


C Compiler and Cache Performance Studies for
PERL RISC Processor

The logo of the Indian Institute of Technology Kanpur is a circular emblem. It features a central gear with a flame-like shape in the center. The text "भारतीय प्रौद्योगिकी संस्थान कानपुर" is written in Hindi around the top inner edge, and "INDIAN INSTITUTE OF TECHNOLOGY KANPUR" is written in English around the bottom inner edge.

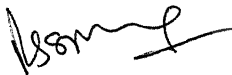
*A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology*

by
G. V. Ramana Kumar

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
February, 1997

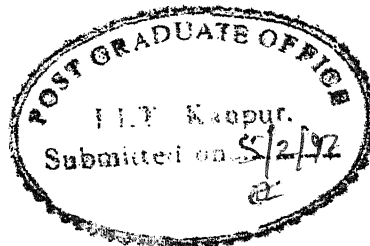
Certificate

Certified that the work contained in the thesis entitled "*C Compiler and Cache Performance Studies for PERL RISC Processor*", by Mr. G. V. Ramana Kumar, 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.

February, 1997



CENTRAL LIBRARY

ISS. No. A. 123146

CSE-1987-M-KUM-COM.



Abstract

PERL RISC, a register-less RISC architecture, embraces all the properties of RISC machines and tries to remove bottleneck found in current day RISC machines. It is designed to be a memory-to-memory architecture because of its relevance today. The availability of large on-chip caches and wider bus bandwidth to reduce memory latency supports this idea. In this thesis, A C compiler which produces assembly code of PERL RISC is developed. The compiler can perform machine independent code optimizations. The cache performance of PERL RISC is studied using trace-driven simulation. The memory bandwidth requirement of PERL RISC is investigated and effectiveness of multi-port caches and other cache bandwidth improvement techniques are discussed. The cache simulation results are compared with similar results obtained for current day RISC processors.

Acknowledgments

I, most thankfully, acknowledge the great amount of help, guidance, suggestions and encouragement given by **Dr. Rajat Moona**. I am also grateful to Mr. P. Suresh for his suggestions in the present work and Mr. T. S. Balaji for his cooperation.

I am thankful to my parents for their moral and emotional support. My sincere thanks to lab people for their help. Thanks are due to mtech@cse, A and B block people of Hall 4 and all those who made my stay at IIT Kanpur memorable.

Contents

1	Introduction	
1.1	Overview of Microprocessors	
1.2	A RISC without Load/Store Overhead	
1.3	Cache Evaluation Methodology	
1.4	Organization of the thesis	
2	Register-Less RISC	
2.1	Architecture	
2.1.1	Data Types	
2.1.2	Addressing Modes	
2.1.3	Instruction Set	
2.1.4	Pipeline	
3	Implementation of Compiler	1
3.1	Working of GNU C Compiler	10
3.2	RTL Representation	11
3.2.1	RTL Expressions	11
3.3	Machine Description	14
3.3.1	Instruction Patterns	14
3.3.2	Defining RTL Sequences for Code Generation	16
3.4	Machine Description for PERL RISC	17
3.4.1	Architecture Specification	17
3.4.2	Instruction Patterns	18

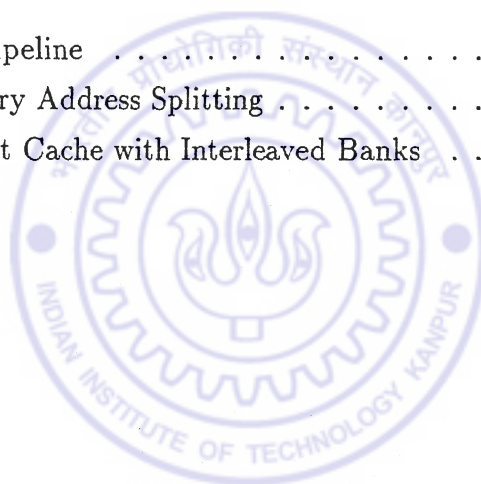
4	Cache Memories	21
4.1	Introduction	21
4.2	Aspects Of Cache Design	22
4.3	Increasing cache bandwidth	23
4.3.1	Increasing Cache Port Efficiency	24
4.3.2	Multi-ported cache	24
4.4	Case Studies	25
4.4.1	Alpha 21164	25
4.4.2	UltraSPARC	26
5	Implementation of Cache Simulator	27
5.1	Simulator Input	27
5.2	Simulator Output	28
6	Cache Simulation Results	29
6.1	Benchmark Programs	30
6.2	Memory Bandwidth Requirement	30
6.3	Effectiveness of Bandwidth Improvement Techniques	32
6.3.1	Load All Wide	32
6.3.2	Multi-port Cache	32
6.4	Primary-level Cache Performance	33
6.4.1	Instruction Cache	33
6.4.2	Data Cache Performance	34
7	Conclusions and Future Work	36
A	Cache Simulator Input Files format	37
A.1	Trace File (<i>trace</i>)	37
A.2	Cache Configuration File (<i>cache.config</i>)	37
B	A Sample Output of Compiler	39
B.1	A C Benchmark Program	39
B.2	Assembly Code Generated by Compiler	40

List of Tables

1	Integer Arithmetic Instructions	6
2	Integer Control Instructions	6
3	Logical and Shift Instructions	7
4	Comparison of Memory Reference Counts	30
5	Comparison of Memory Bandwidth Requirement	31
6	Comparison of Memory Reference Distribution	31
7	Percentage of References Satisfied by Load All Wide	32
8	Instruction Cache Miss Ratios	34
9	Comparison of Data Cache Misses for Permute	34
10	Comparison of Data Cache Misses for Matmul	35
11	Comparison of Data Cache Misses for Timetable program	35

List of Figures

1	Processor Pipeline	8
2	Main Memory Address Splitting	21
3	A Multi-Port Cache with Interleaved Banks	25



Chapter 1

Introduction

PERL RISC [Bal97] (Performance Enhanced Register-Less RISC) is a new memory-to-memory RISC architecture. The factors that influenced this processors design are: availability of faster and cheaper caches which are comparable in speed to registers, the Load/Store overhead associated with all the current RISC processors.

The Performance of PERL RISC is highly dependent on cache performance. In this thesis, we study the cache performance of PERL RISC. A C compiler for PERL RISC which aids the evaluation of this architecture is also developed.

1.1 Overview of Microprocessors

The current day general purpose microprocessors can be classified into two categories: *complex instruction set computers*(CISC) and *reduced instruction set computers*(RISC). CISC-type microprocessors are known for their abundant instruction set, multiple addressing modes, and multiple instruction formats and sizes. These processors, raises the level of the instruction set close to the system software. The control unit of such microprocessors are naturally complex, since they have to distinguish between a large number of opcodes, addressing modes and formats.

