國 立 交 通 大 學

電機學院 IC 設計產業研發碩士班

碩 士 論 文

應用於週期精確指令集模擬器之 高效率 SystemC 建模技術

Efficient SystemC Modeling Technology for Cycle-Accurate Instruction Set Simulator

研 究 生：張詠翔

指導教授：黃俊達　博士

中 華 民 國 九 十 七 年 五 月

應用於週期精確指令集模擬器之
高效率 SystemC 建模技術

Efficient SystemC Modeling Technology for Cycle-Accurate Instruction Set Simulator

研 究 生：張詠翔　　　　Student：Yung-Hsiang Chang

指導教授：黃俊達　博士　　　　Advisor：Dr. Juinn-Dar Huang

國 立 交 通 大 學
電機學院 IC 設計產業研發碩士班
碩 士 論 文

A Thesis
Submitted to College of Electrical and Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Industrial Technology R & D Master Program on
IC Design

May 2008
Hsinchu, Taiwan, Republic of China

中華民國九十七年五月

應用於週期精確指令集模擬器之
高效率 SystemC 建模技術

研 究 生：張詠翔　　　　　　　指導教授：黃俊達　博士

國立交通大學

電機學院產業研發碩士班

摘　　　要

隨著晶片設計複雜度日趨複雜，電子系統層級設計正成為主流的設計方法。SystemC 是一種基於 C++，並支援各種虛擬層級，且廣泛應用於建模及驗證的電子系統層級設計語言。

本論文提出一應用於週期精確指令集模擬器之高效率 SystemC 建模技術。以 32 位元精簡指令集嵌入式處理器為例，我們使用 SystemC 將該處理器建模於兩個不同的虛擬層級—暫存器傳輸層級以及週期精確行為層級。該處理器所採用之指令集架構為 ARM 指令集(第四版)。我們測量所提出的模型及參照之 Verilog 模型之模擬時間來作效能比較。實驗數據顯示，我們提出的週期精確行為層級指令集模擬器之效能比參照之 Verilog 模型快十倍。

Efficient SystemC Modeling Technology for Cycle-Accurate Instruction Set Simulator

Student: Yung-Hsiang Chang Advisor: Dr. Juinn-Dar Huang

Industrial Technology R & D Master Program of
Electrical and Computer Engineering College
National Chiao Tung University

ABSTRACT

With the increase of the complexity of chip design, Electronic System-Level (ESL) design is becoming a mainstream design methodology. SystemC, a language based on C++, supports a broad range of abstraction levels and views, and is widely used as an ESL language both for modeling and verification.

In this thesis, an efficient SystemC modeling technology for cycle-accurate instruction set simulator (ISS) is presented. A 32-bit RISC embedded processor core is modeled in SystemC at two different abstraction levels, the register-transfer level (RTL) and the cycle-accurate behavioral level, as an evaluation case. The instruction set architecture (ISA) of the modeled processor core adopts ARM ISA Version 4. We measure the simulation time of the proposed models and the reference Verilog model for performance comparison. The experimental results show that our proposed cycle-accurate behavioral level ISS is ten times faster than the reference Verilog model.

Acknowledgement

致　　謝

能完成這篇論文，首先我要感謝指導教授黃俊達老師。老師除了授業及解惑專業知識之外，也教導我許多作研究的方法與態度，這些對我的未來裨益甚大。

感謝義隆電子在我的研究所求學生涯中給予生活上的協助。

感謝 ACAR 實驗室同學及學弟妹們，宏光、維聖、孝恩、翊展、士祐、之暉、哲霖、南興、威甝、建德、瀚蔚及于翔，感謝你們從旁給予建議與協助。

感謝清華電機 01 級的大學同學們，沅龍、思儒、建閔、毅軒、吉興、建翔等，感謝你們支持我、鼓勵我。各位分享你們的工作經驗，讓我見賢思齊。在此也向其他關心我的朋友、老同學們說聲感謝。

最後且最重要地，感謝我的父母與家人。感謝父母親二十多年來的養育之恩，有您們無悔的付出，成就了今日的我。

我願將這篇論文獻給支持我的大家。謝謝你們！！

Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Motivation

With the progress of semiconductor technology, more and more transistors can be integrated into a chip and the complexity of chip design increases. In order to reduce chip design costs and ease the pressure of time-to-market, electronic system level (ESL) [1] methodology has been proposed and is becoming a mainstream.

SystemC [2], a language based on C++, provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. It covers a broad range of abstraction levels and views, and is widely used as an ESL language both for modeling and verification.

When developing a processor, an instruction set simulator (ISS) is necessary and is usually written in a high level language like C/C++. An ISS can be used for algorithms testing, architectural exploration and functional verification. Also, software developers can take advantage of an ISS to develop software applications in early design phase and shorten the development time.

The accuracy of an ISS depends on what *view* is taken by the designer such as functional view (FV), programmer's view (PV), architecture view (AV), and verification view (VV). A related work, by Lu et al. [3], shows that simulation speeds of PV model and AV model both written in SystemC are approximately 18 and 500 times faster, respectively, than that of Verilog RTL model. However, it does not point out the performance comparison between SystemC-based cycle-accurate ISS and Verilog-based RTL model in VV.

An efficient cycle-accurate ISS can serve as a golden model to verify the RTL design. Besides, as shown in Fig. 1.1, it can replace the RTL design and serve as a fast cycle-accurate functional model when performing system-level simulation.

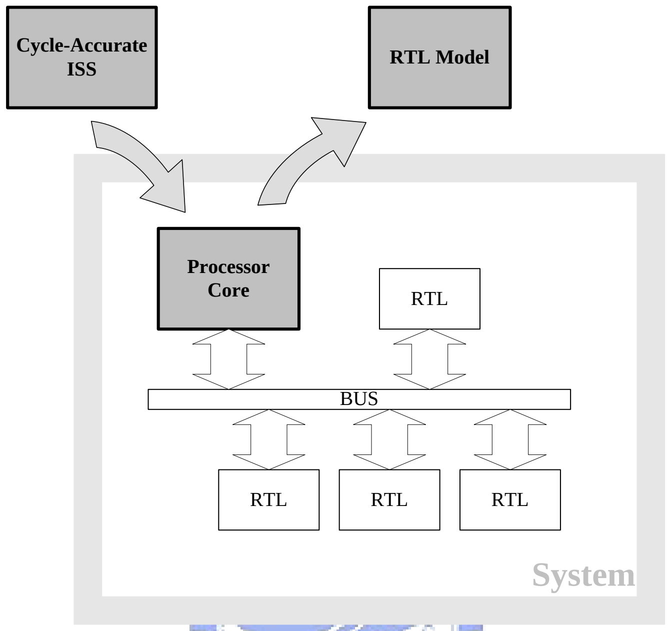


Fig. 1.1 An efficient cycle-accurate ISS can serve as a fast cycle-accurate functional model when performing system-level simulation.

1.2 Contribution

This thesis proposes an efficient SystemC modeling technology for cycle-accurate ISS. One trivial way for cycle-accurate ISS development is modeling the processor at RT-level directly. Our proposed way is modeling the processor in behavioral level with a cycle-accurate wrapper. The experimental results show that the performance comparisons among different modeling techniques. The simulation speed of our proposed cycle-accurate behavioral level ISS is ten times faster than that of reference Verilog RTL design.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 describes preliminaries of this thesis. We give the overview of SystemC and the abstraction levels of processor modeling. Also, we introduce the instruction set architecture (ISA) that we adopt for ISS implementation. Chapter 3 describes the SystemC RTL model. Chapter 4 describes the proposed SystemC cycle-accurate behavioral level ISS. Chapter 5 shows our verification strategies. Chapter 6 discusses the experimental results and Chapter 7 concludes this thesis and describes the future works. The reference is provided afterward.

Chapter 2
Preliminaries

This chapter describes the preliminaries of this thesis. Section 2.1 introduces the SystemC basic components. Section 2.2 describes the abstraction levels and views of processor modeling. At last, Section 2.3 introduces the ARM7TDMI and the related processor ACARM7.

2.1 Overview of SystemC

SystemC is based on the C++ programming language. It can be used at a very high level such as system level and can also be used at lower levels such as the register transfer level. It basically extends the capabilities of C++ to enable hardware description. Fig. 2.1 shows basic components of SystemC.

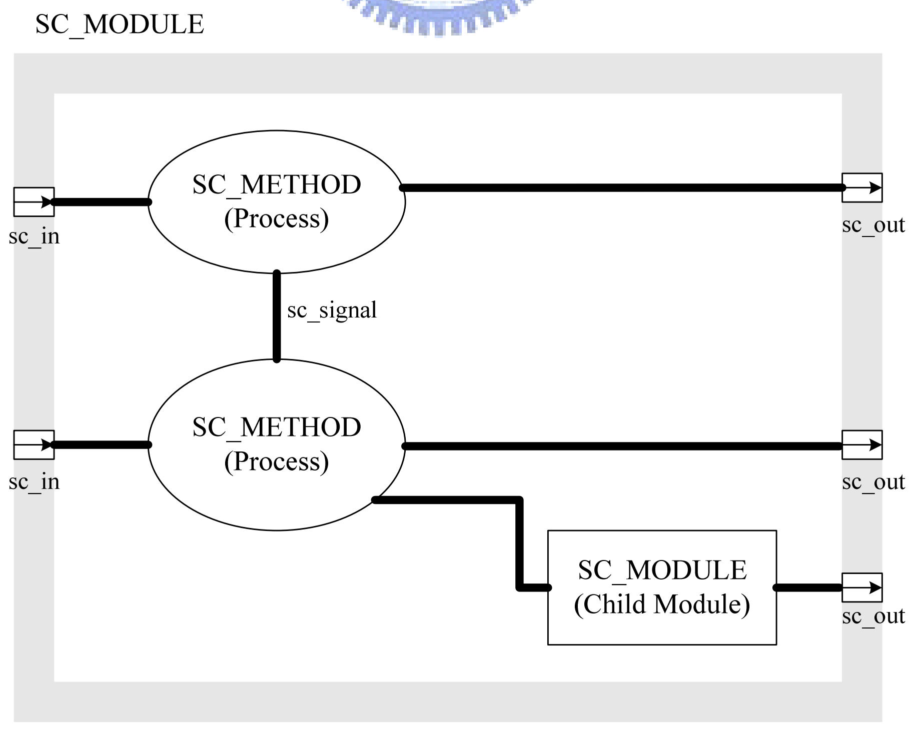


Fig. 2.1 Basic components of SystemC: A module, ports, processes and signals.

- Modules: A module is the basic unit for describing a structure or class in SystemC and can be described by the SC_MODULE macro. A module can be hierarchical in that it can have processes and the other child modules instantiated within it.
- Processes: A process is used to describe functionality. A module can have one or more processes. There are three kinds of processes: SC_METHOD, SC_THREAD, and SC_CTHREAD. An SC_METHOD process can be invoked multiple times but can not be suspend during the execution. An SC_THREAD or an SC_CTHREAD can be invoked only once and it has the option to suspend itself.
- Ports: Ports including input, output, and inout ports are used to communicate with other modules.
- Signals: A signal is used to connect processes and child modules.

Besides the C++ native data types, SystemC also supports synthesizable hardware-oriented data types in SystemC RTL as shown in Table 2.1 [4].

Table 2.1 SystemC data types that are supported in SystemC RTL.

