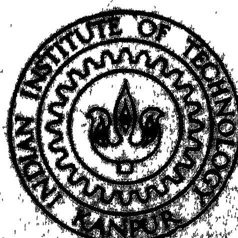
Architectural Enhancements in PERL Processor

by

Suraj Kumar Bhatnagar

Th
CSE/1998/M
B 469a

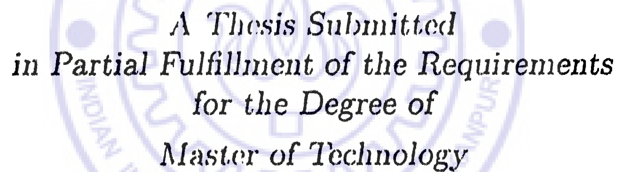
CSE
1998
M
BHA
ARC



Department of Computer Science & Engineering
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MAY, 1998

Architectural Enhancements in PERL Processor

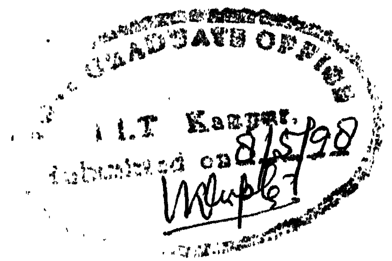


A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Suraj Kumar Bhatnagar

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
May, 1998



Certificate

Certified that the work contained in the thesis entitled "*Architectural Enhancements in PERL Processor*", by Mr.*Suraj Kumar Bhatnagar*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Rajat Moona)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

May, 1998

CENTRAL LIBRARY
I I T KANPUR
Acc No A 125500
CSE-1998-M-BHA-ARC

Abstract

PERL (performance enhanced register-less) architecture is a memory to memory architecture. The main idea behind this architecture is to exploit the effective caching techniques to do away with the register space. The availability of large on-chip caches and wider bus bandwidth to reduce memory latency support this idea. PERL processor embraces all advanced techniques used in other modern processors. **SuperSIM** [Bal97] is an instruction set simulator to analyze the performance of PERL processor. In this thesis, we extend SuperSIM to incorporate prediction for indirect jumps as they lead to pipeline stalls. Further, the avoidable stalls due to non-availability of addresses to the instructions are reduced using extended address forwarding.

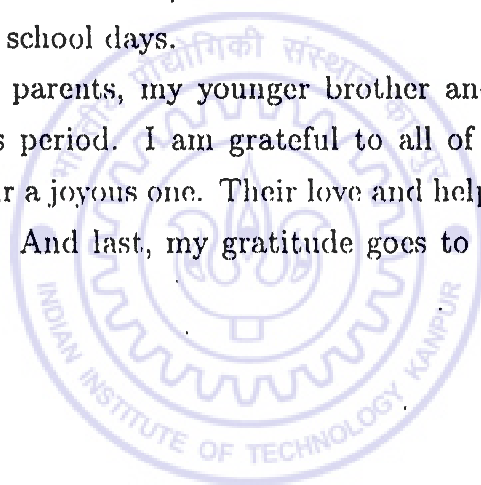
We obtained the results of PERL processor incorporating prediction for indirect jumps and with extended address forwarding. These results are compared with the results obtained by existing simulator for PERL RISC and with similar results obtained for current day processors.

Acknowledgement

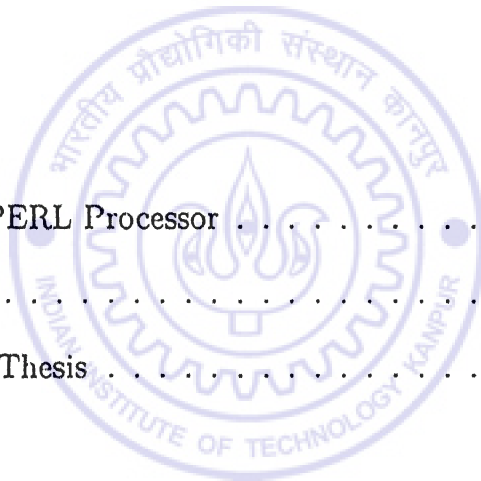
At the very outset I express my sincere gratitude to **Dr. Rajat Moona** for granting me this work under his supervision. He was always helpful and willing to share my difficulties. I am indebted to him for his kindness. I also wish to thank Mr. P. Suresh for his suggestions in this work.

I express my respect to my eldest sister, late Dr. Radha Bhatnagar who inspired me to higher study since my school days.

I am as ever grateful to my parents, my younger brother and my elder sister for bearing with me during this period. I am grateful to all of my class-mates who made my stay at I.I.T. kanpur a joyous one. Their love and help has been a constant source of inspiration for me. And last, my gratitude goes to my 'hall-IV D-block' friends, for their affection.



