The Study of Trace Cache Memory on Superscalar DLX Processor

Apisake Hongwitayakorn

Thesis submitted for the degree of

Master of EngineeringScience

Department of Electrical and Electronic Engineering
Adelaide University
South Australia
5005

June, 2003

Contents

List of Figures

List of Tables

Abstract

Instruction-level parallelism (ILP) is a technique to increase processor performance through the simultaneous execution of multiple instructions. Superscalar processor architectures implement ILP by providing multiple execution units to process instructions in parallel. To achieve high performance, the execution units must be occupied by a continuous series of instructions. Hence, the front-end of the processor has to be expanded in order to supply a continuous stream of instructions for the execution units. Although instruction-cache memory has been successfully used to enhance the fetch mechanism of superscalar processors for years, it cannot perform well enough for contemporary processors because of the nature of the statically ordered instructions stored in the cache. Branch instructions are the major problem because of the two possible directions of the branch outcome. They break up the continuity of the static code into short run-length basic blocks. Therefore, a line of an instruction cache can contain instructions that might be abandoned if they follow a branch that will be taken.

Trace cache architecture has been developed to reduce the effect of the problem. It has a sophisticated logic unit to capture dynamic instruction traces, possibly including multiple basic blocks, and store them in a single line. Therefore, it is most likely able to supply a larger segment of useful instructions in one hit. Moreover, the trace cache was deliberately designed not to lengthen the processor pipeline. It has been shown that trace cache can outperform instruction caches in large-scale microprocessors, e.g. 16-instruction wide processors.

This research studies the effect of trace cache memory on smaller-scale microprocessors like the superscalar DLX model that can process only 2 instructions simultaneously. The study will investigate the performance of the experimental trace cache compared to the existing instruction cache and also investigate the trade-offs in varying trace cache size.

To my parents,

my wife, my family, my incoming child,

and everyone who believes in me.

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Signed: _ Date : 19 / 06 / 2003

(Apisake Hongwitayakorn)

Acknowledgements

First, I would like to thank Michael J. Liebelt, my supervisor, for his help and support in everything. Without his excellent guidance and patience, my work definitely cannot be achievable.

All staff in the department of Electrical and Electronic Engineering, I appreciate their help for all these years since the first day I've been here. Thank you for everything.

I also want to thank my colleagues at Silpakorn University and my students out there for faith and belief that I can do this. In addition, a big thank to AusAID and Thai Royal Government for the scholarship.

And, of course, I'd like to thank all my teachers who gave me good knowledge from a very first day at school.

I have a very long list of friends and relatives who I want to thank. If I write it down, it would dominate the thesis. So, I would like to thank everyone with all my heart.

Last but definitely not least, I want to thank my mom, dad, sister, and brothers for everything. Also, I want to thank my wife who always be there for me and, at the time I wrote this, she is about to deliver the best gift I've ever got. It is the first child of us. What a marvelous timing! So, I can celebrate two things at the same time.

Chapter 1

Introduction

1.1 Overview

The performance requirements of high performance computers are escalating tremendously in order to respond to the complexity of modern software applications. Much research has been conducted on techniques to improve the performance of microprocessors as they are deployed in almost every level of modern computers. The objective is to increase the number of instructions that can be executed per unit time. Researchers in the field of semiconductor technology propose to increase processor clock frequency, the reciprocal of time usage. Meanwhile, computer architects attempt to modify processor microarchitecture and improve compiler technology in order to execute multiple instructions simultaneously.

Instruction-level parallelism (ILP) is the dominant technique exploited in modern processor microarchitecture. Parallelism of incoming static sequential instructions is detected in order to execute multiple instructions concurrently. This technique can be implemented using both software and hardware approaches depending on the type of processor. VLIW (Very Long Instruction Word) and superscalar are two types of ILP processors [23], [29]. The former aggressively uses compiler techniques to obtain high levels of parallelism. Hardware techniques are used in the latter to capture incoming instructions and dynamically determine those that can be executed in parallel. Consequently, software applications can be run on superscalar processors without recompiling [12]. In this thesis, we focus on superscalar processors as they are the more common type and have been for many years.

1.2 Superscalar Architecture

The operation cycle of a superscalar processor begins with fetching instructions from a static program into the processor using the *instruction fetching mechanism* and decoding them at the decoder unit. After this stage, the decoded instructions will be dispatched and temporarily accumulated in an instruction buffer called the *window of execution*. These instructions are no longer constrained by static program order. Therefore, they are free to be executed in parallel and ready to be issued simultaneously into the appropriate functional units located in the *instruction execution mechanism* after their operands become available, subject to data dependence and resource constraints [14], [15], [25], [32]. Figure 1.1 shows the diagram of superscalar architecture organization.

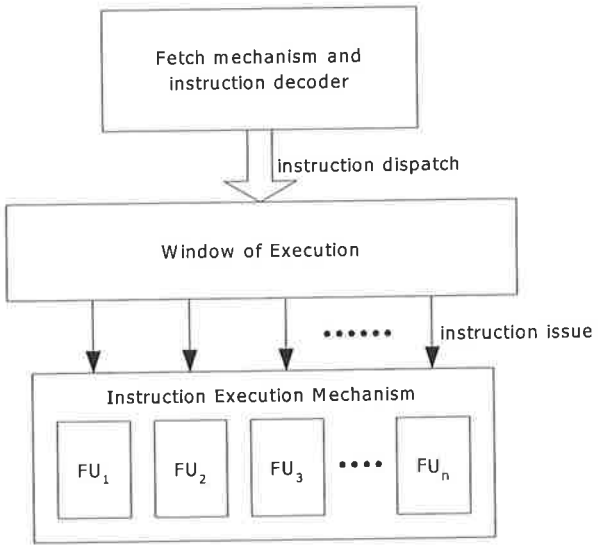


Figure 1.1 Organization of superscalar architecture.

To effectively exploit ILP is to improve superscalar processor performance by widening the window size for the purpose of increasing the possibility of finding data-independent instructions. More functional units are also required in order to be able to execute more instructions concurrently. Ideally, instruction-fetching bandwidth should correspond to the peak instruction dispatch and issue rate, to avoid the bottleneck problem [25]. However the constraint imposed by *control dependence* impedes the ability of the fetching mechanism to fetch instructions continuously, so it becomes important to overcome this constraint.

1.3 Trace Cache Memory

The fetch unit must be able to feed a continuous stream of instructions to the window of execution as quickly as possible. It would be much easier if instructions were all lined up in contiguous fashion from start to finish. Unfortunately, such behavior is not found in typical application programs because they possess branch instructions. Branch instructions, the causes of control dependence, are very common in typical programs [8] and cause the instruction fetching mechanism to wait for branch outcomes to determine whether the branches are taken or not taken. In [1], the term *basic block* has been defined as an instruction group, which has one entry point and one exit point. Whenever a branch instruction is encountered, that will be the end of the basic block. Typically, the average run-length of a basic block is about 4 to 6 instructions [24]. Therefore, the sequentiality of instruction addressing is disrupted and the program is divided into numerous small basic blocks.

The branch prediction method was introduced to lessen the problem of control dependence by speculatively predicting the outcome of branches. However, there is a problem of non-contiguous location of individual basic blocks inside the conventional instruction cache. Basically, there are useless instructions lurking between useful basic blocks that are scattered among different cache lines, so a single fetch might not be so effective. *Trace cache memory* [24] is proposed not only to overcome this crucial drawback which blocks the possibility of fetching multiple basic blocks concurrently, but also to diminish the latency of fetching, which is the flaw of related prior research on high bandwidth fetching mechanisms. Moreover, the trace cache was designed to work outside the main pipeline of the processor. Therefore, it does not introduce an additional pipeline stage that would increase processing time.

Trace cache research has been conducted for very high performance microprocessors, i.e. 16 instruction-wide superscalar processors. Rotenberg et al [24] showed that the fetching performance of a processor using a trace cache is improved by 34% for integer benchmarks and 16% for floating-point benchmarks. Meanwhile, the trace cache work on enhanced features conducted by Patel [21] showed that a trace cache can outperform an aggressive instruction cache scheme by 14% of overall performance and increase the fetch bandwidth by 34%. Recently, Intel Corporation adopted trace cache technology for the Intel NetBurst micro-architecture in its mainstream commercial processor, the Pentium-4 [11].

There has been no reported study of trace cache performance for a small-scale microprocessor. Therefore, this research will study the effect of the different trace cache memory configurations for a VHDL model of a superscalar DLX machine [10], which can process only 2 instructions simultaneously. The design of the trace cache of the experiment will be done for two main configurations, *TC_4* and *TC_8* for 4 instruction-wide and 8 instruction-wide trace cache respectively. Each configuration is studied with a varying number of trace cache lines to understand the trade-offs between performance and cache size.

1.4 Contribution of the Thesis

The contributions of this work can be summarized as follows:

- An analysis of the trade-offs between performance and trace cache size for narrow-issue sperscalar processor.
- An indication of whether trace caches are a worthwhile enhancement for narrow-issue superscalar DLX processor.
- A greater understanding of the performance characteristics of trace cache.

1.5 Outline of the Thesis

The thesis is organized into 6 chapters. Chapter 2 describes the background of trace cache design and related research work. Details of the DLX architecture and the superscalar DLX model that have been used in this research will be presented in Chapter 3.

Chapter 4 explains the experimental setup and methods. The results of the experiment and corresponding analysis will be in Chapter 5. This chapter also includes the discussion of the experiment. Chapter 6 will be the conclusion of the thesis.

Chapter 2

Background

2.1 Overview

Although the conventional instruction cache has served as a good source of instructions at high fetch rates for a long time, it cannot satisfy that high instruction consumption of wide issue processors. Instructions residing in the instruction cache are placed in compiled order and, unfortunately, typical programs possess many branch instructions. Consequently, several small basic blocks exist in run-time execution and disrupt the continuity of static instruction sequence in a wide instruction cache line. Even though the processors are designed to fetch several instructions in each line at the same time, many fetched instructions are abandoned. Therefore, fetching efficiency is low in this circumstance. To avoid this instruction-supply bottleneck, the trace cache was introduced to increase effective instruction fetch bandwidth.

In a superscalar architecture, the sequences of executed instruction from the pipeline are dynamic and divided into several basic-blocks by control instructions (e.g. branches, return, and etc.). These are called instruction traces. Several such instructions grouped together look like a VLIW instruction format but formed in dynamic sequence. The trace cache counts on two important properties of dynamic sequences of instructions, i.e. temporal locality and branch behavior [24]. That is, the most recently used instructions are most likely to be reused in the near future and branches mostly bias to one direction. If these dynamic traces are collected in a special kind of cache memory, the performance of the fetch mechanism will possibly be increased. There will be no need to fetch several times from different lines of the

instruction cache to obtain an instruction sequence possibly spanning several non-contiguous basic blocks.

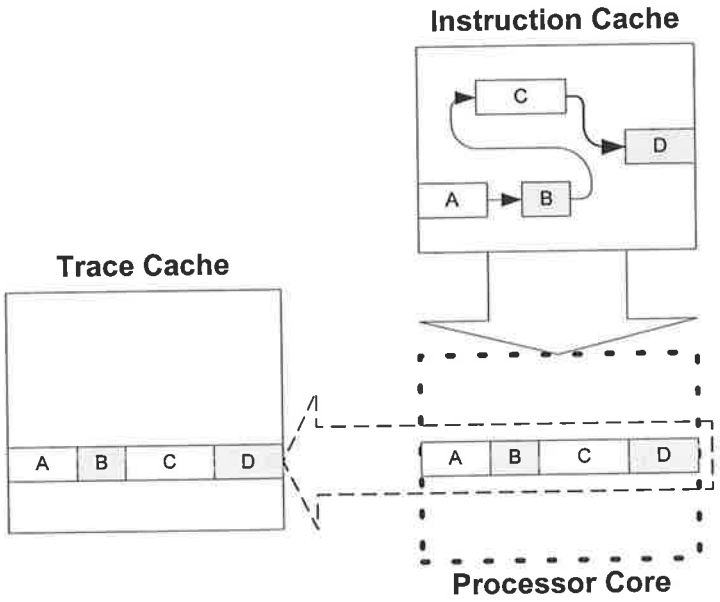


Figure 2.1 Trace cache overview

Figure 2.1 demonstrates the principle of the trace cache scheme. There are four basic blocks (A, B, C, and D) residing in non-contiguous locations in the instruction cache. They are logically connected together in run-time manner. Unfortunately, they are split in physical location due to static-compiled order; this is called *"partial fetch"* since each fetch could obtain just some part of all of the desired instructions. Time is wasted reading these instructions, as 3 cache reads are required (in this example). When these basic blocks are issued through the pipeline of the processor core they are rearranged in dynamic sequence or trace order (A, B, C, and D) to perform the task. This trace can be collected in the trace cache line. According to temporal locality and branch behavior as mentioned earlier, this trace is most likely to be used again in exactly the same sequence corresponding to the matching of fetch address and multiple predicted branches. Then, all instructions in this trace can be read in one fetch from the trace cache to the pipeline. This scheme obviously has the potential to increase fetching efficiency.

2.2 Trace Cache Architecture

The trace cache architecture is composed of four main components:

1. the trace cache (trace container),
2. the fill unit,
3. the branch predictor, and
4. the instruction cache.

As shown in figure 2.2, instructions can be read from the instruction cache or the trace cache depending on the outcome of the *hit logic* which processes the incoming fetch address and the outcomes of the branch prediction unit. If it signals *hit* the trace cache will deliver instructions. Otherwise, instructions are supplied from the instruction cache. Instructions residing in the trace cache are collected by the fill unit, which copies instruction traces entering the processor execution pipeline.

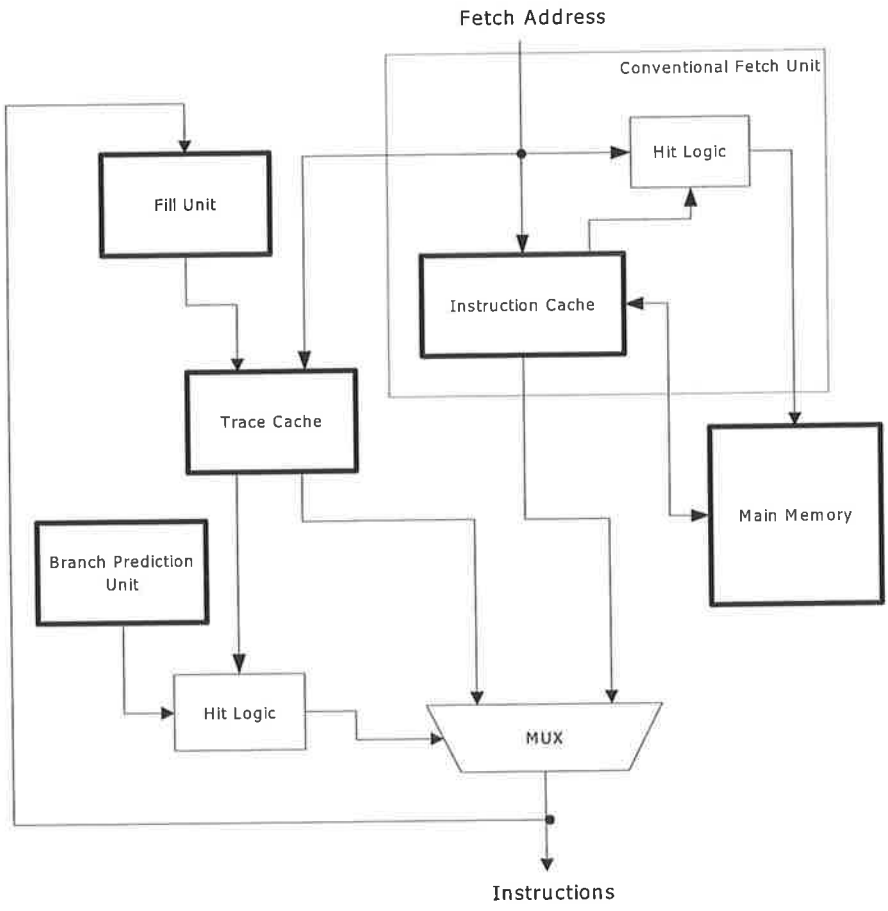


Figure 2.2 Trace Cache Architecture Diagram

2.2.1 The Trace Cache

The trace cache container is an array of fast-access memory, which dominates the area of the trace cache circuit. It collects several lines of trace issued from the fill unit. To each individual line of the trace cache is attached information similar to that in an ordinary instruction cache i.e. a *valid bit* to indicate availability of data in the line and a *tag* to identify the starting address of the trace. Moreover, there are some extended fields related to branch addresses because there might be more than one basic block inside the trace. All of this information is processed by the trace cache *hit logic* to determine whether an instruction fetch results in a trace cache hit or miss.

2.2.2 The Fill Unit

The fill unit is an essential component of the trace cache organization as all of the instructions accommodated in the trace cache come from this section. It gathers dynamic instruction sequences from the processor pipeline, merges the incoming instructions with existing instructions to form a packet, provides the attached information for each trace cache line as described above, and sends the packet to a line of the trace cache container. The essential step in the formation of a trace packet is packet finalization. The maximum number of instructions n and the number of predicted branches m are the main trace-packet delimiters. Both Patel [21] and Rotenberg et al. [24] have built models which carry 16 instructions (n=16) with a maximum of three branch predictions (m=3). Then, four conditions for finalizing the trace-packet are:

1. the packet contains 16 instructions, or
2. the packet contains 3 conditional branches, or
3. the packet contains a single indirect jump, return, or trap instruction, or
4. incoming instructions could not be concatenated with the existing instructions since the sum would exceed 16 instructions.

2.2.3 The Branch Predictor

The performance of any fetch mechanism relies on the precision of the branch predictor because an incorrect branch prediction causes a time penalty due to instruction recovery. In the case of a wide issue processor, a single branch predictor seems to be inadequate because a line of trace cache is likely to contain multiple basic

blocks, as mentioned earlier. Therefore, a trace in the trace cache would be more effective if the predictor can cover all of the branch instructions in a line and if the outcome of the prediction is sufficiently accurate. Otherwise, the penalty would be more severe and waste more time.

Unfortunately, at a present, the technology of multiple branch predictors is still immature and the accuracy is less than that of single branch predictors. However, the scheme known as two-level branch prediction [34] showed impressive prediction accuracy at 97%. This method can be implemented within the trace cache scheme to predict three branch outcomes in a single cycle.

2.2.4 The Instruction Cache

Even though the trace cache plays an important role supplying instructions for the processor, the conventional instruction cache is still needed. When the *hit logic* signals a trace cache miss, the instruction cache has to provide the requested instructions, instead. Moreover, the instruction cache, itself, is the instruction gateway connected between main memory and the processor. However, the size of the instruction cache might be trimmed down to suit such less frequent activities.

2.3 Related Work

There is a large amount of published research, using both hardware- and software-based approaches, on high bandwidth fetch mechanisms. Some hardware-based approaches are listed here for the purpose of tracing back the history of the trace cache. Some of these are currently adopted in parts of the trace cache scheme. The others are significant competitors of the trace cache approach.

2.3.1 Trace Cache History

The history of trace cache development begins with the fill-unit, which was introduced as hardware proposed to increase the front-end performance of the VAX architecture. Melvin et al [16], showed that the parallelism of such a sophisticated instruction set architecture can be exploited by using a fill unit to create large execution atomic units (EAUs) dynamically. Hypothetically, the larger EAUs contain more microoperations able to be executed simultaneously. Each EAU is stored in the

decoded instruction cache to be reused by the execution unit. In subsequent work [17], Melvin and Patt varied the size of EAUs of the dynamically scheduled machines using a fill-unit unit to gather two or more instruction basic blocks in the associated cache. The results showed that larger EAUs effectively enhance the performance of the processor because of the higher utilization of processor pipeline slots.

In 1994, Franklin and Smotherman [6] adopted the fill-unit for their multiple instruction issue architecture. The fill-unit dynamically packs multiple instructions into VLIW-type instructions and stores them in the *shadow cache*. When the instructions in a shadow cache line are required, they can be issued and executed simultaneously. The proposed fill-unit also includes logic for checking data dependencies of stored instructions as well as a unit for dealing with delayed branches. There is also a branch predictor to assist the fetching mechanism with speculative execution in order to create effective cache lines.

In 1994, Peleg and Weiser [22] patented their new instruction cache design, which is similar to the trace cache, namely the *Dynamic Flow Instruction Cache*. This scheme enhances the fetching mechanism for superscalar machines by storing 2 instruction basic blocks in a cache line. The branch instruction at the end of the first basic block has been predicted and the outcome of the prediction is the physical address of the first instruction of the following basic block of the cache line. Instructions in the cache are collected dynamically from the instruction flow and all instructions in a cache line can be fetched in a single access. The difference between this cache scheme and the current trace cache is that in the former each basic block is used as a starting point for each trace packet created.

The other trace cache lookalike is the *Expanded Parallel Instruction Cache (EPIC)* proposed by Johnson in 1994 [13]. This architecture has been designed to enhance in-order superscalar machines by reducing the complexity of the instruction decoding and issuing mechanism. Each line of the *Expansion Cache* contains decoded and dependency analyzed instructions, which were routed to certain execution units. Therefore, it can reduce the processing time once the instructions are fetched. The performance of this design is approximately equal to one of the more complex out-of-order superscalar machines with traditional instruction cache.

Rotenberg et al. [24] designed the trace cache scheme consisting of a small cache with a large instruction cache embedded in a 16-wide issue superscalar

cache design, fill unit design, and in particular, multiple branch prediction. They showed that a large trace cache assisted by a small instruction cache outperforms alternative configurations [24]. Therefore, the instruction cache can be designed less aggressively as it is subject to fewer instruction accesses. Patel et al. continued their work to improve the performance of the trace cache as reported in [18], [19], and [20]. They explored several enhancements to the trace cache model in order to overcome performance limitations. Recently, Patel assembled all of his previous works and some new features of the trace cache into his Ph.D. dissertation [21]. He describes and evaluates the basic trace cache fetch mechanism, which outperforms an aggressive instruction cache. High performance was achieved through the use of several enhancements including:

- Partial Matching – the ability to pick up the useful blocks in a matching trace line and to discard the rest instead of wasting the whole trace due to branch prediction mismatch.
- Inactive Issue – instead of totally discarding useless blocks because of branch prediction mismatch as in Partial Matching, Inactive Issue allows the whole trace to be fetched and marks these mismatch blocks as inactive blocks. There is no effect on fetching performance if branch prediction was correct. Otherwise, the inactive blocks would offer useful instructions to be executed.
- Branch Promotion – in order to reduce the bandwidth of the branch predictor and increase the effectiveness of the fetch mechanism, Branch Promotion embeds the statically predicted information (taken/not taken) to strongly bias branch instructions [25].
- Trace Packing – this enhancement sacrifices trace cache area in order to increase individual fetching capability within the loop as shown in figure 2.4. In case of a 16-instruction trace, segment AB already occupied 11 slots and left 5 slots for the next segment. Unfortunately, segment C has 6 instructions and can not fit in. Therefore, the possible traces would be AB, CA, and BC. Using Trace Packing will store 6 combinations for the dynamically unrolled loop as follows: $A_6B_5C_5$, $C_1A_6B_5$, $C_6A_6B_4$, $B_1C_6A_6$, $B_5C_6A_5$, and $A_1B_5C_6$. The subscripts denote the number of instructions in each particular segment.

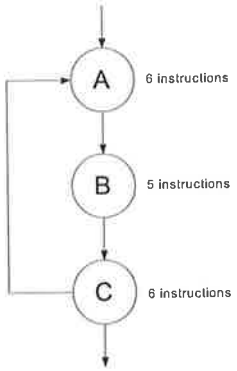


Figure 2.4: A loop contains 3 segments.

The aggregation of Partial Matching, Inactive Issue, Branch Promotion, and Trace Packing, make the trace cache outperform the state-of-the-art *Sequential-Block* instruction cache scheme both in processor performance (IPC metric) and in average fetch rate. Furthermore, Patel's analysis showed that as fetch rate increases, branch resolution time increases. Lastly, a next-generation processor implementation is described which achieves high fetch rates at high branch prediction accuracy. Figure 2.5 shows this trace cache fetch mechanism.

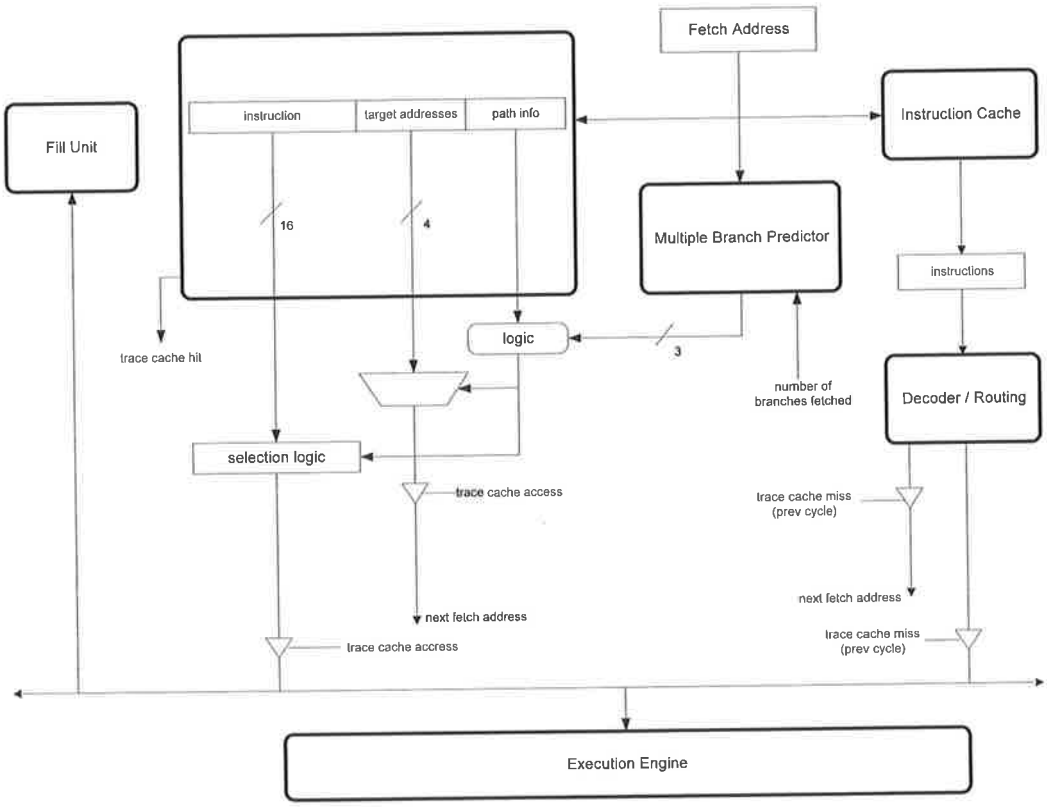


Figure 2.5: The trace cache fetch mechanism [19]

Comparing figures 2.3 and 2.5, even though they are both based on the trace cache fetch mechanism, there are some differences between them affecting overall performance. The former model delivers dynamic instruction streams that have been captured before they are sent to the decoder. On the other hand, in the latter model decoded instructions are sent to the fill unit before being dispatched to the execution engine. Therefore, when a trace cache *hit* is signaled, instructions go directly through the execution engine without passing to the decoder/routing again. Furthermore, these instructions are already analyzed for dependencies and pre-routed to appropriate execution units. The other difference between the models is the information contained in each trace cache line. The latter model includes not only the branch target address for checking trace cache hit/miss, but also path information which facilitates the path enhancement of the model i.e. Partial Matching and Inactive Issue.

2.3.2 Other High Bandwidth Fetch Mechanisms

The *Branch Address Cache* [34] and *Collapsing Buffer* [4] have been previously mentioned as multiple basic block fetch mechanisms. They achieve high effective fetch rate, although they cannot perform as well as a trace cache. However, it is worthwhile to examine them to see why this is so.

In 1993, Yeh et al. proposed the branch address cache scheme [34] shown in figure 2.6. It generates multiple fetch addresses in a single cycle resulting from the branch address cache working together with the branch predictor. These addresses will be calculated as indices to point to the exact location of each basic block residing in the interleaved instruction cache. Finally, all targeted instructions are passed through the alignment and masking network in order to form a packet ready for issue. The problems of this scheme are hardware complexity and its lack of amenability to aggressive branch prediction.

