RC 12434 (#55845) 1/9/87
Computer Science 62 pages

High Performance Reduced Instruction Set Processors

by

Tilak Agerwala

and

John Cocke

IBM Thomas J. Watson Research Center

Yorktown Heights, New York

March 31, 1987

RC 12434 (#55845) 1/9/87
Computer Science 62 pages

High Performance Reduced Instruction Set Processors

Tilak Agerwala and John Cocke

The focus of this paper is on high performance. In general purpose computing environments, a very large fraction of instructions executed are simple: load, store, add, shift, compare, logic, and branch. To obtain high speeds, sequences of simple instructions must be executed as fast as possible. This paper describes compiler, architecture, and machine organization approaches that result in efficient use of hardware and very high execution speeds. We show how pipeline disruptions can be (almost) eliminated by proper architecture design and appropriate compiler optimizations. The penalties due to cache misses can be significant, and approaches to reducing these penalties are presented. We describe a series of techniques and show the effect of each one on performance.

We do not recommend a pure RISC approach for all applications and environments. If complex operations (such as floating point arithmetic) are important, direct hardware implementation of the functions may be necessary to meet performance targets. Even in these cases, however, it is possible to extend the basic RISC approach and expose the hardware to the compiler to obtain better efficiencies. High speed execution of three important complex operations including movement of character strings in memory and floating point arithmetic will be presented in this paper.

Contents

High Performance Reduced Instruction Set Processors

by

Tilak Agerwala

John Cocke

1. Introduction

Reduced instruction sets contain carefully selected, simple instructions. The choice is dependent on hardware and compiler considerations. It should be possible to implement the instructions efficiently in hardware and complex functions should be compilable as compact sequences of simple instructions. This leads to an approach in which performance is obtained by transferring as much work as possible from run time to compile time. The basic characteristics, motivation, and evolution of reduced instruction set machines were very adequately covered in [1]. This paper also provided an overview of three important projects: 801 at IBM [2], MIPS at Stanford [3], and RISC at Berkeley, and touched upon the role of optimizing compilers and machine organization issues.

The focus of our paper is on high performance. In general purpose computing environments, a very large fraction of instructions executed are simple: load, store, add, shift, compare, logic, and branch. To obtain high speeds, sequences of simple instructions must be executed as fast as possible. This paper describes compiler, architecture, and machine organization approaches that result in efficient use of hardware and very high execution speeds. We show how pipeline disruptions can be (almost) eliminated by proper architecture design and appropriate compiler optimizations. The penalties due to cache misses can be significant, and approaches to reducing these penalties are presented. We describe a series of techniques and show the effect of each one on performance. The overall utility of a given approach is, of course, dependent on the application environment and design constraints.

For specialized functions such as fixed point multiply, floating point arithmetic, garbage collection, and type checking, it is possible to provide simple architecture support which can be used by the compiler to provide better performance. In all such cases the impact on the machine data paths and the expected performance gains must be carefully analysed. We do not recommend a pure RISC approach for all applications and environments. If complex operations (such as floating point arithmetic) are important, direct hardware implementation of the functions may be necessary to meet performance targets. Even in these cases, however, it is possible to extend the basic RISC approach and expose the hardware to the compiler to obtain better efficiencies. High speed execution of three important complex operations including movement of character strings in memory and floating point arithmetic will be presented in this paper.

We will focus on an architecture similar to the 32-bit 801 architecture described in [2]. The 801 has fixed length, 32-bit instructions, a 32-bit word (4 bytes), and 32 general purpose registers (GPR). Character, half word and word data types are supported. Load and store instructions use Base/Index or Base/Displacement for effective address generation. "Progressive indexing" can be used. In this case the effective address is the sum of the contents of the base and index registers; the base register is updated with the effective address. The usual complement of arithmetic, logical, and compare operations are provided as register to register (RR) instructions. A 3-address format is utilized in RR operations. In addition, a powerful set of shift and rotate operations with masking are defined. A 4-bit condition register is provided together with "branch on bit" instructions. The architecture uses a rich set of "immediate" fields.

