Technical Report

UCAM-CL-TR-588
ISSN 1476-2986

Number 588

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

MulTEP: A MultiThreaded Embedded Processor

Panit Watcharawitch

May 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted November 2003 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Newnham College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Summary

Conventional embedded microprocessors have traditionally followed the footsteps of high-end processor design to achieve high performance. Their underlying architectures prioritise tasks by time-critical interrupts and rely on software to perform scheduling tasks. Single threaded execution relies on instruction-based probabilistic techniques, such as speculative execution and branch prediction. These techniques might be unsuitable for embedded systems where real-time performance guarantees need to be met [1, 2].

Multithreading appears to be a feasible solution with potential for embedded processors [3]. The multithreaded model benefits from sequential characteristic of control-flow and concurrency characteristic of data-flow [4]. Thread-level parallelism has a potential to overcome the limitations of insufficient instruction-level parallelism to hide the increasing memory latencies. Earlier empirical studies on multithreaded processors [5, 4, 6, 7, 3] demonstrated that exploiting thread-level concurrency not only offers latency tolerance, but also provides predictable performance gain.

A *MulTithreaded Embedded Processor* (MulTEP) is designed to provide high performance thread-level parallelism, real-time characteristics, a flexible number of threads and low incremental cost per thread for the embedded system. In its architecture, a matching-store synchronisation mechanism [4] allows a thread to wait for multiple data items. A tagged up/down dynamic-priority hardware scheduler [8] is provided for real-time scheduling. Pre-loading, pre-fetching and colour-tagging techniques are implemented to allow context switches without any overhead. The architecture supports four additional multithreading instructions (i.e. `spawn`, `wait`, `switch`, `stop`) for programmers and advance compilers to create programs with low-overhead multithreaded operations.

Experimental results demonstrate that multithreading can be effectively used to improve performance and system utilisation. Latency operations that would otherwise stall the pipeline are hidden by the execution of the other threads. The hardware scheduler provides priority scheduling, which is suitable for real-time embedded applications.

Contents

List of Figures

List of Tables

Part I

Background

Chapter 1

Introduction

If we knew what it was we were doing,
it would not be called research, would it?

Albert Einstein

1.1 Prelude

Embedded processors are increasingly deployed in applications requiring high performance with good real-time characteristics whilst consuming low power. Contemporary high performance embedded architecture often follow the footsteps of high-end processor design and prioritise tasks by time-critical interrupts leaving software to perform scheduling. Parallelism has to be extracted in order to improve the performance at an architectural level. Extracting instruction level parallelism requires extensive speculation which adds complexity and increases power consumption.

Multithreading appears to be a feasible alternative candidate approach. The multithreading model benefits from the sequential characteristic of control-flow and concurrency characteristic of data-flow [4]. Many embedded applications can be written in a threaded manner and therefore multithreading in hardware may extract parallelism without speculation whilst keeping each component of the system quite simple.

The structure of this chapter is as follows: Section 1.2 discusses my motivations to research in the area of embedded design and multithreaded architectures. Section 1.3 states the aims of my research. Section 1.4 presents the structure of the remaining chapters of this dissertation.

1.2 Research Motivation

This research is driven by two key observation:

1. There is a rapid growth in the embedded processor market which shows no signs of abating.
2. It is unclear how to provide this market with high performance processors which are low power, have good real-time properties and are not overly complex to implement.

The following sections address the issues that surround these observations.

1.2.1 Inspiration for Targeting the Embedded Processor Market

The functionality of embedded processors is determined by the processor market. Derived from McClure's survey [9], the trend of volume-shipment of general purpose processors[1] shows that approximately 100 million general purpose processors were shipped in year 2000 with a 10% yearly growth rate (see Figure 1.1).

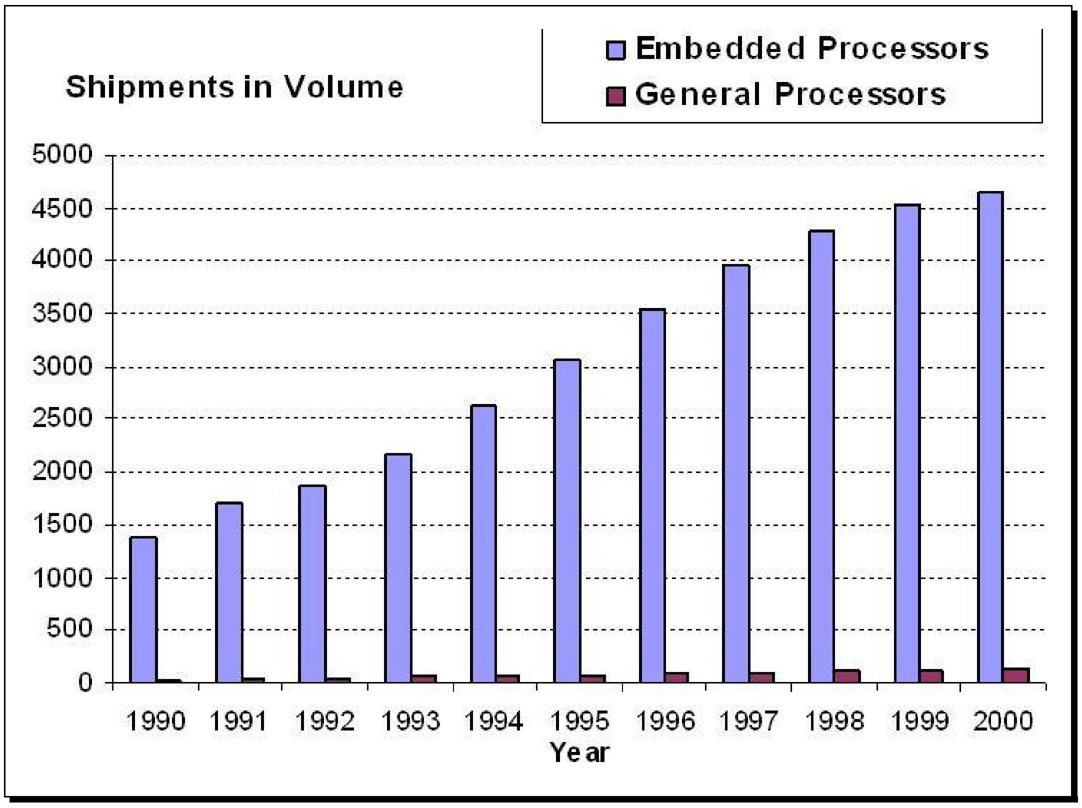


Figure 1.1: Shipment trends of general and embedded processors [9].

Devices ranging from industrial-automated lines to commercial-mobile equipment require embedded processors. Even PCs, which already contain a powerful CPU, also use

[1]General purpose processors from McClure's survey [9] represent microprocessors.

additional embedded processors (e.g. drive controllers and peripheral-interface devices). The shipment of embedded processors is around 4 billion in the year 2000. This is approximately 40 times greater than the shipment of general purpose processors with a potential growth-rate of around 35% per year [9]. This trend reflects that the functional requirements from the market focus on the embedded models. Thus, designing a processor to support embedded system constraints is an interesting research area.

1.2.2 Inspiration for Multithreaded Execution

In a processor design, high computational performance is attained by minimising the execution time (T_{exe}) of a system. The execution time T_{exe} is a combination of the service time of the processor (T_{serv}) and its wasted latency ($T_{latency}$)[2], as shown in Equation 1.1.

$$T_{exe} = T_{service} + T_{latency} \tag{1.1}$$

Many research approaches have been proposed to reduce both the service time and the latency period. Equation 1.2 presents that the service time can be reduced by reducing the time spent per cycle (T_{clk}), reducing the number of instructions (n_{inst}) or increasing the number of *Instructions Per Cycle* (IPC).

$$T_{service} = \frac{T_{clk} \times n_{inst}}{IPC} \tag{1.2}$$

To reduce T_{clk}, faster implementation technologies have been exploited [10]. Architectural techniques such as deeper pipelines, easy to decode instructions (*Reduced Instruction Set Computers* — RISC [11, 12]) and advanced circuit techniques have been used to shorten critical paths. However, the development in these areas is slowing because of physical limitations in semiconductor fabrication technologies [13, 14]. Furthermore, 50 years of research in low level architecture and circuit techniques has left little scope to make improvements here.

To reduce n_{inst}, ideas from *Complex Instruction Set Computers* (CISC) [15, 16, 17, 18, 19] and *Reconfigurable Instruction Set Processors* (RISP) [20] could be exploited. The disadvantage of these processors is the increase in T_{clk} due to their instruction complexity which requires complex implementations.

Single pipelines allow us to improve the IPC up to one instructions per clock cycle. Multiple pipelines are used by multiple instruction issue architectures to push IPC to multiple instructions per clock cycle. Example multi-issue architectures include: superscalar, *Very Long Instruction Word* (VLIW) [21] or *Explicitly Parallel Instruction Computer* (EPIC) [22]. The superscalar approach requires aggressive instruction scheduling at the

[2] $T_{latency}$ occurs when a processor has to wait for memory access or some certain operations (e.g. PC redirection from a branch instruction) during the execution.

hardware level which results in complex implementations which grow alarmingly as the number of pipelines increases. VLIW and EPIC push some of the instruction scheduling burden into the compiler.

Figure 1.2 illustrates an improvement in processor performance of 35-55% a year due to alterations in the architecture[3]. In contrast, memory access latency, $T_{latency}$, only improves around 7% per year. This memory access latency is increasingly a crucial factor which limits performance. Thus, the trend presents a growing gap between processor cycle times and memory access times at approximately 50% per year [23].

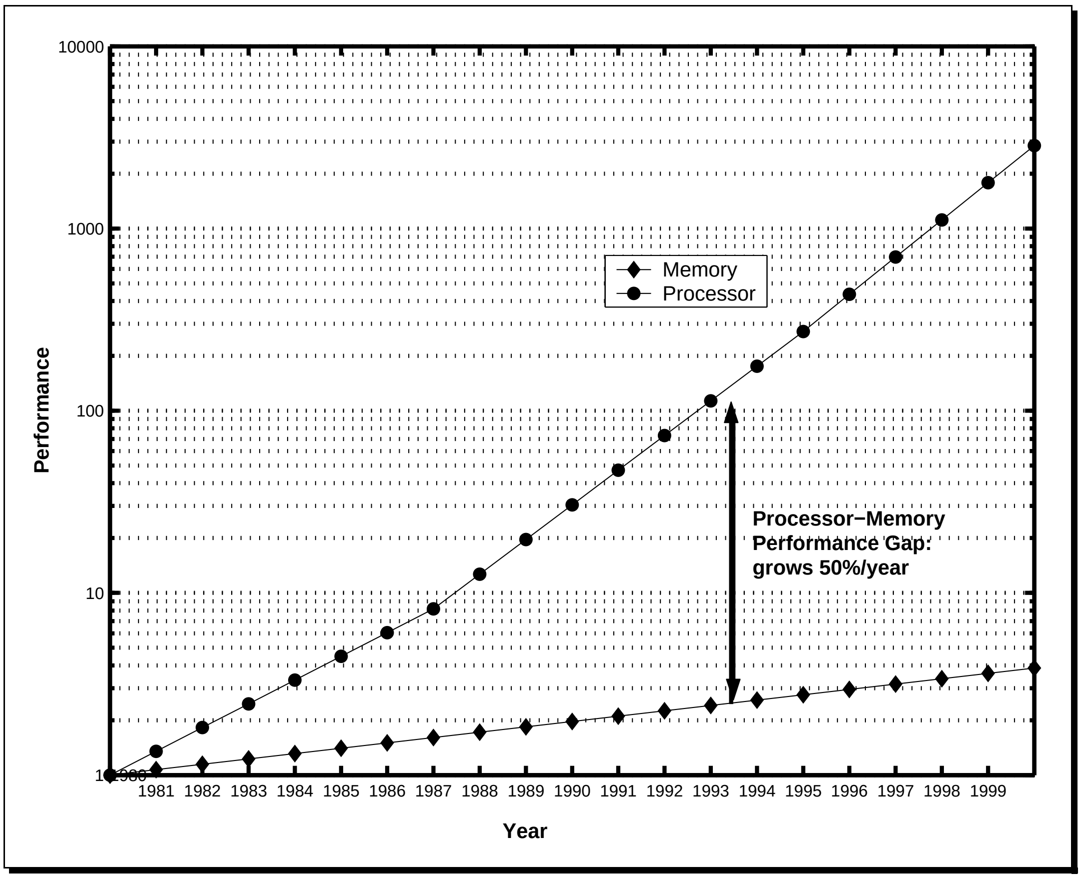


Figure 1.2: The increasing trend of processor-memory performance gap [23].

$T_{latency}$ is reduced by the use of caches [24], vector operations [25] and speculative prefetching/pre-loading mechanisms [26]. Alternatively, latency ($T_{latency}$) can be hidden by concurrent execution. Today's high-end processors execute many instructions per cycle using *Instruction Level Parallelism* (ILP). Instruction level parallelism is exploited using statistical speculations, such as dynamic scheduling and branch prediction. Speculative execution to exploit instruction level parallelism requires additional dedicated hardware units, which consume power, and extra operations to eliminate mispredicted tasks.

Parallelism beyond instruction level parallelism in the processor, such as process level parallelism and thread level parallelism, have been further explored. In 1992, DEC architects believed that up to ten times of performance improvement could be gained by

[3]The improvement is based on the processor performance since 1980.

the contribution of *Thread Level Parallelism* (TLP) [27] with in 20 years. A number of investigations [28, 29, 30] indicate that thread level parallelism, i.e. multithreading, at the hardware level has great potential to efficiently hide the memory and I/O access latencies.

Though thread level parallelism has a potential to improve processing performance, a framework to support thread level parallelism at the hardware level has not yet been standardised. As a result, thread level parallelism has not been utilised to the same degree as a pipelining or superscalar techniques that effectively provide instruction level parallelism. There is plenty of room for research in this area.

1.3 Research Aims

My research aims to develop a high-performance multithreaded processor architecture for embedded environments. A number of design challenges to support such a requirement are first identified (§1.3.1), followed by main contributions that this research intends to offer to the architectural community (§1.3.2).

1.3.1 Challenges

To design a multithreaded embedded architecture, a number of challenges arose:

1. Exploit thread level parallelism to improve performance by hiding operating latencies (e.g. memory access, branch delay slots).
2. Support a flexible number of threads with a minimal incremental cost per thread.
3. Be able to schedule the threads in order to meet real-time constraints.
4. Avoid speculative execution mechanisms that degrade real-time performance, consume extra power executing instructions whose results are never used, and require extra memory to store and analyse statistical data.

1.3.2 Contributions

This section outlines the contributions of this thesis in areas central to the development of a *MultiThreaded Embedded Processor* (MulTEP):

1. An effective framework to support multithreading in hardware:
    - Minimal mechanisms to progress a thread through its life cycle.
    - Minimal mechanisms to provide low overhead context switching.
    - Minimal mechanisms to schedule threads that have a dynamic characteristic (i.e. a change of deadline/priority).
2. A solution to provide a minimal incremental cost per thread.

3. A model to support a flexible number of threads in hardware.
4. A scheduling mechanism that offers real-time response.
5. A strategy to exploit thread level parallelism without resorting to speculative execution.

1.4 Dissertation Outline

The rest of the thesis is structured as follows:

Chapter 2: Architectural Level Parallelism

This chapter reviews the architectural level parallelism in three key areas: the control-flow model, the data-flow model and the memory hierarchy. This chapter critiques related work and determines the main requirements that the design and implementation of a processor architecture must satisfy.

Chapter 3: Multithreaded Processors

In this chapter, key multithreading theories are examined. Design criteria for multithreaded architectures are established. The current state-of-the-art is evaluated.

Chapter 4: Background to Embedded Design

This chapter addresses embedded environment constraints and requirements. Then, current solutions in the embedded design space are surveyed.

Chapter 5: System Overview

In this chapter, three desirable characteristics (*high-performance multithreading*, *real-time operation* and *low-power consumption*) are identified. Key design strategies in the project, their integration and the overall architecture from the programmer's point of view are then described.

Chapter 6: Hardware Architecture

This chapter details the MulTEP hardware architecture, which comprises the *Processor Unit* (PU), the *Load-Store Unit* (LSU), the *Multithreading Service Unit* (MSU) and the *Memory Management Unit* (MMU).

Chapter 7: Software Support

In this chapter, software support tools (i.e. MulTEP assembler, Java-to-native, MulTEP macros and Thread-0 system daemon) for the MulTEP architecture are presented.

Chapter 8: Evaluation and Results

This chapter describes evaluation methodologies and procedures for examining the MulTEP system. Then, evaluation results are presented.

Chapter 9: Conclusions

Conclusions of this research are drawn in this chapter. Areas of possible future research are then suggested.

Chapter 2

Architectural Level Parallelism

The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law

2.1 Introduction

Parallelism has long been employed at an architectural level in order to improve both performance and utilisation efficiency of a processor. In architectural design, there are two predominant architectural models. One is a *control-flow* processor model which executes instructions in sequential order as presented in Figure 2.1(a). The other is a *data-flow* processor model where instructions are executed concurrently in a data dependent ordering as illustrated in Figure 2.1(b).

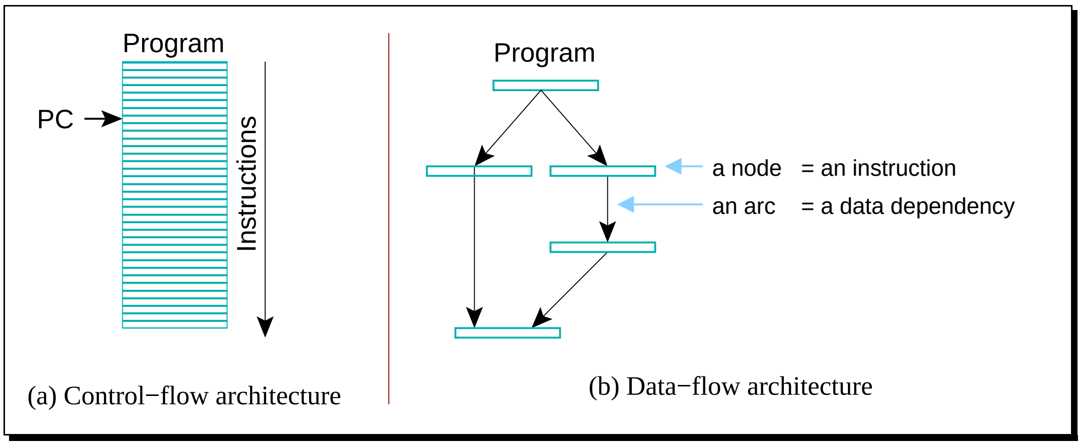

Program
PC
Instructions
(a) Control−flow architecture
Program
a node = an instruction
an arc = a data dependency
(b) Data−flow architecture

Figure 2.1: A control-flow architecture and a data-flow architecture.

This chapter focuses on the design evolution to support parallelism at the architectural level. It first provides an overview of parallelism offered by the control-flow processor model (Section 2.2) and the data-flow processor model (Section 2.3). The memory system to support architectural level parallelism is then investigated (Section 2.4). Finally, a summary of the current architectural level parallel issues are presented (Section 2.5).

2.2 Control-flow Architectures

Based upon Babbage's proposal in 1838 [31], Eckert, Mauchly and von Neumann successfully introduced a control-flow processor model in 1944 [26]. The architecture is a design for processing a list of programming instructions according to their sequence. It uses a *Program Counter* (PC) to indicate the current executed position in the execution sequence as depicted in Figure 2.1(a). The flow of execution is controlled by manipulating the value of PC using instructions such as branches and jumps which may be conditionally executed. Intermediate processed values are typically stored in local registers, an accumulator or a nearby stack [32]. Various mechanisms have been implemented to improve functionality and performance, which has enabled the model to dominate the computer industry for over 50 years.

This section surveys the developments of the control-flow architectural design to support parallelism. The section first focuses on how an instruction encoding influences a control-flow architectural design (§2.2.1). Then, instruction level parallelism in control-flow architectures is explained (§2.2.2), followed by a discussion of adding thread level parallelism to the control-flow models (§2.2.3). Further details of the control-flow architectures in the other aspects can be found in these computer architecture texts [24, 26, 33, 34].

2.2.1 Instruction Encoding

An instruction consists of an opcode and its operands. *Opcode* is shorthand for operation code; it explicitly presents the controls of execution or implicitly represents an index to a set of microcodes that control the different parts of the processing element. *Operands* are either an immediate value, a displacement, an index into a register set, an index to a memory address or an additional operating function. CISC and RISC are two extreme foundational *Instruction Set Architectures* (ISA). The remainder of this section analyses how the two techniques deal with parallelism.

Complex Instruction Set Computer (CISC)

CISC philosophy compresses an instruction into its most compact form as illustrated in Figure 2.2(a). Its encoding technique causes instructions to be vary in length (e.g. from 4 up to 56 bytes in VAX [17]). The compact form offers a set of instructions from the simple (e.g. "add") to the complex (e.g. "evaluate polynomial"). The complex instructions in a compact form enable a program to be compiled into a condensed binary object. The condensed object reduces memory bandwidth requirements and memory space.

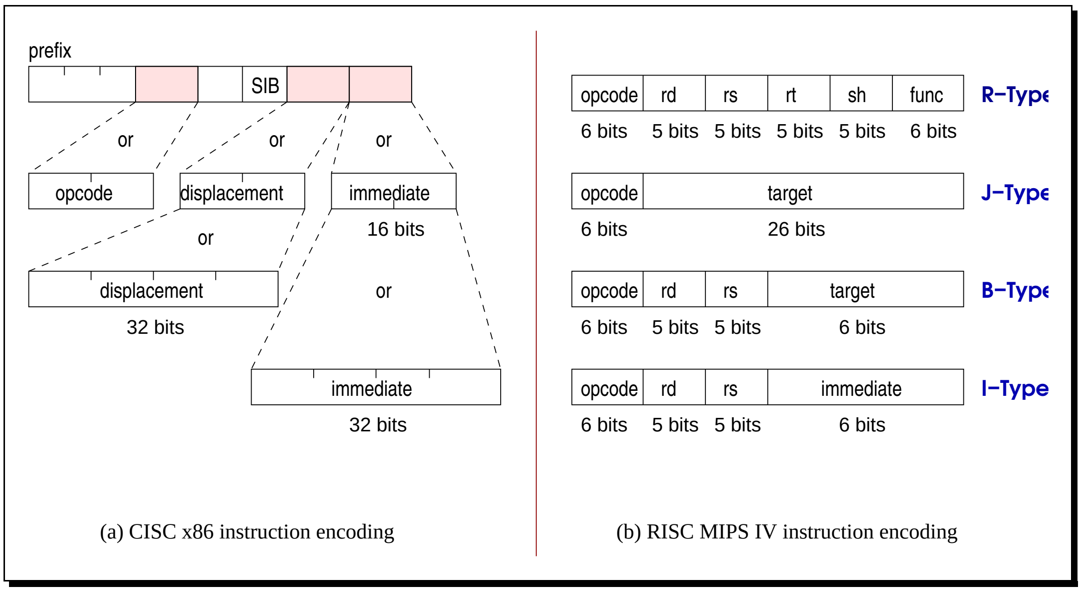


Figure 2.2: Instruction encoding of representative CISC and RISC.

Disadvantages of CISC stem from implementation complexities. The architecture contains much temporal and area overhead in order to decode complicated instructions. Its intermediate computation stages and values are not easily separated. Hence, the architecture has a significant burden when it needs to switch its execution context. In addition, around 20% of instructions are used frequently whilst a number of circuits provided for the other 80% are not effectively utilised. Hardware complexity results in large area and high power dissipation.

This architecture has successfully dominated commodity microprocessors (e.g. Intel x86 [15], Motorola 68k [16]) and was common for minicomputers (VAX [17], PDP-11 [18]). Modern CISCs often decode instructions into one or more RISC-like operations (e.g. Intel's μops [15], AMD's r-ops [19]).

Reduced Instruction Set Computer (RISC)

RISC philosophy is based on the goal of "making the common case fast". Its architecture provides rapid execution by using a small set of simplified fixed-length instructions as presented in Figure 2.2(b). This encoding allows instructions to be aligned on word boundaries to be fetched in a single cycle. Opcodes and operands are easily segmented and quickly decoded due to their fixed positions. This feature allows the architecture to have a short and simpler pipeline [11].

RISC code density is not as good as CISC. Complicated operations need to be constructed from a sequence of instructions. Fixed length instructions limit the number and size of operands so that multiple memory operands become impractical. RISC has to eliminate complex addressing modes and, instead, relies on *register*-based addressing modes and specific *load* and *store* instructions to access memory. Because of this limitation, data needs to be transferred to registers prior to their usage, resulting in a larger number of registers.

Advantages of RISC are its simple architecture that allows a rapid design and implementation cycle; the separation of memory access instructions (i.e. load and store) that helps the compiler to schedule instructions efficiently[1]; the possibility of an independent load/store unit that handles memory access instructions separate from the other computational instructions; and room to add new instructions [12].

Processors based on RISC ISA have been used in the domains of servers (e.g. SPARC [35], MIPS [36], Alpha [27], PowerPC [37], System/370 [25], PA-RISC [38], IA-64 [22]) and embedded applications (e.g. ARM [39], SuperH [40]).

2.2.2 Instruction Level Parallelism

The concept of *Instruction Level Parallelism* (ILP) was first described by Wilkes and Stringer in 1953 [41]:

In some cases it may be possible for two or more micro-operations to take place at the same time.

Early ILP is based on two concepts. The first concept segments operation horizontally into pipeline stages as presented in Figure 2.3(a). The second concept increases parallelism vertically in a superscalar form as illustrated in Figure 2.3(b).

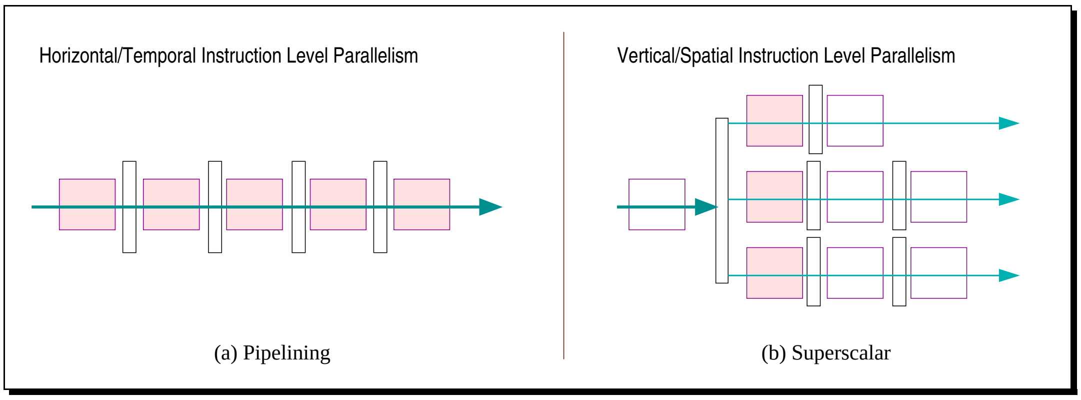

Horizontal/Temporal Instruction Level Parallelism
Vertical/Spatial Instruction Level Parallelism
(a) Pipelining
(b) Superscalar

Figure 2.3: Horizontal and vertical instruction level parallelism.

The following sections review pipelining and superscalar techniques which are used to exploit horizontal and vertical ILP. Then, alternative ILP architectures, a VLIW and an EPIC, are investigated.

[1]Compiler can schedule instructions in the delay slots immediately after a load instruction.

Pipelining

The data path of a processor consists of multiple stages (e.g. instruction fetch, instruction decode, operand fetch, execution, memory access and register write back). *Pipelining* is introduced to improve processor utilisation by dividing the data path into stages in order to reduce cycle per instruction down to 1 [24].

Pipelining uses *latches* to separate intermediate results from each functional block. Thus, a clock cycle is effectively shortened to the amount of time it takes for the slowest stage. The technique automatically exploits temporal instruction level parallelism by overlapping the execution of sequential instructions in various functional units.

Unfortunately, pipelining introduces interdependencies between stages. Thus, the processor requires a *hazard detection* mechanism to deal with the following data, control and structural hazards:

1. *Data hazard*: an instruction requires the result of a previous instruction.
2. *Control hazard*: PC needs to be corrected because of a previous branch instruction.
3. *Structural hazard*: two instructions need access to the same resource.

Once one of the interdependent violations occurs, the hazard detection stalls its relevant functional units until all conflicts are resolved. This technique introduces idle periods in the pipeline, called *bubbles*. In order to reduce bubbles, a number of techniques are used:

To reduce data hazards – *data forwarding* or *bypassing* paths are introduced to make recent intermediate results available before they are written to the register file.

To reduce control hazards – a *branch prediction* mechanism is used. This mechanism reduces branch delays and decreases incorrect instruction fetches. However, it requires large support units such as a *branch history table*, *n*-bit local/global predictors, and a *branch target buffer*.

To reduce structural hazards – more access to shared resources, such as register ports, are introduced in order to avoid resources conflicts.

The disadvantages of pipelining are longer execution latencies for individual instruction and many additional hardware mechanisms, especially for branch prediction. However, the disadvantages are often outweighed by faster clock speeds and higher instruction throughput. As a result, by the early 1990s, all major processors from both the industry and the academia incorporated pipelining techniques into their architectural design [42].

Superscalar

Not all instructions have the same latencies (i.e. integer, floating point, multiply/divide, memory access). To deal with this, multiple subpipelines are introduced in order to simultaneously execute various computations in different functional units. This technique is called *superscalar*, a technique to exploit spatial instruction level parallelism.

Because superscalar needs multiple instructions per cycle, it requires a large window over an instruction stream. In each window, there is no explicit information about dependencies between instructions which must be determined by the hardware. *Dynamic scheduling* is used to extract multiple independent instructions. Where control and data dependencies permit, instructions may even be issued *out-of-order*.

Regrettably, the technique introduces additional structural conflicts on shared resources, especially, when register write-backs from each subpipeline do not occur in order. Thus, a *reorder buffer* and a *register colouring* technique are required for the final stage to keep instructions in order, to remove false dependencies and to undo the changes after the exceptions. The resolution for such structural conflicts require multiple stalls before they can be resolved.

In addition, a number of studies in [43, 44, 45] report that only few instructions (typically 2 to 4) can be executed in parallel in one basic block. It is possible to execute 5 to 9 instructions in parallel, but only when impractical design choices are made: an infinite table for branch prediction, a perfect register alias analysis and a large instruction issuing window [46]. The lack of instruction level parallelism limits the number of functional units that can be utilised.

Speculative execution is proposed to provide sufficient instructions even before their control dependencies are resolved. This technique strives to issue an instruction to each subpipeline every cycle. It requires a *register renaming* mechanism to allocate a virtual register space to each speculative instruction. This mechanism requires a *register alias table* to map processor registers to a much larger set of temporary internal registers, and a *scoreboard*[2] to keep track of register usage. The result is higher utilisation and statistically better performance.

Nevertheless, mis-speculations need to be discarded. All effects of the speculatively executed instructions must be disposed. This wastes both energy and execution time on the mis-speculated path. In terms of hardware cost, the number of functional units goes up linearly with the required degree of parallelism. The complexity of interconnection network between functional units grows exponentially, i.e. $O(n^2)$, with the degree of parallelism (n) [47]. This results in large complex hardware with very significant wiring complexity which pushes up power consumption and forces a lower clock rate.

VLIW and EPIC

A *Very Long Instruction Word* (VLIW) [21] is introduced to exploit instruction parallelism vertically. The architecture gains benefit from a compiler which statically allocates instructions to each subpipeline. Therefore, the architecture issues only one large fixed-length instruction to be processed on multiple independent subpipelines simultaneously. The operation occurs without issue-check resulting in a high clock speed.

[2]Scoreboard bits on registers are similar to presence bits used in the static data-flow model (see §2.3.1).

However, because the utilisation relies heavily on the compiler, a compiler is required to be specifically built for every possible implementation of each architecture in order to distribute suitable instructions to their supported subpipelines. An interrupt/exception in one functional unit causes a detrimental delay in all following functional units because the later static issuing process need to be stalled. The resulting code density is quite poor because a number of `NOP` (no-operation) instructions need to be issued when there is insufficient instruction-level parallelism from the software. Also, a single flow of control still lacks adequate independent instructions for this architecture [46].

To deal with VLIW's poor code density, Intel introduced an evolution of VLIW called *Explicitly Parallel Instruction Computer* (EPIC), aka IA-64 [22]. This architecture uses a *template* to determine an appropriate instruction bundle and a *stop bit* as a barrier between independent groups resulting in a better code density. Therefore, it offers better code compression. Its compiler supports different processors. Nevertheless, it still requires a hardware interlock mechanism to stall the pipeline to resolve structural hazards and heavily relies on state-of-the-art compiler technology in order to deliver high performance.

2.2.3 Thread Level Parallelism

This section reviews the support of thread level parallelism in a traditional control-flow single processor and multiprocessors.

Single processor

A single processor sequentially executes instructions and benefits from efficient use of local storages. Disrupting the process requires the current execution context to be saved in order to be restored later.

To support thread level parallelism in such a processor, interrupts are used to disrupt the flow of control [48]. The context switch is then conducted by the software scheduler. The process includes context save & restore, pipeline flush & refill and cache behaviour manipulation. This context-switching overhead is large so that the technique is only appropriate when the context switch is really necessary and occurs infrequently.

Multiprocessors

Parallel execution of threads may be achieved using multiple control-flow processors. The multiprocessor architecture can be designed in a form of *Multiple Instruction streams, Multiple Data streams* (MIMD) [49]. There are two categories of MIMD architectures. The first one is a *shared-memory* MIMD where all processors have uniform access to a global main memory and require a hardware mechanism to maintain cache coherency. The second one is a *distributed-memory* MIMD where each processor has its own local memory. Data communication on the distributed-memory machine is effectively performed by *message passing* in a form of token to another address. Coarse-grained TLP is achieved when each processor works on a specific thread and communicates when necessary.

Nevertheless, the cost and complexity of the network for communication are very high. With the current compiler technology, it is difficult to extract parallelism from single thread group and still requires programmers to write parallel codes. Thus, multiprocessors are often designed for high computational performance in applications such as scientific, visualisation, financial model and database server.

2.3 Data-flow Architectures

In the early 1970s, Dennis [50, 51] derived the fundamental principles of a data-flow model from a data-flow graph. For this architecture, a program is explicitly represented as a directed graph where *nodes* represent instructions and *arcs* represent data dependencies. The operation in each node is executed, i.e. *fired*, when a *matching-store* mechanism [52] indicates that all inputs are present, and its outcome is propagated to one or more nodes in the form of a *token*. This model naturally offers instruction level parallelism where data dependencies impose order of execution. The architecture eliminates the requirement of a program counter [53, 54]. It requires a coordination from a hardware scheduler to issue independent instructions as appropriate.

The remainder of this section presents three paradigms of the data-flow model – a static data-flow model (§2.3.1), a dynamic data-flow model (§2.3.2) and an explicit token-store data-flow model (§2.3.3).

2.3.1 Static Data Flow

In a static data-flow model (e.g. TI's DDP [55], DDM1[56]), all nodes are pre-specified and at most one *token* (datum) per arc is allowed at a time. A node is enabled, i.e. updated, as soon as a token is present on its arc. For instance, a node is active when a predecessor count reaches zero in the TI's DDP [55]. The system uses a backward signal to inform previous nodes when its result has already been obtained. However, its static style restricts an implementation of shared function. Mutual exclusion is needed to be enforced when issuing operations and needs an inefficient replicating functional units as its solution.

2.3.2 Dynamic Data Flow

The dynamic data-flow model [57, 58] is more flexible than the static model. Additional nodes may be generated at run-time and allow multiple coloured tokens to be stored on one arc. The *colour* consists of the address of the instruction and the computational context identifier in which that value is used.

Each invocation of a function gives a unique colour in order to represent functions recursively. Only tokens with the same colour are allowed to be matched in dyadic operations.

The problems with this model are: the colour-matching mechanism is expensive and temporally unpredictable due to its storage bounds; fan-out is uncontrollable and may result in matching store overflow when too many concurrent parts need to be initiated [4].

2.3.3 Explicit Token-store Data Flow

An explicit token-store data-flow model [59, 60, 61] allocates a separate memory frame per functional invocation called an *Activation Frame* (AF) [59]. This alleviates the matching bottlenecks in the dynamic model by explicitly using the address token-store (computed by the composition of frame address and operand offset) for matching and firing operations.

The firing operation is driven by two different mechanisms. The first mechanism is *data-driven* where operations may be executed when input data is available. Unconstrained parallelism can place a large burden on the matching-store mechanism. The matching mechanism unavoidably ends up being the bottleneck.

The second mechanism is a *demand-driven* mechanism. The demand-driven scheme only fires operations when their results are requested. The disadvantage of this data pull scheme is that an additional operation-lookahead circuit is required [62].

The explicit token-store architecture naturally exploits ILP asynchronously because all active instructions may be executed simultaneously. It exposes maximum instruction level concurrency and allocates option for execution without any need for caches (e.g. tolerates memory latency). However, it requires data synchronisation on every event forcing data to go via a data-matching mechanism which may act as an access bottleneck; the matching scheme may be extended to gain control-flow's advantage by allowing intermediate results to be held in registers which reduces the number of accesses to the token-store matching mechanism (e.g. in EM-4 [60]).

2.4 Memory System

As parallelism increases, so do the demands on the memory system for both programs and data. This section analyses the memory hierarchy (§2.4.1), virtual space (§2.4.2) and protection mechanisms (§2.4.3). The analysis focuses on their features to support architectural level parallelism. Details of further enhancements for multithreading (i.e. thread-level parallelism) will be mentioned in the next chapter.

2.4.1 Memory Hierarchy

Due to an increase in the performance gap between a processor and its main memory mentioned in the previous chapter, several levels of fast *caches* are introduced in order to store some recently-used memory areas to alleviate memory latency problems. However, adding more units along the access path lengthens the main memory's access latency.

The caches are placed in such an order that the fastest, smallest one lies closest to the processor and the other slower, larger caches are located farther away. Valid data can be obtained immediately from caches if their identity fields are matched, i.e. *hit*. Otherwise, if there is a data *miss*, a suitable block of cache needs to be reloaded from a lower level thereby suffering a large *miss*-penalty.

Based on the Harvard architecture [24], the first level cache (L1) has been divided into an instruction cache and a data cache with completely separate data and address buses. Therefore, L1 I-cache can be single-ported, read-only and wide access while L1 D-cache is multiported, read & write. This solution widens the memory bottleneck which results in a higher access utilisation.

When designing a memory organisation, *data locality* and *access latency* are two factors that need to be investigated. This section first reviews methods to provide data locality along with comments about their benefits and possible enhancements for parallelism. Then, the section focuses on *access latency* and how it influences architectural design.

Data locality

The locality of the data has a strong impact on processor performance and power utilisation. It is common practice to exploit the following *principle of locality* [24]:

1. *Spatial locality*: addresses nearby the previous access tend to be referenced in the near future.
2. *Temporal locality*: recently accessed addresses are likely to be accessed again soon.

To help exploit spatial locality, cache fetches data/instructions in a *block*, or a *cache line*. Each cache line contains the requested data/instruction and its neighbours. Figure 2.4 presents an address interpretation for obtaining data from a cache. A *tag* field in the address is used as a *block identification* where an *index* points to. A number of bits in an offset field is an important factor that determines its space.

In order to support temporal locality, a *block replacement* policy incorporates time into its decision. Thus, a close-by cache tends to hold the most recently-used data. However, though a *Least-Recently Used* (LRU) replacement policy is the most natural method, its implementation is inefficient. Instead, a random replacement policy may be used which proves to be efficient. Without adding much extra hardware, random replacement can be further improved by combining it with a *Not Last Used* (NLU) policy.

A *block placement* technique is important. The simple *direct-mapped* cache is cheap in terms of circuit and power, but provides low-hit ratio. At the other extreme, a high-hit ratio *fully associative* cache unavoidably introduces high complexity in the memory system. Thus, in general, a medium n-set associative cache turns out to be preferable and proves to be sufficient with a value of n around 2 to 8 [24].

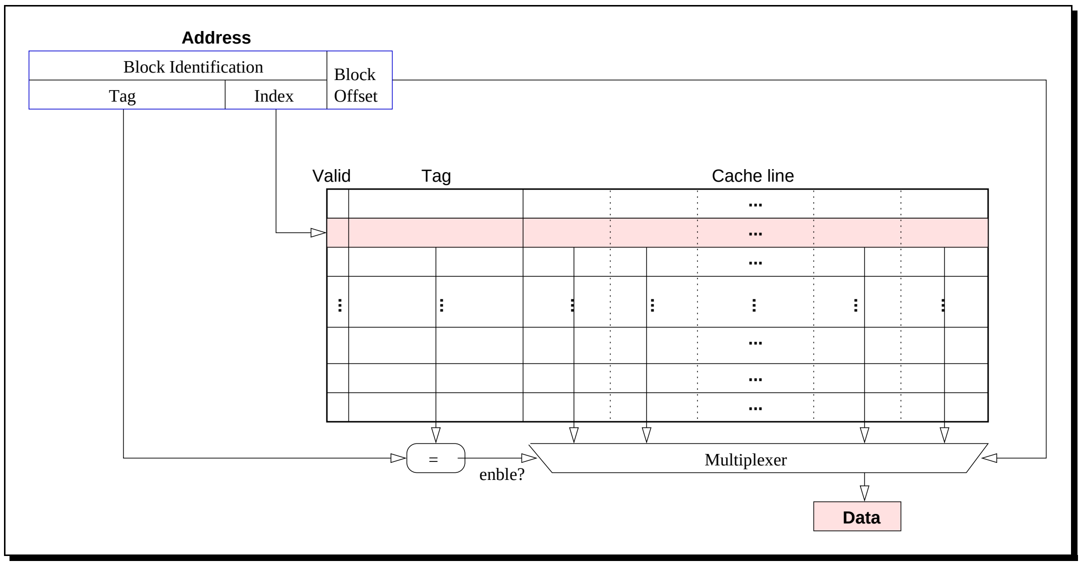


Figure 2.4: An address interpretation for obtaining data from a cache.

Access Latency

A cache access time is a combination of parameters as presented in Equation 2.1.

$$A = (1-p)C + pM \tag{2.1}$$

where	A	is an access time.
	C	is the cost of a hit.
	p	is the proportion of misses.
	M	is the miss penalty.

Parameter A and C are mostly influenced by a physical level design and the structure of the memory hierarchy. Parameters p and M are affected by architectural level design. The miss rate p is reduced by increasing the block size to exploit spacial locality, increasing the cache associativity, and using a suitable replacement policy that works well with the characteristics of the benchmarks. The miss penalty M depends on how cache writes are handled [26]. There are two approaches to handling writes:

1. *Write-through* approach:

   A cache with a write-through approach replicates all written data to the lower levels. The approach takes time when writing, so it needs a write buffer to accept data from the processor at full-rate. The advantage is that cache lines need only be allocated on read misses. Additionally, the approach ensures that the data seen by multiple devices remains coherent.

2. *Write-back* approach:

A cache with a write-back policy stores the data in the cache and only writes it back to the lower level on the cache spill. A cache line is allocated whenever there is a read or write miss. This reduces the write traffic to lower levels of the memory hierarchy. This method makes the store process faster but requires mechanisms to handle the data consistency, especially when the address space is shared among various processes.

In addition, access latency may be reduced by techniques such as allowing hit-after-miss, or fetching with an *early restart*, i.e. *critical word first*. The latter technique is used to deliver a requested word to the processor as quickly as possible without waiting for the complete transfer of the cache line.

2.4.2 Virtual Memory

In general, programs, data or devices are indexed in a memory by a physical address. However, logical address space is normally larger than physical address space. For example, an architecture with a 32 bit address bus can support up to 4GB (2^{32}) of memory. The availability of virtual space is ideal for parallel applications because it allows them to view the memory space independently. Access by virtual address also simplifies the allocation of physical memory resources.

The virtual address space is typically divided into pages. The address of each page has to be translated from a virtual address to a physical address. Virtual-to-physical page translations are normally held in a translation table, which is allocated in the memory itself. However, it is impractical to access memory every time a virtual address needs to be translated. Therefore, an additional cache, called a *Translation Look-aside Buffer* (TLB), is provided to store a set of recent translations [26]. Traditionally, the TLB does not only hold the page translations, but it also provides additional information to let the system validate the access permissions of the transaction before it accesses the physical memory.

2.4.3 Protection Mechanism

A protection mechanism is required to prevent each process from any malicious or accidental modification by unwelcome processes. Its duties are to distinguish between authorised and unauthorised usage, to specify the controls to be imposed and to provide a means of enforcement. Protection information is typically stored with virtual address translations since they both refer to pages in memory.

2.5 Summary

Architectural-level parallelism is required for both performance improvement and processor utilisation. Though the traditional control-flow programmers model executes instructions sequentially, implementations now support instruction level parallelism using pipelining, superscalar, VLIW and EPIC approaches. These implementations allow a processor element to simultaneously execute multiple instructions from a single thread stream. Additionally, concurrency from multiple threads may be imposed on a single processor via an interrupt mechanism leaving scheduling and context switching to software. In shared-memory multiprocessors, thread level parallelism is available but at the expense of a complexity and non-coherent protocol which requires a high performance interconnection network. Loosely-coupled multiprocessors has limited applications.

A data-flow model naturally supports instruction-level concurrency. Both static and dynamic data-flow models exploit uncontrolled parallelism. Their performance is throttled by matching store transactions. Nevertheless, ideas from such architectures inspired processor architects to enhance the control-flow model with data-dependent features, such as a register scoreboarding and an out-of-order execution.

Instruction encoding strongly influences architectural design. The simplicity in RISC architectures allows its design and implementation to be faster than that of CISC architectures. Memory access via load and store instructions in RISC allows flexible instruction scheduling and independent data transaction operations.

Memory access via virtual address imposes order between concurrent processes. The next chapter describes extended models to support thread-level parallelism, demonstrates how an amalgam of control-flow and data-flow techniques can support thread level parallelism in hardware.

Chapter 3

Multithreaded Processors

Computing is similar to cookery. Programs, like recipes, are lists of instructions to be carried out. The raw materials, like ingredients, are data which must be sliced & diced in exactly the right way, and turned into palatable output as quickly as possible.

The Economist, April 19th 2001

3.1 Introduction

A multithreaded processor employs multiple streams of execution to hide idle periods due to memory access latency or a non-deterministic interval when a processors waits for data dependency resolution (e.g. identifies independent instructions in an instruction window for issuance) [6]. The architecture trades design simplicity for efficiency by combining control-flow and data-flow at the hardware level as illustrated in Figure 3.1.

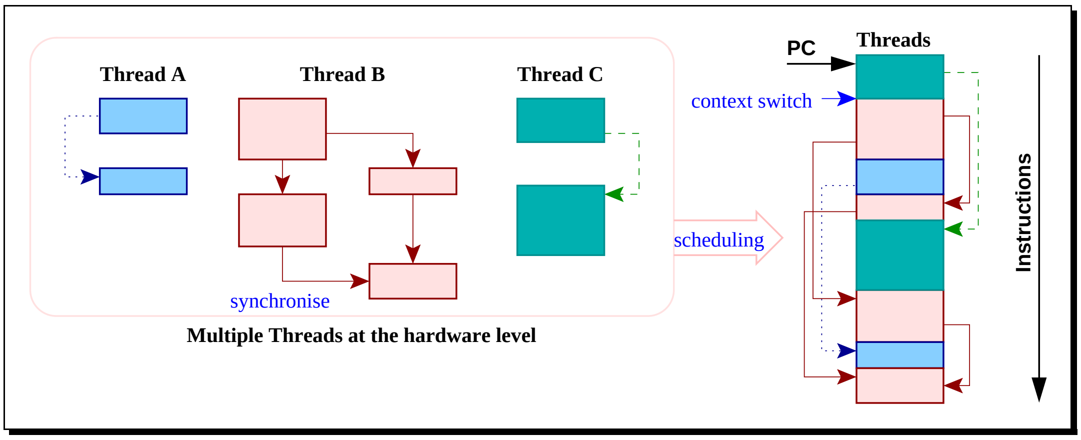


Figure 3.1: The combination of control flow and data flow.

This chapter is divided into four parts. Section 3.2 first reviews the development of multithreaded architectural design. Their supportive software and memory systems are further discussed in Section 3.3. Section 3.4 presents essential efficiency factors and appropriate evaluation methods for multithreaded processors. Section 3.5 summarises the necessary requirements for multithreaded architectural design.

3.2 Multithreaded Architectures

The idea of supporting multiple contexts in a single pipeline was first proposed around 1958 by both Honeywell [63] and Gamma 60 [64] (see Figure 3.2). In 1964, Cray CDC 6600 supercomputer proved that the architecture gained benefit by running multiple independent operations in parallel on its peripheral processor.

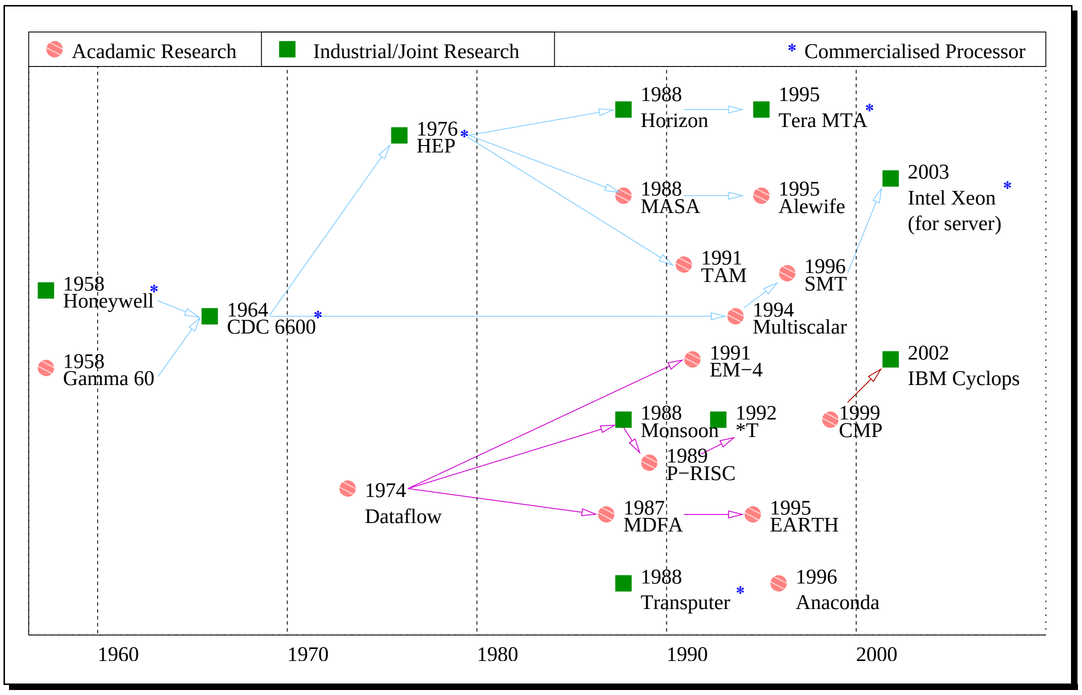


Figure 3.2: The design development of multithreaded architectures.

Later in 1976, the first commercial TLP architecture called *Heterogeneous Element Processor* (HEP) [28] was released. The HEP architecture was designed to support a large scientific calculation. Unfortunately, the HEP was not successful. Perhaps, this is due to the immaturity of software to exploit TLP and the fact that memory access latency was not significant during the period. Nevertheless, its existence demonstrated that a complete multithreaded system was feasible and served as a basis for subsequent architectures.

In 1995, Tera Corporation successfully introduced a *MultiThreading Architecture* (MTA) [29] to the market [5]. The architecture contributes a reduction of programming effort in scientific computing with scalable performance on industrial applications where parallelism is difficult to extract. Recently, Intel released a commodity multithreaded processor for servers, the Pentium IV Xeon, based on a hyper-threading technology, i.e. Simultaneous MultiThreading (SMT) [30].

Various aspects of multithreaded architectural design are presented in the remainder of this section. A context switch mechanism in a single pipeline is first explained (§3.2.1). Then, different styles of execution contexts from a range of architectures are presented (§3.2.2), followed by a concurrency control model of the multithreaded architecture (§3.2.3). The section finishes off with multithreaded extensions to a multiple-issue model (§3.2.4).

3.2.1 Context-switching Mechanisms

For the execution of a single pipeline, there are two main approaches for context switching: a cycle-by-cycle context switching and a block context switching. Figure 3.3 presents the overview of these two switching mechanisms compared with the context switching illustration in the single-threaded processor.

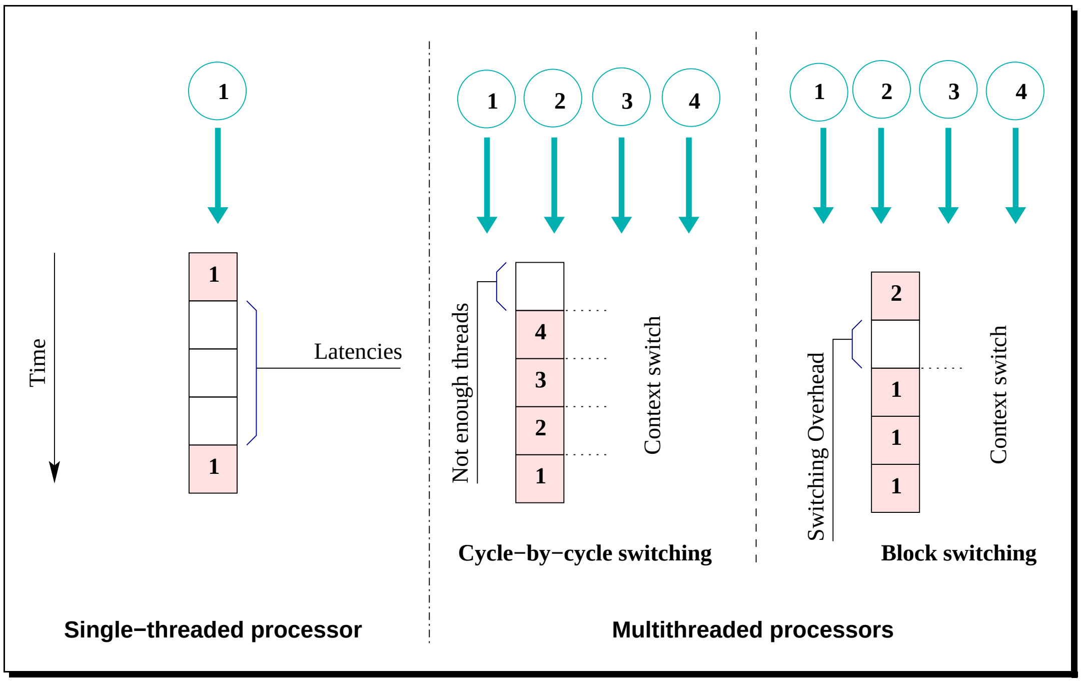


Figure 3.3: Context switching mechanisms.

Cycle-by-cycle Switching

Cycle-by-cycle switching statically interleaves threads every clock cycle using a round-robin schedule (e.g. HEP [28], Tera MTA [29], Monsoon [59], TAM [65]). The advantages of this fine-grained technique are its zero context-switching overhead and the absence of any complex circuits that are needed to handle instruction, data and control dependencies (e.g. issue speculation, data forwarding, and branch prediction)

Nevertheless, cycle-by-cycle switching poses the following disadvantages:

- The optimal number of active threads needs to be equal to or greater than the number of pipeline stages to support sufficient parallelism (e.g. Tera MTA provides 128 threads per pipeline).
- The performance of a single thread is degraded because the execution time is shared by multiple threads.
- The pipeline could be under-utilised when it processes an inadequate number of threads (e.g. less than the number of pipeline stages).

Block Switching

This technique, sometimes referred to as coarse-grain switching, allows a thread to run for a number of cycles before being switched out for the following reasons:

1. *A latency event is encountered* – A switching point is implicitly indicated by a cache *miss*, a branch delay slot, a failed synchronisation (e.g. Alewife's Sparcle processor [66]) or an *invalid* flag in a scoreboard that indicates an invalid value of the required register [67].
2. *A context-switch indicator is reached* – Context switching is explicitly indicated by special instructions. The switching signal is provided to break the current running thread into short control-flow microthreads [4]. The mechanism is necessary to avoid the domination of a single thread.

Context switching to another register space is done by either windowing a different register set on a larger register space or selecting an alternative register set using fast crossbars or multiplexers.

There are a number of advantages of the block-switching technique:

- It requires a smaller number of threads for multithreading.
- The design allows each thread to execute at full processor speed.
- The load instructions can be grouped into one thread for prefetching its subsequent thread operands in order to avoid load latency penalties.

Though switching a register set with these approaches is very efficient, it supports only a restricted number of threads. Boothe and Ranade's study [68] indicates that the mechanism is inefficient at handling short latency events if they occur frequently. This is because the run-length can be overshadowed by the context-switching penalty of several cycles to clear the processor pipeline.

3.2.2 Multiple Execution Contexts

For the control-flow based architecture, the execution context of one thread consists of a *PC* and a *register set*. In order to support multithreading at the hardware level, multiple contexts are employed in the hardware level. A number of contexts in the architecture is generally based on a hardware budget and the cost of context switching.

In early 1980s, multithreaded architectures were designed to minimise context switching by using a minimum context size (e.g. HEP's context is only the PC [28]. An INMOS Transputer's context is just two registers, one for a workspace pointer and the other for the PC [69]). However, intermediate calculation results in such architectures need to be transferred to/from the memory. This imposes a huge burden on the memory hierarchy. Later on, more use was made of registers (e.g. *T reduced data transferring by loading all necessary context into registers [70]). The context switch operation is still rather expensive.

Later models (e.g. Alewife [66], SMT [30]) added more register sets to store a few active contexts in the pipeline. The systems which evolve from data-flow architecture such as EM-4 [60], MDFA [71] and Anaconda [4] store their execution contexts in *Activation Frames* (AF). The usage of AFs breaks through the limitation in a number of contexts because it is relocatable to any memory space. In Anaconda [4], an execution context is additionally associated with presence and deadline bits for concurrency support.

3.2.3 Concurrency Control

Operating multiple threads needs a robust concurrency model to co-ordinate all activities in the system. Parallelism at the hardware level requires mechanisms to handle thread communication, thread synchronisation and thread scheduling as presented below:

Thread Communication

Data dependencies exist between multiple threads. Multithreaded architectures prefer low inter-thread communication overhead. There are two methods through which threads can communicate to one another. The first method is based on a *shared memory* (i.e. HEP [28], EM-4 [71] and Monsoon [59]). The method uses thread synchronisations such as mutual exclusion or monitors in order to wait for the data from the remote memory space. The second method is conducted via *message passing* (i.e. Alewife [66] and Transputer [69]) by which a data package is directly transferred to the other threads.

Thread Synchronisation

Thread synchronisation is a mechanism where an activity of one thread is delayed until some events from other threads occur. Control dependencies exist between multiple threads. The inter-thread synchronisation is required for co-ordinating the dependencies by synchronising their signals and data. There are two types of synchronisation. The first type is a *co-operative* synchronisation where threads wait for either data or controls from one another. An example of this method is a producer-consumer model [72].

The second type is a *competitive* synchronisation which prevents interference on shared resources. Otherwise, simultaneous accesses of multiple threads may lead to data corruption. Mutual exclusion is provided to prevent the competitive case. To support this, only a key is allowed as a guard. Only one thread is handed a key to access, i.e. *lock*, the protected resource while the others have to wait until the resource is released, i.e. *unlocked*. Each thread simultaneously holds many keys, hence it reserves many resources at the same time.

Synchronisation techniques in multithreaded architectures vary greatly. It can be a signal-graph consisting of local counters for fork/join [28, 73], full/empty bits [74], a memory based data-flow sequencing [59] or data-flow presence flags [60, 75, 29].

Thread Scheduling

Thread scheduling is crucial for multithreaded operation. The mechanism avoids resource allocation problems such as a *deadlock* and *starvation*. A thread scheduler determines the execution order of all ready threads. These threads are waiting to be dispatched in accordance with their priorities or deadlines.

