

# 國立交通大學

電子工程學系 電子研究所  
碩士論文

極低功率且高效能之  
32 位元嵌入式處理器設計  
伴隨 JPEG 解碼器系統

**Ultra Low-Power and High-Performance  
32-Bit Embedded Processor  
With JPEG Decoder System**

研究生：許哲霖

指導教授：黃俊達 博士

中華民國九十六年九月

極低功率且高效能之  
32 位元嵌入式處理器設計  
伴隨 JPEG 解碼器系統

**Ultra Low-Power and High-Performance  
32-Bit Embedded Processor  
With JPEG Decoder System**

研究生：許哲霖

Student: Je-Ling Hsu

指導教授：黃俊達 博士

Advisor: Dr. Juinn-Dar Huang



電子工程學系 電子研究所

碩士論文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical & Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements for

the Degree of Master in Electronics Engineering & Institute of Electronics

August 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年九月

# 極低功率且高效能之 32 位元嵌入式處理器設計 伴隨 JPEG 解碼器系統

研究生：許哲霖

指導教授：黃俊達 博士

國立交通大學

電子工程學系 電子研究所



本論文提出一個極低功率且高效能之 32 位元嵌入式處理器 ACARM7(ACademic ARM7)的研究成果報告。其一，此一處理器有較佳的優異性能。此處理器是以 ARM V4 指令集實作，所以此指令集能使用 ADS(ARM Developer Suite)的編譯器將高階程式語言(C, C++)編譯成組合語言，再將其組譯為可供 ACARM7 使用的機器語言，以顯示出此處理器的高使用性。經過與原廠 ARM7TDMI 相比，此處理器不但所消耗的功率更少，所使用的邏輯閘更少，而且操作時脈更為快速。其二，本論文提出一套完整且嚴謹的驗證流程。此流程既能在近 10 億

個模擬週期的比對之下確保其功能行為之正確性，又能確保其合成為低階邏輯閘電路之後整個轉換正確性。其三，本處理器亦可供做 JPEG 解碼器系統。通過此驗證流程之後，此處理器之設計被燒錄在 FPGA 之上，再使用 ARM926EJ-S Versatile 發展板系統，以完成 JPEG 解碼器系統。

總之，本處理器無論在效能、面積、與功率等方面的比較都勝過 ARM7TDMI；此論文提供一套完整且嚴謹的處理器開發驗證流程，以確保更佳之正確性；本處理器還燒錄在 FPGA 之上供做 JPEG 解碼器系統，以展示其提供良好的應用性。



# **Ultra Low-Power and High-Performance 32-Bit Embedded Processor With JPEG Decoder System**

Student: Je-Ling Hsu

Advisor: Dr. Juinn-Dar Huang

Department of Electronics Engineering &  
Institute of Electronics  
National Chiao Tung University



This thesis presents the research result of an ultra low-power and high-performance 32-bit embedded processor with JPEG decoder system. This processor is named ACARM7 (ACademic ARM7). The ISA (Instruction Set Architecture) of ACARM7 adopts the ARM V4 architecture. Hence the ADS (ARM Develop Suite) can be directly used. ADS can first be used to compile the high level programming language (C, C++) written by users to the assembly language, and then can assemble the assemble language to the low level machine code for ACARM7 use. It indicates the high usability of ACARM7. Compared with ARM7TDMI, the power consumed by the proposed processor is lower; the gate-count of the proposed one is less; and the performance is better. Meanwhile, this thesis also provides a thorough and rigorous verification flow which assures both the correctness of functional

behavior of the proposed processor design after more than two billion simulation cycle comparisons and the synthesis correctness of synthesized gate-level netlist circuit. Moreover, the proposed processor is mapped onto the FPGA and integrated within the ARM926EJ-S Versatile Development Board to implement a JPEG decoder system. Based on the experiment result obtained by this research, the higher performance, the smaller area, and the lower power are all the advantages of the proposed processor compared with ARM7TDMI. The thesis also proposes a thorough and rigorous processor verification flow. Moreover, the high applicability of the proposed processor is demonstrated by mapping it into an FPGA for implementing a JPEG decoder system.



# ACKNOWLEDGMENT

## 感謝辭

能夠完成這篇論文，首先我要衷心感謝指導教授黃俊達老師。在整個要求高品質的研究階段中，黃老師總是提供卓越的指導、全力的支持、以及溫暖的鼓勵；他淵博的知識、豐富的經驗、正確的判斷、以及熱情的態度也在其平日諄諄教誨之下，對吾儕漸收「滴水穿石與潛移默化之效」；而黃老師這些做人與做事的特質的確都對我以及實驗室同學深具莫大的啟發與無窮的裨益。

我也要感謝本校 ACAR 實驗室的同學：南興、之暉、詠翔、威毓、瀚蔚、建德以及于翔等諸君，因為你們在各方面給我的協助與建議確實讓我這兩年的研究生涯過得既充實又有長足的進步。由於你們的共同陪伴，我有了更深層的體驗：美德與智慧不能獨立達到最高的境界，所以需要朋友的互助與推愛。

最後我要感謝父母、長輩、家人、與親戚；您的厚愛、照顧、與支持，為我過去、現在、與未來的人生增添絢麗的色彩！這些就是我此生幸福快樂的不歇之泉。我誠願將這篇論文獻給您們，以回報您的慈愛於萬一！

許哲霖 敬書於交大ACAR實驗室

# CONTENTS

ABSTRACT(Chinese).....	II
ABSTRACT(English) .....	IV
ACKNOWLEDGMENT .....	VI
CONTENTS .....	VII
LIST OF TABLES .....	X
LIST OF FIGURES .....	XI
CHAPTER 1 INTRODUCTION.....	12
CHAPTER 2 PREVIOUS WORKS.....	15
2.0 Overview .....	15
2.1 Prerequisites .....	15
2.1.0 Overview of Prerequisites .....	15
2.1.1 Core Architecture of ARM7TDMI .....	15
2.1.2 Pipeline Stage of ARM7TDMI .....	17
2.1.3 Instruction Set Architecture (ISA) of ARM7TDMI.....	19
2.2 Related Works .....	24
2.2.0 Overview of Related Works .....	24
2.2.1 Shift/Rotate Operation by Barrel-shifter.....	25
2.2.2 Mode-switch by Exceptions.....	26
2.3 Summaries of Previous Works.....	32
CHAPTER 3 PROPOSED CORE DESIGNS.....	34
3.0 Overview .....	34
3.1 Architecture of the Proposed Design.....	34
3.1.0 Overview of Architecture of the Proposed Design .....	34
3.1.1 Block Diagram of the Proposed Design .....	34
3.1.2 Control Logic of the Proposed Design.....	39
3.1.3 Arithmetic/ Logical Operation in EX-stage.....	41
3.1.4 Multi-cycle Multiplication in EX-stage.....	43
3.2 Implementation of Low-power Technique.....	44
3.2.0 Overview of Implementation of Low-power Technique .....	44
3.2.1 Unused Registers Gated .....	44
3.2.2 Unexecuted Function Units Gated.....	46
3.2.3 Low-power Consumption Property of Fong-adder .....	47
3.3 Summaries of Proposed Core Designs .....	48
CHAPTER 4 EXPERIMENTAL RESULTS.....	51
4.0 Overview .....	51
4.1 Implementation .....	51



4.2 Discussion of Experimental Results .....	52
4.2.0 Overview of Experimental Results .....	52
4.2.1 Comprehensive Comparison .....	52
4.2.2 Timing Comparison .....	53
4.2.3 Area Comparison .....	54
4.2.4 Power Comparison .....	56
4.2.5 Other Characteristics .....	57
4.3 Summaries of Experimental Results .....	58
CHAPTER 5 PROPOSED VERIFICATION STRATEGY .....	59
5.0 Overview .....	59
5.1 Implementation .....	59
5.2 Functional Verification .....	60
5.2.0 Overview of Functional Verification .....	60
5.2.1 Coding Style Checking by Linting Free .....	61
5.2.2 Deterministic Verification .....	61
5.2.3 Input-constrained Random Verification .....	62
5.2.4 Assertion Based Verification .....	63
5.3 Synthesized Netlist Verification .....	66
5.3.0 Overview of Synthesized Netlist Verification .....	66
5.3.1 Synthesis by Scripts .....	67
5.3.2 Logic Equivalence Checking (LEC) .....	68
5.3.3 Gate Level Simulation .....	69
5.3.4 Power Estimation by Prime-power .....	69
5.4 Test Plan and Testability Measurement .....	69
5.5 Summaries of Proposed Verification Strategy .....	70
CHAPTER 6 JPEG DECODER SYSTEM .....	71
6.0 Overview .....	71
6.1 Implementation .....	71
6.2 Architecture of System Level .....	72
6.2.0 Overview of Architecture of System Level .....	72
6.2.1 Major Components of the Proposed System .....	72
6.2.2 Control Flow of the Proposed System .....	73
6.3 Architecture of Hardware Level .....	74
6.3.0 Overview of Architecture of Hardware Level .....	74
6.3.1 AHB Peripherals in FPGA .....	74
6.3.2 Core Wrapped with AHB Bus Interface .....	76
6.3.3 Control Flow within AHB-Wrapper for ACARM7 in FPGA .....	79
6.4 Program Control Flow in Software Level .....	81

<b>6.5 Experimental Results of Proposed System</b> .....	82
<b>6.6 Summaries of JPEG Decoder System</b> .....	83
<b>CHAPTER 7 CONCLUSIONS</b> .....	85
<b>FUTURE WORKS</b> .....	87
<b>BIBLIOGRAPHY</b> .....	88



# LIST OF TABLES

Table2.1 Exception vector address .....	31
Table3.1 Transitions saved with Clock gated synthesized CG cell .....	46
Table3.2 PDP Analysis about Fong compared to Hybrid K-S Ling .....	48
Table3.3 Timing Analysis about Fong compared to Hybrid K-S Ling.....	48
Table3.4 Area Analysis about Fong compared to Hybrid K-S Ling.....	48
Table4.1 Simulation environment setup for experiments.....	51
Table4.2 Comprehensive comparison among 3 different cores .....	53
Table4.3 Performance comparison among 3 different cores.....	54
Table4.4 Area/Gate-count comparison among 3 different cores.....	55
Table4.5 Power comparison among 3 different cores .....	56
Table4.6 Power-Delay-Product comparison among 3 different cores.....	57
Table5.1 Verification environment setup .....	59
Table5.2 All of the bugs found in functional verification phase .....	64
Table5.3 Un-collapsed Stuck Fault Summary Report .....	69
Table6.1 Verification environment setup .....	72
Table6.2 Experimental results of FPGA .....	83



# LIST OF FIGURES

Fig.2.1 Block diagram of ARM7TDMI architecture .....	17
Fig.2.2 ARM7TDMI single-cycle instruction 3-stage pipeline operation .....	18
Fig.2.3 ARM7TDMI multi-cycle instruction 3-stage pipeline operation .....	18
Fig.2.4 ARM7TDMI instruction set encoding format .....	21
Fig.2.5 ARM7TDMI instruction set .....	22
Fig.2.6 ARM7TDMI instruction set (continued) .....	23
Fig.2.7 ARM shift/rotate mechanism .....	23
Fig.2.8 Condition code summary .....	24
Fig.2.9 A left-shifted example as N=32 .....	25
Fig.2.10 BS MUX Tree with 5 stages adding one final stage .....	26
Fig.2.11 Program status register format .....	30
Fig.2.12 ARM operating mode and register usage .....	30
Fig.2.13 Register organization in ARM state .....	31
Fig.3.1 ACARM7 core architecture diagram .....	35
Fig.3.2 Core architecture diagram in EX-stage .....	36
Fig.3.3 Mul 7-stage FSM diagram .....	37
Fig.3.4 Address Register source selection .....	37
Fig.3.5 Read/Write data selection .....	38
Fig.3.6 FSM of control logic .....	39
Fig.3.7 Detailed Core architecture diagram in EX-stage .....	42
Fig.3.8 Registers with Clock gated by RTL and synthesized CG cell .....	45
Fig.3.9 Input data gated in unexecuted units .....	47
Fig.5.1 Functional Verification flow .....	60
Fig.5.2 Input-constrained Random Verification .....	63
Fig.5.3 Assertion based verification .....	63
Fig.5.4 64x64.bmp .....	65
Fig.5.5 64x64.jpg .....	65
Fig.5.6 176x144.bmp .....	66
Fig.5.7 176x144.jpg .....	66
Fig.5.8 Synthesized Netlist Verification flow .....	67
Fig.6.1 Block diagram in system level .....	73
Fig.6.2 AHB peripherals configured in the FPGA .....	76
Fig.6.3 Core refinement with AHB interface FSMs .....	78
Fig.6.4 AHB Wrapper for ACARM7 (with ARM9EJ-S and SDRAM) .....	80
Fig.6.5 ZBTSRAM memory usage .....	80
Fig.6.6 Program control flow in software level .....	82

# CHAPTER 1

## INTRODUCTION

Nowadays, the number of digital consumer electric products, which includes Personal Digital Assistant (PDA), cell-phone, Playstation-Portable (PSP), Apple iPod, and so on, has grown up so drastically. Either embedded processors or Digital Signal Processors (DSP), whose power is supplied by the batteries, are included in all these portable electric products. Therefore, how to save more power of these portable electric devices is the most important subject in the competitive market. This thesis proposes the research result of an ultra-low power and high performance 32-bit embedded processor implemented by ARM v.4 ISA. This processor is such a quite convenient device due to the fact that the machine code fetched by the proposed one can be obtained by ARM Develop Suite (ADS). Small area of the proposed processor also saves more space in portable electric products which need characteristic of lightness, thinness, shortness, and smallness.

With the prevalent fashion of System on Chip (SOC), designs of digital electric products have become more complicated; therefore, such designs need a more rigorous and complete verification strategy to ensure both functional correctness of the design and the synthesis correctness of synthesized gate-level netlist circuit. This thesis provides a thorough and rigorous verification strategy including the coding-style check, deterministic verification, input-constrained random verification, and assertion-based verification so that this strategy can get functional verification phase passed. In addition, the synthesized netlist verification flow is composed of synthesis procedure by scripts, logic equivalence checking, gate-level simulation, and power estimation by prime-power. Checking the

coding-style of the design by Linting tool will avoid all kinds of simulation errors, unexpected latches, misunderstanding naming issues, the failure of DFT (Design for Test), and so on. In addition, deterministic verification is made to check all regular cases and special corner cases which should be confronted in simulation phase; on the other hand, input-constrained random verification is made to detect all of unexpected situations that the design may confront with; the more the number of simulation cycles, the less the number of bugs existing in the design; and then more robust design can be declared. Moreover, Assertion-based verification (also called property-based verification, PBV) is implemented in the core design to check out whether the functional behavior has been corresponded to the expectation of designers; although some bugs may not cause any simulation answer errors and may be ignored easily by designers, the PBV can definitely find those bugs out. On the other hand, synthesized netlist verification strategy assures that no error occurs in the synthesis procedure. The writing and refining of synthesis scripts is the first step to obtain a netlist with the characteristic of lower power, smaller area, and higher performance. Logic equivalence checking is the second step to verify the consistency of an RTL design as it synthesized into gate-level netlist. Gate-level simulation is the third step to execute to assure functional correctness after synthesis. Preliminary power estimation by prime-power is the last step of the verification flow. As a matter of fact, through the proposed rigorous and complete verification strategy it is guaranteed that the proposed embedded processor owns high quality of an IP (Intelligent Property) with the characteristic of bug-free robustness, high reusability, and convenient porting issue.

Considered only the core design in the hardware issue is not enough in a SOC era, more difficulties in system level will be confronted with. For instance, the proposed core will be wrapped with an AMBA (Advanced Microcontroller Bus

Architecture) wrapper to communicate with other IPs on the AMBA while the core is implemented in an ARM-based embedded system. Moreover, the application software has to be modified before while is used by an embedded system. Therefore, this thesis also provides a developed JPEG decoder system with the proposed core, and the high applicability can be proofed.

As summarized above, the higher performance, the smaller area, and the lower power are all the advantages of the proposed processor compared with ARM7TDMI. The thesis also proposes a thorough and rigorous verification flow to verify the correctness of the proposed core. Moreover, the high applicability of the proposed processor can be proved by configuring it into the FPGA for implementing a JPEG decoder system.

The remainder of this thesis is organized as follows: Chapter 2 concisely describes the previous works related to the proposed design. Chapter 3 presents the architecture of the proposed core and some low power techniques. Chapter 4 shows the experiment results of the proposed core compared to other cores. Chapter 5 provides a complete verification strategy to make the proposed core more robust. Chapter 6 presents a JPEG decoder system which is integrated with the proposed core design. Chapter 7 concludes this thesis. Future works and bibliography are also provided afterward.

# CHAPTER 2

## PREVIOUS WORKS

### 2.0 Overview

This chapter describes some previous works related to ARM7TDMI, which is a member of the Advanced RISC Machines (ARM) family of general purpose 32-bit embedded processor. Section 2.1 describes some prerequisites about ARM7TDMI, and Section 2.2 discusses about some related works. At last, Section 2.3 summarizes the previous works and architecture of the proposed core design ACARM7 with ultra-low power technique supported will be introduced in the next chapter.

### 2.1 Prerequisites



#### 2.1.0 Overview of Prerequisites

This section describes some prerequisites related to ARM7TDMI, which is a member of the Advanced RISC Machines (ARM) family of general purpose 32-bit embedded processor. Section 2.1.1 depicts basic architecture of the processor core. Section 2.1.2 mentions pipeline stage of it. Section 2.1.3 describes instruction set architecture (ISA) of ARM7TDMI. More related works about ARM7TDMI will be discussed in the Section 2.2.