Name	Description
sc_bit	Single bit with two values, ‘0’ and ‘1’
sc_bv<n>	Arbitrary width bit vector
sc_logic	Single bit with four values, ‘0’, ‘1’, ‘X’, ‘Z’
sc_lv<n>	Arbitrary width logic vector
sc_int<n>	Signed integer type, up to 64 bits
sc_uint<n>	Unsigned integer type, up to 64 bits
sc_bigint<n>	Arbitrary width signed integer type
sc_biguint<n>	Arbitrary width unsigned integer type

2.2 Processor Modeling at Different Abstraction Levels and Views

A processor can be modeled at different abstraction levels and views to meet different evaluation requirements. Different abstraction levels such as register-transfer level and transaction-level modeling (TLM), different views such as Programmer's View (PV), Architecture View (AV) and Verification View (VV), all have different functionalities and play different roles in design and verification, as shown in Table 2.2 [3].

Table 2.2 Abstraction of different views of modeling.

View	Model Accuracy & Purpose
Functional View	Event ordering. Functional specification and algorithm development.
Programmer's View	Bit accurate. Software development and verification.
Architecture View	Cycle approximate. Architectural exploration and verification.
Verification View	Cycle accurate. System level Verification.

A Functional View model is used for developing and verifying software functions and algorithms. Only the instruction set of the processor is modeled. This model is just instruction-accurate and un-timed for achieving high simulation speed. Similarly, a Programmer's View model is used for software development too. Typically, the model is bit-accurate and un-timed.

On the other hand, an Architecture View model is used for architecture exploration, which is transaction-accurate and usually a cycle approximate model. As for Verification View, it is a cycle accurate model and typically focuses on hardware-software co-simulation and co-verification.

2.3 Overview of ARM7TDMI and ACARM7 Verilog RTL Model

2.3.1 Core Architecture of ARM7TDMI

ARM7TDMI [5, 6] is a general-purpose 32-bit RISC processor developed by Advanced RISC Machine (ARM) Limited. It employs the von Neumann architecture which uses a single memory to hold both instructions and data. Fig. 2.2 illustrates the ARM7TDMI core diagram [5]. The major components are:

- The address register and incrementer, which select and hold all memory addresses and generate sequential addresses when required.
- The register file, which contains 31 32-bit general purpose registers and six status registers.
- The 32x8 multiplier, which can perform multi-cycle multiply operations.
- The barrel shifter, which can shift or rotate one operand by an immediate offset or a register offset.
- The ALU, which can perform arithmetic and logic operations.
- The read and write data registers, which hold data storing to or lording from memory.
- The instruction decoder and control logic, which can decode instructions and generate associated control signals.

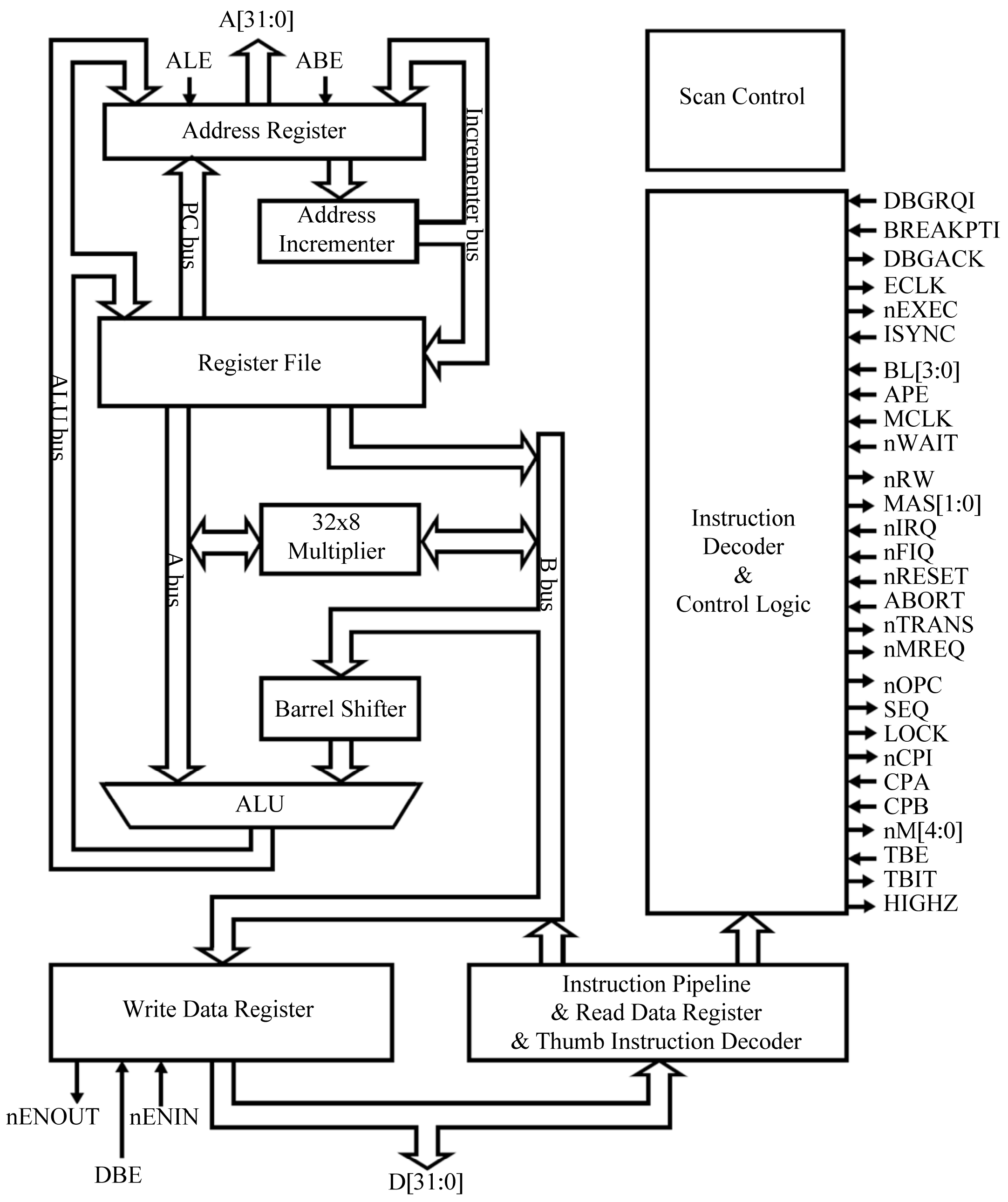


Fig. 2.2 ARM7TDMI core diagram.

The ARM7TDMI adopts a 3-stage pipeline including the instruction fetch (IF) stage, the instruction decode (ID) stage, and the execution (EX) stage. In the IF stage, the instruction is fetched from memory and then queued in the instruction pipeline. In the ID stage, the instruction is decoded and the control signals are generated for the next cycles. In the EX stage, the instruction enters into the datapath. The register bank is read, the second operand is shifted or rotated if needed, and the result is generated by ALU and written back to a destination register. For a single-cycle instruction, such as a data processing instruction, the processor takes one EX stage cycle as shown in Fig. 2.3. For a multi-cycle instruction, the processor takes more than one EX stage cycle. As shown in Fig. 2.4, a sequence of single-cycle ADD instructions, with a data store instruction, STR, occurring after the first ADD instruction. The cycles that access memory are shown with shading and we can see that the memory is used in every cycle. The datapath is likewise used in every cycle since two EX stage cycles, one is for address calculation and the other is for data transfer, are taken by STR instruction. Actually, the memory accessing is the major limiting factor that limits the number of cycles taken by a sequence of instructions.

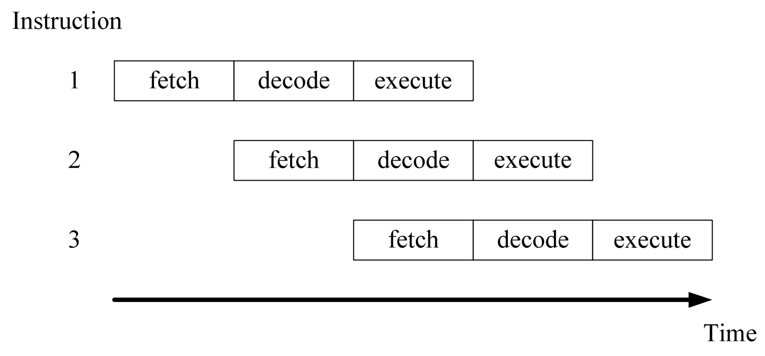


Fig. 2.3 ARM7TDMI single-cycle instruction 3-stage pipeline operation.

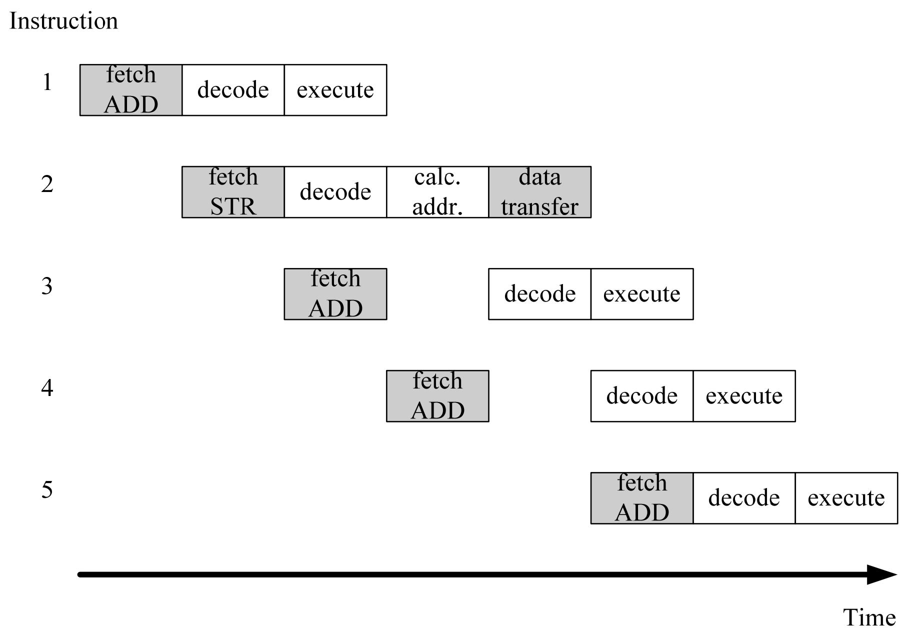
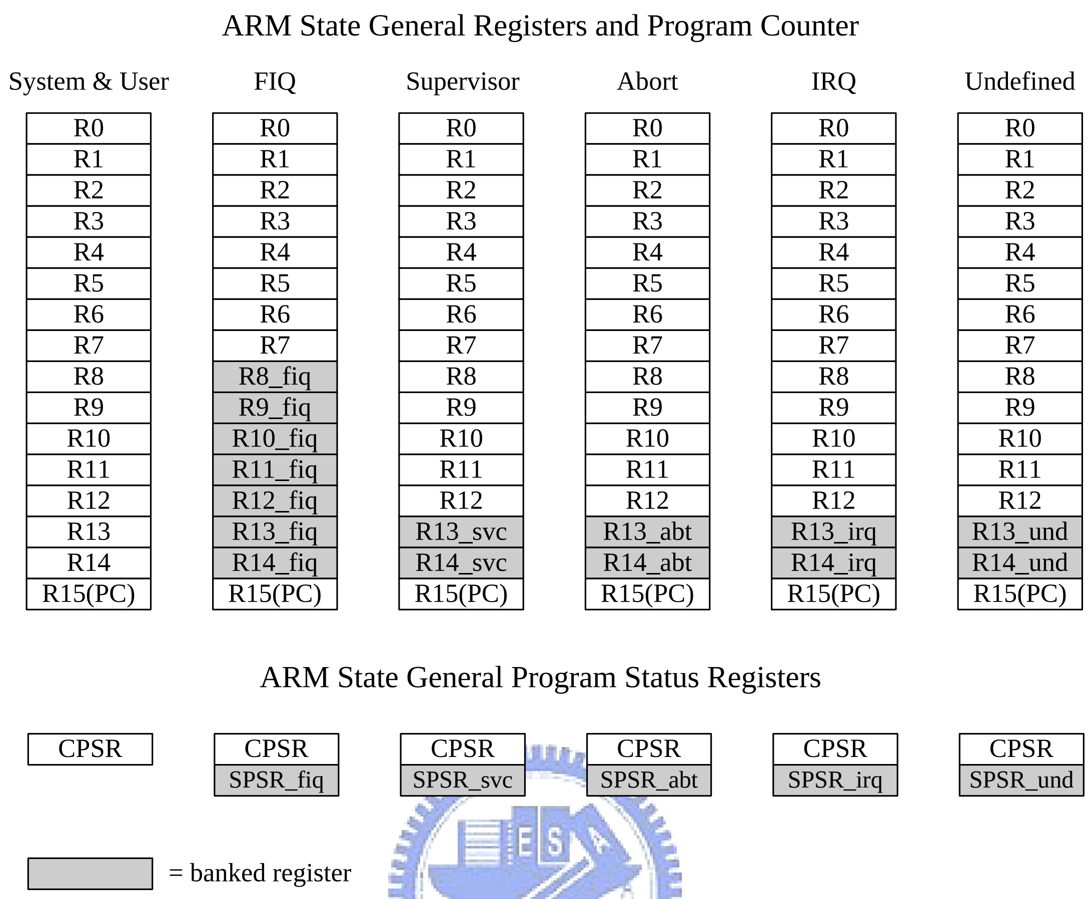