In practice, scheduling multiple threads at the hardware level has usually been implemented with a simple queue that provides *First-In-First-Out* (FIFO) order, or with a simple stack that offers *Last-In-First-Out* (LIFO) order. However, such simple techniques are inadequate both for avoiding the starvation problem or in providing real-time scheduling. Therefore, information from the application level, such as a thread's *priority* or *deadline*, needs to be incorporated into the scheduling mechanism. Thus, a hardware sorting queue is required.

A number of hardware sorting queues have been introduced (e.g. a parallel sorter [76], a sorting network [77], a systolic-array sorter [78]). However, they either require more than a single cycle to make a decision, or need $O(n)$ comparators to sort n elements. Fortunately, Moore introduced a fast scheduling queue called a *tagged up/down priority queue* [8] as depicted in Figure 3.4.

The tagged up/down priority queue provides four advantages:

1. Data *insertion* and data *extraction* are performed every cycle.
2. The identical priorities/deadlines are extracted in the FIFO order of insertion in order to preserve their sequential properties.
3. Queue status (empty and full signals) is provided.
4. Only $\frac{n}{2}$ comparators are required to sort up to n data entries.

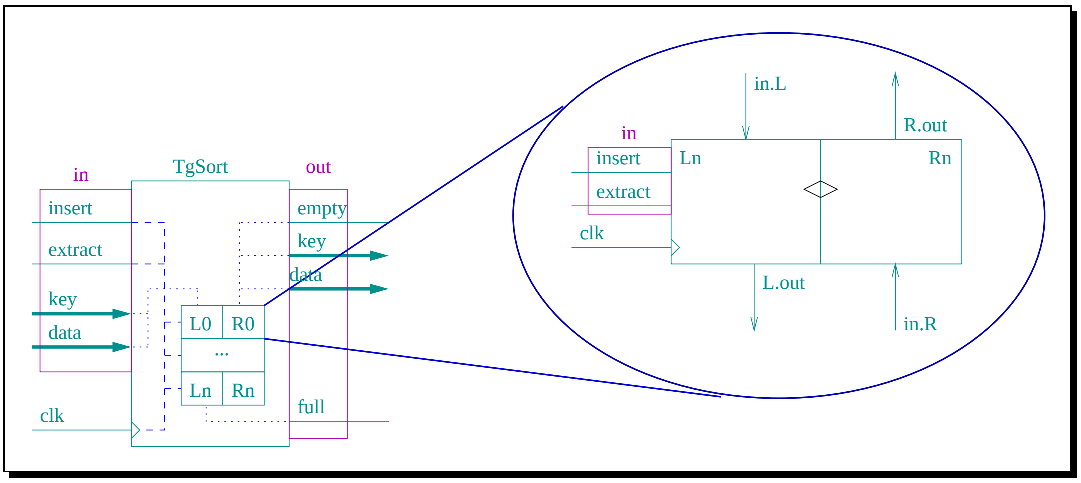


Figure 3.4: The tagged up/down priority queue.

3.2.4 Multiple-issue Models

Scaling conventional multiple-issue processors for multithreading increases complexity and cost. Thus, sufficient parallelism is exploited to meet a high-throughput goal. Three different implementations were introduced to allow instructions to be concurrently issued from multiple threads: a trace processor, a *Simultaneous MultiThreading* (SMT) and a *Chip Multi-Processors* (CMP). Figure 3.5 illustrates their multithreaded operations in comparison to two samples of single-threaded's multiple-issue architectures – a superscalar architecture and a VLIW architecture.

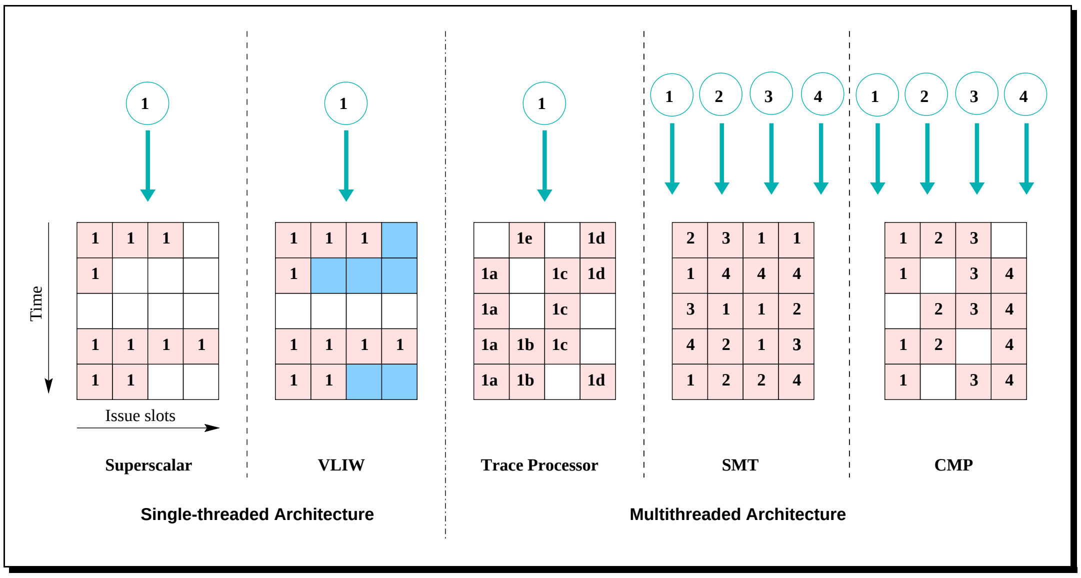


Figure 3.5: The illustration of multithreading with multiple-issue.

Trace Processor

Trace processors require very complex hardware to exploit parallelism by speculatively spawning threads without the need of software re-compilation [79, 80]. They normally use a trace cache [81] to identify threads at runtime. Inter-thread communication is then performed with the help of a centralised global register. The architecture exploits both control-flow and data-flow hierarchies using a statistical method for speculative execution supported by a misprediction handling mechanism.

Unfortunately, the additional complexity in the control logic introduces a delay into the system. Also, a large amount of hardware for speculative execution wastes power during the execution of parallel applications.

Simultaneous MultiThreading

Simultaneous MultiThreading (SMT) [30], or hyper-threading used in Intel Pentium IV Xeon processor for servers [82], is designed to combine hardware multithreading with superscalar's out-of-order execution. The architecture simultaneously issues instructions from a small number of threads. It aggressively shares all functional units among threads in addition to ILP obtained from dynamic scheduling. Different threads may issue instructions in the same cycles. This eliminates poor ILP between the subpipelines and to hide latencies within the subpipelines. This architecture supports register-to-register synchronisation and communication. It uses a special register-name checking mechanism to hide the conflicts from inter-thread competition over shared resources [83].

Nevertheless, the architecture needs to replicate register renaming logic and a return stack pointer; a number of shared resources, such as a re-order buffer, a load/store buffer and a scheduling queue, need to be partitioned; and the scheduling unit is complex because it has to dynamically allocate the resources with a speculative pre-computation that leads to a delay in every cycle.

Chip MultiProcessor

Chip MultiProcessor (CMP) [84, 85] (e.g. Sun's MAJC [86] and Raw [14]) is a symmetric out-of-order multiprocessor. This architecture decentralises functional units and requires a mechanism to distribute threads to each processor.

There are a number of advantages introduced by this architecture. Firstly, a delay from crossbars or multiplexers, used for context switching, is eliminated. This simplifies the circuity and benefits to faster execution rates. Secondly, the number of thread-processing elements for CMP increases *linearly* where as the growth in the other multiple-issue multithreaded architectures is *quadratic* because their register sets and pipelines are fully connected. Lastly, its simpler design lay-out and interconnection effectively reduces the cost of design and verification.

The disadvantages of CMP are a long context-switching overhead and the under-utilised processing elements when programs exhibit insufficient parallelism. This reflects that some functional units could be better utilised by being shared.

3.3 Support Environment

This section investigates two support environments that are crucial for multithreaded operations: a memory management system (§3.3.1) and the support from the software level (§3.3.2).

3.3.1 Memory Management

A processor normally builds up a thread's *soft context* in the memory hierarchy during execution. To support multiple soft contexts in multithreaded architecture, the roles of instruction cache, data cache and TLB need to be adjusted.

In practice, both instructions and data rely on pre-fetching/pre-loading into caches prior to execution. Often, pre-fetching and pre-loading operations are conducted by using a statistical pattern-based table or a special address translation from *Speculative Precomputation* (SP) [87]. The technique reduces performance degradation even though cache misses may have already been hidden by multithreading [88].

The first level cache is typically addressed with virtual addressing to allow the system to perform cache access in parallel with address translation. However, different contexts may use the same virtual address to refer to different physical addresses. This introduces a virtual space conflict. Possible solutions for such a case is to either associate a process identifier to each cache line or to use a virtually indexed, physically tagged cache, although temporal nondeterminism may occur if the number of entries required by active threads exceeds the resources.

Anaconda [4] offers a scalable memory tree, whose topology is extensible by a router where a message-passing scheme is used for communication. The structure results in an $O(log(size))$ memory access latency. PL/PS [89] proposes non-blocking threads where all memory accesses are decoupled from a thread and pre-loaded into a pipelined memory.

3.3.2 Software Support

Concurrency primitives in programming languages are increasingly common. For example, thread libraries or thread objects in Java [90], multithreading ID [91], Sisal [92], and Tera C [93] languages offer various multithreading models. Occam [94] has been designed to facilitate parallel INMOS Transputer [69]. Multilisp language [95] has been implemented to support the nested functional language with additional operators and semantics to handle parallel execution for the MASA architecture [96]. A number of thread libraries exist for high-level programming languages. Among them, the POSIX thread library, the UI thread library and the Java-based thread library have competed with one another to become the standard programming method for multithreading [97].

Operating Systems (OSs) additionally support multiple threads via multitasking. Early OSs (e.g. Window 3.x, Mac 9.x) used a *co-operative multitasking* which relied on the running programs to co-operate with one other. However, because each program is responsible for regulating its own CPU usage, ill-behaved programs may monopolise the processor resulting in the halting of the whole system. Alternatively, *pre-emptive* multitasking is preferred by modern OSs (e.g. UNIX, Window NT). These OSs conform to the IEEE POSIX thread 1003.x [98] standard to be pre-emptive multithreading, symmetric load balancing and portable (e.g. Lynx, VxWorks, Embedded Linux, and PowerMac). Most of them transform thread deadlines into priorities.

Window CE, Mac OS X, Sun Solaris, Digital Unix and Palm OS have a common backbone which naturally supports the execution of multiple threads in accordance with the *Unix International* (UI) standard [97]. In UI standard, a thread of execution contains the start address and the stack size. The multithreading features is the same as those required by PThread such as pre-emptive multithreading, semaphores and mutual exclusion. The pre-emptive multithreading strictly enforces time-slicing. The highest runnable-priority thread is executed at the next scheduling decision, pushing the currently-running thread back to a dispatched queue. Time-slicing can be used for preventing low-priority starvation by running in round-robin order for fairly scheduling threads of the same priority.

3.4 Efficiency Factors

This section introduces the design trade-offs in multithreaded architectures (§3.4.1). Processor efficiency is then reviewed (§3.4.2), followed by the distribution of the efficiency (§3.4.3). The section finishes off with a useful cost-effectiveness model (§3.4.4).

3.4.1 Design Trade-offs

Although multithreaded architectures offer enormous potential for performance improvement, the following trade-offs should be addressed:

- *A higher hardware cost*:
  Hardware cost becomes higher because complex mechanisms, extra storages and interconnections are needed for multithreading.
- *An incremental cycle time*:
  Extra multiplexing/interconnections at the hardware level tends to add computational delay to each execution cycle.
- *Memory Bandwidth bottleneck*:
  Simultaneous execution of multiple threads increases the memory traffic.
- *Poor cache hit ratio*:
  Moving from single thread support to multiple threads reduces data locality, thereby reducing the cache efficiency.

3.4.2 Processor Efficiency

Multithreaded processors exhibit more parallelism than their single threaded counterparts. Performance is effectively gained by context switching rather than stalling, for example, for long latency memory or branch operations. However, the balance between the performance and the cost of hardware needs to be justified. Theoretically, the efficiency (E) of multithreaded processor is determined by the following four factors [99]:

1. The number of contexts supported by the hardware (n).
2. The context-switching overhead (S).
3. The run length in instruction execution cycles between switches (R).
4. The latency of the operations to be hidden (L).

Figure 3.6 presents the relationship between the processor efficiency (E) and the number of execution contexts (n). There are two regions on this processor efficiency graph: a *linear* region on the left and a *saturation* region on the right. The *saturation point* is reached when the service time of the processor completely conceals the latency, i.e. $(n-1)(R+S) = L$. The efficiency is assessed using Equation 3.1 for the linear region and Equation 3.2 for the saturation region.

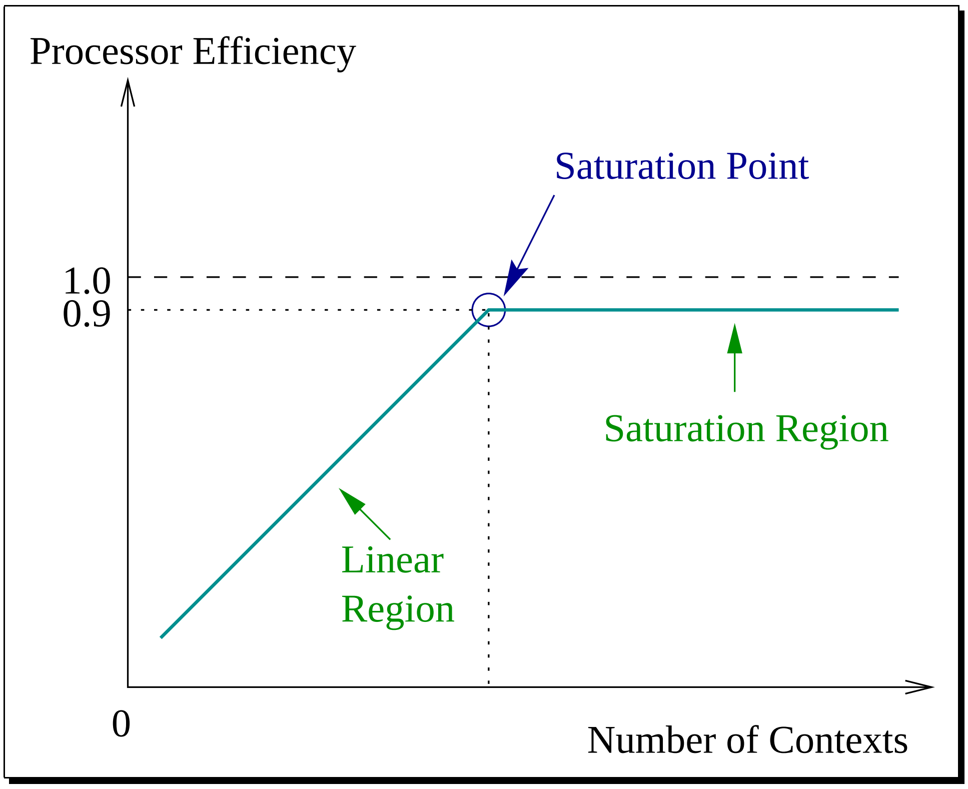


Figure 3.6: Relationship of processor efficiency and the number of contexts.

$$E_{linear} = \frac{nR}{R+S+L} \tag{3.1}$$

$$E_{saturation} = \frac{R}{R+S} \tag{3.2}$$

The equations suggest that, before reaching saturation, the processor efficiency may be improved by reducing L or S, or increasing n. Likewise, in the saturation region, the processor efficiency increases by lowering the value of S.

3.4.3 Efficiency Distribution

Although Equation 3.2 indicates that the system may offer full processor efficiency (i.e. E_{sat}=1.0) if the context-switching overhead S is zero in the saturation region, however, the full efficiency case is impossible because, in reality, a multithreaded processor is utilised dynamically.

For the saturation region, Agarwal extended an analytical model to rely more on service/workload distribution information [100]. In the model, service time intervals between context switches are distributed geometrically. A latency penalty is distributed exponentially. The processor efficiency distribution is based on a $\mathcal{M}/\mathcal{M}/1//\mathcal{M}$ queueing model [101] as presented in Equation 3.3.

$$E(n) = 1 - \frac{1}{\sum_{i=0}^{n} \left(\frac{r(n)}{l(n)}\right)^{i} \frac{n!}{(n-i)!}} \tag{3.3}$$

where	n	is the degree of multithreading
	$r(n)$	is the mean service time distribution
	$l(n)$	is the mean latency penalty distribution
	$E(n)$	is the processor efficiency distribution

3.4.4 Cost-effectiveness Model

It is insufficient to focus on improving only efficiency without considering the increased implementation costs (e.g. power, increased transistor count and design complexity). Culler has proposed a cost-effectiveness (CE) matrix to measure multithreading efficiency [3] as follows:

$$CE(n) = \frac{E(n)}{C(n)} \tag{3.4}$$

$$C(n) = \frac{C_s + nC_t + C_x}{C_b} \tag{3.5}$$

where	$E(n)$	is the processor efficiency distribution
	n	is the degree of multithreading
	C_s	is the cost for a single threaded mechanism
	C_t	is the incremental cost per thread
	C_x	is the incremental cost of thread interactions
	C_b	is the base cost of an equivalent single thread processor
	$C(n)$	is the total implementation costs

When the cost per thread (C_t) increases, the cost-effectiveness decreases as shown in Figure 3.7. Therefore, it is necessary to minimise the number of threads supported in the hardware in order to obtain the peak cost-effectiveness result.

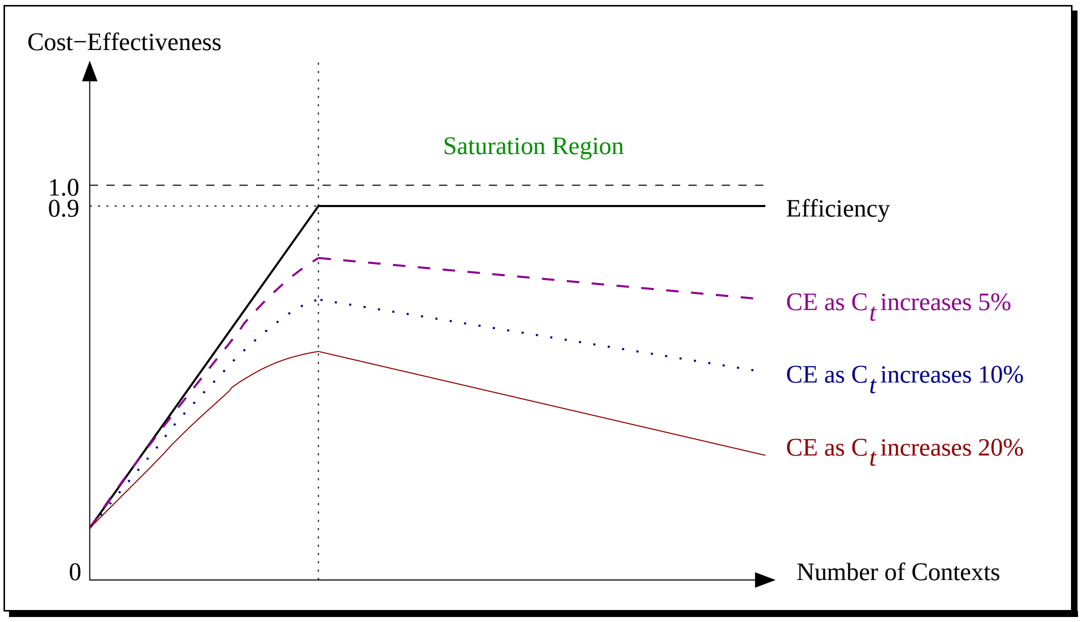


Figure 3.7: Cost-effectiveness of a multithreaded processor when varying C_t.

3.5 Summary

Multithreaded architectures have the potential to offer performance improvements by exploiting thread-level parallelism. This parallelism is capable of overcoming the limitations of instruction-level parallelism. However, sufficient instructions need to be obtained to cover long latencies (e.g. memory access, control dependencies). There are different approaches to multithreaded processor design. On the one hand, a single instruction thread size (e.g. cycle-by-cycle switching model) is desirable because of the absence of pipeline dependencies. However, this approach degrades the performance of a single thread. On the other hand, a coarse-grained thread size has better support for sequential execution.

Its performance on single threaded applications is competitive with conventional control-flow architectures, provided that the context-switching overhead is small.

Hardware support for multithreaded parallelism must address the key issues of communication, synchronisation and scheduling. Thread communication could be performed using shared memory or shared registers. Thread synchronisation is required to avoid conflicts of data. One approach is to use data-flow style synchronisation based on presence flags. Thread scheduling should be simple. However, to meet real-time constraints, a hardware scheduler is required using a priority or a deadline scheme.

Multithreaded architectures can be enhanced to support multiple issues. This is done by either decentralising the activities to compromise with hardware complexities such as the design in chip multiprocessors, or sharing functional units in order to eliminate all horizontal and vertical wastes in a way similar to simultaneous multithreading.

At the software level, a number of standard thread libraries and multithreaded applications are begining to be widely available. Various operating systems naturally provide multiple threads by multitasking. A number of operating systems offer a preemptive multitasking, which explicitly enforces a time-slice during an operation to prevent starvation problems.

The efficiency model indicates the need for a minimum number of execution contexts in the processing elements, a simple concurrency mechanism that provides a low context-switch overhead; a simple multithreading scheduler that offers real-time support; and an efficient synchronisation mechanism. For embedded systems not only execution performance is concerned, but also the cost effectiveness of the hardware and the power efficiency.

Chapter 4

Embedded Systems Design

So you can find the hidden doors to places no one's been before
And the pride you'll feel inside is not the kind that makes you fall,
it's the kind that recognises... the bigness found in being small.

A. A. Milne

4.1 Introduction

The architectures of embedded processors are substantially different from desktop processors. Embedded systems have more stringent constraints that are mostly determined by the characteristics of applications they support. In general, the systems often respond to external events, cope with unusual conditions without human intervention with operations that are subjected to deadlines.

Section 4.2 identifies design constraints that should be considered in developing feasible solutions for embedded systems. Section 4.3 introduces novel embedded architectural techniques and extensions for multithreaded operations. Section 4.4 summarises the issues raised in this chapter.

4.2 Design Constraints and Solutions

There are two important trends in the embedded market. The first trend is the growth of multimedia applications [14]. This has stimulated a rapid development for real-time media calculation (e.g. video/music coding, speech processing, image processing, 3D animation). The second trend is the growth in demand of inexpensive gadgets [102], for example, fully integrated mobile phone, game, camera, recorder, PDA and movies player.

These trends demand higher computing power, yet have minimal size, weight and power consumption. Though different embedded applications have their own sets of constraints, the nature of applications that use embedded devices shares the following five constraints:

1. Real-time response (§4.2.1)
2. Low power consumption (§4.2.2)
3. Restricted cost (§4.2.3)
4. Physical limitation (§4.2.3)
5. Application specifications (§4.2.4)

4.2.1 Real-time Response

In many embedded systems, an ability to respond (produce appropriate output) within a specified time after an input stimulus is as important as the computational performance. However, in embedded architectures the speed of a computation is not as important as knowing the time required to complete a computation. This requirement differs from general fast systems which are typically designed to minimise the average execution time in order to get overall high performance [103] and often respond to an external interrupt with a considerable context-switch overhead. For embedded designers, real-time response is another challenge.

Generally, embedded architectures reduce their interrupt latency by using a flexible register window to each execution process from a large set of registers (e.g. 192 registers in AMD 29K [104] or 6 overlapping sets of registers in ARM [39]). Its context-switching overhead is reduced because the current executing context is not required to be saved immediately.

Systems supported by a flexible register window benefit when used in conjunction with techniques derived from multithreaded architectures, such as the use of priority scheduling in Anaconda [4], or the *ready/switch* technique in MSPARC [105].

One characteristic of real-time processes is the need to accurately predict bounds on execution times to ensure deadlines can be met. Therefore, instruction speculations (i.e. branch prediction, dynamic scheduling), which are non-deterministic probabilistic mechanisms, are usually avoided by embedded systems. Embedded architectures employ alternative techniques. To lessen a role for the branch predictor, a set of *predicated instructions* have been introduced (e.g. ARM [39], PA-RISC [38] and TI DSP [106]). This enhancement extends the opcode of instructions with a conditional code, allowing the processor to execute them with fewer branches. This results in denser code and faster execution.

The correctness of timing is achieved through support in software. Currently, substantial real-time languages (e.g. real-time Java [107], Ada95 [108], SPARK [109], Ravenscar [110]) require mechanisms to handle their timing analysis, communication and concurrency. Timing analysis is normally achieved using *worst case timing* estimation [111] (e.g. assume that all loads will cause a cache miss). Communication and concurrency support in the embedded hardware is an on-going topic of research.

4.2.2 Low Power Consumption

Power consumption is critical in embedded systems. Lowering power dissipation, improves battery life, reduces heat, lowers radio electromagnatic emissions, reduces battery size and minimises manufacturing costs. The power consumption is a combination of dynamic, leakage, and static power as presented in Equation 4.1 [112].

$$\begin{aligned} P &= P_{dynamic} + P_{leakage} + P_{static} \\ &= \left(\sum a_i c_i\right) V_{dd}^2 f + nV_{dd} I_{leakage} + P_{static} \end{aligned} \tag{4.1}$$

where,	P	is the power consumption
	a_i	is the activity factor of unit i
	c_i	is the capacitance of unit i
	V_{dd}	is the power supply voltage
	f	is the input frequency
	n	is the number of transistors
	V_{th}	is the threshold voltage
	$I_{leakage}$	is the leakage current

A quadratic reduction of dynamic power is achieved by lowering the supply voltage (V_{dd}) [113]. The SIA International Technology Road-map (2002 edition) [10] predicts that the voltage could be reduced to 0.4V by 2016.

The left-hand chart in Figure 4.1(a) presents evidence that decreasing the supply voltage (V_{dd}) or increasing the threshold voltage (V_{th}) reduces power consumption. Nevertheless, these adjustments increase the circuit delay as illustrated in Figure 4.1. Therefore, V_{dd} cannot simply be reduced because it affects the necessary timing assumptions. The minimum gate delay of the chip directly depends on the switching threshold of the transistor (V_{th}) and the supply voltage (V_{dd}) as presented in Equation 4.2 [114]. Due to this reason, most embedded processors operate with a slow clock in order to save power.

$$t_{min} \approx \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}} \tag{4.2}$$

where,	t_{min}	is the minimum gate delay (cycle time)
	α	is the constant value, currently around 1.3

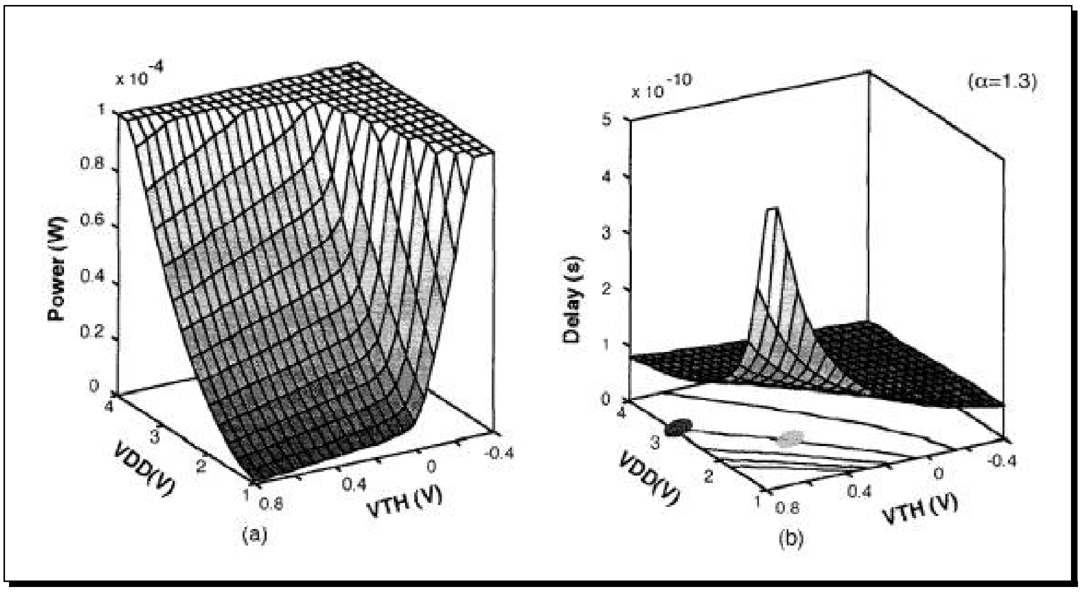


Figure 4.1: The relationships of power consumption and circuit delay with the supply and the threshold voltage [115].

On the other hand, lowering the supply voltage (V_{dd}) also reduces the noise margin and increases leakage. This is because the 10-15% decrement of V_{dd} directly reduces V_{th} as shown in Figure 4.1. The reduction of V_{th} doubles the leakage current (I_{leak}) as presented in Equation 4.3. This shows that V_{dd} can only be reduced to a value that allows reliable operation in the envisaged environment.

$$I_{leak} \quad \propto \quad e^{\frac{-qV_{th}}{kT}} \tag{4.3}$$

where, q, k and T are constant values

Alternative techniques for lowering power consumption are *transistor sizing*, *dynamic voltage scaling* [116, 117], *dynamic frequency scaling*, *dynamic pipeline scaling* [118], *transition reduction* and *clock gating* [119]. The purpose of these techniques is to reduce activity (a_i) and circuitry (c_i), which are responsible for the dynamic power consumption by using *a power operating mode* (i.e. running, standby, suspend and hibernate). These various techniques can be combined to reduce power consumption. But all of these power saving techniques come at the expense of performance [118, 120].

As transistor geometries decrease, leakage power is increasing as presented in Figure 4.2. Instead of wasting leakage power, *instruction level parallelism* utilises otherwise idle functional units by maintaining their operation. However, this technique is speculative and may waste a significant amount of power on mispredicted tasks. Further research is required to exploit parallelism in an energy efficient way.

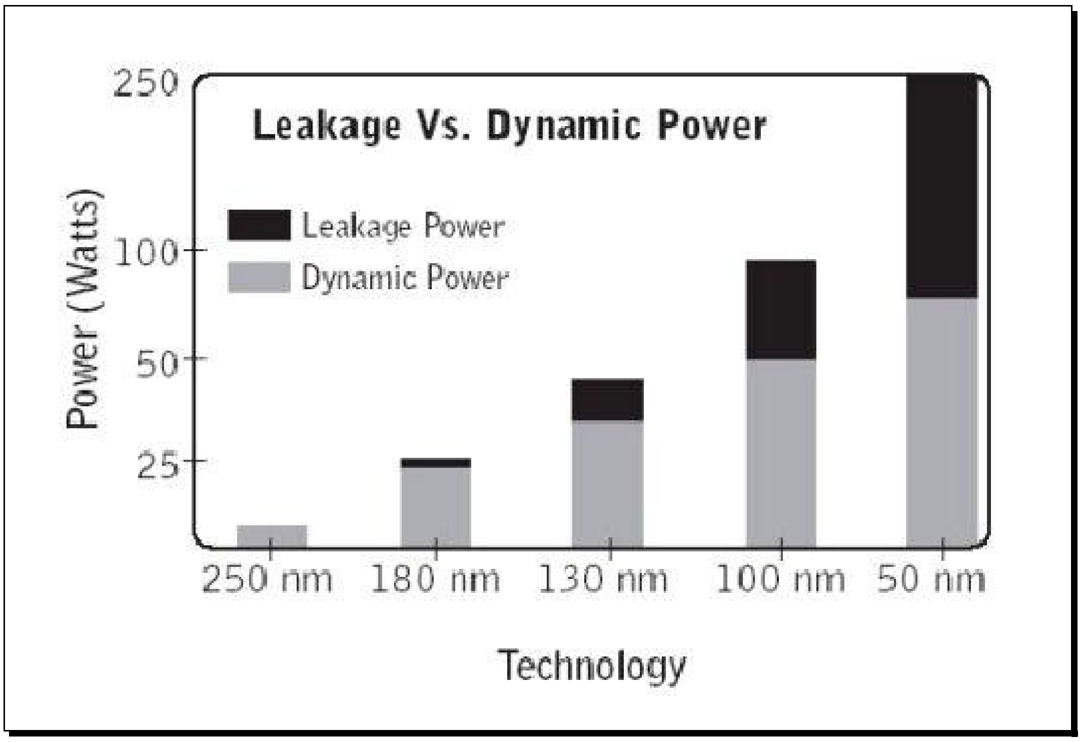


Figure 4.2: Leakage power on the increase [113].

4.2.3 Low Cost and Compact Size

Embedded processors trade performance for a lower cost and a smaller size. For a compact size, embedded architectures often have shallower pipelines, which operates with a small number of registers and uses a reduced bus bandwidth [120]. The cost of a processor is influenced by architectural features, materials, packaging techniques, on-chip circuits, fabrication yields and design & development costs.

By the year 2006, the state-of-the-art in *Very Large Scale Integration* (VLSI) is expected to provide one billion transistors on a single chip [10]. As the number of transistors increases, more components (e.g. memories and peripherals) are being integrated onto one chip to produce *System On Chip* (SOC) [13].

Advances in many factors are driven by improvements in transistor lithography processes [10]. A trend in [121] proposes that on-chip caches increasingly dominate chip area. These caches reduce the number of memory accesses, thereby reducing latency and power. Bigger caches allow a larger working set to be held from one application and reduce contention for concurrent applications.

The performance of caches relies on statistical properties which become less effective when running multiple applications in parallel. The cache area could be reduced without performance penalty provided the cache area is managed better by exploiting knowledge about scheduling decisions. Furthermore, the saving in cache area could be used to support such a management mechanism.

4.2.4 Application Specifications

Performance improvements for embedded systems are often obtained by tailoring the processor design to its application requirements. In general, there are several degrees of application specifics as classified in Figure 4.3. This figure presents the trend of performance improvement, power dissipation and flexibility when architectures dedicate their hardware infrastructures to specific applications.

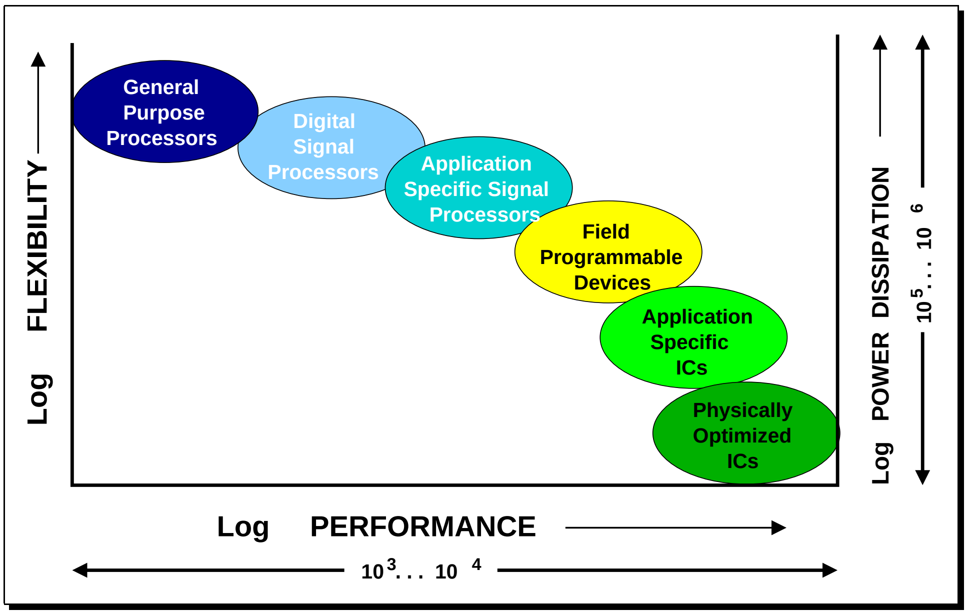


Figure 4.3: Embedded Processor Classification [122].

General-purpose embedded processors (e.g. ARM9E [39], MIPS32 [36], Motorola 68K [16] and AMD 29K [104]) are the most flexible architecture designed for personal handheld equipment such as PDAs, mobiles, cameras, which must support an embedded OS (e.g. Windows CE, Embedded Linux, Symbian). Therefore, the core architectures mostly inherit from successful general-purpose processor designs, however using a lower scale for reducing power and size.

Digital Signal Processors (DSPs) are available for numerically intensive applications with streamed data. Their architectures often offer mechanisms to accelerate signal processing such as single-cycle multiply-and-accumulation, saturation arithmetic, fixed-point notation, extended addressing modes for array & circular buffer handling and multiple memory accesses [106].

A processor can be enhanced when co-designing its core with specific software support in the form of *Application Specific Signal Processors* (ASSP). The ASSP core benefits from application-specific instructions that are tailored for particular calculations to serve a narrowly defined segment of services. These processors are designed to be programmable in order to have flexible extensions. Examples include media processors which are designed for massive floating point calculation with high memory bandwidth [123, 40] and network processors that focuses on rapid communication [124].

Field Programmable Devices(FPD) enable users to customise task-specific processors by using device programming utilities [125]. Better still are *Application Specific Integrated Circuits* (ASIC) which achieve a more compact computation core. Furthermore, full custom *physically-optimised IC* designs is available for the highest level of optimisations.

4.3 Techniques in Embedded Architectures

Embedded processors employ various techniques from microprocessors. In general, embedded architectures rely on a flexible processing element that consists of approximately 3 to 6 pipeline stages. Their execution units approximately support a 16-bit to 32-bit data/instruction width and issue approximately 1-4 instructions per cycle (e.g. integer, floating point, load/store and MAC subpipelines).

The following sections explain the details of critical features of architectural innovation for the embedded domain such as a compact instruction encoding (§4.3.1), predicated instructions (§4.3.2), subword-level manipulation (§4.3.3) and thread level parallelism support (§4.3.4).

4.3.1 Specially Compact Instruction Encoding

An embedded processor can have an instruction set architecture based on CISC (e.g. Motorola 68K [16]) or RISC (e.g. ARM [39], MIPS [36]) approaches. The advantage of CISC instructions is good code density, which allows efficient use of memory, reduces the instruction bandwidth and provides an unaligned access capability. The disadvantages of CISC are that registers are inefficiently used which increases memory traffic, and instruction decoding is complex which results in complex control.

RISC achieves high performance due to hardware simplicity. However, RISC instructions are less dense than CISC ones. Thus, some embedded RISCs offer a solution in the form of two instruction formats: full and compact forms, e.g. ARM and ARM Thumb [39], MIPS32 and MIPS16 [36]. The compact form can offer a 40% reduction in memory footprint [1], despite requiring more instructions to complete the same task.

4.3.2 Predicated Instructions

Processors waste a significant number of cycles because of branch mis-predictions. This is because, all mispredicted instructions must be flushed from the pipeline. Therefore, a branch predictor is required [26]. However, these mechanisms are expensive in terms of circuit area and power consumption.

ARM [39] goes to the extreme of making every instruction predicated. Predicated instructions are used to reduce the number of branches. These instructions are only allowed to commit their results to the registers if their conditions match the status register.

Using them, a high proportion of conditional branches are avoided. The usage of predicated instructions instead of some branches leads to a compact instruction stream due to an elimination of a number of *branch delay slots*. Nevertheless, the implementation of predicated instructions requires additional bits for a predicated field.

A compromise solution to let an instruction remain compact but avoid a number of branch usages is based on partial predication such as *conditional moves* and *conditional loads*. Such technique moves/loads value when its condition is true. Using these instructions has proved to provide an acceptable ratio of branch elimination [27].

4.3.3 Subword-level Manipulation

Programmes that manipulate data at the bit level are necessary in some embedded domains. Bit manipulating instructions such as bit test, bit clear and bit set [16] have been introduced to handle encryption algorithms, calculate checksums, and control peripheral hardware. Furthermore, subword-level manipulation instruction such as BSX (Bit Section eXtension [126]) has been used to reduce the overhead of packing/unpacking narrow width data (e.g. less than a word but more than one bit).

4.3.4 Thread Level Parallelism Support

Conventional embedded processors handle time-critical events by interrupting the current process. In order to simulate concurrency, the single threaded control-flow processor must perform a software driven context switch which has considerable overhead. On the other hand, multithreaded processors provide hardware support for context management. Furthermore, low level scheduling decisions may be performed in hardware [4, 105] rather than in the core of a *Real-Time Operating System* (RTOS).

Hardware support for best-effort scheduling needs only use a couple of shared register sets to hold the contexts and a simple priority based scheduling scheme. For example, the *differential MultiThreading* (dMT) [127] architecture for embedded processors supports two active contexts by duplicating the PC, register file, fetch-decode latch, decode-execute latch and execute-memory latch. The thread is switched out when a cache miss occurs and will be resumed when the cache miss is resolved provided no higher priority thread is available. Of course multithreading can be combined with chip multiprocessor technique (see Chapter 2) to allow more than one thread to be executed in parallel.

Embedded systems with multiple competitive threads require good scheduling to meet real-time needs. A hardware scheduler that makes best-effort decisions on a couple of execution contexts appears to be inadequate. To meet real-time constraints, an alternative multithreading architecture that can make decisions based on all execution contexts is required.

4.4 Summary

The embedded market is growing and requires an architecture that can provide real-time response, low power, low cost, compact size and high performance. Although there are trade-offs between hardware performance and power dissipation in an embedded processor, various techniques have been proposed to meet these challenges:

1. *Power operating modes* -

   Power management decisions need to be made at an architectural level in order that higher level information (e.g. scheduling decisions) can be used to control power saving circuits. This control information might be communicated as power operating modes, e.g. *running*, *standby*, *suspend* and *hibernate*.

2. *Stripped down circuits* -

   To fit with embedded constraints, non-deterministic mechanisms such as speculative scheduling or branch prediction are avoided. Embedded architectures employ alternative techniques to compensate for the elimination of mechanisms such as branch prediction. For example, rather than predicting branches, the number of branches could be reduced by using predicated instructions.

3. *Multithreading* -

   Processing a single thread stream often leaves many functional units of the processor under-utilised which wastes leakage power. Speculative branches combined with instruction level parallelism may improve functional unit utilisation though complexity goes up and predictability reduces. Exploiting concurrency at a thread level appears to be a better approach provided context switch overheads can be minimised. A multithreaded approach appears to offer good thread level parallelism, and could support real-time systems if an appropriate scheduling mechanism were used. Thus, there is room for much improvement.

Part II

Architectural Design

Chapter 5

System Overview

One must learn by doing the thing,
for though you think you know it,
you have no certainty until you try.

Aristotle

5.1 Introduction

The investigation of architectural level parallelism in Chapter 2 and the review of multi-threaded processor architectures in Chapter 3 suggest that thread level parallelism at the hardware level is a tangible approach to improve processor performance. Nevertheless, enhancing a processor to exploit thread level parallelism is often limited by a number of embedded design constraints as reviewed in Chapter 4 . As a result, alternative design choices are proposed in this chapter for my *MultiThreaded Embedded Processor* (MulTEP) architecture.

Section 5.2 starts with the background to MulTEP. Section 5.3 presents MulTEP's programming model. Section 5.4 shows my theoretical investigations on the design choices, which were conducted prior to its implementation. Section 5.5 summarises the system overview of MulTEP.

5.2 Background to the MulTEP Architecture

As mentioned in Chapter 1, the research is motivated by the growth of the embedded systems market and my personal interest to implement a novel architecture. This section addresses my research objectives (§5.2.1), the design challenges of the architecture being considered (§5.2.2) and the design choices made (§5.2.3).

5.2.1 Objectives

My research attempts to bring the advantages of *Thread Level Parallelism* (TLP) to embedded systems. In correspondence to this attempt, my objectives are:

- To improve processing performance by effectively hiding idle slots in one thread stream (e.g. from the increasing memory latency or branch/load delay slots) with the execution of other threads.
- To schedule threads to respond to their requests in real-time.
- To implement thread level parallelism and real-time support at the hardware level without much incremental hardware cost.

5.2.2 Project Challenges

Before implementing a multithreaded embedded architecture, I identified design challenges using the background knowledge as reviewed in Chapter 3 and Chapter 4, and then prioritised them with regards to my research objectives. Table 5.1 presents the criteria with the priorities I assigned.

Aspect of Research	Design Challenge	Priority		
		Critical	High	Moderate
Multithreading	Support TLP in hardware	√		
	Avoid high memory bandwidth		√	
	Avoid cache conflicts		√	
	Reduce physical complexity		√	
	Allow thread scalability		√	
Embedded system	Response in real-time	√		
	Use power effectively	√		
	Compact Size			√
	Work in harsh environments			√
	Low radio emission			√
	Low cost of end product			√

Table 5.1: Project criteria and the given priorities.

For *multithreaded* design, I gave a critical priority to the implementation of thread level parallelism in hardware. The side effect of incorporating thread level parallelism into hardware, namely memory access bottlenecks, cache conflicts and design complexities, are given a high priority since ignoring them may severely degrade the performance. For *embedded* design, I have given real-time response and power efficiency with a critical priority. I assigned a high priority to scalability because it is important for further improvement. For the other embedded challenges, which rely on fabrication techniques, such as cost, size, reliability and radio emissions, a moderate priority is given.

5.2.3 Design Choices

This section presents design choices that are raised to satisfy the selected critical-priority and high-priority challenges presented in the previous section. Table 5.2 shows a list of design choices from the multithreaded design viewpoint.

Challenge	Priority	Design Choice
TLP in hardware	Critical	C1: Provide a mechanism to progress a thread through its life cycle C2: Provide thread concurrency mechanisms C3: Eliminate context-switch overhead C4: Prevent thread starvation
Memory bandwidth	High	H1: Provide a hardware mechanism to handle memory traffic of multiple threads
Cache conflict	High	H2: Enhance cache replacement policy
Physical complexity	High	H3: Avoid very large register sets and complex control structures
Thread scalability	High	H4: Allow the number of threads to scale without significant architectural change

Table 5.2: Multithreaded design challenges.

To implement thread level parallelism in hardware, design choices C1 and C2 are proposed to distinguish multithreaded architecture from the single-threaded architecture. Additionally, I introduced design choice C3 because, in my opinion, a multithreaded processor should benefit from the execution of multiple threads without any wasted cycles on context-switching overhead.

Design choice C4 is concerned with the situation that may arise when concurrency support is migrated from software to hardware.

Side effects of multithreading result in design choices H1 and H2. Design choice H1 reflects my belief that introducing a separate hardware mechanism to handle memory traffic may reduce the memory bandwidth limitations. For cache conflicts (H2), the conflict is a result of multiple threads competing against each other by repeatedly swapping out of each other's working sets. Therefore, I believe that an enhanced cache replacement policy may minimise the problem. For the physical complexity challenge, design choice H3 is proposed since a smaller number of registers will reduce the physical complexity. However, this small number must yield sufficient performance to allow the processor to benefit from multithreading. Design choice H4 is my proposal for flexible scalability in terms of the number of threads.

Table 5.3 presents design choices to satisfy the embedded design challenges. Design choice C5 is inspired by the availability of multiple priority levels for real-time support in many embedded OSs (see Chapter 4). Design choice C6 is required for an effective real-time response. However, there should be a compromise between this choice and the need to reduce physical complexity (H3). The power efficiency goal (C7) reflects the desire to provide power saving control from an architectural level (see Chapter 4) though specific power saving circuits is beyond the scope of this thesis.

Challenge	Priority	Design Choice
Real-time response	Critical	C5: Provide a hardware mechanism to issue threads with regards to their priorities
		C6: Allow a hardware to make a scheduling decision based on the requests of all threads
Power efficiency	Critical	C7: Design a hardware mechanism to provide operating modes for power saving

Table 5.3: Embedded Design challenges.

5.3 MulTEP System

Architectural approaches raised in the previous section are implemented in MulTEP. These approaches are summarised in Table 5.4[1] together with forward references to implementation details.

Choice	Technique	Approach	Implementation
C1:	▷ Hardware thread life cycle model	see §5.3.4	see §6.3
	▷ Software support daemon	see §7.3	see §7.3.1
C2:	▷ Synchronisation mechanism	see §5.3.5*	see §6.3.3
	▷ Scheduling mechanism	see §5.3.6*	see §6.2.3, §6.3.4
	▷ Multiple execution contexts	see §5.4.1	see §5.3.2
C3:	▷ Zero-cycle context switches	see §5.3.6	see §6.2
C4:	▷ Dynamic-priority mechanism	see §5.3.6	see §6.3
C5:	▷ A priority-based scheduler	see §5.3.6	see §6.3.4
C6:	▷ A Tagged up/down scheduler	see §3.2.3*	see §8.2.1
C7:	▷ Power operating modes	see §4.2.2*	see §5.3.8
H1:	▷ Load-store unit	see §6.4	see §6.4
H2:	▷ Tagged priority cache line	see §6.5	see §6.5
H3:	▷ 4 contexts for 2 processing elements.	see §5.4	see §6.2
H4:	▷ An activation frame representation	see §2.3.3*	see §5.4.2

Table 5.4: The MulTEP solutions.

In the next section, an overall picture of the MulTEP system is presented (§5.3.1). The execution context storage is explained in (§5.3.2). This is followed by the instruction set architecture (§5.3.3) and thread life cycle model (§5.3.4). Synchronisation (§5.3.5) and scheduling (§5.3.6) are then introduced followed by the memory protection (§5.3.7) and power operating modes (§5.3.8).

[1]An entry marked by '*' is an idea borrowed/inspired from techniques in the other architectures.

5.3.1 Overall Picture

The overall picture of the system from the user level to the hardware level is illustrated in Figure 5.1. Programs from the user level are compiled to native code. Software tools then either convert the programs into several threads (e.g. Program `A`) or preserve the existing flow of instructions (e.g. the sequential form of Program `B` and the multithreaded form of Program `C`). Threads from the kernel level are issued to the hardware level according to their priority.

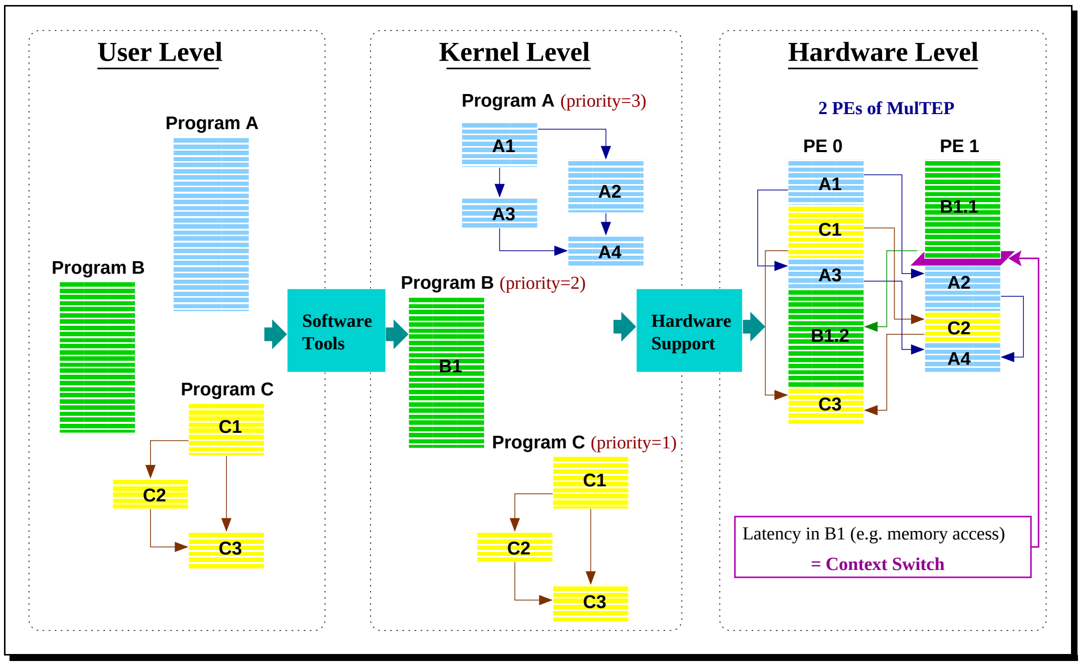


Figure 5.1: The overall picture of MulTEP.

Each program is represented as a directed graph where *nodes* represent *nanothread* blocks [128, 129] and *arcs* represent their communication events. Nanothreads sequentially operate using the control-flow model, yet effectively realises the natural concurrency behaviour of the data-flow model to support multithreading.

The processor consists of two processing elements sharing four execution contexts (see §5.4.1). Excess execution contexts spill out to the memory in forms of activation frames (see §5.3.2). Circuits for non-deterministic operations, such as branch prediction and out-of-order execution, are excluded from the design leaving the architecture to rely on thread-level parallelism.

Threads progress through their life cycle (see §5.3.4) using four multithreading instructions: `spawn`, `switch`, `wait` and `stop` (see §5.3.3). Thread scheduling is dynamic and based on priorities (see §5.3.6). A mechanism to switch execution contexts operates when a stall or an `NOP` occurs (see §6.3.4). MulTEP uses a multiple data entry matching-store for thread synchronisation, which is similar to that used by Anaconda [4].

5.3.2 Execution Context

The processing element prototype is based on the RISC processor DLX [24] since the instruction format is simple to decode, the integer pipeline is simple and yet gives good performance (see Chapter 3 for background requirements). Thus, the execution context of one thread consists of a program counter, 31x32-bit integer registers (excluding register $zero whose data is always 0) and 32x32-bit floating-point registers. The left-hand part of Figure 5.2 presents the execution context of the processor. The right-hand part of the figure shows the preserved form of the execution context, called an *Activation Frame* (AF) [4], which may be spilled to the memory system.

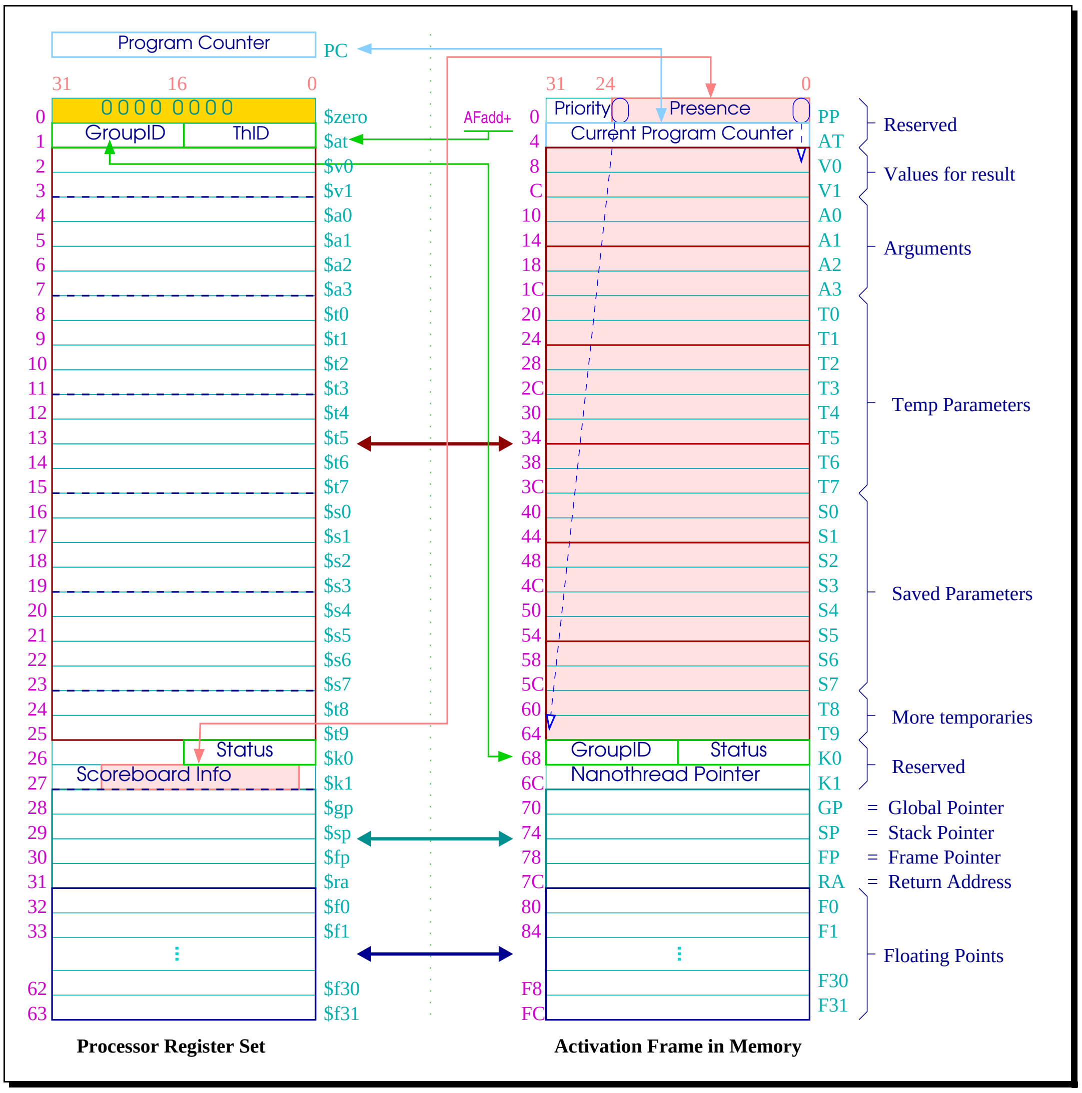


Figure 5.2: The execution context and the activation frame.

In each activation frame, a space for the register `$zero`, called `PP`, is reserved for two purposes. The first 8 bits store a *priority* for scheduling. The next 24 bits store *presence flags*, which indicate the presences of 24 input parameters, for handling complex data dependencies. `AT`, `K0` and `K1` are reserved[2]. `AT` stores a program counter. `K0` stores a group ID and a status[3]. `K1` stores a nanothread pointer (i.e. the start PC of the thread).

5.3.3 Instruction Set Architecture

MulTEP is based on the 32-bit MIPS IV instruction set [130][4]. The processing element is divided into five execution units: a load/store group, an ALU group, a jump/branch group, a miscellaneous group and a multithreading group.

Load/Store Instructions

The load/store instructions support both big-endian and little-endian on byte, halfword, word, double word, unaligned word and unaligned double word data. Signed and unsigned data of different sizes are transferable by sign-extended and zero-extended mechanisms. The *register+displacement* addressing mode is supported by the architecture as follows:

- `[load|store]` *reg*, *offset*(*base*)

opcode	base	reg	offset
6 bits	5 bits	5 bits	16 bits

ALU Instructions

The ALU group consists of arithmetic and logical instructions. Each may operate with an operand from a register or a 16-bit immediate value. Arithmetic instructions are signed and unsigned based on a two's complement representation. The instructions are coded as follows:

- `r-type` *rd*, *rs*, *rt*

opcode	rs	rt	rd	sh	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `i-type` *rt*, *rs*, *immediate*

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- `[mul|div]` *rs*, *rt*

0x00	rs	rt	0x00	mul/div
6 bits	5 bits	5 bits	10 bits	6 bits

Speculative execution of branches is not supported (see §5.3.1). To reduce the performance penalty of branches, predicated instructions are used. The MulTEP architecture adds predication indicators to the *sh* field of register-based arithmetic instruction as a list of ISA pro-fixes presented in Table 5.5.

[2]Registers `$at`, `$k0` and `$k1` are originally preserved for the kernel.

[3]The status area of register `$k0` is reserved for flags Negative (`N`), Zero (`Z`), Carry (`C`), oVerflow (`V`), Greater (`G`), Lower(`L`), Equal (`E`) and execution Mode (`M`).

[4]MulTEP is enhanced with the MIPS IV ISA in addition to the MIPS I ISA from the RISC DLX architecture to match code produced by the cross compiler (see Chapter 7).

ISA pro-fix	Flags				
	Zero	**N**egative	**G**reater	**L**ower	**E**qual
-z	1	-	-	-	-
-ltz	-	1	-	-	-
-gtz	-	0	-	-	-
-eq	-	-	0	0	1
-lt	-	-	0	1	0
-lte	-	-	0	1	1
-gt	-	-	1	0	0
-gte	-	-	1	0	1

Table 5.5: The representations of flags in the status register.

