國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

長管線延遲資料路徑之高面積效率設計與實現

Area-Efficient Design and Implementation of Deep-Pipeline Latency Datapath

研究生： 呂進德

指導教授： 劉志尉

中 華 民 國 九 十 七 年 十 一 月

長管線延遲資料路徑之高面積效率設計與實現

Area-Efficient Design and Implementation of Deep-Pipeline Latency Datapath

研 究 生：呂進德　　Student: Chin-Te Lu

指導教授：劉志尉 博士　　Advisor: Dr. Chih-Wei Liu

國 立 交 通 大 學

電子工程學系 電子研究所班

碩士論文

A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

In partial Fulfillment of the Requirements for the Degree of

Master of Science

in

Electronics Engineering

November 2008

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 七 年 十一 月

長管線延遲資料路徑之高面積效率設計與實現

研究生：呂進德　　　　　　　　　　指導教授：劉志尉 博士

國立交通大學
電子工程學系 電子研究所

摘要

處理器的資料路徑(datapath)通常是影響其效能的最重要部分。隨著不同應用需求，資料路徑的配置與設計也會不同，一般說來，針對高效能處理器，例如 Intel Pentium 處理器、IBM Cell 處理器等，設計者會藉由各種 VLSI 技術，盡可能的提高資料路徑的操作頻率；但另一方面，對於輕量化(lightweight)應用、如嵌入式系統(embedded system)，則會以追求低功率、低晶片面積等方向做最佳化資料路徑設計。同一套指令集架構(instruction set architecture)對於不同的應用而言會有不同的資料路徑設計，針對此，本論文提出一套能針對不同效能需求，而能自動合成一具高面積效率的資料路徑設計流程。此具高面積效率資料路徑產生器，其中包含兩個動作：空間和時間維度做最佳化設計。此具高面積效率資料路徑產生器可延用現有的高效能處理器的指令集、如 IBM Cell，和其相關發展軟體與應用程式，並根據應用所需的效能，有系統的對處理器資料路徑做最佳化。空間維度上的最有效率的應用意指資料共享路徑，包含建立函數模型(function modeling)和週期準確模型(cycle-accurate modeling)設計。另一方面，我們也會針對時間維度上做最佳化，並分析指令的延遲(latency)時間，系統化地建立數學方程式以獲得最小面積的微架構(micro-architecture)。我們以 Cell SPU(Synergistic Processor Unit)資料路徑設計為例，利用所提出的設計流程分析指令集架構，尋找出最高面積效率的微架構。實驗顯示，針對 100MHz 到 800MHz 的嵌入式微處理器的資料路徑設計，我們所提出的設計流程比自動化工具改善約 20%的面積。在 UMC 90nm 的製程下，我們利用前述的設計流程實作 SPU 數位訊號處理器，晶片面積為 2.5mm×2.5mm，而其操作頻率為 400MHz。

Area-Efficient Design and Implementation of Deep-Pipeline Latency Datapath

Student: Chin-Te Lu Advisor: Dr. Chih-Wei Liu

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University

ABSTRACT

Datapath is primarily the most critical element that affects performance. The allocations and design of datapath depends various application requirements. General speaking, for high-performance processors like Intel's Pentium Processors, IBM's Cell Processors and so on, the designers extremely rise up operating frequency by board VLSI techniques. On the contrary, such as lightweight applications in the embedded system, the goal of datapath design is to seek low-power, small chip area and so on. The instruction set architecture (ISA) has different ways of implementation for different application requirements. Therefore, this thesis proposes the design flow to automatically generate the area-efficient datapath for various application requirements. The area-efficient datapath generator includes the two-phased including spatial-optimized and temporal-optimized for datapath optimization. It can systematically develop and optimize datapth of the processors while leveraging the instruction set architecture (ISA) of high performance processor like IBM's Cell and the software toolchain and application programs. Spatial-optimized means that efficient utilization in spatial domain including function modeling and cycle-accurate design. In other phase, temporal-optimization explores the instruction latency to systematically build up mathematical formulation to get the optimal micro-architecture. We take the Cell synergistic processor unit (SPU) as our datapath design example to analyze the optimization space of SPU ISA implementation, and find the area-efficient micro-architecture by using our proposed design flow. In the experiment, the micro-architecture by using our proposed design flow improves about 15-20% of area compared to using CAD tools for datapath design of embedded processors targeted 100MHz to 800MHz. Finally, we use the previous design flow to implement the SPU DSP in the UMC 90nm 1P9M CMOS process. The silicon area is 2.5mm$\times$2.5mm and the clock rate is 400MHz.

誌　　謝

研究生涯轉眼即逝，兩年來受到許多人幫助及鼓勵，才能順利完成碩士學業，在此致上最深的感激。

首先，我要感謝劉志尉老師在我的專業知識和研究態度給予熱誠的指導，使我在這兩方面更臻成熟，老師的豐富學養及學者風範，令我受益良多。特別感謝任建葳教授、周景揚教授及周世傑教授，謝謝你們在百忙之中，撥冗參與論文口試，並對我的研究給予寶貴的意見，讓此篇論文更加完備充實。

另外，我還要感謝林泰吉學長不厭其煩地對我的研究工作步步導引，並培養研究態度以及應有的態度。以及歐士豪學長給我諸多細節的解惑，還有林彥呈同學和甘禮源學弟對我的研究提出意見和討論和諸多的協助。

感謝實驗室學長、同學及學弟妹們。感謝陳信凱、郭羽庭、林禮圳、林佑昆和張彥中，感謝學長們在研究生生涯中的各項協助及鼓勵。感謝張國強、莊明勳、葉世賢、吳聲昀、張雅婷及蔡安綺，謝謝學弟妹們在研究工作上的一切幫忙。

感謝和我一起打拼的顏于凱、洪正堉、李岳泰、張巍瀚。這兩年，我們一同經歷了挑燈夜戰的努力，也共同分享研究成果的喜悅。

最後，感謝我最親愛的家人。爸、媽、妹，感謝你們一路上的支持及鼓勵，沒有你們就沒有今日的我，我愛你們。

謹將此篇論文獻給所有曾支持我、協助我的人，衷心的感謝並祝福你們。

進德

謹誌於 新竹

2008 冬

CONTENTS

1 INTRODUCTION ... 1
1.1 Motivation ... 2
1.2 Problem Description and Distribution ... 2
1.3 Thesis Organization ... 4
2 BACKGROUND ... 5
2.1 Cell Broadband Engine Architecture ... 6
2.2 SPU Instruction Set Architecture ... 10
2.3 SPU Micro-Architecture ... 18
3 DESIGN & OPTIMIZATION FLOW OF DEEP-PIPELINE LATENCY DATAPATH ... 23
3.1 Spatial Optimization ... 24
3.1.1 Function Modeling ... 24
3.1.2 Cycle-Accurate Modeling ... 29
3.2 Temporal Optimization ... 34
3.3 Experimental Results ... 41
4 SILICON IMPLEMENTATION ... 49
4.1 Implementation Design Flow ... 50
4.2 Implementation Result ... 52
5 CONCLUSION & FUTURE WORKS ... 55
REFERENCES ... 57

LIST OF FIGURES

Figure 1-1 Latency exploration 3
Figure 2-1 Die photo of Cell Broadband Engine 6
Figure 2-2 Block diagram of CBE processor 7
Figure 2-3 PPE block Diagram 8
Figure 2-4 SPE architecture 9
Figure 2-5 SPU functional units 10
Figure 2-6 Instruction format 11
Figure 2-7 Register layout of data types and preferred scalar slot 12
Figure 2-8 Example of addition operation 14
Figure 2-9 Example of multiply operation 15
Figure 2-10 Example of form select mask for bytes operation 17
Figure 2-11 Example of shuffle bytes operation 18
Figure 2-12 SPU organization 20
Figure 2-13 SPU pipeline diagram 21
Figure 3-1 Overview of our proposed design flow 24
Figure 3-2 Functional modeling 25
Figure 3-3 Example of behavioral assignment in RTL 26
Figure 3-4 Add/Sub functional unit 28
Figure 3-5 Cycle-accurate modeling 29
Figure 3-6 Tolerable latencies 31
Figure 3-7 Forwarding network 33
Figure 3-8 SPU Pipeline diagram 33
Figure 3-9 Temporal optimization 34
Figure 3-10 Function unit with 3-cycle latency 35
Figure 3-11 Piped S/R FU 36
Figure 3-12 Piped Shuffle FU 37
Figure 3-13 Piped MUL FU 38
Figure 3-14 Timing delay of one-stage pipelined datapath 41
Figure 3-15 Proposed design flow 42
Figure 3-16 Improvement by our proposed design flow 47
Figure 4-1 Implementation flow 50
Figure 4-2 Our SPU interface 51
Figure 4-3 Pipeline diagram of our SPU 52
Figure 4-4 Implementation result 53

LIST OF TABLES

Table 2-1 Binary values in register RC and byte results 17

Table 2-2 Dual issue unit assignments 19

Table 2-3 Unit and instruction latency 20

Table 3-1 Synthesis result of baseline 27

Table 3-2 Instruction latency 30

Table 3-3 Forwarding table of our SPU 32

Table 3-4 Piped S/R FU 36

Table 3-5 Piped Shuffle FU 37

Table 3-6 Piped MUL FU 38

Table 3-7 Number ID of functional unit 39

Table 3-8 Description of equation’s parameter 40

Table 3-9 Synthesized result of baseline and spatial-optimized 43

Table 3-10 Comparison between baseline and spatial-optimized 44

Table 3-11 Improvement by spatial-optimized 44

Table 3-12 All cases for latency spec 45

Table 3-13 Temporal optimization 46

Table 3-14 Area reduction from temporal optimization 47

Table 4-1 Synthesis result 53

1 INTRODUCTION

Today's system-on-a-chip (SoC) has advanced rapidly, and there exists many design considerations, such as time-to-market, production cost, operation speed and so on. These demand the different performance requirement such as, low power for portable devices, small area, and high operating frequency such as computing-intensive for workstation and so on.

In the meanwhile, the cost of software development is more and more expensive in many embedded systems. It is not efficient time-to-market to develop the software and hardware at the same time. By this motivation, we try to develop the hardware for different performance requirement under the software support. In this thesis, we focus on developing processors under different performance requirement while leaving the existing software in order to shrink the time-to-market and then explore the micro-architecture optimization space of the specific ISA implementation

1.1 Motivation

With the increasing performance requirement for system-on-a-chip (SoC) applications, such as lower power, small area, and high operating frequency, developing these applications for many performance requirements is not time-consuming. In the meanwhile, the software development is more and more expensive in the embedded system. However, we can reuse the existing software to develop the hardware for the various performance requirements. That's means that we can reduce the TTM (time-to-market) to develop the hardware for many performance requirements. We exploit the same ISA with suitable implementation can help to reduce design cost.

The Cell Broadband Engine (CBE) is very popular. It provides the open and full software support. Therefore, we can take it into consideration to develop the hardware under its software support. But its datapath is for extremely high-performance. There is a trade-off between the performance and the area. If we design the hardware for low performance compared to Cell processor, such as targeted to several hundred MHz. The original datapath of Cell processor is not the most area-efficient for lower performance. However, that's mean that the same instruction set architecture (ISA) has different ways of implementation for different performance requirement.

1.2 Problem Description and Distribution

With the growing computing requirement, DSPs are becoming prevalent solutions in multimedia applications and telecommunications. In order to save time-to-market, we can develop the processors under different performance requirements with the existing software toolchain. By this above motivation, we can save the design time of software development to develop DSPs for various applications with software support. For example, the software toolchain of the famous Cell Broadband Engine (CBE) is ready to develop the processor with the instruction set architecture (ISA) for various applications in the embedded system.

ISA is the interface between hardware and software. In fact, ISA implementation depends on the various application requirements. That's mean that different ISA implementations have different micro-architecture designs under the target applications. In other words, there are different optimization spaces under various applications. For example, the micro-architecture targeted to several hundred MHz under the same ISA implementation with the binary-compatible software. We can find that the Cell SPU expose the long latency for high-performance and expose the long latency for datapath optimization as show in Figure 1-1. There are three ways for microarchitecture design. We'll propose two-phased design flow to design area-efficient micro-architecture under this constraint.

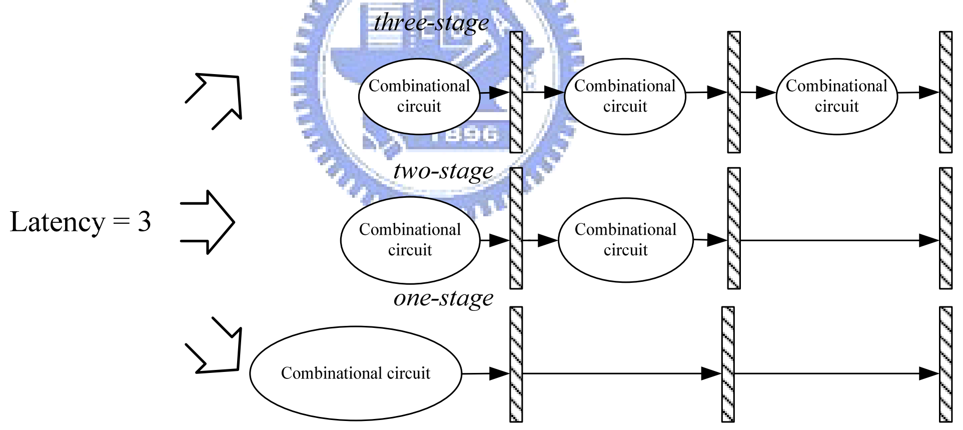


Figure 1-1 Latency exploration

In this thesis, we propose two-phased area-efficient design flow of ISA implementation for DSPs under binary-compatible software. We take the Cell SPU as our design example. Because the Cell SPU is the data-oriented processor, there is cleanly much more optimization space than control-oriented processor, such ARM processors. Our proposed two-phased area-efficient design flow includes spatial optimization and temporal optimization. This two-phased design flow provides the systematical area-efficient micro-architecture design.

Compared with ad-hoc method, using our proposed design flow saves about 20% of area under 100MHz to 800MHz timing constraints.

1.3 Thesis Organization

This thesis focuses primarily on two-phased systematical design flow of processor: Spatial optimization and temporal optimization. This thesis is organized as follows.

