Université de Montréal

ARM Processor Modeling at a Cycle Accurate Level in SystemC

Par:
Hongmei Sun

Départment d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès Sciences (M. Sc.)
en informatique

Avril, 2003

QA
76
U54
2003
no.021

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

ARM Processor Modeling at a Cycle Accurate Level in SystemC

Présenté Par:
Hongmei Sun

A été évalué par un jury composé des personnes suivantes :

Jean Pierre David
Président-rapporteur

El Mostapha Aboulhamid
Directeur de recherche

François-R Boyer
Co-directeur

Marc Feeley
Membre du jury

Mémoire accepté ______________________

Résumé

La technologie de Simulation/Machine virtuelle est aujourd'hui une partie intégrale de beaucoup de systèmes de calcul. Un simulateur de matériel est un logiciel qui émule les dispositifs câblés spécifiques permettant l'exécution du logiciel qui est écrit et compilé pour ces dispositifs sur les systèmes alternatifs. L'ARM est un microprocesseur RISC 16/32-bit embarqué. Il possède un mécanisme de décalage intégré. L'adressage auto-indexé, les instructions load/store multiple et presque toutes les exécutions d'instructions conditionnels permettent au processeur ARM de réaliser un bon équilibre de haute performance, prix et faible consommation électrique, sur une aire de silicium réduite. L'ARM est largement répandu dans les communications portables, les ordinateurs de poche, le multimédia, produits de consommation numérique et solutions embarquées. Il n'existe pas un simulateur de micro-architecture ARM à source ouvert dans le domaine publique.

Dans ce mémoire, nous explorons différents types de stratégies de simulation (simulation au niveau architecture, exécution directe, recompilation dynamique, code chaîné, simulation d'ensemble d'instructions) et de leurs applications. Nous étudions également le comportement du pipeline du processeur ARM et combinons le comportement du pipeline du ARM [25][26] et celui du DLX [24] pour obtenir une description originale d'exécution de micro-architecture de ARM. En conclusion, nous construisons un simulateur cycle précis du processeur ARM (ARM-Simulator), qui simule la micro-architecture du processeur qui inclut un pipeline 5-stage, chemin d'expédition, la logique de couplage, etc. Ce simulateur est mis en application avec SystemC et des concepts de génie logiciel. Ainsi il est modulaire, facile à étendre, et intégré avec d'autres modules. C'est maintenant un projet autonome de logiciel, après encapsulation appropriée, il peut être un composant à brancher à un système d'application, pour simuler et évaluer la performance.

Le simulateur "modèle de noyau ARM" est validé par un simulateur de jeu d'instructions (ISS) ARM. Nous comparons les résultats du modèle de noyau ARM avec ceux de l'ISS

ARM pour nous assurer que le comportement du simulateur ARM est correct. Le code objet de l'ARM est généré par le compilateur croisé ARM-ELF-GCC. Celui-ci compile le code source C en code objet ARM sous le système d'exploitation Linux. Nous entreprenons également des expériences pour l'évaluation des performances.

Mots-clés: Simulation, Simulateur, Langues de simulation, Stratégies de simulation, Simulateur d'ensemble d'instruction, Microprocesseur RISC, Pipeline, Expédition, Couplage, Compilation croisée

Abstract

Simulation/Virtual machine technology is an integral part of many computing systems today. A hardware simulator is a piece of software that emulates specific hardware devices enabling execution of software that is written and compiled for those devices on alternate systems. ARM is a 16/32-bit embedded RISC microprocessor. It has a built-in shift mechanism. Auto-indexed addressing, load/store multiple instructions and almost all instructions conditional execution allow the ARM processor to achieve a good balance of high performance, low cost, power efficient and low silicon area. ARM is widely used in portable communications, hand-held computing, multi-media, digital consumer and embedded solutions. There doesn't exist an open source ARM micro-architecture simulator in public domain.

In this thesis, we explore different kinds of simulation strategies (Architecture level simulation, Direct Execution, Dynamic Recompilation, Threaded code, Instruction Set Simulation) and their applications. We also study ARM processor pipeline behavior, combine ARM pipeline behavior [25][26] and DLX pipeline [24] to obtain an original description of ARM micro-architecture implementation. Finally, construct a cycle accurate ARM processor simulator (ARM-Simulator), which simulates the micro-architecture of the processor that includes 5-stage pipeline, forwarding path, interlock logic, etc. This simulator is implemented with SystemC and software engineering concepts. So it is modular, easy to extend, and integrated with other modules. It is now a standalone software project, after appropriate encapsulating, it can be a component to plug into an application system, simulate and evaluate performance.

The simulator (ARM core model) is validated by an ARM Instruction Set Simulator (ISS). We compare the result from the ARM core model and the result from the ARM ISS, to know if the behavior of the ARM program is correct. The ARM object code is generated by ARM-ELF-GCC cross compiler. It compiles the C source code to ARM object code on the Linux operating system. We also conduct experiments for evaluating performance.

Keywords: Simulation, Simulator, Simulation Languages, Simulation Strategies, Instruction Set Simulator, RISC Microprocessor, Pipeline, Forwarding, Interlock, Cross Compile

Table of Contents

Résumé i

Abstract iii

Table of Contents v

List of Tables vii

List of Figures viii

Acronyms x

Notations xii

Acknowledgement xiii

Chapter 1 Introduction 1

Chapter 2 Background 8

2.1 Overview of Simulation Strategies 8

2.2 Existing work for ARM processor simulator 9

Chapter 3 ARM Processor Core 12

3.1 ARM Processor Architecture introduction 12

3.2 Processor Modes, Registers and PSRs (Program Status Register) 13

3.3 ARM instructions 18

3.4 Addressing Modes 21

3.5 Organization of the 5-stage ARM Pipeline 29

Chapter 4 Implementation of the ARM core model 33

4.1 All the Instructions operation in different stage (except IF) 33

4.2 Forwarding 41

4.3 Interlock [24][25] 47

4.4 Branch instructions [25][26] 47

4.5 Some special instructions' implementation 49

4.5.1 Load Multiple Registers (LDM) 49

4.5.2 Store Multiple Registers (STM) 51

4.5.3 Swap a word/byte (SWP, SWPB) 52
4.5.4 Multiply instructions (MUL, MLA, UMULL, SMULL) 54
4.5.5 Long multiply and accumulate instructions (UMLAL, SMLAL) 55
4.6 Current and Saved Program Status Register (CPSR, SPSR) 57
4.7 Discussion on generalization 60

Chapter 5 Validation of the model 62
5.1 Methodology of Validation 62
5.1.1 The ARM core model 62
5.1.2 Validate the ARM core model with an ISS (Instruction Set Simulator)...... 64
5.2 Experiments 67
5.3 Summary of this validation 68

Chapter 6 Performance Evaluation 70
Chapter 7 Conclusion 73
References 75
Appendix A: Instructions implemented in this model 79
Appendix B: ARM implementation model, source code description 81
Appendix C: How to use the model 83

List of Tables

Table 2-1: Compare the existing ARM simulators and the ARM Core Model 11

Table 3-1: Processor mode description 14
Table 3-2: Registers [26] 16
Table 3-3: Condition code encoding [26] 21
Table 3-4: Addressing mode 22
Table 3-5: Shift types 23
Table 3-6: Comparison of ARM architectureal pipeline depth 30

Table 4-1: All the instructions operation in different stage 34
Table 4-2: Work of every stage 38
Table 4-3: Destination register of corresponding pipeline register temporary data ALUOutput, D, LMD (for the source instruction of the forwarding). 43
Table 4-4: Source register of pipeline register temporary data A, Bb, C (for the destination instruction of the forwarding). 44
Table 4-5: Forwarding paths 44

Table 5-1: Programs for basic test 67
Table 5-2: Programs for combination test 68

Table 6-1: Performance evaluation of the ARM core model 71

List of Figures

Figure 3.1: ARM instruction set summary [26] 19
Figure 3.2: Multiplies and extra load/store instructions [26] 20
Figure 3.3: Miscellaneous [26] 20
Figure 3.4: Addressing mode 1 data processing instruction binary encoding [26] 23
Figure 3.5: LSL operation 24
Figure 3.6: LSR operation 24
Figure 3.7: ASR operation 24
Figure 3.8: ROR operation 24
Figure 3.9: ROX operation 24
Figure 3.10: Addressing mode 2 Single word and unsigned byte transfer instruction binary encoding [26] 25
Figure 3.11: Addressing mode 3 Half-word and signed byte transfer instruction binary encoding [26] 26
Figure 3.12: Addressing mode 4 Multiple register transfer instruction binary encoding [26] 28
Figure 3.13: Addressing mode 5 Coprocessor data transfer instruction binary encoding [26] ... 29
Figure 3.14: ARM 5-stage pipeline organization [25] 32

Figure 4.1: Forwarding paths 41
Figure 4.2: Instruction sequence for forwarding process 42
Figure 4.3: RAW hazard between two adjacent instructions 47
Figure 4.4: Insert a nop to avoid RAW hazard 47
Figure 4.5: Branch instruction operation 48
Figure 4.6: LDM instruction operation 49
Figure 4.7: STM instruction operation 51
Figure 4.8: SWP instruction operation 53
Figure 4.9: Multiply-1 instruction operation 54
Figure 4.10: Multiply-2 instruction operation 55
Figure 4.11: PSRs (progam status register) operation 57

Figure 5.1: The ARM core model 63

Figure 5.2: Validate the ARM core model 64
Figure 5.3: A program executed on the ARM core model (without library)............................. 65
Figure 5.4: A program executed on ISS (with libraries)... 65
Figure 5.5: AN ARM assembly program after cross compile (Fibonacci) 66

Acronyms

AHB	Advanced High performance Bus
ALU	Arithmetic and Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ARM	Acorn Risc Machine
ASB	Advanced System Bus
ASR	Arithmetic Shift Right
CPI	Clock cycles Per Instruction
DSP	Data Signal Processing
EDA	Electronic Design Automation
EXE	EXEcution
FIQ	Fast Interrupt reQuest
HDL	Hardware Description Language
ID	Instruction Decode
IF	Instruction Fetch
IRQ	Interrupt ReQuest
ISDB	Integrated Services Digital Broadcasting
ISS	Instruction Set Simulator
LR	Link Register
LSL	Logical Shift Left
LSR	Logical Shift Right
MEM	MEMory access
OS	Operating System
OSCI	Open SystemC Initiative
PC	Program Counter
PSR	Program Status Register
RISC	Reduced Instruction Set Computer
ROR	Rotate Right
ROX	Rotate Right with eXtend
RTL	Register Transfer Level

SoC	System on Chip
SP	Stack Pointer
StepNP	System level Telecom Experimental Platform for Network Processing
VCD	Value Change Dump
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
WIF	Waveform Intermediate Format
WB	Write Back

Notations

0x123	hex 123
$IR_{m..n}$	sub range from ‘m bit’ to ‘n bit’ of IR
<<	logical shift left
IR[m]	‘bit m’ of IR
mem[addr]	content of memory in address ‘addr’
Regs[m]	content of register Rm
mul	multiplier in figure 3.14
mux	multiplexer in figure 3.14
%	system prompt
◊	a space

<u>bold & underlined</u>	command line
bold & italic	highlights important notes, introduces special terminology

Acknowledgement

This work has been a challenging experience for me. I learned many existing technologies. I received a lot of support and help from the professors and colleagues in our department.

I would like to thank Professor El Mostapha Aboulhamid, my director, and Professor François-R Boyer, my co-director, for their continued guidance, encouragement, enthusiasm, and support are greatly appreciated. Also great thanks to the colleagues for their encouragement and help.

Chapter 1 Introduction

Role of simulation in the design flow

Simulation / Virtual machine technology [1] is an integral part of many computing systems today. Java, SimOS [2][3] and VMware [for running a complete OS as an application on another operating system], Connectix Virtual PC/Game station and Microsoft's .NET are different examples of such systems. This technology is incredibly useful as a secure means for execution of untrusted software in a sandbox environment, and an ideal platform for code-development for new hardware devices.

A hardware simulator is a piece of software that emulates specific hardware devices, enabling execution of software that is written and compiled for those devices on alternate systems. The machines that are simulated will be referred to as *target machines*, and the system on which the simulator is actually running is referred to as the *host machine [4]*.

Simulation at various levels of abstraction has played a key role in the design of computer systems. There are numerous compelling reasons for implementing simulators, most of them obvious. Design teams need simulators throughout all phases of the design cycle.

1. Initially, during high-level design, simulation is used to narrow the design space and establish credible and feasible alternatives that are likely to meet competitive performance objectives.
2. Later, during microarchitectural definition, a simulator helps guide engineering trade-offs by enabling quantitative comparison of various alternatives.
3. During design implementation, simulators are employed for testing, functional validation, and late-cycle design trade-offs.
4. Finally, simulators provide a useful reference for performance validation once real hardware becomes available.

Outside of the industrial design cycle, simulators are also heavily used in the computer architecture academic research community.

Some benefits of using simulators are:

1. Simulators are flexible and thus new features or components can easily be added. It is possible to model features, including those that might not be possible to do on the hardware.
2. It allows stress testing of programs like operating systems by simulating complex interrupt and exception conditions.
3. Since simulators are built in software, they are more deterministic. The deterministic behavior of simulators makes programs execution reproducible, and thus helps in locating problems.

Primary applications of simulators consist of computer architecture studies and performance tuning of compiled software, and the compilation process itself. Various types of simulators exist, each addressing different aspects like clock cycle rate, modeling the microprocessor chip logic, modeling the program execution environment, etc.

StepNP (System-level Telecom Experimental Platform for Network Processing) is a modeling platform for multiprocessor and network processors, under development. The platform will help in performance evaluation, design exploration by changing architecture parameters such as interconnects, pipeline characteristics, instruction sets and memory schemes. The platform should be flexible and modular to allow a 'plug, simulate and evaluate' approach. This allows plugging different kinds of processors and interconnections to evaluate and create a new system. Core models include ARM processor [26] and DLX processor [24]. Interconnects models include the AMBA bus [28] and different interconnect topologies such as chordal rings [5], crossbars, rings etc. The ARM core model implemented in this work was originally a component in this platform. Now it is a standalone project.

ARM is the industry's leading provider of 16/32-bit embedded RISC microprocessor solutions. ARM's microprocessor cores are high-performance, low-cost, power efficient. They are rapidly becoming the volume RISC standard in such markets as portable communications, hand-held computing, multimedia digital consumer and embedded solutions.

Simulation languages

Simulation allows faster development of design and cheaper and easier debugging during the design stage. Hardware description languages provide support for the simulation of concurrent processes and offer constructs to describe inter-process signal transfer. Two hardware simulation languages have become standard design entry tools for simulation and synthesis-Verilog [34] and VHDL [33]. While the results are very accurate the speed of simulation is very slow since they are more suitable for RT (Register Transfer) level design than for higher levels of abstraction. More powerful than these simulation languages are SystemC [35] and Superlog [32]. SystemC was launched by Synopsys and CoWare, it is backed by some 30 electronic design automation (EDA) vendors, which formed the Open SystemC Initiative (OSCI). This independent standards organization promotes SystemC as a common digital framework that will streamline product development for EDA synthesis tools. Superlog is another language for system-level design, developed by CoWare and based on Verilog and C. In this work, we implement an ARM processor simulator using SystemC to obtain an open source core of ARM which is cycle accurate and demonstrate the design flow of SystemC on a real example including validation.

SystemC [27] is a C++ class library and a methodology that we can use to effectively create a model of software algorithms, hardware architecture, and interfaces of SoC (System On a Chip) and system-level designs. We can use SystemC and standard C++ development tools to create a system-level model, quickly simulate to validate and optimize the design, explore various algorithms, and provide the hardware and software development team with an executable specification of the system. SystemC supports hardware-software co-design and the description of the architecture of complex systems consisting of both hardware and software components. It supports the description of hardware, software, and interfaces in a C++ environment. The following features of SystemC allow it to be used as a co-design language:

1. Modules: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
2. Processes: Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions (method process, thread process, clocked thread process) to be used by hardware and software designers.
3. Ports: Modules have ports through which they connect to other modules. SystemC supports single-direction and bi-directional ports.
4. Signals: SystemC supports resolved and unresolved signals. Resolved signals (a bus) can have more than one driver while unresolved signals can have only one driver.
5. Rich set of port and signal types: To support modeling at different levels of abstraction, from the functional to the RTL, SystemC supports a rich set of port and signal types. This is different than languages like Verilog that only support bits and bit-vectors as port and signal types.
6. Rich set of data types: SystemC has a rich set of data types to support multiple design domains and abstraction levels. The fixed precision data types allow for fast simulation, the arbitrary precision types can be used for computations with large numbers, and the fixed-point data types can be used for DSP applications. SystemC supports both two-valued and four-valued data types. There are no size limitations for arbitrary precision SystemC types.
7. Clocks: SystemC has the notion of clocks (as special signals). Clocks are the timekeepers of the system during simulation. Multiple clocks, with arbitrary phase relationship, are supported.
8. Cycle-based simulation: SystemC includes a cycle-based simulation kernel that allows high-speed simulation.
9. Multiple abstraction levels: SystemC supports untimed models at different levels of abstraction, ranging from high-level functional models to detailed clock cycle

accurate RTL models. It supports iterative refinement of high level models into lower levels of abstraction.

10. Communication protocols: SystemC provides multi-level communication semantics that enable us to describe SoC and system I/O protocols with different levels for abstraction.
11. Debugging support: SystemC classes have run-time error checking that can be turned on with a compilation flag.
12. Waveform tracing: SystemC supports tracing of waveforms in VCD, WIF, and ISDB formats.

Using the SystemC approach, the designer does not have to be an expert in multiple languages. SystemC allows modeling from the system level to RTL. The SystemC approach provides higher productivity because the designer can model at a higher level. Writing at a higher level can result in smaller code, which is easier to write and simulates faster than traditional modeling environments. Testbenches can be reused from the system level model to the RTL model saving conversion time. Using the same testbench also gives the designer a higher confidence that the system level and the RTL model implement the same functionality.

The objective of this research is to:

- Explore different kinds of processor simulator and their applications.
- Study ARM processor pipeline behavior, combine ARM pipeline behavior [25][26] and DLX pipeline [24] to obtain an original description of ARM micro-architecture implementation.
- Construct a cycle accurate ARM processor simulator with 5-stage pipeline (ARM-Simulator). Which simulate the micro-architecture of the processor that includes 5-stage pipeline, forwarding path, interlock logic etc. After encapsulating, it will be a component of StepNP modeling platform, can be plugged, simulated and evaluated in different applications. The ARM core model is implemented by using

SystemC and software engineering concepts. So it is modular, easy to extend, and integrated with other modules.

- Validate the ARM core model using an ISS of ARM processor at instruction level. Because the ISS and the ARM core model do not have the same precision, so we validate it at instruction level to ensure the behavior of ARM program is correct.
- Evaluate the performance of the cycle accurate ARM processor simulator.

This thesis is divided into seven chapters. Its content is organized as follows:

Chapter 2 provides background knowledge. At first, it gives an overview of simulation techniques, latter, it introduces the existing work for ARM processor simulator.