Jump/Branch Instructions

The branch instructions are PC-relative and conditional. Jump instructions are either PC-relative unconditional or absolute unconditional. Their formats are:

- `b-type` *rs*, *rt*, *offset*

branch	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

- `j-type` *target*

jump	offset
6 bits	26 bits

Miscellaneous Instructions

The miscellaneous group consists of software exceptions, such as `SYSCALL` and `BREAK` (`s-type`), `sync`, conditional move and pre-fetch instructions, for example:

- `s-type`

0x00	code	func
6 bits	20 bits	6 bits

- `sync`

0x00	code	rt	sync
6 bits	15 bits	5 bits	6 bits

Multithreading Instructions

Four additional instructions to control threads (see §5.3.4) are implemented:

1. `spawn` *reg*, *address*

0x1C	reg	address
6 bits	5 bits	21 bits

2. `wait` *reg*

0x1D	reg		0
6 bits	5 bits	20 bits	1

3. `switch`

0x1D		1
6 bits	25 bits	1

4. `stop` *reg*

0x1E	reg	
6 bits	5 bits	21 bits

5.3.4 Thread Life Cycle

MulTEP is able to progress a thread through its life cycle as presented in Figure 5.3 using four multithreading instructions (see §5.3.3). A thread is created by the `spawn` instruction and then waits in the non-running state (i.e. born, joining, blocked, suspended, sleeping and waiting). A `Store` instruction is sent to synchronise the thread through a hardware matching-store synchroniser. A thread becomes ready when all inputs are present. The hardware scheduler promotes the highest-priority ready thread to the running state.

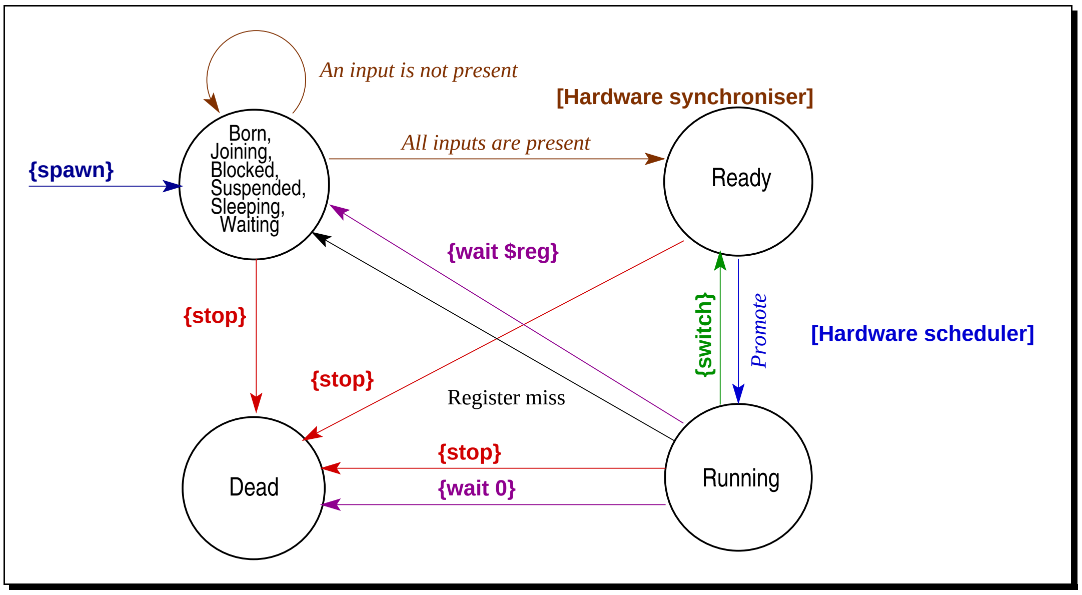


Figure 5.3: A thread life cycle in hardware [131].

The hardware mechanism to enable this life cycle is presented in Section 6.3. While running, a thread can be switched back to the ready state by the `switch` instruction which releases the processing element so that it can be used by another thread. The switch occurs when the execution has to wait for a register. This arises when the execution reaches the `wait $reg` instruction or detects a *register miss*, i.e. accessing a not-ready register indicated by the scoreboard.

The `wait 0` instruction represents the completion of a nanothread block[5]. During execution, a thread can be killed by the `stop` instruction issued either by itself or by another thread.

[5]The pseudo code of the `wait 0` is `end` (see Chapter 7)

5.3.5 Synchronisation Technique

Thread synchronisation is achieved through message passing via a store instruction. As execution contexts are memory-mapped in the form of activation frames, threads communicate by storing data to appropriate activation frames. A thread whose entire presence flags are asserted can be dispatched to the processing unit by the hardware scheduler with regard to its priority. Otherwise, the thread needs to wait.

For example, in Figure 5.4, Thread A1 sends a value N to register x of Thread A2. The matching-store synchroniser translates the destination to a specific memory address and then stores the value N to that address. Concurrently, the synchroniser turns on the presence flag of that register. As the synchroniser detects that all inputs are present, the thread ID and its priority are dispatched to a data and a key of the tagged up/down scheduler, respectively.

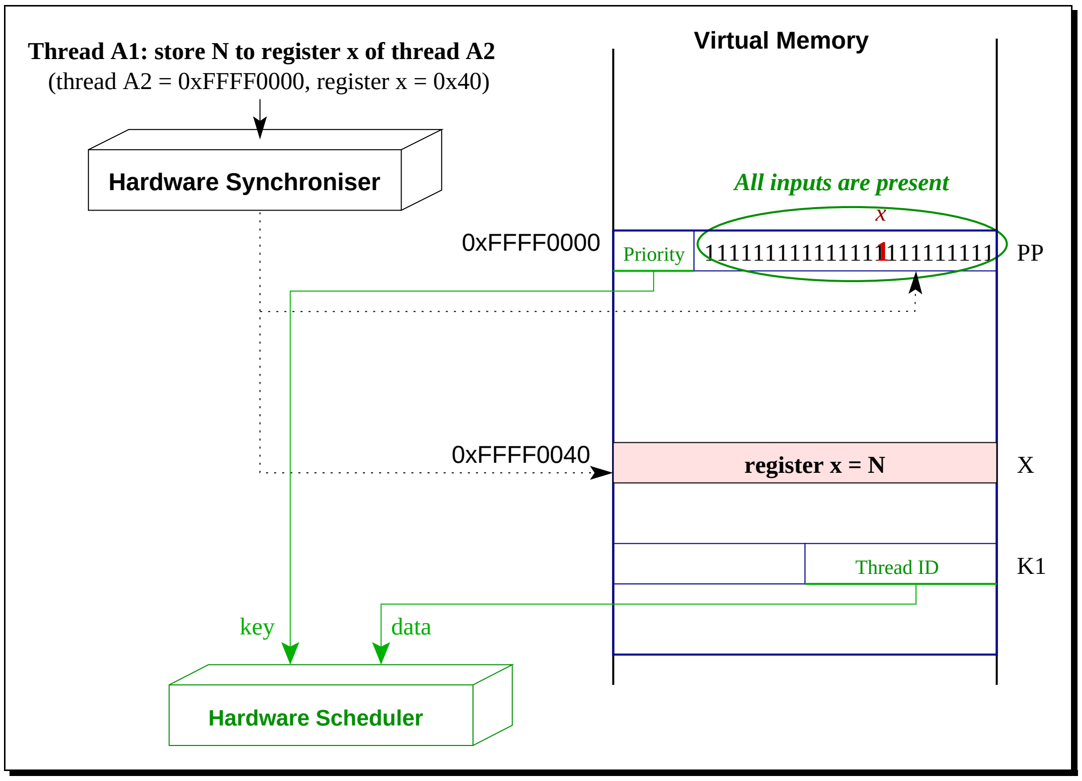


Figure 5.4: The store signal is used for synchronising thread.

5.3.6 Scheduling Technique

Scheduling multiple threads is determined by thread priorities. Pre-emptive multi-tasking is applied to enforce time-slicing. This prevents a high priority thread from dominating shared resources (see §6.2.1). Figure 5.5 presents an example of thread scheduling. The hardware scheduler is based upon the tagged up/down priority queue [8]. To prevent the lowest priority thread from starving, all in-queue priorities are incremental when a new thread arrives (see Appendix A.2.1).

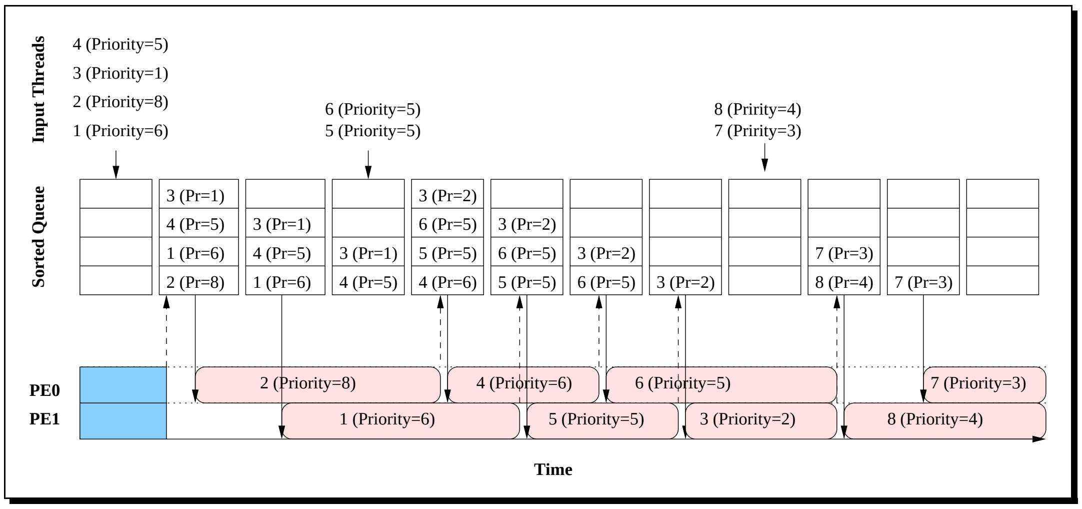


Figure 5.5: Dynamic-priority multithreading illustration.

5.3.7 Memory Protection

To protect unwanted interferences between threads, I introduced two protection modes: *user* mode and *system* mode. Most threads are executed in user mode. The system mode is provided for kernel operations such as exception handling threads, housekeeping system daemon threads and software traps.

The protection mode is tagged with the virtual page on the *Translation Look-aside Buffer* (TLB) line. A group ID is added to provide separate virtual address spaces for groups of threads. To guide the cache line replacement policy, a cache line incorporates the thread priority of its owner. The association of a thread priority is introduced to minimise cache conflicts caused by competitive working sets.

5.3.8 Power Operating Modes

MulTEP offers four power operating modes: *running*, *standby*, *suspend* and *hibernate*. The system operates in the running mode when a thread is available in the pipeline. The standby mode is activated once the pipeline becomes empty, detected by the absence of threads from all execution contexts. The suspend mode is encountered when there are no threads in either the execution contexts or the scheduling queue. The hibernate mode is issued when only a kernel thread is available in the system, which is waiting to be activated by a timing signal.

These four power operating modes are available for implementers to supply a suitable power management technique (see §4.2.2). Figure 5.6 illustrates the state diagram and the I/O interfaces of the power-mode generator. The power operating modes reflect the scheduling information. To maintain this, the generator is allocated in the multithreading service unit (see §6.3).

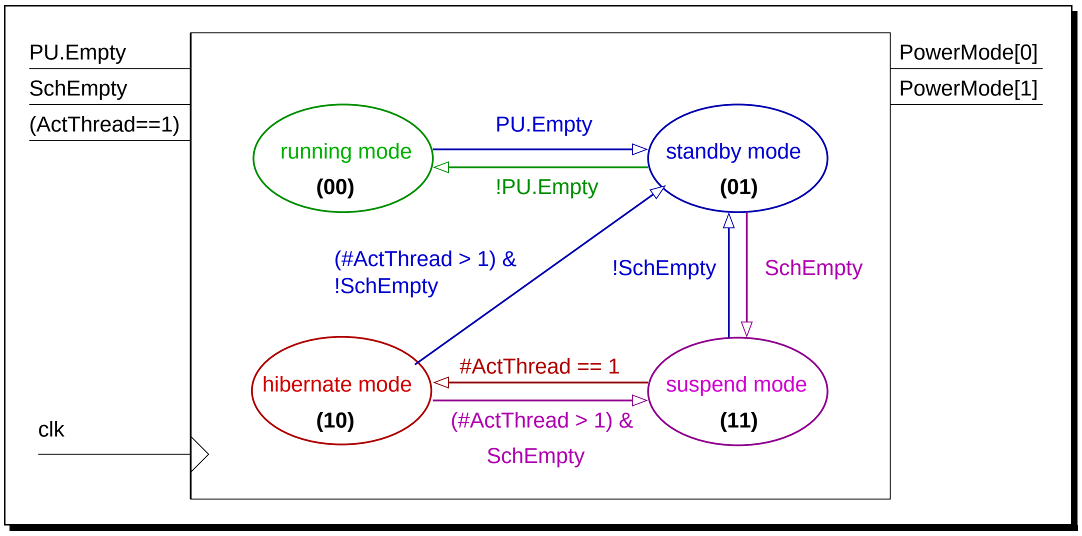


Figure 5.6: An overview of the power-mode generator.

5.4 Architectural Theoretical Investigation

This section presents a couple of investigations that demonstrate how some design choices satisfy the challenges arisen in §5.2.3. Real-time performance, design constraints, cache conflicts and access bottlenecks are left to be evaluated by benchmarking (see Chapter 8 for its evaluation result).

The structure of this section is as follows: relevant factors for maximising the utilisation of processing elements from the usage of multiple threads are first investigated (§5.4.1). The minimum incremental expense per thread is then revealed (§5.4.2).

5.4.1 The Utilisation of the Processing Elements

An investigation into the utilisation of the processing elements was carried out using queueing analysis [101]. The queue is a space-limited set of execution contexts (C) in the processing unit. The service stations of this queue are MulTEP's multiple processing elements (s). The inter-arrival time distribution from the queue and the service time distribution of each processing element are both exponential ($\mathcal{M}$) from Agarwal's multithreading study [100]. Based on these facts, the utilisation of the processing elements was analysed using the $\mathcal{M}/\mathcal{M}/s/C$ model as depicted in Figure 5.7.

The fraction of time each server is busy, i.e. the processing element utilisation (U), in $\mathcal{M}/\mathcal{M}/s/C$ is calculated by Equation 5.1 [101].

$$U = \frac{\sum_{n=0}^{C-1} \lambda p_n}{s\mu} \tag{5.1}$$

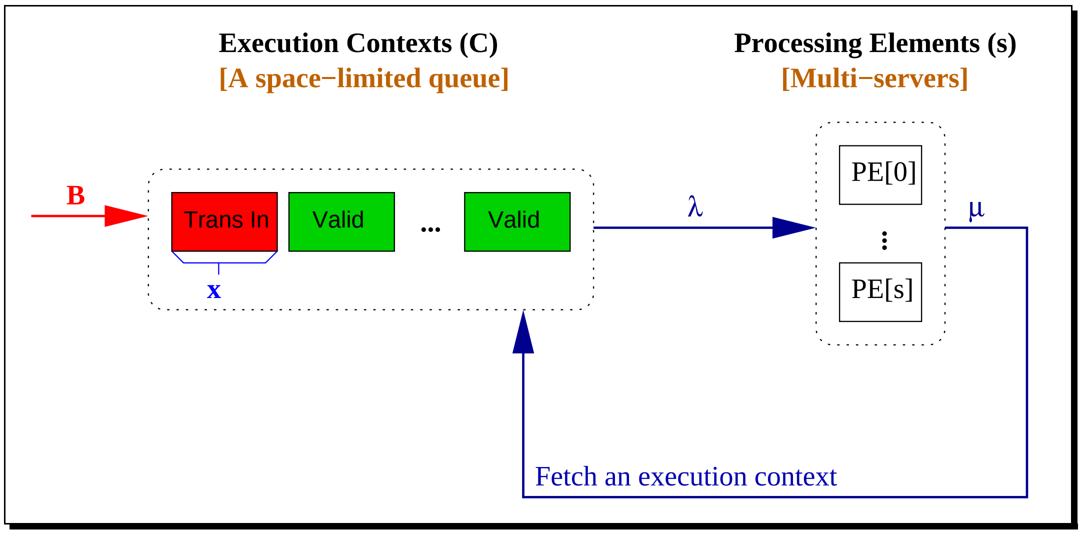


Figure 5.7: The model of the MulTEP processing unit ($\mathcal{M}/\mathcal{M}/s/C$).

$$
\begin{aligned}
\text{where,}\quad p_n &= \begin{cases} \frac{(\rho s)^n}{n!} p_0 & (n = 1, 2, ..., s-1) \\ \frac{\rho^n s^s}{s!} p_0 & (n = s, s+1, ..., C) \end{cases} \\
p_0 &= 1 - \sum_{n=1}^{C} p_n \\
\rho &= \frac{\lambda}{s\mu}
\end{aligned}
$$

To derive the processing element utilisation U, the arrival rate λ (i.e. the fetch rate) and the service rate μ (i.e. $\frac{1}{\text{run length}}$) were first identified. The fetch rate λ was calculated from the number of execution contexts (C), its context size (x), the input bandwidth (B) and the run length (μ^{-1}) [132] as the formula presented in Equation 5.2[6].

$$
\begin{aligned}
\text{The arrival rate } \lambda &= \text{The fetch rate} \\
\lambda &= \frac{F_r + F_t}{C} \\
&= \frac{(C-1)F + (1 \cdot i)F}{C} \\
&= \frac{(C-1)F + \frac{B}{x}F}{C} \\
&= \frac{(C-1+\frac{B}{x})\frac{t_s}{t}}{C} \\
&= \frac{(C-1+\frac{B}{x})\frac{\mu^{-1}}{\mu^{-1}+\frac{x}{B}}}{C} \qquad (5.2)
\end{aligned}
$$

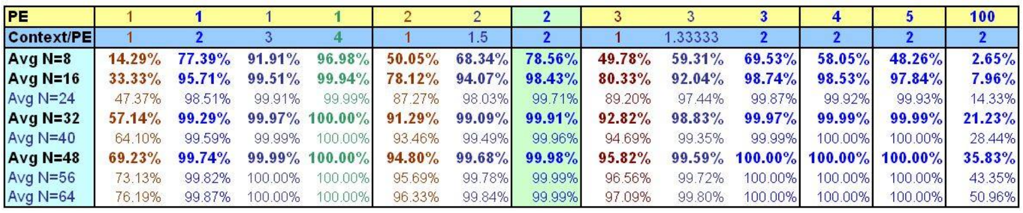
[6]It is assumed that the execution contexts are perfectly utilised.

where,	F_r	is the fetch rate of `Valid` contexts
	F_t	is the fetch rate of the `Trans In` context
	F	is the average fetch rate
	i	is the input rate of the `Trans In` context
	t_s	is the service time per context
	t	is the total consume time per context

To calculate λ for MulTEP, the input bandwidth B is 128 bits/cycle (see Section 6.2). The context size x can be up to 64x32 bits in the worst case when contexts are transferred between the different thread groups.

The service rate μ depends on the mean time to complete the service μ^{-1}. According to the study in [133], a service block larger than 48 instructions per window yield almost no performance benefit. Hence, my investigation focuses on a run length that covers this range and a little bit further (from 8 to 64 instructions per service).

Table 5.6 illustrated the outcomes when the model was analysed with the different number of processing elements (p), the number of contexts (C) and the different sizes of service block (μ^{-1}). These results show that when the service time is shorter than the the context-switching time, the utilisation U is degraded.



PE	1	1	1	1	2	2	2	3	3	3	4	5	100
Context/PE	1	2	3	4	1	1.5	2	1	1.33333	2	2	2	2
Avg N=8	14.29%	77.39%	91.91%	96.98%	50.05%	68.34%	78.56%	49.78%	59.31%	69.53%	58.05%	48.26%	2.65%
Avg N=16	33.33%	95.71%	99.51%	99.94%	78.12%	94.07%	98.43%	80.33%	92.04%	98.74%	98.53%	97.84%	7.96%
Avg N=24	47.37%	98.51%	99.91%	99.99%	87.27%	98.03%	99.71%	89.20%	97.44%	99.87%	99.92%	99.93%	14.33%
Avg N=32	57.14%	99.29%	99.97%	100.00%	91.29%	99.09%	99.91%	92.82%	98.83%	99.97%	99.99%	99.99%	21.23%
Avg N=40	64.10%	99.59%	99.99%	100.00%	93.46%	99.49%	99.96%	94.69%	99.35%	99.99%	100.00%	100.00%	28.44%
Avg N=48	69.23%	99.74%	99.99%	100.00%	94.80%	99.68%	99.98%	95.82%	99.59%	100.00%	100.00%	100.00%	35.83%
Avg N=56	73.13%	99.82%	100.00%	100.00%	95.69%	99.78%	99.99%	96.56%	99.72%	100.00%	100.00%	100.00%	43.35%
Avg N=64	76.19%	99.87%	100.00%	100.00%	96.33%	99.84%	99.99%	97.09%	99.80%	100.00%	100.00%	100.00%	50.96%

Table 5.6: Utilisation U with the different value of p, C and μ^{-1}.

Figure 5.8 depicts the utilisation trends when the mean service time μ^{-1} is increasing with different number of contexts per processing element ($\frac{C}{p}$). Models with a higher number of contexts per processing element ($\frac{C}{p}$) yield better processing element utilisation. However, a high ratio $\frac{C}{p}$ could waste circuits if applications provide an insufficient number of threads to utilise them. Furthermore, the ratio $\frac{C}{p}$ needs to be small to reduce physical complexity in order to result in a cost-effective performance (see §3.4.4).

For cost-effective performance, models with $\frac{C}{p} = 2$ seems to be the suitable choice, especially when the run length is greater than or equal to the context-switching period (16 instructions) where the utilisation is greater than 95%. Among the members of the $\frac{C}{p} = 2$ group, the model with two processing elements provides the highest average utilisation as illustrated in Figure 5.9. Thus, this model is used in the MulTEP architecture.

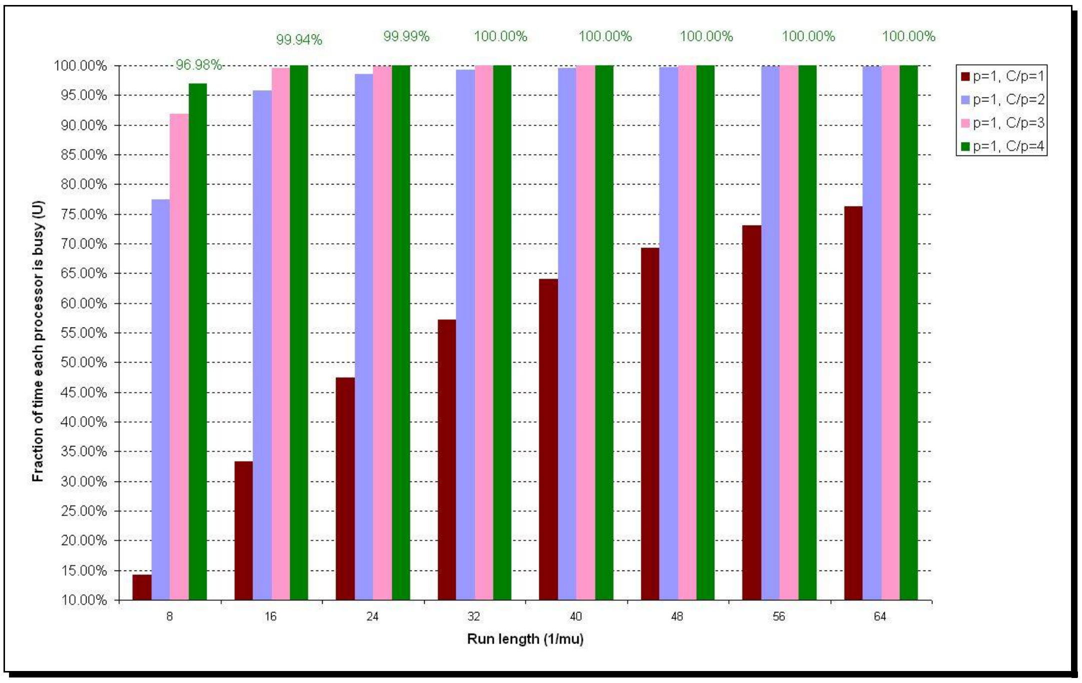


Figure 5.8: Utilisation U with the different value of C and μ^{-1}.

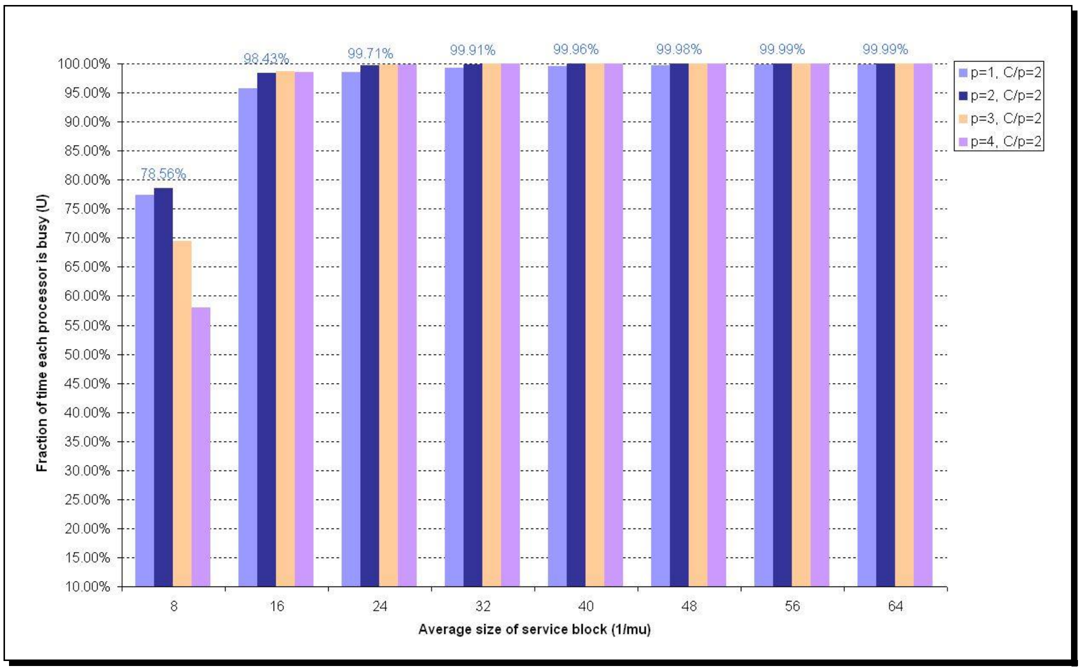


Figure 5.9: Processor utilisation with the different value of p and μ^{-1}.

5.4.2 Incremental Cost per Thread

In conventional multithreading architectures, a simple approach to provide hardware support for multithreading is to duplicate hardware resources to store execution context of additional threads. The register file grows linearly with the number of thread contexts (see Chapter 3 for more details). As such, the incremental hardware-cost per thread is the additional *Program Counter* (PC), *Register set* (Rset) and *Context Pointer* (CP) to be latched in each pipeline stage. Figure 5.10 depicts an additional cost in pink for one processing element.

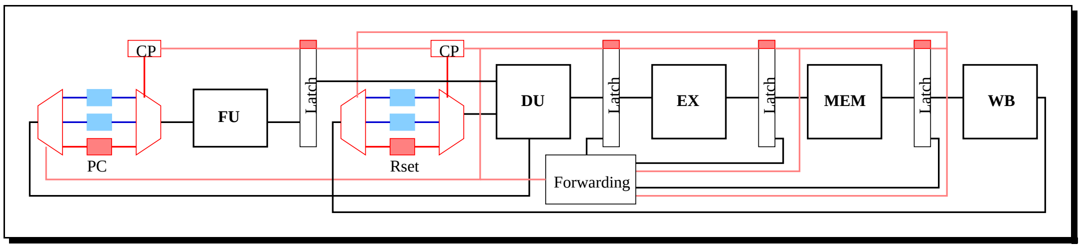


Figure 5.10: Incremental parts per one thread in the processing element.

Figure 5.11 illustrates the incremental cost per thread in MulTEP. MulTEP offers an alternative approach by having only a slightly larger register file and fetch unit but providing a closely-coupled activation-frame cache to store inactive contexts. If the activation-frame cache overfills, contexts will be spilled to the rest of the memory hierarchy. The pink parts presents units that need to be altered if the architecture has to support a number of threads greater than 2^{16}. Otherwise, there is no incremental hardware-cost per thread from this architecture.

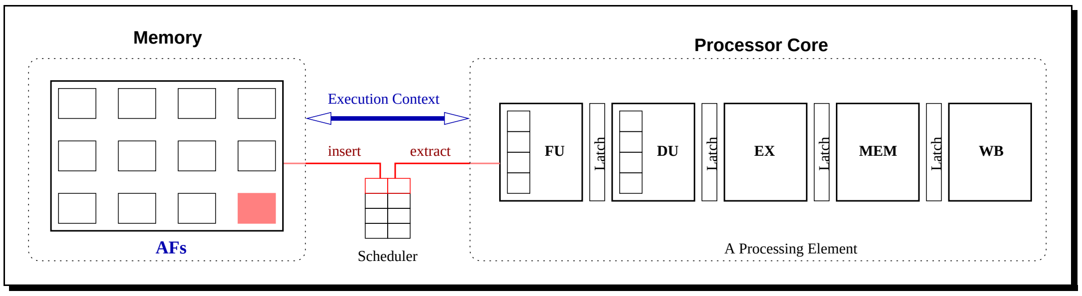


Figure 5.11: The affected part after adding one thread in MulTEP.

5.5 Summary

My research focuses on designing a high performance multithreading architecture for embedded systems. The main objectives of this research are:

- To improve performance by context switching rather than stalling the pipeline
- To schedule threads according to their real-time requirements

- To minimise incremental hardware costs

Thread level parallelism, real-time response and low-power consumption become a set of crucial design challenges. To satisfy these challenges, MulTEP has been implemented. Its processing element is developed from RISC DLX with extensions to support the MIPS IV instruction set architecture. Predicated r-type instructions are added to minimise a number of context-switching indicators, i.e. branch instructions. Furthermore, four multithreading instructions (i.e. `spawn`, `wait`, `switch` and `stop`) are introduced to progress a thread through its life cycle.

MulTEP is modelled with two processing elements and four execution contexts for utilisation and cost-effectiveness. Thread scheduling is priority based using the modified tagged up/down priority queue to meet real-time requirements. The scheduler incorporates dynamic-priority and time-slicing policies to prevent thread starvation. Thread synchronisation are handled by a matching-store mechanism. The synchroniser is facilitated by message passing via a store instruction.

The architecture is capable of executing a large and flexible number of threads in accordance with their priorities. The execution context is that of the program counter, the 31x32 integer registers and the 32x32-floating point registers. Though the number of register sets is fixed to four, excess execution contexts can be spilled out to the memory in the form of an activation frame, which is cached. This scheme results in a minimal incremental hardware-cost per thread.

Low-power dissipation is supported by eliminating non-deterministic mechanisms such as a branch prediction or a speculative instruction scheduling. Power operating modes (i.e. *running*, *standby*, *suspend* and *hibernate*) are provided to allow an implementer to supply suitable power management circuits.

Chapter 6

Hardware Architecture

Two roads diverged in a wood, and I -
I took the one less travelled by,
and that has made all the difference.

Robert Frost

6.1 Introduction

The MulTEP architecture has been designed to provide thread level parallelism which is suitable for embedded environments. In accordance with my architectural decisions in Chapter 5, MulTEP consists of four components connected to one another as illustrated in Figure 6.1. A *Processing Unit* (PU) contains two processing elements. A *Multithreading Service Unit* (MSU) provides multithreading operations for synchronisation and scheduling mechanisms. A *Load-Store Unit* (LSU) is designated to organise the load and store operations. A *Memory Management Unit* (MMU) handles data, instructions and I/O transactions.

The structure of this chapter is as follows: Section 6.2 illustrates the design strategies and the supported mechanisms of the processing unit. Sections 6.3 and 6.4 describe the multithreading service unit and the load-store unit, respectively. Section 6.5 shows the techniques used in the memory management unit. Section 6.6 summarises the MulTEP hardware architecture.

6.2 Processing Unit

The project focuses on implementing hardware multithreading for the embedded systems. Multiple threads often compete for shared computational resources. Thus, support

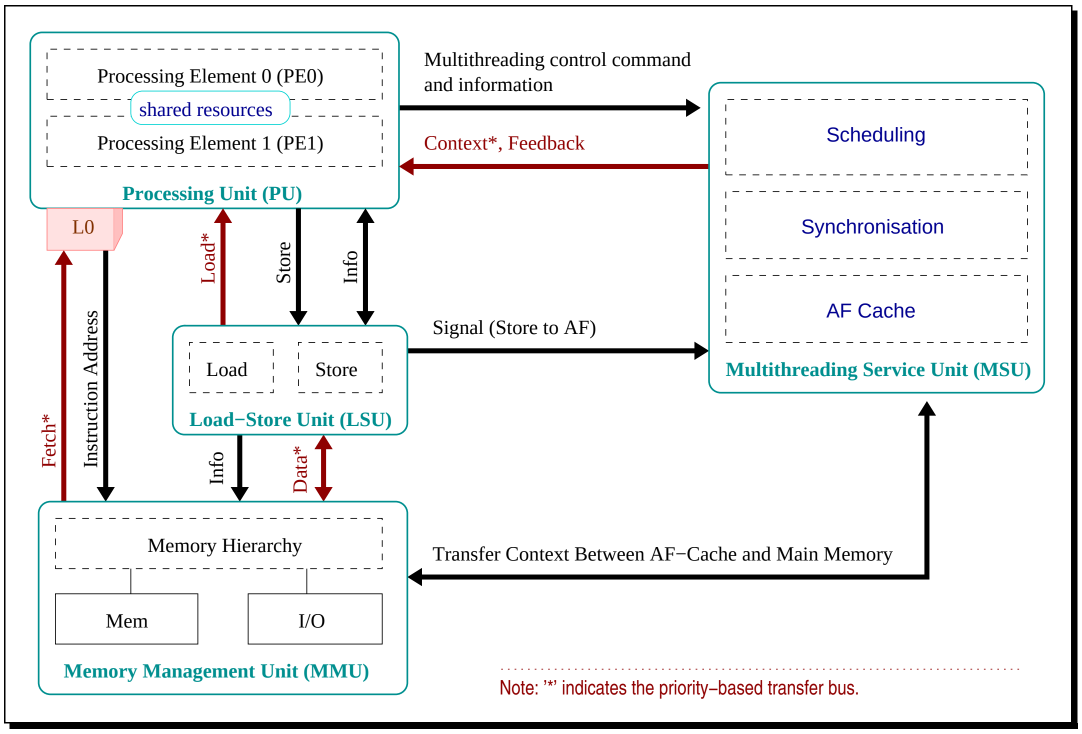

Processing Element 0 (PE0)
shared resources
Processing Element 1 (PE1)
Processing Unit (PU)
Multithreading control command and information
Context*, Feedback
Scheduling
Synchronisation
AF Cache
Multithreading Service Unit (MSU)
L0
Load*
Store
Info
Signal (Store to AF)
Load
Store
Load−Store Unit (LSU)
Fetch*
Instruction Address
Info
Data*
Memory Hierarchy
Mem
I/O
Memory Management Unit (MMU)
Transfer Context Between AF−Cache and Main Memory
Note: '*' indicates the priority−based transfer bus.

Figure 6.1: The MulTEP hardware architecture.

hardware for thread collaboration and competition need to be included in a core computational component. Because of this, the *Processing Unit* (PU) of MulTEP is designed for multithreading by employing a number of potential techniques described in Chapter 3 .

According to the design decisions in Chapter 5, an abstract hardware block of the processing unit is illustrated in Figure 6.2[1]. The processing unit is capable of handling up to four execution contexts from different thread streams by using two processing elements (see §5.4.1). The unit supports thread competition and collaboration by using message passing through the load-store unit where the destination of the message is identified with a thread ID (see Section 6.4).

Contexts are switched when a context-switch instruction is received or when the execution of a thread needs to be stalled because a register is still waiting for data from the memory. The underlying mechanisms of the processing unit incorporate pre-fetching, pre-loading and colour-tagging techniques to allow execution contexts to switch without unnecessary overhead.

For pre-fetching, the fetch stage is associated with a level-0 instruction cache and a block detector to pre-fetch instructions for at least four different threads. For pre-loading, the unit offers four register sets which contain the four highest-priority execution contexts extracted from all runnable threads in the multithreading service unit. The pre-loading process is supported by an outside-PU context-switching mechanism. The operation of each thread is tagged with a register set's identifier, i.e. a colour identity, which enables

[1]The processing unit is depicted with a complete illustration of detailed I/O ports and an interconnections to the other units.

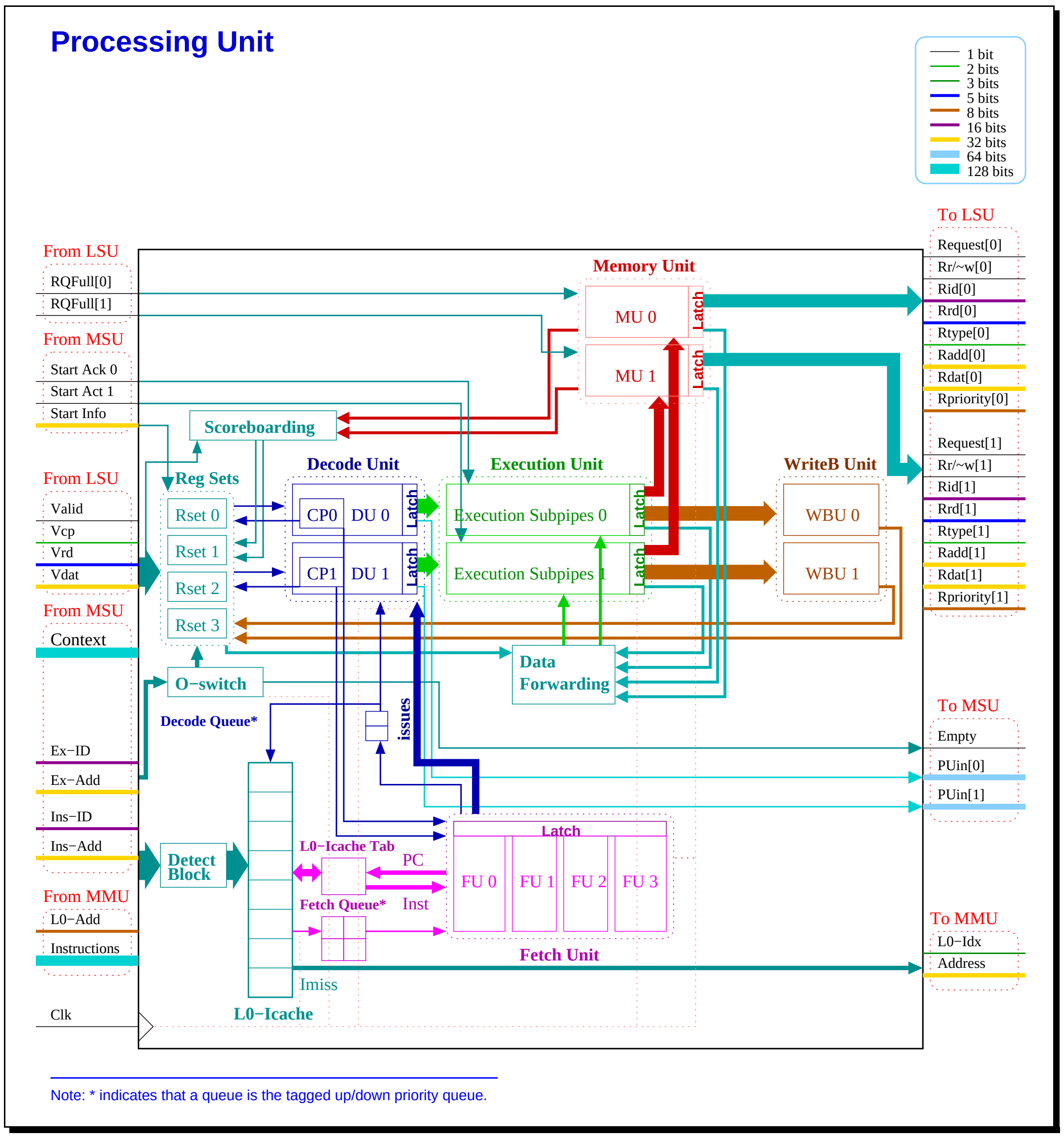


Figure 6.2: An overview of the processing unit.

the write back stage and data-forwarding unit to select a destination with regards to its colour identity.

Each processing element in the processing unit is an enhanced form of the classic RISC *DeLuX* (DLX) pipeline [24] which has 5 pipeline stages: fetch, decode, execute, memory access and write back. Speculative execution mechanisms are eliminated to minimise power consumption (see design motivation in §5.4.1).

The remainder of this section presents detailed design of each component in the processing unit. The level-0 instruction cache is first explained (§6.2.1), followed by the fetch mechanism (§6.2.2), context-switch decoder (§6.2.3), modification of the execution unit (§6.2.4) and write back which uses colour tagging (§6.2.5).

6.2.1 Level-0 Instruction Cache

Multiple instruction streams need to be fetched from the memory in order to exploit thread level parallelism. Unfortunately, pre-fetching multiple thread streams often results in competition for resources. To alleviate the competition and further accelerate multi-threaded operations, a *Level-0 Instruction cache* (L0-Icache) is introduced. The level-0 instruction cache pre-fetches instructions in accordance with the scheduling commands from the multithreading service unit.

Time-slicing preempts thread execution. This prevents the highest priority thread from dominating the pipeline. As such, it helps the scheduling to eliminate starvation that may occur with low-priority threads. Each cache line of the L0 I-cache is designed to support up to 32 instructions. If the run length is more than 32 instructions further cache lines will be used.

In order to support instruction pre-fetching, the L0 I-cache consists of $2n$ cache lines where n represents the number of execution contexts. This number is provided for zero-cycle context switches because n cache lines can hold instructions for n executing thread streams while the other n cache lines are used to pre-fetch the next instruction blocks. Hence, eight cache lines are provided in the L0 I-cache to support four execution contexts in MulTEP.

To make use of the scheduling information from the multithreading service unit, four cache-line states (e.g `None`, `InQ`, `Next` and `Ready`) are provided with different priorities shown in Figure 6.3 and Table 6.1.

State	Priority	Detail
`None`	00	Hold no instructions (owner does not exist in the PU)
`InQ`	01	Hold instructions of the inserted thread in a ready queue
`Next`	10	Hold a next pre-fetched block of the thread in the PU
`Ready`	11	Hold a set of the extracted thread, reserved for execution

Table 6.1: The states of a cache block in the level-0 instruction cache.

The lowest priority cache line is replaced if and only if the replacement cache line is

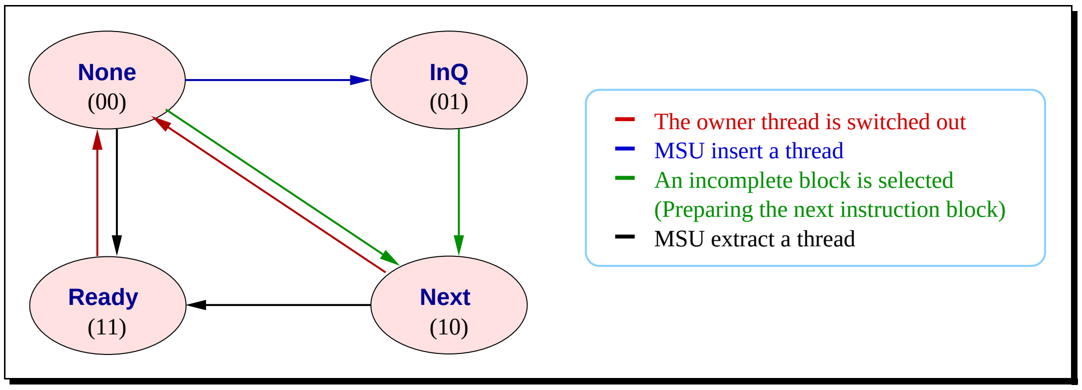

None
(00)
InQ
(01)
Ready
(11)
Next
(10)
The owner thread is switched out
MSU insert a thread
An incomplete block is selected
(Preparing the next instruction block)
MSU extract a thread

Figure 6.3: A state diagram of each level-0 instruction cache line.

of a higher priority[2].

Figure 6.4 illustrates how the pre-fetching mechanism alters a cache-line state. The first cache line has held instructions starting with `Address1` since time n. Its `InQ` state indicates that its owner, i.e. Thread 1, has already been inserted into a scheduler of the multithreading service unit[3] (see more details in Section 6.3). At time $n+2$, the execution context of Thread 1 is extracted to register set 1 of the processing unit. The state of the first cache line is then changed to `Ready`.

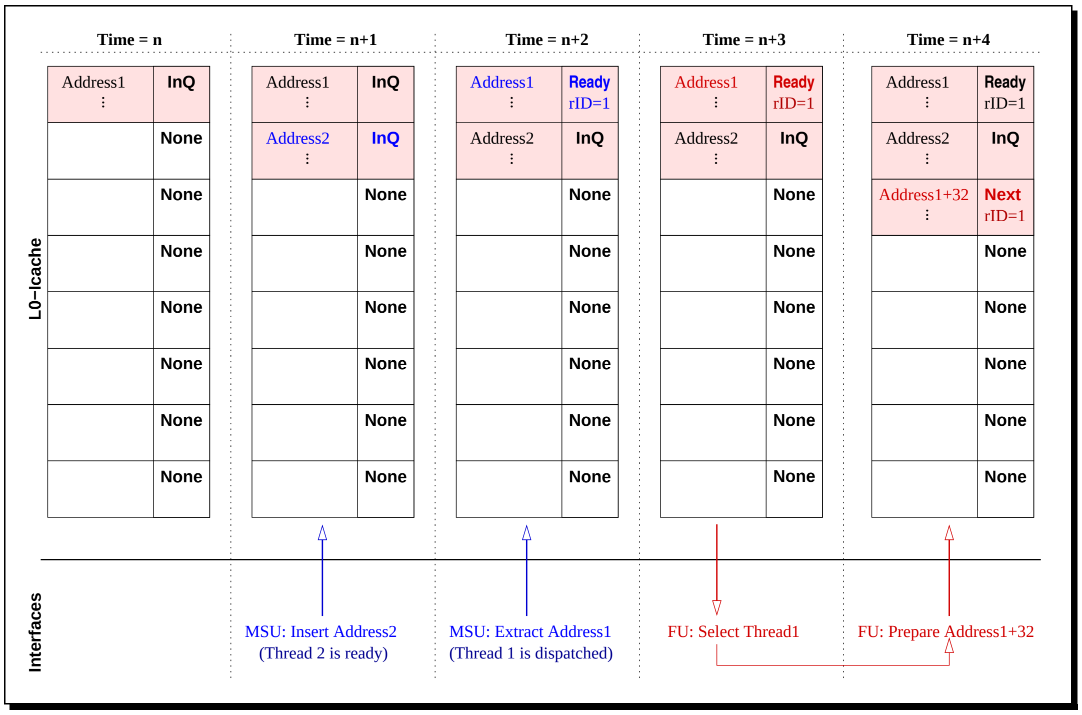


Figure 6.4: Pre-fetching operation on the level-0 instruction cache.

An incomplete instruction block is indicated by a block detector when a `branch`, a `jump` or a `wait` opcode could not be found in the pre-fetched instructions. In the case that

[2]The operation is conducted with the same mechanism as the LRU replacement policy [26].
[3]The operation to set a cache line to `InQ` state is also shown in times $n+1$ of Figure 6.4

the current block is incomplete, the rest of the instructions are required to be pre-fetched. At the next clock cycle, after an incomplete block is selected by the processing element, a new block starting with the next location (Address1+block size) is pre-fetched[4] and the cache-line state is set to `Next`.

Figure 6.5 depicts an access procedure in the L0 I-cache. The upper 25-bit tag field[5] is first compared to every tag field in the L0 cache table. A matched entry with a valid status provides an index to the corresponding cache line. The 5-bit offset is then used to locate the required instruction.

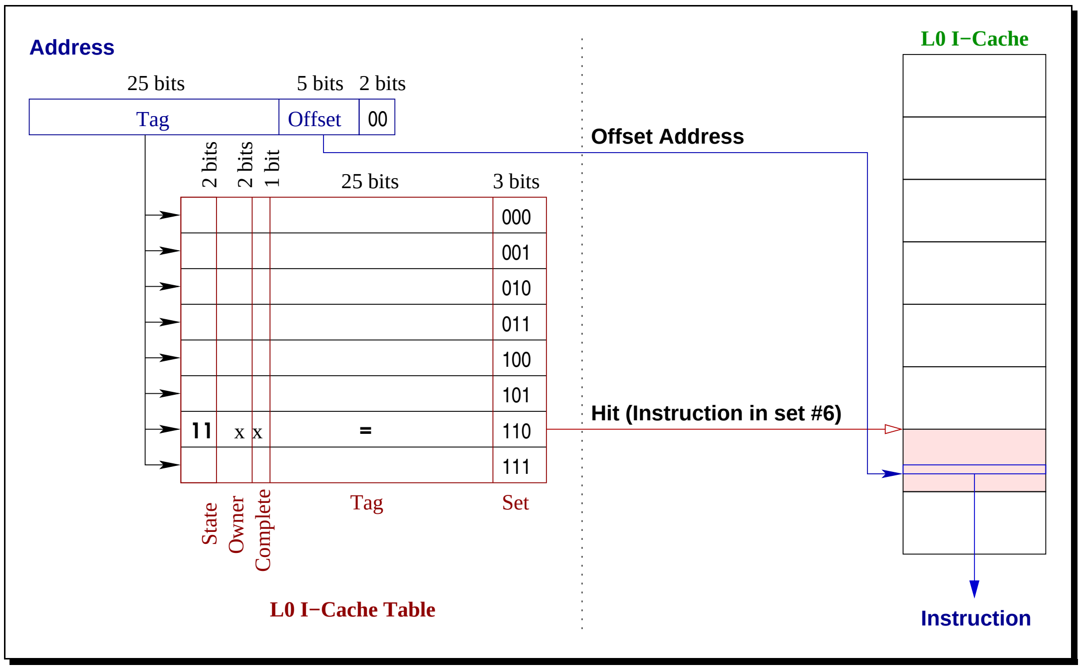


Figure 6.5: The access procedure in the level-0 instruction cache.

6.2.2 Fetching Mechanism

As reviewed in Chapter 2, the operation of a single thread stream often introduces a number of bubbles caused by pipeline conflicts or memory access latency. Chapter 3 demonstrates that these wasted bubbles can be replaced by alternative thread streams. To assist this, multiple thread streams need to be pre-fetched.

In MulTEP, two processing elements share four fetch elements. This requires a special fetching strategy to fill all fetch units in order to switch contexts without any overhead. As mentioned in §6.2.1, pre-fetched instructions of the extracted threads are prepared in the 8-set L0-cache adjacent to the fetch unit. Therefore, a fetch queue is provided to sort the extracted threads' start addresses in accordance with their priorities.

[4]Instruction pre-fetching is conducted if the replacement policy allows this to happen.

[5]25 bits are the rest of the instruction address after its 32 bits is subtracted by the 5-bit offset size (32 instructions per cache line) and the 2-bit instruction size (4 bytes).

The fetch queue is implemented with a tagged up/down priority queue [8] because it is capable of sorting prioritised data within a single clock cycle. In the priority queue, only two LR elements are required to buffer four addresses since the other four addresses of the eight maximum cache lines of the L0-Icache will be held in the fetch unit. The input key of the priority queue is a 8-bit thread priority and the input data is a 30-bit start address[6]. The sorting policy is “extract maximum”[7].

In the example presented in Figure 6.6, three fetch units contain executable threads whose addresses start from A1, A2 and A3. The first two cache lines are limited by the boundary of 32 instructions and still need more instructions. The remaining cache lines starting with A1+32 and A2+32 have already been pre-fetched into cache lines 5 and 6[8]. At time n, one fetch element is available. The fetch unit immediately extracts a L0 cache-line pointer whose priority is the highest (i.e. cache line 4) from the fetch queue.

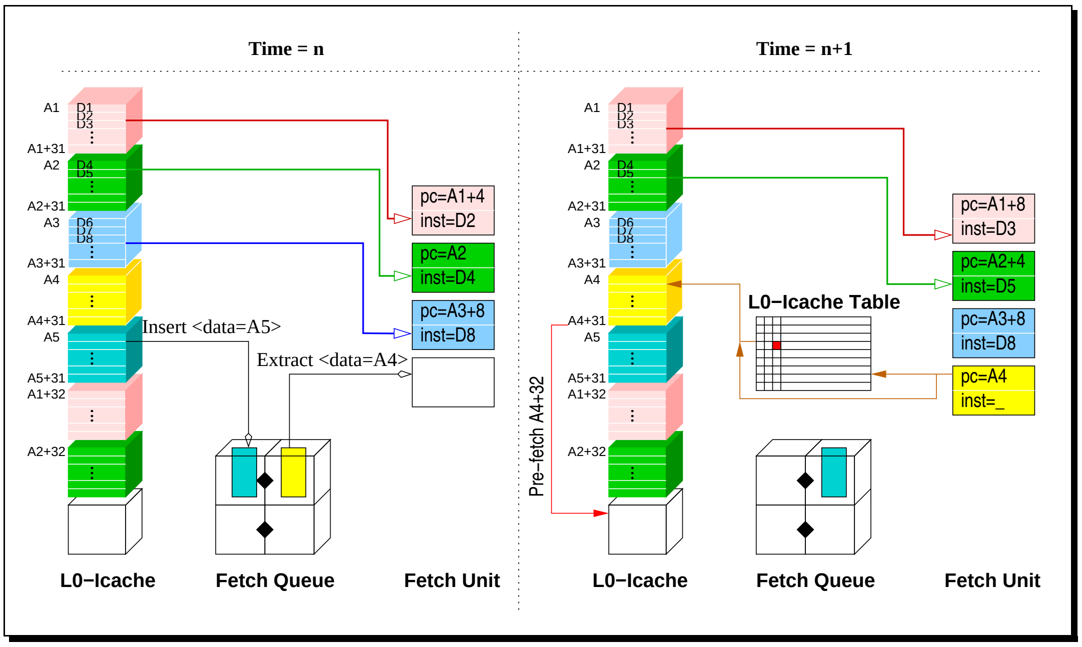


Figure 6.6: Fetching and pre-fetching operation by the fetch unit.

At time $n+1$, the highest prioritised address (i.e. A4) is loaded into the available fetch element. The address is simultaneously used to fetch an instruction from the L0-Icache via the L0-Icache table. The instruction will be returned to the fetched element in the next clock cycle. During the address resolution, if the complete flag in the L0-Icache table is clear[9], the next block of instructions will be pre-fetched to another appropriate cache line in the L0-Icache in the next clock cycle. The pre-fetching address for that appropriate cache line starts with the address of the incomplete cache line plus a block size (A4+32). Consequently, the priority of the continuous cache line is set to `Next`.

[6]Start address is only 30 bits because the lower 2 bits are always 0.

[7]With the extract maximum key policy, the unused key value is set to 0.

[8]Cache lines 5 and 6 are available because their previous instructions are incomplete and those incomplete cache lines have already been selected by the processing unit.

[9]An complete flag indicates that the block is complete. Hence, if the flag is valid, no further instructions need to be loaded.

6.2.3 Context-switch Decoder

To maximise benefits of multithreading, idle units should be detected as early as possible and filled with alternative streams of instructions. Most of the time, bubbles are generally found by the decode stage because it is capable of detecting most of the data, control and structural conflicts. Therefore, the context switch decision is made in this stage of the pipeline. The decoder also understands special multithreading instructions which indicate context switch signals (see §5.3.3).

In general, the decoder translates an opcode to its corresponding control signals[10] and extracts operands according to the field allocation specified by the type of the opcode (see §5.3.3). Decoding the opcode allows the unit to detect `switch`, `wait` and `stop`, which are three special opcodes that indicate context-switches. Decoding the operands allows the unit to detect data-conflict bubbles early (e.g the scoreboard indicates that a data for one of its registers is being loaded). Therefore, the execution can be switched to another thread instead of issuing a bubble.

The decoder is also capable of interpreting stall signals from the other units, such as an instruction miss from the fetch unit. These signals are used to make context switch decisions.

There are two levels of context-switching operations. The first operation is an inside-PU context switch which is provided to switch an execution context without any overhead. The second operation is an outside-PU context switch which is provided to allow the system to support a flexible number of threads. Operations of these two context switches are described as follows:

1. **Inside-PU context switch**

   The inside-PU context switch is simply done by changing a *Context Pointer* (`CP`) of the available decode element to point to an alternative context in the processing unit. This alternative context should have the highest priority and has not been selected by the other decode element.

   In MulTEP, the context pointer is obtained from the decode queue, which is a two-entry tagged up/down priority queue. In the sorting queue, the package's key is an 8-bit thread priority and the package's data is a 2-bit register index. The extracted register index from the queue is also used to signal instruction pre-fetch for the L0-Icache.

2. **Outside-PU context switch**

   An *outside-PU* context switch is provided to exchange an execution context in a register set with its activation frame held in the multithreading service unit. The operation combines the following two methods:

   - *Eliminate unnecessary transferring traffic*

[10]Simple operations such as load/store, multithreaded, logical and arithmetic instructions are directly translated, leaving more complicated opcodes such as multiply, divide and floating point to be used as an index to their microcode [24].

The underlying context-switching method is similar to Dansoft's nanothreading approach [129] which reduces context transferring traffic when contexts within the same thread group need to be exchanged. In such a case[11], the context transaction is reduced by transferring only the priority information, the program counter and the parametric registers[12] (see detail in §5.3.2). Therefore, only 26 words are transferred instead of the whole 64 words. Switching the full execution context occurs when the exchange of threads is required by two different thread groups, or when an incoming context's start PC (field `K1`) is different from the nanothread pointer (field `K1`).

- *Associate context status with each execution context*

  Execution contexts in both the register sets and the program counters can be spilled out as activation frames to the multithreading service unit. Each register set and its corresponding program counter are usually shared by more than one activation frame. Therefore, it is necessary to harmonise multiple execution contexts on each shared resource. As a result, each execution context is associated with a suitable context status indicator.

  In general, the status typically represents whether a context is valid, invalid, being used or being transferred. The transfer states reflect the need to transfer contexts over multiple clock cycles due to bandwidth limitations between the register set and the activation frame. Figure 6.7 presents a state diagram for context transferring. In total, there are eight context states represented by a 3-bit state value. The initialised state is the `Invalid` status.

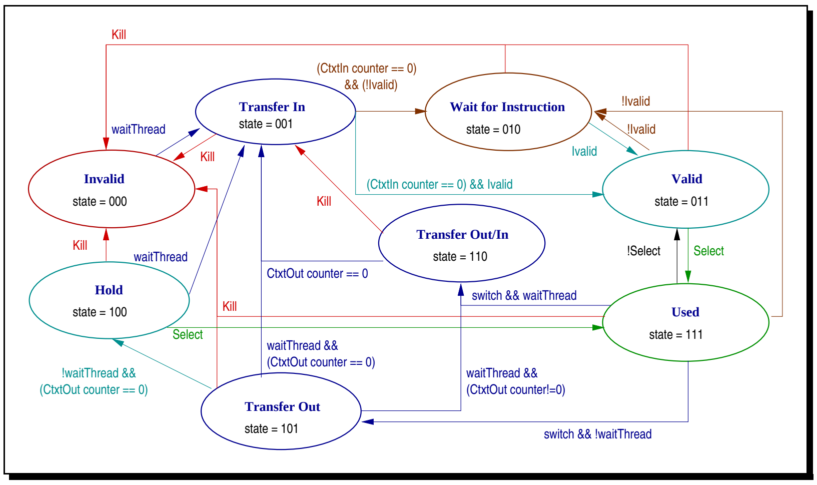


Figure 6.7: The 3-bit state diagram of the context status.