As opposed to the traditional CISC design, the RISC [Pat85] architecture reduces the instruction set to the level of vertical microcode. RISC processors feature

very few addressing modes and fixed instruction widths and format. All RISC processors support memory access only through load and store instructions. All other instructions are register-to-register.

The load/store architecture of RISC has certain overheads. The movement of data into and out of the registers is done explicitly via software instructions. There is another kind of load/store overhead, when calling a subroutine, or switching tasks, it is often necessary to explicitly save and restore all the programmer-visible registers.

1.2 A RISC without Load/Store Overhead

The load/store overhead can be reduced by removing registers and allowing all instructions to directly operate on memory locations. The current growth in technology helps in building caches which are cheaper and comparable in speed to registers. On many RISC machines, 25% or more instructions are loads and stores. The Register-Less RISC can save these load/store instructions.

[Bal97] proposed a Register-Less RISC (PERL RISC) architecture. SuperSIM a simulator which executes assembly language programs of PERL RISC is also developed. The performance of PERL RISC depends on two aspects: saving in number of load/store instructions, and cache performance.

In this thesis we look into cache performance aspects of PERL RISC. The cache performance of PERL RISC is compared with two current RISC processors: DLX [PH94] and Alpha [Sit93].

1.3 Cache Evaluation Methodology

The method we use for evaluating cache is *trace-driven simulation*. Trace-driven simulation is an effective method for evaluating the behavior of memory hierarchy. It uses one or more address *traces* and a cache *simulator*. A trace is a log of a dynamic series of memory references, recorded during execution of a program. A trace is usually gathered by interpretively executing a program and recording every main memory location referenced by the program during its execution. The

information recorded for each reference must include the address of the reference and may include the reference's type (instruction fetch, data read or data write) and other information. A *simulator* is a program that accepts a trace and parameters that describes one or more caches, mimics the behavior of those caches in response to the trace, and computes performance metrics (e.g. miss ratio) for each cache.

A cache simulator and a C compiler for PERL RISC are developed as part of this thesis. The compiler produces PERL RISC assembly code which can be run on the SuperSIM. Trace file produced by the simulator as a result of execution is used as input to the cache simulator.

1.4 Organization of the thesis

The rest of the thesis is organized as follows:

Chapter 2 presents the Register-less RISC architecture. It describes the instruction set and pipeline design of the processor.

Chapter 3 describes the implementation details of the compiler.

In chapter 4, we present the design aspects of cache memories and the techniques for improving cache performance.

We discuss, in chapter 5, the implementation details of the cache simulator and present the performance results in chapter 6. The results also are compared with some current processors.

Finally in chapter 7, we conclude with the implications of the results and possible extensions to this work.

Chapter 2

Register-Less RISC

2.1 Architecture

PERL RISC is a superscalar [SS95] memory-to-memory architecture. It is designed to be a memory-to-memory architecture because of its relevance today. The availability of large on-chip caches and wider bus bandwidth to reduce memory latency supports this idea. PERL RISC instructions do all the operations on memory locations, as there are no registers. It satisfies all common RISC traits [Tab94] except that it features memory-to-memory operations unlike the usual RISC processors. All instructions have a fixed length of 128 bits, and the number of instructions formats are only two. In PERL RISC, the first 32 bits of an instruction is used to specify the addressing modes and data type of the operands. The next three 32-bit words are used to indicate the addresses of each of the three operands.

2.1.1 Data Types

PERL RISC architecture recognizes the following data types.

- 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 float, 4 bytes.
 2. Double precision float, 8 bytes.

2.1.2 Addressing Modes

PERL RISC architecture supports four addressing modes, as practised on most RISC processors.

1. Immediate. In this case the value of the operand is part of the instruction.
2. Direct Addressing. In this case, the instruction itself has the 32-bit address of the operand. This is taken as effective address of the operand.
3. Memory Indirect. An instruction has the 32-bit address of the memory, where the effective address an operand is stored.
4. Base Addressing. The effective address is computed by adding 32-bit offset to the contents of base memory location. In PERL RISC, memory locations 0 to 3 are used to represent up to four base addresses. In the instruction format two bits are used to specify one of the four bases. This mode is primarily used for accessing local variables of a function on the stack. The compiler developed through this thesis currently uses only the first two locations, one for the *stack pointer* (SP) and another for the *frame pointer* (FP).

2.1.3 Instruction Set

The PERL RISC architecture features the following types of instructions:

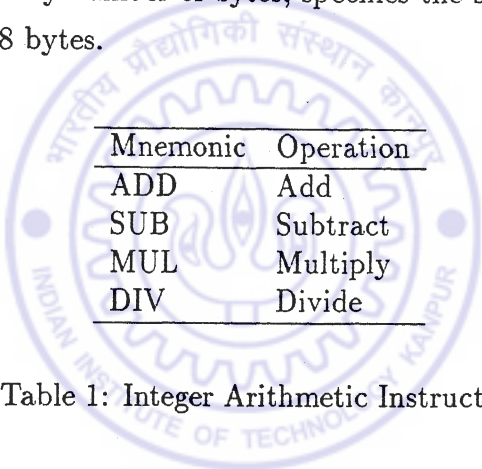
1. Integer
2. Logical and shift

3. Floating-point

4. Miscellaneous

Integer arithmetic instructions

The PERL RISC has four arithmetic instructions corresponding to four basic arithmetic operations. The instructions are listed in Table 1. A suffix to instruction mnemonic, *B* followed by number of bytes, specifies the size of operand. The valid sizes are 1, 2, 4, and 8 bytes.



Mnemonic	Operation
ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide

Table 1: Integer Arithmetic Instructions

Flow control Instructions

Flow control instructions include conditional and unconditional branch instructions. The control instructions are summarized in Table 2, *cond* in conditional jump can be either equality or inequality condition.

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

Table 2: Integer Control Instructions

Logical and shift instructions

The PERL RISC has three logical and three shift instructions. These instructions are listed in table 3.

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

Table 3: Logical and Shift Instructions

Floating-point instructions

All the integer arithmetic and control instructions 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.

Miscellaneous

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

2.1.4 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.

