i*.

In step 4, we check for the existence of the control sets shown in Figure 2.18. After scanning the Verilog code (Appendix B), we find that each set falls into a clearly distinguished case. The cases and special conditions necessary for the existence of each control set are shown in Figure 2.19.

Next, in step 5, we determine all the implications of each control set. We use this information in step 6 to generate an unmerged RISA. For brevity, we only show the output of step 6 in Figure 2.20 using our standard instruction template (Figure 2.13).

In addition, these implications indicate that each data-processing operation sets the

Instruction	Assigned Opcode Bits	Functionality
1a. AND	0101 xxxx xx00 0xxx	R(IR[11:9]) ← R(IR[8:6]) AND R(IR[2:0])
1b. ADD	0001 xxxx xx00 0xxx	R(IR[11:9]) ← R(IR[8:6]) + R(IR[2:0])
2a. AND	0101 xxxx xx1x xxxx	R(IR[11:9]) ← R(IR[8:6]) AND SEXT(IR[5:0])
2b. ADD	0001 xxxx xx1x xxxx	R(IR[11:9]) ← R(IR[8:6]) + SEXT(IR[5:0])
3a. AND	0101 xxxx xx00 0xxx	R(IR[11:9]) ← R(IR[8:6]) AND R(IR[2:0])
3b. ADD	0001 xxxx xx00 0xxx	R(IR[11:9]) ← R(IR[8:6]) + R(IR[2:0])
3c. NOT	1001 xxxx xx11 1111	R(IR[11:9]) ← NOT(R(IR[8:6]))
4a. AND	0101 xxxx xx00 0xxx	R(IR[11:9]) ← R(IR[8:6]) AND R(IR[2:0])
4b. ADD	0001 xxxx xx00 0xxx	R(IR[11:9]) ← R(IR[8:6]) + R(IR[2:0])
5a. AND	0101 xxxx xx1x xxxx	R(IR[11:9]) ← R(IR[8:6]) AND SEXT(IR[5:0])
5b. ADD	0001 xxxx xx1x xxxx	R(IR[11:9]} ← R(IR[8:6]) + SEXT(IR[5:0])

Figure 2.20: Unmerged data-processing reverse-engineered instructions for the LC-2

condition codes in the processor. Although not included above for the sake of brevity, the condition codes are variables (by-products of an instruction), and would appear in each instruction's functionality section of the template in Figure 2.13.

Following the same procedure for the remaining three instruction classes, we are able to generate a complete unmerged RISA for the LC-2. This RISA, except for the data-processing instructions in Figure 2.20, is shown in Figure 2.21.

Step 7 merges similar instructions to form a final RISA. As seen from Figure 2.21, several instructions have the same opcode bits and can be merged into a single instruction. For example, the two instances of the TRAP instruction can be merged. The final instruction list after this merging is shown in Figure 2.22.

Instruction	**Assigned Opcode Bits**	**Functionality**
B1. RET	1101 0000 0000 0000	PC ← R(7)
B2. JSRR	1100 x00x xxxx xxxx	PC ← R(IR[8:6]) + ZEXT(IR[5:0])
B3. BR	1000 xxxx xxxx xxxx	PC ← PC[15:9] @ IR[8:0] Reads condition codes
B4. JSR	0100 x00x xxxx xxxx	PC ← PC[15:9] @ IR[8:0]
B5. TRAP	1111 0000 xxxx xxxx	PC ← Mem[ZEXT(IR[7:0])]
L1. AND	0101 xxxx xxxx xxxx	R(IR[11:9]) ← ALU Sets condition codes
L2. ADD	0001 xxxx xxxx xxxx	R(IR[11:9]) ← ALU Sets condition codes
L3. NOT	1001 xxxx xx11 1111	R(IR[11:9]) ← ALU Sets condition codes
L4. JSR	0100 100x xxxx xxxx	R(7) ← PC
L5. JSRR	0100 100x xxxx xxxx	R(7) ← PC
L6. TRAP	1111 0000 xxxx xxxx	R(7) ← PC
L7. LEA	1110 xxxx xxxx xxxx	R(IR[11:9]) ← PC[15:9] @ IR[8:0] Sets condition codes
L8. LDR	0110 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[R(IR[8:6]) + ZEXT(IR[5:0])] Sets condition codes
L9. LD	0010 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[PC[15:9] @ IR[8:0]] Sets condition codes
L10. LDI	1010 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[Mem[PC[15:9] @ IR[8:0]]] Sets condition codes
S1. STR	0111 xxxx xxxx xxxx	Mem[R(IR[8:6]) + ZEXT(IR[5:0])] ← R(IR[11:9])
S2. ST	0011 xxxx xxxx xxxx	Mem[PC[15:9] @ IR[8:0]] ← R(IR[11:9])
S3. STI	1011 xxxx xxxx xxxx	Mem[Mem[PC[15:9] @ IR[8:0]]] ← R(IR[11:9])

Figure 2.21: Unmerged reverse-engineered instructions for the LC-2

Instruction	Assigned Opcode Bits	Functionality
AND*	0101 xxxx xx00 0xxx	If IR[5]=0: R(IR[11:9]) ← R(IR[8:6]) AND R(IR[2:0]) If IR[5]=1: R(IR[11:9]) ← R(IR[8:6]) AND SEXT(IR[5:0])
ADD*	0001 xxxx xx00 0xxx	If IR[5]=0: R(IR[11:9]) ← R(IR[8:6]) + R(IR[2:0]) If IR[5]=1: R(IR[11:9]) ← R(IR[8:6]) + SEXT(IR[5:0])
NOT*	1001 xxxx xx11 1111	R(IR[11:9]) ← NOT(R(IR[8:6]))
RET	1101 0000 0000 0000	PC ← R(7)
JSRR	1100 x00x xxxx xxxx	PC ← R(IR[8:6]) + ZEXT(IR[5:0]) If IR[11]=1: R(7) ← PC
BR	1000 xxxx xxxx xxxx	PC ← PC[15:9] @ IR[8:0] Reads condition codes
JSR	0100 x00x xxxx xxxx	PC ← PC[15:9] @ IR[8:0] If IR[11]=1: R(7) ← PC
TRAP	1111 0000 xxxx xxxx	PC ← Mem[ZEXT(IR[7:0])] R(7) ← PC
LEA*	1110 xxxx xxxx xxxx	R(IR[11:9]) ← PC[15:9] @ IR[8:0]
LDR*	0110 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[R(IR[8:6]) + ZEXT(IR[5:0])]
LD*	0010 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[PC[15:9] @ IR[8:0]]
LDI*	1010 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[Mem[PC[15:9] @ IR[8:0]]]
STR	0111 xxxx xxxx xxxx	Mem[R(IR[8:6]) + ZEXT(IR[5:0])] ← R(IR[11:9])
ST	0011 xxxx xxxx xxxx	Mem[PC[15:9] @ IR[8:0]] ← R(IR[11:9])
STI	1011 xxxx xxxx xxxx	Mem[Mem[PC[15:9] @ IR[8:0]]] ← R(IR[11:9])

* Denotes instructions that set the condition codes

Figure 2.22: Merged RISA for the LC-2

The final step is to compare this RISA with the original ISA. Comparing with the full LC-2 instruction set (Appendix A), we see that each instruction in the ISA matches one in the RISA. The only bug found via this matching process is a discrepancy between PC and PC + 1. In Implementation A, all uses of PC in the RISA correspond to the PC of the current instruction. In the original ISA, all uses of PC refer to the PC of the next instruction, i.e., PC + 1. Because of this, all reverse-engineered instructions that use the current value of PC (JSRR, JSR, TRAP, LEA, LD, LDI, ST, STI,) do not match with their corresponding instructions in the ISA. To aid in debugging, all unmatched instructions in the ISA and RISA could be reported to the designer. The designer then simply has to examine each set of unmatched instructions and determine that the only difference in the designs lies in their use of PC.

Verification of the other LC-2 designs yielded similar results. Implementations B and D display the same PC + 1 bug. REVE also found that Implementations B and D had a TRAP bug. In these two designs the TRAP instruction causes the processor to change program execution to the trap vector, instead of the address pointed to by the trap vector.

CHAPTER 3
PIPELINED PROCESSOR VERIFICATION

Chapter 2 presented the REVE method and applied it to a small processor. While this example is useful for illustrating how our method works, many methods claim success on similar toy examples. However, most of these methods are unable to handle processors with advanced architectural features such as pipelining. This section discusses the difficulties of pipeline verification and shows how REVE can be extended to pipelined machines.

3.1 Method Overview

Pipelining is a method by which multiple instructions execute in a processor at the same time in a manner analogous to production flow on an assembly line [21]. The processor is broken into various segments called pipeline stages, and each stage operates on a different instruction than other stages. An instruction flows through the pipeline from stage to stage until it completes execution and stores any results it may have generated. An efficient pipeline allows one instruction to complete per cycle, while keeping the cycle time low. This increases throughput, which in turn increases the processor's overall performance.

3.1.1 Pipeline Behavior

Unfortunately, pipelining introduces data and control hazards due to interactions between instructions in different pipeline stages [21]. Data hazards occur when two instructions in the pipeline share the same operands. If an instruction in a late stage writes a result that is also an operand being read in an earlier stage, the earlier instruction will have to wait for the later instruction to complete its execution. A simple solution to this data hazard problem is to stall early stages of the pipeline, letting instructions in late pipeline stages complete before instructions in earlier stages continue [22]. However, stalling is detrimental to a processor's performance, since no useful work is done in the stalled stages. An alternative to stalling is forwarding, which creates special paths to send data back to previous stages where operands are needed. These forwarding paths require extra hardware, but they eliminate, or at least reduce, performance loss due to data hazards.

The second pipeline hazard type is a control hazard, which occurs when a branch is taken and the program counter is changed. The decision whether to take a branch is usually made in a late pipeline stage. If the branch is taken, instructions following the branch instruction in the pipeline should not executed. These instructions therefore need to be eliminated, or "squashed".

A pipelined implementation can be viewed as having four modes of operation. The first is normal mode where there are no data hazards, and instructions in the pipeline do not interact with each other. When a pipelined processor operates in normal mode, it is treated as a multi-

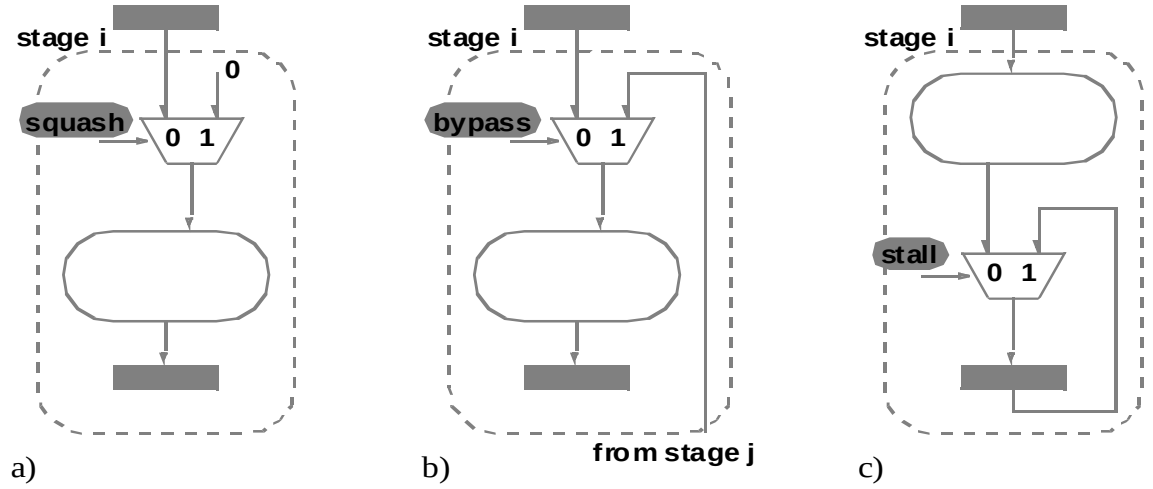


Figure 3.1: Pipeline hazard control mechanisms: a) squashing b) forwarding c) stalling

cycle unpipelined processor, because of the lack of instruction interactions. Therefore, in normal mode, a pipelined design can be reverse engineered using the data-path analysis developed in Chapter 2. The three remaining modes of operation are squashing, forwarding, and stalling, and occur due to the pipeline hazards previously described. Common implementation techniques for these pipeline mechanisms are shown in Figure 3.1. In order to extend REVE to pipelined machines, we need to add knowledge about pipeline hazards into the system.

If we can verify that a pipelined implementation is correct during all four modes of operation, then we assume that we have verified correctness of the pipelining process. In REVE, each mode of operation is verified separately. Steps 1 through 7 in Figure 2.6 verify operation during normal mode, as described in Chapter 2. Step 8 verifies implementation-specific features, such as squashing.

In order to verify operation under normal mode, we must exclude data paths of the kind depicted in Figure 3.1, which are used solely for handling pipeline hazards. Proper implementation of these data paths is verified by our analysis of the remaining three modes of processor operation. The following sections explain how to detect and verify processor operation during squashing, forwarding, and stalling.

3.1.2 Squashing

Squashing results from a pipeline hazard related to control flow and is usually due to a branch operation. Instructions in a pipeline following a taken branch must be invalidated to avoid writing back erroneous results to components such as the register file. There are two common ways to implement squashing. The first is via a signal sent to the reset logic for pipeline registers, clearing information in preceding pipeline stages. The second way, shown in Figure 3.1a, uses a multiplexer to squash register or memory write-back bits in the pipeline. During normal-mode operation, the squash signal to the multiplexer is 0, and the multiplexer propagates the bits. In squashing mode, the squash signal is a 1, and the bits are cleared.

In order to verify the correct implementation of squashing, we must verify that squashing occurs if and only if a squash condition is encountered. For now, we assume that squashes only occur due to a taken branch or system reset. Since squashing requires invalidation of register

```
Main()
If Instruction == Branch then
        Check_Squash_properties(all)
If Condition == Reset then
        Check_Squash_properties(all)
If Instruction_type == Conditional Fail then
        Check_Squash_properties(current)
Check_accidental_squashes()
end

Check_Squash_properties(Stages):
    Identify write-back bits
    If (Stages==all)
        Check for invalidation of write-back bits in all stages
    Else
        Check invalidation of write-back bits in current stage
    End

Check_accidental_squashes
    Check that logic equations never squash unless
        (Instruction == Branch) OR
        (Condition == Reset) OR
        Instruction_type == Conditional Fail)
    End
```

Figure 3.2: ISA information to account for squashing

and memory write-back data, the only necessary knowledge is the location of the corresponding write-enable bits in the pipeline. Since we know from our analysis in Chapter 2 the final location of these bits (in the register file and memory interface), we can automatically trace their propagation through the pipeline. The RISA generated by REVE during verification of the normal mode includes a complete set of branch operations, and it is simple to verify whether each of these branch operations invalidates all write-enable bits. In addition, we need to make sure that squashing can never accidentally occur by verifying that the write-enable bits are never invalidated by anything but a squash condition. This involves checking the logic equations for the write-enable bits and ensuring that only a branch or a reset can produce a squash.

So far, we have assumed that only branches or system resets cause squashing conditions; however, in modern processors, other conditions occur in which squashing is necessary. For example, an implementation with a branch predictor must squash on a mispredicted branch instead of a taken branch. Our original squashing assumption implicitly assumes a branch-not-taken prediction policy, which is frequently implemented in simpler machines. In addition, as we will see in Chapter 4, some ISAs include non-branch instructions that are conditionally executed. In this case, we need to verify that each conditional instruction failure squashes its own write-enable bits in the pipeline. Figure 3.2 shows an extended processor specification containing the necessary information for this type of squashing. The first two if statements check the original branch-and-reset conditions for squashing, while the third if statement is added to cover the conditional instruction failure just described.

3.1.3 Forwarding

Forwarding occurs when there is a data hazard involving operands in two different pipeline stages. Typical forwarding paths exist between the output of execution units such as the ALU to the location where operands are read from the register file. In order to identify forwarding paths, we need to know the ordering of pipeline stages to determine the direction of normal instruction flow. This ordering information needs to be added to the knowledge base; otherwise, long instructions that loop back through the pipeline may be confused with instructions using forwarding paths. An example of a long instruction is multiply, which is sometimes implemented by a series of addition operations, requiring loops that reuse CPU components such as adders. These datapath loops must be distinguished from the feedback paths used for forwarding. One way is to examine the opcodes detected during data-path reverse engineering. If the opcode remains the same for two or more iterations, then a long instruction is detected. If the opcode is not guaranteed to be the same, then we may have a case of forwarding. Not distinguishing between forwarding paths and long instruction paths may cause erroneous detection of long instructions that are really forwarding chains of short instructions, such as an ADD followed by a SUB followed by an AND.

Verification of forwarding requires the identification of all outputs of functional units used during write-back, and all locations where operands are read or used. For correct pipeline implementation, we must verify that paths exist between each output and each operand reading location, which involves a straightforward search of the HDL code. Once we verify the existence of forwarding paths, we must verify that each path is sensitized only during a data hazard condition. This involves examining the logic that controls forwarding multiplexers, and ensuring that each forwarding control signal is only set during a register conflict with a later pipeline stage, e.g. during a data hazard. If we discover that the implementation is missing a forwarding path, or if a forwarding path is sensitized at the wrong time, we output this information to the designer.

3.1.4 Stalling

Stalling is necessary when operations in early pipeline stages must pause in order to let instructions in later stages be completed. Stalling is usually indicated when instructions hold data in a pipeline register for more than one cycle, as shown in Figure 3.1c. Examining the inputs to each pipeline register in the data path identifies these loops, since if a register is capable of being stalled, one of its input paths will trace back to its output, after passing through one or more multiplexers. Our analysis does not account for excessive stalling, since although undesirable, this condition is a performance characteristic and not a functional bug.

In order to prove functional correctness of an implementation of stalling, we assume that stalling only occurs during two events: a data-hazard condition, or while waiting for slow components, such as memory to return data. First, we verify that stalls properly occur during a data-hazard condition. This requires coordination with the forwarding verification described in the previous section, since either forwarding or stalling can be used to avoid data hazards. Therefore, we must verify that any register dependencies not handled by the forwarding are handled by stalling. This involves another examination of the logic equations for control signals to the forwarding muxes in order to determine any conditions during which forwarding paths are not sensitized. Second, we must verify cases when individual pipeline stages, such as memory, require more than one cycle and stall the pipeline. This verification step requires knowledge of

memory wait signals. Once these signals are identified, it is easy to examine the stall logic (logic equations controlling the multiplexers used to implement stalling) and to verify that the pipeline stalls whenever a wait signal is turned on for multiple cycles. Third, we must verify that a stall in a late pipeline stage always stalls earlier pipeline stages; this is necessary to avoid having instructions collide in the pipeline.

In addition to data hazards due to register dependencies, memory dependency hazards may also exist. Until now, we have assumed a simple pipeline that only has one port to memory. However, if more than one pipeline stage includes a memory port, we must check for memory dependencies by examining the address of each memory port. We must verify that a stall occurs if there is an address conflict between memory ports during a write operation in a late pipeline stage and a read operation in an early pipeline stage.

3.2 Pipelined Version of LC-2

In order to demonstrate verification of a pipelined machine, REVE is applied next to a set of pipelined LC-2 designs. The student designers of the unpipelined LC-2 models examined in Chapter 2 also created the pipelined models discussed in this section.

3.2.1 Sample Implementations

The pipelined designs that were verified with REVE are shown in Figure 3.3 and Figure 3.4. The major functional units are identical to those in the corresponding unpipelined cases. Each model employs a 5-stage pipeline consisting of fetch, decode, execute, memory, and write-back stages. Stalling paths are the feedback paths around each register; all forwarding paths are also shown in the diagrams.

3.2.2 Verification Results

In this section, we verify pipelined implementation B in Figure 3.3. REVE must verify operation in normal mode, and under the three pipeline conditions: squashing, forwarding, and stalling. Normal-mode verification is simply steps 1 through 7 in Figure 2.6. For these steps, all stalling and forwarding paths, which are easily seen in Figure 3.3, must be detected and checked. Since this analysis is nearly identical to that of Section 2.2, we skip ahead to the result of step 7, and show the final RISA in Figure 3.5. This implementation does not exhibit the program counter bug detected in Chapter 2, as we can see by the occurrence of PC + 1 rather than PC in the RISA.

Squashing is the first pipeline operation mode that we verify. A correct implementation of squashing requires invalidation of register or memory write-enable bits, which occurs if and only if a taken branch or system reset occurs. Implementation B activates a signal, do_jump, on a taken branch, which is used to squash pipeline registers. A Verilog fragment showing the use of this signal for the squashing of the ir_me register shown in Figure 3.3 appears below.

```
assign #1 ir_me_w =
    (reset ? 16'h0000 :
        (reg_stall_me ? ir_me :
                (reg_stall_ex | do_jump ? 'NOP : ir_ex)))
```

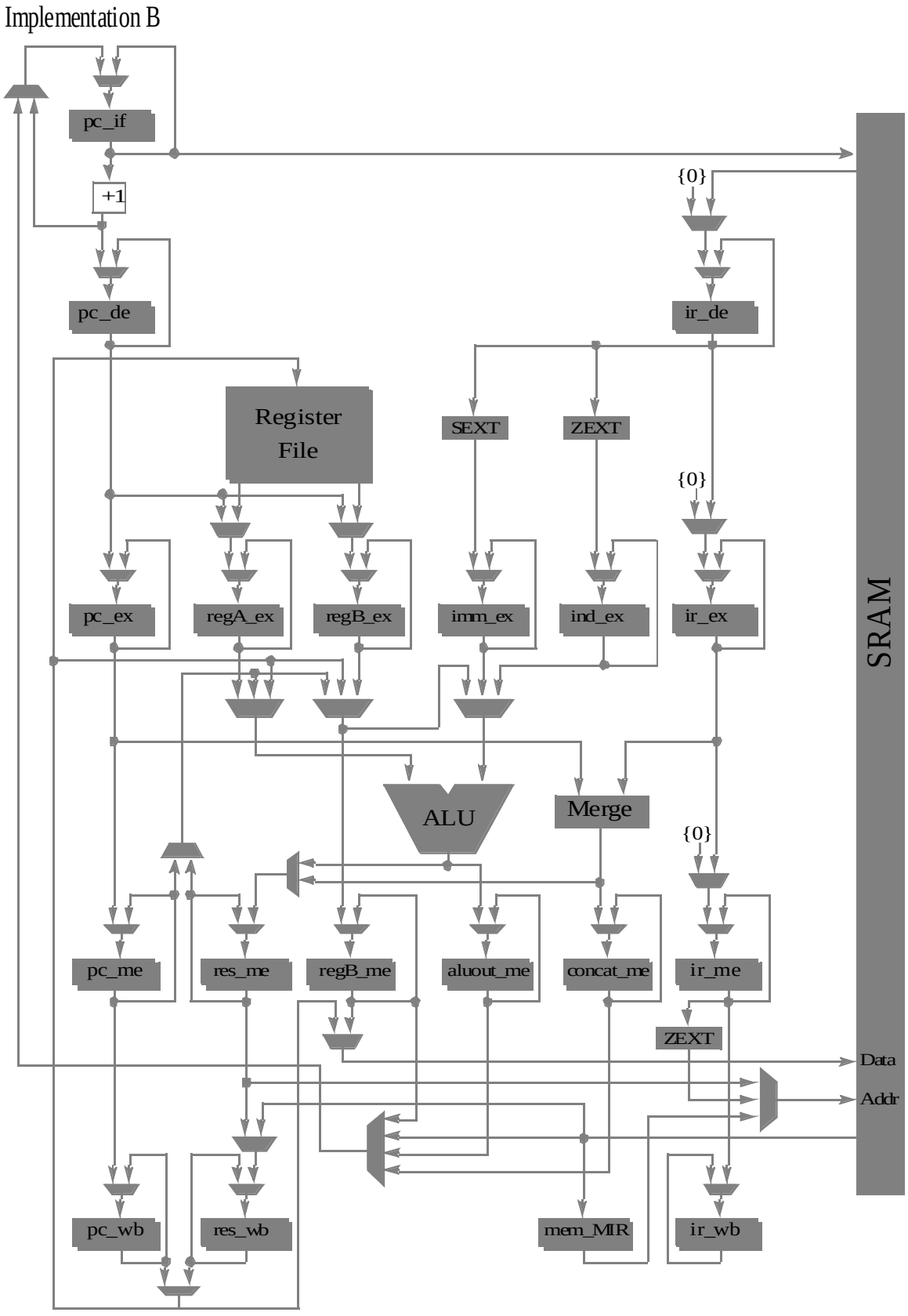


Figure 3.3: Pipelined implementation B of the LC-2

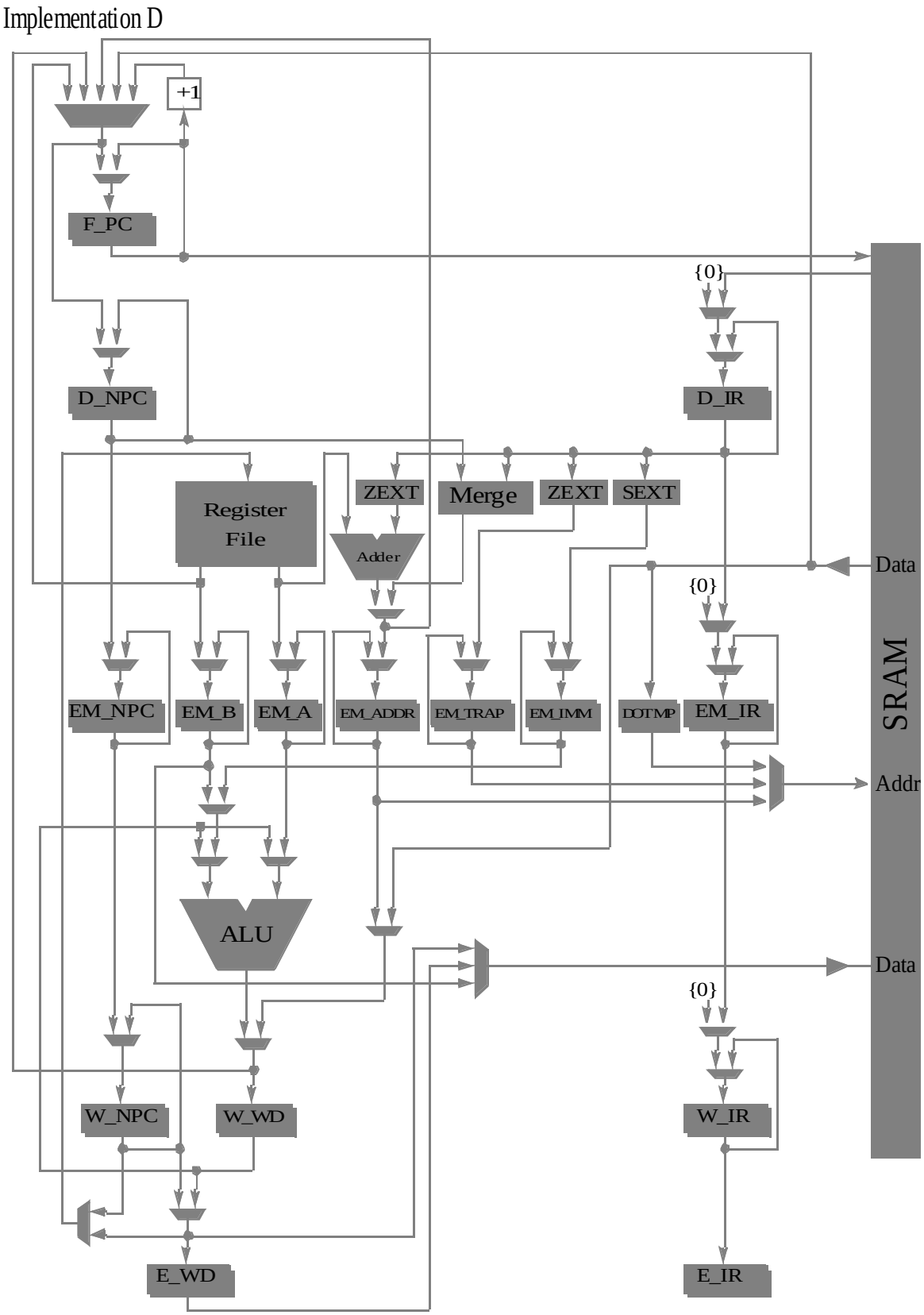


Figure 3.4: Pipelined implementation D of the LC-2

Instruction	Assigned Opcode Bits	Functionality
AND*	0101 xxxx xx00 0xxx	If IR[5]=0: R(IR[11:9]) ← R(IR[8:6]) AND R(IR[2:0]) If IR[5]=1: R(IR[11:9]) ← R(IR[8:6]) AND SEXT(IR[5:0])
ADD*	0001 xxxx xx00 0xxx	If IR[5]=0: R(IR[11:9]) ← R(IR[8:6]) + R(IR[2:0]) If IR[5]=1: R(IR[11:9]) ← R(IR[8:6]) + SEXT(IR[5:0])
NOT*	1001 xxxx xx11 1111	R(IR[11:9]) ← NOT(R(IR[8:6]))
RET	1101 0000 0000 0000	PC ← R(7)
JSRR	1100 x00x xxxx xxxx	PC ← R(IR[8:6]) + ZEXT(IR[5:0]) If IR[11]=1: R(7)← (PC+1)
BR	1000 xxxx xxxx xxxx	PC ← (PC+1)[15:9] @ IR[8:0] Reads condition codes
JSR	0100 x00x xxxx xxxx	PC ← (PC+1)[15:9] @ IR[8:0] If IR[11]=1: R(7) ← (PC+1)
TRAP	1111 0000 xxxx xxxx	PC ← Mem[ZEXT(IR[7:0])] R(7) ← (PC+1)
LEA*	1110 xxxx xxxx xxxx	R(IR[11:9]) ← (PC+1)[15:9] @ IR[8:0]
LDR*	0110 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[R(IR[8:6]) + ZEXT(IR[5:0])]
LD*	0010 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[(PC+1)[15:9] @ IR[8:0]]
LDI*	1010 xxxx xxxx xxxx	R(IR[11:9]) ← Mem[Mem[(PC+1)[15:9] @ IR[8:0]]]
STR	0111 xxxx xxxx xxxx	Mem[R(IR[8:6]) + ZEXT(IR[5:0])] ← R(IR[11:9])
ST	0011 xxxx xxxx xxxx	Mem[(PC+1)[15:9] @ IR[8:0]] ← R(IR[11:9])
STI	1011 xxxx xxxx xxxx	Mem[Mem[(PC+1)[15:9] @ IR[8:0]]] ← R(IR[11:9])

* Denotes instructions that set the condition codes

Figure 3.5: RISA for pipelined implementation B

As seen from this fragment, the do_jump signal causes the instruction in the ir_me register to be replaced with a NOP, e.g. an all-0 instruction, which erases any associated write-enable bits. From the remaining Verilog code, we see that each pipeline register shown in Figure 3.3 uses this squashing signal to invalidate pipeline data whenever a branch is taken. The other squashing condition is system reset, and we can see from the above code that a reset also squashes a pipeline register's data.

