國立交通大學

資訊科學與工程研究所

碩 士 論 文

同步多執行緒架構中可彈性切割與可延展的暫存器檔案設計之研究

Design of a Flexibly Splittable and Stretchable Register File for SMT Architectures

研 究 生：鐘立傑

指導教授：單智君　教授

中 華 民 國 九 十 六 年 八 月

國 立 交 通 大 學

博碩士論文全文電子檔著作權授權書

（提供授權人裝訂於紙本論文書名頁之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程研究所 系統設計 組， 95 學年度第 二 學期取得碩士學位之論文。

論文題目：同步多執行緒架構中可彈性切割與可延展的暫存器檔案設計之研究
指導教授：單智君

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：

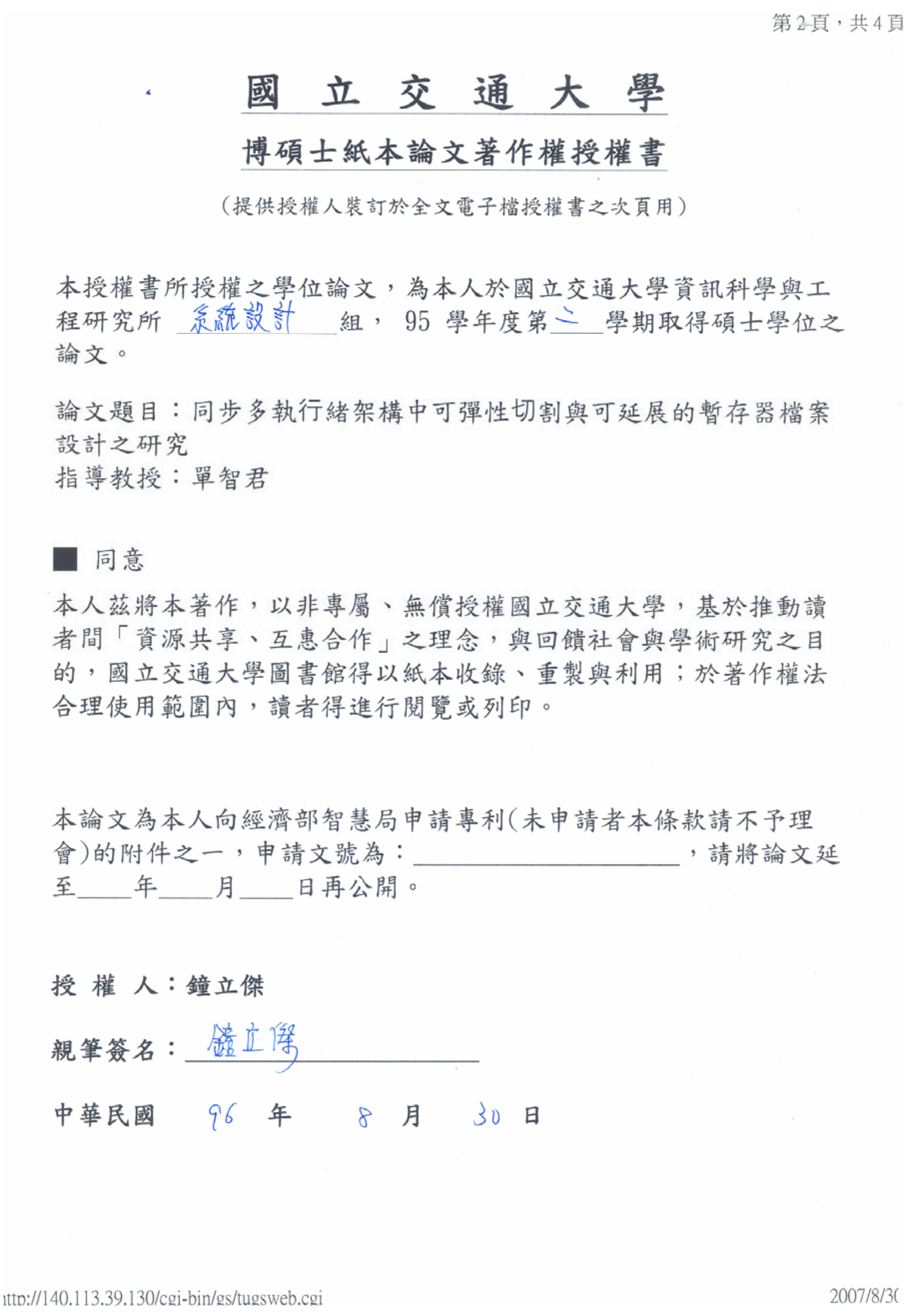
本校及台灣聯合大學系統區域網路	■ 立即公開
校外網際網路	■ 立即公開

■ 全文電子檔送交國家圖書館

授 權 人：鍾立傑

親筆簽名：鍾立傑

中華民國 96 年 8 月 30 日



國立交通大學

博碩士紙本論文著作權授權書

（提供授權人裝訂於全文電子檔授權書之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學資訊科學與工程研究所 系統設計 組， 95 學年度第 二 學期取得碩士學位之論文。

論文題目：同步多執行緒架構中可彈性切割與可延展的暫存器檔案設計之研究
指導教授：單智君

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

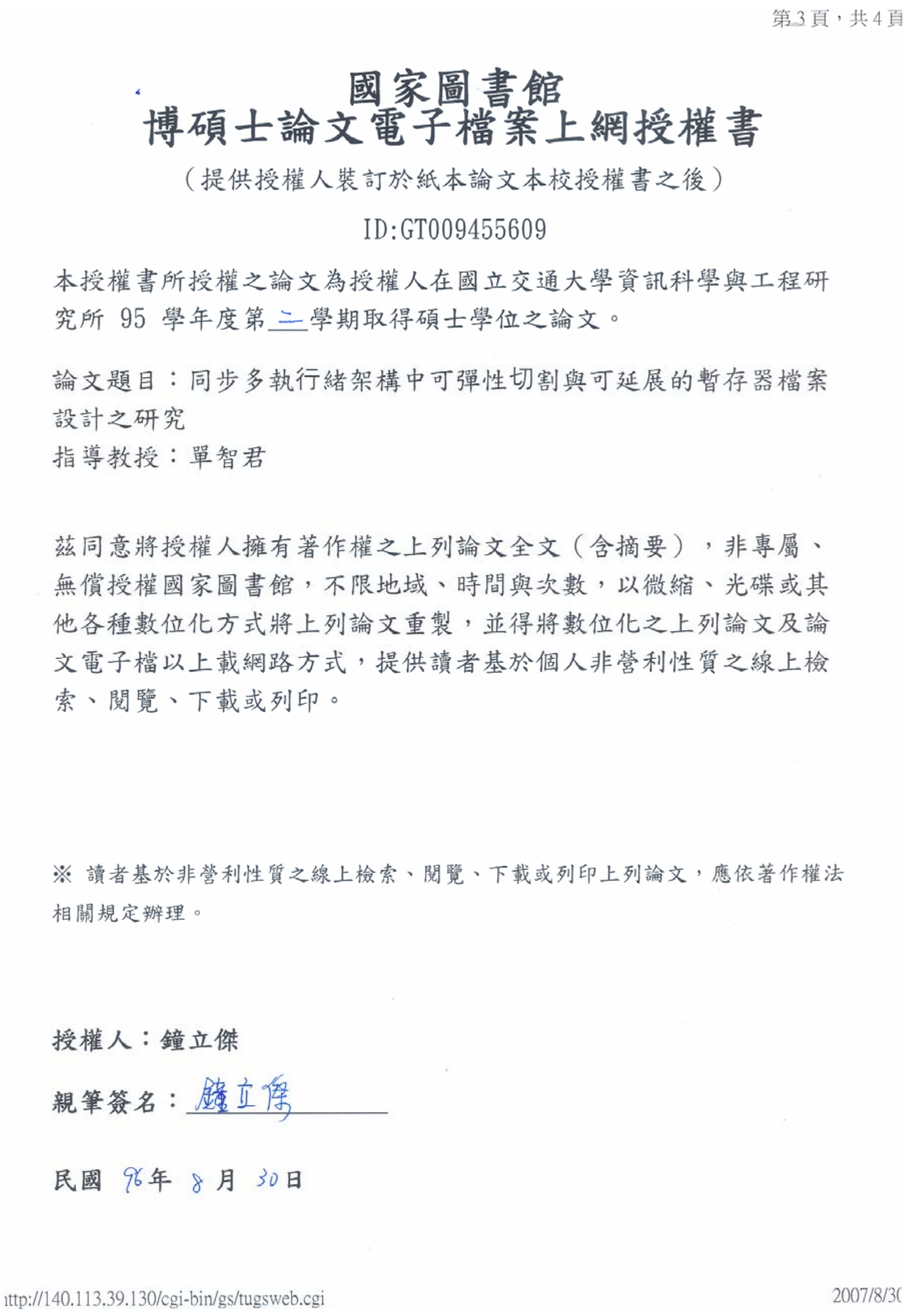
本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：＿＿＿＿＿＿＿＿，請將論文延至＿＿年＿＿月＿＿日再公開。

授 權 人：鐘立傑

親筆簽名： 鐘立傑

中華民國 96 **年** 8 **月** 30 **日**

http://140.113.39.130/cgi-bin/gs/tugsweb.cgi 2007/8/30



第3頁，共4頁

國家圖書館
博碩士論文電子檔案上網授權書

（提供授權人裝訂於紙本論文本校授權書之後）

ID:GT009455609

本授權書所授權之論文為授權人在國立交通大學資訊科學與工程研究所 95 學年度第二學期取得碩士學位之論文。

論文題目：同步多執行緒架構中可彈性切割與可延展的暫存器檔案設計之研究
指導教授：單智君

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授權人：鐘立傑

親筆簽名： 鐘立傑

民國 96 **年** 8 **月** 30 **日**

ttp://140.113.39.130/cgi-bin/gs/tugsweb.cgi 2007/8/3

國立交通大學
研究所碩士班

論文口試委員會審定書

本校 資訊科學與工程 研究所 鐘立傑 君

所提論文:

同步多執行緒架構中可彈性切割與可延展的暫存器檔案設計之研究

Design of a Flexibly Splittable and Stretchable Register File for SMT Architectures

合於碩士資格水準、業經本委員會評審認可。

口試委員: 邱日清 單智君

鍾崇斌 陳青文

指導教授: 單智君

所　　長: 曾文貴

中 華 民 國 九十六 年 七 月 二十三 日

同步多執行緒架構中可彈性切割與可延展的暫存器檔案設計之研究

學生：鐘立傑　　　　　　　　　　　　　　　　指導教授：單智君 博士

國立交通大學資訊科學與工程研究所 碩士班

摘　要

如何利用最少的硬體資源來支援同步多執行緒是一個很重要的研究議題，暫存器檔案(Register file)在微處理器晶片面積中佔有顯著的比例。而且為了支援同步多執行緒，每一個執行緒享有自己的一份暫存器檔案，這樣的設計會增加晶片的面積。

在本篇論文中，我們提出了一份可彈性切割與可延展的暫存器檔案設計，在這個設計裡：1.我們可以在需要的時候彈性切割一份暫存器檔案給兩個執行緒來同時使用，2. 適當的延伸暫存器檔案的大小來增加兩個執行緒共用的機會。

藉由我們設計可以得到的益處有：1.增加硬體資源的使用率，2. 減少對於記憶體的存取以及 3.提升系統的效能。此外我們設計概念可以任意的滿足不同的應用程式的需求。

Design of a Flexibly Splittable and Stretchable Register File for SMT Architectures

Student：Li-Jie Jhing Advisor：Dr, Jean Jyh-Jiun Shann

Institute of Computer Science and Engineering

National Chiao-Tung University

Abstract

How to support simultaneous multithreading (SMT) with minimum resource hence becomes a critical research issue. The register file in a microprocessor typically occupies a significant portion of the chip area, and in order to support SMT, each thread will have a copy of register file. That will increase the area overhead.

In this thesis, we propose a register file design techniques that can 1. Split a copy of physical register file flexibly into two independent register sets when required, simultaneously operable for two independent threads. 2. Stretch the size of the physical register file arbitrarily, to increase probability of sharing by two threads.

Benefits of these designs are: 1. Increased hardware resource utilization. 2. Reduced memory traffic amount. 3. Increased system performance. Moreover, these proposed designs can be arbitrarily mixed as per application need.

誌謝

首先誠摯的感謝指導教授 單智君教授，在老師悉心的教導使我得以一窺資訊工程領域的深奧，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。老師對學問的嚴謹更是我輩學習的典範。同時感謝我的的另一位參與計劃老師兼口試委員 鍾崇斌教授，口試委員 邱日清以及 陳青文教授，由於他們的指導與建議，讓這篇論文更加完整和確實。

兩年裡的日子，實驗室裡共同的生活點滴，學術上的討論、言不及義的閒扯、讓人又愛又怕的宵夜、趕作業的革命情感、因為睡太晚而遮遮掩掩閃進實驗室........，感謝眾位學長姐、同學、學弟妹的共同砥礪(墮落?)，你/妳們的陪伴讓兩年的研究生活變得絢麗多彩。

感謝喬偉豪、蔣昆成、王志男、吳奕緯、翁綜禧、李元化學長、楊惠親學姐們不厭其煩的指出我研究中的缺失，且總能在我迷惘時為我解惑，也感謝康哲瑋、張辰瑋、林慧榛、陳志遠、吳易叡、張延義同學的幫忙，恭喜我們順利走過這兩年。實驗室的黃勁霖、郭員榕、張順傑、張柏駿、邱冠穎學弟們當然也不能忘記，的幫忙及搞笑我銘感在心。

女朋友巧菱在背後的默默支持更是我前進的動力，沒有巧菱的體諒、包容，相信這兩年的生活將是很不一樣的光景。

最後，謹以此文獻給我摯愛的雙親。

鐘立傑

2007. 8. 30

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Overviews of Processor Register and Register File Design in Modern Microarchitecture

1.1.1. Processor register

In computer architecture, a processor register is a small amount of very fast computer memory used to speed the execution of computer programs by providing quick access to commonly used values—typically, the values being calculated at a given point in time. Most, but not all, modern computer architectures operate on the principle of moving data from main memory into registers, operating on them, then moving the result back into main memory—a so-called load-store architecture.