Contents



1 Introduction 1

1.1 Salient Features of PERL Processor 2

1.2 Motivation 3

1.3 Organization of the Thesis 4

2 PERL Processor 5

2.1 Architecture 5

2.1.1 Data Types 5

2.1.2 Addressing Modes 6

2.1.3 Instruction Format 7

2.1.4 Instruction Types 8

2.2 Pipeline 10

2.2.1 Fetch Stage 11

2.2.2 Decode Stage 12

2.2.3 Execute Stage 12

2.2.4 Write Back Stage 13

2.2.5 Result Commit Stage 13

3 Enhancements in PERL RISC processor 14

3.1 Existing Branch Prediction Scheme 14

3.2 Prediction techniques for Indirect Jumps 15

3.2.1 Branch Target Buffer (BTB) 15

3.2.2 BTB with pair of stacks 17

3.2.3 Stack Extension 20

3.3 Reasons of stalls and their reductions 20

3.3.1 Reasons of stalls 20

3.3.2 Reductions in stalls 22

4 Simulation Results, Conclusions and Future Work 24

4.1 Benchmark Programs 25

4.2 Conclusions 32

4.3 Future Directions 33

A User Manual 34

B Commom RISC features 41

C An Example Machine Description File 42

References 45

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

List of Tables

1 Integer Arithmetic Instructions 9

2 Logical and Shift Instructions 9

3 Integer Control Instructions 10

4 Comparison for Permutation Benchmark for different version 26

5 Comparison for Permutation Benchmark with different prediction schemes for indirect jumps 26

6 Comparison for Permutation Benchmark with other RISC 27

7 Comparison for multiplication Benchmark for different version 27

8 Comparison for Matrix Multiplication Benchmark with other RISC processors 28

9 Comparison for Time Table Scheduler Benchmark for different version 29

10 Comparison for Time Table Scheduler Benchmark for various prediction techniques 29

11 Comparison for Time-table Scheduler Benchmark with other RISC . . 30

12 Comparison for across Benchmark for different version 30

13 Comparison for across Benchmark with other RISC 31

14 Comparison for relax Benchmark for different version 31

15 Comparison for relax Benchmark with other RISC 32

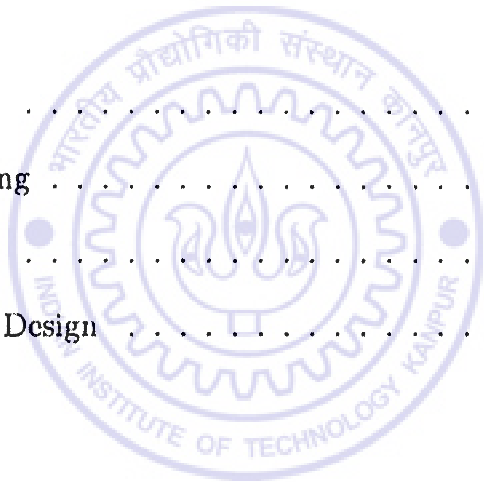
List of Figures

1 Instruction Format 7

2 Instruction Encoding 8

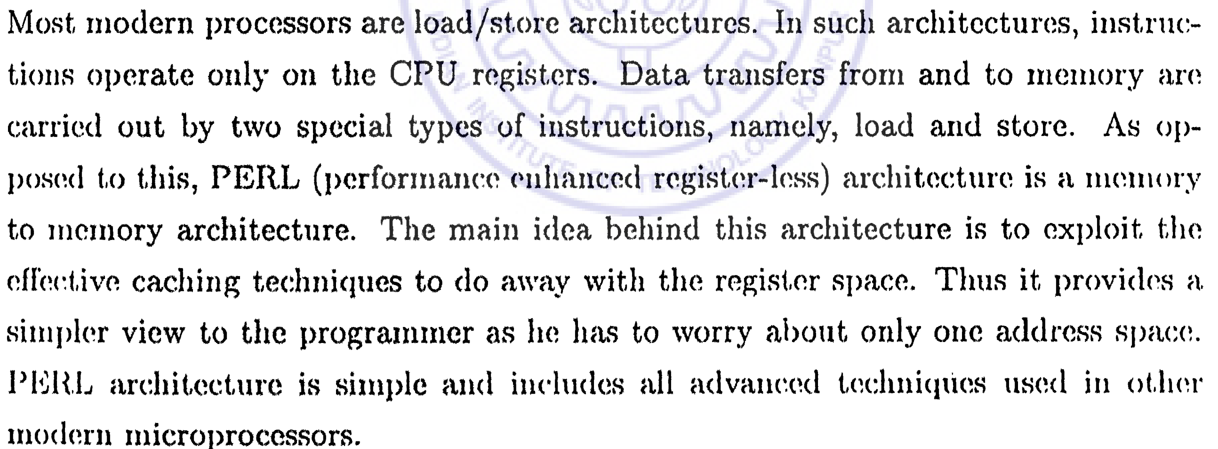
3 Processor Pipeline 11

4 Call/Return Stack Design 18



Chapter 1

Introduction



Most modern processors are load/store architectures. In such architectures, instructions operate only on the CPU registers. Data transfers from and to memory are carried out by two special types of instructions, namely, load and store. As opposed to this, PERL (performance enhanced register-less) architecture is a memory to memory architecture. The main idea behind this architecture is to exploit the effective caching techniques to do away with the register space. Thus it provides a simpler view to the programmer as he has to worry about only one address space. PERL architecture is simple and includes all advanced techniques used in other modern microprocessors.

To analyze the performance of PERL architecture, an instruction set simulator SuperSIM [Bal97] was designed. However, SuperSIM provides a limited functionality in terms of branch prediction and address forwarding, a technique used to pass addresses of operands to the future instructions if it is modified by any instruction in the pipeline. This work extends SuperSIM to incorporate better branch prediction and extended address forwarding, as will be outlined in the thesis.

1.1 Salient Features of PERL Processor

There are many factors that influenced the design of PERL architecture. Firsly, current day microprocessors depend extensively on on-chip caches to offset the memory latencies. Secondly, the availability of faster and cheaper caches which are comparable in speed to registers. Thirdly, the inevitable load/store overheads associated with RISC machines.

PERL architecture follows the RISC philosophy because of the obvious advantages it offers, like simplified control unit, easy fetch and decode logic and efficient handling of pipelines due to uniformity of instruction size and duration of execution.

In PERL, all instructions operate on variables stored in the memory. An instruction can take upto two operands and provides one output, each of which is stored in the memory. Further, addressing modes may demand upto two memory accesses per operand (thus a maximum of 5 memory reads and one write per instruction). However, each instruction execution can be supported by a deep pipeline with each instruction doing much more than an instruction in load/store architecture without affecting simplicity and execution efficiency.

In load/store machines, operations whose operands are less than the size of the register face an overhead of additional operations like sign extentions, masking etc. For example in order to add an integer (32 bits) to a char (8 bits), char is first promoted to an integer and then added. The primary reason for this is that registers do not carry type information whereas memory is typed. Usually compiler uses extra instructions to adjust the data in the registers with respect to the data type. Further, there are additional instructions if the operands are not aligned with respect to the word length of the processor. In a memory to memory architecture, such overheads are eliminated.

In programming, frequently we need to save the current context for restarting at a later time. In PERL, this process is efficient as the machine state is very small comprising of PC and SP (program counter and stack pointer) only.

The instruction set of PERL processor is uniform. It has one opcode and three memory addresses to represent one destination and two operands. Three address format also gives maximum code density [AAD90].

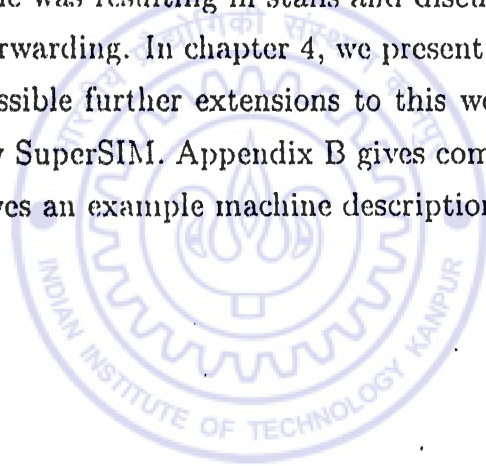
1.2 Motivation

SuperSIM, the simulator for analyzing the performance of PERL processor implements out of order execution, dynamic renaming and branch prediction technique for conditional branches etc. However, SuperSIM does not predict jumps when the target address is stored in memory (for example, return from a function where return address is stored in stack). In most "return from the function" cases (see reasons of stalls in chapter 3), first the stack pointer is adjusted (frame is returned) and then the return is performed. Since return instruction is essentially a jump using SP as a base pointer, this can not proceed till the previous instruction modifying SP is executed. The existing simulator inserts stalls under such cases. However branch prediction can help here. Such is also the case for any other indirect jumps. This influenced us to incorporate an efficient prediction technique for indirect jumps.

Further, the limited use of address forwarding in existing simulator led to avoidable pipeline stalls (see reasons of stalls in chapter 3). An instruction whose destination uses based or indirect addressing mode, can not know the destination address till the previous instruction modifying the base or indirect address is executed. All instructions that follow this instruction can not proceed in the pipeline because we do not know into which location this instruction writes. The existing simulator stalls till the instruction modifying the base or indirect address exits the pipeline. However, address forwarding can reduce such stalls. In this thesis, we also extend the scope of address forwarding to reduce such stalls.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. In chapter 2, we present the PERL architecture. In particular, we describe the instruction set, instruction types, addressing modes, data types and pipeline of the processor. In chapter 3 we present architectural enhancements proposed in PERL. We present the prediction techniques for indirect jumps and their implementation. We also describe circumstances where earlier design of the pipeline was resulting in stalls and discuss techniques to avoid such stalls using address forwarding. In chapter 4, we present the simulation results and conclude by giving possible further extensions to this work. Appendix A lists the command supported by SuperSIM. Appendix B gives common features of RISC and finally Appendix C gives an example machine description file.



Chapter 2

PERL Processor

2.1 Architecture

PERL processor is a superscalar [ss95] memory-to-memory architecture. The availability of large on-chip caches and wider bus bandwidth to reduce memory latency support this idea. PERL RISC is a 64-bit architecture and supports both integer and floating point instructions. It supports four addressing modes. All instructions have a fixed length of 128 bits. The first 32 bits of the instruction are used to specify data types, addressing modes, base pointers for base addressing and the opcode. Other 32-bit words are used to indicate destination and each of the two operands. It follows all RISC features (see appendix B) except that it features memory-to-memory operations unlike the usual RISC processors.

2.1.1 Data Types

PERL processor is a 64-bit architecture and supports the following data types upto 64 bits.

- Integer data types

1. Byte, 8 bits.
2. Half word, 2 bytes.
3. Word, 4 bytes.
4. Long word, 8 bytes.

- Floating-point data types
  1. Single precision floating point number, 4 bytes.
  2. Double precision floating point number, 8 bytes.

All integers are supported in both, signed and unsigned forms. Three bits for each of the three operands are provided in the opcode to encode their data types. For integer instructions, most significant bit in this indicates signed (1) or unsigned (0) arithmetic. Other two bits represent the data size. For floating point instructions, only two values are used. A value of 4 represents single precision while a value of 5 represents double precision arithmatic.