In addition, we must ensure that squashing can never occur accidentally. In the above code fragment, a NOP is also issued when reg_stall_me = 0 AND reg_stall_ex = 1. This is an artifact of the stalling implementation and is not a bug, since when the execute stage is stalled while the memory stage is not, the memory stage should not be processing data.

The second pipeline mode to be checked is forwarding. Verification of forwarding involves detecting functional units capable of write-back. Tracing from the write input of the register file, we find that four components are capable of write-back: the PC, Data bus, ALU, and Merge (PC @ IR). Register operands are not used until the execute stage. Because this implementation has two pipeline stages after execute, there are fourteen necessary forwarding paths, as shown in Figure 3.6.

As we can see from Figure 3.3, all of these paths exist in implementation B. Next, we verify that each path is sensitized if and only if a data-hazard condition occurs. A fragment of Verilog code corresponding to the forwarding multiplexer for the RegA value follows.

1. PC – memory stage to RegA
2. PC – memory stage to RegB
3. PC – write-back stage to RegA
4. PC – write-back stage to RegB
5. ALU result – memory stage to RegA
6. ALU result – memory stage to RegB
7. ALU result – write-back stage to RegA
8. ALU result – write-back stage to RegB
9. Merge – memory stage to RegA
10. Merge – memory stage to RegB
11. Merge – write-back stage to RegA
12. Merge – write-back stage to RegB
13. Data bus – write-back stage to RegA
14. Data bus – write-back stage to RegB

Figure 3.6: Necessary forwarding paths for implementation B

```
assign #1 ex_me_will_write =
    ((me_op==‘ADD) | (me_op==‘AND) | (me_op==‘LD)  | (me_op==‘LDI) |
     (me_op==‘LDR) | (me_op==‘LEA) | (me_op==‘NOT) | (me_op==‘TRAP) |
     ((me_op==‘JSR) & (ir_me[11]==1'b1)) |      // true JSR  (not JMP)
     ((me_op==‘JSRR) & (ir_me[11]==1'b1)))      // true JSRR (not JMPR)
     ? 1'b1 : 1'b0;
assign #1 ex_fwd_a_1 =
    ((ir_ex[‘BaseR]==ir_me[‘DR]) & (ex_me_will_write)) ? 1'b1: 1'b0;
assign #1 ex_fwd_a_2 =
    ((ir_ex[‘BaseR]==ir_wb[‘DR]) & (wb_we)) ? 1'b1: 1'b0;
```

The ex_fwd_a_1 signal handles all memory cases, and ex_fwd_a_2 handles all write-back cases. As we can see, each signal is turned on only when a data-hazard condition exists, i.e., when the memory or write-back stage contains data to be written to the register file, and the destination register conflicts with an operand register from the execute stage. Therefore, each forwarding path is sensitized at the correct time, and forwarding is correctly implemented.

The final pipeline operation mode to be considered is stalling. Verification of stalling involves checking three properties, as described in Section 3.1.4. These are:

1. Data hazard conditions not covered by forwarding are covered by stalling
2. Stages requiring extra execution time stall preceding stages
3. Stalls in late pipeline stages also stall early stages

The first stalling check is for data-hazard conditions not covered by forwarding. From the above forwarding analysis, we concluded that forwarding was implemented correctly for all data-hazard conditions, thus removing the need for additional stalling logic. However, while examining the Verilog code, we discovered that the designer was overly cautious in his stalling implementation, and stalled anytime a data-hazard condition occurred. A fragment of his code for this is shown below.

```
// stall logic
// this is a bit conservative. a stall *may* be necessary under these
```

```
// conditions, but not always
  assign #1 req_stall_ex =
    (((me_op==‘LD) | (me_op==‘LDI) | (me_op==‘LDR)) &
     ((ir_ex[‘BaseR]==ir_me[‘DR]) | (ex_b_sReg==ir_me[‘DR]))) ? 1'b1 : 1'b0;
```

As we can see, the stall is unnecessary due to the implemented forwarding paths, and the presence of this stall actually precludes the use of forwarding paths for data hazards between the execute and memory stages. Therefore, although the implementation of stalling for data hazards was unnecessary, the first stalling property is implemented correctly.

The second property ensures that stages requiring extra execution time stall the pipeline. As described in Section 3.1.4, this requires knowledge of pipeline wait signals. The only wait signal in implementation B is the mem_MIRV signal, which is used in the memory stage of the pipeline by LDI or STI instructions. This stage requires an additional cycle since these instructions access main memory twice. The following code fragment shows the stall condition implemented for these two instructions.

```
// stalling for LDI/STI
assign #1 req_stall_me =
  (((me_op==‘LDI)|(me_op==‘STI)) & (~ mem_MIRV)) ? 1'b1 : 1'b0;
```

Whenever the mem_MIRV signal is 0 with an LDI or STI instruction, a stall occurs. From the control logic for mem_MIRV, we see that detection of LDI or STI sets this signal to 0. During the second cycle, the bit is inverted, and the stall condition is removed. Therefore, this second pipeline property is correctly implemented.

The third stalling property requires detection of all stall signals in the pipeline. From the Verilog code, we see that only two stall signals, reg_stall_ex and reg_stall_me, exist and are turned on by stall conditions in the execute and memory stages, respectively. Further examination reveals that if either of these two signals is turned on, the fetch and decode stages are stalled. In addition, if a stall condition occurs in the memory stage, the execute stage is unconditionally stalled. Therefore, the third property is verified, since stalls in late pipeline stages also stall all earlier stages.

CHAPTER 4
CASE STUDY: ARM 7

Chapters 2 and 3 presented REVE and used it to verify both unpipelined and pipelined versions of the small LC-2 processor. To further evaluate REVE, we use it next to verify an implementation of the much larger ARM 7, a commercial processor, which an undergraduate student at the University of Michigan implemented as a directed study project. Although we would have liked to verify a full commercial implementation, we were unable to obtain one.

4.1 Processor ISA

The ARM 7 is a 32-bit RISC microprocessor that is part of the ARM (Advanced RISC Machines) processor family [23]. It is widely used in commercial applications, primarily as an embedded CPU. The ARM 7 instruction set supports 11 instruction types: two data-processing instructions, three data-transfer instructions, three instructions to control instruction flow and execution privilege, and three instructions to control external coprocessors. The ARM 7 has a 32-bit address bus, and supports data types consisting of bytes or words, that is, 32 bits aligned to

Instruction	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Data Processing PSR Transfer	Cond				00		I	Opcode				S	Rn				Rd				Operand 2											
Multiply	Cond				000000						A	S	Rd				Rn				Rs				1001				Rm			
Single Data Swap	Cond				00010				B		00		Rn				Rd				0000				1001				Rm			
Single Data Transfer	Cond				01		I	P	U	B	W	L	Rn				Rd				Offset											
Undefined	Cond				011			Xxxxxxxxxxxxxxxxxxx																					1	xxxx			
Block Data Transfer	Cond				100			P	U	S	W	L	Rn				Register List															
Branch	Cond				101			L	Offset																							
Coproc Data Transfer	Cond				110			P	U	N	W	L	Rn				CRd				CP#				Offset							
Coproc Data Operation	Cond				1110				CP Opc				CRn				CRd				CP#				CP			0	CRm			
Coproc Reg. Transfer	Cond				1110				CP Opc			L	CRn				Rd				CP#				CP			1	CRm			
Software Interrupt	Cond				1111			Ignored by processor																								

Cond: Condition field	I: Immediate operand	U: Up/Down bit	CRn: Coprocessor operand register
Rn: Operand register	S: Set condition codes	W: Write-back bit	CRd: Coprocessor destionation register
Rd: Destination register	A: Accumulate	L: Load/Store bit (except for Branch)	Crm: Coprocessor operand register
Rs: Operand register	B: Byte/Word bit	N: Transfer length	CP#: Coprocessor number
Rm: Operand register	P: Pre/Post indexing bit	CP Opc:Coprocessor operation code	CP: Coprocessor information

Figure 4.1: Summary of the ARM 7's instruction set

4-byte boundaries, for load and store instructions. It has six modes of operation: user, FIQ, IRQ, supervisor, abort, and undefined. Most applications run in user mode, as the other modes are privileged modes primarily intended for interrupts or other exceptions. The ARM 7 has 37 registers consisting of 31 general-purpose registers and 6 status registers. However, only 16 general registers (R0:R15) and several status registers are visible to the programmer at a given time. A unique feature of the ARM is that the program counter PC is stored in register R15. Since instructions are of the word data type, the last 2 bits of the PC are read as zeros when fetching an instruction. Register R14 is used by subroutine link (call) instructions to store the return value of the PC. A summary of the instruction set is shown in Figure 4.1.

The CPSR (Current Program Status Register) is a register which holds condition codes similar to those in the LC-2. The four conditions are negative / less than (N), zero (Z), carry / borrow / extend (C), and overflow (V). Data-processing instructions may set the condition codes as a result of arithmetic or logical operations. In addition, every instruction in the processor is conditionally executed and reads the condition codes. A 4-bit condition code field is prefixed to each instruction, and the instruction's execution is conditional on the value of the CPSR. This conditional execution improves program performance in if-then-else operations, as it eliminates the explicit branch instruction usually required for these operations.

4.2 Implementation Example

The ARM 7 implementation that we verified with REVE was designed by K. Sit at the University of Michigan in 1996 [24]. Although incomplete, a substantial portion of the processor was implemented. Of the 11 instruction classes, 6 were implemented: two data-processing instructions (data-processing and multiply), three data-transfer instructions (single data swap, single data transfer, and block data transfer), and one flow-control instruction (branch). A detailed description of these instructions can be found in Appendix D. The four condition code flags were included, but user mode was the only mode implemented.

A three-stage pipeline with fetch, decode, and execute stages was also implemented, allowing three instructions to be in process at any time. The execute stage is responsible for tasks such as register read, operand shift, ALU operations, and result write-back. Because both register read and register write-back are included in the execute stage, data hazards do not exist, and forwarding paths were not required in this implementation. However, the processor does exhibit squashing and stalling, the two other pipeline properties. Squashing occurs in the fetch and decode stages of the pipeline if a branch is taken, and stalling occurs in the fetch and decode stages when an instruction spends a long time in the execute stage, e.g., in the case of multiply. If an instruction's condition test fails, the instruction is treated as a no-operation and passes through the pipeline without affecting architectural state.

A block diagram of the ARM 7 implementation under consideration is shown in Figure 4.2. The internal structure of its control unit is shown in Figure 4.3. A PLA generates primary control signals for each instruction. The clocking scheme in this implementation is positive-edge triggering for all clocked datapath elements, and negative-edge triggering for state registers in the control unit.

The instruction pipeline register unit contains three registers holding an instruction for each pipeline stage. This unit sends the instruction currently in the execute stage to the B bus, and the instruction currently in the decode stage to the control unit. Stalling is implemented via a

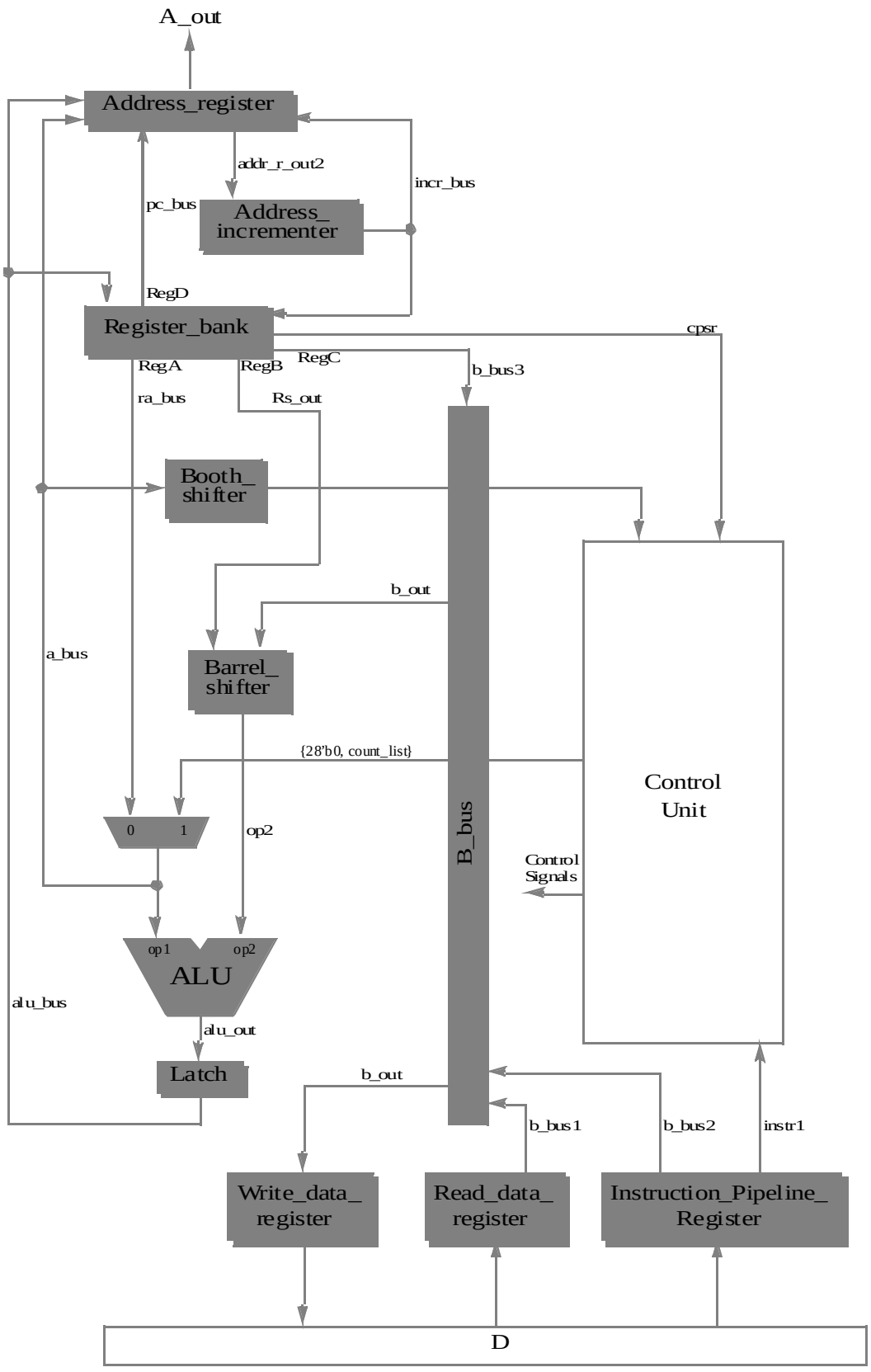


Figure 4.2: Implementation of the ARM 7 processor

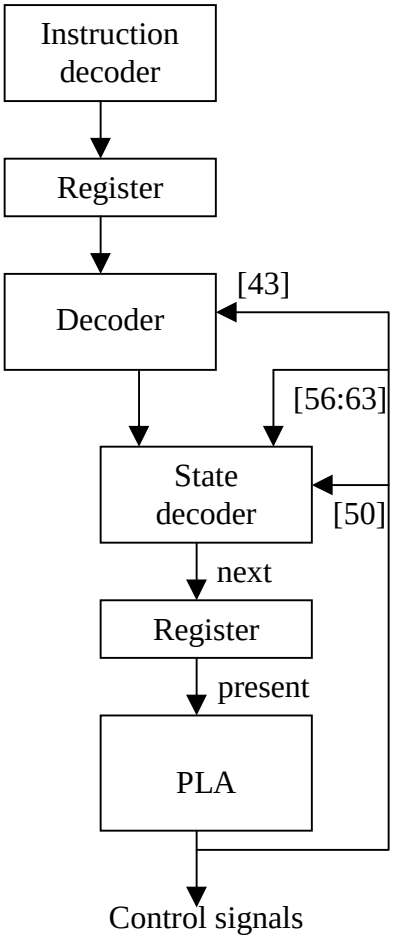


Figure 4.3: Control unit of the ARM 7 implementation

shift-enable signal. When shift-enable is asserted, instructions flow to the next stage. While shift-enable is turned off, each instruction register stalls and holds its state.

The complete Verilog HDL code for this processor is found in Appendix E.

4.3 Verification Process

This section applies REVE to the ARM 7 implementation in Figure 4.2. A full analysis is presented for the first two verification steps of Figure 2.6. However, since the control analysis beginning with step 3 becomes quite lengthy, we limit our discussion to branch instructions for steps 3 through 7. A full analysis of steps 3 through 7 for all instructions in the ARM 7 is in Appendix F.

Step 1 of REVE identifies architectural components specified by the knowledge base KB0 (Figure 2.2). We do not have an automated tool for this task, and we quickly discovered that our initial manual analysis misidentified the program counter. From the block diagram in Figure 4.2, one might guess that *Address_register* is the program counter PC. However, as mentioned earlier, the PC is really register R15 of the register file, and *Address_register* is really an interface to the address bus. This example demonstrates the importance of manually

prompting the user to ensure proper identification in step 1. The final identifications generated by step 1 are shown in Appendix F.

Step 2 in Figure 2.6 reverse engineers all data paths in the processor. Again, they are

Data-processing operations

1a1. Reg <- RegA, [RegB (shift) Data Bus], Address Bus <- ALU
1a2. Reg <- RegA, [RegB (shift) Data Bus], Address Bus <- RegD
1a3. Reg <- RegA, [RegB (shift) Data Bus], Address Bus <- Address Bus
1a4. Reg <- RegA, [RegB (shift) Data Bus], Address Bus <- RegA
1a5. Reg <- RegA, [RegB (shift) Data Bus], Address Bus <- IR
1b. Reg <- RegA, [RegB (shift) IR]
1c. Reg <- RegA, [RegB (shift) RegC]
2a1. Reg <- RegA, [RegB (shift) Data Bus], Address Bus <- ALU
2a2. Reg <- IR, [RegB (shift) Data Bus], Address Bus <- RegD
2a3. Reg <- IR, [RegB (shift) Data Bus], Address Bus <- Address Bus
2a4. Reg <- IR, [RegB (shift) Data Bus], Address Bus <- RegA
2a5. Reg <- IR, [RegB (shift) Data Bus], Address Bus <- IR
2b. Reg <- IR, [RegB (shift) IR]
2c. Reg <- IR, [RegB (shift) RegC]

Branch operations (Very few operations because PC is embedded in the register file)

1a. PC <- Address Bus, Address Bus <- ALU
1b. PC <- Address Bus, Address Bus <- RegD
1c. PC <- Address Bus, Address Bus <- Address Bus
1d. PC <- Address Bus, Address Bus <- RegA
1e. PC <- Address Bus, Address Bus <- IR
2. PC <- ALU

Load operations (Very few operations because most register writes go through the ALU)

1a. Reg <- Address Bus, Address Bus <- ALU
1b. Reg <- Address Bus, Address Bus <- RegD
1c. Reg <- Address Bus, Address Bus <- Address Bus
1d. Reg <- Address Bus, Address Bus <- RegA
1e. Reg <- Address Bus, Address Bus <- IR
2. Reg <- ALU

Store operations

1a. Data Bus <- Data Bus, Address Bus <- ALU
1b. Data Bus <- Data Bus, Address Bus <- RegD
1c. Data Bus <- Data Bus, Address Bus <- Address Bus
1d. Data Bus <- Data Bus, Address Bus <- RegA
1e. Data Bus <- Data Bus, Address Bus <- IR
2a. Data Bus <- IR, Address Bus <- ALU
2b. Data Bus <- IR, Address Bus <- RegD
2c. Data Bus <- IR, Address Bus <- Address Bus
2d. Data Bus <- IR, Address Bus <- RegA
2e. Data Bus <- IR, Address Bus <- IR
3a. Data Bus <- RegC, Address Bus <- ALU
3b. Data Bus <- RegC, Address Bus <- RegD
3c. Data Bus <- RegC, Address Bus <- Address Bus
3d. Data Bus <- RegC, Address Bus <- RegA
3e. Data Bus <- RegC, Address Bus <- IR

Figure 4.4: Operations extracted by step 2 of REVE for the ARM 7

partitioned according to the instruction types data-processing, branch, load, and store. In REVE, the tracing of an instruction's data path is stopped when we reach a functional unit specified by KB0, such as the PC or ALU. The Ones_count(IR) notation refers to the number of 1s in instruction register bits 15 through 0. The complete list of reverse-engineered data paths is shown in Figure 4.4.

The third verification step, step 3 determines appropriate control signals for sensitizing each data path. At this point we focus our analysis on branch instructions. From Figure 4.4 we observe that the branch instruction class involves six data paths. The first five paths originate from the address register, while the last path comes from the ALU. Because the PC is hidden within the register file, branch instructions can be found embedded in the data-processing instructions that also write to the register file. Therefore, to identify any hidden branch instructions for analysis in this section, we searched through the data-processing RISA looking for instructions that modified R15.

The full data-processing RISA is found in Appendix F, and includes branch operations originating from the data-processing paths DP1b and DP1c shown in Figure 4.4. The list of necessary control signals for each branch data path determined by step 3 is shown in Figure 4.5. The control signals' names are taken from the implementation's control.v module. In order to save space, we group control signals by the cycle in which they are activated. We note that two sets of control signals may sensitize the DP1b and DP1c data paths. This is because the master clock does not clock the latch on the ALU output, so we must treat the open latch and closed latch as separate cases.

Step 4 of REVE determines whether each set of control signals exists in the implementation. Figure 4.6 shows the set of control states necessary for sensitization of

Reverse Engineered Instruction	**Conditions**
Branch1a	*i:* rb_w_en=1 *i-1*: incr_en=1 *i-2*: addr_r_w_en=1, addr_r_w_sel=00 *i-2* or *i-3*: l_en=1
Branch1b	*i:* rb_w_en=1 *i-1*: incr_en=1 *i-2*: addr_r_w_en=1, addr_r_w_sel=01
Branch1c	*i:* rb_w_en=1 *i-1*: incr_en=1 *i-2*: addr_r_w_en=1, addr_r_w_sel=10
Branch1d	*i:* rb_w_en=1 *i-1*: incr_en=1 *i-2*: addr_r_w_en=1, addr_r_w_sel=11, mux2_32_sel=0
Branch1e	*i:* rb_w_en=1 *i-1*: incr_en=1P2: addr_r_w_en=1, addr_r_w_sel=11, mux2_32_sel=1
DP1b. Reg <- RegA, [RegB (shift) IR]	*i:* mux2_32_sel=0 && l_en=1 && rb_w_en1=1 *i-1*: B_in_en1=0, B_in_en2=1, B_in_en3=0 OR ---- *i:* l_en=0 && rb_w_en1=1 *i-1*: mux2_32_sel=0 && l_en=1 B_in_en1=0, B_in_en2=1, B_in_en3=0
DP1c. Reg <- RegA, [RegB (shift) RegC]	*i:* mux2_32_sel=0 && l_en=1 && rb_w_en1=1 *i-1*: B_in_en1=0, B_in_en2=0, B_in_en3=1 OR ---- *i:* l_en=0 && rb_w_en1=1 *i-1*: mux2_32_sel=0 && l_en=1 B_in_en1=0, B_in_en2=0, B_in_en3=1

Figure 4.5: Control signals extracted for branch instructions in the ARM 7

Branch1a:	2->3, 1->2, 17->1, 16->17 19->15, 18->19, 17->18, 16->17
Branch1b:	No exercisable data paths
Branch1c:	No exercisable data paths
Branch1d:	No exercisable data paths
Branch1e:	No exercisable data paths
DP1b:	18->19, 17->18
DP1c:	20->4, 15->20

Figure 4.6: Control sets present for branch data path sensitization in the ARM 7

each branch data path. The numbers correspond to internal control states labeled in the PLA. Therefore, the first entry in Figure 4.6 signifies that the Branch1a data path is sensitized when a transition from state 2 to state 3 occurs in cycle *i*, a transition from state 1 to state 2 occurs in cycle *i – 1*, and so on.

1. 2->3, 1->2, 17->1, 16->17*

Assigned Opcode Bits (IR[31:24])	Functionality
xxxx1010	R(15) ← R(15) + shifter_output + 4 shifter_output = if (IR[23]=1) {6'b1, IR[23:0], 2'b0} else {6'b0, IR[23:0], 2'b0} IR[31:28] = Condition codes (tracing back to previous instruction)

2. 19->15, 18->19, 17->18, 16->17*

Assigned Opcode Bits (IR[31:24])	Functionality
xxxx1011	R(15) ← R(15) + shifter_output + 4 Shifter_output = if (IR[23]=1) {6'b1, IR[23:0], 2'b0} else {6'b0, IR[23:0], 2'b0} IR[31:28] = Condition codes (tracing back to previous instruction)

3. 18->19, 17->18*

Assigned Opcode Bits (IR[31:24])	Functionality
xxxx1011	R(14) ← R(15) IR[31:28] = Condition codes (tracing back to previous instruction)

4. 20->4, 15->20*

Assigned Opcode Bits (IR[31:24])	Functionality
xxxx1011	R(14) ← R(14) - 8 (BUG! should be - 4) IR[31:28] = Condition codes (tracing back to previous instruction)

* Condition codes must be true to execute instruction

Figure 4.7: Unmerged RISA for branch instructions in the ARM 7

Instruction	Assigned Opcode Bits (IR[31:24])	Functionality
2a*	xxxx1011	R(15) ← R(15) + {30'b(SEXT(IR[23:0])), 2'b0} + 4 R(14) ← R(15) – 8 IR[31:28]=Condition codes (tracing back to previous instruction)
Final*	xxxx101x	If IR[24]=0: R(15) ← R(15) + 30'b(SEXT(IR[23:0])), 2'b0} + 4 If IR[24]=1: R(15) ← R(15) + 30'b(SEXT(IR[23:0])), 2'b0} + 4 R(14) ← R(15) – 8 IR[31:28]=Condition codes (tracing back to previous instruction)

* Condition codes must be true to execute instruction

Figure 4.8: Merged RISA for branch instructions in the ARM 7

As seen in Figure 4.6, four of the branch data paths are not sensitizable and can be excluded from further analysis. In step 5, REVE determines data and control implications for any data paths retained in step 4. This requires an examination of the PLA to determine the inputs necessary to reach each state along each path; these inputs are traced back to major architectural components such as the instruction register. This process reverse engineers any necessary opcode bits for execution of each instruction. In addition, all PLA outputs are propagated throughout the datapath to determine the propagation of other, non-opcode, instruction register bits, such as ALU control signals. This leads to the results in step 6, where an instruction for each remaining data path is derived. Figure 4.7 shows this unmerged RISA for the four remaining paths listed in Figure 4.6.