Chapter 2 introduces the Cell SPU which includes Cell Broadband Engine Architecture (CBEA), Synergistic Processor Unit (SPU), SPU instruction set architecture (ISA), and SPU micro-architecture. Chapter 3 first describes the first-phased design flow including function modeling and cycle-accurate modeling. This phase design flow is mainly spatial optimization while the second-phased is temporal optimization by formulating mathematical formulation. At last of this chapter, we list the experimental results of our proposed design flow.

Chapter 4 shows the silicon implementation results by using our proposed design flow target to 400MHz. Finally, chapter 5 concludes this thesis and points out the direction of future research.

2 BACKGROUND

Contemporary DSPs are multimedia-rich, involving significant amounts of audio and video processing. Cell Broadband Engine (CBE) processor provides a high-performance for applications in media-rich consumer-electronic devices. This chapter provides background information related to this thesis. Chapter 2.1 introduces the Cell Broadband Engine (CBE) and synergistic processing unit (SPU). Chapter 2.2 and Chapter 2.3 give an overview of the synergistic processing unit (SPU) instruction set architecture (ISA) and micro-architecture respectively.

2.1 Cell Broadband Engine Architecture

The Cell Board Engine (CBE) is the first implementation of a new multiprocessor family conforming to the Cell Broadband Engine Architecture (CBEA, or informally, “Cell”). The CBEA is a new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the CBE are multicore processors jointly developed by SONY, Toshiba, and IBM, known as STI [4]. Figure 2-1 is a die photo of the Cell BE.

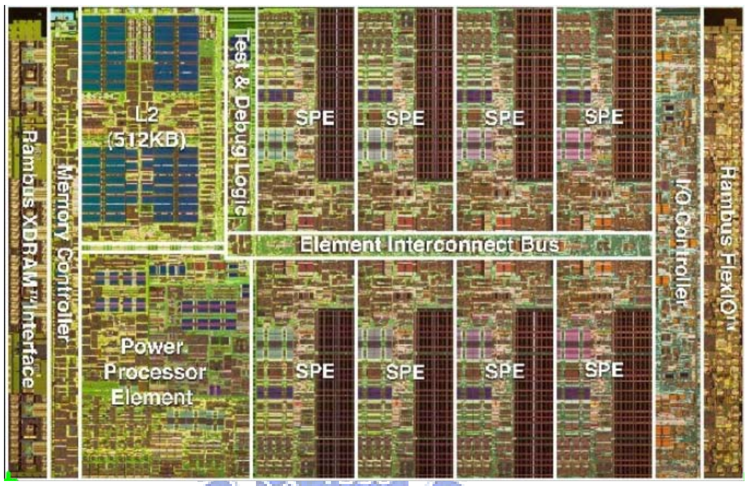


Figure 2-1 Die photo of Cell Broadband Engine

Although the CBE processor is initially intended for multimedia applications in media-rich consumer-electronics devices such as game consoles, the architecture has been designed to extend fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

Figure 2-2 [5] shows the block diagram of Cell processor. The most distinguishing feature is that the CBE processor is a multi-core with 9 processor elements and a shared coherent memory on-a-chip: the Power Processor Element (PPE) and the Synergistic Processor Element (SPE). The CBE processor has one PPE and eight SPEs. There is a mutual dependence between the PPE and the SPEs. The PPE is responsible for running the operating system and coordinating the flow of the data processing threads through the SPEs. This differentiation allows the architectures and implementations of the PPE and SPE to be optimized for their respective workloads and enables significant improvements in performance per transistor.

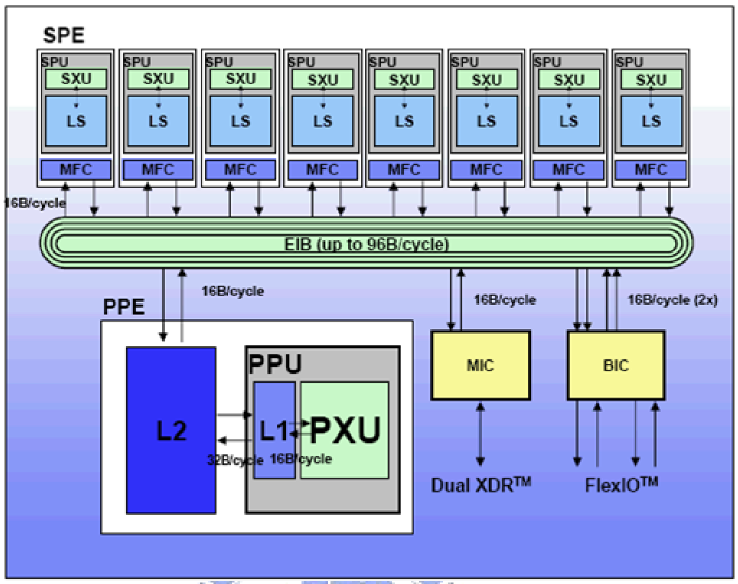


Figure 2-2 Block diagram of CBE processor

◆ PowerPC Processing Elements

The PowerPC Processor Element (PPE) is a 64-bit PowerPC Architecture core optimized for design frequency and power efficiency. It is a general-purpose, dual-thread, 64-bit RISC processor with vector/SIMD extensions. The PPE is responsible for overall control of a CBE system. It runs the operating system for all applications running on PPE and Synergistic Processor Elements (SPEs). The PPE consists of two main units as shown in Figure 2-3 [6], The PowerPC processor unit (PPU) is the computation unit, and the PowerPC processor storage subsystem (PPSS) is for the purpose of storage. More detail information about PowerPC Processing Elements is in [6].

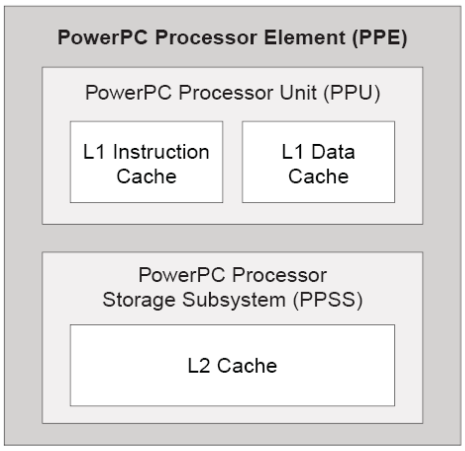

PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L1 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

Figure 2-3 PPE block Diagram

◆ Synergistic Processor Elements

The eight Synergistic Processor Elements (SPEs) execute a new single instruction multiple data (SIMD) instruction set-the Synergistic Processor Unit Instruction Set Architecture. They are independent processors, each running an independent application thread. Each SPE is a 128-bit RISC processor for data-rich, compute-intensive applications and includes a private local store for efficient data and instruction access. Figure 2-4 [6] shows the major elements of the SPE architecture and their relationship. Local storage (LS) is a private memory for SPE instructions and data. The synergistic processor unit (SPU) core is a processor that runs instructions from the LS and can read from or write to the local storage (LS). The direct memory access (DMA) unit transfers data between LS and system memory. The channel unit is a message-passing interface that allows the SPU core to communicate with both the DMA unit and other devices in the Cell processor.

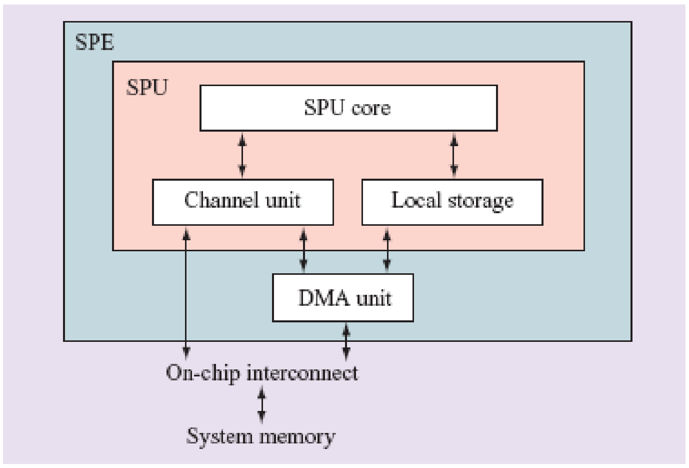


Figure 2-4 SPE architecture

The SPU core is a single-instruction multiple-data (SIMD) reduced instruction set computing (RISC) processor [7]. All instructions are encoded in 32-bit fixed-length instruction formats. The SPU feature 128 general-purpose registers (GPRs) that are used by both floating and integer instructions. Most instructions operate on 128-bit-wide data that perform integer arithmetic, logical operations, loads, stores, compares, and branches. The main SPU functional unist are shown in Figure 2-5 [6]. These include the synergistic execution unit (SXU), the LS, and the SPU register file unit (SRF). The SPU issues two instructions to its two execution pipelines respectively. The pipelines are referred to as even (pipeline 0) and odd (pipeline 1). These units execute the following types of operations:

- Odd Pipeline
  - SPU Odd Fixed-Point Unit (SFS) — Executes byte shift, rotate mask, and shuffle operations on quadwords
  - SPU Load and Store Unit (SLS) — Executes load and store instructions and hint for branch instructions. It also handles DMA requests to the LS
  - SPU Control Unit (SCN) — Fetches and issues instructions to the two pipelines. It performs control functions such as branch instructions, arbitration of access to the

LS and register file, etc.

- SPU Channel and DMA Unit (SSC) — Manages communication, data transfer, and control into and out of the SPU.

- Even Pipeline
  - SPU Even Fixed-Pointed Unit (SFX) — Executes arithmetic instructions, logical instructions, word SIMD shifts and rotations, floating-point comparisons, and floating-point reciprocal and reciprocal square-root estimations.
  - SPU Floating-Point Unit (SFP) — Executes single-precision and double-precision floating point instructions, and conversions, and byte operations. The 32-bit multiplier are implemented in software using 16-bit multiplies.

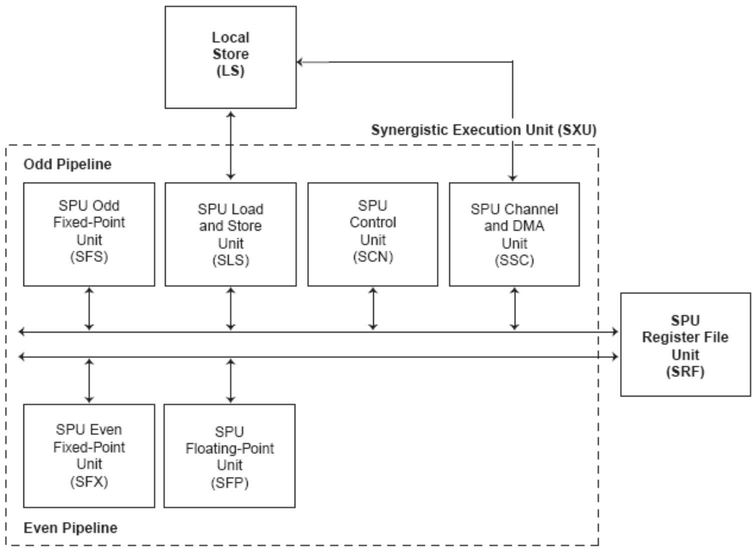

Local Store (LS)
Synergistic Execution Unit (SXU)
Odd Pipeline
SPU Odd Fixed-Point Unit (SFS)
SPU Load and Store Unit (SLS)
SPU Control Unit (SCN)
SPU Channel and DMA Unit (SSC)
SPU Register File Unit (SRF)
SPU Even Fixed-Point Unit (SFX)
SPU Floating-Point Unit (SFP)
Even Pipeline

Figure 2-5 SPU functional units

2.2 SPU Instruction Set Architecture

The instruction set architecture (ISA) is the most important design issue that DSP designer must get right from the start. Instruction set architecture (ISA) serves as an abstraction layer between hardware and software. It should include the following information, instruction sets, instruction format, data representation, data storage, address modes, and exceptional conditions. In the following section, the fixed point SPU Instruction set architecture (ISA) [8] will be described.

◆ Instruction formats

There are six basic instruction formats. These instructions are all 32-bit long. Instructions in memory must be aligned on word boundaries. The instruction formats shown in Figure 2-6.

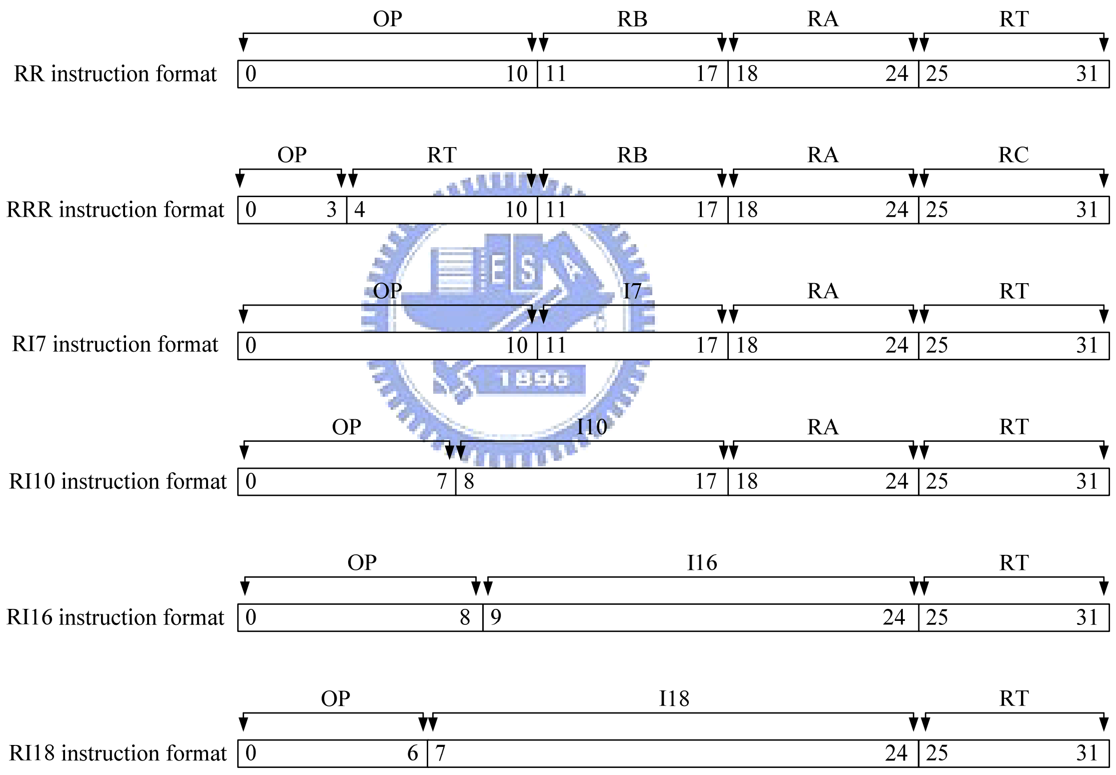

OP RB RA RT
RR instruction format 0 10 11 17 18 24 25 31
OP RT RB RA RC
RRR instruction format 0 3 4 10 11 17 18 24 25 31
OP I7 RA RT
RI7 instruction format 0 10 11 17 18 24 25 31
OP I10 RA RT
RI10 instruction format 0 7 8 17 18 24 25 31
OP I16 RT
RI16 instruction format 0 8 9 24 25 31
OP I18 RT
RI18 instruction format 0 6 7 24 25 31

Figure 2-6 Instruction format

◆ Data representation

The SPU hardware supports the following data types: Byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit), and quadword (128-bit) as shown in Figure 2-7. All GPRs (general-purpose resisters) are 128-bit wide. The leftmost word (bytes 0, 1, 2, and3) of a register is called the preferred slot. When instructions use or produce scalar operands or addresses, the values are in the preferred slot. Because the SPU accesses its LS a quadword at a time, there is a set of store-assist instructions for insertion of bytes, halfwords, words, and doublewords into a quadword for a subsequent load/store.