Chapter 3 introduces the ARM processor, ARM architecture, ARM instructions sets, addressing mode of different instructions and 5-stage pipeline organization.

Chapter 4 presents the implementation of the ARM core model. It introduces how to implement the 5-stage pipeline, the work of every pipeline stage for every instruction. Then, it explains the execution of forwarding, interlock and branch instructions. It also describes the execution of some special instructions. The details of this description can be considered as original since they are not described in the specification of the ARM. At last, it discusses the generalization of the model.

Chapter 5 describes the methodology for validating the model. The ARM core model is validated by an ISS (Instruction Set Simulator). Arm-elf-gcc is the cross compiler that generates the arm object code. Then the object code is run on the ARM core model and ISS. By comparing the results from different model to know if the ARM core model is valid. Since ISS and the ARM core model (cycle accurate levels) do not have the same precision, so this validation is only from the instruction level. We also use some special examples to validate the architecture of the processor core, includig functional units, forwarding path, interlock logic etc. Then it gives some experiments for validating it.

Chapter 6 introduces the ARM core model performance metrics. We do some experiments, compare the execution time of ISS and the ARM core model for the same workload, count the number of cycles and instructions so as to compare the compiled code quality from different compilers, calculate the CPI to evaluate the performance of an application system.

Chapter 7 concludes with the features of the model and the contributions of this thesis and then discusses future work.

Chapter 2 Background

This section gives an overview of simulation strategies and existing work for the ARM processor simulator.

2.1 Overview of Simulation Strategies

The best simulation method depends on the application of the simulation results. This section outlines several simulation strategies and their applications.

Architectural level Simulation [6][7][8]:

Logic designers build Architectural simulators to express and test new designs. These allow emulation of the different parts of a processor, using either the simple core, or the core and the data caches and other components. These are generally not intended for executing target system binaries on alternate platforms, but rather to allow research into the modification of the internal data-paths of the processor.

Direct Execution [1]:

Target machine binaries can be executed natively on the simulator host processor by encasing the program in an environment that makes it execute as though it were on the simulated system. This technique requires that either the host system has the same instruction set as the target, or that the program be recompiled for the host architecture. Instructions that cannot execute directly on the host are replaced with procedure calls to simulator code. This method is also known as Dynamic Recompilation [*Dynarecs*]. Native execution of the recompiled code leads to a much faster execution of the simulated software, but they have lengthy context switching, i.e. when the host processor has to switch to target processor. This may slow down the simulation.

Threaded Code [9][10]:

This is a simulation technique where each op-code in the target machine instruction set is mapped to the address of some (lower level) code in the simulator system, to perform the appropriate operation. This can be implemented efficiently in machine code on most processors by simply performing an indirect jump to the address, which is the next instruction. This method does not suffer from lengthy context switching.

Instruction set simulators [4]:

Instruction set simulator [ISS] executes target machine programs by simulating the effects of each instruction on a target machine, one instruction at a time. The Instruction sets simulators are attractive for their flexibility: they can, in principle, model any computer, gather any statistics, and run any program that the target architecture would run. They easily serve as backend systems for traditional debuggers as well as architecture design tools such as cache simulators. A lot of temporal debuggers have recently started using ISS. An ISS can dispatch instructions by fetching from a simulated memory, isolating the operation code fields, and also branching, based on the values of these fields. Once dispatched, reading and manipulation of variables that represent the target system's state are used to simulate the instruction's semantics. They are not cycle-accurate since they do not take into account the iteming of an instruction pipelining.

2.2 Existing work for ARM processor simulator

There has been a lot of research on software simulation of the ARM processor. These can be categorized according to the level of simulation, whether at the architectural level or the instruction set, or the techniques used, e.g. dynamic recompilation of parts of the simulated software to natively run on the guest system.

Dynarecs [ARMphetamine]

ARMphetamine [1] and tARMac [11] are based on the direct compilation technique. They are fast and accurate ARM emulators. ARM code program segments are translated into native code as they are being emulated. A fetch-decode-execute emulator starts executing the ARM code, and when a specific block of the ARM code has been executed more times than a preset threshold, a translation routine is employed. This generates covers for each source instruction, i.e. chunks of native code that have the same semantics as the translated instructions. These covers are then executed every time the translated block of code needs to be run. The development platform for ARMphetamine and tARMac is linux/x86. They are open source, but not cycle accurate, so can't be used for performance evaluation.

Architecture level [SWARM][6]

SWARM was designed as an ARM module to plug into the SimOS system developed at Stanford University. SimOS allows emulation of various parts of an ARM processor, using either the simple core, or the core and the caches. SWARM was intended not for running ARM binaries on an alternate platform, but rather to allow research into the modification of the internal data-paths of the ARM processor. It implements a small amount of internal co-processors at a basic level, and provides support for the full register/cache/external memory hierarchy. It does not take into account the micro-architecture of ARM processor core and the pipelining execution of an instruction.

Instruction Level [SimARM][12][13]

SimARM [12] is an instruction set simulator (ISS) that interprets ARM programs at the instruction level obviating the need for ARM hardware. ISSs are simpler to implement, but they are slower than simulators based on dynarecs due to the fact that all instructions are strictly interpreted.

ARMulator [13] is another ISS with a slight variation: it ensures identical cycle-count for instructions. This means that instructions take the same number of simulator's cycles to execute as if run on real ARM hardware. This is important for precise simulation since some compilers can optimize code that takes advantage of the cycle-counts of specific instructions. But it is not open source to the public, can't be integrated into SystemC.

In this work, we implement the ARM Core Model, a cycle-accurate micro-architecture simulator. It simulates ARM instruction sets, and also simulates 5-stage pipeline (including hazard detecting, forwarding path, interlock logic and automatic no-op insert). It can act as an ISS to execute ARM program by simulating the behavior of the program. It is a modular design, so it can be integrated in SystemC, as a component can be plugged into an application system. It can also evaluate system performance by counting the number of cycles for executing a program; it can also compare the quality of code compiled by different compilers. It also takes into account all the pipeline effects such as hazard detecting, data forwarding, interlock, automatic no-op inserting etc. Table 2.1 compares the existing ARM simulators and the ARM Core Model (implemented in this work)

Table 2-1: Compare the existing ARM simulators and the ARM Core Model

	ARM Core Model	ARMsim	simARM	ARMulator	SWARM	ARMphetamine
Open source	Yes	No	No	No	Yes	Yes
Simulation level or technique	Cycle-accurate micro-architecture	ISS	ISS	ISS	Architecture	Architecture/ dynamic compile
Cycle-accurate	Yes	No	No	Yes	No	No
Performance	Low	Medium	Medium	Medium	Low	High
Thumb support	No	No	No	Yes	No	No
Library support	No	No	No	Yes	Not all	No

Chapter 3 ARM Processor Core

This section introduces the architecture of ARM processor, its processor modes, registers group, instruction encoding and addressing mode for different instruction, and then describes the 5-stage ARM pipeline organization [31][25][26].

3.1 ARM Processor Architecture introduction

The ARM is a Reduced Instruction Set Computer (RISC), as it incorporates these typical RISC architecture features:

1. A large uniform register file.
2. Load-store architecture, data-processing operations only operate on registers contents, not directly on memory contents.
3. Simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.
4. Uniform and fixed-length instruction fields, to simplify instruction decode.

In addition, the ARM architecture has following characteristics:

1. The ARM is a 32-bit machine with a register-to-register, three-operand instruction set. All operands are 32 bits wide.
2. Control over both the Arithmetic Logic Unit (ALU) and shifter in every data-processing instruction to maximize the use of an ALU and a shifter.
3. Auto-increment and auto-decrement addressing modes to optimize program loops.
4. Load and Store Multiple instructions to maximize data throughput.
5. Conditional execution of all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, low code size, low power consumption and low silicon area.

3.2 Processor Modes, Registers and PSRs (Program Status Register)

ARM processor has seven processor modes (User mode, FIQ mode, IRQ mode, Supervisor mode, Abort mode, Undefined mode, System mode). Every processor mode has different banked general register group. Also there is a CPSR (Current Program Status Register) and a SPSR (Saved Program Status Register) except system mode and user mode (there is no SPSR for system mode and user mode).

Processor modes:

ARM supports five types of exceptions, and a privileged processing mode for each type. The five types of exceptions are:

1. Fast interrupt
2. Normal interrupt
3. Memory aborts, which can be used to implement memory protection or virtual memory
4. Attempted execution of an undefined instruction
5. Software interrupt (SWI) instructions which can be used to make a call to an operating system.

When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have replacement banked registers for R13 and R14. The fast interrupt mode has more registers for fast interrupt processing.

When an exception handler is entered, R14 holds the return address for exception processing. This is used to return after the exception is processed and to address the instruction that caused the exception.

Register R13 is banked across exception modes to provide each exception handler with a private stack pointer. The fast interrupt mode also banks registers R8 to R12 so that interrupt processing can begin without the need to save or restore these registers.

There is a sixth privileged processing mode, System mode, which uses the User mode registers. This is used to run tasks that require privileged access to memory and/or coprocessors, without limitations on which exceptions can occur during task.

All the processor modes are described in Table 3.1

Table 3-1: Processor mode description

Name	Processor mode	Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks

Registers:

The ARM has 15 user-accessible general-purpose registers called R0 to R14 and a current program status register (CPSR) and a program counter R15.

The ARM processor has a total of 37 registers:

- 31 general-purpose registers, including a program counter.
- 6 status registers.

These registers are 32 bits wide. Registers are arranged in partially overlapping banks, with a different register bank for each processor mode. At any time, 15 general-purpose registers (R0-R14), one or two status registers and the program counter are visible.

The general-purpose registers R0-R15 can be split into three groups. These groups differ in the way they are banked and in their special-purpose uses:

- The unbanked registers R0-R7
- The banked registers R8-R14
- R15 is the PC (Program Counter)

Banked register means physical address of the register depends on processor mode. Unbanked register means physical address of the register doesn't depend on processor mode.

The unbanked registers R0-R7:

Each of them refers to the same 32-bit physical register in all processor modes. They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

The banked registers R8-R14:

The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.

R15 is the PC (Program Counter):

When an instruction reads R15, the value read is the address of the instruction plus 8 bytes. All the registers are described in Table 3.2.

PSRs (Program Status Registers):

The current program status register (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a saved program status register (SPSR) that is used to preserve the value of the CPSR when the associated exception occurs. User mode and System mode do not have an SPSR, because they are not exception modes.

Table 3-2: Registers [26]

Modes						
	Privileged modes					
		Exception mode				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R13_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Program Status Registers

31	30	29	28	27	26 8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM(RAZ)	I	F	T	M4	M3	M2	M1	M0

The condition code flags in PSR:

The N, Z, C, V (Negative, Zero, Carry and overflow) bits are collectively known as the condition code flags. The condition code flags in the CPSR can be tested by most instructions to determine whether the instruction is to be executed.

The condition code flags are usually modified by:

- Execution of a comparison instruction (CMN, CMP, TEQ, TST).
- Execution of some other arithmetic, logical or move instruction, where the destination register of the instruction is not R15. Most of these instructions have both a flag-preserving and a flag-setting variant, with the latter being selected by

adding an S qualifier to the instruction mnemonic. Some of these instructions only have a flag-preserving version.

N: is set to bit 31 of the result of the instruction.

Z: is set to 1 if the result of the instruction is 0.

C: is set in one of four ways:

- For an addition, including the comparison instruction CMN (CoMpare Negative), C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, C is set to 0 if the subtraction produced a borrow (that is , an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the register by the shifter.
- For other non-addition/subtractions, C is left unchanged.

V: is set in one of two ways:

- For an addition or subtraction, V is set to 1 if signed overflow occurred.
- For non-addition/subtraction, V is normally left unchanged.

The condition flags can be modified in these additional ways:

- Execution of an MSR (Move to PSR from general-purpose Register) instruction, as part of its function of writing a new value to the CPSR or SPSR.
- Execution of MRC (Move to ARM Register from Coprocessor) instructions with destination register R15. The purpose of such instructions is to transfer coprocessor-generated condition code flag values to the ARM processor.
- Execution of some variants of the LDM instruction. These variants copy the SPSR to the CPSR, and their main intended use is for returning from exceptions.
- Execution of flag-setting variants of arithmetic and logical instructions whose destination register is R15. These also copy the SPSR to the CPSR, and are mainly intended for returning from exceptions.

The control flags in PSR are:

- I: Disables IRQ interrupts when it is set.
- F: Disables FIQ interrupts when it is set.
- T: 0 (ARM execution), 1 (Thumb execution)
- Mode bits: M4…M0: Processor Mode. 0x10 (User), 0x11 (FIQ), 0x12 (IRQ), 0x13 (Supervisor), 0x17 (Abort), 0x1b (Undefined), 0x1f (System).

Other bits in the Program Status Registers are reserved for future expansion.
The format of PSR is described in table 3.2.

3.3 ARM instructions

ARM instructions are 32 bit fixed-length RISC instruction set. Figure 3.1 shows the ARM architecture version 5 instruction set encoding.
Figure 3.2 shows multiplies and extra load/store instructions
Figure 3.3 shows miscellaneous instructions

Almost all instructions can be conditionally executed; which means that they only have their normal effect on the programmer's model state, memory and coprocessors if the N, Z, C and V flags in the CPSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP. Table 3.3 shows the condition code encoding.

Appendix A shows all the instructions implemented in this model.

	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 immediate shift	cond	000	opcode				S	Rn	Rd	Shift amount		shift	0	Rm
Miscellaneous instructions: See figure 3-3	cond	000	10xx				0	xxxxxxxxxxxxxxx					0	xxxx
Data processing register shift	cond	000	opcode				S	Rn	Rd	Rs	0	shift	1	Rm
Miscellaneous instructions: See figure 3-3	cond	000	10xx				0	xxxxxxxxxxxx			0	xx	1	xxxx
Multiplies, extra load/stores: See figure 3-2	cond	000	xxxxxxxxxxxxxxxxx								1	xx	1	xxxx
Data processing immediate	cond	001	opcode				S	Rn	Rd	rotate	immediate			
Undefined instruction	cond	001	10		x	00		xxxxxxxxxxxxxxxxxxxx						
Move immediate to status register	cond	001	10		R	10		mask	SBO	rotate	immediate			
Load/store immediate offset	cond	010	P	U	B	W	L	Rn	Rd	immediate				
Load/store register offset	cond	011	P	U	B	W	L	Rn	Rd	Shift amount		shift	0	Rm
Undefined instruction	cond	011	xxxxxxxxxxxxxxxxxxxx										1	xxxx
Undefined instruction	1111	0	xxxxxxxxxxxxxxxxxxxxxxxxxxx											
Load/store multiple	cond	100	P	U	S	W	L	Rn	Register list					
Undefined instructions	1111	100	xxxxxxxxxxxxxxxxxxxxxxxxx											
Branch and branch with link	cond	101	L	24-bit offset										
Branch and branch with link And change to thumb	1111	101	H	24-bit offset										
Coprocessor load/store and double Register transfers	cond	110	P	U	N	W	L	Rn	CRd	cp-num	8-bit offset			
Coprocessor data processing	cond	1110		opcode1				CRn	CRd	cp-num	opcod2		0	CRm
Coprocessor register transfers	cond	1110		opcode1			L	CRn	CRd	cp-num	opcod2		1	CRm
Software interrupt	cond	1111		swi number										
Undefined instruction	1111	1111		xxxxxxxxxxxxxxxxxxxxxxxx										

Figure 3.1: ARM instruction set summary [26]

	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
Multiply (accumulate)	cond	0000		00		A	S	Rd	Rn	Rs	1001				Rm
Multiply (accumulate) long	cond	0000		1	U	A	S	RdHi	RdLo	Rs	1001				Rm
Swap/swap byte	cond	0001		0	B	00		Rn	Rd	SBZ	1001				Rm
Load/store halfword Register offset	cond	000	P	U	0	W	L	Rn	Rd	SBZ	1011				Rm
Load/store halfword Immediate offset	cond	000	P	U	1	W	L	Rn	Rd	HiOffset	1011				LoOffset
Load/store two words register offset	cond	000	P	U	0	W	0	Rn	Rd	SBZ	1	1	S	1	Rm
Load signed halfword/byte Register offset	cond	000	P	U	0	W	1	Rn	Rd	SBZ	1	1	H	1	Rm
Load/store two words immediate offset	cond	000	P	U	1	W	0	Rn	Rd	HiOffset	1	1	S	1	LoOffset
Load signed halfword/byte Immediate offset	cond	000	P	U	1	W	1	Rn	Rd	HiOffset	1	1	H	1	LoOffset

Figure 3.2: Multiplies and extra load/store instructions [26]

	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
Move status register to register	cond	00010	R	0	0	SBO	Rd	SBZ	0000	SBZ
Move register to status register	cond	00010	R	1	0	mask	SBO	SBZ	0000	Rm
Branch/exchange instruction set	cond	00010	01		0	SBO	SBO	SBO	0001	Rm
Count leading zeros	cond	00010	11		0	SBO	Rd	SBO	0001	Rm
Branch and link/exchange	cond	00010	01		0	SBO	SBO	SBO	0011	Rm
Enhanced DSP add/subtracts	cond	00010	op		0	Rn	Rd	SBZ	0101	Rm
Software breakpoint	cond	00010	01		0	immed			0111	immed
Enhanced DSP multiplies	cond	00010	op		0	Rd	Rn	Rs	1yx0	Rm

Figure 3.3: Miscellaneous [26]

Table 3-3: Condition code encoding [26]

Opcode $IR_{31..28}$	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry Set/unsigned Higher or Same	C set
0011	CC/LO	Carry Clear/unsigned LOwer	C clear
0100	MI	MInus/negative	N set
0101	PL	PLus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed Greater than or Equal	(N=V)
1011	LT	Signed Less Than	(N != V)
1100	GT	Signed Greater Than	Z=0 or N!=V
1101	LE	Signed Less than or Equal	Z=1 or N!=V
1110	AL	ALways (unconditional)	
1111	NV	Depends on architecture version	

3.4 Addressing Modes

ARM Instructions have 5 kinds of addressing modes. Different addressing mode is used in different instructions. Table 3.4 shows the addressing modes and their usage.

Table 3-4: Addressing mode

Addressing mode	Usage
Addressing Mode 1	Data-processing operands
Addressing Mode 2	Load and Store Word or Unsigned Byte
Addressing Mode 3	Miscellaneous Loads and Stores
Addressing Mode 4	Load and Store Multiple
Addressing Mode 5	Load and Store Coprocessor

The following is more detailed description:

Addressing Mode 1 (figure 3.4):

Addressing Mode 1 is used in Data-Processing instructions to generate the second operand (Shifter Operand). Shifter-Operand could be:

Immediate: shifter_operand=8-bit immediate Rotate_Right (#rot * 2)

if #rot==0 then shifter_carry_out=C_flag

else shifter_carry_out=shifter_operand[31]

(it is also the last bit shifted out of the value by the shifter, see figure 3.8)

Register: if $IR_{11..4}$=0, then shifter_operand=Rm

shifter_carry_out=C_flag

Scaled Register: Rm is shifted by the amount of Rs or #shift.