Fig. 2.4 ARM multi-cycle instruction 3-stage pipeline operation.

2.3.2 Programmer's Model

ARM7TDMI can be in one of two states, ARM state or THUMB state. In ARM state, the processor executes 32-bit, word-aligned ARM instructions; while in THUMB state, the processor executes 16-bit, halfword-aligned THUMB instructions. Since the proposed designs described in Chapter 3 and 4 implement ARM ISA without THUMB instructions and coprocessor instructions, we only focus on the ARM state in this thesis.

ARM7TDMI has 37 registers including 31 general-purpose registers and six status registers as shown in Fig. 2.5 [5]. The processor state and operating modes determine which registers are available to the programmer. ARM7TDMI supports seven operating modes including user mode, system mode, fiq mode, supervisor mode, abort mode, irq mode, and undefined mode. The non-user modes, a.k.a. privileged modes, are used for system level programming and typically for handling interrupts or exceptions.



ARM State General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	▨ R8_fiq	R8	R8	R8	R8
R9	▨ R9_fiq	R9	R9	R9	R9
R10	▨ R10_fiq	R10	R10	R10	R10
R11	▨ R11_fiq	R11	R11	R11	R11
R12	▨ R12_fiq	R12	R12	R12	R12
R13	▨ R13_fiq	▨ R13_svc	▨ R13_abt	▨ R13_irq	▨ R13_und
R14	▨ R14_fiq	▨ R14_svc	▨ R14_abt	▨ R14_irq	▨ R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

ARM State General Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	▨ SPSR_fiq	▨ SPSR_svc	▨ SPSR_abt	▨ SPSR_irq	▨ SPSR_und

▨ = banked register

Fig. 2.5 ARM7TDMI register organization in ARM state.

ARM7TDMI has six program status registers (PSRs) including a current program status register (CPSR) and five saved program status registers (SPSRs). These registers are used to hold information about the most recently executed ALU operation, control the enabling and disabling of FIQ/IRQ interrupts, and set the processor operating mode. Fig. 2.6 [5] shows the PSR format. The top four bits (N, Z, C, V), a.k.a. the condition code flags, may be affected by a result of an arithmetic/logical instruction, and may be used to determine whether an instruction should be executed. The bottom eight bits are the control bits. When I bit or F bit set, the processor disables IRQ and FIQ interrupts, respectively. The T bit describes the state of ARM7TDMI. When T bit set, the processor is in THUMB state, otherwise the processor is in ARM state. The M4, M3, M2, M1, M0 bits are the mode bits. The mode bits determine the current operating mode of the processor. Table 2.3 shows the mode bit values and the corresponding operating modes.

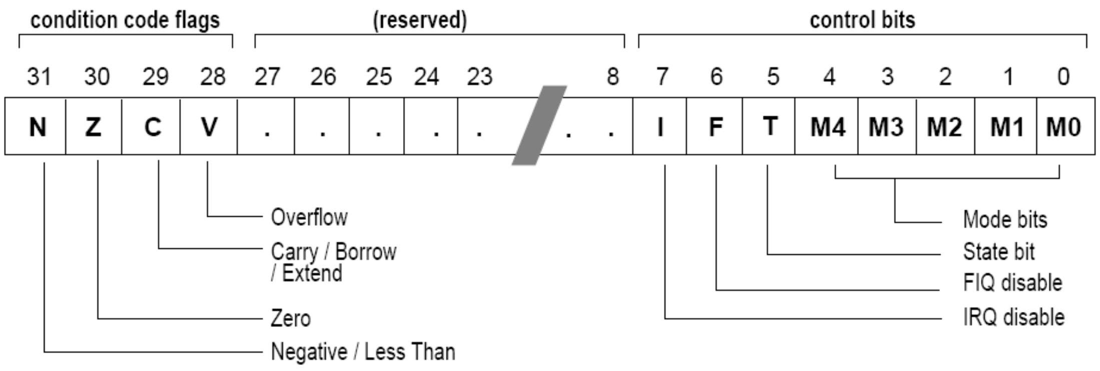


Fig. 2.6 Program status register format.

Table 2.3 PSR mode bit values.

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

ARM exceptions can be classified into three groups:

- Exceptions generated as the direct effect of executing an instruction, such as software interrupt (SWI), undefined instruction, and prefetch abort (instruction fetch memory fault).
- Exceptions generated as the side-effect of an instruction, such as data abort (data access memory fault).
- Exceptions generated externally, unrelated to the instruction flow, such as reset, IRQ (normal interrupt request), FIQ (fast interrupt request).

When two or more exceptions arise at the same time, the exception priorities determine the order in which the exceptions are handled. Table 2.4 shows the exception priorities from high to low.

Table 2.4 Exception priorities.

Priority	Exception
1	Reset
2	Data abort
3	FIQ
4	IRQ
5	Prefetch abort
6	Undefined instruction Software interrupt

2.3.3 ARM Instruction Set Architecture Version 4

The ARM7TDMI adopts the ARM ISA version 4. It can be classified into eight instruction types:

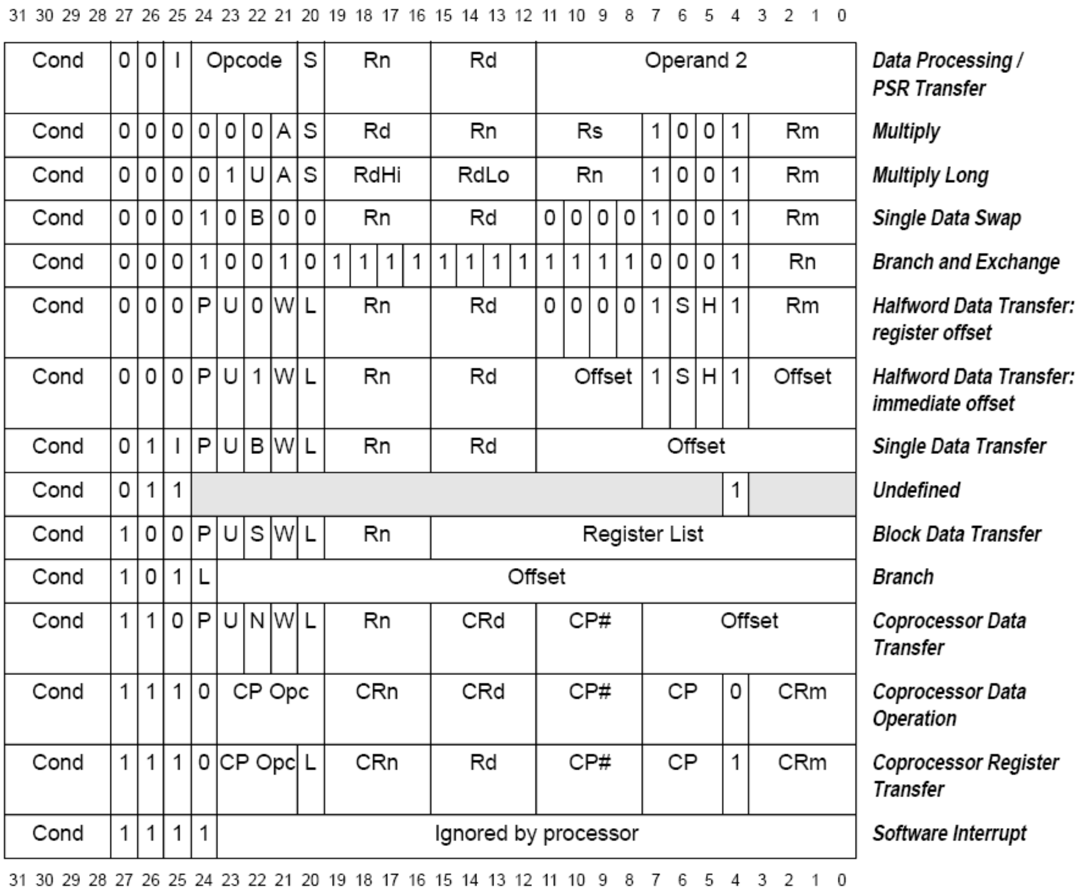
- Data processing instructions. These instructions perform arithmetic and logical operations on data values in registers and typically require two operands and produce one single result. These instructions allow the second register operand performing a shift or rotate operation before it is operated with the first operand.
- Program status register transfer instructions. The MRS instruction moves PSR contents to a register. The MSR instruction moves a register contents or a 32-bit immediate value to the relevant PSR. Note that in user mode, the control bits of the CPSR are protected from change, so only the condition code flags of CPSR can be changed.
- Multiply instructions. These instructions perform multiply and multiply-accumulate operations.
- Data transfer instructions. These instructions copy memory values into registers (lord instructions) or copy register values into memory (store instructions).
- Swap instruction. This instruction exchanges values between a memory location and a register.
- Branch instructions. These instructions execute to switch to a different address, either permanently (B) or saving a return address (BL).
- Coprocessor instructions. This class of instructions is used to tell a coprocessor to perform some data operations or data transfers.
- Software interrupt instruction. The SWI instruction is used to enter supervisor

mode in a controlled manner.

Fig. 2.7 shows the encoding formats, and Table 2.5 lists the complete set of the ISA [5]. In the Fig. 2.7 [5], the most significant four-bit segment of each instruction is called condition field. In ARM state, all instructions are conditionally executed according to the CPSR condition codes and the condition field of the instruction. The instruction is only executed when the condition is true, otherwise it is ignored. Table 2.6 lists the conditions [5].

