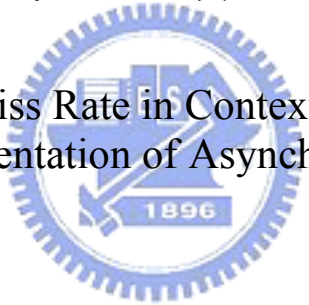
國立交通大學

資訊科學與工程研究所

博士論文

低內容交換失誤率之轉換搜尋緩衝器
與其非同步電路實作之探討

TLB with Low Miss Rate in Context Switching and
Study of Implementation of Asynchronous Circuit



研 究 生：鄭緯民

指導教授：陳昌居　教授

中 華 民 國 九 十 八 年 七 月

低內容交換失誤率之轉換搜尋緩衝器與其非同步電路實作之探討

TLB with Low Miss Rate in Context Switching and Study of Implementation of Asynchronous Circuit

研 究 生：鄭緯民　　Student：Wei-Min Cheng

指導教授：陳昌居　　Advisor：Chang-Jiu Chen

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

博 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

Doctor of Philosophy

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

低內容交換失誤率之轉換搜尋緩衝器與其非同步電路實作之探討

研究生：鄭緯民　　指導教授：陳昌居 教授

國立交通大學資訊學院資訊科學與工程研究所

摘　要

嵌入式處理器廣泛運用在嵌入式系統或手持裝置中，因此低功率、可靠與扎實就成為這類處理器最重要的課題。非同步電路應該是解決這類問題最好的解答，因此，非同步電路非常適合用來實作這些處理器。

眾所周知，這些嵌入式處理器被用來執行許多不同的工作。近來，許多嵌入式系統與手持式裝置開始執行非常複雜的作業系統，像是嵌入式 Linux 或 Windows® mobile, 而為了支援現代作業系統的虛擬記憶體機制，支援虛擬位址與實體位址間的轉換是必需的，這也被認為是影響整體記憶體系統效能的關鍵因素之一。為了提高位址轉換的效能，幾乎所有近代處理器內都具備了轉換搜尋緩衝器，因此，在我們的計畫中，我們提出了一個設計給嵌入式處理器具備低內容轉換失誤率的轉換搜尋緩衝器架構。為了區隔不同的位址空間，我們採取了區隔轉換搜尋緩衝器庫來取代每個轉換搜尋緩衝器項目的位址空間區隔標籤，並且使用了簡單的預取機制來減少可能發生的強迫性失誤，除此之外，因為是設計給非同步嵌入式處理器的轉換搜尋緩衝器，設計上所有的運作行為也都很簡單。

最後，我們以 Balsa 硬體描述語言實作該轉換搜尋緩衝器之控制器，因為實作過程中我們巧妙的安排了通訊交換管道，實作過程中我們就可以比較容易以假設的輸入樣本驗證正確性，儘管也許以這樣的方式驗證我們這個實作是可行的，然而這種方法運用在我們目前進行中的非同步嵌入式處理器計畫既不可能也不合理，因此我們提出建議了一個未來我們計畫進行中的軟硬體共同設計與交互驗證的流程，最後，以 Balsa 工具產生了邏輯閘級的 netlist 也評估了實作所需等效的邏輯閘個數，然而結果顯示如此方式實作成本並不低，等價邏輯閘數為 688,560, 我們也說明了還是依然以此高層次非同步硬體描述語言實作的原因，最重要的，這對未來比較大的非同步設計而言是必需的。

TLB with Low Miss Rate in Context Switching and Study of Implementation of Asynchronous Circuit

Student：Wei-Min Cheng **Advisors：Prof. Chang-Jiu Chen**

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Embedded processors are widely used in many embedded systems and handheld devices. Hence, low power, reliability, and robustness have been becoming the critical issues for these processors. Asynchronous circuits may be one of the best solutions to overcome these problems. Thus it may be more suitable to implement these processors with asynchronous circuits.

It is widely known that these embedded processors are used to execute varieties of tasks. Recently, many new embedded systems and handheld devices begin to execute very complex operating systems, such as embedded Linux or Windows® mobile. In order to support virtual memory mechanism of modern operating systems, address translation from virtual address to physical address should be supported. However, it is widely considered as the critical issue of memory system performance. In order to improve the address translation performance, the Translation Lookaside Buffer (TLB) is implemented inside

almost all contemporary processors. In this work, we propose an alternative TLB architecture with low context switch miss rate for asynchronous embedded processors. We adopted a heuristic TLB banking designs to replace per-entry ASID to identify each address space. In addition, simple prefetching mechanism is used to reduce some possible compulsory misses. Because the architecture is designed for asynchronous embedded processors, all operations are very simple.

Finally, we implemented the TLB controller for the proposed TLB architecture with Balsa HDL. Because we skillfully arrange the communication channels, we can verify the implementation easier with assumed random pattern. Though it's possible to verify our implementation with such simple way, it's impossible and unreasonable to verify the whole asynchronous embedded processor that we are currently working for. We also suggested a hardware/software co-design and cross-verification flow for our future work. Finally, the gate-level netlist was generated with Balsa tools, and the equivalent gate count of the implementation was estimated. The result shows that the cost of the implementation modeled with Balsa HDL is not cheap. The total equivalent gate count is 688,560. However, we also describe why designing asynchronous circuits with such high-level asynchronous HDL. It's needed for future larger design!

Acknowledgement

從入學就讀博士班到現在，終於要離開交大了，多年來在這裡生活感觸很多，也很感謝很多老師與身邊來來去去的許多人的幫助與陪伴，在交大真的有許多回憶，在我的生命的歲月中彌足珍貴與重要，因此在這邊有很多感謝想說。

這些年最想感謝的是多年來一直照顧我幫助我的指導教授陳昌居教授，因為他的幫助與照顧，讓我在這些年受益匪淺，也學到不少道理，擴張我的視野，大同大學資訊工程學系鄭福炯教授與實驗室黃年畤學長，因為他們的引介，我才能夠瞭解除了傳統同步電路的設計以外，還有很廣闊的非同步領域的天空，此外，多年來在實驗室也一直受到黃年畤學長的照顧與協助，而且還能夠在學業以外，有更多活潑的室外休閒活動，讓我不再是完全關在室內的「宅男」，當然，也是許多實驗室伙伴大家共同改變我，此外，在交大資工系修習不少課程，因為系上教授殷殷指導，讓我學習到更多知識與技能，系上老師親切的態度更讓我感動，像是每每遇到鍾崇斌教授，總能感受到他的噓寒問暖，讓我真的很感謝，也感謝我的論文計畫書口試教授陳正教授與范倫達教授，陳正教授除了過往在他的課堂上學到很多，對我的論文方向也給予了許多寶貴建議，至於范倫達教授雖然未曾有機會修習到他的課程，但是這兩三年因為領域關係，實驗室學弟口試總能遇到他，他也常能對我們實驗室研究有不少建議，對我的論文也提出珍貴意見，十分感謝，此外，實驗室學弟張元騰、蔡宏岳更是給予許多支持與幫助，讓我計畫進行順利，至為感謝，還有已經畢業的張繼文學弟，除了他跟我現在是工作上很要好且業務關係密切的同事以外,他在有關 TLB 與 MMU 和現在有關 ESL 的研究上更是與我最好的合作伙伴，也是讓我萬分感謝，此外，其他還有很多要感謝的實驗室伙伴，不管是畢業的或是還在學的，真的，對於不管是曾給予我幫助的系上老師或是學弟妹與伙伴，真的有很多話想說，心中萬千感受，實在千言萬語，難以言盡，只能對所有師長、同學與學弟妹在我心中永遠感謝，謝謝大家。

Contents

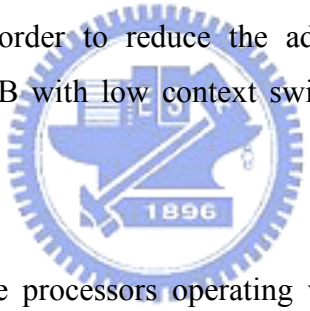
List of Figures

List of Tables

Chapter 1: Introduction

1-1 Motivation

Embedded processors and microcontrollers are widely used in varieties of different embedded systems and handheld devices. Because of new complex applications today, these processors are now required to execute new embedded operating systems. Thus it's very important to provide the capability to support virtual memory mechanism needed in modern operating system. In order to provide fast address translation, the translation lookaside buffer (TLB) should be provided inside these processors now. Furthermore, because of the embedded system or handheld devices nature, simple and easy context switching model should also be provided. In order to reduce the address translation penalty of context switching, a well-designed TLB with low context switching miss ratio is needed for these processors [1,2,3].



In addition, to keep those processors operating with high robustness and low power consumption are the two most important issues. It is widely known that asynchronous circuit is the best solution to address these two issues at the same time [4,5,6,7]. Thus embedded processors and microcontrollers may be suitable to be implemented with asynchronous circuits. However, it's not very easy to implement the TLB that needed for modern operating system for asynchronous processors. In our work, we proposed TLB architecture with low context switching miss ratio that is suitable for embedded systems that runs only some tasks and implement the TLB controllers with asynchronous circuits.

1-2 Introduction to asynchronous circuits

Asynchronous chips improve computer performance
by letting each circuit run as fast as it can!

By Ivan E. Sutherland and Jo Ebergen
"Scientific American", August 2002 [4]

It is widely known that synchronous circuits have some problems that have to be carefully dealt with, such as clock skew, difficulty in clock distribution, worse case performance, not modular, sensitive to variations in physical parameters (temperature, voltage, and process), synchronization failure, and noise (EMI). All these problems derive from the "clock" signal [4,5,6,7]! As the VLSI based systems become larger, more complex, and work with higher clock rate, these problems also become more serious than ever before.

In addition, to reduce the power consumption has already become one of the most important issues in large VLSI system design. It is widely known that the dynamic power dissipation $P \propto fcv^2$[8]. That means that the dynamic power consumption is in proportion to the number of switching activities. In order to improve the circuits or system performance, the clock frequency becomes higher and higher. Thus, the extra power wasted in the clock tree distribution also becomes larger and larger. That's very clear that clock signal consumes a very large proportion power of the whole chip. For example, the clock tree distribution network of DEC (Compaq) Alpha 21064 processor consumes about 40% power when it runs at maximum speed [9]. Similarly, the Motorola MCORE micro-RISC processor consumes 36% power in clock tree distribution [10]. In fact, the clock distribution network should be responsible for an increasing fraction of the dynamic power consumed by modern processors and SoCs [11,12,13]. Thus, if the clock signal can be removed, the power consumption may be reduced with very high possibility. In order to reduce the power consumption, lots of different techniques are proposed and implemented, such as clock gating and dynamic voltage and frequency scaling (DVFS)[14]. Furthermore, higher clock frequency may also cause the temperature of the VLSI chips very high. It's also harmful for embedded systems or handheld

devices. We can say that all these problems cause nightmares for almost all VLSI-based system developments today.

On the contrary, asynchronous circuits can easily reduce the power consumption via removing the "clock" signals that spread the whole VLSI chip. Replaced by the handshaking protocols, asynchronous circuits offer low active power and almost zero standby power [4,5,6]. In fact, because of data-driven nature, the inactive components or parts of asynchronous circuits can be automatically "shut-off." Thus, asynchronous circuits can offer very good power efficiency. For example, the most famous asynchronous ARM compatible processors—the Amulet series processors [15,16,17,18] shows very good power efficiency than its synchronous ARM processor counterparts. Another very famous example, Philips asynchronous 80C51 microprocessor is 4 times power efficient than that of its synchronous counterparts [19]. The most interesting of all is the latest ARM996HS processor that is the first commercial-available synthesible 32-bit CPU built with clockless logic[7,20]. It consumes about 2.8x less power than the clock-gated ARM968E-S core. Table 1-1 shows the comparisons between ARM996HS and ARM968E-S [7]. The table also shows that ARM996HS can operate correctly in varieties of operating environment. It can operate with lower voltage in high temperature environment. Asynchronous circuits are much more robust than synchronous circuits.

In fact, designing the "clock" system has been becoming the critical issue in large VLSI system design today. For example, very complex "clocking architecture" is implemented in the latest Intel® 45*nm* 8-core Xeon® Enterprise processor announced in ISSCC2009 [21]. Figure 1-1 shows its clocking architecture. The design has totally 16 PLLs, 8 DLLs, and independent clock domains for each cores and the uncore. What a complex design it is! Unfortunately, such designs are very popular today. Since the first microprocessor, the Intel® 4004, was announced in 1971, the VLSI technologies have had great improvement. To put one billion transistors on a single chip have been becoming possible. How terrible it is to design the "clock" system on such big system!

However, because of several complex historical and practical reasons, almost all systems today are still implemented with fixed clock period based design. While synchronous design

may introduce lots of problems with systems growing up larger and larger, asynchronous design may overcome these problems via avoiding the use of clock signal. Furthermore, how to accomplish IP reuse easier becomes one of the most important issues for SoC design. Asynchronous circuits may be one of the best solutions to address this issue. Without the influence of the "clock" signal, asynchronous circuits make "software OOP" style design for hardware design possible. All things that the designers need to know are the handshaking protocol interface [4,5,6]. It also makes each designed component or IP more reusable. With growing up mobile device and embedded system markets, all these issues need to be seriously considered. Thus, it's time to implement these systems with asynchronous circuits.

Table 1-1: Comparisons of ARM996HS and ARM968E-S

	Frequency [equiv. MHz]	Performance [DMIPS]	Power [mW/equiv. MHz]	Gate Count [NAND2 equiv.]
ARM996HS	50 (worst, 1.08 V, 125ºC) 77 (nominal, 1.2 V, 25ºC)	54 (worst, 1.08 V, 125ºC) 83 (nominal, 1.2 V, 25ºC)	0.045 (nominal, 1.2 V, 25ºC)	89K
ARM968E-S	100	107	0.13 (nominal, 1.2 V, 25ºC)	88K

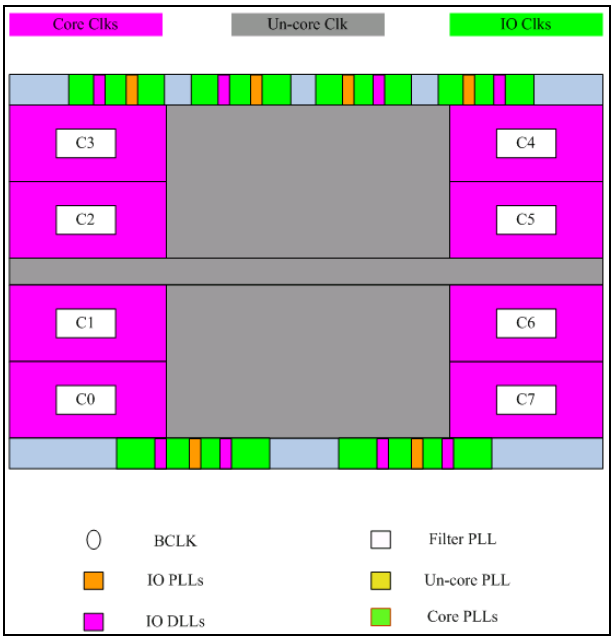


Figure 1-1: Clock distribution domains and generators

1-3 Introduction to Translation Lookaside Buffer

In order to support larger memory requirements for modern applications, it's important for modern operating systems (OS) to provide the virtual memory mechanism. Conceptually, with virtual memory, the movements of code and data of one program between main memory and secondary storage can be automatically achieved, and a single complete and contiguous "memory space" can be given for each program. Thus, only part of code and data of one program needs to be placed in main memory. Programmers do not need to know anything on how the code and data are arranged. Moreover, the program size can be even larger than the real physical memory size. In fact, virtual address space are often much larger than real physical memory space and size. Figure 1-2 shows the conceptual virtual memory. The virtual memory is divided into lots of fixed size blocks called pages and each page has a specific page number called Virtual Page Number (VPN). Similarly, the physical memory is also divided into the same size page frames, and each of it has its own unique page frame number called Physical Page Number (PPN). Via the memory mapping, each page of virtual memory can be mapped to a page frame of physical memory or the secondary storage. With appropriate hardware support, the virtual memory is carefully maintained by the OS [22].

As mentioned before, the OS is responsible to provide the mechanisms to map virtual address to physical address. However, all these virtual address to physical address translations are stored in main memory. To reduce the cost of address translations, the translation lookaside buffers (TLBs) are widely implemented inside the processor [23,24,25,26,27,28]. Figure 1-3 depicts the basic design idea of TLB. Once the virtual address (VA) is sent to TLB, it is compared with all the tag fields to find a matched VPN. If it is a hit, the corresponding PPN will be sent out. The physical address therefore can be generated via the combination of PPN and offset. Otherwise, if it is a miss, the page table traversal will be performed. The OS will take care of the TLB miss handling.

But, the virtual memory mechanism varies with different processor architecture and OS implementation. The page table organization dominates the page table traversal time that

occupies most TLB miss handling time. Though some new architectures use some advanced page table organizations to reduce the page table traversal time such as inverted page table structure [22] such as PowerPC architecture[29], the forward-mapped hierarchical page table structure are still widely used, such as Compaq/DEC Alpha AXP[28], the latest AMD64, and Intel®64 [30,31,32] architectures. It costs several main memory accesses to fetch the correct Page Table Entry (PTE) if any miss occurs. It even possibly needs to traverse 7 levels of different page tables on processors with 64-bit addressing [33]. Figure 1-4 shows the page table structure of IA32e mode with 4KB page size of Intel®64 architecture. If no any TLBs and address caches are implemented inside these processors, traversals of four levels of different tables should be completed to obtain correct PPN. Figure 1-5 shows the page table structure of Compaq/DEC Alpha AXP [28]. It has three levels of page tables. That impacts the overall system performance very seriously. Thus it's important to reduce the TLB miss rates for systems with such page table structure.

In addition, frequently happened context switching may cause some extra TLB misses. Some research even shows that these misses play important role in TLB performance [1,2,3]. Thus most processors have implemented some kinds of address space identifier (ASID) to distinguish each address space [25]. For example, MIPS R10000 processor has an 8-bit ASID for each of its 64-entry TLB to allow context switches without having to invalidate all entries [34]. It is also suggested to provide 8-bit ASID for SPARC architecture [35,36]. However, some processors including the IA32 (*x86*) architecture which is the most popular processor family today simply flush all the TLB entries when the context switching occurs [31,1]. Unfortunately, it's even still the same for the latest IA32 processors. We'll treat the model as the worse case performance. Though lots of different research about TLB has been done, only some notice the influence of context switching. That may be because it's very hard to model and estimate the context switching activities caused by the OS and it's also hard to consider this issue without considering the OS behavior first. In our work, we tried to provide an alternative to address the context switching issue for TLB. To support the proposed mechanism, the OS should be modified a little. In fact, because of architecture differences, these kinds of modifications of OS for TLBs are needed for all architectures. We hope that this simple mechanism can be implemented inside an asynchronous embedded processors or microcontrollers that only run some tasks simultaneously.

To estimate the performance of the proposed architecture, we did some simulations. All the simulations were done by the modified SimpleScalar Version 3.0d tool suite [37] provided by the SimpleScalar LLC with SPEC95. In addition to the performance of traditional 1024-entry fully-associative TLB with *x86*-style assumption, we also compare the performance of 1024-entry fully-associative TLB with ASID and two different pre-fetching mechanisms incorporate with our proposed design. The results show that our banked design can work very well with sequential prefetching (SP, also called linear pre-fetching).

Our work is trying to realize TLB controllers for asynchronous embedded processors or microcontrollers with low TLB miss rate caused by context switching. Though most processors reduce the miss rate caused by context switching with ASID, our work provides an alternative to address this issue. There are several reasons for the proposed architecture. These embedded systems only execute some tasks at the same time. Thus, it really doesn't need to store too many ASIDs. That's why no ASIDs TLB design of StrongARM SA-1100 processor [2,3]. Don't forget these processors are not designed for desktop purpose. Figure 1-6 shows smart phones that execute Windows® mobile OS. In fact, because we wish to implement such TLB for asynchronous embedded processors or microcontrollers, less tag bits may be more important than other issues. In addition, we also discuss why sequential prefetching is more suitable for the proposed design. Moreover, we'll try to realize this design on the asynchronous processor which we currently work for. That would not be too hard to realize the proposed mechanism on an asynchronous processor with same extra handshaking protocols on bundled delay or self-timed design.

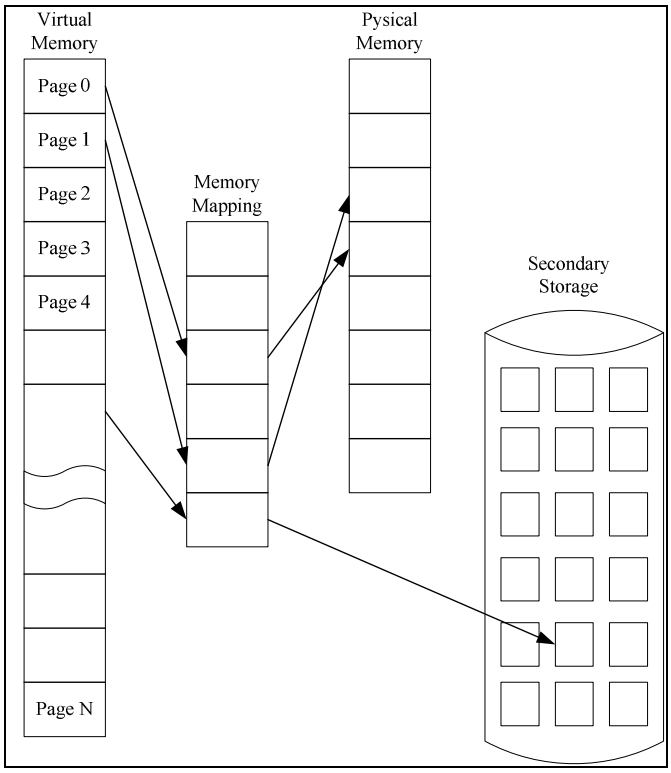


Figure 1-2: Conceptual virtual memory

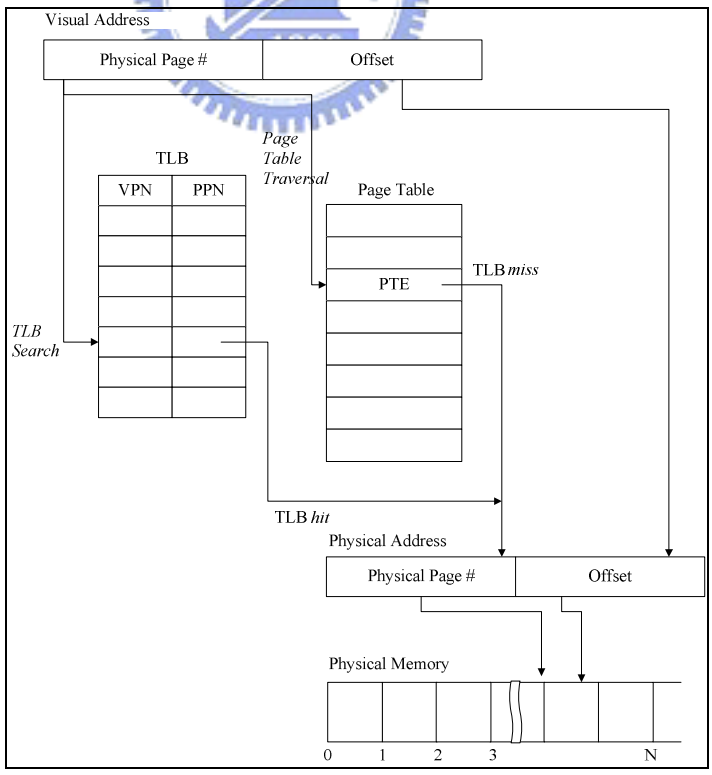


Figure 1-3: Virtual address translation with TLB

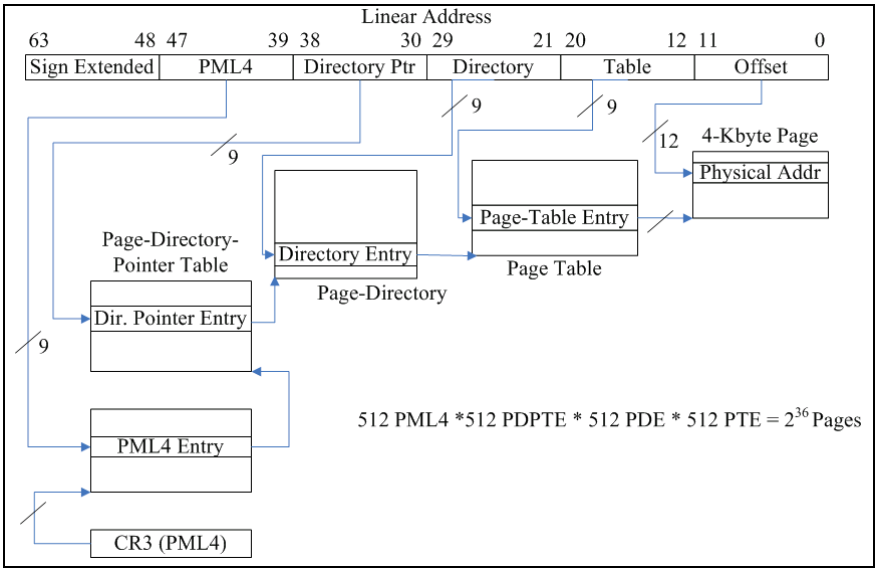


Figure1-4: Page table structure of IA32e mode with 4KB page size

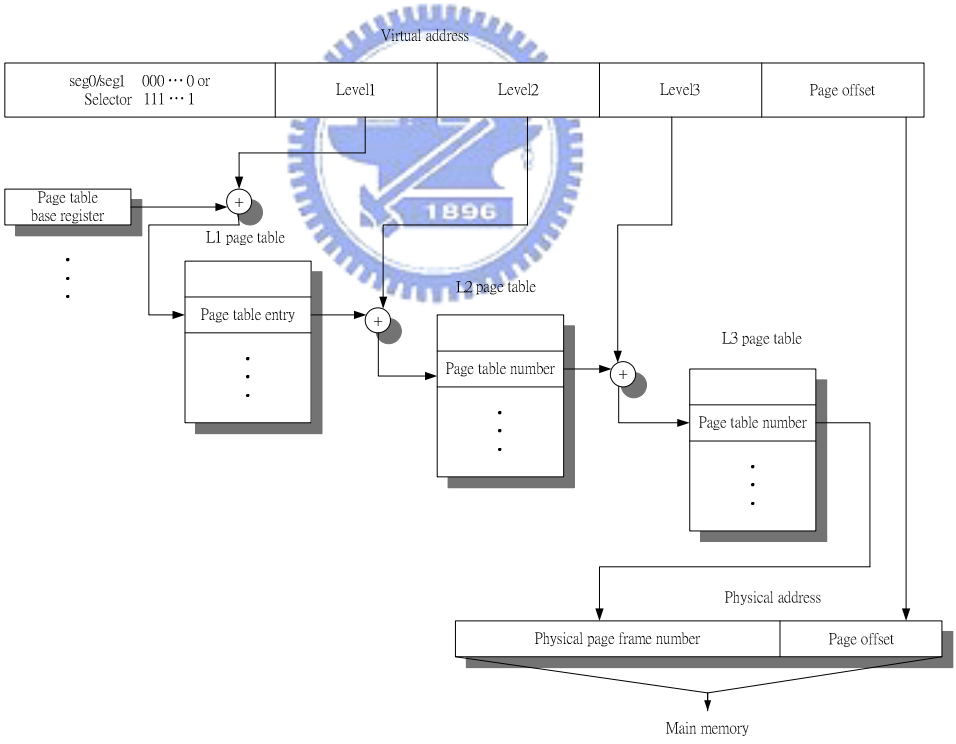


Figure 1-5: Virtual to physical address translation of Alpha AXP

Figure 1-6: Smart phones with Windows® Mobile OS

1-4 Summary

As mentioned in previous section, new embedded systems or handheld devices now begin to execute new modern operating systems. It therefore becomes more and more important for these processors to provide efficient address translations. A well-designed TLB now becomes one of the critical performance issues for these processors. In addition, because embedded systems and handheld devices may operate in varieties of environments, robustness and reliability are two of the most important issues to these processors. Asynchronous circuits can easily address these issues. However, lack of address translation mechanism, most asynchronous processors doesn't support virtual memory directly. In order to support virtual memory for asynchronous processors, asynchronous TLB controller should be implemented. Thus, in this thesis, we propose a TLB architecture for future asynchronous embedded processors, and modeled it with Balsa HDL. Followings are the main contributions of this thesis.