2.1.2 Addressing Modes

PERL architecture supports the following four addressing modes for each of the operands. Two bits per operand are used to encode addressing mode.

1. Immediate. In this case the value of the operand is carried within the instruction.
2. Direct Addressing. In this case, the instruction carries the 32-bit address of the operand.
3. Memory Indirect. An instruction carries the 32-bit address of the memory, which containing the effective address of the operand. This addressing mode is primarily used for accessing array elements.

4. Base Addressing. The effective address is computed by adding 32-bit offset to the contents of base memory location. In PERL, base addresses are stored in fixed memory locations which are permanently cached. Addresses 0 to 3 are used to represent upto four base addresses. A short hand notation is used to encode these memory addresses (32 bits) using two bits. Base addressing mode is primarily used for accessing local variables of a function allocated on the stack. The C compiler [Kum97] for PERL currently uses only the first two locations, one for the *stack pointer* (SP) and another for the *frame pointer* (FP).

   For jump instructions, only two addressing modes are supported. An immediate addressing mode provides the absolute address while the direct mode provides the PC relative jumps.

2.1.3 Instruction Format

All instructions in PERL architecture are coded in 128 bits. An instruction is divided into four parts, each of size 32 bits (see figure 1). The first 32 bits of the instruction are used to specify data types, addressing modes, base pointers for base addressing and the opcode (see figure 2).

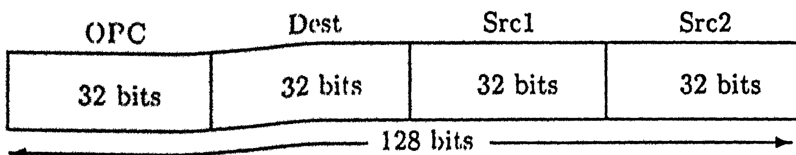


Figure 1: Instruction Format

Other 32-bit words are used to indicate destination and each of the two operands (see figure 1).

Figure 2 gives the instruction encoding. Various fields in this instruction are as follows.

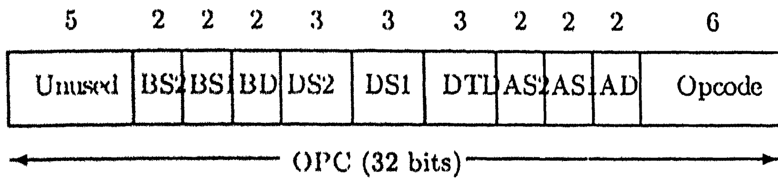


Figure 2: Instruction Encoding

- Opcode. These 6 bits refer to the opcode of the instruction. In our case, 6 bits are sufficient to have a working instruction set as outlined in the next section.
- AD, AS1 and AS2. Each of these fields is 2 bits. These indicate the addressing mode of the three operands, namely, destination, source 1 and source 2.
- DTD, DT1, DT2. Each of these fields is 3 bits. These indicate the data type of the three operands, namely, destination, source 1 and source 2.
- BD, BS1 and BS2. If operand is addressed using a base, the corresponding base address is given by these fields. Each of these is two bit field and is a short hand notation for the memory addresses 0, 1, 2 and 3.

2.1.4 Instruction Types

Following types of instructions are supported.

1. Integer arithmetic instructions
2. Logical and shift instructions
3. Floating-point instructions
4. Flow control Instructions
5. Miscellaneous

Integer arithmetic instructions

The PERL has four arithmetic instructions corresponding to four basic arithmetic operations. The instructions are listed in table 1. Each instruction can be suffixed by a data length tag if all operands are of same size. For example, `addb4 M1, M2, M3` is a short hand notation of `add M1:b4, M2:b4, M3:b4`.

Mnemonic	Operation
ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide

Table 1: Integer Arithmetic Instructions

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Logical and shift instructions

The PERL RISC has three logical and three shift instructions as listed in table 2.

Mnemonic	Operation
AND	Logical AND
OR	Logical OR
XOR	Exclusive OR
SLL	Shift left logical
SRA	Shift right arithmetic
SRL	Shift right logical

Table 2: Logical and Shift Instructions

Floating-point instructions

All arithmetic instructions (as given in table 1) are valid for floating point data types also. The instructions are qualified using a suffix to the mnemonic. The valid suffixes are `F4` and `F8` representing single precision and double precision floating-point data types respectively. For example, `addf4 M1, M2, M3` represents floating point addition.

Flow control Instructions

Flow control instructions include conditional and unconditional branch instructions (see table 3). In conditional jumps, *cond* can be either equality or inequality condition, i.e. `je` and `jne`.

Mnemonic	Operands	Operation
J	dest, src	jump, and store new *PC* in src
J*cond*	des, src1, src2	Jump to des if src1 *cond* src2

Table 3: Integer Control Instructions

Miscellaneous

A *TRAP* instruction is provided, which is mainly used for handling system calls. The TRAP instruction has only single operand, the trap number.

2.2 Pipeline

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. In a computer pipeline, each step in the pipeline completes a part of an instruction.

PERL architecture comprises of a five stage pipeline (figure 3). The five pipeline stages are as follows.

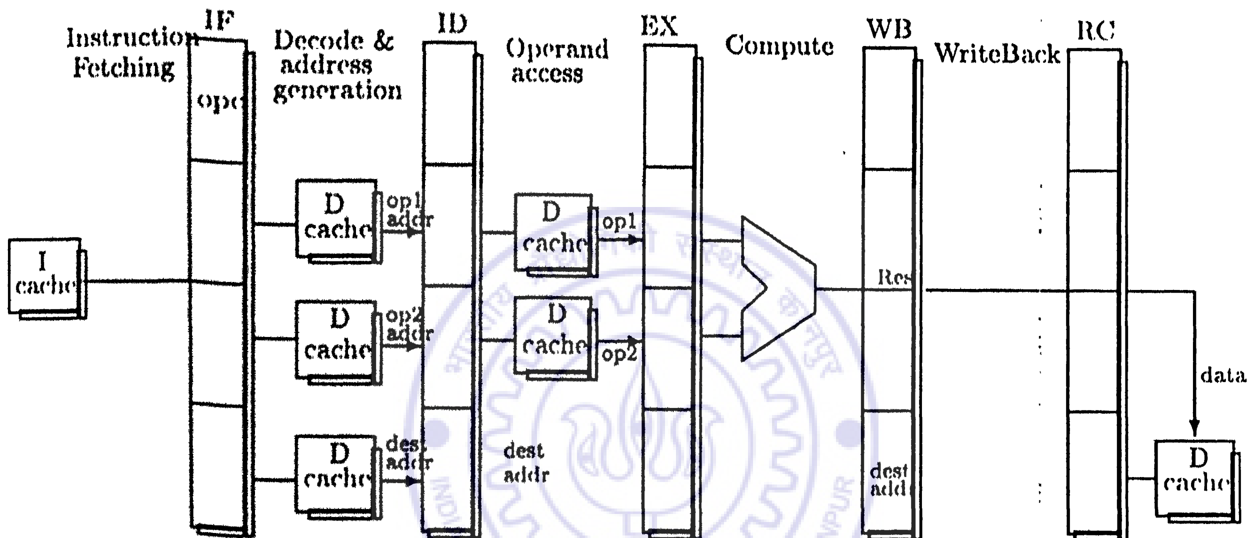


Figure 3: Processor Pipeline

2.2.1 Fetch Stage

This stage takes instructions from the instruction cache and places them in an instruction queue. Branches create problems that hinder the fetch mechanism as the instructions fetch depends on the outcome of the branch execution. The unconditional direct branches do not require any prediction as the outcome is known immediatly. The conditional branches or unconditional indirect branches need the prediction.

PERL processor fetches multiple instructions per clock cycle. Branches may also hinder fetching multiple instructions per clock cycle as the branches and the target instructions may be mis-aligned with the cache block. This way many non-useful instructions may be fetched.