Processor registers are the top of the memory hierarchy, and provide the fastest way for the system to access data. The term is often used to refer only to the group of registers that can be directly indexed for input or output of an instruction, as defined by the instruction set. More properly, these are called the "architectural registers". For instance, the x86 instruction set defines a set of eight 32-bit registers, but a CPU that implements the x86 instruction set will contain many more registers than just these eight.

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". Registers are now usually implemented as a register file, but they have also been implemented using individual flip-flops, high speed core memory, thin film memory, and other ways in various machines

1.1.2. Register file

A register file is an array of processor registers in a central processing unit (CPU). Modern integrated circuit-based register files are usually implemented by way of fast static RAMs with multiple ports. Such RAMs are distinguished by having dedicated read and write ports, whereas ordinary multiported SRAMs will usually read and write through the same ports.

The instruction set architecture of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the chip. In simpler CPUs, these architectural registers correspond one-for-one to the entries in a physical register file within the CPU. More complicated CPUs use register renaming, so that the mapping of which physical entry stores a particular architectural register changes dynamically during execution.

1.1.3. Register File Design in Modern Microarchitecture

Most register files make no special provision to prevent multiple write ports from writing the same entry simultaneously. Instead, the instruction scheduling hardware ensures that only one instruction in any particular cycle writes a particular entry. If multiple instructions targeting the same register are issued, all but one have their write enables turned off.

The crossed inverters take some finite time to settle after a write operation, during which a read operation will either take longer or return garbage. It is common to have bypass multiplexors that bypass written data to the read ports when a simultaneous read and write to the same entry is commanded. These bypass multiplexors are often just part of a larger bypass network that forwards results that have not yet been committed between functional units.

The register file is usually pitch matched to the datapath that it serves. Pitch matching avoids having the many busses passing over the datapath turn corners, which would use a lot of area. But since every unit must have the same bit pitch, every unit in the datapath ends up with the bit pitch forced by the widest unit, which can waste area in the other units. Register files, because they have two wires per bit per write port, and because all the bit lines must contact the silicon at every bit cell, can often set the pitch of a datapath.

Area can sometimes be saved, on machines with multiple units in a datapath, by having two datapaths side-by-side, each of which has smaller bit pitch than a single datapath would have. This case usually forces multiple copies of a register file, one for each datapath.

The DEC Alpha EV-6, for instance, had two copies of the integer register file, and took an extra cycle to propagate data between the two. The issue logic tried to reduce the number of operations forwarding data between the two. The R8000 floating-point unit had two copies of the floating-point register file, each with four write and four read ports, and wrote both copies at the same time.

Processors that do register renaming can arrange for each functional unit to write to a subset of the physical register file. This arrangement can eliminate the need for multiple write ports per bit cell, for a large savings in area. The resulting register file, effectively a stack of register files with single write ports, then benefits from replication and subsetting the read ports. At the limit, this technique would place a stack of 1-write, 2-read register files at the inputs to each functional unit. Since register files with a small number of ports are often dominated by transistor area, it is best not to push this technique to this limit, but it is useful all the same.

The SPARC ISA defines register windows, in which the 5-bit architectural names of the registers actually point into a window on a much larger register file, with hundreds of entries. Implementing multiported register files with hundreds of entries requires a lot of area. The register window slides by 16 registers when moved, so that each architectural register name can refer to only a small number of registers in the larger array, e.g. architectural register r20 can only refer to physical registers #20, #36, #52, #68, #84, #100, #116, if there are just seven windows in the physical file.

To save area, some SPARC implementations implement a 32-entry register file, in which each cell has seven "bits". Only one is read and writeable through the external ports, but the contents of the bits can be rotated. A rotation accomplishes in a single cycle a movement of the register window. Because most of the wires accomplishing the state movement are local, tremendous bandwidth is possible with little power.

This same technique is used in the R10000 register renaming mapping file, which stores a 6-bit virtual register number for each of the physical registers. In the renaming file, the renaming state is checkpointed whenever a branch is taken, so that when a branch is detected to be mispredicted, the old renaming state can be recovered in a single cycle.

1.2. Overviews of Multithreading

Before we explain our register file design, we introduce threads first. Although instruction is minimal unit to use CPU resources, thread is the base unit to be allocated resources in today's computer system. Understand how computer system allocate resources to threads help us design register file more suitable for execution.

1.2.1. Thread

Thread in computer science is short for a thread of execution. Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does.

Threads are distinguished from traditional multitasking operating system processes in that processes are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Multiple threads, on the other hand, typically share the state information of a single process, and share memory and other resources directly. Context switching between threads in the same process is typically faster than context switching between processes. Systems like Windows NT and OS/2 are said to have "cheap" threads and "expensive" processes; in other operating systems there is not so great a difference.

1.2.2. Multithreading

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

There are two main approaches to multithreading. Fine-grained multithreading switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. To make fine-grained multithreading practical, the CPU must be able to switch threads on every clock cycle. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This change relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grain multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high cost stalls, where pipeline refill is negligible compared to the stall time.

1.2.3. Simultaneous Multithreading

Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 1-1 conceptually illustrates the differences in a processor’s ability to exploit the resources of a superscalar for the following processor configurations:

- a superscalar with no multithreading support,
- a superscalar with coarse-grained multithreading,
- a superscalar with fine-grained multithreading, and
- a superscalar with simultaneous multithreading.

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

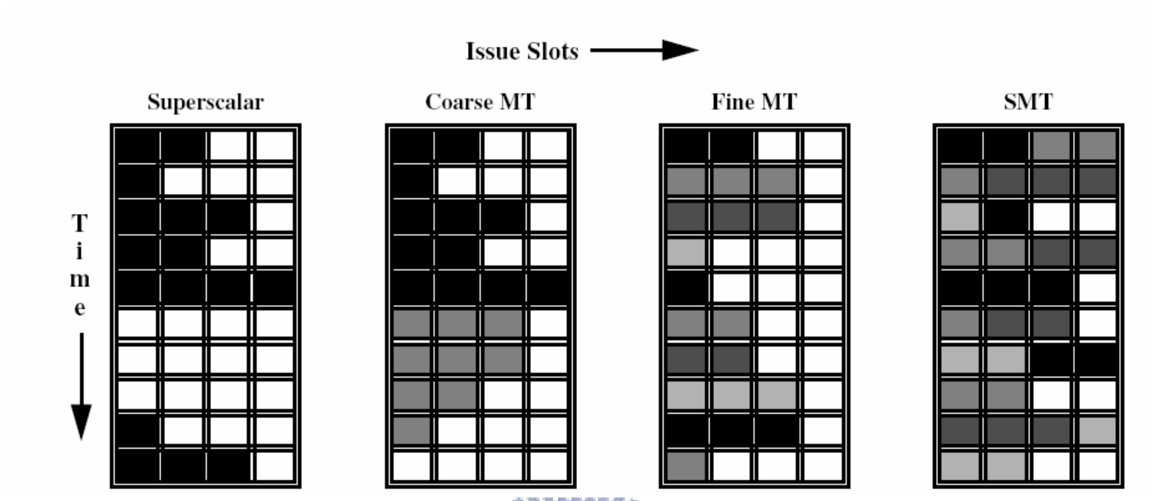


Figure 1-1 This illustration shows how these four different approaches use the issue slots of a superscalar processor.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, within each clock cycle, the ILP limitations still lead to idle cycles. Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle, however, ILP limitations still lead to a significant number of idle slots within individual clock cycles.

In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism

(ILP) are exploited simultaneously; with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors–including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads–can also restrict how many slots are used. Although Figure 1-1 greatly simplifies the real operation of these processors it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

As mentioned above, simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the integrated exploitation of TLP through multithreading. In particular, dynamically scheduled superscalars have a large set of virtual registers that can be used to hold the register sets of independent threads (assuming separate renaming tables are kept for each thread). Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the datapath without confusing sources and destinations across the threads. This observation leads to the insight that multithreading can be built on top of an out-of-order processor by adding a per thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit. There are complications in handling instruction commit, since we would like instructions from independent threads to be able to commit independently. The independent commitment of instructions from separate threads can be supported by logically keeping a separate reorder buffer for each thread.

1.2.4. Design Challenges in SMT Processors

Because a dynamically scheduled superscalar processor is likely to have a deep pipeline, SMT will be unlikely to gain much in performance if it were coarse-grained. Since SMT will likely make sense only in a fine-grained implementation, we must worry about the impact of fine-grained scheduling on single thread performance. This effect can be minimized by having a preferred thread, which still permits multithreading to preserve some of its performance advantage with a smaller compromise in single thread performance. At first glance, it might appear that a preferred thread approach sacrifices neither throughput nor single-thread performance. Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when the preferred thread encounters a stall. The reason is that the pipeline is less likely to have a mix of instructions from several threads, resulting in greater probability that either empty slots or a stall will occur. Throughput is maximized by having a sufficient number of independent threads to hide all stalls in any combination of threads.

Unfortunately, mixing many threads will inevitably compromise the execution time of individual threads. Similar problems exist in instruction fetch. To maximize single thread performance, we should fetch as far ahead as possible in that single thread and always have the fetch unit free when a branch is mispredicted and a miss occurs in the prefetch buffer. Unfortunately, this limits the number of instructions available for scheduling from other threads, reducing throughput. All multithreaded processor must seek to balance this tradeoff.

In practice, the problems of dividing resources and balancing single-thread and multiple-thread performance turn out not to be as challenging as they sound, at least for current superscalar back-ends. In particular, for current machines that issue four to eight instructions per cycle, it probably suffices to have a small number of active threads, and an even smaller number of “preferred” threads. Whenever possible, the processor acts on behalf of a preferred thread. This starts with prefetching instructions: whenever the prefetch buffers for the preferred threads are not full, instructions are fetched for those threads. Only when the preferred thread buffers are full is the instruction unit directed to prefetch for other threads. Note that having two preferred threads means that we are simultaneously prefetching for two instruction streams and this adds complexity to the instruction fetch unit and the instruction cache. Similarly, the instruction issue unit can direct its attention to the preferred threads, considering other threads only if the preferred threads are stalled and cannot issue. In practice, having four to eight threads and two to four preferred threads is likely to completely utilize the capability of a superscalar back-end that is roughly double the capability of those available in 2001.

There are a variety of other design challenges for an SMT processor, including:

- dealing with a larger register file needed to hold multiple contexts,
- maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging, and
- ensuring that the cache conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation.

In viewing these problems, two observation are important. In many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current superscalars is low enough that there is room for significant improvement, even at the cost of some overhead. SMT appears to be the most promising way to achieve that improvement in throughput.

Because SMT exploits thread-level parallelism on a multiple-issue superscalar, it is most likely to be included in high-end processors targeted at server markets. In addition, it is likely that there will be some mode to restrict the multithreading, so as to maximize the performance of a single thread.

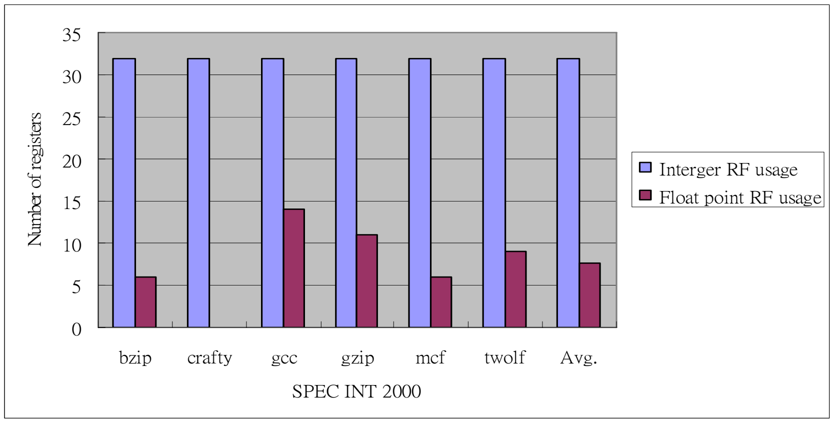
Prior to deciding to abandon the Alpha architecture in mid 2001, Compaq had announced that the Alpha 21364 would have SMT capability when it became available in 2002 In July 2001, Intel announced that a future processor based on the Pentium 4 microarchitecture and targeted at the server market, most likely Pentium 4 Xenon, would support SMT, initially with two-thread implementation. Intel claims a 30% improvement in throughput for server applications with this new support.

1.3. Observation

As the upon mention, modern processor architecture design tends to support SMT because human thirst for performance improving. But SMT also bring lots of problem, one problem we try to solve is the demand for register file. A register file usually occupy a significant portion of area in a processor (In Intel Pentium® 4 Processor Integer Execution Core is about 31.5% and in alpha 21464 processor is about 6%). As we know, most general-purpose processors and embedded processors have 32 architected registers as a register file or more and have two kinds of registers (float-point registers and integer registers). However, a lot of programs just use integer registers or float point registers rarely need the full register file. With appropriate partitioning, those parts of register file could be used by another program. Thus our design try to increase the utilization of register file and solve the area of register file increasing because of supporting SMT by flexibly splitting a register file; moreover, we would try to find the suitable size of our purposed register file to gain the maximal performance. Our purposed register file can be support by single-thread or simultaneous multithread.

1.3.1. Utilization of Register File

We perform some profiling on register usage. Figure 1-2 shows statistics in register utilization of each application. The raw data is retrieved from our simulation environment which is mentioned in chapter 4. We observe the two kinds of register file. One is integer register file usage and the other is float point register file usage. We can see that although programs use lots of integer registers, but they use less of float point registers. This simulation result gives us a good opportunity to accomplish our proposed design in the float point register file.



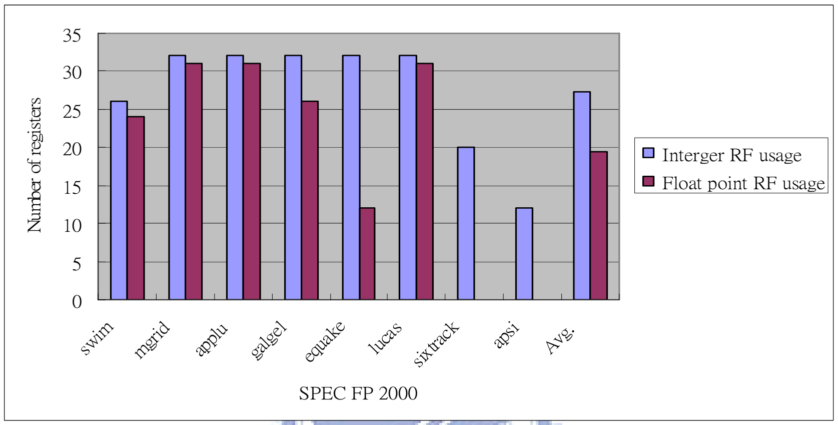


Figure 1-2 Statistics in register utilization of each application

1.4. Motivation and Objective

1.4.1. Motivation

We can see that a register file occupies a significant portion of area in a processor and the portion will be increased because the trend is to increase demand on number of registers in a register file.

