

國立交通大學

電信工程學系

碩士論文



應用於多媒體串流處理之可重組式
運算單元硬體加速矽智產設計

Design and Implementation of
an ALU Cluster Intellectual Property as
a Reconfigurable Hardware Accelerator for
Media Streaming Architecture

研究生：張紹宣

指導教授：闕河鳴博士

中華民國九十五年七月

應用於多媒體串流處理之可重組式

運算單元硬體加速矽智產設計

Design and Implementation of
an ALU Cluster Intellectual Property as
a Reconfigurable Hardware Accelerator for
Media Streaming Architecture

研究生：張紹宣

Student : Shao-Hsuan Chang

指導教授：闕河鳴 博士

Advisor : Herming Chiueh



A Thesis

Submitted to Department of Communication Engineering

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Communication Engineering

July 2006

Hsinchu, Taiwan.

中華民國九十五年七月

應用於多媒體串流處理之可重組式

運算單元硬體加速矽智產設計


研究生：張紹宣

指導教授：闕河鳴 博士

國立交通大學

電信工程學系碩士班

摘要



現在的生活裡，有著越來越多的行動式系統像是行動電話、MP3 播放器、PDA、以及攜帶式電子遊樂器，其功能與複雜度都較過去上升。因此，對於行動式系統來說大量的多媒體運算能力是必須的，而若是使用傳統的硬體架構來執行這些運算，會因為架構上沒有很好的對應到多媒體檔案的特性，沒辦法很有效率的對多媒體檔案作存取，會使得運算效能低落，最糟糕的是無法達到及時的需求。本論文，為參考史丹佛大學所提出的串流處理器架構，設計了對應多媒體特性的運算單元以提供所需運算能力，除此之外，在考量到了未來使用上的便利性，以及能快速與真實的多媒體應用接軌，將此運算單元加上設計的介面電路後使之成為能與 AMBA 相容的矽智產，如此便能在 ARM 的平台上，利用其他現成的矽智產或是週邊，真實的將此設計變成多媒體運算所需要的硬體加速器。本論文以軟式矽智產完成，並且經過 ARM 系列的基板驗證過所設計的介面電路，確定了所設計的介面電路是符合 AMBA 規定的電路。

Design and Implementation of an ALU Cluster Intellectual Property as a Reconfigurable Hardware Accelerator for Media Streaming Architecture

Student: Shao-Hsuan Chang

Advisor: Dr. Herming Chiueh

Department of Communication Engineering

National Chiao Tung University

Hsinchu, Taiwan

Abstract

There are more and more portable systems such as mobiles, MP3 player, PDA, and other entertainment systems in today's life. The functionality and complexity of them thus increase much higher than old-time ones. Therefore, having a great deal ability of multimedia operation is important for portable systems. However, it is tough to have enough amounts of multimedia operations from conventional hardware architecture. This results from the poor match between conventional architecture and features of media applications. It hence leads to inefficient memory access that induces performance degression. The worst case is unable to meet the real time requirement.

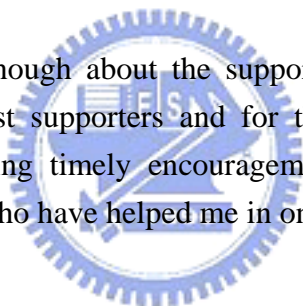
According, this thesis designs an operational unit, ALU cluster, that is referenced from Stanford's stream processor architecture and thus matches to media applications to provide necessary processing requirements for media applications. Besides, considering the issues of convenient usage in the future and rapid integration of real multimedia applications, we wrap ALU cluster as an AMBA-compatible IP by adding designed interface. Then, it is possible to exploit other existing IP and peripherals in the AMBA platform and truly treats our design as hardware accelerator for real multimedia applications. This thesis is finished with a synthesizable soft IP. The designed interface is verified by ARM-series baseboard. This ensures that the interface conforms to AMBA specification.

ACKNOWLEDGMENTS

This thesis would not have been possible without the support of many exceptional people. First and foremost, thanks go to my research advisor, Professor Herming Chiueh. He has always been an inspiration to me and everyone else on this project through his vision and leadership. He also provided irreplaceable guidance for me when I needed for a fascinating problem, good advice, constructive criticism, support, and flexibility.

I would also like to thank all team members of the SoC LAB group, especially my classmates over the years: We-Li Su and Joseph Tsai. They not only put up with me all of those years, but also made my days as an enjoyable graduate student.

Finally, I can not say enough about the support provided by my family. My parents have been my biggest supporters and for that I am forever grateful. My girlfriend has always providing timely encouragement and advice. To all of my friends and family members who have helped me in one way or another over the years, I would like to say thanks.



CONTENTS

摘要	I
Abstract	II
Acknowledgement	III
List of Table	VI
List of Figure	VII
Chapter 1 Introduction	1
1.1 Issues of Media Applications	1
1.2 Proposed design: An ALU Cluster Intellectual Property (IP)	3
1.3 Organization	5
Chapter 2 Background	6
2.1 Reconfigurable Architectures for Media Applications	6
2.1.1 Stream Processor Architecture	7
2.1.2 Stream Processing Model	8
2.1.3 Implementations of Stanford's Stream Processor Architecture	9
2.2 Design Methodology in the SoC Era	11
2.3 Overview of AMBA	13
2.3.1 Overview of AMBA AHB	15
2.3.2 Bus Connection	16
2.3.3 Signals of AHB Interface	17
2.3.4 Address Decoding	18
2.3.5 Basic Transfer	19
2.3.6 Transfer Type	20
2.3.7 Burst Operation	21
2.4 Overview of Emulation Environment	23
2.5 Summary	25

Chapter 3 Design and Implementation of an ALU Cluster Intellectual Property	26
3.1 An ALU Cluster	27
3.1.1 Architecture of an ALU cluster	27
3.1.2 Implementation Results	28
3.1.3 Chip Testing	31
3.1.3.1 Testing Environment	32
3.1.3.2 Testing flow and result	33
3.2 Design and Emulation for the AHB Slave Wrapper of Intellectual Property	35
3.2.1 Architecture of AHB Slave Wrapper	35
3.2.1.1 Finite State Machine of AHB Slave Wrapper	36
3.2.2 Modifications for Baseboard and Data Preparing	38
3.2.3 Emulation Result	40
3.3 An ALU Cluster Intellectual Property	42
3.3.1 Architecture of an ALU Cluster Intellectual Property	42
3.3.2 Functional Verification	43
3.3.2.1 Testbench: 16-tap FIR filter System	44
3.3.2.2 Simulation Results	45
3.3.3 Improvements from ALU Cluster to ALU Cluster IP	49
3.3.4 Extension of ALU cluster IP at magnetic RAM (MRAM)	51
3.3.4.1 Overview of MRAM	51
3.3.4.2 Modifications for MRAN	51
3.3.4.3 Implementation Result	52
3.4 Implementation Comparisons	54
3.5 Summary	56
Chapter 4 Conclusion and Future Work	57
Bibliography	59
Appendix A: Assembly Code for Chip Testing	62
A.1: Assembly Code for FIR	63
A.2: Access Method for Memory Testing	66
Appendix B: Memory Map	69

LIST OF TABLES

Table 2.1	SPECIFICATION OF STREAM PROCESSOR ARCHITECTURE	10
Table 2.2	BURST SIGNAL ENCODING	22
Table 2.3	ACTICE BYTE LANES FOR A 32-BIT LITTLE ENDIAN DATA BUS	23
Table 2.4	ACTICE BYTE LANES FOR A 32-BIT BIG ENDIAN DATA BUS	23
Table 3.1	SUMMARY OF AN ALU CLUSTER	29
Table 3.2	TESTING RESULTS	34
Table 3.3	MODIFICATIONS FROM ALU CLUSTER TO ALU CLUSTER IP	49
Table 3.4	SUMMARY OF IMPLEMENTATION RESULTS	53
Table 3.5	TABLE OF IMPLEMENTATED COMPARSIONS	55
Table A.1	THE OPERATIONS CORRESPONG TO THE ALU UNIT	62
Table A.2	THE OPERATIONS CORRESPONG TO THE MUL UNIT	62
Table A.3	THE OPERATIONS CORRESPONG TO THE DIV UNIT	62
Table A.4	ASSEMBLE CODE FOR FIR	63, 64, 65
Table A.5	ACCESS TABLE FOR ODD BANK MEMORY	66, 67
Table A.6	ACCESS TABLE FOR EVEN BANK MEMORY	67, 68

LIST OF FIGURES

FIGURE 1.1.1	PROCESSOR-MEMORY GAP	2
FIGURE 1.2.1	PROPOSED ALU CLUSTER IP	4
FIGURE 2.1.1	STREAM PROCESSOR ARCHITECTURE	8
FIGURE 2.1.2	STREAM PROCESSING MODEL OF FIR FILTER SYSTEM	9
FIGURE 2.1.3	CHIP IMPLEMENTATION OF STREAM PROCESSOR ARCHITECTURE	10
FIGURE 2.2.1	AN ASIC DESIGN	11
FIGURE 2.2.2	AN EXAMPLE OF PLATFORM DESIGN	13
FIGURE 2.3.1	A TYPICAL AMBA SYSTEM	15
FIGURE 2.3.2	DIAGRAM OF BUS CONNECTION	16
FIGURE 2.3.3	DIAGRAM OF AHB SLAVE	17
FIGURE 2.3.4	SIGNALS TO SELECT WHICH SLAVE	18
FIGURE 2.3.5	SIMPLE TRANSFER	19
FIGURE 2.3.6	TRANSFER WITH TWO WAIT	20
FIGURE 2.3.7	TWO KINDS OF BURST OPERATION	22
FIGURE 2.4.1	VERSATILE PLATFORM BASEBOARD FOR ARM926EJ-S	24
FIGURE 2.4.2	ARCHITECTURE OF VERSATILE PLATFORM BASEBOARD	25
FIGURE 3.1.1	ARCHITECTURE OF AN ALU CLUSTER	27
FIGURE 3.1.2	LAYOUT OF AN ALU CLUSTER	29
FIGURE 3.1.3	FLOORPLAN AND PAD ASSIGNMENT OF AN ALU CLUSTER	30
FIGURE 3.1.4	DIE MICRO PHOTO OF AN ALU CLUSTER	30
FIGURE 3.1.5	PACKAGE OF AN ALU CLUSTER	31
FIGURE 3.1.6	AN ALU CLUSTER ON THE PCB BOARD	32
FIGURE 3.1.7	LOGIC ANALYZER SYSTEM WITH LCD MONITOR DISPLAY	33
FIGURE 3.1.8	TESTING FLOW OF CHIP LEVEL TESTING	34
FIGURE 3.2.1	ARCHITECTURE OF AHB SLAVE WRAPPER	35
FIGURE 3.2.2	FINITE STATE MACHINE	36
FIGURE 3.2.3	PORTS MODIFICATIONS	39
FIGURE 3.2.4	MODIFICATIONS WITH ADDED REGISTER FILES	40
FIGURE 3.2.5	SOFTWARE CODES FOR ADS	42
FIGURE 3.2.6	ADDRESS DEFINITION	42

FIGURE 3.2.7	EMULATION RESULTS	43
FIGURE 3.3.1	ARCHITECTURE OF AN ALU CLUSTER IP	43
FIGURE 3.3.2	COEFFICIENTS OF THE FIR FILTER SYSTEM AND ITS INPUT FUNCTION	44
FIGURE 3.3.3	EXPECTED OUTPUT RESULTS OF THE FIR FILTER SYSTEM	45
FIGURE 3.3.4	FULL WAVEFORM VIEW OF SIMULATION	46
FIGURE 3.3.5	DETAILED WAVEFORM IN EXECUTION STAGE A	47
FIGURE 3.3.6	DETAILED WAVEFORM IN EXECUTION STAGE B	47
FIGURE 3.3.7	DETAILED WAVEFORM IN READING STAGE A	48
FIGURE 3.3.8	DETAILED WAVEFORM IN READING STAGE B	48
FIGURE 3.3.9	MODIFICATIONS FOR MRAM WITH ADDED LOAD_STORE UNIT	52
FIGURE 3.3.10	LAYOUT OF AN ALU CLUSTER IP EXTENDED AT MRAM	53
FIGURE 3.3.11	FLOORPLAN OF AN ALU CLUSTER IP EXTENDED AT MRAM	54



CHAPTER 1

Introduction

With the technology improving, there are many media applications related to our life such as mobiles, MP3 player, PDA, and other portable entertainment system. It is hard to avoid the usage of these products because it is convenient to use them for watching movies, listening to music, and even playing video games at anywhere and anytime. The functionality of them keeps growing day by day. Thus, the processing requirements of media applications are more and more important right now. However, the conventional architecture with only single processor is hard to handle all jobs by itself. Therefore, another solution according to the features of media applications must be brought up to overcome the growing processing requirements.

1.1 Issues of Media Applications

It is known that media applications have three main characteristics: large available parallelism, little data reuse, and a high computation to memory access ratio [1] [2]. The first one, large available parallelism, is due to each streaming is independent to others. Thus, each streaming is possible to be operated solely at the same time. Little data reuse results from the streaming element reading from the memory only accesses memory once and do not revisit again, causing poor performance to the cache in the conventional architecture. Large amount of data operations result in high computation to memory access ratio is needed is the third characteristic. All these characteristics are poorly matched to conventional architectures and the performance will be extremely awful while media applications running on conventional ones. It must have another dedicate design to solve these issues by multiple operational units and efficient bandwidth hierarchy.

From the media characteristics, we know memory access is the most serious problem for processing of media applications. In the mean time, the processor and memory performance gap that reveals the performance growing of memory is much slower than processor, as shown in Figure 1.1.1 [3]. This phenomenon will cause more latency for memory access. That means communication between processor and memory is more precious.

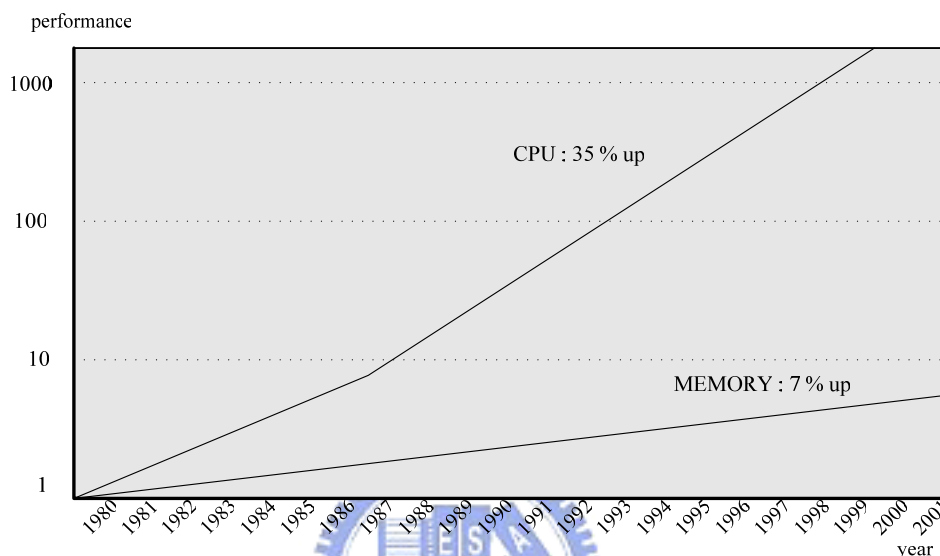


Figure 1.1.1: Processor-Memory Gap

Fortunately, in the modern VLSI technology, as the process goes down to the deep submicron, it is possible to put lots of arithmetic logic unit (ALU) on a single chip in low cost [4] [5] [6]. It means that many operations are able to be executing at the same time by lots of functional units with little area cost. It is a great help for parallel processing. Summing up features of media applications and evolution of technology, there are many researches that are trying to improving the disability of conventional architecture by their own architectures such as Application Specific Integrated Circuit (ASIC), Platform-Based Architecture, and Reconfigurable Architecture. ASIC is specific to one application. It thus can have most balance to power, area, and performance. However, it lacks of flexibility to reuse design while the specification changes. Platform-Based Architecture consists of a processor core, memory, peripherals and other intellectual property (IP) with the same interface as hardware accelerator. By changing different IP, the same platform can be reused for distinct usages. The IP is mostly from ASIC by adding interface to solve different purposes. Although the flexibility of Platform-Based Architecture is better than ASIC, it still needs to change different IP for varied purposes. It is a characteristic, supporting application is limited, from ASIC. Therefore, Reconfigurable Architecture

is another choice. It will use many general purpose operational units. As the demand is modified, the operational units are modified by reconfiguration, too. It therefore uses the same hardware to achieve different purposes. Details about architectures listed above and design methodology will be addressed in Chapter 2.

1.2 Proposed design: An ALU cluster IP

According to the issues of media applications and three associated architectures, we choose IP as a our target to process media applications, because we can get best trade-off from some physical issues, reuse other ready components to simplify developing efforts, and make our IP reusable for further possible development. In order to provide a more powerful ability for our IP, we reference from Stanford's stream processor that is a reconfigurable architecture and thus develop our IP as a reconfigurable hardware accelerator for media applications [7].

The stream processor architecture is designed for media applications. Therefore, it matches the features of media ones and can handle them very well. However, it is a complete and huge system. Thus, it requires many human resources and much time to implement it. As a result, we select the processing part, ALU cluster, to be wrapped as an IP. Then, it can reduce efforts for developing a architecture to deal with media applications and be reused in the further. Details about stream processor architecture will be demonstrated in Chapter 2. The proposed design is shown in Figure 1.2.1. It is an ALU cluster IP that is compatible to AMBA-based platform. This platform may contain some available blocks such as processor, memory, other peripherals, and our intentional IP that is wrapped from an innovated ALU cluster from the stream processor architecture, as shown in the bottom part of the figure.

Therefore, our proposed IP can be divided as hardware and software parts. The hardware part includes AMBA-AHB wrapper, ALU cluster, and memory. As for software, it must have architecture simulator to decide the number of internal operational units, depth of internal memory, and so on. The ALU cluster and architecture simulator are finished by our seniors last year [8] [9]. Taking these two previous designs as basement, some improvements are added to increase its performance that will be shown in Chapter 3.

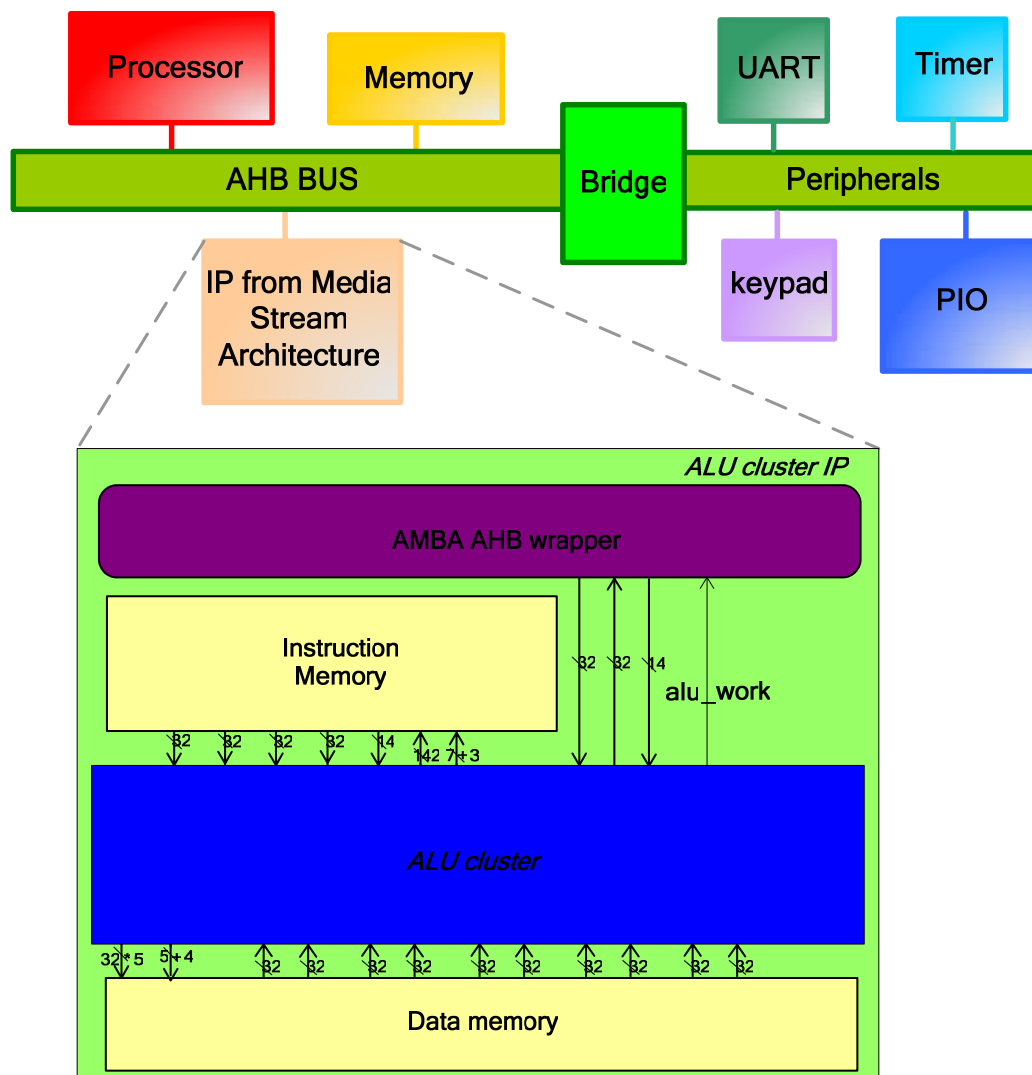


Figure 1.2.1: Proposed ALU Cluster IP

The advantages of the usage of the AMBA-based platform will be addressed in the Chapter 2. One major reason for choosing AMBA bus as a platform is because we take most care of connecting our design with real media applications. This can be benefited from the complete environment of AMBA bus. Except for the free specification of AMBA bus, there are already many baseboards and software that help to accelerate the design procedure. With these baseboards, there are many ready IPs, including processor core, memory, and peripherals, can be used directly. Besides, they are controlled by adequate software. Therefore, these hardware and software are helpful to accelerate development. Details about the used platform and baseboard are in Chapter 2.