2.3.4 ACARM7 Verilog RTL Model

The ACARM7 Verilog RTL model is an ultra low-power, high performance ARM7-like processor core. This Verilog model and our proposed design are co-developed based on the same ISA. We use ACARM7 Verilog RTL model as our reference model for co-simulation, co-verification and performance comparison.



31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	
Cond	0	0	I	Opcode				S	Rn	Rd	Operand 2						*Data Processing / PSR Transfer*
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	*Multiply*
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm	*Multiply Long*
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	*Single Data Swap*
Cond	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rn	*Branch and Exchange*
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	*Halfword Data Transfer: register offset*
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset	1	S	H	1	Offset	*Halfword Data Transfer: immediate offset*
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset						*Single Data Transfer*
Cond	0	1	1												1		*Undefined*
Cond	1	0	0	P	U	S	W	L	Rn	Register List							*Block Data Transfer*
Cond	1	0	1	L	Offset												*Branch*
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset					*Coprocessor Data Transfer*
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP			0	CRm	*Coprocessor Data Operation*
Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP			1	CRm	*Coprocessor Register Transfer*
Cond	1	1	1	1	Ignored by processor												*Software Interrupt*

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

Fig. 2.7 ARM instruction set formats.

Table 2.5 The ARM Instruction set

Mnemonic	Instruction	Action
ADC	Add with carry	Rd := Rn + Op2 + Carry
ADD	Add	Rd := Rn + Op2
AND	AND	Rd := Rn AND Op2
B	Branch	R15 := address
BIC	Bit Clear	Rd := Rn AND NOT Op2
BL	Branch with Link	R14 := R15, R15 := address
BX	Branch and Exchange	R15 := Rn, T bit := Rn[0]
CDP	Coprocesor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags := Rn + Op2
CMP	Compare	CPSR flags := Rn - Op2
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd := (address)
MCR	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}
MLA	Multiply Accumulate	Rd := (Rm * Rs) + Rn
MOV	Move register or constant	Rd : = Op2
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn := PSR
MSR	Move register to PSR status/flags	PSR := Rm
MUL	Multiply	Rd := Rm * Rs
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2
ORR	OR	Rd := Rn OR Op2

Table 2.5 The ARM Instruction set (Continued)

Mnemonic	Instruction	Action
RSB	Reverse Subtract	Rd := Op2 - Rn
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry
STC	Store coprocessor register to memory	address := CRn
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	<address> := Rd
SUB	Subtract	Rd := Rn - Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2
TST	Test bits	CPSR flags := Rn AND Op2

Table 2.6 Condition code summary.

Code	Suffix	Flags	Meanings
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Chapter 3 Proposed SystemC Register-Transfer Level Model

This chapter describes the proposed SystemC RTL design. An ARM7-like, 32-bit RISC embedded processor core is implemented in the register-transfer level with SystemC language. Section 3.1 depicts the block diagram of the proposed design. Section 3.2 describes the control logic of the proposed design. Section 3.3 describes how an arithmetic/logical operation functions in EX stage, and Section 3.4 describes the mechanism of multi-cycle multiplication in EX stage.

3.1 Block Diagram

The proposed SystemC RTL model consists of 10 major functional blocks including the decoder, register file, barrel shifter (BS), arithmetic/logical unit (ALU), 32x8 multiplier (MUL), forwarding unit, address register, read data selector, write data selector, and control logic. Fig. 3.1 shows the block diagram of the proposed design.

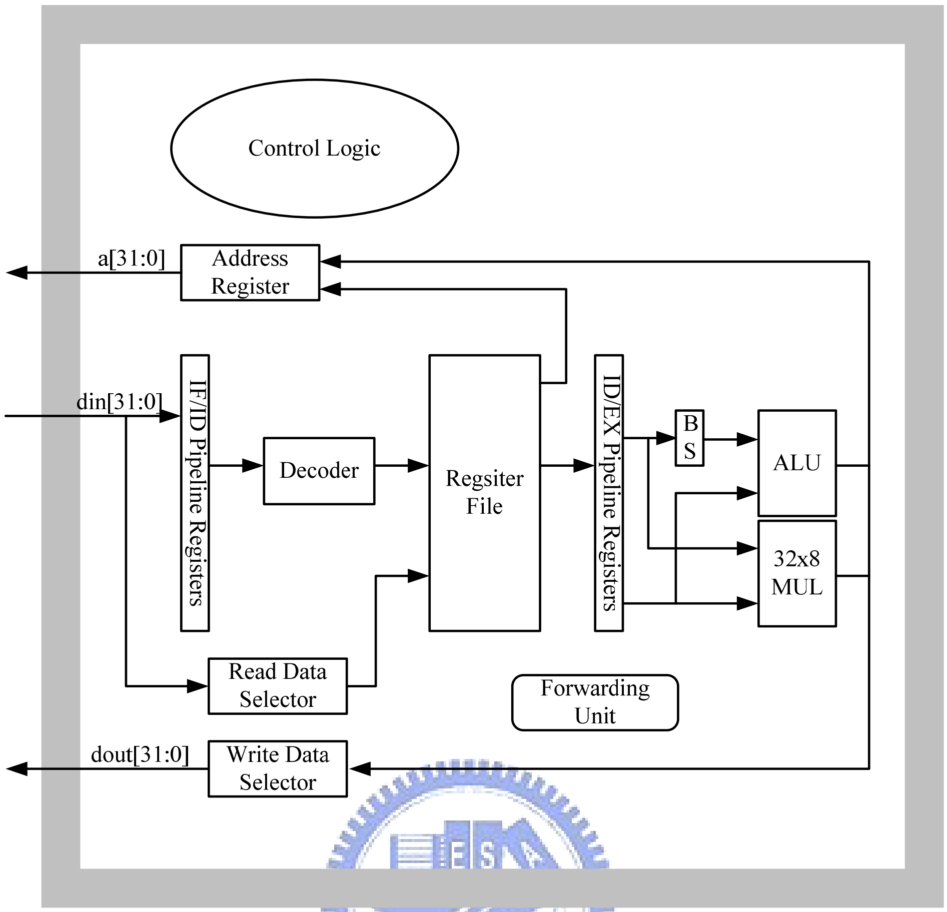


Fig. 3.1 Block diagram of the proposed SystemC RTL model.

The decoder unit obtains the instruction from IF phase and decodes it to generate all the information needed for the other functional units.

The register file is composed of total 37 registers – 31 general-purpose 32-bit registers and six status registers as shown in Fig. 2.5.

The execution stage consists of a BS, an ALU, and a 32x8 multiplier. The arithmetic and logical operations are implemented with the ALU module whose second operand is received from the BS to perform shift operations if needed. The 32x8 multiplier produces a 40-bit product. More details of multi-cycle multiply operation will be described in Section 3.4.

The forwarding unit forwards the output data of EX stage to make sure that the next instruction can get these data as soon as possible.

The address register chooses one valid address from four sources including the PC incrementer, the ALU output, LDM (lord multiple)/STM (store multiple) output, and the interrupt as shown in Fig. 3.2.

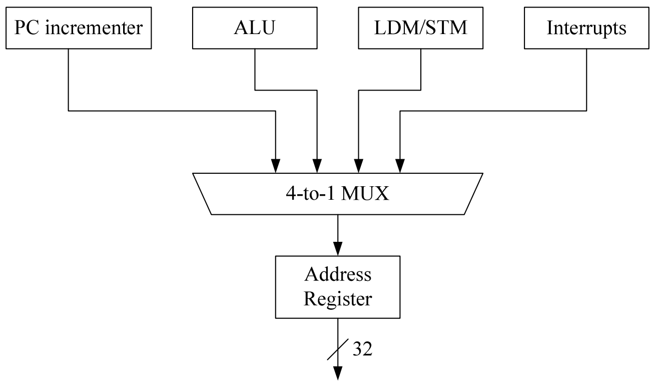


Fig. 3.2 Source selection of the address register.

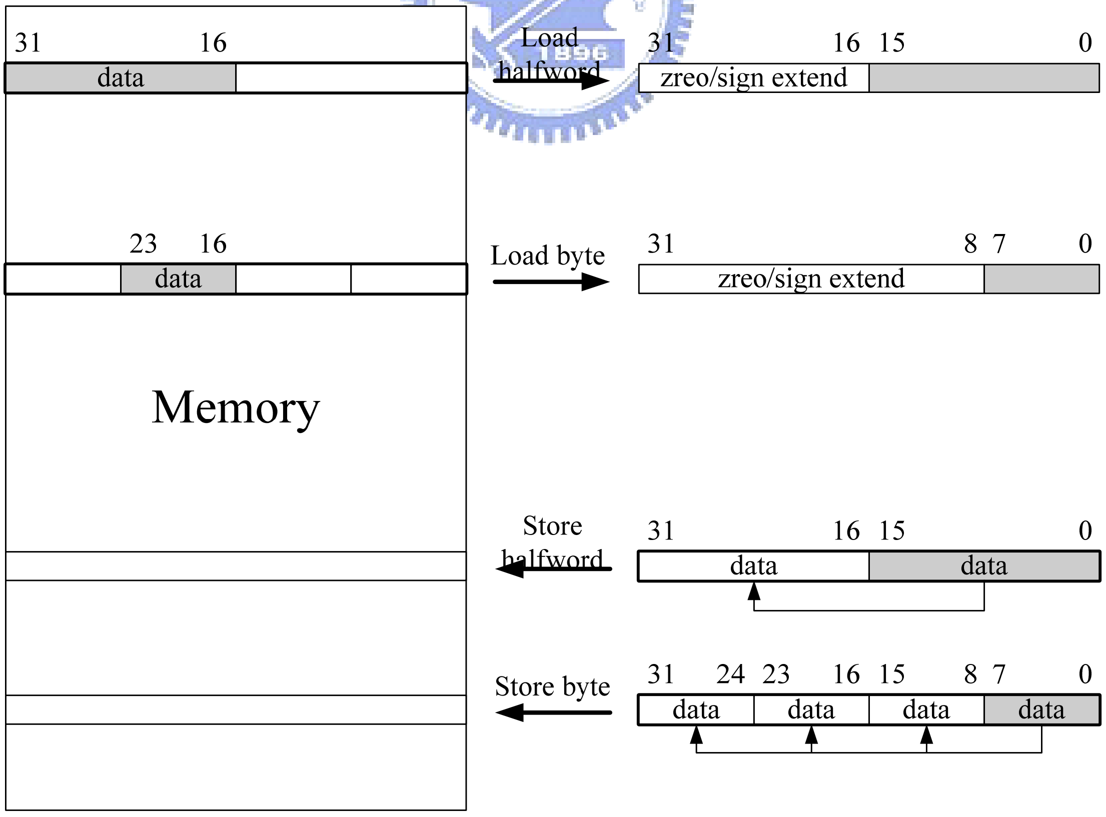


Fig. 3.3 Read/write selector.

The read/write selector performs data alignment operations. As shown in Fig. 3.3, the read selector shifts byte data and halfword data to the bottom of a 32-bit register with zero-extended or sign-extended when the processor reads byte data or halfword data from memory. For writing halfword data to memory, the write selector copies the low halfword part to the high halfword part to fill the 32-bit data width. For writing byte data to memory, the write selector copies the least significant byte to the other three more significant bytes.

3.2 Control Logic

The control logic controls all the data flows of combinational logic and the sequential logic. The instruction operation cycles are controlled by five finite state machines (FSMs) including lord/store sub-FSM, shift sub-FSM, multiplication sub-FSM, branch sub-FSM, and one main FSM as shown in Fig. 3.4.