Many previous designs use minimal number of ports to reduce the area of register file. But it brings lots of problems, for example, Reduce a multi-port register file may cause conflicts and need to stall the pipeline. It also has to design a complex control logic because of access conflict and it possibly cause performance down.

From the simulation result, we observe the opportunity of sharing the register file. When one thread does not require the full size of the register file, parts of the register file may be used by another thread and if we can split the register file into two parts flexibly to be shared for two threads, then we may get high utilization in this splittable register file. Also, we can try to add few registers in the splittable register file, and we may have higher opportunity to share the register file or relax the high pressure of register file usage because of high requirement from threads.

1.4.2. Objective

Design a flexibly splittable and stretchable register file (RF) which may be divided into two parts shared by two threads (programs, tasks) to get high utilization of the register file in SMT processor without increasing read/write ports by sharing ports and decoder in a register file. We also can reduce context switch thickly and hide memory latency in single-processor.

1.5. Organization of This Thesis

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

Chapter 2. Background and Related Work

2.1. Register File Structure

The Structure of a register file contains decoder design, register array and data bus (word line) design. The Register file organization is shown below

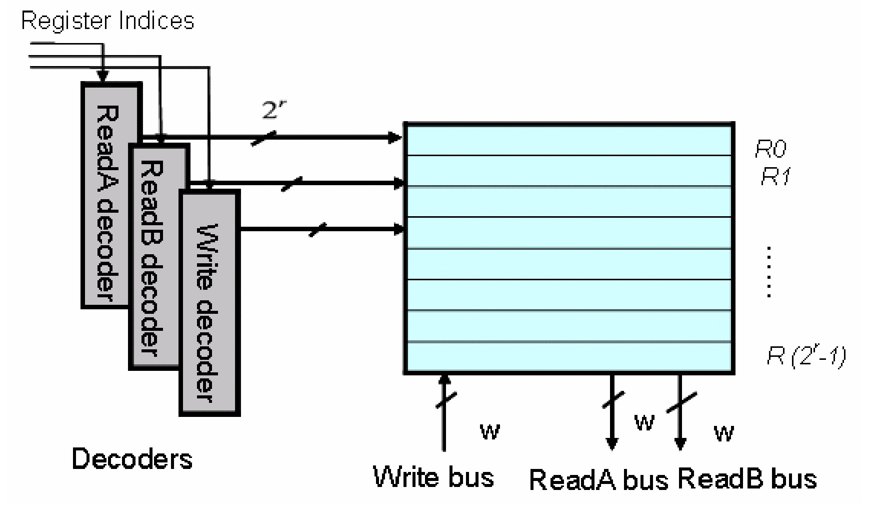


Figure 2-1 Register File Organization

In the decoder design, the decoder has two kinds of address decoders, one is read decoder and the other is write decoder. The decoder is a series of AND gates that drive word lines, and there is one decoder per read or write port. If the array has two read and one write ports, for example, it has three word lines per bit cell in the array, and three AND gates per row in the decoder. Figure 2-2 shows the decoder design.

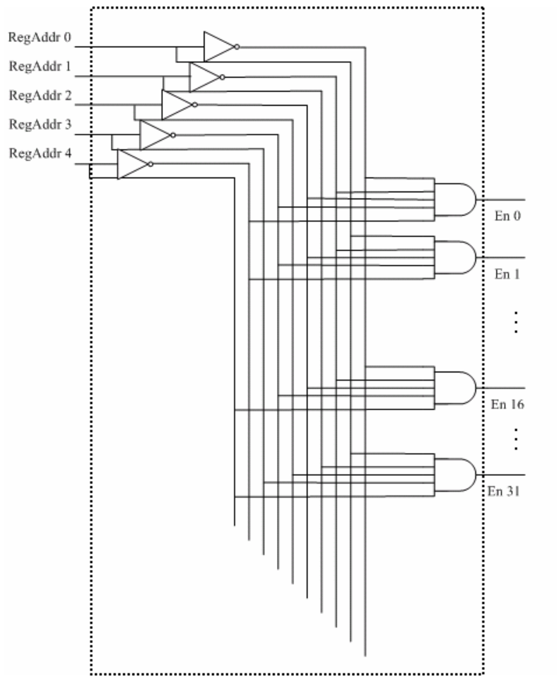


Figure 2-2 Decoder Design

In the register array and bus design, a register array is composed of many bit cells. A bit cell is composed a pair of inverters to store state, a bit line to control mos transister to enable data be read out to word line, and a bit line to control mos transister to enable data be write from word line. A basic 2 read / 1 write port register cell diagram is shown below

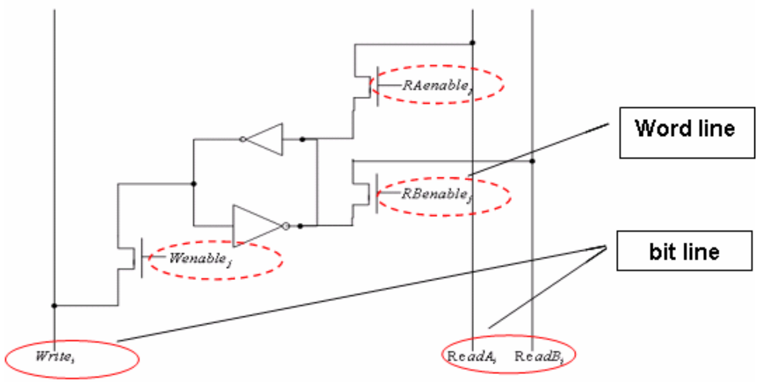


Figure 2-3 Bit cell i of Register j

2.2. Mini-thread: Increasing TLP on Small-Scale SMT

Processors

Before we start to propose possible designs, we first show some researches about the opportunity of sharing the register file on SMT architecture when one task (program, thread, context, etc…) use few registers and also increasing the performance. We also show other design to reduce area by banking register file and complex control logic to solve the conflict problem.

As we knows, SMT is a latency-tolerant CPU architecture that adds multiple hardware contexts to an out-of-order superscalar to dramatically improve machine throughput. While these SMTs increase performance, they still leave modern wide-issue CPUs with underutilized resources.

A primary obstacle to the construction of larger-scale SMTs is the register file. The large register file either inflates cycle time or demands additional stages on today's aggressive pipelines; for example, the Alpha 21464 architecture would have required three cycles to access the register file.

The idea to get performance without increasing the register file size on SMT architecture is proposed in [ ]. The truth is that a significant impediment to the construction of SMT architecture is the register file size required by a large number of tasks. So they propose a idea, called mini-threads, a simple extension to SMT that increases thread level parallelism without the commensurate increase in register file size. Mini-threads, alters the basic notion of a hardware context. On the hardware level, mini-threads add additional per-thread state (aside from general purpose registers) to each SMT hardware context. Using this hardware, an application can exploit more thread-level parallelism within a context, by creating multiple mini-threads that will share the context’s architectural register set.

Figure 2-4 show the main idea of mini-threads, This mechanism focuses on statically partitioning each architectural register set in half between two mini-threads. They suggest two ways of register allocation to accomplish mini-threads, but they don’t realize the ways that they suggest. The design we purpose help mini-threads to accomplish.

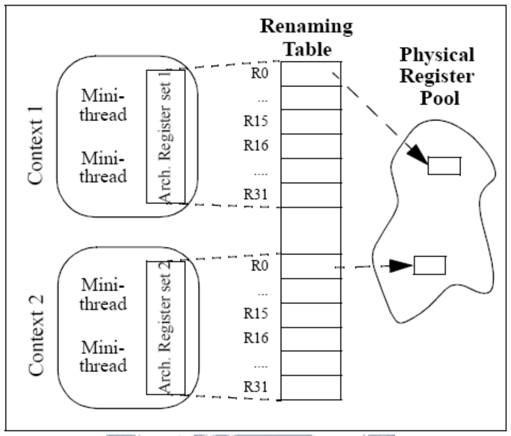


Figure 2-4 Register sharing among mini-threads on an SMT (There are two hardware contexts, each supporting two mini-threads that share architectural registers within the context.)

Figure 2-5 shows the improving performance when the SMT using mini-threads idea. The resulting performance depends on the benefits of additional TLP compared to the costs of executing mini-threads with fewer registers. They demonstrate that mini-threads can improve performance significantly, particularly on small-scale, space-sensitive CPU designs.

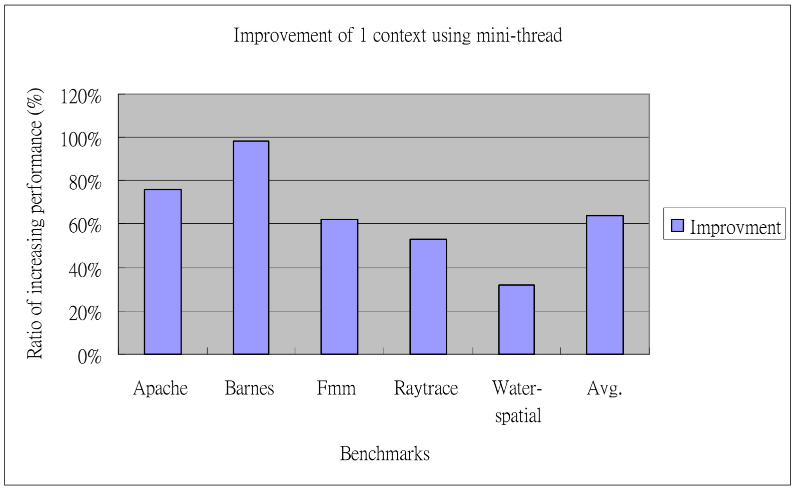


Figure 2-5 The Speedup of each context SMT using mini-threads method

Mini-threads improve on traditional SMT processors in three ways. First, mini-threads conserve registers, because each executing mini-thread does not require a full architectural register set. Second, mt_SMT allows each application the freedom to trade-off ILP for TLP within its hardware contexts. Applications can choose to use mini-threads to increase TLP, or to ignore them to maximize the performance of an individual thread. Third, in addition to the savings in registers, mini-threads open up new possibilities for fine-grained thread programming. Each application can choose how to manage the architectural registers among the mini-threads that share them.

Chapter 3. Design

3.1. Overview of Our Deign

We propose a register file design for a digital computing system. This register file is capable of the following:

1) Splitting the Register File When two independent tasks (processes, threads, etc.) are to be run in the computing system simultaneously, with such a register file, these tasks can share a flexible fraction of the register file in an independent fashion, if the total number of required registers does not exceed the amount of registers available.

2) Flexibly splitting the Register File can make the splittable register file to be used more flexible, not just divide the register file into two parts

3) Stretching the Register File Although an ISA typically defines 2^r logical registers, many applications use only a few of these registers. Furthermore, register files typically occupy a large percentage of chip area. For two processes using only a little more than 2^r registers, sharing of the register file for simultaneous executions is still possible with the stretchable register file design technique.

3.2. The Main Design

We propose a number of design techniques for the register file. With these design techniques, the register file can be very versatile for those purposes above. Note that combinations of these design techniques are possible if so required.

3.2.1. Splittable Register File

The objective to split a 2^r *W register file into two independent partitions of sizes X*W and (2^r -X)*W, both accessible as R0 and up in the split register file. Figure 3-1 illustrates the concept of splittable register file design.

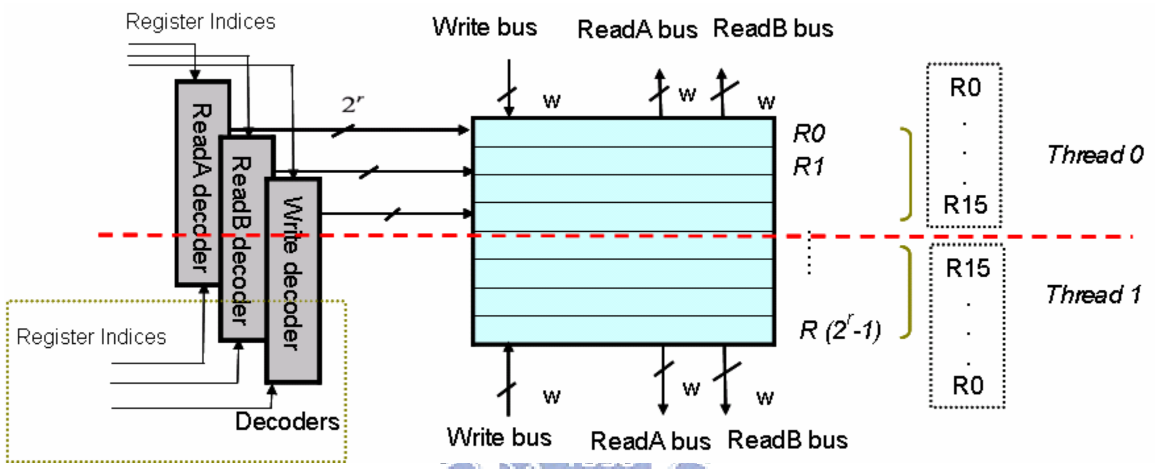


Figure 3-1 Split Register file Design

There are two design issues about how to split the register file:

3.2.1.1. Splittable Decoder Design

Decoders occupy significant silicon area in a register file design. Hence it is desirable if two threads sharing the register can share decoders. Intuitively, one may choose to use two separate sets of Read Decoder A, Read Decoder B, Write Decoder for the two independent register file partitions, wasting much area. We will show that using only one set of decoders is possible.

Registers are numbered as R0~R(2^r -1), or 00…0~11…1, suppose that we want to index both register file partitions as R0~Rmax. Hence if we want to use two sets of decoders, then one set of decoders must be positioned up side down. Or if we want to share only one set of decoders for both register file partitions, then we must first split the decoders at the right point for both small-index portion and large-index portion decoding. Then for the large-index partition, we note that since Not(111…1) = (000…0), if we send in inverted register indices for decoding, then the register uses of this portion will be from R(2^r -1) and down, which matches our need perfectly.