- Plenty surveys of TLB studies
- Plenty surveys of asynchronous circuits, and detailed introductions of how to design circuits with asynchronous circuits
- Studies of performance issue of TLB in context switching
- New alternative TLB architecture with low miss rate in context switching for asynchronous embedded processors
- Studies of implementation of proposed TLB architecture with asynchronous circuits
- Confirming the possibility to design TLB controller for asynchronous processors

Chapter 2: Related Works

In this chapter, we'll discuss the related works of both TLB design and asynchronous systems or circuits design. Because only a few specific research on TLB design for asynchronous processors, we'll discuss them separately in this chapter. Finally, case studies of asynchronous MMU or TLB design will be discussed.

2-1 Recent studies of TLB

As mentioned in Chapter 1, TLB plays an important role in the overall performance of the processors that support virtual memory technique. Thus, lots of different research has been done. Moreover, because of architecture and addressing mode differences, the real implementation may have great differences. The design requirements may even vary from different page modes or new addressing mode support for the same processor. However, that's quite interesting that the TLB designs of most real commercial processors are not too complex. Most of them are not implemented with too complex algorithms or architectures. The key issue of these designs is to reduce the TLB search time. Some related works of TLB research will be described in the following paragraphs. These works will be classified into traditional techniques, advanced techniques, and works of reducing TLB context switching miss rate.

2-1-1 Traditional Techniques

Because TLB in fact is part of the memory hierarchy and can be considered as a special designed cache memory to cache the page table entry, it can be directly perceived through the senses that those traditional techniques to improve the cache performance can also be applied to TLB. That also means the 3Cs misses [28] can be also suitable for the TLB. In fact, those techniques are now widely used in commercial processors in different ways.

In order to reduce the TLB miss rate, most processors increase the size (total entries) of TLBs with fully or set associative. For example, recent AMD Opteron™ processor has both 512-entry L2 instruction TLB (ITLB) and L2 data TLB (DTLB) [38] and the IBM POWER4 processor has a common 1024 entry TLB for each processor core [39]. Furthermore, some processors even try to provide multi-level TLBs, such as 2-level ITLB/DTLB design on recent AMD Opteron™ processor [38] and each core (*Nehalem* architecture) of the latest Intel®Core i7® processor [31]. Figure 2-1 shows the TLB designs of the Intel®Core i7® processor. Each core of the processor has separated the Instruction and Data TLB with a unified Second-level TLB (STLB). In addition, some processors begin to provide larger page sizes to increase the TLB span, such as 2MB or even 4MB page size on all new Intel IA32 Processors after the Pentium® Pro Processor [40]. The Intel IA64 architecture offers 4K to 256MB and 4GB page sizes [41]. The AMD64 architecture also provides 4KB, 2MB, 4MB, and incredible maximum 1GB page sizes [42]. There are several advantages of larger page sizes. First, because the page table entry can be reduced, it can save the page table sizes. Second, it allows for larger physically addressed caches. Third, because each page can map larger memory spaces, fewer page tables and TLB entries can be used. Finally, because the level of page tables can be decreased, the fewer accesses to main memory are needed to generate correct physical page number if TLB miss occurs. Figure 2-2 shows the page table structure of IA32 mode with 4KB page size of IA32 architecture, and Figure 2-3 shows the page table structure of IA32 mode with 4MB page size of IA32 architecture [31]. We can easily find that with larger page size the levels of page tables can be decreased.

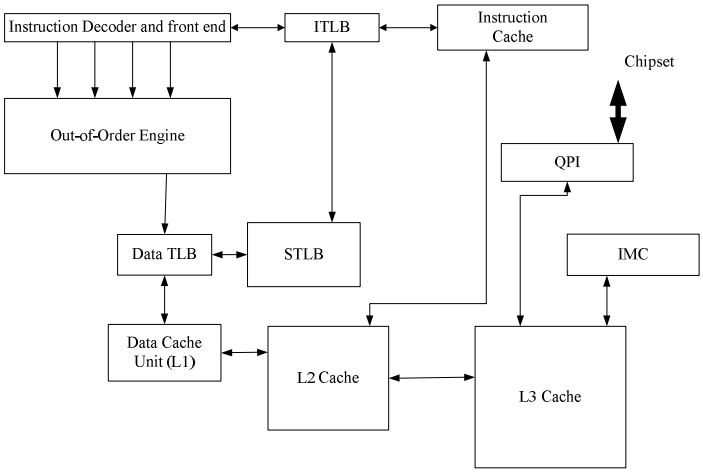


Figure 2-1: Structure of TLBs and cache memories of Intel® Core i7

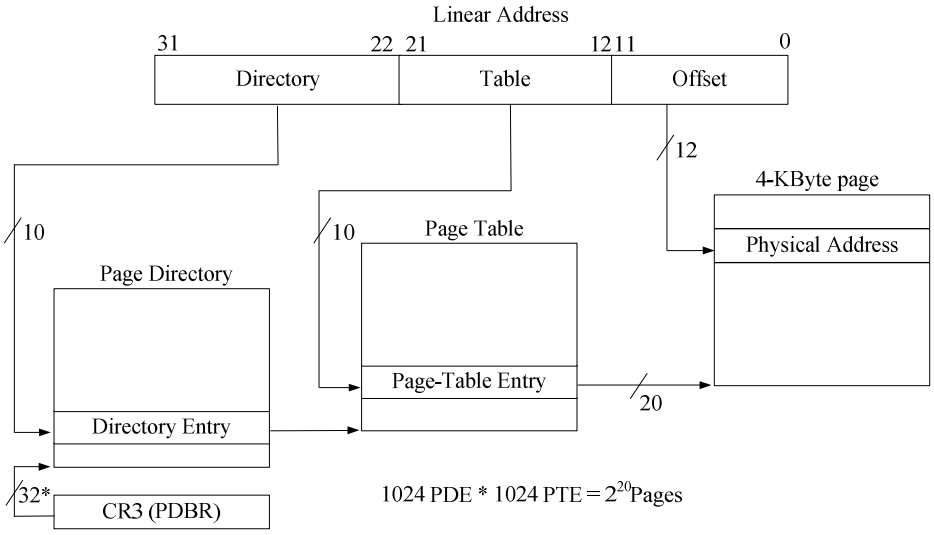


Figure 2-2: IA32 Linear address translation (4-KByte page)

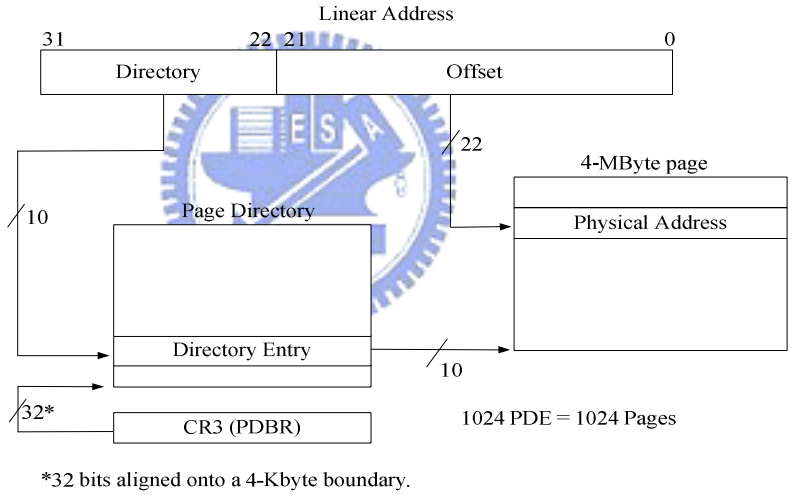


Figure 2-3: IA32 linear address translation (4-MByte page)

2-1-2 Advanced Techniques

As mentioned in previous paragraph, most contemporary processors now provide some different page sizes from 4-KB size to incredible very large sizes. Some even allow these pages with different sizes coexist simultaneously with some augmented page table entry format. Certainly, it needs extra supports of OS. In fact, with small page size, the memory space can be saved. That's because with larger page sizes, memory spaces would be wasted

due to the internal fragmentation. In addition, with small page size, the startup time of small program would be shorter. However, to provide several page sizes, some commercial designs put several TLBs inside the processor for each individual size. Some try to modify the TLB entry format and therefore the TLB can be shared with different page sizes.

In addition to what we mentioned in previous paragraph, several interesting mechanisms are proposed to support superpaging. Several base pages with both virtual and physical address alignment can be merged into a larger page called superpage at run time [43,44,45,46]. With superpage mechanism, the internal fragmentation problem can be resolved. However, to support a superpage, very complex OS and hardware interactions are needed. Furthermore, the virtual and physical memory space aligned limitation seriously impacts the usage of a superpage. Hence, some studies have focused on overcoming the limitation by dynamically supporting the superpage mechanism. Talluri et al. described an advanced method called the complete-subblock which allows a single TLB block to map to multiple base pages without any special OS support [43,44]. In addition, they also described a much smaller design called the partial-subblock which shares PPN and attribute fields across base page mappings. Figure 2-4 shows a complete-subblock TLB block (entry) with factor 4. Lee et al. proposed a novel banked-promotion TLB structure to support two page sizes dynamically [47]. Four 4KB pages can be promoted to a 16KB superpage. To support such mechanism, an interesting promotion TLB was designed. The heuristic promotion algorithm can promote four consecutive entries from small-page TLB bank to large-page TLB bank. Thus, the four 4KB TLB entries can be reused. Furthermore, in order to reduce the power consumption and TLB reference latency, they even divided the TLB for 4KB page into two banks [48]. Figure 2-5 shows the structures of their promotion TLB and banked-promotion TLB. In addition, Swanson et al. presented a novel memory controller (MMC) which can aggressively create superpages even from non-contiguous and unaligned regions of physical memory space [49,50]. Figure 2-6 depicts this design. In this design, they suggested to use a portion of unused physical memory address range to virtualized physical memory in their proposed MMC. The shadow pages are "*shadow*" of accessed page that can be remapped to real physical address by MMC. The TLB reach can be extended via a novel Memory Controller TLB (MTLB). Thus the superpage can be aggressively created from non-contiguous and nonaligned regions of physical memory. Park et al. proposed a way to integrate both partial-subblock with MMC to improve TLB performance [51]. They also

proposed a method called Variable-Size Subblock TLB (VS-TLB) which is an extension of original subblock TLB to support multiple size subblock. Based on the original subblock TLB design, they added subblock size field (SS) for each entry. With this extension, the total TLB reach can be increased via its maximum subblock size. There is still much research about improving TLB performance of superpaging.

Besides previous research, some different and interesting research can also be found. Channon et al. presented the reconfigurable partitioned TLBs to improve the TLB performance [52]. They claim that traditional split instruction and data TLB design is not suitable for unpredictable memory reference pattern. Thus the reconfigurable partitioned TLB can reduce misses between distinct reference types. The reconfigurable partitioned TLB can dynamically adjust the position of the partition in real time. Figure 2-7 shows this design. In addition, some research focus on the low power issue. Besides some architecture improvements to reduce power consumption such as baking skills, some even try to redesign the basic circuit element itself. For example, Juan presented low power CAM and SRAM cells design that can be implemented [53]. They also studied the relationship of power consumption and associativity of TLB. They concluded that small TLB with fully set-associative and implemented with modified cell can save more power. Because TLB is part of memory hierarchy, some research tries to integrate both TLB and cache memory. Among all of these studies, Lee et al. proposed an interesting way to reduce the tag memory of cache memory [54,55]. The design uses share tag memory of both TLB and cache memory. They still use CAM as the tag memory for TLB. However, the cache memory shares the same tag memory. The index tag memory of cache now only stores encoded index of an entry in shared tag memory rather than the PPN. Thus, the total tag memory sizes can be reduced. Figure 2-8 shows this design. In addition to these hardware efforts, lots of different software efforts can be found. Instead of hardware managed TLB, software management TLBs are widely used in lots of new RISC processors, such as SPARC, Alpha AXP, PA-RISC and MIPS architectures [23,25]. In fact, there are still varieties of different studies of TLBs and virtual memory.

Though lots of new TLB designs are proposed, just only a few studies focused on the TLB entries prefetching/preloading. Saulsbury introduces an interesting mechanism, called the Recency-based TLB Preloading (RP), to prefetch the TLB entry according to the

"Recency" of the referenced pages [56]. The mechanism maintains the "Recency Stack" via augmented translation table entry in memory and the TLB inside the processor according to the recently referenced pages. Thus the next possible referenced page number can be prefetched. Figure 2-9 (a) shows how the stack changes inside the processor if the TLB reference is a hit. Because it's a TLB hit, the recency of all translation table entries (TTE) of the translation table will not be changed. Figure 2-9 (b) depicts how the "Recency Stacks" of both TLB and translation page table change if the TLB reference is a miss. After the missed TTE is moved to the top of TLB stack, the recency of both TLB entries and the translation table entries will be changed according to the recency stack position. Finally, the TTE with "recency ± 1" of missed TTE can be prefetched into the prefetch buffer inside the processor. It should be noted that in real implementation all the TTE positions of "Recency Stack" are maintained by the previous and next pointers of each TTE. Figure 2-10 shows the implementation of the translation table in memory. However, the mechanism may increase the memory traffic and the PTE should do some changes to store the stack pointers for the link-list. To solve these possible problems, Kandiraju proposes a new prefetching technique, called the Distance Prefetching (DP), according to the recently referenced pages 'distance (stride)' [57]. The mechanism maintains a table to keep the track of differences between successive address references and do prefetching according to the predicted distance. Figure 2-11 shows the implementation of TLB with DP technique. The paper also shows a generic schematic prefetching hardware and compares other possible prefetching techniques borrowing ideas from the cache prefetching techniques, such as Sequential Prefetching (SP), Arbitrary Stride Prefetching (ASP) and the Markov Prefetching (MP). Figure 2-12 shows the schematic of generic prefetching hardware. Because of the implementation costs, we'll focus on the studying of the SP and DP in our work.

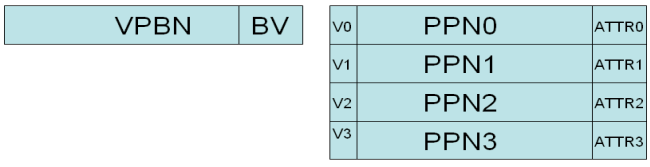


Figure 2-4: Complete-subblock TLB with block factor 4

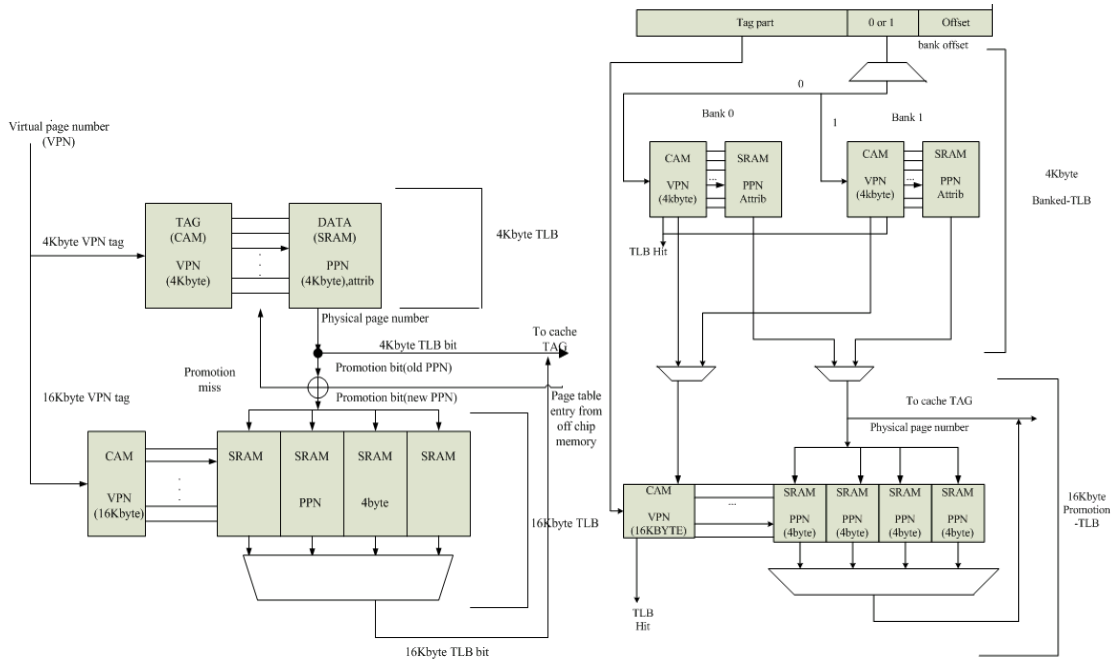


Figure 2-5: Promotion TLB structure & Banked-promotion TLB structure

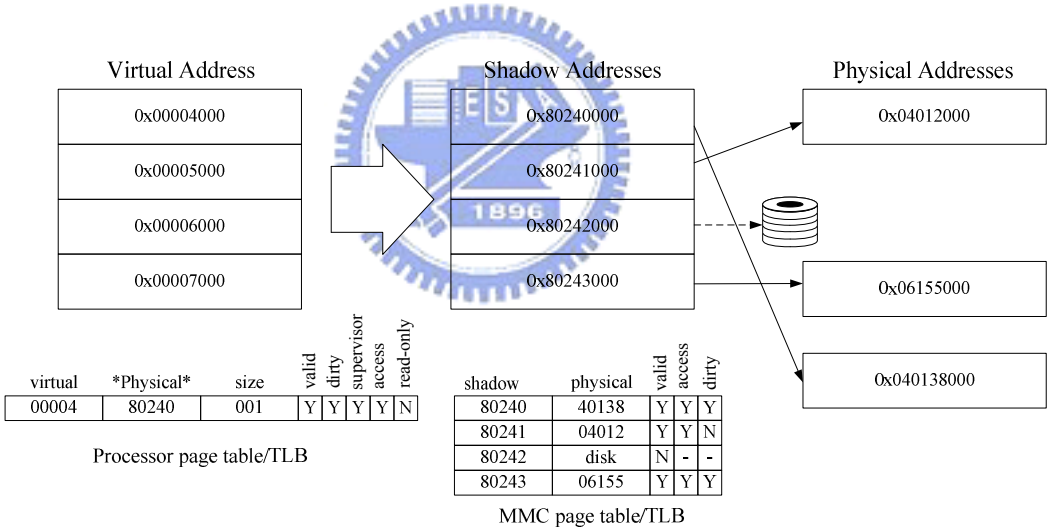


Figure 2-6: MMC example with shadow region

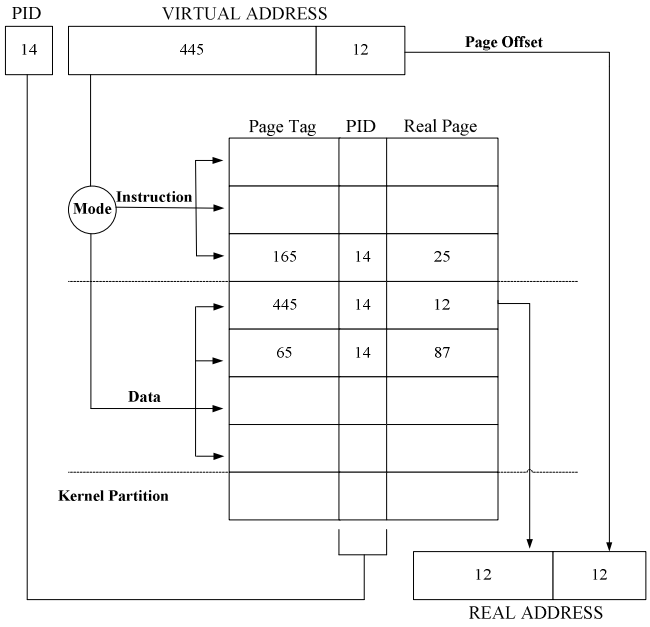


Figure 2-7: Reconfigurable partitioned TLB

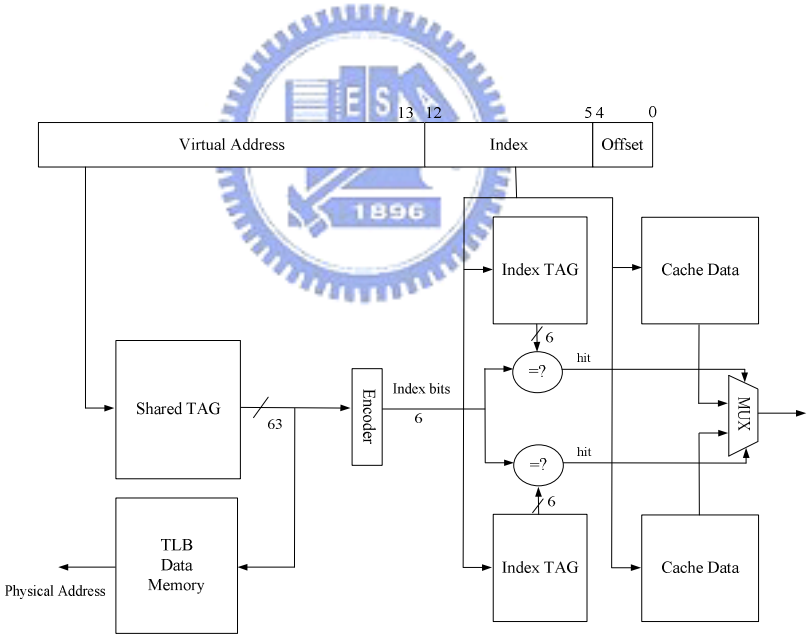
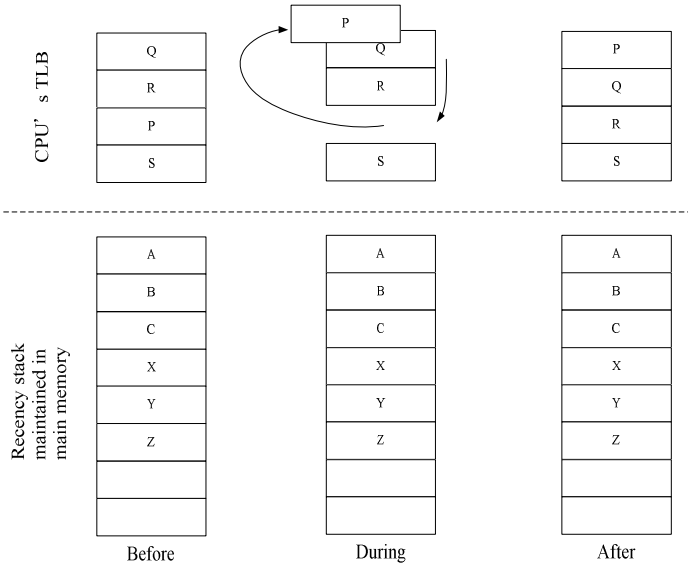
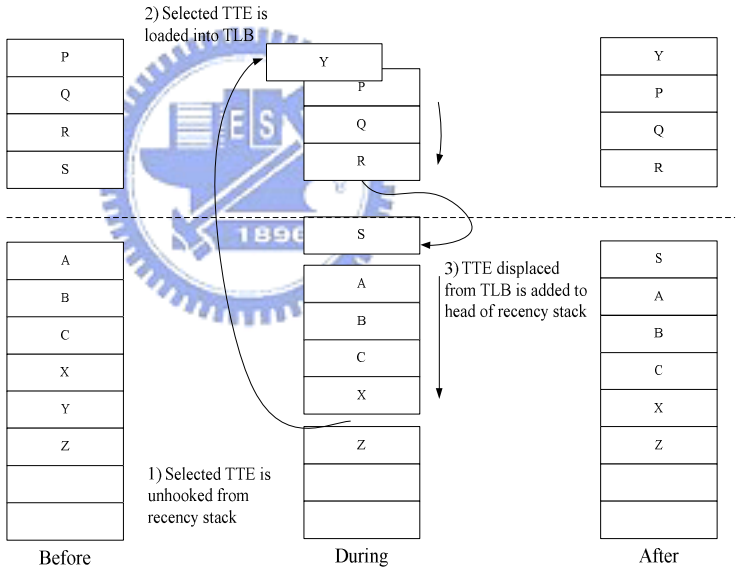