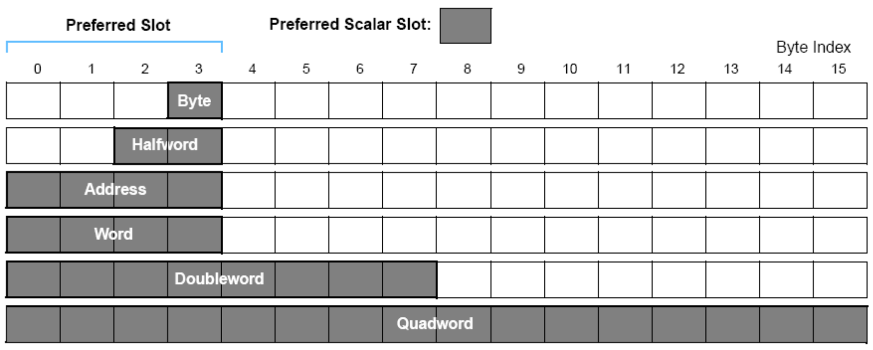


Figure 2-7 Register layout of data types and preferred scalar slot

◆ Data storage

The SPU architecture defines a private memory, also called local storage, which is byte-addressed load and store instructions combined operands from one or two registers or immediate value to form the effective address of the memory operand. The LS is 256 KB, single-ported, non-caching memory. It stores all instructions and data used by the SPU. SPU data-access bandwidth is 16 bytes per cycle, quadword aligned.

◆ Addressing modes

All instructions, except branches, generate address by incrementing a program counter. For load and store instructions that specify a base register, the effective address in memory for a data value is calculated relative to the base register in one of three ways:

■ Resister + Displacement

The displacement (D) forms of the load and store instructions form the sum of a displacement specified by the sign-extended 16-bit immediate field of the instruction plus the contents of the base resister.

■ Register + Register

The index (X) forms of the load and store instructions form the sum of the contents of the index register plus the contents of the base register

■ Register

The load string immediate and store string immediate instructions use the unmodified contents of the base register

◆ Instruction sets

The SPU instruction set used are instructions that are 4 bytes long and word-aligned. It supports 16-byte (128-bit) operand accesses between storage and its 128 registers. For a brief overview of the fixed point SPU instruction set, including data transfer, integer, logical, data transformation.

■ Data transfer instructions

In order to process data in the memory, load/store machine use the load and store instruction to handle memory access issues. Load and store instructions combine operands from one or two registers and an immediate value to form the effective address of the memory operand. Only aligned 16-byte-long quadwords can be loaded and stored. Therefore, the rightmost 4 bits of an effective address are always ignored and are assumed to be zero.

■ Integer and logical instructions

● Addition/subtraction instructions

The instructions of addition or subtraction are the operators of halfword (16-bit) or word (32-bit) of SIMD version. "A" is the word-operator that replaces the destination operand with the sum of the two source registers as shown in Figure 2-8, while "ai" takes one source operand as 128-bit immediate data. "Sf" and "sfi" perform general and immediate subtraction. The 32-bit SIMD version of "A" is supported by the SPU instruction set and shown in Figure 2-8.

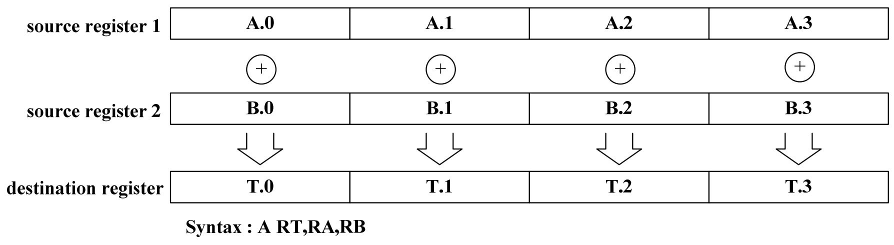


Figure 2-8 Example of addition operation

- Compare instructions

Compare instructions compare the two source operands and store the destination to register. The source operands can be registers or immediate data. It is the operators of byte (8-bit) or halfword (16-bit) or word of SIMD version. For example, "ceqb" (compare equal byte) set the byte-result as 0xFF if the source operand 1 is equal to source operand 2 and set 0x00 vice versa.

- Multiply instructions

Multiply-relative instructions combine multiply and multiply-and-accumulator instructions. These multiply instructions only support 16-bit SIMD multiplication which get the lower or upper part of one word in each register to take the multiply operation and the product maybe be shifted, mask upper or lower, or the additional accumulation with it. For example, multiply-high gets the result of the leftmost 16 bits of the value in one word of register RA are multiplied by of the rightmost 16 bits of the value in one word of register RB, and then the product is shifted left by 16 bits and zero are shifted in at the right for each of four word slots as shown in Figure 2-9.

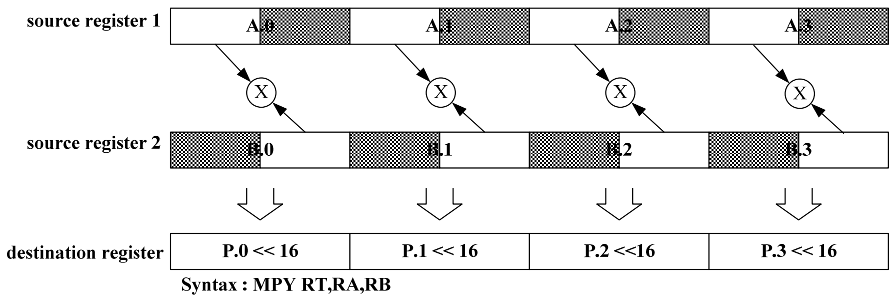


Figure 2-9 Example of multiply operation

- Logical instructions

Logical instructions handle bit-wise Boolean logical operations. The logical operations are composed of AND, OR, XOR, NAND, NOR, and XOR instructions. These instructions perform the general logical operation in the processor.

■ Data transformation instructions

To support the data alignment of application processing, data transformation instructions are include shift/rotator, extend, form, gather and shuffle.

- Shifter / rotator instructions

The shift instruction can shift the source operand arithmetically or logically. It can specify the shift amount in the ways, either register or immediate. It support shift of halfword, word, and quadword, and shift quadword by byte. The rotator instructions also support the same as the above operation.

- Extend instructions

The extend instruction is used to support the data precision. These instructions support byte (8-bit) to halfword (16-bit), halfword (16-bit) to word (32-bit), and word (32-bit) to double word (64-bit). For example, the operation of "xsbh" (extend sign byte to halfword) is that for each of eight halfword slots, the sign of the byte in the right byte of the operand in register RA is propagated to the left byte.

- Gather instructions

The gather instruction is include gather bits from bytes, halfwords, or words. This operation can be used to gather bits of the leftmost bit of one byte, halfword, or word. For example, "gbb" (gather bits from bytes) operates as the following description: a 16-bit quantity is formed in the right half of the preferred slot of register RT by concatenating the rightmost bit in each byte of register RA. The leftmost 16 bits of register RT are extending to zero as the remaining slots of register RT.

- Form instructions

The Form instructions are to create a mask by replicating the rightmost bit of bytes, halfwords, and words. For example, "fsmb" (form select mask for bytes) operates as the following description: the right 16-bit of the preferred slot of register RA are used to create a mask in register RT by replicating each bit eight times. Bits in the operand are related to bytes in the result in a left-to-right correspondence as shown in Figure 2-10.

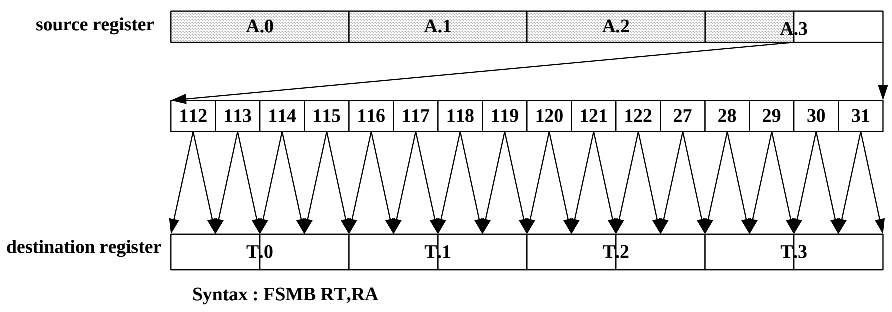


Figure 2-10 Example of form select mask for bytes operation

- Shuffle instructions

The shuffle operation is extremely powerful and finds its way into many applications in which data reordering, selection, or merging is required. Its operation is that register RA and RB are logically concatenated with the least-significant bit of RA adjacent to the most-significant bit of RB. The bytes of the resulting value are considered to be numbered from 0 to 31. For each byte slot in registers RC and RT, the value in register RC is examined, and a result byte is produced as shown in Table 2-1 and Figure 2-11, and then the result byte is inserted into register RT. Other instructions which are not above are enumerated in [8].

Table 2-1 Binary values in register RC and byte results

Value in Register RC (expressed in binary)	Result Byte
10xxxxxx	0x00
110xxxxx	0xFF
111xxxxx	0x80
otherwise	shown in figure 2-11

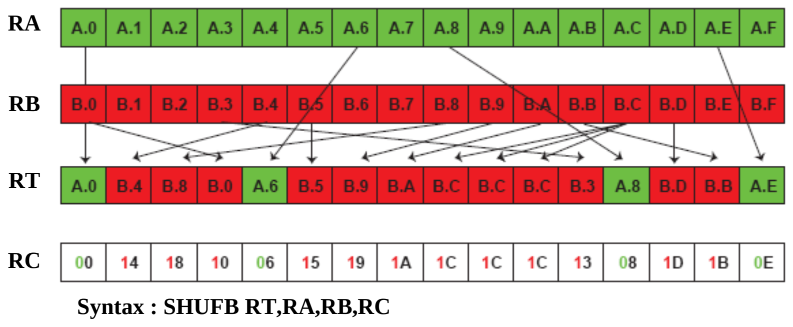

RA
A.0 A.1 A.2 A.3 A.4 A.5 A.6 A.7 A.8 A.9 A.A A.B A.C A.D A.E A.F
RB
B.0 B.1 B.2 B.3 B.4 B.5 B.6 B.7 B.8 B.9 B.A B.B B.C B.D B.E B.F
RT
A.0 B.4 B.8 B.0 A.6 B.5 B.9 B.A B.C B.C B.C B.3 A.8 B.D B.B A.E
RC
00 14 18 10 06 15 19 1A 1C 1C 1C 13 08 1D 1B 0E
Syntax : SHUFB RT,RA,RB,RC

Figure 2-11 Example of shuffle bytes operation

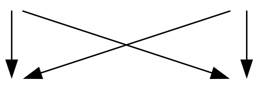
◆ Exceptional conditions

The SPU support a single interrupt handler. The entry point for this handler is address 0 in local store. When a condition is present and interrupts are enabled, the SPU branches to address 0 and disables the interrupt facility. The address of the next instruction to be executed is saved in the SRR0 register. The iret instruction can be used to return from the handler.

2.3 SPU Micro-Architecture

Figure 2-12 [7] shows how the SPU is organized and the key bandwidth (per cycle) between units. Instructions are fetched from the LS in 32 4-byte groups when LS is idle. The fetched lines are sent in two cycles to the instruction line buffer (ILB). Instructions are sent, two at a time, from the ILB to the issue control unit. The SPU issues and completes all instructions in program order and doesn't reorder or rename its instructions. Although the SPU isn't a VLIW processor, it does feature like dual feature and can issue up to two instructions per cycle to nine execution units organized into two pipelines as shown in Table 2-1. Instructions pairs can be issued if the first instruction (from an even address) will be routed to an even pipe unit and the second instruction to an odd pipe unit.

Table 2-2 Dual issue unit assignments

Inst. From addrress 0	Inst. From addrress 4
	
Simple fixed	Permute
Shift	Local store
Single precision	Channel
Floating Integer	Branch
Byte	

Operands are fetched either from the register file or forward network and sent to the execution pipelines. Each of the two pipelines can consume three 16 byte operands and produce a 16 byte result every cycle. The register file has six read ports, two write ports, 128 entries of 128 bits each and is accessed in two cycles. Results produced by functional units are held in the forward macro until they are committed and available from the register file. Loads and stores transfer 16 bytes of data between the register file and the local store.

Table 2-3 details the eight execution units. Simple fixed point [9], floating point [10] and load results are bypassed directly from the unit output to input operands to reduce result latency. Other results are sent to the forward macro where they distribute a cycle later. Figure 2-13 [7] is a pipeline diagram for the SPE that shows how flush and fetch are related to other instruction processing.