2.2.2 Decode Stage

This stage takes instructions from the instruction queue, decodes and dispatches them into their appropriate operation unit: integer or floating point. The processor can decode multiple instructions per clock cycle. Each decoded instruction is assigned an entry in the reorder buffer of its operational unit where it is placed along with its destination location identifier. Once a reorder buffer entry is created for an instruction, it enters the central instruction window of its operational unit from which it gets issued.

This stage is implemented as two different stages in the pipeline. This is done because instruction set supports indirect or based addressing modes also. In such cases, the first stage computes the effective address. A memory access is needed to get the indirect address. The reorder buffer is also searched as the address forwarding is needed in this stage. The second stage uses the effective address to read the operands from the memory. The reorder buffer is searched as the data forwarding is done in this stage also.

2.2.3 Execute Stage

This stage selects the instructions which have valid operands and for whom the functional units are available. This is done by examining the instruction window. When two instructions conflict for the same functional unit, the selection is made based on the age of the instruction in the window: the older one has the higher priority.

For a branch instruction, outcomes are known at this stage. If a branch prediction happens to be incorrect, the instructions following the branch in both reorders buffers are flushed. In this case, the branch target buffer is updated with new prediction information.

2.2.4 Write Back Stage

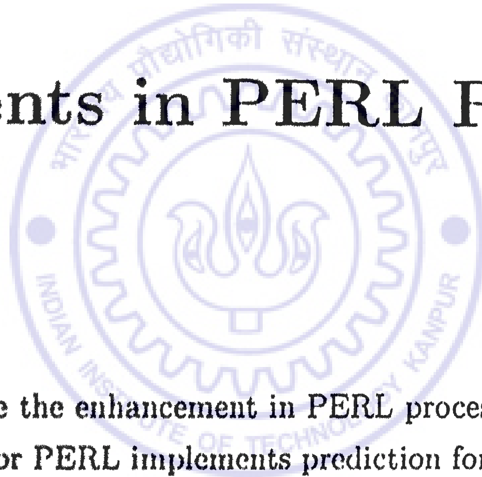
This stage identifies the reorder buffer entries whose results have been computed and validates them. The results of instructions following an incorrect branch prediction are invalidated. The valid results are also forwarded to other instructions in the pipeline which need them. The write back logic also frees the corresponding functional unit.

2.2.5 Result Commit Stage

The results that have been validated during the write back stage are sent to the corresponding memory locations during this stage. The writes are processed in order from the head to the tail of the reorder buffer until an instruction is found with an incomplete result. The completed instructions are removed from the reorder buffer. Invalidated instructions that follow a wrong branch prediction are simply discarded from the reorder buffer.

Chapter 3

Enhancements in PERL RISC processor



In this chapter, we describe the enhancement in PERL processor. The existing SuperSIM [Bal97] simulator for PERL implements prediction for conditional branches to reduce branch penalty. We discussed the prediction techniques for all indirect jumps. Further, extended address forwarding is also implemented in our work to reduce stalls due to non-availability of addresses to the instructions.

3.1 Existing Branch Prediction Scheme

In the existing simulator, branch prediction is used for conditional branches using branch target buffer (BTB). There is no prediction for unconditional indirect jumps. Further, PERL processor fetches multiple instructions per clock cycle even if branch is in the middle of cache block. For example, if PERL is configured for fetching 4 instructions per clock cycle and branch is the first instruction fetched, it fetches remaining 3 instructions after computing the target (actual or predicted) in the single clock cycle. This feature, however, is not possible in a hardware implementation. Moreover, target for indirect jumps is computed in a single clock cycle (at the fetch

stage itself) which in the hardware implementation will certainly need more than one clock cycle. This is simply clear with an example instruction `j -8(sp), #0`. This instruction is an indirect jump instrction which computes the effective address by adding -8 with the contents of SP and then it uses the effective address to read the actual target. It is clear that all this functionality and continution of more instructions can not be done in a single cycle.

To solve this problem, we incorporate prediction for indirect branches in order to reduce branch penalty.

3.2 Prediction techniques for Indirect Jumps

Indirect jumps lead to pipeline stalls as the destination is not known immediatly. Some efficient techniques are studied to avoid stalls due to indirect jumps. We experimented with the following prediction techniques for indirect jumps, any one of which can be incorporated in the simulator, we also compare the performance of these techniques.

1. Branch Target Buffer.

2. Branch Target Buffer with pair of stacks.

3. Stack Extension.

3.2.1 Branch Target Buffer (BTB)

Prediction for conditional branches is implemented by BTB in the existing simulator. We extend it to predict indirect jumps. BTB uses the old history for predicted target. BTB has the following three fields.

-Instruction address(PC).

-Predicted target address.

-Prediction state bit.

When indirect jump instructions are executed first time, prediction can not be applied as BTB does not have any history for these instructions. However BTB is updated after the execution when target address becomes known.

Second time, when an indirect jumps is executed, the BTB is searched with PC value as the key. If it results in a hit in the BTB, predicted target address field in the BTB is taken as the predicted target and the instruction fetch is continued. After the execution when the actual target becomes known, the predicted target is compared with the actual one. If the target prediction was wrong, all instruction that follow indirect jumps are flushed from both the reorder buffer and instruction queue. The program counter is changed with the actual target address and the BTB is updated.

There are two problems with this scheme.

1. The prediction can't be applied when branch instructions encounter first time as BTB does not have entries corresponding to these instructions.

2. As the prediction is based on the IP value, when a function is called from different places in the program, it prediction gets wrong as it uses the old history.

   To understand the reason of mispredictions clearly, we take the program SAMPLE given below.

```
100 j _print, -8(sp)  /* call to function print */
110 ------------
120 ------------
130 j _print, -8(sp)  /* call to function print */
140 ------------
150 ------------
    ------------
```

```
_print:
 500 ------------
 510 ------------
     ------------
 600 j -8(sp), #0       /* return from the function */
```

From address 100, program jumps to the function _print at address 500. On return from the function, BTB is updated with the target 110 corresponding to address 600. From address 130, program again jumps to the same function. On return from _print, BTB is searched at fetch stage. The processor uses predicted target address 110 corresponding to address 600 in the BTB and starts fetching instructions from the wrong location. The correct target is 140.

BTB with pair of stacks [KE91] given in following section avoids the mispredictions due to this problem.

3.2.2 BTB with pair of stacks

This technique needs pair of stacks along with the BTB. The idea behind this technique is to modify the old history of the BTB before its use. Let us say that two stacks are named S1 and S2. The steps used in accessing the predicted target are as follows.

When a jump instruction is used to call a procedure, target address (i.e. the address of the function) and the next sequential address (i.e. the return address) are pushed onto S1 and S2 respectively.

When return instruction executes first time, the computed target is compared with the top of S2. If both are same, predicted target address field in the BTB is updated with the top of the S1 and the prediction state bit is turned on (it denotes that the predicted target will be taken from the stack). Both stacks are poped also.

When next time, same call instruction executes, the target (i.e. address of the function) and the next sequential address (i.e. the return address) are pushed onto

S1 and S2 respectively.

Upon the execution of corresponding return instruction, the BTB is searched. This time search results a hit and therefore, prediction state bit is checked. If it is 0, predicted target is taken as the predicted target address field in the BTB (the usual prediction). However, if it is 1 then the target address field in the BTB is compared with the top of the stack S1. If both are same, the top of S2 is taken as the predicted target.

The pair of stacks can store the history of a number of call instructions. However if the number of called procedures becomes large, the stack gets full. In this case, we fall back to the simple prediction scheme used by the BTB.

The recursive programs create one problem. They grow the stacks with the same value. To overcome with this problem, a counter is used with the stacks. If the same value is pushed onto stacks, it just increments the count corresponding to the stack's entry. The count is decremented before removing an entry from the stack and if it becomes 0, then only the entry is removed.