Figure 2-8: Share tag design of TLB and cache memory



(a)



(b)

Figure 2-9: Operations of "Recency Stack"

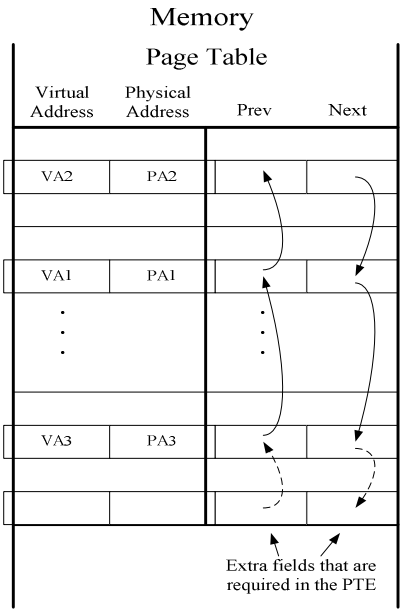


Figure 2-10: Memory translation table of TLB with "Recency Prefetching"

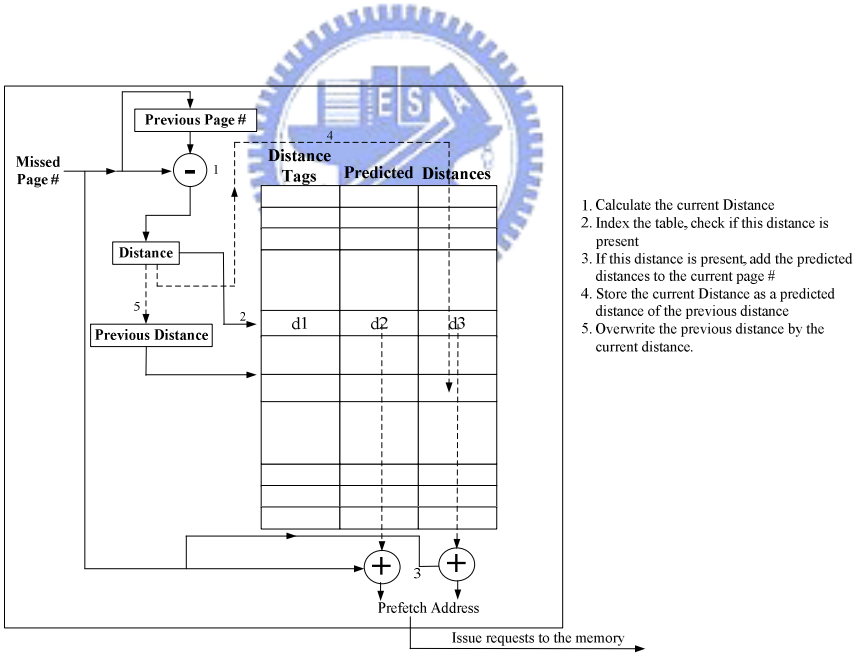


Figure 2-11: TLB with "Distance Prefetching"

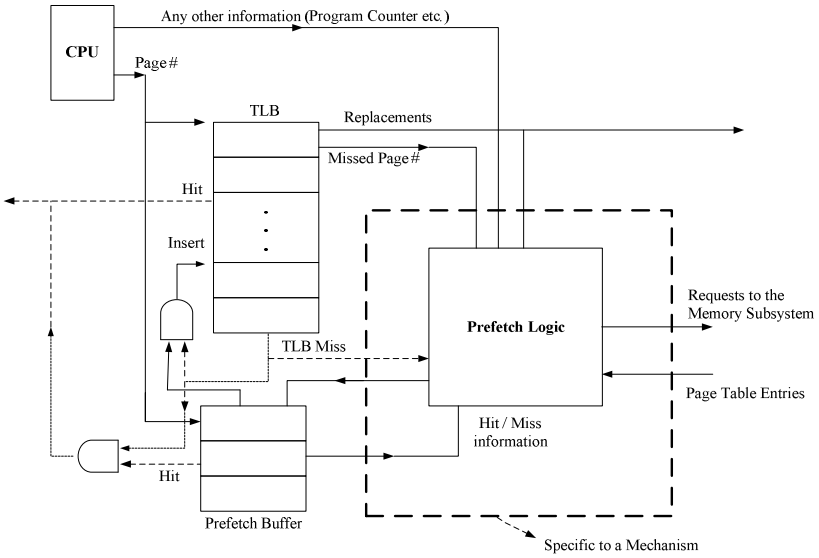


Figure 2-12: Schematic of generic TLB prefetching hardware

2-1-3 Reducing TLB Miss Rate in Context Switching

As mentioned in previous sections, the TLB miss handling requiring several main memory accesses and that impact the overall performance seriously. However, in traditional design, the simplest way to deal with context switching (address space switching) for TLB is to flush all the TLB entries. Thus, that's even worse if the miss caused by TLB flushing of context switching. After the flushing of the TLB, it needs lots of "learning time" to refill these entries. However, only a few studies focus on this topic. Liedtke try to reduce the possibility of TLB flushing of address-space switching via integrating the segmentation mechanism of *x86* [1]. Based on the L4, Wiggins and Heiser try to avoid reloading translation table base register by using a pointer register that points to a caching page directory (CPD) [2,3]. The basic idea of this implementation can be described as following sentences. The CPD contains entries from a number of different address spaces and each of it is defined by its own page table. Once the TLB miss occurs, the hardware only needs to reload the TLB via indexing to the CPD that contains pointers to LPT (Leaf Page Table, an array of 256 entries PTEs) of various address space. If it's a miss, the current thread PD (page directory) should be indexed by handler to find a valid entry. Then the entry should be copied into CPD. The handler restarts the thread. Finally, the hardware can reload TLB. Now, only a valid page table entry

should be found. Figure 2-13 depicted this basic idea. In fact, still lots of other research tries to reduce the possibilities by modifying the OS or page table structures. Besides these software solutions, the basic method supported by TLBs is to provide address space identifier (ASID) for each entry to identify each address space. Figure 2-14 shows the TLB with per-entry ASID tag.

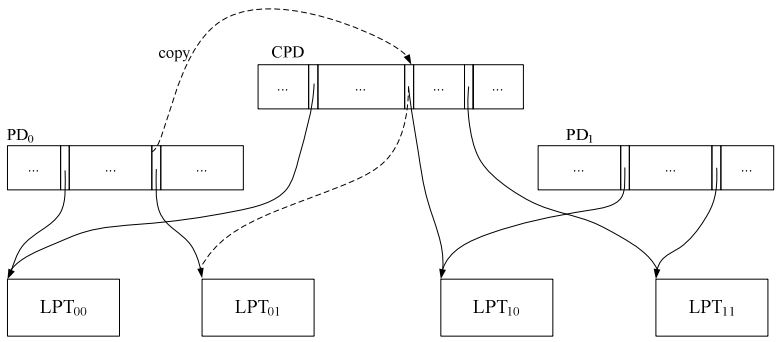


Figure 2-13: CPD and per-address page tables

ASID	VPN	PPN
A	0x8000	0x100
B	0x8000	0x400
B	0x8200	0x500
A	0x8020	0x120
	……	
A	0x8100	0x200

Figure 2-14: TLB with per-entry ASID tag

2-2 Circuit design with asynchronous circuits

The technological trend is inevitable:
In the coming decades, asynchronous design will become prevalent!

By Ivan E. Sutherland and Jo Ebergen
"Scientific American", August 2002 [4]

Asynchronous circuits have been studied since early 1950's; however, synchronous circuits have still dominated the mainstream of digital circuit design [4,6]. Recently, some academic and commercial research shows that it's worth to implement real-life systems with asynchronous circuits. But, without the global synchronization signal called "clock", it makes asynchronous circuit design very difficult. In order to replace the "clock signal", handshaking protocols between each part of asynchronous circuits are needed. It therefore makes the circuit costs of asynchronous circuits much higher than synchronous counterparts. In addition, because of lack of tools and standardization of implementation and design models, there is still not much research on it and that limits applications in commercial products. In fact, it's very hard to find commercial products that are implemented with asynchronous circuits. In this section, we'll discuss topics of asynchronous circuits from the classifications of asynchronous circuits, handshaking protocols, research of asynchronous circuits, and case study of implementation with asynchronous circuits.

2-2-1 Classifications of Asynchronous Circuits

We have discussed so many issues of asynchronous circuits, but you may ask what asynchronous circuits are. In fact, it's not very hard to answer this question. We can say that asynchronous circuits are circuits without any global synchronization signal called "clock." Based on this assumption, asynchronous circuits can be classified into four classes depending upon the delay model of gate and wire of the circuit. The four classes are *Delay-Insensitive* (DI) circuits, *Quasi-Delay-Insensitive* (QDI) circuits, Speed-Independent (SI) circuits, and *Self-Timed* (ST) Circuits [5,6].

Delay-Insensitive (DI) circuits are the most robust and reliable circuits of all. These classes or circuits permit arbitrary (unbounded but finite) delays on gates and wires. The basic concept of DI circuits derives from Clark's "*Macromodular computer systems*" proposed in 1967 [58]. However, because of its "arbitrary delays on gates and wires" nature, only a few circuits belong to this class. Martin already proved it in 1990 [59]. Thus, enormous limitations exist in designing DI circuits.

Because it's too hard to implement pure DI circuits, Quasi-Delay-Insensitive (QDI) circuits relieve a little in arbitrary delay on wires. QDI circuits are DI circuits with isochronous forks. It means that all branches of a forked wire have exactly the same wire delay [60]. Figure 2-15 shows the isochronous fork. In this example, the signal from A can propagate to both B and C with the same wire delay. With this assumption, it permits DI class circuits can be more practical. In fact, in order to meet DI and QDI constraints, the implementation costs of these circuits may be higher. In addition, they should be carefully implemented to avoid violations of the constraints. Thus, to implement such circuits are really very difficult. However, because no extra delay assumptions, DI and QDI circuits may be attractive for asynchronous VLSI circuit synthesis [60].

The concept of Speed-Independent (SI) circuits first appeared in 1959 proposed by David Muller [59,60]. The class of circuits allows arbitrary (unbounded but finite) delays on gates but assumes zero wire delays. The SI circuits can be modeled with Petri net [63].

Self-Timed (ST) circuits are popular in lots of asynchronous circuit implementations. It is introduced by Seitz in 1980 [64]. The ST circuit is composed of a group of ST elements and each of ST elements is inside of an "equipotential region." The wire delays of the region are negligible or well-bounded. The elements can be DI, QDI, SI, or circuits that can operate correctly with some local timing assumptions. There's no any timing assumption on communications between regions. That also means that the communication belongs to DI. For example, Chang et al. proposed a ST torus-network with 1-of-5 DI encoding in 2009 [65]. The implementation uses DI encoding communication between each parts of the whole design.

Figure 2-16 shows the relationship of these models of asynchronous circuits. If the design contains both DI components and ST components, it should be an ST circuit.

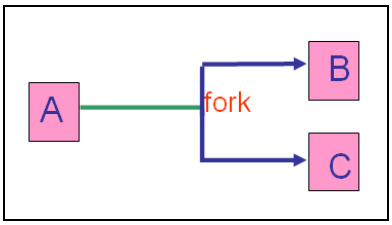


Figure 2-15: Isochronous fork

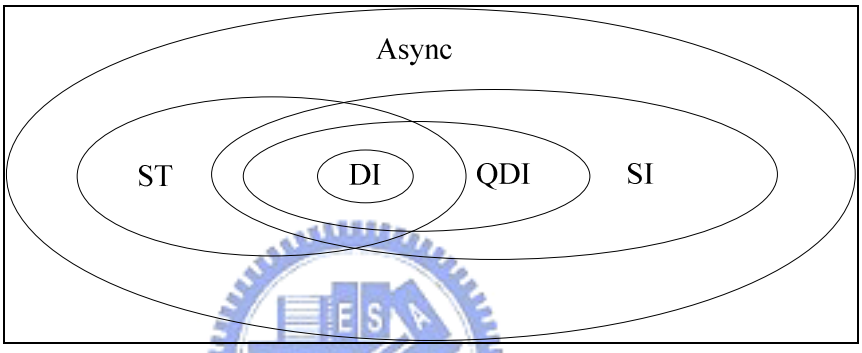


Figure 2-16: Classifications of asynchronous circuits

2-2-2 Handshaking Protocols

Without a clock to govern its actions,
an asynchronous system must rely on local coordination circuits instead!
By Ivan E. Sutherland and Jo Ebergen
"Scientific American", August 2002 [4]

Without clock signal, asynchronous circuits rely on handshaking protocols to make sure the correctness of the circuit operations [5,66,67]. The protocols can be divided into control signaling and data encoding. A complete handshaking protocol is a combination of the control signaling and data encoding. Figure 2-17 shows the 4-phase handshaking protocol. In this protocol, only the rising edge is the valid active transition; thus it's a level signaling or return-to-zero protocol. On the contrary, in the 2-phase handshaking protocol, the falling and

rising edge of request and acknowledge are active signals; thus it's a transition signaling or non-return-to-zero protocol. However, it makes the circuits, especially datapath circuits, very complex and hard to implement. Figure 2-18 shows the 2-phase handshaking protocol. In addition to control signaling, there are also choices for how to encode data (data signaling protocol). The Bundled Data or called Single Rail refers to separate request and acknowledge wires that bundles the data signals with them. Thus total n+2 wires are required to send n-bit data. Figure 2-19 shows the bundled-data model. Besides bundled-data model, there are data encoding methods for DI circuits. However, because of implementation issue, dual-rail encoding is the most popular used DI data encoding scheme. To represent 1-bit data in dual-rail encoding method, two physical wires are used. For example, a valid data, D is represented by two physical data wires, d.0 and d.1. The following equation shows this encoding scheme. (1) D = 0 ; (d.0,d.1) = (0,1) (2) D = 1 ; (d.0,d.1) = (1,0). In particular, (0,0) represents a space which allows us to identify consecutive 0's or 1's. (1,1) state is not used. Data transferring starts from the (0,0) state (called "null" or "empty" data). If a state is changed from (d.0,d.1) = (0,0) to (0,l)/(1,0), which notices the arrival of valid data '0/l'. Thus total 2*n wires are needed to transfer n-bit data. Figure 2-20 shows the dual-rail model.

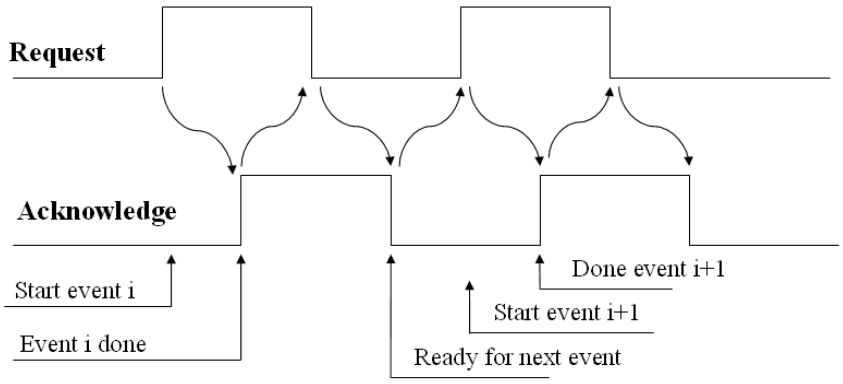


Figure 2-17: The 4-phase protocol

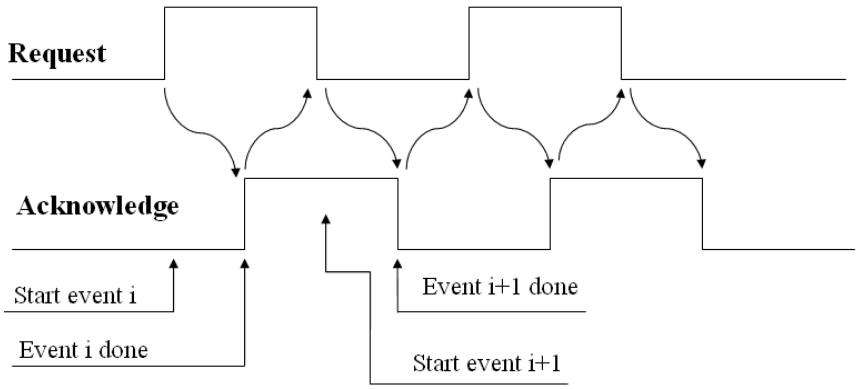


Figure 2-18: The 2-phase protocol

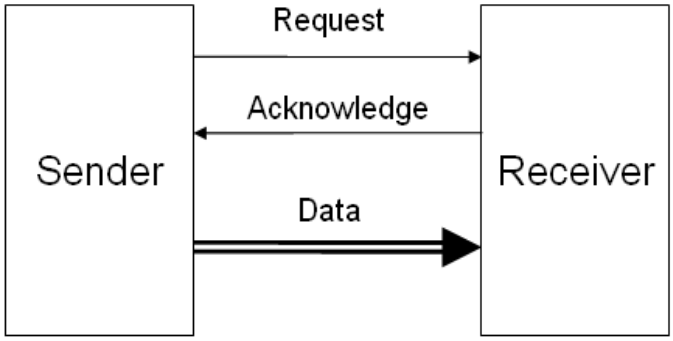


Figure 2-19: Bundled-data signaling model

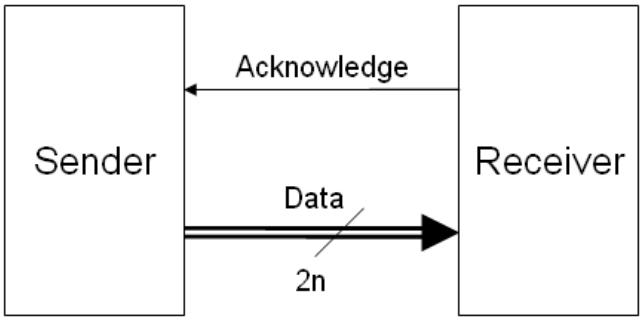


Figure 2-20: Dual-rail data signaling model

2-2-3 Research of Asynchronous Circuits

Though it's not very easy to conclude all studies of asynchronous circuits, we'll discuss

the asynchronous pipeline models first. That's because most asynchronous systems are designed or implemented based on these asynchronous pipeline models. David Muller proposed his famous Muller C-element and Muller pipeline (aka Muller distributor) in 1959 [68,69]. A Muller pipeline is a naturally simple and elegant handshaking control model. The simplest form of Muller pipeline mainly consists of C-elements and inverters. Figure 2-21 shows the schematic symbol and truth table of a two-input C-element. If both inputs are high or low, the output will be high or low; otherwise, the previous value is kept. Figure 2-22 shows the original Muller pipeline model. To understand its behavior, let's consider the *i*th C-element C_i. In the initial state, all C-elements are initialized to 0. The handshaking may be initialized. The *i*th C-element C_i can propagate a 1 from its previous stage the (*i-1*)th C-element only if the next stage C-element (C_{i+1}) is 0. Thus, the signal can be propagated one stage to one stage. It should be notice that the original single-rail model is based on bundled-data model; thus the request signal must be propagated via a matching delay as shown is Figure 6. In fact, the matching issue should be carefully handled on all bundled-data model. The pipeline model can also be constructed as 4-phase dual-rail model as shown in Figure 2-23 [66]. The model can be considered as two Muller pipelines connected in parallel with a common acknowledge signal in per stage. We implemented a 4-phase dual-rail pipeline based QDI 8-bit NCTUAC18 microcontroller core in 2009 [70].

Besides the Muller pipeline, there are also several models were proposed. The most important of all is the micropipeline which was described by Ivan E. Sutherland in his famous Turing Award "Micropipelines" lecture in 1989 [71]. The approach is based on a two-phase bundled-data model with micropipeline as backbone control circuit. Figure 2-24 shows the control circuit of a 4-stage micropipeline model. Without datapath, the micropipeline is a string of Muller-C elements. At each stage, there are one request input signal, R(*n*), and one output acknowledge signal, A(*n*). The request signal can propagate from left-most side, R(*in*) to the right-most side, R(*out*). It's the same as the direction of data flow. The data therefore can flow from the left-most side to the right-most side stage by stage. After the data can be received by the right-most side, the acknowledge signal should be returned from the right-most side, A(*out*). The acknowledge signal, A(*n*), therefore can propagate back to the left-most side, A(*in*), and clear the whole pipeline. Thus the pipeline can keep on operation. Figure 2-25 depicts how to combine the control circuit of micropipeline with datapath. As the most well-known asynchronous circuit design model, lots of different asynchronous systems

have been implemented based on it. It can be used to implement many kinds of different pipelined systems, even processors. For example, the NSR processor is a very simple 16-bit micropipeline based microprocessor with very simple RISC instructions (less than 20 instructions) [72]. The Amulet1 is known as the first ARM compatible processor implemented with asynchronous circuit [15,73]. It was implemented with 2-phase micropipeline architecture.

There are also some different models proposed for asynchronous circuits design. Some try to modify the original "micropipeline" architecture. For example, a new control circuit for micropipeline was proposed by Choy et al. [74] and "Micronets" architecture tries to decentralize the control to the functional units [75]. Furthermore, there have been still several famous asynchronous processor implementation models proposed. Takashi Nanya et al. showed their QDI 8-bit microprocessor model called "TITAC" which uses Martin's Q-element [60] as control circuitry [76]. Figure 2-26 shows the Martin's Q element. With Q element, the control path can be easily built. In addition, they proposed Autosweeping Module (ASM) which is modified from Q element to replace Q element to gain better performance. TITAC2 was proposed to show a new delay model called scalable-delay-insensitive (SDI) [77]. The delay model modified original DI or QDI unbounded gate and wire delay to bounded relative delay ratio between any two components. There are also some works that try to model processor with asynchronous circuits. Martin et al. at Caltech have already shown three generations of different asynchronous processor model [78]. Chen et al. showed an asynchronous RISC processor model in 2002 [79]. In addition, there are also several asynchronous superscalar processor models proposed, for example the Kin architecture [80], Hades project [81], and the most famous of all the counter flow pipeline (CFPP) [82]. The design of CFPP is quite different from traditional design concept. Figure 2-27(a) shows the architecture of a 5-stage CFPP. The design separates the instruction flow and result flow in a counter flow. In this Figure, the instruction is fetched, decoded, and inserted into the instruction pipeline in stage F. At the same time, the source operands needed for this instruction is also inserted into the result pipeline in stage R. Figure 2-27(b) describes the instruction and result bindings. Each binding is composed of register name, valid bit, and data value. Because the instruction flow and data flow walk in counter flow, the instruction can meet needed data in one of the stages. Once the needed operands can be fetched, the instruction can be executed correctly. In addition, if the binding destination of

the instruction matches one of the binding results, the binding result will be updated. Thus the following instructions in the pipeline can obtain correct result value. It may be regarded as special designed data forwarding. However, all these superscalar models are not very easy to implement or just ideas that cannot be realized and certainly not very suitable to be implemented for cores of embedded processors. In fact, because it's very hard to guarantee the instruction execution order in asynchronous design, only some research of asynchronous superscalar processor are really in progress.

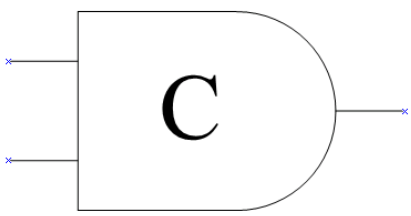
Another issue should be pointed out here. As mentioned before, Chen et al. implemented a 4-phase dual-rail pipelined QDI processor [70]. However, in order to implement the QDI processor, all dual-rail components should be constructed first. These components even include all basic logic components. That's lots of extra efforts for designing a processor. Considering the synchronous circuit design, they can be easily implemented with lots of pre-designed cells, components, or even large modules. In fact, it's also a key to success. Some researchers have been already trying to offer solutions for asynchronous circuit design. Some try to provide basic building element. For example, Smith et al. proposed a new DI digital system called NULL Convention Logic (NCL) [83]. With NCL, DI system can be built easier. Some try to offer new pipeline/FIFO control. For example, a basic control circuits for an asynchronous pipeline called Asynchronous Symmetric Persistent Pulse Protocol, "asP*" was introduced by Molnar [84]. Sutherland and Fairbanks described GasP in 2001 [85]. There is still much different research involving new control circuits or offering new asynchronous elements.

Besides the "pure" asynchronous implementation research, some research topics focus on trying to find applications in other directions. Imaging on a large SoC, each components or IPs may be designed by different teams or even different companies. Integrating them on a single die may be a very difficult job. The most important reason is that these different designs may be operate correctly in different clock frequency. Some research tries to wrap the synchronous circuit with asynchronous wrapper. Thus, the whole system can communicate with asynchronous channels, while each local circuit can operate in their local clock. Thus, some Globally-Asynchronous Locally-Synchronous (GALS) methodologies are proposed. The concept of GALS was proposed by Capiro in his PhD thesis in 1984 [86]. Figure 2-28 depicts this idea. In addition, some research focus on the interconnection networks with

asynchronous circuits. In fact, MPSoCs or multicore processors have been becoming the major trend of system or processor designs nowadays. Thus the design of interconnection networks becomes the most important issue of all. However, lots of different problems may arise in the network design and they should be carefully handled. It is widely known that most of these problems can be resolved easily by asynchronous circuits. Hence, it's really attractive to replace these networks with asynchronous implementations. For example, Dally and Seitz implemented the first torus topology based interconnection networked multiprocessors in 1986 [87]. They implemented the self-timed torus routing chip (TRC) which uses the bundled-data encoding to perform cut-through routing in k-ary n-cube multiprocessor interconnection networks. In 1997, Natvig presented a high-level simulation model of TRC written in Verilog [88]. Chen et al. implemented self-Timed torus interconnect with 1-of-5 encoding in 2009 [65]. In fact, because of asynchronous nature, the routing paths with different distances can operate in different speeds.