Given 2^r registers in an instruction set architecture, the compiler/assembly program writer always can decide to use only any number of registers ≦ 2^r, at the cast of possible extra register spills/refills. An x-to-2^r decoder is shown below (see Figure 3-2). To make such a decoder sharable by two threads, the register file splitting scheme can be applied here.

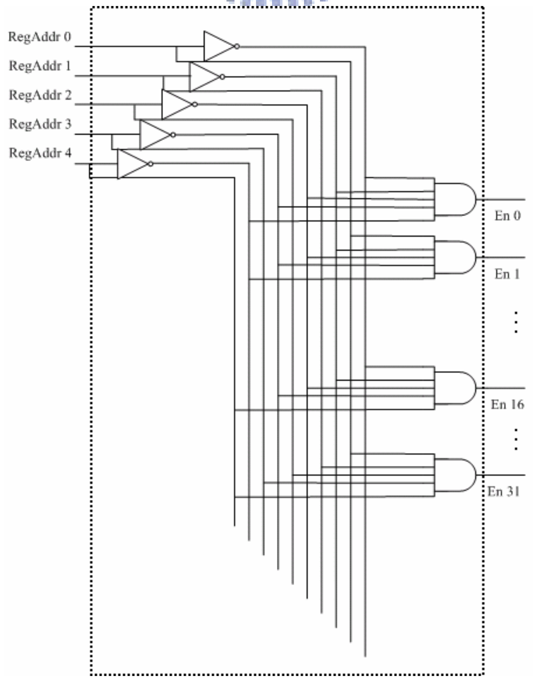


Figure 3-2 Register Decoder Design

Next we discuss the splitting and sharing of only one decoder group. Figure 3-3 shows such a design and the split point selection should be in coincidence with the register split points, and the split control lines for both designs are the same. Since the two tasks sharing the register file both desire to index registers starting from R0, so design change may be needed. Notice that the index 0, if inverted, becomes [2^r -1], or to generalize it, for any x+y=[2^r -1], x,y∈N, $\overline{x}$=y and $\overline{y}$=x. Hence if we

- Invert the ReadA, ReadB, and Write register indices of one of the tasks,
- Split the decoders using pass transistors (transmission gates) as done in data storage part of the register file, and
- Send the inverted registers to the high-index half of the decoders,

Then the sharing of register index decoders becomes very feasible.

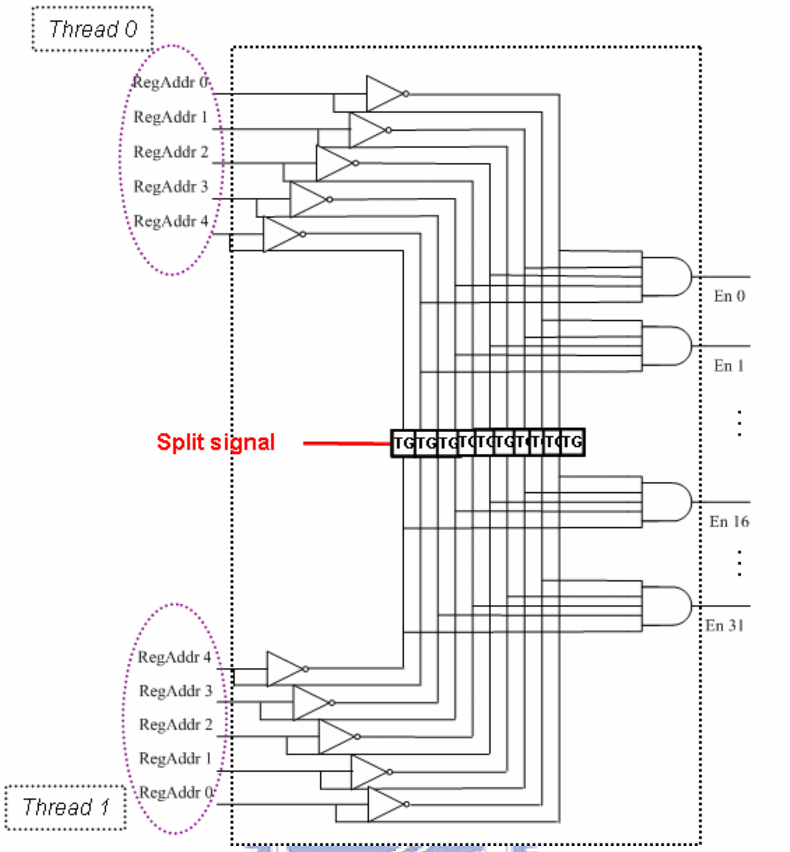


Figure 3-3 Splitting Register Decoder Design in Read/Write Bus, showing only one read bus line

3.2.1.2. Splittable Bus Design

Operand bus design is discussed here. Two designs are possible. The first design is straightforward and expensive: we simply double the number of read/write buses, and read/write ports of a bit cell. This requires much area, and may induce much power and latency penalties. The second design is recommended, since it is cheap and efficient, and also brings a number of additional advantages: The size of register file can then be stretched, the use of register file as two separate sets of register files is hardware enforced and protected, etc. This design sets a number of split points along each read/write bus line, and the read/write data ports can be accessed only at one end of the register file, the R0 end, by the two tasks. Place of the split points are to be determined by register pressures of tasks statistics. Note that the more split points we set, the better sharing flexibility we get, but the worse bus delay will be; and vice versa.

Figure 3-4 shows the design overview of traditional register file with a read decoder. We take a bit line as a example (framed by a red-dotted line) and one bit-line is shown in Figure 3-5.

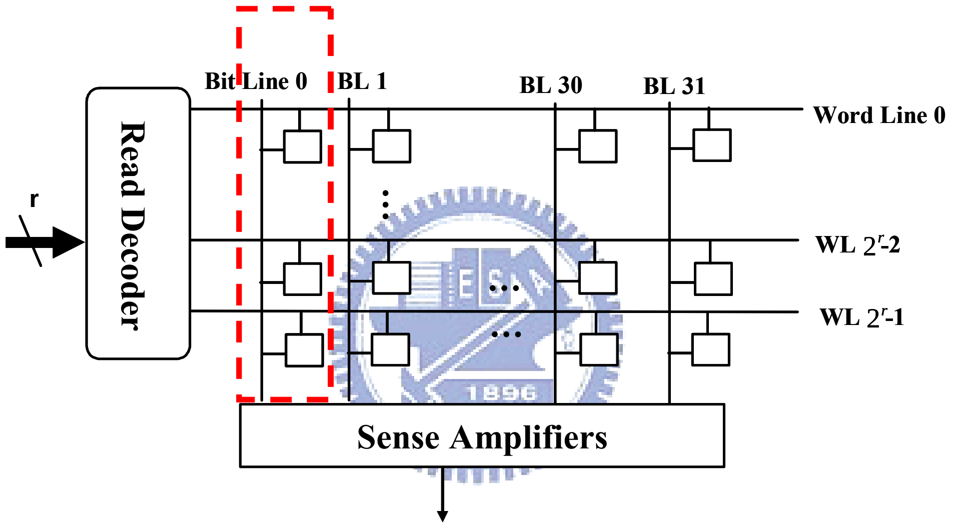


Figure 3-4 Design overview of traditional register file

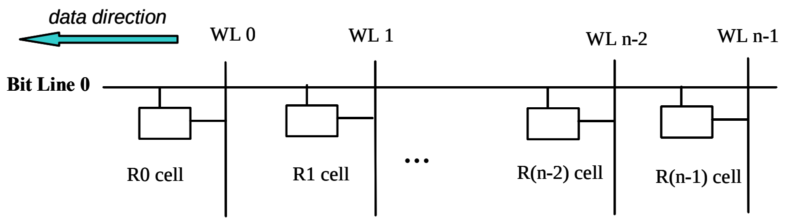


Figure 3-5 Overview of one bit line

Figure 3-6 illustrates our split point read/write bus design. This design implements a number of pass transistors or transmission gates along the bus and since we have two data output, we have to add a extra sense amplifier (SA) and some pre-charge circuits, as illustrated in the figure and the whole design is shown in Figure 3-7.

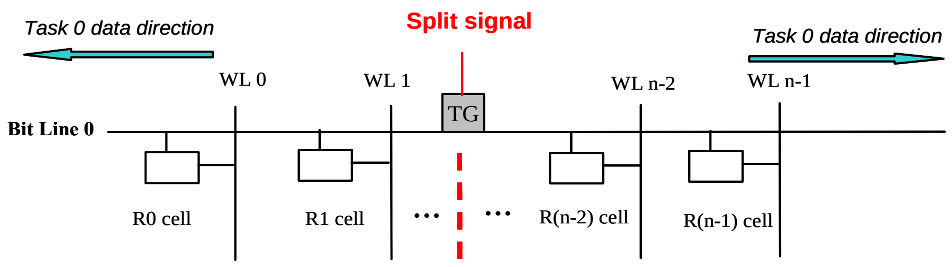


Figure 3-6 One bit-line with split point read/write bus design

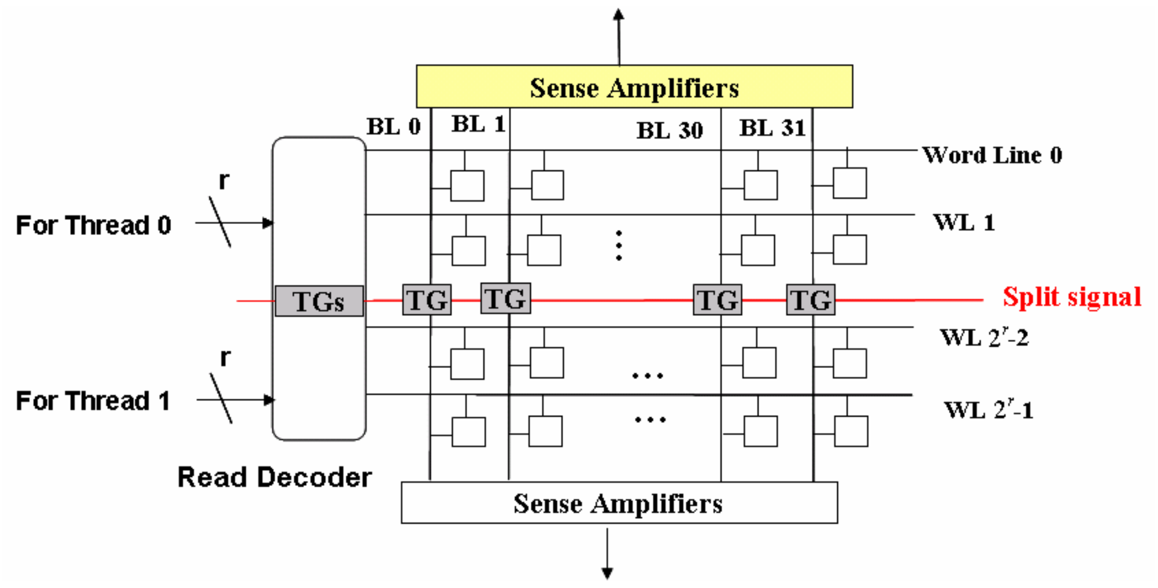


Figure 3-7 A splittable Register File design

3.2.2. Flexible Splittable Register File

Figure 3-8 illustrates a example of our flexible split point read/write bus design. This design implements many groups of pass transistors or transmission gates along the bus. The arrangement of these split points is flexible. For instance, we can implement these points for every pair, or every four registers. We may also reserve at least 2^{r-1} registers for the top set of registers, and/or 2^{r-1} registers for the bottom set of registers. Note that the split points need to exist at most in half of the register spans.

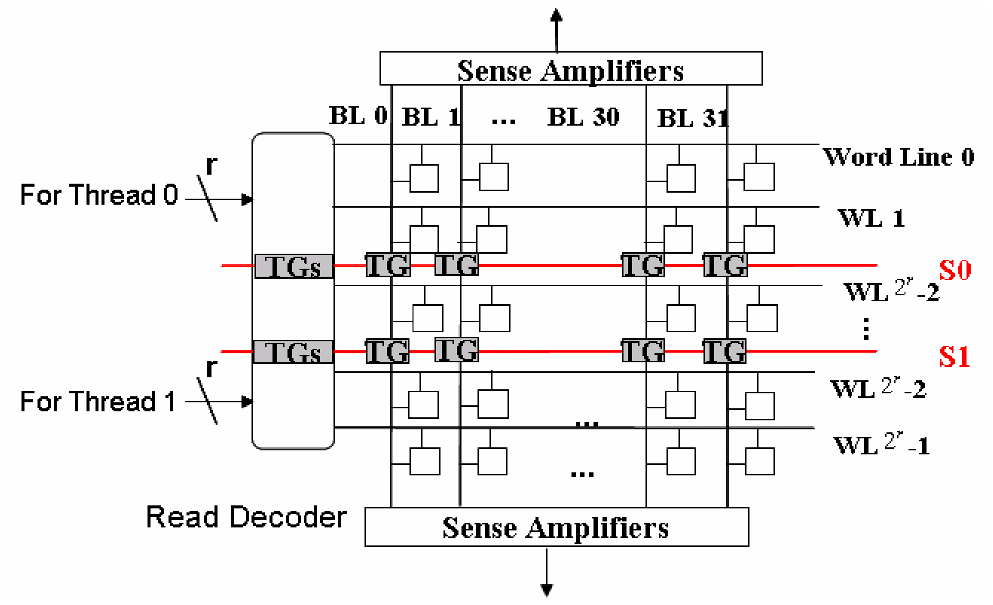


Figure 3-8 Example of a flexible split point Read/Write Bus Design with two groups of transmission gate

S0 and S1 in Figure 3-8 are control signals of split points. Table 3-1 shows the operation when we control the signals. As we know, when signal equals to 0, the split point (transmission gate) on the bus disconnects and connects when signal equals to 1. So we use different signals to make bus operate flexibly and the same way is suitable for flexibly splittable decoder design.