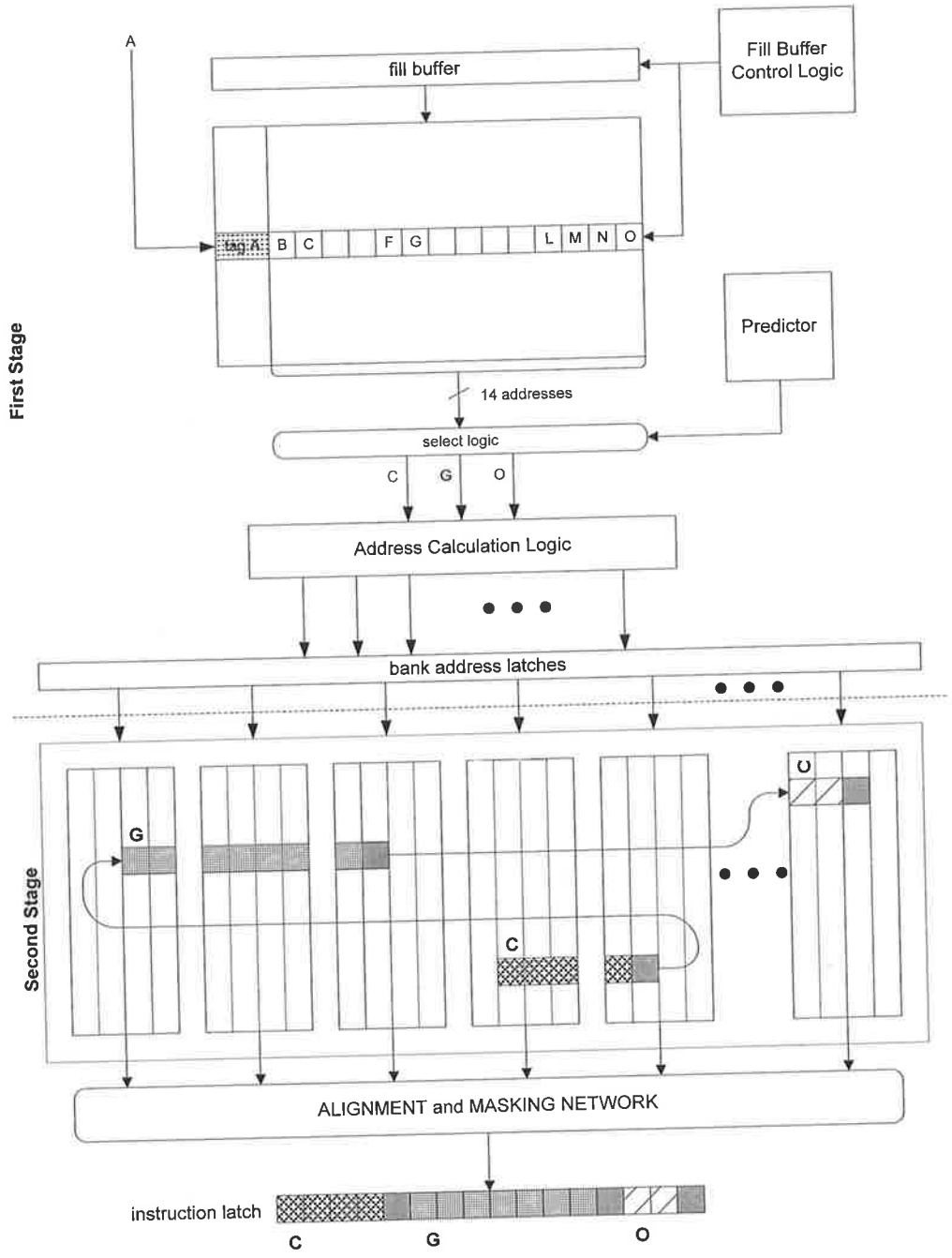


Figure 2.6: The Branch Address Cache.

Conte et al. proposed the Collapsing Buffer [4] as shown in figure 2.7. Two nonadjacent cache lines can be fetched together since the scheme uses two passes through an interleaved branch target buffer. Each pass through the branch target buffer produces a fetch address. Moreover, the BTB can detect any number of branches in a cache line. Therefore, it can detect intrablock branches and eliminate the

unused instructions by using the collapsing buffer in the interchange/masking network. Likewise, this approach adds more process stages to the fetching pipeline and this decreases overall performance.

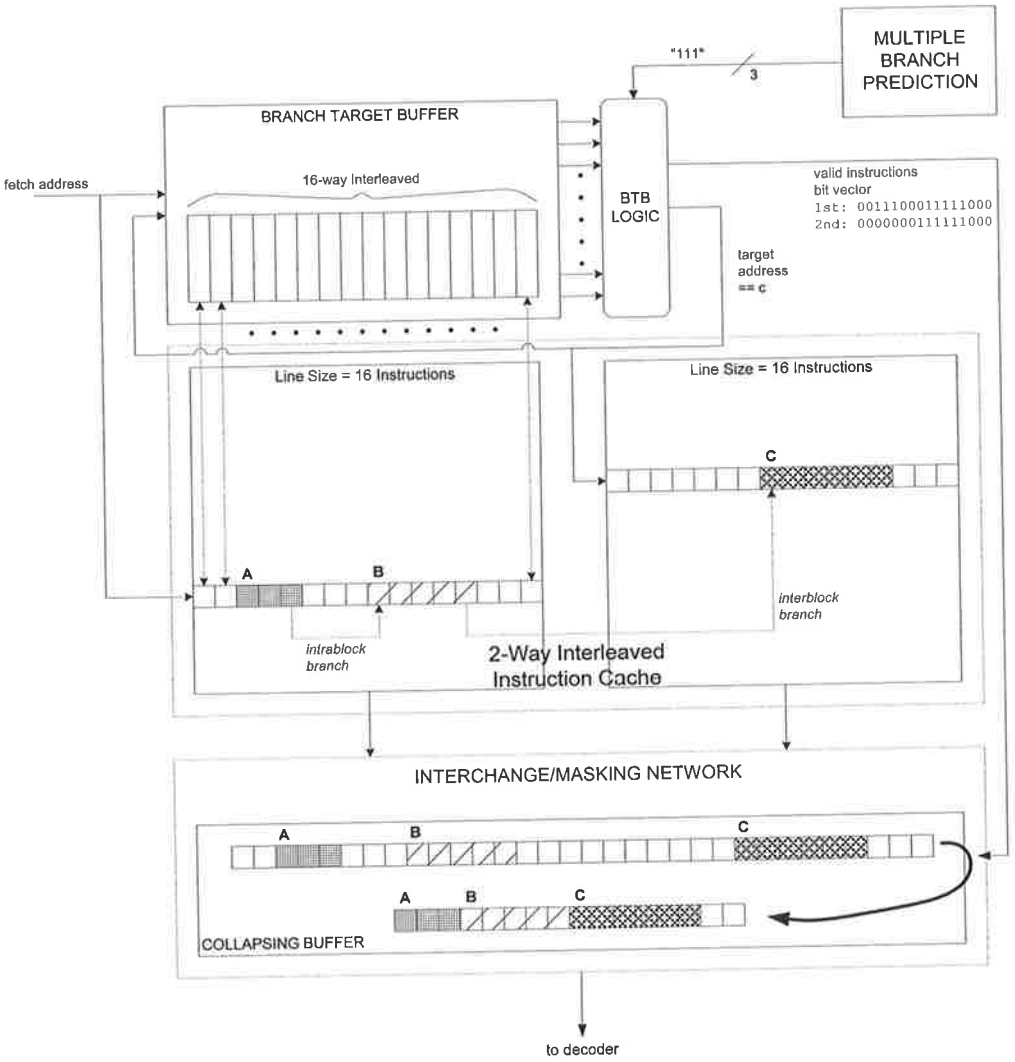


Figure 2.7: Collapsing Buffer

2.4 Conclusion

In summary, the trace cache mechanism can perform better than other aggressive approaches in respect of fetching ability but it needs sophisticated logic to create effective traces and a substantial memory area. Therefore, a trace cache might not be cost-effective for general-purpose processors at the present. However, the previous trace cache studies have been conducted only on wide-issue processors.

Hence, this research focuses on the effectiveness of trace cache on narrow-issue processors. The objective is to find out the significance and trade-offs of TC parameters that affect the performance of the cache scheme and the usage of cache space for the consideration in TC implementation on narrow-issue processors.

Chapter 3
Experimental Processor Model

3.1 Overview

The trace cache experiments in this thesis are based on simulation. A superscalar implementation of the DLX architecture has been chosen as the experimental processor model. The VHDL language is used to describe the simulation model, since the language facilitates both model construction and testbench simulation. In addition, the working model could be used as a foundation to synthesize the processor using suitable VHDL synthesis tools. Fortunately, there is a superscalar DLX processor model [10] in VHDL that is suitable for the proposed experimentation.

3.2 DLX Architecture Summary

The DLX architecture was first introduced by Hennessy and Patterson [9]. It possesses features, which can be commonly found in several successful processors based on the RISC philosophy.

The significant features of the DLX architecture are

- an uncomplicated load/store instruction set,
- pipelining effectiveness,
- an easily decoded fixed-length instruction set, and
- efficient machine code, as targeted from high-level program compilation.

3.2.1 DLX Registers

There are three register types in the DLX architecture. Firstly, the general-purpose registers (GPRs) comprise thirty-two 32-bit registers named R0, R1, …, R31. The value of R0 is permanently set to zero. The GPRs are used for all integer operations and memory addressing modes. Secondly, the floating-point registers (FPRs) comprise thirty-two 32 bit single-precision floating point registers named F0, F1, …, F31. They can be used as double-precision floating point registers (64-bit) by coupling odd and even registers into a register pair (F0, F2, …, F30). These registers are used only for floating-point operations. Lastly, the special-purpose registers comprise several registers for purposes such as masks and flags.

3.2.2 DLX Data Types

There are 8-bit (byte), 16-bit (half word), and 32-bit (word) integer data plus 32-bit single precision and 64-bit double precision floating point data type. They conform to Big Endian byte ordering as illustrated in figure 3.1.

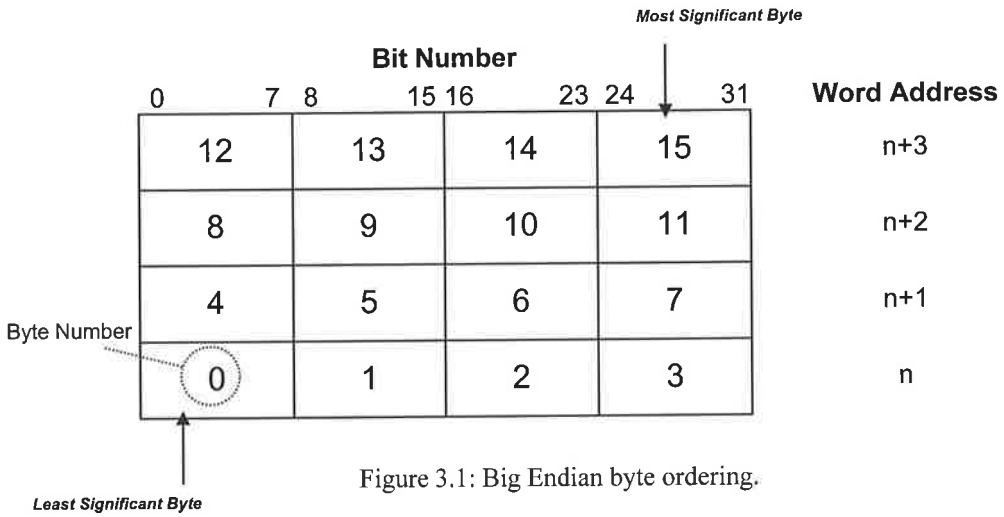


Figure 3.1: Big Endian byte ordering.

3.2.3 DLX Addressing Modes

The explicitly supported data-addressing modes in the DLX are immediate and displacement, using 16-bit fields as immediate data and displacement address fields, respectively. However, putting 0 in the 16-bit displacement field can accomplish the register-deferred mode and using register R0 as a base register associated with 16-bit field can accomplish absolute addressing. Therefore, there are four effective addressing modes available in the DLX.

3.2.4 DLX Instruction Types

There are three different instruction types: I-type (immediate), R-type (register), and J-type (jump). All instructions are 32-bit format as shown in figure 3.2.

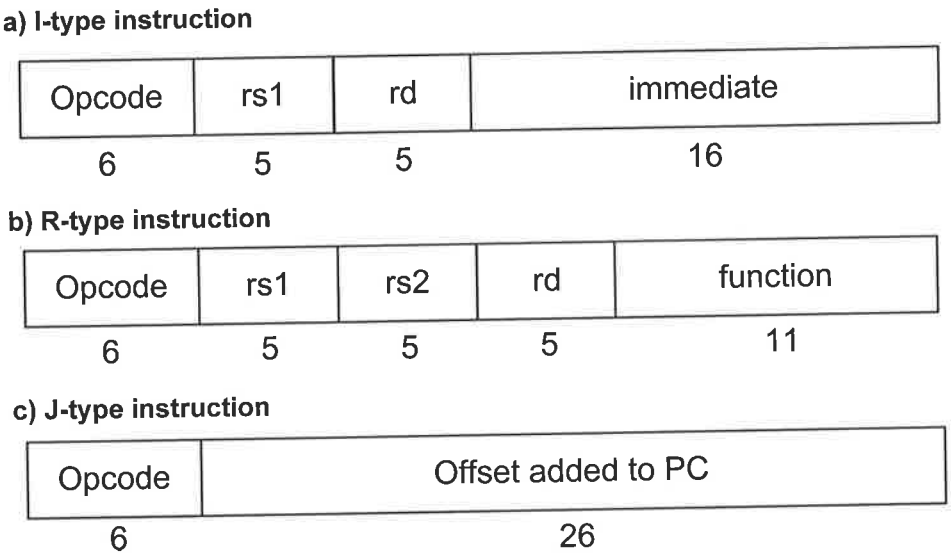


Figure 3.2 DLX instruction format.

Since all instructions are of fixed-length format, instruction decoding is very simple.

DLX is an easy architecture to understand and, moreover, widely studied and modeled. Consequently, it is a useful processor on which to base the study of the trace cache.

3.3 The Superscalar DLX Model

The superscalar DLX model used in this research was created by Horch in the VHDL language [10]. Both the source-code and documentation are provided at URL *http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.htm*. Although the documents were written in German, the source-code is commented in English and is quite simple to follow. Figure 3.3 shows the structure of the superscalar DLX processor.

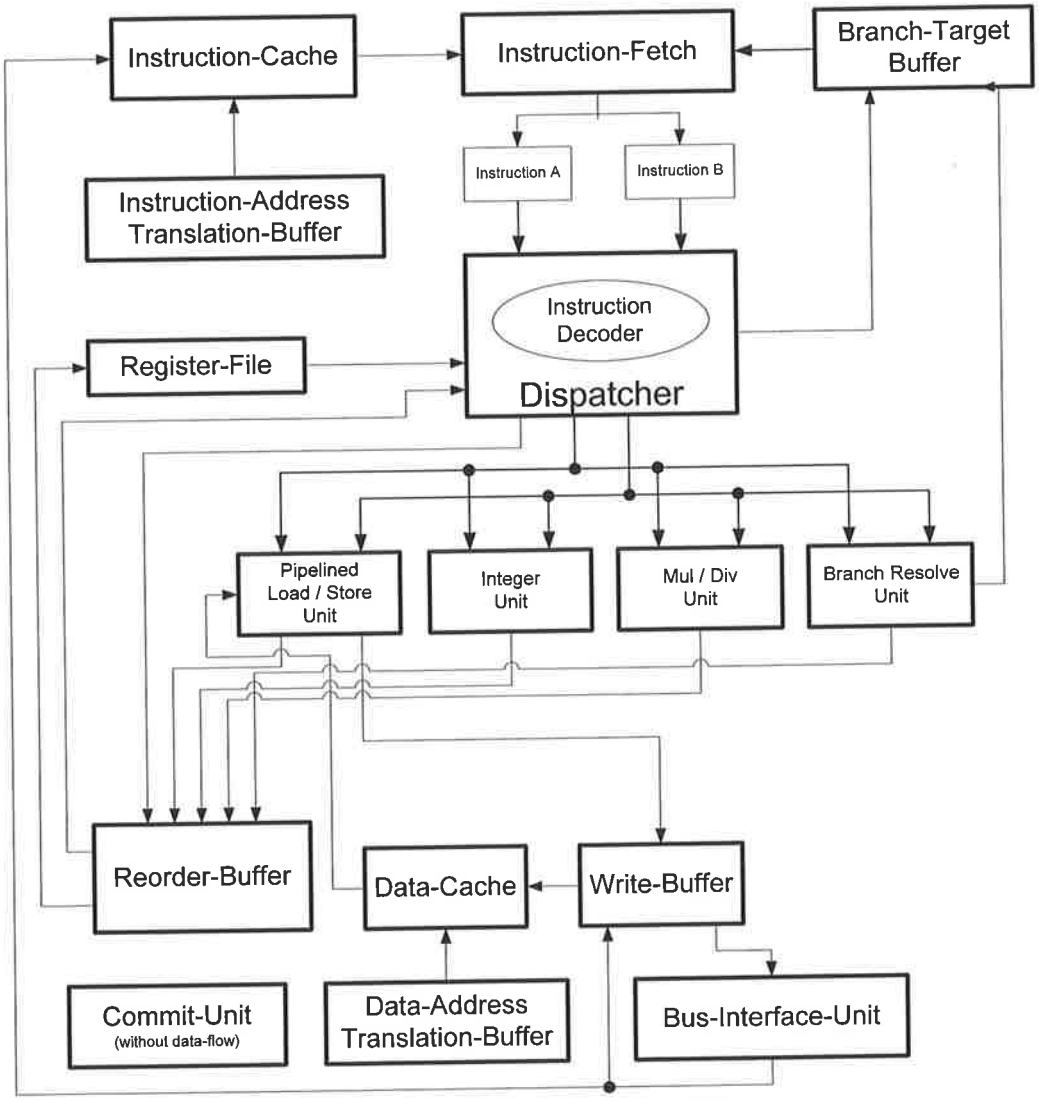


Figure 3.3: Superscalar DLX structure. [10]

The microarchitecture of this model is a pipelined superscalar processor. It can fetch a maximum of two instructions simultaneously in a single cycle. The instruction fetch unit is supported by a 64-byte instruction cache coupled with a 4 entry instruction address translation buffer (ITB). There is a branch target buffer (BTB) to provide the speculative target of branch instructions.

The dispatcher is the heart of the processor since it connects to every major unit of the model. Accordingly, it generates control signals to manipulate all processor activity from instruction entry until instruction commit. Moreover, the dispatcher also manages precise exception processing. This is assisted by the reorder-buffer, which works with the commit unit to commit instructions in program order.

There are four execution units, each with a reservation station: pipelined load-store unit, integer unit (arithmetic logic unit or ALU), multiply-divide unit (MDU),

and branch resolve unit. The load-store unit works cooperatively with the write buffer and 64-byte data cache equipped with 4-entry data address translation buffer (DTB). The ALU executes all logical, shift, and set-on-comparison instructions. Moreover, it mainly does the integer arithmetic calculation for addition and subtraction.

Integer multiplication and division can be performed by the MDU but the implementation of MDU is slightly different from the original DLX architecture. In the original architecture, multiply and divide instructions can be performed only with floating-point registers (F0-F31). Therefore, data type conversion instructions from integer to floating-point and vice versa (i.e. MOVI2FP and MOVFP2I) are available to enable integer multiplication and division using the floating-point multiply/divide unit. To avoid any implementation of floating-point operations, Horch defined a unique register file that can be addressed as GPRs (R0-R31) or FPRs (F0-F31). R0 and F0 are the same physical register and so on. Consequently, multiply and divide instructions (MULT, MULTU, DIV, and DIVU) perform integer multiplication and division on the GPRs. This variation from the standard architecture required some code modification, which will be described in chapter 4.

Lastly, the branch resolve unit determines actual branch outcomes, determines the target address to insert in the BTB and also indicates when a branch misprediction has occurred.

3.4 The Fetch Unit

The fetch unit is the part that is of most interest in this research, since the trace cache is intended to improve the fetching performance beyond the conventional instruction cache. So, the original fetch unit will be described in detail, to provide information on the original model design.

The fetch unit has been designed to fetch a maximum of two instructions from the instruction cache in a single cycle if the address of the first instruction in the program counter is double word aligned. Word order within double word is Big Endian (i.e. 0x00000000 is the high word and 0x00000004 is the low word). The registers for storing the fetched instructions are divided into the stage A register and the stage B register. Both of them can store either high word or low word. Normally, stage A stores the high word and stage B stores the low word. However, stage A can

store the low word in which case stage B will become invalid. In the case of fetching two instructions when the address is double word aligned, but when stage A is not available, stage B can store the low word and the program counter will be increased by 4 bytes.

As mentioned above, there are two main units associated with the fetch unit. They are the instruction cache and the branch-target-buffer (BTB). The instruction cache in the original model has a small capacity and is configured as a direct mapped cache. It has 8 lines containing two instructions each. So, it can contain only 16 instructions (16*4 = 64 bytes) at a time. The availability of instructions in each cache

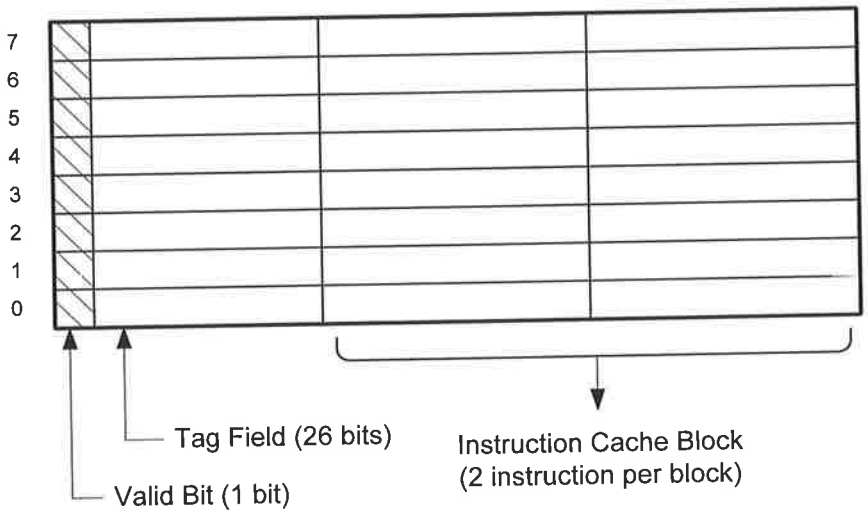


Figure 3.4: Instruction cache structure.

block is indicated by the *valid bit* and the *tag field used* for address matching.

The instruction cache cooperates with the instruction-address-translation buffer (ITB) to convert a virtual page number (bits 31 to 7 of the program counter) into a physical page number and this is joined with bit 6 of the program counter. The results are used to compare with the tags of instruction blocks to determine cache hit or miss. The ITB has a 128-byte page size. Figure 3.5 shows the address translation mechanism of the instruction cache and instruction-address-translation buffer. This configuration is also used for accessing the data cache in this model.

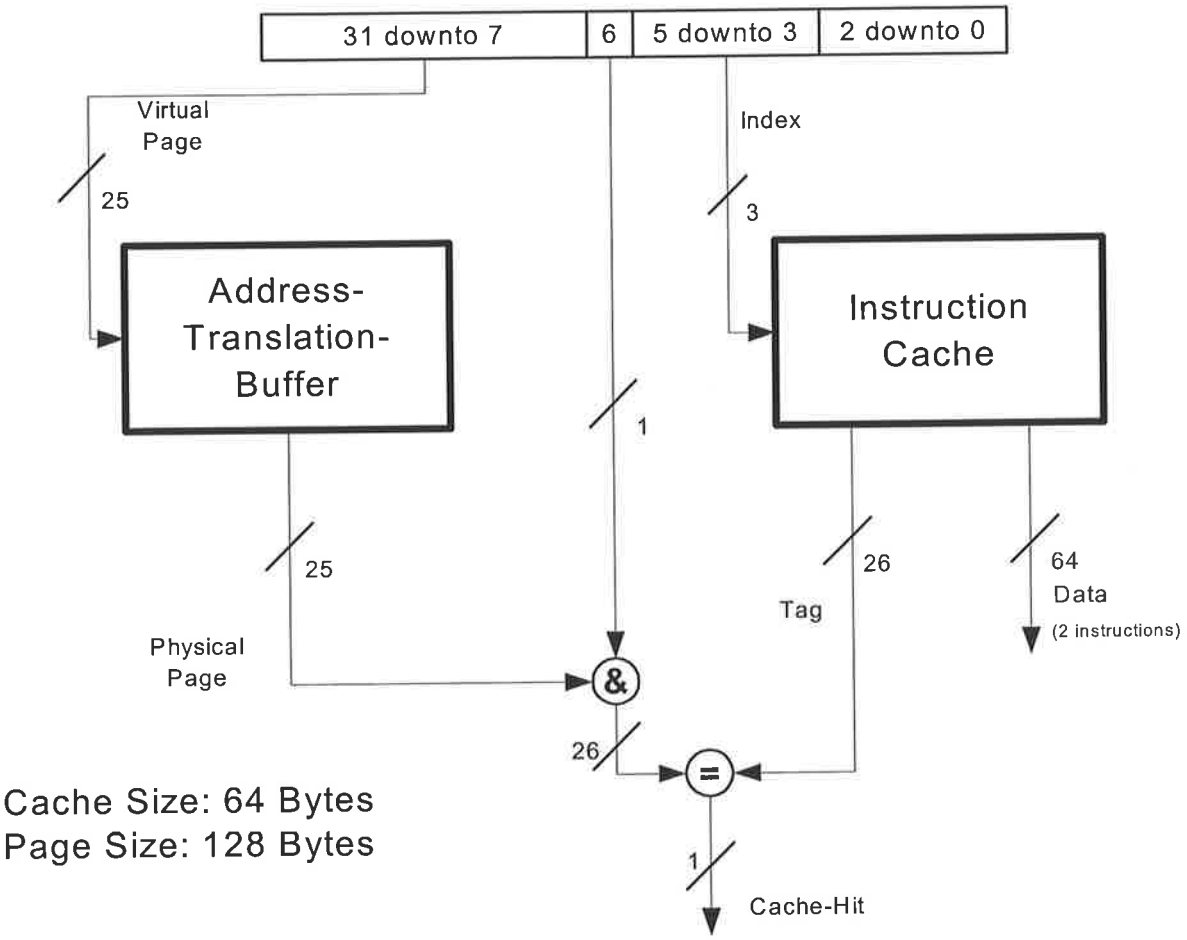


Figure 3.5: Address-translation and cache-access. [10]

The branch-target-buffer (BTB) is a memory that contains destination addresses of previously executed branch instructions. These addresses are most likely to be the target of future branches. When one of these branches is fetched again, the BTB will speculate the direction of the next instructions without waiting for the outcome of branch condition determination. In this model, there are four slots within the BTB to store destination addresses. Like the instruction cache, each entry composes of a *valid bit* for indicating the availability of the BTB data and a *tag field* for address matching.

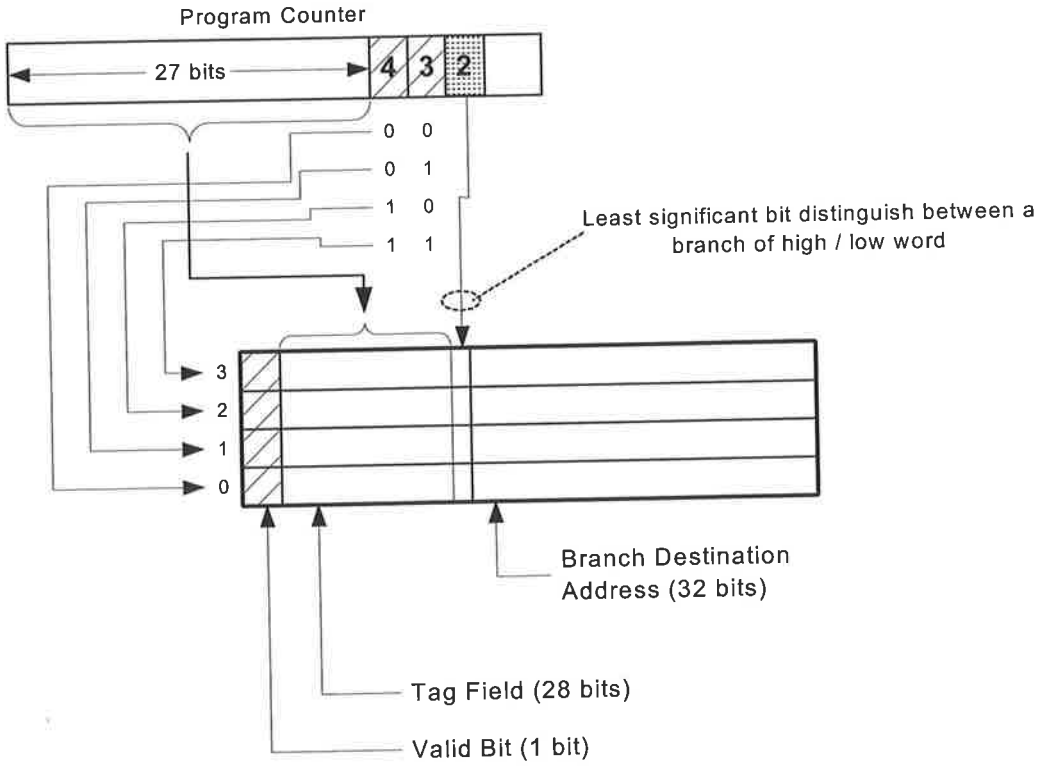


Figure 3.6: Branch-target-buffer structure.