Each of the instructions in Figure 4.7 contains 4 condition code bits IR[31:28]. In the course of reverse engineering, we discovered that each instruction provides an *execute* bit as an input to the PLA, which comes from a condition code check module. REVE detected that this condition code check occurs at the beginning of every instruction, so the condition code values were added to our analysis whenever we encountered the *execute* bit.

Step 7 merges instructions in the RISA. From Figure 4.7, we can see that instructions 2, 3, and 4 have the same opcode bits, and are candidates for merging into a new instruction 2a. The merged instruction combining these three entries is shown in Figure 4.8. In order to generate instruction 2a, we need to know the order of occurrence for instructions 3 and 4. Both of these instructions load a value to R(14), and an incorrect ordering would result in the wrong R(14) assignment in 2a. We therefore examine the state table implied by the PLA to determine which instruction executes first. On doing this, we determine that instruction 3 occurs before instruction 4. Therefore, R(14) is first loaded with the value in R(15), and then R(14) decrements itself by 8. These two operations are expressed by R(14) ← R(15) – 8. In addition, the shifter_output algorithm shown in Figure 4.7 reduces to a sign extension of the IR[23] bit.

Finally, we notice that instruction 1 and the new instruction 2a only differ in the IR[24] bit, and so can also be merged. Therefore, we can collapse all branch instructions into the final instruction given in Figure 4.8.

In order to complete our analysis of pipeline operation under normal mode, we verify that the original ISA is a subset of the RISA. The branch instruction from the original ISA is shown in Figure 4.9. As we can see, the ISA and RISA instructions in Figure 4.8 and Figure 4.9 are

Assigned Opcode Bits (IR[31:24])	Functionality
xxxx101x	If IR[24] = 0: R(15) = PC ← PC + {30'b(SEXT(IR[23:0])), 2'b0} If IR[24] = 1: R(15) = PC ← PC + {30'b(SEXT(IR[23:0])), 2'b0} R(14) ← PC + 4 IR[31:28]=Condition codes (tracing back to previous instruction) * Condition codes must be true to execute instruction

Figure 4.9: Original ISA for branch instructions in the ARM 7

nearly identical. However, there are two small differences indicating bugs, which we analyze next.

In the implementation, the current PC in R(15) points to the address of the instruction in the fetch stage and is updated to point to the next instruction at the end of each instruction cycle. Therefore, the PC value of the instruction in the execute stage is really R(15) – 8. In order to branch correctly we need to account for this – 8, as well as an additional 4 cycles due to the fact that R(15) is not written until the end of the cycle. Therefore, the correct branch location is R(15) + SEXT(offset) – 8 + 4, which simplifies to R(15) + SEXT(offset) – 4. As shown in Figure 4.8, the implementation erroneously branches to R(15) + SEXT(offset) + 4. This bug arises because the designer did not account for the subtraction of 8 due to the pipeline delay.

The second bug relates to the link operation, which loads a subroutine return value into R(14). The implementation loads a value of R(15) – 8, which results in the current PC. However, as seen from Figure 4.9, a correct implementation should load the PC of the next instruction, or PC + 4. Therefore, the correct value to be loaded into R(14) is R(15) – 8 + 4 = R(15) – 4. Since the implementation loads R(15) – 8, this is clearly a bug. In this case, the designer remembered to account for pipeline depth but loaded the current instruction instead of the next instruction.

The above analysis temporarily skipped step 8, which verifies implementation-specific properties about the three-stage pipeline. As previously noted, there are no forwarding paths in the machine. Therefore, we must only verify that squashing and stalling are implemented correctly.

The ARM 7 has three squash conditions. The first two cases occur during a branch or a system reset, as described in Chapter 3. The third case is due to an instruction failing its condition. In this case, we must verify that the failed instruction does not write results back to the register file or memory.

First, we verify that squashing occurs whenever a branch is taken. Since the PC is updated near the end of the instruction cycle, after the branch address has been determined, the instruction in the fetch stage is the correct instruction after the branch. Therefore, we only need to squash the instruction in the decode stage. This implementation handles squashing by performing an extra shift on the instruction pipeline register shown in Figure 4.2 during the branch instruction, after the branch target address is known. This causes the instruction being fetched (which is the branch target) to be shifted to the decode stage, and the instruction in the decode stage to be shifted to execute. At the end of the branch operation, a pipeline shift is again performed, shifting the branch target to the execute stage, and shifting the instruction we needed to squash out of the processor. This extra shift in the middle of a branch instruction avoids the

need for squashing write-enable bits. Since REVE already monitors control flow between states, this extra pipeline shift is detected.

The second squash condition is a system reset. Examination of the Verilog code reveals that each pipeline register takes a variable, RESET_BAR, as a control input. During a reset, this signal is a 0, and all bits in the pipeline registers are erased.

Third, we must verify squashing when an instruction's condition test fails. All instructions pass through a condition code check module, and as stated in Section 4.2, an instruction that fails is treated as a no-operation instruction in the execute stage. Since a no-operation instruction never writes data to the register file or memory, write-back data corresponding to the failed instruction is squashed.

The second pipeline property to verify is stalling. As described in Chapter 3, we must verify three stalling properties. The first is to check that stalling covers any data hazards not covered by forwarding. Since data hazards do not exist in this implementation, this check is unnecessary. The second stalling property is that stages requiring extra execution time stall preceding stages. This involves knowledge of signals such as memory wait signals or loop variables. In the ARM 7 implementation, these signals are encoded in the PLA, and as seen from the PLA (Appendix E), any stage requiring extra execution time turns off a global signal, ipr_shift, which stalls each stage in the processor. This signal also initiates squashing for branch instructions, as described above. The third stalling property requires that stalls in late pipeline stages also stall early stages. The implementation uses the ipr_shift signal to stall all stages. Therefore, anytime a late stage such as execute stalls, all preceding stages are stalled as well.

4.4 Verification Results

In the previous section, REVE was used to verify an implementation of the ARM 7 processor and successfully detected a few bugs in its branch instructions. Appendix F shows the complete verification results for all remaining instructions. This section briefly discusses the initial verification issues, and describes all bugs found.

A big problem we encountered was how to deal with long instructions having datapath loops, specifically the multiply instruction. Multiply is implemented by a series of add and shift instructions using a version of Booth's algorithm [24]. Our first reverse-engineering attempts only detected these individual add and shift operations, and not the full multiply instruction. Therefore, verifying multiply required a tightening of our control analysis constraints. The multiply instruction assigns a loop variable called end_list, which is an input to the control PLA, and determines the processor's next state. Originally, we ignored this variable, which resulted in not knowing when the multiply loop finished.

4.4.1 Bug Descriptions

During verification of the processor under normal mode, REVE discovered no fewer than 9 design errors in the ARM 7 implementation. Although the machine contains a 3-stage pipeline, we did not find bugs during pipeline modes of operation. This is partly due to the large amount of work done by the execute stage requiring very little interaction between pipeline stages, and because of the lack of forwarding. A summary of each bug discovered in the ARM 7 follows.

1. **Branch instruction branches to the wrong location**
   This bug is described in Section 4.3 during our analysis of branch instructions. The net result is that a branch instruction shifts program flow to an address 8 bytes higher than it should. This error occurs because the designer neglected to consider pipeline depth when he calculated the branch address.

2. **Branch-and-link instruction links the wrong return value**
   This bug is also described in Section 4.3. The net result is that the current PC is linked instead of the next PC during a branch. While the designer correctly accounted for pipeline depth in calculating the link value, he forgot to increment the PC value before storing it. This error was likely due to not reading the fine print of the ISA.

3. **Single data transfer and single data swap instructions have incorrect word alignment**
   This error occurs when a single data transfer or swap instruction is performed, we are transferring a word quantity, and the last 2 bits of the address are 11. In order to support word alignment, all data is rotated by an amount based on the last 2 address bits. The algorithm used for implementing word alignment is shown below.

```
else if (shift_format==4'b0111) begin
    if (address_offset==0) begin
        barrel_out = barrel_in;
        end
    else if (address_offset==1) begin
        barrel_out[31:8] = barrel_in[23:0];
        barrel_out[7:0] = barrel_in[31:24];
        end
    else if (address_offset==2) begin
       barrel_out[31:16] = barrel_in[15:0];
       barrel_out[15:0] = barrel_in[31:16];
       end
    else begin
        barrel_out[31:24] = barrel_in[7:0];
        barrel_out[7:0] = barrel_in[31:24];
        end
    end
```

   As highlighted, when the address_offset is 3, bits 23:8 are not rotated at all. A correct implementation would change the highlighted line to:

```
barrel_out[23:0] = barrel_in[31:8]
```

4. **Single data swap instructions do not enforce IR[21:20] assignment**
   The ARM 7 ISA states that a single data swap instruction must have the value 00 in bits 21 and 20. The implementation does not enforce this restriction, which causes problems if an illegal instruction is accidentally issued. Problems may also arise with backward compatibility for future ISA enhancements where bits 21 and 20 that are not equal to 00 may refer to something other than a single data swap.

5. **Data-processing instructions do not correctly implement immediate rotates**

The ARM 7 ISA states that when an immediate operand is used in a data-processing instruction, it should be zero-extended to 32 bits and rotated by twice the value in the instruction's rotate field IR[11:8]. The implementation performs a right shift instead of a right rotate. Consequently, rotated data is not wrapped around properly.

6. **Block data-transfer instructions do not decrement addresses properly**
   In block data-transfer instructions with decrement addressing (where the base is subtracted from the offset), the original address decrements to the wrong value. The implementation decrements the address to Base – (the number of registers in the register list) + 8, while the correct operation specified by the ISA decrements to Base – (the number of registers in the register list). A likely reason for this bug is the designer unnecessarily considering pipeline depth. Since the base register is not the program counter but a register specified in the instruction, there is no need to add 8 to account for pipeline depth.

7. **Block data-transfer instructions fail for a register list with more than 8 elements**
   This bug occurs during a block data-transfer instruction when the number of registers specified by the register list is greater than 8. The internal register which holds the number of registers in the register list is 5 bits wide. The maximal register count specified by the ISA is 16, requiring 4 bits of storage. However, because all addresses are word-aligned, this count is left-shifted by 2 bits in order to create an offset of 4 bytes between each memory location. During this 2-bit shift, the highest bit in the original 4-bit entry is shifted out of the internal register, as the register is only 5 bits wide. Therefore, any register list containing more than 8 registers will expose this bug.

8. **Condition code error occurs in LS**
   This bug is in the module that checks condition codes. The ARM 7 ISA states that the LS (less than) condition occurs when the C flag is clear **OR** the Z flag is set, while in the implementation, LS occurs when C is clear **AND** Z is set. This is a simple logic function error.

9. **Condition code error occurs in LE**
   This bug is also in the module that checks condition codes. The ARM 7 ISA states that the LE (less than or equal) condition occurs when Z is set, **OR** (N is set AND V is clear), OR (N is clear and V is set), while in the implementation, LE occurs when Z is set **AND** ((N is set AND V is clear) OR (N is clear and V is set)). Again, this is a simple logic function error.

CHAPTER 5
CONCLUSIONS

As today's processors increase in complexity, it is becoming harder to adequately verify a design with standard simulation-based methods. In order to detect difficult design errors, such as the notorious Pentium bug, industry is utilizing formal methods in conjunction with simulation. Formal verification methods, however, are rarely scalable to the size of current processors. There is a growing need for high-level design verification methods that are both practical and reliable, which provides the motivation for this work.

5.1 Thesis Contributions

This thesis defines a new verification methodology based on reverse-engineering principles. It can be considered as a "formal" method, although we have not presented it in rigorous mathematical terms. Since other formal methods such as equivalence checking verify designs below the RTL level, we developed REVE to perform high-level verification, specifically to verify that a processor's RTL implementation satisfies its architectural specification, the ISA.

Figure 5.1 compares REVE with several other verification methods. The first four row entries in this table correspond to well-known verification methodologies. The next three rows are promising new methods being researched in academia. The main contribution of our work is a practical method to verify an RTL implementation against its ISA for instruction-set processors. The key features of our method are:

- Analysis is based on data paths instead of state spaces.
- Knowledge bases are used to hold information about basic designs, and can be expanded to enable REVE to verify more complex designs
- The method is easy to apply manually and appears to be automatable.
- Its effectiveness has been demonstrated by uncovering previously unknown bugs in an implementation of the ARM 7

Although several existing methods work at the RTL level, they are either incapable of verifying all aspects of a processor, or are only applicable to relatively small examples [7,8]. As we have shown in Chapters 2-4, REVE is capable of fully verifying an RTL implementation, and can handle processors with advanced features such as pipelining.

Figure 5.1: Comparison of REVE with several other verification methodologies

Methods	Method summary	Primary verification levels	Verification scope	Advantages	Disadvantages
Equivalence checking	Verifies equivalence between two levels of a design hierarchy	RTL to transistor	Small modules	Very reliable and easy to use.	Scope is limited to small combinational modules.
Theorem proving	Mathematically proves theorems about an implementation	Specification to RTL	Datapath	The mathematical proof is complete and fully verifies an implementation.	Very poor automation, slow, and restricted to small implementations. Requires a theorem-proving expert. Requires an extra specification to be written.
Model checking	Checks that various finite state machine properties are satisfied	RTL	FSMs	Easy to use. Good for checking state machine properties.	Runs into state space explosion problems. Less useful for complete processor verification.
STE	Verifies an implementation by symbolic simulation	Specification to RTL or gate-level	Large modules	Based on well understood simulation-based methods.	Requires an extra specification to be written. Limited by circuit complexity.
Unpipelining [10]	Iteratively merges pipeline stages to simplify to a multi-cycle machine	Specification to RTL	Control	Capable of verifying an entire pipeline.	Bugs may slip through since the designer must provide templates to verify an implementation.
Murphi [12]	Exhaustively explores state space for hard-to-find control bugs	RTL	Datapath	Good for corner-case bugs in control logic.	Limited by circuit complexity.
Error-based test generation [14]	Searches for bugs using error models and test generation methodology	RTL	Datapath	Uses well-defined test-generation methodology.	Error models need to be expanded in order to catch all bugs. High-level test generation tools not available.
REVE	Recreates ISA based on internal data path connections and control flow	Specification to RTL	Datapath	Verifies a high-level design based on straightforward data path analysis.	Relies on knowledge bases to guide verification process.

The reverse-engineering principles underlying REVE are applied to data-path analysis. While there is no guarantee that the number of data paths is not exponential with respect to the number of knowledge base components, it is our experience that most implementations have a number of data paths that is much smaller than the number of states in the processor's state-space. For example, REVE reverse engineered 41 data paths for the ARM 7 implementation, as shown in Figure 4.4. The total state-space for this implementation estimated from the number of bits in the control PLA is 2^{29}. It is obvious that, at least for this example, data path analysis is less likely to encounter size constraints than a state-space based method, and is likely to be faster.

The primary disadvantage of our method is the need to construct knowledge bases about the type of implementation. While the information in KB0 is simple and general and applies to most modern processors, implementation-specific information such as that described in Chapter 3 is less general and much harder to define. The pipeline assumptions in Chapter 3 work for common pipeline implementations, but must be extended when unusual design methods are used. For example, as shown in Section 3.1.2, extra specification information was necessary to include conditional instruction execution in our analysis of squashing. For this reason, our method suffers from some of the same drawbacks as the unpipelining approach of [10].

5.2 Extensions and Future Work

Our work has introduced the systematic use of reverse-engineering principles in the field of design verification. However, there is still much work to be done. The major task is to completely formalize the methods used in REVE, and then to create a tool implementing REVE. The experiments in this thesis were done by hand, a very time-consuming process and automation of REVE is very desirable. In the course of our experimentation, we tried to keep CAD issues in mind. While unforeseen issues will undoubtably arise in CAD tool implementation, we feel that the major algorithms in our method can be automated without excessive difficulty.

In addition, REVE can be extended to more complex architectures, such as the superscalar architectures found in most of today's high performance microprocessors. This might be handled by treating each pipeline in the superscalar machine as a separate entity to be verified by REVE. Once each pipeline is individually verified, interactions between the pipelines need to be checked. This involves adding superscalar information to REVE similar to the pipelined information for interactions between instructions that were described in Chapter 3. While implementation of these extensions is necessary before the REVE approach can be applied to the most advanced processors, REVE's demonstrated ability to detect bugs in the widely used ARM 7 microprocessor is very encouraging.

BIBLIOGRAPHY

[1] R. Bryant, "Bit-Level Analysis of an SRT Divider Circuit," *Proc. 33rd Design Automation Conference*, pp. 661-665, 1996.

[2] J. A. Abraham, "Hardware Verification: Theory and Practice," *Proc. IFIP Workshop on Dependable Computing & Applns.*, Johannesburg, Jan. 1998.

[3] C. Pixley, et al, "Commercial Design Verification: Methodology and Tools," *Proc. Int. Test Conference*, pp. 838-848, 1996.

[4] D. P. Appenzeller and A. Kuehlmann, "Formal Verification of a PowerPC Microprocessor," *Proc. Int. Conf. CAD*, pp. 79-84, 1995.

[5] T. Villa et al, *VIS User's Manual*, UC Berkeley, http://www-cad.eecs.berkeley.edu/~vis, pp. 18-23, 1996.

[6] P. Ho, A. Isles, and T. Kam, "Formal Verification of Pipeline Control Using Controlled Token Nets and Abstract Interpretation," *Proc. Int. Conf. CAD*, pp. 529-536, 1998.

[7] M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor," *IEEE Software*, vol. 7, no. 5, pp. 52-64, 1990.

[8] S. P. Miller and M. Srivas, "Formal Verification of the AAMP5 Microprocessor," IEEE *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, pp. 30-43, 1995.

[9] C. H. Seger, *Voss – A Formal Hardware Verification System User's Guide*, University of British Columbia, 1996.

[10] J. Levitt and K. Olukotun, "Verifying correct pipeline implementation for microprocessors," *Proc. Int. Conf. CAD*, pp. 162-169, 1997.

[11] J. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control." *Proc. Computer-Aided Verification Conf.*, pp. 68-80, 1994.

[12] R. Ho, et al, "Architecture Validation for Processors", *Proc. Int. Symp. on Computer Architecture*, pp. 404-413, 1995.

[13] X. Shen and Arvind, "Modeling and Verification of ISA Implementations", *Proc. Australasian Computer Architecture Conf.*, 1998.

[14] D. V. Campenhout, T. Mudge, and J. P. Hayes, "High-Level Test Generation for Design Verification of Pipelined Microprocessors," Digest of Papers, *Int. High Level Design Validation and Test Workshop*, pp. 1-8, 1998.

[15] M. G. Rekoff Jr., "On Reverse Engineering," *3rd IEEE Trans. Systems, Man, and Cybernetics*, vol. SMC-15, no. 2, March-April 1985, pp. 244-252.

[16] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 3, pp. 13-17, 1990.

[17] J. B. Guignet, "Generalized Recognition of Gates," *Proc. European Design & Test Conf.*, p. 608, 1996.

[18] M. C. Hansen, H. Yalcin, and J. P. Hayes. "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design and Test*, to appear.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. McGraw–Hill, New York, 1997.

[20] M. Postiff, *LC-2 Programmer's Reference Manual*, Revision 4.0, The University of Michigan, http://www.eecs.umich.edu/~postiffm/lc2/lc2.html, 1997.

[21] D. Patterson and J. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, 1994.

[22] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 2nd ed*. Morgan Kaufmann, San Francisco, 1996.

[23] *ARM 7 Data Sheet*, ARM DDI 0020C, Advanced RISC Machines Ltd., 1994.

[24] K. Sit, *Computer Simulation using HDL*, EECS499 Directed Study Report, The University of Michigan, 1996.

APPENDICES

APPENDIX A: Condensed Instruction Set of LC-2

This appendix presents a condensed version of the LC-2 instruction set. The instruction encodings can be found in Figure 2.15.

ADD — Addition

Assembler Formats:

```
ADD DR, SR1, SR2
ADD DR, SR1, imm5
```

Operation:

```
if (bit<5> == 0) {
    DR = SR1 + SR2
}
else if (bit<5> == 1) {
    DR = SR1 + SEXT(imm5)
}
```

Condition Codes Modified:

N, Z, P

AND — Bitwise logical AND

Assembler Formats:

```
AND DR, SR1, SR2
AND DR, SR1, imm5
```

Operation:

```
if (bit<5> == 0) {
    DR = SR1 AND SR2
}
else if (bit<5> == 1) {
    DR = SR1 AND SEXT(imm5)
}
```

Condition Codes Modified:

N, Z, P

BR — Branch to location on current page

Assembler Formats:

BR LABEL	BRNZP LABEL
BRN LABEL	BRLT LABEL
BRZ LABEL	BREQ LABEL
BRP LABEL	BRGT LABEL
BRNZ LABEL	BRLE LABEL
BRNP LABEL	BRNE LABEL
BRZP LABEL	BRGE LABEL

Operation:

```
if (condition specified by NZP bits is true) {
    PC = PC<15:9> @ pgoffset9
}
else if (bit<5> == 1) {
    DR = SR1 AND SEXT(imm5)
}
```

Code	Condition Codes Set	Condition	Code	Condition Codes Set	Condition
000		Do not branch under any condition	100	N	Negative, Less Than
001	P	Positive, Greater Than	101	N or P	Negative or Positive, Not Equal
010	Z	Zero, Equal	110	N or Z	Negative or Zero, Less than or Equal
011	Z or P	Zero or Positive, Greater than or Equal	111	N, Z, or P	Negative, Zero, or Positive, Unconditional

Condition Codes Modified:

none

JSR — Jump to Subroutine
JMP — Jump

Assembler Formats:

JSR LABEL (L=1)
JMP LABEL (L=0)

Operation:

```
if (L == 1) {
    R7 = PC
}
PC = PC<15:9> @ pgoffset9
```

Condition Codes Modified:

none

JSRR Jump to Subroutine through Register
JMPR Jump through Register

Assembler Formats:

JSRR BaseR, index6 (L=1)
JMPR BaseR, index6 (L=0)

Operation:

```
if (L == 1) {
    R7 = PC
}
PC = BaseR + ZEXT(index6)
```

Condition Codes Modified:

none

LD Load from memory to register

Assembler Formats:

LD DR, LABEL

Operation:

DR = mem[PC<15:9> @ pgoffset9]

Condition Codes Modified:

N, Z, P

LDI Load Indirect from memory to register

Assembler Formats:

LDI DR, LABEL

Operation:

DR = mem[mem[PC<15:9> @ pgoffset9]]

Condition Codes Modified:

N, Z, P

LDR Load from mem[Base + index] to register

Assembler Formats:

LDR DR, BaseR, index6

Operation:

DR = mem[BaseR + ZEXT(index6)]

Condition Codes Modified:

N, Z, P

LEA — Load Effective Address

Assembler Formats:
LEA DR, LABEL
Operation:
DR = PC<15:9> @ pgoffset9
Condition Codes Modified:
N, Z, P

NOP — No Operation

Assembler Formats:
NOP
Operation:
Nothing
Condition Codes Modified:
none

NOT — Bitwise NOT (invert or complement)

Assembler Formats:
NOT DR, SR
Operation:
DR = NOT(SR)
Condition Codes Modified:
N, Z, P

RET — Return from subroutine

Assembler Formats:
RET
Operation:
PC = R7
Condition Codes Modified:
none

ST — Store from register to memory

Assembler Formats:
ST SR, LABEL
Operation:
mem[PC<15:9> @ pgoffset9] = SR
Condition Codes Modified:
none

STI Store Indirect from register to memory

Assembler Formats:

STI SR, LABEL

Operation:

mem[mem[PC<15:9> @ pgoffset9]] = SR

Condition Codes Modified:

none

STR Store from register to memory[Base + Offset]

Assembler Formats:

STR SR, BaseR, index6

Operation:

mem[(BaseR + ZEXT(index6)] = SR

Condition Codes Modified:

none

TRAP System call

Assembler Formats:

TRAP trapvect8

Operation:

R7 = PC

PC = mem[ZEXT(trapvect8)]

Condition Codes Modified:

none

APPENDIX B: Verilog HDL Code of Unpipelined LC-2 Implementation

Top Level Module

```
/*********************************************************************************
$RCSfile: cpu.v,v $
*********************************************************************************/

module rtcpu(clock,clear,dbus,abus,write_mem_bar,read_mem_bar);

input clock,clear;
inout [15:0] dbus;
output [15:0] abus;
output write_mem_bar,read_mem_bar;

wire [15:0] ir_out;
wire [2:0] R1,R2,W,flags_out;
wire [1:0] sel_rf_mux, sel_pc_mux, sel_ab_mux;

datapath DP(clock,clear,dbus,abus,ir_out,flags_out,R1,R2,W,RE1,RE2,WE,S3,S2,S1,S0,M,
            load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,
            load_reg1_bar,load_reg2_bar,sel_rf_mux, sel_pc_mux,
            sel_mar_mux, sel_ab_mux,sel_alu_mux,reg2_to_dbus_bar,zero_or_sign,
            trapvec_bar);

control CO(clock,clear,write_mem_bar,read_mem_bar,R1,R2,W,RE1,RE2,WE,S3,S2,S1,S0,M,
           load_pc_bar,load_ir_bar, load_mar_bar,
           load_flags_bar,load_reg1_bar,load_reg2_bar,
           sel_rf_mux, sel_pc_mux,sel_mar_mux, sel_ab_mux,
           sel_alu_mux,reg2_to_dbus_bar,zero_or_sign,trapvec_bar,
           ir_out,flags_out);
endmodule
```

Datapath Module

```
/*********************************************************************************
$RCSfile: datapath.v,v $
*********************************************************************************/

module datapath(clock,clear,dbus,abus,ir_out,flags_out,R1,R2,W,RE1,RE2,WE,S3,S2,S1,S0,M,
            load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,
            load_reg1_bar,load_reg2_bar,sel_rf_mux, sel_pc_mux,sel_mar_mux,
            sel_ab_mux,sel_alu_mux,reg2_to_dbus_bar,zero_or_sign,trapvec_bar);

input clock,clear;
inout  [15:0] dbus;
output [15:0] abus;

// TO CONTROL
output [15:0] ir_out;
output [2:0] flags_out;
```

```
// REGFILE
input [2:0] R1, R2, W;
input RE1, RE2, WE;

// ALU
input S3, S2, S1, S0, M;

// REGISTERS
input load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,load_reg1_bar,
    load_reg2_bar;

// MUXS
input [1:0] sel_rf_mux, sel_pc_mux, sel_ab_mux;
input sel_alu_mux, sel_mar_mux;

// TRISTATE
input reg2_to_dbus_bar;

// SPECIAL
input zero_or_sign;
input trapvec_bar;

wire [2:0] flags_in;
wire [15:0] pc_in,read_port1,read_port2,ALU_B_port,alu_out,write_port,
         rf_port1,rf_port2,mar_in,pc_out,mar_out,merge_out,inc_out,extend_out,
         latch_in;
wire clock_bar;

stdinv STI(clock,clock_bar);
alu #(16) ALU0(read_port1,ALU_B_port,1'b0,M,S0,S1,S2,S3,Dummy_COUT,alu_out);
latch #(16) LA(latch_in,clock_bar,write_port);
regfile2r #(16,8,3) RF(write_port,R1,R2,RE1,RE2,W,WE,rf_port1,rf_port2);
dffh_c #(16) REG1 (clock,clear,rf_port1,load_reg1_bar,read_port1),
          REG2 (clock,clear,rf_port2,load_reg2_bar,read_port2);
dffh_c #(16) IR (clock,clear,dbus,load_ir_bar,ir_out),
          MAR (clock,clear,mar_in,load_mar_bar,mar_out);
pregister PC (clock,clear,pc_in,load_pc_bar,pc_out);
dffh_c #(3) FLAGS (clock,clear,flags_in,load_flags_bar,flags_out);
mux4 #(16) RFMUX(alu_out,inc_out,merge_out,dbus,sel_rf_mux[0],sel_rf_mux[1],latch_in),
         PCMUX(inc_out,alu_out,merge_out,dbus,sel_pc_mux[0],sel_pc_mux[1],pc_in);
mux3 #(16) ABMUX(pc_out,mar_out,merge_out,sel_ab_mux[0],sel_ab_mux[1],abus);
mux2 #(16) ALUMUX(read_port2,extend_out,sel_alu_mux,ALU_B_port),
         MARMUX(alu_out,dbus,sel_mar_mux,mar_in);
tribuf #(16) TRB(reg2_to_dbus_bar,read_port2,dbus);
extend EXT(ir_out,zero_or_sign,extend_out);
detect DTC(write_port,flags_in);
inc #(16) INC0(1'b1,pc_out,Dummy_TC,Dummy_TCBAR,inc_out);
merge MRG(pc_out,ir_out,trapvec_bar,merge_out);

endmodule
```