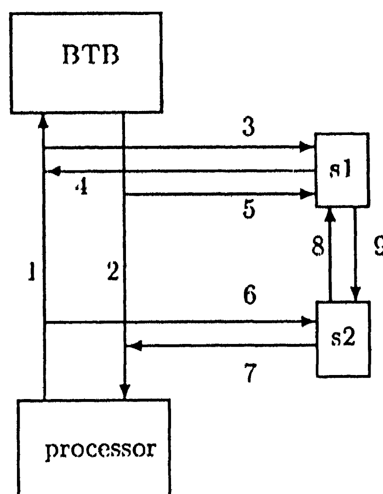


Figure 4: Call/Return Stack Design

To understand it clearly, we take the program SAMPLE given in the previous section.

When a call to function `_print` at address 100 executes, the target address of the function `_print` (500) is pushed onto S1 via path 3 and the return address (110, the address that is sequential to the call instruction) is pushed onto S2 via path 6 shown in figure 4.

When return instruction encounters first time at 600, processor stalls the pipeline till target is computed. As target gets ready, the address 600 and 110 are sent to the BTB along path 1, branch address and branch target respectively. In parallel, 110 is sent to stack S2 along path 6 to see if S2 has an entry and its corresponding entry in S1 (in this case 500) is found via path 8 and sent along path 4 where it replaces the target address on path 1. Then the entry in the BTB has a branch address of 600 and a target address of 500. The prediction state bit in the BTB is turned on to denote that the actual target is in the S2.

When the same call function at address 130 executes again, the target address (500) is pushed onto stack S1 on path 3 and the return address (140) is pushed on stack S2 on path 6.

Next time when the return encounters at 600, the BTB finds an entry for address 600, and that entry will have the prediction state bit turned on. The prediction associated with the entry (branch address set to 600 and branch target set to 500) is sent to path 2, and since the prediction state bit is on, the target field is sent to stack S1 on path 5 to see if S1 has an entry for 500. In this case, S1 does and so the corresponding entry in S2 (140) is identified via path 9 and is put on path 2. (All entries on the stack are compared in parallel.)

The prediction that is made for address 600 will then have the target 140.

This scheme is better than the first one but it can't avoid the first problem. Prediction can not be applied when return encounters first time. The third scheme stack extension given below avoids both the problems that cause mispredictions.

3.2.3 Stack Extension

This is very simple technique and performs better than the two techniqes discussed above. In this sheme, a separate on-chip stack is used to maintain the target addresses (return addresses). When a call instruction executes, the next sequential address is pushed on to on-chip stack. When return instruction encounters, the top of the stack is taken as the target. The top entry is poped from the stack.

Because on-chip stack is of limited size and some programs (i.e. recursive) may overflow the stack, the overflow entries should be maintained somewhere in the main memory. A separate stack is used in the main memory to keep the overflow entries of on-chip stack. Both on-chip and main memory stacks are divided into blocks. the block is the transfer unit between on-chip and main memory stacks.

Because the speed of pushing entries onto on-chip stack may be faster than the speed of swapping the blocks from the on-chip stack into main memory stack, the blocks should be swapped from the on-chip stack before it gets full. The swapping of the block depends on the availability of the bus between on-chip and main memory stacks.

3.3 Reasons of stalls and their reductions

The existing simulator stalls the pipeline due to non-availability of adresses to the instructions. However, stalls due to this can be reduced using extended address forwarding. All the cases where existing simulator stalls the pipeline and their reductions using extended address forwarding is given in the following subsection.

3.3.1 Reasons of stalls

The following three reasons are clearly mentioned where existing simulator inserts stalls.

1. During the entry point to a function. This is because, the value of SP is changed to create a new frame and the temporaries used during this procedure call are stored in this frame. We have commands like the following:

```
;;Establish new frame pointer
addb4 fp, #0, sp
;;Adjust stack pointer
addb4 sp, sp, #-24
;;Save temporary locations
addb4 0(sp), t1, #0
addb4 4(sp), t2, #0
```

   As the destination address of the third addition instruction is not known till the second add is computed, fetch is stalled after second add instruction. Such code appears usually at the entry point of function calls.

2. During exit from a function. In PERL, procedure calls are handled by unconditional jumps. This jump actually saves the return address in the stack. At exit from function calls, we encounter instructions like:

```
;;Restore stack pointer
addb4 sp, #0, fp
;;Restore frame pointer
addb4 fp, -4(fp), #0
;;Return
j -8(sp), #0
```

   The jump instruction restores the PC value so that execution returns to the callee and fetch proceeds from that address. But we do not know what should be loaded into PC till the stack pointer is restored. Thus the fetch is stalled.

3. Due to indirection.

```
addb4 t1, t2, t1
```

```
addb4 (t1), #0, t3
addb4 t2, t2, #1
```

In this case also fetch is stalled after the second add instruction. This is because, the destination address of second add is known only after the first add is executed and all instruction that follow the second add stall because we do not know into which location the second instruction writes. Here, problem is similar to case 1 but there it is based addressing and here it is indirect addressing.

3.3.2 Reductions in stalls

All the three cases almost have the same problem. An instruction whose destination uses based or indirect addressing mode, can not proceed in the pipeline till the instrucion modifying the base or indirect address exits the pipeline. However, stalls due to these reasons can be reduced using address forwarding. We are taking the first case and proceed the instructions in the pipoeline with following steps.

```
;;Establish new frame pointer
addb4 fp, #0, sp
;;Adjust stack pointer
addb4 sp, sp, #-24
;;Save temporary locations
addb4 0(sp), t1, #0
addb4 4(sp), t2, #0
```

In this case, when the third addition instruction goes to decode stage, it searches the reorder buffer with address as SP.

If there is an entry corresponding to SP then the decoding logic checks it whether the value is valid or not.

If the value is valid, it computes the destination address by adding the value of the reorder buffer with the offset and following instructions proceed in the pipeline as usual.

If the entry in the reorder buffer is not valid then it simply takes the reorder buffer entry for destination address and sets the destination address valid bit to zero.

When the second addition instruction computs the result, it is forwarded to the third addition instruction. Now the third addition instruction computes the destination address by adding offset and the following instructions proceed in the pipeline.

Address forwarding reduces the stalls significantly and improves the performance of the PERL processor.

SuperSIM version 1.1 and **SuperSIM version 1.0** simulates the PERL processor with and without prediction for indirect jumps and extended address forwarding respectively. The performance of **SuperSIM version 1.0** is compared with **SuperSIM version 1.1** in chapter 4.

Chapter 4

Simulation Results, Conclusions and Future Work

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

In this chapter, we compare the results obtained by SuperSIM version 1.1 with the existing simulator SuperSIM version 1.0 and with similar results obtained for current day RISC processors. SuperSIM version 1.1 simulates the PERL processor incorporating prediction for indirect jumps and extended address forwarding as discussed in chapter 3. To show the effect of various prediction schemes (see chapter 3) for indirect jumps, the results obtained with different prediction schemes are compared. We compare the results obtained by SuperSIM version 1.1 with DLX [PH94] and DEC Alpha 21064. DLX [PH94] is a hypothetical RISC architecture which can be taken as a representative of all current day RISC processors. The results for DLX are obtained using **SuperDLX** [Mou93], a simulator for DLX. The performance matrics for DEC Alpha 21064 are obtained using the ATOM [Dig93]. Atom is a performance analysis tool which is part of the DEC OSF software kit. As we used Atom on our DEC 21064, which is 2-issue superscalar, we configured **SuperSIM version 1.1** and **SuperDLX** to simulate instructions in 2-issue mode. The results for 4-issue are available only for **PERL** and **DLX**.

The compilers for DLX and DEC Alpha are very efficient and perform lot of optimizations, whereas the compiler for PERL does not perform any kind of machine

dependent optimizations. So the results are compared with and without compiler optimizations.

The performace matrics of interest are:

- IC: The dynamic instruction count.
- CC: Total number of clock cycles required to execute the program.
- PS: Number of cycles the pipe is stalled.
- WP: Percentage of wrong predictions due to indirect jumps .

4.1 Benchmark Programs

The programs used for simulation are:

Permute. This is heavily recursive program which, given an array of n integers, prints all n! permutations. For simulations we have taken of n as 5.

Matmul. An integer matrix multiplication program. The results are obtained for matrices of size 32x32.