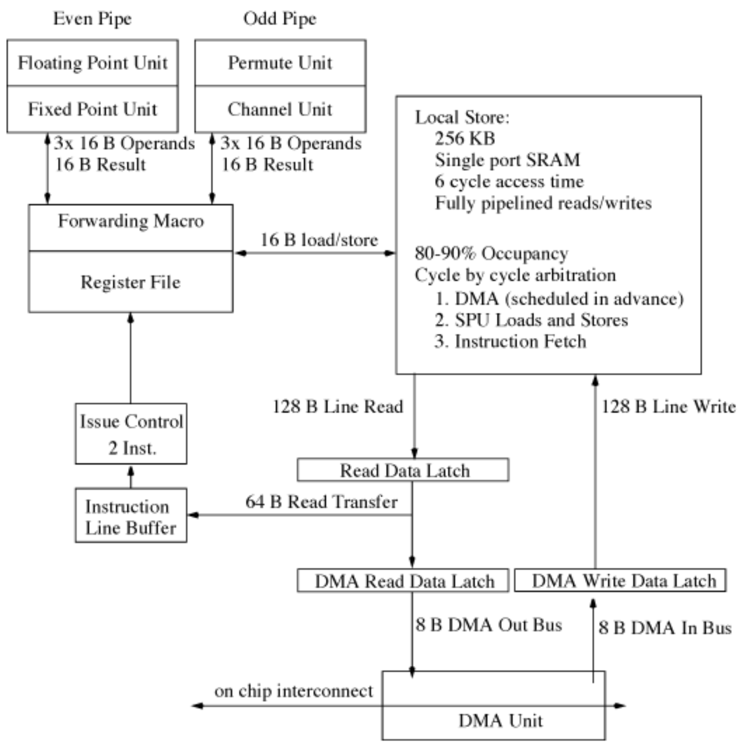


Figure 2-12 SPU organization

Table 2-3 Unit and instruction latency

Unit	Instruction	Instruction Latency
Simple Fixed	word arithmetic, logicals, count leading zeros, selects, and compares	2
Simple Fixed	word shifts and rotates	4
Single Precision	multiply-accumulate	6
Single Precision	integer multiply-accumulate	7
Bytes	pop count, absolute sum of differences, byte average, byte sum	4
Permute	Quadword shifts, rotates, gathers, shuffles as well as reciprocal estimate	4
Local Store	Load and strore	6
Channel	Channel Read/Write	6
Branch	Branches	4

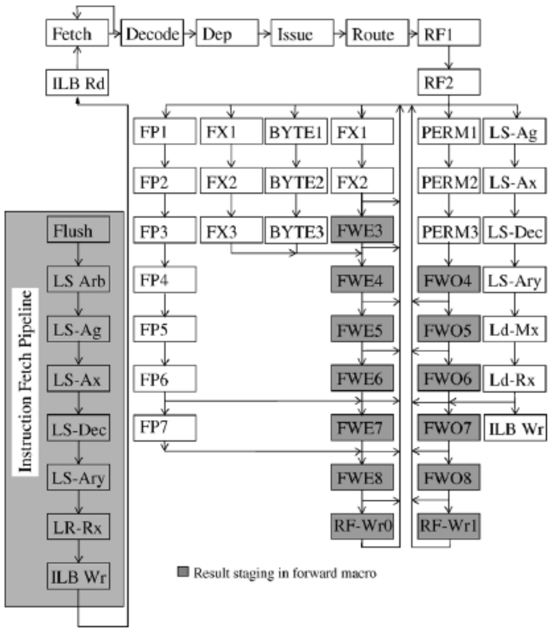


Figure 2-13 SPU pipeline diagram

3 DESIGN & OPTIMIZATION FLOW OF DEEP-PIPELINE LATENCY DATAPATH

Today’s multimedia applications need significant amounts of digital signal processing, so the current trend of many contemporary processors is generally designed for datapath-dominated recently. Cell processor provides a high performance for multimedia applications in the embedded system. One of the key features is the synergistic processing processor (SPU) which is data-oriented core for the requirement computing-intensive operations. In this chapter, we firstly introduce an overview of our proposed two-phased design flow: how to design the SPU datapath systematically. Chapter 3.1 presents the first-phased of design flow called spatial optimization including function modeling and cycle-accurate modeling, and then chapter 3.2 gives the second-phased of design flow, and chapter 3.3 illustrates the experimental results.

3.1 Spatial Optimization

Given the instruction set architecture (ISA) of synergistic processor unit (SPU), how to design the datapath of functional modeling systematically? At this sub-section, we detail the functional modeling and cycle-accurate design of our proposed two-phased design flow as shown in Figure 3-1.

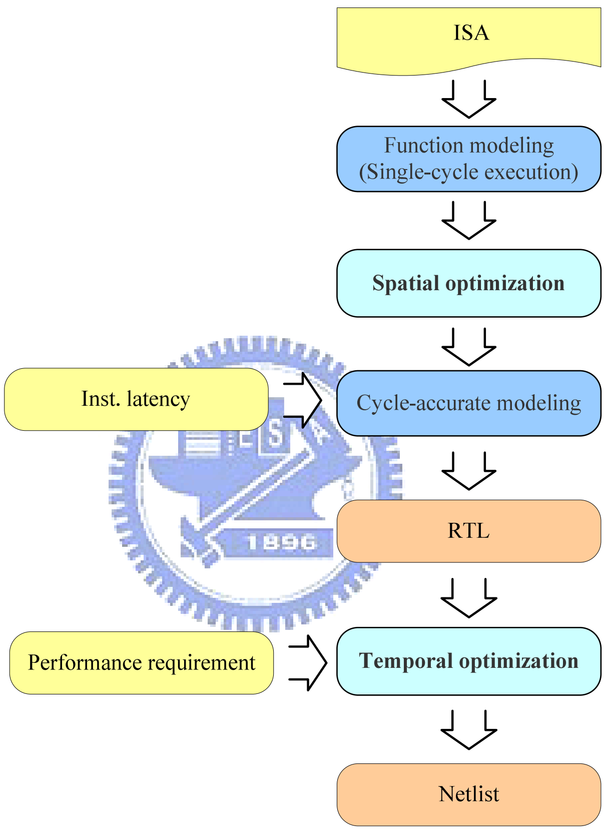


Figure 3-1 Overview of our proposed design flow

3.1.1 Function Modeling

The functional modeling of our proposed first-phased design flow can be divide four steps: instruction grouping, behavioral mode in RTL, synthesize (synthesized for time-optimized and area-optimized), and then the datapath called baseline of this step in order as shown in Figure 3-2.

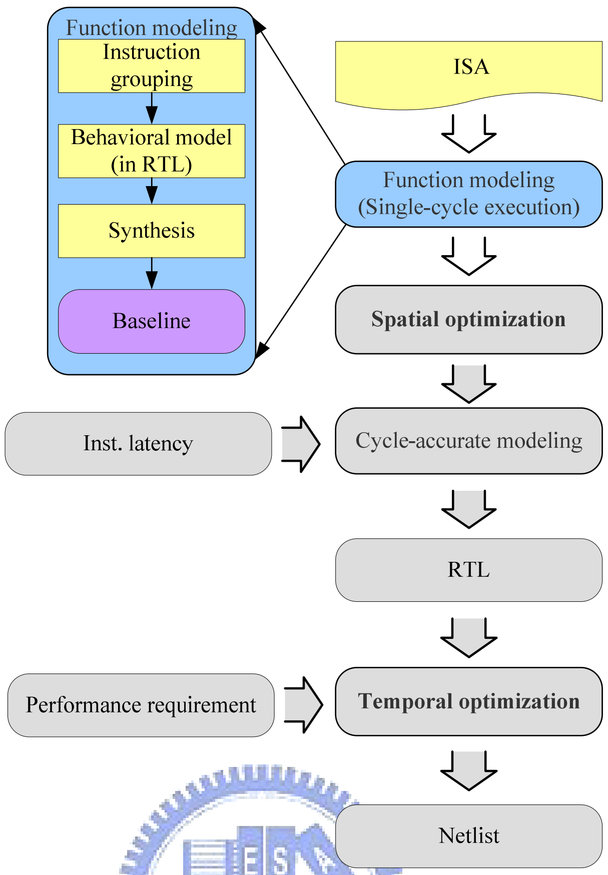


Figure 3-2 Functional modeling

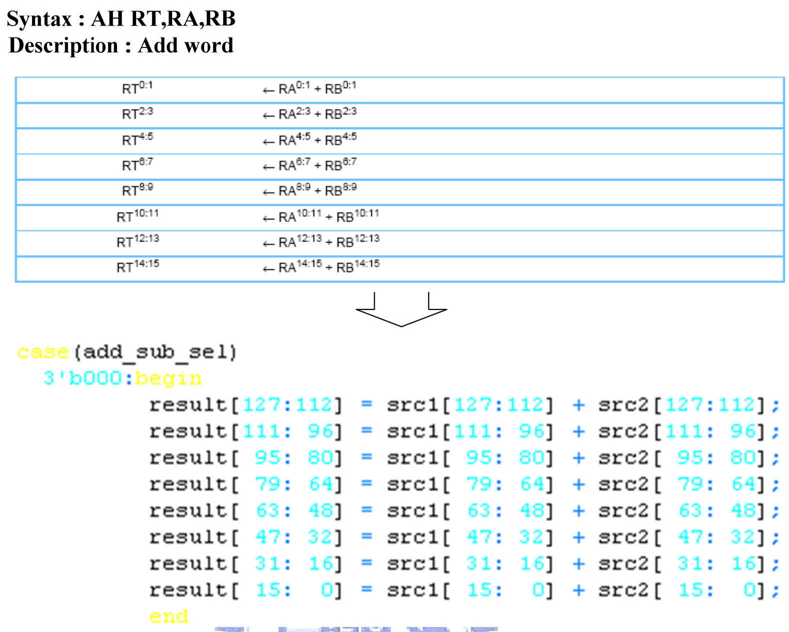
- Instruction grouping

To reduce the effort of SPU datapath design, we profile the common instruction sets used by multimedia applications, such as JPEG, FFT, DCT, FIR, and IIR through the SPU complier. The first step is to categorize these profiled instruction sets mainly by operations. We divide these instruction sets of datapath into seven group that are Add/Sub, Logic, Cmp (compare), Mask, S/R (shifter/rotator), Shuffle, Mpy (multiply) respectively.

- Behavioral model in RTL

After we categorize these instruction sets, we analyze the synthesis result of behavioral assignment in RTL followed the semantics of the SPU instruction sets architecture by CAD tool. This step intends to get the information of optimized-degree by Synopsys Design

Complier. We take the instruction set “ah” (add halfword) for example, as shown in Figure 3-3. The “add_sub_sel” of the Figure 3-3 is the control which instruction of the Add/Sub group. Other instructions can follow the code format like the description of Figure 3-3.



Syntax : AH RT,RA,RB
Description : Add word

$RT^{0:1}$	$\leftarrow RA^{0:1} + RB^{0:1}$
$RT^{2:3}$	$\leftarrow RA^{2:3} + RB^{2:3}$
$RT^{4:5}$	$\leftarrow RA^{4:5} + RB^{4:5}$
$RT^{6:7}$	$\leftarrow RA^{6:7} + RB^{6:7}$
$RT^{8:9}$	$\leftarrow RA^{8:9} + RB^{8:9}$
$RT^{10:11}$	$\leftarrow RA^{10:11} + RB^{10:11}$
$RT^{12:13}$	$\leftarrow RA^{12:13} + RB^{12:13}$
$RT^{14:15}$	$\leftarrow RA^{14:15} + RB^{14:15}$

```
case(add_sub_sel)
  3'b000:begin
         result[127:112] = src1[127:112] + src2[127:112];
         result[111: 96] = src1[111: 96] + src2[111: 96];
         result[ 95: 80] = src1[ 95: 80] + src2[ 95: 80];
         result[ 79: 64] = src1[ 79: 64] + src2[ 79: 64];
         result[ 63: 48] = src1[ 63: 48] + src2[ 63: 48];
         result[ 47: 32] = src1[ 47: 32] + src2[ 47: 32];
         result[ 31: 16] = src1[ 31: 16] + src2[ 31: 16];
         result[ 15:  0] = src1[ 15:  0] + src2[ 15:  0];
         end
```

Figure 3-3 Example of behavioral assignment in RTL

◆ Synthesis (synthesized for timing-optimized and area-optimized)

After finishing the above RTL-coding, we initially analyze the synthesis result of the seven functional units. This result is defined as baseline of synthesis result in this thesis. We can get the shortest delay of each functional unit through synthesized for timing-optimized. At the same, synthesized for area-optimized provides the information of hardware complexity. The information of area and timing is the mainly two topics that we’re most concern in datapath design. In Table 3-1, we can clearly indicate that both the largest area and the longest delay of baseline is the “Mpy” functional unit. By the way, we can see the synthesized result about the resource report, and find out the numbers of synthesized resource provided by DesignWare. Then we find that the CAD tool doesn’t do any optimization for our functional units in datapath. In next subsection, we will provide general strategy to optimize the datapath

Table 3-1 Synthesis result of baseline

Grouping		Baseline	
		Synthesis for timing	Synthesis for area
Add/Sub (#9)	delay(ns)	0.63	4
	area(um^2)	40468	25173
Logic (#9)	delay(ns)	0.45	3
	area(um^2)	14148	7209
Cmp (#18)	delay(ns)	0.62	2.5
	area(um^2)	15914	9977
Mask (#9)	delay(ns)	0.31	1.4
	area(um^2)	3063	1442
S/R (#26)	delay(ns)	0.8	4.8
	area(um^2)	225086	131332
Mpy (#11)	delay(ns)	**2.51**	7
	area(um^2)	518969	**371405**
Shuffle (#7)	delay(ns)	0.66	2
	area(um^2)	42123	26657

◆ Optimization (sharing)

The SPU instruction sets support 128-bit SIMD operations which are 8-bit, 16-bit, 32-bit, and 128-bit. For example, the "Add/Sub" functional unit supports the 16-bit and 32-bit addition and subtraction, and the "Cmp" functional unit even supports 8-bit (byte), 16-bit (halfword), and 32-bit(word) comparison. In order to support varieties of bit-length operations, we intuitively follow the behavioral assignment in RTL followed by the SPU ISA. From the last section, we find that there is no optimization for these functional units in datapath from these synthesized reports. This strategy is not area-efficient in order to transfer the SPU ISA to single-cycle execution datapath. In this step, we describe how to optimize these function units by using the general optimized strategy, such as resource sharing, sub-parallel method [11] for this seven functional units.