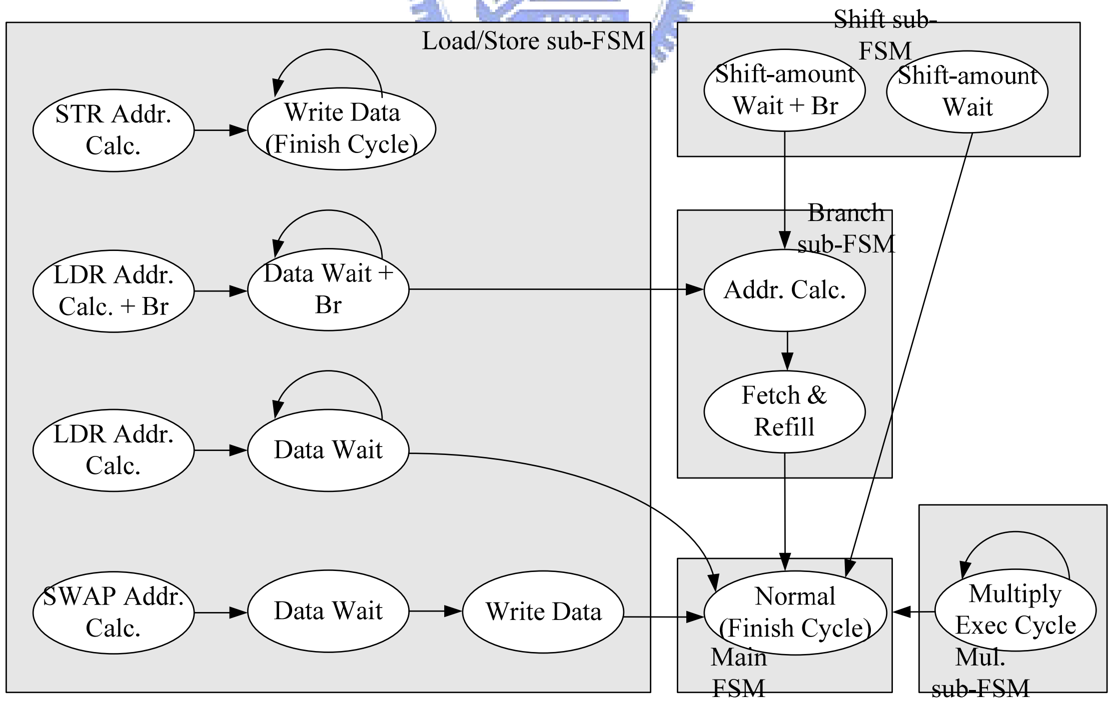


Fig. 3.4 FSMs of the control logic.

The lord/store sub-FSM handles four types of instructions including store, load with branch, lord without branch, and swap. Store instructions contain the single data store instruction and the multiple data store instruction. For the single data store instruction, the destination address is calculated at the first state and the data is written to memory at the second state then finishes the instruction. For the multiple data store instruction, the sub-FSM stays at the second state until all data transfers are done. Except the store instructions, all the other instructions finish their last execution cycle at the normal main state. Like store instructions, load instructions also contain the single data load instruction and multiple data load instruction. The load address is calculated at the first state, and then the data is fetched from memory at the second state. The single data load instruction finishes their last execution cycle at the normal main state; and the multiple data load instruction moves to the normal main state at the last execution cycle until all data transfers are done. The load with branch instruction will moves to the branch sub-FSM after the load operation. The swap instruction performs data exchange between a register and external memory, and can be implemented by a load operation followed by a store operation.

The shift sub-FSM handles data processing instructions with shifted register amount. When the destination register is r15 (PC), the processor moves to branch sub-FSM to perform branch operation.

The multiplication sub-FSM handles the multiply instructions. The finish signal is sent to the main FSM and the processor moves to the normal main state at the last execution cycle. More details will be discussed in Section 3.4 later.

The branch sub-FSM handles the instructions that use r15 as the destination register. A new PC address is calculated at the first state and the proper new instruction is then fetched to refill the pipeline at the second state.

3.3 Arithmetic/Logical Operation in Execution Stage

Fig. 3.5 shows the detailed block diagram of EX stage.

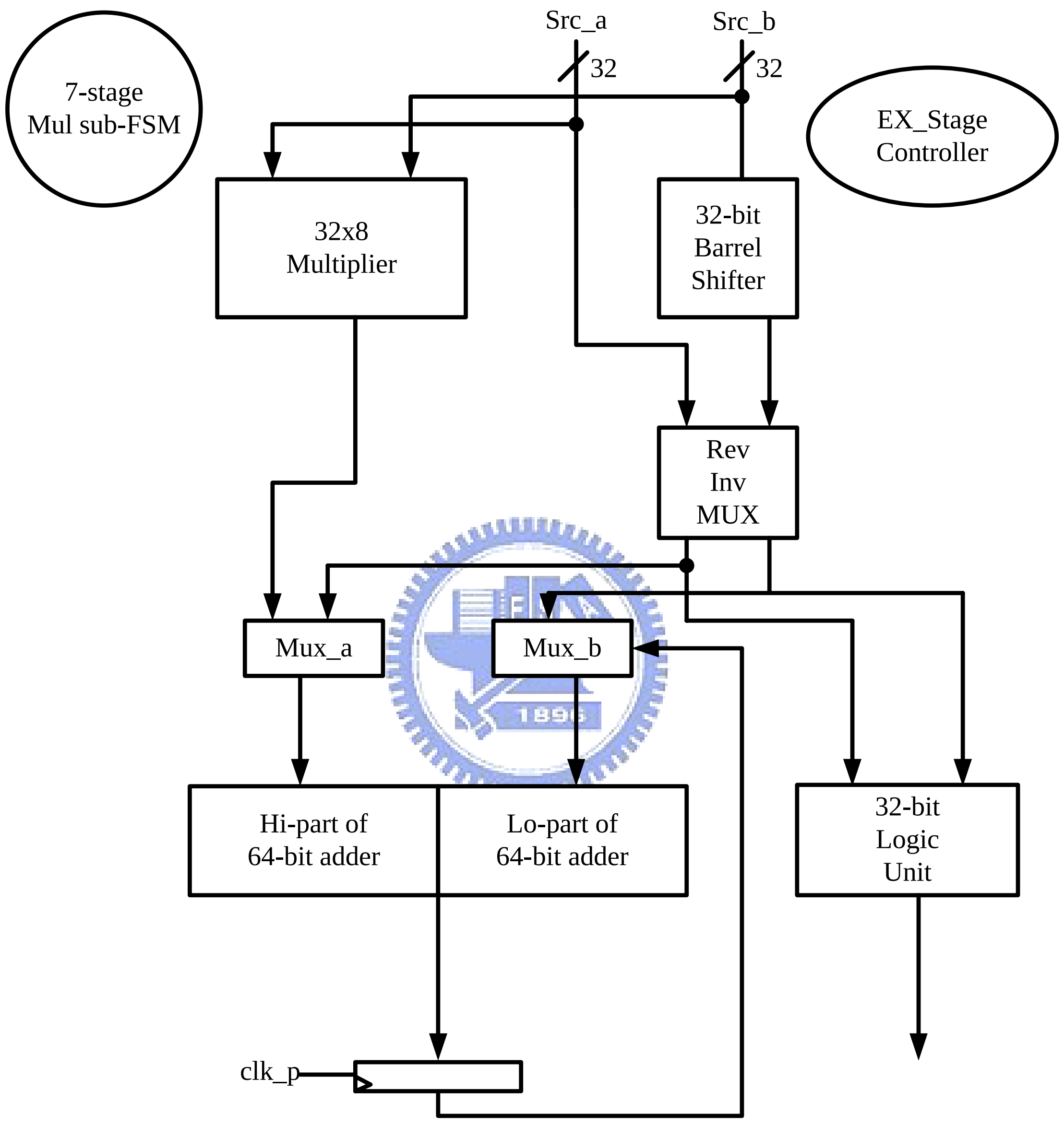


Fig. 3.5 Detailed block diagram of EX stage.

Arithmetic and logical operations are performed by ALU. Src_a and Src_b are the two inputs of the EX stage and are fed to the multiplier directly to perform multiplication instructions which will be discussed in the next section.

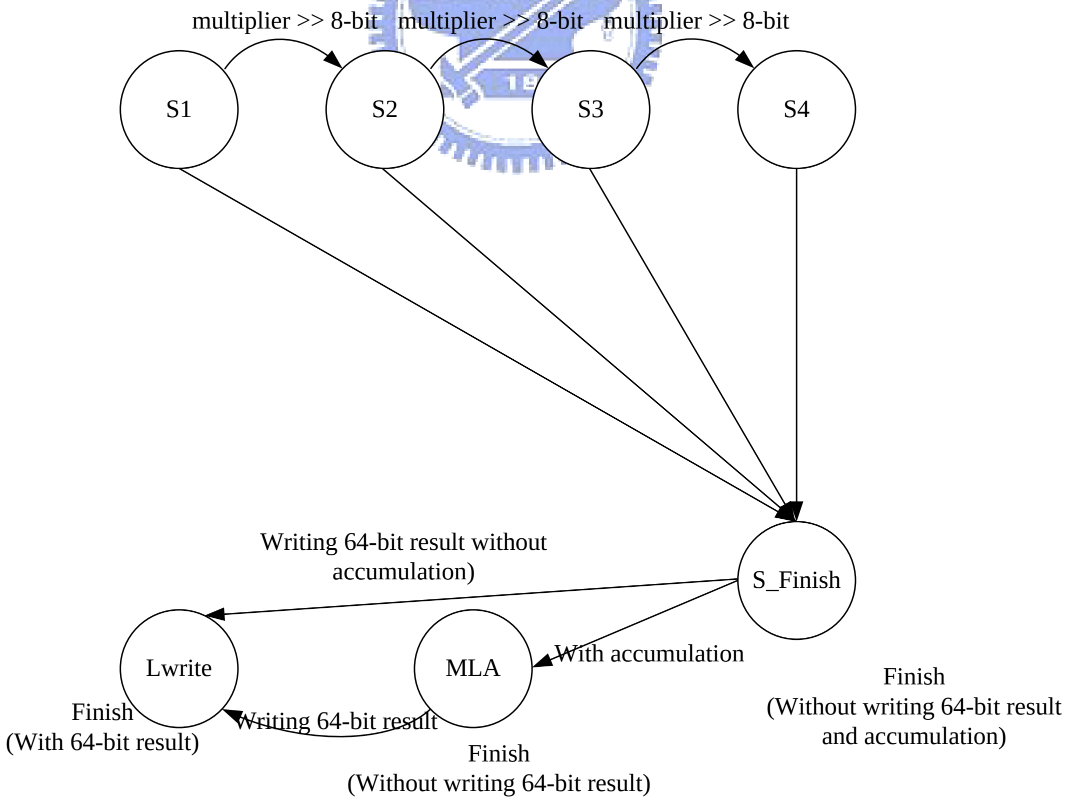
Src_b can be fed into BS to perform any type of shift/rotate operations including logical shift left, logical shift right, arithmetic shift right, and rotate right. Then the output of BS and Src_a are fed into a reverse-inverse-multiplexer which decides whether the two inputs should be inversed due to the consideration about subtraction or reversed for specific instructions like RSB and RSC.

The results of the reverse-inverse-multiplexer are sent to a logical unit to perform a logical operation, or sent to a 64-bit adder [7] to perform an arithmetic operation.