Control Module

```
/*****************************************************************************
$RCSfile: control.v,v $
*****************************************************************************/
module control(clock,clear,write_mem_bar, read_mem_bar,R1,R2,W,RE1,RE2,WE,S3,S2,S1,S0,M,
            load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,
            load_reg1_bar,load_reg2_bar,sel_rf_mux,
            sel_pc_mux,sel_mar_mux, sel_ab_mux,
            sel_alu_mux,reg2_to_bus_bar,zero_or_sign,trapvec_bar,ir_out,flags_out);

input clock,clear;

// TO MEMORY
output write_mem_bar, read_mem_bar;

// TO REGFILE
output [2:0] R1, R2, W;
output RE1, RE2, WE;

// TO ALU
output S3, S2, S1, S0, M;

// TO REGISTERS
output load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,load_reg1_bar,
      load_reg2_bar;

// TO MUXS
output [1:0] sel_rf_mux, sel_pc_mux, sel_ab_mux;
output sel_alu_mux, sel_mar_mux;

// TO TRISTATE
output reg2_to_bus_bar;

// TO SPECIAL
output zero_or_sign;
output trapvec_bar;

// FROM_temp DATAPATH
input [15:0] ir_out;
input [2:0] flags_out;

reg [2:0] machine_state;
reg [2:0] next_state;

reg write_mem_bar_temp, read_mem_bar_temp;
reg [2:0] R1_temp, R2_temp, W_temp;
reg RE1_temp, RE2_temp, WE_temp;
reg S3_temp, S2_temp, S1_temp, S0_temp, M_temp;
reg load_pc_bar_temp, load_ir_bar_temp, load_mar_bar_temp,
    load_flags_bar_temp, load_reg1_bar_temp, load_reg2_bar_temp;
reg [1:0] sel_rf_mux_temp, sel_pc_mux_temp, sel_ab_mux_temp;
reg sel_alu_mux_temp, sel_mar_mux_temp;
reg reg2_to_bus_bar_temp;
reg zero_or_sign_temp;
reg trapvec_bar_temp;

// Machine States
```

```
‘define MRESET_STATE 3’b000
‘define IFETCH_STATE 3’b001
‘define DECODE_STATE 3’b010
‘define EX_MEM_STATE 3’b011
‘define MEMORY_STATE 3’b100

// INSTRUCTIONS
‘define ADD  4’b0001
‘define AND  4’b0101
‘define BR   4’b1000
‘define JSR  4’b0100
‘define JSRR 4’b1100
‘define LD   4’b0010
‘define LDI  4’b1010
‘define LDR  4’b0110
‘define LEA  4’b1110
‘define NOP  4’b0000
‘define NOT  4’b1001
‘define RET  4’b1101
‘define ST   4’b0011
‘define STI  4’b1011
‘define STR  4’b0111
‘define TRAP 4’b1111

// DELAY
‘define FSM_DELAY 6

// STATE MACHINE
always @(posedge clock)
begin
   machine_state = next_state;
end

always @(ir_out[15:5] or ir_out[2:0] or clear or machine_state or flags_out)
begin
  // Generate addresses for RF
  if (ir_out[15:12] == 4’b1101)
     R1_temp = 3’b111;
  else
     R1_temp = ir_out[8:6];
  if (ir_out[13] == 1’b0)
       R2_temp = ir_out[2:0];
  else
       R2_temp = ir_out[11:9];
if ((ir_out[14:12] == 3’b100) || (ir_out[15:12] == 4’b1111))
       W_temp = 3’b111;
  else
       W_temp = ir_out[11:9];

  // Compute next state
if (clear == 1’b0)
       next_state = ‘MRESET_STATE;
  else
    begin
              case (machine_state)
                ‘MRESET_STATE:
```

```
                next_state = ‘IFETCH_STATE;
               ‘IFETCH_STATE:
                next_state = ‘DECODE_STATE;
               ‘DECODE_STATE:
                if (ir_out[15:12] == ‘NOP)
                      next_state = ‘IFETCH_STATE;
                else
                      next_state = ‘EX_MEM_STATE;
               ‘EX_MEM_STATE:
              begin
                      case (ir_out[15:12])
                        ‘AND,‘ADD,‘NOT,‘LEA,‘RET,‘BR,‘JSR,
                        ‘JSRR, ‘LD,‘ST,‘TRAP: next_state = ‘IFETCH_STATE;
                        ‘LDI,‘STI,‘LDR,‘STR:next_state = ‘MEMORY_STATE;
                      endcase
               end
                ‘MEMORY_STATE:next_state = ‘IFETCH_STATE;
             endcase
    end

  // Determine control signals for each state
if (clear == 1'b0)
    begin
      read_mem_bar_temp = 1'b1;
      write_mem_bar_temp = 1'b1;
      load_pc_bar_temp = 1'b1;
      load_ir_bar_temp = 1'b1;
      load_mar_bar_temp = 1'b1;
      load_flags_bar_temp = 1'b1;
      reg2_to_bus_bar_temp = 1'b1;
    end
  else
    begin
            case (machine_state)
              ‘MRESET_STATE:
                    begin
                      read_mem_bar_temp = 1'b1;
                      write_mem_bar_temp = 1'b1;
                      load_pc_bar_temp = 1'b1;
                      load_ir_bar_temp = 1'b1;
                      load_mar_bar_temp = 1'b1;
                      load_flags_bar_temp = 1'b1;
                      reg2_to_bus_bar_temp = 1'b1;
                    end
               ‘IFETCH_STATE:
              begin
                      read_mem_bar_temp = 1'b0;
                      write_mem_bar_temp = 1'b1;
                      RE1_temp = 1'b0;
                      RE2_temp = 1'b0;
                      WE_temp = 1'b0;

                      load_ir_bar_temp = 1'b0;
                      load_pc_bar_temp = 1'b1;
                      load_flags_bar_temp = 1'b1;
                      load_reg1_bar_temp = 1'b1;
```

```
        load_reg2_bar_temp = 1'b1;
        reg2_to_bus_bar_temp = 1'b1
        sel_ab_mux_temp = 2'b00;
end
 'DECODE_STATE:
begin
        read_mem_bar_temp = 1'b1;
        write_mem_bar_temp = 1'b1;
        RE1_temp = 1'b1;
        RE2_temp = 1'b1;
        WE_temp = 1'b0;
        load_ir_bar_temp = 1'b1;
        load_flags_bar_temp = 1'b1;
        load_reg1_bar_temp = 1'b0;
        load_reg2_bar_temp = 1'b0;
        reg2_to_bus_bar_temp = 1'b1;
        if (ir_out[15:12] == 'NOP)
          begin
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
          end
        else
          begin
            load_pc_bar_temp = 1'b1;
          end
end
 'EX_MEM_STATE:
begin
        RE1_temp = 1'b0;
        RE2_temp = 1'b0;
        load_ir_bar_temp = 1'b1;
        load_reg1_bar_temp = 1'b1;
        load_reg2_bar_temp = 1'b1;

        case (ir_out[15:12])
          'AND:
                begin
                  WE_temp = 1'b1;
                  load_pc_bar_temp = 1'b0;
                  sel_pc_mux_temp = 2'b00;
                  zero_or_sign_temp = 1'b1;
                  if (ir_out[5] == 1'b0)
                    sel_alu_mux_temp = 1'b0;
                  else
                    sel_alu_mux_temp = 1'b1;
                  S3_temp = 1'b1;
                  S2_temp = 1'b1;
                  S1_temp = 1'b1;
                  S0_temp = 1'b0;
                  M_temp  = 1'b0;
                  sel_rf_mux_temp = 2'b00;
                  load_flags_bar_temp = 1'b0;
                  reg2_to_bus_bar_temp = 1'b1;
                  read_mem_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
                end
```

```
‘ADD:
        begin
          WE_temp = 1'b1;
          load_pc_bar_temp = 1'b0;
          sel_pc_mux_temp = 2'b00;
          zero_or_sign_temp = 1'b1;
          if (ir_out[5] == 1'b0)
            sel_alu_mux_temp = 1'b0;
          else
            sel_alu_mux_temp = 1'b1;
          S3_temp = 1'b1;
          S2_temp = 1'b0;
          S1_temp = 1'b0;
          S0_temp = 1'b1;
          M_temp  = 1'b1;
          sel_rf_mux_temp = 2'b00;
          load_flags_bar_temp = 1'b0;
          reg2_to_bus_bar_temp = 1'b1;
          read_mem_bar_temp = 1'b1;
                write_mem_bar_temp = 1'b1;
         end
‘NOT:
        begin
          WE_temp = 1'b1;
          load_pc_bar_temp = 1'b0;
          sel_pc_mux_temp = 2'b00;
          S3_temp = 1'b0;
          S2_temp = 1'b0;
          S1_temp = 1'b0;
          S0_temp = 1'b0;
          M_temp  = 1'b0;
          sel_rf_mux_temp = 2'b00;
          load_flags_bar_temp = 1'b0;
          reg2_to_bus_bar_temp = 1'b1;
          read_mem_bar_temp = 1'b1;
                write_mem_bar_temp = 1'b1;
        end
‘LEA:
        begin
          WE_temp = 1'b1;
          load_pc_bar_temp = 1'b0;
          sel_pc_mux_temp = 2'b00;
          sel_rf_mux_temp = 2'b10;
          load_flags_bar_temp = 1'b0;
          trapvec_bar_temp = 1'b1;
          read_mem_bar_temp = 1'b1;
                write_mem_bar_temp = 1'b1;
          reg2_to_bus_bar_temp = 1'b1;
        end
‘RET:
        begin
          WE_temp = 1'b0;
          load_pc_bar_temp = 1'b0;
          sel_pc_mux_temp = 2'b01;
          S3_temp = 1'b1;
          S2_temp = 1'b1;
```

```
            S1_temp = 1'b1;
            S0_temp = 1'b1;
            M_temp  = 1'b0;
            load_flags_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
          end
'BR:
          begin
            WE_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
              if ((flags_out[2] & ir_out[11]) |
                 (flags_out[1] & ir_out[10]) |
                 (flags_out[0] & ir_out[9]))
                        sel_pc_mux_temp = 2'b10;
            else
                   sel_pc_mux_temp = 2'b00;
            load_flags_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
          end
'JSR:
          begin
            if (ir_out[11] == 1'b1)
              begin
                        sel_rf_mux_temp = 2'b01;
                        WE_temp = 1'b1;
              end
            else
              WE_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
          end
'JSRR:
          begin
            if (ir_out[11] == 1'b1)
              begin
                        sel_rf_mux_temp = 2'b01;
                        WE_temp = 1'b1;
              end
            else
              WE_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b01;
            sel_alu_mux_temp = 1'b1;
            zero_or_sign_temp = 1'b0;
            S3_temp = 1'b1;
            S2_temp = 1'b0;
```

```
            S1_temp = 1'b0;
            S0_temp = 1'b1;
            M_temp  = 1'b1;
            load_flags_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
          end
'LD:
          begin
            WE_temp = 1'b1;
            read_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
            sel_rf_mux_temp = 2'b11;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b0;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
          end
'ST:
          begin
            WE_temp = 1'b0;
            write_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b0;
            read_mem_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
          end
'TRAP:
          begin
            sel_rf_mux_temp = 2'b01;
            WE_temp = 1'b1;
            read_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b11;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b0;
            reg2_to_bus_bar_temp = 1'b1;
                  write_mem_bar_temp = 1'b1;
          end
'LDI,'STI:
          begin
            WE_temp = 1'b0;
            read_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            load_mar_bar_temp = 1'b0;
            sel_mar_mux_temp = 1'b1;
```

```
                trapvec_bar_temp = 1'b1;
                reg2_to_bus_bar_temp = 1'b1;
                write_mem_bar_temp = 1'b1;
              end
        'LDR,'STR:
              begin
                WE_temp = 1'b0;
                load_pc_bar_temp = 1'b0;
                sel_pc_mux_temp = 2'b00;
                sel_alu_mux_temp = 1'b1;
                zero_or_sign_temp = 1'b0;
                S3_temp = 1'b1;
                S2_temp = 1'b0;
                S1_temp = 1'b0;
                S0_temp = 1'b1;
                M_temp  = 1'b1;
                load_flags_bar_temp = 1'b1;
                load_mar_bar_temp = 1'b0;
                sel_mar_mux_temp = 1'b0;
                reg2_to_bus_bar_temp = 1'b1;
                read_mem_bar_temp = 1'b1;
                      write_mem_bar_temp = 1'b1;
              end
      endcase
end
 'MEMORY_STATE:
begin
      RE1_temp = 1'b0;
      RE2_temp = 1'b0;

      load_ir_bar_temp = 1'b1;
      load_reg1_bar_temp = 1'b1;
      load_reg2_bar_temp = 1'b1;

      sel_ab_mux_temp = 2'b01;
      load_pc_bar_temp = 1'b1;
      trapvec_bar_temp = 1'b1;
      sel_rf_mux_temp = 2'b11;

      case (ir_out[15:12])
        'LDI,'LDR:
              begin
                WE_temp = 1'b1;
                read_mem_bar_temp = 1'b0;
                write_mem_bar_temp = 1'b1;
                load_flags_bar_temp = 1'b0;
                reg2_to_bus_bar_temp = 1'b1;
              end
        'STI,'STR:
              begin
                WE_temp = 1'b0;

                write_mem_bar_temp = 1'b0;
                read_mem_bar_temp = 1'b1;
                load_flags_bar_temp = 1'b1;
                reg2_to_bus_bar_temp = 1'b0;
```

```
                        end
                    endcase
                end
            endcase
        end
end

// TO MEMORY

assign #`FSM_DELAY write_mem_bar = write_mem_bar_temp;
assign #`FSM_DELAY read_mem_bar = read_mem_bar_temp;

// TO REGFILE

assign #`FSM_DELAY R1 = R1_temp;
assign #`FSM_DELAY R2 = R2_temp;
assign #`FSM_DELAY W = W_temp;
assign #`FSM_DELAY RE1 = RE1_temp;
assign #`FSM_DELAY RE2 = RE2_temp;
assign #`FSM_DELAY WE = WE_temp;

// TO ALU

assign #`FSM_DELAY S3 = S3_temp;
assign #`FSM_DELAY S2 = S2_temp;
assign #`FSM_DELAY S1 = S1_temp;
assign #`FSM_DELAY S0 = S0_temp;
assign #`FSM_DELAY M = M_temp;

// TO REGISTERS

assign #`FSM_DELAY load_pc_bar = load_pc_bar_temp;
assign #`FSM_DELAY load_ir_bar = load_ir_bar_temp;
assign #`FSM_DELAY load_mar_bar = load_mar_bar_temp;
assign #`FSM_DELAY load_flags_bar = load_flags_bar_temp;
assign #`FSM_DELAY load_reg1_bar = load_reg1_bar_temp;
assign #`FSM_DELAY load_reg2_bar = load_reg2_bar_temp;

// TO MUXS

assign #`FSM_DELAY sel_rf_mux = sel_rf_mux_temp;
assign #`FSM_DELAY sel_pc_mux = sel_pc_mux_temp;
assign #`FSM_DELAY sel_mar_mux = sel_mar_mux_temp;
assign #`FSM_DELAY sel_ab_mux = sel_ab_mux_temp;
assign #`FSM_DELAY sel_alu_mux = sel_alu_mux_temp;

// TO TRISTATE

assign #`FSM_DELAY reg2_to_bus_bar = reg2_to_bus_bar_temp;

// TO SPECIAL

assign #`FSM_DELAY zero_or_sign = zero_or_sign_temp;
assign #`FSM_DELAY trapvec_bar = trapvec_bar_temp;

endmodule
```

APPENDIX C: Verification Results for Unpipelined LC-2

REVE Step 1

In our notation, periods indicate a submodule. Therefore, for the Program Counter, the notation "rtcpu.DP.PC" means that the Program Counter is component PC in module DP, which is a submodule of rtcpu. The components we identified were:

Program Counter:	rtcpu.DP.PC
Instruction Register:	rtcpu.DP.IR
Register File:	rtcpu.DP.RF
Data Bus:	rtcpu.dbus
Address Bus:	rtcpu.abus
System Clock:	rtcpu.clock
ALU:	rtcpu.DP.ALU0
Datapath Width:	16

REVE Step 2

The entirety of step 2 is shown in Chapter 2. The data paths traced in our analysis are shown in Figure 2.17.

REVE Step 3

Control signal extraction is shown for the data-processing class of instructions in Figure 2.18. The following figures present the control signal extraction data for the remaining three instruction classes: branch, load, and store.

Data (operand) sources	Signal name	Value	Activation cycle
1. PC			
2. ALU	clear	1	
	load_pc_bar	0	i
	sel_pc_mux[1:0]	01	i
3. PC @ IR	clear	1	
	load_pc_bar	0	i
	sel_pc_mux[1:0]	10	i
4. Data bus	clear	1	
	load_pc_bar	0	i
	sel_pc_mux[1:0]	11	i
With Address bus being ALU	sel_mar_mux	0	i
	load_mar_bar	0	i
	sel_ab_mux[1:0]	01	i
With Address bus being PC	sel_mar_mux	0	i
	load_mar_bar	0	i
	sel_ab_mux[1:0]	00	i
With Address bus being Data bus	sel_mar_mux	0	i
	load_mar_bar	0	i
	sel_ab_mux[1:0]	01	i
With Address bus being PC @ IR	sel_mar_mux	0	i
	load_mar_bar	0	i
	sel_ab_mux[1:0]	10	i

Control signals needed to sensitize branch data paths

Analysis for the branch instructions is shown above. For the ALU input path, the analysis must be combined with the paths from the data-processing instruction class, since the ALU is not an architectural state-holding element at which we stop path tracing. The results of this analysis are summarized in step 4.

Data (operand) sources	Signal name	Value	Activation cycle
1. ALU	clear	1	
	WE	1	
	sel_rf_mux[1:0]	00	
2. PC	clear	1	
	WE	1	i
	sel_rf_mux[1:0]	01	i
3. PC @ IR	clear	1	
	WE	1	i
	sel_rf_mux[1:0]	10	i
4. Data bus	clear	1	
	WE	1	i
	sel_rf_mux[1:0]	11	i
With Address bus being ALU	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	01	i-1
With Address bus being PC	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	00	i-1
With Address bus being Data bus	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	01	i-1
With Address bus being PC @ IR	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	10	i-1

Control signals needed to sensitize load data paths

Data (operand) sources	Signal name	Value	Activation cycle
1. Register file	clear	1	
	RE2	1	i-2
	load_reg2_bar	0	i-2
With Address bus being ALU	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	01	i
With Address bus being PC	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	00	i
With Address bus being Data bus	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	01	i
With Address bus being PC @ IR	sel_mar_mux	0	i-1
	load_mar_bar	0	i-1
	sel_ab_mux[1:0]	10	i

Control signals needed to sensitize store data paths

REVE Step 4

In step 4, we check for the existence of control sets defined in step 3. The following figures show the cases and special conditions necessary for the existence of each control set.

Data (operand) sources	**Case**	**Condtion**
2. ALU	‘RET	IR[5] = 0
	‘JSRR	IR[5] = 0
3. PC @ IR	‘BR	Flags_out(2) & IR[11] OR Flags_out(1) & IR[10] OR Flags_out(0) & IR[9]
	‘JSR	IR[5] = 1
4. Data bus		
With Address bus being ALU	None	
With Address bus being PC	None	
With Address bus being Data bus	None	
With Address bus being PC @ IR	‘TRAP	

Cases where control sets exist for branch data paths

Data (operand) sources	Case	Condtion
1. ALU (this is defined as the data-processing set of instructions)		
2. PC	‘JSR ‘JSRR ‘TRAP	IR[11] = 1 IR[11] = 1
3. PC @ IR	‘LEA	
4. Data bus		
With Address bus being ALU	‘LDR	
With Address bus being PC	None	
With Address bus being Data bus	‘LDI	
With Address bus being PC @ IR	‘LD	

Cases where control sets exist for load data paths

Data (operand) sources	Case	Condtion
1. Register file		
With Address bus being ALU	‘STR	
With Address bus being PC	None	
With Address bus being Data bus	‘ST ‘STI ‘TRAP	
With Address bus being PC @ IR	‘STI	

Cases where control sets exist for store data paths

REVE Steps 5 and 6

In step 5, we determine implications made by each control set, which are used in step 6 to generate an unmerged RISA. This RISA is shown in Figure 2.21.

REVE Step 7 and remaining steps

In step 7, we merge similar instructions to form a final RISA. The remaining analysis can be found in Chapter 2, and the final RISA is shown in Figure 2.22.

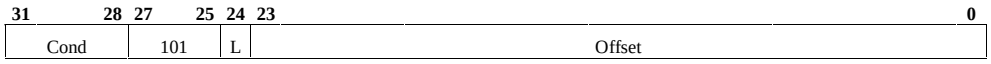
APPENDIX D: Condensed Instruction Set of ARM 7

This appendix presents a condensed version of the ARM 7 instruction set verified with REVE.

Branch and Branch with Link (B,BL)

Assembler syntax:

B{L}{cond} <expression>



Cond: Condition field

L: Link bit

0 = Branch

1 = Branch with Link

This instruction is executed only if the condition is true. Branch instructions contain a signed 2's complement 24-bit offset, which is left shifted by two bits, sign extended to 32 bits, and added to the PC. The instruction can specify a branch of +/- 32 Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

Branches beyond +/- 32 Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

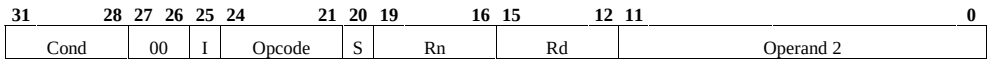
Data processing

Assembler syntax:

(1) MOV, MVN – single operand instructions
<opcode>{cond}{S} Rd,<Op2>

(2) CMP, CMN, TEQ, TST – instructions which do not produce a result
<opcode>{cond} Rn,<Op2>

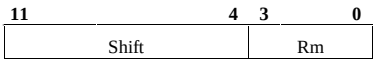
(3) AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, ORR, BIC
<opcode>{cond}{S} Rd,Rn,<Op2>



Cond: Condition field

I: Immediate Operand

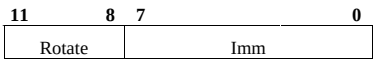
0 = operand 2 is a register



Shift: Shift applied to Rm

Rm: 2nd operand register

1 = operand 2 is an immediate value



Shift: Shift applied to Imm

Rm: Unsigned 8 bit immediate value

Opcode: Operation Code

0000 = AND: Rd = Op1 AND Op2
0001 = EOR: Rd = Op1 EOR Op2
0010 = SUB: Rd = Op1 – Op2
0011 = RSB: Rd = Op2 – Op1
0100 = ADD: Rd = Op1 + Op2
0101 = ADC: Rd = Op1 + Op2 + C
0110 = SBC: Rd = Op1 – Op2 + C – 1
0111 = RSC: Rd = Op2 – Op1 + C – 1
1000 = TST: Set condition codes on Op1 AND Op2
1001 = TEQ: Set condition codes on Op1 EOR Op2
1010 = CMP: Set condition codes on Op1 – Op2
1011 = CMN: Set condition codes on Op1 + Op2
1100 = ORR: Rd = Op1 OR Op2
1101 = MOV: Rd = Op2
1110 = BIC: Rd = Op1 AND NOT Op2
1111 = MVN: Rd = NOT Op2

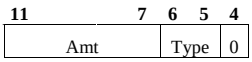
S: Set condition codes

Rn: 1st operand register

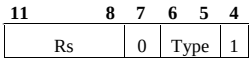
Rd: Destination register

This instruction is executed only if the condition is true. The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction.

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15).



Amt: Shift amount (5 bit unsigned integer)
Type: Shift type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right



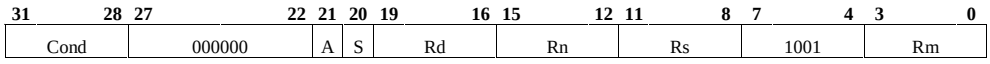
Rs: Shift register (Shift amount specified in bottom byte of Rs)
Type: Shift type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero-extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

Multiply and Multiply-Accumulate (MUL, MLA)

Assembler syntax:

MUL{cond}{S} Rd,Rm,Rs
MLA{cond}{S} Rd,Rm,Rs,Rn



Cond: Condition field
A: Accumulate
0 = multiply only
1 = multiply and accumulate
S: Set condition code
0 = do not alter condition codes
1 = set condition codes
D: Destination register
Rn, Rs, Rm: Operand registers

This instruction is executed only if the condition is true. The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32-bit operands, and may be used to synthesize higher precision multiplications.

The multiply form of the instruction gives Rd = Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd = Rm*Rs + Rn, which can save an explicit ADD instruction in some circumstances.

R15 shall not be used as an operand or as the destination register.

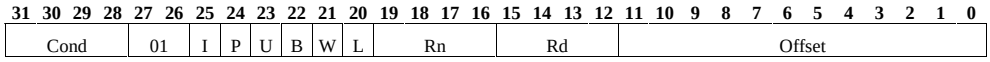
Single data transfer (LDR, STR)

Assembler syntax:

<LDR|STR>{cond}{B}{T} Rd,<Address>

LDR – load from memory into a register

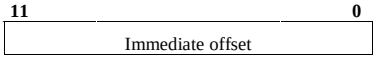
STR – store from a register into memory



Cond: Condition field

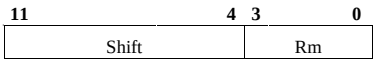
I: Immediate offset

0 = offset is an immediate value



Unsigned 12 bit immediate offset

1 = offset is a register



Shift: Shift applied to Rm

Rm: Offset register

P: Pre/Post indexing bit

0 = post; add offset after transfer

1 = pre; add offset before transfer

U: Up/Down bit

0 = down; subtract offset from base

1 = up; add offset to base

B: Byte/Word bit

0 = transfer word quantity

1 = transfer byte quantity

W: Write-back bit

0 = no write-back

1 = write address into base

L: Load/Store bit

0 = Store to memory

1 = Load from memory

Rn: Base register

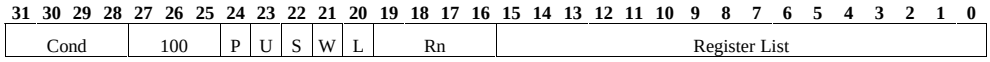
Rd: Source/Destination register

This instruction is executed only if the condition is true. The single date transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if 'auto-indexing' is required.

Block data transfer (LDM, STM)

Assembler syntax:

<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!},<Rlist>{^}
<Rlist> is a list of registers and register ranges enclosed in {}
{!} if present requests write-back (W=1), otherwise W=0
{^} if present set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode



Cond: Condition field

P: Pre/Post indexing bit

0 = post; add offset after transfer
1 = pre; add offset before transfer

U: Up/Down bit

0 = down; subtract offset from base
1 = up; add offset to base

S: PSR & force user bit

0 = do not load PSR or force user mode
1 = load PSR or force user mode

W: Write-back bit

0 = no write-back
1 = write address into base

L: Load/Store bit

0 = Store to memory
1 = Load from memory

Rn: Base register

This instruction is executed only if the condition is true. Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

The instruction can cause the transfer of any registers in the current bank. The register list is a 16-bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on. Any subset of the

registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Single data swap (SWP)

Assembler syntax:

<SWP>{cond}{B} Rd,Rm,[Rn]

31 30 29 28	27 26 25 24 23	22	21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
Cond	00010	B	00	Rn	Rd	0000	1001	Rm

Cond: Condition field
B: Byte/Word bit
0 = swap word quantity
1 = swap byte quantity
Rn: Base register
Rd: Destination register
Rm: Source register

This instruction is executed only if the condition is true. The data swap instruction is used to swap a byte or word quantity between a register and external memory. The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

APPENDIX E: Verilog HDL Code of ARM 7 Implementation

Note to thesis committee

Due to the amount of Verilog code in this appendix, we decided not to distribute it with this copy of the thesis. If necessary, I can supply a copy on request. Otherwise, it will be included in my final draft to be left with the department.