  An *Invalid* context is activated by a `waitThread` signal when there is a waiting thread requiring its context to be transferred in. MulTEP dedicates two separate buses for transferring execution contexts in and out. Each execution

[11]The exchange within the same group occurs when the upper 16 bits of `$at` from the register file equals the upper 16 bits of field `K1` in the activation frame

[12]The parametric registers are ranging from register `$v0` to register `$t9`.

context bus is 128-bit wide to transfer 4 words per cycle[13]. Therefore, the number of cycles (i.e. 7 or 16 cycles) is required for outside-PU context switch.

The `CtxtIn` counter starts with the number of cycles required to undertake the transfer. The context remains in the `Transfer In` state until this counter reaches zero. After that, the context status is changed to be either `Valid` (if an instruction pointed to by the program counter is available in the fetch unit) or `Wait for Instruction` (to be validated when the instruction is available).

A `Valid` context can be selected by a decoder and changed to status `Used`. The context leaves the `Used` status when:

– An I-cache miss is detected.

– The context is deselected.

– The execution of the thread is completed.

– An outside-PU context-switching is activated.

If the context needs to be transferred out, the status is changed to either `Transfer Out` if no thread waits to be transferred in, or `Transfer Out/In` when contexts in the register set are being transferred in and out simultaneously. When the transferring out process is completed and there is no waiting thread to be transferred in, the status is changed to `Hold`.

The context is held ready to be activated later or replaced when another thread in the same thread group needs to be transferred in. The context is capable of resetting back to `Invalid` once a kill signal occurs, except during the `Transfer Out/In` process because the other thread is being transferred in.

6.2.4 Execution Units

The execution unit consists of four subpipelines to support multi-cycle instructions (see Chapter 2). The unit comprises an integer pipeline, a *Floating-Point* (FP) adder subpipeline, an FP/integer multiplier subpipeline and an FP/integer divider subpipeline as presented in Figure 6.8.

The execution of a single-precision floating-point multiplier is based on a Verilog 6-stage confluence FP/integer multiplier core introduced by the Opencore initiative [135]. The FP adder is a standard RISC DLX integer ALU [24]. A single-precision [136] FP/integer divider emulates the 15 levels of pipeline developed by the DFPDIV development of the Digital Core Design [137].

Data feed-forwarding for all subpipelines is similar to the mechanism in DLX [24]. A slight difference is its support for multithreading where additional register set ownership

[13]The size of context bus is designed to be the same as the size of data bus in commercial embedded processors [37, 134].

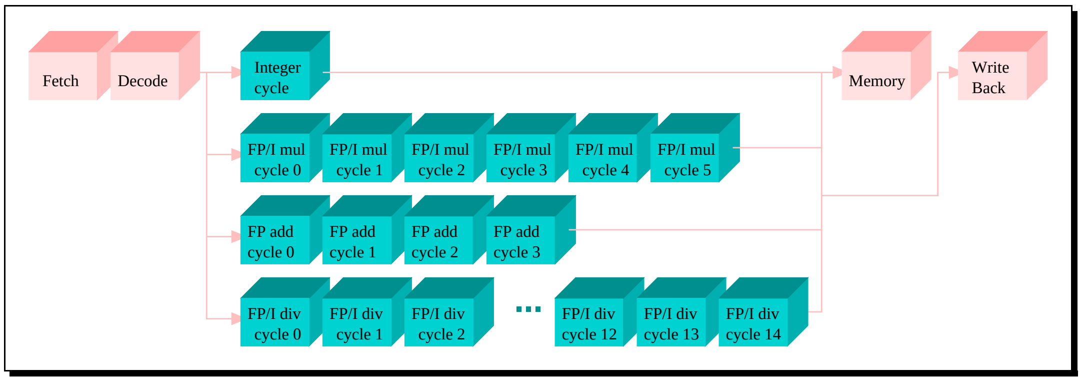


Figure 6.8: The extended execution stage of one processing element.

is required on top of the register index. This requirement only results in two additional bits per data-feed forwarding package[14].

Unexpected exceptions from the subpipelines are handled by imprecise roll-back techniques [24, 26, 33, 34]. Whilst MulTEP takes advantage of multiple instruction issue, speculative execution is not used since speculative techniques consume extra power and reduce real-time predictability. Furthermore, performance gained from thread level parallelism is believed to compensate the need for instruction level parallelism extraction using speculation.

6.2.5 Colour-tagged Write Back

A thread is tagged with an index into its register set called a *colour*. With this technique, each colour-tagged operation flows in a processing element with a reference to its corresponding register set. Figure 6.9 illustrates the flow of multiple threads through two integer subpipelines.

The figure shows that the blue thread stream, executed in the PE1, is switched to the PE0 at time n without any overhead. Instructions are processed independently in accordance with its colour supported by data forwarding and the write-back stage. A data feed-forwarding unit is shared between the two pipelines. This unit is necessary for zero-cycle context switch support in the execution stage (i.e. an integer subpipeline) as illustrated at times n+1 and n+2.

The scoreboard prevents data hazards which may be caused by a duplicate reference to the same register[15]. This allows the write-back stage to be independent. Thus, up to eight write requests[16] may coincidently arrive at the same register set. However, a register set with 8 write ports is impractical. Thus, the design uses a traditional 2 write ports with 4 buffers for each subpipeline[17]. All write-back requests are queued and granted in

[14]Seven bits are required for register identification (two bits is used to identify the register set and five bits is used to identify the register).

[15]The issuance of an instruction whose register is invalid is not permitted.

[16]From 4 subpipelines of 2 processing elements

[17]The pipeline will be stalled if the queue is full.

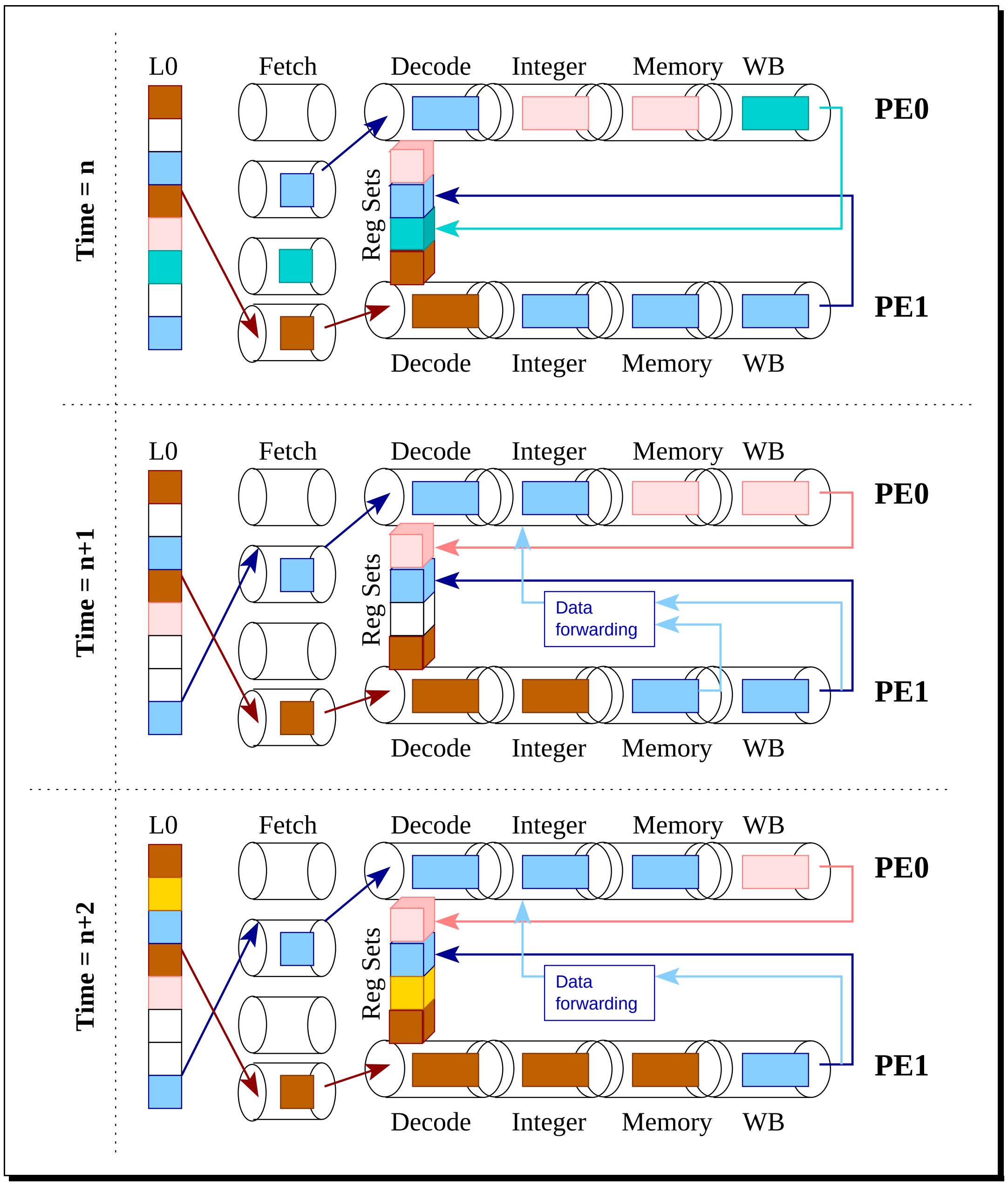


Figure 6.9: The processing unit to support zero context-switching overhead.

a round robin order.

With this colour-tagging scheme, if sufficient[18] execution contexts have already been pre-loaded and sufficient instructions have already been pre-fetched, the processing unit is capable of switching contexts without any overhead.

6.3 Multithreading Service Unit

As discussed in Chapter 1 and §5.2.3, MulTEP aims to be capable of simultaneously handling a large and flexible number of threads. Nevertheless, the processing core is designed to support only a fixed number of threads, i.e. 4 execution contexts, in order to minimise both context switch delay and implementation size. Therefore, excess threads need to be handled by an additional mechanism. As a result, in order to support this requirement, I decided to introduce a *Multithreading Service Unit* (MSU) into the MulTEP architecture.

In accordance with the proposed strategies in Section 5.3, the multithreading service unit holds excess execution contexts in the form of *Activation Frames* (AFs) in a local fully-associative activation-frame cache. The size of the activation-frame cache does not limit the number of execution contexts because they are extensible to the main memory.

Activation frames preserve the execution contexts of both excess *runnable* threads and *non-runnable* threads. The multithreading service unit schedules runnable threads with a tagged up/down priority queue (see scheduling details in §5.3.6). The unit synchronises non-runnable threads by using a matching-store mechanism (see synchronisation details in §5.3.5).

The multithreading service unit additionally provides activation-frame allocation and deallocation mechanisms to facilitate thread creation and thread termination in accordance with messages received from the processing unit. Figure 6.10 presents how these two operations influence the allocation of activation frames in the activation-frame cache. Empty activation frames are held in a linked list. The head of the list is pointed to by an `EmptyH` pointer, which is available for thread creation (see §6.3.2). The tail of the list is pointed to by an `EmptyT` pointer, which is available for thread termination (see §6.3.5).

Underlying mechanisms inside the multithreading service unit are presented in Figure 6.11. The following sections introduce the details of these mechanisms starting with the request arbiter (§6.3.1). Next, four mechanisms to provide multithreading services, namely a spawning mechanism (§6.3.2), a synchronising mechanism (§6.3.3), a switching mechanism (§6.3.4) and a stopping mechanism (§6.3.5), are described.

[18]Sufficient execution contexts occur when the number of valid execution context exceeds the number of processing elements.

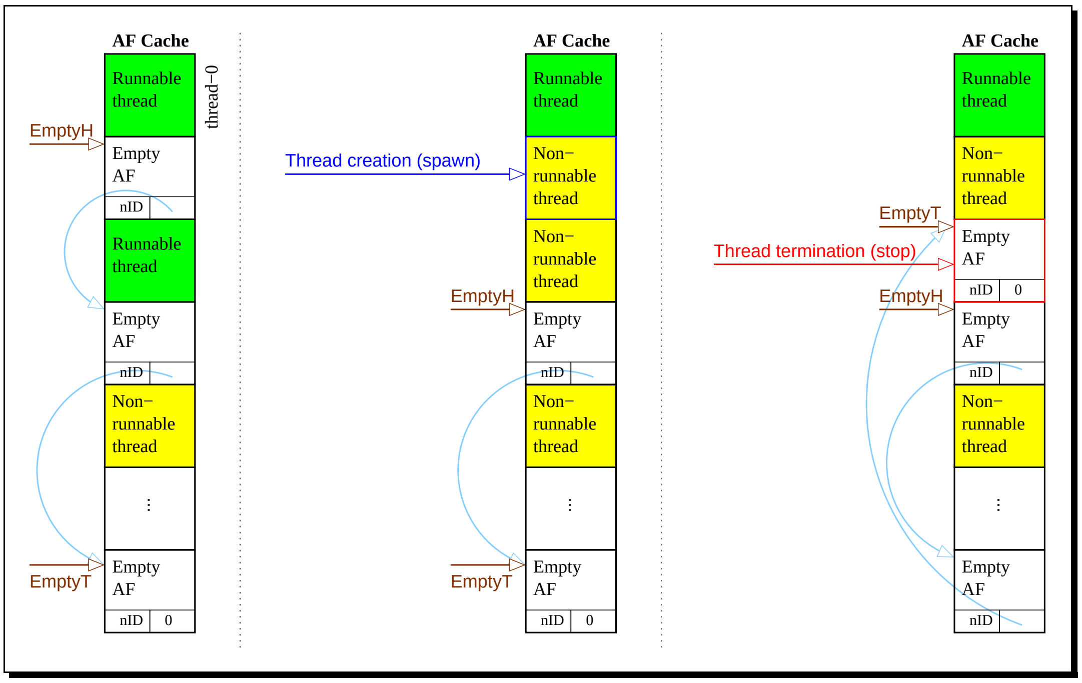


Figure 6.10: Thread creation and termination in the activation-frame cache.

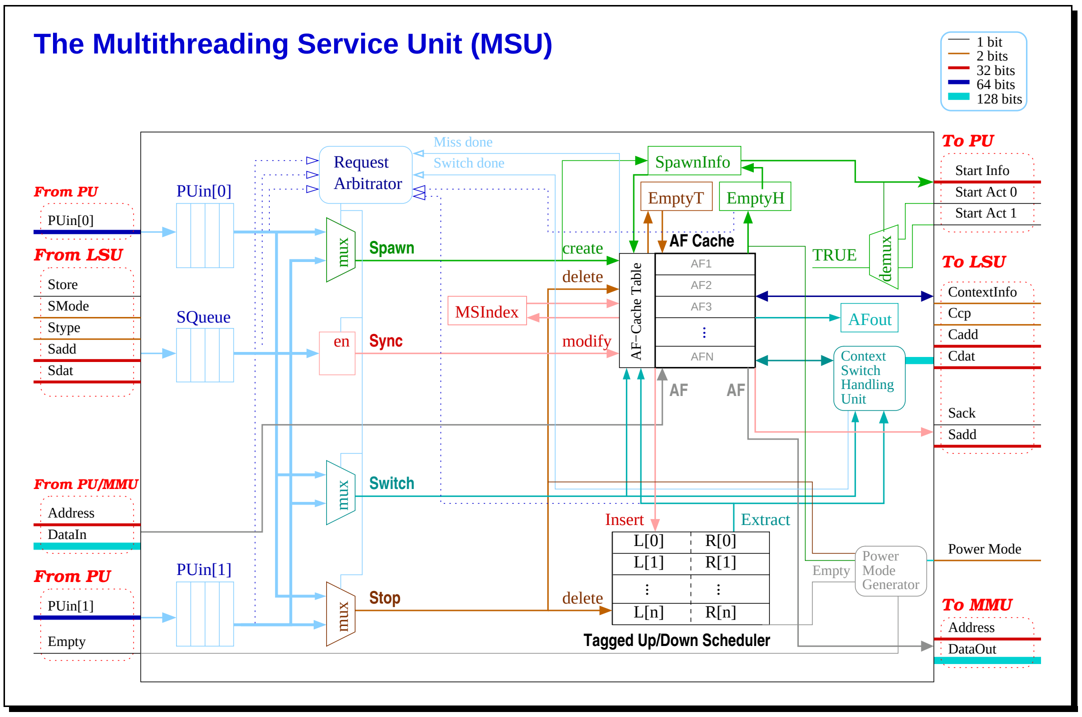


Figure 6.11: The multithreading service unit.

6.3.1 Request Arbiter

More than one multithreaded operation (i.e. spawning, synchronising, switching and stopping) may request to access shared resources (i.e. the activation-frame cache table and the activation-frame cache) at the same time. To prevent an undesirable clash, the multithreading service unit needs a request arbiter at the unit's front end (as depicted in blue on Figure 6.11). With the arbiter, requests from two processing elements of the processing unit (i.e. `PUin[0]` and `PUin[1]`), a request from the load store unit (i.e. `SQueue`), a request from the spawning preparation (see §6.3.2) and a request from the switching preparation (see §6.3.4) are separately selected according to their priorities. If their priorities are the same, then round robin ordering is applied.

In general, the arbiter makes a decision on every clock cycle. However, in the case that a previous operation needs more than one clock cycle to complete, which are a switching operation in §6.3.4 and an activation-frame cache miss penalty, incoming requests are blocked.

6.3.2 Spawning Mechanism

In conventional processors, or even in most multithreaded processors, initialisation of a new thread is performed by software [5]. With such a design, the processor performs the following sequence: reserve stack space and create a thread's initial context. Thus, the spawning operation often consumes a large number of processing cycles which may limit processing performance, obstruct real-time response and waste power. To eliminate the time-consuming spawning operation, MulTEP provides a simple but quick independent spawning mechanism[19] in the multithreading service unit. The spawning mechanism delegates the task to the multithreading service unit freeing up the processing elements.

The spawning operation consists of the following two steps:

1. **Prepare an empty activation frame**

   An empty activation frame is extracted from the head of the empty-AF linked list. This empty activation frame is available in advance. Figure 6.12 illustrates the case that the empty activation frame header is the last entry in the linked list. The spawning mechanism speculates that additional activation frames are required when this last empty activation frame is used. Thus, the mechanism simultaneously sends a request signal for additional empty activation frames to be allocated from the main memory[20].

2. **Spawn a thread**

   Once the token arrives, the mechanism creates a new thread context in the empty activation frame and simultaneously sends the activation-frame address back to a

[19] A spawning operation is additionally supported by a reserved activate thread table in the memory model (see Section 6.5) and a preparation of such a table by the kernel level (see §7.3.1).

[20] Additional empty activation frames are allocated with support from an interrupt daemon (see Chapter 7).

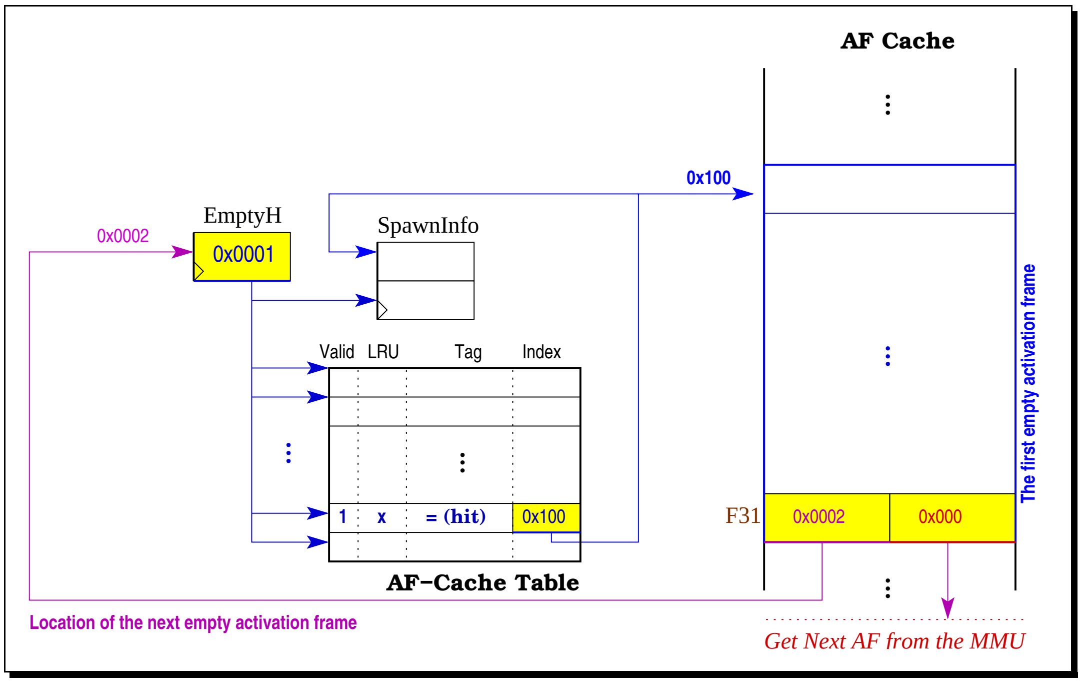


Figure 6.12: Preparing an empty activation frame in advance.

parent thread that issued this spawn instruction as acknowledgement. The mechanism performs and acknowledges the spawning request in a single clock cycle as shown in Figure 6.13.

An acknowledgement package is delivered to its parent thread via the load-store unit[21] in the form of a `STORE_DAT` package (see §6.4.4). Simultaneously, the empty activation frame is initialised as follows (see §5.3.2):

- The `AT` field, i.e. a PC space, and the `K1` field, the re-activate location, are both initialised with the *address*.
- The `K0` field is initialised with the *parent ID* and the *thread ID*.
- The priority and presence fields are initialised with the value in the *PP* operand, obtained from the $v0 register of the parent thread.
- The `SP` field is initialised with a *stack* operand, obtained from the $sp register of the parent thread.

6.3.3 Synchronising Mechanism

The hardware mechanism to support thread synchronisation is similar to Anaconda's *matching-store* mechanism [4]. However, it is slightly different because MulTEP delegates the *match* process to the load-store unit (see Section 6.4). A *store* message to an activation

[21]The parent thread may either remain in the processing unit or have already been switched out (see Section 6.4).

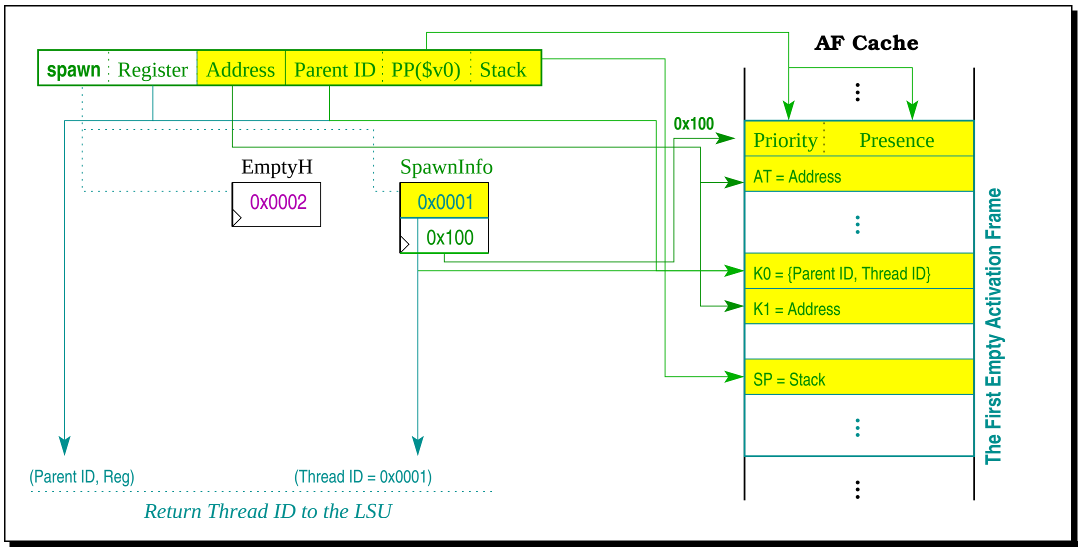


Figure 6.13: Spawning a new thread.

frame is a synchronisation event handled by the multithreading service unit. This process is illustrated in Figure 6.14.

The destination address is identified by a thread ID and a register ID. The thread ID indicates the activation-frame address[22]. The register ID indicates the location where data will be stored and asserts the corresponding presence flag.

The remaining mechanism will validate the status of the updated activation frame. If these presence flags are all present, then the updated thread is runnable and dispatched to the tagged up/down scheduler. In the tagged up/down scheduler, priority `0` is reserved to indicate an empty entry. Therefore, the runnable thread's priority is increased by `1` before being inserted into the queue to avoid a coincidental occurrence of priority `0`.

6.3.4 Switching Mechanism

To support a large number of threads, MulTEP provides a mechanism to switch execution contexts between the processing unit and the multithreading service unit. The switching mechanism collaborates with the outside-PU context-switching operation in the processing unit (see §6.2.3).

From the processing units perspective, the lowest-priority register set, which is not in use[23], is swapped out to the multithreading service unit. Simultaneously, the highest-priority runnable activation frame from the multithreading service unit is swapped in.

Prior to the swapping procedure, a switching decision needs to be made. The switching decision requires two steps because both the address of the highest runnable activation frame and the address of the lowest not-used activation frame have to be queued for accessing the shared activation-frame cache table as follows:

[22]The figure illustrates a case when a thread is presented in the activation-frame cache.
[23]A context status is not `Used` (see §6.2.3).

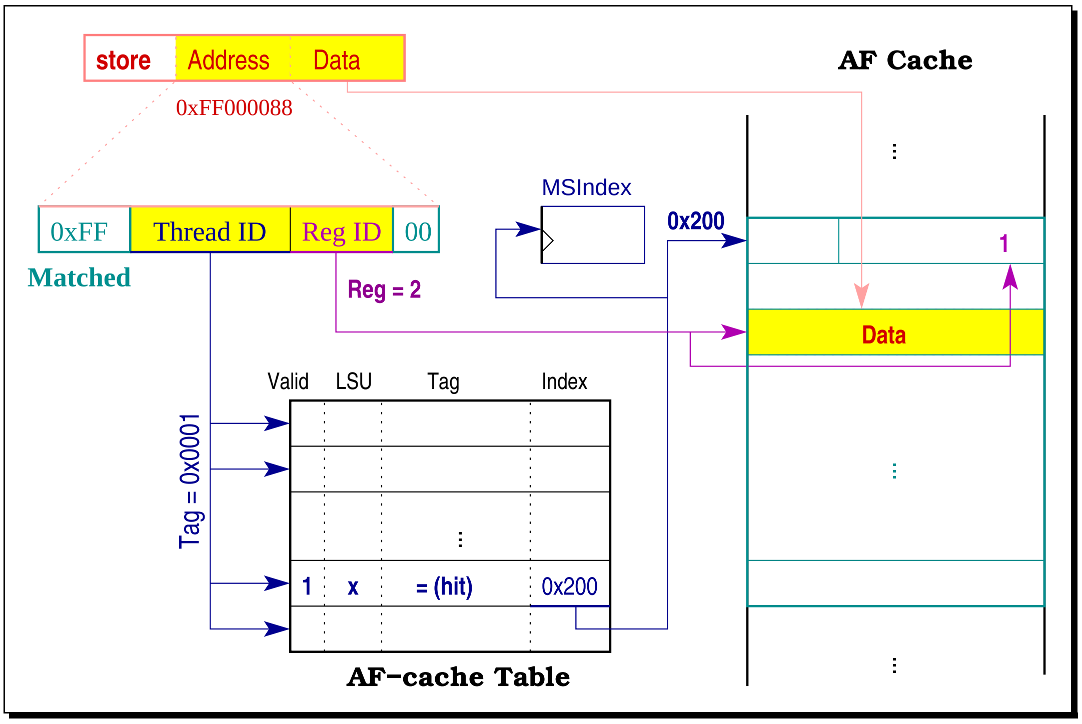


Figure 6.14: The store mechanism.

1. **Prepare the address of the highest runnable activation frame**

   Prior to the arrival of a switch package, the index of the highest-priority activation-frame address is prepared in an `AFout` register. The highest-priority activation frame is obtained from the thread ID of the valid[24] `R[0].Data` of the tagged up/down scheduler (see Figure 6.15).

2. **Issue a switching decision to the context-switch handling unit.**

   The switching decision for the context-switch handling unit consists of:

   - A pointer to a register set where the contexts will be swapped.
   - An "in" activation frame index for an execution context from the processing unit to the multithreading service unit.
   - An "out" activation frame index for an execution context from the multithreading service unit to the processing unit.

   When a switch token arrives, the following process occurs (see Figure 6.16):

   - The *register set pointer* is sent to the context-switch handling unit.
   - The *thread ID* resolves the index of the "in" activation frame and sends it to the context-switch handling unit.
   - The index in the `AFout`, which has already been prepared, is sent to the context-switch handling unit.

[24]The valid entry is indicated by the non-zero value of `R[0].Key`.

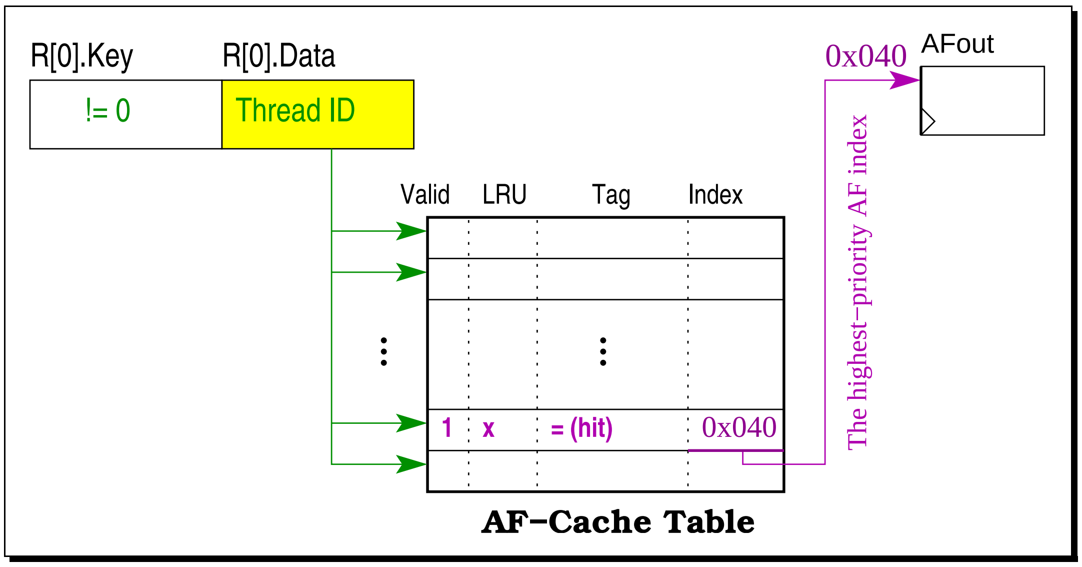


Figure 6.15: Preparing the index to the highest runnable AF in advance.

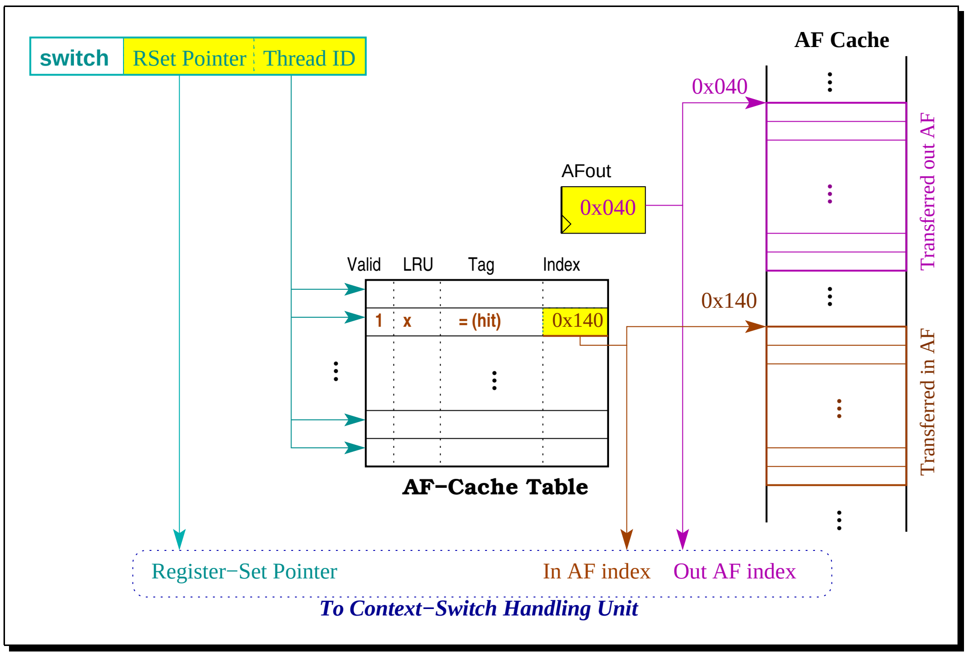


Figure 6.16: Issuing a switching decision to the context-switch handling unit.

As MulTEP uses 128-bit buses to transfer contexts, the context-switch handling unit is capable of transferring a context at the speed of 4 words per cycle to transfer up to 64 words in and out of the multithreading service unit. Based on this, the transferring process consumes either 7 clock cycles for the exchange of execution context within the same thread group, or 16 clock cycles for the exchange of execution context between the different thread groups (see §5.3.2).

When the transferring process is complete, the context-switch handling unit signals the arbiter to release its lock (see §6.3.1). The unit simultaneously notifies the load-store unit via an `UPDATE_PUInfo` command (see §6.4.4) to update the location of the transferred activation frames.

6.3.5 Stopping Mechanism

Terminated activation frames should be freed so that they are available for thread creation. This is provided by a stopping mechanism as shown in Figure 6.17. The mechanism frees the terminated activation frame, indicated by the thread ID of the stop token, by adding the activation frame to the empty-AF linked list. Simultaneously, a `DEL_PUInfo` command (see §6.4.4) is generated to the load-store unit in order to remove all of its correspondence entries (see Section 6.4).

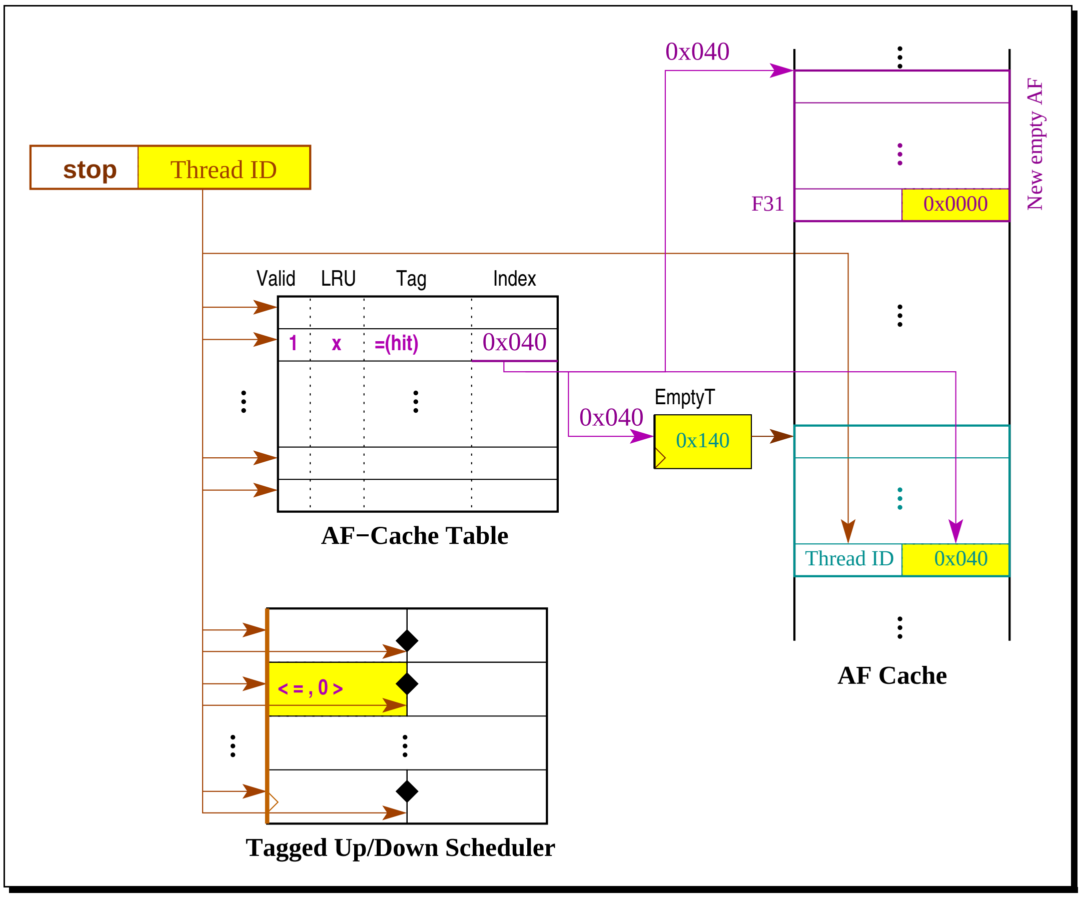


Figure 6.17: The stopping mechanism.

Additionally, a thread can be killed by another thread including a thread which is waiting to be scheduled. To avoid scheduling a terminated thread, the stopping mechanism searches through the tagged up/down scheduler in parallel. If a matched thread ID entry is found, its priority is set to zero and the zero-priority token will be effectively removed by the tagged up/down mechanism[25].

6.4 Load-Store Unit

In MulTEP, store instructions are used for thread synchronisation and play a critical role in hardware multithreading. Analyses of the flow paths of the load and store commands, has resulting in seperate paths between the *Processing Unit* (PU) to the *Multithreading Service Unit* (MSU) and the *Memory Management Unit* (MMU) are separate. This is because the path between the processing unit and the multithreading service unit is provided for thread synchronisation, while the path between the processing unit and the MMU is provided for data transfer.

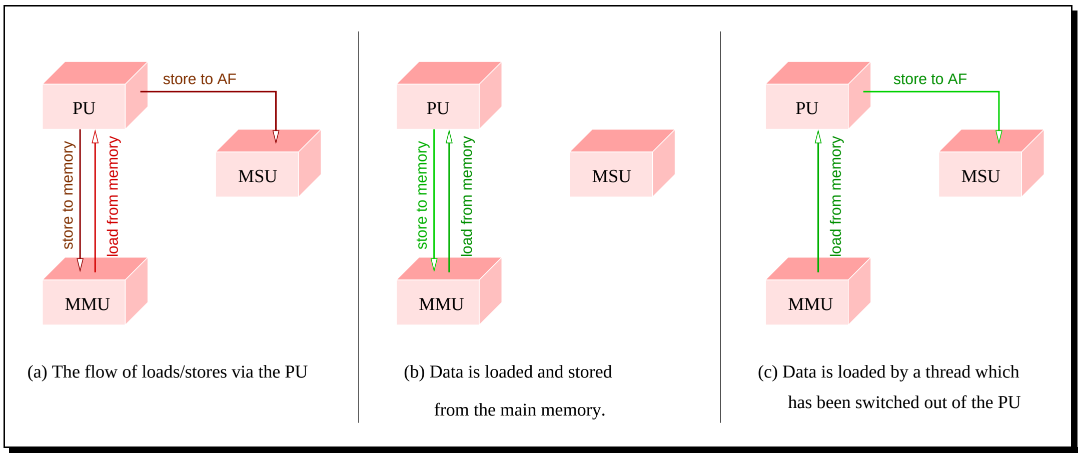

store to AF
PU
MSU
MMU
store to memory
load from memory
(a) The flow of loads/stores via the PU
PU
MSU
MMU
store to memory
load from memory
(b) Data is loaded and stored from the main memory.
store to AF
PU
MSU
MMU
load from memory
(c) Data is loaded by a thread which has been switched out of the PU

Figure 6.18: Instruction flows of load and store instruction without the LSU.

Figure 6.18 illustrates the MulTEP architecture without a load-store unit. Figure 6.18(c) depicts that data could be loaded from the memory by a thread which was switched out of the processing unit. Thus, a return of the loaded data would require the processing unit to generate a corresponding store command to the multithreading service unit. Centralising load/store execution on the processing unit wastes its service time and bus bandwidth, which are critical for the system.

In order that loads can bypass the processing unit, an alternative functional unit called a *Load-Store Unit* (LSU) was introduced into the architecture because it has the potential to alleviate the problem based on the investigation of load/store independence presented in Chapter 2. The load-store unit is provided to handle the flow of load and store instructions in MulTEP as illustrated in Figure 6.19(a).

[25]If the dead thread is in the R[0] of the tagged up/down sorter, the zeroth priority level will be extracted from the scheduler in the next clock cycle

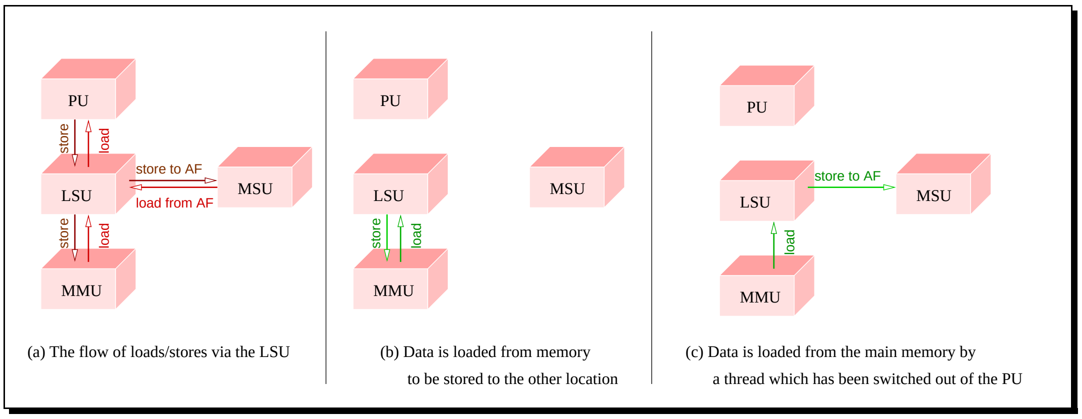

PU
store
load
LSU
store to AF
load from AF
MSU
store
load
MMU
(a) The flow of loads/stores via the LSU
PU
LSU
MSU
store
load
MMU
(b) Data is loaded from memory to be stored to the other location
PU
LSU
store to AF
MSU
load
MMU
(c) Data is loaded from the main memory by a thread which has been switched out of the PU

Figure 6.19: Instruction flows of load and store instruction with the LSU.

The load-store unit does not only handle data independently from the processing unit but also allows the processing unit to continue without stalling for load operations. Figure 6.19(b) and Figure 6.19(c) present the flow of load and store transactions of data that are irrelevant to the processing unit.

The underlying architecture of the load-store unit and its I/O interfaces to the processing unit, the multithreading service unit and the data memory are presented in Figure 6.20. Inside the load-store unit, two separate input queues are available for load/-store commands from each processing element[26] (§6.4.1), and three groups of mechanisms are provided to independently handle transactions as depicted with different colours in the figure. Mechanisms in green support load operations (§6.4.2). Mechanisms in pink handle store signals (§6.4.3). Mechanisms in blue handle all influences caused by load and store communications and support a couple of operations required by multithreaded operations (§6.4.4).

6.4.1 Load/Store Queues

The load-store unit consists of a number of mechanisms to be shared by the two processing elements. In order to prevent a clash access to these shared resources, arrival load/store requests from the two processing elements need to be queued.

The size of each input queue was estimated using the operation analysis [138] (λT). From cache performance results in [139], an arrival rate of load/store instructions (λ) is 23.20% of the total instructions.

Estimated from the cache design and conflicts of 4 threads obtained from [139], the worst data cache-miss rate (p) is 20%. As MulTEP is simulated with a cache miss-penalty (M) of 200 cycles and the cost of a hit (C) is 5 cycles [24], T is computed using Equation 2.1. To maximise the coverage of binary indexing bits, the size of the queue (N) should be a power of two:

[26]The load/store queues are separated because load/store events may arrive at the same time

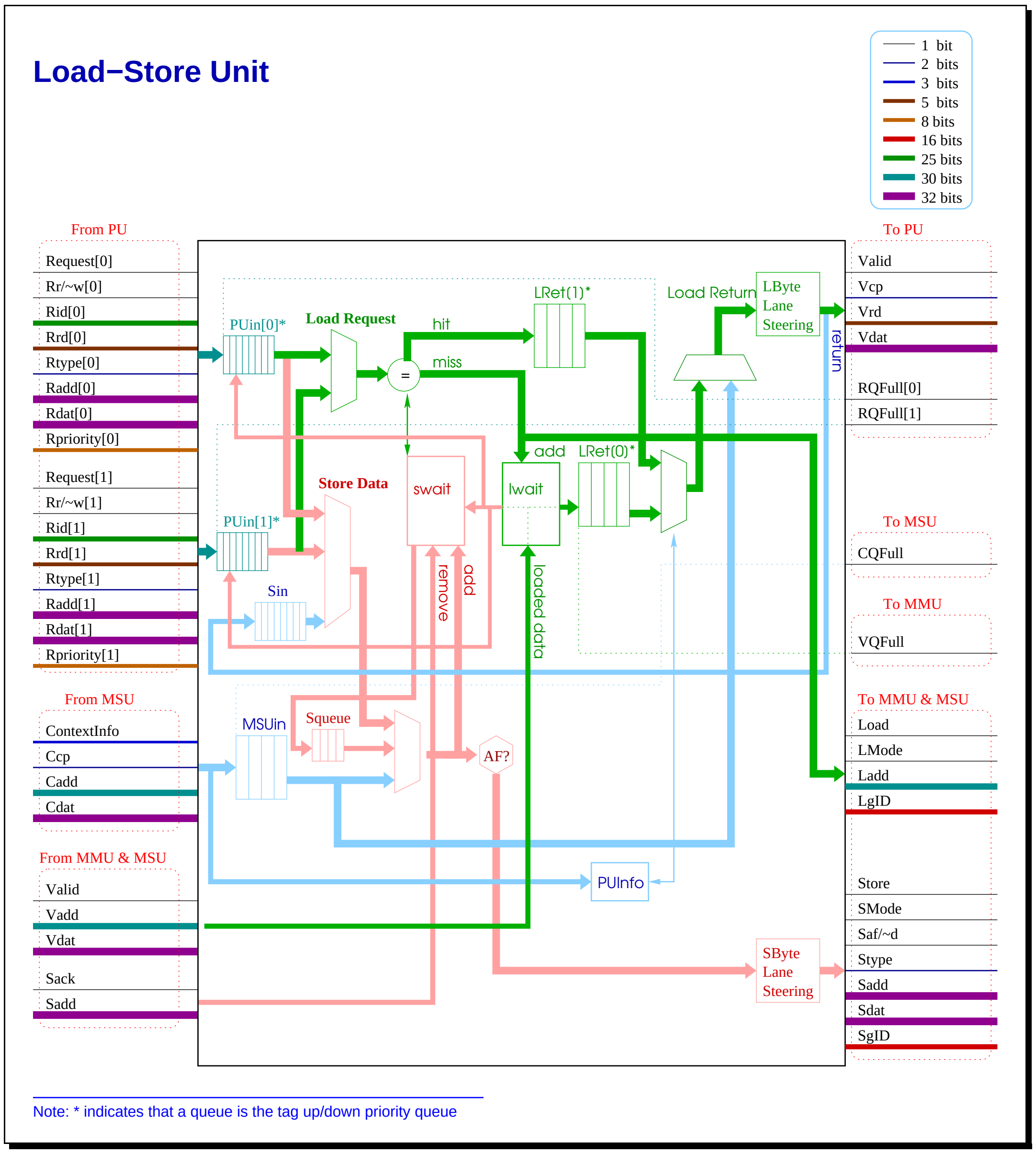


Figure 6.20: The load-store unit.

$$\begin{aligned} N &= 2^{\lfloor log_2 \lambda T \rfloor} && (6.1) \\ &= 2^{\lfloor log_2 \left(0.232023\left(pM+(1-p)C\right)\right) \rfloor} \\ &= 2^{\lfloor log_2 \left(0.232023(0.20\cdot 200+0.8\cdot 5)\right) \rfloor} \\ &= 2^{\lfloor log_2 5.568552 \rfloor} \\ &= 8 && (6.2) \end{aligned}$$

From the calculation, the suitable size of the queue is eight entries. Further evaluation beyond this theoretical estimation of the queue's size is conducted in Chapter 8 using simulations where the length of this queue is varied to figure out the optimum performance.

To support real-time requirements of MulTEP, each `PUin` is a tagged up/down priority queue [8]. The sorting key is an 8-bit thread priority and the data is the input package. To prevent a worst case queue-full situation, an `RQFull` signal is provided. This signal is asserted when the queue becomes full to stall the corresponding integer subpipeline.

6.4.2 Load Operations

The load-store unit redirects load-return transactions to their appropriate destination. For threads which have been switched out of the processing element, its load request is transferred to its activation frame in the multithreading service unit. In the interest of fairness and a simple implementation, load requests from the load/store input queues are selected in round-robin order through a multiplexer. To minimise the wasted state, the multiplexer's selector is capable of waiving the round robin opportunity if the next-turn input queue is empty. I designed a waiver by simply xoring the queue's round-robin turn with the empty flag of the round-robin queue as presented in Table 6.2.

Round robin	`PUin[0]` empty	`PUin[1]` empty	Selector
0	0	-	0
1	-	0	1
0	1	-	1
1	-	1	0

Table 6.2: A round-robin waiver logic table.

To prevent undesired read before write conflicts, the selected load address is compared with all addresses of the waiting store operations in a store-wait buffer (`swait`). If the address matches, then the data is returned to the first load return (`LRet[1]`) queue and waits to be sent back to the processing unit. A load-return data from the `swait` is buffered in the `LRet[1]` queue because multiple load-return data may also arrive from both data memory and the activation frame at the same time.

If the loading address is not located in the `swait` buffer, then the loading address, its protection mode and its group ID are sent out to acquire a datum. Simultaneously, its register destination, its thread priority and its thread ID are all kept in a load wait buffer (`lwait`). Once a valid datum returns, such an entry in the `lwait` buffer is sent to the zeroth load return (`LRet[0]`) queue. Consequently, the entry is removed from the `lwait` buffer. Valid data from load return queues and from some multithreading operations in the multithreading service unit (see details in §6.4.4) are selected in a round-robin order.

Before being transferring out load-return data to the processing unit, it is sent through a byte-lane steering unit. The byte-lane steering operates sign extension and adjusts big-endian/little-endian as appropriate. The unit additionally allows the load data to be unaligned or be represented in a byte, a halfword or a word.

6.4.3 Store Operations

Thread synchronisation of MulTEP is based on the matching-store methodology (see Chapter 2). Store instructions for activation frames are crucial for synchronising the execution of multiple threads. Therefore, it is necessary to let the load-store unit re-direct these store instructions to the multithreading service unit instead of to the data memory.

Two store requests are from the processing unit and the other one is from a store in queue (`Sin`). These requests are handled in round robin order. The `Sin` store-signal queue is an important element to re-direct load-return data to the multithreading service unit instead of to the processing unit. This redirection occurs when the execution context of the package's owner has already been switched out of the processing unit.

To redirect the flow of store transactions to their appropriate locations, an AF-address comparator is provided to let the store mechanism detect whether the address of a store request targets an activation frame (i.e. starts with `0xFF`) or not. If the destination is an activation frame, the store instruction is sent to the multithreading service unit. Otherwise, the instruction is sent to the data memory. Simultaneously, the store instruction is accumulated in the `swait` buffer [27] and remains there until its store acknowledgement is returned. The acknowledged store package is then removed from the `swait` buffer.

MulTEP allows the processor unit to issue an incomplete store request in order to allow the execution to continue to the next instruction without stalling (see an example in §8.2.2). The incomplete request is a store package whose data, address, or both, are not available. To support this, the load-store unit holds incomplete packages in the `swait` buffer and validates them with entries in the `lwait` buffer for data consistency. When an entry becomes complete, it is removed from the `swait` buffer to be transfered to its appropriate destination.

[27]The `swait` buffer is available to prevent the read before write conflicts.

6.4.4 Multithreaded Operations

As mentioned in the first two sections, the load-store unit supports thread communication by filtering incoming addresses and then directing them to their appropriate destination. The destinations of the load-return data are indicated by an information in a `PUInfo` table. The table consists of two entries: one for a thread ID and the other for a pointer to its register set. If the load-return thread ID exists in the table, the flow is sent to the processing unit. Otherwise, it is changed to a store instruction to be sent to the multithreading service unit.

The load-store unit is additionally capable of generating load and store instructions in correspondence with the multithreading requests of the multithreading service unit (see details in Section 6.3). These additional load and store packages are necessary to allow additional multithreading instructions to be effectively executed.

Table 6.3 shows how the 2-bit `ContextInfo` of the multithreading service unit represents four multithreading commands.

Command	**Value**	**Meaning**
`NONE`	00	No request
`UPDATE_PUInfo`	01	Update PU info about contexts allocation
`STORE_DAT`	10	A store instruction from the MSU
`DEL_PUInfo`	11	Delete PU info of a dead thread

Table 6.3: The `ContextInfo` states from the MSU to the LSU.

A default command is `NONE` which represents that there is no request from the multithreading service unit. An `UPDATE_PUInfo` command indicates that an execution context, pointed to by a `Cdat` (see the MSU input in Figure 6.20), has already been sent to a register set, indicated by a `Ccp`. The `UPDATE_PUInfo` command arrives when an outside-PU context-switching finishes. A `STORE_DAT` command is available to support a `spawn` instruction. This command lets the load-store unit generate a load-return package to send the location of a spawned activation frame back to its waiting register (details to be found in §6.3.2).

A `DEL_PUInfo` command is generated when a thread in the processing unit has already been stopped to let the load-store unit remove its corresponding entry from the `PUInfo` table.

6.5 Memory Management Unit

The *Memory Management Unit* (MMU) is a hardware mechanism that handles memory access authorisation, virtual memory addressing and paging translation. The system consists of a cache hierarchy, a couple of translation look-aside buffers, a multithreading access-control mechanism and address translation logic. The memory management unit incorporates the protection mode into the predefined memory domain. Its underlying mechanisms give priority to load data over store data to reduce the access latency of the memory hierarchy.

The following section starts with the memory hierarchy of MulTEP (§6.5.1). The predefined memory mapping area is then explained (§6.5.2), followed by the address translation (§6.5.3).

6.5.1 Memory Hierarchy

The memory hierarchy of MulTEP is illustrated in Figure 6.21. Virtual addresses are translated and validated through the TLB before accessing caches. This avoids the alias problem when multiple virtual addresses refer to the same physical address (see Chapter 2). L1 I-cache is a 16 kB direct-mapped unit with 1 kB block size. It needs a 5-cycle access latency supported by a *least-priority*[28] but *Not Last Used* (NLU) replacement with read allocation policy.

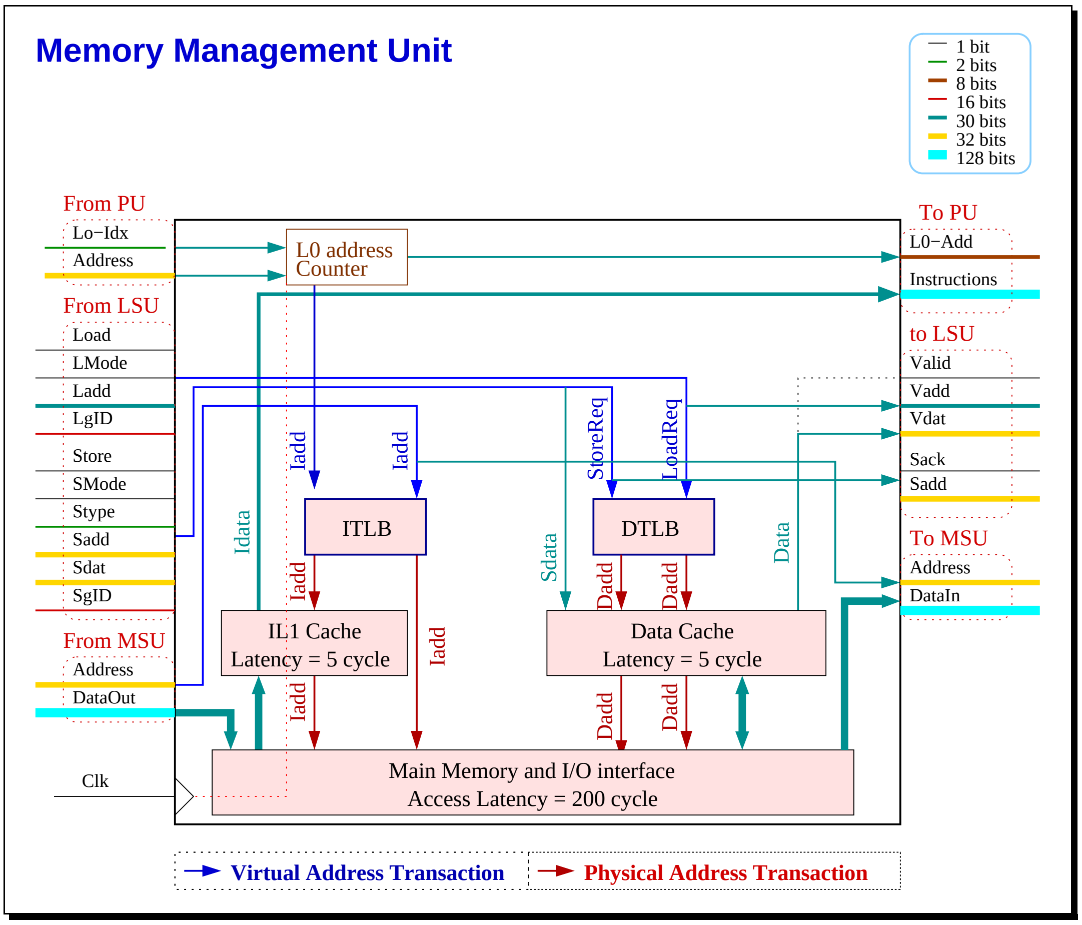


Figure 6.21: The memory management unit.

The D-cache is 16 kB 4-way set-associative with 128-byte block size and write back. The access latency is set out to be 5 cycles since this reflects current cache designs [140].

[28]An 8-bit priority is associated to every cache lines.

The replacement policy is least-priority but not last used. Main memory access latency is assumed to be 200 cycles in simulation.

6.5.2 Memory Address Mapping

MulTEP's virtual memory is segmented and reserved for different purposes as presented in Figure 6.22. MulTEP requires additional specific areas for keeping track of multithreaded operations and storing execution contexts in the form of activation frames. The areas in red are reserved for system accesses only and consist of the kernel system daemons, the address translation table, and a set of tables for multithreading (see §7.3.1). The instruction, data, and stack area are similar to the other architectures as found in [24, 26, 33, 34].