Register-less RISC has a five stage pipeline (Figure 1). The five pipeline stages are:

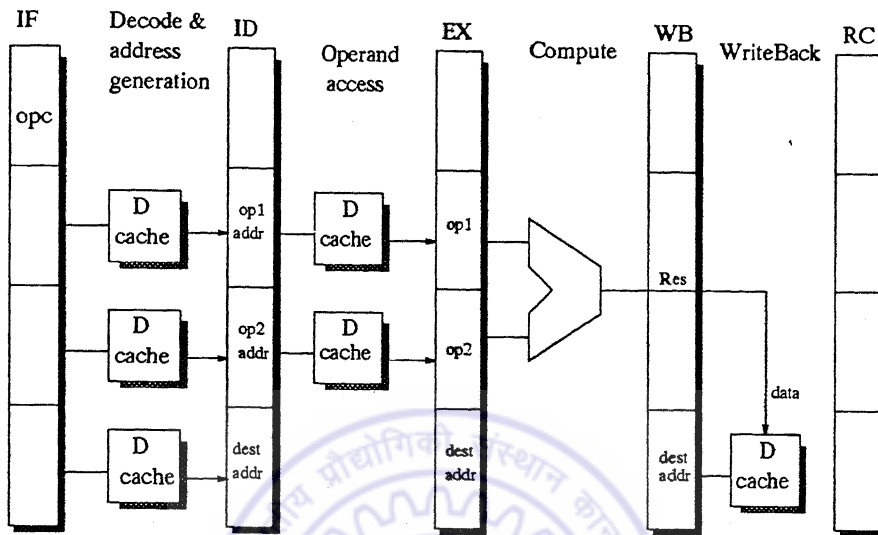


Figure 1: Processor Pipeline

Fetch Stage This stage takes instructions from the instruction cache and places them in an instruction queue.

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 more than one instruction per clock cycle.

Execute Stage The issue logic examines the instruction window and selects the ready ones: those for which operands and a functional unit are available. when two instructions conflict for the same functional unit, selection is made based on the age of the instruction: the oldest one has the highest priority.

Write Back Stage The write back logic identifies the completed operations and frees the corresponding functional units. The completed results are forwarded to the instructions that needs them.

Result Commit Stage The result commit stage sends the results to the memory locations.

Figure 1 also shows some additional stages where the cache is accessed. If the access to the cache are included, we have a seven stage pipeline. In fact, before the

fetch stage, there is an access to the instruction cache which is not shown in the figure.



Chapter 3

Implementation of Compiler

The compiler takes *C* language program as input and produces assembly code of PERL RISC as output and is built upon *GNU C* [Sta92] compiler. *GNU C* compiler (GCC) is a fast and highly portable compiler available in source form. GCC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. In this chapter, we describe the working of GCC, and its portability process including its machine description format and corresponding PERL RISC description.

3.1 Working of GNU C Compiler

The entire process of converting a *C* language file to assembly is done in several passes. Each pass belongs to one of the four phases of the Compiler: *Syntax Analysis*, *Intermediate Code Generation*, *Code Optimization* and *Code Generation* [AD85].

The *Syntax Analysis* phase is completed in a single parsing pass. This pass reads the entire text of a function definition, constructing syntax trees. *C* data type analysis is also done in this pass.

Intermediate code generation is also done in a single pass. This is the conversion of syntax tree into an intermediate code. GCC uses Register Transfer Language (RTL) [Sta92] as intermediate code which is described in section 3.2. The compiler's strategy is to generate RTL code assuming an unlimited number of *pseudo registers*,

and later convert them into hard registers or memory references. This and the later passes use the target machine description. All later passes work on the RTL code generated here.

Code optimization phase requires several passes. In a non-optimized compilation the compiler do jump optimization, common subexpression elimination and loop optimization. It uses simple register allocation strategy. Further in a optimized compilation *data flow analysis, instruction combination, local and global register allocation* etc. are done.

The *reload* pass of the *code generation* phase rennumbers pseudo registers with hard register numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Finally the last pass outputs assembler code for the function. Entry and exit assembly code sequences for a function are generated directly in this pass.

3.2 RTL Representation

RTL for the GCC is a LISP [SAN95] like language. In this language, Instructions to be output are described, one by one in an algebraic form that describes the instruction behavior. RTL uses four kinds of objects: *expressions, integers, strings* and *vectors*. Expressions are the important ones. RTL vector contains an arbitrary, specified number of pointers to expressions.

3.2.1 RTL Expressions

Expressions are building blocks for RTL instructions. Expressions are classified by expression codes. Expression code determines how many operands the expression contains. RTL expressions can be classified into five categories: *constant, register and memory, arithmetic, comparison and side effect expression*. Most of the expressions also have a machine mode, which describes size of a data object and the representation used for it.

Machine modes are classified into two categories:

1. Integer Modes: integer (SI), half integer (HI), quarter integer (QI), and double integer (DI).
2. Floating-point modes: Single precision(SF) and Double precision(DF).

Constant expressions

The simplest RTL expressions are those which represent constant values.

(**const_int i**) This expression represents the integer value *i*.

(**symbol_ref symbol**) Represents the value of an assembler label for data.
symbol is a string that describes the name of the assembler label.

(**label_ref label**) Represents the value of an assembler label for code.
label is a code_label that appears in the instruction sequence.

Register and memory expressions

These are the RTL expressions for describing access to machine registers and to main memory.

(**reg:m n**) For small values of the integer *n*, this stands for a reference to machine register number *n*: a hard register. For larger values of *n*, it stands for a temporary value or pseudo register. *m* is the machine mode of the reference.

(**cc0**) This refers to machine's condition code register. It is valid in only two contexts, as destination of an assignment and in comparison operators comparing against zero.

(**pc**) This represents the machine's program counter. *pc* is valid only in certain specific contexts in jump instructions.

(**mem:m addr**) This represents a reference to main memory at an address represented by the RTL expression *addr*. *m* specifies how large a unit of memory is accessed.

Arithmetic expressions

Syntax of an arithmetic expression is $(operation:m x)$ or $(operation:m x y)$. This represents the value obtained when *operation* is carried out on operands *x* and *y* in machine mode *m*. *operation* is either binary or unary arithmetic operation, e.g. *plus*, *minus*, and *not*. The number of operands depend on the operation type. *x* and *y* are RTL expressions of same machine mode.

Comparison expressions

Comparison operators test a relation on two operands and are considered to represent the value 1 if the relation holds, or zero if it does not. Inequality comparisons come in two flavors, signed and unsigned. Signed comparisons are also used for floating point values.

Syntax of comparison expression is $(relop x y)$, where *relop* is a relational operator name, such as *eq*, *gt* and *gtu*. Operands *x* and *y* should be in the same machine mode.

Side effect expressions

Expression codes described so far only represent values and not actions. But machine instructions never produce values; they are meaningful only for their side effects on the state of the machine. Special expression codes are used to represent side effects.

Body of an instruction is always one of these side effect codes; the codes described above, which represent values, appear as the operands of these.

(set lval x) Represents the action of storing value of *x* into the place represented by *lval*. *lval* must be an expression representing a place that can be stored in: *reg*, *mem*, *pc* or *cc0*.

(call function nargs) Represents a function call. *function* is a mem expression representing the address of the function to be called. *nargs* is an expression which can be used for two purposes: on some machines it represents the number of bytes of stack argument; on others, it represents the number of registers.

(return) Represents a return from the current function.

(if_then_else cond then else) This expression represents a choice, according to *cond*, between the value represented by *then* and the one represented by *else*. This expression is valid only to express conditional jumps.

3.3 Machine Description

A machine description has two parts: a file of instruction patterns ('.md' file) and a C header file of macro definitions.