The shift type (table 3.5) can be: LSL (00), LSR (01), ASR (10), ROR (11), ROX (11) ($IR_{11..7}$=0).

Table 3-5: Shift types

Shift	Description
LSL (figure 3.5)	Logical Shift Left
LSR (figure 3.6)	Logical Shift Right
ASR (figure 3.7)	Arithmetic Shift Right
ROR (figure 3.8)	ROtate Right
ROX (figure 3.9)	ROtate right with eXtend

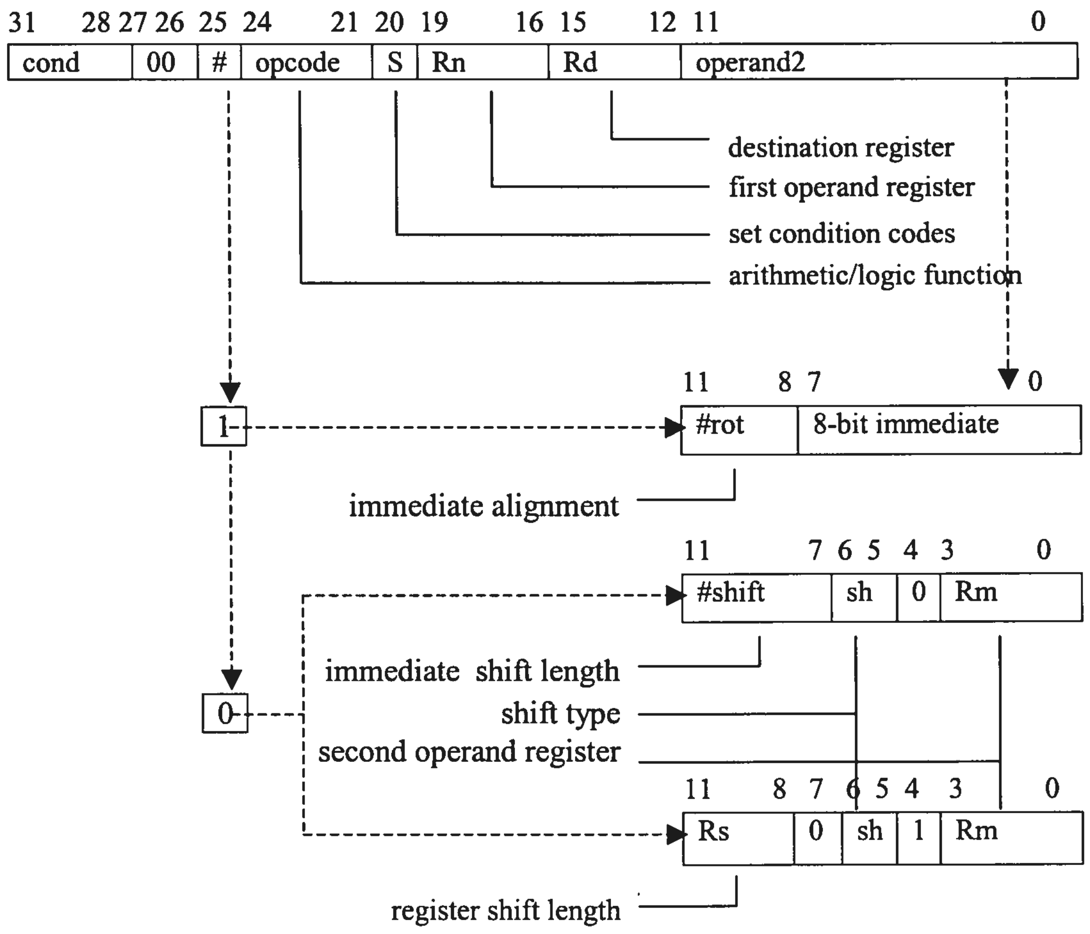


Figure 3.4: Addressing mode 1 data processing instruction binary encoding [26]

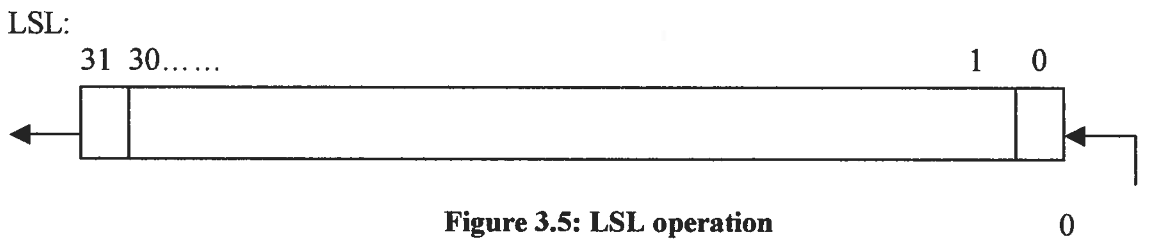


Figure 3.5: LSL operation

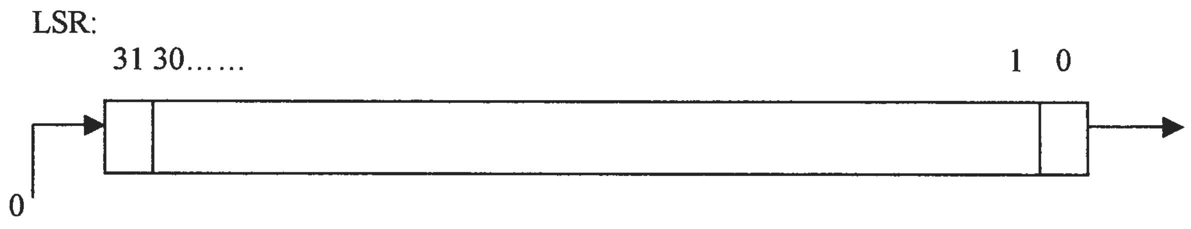


Figure 3.6: LSR operation

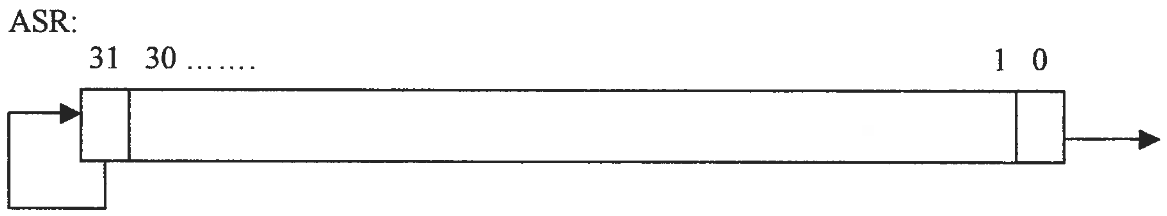
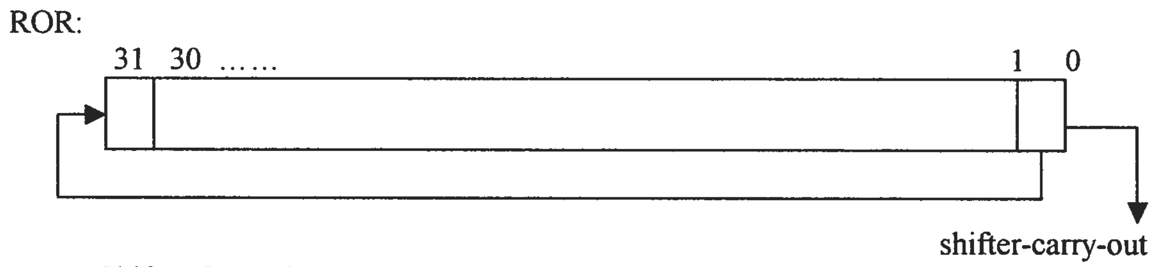
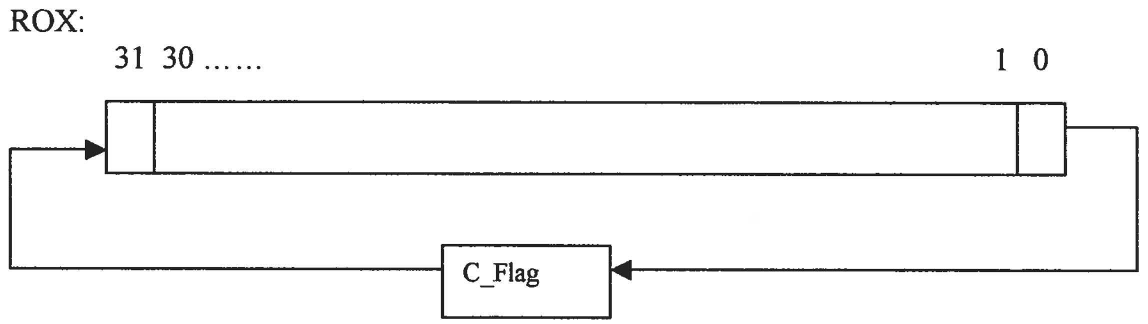


Figure 3.7: ASR operation



Shifter-Carry-Out=Rm[2*rot-1]

Figure 3.8: ROR operation



Shifter-Carry-Out=Rm[0]

Figure 3.9: ROX operation

Addressing Mode 2 (figure 3.10):

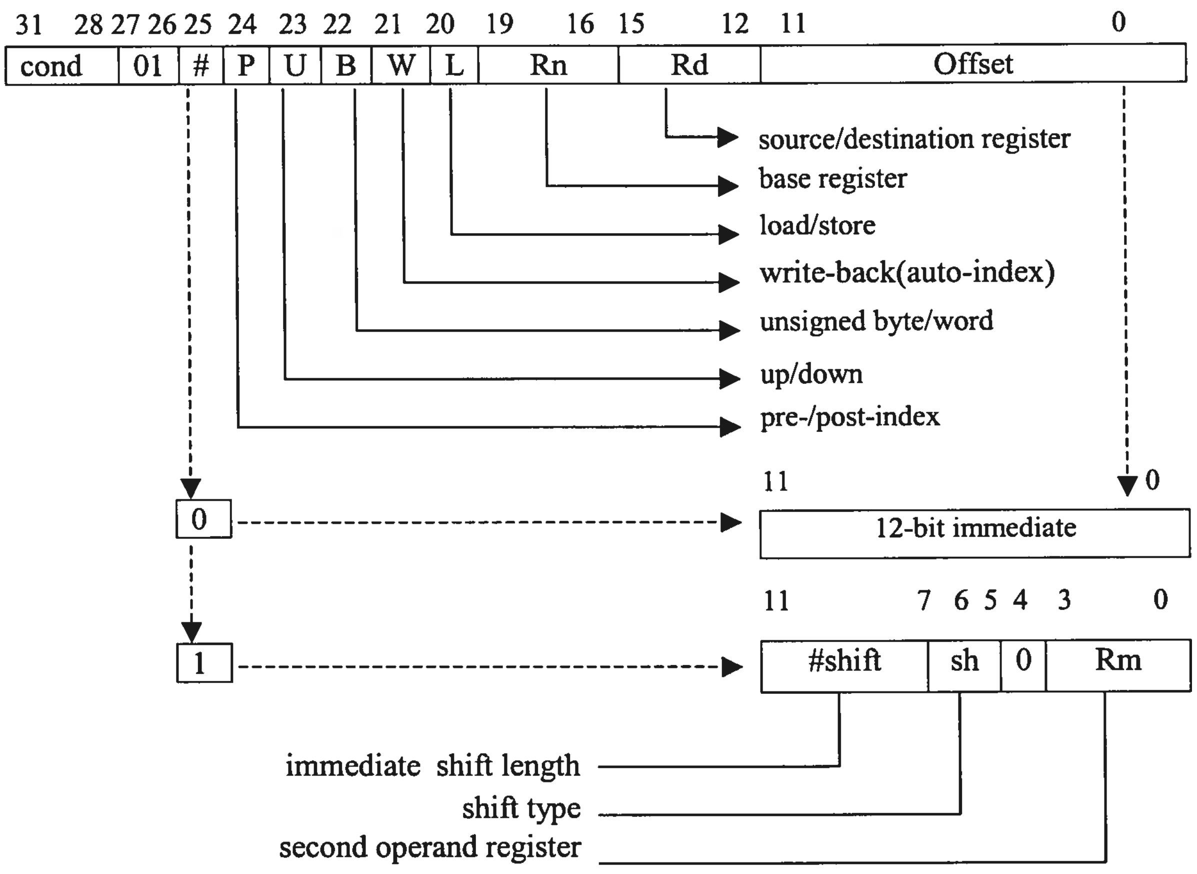


Figure 3.10: Addressing mode 2 Single word and unsigned byte transfer instruction binary encoding [26]

It is used in Load/Store Word/Unsigned Byte instructions addressing.

The instructions are: LDR, LDRB, LDRBT, LDRT, STR, STRB, STRBT, STRT.

P and W combination decides the index mode:

P=0: post-index

W=0: post index

W=1: LDRT, only post index

P=1: pre-index and register offset

W=0: register offset

W=1: pre-index

Offset could be:

- Immediate offset: offset=12-bit immediate
- Register offset: $IR_{11..4}$=0 then offset=Rm
- Scaled register offset: Rm is shifted by the amount of #shift.

U is used to indicate Rn Plus (U=1) or Minus (U=0) the offset.

B is used to indicate Unsigned Byte (B=1) or Word (B=0).

L is Load (L=1) or Store (L=0) operation.

Addressing mode 3 (figure 3.11):

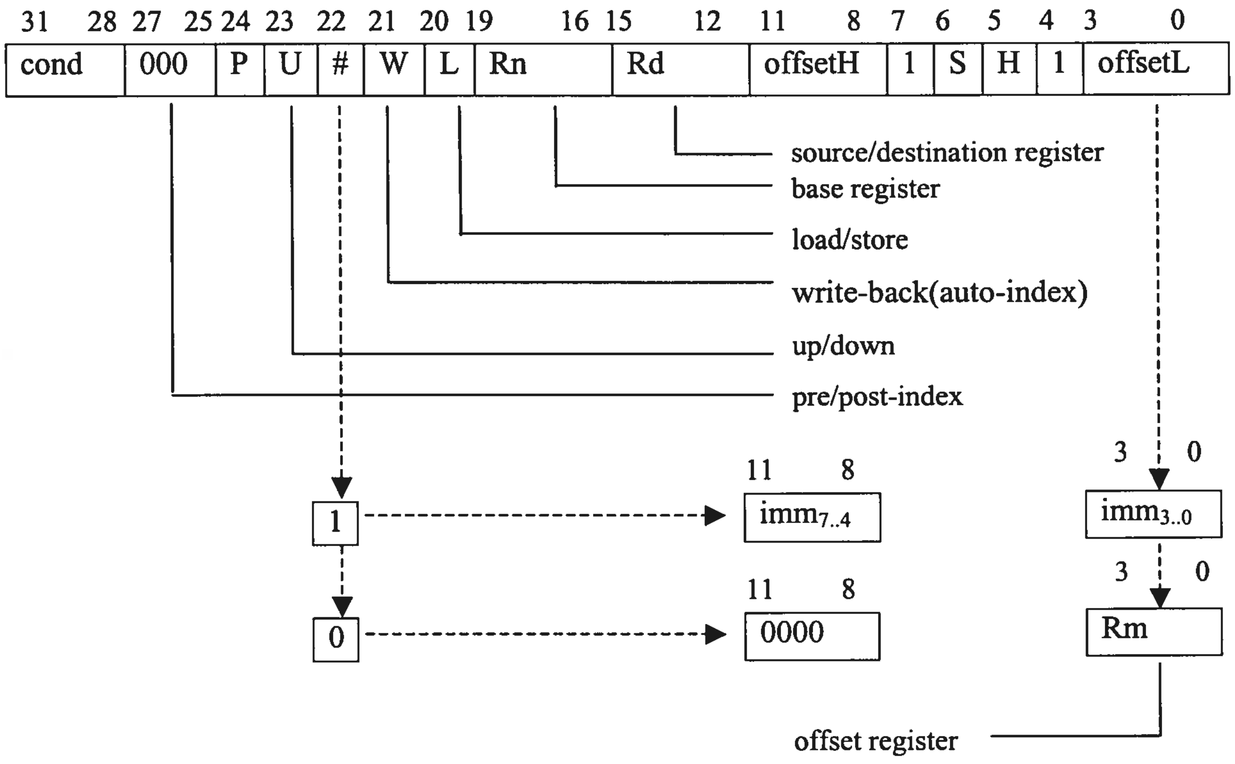


Figure 3.11: Addressing mode 3 Half-word and signed byte transfer instruction binary encoding [26]

It is used in load/store half word, signed half word signed byte and double word instructions. The instructions are: LDRH, STRH, LDRSH, LDRSB, LDRD, STRD.

P and W combination is similar with addressing mode 2.

U: Indicates whether the offset is added (U=1) to the base or subtracted (U=0) from the base.

L: Indicates Load (L=1) or Store (L=0) instruction.

S: It distinguishes Signed (S=1) or Unsigned (S=0) half-word access.

H: Indicates Half-word (H=1) or Byte (H=0) access.

Offset could be:

- Immediate offset: offset=(offsetH << 4) or offsetL
- Register offset: offset=Rm

Addressing mode 4 (figure 3.12):

It is used in load/store multiple instructions.

Load Multiple instructions load a subset (possibly all) of the general-purpose registers from memory. Store Multiple instructions store a subset (possibly all) of the general purpose registers to memory.

Load and Store Multiple addressing modes produce a sequential range of addresses. The lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address.

The general instruction syntax is:

```
LDM|STM {<cond>}<addressing_mode> <Rn>, <registers>
```

addressing_mode is one of the following 4 addressing modes:

- IA: P=0 U=1, Start_address=Rn, End_address=Rn+4*N-4
- IB: P=1 U=1, Start_address=Rn+4, End_address=Rn+4*N
- DA: P=0 U=0, Start_address=Rn-4*N+4, End_address=Rn
- DB: P=1 U=0, Start_address=Rn-4*N, End_address=Rn-4

N: is the number of set bits in register list.

P: 0 (include Rn)

1 (exclude Rn)

Rn: when W=1, change the base register Rn

U=1: Rn is set to Rn+4*N

U=0: Rn is set to Rn-4*N

L: Indicates Load (L=1) or Store (L=0) operate.