1.3 Organization

The motivation for the thesis is introduced in this Chapter. The remaining of organization about this thesis is as follow. In the Chapter 2, the first part will introduce architectures mentioned above for media applications, and the second part will discuss the design methodology. This section induces us to establish AMBA-base platform. Then, the overview of AMBA on-chip bus and AHB slave protocol will be presented. The last section is the introduction of emulation environment

The Chapter 3 is the main chapter about the design and implementaion of the ALU cluster IP. The first section demonstrates the implementation results of ALU cluster, prototype I, and its testing outcomes. The second part is the hardware emulation of AHB wrapper by ARM baseboard. The following is the architecture, simulation results, and modifications of the ALU cluster IP. The last one is about IP that is extended to implement a real chip with magnetic RAM (MRAM) as data memory [10]. Then, the conclusion and future work will be addressed in the last chapter.



CHAPTER 2

Background

In this chapter, the related review of background about two architectures are used to deal with media applications, design methodology, AMBA specification of AHB protocol for the design, and used platform baseboard in this thesis is described. The first section will address reconfigurable architectures for media applications; they are DIVA and stream processor architecture [7] [11]. We will focus on stream processor architecture that is what we choose to further development. The second section is about the trend of design methodology, it reviews from ASIC to platform-based architecture. This section induces us to establish an AMBA-based platform that will advantage to further development. The third section is overview of the AMBA on-chip bus and the AHB slave protocol will be introduced most. As for the other details about AMBA bus, they could be found in the AMBA specification. The last section is about emulation environment that is our used platform baseboard in this thesis.

2.1 Reconfigurable Architectures for Media Applications

As we know from the introduction, media applications need an architecture that is different from conventional one to increase the processing requirements. Two kinds of architectures will be shown up. First, DIVA architecture will be introduced shortly. Then, the stream processor architecture is addressed.

Brief speaking, DIVA is an architecture that integrates processor logic and memory in a processing-in-memory (PIM) chip. The advantages of its architecture is that internal PIM processors are directly connected to the memory banks, the memory bandwidth is dramatically increased (up to 2 orders of magnitude, tens or even hundreds of gigabits aggregate bandwidth on a chip). Latency to on-chip logic is also

reduced, down to as little as one-fourth that of a conventional memory system, because of the usages of internal memory access without the associated delays by communicating to off chip. It thus overcomes the large amount of memory demands and provides the ability to handle media applications. However, this architecture must have completely mechanism to communicate between each PIM. This will cause that extra efforts are needed to investigate the communications. Besides, the communicated protocol is not a general one that could not be portable to other hardware. Thus, it is not worthy to put much more efforts on it. We therefore choose stream processor as our reference architecture for further development.

2.1.1 Stream Processor Architecture

By trading off from features of media applications and growing technology, Stanford proposes the programmable stream processor architecture with three accents that are locality, concurrency and bandwidth hierarchy to better the processing performance of media applications. The stream processor architecture is shown in Figure 2.1.1 [12] [13]. It consists of ALU clusters, stream register file, streaming memory system, stream controller, micro controller, and host interface. The core of working part of that architecture is the eight ALU clusters. These eight ALU clusters achieve one of the features, the concurrency processing. They are controlled by microcontroller in single instruction multiple data (SIMD) manner. Therefore, all ALU clusters getting the same instructions from microcontroller and operate on different data streaming. Besides, these are very long instruction word (VLIW) instructions, performing compound stream operations on each streaming elements.

There are many embedded local register files in the ALU clusters. The embedded local register files make the processing in local and accomplish the feature of locality. The streaming memory system is the interface to connect off chip SDRAM and is able to solve rare data bandwidth and provide ability to schedule dynamically. The stream register file is the novel organization of high performance memory pool and is used to store streaming of any length. Together with local register file, the stream register file can efficiently re-circulate streaming with different ALU clusters and perform the last characteristic of bandwidth hierarchy by storing the data from the streaming memory system to itself. By programming varied instructions, it is possible to execute different media applications. Stanford use stream processing model to map the media application into the stream architecture. The stream programming model allows simple control, makes explicit communication, and exposes the inherent parallelism of media applications. It will be discussed in next section.

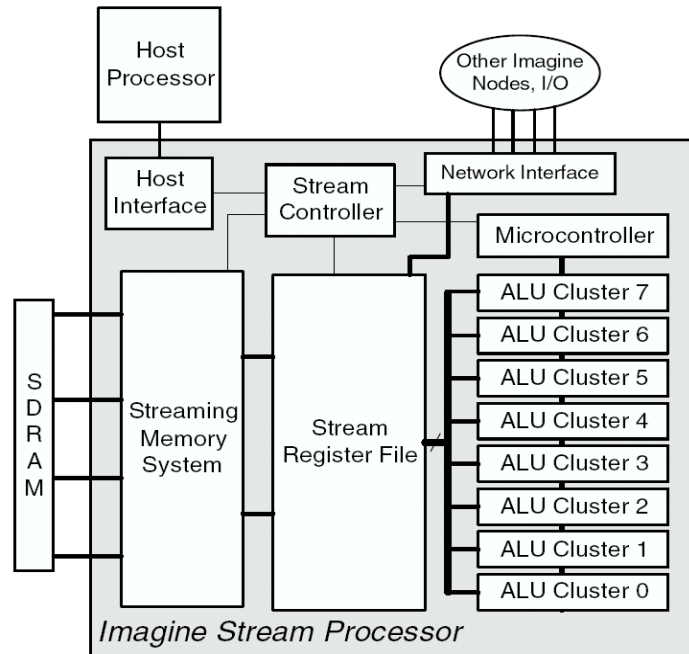


Figure 2.1.1: Stream Processor Architecture

2.1.2 Stream Processing Model

The processing flow of media applications is certainly like a sequence of computation kernels that operate on long data streaming. The kernel is a small program that repeats its operation for the continuous input streaming and produces the output streaming for the following subsequent kernels. The streaming is a variable length collection of records and these records are logical grouping of media data. An example of stream processing model is shown in Figure 2.1.2. It is one kind of media applications, the finite impulse response (FIR) filter system [14]. An FIR filter is a one dimensional convolution. It only has two kinds of kernel, MUL and ADD which are also the common operations in the media operations, over a long data streaming. Thus, the FIR filter system is a good demonstration for stream processing model. The equation 2.1 describes its operation:

$$y[n] = \sum_{k=0}^{M-1} b_k * x[n-k] \quad (2.1)$$

The $y[n]$ represents the output data streaming of the FIR filter system. It is the weighted average of the of input data streaming, $x[n]$, from $x[n]$ to $x[n-M+1]$, with the weights given by the coefficients b_k . The coefficient of n represents its instant time for both $x[n]$ and $y[n]$. The coefficient of k stands for the number of taps in the filter, and b_k is n th tap coefficient.

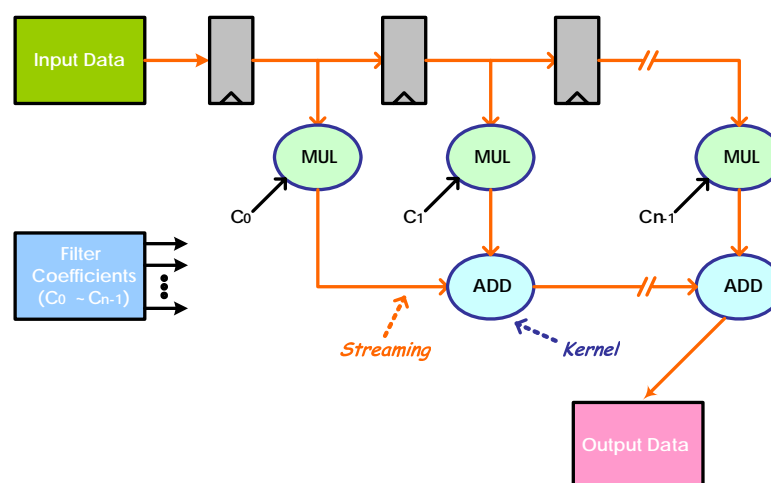


Figure 2.1.2: Stream Processing Model of FIR Filter System

The kernel in the stream processing model reveals the possibility to process media applications: parallel and pipeline execution. Multiple kernels can activate at the same time on independent streaming data. The front side kernel handles its input streaming and produces the output streaming and the streaming data is then passed to be processed by the next kernel. Therefore, the kernels are not only parallel but also pipeline to cope with the media data. Since all data are organized as streaming, only single memory access from off chip is required and the succeeding transfers are from register to register. It is helpful to optimize for bandwidth requirement. As a result, the memory bandwidth demands of the media applications can be solved by exploiting this stream processing model.

Mapping the media applications to proper kernels and streams expose the characteristics that are enumerated in previous section, large available parallelism, little data reuse, and high computation to memory access ratio, so that the hardware can easily exploit them and achieve the needed performance requirements.

2.1.3 Implementations of Stanford's Stream Processor Architecture

We now take a look at Stanford's work. The chip is shown in Figure 2.1.3. The eight ALU clusters are in the right side, and the microcontroller is on top of them. The stream register file is at the center. The left side is the interface to off-ship DRAM. The other modules are in the top side. The chip is placed similar to the block diagram as shown Figure 2.1.1.

The specification is shown in table 2.1. It is fabricated in the 1.5V 0.15 μm process with five layers of aluminum metal by Texas Instruments (TI). The peak

performance is at 180M Hz. The power dissipation running at 180M Hz is around 8 to 10W, and the best power-efficient is at the operating point of 1.2v at 96M Hz. The total gate count is 21 million, and the chip area is 2.6 cm².

As shown in figure and table, it is obviously that the whole architecture of the stream processor architecture is complete and huge. The inventor, Stanford, spends eight graduate students to complete all the architecture including design and verification. As shown in the Figure 2.1.3, there are many different modules of different functionality. The module level test must be exercised completely to ensure the pre-defined functionality is correct. Then, it is necessary to verify the integration of these modules. However, the fully system simulation takes much time and needs high level cycle accurate simulation model to shorten it. This model will also require some human resources. Aside from these issues, how to map the stream processor architecture to operating system (OS) and perform a real application on the stream processor architecture are the other two subjects. As long as running real media applications on the design, it thus is convincible.

Table 2.1: Specification of Stream Processor Architecture

Process	TI 0.15 um (1.5V)
Performance	180 MHz
Power (At 180 MHz , 2.0 V)	8~10 W
Power (At 96 MHz , 1.2 V)	1.6~1.9W
Gate count	21-million
Area	2.6 cm ²

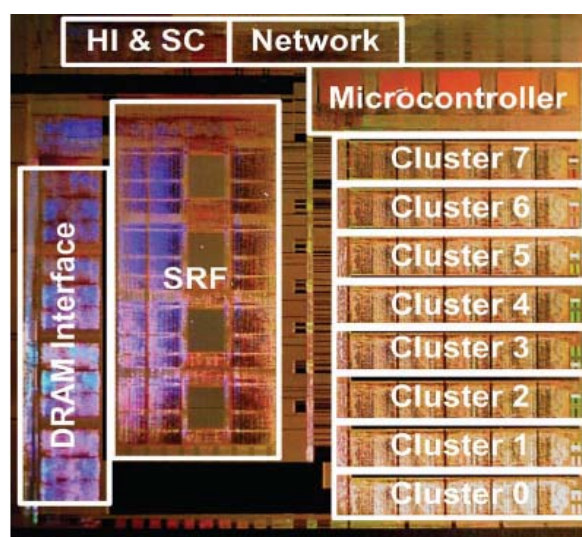


Figure 2.1.3: Chip Implementation of Stream Processor Architecture

2.2 Design Methodology in the SoC Era

In today's technology, although the development methodology is different for system designers and processor designers as well as between DSP developers and chipset developers, there are some common problems in designing chips [15].

- Time-to-market pressures
- Difficulty in verification of increasing chip complexity
- Difficulty in timing closure while entering deep submicron sizes
- Difficulty in integrating different levels and areas expertise

Taking ASIC as example, it is the conventional design methodology. However, in the SoC era, more and more transistors are placed into the chip causing the complexity of chip increases dramatically higher than used to. This results in problems listed above get worsen and loss competitiveness if the ASIC design is not able to come out betimes. The basic principle of ASIC is shown in Figure 2.2.1. The chip implementation could be finished as quickly as soon as the well-defined specification. The physical area, operating frequency, and power consumption are able to be optimized as being defined in the specification. The advantage of ASIC is that it could design a specific architecture that fully matches the pre-defined specification and thus incurs the best trade off between performance, power, and area. This is why the ASIC is still popular now day. However, it has a serious drawback of lacking of flexibility for reusing the design. As long as the specification changes, the chip must also change.

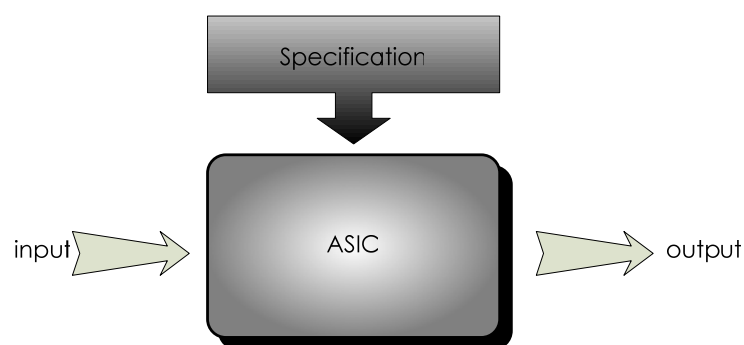


Figure 2.2.1: An ASIC design

Platform-based architecture aims the target of design reuse. It consists of a core and many pre-defined robust intellectual property (IP, sometimes called cores, block, or macro) as hardware accelerator that have already been designed and verified solidly with the same interface. There are some existing platforms are developed, such

as AMBA on-chip bus from ARM, device control bus (DCR) from IBM [16], and automated bandwidth allocation bus system from Silicon Backplane of Sonics [17]. The topologies of them are similar to each other. They communicate through the share bus with the same interface of protocol, although the needed standard interface will extend the design cycle and have extra overhead of timing and area.

With the same communicating protocol, it could be helpful to integrate many different modules tightly. By adding or deleting some IP could set up a different system as fast as possible. Therefore, if there is already plenty of reusable IP with the same interface, it can reduce much time to re-develop a new system depending on different requirements. One more attractive thing is that these platforms have been set up with a developing baseboard. There are many common IP such as processor, memory, and peripherals on the baseboard that benefits to fast prototyping. Forbye, they have OS in existence and can reduce the effort of connecting the real applications to intent design.

Furthermore, it makes the physical design problems, interconnect issue, clock skew, power consumption, etc, be engaged in the early stage of design process. The pre-defined IP not only can solve the physical problems early but also is easy to verify in advance. It changes the global problems to local ones to be dealt with. Therefore, because of lots of advantages of this design methodology, it gradually becomes a trend in the SoC era. Not only it is able to ease off the common problems but also has possibility to low down overall cost by reusing ready IP.

There is one more thing needed to be noticed in order to finish a platform design, only preparing robust IP for reuse is not enough. The designers have to provide other things that are complete documents and enough deliverables, such as functional, timing, synthesis, physical models and so on. Otherwise, the effort to integrate a pre-existing block into new designs will still be prohibitive high.

Here is simple example to illustrate platform-based design, as shown in Figure 2.2.2. It is a simplified block diagram of cell phone system that consists of a processor, on-chip memory, DSP core, baseband codec, LCD controller, and USB controller and each block has the same protocol. The blocks communicate with others by bus through the same interface. It is obviously that there are many blocks in this platform are able to be reused in other system. For example, the USB controller can be reused directly in another design such as a MP3 player if they have the same interface.

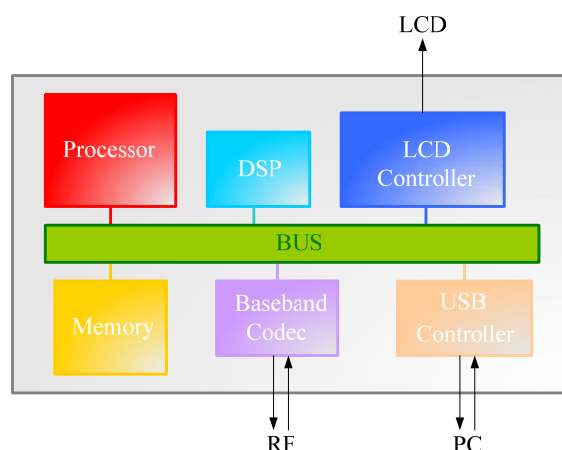


Figure 2.2.2: An Example of Platform Design

2.3 Overview of AMBA

The AMBA specification defines an on-chip communications standard for designing high-performance embedded microcontrollers [18] [19]. There are four objectives of the AMBA on-chip bus system; right-first-time, technology-independent, modular system design, and minimization the silicon infrastructure.

The AMBA system could be used to facilitate the right-first-time development of embedded microcontroller products with one or more CPUs or signal processors. Being technology-independent and ensuring that highly reusable peripheral and system macrocells can be migrated across a diverse range of IC processes and be appropriate for full-custom, standard cell and gate array technologies. Improving processor independence and providing a development road-map for advanced cached CPU cores and the results from the modular system design. Then, it minimizes the silicon infrastructure requirement and supports efficient on-chip and off-chip communication for both operation and manufacturing test. There are three distinct buses are defined with the AMBA system.

➤ **The *Advanced High-performance Bus (AHB)***

The AHB is for high performance, high clocking frequency system module. It acts as the high-performance system backbone bus and provides the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions. It is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques.

➤ **The Advanced System Bus (ASB)**

The AMBA ASB is also for high-performance system modules. It is an alternative system bus suitable for use where the high-performance features of AHB are not required. ASB also supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral macrocell functions as AHB does.

➤ **The Advanced Peripheral Bus (APB).**

The AMBA APB is for low-power peripherals. It is optimized for minimal power consumption and reduces complexity of interface to peripheral functions. APB can be used in conjunction with either version of the system bus.

An AMBA-based microcontroller typically consists of a high-performance system backbone bus (AMBA AHB or AMBA ASB), which is able to sustain the external memory bandwidth, on-chip memory and other *Direct Memory Access* (DMA) devices reside. This backbone bus is able to provide a high-bandwidth interface between the elements that are involved in the majority of transfers. APB is a lower bandwidth communication bus and is also located on the high performance bus by the bridge. Most of the peripheral devices in the system are located in APB and are accessed through the bridge, as shown in Figure 2.3.1: A Typical AMBA SystemFigure 2.3.1.

The characteristics of AMBA AHB, ASB, and APB are listed in the bottom of the figure. Basically, AHB has many advanced features such as pipeline operation, multiple bus masters, burst transfer, and split transactions. ASB is the former protocol compared to AHB by lacking the burst transfers and split transactions. APB is defined according to the attribute of low power issue. It latched address and control signal to save power. The interface is simpler than the other two. It thus is suitable for many peripherals.

There is one thing different between APB and ASB or AHB. ASB and AHB have ability to wait the transfer while it is not ready whether the wait situation is from on-chip bus or itself. APB must response the transaction immediately for avoiding the loss of utility for the on-chip bus.

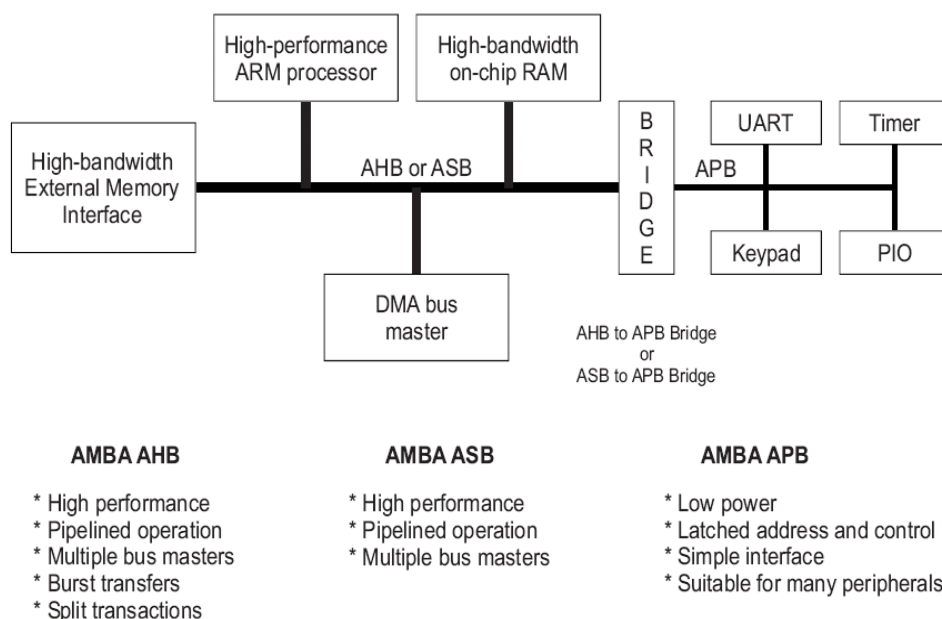


Figure 2.3.1: A Typical AMBA System

2.3.1 Overview of AMBA AHB

AHB is one kind of AMBA bus which is intended to address the requirements of high-performance synthesizable designs. It sits above the APB and implements the features required for high-performance, high clock frequency systems including: burst transfers, split transactions, single cycle bus master handover, single clock edge operation, and non-tristate implementation. A typical AMBA AHB system design contains the following components: master, slave, arbiter, and decoder.

➤ AHB master

A bus master is able to start the reading and writing operations by providing appropriate address and control information to slaves. Only one bus master is allowed to actively use the bus at any one time, otherwise, the data may crash in the common bus.

➤ AHB slave

AHB slave responds the reading or writing operation according the information from master within a given address-space range. The bus slave signals back to the active master the success, failure or waiting of the data transfer.

➤ AHB arbiter

The existence of AHB arbiter means the multiple AHB masters and slaves are available in the AHB system. The AHB arbiter has the responsibility to ensure that only one bus master at a time is permitted to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as *highest priority* or *fair* access can be implemented depending on the application requirements. The AHB would include only one arbiter, although this would be trivial in single bus master systems.

