國立交通大學

資訊科學與工程研究所

碩 士 論 文

可被窄寬度運算共用之單一管線資料路徑設計

A Single Pipeline Datapath Design for Joinable Narrow-operand Operations

研 究 生：林聖勳

指導教授：鍾崇斌　教授

中 華 民 國　九 十 五　年 八 月

可被窄寬度運算共用之單一管線資料路徑設計

A Single Pipeline Datapath Design for Joinable Narrow-operand Operations

研 究 生：林聖勳　　　Student：Sheng-Hsun Lin

指導教授：鍾崇斌　　　Advisor：Chung-Ping Chung

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年八月

可被窄寬度運算共用之單一管線資料路徑設計

學生：林聖勳　　　　　　　　　　　　　指導教授：鍾崇斌 博士

國立交通大學 資訊科學與工程研究所 碩士班

摘要

現今大多數之處理器之架構都採 32 位元或者更高之位元數. 然而整數運算大多數時間都不會用到資料路徑中完整的寬度. 若將 Source Operand Bus 及 ALU 分割為多組 block, 則此資料路徑便具備同時處理一道以上運算之能力.

本論文提出讓單一資料路徑被兩道具備窄寬度特性之指令合用之設計. 讓其中一道運算之 Operand Block 以反轉(Turnaround)之次序, 來達到極有效率之資料路徑合用機制. 相較於傳統以直接 Shift 方式, 本設計不論就面積與電路延遲上, 都有較佳之表現. 此外, 本論文亦提出一縮短 ALU 因分割為多組 ALU Block 所造成延遲之設計. 最後, 本論文提出將此機制整合至一典型五階 MIPS 管線之方式.

A Single Pipeline Datapath Design for Joinable Narrow-operand Operations

Student: Sheng-Hsun Lin Advisor: Dr. Chung-Ping Chung

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Most general-purpose processors and embedded processors have 32-bit word widths or wider. However, integer operations rarely need the full 32-bit dynamic range of the datapath. If we partition the operand bus, result bus, and ALU into several blocks, the datapath could perform more than one operation in parallel.

In this thesis, mechanisms to join two narrow-operand operations together to share a single datapath are proposed. We proposed one novel ALU-sharing scheme by turning around operation-block ordering. Efficient designs to merge operands to share buses and ALU based on the technique are proposed and discussed. Compared with traditional "shift" approach, the turnaround approach has many advantages on area and delay. Besides, a technique to mitigate the delay overhead of the partitioned ALU by swapping operands is proposed. We also made performance simulation to help decide how to partition the datapath. Finally, how to integrate such datapath into a MIPS five-stage pipeline and required modifications are discussed.

誌謝

首先我要感謝我的指導教授 鍾崇斌老師。在學業上老師給了我非常嚴謹的指導及許多寶貴的建議，讓我在觀察、思考與解決問題的能力上都成長了許多，並且得以順利完成碩士論文；老師除了在學業上的指導之外，在生活態度上也給了我很大的啟發。我也要感謝實驗室的另一位指導老師 單智君老師，除了擔任我的口試委員，在小組討論的時候也給了我非常多的幫助與建議。還要感謝我的口試委員 邱日清老師與 邱舉明老師，在口試時給了我許多客觀的建議，並指出論文中不足之處，使本論文的層面更加完整。

我也要感謝實驗室的所有學長、同學們在我進行碩士論文研究期間給我的協助和鼓勵，讓我能得以克服遭遇到的種種困難，在士氣低落時給我的鼓舞。特別感謝喬偉豪學長、吳奕緯學長，Low Power 組的葉文涵、黃富群、莊富源、田濱華同學，以及黃士嘉同學及汪威定學弟。

最後，要感謝我的家人以及百潔給我的支持，才讓我可以全心全力地將碩士論文完成。

林聖勳
2006 年八月

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Most general-purpose processors and embedded processors have 32-bit word widths or wider. However, integer operations rarely need the full 32-bit dynamic range of the datapath. In other words, part of the datapath, including part of ALU and bus lines, is occupied by redundant bits which are the duplicates of the sign-bit of the values. With appropriate partitioning, those parts of datapath could be used by another operation.

1.1. Narrow-Operand ALU Operations

Bits of a value can be classified into the following two parts:

i. Significant part

ii. Insignificant part

Bits which are sufficient to represent the magnitude and its sign (or leading bits which are the same with its most significant bit) are called the significant part of the value, while the rest of bits in the value are called the insignificant part.

When performing an operation, bus lines and part of the ALU occupied with higher order data bits which belong to the insignificant part of both operands and result might always represent the same bit-value with the sign-bit of the significant part. In other words, part of the ALU generates result bits identical to the sign-bit of the significant part on the result bus lines.

Assuming an ALU is partitioned into several blocks, an ALU operation whose operands and results all have narrower significant part so that at least one ALU block is not necessary for calculating its correct result[1] is called a narrow-operand ALU operation.

With appropriate modifications on circuits, including partitioning the ALU and buses and additional checking mechanism, another narrow-operand operation could be joined into the datapath to exploit the part of datapath which is originally used to carry and calculate result for insignificant part of an operation.

1.2. Significant Bit-width of an ALU Operation

So how many bits should be reserved for one ALU operation? The required bit-width of an operation determines how many bus lines and ALU blocks should be allocated. Before execution of an operation, we can obtain the significant bit-width of all its operands. From the relationship between the significant bit-widths of operands and results, we classify ALU operations into two major types:

i. Operations that *won't increase* the number of significant bits

Bit-wise Logic Operation:

Since one bit representing for the leading bit value is reserved, the result of a bit-wise logic operation could certainly be "sign-extended" from the bit position that we reserved to represent its most significant bit.

$$\begin{array}{rr} & 00000110 \\ \overline{\oplus} & 00000010 \\ \hline & 11111011 \end{array}$$

Figure 1-1 a 8-bit XNOR operation

[1] "Correct" means we can get its full-width result by sign-extending the "narrower" result.

ii. Operations that *might increase* the number of significant bits

Addition:

Significant bit-width of the result of an addition operation could be more than the maximum bit-widths of its operands due to carry-in.

$$\begin{array}{rr} & 00001100 \\ + & 00001001 \\ \hline & 00010101 \end{array}$$

Figure 1-2 A 8-bit add operation

Subtraction:

The same situation of increased significant bit-width of the result would occur under subtraction.

$$\begin{array}{rr} & 00000001 \\ - & 11111101 \\ \hline & 00000100 \end{array}$$

Figure 1-3 A 8-bit subtraction operation

Shift / Rotation:

To estimate the significant bit-width of the result for a shift operation, not only the significant bit-width of its operand is required but also the direction it shifts to and its shift amount. Besides, to partition a shifter into two or more parts to shift for different amounts is more complex than partitioning an ALU into ALU blocks. The shift operation is not discussed in this thesis.

Multiplication:

The significant bit-width of the result from a multiplication operation can be decided from its multiplicand and multiplicator. But multiplication is not very frequently used and to partition the multiplier is also more complex than partitioning the ALU, how to share a multiplier is not discussed in this thesis.

We focus on the mechanism to efficiently share the buses and ALU only. The ALU in our design supports addition, subtraction, and bit-wise logic operations. So how do we decide the number of bits which should be reserved for an ALU operation? From the two classifications above, for simplicity, *the required bit-width of an ALU operation is equal to the larger one of the significant bit-widths of operands plus one bit*. The reserved bit is for the consideration of addition and subtraction. Besides, whether the addition or subtraction operation is signed or unsigned, the full-width result could be correctly sign-extended.

1.3. Organization of this Thesis

In Chapter 2, we described a related work about datapath-sharing and background for flexible ALU-sharing. Our motivation and objective are. In Chapter 3, designs about sharing the datapath are proposed. Chapter 4 shows the experiments and simulation results. A final proposal about designing the ALU is suggested. In Chapter 5, we summarized our conclusions.

Chapter 2. Background and Related Work

2.1. Distribution of Significant Bit-widths of ALU Operations

Before we start to propose possible designs, we first perform some profiling on dynamic instruction behaviors. Figure 2-1 shows statistics in dynamic instruction-type distributions. The raw data is retrieved from our simulation environment which is mentioned in chapter 4. Data-processing instructions which require only ALU in execution stage fall into the category of ALU, while those require the shifter are classified into Shift category. Data-transferring instructions, such as Load and Store, belong to the category Memory. Branch and jump instructions fall into the category "Branch". Instructions performing multiplication and other operations are classified into "Misc" category.

From figure 2-1, we can see that about 50% of executed instructions belong to ALU category. Considering instructions classified into Memory[2] category, about 70% of executed instructions requires ALU[3].

[2] In our reference model, data-transferring instructions need ALU to calculate their data-address.
[3] Branch instructions have an independent address adder in our reference model.

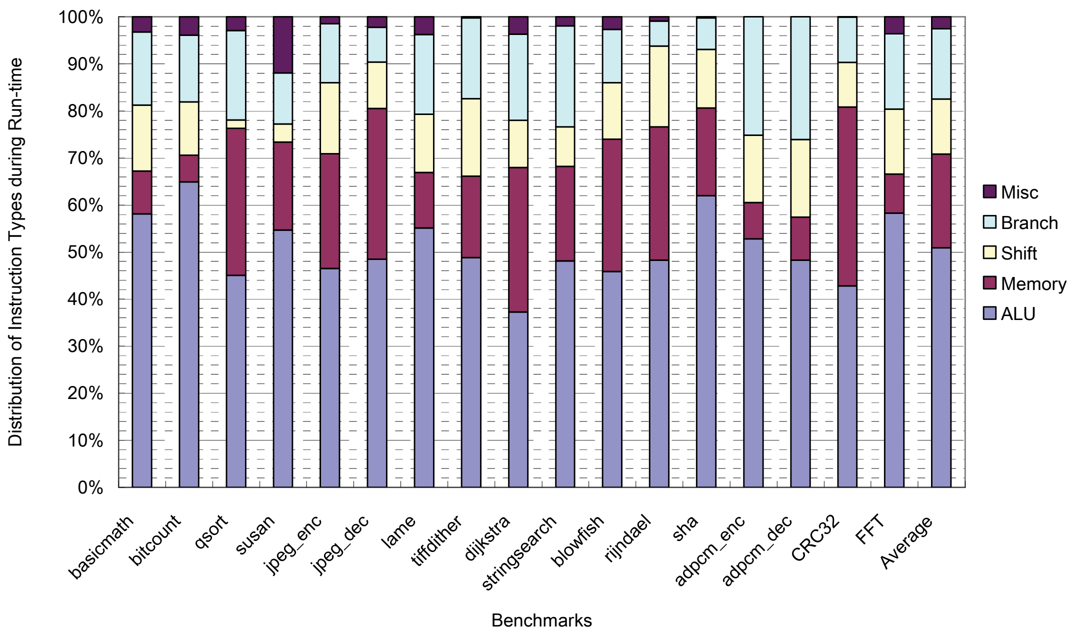


Figure 2-1 Run-time instruction types distributions

Next we perform significant bit-width analysis in performed ALU operations in our benchmark suite. Figure 2-2 shows cumulative ratio of executed ALU operations by data-processing instructions.

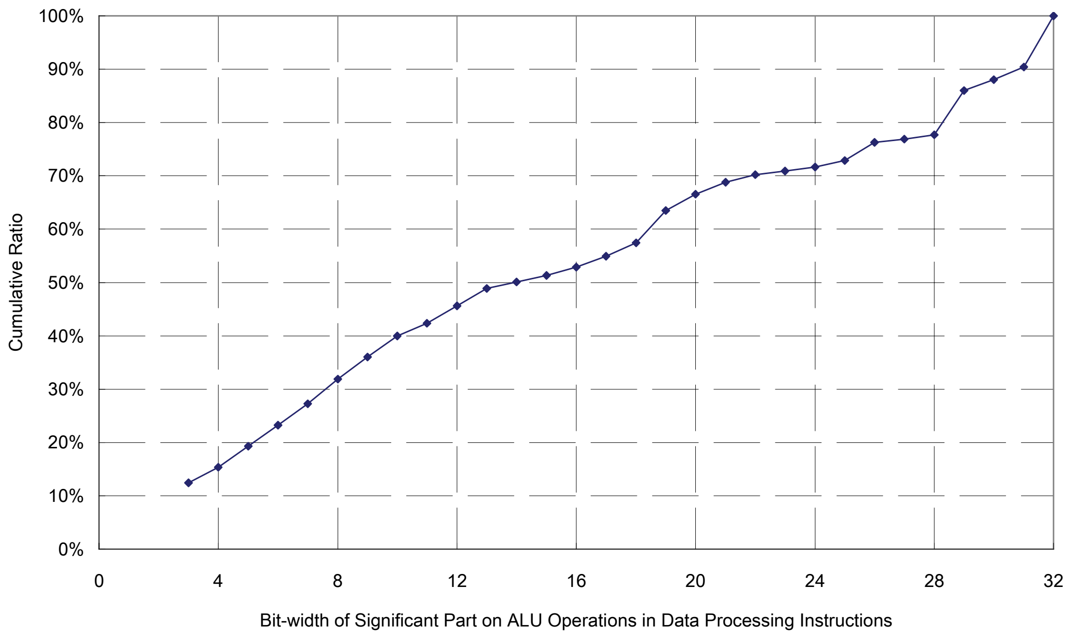


Figure 2-2 Cumulative distributions of significant bit-widths of ALU operations in data-processing

instructions

As we can see, about 53% of such operations only require half or less width of ALU in a 32-bit architecture.

Figure 2-3 shows the same statistics on ALU operations executed by data-transferring instructions.

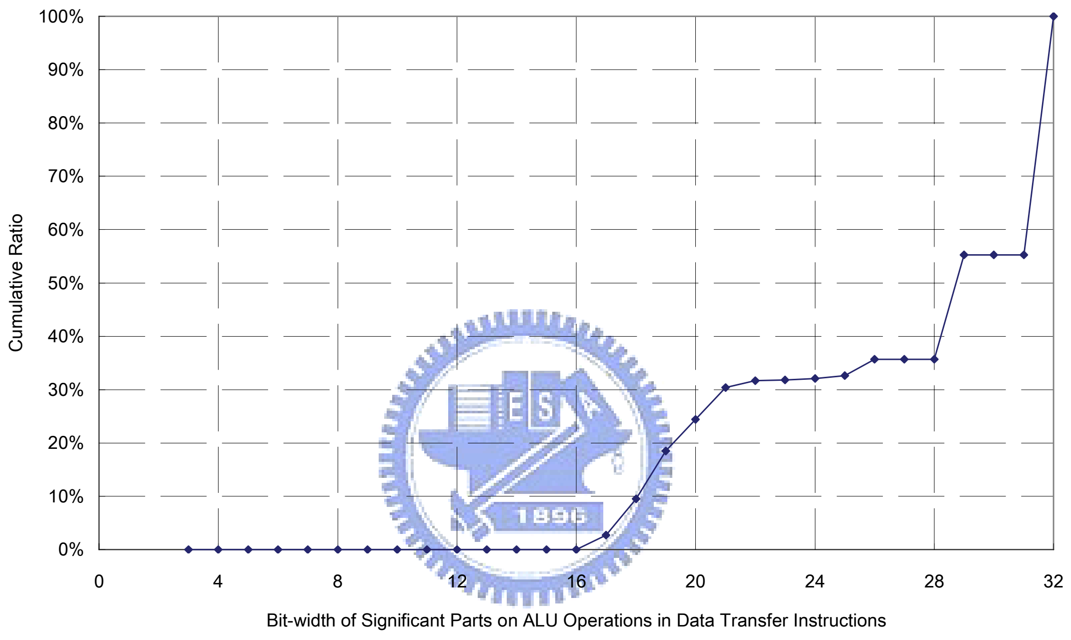


Figure 2-3 Cumulative distributions of significant bit-widths of ALU operations in data-transferring instructions

Due to the characteristics of addressing range, most ALU operations performed by Load / Store have wider significant part.

Figure 2-4 shows statistics on all executed ALU operations. This is the distribution of significant bit-width of ALU operations in a typical 5-stage MIPS-like single pipeline machine.