The '.md' file for a target machine contains a pattern for each instruction that the target machine supports. Information about the target machine architecture such as registers, addressing modes, stack organization etc. is supplied in a C header file.

3.3.1 Instruction Patterns

Each instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a `define_insn` expression.

A `define_insn` is an RTL expression containing four or five operands.

1. An optional name. The presence of a name indicates that this instruction pattern can perform a certain standard job for the RTL generation pass of the compiler. Nameless instruction patterns are used to combine several simpler instructions. The names that are meaningful are *addm3*, *subm3* etc. *m* stands for machine mode.
2. The RTL template is a vector of incomplete RTL expressions which show that what the instruction should look like. It is incomplete because it may contain *match_operand* and *match_dup* expressions that stand for operands of instruction.

- Syntax of *match_operand* is (*match_operand:m n pred constraints*). This expression is a place holder for operand *n* of the instruction. When constructing an instruction, operand number *n* will be substituted at this point. *pred* is a string that is the name of a C function that accepts two arguments, an expression and a machine mode. During matching, the function will be called with operand as the expression and *m* as the mode argument. If it returns zero, this instruction pattern fails to match.
 - (*match_dup n*). This expression is a placeholder for operand number *n*. It is used when the operand needs to appear more than once in the instruction.
 - (*match_operator:m n predicate [operands..]*). This pattern is a kind of placeholder for a variable RTL expression code. when constructing an instruction, it stands for an RTL expression whose expression code taken from that of operand *n*, and whose operands are constructed from the patterns *operands*.
3. A condition. This is a string which contains a C expression that is the final test to decide whether an instruction body matches this pattern.
 4. The output template: a string that says how to output matching instructions assembler code. '%' followed by number of operand in this string specifies where to substitute the value of operand. When sample substitution is not general enough, a piece of C code can be specified to compute the output.
 5. Optionally, some machine description.

Example of `define_insn`

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "r")
        (plus:SI
          (match_operand:SI 1 "register_operand" "r")
          (match_operand:SI 2 "register_operand" "r")))]
  ""
```



```
"add %0,%1,%2")
```

This is an integer *add* instruction in a RISC machine. All operands must be registers. *r* in the operand constraints specifies that any general register is valid.

3.3.2 Defining RTL Sequences for Code Generation

On some target machines, some standard pattern names for RTL generation cannot be handled with single instruction, but a sequence of RTL instructions can represent them. For these target machines, a `define_expand` expression can be used to specify how to generate the sequence of RTL.

A `define_expand` is an RTL expression that looks almost like a `define_insn`; but, unlike the latter, a `define_expand` is used for RTL generation and it can produce more than one RTL instructions.

A `define_expand` expression has four operands:

- The name. Each `define_expand` must have a name, since the only use for it is to refer to it by name.
- The RTL template. This is a vector of RTL expressions each being one instructions.
- The condition, a string containing a C expression. This is just like the condition of a `define_insn`.
- The preparation statements, a string containing zero or more C statements which are to be executed from the RTL template.

There are two special macros defined for use in the preparation statements: `DONE` and `FAIL`.

`DONE` The RTL template will not be generated.

`FAIL` Make the pattern fail. When a pattern fails, it means that the pattern was not truly available. Calling routines in the compiler will try other strategies for code generation using other patterns.

3.4 Machine Description for PERL RISC

3.4.1 Architecture Specification

Storage layout

The processor is defined as *big endian*, most significant byte in a word has the lowest number. In a multi-word the most significant word has the lowest number. The least addressable storage unit is byte, which has eight bits. Word size and the addresses are 32 bits. Function entry points and instruction addresses such as branch target are aligned on sixteen byte blocks (instruction boundaries), to fetch an instruction in a single read request. All other objects are aligned on word boundaries.

Temporary locations usage

The number of registers of the processor, and their usage is supplied to GCC through C Macro definitions. Compiler needs at least two registers to be specified, stack pointer (SP) and frame pointer (FP). In order to access indirect operands, we use temporary locations in PERL RISC. As GCC has no concept of temporary locations, these are specified as registers. Assembly code declares temporaries for global references. Size of each temporary location is four bytes. Currently we are using sixteen such temporary variables.

Stack layout

The PERL RISC has no hardware stack. The stack has to be implemented in software itself. The space for local variables of a function is allocated on the stack. Each function has a frame allocated for it on the stack.

The first 12 bytes on the frame has fixed usage.

- The first 4 byte location is used to store the old frame pointer at the function entry point.
- The next 4 byte location is used to store the return address from the function.

- The last 4 bytes are used pass the function results to the caller function. The address where the return value is present is passed in these 4 bytes.

In PERL RISC there is no function call instruction. Stack adjustment is done by the explicitly generated assembly code, both at the function entry and exit points.

3.4.2 Instruction Patterns

All the available assembly instructions in PERL RISC are specified to the compiler using instruction pattern.

For the purpose of specifying instruction patterns, the assembly instructions can be broadly classified into two types: arithmetic and flow control instructions.

Arithmetic instructions

For each available assembly instruction, a named `define_insn` pattern is specified.

For example the `addb4` instruction is specified by using the following instruction pattern:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_or_addr_operand" "")
        (plus:SI (match_operand:SI 1 "general_or_addr_operand" "")
                 (match_operand:SI 2 "general_or_addr_operand" "")))]
  ""
  "*"
  {
    return \"addb4 %0,%1,%2\";
  })
```

The function `general_or_addr_operand` checks the operand addressing mode. The functions matches the instruction pattern if the addressing mode is a valid addressing mode.

Control instructions

GCC assumes that the machine has a condition code. A comparison instruction sets the condition code, recording the results of both signed and unsigned comparison of the given operands. A separate branch instruction tests the condition code and branches or not according its value.

PERL RISC has compare-and-branch instructions and has no condition code. As there is no assembly instruction in PERL RISC corresponding to a comparison instruction generated by the compiler, a `define_expand` expression is specified to record the operands in two static variables.

For example a `define_expand` expression for integer comparison instruction is specified as follows:

```
(define_expand "cmpsi"
  [(set (cc0) (compare
            (match_operand:SI 0 "general_or_addr_operand" "")
            (match_operand:SI 1 "general_or_addr_operand" "")))]
  ""
  ""
  {
    compare_op0 = operands[0];
    compare_op1 = operands[1];
    DONE;
  })
```

The `DONE` macro in C preparation statements specifies that no RTL code will be generated for this instruction. When outputting the branch-on-condition-code instruction that follows, the compiler actually outputs a compare-and-branch instruction that uses the remembered operands.

For example a branch-on-equal instruction is specified as follows:

```
(define_insn "bge"
  [(set (pc)
```

```

        (if_then_else (eq (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
""
"*
{
  operands[1] = compare_op0;
  operands[2] = compare_op1;
  return \"jeqb4 %10,%1,%2\";
}

```

For call instructions an unconditional jump instruction, j function, $-8(sp)$, is generated. The jump instruction stores the return address on the stack, which is later used to return from the function.