Indexing to the BTB slot uses bits 4 and 3 of program counter (2 bits = 4 combinations). Then, the last three bits make all entries represent 8-byte aligned addresses. Consequently, the destination address stored in each slot has to indicate whether the branch is a high-word or low-word instruction. This is accomplished by attaching the extra bit as the least significant bit of the 28-bit tag portion. The extra bit comes from bit 2 of the program counter.

3.5 Conclusion

The superscalar DLX model [10] is a narrow-issue processor model, which was written in VHDL format. It can execute integer programs including integer multiply and divide instructions without conversion between floating-point and integer data type. The implementation details of a trace cache on this processor model are described in the next chapter.

Chapter 4

Experimental Setup

This research was conducted to find out whether and how a trace cache memory can help a narrow issue superscalar processor to fetch instructions. As mentioned in chapter 3, an existing superscalar DLX processor model [10] was chosen to avoid spending the time required to build the processor from a scratch and give more time to focus on the trace cache model which is the target of this research. This chapter contains the explanation of an experimental setup that was used to implement the trace cache and to gain results for experimental analysis.

4.1 Trace Cache in the Superscalar DLX Processor

The trace cache is a source of instructions containing dynamic traces of instructions instead of static ones as an ordinary instruction cache does. Therefore, the trace cache is supposed to be an alternative repository, to compete with the embedded instruction cache to supply instructions to the execution unit.

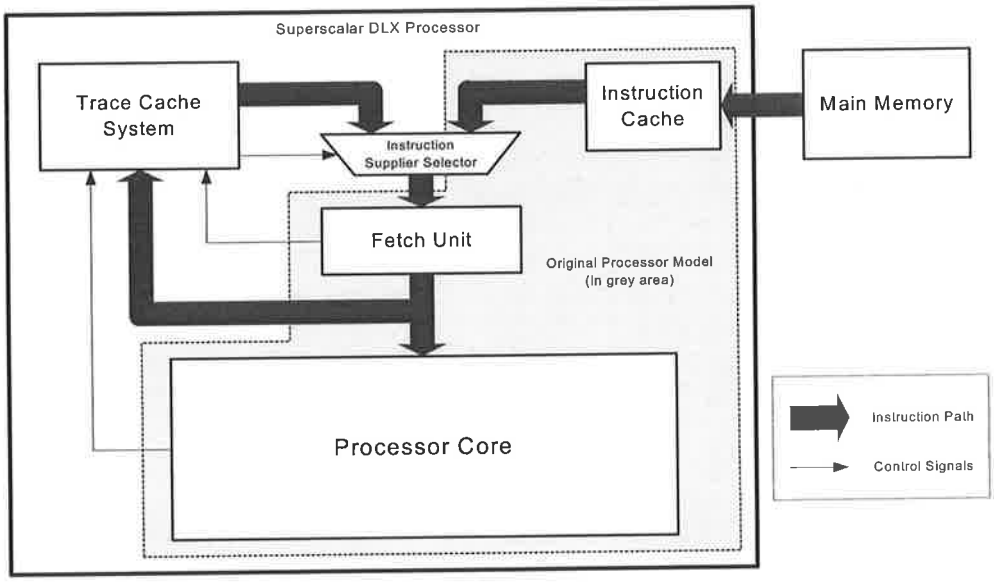


Figure 4.1: Trace cache placement in the superscalar DLX machine.

Figure 4.1 shows the placement of a trace cache in a way that would fit in with the original superscalar DLX processor model. The task of the trace cache is to collect instructions from the fetch unit and pack these into traces with help from control signals of the fetch unit itself and other units in the processor core, which determine how to pack them. The trace cache can feed these traces back to the fetch unit. However, the existing instruction cache is still the main supplier but also is the competitor of the trace cache. Therefore, the trace cache system must be equipped with trace cache 'hit' logic to make a decision on which instruction supplier would do the job.

It is possible to build a trace cache by gathering instructions from the main processor pipeline and producing instruction traces. However, incorporation of a trace cache would add a lot of complexity due to the original processor model, which was not designed for this kind of expansion. Hence, the other way to accomplish this mission is to leave the original superscalar DLX model untouched and examine the utilization of the trace cache passively. In the other words, we build the whole trace cache mechanism and investigate whether instructions collected in the trace cache match the current fetched address in the program counter register of the processor.

This allows us to determine the effectiveness of the trace cache by measuring trace cache hit rate, but we do not provide instructions from the trace cache to the processor core since we are not attempting to measure the overall performance

increase due to the trace cache. In other words, we will assess trace cache performance in terms of trace cache hit rate and not be concerned with processor performance improvement due to the higher fetch rate produced by the trace cache. The latter is highly dependent on implementation technology.

4.2 Trace Cache Line Size

In prior trace cache models [21,24], the superscalar processor models have wide instruction-paths, which are significantly different from the modest 2-instruction-wide superscalar DLX machine we use in this research. In this circumstance, the trace cache model used in this experiment is likely to be quite different from them.

On a 16-instruction-wide superscalar processor, the individual trace cache line is designed to fit 16 instructions and is able to feed a maximum of 16 instructions simultaneously to the fetch unit. This approach would not fit with the 2-instruction-wide processor (i.e. making the trace cache to accommodate just only 2 instructions in a cache line) because the average basic block for typical applications is about 4 to 6 instructions [24]. More importantly, the significant idea of the trace cache system is to provide a trace that covers the basic block and to overcome the penalty of branch misprediction. In this circumstance, we will provide a trace cache line size of 4-instruction-wide or 8-instruction-wide to cover at least one basic block. The reason for making 2 versions of trace cache line size is to find out an appropriate configuration from the experiment results. In this project, we call the 4-instruction-wide model and 8-instruction-wide model *TC_4* and *TC_8*, respectively.

4.3 Trace Cache Model Components

There are 4 main parts that work in concert, starting from gathering instructions from the fetch unit until determining a trace cache 'hit' or 'miss'.

4.3.1 Instruction Gathering Unit

This unit works closely with the fetch unit. In the original processor model, the fetch unit provides two registers for holding fetched instructions. Both instructions

could be dispatched through the execution windows simultaneously or just one of them depending on the availability schedule of the required execution unit for each instruction. The instruction that was left behind will be fed in the next clock period in which the functional unit is available. To avoid double copying of the same instruction from different fetch cycles, the instruction-gathering unit must be able to determine how many instructions could be collected and which one of them should be collected in the case that only one instruction could be dispatched. The determination can be made by consulting a group of control signals. These signals are created by the dispatcher in which they are originally used for checking the validity of incoming instructions at the fetch stage. After the determination is accomplished, the individual valid instruction is ready to be placed into the appropriate position of the fill-buffer under the fill-policy for creating a dynamic instruction trace.

4.3.2 Fill-Buffer

This buffer is a temporary memory which stores valid incoming instructions as traces, before transferring them to the trace cache memory space. However, it is the most significant part of the mechanism because the usability of packed traces depends directly on the fill-policy that crafts the individual trace. Because of the different trace cache line size and different number of basic block coverage, the fill-policy will be different from the previous trace cache works in [21] and [24]. Details of fill-policy will be described in section *4.3.2.2 Fill-Logic and Fill-Policy*.

4.3.2.1 Fill-Buffer Configuration

The buffer comprises two main sections, *Trace Content* and *Trace Information*. *Trace Content* stores collected instructions and their addresses. Meanwhile, *Trace Information* stores information associated with the trace which is used during fill-buffer to trace cache transfers. *Trace Content* is constructed as 4 lines (line numbers 0-3) by the number of instructions (4 and 8 instruction slots for *TC_4* and *TC_8*, respectively.) Figure 4.2 shows the structure of the fill-buffer.

- **Branch Existing Flag:** This flag indicates whether or not there is a branch instruction in the buffer line.
- **Branch Position:** Together with the previous flag, this field pinpoints the location of the available branch instruction of the line.

The bit-length of the information fields *Trace Size* and *Branch Position* depend on trace cache line size. Each of them is 2 bits for *TC_4* and 3 bits for *TC_8*. Note that, as explained in the following section, each trace cache line will contain no more than one conditional branch (i.e. 2 basic blocks).

4.3.2.2 Fill-Logic and Fill-Policy

The most significant part of the whole fill-buffer is the fill-logic, which determines how to fill instructions into the buffer, because the usefulness of traces directly depends on the characteristic of the traces themselves. The implementation of the fill-logic is ruled by the fill-policy, which defines how to construct and when to terminate a trace from instructions collected from the instruction gathering unit.

The collected instructions will be put into available slots one after another in the current incomplete trace until the trace is terminated by one of the following conditions:

1. when the size of the current trace including one or both of the new incoming instructions equals the buffer line size, or
2. when either one of the new incoming instructions is an unconditional branch (jump), RFE, or trap, or
3. when the current trace already possesses a conditional branch and either one of the new incoming instructions is also a conditional branch. It was decided that there must be maximum of 2 basic blocks per line because general programs have basic block run-length about 4-6 instructions [24]. Hence, the narrow cache width like *TC_4* and *TC_8* will rarely be able to accommodate more than 2 basic blocks.

Accordingly, the fill-logic fundamentally composes of a set of pointers used to locate the current row and slot in the trace content space of the fill-buffer for each of the incoming instructions and their addresses. The most important task of this unit is to manipulate the pointers to implement the above rules correctly.

Rule 1 is the simplest way of terminating the line, when an incoming instruction makes the current trace reach the limit of the buffer line size. Note that there might be either one or two instructions collected from the fetch unit. In case there is only one instruction, if the length of the incomplete current line plus this instruction equals the buffer line size, the instruction will be placed into the line and also terminates this line. Meanwhile, the pointers will be updated to the beginning of the next line. However, if there are 2 incoming instructions, there could be two distinct cases.

- Case 1: The first incoming instruction of the two occupies the last slot of the current buffer line. Therefore, this line will be terminated and the other instruction will be placed in the beginning slot of the next line.
- Case 2: There are two slots left in the current line while there also are 2 incoming instructions. The logic can place both instructions into the slots, terminate the current line, and start the next line for the next incoming instructions.

If the trace was terminated by rule 1, it means that the line was fully occupied by instructions that are most likely coming from the same basic block and they can be put in the buffer very easily in practice. This scenario is quite rare in reality because there are many instructions that break into several small basic blocks [24]. Therefore, rule 2 and rule 3 are often the ones that terminate the trace.

Rule 2 and rule 3 handle instruction-path changing instructions (i.e. unconditional branches, RFE, and traps) and conditional branches, respectively. Therefore, it is necessary to enable the fill-buffer to classify instruction types. If the former was detected in either one of the incoming instructions, rule 2 will be applied. Basically, that instruction will be put in the current position provided by the pointers and then the line will be terminated immediately.

According to the structure of the fill-buffer (see figure 3.3), there are 2 fields in the trace information concerning branch instructions. The 'branch existing flag' indicates whether or not there is a branch existing in the line yet. This field will be set once a conditional branch instruction was inserted in the line. If the flag is set and if one of the incoming instructions was detected as a conditional branch, the logic will push that branch to start in the new line next to the current line even though there is a space left to fit that instruction.

Apart from manipulating pointers to place instructions into their places by applying the fill policy rules, the fill-logic also has to complete the trace information of each buffer line. However, this task has to be done in parallel with the pointer manipulation.

- The 'buffer line ready' flag is set immediately after the current line was terminated.
- The 'trace size' field is the counter that counts the number of instructions placed in the buffer line continuously until the line is terminated. Once the line was terminated, this field can tell how many instructions are in the particular buffer line.
- The 'branch instruction flag' was mentioned above.
- The 'branch position' indicates the location of the branch instruction within the trace. This field is updated once the branch instruction was placed in the buffer line. This information can be extracted from the pointers that locate the position of the instruction.

4.3.3 Trace Cache Memory

4.3.3.1 Trace cache memory structure

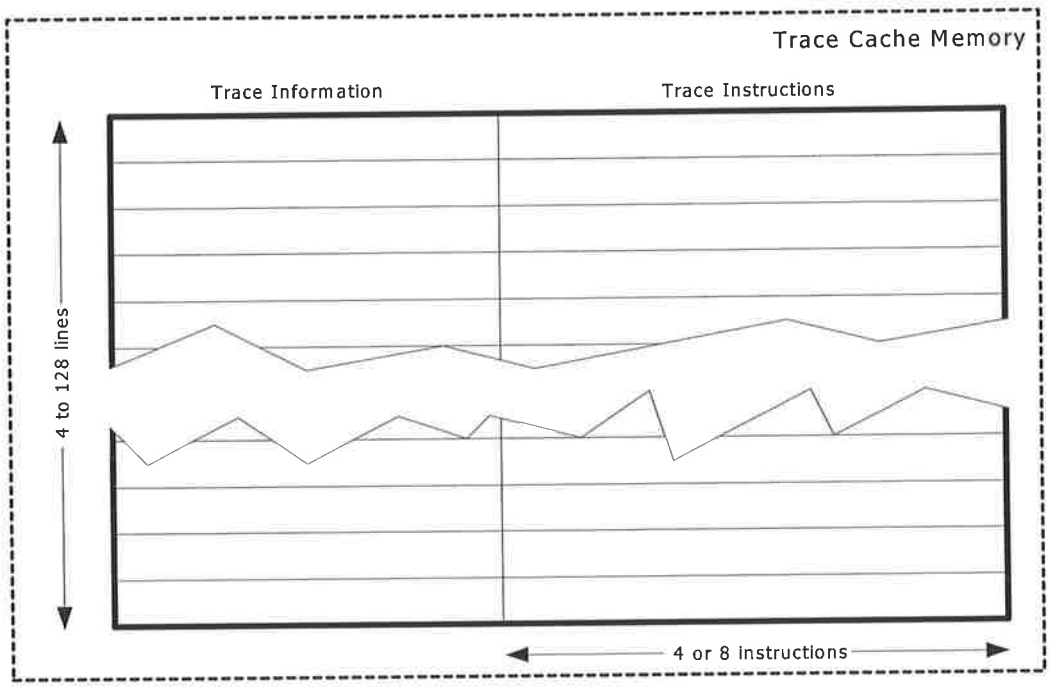


Figure 4.4: Trace cache memory structure.

The structure of the trace cache memory is quite similar to that of the fill-buffer. Figure 4.4 shows the structure of the trace cache memory space. There are also 2 sections: *Trace Information* and *Trace Instructions*.

- **Trace Instructions** stores only sequences of instructions since it is not necessary to store instruction addresses anymore. However, significant instruction addresses of the trace (i.e. the address of the first instruction and the address of the branch target instruction (if any) of the trace) are kept and appear as tags, which will be stored in the trace information portion.
- **Trace Information** comprises 6 information fields.

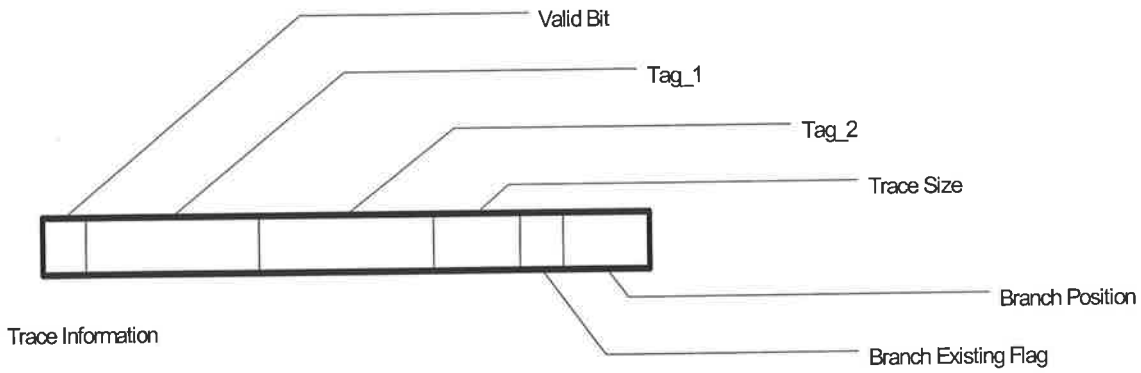


Figure 4.5: Trace Information portion of the trace cache memory.

- Valid bit: This is a flag to indicate whether or not the particular trace cache line is occupied by valid cache content.
- Tag_1: This is the tag field of the first instruction in the line.
- Tag_2: This is the tag field of the branch destination instruction address if there is a branch instruction available in the line. Otherwise, this field is an identical copy of tag_1.
- Trace Size.
- Branch Existing Flag.
- Branch Position.

The last three fields of trace information are identical copies of 'trace size', 'branch existing flag', and 'branch position' fields of the associated fill-buffer entry as described in section *4.3.2.1 Fill-Buffer Configuration*.

In this experiment, the number of trace cache memory lines is one of the interesting parameters to investigate. It ranges from 4 up to 512 cache lines (increasing by factor of 2) to analyze the effect of trace cache memory size on the trace cache utilization. This parameter will be varied for both *TC_4* and *TC_8* configuration.

4.3.3.2 Buffer-cache transfer

Every clock cycle, there must be a procedure to check whether there are any traces ready in the fill-buffer waiting to be transferred into the trace cache memory space. The buffer-cache transfer unit was built to accomplish this task. Moreover, the unit has to make a decision whether the new trace should be placed into the trace cache memory or dropped out.

- **Trace cache memory line selection**

  When there is a ready fill-buffer line, the address of the first instruction of the buffer line will be used as the trace cache memory line selector. Based on a direct mapped cache, the number of extracted bits used for line selection depends on the number of lines of trace cache memory (i.e. 2, 3, 4, 5, 6, and 7 bits are for 4, 8, 16, 32, 64, and 128 lines, respectively). The position of the extracted bits starts from the third bit of the address (see figure 4.6).

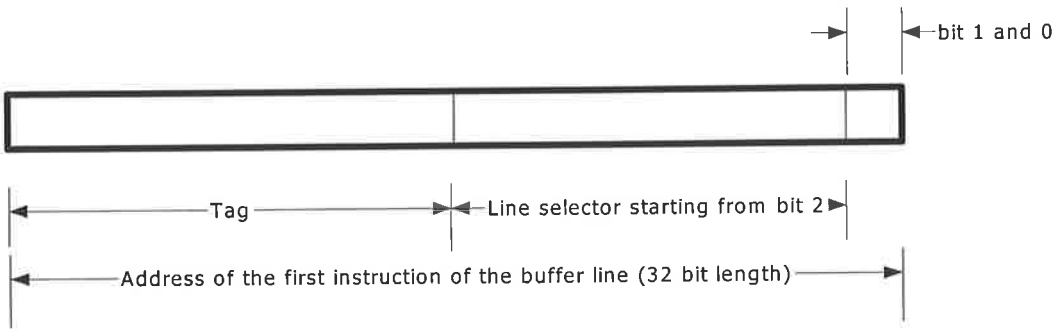


Figure 4.6: Trace cache line selector is extracted starting from bit 3 of the address word.

- **Commencing the transfer**

  Once the destination line was decoded, the transfer would be commenced if:

- There is more than one instruction in the ready-to-transfer fill-buffer line. This avoids single-instruction traces, which are not likely to be very useful, from occupying an entire TC line.
- The trace size of the ready-to-transfer fill-buffer line is longer than the existing trace in the selected memory line. Hypothetically, a longer trace provides more instructions and this would increase the probability of finding more useful instructions.

- **The contents to transfer**

  The contents from the fill-buffer line are:

  - All instructions in the trace (note: addresses of these instructions will be abandoned).
  - The extract (Bits 31 to 2) of the address of the first instruction and the address of the branch instruction destination (if any) to fit in 'Tag_1' and 'Tag_2' of the cache line.
  - Identical copies of 'Trace size', 'Branch Existing Flag', and 'Branch Position' from the buffer line for each field with the same name of the cache line.

- **Finishing the transfer**

  After the transfer was complete, there are 2 tasks to be done.

  - Reset all fields in the buffer line to make it ready to accommodate a new trace.
  - Set the 'Valid Bit' of the selected cache line to signal the validity of the content.

4.3.4 Trace Cache Hit Logic

As mentioned earlier, the original DLX model will be left untouched and the performance measurement will be done passively. The trace cache hit logic is the unit assigned to find out whether the instruction at the current address in the program counter register and its successors can be found in the trace cache. Therefore, this function is the point at which can be made the measurement of trace cache hit rate.

The typical instruction-cache 'hit' is the outcome of comparison between the value in the program counter register of the processor and the tag of the selected line. This is the valid hit although the required instruction is not necessarily the first

instruction of the line since its tag covers all of the instructions of the selected line. This is different from the trace cache 'hit' definition, particularly, the trace cache configuration of this experiment.

- **Trace cache information for 'hit' or 'miss'**

  The trace cache is supposed to collect instructions from the dynamic instruction stream. Although a trace cache line has a fixed size line into which instructions are placed, we can not forecast which instruction would be the first instruction of the cache line and how many instructions it can collect for a trace. Moreover, some traces may contain a branch instruction with a destination instruction whose address is not in consecutive order. Consequently, it is not possible to make the tag address cover all of the instructions in a trace cache line. In addition, the execution-path of the processor is only 2 instructions-wide. Then, all instructions from the selected trace cache line can not flow through the instruction-path simultaneously like those in the original instruction cache. One trace cache line might contain instructions to be fed through the instruction-path in several successive cycles. Therefore, the trace cache 'hit' or 'miss' depends on the corresponding trace information and the trace information must be able to indicate:

  - how many instructions there are in a particular trace,
  - the address of those instructions,
  - whether the trace possesses a branch,
  - the direction (taken / not taken) of that branch instruction.

- **Trace cache 'hit' or 'miss' determination**

  There are 2 types of trace cache 'hit': a hit on the first instruction of the cache line and a hit on the rest of the line. The former can be detected by matching the current value in the program counter (PC) with the 'tag_1' of the selected cache line. After a hit on the first instruction, it is possible to have a hit on the rest of the line in the next fetch. Thus, there must be a line-hit flag to indicate that the first instruction of that line has been hit. This method will enable the hit logic to check out the rest instructions.

2. Add the run time startup code *crt0.o* to the start of the compiler output.
3. Run this file through the standard link editor, *ld.*
4. Edit the *a.out* file to set the load addresses for the text and data segments, as required by the simulation model.
5. Use the perl script to transform the floating-point instructions.
6. Edit the file to add nops around the *jr* instructions.
7. Assemble the resulting file ("*dlx.asm*") into object code ("*dlx.out*") using *dlxasm*.

The *crt0.o* file and perl script are listed in Appendix D and are included in the companion CD-ROM.

At the end, the assembly codes were assembled into binary code as *.out* file for the processor simulation. The assembler named *dlxasm* (downloaded from *http://www.ashenden.com.au/designers-guide/DG-DLX-material.html*).

4.5 Simulation Testbench Configuration

The testbench configuration for simulating the superscalar DLX processor model has been set to run DLX binary-assembled files. A program used to run on the simulation must be named as 'dlx.out' and fit within 32 kilobytes memory range (0x0000 to 0x7FFF). Originally, the capacity of the main memory was only 16 kilobytes but this was expanded to accommodate larger test programs. Note that there must not be floating-point instructions in the test programs due to the processor design. The output file will be created as 'dlx.dump' if it was programmed to generate outputs.

4.6 Measuring the Trace Cache

In order to analyse the performance of the trace cache, the number of TC hits and misses were collected. Hit and miss counts of the original instruction cache and also the total cache accesses were also required for referencing purposes. The final sum of trace cache hits and misses from the trace cache lines is too coarse a metric to make any detailed analysis, therefore, the activities of each line of trace cache memory were recorded as described in the Appendix A.

In addition, there must also be analysis of the cache space usage because the trace cache model occupies real estate on the chip once it is implemented.

4.7 Conclusion

This research project benefits from the use of an existing processor model in that it was not necessary to set up the experiment from scratch. However, this model constrains the implementation of the original processor and the ability to expand the instruction cache. This chapter has described the way in which the trace cache was constructed and the method used to measure the trace cache performance in the aspects of usefulness relative to the instruction-cache and space usage. VHDL source code is included in Appendix B.

Chapter 5

Results

The trace cache was simulated in two configurations: a 4-instruction wide (*TC_4*) and an 8-instruction wide (*TC_8*) trace cache. In each case the number of trace cache lines was varied from 4 to 512. We will use 4L, 8L, and so on to denote individual cache line configurations. Each one of them will be simulated on 4 different test programs: *bubblesort (bs-a, bs-r, and bs-d)*, *primenumber (pn-20, pn-50, and pn-100)*, *permutation*, and *DCT*. For *bubblesort* and *primenumber*, the simulations were performed for 4L to 128L only, because trace cache performance became steady before reaching 128L and certainly would not vary for 256L and 512L configurations.

5.1 Overview

This experiment is meant to determine the effect of a trace cache on a narrow-issue processor like the Superscalar DLX in order to be able to determine whether the trace cache is worth considering for implementation on this kind of microprocessor. Obviously, performance comparison between the trace cache and the originally embedded instruction cache seems to be inevitable. Unlike the trace cache, however, it proved to be impractical to increase the capacity of the instruction cache in order to make a fair comparison between the two. For this reason the instruction cache capacity was not varied in these studies. The instruction cache can hold a maximum of 16 instructions when the trace cache can increase virtually unlimited. The best case for fair comparison would be *TC_4* at 4 lines of trace cache, in which the total capacity of the cache is 16 instructions (*TC_4* = 1 line contains 4 instructions).

Therefore, this analysis of this experiment will not focus on a head-to-head comparison of the performance between the two caches. Instead we will focus on the performance of different trace cache configurations.

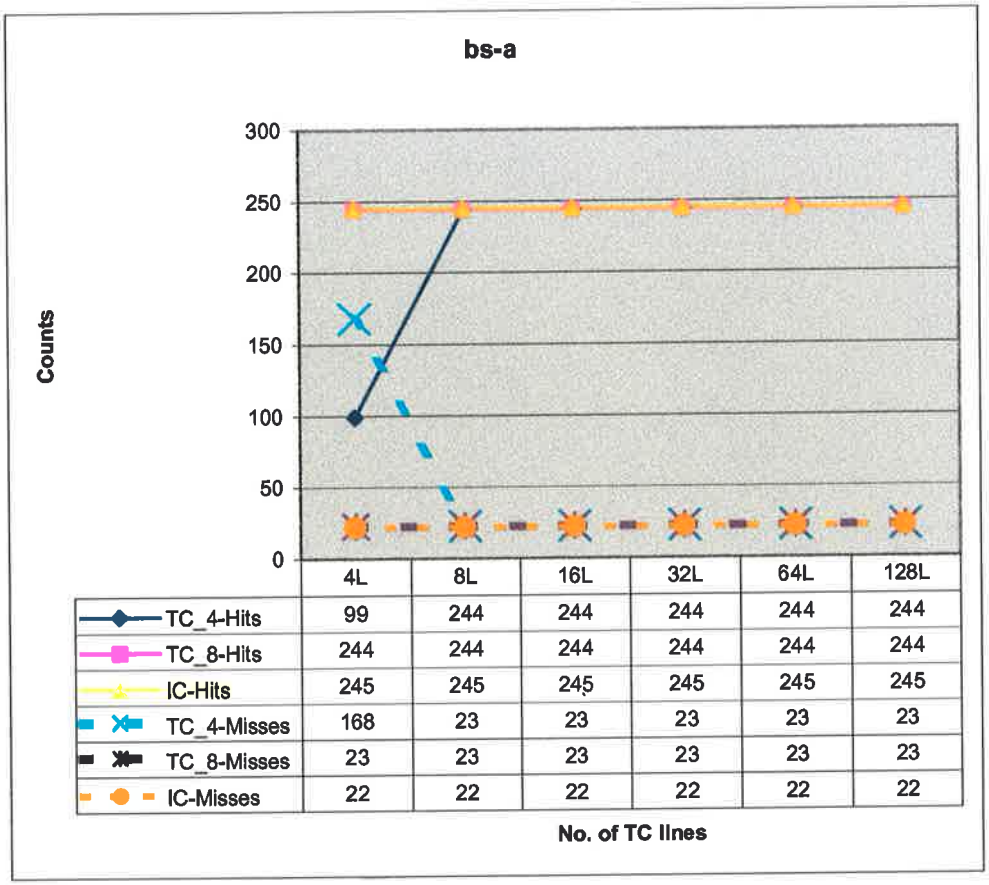
There are three sections analyzing the performance of the trace cache from different points of view. The first section shows the hit and miss counts on the trace cache while the capacity of the cache is increasing in both the width of the trace cache line and the number of trace cache lines. This section also shows hit and miss counts of the instruction cache to provide a reference point for trace cache performance.