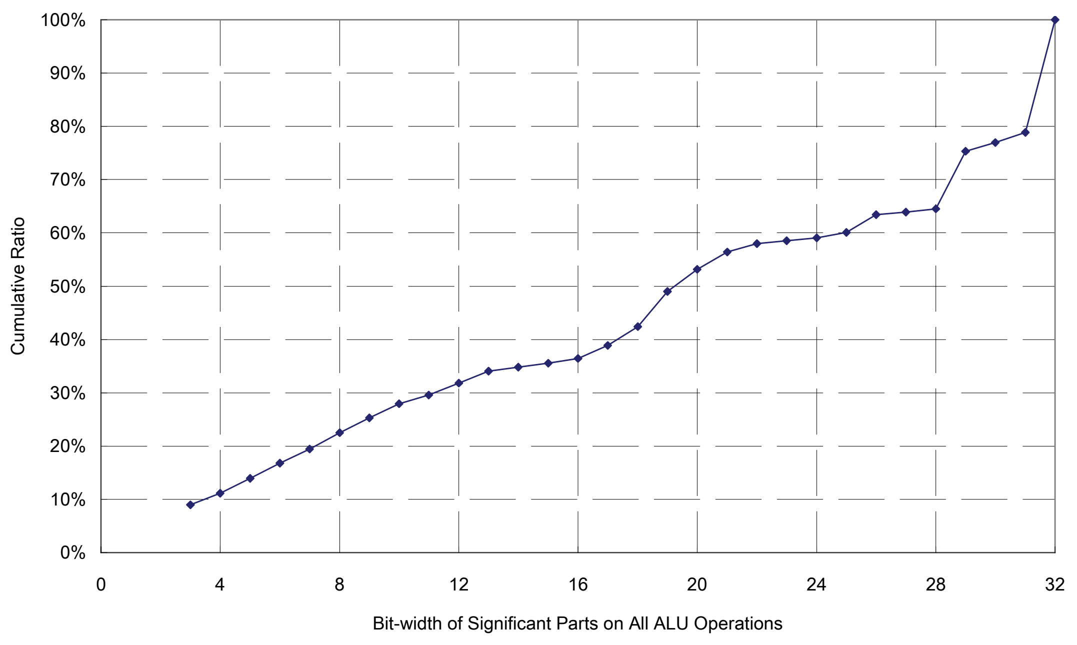


Figure 2-4 Cumulative distributions of significant bit-widths of all ALU operations

Although significant bit-widths of ALU operations in data-transferring instructions make the overall significant bit-width wider, about 49% of ALU operations require less than or equal to 19-bit of actual ALU width.

From the profiling results above, if the part of the ALU which is originally occupied with insignificant parts could be shared with other ALU operations, we have high opportunity to improve the performance.

2.2. Datapath in Multi-Bitwidth Pipeline

A single 64-bit datapath design which can perform one 64-bit operation or four 16-bit operations is proposed in [Loh 2002]. They propose a Multi-Bit-Width (MBW) micro-architecture which takes the wires normally used to route the operands and bypass the result of a 64-bit instruction, and instead uses them for multiple narrow-width instructions.

They divide the ALU into four 16-bit ALU blocks, each having *independent function-controls*. Operand sources are read from reservation station (RS). Figure 2-5 shows the bus lines and ALU usage when performing a 64-bit operation. Bus lines of each operand and result are divided into four groups in the figure, each group contains 16 bits of data.

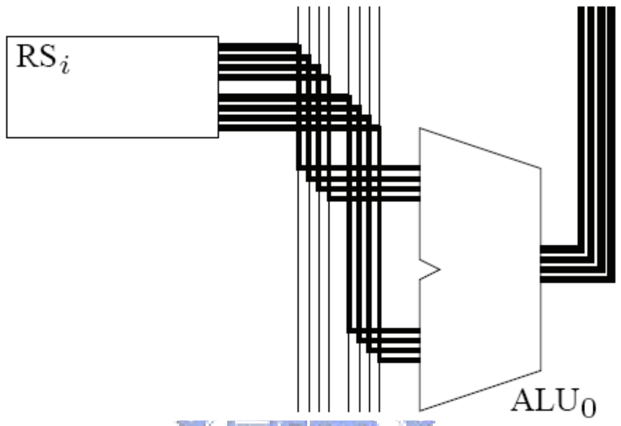


Figure 2-5 Data path for a single 64-bit instruction

Figure 2-6 shows the bus lines and ALU usage when performing two 16-bit operations. Operands of the two operations, j and k, are stored in RS_j and RS_k. Significant bit-widths of both operations are 16-bit and two ALU blocks are to be used. Bit 0 to 15 of the operand bus lines are occupied by the lower 16-bit operand-bits of operation j and so are the ALU block whose input were the lowest 16-bit data on the bus. Significant operand-bits of operation k are to be place between the 16th to 31st lines. Before operand-bits are put on those lines, they are firstly *shifted* the correct bit-positions.

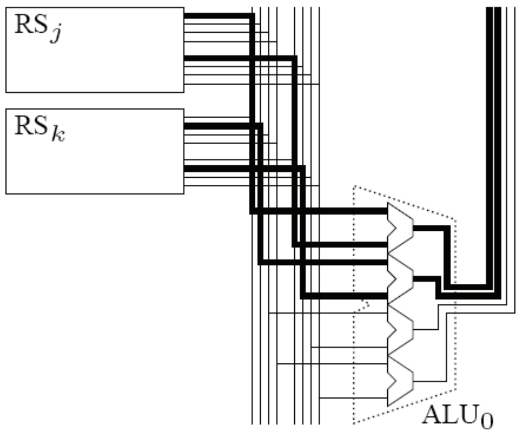


Figure 2-6 Data path for two 16-bit instructions sharing a single 64-bit datapath

This technique increases the effective issue width of a superscalar processor without adding many additional wires by reusing already existing datapaths. But it has the limitation that only instructions with data-width of 16-bit could share the proposed datapath. For two operations, one's data-width is 16-bit and the other's is 48-bit, the datapath cannot be shared by them.

2.3. More Flexible ALU-sharing Mechanism

The approach in the related work limits the significant bit-widths of instructions to 16-bit. We could relax the limitation by allowing operations which requires the multiples of 16-bits to share the ALU as long as the ALU could accommodate them. In the case of partitioning the ALU into four blocks and share by two operations, the relationship between possible inputs operand blocks and target ALU blocks is as figure 2-7[4]. Those upper B's mean blocks of operands of the operations. The lower B's means blocks of operands bus belonging to the ALU blocks.

[4] Similar concepts are proposed in "Value-Based Clock Gating and Operation Packing: Dynamic Strategies for Improving Processor Power and Performance", DAVID BROOKS and MARGARET MARTONOSI, 2000. But all operations must be the same. It's like a dynamic form of SIMD.

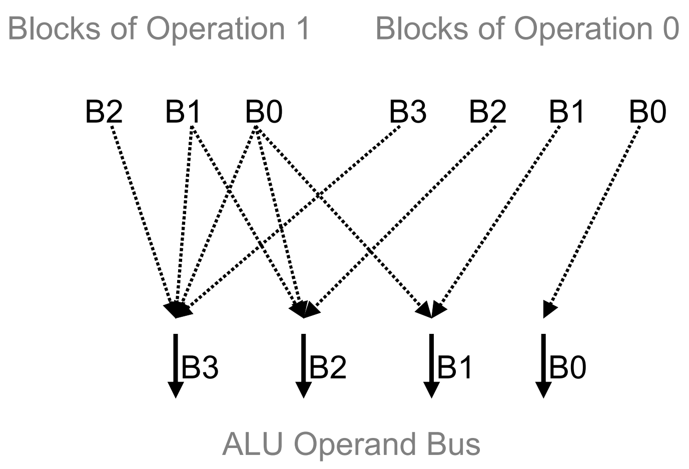


Figure 2-7 Relationship between possible inputs and target blocks

The limitations on data-width could also be relaxed by partitioning the ALU with finer granularity, which means the ALU blocks become smaller. For example, if we partition an ALU into eight ALU blocks. The number of possible combinations of data-widths of joined operations is increased.

But partitioning the ALU into smaller blocks certainly has some effects:

i. More complex circuits for aligning operands

In figure 2-8, the figure shows the differences when we partition the ALU with doubled number of ALU blocks.

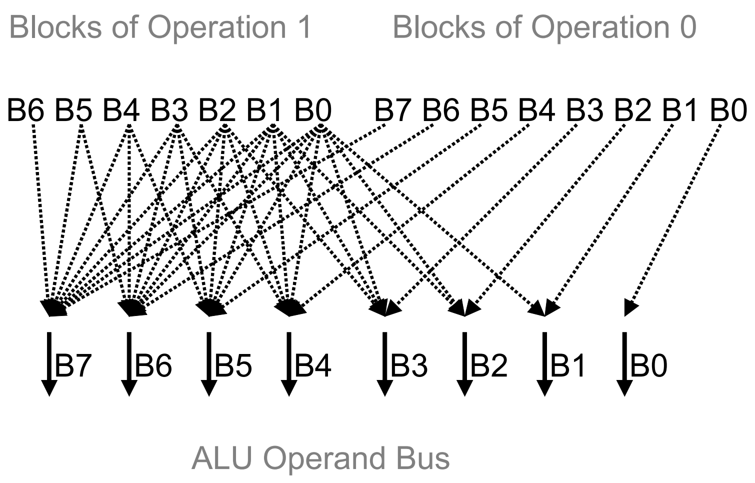


Figure 2-8 Relationship between possible inputs and target blocks of finer granularity

As the block size becomes smaller, the complexity for the bus lines to select operand bits increases. If we partition the ALU into *n* ALU blocks, the complexity of the number of inputs is $O(n^2)$. And in this case, it only supports sharing by two operations. If the ALU is going to be shared by more than two operations, the number of possible inputs would significantly increase.

ii. ALU consisting of smaller ALU blocks leads to longer delay

Another effect is the delay by the carry-propagation between ALU blocks. As we divide the ALU into smaller blocks and make the carry-signal propagate serially, the critical path could increase.

2.4. Motivation

When an operation is going to be joined to share the datapath, if its operand blocks occupy ALU blocks with turned around ordering, i.e. its least significant block is allocated to the ALU block in the highest block position, then each ALU block selects its input blocks from only two possible blocks. Figure 2-9 shows the relationship when we partition the ALU into eight blocks.

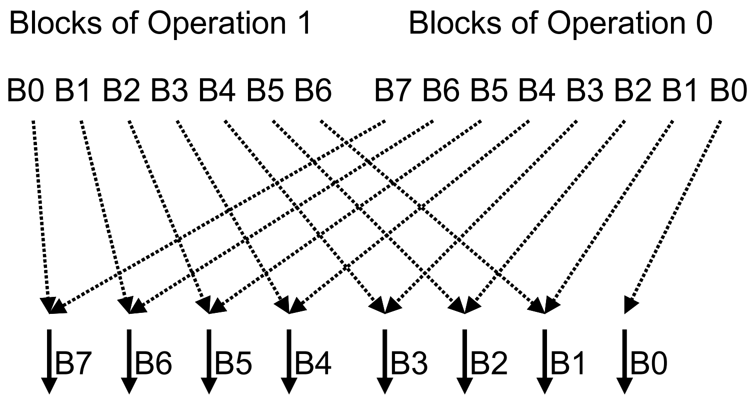


Figure 2-9 Turnaround approach

The complexity of the number of inputs now becomes *O(n)*. With the same flexibility, the required circuit for aligning operand blocks is significantly reduced.

2.5. Objective

In this work, we focus on *sharing an ALU by two operations*. The ALU in this thesis supports addition, subtraction, and bit-wise logic operations only. We designed a mechanism to share the buses and ALU in one single pipeline datapath by two ALU operations with:

i. High utilization of ALU and buses by flexible sharing

ii. Low overhead on space

Modifications on other part of datapath to integrate the design into a single pipeline are proposed to make the whole mechanism work.

Chapter 3. Design

3.1. Overview of Datapath-Sharing

In this section, we discussed about issues that must be considered when sharing one single pipeline datapath by two ALU operations. In the following discussions, our reference machine is a five-stage MIPS-like pipeline.

3.1.1. Constraints for Joining Two Instructions

When joining operations from two instructions, there are some constraints between the two instructions. These constraints come from structural hazards and data hazards.

3.1.1.1. Structural Hazards

Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously. Our design shares the ALU in one single pipeline datapath by two operations. In some cases the datapath can accommodate two operations and in other cases it cannot. This is determined by whether the ALU blocks could accommodate the two operations. Such procedure is called *Width-Check*. The width-check could be performed when information about the significant bit-widths (or blocks) for each operand is available. The width-check should be completed before we merge the operand blocks to share the ALU.

Except sharing the ALU and the operand / result buses, we did not duplicate other units such as data memory in current design, so we must make sure that no other resource conflicts exist between the two instructions except ALU and the operand /

results buses. This procedure is called *Type-Check* in our design. This could be performed by checking the opcode fields of incoming instructions while decoding.

3.1.1.2. Data Hazards

Since the two operations are executed in parallel, there must be no data-dependencies between the two. To illustrate the relationship among operations, we consider operations *i*, *j*, and k, with *i* occurring before *j* in program order. There are three types of data hazards that we should consider:

RAW (read after write):

Since the two operations are executed in parallel, if there exists RAW between *i* and *j*, *i* should be performed alone. In next cycle, operation *j* could be performed due to the availability of its operand value.

WAW (write after write):

When *i* and *j* are executed in parallel and try to write the same destination, the WAW may arise. Since no out-of-order execution in our design is adopted, the WAW hazards should not occur.

WAR (write after read)

When *j* tries to write a destination before it is read by I, WAR would arise. In our design, we only check two consecutive instructions to see when they can be joined. This is a static issue pipeline so that WAR will never happen since operands of the two operations are read simultaneously in ID stage.

Summarize the discussion above. When performing data-dependency check, we only have to make sure that there exist no RAW dependencies between the two operations. The data-dependency check could also be performed by checking the destination register filed of *i* and source register field of *j* while instruction-decoding.

3.1.2. How ALU is Shared by Two Operations

To share a single ALU by two different operations, we divide the ALU[5] into several ALU blocks and bits of operands are also grouped into blocks according to their bit-positions. Each ALU block has its own function-select signals. When the ALU is going to be shared by two operations, four *m-bit* operands from the two operations are first merged into two *n-bit* operands by OML (operand-merging logic). The merged *n-bit* operands are input to the ALU. The output *n-bit* result consists of result blocks from the two operations. Before the results are written into register-file or used as address to data-memory, the results must be sign-extended to full-width results by SXL (sign-extending logic).

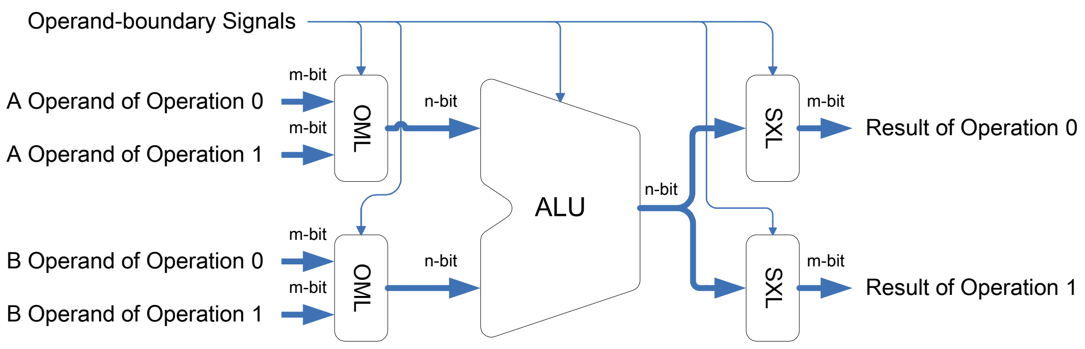


Figure 3-1 Block diagram of sharing the ALU by two operations

Extra control signals are required for the OML, ALU, and SXL to known which

[5] The ALU in our design performs addition, subtraction, and bit-wise logic operations

blocks belong to which operation. Such signals are named after *operand-boundary signals*.

3.1.3. Possible Modifications in a Five-stage MIPS-like Pipeline

Now we propose how these mechanisms could be possibly integrated into a five-stage MIPS-like pipeline.

Instruction-Fetch:

Since we are able to perform two ALU operations, up to two consecutive instructions are fetched. An instruction queue with two entries (entry *0* and *1* – the instruction in entry 0 are before the instruction in entry 1 in program order) is required. When only one (the instruction originally in entry 0) is consumed in previous cycle, the instruction originally in entry 1 is moved to entry 0 and the fetcher fetches one instruction and put it into entry 1. If the two instructions can be joined, then next two instructions are fetched and put into the queue.

Instruction-Decode:

We have to add one extra instruction-decoder for the additional instruction that might be joined. Logic for the additional decoder could be simpler than a regular instruction-decoder since the decoder doesn't have to recognize all types of operations – only operations that require ALU during execution stage must be recognized. Besides, the type-check and data-dependencies check should be performed while instruction-decoding.