In addition, we have already pointed out that almost all commercial digital systems are implemented with synchronous circuits. One very important reason is lack of suitable EDA tools that can be used to implement asynchronous circuits directly. In fact, it's also hard to directly model your design in behavior or RTL model with traditional HDL directly. Thus, most designs should be implemented in gate-level. In order to reduce the efforts in designing asynchronous systems and circuits, specific HDLs for designing asynchronous systems and circuits are needed. Tangram and Balsa HDLs are the most famous two of all related frameworks. The Philips Research Laboratories started to develop the Tangram tool over 20 years ago [89]. Now the tool is offered by Handshake Solutions. In fact, the ARM 996HS was also developed via it [7,20]. Handshake Solutions now provides Haste Design Language for describing the behavior of asynchronous circuits. In addition, an integrated easy-to-use tool suite called TiDE^{TM} (Timeless Design Environment) is also offered [90]. In fact, it's the most successful commercial EDA tools for asynchronous circuit design. However, Balsa is a framework for providing an asynchronous HDL and synthesizing of asynchronous circuits and systems. It's an open source and free solution developed and offered by the University of Manchester [91,92,93]. In fact, part of Amulet 3 was designed with Balsa [18]. In addition, Chen et al. also proposed an asynchronous pipelined 8051 soft-core with Balsa [94,95]. Zhang and Theodoropoulos modeled an asynchronous MIPS core with Balsa called SAMIPS [96]. An asynchronous MP3 decoder was also modeled by us with Balsa [97]. In addition, there are

still several works in developing EDA tools for asynchronous circuits. In addition, asynchronous research group in Caltech provides Communicating Hardware Processes (CHP) and its synthesis tool as asynchronous circuit design tool [98]. The notation of CHP is inspired by CSP. In fact, three generations of their asynchronous processors were designed with CHP [76], including a very large MiniMIPS processor [99]. The most interesting of all is SoCAD developed at Tatung University, Taipei, Taiwan [100]. They don't develop any special HDL for asynchronous circuit design. Instead of specific HDL, Data dependency graphs (DDG) or Java language can be used to model the behavior of the design. Via several translation processes proposed by Cheng, the DDG or Java models will be translated into VHDL and mapped to lots of pre-defined cell-based designed asynchronous components. With SoCAD, the goal of hardware/software codesign can easily be achieved. They also modeled a very robust asynchronous Java Chip with it [101]. Unfortunately, though several EDA tools for asynchronous circuit design can be found, it still has a very long way to go for these tools.



a	b	y
0	0	0
0	1	No change
1	0	No change
1	1	1

Figure 2-21: The Muller C-element: symbol & truth table

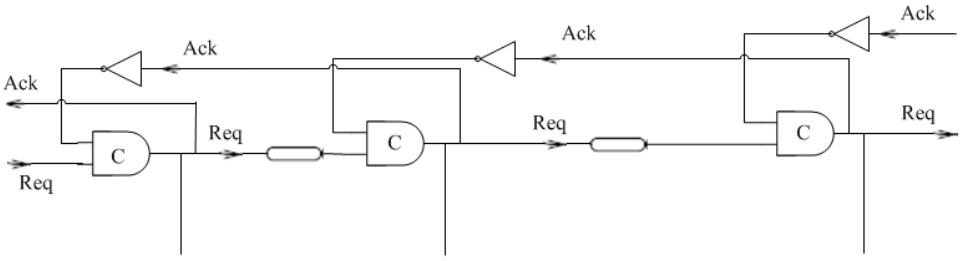


Figure 2-22: The Muller pipeline

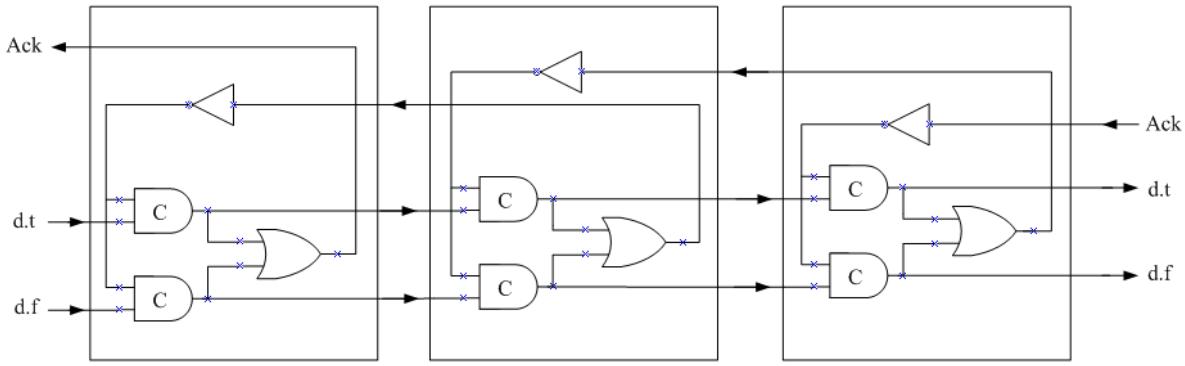


Figure 2-23: A three-stage 1-bit wide 4-phase dual-rail pipeline

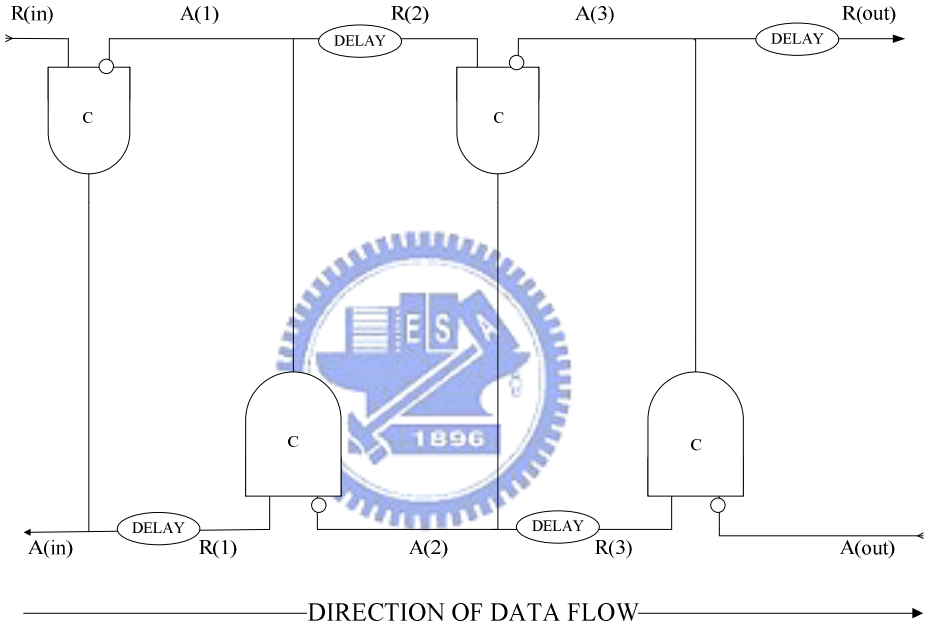


Figure 2-24: Control circuit of micropipeline

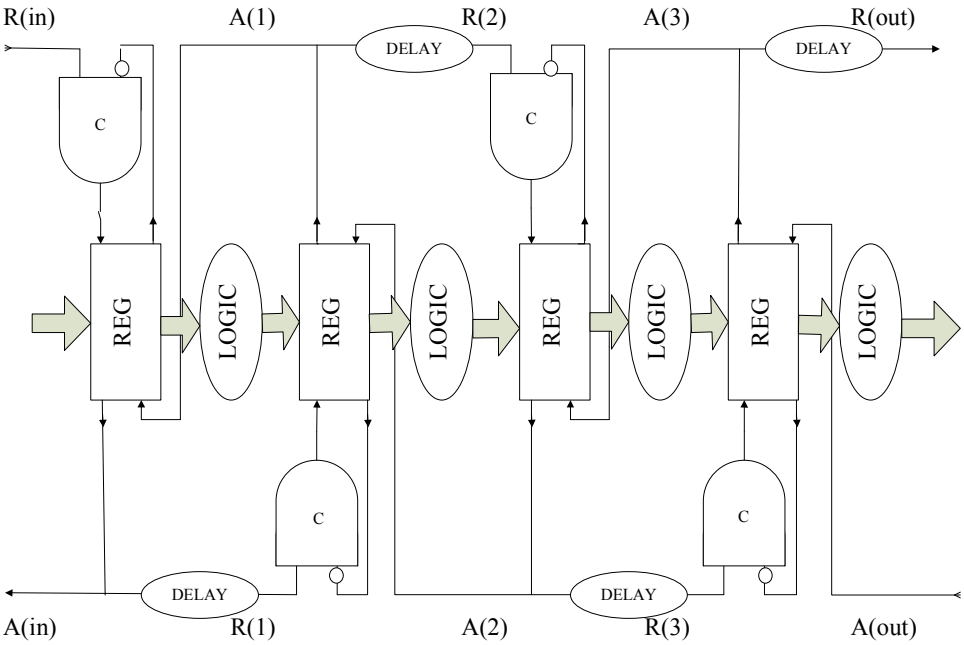


Figure 2-25: Micropipeline architecture

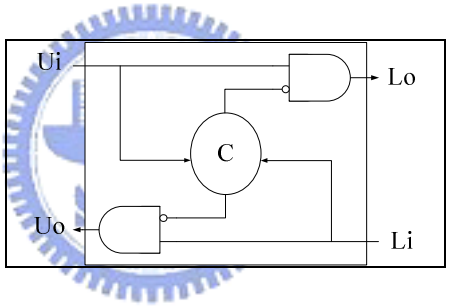


Figure 2-26: Q element

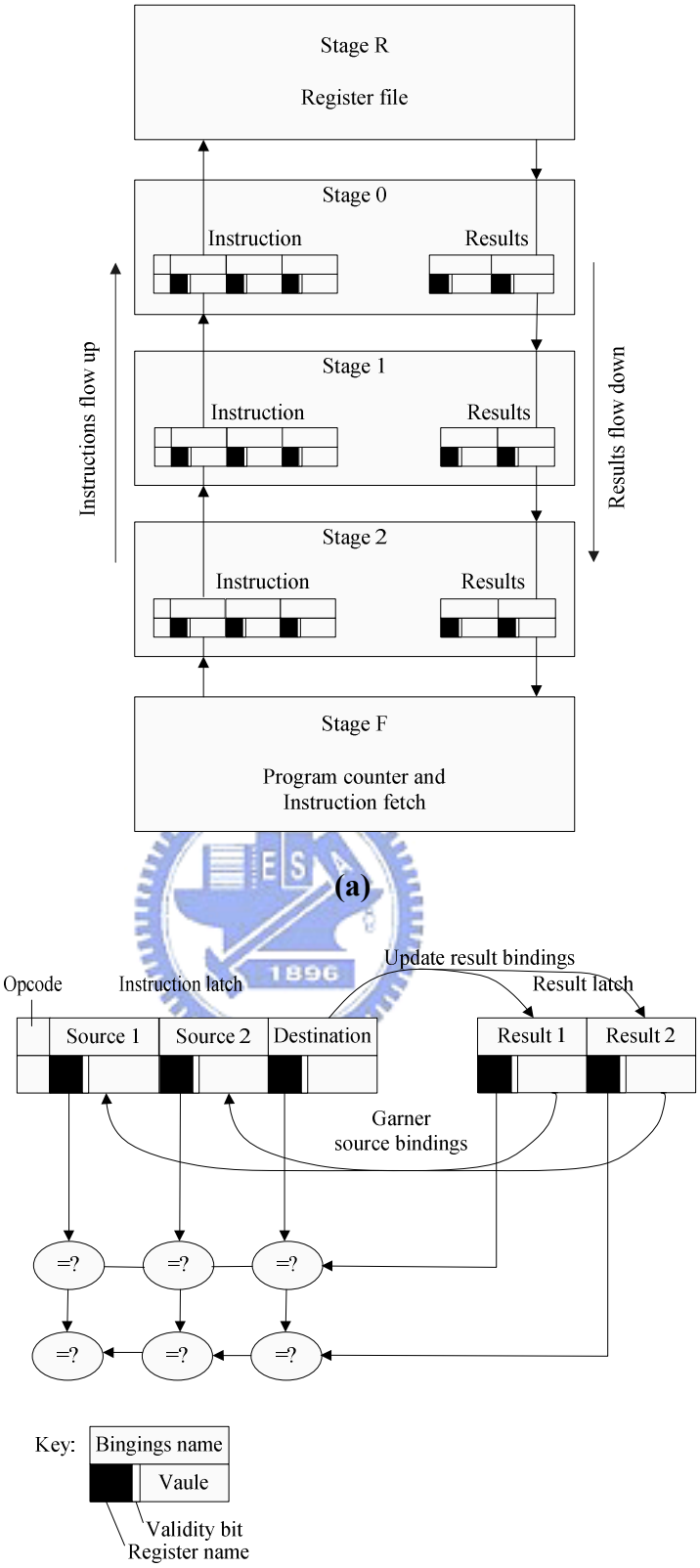


Figure 2-27: The architecture of CFPP

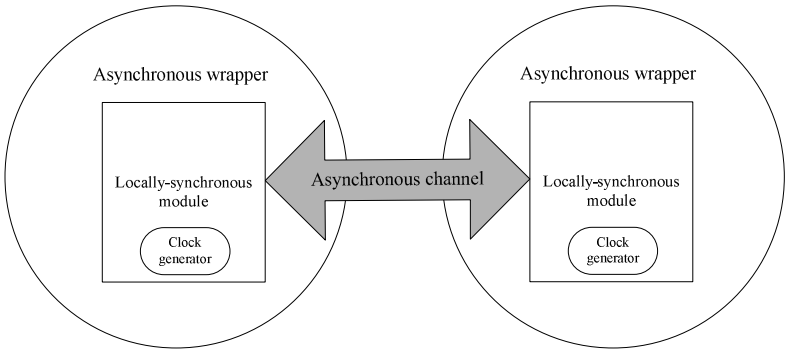


Figure 2-28: Concept of GALS

2-2-4 Case Study of Asynchronous Circuit Design

As mentioned in previous sections, it's difficult to design and implement asynchronous circuits directly. Most designs cannot be implemented via writing RTL of Verilog or VHDL. In our group, some circuits are implemented with Balsa HDL, and some are implemented via writing gate-level descriptions of Verilog HDL. In this section, the two methods will be discussed.

It is widely known that it's hard work to implement all designs with gate-level descriptions. It's not worth to implement all circuits with gate-level descriptions. With higher level modeling, we can pay more attention on design itself. That's the same for asynchronous circuit design. Thus, we select Balsa framework as our tool. Because the details of Balsa HDL and framework will be described in section 4.1, we'll describe how to model a design with Balsa. We'll describe how to model a pipelined asynchronous 8051 core here [94,95]. The first step, you must define your design model and the asynchronous communication channels between each part of your design. Figure 2-29 shows the AsyncPA8051 model and its interfaces and channels between each part of the model. Then, each part of the design can be described with high-level Balsa descriptions. Following segment shows the top module of the AsyncPA8051. It should be noted that components are connected with communication channels.

```
procedure PA8051_IFIDOF(output p0,p1,p2,p3 : byte) is

channel IF_2_ID_data : byte
channel IF_2_mem_addr : Address
channel ID_2_IF_addr : Address
…….

BalsaMemory_interface(IF_2_mem_addr,read,...)||
IF(mem_2_IF_data,ID_2_IF_addr,...) ||
ID_top(IF_2_ID_data,jmp,ID_data,...)||
PA8051_OF(ReadS, WriteSin, MEM_OF,...)||
RAM(maddr,wr,mem_in,mem_out,p0,p1,p2,p3)||
MEM_INTERFACE(MemIn,....)||
PA8051_EXE(src1, src2, src3,...)||
PA8051_WB(EXE_WB, ...)||
Ram_Read_Arbitor(MEM_data,valid_face_2_arbitor,...)||
  end
end
```

However, the costs of asynchronous circuits generated by Balsa HDL are sometimes not very cheap. In addition, the gate-level descriptions of bundled-data circuits generated by Balsa cannot be optimized by your target EDA tools in order to keep the delay elements inserted by Balsa. Furthermore, you should still pay attention on these matching delays. In fact, that's the most important issue in implementation of bundled-data circuits. Therefore, we also try to implement some of our designs with Verilog gate-level descriptions. Following example is used to describe how we implemented our design in Verilog gate-level descriptions. The example is a 4-phase dual-rail pipelined based 8-bit QDI microcontroller core called NCTUAC18 [70]. In order to implement circuits with dual-rail QDI model, all basic dual-rail DI/QDI building cells and components should be constructed first. We implemented all needed basic QDI dual-rail gates and constructed all building blocks with these QDI dual-rail gates. The most important of all is the C-element. The generalized transistor-level C-element implementation is shown in Figure 2-30 (a); however, to provide synthesizable model for FPGA we also modeled it with gate-level design as shown in Figure 2-30 (b). In addition, we also implemented C-element with reset for pipeline latch. With C-element, other basic dual-rail components can also be constructed easily, for example dual-rail OR gate as shown in Figure 2-31.

In addition, we developed our own QDI register set. Figure 2-32 shows a 1-bit dual-rail register. When a valid codeword is sent to *din.t* and *din.f*, the two NOR gates can correctly hold it. If the data item is written into the register, it issues acknowledgement to its previous stage to inform the written operation done. Because of the dual-rail nature, we designed the acknowledge signal simply through ORing two in/out signal of the two NOR gates. To read data from the register, just send read request signal to it and thus the dual-rail data can be correctly read out via *dout.t* and *dout.f*. Our register design does not deliver much higher cost than traditional register for synchronous systems.

Because lack of synthesis tool the design cannot be written in RTL model. Thus, the whole circuit should be carefully written in gate-level design. If the asynchronous design should be implemented with CMOS VLSI, some components had better to be created with full-custom design. For example, to implement efficient CMOS C-element, manually designed C-element cell is needed. Besides modeling it with transistor-level as shown in Figure 2-30(a), we also modeled it with gate-level as shown in Figure 2-30(b). Thus, it can be synthesized with CAD tool. In addition, because of DI nature, all components should be constructed carefully with DI model. Thus, the implementation cost is very high. The circuit should be optimized manually. Then, the design can be implemented with the pre-constructed components.

After all building blocks were constructed, the circuit control model should be decided. In this example, the 4-phase dual-rail pipeline model as shown in Figure 2-23 was selected. Finally, each execution stage should be designed and put into the pipeline. Figure 2-33 shows the system block diagram of NCTUAC18 microcontroller core.

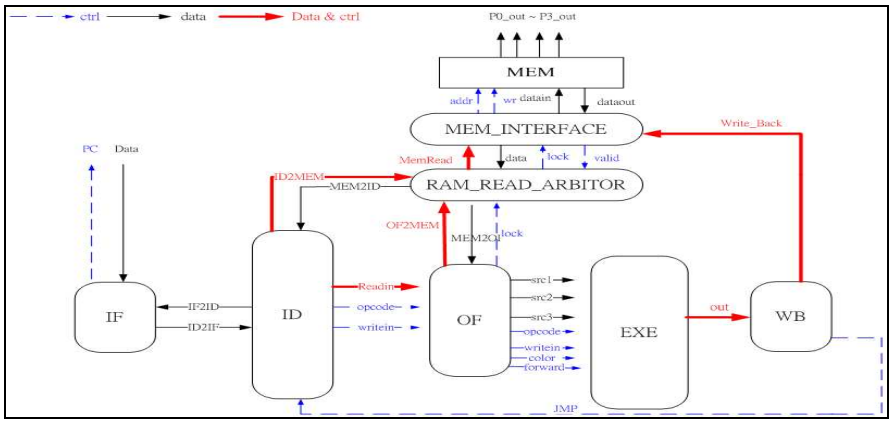
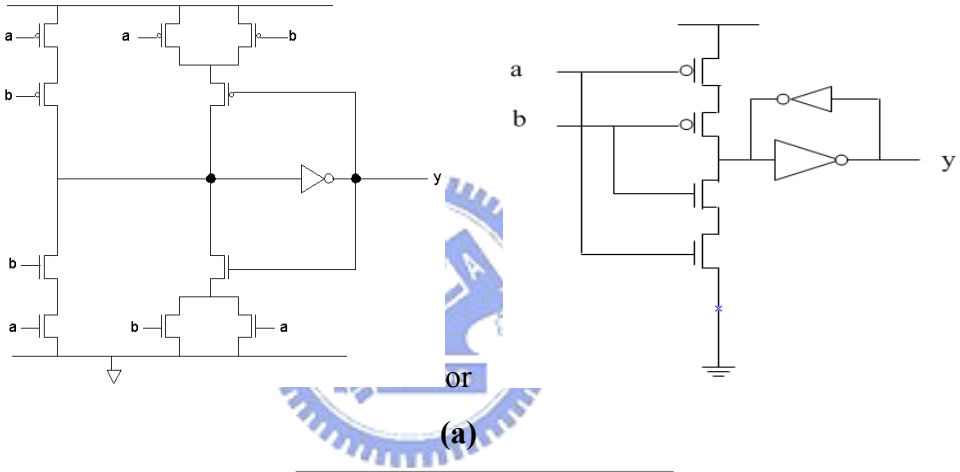
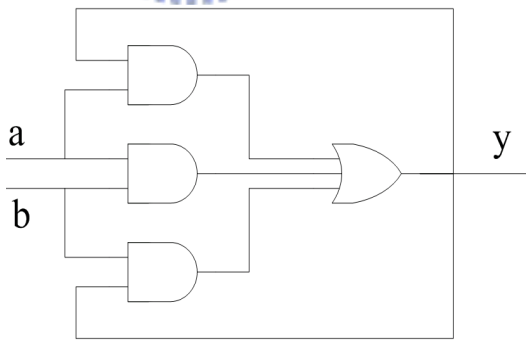


Figure 2-29: Asynchronous pipelined 8051 architecture



(a)



(b)

Figure 2-30: (a) Generalized transistor-level C-element implementation
(b) Gate-level C-element implementation

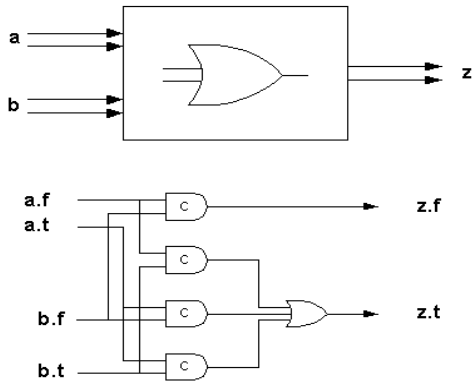


Figure 2-31: Dual-rail OR gate symbol and gate-level implementation

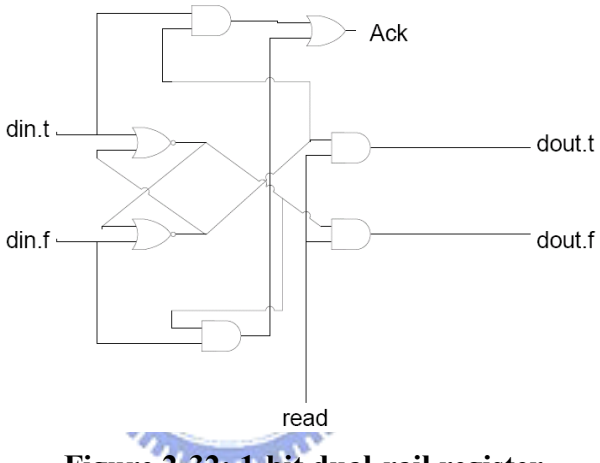


Figure 2-32: 1-bit dual-rail register

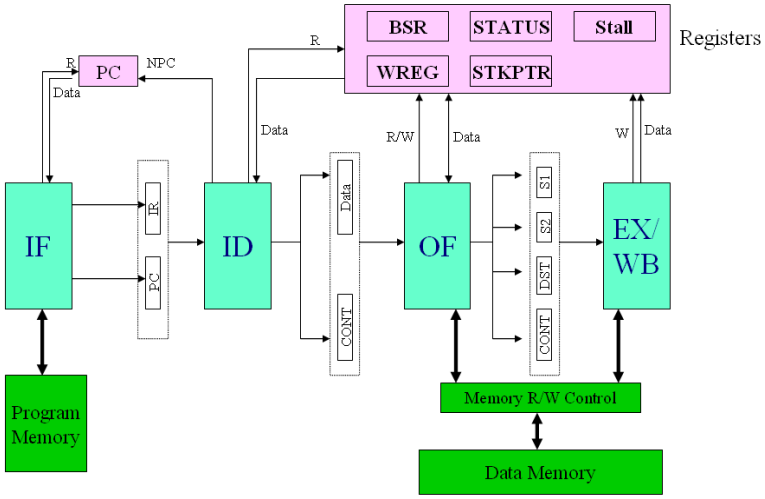
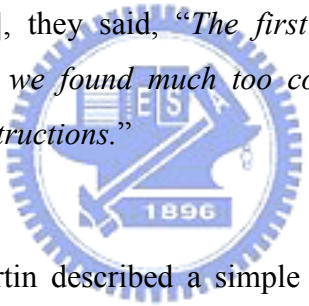


Figure 2-33: Architecture of NCTUAC18 microcontroller core

2-3 Previous Asynchronous TLB or MMU Design

Though some asynchronous processors are proposed for the past years, most of these processors are very simple. Thus, no very complex memory address translation mechanisms were implemented inside these processors. Furthermore because of the complication, it's really very difficult for us to find TLB design inside commercial asynchronous processors. In fact, even no what is called memory management unit by ARM is implemented inside the first commercial licensable 32-bit asynchronous core, the ARM996HS [7,20]. Only what is called enhanced memory protection unit (MPU) by ARM is implemented. That means that it only supports hardware memory protection over software-designated regions. The processor has no virtual memory supporting hardware. The research group of Caltech delivers three generations of asynchronous processors [78]. The biggest one is the MiniMIPS that is an MIPS R3000 compatible asynchronous processor [99]. We still cannot find TLB design inside this processor [102]. In [102], they said, "*The first prototype misses the TLB (address translation mechanism) which we found much too complicated, the partial-word memory operations, and some cache instructions.*"



However, Myers and Martin described a simple memory management unit with CSP specification [103] for an asynchronous processor [104]. Figure 2-34 shows the MMU that they described. This MMU can generate 24-bit real address via concatenating 16-bit memory address from the memory address (*ma*) bus and 8-bit address from one of the two segment register, *sr* (Segment Read register) and *sw* (Segment Write register). Once the real address can be generated, it will be placed on the real address (*ra*) bus. Thus the data can be transferred from memory interface and microprocessor via 16-bit data bus. However, it should be noted that the *sr* and *sw* can only be accessed via memory read/write to address 0xFFFF and 0xFFFE respectively. The contents of segment register are transferred via low 8-bit of data bus. The data bus therefore can transfer data between microprocessor and memory or the two segment registers. In addition, the microprocessor initiates memory or the two-segment-register read/write communication to MMU via *MDl* (memory data load) and *MDs* (memory data store) control signals. Through comparisons of memory address, MMU decide if it's a memory or the two-segment-register read/write communication. If it's a memory read/write communication, the MMU can initiates *MSl* (memory storage load), or

MSs (memory storage store) control signals to memory interface. In addition, the real address will be placed on the *ra* bus. Thus, the load and store operations are roughly described in the following descriptions.

$$
\begin{aligned}
&* [[\, \overline{MDl} \rightarrow (b1,b2,b3 := \\
&\quad (ma = FFFF),(ma = FFFE),\neg(ma = FFFF) \wedge \neg(ma = FFFE)); \\
&\quad [\ \ b1 \rightarrow data := sr; MDl \\
&\quad [] \ b2 \rightarrow data := sw; MDl \\
&\quad [] \ b3 \rightarrow ra := sr; MSl; MDl] \\
&\ [] \ [\, \overline{MDs} \rightarrow (b1,b2,b3 := \\
&\quad (ma = FFFF),(ma = FFFE),\neg(ma = FFFF) \wedge \neg(ma = FFFE)); \\
&\quad [\ \ b1 \rightarrow sr := data; MDs \\
&\quad [] \ b2 \rightarrow sw := data; MDs \\
&\quad [] \ b3 \rightarrow ra := sw; MSs; MDs]]
\end{aligned}
$$

Though Myers and Martin modeled an asynchronous MMU, it's really just a very rudimentary design. The functionality of this design may be not suitable for most applications. However, they still clearly demonstrated how to design asynchronous MMU with high-level descriptions. In addition, they also demonstrated how to optimize the design and implemented with the asynchronous circuits.