➤ AHB decoder

The AHB decoder is used to decode by address of each transfer and provide a signal to select which slaves that is involved in the transfer. A single centralized decoder is required in all AHB implementations.

In the following subsection, the bus connection, signals of AHB interface, basic transfer, transfer type, and burst operation will be presented. The other details about arbitration will be in the AMBA specification.

2.3.2 Bus Connection

The AMBA AHB bus protocol is designed to be used with a central multiplexer interconnection scheme rather than the tri-state interconnection method, as shown in Figure 2.3.2. By using this scheme, all bus masters drive out the address and control signals indicating the transfer they wish to perform. Then, the arbiter receives the information from masters and determines which one has the ability to get the access privilege. Then, the master routes address and control signals to all of the slaves. However, only one slave will start to be accessed by the signal from arbiter. A central decoder is required to handle the necessary control of reading data and multiplexer to response correct signal, which selects the appropriate signals from the slave that is involved in the transfer.

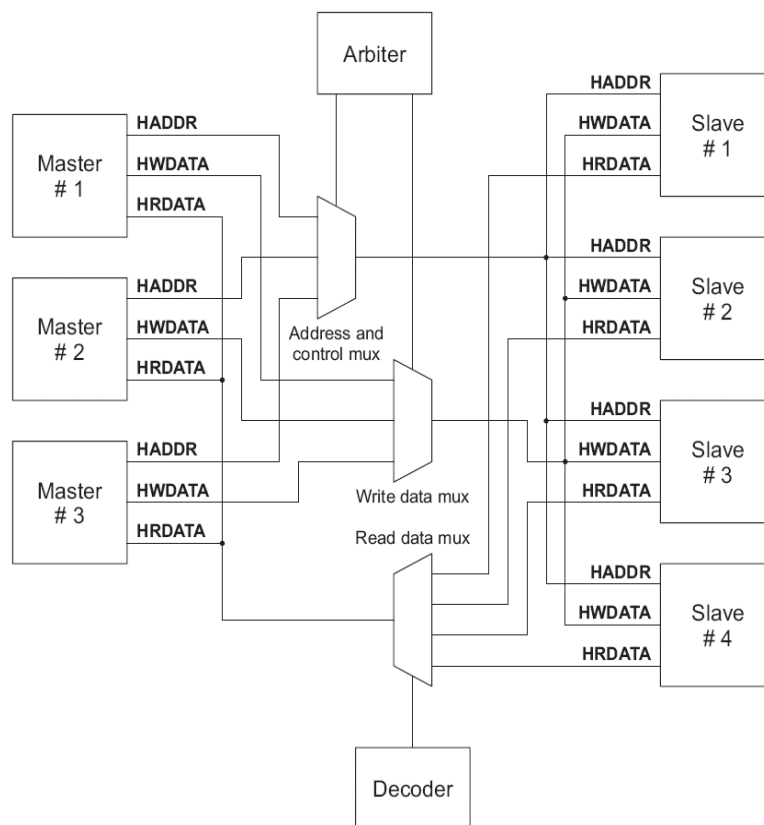


Figure 2.3.2: Diagram of Bus Connection

2.3.3 Signals of AHB Interface

The diagram of AHB slave is depicted in Figure 2.3.3. It shows total input and output ports. The bandwidth demonstrates here is 32 bits. Actually, the AMBA system is flexible to other bandwidth, such as 64, 128, or more bits are possible. The input ports could be divided into six groups according to its functionality; select, address and control, data, reset, clock, and split capable input. The select signal, HSELx, is from the arbiter to enable AHB slave to work. The address and control signals, HADDR, HTRANS, HSIZE, and HBURST, are together from AHB master. The master forwards the requirements to slave depending on these two kinds of information. Details will be presented in the following section. The data, HWDATA, clock, HCLK, and reset, HRESTn, are the common signals as we know. The last one group only exists in split capable slave. This group of signals is used to support the split transaction.

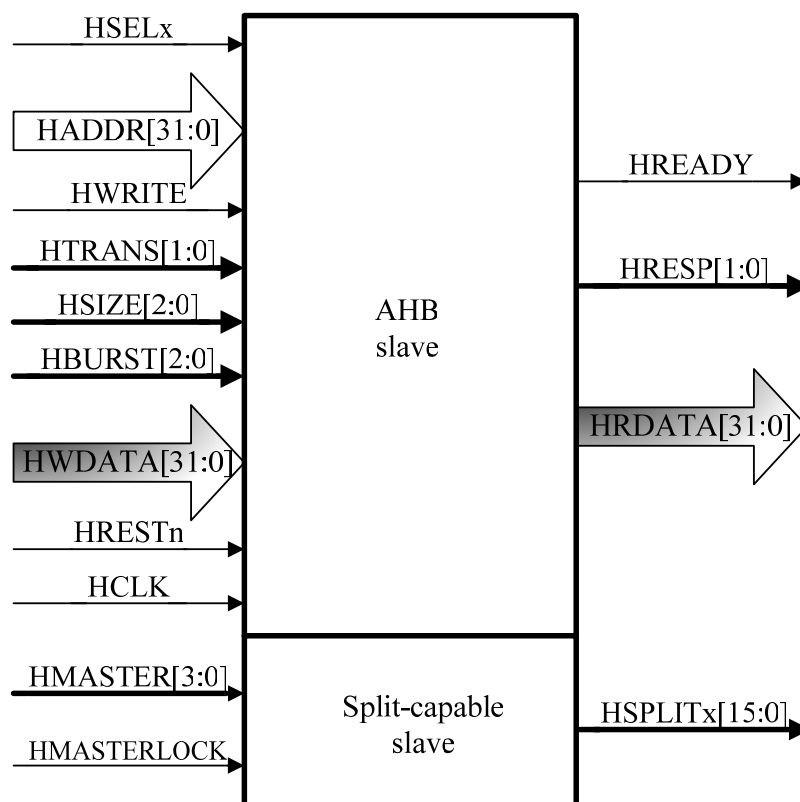


Figure 2.3.3: Diagram of AHB slave

The output ports are able to separate in three groups; transfer responses, data, and split capable output. The response signals, HREADY and HRESP, are the information that the slave feeds back to the master. They are used to judge if the transaction is finished correctly or failed. The data, HRDATA, is what master willing to read. HSPLIT_x is also used while the slave supports the split transaction. In fact, the AMBA specification doesn't stress on this point. The usage of split transfer type depends on the designer's intent.

2.3.4 Address Decoding

The signal HSEL_x is used to enable slave and is provided by the decoder, as shown in Figure 2.3.4. The decoder decodes the high-order address signals and then chooses one of the slaves to active. The minimum address space that can be allocated to a single slave is 1kB. All bus masters are designed such that they will not perform incrementing transfers over a 1kB boundary, thus ensuring that a burst never crosses an address decode boundary.

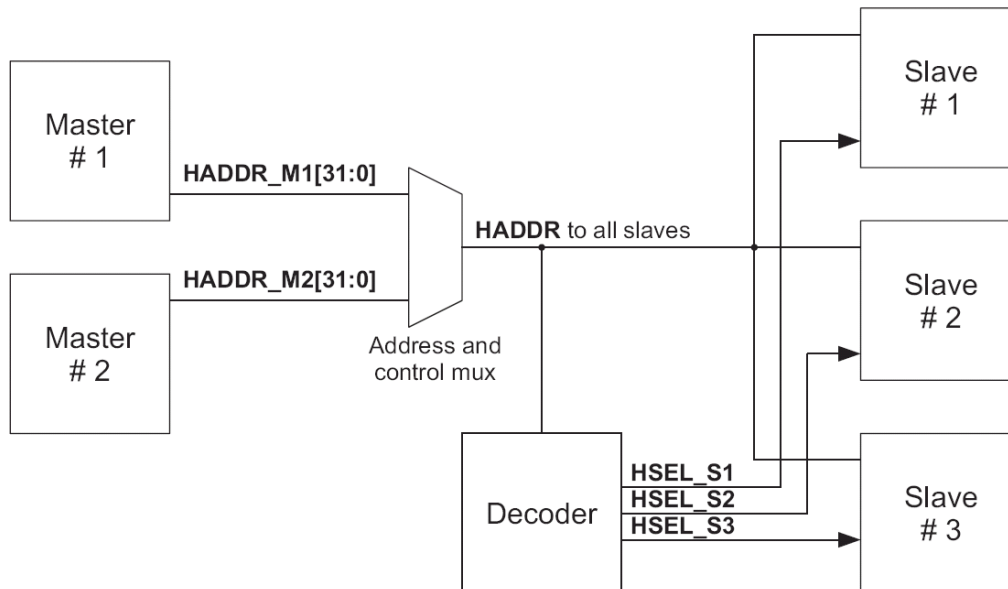


Figure 2.3.4: Signals to Select Which Slave

2.3.5 Basic Transfer

The transfer of AHB protocol is based on the pipeline operation. It separates the data and address phase into two stages, as shown in Figure 2.3.5. The necessary information including the HADDR and control, which are HADDR, HBURST, HSIZE, HTRANS, and HWRITE, will be broadcasted by master in the first stage. The second stage is data phase that passes or receives data in several cycles depending on the HRADY and HRESP.

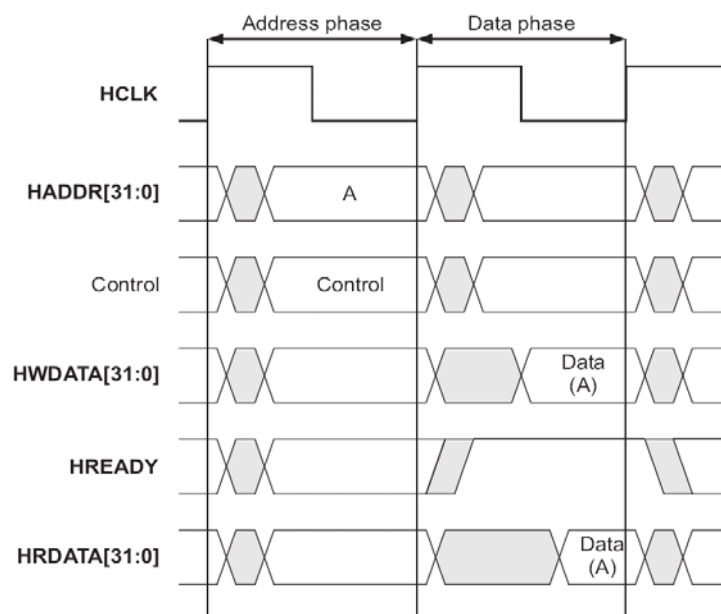


Figure 2.3.5: Simple Transfer

The HREADY indicates the transfer state. The logic high represents that it is ready to be finished and the logic low is contrast to be extended. Figure 2.3.6 is the example that the transfer is extended with two cycles. The first stage is address phase the same as shown in Figure 2.3.5. However, in the second stage, the logic low for HREADY means the transfer is not ready to be completed. This may result from both master and slave depends on the transfer type, HTRANS. It will be introduced in the later. Therefore, by using the HREADY, the transfer is lengthened to ensure the transaction could be terminated with expected results.

2.3.6 Transfer Type

Every transfer can be classified into one of four different types, as indicated by the HTRANS[1:0] signals. They are IDLE, BUSY, NONSEQ, and SEQ and controlled by 2'b00, 2'b01, 2'b10, and 2'b11 respectively. They are listed as below.

➤ IDLE

It indicates that no data transfer is required and is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.

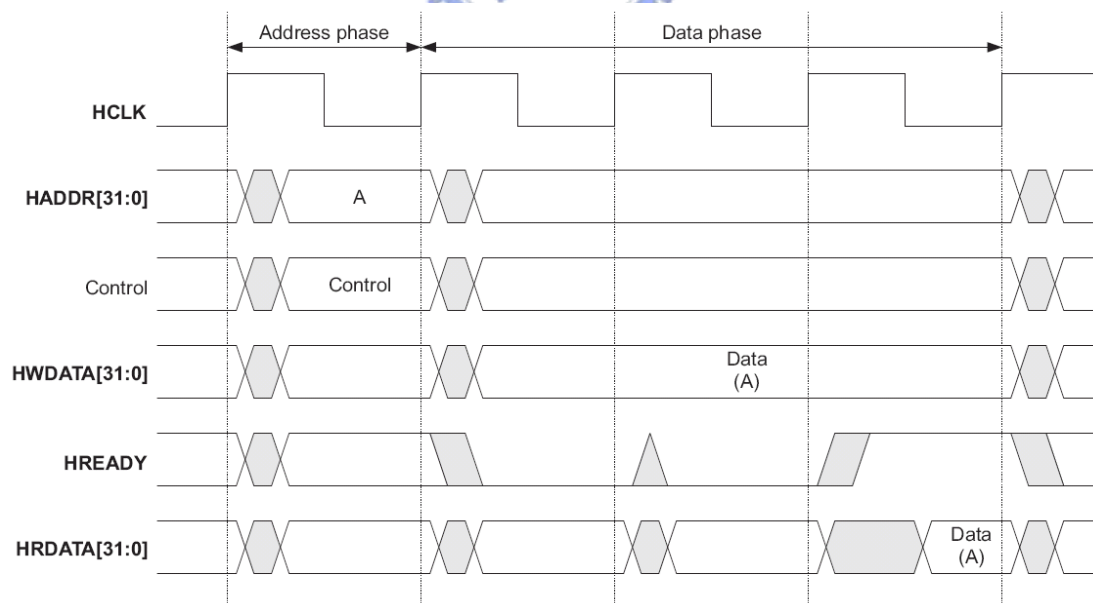


Figure 2.3.6: Transfer with Two Wait

➤ BUSY

The BUSY transfer type allows bus masters to insert idle cycles in the middle of bursts of transfers. This transfer type is used while the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.

➤ NONSEQ

This transfer type represents the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Because single transfers on the bus are treated as bursts of one, therefore the transfer type is also NONSEQUENTIAL.

➤ SEQ

The remaining transfers in a burst are SEQUENTIAL and the address must be related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes) in the incrementing burst. In the case of a wrapping burst, the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

2.3.7 Burst Operation

Burst information is provided by the use of HBURST and the eight possible types are shown in Table 2.2. Incrementing burst is the sequential access of locations and the address of each transfer in the burst is just an increment of the previous address depends on the transfer size.

For wrapping burst transfers, if the address of the first transfer is not aligned to the total number of bytes in the burst (size x beats), then the address of the transfers in the burst will wrap when the boundary is reached. For example, a four-beat wrapping burst of word (4-byte) accesses will wrap at 16-byte boundaries. Therefore, if the start address of the transfer is 0x34, then it consists of four transfers to addresses 0x34,

0x38, 0x3C and 0x30. The last transfer will wrap back to 0x30. As shown in Figure 2.3.7, the read rectangular is the incrementing burst and the blue rectangular is the wrapping burst which is depicted as the blue arrow.

Table 2.2: Burst Signal Encoding

HBURST[2:0]	Type	Description
000	SINGLE	Single transfer
001	INCR	Incrementing burst with undefined length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

The transfer size is controlled by HSIZE and supports eight dimensions; 8, 16, 32, 64, 128, 256, 512, and 1024 bits. It thus may suffer issues of alignment, as shown in table 2.3 and 2.4. The selection between big-endian and little-endian are defined in the specification directly and depend on the designer's decision. As long as the designs all follow the same endian, it will not be a serious problem.

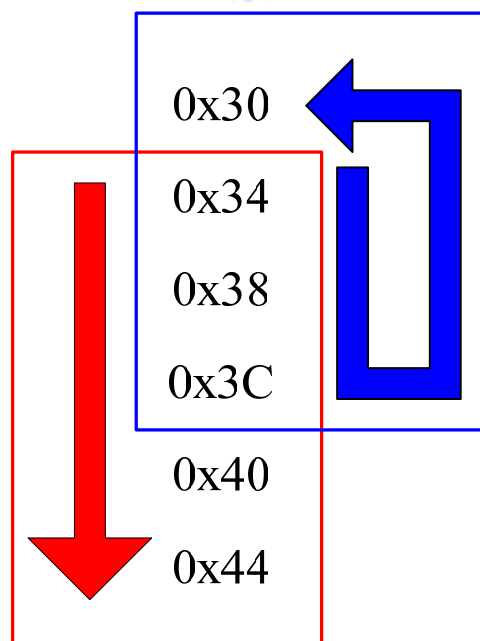


Figure 2.3.7: Two Kinds of Burst Operation

Table 2.3: Active Byte Lanes for a 32-bit Little-Endian Data Bus

Transfer size	Address offset	Data [31:24]	Data [23:16]	Data [15:8]	Data [7:0]
Word	0	✓	✓	✓	✓
Halfword	0	-	-	✓	✓
Halfword	2	✓	✓	-	-
Byte	0	-	-	-	✓
Byte	1	-	-	✓	-
Byte	2	-	✓	-	-
Byte	3	✓	-	-	-

Table 2.4: Active Byte Lanes for a 32-bit Big-Endian Data Bus

Transfer size	Address offset	Data [31:24]	Data [23:16]	Data [15:8]	Data [7:0]
Word	0	✓	✓	✓	✓
Halfword	0	✓	✓	-	-
Halfword	2	-	-	✓	✓
Byte	0	✓	-	-	-
Byte	1	-	✓	-	-
Byte	2	-	-	✓	-
Byte	3	-	-	-	✓

2.4 Overview of Emulation Environment

In this thesis, the protocol verification is by the RealView Versatile Platform Baseboard for ARM92EJ-S (VPB926EJ-S), as shown in Figure 2.4.1 [20]. It is the first baseboard in the Versatile family and is designed for ASIC emulation and prototyping. By using the baseboard, it takes great advantage of verifying validity of AMBA protocol, reducing the overall time about integrating our IP with others, and finally running real media applications on it.



Figure 2.4.1: Versatile Platform Baseboard for ARM926EJ-S

This baseboard is designed specifically for ASIC emulation and prototyping, and supports advanced 3D graphics application development around ARM and PowerVR MBX cores. It also provides an AMBA Multi-layer AHB prototyping environment. Thus, we can prototype our design to verify the correctness of protocol. The architecture is shown in Figure 2.4.2.

The upper part in the Figure is a powerful processing core; development chip. It can integrate high performance IP such as memory and DMA controllers and the ARM VFP9-S coprocessor around the ARM926EJ-S core. The vector processing capability of the ARM VFP9-S coprocessor offers increased performance for imaging applications such as scaling, 2D and 3D transforms, font generation, and digital filters. The development chip also includes an implementation of the ARM MOVE coprocessor which significantly improves the motion estimation capability required for applications like MPEG encode through hardware assistance for sum-of-absolute differences (SAD) calculations. The bottom part is the FPGA and it can be expanded with logic tile. They are the programmable parts for HDL languages and share a common bus to communicate with other peripherals such as USB, I/O, Mse, Kbd, and etc. One different feature of the logic tile compared to FPGA is that it is able to communicate with on-chip memory as well as development chip. As a result, the logic tile, that we select is LT-XC2V8000 [21], is also needed for possible development in the future. It is a Virtex-II FPGA product of Xilinx and supports up to 8M gate counts. The detail information can be found in their individual document.

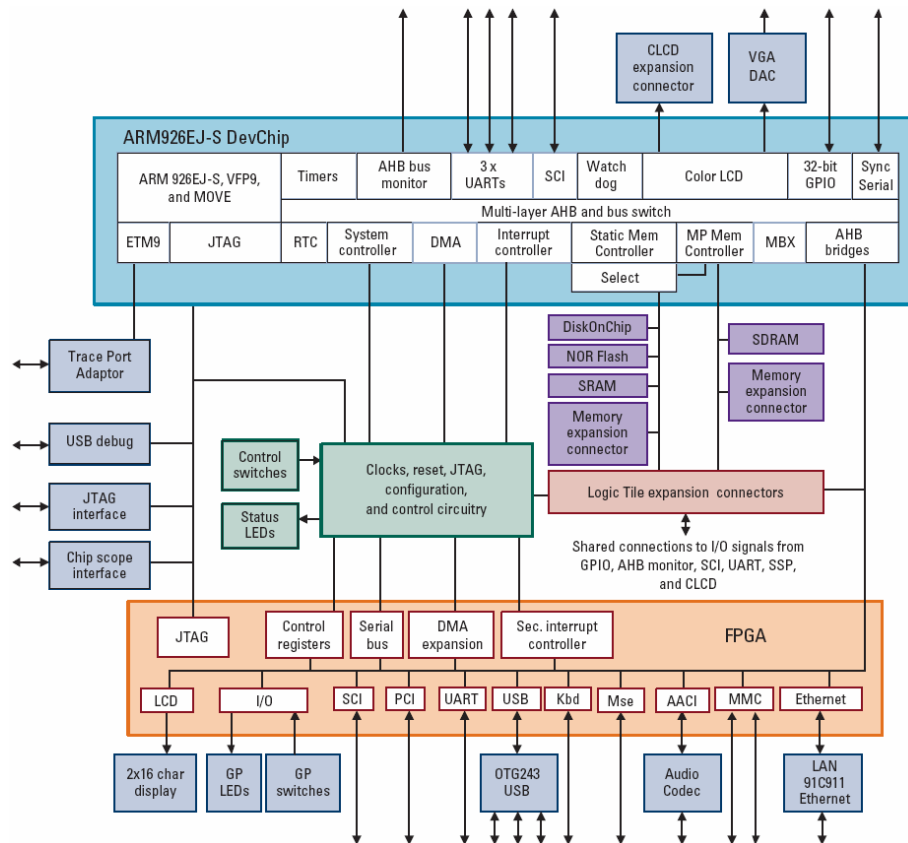


Figure 2.4.2: Architecture of Versatile Platform Baseboard

2.5 Summary

We have introduced four background references to finish this thesis. These references include the reasons, related knowledge, and verified ways to design our IP for media applications. The first two are a reconfigurable architecture for media applications and design methodology. According to these two references, we thus propose our designed IP. As a result, there is an overview about the protocol of our proposed IP in the following. Then, the last one is about the emulation environment that is used to verify the correctness of the protocol for our design.