Resource sharing is the general method that the same bit-length computation of on functional unit uses the same hardware to compute with encoder that decides which instruction set to execute. We use this method in these functional units called "Logical", "Mask", "S/R", and "Mpy".

Sub-parallel method is that using multiple sub-word length hardware for word length computation. For a subword adder, this method is achieved by inserting multiplexers in the subword boundaries to propagate or prevent the subword carries in the carry chain [12]. For example, the “Add/Sub” functional unit is support 16-bit and 32-bit operations. We use two 16-bit adders to support 32-bit adder by adding an and-gate to control the 16-bit adder result’s carry as shown in Figure 3-4. If the word control bit is “1”, this hardware is to execute 32-bit addition or subtraction, and execute 16-bit operations vice versa. Other functional unit, such “Cmp”, can follow this method to do 8-bit, 16-bit, or 32-bit operations

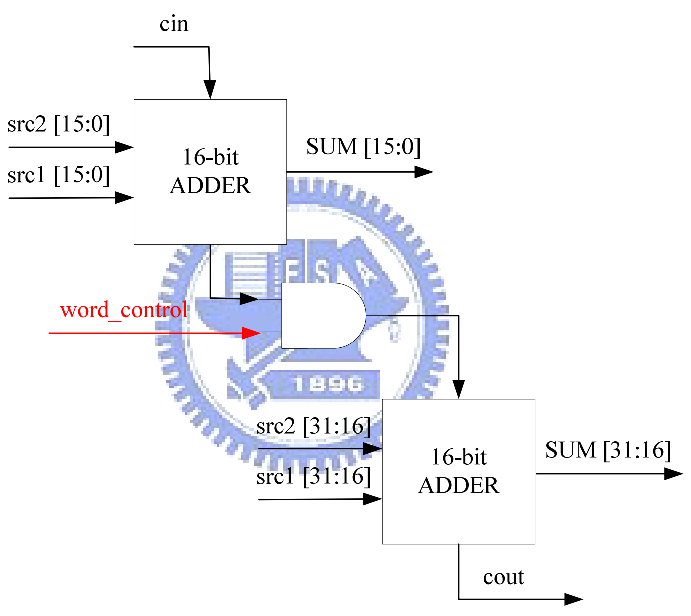


Figure 3-4 Add/Sub functional unit

We design the SPU datapath by using the above these optimized methods. In fact, these above optimization methods are the spatial optimization on the contrary to the temporal optimization in the 3.2 chapter.

3.1.2 Cycle-Accurate Modeling

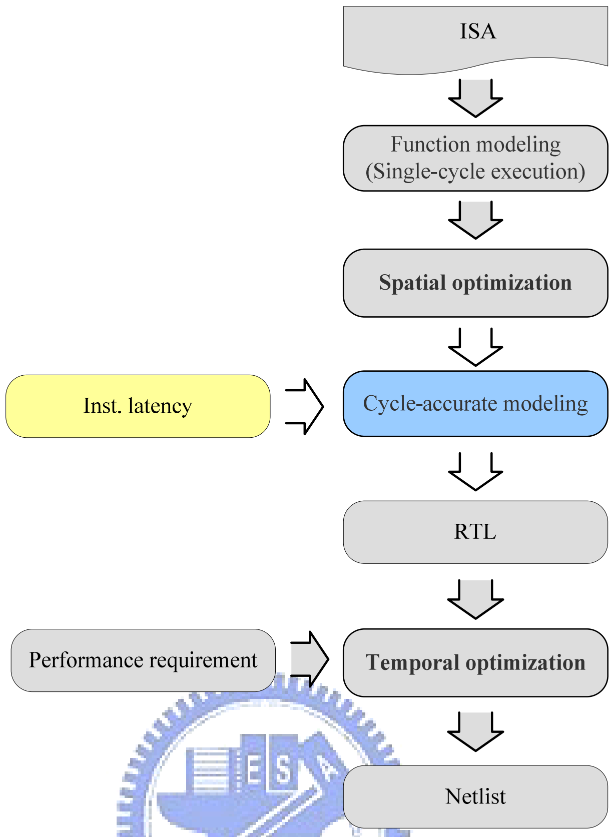


Figure 3-5 Cycle-accurate modeling

Because the front sub-section is just single-execution in datapath, we must take the latency spec. of the SPU ISA into consideration in datapath design as shown in Figure 3-5. In Table 3-2, it provides the instruction latency of all seven functional units. Instruction latency means that the number of clock cycles it takes for the instruction to get the available result through the pipeline. For example, the “Add/Sub” has two instruction latencies that means its result must be produced within two-cycle. At this step called “cycle-accurate modeling”, we combine the previous optimized single-execution with the instruction latency spec. of the SPU to design the micro-architecture. We use the main two methods, queue-sharing and the forwarding unit design. Queue-sharing means that these single-execution functional units bypass the same pipelined-register to meet the instruction latency, and the forwarding unit uses the above pipelined-register to forward the data to operand fetch unit. Next, we will introduce how to design the forward unit.

Table 3-2 Instruction latency

Grouping	# latency
Add/Sub	2
Logic	2
Cmp	2
Mask	2
S/R	4
Shuffle	4
Mpy	7

Data forwarding is a well-known technique to reduce the number of extra execution cycles. However, the complexity of forwarding network is rapidly increasing and usually constitutes the critical path [14]. In order to design forwarding unit systematically, we sort out pipelined-stage that producer (produce data) or consumer (consume data) and divide them into two categories. The analysis of the data forwarding paths includes two domains. One is temporal domain analysis and the other is spatial domain analysis. The temporal domain analysis checks the results produced by previous instructions but still in execution unit pipeline, while the spatial domain analysis checks all possible paths between every producer and consumer stages.

We defined the tolerable latency (TL) [14] of forwarding unit is the latency between data producing and data consuming:

TL (tolerable latency) = data consuming time – data producing time

The TL indicates the available latencies between consumer and producer. If the TL is less than the latency of forwarding unit, the data forwarding is impossible and has to stall several cycles till the TL is equal to the latency of forwarding unit. Figure 3-6 shows the example of TL in our SPU datapath design.

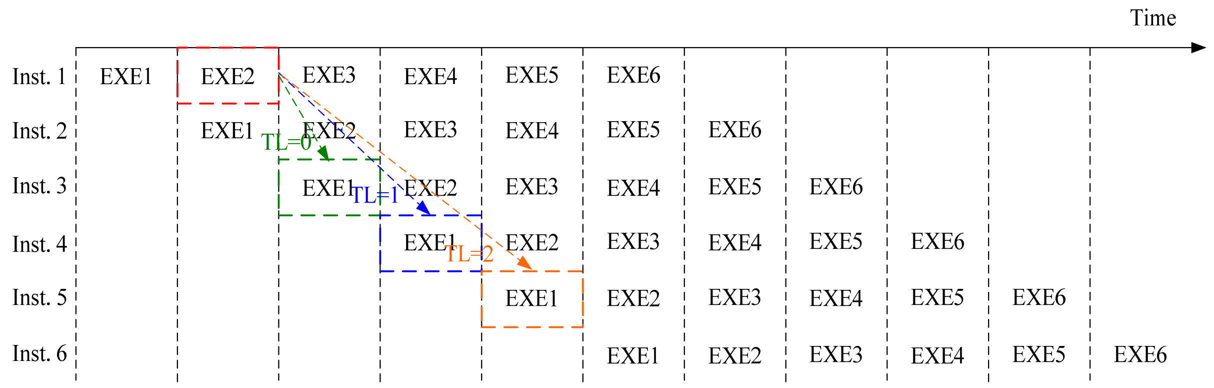


Figure 3-6 Tolerable latencies

For the forwarding unit which has one-cycle latency, there are three possible forwarding cases depending on the TL:

1. Non-causal (TL < 0).
2. Timing critical (TL = 0).
3. Normal (TL ≥ 1).

The first one is non-causal path. It happens that the consumer is executed earlier than the producer that results in a non-causal forwarding condition. The second one is the producer is directly forwarding to the consumer. That is, the data of producer is directly forwarded to consumer at next instruction cycle that means non-tolerable extra latency on the forwarding path. The final one is normal paths which have multi-cycle tolerable latencies between consumer and producers. In this case, the result produced by producer can’t be forwarded to consumer directly but has to queue for multiple cycles. In our datapath, we divide seven functional units into main three groups having the same instruction latencies, which have two-latency, four-latency, and seven-latency respectively. Table 3-3 shows all of our forwarding paths. In this table, all possible paths between each producer and consumer are categorized into three forwarding cases mentioned above. The instruction number indicates the instruction cycle latencies between consumer and producer.

Table 3-3 Forwarding table of our SPU

			Producer		
			2-latenty of FUs	**4**-latenty of FUs	**7**-latenty of FUs
Consumer	**2**-latenty of FUs	inst. 1	timing critical	non-causal	non-causal
		inst. 2	normal (TL=1)	non-causal	non-causal
		inst. 3	normal (TL=2)	timing critical	non-causal
		inst. 4	normal (TL=3)	normal (TL=1)	non-causal
		inst. 5	normal (TL=4)	normal (TL=2)	non-causal
		inst. 6	normal (TL=5)	normal (TL=3)	timing critical
	4-latenty of FUs	inst. 1	timing critical	non-causal	non-causal
		inst. 2	normal (TL=1)	non-causal	non-causal
		inst. 3	normal (TL=2)	timing critical	non-causal
		inst. 4	normal (TL=3)	normal (TL=1)	non-causal
		inst. 5	normal (TL=4)	normal (TL=2)	non-causal
		inst. 6	normal (TL=5)	normal (TL=3)	timing critical
	7-latenty of FUs	inst. 1	timing critical	non-causal	non-causal
		inst. 2	normal (TL=1)	non-causal	non-causal
		inst. 3	normal (TL=2)	timing critical	non-causal
		inst. 4	normal (TL=3)	normal (TL=1)	non-causal
		inst. 5	normal (TL=4)	normal (TL=2)	non-causal
		inst. 6	normal (TL=5)	normal (TL=3)	timing critical

The forwarding module can be imaged as a pipelined stage in the forwarding paths which isolates the complicated network from datapath by output registers. Once the forwarding table is established, we can design the forwarding micro-architecture as shown in Figure 3-7.

By using sharing queue and forwarding unit, we can design the micro-architecture of SPU. Because the SPU is dual-issue, we divide the datapath into two pipelined path. Figure 3-8 shows the pipelined diagram which meets the instruction latency of the SPU.

One of the fundamental decisions to be made in the design of a processor is the choice of the structure of the pipeline. In next chapter, we explore this issue to get an optimal area-efficient pipeline stage for each functional unit of SPU, given the instruction latency of ISA with a targeted performance requirement. This issue is treated both analytically and by simulation. We use the spatial optimization for our datapath design. At first, we have the preliminary analysis for the latency exploration. Finally, we use the mathematical formulation to get the optimal architecture.