Tts. This is a time table scheduler program. Given a list of courses and preferences for timing for allotting slots to the course and a given set of class rooms, the program uses a heuristic approach to get the best optimized output.

relax and across. These two programs are taken from nasa test suite for parallelizing compilers. Both of them contain nested do loops and operate on vectors.

1. Permutation. This is a heavily recursive program. Table 4 compares the results of PERL processor obtained by **SuperSIM verion 1.0** with **SuperSIM version 1.1**. Because the program is heavily recursive, stalls due to case 1 and 2 (see reasons of stalls in chapter 3) occur more often in SuperSIM version

CENTRAL LIBRARY
I. I. T. KANPUR

Architecture	*version 1.0*		*version 1.1*	
	CC	PS	CC	PS
PERL (2 issue)	13522	7147	8059	249
PERL (4 issue)	11655	7591	6094	698

Table 4: Comparison for Permutation Benchmark for different version

Architecture	*without pred.*		*pred. with BTB*		*with stacks pair*		*with stack extn.*	
	CC	WP	CC	WP	CC	WP	CC	WP
PERL (2)	9590	100	8658	53	8075	0	8059	0
PERL (4)	7592	100	6693	53	6110	0	6094	0

Table 5: Comparison for Permutation Benchmark with different prediction schemes for indirect jumps

1.0. Table 4 shows that *the enhanced version* reduces the stalls and improves the performance by approx. 40-46 percentage.

Table 5 shows the results of PERL processor obtained by **SuperSIM version 1.1** with different prediction schemes for indirect jumps. Table 5 shows that prediction for indirect jumps gives a significant improvement in PERL processor. In permutation program, same functions are called from different places in the program. So the BTB gives the wrong predicted target as it uses the old history. The second scheme, BTB with pair of stacks avoids mispredictions due to this reasons and improves the performance significantly. With stack extention scheme, prediction is also applied when indirect jumps execute first time. So it takes few less clock cycles in comparision to stacks pair scheme.

Table 6 compares the results of PERL processor obtained by SuperSIM version 1.1 with the results obtained for DLX and Alpha21064. Table 6 shows that PERL processor takes fewer clock cycles than DLX and Alpha21064. Even though DLX and 21064 represent the same class of RISC architectures, DLX executes more instructions than 21064. The reason is to be the code generated by the compilers. The C compiler for DLX generates a "nop" instruction

Architecture	*with compiler opt.*			*without compiler opt.*		
	IC	CC	PS	IC	CC	PS
PERL (2 issue)	10525	8059	249	12021	9464	339
PERL (4 issue)	10525	6094	698	12021	7590	1721
DLX (2 issue)	14392	8709	894	16032	10226	950
DLX (4 issue)	14392	8701	4118	16032	9799	4199
Alpha 21064	9661	8731	1326	9767	11745	4447

Table 6: Comparison for Permutation Benchmark with other RISC

after every load, store and branch instruction. This accounts for the extra instructions executed by DLX. A 4-issue **DLX** performs almost same as a 2-issue machine. But the 4-issue **PERL** processor performs better than the 2-issue version for the same machine configuration, as it takes fewer clock cycles. This shows that **PERL RISC** gives more scope for multi-issue.

The results shows that PERL processor performs better than other RISC processor.

2. Matrix Multiplication. This is another program which we used for simulation. This program does not have call/return instructions. So, the prediction for indirect jumps does not matter here. In this program, the stalls are only due to case 3 (see reasons of stalls in chapter 3). Table 7 compares the results of PERL processor obtained by **SuperSIM verion 1.0** with **SuperSIM version 1.1**.

Architecture	*version 1.0*		*version 1.1*	
	CC	PS	CC	PS
PERL (2 issue)	469535	127061	347706	4716
PERL (4 issue)	384473	178327	214549	5965

Table 7: Comparison for multiplication Benchmark for different version

Table 8 shows that 4-issue version of PERL processor performs better than 4-issue version of DLX with or without compiler optimization. The 2-issue version of PERL RISC performs slightly worse than 2-issue version of DLX

with compiler optimization because compiler for PERL RISC does not perform any kind of machine dependent optimizations. But the 2-issue version of PERL RISC performs far better than 2-issue version of DLX without compiler optimization.

Architecture	*with compiler opt.*			*without compiler opt.*		
	IC	CC	PS	IC	CC	PS
PERL (2 issue)	677413	347706	4716	1077797	548387	5585
PERL (4 issue)	677413	214549	5965	1077797	315906	7958
DLX (2-issue)	681760	345275	1104	1446734	744664	1106
DLX (4-issue)	681760	344186	1109	1446734	743060	1110
Alpha 21064	459296	980520	656649	1194567	1790781	804495

Table 8: Comparison for Matrix Multiplication Benchmark with other RISC processors

Table 8 shows the great disparity in the number of instructions executed for DLX and 21064. The reason is the "s4addq" instruction in Alpha which performs scaled addition.

```
s4addq s_reg1,s_reg2/imm,d_reg
```

The above instruction computes the sum of two signed 32-bit values. This instruction scales (multiplies) the contents of s_reg1 by 4 and then adds the contents of s_reg2 or imm. The result is stored in d_reg. Such a powerful instruction is provided to access the elements of an array. The address of element a_i, for example can be calculated within a loop by this single instruction, whereas **DLX** and **PERL** need one add and one multiply instruction. As the matrix multiplication program accesses the elements of two-dimensional arrays, this instruction in Alpha is very useful.

Table 8 shows that **PERL** processor performs better than **DLX** and **Alpha21064**.

3. Time table scheduler. This program is also used for simulation. Table 9 compares the results of PERL processor obtained by **SuperSIM verion 1.0**

with **SuperSIM version 1.1**. Because the prediction for indirect jumps is used in version 1.1, it performs better than version 1.0.

Architecture	*version 1.0*		*version 1.1*	
	CC	PS	CC	PS
PERL (2-issue)	3940879	1709797	3695092	1400934
PERL (4-issue)	3595616	2143987	2976955	1498128

Table 9: Comparison for Time Table Scheduler Benchmark for different version

Architecture	*without pred.*		*pred. with BTB*		*with stacks pair*		*with stack extn.*	
	CC	WP	CC	WP	CC	WP	CC	WP
PERL (2)	3989430	100	3734849	26.3	3695092	0	3693942	0
PERL (4)	3386307	100	3131860	26.3	2976955	0	2975675	0

Table 10: Comparison for Time Table Scheduler Benchmark for various prediction techniques

In this program also, same functions are called from different places. So the BTB gives the wrong predicted target as it uses the old history. BTB with pair of stacks avoids mispredictions due to this reasons and improves the performance significantly. With stack extention, prediction is also applied when indirect jumps execute first time. So it takes few less clock cycles in comparision to stacks pair scheme.

The C compiler [Kum97] for PERL RISC is not optimizing the code for this program. So Table 11 presents the results only for the unoptimized code for all three machines. The reason of disparity in the numbers of instructions executed for DLX and Alpha21064 is "s4addq" and "s8addq" instructions in Alpha21064 and the "nop" instructions generated by the compiler for DLX. Table 11 shows that **PERL** processor performs better or as good as **DLX** and **Alpha21064**.

4. across. This program performs operation on vector. The instructions are

Architecture	*without compiler opt.*		
	IC	CC	PS
PERL RISC (2-issue)	3669564	3695092	1400934
PERL RISC (4-issue)	3669564	2976955	1498128
DLX (2-issue)	7172897	3772762	176408
DLX (4-issue)	7545677	3672534	1293016
Alpha 21064	3060510	2885116	469223

Table 11: Comparison for Time-table Scheduler Benchmark with other RISC

heavily dependent within the loop. Table 12 compares the results obtained by **SuperSIM verion 1.0** with **SuperSIM version 1.1**.

Architecture	*version 1.0*		*version 1.1*	
	CC	PS	CC	PS
PERL (2-issue)	5996	4391	3110	1467
PERL (4-issue)	5796	4693	3105	1983

Table 12: Comparison for across Benchmark for different version

Because the instructions are heavily dependent within the loop, stalls due to case 3 (see reasons of stalls in chapter 3) occurs more often in version 1.0. The *enhanced version* reduces the stalls and performs better as it takes approx. 40-46 percentage less clock cycles. As there are no call/retun instructions in this program, the prediction for indirect jumps does not matter here also. Table 12 shows a significant improvement in PERL processor with enhancement.