CHAPTER 3

Design and Implementation of an ALU Cluster Intellectual Property

According to the previous two chapters, the motivation and necessary background are viewed. Thus, the design and implementation of proposedd ALU cluster IP will be shown up in this chapter. Our IP consists of AHB slave wrapper, ALU cluster, and memory. Memory is the easiest component and can be instanced from memory compiler. Thus, it will not be mentioned frequently in this chpater. The design and implementation of the other two components are the key points in this chapter. The first section is the demonstration of the previous ALU cluster. It has been tapped out by UMC with 0.18 um process and Artisan design kit. Therefore, the die photo and summary of characteristics are listed in the beginning. The chip is then tested by Agilent's tools. Through the testing of real silicon, it is convinced that the ALU cluster is practicable to handle media applications.

The second section is about the design of the AHB slave wrapper which stands for interface between ALU cluster and AMBA bus. It will accept control signals from AMBA bus and manage the ALU cluster to execute its function. It is then programmed into the ARM series baseboard for hardware emulation. This will verify the correctness of protocol of the proposed AHB slave wrapper.

The third section is about the ALU cluster IP that is a complete integration of improved ALU cluster and designed AHB slave wrapper together. The improved ALU cluster is referenced from first section and is designed and implemented to advance its performance. The results of simulation will be introduced then. After the successful

integration, a synthesizable IP is finished. Then, there is one extension of this IP. The main core of our IP is unchanged and the data memory is replaced by a novel storage, magnetic RAM (MRAM). The last two sections are the implementation comparisons and summary about the development

3.1 An ALU Cluster

The first part of this section is the brief report of the previous design, an ALU cluster, including the architecture, related layout, pad assignment and floorplan, die photo, and circuit summary. The testing is in the second part. It contains the testing environment, testing flow, and testing results. After the chip is tested, the ALU cluster is convincing that it is able to handle media applications as we expect.

3.1.1 Architecture of an ALU cluster

The architecture of an ALU cluster is shown in Figure 3.1.1. It consists of two ALU of two-stage pipeline, two multipliers of four-stage pipeline, one divider taking sixteen cycles to finish operation, a scratch pad register file (SPRF), ten banks of intra register file, a controller, and a decoder. The ALU can execute up to thirteen instructions; they are ADD, SUB, ABS, AND, OR, XOR, NOT, SLL, SRL, SRA, LT, GT, and EQ. The MUL can perform multiplication. The DIV can carry out three jobs: they are division that gets the quotient and remainder and find the square root. There are intra register files (IRF) embedded in the operation unit. The IRF are registers local to the operation unit. They can store streaming temporary. The SPRF is used to store coefficients using in common media applications and is used by ALU, MUL, and DIV to communicate with each other.

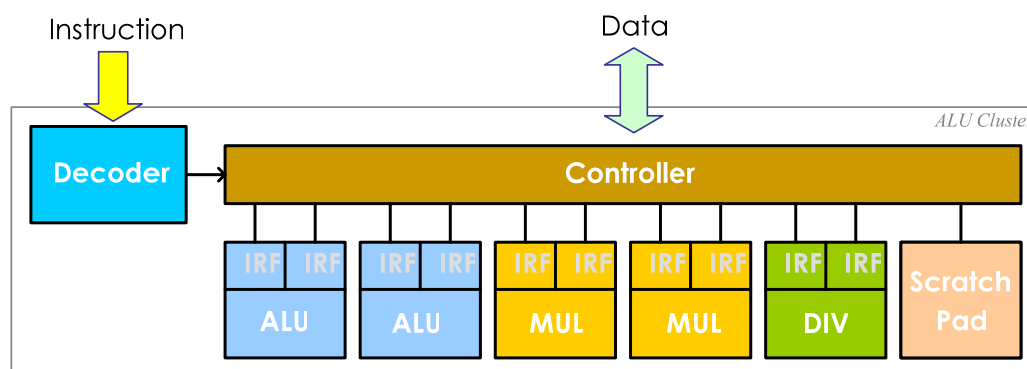


Figure 3.1.1: Architecture of an ALU Cluster

The decoder can parse the instructions and provides control signals to control the ALU cluster. When the ALU cluster is in working state, the controller decides which input sources, IRF, SPRF, and data memory, to the functional unit. When it works in reading or writing state, the controller manages the data flowing to or from IRF, SPRF, and outer data memory.

3.1.2 Implementation Results

The summary of the circuit characteristics of the ALU cluster is listed in Table 3.1. The 0.18 μm process of UMC and cell based design kit of Artisan are utilized for the implementation. The operating frequency of post-layout simulation is 100 MHz. The chip size and core size are about $3 \times 3 \text{ mm}^2$ and $2.2 \times 2.2 \text{ mm}^2$. The gate count of the core is 411491. It contains fifteen banks of memory for data and instruction. They are four 32×128 single port static RAM (SRAM), one 14×128 single port SRAM, and ten 32×32 single port SRAM. They are generated by memory compiler with Artisan library. The memory of 128 entries is used for instruction memory and they could support output bandwidth of 142 bits per cycle to the VLIW instructions. The data memory has ten banks and each bank has 32 entries. They can work together to provide up to the bandwidth of 320 bits per cycle. The power dissipation is 968.35 mW for the total chip when runs the media example of FIR filter that will have been introduced in the first chapter. The pure core size without these two kinds of memory is about $1.8 \times 1.2 \text{ mm}^2$. Its gate count is 255669 and the power is down to 312.38mW. The physical layout of the ALU cluster is shown in Figure 3.1.2.

The core utilization is close to 88.8%. The data memory is the bottom rectangular and the instruction memory is the left rectangular. It can be seen clearly in Figure 3.1.3 about the floorplane and the pad assignment. There are total 127 input/output (I/O) pads, where 47 input pads, 32 output pads, and 48 power pads. The die microphotograph is shown in Figure 3.1.4. It is packaged with CQFP128 and its photograph is shown in Figure 3.1.5

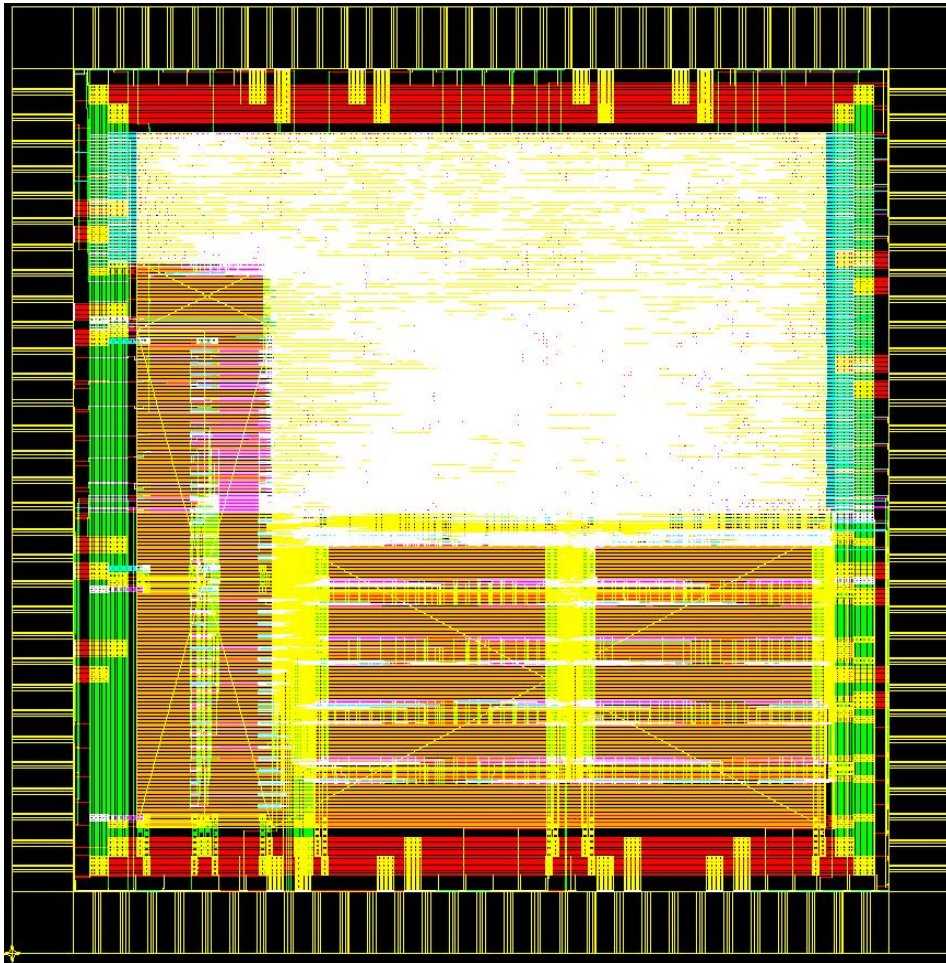


Figure 3.1.2: Layout of an ALU cluster

Table 3.1: Summary of an ALU Cluster

Process	UMC 0.18 um
Library	Artisan SAGE-x Standard Cell Library
Post-layout Clock Rate	100 MHz
Chip Size	2.98 x 2.98 mm ²
Core Size (without memory)	2.2 x 2.2 mm ² (1.8 x 1.2 mm ²)
Gate Count (without memory)	411491 (255669)
Power Dissipation (without memory)	968.35 mW (312.38 mW)
On-chip memory	10 32x32 single port SRMA 4 32x128 single port SRMA 14x128 single port SRAM
Pad	Input: 47 pins Output: 32 pins Power: 48 pins

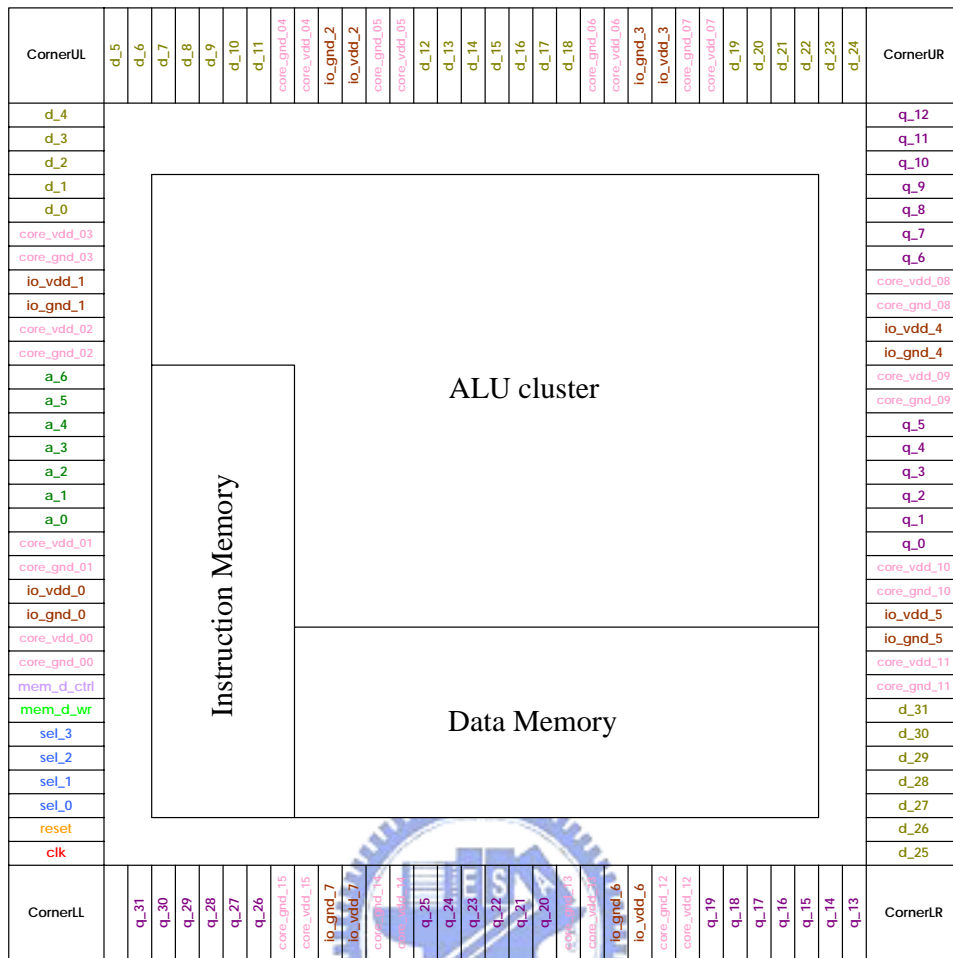


Figure 3.1.3: Floorplan and Pad Assignment of an ALU Cluster

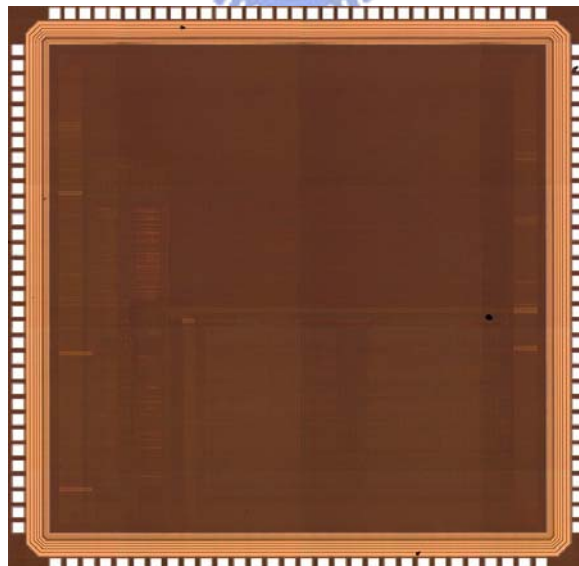


Figure 3.1.4: Die Micro photo of an ALU Cluster

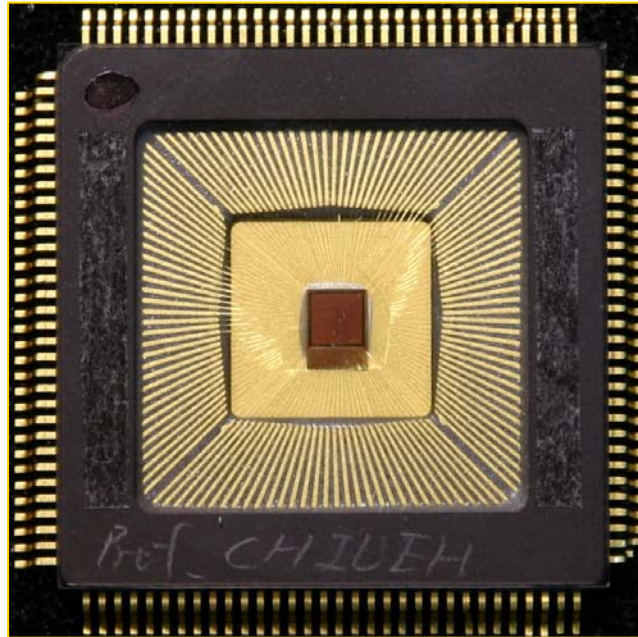


Figure 3.1.5: Package of an ALU Cluster

3.1.3 Chip testing

There are two main testing targets for the chip of the ALU cluster; they are functionality and performance. The testing of functionality is divided into three parts which are memory, instruction, and real program testing. The performance testing is by the 16-tap FIR filter. The memory testing is divided into five stages as listed below. The purpose of using these five stages is to find out the possibility of stuck at one and stuck at zero failures. Isolating the memory testing is because memory is the most common failure part of the chip.

- Write all memory 32'hFFFF-FFFF, and then read them.
- Write all memory 32'h0000-0000, and then read them.
- Write all memory 32'hAAAA-AAAA, and then read them.
- Write all memory 32'h5555-5555, and then read them.
- Write all memory with interleaving 32'hAAAA-AAAA and 32'h5555-5555, and then read them.

All bits of memory are written by one and zero in the first two stages. In the third stage, one and zero in interleaving manner are written to the memory. The fourth stage is the opposite of the third stage which are zero and one instead of one and zero. Two continuous data with 32'hAAAA-AAAA and 32'h5555-5555 are written in the fifth stage. This stage cause the continuous data is interleaving with one and zero.

We then test functionality of instruction by feeding the random sources into all operational units. All instructions of different operation units must be gone through at least once to ensure the correctness. The last testing target is the real program by using 16-tap FIR filter as testbench that is the same benchmark of performance testing. The details will be in Appendix A.

3.1.3.1 Testing environment

The environment for the chip testing is shown in Figure 3.1.6 and Figure 3.1.7. The chip is placed on the PCB board as shown in Figure 3.1.6. Most of the pins are placed in the order sequence that helps to chip testing. Figure 3.1.7 shows what environment we use. It is Agilent 16902A Logic Analysis System together with Agilent 16720A pattern generator and Agilent 16910A logic analyzer module [22]. This logic analyzer system can support up to 48 pins' pattern input from the pattern generator module and catch up to 64 pins' output results by logic analyzer module. The maximum frequency support to half pins' usage of the pattern generator is 300MHz and 180MHz in full pins' usage. It thus is adequate for the testing of our chip.

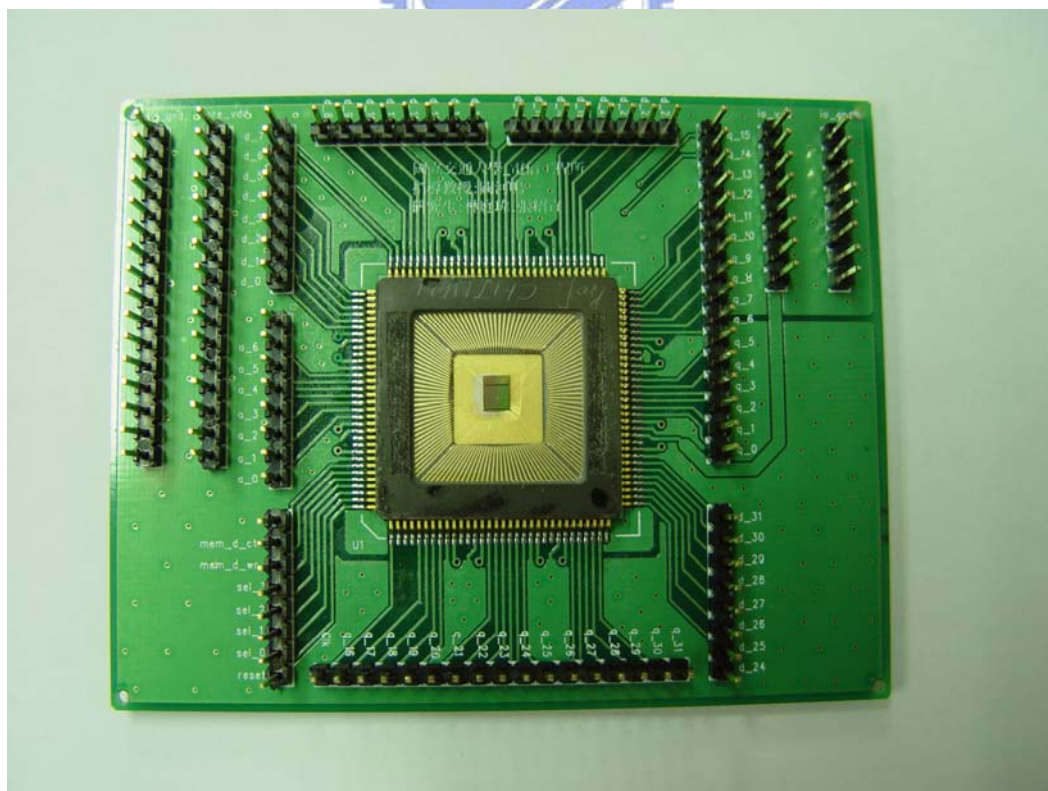


Figure 3.1.6: ALU Cluster on the PCB board

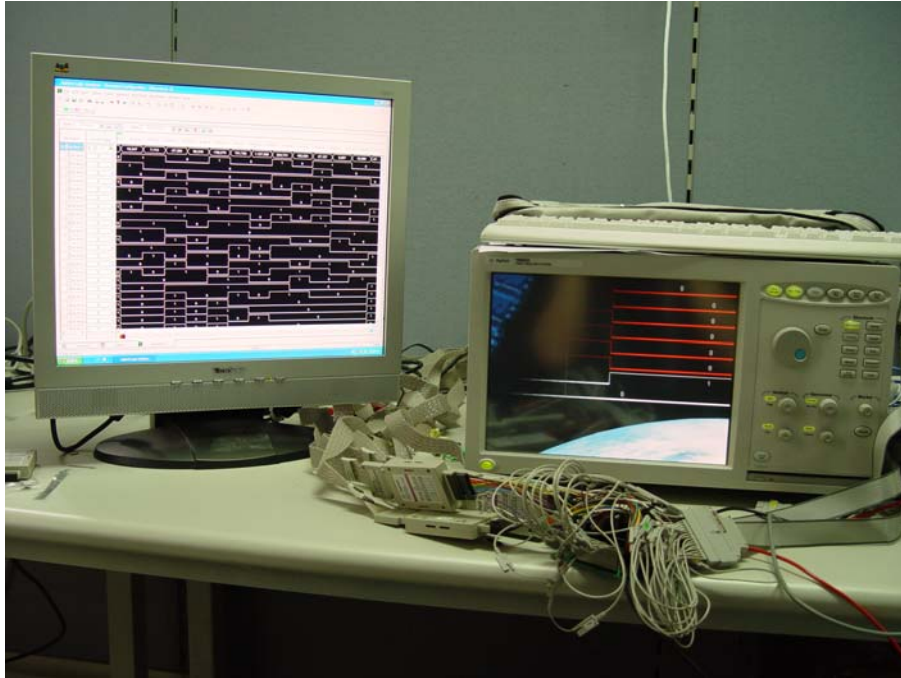


Figure 3.1.7: Logic Analyzer System with LCD Monitor Display

3.1.3.2 Testing flow and results

The testing flow is shown in Figure 3.1.8. It is divided into three stages. The first stage is in the upper rectangular for preparing test pattern. Because the visibility of chip testing is very low, it is not easy to judge the sources of the problems. We thus run the testing target at the software level simulation to avoid the possibility of invalid testing pattern. The simulation results are then record and prepared to be compared with the outcomes of chip testing. The necessary format of test pattern for the pattern generator is produced with the simulation at the same time. The second stage is in the middle rectangular that is chip testing in the Logic Analysis System. The pattern generator module produces the same stimulus as simulation for the chip and then the logic analyzer module receives the results after chip processes the incomes. Therefore, the fruits from the logic analyzer module are compared with the ones from simulation in the third stage.