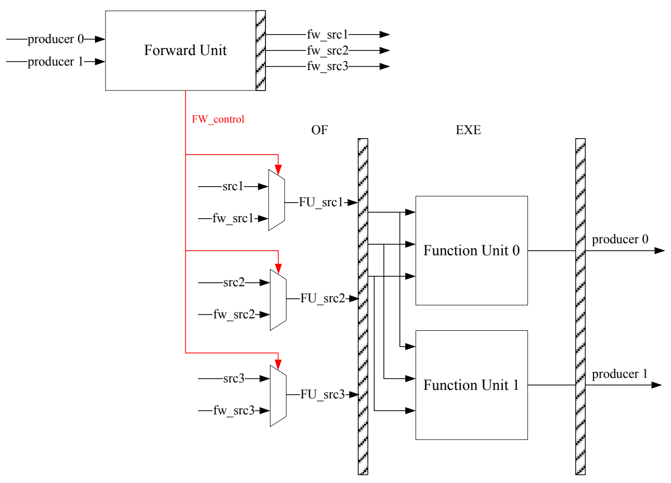


Figure 3-7 Forwarding network

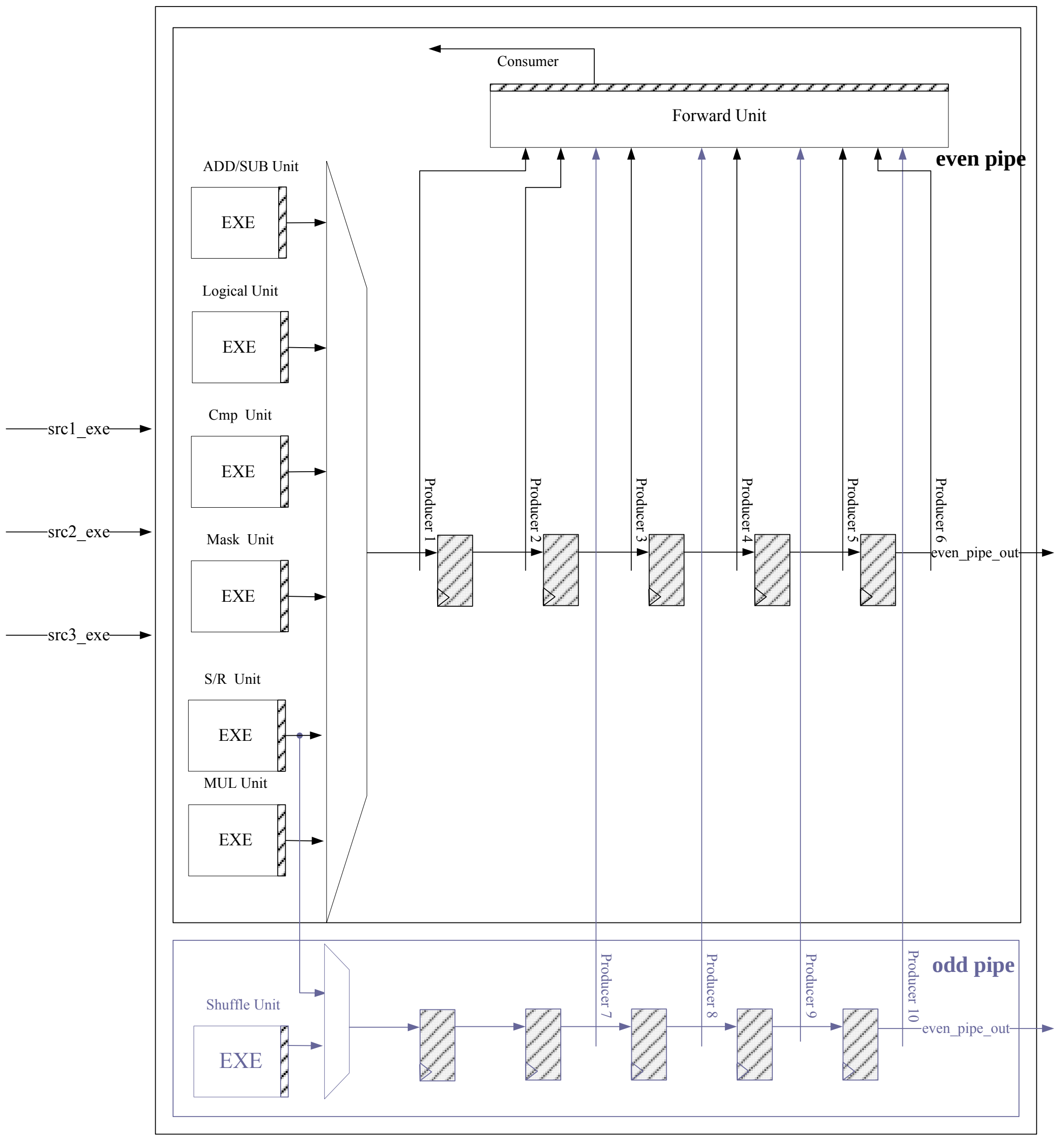


Figure 3-8 SPU Pipeline diagram

3.2 Temporal Optimization

As shown in Figure 3-8, the pipeline datapath just use bypassing-register to meet the instruction latency of functional units, but we can explore the latency spec. to optimize the datapath as shown in Figure 3-9. This is the question as to an optimum pipeline depth for a processor, given the latency spec. of ISA. Retiming is a structural optimization technique that relocates the registers in a logic circuit with the objective of minimizing their total gate counts, maximizing the circuit performance, or achieving both simultaneously [15][16]. We apply retiming to make functional units run at the required timing constraints containing a minimum number of registers.

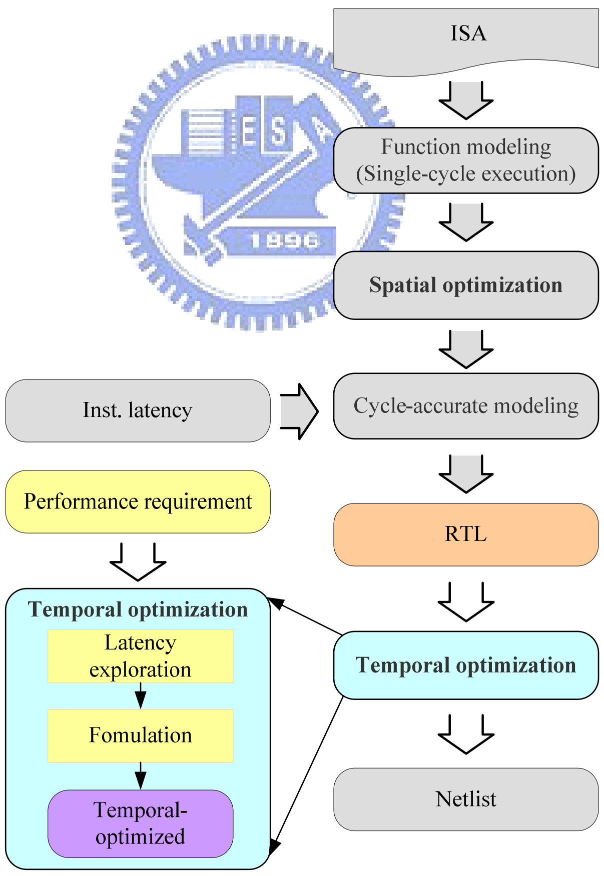


Figure 3-9 Temporal optimization

◆ Latency exploration

Retiming [15] is a transformation technique used to change the locations of delay element in a circuit without affecting the input/output characteristics of the circuit. It is a useful method for optimize the performance in synchronous circuit design. These include reducing the clock period of the circuit, reducing the number of registers in the circuit, reducing the power consumption of the circuit, and logic synthesis. In this thesis, we use the retiming to reduce the number of registers in out datapath.

In the following pipelined functional units, we use the CAD tool called pipeline_design of Synopsys Design Complier to pipeline the functional units. For example, we have three ways to decide the pipeline structure with the given latency of functional units as shown in Figure 3-10. The first way is that the functional unit is directly bypassing three pipelined register without pipelining the functional unit. The second way is that the functional unit is pipelined 2-stage by CAD tool and then bypassing two pipelined registers. The final way, we use CAD tool to pipeline the functional unit with 3-atage and bypassing the output register. We find that the trivial stage-selection, that is 3-stage, is not surely the best area-efficient with targeted frequency, so we will analyze the synthesized area trend of pipelined functional units to help formulate mathematical formulation.

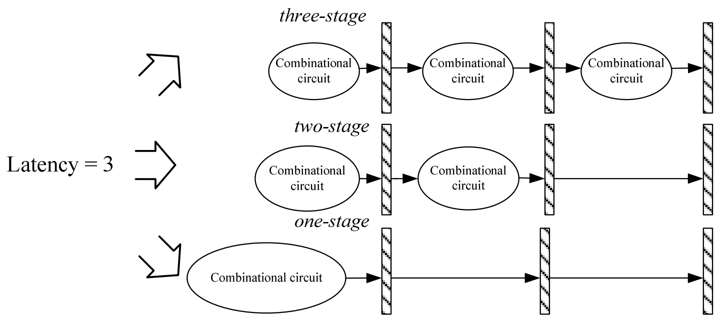


Figure 3-10 Function unit with 3-cycle latency

Next, we'll analyze the multiple-cycle latency basic module of functional units. That is, we will analyze these functional units, such as 16-bit “S/R”, 8-bit “Shuffle”, and 16-bit “MUL”

- ◆ Functional unit characterization

  - ■ S/R

The “S/R” functional unit has 3-latency, so there is three ways to decide the pipelined structure. We use the 16-bit shifter to estimate the area trend of all three ways by using the synthesized result of pipelined functional unit and estimate the 16-bit pipelined register under the 1.25ns timing constraints. As shown in both Table 3-4 and Figure 3-11, we can see that the first column is the possible stage number of “S/R” functional unit. At the same time, we try to estimate the pipelined register the third column and the 16-bit register is 288 um^2. For example, the first case is that the functional unit is directly bypassing three-stage pipelined-registers, so 288 multiplied by three is 864 um^2. We use this way to estimate the synthesized trend, and cleanly see the area is proportion to the stage number as shown in Figure 3-11.

Table 3-4 Piped S/R FU

Piped S/R FU

Piped Stage	Piped-FU	Bypassing Register	Area (um^2)
1	1514	864	2378
2	2237	576	2813
3	2528	288	2816

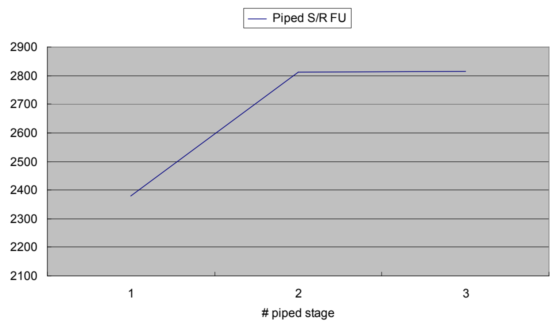


Figure 3-11 Piped S/R FU

■ Shuffle

The "Shuffle" is also a 3-latency functional unit, so it has three choice of pipelined-stage. We use 8-bit shuffle module to approximate the area trend of the three possible cases for pipelined stage under the 1.25ns timing constraints. In both Table 3-5 and Figure 3-12, we can find the area trend of pipelined "Shuffle" is almost proportion to the pipelined stage of functional unit. We can see the slight difference between 2-stage and 3-stage, but it will be more distinct from the multiple modules in our "Shuffle" unit.

Table 3-5 Piped Shuffle FU

Piped Shuffle FU

Piped Stage	Piped-FU	Bypassing Register	Area (um^2)
1	1433	432	1865
2	1580	288	1868
3	1724	144	1868

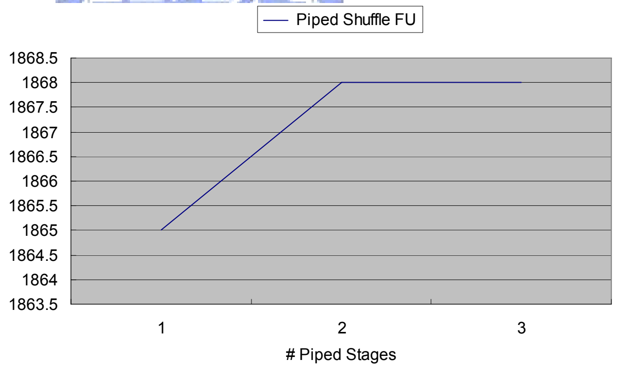


Figure 3-12 Piped Shuffle FU

■ MUL

The multiplier is the main critical functional unit, so multiplier typically has much more pipelined-stages than other functional units in order to target high frequency. It means that there is deeper pipeline in multiplier, so we have more design space to decide the pipelined-stages. Our "MUL" functional unit has 6-latency, so it has six possible selections of pipelined-stages. Here, we use 16-bit multiplier synthesized under 1.25ns timing constraints. In both Table 3-6 and Figure 3-13, we can cleanly see the trend of the area of pipelined-stage "MUL". Different from the previous functional units, there is a non-available synthesized result in the second row. Because the non-pipelined MUL's critical path is longer than 1.25 ns, it requires at least 2 pipelined-stages to target our operating frequency, 800MHz. In Figure 3-13, we can find there is a smooth curve between the 2-stage and 3-stage functional unit. At the same time, there is a steep curve between 3-stage and 4-stage functional unit. That's because the synthesized result is not absolutely linear growing up with the pipelined-stage functional unit. General speaking, the area of MUL is proportion to the pipelined-stage.

Table 3-6 Piped MUL FU

Piped MUL FU

Piped Stage	Piped-FU	Bypassing Register	Area (um^2)
1	NA	NA	NA
2	8521	2880	11401
3	9403	2304	11707
4	9981	1728	11709
5	11340	1152	12492
6	12140	576	12716

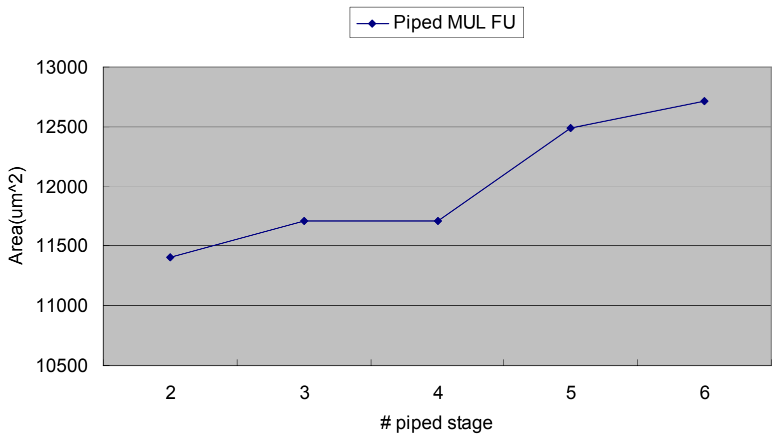


Figure 3-13 Piped MUL FU

In one word, we use smaller modules to estimate the area of functional unit with the above synthesized trend in reality. These results help us to formulate the following mathematical formulation. After using register estimation and pipeline functional unit to estimate the area trend, we'll introduce how to formulate the mathematical formulation for our datapath design.

◆ Formulation

Retiming is a structural optimization technique that relocates the registers in datapath that is targeted to minimize total gate count and maximize the circuit performance. We formulate the mathematical formulation with retiming method that is to minimize the area under timing constraint.

The mathematical formulation is as follows. We solve the equations to minimize the total area of all pipelined functional units under timing constraints. As shown in Table 3-8, the first parameter "i" gives the identified number to each functional unit. Table 3-7 shows the ID number of each functional unit. The second A_{pi} is the total area of each pipelined functional unit, P_i and L_i is respectively the pipelined-stage and latency spec. of each functional unit. Finally, we introduce the following timing parameter. The first parameter C_i is the control delay of each functional unit, means the timing delay of multiplex before function unit. The Means of t_i and t_p are timing delay of pipelined-register and pure combinational respectively. We estimate the parameter "t_p" about 0.15 ns to 0.2 ns from the manual of UMC 90 process. The least timing constraint is "t" that is our targeted operating frequency 800MHz (1.25ns).

Table 3-7 Number ID of functional unit

FU	ID number (i)
Add/Sub	1
Logic	2
Cmp	3
Mask	4
S/R	5
Shuffle	6
Mpy	7

Table 3-8 Description of equation's parameter

Parameter	Description
i	ith FU
A_i	area of ith FU
P_i	# pipelined stage
L_i	latency spec.
C_i	control delay
t_i	timing of non-piped FU
t_p	pipelined register
t	timing contraints

$$\text{Minimize } A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 \tag{3-1}$$

$$\begin{cases} P_1 \le L_1 \\ P_2 \le L_2 \\ P_3 \le L_3 \\ P_4 \le L_4 \\ P_5 \le L_5 \\ P_6 \le L_6 \\ P_7 \le L_7 \end{cases} \tag{3-2}$$

$$\begin{cases} \frac{t_1}{P_1} + C_1 + t_p < t \\ \frac{t_2}{P_2} + C_2 + t_p < t \\ \frac{t_3}{P_3} + C_3 + t_p < t \\ \frac{t_4}{P_4} + C_4 + t_p < t \\ \frac{t_5}{P_5} + C_5 + t_p < t \\ \frac{t_6}{P_6} + C_6 + t_p < t \\ \frac{t_7}{P_7} + C_7 + t_p < t \end{cases} \tag{3-3}$$

In Equation 3-1, we try to derive the minimum area of each one of the seven functional units. We formulate Equation 3-2 and Equation 3-3 with parameters. We list a simultaneous inequality by the mainly concepts, that is the total delay must be shorter than timing constraints (1.25ns) within one-stage pipelined datapath as shown in Figure 3-14. The parameter t_i can be derived from the column of synthesis for timing of every functional unit in Table 3-8. Equation 3-2 is the latency spec. that means combinational circuit of every functional unit can be divided into at most stage.