Address incrementer (Address_incrementer.v)

```
// This is a simulation of Address Incrementer of ARM 710
module Address_incrementer (out, in, incr_en, clk);
  // output to incrementer bus
  output [31:0] out;
  // input from address register
  input [31:0] in;
  // enable line
  input [0:0] incr_en;
  // clock
  input [0:0] clk;

  reg [31:0] out;
  parameter incr_amount = 4;

  always @ (posedge clk) begin
    if (incr_en) begin
      out = in + incr_amount;
      $display($time, "AI: AI_out=%b, AI_in=%b", out, in);
    end
  end

endmodule //of Address_incrementer
```

Address register (Address_register.v)

```
// This is a module to simulate the address register of ARM 710
module Address_register (data_out1, data_out2, data_in1, data_in2, data_in3,
  data_in4, write_en, sel, nBW, clk, reset_bar);

  // data output to memory
  output [31:0] data_out1;
  // data output to address incrementor
  output [31:0] data_out2;
  // input data from alu
  input [31:0] data_in1;
  // input data from register bank
  input [31:0] data_in2;
  // input data from address incrementer
  input [31:0] data_in3;
```

```
  // input data from Rn
  input [31:0] data_in4;
  // write_en control line
  input [0:0] write_en;
  // select line to select source of input data
  // 00 = alu
  // 01 = register
  // 10 = incrementer
  // 11 = Rn
  input [1:0] sel;
  // not Byte/Word
  input [0:0] nBW;
  // clock
  input [0:0] clk;
  input [0:0] reset_bar;

  reg [31:0] data_out1;
  reg [31:0] data_out2;

  // simulate address register behavior
  always @ (posedge clk)
  begin
    if (write_en && sel==2'b00) begin
//      $display("sel = 0");
      data_out1 = data_in1;
      data_out2 = data_in1;
    end
    if (write_en && sel==2'b01) begin
//      $display("sel = 1");
      data_out1 = data_in2;
      data_out2 = data_in2;
    end
    if (write_en && sel==2'b10) begin
//      $display("sel = 2");
      data_out1 = data_in3;
      data_out2 = data_in3;
    end
    if (write_en && sel==2'b11) begin
//      $display("sel = 3");
      data_out1 = data_in4;
      data_out2 = data_in4;
    end
  end

  // Byte or Word
  always @ (nBW) begin
    if (nBW) begin
      data_out1 = {data_out1[31:2], 2'b00};
      data_out2 = {data_out2[31:2], 2'b00};
    end
  end

  // reset the address register
  always @ (reset_bar) begin
    if (!reset_bar) begin
      // set address incrementer to 0
```

```
    data_out1 = 32'b00000000000000000000000000000000;
    data_out2 = 32'b00000000000000000000000000000000;
  end
 end

endmodule //of Address_register
```

ALU (Alu.v)

```
// This is a module to simulate the alu of ARM 710
module Alu (alu_out, carry, op1, op2, alu_mode, opcode);
 // output of alu
 output [31:0] alu_out;
 // output of carry bit
 output [0:0] carry;
 // operand1
 input [31:0] op1;
 // operand2
 input [31:0] op2;
 // alu mode
 input [0:0] alu_mode;
 // opcode
 input [3:0] opcode;

 wire [32:0] alu;
 wire [0:0] carry;

 // register for internal operations
 reg [31:0] op1_in;
 reg [31:0] op2_in;
 reg [31:0] oss;
 reg [31:0] sso;
 reg [31:0] oas;

 // simulate alu behavior
 always @ (opcode or op1 or op2) begin
 // 2's complement conversion
 if (op1[31]) begin
  op1_in = ~(op1 - 1);
 end
 else begin
  op1_in = op1;
 end
 if (op2[31]) begin
  op2_in = ~(op2 - 1);
  $display("op2_in=%b", op2_in);
 end
 else begin
  op2_in = op2;
 end

 if (op1[31] && op2[31]) begin
  oss = 0 - op1_in + op2_in;
  sso = 0 - op2_in - op1_in;
```

```
  oas = 0 - op1_in - op2_in;
end
else if (op1[31] && !op2[31]) begin
  oss = 0 - op1_in - op2_in;
  sso = op2_in + op1_in;
  oas = 0 - op1_in + op2_in;
end
else if (!op1[31] && op2[31]) begin
  oss = op1_in + op2_in;
  sso = 0 - op2_in - op1_in;
  oas = op1_in - op2_in;
  $display("oas=%b", oas);
end
else begin
  oss = op1_in - op2_in;
  sso = op2_in - op1_in;
  oas = op1_in + op2_in;
end

// Alu operations defined by instruction opcode
if (alu_mode) begin
// All the alu outputs will be written to Rn except 1000 to 1011
case (opcode)
  4'b0000: force alu = op1 & op2;
  4'b0001: force alu = op1 ^ op2;
  4'b0010: force alu = oss;
  4'b0011: force alu = sso;
  4'b0100: force alu = oas;
  4'b0101: force alu = oas + carry;
  4'b0110: force alu = oss + carry - 1;
  4'b0111: force alu = sso + carry - 1;
  // The following 4 operations will compute the logic and arithmetic result
  // and write it to condition codes
  4'b1000: force alu = op1 & op2;
  4'b1001: force alu = op1 ^ op2;
  4'b1010: force alu = oss;
  4'b1011: force alu = oas;
  4'b1100: force alu = op1 | op2;
  4'b1101: force alu = op2;
  4'b1110: force alu = op1 & ~op2;
  4'b1111: force alu = ~op2;
endcase
end

// Alu operations defined by Controller
else begin
case (opcode)
  // 0000 -> op1
  // 0001 -> op2 - 8
  // 0010 -> op1 - op2
  // 0011 -> op2 - op1
  // 0100 -> op1 + op2
  // 1101 -> op2
  // 1110 -> op2 + 8
  // 1111 -> op2 - op1 + 8
  4'b0000: force alu = op1;
```

```
    4'b0001: force alu = op2 - 8;
    4'b0010: force alu = oss;
    4'b0011: force alu = sso;
    4'b0100: force alu = oas;
    4'b0110: force alu = 33'b000000000000000000000000000000000;
    4'b1101: force alu = op2;
    4'b1110: force alu = op2 + 8;
    4'b1111: force alu = sso + 8;
   endcase
   end

   force alu_out = alu[31:0];
   force carry = alu[32];
   $display($time, " ALU: op1=%b, op2=%b", op1, op2);
   $display($time, " ALU: alu_mode=%b, opcode=%b, alu_out=%b", alu_mode, opcode, alu_out);
   end
endmodule //of alu
```

B bus (B_bus.v)

```
// This is a module to simulate the B bus of ARM 710
module B_bus (B_out, B_in1, B_in2, B_in3, B_in1_en, B_in2_en, B_in3_en);
  output [31:0] B_out;
  // input pins
  input [31:0] B_in1;
  input [31:0] B_in2;
  input [31:0] B_in3;
  // input enable pins
  input [0:0] B_in1_en;
  input [0:0] B_in2_en;
  input [0:0] B_in3_en;
  reg [31:0] B_out;

  initial begin
    B_out = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
  end

  always @ (B_in1 or B_in2 or B_in3 or B_in1_en or B_in2_en or B_in3_en) begin
  // only one input pins can be enable at any time
    if (B_in1_en && B_in2_en) begin
      $display("B_bus Enable Error");
      B_out = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
    end
    else if (B_in1_en && B_in3_en) begin
      B_out = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
    end
    else if (B_in2_en && B_in3_en) begin
      B_out = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
    end
    else if (B_in1_en) begin
      B_out = B_in1;
    end
    else if (B_in2_en) begin
      B_out = B_in2;
```

```
    end
    else if (B_in3_en) begin
      B_out = B_in3;
    end

//    $display($time, " B_bus: B_bus1=%b, B_bus2=%b, B_bus3=%b, B_en1=%d,
//  B_en2=%d, B_en3=%d", B_in1, B_in2, B_in3, B_in1_en, B_in2_en, B_in3_en);
  end

endmodule //of B_bus
```

Barrel shifter (Barrel_shifter.v)

```
// This is a module to simualate the Barrel shifter in ARM 710

// ROTATE FUNCTION SHOULD BE IMPLEMENTED ALSO
// If shift_format = 0000 -> mode1
// If shift_format = 0001 -> mode2
// If shift_format = 0010 -> immediate rotate
// If shift_format = 0011 -> 2 left shift
// If shift_format = 0100 -> immediate offset
// If shift_format = 0101 -> no shift
// If shift_format = 0110 -> byte shift
// If shift_format = 0111 -> word shift
// If shift_format = 1000 -> *2^n
// If shift_format = 1001 -> *2^(n+1)

module Barrel_shifter (barrel_out, barrel_carry, barrel_in, shift_info,
  rotate, immediate, address_offset, n, shift_format, shifter_en, clk);

  // output of barrel shifter
  output [31:0] barrel_out;
  // carry of barrel shifter
  output [0:0] barrel_carry;
  // input of barrel shifter
  input [31:0] barrel_in;
  // shift information define by bit 11 to 0 in instruction
  input [11:0] shift_info;
  // rotate amount;
  input [31:0] rotate;
  // immediate operand bit
  input [0:0] immediate;
  // Address offset
  input [1:0] address_offset;
  // n bit for multiplication
  input [4:0] n;
  // shift format
  input [3:0] shift_format;
  // shifter enable
  input [0:0] shifter_en;
  // clock
  input [0:0] clk;

  reg [31:0] barrel_out;
```

```
reg [0:0] barrel_carry;
reg [31:0] shifter;
reg [31:0] rotate_tmp;
integer i;

initial begin
  barrel_out = 32'b00000000000000000000000000000000;
  barrel_carry = 1'b0;
end

always @ (posedge clk) begin
if (shifter_en) begin
  shifter = barrel_in;

  // Shift mode 1
  if (shift_format == 4'b0000 || (shift_format == 4'b0100 && immediate)) begin
    // Logic Left
    if (shift_info[6:5] == 2'b00) begin
      barrel_out = (shifter << shift_info[11:7]);
    end
    // Logic Right, Arithmetic Right and Rotate Right
    else begin
      // Logic Right
      barrel_out = (shifter >> shift_info[11:7]);
      // Arthmetic Right
      if (shift_info[6:5] == 2'b10) begin
        for (i = 0; i < shift_info[11:7]; i = i + 1) begin
              barrel_out[31-i] = shifter[31];
            end
      end
      // Rotate Right
      else if (shift_info[6:5] == 2'b11 && shift_info[11:7] > 0) begin
        for (i = 0; i < shift_info[11:7]; i = i + 1) begin
              barrel_out[31-i] = shifter[shift_info[11:7]-1-i];
            end
      end
    end
    // print statement
    $display($time, "BS: Shift=0000");
  end

  // Shift mode 2
  else if (shift_format==4'b0001) begin
    rotate_tmp = rotate%32;
    if (rotate < 32 || (shift_info[6:5] == 2'b11 && rotate != 32)) begin
    // Logic Left
    if (shift_info[6:5] == 2'b00) begin
      barrel_out = (shifter << rotate_tmp);
    end
    // Logic Right, Arithmetic Right and Rotate_Tmp Right
    else begin
      // Logic Right
      barrel_out = (shifter >> rotate_tmp);
      // Arthmetic Right
      if (shift_info[6:5] == 2'b10) begin
        for (i = 0; i < rotate_tmp; i = i + 1) begin
```

```
            barrel_out[31-i] = shifter[31];
          end
    end
    // Rotate_Tmp Right
    else if (shift_info[6:5] == 2'b11 && rotate_tmp > 0) begin
      for (i = 0; i < rotate_tmp; i = i + 1) begin
            barrel_out[31-i] = shifter[rotate_tmp-1-i];
          end
    end
  end
  end
  else begin
    // Logic Left
    if (shift_info[6:5] == 2'b00) begin
            barrel_out = 32'b00000000000000000000000000000000;
      if (rotate_tmp == 32) begin
            barrel_carry = barrel_in[0];
          end
          else begin
            barrel_carry = 1'b0;
          end
    end
    // Logic Right, Arthmetic Right
    else if (shift_info[6:5] == 2'b01 || shift_info[6:5] == 2'b10) begin
            barrel_out = 32'b00000000000000000000000000000000;
      if (rotate_tmp == 32) begin
            barrel_carry = barrel_in[31];
          end
          else begin
            barrel_carry = 1'b0;
          end
    end
    // Rotate Right
    else if (shift_info[6:5] == 2'b11) begin
      barrel_out = barrel_in;
          barrel_carry = barrel_in[31];
    end
    $display($time, "BS: Shift=0001");
  end
end

// Immediate Rotate
else if (shift_format==4'b0010) begin
  barrel_out = {24'b0, shift_info[7:0]};
  barrel_out = barrel_out >> 2*shift_info[11:8];
  $display($time, " BS: Shift=0010");
end

else if (shift_format==4'b0011) begin
  if (barrel_in[23]) begin
    barrel_out = {8'b11111111, barrel_in[23:0]};
    barrel_out = barrel_out << 2;
  end
  else begin
    barrel_out = {8'b0, barrel_in[23:0]};
    barrel_out = barrel_out << 2;
```

```
    end
    $display($time, " BS: Shift=0011");
   end

   else if (shift_format==4'b0100) begin
    barrel_out = {24'b0, shift_info};
    $display($time, " BS: Shift=0100");
   end

   else if (shift_format==4'b0101) begin
    barrel_out = barrel_in;
    $display($time, " BS: Shift=0101");
   end

   else if (shift_format==4'b0110) begin
    barrel_out = barrel_in >> address_offset*8;
    $display($time, " BS: Shift=0110");
   end

   else if (shift_format==4'b0111) begin
    if (address_offset==0) begin
      barrel_out = barrel_in;
    end
    else if (address_offset==1) begin
      barrel_out[31:8] = barrel_in[23:0];
          barrel_out[7:0] = barrel_in[31:24];
    end
    else if (address_offset==2) begin
          barrel_out[31:16] = barrel_in[15:0];
          barrel_out[15:0] = barrel_in[31:16];
    end
    else begin
          barrel_out[31:24] = barrel_in[7:0];
          barrel_out[7:0] = barrel_in[31:24];
    end
    $display($time, " BS: Shift=0111");
   end

   else if (shift_format==4'b1000) begin
    barrel_out = barrel_in << 2*n;
    $display($time, " BS: Shift=1000");
   end

   else if (shift_format==4'b1001) begin
    barrel_out = barrel_in << (2*n+1);
    $display($time, " BS: Shift=1001");
   end

   $display($time, " Barrel_Shifter: in=%b, out=%b", barrel_in, barrel_out);

  end
  end

endmodule //of Barrel_shifter
```

Top level module (Block.v)

```
// This is a module to simulate ARM 710
module Block (A_out, nRW, nBW, nMREQ, D, CLK, RESET_BAR);
  // output address for memory request
  output [31:0] A_out;
  // Memory signals
  output [0:0] nRW;
  output [0:0] nBW;
  output [0:0] nMREQ;
  // in and out to memory
  inout [31:0] D;
  // clock
  input [0:0] CLK;
  // reset bar
  input [0:0] RESET_BAR;

  // alu_bus --> output from alu, 1st input to address register and 1st
  //             input to register bank
  // alu_carry --> output from carry bit
  // a_bus --> 1st input of alu, 1st output of register bank
  // op2 --> 2nd input of alu, output of barrel_shifter
  // b_bus3 --> 2nd output of register bank
  // incr_bus --> output of incrementer, 3rd input of address register and
  //              2nd input of register bank
  // pc_bus --> 3rd output of register bank, 2nd input to address register
  // addr_r_out2 --> connect address register and address incrementer
  // b_bus1 --> data from data register to b_bus
  // ipr_out --> data to b_bus from instruction pipeline register
  // dr_out --> data to b_bus from read data register
  // instr1 --> instruction output from instruction register 1
  // b_bus2 --> instruction data to b_bus
  // b_out --> output of b_bus
  // cpsr --> output from cpsr to control
  // cpsr_flag  --> cpsr_flag store to cpsr from control
  // Rs_out --> Rotate register from register bank to shifter

  // addr_r_w_en --> address register write enable
  // addr_r_sel --> address register select
  // incr_en --> incrementer enable
  // shift amount --> barrel shifter shift amount
  // shift type --> barrel shifter shift type
  // shifter enable --> enable shifter
  // rb_w_en1 --> register bank write enable1
  // rb_w_en2 --> register bank write enable2
  // register numbers -->  reg_num_in1, reg_num_in2, reg_num_out1, reg_num_out2
  //                       reg_num_out3
  // opcode --> input of alu
  // wdr_w_en --> write data register write enable
  // rdr_w_en --> read data register write enable
  // ipr_w_en --> instruction pipeline register write enable
  // ipr_shift --> instruction pipeline register shift control
  // ipr_w_en --> instruction pipeline register write enable
  // ipr_shift --> instruction shift enable
  // b_en1 --> bus enable1
```

```
// b_en2 --> bus enable2
// b_en3 --> bus enable3
// cpsr_w_en --> cpsr write enable
// l_en1 --> latch enable 1

wire [31:0] alu_bus;
wire [0:0] alu_carry;
wire [31:0] a_bus;
wire [31:0] ra_bus;
wire [31:0] op2;
wire [31:0] b_bus3;
wire [31:0] incr_bus;
wire [31:0] pc_bus;
wire [0:0] barrel_carry;
wire [31:0] addr_r_out2;
wire [31:0] b_bus1;
wire [31:0] ipr_out;
wire [31:0] dr_out;
wire [31:0] instr1;
wire [31:0] b_bus2;
wire [31:0] b_out;
wire [31:0] cpsr;
wire [3:0] cpsr_flag;
wire [31:0] Rs_out;

wire [0:0] addr_r_w_en;
wire [1:0] addr_r_sel;
wire [0:0] incr_en;
wire [3:0] shift_format;
wire [0:0] shifter_en;
wire [0:0] rb_w_en1;
wire [0:0] rb_w_en2;
wire [3:0] reg_num_in1;
wire [3:0] reg_num_in2;
wire [3:0] reg_num_out1;
wire [3:0] reg_num_out2;
wire [0:0] alu_mode;
wire [0:0] mux2_32_sel;
wire [4:0] count_list;
wire [3:0] opcode;
wire [0:0] wdr_w_en;
wire [0:0] rdr_w_en;
wire [0:0] ipr_w_en;
wire [0:0] ipr_shift;
wire [0:0] b_en1;
wire [0:0] b_en2;
wire [0:0] b_en3;
wire [0:0] cpsr_w_en;
wire [0:0] l_en1;
wire [31:0] alu_out;
wire [2:0] booth_case;
wire [0:0] borrow;
wire [4:0] n;
wire [0:0] b_stop;
wire [0:0] booth_en;
wire [0:0] n_incr;
```

```
  Address_register ar (A_out, addr_r_out2, alu_bus, pc_bus, incr_bus, a_bus,
  addr_r_w_en, addr_r_sel, nBW, CLK, RESET_BAR);

  Address_incrementer ai (incr_bus, addr_r_out2, incr_en, CLK);

  // need a mux for shift amount --------|

  Barrel_shifter bas (op2, barrel_carry, b_out, b_bus2[11:0], Rs_out,
  b_bus2[25], A_out[1:0], n, shift_format, shifter_en, CLK);

  Register_bank rb (ra_bus, b_bus3, Rs_out, pc_bus, cpsr, alu_bus,
  incr_bus, cpsr_flag , rb_w_en1, rb_w_en2, cpsr_w_en,  reg_num_in1, reg_num_in2,
  reg_num_out1, reg_num_out2, b_bus2[11:8], CLK, RESET_BAR);

  Booth_shifter bos (booth_case, borrow, n, b_stop, a_bus, booth_en, n_incr, CLK);

  Alu alu (alu_out, alu_carry, a_bus, op2, alu_mode, opcode);

  Write_data_register wdr (D, b_out, A_out[1:0], nBW, wdr_w_en, CLK, RESET_BAR);

  Read_data_register rdr (b_bus1, D, rdr_w_en, CLK, RESET_BAR);

  Instruction_Pipeline_register ipr (instr1, b_bus2, D,
  ipr_w_en, ipr_shift, CLK, RESET_BAR);

  B_bus b_bus (b_out, b_bus1, b_bus2, b_bus3, b_en1, b_en2, b_en3);

  Latch l1(alu_bus, alu_out, l_en1);

  Mux2_32 mux2_32 (a_bus, ra_bus, {27'b0, count_list}, mux2_32_sel);

  Control c (addr_r_w_en, addr_r_sel, incr_en, shift_format, shifter_en,
  rb_w_en1, rb_w_en2, reg_num_in1, reg_num_in2, reg_num_out1, reg_num_out2,
  alu_mode, mux2_32_sel, count_list, opcode, wdr_w_en, rdr_w_en, ipr_w_en,
  ipr_shift, b_en1, b_en2, b_en3, l_en1, cpsr_w_en, cpsr_flag, nRW, nBW, nMREQ,
  booth_en, n_incr, instr1, b_bus2, alu_bus, barrel_carry, booth_case,
  borrow, b_stop, cpsr, CLK, RESET_BAR);

endmodule //of Block
```

Booth shifter (Booth_shifter.v)

```
// This is a module to simulate the Booth's alogorithm shifter
module Booth_shifter (booth_case, borrow, n, b_stop, booth_in, booth_en,
  n_incr, clk);
  // Cases defined for alu operations
  // borrow = 0; op2%4 = 0; case0
  // borrow = 0; op2%4 = 1; case1
  // borrow = 0; op2%4 = 2; case2
  // borrow = 0; op2%4 = 3; case3
  // borrow = 1; op2%4 = 0; case4
  // borrow = 1; op2%4 = 1; case5
  // borrow = 1; op2%4 = 2; case6
```

```
// borrow = 1; op2%4 = 3; case7
output [2:0] booth_case;
// borrow bit
output [0:0] borrow;
// n bit (2-bit shift per internal cycle)
output [4:0] n;
// booth's shift stop signal, notice Controller the end of multiplication
output [0:0] b_stop;
// input value
input [31:0] booth_in;
// booth shifter enable
input [0:0] booth_en;
// increment enable of n bit
input [0:0] n_incr;
input [0:0] clk;

reg [31:0] booth_reg;
reg [2:0] booth_case;
reg [0:0] borrow;
reg [4:0] n;
reg [0:0] b_stop;

// initialization
initial begin
  booth_reg = 32'b00000000000000000000000000000000;
  booth_case = 3'b000;
  borrow = 1'b0;
  n = 1'b0;
  b_stop = 1'b0;
end

// define Booth's Algorithm cases
always @ (borrow or n or booth_reg) begin
    if (borrow == 0) begin
      case (booth_reg[1:0])
            2'b00: booth_case = 0;
            2'b01: booth_case = 1;
            2'b10: booth_case = 2;
            2'b11: booth_case = 3;
          endcase
    end
    else begin
      case (booth_reg[1:0])
            2'b00: booth_case = 4;
            2'b01: booth_case = 5;
            2'b10: booth_case = 6;
            2'b11: booth_case = 7;
          endcase
    end
end

// Booth's Algorithm
always @ (posedge clk) begin
  if (booth_en) begin
    if (n == 0) begin
      booth_reg = booth_in;
```

```
            b_stop = 0;
        $display($time, " BOS: booth_in=%b, booth_reg=%b" , booth_in, booth_reg);
      end
    end
  end

  // n bit incrementation
  always @ (posedge clk) begin
    if (n_incr) begin
      if (borrow == 0) begin
       case (booth_reg[1:0])
              2'b10: borrow = 1;
              2'b11: borrow = 1;
            endcase
      end
      else begin
        case (booth_reg[1:0])
              2'b00: borrow = 0;
              2'b01: borrow = 0;
            endcase
      end
      n = n + 1;
      booth_reg = booth_reg/4;
    end
  end

  // b_stop signal
  always @ (n or booth_reg or borrow) begin
    if ((booth_reg != 0 || borrow == 1) && n < 16) begin
      $display("Booth_shifter: Carry on >>>>");
      b_stop = 0;
    end
    else begin
      n = 0;
      b_stop = 1;
    end
  end

endmodule //of Booth_shifter
```

Clock (Clock.v)

```
// This is a module to simulate clock
module Clock (clk, clk_bar, reset_bar);
  output [0:0] clk;
  output [0:0] clk_bar;
  input [0:0] reset_bar;

  parameter p=20;  // 25 MHZ clock

  initial begin
    force clk = 0;
  end
```

```
  //generate clock
  always begin
    if (!reset_bar) begin
      force clk = 0;
      force clk = 1;
    end
    #(p/2) force clk = 1;
    #(p/2) force clk = 0;
  end

endmodule //of Clock
```

Condition decoder (Condition_decoder.v)

```
// This is a module for Condition Field Decoder in ARM 710
module Condition_decoder (execute, cond_field, cpsr, CLK);
  // 1 -> execute the instruction
  // 0 -> abort the instruction, fetch and decode next instruction
  output [0:0] execute;
  // conditional field from instruction
  input [3:0] cond_field;
  // Current Program Status Register
  input [3:0] cpsr;
  // Clock input
  input [0:0] CLK;
  reg [0:0] execute;

  // Decoder Delay
  parameter dd = 10;

  initial begin
    execute = 1'b0;
  end

  // Call tasks to check bits set
  always @ (cond_field or cpsr or CLK) begin
  #dd
  case (cond_field)
    // check equal
    4'b0000: check_set (execute, cpsr[2]);
    // check not equal
    4'b0001: check_clear (execute, cpsr[2]);
    // check unsigned higher or same
    4'b0010: check_set (execute, cpsr[1]);
    // check unsigned lower
    4'b0011: check_clear (execute, cpsr[1]);
    // check negative
    4'b0100: check_set (execute, cpsr[3]);
    // check positive or zero
    4'b0101: check_clear (execute, cpsr[3]);
    // check overflow
    4'b0110: check_set (execute, cpsr[0]);
    // check no overflow
    4'b0111: check_clear (execute, cpsr[0]);
```

```
    // check unsigned higher
    4'b1000: hi (execute, cpsr);
    // unsigned lower or same
    4'b1001: ls (execute, cpsr);
    // greater or equal
    4'b1010: ge (execute, cpsr);
    // less than
    4'b1011: lt (execute, cpsr);
    // greater than
    4'b1100: gt (execute, cpsr);
    // less than or equal
    4'b1101: le (execute, cpsr);
    // always
    4'b1110: execute = 1;
    // never
    4'b1111: execute = 0;
    // default case
    default: $display("Unsolved Conditonal Field");
  endcase
    $display($time, " CD: cond_field=%b, cpsr=%b, execute=>%b", cond_field, cpsr, execute);
  end

// Define checking bit task
task check_set;
  output [0:0] execute;
  input [0:0] cpsr_bit;
  begin
    if (cpsr_bit) begin
      execute = 1;
    end
    else begin
      execute = 0;
    end
  end
endtask //of check_set

task check_clear;
  output [0:0] execute;
  input [0:0] cpsr_bit;
  begin
    if (cpsr_bit) begin
      execute = 0;
    end
    else begin
      execute = 1;
    end
  end
endtask //of check_clear

task hi;
 output [0:0] execute;
 input [3:0] cpsr;
 begin
   if (!cpsr[2] && cpsr[1]) begin
     execute = 1;
   end
```

```
   else begin
     execute = 0;
   end
 end
endtask //of hi

task ls;
 output [0:0] execute;
 input [3:0] cpsr;
 begin
   if (cpsr[2] && !cpsr[1]) begin
     execute = 1;
   end
   else begin
     execute = 0;
   end
 end
endtask //of ls

task ge;
 output [0:0] execute;
 input [3:0] cpsr;
 begin
   if ((cpsr[3] && cpsr[0]) || (!cpsr[3] && !cpsr[0])) begin
     execute = 1;
   end
   else begin
     execute = 0;
   end
 end
endtask //of ge

task lt;
 output [0:0] execute;
 input [3:0] cpsr;
 begin
   if ((cpsr[3] && !cpsr[0]) || (!cpsr[3] && cpsr[0])) begin
     execute = 1;
   end
   else begin
     execute = 0;
   end
 end
endtask //of lt

task gt;
 output [0:0] execute;
 input [3:0] cpsr;
 begin
   if (!cpsr[2] && ((cpsr[3] && cpsr[0]) || (!cpsr[3] && !cpsr[0]))) begin
     execute = 1;
   end
   else begin
     execute = 0;
   end
 end
```

```
endtask //of gt

task le;
 output [0:0] execute;
 input [3:0] cpsr;
 begin
   if (cpsr[2] && ((cpsr[3] && !cpsr[0]) || (!cpsr[3] && cpsr[0]))) begin
     execute = 1;
   end
   else begin
     execute = 0;
   end
 end
endtask //of le

endmodule //of Condition_decoder
```