After the outcomes are compared between software simulation and hardware testing, the results are listed, as shown in Table 3.2. The clock rate of memory and instruction testing is lower down to 1MHz for simplifying the possible problems from chip. Through the testing flow, we make sure that all the memory banks including data and instruction memory are correct. Also, the instructions of different functional units are true. The outcomes of program of FIR filter system are correct, too.

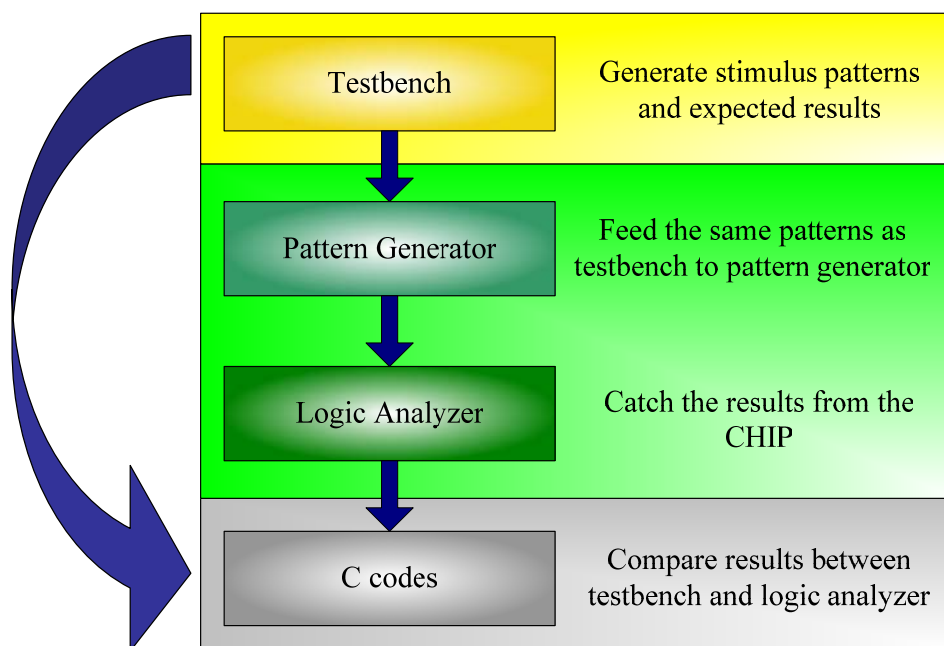


Figure 3.1.8: Testing flow of chip level testing

Table 3.2: Testing Results

Memory	Data	Correct
	Instruction	Correct
Instruction Functionality	ALU0	Correct
	ALU1	Correct
	MUL0	Correct
	MUL1	Correct
	DIV0	Correct
FIR	Correct around 16~18 MHz	

However, the performance of the FIR filter system is down to around sixteen to eighteen MHz. It is more than five times slower than the post-layout simulation. The reason of this performance loss is because of the large loading through the probe lead set and the PCB board. It reminds us that it is necessary to connect the pod of Agilent to the chip directly in the future. Otherwise, it is hard to get the true performance of the chip from the huge loading.

3.2 Design and Emulation for the AHB Slave Wrapper of Intellectual Property

In this section, there are three parts to demonstrate the design and hardware emulation. The architecture of the AHB slave wrapper is showed first. It shows up the functionality of the wrapper and how it controls the ALU cluster to execute media applications by finite state machine. Then, it will be programmed into ARM series baseboard to proceed hardware emulation for verifying its protocol. However, there are some necessary modifications for the baseboard, it will be displayed in the second parts. The last one is the emulation results. According to the hardware emulation, we can make sure that the correctness of protocol for proposed wrapper is verified.

3.2.1 Architecture of AHB Slave Wrapper

The architecture of the proposed wrapper consists of two components that are finite state machine (FSM) and address generation unit (AGU), as shown in Figure 3.2.1. The FSM is in charge of receiving the signals from the AMBA bus and administrating the IP to obey the protocol of AHB slave. It will tell the AGU to produce the necessary address whether the ALU cluster is accessed in bursting incrementing or wrapping mode. There is one signal, *alu_work*, from the ALU cluster to FSM. It will be used to identify if the ALU cluster finishes executing the media applications or not. With the necessary information from the AMBA bus and ALU cluster, the FSM just can control the ALU cluster including, write data, read data, and execute instruction.

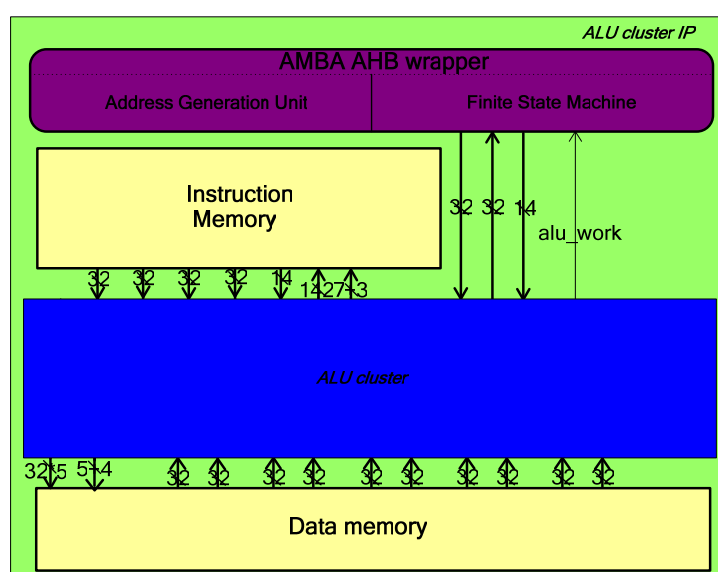


Figure 3.2.1: Architecture of AHB Slave Wrapper

3.2.1.1 Finite State Machine of AHB Slave Wrapper

The FSM of the wrapper is used to control the states of the IP and response the request of AMBA bus, which is used to dealing with issues such as reading and writing with on-chip bus and activating the ALU cluster. It has the responsibility to ensure that the IP is able to communicate with other modules from the AMBA bus. As a result, it must have some prescribed response back to AMBA bus. Also, it will identify the input signals from the AMBA bus and tell the ALU cluster if it has to execute the applications or still in its original state. Therefore, the FSM is designed with six states; they are Idle, Accessible, ALU_Work, Un-readable Wait, Un-writable Wait, and Error, as shown in Figure 3.2.2.

➤ Idle:

When the IP is not accessed and the ALU cluster finishes its executing, the FSM will be in the Idle state. It will go to other states while the bus is granted and the IP will be accessed or the ALU cluster is activated. Not only until IP has done the employment but also suffers some error, it will come back to the Idle state. In this state the wrapper will be ready to get the signals from AMBA bus and prepare next work. Basically, only while the HTRANS is equal to NONSEQ, it has the chances to move to other states. Otherwise, it will keep the Idle state. If the NONSEQ signal is encountered, it identifies which operation the IP is requested by HWRITE. Then, the FSM can move to its target state.

➤ Accessible

The FSM will directly move to Accessible state if the HTRANS is equal to NONSEQ and HWRITE is high. In the Accessible state, the FSM continuously checks if the IP is accessed repeatedly. It will be two different kinds of accessing, but cause FSM still in the Accessible state. One of them is that HTRANS is still equal to NONSEQ. It means the the IP is accessed with different address. The other is that HTRANS is equal to SEQ. It represents that previous access is continuous with burst mode of wrapping or incrementing manner.

There are three cases that the FSM will move away; the HTRANS changes to IDLE or BUSY and the reading data is not ready. These three cases mean that the access is finished, busy to write, and ready to read but data is not ready and will induce the state move to Idle, Un-readable Wait, and Un-writable Wait, respectively.

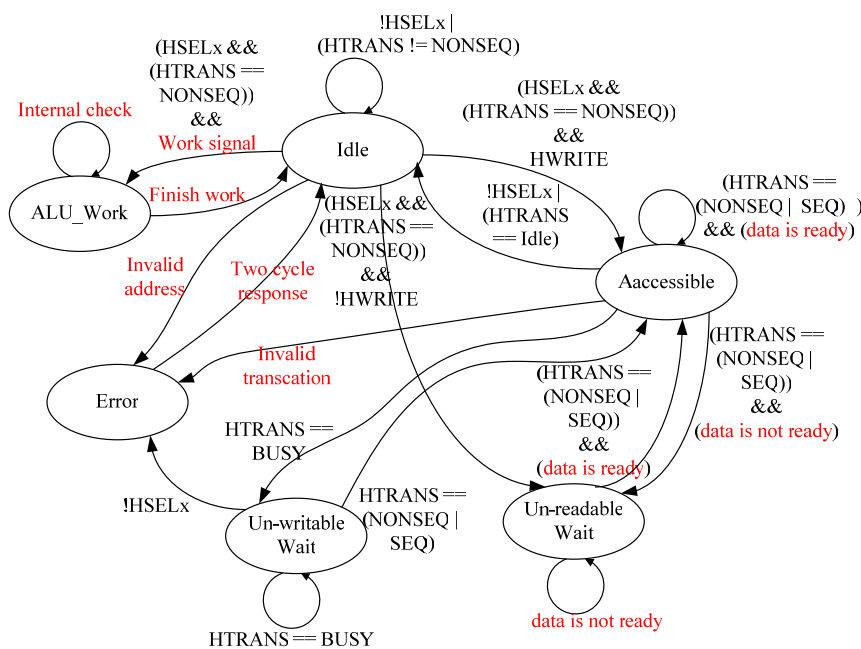


Figure 3.2.2: Finite State Machine

➤ Un-readable Wait

There are two possible paths that FSM will enter the Un-readable Wait state. One of the two paths is while the FSM is in the Idle state and the TRANS is NONSEQ and the HWRITE is low. It means the IP is being read. However, the first reading operation needs two cycles to prepare necessary data. Thus it must be in the Un-readable Wait state to wait the data. Until the data is ready, the FSM moves to Accessible state to carry out the following reading request. The other path is from Accessible state to Un-readable Wait state. The reason that the FSM must move to this state is the same as the first path, the necessary latencies.

➤ Un-writable Wait

There is only one reason that will enforce the FSM move to Un-writable Wait state. It is when the IP is being written data in burst mode of wrapping or incrementing way, but the TRANS is changed to BUSY. Pending the TRANS changes back to NONSEQ or SEQ, the FSM will return to Accessible state.

➤ Error

The FSM will go to the Error state while the IP is accessed in wrong ways. They are invalid address and invalid transaction. The invalid address is because the depth of

the data and instruction memory is limited. If the depth is over the real memory, it will be found by the FSM and go to the Error state. The invalid transaction is to avoiding the wrong combinations of HTRANS that violates the AHB protocol, although this might not be happened. The Error state will also occur if the IP is being accessed and is not granted unexpectedly, controlled by HSEL. This is designed to provide ability against accident. If the Error occurs, the Error state must obey the AHB protocol and thus have two cycles response, replying the AMBA BUS with proper HRESP and HREADY as defined in the specification.

➤ ALU_Work

The ALU_Work state manifests the IP is in the working service. The ALU cluster executes applications according to the instructions. There is only one path going to this state that is from Idle state to this one. The ALU cluster will be invoked by a strictly method that is to write the IP the end value of the program counter at predefined location. The details will be in Appendix B. In this state, if the wrapper is accessed whether it is a reading or writing operation, the FSM has the ability to reply the two cycles responses, RETRY, to the AMBA bus. At the same time, the ALU cluster keep working without being affected by the unexpected access until the end of the application. It will have internal check in the ALU cluster to judge if the execution is finished. Then, through signal, `alu_work`, the wrapper gets the status of ALU cluster.

There is one thing needed to be emphasized. Because the execution must go through many stages to finish one instruction, it will take more cycles to write back the executed results. The extra cycles are depending on the different functional units. For example, the ALU is a two stages pipeline functional unit so that it takes six cycles that is two plus four to finish its operation. The four cycles is necessary course for every operation, such as instruction decoding, data source selecting, and result writing. Thus, the four stages functional unit, MUL, will take eight cycles and the sixteen cycles' functional unit, DIV, will take twenty cycles to write back the result.

3.2.2 Modifications for Baseboard and Data Preparing

There are some modifications needed to program our wrapper into the baseboard. One of them is the interface of the wrapper to the outer bus because of the mismatch of the communication bus between the wrapper and baseboard. It can be found in the document of the baseboard that the data ports through the AHB bridges are shared by

writing and reading data signals. Thus, data signals of the wrapper to outer bus must be replaced with single tri-state bus, as shown in Figure 3.2.3. When the IP is accessed as writing transaction, it stands for input ports. As for reading state, the share bus then becomes output ports. It is needless of any revision for other ports because the baseboard is originally designed to be able compatible with AMBA system.

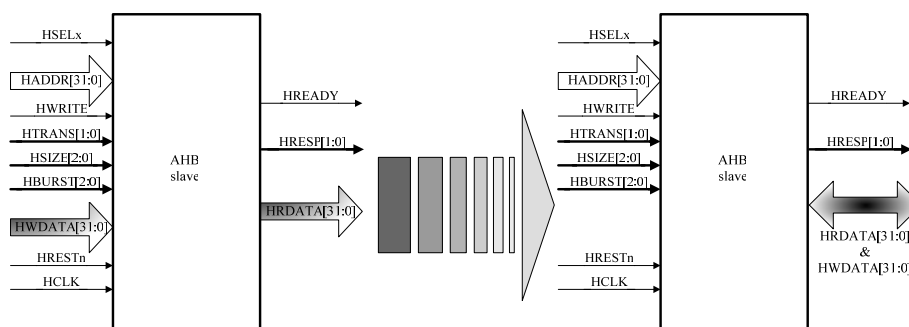


Figure 3.2.3: Ports modification

Because programming ALU cluster into the baseboard is hardly to verify the performance due to huge system, the baseboard doesn't contain enough memory, and the functionality of the ALU cluster is verified through silicon, it is not necessary to program total IP into the baseboard. Thus, we take off the ALU cluster, instruction memory, and data memory and replace them as register files, as shown in Figure 3.2.4. Notice that only the wrapper and register files are prepared to be programmed into the baseboard. The register files are used to store the accessed results and return them back by the wrapper with the on-chip bus. The connections to the register files have taken the place of the ones to original ALU cluster. As to the signal `alu_work`, it is tied to high that represents the ALU cluster is unused. Thus, this modification can simplify the effort of protocol verification by only access the register files through the wrapper.

There are also many data needed to be programmed into the baseboard with our wrapper together. They are Arbiter, Decoder, and Multiplexer. These components are used to select control the AMBA bus. All of them are ready and can be got from ARM's website. Thus, we put all these components including our wrapper and register files to software of Xilinx ISE to do board level synthesis, placement, and routing [23]. After the software is finished, the programming bit file is then generated. With the bit file, we therefore can use the ARM Multi-ICE to program it into the baseboard [24].

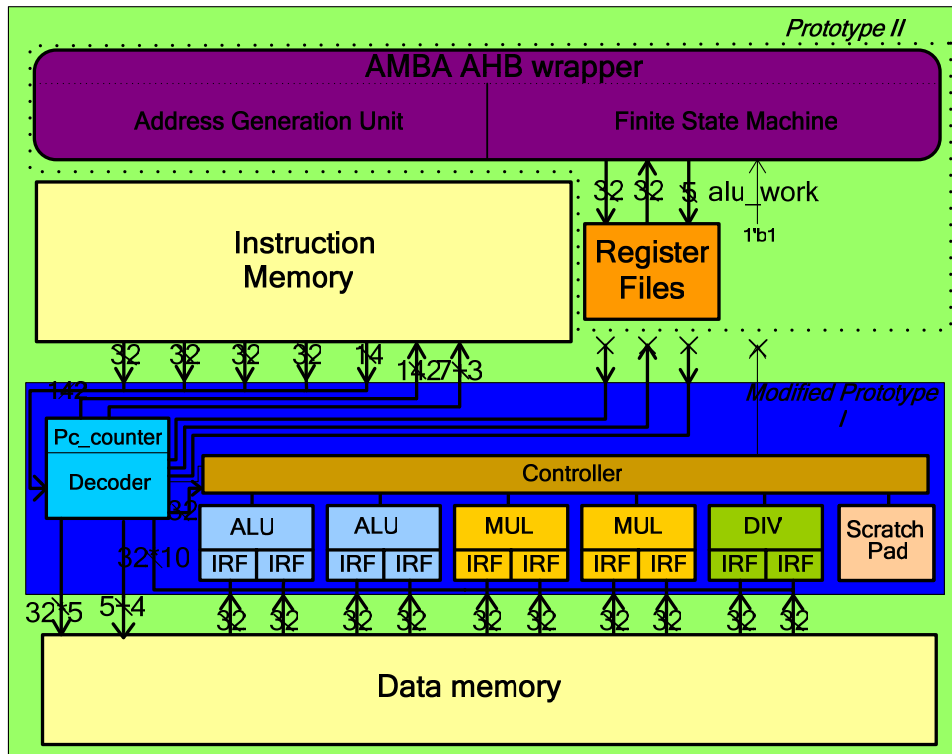


Figure 3.2.4: Modifications with Added Register Files

3.2.3 Emulation Result

After successfully programming our wrapper into the baseboard, the software of ARM Developer Suite (ADS) is then used to proceed the emulation [25]. By the ADS, the software codes are then assembled, compiled, and linked to be able to control the communications between ARM core and our wrapper. The software codes are demonstrated in Figure 3.2.5 and Figure 3.2.6. Figure 3.2.5 is the simple C code that has 32 entries of integer array that store 32 constants. These constants are then written to the specific address, ACT_ADD, which is the address of our wrapper and then read them out. The address is defined in Figure 3.2.6. WRAPPER_BASE is the base address of the wrapper and WORD_OFFSET is the offset to execute word access. Then, the read and written constants are compared to ensure that correctness of the transaction between ARM core and wrapper. The results are shown in Figure 3.2.7. It lists all the written and read results and comparison.


After the successful emulation, we therefore can make sure our wrapper is verified through the baseboard and is able to be integrated with other existing IP. Besides, the flow of emulation is set up from FPGA synthesis to ADS emulation. This flow is reusable to further ARM series development in the future.

```

int read_data[32]={0};
volatile int *io;
volatile int data[32] = {0x12345678,0x23456789,0x34567890,0x45678901,5,6,7,8,
volatile int WRAPPER_BASE_ADDR = WRAPPER_BASE;
volatile int ADDR_OFFSET = WORD_OFFSET;
volatile int ACT_ADD = WRAPPER_BASE_ADDR;
printf("start to write words\n\n");
for (i=0;i<32;i++)
{
    io = (int *)ACT_ADD;
    *io = data[i];
    ACT_ADD += ADDR_OFFSET;
    printf("write the %3dth data = %8x  ",i,data[i]);
    if (i%4 == 3)
        printf("\n"); }
ACT_ADD = WRAPPER_BASE_ADDR;
printf("\n\nstart to read words \n");
for (i=0;i<32;i++)
{
    io = (int *)ACT_ADD;
    read_data[i] = *io;
    ACT_ADD += ADDR_OFFSET;
    printf(" read the %3dth data = %8x  ",i,read_data[i]);
    if (i%4 == 3)
        printf("\n");}
printf("\n\nstart to compare read and write results\n\n");
for (i=0;i<32;i++)
{
    if (read_data[i] != data[i])
    {
        printf("Emulation Fail!!");
        return 0;}}
printf("Emulation Successful!!");

```

Figure 3.2.5: Software codes for ADS



```

// Versatile/PB926EJ-S baseboard registers
#define VPB_HDR_BASE          0x1000000
#define VPB_HDR_ID            ((volatile unsigned int *) (VPB_HDR_BASE + 0x000))
#define VPB_HDR_SW            ((volatile unsigned int *) (VPB_HDR_BASE + 0x004))
#define VPB_HDR_LED           ((volatile unsigned int *) (VPB_HDR_BASE + 0x008))
#define VPB_HDR_OSC0          ((volatile unsigned int *) (VPB_HDR_BASE + 0x00C))
#define VPB_HDR_OSC1          ((volatile unsigned int *) (VPB_HDR_BASE + 0x010))
#define VPB_HDR_LOCK          ((volatile unsigned int *) (VPB_HDR_BASE + 0x020))
#define VPB_HDR_OSCRESET0     ((volatile unsigned int *) (VPB_HDR_BASE + 0x08C))
#define VPB_HDR_OSRESET1     ((volatile unsigned int *) (VPB_HDR_BASE + 0x090))

// Versatile/CT header system registers
#define CT_BASE                0x1040000
#define CT_ID                  ((volatile unsigned int *) (CT_BASE + 0x000))
#define CT_PROC                ((volatile unsigned int *) (CT_BASE + 0x004))
#define CT_OSC                 ((volatile unsigned int *) (CT_BASE + 0x008))
#define CT_STAT                ((volatile unsigned int *) (CT_BASE + 0x010))
#define CT_LOCK                ((volatile unsigned int *) (CT_BASE + 0x014))
#define CT_VOLTAGE0            ((volatile unsigned int *) (CT_BASE + 0x080))
#define CT_VOLTAGE1            ((volatile unsigned int *) (CT_BASE + 0x084))
#define CT_VOLTAGE2            ((volatile unsigned int *) (CT_BASE + 0x088))
#define CT_VOLTAGE3            ((volatile unsigned int *) (CT_BASE + 0x08C))
#define CT_VOLTAGE4            ((volatile unsigned int *) (CT_BASE + 0x0A0))
#define CT_VOLTAGE5            ((volatile unsigned int *) (CT_BASE + 0x0A4))
#define CT_VOLTAGE6            ((volatile unsigned int *) (CT_BASE + 0x0A8))
#define CT_VOLTAGE7            ((volatile unsigned int *) (CT_BASE + 0x0AC))
#define CT_PLD                 ((volatile unsigned int *) (CT_BASE + 0x094))

#define WRAPPER_BASE          0xC200200
#define WORD_OFFSET           0x00000004

```

Figure 3.2.6: Address Definition

```

ARM926EJ_S_0 - Console
start to write words
write the 0th data = 12345678 write the 1th data = 23456789 write the 2th data = 34567890 write the 3th data = 45678901
write the 4th data = 5 write the 5th data = 6 write the 6th data = 7 write the 7th data = 8
write the 8th data = 9 write the 9th data = a write the 10th data = b write the 11th data = c
write the 12th data = d write the 13th data = e write the 14th data = f write the 15th data = 10
write the 16th data = 11 write the 17th data = 12 write the 18th data = 13 write the 19th data = 14
write the 20th data = 15 write the 21th data = 16 write the 22th data = 17 write the 23th data = 18
write the 24th data = 19 write the 25th data = 1a write the 26th data = 1b write the 27th data = 1c
write the 28th data = 1d write the 29th data = 1e write the 30th data = 1f write the 31th data = 20

start to read words
read the 0th data = 12345678 read the 1th data = 23456789 read the 2th data = 34567890 read the 3th data = 45678901
read the 4th data = 5 read the 5th data = 6 read the 6th data = 7 read the 7th data = 8
read the 8th data = 9 read the 9th data = a read the 10th data = b read the 11th data = c
read the 12th data = d read the 13th data = e read the 14th data = f read the 15th data = 10
read the 16th data = 11 read the 17th data = 12 read the 18th data = 13 read the 19th data = 14
read the 20th data = 15 read the 21th data = 16 read the 22th data = 17 read the 23th data = 18
read the 24th data = 19 read the 25th data = 1a read the 26th data = 1b read the 27th data = 1c
read the 28th data = 1d read the 29th data = 1e read the 30th data = 1f read the 31th data = 20

start to compare read and write results
Emulation Successful!!