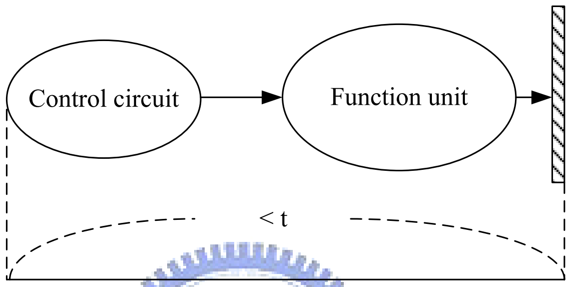


Figure 3-14 Timing delay of one-stage pipelined datapath

From these above constraints and Equation 3-2 and 3-3, we can derive the solution as the Equation 3-4.

$$\begin{cases} P_1 = 1 \\ P_2 = 1 \\ P_3 = 1 \\ P_4 = 1 \\ P_5 = 3 \\ P_6 = 1 \\ P_7 = 3 \end{cases} \tag{3-4}$$

3.3 Experimental Results

In this chapter, we show the experimental result of our proposed design flow. It includes spatial optimization and temporal optimization as shown in Figure 3-15.

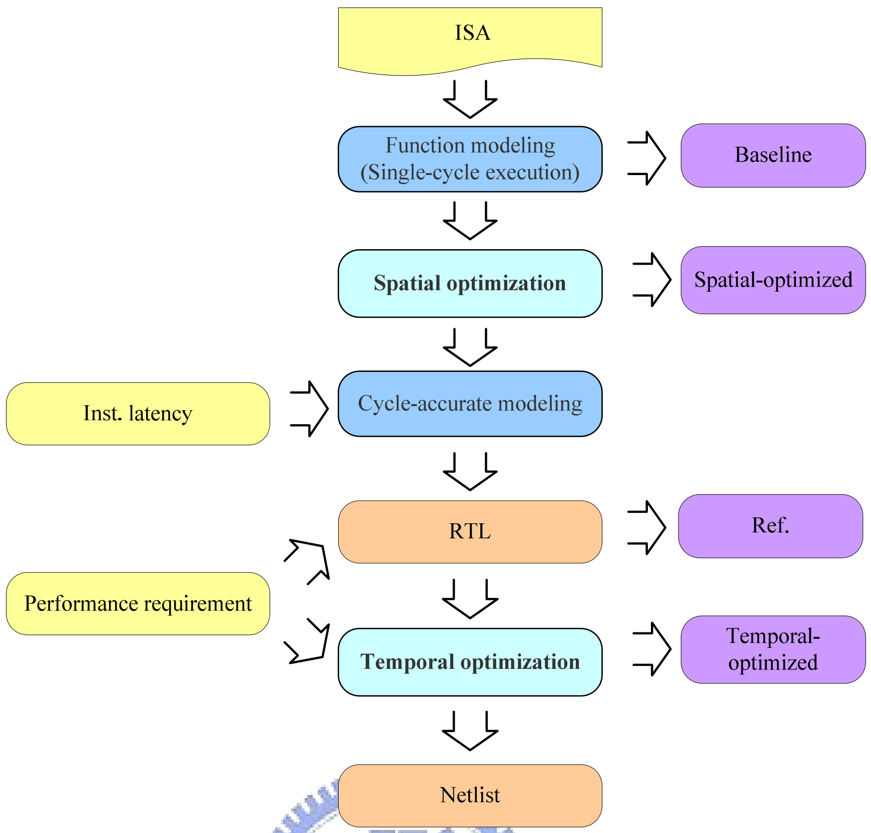


Figure 3-15 Proposed design flow

Spatial optimization

Table 3-9 shows the synthesized result by using the above optimized method that is defined as spatial-optimized. It shows the timing critical and the hardware complexity of each grouping. At the same time, we also set the timing constraint as 2.5ns (400MHz) as the typical case. By the way, we can derive the general case of common operating frequency in DSP.

Table 3-9 Synthesized result of baseline and spatial-optimized

Grouping		Synthesis for timing		Synthesis for area		Target 400 MHz	
		Baseline	Spatial-optimized	Baseline	Spatial-optimized	Baseline	Spatial-optimized
Add/Sub	delay(ns)	0.63	0.82	4	4.6	2.5	2.5
	area(um^2)	40468	8811	25173	5635	26183	9530
Logic	delay(ns)	0.45	0.45	3	3	2.5	2.5
	area(um^2)	14148	14282	7209	7209	7215	7215
Cmp	delay(ns)	0.62	0.72	2.5	2.8	2.5	2.5
	area(um^2)	15914	7040	9977	5515	9977	5547
Mask	delay(ns)	0.31	0.31	1.4	1.4	2.5	2.5
	area(um^2)	3063	3063	1442	1442	1443	1443
S/R	delay(ns)	0.8	1.4	4.8	9.3	2.5	2.5
	area(um^2)	225086	94460	131332	49693	132160	50341
Mpy	delay(ns)	2.51	2.01	7	7.6	2.5	2.5
	area(um^2)	518969	68055	371405	39317	NA	47741
Shuffle	delay(ns)	0.66	0.66	2	2	2.5	2.5
	area(um^2)	42123	30393	26657	24388	26657	24399

In Table 3-10, we can find the hardware complexity of spatial-optimization is much lower than that of baseline from the third column "Area-optimized". In order to sharing the hardware resource, we add some control hardware like encoder which adds timing delay slightly in datapath. In the "Synthesis for timing" column of Table 3-10, we can find the timing delay of most functional units is increasing slightly except the "Mpy" functional unit. Because the hardware complexity of the "Mpy" functional unit of baseline with larger encoder is much lager than spatial-optimized with smaller encoder, the timing delay of baseline is longer than spatial-optimization. By the way, the "Logic" and "Mask" unit are the same in both baseline and spatial-optimized because these units use simple logical gate or just wiring. If we add some controller to share the same logical gate, there is much overhead compared to the original controller in these two units. Finally, we use a typical case target to 400MHz (2.5ns) to confirm the above optimized method for area. We show the improvement of these functional units as shown in Table 3-11. In the "Mpy" unit of the 400MHz column, the "NA" means no available because its timing delay is longer than 2.5ns. In the next section, we'll take the latency spec. of the SPU into consideration for micro-operation design.

Table 3-10 Comparison between baseline and spatial-optimized

	Timing-optimized (ns)		Area-optimized (μm^2)		400MHz (μm^2)	
Grouping	Baseline	Spatial-optimized	Baseline	Spatial-optimized	Baseline	Spatial-optimized
Add/Sub	0.63	0.82	25173	5635	26183	9530
Logic	0.45	0.45	7209	7209	7215	7215
Cmp	0.62	0.72	9977	5515	9977	5547
Mask	0.31	0.31	1442	1442	1443	1443
S/R	0.8	1.4	131332	49693	132160	50341
Mpy	2.51	2.01	371405	39317	NA	47741
Shuffle	0.66	0.66	26657	24388	26657	24399

Table 3-11 Improvement by spatial-optimized

Grouping	Timing	Area	400MHz
Add/Sub	-30.2%	77.6%	63.6%
Logic	0.0%	0.0%	0.0%
Cmp	-16.1%	44.7%	44.4%
Mask	0.0%	0.0%	0.0%
S/R	-75.0%	62.2%	61.9%
Mpy	19.9%	89.4%	***NA***
Shuffle	0.0%	8.5%	8.5%

◆ Temporal optimization

We'll prove our proposed temporal optimization that is optimal micro-architecture. In fact, we can synthesize for all cases of the latency spec. of functional units. But, this method is too trivial to consume the design time in order to get the optimal micro-architecture with the deeper pipelined datapath. Our proposed temporal optimization not only saves the iterative time, but also gets the optimal selection of pipelined-stage as shown in Table 3-12.

Table 3-12 All cases for latency spec.

Grouping	Latency(#)		under timing constraint **1.25**					
			1	2	3	4	5	6
Add/Sub	1	delay(ns)	1.25	*NA*	*NA*	*NA*	*NA*	*NA*
		area(um^2)	9490	*NA*	*NA*	*NA*	*NA*	*NA*
Logic	1	delay(ns)	1.25	*NA*	*NA*	*NA*	*NA*	*NA*
		area(um^2)	10287	*NA*	*NA*	*NA*	*NA*	*NA*
Cmp	1	delay(ns)	1.25	*NA*	*NA*	*NA*	*NA*	*NA*
		area(um^2)	7690	*NA*	*NA*	*NA*	*NA*	*NA*
Mask	1	delay(ns)	1.25	*NA*	*NA*	*NA*	*NA*	*NA*
		area(um^2)	3710	*NA*	*NA*	*NA*	*NA*	*NA*
S/R	3	delay(ns)	1.25	1.25	1.25	*NA*	*NA*	*NA*
		area(um^2)	NA	NA	78639	*NA*	*NA*	*NA*
Shuffle	3	delay(ns)	1.25	1.25	1.25	*NA*	*NA*	*NA*
		area(um^2)	31283	32653	34888	*NA*	*NA*	*NA*
Mpy	6	delay(ns)	1.25	1.25	1.25	1.25	1.25	1.25
		area(um^2)	NA	NA	81782	83195	83554	85655

In Table 3-12, the blue-color word is derived form our proposed temporal optimization. For each functional unit, the blue-word selection is the best choice for the optimal micro-architecture. Maybe we can try to synthesize for all case, but it is time-consuming. For example, the Mpy functional unit has six latencies, that's mean that it have six possible pipelined structure. If we try to synthesize for all cases, it is not timing-efficient. This situation is becoming serious for functional unit with more and more latencies of deeper function unit in order to target high operation frequency. Finally, we show the improvement compared to trivial approach with our proposed temporal optimization as shown in Table 3-13.

Table 3-13 shows the comparison of all pipelined functional units between our proposed temporal optimization and the trivial approach with the pipeline diagram of the reference paper [7]. The Reference version uses the spatial-optimized datapath to explore the latency directly by the pipelined-diagram of reference, and the temporal-optimized uses the same datapath by temporal optimization. The first column is all functional units, and the second is the latency spec. that means the maximum number pipelined-stage of each functional unit. We can see that the reference is directly used the latency spec. to pipeline each functional unit by

CAD tool. However, the spatial optimization uses our proposed temporal optimization. Seeing the “Improvement” column, we can see improvement by 0% from Add/Sub to S/R functional units, because these functional units have no latency to explore. In other words, these functional units have only one-latency and the S/R must pipeline 3 stages into it in order to target high frequency. In both “Shuffle” and “Mpy”, we improve the area compared to the version 3 by 10% and 4.5% respectively.

Table 3-13 Temporal optimization

FU	latency(#) (800MHz)		Temporal-optimized	Ref.	Improvement (%)
Add/Sub	1	pipelined-stage	1	1	0%
		area	9490	9490	
Logic	1	pipelined-stage	1	1	0%
		area	10033	10033	
Cmp	1	pipelined-stage	1	1	0%
		area	7690	7690	
Mask	1	pipelined-stage	1	1	0%
		area	3710	3710	
S/R	3	pipelined-stage	3	3	0%
		area	78639	78639	
Shuffle	3	pipelined-stage	1	3	10.33%
		area	31283	34888	
Mpy	6	pipelined-stage	3	6	3.14%
		area	82964	85655	

Finally, we show the area-efficient micro-architecture target to lightweight applications target to 100MHz to 800MHz as shown in Table 3-14. Our proposed design flow improves the area of micro-architecture by approximate 20%. The case “800 MHz” is just improved by 3% because its optimization space is limited by timing-optimized. General speaking, we can see the trend of micro-architecture is area-efficient by using our proposed design flow. In Table 3-14 and Figure 3-16, we can see the other case is area-efficient micro-architecture improved 15% to 20% of area.

Table 3-14 Area reduction from temporal optimization

Freq.	800	700	600	500	400	300	200	100
Ref	231180	227291	222731	220023	211618	205309	204331	200748
Proposed	224884	192097	177805	174614	169417	160588	157005	154830
Improvement	2.72%	15.48%	20.17%	20.64%	19.94%	21.78%	23.16%	22.87%

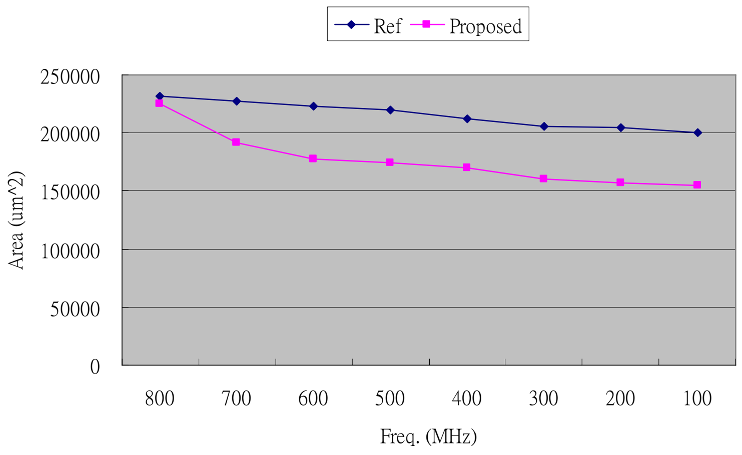


Figure 3-16 Improvement by our proposed design flow

4 SILICON IMPLEMENTATION

In this chapter, the silicon implementation is to implement the design with cell-based flow, and the result shows the area and timing after physical implementation. The result of silicon implementation contains two parts. The first one is the implementation flow and the second one is the implementation result of SPU and layout of SPU chip.