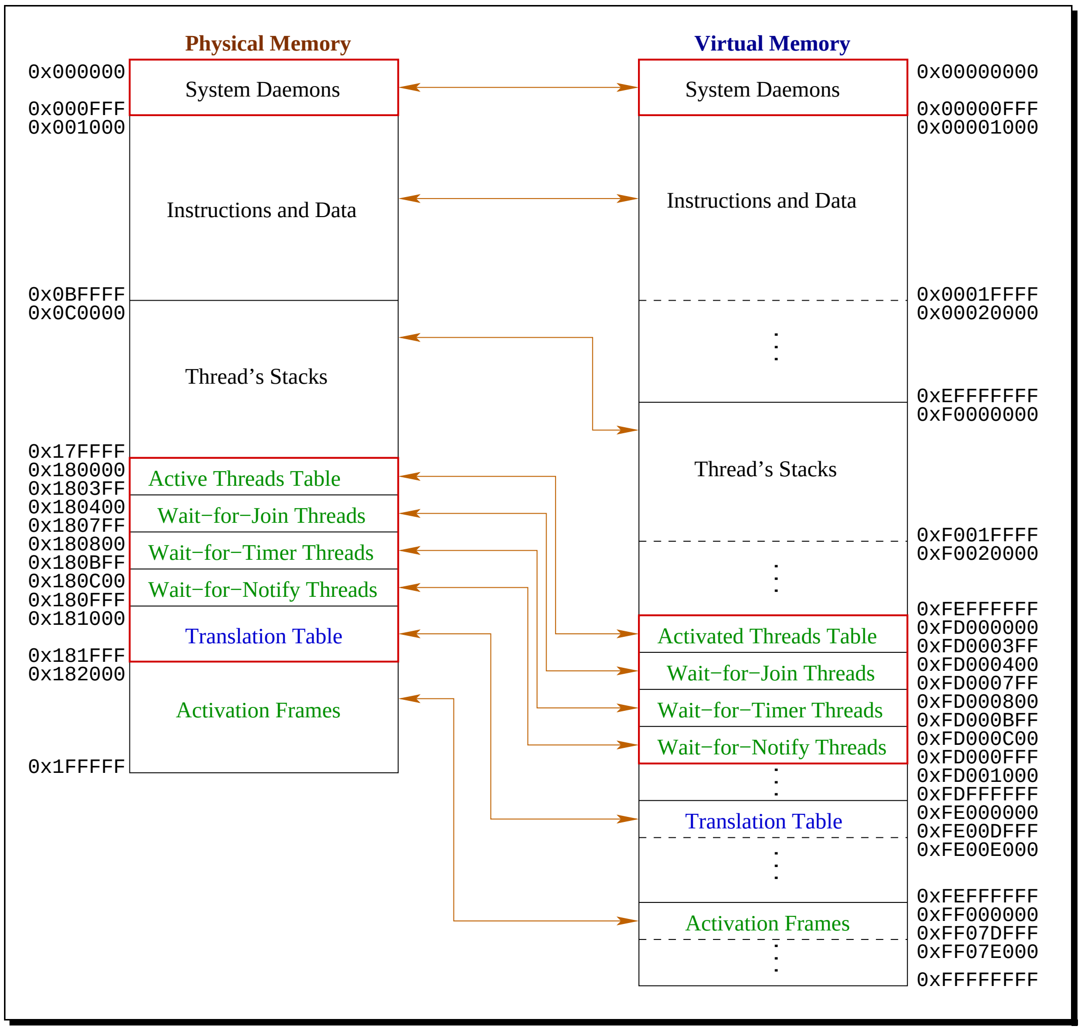


Figure 6.22: Virtual address segmentation and address mapping.

With the assigned mapping presented in Figure 6.22[29], a translation entry can be identified by a tag in a translation base address as shown in Figure 6.23 (see §6.5.3). This

[29]This physical memory is used in the initialised model of the simulator (see Chapter 8)

division of the translation base address allows one tag to cover up to 2M page entries (2^{21}).

Translation Base Address

FE	Tag	000
8 bits	21 bits	3 bits

Figure 6.23: Translation Base Address.

The virtual address of an activation frame is represented in Figure 6.24. MulTEP supports up to 2^{16} number of threads each of which contain 64 bytes for storing a thread's execution context. The mapping on the physical memory presented is an initialised mapping. Further physical area can be added by the dynamic address allocation which will create more entries in the translation table area and will rearrange the mapped addresses of the physical memory.

Activation Frame Address

FF	ThID	RegID	xx
8 bits	16 bits	6 bits	2 bits

Figure 6.24: Activation Frame Address.

6.5.3 Address Translation

A virtual address needs to be translated into a physical address. A translation look-aside buffer is used to cache recent virtual-to-physical address translations for rapid translation. The translation of a virtual address in the multithreaded system needs to incorporate some thread information in each entry in order to eliminate conflicts caused by duplicated virtual addresses from multiple threads. Thus, a huge amount of space for translation entries is required.

However, some of them may never be used. To avoid the waste of space, a multilevel paging scheme is selected for MulTEP to minimise space required for address translation [24]. For an address translation process, a translation look-aside buffer is first searched in parallel when the virtual address arrives (see Figure 6.25). The matched tag indicates an appropriate virtual page translation and a protection for the incoming virtual address. A "hit" on the translation look-aside buffer occurs when the incoming tag field is matched with the virtual page, and the incoming information is qualified. Otherwise, a "miss" on the translation look-aside buffer occurs indicating that the translation information should be loaded from memory.

In Figure 6.25, an input load request consists of an operation mode[30], a read/write flag, a 16-bit group ID and a 32-bit virtual address. The upper 20 bits of the virtual

[30]Value 0 represents a user mode, value 1 represents a system mode

memory is first matched with all tags of the DTLB in parallel. The protection bits (`valid`, `sys_read`, `sys_write`, `user_read` and `user_write`) are validated. The group ID is checked, if and only, if the private bit is equal to 1 indicating that the target page does not allow public access. If the physical page is available, then it is concatenated with the rest of the 12 bits of the page offset and sent as a physical address to the data cache.

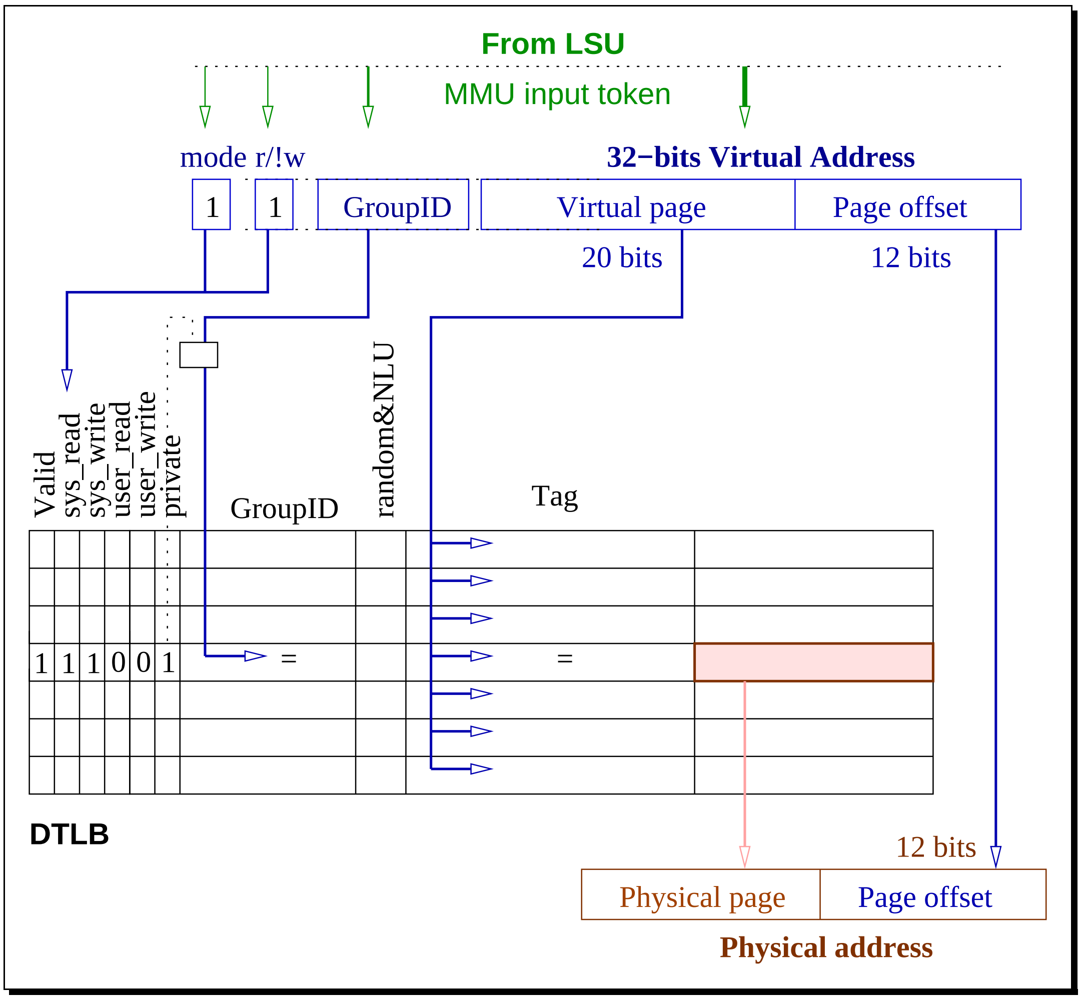


Figure 6.25: A virtual address translation when TLB hit.

A translation which misses in the translation look-aside buffer needs to be retrieved from the main memory. Multilevel paging requires two accesses to the memory as presented in Figure 6.26. For the first access, a 10-bit Tag 1 of the miss virtual address is used together with a 20-bit page base[31] to retrieve the second-level page based address.

For the second access, the second-level page base address is added to 32 bits of Tag 2, shifted left by 3 bits. The result is a pointer to the required 8-byte translation entry. The physical page of this translation entry is then concatenated with the page offset, a lower section of the input virtual address, to construct the physical address. Simultaneously, the translation entry replaces a random but not-last-used entry in the TLB.

[31]The 20-bit page base is provided in the page-base register of the memory management unit

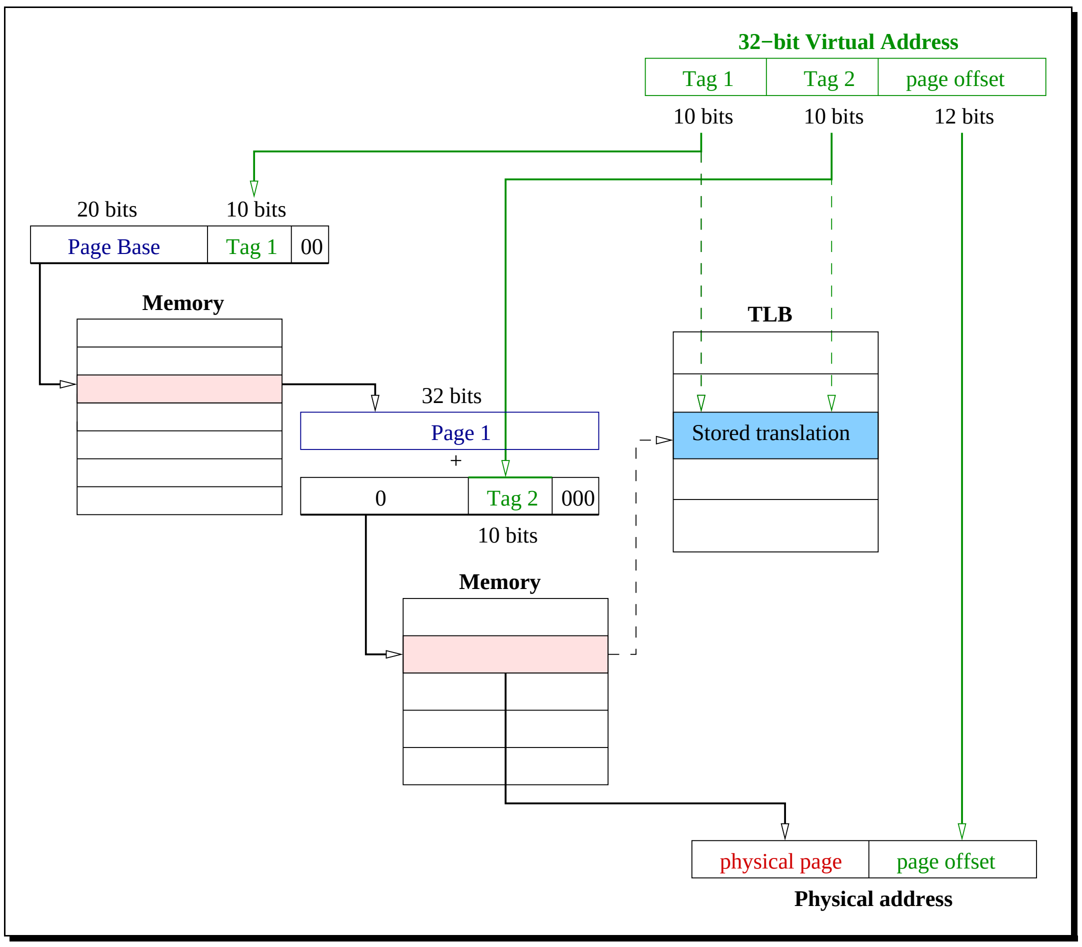


Figure 6.26: A virtual address translation when TLB miss.

Figure 6.27 illustrated an initialised translation table in the main memory of MulTEP. Entries in pink are reserved for the first-level page-bases. Entries in yellow are reserved for the second-level virtual-to-physical address translations.

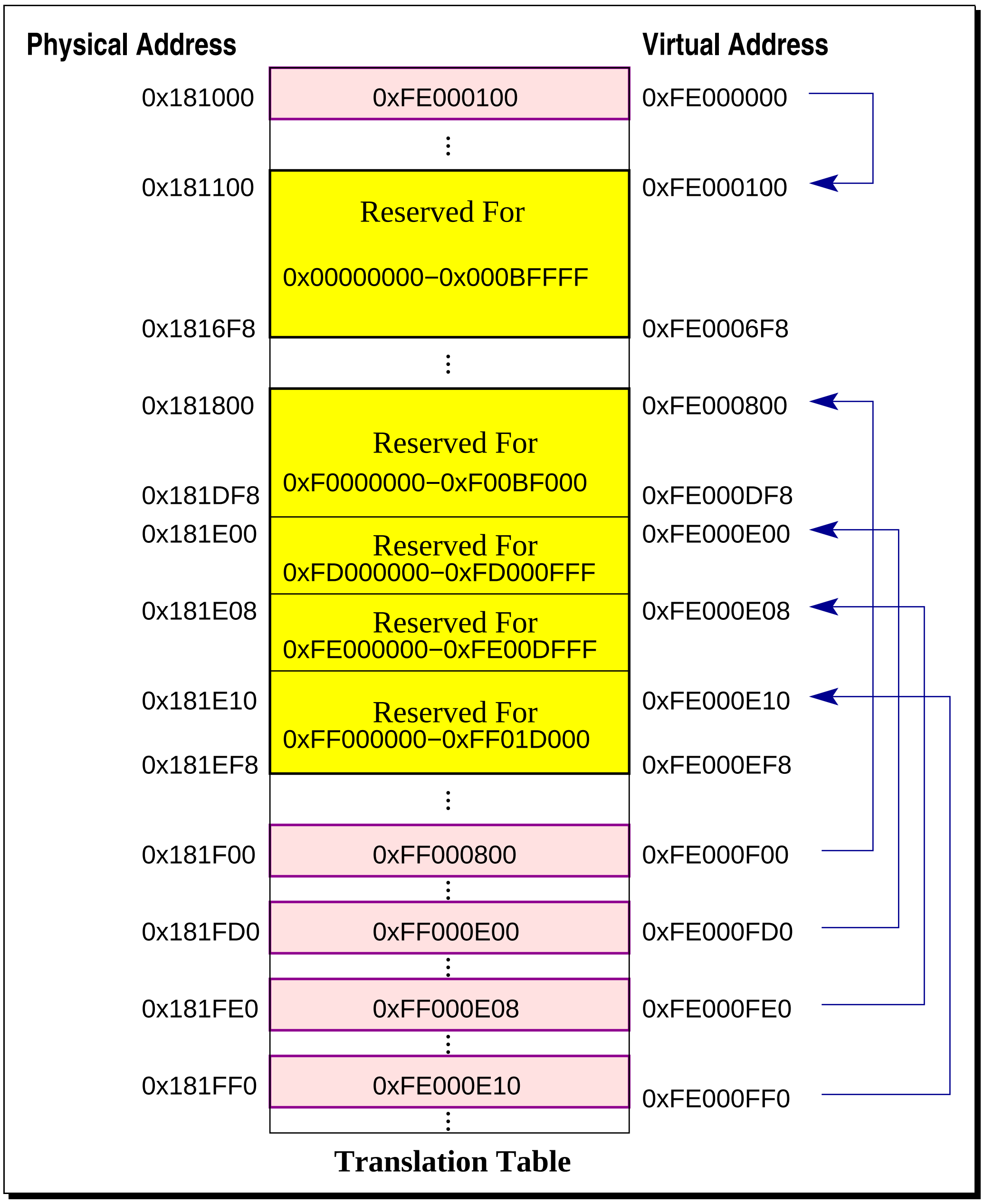


Figure 6.27: The initial state of a translation table in MulTEP.

6.6 Summary

The MulTEP hardware architecture supports a large number of threads based on a data-driven nanothread model. Priority-based switching and scheduling mechanisms are included for high-performance multithreading. Load and store instructions are used for thread synchronisation. These operations are supported by the following four hardware components:

1. **The Processing Unit (PU)**

   The processing unit has two processing elements, each of which consists of four subpipelines (integer, FP/multiplier, FP adder and FP/divider). Both processing elements share four execution contexts in order to support thread level parallelism at the hardware level.

   MulTEP uses pre-fetching, pre-loading and colour-tagging mechanisms to switch threads without any context switch overhead. *Pre-fetching* is provided by the collaboration of the level-0 instruction cache and the fetch sorted queue. *Pre-loading* is supported by an outside-PU context switch mechanism. This switching mechanism allows the context to be available in advance. *Colour-tagging* is the association of a register set index to the package that flows within the processing unit. With colour-tagging, the package is allowed to flow independently while maintaining a reference to its corresponding content.

2. **The Multithreading Service Unit (MSU)**

   The multithreading service unit supports a flexible number of threads. The underlying mechanisms consist of the activation-frame cache, synchronisation circuit and tagged up/down scheduler. This unit handles thread creation, thread synchronisation, thread scheduling and thread termination.

3. **The Load Store Unit (LSU)**

   The load-store unit reduces traffic to the processing unit. It enables faster loads and allows the processor to continue without stalling due to load and store operations. The underlying architecture supports data transfer of signed, unsigned, byte, halfword and word in both big-endian and little-endian data formats.

4. **The Memory Management Unit (MMU)**

   The memory management unit supports virtual memory addressing and memory protection. The memory hierarchy consists of two separated TLBs and caches for instructions and data.

Chapter 7

Software Support

Daring ideas are like chessmen moved forward;
they may be defeated, but they start a winning game.

Johann Wolfgang von Goethe

7.1 Introduction

Two software tools (i.e. the `j2n` and the MulTEP assembler) were written to facilitate multithreaded program creation for MulTEP. These tools support a number of thread standards and are adequate for evaluating the MulTEP architecture. An overview of software support for MulTEP is displayed in Figure 7.1.

The remainder of this chapter is structured as follows. Section 7.2 presents MulTEP low-level software support (as categorised in Figure 7.1). Section 7.3 shows MulTEP kernel-level support. Section 7.4 describes how some high-level languages make use of the lower level support along with suggestions for further implementation. Section 7.5 summarises the software tools designed for the MulTEP system.

7.2 Low-level Software Support

This section introduces two MulTEP low-level software tools. The first tool is a MulTEP assembler (§7.2.1) which is required to compile assembly language for MulTEP's instruction set architecture. The second tool is a set of MulTEP assembly macros (§7.2.2) which are provided to allow software to easily use multithreading features in MulTEP.

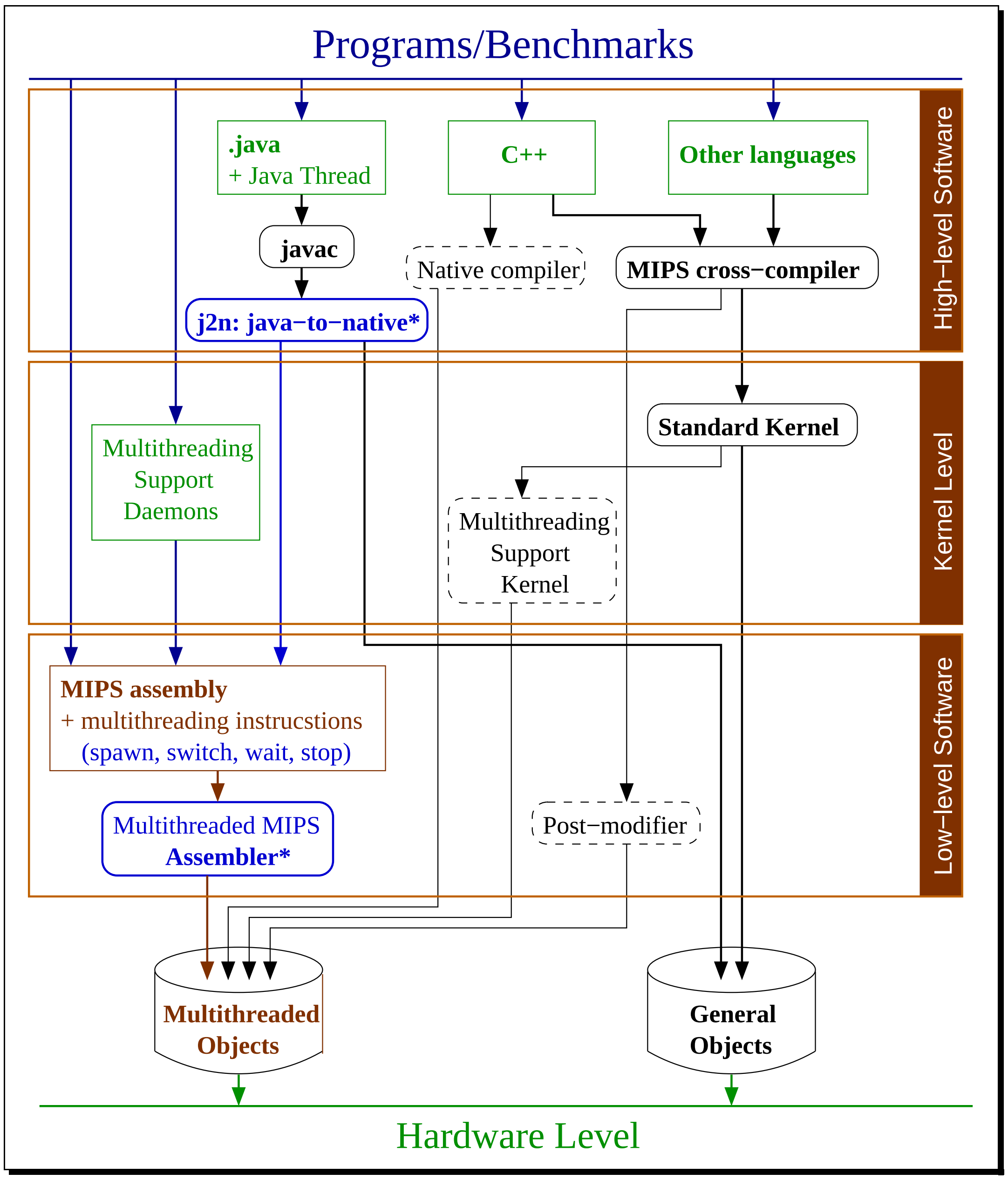


Figure 7.1: Software Overview for MulTEP.

7.2.1 MulTEP Assembler

In MulTEP, multithreaded operations are represented in assembly as a set of referable macros. A MulTEP assembler was built to support MIPS IV and multithreading instruction extensions (i.e. `spawn`, `switch`, `wait` and `stop` instructions). The assembler was implemented in C. There are two phases to the assembly process:

1. Code and data segments are separated. Instructions are re-ordered after a branch (i.e. add a single delay slot after a branch instruction). All label variables are represented with their correct absolute addresses.
2. All label variables are replaced with absolute addresses. Opcodes and operands are encoded in a series of 32-bit instructions.

The assembler parses MulTEP instructions (see §5.3.3) based on *Backus-Naur Form* (BNF) grammar as shown in Figure 7.2 and Appendix D.

```
<code>       →  {<comms>} .text <main> .data <data> .end

<comms>      →  #<text> {<comms>}

<main>       →  main: <inst> {<comms>} <m_inst> end
<inst>       →  <LSopcode> <reg>,<number>(<reg>) |
                <Sopcode> k1,<reg>(<reg>) |
                <Ropcode><cond> <reg>,<reg>,<reg> |
                <Ropcode> <reg>,<reg>,<reg> |
                <Iopcode> <reg>,<reg>,<number> |
                <Bopcode> <reg>,<reg>,<label> |
                <BZopcode> <reg>,<label> |
                <Sopcode> <reg>,<label> |
                <Jopcode> <number> |
                <ROpcode2> <reg>,<reg> |
                <IOpcode2> <reg>,<number> |
                <Opcode1> <reg> |
                <Opcode0>
<LSopcode>   →  <Lopcode> |
                <Sopcode>

<m_inst>     →  {<label>:} <inst> |
                {<label>:} <inst> <m_inst>

<data>       →  <label>: d<Dtype> <Ddat> {<comms>} |
                <label>: d<Dtype> <Ddat> {<comms>} <data>
```

Figure 7.2: MulTEP assembler's grammar in BNF.

7.2.2 MulTEP Assembly Macros

A generic life cycle of a thread is presented in Figure 7.3. There are four thread states in the life cycle, namely a *new thread* state, a *runnable* state, a *non-runnable* state and a *dead* state. To progress a thread through its life cycle, the system requires four commands, namely `create`, `block`, `resume` and `terminate`.

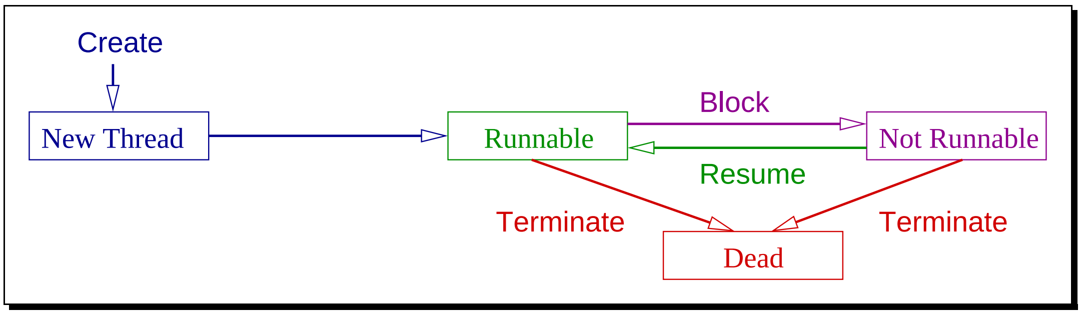


Figure 7.3: The generic life cycle of a thread.

Figure 7.4 illustrates Java thread life-cycle[1] as one example model to show that it is compatible with the generic model (Figure 7.3). A thread cycles through the Java thread model based on pre-emptive multithreading using a time-slicing methodology. The diagram uses the same style of group colours as shown in the generic model: blue for the new thread group, green for the runnable group, pink for the non-runnable group and red for the dead group.

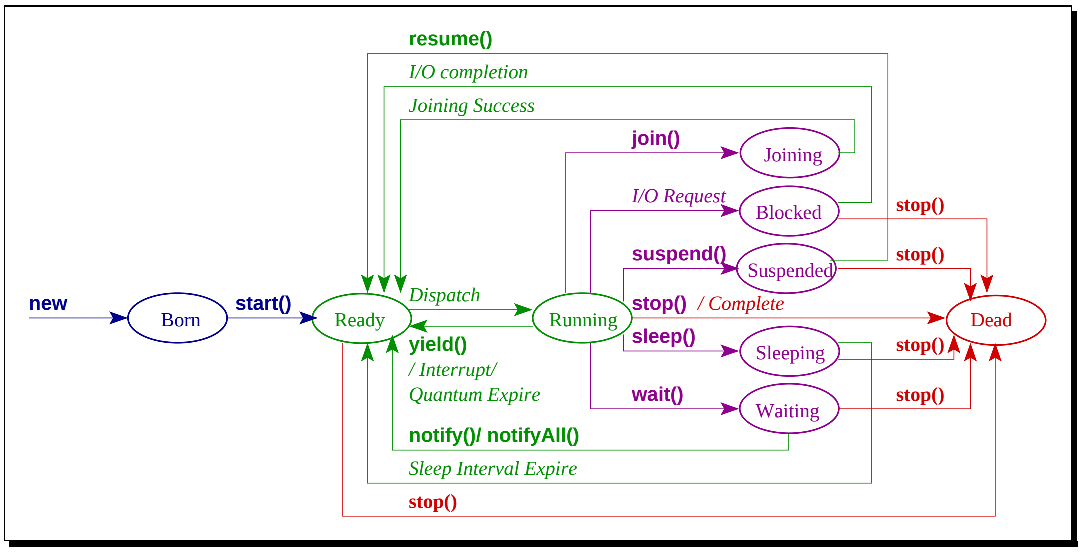


Figure 7.4: The life cycle of a Java thread.

To progress a thread through its life cycle, a number of MulTEP assembly macros are provided for used by high-level languages/compilers. These macros make use of MulTEP's multithreading feature for create, block, resume and terminate commands as follows.

[1]The reason that the Java model is presented in details is because its thread object was used in benchmarking the system (see Chapter 8).

Create

To initialise a new thread, two steps must be undertaken. The first step is thread construction where a thread and its environment are created. In object-based languages such as Java [90] and C# [141], a thread is born after being instantiated with a `new` indicator. In functional languages using a thread library such as PThread [98] or UI [97], a `create` function is used. To support thread construction, a macro called `new` is provided (see Figure 7.5). The operation of this macro first sets a priority, updates all presence flags, creates a stack and then spawns the thread.

```
;; ---------------------------------------------------------------
;; Macro:    new
;; Input:    $a0 is the start address of a spawn thread
;;           $a1 is the initial priority
;;           $a2 is the presence flags
;;           $a3 is the stack location
;; Output:   $v0 is the AF address of the spawn thread
;; ---------------------------------------------------------------

sll       $v0,  $a1,  24    ; Set the priority
or        $v0,  $v0,  $a2   ; Set presence flags
mov       $sp,  $a3         ; Set the stack location
spawn     $v0,  $a0         ; Spawn a thread pointed to by $a0
```

Figure 7.5: A new macro.

The second step for thread creation is to activate a thread. A `start` macro is provided to support this as presented in Figure 7.6.

```
;; ---------------------------------------------------------------
;; Macro:    start
;; Input:    $a0 is the AF address of a start thread
;; Output:   -
;; ---------------------------------------------------------------

xor       $s0,  $zero,  $zero   ; Prepare presence flags
swl       $s0,  0x0($a0)        ; Start a thread pointed to by $a0
```

Figure 7.6: A start macro.

Block

Blocking is used to synchronise execution ordering and to share data among threads [72]. Examples for the these operations are `wait()`, `suspend()` and `join()` functions from object-oriented languages [90, 141], and `cond_wait` and `cond_signal` functions from thread libraries [98, 97]. To support all these blocking functions, at least four blocking characteristics are required. The first blocking characteristic is a suspension with timer. I decided to provide a `sleep` macro for this as presented in Figure 7.7.

```
;; ----------------------------------------------------------------
;; Macro:    sleep
;; Input:    $a0 is the AF address of a thread
;;           $a1 is the start timer
;;           $s6 is an available wait-for-timer entry
;; Output:   -
;; ----------------------------------------------------------------

sw        $a0,  0x0($s6)   ; Create a new sleeping entry
sw        $a1,  0x4($s6)   ; Set the start timer
wait      $v0              ; Wait for time up (signal via $v0)
```

Figure 7.7: A sleep macro.

The second type of block is when a thread is waiting for completion of another thread. A `join` macro is provided as presented in Figure 7.8.

```
;; ----------------------------------------------------------------
;; Macro:    join
;; Input:    $a0 is the AF address of a waiting thread
;;           $a1 is the AF address of a waited thread
;;           $s5 is the wait-for-join pointer
;; Output:   -
;; ----------------------------------------------------------------

sw        $a0,  0x0($s5)   ; Create a join entry
sw        $a1,  0x4($s5)   ; Set the thread ID to be waited for
wait      $v1              ; Wait for join (signal via $v1)
```

Figure 7.8: A join macro.

The third type of block is a wait for a resume signal which needs to specifically target the waiting thread. To support this, I decided to provide a `suspend` macro as presented in Figure 7.9.

The forth type of block is a wait for a general notify signal which will activate the first wait-for-notify thread in the queue. A `wnotify` macro is provided for this purpose as shown in Figure 7.10.

```
;; ------------------------------------------------------------
;; Macro:    suspend
;; Input:    -
;; Output:   -
;; ------------------------------------------------------------

wait        $a0         ; Wait for resume (signal via $a0)
```

Figure 7.9: A suspend macro.

```
;; ------------------------------------------------------------
;; Macro:    wnotify
;; Input:    -
;; Output:   -
;; ------------------------------------------------------------

wait        $a1         ; Wait for notify (signal via $a1)
```

Figure 7.10: A wnotify macro.

Resume

A resume operation allows a non-runnable thread to return to its runnable state. Functional libraries [98, 97] uses `unlock` functions with different parameters to resume from the lock status. Object-oriented languages [90, 141] use either a *specific resume* signal (i.e. a join-success notification, an I/O completion, an interval expiration or a `resume()` function) or a *generic resume* signal (i.e. a `notify()` function or a `notifyAll()` function). To meet these requirements, MulTEP provides both specific and generic macros. The specific resume is a `resume` macro (see Figure 7.11).

```
;; ------------------------------------------------------------
;; Macro:    resume
;; Input:    $a0 is the AF address of the resuming thread
;; Output:   -
;; ------------------------------------------------------------

sw          $k1,   $a0($a0)   ; Signal a wait-for-resume field (A0)
```

Figure 7.11: A resume macro.

The generic resume is a `notify` macro (see Figure 7.12).

```
;; ------------------------------------------------------------
;; Macro:   notify
;; Input:   $s7 is the wait-for-notify pointer
;; Output:  -
;; ------------------------------------------------------------

lw       $t0,   0x0($s7)   ; Load a wait-for-notify thread ID
sw       $k1,   $a1($t0)   ; Signal a wait-for-notify field (A1)
```

Figure 7.12: A notify macro.

Termination

Thread termination eliminates the thread and its environment (i.e. its stack space and its activation frame). Objected-oriented languages [90, 141] kill a thread via a `stop()` function. Functional libraries [98, 97] uses a `return` indicator to identify completion of the thread and a `kill` function to terminate the thread. Since MulTEP has the `stop` instruction, which effectively handles the thread termination in hardware, the `kill` macro is trivial (see Figure 7.13).

```
;; ------------------------------------------------------------
;; Macro:   kill
;; Input:   $a0 is the AF address of a thread to be killed
;; Output:  -
;; ------------------------------------------------------------

stop     $a0,        ; Kill a thread pointed to by $a0
```

Figure 7.13: A kill macro.

7.3 Support for System Kernel

This section describes three kernel-level MulTEP multithreading features: system daemons (§7.3.1), interrupt daemons (§7.3.2) and non-running states (§7.3.3).

7.3.1 System Daemon

Daemon threads are a set of endless loops waiting to provide important services to the other threads. A high-priority timer daemon is required to be executed at regular intervals to wake up all sleeping threads whose timers have expired. The wake up process is done through a quantum expiration signal. A low-priority garbage collector is used to clear up the stack spaces and unwanted data that is left over in the system.

A system daemon, called Thread 0, is provided to handle multithreading services and is given the highest priority level (see Appendix C.3). Thread 0 is a very small thread. It is woken up at regular intervals by a signal sent to its register, `$t0`, once the daemon-thread timer reaches zero. It undertakes house-keeping procedures such as checking the timeout periods of suspended threads, joining threads after wait-to-be-joined threads are complete, notifying waiting threads and checking the status of the hardware scheduler.

A range of virtual addresses from `0xFD000000` to `0xFD000FFF` is specially provided for Thread 0. These addresses are permanently available in the D-cache because their owner is the system daemon with the highest priority. The segment consists of an activated-thread table, a wait-for-join table, a wait-for-timer table and a wait-for-notify table, which are pointed to by registers `$s4`, `$s5`, `$s6`, and `$s7`, respectively.

Active-thread Table

When a thread is terminated, its execution environment needs to be cleared from the system. In some hardware units, such as a pipeline and caches, garbage from dead threads is naturally eliminated when time passes by or cleared via the stop operation in the multithreading service unit (see §6.3.5). However, skeletons of dead threads held by the software still remain. These skeletons are threads that could be waiting for a timer, to be joined or to be notified. To eliminate them from the memory system, I decided to provide a record of all current active threads in an active-thread table as presented in Figure 7.14. The structure of the table is implemented as a linked list[2]. Register `$t1` in Thread 0 indicates the head of empty space for additional active threads. Thus, insertion is conducted in a single clock cycle by hardware support in the multithreading service unit (see Section 6.3). Register `$s4` in Thread 0 indicates the head of the active thread linked list. Search and extract operations take $O(n)$ cycles in software[3].

One active-thread entry consists of four fields: a 16-bit `thread ID`, a 3-bit thread `state` (see 7.3.3), a single-bit expandable flag (`Ex`) and a 12-bit `link` for the next entry (within the active-table area). To allow a system to go beyond 512 threads limited by the active table space, an expandable mode is provided. When the table is almost full, the single-bit expandable flag `Ex` of the penultimate entry asserts. As shown on the right of Figure 7.14, the next referenced entry holds a virtual address of the next thread in the main memory.

[2]The linked-list structure is used because of the limited implementation time (i.e. most of the time are spent for hardware improvement) as the structure does not much degrade the processing performance due to its infrequent usage.

[3]Though the operation performance is not yet critical, to improve both searching and extracting performance, a better data structure than an unsorted linked list such as a hash table may be implemented instead (see §9.5.7).

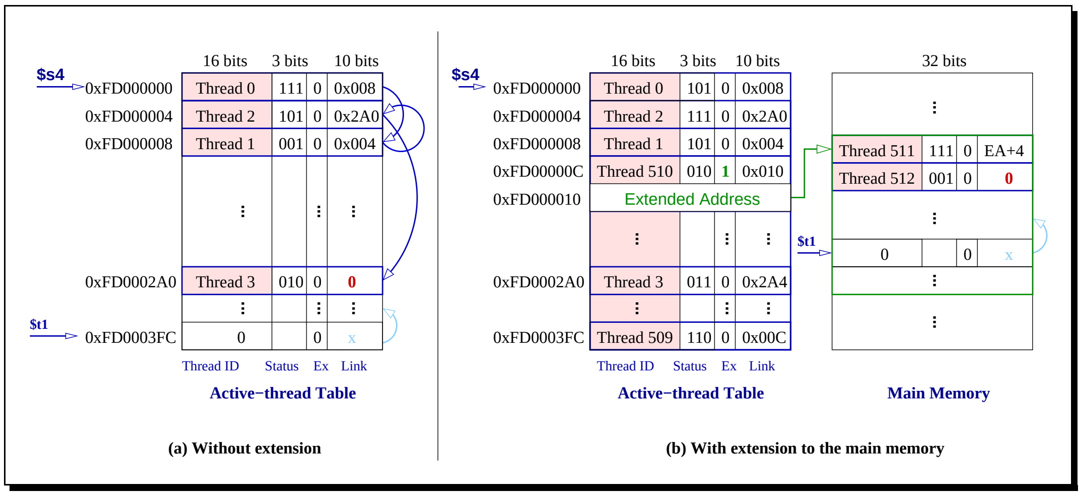


Figure 7.14: An active-thread table and its extension to the main memory.

Wait-for-join Table

With the matching-store paradigm, synchronisation is conducted in a single clock cycle in the multithreading service unit via a store instruction to an appropriate activation frame (see §6.3.3). One of the typical thread synchronisation function is `join`. A thread uses `join` to suspend its operation when it has to wait for other threads to complete their operations.

MulTEP offers direct support in Thread 0 to retrieve the join location in a couple of cycles. Figure 7.15 illustrates the structure of a wait-for-join table, pointed to by the `$s6` register in Thread 0. The structure is a linked list. Insertion completes in a single clock cycle by the multithreading service unit with the help of Thread 0's register \$t2, a pointer to the table's empty space. Search and extract operations take $O(n)$ cycles[4].

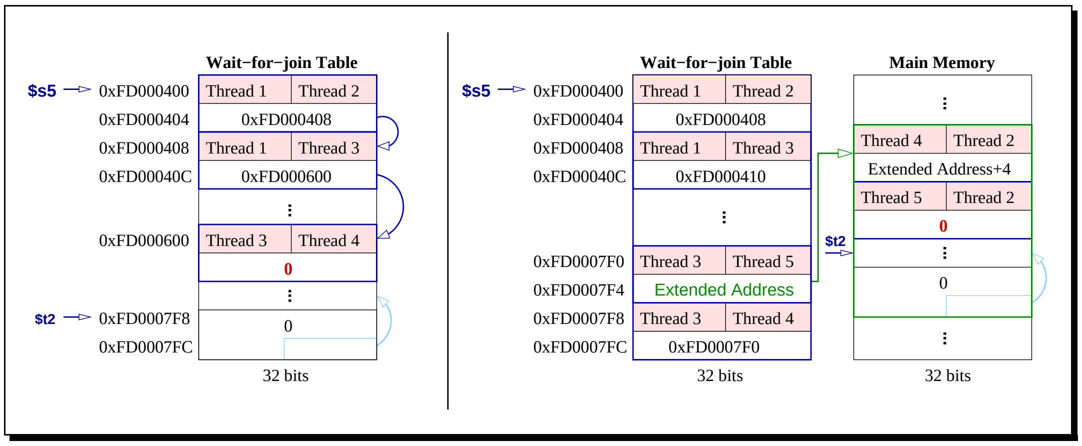


Figure 7.15: A wait-for-join table and its extension to the main memory.

One entry consists of a 16-bit thread ID of the waiting thread, a 16-bit thread ID

[4]To improve both searching and extracting performance, a better data structure than an unsorted linked list, such as a heap sort, can be implemented (see §9.5.7).

of a thread to be waited on and a 32-bit next entry pointer. Though, 256 entries are reserved to store a wait-for-join in the D-cache, the table can be extended to the main memory via a simple reference method as presented in the right-hand part of Figure 7.15.

Wait-for-timer Table

Timing is obviously important for embedded environments. In most conventional architectures, the kernel is used to manage time-constrained synchronisation. This consumes a number of cycles and may be inefficient for real-time systems. Instead, MulTEP provides timing information for thread synchronisation via a wait-for-timer table. This table stores the timer of all suspended threads which are waiting for quantum expirations. These timers are decremented at regular intervals. If a quantum-expired thread is detected, a thread will be re-activated within a couple of clock cycles.

The linked list structure of a wait-for-timer table is presented in Figure 7.16. One entry comprises a 16-bit thread ID, a 12-bit next entry link and a 32-bit timer.

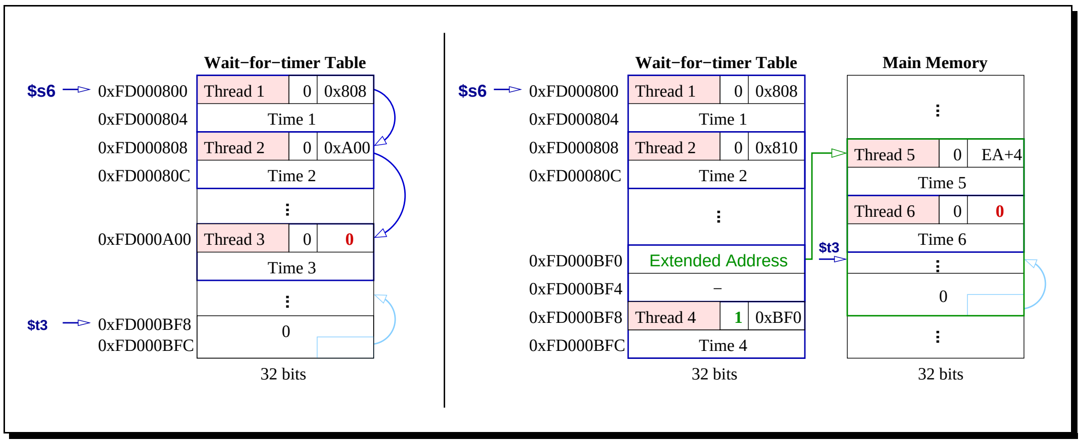


Figure 7.16: A wait-for-timer table and its extension to the main memory.

To conform to the standard kernel requirements [97, 98], the timing resolution is one millisecond and supports up to 2^{32} ms, i.e. almost 50 days [97, 98]. Up to 256 entries are provided in the data cache. Excess entries are expandable to the main memory as shown in the right part of Figure 7.16.

Wait-for-notify Table

A number of programming languages provide thread synchronisation through wait and notify pairs to protect shared resources [90, 141]. With this scheme, a thread waits to be notified by any other threads in first-in, first-out ordering. To support this, MulTEP provides a record of wait-for-notify entries in a linked list as illustrated in Figure 7.17. One entry consists of a 16-bit thread ID, an expandable bit and a 12-bit next entry pointer. The table supports up to 512 entries in the data cache and can be spilled to the main memory using the same method as the active-thread table.

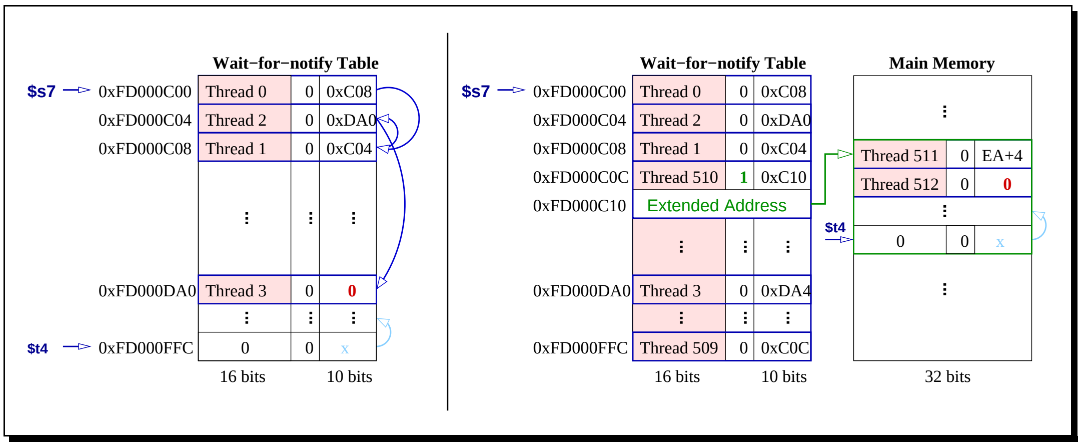


Figure 7.17: A wait-for-notify table and its extension to the main memory.

7.3.2 Exception/Interrupt Daemons

Exception/interrupt daemons, such as an arithmetic error and divide by zero, are high priority threads (their priority levels are still lower than the system daemon). These threads wait to be activated by store messages to exception/interrupt activation frames.

7.3.3 Non-runnable States

Instead of wasting a number of cycles to figure out which appropriate interpretation should be applied to each incoming synchronisation request, Thread 0 directly obtains thread non-runnable state from the active-thread table and operates a suitable procedure.

There are 8 thread states as shown in Table 7.1 (see §5.3.4). Six of them are non-runnable states which are *born*, *sleeping*, *joining*, *blocked*, *suspended* and *waiting*. A born thread waits for the data of all parametric registers, i.e. from `$v0` to `$t9`. A sleeping thread, a joining thread, a suspended thread and a waiting thread monitor a signal to register `$v0`, `$v1`, `$a0` and `$a1`, respectively. A blocked thread is a general thread waiting for any incomplete data. Specific register locations wait for incoming store signals that can activate a thread to be runnable when all parametric registers are present.

State	Value	Wait For
`dead`	000	-
`born`	001	all
`sleeping`	010	`$v0`
`joining`	011	`$v1`
`blocked`	100	x
`suspended`	101	`$a0`
`waiting`	110	`$a1`
run	111	-

Table 7.1: A state of the thread.

7.4 Support for High-level Languages

This section starts with a discussion about real-time embedded languages and their relation to MulTEP (§7.4.1). Then, a Java-to-native post-compiler, i.e. `j2n`, is described (§7.4.2). After that, suggestions for implementing a MulTEP native compiler is explained (§7.4.3).

7.4.1 Real-time Embedded Languages

A number of real-time languages for embedded systems such as concurrent Ada 95 [142] and real-time Java [143] are supported by concurrent operations [144]. The languages are designed to provide real-time, pre-emptive multitasking applications (i.e. Real-time Java is supported by a `javax.realtime` package [145]). The specialities of these real-time embedded languages are their features that allow users to access the hardware such as thread scheduling and event handling and controlling memory accesses.

Applications written in real-time embedded languages are capable of providing a rapid interrupt response through a fast context switch and a priority-based scheduling. As MulTEP naturally supports concurrency in hardware, special features of these languages directly benefit from the architecture as follows:

1. **The asynchronous event handler**

   Events often occur asynchronously and need to be handled in real time. The event handler (e.g. an `AsyncEventHandler` class in the real-time Java, or an `Asynchronous_Task_Control` class in Ada 95) should be generated dynamically in a form of a thread. In MulTEP, the event handler object can be initially created with a `spawn` instruction with one or more incomplete parameters (i.e. some registers are still waiting for data). The asynchronous event (e.g. an `AsyncEvent` object in the real-time Java) is then transformed to a `store` instruction targeting the missing parameter of the suitable event handler.

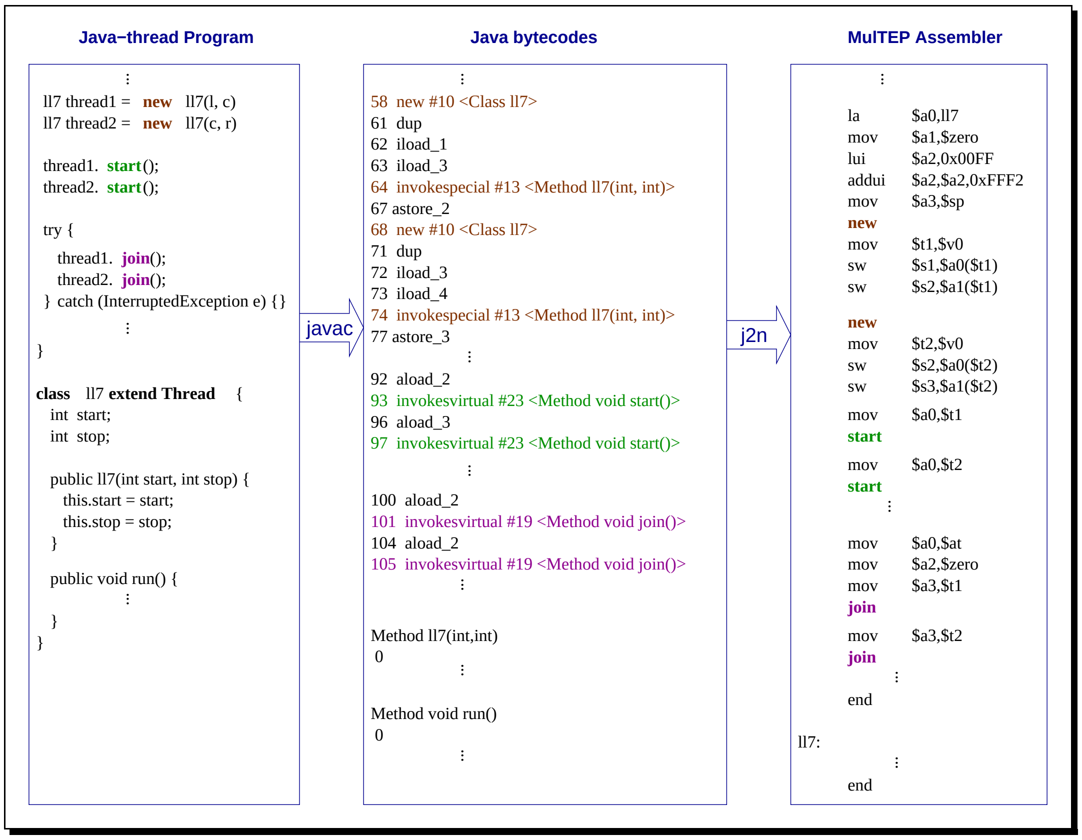
2. **The priority-based scheduler**

   Real-time embedded languages have features that allow a thread to be associated with a priority and pre-emptive scheduled in accordance to its priority level. Real-time Java's 128 priority levels (e.g. in a `RealtimeThread` class) and concurrent Ada 95's 256 priority levels (e.g. in a `Dynamic_Priority` package) fit well within MulTEP's 256 priority levels. Furthermore, an instance of a scheduler/dispatcher can be directly handled by the MulTEP hardware (i.e. the multithreading service unit) thereby removing software overhead.

7.4.2 Java-to-native Post-compiler

The *java-to-native* post-compiler (`j2n`) was implemented for system benchmarking via Java-threaded programs (see Chapter 8). A Java thread is created by sub-classing the

Thread class [90]. The public method *run()* is in the main body. Its life cycle is shown in Figure 7.4. The underlying mechanism for porting Java's byte-codes to MulTEP assembler is extended from the Kaffe virtual machine [146]. A Java program is first compiled into Java byte-codes by a `javac` compiler (see Figure 7.18[5]).



```
        ⋮
 ll7 thread1 =  new  ll7(l, c)
 ll7 thread2 =  new  ll7(c, r)

 thread1. start();
 thread2. start();

 try {
    thread1. join();
    thread2. join();
 } catch (InterruptedException e) {}
        ⋮
}

class  ll7 extend Thread  {
  int start;
  int stop;

  public ll7(int start, int stop) {
    this.start = start;
    this.stop = stop;
  }

  public void run() {
        ⋮
  }
}
```

```
        ⋮
58  new #10 <Class ll7>
61  dup
62  iload_1
63  iload_3
64  invokespecial #13 <Method ll7(int, int)>
67 astore_2
68  new #10 <Class ll7>
71  dup
72  iload_3
73  iload_4
74  invokespecial #13 <Method ll7(int, int)>
77 astore_3
        ⋮
92  aload_2
93  invokesvirtual #23 <Method void start()>
96  aload_3
97  invokesvirtual #23 <Method void start()>
        ⋮
100  aload_2
101  invokesvirtual #19 <Method void join()>
104  aload_2
105  invokesvirtual #19 <Method void join()>
        ⋮

Method ll7(int,int)
 0
        ⋮

Method void run()
 0
        ⋮
```

```
          ⋮
        la      $a0,ll7
        mov     $a1,$zero
        lui     $a2,0x00FF
        addui   $a2,$a2,0xFFF2
        mov     $a3,$sp
        new
        mov     $t1,$v0
        sw      $s1,$a0($t1)
        sw      $s2,$a1($t1)

        new
        mov     $t2,$v0
        sw      $s2,$a0($t2)
        sw      $s3,$a1($t2)
        mov     $a0,$t1
        start
        mov     $a0,$t2
        start
          ⋮

        mov     $a0,$at
        mov     $a2,$zero
        mov     $a3,$t1
        join
        mov     $a3,$t2
        join
          ⋮
        end
ll7:
          ⋮
        end
```

Figure 7.18: Java-threaded code compilation.

The `j2n` post compiler transforms Java byte-codes into the MulTEP assembly codes by replacing threaded commands with the MulTEP macros (see Section 7.2) to utilise MulTEP's multithreading instructions (`spawn`, `switch`, `wait` and `stop`). The number of instructions is reduced since `j2n` uses load registers instead of emulating Java's operand stack.

7.4.3 Native Compiler Implementation

A native compiler is required to exploit the full potential of the MulTEP architecture. However, because the project is constrained by the limited research period, a suitable native compiler tool has not yet been implemented. A framework for implementing a MulTEP compiler is, instead, suggested in §9.5.7.

[5]For the simple presentation, `new`, `start` and `join` macros are not expanded in Figure 7.18.

7.5 Summary

To utilise the multithreading features in MulTEP, a couple of software tools are provided. From the low-level software, the MulTEP assembler offers methods for assembling `spawn`, `switch`, `wait` and `stop` multithreading instructions. The MulTEP macros using these instructions were implemented for multithreaded operations to progress a thread through its life cycles: thread creation is supported by the `new` and `start` macros; thread blocking is supported by the `sleep`, `join`, `suspend` and `wait` macros; thread resuming is supported by `resume` and `notify` macros; and thread termination is supported by the `kill` macro.

MulTEP's kernel-level software performs key house keeping functions which ensures the hardware runs smoothly. The system daemon, Thread 0, is provided with the highest priority level. Its data is stored in a reserved address that always remains in the data cache. The reserved address range contains the `active-thread`, `wait-for-join`, `wait-for-timer` and `wait-for-notify` tables. By verifying information within these tables, the daemon adjusts the thread state to be `dead`, `born`, `sleep`, `join`, `wait`, `suspend`, `blocked` or `run`.

Support for high-level software includes the java-to-native post-compiler called `j2n`. This tool is built for benchmarking purpose. It compiles Java-threaded programs into the MulTEP assembly with the help of the `javac` compiler and the MulTEP macros for multithreaded operations.

Part III

Evaluation and Conclusions

Chapter 8

Evaluation and Results

All that we are is the result of what we have thought.

Buddha

8.1 Introduction

Empirical studies were performed to ensure that the MulTEP architecture complies with its proposed design goal:

> *A design and implementation of a high-performance architecture for multi-threading in embedded processors.*

An object-based MulTEP simulator was built for easy implementation and evaluation of the research instead of constructing massively detailed circuitry. The MulTEP simulator was implemented using the hardware design presented in Chapter 6. The architecture was evaluated using the software tools mentioned in Chapter 7: the MulTEP assembler, the MulTEP macros, the `j2n` post-compiler and the thread-0 system daemon.

This chapter is structured as follows: Section 8.2 describes how the MulTEP architecture is simulated. Section 8.3 presents results in terms of processor performance, efficiency of the multithreading mechanisms, real-time response and memory side-effects. Section 8.4 remarks on MulTEP's evaluation and results.

8.2 Simulating the MulTEP Architecture

Practical means for evaluating the MulTEP architecture were investigated prior to the implementation. In my trial experiments, a number of investigations were conducted using hardware design tools, such as the Altera's Max+Plus and Cadence's Verilog-XL. These tools required detailed descriptions for most circuits and interconnection routes. This resulted in inflexible models which were difficult to parameterise in order that a wide range of design choices could be assessed.

Instead of using the hardware description tools, I decided to investigate alternative discrete event simulator-based methodologies. On the one hand, most relevant simulators, such as SimpleScalar [147] and Kaffe [146], are unfortunately hand-written in sequential programming languages that require much time for re-targeting a new architecture. On the other hand, *Architecture Description Languages* (ADLs) [148, 149, 150, 151] focus on modelling instruction set architecture. However, I would like to analyse details at the micro-architecture level. Hence, the features of ADLs are insufficient for my investigation.

Consequently, I chose to create a detailed cycle-accurate simulator for my research. A MulTEP simulator based on C++ object modelling has been specially implemented. The remainder of this section first describes the preparations undertaken prior to the implementation of the simulator (§8.2.1). The MulTEP simulator is then explained in detail (§8.2.2), followed by the analysis of its simulating abilities (§8.2.3).

8.2.1 Pre-simulation Strategies

It was necessary to validate my selected simulation methodology prior to real usage. Two validation procedures were conducted. The first step was to simulate a priority-based tag up/down priority queue and then was validate with its Verilog *Hardware Description Language* (HDL) version for a detailed cycle-accurate comparison. Next, the simulation of a MIPS R3000 5-stage pipeline with only integer support was created and validated with its Verilog HDL version for an abstract architectural level comparison.

Detailed Cycle-accurate Comparison

The tag up/down priority queue is a necessary hardware circuit for scheduling multiple competitive threads in the architecture. The underlying model emulates the design in [8] with a modification to extract a maximum priority instead of a minimum deadline (see detail in Chapter 6). The Verilog HDL code for the modified hardware[1] is presented in Appendix A.1. The simulation output of the Verilog model was monitored in a waveform viewing tool as presented in Figure 8.1.