#### 2.1.1 Core Architecture of ARM7TDMI

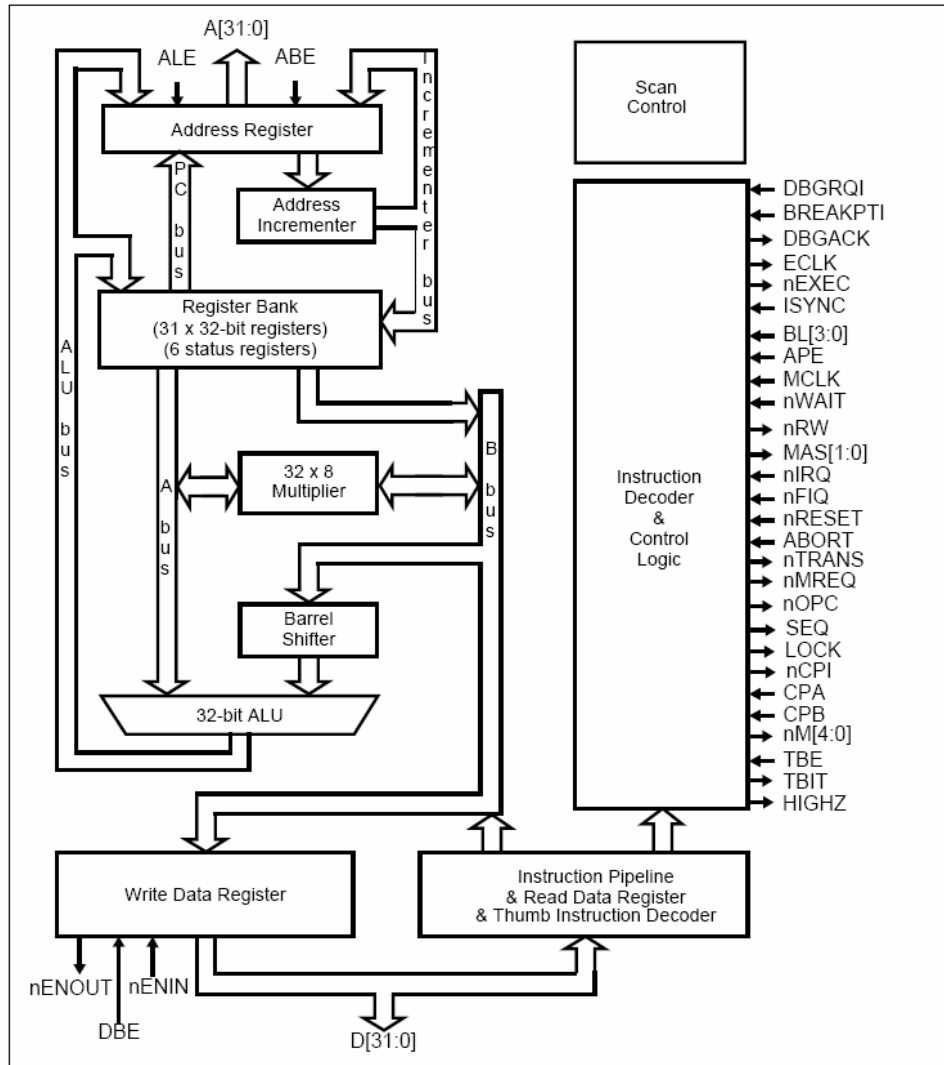
The ARM7TDMI is based on Reduced Instruction Set Computer (RISC) principles, and the ISA and related decode mechanism are much simpler than those



of microprogramming Complex Instruction Set Computers (CISCs). This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

The 3-stage pipeline ARM7TDMI organization is illustrated in Fig.2.1 [1]. It can be seen that two read ports and one write port of the register bank which access all registers. Additional read port and write port are added to access program counter (PC).

The Barrel-shifter, which manipulates the input data to do an operation like left-shifted, right-shifted arithmetic, right-shifted logical, and right-rotated and which operation should be done depending on the operation-code (OP-code). The ALU responds in all of the arithmetic and logical operations, while the address register with incrementer, which selects and holds all memory address and generate sequential address as required. On the other hand, data register holding data read from or written to memory with data alignment. The instruction decoder and associated control logic are also included in ARM7TDMI.



**Fig.2.1 Block diagram of ARM7TDMI architecture**

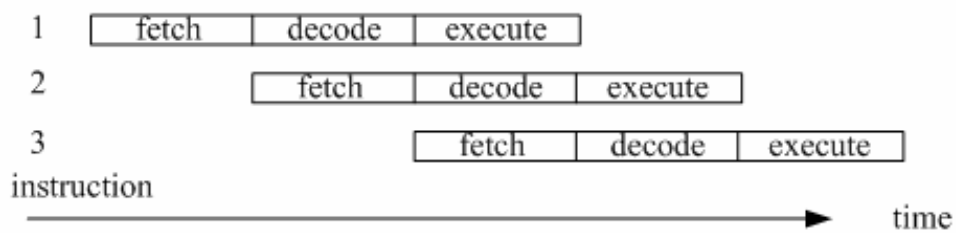
### 2.1.2 Pipeline Stage of ARM7TDMI

ARM7TDMI using a 3-stage pipeline structure includes fetch-stage, decode-stage, and execute-stage [2]. The first stage fetches instructions from memory and places them in the instruction pipeline. The decode-stage decodes the fetched instruction for next cycle execution. The last one of the 3-stage makes input data shifted or rotated if needed and executes them with ALU or multiplier and the processed data will write back the calculated result to the destination register finally.

At any time, each of the three stages may be occupied by three different

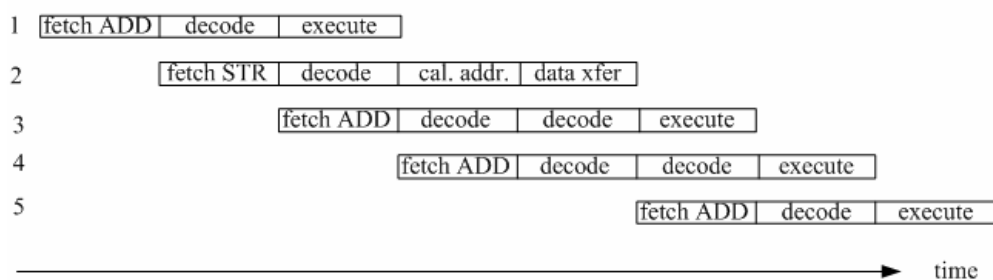
instructions; therefore, each stage of the hardware is capable handling the instruction within independently.

While a simple data processing instruction is executed by the processor, one instruction can be completed every clock cycle but with three-cycle latency; however, the throughput of the processor is one instruction per cycle and this kind of instructions is also called single-cycle instructions, as shown in the Fig.2.2.



**Fig.2.2 ARM7TDMI single-cycle instruction 3-stage pipeline operation**

In contrast, the execution flow of a multi-cycle instruction is less regular, for example, a sequential ADD instructions with a data store instruction STR, occurring after the first ADD instruction, as showed in Fig.2.3. The action of address calculation and the data transfer which both stay in the execution-stage take two cycles; as a result, the third instruction behind the STR stays in the decode-stage until the STR instruction completes the data transfer. Therefore, in this instruction sequence, all parts of the processor are activated in every cycle and the memory access behavior is the mainly limiting factor that limits the cycles taken by a sequential instructions.



**Fig.2.3 ARM7TDMI multi-cycle instruction 3-stage pipeline operation**

### 2.1.3 Instruction Set Architecture (ISA) of ARM7TDMI

This section describes the instruction set architecture of ARM7TDMI [1]. However, ISA of the proposed design is a version of ARM7TDMI adapted to only 32-bit instruction without Thumb and coprocessor instructions; therefore, only the 32-bit instruction set of ARM7TDMI will be focused in this section.

ISA of ARM7TDMI can be classified into 3 types, data processing instructions, data transfer instructions, and control flow instructions. The encoding formats can be seen in the Fig.2.4. Moreover, all instructions are listed in the Fig.2.5 and Fig.2.6.

Data processing instructions are the only one of 3 classified instruction set types listed above which enables to modify data values came from registers by executing arithmetic or logical operation on them. They typically require 2 operands and produce one single result. Such instructions include arithmetic operations, bit-wise logical operations, register movement operations, and comparison operations. ADD, ADC, and SUB are parts of the first while AND, OR, and XOR belong to the second. Register movement operations include MOV and MVN and comparison operations include CMP, CMN, TST, and so on. The Operand2 field of each instructions of data processing/PSR transfer includes 12 bits and has the ability to make operand2 shifted or rotated by a Barrel-shifter, as can be seen in the Fig.2.4. It shows shift/rotate operation mechanism in the Fig.2.7 and means all data processing instructions before arithmetic/logical operations can be shifted/rotated if needed. Moreover, the conditional codes of each instruction top 4-bit can be modified with any of the data processing instructions and more details about conditional codes will be discussed in the following paragraphs.

Data transfer instructions moving data between registers of ARM and memory and can be classified as 3 basic forms in the ARM instruction set. They are single

register load/store instructions, multiple register load/store instructions, and single register swap instructions. Single register load/store instructions, supporting all the byte transfer, half-word transfer, and word transfer, use most broadly to transfer data between registers and memory. Multiple register load/store instructions transfer data between memory and registers by word, less flexible than single data transfer instructions; however, being capable of large quantities of data to be transferred more efficiently, they are used for procedure entry and exit, to save and restore workspace registers, and to copy blocks of data around memory. Single register swap instructions allow a value in a register to be exchanged with a value in memory, effectively doing both load/store operation in one instruction, and they are little used in user-level programs.

Control flow instructions neither process data nor move it around; however, they simply determine which instructions get executed next. The processor normally executes instructions sequentially instead it reaches the branch instruction and proceeds directly to the destination address which the branch instruction comprehends. The mechanism used to control loop entry or exit is conditional branch which executes the instruction of branch only if the conditional codes of them get the correct value to fulfill some conditions.

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											<i>Data Processing / PSR Transfer</i>									
Cond	0	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	<i>Multiply</i>													
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm	<i>Multiply Long</i>														
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0	0	0	0	1	0	0	1	Rm	<i>Single Data Swap</i>											
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	<i>Branch and Exchange</i>								
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0	0	0	0	1	S	H	1	Rm	<i>Halfword Data Transfer: register offset</i>											
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset				1	S	H	1	Offset	<i>Halfword Data Transfer: immediate offset</i>											
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset											<i>Single Data Transfer</i>									
Cond	0	1	1																		1		<i>Undefined</i>								
Cond	1	0	0	P	U	S	W	L	Rn	Register List											<i>Block Data Transfer</i>										
Cond	1	0	1	L	Offset																	<i>Branch</i>									
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset										<i>Coprocessor Data Transfer</i>									
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP	0	CRm	<i>Coprocessor Data Operation</i>																
Cond	1	1	1	0	CP Opc				L	CRn	Rd	CP#	CP	1	CRm	<i>Coprocessor Register Transfer</i>															
Cond	1	1	1	1	Ignored by processor																	<i>Software Interrupt</i>									
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.2.4 ARM7TDMI instruction set encoding format**

Mnemonic	Instruction	Action
ADC	Add with carry	$Rd := Rn + Op2 + Carry$
ADD	Add	$Rd := Rn + Op2$
AND	AND	$Rd := Rn \text{ AND } Op2$
B	Branch	$R15 := \text{address}$
BIC	Bit Clear	$Rd := Rn \text{ AND NOT } Op2$
BL	Branch with Link	$R14 := R15, R15 := \text{address}$
BX	Branch and Exchange	$R15 := Rn,$ $T \text{ bit} := Rn[0]$
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	$CPSR \text{ flags} := Rn + Op2$
CMP	Compare	$CPSR \text{ flags} := Rn - Op2$
EOR	Exclusive OR	$Rd := (Rn \text{ AND NOT } Op2)$ $\text{OR } (Op2 \text{ AND NOT } Rn)$
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	$Rd := (\text{address})$
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	$Rd := 0xFFFFFFFF \text{ EOR } Op2$
ORR	OR	$Rd := Rn \text{ OR } Op2$

**Fig.2.5 ARM7TDMI instruction set**

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

Fig.2.6 ARM7TDMI instruction set (continued)

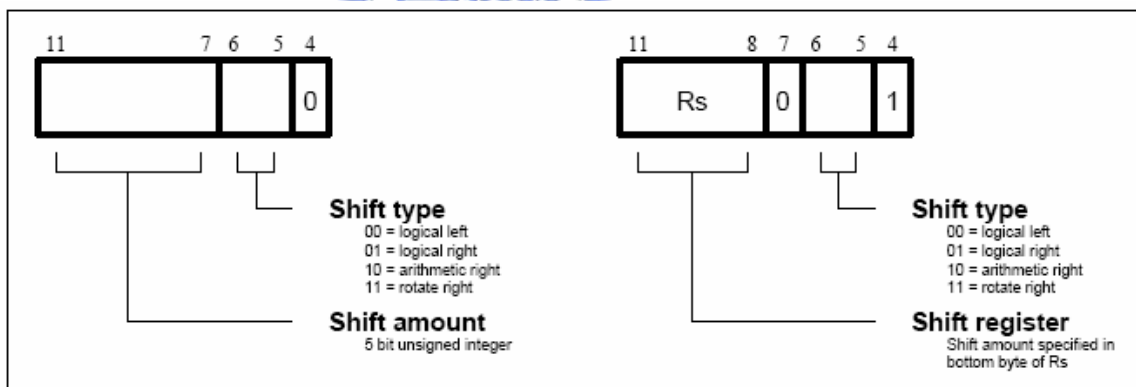


Fig.2.7 ARM shift/rotate mechanism

As mentioned earlier, one unique key-feature of the ARM instruction set is that every instruction is conditionally executed which implemented by top 4-bit conditional field of each 32-bit instruction field. Each of 15 values (instead of code equals to '1111', NV) of the conditional field causes the instruction to be executed or skipped according to the value of the N, Z, C and V flags in the CPSR. The 15 conditions are given in fig.2.8 and every ARM instruction mnemonic may be



extended by appending the 2 letters defined here. For example, ‘AL’ means ‘always’ condition which may be omitted since the instruction is definitely executed no matter what, the execution condition of others are listed in the Fig.2.8.

<b>Code</b>	<b>Suffix</b>	<b>Flags</b>	<b>Meaning</b>
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

**Fig.2.8 Condition code summary**

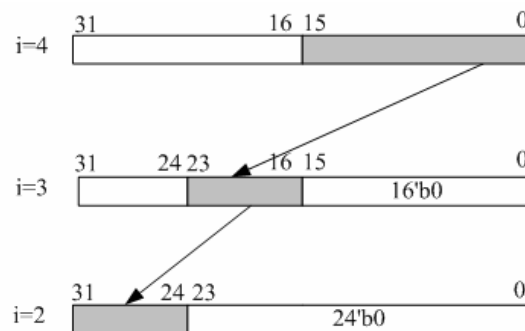
## **2.2 Related Works**

### **2.2.0 Overview of Related Works**

Prerequisites have been discussed in last section, and this section describes some related works about ARM7TDMI. Section 2.2.1 depicts shift/rotate operation by Barrel-shifter (BS). Section 2.2.2 depicts Mode-switch by Interrupt Mechanism of ARM7TDMI. Summaries of Chapter 2 Previous Works will be discussed in

### 2.2.1 Shift/Rotate Operation by Barrel-shifter

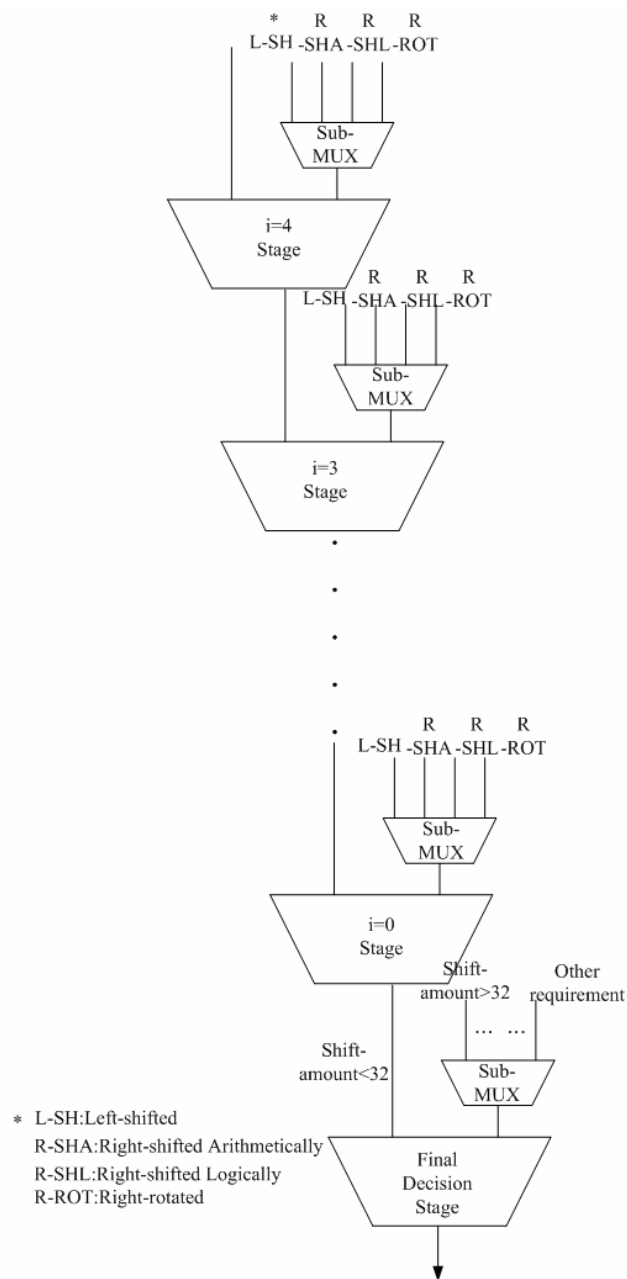
Shift and rotate operation is one of the most essential and basic part in data processing phase of a microprocessor and it presents an operation that a data is left-shifted, right-shifted arithmetically, right-shifted logically, and right-rotated. It is well known that in term of design style this kind of operation is usually implemented by a technique named Barrel-shifter, one kind of logarithmic shifter. This shifter completes the operation in  $\log_2(N)$  stages, where  $N=2^n$  is the word length. The moved amount of data in each stage is  $2^i$ , where  $0 \leq i \leq (n-1)$ , and the data shifted by the preceding stage shifts continuously in the present stage; hence, it takes  $n$  stages to obtain the final result. A left-shifted operation with  $N=32$  example can be shown in the Fig.2.9. The lower data from bit0 to bit15 can be moved to higher 16-bit and filled with zero in lower 16-bit in  $i=4$  stage; as the same as  $i=4$  stage, the 8-bit from bit16 to bit23 can be moved higher 8-bit and filled with zero in lower 8-bit in  $i=3$  stage.



**Fig.2.9 A left-shifted example as  $N=32$**

In each stage, one of four types of shift/rotate operation which include left-shifted, right-shifted arithmetically, right-shifted logically, and right-rotated is selected. More hardware detections about conditional codes and greater than 32-bit shift-amount should be added in the final stage of BS MUX tree, as shown in the

Fig.2.10.



**Fig.2.10 BS MUX Tree with 5 stages adding one final stage**

## 2.2.2 Mode-switch by Exceptions

Most programs operate in user mode, however, other privileged operating modes which are used to handle exceptions and supervisor calls are switched by

different interrupt types.

The current operating mode is defined by bottom 5-bit M0 to M4 of the CPSR register, as shown in the Fig.2.11. In addition, the interpretation of these bits is summarized in Fig.2.12 and the privileged mode has associated with it a Saved Program Status Register (SPSR), which is used to save the state of the CPSR while the privileged mode is entered in order that the user state can be fully restored as the user process is resumed.

Exceptions are generally used to handle unexpected events like interrupts or memory faults which arise during the execution of a program. Besides, exceptions is also used to cover software interrupt, undefined instruction traps, and the system reset function which logically arises before rather than during the execution of a program. These events are all grouped under the exception handling since they have all the same mechanism within the processor. Therefore, ARM exception can be divided into 3 groups, which include directly generated by executing instructions, generated as a side-effect of executing instructions, and generated externally, unrelated to the instruction flow. Software interrupt, undefined instructions and prefetch aborts belong to the first group; data aborts belongs to the second group; reset, IRQ, (normal interrupt) and FIQ (fast interrupt) belong to the third group.

Exception entry which is entered while an exception arises and it caused by a side-effect like data abort or an external event like IRQ uses the next instruction in the current sequence; however, direct-effect exceptions like software interrupt are handled in sequence as they arise. In fact, the processor performs the actions of following sequence as exceptions occur.

- It changes to the operating mode corresponding to the particular exception.
- It saves the address of the instruction following the exception entry instruction in r14 of the new mode.

- It saves the old value of the CPSR in the SPSR of the new mode.
- It disables IRQs by setting bit7 of the CPSR while it disables further fast interrupt by setting bit6 of the CPSR if the exception is a fast interrupt.
- It forces the PC to begin executing as the relevant vector address given by Table2.1.