S: For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

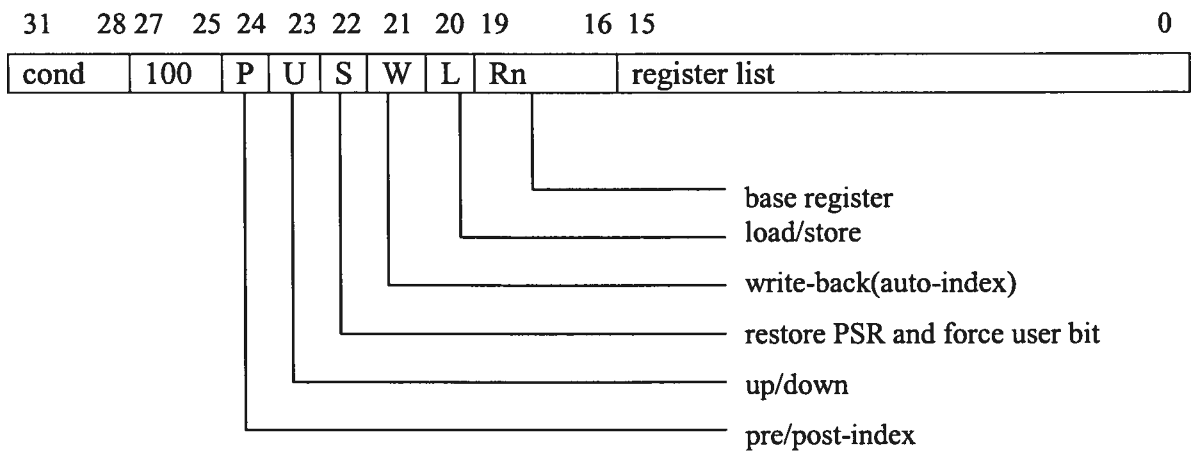


Figure 3.12: Addressing mode 4 Multiple register transfer instruction binary encoding [26]

Addressing mode 5 (figure 3.13):

It is used in load/store coprocessor instructions. LDC and STC.

The combination of P and W is similar with addressing mode 2.

U and L have the same meaning with addressing mode 2.

N: is coprocessor-dependent. Its recommended use is to distinguish between different-sized values to be transferred.

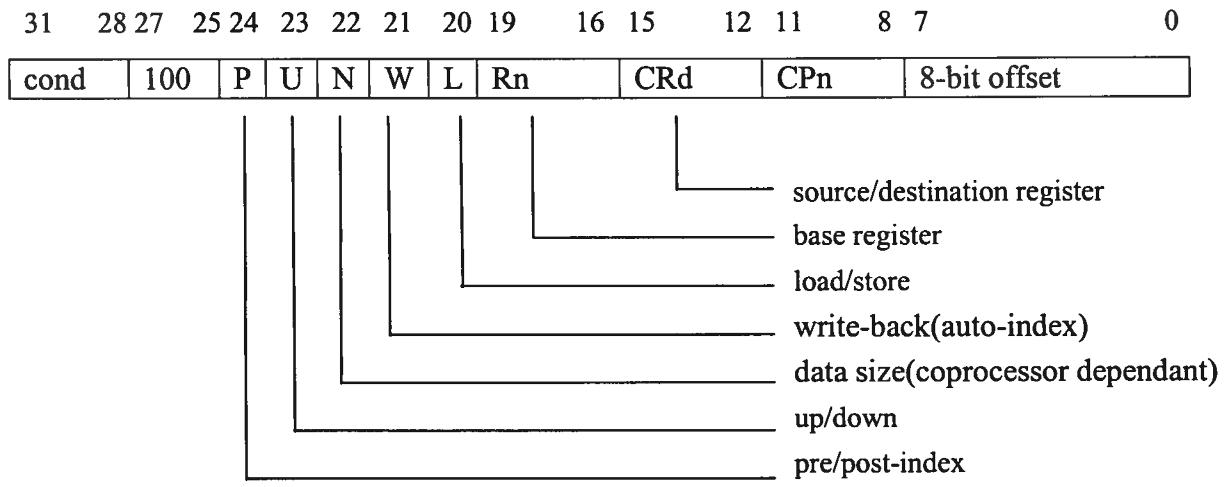


Figure 3.13: Addressing mode 5 Coprocessor data transfer instruction binary encoding [26]

3.5 Organization of the 5-stage ARM Pipeline

ARM architecture describes the processor's instruction set and its interfaces with its closest memory resources. It includes version 3, version 4, version 5 and the latest architecture version 6. ARM micro-architecture is the implementation of its architecture. Table 3-6 compares ARM architectural pipeline depth, it starts with ARM7 with three stages, ARM9 and StrongARM with five stages, and XScale with seven stages, ends with ARM11, which now has an eight-stage pipeline. In this work, we construct an ARM micro-architecture simulator of StrongARM with 5 pipeline stages, which implements ARM instruction set version 5. The ARM processors that use a 5-stage pipeline and separate instruction and data memory are organized as figure 3. 14.

The 5 pipeline stages:

Instruction Fetch: The instruction is fetched from memory and placed in the instruction pipeline.

Instruction Decode: The instruction is decoded and register operands read from the register file. There are three read ports in the register file. All the data processing instructions with register shift, short multiply instructions and long multiply instructions without accumulation instructions have 3 source register operands, there are also other instructions that need 1 or 2 source register operands, so most ARM instructions can read all the source operands in one cycle. Except SMLAL and UMLAL, they need 4 register operands. Adding a 4th port would be bigger for saving only one cycle rarely, so these 2 instructions need 2 cycles in this stage.

Execute: An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU. If the instruction is LDM or STM and is the first cycle of LDM or STM executed in this stage, the start address and end address of the memory block is computed in the ALU, and address incremented in the following cycles.

Buffer/data: Data memory is accessed if required; otherwise the ALU result is simply buffered for one clock cycle to allow the same pipeline flow for all instructions.

Table 3-6: Comparison of ARM architectureal pipeline depth

Pipeline Stages								Micro-architecture Clock (MHz)
1	2	3	4	5	6	7	8	
Fetch	Decode	Execute						(ARM7) 150
Fetch	Decode	ALU	Cache	WB				(strongARM) 233
Fetch	Issue	Decode	Execute	Memory	WB			(ARM10) 266-325
Fetch1	Fetch2	Decode	Shifter	Execute	Exceptn	WB		(XScale) 733 planned 1000
Fetch1	Fetch2	Decode	Issue	Shifter	ALU	SAT	WB	(ARM11) 350-500 estimate>1000

Write-back: The results generated by the instruction are written back to the register file, including any data loaded from memory. For those load instructions and LDM instructions that have auto-index addressing need to change the base register Rn, also write back in this stage, so there are two write ports in the register file.

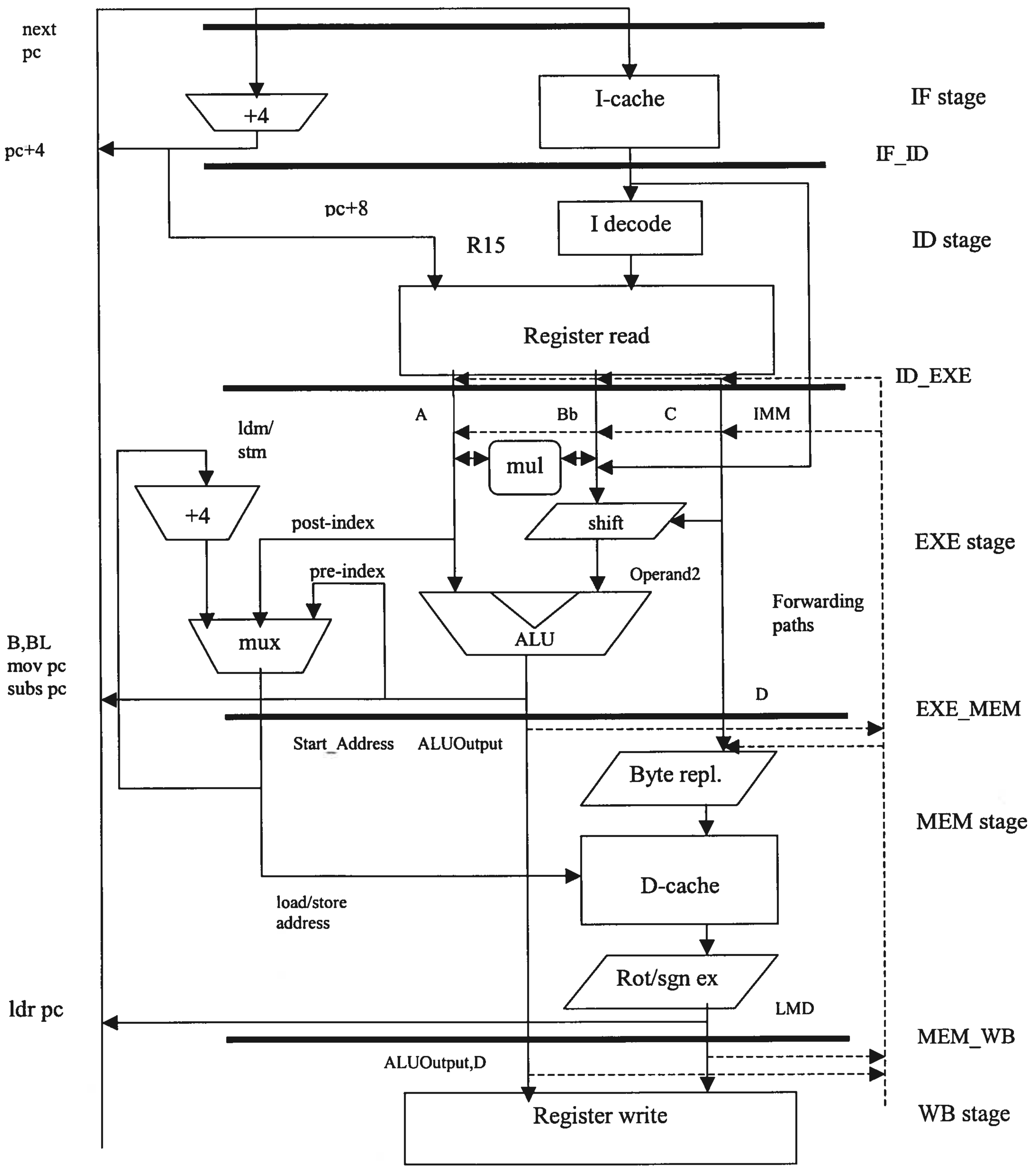


Figure 3.14: ARM 5-stage pipeline organization [25]

Chapter 4 Implementation of the ARM core model

This chapter presents the way we implement the regular pipeline, advanced properties (data forwarding, interlock), branch, CPSR, SPSR and some special instructions.

4.1 All the Instructions operation in different stage (except IF)

Signal description:

Since the pipeline structure is not described in the specification manual we are strongly following the notation and methodology described by Hennessy Patterson [24].

Pipeline Registers (showed in figure 3.14): The pipeline registers are labeled with the names of the stages they connect. For example, IF_ID is the pipeline register between IF and ID stage, the same as ID_EXE, EXE_MEM and MEM_WB. The pipeline registers carry both data and control from one pipeline stage to the next. They hold values temporarily between clock cycles. Any value needed on a later pipeline stage must be placed in such a register and copied from one pipeline register to the next, until it is no longer needed. We note the name of the temporary value 'pipeline register name' + '_' + 'temporary name'. For example IR in IF_ID is IF_ID_IR. All these temporary values are transferred as signal in parallel between pipeline stages, the calculation inside a stage is sequential. These temporary values in different pipeline registers are:

IF_ID:	IR (instruction)
ID_EXE:	IR
	TYPE (type of instruction showed in figure 3.1, 3.2, 3.3)
	OPERATE (showed in instruction column of instruction in table 4.1)
	CPSR (current program status)
	SPSR (saved program status)
	A (Rn/MUL_Rn) (illustrated in figure 3.14)
	Bb (Rm/UMLAL_RdHi/SMLAL_RdHi)
	C (Rs/Rd of store instrucitons/UMLAL_RdLo/SMLAL_RdLo)
	IMM (Imm12/Imm20/Imm24)

EXE_MEM: IR

TYPE

OPERATE

CPSR

SPSR

ALUOutput (output of data processing instructions /UMLAL_RdHi /SMLALL_RdHi /address of load/store)

C (Rd of store instrucitons/Rm of SWP and SWPB)

D (changed base register value)

Start_Address (for LDM and STM)

End_Address (for LDM and STM)

Change_Base (indicate if the base register is to be changed)

MEM_WB: IR

TYPE

OPERATE

CPSR

ALUOutput

LMD (data read from memory)

D

Change_Base

Table 4.1 shows all the instructions operation in different stage (except IF).

Table 4.2 summarizes the work of every stage.

Table 4-1: All the instructions operation in different stage

Instruction type	Instruction	ID	EXE	MEM	WB
Data processing immediate shift		A←Rn Bb←Rm	ALUOutput←A func operand2 (except cmn, cmp, tst, teq), operand2 is showed in figure 3.4 addressing mode 1	ALUOutput←ALUOutput	Rd← ALUOutput

Instruction type	Instruction	ID	EXE	MEM	WB
Data processing register shift		A←Rn Bb←Rm C←Rs	…	…	…
Data processing immediate		A←Rn	…	…	…
Miscellaneous instructions1	MRS	*	ALUOutput←PSR	ALUOutput← ALUOutput	Rd← ALUOutput
	MSR	Bb←Rm	PSR←Bb	*	*
Move immediate to status register	MSR	*	PSR←operand2 (operand2 is showed in figure 3.4 addressing mode 1)	*	*
Miscellaneous instructions2	BX	Bb←Rm	Branch to Bb, change CPSR	*	*
	CLZ	Bb←Rm	ALUOutput← number of '0' bits before the first '1' in Bb	ALUOutput← ALUOutput	Rd← ALUOutput
	BLX2	Bb←Rm	ALUOutput←PC+4, branch to Bb, change CPSR	…	LR← ALUOutput
	BKPT	*	ALUOutput←PC+4, write SPSR, CPSR change mode, I bit, T bit (showed in table 3.2), and branch	…	LR← ALUOutput
Multiplies extra load store	MUL	Bb←Rm C←Rs	ALUOutput←Bb*C More detailed in 4.5.4	…	MUL_Rd← ALUOutput
	MLA	A←MUL_Rn Bb←Rm C←Rs	ALUOutput←A+Bb*C More detailed in 4.5.4	…	MUL_Rd← ALUOutput
	UMULL	Bb←Rm C←Rs	ALUOutput←$(Bb*C)_{63..32}$ D←$(Bb*C)_{31..0}$ More detailed in 4.5.4	ALUOutput← ALUOutput D←D	RdHi← ALUOutput RdLo←D
	SMULL	Bb←Rm C←Rs	ALUOutput←$(Bb*C)_{63..32}$ D←$(Bb*C)_{31..0}$ More detailed in 4.5.4	…	RdHi← ALUOutput RdLo←D
	UMLAL	Bb←Rm(1) C←Rs(1) Bb←RdHi(2) C←RdLo(2)	D←C(2)+$[Bb(1)*C(1)]_{31..0}$ ALUOutput←Bb(2)+$[Bb(1)*C(1)]_{63..32}$ + carry for calculating D More detailed in 4.5.5	…	RdHi← ALUOutput RdLo←D

Instruction type	Instruction	ID	EXE	MEM	WB
	SMLAL	Bb←Rm(1) C←Rs(1) Bb←RdHi(2) C←RdLo(2)	D←C(2)+$[Bb(1)*C(1)]_{31..0}$ ALUOutput←Bb(2)+[Bb(1) $*C(1)]_{63..32}$ + carry for calculating D More detailed in 4.5.5	…	RdHi← ALUOutput RdLo←D
	SWP	A←Rn Bb←Rm	ALUOutput←A C←Bb More detailed in 4.5.3	LMD←mem[ALUOutput] mem[ALUOutput]←C	Rd← LMD
	SWPB	…	…	LMD←$\text{mem[ALUOutput]}_{7..0}$ mem[ALUOutput]←$Rm_{7..0}$	Rd← LMD
	LDRH(R)	A←Rn Bb←Rm	ALUOutput←offset (showed in figure 3.11 addressing mode 3), D←changed base	LMD←$\text{mem[ALUOutput]}_{15..0}$ D←D	Rd←LMD Rn←D
	LDRH(I)	A←Rn	…	…	Rd←LMD Rn←D
	STRH(R)	A←Rn Bb←Rm C←Rd	… C←C	mem[ALUOutput]←$C_{15..0}$ D←D	Rn←D
	STRH(I)	A←Rn C←Rd	… C←C	…	Rn←D
	LDRSB(R)	A←Rn Bb←Rm	…	LMD←$\text{mem[ALUOutput]}_{7..0}$ signed extend; D←D	Rd←LMD Rn←D
	LDRSH(R)	A←Rn Bb←Rm	…	LMD←$\text{mem[ALUOutput]}_{15..0}$ signed extend; D←D	Rd←LMD Rn←D
	LDRSB(I)	A←Rn	…	LMD←$\text{mem[ALUOutput]}_{7..0}$ signed extend; D←D	Rd←LMD Rn←D
	LDRSH(I)	A←Rn	…	LMD←$\text{mem[ALUOutput]}_{15..0}$ signed extend; D←D	Rd←LMD Rn←D
Load store immediate offset	LDR(I)	A←Rn	ALUOutput←offset (showed in figure 3.10 addressing mode 2), D←changed base	LMD←mem[ALUOutput] D←D	Rd←LMD Rn←D
	STR(I)	A←Rn C←Rd	… C←C	mem[ALUOutput]←C; D←D	Rn←D
Load store register offset	LDR(R)	A←Rn Bb←Rm	…	LMD←mem[ALUOutput]; D←D	Rd←LMD Rn←D
	STR(R)	A←Rn Bb←Rm C←Rd	… C←C	mem[ALUOutput]←C; D←D	Rn←D

Instruction type	Instruction	ID	EXE	MEM	WB
Load store multiple	LDM	A←Rn	Calaulate Start_address end_address (showed in figure 3.12 addressing mode 4) D←changed base More detailed in 4.5.1	LMD←mem[Start_address]; D←D	Ri←LMD Rn←D
	STM	…	… More detailed in 4.5.2	mem[Start_address]←Ri D←D	Rn←D
Branch and branch with link	B	*	Branch to PC+(Imm<<2) Imm is the lower 24 bits	*	*
	BL	*	ALUOutput←PC+4 branch to PC+(Imm<<2), Imm is the same as B instruction	ALUOutput← ALUOutput	LR← ALUOutput
Branch and branch with link and change to thumb	BLX1	*	ALUOutput←PC+4 change T bit of CPSR (showed in table 3.2), branch to PC+(Imm<<2)+(H<<1), Imm is the same as B instruction, H is bit 24 of IR	…	LR← ALUOutput
Coprocessor load store and double register transfers	LDC	no	no	no	no
	STC	no	no	no	no
Coprocessor data processing	CDP	no	no	no	no
Coprocessor register transfers	MCR	no	no	no	no
	MRC	no	no	no	no
Software interrupt	SWI		ALUOutput←PC+4 save SPSR change CPSR I_bit T_bit processor mode (showed in table 3.2), branch	ALUOutput←ALUOutput	LR← ALUOutput

Notes:

- *: Nothing to do in that stage.
- …: It does the same work with last instruction operation.
- (1): is the first cycle.
- (2): is the second cycle.