Operand-reading is also performed in this stage. Before we perform the width-check to make sure the ALU is able to accommodate the two operations, the number of significant blocks of each operand is required. Since the data-width of each operand must be decided dynamically, we could obtain the number of significant blocks of each block when it's read out from register-file or the immediate filed in instruction words. The procedure which determines the number of significant blocks of a value is called *width-determination* and performed by *WDL (Width-Determination Logic)* in our design. The results from WDL are sent to width-check logic. In this stage, we should perform three checks – type-check, data-dependency check, and width-check. Only when all the three checks are passed the two operations can be joined. Whether the two operations can be joined must inform the instruction-fetcher so it can correctly update the instruction queue.

In this stage, the operand-boundary signals must also be generated so that in the latter stages the logic involved with the two joined operations could function correctly.

Execute:

Operands from the two operations are merged and put on the operand bus to the ALU. Before input to ALU, the merged operands might be swapped.

Memory Access and Write-back:

Since results are also joined, so they should be separated and sign-extended when sending address to memory address port and writing back to register-file. How the joined results should be sign-extended is decided by operand-boundary signals.

In the following sub-sections, we have closer discussions about the design.

3.2. Modifications in Instruction-Decode Stage

3.2.1. Type-Check and Data-dependency Check

An additional instruction-decoder for the instruction that might be joined is required. The decoder checks the opcode in the instruction word and to see if it falls in the type that could be joined or not.

The type-check and data-dependency check are performed while instruction-decoding. Since the information that is required for performing these two checks all exists in instruction words, the two checks can be performed in the early half-cycle in ID stage. If one of the two checks fails, the two instructions cannot be joined. Each of the check outputs a signal to represent whether it passed. The signals are useful for width-check for generating operand-boundary signals and have influences on instruction-fetching.

3.2.2. Width-Check

Besides type-check and data-dependency check, width-check is performed in ID stage. The information that the width-check needs is the number of significant blocks for each operand and the results from type-check and data-dependency check. The width-check logic generates operand-boundary signals according to whether the two operations can be joined together.

3.2.2.1. Width-Determination Logic

Width-determination logic is used to decide the number of significant blocks of a value. The significant bits of a value should be sufficient to represent its magnitude and sign (or its most significant bit). But according to our observation, to correctly sign-extend the result to full-width, the required bit-width of an operation should be the maximal significant bit-width among its operands plus one bit. So the width-determination logic checks how many blocks are required to accommodate the significant bits of a value plus one bit.

To determine the bit-width of a value, we must know that beginning from the most significant bit in the value to the least significant bit, at which bit-position the bit value firstly differs from the most significant bit. Once we found the bit-position, say bit *i*, we know that the significant bit-width of the value is *i+3*. (If *i+3* exceeds the word width of the architecture, then the significant bit-width should be the full-width of the architecture.) Such function can be simply achieved by exclusive-OR each bit with the bit value of most significant bit and connect result to a priority-encoder. The output from the priority encoder plus 3 is the significant bit-width we defined.

In our design, each bit in a value is divided into several blocks according to its bit position. Assume a value is divided into *m* blocks (from block *0* to block *m-1*) and each block has a fixed size *n*. A block might be insignificant when the n bits within it all have the same values. But simply checking the *n* bits is not enough, it can only tells between whether this block *might* fall in the bit-range to represent the magnitude. In our definition, besides bits to represent the magnitude, we have to reserve two bits. For block *i*, only when the *n* bits in block *i* and the highest two bits in block *i-1* all have the same bit value, block *i* could be insignificant. If we check only the highest bit in block *i-1*, when we try to sign-extend result from the highest bit in block *m-1* could be incorrect due to carry in an addition operation that changes the bit-value of the bit-position which we originally regard as the correct bit for sign-extension.

$$BR_i = \overline{\left(b_{(i*n)+n-1}b_{(i*n)+n-2}\cdots b_{(i*n)}b_{(i*n)-1}b_{(i*n)-2}\right) + \overline{\left(b_{(i*n)+n-1} + b_{(i*n)+n-2} + \cdots + b_{(i*n)} + b_{(i*n)-1} + b_{(i*n)-2}\right)}}$$

$$BR_0 = \overline{\left(b_{n-1}b_{n-2}\cdots b_0\right) + \overline{\left(b_{n-1} + b_{n-2} + \cdots + b_0\right)}}$$

Figure 3-2 Logic functions for generating BR signal

Assume we have a bit with its highest order bit which differs from the most significant bit at bit *j*. Bit *j* falls in the bit-range of block *k*. The BR signals of block *k+2*, *k+3*, …, *m–1* must all be unset since all the bits within them are all the same. BR_{k+1} is unset only when $j < (k+1)*n-2$ and is set only when $j \geq (k+1)*n-2$, i.e. the highest two bits in most significant block must be the same with the most significant bit of the full-width value. Figure 3-3 shows the logic for generating a BR signal. The "One-Detection" logic is logically a NAND gate. The "Zero-Detection" logic is logically a NOR gate.

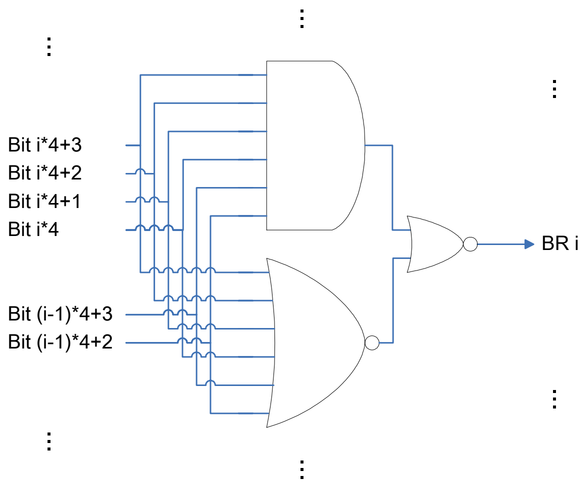


Figure 3-3 Logic to Generate BR Signals

Beginning from BR_{m-1}, one we find the first BR which is set, say BR_k, then block *k*, block *k-1*, …, block *0* all belong to significant blocks. If we connect those BR signals to a priority encoder, then its output is an encoded value which represents the number of significant blocks of the value.

The priority encoder encodes the input BR signals as the following rules:

Input BR Signals					Encoded Output Value
BR_{m-1}	BR_{m-2}	…	BR_1	BR_0	(Number of Significant Blocks)
1	x	xxx	x	x	m-1
0	1	xxx	x	x	m-2
		……			…
0	0	0	1	x	1
0	0	0	0	1	0

Table 3-1 Function of priority encoder for width-determination logic

Figure 3-4 shows the block diagram of the width-determination logic.

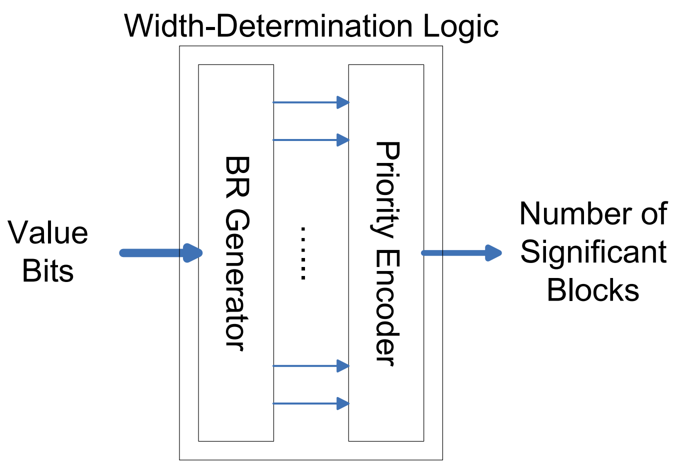


Figure 3-4 Width-Determination Logic

Figure 3-5 shows the schematic of priority encoder in Width-Determination Logic.

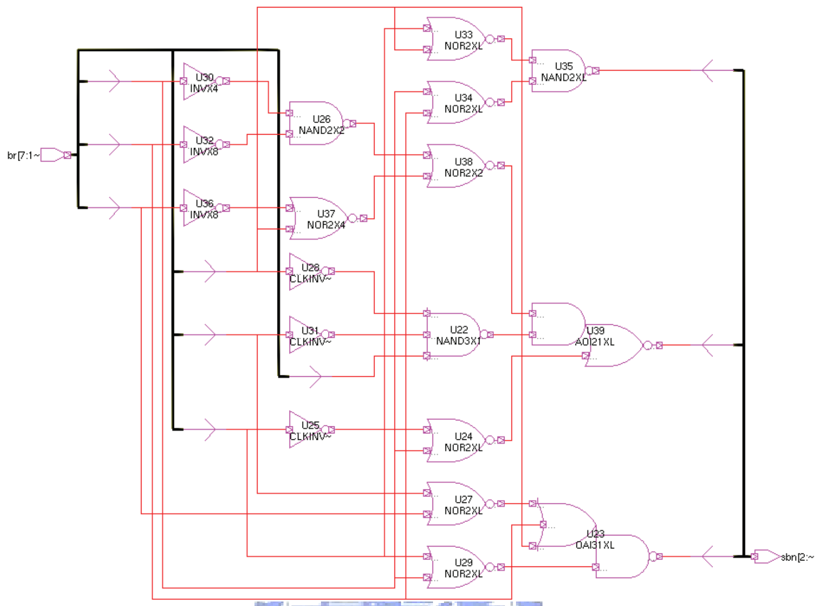


Figure 3-5 Schematic of the Priority Encoder

The delay of width-determination logic is proportional to the block size *n* and the number of blocks, *m*. The actual delay of WDL in .18 um process technology is about 0.47ns when m=10 and n=4. However, the numbers of significant blocks generated by width-determination logic is required for width-check. If we always perform width-determination when operands are ready from register-file, then the width-check may not fit in the ID stage. In the load / store architecture, operands come either from register-file or immediate field in instruction word and writing a value into register-file usually requires shorter time than reading. If we perform width-determination when a value is written into the register-file and store the number of significant blocks in extra fields in register-file, then the delay between register-read and width-check could be removed.

3.2.2.2. Operand-boundary Signals

The operand-boundary signals are used to represent how two operations share the ALU and bus. It indicates that from which block the joined operation starts to occupy. Thus the operand-merging logic (OML), ALU, sign-extending logic (SXL) all needs the operand-boundary signals.

Operand-boundary signals are generated by width-check logic. The number of conditions it represents amounts to the number of partitioned operand-blocks.

Rules to decide the operand-boundary signals:

Assumption:

Operation i requires x blocks

Operation j requires y blocks

ALU is partitioned into m blocks with the same size

(x and y must be less than or equal to m)

If $x + y \leq m$ and both type-check and data-dependency check passed, operand-boundary signal is set to $x - 1$. The first x blocks are occupied by operation i and the rest are for operation j.

If $x + y > m$ or type-check or data-dependency check does not pass, then operand-boundary signal is set to $n - 1$, which means all blocks are allocated to operation i.

There exists decoding logic to generate control for each block in OML, ALU, and SXL. The decoding logic generates control signal as the following table:

Operand-boundary Signals	Signal for block m-1	Signal for block m-2	……	Signal for block 2	Signal for block 1	Signal for block 0
m-1	0	0	…	0	0	0
m-2	1	0	…	0	0	0
…	……					
2	1	1	…	0	0	0
1	1	1	…	1	0	0
0	1	1	…	1	1	0

Table 3-2 Function of decoding operand-boundary signals

For OML, the decoded signals can be used to select blocks from operation *i* (when the signal = 0) or operation j (when the signal = 1). For each ALU blocks, the decoded signals can be used to select function-select signals and carry-in sources. For SXL, these decoded signals can be used to indicate the block value should be directly bypassed or filled with sign bits.

3.2.2.3. Width-Check Logic

Width-check logic is used to determine whether two operations can be joined by checking the limitation of ALU width. Its inputs are number of significant blocks of operands and the result of type-check and data-dependency check. It outputs the operand-boundary signals.

The width-check first decides the required number of blocks for each operation – which is identical to the larger number of significant blocks of its operands. Then required numbers of blocks for the two operations are added together to see if it exceeds the number of ALU blocks. Finally, the width-check logic generates operand-boundary signals.

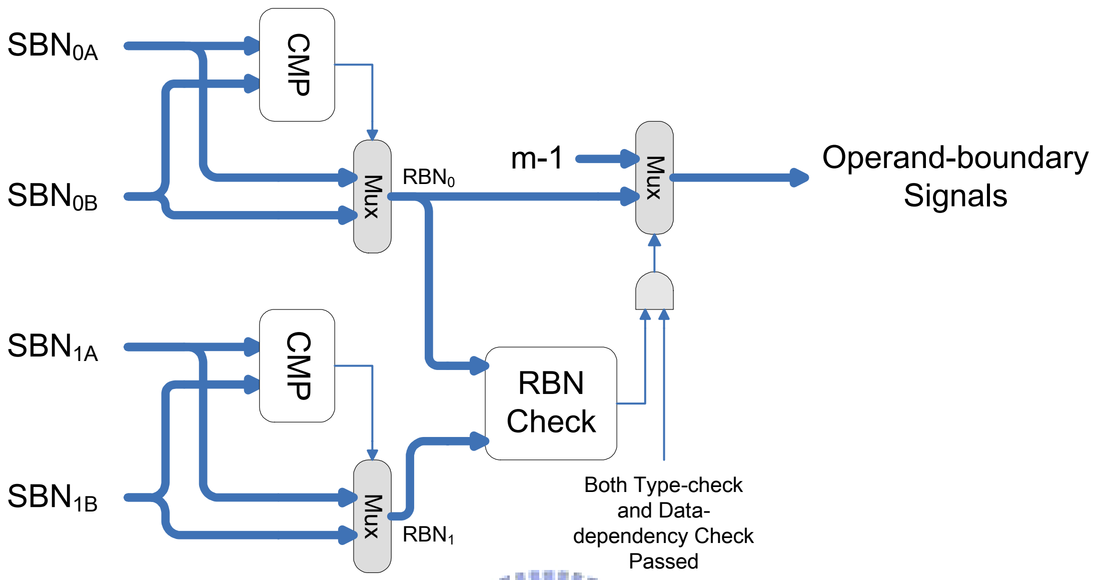


Figure 3-6 Block diagram of the Width-Check Logic

Figure 3-6 shows the block diagram of the width-check logic. The widths of the bold lines are identical. Assume we divide the operands into *m* blocks. In the figure, SBN means number of significant blocks. SBN is generated by Width-Determination Logic. SBN_{0A} means the number of significant blocks of A operand of operation 0. The SBN signals are sent into a comparator (CMP) to tell which one is larger. Since the larger one could represent the number of required blocks for the operation. The CMP generates a select signal for the mux to choose output between the two SBNs. Then the number of required blocks of the operation is selected from the larger SBN and called RBN.

To compare two SBN, the delay of CMP logic varies with $\log_2 m$ and the fan-in of logic gates. The output equation of the CMP logic is whether SBN_A is greater than SBN_B. The Boolean equation is

$$SBN_A gtSBN_B = a_{n-1}\overline{b_{n-1}} + i_{n-1}a_{n-2}\overline{b_{n-2}} + \cdots + i_{n-1}i_{n-2}\cdots i_1 a_0 \overline{b_0}$$

$$\text{Where} \quad n = \lceil \log m \rceil, \;\; i_k = a_k \oplus b_k,$$

For the case when m=8,

$$SBN_A gtSBN_B = a_2\overline{b_2} + i_2 a_1 \overline{b_1} + i_2 i_1 a_0 \overline{b_0}$$

The schematic of the CMP logic is shown in figure 3-7.