Weigel proposed two much more practical architectures of asynchronous MMU and TLB [105]. With aid of original author of Balsa, he modeled the two architectures with Balsa HDL carefully. The two architectures were designed to connect to a modified ARM coprocessor interface. Figure 2-35 shows the baseline architecture of the MMU. In this architecture, all components are activated in sequence. Because all operations are performed sequentially in Baseline Architecture, he also proposed architecture called Performance Architecture in order to improve the performance through speculative performing operations in parallel and pipelining. Figure 2-36 shows the architecture of performance architecture. However, in order to improve the translation performance, a TLB model was described in his design. He described the TLB in three aspects. The behavior of entry organisation, the entry lookup, and entry invalidation were all introduced in Balsa descriptions. The Balsa descriptions of three aspects were all detailed listed. In fact, the TLB described only a very basic design here. In addition, because of limitations of Balsa tool, Weigel suggested the real

implementation of TLB may try to reference other asynchronous cache design.

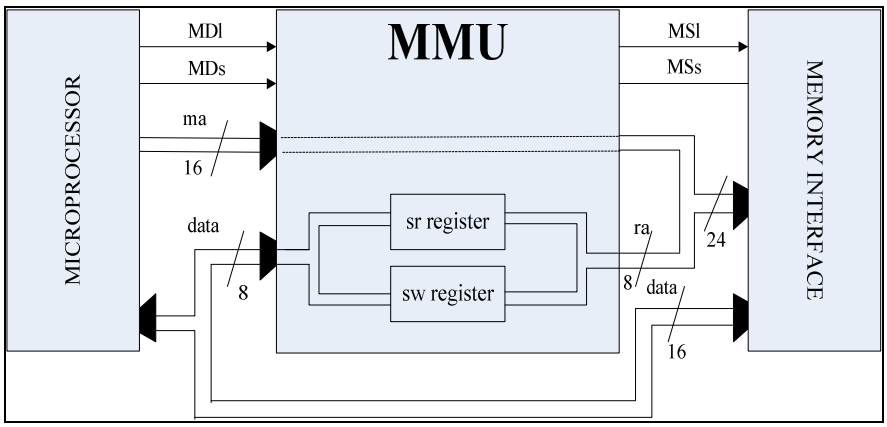


Figure 2-34: Overview of Myers and Martin's asynchronous MMU

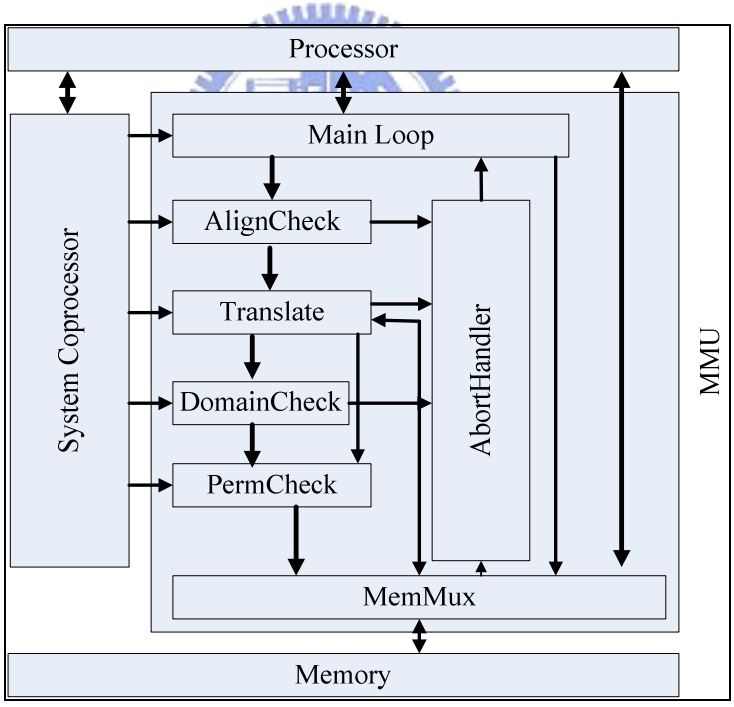


Figure 2-35: Architecture of baseline asynchronous MMU

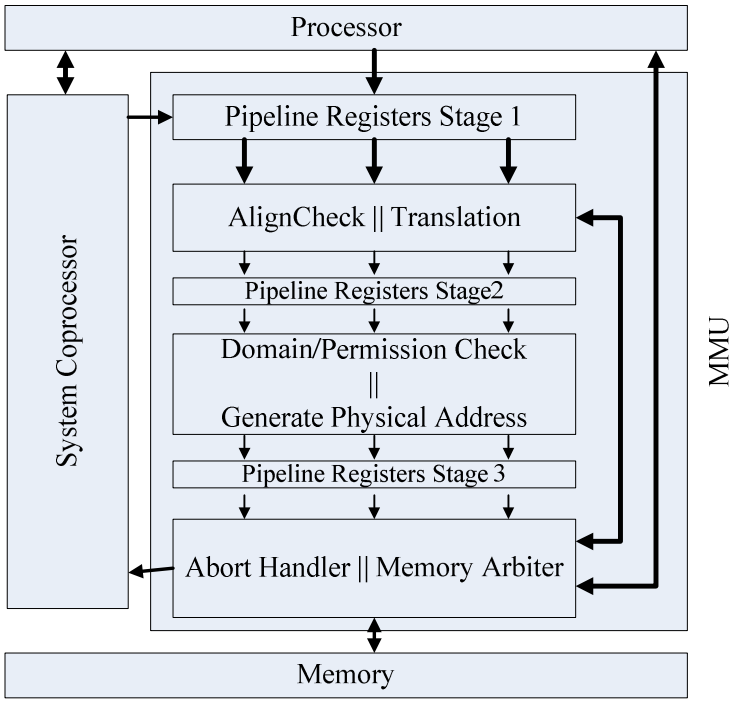


Figure 2-36: Architecture of asynchronous MMU with performance architecture

Chapter 3: Proposed TLB architecture for asynchronous embedded processor

As mentioned in Section 1-3, OS support is an important new issue for designing embedded systems or handheld devices. In order to support these modern embedded OSs, providing virtual memory is becoming more and more important. A well-designed TLB will become one of the critical issue in these embedded processor performance. It should be noted that page table traversal is much more expensive in embedded system than that in desktop system. However, in order to reduce the implementation costs, some designs including the most popular general purpose IA32 family processors simply flush the TLB entries in context switching (address space switching). It is widely known that per-entry ASID tag can reduce such misses. But it may increase the overall costs in tag bits. That may be a bad idea for asynchronous processor. In our work, we try to provide an alternative architecture via the concept of banking TLB. This architecture therefore can be implemented in our future asynchronous embedded processor core. In addition, we also hope that this architecture can also be implied for IA32 processors. We'll discuss this architecture in this chapter.

3-1 Relationship between the TLB miss rate and sizes

It is widely known that the two most important issues for cache system performance are lower miss rate and the miss penalty. It's almost the same for the TLB performance. In fact, because the miss rate has the greatest impact on TLB performance, most studies focus on it. In this section, we consider the relationships among miss rates, page sizes and TLB sizes. In order to study this topic, we have done simulations with different TLB configurations..

Let's consider the relationship between the miss rates and TLB sizes with 4KB page size. Figure 3-1 shows the relationship between TLB sizes and miss rates of running *gcc*. The two results show that the miss rates would be lower if the TLB sizes can be increased. We can also find that in order to obtain better performance for 4KB page the size should be at least 64 entries. However, that's not always true for all applications. Let's observe the result of *ijpeg*

showing in Figure 3-2. It's very clear that a 16-entry TLB is enough. It's useless to increase the number of TLB entries. In fact, it's almost the same for some other benchmark programs, such as *vortex* and *li*. However, the results vary from application to application.

Another solution to improve the performance of TLB is to extend the page size into larger one. In fact, most modern processors provide multiple page sizes, such as 4KB, 2MB, and 4MB on all new Intel® IA32 series processors [31]. The advantages of larger page size are not only obtaining better performance but saving the implementation cost with shorter tags of virtual page number (VPN) and translations (physical page number, PPN) needed to be stored. It is also a good method to reduce the cost on TLB implementation of processors with larger addressing space, such as processors with 64-bit addressing capability. Certainly, larger page size is suitable to be implemented for processor core of SoC or embedded systems. Figure 3-3 shows the miss rate of *compress* for 4KB, 16KB, 32KB, 64KB, and 1MB page sizes with different TLB sizes. Observing the results, we can easily find that the performance of 1MB page size of TLB with only 8 entries can even outperform 4KB page size of TLB with 256 entries. In fact, with the larger page size the larger working set can be covered. In addition, we can also find that the performance of 32KB page size TLB with 32 entries is good enough for *compress*. With prefetching mechanism, the performance would be even better. However, according to the previous discussion, even with 4KB page size, the total TLB entries needed may still vary from application to application. Sometimes, even 16-entry TLB is good enough for 4KB page. In fact, the new proposed architecture can be implemented to support different page size. Furthermore, the TLB size of each bank is also configurable depending upon the system needs. It's an implementation tradeoff!

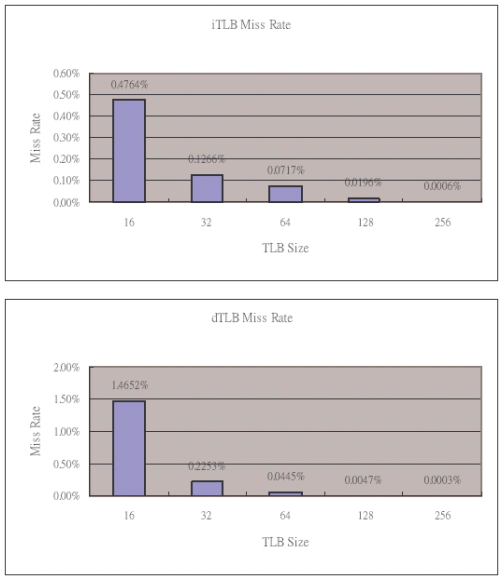


Figure 3-1: ITLB/DTLB miss rate for *gcc* with 4KB page

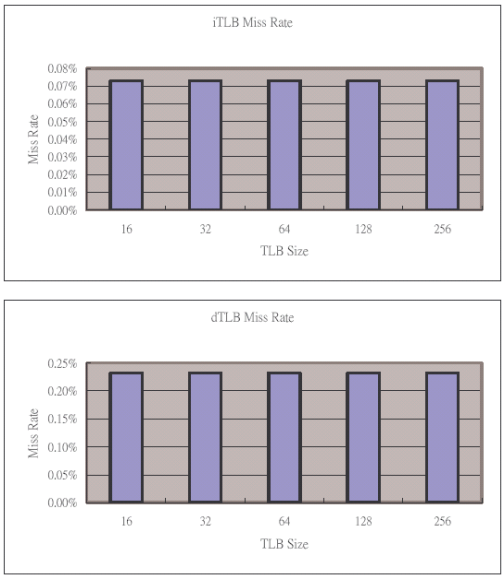


Figure 3-2: ITLB/DTLB miss rate for *ijpeg* with 4KB page

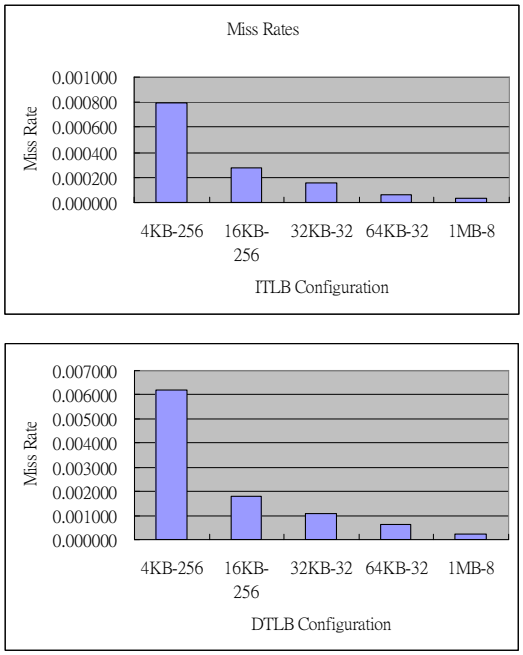


Figure 3-3: ITLB/DTLB miss rate for *compress* with different page sizes and TLB sizes

3-2 The proposed TLB architecture

This section describes in detail of the new TLB structure and mechanism we proposed for embedded processors. The new novel design can be implemented not only in contemporary processors but future high performance processors comprised with billion of transistors. Furthermore, the mechanism is especially suitable to be implemented on processors with larger addressing space than current processors with just 32-bit addressing ability.

3-2-1 Overview

Figure 3-4 shows in detail the proposed TLB structure to reduce the miss rate in context switching. According to the studies of previous section, we'll assume the page size is 32-KB.

However, it should be noted that the study is based on analysis on general desktop requirements. In fact, it can be easily changed to adequate for different page sizes with little configuration changes. Furthermore, we'll have other new study for TLB to provide superpages with page promotion mechanism.

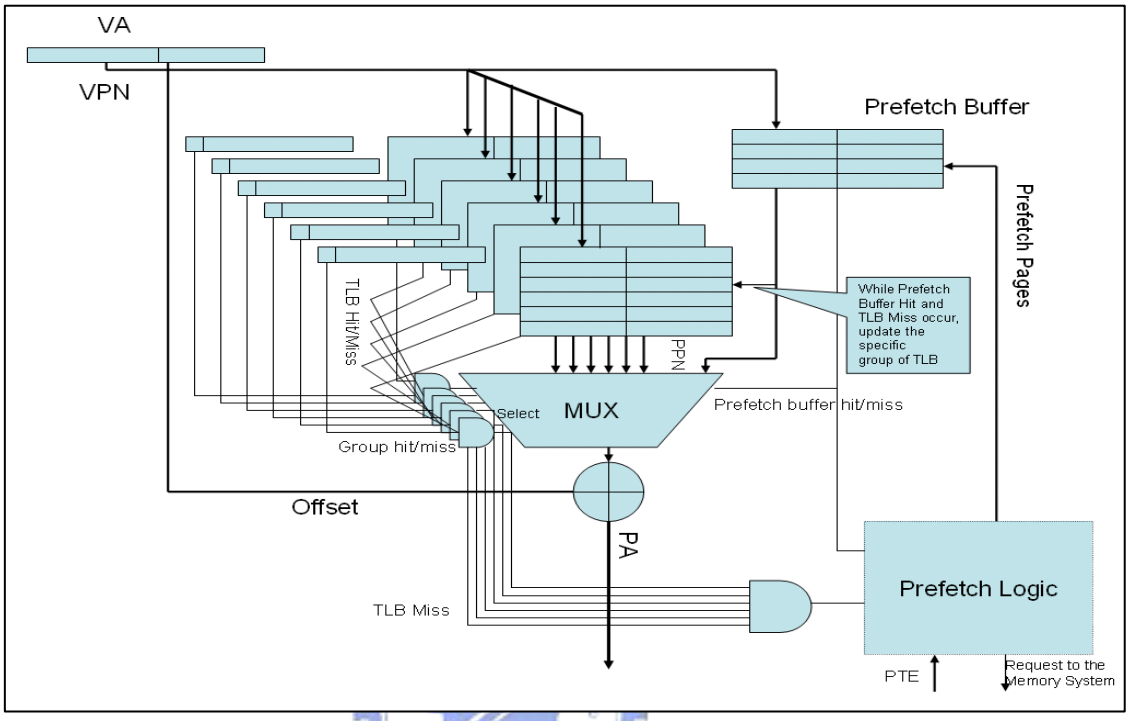


Figure 3-4: The proposed TLB architecture

The proposed structure consists of the following parts – 32 TLB banks with group tags to store the address translations, a multiplexer to select specific TLB banks, a prefetch buffer to store the prefetching entries, and the prefetch & control logic to activate the prefetching mechanism. Each TLB bank has 32 entries and it can be implemented with CAM (content addressable memory) which is commonly used in the traditional TLB. Furthermore, each TLB bank was implemented with fully associativity with the LRU entry replacement policy. That means each bank can be easily implemented the same as traditional design. Thus there are totally 1024 entries in this new design. However, we can easily find that other new processors also try to increase the total entries of their TLB (TLB size) to reduce the possibilities of the TLB misses, such as 1024-entry common TLB for each processor core of IBM POWER4 processor [37]. In addition to the 32 TLB banks, there are also 32 extra registers to store the bank tag for each bank as shown in Figure 3-4. The register contains task tag to identify each task, the current bit to identify the current task, the valid bit to validate a bank, and the LRU

bits to replace the victim bank. It should be noted that the task tag can be any address space identifier (ASID) which the processor itself provides or the PPN (Physical Page Number) of the executing instruction when the context switching occurs on processors without any ASID support (IA32 based processors). On processors without ASID support, the PPN of the executing instruction when the context switching occurs from the PPN field (or last translation) is used. Considering the worse IA32-style case, the PPN is selected; however, the implementation with ASID provided by the processor itself can be more easily. The discussion will be ignored in this paper. However, we still have to point out that we treat ITLB and DTLB as a couple, and they share the same bank tag. That means they stores translations for the same task in the same related bank.

Besides previous discussed parts, the remainder parts are designed for the entry prefetching mechanism. The prefetch & control logic initiates when the TLB misses occurs. When the lookup misses in the current TLB bank but hits in the prefetch buffer, the address translation is generated from that hit entry and it will be inserted into the current TLB bank that is the same as traditional TLB entry replacement. Then, the prefetch & control logic tries to prefetch other entries into the prefetch buffer. If the lookup are missed in both current TLB bank and the prefetch buffer, the traditional address translation mechanism is initiated to generate the correct address translation and then the prefetch & control logic prefetches new entries into the prefetch buffer depending upon the current address. The ‘Prefetch Logic’ can be SP or DP described in [57].

3-2-2 OS Modification

In order to implement the mechanism, the OS is needed to do a little modification. In addition to the page size issue, the OS is required to send ‘the clear TLB signal’ to the processor only when page swapping with disks occurs or page frames release. If the signal is received by the control logic, the control logic should flush all the TLB banks and the prefetch buffer for the worse case example or the corresponding TLB bank and the prefetch buffer for the general cases. Fortunately, it's not hard to realize. In fact, almost all modern

processors, provide some ways to flush TLB entries, such as STA instruction with alternative addresses on SPARC architecture [35,36]. In fact, even IA32 also begins to provide simple way to protect important global entries. [31,32]

3-2-3 Mechanism of the proposed architecture

The proposed TLB structure is divided into 32 banks and once the virtual address is generated from the CPU, the virtual page number (VPN, from the most significant bit to the previous bit of the offset, for example [31:15] in 32-bit addressing environment with 32KB page) is sent to the 32 banks and the prefetch buffer in parallel. Each bank and the prefetch buffer work as the conventional TLB, and the PPN of the hit entries of each bank and prefetch buffer are sent to a multiplexer. In addition, the select signals are obtained from 'AND' of the current bit of group tags and hit signal of each TLB bank, and also the hit signal from the prefetch buffer, to select the correct translation. If it's a hit in current TLB bank, the current TLB bank works as conventional TLB. The physical address can be simply generated by combining the output PPN and the offset from the virtual address. If it's a miss in current TLB bank but a hit in prefetch buffer, the operations are the same as what mentioned in the previous section. However, besides the simplest situation, all other conditions should be carefully handled by the prefetch & control logic. The details will be described in the following paragraphs.

1) No current bit set in all banks: The situation could be happened only when the first instruction fetching after a context switching for ITLB, the system initialization, or swapping pages with disks occurs. In this situation, no valid physical address can be provided via TLB translation. The address should be generated in conventional way by the OS and MMU. After the physical address or address space identifier (ASID) supported by the architecture is generated, it is compared with the task field of bank tags. If any of it is hit with a valid bank tag, the current bit of that bank tag is set, and then the current TLB bank performs as a conventional TLB. On the contrary, if it's a miss, the prefetch & control logic should try to select a victim bank with invalid bit and LRU bits from the bank tag and flush all its 32 entries (both related ITLB and DTLB). Then the current bit of this bank should be set and the

LRU bits of all bank tags should be updated. Then the correct translation is stored into the current ITLB bank entry, and the task tag of the current bank tag should be set. Moreover, it is the generated PPN or ASID provided by the processor that is stored into the task tag field of the current bank tag. Finally, the prefetch logic & control logic initiates the prefetching mechanism that is the same as what mentioned in previous section.

2) One current bit found but no valid translation in both current bank and prefetch buffer: If one current bit is found but no valid translation can be generated, that means the TLB (ITLB or DTLB) reference of the current task is available before but the missed page has not referenced yet. The operation of the current TLB bank just simply acts as a conventional TLB, and no bank tag modification is needed. Then the prefetch mechanism is worked as what mentioned in previous section.

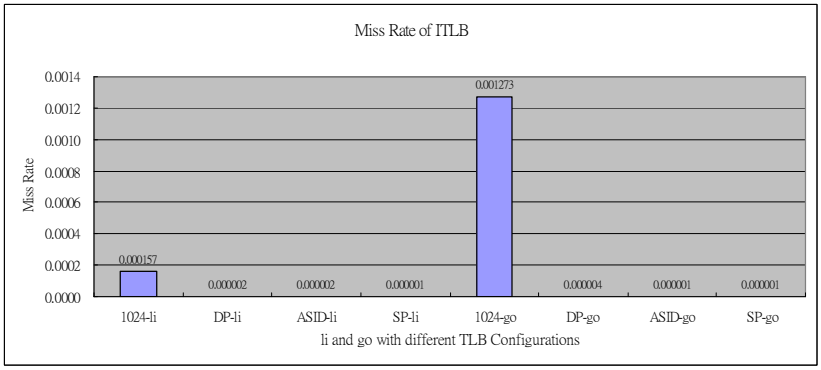
3) Context switching: Once the context switching occurs, the MMU just needs to clear the current bit of the bank tags and flush the prefetch buffer. No more other actions are needed.

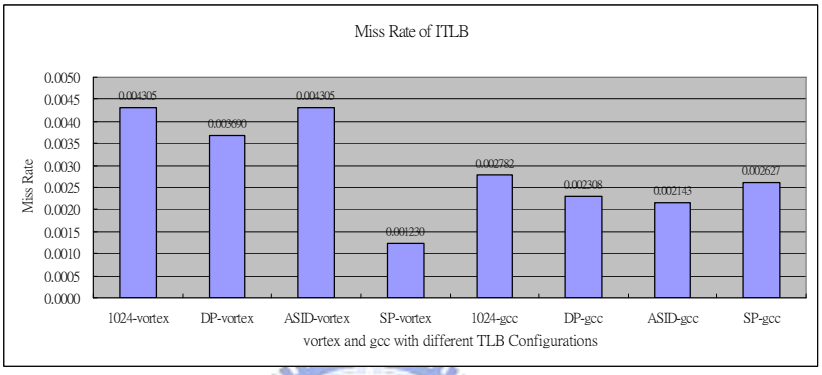
4) Page swapping with disk occurring or page frame releasing: If the page swapping with disks or page frame releasing occurs, the modified OS that we already discussed sends the 'clear TLB signal' to the MMU. Hence, the prefetch & control logic can clear the valid bit of all bank tags on system without architecture supported ASID (*x86*) and flush the prefetch buffer.