Likewise, the priority queue was simulated in C++. The `TgSort` class (see Appendix A.2.1) inherits properties from a generic `SimObj` class (see §8.2.2). The `TgSort` object (see Appendix A.2.2) consists of 10 `LR` objects to match with the number of the `LR`

[1]Without a dynamic-priority modification

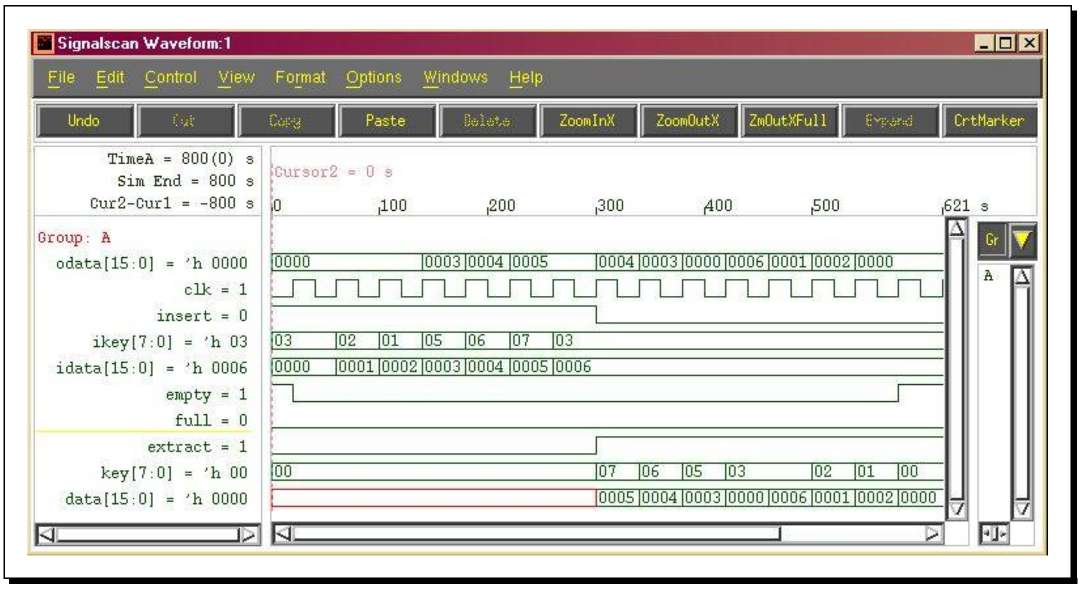
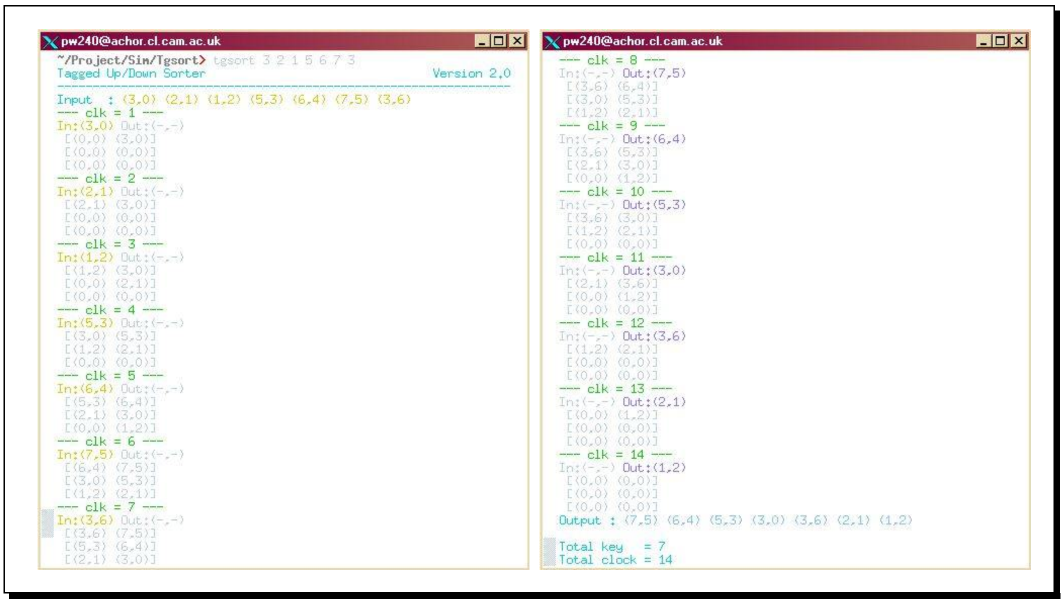


Figure 8.1: Results from simulating `tgsort` in Verilog.

modules in the Verilog model. All event simulations (e.g. clock, inputs) and the necessary execution details in each clock cycle (e.g. the *key* and the *data* values of each `LR` unit) were generated as presented in Figure 8.2.



```
pw240@achor.cl.cam.ac.uk
~/Project/Sim/Tgsort> tgsort 3 2 1 5 6 7 3
Tagged Up/Down Sorter                                      Version 2.0
----------------------------------------------------------------------
Input   : (3,0) (2,1) (1,2) (5,3) (6,4) (7,5) (3,6)
--- clk = 1 ---
In:(3,0) Out:(-,-)
 [(0,0) (3,0)]
 [(0,0) (0,0)]
 [(0,0) (0,0)]
--- clk = 2 ---
In:(2,1) Out:(-,-)
 [(2,1) (3,0)]
 [(0,0) (0,0)]
 [(0,0) (0,0)]
--- clk = 3 ---
In:(1,2) Out:(-,-)
 [(1,2) (3,0)]
 [(0,0) (2,1)]
 [(0,0) (0,0)]
--- clk = 4 ---
In:(5,3) Out:(-,-)
 [(3,0) (5,3)]
 [(1,2) (2,1)]
 [(0,0) (0,0)]
--- clk = 5 ---
In:(6,4) Out:(-,-)
 [(5,3) (6,4)]
 [(2,1) (3,0)]
 [(0,0) (1,2)]
--- clk = 6 ---
In:(7,5) Out:(-,-)
 [(6,4) (7,5)]
 [(3,0) (5,3)]
 [(1,2) (2,1)]
--- clk = 7 ---
In:(3,6) Out:(-,-)
 [(3,6) (7,5)]
 [(5,3) (6,4)]
 [(2,1) (3,0)]
```

```
pw240@achor.cl.cam.ac.uk
--- clk = 8 ---
In:(-,-) Out:(7,5)
 [(3,6) (6,4)]
 [(3,0) (5,3)]
 [(1,2) (2,1)]
--- clk = 9 ---
In:(-,-) Out:(6,4)
 [(3,6) (5,3)]
 [(2,1) (3,0)]
 [(0,0) (1,2)]
--- clk = 10 ---
In:(-,-) Out:(5,3)
 [(3,6) (3,0)]
 [(1,2) (2,1)]
 [(0,0) (0,0)]
--- clk = 11 ---
In:(-,-) Out:(3,0)
 [(2,1) (3,6)]
 [(0,0) (1,2)]
 [(0,0) (0,0)]
--- clk = 12 ---
In:(-,-) Out:(3,6)
 [(1,2) (2,1)]
 [(0,0) (0,0)]
 [(0,0) (0,0)]
--- clk = 13 ---
In:(-,-) Out:(2,1)
 [(0,0) (1,2)]
 [(0,0) (0,0)]
 [(0,0) (0,0)]
--- clk = 14 ---
In:(-,-) Out:(1,2)
 [(0,0) (0,0)]
 [(0,0) (0,0)]
 [(0,0) (0,0)]
Output : (7,5) (6,4) (5,3) (3,0) (3,6) (2,1) (1,2)

Total key   = 7
Total clock = 14
```

Figure 8.2: Results from simulating `tgsort` in C++.

The results from both methods in terms of the order of sort and the number of clock cycles exactly matched. This demonstrates that the C++ simulation is capable of modelling cycle-accurate operations. Furthermore, the output of the C++ simulator are easier to interpret, for instance, the detailed movement of keys and data inside the sorting elements.

Abstract Architectural Level Comparison

The MIPS R3000 integer pipeline was developed in Verilog and C++. The main pipeline is the `PE` object. The original `PE` requires support from five objects, namely: `fetch`, `decode`, `execute`, `memory` and `writeb` units. The Fibonacci algorithm was executed to appraise the correctness of the simulated pipeline. When simulating with Verilog, operations in every stage and the total number of clock cycles for completion were monitored (see an example in Figure 8.3).

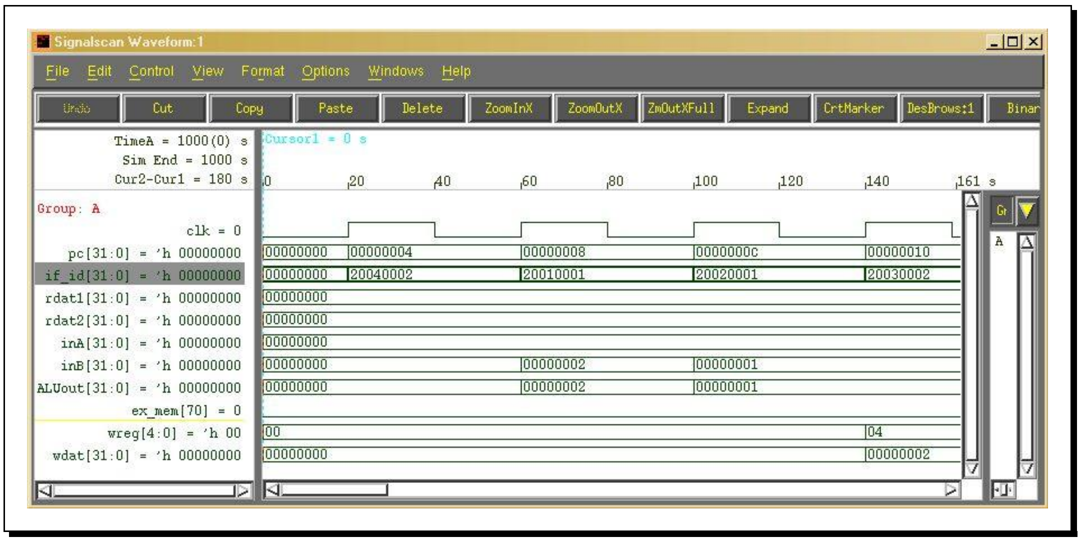

Signalscan Waveform:1
File Edit Control View Format Options Windows Help
Undo Cut Copy Paste Delete ZoomInX ZoomOutX ZmOutXFull Expand CrtMarker DesBrows:1 Binar
TimeA = 1000(0) s
Sim End = 1000 s
Cur2-Cur1 = 180 s
Cursor1 = 0 s
0 20 40 60 80 100 120 140 161 s
Group: A
clk = 0
pc[31:0] = 'h 00000000
if_id[31:0] = 'h 00000000
rdat1[31:0] = 'h 00000000
rdat2[31:0] = 'h 00000000
inA[31:0] = 'h 00000000
inB[31:0] = 'h 00000000
ALUout[31:0] = 'h 00000000
ex_mem[70] = 0
wreg[4:0] = 'h 00
wdat[31:0] = 'h 00000000
00000000 00000004 00000008 0000000C 00000010
00000000 20040002 20010001 20020001 20030002
00000002 00000001
04
00000002

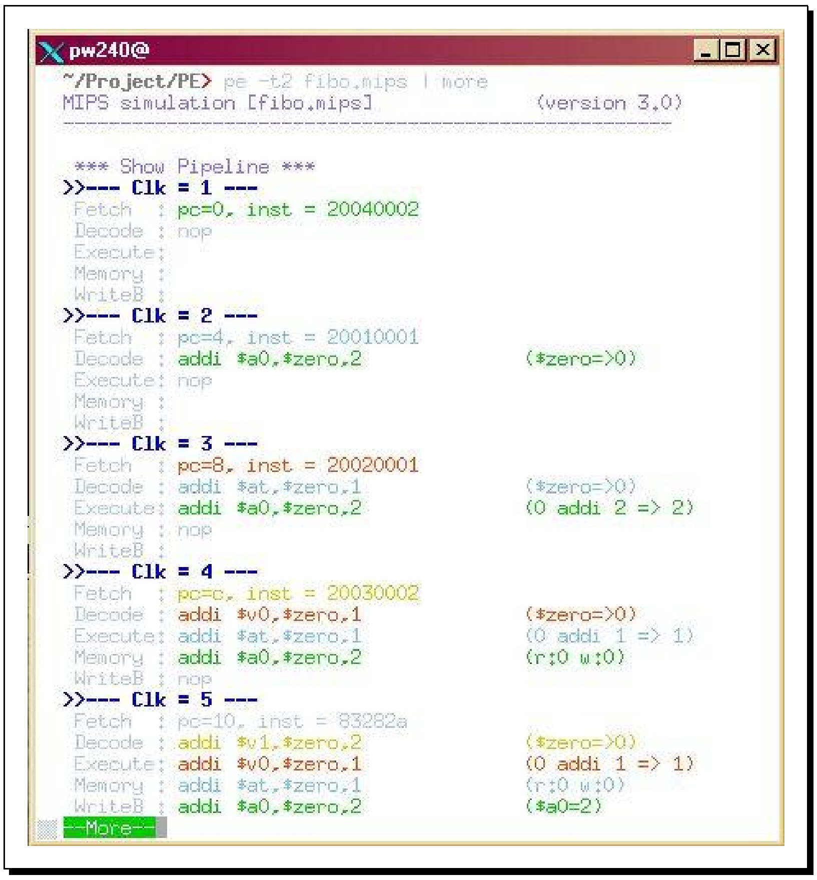
Figure 8.3: Results of the Verilog simulation of the MIPS pipeline.

The results of the Verilog simulation were stored in a file via the `monitor` command in Verilog-XL. Figure 8.4 samples monitored data from 5 top-level pipeline stages in the C++ simulator. The monitored data from both the C++ and Verilog-XL simulations were repeatedly compared with the Fibonacci loop counter n (n ranged from 2 to 1,000). Results from both simulations exactly matched. This confirms that the C++ simulation is capable of accurately modelling an abstract-level architectural design.

8.2.2 The MulTEP Simulator

My cycle-accurate simulator is implemented using C++ objects corresponding to hardware modules with characteristics conforming to the recommendation in the Liberty simulation environment [152]. Figure 8.5 presents an interpretation of the `SimObj` template. The template is used as the generic parent for all hardware units. Its input and output signals are referred by variables in the `InPut` and `OutPut` classes, respectively. The `function()` of an object that inherits from `SimObj` encapsulates both characteristic and underlying mechanism of the unit. The `function()` is activated when the `simclk()` function is invoked.

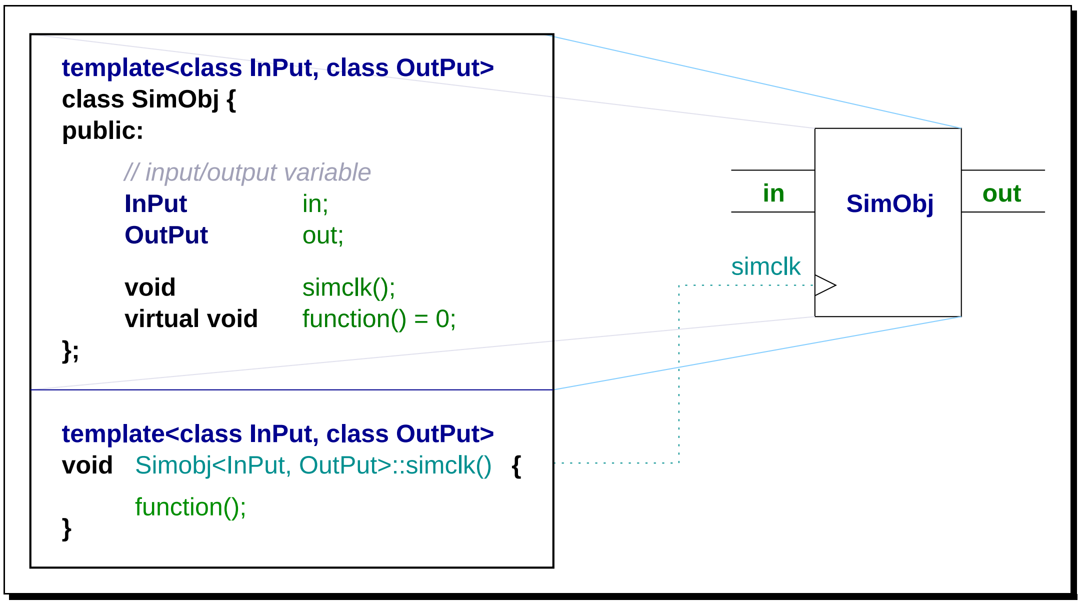
In accordance with the design described in Chapter 6, an illustration of how the embedded objects are structured in the architecture is depicted in Figure 8.6.



```
~/Project/PE> pe -t2 fibo.mips | more
MIPS simulation [fibo.mips]                    (version 3.0)
-------------------------------------------------------------

 *** Show Pipeline ***
>>--- Clk = 1 ---
 Fetch   : pc=0, inst = 20040002
 Decode  : nop
 Execute:
 Memory  :
 WriteB  :
>>--- Clk = 2 ---
 Fetch   : pc=4, inst = 20010001
 Decode  : addi $a0,$zero,2                    ($zero=>0)
 Execute: nop
 Memory  :
 WriteB  :
>>--- Clk = 3 ---
 Fetch   : pc=8, inst = 20020001
 Decode  : addi $at,$zero,1                    ($zero=>0)
 Execute: addi $a0,$zero,2                    (0 addi 2 => 2)
 Memory  : nop
 WriteB  :
>>--- Clk = 4 ---
 Fetch   : pc=c, inst = 20030002
 Decode  : addi $v0,$zero,1                    ($zero=>0)
 Execute: addi $at,$zero,1                    (0 addi 1 => 1)
 Memory  : addi $a0,$zero,2                    (r:0 w:0)
 WriteB  : nop
>>--- Clk = 5 ---
 Fetch   : pc=10, inst = 83282a
 Decode  : addi $v1,$zero,2                    ($zero=>0)
 Execute: addi $v0,$zero,1                    (0 addi 1 => 1)
 Memory  : addi $at,$zero,1                    (r:0 w:0)
 WriteB  : addi $a0,$zero,2                    ($a0=2)
--More--
```

Figure 8.4: Results of the C++ simulation of the MIPS pipeline.



```
template<class InPut, class OutPut>
class SimObj {
public:
        // input/output variable
        InPut           in;
        OutPut          out;

        void            simclk();
        virtual void    function() = 0;
};

template<class InPut, class OutPut>
void   Simobj<InPut, OutPut>::simclk()  {
         function();
}
```

Figure 8.5: An interpretation of the SimObj definition.

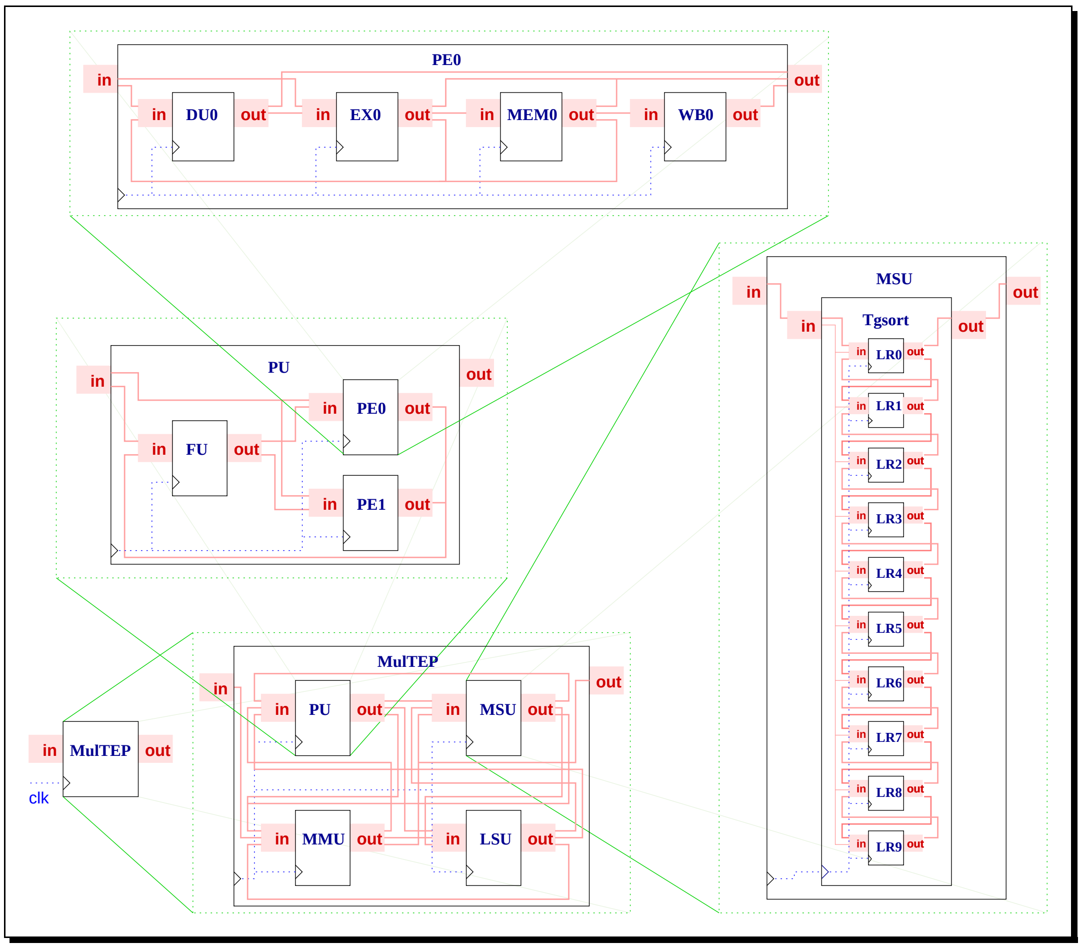


Figure 8.6: The structure of the sample embedded objects in MulTEP.

From the top level, MulTEP consists of four main units, namely the *Processing Unit* (PU), the *Multithreading Service Unit* (MSU), the *Memory Management Unit* (MMU) and the *Load-Store Unit* (LSU). The MSU is embedded with the *Tag up/down priority queue* (Tgsort) for thread scheduling. Tgsort unit also has ten LRs sorting elements embedded inside. The PU comprises the *Fetch Unit* (FU) and two processing elements, each of which contains the subpipeline of four objects, ranging from the *Decode Unit* (DU) to the *Write Back unit* (WB).

For the simulation process, one invocation of the simclk() function virtually generates one `clk` pulse. The pulse is used to activate all embedded modules and to enable all data to be transferred simultaneously.

8.2.3 MulTEP Simulation Analysis

The simulation of MulTEP was analysed using three criteria: dynamic multithreading behaviour, data movement in the critical path and synchronisation of multiple threads.

Dynamic Multithreading

For dynamic multithreading analysis, three threads with different characteristics were issued to the system. The PEs, L0 cache and two multithreading-service units were monitored as depicted in Figure 8.7.

In this example, the priority of Thread 1 was set to 128 to simulate a high-priority interrupt event raised by a kernel system. Thread 2 and Thread 3 represented two normal threads with different run lengths and low priority levels. At time `T5`, a stall occurred in Thread 1. The system switched to Thread 3 since its instruction set and execution context had already been fetched and loaded into the processing unit, respectively. At time `T6`, the stall of Thread 1 was resolved. Thread 1 with address `A1+15` was immediately sent to the processing element ready queue and then was consequently dispatched to the PE0. This event interrupted the execution of Thread 3 because it has the lowest priority in the processing unit.

The context-switch mechanism reserved the interrupted location of Thread 3 in the ready queue for its later execution. When the PE1 was available (at time `T9`), the context of Thread 3 was switched in. The execution carried on until the completion of its block at time `T12`[2]. Because the processing unit found out that there was no other thread to be scheduled in the queue, execution of Thread 3 continues on the next instruction block[3]. This demonstration illustrates that the architecture supports dynamic behaviours of multiple threads including unexpected events, such as stalls. Priority-based scheduling determines the order in which contexts are switched.

[2]There is no other interruption during the execution of Thread 3 from `T9` to `T12`.

[3]The continuation is allowed if and only if L0-cache has already pre-fetched its next instruction block such as the complete of preparation in time `T6` in the example.

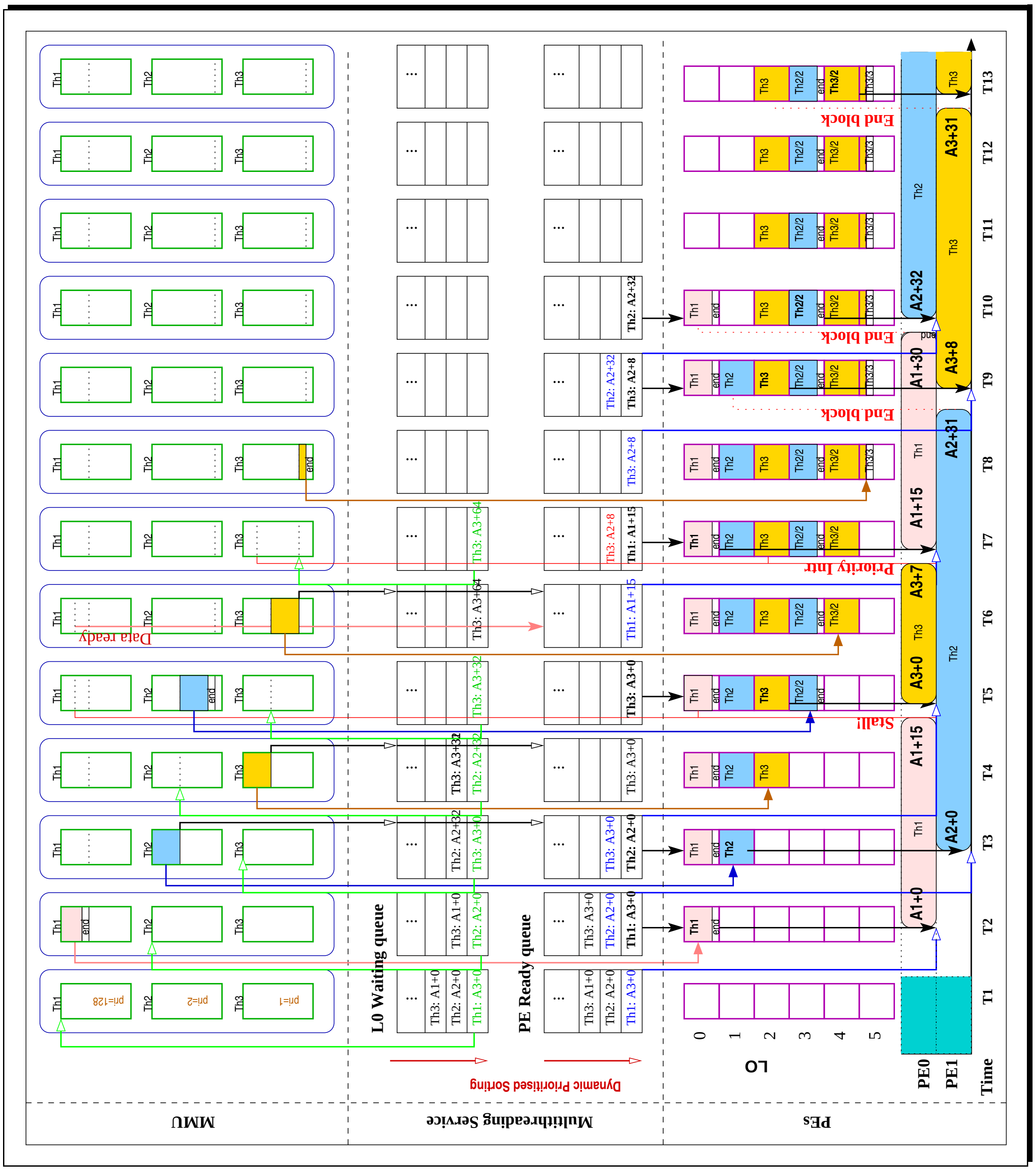


Figure 8.7: The illustration of nanothreading in the system.

Data Movement in the Critical Path

Thread synchronisation in MulTEP is performed using store commands. Hence, data movement from such commands are crucial to the system. Because of this, the load-store unit plays an important role in handling thread synchronisation over and above its duty to alleviate the data-bandwidth bottleneck between the other units (see Section 6.4 for details of how bottleneck is alleviated).

Data movement in the critical path in the load-store unit have been monitored in the simulation during the implementation process. Figure 8.8 presents an example when both processing elements issue different store packages to the load-store unit[4]. The example for PE1 is the case when a thread required the content from a register whose data is still being loaded. When the processing unit is waiting for the data from the load-store unit, context switch occurs[5]. In such a case, a store package embedded with the invalid data whose source register is $r2 was released. The invalid status at the data source indicated that the field was waiting for load-return data.

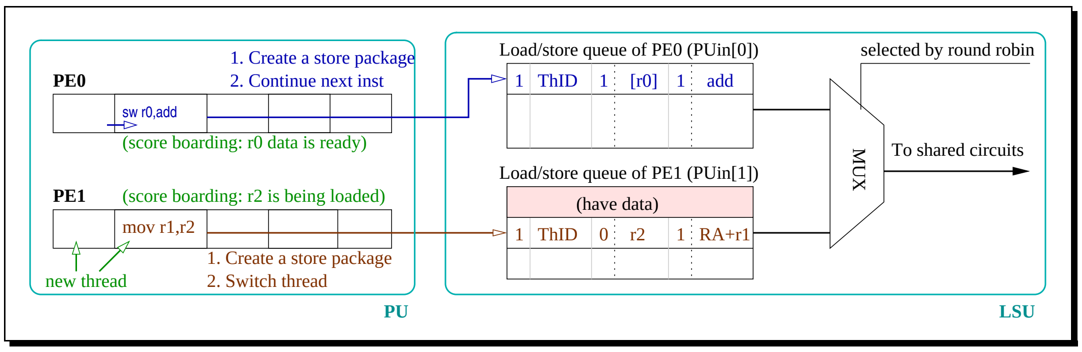

PE0
1. Create a store package
2. Continue next inst
sw r0,add
(score boarding: r0 data is ready)
PE1
(score boarding: r2 is being loaded)
mov r1,r2
1. Create a store package
2. Switch thread
new thread
PU
Load/store queue of PE0 (PUin[0])
1 ThID 1 [r0] 1 add
Load/store queue of PE1 (PUin[1])
(have data)
1 ThID 0 r2 1 RA+r1
selected by round robin
MUX
To shared circuits
LSU

Figure 8.8: The PEs issue different store packages to the MSU.

Figure 8.9 presents a situation when load-return data arrived. Thereby, the status of the field in the package was changed to be valid.

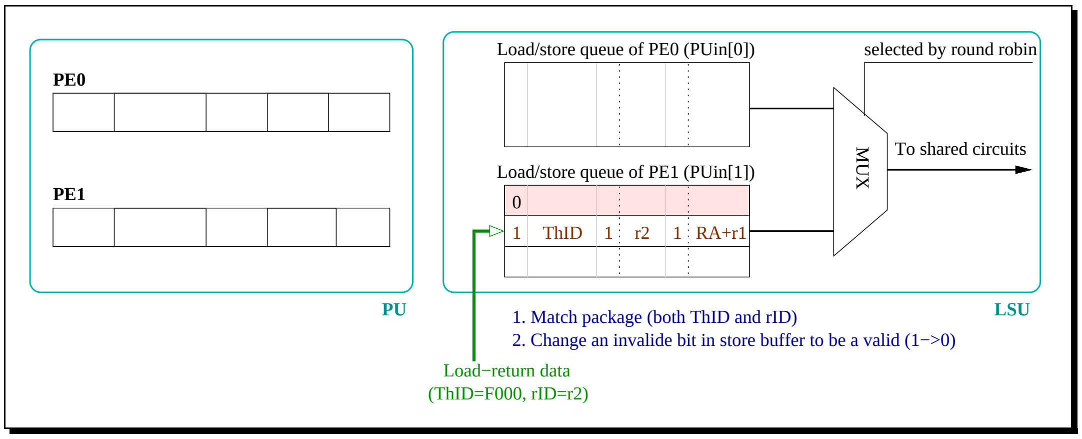

PE0
PE1
PU
Load/store queue of PE0 (PUin[0])
Load/store queue of PE1 (PUin[1])
0
1 ThID 1 r2 1 RA+r1
selected by round robin
MUX
To shared circuits
LSU
1. Match package (both ThID and rID)
2. Change an invalide bit in store buffer to be a valid (1−>0)
Load−return data
(ThID=F000, rID=r2)

Figure 8.9: The required loaded data updates an invalid stored package.

[4]Information are extracted from the operation of two LL7 threads (see Section §8.3.2)
[5]`move` is a special store operation if its operand is indicated invalid in a scoreboard.

Owing to the analysis of data movement in the load-store unit, I had found that the size of the queue is one of the most crucial factors which directly effects the processing performance. This is because a decrease in queue size directly increases the data-bandwidth bottleneck.

An assessment of queue size on performance was undertaken using run-times as the metric. Run-times for a particular queue size were normalised against run-times for an infinite queue size (i.e. where the queue is no longer a bottleneck). Figure 8.10 depicts the normalised run-times of three *Livermore Loop 7* (LL7) workloads as presented in Figure 8.11 with different numbers of loop iterations (n is equal to 20, 30 and 50).

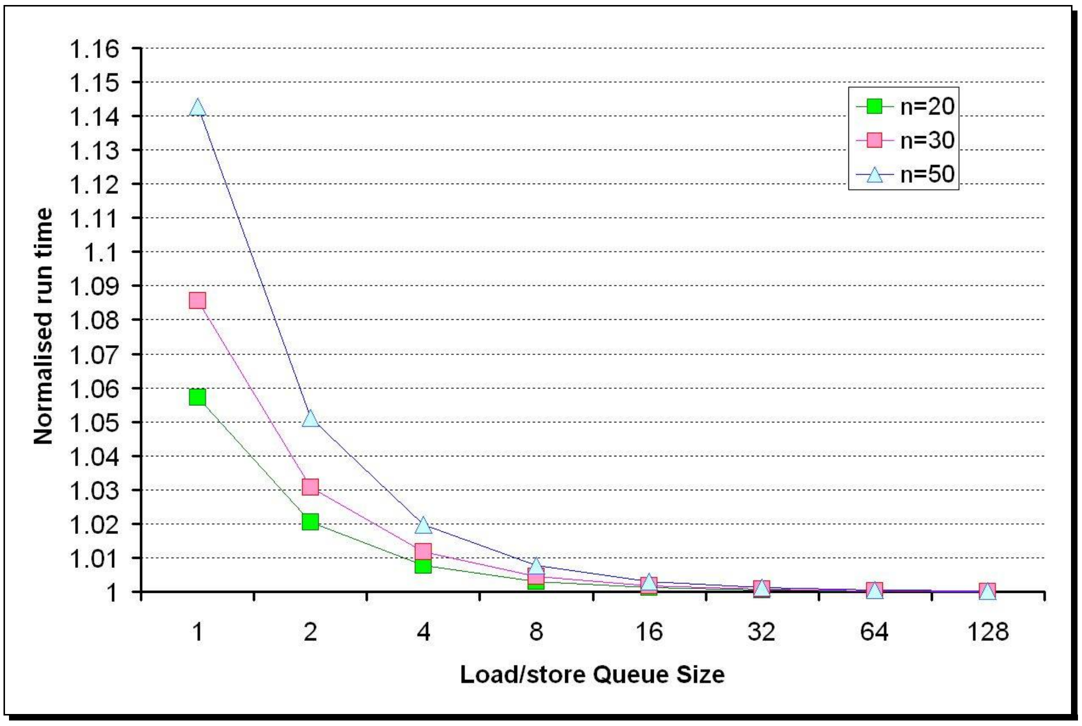


Figure 8.10: The normalised run time with different queue size.

Figure 8.10 shows that the data-bandwidth bottleneck is reduced when the queue size increases. When the queue size is at least eight, the difference of all normalised runtimes become less than 1%. Load/store input queue packages of size eight are used and conform to the sizes of the operation analysis of the load-store unit in Section 6.4.

Thread Synchronisation

Synchronisation and communication are two of the three crucial features in multithreaded design that need to be evaluated[6]. The underlying synchronising mechanicsm that support extensive communication of data and control were validated before benchmarking the system. To assess thread synchronisation, multithreaded version of Livermore loop 7 [4] (see Appendix B.3) was created with a large number of inter-loop dependencies resolved using inter-thread synchronisation. Results are shown in Figure 8.11.

[6]The other is a scheduling mechanism, which has already been analysed in §8.2.1

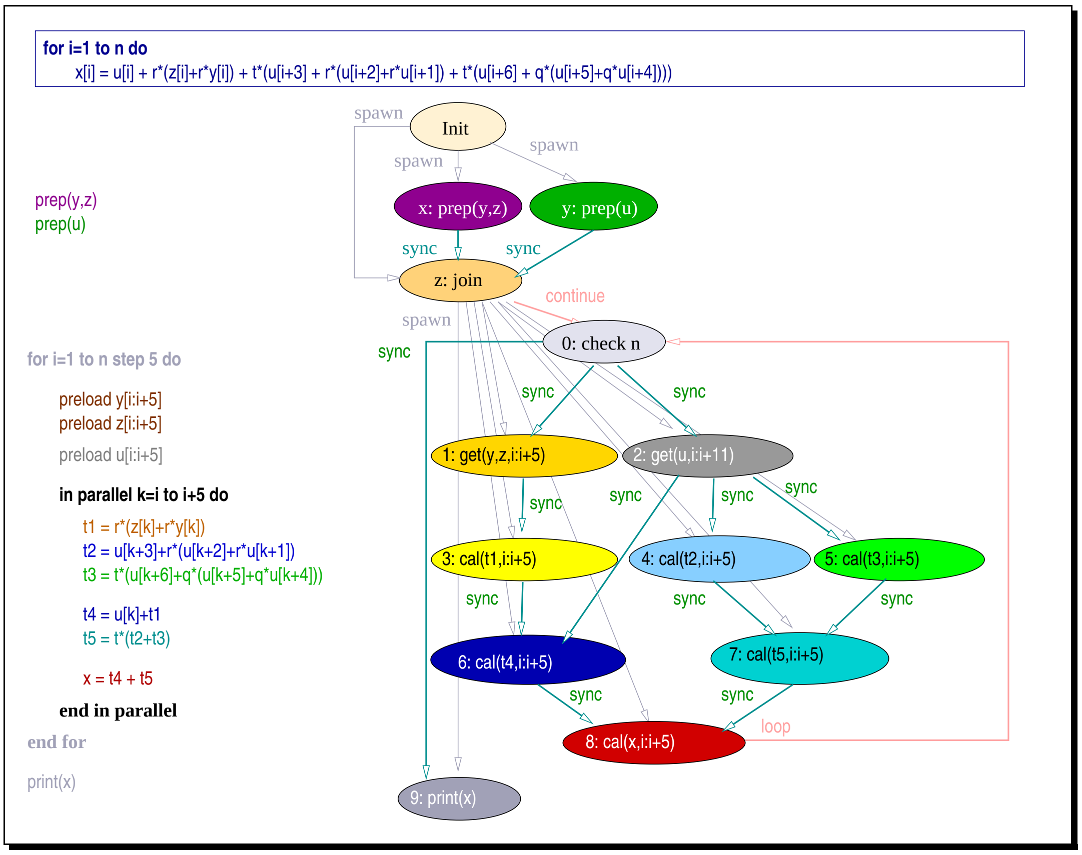


Figure 8.11: The multithreading version of Livermore Loop 7.

The main thread initially spawned 3 sub-threads, namely Threads x, y and z, which inject data. Thread z then spawns the main loop which consists of Threads 0 to 9. Thread 0 handles the loop termination checking. Threads 1 to 8 represent calculation of six unrolled-loops. These threads were separated with respect to their shared data and independent characteristics. When the calculation was accomplished, Thread 9 was activated to summarise the results.

Concurrent execution of more than one LL7 programs require extensive data and control synchronisation. Figure 8.12 illustrates the execution of two LL7 programs. The intra-process synchronisation is represented by a pink line on the two top diagrams. The inter-process synchronisation on shared resources, such as the execution contexts and the processing elements, is reflected in the form of the coherency of resource utilisation (see two bottom diagrams of Figure 8.12).

Evaluating the synchronisation of multiple threads with the LL7 benchmark demonstrates the correctness of the synchronisation mechanism. The monitored data indicates that the context switches in the processing unit and the changes in context were correct. Their behaviour matches the dynamic priority and the context-switching state diagram.

8.3 MulTEP Simulation Results

This section presents simulation results of the MulTEP architecture. The section starts with an introduction of the selected benchmarks (§8.3.1), followed by the MulTEP evaluation results in terms of the processor performance (§8.3.2), the efficiency of multithreading mechanisms (§8.3.3), the real-time response (§8.3.4) and the memory side-effects (§8.3.5).

8.3.1 Selected Benchmarks

There are two areas where the MulTEP system needs to be evaluated. The first area is its performance when the multithreading mechanisms are utilised. To support this, a benchmark with a high-degree of thread level parallelism is required. Because of this, the multithreading version of Livermore Loop 7 (see §8.2.3) was selected. The calculation characteristic of Livermore Loop 7 is also very close to some highly parallel embedded applications such as the hidden markov model and a neural network model, which are generally used for image and speech recognition.

The second area is to evaluate MulTEP with different applications. Hence, a set of wellknown benchmarks with variety functional characteristics becomes the area of interest, especially those that have been selected by the majority of multithreaded architectures. Based on this, six programs in the standard SPEC CPU2000 benchmark are selected. These programs represent different types of applications covering both integer and floating point calculation (see Table 8.1).

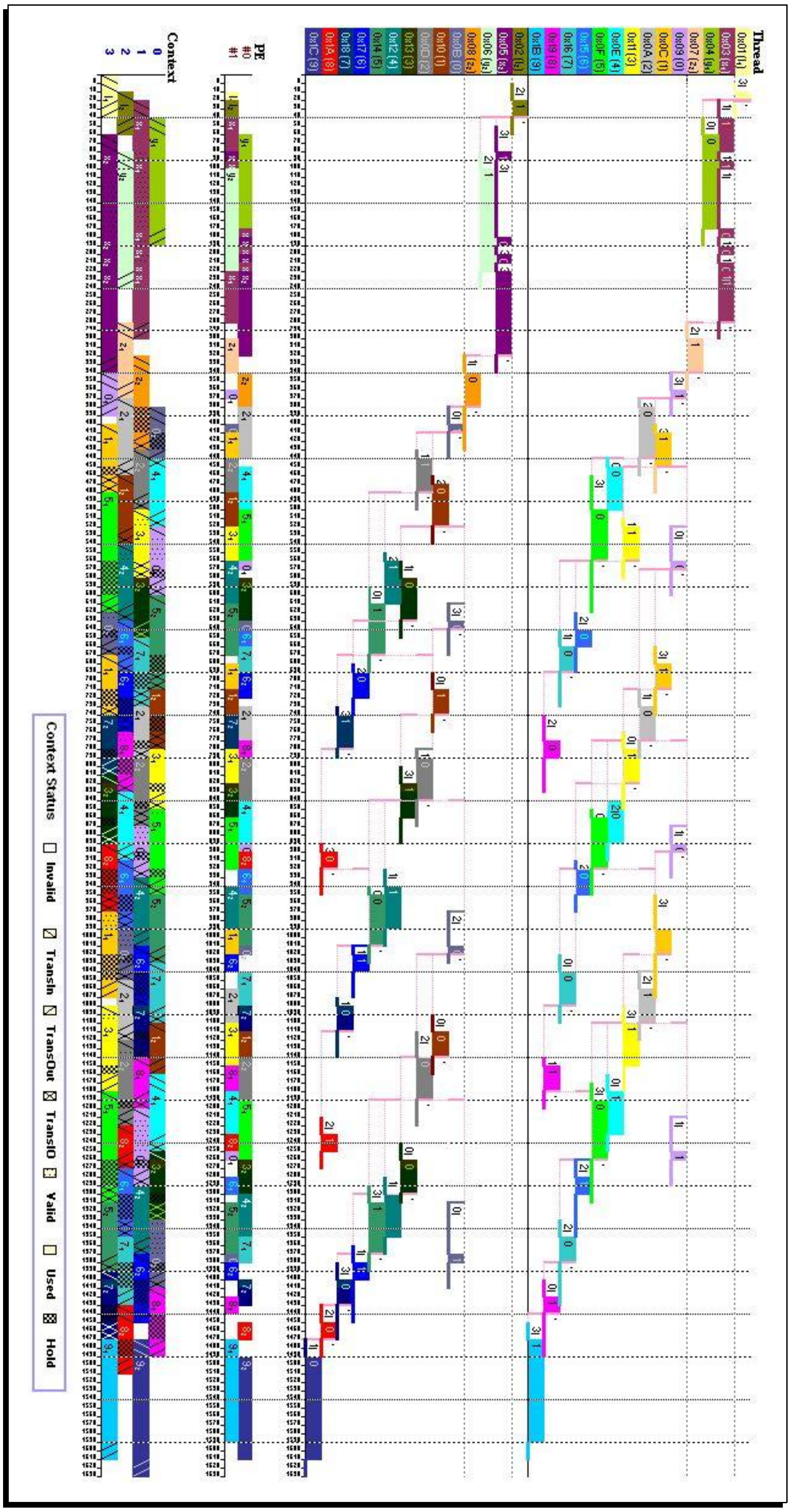


Figure 8.12: The multithreading results of 2 LL7 programs in MulTEP.

8.3.2 Processor Performance

Two set of evaluations were performed to evaluate MulTEP performance. The first set is an evaluation when MulTEP was supplied with multithreaded workloads. The second set is an evaluation when MulTEP was supplied with a number of single-threaded workloads from the SPEC CPU2000 benchmark suite.

Performance of Multithreaded Workloads

The first set of evaluations focuses on the system performance of MulTEP when operating with multithreaded workloads. The objective of this multithreading appraisal is to answer the following questions:

1. How well can MulTEP perform over a baseline single-threaded processor?
2. How much can MulTEP benefit from multithreaded code?

I used a 2-issue MIPS R3000 with static scheduling as baseline integer processor. For system benchmarking, two versions of the standard LL7 algorithm with 1,000 iterations were implemented in Java. The first version is single-threaded code (see Appendix B.1) and the second version is multithreaded code (see Appendix B.2). The multithreaded code spawns 5 sub-threads, each of which separately handles 200 iterations of the standard LL7 algorithm.

Three experiments were conducted. The first experiment monitors the run-times of the baseline processor operating with the different numbers of single-threaded workloads (coded `SB`: *Single-threaded code on the Baseline CPU*). The second experiment focuses on the run-times of MulTEP operating with the same number of single-threaded workloads (coded `SM`: a *Single-threaded code on MulTEP*). The third experiment examines the run-times of MulTEP, operating with some number of multithreaded codes (coded `MM`: *Multithreaded code on MulTEP*).

To answer the first question, Figure 8.13 displays the speedup of the `SM` and `MM` based on the `SB`. The workload on an x-axis represents the number of LL7 programs excluding kernel thread (thread 0). The figure shows that MulTEP offers a speedup (i.e. 24.29% on average) when it is supplied with more than 2 single-threaded workloads[7]. When the number of threads is greater than the number of the processing elements (2), a significant trend of performance improvement is present (i.e. 31.23% average speedup). This reflects that MulTEP tolerates data transfer latencies when it is able to schedule additional workloads to achieve significant speedup. When the number of programs is less than that of the processing elements, the multithreaded performance of MulTEP is worse than the baseline experiment. This is because the program length of multithreaded code plus thread-0 are longer than the program length of single-threaded code[8].

Figure 8.13 illustrates that the highest speedup (60%) is obtained when executing three single-threaded programs. However, as depicted in Figure 8.14, some processing

[7]The number of single-threaded workloads are more than the number of the PEs.

[8]The length of the thread 0 approximately equals the length of the LL7 code.

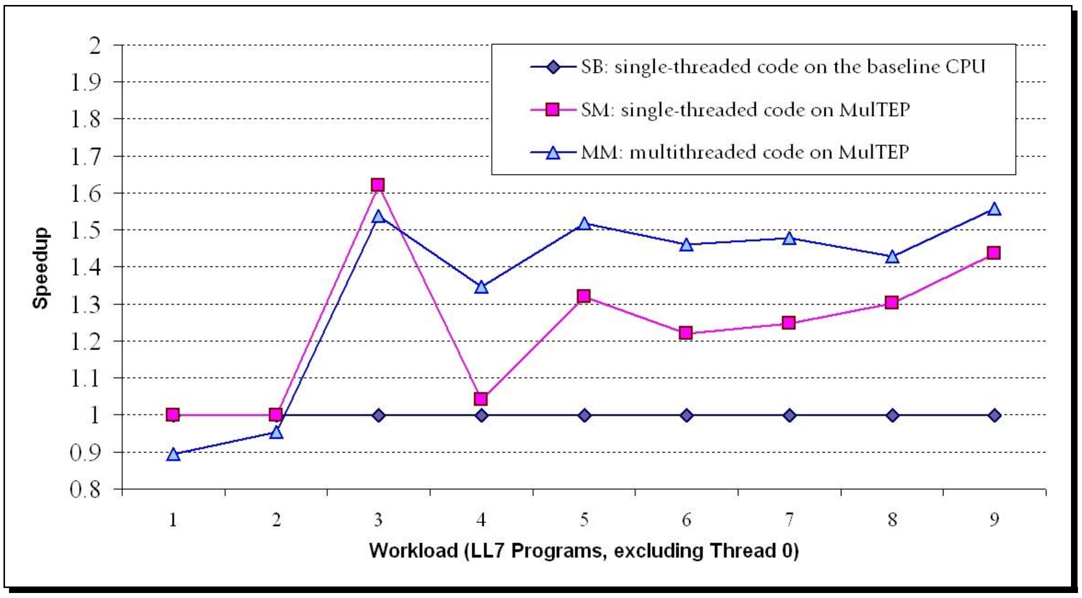


Figure 8.13: Speedups of MulTEP with single-/multi-threaded codes.

elements and some execution contexts are partly utilised when the number of threads is less than four (i.e. the number of the execution contexts).

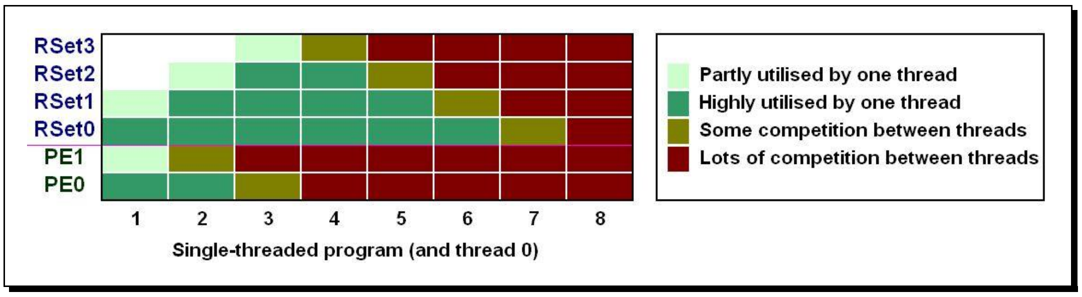


Figure 8.14: PE/Rset Utilisation of single-threaded code.

To answer the second question, Figure 8.15 depicts the speedup of the `MM` run-times compared with the `SM` run-times. These speedups reflect the benefits of the multithreading features in MulTEP if the code is supplied with MulTEP's multithreading instructions (i.e. `spawn`, `switch`, `wait`, `stop`). The figure presents that the operation of multithreaded codes offers 16% average speedup[9] when the number of workloads is greater than three.

However, the performance of multithreaded code is worse than the performance of single-threaded code when the number of workloads is no more than three. There are two reasons for this. The first reason is that the run length of multithreaded code is 40% longer than the run length of single-threaded code due to overheads associated with handling Java's threads. The second reason arises when the number of threads is less than the number of execution contexts (4) since executing from one to three threads can be conducted without much competition for execution contexts. Thus, the execution from

[9]There is a decreasing trend in the multi-threading features speedup. Further analysis to find out the root cause of this decreasing trend is conducted in 8.3.5.

one to three single-threaded codes can avoid the outside-PU context-transferring overhead (i.e. 7 to 16 cycles per transaction).

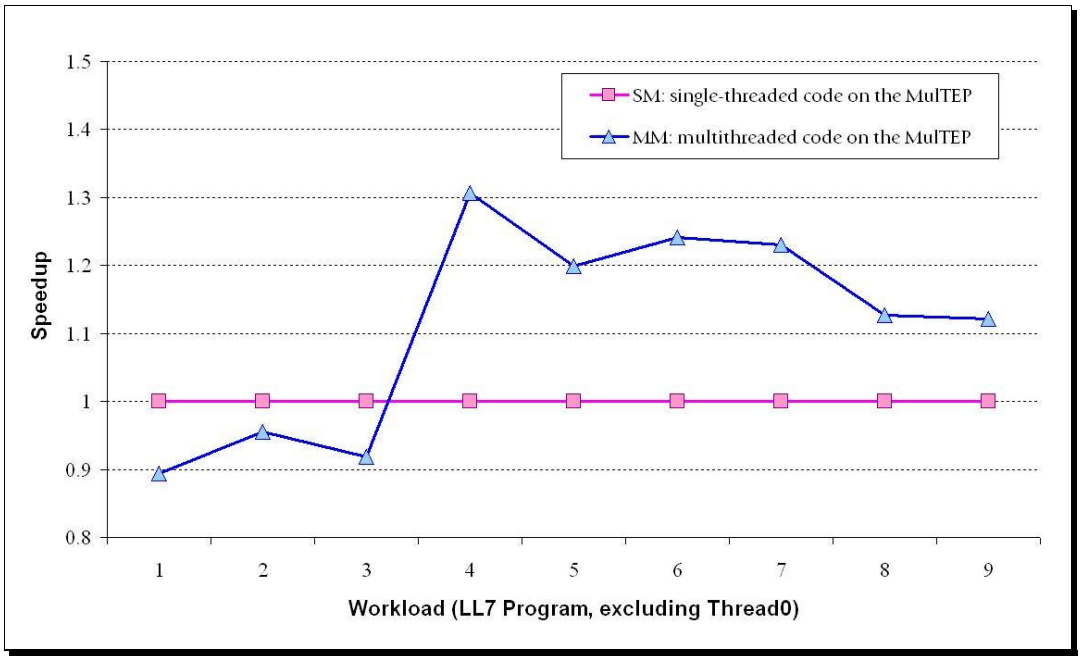


Figure 8.15: Speedup from multi-threading features in MulTEP.

Performance of Single-threaded Benchmarks

Although the previous section presents performance improvement of a single-threaded LL7 when the number of workload is more than the number of processing elements, it is insufficient to draw conclusions based on one program. Therefore, additional experiments were conducted using the SPEC CPU2000 suite. Table 8.1 lists the selected benchmarks from the suite for system evaluation[10]:

Program	Description	Parameters
`164.gzip`	Data compression utility	`smred.log 1`
`175.vpr`	FPGA circuit routing	`test small.arch.in -route_only ..`
`181.mfc`	Minimum cost network flow solver	`lgred`
`197.parser`	Natural language processing	`2.1.dict -batch < lgred.in`
`179.art`	FP neural network simulation	`-stride 5 ... -objects 1`
`188.ammp`	FP computational chemistry	`< lgred.in`

Table 8.1: Selected single-threaded benchmarks from SPEC CPU2000.

SimpleScalar [147], a well-known tool in the architecture community, is used as a baseline model for the evaluation. The SimpleScalar PISA model, a MIPS-like model, was augmented with similar memory hierarchy and functional units as that for the MulTEP architecture as shown in Figure 8.16.

[10]Only the base version is used (not the peak/optimised version)

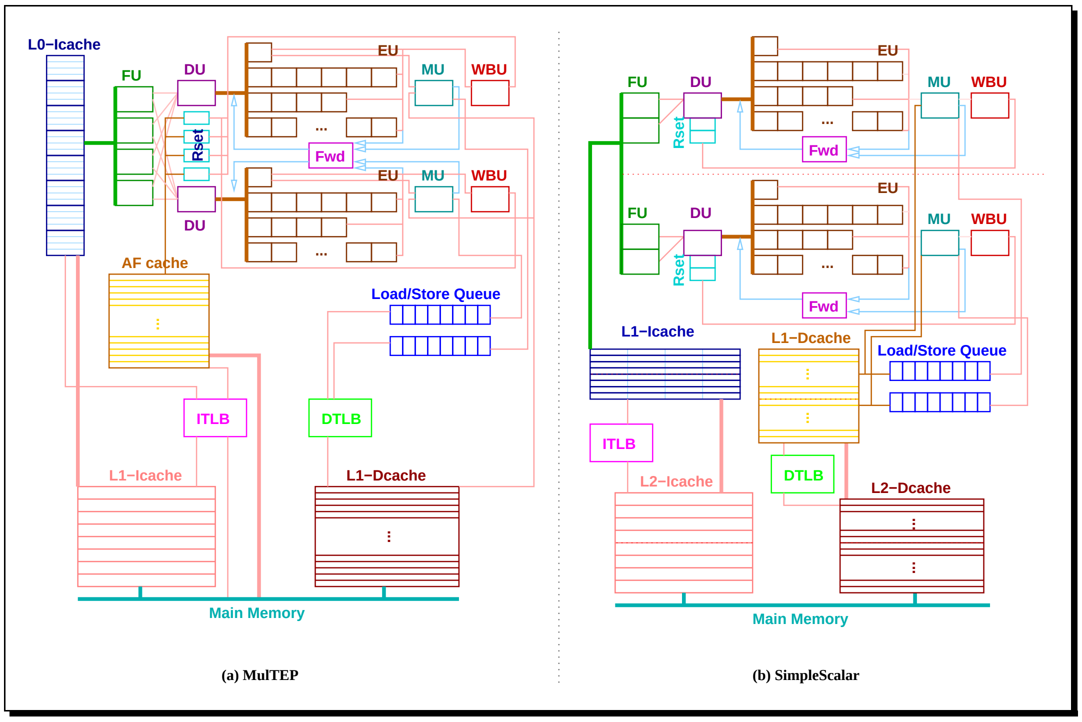


Figure 8.16: Similarity in memory hierarchy and functional units.

PISA binaries[11] of the benchmark programs listed in Table 8.1 were obtained from the MIRV project [153]. The `sstrix-objdump` tool was used to disassemble the PISA objects and the `mipsel-as` assembler, obtained from the Linux VR project [154], repacked them into MIPS objects for the simulation. The MinneSPEC data [155] were used as inputs to the benchmark programs.

The benchmarks were executed on both the SimpleScalar's `sim-outorder` (see Appendix C.2) and the MulTEP simulations. Figure 8.17 presents the run-time speedup of MulTEP compared with SimpleScalar. MulTEP performs worse than SimpleScalar when the number of threads is less than the number of the processing elements. This is because MulTEP needs to execute additional instructions and it cannot extract instruction level parallelism. A few instructions are required for multithreading and Thread-0 imposes some overhead[12].

MulTEP offers 12% average speedup and 18% average speedup when the number of threads is equal to or greater than the number of the processing elements. The benchmark results demonstrate that performance is effectively gained when the number of threads is sufficient for multithreading.

Figure 8.18 presents the speedup of MulTEP's performance based on the performance of two enhanced superscalars with the number of threads ranges from 1 to 6. The enhanced superscalar (simulated by SimpleScalar's tool `sim-outorder`) is configured with the parameter in Table 8.2[13].

[11]PISA instructions are close to MIPS instructions but encoded with different format
[12]Thread 0's time interval is 10,000 cycles in the MulTEP simulation
[13]The enhanced superscalar is inspired by architectures in MICRO-35 Conference.

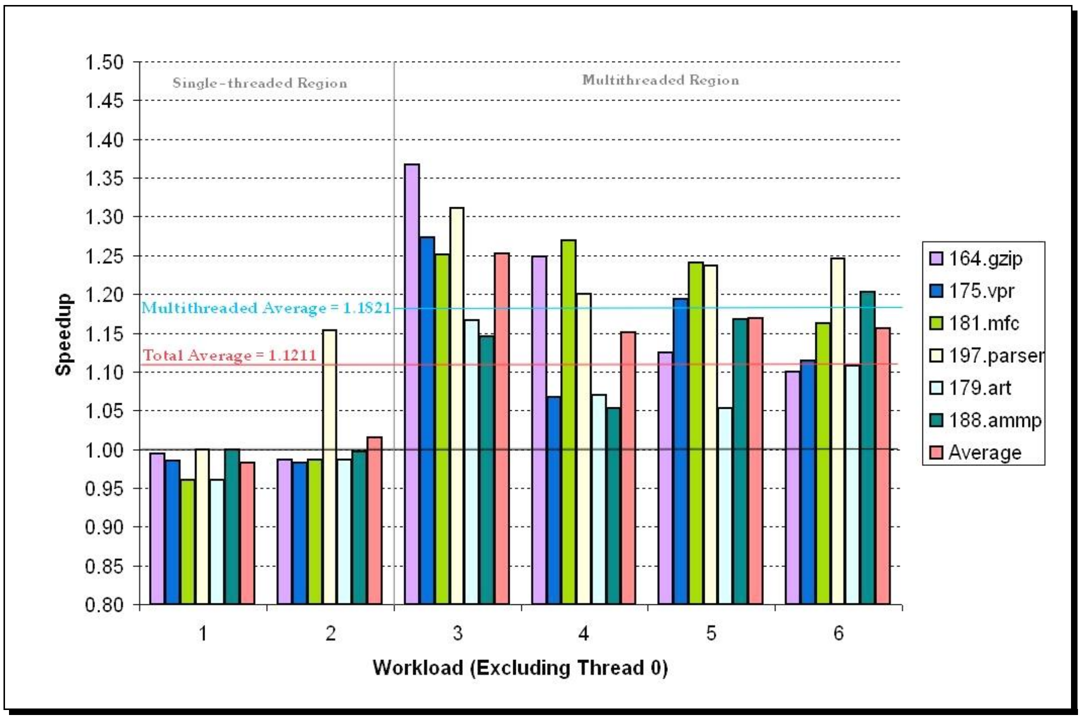


Figure 8.17: The speedup of MulTEP based on the SimpleScalar simulation.