Chapter 4

Cache Memories

4.1 Introduction

Cache is a high speed memory, which bridges speed gap between Microprocessor and slow memories. In most of the new processors small caches are placed right on the chip itself, enabling a faster access. Some newer chips actually dedicate as many transistors (and therefore the real chip area) for the cache as they do for the processor itself.

The data stored in cache do not have a separate address. They are referred to by their addresses in the main memory and data in the cache is found using an address mapping mechanism.

In most existing systems cache is subdivided into sets. Each set contains a number of lines. Main memory address is viewed as three parts as shown in figure 2. Using different fields, cache byte may be accessed by address mapping, as practiced on most systems, called set-associative mapping. If a set contains L lines it is called L -way set-associative. If each set contains only one line, that is called *direct mapped* cache.

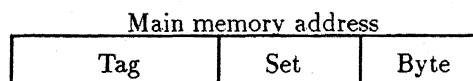


Figure 2: Main Memory Address Splitting

When a CPU attempts to access any item of information, there are two possible outcomes. The item is in the cache, a *hit*, or not in the cache, a *miss*. When a miss occurs, the line containing the missing item is loaded into cache, replacing another line.

4.2 Aspects Of Cache Design

Cache fetch algorithm

The cache fetch algorithm is used to decide when to bring information into the cache. Several possibilities exist: information can be fetched on demand (when it is needed) or prefetched (before it is needed).

Line size

Selecting the line size is an important part of the memory system design. While the transmission time for moving a small line to cache is shorter than that for a long line, and it reduces the wait for processor; using bigger lines are more efficient if more information in a line is being used.

Replacement algorithm

When information is requested by the CPU from main memory and the cache is full, some information in the cache must be selected for replacement. The most frequently used strategy for selecting which block to replace is *Least recently used* (LRU). The block replaced is the one that has been unused for the longest time.

Main memory update algorithm

When the CPU performs a write to memory, that operation can actually be reflected in the cache and main memory in a number of ways. For example, the cache memory can receive the write and the main memory can be updated when that line is replaced

in the cache. This strategy is called as *write-back*. Another strategy, known as *write-through*, immediately updates main memory when a write occurs.

Data/Instruction cache

Another cache design strategy is to split the cache into two parts: one for data and one for instructions. This has the advantage that the bandwidth of the cache is increased and the access time can be decreased.

Cache size

Large caches increase the probability of finding the needed information in it. But cache sizes cannot be expanded, for several reasons: cost, physical size and access time. Small caches can be put on-chip, and access times can be reduced.

Multilevel cache

Large caches can be split into two levels: a small, high-level cache, which is faster, smaller, and more expensive per byte, and a larger, second-level cache.

Cache bandwidth

The cache bandwidth must be sufficient to support the proposed rate of instruction execution. Bandwidth can be improved by increasing the width of the data path, interleaving the cache, operating in nonblocking mode and decreasing access time.

4.3 Increasing cache bandwidth

The memory bandwidth requirement for PERL RISC is very high. Each instruction may require up to a maximum of six memory requests: three for resolving operand addresses, two for operand reads, and one for write. This high bandwidth requirement can be satisfied either by increasing the efficiency of cache port or by using a multi-ported cache.

4.3.1 Increasing Cache Port Efficiency

The cache port bandwidth can be increased by using *non-blocking* cache. Non-blocking cache allows the service of multiple misses to be overlapped, in a pipelined fashion, with a packet-switched bus in which the primary level cache to secondary-level cache (L1-L2) bus is not held for the duration of the memory request.

Kenneth [Ken91] proposed techniques for improving cache port efficiency. In those techniques *load all*(LA) and *load all wide* (LAW) will be most suitable ones for PERL RISC.

Load all increases the cache bandwidth by satisfying as many outstanding loads in parallel as possible when data is returned from the cache. **Load all wide** builds upon **load all** by widening the single cache port up to the cache block size to increase cache bandwidth. All the outstanding loads reside in cache access buffer. To make use of an entire cache line, each cache access buffer entry must contain a multiplexer as well as a comparator. If the comparator detects that tags are equal, then the multiplexer is used to select the correct data block from the returning cache line.

4.3.2 Multi-ported cache

The two techniques for implementing multiple cache ports are: (i) to duplicate the cache and (ii) to interleave the cache [SF91].

Duplicate cache banks

A straightforward way to implement multiple read ports is to provide multiple copies of the cache. For example, 4 read ports can be provided to a 16 K-byte cache by having four 16 K-byte caches that have identical contents. This approach has a significant overhead in the amount of memory used, especially when considering an on-chip cache. Identical multiple copies use only a single write port.

Interleaved banks

A better way to provide multiple cache ports is to interleave the cache blocks amongst multiple banks, much in the same way as in an interleaved memory. A

cache block is present entirely in one single cache bank. Figure 3 shows how an interleaved L1 (primary) cache could be placed in the CPU.

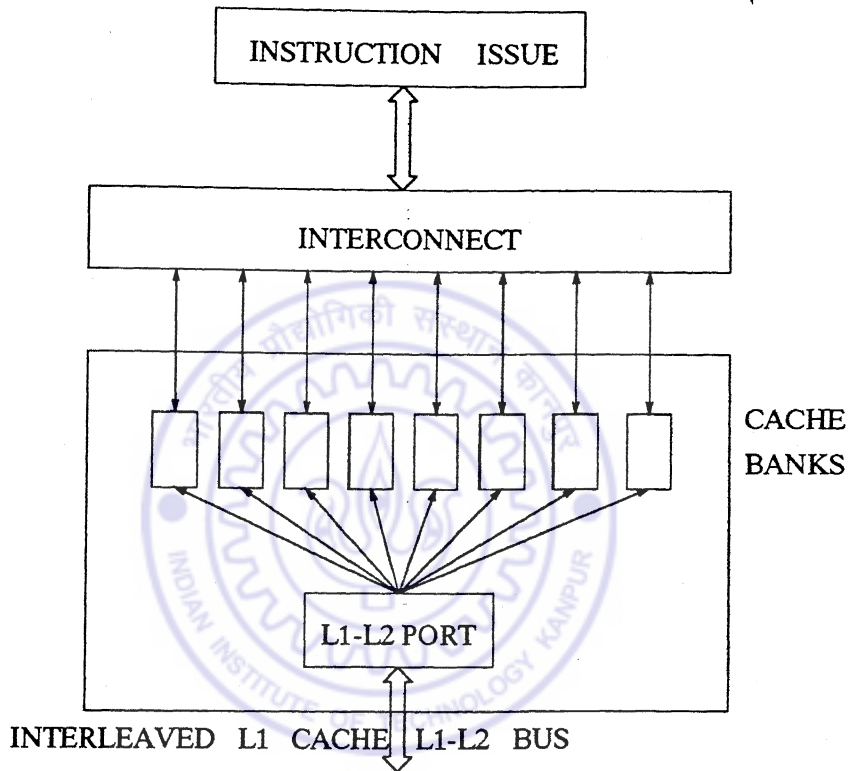


Figure 3: A Multi-Port Cache with Interleaved Banks

4.4 Case Studies

In this section we look at the cache organization in the two latest RISC processors, DEC Alpha 21164 [ERB⁺95] and UltraSPARC [TGN95].