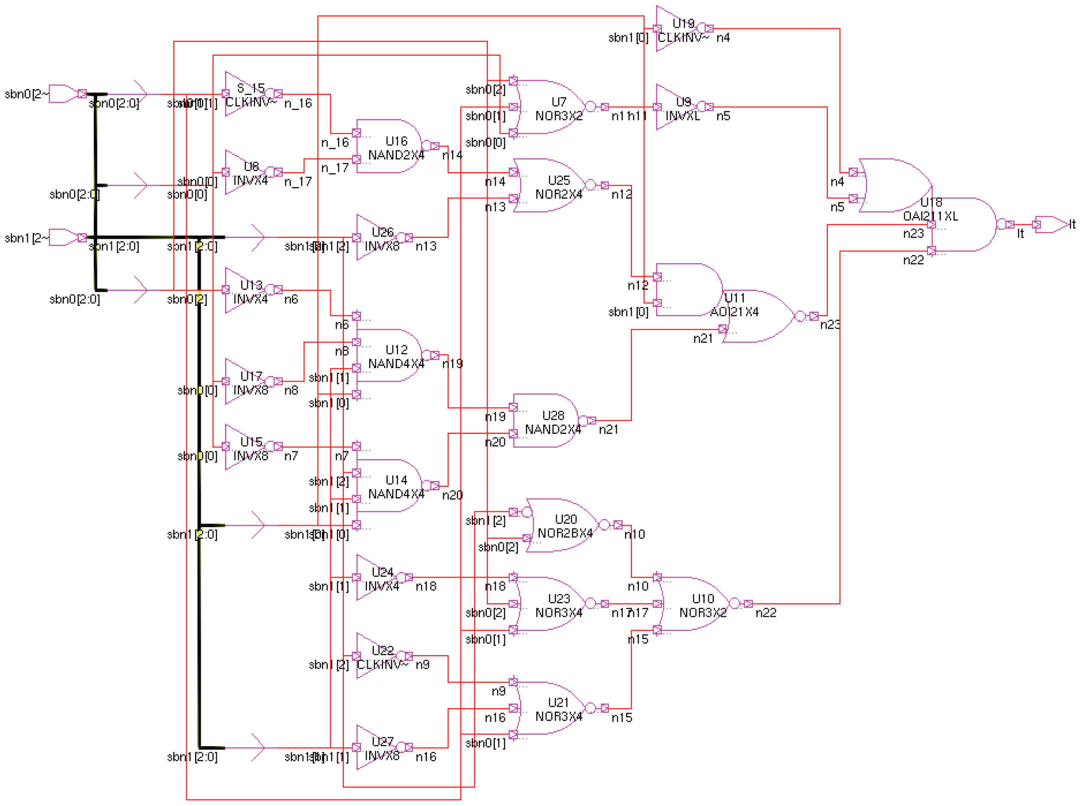


Figure 3-7 Schematic of CMP Logic

Then the mux could choose the larger one between SBN_A and SBN_B to represent the number of required blocks of the operation. We call the number of required blocks of an operation *RBN*. After choosing the numbers of required blocks for the two operations, they are sent to logic to check whether the two belong to combinations that the ALU can accommodate. The check is performed with the RBN Check logic according to the following equation.

$$RBN_0 + RBN_1 \le m - 2$$

Take the case when m=8 for example. The RBN check is passed when the following conditions occurs.

RBN_0	RBN_1
110	000
101	000, 001
100	000, 001, 010
011	000, 001, 010, 011
010	000, 001, 010, 011, 100
001	000, 001, 010, 011, 100, 101
000	000, 001, 010, 011, 100, 101, 110

Table 3-3 Possible RBN combinations when m = 8

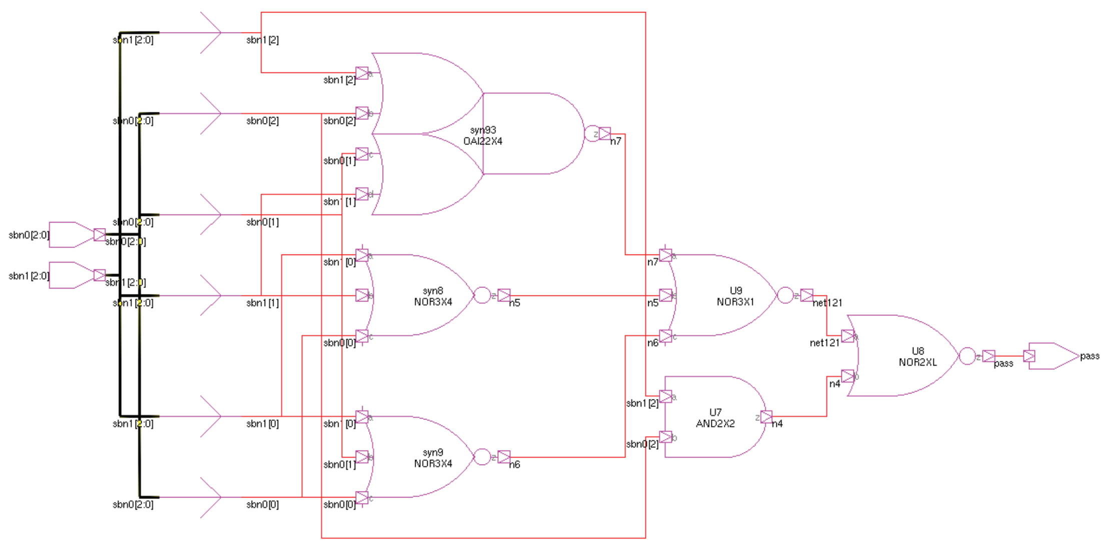


Figure 3-8 Schematic of the RBN check logic (m = 8)

If the RBN passed and both type-check and data-dependency check passed, the operand-boundary signals are set to the RBN of operation 0. Otherwise, the operand-boundary signals are set to m-1, which means all ALU blocks are occupied by operation 0.

3.3. Modifications in Execution Stage

When the ALU is going to be shared by two operations, four *m-bit* operands from the two operations are first merged into two *n-bit* operands by OML (operand-merging logic). The merged *n-bit* operands are input to the ALU. Before the *n-bit* results are written into register-file or used as address to data-memory, they are sign-extended to full-width results by SXL (sign-extending logic).

Based on our observation, for sharing the ALU blocks by two operations, one operation occupies ALU blocks in regular ordering and the other does in turned around block ordering to increase the flexibility of sharing. In following sub-sections, we will show the design based on the turn-around approach.

3.3.1. Deciding the Block Width

The first decision we met is how to group operand bits into blocks. Should every block have the same size? From the observation on significant bit-width distribution in chapter 2, there's no obvious tendency towards some specific bit-width. So bits in a value are uniformly grouped into blocks with the same size in our design. From the block width of operand blocks, we can infer the minimal ALU block width so that the ALU could satisfy the granularity. To deserve to be mentioned, unlike operand blocks, *the ALU does not always need to be partitioned into ALU blocks with the same width.*

This issue will be discussed in section 3.3.3.1. Now we assume each ALU blocks has the same width that is identical to the width of an operand block.

So what width is good considering both circuit delay and the flexibility to fully use the ALU? As the block width goes narrower, the flexibility increases but circuit delay may also become longer. To make the decision, related experiments are performed in chapter 4.

3.3.2. Merging Operands from Two Operations

Assume we divide the ALU into *n* blocks and two operations, operation 0 and operation 1, which can be joined together. To merge operand blocks of operation 1 with the significant blocks of operation 0, extra 2-to-1 multiplexers to select operand bits from operation 0 and operation 1 are required. Those multiplexers are controlled according to operand-boundary signals. Those multiplexers are grouped into eight groups, from multiplexer group 0 to multiplexer group *n-1*. Inputs to the multiplexer group 0 are operand block 0 of operation 0 and operand block *n-2* of operation 1; inputs to the multiplexer group 1 are operand block 1 of operation 0 and operand block *n-3* of operation 1 and so on.

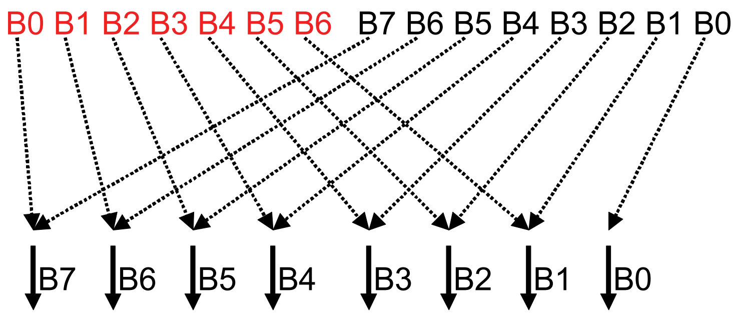


Figure 3-9 Possible operand blocks with turn-around approach when n = 8

In figure 3-10, relationship between operand blocks and the grouped multiplexers are shown.

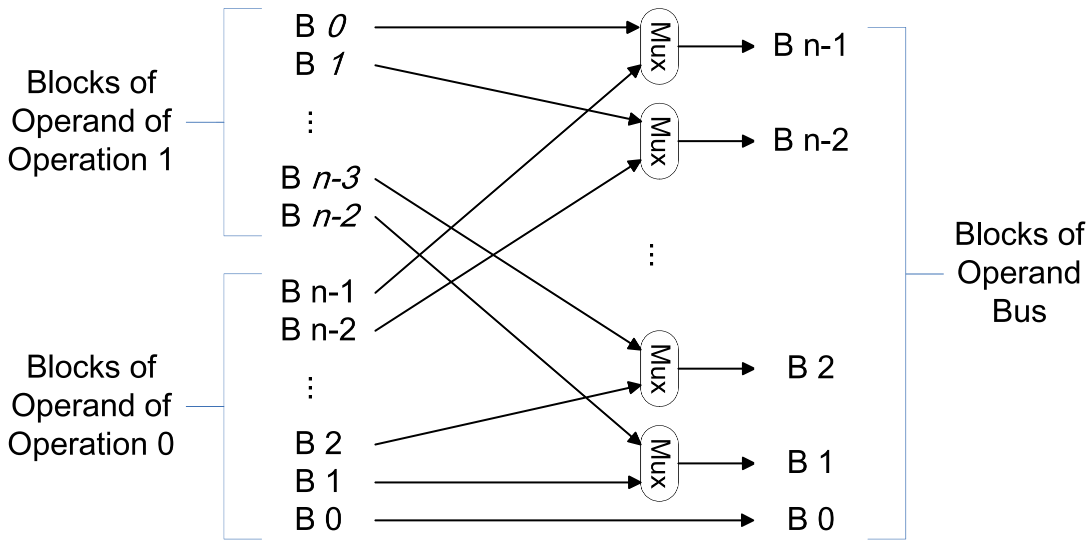


Figure 3-10 Logic to merge operand blocks with turn-around approach when n = 8 (each mux consists of several 2-to-1 multiplexers)

Due to the turn-around approach, there exists no alignment problem so that the circuits are so simple.

The logic for merging operands from two operations is called OML (Operand-Merging Logic) in our design. Figure 3-11 shows the block diagram.

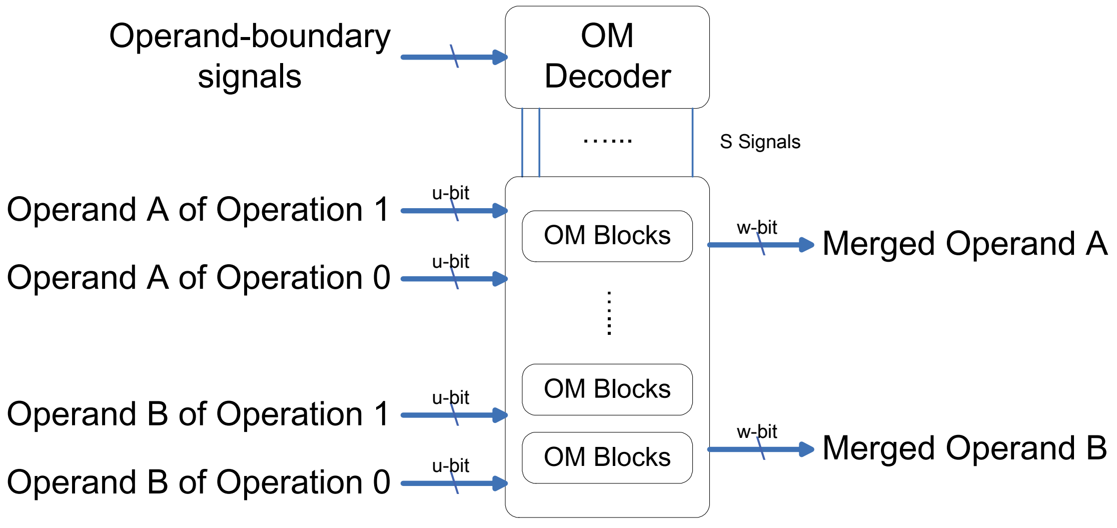


Figure 3-11 Block diagram of OML

The OM Decoder decodes Operand-boundary signals and generates S signals to control OM blocks so that they select the correct blocks from input blocks. Each block has an independent S signal. The relationship between S signals and operand-boundary signals is as table 3-2. A sample logic diagram for generating S signals for five blocks is shown in figure 3-12.

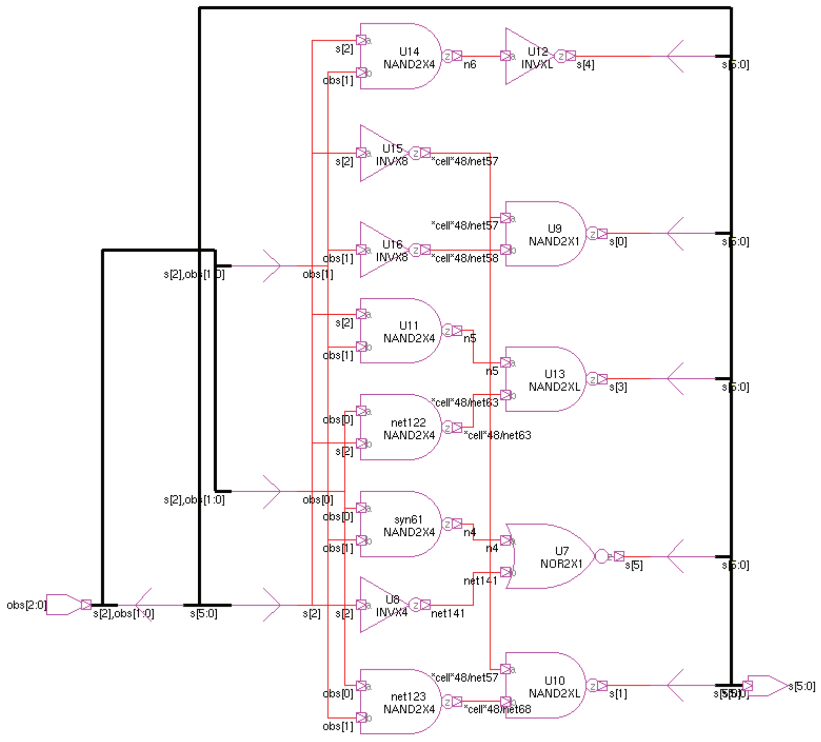


Figure 3-12 OM Decoder Logic

The design of an OM block is shown in figure 3-13.

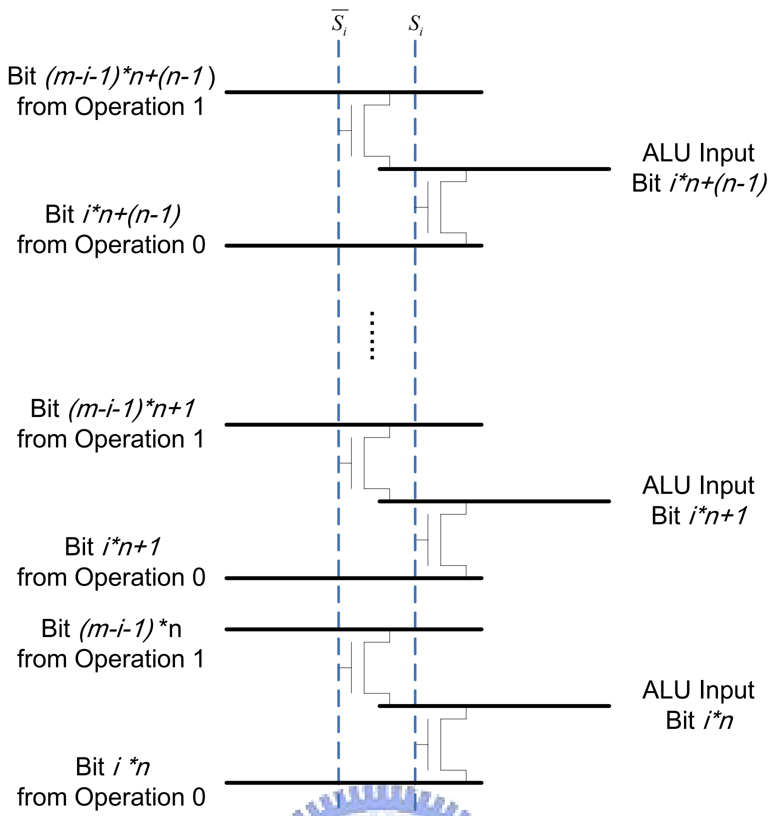

$\overline{S_i}$
S_i
Bit *(m-i-1)*n+(n-1)* from Operation 1
ALU Input Bit *i*n+(n-1)*
Bit *i*n+(n-1)* from Operation 0
Bit *(m-i-1)*n+1* from Operation 1
ALU Input Bit *i*n+1*
Bit *i*n+1* from Operation 0
Bit *(m-i-1)**n from Operation 1
ALU Input Bit *i*n*
Bit *i *n* from Operation 0

Figure 3-13 OM Block

3.3.3. ALU in the Turnaround Approach

Since the ALU is performing two operations, function-select signals for two operations are required. When the ALU is shared by two operations, function-select for ALU blocks in higher positions differs from that for ALU blocks in lower positions. To reuse the existing lines for function-select signals, we put function-select signals for the two operations on two ends of the lines and pass-transistors are put in the lines to “cut” the two function-select controls at appropriate position. the relationship between function-select lines, pass-transistors, and ALU blocks is shown in figure 3-14.