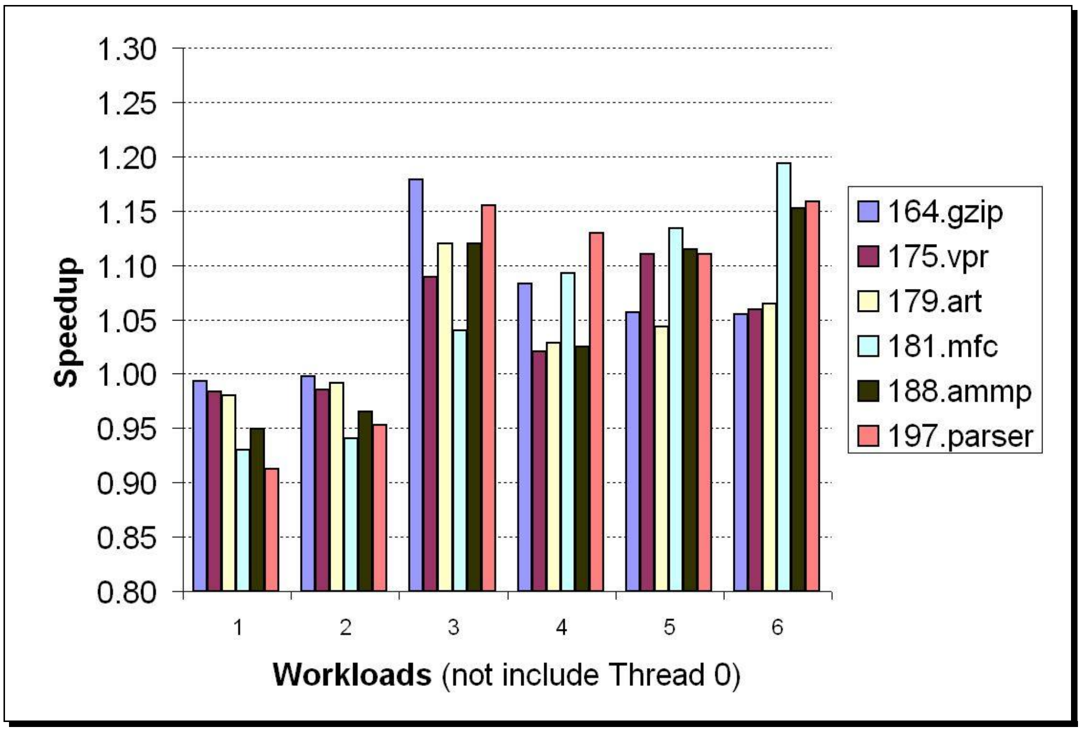


Figure 8.18: The speedup of MulTEP based on 2 out-of-order superscalars.

L1-Icache	1KB of 8 sets, fully associative, 1 cycle
L2-Icache	16KB of 16 sets, direct-mapping, 5 cycles
L1-Dcache	16KB of 128 sets, 4-way set-associative, 5 cycles
Cache Miss Penalty	200 cycles
Issue to Decoder	16 instructions
Register Renamer	16 instructions
Branch Predictor	128K-entry global & share, 64K-entry local
Branch Target Buffer	4K entries
Return Stack	32 entries
Mis-prediction Latency	20 cycles
Out-of-order Window	512 entries

Table 8.2: The configuration of the enhanced superscalar.

The overall speedup over the out-of-order superscalar is lower than the previous speedup results. The reduced speedup value is due to a better performance of the out-of-order superscalar which reflects in a degraded speedup when the number of threads is only 1 or 2. However, MulTEP is capable of offering 9.7% average speedup when the number of threads is equal to or exceeds the number of the processing elements.

8.3.3 Efficiency of Multithreading Mechanisms

The previous section indicates that the performance can be improved when there are sufficient threads to allow data access latencies to be hidden by rescheduling. This section evaluates the efficiency of the multithreading mechanisms: pre-fetching, pre-loading and colour-tagging.

The efficiency of the multithreading mechanisms can be measured by looking at the percentage of zero-cycle context switches. Figure 8.19 presents the zero-cycle overhead ratio when the MulTEP system was benchmarked with the number of threads ranging from 1 to 6.

The results of Figure 8.19 illustrate that the percentage of zero-cycle context switches dramatically increases when the number of workloads increases from 1 to 3. The zero-cycle context-switching percentage is more than 90% when the number of workloads is at least four, which is the number of execution contexts.

The high percentage of zero-cycle context switches indicates that the pre-loaded execution contexts are available in the system prior to the usage. This is because the pre-loading mechanism executes in accordance with the scheduling decision indicated by the multithreading service unit.

Instruction pre-fetching in the L0-Icache also benefits from the scheduling information indicated by the multithreading service unit. Figure 8.20 illustrates the hit ratio of the L0-Icache when MulTEP was benchmarked with different workloads. The results show a 99.57% average hit ratio. The results reflect a high efficiency of the pre-fetching mechanism when compared to most conventional architectures where the average hit ratio

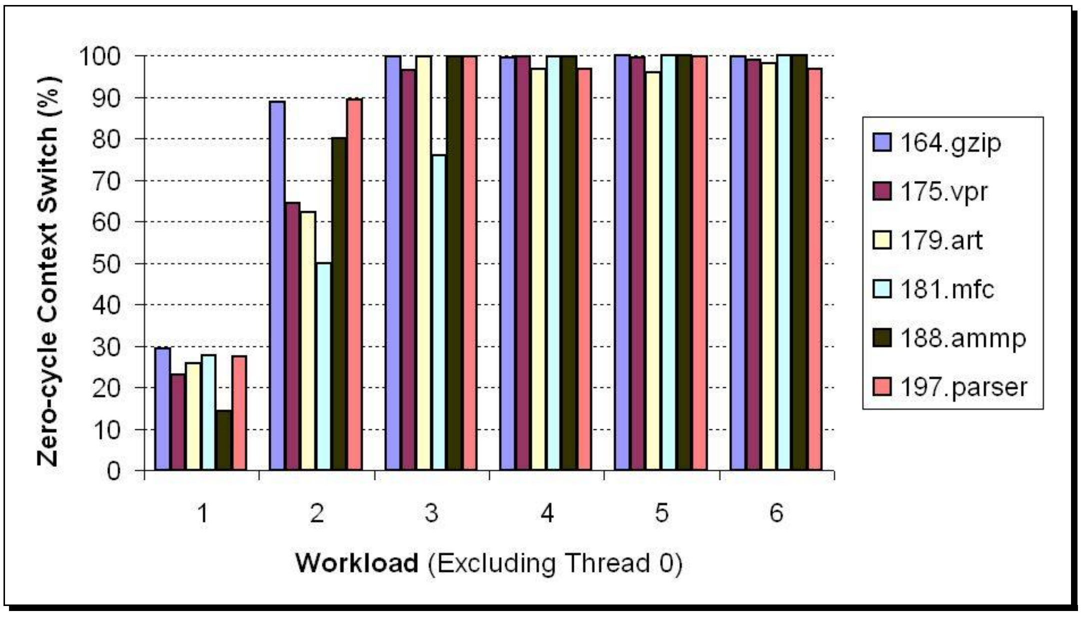


Figure 8.19: The percentages of zero-cycle context switches.

on a similar instruction cache can provide only 98% [139].

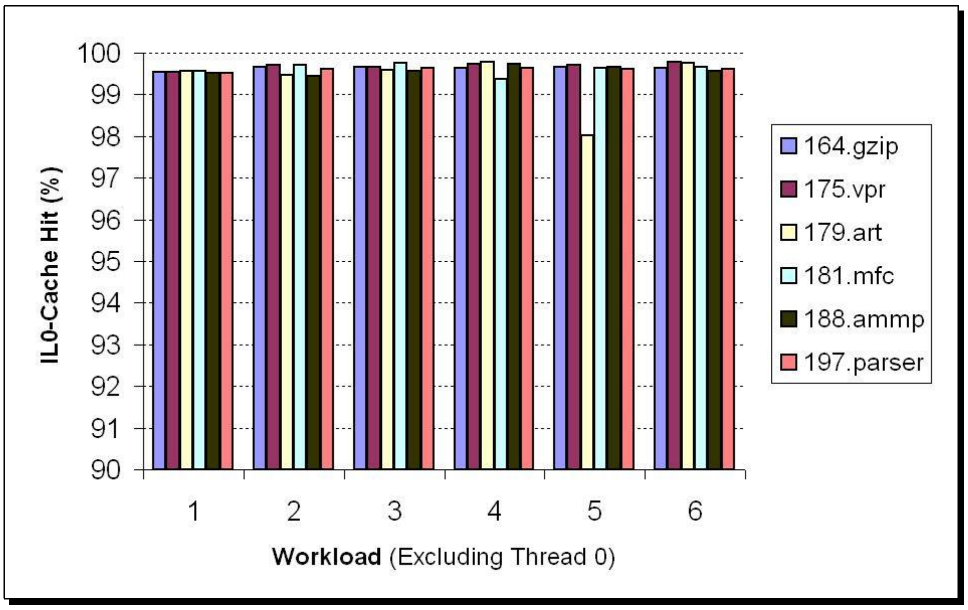


Figure 8.20: Hit ratios of the L0-Icache with various numbers of workloads.

8.3.4 Real-Time Response

A thread is associated with a priority. A priority can be assigned by either real-time embedded languages or a real-time kernel (see Chapter 7). The evaluation of MulTEP's real-time feature was based on the dynamic scheduling capability. In the evaluation, the start and completion time of each thread were analysed to determine whether they conformed to their priorities or not.

Observations were firstly made in §8.2.3. As displayed in Figure 8.12, the upper set of threads which have a higher priority are handled earlier than the lower set of threads in every decision making period. To extend the real-time analysis to the SPEC CPU2000 benchmark, different priorities were associated with six benchmarking workloads as illustrated in Figure 8.21.

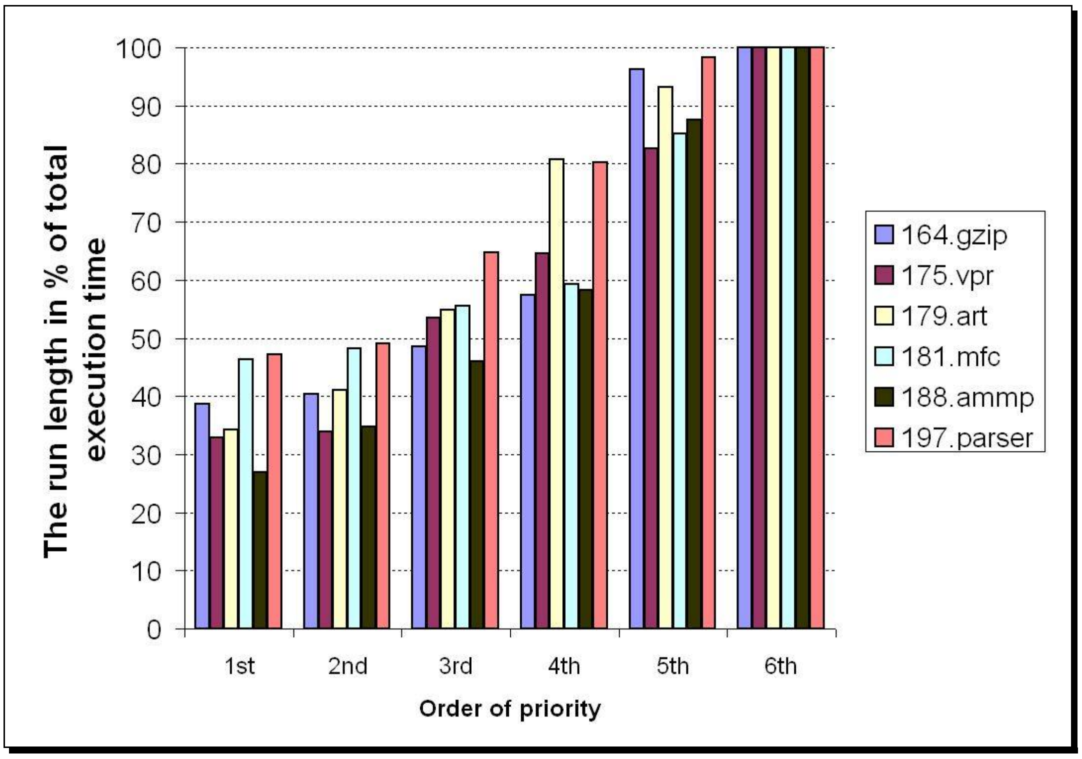


Figure 8.21: The run length in % of execution time.

The priority (p) is 128 for the first thread and reduced by $\frac{p}{n}$ for the following spawned thread (n represents the total number of threads). A thread with a higher priority is serviced and completes its execution prior to a thread with a lower priority. This result demonstrates that the scheduler supports a priority-based scheduling policy. This scheduling policy matches many of those present in real-time OSs (see Chapter 7).

8.3.5 Side-effect Evaluation

One drawback of multithreaded operation is the side effect on other competitive resources like the caches. The results in §8.3.2 present a slightly decreasing trend in performance when the number of threads increases, even though there are a sufficient number of threads to provide concurrency. The cause of the decrease is suspected to be a result of thread competition in both first level instruction and data caches.

The suspicion is based on the fact that L1-Icache and D-caches are the first resources shared by multiple threads without the scheduling information from the multithreading service unit[14]. Figure 8.22 presents the hit ratio in the L1-Icache[15]. The figure depicts decreasing hit ratios when the number of threads increases, especially when the number of threads is equal to or greater than the number of the execution contexts in the processing unit.

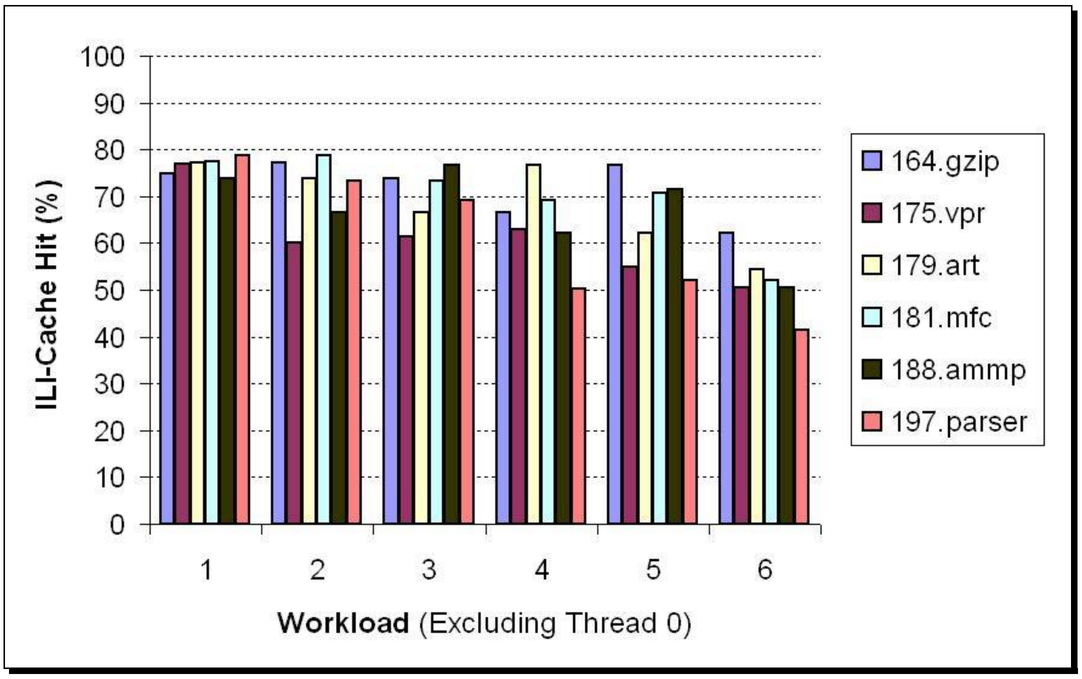


Figure 8.22: Hit ratios of L1-Icache with various numbers of workloads.

Figure 8.23 illustrates the hit ratio in the L1-Dcache[16]. The figure presents a decreasing trend in the hit ratio when the number of threads is on the increase. Even though a cache line has already associated with a thread identifier, each replacement of the cache block is based on priority and since each thread has a different priority value, the cache conflicts of multiple threads still occur.

8.4 Summary

A C++ simulation of MulTEP was produced. This simulation demonstrates a range of architectural features, including the ability to dynamically schedule and synchronise threads. A number of benchmark simulation models were used to test that the C++ simulation framework produced cycle accurate results in comparison with equivalent Verilog models.

The results of the Livermore Loop 7 and SPEC CPU2000 benchmark evaluations reflect that MulTEP is efficient when there are at least as many threads as processing

[14]L1 caches cannot utilise the scheduling in the MSU because the length of all instructions to be loaded and the space allocation of data on the memory are unpredictable.

[15]Instruction cache level 1 is a 16 kB cache with 16 sets using a direct-mapping policy (see Chapter 6 for more details)

[16]L1-Dcache is 16 kB with 128 sets using a 4-way set-associative policy (see Chapter 6).

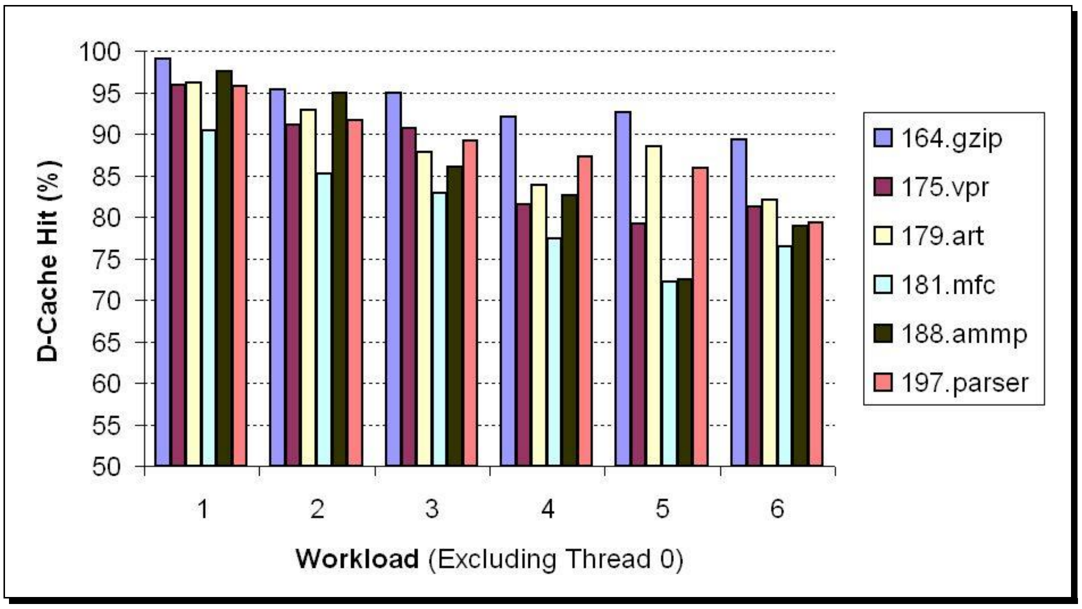


Figure 8.23: Hit ratios of DCache with various numbers of workloads.

elements. These results demonstrate that sufficient thread level parallelism provides high execution performance, especially when multithreaded source code is provided.

The efficiency of the multithreading mechanisms is relatively high. Pre-fetching instructions in L0-Icache offers approximately 99.57% hit ratio. Contexts are switched without any overhead for more than 90% of the total context switches when there are at least as many threads as execution contexts. This illustrates that pre-loaded contexts are well utilised. These features guarantee that real-time response can be attained when threads are appropriately prioritised.

The processing performance is degraded when the number of threads increases. This is due to conflicts on shared resources, primarily the first level of data and instruction caches. Nevertheless, even with this cache side effect, the results demonstrate that MulTEP is capable of providing 12% average speedup.

Chapter 9

Conclusions and Future Directions

The day of work is done.
Hide my face in your arms, mother.
Let me dream.

Rabindranath Tagore

9.1 Introduction

This thesis tackles the demands of high performance embedded processors with a novel architecture called the *MultiThreaded Embedded Processor* (MulTEP). The MulTEP system aims to deliver high performance processing with real-time response by exploiting thread level parallelism whilst avoiding non-deterministic speculative execution.

This chapter summarises key contributions derived from my study, starting with an overview of the MulTEP system (Section 9.2). Key results from implementing the MulTEP model are then discussed (Section 9.3). A comparison with related architectures is then presented (Section 9.4). Finally, the future directions are suggested (Section 9.5).

9.2 Overview of MulTEP

MulTEP combines RISC processing engines extended to support a small number of threads with an activation frame cache to store spilled thread contexts. The memory hierarchy is tuned to handle multiple threads. A scheduling mechanism is provided to issue threads in priority order. Synchronisation is supported by a matching store.

Programs are represented by multiple data-driven nanothreads. Each nanothread is a non pre-emptable control-flow section of code containing 1 to 32 instructions. In the processing unit, up to 8 nanothreads are pre-fetched and up to 4 execution contexts are pre-loaded. Each thread stream is tagged with its register identifier (colour tagging). These pre-loading, pre-fetching and colour-tagging techniques are provided to switch contexts without any overhead. Simulation results indicate that contexts are switched with zero-cycle overhead for more than 90%, if all execution contexts can be preloaded.

Thread level parallelism yields good performance gains by increasing processing element utilisation. Its underlying model allows concurrency to be expressed without the need for speculative execution and branch prediction. The elimination of these mechanisms simplifies the architecture.

The synchronisation and scheduling mechanisms used by MulTEP build on the Anaconda processor [4]. In the Anaconda architecture, data-flow style concurrency primitives supplement control-flow sequential execution primitives. Control-flow like execution allows efficient use of intermediate results. Data-flow like synchronisation allows a thread to wait for multiple data items.

The matching-store synchroniser allows a thread to wait for up to 24 data items. Each item is sent via a store instruction to a suitable activation frame. The hardware scheduler is capable of issuing the highest priority thread in a single clock cycle. This mechanism is provided to support real-time execution. To avoid starving low-priority threads, priorities are dynamically increased each time a thread is inserted into the scheduler.

The execution pipelines are similar to simple RISC DLX pipelines but with slightly more functional units to support multiple execution contexts. The majority of extra hardware is contained in the additional multithreading service unit. This unit provides hardware support to progress a thread through its life cycle.

Furthermore, the multithreading service unit handles excess execution contexts which are spilled to the memory hierarchy in the form of activation frames. The underlying mechanisms in this unit allow MulTEP to support up to 2^{16} threads without any incremental cost per thread.

Four additional multithreading instructions (i.e. `spawn`, `switch`, `wait` and `stop`) are provided for programmers and advance compilers to allow a thread to progress through its life cycle. The processing unit decodes these instructions and sends them to be serviced by the multithreading service unit. The multithreading service unit provides most of the multithreaded support in hardware though it does rely on the thread-0 system daemon to perform house keeping.

To reduce memory traffic, the load-store unit operates independently. The underlying mechanism of the load-store unit sends load and store tokens to either the activation-frame cache or the data cache. If the thread is switched out of the processing unit, its load-return data will be redirected to its activation frame by the load-store unit.

As each virtual-to-physical address translation includes a thread group ID, MulTEP allows multiple threads to refer to the overlapping virtual memory space. The same virtual addresses of different threads are translated to different physical locations. The system tries to resolve cache conflicts by associating a thread priority with each cache line in the first level cache. Nevertheless, my analysis shows that this enhanced replacement strategy is not sufficient to resolve all conflicts caused by multiple threads.

To save power, MulTEP analyses the status of the processing unit along with the scheduling information and then offers four power operating modes. The modes are: running, standby, suspend and hibernate. These operating modes could be used as part of the power management strategy for an implementation.

The performance of some standard benchmarks indicates that MulTEP benefits from multithreading when there are enough threads (i.e. there are at least as many threads as processing elements). Furthermore, the system demonstrates that the priority-based hardware scheduler may support real-time execution.

9.3 Key Contributions

The design work undertaken for MulTEP resulted in the following novel architectural features:

- **A mechanism to support zero-cycle context switches**

  To switch the execution context without any overhead, it is necessary to pre-fetch instructions, pre-load execution contexts and tag execution streams with a thread identifier. To maximise zero-cycle context switching, the optimum number of registers sets was found to me $2p$ to support p processing elements. The optimum number of L0 caches lines is $4p$. This allowed the processing elements to be kept busy (typically 99%). Fewer register sets or a smaller L0 cache reduced performance significantly. More register sets or a larger L0 cache yielded little benefit and increased the hardware cost.

- **Combining multithreading with parallel processing elements**

  Two mechanisms are introduced to support multithreading across parallel processing elements. Firstly, the data-forwarding unit is extended to allow forwarding of results between processing elements. Secondly, for each processing element, a queue needs to be added to the load/store unit and multithreading-service unit. These queues are serviced by each unit in thread priority order, or round robin order if the priorities are the same.

- **A framework to support flexible multithreading with fixed implementation cost**

  As summarised in Section 9.2, MulTEP offers a framework to support a flexible number of threads by using an activation-frame cache. This cache holds spilled contexts close to the processing elements. Consequently, only 2 contexts per processing element are required. At any one time, the p processing elements are executing threads with contexts stored in p register sets. The remaining p register sets hold contexts of threads waiting to execute, or the context is being transfered to or from the activation frame cache.

- **Instructions to support multithreaded operation**

MulTEP's programmer's model allows programs to control the progress of each thread using only four additional multithreading instructions: `spawn`, `switch`, `wait` and `stop`. These instructions are demonstrably sufficient to support a wide range of thread models including the Java virtual machine.

9.4 Related work

A Comparison with related architectures has been conducted by analysing both multithreading efficiency and embedded system design aspects. Based on an investigation proposed in the Anaconda project [4], one view of the multithreaded design space can be obtained by plotting the amount of available concurrency supported by the hardware against the size of the non-preemptable executable unit. Figure 9.1 presents a range of architectures from traditional control-flow (CISC and RISC) designs which exhibit little concurrent behaviour, to data-flow machines which exhibit excessive concurrent behaviour.

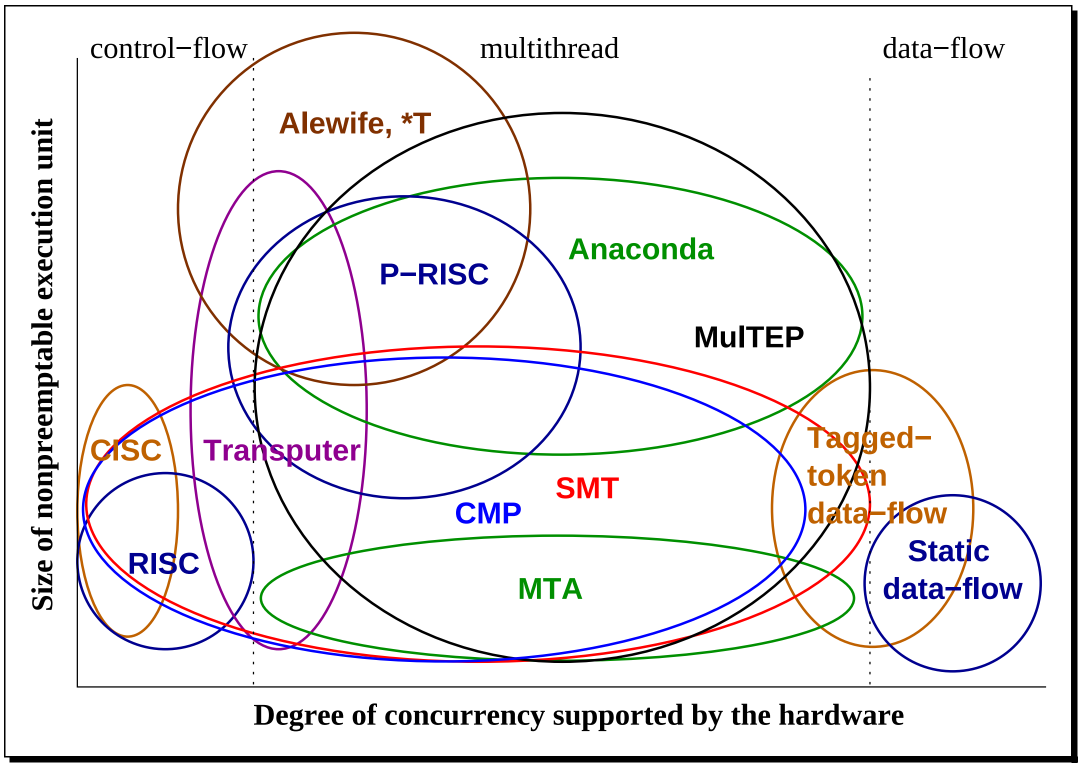


Figure 9.1: Comparison with other related work.

Further comparison has been conducted. On the one hand MulTEP was compared with its anchester, Anaconda. On the other hand, it was compared with the following architectural designs, which have been used in, or are purposed to be used for, embedded systems.

- *MultiThreading Architecture* (MTA)

Fine-grained interleaving technique from TERA's MTA is used in Sanders' MTA real-time engine [156] and XInC wireless processor [157].

- *Simultaneous MultiThreading* (SMT)

  This technique is used in MIPS's MultiThread *Application-Specific Extension* (MT-ASE) for embedded systems, Ubicom's MASI embedded processor and XStream processor core for embedded application [158].

- *Chip MultiProcessor* (CMP)

  CMP architectural technique is used in Edinburgh's asynchronous multithreaded architecture [159, 160], the *VLSI Architecture Using Lightweight Threads* (VAULT) [161], the extended CMP version of the *differential MultiThreading* (dMT-CMP), ClearSpeed's CS301 processor [162] and Intel's IXP 1200 network processor [163].

An overall comparison is presented in Table 9.1 where r represents the number of register sets on each processing unit, C represents the context switching overhead in cycles, c represents the context size, N represents the number of threads, R represents the run length per thread and n represents the size of a thread.

Features	`MTA`	`SMT`	`CMP`	`Anaconda`	`MulTEP`
Switching overhead (C)	0	0	0-c	0-8	0-16
Number of threads (N)	r	r	$p \times r$	> 1	> 1
Run length (R)	1	1-n	1-n	8-n	1-n
Thread synchronisation	software	hardware	hardware	hardware	hardware
Thread scheduling	FIFO	speculative	speculative	deadline	priority
Design complexity	low	high	medium	medium	medium
Change thread state	software	software	software	software	hardware

Table 9.1: A comparison of multithreaded embedded architectures.

As presented in Chapter 3, high processor efficiency is obtained with a low value of context switching overhead (C), a large number of threads (N) and a long run length (R). For the first parameter (C), all multithreaded architectural techniques for embedded system have been designed to be capable of switching contexts with very low overhead (e.g. $C \to 0$). MTA uses a simple context interleaving to guarantee zero-cycle context switches. SMT benefits from aggressive data/control prediction and dynamic scheduling to utilise its shared resource hence it automatically provides zero-cycle context switches. Nevertheless, this technique relies on statistical speculation and wastes power in mispredicted tasks. CMP maintains zero-cycle context switching on a condition that it has a sufficient number of register sets in each processing element, i.e. $r > 1$, and the run length is sufficiently long. Otherwise, its context switch is likely to be longer than zero cycle due to the bus bandwidth limitation. MulTEP has a hardware mechanism which typically enables over 90% of the context switches to complete in zero cycle (see Chapter 8).

For the second parameter (N), the number of threads in both MTA and SMT are limited by the number of register sets, which is restricted because a large register file will directly slow down the pipeline's decoding stage. CMP escapes from such a restriction by adding more processors ($p > 1$). Nevertheless, the number of threads is limited by the total number of registers in the system. In contrast, Anaconda and MulTEP have a comparatively small number of registers and rely on an activation frame cache mechanism to store excess thread contexts.

For the third parameter (R), MTA's run length is very small (1 cycle) so it requires a large number of threads to hide memory latency. SMT provides flexible fine-grained instruction level parallelism ($R > 1$) by relying on data and control prediction together with a dynamic scheduling. The run lengths of CMP, Anaconda and MulTEP architectural techniques are also flexible. However, as the context switching overhead in CMP is often longer than zero cycle, CMP's run length is required to be long enough to give a good processor utilisation.

In terms of the architectural details, MTA is not complex but the system relies on software for thread synchronisation. Software synchronisation is substantially slower than hardware synchronisation hence the architecture is not a suitable choice for multithreaded applications which have significant data interdependancies between threads. Thread scheduling on MTA is also rather primative: a FIFO queue provides round robin scheduling which is insufficient for real-time systems. SMT architecture is more complex because it requires an aggressive speculation to utilise its shared resources and to support dynamic thread scheduling. CMP and Anaconda are moderately complex because they also need some units to support multithreading activities (e.g. thread synchronisation and scheduling).

MulTEP introduces some design complexity in its multithreading service unit (see Chapter 6). The advantage of having such a unit is that it allows a thread to progress through its life cycle in hardware within a couple of clock cycles without relying on software routines. This feature differentiates the MulTEP architecture from the other models. Furthermore, MulTEP dynamically determines when context switches occur where as some other architectures require explicit thread controls to be inserted. The architecture does provide thread control instructions and these are capable of supporting a wide range of software thread models. MulTEP also supports an arbitrary run length and consequently it can efficiently execute single threaded code as well as multithreaded code.

9.5 Future Directions

This section presents my suggestions for possible future extensions. The suggestions emanate from experiences that I have gained during this research project.

9.5.1 Multi-ported Priority Queue

Currently, the scheduling of threads inside the MulTEP's processing unit is based on a 2-element pre-scheduled queue. The queue is implemented using a tagged up/down sorter which is limited to output one item per clock cycle. If we wish to simultaneously schedule more than one thread, this queue becomes a bottleneck. A multiported front-end to the queue would be a small advantage.

9.5.2 Further Enhancements to the L0-cache

The current L0-cache is a traditional fully associative cache. A pre-fetched nanothread is 1 to 32 instructions and is placed in a 32 word cache line. Therefore, even though the run length of a nanothread is very fine grained (e.g. less than 32 instructions), the current I0-cache ends up fetching unrelated instructions. In the degenerate case, the required instructions straddle two cache lines resulting in a considerable number of extraneous instructions being fetched. This wastes both time and power.

One possible solution is to allow cache-lines to be filled starting at arbitrary addresses rather than being on fixed cache-line boundaries.

9.5.3 Thread-0 Daemon Performance Improvements

The Thread-0 daemon performs house keeping functions. To simplify the design during development, link lists were used to store the active-thread and wait-for-join tables. This design decision had little impact on the benchmark results. Nonetheless, performance gains could me made by using more sophisticated data structures (e.g. a hash table for the active-thread table and a heap for the wait-for-join table).

9.5.4 Processor Scalability

Even though MulTEP is designed with two processing elements and four execution contexts, further modification can be conducted. As presented in Equation 5.2, the utilisation of the processing elements relies on the characteristic of a thread's run length, its context size and its input bandwidth. In order to scale the number of processing elements, these parameters need to be adjusted. The first parameter, a thread's run length (μ^{-1}), typically depends on the application which is imposed on the hardware. As a result, the later two parameters, the context size (x) and input bandwidth (B), become the key architecture scalability factors.

With regards to the queueing model as presented in Section 5.4, the number of processing elements (s) can be increased while still providing a good processing element utilisation ($U \rightarrow 1$) in accordance to the following equation:

$$s \;\propto\; \sum_{n=0}^{C-1}\left[(C-1+\frac{B}{x})\Big(\frac{\mu^{-1}}{\mu^{-1}+\frac{x}{B}}\Big)\right]^{n} \tag{9.1}$$

The number of processing elements s can be increased when the context size x decreases or the bus bandwidth B increases if the run length is sufficiently long (i.e. more than $\frac{x}{B}$). However, such an increment is bounded with the minimum context size and the maximum bandwidth of the bus.

For a large number of processing elements, alternative structures, such as a cluster of the processing units that consists of a small number of processing elements and execution contexts together with a local multithreading service unit that contains only a small activation-frame cache, might be a more appropriate design. These clusters can share a global activation-frame cache in the similar fashion to the memory hierarchy. The detail of their mechanisms, such as an interconnection network and a wide-issue hardware scheduler, are interesting areas for further investigation.

9.5.5 Cache Conflict Resolution

As presented in the side effect experiments (8.3.5), the association of thread priorities is not sufficient to resolve the conflicts caused by multiple threads competing for unsupervised shared resources (i.e. L1 data and instruction caches). As for the L0 I-cache, scheduling information could be used to assist the cache management of the L1 data and instruction caches.

9.5.6 Application Specific Processor Refinement

MulTEP was designed to be a general purpose processor, but it could be tuned for a particular application adding application specific features and removing unused features. For example, the activation frame cache size or the number of register sets could be adjusted or the floating point instructions could be removed.

9.5.7 Software Support

There is a great deal of scope to design better compilers to support multithreaded processors. There is probably many PhDs worth of work in extracting thread level concurrency from sequential programs.

Intel's OpenMP high-level hyper-threading compiler for C++ and Fortran 95 [164] were investigated. The OpenMP compiler incorporates a number of well-known optimisation techniques for a high degree of parallelism, such as automatic loop parallelisation and OpenMP directive-guided and pragma-guided parallelisation.

OpenMP is a portable programmability model for shared-memory parallelism [165, 166] which exploits medium- and coarse-grained parallelism for the architecture. There are four phases of compilation in total. The first phase reconstructs generic procedures of the code and provides inter-procedural optimisation. The second phase incorporates OpenMP, automatic parallelisation and vectorisation. This phase is crucial for providing the multithreaded result. The third phase applies high-level and scalar optimisation. The forth phase offers lower-level code generation where MulTEP's multithreading functions could be used (e.g. via the provided macros in Chapter 7).

Finally, the thread-0 system daemon would need to be included into a standard operating systems. Thus, there is much room left for future research into software support.

Appendix A

Tagged Up/Down Sorter

A.1 The verilog HDL source code

```
/*****************************************************\
* Program: tgsort.v                                  *
*                                                    *
* A verilog HDL file to simulate Tag sorting queue   *
*                                                    *
* by:         Panit Watcharawitch (pw240)            *
* version:    2.0                                    *
* Last update: 27/03/01                              *
\*****************************************************/

// General Definition
`define MAX_KEY     8
`define KEY_MIN     8'h00
`define KEY_MAX     8'hFF

`define MAX_DATA    16
`define DATA_MIN    16'h0000
`define DATA_MAX    16'hFFFF

module tgsort();
   reg                    clk;

   // Input
   reg                    insert;
   reg                    extract;
   reg [`MAX_KEY-1:0]     ikey;
   reg [`MAX_DATA-1:0]    idata;

   // Output
   reg [`MAX_KEY-1:0]     key;
   reg [`MAX_DATA-1:0]    data;
   wire                   empty;
   wire                   full;

   // Initial
   initial begin
      $monitor("time=%0d extract=%0d okey=%0d odata=%0d",$time, extract, okey, odata);
      clk = 0;

//       $clkdef(clk,20,40,40);  // only for cadence 1999 version

      // insertion
      insert  = 1;
      extract = 0;
      key     = 0;

          ikey = 3; idata=0;
      #60 ikey = 2; idata=1;
      #40 ikey = 1; idata=2;
      #40 ikey = 5; idata=3;
      #40 ikey = 6; idata=4;
      #40 ikey = 7; idata=5;
      #40 ikey = 3; idata=6;

      #40 // stop insert
      insert  = 0;
      extract = 1;

      #500 $stop;
      $finish;
   end // initial begin

   always
     #20 clk = ~clk;  // Delay (20ns) is set to half the clock cycle
                      // only for cadence (verilogy-XL) 2001 version

   // Core of Tgsort
   wire [`MAX_KEY+`MAX_DATA:0] L0, L1, L2, L3, L4, L5, L6, L7, L8, L9;
   wire [`MAX_KEY+`MAX_DATA:0] R0, R1, R2, R3, R4, R5, R6, R7, R8;
   wire [`MAX_KEY-1:0]         okey;
   wire [`MAX_DATA-1:0]        odata;
   wire                        otag;

   always @(posedge clk&extract) begin
      key  <= okey;
      data <= odata;
   end

   // Connection here;
   LR lr0(clk, insert, extract, {ikey,idata,1'b0}, L0, R0, {okey,odata,otag});
   LR lr1(clk, insert, extract, L0, L1, R1, R0);
   LR lr2(clk, insert, extract, L1, L2, R2, R1);
   LR lr3(clk, insert, extract, L2, L3, R3, R2);
   LR lr4(clk, insert, extract, L3, L4, R4, R3);
   LR lr5(clk, insert, extract, L4, L5, R5, R4);
   LR lr6(clk, insert, extract, L5, L6, R6, R5);
   LR lr7(clk, insert, extract, L6, L7, R7, R6);
   LR lr8(clk, insert, extract, L7, L8, R8, R7);
   LR lr9(clk, insert, extract, L8, L9, {`KEY_MIN,`MAX_DATA'b0,1'b0}, R8);

   assign empty = (key==`KEY_MIN)&(okey==`KEY_MIN);
   assign full  = ((L9>>(`MAX_DATA+1))!=`KEY_MIN);
endmodule


module LR(clk, insert, extract, inL, outL, inR, outR);
   input  clk;
   input  insert;
   input  extract;
   input  [`MAX_KEY+`MAX_DATA:0]    inL;
   input  [`MAX_KEY+`MAX_DATA:0]    inR;
   output [`MAX_KEY+`MAX_DATA:0]    outL;
   output [`MAX_KEY+`MAX_DATA:0]    outR;

   reg [`MAX_KEY-1:0]  Lkey,  Rkey;
   reg [`MAX_DATA-1:0] Ldata, Rdata;
   reg                 Ltag,  Rtag;

   reg oldx;

   initial begin
      Lkey  = `KEY_MIN;
      Ldata = 0;
      Ltag  = 0;
      Rkey  = `KEY_MIN;
      Rdata = 0;
      Rtag  = 0;
      oldx  = 0;

      #10
      Lkey  = `KEY_MIN;
      Rkey  = `KEY_MIN;
   end

   wire x   = ((oldx&(Lkey==Rkey))|(Lkey<Rkey)|(~oldx&Rtag))&~(oldx&Ltag);
   wire ac  = (x&insert)|(~x&extract);
   wire bc  = (~x&insert)|(x&extract);
   wire atc = insert|(~x&extract);
   wire btc = insert|(x&extract);

   wire [`MAX_KEY+`MAX_DATA:0] Ain = (x) ? inL : inR;
   wire [`MAX_KEY+`MAX_DATA:0] Bin = (x) ? inR : inL;

   always @(posedge clk) begin
      if (ac) begin
         Lkey  <= Ain>>(`MAX_DATA+1);
         Ldata <= Ain>>1;
      end

      if (atc) begin
         Ltag  <= (Ain&1)|~x;
      end

      if (bc) begin
         Rkey  <= Bin>>(`MAX_DATA+1);
         Rdata <= Bin>>1;
      end

      if (btc) begin
         Rtag <= (Bin&1)|x;
      end

      if (insert|extract) begin
         oldx <= x;
      end
   end // always @ (posedge clk)

   assign outL = (x) ? {Lkey, Ldata, Ltag} : {Rkey, Rdata, Rtag};
   assign outR = (x) ? {Rkey, Rdata, Rtag} : {Lkey, Ldata, Ltag};
endmodule
```

A.2 The simulation source code

A.2.1 tgsort.hh

```
/*------------------------------------------------------------------*\
|  Program: tgsort.hh                                                |
|  Details: Tagged Up/Down Sorter                                    |
|  Version: 2.0 (Mar 16th, 2001)                                     |
|  By:      Panit Watcharawitch                                      |
\*------------------------------------------------------------------*/
#include "conio.h"

#define Q_SIZE 100
#define S_SIZE 10     // max sort = 20 element
#define K_TYPE unsigned int
#define D_TYPE int

#define ZERO_PRI false
#define DYNAMIC  false

#define TRACE  1
#define T_SIZE 3

int trace;

struct Package {
  K_TYPE key;
  D_TYPE data;
  BIT    tag;
};

/*************
 * LR Object *
 *************/
struct LR_In {
  Package L, R;
  BIT     insert, extract;
};

struct LR_Out{
  Package L, R;
};

template<class InPut, class OutPut>
class LR : public SimObj<InPut, OutPut>{
public:
  LR();

  void function();

private:
  Package L, R;

  void cmp_swap();
};

// House keeping
template<class InPut, class OutPut>
LR<InPut, OutPut>::LR() {
  in.L.key  = (K_TYPE)0;
  in.L.data = 0;
  in.R.key  = (K_TYPE)0;
  in.R.data = 0;
  in.R.tag  = false;

  L.key  = (K_TYPE)0;
  L.data = 0;
  R.key  = (K_TYPE)0;
  R.data = 0;
  R.tag  = false;
}

// Interface function
template<class InPut, class OutPut>
void LR<InPut, OutPut>::function() {
  if (in.insert)  L = in.L;
  if (in.extract) R = in.R;

  cmp_swap();
  if (trace) printf(" [(%d,%d) (%d,%d)]\n", L.key, L.data, R.key, R.data);
  out.L = L;
  out.R = R;
}

template<class InPut, class OutPut>
void LR<InPut, OutPut>::cmp_swap() {
  Package Temp;

  if ((L.key>R.key)||L.tag) {                // comparison
    Temp   = L;                              // swap
    L      = R;
    R      = Temp;
    R.tag = true;
  }
}

/**********************
 * Tag Sorting Object *
 **********************/
struct TgSort_In {
  BIT     insert;
  BIT     extract;
  K_TYPE key;
  D_TYPE data;
};

struct TgSort_Out {
  K_TYPE key;
  D_TYPE data;
  BIT     empty;
  BIT     full;
};

template<class InPut, class OutPut>
class TgSort : public SimObj<InPut, OutPut>{
public:
  void function();         // Interface functions

private:
  LR<LR_In, LR_Out> lr[S_SIZE];

  void connection();     // Inside functions
};

// Interface functions
template<class InPut, class OutPut>
void TgSort<InPut, OutPut>::function() {
  short i;

  if (TRACE) {
    _textattr(GREEN); printf("--- clk = %d ---\n", clk+1); _textattr(NORMAL);
    if (in.insert)  {
      _textattr(BROWN); printf("In:(%d,%d) ", in.key, in.data);
      _textattr(NORMAL);
    } else printf("In:(-,-) ");
    if (in.extract) {
      _textattr(YELLOW);
      printf("Out:(%d,%d)\n", lr[0].out.R.key, lr[0].out.R.data);
      _textattr(NORMAL);
    } else printf("Out:(-,-)\n");
  }

  // pre-clk
  if (ZERO_PRI) lr[0].in.L.key = in.key + 1;
  else          lr[0].in.L.key    = in.key;
  lr[0].in.L.data         = in.data;
  lr[0].in.L.tag          = false;
  lr[S_SIZE-1].in.R.tag = false;
  if (ZERO_PRI) out.key = lr[0].out.R.key - 1;
  else out.key           = lr[0].out.R.key;
  out.data               = lr[0].out.R.data;

  // main function
  for (i=0; i<S_SIZE; i++) {
    trace = TRACE&&(i<T_SIZE);

    lr[i].in.insert   = in.insert;
    lr[i].in.extract = in.extract;
    lr[i].simclk();
  }

  connection();                          // wire-connection
}

// Inside functions
template<class InPut, class OutPut>
void TgSort<InPut, OutPut>::connection() {
  short i;

  for (i=0; i<S_SIZE; i++)
    if (DYNAMIC) {
      lr[i+1].in.L = lr[i].out.L    + 1;
      lr[i].in.R    = lr[i+1].out.R + 1;
    } else  {
      lr[i+1].in.L = lr[i].out.L;
      lr[i].in.R   = lr[i+1].out.R;
    }

  out.empty = (lr[0].out.R.key==(K_TYPE)0);
  out.full  = (lr[S_SIZE-1].out.L.key!=(K_TYPE)0);
}
```

A.2.2 tgsort.cc

```
/*---------------------------------------------------------------------*\
|  Program:  tgsort.cc                                                  |
|  Details:  Tagged Up/Down Sorter                                      |
|  Version:  2.0 (Mar 16th, 2001)                                       |
|  By:       Panit Watcharawitch                                        |
\*---------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include "simobj.hh"
#include "tgsort.hh"

// Global Variable
K_TYPE queue[Q_SIZE];
D_TYPE data[Q_SIZE];
int    clk;

/****************
 * Main Program *
 ****************/

void show_help() {
  printf("Format : tgsort number(s)...\n");
  printf("Example: tgsort 2 1 5 6\n");

  exit(EXIT_FAILURE);
}

void initailise(int argc, char *argv[]) {
  int i;

  clk=0;

  _textattr(CYAN);
  printf("Tagged Up/Down Sorter                                   Version 2.0\n");
  printf("-------------------------------------------------------------------\n");
  printf("Input  : ");
  _textattr(BROWN);

  for (i=0; i<argc-1; i++) {
    queue[i] = atoi((char *)(argv[i+1]));
    printf("(%d,%d) ", queue[i], i);
  }
  _textattr(NORMAL);

  printf("\n");
}

void finalise(int size) {
  short i;

  _textattr(CYAN); printf("Output : "); _textattr(YELLOW);
  for (i=0; i<size; i++) printf("(%d,%d) ", queue[i], data[i]);
  _textattr(CYAN);
  printf("\n\n");
  printf("Total key   = %d\n", size);
  printf("Total clock = %d\n", clk);
  printf("-------------------------------------------------------------------\n");
  _textattr(NORMAL);
}

int main(int argc, char *argv[]) { TgSort<TgSort_In, TgSort_Out> tgsort;
  short  i;
  int    size = argc-1;

  if (argc<2) show_help();

  initailise(argc, argv);

  // Insertion
  tgsort.in.insert  = true;
  tgsort.in.extract = false;
  for (i=0; i<size; i++) {
    tgsort.in.key  = queue[i];
    tgsort.in.data = clk;
    tgsort.simclk();
    clk++;
  }

  // Extraction
  tgsort.in.insert  = false;
  tgsort.in.extract = true;
  for (i=0; i<size; i++) {
    tgsort.simclk();
    queue[i] = tgsort.out.key;
    data[i]  = tgsort.out.data;
    clk++;
  }

  finalise(size);

  return EXIT_SUCCESS;
}
```

Appendix B

Livermore Loop 7

B.1 The single-thread java code

```
class ll7_s {
    public static void main(String[] args) {
        int N  = 1000;
        int LL = 7;

        int[] u = new int[N+LL];
        int[] x = new int[N];
        int[] z = new int[N];
        int[] y = new int[N];

        int t = 1;
        int r = 2;
        int l, k;

        // initialise
        for (l=0; l<N+LL; l++) u[l] = LL-l;
        for (l=0; l<N; l++) {
            z[l] = l;
            y[l] = 2*l;
        }

        // livermore loop 7
        for ( k=0 ; k<N ; k++ ) {
            x[k] = u[k] + r*( z[k] + r*y[k] ) +
                   t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
                      t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
        }

        System.out.println("Livermore loop 7:");

        for ( k=0 ; k<N ; k++ )
            System.out.println("x[" + k + "] = " + x[k]);

    }
}
```

B.2 The multithreaded java code

```
class ll7_m {
    static int N  = 15;
    static int LL = 7;
    static int th = 5;
    static int[] x = new int[N];
    static int[] u = new int[N+LL];
    static int[] z = new int[N];
    static int[] y = new int[N];
    static int t = 1;
    static int r = 2;

    public static void main(String[] args) {
        int l;

        // initialise
        for (l=0; l<N+LL; l++) u[l] = LL-l;
        for (l=0; l<N; l++) {
            z[l] = l;
            y[l] = 2*l;
        }

        // livermore loop 7
        l17 t1 = new l17(0,3);
        l17 t2 = new l17(3,6);
        l17 t3 = new l17(6,9);
        l17 t4 = new l17(9,12);
        l17 t5 = new l17(12,N);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();

        try {
            t1.join();
            t2.join();
            t3.join();
            t4.join();
            t5.join();
        } catch (InterruptedException e) {}

        System.out.println("Livermore loop 7:");

        for ( l=0 ; l<N ; l++ )
            System.out.println("x[" + l + "] = " + x[l]);
    }
}

class l17 extends Thread {
    int start;
    int stop;

    // constructor
    public l17(int start, int stop) {
        this.start = start;
        this.stop  = stop;
    }

    public void run() {
        int k;

        for (k=start; k<stop; k++) {
            l17_m.x[k] = l17_m.u[k] +
                l17_m.r*(l17_m.z[k]+l17_m.r*l17_m.y[k]) +
                l17_m.t*(l17_m.u[k+3] +
                         l17_m.r*(l17_m.u[k+2] +
                                  l17_m.r*l17_m.u[k+1] ) +
                         l17_m.t*( l17_m.u[k+6] +
                                   l17_m.r*( l17_m.u[k+5] +
                                             l17_m.r*l17_m.u[k+4])));
        }
    }
}
```