- mem[addr]: is memory data in address ‘addr’

Table 4-2: Work of every stage

Stage	Any instruction		
	Arithmetic instruction	Load/store instruction	Branch instruction

IF	If EXE_changePC then PC←EXE_NPC;
	
	Else if MEM_changePC then PC←MEM_NPC (branch related signal showed in figure 4.5);
	
	Else PC←PC0 (PC0 is used only in IF stage to identify the address of next instruction);
	
	IF_ID_IR←mem[PC];
	PC0←PC+4;
	Reg[15]←PC+8;		

ID	ID_EXE_IR←IF_ID_IR; ID_EXE_TYPE← decode type of instruction (table 3.1 3.2 3.3); ID_EXE_OPERATE← decode operate of instruction (table 3.4 instruction name); ID_EXE_PC←Regs[15]; ID_EXE_A←Regs[Rn]; If (instruction is UMLAL or SMLAL) and (is 2nd cycle of multiply) then ID_EXE_Bb←Regs[RdHi] Else ID_EXE_Bb←Regs[Rm]; If (instruction is UMLAL or SMLAL) and (is 2nd cycle of multiply) then ID_EXE_C←Regs[RdLo] Else if instruction is store then ID_EXE_C←Regs[Rd] Else ID_EXE_C←Regs[Rs]; If instruction is B, BL, BLX, SWI then ID_EXE_Imm←IMM24 (lower 24 bits of IR); Else if instruction is BKPT then ID_EXE_Imm←IMM20 (lower 20 bits of IR); Else ID_EXE_Imm←IMM12 (lower 12 bits of IR).		
Stage	Arithmetic instruction	Load/store instruction	Branch instruction

EXE	EXE_MEM_IR←ID_EXE_IR;
	EXE_MEM_TYPE←
	ID_EXE_TYPE;
	EXE_MEM_OPERATE←
	ID_EXE_OPERATE;
	
	If destination register is PC (Rd=15) then
	EXE_NPC←A func operand2(Bb,C,Imm) (showed in figure 3.4 addressing mode 1);
	EXE_changePC←1;
	
	Else
		EXE_MEM_IR←ID_EXE_IR;
		EXE_MEM_TYPE←
		ID_EXE_TYPE;
		EXE_MEM_OPERATE←
		ID_EXE_OPERATE;
		
		EXE_MEM_ALUOutput←
		address(A,Bb,C,Imm) (showed in figure 3.10, 3.11 addressing mode 2,3);
		
		If change base register Rn then
		EXE_MEM_D←changedbase;
		
		If instruction is store
			EXE_MEM_IR←ID_EXE_IR;
			EXE_MEM_TYPE←
			ID_EXE_TYPE;
			EXE_MEM_OPERATE←
			ID_EXE_OPERATE;
			
			EXE_NPC←
			ID_EXE_PC+Imm;
			
			EXE_changePC←1;
			
			If instruction is BL, BLX, SWI, BKPT then
			EXE_MEM_ALUOutput←
			PC-4;

Stage	Any instruction		
	Arithmetic instruction	Load/store instruction	Branch instruction

	EXE_MEM_ALUOuptut← A func operand2(Bb,C,Imm);	EXE_MEM_C← ID_EXE_C;
		
		If instruction is LDM STM then
		EXE_MEM_Start_Address← Start_address;
		EXE_MEM_End_Address← End_address (showed in figure 3.12) addressing mode 4);	

MEM	MEM_WB_IR← EXE_MEM_IR;
	MEM_WB_TYPE← EXE_MEM_TYPE;
	MEM_WB_OPERATE← EXE_MEM_OPERATE;
	
	MEM_WB_ALUOutput← EXE_MEM_ALUOutput;
		MEM_WB_IR← EXE_MEM_IR;
		MEM_WB_TYPE← EXE_MEM_TYPE;
		MEM_WB_OPERATE← EXE_MEM_OPERATE;
		
		If (instruction is load) and destination register is PC (Rd=15) then
		MEM_NPC← mem[EXE_MEM_ALUOutput];
		
		Else if instruction is load then
		MEM_WB_LMD← mem[EXE_MEM_ALUOutput];
		
		Else if instruction is LDM then
		MEM_WB_LMD← mem[EXE_MEM_Start_Address]
		MEM_WB_Start_Address← EXE_MEM_Start_Address+4;
		
		Else if instruction is store then
		mem[EXE_MEM_ALUOutput]= EXE_MEM_C;
		
		Else if instruction is STM then
		mem[EXE_MEM_Start_Address]= EXE_MEM_C; Start_Address← EXE_MEM_Start_Address+4;
		MEM_WB_End_Address← EXE_MEM_End_Address;
		
		MEM_WB_D← EXE_MEM_D;
		MEM_WB_ChangeBase← EXE_MEM_ChangeBase (notify if the base register is to be changed);
			If instruction is BL, BLX, SWI, BKPT then
			MEM_WB_ALUOutput← EXE_MEM_ALUOutput;

WB	Regs[Rd]← MEM_WB_ALUOutput;	If instruction is LDM then Regs[Ri]← MEM_WB_LMD;	If instruction is BL, BLX, SWI, BKPT then

Stage	Any instruction		
	Arithmetic instruction	Load/store instruction	Branch instruction

	If instruction is long multiply
	Regs[RdHi]← MEM_WB_ALUOutput;
	Regs[RdLo]← MEM_WB_D;
	
	Else if instruction is multiply then
	Regs[Rd_MUL]← MEM_WB_ALUOutput.
		Else Regs[Rd]← MEM_WB_LMD;
		
		If changebase
		Regs[Rn]← MEM_WB_D.
			Regs[LR]← MEM_WB_ALUOutput.

In short, more human readable form, each stage is doing the following:

IF stage:

- Instruction fetch.
- Write PC+4 to PC0 for next instruction address.
- Modify PC to PC+8, in order to be compatible with 3 stages pipeline.
- Pass along values needed in the next stage.

ID stage:

- Decode instruction to know its type and operate.
- Read PC for branch instruction to calculate the new address.
- Read registers (3 register read ports).
- Extend sign of immediate (lower 24 bits or 20 bits or 12 bits of the instruction).
- Pass along values needed in the next stage.

EXE stage:

- Perform an ALU operation for data processing instructions. If the destination register is PC, signal a branch.
- Calculate address for load/store instructions.
- Calculate the start address and end address of memory block for LDM/STM.
- Calculate the new instruction address for branch instructions.
- Pass along values needed in the next stage.

MEM stage:

- For load instruction, read memory data, if the destination register is PC, signal a branch. For store instruction, write data to memory.
- For LDM/STM instruction, modify next access memory address besides memory access.
- Change base register Rn if needed.
- Pass along values needed in the next stage.

WB stage:

- Write destination register for data processing instructions, load instructions.
- Write link register LR for branch and link instructions.
- Write base register Rn for load and store instructions if needed.

4.2 Forwarding

Forwarding [24][25]: There is a data dependency from instruction A to instruction B when a result from A is needed for the execution of B. Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM pipeline requires each of the three source operands (A, Bb, C) to be forwarded from any intermediate result registers (ALUOutput, D, LMD). Figure 4.1 shows the forwarding path (also illustrated in figure 3.14 Forwarding path). Figure 4.2 shows the instruction sequence for forwarding process

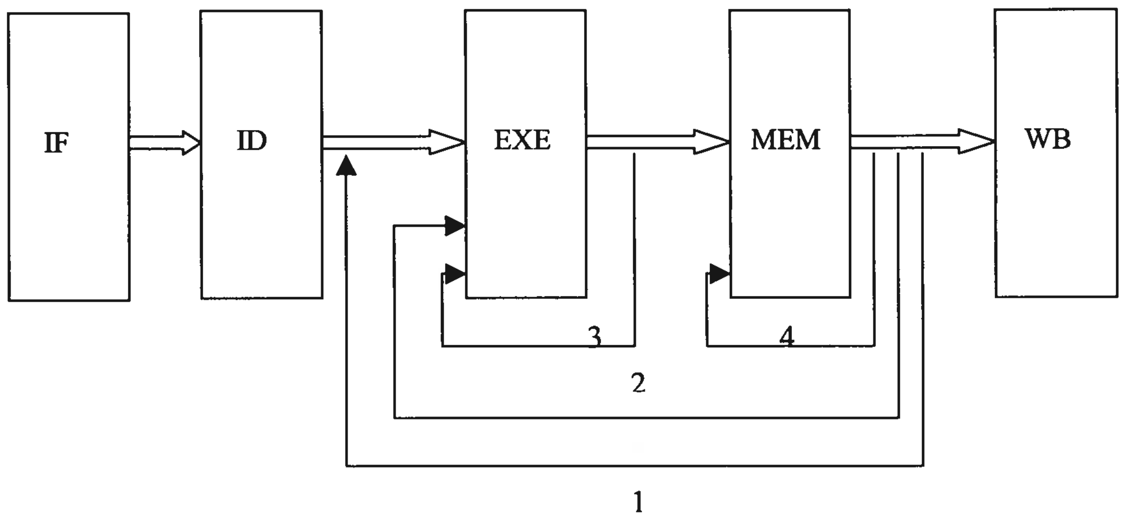
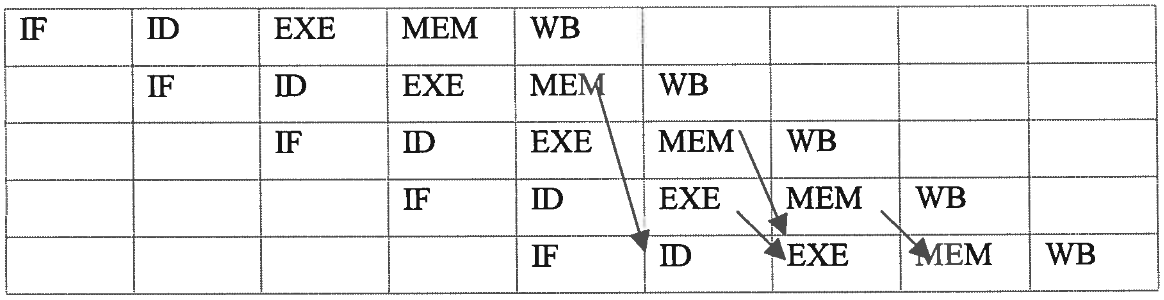


Figure 4.1: Forwarding paths



IF	ID	EXE	MEM	WB				
	IF	ID	EXE	MEM	WB			
		IF	ID	EXE	MEM	WB		
			IF	ID	EXE	MEM	WB	
				IF	ID	EXE	MEM	WB

Figure 4.2: Instruction sequence for forwarding process

Explanation of the forwarding path (Figure 4.1, Figure 4.2 and Figure 3.14):

Path 1: From the end of MEM stage (MEM_WB) to the end of ID stage

Data in MEM is (MEM_WB_) ALUOuptut, D, LMD (path 1 data to be forwarded)

(MEM_WB_) ALUOutput:

- The output of arithmetic instructions, MRS, CLZ, MRC
- The output of multiply instructions (MUL, MLA)
- The higher 32 bits of long multiply output (UMULL, UMLAL, SMULL, SMLAL)

(MEM_WB_) D:

- The lower 32 bits of long multiply output (UMULL, UMLAL, SMULL, SMLAL)
- The value of base register after changing (load, store, LDM, STM, LDC, STC)

(MEM_WB_) LMD: all kinds of load instructions' Load Memory Data

Data in ID is A, Bb, C (path 1 data that forward to)

- A: Rn/MUL_Rn
- Bb: Rm/UMLAL_RdHi/SMLAL_RdHi
- C: Rs/Rd/UMLAL_RdLo/SMLAL_RdLo

Path 2: From the end of MEM stage (MEM_WB) to the start of EXE stage (ID_EXE)

Data in MEM is ALUOutput, D, LMD (path 2 data to be forwarded)

Data in EXE is A, Bb, C (path 2 data that forward to)

Path 3: From the end of EXE stage (EXE_MEM) to the start of EXE stage (ID_EXE)

Data in the end of EXE is ALUOutput, D (path 3 data to be forwarded)

Data in the start of EXE is A, Bb, C (path 3 data that forward to)

Path 4: From the end of MEM stage (MEM_WB) to the start of MEM stage (EXE_MEM)

Data in the end of MEM is LMD (load, SWP,SWPB and LDM) (to be forwarded)

Data in the start of MEM is C (for store instructions and SWP,SWPB,STM) (forward to)

Table 4.3, 4.4, 4.5 shows the forwarding operation.

Table 4-3: Destination register of corresponding pipeline register temporary data ALUOutput, D, LMD (for the source instruction of the forwarding).

Pipeline register temporary data	Stage	Destination register of the instruction
ALUOutput	EXE and MEM	Rd: for arithmetic instructions; MUL_Rd: for short multiply instructions; MULL_RdHi: for long multiply instructions
D	EXE and MEM	RdLo: for long multiply instructions; Rn: for load/store instructions that change the base register;
LMD	MEM	Rd: for load and swp instructions; Ri: for LDM instructions;

Table 4-4: Source register of pipeline register temporary data A, Bb, C (for the destination instruction of the forwarding).

Pipeline register temporary data	Stage	Source register identifier	Source register of the instruction
A	ID and EXE	A_source_ID	Rn: for arithmetic, swp, load/store instructions; MUL_Rn: for short multiply instructions;
Bb	ID and EXE	B_source_ID	Rm: for arithmetic, load/store instructions; RdHi: for long multiply accumulate instructions
C	ID and EXE	C_source_ID	Rs: for arithmetic multiply instructions; Rd: for store instructions; Ri: for STM instructions; RdLo: for long multiply accumulate instructions;
	MEM	C_source_MEM	Rm: for swp instructions; Rd: for store instructions; Ri: for STM instructions;

Table 4-5: Forwarding paths

No.	Forwarding path	Source pipeline (source instruction)	Source instruction operation	Destination pipeline (destination instruction)	Destination instruction operation	Compare condition
1	1	MEM/WB	Arithmetic; multiply; long multiply; (ALUOutput)	IF/ID	Arithmetic; swp; load/store; multiply;	Destination register of source pipeline instruction = source register in destination pipeline instruction (A_source_ID)
2	1	MEM/WB	Long multiply; load/store and change base; (D)	IF/ID	Arithmetic; swp; load/store; multiply;	The same as above

No.	Forwarding path	Source pipeline (source instruction)	Source instruction operation	Destination pipeline (destination instruction)	Destination instruction operation	Compare condition
3	1	MEM/WB	Swp; load/store; LDMs; (LMD)	IF/ID	Arithmetic; swp; load/store; multiply;	The same as above
4	1	MEM/WB	Arithmetic; multiply; long multiply; (ALUOutput)	IF/ID	Arithmetic; load/store; long multiply accumulate;	Destination register of source pipeline instruction = source register in destination pipeline instruction (B_source_ID)
5	1	MEM/WB	Long multiply; load/store and change base; (D)	IF/ID	Arithmetic; load/store; long multiply accumulate;	The same as above
6	1	MEM/WB	Swp; load/store; LDMs; (LMD)	IF/ID	Arithmetic; load/store; long multiply accumulate;	The same as above
7	1	MEM/WB	Arithmetic; multiply; long multiply; (ALUOutput)	IF/ID	Arithmetic; multiply; store; STMs	Destination register of source pipeline instruction = source register in destination pipeline instruction (C_source_ID)
8	1	MEM/WB	Long multiply; load/store and change base; (D)	IF/ID	Arithmetic; multiply; store; STMs	The same as above
9	1	MEM/WB	Swp; load/store; LDMs; (LMD)	IF/ID	Arithmetic; multiply; store; STMs	The same as above
10	2	MEM/WB	Arithmetic; multiply; long multiply; (ALUOutput)	ID/EXE	Arithmetic; swp; load/store; multiply;	Destination register of source pipeline instruction = source register in destination pipeline instruction (A_source_ID)
11	2	MEM/WB	Long multiply; load/store and change base; (D)	ID/EXE	Arithmetic; swp; load/store; multiply;	The same as above
12	2	MEM/WB	Swp; load/store; LDMs; (LMD)	ID/EXE	Arithmetic; swp; load/store; multiply;	The same as above
13	2	MEM/WB	Arithmetic; multiply; long multiply; (ALUOutput)	ID/EXE	Arithmetic; load/store; long multiply accumulate;	Destination register of source pipeline instruction = source register in destination pipeline instruction (B_source_ID)
14	2	MEM/WB	Long multiply; load/store and change base; (D)	ID/EXE	Arithmetic; load/store; long multiply accumulate;	The same as above

No.	Forwarding path	Source pipeline (source instruction)	Source instruction operation	Destination pipeline (destination instruction)	Destination instruction operation	Compare condition
15	2	MEM/WB	Swp; load/store; LDMs; (LMD)	ID/EXE	Arithmetic; load/store; long multiply accumulate;	The same as above
16	2	MEM/WB	Arithmetic; multiply; long multiply; (ALUOutput)	ID/EXE	Arithmetic; multiply; store; STMs	Destination register of source pipeline instruction = source register in destination pipeline instruction (C_source_ID)
17	2	MEM/WB	Long multiply; load/store and change base; (D)	ID/EXE	Arithmetic; multiply; store; STMs	The same as above
18	2	MEM/WB	Swp; load/store; LDMs; (LMD)	ID/EXE	Arithmetic; multiply; store; STMs	The same as above
19	3	EXE/MEM	Arithmetic; multiply; long multiply; (ALUOutput)	ID/EXE	Arithmetic; swp; load/store; multiply;	Destination register of source pipeline instruction = source register in destination pipeline instruction (A_source_ID)
20	3	EXE/MEM	Long multiply; load/store and change base; (D)	ID/EXE	Arithmetic; swp; load/store; multiply;	The same as above
21	3	EXE/MEM	Arithmetic; multiply; long multiply; (ALUOutput)	ID/EXE	Arithmetic; load/store; long multiply accumulate;	Destination register of source pipeline instruction = source register in destination pipeline instruction (B_source_ID)
22	3	EXE/MEM	Long multiply; load/store and change base; (D)	ID/EXE	Arithmetic; load/store; long multiply accumulate;	The same as above
23	3	EXE/MEM	Arithmetic; multiply; long multiply; (ALUOutput)	ID/EXE	Arithmetic; multiply; store; STMs	Destination register of source pipeline instruction = source register in destination pipeline instruction (C_source_ID)

No.	Forwarding path	Source pipeline (source instruction)	Source instruction operation	Destination pipeline (destination instruction)	Destination instruction operation	Compare condition
24	3	EXE/MEM	Long multiply; load/store and change base; (D)	ID/EXE	Arithmetic; multiply; store; STMs	The same as above
25	4	MEM/WB	Swp; load/store; LDMs; (LMD)	EXE/MEM	Swp; store; STMs	Destination register of source pipeline instruction = source register in destination pipeline instruction (C_source_MEM)

4.3 Interlock [24][25]

If the instruction1 is load, SWP, SWPB or the last execution of LDM, and its output is the input of instruction2 (except C of store or STM instruction, and Bb of SWP or SWPB instruction), there is also a RAW hazard between the 2 instructions (Figure 4.3) after forwarding, so it is necessary to stall the pipeline until the hazard is cleared. (Figure 4.4)

Instruction1	IF	ID	EXE	MEM	WB			
Instruction2		IF	ID	EXE	MEM	WB		
Instruction3			IF	ID	EXE	MEM	WB	
Instruction4				IF	ID	EXE	MEM	WB

Figure 4.3: RAW hazard between two adjacent instructions

Instruction1	IF	ID	EXE	MEM	WB				
Instruction2		IF	ID	stall	EXE	MEM	WB		
Instruction3			IF	stall	ID	EXE	MEM	WB	
Instruction4				stall	IF	ID	EXE	MEM	WB

Figure 4.4: Insert a nop to avoid RAW hazard

4.4 Branch instructions [25][26]

Instructions that change PC are:

EXE stage:

- Arithmetic instructions (except TST,TEQ,CMN,CMP) and destination register (Rd) is PC (Rd=15).
- BX, BLX2, BKPT, B, BL, MRC (destination register is PC, Rd=15), SWI

MEM stage:

- LDR, LDRT (and destination register is PC, Rd=15), LDM last execution ($IR_{15..0}$=0x8000)

The processing of branch instructions is illustrated in figure 4.5. When a branch instruction is encountered in EXE stage, it gives a signal EXE_changePC to IF and ID stage to cancel the instructions in those stages, also gives the new PC value EXE_NPC to IF stage so it can fetch a new instruction in EXE_NPC address in the new cycle. The same as MEM stage, it cancels IF, ID, EXE stage instructions by signal MEM_changePC and gives the new instruction address by MEM_NPC.