Control (Control.v)

```
// This is a module to simulate the Control in ARM 710
module Control (addr_r_w_en, addr_r_sel, incr_en,
  shift_format, shifter_en, rb_w_en1, rb_w_en2, reg_num_in1,
  reg_num_in2, reg_num_out1, reg_num_out2, mux2_sel, mux2_32_sel,
  count_list, opcode, wdr_w_en, rdr_w_en, ipr_w_en, ipr_shift, b_en1,
  b_en2, b_en3, l_en1, cpsr_w_en, cpsr_flag, nRW, nBW, nMREQ, booth_en,
  n_incr, instr1, instr_exe, alu_out, barrel_carry, booth_case, borrow,
  b_stop, cpsr, CLK, RESET_BAR);

  // addr_r_w_en --> address register write enable
  output [0:0] addr_r_w_en;
  // addr_r_sel --> address register select
  output [1:0] addr_r_sel;
  // incrementer enable
  output [0:0] incr_en;
  // shift format
  output [3:0] shift_format;
  // shifter enable
  output [0:0] shifter_en;
  // register bank write enable 1
  output [0:0] rb_w_en1;
  // register bank write enable 2
  output [0:0] rb_w_en2;
  // 1st register num writing to register bank
  output [3:0] reg_num_in1;
  // 2nd register num writing to register bank
  output [3:0] reg_num_in2;
  // 1st register num reading from register bank
  output [3:0] reg_num_out1;
  // 2nd register num reading from register bank
  output [3:0] reg_num_out2;
  // mux and alu operation mode
  output [0:0] mux2_sel;
  // mux choose the alu op1
  output [0:0] mux2_32_sel;
```

```
// counter list
output [4:0] count_list;
// opcode to alu
output [3:0] opcode;
// write data register enable
output [0:0] wdr_w_en;
// read data register write enable
output [0:0] rdr_w_en;
// instruction pipeline write enable
output [0:0] ipr_w_en;
// instruction pipeline shift enable
output [0:0] ipr_shift;
// b_bus enable
output [0:0] b_en1;
output [0:0] b_en2;
output [0:0] b_en3;
// latch enable
output [0:0] l_en1;
// current program status register write enable
output [0:0] cpsr_w_en;
// current program status register bit change
output [3:0] cpsr_flag;
// not Read/Write (Memory)
output [0:0] nRW;
// not Byte/Word (Memory)
output [0:0] nBW;
// not Memory Request
output [0:0] nMREQ;
// booth's shifter
output [0:0] booth_en;
output [0:0] n_incr;
// instruction 1
input [31:0] instr1;
// instruction in execution
input [31:0] instr_exe;
// alu output
input [31:0] alu_out;
// carry out of barrel shifter
input [0:0] barrel_carry;
// current program status register
input [31:0] cpsr;
// booth shifter
input [2:0] booth_case;
input [0:0] borrow;
input [0:0] b_stop;
// clock
input [0:0] CLK;
input [0:0] RESET_BAR;

// condition decode execute enable
wire [0:0] execute;
// Instruction decode result
wire [3:0] in_instr_type;

// Output of decoder register to pla
wire [3:0] instr_type;
```

```
  // Mux selects
  wire [2:0] mux5_sel;
  wire [1:0] mux4a_sel;
  wire [1:0] mux4b_sel;
  // register number
  wire [3:0] reg_in1;
  wire [3:0] reg_in2;
  wire [3:0] reg_out1;
  wire [3:0] reg_out2;
  // pla opcode line
  wire [3:0] pla_opcode;
  // state register
  wire [7:0] present;
  wire [7:0] next;
  // register list
  wire [0:0] end_list;
  wire [0:0] rld_shift;
  wire [3:0] Rp;

  // State Decoder
  wire [7:0] next_state_in;
  wire [0:0] sd_sel;

  Condition_decoder cd (execute, instr1[31:28], cpsr[31:28], CLK);

  Decoder_register dr (instr_type, in_instr_type, ipr_shift, CLK);
  Instruction_decoder id (instr1, in_instr_type);

   Pla pla ({addr_r_w_en, addr_r_sel, incr_en, //0-3{1,2,1}
               shift_format, shifter_en, rb_w_en1, rb_w_en2, reg_in1, //4-14{4,1,1,1,4}
               reg_num_in2, reg_out1, reg_out2, mux2_sel, mux2_32_sel, mux4b_sel, //15-30{4,4,4,1,1,2}
               mux4a_sel, mux5_sel, pla_opcode, wdr_w_en, rdr_w_en, ipr_w_en, ipr_shift,//31-
43{2,3,4,1,1,1,1}
               rld_shift, b_en1, b_en2, b_en3, l_en1, cpsr_w_en, sd_sel, nRW,//44-51{1,1,1,1,1,1,1,1}
               nBW, nMREQ, booth_en, n_incr, next_state_in},//52-63{1,1,1,1,8}
              {present, execute, end_list,//0-9{8,1,1}
               instr_exe[25:20], instr_exe[15:12],instr_exe[4], instr_type, booth_case, b_stop});//10-
28{6,4,1,4,3,1}

  State_register sr (present, next, CLK, RESET_BAR);

  Mux2 mux2 (opcode, pla_opcode, instr_exe[24:21], mux2_sel);
  Mux4 mux4a (reg_num_in1, reg_in1, instr_exe[15:12], instr_exe[19:16], Rp,
  mux4a_sel);
  Mux4 mux4b (reg_num_out1, reg_out1, instr_exe[19:16], instr_exe[11:8],
  instr_exe[15:12], mux4b_sel);
  Mux5 mux5 (reg_num_out2, reg_out2, instr_exe[3:0], instr_exe[15:12],
  instr_exe[19:16], Rp, mux5_sel);

  Register_list_decoder rld (Rp, count_list, end_list, instr_exe[15:0],
  rld_shift, CLK );

  State_decoder sd (next, next_state_in, instr_type, sd_sel);

  Cpsr_set_control csc (cpsr_flag, cpsr[31:28], reg_num_in1, instr_exe[20],
```

```
  barrel_carry, alu_out);

endmodule //of Control
```

Condition code flag control (Cpsr_set_control.v)

```
// This is a module to simulate ARM 710 cpsr set flag control
module Cpsr_set_control (cpsr_out, cpsr_in, Rd, s, barrel_carry, alu_out);
  output [3:0] cpsr_out;
  // Input to determine cpsr set
  input [3:0] cpsr_in;
  input [3:0] Rd;
  input [0:0] s;
  input [0:0] barrel_carry;
  input [31:0] alu_out;

  reg [3:0] cpsr_out;
  // 3 2 1 0
  // n z c v

  initial begin
    cpsr_out = 4'b0000;
  end

  always @ (Rd or s or barrel_carry or alu_out) begin
    cpsr_out = cpsr_in;
    // set carry bit
    if (s && Rd != 4'b1111) begin
      cpsr_out = {cpsr_out[3:2], barrel_carry, cpsr_out[0]};
    end
    // set zero bit
    if (!alu_out) begin
      cpsr_out = {cpsr_out[3], 1'b1, cpsr_out[1:0]};
    end
    // set n bit
    cpsr_out = {alu_out[31], cpsr_out[2:0]};
    $display($time, " CPSR_SET: cpsr = %b", cpsr_out);
  end

endmodule //of Cpsr_set_control
```

Decoder register (Decoder_register.v)

```
// This is a module to simulate a Decoder Register in ARM 710
module Decoder_register (instr_type, in_instr_type, shift, clk);
  // Instruction decoder output
  // for instruction in executing stage
  output [3:0] instr_type;

  // Instruction decoder input
  // from instruction decoder in decoding stage
  input [3:0] in_instr_type;
```

```
  // shift enable
  input [0:0] shift;
  input [0:0] clk;

  reg [3:0] instr_type;

  always @ (posedge clk) begin
    if (shift) begin
      instr_type = in_instr_type;
    end
  end

endmodule //of Decoder_register
```

Instruction decoder Instruction_decoder.v)

```
// This is a module to simulate the instruction decoder in ARM 710
module Instruction_decoder (instr, instr_type);
  // input instruction
  input [31:0] instr;
  // defined instruction type
  output [3:0] instr_type;
  reg [3:0] instr_type;
  parameter Instruction_decoder_delay = 0;

    // instr_type = 0000 shift format1
    // instr_type = 0001 shift format2
    // instr_type = 0010 immediate
    // instr_type = 0011 Branching
    // instr_type = 0100 Single Data Transfer
    // instr_type = 0101 Single Data Swap
    // instr_type = 0110 Multiply and Multiply-Accumulate
    // instr_type = 0111 Block Data Transfer

  initial begin
    instr_type = 4'b0000;
  end

  always @ (instr)
  #Instruction_decoder_delay

  begin
  // use  case to decode instructions
  casez (instr)
    // 0 Data Processing/PSR Transfer (no set bit, shift format1)
    32'b????0000????????????????????0????: instruction0(instr, instr_type);
    // 1 Data Processing/PSR Transfer (no set bit, shift format2)
    32'b????0000???????????????0??1????: instruction0(instr, instr_type);
    // 0 Data Processing/PSR Transfer (no set bit, Immediate)
    32'b????0010????????????????????????????: instruction0(instr, instr_type);
    // 2 Data Processing/PSR Transfer (no set bit, shift format1)
    32'b????00?11????????????????????0????: instruction0(instr, instr_type);
    // 3 Data Processing/PSR Transfer (no set bit, shift format2)
```

```
    32'b????00?11??????????????0??1????: instruction0(instr, instr_type);
    // 4 Data Processing/PSR Transfer (with set bit)
    32'b????00?10??1??????????????????????: instruction0(instr, instr_type);
    // Branch
    32'b????101????????????????????????????: instruction1(instr, instr_type);
    // Single Data Transfer
    32'b????01????????????????????????????: instruction2(instr, instr_type);
    // Single Data Swap
    32'b????00010???????????00001001????: instruction3(instr, instr_type);
    // Multiply
    32'b????000000??????????????1001????: instruction4(instr, instr_type);
    // Block Data Transfer
    32'b????100????????????????????????????: instruction5(instr, instr_type);

    // Invalid instruction
    default instruction_invalid(instr);
  endcase
//    $display("instr=%b", instr);
  end

// Data Processing
task instruction0;
  input [31:0] instr;
  output [3:0] instr_type;
  begin
    // instr_type = 0000 shift format1
    // instr_type = 0001 shift format2
    // instr_type = 0010 immediate
    if (instr[25]) begin
      instr_type = 4'b0010;
      $display("Visit instr_type 0010");
    end
    else begin
      if (instr[4]) begin
        instr_type = 4'b0001;
      end
      else begin
            instr_type = 4'b0000;
      end
    end
    $display($time, " ID: instruction0 (Data Processing)");
    $display($time, " ID: instr_type = %b", instr_type);
  end
endtask //of instruction0

// Branch Branching
task instruction1;
  input [31:0] instr;
  output [3:0] instr_type;
  begin
    // instr_type = 0011 Branching
    instr_type = 4'b0011;
    $display("Visit instr_type 0011");
    $display($time, " ID: instruction1 (Branching)");
    $display($time, " ID: instr_type = %b", instr_type);
  end
```

```
endtask //of instruction1

// Single Data Transfer
task instruction2;
  input [31:0] instr;
  output [3:0] instr_type;
  begin
    // instr_type = 0100 Single Data Transfer
    instr_type = 4'b0100;
    $display("Visit instr_type %b", instr_type);
    $display($time, " ID: instruction2 (Single Data Transfer)");
    $display($time, " ID: instr_type = %b", instr_type);
  end
endtask //of instruction2

// Single Data Swap
task instruction3;
  input [31:0] instr;
  output [3:0] instr_type;
  begin
    instr_type = 4'b0101;
    $display("Visit instr_type %b", instr_type);
    $display($time, " ID: instruction3 (Single Data Swap)");
    $display($time, " ID: instr_type = %b", instr_type);
  end
endtask //of instruction3

// Multiply and Multiply-Accumulate
task instruction4;
  input [31:0] instr;
  output [3:0] instr_type;
  begin
    instr_type = 4'b0110;
    $display("Visit instr_type %b", instr_type);
    $display($time, " ID: instruction4 (Multiply and Multiply-Accumulate)");
    $display($time, " ID: instr_type = %b", instr_type);
  end
endtask //of instruction4

// Block Data Transfer
task instruction5;
  input [31:0] instr;
  output [3:0] instr_type;
  begin
    instr_type = 4'b0111;
    $display("Visit instr_type %b", instr_type);
    $display($time, " ID: instruction5 (Block Data Transfer)");
    $display($time, " ID: instr_type = %b", instr_type);
  end
endtask //of instruction5

task instruction_invalid;
  input [31:0] instr;
  $display("----------instruction invalid!!!!!!!!!!!!!!!----------");
endtask //of instruction_invalid
```

```
endmodule //of Instruction_decoder
```

Instruction pipeline register (Instruction_Pipeline_register.v)

```
// This is a module to simulate the Instruction Pipeline Register in ARM 710
module Instruction_Pipeline_register (instr1, instr_data_out, instr_data_in,
  write_en, shift, clk, reset_bar);
  // instruction in decoding stage
  output [31:0] instr1;
  // instruction in executing stage
  output [31:0] instr_data_out;
  // input of new instruction
  input [31:0] instr_data_in;
  // input instruction write enable
  input [0:0] write_en;
  // Instruction pipeline shift enable
  input [0:0] shift;
  input [0:0] clk;
  input [0:0] reset_bar;
  // connect Instruction Data Register and Pipeline Register
  wire [31:0] instr;

  Pipeline_register pr (instr1, instr_data_out, instr, shift, clk, reset_bar);
  Instruction_register ir (instr, instr_data_in, write_en, clk, reset_bar);

endmodule //of Instruction_Pipeline_register


// This is a module to simulate the Pipeline registers in ARM 710
module Pipeline_register (instr1, instr2, instr, shift, clk, reset_bar);
  // instruction decoding
  output [31:0] instr1;
  // instruction in execution
  output [31:0] instr2;
  // instruction input from instruction and data register
  input [31:0] instr;
  // enable for shifting
  input [0:0] shift;
  input [0:0] clk;
  input [0:0] reset_bar;
  // instruction registers
  reg [31:0] instr1;
  reg [31:0] instr2;

  // shift at positive clk
  always @ (posedge clk)
  begin
    if (shift) begin
      instr2 = instr1;
      instr1 = instr;
      $display($time, " IPR: instr=%b, instr1=%b, instr2=%b", instr, instr1, instr2);
    end
  end
  // reset
```

```
  always @ (reset_bar) begin
    if (!reset_bar) begin
      instr1 = 32'b00000000000000000000000000000000;
      instr2 = 32'b00000000000000000000000000000000;
    end
  end
endmodule //of Pipeline_register


// This is a module to simulate instruction register of ARM 710
module Instruction_register (data_out, data_in, write_en, clk, reset_bar);
  // data out to instruction decoder
  output [31:0] data_out;
  // data in from memory
  input [31:0] data_in;
  // write enable line
  input [0:0] write_en;
  // clock
  input [0:0] clk;
  input [0:0] reset_bar;
  // instruction register
  reg [31:0] data_out;

  // Data in
  always @ (posedge clk)
  begin
    if (write_en) begin
      data_out = data_in;
    end
  end
  // reset
  always @ (reset_bar) begin
    if (!reset_bar) begin
      data_out = 32'b00000000000000000000000000000000;
     end
  end

endmodule //of Instruction_register
```

Latch, 32-bit (Latch.v)

```
// This is a simulation of a 32-bit latch
module Latch (out, in, en);
  output [31:0] out;
  input [31:0] in;
  // enable line
  input [0:0] en;

  reg [31:0] out;
  always @ (in or en) begin
    if (en) begin
      out = in;
    end
  end
```

```
endmodule //of Latch
```

Mux – 2 to 1, 4 bits (Mux2.v)

```
// This is a module to simulate a 2 to 1 Mux
module Mux2 (out, in0, in1, sel);
  output [3:0] out;
  input [3:0] in0;
  input [3:0] in1;
  input [0:0] sel;
  wire [3:0] out;

  always @ (sel or in0 or in1) begin
    if (!sel) begin
      force out = in0;
    end
    else begin
      force out = in1;
    end
//  $display("in0=%b, in1=%b, sel=%b", in0, in1, sel);
  end

endmodule //of Mux2
```

Mux – 2 to 1, 32 bits (Mux2_32.v)

```
// This is a module to simulate a 2 to 1 Mux
module Mux2_32 (out, in0, in1, sel);
  output [31:0] out;
  input [31:0] in0;
  input [31:0] in1;
  input [0:0] sel;
  wire [31:0] out;

  always @ (sel or in0 or in1) begin
    if (!sel) begin
      force out = in0;
    end
    else begin
      force out = in1;
    end
//  $display("in0=%b, in1=%b, sel=%b", in0, in1, sel);
  end

endmodule //of Mux2_32
```

Mux – 3 to 1, 4 bits (Mux3.v)

```
// This is a module to simulate a 3 to 1 Mux
```

```
module Mux3 (out, in0, in1, in2, sel);
  output [3:0] out;
  input [3:0] in0;
  input [3:0] in1;
  input [3:0] in2;
  input [1:0] sel;
  wire [3:0] out;

  always @ (sel or in0 or in1 or in2) begin
    if (sel == 0) begin
      force out = in0;
    end
    else if (sel == 1) begin
      force out = in1;
    end
    else begin
      force out = in2;
    end
  end

endmodule //of Mux3
```

Mux – 4 to 1, 4 bits (Mux4.v)

```
// This is a module to simulate a 4 to 1 Mux
module Mux4 (out, in0, in1, in2, in3, sel);
  output [3:0] out;
  input [3:0] in0;
  input [3:0] in1;
  input [3:0] in2;
  input [3:0] in3;
  input [1:0] sel;
  wire [3:0] out;

  always @ (sel or in0 or in1 or in2 or in3) begin
    if (sel == 0) begin
      force out = in0;
    end
    else if (sel == 1) begin
      force out = in1;
    end
    else if (sel == 2) begin
      force out = in2;
    end
    else begin
      force out = in3;
    end
  end

endmodule //of Mux4
```

Mux – 5 to 1, 4 bits (Mux5.v)

```
// This is a module to simulate a 5 to 1 Mux
module Mux5 (out, in0, in1, in2, in3, in4, sel);
  output [3:0] out;
  input [3:0] in0;
  input [3:0] in1;
  input [3:0] in2;
  input [3:0] in3;
  input [3:0] in4;
  input [2:0] sel;
  wire [3:0] out;

  always @ (sel or in0 or in1 or in2 or in3 or in4) begin
    if (sel == 0) begin
      force out = in0;
    end
    else if (sel == 1) begin
      force out = in1;
    end
    else if (sel == 2) begin
      force out = in2;
    end
    else if (sel == 3) begin
      force out = in3;
    end
    else begin
      force out = in4;
    end
  end

endmodule //of Mux5
```