As can be seen in the Table.2.1, each of exceptions has the corresponded vector address and contains a branch instruction to jump to the corresponded interrupt service routine (ISR) which includes all the actions should be done while an interrupt occurs; however, the vector address of FIQ needs no branch actions but starts its ISR immediately since it occupies the highest vector address.

The two banked registers in each privileged mode, r13\_x and r14\_x, a stack pointer which may be used to save other user registers and a return address holder, can be used by ISR in each privileged mode; however, FIQ mode has more additional private registers to give better performance by avoiding the need to save user registers in most cases where it is used, as shown in the Fig.2.13.

Once the exception has been handled the user task is normally resumed and some following facts need to be guaranteed.

- Any modified user registers must be restored from the stack of ISR.
- The CPSR must be restored from the appropriate SPSR.
- The PC must be changed back to relevant instruction address in the user instruction stream.

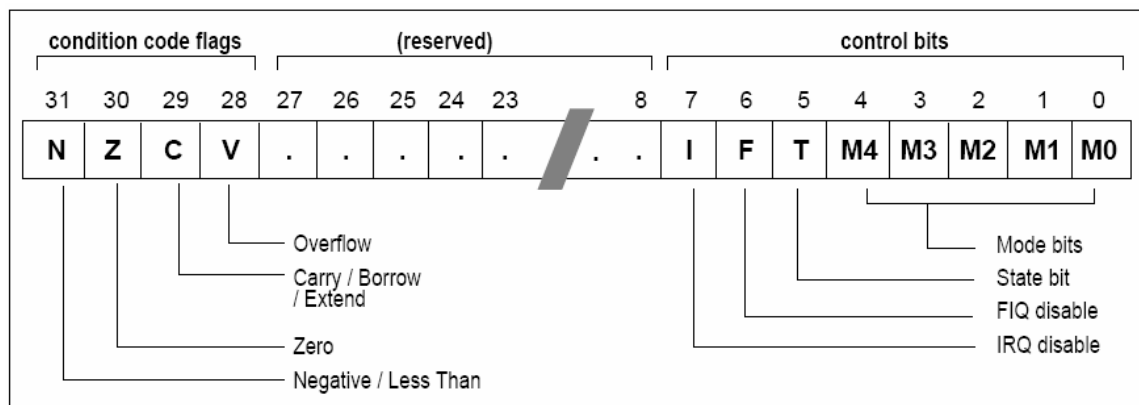
To note that the last two steps above need to be carried out together.

It is necessary to define a priority order to determine the order in which exceptions should be handled first since multiple exceptions could arise at the same time. These are priorities of all interrupts of ARM:

1. reset (highest priority);

2. data abort;
3. FIQ;
4. IRQ;
5. prefetch abort;
6. Software interrupt (SWI) and undefined instructions; these two interrupts are mutually exclusive instruction encodings and have no chance to occur simultaneously.

Reset starts the processor from a known state and renders all other pending exceptions irrelevant. The most complex exception scenario is where an FIQ, an IRQ and a third exception (excluding reset) occurs simultaneously. FIQ has higher priority than IRQ and also mask it out, so the IRQ will be ignored until the FIQ handler explicitly enables IRQ or return to user code. However, as the third exception is a data abort, the processor will enter the data abort handler and immediately enter FIQ handler, since data abort entry does not mask FIQ out. The data abort is remembered in the return path and will be processed when the FIQ handler returns. On the other hand, as the third exception is not a data abort, the FIQ handler will be entered immediately. While FIQ and IRQ have both completed, the program returns to the instruction which generated the third exception, and in all the remaining cases the exception will recur and be handled accordingly.



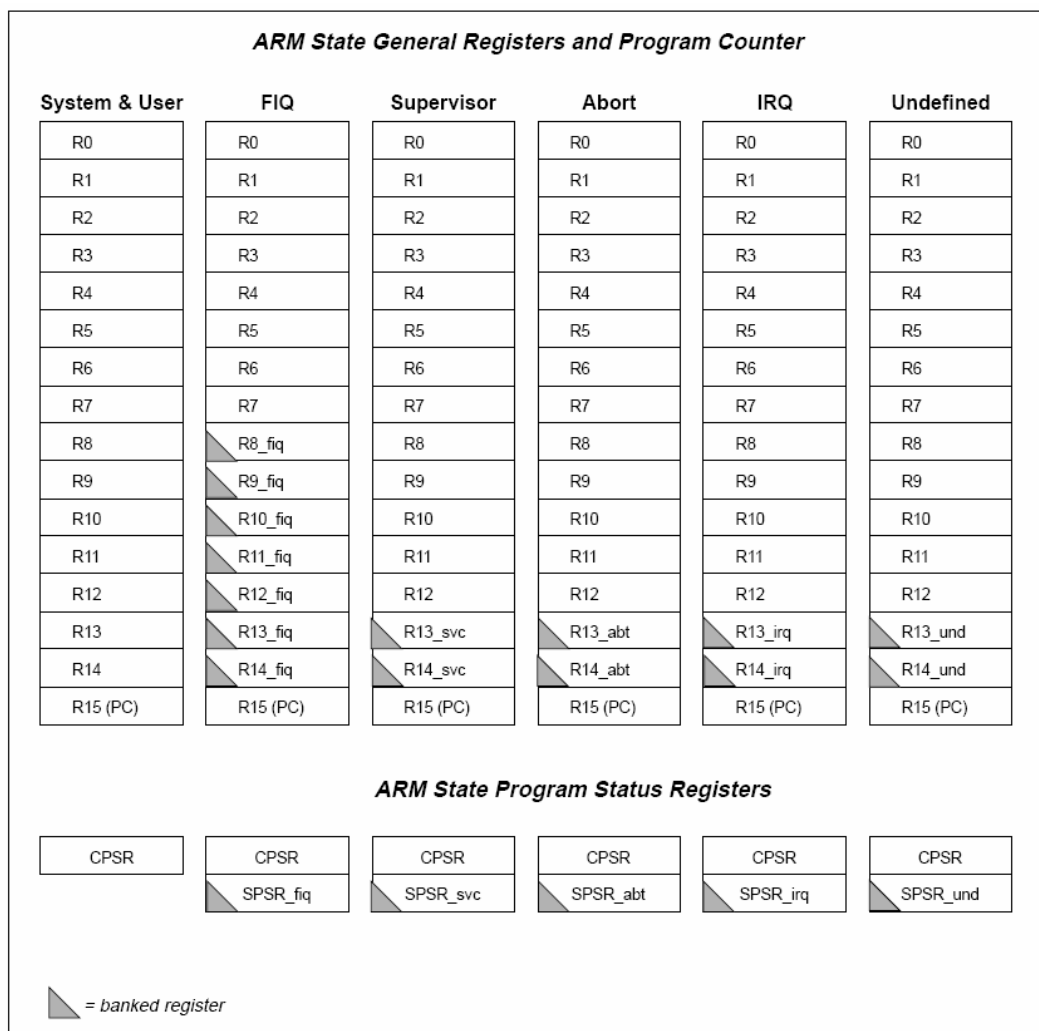
**Fig.2.11 Program status register format**

<b>M[4:0]</b>	<b>Mode</b>	<b>Visible THUMB state registers</b>	<b>Visible ARM state registers</b>
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq..R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc..R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt..R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und..R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

**Fig.2.12 ARM operating mode and register usage**

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	<i>Reserved</i>	<i>Reserved</i>
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

**Table2.1 Exception vector address**



**Fig.2.13 Register organization in ARM state**



## 2.3 Summaries of Previous Works

This chapter describes some previous works about ARM7TDMI processor design which includes core architecture, pipeline stage, and instruction set architecture and some related works like shift/rotate operation by a Barrel-shifter and mode switch by exceptions.

Core architecture of ARM7TDMI is designed by obeying Reduced Instruction Set Computer (RISC) principles with 3-stage pipeline implementation which includes fetch-stage, decode-stage, and execution-stage. Each of the three stages may be executed the instruction within independently and efficiently, therefore, the throughput of the processor is one instruction per cycle in an ideal case. The ARM7TDMI instruction set including only 32-bit instruction without Thumb is discussed in the section for the proposed design is a 32-bit embedded processor. Moreover, the ISA of ARM7TDMI can be classified into data processing instructions, data transfer instructions, and control flow instructions, and with condition code in the top 4-bit in every instruction to execute instructions conditionally.

Shift and rotate operation is one of the most essential and basic part in data processing phase of a microprocessor and is usually implemented by a technique named Barrel-shifter, which completes such a operation within  $\log_2(N)$  stages, where  $N=2^n$  is the word length. On the other hand, programs operate not only in user mode, but also other privileged operating modes which are used to handle exceptions and supervisor calls are switched by different interrupt types. All exceptions has its own vector address to the corresponded ISR and all of user registers must be recovered as exception handling is over and comes back to user programs.

Some previous works have been discussed and some important characteristics

have been reminded in this chapter; consequently, the proposed design ACARM7 improves the operating performance, makes power consumption as low as it possible, and uses fewer gate-counts to save area consuming while utilizing these previous works discussed this section. The proposed design will be described in detail in the next section.



# CHAPTER 3

## PROPOSED CORE DESIGNS

### 3.0 Overview

This chapter describes the architecture of the proposed core design and implementation of low-power technique. Section 3.1 depicts basic architecture of the proposed core design. Section 3.2 discusses the implementation of low-power technique. Section 3.3 summarizes the proposed core design and experiment results of the proposed core will be presented in the next chapter.

### 3.1 Architecture of the Proposed Design

#### 3.1.0 Overview of Architecture of the Proposed Design

This section describes architecture of the proposed design, ACARM7, a 32-bit embedded processor with characteristics of ultra low-power, high-performance, and low area consuming. Section 3.1.1 depicts the block diagram of the proposed design; Section 3.1.2 depicts control logic of the proposed design; Section 3.1.3 depicts how an arithmetic/logical operation is functioned in EX-stage; and Section 3.1.4 describes the mechanism of multi-cycle multiplication in EX-stage. The implementation of low-power techniques will be discussed in the Section 3.2.

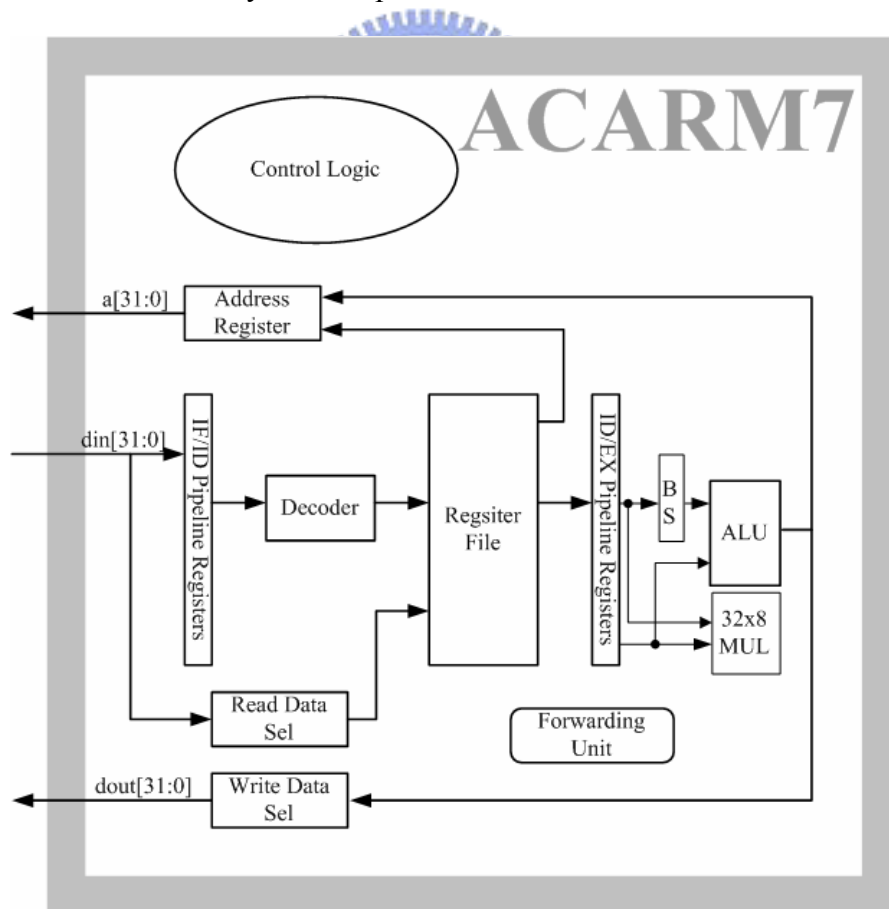
#### 3.1.1 Block Diagram of the Proposed Design

The proposed core architecture composed of 8 major functional blocks which include decoder, register-file, ALU, 32x8 Multiplier, forwarding unit, address

register, read/write data selection, and control logic. The block diagram of the proposed design can be seen in the Fig.3.1.

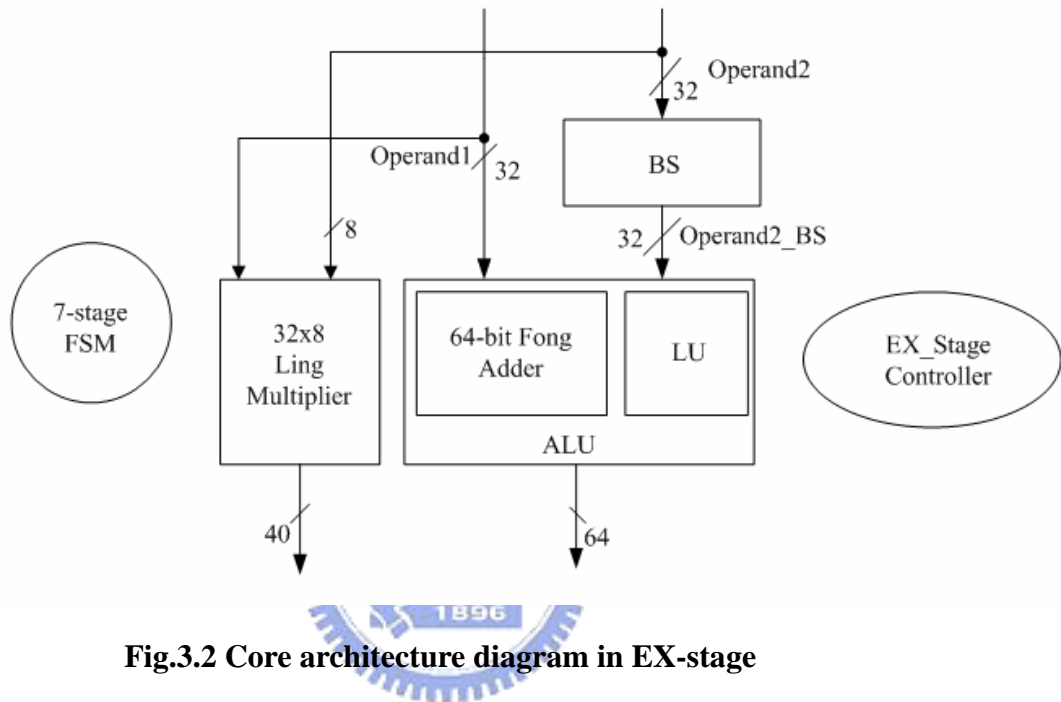
The decoder unit obtains the instruction from fetch phase and decodes it into all information that other function units need.

The register-file is composed of 31 general purpose 32-bit registers which can be divided into 6 banks and 6 status registers. The 6 banks of 31 general purpose registers can be used in 6 different modes of ACARM7 execution states including system&user mode, FIQ mode, supervisor mode, abort mode, IRQ mode, and undefined mode. As shown in the Fig.2.11, Each of 6 status-registers which record conditional codes and some control bits including modes bits, state bit, FIQ bit , and IRQ bit and it is not necessary to be implemented in 32 bits.



**Fig.3.1 ACARM7 core architecture diagram**

The execution stage includes Barrel-shifter (BS), 64 bits Fong adder [3], and a Ling multiplier [4] with 32-bit multiplicand multiplying 8-bit multiplier. , as can be shown in Fig.3.2, both arithmetic operation and logical operation are implemented with an ALU module whose two inputs, operand1 and operand2, are received from a BS to perform a shift operation if needed.



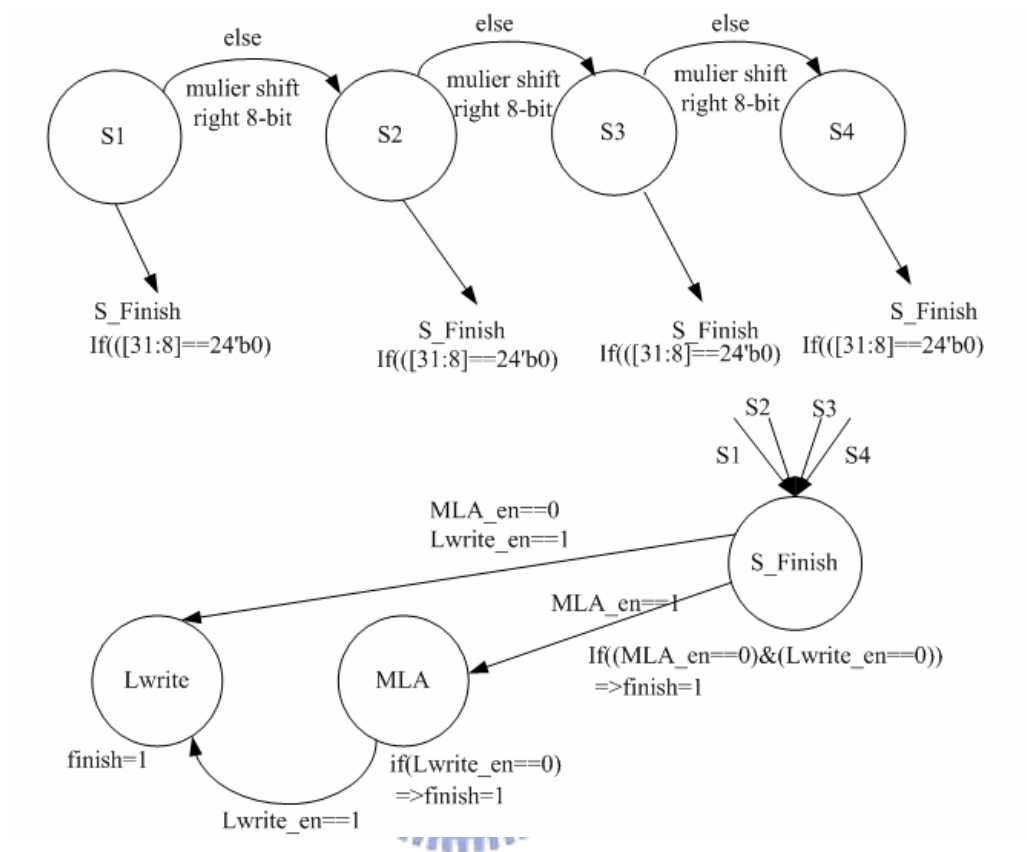
**Fig.3.2 Core architecture diagram in EX-stage**

The 40-bit product of the Ling-multiplier is calculated by the two operand inputs with 32-bit operand1 and bottom 8-bit operand2. The cycles of finishing a multiplication from 2 to 5 and can be decided by the EX\_stage Controller and a 7-stage FSM (Finite State Machine) in Fig.3.3. More details will be discussed in Section 3.1.3 Multi-cycle Multiplication in EX-stage. In an addition, the low-power technique is also considered to gate inputs of unexecuted function units and will be described in detail in the Section 3.2.