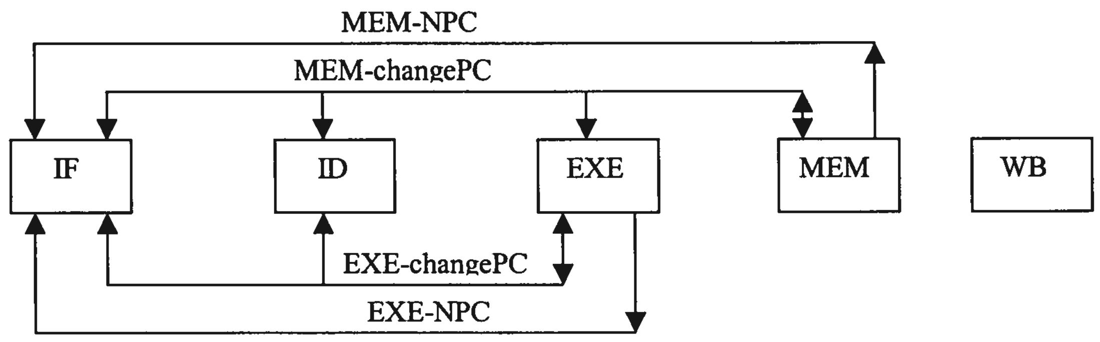


Figure 4.5: Branch instruction operation

Notes:

1. There is no delay slot.
2. For the instructions that change PC in EXE stage, it should cancel the instruction in ID, EXE and give new PC in IF at next cycle.
3. For the instructions that change PC in MEM stage, it should cancel the instruction in ID, EXE (before execution), MEM and give new PC in IF at next cycle.

4.5 Some special instructions' implementation

Special instructions include load and store multiple, swap the content of register and memory and all multiplications (LDM, STM, SWP, SWPB, MUL, MLA, UMULL, SMULL, UMLAL, SMLAL).

4.5.1 Load Multiple Registers (LDM)

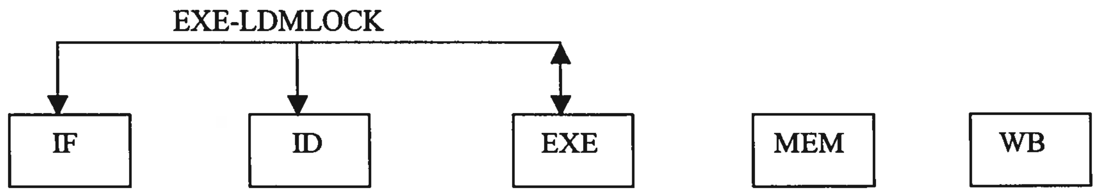


Figure 4.6: LDM instruction operation

Figure 4.6 illustrates the processing of instruction LDM [25][26]. When a LDM instruction is in EXE stage, it gives a signal EXE_LDMLOCK to IF and ID stages and lock the two stages, continue the last 3 stages EXE, MEM and WB until the last execution of LDM, it changes the value of EXE_LDMLOCK to unlock IF and ID stages.

General format:

LDM{<cond>} <addressing_mode> <Rn>, <registers>

The registers' subset can't be empty, otherwise it is an invalid instruction.

There are 3 special forms of LDM.

LDM1, this form of the LDM instruction is useful for block loads, stack operations and procedure exit sequences. It loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. The general-purpose registers loaded can include the PC. If they go, the word loaded for the PC is treated as an address and a branch occurs to that address but do not change the CPSR.

LDM2, it loads user mode registers when the processor is in privileged mode, the instruction loads a non-empty subset of the user mode general-purpose registers from sequential memory locations (PC can't be loaded).

LDM3, this form is useful for returning from an exception. It loads a subset of the general-purpose registers and the PC from sequential memory locations. Also, the SPSR of the current mode is copied to the CPSR.

LDM loads multiple register value from memory. It takes 1 cycle per memory access, so the number of cycles it needs should be the number of registers. Rn is the base register, it can be modified according to the addressing-mode, registers are indicated in least significant 16 bits of the instruction, every bit i is '1' means the corresponding register Ri needs load data from memory. The sequence of loading data is from lower register number to higher register number.

General work: LDM instruction locks the IF, ID stage and continues EXE, MEM, WB stage.

Execution: LDM first execution should calculate the start address and end address of memory to be accessed, at the same time, change base register value according to the instruction format in EXE stage. Access memory in MEM stage, write register and change base register in WB stage. The following execution of LDM only change memory address in EXE, access memory in MEM, write register in WB.

Forwarding: For the first execution of LDM, the first loaded register value and base register value can be forwarded if necessary. For the later executions of LDM, only the loaded register value can be forwarded.

Signal: EXE_LDMLOCK is used to indicate if a LDM instruction is being executed. It is initialized to 0, when LDM instruction is encountered, it is set to 1, until the last execution (last register to be loaded), set it to 0.

Instruction select: When EXE_LDMLOCK is 1, the instruction executed in EXE stage next cycle will be the instruction from MEM stage, and clear 1 bit (the least significant '1' bit). When EXE_LDMLOCK is 1, IF and ID stage will be locked, keep the instruction in that stage. When EXE_LDMLOCK is 0, IF and ID stage continue to execute.

4.5.2 Store Multiple Registers (STM)

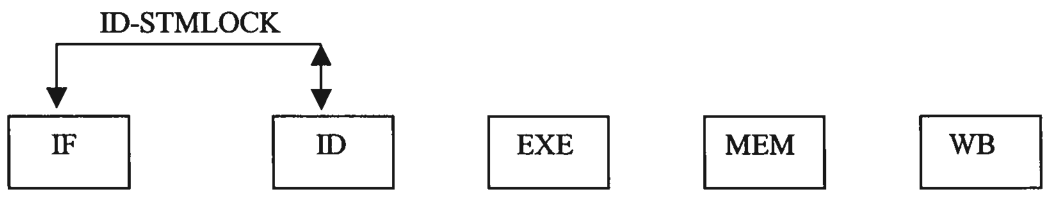


Figure 4.7: STM instruction operation

Figure 4.7 illustrates the processing of STM [25][26] instruction. When a STM instruction is in ID stage, it gives a signal ID_STMLOCK to IF stage and lock it, continue the last 4 stages ID, EXE, MEM and WB until the last execution of STM, it changes the value of ID_STMLOCK to unlock IF stage.
General format:

STM {<cond>} <addressing_mode> <Rn>, <registers>

The registers subset can't be empty, otherwise it is an invalid instruction.

There are 2 special forms of STM.
STM1, this form of the STM instruction stores a non-empty subset of the general-purpose registers to sequential memory locations.

STM2, this form of STM stores a subset of the user mode general-purpose registers to sequential memory locations when the processor is in privileged mode.

STM stores multiple register value to memory. It takes 1 cycle per memory access, so the number of cycles it needs should be the number of registers. Rn is the base register, it can be modified according to the addressing-mode, registers are indicated in least significant 16 bits of the instruction, every bit i is '1' means the corresponding register Ri value needs to be stored in memory. The sequence of storing data is from lower register number to higher register number.

General work: STM instruction lock the IF stage and continue ID, EXE, MEM, WB stage.

Execution: STM first execution should read register value to be stored and base register value in ID, calculate the start address and end address of memory to be accessed, at same time, change base register value according to the instruction format in EXE stage, Access memory in MEM stage, write base register in WB stage. The following execution of STM read register value to be stored in ID stage, change memory address in EXE stage, access memory in MEM stage, nothing to do in WB stage.

Forwarding: Only the first execution of STM, base register value (D) can be forwarded if necessary.

Signal: ID_STMLOCK value can be:

-1: Initial value

3: First execution and need lock

2: First execution and no need lock

1: Not first execution and need lock

0: Not first execution and no need lock

Instruction select: When ID_STMLOCK is 1 or 3, the instruction executed in ID next cycle will be the same instruction, and clear 1 bit (the least significant '1' bit). When ID_STMLOCK is 1 or 3, IF stage will be locked, keep the instruction in that stage. When ID_STMLOCK is 0, 2 or -1, IF stage continues to execute (unlock IF).

4.5.3 Swap a word/byte (SWP, SWPB)

Figure 4.8 illustrates the processing of SWP, SWPB [25][26] instruction. When a SWP instruction is in MEM stage, it gives a signal MEM_SWPLOCK to IF, ID and EXE stages and lock the three stages, continue the last 2 stages MEM for 1 cycle, then it changes the value of MEM_SWPLOCK to unlock IF, ID and EXE stages.

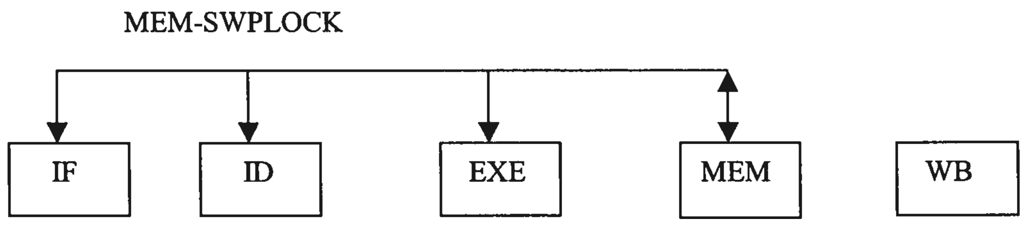


Figure 4.8: SWP instruction operation

General format:

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

It loads a memory [Rn] data to register Rd and store another register Rm's content to memory [Rn] atomically, so it needs 2 cycles for memory access.

General work: SWP instruction locks the IF, ID, EXE stage for 1 cycle and continue MEM, WB stage.

Execution: SWP first execution should read memory in MEM stage, write register in WB stage. The following execution of SWP, SWPB should write register to memory in MEM stage, nothing to do in WB stage.

Forwarding: For the first execution, data loaded from memory (for Rd) can be forwarded if necessary.

Signal: MEM_SWPLOCK is initialized to 0. When SWP or SWPB instruction is encountered, it is set to 1, after 1 cycle it is set to 0.

Instruction select: When MEM_SWPLOCK is 1, the instruction executed in MEM stage will be the same instruction. When MEM_SWPLOCK is 1, IF, ID and EXE stage will be locked, keep the instruction in that stage. When MEM_SWPLOCK is 0, IF, ID and EXE stage continue to execute.

4.5.4 Multiply instructions (MUL, MLA, UMULL, SMULL)

Figure 4.9 illustrates the processing of these instructions. When a multiplication instruction (MUL, MLA, UMULL, SMULL) is in EXE stage, it gives a signal EXE_MULLOCK to IF and ID stages and lock the two stages, continue the last 3 stages EXE, MEM and WB until the last cycle of the multiply instruction, it changes the value of EXE_MULLOCK to unlock IF and ID stages.

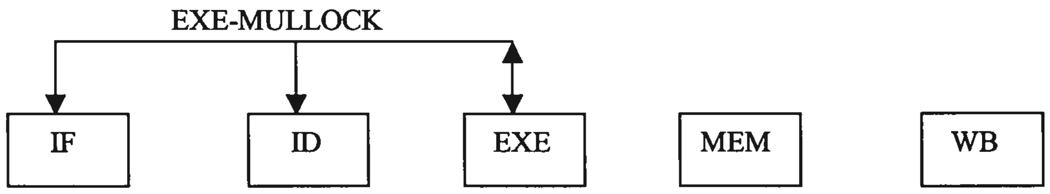


Figure 4.9: Multiply-1 instruction operation

Instructions are:

MUL: multiply instruction

MLA: accumulated multiply instruction

UMULL: unsigned long multiply instruction

SMULL: signed long multiply instruction

All multiply instructions was integrated in the integer unit, they take much more cycles in EXE stage, the number of cycles that the instruction needs depend on the source operands. The greater operand needs more cycles. The calculation method can also be changed according to the real hardware.

General work: The multiply instructions lock the IF, ID, multiple cycles and continue EXE, MEM, WB stage.

Execution: Until the last cycle, EXE output its result to MEM, and then WB to write back the register.

Signal: EXE_MULLOCK is initialized to 0. When MUL, MLA, UMULL or SMULL instruction is encountered, it is set to the number of cycles that the multiply needs, then it is decreased by 1 every cycle, until 0, it unlocks IF and ID.

Instruction select: When EXE_MULLOCK is not 0, the instruction executed in EXE stage will be the same instruction. When EXE_MULLOCK is not 0, IF and ID will be locked, keep the instruction in that stage. When EXE_MULLOCK is 0, IF and ID stage continue to execute (unlock IF and ID).

4.5.5 Long multiply and accumulate instructions (UMLAL, SMLAL)

Figure 4.10 illustrates the processing of UMLAL and SMLAL [25][26] instructions. When a UMLAL or SMLAL instruction is in ID stage, it gives a signal ID_LMULLOCK to IF stage and lock the stage IF, so it can read the source registers in 2 cycles (it needs 2 cycle to read the 4 source registers for UMLAL or SMLAL), if the multiply instruction needs only 1 cycle to execute in EXE stage, it unlock the IF stage, otherwise it lock IF and ID stages until the last cycle.

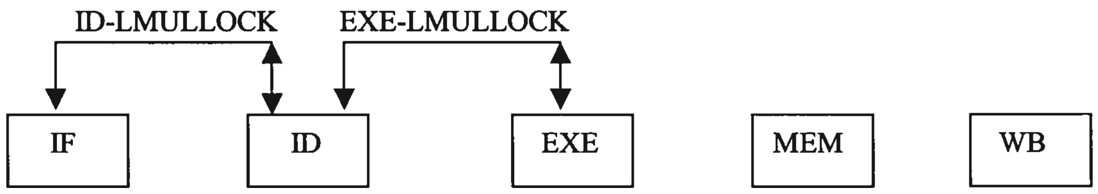


Figure 4.10: Multiply-2 instruction operation

Instructions are: UMLAL and SMLAL.

UMLAL: unsigned accumulated long multiply

SMLAL: signed accumulated long multiply.

General format:

UMLAL(SMLAL){<cond>}{S} <RdLo>, <RdHi>, <Rm>, <Rs>

RdLo $\leftarrow$RdLo+$[Rm*Rs]_{31..0}$
RdHi $\leftarrow$RdHi+$[Rm*Rs]_{63..32}$ + carry from calculating RdLo
S indicates that if the instruction changes the CPSR.

The number of cycles in EXE they need is the same as other multiply instructions. Both of the instructions need four source register operands, but this kind of ARM micro-structure has only 3 register read ports. It needs 2 cycles to read the source operand in ID stage.

It needs 2 signal ID_LMULLOCK and EXE_LMULLOCK to implement the instructions. ID_LMULLOCK is used to lock IF when UMLAL or SMLAL is encountered. EXE_LMULLOCK is used to indicate the number of cycles it needs in EXE, and to release ID_LMULLOCK 1 cycle before it finish in EXE stage, so that the pipeline can continue (2 instructions in IF and ID).

General work: The multiply instructions lock the IF, ID, multiple cycles and continue EXE, MEM, WB stage.

Execution: Until the last cycle, EXE output its result to MEM, and then WB to write back the register.

Signal: ID_LMULLOCK and EXE_LMULLOCK are initialized to 0. When UMLAL or SMLAL instruction is encountered in ID, ID_LMULLOCK is set to 2, ID read 2 register operands Rm and Rs, and lock the IF stage. Then set to 1 or 0 according the number of cycles it needs in EXE. If it needs more cycles in EXE, it is set to 1 to lock IF, at the same time, ID keeps the other 2 operands RdHi and RdLo. 1 cycle before finishing execution in EXE it is set to 0, and unlock the IF stage so that the pipeline can continue.

Instruction select: When ID_LMULLOCK is not 0, the instruction executed in ID will be the same instruction, IF will be locked, keep the instruction in that stage. When ID_LMULLOCK is 0, IF continue to execute. When EXE_LMULLOCK is not 0, the instruction executed in EXE will be the same instruction.

4.6 Current and Saved Program Status Register (CPSR, SPSR)

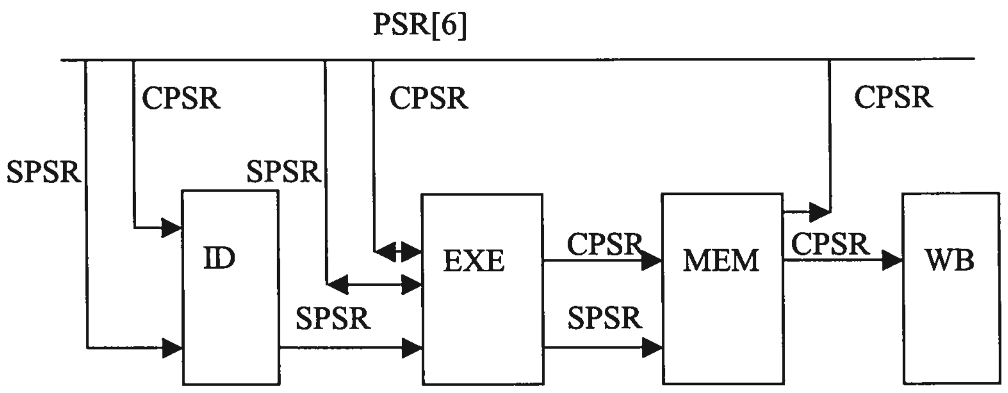


Figure 4.11: PSRs (progam status register) operation

Figure 4.11 illustrates the processing of CPSR and SPSR [25][26]: ID stage read CPSR and SPSR, they are together with this instruction. In EXE stage, the instruction can read or write CPSR and SPSR. If CPSR changes in EXE stage, we need to modify the CPSR for following instructions, so when instruction arrives to EXE stage, it read the newest CPSR. MEM stage can modify CPSR. WB stage needs CPSR for writing back the appropriate register.

CPSR:

The stages that need CPSR are:

- ID stage (for read register in different processor mode).
- EXE stage (for conditioned execution instructions, MRS, BKPT, SWI).
- WB stage (for write register in different processor mode).