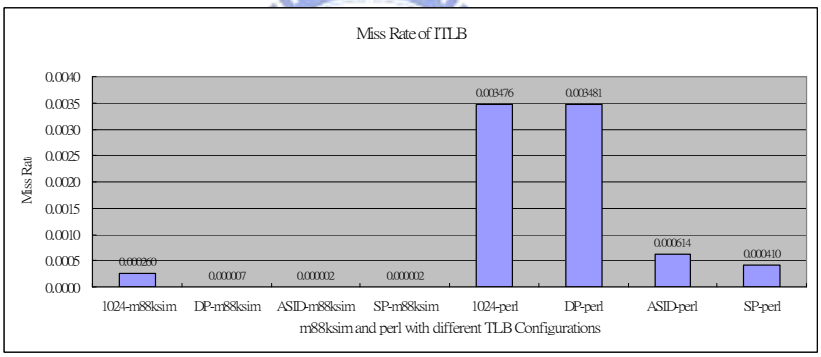
3-3 Performance evaluation of the proposed architecture

All of the simulations were done with modified SimpleScalar Version 3.0d tool suite [37]. The SPEC95 benchmark programs were simulated to estimate the performance. We assume that the context switching would happen after executing one million instructions, and we also assume that the compared 1024-entry TLB is the worse case IA32 (*x86*)-style example. In addition, we compared the miss rates of worse case style 1024-entry

fully-associative TLB with the proposed TLB structure of 32 entries each bank with SP and DP prefetching mechanism after correctly keeping the entries and 1024-entry full-associative TLB with ASID of the same workload assumption with proposed TLB structure. We assume that the SP can prefetch entries with VPN of +9 and -8. That means total 18 entries are prefetched. Moreover, we also assume that the DP can prefetch total 16 entries with 64-row distance table and each row has 2 predicted distance slots. Though we assume the DP with only 16-entry prefetch buffer, the costs of DP is still higher than SP. That's because the extra distance table is required in the DP methodology. Figure 3-5 and 3-6 give the simulation results of SPEC95 benchmark.







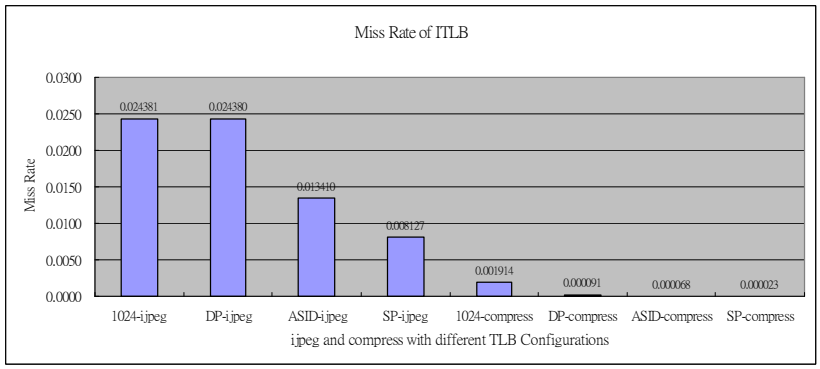


Figure 3-5: ITLB miss rates for SPEC95 benchmarks

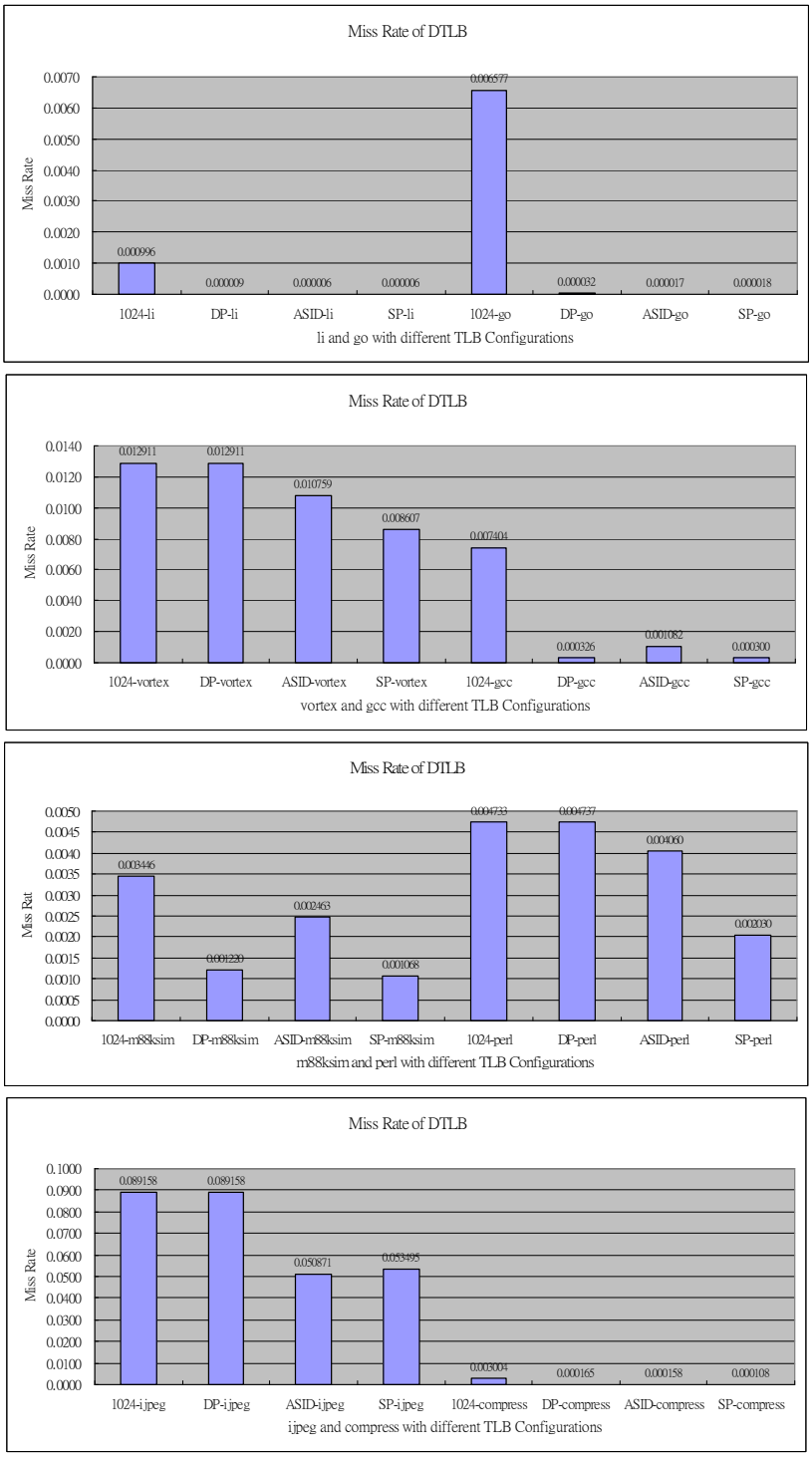


Figure 3-6: DTLB miss rates for SPEC95 benchmarks

Figure 3-5 and Figure 3-6 show the simulation results for ITLB and DTLB with 1024-entry conventional TLB, new TLB structures with DP and SP prefetching mechanism, and 1024-entry conventional TLB with ASID respectively. Observing the simulation results, we can find that our design can deliver better performance than conventional TLB structure if correct TLB entries can be kept. Furthermore, we can also find that the proposed banked TLB

with SP prefetching mechanism can deliver better performance than DP prefetching mechanism and conventional TLB with ASID under multiprogramming environment. Through observing the simulation results, we can also find that prefetching mechanism may be sometimes more important than just increasing more entries. For example, both of the DTLB performances of new TLB structures with SP and DP mechanism are better than conventional TLB with ASID for *gcc*. However, in most cases, the performance of new TLB structure with DP prefetching mechanism is still worse than conventional TLB with ASID. That's because after the context switching occurring the DP prefetching mechanism needs the learning time to fill in the distance table. According to the simulation results, we strongly suggest to use the simplest SP prefetching mechanism in our design.

Even so, we still have to point out several important issues. Firstly, it's not really very fair to assume the conventional fully associative TLB works as the worse case IA32 (*x86*)-style. That's because only some older processors or embedded processors flush their TLBs in context switching. Most modern high-performance processors incorporate their own address space identifiers with TLB tags. These designs, including our methodology, incorporated tags with ASID may have almost the same performance. However, our structure can save some tag bits because of our banking method. As shown in Figure 2-13, it's very clear that the design of TLB entries with ASID tag needs more tag bits than our design. We provide an alternative method to store the ASID. Secondly, it's not a very nice model to assume that context switching occurs after executing each one million instructions. In fact, it may differ from different environments. Most OS defines its own time slice with several milliseconds, and with different processors, the total instructions executed may have enormous differences. In addition, the real situation depends upon real OS running situation. In fact, we seriously consider developing a new generic simulator incorporated with Linux OS to model more accurate real environment. Thirdly, though the page size we assume here is 32KB, it's not very hard to change it to other sizes with some configurations change. In fact, in this thesis we model our asynchronous TLB controller for 4KB page system. Finally, though only a few studies about TLB entry prefetching, it still possible to provide more heuristic prefetching mechanism for TLB entry prefetching. Furthermore, it may be also possible to incorporate other prefetching mechanism with the proposed architecture.

3-4 Discussions of the proposed architecture

The TLB misses cause serious performance degradation on modern processors. In addition, the context switching under the multiprogramming OS may cause this problem even more seriously. However, only some studies focus on the context switching issue. In our work, we presented an alternative TLB mechanism to reduce the miss rate in context switching for embedded processors or microcontrollers. We also discuss how OS should be modified to support this mechanism. Furthermore, we also discuss how to implement TLB entry prefetching mechanism in the proposed architecture. Finally, according to the simulation results, we suggested just simply to use the sequential prefetching (SP) mechanism in this design. Besides the proposed mechanism, we have already begun to find solution to integrate the proposed structure to support superpaging with bank promotion methodology. To obtain more accurate performance evaluation in real environment, the new simulation model and simulator will be developed. In addition, we've already implemented asynchronous TLB controller to support this mechanism in our current new RISC asynchronous processor project. We'll discuss the implementation in the next chapter. We believe that still lots of work should be done in this field. The major features of the proposed TLB architecture are listed as follows.

- An alternative way to reduce TLB misses in context switching with good performance for (asynchronous) embedded processors
- Using banking mechanism to replace per-entry ASID tags
- Adopting prefetching mechanism to reduce compulsory misses
- Easy and simple architecture and operations
  - Bank-based operations to replace per-entry based operations in context switching
  - Especially suitable for asynchronous embedded processors

Chapter 4: Implementation the TLB Controller with Asynchronous Circuits

Because there are no standard ways to implement asynchronous circuits, we'll try to design our TLB controllers with Balsa HDL which is a CSP-based asynchronous HDL, and then the design will be synthesized by Balsa synthesis tool. Thus, the designed circuit can be easily reused or modified. That's why we implemented our design with Balsa HDL.

4-1 Interface

In Section 2-2-4, we have clearly discussed our current asynchronous circuit and system design philosophy. Based on the design philosophy, we'll describe our design sequentially. For an asynchronous circuit design, the design can be conceptually regarded as a black box. Thus, the environment can communicate with it via communication channels. Because the design is based on bundled-data model as shown in Figure 2-19, separate request and acknowledge wires are bundled with data signals. Table 4-1 shows the interface of the proposed TLB controller architecture with bundled-data protocol. As shown in Figure 2-19, these signals can be conceptually divided into 8 communication channels. Because of asynchronous nature, no clock signal is needed. In fact, it's easy to put the design into any design that accepts the same protocols! The 4-phase handshaking protocol is shown in Figure 2-17.

Table 4-1: Definitions of asynchronous TLB interface

Port Name	Direction	Width	Meaning
VA_req	in	1	request signal
VA_data	in	32	Virtual Address
VA_ack	out	1	acknowledge signal
PTE_req	in	1	request signal
PTE_data	in	32	Page Table Entry
PTE_ack	out	1	acknowledge signal
clr_TLB_req	in	1	request signal
clr_TLB_data	in	1	‘1’: clr all TLB banks
clr_TLB_ack	out	1	acknowledge signal
ASID_req	in	1	request signal
ASID_data	in	5	Address Space Identifier
ASID_ack	out	1	acknowledge signal
CMW_req	in	1	request signal
CMW_data	in	1	‘1’: Context Switching occurs
CMW_ack	out	1	acknowledge signal
PA_req	out	1	request signal
PA_data	out	32	Physical Address
PA_ack	in	1	acknowledge signal
PFE_req	out	1	request signal
PFE_data	out	1	‘1’: Prefech an entry from page table
PFE_ack	in	1	acknowledge signal
TLB_hit_req	out	1	request signal
TLB_hit_data	out	1	‘1’: TLB hit; ‘0’: TLB miss
TLB_hit_ack	in	1	acknowledge signal

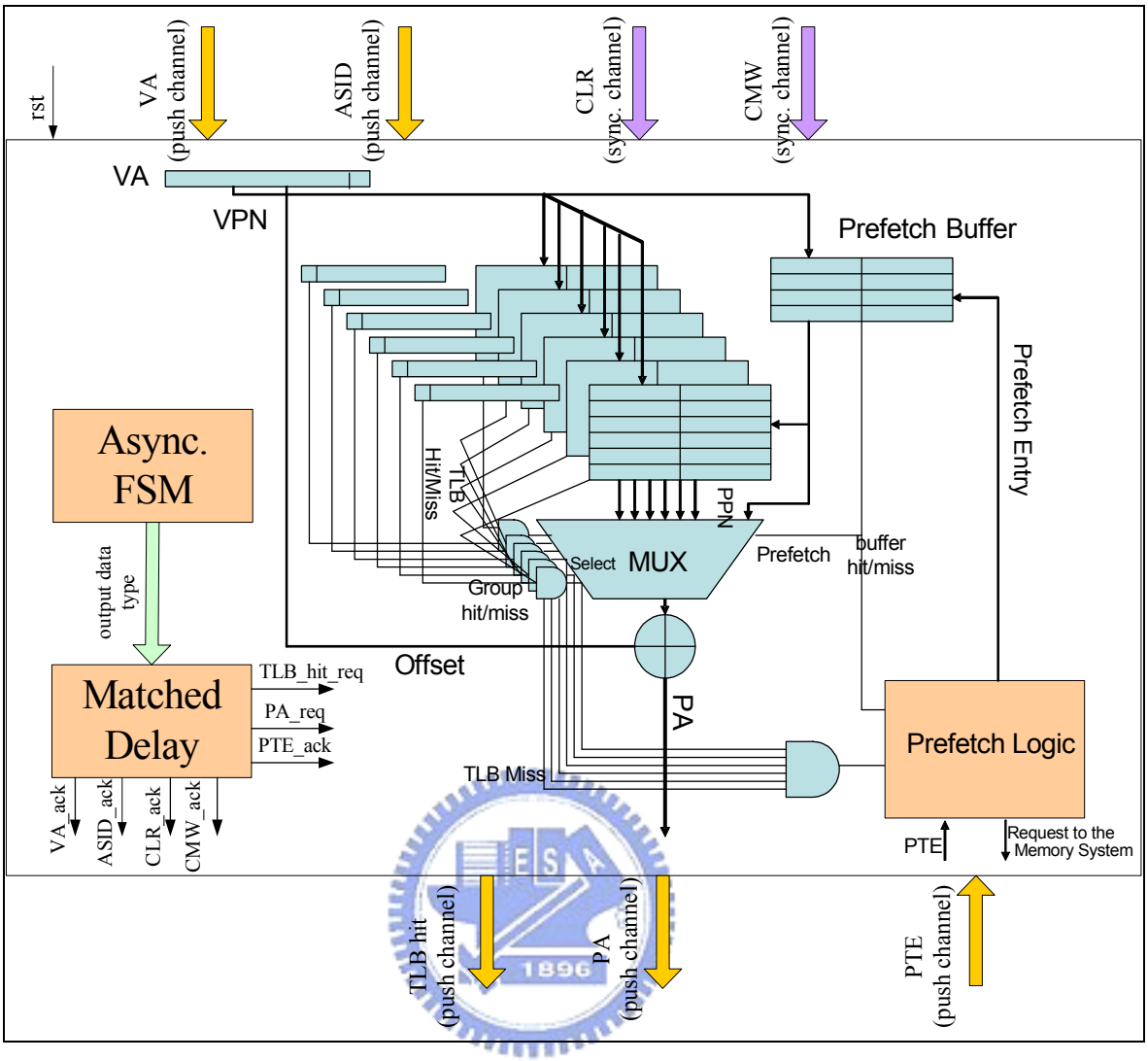


Figure 4-1: Block diagram of the TLB interface

4-2 The Balsa Framework

We have already pointed out that it's not easy to implement asynchronous circuits directly with gate-level and RTL descriptions of traditional HDL. In addition, with such "fixed" descriptions, you cannot change the handshaking protocol that you have already implemented. On the contrary, if you model your design with Balsa HDL or other asynchronous HDL such as Haste description language [90], or Java and DDG of SoCAD [100], the target handshaking protocol you wish to use can be decided during synthesis phase. Your design therefore can be changed to any handshaking protocols which supported by your asynchronous tool. You can put more efforts on your asynchronous algorithm and architecture

design! That gives you more design and implementation flexibility. We select Balsa as our tool not only because it's the most popular open source solution, but also because it have been already used in many successful design. In addition, the APT (Advanced Processor Technologies) group of the University of Manchester can affords some needed supports for users [106].

Figure 4-2 shows the Balsa design flow. The Balsa back-end can generate gate-level netlists that can be imported into target CAD systems. Balsa now supports three commercial EDA tools: Compass Design Automation tools from *Avant*, *Xilinx FPGA* design tools and *Cadence Design Framework II*. It supports three back-end protocols for use with each technology: bundled-data scheme using a 4-phase-broad/reduced-broad signaling protocol, a delay-insensitive dual-rail encoding and a delay-insensitive 1-of-4 encoding. Thus it makes it easy to design asynchronous circuits or systems for these three protocols with Balsa HDL.

To generate netlists for target CAD systems, the Balsa system make use of basic cells of these target CAD systems, such as AND, OR, NOR, XOR, NADN, BUF, XNOR, INV, FD (D-type flip-flop), FDC and FDCE of *Xilinx FPGA* technology. In addition, it also provides specific cells needed for asynchronous circuit implementation. The most important of all is the Muller C-element as described in Section 2.2. In addition to the C-element, it also provides special designed cells. Figure 4-3 shows the NC2P element. Once the input i0 is equal to 0, the output will be 1. When both inputs are 1, the output will be 0. Finally, if the input i0 is 1 and i1 is 0, the output value will not be changed. With NC2P element, the S-element which performs a series of handshaking can be constructed. Figure 4-4 shows the S-element and its behavior. The S-element has 4 input that includes 2 request/acknowledge handshake pairs – 'Ar'/'Aa' and 'Br'/'Ba'.

With these basic cells, the Balsa system provides total 40 handshake components. We'll describe some of them here. Figure 4-5 shows the fetch component which is the most common way to control a datapath from a control tree. It transfers input data to variable, from variable to output channel, or from variable to variable. Figure 4-6 and Figure 4-7 are the symbol of sequence and concurrent components. The sequence components output the control signals in sequence, and the concurrent component outputs the control signals in parallel. They can be used to activate a number of operations.

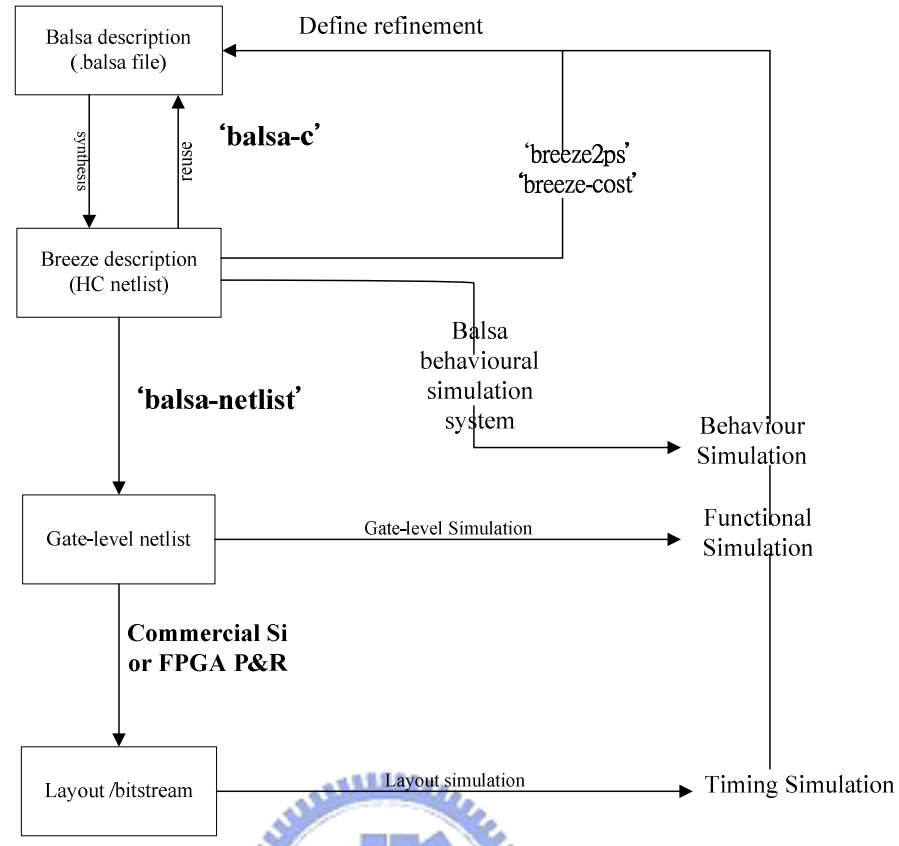
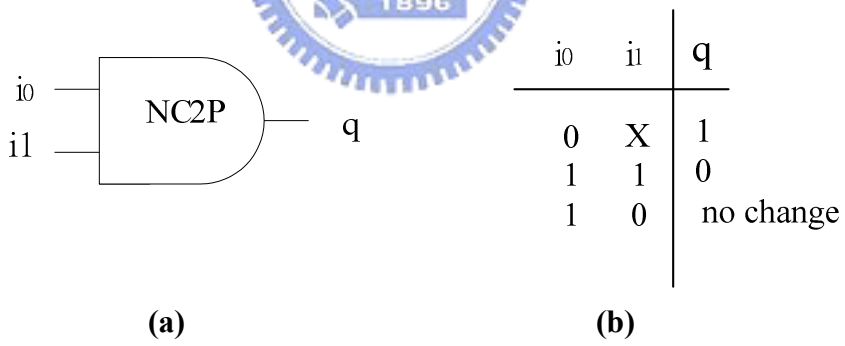
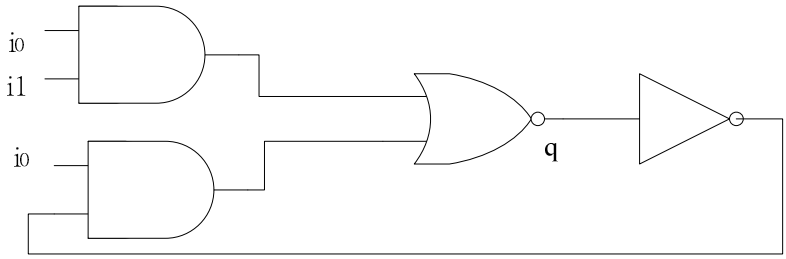


Figure 4-2: The Balsa design flow



(a) **(b)**



(c)

Figure 4-3: The NC2P-element (a) symbol (b) true table (c) gate-level implementation

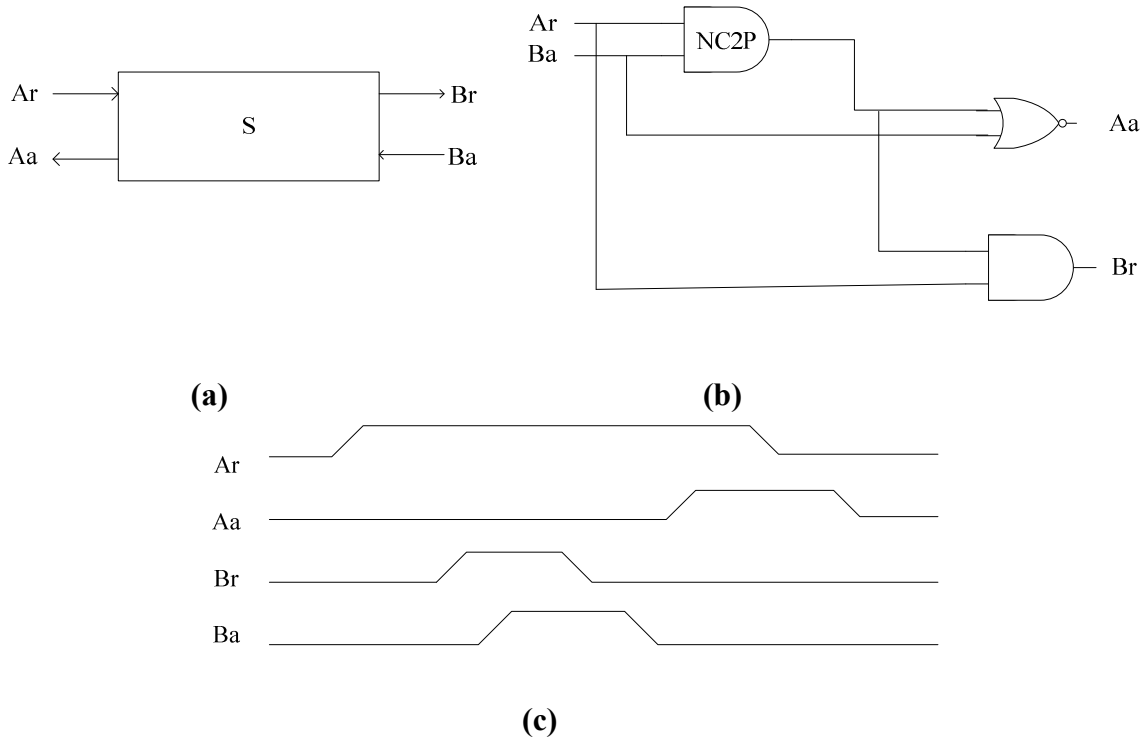


Figure 4-4: The S-element (a) symbol (b) gate-level implementation (c) handshaking protocol

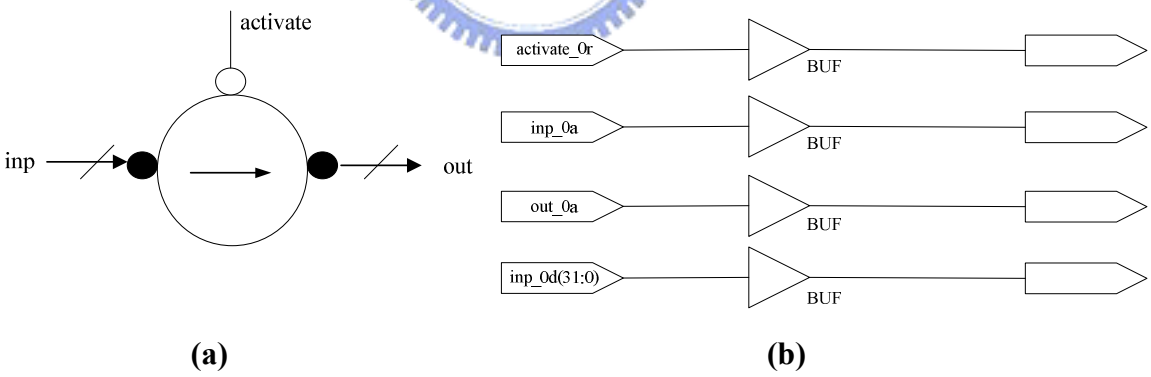


Figure 4-5: The Fetch component (a) handshake component (b) gate-level implementation