S0	S1	One or two tasks
0	1	two
1	0	two
1	1	one
0	0	No operation

Table 3-1 The operation when we control the signals

3.2.3. Stretchable Splittable Register File

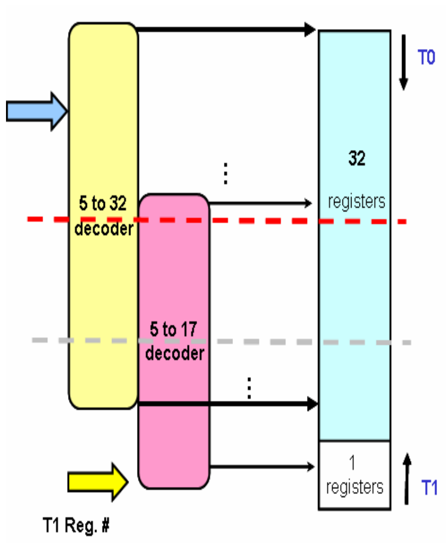
Based on the flexibly splittable register file design, if two threads try to share this register file need an aggregated size of $> 2^r$ registers, we can provide more spare registers in register file, making total # of registers 2^r +s, where s is a small positive integer, to make this sharing more possible.

The design issues of flexibly splittable and stretchable register file are :

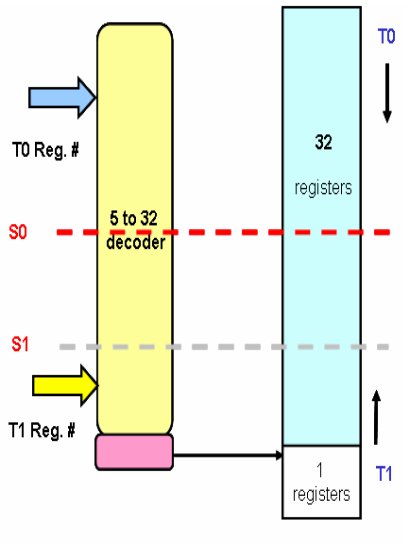
- This flexibly splittable register file needs to preserve one register file property. When added s registers, the one-register file feature must be preserved.
- How to design the decoder to map 2^r +s registers without complex circuit.

The main problem is the decoder design. Decoder must support when increasing the registers. If thread 1 access register from stretched registers, splittable decoder design may not be used. We need to duplicate the decoder to map the register index from thread 1 to decoder or use offset mechanism. If we put stretched registers at the end of registers that thread 1 can access, the mapping from new decoder to a stretched registers is easy and saving area.

There are two design issues of the decoder design when adding extra registers. The design is shown in Figure 3-9. Figure 3-9(a) shows a dedicated decoder for thread 1 and thus thread 1 can start from the stretched registers. The idea of the design 1 is two decoders for two threads and it may increase area. Figure 3-9(b) shows an extra decoding circuit for stretched registers and it can be implemented in the splittable decoder. Desirable features of this design over other alternatives are first, both threads can share the same set of register index decoders, and secondly, it greatly increases the probability of two tasks sharing this stretched register file, since the sharing increases much register pressure.



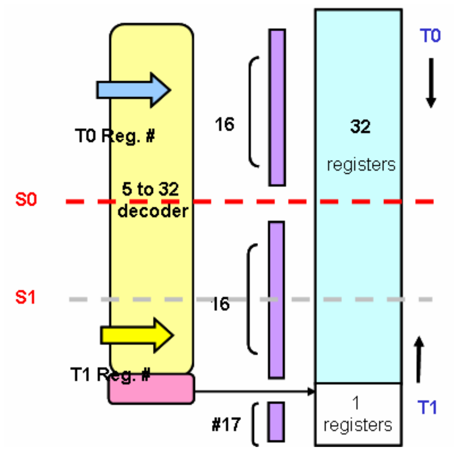
(a). Design 1 (Two decoder mapping)



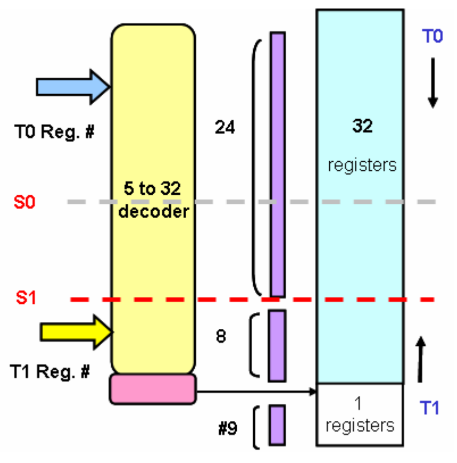
(b). Design 2 (Extra decoding circuit mapping)

Figure 3-9 Schemes of adding extra registers in register file

Design 1 of the decoder is simple and straightforward and design 2 is much complex. The main idea of design 2 is shown in Figure 3-10. Figure 3-10 shows main idea of thread 1 using the extended registers flexibly, Figure 3-10(a) and (b) show different condition of flexibly splittable and stretchable register file.



(a) Condition 1

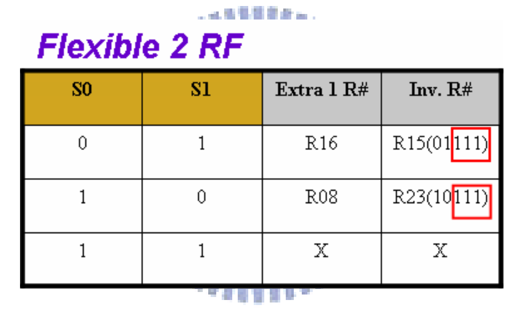


(b) Condition 2

Figure 3-10 The main idea of thread 1 using the extended registers flexibly.

Let’s use a flexibly splittable register file with 2 split points as a example to realize design 2. We can see that when S0=0, S1=1 in Table 3-2, the register split into two equal parts, as shown in Figure 3-11(a), and if thread 1 want to enable the extra register, it has to send the signal No.17 (R16) of register index because the start register index is R0. When S0=1, S1=0 in Table 3-2, the register split into two different size parts, as shown in Figure 3-11(b), and if thread 1 want to enable the extra register, it has to send the signal No.9 (R8) of register index.

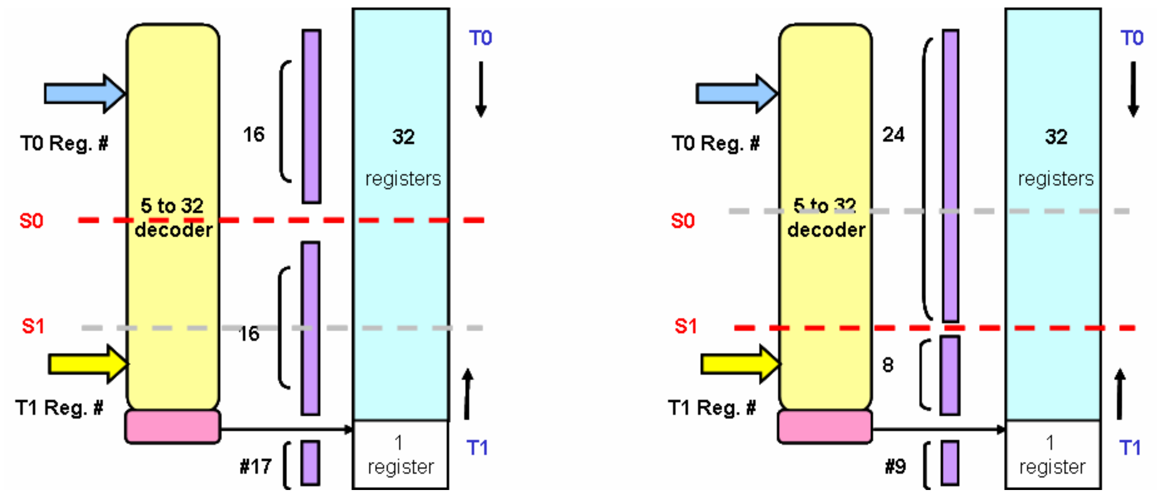
We also observe that the extended register has the same register address of thread 1 at the back and the length is equal to the interval of Flexibility. See Table 3-2, we use a red frame to show.



Flexible 2 RF

S0	S1	Extra 1 R#	Inv. R#
0	1	R16	R15(01111)
1	0	R08	R23(10111)
1	1	X	X

Table 3-2 The enable signal of Flexibly splittable RF (2 split points) to a extra register



(a) Condition 1 (b) Condition 2

Figure 3-11 a example of flexibly splittable register file design with 2 split points when adding one extra register.

Thus we cut down the fixed variable and use these remaining variables to derive the Boolean function. For example, the original Boolean function of extra register of Figure 3-11 is:

$$\text{Extra 1} = (S0' \cdot S1 \cdot addr4' \cdot addr3 + S0 \cdot S1' \cdot addr4 \cdot addr3') \cdot addr2 \cdot addr1 \cdot addr0$$

Figure 3-12 shows an example of flexibly splittable decoder design with one extra register.

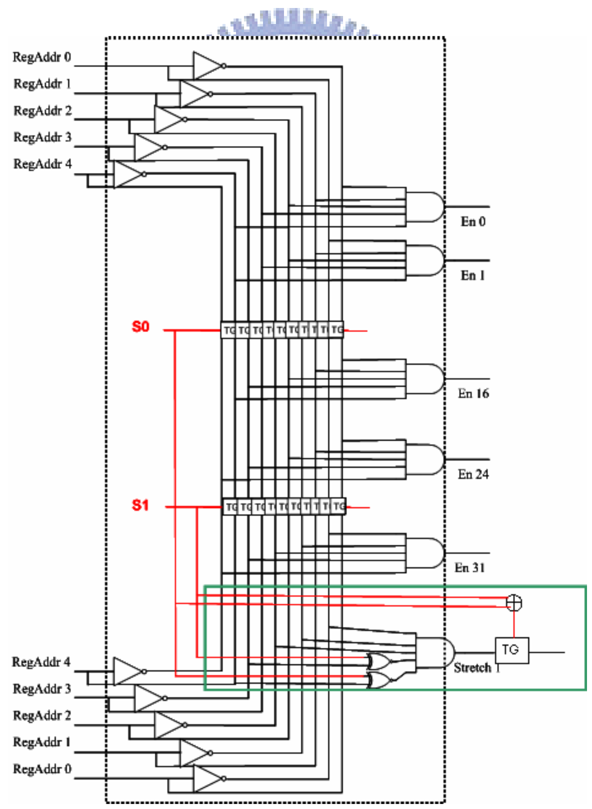


Figure 3-12 An example of flexibly splittable decoder design with one extra register.

3.3. Compiler and OS support

The new register file design implemented in processor has several compiler and operating system support issues. First, when programs are compiled to be object codes, registers have to be allocated successively. If a program use three registers and compiler allocate R0, R1 and R18, it is hard to use our purposed design of splittable register file. Second, When OS schedule the compiled threads (or programs, tasks, etc.), OS has to know register usage of each thread and need to solve the problem when two threads don’t finish the use of CPU simultaneous. We will discuss later.

To solve the first problem, we propose an idea about register gathering [] when compiler executes register allocation. Register gathering is a method to gather registers when register allocation in compile time to name successive register and thus make splittable register design available. For example, if a program use three registers and compiler allocate R0, R1 and R18, we use register gathering to be reallocated to R0, R1 and R2. Figure 3-13 shows the diagram of operation of a compiler.

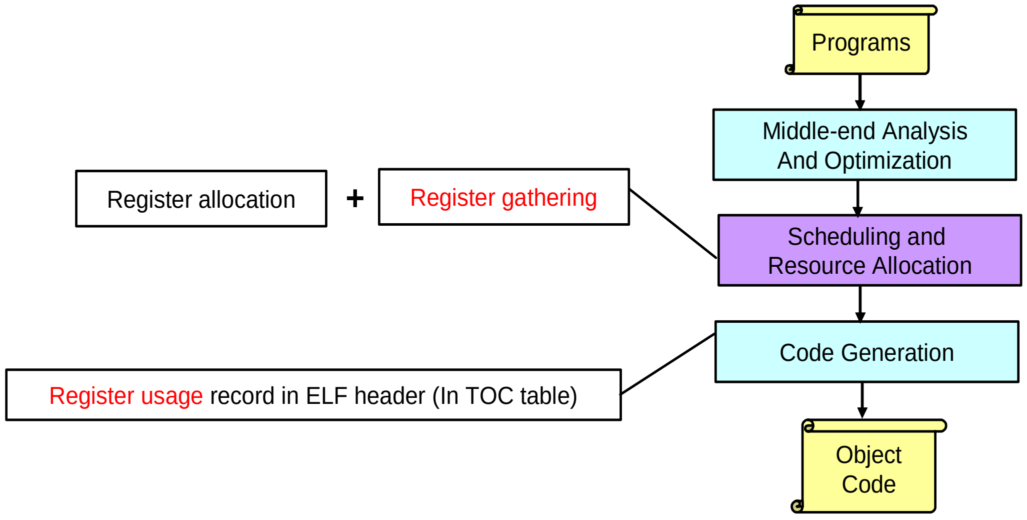

Programs
Middle-end Analysis And Optimization
Register allocation
+
Register gathering
Scheduling and Resource Allocation
Code Generation
Register usage record in ELF header (In TOC table)
Object Code

Figure 3-13 The diagram of operation of a compiler

About the second issue, we discuss here. How dose OS know the register usage when a thread want to be executed. Register usage already be recorded in file header. For example, the register usage of each thread has been recorded in TOC Table of Executable and Linking Format (ELF, a common standard file format) header. [] When OS get the information of register usage of each thread, it can set the information as a priority factor to schedule these threads and OS can signal the processor to split register or not according to the register utilization of each thread. Figure 3-14 shows the diagram of OS signals the processor.

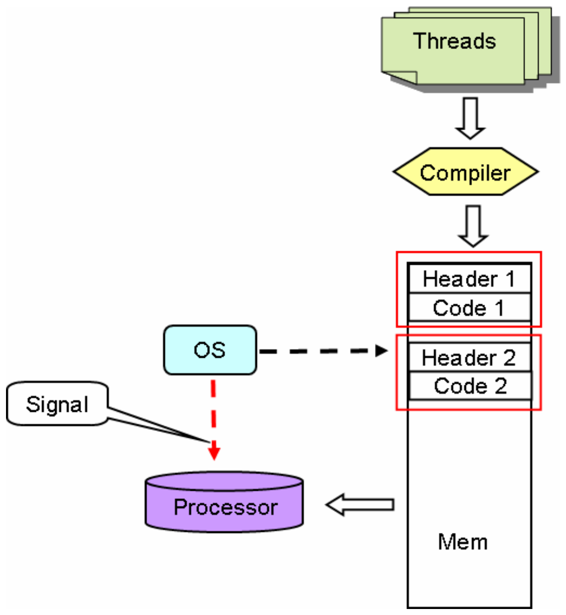


Figure 3-14 A diagram of OS signals the processor

How to solve the problem when two threads don't finish the use of CPU simultaneous? Figure 3-15 shows an example of OS choose the next suitable thread. We can use a table or special registers to solve this problem. We use a table or special registers to save the register usage when the thread is execution. When two threads execute simultaneous, one thread finish first, OS choose the next thread in the ready queue. OS will decide if the thread in execution plus the next thread which want to execute is smaller than the entries of register file that can support. If yes, then the next thread can be a candidate to be execution. If no, then OS choose another suitable thread to execute. Thus the second issue can be solved.