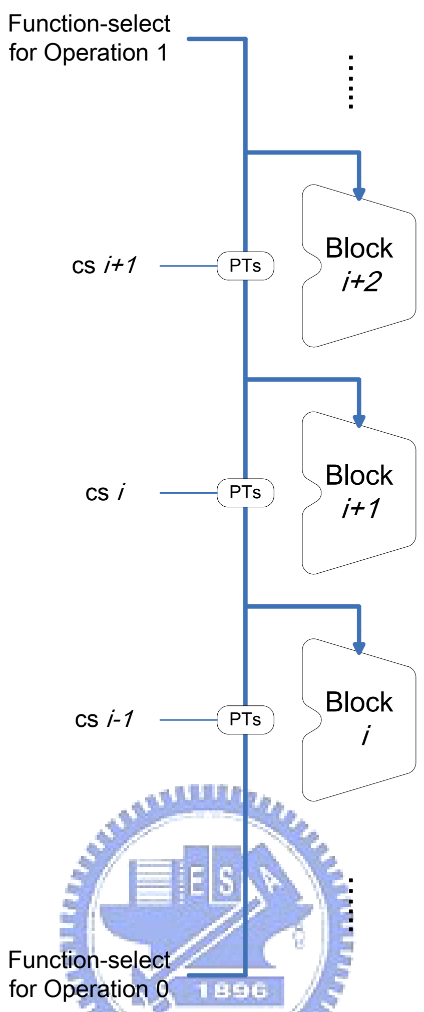

Function-select for Operation 1
cs *i+1*
PTs
Block *i+2*
cs *i*
PTs
Block *i+1*
cs *i-1*
PTs
Block *i*
Function-select for Operation 0

Figure 3-14 Function-Select Lines, Pass-Transistors, and ALU Blocks

The CS signals are used to control the pass-transistors so that the signals on the lines could be stopped at appropriate position. The CS signals are generated according to operand-boundary signals and the mapping is shown in table 3-4.

Operand-boundary Signals	CS m-1	CS m-2	……	CS 2	CS 1	CS 0
m-1	0	1	…	1	1	1
m-2	1	0	…	1	1	1
…			……			
2	1	1	…	0	1	1
1	1	1	…	1	0	1
0	1	1	…	1	1	0

Table 3-4 Function of CS Decoder

The PTs is a group of pass-transistors controlled by a single CS signal. The logic for generating CS signals in shown in figure 3-15.

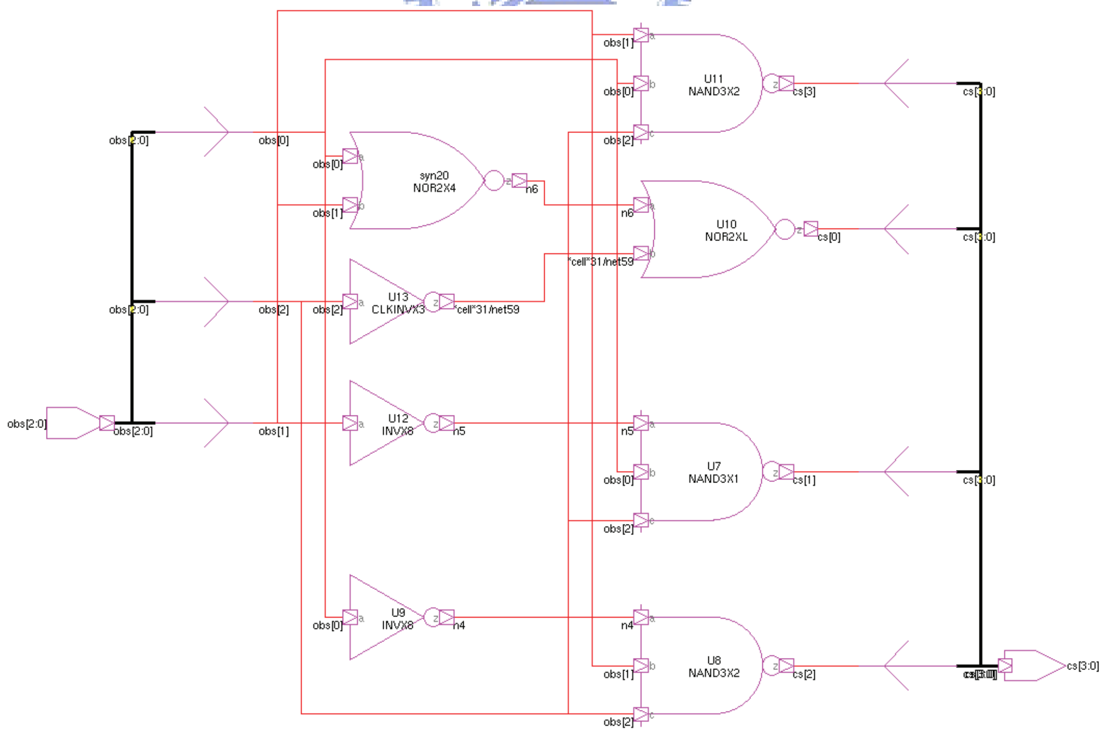


Figure 3-15 CS Generating Logic

A PTs is shown in figure 3-16.

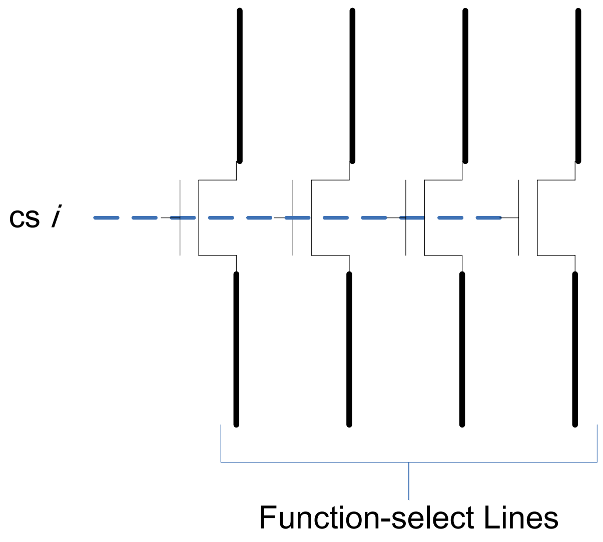
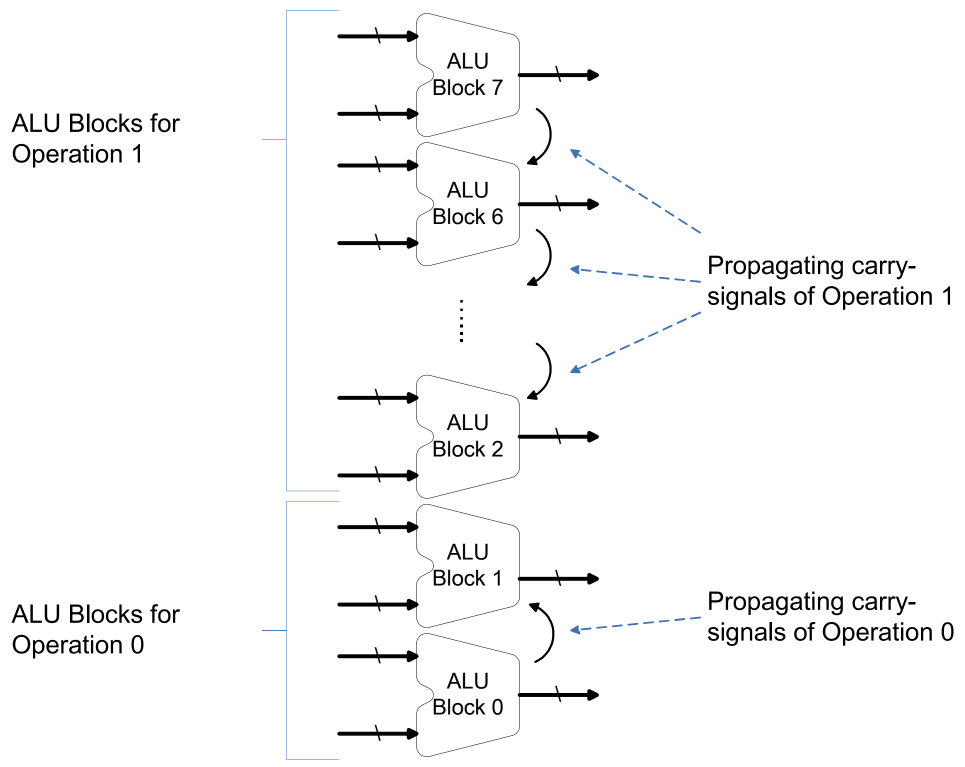


Figure 3-16 Schematic of a PTs

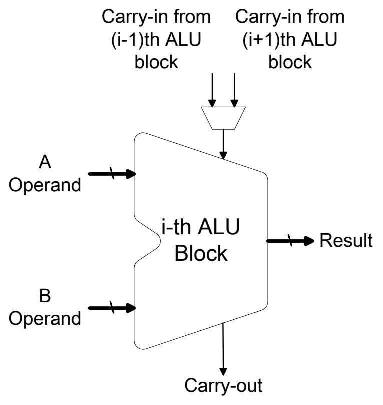
The turnaround ordering of the joined operand blocks has no effect on bit-wise operations but its does on addition. The direction of carry-chain for the joined operation is different from traditional designs. Carry-signal is propagating from higher stage of ALU block to lower stage. So the carry-in signal is the carry-out from its precedent stage if the ALU block is working for operation 0. If the ALU block is working for operation 1, its carry-in signal is the carry-out from its following stage. Figure xxx shows the directions of carry-propagating between ALU blocks. Figure 3-17 shows the direction of carry-propagation. The ALU is divided into eight blocks. Operation 0 requires 2 blocks and operation 1 requires 6 blocks.



ALU Blocks for Operation 1
ALU Block 7
ALU Block 6
ALU Block 2
ALU Block 1
ALU Block 0
Propagating carry-signals of Operation 1
ALU Blocks for Operation 0
Propagating carry-signals of Operation 0

Figure 3-17 Directions of carry-propagation in turnaround ordering scheme

A multiplexer is placed at the carry-in port of each ALU block as figure 3-18 to accomplish the requirement.



Carry-in from (i-1)th ALU block
Carry-in from (i+1)th ALU block
A Operand
B Operand
i-th ALU Block
Result
Carry-out

Figure 3-18 Multiplexer to select carry-in sources for block-reversed ordering

Those multiplexers could incur additional delay. The worst case delay of ALU with such organization occurs when the ALU is not shared by two operations, i.e. the carry signal propagates from the lower order block to the highest order block.

Besides extra circuits to accelerate the generation of carry signal, we also propose an enhancement design by avoiding unnecessary partitioning the ALU.

3.3.3.1. Operation Swapping

Observing the ALU block requirements, we found that when the ALU can be shared by two operations, one operation must require less than or equal to half width of the ALU and the other operation must require larger than or equal to half width of the ALU. If we always put operand blocks of the operation which is allocated wider width of ALU at the lower stages of the ALU, the implementation of the lower half of ALU can be one large ALU block.

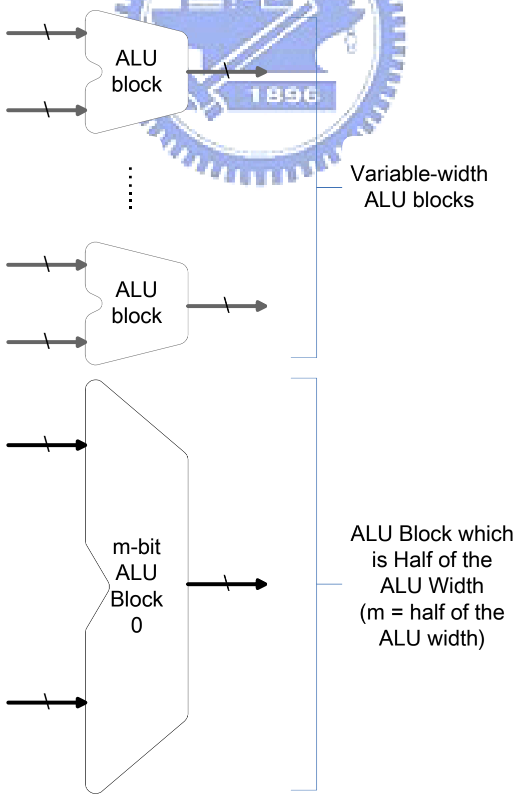

ALU block
Variable-width ALU blocks
ALU block
m-bit ALU Block 0
ALU Block which is Half of the ALU Width (m = half of the ALU width)

Figure 3-19 ALU organization for operation-swapping

Not only extra circuits and wires for function-selecting and selection of carry-in sources are simplified, but also the worst case delay of the ALU could be reduced.

To achieve such function, following signals are affected:

i. Block-ordering of the input operands should be turned around

ii. ALU control signals of the two operations should be swapped

iii. Destination register indexes are swapped

iv. Signals which decide the boundary of the two operations need to be re-mapped

Logic for changing the signals above is not complex since they are all two-to-one mappings and the block diagram of the unit is shown in figure 3-20.

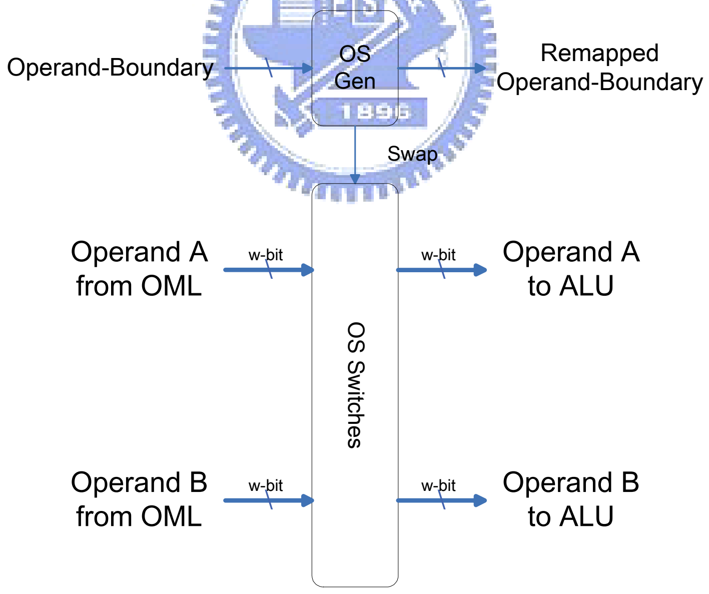


Figure 3-20 Block Diagram of the Operation-Swapping Logic

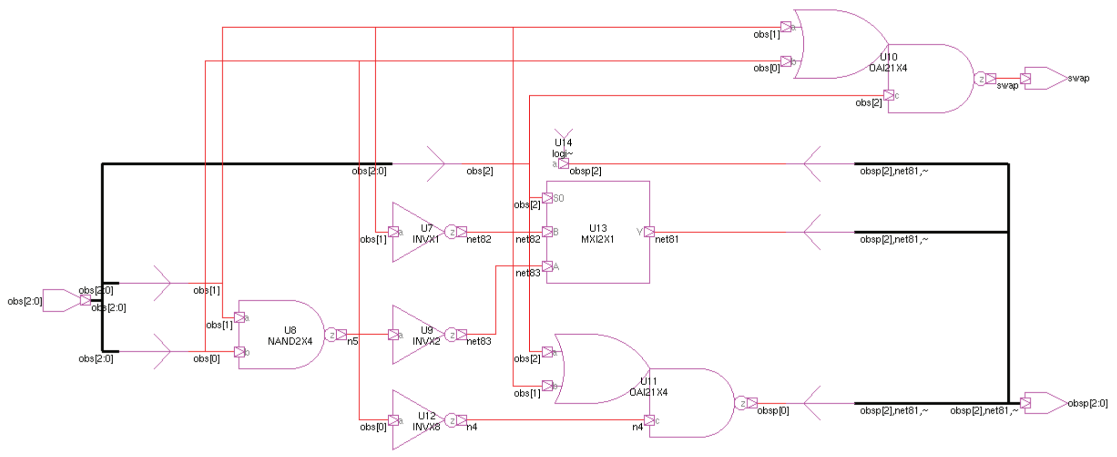


Figure 3-21 Schematic of OS Gen Logic

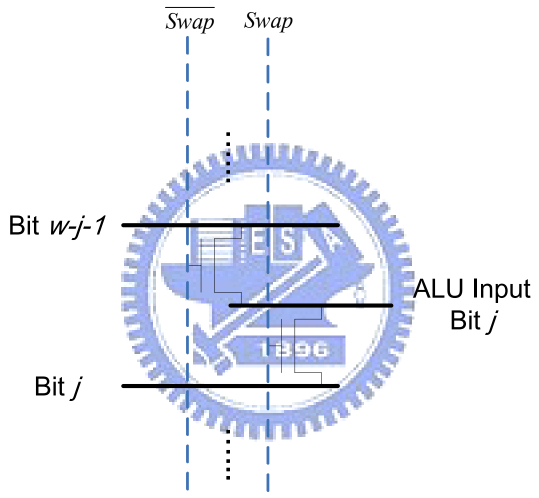


Figure 3-22 Schematic of the OS Switches

Although the operation-swapping mechanism reduces the number of ALU blocks, the lines that the operand-boundary requires might not be reduced. If we divide the operands into m blocks with the same size, $\lceil \log_2 m \rceil$ lines are required. By adopting the operation-swapping mechanism so that half-width of the ALU is implemented as a block, the conditions become to $\frac{m}{2}+1$. The number of required lines is $\left\lceil \log_2\left(\frac{m}{2}+1\right)\right\rceil$ and thus it may not decrease.