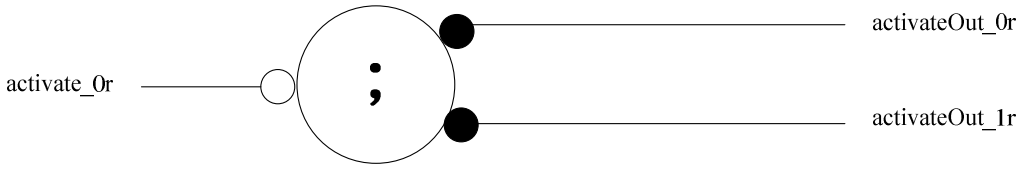


Figure 4-6: The Sequence component

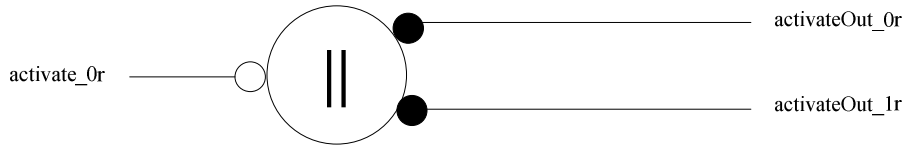


Figure 4-7: The Concurrent component (a) handshake component

4-3 The Design with Balsa

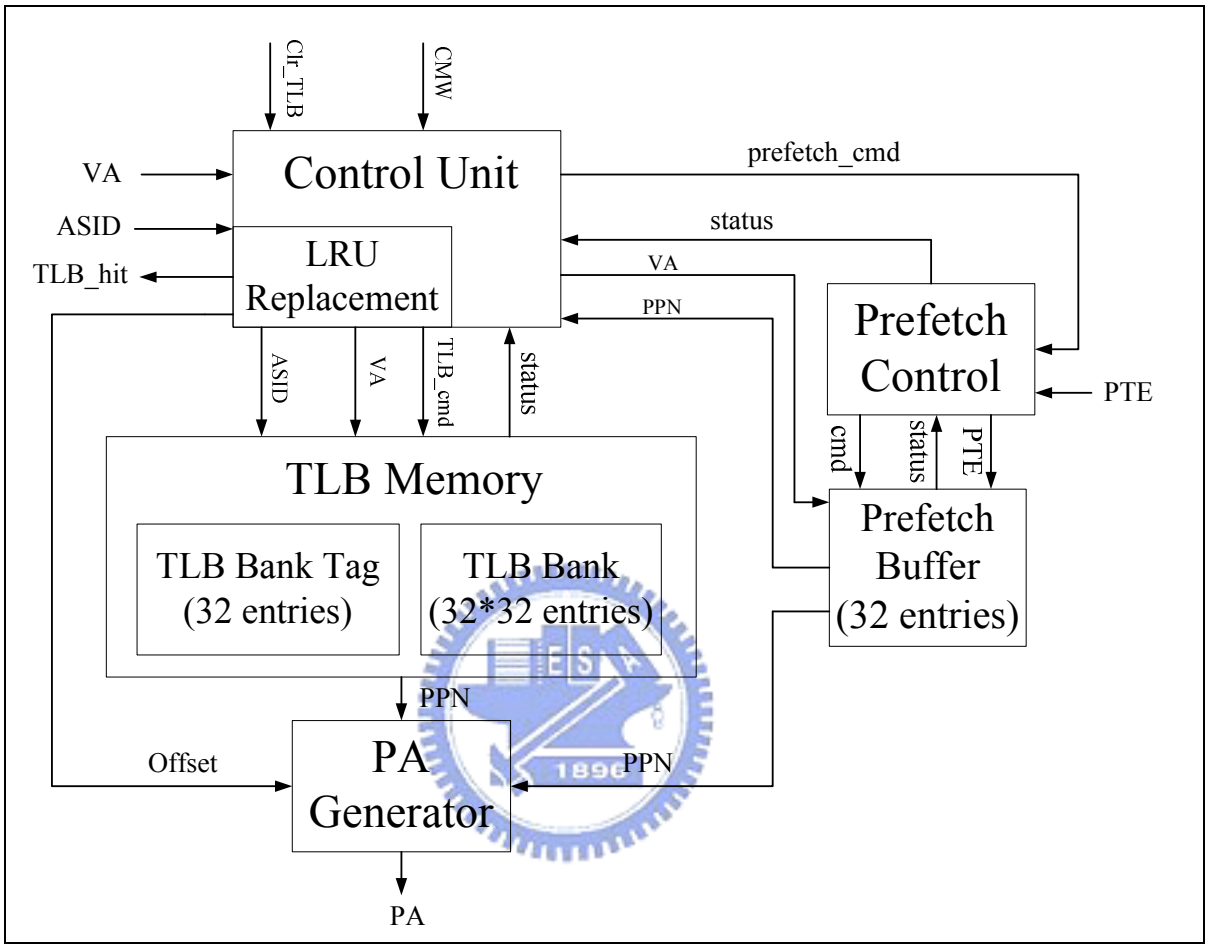
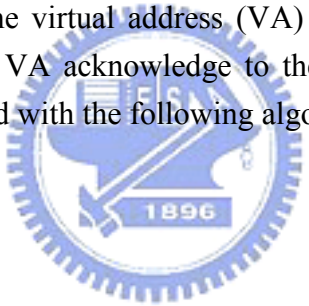


Figure 4-8: Architecture of asynchronous TLB modeled with Balsa HDL

The design of proposed architecture with Balsa will be described in this section. Different from what described in Weigel's work [105], we focus on modeling an advanced TLB architecture while they focused on implementation of MMU. Thus, the TLB they modeled is only simple architecture that supports lookup and flush functionality. In our work, we implemented the advanced TLB architecture that we proposed. According to the interface defined in Figure 4-1 and table 4-1, we carefully designed our proposed architecture. Figure 4-8 shows the architecture of the whole design. The design is divided into the following parts — *TLB Memory*, *Control Unit*, *Prefetch Control Unit*, and *Physical Address Generator*. The following Balsa descriptions show the top module definitions of the whole design. With this definition, we can easily observe the communication channels that connect the design and the environment. In fact, that's what we defined in section 4.1.

```
procedure TLB_CTRL_TOP(
--input channels
   input iCLR_TLB: bit;
   input iCMW: 1 bit;
   input iASID: 5 bits;
   input iVA: 32 bits;
   input iPTE: 32 bits;
--output channels
   output oPA: 32 bits;
   output oHIT:    bit
) ...
```

In order to implement the proposed architecture, the TLB controller algorithm was designed carefully. It should be noted that because it's an asynchronous implementation, the data signals from the sender will not be cleared until the receiver replies the acknowledge signal. That also means that the virtual address (VA) from the processor will be remained until the controller returns the VA acknowledge to the processor. Thus, the proposed TLB architecture can be implemented with the following algorithm.



Asynchronous TLB controller algorithm:

```
if   ('clr_TLB channel' is activated)
     clear valid bits in all TLB banks ()
   || flush the prefetch buffer ()
else if ('CMW channel' is activated)
     clear current bits ()
   || flush the prefetch buffer ()
else if ('VA channel' is activated)
   if (no current bit is set in all TLB banks)
       (activate 'ASID channel'
      || activate 'TLB_hit channel')          // TLB_hit_data = '0', indicating TLB miss
      ; if (ASID is found in a TLB bank)
            set current bit in this TLB bank ()
        else                                  // ASID is not found
            (select a victim bank with invalid bit or according to LRU bits ()
            ; flush 32 entries in the victim bank ()
            ; set current bit, task tag, and valid bit
            ; update all LRU bits)
            || activate prefetch mechanism ()   // activate PTE channel multiple times
   else if (TLB hit)
      activate 'PA channel' || activate 'TLB_hit channel'
   else if (TLB miss)
      if (prefetch buffer hits)
         (put this entry into TLB bank() || activate 'TLB_hit channel' ||
          activate 'PA channel')
         ; activate 'PTE channel'
      else if (prefetch buffer miss)
         activate 'TLB hit channel'
         ; activate 'PTE channel'
```

With Balsa HDL, designing the proposed TLB architecture can be simplified as describing each part of the TLB architecture with Balsa HDL high-level descriptions. Each part of the design will be described in detail in the following paragraphs.

TLB Memory:

There are total 32 TLB banks of the TLB memory architecture. Each of the TLB banks is

composed of 32 entries. Because the real implementation of memory cells themselves should be closely related to the process and should be additionally designed carefully, we only modeled the TLB memory with Balsa data structure and simulated under the Balsa environment with Balsa block memory. The data structure of TLB entry is defined as show in table 4-2.

Table 4-2: Structure of each TLB entry

Field	**valid**	**lru**	**tag**	**ppa**
bit	35	34	33...17	16...0

In chapter 3, we have already discussed the proposed TLB architecture. In the proposed architecture, each TLB bank has its associated TLB bank tag. Therefore, there are total 32 entries in the TLB bank tag. We defined total 4 fields for each entry. It should be noted that in order to simplify the design, we only defined 5 bits to represent the ASID. In fact, in most systems the width of ASID is 8 bits. In addition, we use the 5 bits to index the 32 banks. Thus the data structure of each bank tag is defined as table 4-3.

Table 4-3: Structure of each TLB bank tag

Field	**ASID**	**current**	**valid**	**lru**
bit	7...3	2	1	0

With the previous definitions, the TLB entry and TLB bank TAG entry can be easily described in the following Balsa descriptions. In fact, it's really easily to describe TLB memory model with Balsa HDL. Following Balsa descriptions depicts the data structures that define the TLB entry and the TLB bank tag.

```
type TLB_ENTRY is record
   valid: bit;
   tag: 17 bits;
   ppa: 17 bits;        --Physical Page Address
   lru_bit: bit
end

type TLB_BANK_TAG is record
   task_tag: 5 bits;
   cur_bit: bit;
   valid: bit;
   lru_bit: bit
end
```

In addition, we also modeled all TLB bank control with Balsa HDL. Following Balsa descriptions depict how we modeled the valid bit clear, LRU bit clear, and TLB entry search.

```
shared clr_all_valid is begin
 for ; i    in 0..31 then
    tlb_bank[i].valid := 0
 end -- for loop
end

shared clr_all_lru_bit is begin
 for ; i in 0..31 then
    tlb_bank[i].lru_bit := 0
 end -- for loop
end

shared search_tlb_bank is begin
 if tlb_bank[0].valid =  1 and tlb_bank[0].tag  = tmp_tag then tmp_hit := 1 ||
    tmp_ppa := tlb_bank[0].ppa
 |  tlb_bank[1].valid =  1 and tlb_bank[1].tag  = tmp_tag then tmp_hit := 1 ||
    tmp_ppa := tlb_bank[1].ppa
…
```

Control Unit

The control unit communicates with the environment through these handshake channels, clr_TLB (pull channel), CMW (pull channel), TLB_hit (push_channel), and VA (pull channel). Once one of these channels is activated, this control unit will adopt the corresponding actions. It should be noted that in this implementation these pull channels can not be activated simultaneously to avoid the occurrence of deadlock. The control unit will issue control commands to the TLB memory when it is required to change the content of TLB memory, check TLB hit or miss, or perform the entry replacement.

The 1-bit LRU (Least Recently Used) replacement policy is adopted by us to reduce the overhead in hardware cost. Because there's no global clock signal, we implemented our LRU algorithm as follows. Initially, all LRU bit are all cleared. When TLB hit or replacement occurs, the LRU bit at assigned entry is set. The entry with LRU bit equal to 0 is replaced first. Once the LRU bits in all entries are set and replacement is needed, all LRU bits are cleared.

The 4-bit data signal of the TLB command is defined as table 4-4.

Table 4-4: Structure of 4-bit TLB command

Field	clr_valid	clr_lru_bit	search_va	lru_replace
bit	3	2	1	0

clr_valid: clear all valid bits in current TLB bank
clr_lru_bit: clear all lru bits in current TLB bank
search_va: seach ppa in current TLB bank
lru_replace: perform LRU replacement in current TLB bank

Finally, only 1 bit data signal of TLB_hit channel to indicate the status of TLB. It is defined as table 4-5.

Table 4-5: Data signal of TLB_hit channel

Field	hit
bit	0

hit: indicate whether TLB hit or not

Prefetch Control Unit

To reduce the complexity, we implemented the simplest way of prefetch control unit. Once an empty entry exists in the prefech buffer, the prefetch control unit will fetch a new page table entry through the channel PTE. Then, the Prefetch Control Unit requires the prefech buffer to store it.

We have described that the Balsa descriptions will be synthesized into handshaking components netlists. Figure 4-9 shows the handshaking component graph of CU and prefetch control unit. Observing this graph, all our Balsa descriptions were mapped into built-in handshaking components of Balsa framework. These handshaking components then can be used to synthesize into gate-level netlists depending on the selected target handshaking protocol.

Figure 4-9: Handshaking component graph of CU

PA Generator

The PA Generator generates the physical address via concatenating the offset of virtual memory and physical page number (PPN) generated by TLB. Finally, the generated physical address is sent out through the PA handshake channel. In fact, it's the simplest part of the design. Figure 4-10 shows the handshaking components graph of the PA generator. In this graph, we can easily find that it selects one of the hit PPN from prefetch buffer or TLB bank output and combines it with the offset to generate the output PA.

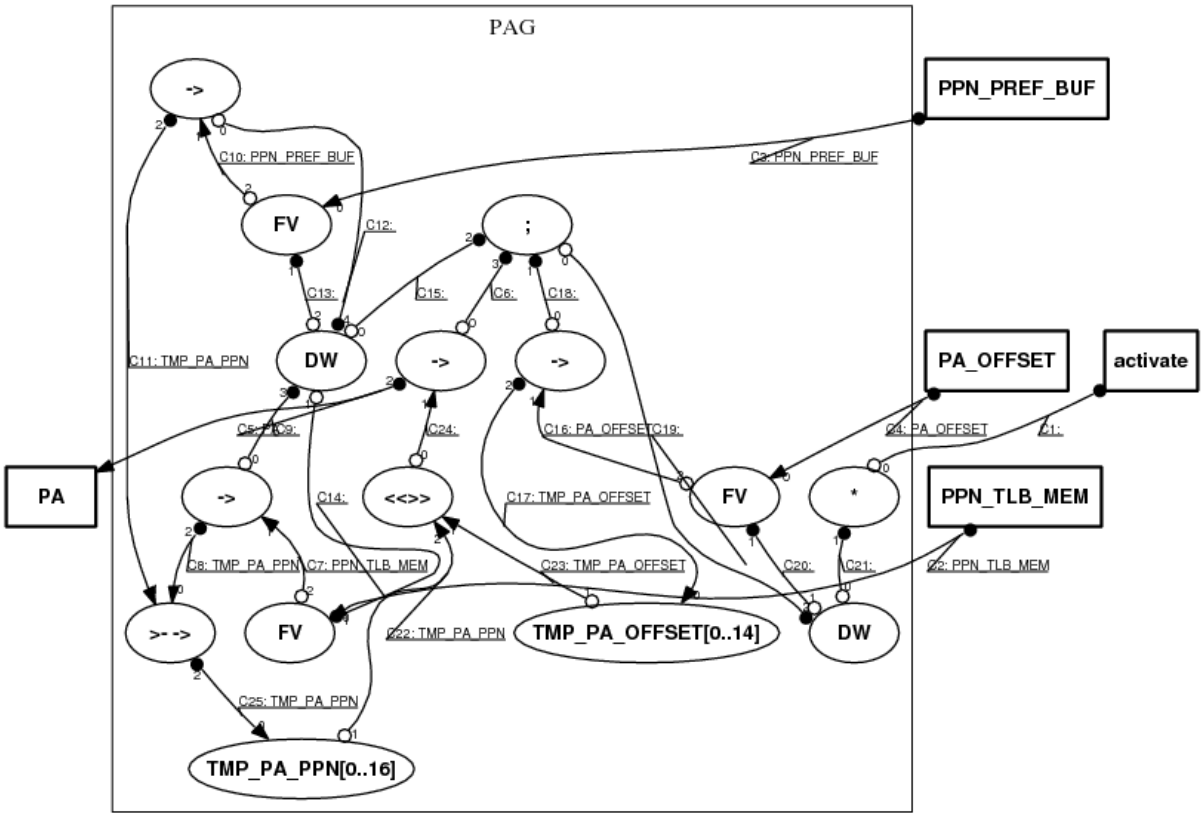


Figure 4-10: Handshaking component graph of PA Generator

4-4 Implementation

Finally, the designed was verified manually with random pattern under Balsa environment. Because only several channels should be verified, the design can be verified easily. Thus, we verified the design via monitoring the communication channels. Figure 4-11 shows the waveform of these communication channels. However, it's not a good and formal way to verify the functionality. We'll discuss this issue in the next chapter. The gate-level

netlist was then generated with Balsa tools, and synthesized with Design Compiler with TSMC 0.13 μm process. Table 4-6 shows the equivalent gate count (NAND gate).

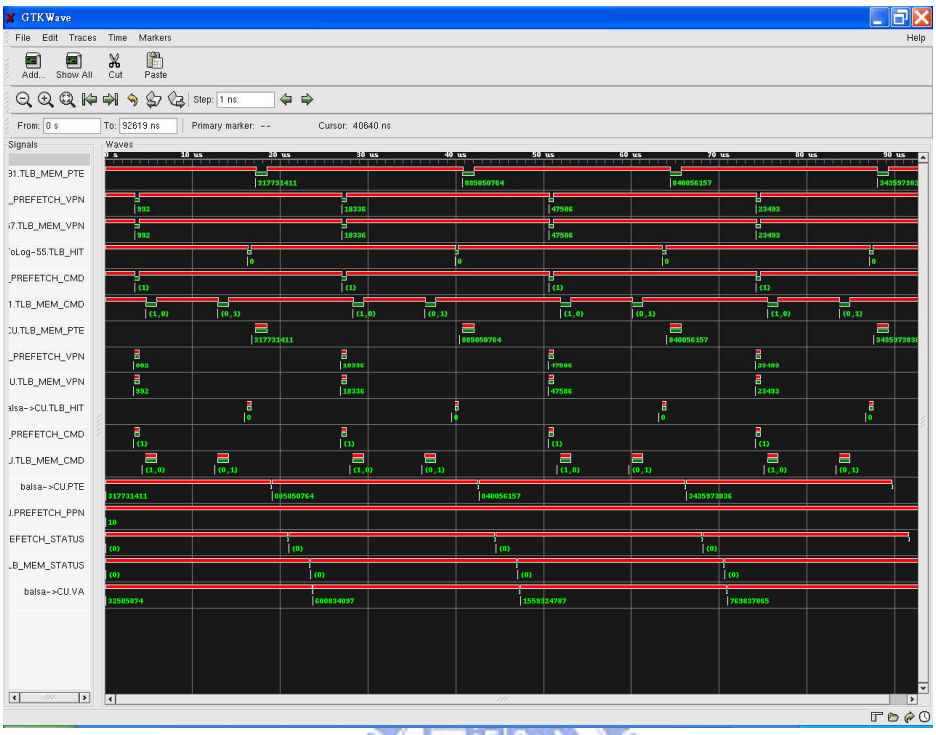


Figure 4-11: Waveform of circuit simulation

Table 4-6: Equivalent gate count

	Equivalent Gate Count (NAND gate)
CU and Prefetch Control	1,441
TLB_Memory	687,119
Total	688,560

not include memory and matching delay elements

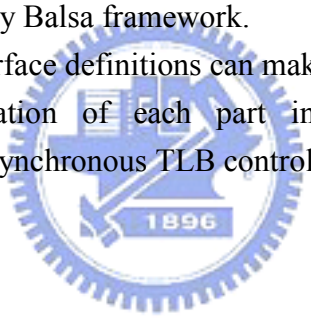
The costs are really high, in fact, far from our estimation. However, it's possible. The result in [97] that was synthesized with the same handshaking protocol also shows that the costs of circuits generated by Balsa tool suite may not be very cheap. In that design, total 3,134,953 gates are used to implement Asynchronous MP3 decoder with Xilinx FPGA. That's because all Balsa descriptions will be translated into handshaking components, and the cost of these components may be not too cheap.

However, we found that the costs of CU and prefetch control unit are only 1,441. That's because in our real implementation the CU and prefetch control parts only needs to handle the input signals and send correspondence signals out. Unfortunately, the total equivalent gate

count of control circuits of TLB bank and prefetch buffer are 687,119. That's not only because we have to design lots of different functionalities of TLB memory parts, but we also found that besides the control circuits that we modeled to control the behavior of our design the Balsa framework also adds lots of extra control circuits to control the memory models. We had already tried to model our circuits with as many as possible shared components. It's still very expensive. Maybe these parts should be designed with gate-level directly.

We also estimated the costs of CU and prefetch control unit with 4-phase dual-rail protocol. Though we expected that the costs may be doubled, it didn't occur. The equivalent gate count is 2,450. That's really interesting. That's because the dual-rail circuits may also be possibly implemented in reasonable costs. However, it's not reasonable to have any "dual-rail" memories. Thus it's not reasonable to implement any memory related components with dual-rail protocol. That's why we implemented our design with 4-phase bundled-data protocol. Following items list the major features of the implemented Balsa model.

- Modeled with Balsa HDL, the TLB controller can be synthesized into handshaking protocols supported by Balsa framework.
- Simple and clear interface definitions can make the design be used easier.
- Unambiguous separation of each part in real asynchronous design makes verifications of the asynchronous TLB controller easier.



Chapter 5: Conclusions and Future Works

In this thesis, we proposed a novel TLB architecture for asynchronous embedded processors. In addition, we also modeled it with Balsa HDL which is a CSP-based asynchronous HDL. We demonstrated how to transfer the proposed architecture into asynchronous circuits. In this chapter, the conclusions and future works will be summarized.

5-1 Conclusions

The computing devices have enormous changing for the past decades. Only recent years, the embedded systems and mobile devices have been becoming the major trend in computing devices. For the past years, because early applications of these systems are simple, no extra complex operating systems are needed. However, new embedded systems and mobile devices have begun to support very complex operating systems, such as Windows® Mobile and embedded Linux. Google even tries to provide very powerful software stack platform based on embedded Linux called Android [107]. All these new applications need very efficient supporting for embedded operating systems. Traditional design needs specific microcontroller or processor to execute OS, and other DSP or accelerator processor to boost computing performance. Recently, some designs try to provide an alternative solution. These designs integrate both general purpose processor and DSP or accelerator processor into a single processor, such as cores of Blackfin [108] and TILE processors [109]. All these new trends demonstrate the importance of OS in embedded systems or handheld devices. In order to provide high performance address translation from virtual address to physical address of modern OS, the high efficient TLB design is needed. The TLB misses cause serious performance degradation on modern processors. In addition, the context switching under the multiprogramming OS may cause this problem even more seriously. However, only some studies focus on the context switching issue for embedded processors. In our work, we presented an alternative TLB design to reduce the miss rate in context switching for embedded processors.

In addition, it is widely known that synchronous circuit has some disadvantages, such as clock skew, higher power consumption, worse-case performance, and poor reusability. However, asynchronous circuit can easily address these problems. In addition, asynchronous circuit has higher reliability and robustness than its synchronous counterparts. In fact, all these are all critical issues for embedded processors or microcontrollers. But it's very hard to implement digital systems with asynchronous circuits.

In our work, we implemented the proposed TLB controller for the proposed TLB architecture with asynchronous circuits. We implemented our proposed TLB controller with the 4-phase bundled-dada handshaking protocol. The bundled-data model was implemented with Balsa HDL which is a CSP-based asynchronous HDL. With the Balsa HDL, we can focus on the asynchronous architecture and algorithm designs without considering too much on the handshaking protocol issues. In addition, because several target handshaking protocols are supported by the Balsa tools, you don't need to implement each HDL model for each handshaking protocol. Thus, higher flexibility can be provided. Unfortunately, the synthesized result shows that total equivalent gate count of the TLB controller without memory is 688,560. That's really not cheap. However, we also found that the CU and prefetch control parts are not very expensive. It costs only 1,441 equivalent gates, but the TLB memory parts costs 687,119 equivalent gates. That's not only because we modeled lots of functionalities for this part but also lots of extra memory control circuitry is added by Balsa tool suite. However, we still successfully demonstrated an advanced asynchronous TLB controller than other related works. Thus, the following items are the main features of the proposed asynchronous TLB controller.

- An alternative TLB architecture was proposed to reduce the miss rate in context switching for the asynchronous embedded processor.
- Instead of per-entry ASID, TLB banking is used to separate different address space.
- Simple TLB entry prefetching mechanism is used to reduce some possible compulsory misses.
- Modeled with Balsa HDL, the TLB controller can be synthesized into handshaking protocols supported by Balsa framework.
- Simple and clear interface definitions can make the designed be used easily.

- Unambiguous separation of each part in real asynchronous design makes verifications of the asynchronous TLB controller easier.

5-2 Future Works

In this thesis, we propose an alternative TLB architecture to reduce miss rate in context switching for asynchronous embedded processor. As mentioned in section 3-2, to estimate miss rate more accuracy the simulator should be integrated with OS. Therefore, new simulator model should be developed for further study. In addition, as mentioned in section 3-1, the performance of TLB not only relies on miss rate but also miss penalty. That's means the execution time should be taken into consideration. However, because lack of information of processor architecture and memory system, it's not very easy to estimate it directly. The design should be placed into a real processor.