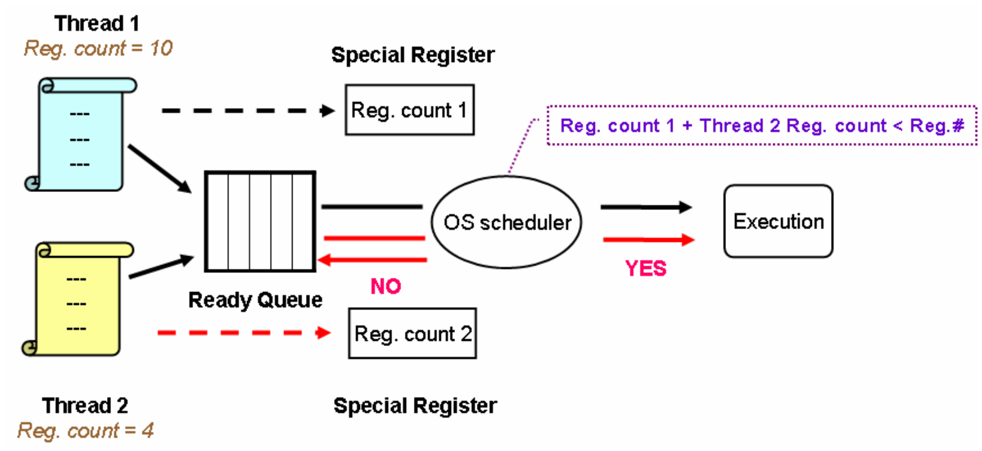


Figure 3-15 An example of OS choose the next suitable thread.

Chapter 4. Experiments

4.1. Goals of Our Experiments

In the experiments, we'll compare different entries of register file using our splittable design and other mechanisms, from area and circuit delay.

We also wish to choose parameters for designing the Register File. We'll find the suitable number of split points by considering both delay and the utilization of register file. Simulation for execution-time reduction with different register file designs is also performed.

Another register file design issue is how many extra registers the register file should be? Since increasing the extra registers could improve the opportunity of joining two tasks together, we observe the effects on adding number of registers to make the conclusion.

4.2. Simulation Environment

Synthesis Environment and Constraints

Tool: Synopsis Design Compiler

Technology:.18um

Register file Implementation:

Provides AND / Tri-state buffer / Flip-flop / XOR / NOT Gates.

The cell of each register is implemented as 2-read / 1-write and 4-read / 2-write.

Software Simulation Environment

Simulator: M-Sim []

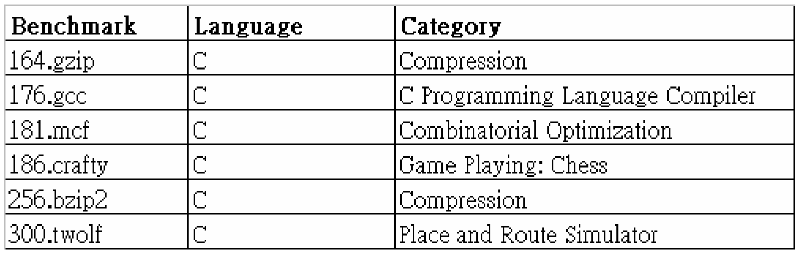
M-Sim is a multi-threaded microarchitectural simulation environment with a detailed cycle-accurate model for the key pipeline structures. M-Sim extends the SimpleScalar 3.0d toolset with accurate models of the pipeline structures, M-Sim supports for the concurrent execution of multiple threads according to the Simultaneous Multithreading (SMT) model and gives each thread a copy of integer and float point register file.

Benchmark: SPEC CPU 2000 Suite []

SPEC CPU 2000 is the industry-standardized CPU-intensive benchmark suite. SPEC designed CPU 2000 to provide a comparative measure of compute intensive performance across the widest practical range of hardware. The implementation resulted in source code benchmarks developed from real user applications. These benchmarks measure the performance of the processor, memory and compiler on the tested system.

A survey of the benchmarks that we use comprising each SPEC CPU2000 component suite:

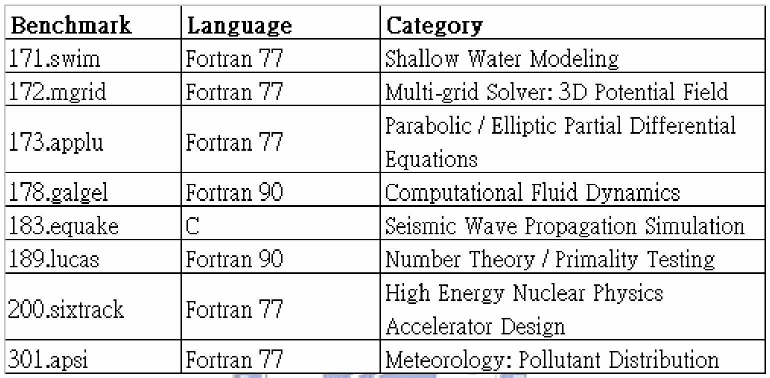
- CINT2000 - The Integer Benchmarks. See Table 4-1



Benchmark	Language	Category
164.gzip	C	Compression
176.gcc	C	C Programming Language Compiler
181.mcf	C	Combinatorial Optimization
186.crafty	C	Game Playing: Chess
256.bzip2	C	Compression
300.twolf	C	Place and Route Simulator

Table 4-1 CINT2000

- CFP2000 - The Floating Point Benchmarks. See Table 4-2



Benchmark	Language	Category
171.swim	Fortran 77	Shallow Water Modeling
172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field
173.applu	Fortran 77	Parabolic / Elliptic Partial Differential Equations
178.galgel	Fortran 90	Computational Fluid Dynamics
183.equake	C	Seismic Wave Propagation Simulation
189.lucas	Fortran 90	Number Theory / Primality Testing
200.sixtrack	Fortran 77	High Energy Nuclear Physics Accelerator Design
301.apsi	Fortran 77	Meteorology: Pollutant Distribution

Table 4-2 CFP2000

Simulation Methodology:

We first observe the register file utilization of each application, as show in Figure 1-2 (This figure is shown in observation of Chapter 2). We divide these applications into 2 parts, one is higher float point registers utilization (>16) and the other is lower float point registers utilization (<16). Then we compare the performance of higher-higher applications, higher-lower applications and lower-lower applications running in traditional SMT architecture and my design in SMT architecture. Table 4-3 shows the classification of Register file utilization in each benchmark.

The classification of RF utilization	**benchmarks**
Higher FP registers utilization	Swim, mgrid, applu, galgel, lucas
Lower FP registers utilization	Bzip, crafty, gcc, gzip, mcf, twolf, equake, sixtrack, apsi

Table 4-3 The classification of Register file utilization in each benchmark

We simulate the IPC of 1-thread and all kinds of 2-thread workloads in traditional SMT architecture and our design. Table 4-4 shows a example of simulated 1-thread and 2-thread workloads.

1 thread	Swim, mgrid, applu, galgel, lucas, bzip, crafty, gcc, gzip, mcf, twolf, equake, sixtrack, apsi		
2 threads	higher – higher (H-H)	higher – lower (H-L)	lower – lower (L-L)
	Swim + Swim	Swim + apsi	bzip + crafty
	Mgrid + mgrid	mgrid + crafty	crafty + gcc
	applu + applu	applu + gcc	gcc + gzip
	galgel + galgel	galgel + bzip	gzip + mcf
	lucas + lucas	lucas + twolf	mcf + twolf
	Swim + lucas	Swim + mcf	twolf + equake
	mgrid + Swim	mgrid + equake	equake + sixtrack
	applu + mgrid	applu + sixtrack	sixtrack + apsi
	galgel + applu	galgel + apsi	apsi + bzip
	lucas + galgel	lucas + gzip	bzip + crafty
	….	……	……

Table 4-4 Simulate 1-thread and 2-thread workloads

Then we calculate the average IPC by using the formula is shown below :

$$\text{Avg. IPC} = \frac{(\text{Avg. H - H IPC})*15+(\text{Avg. H - L IPC})*45+(\text{Avg. L - L IPC})*45}{105}$$

4.3. Area Simulation of Different Designs vs. Splittable Register File Design

Based on our idea, we implement the splittable design in different size of register file and compare with a copy of traditional register file (2r/1w, n entries), two copies of traditional register file (2r/1w, 2n entries) and a multi-port register file (4r/2w, n entries). Figure 4-1 shows the area comparison of these designs and splittable RF design.

According to the simulation result, we could know the overhead of the splittable RF design doesn’t get a heavy proportion compared with a copy of traditional register file (Adding avg. 3.4 % overhead in splittable RF design).Because the overhead we increase is regular, the proportion of overhead decrease when the size of RF increase, as show in Figure 4-2.

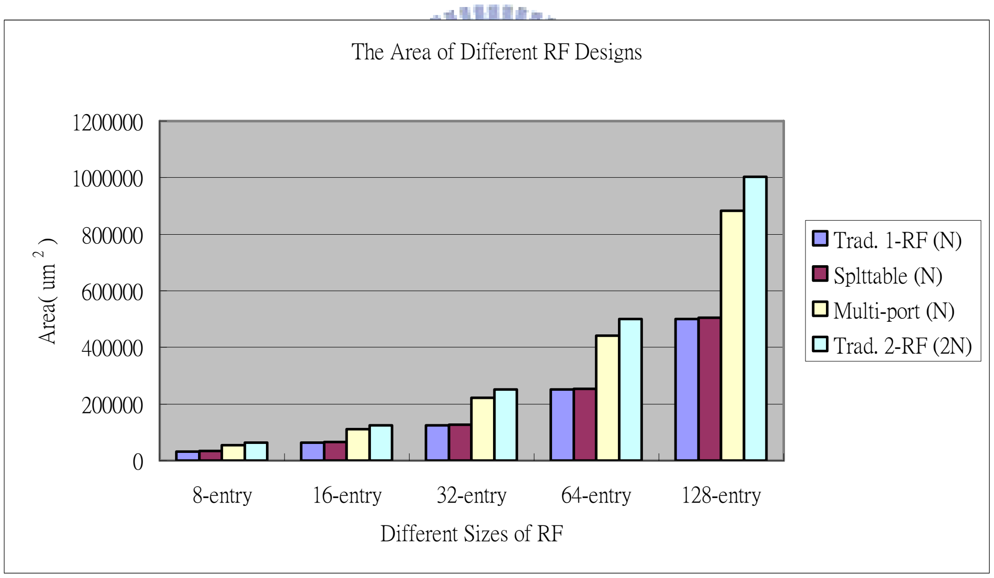


Figure 4-1 The area comparison of other RF designs and splittable RF design

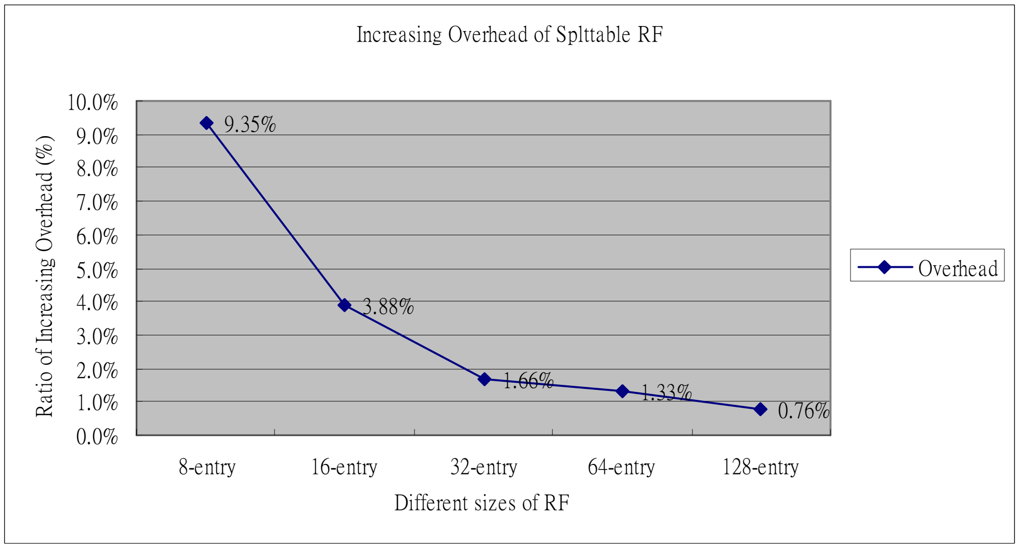


Figure 4-2 The ratio of increasing overhead of splittable RF design

However, we use tri-state buffer to implement the transmission gate, it brings lots of delay. Figure 4-3 shows the access time of a traditional register file and splittable RF design. How to solve the delay problem that the splittable design brings? Since we try to use a splittable n-entry register file to replace a traditional 2n-entry register file, the delay problem can be relaxed. We also can add an extra pipeline for register reading or writing. In the superscalar, it costs two pipeline stage times for register reading.