4.4.1 Alpha 21164

Alpha 21164 processor has two levels of on-chip cache. In first-level there are separate Instruction and data caches, each having a size of 8-Kbyte and direct-mapped. The second-level cache is 96 K-byte and organized as three-way set associative.

Direct-mapped, off-chip, third-level cache can range from 1 M-byte to 64 M-bytes in size. Both instruction and data caches contain 32-byte blocks. Second level cache contains one tag per 64-byte block. Primary (L1) data cache is write-through, write-allocate, while data cache at L2 is write-back.

Two read ports are provided in data cache, supporting dual-issue and parallel execution of load instructions. The latency of a load instruction that hit in the data cache is two cycles, while that for second level cache is seven cycles. At all levels memory system operates in nonblocking mode.

4.4.2 UltraSPARC

UltraSPARC memory hierarchy consists of an instruction cache (I-cache), a data cache (D-cache) and an external cache (E-cache).

I-cache size is 16kB, organized in 32 byte blocks. Stored with each instruction in the I-cache are predecoded bits that are used for instruction fetching. The I-cache has qualities of both a direct-mapped and a two-way set-associative cache. It can be considered as two-way in that a particular address can be present in two different locations in the cache. It is considered direct-mapped in terms of access time in that the 'set' to access is predicted ahead of time, so there is no comparison function in the access path.

D-cache is a 16 kB direct-mapped cache. It has a 32 byte line size, with 16 byte sub-blocks. It is indexed using virtual address, while tags store the physical address. D-cache operates on a write-through, no write-allocate policy and is nonblocking so that D-cache misses and other conditions which delay memory operations do not necessarily penalize subsequent instructions. D-cache load latency is one cycle.

The E-cache lies between the primary caches and main memory. It is direct-mapped and both physically indexed and physically tagged. UltraSPARC supports E-cache sizes between 512 Kb and 4 Mb. E-cache operates on a write-allocate policy and is pipelined so that one operation can be computed every clock cycle.

Chapter 5

Implementation of Cache Simulator

The simulator is a modified version of an existing cache simulator [New]. The existing simulator is modified to simulate 'Load All Wide' technique and multi-port caches. It can be used either as a stand alone trace-driven simulator or in conjunction with a processor simulator as a memory-hierarchy simulator to simulate entire memory-hierarchy. In this chapter, we describe details of the simulator.

5.1 Simulator Input

The input to the simulator is a memory trace file and a cache configuration file. The format of these files is given in appendix A.

The trace file is generated by the processor instruction set simulator while executing a program. For each reference a trace contains the following information in addition to the address of memory reference.

- **Type of reference.** Specifies whether the reference is fetch, read or write.
- **Clock cycle.** Specifies the processor clock cycle with reference to the start of execution in which the reference is made.

The configuration file supplies different parameters of cache hierarchy. It specifies number of levels of cache hierarchy and the following for each level:

- Type of cache: Unified or split cache. In case of split cache the following cache parameters are specified separately for both Instruction and data cache. In case of Unified cache, these parameters are specified only once.
- Number of cache lines.
- Line size.
- Associativity.
- Number of interleaved cache ports.
- Number of duplicate cache ports.
- Write policy, only in case of Unified or data cache. This can be either write back or write through.
- Number of clock cycles required to satisfy the request, in case of a cache miss but a hit at the next level.

All the parameters of cache can be configured except cache block replacement policy, which is fixed as *least recently used*.

5.2 Simulator Output

The simulator gives the following performance metrics for each level of cache.

- Miss-ratio for each category of misses.
- Total number of write backs required in case of a write back cache.

In addition the simulator also gives the number of references that are serviced by *load all wide* optimization and the worst case clock cycles required for execution of the program. Worst case clock cycles are calculated by adding penalties due to cache bank clashes and misses. Simulator services references in a clock cycle only after servicing all references of the previous clock cycle.

Chapter 6

Cache Simulation Results

This chapter compares the cache performance of PERL RISC for some C benchmark programs with other RISC processors. The results are compared with that of DLX [PH94] and DEC ALPHA 21064 [Sit93]. DLX is a hypothetical RISC architecture which can be taken as a representative of all current day RISC processors. We obtain results for DLX using trace-driven simulation. **SuperDLX** [Mou93], a simulator for DLX, is used to obtain the memory traces. A C compiler for DLX is also provided with **SuperSIM** simulator. The cache performance metrics for DEC Alpha 21064 are obtained using the ATOM [Dig93, Dig94] performance analysis tool which is part of the DEC OSF software kit. Results provided by ATOM are only simulated values, and not the measured values. ATOM is built using OM, a link-time code modification system. OM takes as input a collection of object files and libraries that make up a complete program, builds a symbolic intermediate representation, patches code for the instrumentation and optimizations to the intermediate representation, and finally outputs an executable. Instrumentation routines can be customized for one's needs. We used the routines which are provided with ATOM distribution. ATOM distribution contains tools to get cache performance, instruction profiling etc. As we used ATOM on DEC Alpha 21064, which is 2-issue superscalar, we configured **SuperSIM** and **SuperDLX** to simulate in 2-issue mode to have comparable platforms.

6.1 Benchmark Programs

The programs used for simulation are:

1. **Permute.** This is a heavily recursive program which, given an array of n integers, prints all $n!$ permutations. The results are obtained for n equal to 5.
2. **Matmul.** An integer matrix multiplication program. The results are obtained for matrices of size 32 rows and 32 columns.
3. **Tts.** This is 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, this program uses a heuristic approach to get the best optimized output. The size of this program C source code is 900 lines.

6.2 Memory Bandwidth Requirement

The memory bandwidth requirement for the three processors is compared in this section. Table 4 presents the instruction and data reference counts for the benchmark programs separately.

<i>Benchmarks</i>	<i>Insn. Ref.</i>			<i>Data Ref.</i>		
	ALPHA	DLX	PERL RISC	ALPHA	DLX	PERL RISC
Permute	10014	14316	8947	4817	8579	10557
Matmul	523982	696021	357421	133331	101414	689821
Tts	3034509	2666910	1123283	666824	946293	1138070

Table 4: Comparison of Memory Reference Counts

The PERL RISC data reference counts presented in the table does not contain sp and fp references. About 30% of total references are sp and fp references. These references are removed to reduce cache bandwidth requirement. These two locations can be allocated to special registers in the processor itself.