3.3.3.2. Widening the ALU

The total number of ALU blocks is a constraint for joining another operation. From our profiling on bit-width distribution, in a single pipeline machine, some ALU operations are used to calculate effective addresses by load / store instructions. It raises the bit-width distribution so that about half of ALU operations requires less than or equal to 19-bit. We can relax the limitation by appending extra ALU blocks. As for how wide the ALU should be extended, we performed some performance simulations. Based on the result of simulation, a recommended ALU width considering the increased ratio of performance improvement over increased ALU hardware is proposed.

The widened part of ALU won't increase the maximal delay of the ALU since the longest delay still occurs when only one operation is using the ALU. Besides, the extended ALU blocks could be implemented as a single block. And widening the ALU won't increase the number of conditions that the operand-boundary signals should be able to represent.

3.4. Modifications in Memory Access and Write-back Stage

Results of the two joined operations are also joined together on the result bus. Before any units, such as register-file or memory, requires the result, it should be sign-extended to full-width results. How it is sign-extended is determined by the most significant bit in its most significant block.

3.4.1. Sign-extending the Joined Results

To sign-extend the joined results to full-width results, the sign-extending logic must know

i. The correct sign-bit for extending

ii. Which block should be filled with sign bits and which block should simply be bypassed

Each result block which might contain significant bits or should be sign-extended is connected to an Sign-Extending Block (SEB), which is controlled by a pass signal and sign-bit signal. When the pass signal is set, the SEB passes the block. When the pass signal is not set, the SEB fills the output with the sign-bit. The schematic of a sign-extending logic is shown in figure 3-23.

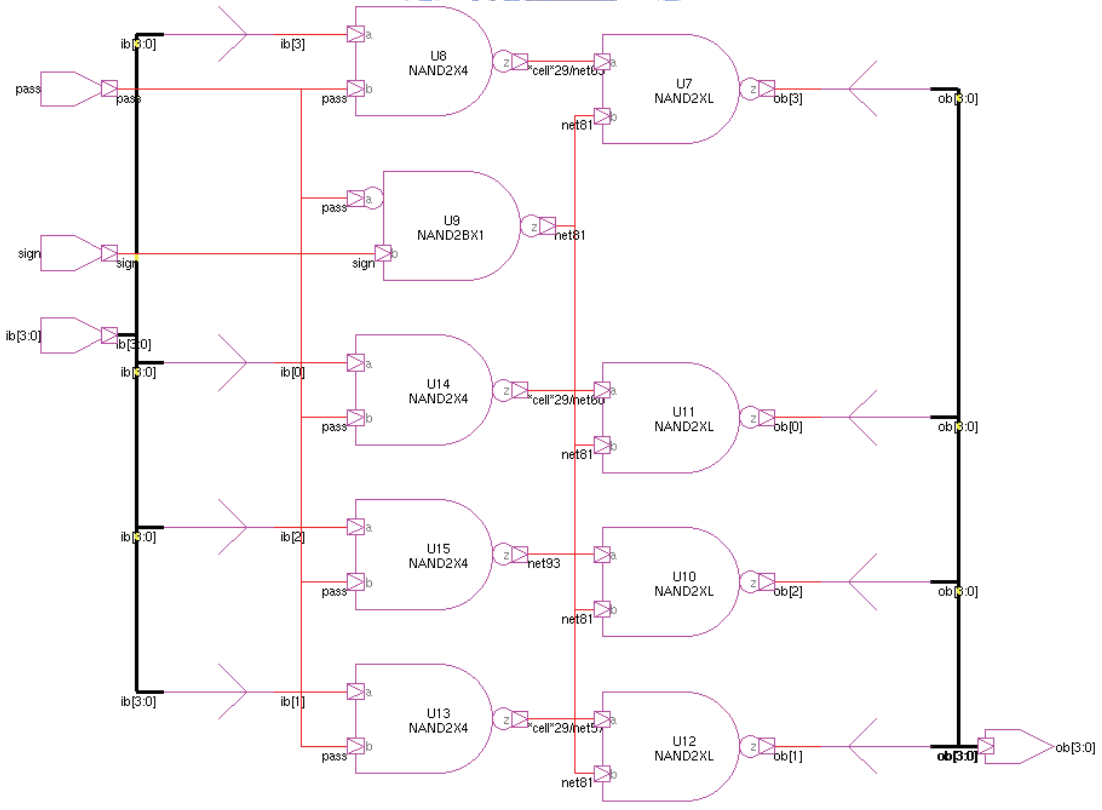


Figure 3-23 Schematic of Sign-Extending Block

The sign-bit could be decided from the most significant bit in the most significant block. The logic for select sign-bit according to operand-boundary signals and most significant bits in possible blocks is shown in figure 3-24.

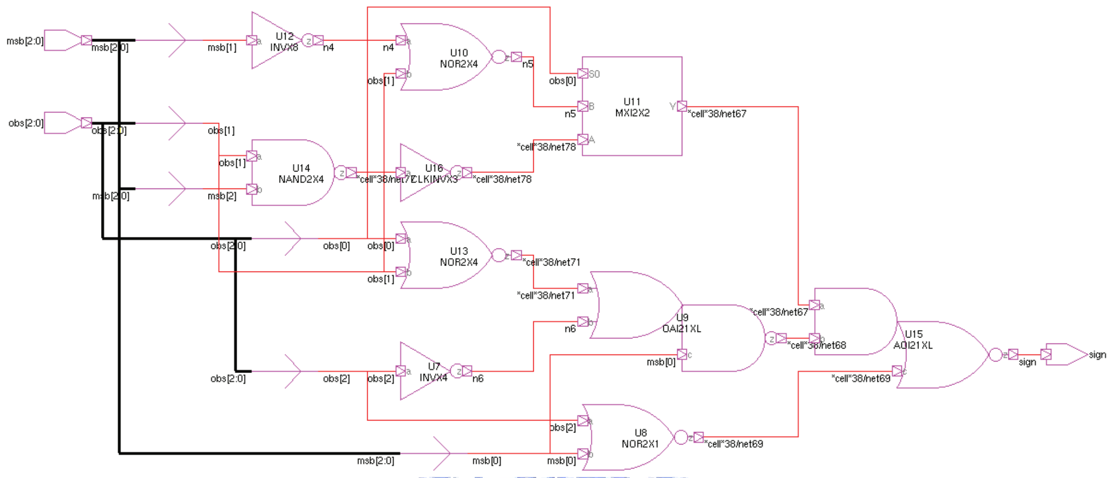


Figure 3-24 Schematic of Sign-bit Generating Logic

The logic for generating pass signals according to operand-boundary signals to each sign-extending block is shown in figure 3-25.

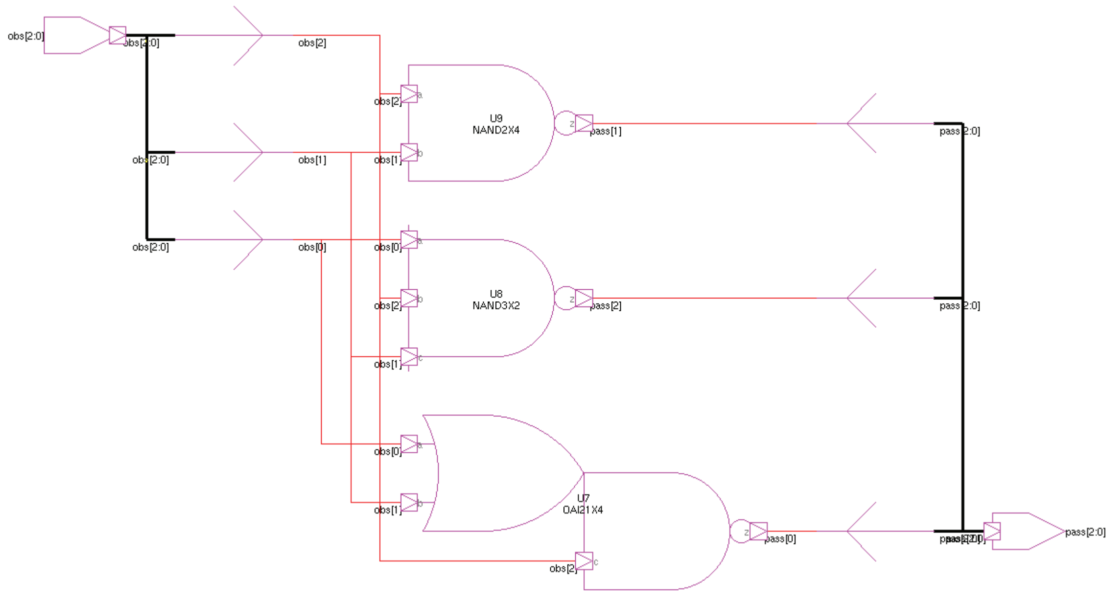


Figure 3-25 Schematic of Pass Generating Logic

3.5. Integrating the Design into a Five-stage MIPS-like Pipeline Datapath

Where the mechanism should be integrated into the pipeline datapath is discussed in previous sub-sections, in figure 3-26, we show the possible organization.

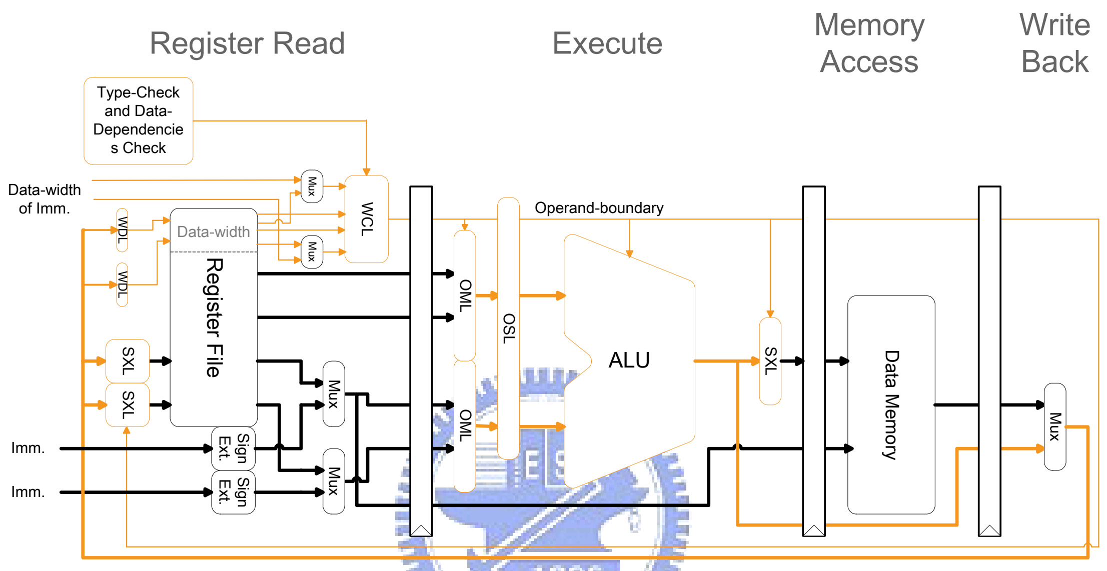


Figure 3-26 A possible pipeline datapath design

The instruction-fetch is not shown due to it has no significant difference with a dual-issue machine. In such organization, extra fields in register-file are used to store the numbers of significant blocks of values. Two instructions are firstly performed with type-check and data-dependency check and pass the result to WCL (width-check logic). The WCL also gets the information of numbers of significant blocks from register-file and that of the immediate value. Then it generates the operand-boundary signals. The operand-boundary signals control the OML (operand-merging logic), ALU, and SXL (sign-extending logic). Finally, when the results are written into register-file, the WDL (width-determination logic) calculate the significant blocks of values and write them into register-file.

But delay by the width-check logic may still be too long to be fit in the latter-half cycle of ID stage. For this concern, we could put the WDL, WCL, OML, and OSL into a dedicated stage.

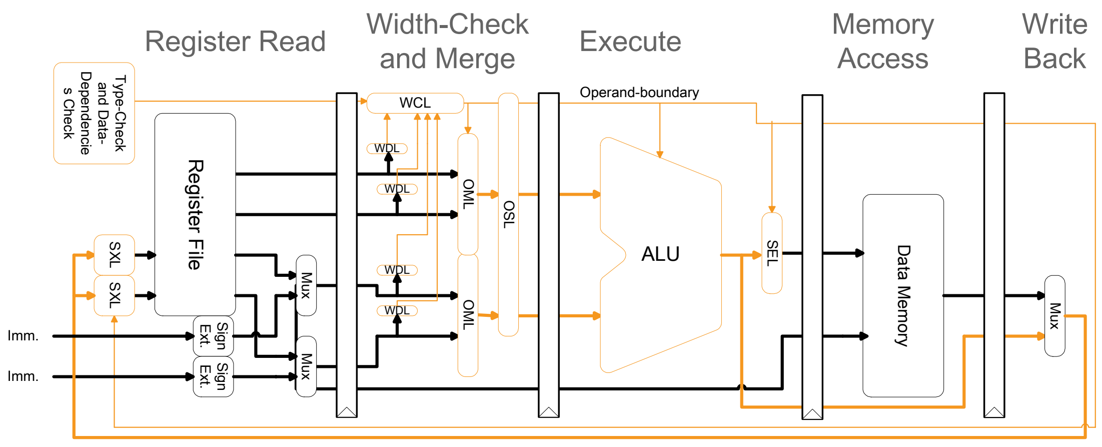


Figure 3-27 A possible pipeline datapath design with six-stage (the IF stage is not shown)

Putting the WDL and WCL together has another advantage – the logic for performing width-determination and width-check could be simplified and the delay could be shorter. Figure 3-28 shows the block diagram of simplified logic for width-determination and width-check.

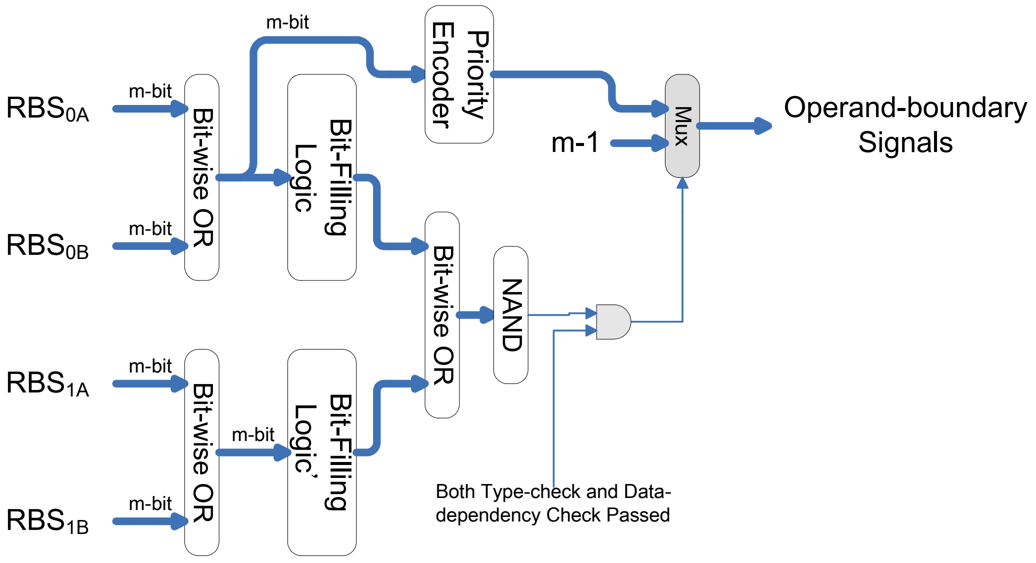


Figure 3-28 Simplified width-check logic in the six-stage design

Assuming the operands are divided into *m* blocks. The RBS signals are generated by RB generator in the width-determination logic. In the RBS signals, bits corresponding to block that might be required are set. By performing the bit-wise OR operation between RBSs of the two operands of an operation, the result bits could represent which block are actually required by the operation.