Table 13 shows that PERL processor performs worse than DLX and Alpha21064 when compiler optimization is done. Again the reason is to be the code generated by compilers. The compilers for DLX and DEC Alpha are very efficient and perform lot of optimizations, whereas the compiler for PERL RISC does not perform any kind of machine dependent optimizations. But with unoptimized code for all three machines, PERL RISC performs better than other two machines.

Architecture	*with compiler opt.*			*without compiler opt.*		
	IC	CC	PS	IC	CC	PS
PERL (2-issue)	2804	3110	1467	5975	4707	1382
PERL (4-issue)	2804	3105	1983	5975	4706	2885
DLX (2-issue)	4634	2337	9	11355	6202	194
DLX (4-issue)	4634	2336	13	11355	5707	19
Alpha 21064	2861	2506	433	8105	9690	3918

Table 13: Comparison for across Benchmark with other RISC

Again the reason of disparity in the numbers of instructions executed for DLX and Alpha21064 is "s4addq" instructions in Alpha21064 and the "nop" instructions generated by the compiler for DLX.

Table 13 shows that **PERL** processor performs better or as good as **DLX** and **Alpha21064**.

5. relax. This is another benchmark from nasa test suit. Table 14 compares the results of PERL processor obtained by **SuperSIM verion 1.0** with **SuperSIM version 1.1**. In this program also, pipeline stalls due to case 3 (see reasons of stalls in chapter 3) occur often in version 1.0. Table 14 shows that the *enhanced version* reduces the stalls and perfoms better than *previous version*.

Architecture	*version 1.0*		*version 1.1*	
	CC	PS	CC	PS
PERL (2-issue)	2084191	99119	2044982	57829
PERL (4-issue)	1882311	599023	1313903	160941

Table 14: Comparison for relax Benchmark for different version

Again the compiler for PERL RISC is not optimizing the code for this program also. So Table 15 presents the results only for the unoptimized code for all three machines.

Again the reason of disparity in the numbers of instructions executed for DLX and Alpha21064 is "s4addq" instructions in Alpha21064 and the "nop" instructions generated by the compiler for DLX. Table 15 shows that PERL RISC performs better than DLX and Alpha21064.

Architecture	*without compiler opt.*		
	IC	CC	PS
PERL (2-issue)	3858828	2044982	57829
PERL (4-issue)	3858828	1313903	160941
DLX (2-issue)	11875411	6435276	283342
DLX (4-issue)	11875411	6432039	289230
Alpha 21064	3987102	3920541	724782

Table 15: Comparison for relax Benchmark with other RISC

From the above statistics, it should be clear that *the enhanced version* of **PERL RISC** reduces a lot of stalls and performs better than the previous version as it takes fewer clock cycles. The results given above also show, **PERL RISC** performs better or at least as good as existing RISC machines. The C compiler [Kum97] for **PERL RISC** is not performing any kind of machine dependent optimizations while the compilers for DLX and Dec Alpha are performing lot of optimizations. The results obtained for **DLX** and **Alpha 21064** show that compiler optimizations give a great improvement in performance. The compiler optimization phase should be modified to produce fully optimized code for **PERL RISC** and then compare the results of simulating that code with those of existing machines.

4.2 Conclusions

In this thesis, **PERL** processor is enhanced by incorporating prediction for indirect jumps. Moreover, the pipeline stalls due to non-availability of addresses are reduced by implementing extended address forwarding. From the results obtained, it is clear that *the enhanced version* of **PERL** processor performs

better than *the previous version*. The results also shows that **PERL RISC** performs better or at least as good as existing RISC machines.

4.3 Future Directions

A C compiler for PERL RISC is built on top of GCC and generated code is not fully optimized code for PERL RISC. The reason being that GCC is a compiler written for machines with many registers. The notion that registers are faster than memory is built into compiler. The compiler optimization phase should be modified to produce fully optimized code for PERL RISC.

The prediction mechanism for conditional branches permits to start execution of one conditional path of a branch. It will be interesting to experiment branch speculation on both paths of the branch and compare the two branch processing techniques. This type of branch speculation is certainly very complex to implement; buffering of instructions has to be provided to support execution of two conditional streams of instructions at a time, which implies some kind of replication of the reorder buffer and instruction window. Duplication of hardware elements has to stay within realistic limits.

Appendix A

User Manual

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

NAME

SuperSIM - Simulator for assembly programs of PERL architecture.

SYNOPSIS

`supersim`

OPTIONS

[-f filename] [-bp] [-bi] [-bS] [-bs] [-ms#]

-f filename machine description file specifying user parameters.

-bp Turn on the conditional branch prediction scheme.

-bi Turn on the conditional branch prediction scheme and the prediction for indirect jumps using BTB.

-bS Turn on the conditional branch prediction scheme and the prediction for indirect jumps using BTB with pair of stacks.

-bs Turn on the conditional branch prediction scheme and the prediction for indirect jumps using stack extension.

-ms# Specify the size of the memory.

DESCRIPTION

SuperSIM is an interactive program that loads assembly programs and simulates the operation of PERL processor on those programs. Once started, **SuperSIM** loops forever reading commands from standard input and printing results on standard output.

NUMBERS

Whenever **SuperSIM** reads a number, it will accept the number in either decimal notation, hexadecimal notation if the first two characters of the number are **0x** (e.g. 0x3acf), or octal notation if the first character is **0** (e.g. 0342).

ADDRESS EXPRESSIONS

Many of **SuperSIM**'s commands take as input an expression identifying a register or memory location. Symbolic expressions may be used to specify memory addresses. The simplest form of such an expression is a number, which is interpreted as a memory address. More generally, address expressions may consist of numbers, symbols (which must be defined in the assembly files currently loaded), the operators *, /, %, +, −, <<, >>, &, |, and ↑ (which have the same meanings and precedences as in C), and parentheses for grouping.

COMMANDS

SuperSIM provides the following application-specific commands:

`asm` *file file file* ...

Read each of the given *files*. Treat them as assembly language files for PERL machine, and load memory as indicated in the files. Code (text) is normally loaded starting at address 0x100, but the **codeStart** variable may be used to set a different starting address. Data is normally loaded starting at address 0x8000, but a different starting address may be specified in the **dataStart** variable. The return value is either an empty string or an error message describing problems in reading the files. A list of directives that the loader

understands is in a later section of this manual.

`go` [*address*]

Start simulating the machine. If *address* is given, execution starts at that memory address. Otherwise, it continues from wherever it left off previously. This command does not complete until simulated execution stops. The return value is an information string about why execution stopped and the current state of the machine.

`help` [`command`]

When followed by a command name, it gives information on the command and how to use it. Otherwise, it gives all the available commands.

`print` [`queue`] [`window`] [`reorder`] [`unit`] [`all`] [`help`]

This command displays the contents of the machine elements specified by the options at the clock cycle when simulation last stopped. Any combination of options may be selected. The options and the result of choosing them are as follows:

`queue` display the instruction queue in a table with the following column header:

- #: entry number.
- icount: dynamic instruction count.
- address: instruction address in memory.
- opcode: the instruction opcode.
- rs1: source operand 1.
- rs2: source operand 2.
- rd: register destination.

`window` print both instruction windows: integer and floatingpoint with the following column headers:

- #: instruction entry number in the window.

- opcode: instruction opcode.
- address: instruction address (Program Counter).
- rb entry: reorder buffer entry number of the instruction
- operand1: contains either the value of the firstoperand or a reorder buffer entry from where the value will be taken when computed.
- ok1: the validity field of the firstoperand: it contains 0 if the operand refers to a reorder buffer entry and 1 if the operand value is available.
- typ1: type of the firstoperand (INT: integer, FPS: floating point simple, FPD: floatingpoint double IMM: immediate). The type of the operand indicates from which reorder buffer the operand value can be taken.
- operand2: second operand, just like the firstoperand, it can contain either a value or a reorder buffer entry.
- ok2: validity field of the second operand.
- typ2: type of the second operand (same as type1).
- pred: only for branch instructions, indicates what prediction has been made at fetch stage (PT: predict taken, PNT: predict not taken).