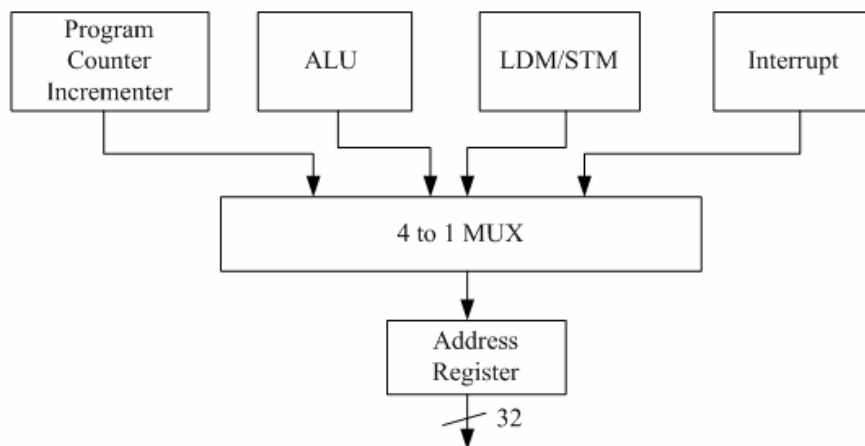
The forwarding unit forwards the data calculated from the output of the execution stage to make next instruction obtain the answer as soon as possible if needed, as shown in Fig.3.1.

One valid address is choosing from four address sources which include

program counter incrementer, ALU output, LDM (Load Multiple)/STM (Store Multiple) output, and source of interrupt. Consequently, the valid address will be sent to the address register (program counter, PC), as can be shown in Fig.3.4

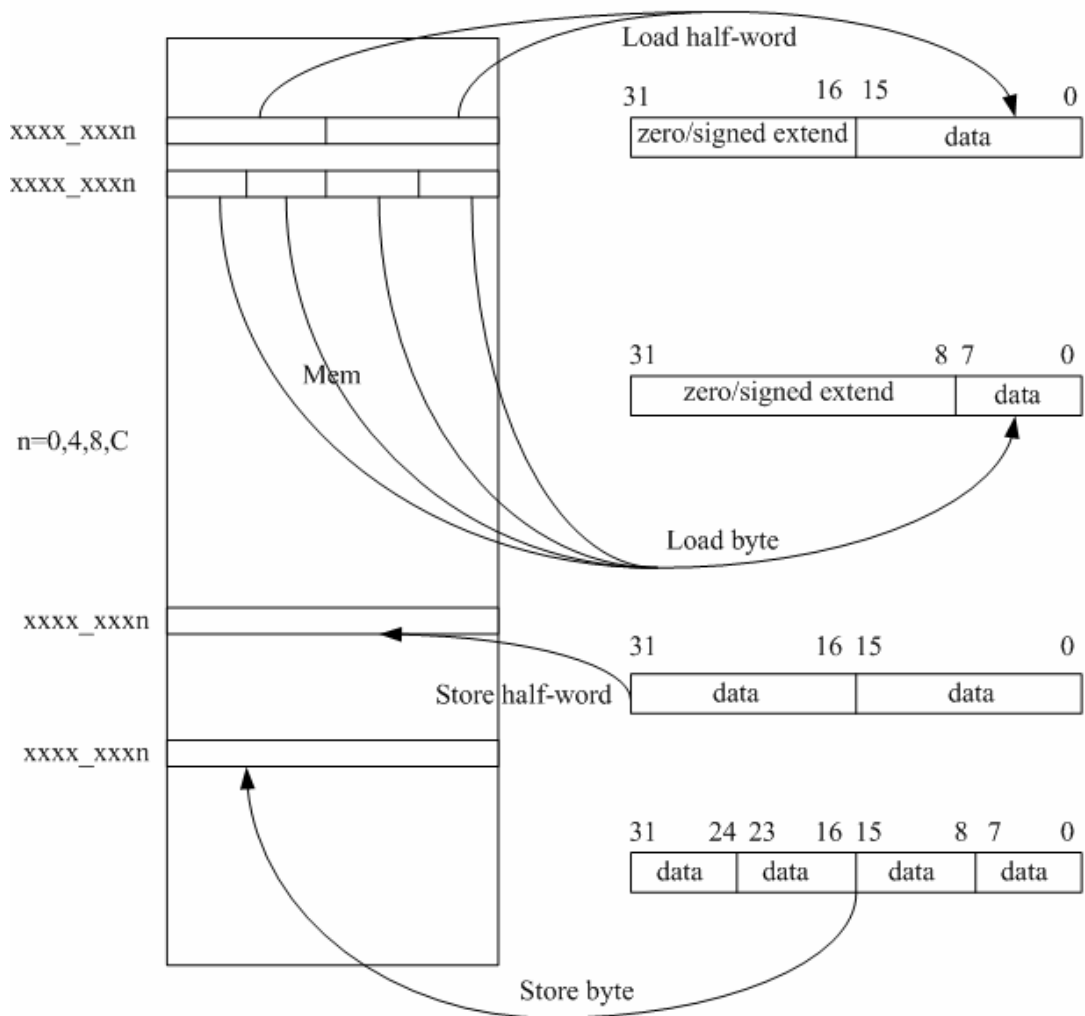


**Fig.3.3 Mul 7-stage FSM diagram**



**Fig.3.4 Address Register source selection**

The function performed by read/write data selection is non-word data aligned either read data from a memory or write data to memory in Fig.3.5. As a result, byte data or half-word data read from a memory will be shifted to bottom of a 32-bit width register and either zero-extended or sign-extended will also be performed to complete rest bits to fill with 32-bit data width. On the other hand, byte data written to a memory will be copied to 4 pieces to fill with 32-bit data width and half-data will be copied to 2 pieces as can be predicted. On the other hand, the discussion of control logic implementation of the proposed design will be presented in Section 3.1.2 Control Logic of the Proposed Design.



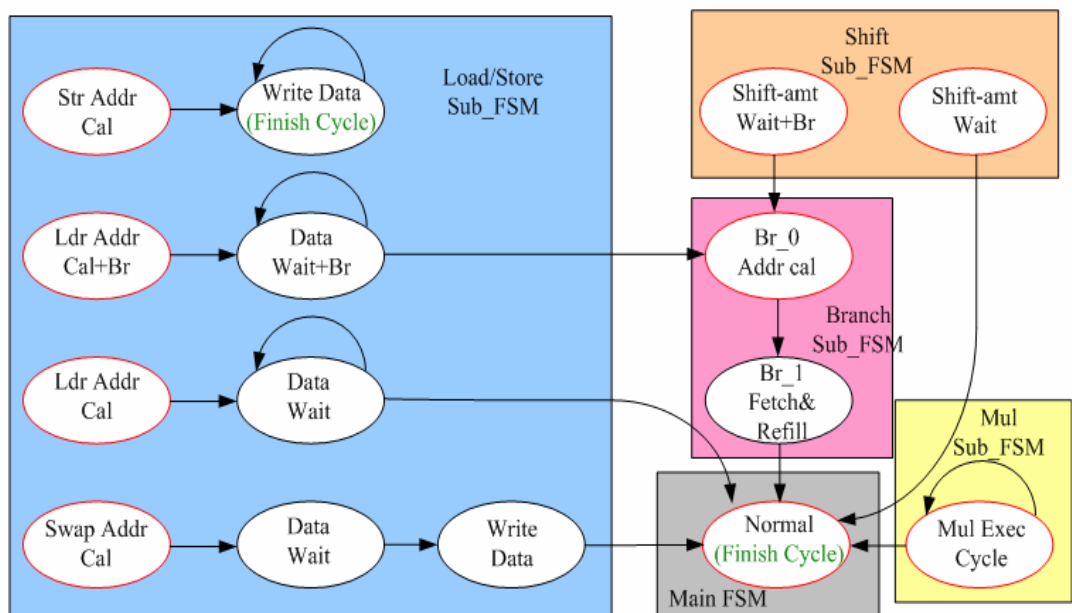
**Fig.3.5 Read/Write data selection**

The function of each primary block unit of the proposed core design has been

described individually in this section and mechanisms and implementation of control logic, arithmetic/logical operation in EX-stage, and multi-cycle multiplication operation in EX-stage, will be discussed in detail in the following sections.

### 3.1.2 Control Logic of the Proposed Design

The control logic controls all the data flows of combinational logic and all the state transitions of sequential logic. As can be seen in the Fig.3.6, four Sub-FSMs, load/store sub-FSM, shift sub-FSM, multiplication sub-FSM, and branch sub-FSM, are controlled by one main normal operation FSM. The main normal operation FSM includes only one state and all of the single-cycle instructions can finish their execution in this state; for instance, general arithmetic or logical operation. However, other instructions can not finish their execution within just one cycle, and they will leave for other 4 sub-FSMs.



**Fig.3.6 FSM of control logic**

Load/Store sub-FSM includes four types of instructions, store, load without branch, load with branch, and swap. Store instructions include single data store and



multiple data store, as can be seen in the Fig.3.6. The first state calculates the destination address and then the second state writes the data to memory and finishes the single data store instruction. While multiple data store instruction continuously stays in the second state until all data transfers are done. Besides store instructions, all other instructions will finish their last execution cycle backing to the normal main state. Load is also composed of single data load and multiple data load like store instructions. The load address is calculated in the first cycle, and waits data from memory in the second state. Single data load instructions will finish their execution in the normal main state; on the other hand, multiple data load instructions back to main state until all of their data transfers finished. In contrast, the cycle operation of load with branch is the same as load without branch in the first two cycles; the difference between them is the former will leave for branch sub-FSM which will be discussed later to continue its execution. Swap instruction makes data exchange from a register and external memory, and implemented by a lock operation of store after load. Therefore, it combines load and store instruction operations together; the changing order of states is address calculation, data wait from memory, write data to memory, and backing normal main state finally.

Shift sub-FSM includes two types of instructions, shift with branch and shift without branch, and each of them occurs as ALU needs a shift-amount not coming from immediate field but contents stored in registers. More cycle is needed since shift-amount stored in registers can not be obtained with the other two operands at the same time. Therefore, this kind of instruction will lead FSM to jump to the shift sub-FSM while the shift-amount stored in the r15 (PC) will lead FSM to jump to the shift sub-FSM with branch.

Multiplication sub-FSM includes all kinds of instructions about multiplication, and lasts from 2 cycles to most 7 cycles until the instruction finishes the execution.

The finish signal will be sent to the main control logic to be acknowledged and backed to main normal FSM. More details about multiplication will be discussed in the Section 3.1.4 Multi-cycle Multiplication in EX-stage later.

As in the branch sub-FSM, a new PC address is calculated in the first state and next new instruction will be fetched in the pipeline to fill the empty space behind the present instruction in the second state. It has the lowest priority of all the five FSMs and can be entered from load/store sub-FSM and shift-sub FSM and both two sub-FSMs and Mul sub-FSM are mutually exclusive. Moreover, the four other sub-FSMs instead of branch sub-FSM have equal priority and which FSM should be entered decided by instruction decoded information.

### **3.1.3 Arithmetic/ Logical Operation in EX-stage**

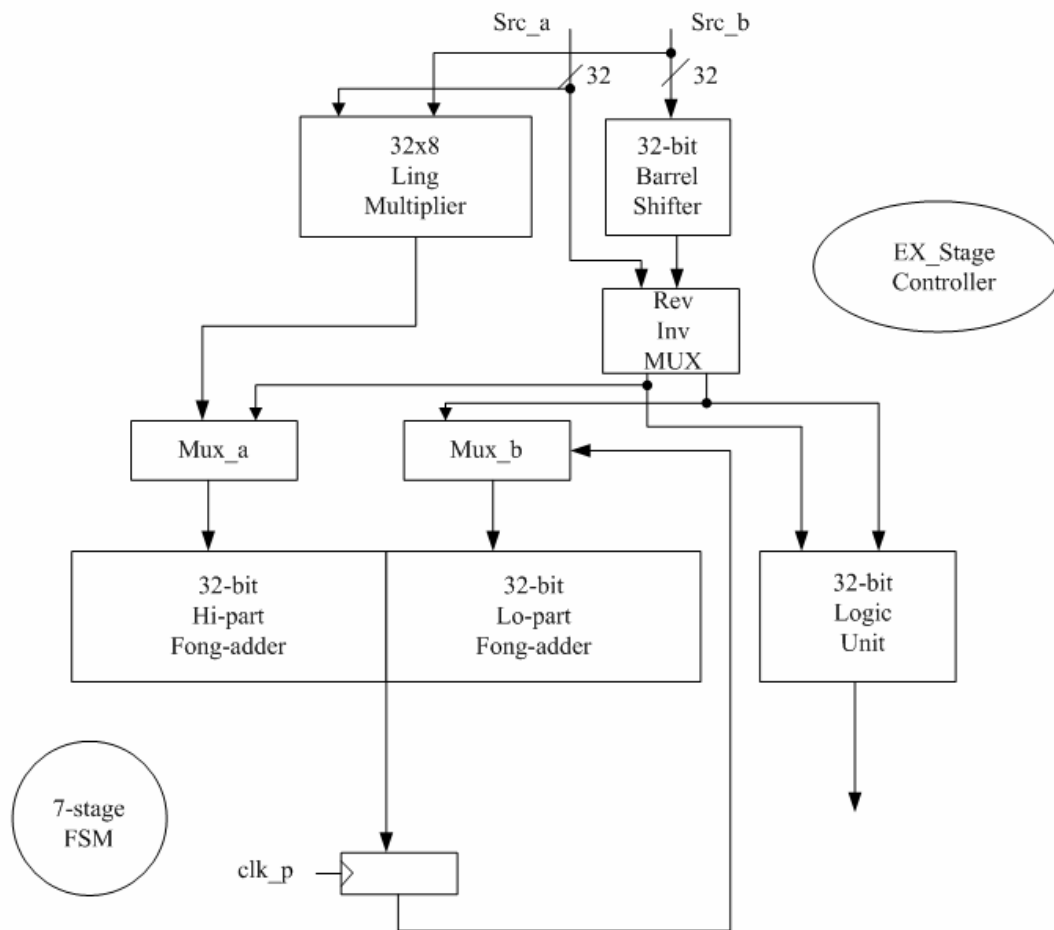
Arithmetic or logical operation is handled in the execution stage by ALU, as can be seen in the Fig.3.7, a detailed description with block diagram to Fig.3.2. Two data inputs, Src\_a and Src\_b, can be used for 2 data inputs of arithmetic/ logical operation instruction or 2 data inputs of multi-cycle multiplication in the pipeline stage Ex-stage; the later will be discussed in Section 3.1.3.

Src\_b is fed into a BS which has been discussed in Chapter 2 before and can be shifted or rotated by any amounts with any types including left-shifted, right-shifted arithmetically, right-shifted logically, and right-rotated according to an EX-stage controller. Both output signals from the BS and Src\_a are fed into a Reverse-Inverse-Multiplexer which decides that whether the 2 inputs enter this multiplexer should be inversed due to consideration about subtraction or reversed for specific instructions RSB or RSC discussed in Chapter 2.

The 2 results of the multiplexer are also sent to a logical unit while a logical

operation is decoded or a 64-bit Fong adder [3] while an arithmetic operation is decoded. All the logical instructions like AND, ORR, and so on will be handled in this logical unit; moreover, 4 conditional flags, as shown in fig.2.11, excluding V-flag will also be set at the same time.

On the other hand, all the arithmetic instructions like ADD, SUB, ADC, and so on. will be handled in the higher 32-bit part of 64-bit Fong-adder since normal addition or subtraction needs only 32-bit data width and lower 32-bit part will be filled with zero as no multiplication instructions are executed.



**Fig.3.7 Detailed Core architecture diagram in EX-stage**

### 3.1.4 Multi-cycle Multiplication in EX-stage

As the instruction is decoded a multiplication type, Mux\_a and Mux\_b of the Fig.3.7 will select the sources directly from a 32x8 Ling-Multiplier [4] and 64-bit result saved in a register calculated last cycle. (While in the first cycle operation, the 64-bit result is zero in default) These 2 resources will use all 64-bit of the Fong-adder [3] since a 32-bit multiplicand multiplies a 32-bit multiplier obtains most 64-bit product.

As can be seen in the Fig.3.3, normal multiplication without accumulation operation and writing 64-bit result out needs at least 2 cycles since 40-bit product is obtained in the first one cycle and added into 64-bit Fong adder to get final 64-bit result. While all 32-bit of multiplier is valid, 4 cycles is needed for 8-bit is calculated once. In each cycle, the 64-bit Fong adder result is added by the 40-bit product obtained from the preceding cycle. Therefore, normal multiplication takes most 5 cycles to finish the multiplication. However, multiplication with accumulation operation takes one more cycle and writes Long-result of 64-bit out to two 32-bit registers also takes one more cycle; as a result, a 32-bit multiplicand multiplies a 32-bit multiplier with accumulation and writing Long-result of 64-bit out to 2 registers takes 7 cycles in total.

While the multi-cycle multiplication finishes, a finish signal will be sent from Mul Sub-FSM to main FSM to tell that the multi-cycle operation is finished and may take next instruction into EX-stage to continue the program flow.

## **3.2 Implementation of Low-power Technique**

### **3.2.0 Overview of Implementation of Low-power Technique**

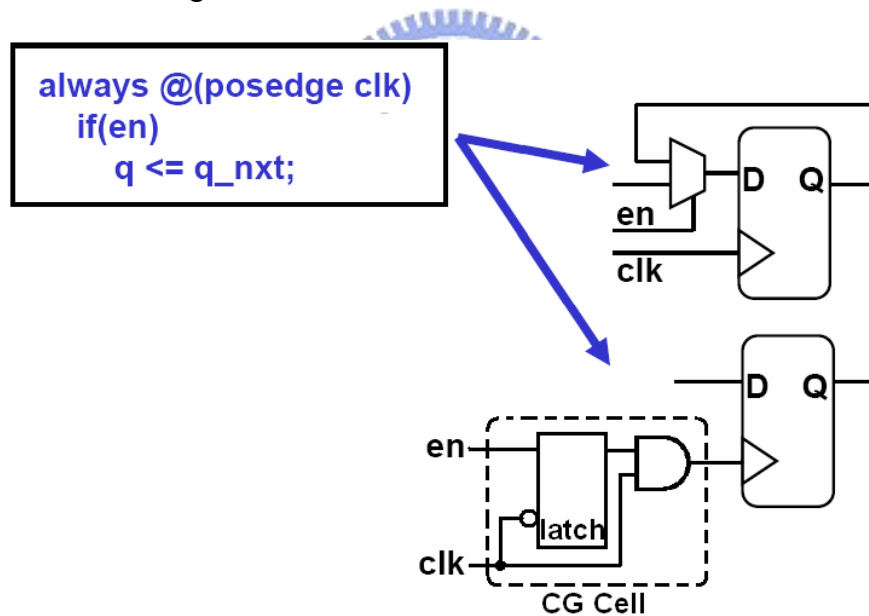
The proposed design is convinced to have a characteristic of ultra low-power and is implemented with three significant methods which are composed of unused registers gated, unexecuted function units gated, and low-power consumption property of Fong-adder [3]. The first will be discussed in Section 3.2.1 Unused Registers Gated; the second will be discussed in Section 3.2.2 Unexecuted Function Units Gated; and the third will be discussed in Section 3.2.3 Low-power Consumption Property of Fong-adder.