The result bits output from the first bit-wise logic could be 0…01XXXX where the leading 0's comes from the insignificant part and the 1XXXX comes from the significant part. If we could change the value into 0…011111 and perform bit-wise OR operation with the result bits of the other operation with *turnaround bit-ordering*, we're able to recognize which block is actually occupied by any of the operations.

The bit-filling logic is used to set the bits which belong to significant parts. Its input and output relationship is in table 3-5.

Input RB Signals					Output bits				
b_{m-1}	b_{m-2}	…	b_1	b_0	b_{m-1}	b_{m-2}	…	b_1	b_0
1	X	…	X	X	1	1	…	1	1
0	1	…	X	X	0	1	…	1	1
……					…				
0	0	…	1	X	0	0	…	1	1
0	0	…	0	1	0	0	…	0	1

Table 3-5 Function of bit-filling logic

But the bit-filling logic for the two operations cannot be identical. If they are identical, for two operations both requires $\frac{m}{2}$ blocks, the result bits by bit-wise OR will all be set and regarded as that the width-check failed. So the mapping by the bit-filling logic used for the other operation must differ by shifting the result one bit so that when the

Input RB Signals					Output bits				
b_{m-1}	b_{m-2}	…	b_1	b_0	b_{m-1}	b_{m-2}	…	b_1	b_0
1	X	…	X	X	*0*	1	…	1	1
0	1	…	X	X	*0*	0	…	1	1
……					…				
0	0	…	1	X	*0*	0	…	0	1
0	0	…	0	1	*0*	0	…	0	0

Table 3-6 Function of bit-filling logic'

By the following function we could know whether these two operations can pass the width-check.

$$pass = \overline{\left(b_{m-1}^{1} + b_{0}^{2}\right) \bullet \left(b_{m-2}^{1} + b_{1}^{2}\right) \bullet \cdots \bullet \left(b_{0}^{1} + b_{m-1}^{2}\right)}$$

b_j^i means block j of operation i

The area and delay by the width-determination logic and width-check logic can be reduced. Besides, if the two operations can be joined, the decoded operand-boundary signals for controlling OML are the inverted bits from the bit-filling logic!

Chapter 4. Experiments

4.1. Goals of Our Experiments

In the experiments, we'll compare two different operand-merging mechanisms, one is turnaround-based and the other is shifting-based, from area and circuit delay.

We also wish to choose parameters for designing the ALU. We'll find the suitable ALU block width by considering both ALU delay and the utilization of ALU. Simulation for execution-time reduction with different ALU block width is also performed.

Another ALU design issue is how width the ALU should be? Since increasing the ALU width could improve the opportunity of joining two operations together, we observe the effects on wider ALU width over ratio of increased ALU bits to make the conclusion.

4.2. Simulation Environment

Synthesis Environment and Constraints

Tool: Synopsis Design Compiler

Technology:.18um

Operating voltage: 1.62V

Working frequency: 166Mhz

ALU Implementation:

Provides ADD / AND / OR / XOR.

Adder is implemented as carry-lookahead adder blocks with ripple carry-propagating.

Software Simulation Environment

ISA: SPARC v8

Assuming no cache misses

Benchmark: MiBench Suite

Simulation Methodology:

Generate instruction-trace with significant bit-width information from ArchC Simulator. Statistics about execution cycles and number of joinable operations is gathered from traces.

4.3. Comparison between Different Operand-Merging Schemes

Based on different ALU-sharing schemes, the turnaround-based and traditional shift-based, the circuits for merging operands differs since the shift-based approach has the requirement for shifting the joined operands to correct position while the turnaround-based approach does not.

Although in our design, the circuit for merging the operand bits is implemented in pass-transistor logic, it cannot be estimated in cell-based design. This experiment is implemented with logic gates so that the results only show the trends of the difference, *NOT the actual areas of the OML.*

Figure 4-1 shows the area requirements under different block widths with the two approaches.

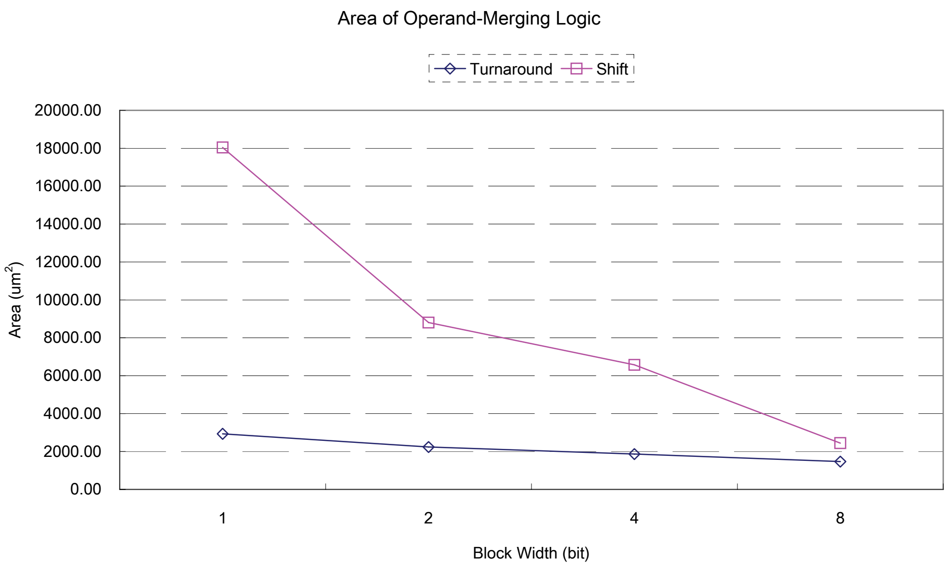


Figure 4-1 Area overhead of circuits to merge operands

As we can see, the turnaround-approach has quite little area overhead over the shift-based scheme. As the block width goes finer, the difference in area overhead becomes grater.

Figure 4-2 shows the delay of the operand-merging logic under the two approaches.

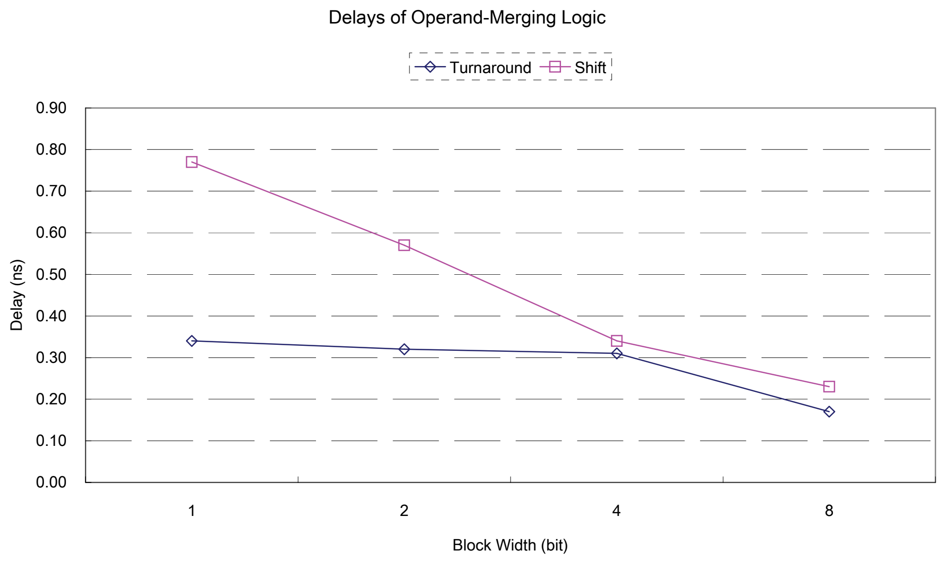


Figure 4-2 Delay of circuits to merge operands

Turnaround-based approach also has advantage on delay over the shift-based approach as granularity goes finer. From figure xxx, the delay of turnaround-based approach has only little improvement over the shift-based approach. This is because the major delay comes from the decoder logic for controlling switches when the block-width is wider than 4-bit, i.e. eight blocks.

Figure 4-3 shows the area and delay reduction by turnaround-based approach over the shift-based version.

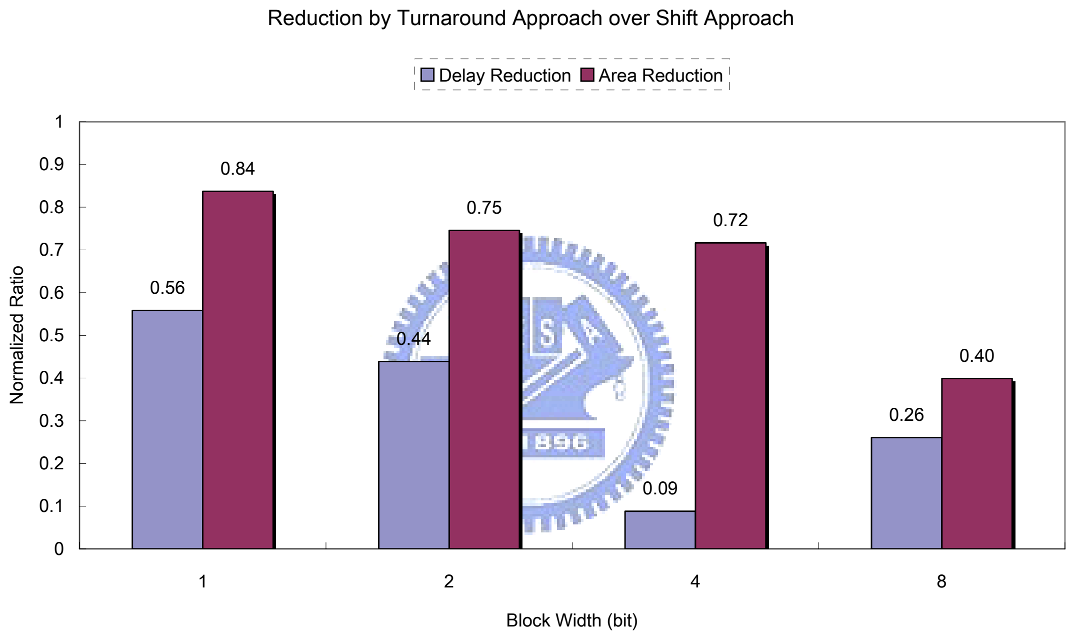


Figure 4-3 Reductions of Turnaround approach over shift approach

4.4. Hardware Cost of Operation-Swapping Mechanism

In section 3.3.3.1, we proposed the operation-swapping mechanism which could avoid unnecessary ALU partitioning. An ALU block whose width is half the datapath width could be adopted in the design to shorten delay on ALU. In this section, we examine the overhead by circuits to swap the operands and see why it is not suitable for shift-based approach.

Figure 4-4 shows the delay of operation-swapping mechanism with different ALU block widths. We also shows the delay when we adopt such mechanism on shift-based approach.

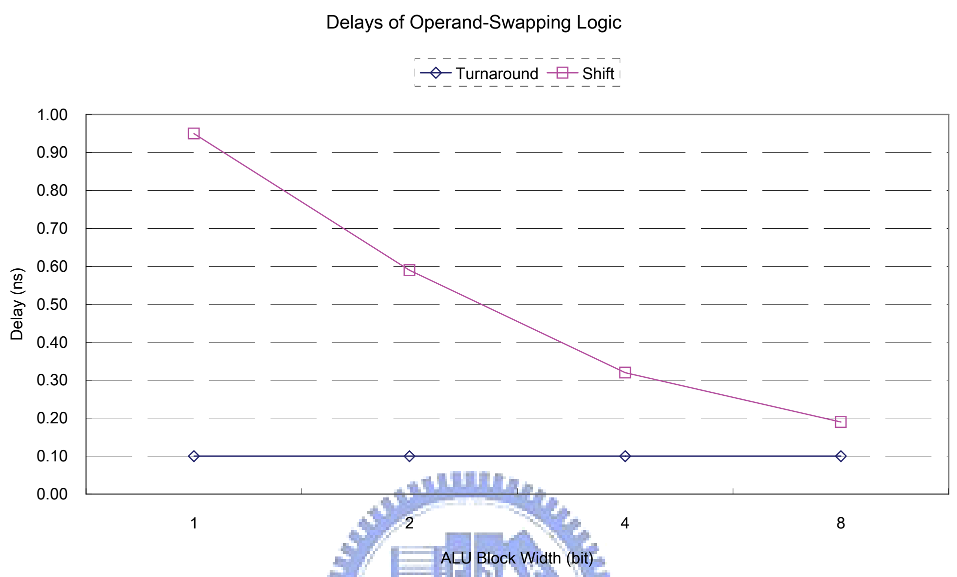


Figure 4-4 Delay of Operand-Swapping Logic

The delay in turnaround-based approach is constant since the mapping between input and output is fixed. But in the shift-based approach, the delay is much longer since the mapping between inputs to outputs could be a many-to-one mapping, i.e. operand blocks has to be re-aligned to correct block positions.

Figure 4-5 shows the area overhead of operation-swapping logic[6].

[6] The OSL in this experiment is implemented with logic gates. The results are NOT the actual areas of the OSL.

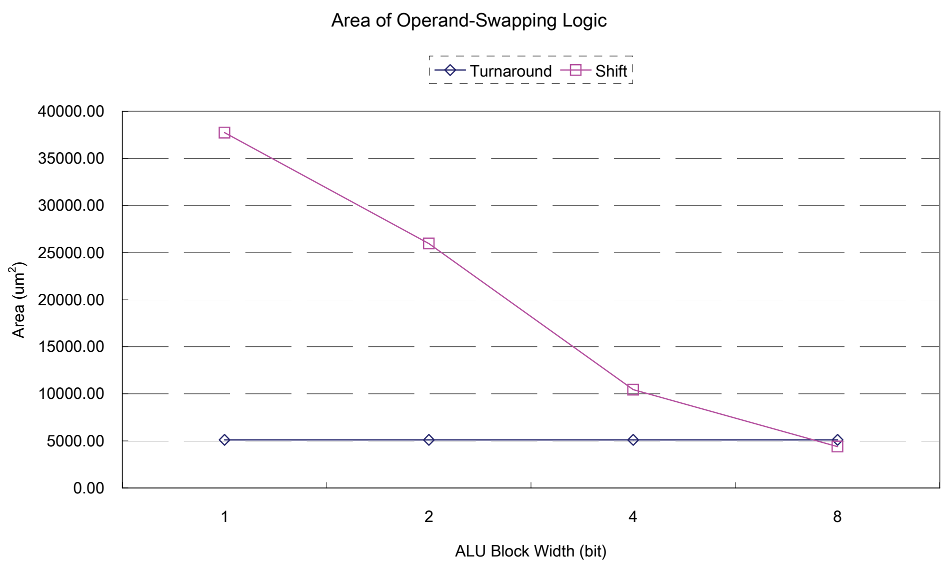


Figure 4-5 Area of Operand-Swapping Logic

For the same reason we used to discuss about the delay, the requirements of area for the two approaches have the same trend.

From the observation above, we can see under all width configures, the logic for turnaround-based approach is constant. But for the shift-based approach, it requires quite large area to re-align the operand-blocks. This is why it's not suitable for the shift-based approach.

4.5. Choosing ALU Block Width

An ALU with smaller ALU block width makes the sharing more flexible and is capable of high bit-utilization in the optimal case – all blocks input to the ALU are significant blocks from the two operations. But not every bit in significant blocks is actually significant. There might be several bits belonging to insignificant part. The circuits for processing these bits are not really utilized. By making the block width smaller, the number of really utilized bits is increased in the optimal case. But make the ALU block smaller would also increase the delay of the ALU circuit.