The arithmetic instructions like ADD, SUB, ADC, …, use the higher 32-bit part of the adder. The lower 32-bit part is filled with zeros.

3.4 Multi-cycle Multiplication in Execution Stage

For a multiply instruction, Src_a and Src_b are fed to the multiplier [8] directly to perform multiply operation. Fig. 3.6 shows the 7-stage multiplication sub-FSM.



multiplier >> 8-bit
multiplier >> 8-bit
multiplier >> 8-bit
S1
S2
S3
S4
Writing 64-bit result without accumulation)
S_Finish
Lwrite
MLA
With accumulation
Finish (With 64-bit result)
Writing 64-bit result
Finish (Without writing 64-bit result)
Finish (Without writing 64-bit result and accumulation)

Fig. 3.6 7-stage multiplication sub-FSM.

The 40-bit product of the multiplier is calculated by the 32-bit multiplicand and bottom 8-bit of the multiplier. A normal multiply instruction without accumulation needs at least 2 cycles since a 40-bit product is obtained in the first one cycle and added into 64-bit adder to get final 64-bit result. While all 32-bit of multiplier is valid, four cycles are needed (8-bit once). In each cycle, the 64-bit adder result is added by the 40-bit product obtained from the preceding cycle. Therefore, a normal multiply instruction takes at most 5 cycles to finish the multiplication. However, a multiply with accumulate instruction takes one more cycle and writing 64-bit result out to two 32-bit registers also takes one more cycle. As a result, a 32-bit multiplicand multiplying a 32-bit multiplier with accumulation and writing a 64-bit result to 2 32-bit registers takes 7 cycles totally.

While the multi-cycle multiply operation is finished, a finish signal is sent from multiplication sub-FSM to main FSM to tell that the multi-cycle operation is finished and the next instruction may get executed to continue the program flow.

Chapter 4 Proposed SystemC Cycle-Accurate Instruction Set Simulator

This chapter describes the proposed cycle-accurate instruction set simulator. Section 4.1 depicts the block diagram. Section 4.2 to 4.5 describes the decoder, the instruction execution unit, the exception detector process, and the instruction cycle operations process of the proposed design.

4.1 Block Diagram

The proposed cycle-accurate ISS implementation is embedded in a SystemC wrapper. The proposed ISS consists of two method processes which include exception detector process and instruction cycle operations process, and three child functional units which include decoder, register file, and instruction execution unit. Fig. 4.1 shows the core diagram of the proposed implementation.

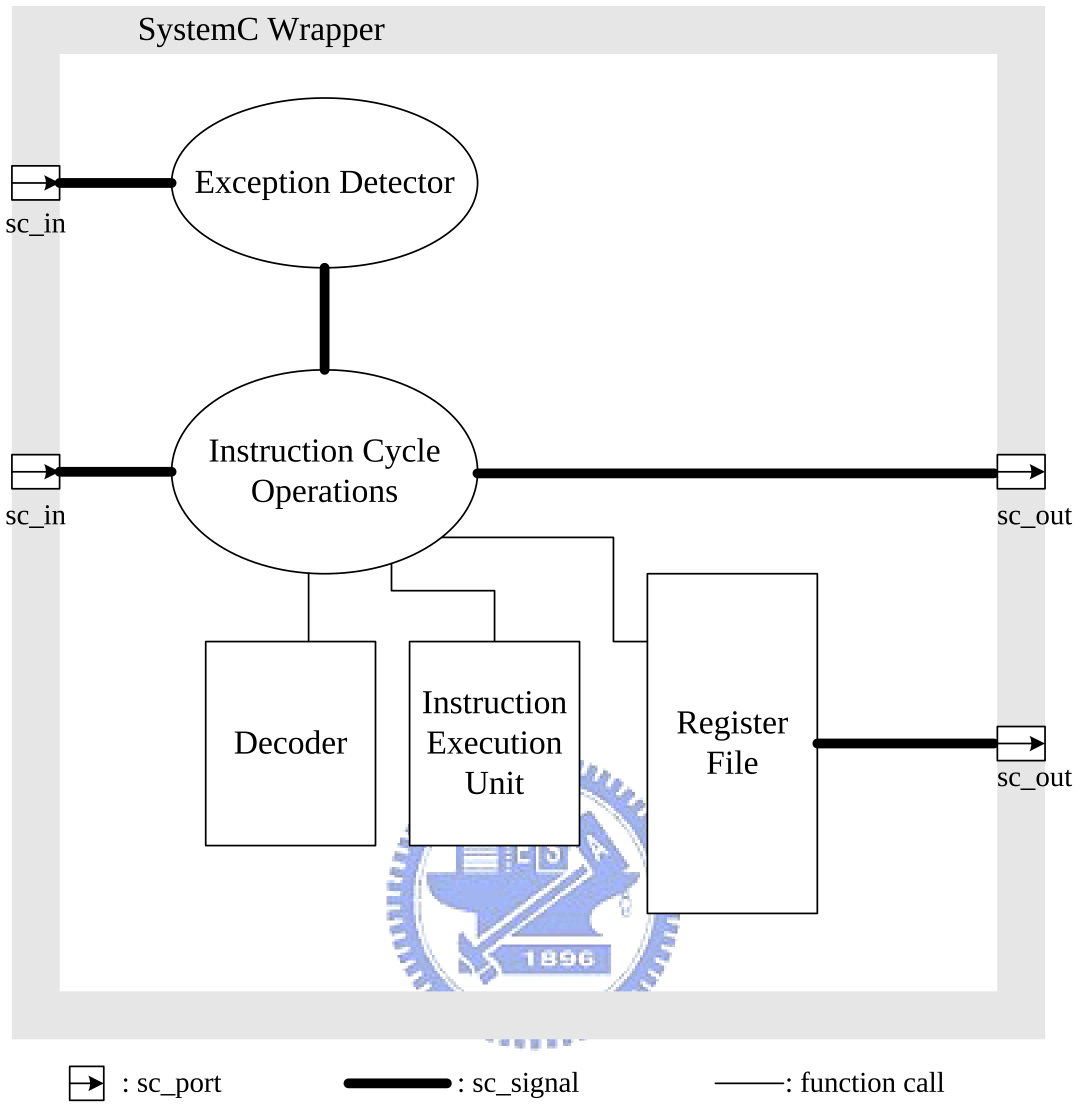


Fig.4.1 Core diagram of the SystemC ISS.

The SystemC wrapper contains all I/O ports of the ISS. The input ports are declared using `sc_in`, and the output ports are declared using `sc_out`.

The exception detector process (`excpt_detect_proc`) and the instruction cycle operations process (`cycle_op_proc`) are triggered by the negative edge of mclk and nreset signals. Then the instruction cycle operations process calls sub-routines such as decoder, instruction execution unit, and register file.

4.2 Decoder

The instructions of the ARM ISA are classified into 37 instruction types. Table 4.1 shows these instruction types with index values.

Table 4.1 Instruction type index.

Index	Instruction Type	Description
101	IT_BX	Branch and exchange
102	IT_MSR_REG	Transfer register contents to PSR
103	IT_MSR_REG_FLG	Transfer register contents to PSR flag bits only
104	IT_MRS	Transfer PSR Contents to a register
105	IT_SWP	Single data swap
106	IT_MUL	Multiply
107	IT_MLA	Multiply and accumulate
108	IT_MULL	Multiply long
109	IT_MLAL	Multiply and accumulate long
110	IT_HLDR_REGOFF	Lord halfword, register offset
111	IT_HLDR_REGOFF_BR	Lord halfword, register offset, dest = pc
112	IT_HSTR_REGOFF	Store halfword, register offset
113	IT_HLDR_IMMOFF	Lord halfword, immediate offset
114	IT_HLDR_IMMOFF_BR	Lord halfword, immediate offset, dest = pc
115	IT_HSTR_IMMOFF	Store halfword, immediate offset
116	IT_DP_REG_SHIFT	Data processing, shift by register
117	IT_DP_REG_SHIFT_BR	Data processing, shift by register, dest = pc
118	IT_DP_IMM_SHIFT	Data processing, shift by immediate
119	IT_DP_IMM_SHIFT_BR	Data processing, shift by immediate, dest = pc
120	IT_MSR_IMM_FLG	Transfer immediate value to PSR flag bits only
121	IT_DP_IMM	Data processing, immediate offset
122	IT_DP_IMM_BR	Data processing, immediate offset, dest = pc
123	IT_LDR_IMMOFF	Lord word, immediate offset
124	IT_LDR_IMMOFF_BR	Lord word, immediate offset, dest = pc
125	IT_STR_IMMOFF	Store word, immediate offset
126	IT_LDR_REGOFF	Lord word, register offset
127	IT_LDR_REGOFF_BR	Lord word, register offset, dest = pc
128	IT_STR_REGOFF	Store word, register offset

129	IT_UND	Undefined instruction
130	IT_LDM_1R	Lord Multiple, 1 register
131	IT_LDM_1R_BR	Lord Multiple, 1 register, dest = pc
132	IT_STM_1R	Store Multiple, 1 register
133	IT_LDM	Lord Multiple
134	IT_LDM_BR	Lord Multiple, dest = pc
135	IT_STM	Store Multiple
136	IT_B	Branch instruction
137	IT_SWI	Software interrupt

The decoder decodes the instruction into the corresponding instruction type and generates the information that the instruction execution unit and instruction cycle operations process need. Refer to Fig. 2.7, these information contain the register number of operands (Rn, Rm, Rs, Rd, RdHi, and RdLo), the opcode of data processing instructions, the shift type, and the immediate offset of instructions.

Besides, the decoder performs the condition test of the instruction. The instruction is only executed if the condition is true, or otherwise it is ignored and replaced with a NOP instruction.

4.3 Register File and Instruction Execution Unit

As shown in Fig. 2.5, the register file consists of 31 general-purpose registers and six status registers. However, not all registers are visible at the same time. According to the mode type signal (`acarm7_mode`), a set of pointers in the register file is used to point the proper register as the visible one. Then the instruction operands will point to the corresponding registers by the decoded register numbers. Fig. 4.2 shows an example.

```
switch(acarm7_mode)
{
  …
  case 0x12: //M[4:0] == 10010
    r0_ptr   = &r0_tmp;
    r1_ptr   = &r1_tmp;
    r2_ptr   = &r2_tmp;
    r3_ptr   = &r3_tmp;
    r4_ptr   = &r4_tmp;
    r5_ptr   = &r5_tmp;
    r6_ptr   = &r6_tmp;
    r7_ptr   = &r7_tmp;
    r8_ptr   = &r8_tmp;
    r9_ptr   = &r9_tmp;
    r10_ptr  = &r10_tmp;
    r11_ptr  = &r11_tmp;
    r12_ptr  = &r12_tmp;
    r13_ptr  = &r13irq_tmp;
    r14_ptr  = &r14irq_tmp;
    r15_ptr  = &pc;
    spsr_ptr = &spsrirq_tmp;
    break;
    …
}

switch(rd_num)
{
  …
  case 3 : rd = &r3_ptr;  break;
  case 4 : rd = &r4_ptr;  break;
  case 5 : rd = &r5_ptr;  break;
  …
}

switch(rn_num)
{
  …
  case 3 : rn = &r3_ptr;  break;
  case 4 : rn = &r4_ptr;  break;
  case 5 : rn = &r5_ptr;  break;
  …
}

switch(rm_num)
{
  …
  case 4 : rn = &r4_ptr;  break;
  case 5 : rn = &r5_ptr;  break;
  case 6 : rn = &r6_ptr;  break;
  …
}
```

Fig. 4.2 Code segment of SystemC ISS model.