#### **3.2.1 Unused Registers Gated**

The data stored in Registers are updated in every clock cycle without any clock gating operation; as a result, huge power is consumed and some manipulations must be done to avoid so much power wasting of the proposed design.

One manipulation that makes new data to transmit into D port of the FLIP-FLOP only when the enable signal is high can be implemented in RTL level, as can be seen in the Fig.3.8. The disadvantage is that the stored data of registers is still re-written for holding old data every clock; as a matter of fact, it is still power wasting and other manipulations should be done. As can be seen in the Fig.3.8, a lower part with a CG cell composed of an enable signal controlled latch will make clock of the unused registers gated and guarantee that no data updated any more for holding old value. This method can absolutely shut down updating operations of unused registers and save unnecessary power consumption as much as it can. This synthesis method by using scripts will be discussed in detail in Chapter 5 Verification Strategy.

The numerical report shows that the number of FLIP-FLOPs could be gated is 1,576 and the number of FLIP-FLOPs could not be gated is 27; in addition, the benchmark we used is Dhrystone 2.1 with 8,332 clock cycles, as can be seen in the Table3.1. The amazing thing is that while clock gated method is implemented, only 796,517 transitions occurs. It compares to 13,131,232 transitions occurring without implementing gating technique and 93.93% unnecessary power consumption caused by FLIP-FLOP transitions will be saved. Even considering the FLIP-FLOPs could not be gated, it still saves power consumption about 92.35%. This report is so significant and credible that why our proposed design is called ultra low-power. Moreover, other methods about implementation of low-power technique will be introduced in following sections.



**Fig.3.8 Registers with Clock gated by RTL and synthesized CG cell**

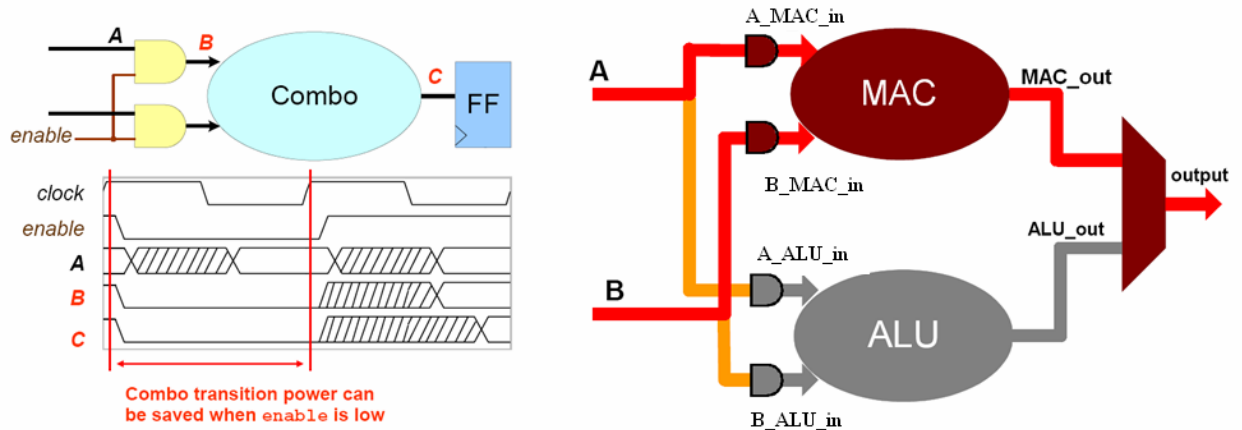
Dhrystone 2.1 with 8,332 clock cycles	FLIP-FLOP numbers	FLIP-FLOP Transitions no clock gated implemented	FLIP-FLOP Transitions clock gated implemented	Transitions can be saved (%)
FF could be gated	1,576	13,131,232	796,517	93.93%
FF could not be gated	27	224,964	224,964	0%
Total	1,603	13,356,196	1,021,481	92.35%

**Table3.1 Transitions saved with Clock gated synthesized CG cell**

### 3.2.2 Unexecuted Function Units Gated

Last section a low-power technique used for sequential logic to gate unused registers has been proposed, furthermore, another low-power technique used for combinational logic of unexecuted units is proposed in this section. For instance, signal-A from the Fig.3.9, will be gated in front of combinational logic in the unexecuted units to avoid unnecessary combinational logic transitions to cause extra power wasting. As can be seen in the Fig.3.9, signal-B and signal-C can be forced to zero as enable signal is low; therefore, the combinational logic from signal-B to signal-C of unexecuted units can be shut down to save any unnecessary combinational logic transitions. Moreover, the Fig.3.9 also shows an example that while an instruction is executed in MAC all the combinational logics in ALU will be shut down to save power.

Two different low-power techniques are implemented in Sequential logic in preceding section and combinational logic this section; then another low-power technique from a point of view of property of function units will be discussed in the next section.



**Fig.3.9 Input data gated in unexecuted units**

### 3.2.3 Low-power Consumption Property of Fong-adder

Fong-adder [3] is one of the most significant function units in the proposed core design due to its low-power consumption, and smaller area using with no increasing performance overhead.

As can be seen in the Table3.2, Fong-adder compares to Hybrid K-S Ling-adder [5], a fastest adder at present but power consumption is high and area usage is enormous. The former saves 32.40% power as data width is 32-bit and 12.05% as data width is 64-bit. As can be seen in the Table3.3 and Table3.4, the timing analysis of Fong-adder compares to Hybrid K-S Ling-adder is %2 better as data width is 32-bit and %6.56 better as data width is 64-bit; in addition, the area analysis of Fong-adder compares to Hybrid K-S Ling-adder is also %21.40 better as data width is 32-bit and 17.72% better as data width is 64-bit.

The proposed design implemented with Fong-adder has a characteristic of ultra-low power consumption and small area usage but still remains high-performance. In fact, this is one of the most outstanding advantages of the proposed core design.



Power Delay Product Analysis (UMC 0.18um/TT corner)					
Data Width	Hybrid K-S Ling-adder (power)(mW)	Fong-adder (power)(mW)	Hybrid K-S Ling-adder (PDP)(pJ)	Fong-adder (PDP)(pJ)	PDP Saving
32	18.98	12.83	18.98*1.02	12.83*1.00	33.73%
64	32.77	28.82	32.77*1.22	28.82*1.14	17.82%

**Table3.2 PDP Analysis about Fong compared to Hybrid K-S Ling**

Timing Analysis (UMC 0.18um/TT corner)			
Data Width	Hybrid K-S Ling-adder (ns)	Fong-adder (ns)	Timing Saving
32	1.02	1.00	2.00%
64	1.22	1.14	6.56%

**Table3.3 Timing Analysis about Fong compared to Hybrid K-S Ling**

Area Analysis (UMC 0.18um/TT corner)			
Data Width	Hybrid K-S Ling-adder (um <sup>2</sup> )	Fong-adder (um <sup>2</sup> )	Area Saving
32	14173.790	11140.114	21.40%
64	28839.888	23730.583	17.72%

**Table3.4 Area Analysis about Fong compared to Hybrid K-S Ling**

### 3.3 Summaries of Proposed Core Designs

This section has discussed architecture of the proposed core design which includes discussion about block diagrams, control logic, arithmetic/logical operation executed in EX-stage, and the mechanism of multi-cycle multiplication in EX-stage. Three different implementation types of low-power techniques which include unused registers gated, unexecuted function units gated, and low-power consumption property of Fong-adder have also been discussed.

The proposed core architecture is composed of 8 major functional blocks

which include instruction decoder, a register-file with 31 general purpose 32-bit registers with 6 banks division and 6 status registers, an ALU with a 64-bit Fong-adder and a logic unit, a 32x8 Ling-multiplier controlled by multi-cycle multiplication Sub-FSM, a forwarding unit, an address register selecting one valid address from different sources, read/write data selection with data alignment, and control logic.

The control logic of the proposed design is composed of 4 Sub-FSMs and 1 main FSM. The 4 sub-FSMs are load/store sub-FSM, shift sub-FSM, multiplication sub-FSM, and branch sub-FSM. They are all controlled by the main FSM. Branch sub-FSM has the lowest priority of all the 5 FSMs and can be entered from load/store sub-FSM and shift-sub FSM. Both load/store sub-FSM and shift-sub FSM and Mul sub-FSM are mutually exclusive; moreover, the 4 FSMs instead of branch sub-FSM have equal priority and which FSM should be entered decided by instruction decoded information.

Arithmetic or logical operation is handled in the execution stage by ALU which is composed by a 64-bit Fong-adder and a logic unit. Src\_A and Src\_B going through a BS will be multiplexed by a Reverse-Inverse-MUX; then 2 output results from the Reverse-Inverse-MUX are sent to the logic unit or higher 32-bit part of Fong-adder depending on which types of instruction are decoded.

Normal multiplication without accumulation and writing Long-result of 64-bit out needs at least 2 cycles and at most 5 cycles; however, multiplication with accumulation operation takes one more cycle and writes Long-result of 64-bit out to two 32-bit registers also takes one more cycle. Therefore, a 32-bit multiplicand multiplies a 32-bit multiplier with accumulation and needs writing Long-result of 64-bit out takes 7 cycles in total.

The data stored in Registers are updated in every clock cycle without any clock

gating operation to cause huge power consumed. One manipulation that makes new data to transmit into D port of the FLIP-FLOP only when the enable signal is high can be implemented in RTL level; on the other hand, a synthesized CG cell will make clock of the unused registers gated and guarantee that no data updated any more for holding old value. This method using synthesis scripts can absolutely shut down updating operations of unused registers and save unnecessary power consumption as much as it can and will be discussed in Chapter 5.

A low-power technique used for combinational logic of unexecuted units is that input data will be gated in front of combinational logic in the unexecuted units to avoid unnecessary combinational logic transitions to cause extra power wasting.

Besides the two different low-power techniques implemented in Sequential logic and combinational logic, the third low-power technique is implemented by a Fong-adder since its characteristics of ultra-low power consumption, small area usage, and remaining high-performance compared to Hybrid K-S Ling-adder which is the fastest adder at present.

The proposed design has been discussed in detail this section, and some experiment results about power, timing, and area of the proposed design including Pre-SIM and Post-SIM will be shown and be analyzed with comparison to other processor cores in the next section.

# CHAPTER 4


## EXPERIMENTAL RESULTS

### 4.0 Overview

In this chapter, we provide the experimental results of the proposed design. Section 4.1 elaborates the environment for implementation. Section 4.2 provides the data and statistics of the experiment and discusses the experimental results.

### 4.1 Implementation

This section describes implementation of the simulation environments which include coding by Verilog HDL, simulators with Cadence Verilog®-XL and Debussy, a synthesizer with Synopsys Design Compiler®, a power analyzer with Synopsys PrimePower®, cell library with Artisan TSMC 0.18  $\mu\text{m}$  technology, and Place&Route with Cadence SOC Encounter®. All of simulation environment are listed in Table4.1.



Coding	<i>Verilog</i> HDL
Simulator	Cadence <i>Verilog</i> ® -XL; Debussy
Synthesizer	Synopsys <i>Design Compiler</i> ®
Power Analyzer	Synopsys <i>PrimePower</i> ®
Cell Library	Artisan TSMC 0.18 $\mu\text{m}$ technology
Place&Route	Cadence SOC Encounter®

**Table4.1 Simulation environment setup for experiments**

## 4.2 Discussion of Experimental Results

### 4.2.0 Overview of Experimental Results

All results are shown in tabular form with discuss under the tables; besides, the improvement rate of each comparison relative to the proposed design are also provided in percentage.

The result of critical path delay in worst case, area cost at critical timing, and power consumption will be reported and compared at the following sections.

### 4.2.1 Comprehensive Comparison

This section provides all experimental results with a comprehensive comparison compared among the proposed core ACARM7, ARM7 Compatible Processor (Korus 2005) and ARM7TDMI. As can be seen in the Table4.2, the process technology used by the proposed core ACARM7 and ARM7-Korus2005 are both 0.18um while ARM7TDMI uses 0.25um technology; therefore, all comparisons in the following sections will be normalized to 0.18um technology for correctness.

The row named “Other characteristics” of the Table4.2 shows some specific characteristics which are unique to each of different cores; in addition, the row named “Note” indicates some attention notes to clear some misunderstandings might be made. The p.s.1 of the note is that all the experimental results of ARM7TDMI are come from a TSMC 0.25um hard macro; on the hand, p.s.2 and p.s.3 mentions that both benchmarks for power estimation of ACARM7 and ARM7TDMI are based on Dhrystone with temperature 25C, but the former is in voltage condition of 1.8V and the later is in 2.5V. The last note reminds readers that all experimental results are obtained in worse case except power estimation is measured in typical case.

All the information obtained from Table4.2 like timing, area, and power will

be analyzed and compared among the 3 different cores in the following sections.

<b>Comprehensive Comparisons among Three Different Cores</b>			
	Proposed Core ACARM7	ARM7 Compatible Core (Korus 2005)	ARM7TDMI (official released) (p.s.1)
Process(um)	0.18	0.18	0.25
Gate-count(k)	35.75	52	N/A
Area(mm <sup>2</sup> )	0.3567	N/A	1.05
Performance(MHz)	110	90	55
Power(mW)	0.17(p.s.2)	1	0.78(p.s.3)
Pipeline-stage	3	3	3
Other characteristics	No Thumb; DFT supported	No DFT supported	Hard Core No DFT supported
Note	p.s.1:TSMC 0.25um hard macro		
	p.s.2:Dhrystone,1.8V,Temp=25° C		
	p.s.3:Dhrystone,2.5V,Temp=25° C		
	All experiment results are measured in worse case except power estimation is in typical case		

**Table4.2 Comprehensive comparison among 3 different cores**

## 4.2.2 Timing Comparison

This section discusses the timing comparison among 3 different cores and the performance of ARM7TDMI obtained from Table4.2 should be normalized to 0.18um technology first by using an equation below:

$$55 \times \left( \frac{0.25}{0.18} \right) = 76$$

After normalization, the performance of ARM7TDMI is normalized to 76MHz. However, it is still lower than 90MHz of ARM7-Korus2005 and 110MHz of

ACARM7. The proposed design has such a high performance because it uses a high-performance IPs like Fong-adder and Ling-Multiplier and implements the ISA without Thumb instructions. It gains a significant improvement in the critical path.

As shown in Table4.3, the performance of ACARM7 is improved 44.74% compared to a normalized result from ARM7TDMI and even 22.22% better than ARM7-Korus2005.

<b>Performance Comparison among Three Different Cores</b>			
	Proposed Core ACARM7	ARM7 Compatible Core (Korus 2005)	ARM7TDMI (official released)
Performance(MHz)	110	90	76(p.s.1)
Improvement compared to ARM7TDMI (%)	44.74	18.42	
Improvement compared to ARM7-Korus2005 (%)	22.22		-15.56
Note	p.s.1: Obtained from normalization to 0.18um		

**Table4.3 Performance comparison among 3 different cores**

The high-performance of the proposed design has been discussed and analyzed with experimental results this section, and other analyses which are interested by readers will be described in following sections.

### 4.2.3 Area Comparison

This section discusses the area comparison among the cores. It should pay attention that the area of ACARM7 is in technology of 0.18um and the area of ARM7TDMI is in technology 0.25um. Two results of the area with different

technologies can not be normalized and compared since design rule check (DRC) with different technologies can not be directed normalized by a single equation. Nevertheless, the gate-count of ACARM7 is much lower than the one of ARM7-Korus2005 and improves 31.25% compared to ARM7-Korus2005, as shown in Table4.4.

<b>Area/Gate-count Comparison among Three Different Cores</b>			
	Proposed Core ACARM7	ARM7 Compatible Core (Korus 2005)	ARM7TDMI (official released)
Gate-count(k)	35.75	52	N/A
Area(mm <sup>2</sup> )	0.3567(p.s.1)	N/A	1.05(p.s.2)
Gate-count saving compared to ARM7-Korus2005 (%)	31.25		N/A
Note	p.s.1: 0.18um technology		
	p.s.2: 0.25um technology		

**Table4.4 Area/Gate-count comparison among 3 different cores**

The gate-count comparison between the proposed design ACARM7 and ARM7-Korus2005 shows that the core size of the former is much smaller for 2 primary reasons. The first is that no Thumb instruction implementation strategy and the second is that more design considerations are taken into account. The small area/gate-count is also one reason for low-power designs and is what designers and users want to see; On the other hand, power comparison will be analyzed in next section.



## 4.2.4 Power Comparison

This section describes power comparisons among the cores above and the power of ARM7TDMI should be normalized to 0.18um technology first by an equation come from  $P=CV^2$  and listed below:

$$0.78 \times \left(\frac{0.18}{0.25}\right)^2 = 0.404$$

As can be seen in the Table4.5, the power of ARM7TDMI is normalized to 0.404mW after normalization. However, it is still higher than 0.17mW of ACARM7 but lower than 1mW of ARM7-Korus2005. An amazing thing occurs that the power improvement of ACARM7 compared to the normalized power of ARM7TDMI is 57.9% better and is 83% better compared to ARM7-Korus2005.

<b>Power Comparison among Three Different Cores</b>			
	Proposed Core ACARM7	ARM7 Compatible Core (Korus 2005)	ARM7TDMI (official released)
Power(mW)	0.17	1	0.404(p.s.1)
Power saving compared to ARM7TDMI (%)	57.9	-147	
Power saving compared to ARM7-Korus2005 (%)	83		59.6
Note	p.s.1: Obtained from normalization to 0.18um		

**Table4.5 Power comparison among 3 different cores**

The proposed design has a characteristic of ultra low-power for many reasons and almost the same with the reasons discussed in Section 4.2.2; in fact, a right implementation strategy with no Thumb instructions, a better choice of high quality

IP like Fong-adder and Ling-multiplier, and more design consideration taken into account brings such a good performance of the proposed design. On the other hand, Power-Delay-Product (PDP) is also provided in Table4.6. The table still shows the PDP of ACARM7 is 70.7% lower compared to ARM7TDMI and 86% lower compared to ARM7-Korus2005. As a matter of fact, ACARM7 is the only one of the three cores which can look after both sides inclusive of low-power consumption and high-performance.

<b>Power-Delay-Product Comparison among Three Different Cores</b>			
	Proposed Core ACARM7	ARM7 Compatible Core (Korus 2005)	ARM7TDMI (official released)
Power(mW)	0.17	1	0.404(p.s.1)
Delay(ns)	9.14	11.1	13.16(p.s.1)
PDP(pJ)	1.5538	11.1	5.31664(p.s.1)
PDP saving compared to ARM7TDMI (%)	70.7	-108.7	
PDP saving compared to ARM7-Korus2005 (%)	86		52.1
Note	p.s.1: Obtained from normalization to 0.18um		

**Table4.6 Power-Delay-Product comparison among 3 different cores**

#### **4.2.5 Other Characteristics**

Considering of other characteristics of the Table4.2, DFT (Design for Testability) is supported by ACARM7 since chips might fail to work due to manufacture problem; as a result, additional test circuit shall be added such as scan circuit and built-in self test (BIST) circuit.