The next section shows the percentage of hits and misses of the trace cache for different test programs. Hits and misses are presented in separated graphs to facilitate analysis of each of them individually. The last section displays how much of the trace cache space has been used and how much of it was left unused when the capacity of the trace cache is expanding. Please note that the words *trace cache* and *instruction cache* might be, from time to time, replaced with the abbreviations *TC* and *IC*, respectively.

5.2 Hits and Misses of the Trace Cache

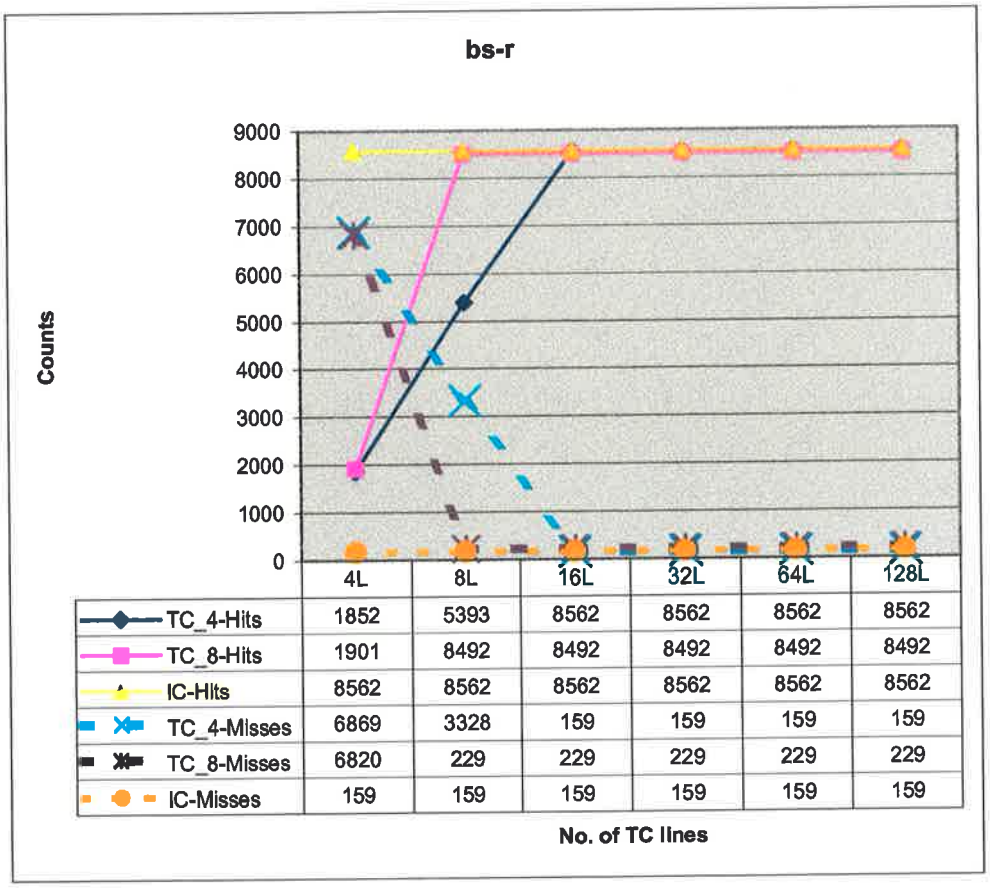
Fundamentally, the number of cache hits and misses is the performance indicator of cache memories. If there are more cache hits and fewer misses, it represents a better performance of the cache. This experiment has two main parameters that affect the performance of the trace cache when they vary, the size of a trace cache line and the total number of trace cache lines. The product of these parameters is actually the capacity of the trace cache but there may be different results for the same capacity from different parameter combinations because of the trace cache mechanism. Generally, a bigger trace cache capacity should perform better than a smaller one. However, it is essential to observe the actual results from these parameters that come into play with the fill policy in order to understand the design trade-offs.

The results are presented as graphs with associated data tables of individual test programs (figures 5.1a to 5.1h). Each of them shows the acquired number of hits and misses of all configurations of the trace cache and also of the original instruction cache.



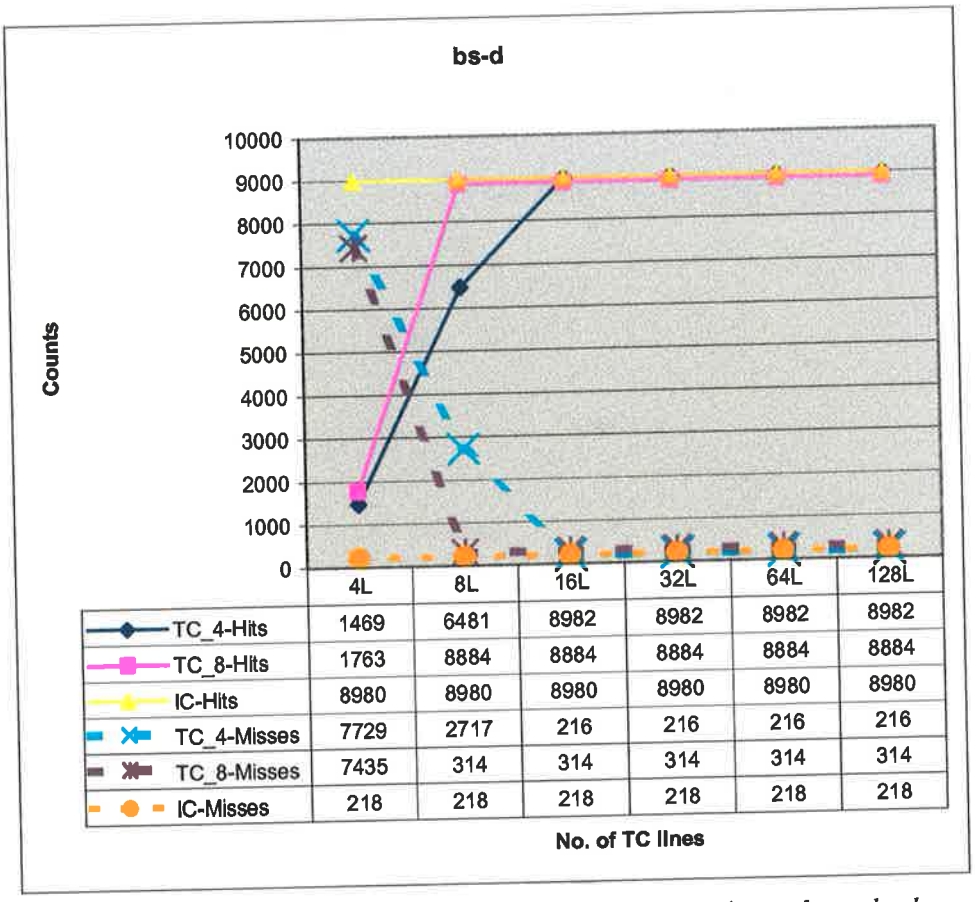
	4L	8L	16L	32L	64L	128L
TC_4-Hits	99	244	244	244	244	244
TC_8-Hits	244	244	244	244	244	244
IC-Hits	245	245	245	245	245	245
TC_4-Misses	168	23	23	23	23	23
TC_8-Misses	23	23	23	23	23	23
IC-Misses	22	22	22	22	22	22

Figure 5.1a : Hits and misses of the trace cache and the instruction cache on *bs-a*.



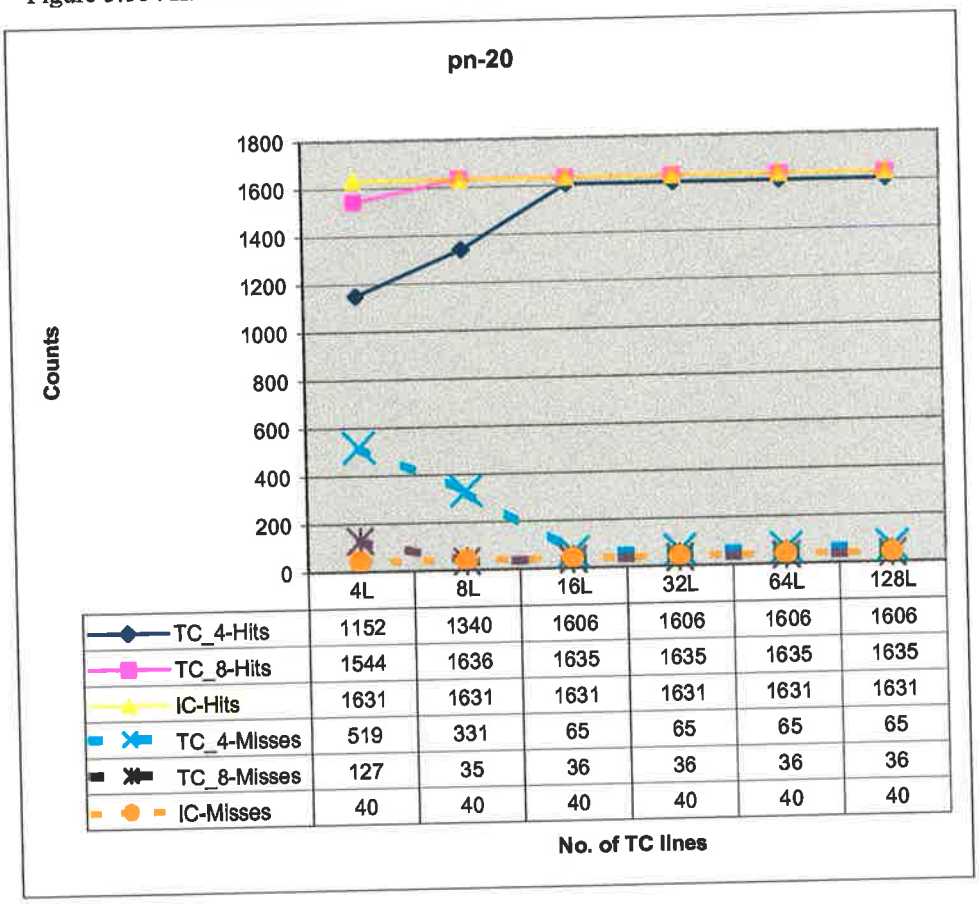
	4L	8L	16L	32L	64L	128L
TC_4-Hits	1852	5393	8562	8562	8562	8562
TC_8-Hits	1901	8492	8492	8492	8492	8492
IC-Hits	8562	8562	8562	8562	8562	8562
TC_4-Misses	6869	3328	159	159	159	159
TC_8-Misses	6820	229	229	229	229	229
IC-Misses	159	159	159	159	159	159

Figure 5.1b : Hits and misses of the trace cache and the instruction cache on *bs-r*.



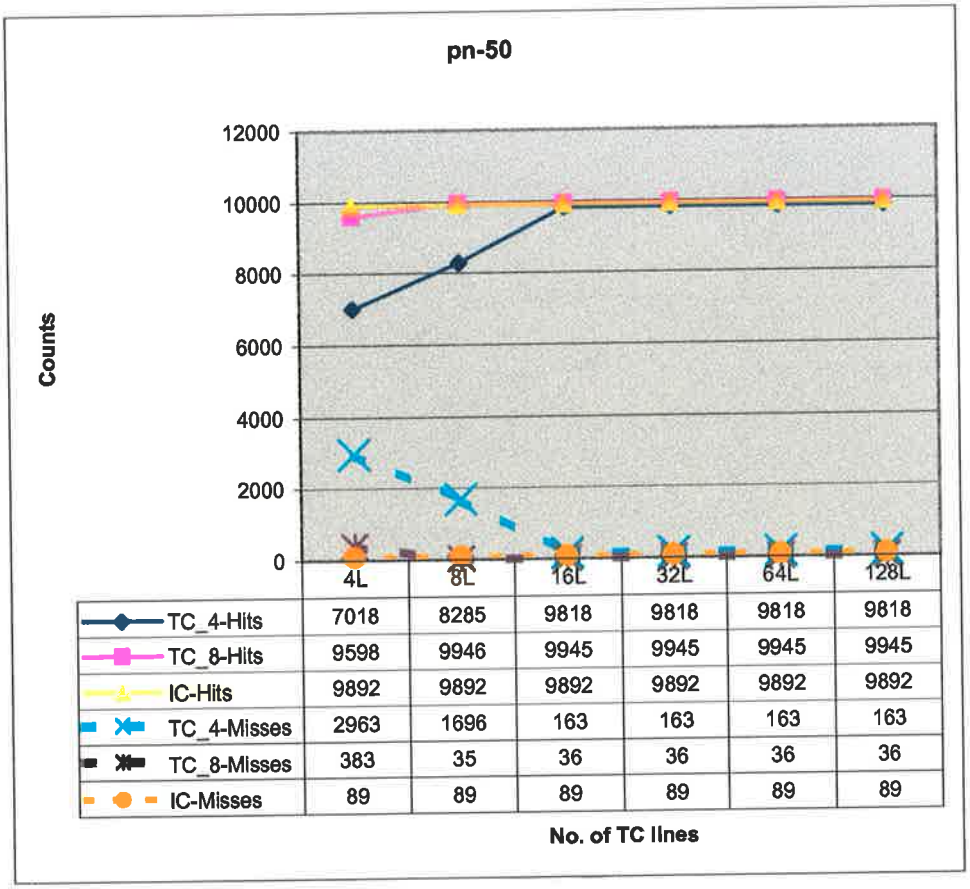
	4L	8L	16L	32L	64L	128L
TC_4-Hits	1469	6481	8982	8982	8982	8982
TC_8-Hits	1763	8884	8884	8884	8884	8884
IC-Hits	8980	8980	8980	8980	8980	8980
TC_4-Misses	7729	2717	216	216	216	216
TC_8-Misses	7435	314	314	314	314	314
IC-Misses	218	218	218	218	218	218

Figure 5.1c : Hits and misses of the trace cache and the instruction cache on *bs-d*.



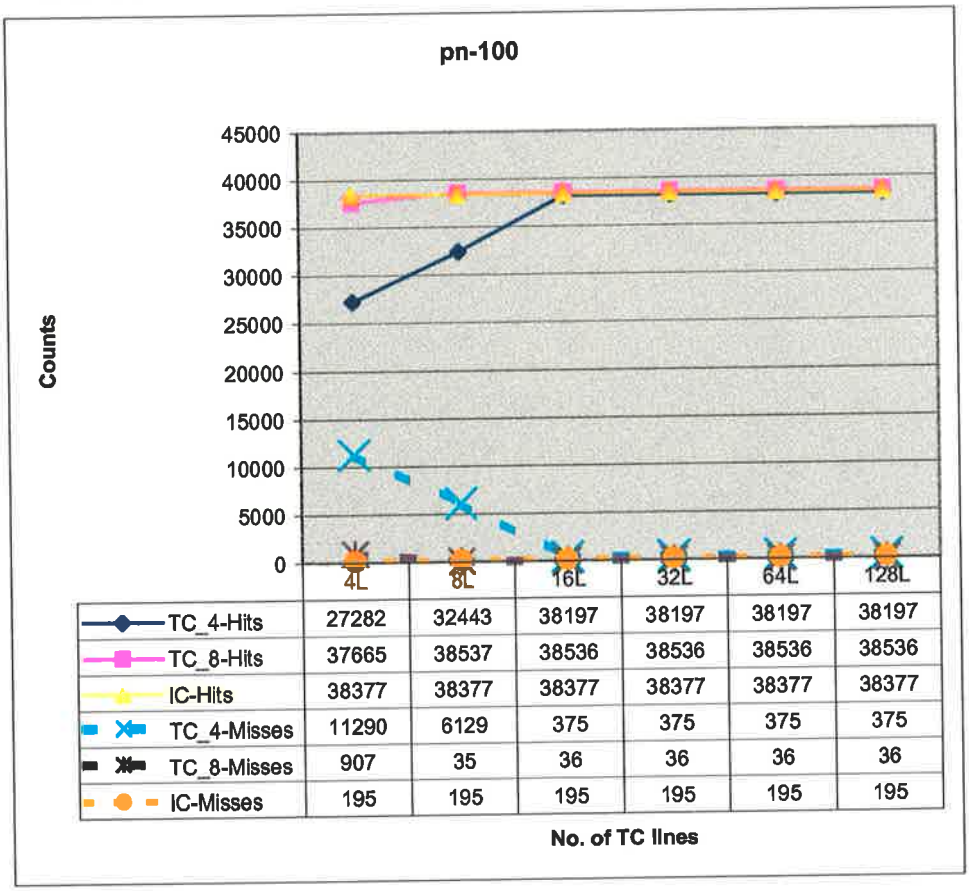
	4L	8L	16L	32L	64L	128L
TC_4-Hits	1152	1340	1606	1606	1606	1606
TC_8-Hits	1544	1636	1635	1635	1635	1635
IC-Hits	1631	1631	1631	1631	1631	1631
TC_4-Misses	519	331	65	65	65	65
TC_8-Misses	127	35	36	36	36	36
IC-Misses	40	40	40	40	40	40

Figure 5.1d : Hits and misses of the trace cache and the instruction cache on *pn-20*.



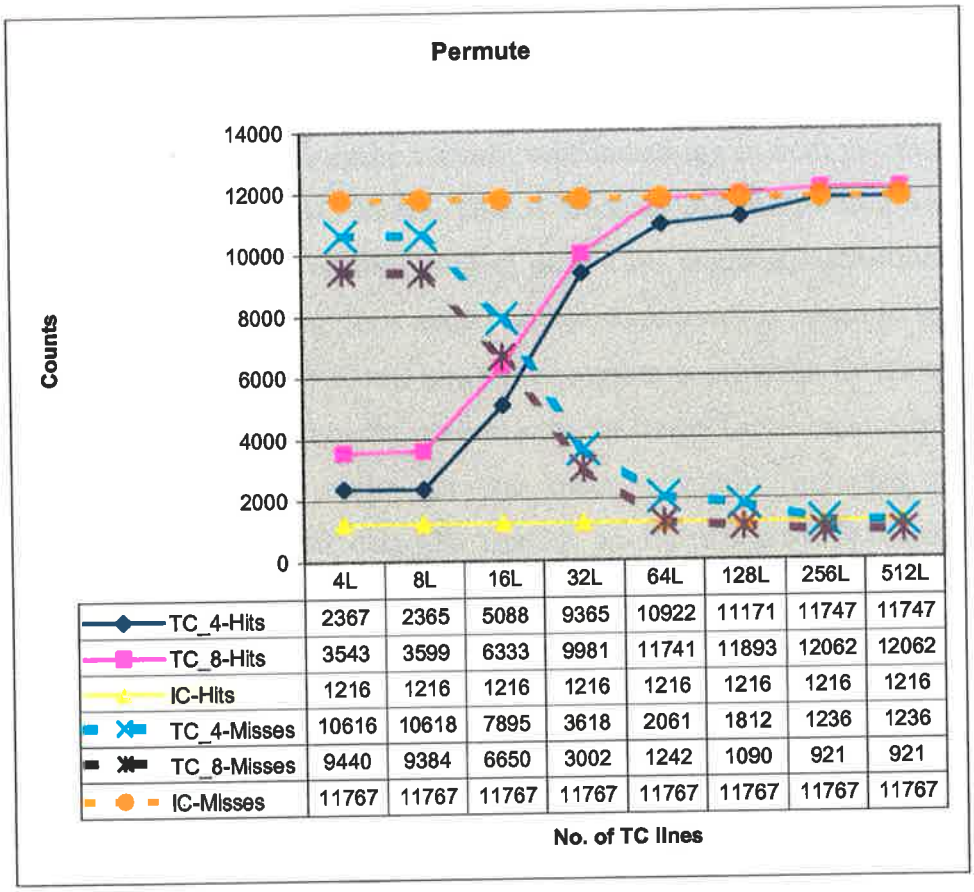
	4L	8L	16L	32L	64L	128L
TC_4-Hits	7018	8285	9818	9818	9818	9818
TC_8-Hits	9598	9946	9945	9945	9945	9945
IC-Hits	9892	9892	9892	9892	9892	9892
TC_4-Misses	2963	1696	163	163	163	163
TC_8-Misses	383	35	36	36	36	36
IC-Misses	89	89	89	89	89	89

Figure 5.1e : Hits and misses of the trace cache and the instruction cache on *pn-50*.



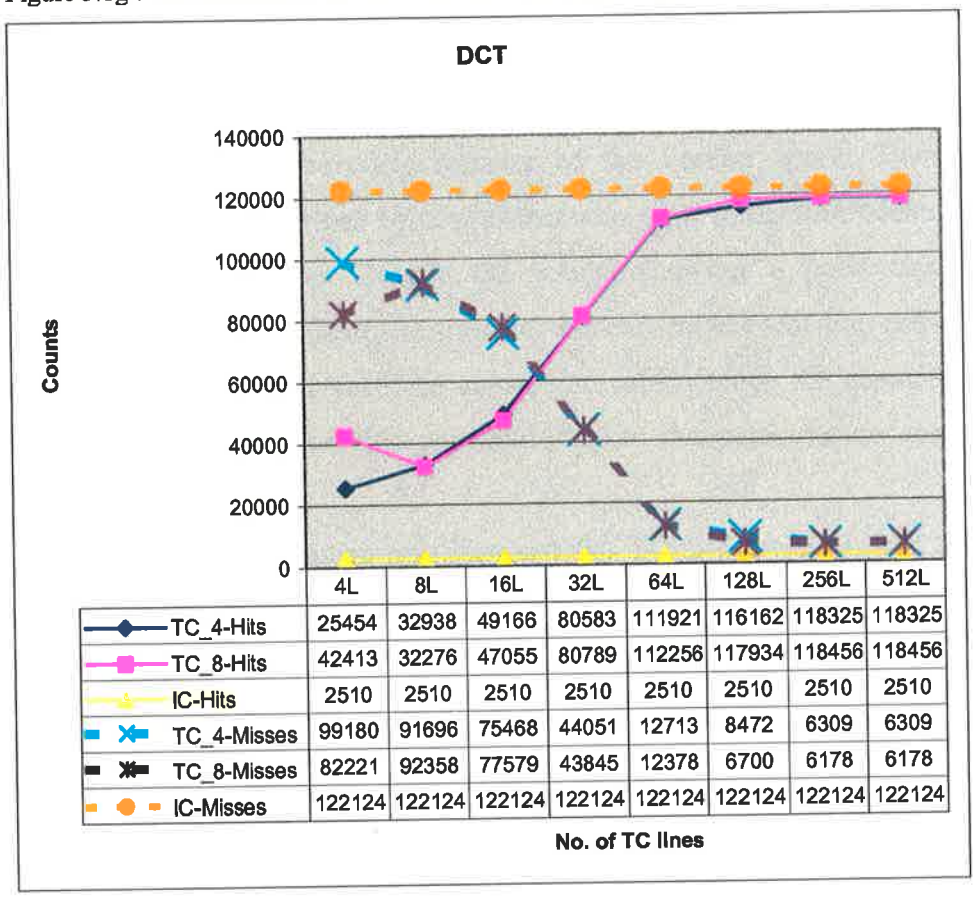
	4L	8L	16L	32L	64L	128L
TC_4-Hits	27282	32443	38197	38197	38197	38197
TC_8-Hits	37665	38537	38536	38536	38536	38536
IC-Hits	38377	38377	38377	38377	38377	38377
TC_4-Misses	11290	6129	375	375	375	375
TC_8-Misses	907	35	36	36	36	36
IC-Misses	195	195	195	195	195	195

Figure 5.1f : Hits and misses of the trace cache and the instruction cache on *pn-100*.



	4L	8L	16L	32L	64L	128L	256L	512L
TC_4-Hits	2367	2365	5088	9365	10922	11171	11747	11747
TC_8-Hits	3543	3599	6333	9981	11741	11893	12062	12062
IC-Hits	1216	1216	1216	1216	1216	1216	1216	1216
TC_4-Misses	10616	10618	7895	3618	2061	1812	1236	1236
TC_8-Misses	9440	9384	6650	3002	1242	1090	921	921
IC-Misses	11767	11767	11767	11767	11767	11767	11767	11767

Figure 5.1g : Hits and misses of the trace cache and the instruction cache on *Permute*.



	4L	8L	16L	32L	64L	128L	256L	512L
TC_4-Hits	25454	32938	49166	80583	111921	116162	118325	118325
TC_8-Hits	42413	32276	47055	80789	112256	117934	118456	118456
IC-Hits	2510	2510	2510	2510	2510	2510	2510	2510
TC_4-Misses	99180	91696	75468	44051	12713	8472	6309	6309
TC_8-Misses	82221	92358	77579	43845	12378	6700	6178	6178
IC-Misses	122124	122124	122124	122124	122124	122124	122124	122124

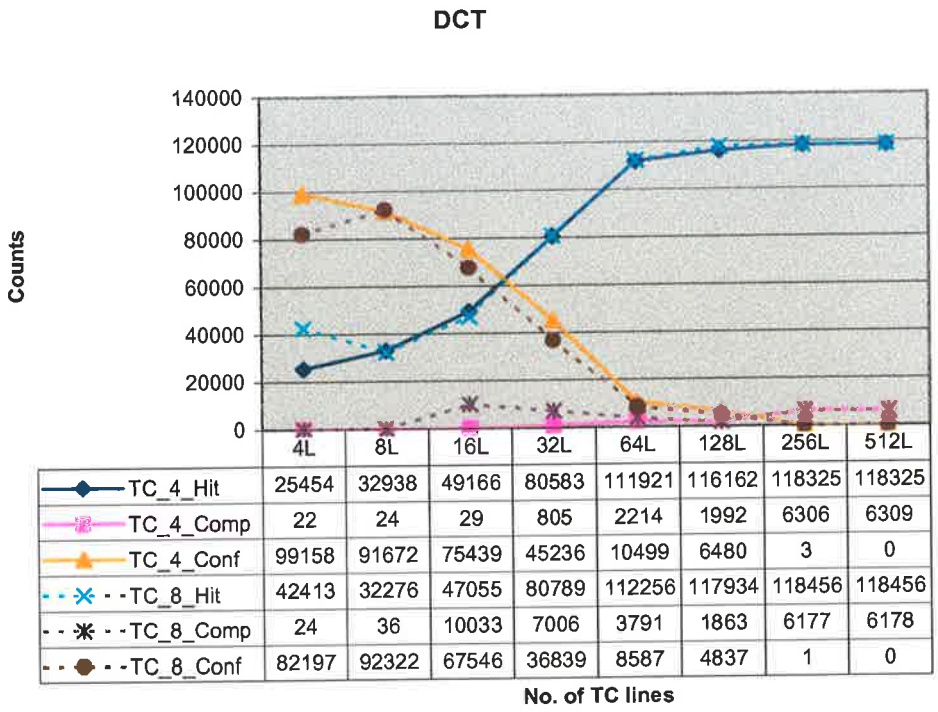
Figure 5.1h : Hits and misses of the trace cache and the instruction cache on *DCT*.

In general, *TC_8* performs better than *TC_4* in particular at the same number of TC lines except for *DCT*. Undoubtedly, the longer traces of *TC_8* increase the opportunity to find useful instructions in a single trace and the larger space allows more instructions to fit in and also decreases the chances of overwriting useful ones due to space contention. However, this advantage is not effective in every program as mentioned in the explanation above. The effectiveness depends on the pattern of dynamic execution of each particular program, so the advantage is not perfectly predictable.

For the case of *bubblesort*, *TC_8* at 4L is not as effective as *TC_4* because of two significant reasons evident from the raw data from the simulation (referencing the companion CD-ROM). This analysis is based on the comparison of 4L and 8L of *bs-r* and *bs-d*. The first reason is that 4L provides less space to hold useful traces long enough to offer required instructions and that particular traces were replaced by other traces that are not well used and live too long and, therefore, result in a lot of misses. The other reason is there is too much overwriting to the same line too frequently, so the useful traces cannot live long enough to produce hits. All of this is chiefly the problem of cache space contention combined with the direct-mapped scheme. Therefore, more TC lines can relax this drawback and offer more TC hit counts as we can see from the results.

DCT is an exception from all of the graphs mentioned above. The performance of *TC_8* is not better than *TC_4* at the same number of TC lines. This peculiarity can be explained by the comparison graphs of TC hit, compulsory miss, and conflict miss of *TC_4* and *TC_8* in figure 5.2.

In the following discussion we refer to misses as either *compulsory misses* or *conflict misses*. When the line was selected at the first time but there is no instruction in it (valid bit = 0), it is called a *compulsory miss*. In contrast, if the selected line contains valid data but not the required instructions, it is called a *conflict miss*.



	4L	8L	16L	32L	64L	128L	256L	512L
TC_4_Hit	25454	32938	49166	80583	111921	116162	118325	118325
TC_4_Comp	22	24	29	805	2214	1992	6306	6309
TC_4_Conf	99158	91672	75439	45236	10499	6480	3	0
TC_8_Hit	42413	32276	47055	80789	112256	117934	118456	118456
TC_8_Comp	24	36	10033	7006	3791	1863	6177	6178
TC_8_Conf	82197	92322	67546	36839	8587	4837	1	0

Figure 5.2: Comparison between TC hit and Compulsory Miss and Conflict Miss of *DCT*.