The stages that modify CPSR are:

- EXE stage
- MEM stage

In EXE stage,

- The arithmatic instructions, when bit 20 of IR is 1, it changes N,Z,C,V in CPSR, when destination register is PC (Rd=15), CPSR=SPSR.
- MSR instruction changes CPSR or SPSR with register operand or immediate operand.
- BX, BLX2 modify T flag.
- BKPT changes processor mode to abort mode, and set SPSR_abt=CPSR.
- BLX1 sets T flag to 1.
- MRC changes N Z C V when destination register is PC (Rd=15).
- SWI changes processor mode to supervisor mode, and set SPSR_svc=CPSR.

In MEM stage,

- LDR and LDRT, when destination register is PC (Rd=15), it changes T flag.
- LDM1, when IR bit 15 is '1', it changes T flag.
- LDM3, it sets CPSR with SPSR of corresponding processor mode.

SPSR:

The stages read SPSR are:

- EXE stage
- MEM stage

In EXE stage,

- The arithmatic instructions, when destination register is PC (Rd=15), CPSR=SPSR.
- MRS, when the PSR is SPSR, Rd<--SPSR.

In MEM stage,

- LDM3, it sets the CPSR with corresponding SPSR.

The stage writes SPSR is EXE.

Instructions write SPSR are:

- MSR, when the PSR is SPSR, it writes the SPSR with register operand or immediate operand.
- BKPT, it writes SPSR_abt with CPSR.
- SWI, it writes SPSR_svc with CPSR.

MSR changes processor mode in EXE stage, so it is necessary to forward MSR execution to ID stage, at the same time, for MSR with register operand, B operand needs to be forwarded to ID stage.

Implementation:

- Forwarding: Source register operands need to be read in ID stage, it needs to know the processor mode. Only MSR instruction can change processor mode and do not result in branch operation, so it is necessary to forward MSR execution to ID stage, at the same time, for MSR instruction with register operand Bb, Bb also needs to be forwarded to ID stage.
- For reading CPSR/SPSR, read it in ID stage, the instruction in ID stage needs processor mode in CPSR to read register file. In EXE stage, read CPSR/SPSR again, since some instructions in EXE stage may change condition code of CPSR/SPSR and do not change processor mode. Transfer the newest CPSR/SPSR to MEM stage and WB stage.
- For writing SPSR, in EXE sage, BKPT and SWI result in branch operation, this must cancel the instructions in IF stage and ID stage. Or in ID stage, MSR instruction can modify the SPSR, so there are no multiple copies.
- For writing CPSR, in MEM stage, LDR and LDM must result in branch operation, this will cancel the instructions in IF stage, ID stage and EXE stage. In EXE stage instructions that change CPSR are BX, BLX2, BKPT, BLX1, MRC, SWI, these

instructions also result in branch operation, it will cancel the instructions in IF stage and ID stage. In ID stage, only MSR instruction can change CPSR, so there are no multiple copies.

4.7 Discussion on generalization

This section discusses the generalizaiton of the ARM core model from the following aspects:

- It is compatible with 3-stage and 5-stage pipeline architecture.
- It can be extended to support Thumb instruction set.
- It is easy to add some new instructions.
- For those processors that change the number of pipeline stages, it is difficult to implement with a little change in this processor model.
- By encapsulating the ARM core model, separating its interface and implement, it can be a component to plug into a system and communicate with other components.

The ARM core model is based on a general ARM 5-stage pipeline micro-architecture with forwarding paths, automatic nop inserting when interlocking. It is compatible with the 3-stage pipeline micro processor core, it also supports 5-stage pipeline processor core architecture without automatic nop inserting.

The ARM core model supports 32-bits ARM instruction sets. It can be generalized to support thumb instruction sets (16-bits).

- The thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus and to allow better code density than ARM. Every thumb instruction is encoded in 16 bits.
- Thumb does not alter the underlying programer's model of the ARM architecture. All thumb data-processing instructions operate on full 32-bit values, and full 32-bit addresses are produced by both data-access instructions and instruction fetches.

- When the processor is executing thumb instructions, registers R0-R7 are available. Some instructions can access PC (Program Counter), LR (Link Register), SP (Stack Pointer). Further instructions allow limited access to R8-R15.
- Thumb does not provide direct access to the CPSR or any SPSR.

After adding a module for decoding Thumb instructions and access limit, the ARM core model can be generalized to support Thumb instruction sets.

For supporting new ARM instructions, we only need to modify ID (instruction decode) module and EXE (execution) module without changing the other parts (forwarding, interlock, IF, MEM, WB), the whole pipeline can work well. For those processor core that change the pipeline stages (more than 5 stages), we have to adjust work of every stage, forwarding path, interlock condition, etc. So it is difficult to support them with little change to the ARM core model.

The ARM core model is now a stand alone software project. We can encapsulate it to be a component to plug into a system and communicate with others by separating its interface (input/output request) and implementation with little source code modification.

Chapter 5 Validation of the model

In order to test the validity of the ARM core model, some experiments were carried out on this model. And also they were run on another ARM Instruction Set Simulator, and then compare the results from the two models, to ensure if the ARM core model is valid.

5.1 Methodology of Validation

This section introduces the ARM core model, Instruction Set Simulator and methodology of validation.

5.1.1 The ARM core model

The ARM core model is a cycle accurate micro-architecture simulator. It simulates the 5-stage pipeline, IF (Instruction Fetch), ID (Instruction Decode), EXE (Execution), MEM (Memory access), WB (Write back), separate instruction cache and data cache (not the complete memory hierarchy subsystem), registers, forwarding path, interlock logic, hazard detect. In EXE stage, there are ALU, shifter, multiplier, auto-indexed addressing, etc. showed in figure 3-14.

The ARM core model is a simulator of the 32-bit ARM RISC processor. It simulates the entire instruction set except for those requiring use of the coprocessor unit. The coprocessor is not a functional unit standard across all ARM processors. Coprocessor instructions vary widely from system to system depending on the actual coprocessor module available on the chip. It was not possible to come up with a common subset of operations that all coprocessors would support.

The ARM core model executes one instruction at a time and updates the processor state accordingly. Figure 5.1 illustrates the processor of the ARM as simulated by the ARM core model.

1. The input program is ARM object code. It can be obtained by one of the following modes:

- ARM object code.
- ARM assembler to assemble the ARM assembly program.
- ARM GCC cross compiler to compile C source file to ARM object code.

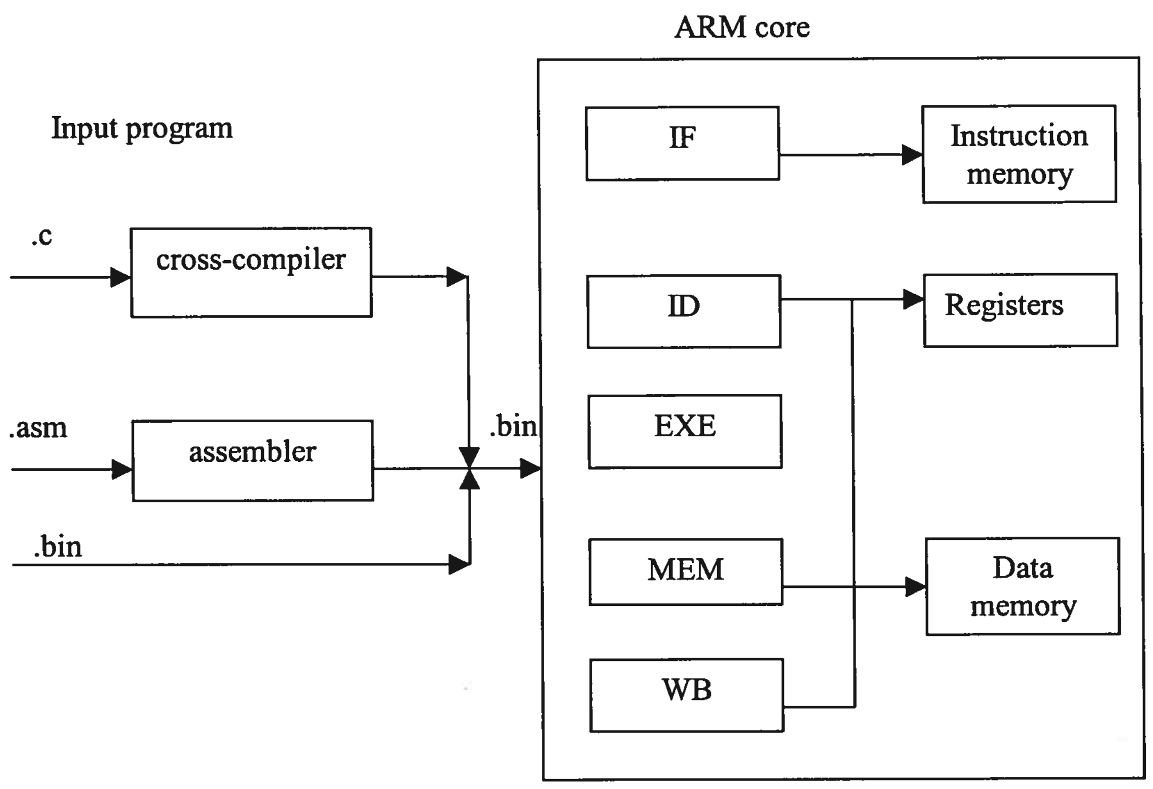


Figure 5.1: The ARM core model

2. The object code was stored in the instruction cache. The ARM core model does not simulate the ARM processor in the 16-bit THUMB mode [25][26]. This has the effect that all instructions are stored at word-aligned addresses, and all instructions fetch operation as 32-bit data transfers.
3. Data was stored in data cache in the little-endian representation. ARM supports data transfer in three different sizes: byte, halfword, and word between the registers and memory. The ARM core model implements the transfer of memory data of all the sizes.

4. The program behavior must be repeating for testing purposes. Being completely software-simulated, ARM core model guarantees that the outcome of program behavior is deterministic. The ARM core model simulates program execution by iterating through a cycle of instruction fetching, decoding, execution, memory access and write-back.

5.1.2 Validate the ARM core model with an ISS (Instruction Set Simulator)

An ISS is used to validate the ARM Core Model. The ARM core model and ISS do not have the same precision. The ARM core model is a cycle accurate simulator that simulates the micro-architecture with 5 pipeline stages. ARM ISS executes ARM programs by simulating the effects of each instruction on a target machine. It interprets ARM programs at the instruction level. So we use an ARM ISS to validate the ARM core model at the instruction level. The validation methodology is show in figure 5.2:

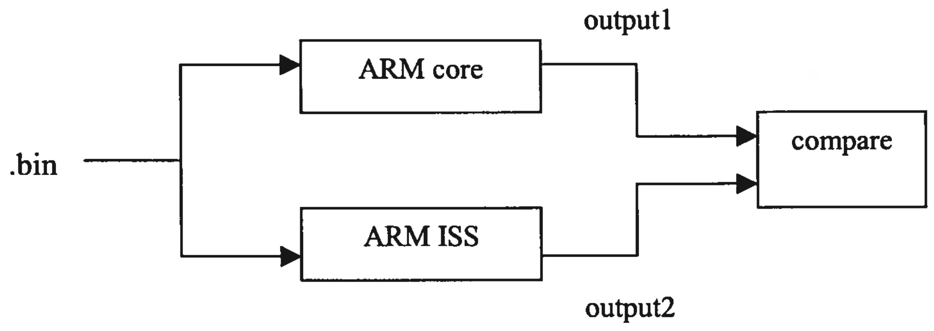


Figure 5.2: Validate the ARM core model

1. The C source program (example code in figure 5.4) is cross compiled by:

   % **<u>arm-elf-gcc ◊ –static ◊ source.c ◊ –o ◊ objfile1</u>** (The objfile1 can be run on the ARM ISS)

2. Modify the C source program to remove ‘main’ and ‘printf’ (example code in figure 5.3). Cross compile it by:

% **arm-elf-gcc ◊ –static ◊ –nostartfiles ◊ –nostdlib ◊ source.c ◊ –o ◊ objfile2**

The ARM binary code extracted from objfile2 can be executed on the ARM core model. The start file is operating system dependent, we don't have an operating system in the ARM core model. The ARM core model does not support start file and library, so we use the two options: nostartfiles and nostdlib.

3. Run the object code on the appropriate model, and compare the outputs from the two models.

For example the code of test_nolib_fib:

```
Int fib(int i);
Extern "C" int _start()
{
  int a=15;
  return fib(a);
}
int fib(int i)
{
  if ((i==1)||(i==2)) return 1;
  else return fib(i-2)+fib(i-1);
}
```

Figure 5.3: A program executed on the ARM core model (without library)

```
#include <stdio.h>
#include <stdlib.h>
int fib(int i);
void main()
{
  int a=15;
  int b;
  b=fib(a);
  printf("%d",b);
}
int fib(int i)
{
  if ((i==1)||(i==2)) return 1;
  else return fib(i-2)+fib(i-1);
}
```

Figure 5.4: A program executed on ISS (with libraries)

```
nolib_fib1:     file format elf32-littlearm

Disassembly of section .text:

00008000 <_start>:
8000:   e1a0c00d        mov     r12, sp
8004:   e92dd800        stmdb   sp!, {r11, r12, lr, pc}
8008:   e24cb004        sub     r11, r12, #4        ; 0x4
800c:   e24dd004        sub     sp, sp, #4          ; 0x4
8010:   e3a0300f        mov     r3, #15  ; 0xf
8014:   e50b3010        str     r3, [r11, -#16]
8018:   e51b0010        ldr     r0, [r11, -#16]
801c:   eb000005        bl      8038 <fib__Fi>
8020:   e1a03000        mov     r3, r0
8024:   e1a00003        mov     r0, r3
8028:   ea000001        b       8034 <_start+0x34>
802c:   ea000000        b       8034 <_start+0x34>
8030:   eaffffff b      8034 <_start+0x34>
8034:   e91ba800        ldmdb   r11, {r11, sp, pc}

00008038 <fib__Fi>:
8038:   e1a0c00d        mov     r12, sp
803c:   e92dd810        stmdb   sp!, {r4, r11, r12, lr, pc}
8040:   e24cb004        sub     r11, r12, #4        ; 0x4
8044:   e24dd004        sub     sp, sp, #4          ; 0x4
8048:   e50b0014        str     r0, [r11, -#20]
804c:   e51b3014        ldr     r3, [r11, -#20]
8050:   e3530001        cmp     r3, #1   ; 0x1
8054:   0a000003        beq     8068 <fib__Fi+0x30>
8058:   e51b3014        ldr     r3, [r11, -#20]
805c:   e3530002        cmp     r3, #2   ; 0x2
8060:   0a000000        beq     8068 <fib__Fi+0x30>
8064:   ea000002        b       8074 <fib__Fi+0x3c>
8068:   e3a00001        mov     r0, #1   ; 0x1
806c:   ea00000f        b       80b0 <fib__Fi+0x78>
8070:   ea00000c        b       80a8 <fib__Fi+0x70>
8074:   e51b2014        ldr     r2, [r11, -#20]
8078:   e2423002        sub     r3, r2, #2          ; 0x2
807c:   e1a00003        mov     r0, r3
8080:   ebffffec        bl      8038 <fib__Fi>
8084:   e1a04000        mov     r4, r0
8088:   e51b2014        ldr     r2, [r11, -#20]
808c:   e2423001        sub     r3, r2, #1          ; 0x1
8090:   e1a00003        mov     r0, r3
8094:   ebffffe7        bl      8038 <fib__Fi>
......
```

Figure 5.5: AN ARM assembly program after cross compile (Fibonacci)

5.2 Experiments

In order to validate the ARM core model, we do some experiments. They can be classified into two groups:

- Basic test
- Combination test

Basic test is to test some special instructions' execution, interlock, forwarding, program status register (PSR) related operations. It includes the following programs in table 5.1.

Table 5-1: Programs for basic test

Program name	Usage
test_LDM	test load multiple register (LDM) instruction
test_STM	test store multiple register (STM) instruction
test_SWP	test SWP instruction: to swap memory and register content
test_MUL	test short multiply instruction
test_UMULL	test long multiply instruction
test_UMLAL	test long multiply and accumulate instruction
test_MSR	test move general purpose register to program status register (MSR)instruction
test_LDRLOCK	test interlock between load register instruction and other instructions
test_LDMLOCK	test interlock between load multiple register (LDM) instruction and other instructions
test_FORWARD	test pipeline forwarding operation

These test programs were directly on the ARM core model. They were written in ARM object code. It is not necessary to compare the two results from the two models. By analyzing the results, to know if the ARM core model is correct in these operations. Implicitly test different function unit inside the ARM core. For example shift operation, ALU, auto-indexed addressing etc.

Combination test is to test different instructions combination. It includes the following programs in table 5.2:

Table 5-2: Programs for combination test

Program name	Description
Test_sort	It is to sort elements in increased order, it is used to test different instruction combination and branch. It was written in ARM object code
Test_fibonacci1	Calculate the first ten fibonacci numbers. There is no function call. It is used to test different instruction combination. It was written in ARM object code
Test_nolib	Calculate the sum of an array. It was written in C
Test_nolib_fib	Calculate the first ten fibonacci numbers. There is recursively function call. it is used to test stack processing when function call and return. It was written in C

For the first 2 programs written in ARM object code, can be directly run on the ARM core model. Then analyze the results to know if it is correct.

For the last 2 programs written in C, they should be cross compiled to ARM object code with and without library support by using arm-elf-gcc, then run on ARM ISS and the ARM core model, by comparing the two results form the two model, to ensure the validity of the ARM core model being validated.

5.3 Summary of this validation

From basic test, the following conclusions can be made:

- The functional units of this ARM core model work correctly corresponding to these basic test cases. The functional units are ALU, shifter, multiplier, auto-indexed addressing unit, conditional execution etc.
- CPSR and SPSR operate correctly on the basic test cases.
- The forwarding paths are correct corresponding to the basic test cases. The paths described in 4.2.

- The interlock logic is valid corresponding to these basic test cases. The logic is data-hazard between register load instructions and other instructions. A nop was inserted automatically.

From combination test, the ARM core model seems to be valid. It can run the programs output from arm-elf-gcc cross compiler giving the same results on the test cases as an ARM ISS.

Chapter 6 Performance Evaluation

The best simulation method depends on the application of the simulation results. Architecture level simulator is used to research the internal data-paths of the processor, is not intended for executing target system binaries on an alternate platform. Direct execution and threaded code simulation technique makes the simulation faster. It is used in increasing simulation speed. Instruction Set Simulator is used to run system binaries program that executed on the target architecture, and gather some statistics information, test concepts and processor design tradeoffs. Flexibility is important and speed is not of primary importance.

Cycle accurate processor simulators that simulate the micro-architecture of processors are essential and commonly used for research and design of processors. The ARM core model is a cycle accurate micro-architecture simulator, it simulates the implementation of the 5-stages ARM pipeline, on each cycle, the pipeline model is advanced (subject to stalls and interlocks), and at any given point, several instructions may be in various stages of execution. The simulator can be used to:

1. Run target system binaries code, simulate the overall behavior of execution of programs that are intended for execution on an ARM system.
2. Evaluate the performance of target processor by counting the number of cycles of specific instructions.
3. Gather some statistics information.
4. Compare the quality of compiled code as produced by different compilers.