4.1 Implementation Design Flow

We will design the SPU processor based on UMC 90nm 1P9M Process Low-K. Figure 4-1 is a flow chart which illustrates the design flow for our SPU design.

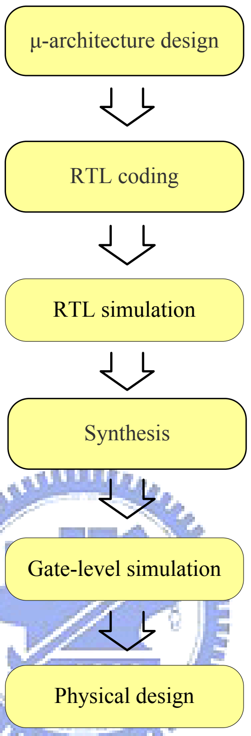


Figure 4-1 Implementation flow

Figure 4-2 illustrates the I/O interface. The SPU has 32KB on-chip instruction memory and 64KB on-chip data memory. The datapath is dual-issue as shown in Figure 4-3. By our proposed design flow of last chapter, we can decide systematically the pipelined stage of every functional unit to design area-efficient micro-architecture. By the way, the S/R and Mul are pipelined into two stages respectively to meet the timing constraints. Due to the critical path determined by the memory modules provided in cell library, we set 400 MHz (2.5ns) as the operating frequency of our SPU core. We can see that the LS pipe is pipelined into 5 stages because these stages are to do data-gather due to the 32-bit output of data memory. In order to gather the data, we need four cycles to do that. After getting the available result of each functional unit, we use shared by-passing register to pass the result of every functional unit.

According the micro-architecture proposed from the last chapter, it defines pipeline stage to facilitate the RTL design. And then the forwarding path will take into consideration to avoid redundant routing paths. On the basis of pipeline stage and the I/O definition, we define the micro-operation of all instructions in every pipeline stage. By doing this work, hardware resources will be further defined. After these above analysis and design, the RTL (register transfer level) model can be built up. We execute behavioral-level simulation in the RTL model by using NC-Verilog simulator. After the RTL code is bug free, we use synthesis tool (Synopsys design complier) to synthesis our RTL code into gate-level netlist. The gate-level simulation will be performed to sure the logic gates work correctly. When the gate-level netlist is ready, we can use Cadence SoC encounter to implement physical design.

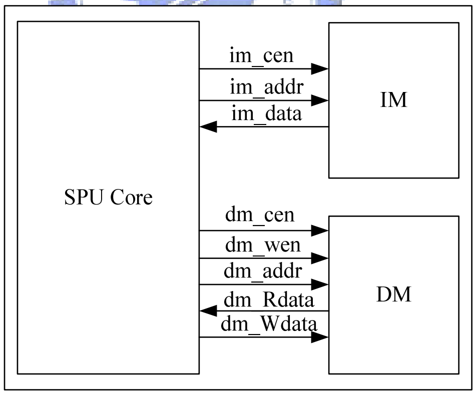


Figure 4-2 Our SPU interface

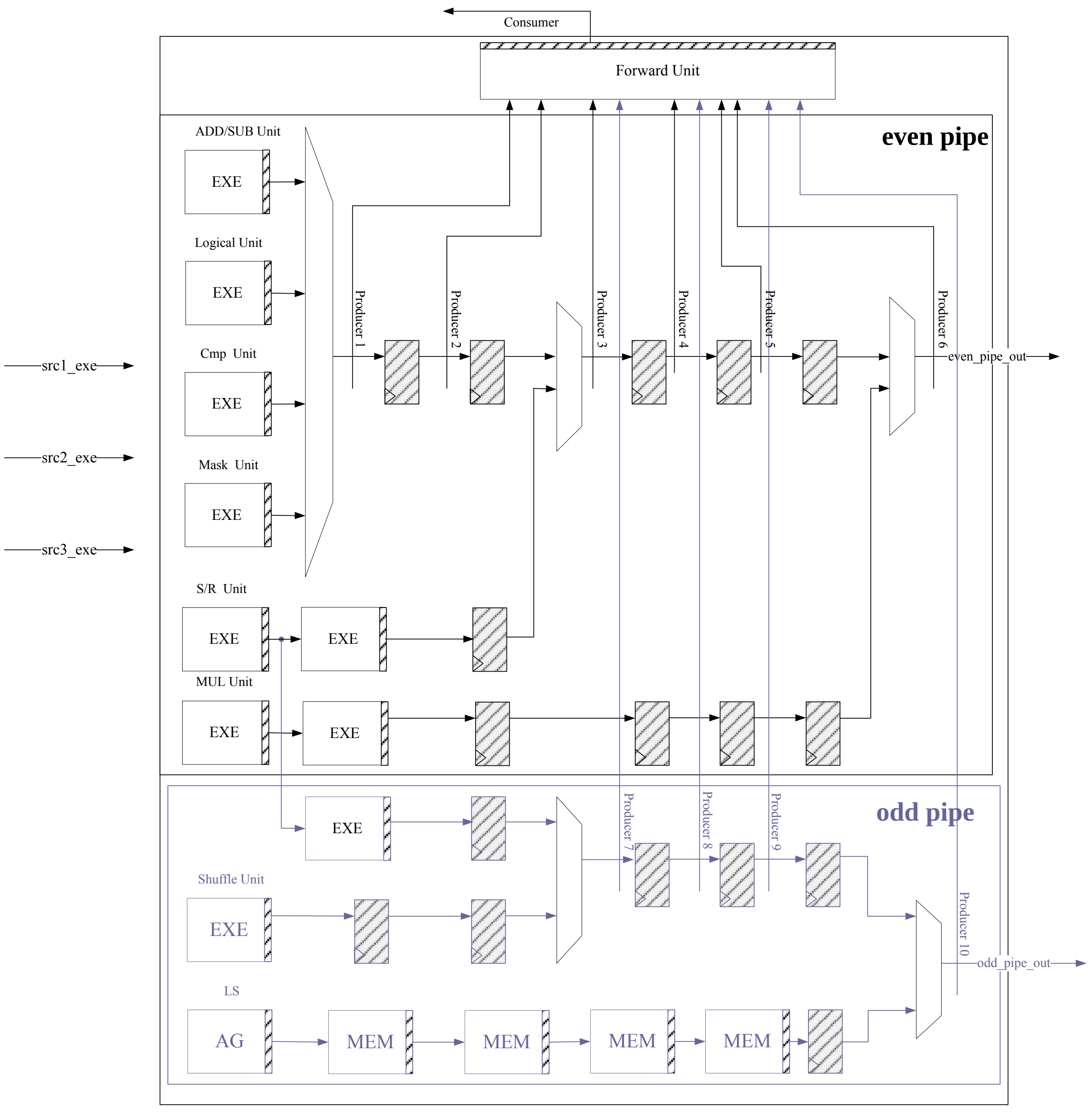


Figure 4-3 Pipeline diagram of our SPU

4.2 Implementation Result

In Table 4-1 , the synthesis result which uses UMC 90nm 1P9M Process Low-K and operates in worst case shows the area and timing. Figure 4-4 shows the layout of our SPU_CHIP. This processor is pipelined into 10 stages (4 instruction pipeline and 6 execution pipeline). According to simulation and APR result, SPUCHIP can operate at 400MHz and core size is 2.5mm x 2.5mm shows the summary of APR results.

Table 4-1 Synthesis result

Technology	UMC 90nm 1P9M Process Low-K
Total area	927,326
Die size	2.5mm x 2.5mm
Operating freq.	400 MHz
Power	320 mW

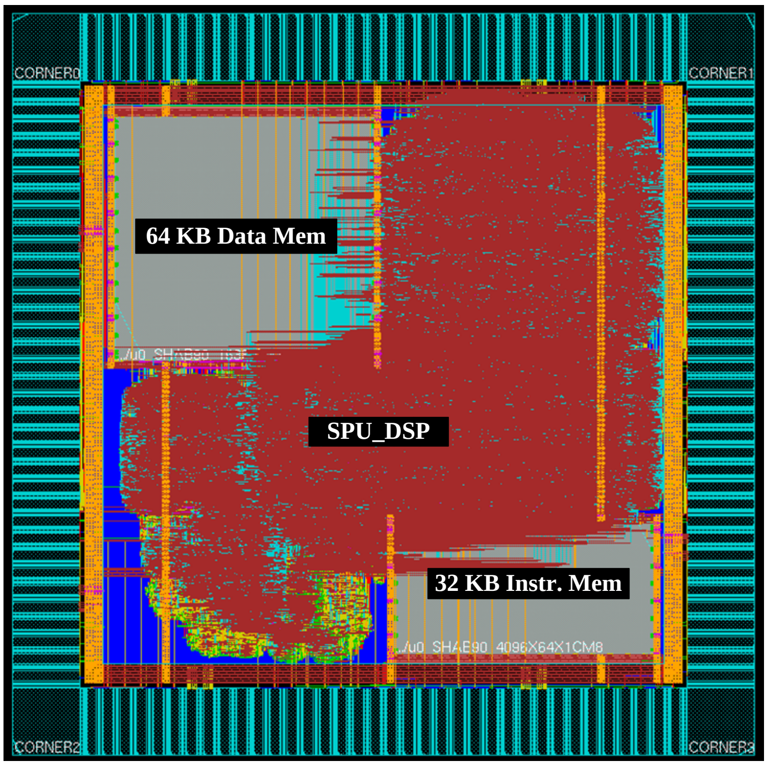


Figure 4-4 Implementation result

5 CONCLUSION & FUTURE WORKS

In this thesis, we proposed two-phased design flow to explore the optimization space with respect to the specific ISA implementation for embedded applications. The proposed two-phased design flow is to get the optimal micro-architecture under the various timing constraints with the software support. First-phased design flow is spatial-optimized includes function modeling and cycle-accurate modeling. In one word, the first-phased design flow is mainly spatial optimization. Second-phased design flow is temporal optimization to explore the latency by building mathematical formulation. Using formulating the mathematical formulation by our proposed temporal optimization, we can get the area-efficient micro-architecture systematically. We take the Cell SPU as our design example because the Cell SPU is data-oriented processor that exposes long latency. Our proposed design flow is more area-efficient than the ad-hoc method to design micro-architecture. The experimental result shows that our proposed design flow is more area-efficient 15% to 20% than the micro-architecture of reference [7] directly under timing constraints 100 MHz to 800 MHz.

Besides, our proposed design flow can be applied to the synthesis of ASIPs (Application

Specific Instruction set Processor), such as an automatic processor for optimal micro-architecture under the targeted timing constraints with the existing software support. For example, within the last year commercial tools like LISATek [18] framework came up, that allows to designing ASIP architectures by using their own description language. It shortens the design time dramatically compared to classical register-transfer-level (RTL) based approaches. For the micro-architecture of the specific ISA, the processor generator can be used to get the area-efficient micro-architecture under the target timing constraints with the existing software support when the RTL model of the processor is ready.

REFERENCES

[1] Y. H. Hu, *Programmable Digital Signal Processors – Architecture, Programming, and Applications*, Marcel Dekker Inc., 2002

[2] R.B. Lee, “Multimedia extensions for general-purpose processors,” in *Proc. IEEE Workshop Signal Processing Systems*, pp. 9-23, Nov.1997.

[3] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales, “AltiVec extension to PowerPC accelerates media processing,” *IEEE Micro*, vol. 20, no. 2, pp. 85-95, Mar./Apr. 2000.

[4] J.A Kahle et al., “Introduction to the Cell multiprocessor,” IBM J. Research and Development, vol. 49, no. 4/5, July 2005, pp.589-604

[5] The Cell architecture. [Online]. Available: http://domino.watson.ibm.com/comm/research.nsf/pages/r.arch.innovation.html

[6] *Cell Broadband Engine Programming Handbook version 1.1*, IBM, 2007

[7] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, et. al., "The microarchitecture of the synergistic processor for a Cell processor," *IEEE J. Solid State Circuits* 41, No. 1, 63-70 (2006).

[8] Synergistic Processor Unit Instruction Set Architecture, Version 1.2, IBM Corporation, Sony Computer Entertainment Corporation, and Toshiba Corporation. [Online]. http://www.ibm.com/chips/techlib/techlib.nsf/techdecs/ 76CA6C7304210F3987257060006F2C44/$file/ SPU_ISA_v1.2_27Jan2007_pub.pdf.

[9] J. Leenstra et al., “The vector fixed point unit of the streaming processor of a CELL processor,” presented at the *Symp. VLSI Circuits*, Kyoto, Japan, 2005.

[10] H. Oh et al., “A fully-pipelined single-pipelined single-precision floating point unit in the streaming processing unit of a CELL processor,” presented at the *Symp. VLSI Circuits*, Kyoto, Japan, 2005.

[11] S. Krithivasan and M.J. Schutle, “Multiplier Architecture for Media Processing,” in *Proc. 37th Asilomar Conf. Signals, Systems, and Computers*, pp. 2193-2197, Nov. 2003

[12] Suzuki, K. et al.,"A 2000-MOPS embedded RISC processor with a Rambus DRAM controller," *IEEE J. Solid-State Circuit*, vol. 34, pp. 1010-1021, 1999

[13] A. Terechko, M. Garg, and H. Corporaal, "Evaluation of speed and area of clustered VLIW Processors," *in Proc. VLSID*, pp.557-563, 2005

[14] P.C. Hsiao, T. J. Lin, C. W. Liu, and C. W. Jen, "Efficient datapath design for clustered &pipelined digital signal processors," *in Proc. VLSI design/CAD*, Aug. 2005

[15] C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conference* on VLSI, pp. 87-116, 1983

[16] C. Leiserson, F. Rose, and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol.6, pp. 5-35, 1911

[17] K. K. Parhi, *VLSI Digital Signal Processing Systems – Design and Implementation*, John Wiley & Sons, 1999

[18] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publishers, 2002

作者簡歷

呂進德，1983 年 9 月 13 日出生於台南縣。2006 年取得國立中央大學電機工程學系學士學位，並在國立交通大學電子工程研究所攻讀碩士。2008 年在劉志尉教授指導下，取得碩士學位。本篇論文「長管線延遲資料路徑之高面積效率設計與實現」為其碩士論文。