Figure 5.2 shows the comparison of three features (TC hit and both TC miss types) of *TC_4* and *TC_8*. Suppose that the trend of the *TC_4* graphs draws the baseline of normal behavior in which the TC hit rate is increasing, representing better performance when there are more TC lines. Meanwhile, the conflict miss rate falls and the compulsory miss rate is a little higher as the number of lines increases. We can see that the behavior of *TC_8*'s graphs is different. If *TC_8* consistently performed better than *TC_4*, either of compulsory miss rate or conflict miss rate should be distinctly lower than those of *TC_4* from 4L to 512L. But it is only at 4L that *TC_8* performs better than *TC_4* because of the low conflict miss rate. At 8L, the conflict miss rate of *TC_8* is higher than that of *TC_4*. This unexpected effect has to be explored by consulting the raw data on Appendix C which contains excepts from the corresponding simulation *log* files of *DCT*. Comparing the data of *TC_4* and *TC_8* at 8L for *DCT*, it reveals that trace line number 2 of *TC_8* has a noticeably higher conflict miss than *TC_4*. Although the overwriting count is only a few in *TC_8*, the conflict miss rate is high. This means that useful traces are overwritten by less useful ones. Moreover, the less useful traces occupy lines for too long. Comparing this to the same line of *TC_4*, there are more trace overwrites but fewer conflict misses.

At 16L, although the conflict miss rate decreases the compulsory miss rate suddenly gets higher than the compulsory miss rate of *TC_4* which makes the TC hit

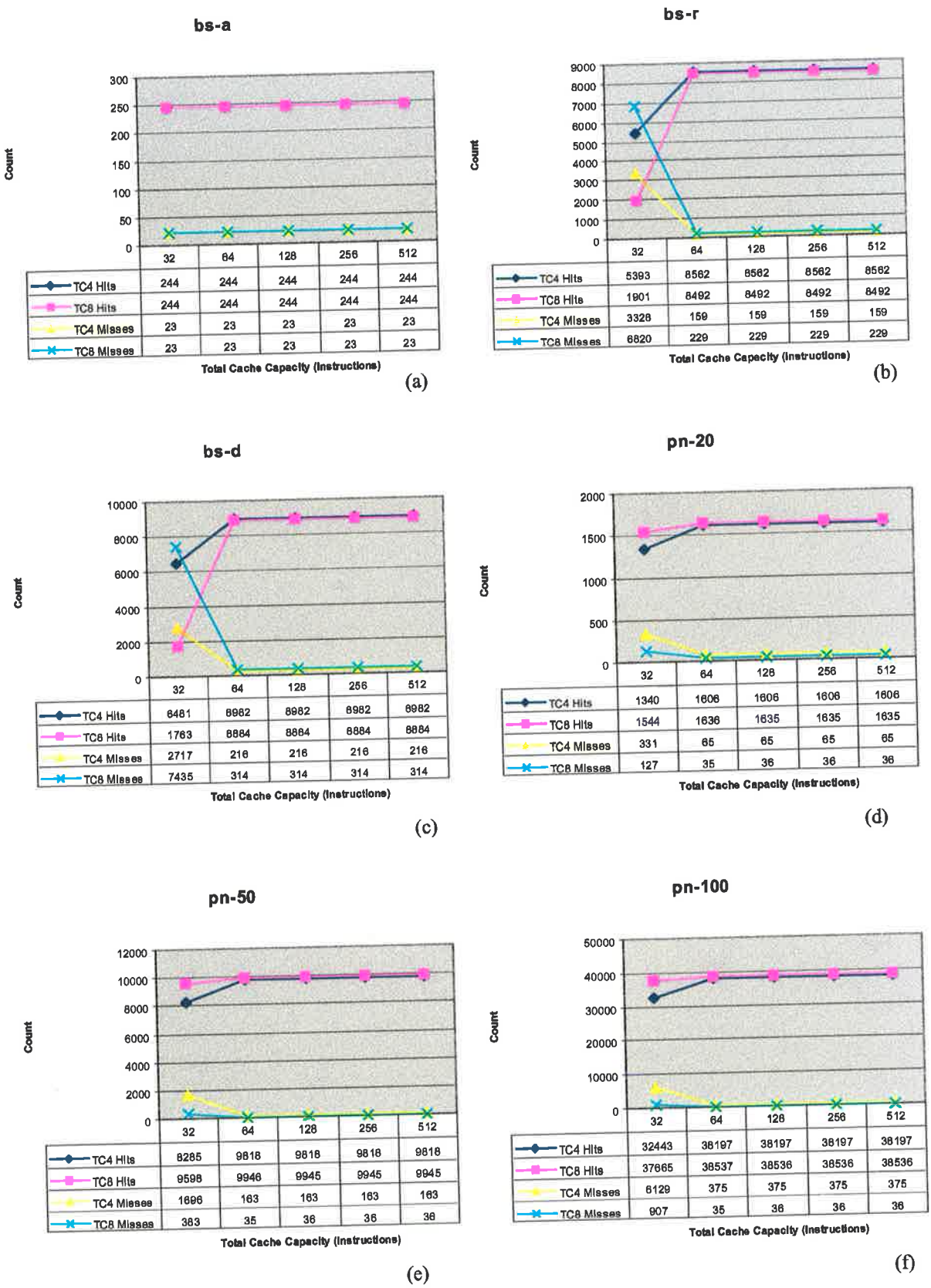
rate of *TC_8* still less than *TC_4*'s. After 16L, the falling conflict miss rate of *TC_8* is offset by an increasing compulsory miss rate. The raw data for *TC_8* at 16L to 64L of *DCT* gives the insight into this occurrence. This happens at trace cache line number 2 that has high compulsory miss which dominates the total compulsory misses.

The above comparison between *TC_4* and *TC_8* for the same number of TC lines is actually not fair because *TC_8* naturally has more room for instructions. At a particular number of TC lines, *TC_8* has twice the capacity of *TC_4*. For example, the 8L-trace cache of *TC_8* is able to store 64 instructions as is the 16L-trace cache of *TC_4*. Hence, it is interesting to make a comparison using total capacity to categorize a particular comparison between *TC_4* and *TC_8* as shown in table 5.2.

TC configuration	Number of TC lines						
TC_4	8L	16L	32L	64L	128L	256L	512L
TC_8	4L	8L	16L	32L	64L	128L	256L
Cache capacity (instructions)	32	64	128	256	512	1024	2048

Table 5.2: The equivalent cache capacity of different trace cache configurations.

For the *bubblesort* and *primenumber* programs, the comparison will stop at 128L of *TC_4* while all of the above configurations (8L-512L of *TC_4*) will be shown for *Permute* and *DCT*. Figures 5.3a to 5.3h show the graphs of hits and misses based on the total cache capacity of each TC line width.



bs-a	32	64	128	256	512
TC4 Hits	244	244	244	244	244
TC8 Hits	244	244	244	244	244
TC4 Misses	23	23	23	23	23
TC8 Misses	23	23	23	23	23

bs-r	32	64	128	256	512
TC4 Hits	5393	8562	8562	8562	8562
TC8 Hits	1901	8492	8492	8492	8492
TC4 Misses	3328	159	159	159	159
TC8 Misses	6820	229	229	229	229

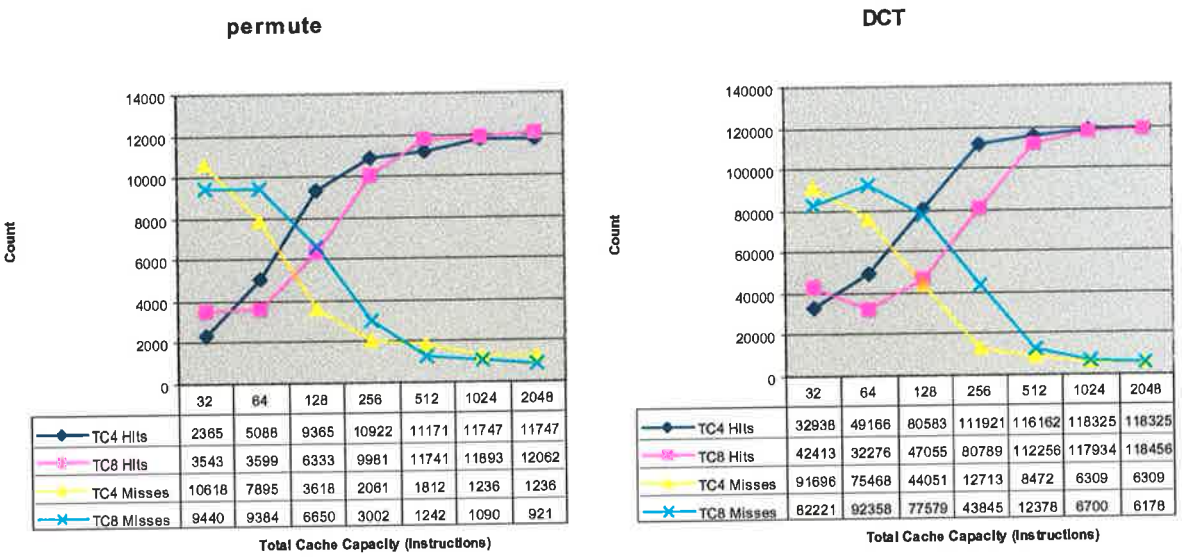
bs-d	32	64	128	256	512
TC4 Hits	6481	8982	8982	8982	8982
TC8 Hits	1763	8884	8884	8884	8884
TC4 Misses	2717	216	216	216	216
TC8 Misses	7435	314	314	314	314

pn-20	32	64	128	256	512
TC4 Hits	1340	1606	1606	1606	1606
TC8 Hits	1544	1636	1635	1635	1635
TC4 Misses	331	65	65	65	65
TC8 Misses	127	35	36	36	36

pn-50	32	64	128	256	512
TC4 Hits	8285	9818	9818	9818	9818
TC8 Hits	9598	9946	9945	9945	9945
TC4 Misses	1696	163	163	163	163
TC8 Misses	383	35	36	36	36

pn-100	32	64	128	256	512
TC4 Hits	32443	38197	38197	38197	38197
TC8 Hits	37665	38537	38536	38536	38536
TC4 Misses	6129	375	375	375	375
TC8 Misses	907	35	36	36	36

Figure 5.3: Hit and miss comparison between *TC_4* and *TC_8* at the same cache capacity of (a) *bs-a*, (b) *bs-r*, (c) *bs-d*, (d) *pn-20*, (e) *pn-50*, and (f) *pn-100*.



(g) (h)

Figure 5.3 : Hit and miss comparison between *TC_4* and *TC_8* at the same cache capacity of (g) *Permute*, and (h) *DCT*.
(continue)

Figures 5.3a to 5.3h show that *TC_8* does not significantly outperform *TC_4* as it might be expected to when the total cache space is equal between the two. For *Permute* and *DCT*, *TC_8* performs generally worse than *TC_4*. The hypothetical reason for this is the importance of TC line numbers. Although the wider TC line offers the opportunity to get more hits on the contents of traces, there is also the possibility that not all of the contents are useful and, in addition, not all fully useful traces are longer than 4 instructions. Therefore, when comparing *TC_4* with *TC_8* on a fair basis (equal capacity), *TC_4* generally performs better than *TC_8*. *Permute* and *DCT* are good basis for this comparison since *bubblesort* and *primenumber* are too short to tell the difference due to their early saturation of TC hit rate.

In the end, these results are useful for determination of chip area investment in the stage of hardware implementation in which the choice between increasing the number of TC lines and widening the traces is considered.

5.3 Percentage of Trace Cache Hits and Misses

This section focuses on the percentage of trace cache hits and misses in order to show and compare the nature of the hits and misses of each test programs. Trace cache hits will be discussed in term of 3 figures.

- *Total trace cache hit* – This is the percentage of trace cache hits relative to total cache accesses. Total trace cache hit is the sum of the *trace cache first tag hit* and the *trace cache line content hit.*
- *Trace cache first tag hit* – The first tag of the trace cache line represents the first instruction of the trace. When the first tag was hit, it means the line can start to supply instructions. However, the other instructions in that line can be fetched or not depending on another hit signal – *trace cache line content hit.*
- *Trace cache line content hit* – As mentioned above, this hit is counted when instructions after the hit first instruction are eligible to be fetched.

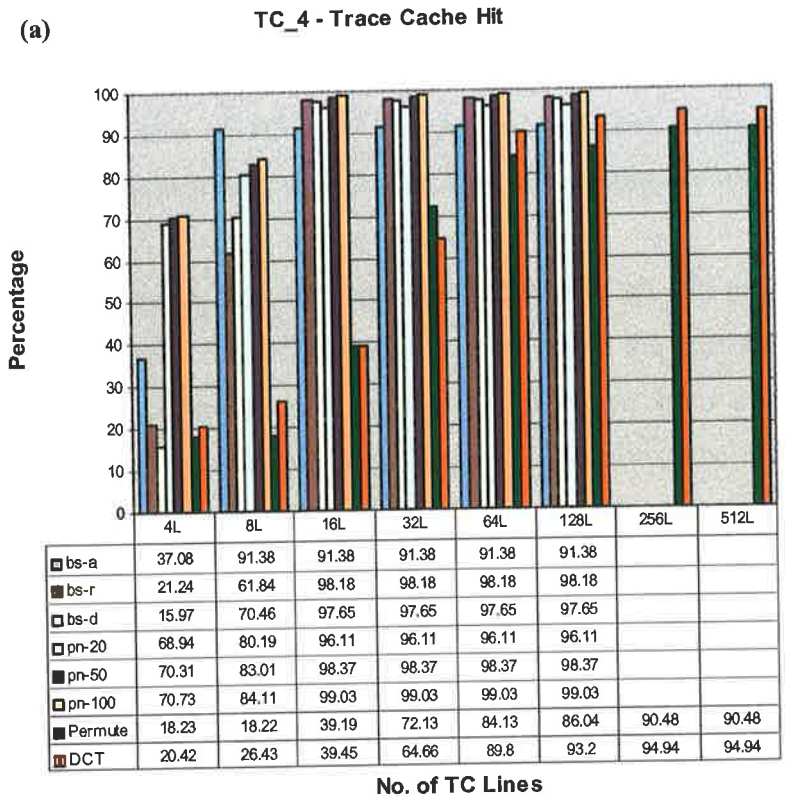
As we described the definition of *compulsory misses* or *conflict misses* earlier, both of these figures are elements of the total trace cache miss percentage, which is the percentage of total trace cache hits subtracted from 100.

Each of figures 5.4 to 5.8 shows the graphs of percentage of particular hit and miss of *TC_4* and *TC_8* together for comparison purposes.

5.3.1 Trace cache hits

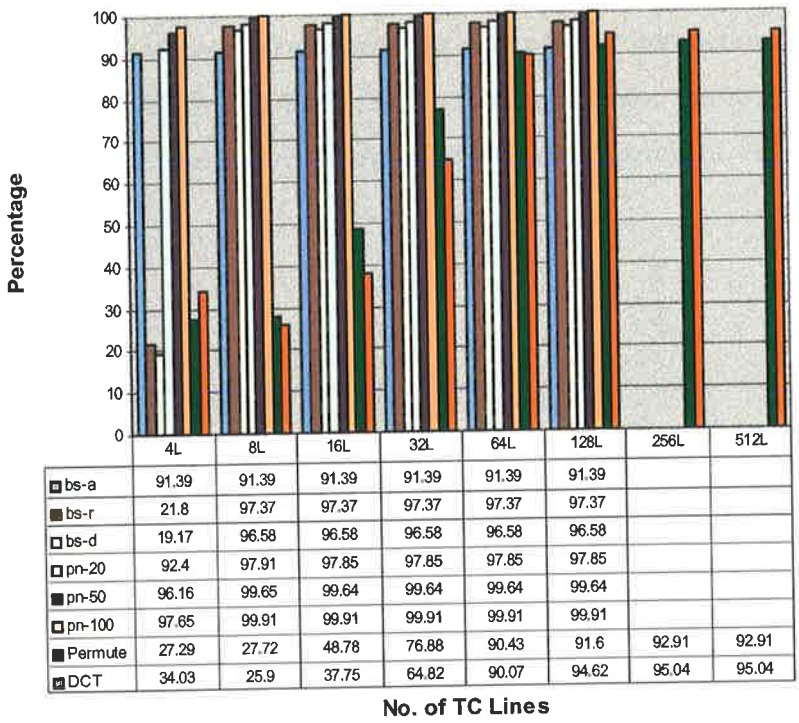
Figure 5.4 shows the graph of percentage of total trace cache hit of (a) *TC_4* and (b) *TC_8*. Hence, they provide comparable figures among all test programs in which the numbers of total cache accesses are different.

(a)



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	37.08	91.38	91.38	91.38	91.38	91.38		
bs-r	21.24	61.84	98.18	98.18	98.18	98.18		
bs-d	15.97	70.46	97.65	97.65	97.65	97.65		
pn-20	68.94	80.19	96.11	96.11	96.11	96.11		
pn-50	70.31	83.01	98.37	98.37	98.37	98.37		
pn-100	70.73	84.11	99.03	99.03	99.03	99.03		
Permute	18.23	18.22	39.19	72.13	84.13	86.04	90.48	90.48
DCT	20.42	26.43	39.45	64.66	89.8	93.2	94.94	94.94

(b)



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	91.39	91.39	91.39	91.39	91.39	91.39		
bs-r	21.8	97.37	97.37	97.37	97.37	97.37		
bs-d	19.17	96.58	96.58	96.58	96.58	96.58		
pn-20	92.4	97.91	97.85	97.85	97.85	97.85		
pn-50	96.16	99.65	99.64	99.64	99.64	99.64		
pn-100	97.65	99.91	99.91	99.91	99.91	99.91		
Permute	27.29	27.72	48.78	76.88	90.43	91.6	92.91	92.91
DCT	34.03	25.9	37.75	64.82	90.07	94.62	95.04	95.04

Figure 5.4 : Percentage of total trace cache hit of (a) *TC_4* and (b) *TC_8*.

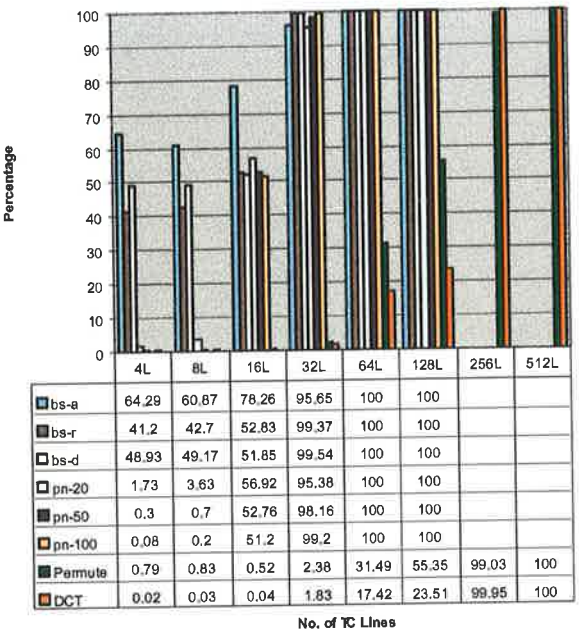
5.3.1 Trace cache misses

There are two kinds of TC misses – *compulsory miss* and *conflict miss* – as explained earlier. Figures 5.7 and 5.8 show the percentage of each kind of miss for both *TC_4* and *TC_8*, taking the total miss count as 100% and each miss is the share of the total miss count.

We would expect that the percentage of *compulsory miss* would be increasing when the number of TC lines is higher while *conflict miss* tends to go the opposite way. This can be explained by the nature of the cache scheme. When there are a few TC lines, it is most likely that an existing trace is replaced by a new incoming trace since they are mapped at the same line. Therefore, when that line is engaged by a new one that is not matched with the requirements of the fetch unit, it signals *conflict miss*. *Conflict misses* can be resolved by increasing the number of TC lines and eventually when there is enough room to store most or all of the instructions, the *conflict miss* rate is zero. Likewise, *compulsory miss* can be explained from the same effect of increasing the number of TC lines. More TC lines increase the probability of a hit on an empty cache line. Hence, eventually all of the TC misses are *conflict misses* when the trace cache is big enough to cover all executed instructions.

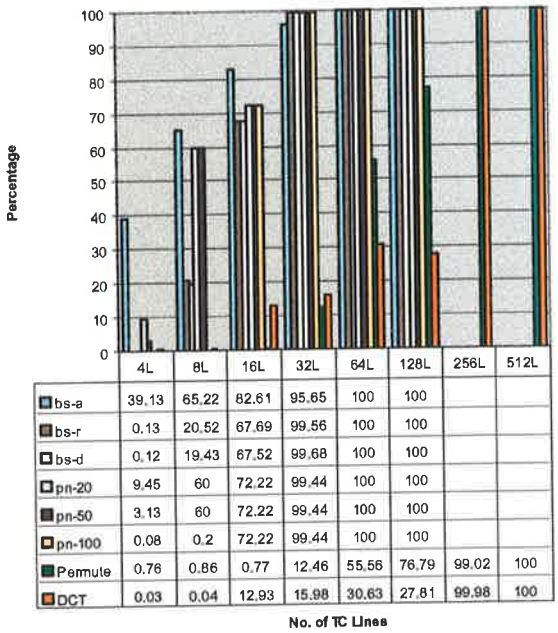
If we look at the results of TC misses, each of them has the tendency as hypothesized. However, the results of each test program are different and unpredictable when two parameters – the trace size and the number of TC lines – are varied. Therefore, no firm conclusion can be made about how the variation of trace cache parameters affects the behavior of trace cache misses.

(a)



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	64.29	60.87	78.26	95.65	100	100		
bs-r	41.2	42.7	52.83	99.37	100	100		
bs-d	48.93	49.17	51.85	99.54	100	100		
pn-20	1.73	3.63	56.92	95.38	100	100		
pn-50	0.3	0.7	52.76	98.16	100	100		
pn-100	0.08	0.2	51.2	99.2	100	100		
Permute	0.79	0.83	0.52	2.38	31.49	55.35	99.03	100
DCT	0.02	0.03	0.04	1.83	17.42	23.51	99.95	100

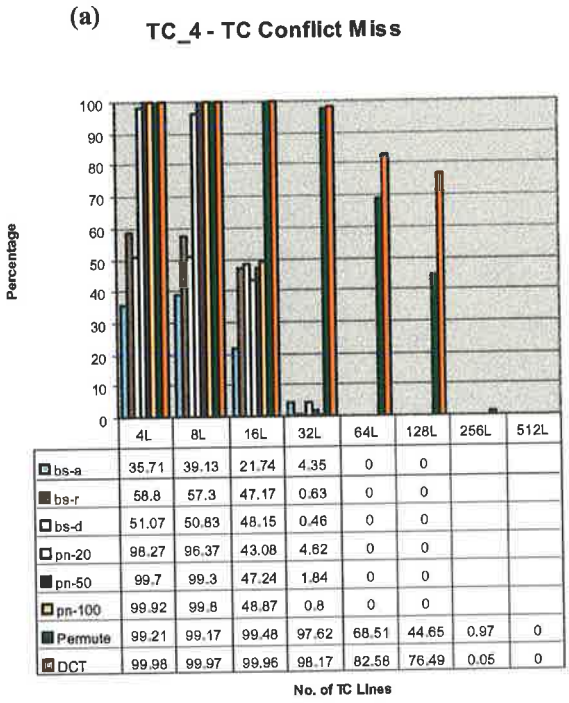
(b)



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	39.13	65.22	82.61	95.65	100	100		
bs-r	0.13	20.52	67.69	99.56	100	100		
bs-d	0.12	19.43	67.52	99.68	100	100		
pn-20	9.45	60	72.22	99.44	100	100		
pn-50	3.13	60	72.22	99.44	100	100		
pn-100	0.08	0.2	72.22	99.44	100	100		
Permute	0.76	0.86	0.77	12.46	55.56	76.79	99.02	100
DCT	0.03	0.04	12.93	15.98	30.63	27.81	99.98	100

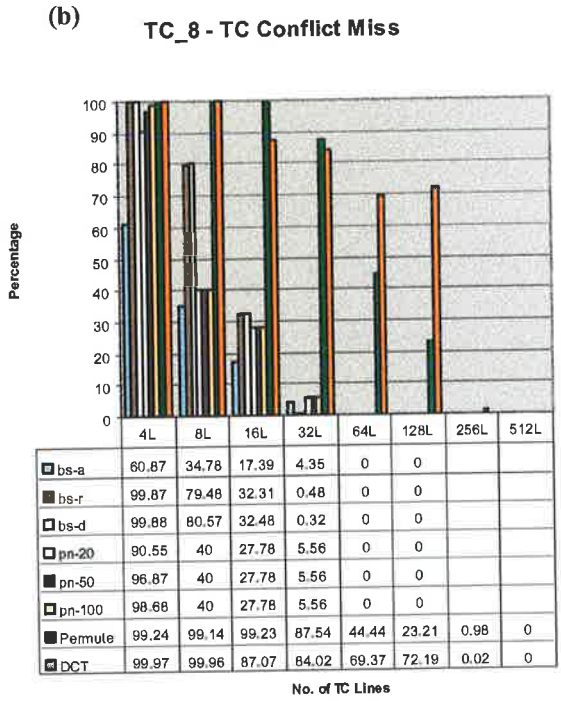
Figure 5.7 : Percentage of TC *Compulsory Miss* of (a) *TC_4* and (b) *TC_8*.

(a)



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	35.71	39.13	21.74	4.35	0	0		
bs-r	58.8	57.3	47.17	0.63	0	0		
bs-d	51.07	50.83	48.15	0.46	0	0		
pn-20	98.27	96.37	43.08	4.62	0	0		
pn-50	99.7	99.3	47.24	1.84	0	0		
pn-100	99.92	99.8	48.87	0.8	0	0		
Permute	99.21	99.17	99.48	97.62	68.51	44.65	0.97	0
DCT	99.98	99.97	99.96	98.17	82.58	76.49	0.05	0

(b)



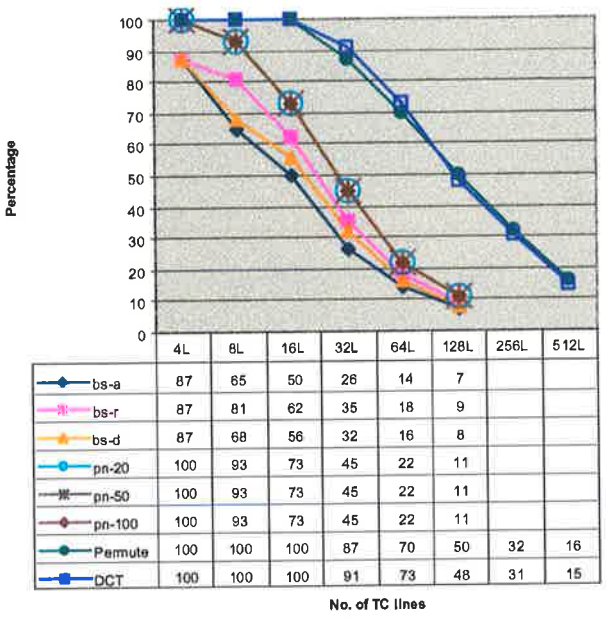
	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	60.87	34.78	17.39	4.35	0	0		
bs-r	99.87	79.48	32.31	0.48	0	0		
bs-d	99.88	80.57	32.48	0.32	0	0		
pn-20	90.55	40	27.78	5.56	0	0		
pn-50	96.87	40	27.78	5.56	0	0		
pn-100	98.68	40	27.78	5.56	0	0		
Permute	99.24	99.14	99.23	87.54	44.44	23.21	0.98	0
DCT	99.97	99.96	87.07	84.02	69.37	72.19	0.02	0

Figure 5.8 : Percentage of TC *Conflict Miss* of (a) *TC_4* and (b) *TC_8*.

5.4 Trace Cache Space Usage

Increasing trace cache space seems to improve the hit rate but also introduces additional expenses. Therefore, it is important to obtain some indication of how efficiently each configuration uses the available memory. Figures 5.9 and 5.10 show the percentage of trace cache space usage of *TC_4* and *TC_8*, respectively. The individual percentage was calculated from the total number of instructions stored in the cache divided by the maximum instruction capacity of the cache.