ACARM7 is supported a characteristic of cycle-accurate verified by a behavior systemC model and will be discussed in Chapter 5.

### **4.3 Summaries of Experimental Results**

This chapter has analyzed the experimental results among different cores inclusive of the proposed design ACARM7, ARM7 Compatible Core (Korus 2005), and ARM7TDMI. First of all, the implementation of the simulation environments is introduced and a comprehensive comparison is provided for discussions about timing, area, and power in detail.

In timing comparison, the performance of ARM7TDMI obtained from Table4.2 should be normalized to 0.18 $\mu$ m technology first and the performance of the proposed design improves 44.74% compared to a normalized result from ARM7TDMI and 22.22% better than ARM7-Korus2005.

In area comparison, two areas with different technologies can not be normalized and compared directly; however, the gate-count of ACARM7 is much lower than the one of ARM7-Korus2005 and improves 31.25% compared to the later.

In power comparison, the power of ARM7TDMI should be normalized to 0.18 $\mu$ m technology first and the power of ACARM7 improves 57.9% compared to a normalized result from ARM7TDMI and 83% better than ARM7-Korus2005. On the other hand, the PDP of ACARM7 is 70.7% lower compared to ARM7TDMI and 86% lower compared to ARM7-Korus2005.

DFT and a characteristic of cycle-accurate verified with a systemC model are also supported by ACARM7 and will be discussed in next chapter.

# CHAPTER 5

## PROPOSED VERIFICATION STRATEGY

### 5.0 Overview

With the prevalent fashion of SOC, designs nowadays become more and more complicated. The thesis provides a thorough and rigorous verification flow strategy. Section 5.1 describes the implementation of verification environment. Section 5.2 provides functional verification to ensure functional correctness of the design. Section 5.3 proposes synthesized netlist verification to ensure the correctness of the synthesis procedure. Section 5.4 provides test plan and testability measurement.

### 5.1 Implementation

These are implementation of verification environment and listed in Table5.1. Design rule check is checked by Novas *nLint*® with adopted Freescale Semiconductor Reuse Standard (SRS); equivalence check is checked by Cadence *Encounter*™ *Conformal*® Equivalence Checker; coverage analysis is checked by TransEDA *Verification Navigator*®; and assertion-based verification is checked by Accelera's Property Specification Language (PSL).

Verification	
Design Rule Check	Novas <i>nLint</i> ® with adopted Freescale Semiconductor Reuse Standard (SRS)
Equivalence Check	Cadence <i>Encounter</i> ™ <i>Conformal</i> ® Equivalence Checker
Coverage Analysis	TransEDA <i>Verification Navigator</i> ®
Assertion-based Verification	Accelera's Property Specification Language (PSL)

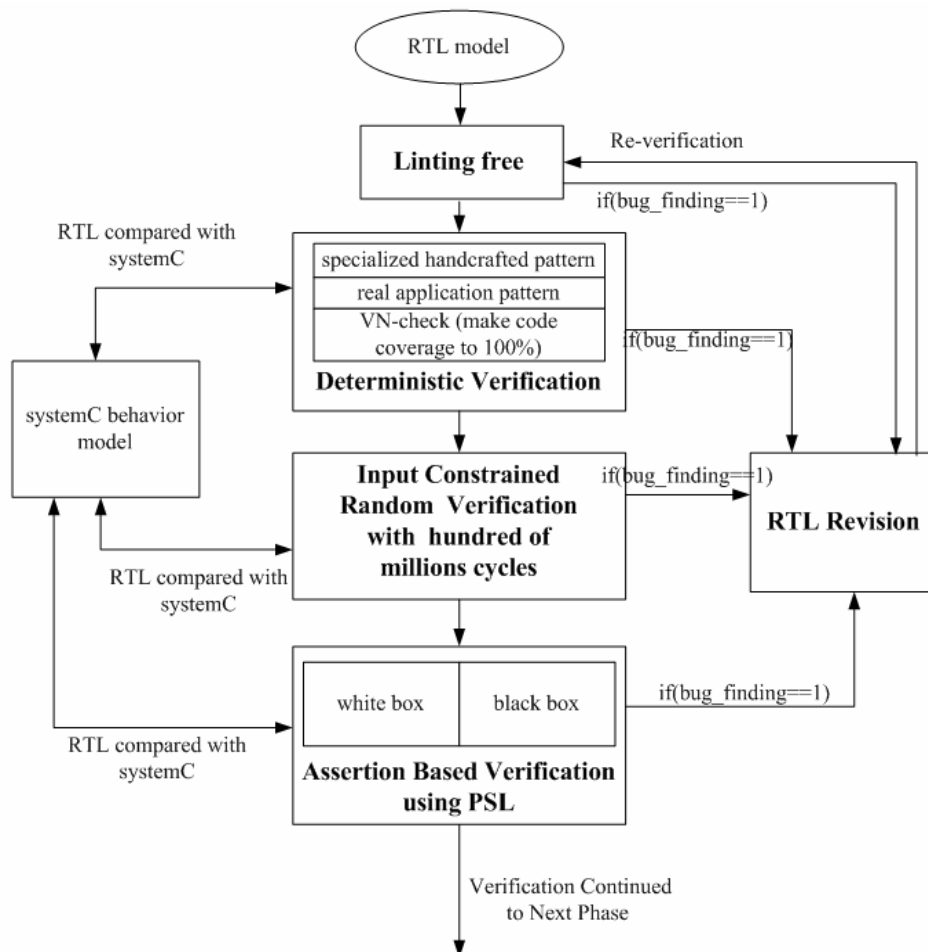
**Table5.1 Verification environment setup**

## 5.2 Functional Verification

### 5.2.0 Overview of Functional Verification

A functional verification flow is proposed in this section. Section 5.2.1 proposes coding style checking by Linting; Section 5.2.2 proposes a deterministic verification; Section 5.2.3 proposes an input-constrained random verification; Section 5.2.4 proposes an assertion-based verification.

The functional verification flow diagram is listed in the Fig.5.1 and in each of steps the RTL model will be verified with a systemC behavior model which is designed for matching all the cyclic behaviors of ADS. All mismatches from the comparison will cause the flow back to RTL revision step to modification.



**Fig.5.1 Functional Verification flow**

## 5.2.1 Coding Style Checking by Linting Free

This section describes a static coding style check which improves the quality of the design in reuse and verification perspectives of RTL verilog. Checking the coding-style of the design by Novas nLint tool with 328 lint rules of adopted Freescale Semiconductor Reuse Standard (SRS) will avoid all kinds of warnings and errors including naming, synthesis, simulation, common syntax, undeclared objects, unexpected latches, DFT issue, and so on.

## 5.2.2 Deterministic Verification

This section describes deterministic verification which is made to check all regular cases and special corner case should be confronted with in the simulation phase. Deterministic verification is composed of 3 parts including specialized handcrafted pattern, real application pattern, and verification by VN-check (makes code coverage to 100%).

The handcrafted pattern is written in all cases of instructions implemented in the proposed design. It checks all results of instructions to be right in the first step in deterministic verification.

The real application patterns are implemented by some benchmarks like Dhrystone, Whetstone, and DSPstone. Moreover, a JPEG encoder program is also provided to verify the correctness of the design and an encoded result will be showed Section 5.2.4. This kind of pattern checks real cases met in applications of real world and is essential in deterministic verification phase.

The third phase of deterministic is code coverage checking by TransEDA's Verification Navigator® (VN-check) which gives metrics on how well the design is being verified. For instance, statement coverage shows the number of times each

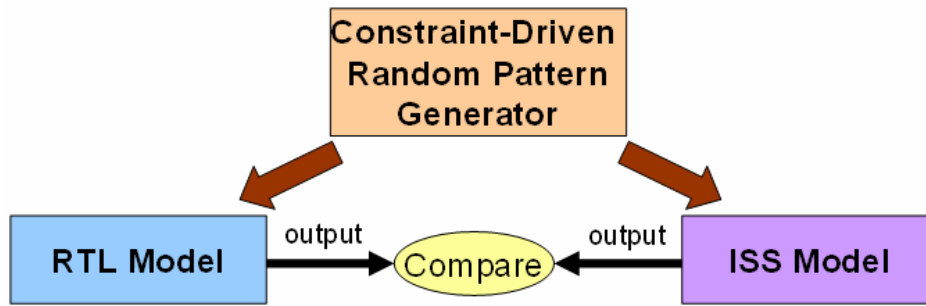
HDL statement is executed and un-executed statements are likely to be redundant in the design; and a state coverage inspects that whether any unreachable states exists in FSM similarly. According to Motorola's Semiconductor Reuse Standards, the statement, branch and state coverage of a design should achieve 100% while the lower bound of condition coverage is not listed.

After the code coverage verification, it suggests that the test vectors applied to the design under verification are sufficient and the next verification step input-constrained random verification can be entered.

### **5.2.3 Input-constrained Random Verification**

Input-constrained random verification is implemented by a constraint-driven random pattern generator which generates random pattern to both ISS model and RTL model and values of both models are compared cycle by cycle, as shown in Fig.5.2. This kind of verification is made to detect all of unexpected cases and the random pattern is constrained to the meaningful range to avoid undefined instructions generation. More simulation cycles are verified, more vector space is spanned by random patterns; therefore, less bugs exists in the design and more robust design can be declared. In fact, two billion cycles have been simulated until now and 16 bugs founded in the first 100 million cycles. No more bugs have been found after then. All bugs found in functional verification will be listed in Table5.2

On the other hand, more cycles have been simulated, harder the bugs can be found; therefore, another different powerful verification method will be discussed in the next section.

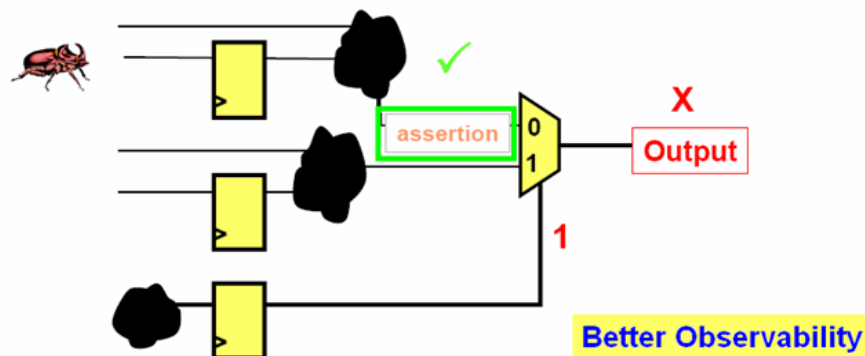


**Fig.5.2 Input-constrained Random Verification**

### 5.2.4 Assertion Based Verification

This section provides a powerful verification method and is indispensable in the verification strategy; that is assertion-based verification. An assertion is a statement about intended behavior of designers, which should be verified. The purpose is to ensure the created design behaves according to the intention of designers.

Unlike traditional simulated-based “black-box” verification, which requires stimulus to trigger bugs and the propagate the error response to output, assertion based “white-box” verification inserts monitors inside the design signals to report incorrect behavior at run-time and allows designers to identify and locate bugs instantly. As can be seen in the Fig.5.3, bugs will not be found if no monitoring by assertion implemented.



**Fig.5.3 Assertion-based verification**



There are many benefits of using assertion based verification. First of all, assertion improves the ability of observation of a design under verification; second, assertion reduces the debug time because of better observation and better isolation of error; in addition, assertion can be used to check the interfaces of a design, thus improving the integration through correct usage checking; moreover, assertion can also facilitate formal analysis to help verify a design; finally, assertion also helps to reveal the intend of designers clearly by specifying correct behavior unambiguously, so that design could be easily reused and verified.

All bugs found in functional verification phase have been listed below:

<b>All of the Bugs Found in Functional Verification Phase</b>			
Categories of Patterns	Number	Note	The Reason for Not Early Founded
Real Programs (Benchmarks) of Deterministic Verification	<b>3</b>	(a) Control signal decoded error about FSMs	(g) Real application programs are more practical than handcrafted programs.
Input-Constrained Random Verification	<b>16</b>	(b) The execution cycle of multi-cycle multiplication mismatch with ISS model	(h) Bugs need random patterns through a great quantity of simulation time to be found out since handcrafted pattern is not enough.
		(c) Same with (a)	(i) Same with (g)
		(d) Exception handler cycle determined mismatch	(j) Same with (g)
		(e) Errors occur in conditional execution	(k) Same with (g)
Assertion-based Verification	<b>1</b>	(f) Two different interrupts occur simultaneously but correctness is not influenced.	(l) Without assertion, unexpected behaviors will not be found if functionality is still correct. It lowers the quality of an IP.

**Table5.2 All of the bugs found in functional verification phase**

As can be seen in the Table 5.2, after handcrafted pattern verification, bugs still exist in the design and other efficient verification methodology should be used to make design more robust. Real programs by some benchmarks have found 3 another bugs about control signal decoded error of FSMs since real application programs is more practical than handcrafted programs. In addition, input-constrained random verification has found 16 bugs by many different reasons with the same idea. This idea is that some bugs of complicated design need random patterns through a great quantity of simulation time to be found out since the quantity of handcrafted pattern is not enough. However, some bugs will never be found if we used traditional simulated-based “black-box” verification; assertion-based “white-box” verification monitors the behavior of the design and finds out any bugs causing unexpected behavior even no error answer occurs.

After thorough and rigorous functional verifications, a real application program JPEG encoder is used for verifying the functionality of the proposed design and the result can be seen below. The Fig.5.4 is an original bmp file and the Fig.5.5 is an encoded jpeg file. The same relation is between the Fig.5.6 and the Fig.5.7. The time for encoding a 64x64 bmp-file to a 64x64 jpg-file by RTL simulation takes 1 hour and for a 176x144 bmp-file to a 176x144 jpg-file takes 6 hours.



**Fig.5.4 64x64.bmp**



**Fig.5.5 64x64.jpg**



**Fig.5.6 176x144.bmp**



**Fig.5.7 176x144.jpg**

Therefore, this section has discussed a thorough and rigorous verification flow strategy in functional verification phase, and synthesized netlist verification phase will be discussed in next section.



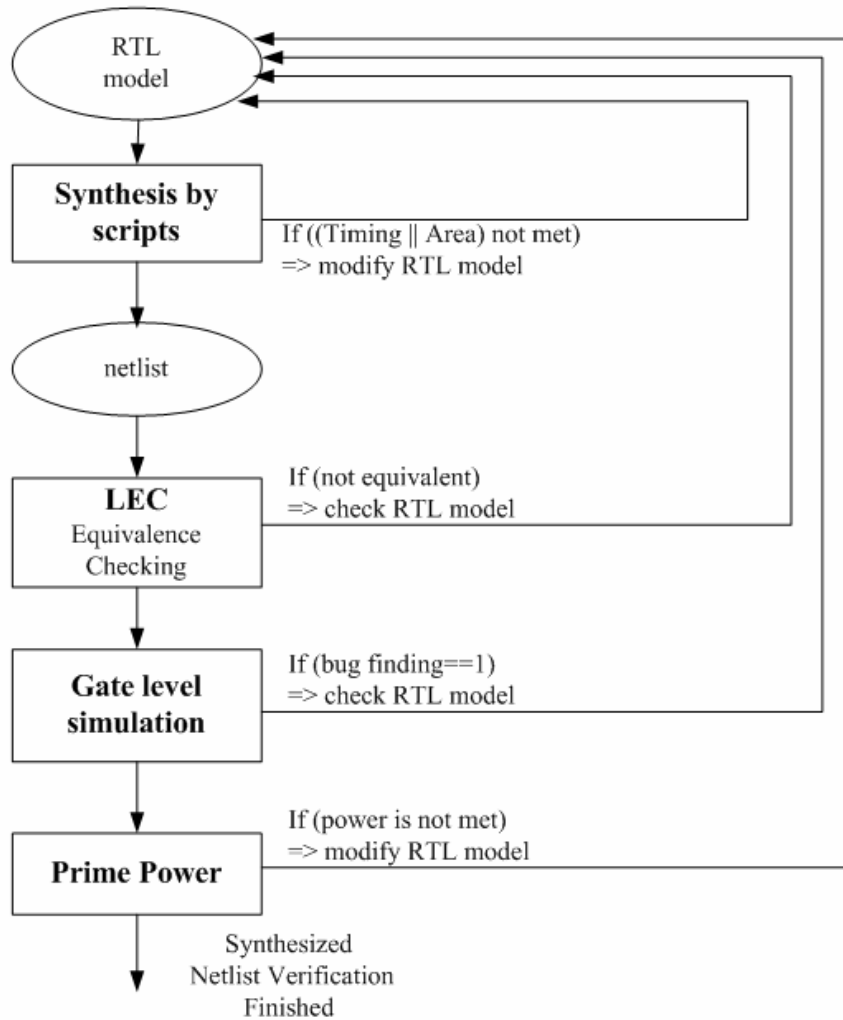
## **5.3 Synthesized Netlist Verification**

### **5.3.0 Overview of Synthesized Netlist Verification**

This section discusses synthesized netlist verification flow. The synthesis procedure by scripts is discussed in Section 5.3.1. Logic equivalence checking is discussed in Section 5.3.2. Gate-level simulation is discussed in Section 5.3.3. Power estimation by prime-power is discussed in Section 5.3.4.

The completed synthesized netlist verification flow diagram is listed below in the Fig.5.8 and in first step the synthesized netlist is checked whether the timing and area meets the specification or RTL of the design will be modified; in second step LEC checking will verify the consistency of an RTL design as it synthesized into gate-level netlist; in third step gate level simulation will be executed to ensure

functional correctness after synthesis; finally, preliminary power estimation by prime power is the last step of the verification flow and exact power estimation has to be measured after P&R and has been discussed in Chapter 4.



**Fig.5.8 Synthesized Netlist Verification flow**

### 5.3.1 Synthesis by Scripts

This section describes synthesis by scripts. Writing and refining the synthesis scripts is the first step to obtain the characteristic of lower power, smaller area, and higher performance netlist. Two methods written in scripts are worth being discussed below:

1. The gated clock technique discussed in Chapter 3 is not directly written in RTL but it is implemented by adding commands in scripts and listed below:

- i. `set_clock_gating_style -sequential_cell latch -minimum_bitwidth 4 -setup 0 -hold 0`
- ii. `insert_clock_gating -module_level`
- iii. `propagate_constraints -gate_clock`

A CG-cell with gated clock will be generated by these commands in scripts. The advantage is that the clock signal of RTL is not operated or handled directly and makes the RTL design ease to port to other embedded systems; for instance, the JPEG decoder system will be discussed in Chapter 6.

2. A command written in scripts ignores the series path which is used for scan chain of DFT and makes no timing optimization to that path since many buffers will be added in this path and unnecessary area will be increased. The command is listed below:

- i. `set_case_analysis 0 test_se`

### **5.3.2 Logic Equivalence Checking (LEC)**

This section describes one common formal verification technique named logic equivalence checking. Tools of LEC use formal mathematical techniques to verify logic functions by comparing input/output conditions and matching an iteration of design with the next. LEC enables the determination of logic function equivalence between one design and another. Moreover, LEC can verify the consistency of an RTL design as it is synthesized into gate-level netlist.

### 5.3.3 Gate Level Simulation

This section describes that gate level simulation is executed after synthesis procedure to ensure functional correctness in low-level netlist.