```

```

System Output Monitor
RDI Log Debug Log
Log file:
Program terminated normally.

```

Figure 3.2.7: Emulation Results

3.3 An ALU Cluster Intellectual Property

In this section, we combine the improved ALU cluster and designed wrapper together. The improvement of the ALU cluster is for control and internal storages. As for the operational units, they are the same with old ones. It will be showed in the first part in this section. The second part is about the functional simulation. The testbench is 16-tap FIR program that is one common media application. Then, the last part is the summary of the improvements.

3.3.1 Architecture of an ALU Cluster Intellectual Property

The architecture of IP consists of the wrapper, ALU cluster, instruction memory and data memory, as shown in Figure 3.3.1. The wrapper is introduced in previous section. The decoder in the ALU cluster is the main improvement. One of the improvements is to improving the ability of reading source and writing destination. It makes all memory, including data and instruction memory, expose to the AMBA bus. They can be accessed directly from AMBA bus. Besides, it betters the performance in shortening access cycles. The reading takes two cycle's latencies in bursting reading. After the latencies, the reading data comes out every cycle. As for the original one, it must have four cycles to access one burst read.

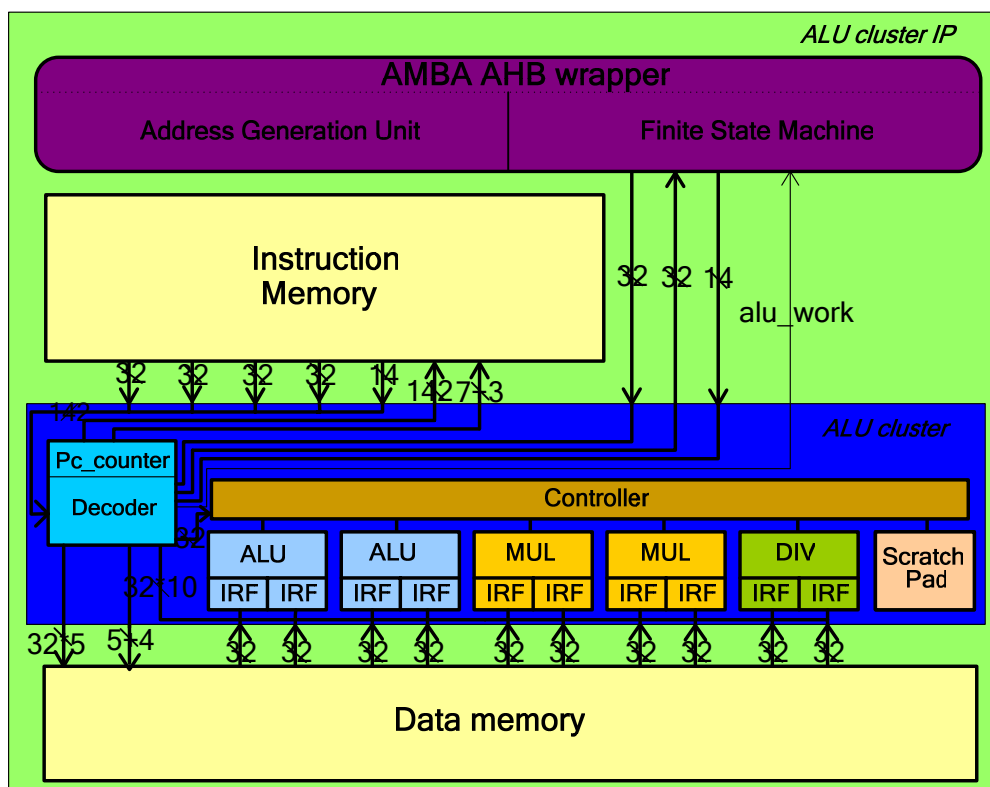


Figure 3.3.1: Architecture of an ALU Cluster IP

In order to let the ALU cluster is able to execute while AMBA bus is granted by other master, the ALU cluster needs a module that feeds instruction memory the address automatically. The new component, Pc_counter, is added to handle this job. It will increase the program counter by one every clock cycle. The decoder will have the end value of Pc_counter and compare it every cycle to check if the ALU cluster finishes the job. If the job is done, it activates the alu_work signal to let wrapper know the situation. If the alu_work is inactive, the IP can not be accessed simply and will return RETRY response back to AMBA bus. There is one special input combination can erase end value of Pc_counter in the decoder and thus coerce the IP to stop execution. This coercive mechanism is designed for avoiding possibility of deadlock occurring.

3.3.2 Functional Verification

The development of the IP including finite state machine in the wrapper and total architecture is shown as above. In this part, the simulation of the testbench of 16-tap FIR filter and its results will be demonstrated to verify the executing ability of the IP for media applications.

3.3.2.1 Testbench : 16-tap FIR filter System

The IP is simulated with the 16-tap FIR as benchmark. The FIR filter system is chosen as the testbench for the functional verification since it is suitable for one dimensional architecture, needs repeat and high percentage of addition and multiplication, and applies for wide DSP applications, such as matched filtering, pulse shaping, equalization, etc. A brief review of FIR filter system is illustrated in the following. The equation 3.1 is a description for FIR filter system. The M represents the length of FIR filter, b_k are the coefficients of the it, $x[n-k]$ is the data sample at time instance $n-k$, and $y[n]$ response the output to the instance time n .

$$y[n] = \sum_{k=0}^{M-1} b_k * x[n-k] \quad (3.1)$$

As shown in the left Figure 3., they are the coefficients of b_k and it is a 16-tap Kaiser window FIR bandpass filter. The right figure is the input function, which is exponential equation with ten sampling points. The FIR filter is simulated in advance by the Mathworks Matlab to get the correct results, as shown in Figure 3.3.2. The usage of the Matlab is to obtain the correct results with the input function is fed into the FIR filter. After comparing the results between the results from the Matlab and the IP, we thus can sure that the functionality is correct with the equality after the comparison.

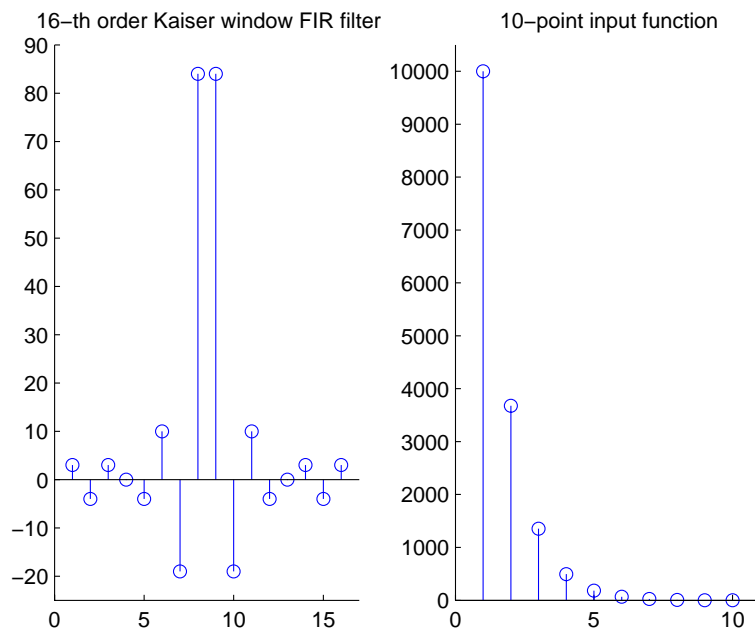


Figure 3.3.2: Coefficients of the FIR Filter System and Its Input Function

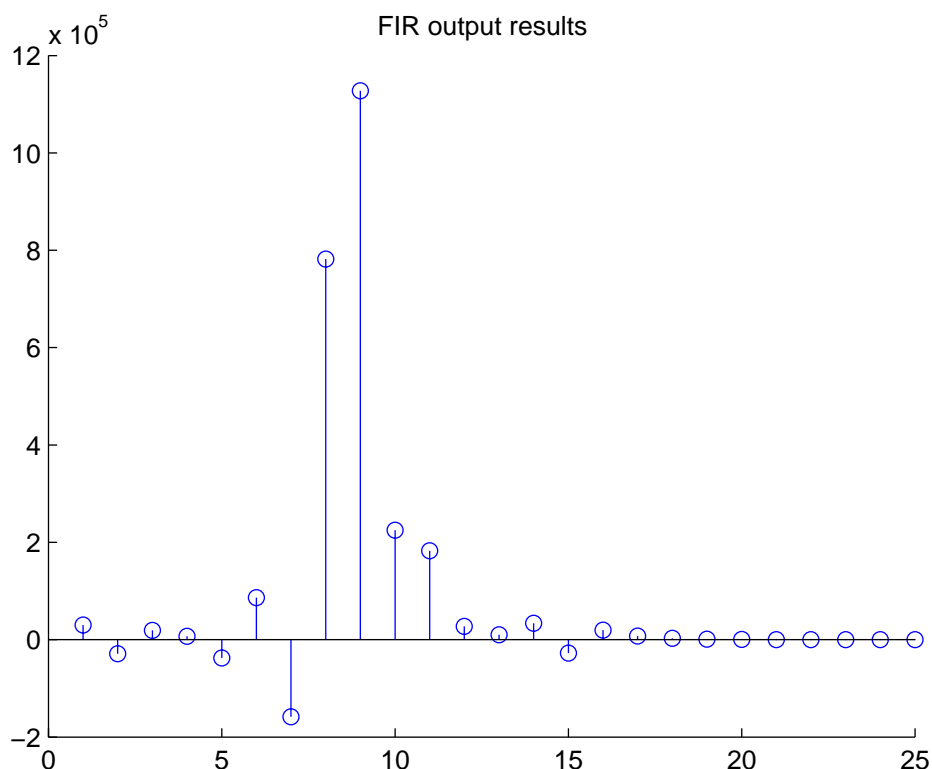


Figure 3.3.3: Expected Output Results of the FIR Filter System

3.3.2.2 Simulation Results

The simulation environment is based on UMC 0.18 libraries and is simulated after logic synthesizing. The clock cycle is set up with 6.5 ns. In other words it is simulated in 153.87 MHz. The simulation has four steps, as shown in Figure 3.3.4. It is the full view of the simulation, including all steps. The first one, as depicted in the red rectangular, is to write necessary coefficients into IP. It is because media applications usually have lots of tables and reusable coefficients throughout the total executing journey. Writing the common used coefficients helps to accelerate the execution by abating memory reference. Total numbers of coefficients written in the first step are fifty-two. They will be fed in to the following operations.

The second step is to configure IP, as depicted in the pink rectangular. The instructions are written into the instruction memory through AMBA bus. The needed configuring time depends on the different applications. As introducing in the previous chapter, the instruction format is a 142 bits VLIW instruction set. However, the input bandwidth from AMBA bus to the wrapper is only 32 bits. One instruction thus needs five times to be configured. The additional 18 bits, 160 bits for five clock cycles

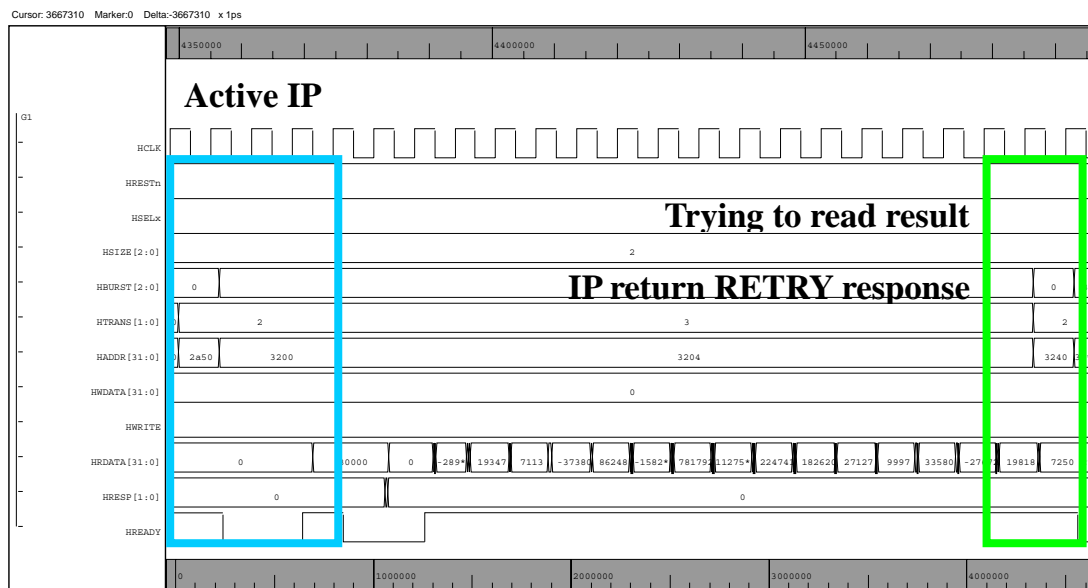


Figure 3.3.5: Detailed Waveform in Execution Stage A

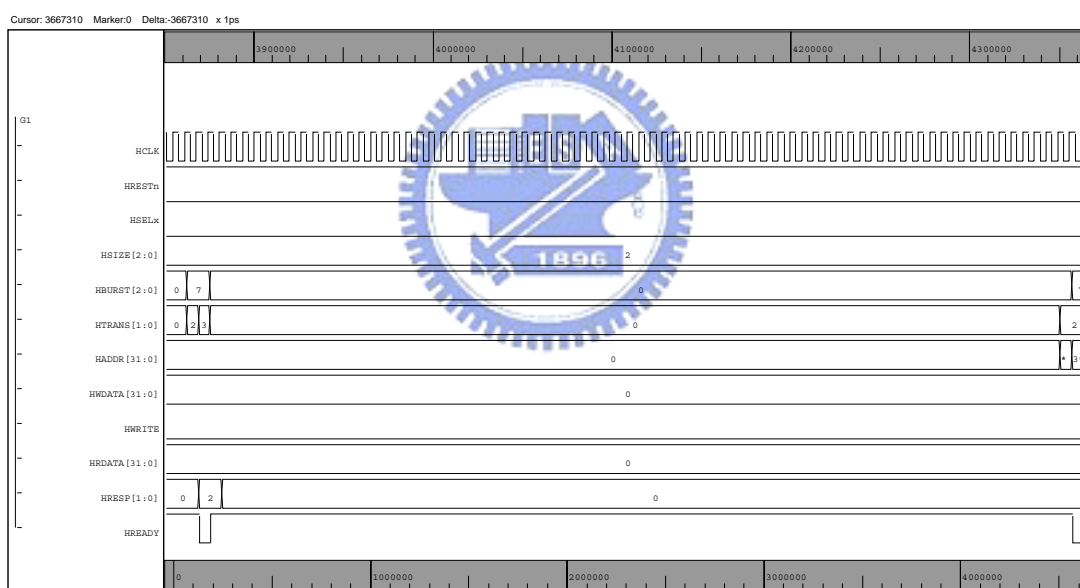


Figure 3.3.6: Detailed Waveform in Execution Stage B

The fourth stage is the reading stage, as depicted in the deep green rectangular in Figure 3.3.4. In this stage, the IP will read out the results to the AMBA bus. Figure 3.3.7 and Figure 3.3.8 are a clearly chart showing the reading procedure.

The yellow rectangular in the Figure 3.3.7 reveals the first data is read out. However, it can not be read out directly. The reading procedure must wait until the data is read out from data memory. Thus, it will have two cycles' latency. The HREADY therefore must be low at these two cycles to response the AMBA bus. Until

the third reading cycle, the data is read out successful and the HREADY changes to high. The green rectangular in Figure 3.3.8 is also a reading procedure. However, it is a 16-beats burst mode incremental reading. The first two cycles are the necessary latencies the same as above. Then the sixteenth data are then read out in order and the HREADY responses the correct reading with logic high. Figure 3.3.7 is the waveform behind Figure 3.3.8.

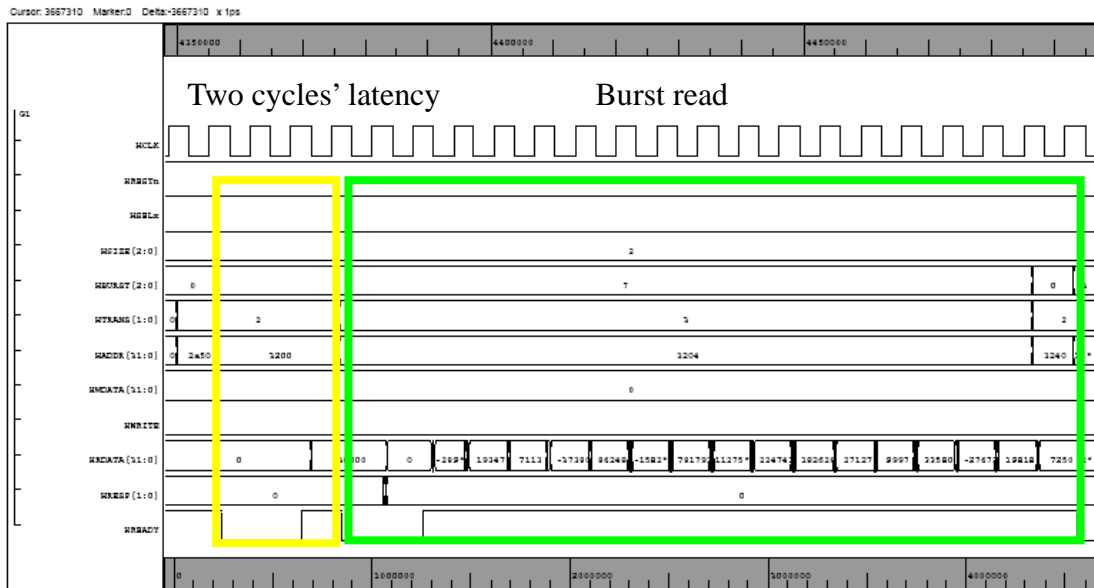


Figure 3.3.7: Detailed Waveform in Reading Stage A

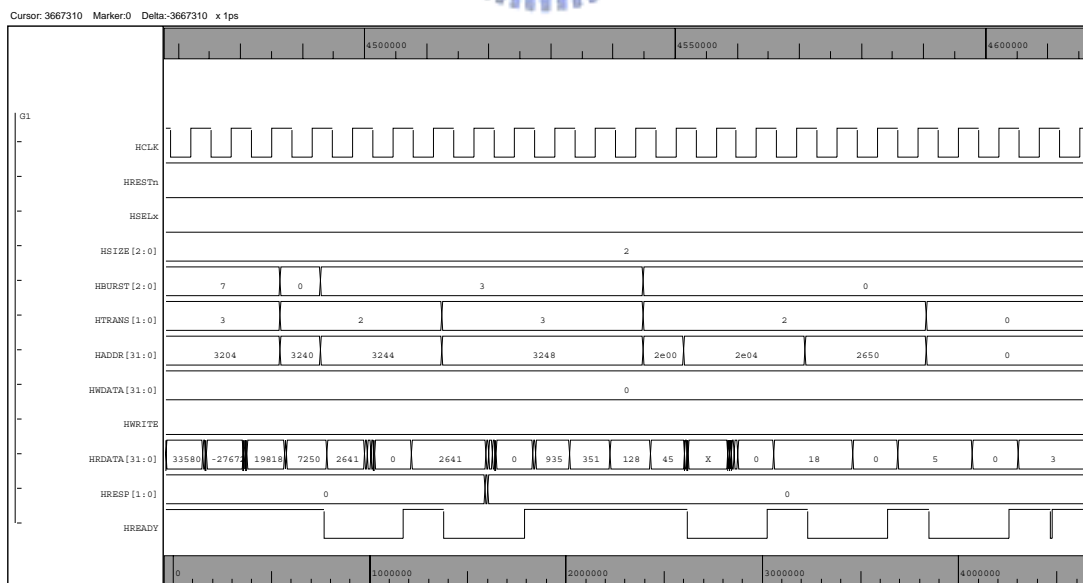


Figure 3.3.8: Detailed Waveform in Reading Stage B

The correct results from the IP must be gone along with the HREADY is high and HRESP is OKAY as specifying by protocol of AMBA bus. If either HREADY is low or HRESP is not OKAY, the simulation results are not what we wanted. The simulation results are then compared with the ones from Matlab, as presented in last section. We now can make sure that the functionality of IP is correct.