2. A Simple Machine

To achieve a sustained rate of one cycle per instruction (C/I) on a long sequence of simple instructions, we start with the simple data flow shown in Figure 1. Memory bandwidth greater than 1 word per cycle is required since each instruction executed must be fetched, and typically 40% of the instructions will reference memory. To achieve this, two high speed caches are provided, one for instructions (I) and the other for data (D). Each cache has a latency of 1-cycle and can simultaneously deliver one word every cycle. Split I and D caches are chosen over a dual-port or interleaved cache design, since split caches are simpler to implement. The architecture does not support storing into the instruction stream, so the I and D caches do not have to be synchronized. Splitting the caches also allows them to be separately optimized if so desired, (e.g., a small

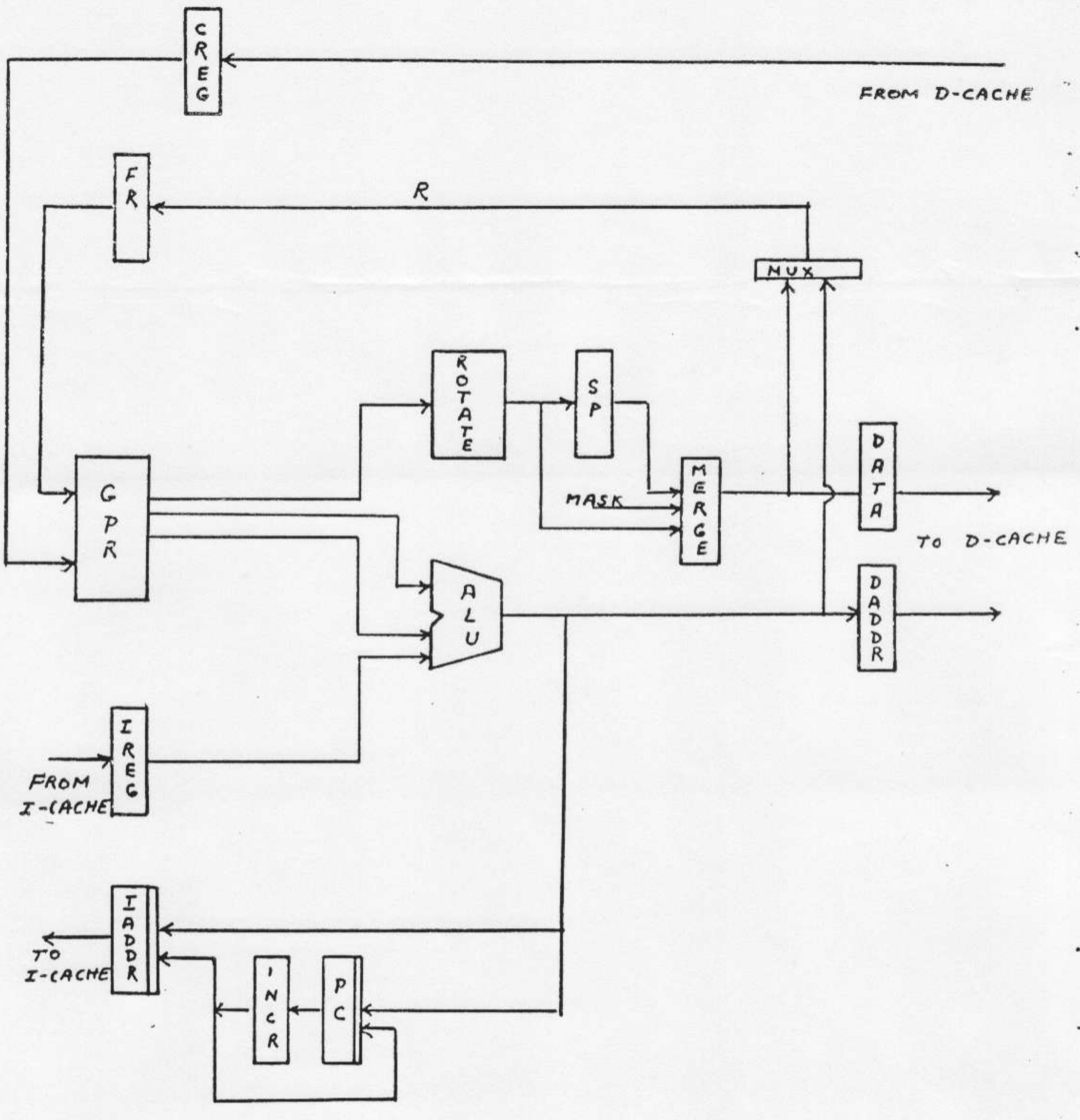


FIGURE 1 : DATA PATHS OF A SIMPLE MACHINE

1-cycle I-cache and a larger 2-cycle D-cache). Data read from the data cache is buffered in a temporary register (CREG) prior to use.