In this section we'll choose a suitable block width for the consideration of both circuit delay and the capability of bit-utilization of the ALU in optimal case. Under uniform distribution, for the block width n, the insignificant bits in the most significant block of an operation is $\frac{n}{2}$. For a m-bit ALU, the average utilized bits in optimal case is $m-\frac{n}{2}-\frac{n}{2}$. We choose a suitable block width by the ratio of circuit delay over the average utilized bits in optimal case. Figure xxx shows the result.

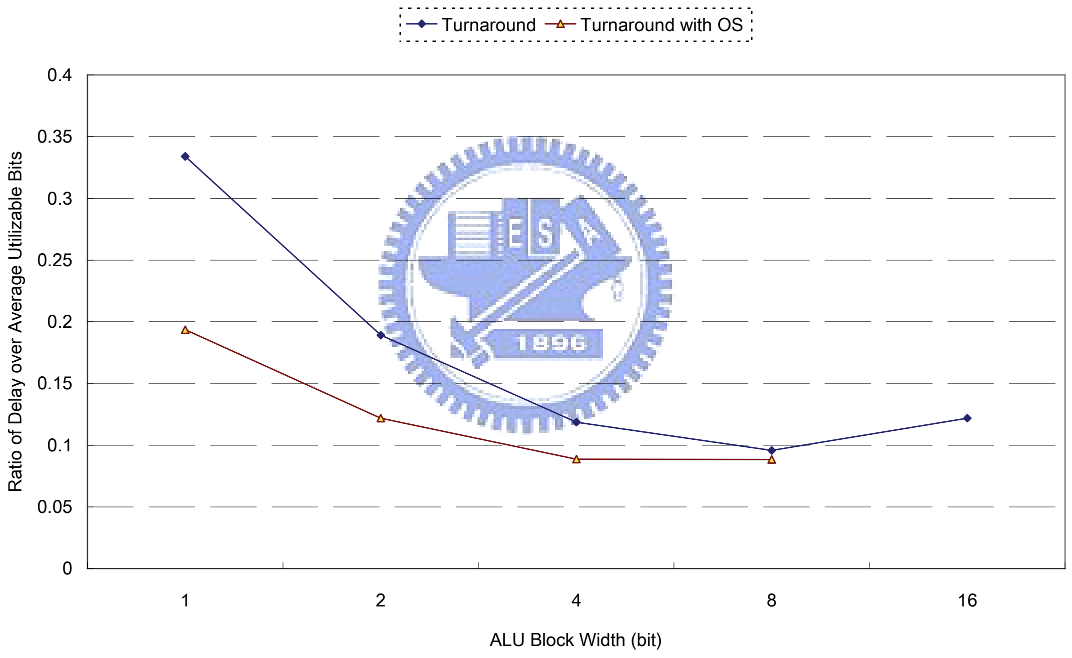


Figure 4-6 Ratio of delay over averaged utilized bits in optimal case

As we can see, without the operation-swapping mechanism, ALU block width of 8-bit has the lowest ratio. But by adopting the operation-swapping mechanism, block width of 4-bit and 8-bit has similar ratios.

Widths of ALU blocks affect the combination flexibility of different operand-widths. We replayed the instruction traces to see how this factor affects the number of joined ALU operations.

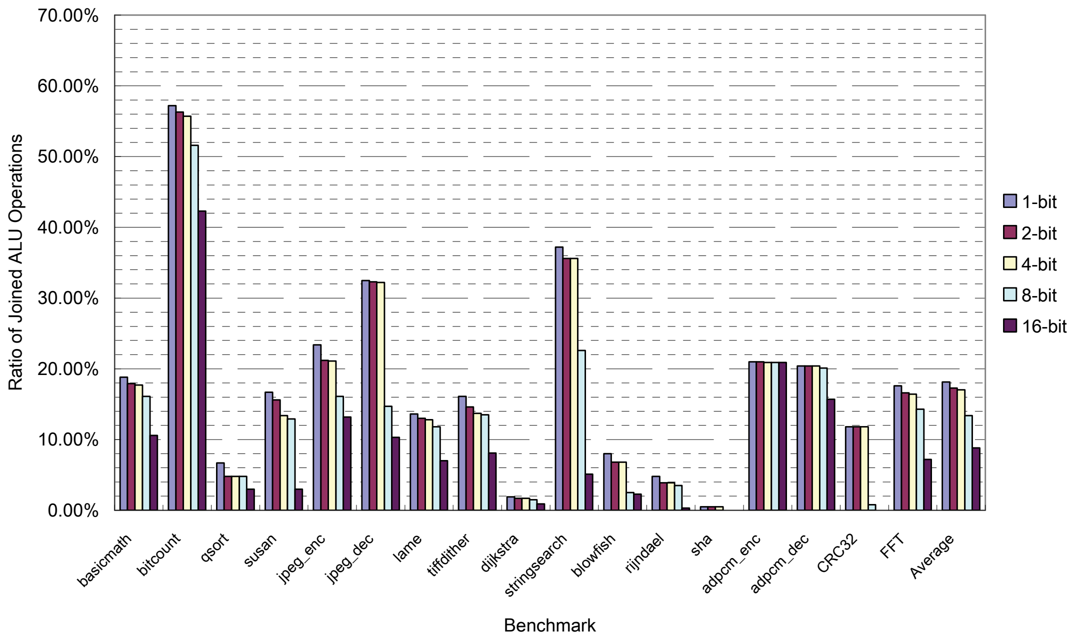


Figure 4-7 Ratio of joined ALU operations with different ALU widths

As we expected, ALU block width of 1-bit has most and ALU block width of 2-bit has similar ratio with 4-bit configuration. 8-bit and 16-bit configurations have apparently lower ratios than configurations of finer granularities. Taking the worst case delay of the ALU into account, the ratio of ALU delay over joined ALU operations is shown in figure 4-8.

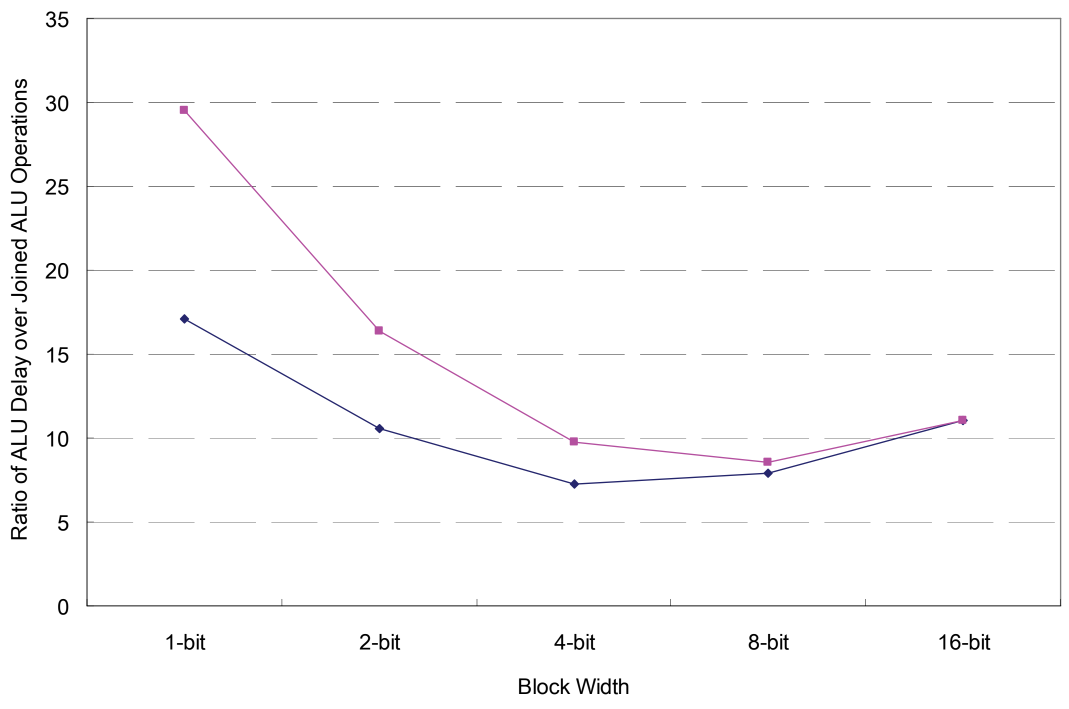


Figure 4-8 Ratio of ALU Delay over Joined ALU Operations

The points consisting the lower lines is the design with operation-swapping mechanism while the higher one is the design without operation-swapping design so that its ALU is uniformly divided into ALU blocks. From the figure, the block width of 4-bit is a good choice.

4.6. Choosing ALU Width

Performance could be increased if we increase the ALU width since the opportunities of joining two operations together would be increased. We performed experiments from ALU of 32-bit width to 64-bit with 4-bit step.

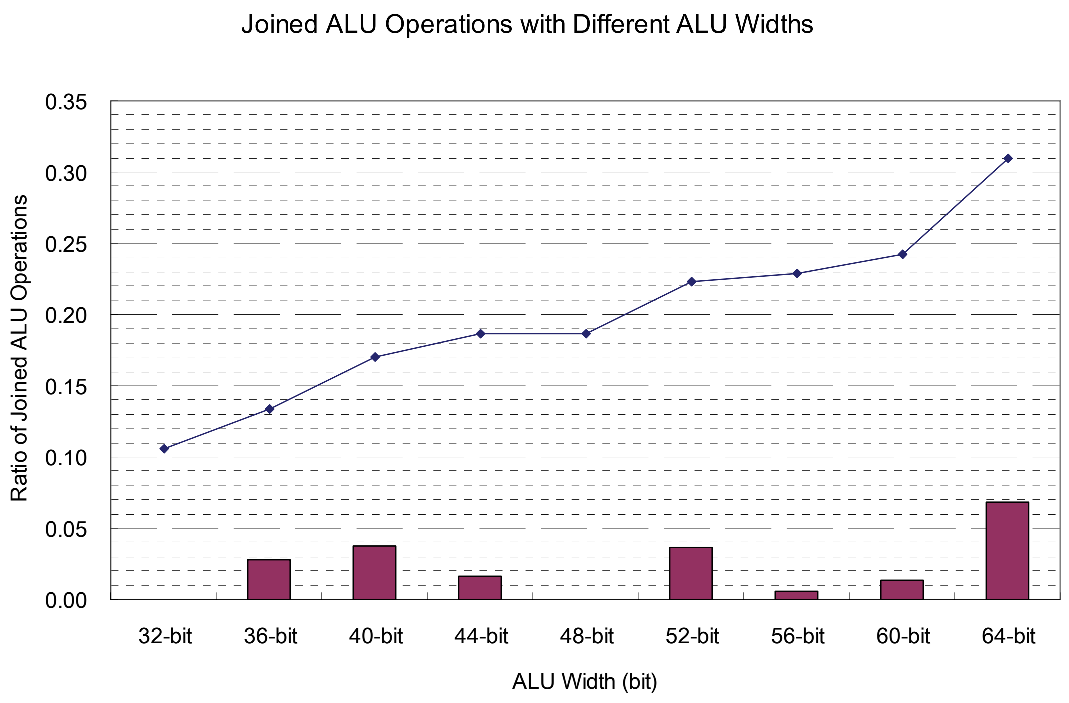


Figure 4-9 Ratio of Joined ALU Operations with Different ALU Widths

The histogram in figure 4-7 is the increment compared with the configuration which is 4-bit narrower than it. When the widths are 40-bit, 52-bit, and 64-bit, the number of increased ratio is larger. And in the intervals between 40-bit~48-bit and 52-bit~60-bit, the increases are quite small (less than 0.2). If we extend the ALU, the source operand and results buses are also widened. Considering the ratio of cost, in our design, we widen the ALU into 40-bit.

4.7. Final Proposal of the ALU Design

From the experiment results, we have several conclusions on the proposed design.

i. Operand-merging Mechanism

Bus design is related to the arrangement of ALU blocks. In this work, we proposed the block-reversed ordering scheme and compared it with the regular ordering scheme. Considering the hardware complexity and timing from the results in section 4.3, we suggest that the block-reversed ordering scheme is better.

ii. ALU Block Width

From the perspective of ALU hardware implementation, block width of 4-bit or 8-bit is commonly used as a basic ALU block due to the adder function. And from the results in section xxx, in most cases and on average the ratios of joined operations for block width of 4-bit configuration are similar to those of 1-bit and 2-bit configurations. And 4-bit configuration has apparently better ratio than 8-bit and 16-bit configurations.

iii. ALU Width

From the experiment in section 4.6, 40-bit is a good choice.

iv. Operation Swapping

This technique we proposed in section xx could improve the delay incurred by the additional multiplexers between the ALU blocks and control lines. It also simplifies the design of the circuits for sign-extending results.

Chapter 5. Conclusion and Future Work

5.1. The Turnaround-based Sharing Mechanism

The turnaround-based sharing mechanism provides high flexibility with little hardware overhead. Due to its characteristic that no alignment is required, quite small overhead in both area and delay is incurred. Compared with the shifting approach with the same flexibility, as the flexibility increases, the reduction becomes more significant.

A prerequisite for sharing a unit with the turnaround-based approach is that the unit could perform the joined operation with its operand blocks in turnaround style. How much effort is required for a unit to support such capability depends on its logic characteristics. In this thesis, we partitioned the ALU into blocks. To make the ALU able to process the joined operation, the major modification is the re-organize the carry-chains. Such modifications have no significant overhead on area. For a unit which satisfies the prerequisite, the turnaround-based sharing mechanism could reveal high utilization of the chip area.

5.2. Reducing Additional Delay by Avoiding Unnecessary Partitioning

To reduce additional delay by avoiding unnecessary partitioning is a technique which benefits from the operation-swapping mechanism. Assuming the ALU is not widened, we may improve the delay by *not* partitioning the higher order ALU blocks. For example, considering a *m-bit* ALU with the minimal block size *n-bit*, the widths of ALU blocks could be *2n, n, ..., n*, and $\frac{m}{2}$ while $2n + n + ... + n + \frac{m}{2} = m$.

This is a trade-off between flexibility and circuit delay. For this example, two operations, one requires *n-bit* and the other requires (*m-n)-bit*, cannot be joined. The optimal partitioning could be decided with statistical methods.

5.3. Applying the Design to Architectures with Shifter Concatenated with ALU

On some architecture the shifter is concatenated with ALU, for example, the ARM architecture. Unlike partitioning an ALU, the shifter could not easily be partitioned into two parts to process two operations with different shift amounts. To achieve the goal, lots of internal bypass logic and wires are required. Furthermore, considering rotation operation, circuits become much more complex when the input operands are merged from two operations.

For such architectures, we classified the joined operation pairs by requirement of shifter:

i. Both don't perform shifting

Our datapath-sharing mechanism works fine.

ii. One performs shifting and the other does not

We need to add extra bus lines to bypass the shifter for operand bits.

iii. Both performs shifting

Due to our observation, to modify the shifter architecture to support two shift operations with different amounts is more complex than ALU-sharing. When both operations requires shifting, we could either leave it as a non-exploitable combinations or place the shifter which is also designed to be flexibly shared two operations.

Besides, the width-check logic must also consider the shift amount and direction of the joined operations so that the operand-boundary signals could be correct.

5.4. Clock Gating to Freeze Unused Blocks

In previous researches, the abundance of narrow-operand operations is often used for clock-gating. With the number of significant blocks, unused ALU blocks could be gated and easily integrated in this design by adding extra latches. In other hand, our design could be integrated into such design without too much overhead. Logic which has similar function with WDL and SXL should already exist in such architecture. The major cost comes from the WCL and it amounts to about half an ALU.

5.5. Future Work

Efficient partitioning mechanisms for shifters and multipliers could make more types of operations joinable.

Although the significant bit-widths of operations must be determined run-time, previous research shows this characteristic has spatial locality. If we could profile the execution behaviors of a program and records the bit-width of each operation, this information could assist compiler to schedule the program codes.

The dynamically scheduling mechanism considering the significant bit-widths information in a superscalar architecture is also a topic to efficiently utilize this design.

References

〔1〕 Gabriel H. Loh – Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth, Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35)

〔2〕 R Sheen, S Wang, OTC Chen, Ruey-Liang Ma – Power consumption of a 2's complement adder minimized by effective dynamic data ranges, Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, 1999. ISCAS '99

〔3〕 D Brooks, M Martonosi – Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, Proceedings. Fifth International Symposium On High-Performance Computer Architecture, 1999

〔4〕 Steve Furber, ARM System-on-Chip Architecture, 2nd Edition

〔5〕 Wayne Wolf, Modern VLSI Design – System-on-Chip Design

〔6〕 John L. Hennessy and David A. Patterson, Computer Architecture – A Quantitative Approach 3rd Edition

〔7〕 LEON2 Processor, http://www.gaisler.com/products/leon2/leon.html