3.3.3 Improvements from ALU Cluster to ALU Cluster IP

The implementation from ALU cluster to ALU cluster IP not only adds the interface, AHB slave wrapper, but also has some intentional parts for improving previous one. The main modifications are listed in the table 3.3. It includes the reading ability of memory, synthesizing register, readable memory, and type of instruction fetch.

Table 3.3: Modifications from ALU cluster to ALU cluster IP

	ALU cluster IP	ALU cluster
Reading ability	One cycle to read one data (two cycles' latencies)	Four cycles to read one data
Register	Only positive edge trigger	Mix positive and negative edge trigger
Readable memory	All data and instruction memory	Only odd bank of data memory
Type of instruction fetch	Program counter	Must assign instruction address to be fetched

These four main modifications can be indicated in the table. The first one is the reading ability. The ALU cluster takes four cycles to read one data even if it is in burst mode to read data. This will cause the reading time is very long. Our IP amends this situation by shortening the necessary cycle for reading by only two cycle latencies. Moreover, if the reading is in burst mode, it can continuously read out the data after the first two cycle latencies. The reading of burst modes with both wrapping and incrementing styles works with this reading ability.

The second modification for the IP is the internal sequential element, register. The data and instruction memory in ALU cluster is negative edge trigger so that some registers in it also are negative edge trigger ones. This mixed type usage of register will cause some propagation only take less than half clock cycles to be used and the critical path thus arises on the propagation path. This degeneration of total performance is not on the execution unit is an unfortunately thing that have too many waste timing loss due to the half cycle usage. Besides, because the clock's double edges are all needed to trigger by different registers, it will make the clock tree hardly to establish in the auto placement and route (APR) stage. Keeping the duty cycle takes also more efforts for APR tools. Owing to so many induced problems, we replace the memory and its related negative edge trigger registers to positive edge trigger ones to solve these problems. The related performance improving will be shown in the latter section.

The third modification arises from the testing. In the ALU cluster, not all the memory can be read out. Only odd bank of the data memory is visible. The even bank memory and the instruction memory are invisible. They all have ability to be written data but lack of aptitude to be read out. This is very inconvenient for testing issue. If the data is willing to be read out to find out the possible defects, the visibility for the memory is very preciousness. Although it will have more wires and multiplexers to the requirement that is more compulsory gate counts, we still amend the ALU cluster the reading ability.

The last modification is the type of the instruction fetch. The type of instruction fetch in original design of the ALU cluster must pass instruction address to activate the procedure of fetching instruction. It is flexible to choose the wanted instruction by feeding its address. The stall, jump and other branch instructions can be easily substituted with this way. However, it is not a comfortable way compared to the method that is able to fetch instruction automatically. It will add another responsibility to tell the instruction memory its necessary address. It means that it must have a mechanism to keep sight of the fetching procedure. It is obviously not matched to our intended design, which checks automatically by itself.

Therefore, an incremental adder, Pc_counter, which is introduced in the previous section, is added. It will automatically pass address to instruction memory to get the instructions. The advantage of using the Pc_counter to generate the necessary address is that the other components on the AMBA bus can spare the attention when the IP is executing the media applications. It thus increases the bus utility.

3.3.4 Extension of ALU cluster IP at magnetic RAM (MRAM)

This part will address that the IP is extended by replacing the data memory of the IP with a new type of memory with magnetism. Most of the other blocks are unchanged. Only a few blocks are slightly modified to be adequate for MRAM. The first sub part is the overview of MRAM. The second one is about the necessary modifications for the connecting interface between IP and MRAM. Then, there is one another load_store unit that takes the responsibility of communication. Finally, the results of implementation are listed in the final subsection.

3.3.4.1 Overview of MRAM

General purpose memory stands for the data accessing, such as SRAM and DRAM. They have a characteristic of high speed accessing. However, once the power is off, the stored data will also be cleared. It thus can not store data in long time. Batter-SRAM is provided to support problems of data retention but it costs with large power consumption and area overhead. The non-volatile memory therefore is invited to overcome the hardness like EEPROM, Flash memory, and etc. Nevertheless, for the non-volatile memory the accessing control is much more complex and has a limit of number to re-read and re-write.

MRAM is then innovated with a kind of non-volatile memory. It is not like conventional non-volatile memory by extra processing with gate of transistor. It uses magnetism to represent the logic state. There are several advantages of MRAM; process compatible, large number times of data accessing, and non-volatile long life time. Because MRAM is fabricated by the upper metal layer, it thus is compatible to CMOS technology and will have low extra area overhead. The times of data accessing, over 10^{15} , is much greater than conventional non-volatile memory. This is an attractive feature for Consumer Electronic Product. It thus can be reused nearly forever. Therefore, MRAM has a large potential to be another trend of memory.

3.3.4.2 Modifications for MRAM

Because the interface of MRAM and SRAM is different and the supporting bandwidth of MRAM is smaller than SRAM, it must be adjusted a lot to be able to connect MRAM with IP. We thus add an extra unit, Load_store unit (LSU), to account for this issue, as shown in Figure 3.3.9. Therefore, the decoder must have ability for the LSU unit. We increase conventional 142 bits to 143 bits for the instruction. If the

143rd bit is not set, the whole IP acts the same as before to execute the applications. As the 143rd bit is set, the LSU operate to access data between IRF and MRAM depending on the instruction. The data bandwidth between IRF and MRAM is restricted by MRAM. We thus also modify the bandwidth of IRF to support byte access. The byte access will not get any trouble to the AHB wrapper. The wrapper is originally designed for byte, half word, and word access in little endian manner.

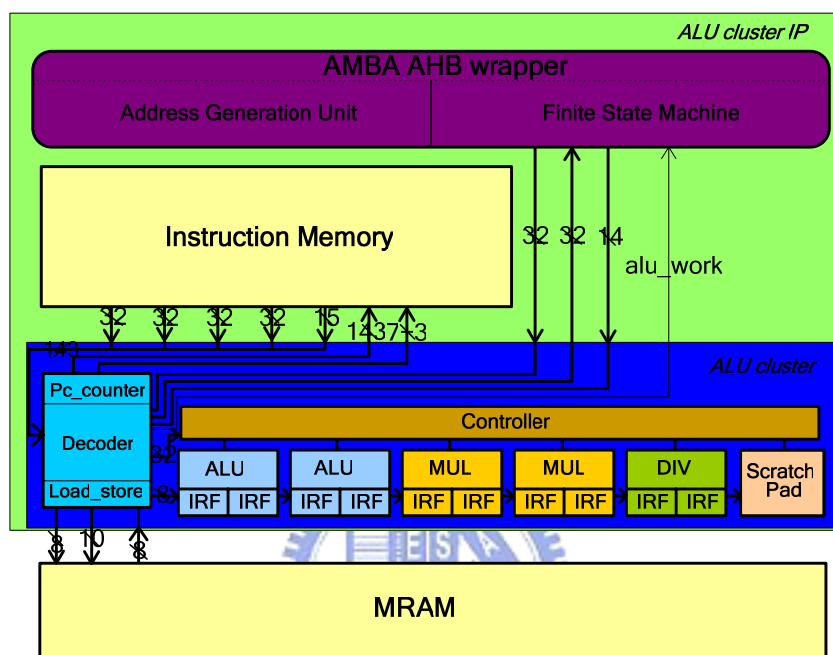


Figure 3.3.9: Modifications for MRAM with Added Load_store Unit

3.3.4.3 Implementation Results

The summary of implementation results are listed in table 3.4. It is through 0.18 μm process of TSMC and cell based design kit of Artisan. The operating frequency of post-layout simulation is 105 MHz. The chip size is about $3.15 \times 3.15 \text{mm}^2$, and its core size is $2.31 \times 2.30 \text{mm}^2$. The gate count of the core is 260910. It must be noted that the data memory in the IP is replaced with MRAM such that there is no area cost for data memory. The instruction memory is the same of the IP with only one difference. It is the adding bit for load_store unit. Therefore, we need eighteen 8×128 single port static RAM (SRAM) which are generated by memory compiler with Artisan library. The power dissipation is 403.36 mW for the total chip while is simulated by 0.9 net toggle probability. The pure core size without instruction memory is about $1.43 \times 1.43 \text{mm}^2$. Its pure gate is 203893.67 and the power dissipation is down to 273.6mW. The physical layout of this co-project is shown in Figure 3.3.10 and its floorplan is in Figure 3.3.11.

Table 3.4: Summary of Implementation Results

Process	TSMC 0.18 um
Library	Artisan SAGE-x Standard Cell Library
Post-layout Clock Rate	105 MHz (9.5ns)
Chip Size	3.15x3.15 mm ²
Core Size (without memory)	2.31x2.30 mm ² (1.43x1.43 mm ²)
Gate Count (without memory)	260910 (203893.67)
Power Dissipation (without memory)	403.36 mw (273.6 mW)
On-chip memory	18x128x8single port SRAM
Pad	Input: 34 pins Output: 25 pins Inout: 32pins Power: 40 pins

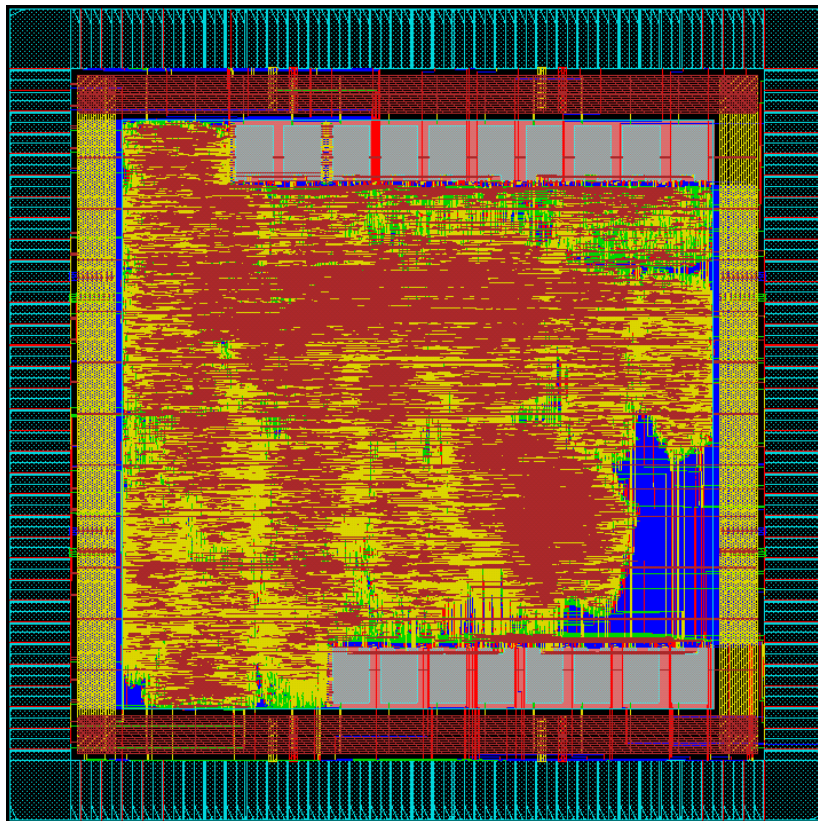


Figure 3.3.10: Layout of an ALU Cluster IP Extended at MRAM

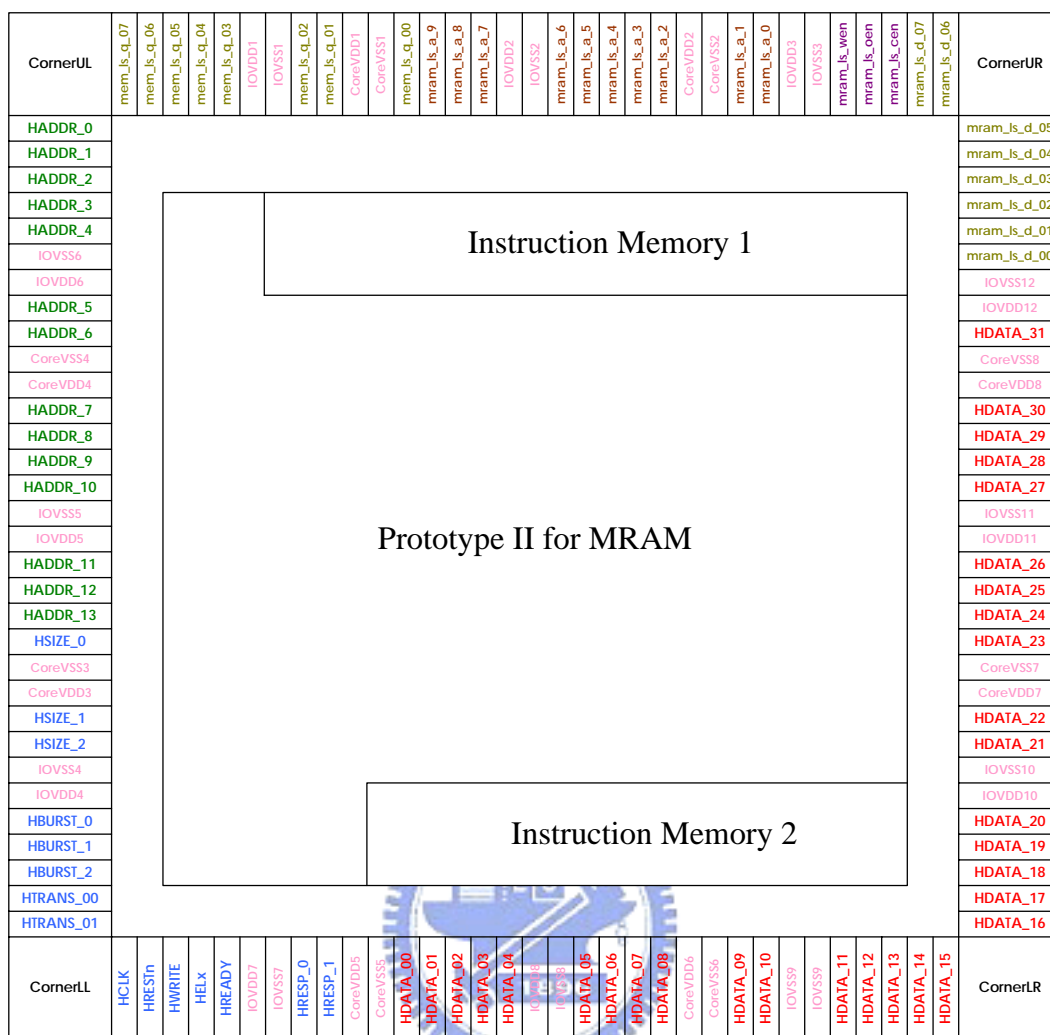


Figure 3.3.11: Floorplan of an ALU Cluster IP Extended at MRAM

In this section, a synthesizable ALU cluster IP is finished. It is designed from previous ALU cluster with added AHB slave wrapper. Thus, a hardware accelerator for media applications is complete. Owing to it is a soft IP, it is portable to different process. The extension of MRAM thus is an example that the process is changed from UMC to TSMC. The results of implementation for different process will be in next section.

3.4 Implementation Comparisons

There are four implemented results about the related work in different process, as listed in table 3.5. The first one is the ALU cluster that is designed by senior in our lab and has real silicon back. The second implementation is the synthesizable ALU cluster IP that is my proposed design. These two are implemented with the same

process, UMC 0.18 um. The third and forth implementation is the ALU cluster IP that is extended at MRAM. They are implemented in process TSMC 0.18 um and 0.15 um. Therefore, we take a look at first two results. Although one of them is after auto placement and route (APR) and the other one is after synthesis, the comparisons are not very fair. It still shows some useful data: the gate count is reduced about 25% and the clock cycle is reduced from 10ns to 6.5ns. Even though these two data will be worst after APR, it presents some advancements. Through the first and third results which are both implemented after APR, we can see that the gate count is about 20% better than older one. This is a great reduced for area. As for the clock cycle, it is nearly equal to the older one. This results from the usage of inout PAD causing extra time to select to type of that PAD.

Table 3.5: Table of Implemented Comparisons

	ALU cluster (after APR)	ALU cluster IP (after synthesis)	ALU cluster IP (after APR) (MRAM)	ALU cluster IP (after synthesis) (MRAM)
Process	UMC 0.18 um	UMC 0.18 um	TSMC 0.18 um	TSMC 0.15 um
Simulation frequency	100MHz (1/10ns)	153.87MHz (1/6.5ns)	105.26 MHz (1/9.5ns)	166.67MHz (1/6ns)
Gate Count (include instruction memory)	255669 (295365)	192767 (249784)	203894 (260910)	267473 (include instruction memory)
Gate Count (wrapper)	-	1699 (0.88%)	1708 (0.84%)	1202(0.45%)
Pad	Input: 47 pins Output: 32 pins Power: 48 pins	-	Input: 34 pins Output: 25 pins Inout: 32pins Power: 40 pins	-

The forth data shows that when the process go down, the performance will be better. Its huge gate cout is because we lack of memory compiler at this process. Thus, instruction memory is changed from SRAM to registers. This increases the gate cout mostly. As for the gate count of the wrapper, it is obviously that the area overhead for the wrapper is slightly. It contains less than 1% area cost for each wrapper in three cases.

3.5 Summary

The proposed design is finished as a synthesizable IP that is portable to different CMOS process as showing in three implementations that the process is changed from UMC to TSMC. The verification for our IP is separated to two parts; hardware emulation and chip testing. Hardware emulation is used to ensure that designed wrapper is realistic to implement into hardware and is compatible to AMBA system. The processing core, ALU cluster, is verified by chip testing. The memory is tested to guarantee that the process of UMC 0.18 um is stable to further development. The operational units are also tested to promise our ALU cluster is workable to process media applications. Then, our proposed ALU cluster IP is simulated to demonstrate that it has the same processing ability with the older ALU cluster and betters its reading ability. Besides, it is extended to replace data memory with MRAM as storage elements. Therefore, our IP has a non-volatile data memory for media applications.



CHAPTER 4

Conclusion and Future Work

This thesis proposes a reconfigurable architecture for multimedia applications. That is an ALU cluster IP as a reconfigurable hardware accelerator in platform-based design. It is proven by simulating with a testbench, 16 tap FIR. Therefore, it can take the responsibility to accelerate the performance for media applications.

The proposed ALU cluster IP has been disassembled to two parts, designed ALU cluster and wrapper, for testing and verification. Designed ALU cluster is proved by silicon. It thus ensures us that the operational unit, ALU cluster, is workable to handle media ones. Designed wrapper is emulated in AMBA platform. Through this emulation, we not only verify the correctness of protocol but also set up the emulating flow that is reusable for further development.

The designed soft IP is fully synthesizable and compatible to different CMOS processes. According the synthesis scripts, it is effortless to change process from UMC to TSMC and from 0.18 um to 0.15 um.

However, there are several future works can be carry out to investigate this design. In hardware part, the wrapper is emulated through VPB926EJ-S. However, the processing core, ALU cluster, is not programmable into the FPGA board. It is pity to emulate total design. Thus, after our IP is taped out and has silicon back, it is able to setup the enviornment of it that is compatible to VPB926EJ-S and can be plugged in it as a daughter board. Therefore, our IP can truly stand for a hardware accelerator in VPB926EJ-S.

As for the benchmarks, because there is only one benchmark, 16 tap FIR, it is not enough to convince everyone that proposed IP is powerful for media applications.

Thus, it is necessary for our IP to be verified through other media related benchmarks. For this reason, a wisely compiler is needed to schedule the instructions for the VLIW instruction sets of the ALU cluster. This will be a great help to prepare needed media applications.



BIBLIOGRAPHY

- [1] W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, A. Das, "Stream Processors: Programmability with Efficiency," *ACM Queue*, pages 52-62, March 2004.
- [2] S. Rixner¹, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," *Proceeding of 31st Annual ACM/IEEE International Symposium on Microarchitecture*, page 3-13, November 1998.
- [3] L. Hennessy, A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann Publishers, 2003.
- [4] W. Wolf, *Modern VLSI Design: System-on-Chip Design*, Third Edition, Prentice Hall Modern Semiconductor Design Series, 2002.
- [5] J. Rabaey, A. Chandrakasan, B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, Second Edition, Prentice Hall Electronics and VLSI Series, 2003.
- [6] N. H. E. Weste, K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, Second Edition, Addison-Wesley VLSI Systems Series, 1993.
- [7] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca, "The Architecture of the DIVA Processing-In-Memory Chip," *Proceedings of the International Conference on Supercomputing*, pages 14-25, June 2002.
- [8] T.W. Lin, "An ALU cluster Design for Media Streaming processors Architecture," Master dissertation, University of Chiao Tung University, 2005.
- [9] T. W. Lin, M. C. Lee, F. J. Lin, H. Chiueh, "A Low Power ALU Cluster Design for Media Streaming Architecture," *to appear: IEEE 48th International Midwest Symposium on Circuits and Systems*, August 2005.