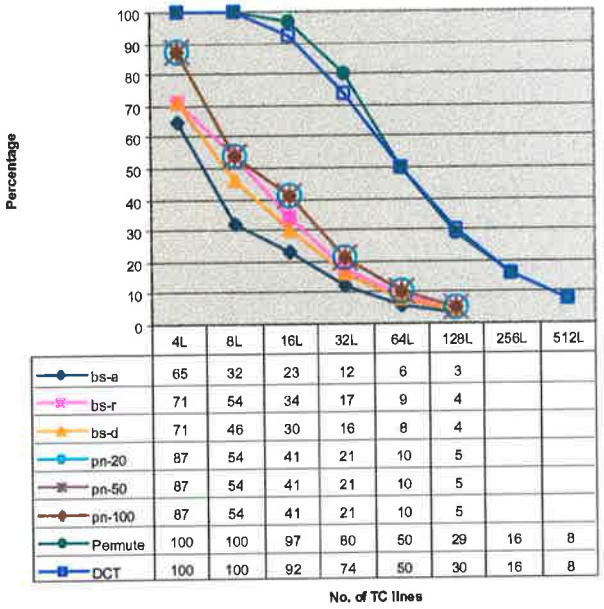
TC_4 - Percentage of Cache Space Usage



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	87	65	50	26	14	7		
bs-r	87	81	62	35	18	9		
bs-d	87	68	56	32	16	8		
pn-20	100	93	73	45	22	11		
pn-50	100	93	73	45	22	11		
pn-100	100	93	73	45	22	11		
Permute	100	100	100	87	70	50	32	16
DCT	100	100	100	91	73	48	31	15

Figure 5.9

TC_8 - Percentage of Cache Space Usage



	4L	8L	16L	32L	64L	128L	256L	512L
bs-a	65	32	23	12	6	3		
bs-r	71	54	34	17	9	4		
bs-d	71	46	30	16	8	4		
pn-20	87	54	41	21	10	5		
pn-50	87	54	41	21	10	5		
pn-100	87	54	41	21	10	5		
Permute	100	100	97	60	50	29	16	8
DCT	100	100	92	74	50	30	16	8

Figure 5.10

5.4.1 Results of *TC_4* and *TC_8*

According to figures 5.9 and 5.10, there is one common feature among all test programs: the utilization of cache space is dropping while the cache capacity is expanding.

5.4.2 Analysis

This result is quite predictable since the more cache space is available the more space tends to be wasted, especially for very short programs like *bubblesort* and

primenumber. Although *Permute* and *DCT* are longer in terms of program span, at 512L there is clearly much more space available than required. According to the raw data (in the CD-ROM), there are two main effects that waste cache space. First, some cache lines were occupied by short traces (less than 4 and 8 instructions for *TC_4* and *TC_8,* respectively). Second, there are some cache lines that have never been occupied by any trace; this is worse when the number of TC lines increases. Both of them are inevitable because, for the former, there is no certainty of the width of an individual trace, one might contain only 1 instruction while another contains more instructions up to the maximum number (4 or 8 instructions) due to the fill-policy. Consequently, the mapping of a trace entirely depends on the address of its first instruction. Some cache lines, then, might be unused because it is unlikely that the cache line following the line occupied by the current trace will be the place for the succeeding trace.

5.5 Conclusion

The results show that the hit rate of the trace cache tends to increase while the trace cache size is increasing from both the bigger number of instructions in each line and the larger number of trace cache lines. However, the increasing rate will come to saturate once the number of trace cache lines can cover all of the instructions of the test program.

In short test programs, the results show that the hit rate of the trace cache in *TC_8* is increasing and becoming steady earlier comparing to *TC_4*. It shows that the larger cache space allows more instructions to fit in and, certainly, the probability of finding the right instructions is increasing. Meanwhile, the hit rate also saturates faster because of the larger cache space, which can cover all of the instructions in fewer lines. On long programs, the results show that extending cache capacity both by increasing the number of trace cache lines and increasing the line width increases the performance of the trace cache as well.

Increasing the number of TC lines might improve performance of the trace cache but also leads to trace cache space waste because of the escalation of unused trace cache lines accoding to the direct-mapped scheme.

The wider trace on *TC_8* also increases the performance of the trace cache over *TC_4* if the comparison has been made at the same number of TC lines. But, if we compares at the same capacity, for example, 16L of *TC_4* and 8L of *TC_8* for the capacity of 64 instructions, the performance of both is similar or most of the time *TC_4* performs better than *TC_8*.

Chapter 6
Conclusion

6.1 Summary

This research has been conducted to study the performance of trace cache memory on a small-scale superscalar microprocessor. The superscalar DLX machine [9] was chosen as the basis for the experiment. The original model can process two instructions simultaneously with the help of a very small instruction cache to supply instructions. The trace cache memory was designed with less complexity than the previous works [20], [23]. There is no sophisticated branch prediction unit for packing the instruction traces and the number of instructions in one trace cache line was reduced to balance with the issue width of the processor. The experiment has been performed on 2 main configurations: *TC_4* and *TC_8*, which are 4 instruction-wide and 8 instruction-wide trace cache, respectively. Each configuration has a number of cache lines varied from 4 to 512 lines. Test programs used in this experiment can be categorized into 2 groups: the short ones (e.g. *bubblesort* and *primenumber*) and the longer ones (e.g. *permute* and *DCT*).

6.2 Conclusions

The experiment shows that the crucial parameters affecting the performance of the trace cache are the number of TC lines and the width of trace. An increment of both parameters leads to better performance of the trace cache indicated by the increasing number of hits. However, increasing the number of TC lines also causes more unused cache space according to the nature of the cache scheme.

The performance of *TC_8* is generally better than *TC_4* if the comparison has been made at the same number of TC lines but if we compare them at the same cache capacity, *TC_8* does not really outperform *TC_4* and most of the time performance of the former is lower than that of the latter.

Apart from those parameters that affect the performance of the trace cache, the policy of the fill-unit and also the logic unit for transferring traces from the fill-buffer to the trace cache memory are also crucial. Investigation of the effect of these policies is a matter for future work. From these results, the trace cache is not able to demonstrate clear advantage over the instruction cache as expected. On the other hand, even this trace cache model without sophisticated fill strategies is quite complex comparing with the original instruction cache. In that case, we found no evidence that it is worthwhile to invest the chip area to implement such model while a simple instruction cache works quite well for narrow-issue processors.

6.3 Further Work

From the results of these experiments, we gain some insight into characteristics of the designed trace cache on a narrow-issue processor and also some indications of pitfalls of the model. This section is a discussion of these drawbacks, which were not resolved in this research because of the time limitations.

The results show that the number of TC lines and the width of traces are crucial parameters in the aspect of trace cache performance. However, there is another parameter that is also vital but was paid less attention. It is the functional unit for transferring traces from the fill-buffer to trace cache memory. Some trace cache lines are not as useful as they should be and cause a significant number of misses. This is because the strategies to put a trace into the destination TC line were not as effective in avoiding misses as they could have been. Therefore, this unit should be investigated further to find the optimum strategies. Clearly, this unit involves significant complexity and adds to the implementation cost of the trace cache.

The advantage of a trace cache is the ability to contain two or more non-consecutive basic-blocks, which an instruction cache cannot. The best metric we have to evaluate the usefulness of the trace cache is *TC Line Content Hit* in which the hit counts indicate the possibility of taking advantage of the trace cache. This feature

should have been explicitly gathered for the purpose of trace cache analysis. Yet, it was not implemented in the design because of the complexity of modifying the existing DLX model to collect this data.

In this experiment, there are only two more additional test programs, *permute* and *DCT*, apart from the originally provided programs, *bubblesort* and *primenumber*, for the simulation. They can be categorized as long programs and short programs according to the span of the particular sourcecode. The results and analysis would be more reliable if there were more long programs simulated. However, there are two main constraints that obstruct us for gathering more test programs. The first one is converting the sourcecode of the prospective test programs to binary code is a time-consuming process because of the hand conversion of the source code described in chapter 4. The second constraint is the simulation for the long programs takes a very long time.

Finally, the experiment reveals that the other parameter that should taken into account to gain more insight into the trace cache is the strategies used to decide whether to hold the an existing trace or to replace it with a new coming trace that mapped at the same TC line. A more intelligent scheme would improve performance of the trace cache because it can hold the useful trace and ignore the less useful one at the right time. However, investigation of this feature needs more time and certainly would increase the complexity of the trace cache. Therefore, it is not included in this research.

Appendix A
Companion CD-ROM Contents

This CD contains essential materials that can be used to reproduce the simulation and the results created from our simulation for referencing purposes. At root directory, there are three subdirectories: *DLX Sourcecode*, *Test Programs*, and *Simulation log files*.

A.1 DLX Sourcecode

This subdirectory contains sets of VHDL sourcecode of the Superscalar DLX processor model categorized by processor configurations. Each set of VHDL sourcecode comprises four files: *Dlx.vhd*, *DlxPackage.vhd*, *Environment.vhd*, and *Testbench.vhd*.

Dlx.vhd is the main VHDL file that describes the architecture of the DLX processor. Most of the trace cache code is in this file.

DlxPackage.vhd is the package file that contains types, subtypes, constants, and functions in which they are used along with *Dlx.vhd*. This file also includes some code for the trace cache.

Environment.vhd creates the environment for the simulation. It describes how the processor model interfaces with the outside world and the system organization including the memory configuration. Originally, the memory capacity was 16Kbyte, which was not enough to run *Permute* and *DCT*. Therefore, this file was modified to increase the memory capacity to 32Kbyte.

Testbench.vhd connects all files together to make the simulation possible. This file is the only original file that was not modified for the trace cache.

To run each simulation configuration, we have to compile *DlxPackage.vhd* first followed by *Dlx.out*, *Environment.vhd*, and *Testbench.vhd*.

A.2 Test Programs

There are two subdirectories: *out files* and *Test programs sourcecode*. The *out files* subdirectory contains *.out* files for use as test programs. To use these files, we have to change the filename of the desired one into *dlx.out* and place it into the same directory as the desired DLX VHDL code. For example, if we want to run *DCT* in the simulation of *TC_4* at 8L, change from *dct.out* to *dlx.out* and put it into directory *tc4_8l*. When the simulation is halted, it will create a *result.log* file of *DCT* for the chosen configuration.

In *Test programs sourcecode*, there are assembly files of all test programs. *Bubblesort* and *Primenumber* are the original assembly sourcecode (*bs-r* and *pn-20*) and the manually modified assembly sourcecode (*bs-a*, *bs-d*, *pn-50*, and *pn-100*). *Permute* and *DCT* are the ones created by *GNU-dlxcc* of *permute.c* and *dct.c* from *C Sourcecode* subdirectory and patched as described in 4.4, which is included in that subdirectory. All assembly files are in *Assembly Sourcecode* subdirectory.

A.3 Simulation log files

This directory contains results created by each TC configuration as *.log* files. An individual filename was changed from *result.log* created from the simulation after the name of the test program. Each file contains information as follows:

- Log file banner – identifies the trace cache configuration and the name of the test program used in the simulation.

Example:

```
************************************************
TC_4 :   4 Lines       *Test Program: bs-a.out
************************************************
```

- General Information – this section provides information about the number of instructions that have been fetched into the processor (*Total Fetched Instructions*), committed by the Commit Unit (*Committed Instructions*),

rejected (*Omitted Instructions*), and the number of instructions that have been accessed from the instruction cache (*Cache Memory Access (fetch)*).

- Instruction-Cache Info – indicates how many instruction cache hits there are in the simulation of the test program and the percentage of hits by the total instruction cache accesses.
- Trace-Cache Info – this section shows how many trace cache hits (including *TC-First Tag Hit* and *TC-Content Hit*) and misses (including *Compulsory Misses* and *Conflict Misses*) there are in the simulation of the test program and the percentage of hits and misses.
- Information Collected From Individual Trace Cache Line – this table is the information gathered from each TC line and contains the following items:
    - Line – the trace cache line number,
    - Comp-Miss – the number of compulsory misses on a TC line,
    - Conf-Miss – the number of conflict misses on a TC line,
    - TC-Write – the number of traces written on a TC line,
    - TC-O_Write – the number of traces that were overwritten with a different trace content on a TC line,
    - TC-Size – the longest trace size existing on a TC line,
    - TC-Hit – the number of hits on a TC line,
    - FTag-Hit – the number of *TC-First Tag Hit* on a TC line, and
    - Cont-Hit – the number of *TC-Content Hit* on a TC line.
- Percentage of Cache Space Usage – this section indicates the percentage of cache space that has been written by traces.

Appendix B
VHDL Code of Trace Cache

Three files have been modified from the original Superscalar DLX model. They are *Dlx.vhd*, *DlxPackage.vhd*, and *Environment.vhd*.