An instruction, ADD r3, r4, r5, is executed in the IRQ mode. The mode bits of CPSR is M[4:0] == 10010. The value of rd_num, rn_num, and the rm_num is 3, 4, and 5 respectively. In the first switch-case statement, the IRQ mode registers are pointed. In the following three switch-case statement, the operand rd, rn, and rm points to the r3_ptr, r4_ptr, and r5_ptr respectively.

The instruction execution unit contains 10 sub routines to be called by cycle_op_proc and to perform the executed instruction. Table 4.2 shows these sub-routines.

Table 4.2 Functions of instruction execution unit.

Function	Description
nop	to perform the nop instruction
shifter	to perform the shift/rotate operation
alu	to perform the arithmetic and logical instruction
mac	to perform the multiply/multiply-and-accumulate instruction
sdt	to perform the single data transfer instruction
hdt	to perform the halfword data transfer instruction
bdt	to perform the block data transfer instruction
swp	to perform the swap instruction
bbx	to perform the branch, and branch and exchange instructions
excpt	to perform the exception operation

A multiple-cycle instruction may call the sub-routine more than once and a variable ex_count (execution cycle count) determines what operations should be done in that cycle. For a load halfword instruction, the first cycle, ex_count == 1, performs the address calculation; the second cycle, ex_count == 2, performs the base register modification; and the third cycle, ex_count == 3, performs the data fetch from memory. In other words, a small FSM exists in a multi-cycle instruction controlled by the ex_count signal.

4.4 Exception Detector Process

The exception detector process plays a role of a synchronizer and an exception detector.

When ISYNC is low, nFIQ and nIRQ inputs are considered asynchronous. A cycle delay for synchronization is incurred before the interrupt can affect the processor flow. After synchronization, the exception detector process detects whether prefetch abort, data abort, FIQ, and IRQ exceptions occur.

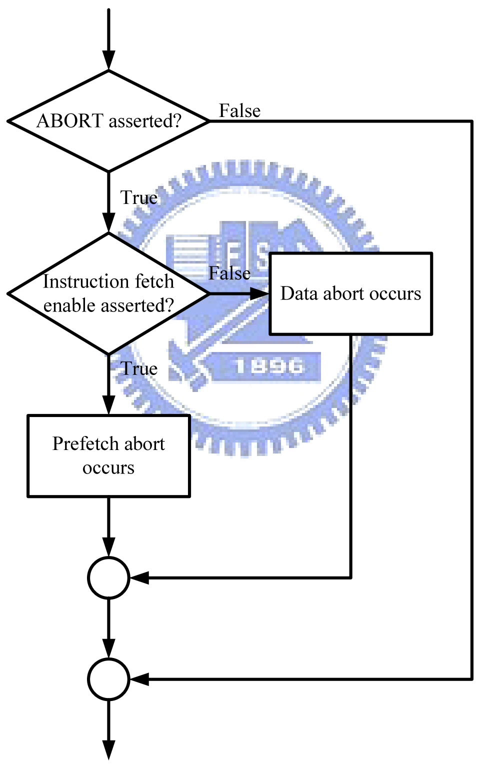


Fig. 4.3 Detection flow of abort exceptions.

Fig. 4.3 shows the abort exception detection flow. If the abort signal is high, the process checks the instruction fetch enable (`i_fetch_en`) signal. If the `i_fetch_en` is high, then a prefetch abort exception occurs, else a data abort exception occurs.

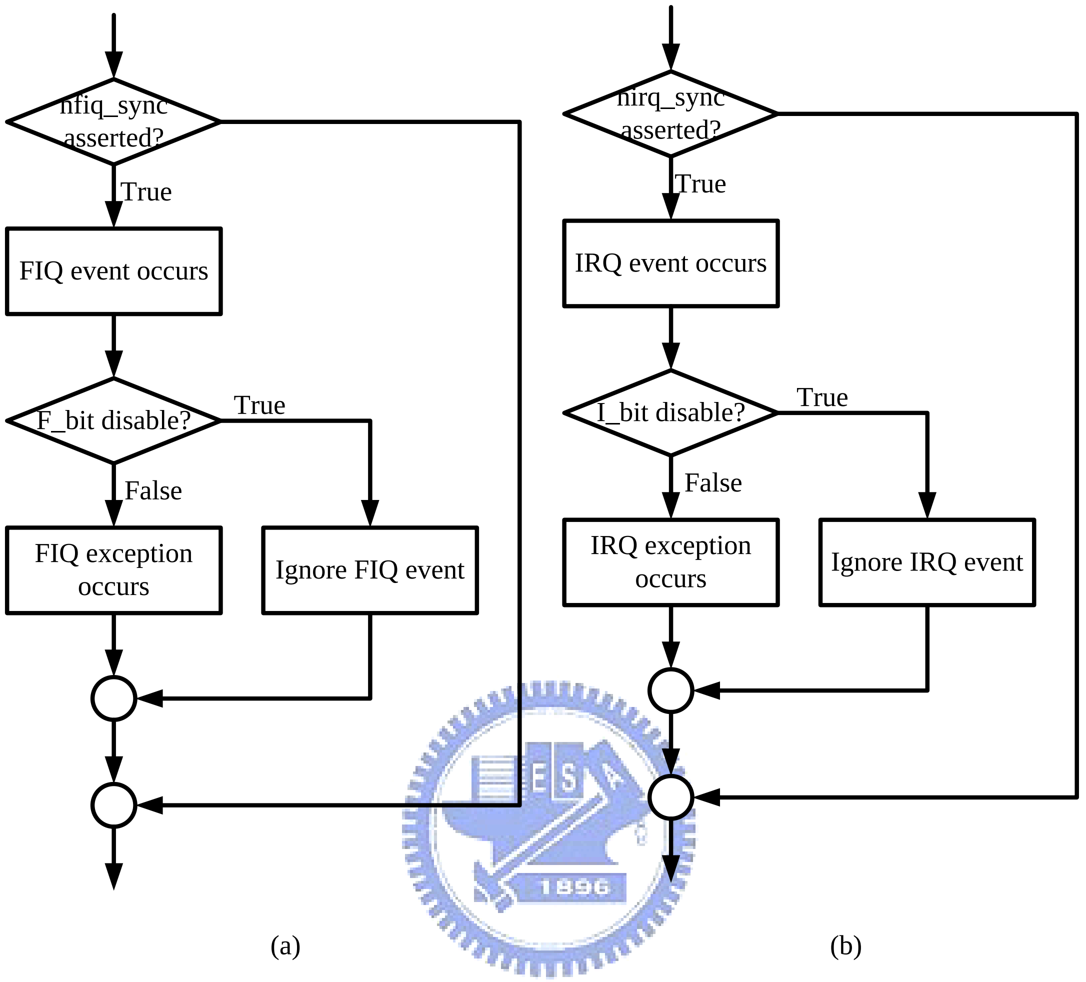
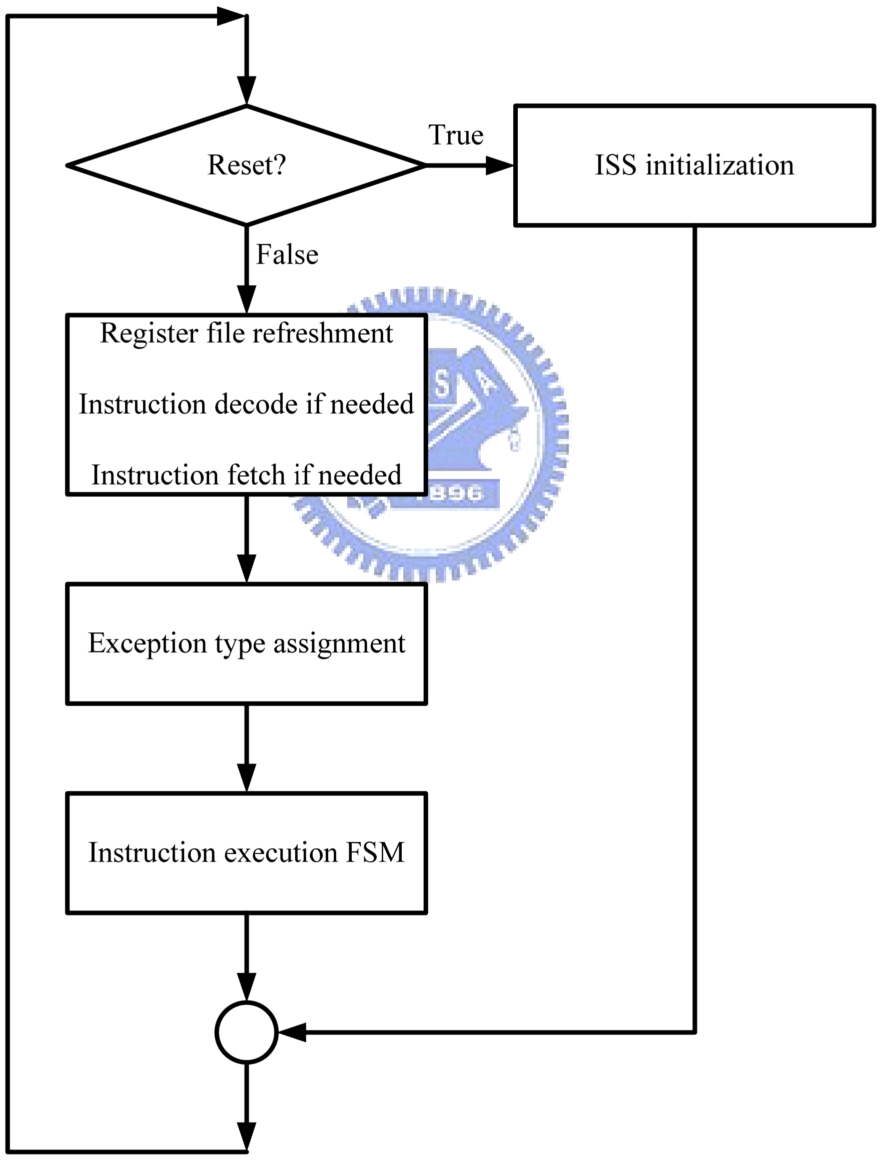


Fig. 4.4 Detection flow of (a) FIQ exception and (b) IRQ exceptions.

Fig. 4.4 (a) shows the detection flow of the FIQ exception. If the nfiq signal is low, an FIQ event can be marked. However, the process has to check the F-bit of CPSR. If the F-bit is high, the FIQ event is ignored and the exception does not occur in the ISS. The detection flow of the IRQ exception is similar to that of the FIQ one as shown in Fig. 4.4 (b).

4.5 Instruction Cycle Operations Process

The instruction cycle operations process is the main process of the ISS. The process contains the instruction execution FSM. The instruction type (`inst_type`) and the execution cycle count (`ex_count`) signals determine the state of FSM. Fig. 4.5 shows the process flow.



Reset?
True
ISS initialization
False
Register file refreshment
Instruction decode if needed
Instruction fetch if needed
Exception type assignment
Instruction execution FSM

Fig. 4.5 Flow of the instruction cycle operations process.

First, if the nRESET signal is asserted, the ISS initialization is performed, else the register file refreshment is performed. The instruction decode and instruction fetch are performed depend on whether the enable signals (`i_fetch_en` and `decode_en`) are asserted.