For evaluating the performance of the simulator, the most important quality metric is its execution time of a workload [14][15]. The execution time [24] is especially relevant for the development of high performance systems, where being able to perform simulation in real time is desired. However, the execution time varies greatly depending on the application and what features are enabled. The slowdown [17] is presented as the ratio of time to complete the workload execution on the simulator to the execution time on the target architecture.

By counting the number of cycles of specific instructions, the simulator can be used to evaluate the performance of ARM processor and compare the compiled code's quality generated by different compilers. Table 6-1 shows the execution time of ISS and the ARM core model for different workload and other quantity metrics.

Table 6-1: Performance evaluation of the ARM core model

C Program	Simulator	Execution time (s)	Number of instructions	Number of cycles	CPI
Fibonacci (15)	ISS	0.501			
	ARM core model	6.058	30631	46626	1.52
Summary	ISS	0.394			
	ARM core model	13.072	14373	18478	1.29
Array	ISS	0.275			
	ARM core model	0.031	297	370	1.25
Test2	ISS	0.066			
	ARM core model	0.029	75	117	1.56
Test3	ISS	0.095			
	ARM core model	0.039	103	135	1.31
Test4	ISS	It doesn't work, maybe there is a bug in the ISS			
	ARM core model	0.093	423	537	1.27
Test5	ISS	61.89			
	ARM core model	190.818	1,083,377	1,418,899	1.31

'Fibonacci' is recursive function call to calculate the numbers, also used to test function call, stack use. 'Summary' is to accumulate the number iteratively. 'Array' is also used to test memory access operations. They all work correctly on the ARM Core Model and the ISS.

'Test2' to 'Test5' are downloaded from [36]. They are used in EQNTOTT benchmark. EQNTOTT benchmark is a compute-intensive program that spends the majority of execution time in the function cmppt(). The function cmppt() has a loop that compares two strings of short integers. This function is similar to strncmp(), except it compares short integers instead of characters. Some compilers use EQNTOTT-specific optimizations to achieve the best possible run-time performance, but sometimes the compilers generate incorrect object code for those similar to EQNTOTT. ‘Test2’ to ‘Test5’ are used to test compilers’ output for those are only slightly different from EQNTOTT. ‘Test2’, ‘Test3’ and ‘Test5’ are run correctly, ‘Test4’ is run correctly on the ARM core model, it doesn’t wok on ISS, maybe there is a bug in the ISS.

‘Fibonacci’, ‘Summary’ and ‘Test5’ are long programs, the execution time on ISS is much shorter than on the ARM core model for the same workload, because ISS is only for run ARM object code, ARM core model not only run object code but also collect some information. The others are relative small programs, their execution time on ISS is a little longer than on ARM core model, this is because ISS also simulates virtual memory, system boot etc., for these programs, the ARM core model is more efficient than ISS.

CPI (Clock cycles Per Instruction) of the ARM core model, it can be used to evaluation the performance of an application when the ARM core model is plugged. The number of cycles and the number of instructions can be used to evaluate the compiled code’s quality.

Other quality metrics include extensibility [16], we can hook up to co-design systems, external bus models, memory hierarchies or coprocessor models without access to sources; interoperability [16], which has to do with its capability to integrate with other tool, such as debugger hardware simulator, etc. the debugger can obtain a snapshot of all the instructions in the pipeline and which instruction in each stage (e.g., fetch, decode, execute, memory cycle, or register write-back); traceability [16], which has to do with how flexible the simulator can collect useful statistics, such as instruction profiling; retargetability [16], which has to do with how easy the tool can be extended to handle new host platforms.

Chapter 7 Conclusion

ARM is a 32-bit machine with a register-to-register, three-operand instructions, control over both the Arithmetic Logic Unit and shifter in every data processing instruction, auto-increment and auto-decrement addressing modes, load and store multiple instructions, conditional execution of all instructions, it also has seven processor modes, every processor mode has its CPSR and SPSR except system mode and user mode (they do not have SPSR), so it can support multiple level interrupt.

The ARM core model is a cycle-accurate micro-architecture simulator of ARM processor that has 5-stage pipeline with forwarding path, hazard detect and interlock. Since the pipeline structure and its advanced properties are not described in the specification manual, so we combine the description in ARM specification manual [25][26] and methodology described in "Computer Architecture-A Quantitative Approach" [24] in this work. The main contributions of this work are:

- Provide a description of ARM pipeline implementation, this description can be considered as original.
- Present an open source of ARM cycle accurate micro-architecture simulator in SystemC, which doesn't exist in the public domain.

The ARM core model was validated by using an ARM ISS (Instruction Set Simulator) at instruction level (the ARM core model and ISS do not have the same precision). We also present the metric for performance evaluation. The simulator is also compatible with 3-stage pipeline simulator. The programs have the same result running on this model as running on other ARM simulator (3-stage pipeline). Its main uses are:

- Simulate the overall behavior of execution of programs that are intended for execution on an ARM system.
- Compare the quality of compiled code as produced by different compilers and/or compiler options.

- Evaluate the performance of an application by counting the number of cycles, calculating the CPI (Cycles Per Instruction).

Despite our effort to make the ARM core model a robust system, there are still areas for improvement. Some areas for future work are considered below:

- The ARM core model is now a standalone software project. We can encapsulate the ARM core model according to Object-Oriented methodology, separate the interface and implementation, so it can be a component to plug into a system and communication with other modules.
- Extend and integrate with other modules. For example memory hierarchy without having to rewrite major parts of the system, this can then be used to monitor memory access patterns in test programs such as temporal and spatial locality, size of memory requirement, etc.
- Plug into a system without changing the other modules in the system, only change the composition of the hardware system.
- Incorporate more precise clock cycles-per-instruction (CPI) for each class of instruction; possibly include some mechanism for adjusting the CPI for instructions that cause a cache miss.
- Extend to support SWI instruction and thumb instruction set.
- Implement 2 ARM processors integrated using AMBA bus with the different integration methodology described in [22].
- Extend to support hardware interrupts.

References

[1] "The Design of ARMphetamine 2, " Julian Brown, University of Bristol. http://www.cs.bris.ac.uk/~brown/docs/armphetamine2-design.html

[2] R. Cmelik, D. Keppel, "Shade: A Fast Instruction Set Simulator for Execution Profiling, " *Proceedings of Proceedings of the 1994 ACMSIGMETRICS* Conference on Measurement and Modeling of Computer Systems, pp. 128-137, May 1994.

[3] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems, " *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 1, pp. 78-103, January 1997.

[4] "ARMSim: An Instruction-Set Simulator for the ARM processor, " Alpa Shah Columbia University. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/reports/alpa.pdf

[5] P. G. Paulin, F. Karim, P. Bromley, "Network Processors: A Perspective on Market Requirements, Processor Architectures and Embedded S/W Tools, " *Proc. of the Design Automation and Test in Europe*, pp. 420-427, November 2000.

[6] "SWARM 0.44 Documentation" Michael Dales, Department of Computing Science, University of Glasgow,Scotland. http://www.dcs.gla.ac.uk/~michael/phd/bin/swarm-0.44.pdf

[7] R. C. Bedichek, "Some efficient architecture simulation techniques," In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference, January 22-26, 1990,Washington, DC, USA*, pp. 53-64, Berkeley, CA, USA, USENIX. January 1990.

[8] P. Magnusson and B. Werner, "Efficient memory simulation in SimICS, " In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[9] James R. "Bell Threaded code, " CACM, 16(6), June 1973.

[10] Bedicheck, R., "Talisman: Fast and accurate multicomputer simulation, " In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* pp. 14-24, May 1995.

[11] Dynamic compiler http://www.dcs.warwick.ac.uk/~csuix/project/

[12] SimARM; Green Hills Software Inc. http://www.ghs.com/

[13] The ARMulator; Document number: ARM DAI 0032; Issued 1999. http://www.arm.com/

[14] Emmett Witchel, Mendel Rosenblum, "Embra: Fast and Flexible Machine Simulation, " *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Philadelphia, pp. 68-79, 1996.

[15] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti, "Precise and Accurate Processor Simulation, " In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pp. 13-22, February 2002.

[16] J. Zhu and D.D. Gajski, "A retargetable, ultra-fast instruction set simulator, " In *Proceedings of Design, Automation and Test in Europe Conference,* pp. 9-12, March 1999.

[17] A. D. Pimentel and L. O. Hertzberger, "An Architecture Workbench for Multicomputers," *in Proc. of the 11th International Parallel Processing Symposium*, pp. 94-99, Geneva, Switzerland, IEEE Computer Society Press, April 1997.

[18] System on Chip design and reuse: http://www.us.design-reuse.com

[19] D. Flynn, "AMBA: Enabling Reusable On-Chip Designs, " *IEEE Micro*, Vol. 17, No. 4, pp. 20-27, July/Aug 1997.

[20] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient Exploration of the SoC Communication Architecture Design Space, " in *Proc. Int. Conf. Computer-Aided Design*, pp. 424-430, November 2000.

[21] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, "Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design, " *Design Automation Conference, DAC '01*, pp. 667-672, June 2001.

[22] Robert L. Veal, Levon Petrosian, Dr. Neal S. Stollon, "Multi-core SoC Platform Integration using AMBA, " *Proceedings of DesignCon2002 System-on-Chip and IP Design Conference*, January 2002.

[23] Andrew Burdass et al,"Embedded Test and Debug of Full Custom and Synthesisable Microprocessor Cores, " In *Proceedings IEEE European Test Workshop (ETW)*, pp. 17-22, Cascais, Portugal, May 2000.

[24] John Hennessy and David Patterson. "Computer Architecture A Quantitative Approach, " 2nd ed., Morgan Kaufman, 1996.

[25] Steve furber. "ARM System-on-Chip Architecture, " 2nd edition, Addison-wesley, 1999.

[26] David Seal. "ARM Architecture Reference Manual, " 2nd edition. Addison-wesley, 2000.

[27] "SystemC User's Guide, " version 2.0, Synopsys, Inc., CoWare, Inc., Frontier Design, Inc. http://www.ece.cmu.edu/~ece767/cad_tools/systemc-2.0.1/UserGuide20.pdf

[28] AMBA specification (Rev 2.0) ARM IHI 0011A. http://www-micro.deis.unibo.it/~magagni/amba99.pdf

[29] MIPS simulation tools http://www.mips.com/devTools/catalog_2001/SimTools.html

[30] Embedded system: www.embedded.com

[31] ARM company: www.arm.com

[32] Superlog language: www.superlog.org

[33] Accellera organization: www.accellera.org

[34] Verilog Resources: www.verilog.com

[35] SystemC Community: www.systemc.org

[36] EQNTOTT benchmarks: www.nullstone.com/eqntott/eqntott.htm

Appendix A: Instructions implemented in this model

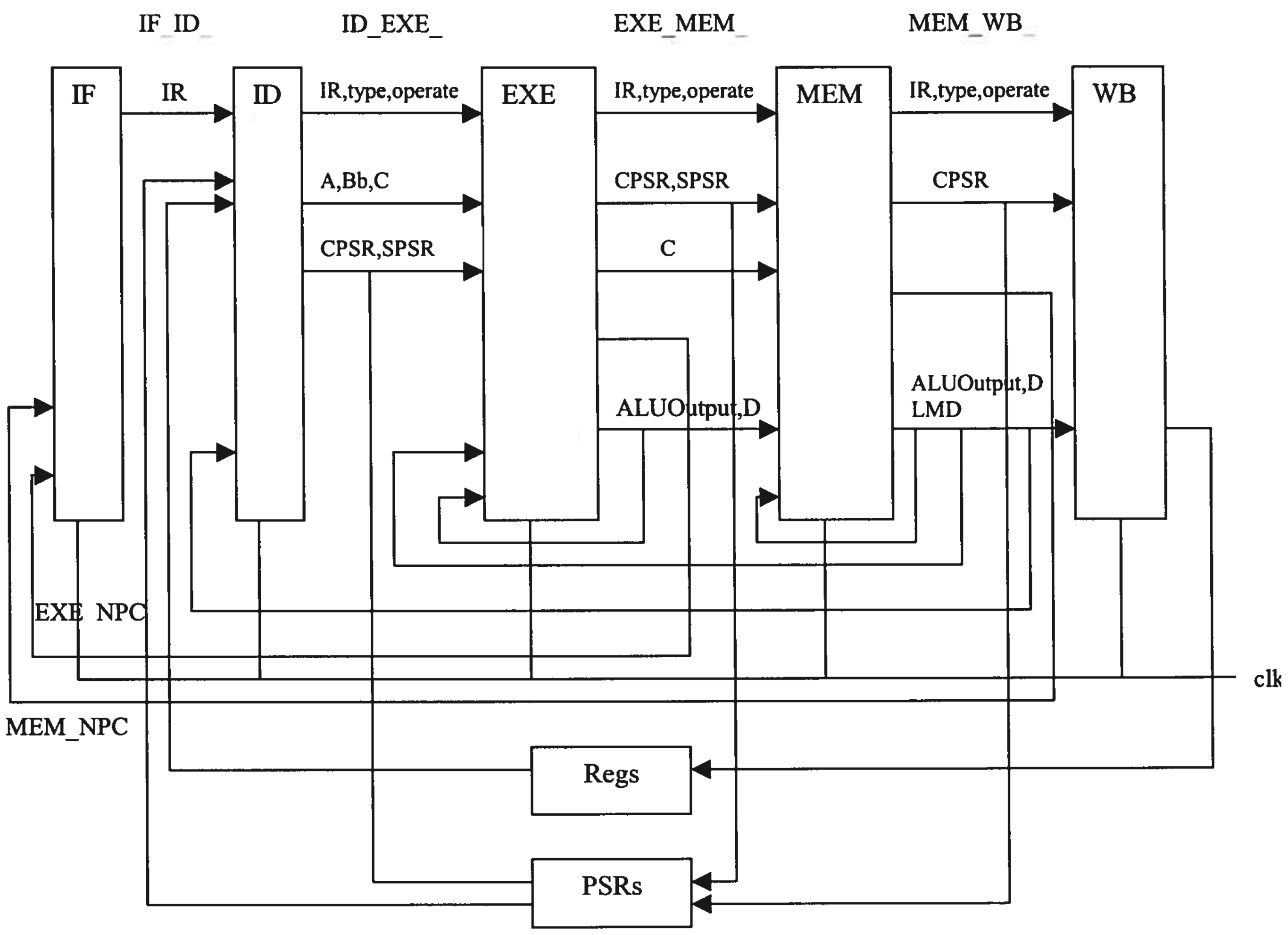
Instructions implemented in this model

Type of instruction	Instruction
Data processing	AND: logical AND. EOR: logical Exclusive OR. SUB: Subtract. RSB: Reverse Subtract. ADD: Add. ADC: Add with Carry. SBC: Subtract with Carry. RSC: Reverse Subtract with Carry. TST: Test. TEQ: Test Equal. CMP: Compare. CMN: Compare Negative. ORR: logical OR. MOV: Move. BIC: Bit Clear. MVN: Move Negative.
Multiply	MUL: Multiply. MLA: Multiply Accumulate. UMULL: Unsigned Long Multiply. UMLAL: Unsigned Long Multiply Accumulate. SMULL: Signed Long Multiply. SMLAL: Signed Long Multiply Accumulate.
CLZ	CLZ*:* Returns the number of binary zero bits before the first binary one bit in a register value
CPSR SPSR Access	MRS: move PSR to general-purpose register. MSR: move general-purpose register to PSR

Type of instruction	Instruction
LD/ST	LDR: Load Register. LDRB: Load Register Byte. LDRBT: Load Register Byte with Translation. LDRH: Load Register Half-word. LDRSB: Load Register Signed Byte. LDRSH: Load Register Signed Half-word. LDRT: Load Register with Translation. STR: Store Register. STRB: Store Register Byte. STRBT: Store Register Byte with Translation. STRH: Store Register Half-word. STRT: Store Register with Translation. LDM(3): Load Multiple Registers. There are 3 kinds of format. STM(2): Store Multiple Registers. There are 2 kinds of format. SWP: Swap a word. SWPB: Swap a Byte.
Exeception generate	SWI: Software Interrupt. BKPT: Breakpoint.
Coprocessor	CDP: Coprocessor Data Processing. STC: Store Coprocessor. LDC: Load Coprocessor. MCR: Move to Coprocessor from ARM Register. MRC: Move to ARM Register from coprocessor.
Branch	B: Branch. BL: Branch and Link. BLX(2): Branch and Link with exchange to Thumb instruction sets or ARM instruction sets. BX: Branch with an optional switch to Thumb execution.

Appendix B: ARM implementation model, source code description

1. ARM implementation model



2. The source file description

• IF.h	IF module interface declaration
• IF.cpp	IF module definition including open file and read ARM binary code
• ID.h	ID module interface declaration
• ID.cpp	ID module definition including interlock operation
• EXE.h	EXE module interface declaration

- EXE.cpp EXE module definition including calculate the result of arithmetic instructions and the address of load/store instructions
- MEM.h MEM module interface declaration
- MEM.cpp MEM module definition including memory access
- WB.h WB module interface declaration
- WB.cpp WB module definition including register write back
- main.cpp testbench including instantiate 5 stages, clock signal, open a file for writing trace signals, and then start simulation, write trace signals to the trace file for analyze
- forwarding0.cpp extern functions of forwarding operation related
- arm_inst.h constant and macro definition

3. Make file: makefile.linux is used in linux operating system (run: make –f makefile.linux) to generate the executable file (run.x).
4. Test case
    - Basic test: include files showed in table 5-1
    - Combination test: include files showed in table 5-2 and table 6-1

Appendix C: How to use the model

1. Generate the ARM binary code
   - Get object file by running:

     % **arm-elf-gcc ◊ –static ◊ –nostartfiles ◊ –nostdlib ◊ source.c ◊ –o ◊ objfile**
   - Get disassemble code and binary code by running:

     % **arm-elf-objdump ◊ –d ◊ objfile**
   - Extract binary code by running:

     % **arm-elf-objdump ◊ –d ◊ objfile | sed ◊ "s/.*\:\([^◊]#\).*/\1/" ◊ > armbinaryfile**
2. Add two instructions at the beginning of the file ‘armbinaryfile’ 0xe3a0ec01 (mov lr, 0x100) and 0x3a0dc01 (mov sp, 0x100), to initiate LR (Link Register) and SP (Stack Point) to 0x100 or other appropriate value (LR is used to terminate the simulation when finishing the program simulate, so it must be initialized to an area with consecutive 6 ‘00000000’ instructions. SP must be initialized to an empty area for stack operation).
3. Initiate RAM by storing enough data in the file named ‘ram’, if the ARM program needs to initiate some data in RAM, the data must be in the appropriate position (at the end of ARM program).
4. Run the simulator by typing: **./run.x ◊ armbinaryfile ◊ tracefile**

   It will simulate the ARM processor by running the ARM program in armbinaryfile and write the trace signal into the tracefile.