PLA (Pla.v)

```
// This is a module to simulate the control unit PLA of ARM 710
module Pla (Pla_out, Pla_in);
  output [63:0] Pla_out;
  input [28:0] Pla_in;
  reg [3:0] cpsr;

  // Use case statement to represent control bits
  always @ (Pla_in) begin
//    $display($time, "***** Pla_in change %b *****", Pla_in);

    casex (Pla_in)
      // From Reset state 0 -> 1
      // Memory request
      29'b00000000xxxxxxxxxxxxxxxxxxxxx: begin
      force Pla_out =
      64'b0000000000000000000000000000000000000000000000000001000100000000001;
      $display($time, " line0->1");
      end
```

```
// From state 1 -> 2
// Memory request, AI++, IR=MEM[AR]
29'b00000001xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001000000000000000000000000000000000000000100000100010000000010;
$display($time, " line1->2");
end

// From state 2 -> 3
// Memory request, IPR shift, AI -> PC, AI -> AR
29'b00000010xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000000100001111000000000000000000000000100001000100000000011;
$display($time, " line2->3");
end

// From state 3 -> 4
// Conditional code pass
// Memory request, AI++, IR=MEM[AR], Decoding
29'b00000011xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001000000000000000000000000000000000000000100000100010000000100;
$display($time, " line3->4");
end

// From state 4 -> 3
// Conditional code fail
// Memory request, IPR shift, AI -> PC, AI -> AR, Decoding
29'b000001000xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000000100001111000000000000000000000000100001000100000000011;
$display($time, " line4->3");
end

// DATA PROCESSING ***********************************************
// ***************************************************************

// From 5 -> 10 (no set bit, R15)
// This is done first to determine if Rd=>R[15] in verilog
// However in real Pla, 4 checking lines should be put to 5 -> 8 state
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000101xx0xxxx0111100000xxxx: begin
force Pla_out =
64'b0001000010000000000000000010010000100000010000110001000000001010;
$display($time, " line5->10");
end

// From state 5 -> 8 (no set bit, !R15)
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000101xx0xxxx0xxxx00000xxxx: begin
force Pla_out =
64'b0001000010000000000000000010010000100000010000110001000000001000;
$display($time, " line5->8");
```

```
end

// From state 5 -> 9 (set bit)
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000101xx0xxxx1xxxx00000xxxx: begin
force Pla_out =
64'b0001000010000000000000000001001000010000001000011100100000001001;
$display($time, " line5->9");
end

// ****************************************************************

// From 6 -> 10 (no set bit, R15)
// This is done first to determine if Rd=>R[15] in verilog
// However in real Pla, 4 checking lines should be put to 6 -> 8 state
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000110xx0xxxx0111110001xxxx: begin
force Pla_out =
64'b0001000110000000000000000001001000010000001000011000100000001010;
$display($time, " line6->10");
end

// From state 6 -> 8 (no set bit, !R15)
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000110xx0xxxx0xxxx10001xxxx: begin
force Pla_out =
64'b0001000110000000000000000001001000010000001000011000100000001000;
$display($time, " line6->8");
end

// From state 6 -> 9 (set bit)
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000110xx0xxxx1xxxx10001xxxx: begin
force Pla_out =
64'b0001000110000000000000000001001000010000001000011100100000001001;
$display($time, " line6->9");
end

// ****************************************************************

// From state 7 -> 10 (no set bit, R15)
// This is done first to determine if Rd=>R[15] in verilog
// However in real Pla, 4 checking lines should be put to 7 -> 8 state
// Data Process (type 0, sub_type=shift mode1)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000111xx1xxxx01111x0010xxxx: begin
force Pla_out =
64'b0001001010000000000000000001001000010000001000101000100000001010;
$display($time, " line7->10");
end

// From state 7 -> 8 (no set bit, !R15)
```

```
// Data Process (type 0, sub_type=shift immediate)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000111xx1xxxx0xxxxx0010xxxx: begin
force Pla_out =
64'b0001001010000000000000000001001000010000001000101000100000001000;
$display($time, " line7->8");
end

// From state 7 -> 9 (set bit)
// Data Process (type 0, sub_type=shift immediate)
// AI++, IR=MEM[AR], Decoding, Memory Request, Shifter
29'b00000111xx1xxxx1xxxxx0010xxxx: begin
force Pla_out =
64'b0001001010000000000000000001001000010000001000101100100000001001;
$display($time, " line7->9");
end

// ****************************************************************

// From state 8 -> 3 (Fail conditional code)
// Data Process
// ALU -> Rd, AI -> PC, AI -> AR, Memory Request, IPR Shift
29'b000010000xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000001100001111000000001001010000000000100000000100000000011;
$display($time, " line8->3");
end

// From state 8 -> next instr_type
// Data Process
// ALU -> Rd, AI -> PC, AI -> AR, Memory Request, IPR Shift
29'b000010001xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000001100001111000000001001010000000000100000010100xxxxxxxx;
$display($time, " line8->next execution");
end

// ****************************************************************

// From 9 -> 11
// Set conditional bits
29'b00001001xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000000000000000000000000000000000000000000000000100100000001011;
$display($time, " line9->11");
end
// From state 11 -> 3 (Fail conditional code)
// Data Process with set bit
// Set flags, AI -> PC, AI -> AR, Memory Request, IPR Shift
29'b000010110xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000000100001111000000001001010000000000100000000100000000011;
$display($time, " line11->3");
end

// From state 11 -> next instr_type
```

```
// Data Process
// ALU -> Rd, AI -> PC, AI -> AR, Memory Request, IPR Shift
29'b000010111xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000000100001111000000001001010000000000100000010100 0xxxxxxxx;
$display($time, " line11->next execution");
end

// ************************************************************
// From state 10 -> 1 (Refill Pipeline)
// Data Process
// ALU -> PC, ALU -> AR, Memory Request, IPR Shift
29'b00001010xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1000000001011110000000000001001000000000000100000000100000000001;
$display($time, " line10->1");
end

// BRANCHING ***************************************************
// ************************************************************

// From state4 -> next instr_type
// ALU -> PC, ALU -> AR, Memory Request, IPR Shift, Decoding
29'b000001001xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100000000100001111000000001000000000000000100001010100 0xxxxxxxx;
$display($time, " line4->next execution");
end

// From state16 -> 17
// PC + IR(shift left by 2 bit) -> ALU, IR = MEM[AR], Memory Request,
// Decoding
29'b00010000xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000001110000000000111100000000000000100001000101000100000010001;
$display($time, " line16->17");
end

// From state17 -> 1
// No Link (Go back to state1 for pipeline refill)
// ALU -> AR, Memory Request, Decoding
29'b00010001xxx0xxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1000001100000000000111100000000000000100000000100000100000000001;
$display($time, " line17->1");
end

// From state17 -> 18
// With Link (Write back PC -> R[14])
// ALU -> AR, Memory Request, Decoding
29'b00010001xxx1xxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1000001100000000000111100000000000000100000000100000100000010010;
$display($time, " line17->18");
end
```

```
// From state18 -> 19
// IR = MEM[AR], AI++, R[14]=R[15], Memory Request
29'b00010010xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
//                          ||||
64'b0001000001011100000111100000000000000000010000010001000000010011;
$display($time, " line18->19");
end

// From state19 -> 15
// AI -> PC, AI -> AR, Memory Request, IPR Shift
29'b00010011xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100010110100001111000000001000000000000000010000100010000000001111;
$display($time, " line19->15");
end

// From state15 -> 20
// no Shift R[14]
29'b00001111xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000010110000000000000011100000000000001000000011000100000010100;
$display($time, " line15->20");
end

// From state20 -> 4
// IR = MEM[AR], AI++, R[14]=R[14]-4, Memory Request, Decoding
29'b00010100xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
//                          ||||
64'b0001000001011100000111000000000000000001001000001000100000000100;
$display($time, " line19->20");
end

// Single Data Transfer
// ****************************************************************

// From state 24-> 28
// Rn+Rm(shifted) -> ALU, IR = MEM[AR], AI++, Memory Request, Decoding
29'b00011000xx1x1xxxxxxxxxxxxxxxx: begin
force Pla_out =
//
64'b0001000010000000000000000000010000101000010000110001000000011100;
$display($time, " line24->28(shift, add)");
end

// From state 24-> 28
// Rn-Rm(shifted) -> ALU, IR = MEM[AR], AI++, Memory Request, Decoding
29'b00011000xx1x0xxxxxxxxxxxxxxxx: begin
force Pla_out =
//
64'b0001000010000000000000000000010000100100010000110001000000011100;
$display($time, " line24->28(shift, sub)");
end

// From state 24-> 28
```

```
// Rn+offset -> ALU, IR = MEM[AR], AI++, Memory Request, Decoding
29'b00011000xx0x1xxxxxxxxxxxxxxxx: begin
force Pla_out =
//
64'b0001010010000000000000000000010000001000010001010001000000011100;
$display($time, " line24->28(immediate, add)");
end

// From state 24-> 28
// Rn-offset -> ALU, IR = MEM[AR], AI++, Memory Request, Decoding
29'b00011000xx0x0xxxxxxxxxxxxxxxx: begin
force Pla_out =
//
64'b0001010010000000000000000000010000001000010001010001000000011100;
$display($time, " line24->28(immediate, sub)");
end

// ****************************************************************
// From state 28 -> 29
// AI -> PC, ALU -> AR / Rn output -> AR (no write back)
// First scan --> P=1, W=0; the rest are write back
// (only for software implementation)
// Pre-index
29'b00011100xxx1xx00xxxxxxxxxxxxx: begin
force Pla_out =
// |                   |               |
64'b1000000000100001111000000000001000000000000000000000100000011101;
$display($time, " line28->29(Pre-index, store)");
end

// From state 28 -> 30
// AI -> PC, ALU -> AR / Rn output -> AR (write back)
// Post-index
29'b00011100xxx0xxx0xxxxxxxxxxxxx: begin
force Pla_out =
64'b1110000000100001111000000000001000000000000000000000100000011110;
$display($time, " line28->29(Post-index, store)");
end
// Pre-index
29'b00011100xxx1xxx0xxxxxxxxxxxxx: begin
force Pla_out =
// |                   |               |
64'b1000000000100001111000000000001000000000000000000000100000011110;
$display($time, " line28->29(Pre-index, store)");
end

// From state 29 -> 31
// Rd -> DOR
//
29'b00011101xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
// |                   |               |
64'b0000000000000000000000000000010010000010000001000011000001111;
$display($time, " line29->31(store)");
end
```

```
// From state 30 -> 31
// ALU -> Rn, Rd -> DOR
//
29'b00011110xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
// |                   |               |
64'b0000000001000000000000000000001001000001000000100001100000011111;
$display($time, " line30->31(store)");
end

// From 31 -> 37
// Memory Write (byte)
// PC -> AR, Memory Request, IPR Shift
29'b00011111xxxxx1xxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0010000000000000000000000000000000000000000000000001000000100101;
$display($time, " line31->37(byte)");
end
// Memory Write (Word)
// PC -> AR, Memory Request, IPR Shift
29'b00011111xxxxx0xxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0010000000000000000000000000000000000000000000000001100000100101;
$display($time, " line31->37(word)");
end


// From state 37 -> next instr_type
// PC -> AR, Memory Request, IPR Shift
//
29'b001001011xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1010000000000000000000000000000000000000000001000000101000xxxxxxxx;
$display($time, " line37->next instr_type");
end

// From state 37 -> 3 (Fail conditional code)
// PC -> AR, Memory Request, IPR Shift
//
29'b001001010xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1010000000000000000000000000000000000000000001000000001000000000011;
$display($time, " line37->3");
end

// ************************************************************
// From state 28 -> 33
// AI -> PC, ALU -> AR / shifter output -> AR (no write back)
// First scan --> P=1, W=0; the rest are write back
// (only for software implementation)
// Pre-index
29'b00011100xxx1xx01xxxxxxxxxxxxx: begin
force Pla_out =
// |                   |               |
64'b1000000000100001111000000000010000000000000000000000100000100001;
$display($time, " line28->33(Pre-index, load)");
```

```
end

// From state 28 -> 34
// AI -> PC, ALU -> AR / shifter output -> AR (with write back)
// Post-index
29'b00011100xxx0xxx1xxxxxxxxxxxxx: begin
force Pla_out =
64'b11100000001000011110000000000010000000000000000000000100000100010;
$display($time, " line28->34(Post-index, load)");
end
// Pre-index
29'b00011100xxx1xxx1xxxxxxxxxxxxx: begin
force Pla_out =
//  |                   |               |
64'b10000000001000011110000000000010000000000000000000000100000100010;
$display($time, " line28->34(Pre-index, load)");
end

// From state 33 -> 35
// PC -> AR, Rd -> DIR
29'b00100001xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b00000000000000000000000000000010000000001000000000100000100011;
$display($time, " line33->35(load)");
end

// From state 34 -> 35
// PC -> AR, ALU -> Rn, Rd -> DIR
29'b00100010xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b00000000010000000000000000000010000000001000000000100000100011;
$display($time, " line34->35(load)");
end

// From state 35 -> 36
// Shift DIR (Word)
29'b00100011xxxxx0xxxxxxxxxxxxxxx: begin
force Pla_out =
64'b00000111100000000000000000000000000011010000010010001100001001000;
$display($time, " line35->36(Word, load)");
end
// Shift DIR (Byte)
29'b00100011xxxxx1xxxxxxxxxxxxxxx: begin
force Pla_out =
64'b00000110100000000000000000000000000011010000010010001100001001000;
$display($time, " line35->36(Byte, load)");
end

// From state 36 -> next instr_type
// ALU -> Rd, PC -> AR, IPR Shift, Memory Request
29'b001001001xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b10100000010000000000000000000010000000001000000101000xxxxxxxx;
$display($time, " line36->next instr_type");
end
```

```
// From state 36 -> 3 (Fail conditional code)
// ALU -> Rd, PC -> AR, IPR Shift, Memory Request
29'b001001000xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000000001000000000000000000001000000000010000000100000000011;
$display($time, " line36->3");
end

// Single Data Swap
// ****************************************************************
// From state 40 -> 41
// IR = MEM[AR], AI++, DOR = Rm, Memory Request, Fetch
29'b00101000xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001000000000000000000000000000010000101000011000100000101001;
$display($time, " line40->41");
end

// From state 41 -> 42
// Rn -> AR, AI -> PC, Memory Request
29'b00101001xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1110000000100001111000000000010000000000000000010001000000101010;
$display($time, " line41->42");
end

// From state 42 -> 43
// Databus -> DIR
29'b00101010xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000000000000000000000000000000000000000010000001000100000101011;
$display($time, " line42->43");
end

// From state 43 -> 44
// DIR -> Shift, Memory Write (Word)
29'b00101011xxxxx0xxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000011110000000000000000000000000011010000010010011000000101100;
$display($time, " line43->44");
end
// DIR -> Shift, Memory Write (Byte)
29'b00101011xxxxx1xxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000011010000000000000000000000000011010000010010010000000101100;
$display($time, " line43->44");
end

// From state 44 -> 3(Fail conditional code)
// DIR(shift) -> Rd, PC -> AR, IPR Shift, Memory Request
29'b001011000xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1010000001000000000000000000000010000000001000000001000000000011;
$display($time, " line44->3");
end
```

```
// From state 44 -> next instr_type
// DIR(shift) -> Rd, PC -> AR, IPR Shift, Memory Request
29'b001011001xxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1010000001000000000000000000001000000000010000001010000xxxxxxxx;
$display($time, " line44->next instr_type");
end

// Multiply and Multiply-Accumulate
// ****************************************************************
// From state 48 -> 49
// IR = MEM[AR], Rs -> BOOTH SHIFTER, AI++, Memory Request
29'b00110000xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001010110000000000000000000010100100110001000011000101000110001;
$display($time, " line48->49");
end

// From state 49 -> 50
// AI -> AR, AI -> PC (A=0)
29'b00110001xxxxxx0xxxxxxxxxxxxx: begin
force Pla_out =
64'b1100010111100001111000000000010100100110000000011000100000110010;
$display($time, " line49->50(A=0)");
end
// AI -> AR, AI -> PC (A=1)
29'b00110001xxxxxx1xxxxxxxxxxxxx: begin
force Pla_out =
64'b1100010111100001111000000000010100101101000000011000100000110010;
$display($time, " line49->50(A=1)");
end

// From state 50 -> 51
// Decision: borrow, op2%3
// Action: ALU = ALU(+/-)op1*(2^(2*n))
// borrow = 0, op2%3=0
29'b00110010xxxxxxxxxxxxxxxxx0000: begin
force Pla_out =
//      |||                       ||||
64'b0000010110000000000000000000001100010000000000011000110000110011;
$display($time, " line50->51(borrow=0 0)");
end

// borrow = 0, op2%3=1
29'b00110010xxxxxxxxxxxxxxxxx0010: begin
force Pla_out =
//      ||||                      ||||
64'b0000010001000000000000000000001100010100000000011000110000110011;
$display($time, " line50->51(borrow=0 1)");
end

// borrow = 0, op2%3=2
29'b00110010xxxxxxxxxxxxxxxxx0100: begin
force Pla_out =
//      |||                       ||||
64'b0000010011000000000000000000001100010010000000011000110000110011;
```

```
$display($time, " line50->51(borrow=0 2)");
end

// borrow = 0, op2%3=3
29'b00110010xxxxxxxxxxxxxxxxx0110: begin
force Pla_out =
//      |||                       ||||
64'b0000100010000000000000000000001100010010000000011000110000110011;
$display($time, " line50->51(borrow=0 3)");
end


// borrow = 1, op2%3=0
29'b00110010xxxxxxxxxxxxxxxxx1000: begin
force Pla_out =
//      |||                       ||||
64'b0000100010000000000000000000001100010100000000011000110000110011;
$display($time, " line50->51(borrow=1 0)");
end

// borrow = 1, op2%3=1
29'b00110010xxxxxxxxxxxxxxxxx1010: begin
force Pla_out =
//     ||||                       ||||
64'b0000100110000000000000000000001100010100000000011000110000110011;
$display($time, " line50->51(borrow=1 1)");
end

// borrow = 1, op2%3=2
29'b00110010xxxxxxxxxxxxxxxxx1100: begin
force Pla_out =
//      |||                       ||||
64'b0000100010000000000000000000001100010010000000011000110000110011;
$display($time, " line50->51(borrow=1 2)");
end

// borrow = 1, op2%3=3
29'b00110010xxxxxxxxxxxxxxxxx1110: begin
force Pla_out =
//      |||                       ||||
64'b0000010110000000000000000000001100010000000000011000110000110011;
$display($time, " line50->51(borrow=1 3)");
end


// From State 51 -> 50
// ALU -> Rd, Booth's shifter<-2, n++
29'b00110011xxxxxxxxxxxxxxxxxxxx0: begin
force Pla_out =
64'b0000000001000000000000000000010100010000000000000000110100110010;
$display($time, " line51->50");
end


// From state 51 -> 52 (no set bit)
// ALU -> Rd, Booth's shifter<-2, n++
```

```
29'b00110011xxxxxxx0xxxxxxxxxxxx1: begin
force Pla_out =
64'b0000000001000000000000000000101000100000000000000000100000110100;
$display($time, " line51->52(no set bit)");
end

// From state 51 -> 52(with set bit)
// ALU -> Rd, Booth's shifter<-2, n++
29'b00110011xxxxxxx1xxxxxxxxxxxx1: begin
force Pla_out =
64'b0000000001000000000000000000101000100000000000000100100000110100;
$display($time, " line51->52(with set bit)");
end

// From state 52 -> 3
// IPR shift
29'b001101000xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000000000000000000000000000101000100000001000000001000000000011;
$display($time, " line52->3");
end

// From state 52 -> next instr_type
// IPR shift
29'b001101001xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000000000000000000000000000101000100000001000000101000xxxxxxxx;
$display($time, " line52->next instr_type");
end


// Block Data Transfer ****************************************
// ************************************************************
// From state 56 -> 54
// IR=MEM[AR], RLD counter, AI++, Memory Request
29'b00111000xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001000000000000000000000000100000110000001010001000100000110110;
$display($time, " line56->54");
end

// From state 54 -> 57
// Shift Rn
29'b00110110xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0000010110000000000000000000100000111101000000011000110000111001;
$display($time, " line54->57");
end

// From state 57 -> 58
// AR = Rn
29'b00111001xxx01xxxxxxxxxxxxxxxx: begin
force Pla_out =
//                              ||||
64'b1000010110000000000000000000100000111101000000011000110000111010;
$display($time, " line56->58(case1)");
```

```
end

// AR = Rn+1
29'b00111001xxx11xxxxxxxxxxxxxxxx: begin
force Pla_out =
//                              ||||
64'b1000010110000000000000000000010000011111000000001100011000011101O;
$display($time, " line56->58(case2)");
end

// AR = Rn-n+1
29'b00111001xxx00xxxxxxxxxxxxxxxx: begin
force Pla_out =
//                              ||||
64'b1000010110000000000000000000010000011111000000001100011000011101O;
$display($time, " line56->58(case3)");
end

// AR = Rn-n
29'b00111001xxx10xxxxxxxxxxxxxxxx: begin
force Pla_out =
//                              ||||
64'b1000010110000000000000000000010000011001100000001100011000011101O;
$display($time, " line56->58(case4)");
end

// From state 58 -> 59
// AI -> PC, Rn = Rn - n (Write back, U = 0, Load)
29'b00111010xxxx0x11xxxxxxxxxxxx: begin
force Pla_out =
64'b0000010111100001111000000000100100110011000000011000100000111011;
$display($time, " line58->59(Load, write back, U=0)");
end

// AI -> PC, Rn = Rn + n (Write back, U = 1, Load)
29'b00111010xxxx1x11xxxxxxxxxxxx: begin
force Pla_out =
64'b0000010111100001111000000000100100110100000000011000100000111011;
$display($time, " line58->59(Load, write back, U=1)");
end

// AI -> PC, (no Write back, Load)
29'b00111010xxxxxx01xxxxxxxxxxxx: begin
force Pla_out =
64'b0000010110100001111000000000100100110000000000001000100000111011;
$display($time, " line58->59(Load, no write back)");
end

// From state 59 -> 63
// AI++, DIR = MEM[AR], Memory Request
29'b00111011xxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001010110000000000000000000100110001101010001001000100000111111;
$display($time, " line59->63");
end
```

```
// From state 63 -> 60
// DIR -> shift(no shift)
29'b00111111xxxxxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b0001010110000000000000000000100110001101000001001000100000111100;
$display($time, " line63->60");
end

// From state 60 -> 55
// DIR -> Rp
29'b00111100x1xxxxxx0xxxxxxxxxxx: begin
force Pla_out =
64'b0010010111000000000000000000100110001101000000001000100000110111;
$display($time, " line60->55");
end

// From state 60 -> 1 (Pipeline refill)
// AR = PC (Load, R15 in list)
// This conditional is checked first to save further checking in
// load or store in the case statement
29'b00111100x1xxxxxx1xxxxxxxxxxx: begin
force Pla_out =
64'b0010010111000000000000000000100110001101000100001000100000000001;
$display($time, " line60->1");
end

// From state 55 -> next instr_type
// AR = PC, IPR Shift
29'b0011011111xxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1010010110000000000000000000100110001101000101001010101000xxxxxxxx;
$display($time, " line55->next instr_type");
end

// From state 55 -> 3 (Fail Conditional Code)
// AR = PC
29'b0011011101xxxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1010010110000000000000000000100110001101000000001000100000000011;
$display($time, " line55->3");
end

// From state 60 -> 59
// DIR -> Rp, AR = PC
29'b00111100x0xxxxxxxxxxxxxxxxx: begin
force Pla_out =
64'b1100010111000000000000000000100110001101000011001000100000111011;
$display($time, " line60->59");
end

// ************************************************************

// From state 58 -> 61
// AI -> PC, Rn = Rn - n (Write back, U = 0, Store)
29'b00111010xxxx0x10xxxxxxxxxxx: begin
force Pla_out =
```

```
    64'b0000010111000001111000000000100100110011000000011000100000111101;
    $display($time, " line58->61(Store, write back, U=0)");
    end

    // AI -> PC, Rn = Rn + n (Write back, U = 1, Store)
    29'b00111010xxxx1x10xxxxxxxxxxxx: begin
    force Pla_out =
    64'b0000010111000001111000000000100100110100000000011000100000111101;
    $display($time, " line58->61(Store, write back, U=1)");
    end

    // AI -> PC, (no Write back, Store)
    29'b00111010xxxxxx00xxxxxxxxxxxx: begin
    force Pla_out =
    64'b0000010111000001111000000000100100110000000000001000100000111101;
    $display($time, " line58->61(Store, no write back)");
    end

    // From state 61 -> 62
    // AI++, DOR = Rp, Memory Write
    29'b00111101xxxxxxxxxxxxxxxxxxxx: begin
    force Pla_out =
    64'b0001000000000000000000000000100111001101100000011000110000111110;
    $display($time, " line61->62");
    end

    // From state 62 -> 55
    // RLD Shift
    29'b00111110x1xxxxxxxxxxxxxxxxxx: begin
    force Pla_out =
    64'b0000000000000000000000000000100111001101000010001001100000110111;
    $display($time, " line62->55");
    end

    // From state 62 -> 53
    // RLD Shift, AR = AI
    29'b00111110x0xxxxxxxxxxxxxxxxxx: begin
    force Pla_out =
    64'b0100000000000000000000000000100111001101000010001001100000110101;
    $display($time, " line62->53");
    end

    // From state 53 -> 61
    // AR = AI
    29'b00110101xxxxxxxxxxxxxxxxxxxx: begin
    force Pla_out =
    64'b1100000000000000000000000000100111001101000000000011100000111101;
    $display($time, " line62->53");
    end

    // ****************************************************************
    default: begin
      $display("\n**********************************Nothing Match in
Pla!!!!!!!!!!!!.....>>>>>>>>>>>>>>>>>\n");
    end
```

```
   endcase
  end

endmodule //of Pla
```

Read data register (Read_data_register.v)

```
// This is a module to simulate Write data register in ARM 710
module Read_data_register (data_out, data_in, write_en, clk, reset_bar);
  // data out to memory
  output [31:0] data_out;
  // data in from b bus
  input [31:0] data_in;
  // Read enable line
  input [0:0] write_en;
  // clock
  input [0:0] clk;
  input [0:0] reset_bar;

  reg [31:0] data_out;

  always @ (posedge clk)
  begin
    if (write_en) begin
      data_out = data_in;
      $display($time, " RDR: rdr is written to %b", data_out);
    end
  end
  always @ (reset_bar) begin
    if (!reset_bar) begin
      data_out = 32'b00000000000000000000000000000000;
   end
  end

endmodule //of Read_data_register
```

Register file (Register_bank.v)

```
// This is a module to simulate register bank of ARM 710
module Register_bank (a_out, b_out, Rs_out, pc_out, cpsr, alu_in, incr_in,
  cpsr_flag, write_en1, write_en2, cpsr_w_en, reg_num_in1, reg_num_in2,
  reg_num_out1, reg_num_out2, reg_num_out3, clk, reset_bar);
  // output to a bus
  output [31:0] a_out;
  // output to b bus
  output [31:0] b_out;
  // output for rotate amount
  output [7:0] Rs_out;
  // output to pc bus
  output [31:0] pc_out;
  // output to control;
  output [31:0] cpsr;
```

```
// input from alu bus
input [31:0] alu_in;
// input from incrementer bus
input [31:0] incr_in;
// input from control
input [3:0] cpsr_flag;
// read and write enables
input [0:0] write_en1;
input [0:0] write_en2;
input [0:0] cpsr_w_en;
// register number
input [3:0] reg_num_in1;
input [3:0] reg_num_in2;
input [3:0] reg_num_out1;
input [3:0] reg_num_out2;
input [3:0] reg_num_out3;
// clock
input [0:0] clk;
input [0:0] reset_bar;

reg [31:0] a_out;
reg [31:0] b_out;
reg [31:0] Rs_out;
reg [31:0] pc_out;
reg [31:0] cpsr;
reg [27:0] temp;
// 37 register spaces
reg [31:0] R[36:0];

// write at the positive edge of write signal
always @ (posedge clk)
  begin
    if (write_en1) begin
      R[reg_num_in1] = alu_in;
           $display($time, " R[%d] is written to %b", reg_num_in1,
           R[reg_num_in1]);
    end
  end
always @ (posedge clk)
  begin
    if (write_en2) begin
      R[reg_num_in2] = incr_in;
           $display($time, " R[%d] is written to %b", reg_num_in2,
           R[reg_num_in2]);
    end
  end
always @ (posedge clk)
  begin
    if (cpsr_w_en) begin
      temp = R[16];
      R[16] = {cpsr_flag, temp};
           $display($time, " R[16] is written to %b", R[16]);
    end
  end

// output of register bank
```

```
 always @ (reg_num_out1 or reg_num_out2 or R[reg_num_out1] or
 R[reg_num_out2] or R[15] or R[16])
  begin
    a_out = R[reg_num_out1];
    b_out = R[reg_num_out2];
    pc_out = R[15];
    cpsr = R[16];
  end

 // output of Rs
 always @ (reg_num_out3 or R[reg_num_out3])
  begin
    Rs_out = R[reg_num_out3];
  end

 // reset
 always @ (reset_bar) begin
  if (!reset_bar) begin
    // ***** The following 2 items are omitted *****
    // set R14 -> R14_svc
    // set CPSR -> SPSR_svc
    // ***********************************************
    // set cpsr M[4:0] = 10011
    R[16] = 32'b00000000000000000000000011010011;
    // set PC = 0
    R[15] = 32'b00000000000000000000000000000000;
  end
 end

endmodule //of Register_bank
```

Register list decoder (Register_list_decoder.v)

```
// This is a module to simulate register list decoder in ARM 710
// This module input the register list the first time it is enabled
// The register number will be returned every time it is enabled
// until the end of the list
module Register_list_decoder (reg_num, count_list, end_list, reg_list,
 shift_en, clk);
 // current register number loading or storing
 output [3:0] reg_num;
 // number of register in list
 output [3:0] count_list;
 // notice Controller the end of register list
 output [0:0] end_list;
 // register list from instruction
 input [15:0] reg_list;
 // register list shift enable
 input [0:0] shift_en;
 input [0:0] clk;

 // A register to indicate this is the first call of the decoder
 reg [0:0] first_visit;
 // Count number of 1's in the reg_list
```

```
  reg [4:0] count_list;
  // A register to indicate the whole list has been scanned through
  reg [0:0] end_list;
  // An internal register to store the register list
  reg [15:0] list;
  // Internal register number
  reg [4:0] list_num;
  // Output register number
  reg [3:0] reg_num;

  reg [4:0] temp;

  initial begin
    first_visit = 1;
    reg_num = 0;
    end_list = 1;
  end

  always @ (posedge clk) begin
    // check for shift enable
    if (shift_en) begin
      // if first visit, put register list to list, put end list to null
      if (first_visit) begin
        first_visit = 0;
        list = reg_list;
            list_num = 0;
            end_list = 0;
            // Call task to count number of register list
            count (reg_list, count_list);
            count_list = count_list*4;
            $display("RLD: count_list=%d", count_list);
      end
//      $display($time, " first_visit=%d, list=%b, end_list=%d", first_visit, list, end_list);

       // if we finished checking all the 1's set back to starting set up
      if (!list) begin
        end_list = 1;
            first_visit = 1;
      end

      // if list not equal to zero, find the next 1
      if (list) begin
        for (list_num = list_num; list[list_num]<1 ;
            list_num=list_num + 1) begin
//      $display($time, " list[%d]=%d, loop=%d", list_num, list[list_num], list_num);
            end
      end
      // if list[list_num] = 1 is found, set it to 0
      if (list[list_num]) begin
        list[list_num] = 0;
            reg_num = list_num;
      end
      else begin
            $display($time, " Null list Inputed");
      end
//      $display("End of a Digit");
```

```
    end
  end

task count;
  input [15:0] reg_list;
  output [4:0] count_list;
  integer i;
  begin
    count_list = 4'b0000;
    for (i = 0; i < 16; i = i + 1) begin
      if (reg_list[i]) begin
        count_list = count_list + 1;
      end
    end
  end

endtask //of count

endmodule //of Register_list_decoder.v
```

State decoder (State_decoder.v)

```
// This is a simulation model of a state decoder
// We have assigned instruction type number to different kind of instruction
// In order to allow immediate loop back in the last execution phase in the
// pipeline to continue the next execution phase, a state decoder is introduced
// (Note that state assignment and instruction type number assignment can be
//  done in such a fashion that a state decoder can be omitted. However, due
//  to the complexity in PLA, we introduce the state decoder.)

module State_decoder (next_state, next_state_in, instr_type, sel);
  output [7:0] next_state;
  // next state from Controller
  input [7:0] next_state_in;
  // instruction type from instruction decoder
  input [3:0] instr_type;
  input [0:0] sel;

  reg [7:0] next_state;

  // 0000 -> Data Processing, shift mode 1
  // 0001 -> Data Processing, shift mode 2
  // 0010 -> Data Processing, immmediate
  // 0011 -> Branching
  // 0100 -> Single Data Transfer
  // 0101 -> Single Data Swap
  // 0110 -> Multiply and Multiply-Accumulate
  // 0111 -> Block Data Transfer

  always @ (next_state_in or instr_type or sel) begin
    if (sel) begin
      case (instr_type)
       4'b0000: next_state = 8'b00000101;
       4'b0001: next_state = 8'b00000110;
```

```
      4'b0010: next_state = 8'b00000111;
      4'b0011: next_state = 8'b00010000;
      4'b0100: next_state = 8'b00011000;
      4'b0101: next_state = 8'b00101000;
      4'b0110: next_state = 8'b00110000;
      4'b0111: next_state = 8'b00111000;

      default: $display("No instr_type  matched in State_decoder");
     endcase
   end
   else begin
     next_state = next_state_in;
   end
//    $display("State Decoder: next_state_in=%d, instr_type=%b, sel=%b",
//    next_state_in, instr_type, sel);
   $display($time, " State Decoder: next_state=%d", next_state);

  end

endmodule //of State_decoder
```

State register (State_register.v)

```
// This is a module to simulate the state register in ARM 710
module State_register (present, next, clk, reset_bar);
  // present state
  output [7:0] present;
  // next state
  output [7:0] next;
  input [0:0] clk;
  input [0:0] reset_bar;

  reg [7:0] present;

  always @ (negedge clk) begin
    present = next;
    $display($time, " SR: Present State=%d", present);
  end

  always @ (reset_bar) begin
    if (!reset_bar) begin
      present = 8'b00000000;
    end
  end

endmodule //of State_register
```

Write data register (Write_data_register.v)

```
// This is a module to simulate Write data register in ARM 710
module Write_data_register (data_out, data_in, address, nBW, write_en, clk, reset_bar);
  // data out to memory
```

```
 output [31:0] data_out;
 // data in from b bus
 input [31:0] data_in;
 // 2 lsb of the Address register
 input [1:0] address;
 input [0:0] nBW;
 // write enable line
 input [0:0] write_en;
 // clock
 input [0:0] clk;
 input [0:0] reset_bar;

 reg [31:0] data;
 reg [31:0] data_out;

 always @ (posedge clk)
 begin
  if (write_en) begin
    data = data_in;
    $display($time, " WDR: wdr is written to %b", data);
  end
 end

 always @ (clk) begin
  data_out = data_out;
 end

 always @ (data or nBW) begin
  if (nBW) begin
    data_out = data;
  end
  else begin
    if (address == 0) begin
      data_out = {data[7:0], data[7:0], data[7:0], data[7:0]};
    end
    else if (address == 1) begin
      data_out = {data[15:8], data[15:8], data[15:8], data[15:8]};
    end
    else if (address == 2) begin
      data_out = {data[23:16], data[23:16], data[23:16], data[23:16]};
    end
    else begin
      data_out = {data[31:24], data[31:24], data[31:24], data[31:24]};
    end
  end
 end


 always @ (reset_bar) begin
  if (!reset_bar) begin
    data_out = 32'b00000000000000000000000000000000;
 end
 end

endmodule //of Write_data_register
```

APPENDIX F: Verification Results for ARM 7

Note to thesis committee

Due to the amount of data in this appendix, we also decided not to distribute it with this copy of the thesis. If necessary, I can supply a copy on request. Otherwise, it will be included in my final draft to be left with the department.

REVE Step 1

In our notation, periods indicate a submodule. Therefore, for the Program Counter, the notation "rb.R[15]" means that the Program Counter is register 15 in the register bank rb. The top-level module is Block.v. The components we identified were:

Program Counter:	rb.R[15]
Instruction Register:	ipr
Register File:	rb
Data Bus:	D
Address Bus:	A_out
System Clock:	CLK
ALU:	alu
Datapath Width:	32

REVE Step 2

The entirety of step 2 is shown in Chapter 4. The data paths traced in our analysis are shown in Figure 4.4.

REVE Step 3

Control signal extraction is shown for the branch class of instructions in Figure 4.5. The following figures present the control signal extraction data for the remaining three instruction classes: data-processing, load, and store.