B.1 Dlx.vhd

This file describes the architecture of the trace cache. The first part is signal declaration.

```
-- Fill Buffer Structure
signal FB_InstrBuffer : TypeArrayInstr( 0 to cInstrRow, 0 to cInstrSlot );
signal FB_InstrAddrBuffer : TypeArrayInstr( 0 to cInstrRow, 0 to cInstrSlot );

signal FB_BufferReady : unsigned( 0 to cInstrRow);
signal FB_TraceSize : TypeArraySlotCount( 0 to cInstrRow);
signal FB_BranchExisting : unsigned( 0 to cInstrRow);
signal FB_BranchSlot : TypeArraySlot( 0 to cInstrRow);

-- Signal to inform trace line counter when there are 2 instructions sit in the line simultaneously
signal FB_TraceCount2up : bit;

-- Buffer line termination signals
signal FB_RowTerminatedByA : bit;
signal FB_RowTerminatedByB : bit;

signal FB_FinishRowNumber_A : TypeRow;
signal FB_FinishRowNumber_B : TypeRow;

signal FB_BranchInstrA_Row : TypeRow;
signal FB_BranchInstrB_Row : TypeRow;

-- Instruction Write Enable
signal FB_InstrAWrite : bit;
signal FB_InstrBWrite : bit;
signal FB_InstrWrite_A_B : unsigned( 0 to 1 );

-- Buffer Index suite: Instruction A Index, Instruction B Index, and Reference Index (current index)
signal FB_InstrA_Row : TypeRow:=0;
signal FB_InstrA_Slot : TypeSlot:=0;
signal FB_InstrB_Row : TypeRow:=0;
signal FB_InstrB_Slot : TypeSlot:=0;
signal FB_CurrentRow : TypeRow:=0;
signal FB_CurrentSlot : TypeSlot:=0;

-- Instruction Type Flags
signal FB_InstrA_IsBranch : bit := '0';
signal FB_InstrB_IsBranch : bit := '0';
signal FB_InstrA_IsDelimiter : bit := '0';
signal FB_InstrB_IsDelimiter : bit := '0';

-- For Experiment
signal FB_LastInstr : TypeWord;
signal FB_LastInstrShift : TypeWord;
signal FB_LastInstrIsBranch : bit:='0';

signal FB_Test :bit:='0';
```

```
                        '1' when Clock = '0' and FB_InstrAWrite = '1' and
                                 ( FB_InstrA_IsDelimiter = '1' or
                                   FB_CurrentSlot = cInstrSlot or
                                   ( FB_InstrA_IsBranch = '1' and FB_BranchExisting(FB_CurrentRow) = '1' )
) else
                        '1' when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                 FB_InstrA_IsBranch = '1' and
                                 FB_InstrB_IsBranch = '1' else
                        '0';

        FB_RowTerminatedByB <=

                        -- Test
                        '0' when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                 FB_CurrentSlot /= 0 and
                                 (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                 (FB_LastInstrIsBranch='0') else

                        '1' when Clock = '0' and FB_InstrWrite_A_B = "01" and
                                 FB_CurrentSlot /= 0 and
                                 (Equal(IF_InstrAddrRegB_Input,FB_LastInstrShift)='0') and
                                 (FB_LastInstrIsBranch='0') else

                        -- When only instruction B is coming
                        '1' when Clock = '0' and FB_InstrWrite_A_B = "01" and
                                 ( FB_InstrB_IsDelimiter = '1' or
                                   FB_CurrentSlot = cInstrSlot or
                                   ( FB_InstrB_IsBranch = '1' and FB_BranchExisting(FB_CurrentRow) = '1' )
) else

                        -- When both instructions are coming
                        '0' when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                 FB_InstrA_IsBranch = '1' and
                                 FB_BranchExisting(FB_CurrentRow) = '1' and
                                 FB_InstrB_IsBranch = '0' and
                                 FB_InstrB_IsDelimiter = '0' else

                        '0' when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                 FB_InstrA_IsBranch = '1' and
                                 FB_InstrB_IsBranch = '1' and
                                 FB_BranchExisting(FB_CurrentRow) = '0' else

                        '1' when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                 FB_InstrA_IsBranch = '1' and
                                 FB_InstrB_IsBranch = '1' and
                                 FB_BranchExisting(FB_CurrentRow) = '1' else

                        '1' when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                 ( FB_InstrB_IsDelimiter = '1' or
                                   FB_CurrentSlot = cInstrSlot-1 or
                                   ( FB_InstrA_IsBranch = '0' and
                                     FB_InstrA_IsDelimiter = '0' and
                                     FB_InstrB_IsBranch = '1' and
                                     FB_BranchExisting(FB_CurrentRow) = '1' ) ) else
                        '0';

        -- Identify which buffer line(s) is(are) terminated.
        FB_FinishRowNumber_A <=

                        -- Test
                        FB_CurrentRow when Clock = '0' and FB_InstrAWrite = '1' and
                                           FB_CurrentSlot /= 0 and
                                           (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                           (FB_LastInstrIsBranch='0') else

                        FB_CurrentRow when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                           FB_InstrA_IsBranch = '1' and
                                           FB_BranchExisting(FB_CurrentRow) = '1' and
                                           FB_InstrB_IsBranch = '0' and
                                           FB_InstrB_IsDelimiter = '0' else

                        FB_InstrA_Row when Clock = '0' and FB_InstrAWrite = '1' and
                                           ( FB_InstrA_IsDelimiter = '1' or
                                             ( FB_CurrentSlot = cInstrSlot and
                                               FB_InstrA_IsBranch = '0' ) or
                                             ( FB_CurrentSlot = cInstrSlot and
                                               FB_InstrA_IsBranch = '1' and
                                               FB_BranchExisting(FB_CurrentRow) = '0' ) ) else

                        FB_CurrentRow when Clock = '0' and FB_InstrAWrite = '1' and
                                           ( FB_InstrA_IsBranch = '1' and
FB_BranchExisting(FB_CurrentRow) = '1' ) else

                        FB_InstrA_Row when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                           FB_InstrA_IsBranch = '1' and
                                           FB_InstrB_IsBranch = '1' else

                        unaffected;
         FB_FinishRowNumber_B <=

                        -- Test
                        FB_CurrentRow when Clock = '0' and FB_InstrWrite_A_B = "01" and
                                           FB_CurrentSlot /= 0 and
                                           (Equal(IF_InstrAddrRegB_Input,FB_LastInstrShift)='0') and
                                           (FB_LastInstrIsBranch='0') else

                        -- When only instruction B is coming
                        FB_InstrB_Row when Clock = '0' and FB_InstrWrite_A_B = "01" and
                                           ( FB_InstrB_IsDelimiter = '1' or
                                             ( FB_CurrentSlot = cInstrSlot and
                                               FB_InstrB_IsBranch = '0' ) ) else
```

```
                            FB_CurrentRow when Clock = '0' and FB_InstrWrite_A_B = "01" and
                                                ( FB_InstrB_IsBranch = '1' and
FB_BranchExisting(FB_CurrentRow) = '1' ) else

                            -- When both instructions are coming
                            FB_InstrA_Row when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                               FB_InstrA_IsBranch = '0' and
                                               FB_InstrA_IsDelimiter = '0' and
                                               FB_InstrB_IsBranch = '1' and
                                               FB_BranchExisting(FB_CurrentRow) = '1' else

                            FB_InstrA_Row when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                               FB_InstrA_IsBranch = '1' and
                                               FB_InstrB_IsBranch = '1' and
                                               FB_BranchExisting(FB_CurrentRow) = '1' else

                            FB_InstrB_Row when Clock = '0' and FB_InstrWrite_A_B = "11" and
                                               ( FB_InstrB_IsDelimiter = '1' or
                                                 FB_CurrentSlot = cInstrSlot-1 or
                                                 ( FB_InstrA_IsBranch = '0' and
                                                   FB_InstrA_IsDelimiter = '0' and
                                                   FB_InstrB_IsBranch = '1' and
                                                   FB_BranchExisting(FB_CurrentRow) = '1' ) ) else
                            unaffected;

        -- Logic for informing when there are 2 instructions sitting in the same line
        -- ( for trace line counter mechanism )
        FB_TraceCount2Up <= '1' when Clock = '0' and Clock'event and
                                     FB_InstrWrite_A_B = "11" and
                                     FB_CurrentSlot <= cInstrSlot-1 and
                                     ( ( FB_InstrA_IsBranch = '0' and
                                         FB_InstrB_IsBranch = '0' and
                                         FB_InstrA_IsDelimiter = '0' ) or
                                       ( ( ( FB_BranchExisting(FB_CurrentRow) = '0' and
                                             ( ( FB_InstrA_IsBranch = '1' and
                                                 FB_InstrB_IsBranch = '0' ) or
                                               ( FB_InstrA_IsBranch = '0' and
                                                 FB_InstrA_IsDelimiter = '0' and
                                                 FB_InstrB_IsBranch = '1' ) ) ) or
                                           ( FB_BranchExisting(FB_CurrentRow) = '1' and
                                             ( FB_InstrA_IsBranch = '1' and
                                               FB_InstrB_IsDelimiter = '0' and
                                               FB_InstrB_IsBranch = '0' ) ) ) ) ) else
                               '0';

        -- Determining whether incoming branch would sit in the current row or next possible row
        FB_BranchInstrA_Row <=
                            0 when ( Clock = '0' and Clock'event ) and
                                   ( FB_InstrA_IsBranch = '1' and FB_BranchExisting( FB_CurrentRow ) = '1' ) and
                                     FB_CurrentRow = cInstrRow else
                            FB_CurrentRow + 1 when          ( Clock = '0' and Clock'event ) and
                                                            ( FB_InstrA_IsBranch = '1' and FB_BranchExisting(
FB_CurrentRow ) = '1' ) and
                                                              FB_CurrentRow < cInstrRow else
                            FB_CurrentRow;

        FB_BranchInstrB_Row <=
                            -- When only instruction B is coming
                            0 when ( Clock = '0' and Clock'event ) and
                                   FB_InstrWrite_A_B = "01" and
                                   ( FB_InstrB_IsBranch = '1' and FB_BranchExisting( FB_CurrentRow ) = '1' ) and
                                     FB_CurrentRow = cInstrRow else
                            FB_CurrentRow + 1 when          ( Clock = '0' and Clock'event ) and
                                                            FB_InstrWrite_A_B = "01" and
                                                            ( FB_InstrB_IsBranch = '1' and FB_BranchExisting(
FB_CurrentRow ) = '1' ) and
                                                              FB_CurrentRow < cInstrRow else
                            -- When both instructions are coming
                            0 when    ( Clock = '0' and Clock'event ) and
                                      FB_InstrWrite_A_B = "11" and
                                      ( ( FB_CurrentRow = cInstrRow and
                                          ( ( FB_InstrA_IsBranch = '0' and
                                              FB_InstrA_IsDelimiter = '0' and
                                              FB_InstrB_IsBranch = '1' and
                                              FB_CurrentSlot = cInstrSlot ) or
                                            ( FB_InstrA_IsBranch = '0' and
                                              FB_InstrA_IsDelimiter = '0' and
                                              FB_InstrB_IsBranch = '1' and
                                              FB_BranchExisting(FB_CurrentRow) = '1' ) or
                                            ( FB_InstrA_IsDelimiter = '1' and
                                              FB_InstrB_IsBranch = '1' ) or
                                            ( FB_InstrA_IsBranch = '1' and
                                              FB_InstrB_IsBranch = '1' and
                                              FB_BranchExisting(FB_CurrentRow) = '1' ) ) ) or
                                        ( FB_CurrentRow = cInstrRow-1 and
                                          FB_InstrA_IsBranch = '1' and
                                          FB_InstrB_IsBranch = '1' and
                                          FB_BranchExisting(FB_CurrentRow) = '1' ) ) else

                            FB_CurrentRow + 1 when          ( Clock = '0' and Clock'event ) and
                                                            FB_InstrWrite_A_B = "11" and
                                                            FB_CurrentRow < cInstrRow and
                                                            ( ( FB_InstrA_IsBranch = '0' and
                                                                FB_InstrA_IsDelimiter = '0' and
                                                                FB_InstrB_IsBranch = '1' and
                                                                ( FB_CurrentSlot = cInstrSlot or
                                                                  ( FB_BranchExisting(FB_CurrentRow) = '1' ) ) ) or
                                                              ( FB_InstrA_IsDelimiter = '1' and
                                                                FB_InstrB_IsBranch = '1' ) or
                                                                ( FB_InstrA_IsBranch = '1' and
                                                            FB_InstrB_IsBranch = '1' and
                                                            FB_BranchExisting(FB_CurrentRow) = '0' ) ) else
```

```
                        FB_Currentrow + 2 when          ( Clock = '0' and Clock'event ) and
                                                        FB_InstrWrite_A_B = "11" and
                                                        FB_CurrentRow < cInstrRow-1 and
                                                        FB_InstrA_IsBranch = '1' and
                                                        FB_InstrB_IsBranch = '1' and
                                                        FB_BranchExisting(FB_CurrentRow) = '1' else

                        FB_CurrentRow;


        -- Index for instruction A
        FB_InstrA_Row <=

                        -- Test
                        0 when ( Clock = '0' and Clock'event ) and
                                  FB_InstrAWrite = '1' and
                                  FB_CurrentRow = cInstrRow and
                                  FB_CurrentSlot /= 0 and
                                  (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                        FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                             FB_InstrAWrite = '1' and
                                             FB_CurrentRow < cInstrRow and
                                             FB_CurrentSlot /= 0 and
                                             (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                             (FB_LastInstrIsBranch='0') else

                        0 when ( Clock = '0' and Clock'event ) and
                                   FB_InstrA_IsBranch = '1' and
                                 ( FB_BranchExisting(FB_CurrentRow) = '1' and FB_CurrentRow = cInstrRow )
and
                                 ( FB_InstrWrite_A_B = "10" or FB_InstrWrite_A_B = "11" ) else
                        FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                               FB_InstrA_IsBranch = '1' and
                                             ( FB_BranchExisting(FB_CurrentRow) = '1' and FB_CurrentRow
< cInstrRow ) and
                                             ( FB_InstrWrite_A_B = "10" or FB_InstrWrite_A_B = "11" )
else
                        FB_CurrentRow when ( Clock = '0' and Clock'event ) and
                                           ( FB_InstrWrite_A_B = "10" or FB_InstrWrite_A_B = "11" ) else
                        unaffected;
        FB_InstrA_Slot <=

                         -- Test
                         0 when ( Clock = '0' and Clock'event ) and
                                  FB_InstrAWrite = '1' and
                                  FB_CurrentSlot /= 0 and
                                  (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                         0 when ( Clock = '0' and Clock'event ) and
                                    FB_InstrA_IsBranch = '1' and
                                  ( FB_InstrWrite_A_B = "10" or FB_InstrWrite_A_B = "11" ) and
                                  ( FB_BranchExisting(FB_CurrentRow) = '1' ) else
                         FB_CurrentSlot when ( Clock = '0' and Clock'event ) and
                                             ( FB_InstrWrite_A_B = "10" or FB_InstrWrite_A_B = "11" )
else
                         unaffected;

        -- Index for instruction B
        FB_InstrB_Row <=

                        -- Test
                        0 when ( Clock = '0' and Clock'event ) and
                                  FB_InstrWrite_A_B = "11" and
                                  FB_CurrentRow = cInstrRow and
                                  FB_CurrentSlot /= 0 and
                                  (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                         FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                              FB_InstrWrite_A_B = "11" and
                                              FB_CurrentRow < cInstrRow and
                                              FB_CurrentSlot /= 0 and
                                              (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                              (FB_LastInstrIsBranch='0') else


                        0 when ( Clock = '0' and Clock'event ) and
                                  FB_InstrWrite_A_B = "01" and
                                  FB_CurrentSlot /= 0 and
                                  FB_CurrentRow = cInstrRow and
                                  (Equal(IF_InstrAddrRegB_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                        FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                             FB_InstrWrite_A_B = "01" and
                                             FB_CurrentSlot /= 0 and
                                             FB_CurrentRow < cInstrRow and
                                             (Equal(IF_InstrAddrRegB_Input,FB_LastInstrShift)='0') and
                                             (FB_LastInstrIsBranch='0') else

                        0 when ( Clock = '0' and Clock'event ) and
                                  FB_InstrAWrite = '1' and
                                  FB_CurrentSlot /= 0 and
                                  FB_CurrentRow = cInstrRow and
                                  (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                         FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
```

```
                                        FB_InstrAWrite = '1' and
                                        FB_CurrentSlot /= 0 and
                                        FB_CurrentRow < cInstrRow and
                                        (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                        (FB_LastInstrIsBranch='0') else

                                -- Current row has branch and Both instructions are coming
                                1 when     ( Clock = '0' and Clock'event ) and
                                          FB_InstrWrite_A_B = "11" and
                                          FB_BranchExisting(FB_CurrentRow) = '1' and
                                          FB_CurrentRow = cInstrRow and
                                          FB_InstrA_IsBranch = '1' and
                                          FB_InstrB_IsBranch = '1' else

                                FB_CurrentRow+2 when ( Clock = '0' and Clock'event ) and
                                                       FB_InstrWrite_A_B = "11" and
                                                       FB_BranchExisting(FB_CurrentRow) = '1' and
                                                       FB_CurrentRow < cInstrRow-1 and
                                                       FB_InstrA_IsBranch = '1' and
                                                       FB_InstrB_IsBranch = '1' else

                                0 when     ( Clock = '0' and Clock'event ) and
                                          ( FB_InstrWrite_A_B = "11" and
                                            FB_BranchExisting(FB_CurrentRow) = '1' ) and
                                          ( ( FB_CurrentRow = cInstrRow and
                                            ( ( FB_InstrA_IsBranch = '1' and FB_InstrB_IsBranch = '0' ) or
                                              ( FB_InstrA_IsBranch = '0' and FB_InstrB_IsBranch = '1' ) ) ) or
                                            ( FB_CurrentRow = cInstrRow-1 and
                                              FB_InstrA_IsBranch = '1' and
                                              FB_InstrB_IsBranch = '1' ) ) else

                                FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                                       ( FB_InstrWrite_A_B = "11" and
                                                         FB_BranchExisting(FB_CurrentRow) = '1' ) and
                                                       ( FB_CurrentRow < cInstrRow and
                                                         ( ( FB_InstrA_IsBranch = '1' and FB_InstrB_IsBranch = '0'
) or
                                                             ( FB_InstrA_IsBranch = '0' and FB_InstrB_IsBranch =
'1' ) ) ) else

                                -- Current row has NO branch and Both instructions are coming
                                0 when     ( Clock = '0' and Clock'event ) and
                                          ( FB_InstrWrite_A_B = "11" and
                                            FB_BranchExisting(FB_CurrentRow) = '0' and
                                            FB_CurrentRow = cInstrRow ) and
                                          ( ( FB_CurrentSlot = cInstrSlot and
                                              ( ( FB_InstrA_IsBranch = '1' and FB_InstrB_IsBranch = '0' ) or
                                                ( FB_InstrA_IsBranch = '0' and FB_InstrB_IsBranch = '1' ) ) ) or
                                            ( FB_InstrA_IsBranch = '1' and
                                              FB_InstrB_IsBranch = '1' ) ) else

                                FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                                       ( FB_InstrWrite_A_B = "11" and
                                                         FB_BranchExisting(FB_CurrentRow) = '0' and
                                                         FB_CurrentRow < cInstrRow ) and
                                                       ( ( FB_CurrentSlot = cInstrSlot and
                                                             ( ( FB_InstrA_IsBranch = '1' and FB_InstrB_IsBranch =
'0' ) or
                                                               ( FB_InstrA_IsBranch = '0' and FB_InstrB_IsBranch =
'1' ) ) ) or
                                                           ( FB_InstrA_IsBranch = '1' and
                                                             FB_InstrB_IsBranch = '1' ) ) else

                                -- Current row has branch and only instruction B is coming and it is a branch
                                0 when     ( Clock = '0' and Clock'event ) and
                                           FB_InstrWrite_A_B = "01" and
                                           FB_BranchExisting(FB_CurrentRow) = '1' and
                                           FB_InstrB_IsBranch = '1' and
                                           FB_Currentrow = cInstrRow else

                                FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                                       FB_InstrWrite_A_B = "01" and
                                                       FB_BranchExisting(FB_CurrentRow) = '1' and
                                                       FB_InstrB_IsBranch = '1' and
                                                       FB_Currentrow < cInstrRow else

                                -- In case of Instruction A is a delimiter instruction (Jumps,Trap,RFE)

                                0 when ( Clock = '0' and Clock'event ) and
                                          FB_InstrWrite_A_B = "11" and
                                          FB_CurrentRow = cInstrRow and
                                          FB_InstrA_IsDelimiter = '1' else
                                FB_CurrentRow+1 when ( Clock = '0' and clock'event ) and
                                                       FB_InstrWrite_A_B = "11" and
                                                       FB_CurrentRow < cInstrRow and
                                                       FB_InstrA_IsDelimiter = '1' else

                                -- Original cases
                                FB_CurrentRow when ( Clock = '0' and Clock'event ) and FB_InstrWrite_A_B = "01" else
                                0 when ( Clock = '0' and Clock'event ) and
                                       ( FB_InstrWrite_A_B = "11" and FB_CurrentRow = cInstrRow and FB_CurrentSlot =
cInstrSlot ) else
                                FB_CurrentRow+1 when ( Clock = '0' and Clock'event ) and
                                                       ( FB_InstrWrite_A_B = "11" and FB_CurrentSlot = cInstrSlot )
else
                                FB_CurrentRow when ( Clock = '0' and Clock'event ) and FB_InstrWrite_A_B = "11" else
                                unaffected;

        FB_InstrB_Slot <=

                                -- Test
                                0 when ( Clock = '0' and Clock'event ) and
                                          FB_InstrWrite_A_B = "01" and
```

```
                                    FB_InstrWrite_A_B = "11" and
                                    FB_BranchExisting(FB_CurrentRow) = '1' and
                                    FB_currentRow < cInstrRow and
                                    ( ( FB_InstrA_IsBranch = '1' and
                                        FB_InstrB_IsBranch = '0' and
                                        FB_InstrB_IsDelimiter = '0' ) or
                                      ( FB_InstrA_IsBranch = '0' and
                                        FB_InstrA_IsDelimiter = '0' and
                                        FB_InstrB_IsBranch = '1' ) or
                                      ( FB_InstrA_IsDelimiter = '1' and
                                        FB_InstrB_IsBranch = '1' ) ) else

1 when     ( Clock = '1' and Clock'event ) and
           FB_InstrWrite_A_B = "11" and
           FB_BranchExisting(FB_CurrentRow) = '1' and
           FB_currentRow = cInstrRow and
           ( ( FB_InstrA_IsBranch = '1' and
               FB_InstrB_IsDelimiter = '1' ) or
             ( FB_InstrA_IsBranch = '1' and
               FB_InstrB_IsBranch = '1' ) ) else

0 when     ( Clock = '1' and Clock'event ) and
           FB_InstrWrite_A_B = "11" and
           FB_BranchExisting(FB_CurrentRow) = '1' and
           ( ( FB_CurrentRow = cInstrRow and
               ( ( FB_InstrA_IsBranch = '1' and
                   FB_InstrB_IsBranch = '0' and
                   FB_InstrB_IsDelimiter = '0' ) or
                 ( FB_InstrA_IsBranch = '0' and
                   FB_InstrA_IsDelimiter = '0' and
                   FB_InstrB_IsBranch = '1' ) or
                 ( FB_InstrA_IsDelimiter = '1' and
                   FB_InstrB_IsBranch = '1' ) ) ) or
             ( FB_CurrentRow = cInstrRow-1 and
               ( ( FB_InstrA_IsBranch = '1' and
                   FB_InstrB_IsDelimiter = '1' ) or
                 ( FB_InstrA_IsBranch = '1' and
                   FB_InstrB_IsBranch = '1' ) ) ) ) else

-- Current row has NO branch and both instructions is coming
1 when     ( Clock = '1' and Clock'event ) and
           FB_InstrWrite_A_B = "11" and
           FB_BranchExisting(FB_CurrentRow) = '0' and
           FB_CurrentSlot = cInstrSlot and
           FB_currentRow = cInstrRow and
           FB_InstrA_IsBranch = '1' and
           FB_InstrB_IsDelimiter = '1' else

0 when     ( Clock = '1' and Clock'event ) and
           FB_InstrWrite_A_B = "11" and
           FB_BranchExisting(FB_CurrentRow) = '0' and
           FB_CurrentSlot = cInstrSlot and
           ( ( FB_CurrentRow = cInstrRow and
               ( ( FB_InstrA_IsBranch = '1' and
                   FB_InstrB_IsBranch = '0' and
                   FB_InstrB_IsDelimiter = '0' ) or
                 ( FB_InstrA_IsBranch = '0' and
                   FB_InstrA_IsDelimiter = '0' and
                   FB_InstrB_IsBranch = '1' ) or
                 ( FB_InstrA_IsDelimiter = '1' and
                   FB_InstrB_IsBranch = '1' ) ) ) or
             ( FB_CurrentRow = cInstrRow-1 and
               FB_InstrA_IsDelimiter = '1' and
               FB_InstrB_IsDelimiter = '1' ) ) else

FB_CurrentRow+1 when ( Clock = '1' and Clock'event ) and
                       FB_InstrWrite_A_B = "11" and
                       FB_CurrentRow <= cInstrRow-1 and
                       FB_InstrA_IsBranch = '1' and
                       FB_InstrB_IsBranch = '1' else

FB_CurrentRow+1 when ( Clock = '1' and Clock'event ) and
                       FB_InstrWrite_A_B = "11" and
                       FB_BranchExisting(FB_CurrentRow) = '0' and
                       FB_CurrentSlot = cInstrSlot and
                       ( -- ( FB_CurrentRow <= cInstrRow-1 and
                         --   FB_InstrA_IsBranch = '1' and
                         --   FB_InstrB_IsBranch = '1' ) or
                         ( FB_CurrentRow < cInstrRow and
                           ( ( FB_InstrA_IsBranch = '1' and
                               FB_InstrB_IsBranch = '0' and
                               FB_InstrB_IsDelimiter = '0' ) or
                             ( FB_InstrA_IsBranch = '0' and
                               FB_InstrA_IsDelimiter = '0' and
                               FB_InstrB_IsBranch = '1' ) ) ) or
                         ( FB_CurrentRow = cInstrRow-1 and
                           FB_InstrA_IsDelimiter = '1' and
                           FB_InstrB_IsBranch = '1' ) ) else

-- Current row has a branch and instruction A is coming (it is a branch)
0 when     ( Clock = '1' and Clock'event ) and
           FB_InstrWrite_A_B = "10" and
           FB_BranchExisting(FB_CurrentRow) = '1' and
           FB_CurrentRow = cInstrRow and
           FB_InstrA_IsBranch = '1' else

FB_CurrentRow+1 when ( Clock = '1' and Clock'event ) and
                       FB_InstrWrite_A_B = "10" and
                       FB_BranchExisting(FB_CurrentRow) = '1' and
                       FB_CurrentRow < cInstrRow and
                       FB_InstrA_IsBranch = '1' else
```

```
                        -- Current row has a branch and instruction B is coming (it is a branch)
                        0 when     ( Clock = '1' and Clock'event ) and
                                   FB_InstrWrite_A_B = "01" and
                                   FB_BranchExisting(FB_CurrentRow) = '1' and
                                   FB_CurrentRow = cInstrRow and
                                   FB_InstrB_IsBranch = '1' else

                        FB_CurrentRow+1 when ( Clock = '1' and Clock'event ) and
                                             FB_InstrWrite_A_B = "01" and
                                             FB_BranchExisting(FB_CurrentRow) = '1' and
                                             FB_CurrentRow < cInstrRow and
                                             FB_InstrB_IsBranch = '1' else

                        -- If Instruction A is delimiter instruction
                        FB_InstrA_Row+1 when ( Clock = '1' and Clock'event ) and
                                             FB_InstrWrite_A_B = "10" and
                                             FB_InstrA_IsDelimiter = '1' and
                                             FB_InstrA_Row < cInstrRow else
                        0 when ( Clock = '1' and clock'event ) and
                                  FB_InstrWrite_A_B = "10" and
                                  FB_InstrA_IsDelimiter = '1' and
                                  FB_InstrA_Row = cInstrRow else
                        FB_InstrB_Row when ( Clock = '1' and Clock'event ) and
                                             FB_InstrWrite_A_B = "11" and
                                             FB_InstrA_IsDelimiter = '1' and
                                             FB_InstrB_IsDelimiter = '0' else
                        -- If Instruction B is delimiter instruction
                        FB_InstrB_Row+1 when ( Clock = '1' and Clock'event ) and
                                             FB_InstrB_IsDelimiter = '1' and
                                             FB_InstrB_Row < cInstrRow else
                        0 when ( Clock = '1' and Clock'event ) and
                                  FB_InstrB_IsDelimiter = '1' and
                                  FB_InstrB_Row = cInstrRow else

                        -- Original cases
                          FB_CurrentRow+1 when ( Clock = '1' and Clock'event ) and
                                               ( ( FB_InstrWrite_A_B = "10" and
                                                   FB_InstrA_Row < cInstrRow and
                                                   FB_InstrA_Slot = cInstrSlot ) or
                                                 ( FB_InstrWrite_A_B = "01" and
                                                   FB_InstrB_Row < cInstrRow and
                                                   FB_InstrB_Slot = cInstrSlot ) or
                                                 ( FB_InstrWrite_A_B = "11" and
                                                   ( ( FB_InstrB_Row /= 0 and FB_InstrA_Slot =
cInstrSlot ) or
                                                     ( FB_InstrB_Row < cInstrRow and FB_InstrB_Slot =
cInstrSlot ) ) ) ) else
                            0 when ( Clock = '1' and Clock'event ) and
                                     ( ( FB_InstrWrite_A_B = "10" and
                                         FB_InstrA_Row = cInstrRow and
                                         FB_InstrA_Slot = cInstrSlot ) or
                                       ( FB_InstrWrite_A_B = "01" and
                                         FB_InstrB_Row = cInstrRow and
                                         FB_InstrB_Slot = cInstrSlot ) or
                                       ( FB_InstrWrite_A_B = "11" and
                                         ( ( FB_InstrB_Row = 0 and FB_InstrA_Slot = cInstrSlot ) or
                                           ( FB_InstrB_Row = cInstrRow and FB_InstrB_Slot = cInstrSlot ) ) ) )
else
                            unaffected;

                FB_CurrentSlot <=

                        -- Test
                        2 when ( Clock = '1' and Clock'event ) and
                                  FB_InstrWrite_A_B = "11" and
                                  FB_CurrentSlot /= 0 and
                                  (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else


                        1 when ( Clock = '1' and Clock'event ) and
                                  FB_InstrAWrite = '1' and
                                  FB_CurrentSlot /= 0 and
                                  (Equal(IF_InstrAddrRegA_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                        1 when ( Clock = '1' and Clock'event ) and
                                  FB_InstrWrite_A_B = "01" and
                                  FB_CurrentSlot /= 0 and
                                  (Equal(IF_InstrAddrRegB_Input,FB_LastInstrShift)='0') and
                                  (FB_LastInstrIsBranch='0') else

                        -- Current row has branch and both instructions are coming
                        2 when     ( Clock = '1' and Clock'event ) and
                                  FB_InstrWrite_A_B = "11" and
                                  FB_BranchExisting(FB_CurrentRow) = '1' and
                                  FB_InstrA_IsBranch = '1' and
                                  FB_InstrB_IsBranch = '0' and
                                  FB_InstrB_IsDelimiter = '0' else

                        0 when     ( Clock = '1' and Clock'event ) and
                                  FB_InstrWrite_A_B = "11" and
                                  FB_BranchExisting(FB_CurrentRow) = '1' and
                                  FB_InstrA_IsBranch = '1' and
                                  FB_InstrB_IsDelimiter = '1' else

                        1 when     ( Clock = '1' and Clock'event ) and
                                  FB_InstrWrite_A_B = "11" and
                                  FB_BranchExisting(FB_CurrentRow) = '1' and
                                  ( ( FB_InstrA_IsBranch = '0' and
                                      FB_InstrA_IsDelimiter = '0' and
                                      FB_InstrB_IsBranch = '1' ) or
```

```
                    ( FB_InstrA_IsBranch = '1' and
                      FB_InstrB_IsBranch = '0' ) or
                    ( FB_InstrA_IsBranch = '1' and
                      FB_InstrB_IsBranch = '1' ) ) else

1 when    ( Clock = '1' and Clock'event ) and
          FB_InstrWrite_A_B = "10" and
          FB_BranchExisting(FB_CurrentRow) = '1' and
          FB_InstrA_IsBranch = '1' else

1 when    ( Clock = '1' and Clock'event ) and
          FB_InstrWrite_A_B = "01" and
          FB_BranchExisting(FB_CurrentRow) = '1' and
          FB_InstrB_IsBranch = '1' else

-- Current row has NO branch and both instructions are coming
1 when    ( Clock = '1' and Clock'event ) and
          FB_InstrWrite_A_B = "11" and
          FB_BranchExisting(FB_CurrentRow) = '0' and
          ( ( FB_CurrentSlot = cInstrSlot and
              FB_InstrA_IsBranch = '0' and
              FB_InstrA_IsDelimiter = '0' and
              FB_InstrB_IsBranch = '1' ) or
            ( FB_InstrA_IsBranch = '1' and
              FB_InstrB_IsBranch = '1' ) ) else

0 when    ( Clock = '1' and Clock'event ) and
          FB_InstrWrite_A_B = "11" and
          FB_BranchExisting(FB_CurrentRow) = '0' and
          ( ( FB_CurrentSlot = cInstrSlot-1 and
              FB_InstrA_IsBranch = '0' and
              FB_InstrA_IsDelimiter = '0' and
              FB_InstrB_IsBranch = '1' ) or
            ( FB_InstrA_IsDelimiter = '1' and
              FB_InstrB_IsBranch = '1' ) ) else

FB_CurrentSlot+2 when ( Clock = '1' and Clock'event ) and
                      FB_InstrWrite_A_B = "11" and
                      FB_BranchExisting(FB_CurrentRow) = '0' and
                      FB_CurrentSlot < cInstrSlot-1 and
                      FB_InstrA_IsBranch = '0' and
                      FB_InstrA_IsDelimiter = '0' and
                      FB_InstrB_IsBranch = '1' else


-- In case of whether Instruction A and/or Instruction B is a delimiter instruction
0 when ( Clock = '1' and Clock'event ) and
       ( ( FB_InstrWrite_A_B = "10" and
           FB_InstrA_IsDelimiter = '1' ) or
           FB_InstrB_IsDelimiter = '1' ) else
FB_InstrB_Slot+1 when ( Clock = '1' and Clock'event ) and
                      FB_InstrWrite_A_B = "11" and
                      FB_InstrA_IsDelimiter = '1' and
                      FB_InstrB_IsDelimiter = '0' else


-- Original cases
  0 when ( Clock = '1' and Clock'event ) and
           ( ( FB_InstrWrite_A_B = "10" and FB_InstrA_Slot = cInstrSlot ) or
             ( FB_InstrWrite_A_B = "01" and FB_InstrB_Slot = cInstrSlot ) or
             ( FB_InstrWrite_A_B = "11" and FB_InstrB_Slot = cInstrSlot ) ) else
  1 when ( Clock = '1' and Clock'event ) and
           ( FB_InstrWrite_A_B = "11" and FB_InstrA_Slot = cInstrSlot and
FB_InstrB_Slot = 0 ) else

  FB_CurrentSlot+1 when ( Clock = '1' and Clock'event ) and
                        ( ( FB_InstrWrite_A_B = "10" and FB_InstrA_Slot <=
cInstrSlot-1 ) or
                          ( FB_InstrWrite_A_B = "01" and FB_InstrB_Slot <=
cInstrSlot-1 ) ) else

  FB_CurrentSlot+2 when ( Clock = '1' and Clock'event ) and
                        ( FB_InstrWrite_A_B = "11" and FB_InstrB_Slot <=
cInstrSlot-1 ) else

  unaffected;
```

This part is the concurrent part of the trace cache memory and trace cache hit logic.

```
-------------------------------------------------
--        Trace Cache Portion (Concurrent)       --
-------------------------------------------------

TC_FirstInstrAddr <=          FB_InstrAddrBuffer(0,0) when FB_TraceSize(0) > 1 and FB_BufferReady(0) =
'1' else
                              FB_InstrAddrBuffer(1,0) when FB_TraceSize(1) > 1 and FB_BufferReady(1) =
'1' else
                              FB_InstrAddrBuffer(2,0) when FB_TraceSize(2) > 1 and FB_BufferReady(2) =
'1' else
                              FB_InstrAddrBuffer(3,0) when FB_TraceSize(3) > 1 and FB_BufferReady(3) =
'1' else
                              unaffected;

TC_DestInstrAddr <=           FB_InstrAddrBuffer(0,FB_BranchSlot(0)+1) when FB_TraceSize(0) > 1 and
                                                                          FB_BufferReady(0) =
'1' and                                                                   FB_BranchSlot(0) <
FB_TraceSize(0)-1 and
FB_BranchExisting(0) = '1' else
```

```
                              FB_InstrAddrBuffer(1,FB_BranchSlot(1)+1) when FB_TraceSize(1) > 1 and
                                                                                    FB_BufferReady(1) =
'1' and
                                                                                    FB_BranchSlot(1) <
FB_TraceSize(1)-1 and
FB_BranchExisting(1) = '1' else
                              FB_InstrAddrBuffer(2,FB_BranchSlot(2)+1) when FB_TraceSize(2) > 1 and
                                                                                    FB_BufferReady(2) =
'1' and
                                                                                    FB_BranchSlot(2) <
FB_TraceSize(2)-1 and
FB_BranchExisting(2) = '1' else
                              FB_InstrAddrBuffer(3,FB_BranchSlot(3)+1) when FB_TraceSize(3) > 1 and
                                                                                    FB_BufferReady(3) =
'1' and
                                                                                    FB_BranchSlot(3) <
FB_TraceSize(3)-1 and
FB_BranchExisting(3) = '1' else
                              unaffected;

TC_SelectedEntry <= To_Integer(TC_FirstInstrAddr(3 downto 2));        -- <--- No of bit for TC line
amount


----------------------------------------
--        Trace Cache Hit Logic                --
----------------------------------------
TC_FirstTagHit <= Equal( TC_ValidBit(To_Integer(IF_InstrCounterReg(3 downto 2)))  &
                         TC_TagReg_01(To_Integer(IF_InstrCounterReg(3 downto 2))) ,
                         '1' & IF_InstrCounterreg(31 downto 2)); -- ^^^^ Change here for No of bit
for TC line amount
TC_OtherHit <= '1' when TC_HitLine = '1' and
                        TC_FirstTagHit = '0' and
                        ( To_Integer(IF_InstrCounterReg) >
To_Integer(TC_TagReg_01(TC_HitLineNumber) & "00") ) and
                        ( ( TC_BranchExisting(TC_HitLineNumber) = '0' and
                          To_Integer(IF_InstrCounterReg) <
To_Integer(TC_TagReg_01(TC_HitLineNumber) & "00" + ( TC_TraceSize(TC_HitLineNumber) * 4 ))
                        ) or
                          ( TC_BranchExisting(TC_HitLineNumber) = '1' and
                            (
                              (
                               To_Integer(IF_InstrCounterReg) <
To_Integer(TC_TagReg_01(TC_HitLineNumber) & "00" + ( ( TC_BranchSlot(TC_HitLineNumber)+1)*4))
                              ) or
                              (
                               To_Integer(IF_InstrCounterReg) <
To_Integer(TC_TagReg_02(TC_HitLineNumber) & "00" + ( ( TC_TraceSize(TC_HitLineNumber)-
TC_BranchSlot(TC_HitLineNumber)+1) *4))
                              )
                            )
                          )
                        )
                    else
                '0';

TC_Hit <= TC_FirstTagHit or TC_OtherHit;
```

This section is the sequential portion of the VHDL code that defines the conditions to put instructions and the associated addresses into fill-buffer.

```
-----------------------------------------------------------------
--                        Fill Buffer                          --
-----------------------------------------------------------------

                -- Place instruction(s) and address(es) into fill-buffer
                if FB_InstrAWrite = '1' then
                        FB_InstrBuffer( FB_InstrA_Row , FB_InstrA_Slot ) <= IF_InstrRegA_Input;
                        FB_InstrAddrBuffer(     FB_InstrA_Row     ,     FB_InstrA_Slot     )     <=
IF_InstrAddrRegA_Input;
                end if;

                if FB_InstrBWrite = '1' then
                        FB_InstrBuffer( FB_InstrB_Row , FB_InstrB_Slot ) <= IF_InstrRegB_Input;
                        FB_InstrAddrBuffer(     FB_InstrB_Row     ,     FB_InstrB_Slot     )     <=
IF_InstrAddrRegB_Input;
                end if;

                -- For Experiment
                if FB_InstrWrite_A_B = "10" then
                        FB_LastInstr <= IF_InstrAddrRegA_Input;
                        FB_LastInstrShift <= IF_InstrAddrRegA_Input+4;
                        FB_LastInstrIsBranch <= IsBranch(IF_InstrRegA_Input);
                end if;

                if FB_InstrWrite_A_B = "01" or FB_InstrWrite_A_B = "11" then
                        FB_LastInstr <= IF_InstrAddrRegB_Input;
                        FB_LastInstrShift <= IF_InstrAddrRegB_Input+4;
                        FB_LastInstrIsBranch <= IsBranch(IF_InstrRegB_Input);
                end if;
```

```
                -- Any row termination(s)?
                -- If so, which instruction? (A and/or B did it) and which line?
                -- When known, set the "Buffer Ready flag" to indicate the incident
                if FB_RowTerminatedByA = '1' then
                        FB_BufferReady( FB_FinishRowNumber_A ) <= '1';
                end if;

                if FB_RowTerminatedByB = '1' then
                        FB_BufferReady( FB_FinishRowNumber_B ) <= '1';
                end if;

                -- Counting the trace size of fill-buffer line(s)
                if FB_InstrWrite_A_B = "10" then
                        FB_TraceSize( FB_InstrA_Row ) <= FB_TraceSize( FB_InstrA_Row ) + 1;
                end if;

                if FB_InstrWrite_A_B = "01" then
                        FB_TraceSize( FB_InstrB_Row ) <= FB_TraceSize( FB_InstrB_Row ) + 1;
                end if;

                if FB_InstrWrite_A_B = "11" then
                        if FB_TraceCount2Up = '1' then
                                FB_TraceSize( FB_InstrA_Row ) <= FB_TraceSize( FB_InstrA_Row ) +
2;
                        else
                                FB_TraceSize( FB_InstrA_Row ) <= FB_TraceSize( FB_InstrA_Row ) +
1;
                                FB_TraceSize( FB_InstrB_Row ) <= FB_TraceSize( FB_InstrB_Row ) +
1;
                        end if;
                end if;

                -- Updating "Branch Existing Flag" and "Branch Slot" of trace information when there
comes the branch
                if FB_InstrA_IsBranch = '1' then
                        FB_BranchExisting( FB_BranchInstrA_Row ) <= '1';
                        FB_BranchSlot( FB_InstrA_Row ) <= FB_InstrA_Slot;
                end if;

                if FB_InstrB_IsBranch = '1' then
                        FB_BranchExisting( FB_BranchInstrB_Row ) <= '1';
                        FB_BranchSlot( FB_InstrB_Row ) <= FB_InstrB_Slot;
                end if;
```

This section describes the transfer function of instructions from fill-buffer to trace cache memory and the trace cache hit consideration.

```
-- Trace transfer function and line reset
        for line in FB_BufferReady'range loop
                if FB_BufferReady( line ) = '1' then
                -- If there are more than one instruction in the line, the transfer function will
commence.
                -- Otherwise, the line would be abandoned.
                        if ( FB_TraceSize( line ) > 1 ) and
                           (
                             (TC_FirstInstrAddr(31 downto 2) /= TC_TagReg_01(TC_selectedEntry)) or
                             (FB_TraceSize( line ) >= TC_TraceSize(TC_SelectedEntry))
                           ) then
                                -- Transfer function commencing
                                TC_ValidBit(TC_SelectedEntry) <= '1';
                                TC_TagReg_01(TC_SelectedEntry) <= TC_FirstInstrAddr(31 downto 2);
                                if ( FB_BranchExisting(line) = '1' and FB_BranchSlot(line) <
FB_TraceSize(line)-1 ) then
                                        TC_TagReg_02(TC_SelectedEntry) <= TC_DestInstrAddr(31
downto 2);
                                else
                                        TC_TagReg_02(TC_SelectedEntry) <= TC_FirstInstrAddr(31
downto 2);
                                end if;
                                TC_TraceSize(TC_SelectedEntry) <= FB_TraceSize(line);
                                TC_BranchExisting(TC_SelectedEntry) <= FB_BranchExisting(line);
                                TC_BranchSlot(TC_SelectedEntry) <= FB_BranchSlot(line);

                                -- For counting the number of writing to the individual cache
line
                                TC_TraceWrite(TC_SelectedEntry) <=
TC_TraceWrite(TC_SelectedEntry)+1;

                                -- Number of overwriting

                                if ( ( TC_TraceSize(TC_selectedEntry) /= FB_TraceSize(line) ) or
                                     ( ( TC_TagReg_02(TC_selectedEntry) /= TC_DestInstrAddr(31
downto 2) ) and
                                       ( FB_BranchExisting(line) = '1' and FB_BranchSlot(line) <
FB_TraceSize(line)-1 ) ) or
                                     ( ( TC_TagReg_02(TC_selectedEntry) /= TC_FirstInstrAddr(31
downto 2) ) and
                                       not ( FB_BranchExisting(line) = '1' and
FB_BranchSlot(line) < FB_TraceSize(line)-1 ) ) ) then
                                        TC_TraceOverwrite(TC_SelectedEntry) <=
TC_TraceOverwrite(TC_SelectedEntry)+1;
```

```
                                        end if;

                                        -- Recording the longest trace in a particular line
                                        if ( FB_TraceSize( line ) > TC_LongestTrace( TC_SelectedEntry ) )
then
                                                TC_LongestTrace(  TC_SelectedEntry  )  <=  FB_TraceSize(
line );
                                        end if;

                                        for trace_slot in 0 to cInstrSlot loop
                                                TC_Instr(TC_SelectedEntry,trace_slot)                   <=
FB_InstrBuffer(line,trace_slot);
                                                TC_InstrAddr(TC_SelectedEntry,trace_slot)               <=
FB_InstrAddrBuffer(line,trace_slot);
                                        end loop;

                                end if;
                                FB_BufferReady( line ) <= '0';
                                FB_TraceSize( line ) <= 0;
                                FB_BranchExisting( line ) <= '0';
                                FB_BranchSlot( line ) <= 0;
                                for slot in 0 to cInstrSlot loop
                                        FB_InstrBuffer( line , slot ) <= ( others => '0' );
                                        FB_InstrAddrBuffer( line , slot ) <= ( others => '0' );
                                end loop;
                        end if;
                end loop;

                --------------------------
                --      TC Hit Logic      --
                --------------------------
                If TC_FirstTagHit = '1' then
                        TC_HitLine <= '1';
                        TC_HitLineNumber <= To_Integer(IF_InstrCounterReg(3 downto 2));         --        <----
Change here for TC lines
                end if;
```