As mentioned before, most asynchronous processors today are very simple; thus, most of them do not support virtual memory. In our work, we hope to provide a general asynchronous TLB architecture that can be implemented in asynchronous processors. That's why we modeled our design with Balsa HDL. With high-level asynchronous HDL, the design can be synthesized into all supported handshaking protocols by Balsa tool suite. However, the Balsa tool suite cannot provide the real TLB memory; thus it should be implemented separately. In this work, we only simply use latches to replace the real TLB memory for verification. It's not reasonable. Thus, this part should be carefully handled in our future work. In addition, as the analysis in section 4-4, besides the functionalities we modeled to control behavior of the TLB memory the extra circuitry added by Balsa tool suite is very huge. The part really should be redesigned manually in the future.

Finally, our goal is to design our own asynchronous RISC core with virtual memory support for embedded systems or handheld devices. In fact, we hope to design asynchronous-based SoC or MPSoC with our own asynchronous processor core. As mentioned in section 2-2-3, there are some studies of asynchronous interconnections and GALS. In fact, the clock issue has been becoming one of the most critical issues in large SoC designs. As mentioned in section 1-2, ideally, asynchronous circuits may make software "OOP"-style design on hardware possible. Imaging, without the global clock issue, designing SoC might be a little like playing the LEGO® bricks. Ideally, each asynchronous IPs can be plugged in the design if they "talk" the same "handshake protocol." That's why we hope to design our own asynchronous processor core. The design of asynchronous TLB is one of the

critical parts of the asynchronous processor core. In order to verify our future processor core more formally, we'll suggest a new asynchronous processor design flow that can support not only architecture exploration but also facilitate hardware/software co-design. We'll discuss this topic in the next section.

5-3 Verification Issue for future work

In traditional synchronous based design, the verification can be easier than that of asynchronous ones. You can verify your design based on the "clock." That means that you can verify the status of the design based on the clock cycles. Figure 5-1 shows a very simplified VLSI design flow. The design ideas are described in cycle-based functional specification descriptions. Traditionally, the functional specification can be described with C programming language. Thus, the cycle-based simulator can be used to prove the design ideas. Then, the design will be implemented in RTL/gate-level design. To verify the implementation, the two models will be verified via cycle-by-cycle cross-verification. Finally, the design can be transferred into layout. Certainly, the cycle-based equivalence checking should be done between RTL/gate-level design and layout. On the contrary, without the global clock, each part of the design may work in its own speed and it's not easy to make sure if the design operates correctly in any specific time. It will be even worse that the operation times of the same component may be also different depending upon the input. That's especially on most DI/QDI designs. You can be very sure what status should be of your design at 10^{th} cycle, but how can you do the same thing on system without clock? Imaging in a 2-phase bundled data design and given a specific time, how can you make sure the status should be? As mentioned in section 2-2-2, in such systems each part of the design may begin to operate whether the request or acknowledge signals are rising edge or falling edge. Verifications of different models of asynchronous circuits may also be a good research topic.

We have already pointed out that lots of new issues should be carefully dealt with in developing embedded processors. Because most of these problems can be resolved with asynchronous circuits, that's why we put lots of efforts in developing asynchronous processors. In addition, because of some new application requirements, new features should be supported by these processors. However, it's important to do some architectural explorations before these features can be supported. Thus, we'll suggest a design flow that can be used to design new asynchronous embedded processors from architectural exploration to functional verification. Figure 5-2 shows our new design flow. We'll introduce the use of architecture description language (ADL). LISA will be selected as our design tool [110]. That's not only because LISA is the most popular and successful ADL but also it's a mixed

structural and behavioral ADL. Thus, the design described with LISA can be used to generate simple toolchains including (compiler), assembler, linker, and simulator. It can also be used to generate RTL of Verilog HDL. Thus, hardware/software co-design can be easily achieved. CoWare® Inc. now provides a complete GUI IDE based LISA development environment called CoWare® Processor Designer [111]. With CoWare® Processor Designer, it makes LISA easy to learn and use. The first, the design specification should be implemented with LISA descriptions manually. Then the CoWare® Processor Designer can be used to generate toolchains and simulator. It should be noted that in order to achieve the goal of hardware/software co-design the application software can be developed simultaneously. In addition, if the designed architecture is described in structural model, the RTL can also be generated. Though the RTL model generated is not a very efficient implementation, it still can be used as reference synchronous model for evaluation. In fact, after simulator and toolchains can be generated, the performance of designed architecture can be roughly estimated. Then the generated simulator can be used as golden model in order to do cross-verification with new designed asynchronous processor. However, because it's impossible to do clock-by-clock cross-verification with asynchronous circuits, we suggest using the "instruction-based" cross-verification. That means we can compare the execution results instruction-by-instruction. With this design flow, we can develop our new asynchronous embedded processor more effectively.

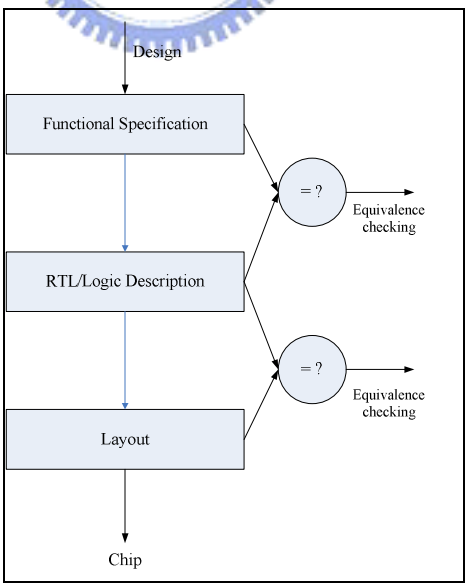


Figure 5-1: Simple VLSI design flow

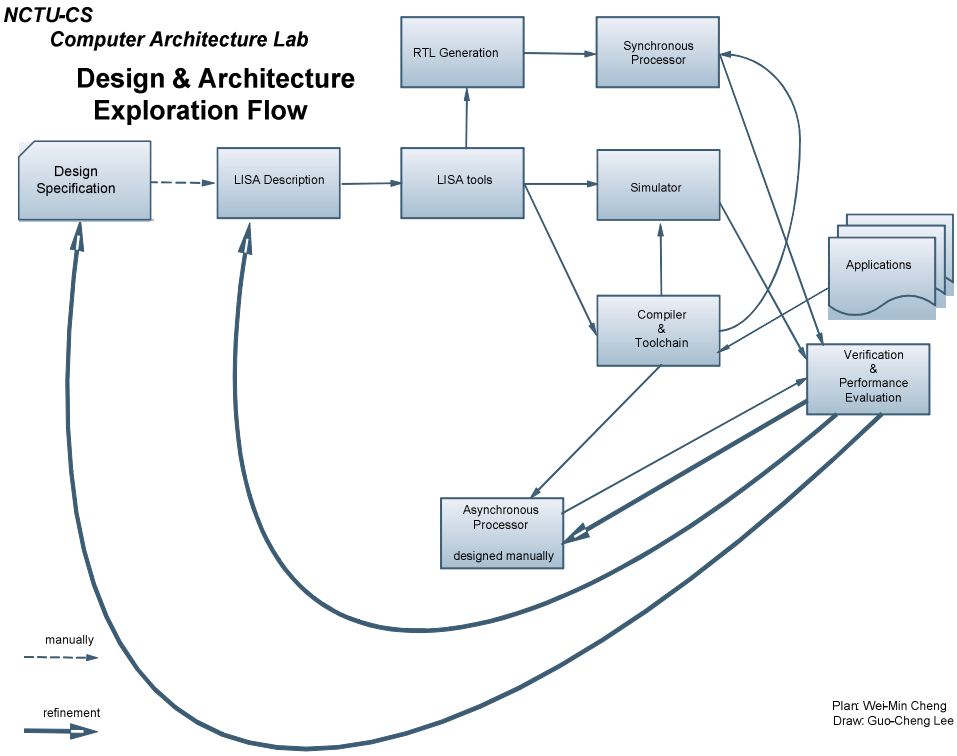


Figure 5-2: Our asynchronous processor design flow

Reference

[1] J. Liedtke, “Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Adress Spaces,” *GMD Technical Report*, No. 933, German National Research Center for Information Technology, Nov. 1995.

[2] A Wiggins, G Heiser, “Fast Address-Space Switching On The StrongArm SA-1100 Processor,” *Technical Report, UNSW-CSE-TR-9906*, The University of New South Wales, Austrila, 1999.

[3] A Wiggins, G Heiser, “Fast Address-Space Switching On The StrongArm SA-1100 Processor,” in In Proceedings of the 5th Australasian Computer Architecture Conference (ACAC), 2000, pp. 97 – 104.

[4] I. E. Sutherland and J. Ebergen, “Computers without Clocks,” Scientific American, August 2002, pp. 62-69.

[5] A. Davis and S.M. Nowick, “An Introduction to Asynchronous Circuit Design,” *Technical Report, UUCS-97-013*, Computer Science Department, University of Utah, Sep. 1997.

[6] S. Hauck, “Asynchronous design methodologies: an overview,” *Proceedings of the IEEE*, Vol. 83, Issue 1, Jan. 1995, pp.69-93

[7] A. Bink and Mark de Clercq, “ARM996HS Synthesizable CPU with Clockless Technology,” *Information Quarterly*, Vol. 5, No. 4, 2006, pp. 20-24.

[8] Neil H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective*, 2ed. Addison-Wesley Publishing Co., 1993

[9] P. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, R. L. Allmon, High-Performance Microproessor Design, *IEEE Journal of Solid-State Circuits*, Vol. 33, No. 5, pp. 676-686, May 1998.

[10] D. R. Gonzales, Micro-RISC Architecture for the Wireless Market, *IEEE Micro*, Vol. 19, No. 4, pp. 30-37, July-August 1999.

[11] R. Y. Chen, N. Vijaykrishma, M. J. Irwin, “Clock Power Issues in System-on-a-Chip Designs,” in *Proceedings of the IEEE Computer Society Workshop on VLSI'99*, 1999, pp. 48.

[12] D. Duarte, V. Narayanan, and M. J. Irwin, “Impact of Technology Scaling in the Clock

System Power," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2002, pp. 59.

[13] T. Mudge, "Power: A First-Class Architectural Design Constraint," *IEEE Computer*, *Vol. 34, No. 4*,pp. 52-58, April 2001.

[14] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner, "Theoretical and practical limits of dynamic voltage scaling," in *Proceedings of the 41st annual Design Automation Conference*, pp. 868-873, 2004.

[15] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple, "AMULET1: an asynchronous ARM microprocessor," *IEEE Trans. Computers*, Vol. 46, April 1997, pp. 385-398.

[16] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, N. C. Paver, "AMULET2e: an asynchronous embedded controller," in *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997, pp. 290 – 299.

[17] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver, "AMULET2e:An Asynchronous Embedded Controller," *Proceedings of the IEEE*, Vol. 87, Issue 2, Feb. 1999, pp. 243 – 256.

[18] S. B. Furber, D. A. Edwards and J. D. Garside, "AMULET3: a 100 MIPS Asynchronous Embedded Processor", in *Proceedings of the International Conference on Computer Design*, 2000, pp. 329-334.

[19] H. v. Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann, An asynchronous low-power 80c51 microcontroller, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 96–107, 1998.

[20] A. Bink and R. York, "ARM996HS: The First Licensable, Clockless 32-bit, Processor Core," *IEEE Micro*, Vol. 27, Issue 2, pp. 58-68, March-April, 2007.

[21] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, Ma. Ratta, S. Kottapalli, "A 45nm 8-Core Enterprise Xeon® Processor," in *IEEE International Solid-State Circuits Conference, ISSCC*, 2009.

[22] A. Silberschatz, P. Galvin,G. Gagne, *Operating Systems Concepts*, 7th ed. John Wiley & Sons, 2005.

[23] B. Jacob and T. Mudge, "Virtual Memory: Issues of Implementation," *IEEE Computer*, Vol. 31, NO. 6, June 1998, pp.33-43.

[24] R. Case and A. Padegs. *Architecture of the IBM System/370*, McGraw-Hill Book Company, New York , 1982.

[25] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE MICRO*, July 1998, pp. 60-75.

[26] D. W. Clark and J. S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," ACM Trans. on Computer Systems. Vol. 3, 1985, pp. 31-62.

[27] Michael J. Flynn, *Computer Architecture – Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, Boston, 1995.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed, Morgan Kaufman, 2006.

[29] J. P. Shen, M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill Professional, 2004.

[30] Advanced Micro Devices, Inc., *AMD64 Technology- AMD64 Architecture Programmer's Manual Volume 2: system Programming*, AMD, September 2006.

[31] Intel Corp.,*Intel®64 and IA-32 Architectures: Software Developer's Manual Vol. 3A: System Programming Guide Part 1*, Intel Corp., March 2009.

[32] Intel Corp.,*TLBs, Paging-Structure Caches, and Their Invalidation: Application Note*, Intel Corp., 2008.

[33] M. Talluri, M. D. Hill, Y. A. Khalidi, "A new page table for 64-bit address spaces," *ACM SIGOPS Operating Systems Review*, Vol. 29, Issue 5, Dec. 1995, pp. 184 – 200.

[34] MIPS Technologies, Inc., *MIPS R10000 Microprocessor User's Manual*, Ver. 2.0, MIPS Technologies, Inc., 1996.

[35] SPARC International Inc., *The SPARC Architecture Manual Version 8*, SPARC International Inc., 1992.

[36] D.L. Weaver and T. Germond, *The SPARC Architecture Manual Version 9*, SPARC International Inc., 2000.

[37] Todd Austin, *SimpleScalar LLC*, http://www.simplescalar.com/ (2009/06)

[38] Advanced Micro Devices, Inc., *Software Optimization Guide for AMD Athlon™ 64 and AMD OpteronTM Processors*, AMD, September 2003.

[39] J. M. Tendler, J.S. Dodson, J.S. Fields Jr, H. Le, B. Sinharoy, *POWER4 System Microarchitecture*, IBM J. RES. & DEV. 46, 2002.

[40] Intel® Corp., *Pentium® Pro Family Developer's Manual Vol. 3 – Operating System Writer's Guide*, Intel Corp., Dec. 1995.

[41] Intel Corp., *Intel® Itanium® Architecture Software Developer's Manual Vol. 2: System Architecture Rev. 2.2*, Intel Corp., Jan. 2006.

[42] Advanced Micro Devices, Inc., *AMD64 Architecture Programmer's Manual Vol. 2: System Programming Rev. 3.14*, Advanced Micro Devices, Inc., Sep. 2007.

[43] M. Talluri, *Use of Superpages and Subblocking in the Address Translation Hierarchy* , Ph.D. thesis, Dep. Of CS, University of Wisconsin at Madison, 1995.

[44] M. Talluri and M. Hill. "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *Proceedings of the Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp.171–182.

[45] M. Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. "Tradeoffs in Supporting Two Page Sizes," In *Proceedings of the 19th Annual Int'l Symp. on Computer Architecture*, May 1992, pp.415-424.

[46] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and Memory Overhead Using Online Superpage Promotion," in *Proceedings of the 22nd Annual Int'l Symp. on Computer Architecture*, 1995, pp.176-187.

[47] Jung-Hoon Lee, Jang-Soo Lee, and Shin-Dug Kim, "A dynamic TLB management structure to support different page sizes," *Proceedings of the Second IEEE Asia-Pacific Conference on ASICs*, 2000, pp. 299-302.

[48] Jung-Hoon Lee, Jang-Soo Lee, She-Woong Jeong, and Shin-Dug Kim, "A Banked-Promotion TLB For High Performance and Low Power," *Proceedings of the 2001 International Conference on Computer Design*, 2001, pp. 118-123.

[49] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach Using Superpages Backed by Shadow Memory," *Proceedings of the 25th Annual International Symposium On Computer Architecture*, 1998, pp. 204-213.

[50] Zhen Fang, Lixin Zhang, John B. Carter, Wilson C. Hsieh, and Sally A. Mckee, "Reevaluating Online Superpage Promotion with Hardware Support," in *Proceedings of the 7th Int'l Symp. on High-Performance Computer Architecture*, 2001, pp.63-72.

[51] C. H. Park, J. Chung, B. H. Seong, Y. Roh, and D. Park, "Boosting Superpage Utilization with the Shadow Memory and the Partial-Subblock TLB," in *Proceedings of the 14th international conference on Supercomputing*, 2000, pp. 187-195.

[52] David Channon and David Koch, "Performance Analysis of Re-configurable Partitioned TLBs," *Proceedings of the 30th Hawaii International Conference on System Sciences*, 1995, Vol. 5, pp.168-177.

[53] T. Juan, T. Lang, J. J. Navarro, "Reducing TLB power requirements," in *International Symposium on Low Power Electronics and Design*, 1997, pp. 196-201.

[54] Y. Lee, T. Lee, S. An, and Y. Lee, "Indirectly-compared cache tag memory using a share tag in a TLB," *IEE Electronics Letters*, Vol. 33, No21, 1997, pp. 1764-1766.

[55]Y. Lee, T. Lee, S. An, and Y. Lee, "Shared tag for MMU and cache memory," in *International Semiconductor Conference*, CAS'97, Vol. 1, Oct. 1997, pp. 77-80.

[56] A. Saulsbury, F. Dahlgren, and P. Stenstrom, "Recency-Based TLB Preloading," *Proceedings of the 27th International Symposium on Computer Architecture*, 2000, pp.117-127.

[57] G. B. Kandiraju and A. Sivasubramaniam, "Going Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[58] W. A. Clark, "Macromodular computer systems," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS Joint Computer Conferences)*, April 18-20, 1967, pp. 335-336.

[59] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the sixth MIT conference on Advanced research in VLSI*, 1990, pp. 263-178.

[60] A. J. Martin, *Programming in VLSI: from communicating processes to delay-insensitive circuits*, University Of Texas At Austin Year Of Programming Series, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[61] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proceedings of*

an International Symposium on the Theory of Switching, The Annuals of Computation Laboratory of Harvard University, Vol. 29, Part I, Harvard University Press, Cambridge, 1959, pp. 204-243.

[62] J. B. Dennis and S. S. Patil, "Speed Independent Asynchronous Circuits," in *Proceedings of the Fourth Hawaii International Conference on System Sciences*, 1971, pp. 55-58.

[63] D. Misunas, "Petri nets and Speed Independent Design," *Communications of the ACM*, Vol. 16, Issue 8, August 1973, pp.474-481.

[64] C. L. Seitz, *System Timing Introduction to VLSI Systems Ch.7*, Addison-Wesley Pub. Co., 1980.

[65] Y. T. Chang, M. C. Huang, W. M. Cheng, H. Y. Tsai, C. J. Chen, F. C. Cheng, "Self-Timed Torus Network with 1-of-5 Encoding," in *Proceedings of the 13th IEEE International Symposium on Consumer Electronics*, Kyoto, Japan, May 25-28, 2009.

[66] J. Sparsø and S. Furber, *Principles of asynchronous circuit design – a systems prospective*, Kluwer Academic Publishers, London, 2001, pp. 11-25.

[67] Chris J. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, Inc.,2003.

[68] D. Muller and W. Bartky, "A theory of asynchronous circuits," in *Proceedings of an International Symposium on the Theory of Switching*, April 1959, pp. 204-243.

[69] J. Gunawardena, "A generalized event structure for the Muller unfolding of a safe net," in *Proceedings of the 4th International Conference on Concurrency Theory*, June, 1993, pp. 278-292.

[70] C. J. Chen, W. M. Cheng, H. Y. Tsai, and J. C. Wu, "A Quasi-Delay-Insensitive Microprocessor Core Implementation for Microcontrollers," *Journal of Information Science and Engineering*, Vol. 25, No. 2, March 2009, pp. 543-557.

[71] I.E. Sutherland, "Micropipelines," Turing Award Lecture, *Communications of the ACM*, Vol.32, Number 6, June 1989, pp 720-738.

[72] E. Brunvand, "The NSR Processor," in *Proceeding of the 26th Hawaii International Conference on System Sciences*, 1993, pp. 428-435.

[73] S. B. Furber, "Computing without Clocks: Micropipelining the ARM Processor," in *Asynchronous Digital Circuit Design, Proceedings of the 1993 VIIth Banff High Order*

Workshop, Springer Verlag, January 1995, pp. 211-262.

[74] C. S. Choy, J. Butas, J. Povazanic, C.F. Chan, "A new control circuit for asynchronous micropipelines," *IEEE Trans. Computers*, Vol. 50, Sep. 2001, pp.992-997.

[75] D. K. Arvind, R.D. Mullins,V. E. F. Rebello, "Micronets: a model for decentralising control in asynchronous processor architectures," in *Second Working Conference on Asynchronous Design Methodologies*, May 1995, pp.190-199.

[76] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takmura, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor", *IEEE Design & Test of Computer*, Summer 1994, pp. 50-63.

[77] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC-2: A 32-bit Asynchronous Microprocessor based on Scalable-Delay-Insensitive Model," in *Proceedings of the International Conference on Computer Design*, Oct. 1997, pp. 288-294.

[78] Alain J. Martin, Mika Nyström, and Catherine G. Wong, "Three Generations of Asynchronous Microprocessors," *IEEE Design & Test of Computers*, Nov.-Dec. 2003, pp. 9-17.

[79] C. J. Chen, C. C. Shiu, and M. S. Wu, "The Design of Asynchronous Processor", in *Proceedings of International Computer Symposium*, 2002.

[80] R. Kol, and R. Ginosar, "Kin: a high performance asynchronous processor architecture," in *Proceedings of the 12th International Conference on Supercomputing*, July 1998, pp. 433-440.

[81] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven, "Hades—An Asynchronous Superscalar Processor," in *IEE Colloquium on Design and Test of Asynchronous Systems*, 1996.

[82] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar, "The counterflow pipeline processor architecture," *IEEE Design & Test of Computers*, pp. 48-59, Fall 1994.

[83] S. C. Smith, R. F. Demara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "Delay-insensitive gate-level pipelining," *Integration*, *The VLSI Journal*, Vol. 30, No. 2, October 2001, pp. 103-131.

[84] C. E. Molnar, I. W. Jones, W. S. Coates, and J. K. Lexau, "A FIFO ring performance

experiment," in *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997, pp. 279-289.

[85] I. Sutherland and S. Fairbanks, "GasP: A Minimal FIFO Control," in *Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, 11-14 March, 2001, pp. 46-53.

[86] D. M. Chapiro, *Globally-asynchronous locally-synchronous systems*, Ph.D. Thesis Stanford Univ., CA. Dept. of Computer Science, 1984.

[87] W. J. Dally and C. L. Seitz, "The torus routing chip", *Distributed Computing*, Vol. 1, pp. 187-196, 1986.

[88] Lasse Natvig, "High-level Architectural Simulation of the Torus Routing Chip," in *IEEE International Verilog HDL Conference*, 1997, pp. 48-55.

[89] K. V. Berkel, J. Kessels, M. Roncken, R. Saeijs, P. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proceedings of the European Conference on Design Automation. EDAC*, 25-28 Feb. 1991, pp. 384 – 389.

[90] http://www.handshakesolutions.com/ (2009/06)

[91] A. Bardsley, *Implementing Balsa Handshake Circuits*, Ph.D. thesis, Dep. of Computer Science, Univ. of Manchester, 2000.

[92] A. Bardsley, D. A. Edwards, *The Balsa Asynchronous Circuit Synthesis System*, Dep. of Computer Science, Univ. of Manchester, 2000.

[93] A. B. Doug Edwards, *Balsa: A Turorial Guide version 3.4*, Dep. of Computer Science, Univ. of Manchester, 2004.

[94] C. J. Chen, W. M. Cheng, R. F. Tsai, H. Y. Tsai, T. C. Wang, "A Pipelined Asynchronous 8051 Soft-core Implemented with Balsa," in *9th IEEE Asia Pacific Conference on Circuits and Systems*, Macao, China, Nov. 30 - Dec. 3, 2008, pp. 976-979.

[95] C. J. Chen, W. M. Cheng, T. C. Wang, Y. T. Chang, H. Y. Tsai, "Instruction Decoder Implemented with Balsa for an Asynchronous Pipelined 8051 compatible Microcontroller," in *International Computer Symposium*, Taipei, Taiwan, 13-15 Nov., 2008.

[96] Q. Zhang and G. Theodoropoulos, "Modelling SAMIPS: a synthesisable asynchronous MIPS processor," in *Proceedings of 37th Annual Simulation Symposium*, 18-22 Apr, 2004, pp. 205-212.

[97] C. J. Chen, W. M. Cheng, H. W. Lo, Y. T. Chang, H. Y. Tsai, I. H. Hsieh, F. C. Cheng, "An Asynchronous MP3 Decoder Implemented with Balsa," in *IEEE Regional Symposium on Micro and Nano Electronics*, 2009. (to appear)

[98] A. J. Martin, *Synthesis of Asynchronous VLSI Circuits*, Technical Report [Caltech-CS-TR-93-28], California Institute of Technology, 1991.

[99] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings and T. K. Lee, "The Design of an Asynchronous MIPS R3000 Microprocessor," in *Proceedings of the 17th Conference on Advanced Research in VLSI*, 1997, pp. 164.181.

[100] http://4c.cse.ttu.edu.tw/snipsnap/space/SoCAD (2009/06)

[101] F. C. Cheng, C. R. Wang, "Specification and design of a quasi-delay-insensitive Java card microprocessor," in *Proceedings oh 13th International Conference on VLSI Design*, 3-7 Jan. 2000. pp. 356-361.

[102] http://www.async.caltech.edu/mips.html (2009/06)

[103] C.A.R. Hoare, "Communicating Squential Processes," *Communications of the ACM*, Vol. 21, No. 8, 1978, pp. 666-677.

[104] C. Myers and A. Martin, "*The design of an asynchronous memory management unit*," Technical Report [Caltech-CS-TR-92-25], California Institute of Technology, 1992.

[105] F. Weigel, *An Asynchronous ARM Compatible Memory Management Unit Design and Implementation*, M.S. Thesis, Dep. of Computer Science, Univ. of Manchester, 2002.

[106] http://intranet.cs.man.ac.uk/apt/projects/tools/balsa/ (2009/06)

[107] http://code.google.com/intl/zh-TW/android/ (2009/06)

[108] http://www.analog.com/en/embedded-processing-dsp/blackfin/adsp-bf561/processors/product.html (2009/06)

[109] http://www.tilera.com/products/processors.php (2009/06)

[110] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 11, Nov. 2001, pp. 1338-1354.

[111] http://www.coware.com/products/processordesigner.php (2009/06)