Data-processing operations

Reverse engineered Instruction	Conditions
1a1. Reg <- Reg1, [Reg3 (shift) Data Bus] Addr Bus <- ALU	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=00 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=00
1a2. Reg <- Reg1, [Reg3 (shift) Data Bus] Addr Bus <- Reg_PC	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=01 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=01
1a3. Reg <- Reg1, [Reg3 (shift) Data Bus] Addr Bus <- Addr Bus	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=10 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=10
1a4. Reg <- Reg1, [Reg3 (shift) Data Bus] Addr Bus <- Reg1	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0
1a5. Reg <- Reg1, [Reg3 (shift) Data Bus] Addr Bus <- Control	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1
1b. Reg <- Reg1, [Reg3 (shift) IR]	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=0, B_in_en2=1, B_in_en3=0 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=0, B_in_en2=1, B_in_en3=0
1c. Reg <- Reg1, [Reg3 (shift) Reg2]	i: mux2_32_sel=0 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=0, B_in_en2=0, B_in_en3=1 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=0 && l_en=1 B_in_en1=0, B_in_en2=0, B_in_en3=1
2a1. Reg <- Control, [Reg3 (shift) Data Bus] Addr Bus <- ALU	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1

	i-3: addr_r_w_en=1, addr_r_sel=00 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=00
2a2. Reg <- Control, [Reg3 (shift) Data Bus] Addr Bus <- Reg_PC	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=01 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=01
2a3. Reg <- Control, [Reg3 (shift) Data Bus] Addr Bus <- Addr Bus	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=10 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=10
2a4. Reg <- Control, [Reg3 (shift) Data Bus] Addr Bus <- Reg1	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0
2a5. Reg <- Control, [Reg3 (shift) Data Bus] Addr Bus <- Control	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=1, B_in_en2=0, B_in_en3=0 i-2: rdr_w_en=1 i-3: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1
2b. Reg <- Control, [Reg3 (shift) IR]	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=0, B_in_en2=1, B_in_en3=0 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=0, B_in_en2=1, B_in_en3=0
2c. Reg <- Control, [Reg3 (shift) Reg2]	i: mux2_32_sel=1 && l_en=1 && rb_w_en1=1 i-1: B_in_en1=0, B_in_en2=0, B_in_en3=1 OR ---- i: l_en=0 && rb_w_en1=1 i-1: mux2_32_sel=1 && l_en=1 B_in_en1=0, B_in_en2=0, B_in_en3=1

Load Operations

(Very few operations because most Reg-writes go through ALU)

Reverse engineered Instruction	**Conditions**
1a. Reg <- Addr Bus Addr Bus <- ALU	Same as Branch, since rb_w_en2 is never on except when reg_num_in2=1111
1b. Reg <- Addr Bus Addr Bus <- Reg_PC	Same as Branch, since rb_w_en2 is never on except when reg_num_in2=1111
1c. Reg <- Addr Bus Addr Bus <- Addr Bus	Same as Branch, since rb_w_en2 is never on except when reg_num_in2=1111
1d. Reg <- Addr Bus Addr Bus <- Reg1	Same as Branch, since rb_w_en2 is never on except when reg_num_in2=1111
1e. Reg <- Addr Bus Addr Bus <- Control	Same as Branch, since rb_w_en2 is never on except when reg_num_in2=1111
2. Reg <- ALU	Covered in ALU operations

Store Operations

Reverse engineered Instruction	Conditions
1a. Data Bus <- Data Bus Addr Bus <- ALU	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=00, wdr_w_en=1 b_bus1=1, b_bus2=0, b_bus3=0 P1 or i-2: l_en1=1 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=00 i-2: rdr_w_en=1 P2 or i-3: l_en1=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=00, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 P2 or i-3: l_en1=1 i-3: rdr_w_en=1
1b. Data Bus <- Data Bus Addr Bus <- Reg_PC	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=01, wdr_w_en=1 b_bus1=1, b_bus2=0, b_bus3=0 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=01 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=01, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-3: rdr_w_en=1
1c. Data Bus <- Data Bus Addr Bus <- Addr Bus	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=10, wdr_w_en=1 b_bus1=1, b_bus2=0, b_bus3=0 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=10 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=10, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-3: rdr_w_en=1
1d. Data Bus <- Data Bus Addr Bus <- Reg1	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=1 mux2_32_sel=0 b_bus1=1, b_bus2=0, b_bus3=0 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=0 mux2_32_sel=0 i-2: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-3: rdr_w_en=1
1e. Data Bus <- Data Bus Addr Bus <- Control	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=1 mux2_32_sel=1 b_bus1=1, b_bus2=0, b_bus3=0 i-2: rdr_w_en=1 ----

	i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1 i-2: rdr_w_en=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=0 mux2_32_sel=1 i-2: wdr_w_en=1, b_bus1=1, b_bus2=0, b_bus3=0 i-3: rdr_w_en=1
2a. Data Bus <- IR Addr Bus <- ALU	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=00, wdr_w_en=1 b_bus1=0, b_bus2=1, b_bus3=0 P1 or i-2: l_en1=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=00 P2 or i-3: l_en1=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=00, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0 P2 or i-3: l_en1=1
2b. Data Bus <- IR Addr Bus <- Reg_PC	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=01, wdr_w_en=1 b_bus1=0, b_bus2=1, b_bus3=0 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=01 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=01, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0
2c. Data Bus <- IR Addr Bus <- Addr Bus	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=10, wdr_w_en=1 b_bus1=0, b_bus2=1, b_bus3=0 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=10 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=10, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0
2d. Data Bus <- IR Addr Bus <- Reg1	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=1 mux2_32_sel=0 b_bus1=0, b_bus2=1, b_bus3=0 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=0 mux2_32_sel=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0
2e. Data Bus <- IR Addr Bus <- Control	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=1 mux2_32_sel=1 b_bus1=0, b_bus2=1, b_bus3=0 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0 i-2: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=0 mux2_32_sel=1

	i-2: wdr_w_en=1, b_bus1=0, b_bus2=1, b_bus3=0
3a. Data Bus <- Reg2 Addr Bus <- ALU	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=00, wdr_w_en=1 b_bus1=0, b_bus2=0, b_bus3=1 P1 or i-2: l_en1=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1 i-2: addr_r_w_en=1, addr_r_sel=00 P2 or i-3: l_en1=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=00, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1 P2 or i-3: l_en1=1
3b. Data Bus <- Reg2 Addr Bus <- Reg_PC	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=01, wdr_w_en=1 b_bus1=0, b_bus2=0, b_bus3=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1 i-2: addr_r_w_en=1, addr_r_sel=01 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=01, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1
3c. Data Bus <- Reg2 Addr Bus <- Addr Bus	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=10, wdr_w_en=1 b_bus1=0, b_bus2=0, b_bus3=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1 i-2: addr_r_w_en=1, addr_r_sel=10 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=10, wdr_w_en=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1
3d. Data Bus <- Reg2 Addr Bus <- Reg1	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=1 mux2_32_sel=0 b_bus1=0, b_bus2=0, b_bus3=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1 i-2: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=0 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=0 mux2_32_sel=0 i-2: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1
3e. Data Bus <- Reg2 Addr Bus <- Control	i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=1 mux2_32_sel=1 b_bus1=0, b_bus2=0, b_bus3=1 ---- i: nRW=1, nMREQ=0 i-1: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1 i-2: addr_r_w_en=1, addr_r_sel=11, mux2_32_sel=1 ---- i: nRW=1, nMREQ=0 i-1: addr_r_w_en=1, addr_r_sel=11, wdr_w_en=0 mux2_32_sel=1 i-2: wdr_w_en=1, b_bus1=0, b_bus2=0, b_bus3=1

REVE Step 4

In step 4, we check for the existence of control sets defined in step 3. This analysis is shown for the branch instruction class in Figure 4.6. The following figures show the conditions necessary for the existence of each control set for the remaining three instruction classes: data-processing, load, and store. The state numbering system is the same as the one used in Figure 4.6.

Data-processing instructions

DP1a1: 36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+Rm(shifted))
36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-Rm(shifted))
36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+offset)
36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-offset)
36→next_instr_type, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn+Rm(shifted))
36→next_instr_type, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn-Rm(shifted))
36→next_instr_type, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn+offset)
36→next_instr_type, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn-offset)
36→next_instr_type, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn+Rm(shifted))
36→next_instr_type, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn-Rm(shifted))
36→next_instr_type, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn+offset)
36→next_instr_type, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn-offset)
36→next_instr_type, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn+Rm(shifted))
36→next_instr_type, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn-Rm(shifted))
36→next_instr_type, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn+offset)
36→next_instr_type, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn-offset)

Data-processing instructions con't.

DP1a1: 36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+Rm(shifted))
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-Rm(shifted))
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+offset)
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-offset)
36→3, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn+Rm(shifted))
36→3, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn-Rm(shifted))
36→3, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn+offset)
36→3, 35→36 (Word) , 34→35, 28→34, 24→28 (Rn-offset)
36→3, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn+Rm(shifted))
36→3, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn-Rm(shifted))
36→3, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn+offset)
36→3, 35→36 (Byte) , 33→35, 28→33, 24→28 (Rn-offset)
36→3, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn+Rm(shifted))
36→3, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn-Rm(shifted))
36→3, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn+offset)
36→3, 35→36 (Byte) , 34→35, 28→34, 24→28 (Rn-offset)

DP1a2: No exercisable data paths

DP1a3: No exercisable data paths

DP1a4: 36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+Rm(shifted))
36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-Rm(shifted))
36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+offset)
36→next_instr_type, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-offset)
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+Rm(shifted))
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-Rm(shifted))
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn+offset)
36→3, 35→36 (Word) , 33→35, 28→33, 24→28 (Rn-offset)
44→3, 43→44 (Word), 42→43, 41→42
44→3, 43→44 (Byte), 42→43, 41→42
44→next_instr_type, 43→44 (Word), 42→43, 41→42
44→next_instr_type, 43→44 (Byte), 42→43, 41→42

DP1a5: No exercisable data paths

Data-processing instructions con't.

DP1b:	18→19
	8→3 (Fail conditional code), 7→8
	8→next_instr_type, 7→8
	10→1, 7→10
	30→31, 28→30 (Post index), 24→28 (Rn+offset)
	30→31, 28→30 (Post index), 24→28 (Rn–offset)
	30→31, 28→30 (Pre index), 24→28 (Rn+offset)
	30→31, 28→30 (Pre index), 24→28 (Rn–offset)
	34→35, 28→34 (Post index), 24→28 (Rn+offset)
	34→35, 28→34 (Post index), 24→28 (Rn–offset)
	34→35, 28→34 (Pre index), 24→28 (Rn+offset)
	34→35, 28→34 (Pre index), 24→28 (Rn–offset)
DP1c:	20→4
	49→50 (A=0)
	49→50 (A=1)
	8→3 (Fail conditional code), 5→8
	8→3 (Fail conditional code), 6→8
	8→next_instr_type (Fail conditional code), 5→8
	8→next_instr_type (Fail conditional code), 6→8
	10→1 (Refill pipeline), 5→10
	10→1 (Refill pipeline), 6→10
	30→31, 28→30 (Post index), 24→28 (Rn+offset)
	30→31, 28→30 (Post index), 24→28 (Rn–offset)
	30→31, 28→30 (Pre index), 24→28 (Rn+offset)
	30→31, 28→30 (Pre index), 24→28 (Rn–offset)
	34→35, 28→34 (Post index), 24→28 (Rn+offset)
	34→35, 28→34 (Post index), 24→28 (Rn–offset)
	34→35, 28→34 (Pre index), 24→28 (Rn+offset)
	34→35, 28→34 (Pre index), 24→28 (Rn–offset)
	51→50, 50→51 (each Booth case)
	51→52 (no set bit), 50→51 (each Booth case)
	51→52 (with set bit), 50→51 (each Booth case)
DP2a1:	No exercisable data paths
DP2a2:	No exercisable data paths
DP2a3:	60→55
	60→1
	60→59
DP2a4:	No exercisable data paths
DP2a5:	No exercisable data paths
DP2b:	No exercisable data paths
DP2c:	58→59 (Rn=Rn–n)
	58→59 (Rn=Rn+n)
	58→61 (Rn=Rn–n)
	58→61 (Rn=Rn+n)

Store instructions

Store1a: No exercisable data paths
Store1b: No exercisable data paths
Store1c: No exercisable data paths
Store1d: No exercisable data paths
Store1e: No exercisable data paths
Store2a: No exercisable data paths
Store2b: No exercisable data paths
Store2c: No exercisable data paths
Store2d: No exercisable data paths
Store2e: No exercisable data paths
Store3a: 31→37 (Byte), 29→31, 28→29, 24→28 (Rn+Rm(shifted))
31→37 (Byte), 29→31, 28→29, 24→28 (Rn–Rm(shifted))
31→37 (Byte), 29→31, 28→29, 24→28 (Rn+offset(shifted))
31→37 (Byte), 29→31, 28→29, 24→28 (Rn–offset(shifted))
31→37 (Byte), 30→31, 28→30 (Pre-index), 24→28 (Rn+Rm(shifted))
31→37 (Byte), 30→31, 28→30 (Pre-index), 24→28 (Rn–Rm(shifted))
31→37 (Byte), 30→31, 28→30 (Pre-index), 24→28 (Rn+offset(shifted))
31→37 (Byte), 30→31, 28→30 (Pre-index), 24→28 (Rn–offset(shifted))
31→37 (Word), 29→31, 28→29, 24→28 (Rn+Rm(shifted))
31→37 (Word), 29→31, 28→29, 24→28 (Rn–Rm(shifted))
31→37 (Word), 29→31, 28→29, 24→28 (Rn+offset(shifted))
31→37 (Word), 29→31, 28→29, 24→28 (Rn–offset(shifted))
31→37 (Word), 30→31, 28→30 (Pre-index), 24→28 (Rn+Rm(shifted))
31→37 (Word), 30→31, 28→30 (Pre-index), 24→28 (Rn–Rm(shifted))
31→37 (Word), 30→31, 28→30 (Pre-index), 24→28 (Rn+offset(shifted))
31→37 (Word), 30→31, 28→30 (Pre-index), 24→28 (Rn–offset(shifted))
62→55, 61→62, 57→58 (AR=Rn), 54→57
62→55, 61→62, 57→58 (AR=Rn+1), 54→57
62→55, 61→62, 57→58 (AR=Rn–n+1), 54→57
62→55, 61→62, 57→58 (AR=Rn–n), 54→57
62→53, 61→62, 57→58 (AR=Rn), 54→57
62→53, 61→62, 57→58 (AR=Rn+1), 54→57
62→53, 61→62, 57→58 (AR=Rn–n+1), 54→57
62→53, 61→62, 57→58 (AR=Rn–n), 54→57
Store3b: No exercisable data paths
Store3c: 62→55, 61→62, 53→61
62→53, 61→62, 53→61
Store3d: No exercisable data paths
Store3e: 31→37 (Byte), 30→31, 28→30 (Post-index)
31→37 (Word), 30→31, 28→30 (Post-index)
43→44 (Word), 41→42, 40→41
43→44 (Byte), 41→42, 40→41

REVE Steps 5 and 6

In step 5, we determine implications made by each control set, which are used in step 6 to generate an unmerged RISA. This RISA for all branch instructions is shown in is shown in Figure 4.7. The unmerged RISA for all remaining ARM 7 is shown below.

Block data transfer instructions

Instruction Type	Assigned Opcode Bits (IR[31:0])	Functionality
DP2a3	xxxx 100x xxxx xxxx 0xxx xxxx xxxx xxxx	When end_list=1 R(Rp)= M[Prev(Address Bus+4)] IR[31:28] = Condition codes (tracing back to previous instruction)
DP2a3	xxxx 100x xxxx xxxx 1xxx xxxx xxxx xxxx	When end_list=1 R(Rp)=M[Prev(Address Bus+4)] IR[31:28] = Condition codes (tracing back to previous instruction)
DP2a3	xxxx 100x xxxx xxxx xxxx xxxx xxxx xxxx	When end_list=0 R(Rp)= M[Prev(Address Bus+4)] IR[31:28] = Condition codes (tracing back to previous instruction)
DP2c	xxxx 100x 0x11 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – one's count(Instr[15:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP2c	xxxx 100x 1x11 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) + one's count(Instr[15:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP2c	xxxx 100x 1x11 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) + one's count(Instr[15:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP2c	xxxx 100x 1x10 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – one's count(Instr[15:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP2c	xxxx 100x 1x10 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) + one's count(Instr[15:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1000 1xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16]]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1001 1xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16]]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1000 0xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16] – (one's count of Instr[15:0]) + 8]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1001 0xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16] – (one's count of Instr[15:0])]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1000 1xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16]]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1001 1xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16]]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1000 0xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16] – (one's count of Instr[15:0]) + 8]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3a	xxxx 1001 0xxx xxxx xxxx xxxx xxxx xxxx	With end_list=1 M[R[Instr[19:16] – (one's count of Instr[15:0])]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)

Store3c	xxxx 100x xxxx xxxx xxxx xxxx xxxx xxxx	M[Address Bus]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)
Store3c	xxxx 100x xxxx xxxx xxxx xxxx xxxx xxxx	M[Address Bus]=Rp IR[31:28] = Condition codes (tracing back to previous instruction)

Data processing instructions

Instruction Type	Assigned Opcode Bits (IR[31:0])	Functionality
DP1b	xxxx 001x xxx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output Shifter_output = R-shift Instr[7:0] by 2*Instr[11:7] BUG!! (Should be R-rotate, not R-shift) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 001x xxx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output Shifter_output = R-shift Instr[7:0] by 2*Instr[11:7] BUG!! (Should be R-rotate, not R-shift) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 001x xxx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output Shifter_output = R-shift Instr[7:0] by 2*Instr[11:7] BUG!! (Should be R-rotate, not R-shift) IR[31:28] = Condition codes (tracing back to previous instruction)

DP1b	xxxx 001x xxx0 xxxx
	xxxx xxxx xxx0 xxxx
		R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1b	xxxx 001x xxx0 xxxx
	xxxx xxxx 0xx0 xxxx
		R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1b	xxxx 001x xxx0 xxxx
	xxxx xxxx xxx0 xxxx
		R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1b	xxxx 001x xxx0 xxxx xxxx xxxx 0xx0 xxxx	R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output shifter_output = If Instr[6:5]=00 then

		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1b	xxxx 001x xxx0 xxxx
	xxxx xxxx xxx0 xxxx
		R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1b	xxxx 001x xxx0 xxxx
	xxxx xxxx 0xx0 xxxx
		R(Instr[15:12])=R(Instr[19:16]) (opcode) shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

Multiply instructions

Instruction Type	**Assigned Opcode Bits (IR[31:0])**	**Functionality**
DP1c	xxxx 0000 000x xxxx xxxx xxxx 1001 xxxx	R(Instr[19:16]) = 32'b0 IR[31:28] = Condition codes (tracing back to previous instruction)
DP1c	xxxx 0000 001x xxxx xxxx xxxx 1001 xxxx	R(Instr[19:16]) = R(Instr[15:12]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1c	xxxx 0000 00xx xxxx xxxx xxxx 1001 xxxx	** n comes from loop in Booth Shifter R(Instr[19:16]) = R(Instr[19:16]) Booth_Case=000 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^n} Booth_Case=001 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^(n+1)} Booth_Case=010 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^n} Booth_Case=011 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^n} Booth_Case=100 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^(n+1)} Booth_Case=101 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^n} Booth_Case=110 , B_stop=0

		R(Instr[19:16]) = R(Instr[19:16]) Booth_Case=111 , B_stop=0 IR[31:28] = Condition codes (tracing back to previous instruction)
DP1c	xxxx 0000 00xx 0xxx xxxx xxxx 1001 xxxx	** n comes from loop in Booth Shifter R(Instr[19:16]) = R(Instr[19:16]) Booth_Case=000 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^n} Booth_Case=001 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^(n+1)} Booth_Case=010 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^n} Booth_Case=011 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^n} Booth_Case=100 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^(n+1)} Booth_Case=101 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^n} Booth_Case=110 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) Booth_Case=111 , B_stop=0 IR[31:28] = Condition codes (tracing back to previous instruction)
DP1c	xxxx 0000 00xx 1xxx xxxx xxxx 1001 xxxx	** n comes from loop in Booth Shifter R(Instr[19:16]) = R(Instr[19:16]) Booth_Case=000 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^n} Booth_Case=001 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^(n+1)} Booth_Case=010 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^n} Booth_Case=011 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^n} Booth_Case=100 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) + {R(Instr[3:0]) * 2^(n+1)} Booth_Case=101 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) - {R(Instr[3:0]) * 2^n} Booth_Case=110 , B_stop=0 R(Instr[19:16]) = R(Instr[19:16]) Booth_Case=111 , B_stop=0 IR[31:28] = Condition codes (tracing back to previous instruction)

Single data swap instructions

Instruction Type	**Assigned Opcode Bits (IR[31:0])**	**Functionality**
DP1a4	xxxx 0001 00xx xxxx xxxx 0000 1001 xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] ˝ BUG!! If Word_align=3, not all of shifter_out is defined. ˝ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0001 01xx xxxx xxxx 0000 1001 xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16])] ˝ BUG!! If Word_align=3, not all of shifter_out is defined. ˝ IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0001 00xx xxxx xxxx 0000 1001 xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] ˝ BUG!! If Word_align=3, not all of shifter_out is defined. ˝ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0001 01xx xxxx xxxx 0000 1001 xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16])] ˝ BUG!! If Word_align=3, not all of shifter_out is defined. ˝ IR[31:28] = Condition codes (tracing back to previous instruction)
Store3d	xxxx 0001 00xx xxxx xxxx 0000 1001 xxxx	M[R(Instr[19:16])]=R(Instr[3:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
Store3d	xxxx 0001 01xx xxxx xxxx 0000 1001 xxxx	M[R(Instr[19:16])]=R(Instr[3:0]) ˝ IR[31:28] = Condition codes (tracing back to previous instruction)
Store3d	xxxx 0001 01xx xxxx xxxx 0000 1001 xxxx	M[R(Instr[19:16])]=R(Instr[3:0]) ˝ IR[31:28] = Condition codes (tracing back to previous instruction)

˝

Single data transfer instructions

Instruction Type	**Assigned Opcode Bits (IR[31:0])**	**Functionality**

DP1a4	xxxx 0111 1001 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) + shifter_output]
		˝
		BUG!! If Word_align=3, not all of shifter_out is defined.
		˝
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 0001 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) – shifter_output]
		˝
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 1001 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] ˝

		BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 0001 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] ″ BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 10x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) +
		shifter_output]
		″
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 00x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) –
		shifter_output]
		″
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 10x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] ″ BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 00x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] ″ BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 1101 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) +
		shifter_output]
		″
		shifter_output = If Instr[6:5]=00 then
		″
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		″
		R-rotate R(Instr[3:0]) by Instr[11:7]

		″ IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 0101 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – shifter_output]
		″
		shifter_output = If Instr[6:5]=00 then
		″
		L-shift R(Instr[3:0]) by Instr[11:7]
		″
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 1101 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] ″ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 0101 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 11x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) +
		shifter_output]
		″
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 01x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – shifter_output]
		″
		shifter_output = If Instr[6:5]=00 then
		″
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 11x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] ″ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 01x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] ″ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0111 1001 xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) +

	xxxx xxxx xxxx xxxx	shifter_output]
		″
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 0001 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) –
		shifter_output]
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 1001 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 0001 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 10x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) +
		shifter_output]
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 00x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) –
		shifter_output]
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 10x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 00x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 1101 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) + shifter_output]
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 0101 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – shifter_output]
		″
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 1101 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] ″ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 0101 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] ″ IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 11x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) + shifter_output]
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0111 01x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – shifter_output] ″ shifter_output = If Instr[6:5]=00 then L-shift R(Instr[3:0]) by Instr[11:7] If Instr[6:5]=01 then R-shift R(Instr[3:0]) by Instr[11:7]

		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0101 11x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) + Zext(Instr[11:0])] ˝ IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0101 01x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Byte Aligned M[R(Instr[19:16]) – Zext(Instr[11:0])] ˝ IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0110 10x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16])]
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0110 00x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16])]
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0100 10x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0100 00x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0110 10x1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[15:12])= Word Aligned M[R(Instr[19:16])]
		BUG!! If Word_align=3, not all of shifter_out is defined.
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0110 00x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] BUG!! If Word_align=3, not all of shifter_out is defined. shifter_output = If Instr[6:5]=00 then

		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1a4	xxxx 0100 10x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1a4	xxxx 0100 00x1 xxxx xxxx xxxx xxxx xxxx	R(Instr[15:12])= Word Aligned M[R(Instr[19:16])] BUG!! If Word_align=3, not all of shifter_out is defined. IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0100 1xx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0100 0xx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0101 1xx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0101 0xx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0100 1xx1 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0100 0xx1 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0101 1xx1 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)
DP1b	xxxx 0101 0xx1 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – Zext(Instr[11:0]) IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0110 1xx0 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[19:16])=R(Instr[19:16]) + shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0110 0xx0 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[19:16])=R(Instr[19:16]) – shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0111 1xx0 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) + shifter_output shifter_output = If Instr[6:5]=00 then L-shift R(Instr[3:0]) by R(Instr[11:8])

		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0111 0xx0 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[19:16])=R(Instr[19:16]) – shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0110 1xx1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[19:16])=R(Instr[19:16]) + shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0110 0xx1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[19:16])=R(Instr[19:16]) – shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0111 1xx1 xxxx
	xxxx xxxx xxxx xxxx
		R(Instr[19:16])=R(Instr[19:16]) + shifter_output
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

DP1c	xxxx 0111 0xx1 xxxx xxxx xxxx xxxx xxxx	R(Instr[19:16])=R(Instr[19:16]) – shifter_output shifter_output = If Instr[6:5]=00 then L-shift R(Instr[3:0]) by R(Instr[11:8]) If Instr[6:5]=01 then

		R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by R(Instr[11:8])
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by R(Instr[11:8])
		Note: Only last 5 bits of R(Instr[11:8]) are used
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 1100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 0100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 1100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 0100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 1100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]

		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 0100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 1100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 0100 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 1000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 0000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 1000 xxxx xxxx xxxx xxxx xxxx	M[Instr[19:16]+shifter_output]=R(Instr[15:12]) shifter_output = If Instr[6:5]=00 then L-shift R(Instr[3:0]) by Instr[11:7] If Instr[6:5]=01 then R-shift R(Instr[3:0]) by Instr[11:7]

		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 0000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 1000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0111 0000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 1000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]+shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3a	xxxx 0101 0000 xxxx
	xxxx xxxx xxxx xxxx
		M[Instr[19:16]–shifter_output]=R(Instr[15:12])
		shifter_output = If Instr[6:5]=00 then
		L-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=01 then
		R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=10 then
		Arith. R-shift R(Instr[3:0]) by Instr[11:7]
		If Instr[6:5]=11 then
		R-rotate R(Instr[3:0]) by Instr[11:7]
		IR[31:28] = Condition codes (tracing back to previous instruction)

Store3d	xxxx 01x0 x1x0 xxxx xxxx xxxx xxxx xxxx	M[R(Instr[19:16])]=R(Instr[15:12]) IR[31:28] = Condition codes (tracing back to previous instruction)
Store3d	xxxx 01x0 x0x0 xxxx xxxx xxxx xxxx xxxx	M[R(Instr[19:16])]=R(Instr[15:12]) IR[31:28] = Condition codes (tracing back to previous instruction)

REVE Step 7 and remaining steps

In step 7, we merge similar instructions to form a final RISA. The final RISA for the branch instruction class is shown in Figure 4.8, and the RISA for all remaining instructions is shown below. Analysis for all remaining steps is in Chapter 4.

Instruction	**Assigned Opcode Bits (IR[31:0])**	**Functionality**
Block data transfer	xxxx 100x xxxx xxxx xxxx xxxx xxxx xxxx	Loop instruction through one's count of Instr[15:0]. Count only goes from 0 to 7 however (due to the bug noted in Section 4.4.1). If IR[20]=0: If IR[24]=0: Align(M[R(Instr[19:16]) – 4*(one's count Instr[15:0])]) = R(Instr[15:12]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) – 4*(one's count Instr[15:0]) If IR[23]=0: Add offset after store If IR[23]=1: Add offset before store If IR[24]=1: Align(M[R(Instr[19:16]) + 4*(one's count Instr[15:0])]) = R(Instr[15:12]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) + 4*(one's count Instr[15:0]) If IR[23]=0: Add offset after store If IR[23]=1: Add offset before store If IR[20]=1: If IR[24]=0: R(Instr[15:12]) = Align(M[R(Instr[19:16]) – 4*(one's count Instr[15:0])]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) – 4*(one's count Instr[15:0]) If IR[23]=0: Add offset after load If IR[23]=1: Add offset before load If IR[24]=1: R(Instr[15:12]) = Align(M[R(Instr[19:16]) + 4*(one's count Instr[15:0])]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) + 4*(one's count Instr[15:0]) If IR[23]=0: Add offset after load If IR[23]=1: Add offset before load *Addresses decremented improperly, as noted in Section 4.4.1 Condition codes must be true to execute instruction
Data processing	xxxx 00xx 00xx xxxx xxxx xxxx xxxx xxxx	If IR[20] = 0 set condition codes If IR[25] = 0: R(Instr[15:12]) = R(Instr[19:16]) (opcode) shift(R(Instr[3:0])) Shift = If IR[4] = 0: If IR[6:5]=00: logical left by IR[11:7] If IR[6:5]=01: logical right by IR[11:7] If IR[6:5]=10: arithmetic right by IR[11:7] If IR[6:5]=11: rotate right by IR[11:7]

		If IR[4] = 1 AND IR[7] = 0: If IR[6:5]=00: logical left by R(IR[11:8]) If IR[6:5]=01: logical right by R(IR[11:8]) If IR[6:5]=10: arithmetic right by R(IR[11:8]) If IR[6:5]=11: rotate right by R(IR[11:8]) If IR[25] = 1: R(Instr[15:12]) = R(Instr[19:16]) (opcode) shift(Instr[7:0]) Shift = rotate by IR[11:8] *Incorrect immediate rotate, as noted in Section 4.4.1 Opcode = Instr[24:21] Condition codes must be true to execute instruction
Multiply	xxxx 0000 00xx xxxx xxxx xxxx 1001 xxxx	If IR[20] = 0 set condition codes If IR[21] = 0: R(Instr[19:16]) = R(Instr[11:8]) * R(Instr[3:0]) If IR[21] = 1: R(Instr[19:16]) = R(Instr[11:8]) * R(Instr[3:0]) + R(Instr[15:12]) Condition codes must be true to execute instruction
Single data swap	xxxx 0001 0xxx xxxx xxxx 0000 1001 xxxx	Align(M[R(Instr[19:16])]) = R(Instr[3:0]) R(Instr[15:12]) = Align(M[R(Instr[19:16])]) Align = If IR[22] = 0: Word *Incorrect word alignment as noted in Section 4.4.1 If IR[22] = 1: Byte Condition codes must be true to execute instruction
Single data transfer	xxxx 01xx xxxx xxxx xxxx xxxx xxxx xxxx	If IR[20]=0: If IR[24]=0: Align(M[R(Instr[19:16]) – shift]) = R(Instr[15:12]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) – shift If IR[23]=0: Add offset after store If IR[23]=1: Add offset before store If IR[24]=1: Align(M[R(Instr[19:16]) + shift]) = R(Instr[15:12]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) + shift If IR[23]=0: Add offset after store If IR[23]=1: Add offset before store If IR[20]=1: If IR[24]=0: R(Instr[15:12]) = Align(M[R(Instr[19:16]) – shift]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) – shift If IR[23]=0: Add offset after load If IR[23]=1: Add offset before load If IR[24]=1: R(Instr[15:12]) = Align(M[R(Instr[19:16]) + shift]) If IR[21]=1: R(Instr[19:16]) = R(Instr[19:16]) + shift If IR[23]=0: Add offset after load If IR[23]=1: Add offset before load Align = If IR[22] = 0: Word *Incorrect word alignment as noted in Section 4.4.1 If IR[22] = 1: Byte Shift = If IR[25] = 0: Instr[11:0] If IR[25] = 1: shift R(Instr[3:0]) by Instr[11:4] Condition codes must be true to execute instruction