Then the exception type assignment is performed if any of the output signals of exception detector process is asserted. According to the type of exception, the instruction type and the mode type signals is set to the corresponding values and the proper set of registers is selected. According to the instruction type and execution cycle count signal, the instruction execution FSM calls the proper subroutine of the instruction execution unit and assigns the right values to the output signals of ISS.

Chapter 5
Verification Strategies

We use the simulation and co-simulation to verify the proposed designs. The commercial tools are available for SystemC and Verilog co-simulation. This thesis provides two kinds of verification strategies to ensure the functional correctness of the proposed designs. Section 5.1 describes the deterministic verification. Section 5.2 describes the constrained random verification.

5.1 Deterministic Verification

Deterministic verification is performed in the simulation phase and composed of two parts including the corner-case verification and the real application verification.

According to the datasheet, we analyze the corner cases of each kind of instructions, and write them into the verification patterns. We can discover the corner-case bugs by comparing the simulation results with the correct pre-calculated results.

After the corner-case verification, we start to perform the real application verification. The real application verification patterns consist of some benchmarks like Dhrystone, and DSPstone. Moreover, a JPEG encoder program is also used to verify the correctness of the proposed models. The real application verification helps the designer avoid misunderstanding the specification and is essential in deterministic verification phase.

After deterministic verification, the next verification stage, constrained random verification, can be started.

5.2 Constrained Random Verification

The constrained random verification is performed by the constraint-driven random pattern generator which generates random patterns to both ISS model and Verilog RTL model, and output values of both models are compared on-the-fly cycle by cycle. Fig. 5.1 shows the constrained random verification mechanism. This kind of verification is performed to detect the unanticipated cases and the random patterns are constrained to avoid generating undefined instructions. More simulation cycles are verified, larger vector space is explored by random patterns. Therefore, the design models can be more robust. Until now, over one billion cycles have been verified.

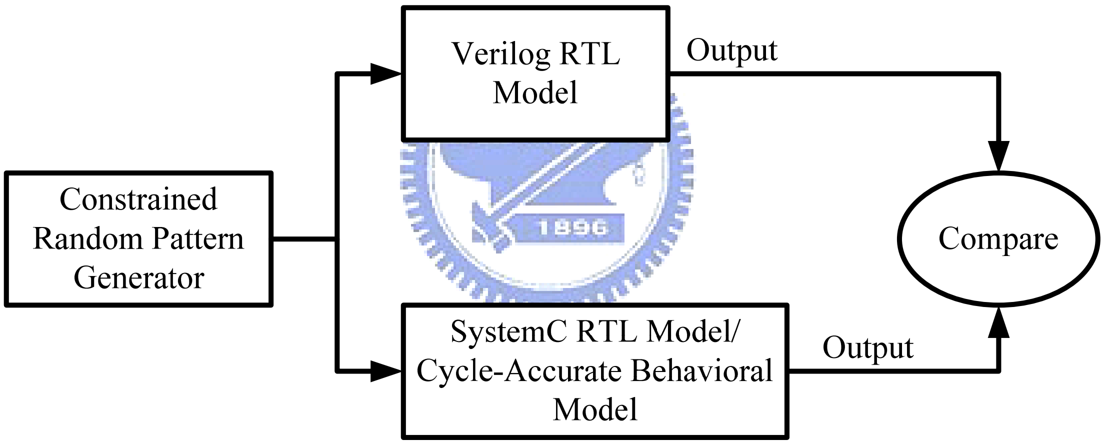

Verilog RTL Model
Output
Constrained Random Pattern Generator
Compare
SystemC RTL Model/ Cycle-Accurate Behavioral Model
Output

Fig. 5.1 Constrained random verification mechanism.

Chapter 6
Experimental Results

This chapter provides the experimental results of the proposed models. Section 6.1 describes the experimental environment. The experimental benchmarks are introduced in Section 6.2. Section 6.3 presents the experimental results, and followed by the discussion in Section 6.4.

6.1 Experimental Environment

We run all the experiments on a HP wx8400 workstation. The commercial simulators are the Cadence NC-Verilog simulator and the Cadence NC-SC simulator. For pure SystemC simulation, free tools are available from Open SystemC Initiative (OSCI). Table 6.1 shows the detail of the experimental hardware and software platform.

Table 6.1 Experimental environment.

Hareware
CPU: Intel® Xeon® CPU 2.0GHz
RAM: DDR2-667 ECC FB-DIMM 14GB
Software
OS: CentOS 5 x86_64 (with Linux 2.6 kernel)
Cadence NC-Verilog version 6.1
Cadence NC-SC version 6.1
OSCI SystemC version 2.2

6.2 Experimental Benchmarks

The experimental benchmarks are parts of MiBench [9]. MiBench is a free, commercially representative embedded benchmark suite. It consists of six categories including automotive, industrial control, network, security, consumer devices, office automation, and telecommunications. We choose one benchmark per category as shown in Table 6.2.

Table 6.2 Benchmarks of MiBench.

Auto./Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg	ghostscript	dijkstra	blowfish	CRC32
bitcount	lame	ispell	patricia	pgp	FFT
qsort	mad	rsynth		rijndael	IFFT
susan	tiff2bw	sphinx		sha	ADPCM
	tiff2rgba	stringsearch			GSM
	tiffdither				
	tiffmedian				
	typeset				

These six benchmarks, bitcount, jpeg, stringsearch, dijkstra, sha and CRC32, are described as follows:

bitcount: The bit count algorithm tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers.

jpeg encode/decode: JPEG is a standard, lossy compression image format. It is a representative algorithm for image compression and decompression and is commonly used to view images embedded in documents.

stringsearch: This benchmark searches for given words in phrases using a case insensitive comparison algorithm.

dijkstra: The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.

sha: *SHA* is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures. It is also used in the well-known MD4 and MD5 hashing functions.

CRC32: This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. The data input is the sound files from the ADPCM benchmark.

The simulation cycle counts of these selected benchmarks are from 2.8 million to 28 million as shown in Table 6.3.

Table 6.3 Cycle counts of selected benchmarks.

Benchmark	Simulation Cycle Counts
bitcount	9,087,665
jpeg encoder	2,893,310
jpeg decoder	8,796,271
stringsearch	2,916,981
dijkstra	12,075,954
sha	21,041,117
CRC32	28,148,849

6.3 Experimental Results and Discussions

We measure the simulation execution time of these benchmarks with models including ARM DSM, Verilog RTL, SystemC RTL, and SystemC ISS. The Verilog RTL model is applied to the NC-Verilog simulator. The ARM DSM, SystemC RTL, and SystemC ISS is applied to the NC-SC simulator. These results are presented in Table 6.4. Table 6.5 shows the simulation speed of other models as the normalized speed by setting the Verilog RTL speed as 1.

Table 6.4 Simulation execution time of proposed and reference models.

	ARM DSM @NC-Verilog	Verilog RTL @NC-Verilog	SystemC RTL @NC-SC	SystemC ISS @NC-SC	SystemC ISS @OSCI SystemC
bitcount	9941	240	500	202	23.56
jpeg encoder	1944	104	267	58	7.48
jpeg decoder	5833	263	543	220	22.51
stringsearch	1752	74	155	55	7.40
dijkstra	6732	292	622	245	30.75
sha	13967	582	1259	527	54.46
CRC32	18848	788	1682	706	71.73
Time Unit: sec					

Table 6.5 Simulation speed of proposed and reference models.

	ARM DSM @NC-Verilog	Verilog RTL @NC-Verilog	SystemC RTL @NC-SC	SystemC ISS @NC-SC	SystemC ISS @OSCI SystemC
bitcount	0.02	1	0.48	1.19	10.19
jpeg encoder	0.05	1	0.39	1.79	13.90
jpeg decoder	0.05	1	0.48	1.20	11.68
stringsearch	0.04	1	0.48	1.35	10.00
dijkstra	0.04	1	0.47	1.19	9.50
sha	0.04	1	0.46	1.10	10.69
CRC32	0.04	1	0.47	1.12	10.99

From the Table 6.5, the ranges of the simulation speed of the other models are from 0.02 to 13.90 times faster than the simulation speed of the Verilog RTL model.

The ARM DSM, a timing accurate model, has features of pin-to-pin delays and output delays for timing simulation so that the simulation speed is the slowest.

By comparing the simulation speed of the Verilog RTL and the SystemC RTL, the simulation speed of the SystemC RTL is only about half of the Verilog one. It indicates that although SystemC can acts as an HDL. However, the efficiency of RTL modeling of SystemC is worse than that of Verilog.

The SystemC ISS @NC-SC is exported as a Verilog module by using the

NCSC_MODULE_EXPORT macro [10]. The memory system of SystemC ISS @NC-SC is written in Verilog. On the other hand, the SystemC ISS @OSCI SystemC is simulated by the OSCI SystemC kernel. The memory system of SystemC ISS @OSCI is written in SystemC. The simulation speed of the former is about nine times faster than the simulation speed of the latter. Therefore, simulation environments affect simulation speed very much.

Finally, we can observe that the simulation speed of our proposed SystemC ISS is one order faster than that of Verilog RTL. The proposed SystemC ISS can get the best performance under the pure SystemC simulation environment.

Chapter 7
Conclusions and Future Works

In this thesis, two abstraction levels of processor modeling in SystemC are proposed. One is SystemC RTL model, and the other is SystemC behavioral ISS model. SystemC can act as an HDL language for RTL modeling, or can be a high level language like C++ with hardware-oriented class extension to implement higher abstraction level model. The experimental results regarding the performance comparison between SystemC ISS and Verilog RTL model in VV indicate that our ISS model is indeed efficient and suitable for co-simulation and co-verification.

In the future, we will focus on an efficient configurable SystemC architecture of cycle-accurate ISS implement. If two processors, e.g. ARM7TDMI and ARM9TDMI, adopt the same ISA, the functional model should the same. The difference between them is the instruction cycle operations since the number of pipeline stages is different. We will develop a configurable SystemC ISS including a configurable un-timed programmer view model plus two timed modules corresponding to the two processors to achieve this goal.

Reference

[1] http://www.esl-now.com/

[2] D. Black, J. Donovan, *SYSTEMC: FROM THE GROUND UP*, Kluwer Academic Publishers, 2004.

[3] Y. -J. Lu, et. al., "Microprocessor Modeling and Simulation with SystemC," in *VLSI-DAT 2007*, Apr. 2007, pp. 4-7.

[4] J. Bhasker, *A SystemC Primer*, 2nd ed., Star Galaxy Publishing, 2002.

[5] *ARM7TDMI datasheet*, Advanced RISC Machine Ltd., 1995.

[6] S. Furber, *ARM System-on Chip Architecture*, 2nd ed., Addison Wesley, 2000.

[7] Y. -C. Fong, "A High-Speed Area-Minimized Reconfigurable Adder Design," Master's thesis, National Chiao Tung University, Department of Electronics Engineering, Jul. 2006.

[8] H. -K. Ling, "A High-Performance Reconfigurable Sub-Word Parallel Multiplier-Accumulator Design," Master's thesis, National Chiao Tung University, Department of Electronics Engineering, Jul. 2006.

[9] M. Guthaus et. al., "MiBench: A free, commercially representative embedded benchmark suite", in *WWC-4. 2001 IEEE International Workshop on Workload Characterization*, Dec. 2001, pp. 3-14.

[10] NC-SC® Simulator User Guide, Cadence Design Systems, Inc., Apr. 2007, pp. 222-224.