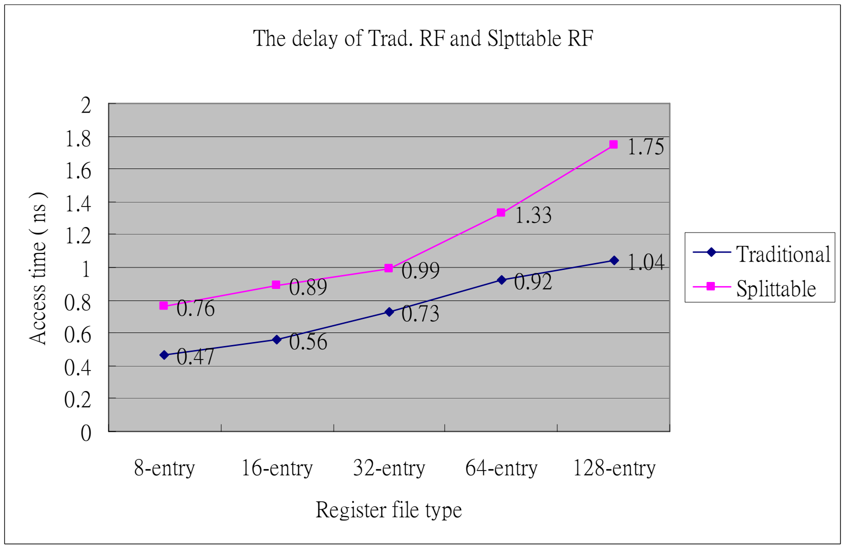


Figure 4-3 The access time of each size of a traditional register file and splittable RF design

We then compare the multi-port register file with our design in equal entries. Since we can use multi-port register file to reach the same purpose of our design. But we get less overhead compared with multi-port register file design. Figure 4-4 shows saving area (Avg. 41.2%) if we replace multi-port register file (4-read/2-write ports, n entries) by splittable RF (n entries). We also see that as the size of RF increase, the proportion of save area increases.

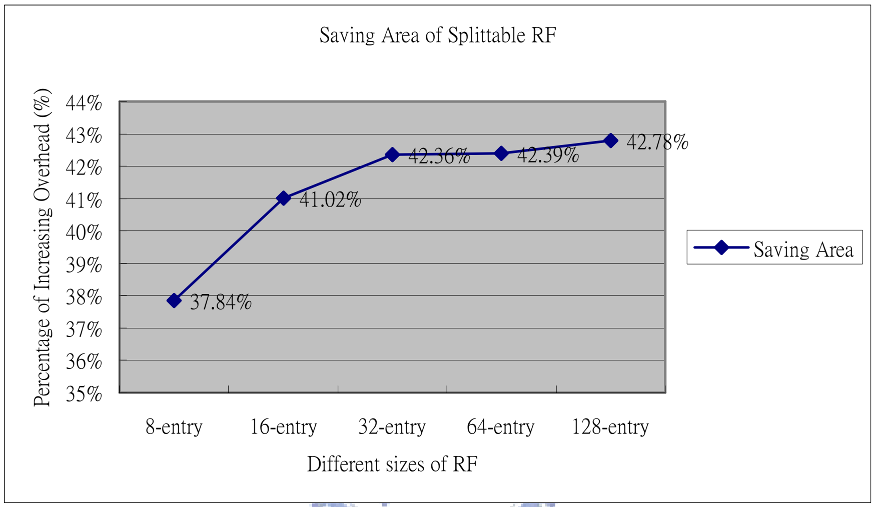


Figure 4-4 The saving area of multi-port register file (4-read/2-write ports, n entries) vs. splittable RF (n entries) design

We also compare the traditional SMT architecture using register file with our design. Traditional SMT architecture use duplicated register file design (ex: Alpha 21464). It gives each ALU a copy of traditional register file and is easy to implement without complex control logic. Figure 4-5 shows the saving area if we replace two copies of traditional register file (2-read/1-write ports, 2n entries) by splittable RF (n entries). We can see that the splittable RF saves lots of area (Avg. 48.3 %) of two copies of traditional RF and as the size of RF increase, the proportion of save area increase.

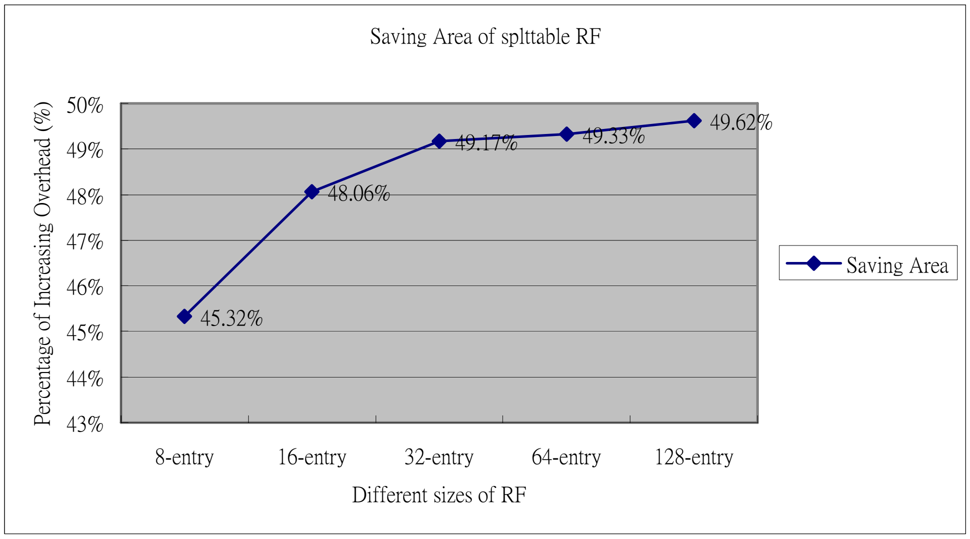


Figure 4-5 The saving area of two copies of traditional RF(n entries) vs.

splittable RF(n entries) design

4.4. Simulation of Different Flexibility Designs

In section 3.2.2., we proposed the flexibility mechanism which could increase more opportunity to execute two tasks simultaneous. We add more transmission gates as split-point to make our design more flexible. Figure shows area and delay overhead of 32-entry register file with different split-points.

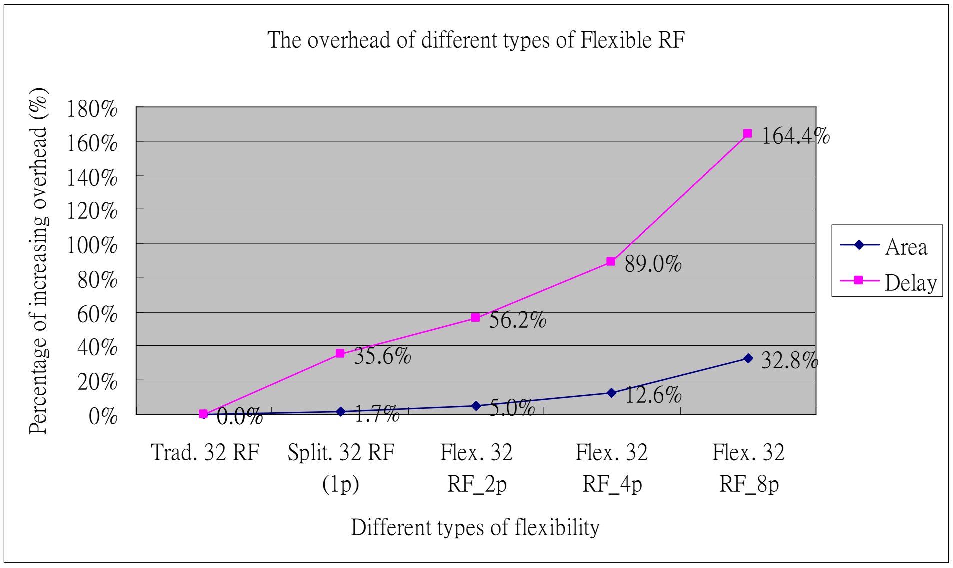


Figure 4-6 Area and delay overhead of 32-entry register file with different split-points

From the Figure 4-6 above, we can see under all flexibility configures, the area for flexibility approach is increased slowly. But it requires quite large delay time to access this design.

4.5. Simulation of Different Stretchability Designs

Performance could be increased if we increase the entries of register file since the opportunities of joining two tasks together would be increased. But how many entries should be extended is most economic? We could refer to the overhead and the ratio of increased performance over increased entries of register file.

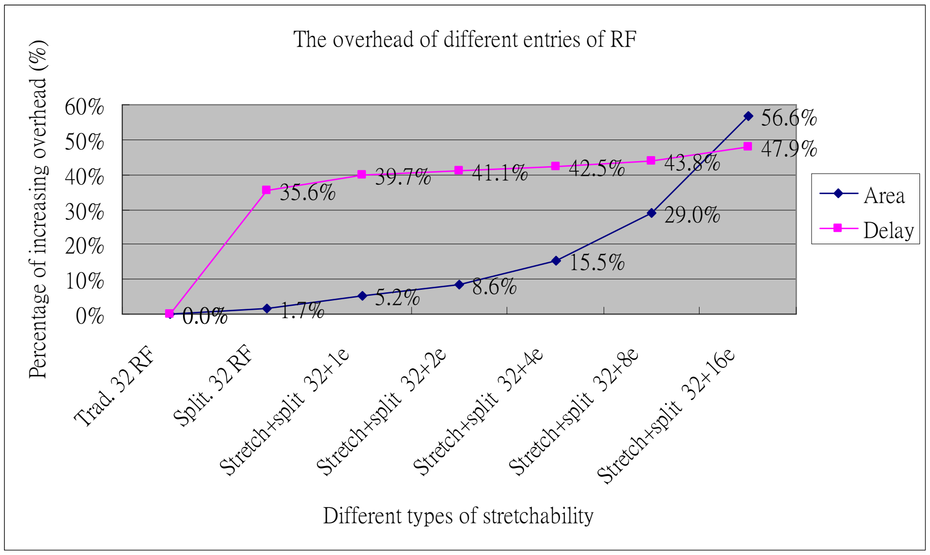


Figure 4-7 Ratio of overhead over increased Register-entry

In Figure 4-7 we show the ratio with power of 2 entries step in 32-entry splittable register file. Increasing the entries don’t increase much delay (access) time, but increase the area dramatically. Overhead of splittable register file is contrary to the overhead of flexible register file design. We will simulate the performance in next section to choose the most suitable design that we purpose.

4.6. Performance Simulation on All Kinds of Our Deign

Figure 4-8 shows the performance of a copy of traditional register file (Trad. 1-T), different flexibilities of register file under three kinds of benchmark sets and two copies of traditional register file (Trad. 2-T). Since multi-port register file has the same ability with our purposed design in the same entries, we consider them get the same performance.

Then we can see that the different flexibilities impact on the higher-lower benchmark set. The flexible register file doesn't impact on Higher-Higher benchmark because the Higher-Higher benchmark needs more register but not flexibility and we always can execute two tasks simultaneously in Lower-Lower benchmark set.

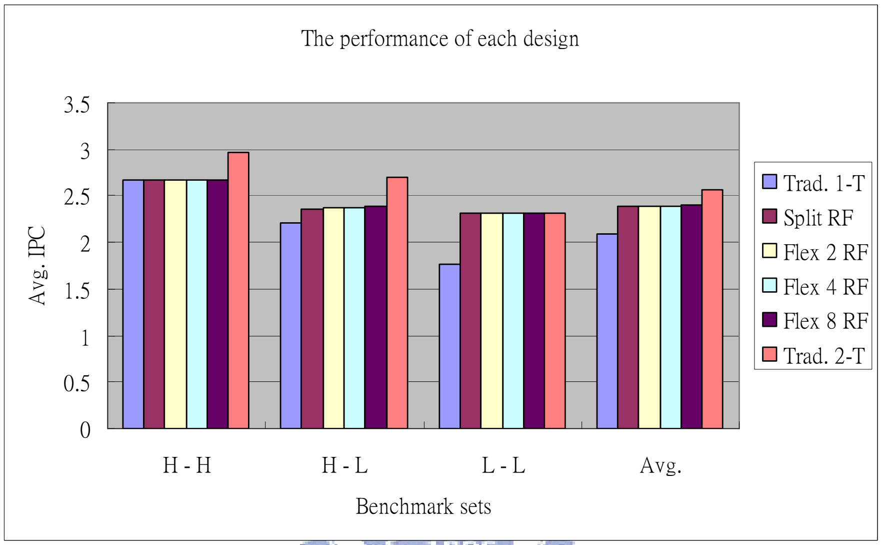


Figure 4-8 The performance of different flexibilities of register file under three kinds benchmark sets

We calculate the average IPC of Figure 4-8 above and show it in the Figure 4-9 below. We can observe that if we give two copies of traditional register file (Traditional SMT), it can improve the performance 23.3% over one copy of traditional register file. When we use the splittable register file, it can improve the performance 14.1% over one copy of traditional register file. Thus we save almost a copy of register file (about 49.1%) but just lose a little performance (about 9.1%) compared with two copies of traditional register file.

Figure 4-9 also shows that the performance gets slightly increasing with the flexibility increasing. It gets 1% performance increasing in flexible register file with 8 split-points compared with the splittable register file.

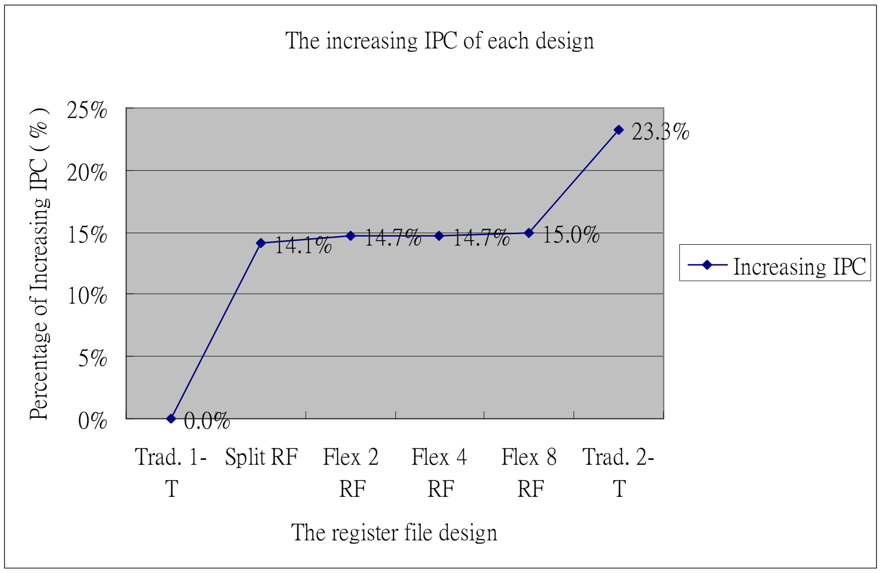


Figure 4-9 Avg. IPC of different flexibilities of register file

In Figure 4-10 we show the ratio of performance improvement over all of our design in 32-entry register file. We can know that with more entries we can get more performance improvement. When we add extra 16 entries in our purposed design, we get slightly performance lose (about 2%) and we still can save over 21.6% area compared with two copies of traditional register file.