- [10] D.D. Tang, P.K. Wang, V. S. Speriosu, S. Le, R.E. Fontana, S. Rishton, "An IC process compatible Nonvolatile Magnetic RAM," *Electron Devices Meeting*, pages 997-1000, 1995.
- [11] J. Draper, J. Sondeen, S. Mediratta, I. Kim, "Implementation of a 32-bit RISC Processor for the Data-Intensive Architecture Processing-In-Memory Chip," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 163-172, July 2002.
- [12] B. Khailany, J. W. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, S. Rixner, "Imagine: Media Processing with Streams," *IEEE Micro*, pages 35-46, March-April 2001.
- [13] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, J. D. Owens, "Programmable Stream Processors," *IEEE Computer*, pages 54-62, August 2003.
- [14] A.V. Oppenheim, R. W. Schaffer, J. R. Buck, *Discrete-Time Signal Processing*, Second edition, Prentice Hall Signal Processing Series, 1999.
- [15] M. Keating, P. Bricaud, *Reuse Methodology Manual for System-on-Chip Designs*, Third Edition, Kluwer Academic Publishers, 2002.
- [16] R. Hofmann, B. Drerup, "Next Generation CoreConnect™ Processor Local Bus Architecture," *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 221- 225, Sept. 2002.
- [17] D. Wingard, A. Kurosawa, "Integration architecture for system-on-a-chip design," *Custom Integrated Circuits Conference, 1998., Proceedings of the IEEE 1998*, pages 85-88, May 1998.
- [18] D. Flynn, "AMBA: enabling reusable on-chip designs," *Micro, IEEE*, Volume: 17, pages 20-27, Jul/Aug 1997.
- [19] ARM Corp., "AMBA(TM) Specification (Rev 2.0)," http://www.arm.com/products/solutions/AMBA_Spec.html

- [20] ARM Corp., “RealView Platform Baseboard for ARM926EJ-S HBI-0117 User Guide,” http://www.arm.com/pdfs/DUI0224C_pb926ej-s_user_guide.pdf
- [21] ARM Corp., “Versatile/LT-XC2V4000+ Logic Tile User Guide, ” http://www.arm.com/pdfs/DUI0186B_lt_xc2v_userguide.pdf
- [22] <http://www.home.agilent.com/>
- [23] <http://www.xilinx.com/>
- [24] ARM Corp., “Multi-ICE Version 2.2 User Guide,” http://www.arm.com/pdfs/DUI0048F_MICE2_2.pdf
- [25] ARM Corp., “ARM Developer Suite Version 1.2 Developer Guide,” http://www.arm.com/pdfs/DUI0056D_ADS1_2_Dev.pdf



APPENDIX A

Assembly Codes for Chip Testing

Table A.1: The Operations Correspond to the ALU Unit

<i>Operation</i>	<i>Description</i>
ADD	Add
SUB	Subtract
ABS	Absolute value
AND	Bitwise AND
OR	Bitwise OR
XOR	Bitwise XOR
NOT	Bitwise invert
SLL	Logical shift left
SRL	Logical shift right
SRA	Arithmetic shift right
LT	Less-than
GT	Greater-than
EQ	Equal

Table A.2: The Operation Corresponds to the MUL Unit

<i>Operation</i>	<i>Description</i>
MUL	Multiply

Table A.3: The Operations Correspond to the DIV Unit

<i>Operation</i>	<i>Description</i>
DIV	Quotient
REM	Remainder
SQR	Square root

Instruction format of ALU cluster: S1_A1_S2_A2_DE_DA_OP

S1: source selection1

A1: address of selection1

S2: source selection2

A2: address of selection2

DE: destination selection

DA: address of destination

OP: operation code

DM, D1~D9 : data memory

RF, I1~I9 : local register file (RF is source; Ix is destination)

SP : scratch pad register file

A.1 Assembly Code of FIR

Table A.4: Assembly code for FIR

#	ALU_0	ALU_1	MUL_0	MUL_1	DIV_0
1			DM_16_DM_01_DM_20_MUL	DM_15_DM_01_I9_00_MUL	
2			DM_16_DM_02_I8_00_MUL	DM_14_DM_01_I7_00_MUL	
3			DM_15_DM_02_I6_00_MUL	DM_16_DM_03_I9_01_MUL	
4			DM_13_DM_01_I7_01_MUL	DM_14_DM_02_I6_01_MUL	
5			DM_15_DM_03_I7_02_MUL	DM_16_DM_04_I6_02_MUL	
6			DM_12_DM_01_I7_03_MUL	DM_13_DM_02_I6_03_MUL	
7			DM_14_DM_03_I7_04_MUL	DM_15_DM_04_I6_04_MUL	
8					
9			DM_16_DM_05_I8_01_MUL	DM_11_DM_01_I7_05_MUL	
10	RF_00_RF_00_DM_00_ADD		DM_12_DM_02_I6_05_MUL	DM_13_DM_03_I7_06_MUL	
11		RF_00_RF_00_SP_00_ADD	DM_14_DM_04_I6_06_MUL	DM_15_DM_05_I7_07_MUL	
12		RF_01_RF_01_I9_02_ADD	DM_16_DM_06_I6_07_MUL	DM_10_DM_01_I7_08_MUL	
13		RF_02_RF_02_I8_02_ADD	DM_11_DM_02_I6_08_MUL	DM_12_DM_03_I7_09_MUL	
14		RF_03_RF_03_I9_03_ADD	DM_13_DM_04_I6_09_MUL	DM_14_DM_05_I7_10_MUL	
15		RF_04_RF_04_I8_00_ADD	DM_15_DM_06_I6_10_MUL	DM_16_DM_07_I9_00_MUL	
16	RF_01_SP_00_DM_01_ADD		DM_09_DM_01_I7_00_MUL	DM_10_DM_02_I6_00_MUL	
17		RF_05_RF_05_I8_03_ADD	DM_11_DM_03_I7_01_MUL	DM_12_DM_04_I6_01_MUL	
18	RF_02_RF_02_DM_02_ADD	RF_06_RF_06_I9_04_ADD	DM_13_DM_05_I7_02_MUL	DM_14_DM_06_I6_02_MUL	
19	RF_03_RF_01_I9_05_ADD	RF_07_RF_07_I8_04_ADD	DM_15_DM_07_I7_03_MUL	DM_16_DM_08_I6_03_MUL	
20		RF_08_RF_08_I8_05_ADD	DM_08_DM_01_I7_04_MUL	DM_09_DM_02_I6_04_MUL	

Appendix A: Assembly Codes for Chip Testing

21		RF_09_RF_09_19_01_ADD	DM_10_DM_03_17_11_MUL	DM_11_DM_04_16_11_MUL	
22		RF_10_RF_10_18_06_ADD	DM_12_DM_05_17_05_MUL	DM_13_DM_06_16_05_MUL	
23	RF_04_RF_03_19_06_ADD	RF_00_RF_00_18_07_ADD	DM_14_DM_07_17_06_MUL	DM_15_DM_08_16_06_MUL	
24	RF_05_RF_00_DM_03_ADD	RF_01_RF_01_19_02_ADD	DM_16_DM_09_18_01_MUL	DM_07_DM_01_17_07_MUL	
25	RF_00_RF_05_19_03_ADD	RF_02_RF_02_18_02_ADD	DM_08_DM_02_16_07_MUL	DM_09_DM_03_17_08_MUL	
26		RF_03_RF_03_19_07_ADD	DM_10_DM_04_16_08_MUL	DM_11_DM_05_17_09_MUL	
27	RF_01_RF_06_18_08_ADD	RF_04_RF_04_19_08_ADD	DM_12_DM_06_16_09_MUL	DM_13_DM_07_17_10_MUL	
28	RF_06_RF_04_DM_04_ADD	RF_11_RF_11_18_03_ADD	DM_14_DM_08_16_10_MUL	DM_15_DM_09_17_00_MUL	
29	RF_02_RF_07_18_00_ADD	RF_05_RF_05_19_04_ADD	DM_16_DM_10_16_00_MUL	DM_06_DM_01_17_01_MUL	
30		RF_06_RF_06_18_05_ADD	DM_07_DM_02_16_01_MUL	DM_08_DM_03_17_02_MUL	
31	RF_07_RF_02_19_00_ADD		DM_09_DM_04_16_02_MUL	DM_10_DM_05_17_03_MUL	
32	RF_08_RF_01_19_01_ADD	RF_07_RF_07_18_06_ADD	DM_11_DM_06_16_03_MUL	DM_12_DM_07_17_04_MUL	
33	RF_03_RF_08_DM_05_ADD	RF_08_RF_08_19_05_ADD	DM_13_DM_08_16_04_MUL	DM_14_DM_09_17_11_MUL	
34	RF_04_RF_03_19_02_ADD	RF_09_RF_09_18_04_ADD	DM_15_DM_10_16_11_MUL	DM_05_DM_01_17_05_MUL	
35		RF_10_RF_10_19_06_ADD	DM_06_DM_02_16_05_MUL	DM_07_DM_03_17_06_MUL	
36	RF_00_RF_00_DM_06_ADD	RF_00_RF_00_18_02_ADD	DM_08_DM_04_16_06_MUL	DM_09_DM_05_17_12_MUL	
37	RF_01_RF_05_18_01_ADD	RF_01_RF_01_19_07_ADD	DM_10_DM_06_16_12_MUL	DM_11_DM_07_17_07_MUL	
38	RF_05_RF_06_19_03_ADD	RF_02_RF_02_18_07_ADD	DM_12_DM_08_16_07_MUL	DM_13_DM_09_17_08_MUL	
39		RF_03_RF_03_19_04_ADD	DM_14_DM_10_16_08_MUL	DM_04_DM_01_17_09_MUL	
40	RF_06_RF_04_19_08_ADD	RF_04_RF_04_18_03_ADD	DM_05_DM_02_16_09_MUL	DM_06_DM_03_17_10_MUL	
41		RF_11_RF_11_18_00_ADD	DM_07_DM_04_16_10_MUL	DM_08_DM_05_17_00_MUL	
42	RF_02_RF_01_DM_07_ADD	RF_05_RF_05_19_00_ADD	DM_09_DM_06_16_00_MUL	DM_10_DM_07_17_01_MUL	
43	RF_03_RF_02_SP_00_ADD	RF_06_RF_06_18_05_ADD	DM_11_DM_08_16_01_MUL	DM_12_DM_09_17_02_MUL	
44	RF_07_RF_07_19_01_ADD	RF_12_RF_12_18_06_ADD	DM_13_DM_10_16_02_MUL	DM_03_DM_01_17_03_MUL	
45	RF_04_RF_03_18_04_ADD	RF_07_RF_07_19_05_ADD	DM_04_DM_02_16_03_MUL	DM_05_DM_03_17_04_MUL	
46		RF_08_RF_08_19_06_ADD	DM_06_DM_04_16_04_MUL	DM_07_DM_05_17_11_MUL	
47		RF_09_RF_09_19_02_ADD	DM_08_DM_06_16_11_MUL	DM_09_DM_07_17_05_MUL	
48	RF_08_SP_00_DM_08_ADD	RF_10_RF_10_18_01_ADD	DM_10_DM_08_16_05_MUL	DM_11_DM_09_17_06_MUL	
49	RF_01_RF_00_19_03_ADD	RF_00_RF_00_18_02_ADD	DM_12_DM_10_16_06_MUL	DM_02_DM_01_17_12_MUL	
50	RF_00_RF_05_18_03_ADD	RF_01_RF_01_19_04_ADD	DM_03_DM_02_16_12_MUL	DM_04_DM_03_17_07_MUL	
51	RF_05_RF_06_18_07_ADD	RF_02_RF_02_19_07_ADD	DM_05_DM_04_16_07_MUL	DM_06_DM_05_17_08_MUL	
52		RF_03_RF_03_19_09_ADD	DM_07_DM_06_16_08_MUL	DM_08_DM_07_17_09_MUL	
53	RF_02_RF_01_19_08_ADD	RF_04_RF_04_18_08_ADD	DM_09_DM_08_16_09_MUL	DM_10_DM_09_17_10_MUL	
54	RF_03_RF_04_DM_09_ADD	RF_11_RF_11_19_01_ADD	DM_11_DM_10_16_10_MUL	DM_01_DM_01_17_00_MUL	
55	RF_06_RF_03_19_00_ADD	RF_05_RF_05_18_00_ADD	DM_02_DM_02_16_00_MUL	DM_03_DM_03_17_01_MUL	
56	RF_04_RF_02_18_05_ADD	RF_06_RF_06_19_05_ADD	DM_04_DM_04_16_01_MUL	DM_05_DM_05_17_02_MUL	
57		RF_12_RF_12_18_06_ADD	DM_06_DM_06_16_02_MUL	DM_07_DM_07_17_03_MUL	

Appendix A: Assembly Codes for Chip Testing

58	RF_09_RF_08_18_01_ADD	RF_07_RF_07_19_02_ADD	DM_08_DM_08_16_03_MUL	DM_09_DM_09_17_04_MUL	
59		RF_08_RF_08_18_04_ADD	DM_10_DM_10_16_04_MUL	DM_01_DM_02_17_11_MUL	
60	RF_00_RF_07_DM_10_ADD	RF_09_RF_09_19_03_ADD	DM_02_DM_03_16_11_MUL	DM_03_DM_04_17_05_MUL	
61	RF_07_RF_05_18_02_ADD	RF_10_RF_10_19_04_ADD	DM_04_DM_05_16_05_MUL	DM_05_DM_06_17_06_MUL	
62	RF_01_RF_00_18_03_ADD	RF_00_RF_00_19_06_ADD	DM_06_DM_07_16_06_MUL	DM_07_DM_08_17_12_MUL	
63	RF_05_RF_01_19_09_ADD	RF_01_RF_01_18_09_ADD	DM_08_DM_09_16_12_MUL	DM_09_DM_10_18_08_MUL	
64	RF_02_RF_06_18_10_ADD	RF_02_RF_02_19_10_ADD	DM_01_DM_03_17_07_MUL	DM_02_DM_04_16_07_MUL	
65	RF_03_RF_04_19_00_ADD	RF_03_RF_03_SP_00_ADD	DM_03_DM_05_17_08_MUL	DM_04_DM_06_16_08_MUL	
66	RF_08_RF_02_DM_11_ADD	RF_04_RF_04_19_07_ADD	DM_05_DM_07_17_09_MUL	DM_06_DM_08_16_09_MUL	
67		RF_11_RF_11_18_00_ADD	DM_01_DM_04_17_10_MUL	DM_02_DM_05_16_10_MUL	
68	RF_09_RF_03_DM_12_ADD	RF_05_RF_05_19_01_ADD	DM_03_DM_06_17_00_MUL	DM_04_DM_07_16_00_MUL	
69	RF_04_RF_10_18_01_ADD	RF_06_RF_06_19_02_ADD	DM_05_DM_08_17_01_MUL	DM_06_DM_09_16_01_MUL	
70	RF_06_RF_09_18_04_ADD	RF_12_RF_12_SP_01_ADD	DM_07_DM_10_19_03_MUL	DM_01_DM_05_17_02_MUL	
71	RF_10_SP_00_19_05_ADD	RF_07_RF_07_18_02_ADD	DM_02_DM_06_16_02_MUL	DM_03_DM_07_17_03_MUL	
72		RF_08_RF_08_SP_02_ADD	DM_04_DM_08_16_03_MUL	DM_05_DM_09_17_04_MUL	
73	RF_01_RF_00_19_08_ADD	RF_09_RF_09_18_03_ADD	DM_06_DM_10_16_04_MUL	DM_01_DM_06_17_05_MUL	
74	RF_00_RF_01_DM_13_ADD	RF_10_RF_10_18_05_ADD	DM_02_DM_07_16_05_MUL	DM_03_DM_08_17_06_MUL	
75	RF_07_RF_04_18_07_ADD	RF_00_RF_00_19_04_ADD	DM_04_DM_09_16_06_MUL	DM_05_DM_10_18_06_MUL	
76	RF_02_SP_01_19_06_ADD	RF_01_RF_01_18_09_ADD	DM_01_DM_07_17_07_MUL	DM_02_DM_08_16_07_MUL	
77	SP_02_RF_02_19_09_ADD		DM_03_DM_09_17_08_MUL	DM_04_DM_10_16_08_MUL	
78		RF_02_RF_02_18_00_ADD	DM_01_DM_08_17_09_MUL	DM_02_DM_09_16_09_MUL	
79	RF_03_RF_05_18_01_ADD	RF_03_RF_03_19_01_ADD	DM_03_DM_10_16_11_MUL	DM_01_DM_09_17_10_MUL	
80	RF_05_RF_07_DM_14_ADD	RF_04_RF_04_18_04_ADD	DM_02_DM_10_16_10_MUL	DM_01_DM_10_DM_20_MUL	
81	RF_08_RF_08_18_10_ADD	RF_05_RF_05_SP_00_ADD	DM_07_DM_09_17_30_MUL	DM_08_DM_10_16_30_MUL	
82	RF_04_RF_09_19_02_ADD	RF_06_RF_06_18_02_ADD			
83	RF_09_RF_03_SP_63_ADD	RF_07_RF_07_19_07_ADD			
84	RF_01_RF_00_19_03_ADD	RF_08_RF_08_18_05_ADD			
85		RF_09_RF_09_17_00_ADD			
86	RF_06_RF_10_DM_15_ADD				
87	SP_00_RF_06_19_00_ADD				
88		RF_10_RF_10_DM_01_ADD			
89	RF_03_RF_04_DM_18_ADD	RF_30_RF_30_18_31_ADD			
90	RF_07_RF_05_DM_20_ADD	RF_00_RF_11_DM_00_ADD			
91	RF_02_RF_01_DM_17_ADD				
92	RF_00_RF_02_DM_19_ADD				
93					
94	SP_63_RF_31_DM_16_ADD				

A.2 Access method for Memory Testing

Parameter

```

all_one  = 32'hFFFFFFFF // 11111111
all_zero = 32'h00000000 // 00000000
one_zero = 32'hAAAAAAAA // 10101010
zero_one = 32'h55555555 // 01010101

```

Because only odd bank of memory could be directly access, we separate memory testing between odd and even bank. As shown in table A.5, In1 and In2 are input data. According to the testing flow in chapter 3, In1 and In2 are both equal to all_one, parameter listed above, in first stage. The In1 and In2 change to all_zero, one_zero, and zero_one in the following three stages. In the fifth stage, In1 is one_zero and In2 is zero_one to get mixing input. Then the data is read out the same as table A.5.

Table A.5: Access Table for Odd Bank Memory

Bank Address	D9	D7	D5	D3	D1
0	In1	In1	In1	In1	In1
1	In2	In2	In2	In2	In2
2	In1	In1	In1	In1	In1
3	In2	In2	In2	In2	In2
4	In1	In1	In1	In1	In1
5	In2	In2	In2	In2	In2
6	In1	In1	In1	In1	In1
7	In2	In2	In2	In2	In2
8	In1	In1	In1	In1	In1
9	In2	In2	In2	In2	In2
10	In1	In1	In1	In1	In1
11	In2	In2	In2	In2	In2
12	In1	In1	In1	In1	In1
13	In2	In2	In2	In2	In2
14	In1	In1	In1	In1	In1
15	In2	In2	In2	In2	In2
16	In1	In1	In1	In1	In1
17	In2	In2	In2	In2	In2
18	In1	In1	In1	In1	In1

19	In2	In2	In2	In2	In2
20	In1	In1	In1	In1	In1
21	In2	In2	In2	In2	In2
22	In1	In1	In1	In1	In1
23	In2	In2	In2	In2	In2
24	In1	In1	In1	In1	In1
25	In2	In2	In2	In2	In2
26	In1	In1	In1	In1	In1
27	In2	In2	In2	In2	In2
28	In1	In1	In1	In1	In1
29	In2	In2	In2	In2	In2
30	In1	In1	In1	In1	In1
31	In2	In2	In2	In2	In2

The even bank memory could only be access by writing. They can't be read out directly and must be read with indirect method. Thus the table A.6 only represents the writing data to these even bank memory. The input data is the same as the odd bank memory testing with five stages flow. After the data is written into all even bank memory, the D8 and D6 are access by ALU to perform operation of adding by zero and write its result to D9 and D7. The data of D4 and D2 are in the similar way to be access by MUL to execute operation of multiply by one and write back to D5 and D3. Finally, the D0 is performed by divider and responses to D1. Therefore, the results are able to be access the same as table A.5.

Table A.6: Access Table for Even Bank Memory

Bank Address	D8	D6	D4	D2	D0
0	In1	In1	In1	In1	In1
1	In2	In2	In2	In2	In2
2	In1	In1	In1	In1	In1
3	In2	In2	In2	In2	In2
4	In1	In1	In1	In1	In1
5	In2	In2	In2	In2	In2
6	In1	In1	In1	In1	In1
7	In2	In2	In2	In2	In2
8	In1	In1	In1	In1	In1
9	In2	In2	In2	In2	In2

10	In1	In1	In1	In1	In1
11	In2	In2	In2	In2	In2
12	In1	In1	In1	In1	In1
13	In2	In2	In2	In2	In2
14	In1	In1	In1	In1	In1
15	In2	In2	In2	In2	In2
16	In1	In1	In1	In1	In1
17	In2	In2	In2	In2	In2
18	In1	In1	In1	In1	In1
19	In2	In2	In2	In2	In2
20	In1	In1	In1	In1	In1
21	In2	In2	In2	In2	In2
22	In1	In1	In1	In1	In1
23	In2	In2	In2	In2	In2
24	In1	In1	In1	In1	In1
25	In2	In2	In2	In2	In2
26	In1	In1	In1	In1	In1
27	In2	In2	In2	In2	In2
28	In1	In1	In1	In1	In1
29	In2	In2	In2	In2	In2
30	In1	In1	In1	In1	In1
31	In2	In2	In2	In2	In2

APPENDIX B

Memory Map

HADDR[13:0] is used to access data and instruction memory. The bank selection of these two memory is shown in table B.1. They are selected by HADDR[12:9]. Form 4'b0000 to 4'b1110, it could select fifteen banks of data and instruction memory. The last blank of 4'b1111 will be used to store the end value of pc_counter. The other 7 bits, HADDR[8:2] is used as address bits. There is one thing needed to be reminded. The depth of data memory is thirty-two and the depth of instruction is one hundred and twenty-eight. Thus, it takes five bits for data memory and seven bits for instruction memory.

Table B.1: Destination bank of memory,
D for data memory and I for instruction memory

Destination Memory	HADDR [12:9]
D9 (left bank of ALU_0)	0000 _b
D8 (right bank of ALU_0)	0001 _b
D7 (left bank of ALU_1)	0010 _b
D6 (right bank of ALU_1)	0011 _b
D5 (left bank of MUL_0)	0100 _b
D4 (right bank of MUL_0)	0101 _b
D3 (left bank of MUL_1)	0110 _b
D2 (right bank of MUL_1)	0111 _b
D1 (left bank of DIV_0)	1000 _b
D0 (right bank of DIV_0)	1001 _b
I5	1010 _b
I4	1011 _b
I3	1100 _b
I2	1101 _b
I1	1110 _b
End of Pc_counter	1111 _b

HADDR[1:0] is used for byte and half word access. HADDR[13] is connected to original port that is mem_d_ctrl in older ALU cluster.