B.3 The 14-thread assembly code

```
        ;;------------------------------------------------------------
        ;; l17_m.s (version 4.1 -- 19/2/2002 by Panit Watcharawitch)
        ;;------------------------------------------------------------
.text

main:   lui     $v1,0x00FF
        addiu   $v0,$v1,0xFF7F   ; wait for $t1
        start   $s6,prepX        ; *** prepX ***        (10c)

        addiu   $v0,$v1,0xFF0F   ; wait for $a2-t1
        start   $t1,af1          ; *** af1 ***          (1c0)
        sw      $t1,$t1($s6)     ; af1 -> prepX

        addiu   $v0,$v1,0xFEFF   ; wait for $t2
        start   $s7,prepY        ; *** prepY ***        (14c)

        addiu   $v0,$v1,0xFF83   ; wait for $t0, $a0-3
        start   $t2,af2          ; *** af2 ***          (23c)
        sw      $t2,$t2($s7)     ; af2 -> prepY

        addiu   $v0,$v1,0x307F   ; wait for $t1-7, s0-1
        start   $t0,af0          ; *** af0 ***          (17c)

        addiu   $v0,$v1,0x001F   ; wait for $a3, $t0-s1
        start   $t4,af4          ; *** af4 ***          (380)

        addiu   $v0,$v1,0x0017   ; wait for $a1, $a3, $t0-s1
        start   $t5,af5          ; *** af5 ***          (43c)
        lui     $v1,0x00C0
        addiu   $v0,$v1,0x007F   ; wait for $t1-7, $s2-7
        start   $t3,af3          ; *** af3 ***          (2dc)

        addiu   $v0,$v1,0xC08F   ; wait for $t2-7, $s2-7, $a2-3, $t0
        start   $t6,af6          ; *** af6 ***          (528)

        addiu   $v0,$v1,0x00F3   ; wait for $t2-7, $s0-7, $a0-1
        start   $t7,af7          ; *** af7 ***          (570)

        addiu   $v0,$v1,0x00BB   ; wait for $t0-7, $s0-7, $a0, $v1
        start   $t8,af8          ; *** af8 ***          (5e8)

        addiu   $a0,$sp,232      ; x ($a0)
        sw      $a0,$s0($t0)     ; x   -> af0
        sw      $zero,$s1($t0)   ; 0   -> af0 (init)
        sw      $t1,$t1($t0)     ; af1 -> af0
        sw      $t2,$t2($t0)     ; af2 -> af0
        sw      $t3,$t3($t0)     ; af3 -> af0
        sw      $t4,$t4($t0)     ; af4 -> af0
        sw      $t5,$t5($t0)     ; af5 -> af0

        sw      $t3,$t1($t1)     ; af3 -> af1

        sw      $t4,$a1($t2)     ; af4 -> af2
        sw      $t5,$a2($t2)     ; af5 -> af2
        sw      $t6,$a3($t2)     ; af6 -> af2

        li      $s1,2            ; r ($s1)
        sw      $s1,$s1($t3)     ; r   -> af3
        sw      $t1,$t1($t3)     ; af1 -> af3
        sw      $t6,$s0($t3)     ; af6 -> af3

        sw      $t7,$s0($t4)     ; af7 -> af4
        sw      $t2,$a3($t4)     ; af2 -> af4
        sw      $s1,$s1($t4)     ; r   -> af4

        li      $a1,1            ; t ($a1)
        sw      $a1,$a1($t5)     ; t -> af5
        sw      $t7,$s0($t5)     ; af7 -> af5
        sw      $t2,$a3($t5)     ; af2 -> af5
        sw      $s1,$s1($t5)     ; r   -> af5

        sw      $t2,$a2($t6)     ; af2 -> af6
        sw      $t3,$a3($t6)     ; af3 -> af6
        sw      $t8,$t0($t6)     ; af8 -> af6

        sw      $a1,$a1($t7)     ; t   -> af7
        sw      $t4,$s0($t7)     ; af4 -> af7
        sw      $t5,$s1($t7)     ; af5 -> af7
        sw      $t8,$a0($t7)     ; af8 -> af7

        li      $t9,st2
        sw      $t9,$a0($t8)     ; st2 -> af8
        sw      $zero,$a1($t8)   ; 0   -> af8
        sw      $a0,$t0($t8)     ; x   -> af8
        sw      $t6,$s0($t8)     ; af6 -> af8
        sw      $t7,$s1($t8)     ; af7 -> af8

        li      $a0,st1
        jal     0                ; printf st1
        stop    $at
        end                      ; end main

        ;;--------------------
        ;; af1 -> $t1

prepX:  addiu   $a2,$sp,292      ; y
        sw      $a2,$a2($t1)     ; signal (y -> af1)
        addiu   $a3,$sp,352      ; z
        move    $t3,$a3
        move    $v1,$zero
        li      $a0,15

initzy: sll     $v0,$v1,0x1      ; $v0 = $v1 x 2
        sw      $v1,0($a3)
        sw      $v0,0($a2)
        addiu   $v1,$v1,1
        addiu   $a3,$a3,4
        addiu   $a2,$a2,4
        bne     $v1,a0,initzy

        sw      $t3,$a3($t1)     ; signal (z -> af1)
        stop    $at
        end

        ;;--------------------
        ;; af2 -> $t2

prepY:  move    $t3,$sp          ; u
        move    $a0,$t3
        li      $v1,21
        li      $t1,7

initu:  addi    $v1,$v1,-1
```

```
        sw      $t1,0($t3)
        addiu   $t3,$t3,4
        addi    $t1,$t1,-1
        bgez    $v1,initu

        sw      $a0,$a0($t2)     ; signal (u -> af2)
        stop    $at
        end

        ;; -------------------
        ;; x          -> $s0
        ;; init(0) -> $s1
        ;; af1-5    -> $t1-5

af0:    slti    $v0,$s1,15
        move    $t8,$s1
        addiu   $s1,$s1,6        ; i += 6            // here for fast cal
        blez    $v0,finish

        sw      $at,$t0($t1)     ; signal to $t0 of af1 (activate)
        sw      $at,$t0($t2)     ; signal to $t0 of af2 (activate)
        lui     $v0,0x01FF
        addiu   $v0,$v0,0xFE7F   ; wait for $t1-2
        sw      $v0,0($at)       ; reset af0 itself (wait for $v0, $v1)
        j       endaf0

finish: stop    $t1
        stop    $t2
        stop    $t3
        stop    $t4
        stop    $t5
        stop    $at

endaf0: end

        ;; -------------------
        ;; y    -> $a2
        ;; z    -> $a3
        ;; af0 -> $t0
        ;; af3 -> $t1

af1:    lw      $t2,0($a2)       ; $t2 = y[i]
        lw      $t3,4($a2)       ; $t3 = y[i+1]
        lw      $t4,8($a2)       ; $t4 = y[i+2]
        lw      $t5,12($a2)      ; $t5 = y[i+3]
        lw      $t6,16($a2)      ; $t6 = y[i+4]
        lw      $t7,20($a2)      ; $t7 = y[i+5]

        lw      $s2,0($a3)       ; $s2 = z[i]
        lw      $s3,4($a3)       ; $s3 = z[i+1]
        lw      $s4,8($a3)       ; $s4 = z[i+2]
        lw      $s5,12($a3)      ; $s5 = z[i+3]
        lw      $s6,16($a3)      ; $s6 = z[i+4]
        lw      $s7,20($a3)      ; $s7 = z[i+5]

        sw      $t2,$t2($t1)     ; store to $t2 @ af3
        sw      $t3,$t3($t1)     ; store to $t3 @ af3
        sw      $t4,$t4($t1)     ; store to $t4 @ af3
        sw      $t5,$t5($t1)     ; store to $t5 @ af3
        sw      $t6,$t6($t1)     ; store to $t6 @ af3
        sw      $t7,$t7($t1)     ; store to $t7 @ af3
        sw      $s2,$s2($t1)     ; store to $s2 @ af3
        sw      $s3,$s3($t1)     ; store to $s3 @ af3
        sw      $s4,$s4($t1)     ; store to $s4 @ af3
        sw      $s5,$s5($t1)     ; store to $s5 @ af3
        sw      $s6,$s6($t1)     ; store to $s6 @ af3
        sw      $s7,$s7($t1)     ; store to $s7 @ af3

        addiu   $a2,$a2,24       ; y[i] <- y[i+6]
        addiu   $a3,$a3,24       ; z[i] <- z[i+6]

        lui     $v0,0x00FF
        addiu   $v0,$v0,0xFF3F   ; wait for $t0, $t1
        sw      $v0,0($at)       ; reset af1 itself
        sw      $at,$t1($t0)     ; signal af0
        end                      ; end af1

        ;; -------------------
        ;; u    -> $a0
        ;; af0 -> $t0
        ;; af4 -> $a1
        ;; af5 -> $a2
        ;; af6 -> $a3

af2:    lw      $t4,16($a0)      ; $t4 = u[i+4]
        lw      $t5,20($a0)      ; $t5 = u[i+5]
        lw      $t6,24($a0)      ; $t6 = u[i+6]
        lw      $t7,28($a0)      ; $t7 = u[i+7]
        lw      $s0,32($a0)      ; $a0 = u[i+8]
        lw      $s1,36($a0)      ; $a1 = u[i+9]
        lw      $s2,40($a0)      ; $a2 = u[i+10]
        lw      $s3,44($a0)      ; $a3 = u[i+11]
        lw      $t1,4($a0)       ; $t1 = u[i+1]
        lw      $t2,8($a0)       ; $t2 = u[i+2]
        lw      $t3,12($a0)      ; $t3 = u[i+3]
```

```
         lw       $t9,0($a0)        ; $t9 = u[i]

         sw       $t4,$t0($a2)      ; store to $t0 @ af5
         sw       $t5,$t1($a2)      ; store to $t1 @ af5
         sw       $t6,$t2($a2)      ; store to $t2 @ af5
         sw       $t7,$t3($a2)      ; store to $t3 @ af5
         sw       $s0,$t4($a2)      ; store to $t4 @ af5
         sw       $s1,$t5($a2)      ; store to $t5 @ af5
         sw       $s2,$t6($a2)      ; store to $t6 @ af5
         sw       $s3,$t7($a2)      ; store to $t7 @ af5

         sw       $t1,$t0($a1)      ; store to $t0 @ af4
         sw       $t2,$t1($a1)      ; store to $t1 @ af4
         sw       $t3,$t2($a1)      ; store to $t2 @ af4
         sw       $t4,$t3($a1)      ; store to $t3 @ af4
         sw       $t5,$t4($a1)      ; store to $t4 @ af4
         sw       $t6,$t5($a1)      ; store to $t5 @ af4
         sw       $t7,$t6($a1)      ; store to $t6 @ af4
         sw       $s0,$t7($a1)      ; store to $t7 @ af4

         sw       $t9,$t2($a3)      ; store to $t2 @ af6
         sw       $t1,$t3($a3)      ; store to $t3 @ af6
         sw       $t2,$t4($a3)      ; store to $t4 @ af6
         sw       $t3,$t5($a3)      ; store to $t5 @ af6
         sw       $t4,$t6($a3)      ; store to $t6 @ af6
         sw       $t5,$t7($a3)      ; store to $t7 @ af6

         addiu    $a0,$a0,24        ; u[i] <- u[i+6]

         lui      $v0,0x00FF
         addiu    $v0,$v0,0xFF87    ; wait for $t0, $a1-3
         sw       $v0,0($at)        ; reset af2 itself
         sw       $at,$t2($t0)      ; signal af0
         end                        ; end af2

         ;;------------------
         ;; af6 -> $s0
         ;; r   -> $s1
         ;; af1 -> $t1

af3:     mult     $s1,$t2           ;
         mflo     $t2               ;
         addu     $s2,$s2,$t2       ;
         mult     $s1,$s2           ;
         mflo     $s2               ; $s2 = r*(z[i]+r*y[i])

         mult     $s1,$t3           ;
         mflo     $t3               ;
         addu     $s3,$s3,$t3       ;
         mult     $s1,$s3           ;
         mflo     $s3               ; $s3 = r*(z[i+1]+r*y[i+1])

         mult     $s1,$t4           ;
         mflo     $t4               ;
         addu     $s4,$s4,$t4       ;
         mult     $s1,$s4           ;
         mflo     $s4               ; $s4 = r*(z[i+2]+r*y[i+2])

         mult     $s1,$t5           ;
         mflo     $t5               ;
         addu     $s5,$s5,$t5       ;
         mult     $s1,$s5           ;
         mflo     $s5               ; $s5 = r*(z[i+3]+r*y[i+3])

         mult     $s1,$t6           ;
         mflo     $t6               ;
         addu     $s6,$s6,$t6       ;
         mult     $s1,$s6           ;
         mflo     $s6               ; $s6 = r*(z[i+4]+r*y[i+4])

         mult     $s1,$t7           ;
         mflo     $t7               ;
         addu     $s7,$s7,$t7       ;
         mult     $s1,$s7           ;
         mflo     $s7               ; $s7 = r*(z[i+5]+r*y[i+5])

         sw       $s2,$s2($s0)      ; store to $s2 @ af6
         sw       $s3,$s3($s0)      ; store to $s3 @ af6
         sw       $s4,$s4($s0)      ; store to $s4 @ af6
         sw       $s5,$s5($s0)      ; store to $s5 @ af6
         sw       $s6,$s6($s0)      ; store to $s6 @ af6
         sw       $s7,$s7($s0)      ; store to $s7 @ af6

         lui      $v0,0x00C0
         addiu    $v0,$v0,0x80FF    ; wait for $t2-7, $s2-7, $s0
         sw       $v0,0($at)        ; reset af3 itself
         sw       $at,$t1($t1)      ; signal af1
         end                        ; end af3

         ;;------------------
         ;; af7 -> $s0
         ;; af2 -> $a3
         ;; r   -> $s1

af4:     mult     $s1,$t0           ;
```

```
        mflo    $t0             ;
        addu    $t0,$t0,$t1     ;
        mult    $s1,$t0         ;
        mflo    $t0             ;
        addu    $t0,$t0,$t2     ; $t0 = u[i+3]+r*(u[i+2]+r*u[i+1])

        mult    $s1,$t1         ;
        mflo    $t1             ;
        addu    $t1,$t1,$t2     ;
        mult    $s1,$t1         ;
        mflo    $t1             ;
        addu    $t1,$t1,$t3     ; $t1 = u[i+4]+r*(u[i+3]+r*u[i+2])

        mult    $s1,$t2         ;
        mflo    $t2             ;
        addu    $t2,$t2,$t3     ;
        mult    $s1,$t2         ;
        mflo    $t2             ;
        addu    $t2,$t2,$t4     ; $t2 = u[i+5]+r*(u[i+4]+r*u[i+3])

        mult    $s1,$t3         ;
        mflo    $t3             ;
        addu    $t3,$t3,$t4     ;
        mult    $s1,$t3         ;
        mflo    $t3             ;
        addu    $t3,$t3,$t5     ; $t3 = u[i+6]+r*(u[i+5]+r*u[i+4])

        mult    $s1,$t4         ;
        mflo    $t4             ;
        addu    $t4,$t4,$t5     ;
        mult    $s1,$t4         ;
        mflo    $t4             ;
        addu    $t4,$t4,$t6     ; $t4 = u[i+7]+r*(u[i+6]+r*u[i+5])

        mult    $s1,$t5         ;
        mflo    $t5             ;
        addu    $t5,$t5,$t6     ;
        mult    $s1,$t5         ;
        mflo    $t5             ;
        addu    $t5,$t5,$t7     ; $t5 = u[i+8]+r*(u[i+7]+r*u[i+6])

        sw      $t0,$t2($s0)    ; store to $t2 @ af7
        sw      $t1,$t3($s0)    ; store to $t3 @ af7
        sw      $t2,$t4($s0)    ; store to $t4 @ af7
        sw      $t3,$t5($s0)    ; store to $t5 @ af7
        sw      $t4,$t6($s0)    ; store to $t6 @ af7
        sw      $t5,$t7($s0)    ; store to $t7 @ af7

        lui     $v0,0x00FF
        addiu   $v0,$v0,0xC03F  ; wait for $t0-7
        sw      $v0,0($at)      ; reset af4 itself
        sw      $at,$a1($a3)    ; signal af2
        end                     ; end af4

        ;;--------------------
        ;; af7 -> $s0
        ;; af2 -> $a3
        ;; r   -> $s1
        ;; t   -> $a1

af5:    mult    $s1,$t0         ;
        mflo    $t0             ;
        addu    $t0,$t0,$t1     ;
        mult    $s1,$t0         ;
        mflo    $t0             ;
        addu    $t0,$t0,$t2     ;
        mult    $a1,$t0         ;
        mflo    $t0             ; $t0 = t*(u[i+6]+r*(u[i+5]+r*u[i+4]))

        mult    $s1,$t1         ;
        mflo    $t1             ;
        addu    $t1,$t1,$t2     ;
        mult    $s1,$t1         ;
        mflo    $t1             ;
        addu    $t1,$t1,$t3     ;
        mult    $a1,$t1         ;
        mflo    $t1             ; $t1 = t*(u[i+7]+r*(u[i+6]+r*u[i+5]))

        mult    $s1,$t2         ;
        mflo    $t2             ;
        addu    $t2,$t2,$t3     ;
        mult    $s1,$t2         ;
        mflo    $t2             ;
        addu    $t2,$t2,$t4     ;
        mult    $a1,$t2         ;
        mflo    $t2             ; $t2 = t*(u[i+8]+r*(u[i+7]+r*u[i+6]))

        mult    $s1,$t3         ;
        mflo    $t3             ;
        addu    $t3,$t3,$t4     ;
        mult    $s1,$t3         ;
        mflo    $t3             ;
        addu    $t3,$t3,$t5     ;
        mult    $a1,$t3         ;
        mflo    $t3             ; $t3 = t*(u[i+9]+r*(u[i+8]+r*u[i+7]))
        mult    $s1,$t4           ;
        mflo    $t4               ;
        addu    $t4,$t4,$t5       ;
        mult    $s1,$t4           ;
        mflo    $t4               ;
        addu    $t4,$t4,$t6       ;
        mult    $a1,$t4           ;
        mflo    $t4               ; $t4 = t*(u[i+10]+r*(u[i+9]+r*u[i+8]))

        mult    $s1,$t5           ;
        mflo    $t5               ;
        addu    $t5,$t5,$t6       ;
        mult    $s1,$t5           ;
        mflo    $t5               ;
        addu    $t5,$t5,$t7       ;
        mult    $a1,$t5           ;
        mflo    $t5               ; $t5 = t*(u[i+11]+r*(u[i+10]+r*u[i+9]))

        sw      $t0,$s2($s0)      ; store to $s2 @ af7
        sw      $t1,$s3($s0)      ; store to $s3 @ af7
        sw      $t2,$s4($s0)      ; store to $s4 @ af7
        sw      $t3,$s5($s0)      ; store to $s5 @ af7
        sw      $t4,$s6($s0)      ; store to $s6 @ af7
        sw      $t5,$s7($s0)      ; store to $s7 @ af7

        lui     $v0,0x00FF
        addiu   $v0,$v0,0xC03F    ; wait for $t0-7
        sw      $v0,0($at)        ; reset af5 itself
        sw      $at,$a2($a3)      ; signal af2
        end                       ; end af5

        ;;-------------------
        ;; af2 -> $a2
        ;; af3 -> $a3
        ;; af8 -> $t0

af6:    addu    $t2,$t2,$s2       ; $t2 = u[i]+r*(.z.y.)
        addu    $t3,$t3,$s3       ; $t3 = u[i+1]+r*(.z.y.)
        addu    $t4,$t4,$s4       ; $t4 = u[i+2]+r*(.z.y.)
        addu    $t5,$t5,$s5       ; $t5 = u[i+3]+r*(.z.y.)
        addu    $t6,$t6,$s6       ; $t6 = u[i+4]+r*(.z.y.)
        addu    $t7,$t7,$s7       ; $t7 = u[i+5]+r*(.z.y.)

        sw      $t2,$t2($t0)      ; store to $t2 @ af8
        sw      $t3,$t3($t0)      ; store to $t3 @ af8
        sw      $t4,$t4($t0)      ; store to $t4 @ af8
        sw      $t5,$t5($t0)      ; store to $t5 @ af8
        sw      $t6,$t6($t0)      ; store to $t6 @ af8
        sw      $t7,$t7($t0)      ; store to $t7 @ af8

        lui     $v0,0x00C0
        addiu   $v0,$v0,0xC0BF    ; wait for $t0, $t2-7, s2-7
        sw      $v0,0($at)        ; reset af6 itself
        sw      $at,$a3($a2)      ; signal af2
        sw      $at,$s0($a3)      ; signal af3
        end                       ; end af6

        ;;-------------------
        ;; af8 -> $a0
        ;; t   -> $a1
        ;; af4 -> $s0
        ;; af5 -> $s1

af7:    addu    $t2,$t2,$s2       ;
        mult    $a1,$t2           ;
        mflo    $t2               ; $t2 = t*(321+t654)

        addu    $t3,$t3,$s3       ;
        mult    $a1,$t3           ;
        mflo    $t3               ; $t3 = t*(321+t654)

        addu    $t4,$t4,$s4       ;
        mult    $a1,$t4           ;
        mflo    $t4               ; $t4 = t*(321+t654)

        addu    $t5,$t5,$s5       ;
        mult    $a1,$t5           ;
        mflo    $t5               ; $t5 = t*(321+t654)

        addu    $t6,$t6,$s6       ;
        mult    $a1,$t6           ;
        mflo    $t6               ; $t6 = t*(321+t654)

        addu    $t7,$t7,$s7       ;
        mult    $a1,$t7           ;
        mflo    $t7               ; $t7 = t*(321+t654)

        sw      $t2,$s2($a0)      ; store to $s2 @ af8
        sw      $t3,$s3($a0)      ; store to $s3 @ af8
        sw      $t4,$s4($a0)      ; store to $s4 @ af8
        sw      $t5,$s5($a0)      ; store to $s5 @ af8
        sw      $t6,$s6($a0)      ; store to $s6 @ af8
        sw      $t7,$s7($a0)      ; store to $s7 @ af8
        lui     $v0,0x00C0
        addiu   $v0,$v0,0xC0FB   ; wait for $a0, $t2-7, s2-7
        sw      $v0,0($at)       ; reset af7 itself
        sw      $at,$s0($s0)     ; signal af4
        sw      $at,$s0($s1)     ; signal af5
        end                      ; end af7

        ;;--------------------
        ;; af6 -> $s0
        ;; af7 -> $s1
        ;; x   -> $t0
        ;; n   -> $a1
        ;; st2 -> $a0
        ;; wait for $s2-7, $t2-7

af8:    addu    $t2,$t2,$s2      ; $t2 = u[i]+rzy+t321654
        addu    $t3,$t3,$s3      ; $t3 = u[i+1]+rzy+t321654
        addu    $t4,$t4,$s4      ; $t4 = u[i+2]+rzy+t321654
        addu    $t5,$t5,$s5      ; $t5 = u[i+3]+rzy+t321654
        addu    $t6,$t6,$s6      ; $t6 = u[i+4]+rzy+t321654
        addu    $t7,$t7,$s7      ; $t7 = u[i+5]+rzy+t321654
        sw      $t2,0($t0)       ; x[i] = $t2
        sw      $t3,4($t0)       ; x[i+1] = $t3
        sw      $t4,8($t0)       ; x[i+2] = $t4
        sw      $t5,12($t0)      ; x[i+3] = $t5
        sw      $t6,16($t0)      ; x[i+4] = $t6
        sw      $t7,20($t0)      ; x[i+5] = $t7

        move    $t9,$zero
print:  lw      $a2,0($t0)       ; set output parameter 2
        addiu   $t9,$t9,1
        slti    $v0,$t9,6
        addiu   $a1,$a1,1
        addiu   $t0,$t0,4
        jal     0                ; cal printf st2 with a1 & a2
        beq     $v0,$zero,next

        slti    $v0,$a1,15
        bnez    $v0,print

        stop    $s0
        stop    $s1
        stop    $at

next:   sw      $at,$t0($s0)     ; signal af6
        sw      $at,$a0($s1)     ; signal af7
        lui     $v0,0x00C0
        addiu   $v0,$v0,0xC0FF   ; wait for $t2-7, s2-7
        sw      $v0,0($at)       ; reset af8 itself
        end

.data

st1:    ds      "Livermore loop 7:"
st2:    ds      " * x[%d] = %d"

.end
```

Appendix C

The Simulator

C.1 The MulTEP configuration

```
/**************************
 * sim.hh       version 4.1 *
 * June 18th, 2003          *
 * -- Panit Watcharawitch   *
 **************************/

// General
#define MAX_PE            2              // number of PE
#define MAX_CTXT          4              // number of contexts.
#define MAX_REG           64             // registers in each PE
#define MAX_SWQ           32             // max store queue
#define AF_BW             4              // 4 words (128 bits)
#define MAX_IDLE          500            // Change from 64

#define TEMP_SIZE         100
#define INST_SIZE_HEX     8
#define RUN_LENGTH        32
#define TIMER             10000

#define IBLK              0x1000

// Thread Level
#define USER              0
#define SYSTEM            1

// Context Info Signal (From MSU to LSU)
#define NONE              0              // 00
#define UPDATE_PUInfo     1              // 01
#define DEL_PUInfo        3              // 11
#define STORE_DAT         2              // 10

// Register Files Status  // used this now
#define INVALID           0              // 000
#define TRANSIN           1              // 001
#define WAITINST          2              // 010
#define VALID             3              // 011
#define HOLD              4              // 100
#define TRANSOUT          5              // 101
#define TRANSIO           6              // 110
#define USED              7              // 111

// Multithreading Commands (PU -> MSU : 2 bits) - note: NONE is 00
#define STOP_THREAD       1              // 01
#define START_THREAD      2              // 10
#define SWITCH            3              // 11

// DATA SIZE
#define BYTE             8
#define HALFWORD         16
#define WORD             32

// Definitions for PE
#define RESET_PC          0x40000040
#define EXITCLK           10000000

/******* Definitions for MMU (in words) ***********/
#define MAX_MEM           0x680000       // 26M Byte Physical Memory
#define MAP_ENTRIES       MAX_MEM/0x400  // lower 12 bits for page offset
// MMU Virtual Space
```

```
#define DM_MEM            0x00000000          // DM Area    0x00000000-0x00000FFF
#define USER_MEM          0x00001000          // User Area 0x00001000-0xFEFFFFFF
#define START_SP          0xF0000000          // Start Stack Area
#define END_SP            0xF001FFFF          // End Stack Area
#define AV_MEM            0xFD000000          // AV Area    0xFD000000-0xFD0003FF
#define WJ_MEM            0xFD000400          // WJ Area    0xFD000400-0xFD0007FF
#define WT_MEM            0xFD000800          // WT Area    0xFD000800-0xFD000BFF
#define WN_MEM            0xFD000C00          // WN Area    0xFD000C00-0xFD000FFF
#define END_WN            0xFD001000          // END WN Area
#define TL_MEM            0xFE000000          // TL Area    0xFD001000-0xFD001FFF
#define START_AP          0xFF000000          // AF Area    0xFF000000-0xFFFFFFFF
#define MAX_VM            0xFFFFFFFF          // Max Virtual Memory
// General Definition
#define SYS_MEM           0xFD000000
#define SYS_MASK          0xFF000000
#define AV_BLK            8
#define WJ_BLK            12
#define WT_BLK            12
#define WN_BLK            8
#define TL_TAG1           10                  // Translation Tag1
#define TL_TAG2           10                  // Translation Tag2
#define TL_OFFSET         12                  // Translation Offset
#define TL_TAG_MASK       0x3FF               // Translation Tag Mask
#define TL_MASK           0xFFF               // Translation Mask
#define PHY_BIT           9                   // Physical Page Bits
#define PHY_MASK          0x1FF               // Physical Page Mask
#define LRU_SIZE          3                   // LRU Size
#define MAX_DTLB          32                  // MAX Data TLB
#define MAX_ITLB          32                  // MAX Instruction TLB
#define MAX_AFTLB         16                  // MAX AF TLB
#define LQUEUE_SIZE       32                  // Load FIFO buffer
#define D_LATENCY         5                   // PE-D clock-cycle latency
#define L1_LATENCY        5                   // L0-L1 clock-cycle latency
#define MEM_LATENCY       200                 // L1/D-MEM clock-cycle latency
#define TL_LATENCY        200                 // TLB-MEM clock-cycle latency
// L0 Instruction Cache                                  -- Fully associative (no idx) --
#define L0_SIZE           0x100                          // 1KB L0-Cache for inst
#define L0_BLOCK_SIZE     0x20                           // L0 Block Size (32W -> 128B)
#define L0_SET            L0_SIZE/L0_BLOCK_SIZE          // 8 Sets
#define L0_OSIZE          7                              // 2^(5+2) = 32*4
#define L0_OMASK          0x1F                           // 5 bits
#define L0_TAG_MASK       0x3FFF                         // 14 bits (+7 = 21 bits)
// L1 Instruction Cache                                  -- Direct Mapping --
#define L1_SIZE           0x1000                         // 16KB L1-Cache for inst
#define L1_BLOCK_SIZE     0x100                          // L1 Block Size (1KB)
#define L1_SET            L1_SIZE/L1_BLOCK_SIZE          // 16 Sets
#define L1_OSIZE          10                             // 2^(8+2) = 256*4
#define L1_OMASK          0xFF                           // 8 bits
#define L1_INOFF_SIZE     4+L1_OSIZE                     // 2^4 = 16 = 4K/256
#define L1_IDX_MASK       0xF                            // 4 bits
#define L1_TAG_MASK       0x7F                           // 7 bits (+4+10 = 21 bits)
// Data Cache                                            -- 4-way set-associative --
#define D_WAY             4                              // 4-ways (2 bits)
#define D_SIZE            0x1000                         // 16KB D-Cache for data
#define D_BLOCK_SIZE      0x20                           // Cache Block Size (128 B)
#define D_SET             D_SIZE/D_BLOCK_SIZE            // 128 sets
#define D_OSIZE           7                              // 2^(5+2) = 32*4
#define D_OMASK           0x1F                           // 5 bits
#define D_INOFF_SIZE      7+D_OSIZE                      // 2^7 = 4K/32
#define D_IDX_MASK        0x7F                           // 7 bits
#define D_TAG_MASK        0x7F                           // 7 bits (+7+5+2 = 21 bits)
// Activation Frame Cache                                -- Fully associative --

// For Activation Frame holding all 64 registers
#define ZERO              0                   // Zero
#define PP                0                   // priority/presence bits ($zero)
#define AT                1                   // micro-thread code pointer
#define V0                2                   // values for results & exp.
#define V1                3                   // values for results & exp.
#define A0                4                   // arguments
#define A1                5                   // arguments
#define A2                6                   // arguments
#define A3                7                   // arguments
#define T0                8                   // temporaries
#define T1                9                   // temporaries
#define T2               10                   // temporaries
#define T3               11                   // temporaries
#define T4               12                   // temporaries
#define T5               13                   // temporaries
#define T6               14                   // temporaries
#define T7               15                   // temporaries
#define S0               16                   // saved
#define S1               17                   // saved
#define S2               18                   // saved
#define S3               19                   // saved
#define S4               20                   // saved
#define S5               21                   // saved
#define S6               22                   // saved
#define S7               23                   // saved
#define T8               24                   // more temporaries
#define T9               25                   // more temporaries
#define K0               26                   // operating system reserve
#define K1               27                   // operating system reserve
#define GP               28                   // global pointer
#define SP               29                   // stack pointer
```

```
#define FP              30              // frame pointer
#define RA              31              // return address
#define F0              32              // FPreg 0
#define F1              33              // FPreg 1
#define F2              34              // FPreg 2
#define F3              35              // FPreg 3
#define F4              36              // FPreg 4
#define F5              37              // FPreg 5
#define F6              38              // FPreg 6
#define F7              39              // FPreg 7
#define F8              40              // FPreg 8
#define F9              41              // FPreg 9
#define F10             42              // FPreg 10
#define F11             43              // FPreg 11
#define F12             44              // FPreg 12
#define F13             45              // FPreg 13
#define F14             46              // FPreg 14
#define F15             47              // FPreg 15
#define F16             48              // FPreg 16
#define F17             49              // FPreg 17
#define F18             50              // FPreg 18
#define F19             51              // FPreg 19
#define F20             52              // FPreg 20
#define F21             53              // FPreg 21
#define F22             54              // FPreg 22
#define F23             55              // FPreg 23
#define F24             56              // FPreg 24
#define F25             57              // FPreg 25
#define F26             58              // FPreg 26
#define F27             59              // FPreg 27
#define F28             60              // FPreg 28
#define F29             61              // FPreg 29
#define F30             62              // FPreg 30
#define F31             63              // FPreg 31

// For MSU
#define MAX_AF          32              // Will correct soon
#define START_LOCAL     V0
#define END_LOCAL       T9
#define PRESENCE        (END_LOCAL-START_LOCAL+1)
#define NOT_TRANS       MAX_REG-PRESENCE-2
#define PRIORITY_MASK   0xFF
#define PRESENCE_LOC    2
#define PRESENCE_MASK   0xFFFFFF
#define MAX_GTRANS      MAX_REG/AF_BW

// For LSU
#define MAX_LWAIT  16

#define lID(i)          (Ctxt[i].r[AT]&0xFFFF)
#define gID(add)        (((add)>>16)&0xFFFF)
#define thID(add)       (((add)>>7)&0xFFFF)
#define regID(add)      (((add)>>2)&0x1F)
#define thEq(i,n)       (lID(i)==n)
#define getpri(pp)      (((pp)>>PRESENCE)&PRIORITY_MASK)
#define APadd(th,r)     (START_AP+((th)<<7)+((r)<<2))
#define presence(x)     (((x)&PRESENCE_MASK)==PRESENCE_MASK)

// Global non-clock component;
struct Context {
  short status;                   // 2  bits
  REG   pc;                       // 30 bits
  DAT   fetch;                    // 32 bits
  BIT   inst;                     // 1  bit
  REG   r[MAX_REG];               // 32 x 32 bits
};

FILE  *flog;
short workload, numthread;

DAT   SE_AV, SE_WJ, SE_WT, SE_WN; // start empty pointers of AF tables
DAT   EE_AV, EE_WJ, EE_WT, EE_WN; // end empty pointers oo AF tables
DAT   ED_AV, ED_WJ, ED_WT, ED_WN; // end data pointers oo AF tables

// *** Global function ***
char *r_name(char r) {
  switch (r) {
  case ZERO: return "$zero";  break; // the constant value 0
  case AT:   return "$at";    break; // reserved for assembler
  case V0:   return "$v0";    break; // val for result & expr eval
  case V1:   return "$v1";    break; // val for result & expr eval
  case A0:   return "$a0";    break; // arg
  case A1:   return "$a1";    break; // arg
  case A2:   return "$a2";    break; // arg
  case A3:   return "$a3";    break; // arg
  case T0:   return "$t0";    break; // temp
  case T1:   return "$t1";    break; // temp
  case T2:   return "$t2";    break; // temp
  case T3:   return "$t3";    break; // temp
  case T4:   return "$t4";    break; // temp
  case T5:   return "$t5";    break; // temp
  case T6:   return "$t6";    break; // temp
  case T7:   return "$t7";    break; // temp
  case S0:   return "$s0";    break; // saved
  case S1:   return "$s1";    break; // saved
```

```
    case S2:    return "$s2";    break; // saved
    case S3:    return "$s3";    break; // saved
    case S4:    return "$s4";    break; // saved
    case S5:    return "$s5";    break; // saved
    case S6:    return "$s6";    break; // saved
    case S7:    return "$s7";    break; // saved
    case T8:    return "$t8";    break; // more temp
    case T9:    return "$t9";    break; // activation frame pointer
    case K0:    return "$k0";    break; // reserved for OS
    case K1:    return "$k1";    break; // reserved for OS
    case GP:    return "$gp";    break; // global pointer
    case SP:    return "$sp";    break; // stack pointer
    case FP:    return "$fp";    break; // frame pointer
    case RA:    return "$ra";    break; // return address
    case F31:   return "$f31";   break; // floating point register
    case F30:   return "$f30";   break; // floating point register
    case F29:   return "$f29";   break; // floating point register
    case F28:   return "$f28";   break; // floating point register
    case F27:   return "$f27";   break; // floating point register
    case F26:   return "$f26";   break; // floating point register
    case F25:   return "$f25";   break; // floating point register
    case F24:   return "$f24";   break; // floating point register
    case F23:   return "$f23";   break; // floating point register
    case F22:   return "$f22";   break; // floating point register
    case F21:   return "$f21";   break; // floating point register
    case F20:   return "$f20";   break; // floating point register
    case F19:   return "$f19";   break; // floating point register
    case F18:   return "$f18";   break; // floating point register
    case F17:   return "$f17";   break; // floating point register
    case F16:   return "$f16";   break; // floating point register
    case F15:   return "$f15";   break; // floating point register
    case F14:   return "$f14";   break; // floating point register
    case F13:   return "$f13";   break; // floating point register
    case F12:   return "$f12";   break; // floating point register
    case F11:   return "$f11";   break; // floating point register
    case F10:   return "$f10";   break; // floating point register
    case F9:    return "$f9";    break; // floating point register
    case F8:    return "$f8";    break; // floating point register
    case F7:    return "$f7";    break; // floating point register
    case F6:    return "$f6";    break; // floating point register
    case F5:    return "$f5";    break; // floating point register
    case F4:    return "$f4";    break; // floating point register
    case F3:    return "$f3";    break; // floating point register
    case F2:    return "$f2";    break; // floating point register
    case F1:    return "$f1";    break; // floating point register
    case F0:    return "$f0";    break; // floating point register
    default:    return "";
    }
}
```

C.2 The SimpleScalar configuration

```
-fetch:ifqsize  2                   # Fetch width per PE (2 PEs)
-fetch:speed    2                   # Front end fetch ratio 1:1 for each FU
-decode:width   1                   # decode 1 instruction per PE
-issue:width    1                   # issue 1 instruction per PE
-issue:inorder                      # In order execution (remove when OOO required)
-ruu:size       2                   # 2 register file for 2 PEs
-lsq:size       8                   # load/store queue size
-res:ialu       1                   # an integer ALU
-res:imult      1                   # an interger mult (emulates FP dividers)
-res:fpalu      1                   # a floating point ALUs
-res:fpmult     1                   # a floating point multiplers
-bpred          nottaken            # Nottaken branch preditor
-cache:dl1lat   1                   # Activation Frame access latency
-cache:dl1      dl1:16:64:32:l      # AF, 32 sets, 64 regs, full assc, LRU
-cache:dl2lat   5                   # Level 1 data access latency
#-cache:dl2     dl2:64:16:4:l       # L1-Dcache, 128 sets, 16 words, 4 ways, random
-cache:il1lat   1                   # Level 0 instruction access latency
-cache:il1      il1:4:64:8:l        # L0-Icache, 8 sets, 64 words, full assc, LRU
-cache:il2lat   5                   # Level 1 instruction access latency
-cache:il2      il2:8:128:1:l       # L0-Icache, 8 sets, 64 words, full assc, LRU
-mem:lat        200 1               # Main memory access latency [first rest]
-mem:width      8                   # Memory bus 8 bytes
-tlb:lat        200                 # TLB miss 200 cycles
-tlb:dtlb       dtlb:16:2048:4:l    # 32 entries data TLB
-tlb:itlb       itlb:16:2048:4:l    # 32 entries instruction TLB
#-icompress                         # Compress instruction to 32 bits
```

C.3 The Thread-0 System Daemon

```
        ;; $t0 = timer
        ;; $s0 = number of activation threads
        ;; $s1 = number of wait_for_join threads
        ;; $s2 = number of wait_for_timer threads
        ;; $s3 = number of wait_for_notify threads
        ;; $s4 = pointer to active-thread table
        ;; $s5 = pointer to wait_for_join table
        ;; $s6 = pointer to wait_for_timer table
        ;; $s7 = pointer to wait_for_notification table
.text

main:   wait    $t0               ; Th0, wait for timing (contains timer info)

        move    $t9,$s2
        move    $t5,$s6
Mtimer: bne     $t9,$zero,Ctime   ; Check wait_for_timer entries

        move    $t9,$s1
        move    $t5,$s5
Mjoin:  bne     $t9,$zero,Cjoin   ; Check wait_for_join entries

Mnoti:  bne     $s3,$zero,Cnoti   ; Check wait_for_notify entries
        j       main

        ;; Check wait_for_join entries

Cjoin:  lw      $a1,4($t5)        ; Waiting for Thread ID (in a form of AF loc)
        lw      $a2,8($t5)        ; Next wait_for_join entry

        move    $t7,$s4
        move    $t4,$s0
NTID:   beq     $t4,$zero,Ajoin   ; No Thread - Activate wait for join
        lw      $v0,0($t7)        ; Activate Thread ID (in a form of AF loc)
        lw      $v1,4($t7)        ; Next activate thread
        beq     $v0,$a1,Jnext     ; Equal - break (the entry has to 'wait')
        move    $t7,$v1           ; Not Equal - check next ThID
        addi    $t4,$t4,-1
        j       NTID

Ajoin:  lw      $a0,0($t5)        ; Request ThID & reg (in a form of AF loc)
        addi    $s1,$s1,-1        ; Remove the activated entry
        sw      $k1,0($a0)        ; Activate wait_for_join entry
        bne     $s5$,$t5,Nst1     ; $t5 isn't a start position
        move    $s5,$a2           ; Remove the first entry
        j       Jnext

Nst1:   sw      $a2,8($t6)
Jnext:  move    $t6,$t5
        move    $t5,$a2
        addi    $t9,$t9,-1
        j       Mjoin             ; Check more wait_for_join entries

        ;; Check wait_for_timer entries

Ctime:  lw      $a0,0($t5)        ; Timed Thread ID
        lw      $a1,4($t5)        ; Timer
        lw      $a2,8($t5)        ; Next wait_for_timer entry

        sub     $a1,$a1,$t0       ; Reduce timer
        blez    $a1,Txpire        ; Time expire
        sw      $a1,4($t5)        ; store new timer
        j       Tnext

Txpire: sw      $k1,0($a0)        ; Activate wait_for_timer entry
        addi    $s2,$s2,-1        ; Remove the activated entry
        bne     $s5$,$t5,Nst2     ; $t5 isn't a start position
        move    $s5,$a2
        j       Tnext

Nst2:   sw      $a2,8($t6)
Tnext:  move    $t6,$t5
        move    $t5,$a2
        addi    $t9,$t9,-1
        j       Mtimer

        ;; Check wait_for_notification entries

Cnoti:  lw      $a0,0($s7)
        lw      $a2,4($s7)
        sw      $k1,0($a0)        ; Activate wait_for_notify entry
        move    $s7,$a2
        addi    $s3,$s3,-1
        j       Mnoti

.end
```

Appendix D

Additional BNF for MulTEP assembler

```
<Lopcode>    →  lb | lbu | ld | ldc1 | ldc2 | ldl | ldr | lh | lhu | ll |
                lld | lw | lwc0 | lwc1 | lwc2 | lwc3 | lwl | lwr
<Sopcode>    →  sb | sc | scd | sdc1 | sdc2 | sdl | sdr | sh | sw | swc0 |
                swc1 | swc2 | swc3 | swl | swr
<Ropcode>    →  sll | sllv | sll | slt | sltu | sra | srav | srl | srlv |
                sub | add | addu | and | dadd | movn | movz | nor | or |
                xor | cvt.s.w | cvt.d.s | l.s
<Iopcode>    →  slti | sliu | addi | addiu | andi | daddi | daddiu |
                ori | pref | xori
<Bopcode>    →  beq | beql | bne | bgt | bnel
<BZopcode>   →  bgez | bgezal | bgezl | bgtz | bgtzl | blez | blezl |
                bltz | bltzal | bltzall | blezl | bnez
<Sopcode>    →  spawn
<Jopcode>    →  j | jal2 | jalr | jr
<ROpcode2>   →  div | div.s | divu | dsll | dsll32 | dsllv | dsra |
                dsra32 | dsrav | dsrl | dsrl32 | dsrlv | dsub |
                dsubu | move | mult | multu | teq | tge | tgeu | tlt |
                tltu | tne | abs.s | abs.d | add.s | add.d |
                c.cond.s | c.cond.d | ceil.l.s | ceil.l.d |
                ceil.w.s | ceil.w.d | cfc1 | ctc1 | signal
<IOpcode2>   →  li | lui | teqi | tgei | tgeiu | tlti | tltiu
```

<Opcode1>	→	cop0 | cop1 | cop2 | cop3 | mfhi | mflo | mthi | mtlo | bc1f | bc1fl | bc1t | bc1tl | wait | stop
<Opcode0>	→	break | sync | syscall | switch | end | nop
<cond>	→	z | lz | gtz | eq | lt | lte | gt | gte
<reg>	→	$zero | $at | $v0 | $v1 | $a0 | $a1 | $a2 | $a3 | $t0 | $t1 | $t2 | $t3 | $t4 | $t5 | $t6 | $t7 | $s0 | $s1 | $s2 | $s3 | $s4 | $s5 | $s6 | $s7 | $t8 | $t9 | $k0 | $k1 | $gp | $sp | $fp | $ra | $f0 | $f1 | $f2 | $f3 | $f4 | $f5 | $f6 | $f7 | $f8 | $f9 | $f10 | $f3 | $f11 | $f12 | $f13 | $f14 | $f15 | $f16 | $f17 | $f18 | $f19 | $f20 | $f21 | $f22 | $f23 | $f24 | $f25 | $f26 | $f27 | $f28 | $f29 | $f30 | $f31

Bibliography

[1] P. Koopman. Embedded System Design Issue (the Rest of the Story). In *IEEE International Conference on Computer Design*, 1996.

[2] M. Sgroi and L. Lavagno and A. S. Vincentelli. Formal Models for Embedded System Design. *IEEE Design & Test of Computers*, 17(2):14–27, June 2000.

[3] David E. Culler. Multithreading: Fundamental Limits, Potential Gains, and Alternatives. *Multithreaded Computer Architecture*, pages 97–138, 1994.

[4] Simon W. Moore. *Multithreaded Processor Design.* Kluwer Academic Publishers, Boston, June 1996.

[5] R. A. Iannucci. *Multithreaded Computer Architecture: A Summary of the State of the Art.* Kluwer Academic Publishers, January 1994.

[6] G. T. Byrd and M. A. Holliday. Multithreaded Processor Architectures. *IEEE Spectrum*, pages 38–46, August 1995.

[7] J. Tsai and Z. Jiang and E. Ness and P. Yew. Performance Study of a Concurrent Multithreaded Processor. In *HPCA-4*, pages 24–35, February 1998.

[8] Simon W. Moore and B. T. Graham. Tagged Up/Down Sorter – A Hardware Priority Queue. *The Computer Journal*, 38(9), 1995.

[9] L. H. McClure. Embedded System Design. In *Lecture Note*. Department of Electrical & Computer Engineering , University of Colorado, 1999.

[10] International Technology Roadmap for Semiconductors. Technical report, Semiconductor Industry Association (SIA), 2002.

[11] David A. Patterson. Reduced Instruction Set Computers. *Comm. ACM*, 28(1):8–21, January 1985.

[12] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *Computer Architecture News 8:6*, pages 25–33, October 1980.

[13] D. Burger and J. R. Goodman. Billion-Transistor Architectures. *IEEE Computer*, pages 46–50, September 1997.

[14] Christoforos E. Kozyrakis and David A. Patterson. A New Direction for Computer Architecture Research. *IEEE Computer Magazine*, pages 24–32, November 1998.

[15] Intel Architecture Software Developer's Manual. *Volume 1: Basic Architecture.* Intel, 2001.

[16] *M68000 8-/16-/32-Bit Microprocessors User's Manual.* Motorola Inc., 1993.

[17] T.E. Leonard. *VAX architecture reference manual.* Digital Press, 1987.

[18] *PDP 11/45 Processor Handbook.* Digital Equipment Corporation (DEC), 1971.

[19] AMD Athlon Processor. Technical report, Advanced Micro Devices, Inc., 1999.

[20] Francisco Barat, Rudy Lauwereins and Geert Deconnick. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. In *IEEE Transactions on Software Engineering*, volume 28, pages 847–862. Institute of Electrical and Electronics Engineers, Inc., September 2002.

[21] M.A. Shutte and J.P. Shen. Instruction-level experimental evaluation of the Multiflow TRACE 14/300 VLIW computer. In *The Journal of Supercomputing*, pages 7:249–271, 1993.

[22] Tom R. Halfhill. Inside IA-64. *Byte Magazine*, 23(6):81–88, June 1998.

[23] D. Patterson. New Direction in Computer Architecture. In *PARCON: Symposium on New Directions in Parallel and Concurrent Computing*, November 1998.

[24] John Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Faufmann, 2 edition, 1996.

[25] W. Buchholz. The IBM System/370 Vector Architecture. Technical report, June 1986.

[26] D. A. Patterson and J. Hennessy. *Computer Organization and Design.* Morgan Kaufmann Publisher, 1996.

[27] *Alpha Architecture Handbook.* Digital Equipment Corporation (DEC), 1992.

[28] J. S. Kowalik. *Parallel MIMD computation: the HEP Supercomputer and its applications.* MIT Press, 1985.

[29] R. Alverson and D. Callahan and D. Cummings and B. Koblenz and A. Porterfield and B. Smith. The Tera Computer System. In *ICS*, pages 1–6, June 1990.

[30] S. J. Eggers and J. S. Emer and H. M. Levy and J. L. Lo and R. L. Stamm and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, pages 12–19, September/October 1997.

[31] M. V. Wilkes. Memoirs of a computer pioneer. *MIT Press*, 1985.

[32] P.V. Argade and et al. Hobbit – a high-performance, low-power microprocessor. In *38th annual IEEE CompCon, San Francisco, CA D930222-26*, Spring 1993.

[33] D.P. Siewiorek, C.G. Bell and A. Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, 1982.

[34] William Stalling. *Computer Organization and Architecture.* Prentice Hall, 2000.

[35] David L. Weaver and Tom Germond. *The SPARC Architecture Manual.* SPARC International, Inc.

[36] *R4300i Microprocessor Data Sheet.* MIPS Open RISC Technology, April 1997.

[37] IBM Corp. *The PowerPC Architecture: A Specification for the New Family of RISC Processors.* Morgan Kaufmann, 1994.

[38] Gerry Kane. *PA-RISC 2.0 Architecture.* Prentice Hall PTR.

[39] David Seal. *ARM Architectural Reference Manual.* Addison-Wesley, 2^{nd} edition, 2000.

[40] *SH-4 Core Architecture Manual.* SuperH, Inc.

[41] M. V. Wilkes and J. S. Stringer. Micro Programming and the Design of the Control Circuits in an Electronic Digital Computer. *Proc. Cambridge Philosophy Society*, 49, 1953.

[42] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective. Technical report, Computer Systems Laboratory, Hewlett Packard, October 1992.

[43] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1989.

[44] Michael D. Smith, Mike Johnson and Mark A. Horowitz. Limits on multiple instruction issue. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1989.

[45] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of parallel instruction. *IEEE Transactions on Computers C-19*, 1970.

[46] David W. Wall. Limits of Instruction-Level Parallelism. Technical report, Western Research Laboratory, Digital Inc., December 1990.

[47] Paul Mazzucco. Fundamentals of Multithreading. *SLCentral*, June 2001.

[48] A. Burns. Scheduling hard real-time systems: a review. Technical report, Department of Computing, University of Bradford, 1988.

[49] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computer*, pages 948–960, 1972.

[50] Jack B. Dennis. The varieties of data flow computer. In *1st International Conference in Distributed Computer System*, pages 430–439, October 1979.

[51] J. B. Dennis. *Data flow supercomputers.* IEEE Computer, 1980.

[52] Arvind, K.P. Gostelow and W. Plouffe. An asynchronous programming language and computing machine. Technical report, Universities of California at Irvine, December 1978.

[53] D. A. Adams. A Computation Model With Data Flow Sequencing. Technical report, Computer Science Department, Stanford University, December 1968.

[54] J. A. Sharp. *Data flow Computing: Theory and Practice*. Ablex Publishing Corporation, 1992.

[55] M. Cornish. The TI dataflow architecture: The power of concurrency for avionics. In *3rd Conference Digital Avionics System*, pages 19–25, November 1979.

[56] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. pages 210–215, April 1978.

[57] J. R. Gurd and et al. The Manchester Prototype Dataflow computers. *Communication of the ACM*, January 1985.

[58] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computer C-39*, pages 300–318, 1990.

[59] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 82–91. Seattle, May 1990.

[60] S. Sakai and et al. Prototype implementation of a highly parallel dataflow machine EM-4. In *5th International Parallel Processing Symposium*. IEEE, 1991.

[61] V. G. Grafe and J. E. Hoch. The Epsilon-2 multiprocessor system. In *J. Parallel Distributed Computer 10*, pages 309–318, 1990.

[62] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *16th Annual International Symposium on Computer Architecture*, pages 262–272, 1989.

[63] B A. Maynard. Honeywell 800 System. *Manual of Computer System*, 1964.

[64] P Dreyfus. System design of the Gamma 60. In *Proceedings of the Western Joint Computer Conference*, pages 130–133, 1958.

[65] D. Culler and A. Sah, K. Schauser and T. von Eicken and T. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-controlled Threaded Abstract Machine. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[66] A. Agarwal and el al. The MIT Alewife machine: Architecture and performance. In *Proc. 22nd ISCA*, pages 2–13, June 1995.

[67] J. Silc and B. Robic and Theo Ungerer. Asynchrony in Parallel Computing: From Dataflow to Multithreading. In *Parallel and Distributed Computing Practises*, volume 1:1. March 1998.

[68] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings for the 19th Annual International Symposium on Computer Architecture*, 1992.

[69] INMOS. *Transputer Reference Manual*. Prentice Hall, 1988.

[70] R. S. Nikhil and G. M. Papadopoulos and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *ISCA-19*, pages 156–167, May 1992.

[71] G. R. Gao. An efficient hybrid dataflow architecture. In *Journal of Parallel and Distributed Computing*, pages 19:293–307, 1993.

[72] Jean Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distribution.* Addison-Wesley Longman, Incorporated, 2 edition, January 1997.

[73] Arvind and S. A. Brobst. The evaluation of dataflow architectures from static dataflow to P-RISC. In *International Journal in High Speed Computing*, pages 125–153, 1993.

[74] William J. Dally and Stephen W. Keckler and Nick Carter and Andrew Change and Marco Fillo and Whay S. Lee. M-Machine Architecture v1.0. Technical report, MIT, August 1994.

[75] G. M. Papadopoulos. Implementation of a general purpose dataflow multiprocessor. In *Research Monographs in Parallel and Distributed Computing.* MIT Press, 1991.

[76] D. Bitton and D. J. DeWitt and D. K. Hsiao and J. Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3):287–318, 1984.

[77] E.W. Dijkstra. A heuristic explanation of Batcher's baffler. *Science of Computer Programming*, 9:213–220, 1987.

[78] Yen-Chun Lin. On balancing sorting on a linear array. *IEEE Transactions on Parallel and Distributed Systems*, 4, May 1993.

[79] E. Rotenberg and Q. Jacobson and Y. Sazeides and J. Smith. Trace Processors. In *International Symposium on Microarchitecture*, pages 138–148, 1997.

[80] J. Y. Tsai and P. C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-time Data dependence checking and control speculation. In *PACT'96*, pages 35–46, October 1996.

[81] E. Rotenberg and S. Bennett and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *the 29th International Symposium on Microarchitecture*, pages 24–34, December 1996.

[82] Deborah T. Marr and et al. Hyper-Threading Technology Architecture and Microarchitecture. Technical report, Intel Technology Journal Q1, 2002.

[83] D. M. Tullsen and S. J. Eggers and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA 22*, pages 392–403, June 1995.

[84] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *Special Issue on Multithreaded Architecture, IEEE Transactions on Computer*, December 1999.

[85] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *13th ICS*, June 1999.

[86] MAJC Architecture Tutorial, White Paper. Technical report, Sun Microsystems, San Antonio, USA.

[87] K. Boland and A. Dollas. Predicting and precluding problems with memory latency. In *IEEE Micro*, volume 14(8), pages 59–67, 1994.

[88] John P. Shen. Multi-Threading for Latency. In *MTEAC-5 Keynote*, 2001.

[89] Krishna M. Kavi and Hee Yong Youn and Ali R. Hurson. PL/PS: A Non-Blocking Multithreaded Architecture with Decoupled Memory And Pipelines. In *The fifth International Conference on Advanced Computer (ADCOMP)*, 1997.

[90] Laurence Vanhelsuwé and et al. *Mastering Java*. BPB Publications, 1996.

[91] R. S. Nikhil. The parallel programming language ID and its complication for parallel machines. *International Journal High Speed Computing 5*, pages 171–223, 1993.

[92] J. McGraw and S. Skedzielewski and S. Allan and D. Grit and R. Odehoeft and J. Glauert and J. Glauert and P. Hohensee and I. Dobes. SISAL Reference Manual. Technical report, Lawrence Livermore National Laboratory, 1984.

[93] D. Callahan and B. Smith. A Future-based Parallel Language for a General-purpose Highly-parallel Computer. *Languages and Compilers for Parallel Computing*, pages 95–113, 1990.

[94] Dick Pountain. A Tutorial introduction to OCCAM programming. Technical report, INMOS.

[95] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Programming Languages and Systems*, pages 501–538, 1985.

[96] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceeding of the 15th ISCA*, pages 443–451, May 1988.

[97] Steve Kleiman and Devang Shah and Bart Smaalders. *Programming with pthread.* Sun Microsystem, 1996.

[98] *ISO/IEC Standard 9945-1: Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C language].* Institute of Electrical and Electronic Engineers, 1996.

[99] K. Hwang. *Advanced computer architecture: Parallelism, Scalability, Programmability.* McGraw-Hill Series in Computer Science, march 1993.

[100] Anant Agarwal. Performance tradeoffs in multithreaded processors. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology (MIT), April 1991.

[101] Leonard Kleinrock. *Queueing Systems*, volume 1. John Wiley & Sons, 1975.

[102] P. Koopman. Challenges in Embedded Systems Research & Education. *Presentation, Electrical & Computer Engineering Department, Carnegie Mellon*, 2001.

[103] S. W. Keckler and A. Chang and W. S. Lee and S. Chatterjee and W. J. Dally. Concurrent Event Handling through Multithreading. *IEEE Computers*, September 1999.

[104] AMD29000 User's Manual. Technical report.

[105] K. Lüth and A. Metzner and T. Peikenkamp and J. Risau. The EVENTS Approach to Rapid Prototyping for Embedded Control Systems. In *ZES-ARCS*, Germany, September 1997.

[106] *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, Inc., 2000.

[107] G. Bollella and J. Gosling and B. M. Brosgol and P. Dibble and S. Furr and D. Hardin and M. Turnbull. *The Real-time Specification for Java*. Addison Wesley, 2000.

[108] *Ada 95 Reference Manual*. Intermetrics, January 1995.

[109] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[110] A. Burns and B. Dobbing and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, volume 1411, pages 263–275, 1998.

[111] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, Chong Sang Kim. An Accurate Worst Case Timing Analysis For RISC Processors. *IEEE Transactions on Software Engineer*, July 1995.

[112] Tadahiro Kuroda. Low-Power, High-Speed CMOS VLSI Design. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002.

[113] Managing Power in Ultra Deep Submicron ASIC/IC Design. Technical report, Synopsys Inc., May 2002.

[114] Sameer Patel. Low-power design techniques span RTL-to-GDSII flow. *EEdesign*, June 2003.

[115] Zhanhai Qin. Power Issue in the Deep-submicron Era. Technical report, University of California, San Diego, 2003.

[116] Y. Lee and C. Krishna. Voltage Clock Scaling for Low Energy Consumption in Real-Time Embedded Systems. In *6th International Conference on Real-Time Computing Systems and Application*, December 1999.

[117] J. Loch and A. Smith. Improving dynamic voltage scaling algorithm with pace. In *Symmetrics*, June 2001.

[118] Jinson Koppanalil and Prakash Ramrakhyani and Sameer Desai and Anu Vaiyanathan and Eric Rotenberg. A Case for Dynamic Pipeline Scaling. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2002.

[119] David Brooks and Magaret Martonosi. Value-based clock gating and operation packing: Dynamic Strategies for improving processor power and performance. *ACM Transactions on Computer Systems*, 18(2):89–126, may 2000.

[120] J. Turley. *Trends in Microprocessors for Embedded Applications.* MicroDesign, 1999.

[121] Ralph H.J.M. Otten and Paul Stravers. Challenges in Physical Chip Design. In *ICCAD*, 2000.

[122] T. Noll. Designing Complex SOC's for Wireless Communications. In *ISLPED*, 2002.

[123] PNX1300 Series Product Profile. Technical report, Philips.

[124] NP-1 Family The world's Most Highly Integrated 10-Gigabit 7-Layer Network Processors. Technical report, 1999.

[125] Clive Maxfield. Field-programmable devices. Technical report, 1996.

[126] Bengu li and Rajiv Gupta. Bit Section Instruction Set Extension of ARM for Embedded Applications. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2002.

[127] J. W. Haskins and K. R. Hirst and K. Skadron. Inexpensive Throughput Enhancement in Small-Scale Embedded Microprocessors with Block Multithreading: Extensions, Characterization, and Tradeoffs. In *Proc. of the 20th IEEE International Performance, Computing and Communication Conference*, April 2001.

[128] C.D. Polychronopoulos and N. Bitar and S. Kleiman. Nanothreads: A user-level threads architecture. Technical report, CSRD, Univ. of Illinois at Urbana-Champaign, 1993.

[129] L. Gwennap. DanSoft develops VLIW design. *Microprocessor Report*, 11:18–22, February 1997.

[130] Charles Price. MIPS IV Instruction Set. Technical report, MIPS Technologies, Inc., 1995.

[131] Panit Watcharawitch and Simon Moore. JMA: The Java-Multithreading Architecture for Embedded Processors. In *International Conference on Computer Design (ICCD).* the IEEE Computer Society, September 2002.

[132] Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory.* John Wiley and Sons, Inc., Third edition, 1998.

[133] Kevin Skadron and Pritpal S. Ahuja and Margaret Martonosi and Douglas W. Clark. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques. *IEEE Transactions on Computer*, 48(11), November 1999.

[134] Tiger SHARC Embedded Processor. Technical report, 2003.

[135] Launchbird Design Systems and Arithlogic. Project: Confluence Floating Point Multiplier (http://www.opencores.org/projects/cf_fp_mul). Technical report, Opencores.org, March 2003.

[136] IEEE 754, Standard for Binary Floating-Point Arithmetic. Technical report, Institute of Electrical and Electronics Engineers, Inc., 1985.

[137] DFPDIV: Floating Point Pipelined Divider Unit. Technical report, Digital Core Design.

[138] Peter J. Denning and Jeffrey P. Buzen. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys (CSUR)*, 10(3):225–261, September 1978.

[139] Jason F. Cantin. Cache Performance for SPEC CPU2000 Benchmarks. Technical report, Department of Electrical and Computer Engineering, University of Wisconsin-Madison, May 2003.

[140] Jun Yang and Rajiv Gupta. Energy Efficient Frequent Value Data Cache Design. In *International Symposium on Microarchitecture (MICRO-35)*, pages 197–207, 2002.

[141] Raffi Krikorian. *Multithreading with C#*. O'Reilly, 2001.

[142] Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, 2 edition, 1998.

[143] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin and Turnbull. *The Real-Time Specification for Java*. Sun MicroSystem, 1 edition, 2000.

[144] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison Wesley, 3 edition, 2001.

[145] Sun's Java Community Process Real-Time Expert Group. Proposed Java real time API, v0.2. Technical report, Java Software Division of Sun Microsystem, December 1998.

[146] Sadaf Mumtaz and Naveed Ahmad. Architecture of Kaffe. Technical report, Kaffe.org, 2002.

[147] Doug Burger, Todd M Austin and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin, 1996.

[148] S. Öder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, May 1998.

[149] S. Pees and A. Hoffmann and V. Živojnović and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *ACM/IEEE Design Automation Conference (DAC)*, pages 933–938, 1999.

[150] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *The 11^{t}h International Symposium on System Synthesis (ISSS)*, December 1998.

[151] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *The European Conference on Design Automation and Test (DATE)*, March 1999.

[152] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, David I. August. Microarchitectural Exploration with Liberty. In *The 35^{t}h IEEE/ACM International Symposium on Microarchitecture*, pages 271–282, November 2002.

[153] Matthew Postiff and David Greene and Charles Lefurgy and Dave Helder and Trevor Mudge. The MIRV SimpleScalar/PISA Compiler. Technical report, University of Michigan EECS Department, April 2000.

[154] Bradley D. LaRonde and Jay Carlson and Mike Klar. Linux VR Tools HOWTO. Technical report, http://www.linux-vr.org/tools.html.

[155] AJ Klein Osowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. Technical report, Laboratory for Advance Research in Computing Technology and Compilers, October 2002.

[156] DARPA Funds SDSC to Investigate Tera Multithreaded Computer Architecture. Technical report, San Diego Supercomputer Centre, University of San Diego, August 1997.

[157] Opportunities for Multithreaded Processors in Wireless, Portable Devices and Other Embedded Processor Application. Technical report, Eleven Engineering Inc., 2002.

[158] Brian Neal. On Simultaneous Multithreading. Technical report, Ace's Hardware, October 2000.

[159] Arvind, D. K. and Rangaswami, R. Asynchronous Multithreaded Processor Cores for System Level Integration. In *IP'99*, November 1999.

[160] Arvind, D. K., Hossell, J., Koppe, A. and Rangaswami, R. Java Compilation for Multithreaded Architectures. In *the 9th workshop on compilers for parallel computers*, June 2001.

[161] A Al-Mahdy, I Watson and G Wright. VLSI Architecture Using Lightweight Threads (VAULT). In *Workshop on hardware support for objects and microarchitectures in Java*, October 1999.

[162] Randy Angstorm. ClearSpeed unviels multithreaded array processor. Technical report, TechNewsWorld, October 2003.

[163] IXP1200 Network Processor Datasheet. Technical report, Intel Corporation.

[164] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito and Ernesto Su. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. Technical report, Intel Corporation, 2002.

[165] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface (Version 1.0). Technical report, http://www.openmp.org, October 1998.

[166] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface (Version 1.0). Technical report, http://www.openmp.org, October 1998.

Index