This section shows the resetting of the trace cache signal at the start of the simulation (in **bold**).

```
-----------------------------------------------------------------------
--                  External RESET                                   --
-----------------------------------------------------------------------

        if Reset = '1' then
                IF_ValidFlagA <= '0';
                IF_ValidFlagB <= '0';
                BTB_ValidFlag <= ( others => '0' );
                DP_HaltFlag <= '0';
                DP_InterruptEnableFlag <= '0';
                DP_ProcessIdentifierReg <= ( others => '0' );
                RB_ValidFlag <= ( others => '0' );
                BRU_ValidFlag <= '0';
                ALU_ValidFlag <= '0';
                MDU_ValidFlag <= '0';
                LSU_ValidFlag <= '0';
                LSU_EA_ValidFlag <= '0';
                LSU_SPR_ValidFlag <= '0';
                CU_NextCommitPointerReg <= "10000";             -- Initialize pointer
                ITB_ValidFlag <= ( others => '0' );
                IC_ValidFlag <= ( others => '0' );
                DTB_ValidFlag <= ( others => '0' );
                DC_ValidFlag <= ( others => '0' );
                WB_EntranceValidFlag <= '0';
                WB_ValidFlag <= ( others => '0' );
                BIU_ActiveLoadFlag <= '0';
                BIU_ActiveFetchFlag <= '0';
                BIU_ActiveStoreFlag <= '0';
                BIU_FirstBusClockOfActiveCycleFlag <= '0';

                TC_TraceWrite <= ( others => 0 );
                TC_TraceOverwrite <= ( others => 0 );

        end if;
```

This is the last part of *Dlx.vhd* for writing the *log* file of the simulation.

```
-----------------------------------
-- Write acquisited data to file --
-----------------------------------

process
begin
        -- BusClock continues while DLX is halted.
```

```
        wait on IncomingClock until IncomingClock = '1';

        BIU_BusClock <= not BIU_BusClock;
end process;

---------------------------------
--      Model data logger      --
---------------------------------
process

        -- Name of experiment set
        constant exp_name : string(1 to 49)    := "TC_4 :   4 Lines      *Test Program:              ";
        constant exp_name_ul : string(1 to 49) := "*********************************************";

        -- Type of Hit/Miss
        constant TraceMiss : string(1 to 26) :=    "No. of Trace Cache Miss = ";
        constant TCAccessMiss : string(1 to 26) := "No. of TC Access Miss   = ";

        constant CompMiss : string(1 to 28) :=     "No. of TC Compulsary Miss = ";
        constant ContMiss : string(1 to 28) :=     "No. of TC Conflict Miss   = ";

        constant CacheAccess : string(1 to 26)  := "No. of All Cache Access = ";


        -- Disposal
        constant ICache_Hit : string(1 to 30) := "Total Instruction Cache Hit = ";
        constant TCache_FirstTagHit : string(1 to 21) := "TC (First Tag) Hit = ";
        constant TCache_OtherHit : string(1 to 17) := "TC (other) Hit = ";
        constant PC : string(1 to 26) := "Program Counter Address : ";
        constant InstrA : string(1 to 26) := "Instruction A Address   : ";
        constant InstrB : string(1 to 26) := "Instruction B Address   : ";
        constant Commit : string(1 to 30) := "Committed Instruction Count = ";
        constant Omit : string(1 to 28) := "Omitted Instruction Count = ";
        constant MemFetch : string(1 to 29) := "I-Cache Fetch Memory Count = ";


        use std.textio.all;

        -- Result file name
        file log: text open write_mode is "result.log";

        variable log_line : line;

        -- Variables for experiment result
        variable TraceMissCount : natural :=0;
        variable TCAccessMissCount : natural :=0;

        variable CompMissCount : natural :=0;
        variable ContMissCount : natural :=0;
        variable CacheAccessCount : natural :=0;

        variable Instr_Count : natural:=0;
        variable InstrA_Count : natural:=0;
        variable InstrB_Count : natural:=0;


        variable ICache_HitCount : natural:=0;
        variable TCache_FirstTagHitCount : natural:=0;
        variable TCache_OtherHitCount : natural:=0;


        type TraceCacheLine is array (0 to cTC_Entry) of natural;
        variable CompMissLineCount : TraceCacheLine;
        variable ContMissLineCount : TraceCacheLine;

        variable FirstTagHitCount : TraceCacheLine;
        variable ContentHitCount  : TraceCacheLine;

        variable Commit_Count : natural:=0;
        variable MemFetch_Count : natural:=0;
        variable PC_Word : string(1 to 8);
        variable InstrA_word : string(1 to 8);
        variable InstrB_word : string(1 to 8);

        variable CacheSpaceUsage : integer:=0;

        -- Datatype conversion functions
        function NumberToDigit( Number : natural ) return character is
        begin
                if (Number >= 0) and (Number <= 9) then
                        return character'val( character'pos('0') + Number );
                elsif (Number >= 10) and (Number <= 15) then
                        return character'val( character'pos('A') - 10 + Number );
                else
                        report "Invalid Hex-Number"
                        severity error;
                        return '0';
                end if;
        end NumberToDigit;

        function NaturalToString( Number : natural  ) return string is
                variable StringResult : string(1 to 8);
                variable WorkNumber : natural := Number;
        begin
                for i in 8 downto 1 loop
                        StringResult( i ) := NumberToDigit( WorkNumber mod 16 );
                        WorkNumber := WorkNumber / 16;
                end loop;
                return StringResult;
        end NaturalToString;

        function WordToString( Word : unsigned ) return string is
```

```
                variable StringResult : string(1 to 8);
                variable Digit : unsigned( 3 downto 0 );
        begin
                for i in 1 to 8 loop
                        Digit := Word( 4*(8-i)+3 downto 4*(8-i) );
                        StringResult( i ) := NumberToDigit( natural( To_Integer( Digit ) ) ) ;
                end loop;
                return StringResult;
        end WordToString;


begin
        wait on Clock until Clock = '1';

        PC_Word := WordToString(IF_InstrCounterReg);
        InstrA_Word := WordToString(IF_InstrAddrRegA_Input);
        InstrB_Word := WordToString(IF_InstrAddrRegB_Input);


        if IF_InstrCounterRegWrite = '1' then

                CacheAccessCount := CacheAccessCount + 1;

                if not ( TC_FirstTagHit = '1' or TC_OtherHit = '1' ) then
                        TraceMissCount := TraceMissCount + 1;
                end if;

                if ( TC_FirstTagHit = '0' and TC_OtherHit = '0' ) then
                        TCAccessMissCount := TCAccessMissCount + 1;
                        if TC_ValidBit(To_Integer(IF_InstrCounterReg(3 downto 2))) = '0' then
                                CompMissCount := CompMissCount + 1;
                                CompMissLineCount(To_Integer(IF_InstrCounterReg(3 downto 2))) :=
CompMissLineCount(To_Integer(IF_InstrCounterReg(3 downto 2))) + 1;

                        else
                                ContMissCount := ContMissCount + 1;
                                ContMissLineCount(To_Integer(IF_InstrCounterReg(3 downto 2))) :=
ContMissLineCount(To_Integer(IF_InstrCounterReg(3 downto 2))) + 1;
                        end if;
                end if;

                if TC_FirstTagHit = '1' then
                        FirstTagHitCount(To_Integer(IF_InstrCounterReg(3 downto 2))) :=
FirstTagHitCount(To_Integer(IF_InstrCounterReg(3 downto 2))) + 1;
                end if;

                if TC_OtherHit = '1' then
                        ContentHitCount(TC_HitLineNumber) := ContentHitCount(TC_HitLineNumber) + 1;
                end if;

        end if;


        if DP_HaltDlx = '1' then

                -- Display the experiment set name
                write(log_line,exp_name_ul);
                writeline(log,log_line);
                write(log_line,exp_name);
                writeline(log,log_line);
                write(log_line,exp_name_ul);
                writeline(log,log_line);
                writeline(log,log_line);

                -- ********************
                -- General Information
                -- ********************
                write(log_line,string'("--------------------")); writeline(log,log_line);
                write(log_line,string'("General Information")); writeline(log,log_line);
                write(log_line,string'("--------------------"));
                writeline(log,log_line);

                -- Fetched Instruction Count
                write(log_line,string'("Total Fetched Instructions = "));
                write(log_line,Instr_Count);
                writeline(log,log_line);

                -- Commited/Omitted Instructions
                write(log_line,string'("Committed Instructions = "));
                write(log_line,Commit_Count);
                writeline(log,log_line);
                write(log_line,string'("Omitted Instructions   = "));
                write(log_line,Instr_Count-Commit_Count);
                writeline(log,log_line);

                -- ******************
                -- Cache Information
                -- ******************
                -- Cache Access
                write(log_line,string'("Cache Memory Access (fetch) = "));
                write(log_line,CacheAccessCount);
                writeline(log,log_line);
                writeline(log,log_line);

                -- Instruction Cache
                write(log_line,string'("-----------------------")); writeline(log,log_line);
                write(log_line,string'("Instruction-Cache Info")); writeline(log,log_line);
                write(log_line,string'("-----------------------"));
                writeline(log,log_line);
                write(log_line,string'("Instruction-Cache Hit = "));
                write(log_line,ICache_HitCount);
                writeline(log,log_line);
```

```
                        write(log_line,string'("Percent of IC Hit = "));
                        write(log_line,real(ICache_HitCount*100)/real(CacheAccessCount),digits => 2);
                        writeline(log,log_line);

                        writeline(log,log_line);

                        -- Trace Cache
                        write(log_line,string'("----------------")); writeline(log,log_line);
                        write(log_line,string'("Trace-Cache Info")); writeline(log,log_line);
                        write(log_line,string'("----------------"));
                        writeline(log,log_line);
                        write(log_line,string'("Trace-Cache Hit = "));
                        write(log_line,TCache_FirstTagHitCount + TCache_OtherHitCount);
                        write(log_line,string'(" ( "));
                        write(log_line,string'("TC-First Tag Hit = "));
                        write(log_line,TCache_FirstTagHitCount);
                        write(log_line,string'(" / TC-Content Hit = "));
                        write(log_line,TCache_OtherHitCount);
                        write(log_line,string'(" )"));
                        writeline(log,log_line);

                        write(log_line,string'("Percent of TC Hit = "));
                        write(log_line,real((TCache_FirstTagHitCount
TCache_OtherHitCount)*100)/real(CacheAccessCount),digits => 2);
                        write(log_line,string'(" ( "));
                        write(log_line,string'("TC-First Tag Hit = "));
                        write(log_line,real(TCache_FirstTagHitCount*100)/real(CacheAccessCount),digits => 2);
                        write(log_line,string'(" / TC-Content Hit = "));
                        write(log_line,real(TCache_OtherHitCount*100)/real(CacheAccessCount),digits => 2);
                        write(log_line,string'(" )"));
                        writeline(log,log_line);

                        writeline(log,log_line);
                        write(log_line,string'("Total Trace Cache Miss = "));
                        write(log_line,TraceMissCount);
                        write(log_line,string'(" ( "));
                        write(log_line,string'("TC - Compulsary Miss = "));
                        write(log_line,CompMissCount);
                        write(log_line,string'(" / TC - Conflict Miss = "));
                        write(log_line,ContMissCount);
                        write(log_line,string'(" )"));
                        writeline(log,log_line);

                        write(log_line,string'("Percent of ( "));
                        write(log_line,string'("Compulsary Miss = "));
                        write(log_line,real(CompMissCount*100)/real(TraceMissCount),digits => 2);
                        write(log_line,string'(" / Conflict Miss = "));
                        write(log_line,real(ContMissCount*100)/real(TraceMissCount),digits => 2);
                        write(log_line,string'(" )"));
                        writeline(log,log_line);

                        writeline(log,log_line);

                        -- *****************
                        -- Information Table
                        -- *****************

                        write(log_line,string'("-----------------------------------------------------"));
writeline(log,log_line);
                        write(log_line,string'("Information Collected From Individual Trace Cache Line"));
writeline(log,log_line);
                        write(log_line,string'("-----------------------------------------------------"));
                        writeline(log,log_line);
                        writeline(log,log_line);

                        write(log_line,string'("Line     Comp-Miss     Conf-Miss     TC-Write     TC-O_Write
TC-Size      TC-Hit     FTag-Hit     Cont-Hit")); writeline(log,log_line);
                        write(log_line,string'("-------------------------------------------------------------
-----------------------------------------------")); writeline(log,log_line);

                        for index in 0 to cTC_Entry loop
                                write(log_line,index,justified     =>     right,     field     =>     4);
write(log_line,string'("   "));
                                write(log_line,CompMissLineCount(index),justified => right, field => 10);
write(log_line,string'("   "));
                                write(log_line,ContMissLineCount(index),justified => right, field => 10);
write(log_line,string'("   "));
                                write(log_line,TC_TraceWrite(index),justified => right, field => 10);
write(log_line,string'("   "));
                                write(log_line,TC_TraceOverWrite(index),justified => right, field => 10);
write(log_line,string'("   "));
                                write(log_line,TC_LongestTrace(index),justified => right, field => 10);
write(log_line,string'("   "));
                                write(log_line,FirstTagHitCount(index)+ContentHitCount(index),justified =>
right, field => 10); write(log_line,string'("   "));
                                write(log_line,FirstTagHitCount(index),justified => right, field => 10);
write(log_line,string'("   "));
                                write(log_line,ContentHitCount(index),justified => right, field => 10);
write(log_line,string'("   "));
                                writeline(log,log_line);
                                -- Calculate the sum of Cache Space Usage
                                CacheSpaceUsage := CacheSpaceUsage + TC_LongestTrace(index);
                        end loop;

                        writeline(log,log_line);
                        write(log_line,string'("--------------------------------")); writeline(log,log_line);
                        write(log_line,string'("Percentage of Cache Space Usage")); writeline(log,log_line);
                        write(log_line,string'("--------------------------------")); writeline(log,log_line);
                        write(log_line,string'("Trace Cache Space Usage = "));
                        write(log_line,real( CacheSpaceUsage * 100 / ( (cTC_Entry+1) * (cInstrSlot+1) ) ) ,
digits => 2);
                        write(log_line,string'(" %"));
```

```
                writeline(log,log_line);
            end if;

            -------------------------
            -- Counting Mechanism --
            -------------------------
        if IF_InstrCounterRegWrite = '1' then

            if IC_Hit = '1' then
                ICache_HitCount := ICache_HitCount+1;
            end if;

            if TC_FirstTagHit = '1' then
                TCache_FirstTagHitCount := TCache_FirstTagHitCount+1;
            end if;

            if TC_OtherHit = '1' then
                TCache_OtherHitCount := TCache_OtherHitCount+1;
            end if;

        end if;

            if IC_FetchRequest = '1' then
                MemFetch_Count := MemFetch_Count+1;
            end if;

            if IF_InstrCounterRegWrite = '1' then
                if ( IF_StageA_Write = '1' and IF_StageB_Write = '1' ) then
                    if ( IF_InstrCounterReg = IF_InstrAddrRegA_Input ) and
                       ( IF_InstrCounterReg /= IF_InstrAddrRegB_Input ) then
                            Instr_Count := Instr_Count+2;
                    elsif ( ( IF_InstrCounterReg /= IF_InstrAddrRegA_Input ) and
                            ( IF_InstrCounterReg = IF_InstrAddrRegB_Input ) ) or
                          ( ( IF_InstrCounterReg = IF_InstrAddrRegA_Input ) and
                            ( IF_InstrCounterReg = IF_InstrAddrRegB_Input ) ) then
                                Instr_Count := Instr_Count+1;
                    end if;
                elsif ( IF_StageA_Write = '1' and IF_StageB_Write = '0' ) or
                      ( IF_StageA_Write = '0' and IF_StageB_Write = '1' ) then
                        Instr_Count := Instr_Count+1;
                end if;
            end if;

            if ( CU_CommitInstrA = '1' and CU_CommitInstrB = '1' ) then
                Commit_Count := Commit_Count+2;
            elsif ( CU_CommitInstrA = '1' and CU_CommitInstrB = '0' ) or
                  ( CU_CommitInstrA = '0' and CU_CommitInstrB = '1' ) then
                Commit_Count := Commit_Count+1;
            end if;

        end process;
```

B.2 DlxPackage.vhd

This section is to define types, subtypes, and constants used in the trace cache.

```
-------------------------------------------------
--        For Trace Cache Model                --
-------------------------------------------------

type TypeArrayInstr is array (natural range<>,natural range<>) of unsigned (31 downto 0);

-- TC_4 : cInstrSlot = 3
-- TC_8 : cInstrSlot = 7
constant cInstrRow : integer :=3;     -- These constant has to be added by 1 for actual amount.
constant cInstrSlot : integer :=3;  -- Since they will be mainly used for counter that start at 0 instead
of 1.

constant cTC_Entry : integer :=3;        -- <--- Lines of trace cache

subtype TypeRow is integer range 0 to cInstrRow;
subtype TypeSlot is integer range 0 to cInstrSlot;
subtype TypeSlotCount is integer range 0 to cInstrSlot+1;

type TypeArraySlot is array (natural range<>) of Typeslot;
type TypeArraySlotCount is array (natural range<>) of TypeSlotCount;

type TypeArrayWriteCount is array (natural range<>) of integer;

type TypeArrayTag is array (natural range<>) of unsigned(31 downto 2);
```

This is the declaration of additional functions used in the trace cache. Function *IsBranch* is for checking whether the instruction is any kind of branch instruction and function *IsDelimiterInstr* is for checking, whether the instruction is a jump, trap, or rfe.

```
-- Functions for TraceCache Model
function IsBranch( Instruction : TypeWord ) return bit;
function IsDelimiterInstr( Instruction : TypeWord ) return bit;
```

These functions are here:

```
-- Functions for TraceCache Model
function IsBranch( Instruction : TypeWord ) return bit is
alias InstructionOpcode : TypeDlxOpcode is Instruction( 31 downto 26 );
variable Result : bit := '0';
begin
case InstructionOpcode is
          -- branches
          when cOpcode_beqz => Result := '1';
          when cOpcode_bnez => Result := '1';
          -- not a branch
          when others => Result := '0';
end case;

return Result;
end IsBranch;

function IsDelimiterInstr( Instruction : TypeWord ) return bit is
alias InstructionOpcode : TypeDlxOpcode is Instruction( 31 downto 26 );
variable Result : bit := '0';
begin
case InstructionOpcode is
          -- jumps
          when cOpcode_j => Result := '1';
          when cOpcode_jr => Result := '1';
          when cOpcode_jal => Result := '1';
          when cOpcode_jalr => Result := '1';
          -- trap
          when cOpcode_trap => Result := '1';
          -- rfe
          when cOpcode_rfe => Result := '1';
          -- other instructions
          when others => Result := '0';
end case;

return Result;
end IsDelimiterInstr;
```

B.3 Environment.vhd

This file has been modified to increase the memory capacity from 16Kbyte to 32 Kbyte to run *Permute* and *DCT*. Therefore, these two lines are changed.

```
constant cMemorySize : positive := 32768;


constant cHighAddress_unsigned : unsigned := X"0000_7FFF";
```

Originally, the constant *cMemorySize* was 16384 (16Kbyte) and *cHighAddress_unsigned* was X"0000_3FFF".

Appendix C

Excerpts from *log* files of *DCT*

8L

Line	Comp-Miss	Conf-Miss	TC-Write	TC-O_Write	TC-Size	TC-Hit	FTag-Hit	Cont-Hit
tc_4:								
2	12	**19599**	4456	**1335**	4	18134	4543	13591
tc_8:								
2	22	**22667**	3704	**134**	8	7314	4545	2769

16L

Line	Comp-Miss	Conf-Miss	TC-Write	TC-O_Write	TC-Size	TC-Hit	FTag-Hit	Cont-Hit
tc_4:								
2	8	6271	120	113	4	1642	519	1123
tc_8:								
2	**9861**	0	**0**	**0**	0	0	0	**0**

32L

Line	Comp-Miss	Conf-Miss	TC-Write	TC-O_Write	TC-Size	TC-Hit	FTag-Hit	Cont-Hit
tc_4:								
2	5	831	64	1	4	1978	575	1403
tc_8:								
2	**1236**	0	**0**	**0**	0	0	0	**0**

64L

Line	Comp-Miss	Conf-Miss	TC-Write	TC-O_Write	TC-Size	TC-Hit	FTag-Hit	Cont-Hit
tc_4:								
2	5	519	64	1	4	1342	575	767
tc_8:								
2	**1029**	0	**0**	**0**	0	0	0	**0**

Appendix D

Runtime Startup Code and Perl Script Listings

Runtime Startup Code (*crt0.o*)

```
.text
.proc ___main
.global ___main
___main:
            jr      r31
            nop
.endproc ___main
.proc start
.global start
start:
;  Starting point for simulations: loads r29 with memSize and calls main with
;  argc and argv
            lhi     r29,(((memSize-8)>>16)&0xffff)
            addui   r29,r29,((memSize-8)&0xffff)
            addir29,r29,#-16
            add     r1,r0,r0
            lhi     r1,((argc>>16)&0xffff)
            addui   r1,r1,(argc&0xffff)
            lw      r2,(r1)
            sw      (r29),r2
            add     r1,r0,r0
            lhi     r1,((argv>>16)&0xffff)
            addui   r1,r1,(argv&0xffff)
            lw      r2,(r1)
            sw      4(r29),r2
            add     r1,r0,r0
            lhi     r1,((_environ>>16)&0xffff)
            addui   r1,r1,(_environ&0xffff)
            lw      r2,(r1)
            sw      8(r29),r2
            jal     _main
            nop
            addi    r29,r29,#16
            jal     _exit
            nop
.endproc start
.data
.align 2
.global argc
argc:
.word 0
.global argv
argv:
.word 0
.global _environ
_environ:
.word 0
```

Perl Script (*filter.pl*)

```
#!/usr/local/bin/perl
eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}'
    if $running_under_some_shell;
                                # this emulates #! processing on NIH machines.
                                   # (remove #! line above if indigestible)

eval '$'.$1.'$2;' while $ARGV[0] =~ /^([A-Za-z_0-9]+=)(.*)/ && shift; #'
                                          # process any FOO=bar switches

$[ = 1;                                          # set array base to 1
$, = ' ';           # set output field separator
$\ = "\n";          # set output record separator

%regmap = ();
```

```
line: while (<>) {
    chop;   # strip record separator
    @Fld = split(' ', $_, 9999);
    if ($Fld[1] eq 'movi2fp') {
        print ";;; " . $_ ;
        @Parlist = split(',', $Fld[2], 9999); #Should strip ; first
            $regmap{$Parlist[1]} = $Parlist[2];
            next;
    }
    if ($Fld[1] =~ /mult/ || $Fld[1] =~ /div/ || $Fld[1] =~ /multu/ ||

      $Fld[1] =~ /div/) {
        print ";;; " . $_ ;
        @Parlist = split(',', $Fld[2], 9999); #Should strip ; first
            $operation = $Fld[1];
            $oprnd1 = $Parlist[1];
            $oprnd2 = 'f' . substr($regmap{$Parlist[2]}, 2, 999999);
            $oprnd3 = 'f' . substr($regmap{$Parlist[3]}, 2, 999999);
            next;
    }
    if ($Fld[1] eq 'movfp2i') {
        print ";;; " . $_ ;
        @Parlist = split(',', $Fld[2], 9999); #Should strip ; first
            if ($Parlist[2] ne $oprnd1) {               #???
                print 'Translation sequence error at line' . $.;
                last line;
            }
            else {
                print "\t" . "nop";
                print "\t" . "nop";
                print "\t" . $operation . "\t" . 'f' . substr($Parlist[1], 2,

                  999999) . ',' . $oprnd2 . ',' . $oprnd3 . "\t; Patched from" .

                  $_;
            }
            next;
    }
    print $_;
}
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compiler: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[2] P. J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[3] D. Burger, T. Austin, and S. Bennett, "Evaluating Future Microprocessor: the SimpleScalar Tool Set," Technical Report 1308, University of Wisconsin-Madison Technical Report, July 1996.

[4] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of Instruction Fetch Mechanism for High Issue Rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[5] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," in *Proceedings of the 28th ACM/IEEE Annual International Symposium on Microarchitecture*, pp. 258-263, 1995.

[6] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 162-171, 1994.

[7] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative Fetch and Issue Techniques from the Trace Cache Fetch Mechanism," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[8] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, CA, second edition, 1996.

[10] J. Horch, "A Superscalar Version of the DLX Processor," *Superscalar DLX Processor*, 1997. http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html (9 July 1999).

[11] Intel Corporation, "Intel® NetBurst™ Microarchitecture," *The Intel® Pentium® 4 Processor Product Overview*, 2002. http://www.intel.com/design/Pentium4/prodbref/index.htm (15 July 2002).

[12] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[13] J. D. Johnson, "Expansion Caches for Superscalar Microprocessors," Technical Report CSL-TR-94-630, Stanford University, Palo Alto CA, June 1994.

[14] S. Jourdan, P. Sainrat, and D. Litalize, "Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 117-125, 1995.

[15] S. Jourdan, P. Sainrat, and D. Litalize, "An Investigation of the Performance of Various Instruction-Issue Buffer Topologies," in *Proceedings of the 28th ACM/IEEE Annual International Symposium on Microarchitecture*, pp. 279-284, 1995.

[16] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," in *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 60-63, 1988.

[17] S. W. Melvin and Y. N. Patt, "Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines," in *Proceedings of Supercomputing '89*, pp. 427-432, 1989.

[18] S. J. Patel, M. Evers, and Y. N. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[19] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism," Technical Report CSE-TR-335-97, University of Michigan Technical Report, May 1997.

[20] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Evaluation of Design Options for the Trace Cache Fetch Mechanism," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 435-446, February 1999.

[21] S. J. Patel, "Trace Cache Design for Wide-Issue Superscalar Processor," PhD Dissertation, University of Michigan, Ann Arbor MI, 1999.

[22] A. Peleg and U. Weiser. *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line*. U.S. Patent Number 5,381,533, 1994.

[23] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," *Journal of Supercomputing*, vol. 7, no. 1/2, pp. 9-50, 1993.

[24] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," Technical Report 1310, University of Wisconsin-Madison Technical Report, April 1996.

[25] E. Rotenberg, S. Bennett, and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation," *IEEE Transaction on Computers*, vol. 48, no. 2, pp. 111-120, February 1999.

[26] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith, "Trace Processors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[27] R. H. Saavedra and A. J. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes," *IEEE Transaction on Computers*, vol. 44, no. 10, pp. 1223-1235, October 1995.

[28] P. M. Sailer and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook*, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[29] M. Schlansker et al., "Compilers for Instruction-Level Parallelism," *Computer*, pp. 63-69, December 1997.

[30] D. Sima, "Superscalar Instruction Issue," *IEEE Micro*, vol. 17, pp. 28-39, September-October 1997.

[31] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473-530, September 1982.

[32] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, vol. 83, pp. 1609-1624, December 1995.

[33] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 219-229, 1995.

[34] T-Y. Yeh, D. Marr and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," in *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 67-76, 1993.