Memory reference counts show that PERL RISC on the average executes 40% less instructions than the other two RISC processors. But the size of each instruction of PERL RISC is 16 bytes, where as the other processors instruction size is only 4 bytes. So the PERL RISC program size will be about double the RISC processor's program size. Data references of PERL RISC are also more than the RISC processor's references.

Average memory bandwidth requirement of the three processors is presented in table 5, which is expressed as number of references per clock cycle.

<i>Benchmarks</i>	<i>Insn. Refs./cycle</i>			<i>Data Refs./cycle</i>		
	ALPHA	DLX	PERL RISC	ALPHA	DLX	PERL RISC
Permute	0.88	1.75	0.78	0.42	1.05	0.92
Matmul	0.42	1.99	1.08	0.11	0.29	2.08
Tts	1.06	1.47	1.08	0.33	0.83	1.10

Table 5: Comparison of Memory Bandwidth Requirement

Number of instruction references per clock cycle in PERL RISC is around 1. An instruction cache can definitely provide this bandwidth. It is significant that the data reference bandwidth is also around 1 for two benchmark programs. It is observed that though the average references are as given for benchmark programs, actual number of references may vary from 0 to 5. Table 6 gives the memory reference distribution for Tts.

<i>Processor</i>	<i>Percentage of cycles with Refs.</i>				
	0	1	2	3	> 4
DLX	52	44	4	< 1	0
PERL RISC	24	45	28	2	< 1

Table 6: Comparison of Memory Reference Distribution

The statistics shows that 30% of the clock cycles in PERL RISC requires two references, where as for DLX most of the clock cycles requires either zero or one

reference only. In both processors, the percentage of clock cycles with more than two references is very small.

6.3 Effectiveness of Bandwidth Improvement Techniques

This section presents the cache performance improvement due to *load all wide* technique and multi-port caches, which are discussed in section 4.3. The performance of multi-port caches depend only on memory reference pattern. The number of references satisfied by 'load all wide' depends on both cache block size and memory reference pattern.

6.3.1 Load All Wide

The percentage of references satisfied by this technique for the three benchmark programs are given in table 7. These results are obtained for cache block size of 32 bytes for both DLX and PERL RISC.

Benchmark	PERL RISC	DLX
Permute	12	14
Matmul	15	< 0.01
Tts	1	< 0.4

Table 7: Percentage of References Satisfied by Load All Wide

In case of **Permute** and **Matmul** programs, this technique satisfied more than 10% references. Though it satisfies only 1% references for **Tts** program, it is significant that each reference satisfied reduces a cache port clash.

6.3.2 Multi-port Cache

This section presents the cache performance improvements due to multiple cache ports. The metric we use to evaluate the performance is additional clock cycles

needed to service the memory requests assuming that all requests in a clock cycle must be satisfied before servicing any request from the next cycle. It is observed that for the three benchmark programs DLX requires less than 1% clock cycles even for a single port cache.

In case of PERL RISC, this value varies from 20% to 5% for **Matmul** as the number of ports increased to 2 duplicate ports. It requires no additional clock cycles if the cache has 2 interleaved and 2 duplicate ports. For **Tts** program PERL RISC require 3% more clock cycles for a single port cache, and a dual port cache reduces the requirement to less than 1%.

6.4 Primary-level Cache Performance

The primary-level cache can be either a unified cache or split cache. We used split cache organization for performance study. Performance results for instruction cache and data cache are presented separately in this section.

6.4.1 Instruction Cache

The cache sizes for simulation are selected based on the size of the program. For **Matmul** and **Permute** programs both processors reported very high (more than 95%) hit rate even for a 1 KByte cache. This can be attributed to the fact that, being small programs the entire program can fit into 1 Kbyte cache and all the misses are compulsory misses. But it is observed that PERL RISC has more number of misses compared to DLX. The misses in PERL RISC are decreased when we increased the cache block size keeping the cache size same. This is due to decrease in compulsory misses, as more instruction fit into same cache block.

Table 8 shows the miss ratios for **Tts** program for DLX and PERL RISC. The miss ratio for ALPHA for the same program is 0.36%. ALPHA has an 8 K-byte instruction cache. The miss ratios shows that PERL RISC requires a large and wider block size cache to have lower miss ratio. Data transfer rate between second-level cache and instruction cache is also more in case of PERL RISC.

<i>Processor</i>	<i>Cache Size</i>		
	2K	4K	8K
DLX(32 byte block)	1.33%	0.35%	0.11%
PERL RISC (32 byte block)	5.45%	2.69%	1.25%
PERL RISC (64 byte block)	3.69%	1.60%	0.68%
Alpha (32 byte block)	-	-	0.36%

Table 8: Instruction Cache Miss Ratios

6.4.2 Data Cache Performance

We use number of misses rather than miss ratios as metric for evaluating data cache performance, Because a program access the same set of data even when executing on different processors.

The number of misses for **Permute** program are given in table 9. Because the program accesses only a small array, even a cache of 1 K-byte has very few misses. The results shows that most of the misses in PERL RISC are conflict misses. The large cache also has the same number of misses.

<i>Cache Size</i>			DLX	PERL RISC
Blocks	Block Size	Associativity		
32	32	1	9	177
32	32	2	9	10
64	32	1	9	177

Table 9: Comparison of Data Cache Misses for Permute

Table 10 gives the misses for **Matmul**. Increased associativity has considerable impact on cache performance in PERL RISC for this program also.

For a 16 K cache all misses are compulsory misses because all the three matrices can fit into it, and both processors have the same number of misses. The number of misses in ALPHA for this program are 17262. ALPHA has a 8 K-byte data cache. The high number of misses can be attributed to fact that the misses are recorded while executing the program in an operating system environment.

<i>Cache Size</i>			DLX	PERL RISC
Blocks	Block Size	Associativity		
128	32	1	6332	10320
128	32	2	2860	2019
256	32	1	1351	4324
256	32	2	452	893
512	32	1	421	435
512	32	2	421	420

Table 10: Comparison of Data Cache Misses for Matmul

The number of misses for Tts program are given in table 11 for different cache configurations.

<i>Cache Size</i>			DLX	PERL RISC
Blocks	Block Size	Associativity		
64	32	1	11097	10061
64	32	2	5819	3345
128	32	1	4615	2625
128	32	2	2764	1316
256	32	1	2840	1455
256	32	2	869	519

Table 11: Comparison of Data Cache Misses for Timetable program

For this program PERL RISC has less number of misses compared to the DLX for all cache configurations considered. Cache associativity is an important factor for cache performance in this program also. For the same program, cache misses in ALPHA are 35883.

Chapter 7

Conclusions and Future Work

In this thesis, A C compiler for PERL RISC is developed and cache performance of PERL RISC is studied. The compiler is built on top of GCC and the generated code is not a 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 can be modified to produce fully optimized code for PERL RISC.