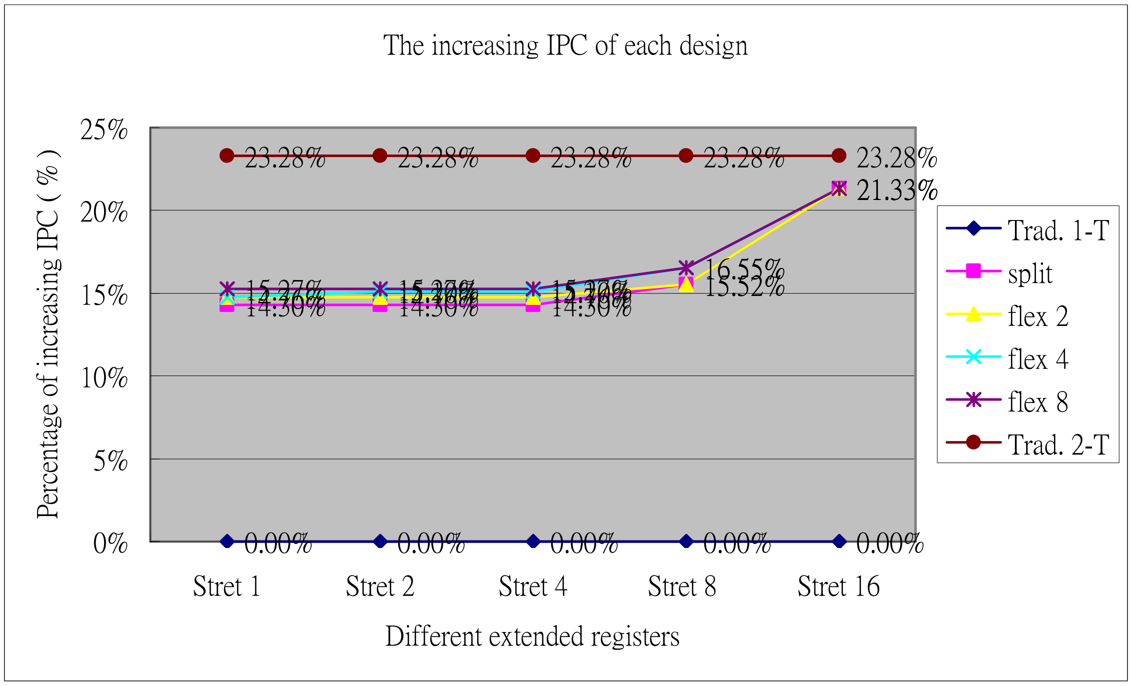


Figure 4-10 Ratio of performance improvement over increased register entry with different flexibility of splittable design

4.7. Hardware Design Issue

When we design a splittable register file, we may have a problem that how many pass transistors (transmission gates) should we put on our design.

In splittable decoder design, if a decoder is n-to-2^n, we set 2*n pass transistors in the decoder. Because an n-to-2^n decoder has 2n control lines in PLA design. In splittable bus design, if a bus is 32-bit, we set 32 pass transistors in the bus.

We take a register file with 32 registers and the bus width is 32-bit an example. We set (10 + 32) pass transistors in the splittable register file. Figure 4-11 shows the number of extra pass transistors setting in different sizes of the splittable register file.

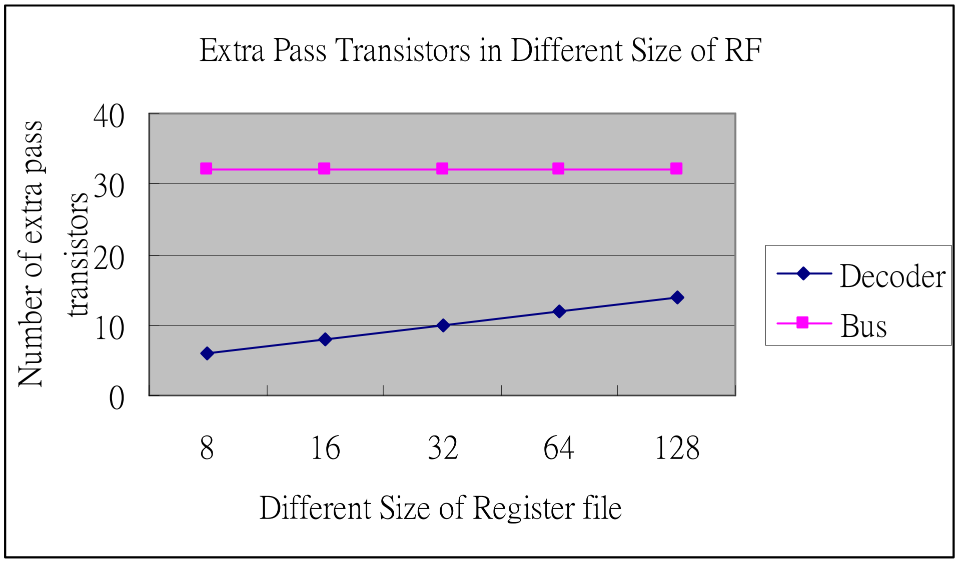


Figure 4-11 The number of extra pass transistors setting in different size of the splittable register file.

In flexibly splittable register file, if the design has N split points then we set N times of pass transistors compared with splittable register file design. If we have two split points, we set 2*(10+32) pass transistors in this register file. Figure 4-12 shows the number of extra pass transistors setting in different split-points of the flexibly splittable register file with 32 registers.

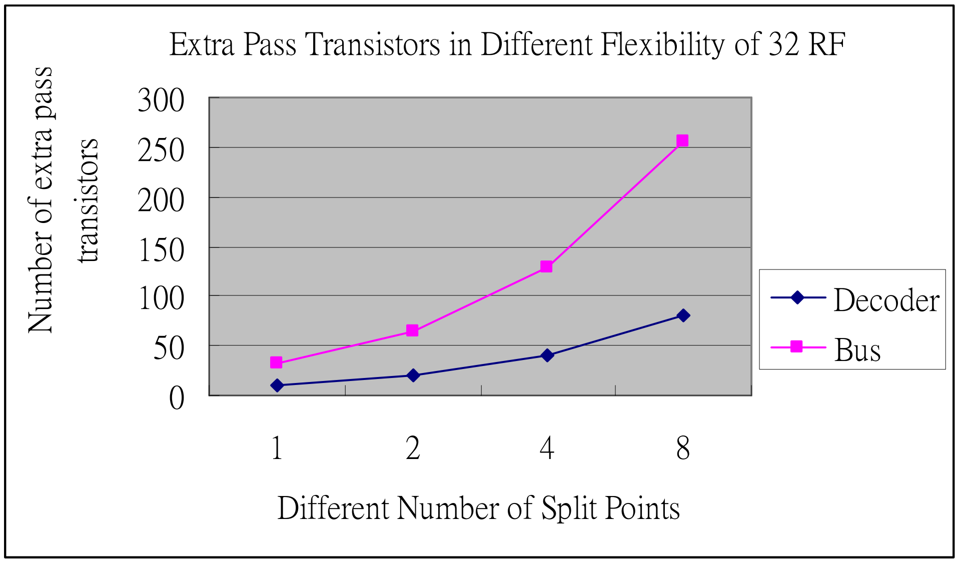


Figure 4-12 The number of extra pass transistors setting in different split-points of the flexibly splittable

register file with 32 registers.

Our purposed design use lots of pass transistors, while the area overhead of these extra circuits does not occupy a significant portion of the area of register file. Even we use the tri-state buffers as pass transistors the area overhead increase 1.7% on traditional register file with 32 registers, 2read/1 write ports.

The main idea of our purposed design try to save the registers of register file, thus we can save area. Traditionally, if we want to execute two threads simultaneously, we will design two times number of registers than one thread executing. We will show our design can use less registers but get performance improving comparing with the same number of registers and even use two times number of registers, we would not lose too much performance. Figure 4-13 shows the overhead and performance of different register file designs. We use a traditional register file (2r/1w) with 32 registers as the standard. We can see that our purposed design does not increasing the much overhead of the traditional register file but increase the performance up to 14.1% and while we stretch 16 registers to our purposed design, we almost attain the performance (lose 2%) of two copies traditional register file.

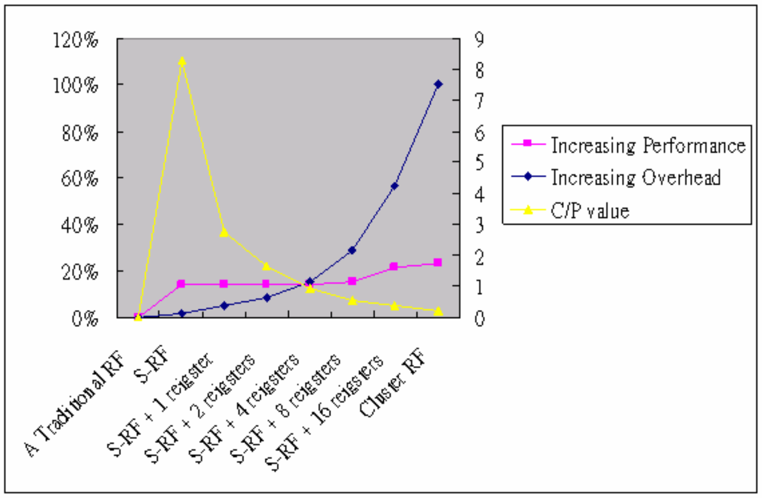


Figure 4-13 The overhead and performance of different register file designs.

Another register file design is the multi-port register file. We double the read/write ports to execute two threads simultaneously with complex control logic, but it also increases the area of the register file. Since our design shares the read/write ports, we do not increase the ports and we still can execute two threads simultaneously without complex control logic. Table 4-5 shows the number of ports with different register file designs. Flexibly splittable and stretchable register file (FSS-RF) is our purposed design, 2T-RF is two copies of traditional register file, it means a thread has its own register file and MP-RF is multi-ports register file.

Register File Design	FSS-RF	2T-RF	MP-RF
Read/Write Ports	2r/1w	2r/1w	4r/2w

Table 4-5 The number of ports with different register file designs.

4.8. Final Proposal of the Register File Design

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

1. The overhead of our purposed design

The ratio of area overhead of our purposed design is reduced when the register file size increase. In 128-entry register file, the ratio of area overhead is about 1.1%. But the ratio of delay overhead is increasing while register file size increase. Since we purpose our design to replace two copies of the traditional register file or twice entries of register file, the delay problem can be relax.

2. The flexibility of splittable register file

According to our hardware and performance simulation, we can see that the area overhead is slightly, but the more flexibility we get the more access time we have. We also observe that the performance increasing in Higher-Lower benchmark set while we have more flexibility. Although it doesn't improve much performance, the flexible design is still useful if the register usage is more equally.

3. The stretchability of splittable register file

Considering the increase performance over linearly increased, to extend 16 entries in 32-entry register file is a good choice. If the we consider the area overhead of the stretchable register file, the splittable register file without stretchability is a choice.

Chapter 5. Conclusion and Future Work

In this work, a splittable register file design is presented for supporting current SMT architecture. The previous chapters have discussed our designs and experimental results. This chapter briefly outlines the conclusion of the work, and provides some directions for future work.

5.1. Conclusion

We show that the splittable register file design provides high performance with little hardware overhead compared with one copy of traditional register file design and the proportion of hardware overhead decreases when the size of RF increases. We assess that the number of allocated split points will be very limited, hence the circuit overhead is low and fixed. To provide high flexibility, we also provide different flexibility of splittable register file design by increasing the number of the split points. We then show the show the relationship between different flexibility and performance improving. Although different flexibility of splittable register file design get little performance improving compared with splittable register file, but according to the simulation result, we can observe that the high flexibility of splittable register file design performs better on high and low register utilization applications running simultaneous.

In the stretchable part, since the invested extra hardware turns into useful registers, this design is entirely cost-effective. We observed that extend extra half size of 32-enrty register file almost get the same performance compared with two copies of traditional register file, Thus we can save almost 21.6% area in two copies of traditional register file. And with the stretching, the discussion about split points remains valid.

These designs are very effective and low-cost. With the multi-threading trends, these designs will find themselves very useful. The same ideas can be extended to other register file designs which can utilize the register file more efficient or save more area without performance down.

5.2. Future work

A Research in our laboratory is purposed to share the ALU, and the shared ALU needs double register read/write ports, our design can resolve the problem. We can try to integrate splittable register file design with shared ALU

We also need an efficient partitioning mechanism for split the register file to make more types of threads joinable and a efficient scheduling mechanism for threads to share splittable register file to increase performance. The scheduling mechanism has to consider profiling-based compiler techniques to register usage of each thread and dynamic scheduling mechanisms to the real register utilization of each thread.

To extend this work, we suggest that related instruction set extension work, application profiling, and layout/timing/area analyses, be undertaken. Particularly, processor architecture supports and interactions need to be investigated.

References

[1] Keith I. Farkas, Norman P. Jouppi, Paul Chow, "Register File Design Considerations in Dynamically Scheduled Processors", HPCA, 1996

[2] SJ Eggers, JS Emer, HM Leby, JL Lo, RL Stamm, DM, "Simultaneous multithreading: a platform for next-generation processors", Micro, IEEE, 1997

[3] J. Tseng, "Energy-efficient register file design", Massachusetts Institute of Technology, 1999

[4] Nam Sung Kim, Trevor Mudge, "The microarchitecture of a low power register file", ISLPED, 2003

[5] J. A. Redstone, S. J. Eggers, H. M. Levy, "Mini-Threads: Increasing TLP on Small-Scale SMT Processors.", HPCA-9, 2003

[6] J. H. Tseng, Krste Asanović, " Banked multiported register files for high-frequency superscalar microprocessors", ISCA, 2003

[7] J. H. Tseng, Krste Asanović, " Banked Register File for SMT Processors", BARC, 2004

[8] C Zang, S Imail, S Kimurat, "Duplicated Register File Design for Embedded Simultaneous Multithreading Microprocessor", ASIC, 2005

[9] J. Sharke. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. Tech Report CS-TR-05-DP01, Dept. of C.S., State Univ of New York at Binghamton, Oct 2005. http://www.cs.binghamton.edu/~jsharke/m-sim

[10] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach 3rd Edition", Reading MA: Morgan Kaufmann, 2002

[11] John P. Shen, Mikko H. Lipasti, "Modern Processor Design: Fundamentals of Superscalar Processors", Reading MA: McGraw-Hill, 2004