### 5.3.4 Power Estimation by Prime-power

This section depicts that preliminary power estimation by prime power is the last step of the verification flow and exact power estimation has to be measured after P&R and has been discussed in Chapter 4.

## 5.4 Test Plan and Testability Measurement

Due to the failure in manufacturing process, 5%-40% of chips will be failed after being manufactured and the extra cost can be huge if the chips delivered do not work; hence, testing becomes an important procedure to ensure the delivered chips work. Design-For-Testability (DFT) synthesis using scripts is one of the test methods and additional scan-chain circuit is generated.

As can be seen in the Table5.3, Test coverage of the proposed design can not reach 100% for gated-clock CG-cells composed of latches and AND gate. In fact, it is worth using this kind of low-power technique to lower power consumption with a small defect.

<b>Un-collapsed Stuck Fault Summary Report</b>	
Total faults	87428
Test coverage	99.38%
#Internal patterns	418
#Basic scan patterns	418

**Table5.3 Un-collapsed Stuck Fault Summary Report**

## 5.5 Summaries of Proposed Verification Strategy

This chapter provides a thorough and rigorous verification flow strategy in two phases, functional verification and synthesized netlist verification. The test plan and testability measurement of the proposed design is also provided.

A functional verification has been proposed that in each of steps of the flow the RTL model will be verified with a systemC behavior model which is designed for matching all the cyclic behaviors of ADS and all mismatches will cause flow back to RTL revision step to modification. Deterministic verification, which is made to check all regular cases and special corner case should be confronted in simulation phase, is composed of 3 parts including specialized handcrafted pattern, real application pattern, and verification by VN-check (make code coverage to 100%). Input-constrained random verification is made to detect all of unexpected cases and the random pattern is constrained to the meaningful range to avoid undefined instructions generation. More simulation cycles are verified, more vector space is spanned by random patterns; therefore, less bugs exists in the design and more robust design can be declared. Assertion-based verification finds out the bugs which will never be found if we used traditional simulated-based “black-box” verification; however, assertion-based “white-box” verification monitors the behavior of the design and finds out any bugs causing unexpected behavior even no error answer occurs. The completed synthesized netlist verification flow is composed of synthesis procedure by scripts, LEC checking, gate level simulation, and prime power.

After the verification flow, it is guaranteed that the proposed embedded processor owns high quality of an IP, a characteristic of bug-free robustness, high reusability and convenient porting issue. On the other hand, a JPEG decoder system will be presented in the next chapter

# CHAPTER 6

## JPEG DECODER SYSTEM

### 6.0 Overview

This chapter presents a JPEG decoder system implemented by the proposed core design which is configured in the FPGA as a specific purpose processor and all system behaviors are controlled by a ARM9EJ-S processor on the develop board. Section 6.1 depicts the implementation of the system. Section 6.2 describes architecture of system level. Section 6.3 describes architecture of hardware level. Section 6.4 describes program control flow in software level. Section 6.5 discusses the experimental results of proposed system. Section 6.6 discusses the Summaries of JPEG Decoder System.



### 6.1 Implementation

The development board of JPEG decoder system is implemented by *Versatile RealView Platform Baseboard for ARM926EJ-S®* [6]; the proposed core ACARM7 is configured in an FPGA of *Versatile LT-XC2V6000 (Xilinx VirtexII)* [7]; the decoded file is displayed by a LCD panel, 8.4 VGA (640x480) Color VCD Panel; the system is semi-hosted by a PC with an *ARM® RealView MultiICE®*; All develop environment is based on *ARM® Developer Suite (ADS) version 1.2*; The proposed design will be transformed to BIT-file to configure in the FPGA by *Xilinx® Integrated Software Environment (ISE) 6.2i*. All environment setups can be seen in the Table6.1.



<b>Decoder System</b>	
Development Board	<i>Versatile RealView Platform Baseboard for ARM926EJ-S®</i>
Logic Tile	<i>Versatile LT-XC2V6000 (Xilinx VirtexII)</i>
LCD Kit	8.4 VGA (640x480) Color VCD Panel
Multi-ICE	<i>ARM® RealView MultiICE®</i>
ADS	<i>ARM® Developer Suite (ADS) version 1.2</i>
Xilinx ISE	<i>Xilinx® Integrated Software Environment (ISE) 6.2i</i>

**Table6.1 Verification environment setup**

## **6.2 Architecture of System Level**

### **6.2.0 Overview of Architecture of System Level**

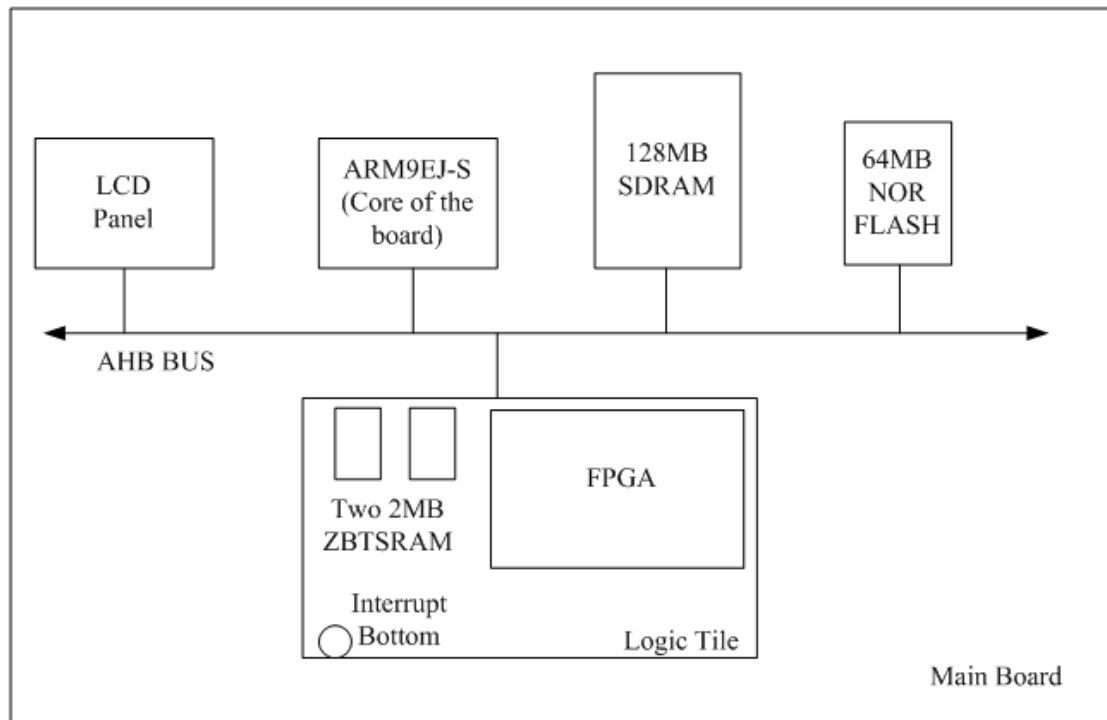
This section introduces the overview of the proposed system in system level view; Section 6.2.1 describes major components of the proposed system and Section 6.2.2 describes all control flow of the proposed system.

#### **6.2.1 Major Components of the Proposed System**

This section describes major components of the proposed decoder system, which is composed of an ARM9EJ-S core, a 128MB SRAM, a 64MB NOR flash, a LCD panel, and a Logic Tile. The ARM9EJ-S core controls all behaviors and components of the entire system; the data in the SRAM can be loaded or stored by the core; The 64MB NOR flash is a nonvolatile storage and data or programs can be stored in the flash prepared to be loaded into SRAM automatically as powered up; a LCD panel displays the image of decoded pictures; and a Logic Tile includes an FPGA, two 2MB ZBTSRAM(Zero Bus Turnaround SRAM), a push bottom which is used for interrupt input of the system, and many other devices and components. All

components discussed above are shown in the Fig.6.1.

This section has described major components of the proposed system and more details of control flow in system level of the proposed system will be discussed in the next section.

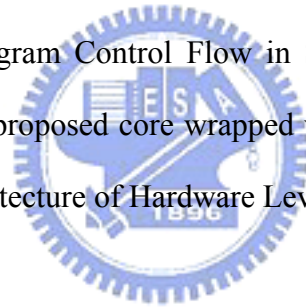


**Fig.6.1 Block diagram in system level**

## **6.2.2 Control Flow of the Proposed System**

This section describes the control flow of the proposed system. First, all the binary data is obtained from ADS tool and is written into the NOR flash in advance by a multi-ICE with semi-hosting mechanism; the load address to the SDRAM or ZBTSRAM is also designated at the same time. Second, all the binary data is auto-loaded from the NOR-flash into SDRAM or ZBTSRAM respectively; In this step, the instructions of JPEG decoder are loaded into both ZBTSRAMs on the

Logic Tile and all of the un-decoded JPEG files are also loaded into the SDRAM. In addition, the main control flow program is also load into SDRAM and executed by ARM9EJ-S in the third step. Third, ARM9EJ-S moves two un-decoded JPEG files from SDRAM into both ZBTSRAMs; therefore, both instructions and un-decoded data are prepared for ACARM7 configured in FPGA of the Logic Tile. Fourth, ARM9EJ-S sends a signal to tell ACARM7 to begin to decode the data in ZBTSRAM. Fifth, while ACARM7 finishes decoding procedure, it will send a finish signal to tell ARM9EJ-S. Sixth, ARM9EJ-S moves the decoded data from both ZBTSRAMs of the Logic Tile to the SDRAM. Finally, ARM9EJ-S calls the LCD display program to show decoded data on the LCD panel. These are primary steps of system control flow and more delicate and efficient control techniques will be discussed in Section 6.4 Program Control Flow in Software Level; On the other hand, more details about the proposed core wrapped with an AHB interface will be discussed in Section 6.3 Architecture of Hardware Level.



## **6.3 Architecture of Hardware Level**

### **6.3.0 Overview of Architecture of Hardware Level**

This section describes hardware architecture of the proposed core wrapped with AMBA interface in FPGA of the proposed system. Section 6.3.1 describes all the AHB peripherals in FPGA. Section 6.3.2 describes core wrapped with AHB bus interfece. Section 6.3.3 describes control flow within AHB peripherals in FPGA.

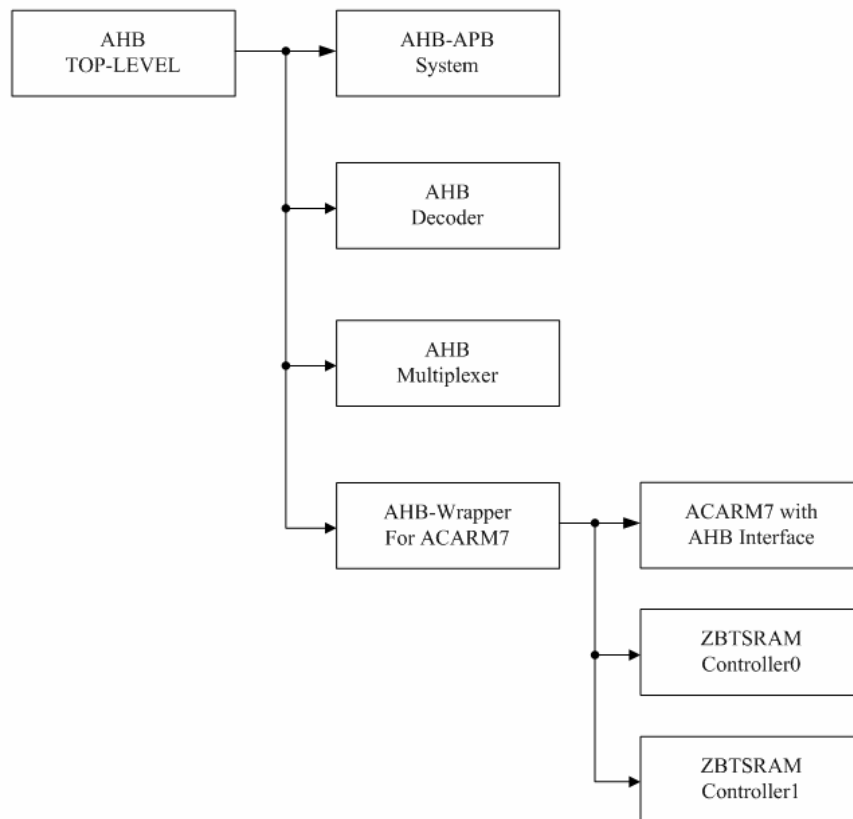
#### **6.3.1 AHB Peripherals in FPGA**

This section describes all the AHB peripherals in FPGA, as can be seen in the Fig.6.2. The RTL design should be wrapped to correspond to the AHB specification

in advance before they configure into an FPGA or will never be used and activated in an FPGA. An example code [8] provided by ARM is implementing AHB peripherals in FPGA of Logic Tiles and will be modified for usage of ACARM7.

As shown in Fig.6.2, the AHB TOP-LEVEL block is the top level HDL (Hardware Description Language) configured in the FPGA and instantiates and interconnects the main block in the all system; the AHB-APB System block includes the bridge of AHB to APB and all the APB peripherals including LED light and interrupt controller. On the other hand, the AHB Decoder block decodes the received address from the AHB Bus to different IP selection signals which are used to activate the corresponded IP to wake up and execute; the AHB Multiplexer block selects the right answer from all different source results and selection is decided by the IP selection signal generated by AHB Decoder. Furthermore, the AHB-Wrapper for ACARM7 block wraps ACARM7 with AHB-wrapper and includes three sub-blocks which are ACARM7 with AHB interface, and two ZBTSRAM controllers. ACARM7 with AHB Interface block will be discussed in Section 6.3.2 in detail. Both ZBTSRAM Controllers control all the data movements of the two ZBTSRAMs on the Logic Tile.

This section has discussed all the AHB peripherals in FPGA, and more details about core wrapped with AHB bus interface will be described in the next section.



**Fig.6.2 AHB peripherals configured in the FPGA**



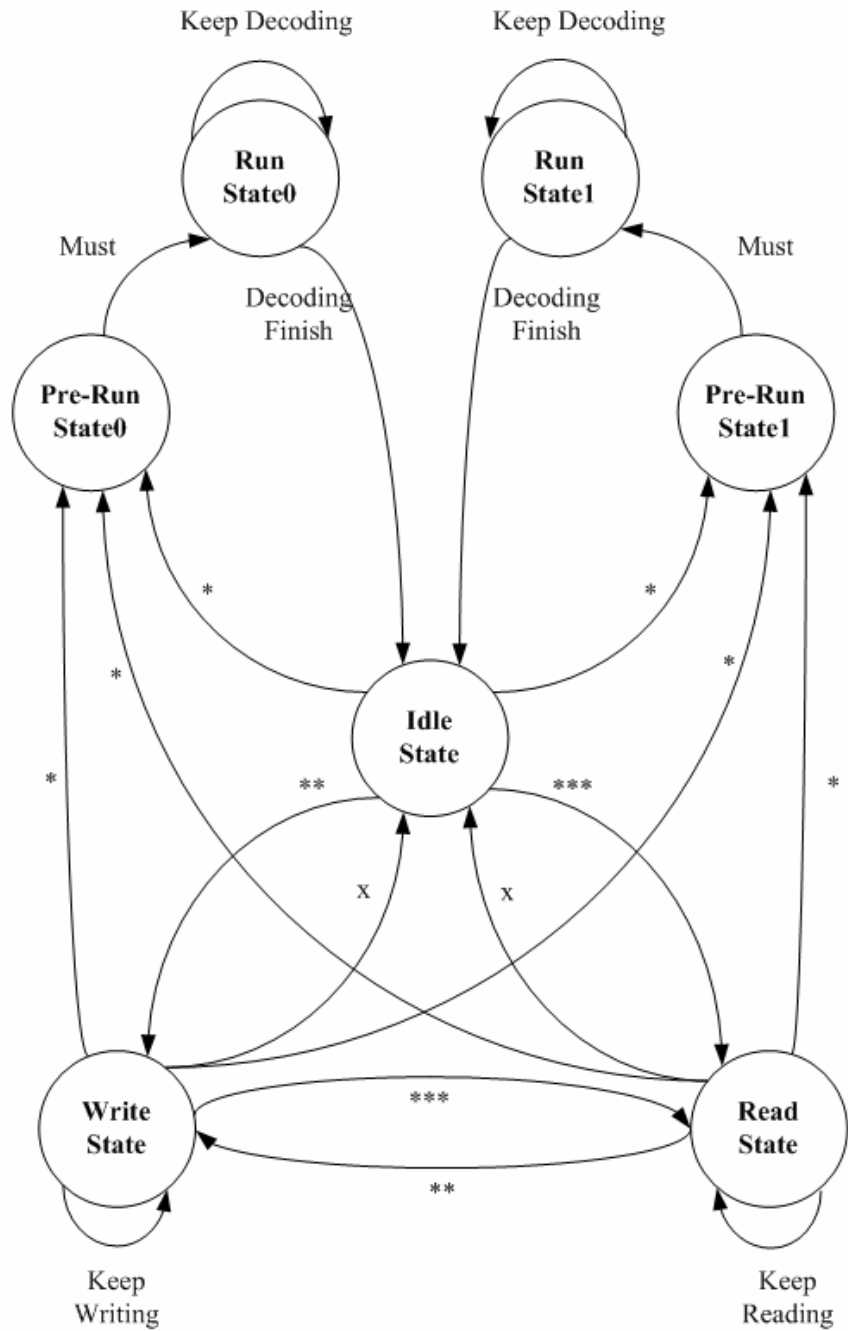
### **6.3.2 Core Wrapped with AHB Bus Interface**

This section discusses the core wrapped with AHB bus interface in FPGA. As can be seen in the Fig.6.3, seven FSMs of the AHB-interface block (shown in the fig.6.4) control all the behaviors between AHB Bus and ACARM7.

As the bus in the Idle State which could be due to decoding finish or not valid operation, no tasks are executed in this cycle and other tasks will be executed in another non-idle state. As the bus jumps into the Write State which is due to writing task requested by ARM9EJ-S , the data from AHB Bus will be written into Current Status Registers (CSR) which controls behaviors of the bus; on the other hand, the information of CSRs of ACARM7 will be read out to put on AHB Bus as the bus jumps into the Read State which is due to reading task requested by ARM9EJ-S; for

example, as ACARM7 finishes a decoding task, a finishing signal will be put on the FINISH register of CSRs. The contents of FINISH register of CSRs will be read out to put on AHB Bus to tell ARM9EJ-S that the decoding task has done as the bus is in Read State. While ARM9EJ-S sends a request to tell ACARM7 to execute the decoding task, the Pre-Run State0 or the Pre-Run State1 will be entered. Which state will be selected from both of them depends on which ZBTSRAM of both ZBTSRAMs is used. The action executed by ACARM7 in both Pre-Run States is to reset the bus first to clear all internal registers of the bus to zero and assures the correctness of ACARM7 's execution before entering into both Run-States every time. ACARM7 does decoding procedure in the Run State and stays in the state until it finishes the decoding task. As can be seen in the fig.6.3, while ZBTSRAM0 is used for JPEG decoding, Run State0 goes after Pre-Run State0 continuously. So does Run State1 and Pre-Run State1 while ZBTSRAM1 is used.

This section has discussed the AHB bus interface of the core, ACARM7, with AHB-wrapper in FPGA and more details about control flow within AHB peripherals in FPGA will be described in next section.