`reorder` display the integer and the foatingpoint reorder buffer, in tables with the following column headers:

- #: instruction entry number.
- icount.
- opcode.

- loc: memory location address for the result.
- unit: functional unit where the instruction executes.
- result: instruction result when produced (some instructions such as branches produce no result).
- ok: validity of the result.
- ready: indicates at what clock cycle the computation of the result will be over.
- flsh: the flush bit; it is set to 1 when the instruction follows a wrong branch prediction. In that case, when the instruction arrives at the head of the reorder buffer, it is discarded.

unit show the latency, number, and current usage of the functional units.

help print help information on how to interpret the displayed data, for any of the above machine elements. The machine components for which information is requested must be passed as arguments to the commands. If none of the components is specified, information is provided for all of them.

reset

This command resets the superscalar machine, though keeping the parameters specified by the machine configuration file. The permits to run different simulations without quitting the simulator. After resetting the machine, the .s files will have to be reloaded using the **asm** command.

stats [-p#] [-g] [-r] [-i] [-b] [-f **filename**] [-a]

This command will dump various statistics collected during

simulation. Various options are provided to take those statistics that are necessary and also to redirect the output to a file.

`-f filename` Redirects the output to a file. The file name is specified one space after f.

`-p` Set the minimum value for percentages (percentages lower than this value will not be output). The value is specified just after p. It can be an integer or a real number.

`-g` General information display.

- Number of fetched, decoded and issued instructions.
- Number of committed instructions, writes to memory locations and useless writes. A useless write is a write to a location that could have been avoided because the writing instruction is followed in the reorder buffer by another instruction writing to the same location.
- Per cycle rates of fetch, decode, issue and commit.
- Branch information: correct and wrong prediction (if branch prediction is performed).
- Fetch stalls.
- Information on renaming
  - Renamed Operands: a table gives the distribution of the number of operands renames per clock cycle.
  - Operands searching information: a table gives the distribution of the number of operands searched (in the reorder buffer) per clock cycle.

`-i` Instruction process information:

- Instruction issue distribution table per clock cycle.
- Instruction delay distribution table: the distribution of the number of cycles that instructions have to wait in the central window before issue.
- Instruction commit distribution table, per clock cycle.

`-b` Occupancy rate tables of the instruction windows and the reorder buffers, throughout simulation.

`-a` Display all the above statistics. This is also the default when no options are given.

step [*address*]

If no *address* is given, the **step** command executes a single instruction, continuing from wherever execution previously stopped. If *address* is given, then the program counter is changed to point to *address*, and a single instruction is executed from there. In either case, the return value is an information string about the state of the machine after the single instruction has been executed.

trace on/off filename

This command is used to write the memory access trace to a file. This is later fed to the cache simulator to identify the performance of different cache configurations, like multiported caches, interleaved cache banks etc. When trace is turned on, the type of memory access, address accessed and clock cycle are written to the file. The trace can also be turned off selectively.

quit

Exit the simulator.

Appendix B

Commom RISC features

(a) All instructions have fixed size and simple format.

(b) These machines have only a handful (upto 4) of addressing modes.

(c) They use hardwired logic for control.

(d) Only Load/Store instructions access memory.

(e) All arithmetic operations are performed on registers with no instruction combining Load/Store with arithmetic.

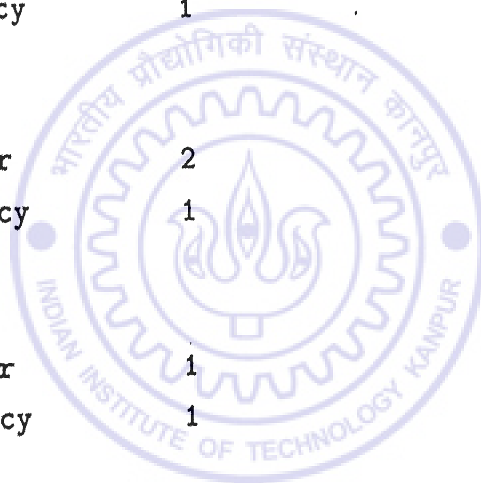
(f) They aim at more performance from current compiler technology.

Appendix C

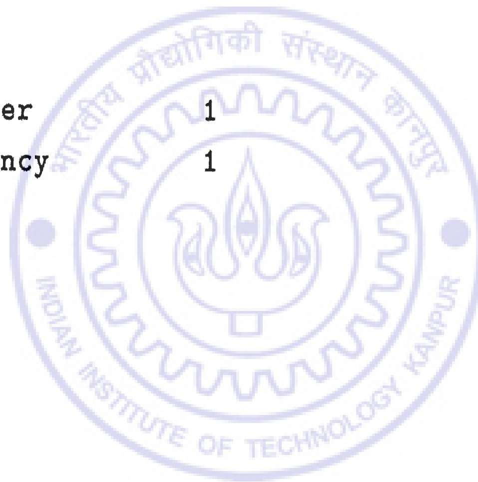
An Example Machine Description File

```
Instructions_process_per_cycle

        fetch           4
        commit          4
        decode          4

Memory
        size            262144
        latency         1
        accesses        4

Reorder_Buffer_sizes
        integer         40
        float           20

Instruction_Window_sizes
        integer         40
        float           20

Instruction_Queue_size  40
```

```
Branch_Buffer_size      111

Integer_Functional_Units

alu
        number          4
        latency         1

shift
        number          2
        latency         1

branch
        number          1
        latency         1

mult
        number          2
        latency         5

div
        number          2
        latency         10

Floating_Point_Units

add
        number          4
        latency         2
```



```
mult
        number      4
        latency     5

div
        number      2
        latency     10

branch
        number      1
        latency     1
```



References

[AAD90] D. Alpert, A. Averbuch, and O. Danieli. Performance Comparison of Load/Store and Symmetric Instruction set Architectures. In *ACM SIGARCH*, volume 18, pages 172 181, 1990.

[Bal97] T. S. Balaji. Design and Simulation of PERL RISC. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, January 1997.

[BUOO94] Brad Burgers, Nasr Ullah, Peter Van Overen, and Deene Ogden. The PowerPC 603 Microprocessor. *Communications of the ACM*, 37:34 46, June 1994.

[Dig93] Digital Equipment Corporation. *Atom Reference Manual*, Dec 1993.

[FMM87] M.J. Flynn, J.M. Mitchell, and J.M. Mulder. And Now a case for more Complex Instruction Sets. *IEEE Computer*, pages 71–83, Sep 1987.

[KE91] D.R. Kaeli and P.G. Emma. Data prefetching and branch prediction. In *ACM SIGARCH*, volume 19, pages 72–81, May 1991.

[Kum97] G.V. Ramana Kumar. Cache Simulation and Porting gcc to PERL-RISC. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, Jan 1997.

[LB92] L.B.Hostetler and B.Mirtich. DLXsim-A simulator for DLX. Technical report, University of California, Berkeley, Jan 1992.

[McL93] Edward McLeelan. The Alpha AXP Architecture and 21064 Processor. *IEEE Micro*, pages 36–47, June 1993.

[Mey78] G.J. Meyers. The evaluation of expressions in a storage-to-storage architecture. *Computer Architecture News*, 7:3:20–23, Oct 1978.

[Mou93] Cecile Moura. SuperDLX-A Generic Superscalar Simulator. Master's thesis, Advanced Compilers, Architectures and Parallel Systems Group, McGill University, May 1993.

[Pat85] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28:8–21, Jan 1985.

[PH94] David A. Patterson and John L. Hennessy. *Computer Architecture—A Quantative Approach.* Morgan Kaufmann Publishers Inc., second edition, 1994.

[Sit93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36:33–44, Feb 1993.

Kanpur.