The cache performance studies shows that:

- Instruction cache misses for PERL RISC are more compared to other RISC processors. PERL RISC requires large and wider block size instruction cache to have fewer misses.
- A dual port data cache can satisfy the increased bandwidth requirement of PERL RISC.
- Data cache performance of PERL RISC is comparable to other RISC processors.

The cache performance results are obtained for only three benchmark programs. The results should be substantiated by simulating more programs.

Appendix A

Cache Simulator Input Files format

A.1 Trace File (*trace*)

#reference type	memory address(in hexadecimal)	clock-cycl
2 (Fetch)	100	1
1 (Write)	4	5
0 (Read)	4	7

A.2 Cache Configuration File (*cache.config*)

```
#Number of Cache levels
2
#Level 2 cache specification
#Type of cache      Number of blocks  Block size(bytes)  Associativity
0 (Unified cache)   1024                64                  4
#Number of Interleaved ports  Duplicate Ports  Write Policy
1                        1                0 (write back)
#miss penalty for this level
10
```

#Level 1 cache specification

#Type of cache

1 (Split cache)

#Instruction cache specification

#Number of Blocks Block size Associativity

256 64 1

#Number of Interleaved ports Duplicate Ports

1 1

#Data Cache specification

#Number of Blocks Block size Associativity

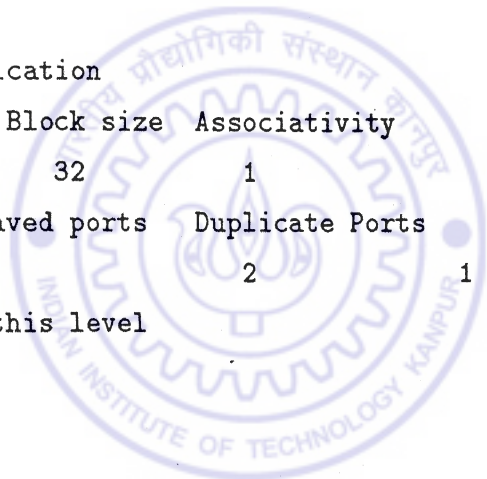
512 32 1

#Number of Interleaved ports Duplicate Ports Write Policy

2 2 1 (write through)

#miss penalty for this level

4



Appendix B

A Sample Output of Compiler

B.1 A C Benchmark Program

/* This is a simple benchmark program,DAXPY.

It does the following operation:

$$Y = AX + Y$$

where X and Y are vectors (of double precision numbers) of size 'n'. Each element of vector X, X[i], is multiplied by double precision number A and the result is added to Y[i].

*/

```
double x[10],y[10],a;
main()
{
    int i;
    a = 3.5 ;
    for(i=0;i<10;i++)
        y[i] = a*x[i]+y[i];
}
```

B.2 Assembly Code Generated by Compiler

The assembly code generated by the compiler for the C program is given in this section. The compiler generates traps for systems calls and library functions. The simulator handles them by using the corresponding UNIX system calls or library functions in the trap handler. Simulator automatically recognizes *sp* and *fp* as stack pointer and frame pointer which are used as base addresses. Traps that are not relevant to this example are removed. First the *sp* is initialized to *memSize*, a simulator variable which indicates the highest address of processor's memory.

```
.global _exit
.global _printf
.global t1
.global t2
.global t3
.global t4
.global t5
.global t6
.global t7
.global t8
.global t9
.global t10
.global t11
.global t12
.global t13
.global t14
```

```
.align 2
```

```
LC0:
```

```
.double 3.50000000000000000000000000000000
```

```
.align 4
```

```
.global _main
```

```
_main:
```

```

addb4 sp,#memSize, #0
;; Save the old frame pointer
addb4 -4(sp),fp,#0
;; Establish new frame pointer
addb4 fp,#0,sp
;; Adjust Stack Pointer
addb4 sp,sp,#-44
;; Save Temporary locations
addf8 _a,#0,LC0
addb4 t3,#0,#0
addb4 t5,#0,@_y
addb4 t4,#0,@_x

```

L5:

```

sllb4 t2,t3,#3
addb4 t1,t5,t2
addb4 t2,t4,t2
addb4 t6,#0,t2
mulf8 -20(fp),_a,(t6)
addb4 t6,#0,t1
addf8 (t6),-20(fp),(t6)
addb4 t3,t3,#1
jleb4 L5,t3,#9
;; Restore the saved Temporary locations
;; Restore stack pointer
addb4 sp,#0,fp
;; Restore frame pointer
addb4 fp,-4(fp),#0
;; HALT
j _exit,#0

```

_exit:

```

trap #0

```

```
        j -8(sp),#0
_printf:
        trap #5
        j -8(sp),#0
t1:     .space 4
t2:     .space 4
t3:     .space 4
t4:     .space 4
t5:     .space 4
t6:     .space 4
t7:     .space 4
t8:     .space 4
t9:     .space 4
t10:    .space 4
t11:    .space 4
t12:    .space 4
t13:    .space 4
t14:    .space 4
.global _a
_a:     .space 8
.global _y
_y:     .space 80
.global _x
_x:     .space 80
```



Bibliography

- [AD85] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1985.
- [Bal97] T. S. Balaji. Design and Simulation of PERL RISC. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, January 1997.
- [Dig93] Digital Equipment Corporation. *Atom Reference Manual*, Dec 1993.
- [Dig94] Digital Equipment Corporation. *Atom User Manual*, Mar 1994.
- [ERB⁺95] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, et al. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, 1995.
- [Ken91] Kenneth M. Wilson, Kunle Olukotun, and Mendel Rosenblum. Increasing Cache Port Efficiency for Dynamic SuperScalar Microprocessors. In *International Symposium on Computer Architecture*, volume 23, pages 147-157, 1991.
- [Mou93] Cecile Moura. SuperDLX-A Generic Superscalar Simulator. Master's thesis, Advanced Compilers, Architectures and Parallel Systems Group, McGill University, May 1993.
- [New] New Mexico State University. *ACS: Cache Simulator*.
- [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.
- [SAN95] RAJEEV SANGAL. *LISP Programming*. TATA MCGRAW-HILL, 1995.
- [SF91] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Architectural Support for Programming Languages and Operating Systems*, volume 19, pages 53-62, 1991.
- [Sit93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36:33-44, Feb 1993.
- [Smi82] Alan Jay Smith. Cache memories. In *ACM Computing Surveys*, volume 14.3, pages 473-530, 1982.
- [SS95] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. In *Proceedings of the IEEE*, volume 83, pages 1609-1624, 1995.
- [Sta92] Richard M. Stallman. *Using and Porting GNU C Compiler*. Free Software Foundation, 1992.
- [Tab94] Daniel Tabak. *Advanced Microprocessors*. McGraw Hill, second edition, 1994.
- [TGN95] Marc Tremblay, Dale Greenley, and Kevin Normoyle. The Design of the Microarchitecture of UltraSPARC-I. In *Proceedings of the IEEE*, volume 83, pages 1653-1671, 1995.