- \* Decoding Task Requested By ARM9
- \*\* Writing Task Requested By ARM9
- \*\*\* Reading Task Requested By ARM9
- X Not Valid

**Fig.6.3 FSMs of the AHB bus interface**

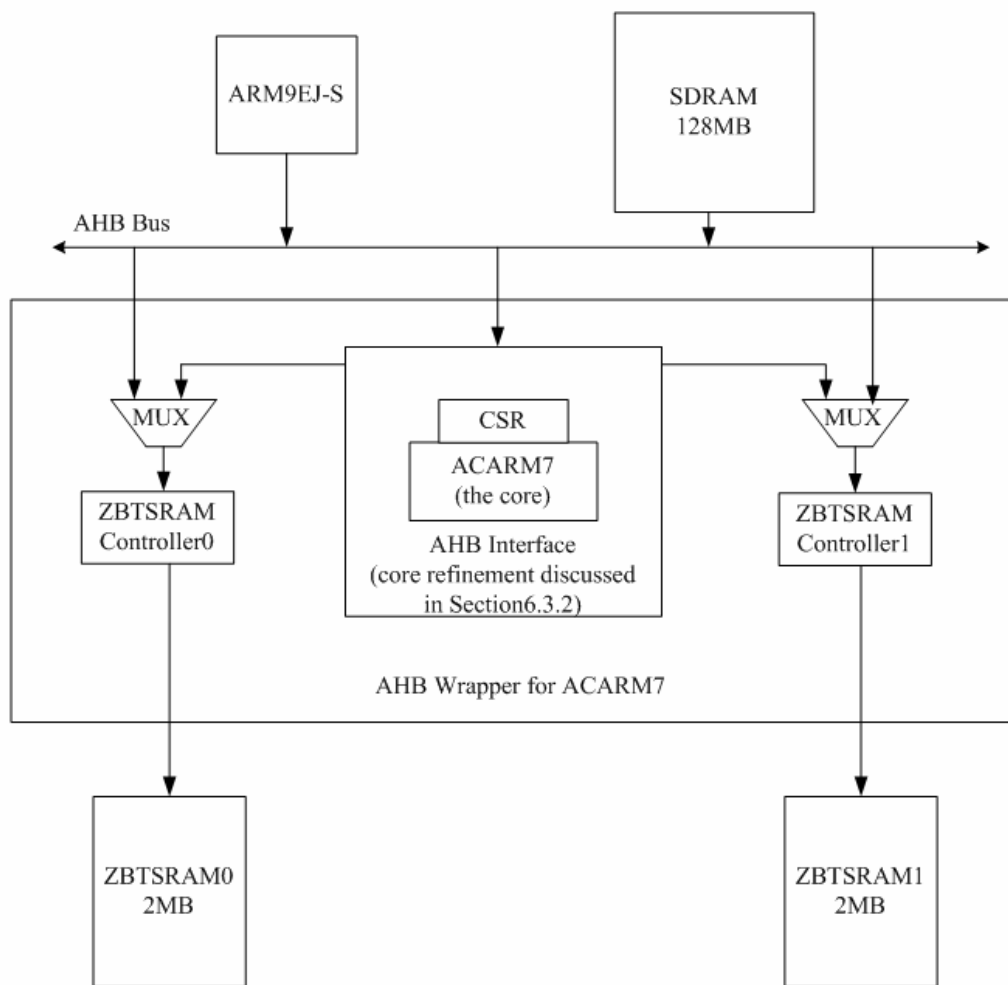
### 6.3.3 Control Flow within AHB-Wrapper for ACARM7 in FPGA

This section discusses the control flow within the block of AHB wrapper for ACARM7, as shown in the Fig.6.2 above. As can be seen in the Fig.6.4, the block AHB-Wrapper for ACARM7 is the same one in Fig.6.2 and the block AHB Interface is the one discussed in Section 6.3.2. ARM9EJ-S, one SDRAM with 128MB, and two ZBTSRAMs with 2MB are also displayed in the Fig.6.4.

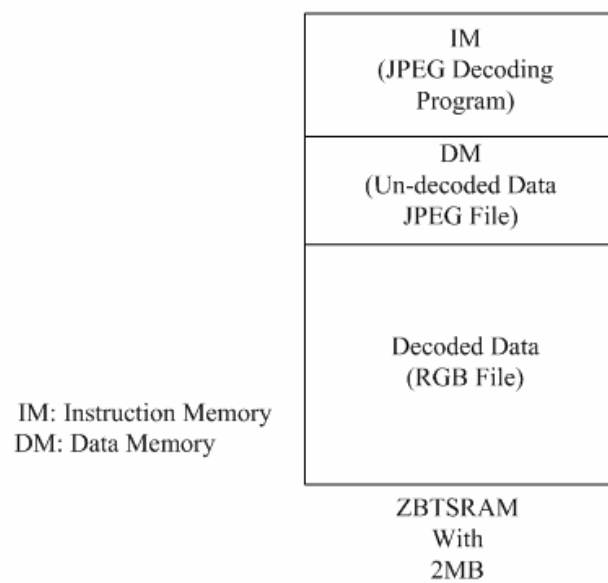
Both ZBTSRAMs in the Logic Tile can be used by either ARM9EJ-S or ACARM7 with the AHB-Wrapper. As can be seen in the Fig.6.5, the memory usage of a ZBTSRAM is divided into three parts which include IM (Instruction Memory), DM (Data Memory), and decoded data. The content of the first part comes from NOR-flash auto-loading as the whole system powers up. On the other hand, ARM9EJ-S requests the un-decoded data moved from the SDRAM to the second part of the ZBTSRAM as a DM and this mechanism will be described more in Section 6.4. Therefore, the necessary information with both instruction and data are prepared for ACARM7 and ACARM7 can use the ZBTSRAM to execute decoding procedure. ZBTSRAM Controller is an essential part to access ZBTSRAM and needs all information received from a processor. As a result, a multiplexer is implemented and selects which processor is in charge of the ZBTSRAM. After decoding procedure by ACARM7 has finished, ARM9EJ-S will take over the ZBTSRAM at the moment and moves the decoded data to the SDRAM for display on LCD panel later.

This section has described the control flow within AHB-Wrapper for ACARM7 and program flow in software level will be described in next section.





**Fig.6.4 AHB Wrapper for ACARM7 (with ARM9EJ-S and SDRAM)**

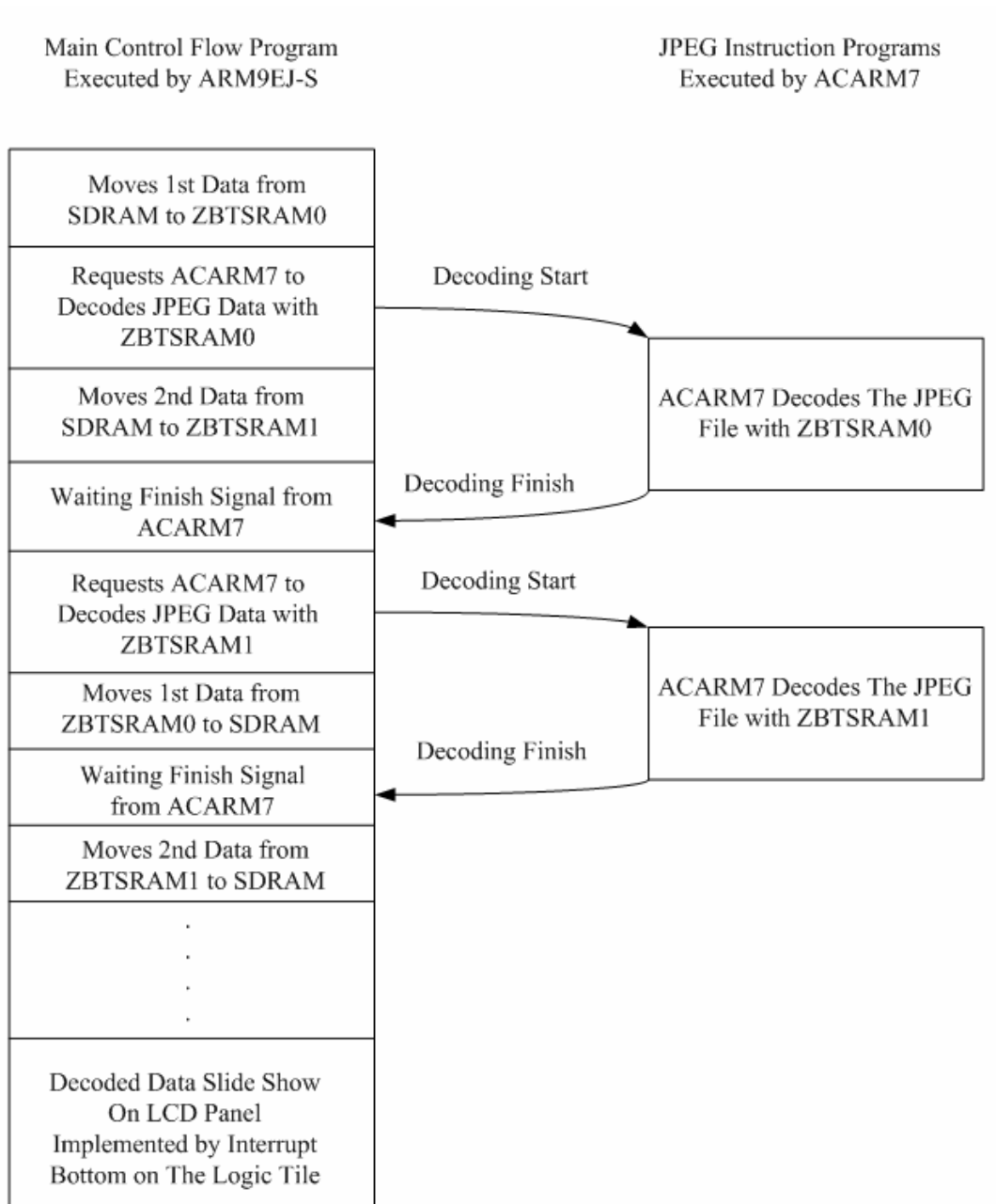


**Fig.6.5 ZBTSRAM memory usage**

## 6.4 Program Control Flow in Software Level

This section describes the program flow in software level. As can be seen in the Fig.6.6, the main control flow program is executed by ARM9EJ-S and JPEG instruction program is executed by ACARM7. First of all, ARM9EJ-S moves the first un-decoded data from SDRAM to the first ZBTSRAM ZBTSRAM0, and then ARM9EJ-S requests ACARM7 to decode the data just moved into ZBTSRAM0. The second un-decoded data is moved to ZBTSRAM1 like the first one and then ARM9EJ-S waits the finish signal from ACARM7 by a polling mechanism. After ACARM7 finishes the first data decoding procedure, ARM9EJ-S requests ACARM7 to decode the second data in ZBTSRAM1 continuously. ARM9EJ-S moves the first decoded data from ZBTSRAM0 to SDRAM and waits ACARM7 to finishes the second decoding procedure in ZBTSRAM1. After ACARM7 finishes the second decoding procedure, ARM9EJ-S moves the second decoded data from ZBTSRAM1 to SDRAM in next step. These steps will be proceeding iteratively. After all the decoding tasks have been done, ARM9EJ-S will call a LCD display program to show all decoded data on the LCD panel to demo the whole system.

This section has described the main program control flow in software level and some experimental results of the proposed system will be provided in the next section.



**Fig.6.6 Program control flow in software level**

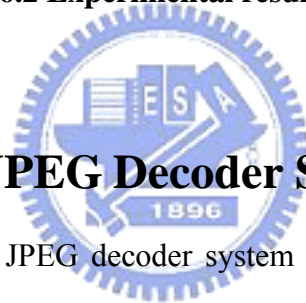
## 6.5 Experimental Results of Proposed System

This section provides some experimental results of the proposed system, as can be seen in the Table6.2. The usage of slice of FPGA is about 10% and the minimum period of the system after Place & Route is 27.6 ns (maximum frequency is about

36.23 MHz).

Device utilization Summary(FPGA)			
Number of	Design Used	Total of FPGA	Usage in Percentage (%)
Slices	3632	33792	10
Slices FLIP FLOPs	2232	67584	3
4 input LUTs	6814	67584	10
Bounded IOBs	401	1104	36
TBUFs	1	16896	0
GCLKs	2	16	12
Selected Device: 2v6000ff1517-6			
<b>P.S.: Minimum period: 27.6ns (Maximum frequency: 36.23MHz)</b>			

**Table6.2 Experimental results of FPGA**



## 6.6 Summaries of JPEG Decoder System

This chapter presents a JPEG decoder system implemented by the proposed core design which is configured in the FPGA as a specific purpose processor and all system controlled by a ARM9EJ-S processor on the develop board. All system can be divided into 3 aspects which include system level, hardware level and software level.

In system level, major components of the proposed decoder system, which is composed of an ARM9EJ-S core, a 128MB SRAM, a 64MB NOR flash, a LCD panel, and a Logic Tile which includes an FPGA, two 2MB ZBTSRAMs.

The control flow in the system level can be listed below. First, the data in a NOR flash is auto-loaded into the SDRAM as the system powered up. Second, the instructions of JPEG decoder and un-decoded JPEG files are loaded into both ZBTSRAMs on the Logic Tile. Third, ARM9EJ-S sends a signal to tell ACARM7 to

begin to decode the data in ZBTSRAM. Fourth, while ACARM7 finishes decoding procedure, it will send a finish signal to tell ARM9EJ-S. Fifth, ARM9EJ-S moves the decoded data from ZBTSRAM of the Logic Tile backing to the SDRAM. Finally, ARM9EJ-S calls the LCD display program to show decoded data on the LCD panel.

In hardware level, all major components refigured in the FPGA must connect to an AHB Bus first. These major components include the AHB TOP-LEVEL block, the AHB-APB System block, the AHB Decoder block, the AHB multiplexer block, and the AHB-Wrapper for ACARM7 block which are composed of ACARM7 with AHB interface, and two ZBTSRAM Controllers.

The core wrapped with an AHB interface is the sub-block of the AHB-Wrapper for ACARM7 block. Seven stages, Idle, Write, Read, Pre-run0, Pre-run1, Run0, Run1, are all included in the sub-block.

The control flow within the block of AHB-Wrapper for ACARM7 has been discussed in Section 6.3.3. The sub-block AHB Interface is the one discussed in Section 6.3.2. ARM9EJ-S, one SDRAM with 128MB, and two ZBTSRAM with 2MB are the other components in the AHB-Wrapper for ACARM7 block..

The program flow in software level includes the main control flow program executed by ARM9EJ-S and the JPEG instruction program executed by ACARM7 and the discussion can be founded in Section 6.4.

Section 6.5 provides some experimental results of the proposed system. The usage of slice of FPGA is about 10% and the minimum period of the system after Place & Route is 27.6 ns (maximum frequency is about 36.23 MHz).

# CHAPTER 7

## CONCLUSIONS

Chapter 1 introduces the motivation that the number of digital consumer electric products increases so dramatically nowadays and they are all powered by the batteries. Therefore, how to save more power of these portable electric devices is the most important subject in the competitive market. An ultra-low power and high-performance with small area embedded processor is proposed here. This thesis also provides a thorough and rigorous verification strategy to guarantee that the proposed embedded core has a high quality of IP characteristic. Moreover, not only considering the proposed core in the hardware aspect, but also considering the design in the system level to configure the proposed core in an FPGA as a JPEG decoder system.

Chapter 2 describes some previous works related to ARM7TDMI, which is a member of general purpose 32-bit embedded processor of the Advanced RISC Machines (ARM) family.

Chapter 3 describes the proposed core design ACARM7 which improves the operating performance as high as possible and makes power consumption as low as possible. The gate-count used in the design also quite small. The architecture of the proposed design has been discussed in this chapter and some low-power techniques have been also proposed.

Chapter 4 analyzes the experimental results among different cores inclusive of the proposed design ACARM7, ARM7 Compatible Core (Korus 2005), and ARM7TDMI. In timing comparison, the performance of the proposed design is 44.74% better compared to a normalized result from ARM7TDMI and 22.22% better than ARM7-Korus2005. In area comparison, the gate-count of ACARM7 is

much lower than the one of ARM7-Korus2005 and is 31.25% better compared to the later. In power comparison, the power of ACARM7 is improved by 57.9% compared to a normalized result from ARM7TDMI and 83% better than ARM7-Korus2005. On the other hand, the PDP of ACARM7 is 70.7% lower compared to ARM7TDMI and 86% lower compared to ARM7-Korus2005.

Chapter 5 provides a thorough and rigorous verification flow strategy in two phases, functional verification and synthesized netlist verification. The test plan and testability measurement of the proposed design are also provided in this chapter. A functional verification has been discussed that in each of steps of functional verification flow. The RTL model is verified with a systemC behavior model which is designed for matching all the cycle behaviors of ADS. All mismatches will guide the flow back to the RTL revision step for modification. The completed synthesized netlist verification flow is composed of synthesis procedure by scripts, LEC checking, gate-level simulation, and prime power. After the verification flow, it is guaranteed that the proposed embedded processor owns high quality of an IP, a characteristic of bug-free robustness, high reusability and convenient porting issue.

Chapter 6 presents a JPEG decoder system implemented by ACARM7 which is configured in an FPGA as a specific purpose processor and all system behaviors are controlled by an ARM9EJ-S processor on the development board. The system can be divided into 3 aspects which include system level, hardware level and software level. The usage of slice of FPGA is about 10% and the minimum period of the system after Place & Route is 27.6 ns (maximum frequency is about 36.23 MHz).

Based on the experiment result, the higher performance, the smaller area, and the lower power are all the advantages of the proposed processor compared with ARM7TDMI. The thesis also proposes a thorough and rigorous verification flow.

Moreover, the high applicability of the proposed processor can be demonstrated by configuring it into an FPGA for implementing a JPEG decoder system.

## **FUTURE WORKS**

A more efficient embedded system with a variety of functionality is being developed. More control mechanisms about vector interrupt controller will be developed continuously; In addition, the issue of software and hardware partition is also under research. On the other hand, more functionalities of a system about the multimedia application like MP3 decoding will be studied in the future.





# BIBLIOGRAPHY

- [1] Advanced RISC Machines Ltd (ARM) Data Sheet, *ARM7TDMI Data Sheet*, Advanced RISC Machines Ltd (ARM) Data Sheet, 1995.
- [2] Steve Furber, *ARM System-on-Chip Architecture*, Second Edition, Addison Wesley, 2000, pp.49-68, pp.74-78, pp.105-112.
- [3] Y. -C. Fong, "A High-Speed Area-Minimized Reconfigurable Adder Design," Master's thesis, National Chiao Tung University, Department of Electronics Engineering, Jul. 2006.
- [4] 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.
- [5] Dimitrakopoulos, G.; Nikolos, D., "High-speed parallel-prefix VLSI Ling adders" , IEEE Trans. Computers, vol. 54, Issue 2, pp. 225-231, Feb. 2005.
- [6] Advanced RISC Machines Ltd (ARM) User Guide, *RealView Olatform Baseboard for ARM926EJ-S HBI-0117*, Advanced RISC Machines Ltd (ARM) Data Sheet, 2003.
- [7] Advanced RISC Machines Ltd (ARM) User Guide, *Versatile/LT-XC2V4000+ Logic Tile*, Advanced RISC Machines Ltd (ARM) Data Sheet, 2002.
- [8] Advanced RISC Machines Ltd (ARM), Appl. Note119.