The general purpose registers (GPRs) require 3 read and 2 write ports. Three simultaneous reads are needed to sustain 1 cycle per instruction while executing a series of store instructions. Two writes are needed to update the GPRs from the execution unit and simultaneously from the data cache.

RR operations leave their result in the FR register. The GPRs are designed so that at the start of an instruction execution cycle, FR is written into a GPR and simultaneously "written through" to the execution units if the GPR is being read by the current instruction. The CREG is written through in the same way. Since the register addresses are always in the same fields of the 32-bit instruction, three registers are always read out at the start of each cycle. In parallel, the instruction is decoded. Some or all of the values which were read out are utilized. Progressive indexing does not require any extra data paths. The effective address is left in FR and the standard data path is used to update the base register. The remainder of the data flow is quite straight forward. The timing of the 3 generic instruction types is given below. Updating and accessing the GPRs always takes place simultaneously with the decode of the instruction. For brevity we denote all three operations as "decode".

1. RR ops

```
| Decode  ,  ALU/ROTATE → FR |

        One Cycle
```

2a. Load

```
|            ALU → DADDR      |
| Decode  ,  ALU → FR         |        D-CACHE → CREG        |
```

2b. Store

```
|            ALU → DADDR      |                              |
| Decode  ,  ROTATE → DATA    |        DATA → D-CACHE        |
|            ALU → FR         |                              |
```

3. Branches

```
| Decode  ,  ALU → IADDR      |        I-CACHE → IREG        |
```

The processor has 3 separate cycles. The first cycle of each instruction will be called "AE" (for address generate or execute), the second cycle of loads and stores will be called "DC" (for data cache access), and the second cycle of branch instructions, "IC", (for instruction cache access). Fetching of the next sequential instruction access occurs continuously, and this cycle is also called "IC".

2.1 Cycle Time

Cycle time is dependent on technology and packaging issues which are beyond the scope of this paper. Functions such as GPR access, shift, and add have similar complexity. In ECL technology these functions can be performed in somewhat less than 10 levels of logic, in TTL and NMOS, in somewhat greater than 10 levels. Cache access time is dependent on cache size and the speed and density of array chips. The simple machine has a "fat" cycle since 2 basic functions are performed sequentially, every cycle.

2.2 Timing of Some Sequences

Timing diagrams in this paper have the following format: instructions are labelled with alphabets, A, B, C,.....; time increases horizontally and the cycles are labelled 1, 2, 3, ...; the stages in the pipe are represented vertically.

(1) $R_1 + R_2 \rightarrow R_3$ [A]

$R_3 + R_4 \rightarrow R_5$ [B]

```
IC    | A | B |
AE          | A | B |
         1    2    3
```

(2) Load R_1 , Address [A]

$R_1 + R_2 \rightarrow R_3$ [B]

```
IC    | A | B |
AE          | A |     | B |
DC                | A |
          1    2    3    4
```

At the end of cycle 3, the value destined for R_1 is in FR. B can execute in cycle 4 because of the write through from the CREG to the execution units. Even so, a 1-cycle penalty occurs.

(3) Compare R_1 , R_2 [A]

Branch Address [B]

(Next sequential instruction is C, and the target is T)

```
IC    | A   | B | C | T |
AE          | A | B |     | T |
DC
         1     2    3    4    5
```

If the branch is not taken, C executes in cycle 4. If the branch is taken, the target executes in cycle 5 for a 1-cycle penalty.

2.3 Performance of the Simple Machine

Given the timing diagrams of important sequences of instructions, the performance of the simple machine can be estimated. The machine has a peak performance of one cycle per instruction. However, loads, stores, and branches degrade the performance. Estimated frequencies[1] of occurrence of these instructions are given below:

Loads, stores, and one cycle RR operations have frequencies of 25%, 15% and 40%, respectively. Branches account for the remaining 20%. A third of these are unconditionally taken, a third are conditionally taken and the remaining are not taken.

---

[1] Throughout this paper we will make assumptions about code characteristics, compiler optimizations, cache hit ratios, etc. These numbers can vary substantially from application to application. We do not claim that the numbers are typical. The estimates are based on experience with some large codes and allow us to discuss various performance inhibitors and quantify the effectiveness of the performance enhancement techniques discussed in this paper. We make this disclaimer once and will not repeat it everywhere.

The performance of the simple machine can be estimated as follows: Each instruction takes at least 1 cycle. Based on the timing diagram for branches immediately following a compare, there is no delay for a branch that is not taken and a one cycle delay for branches that are taken. The total penalty for branches, with a one-cycle cache is therefore $0.2 \times (2/3) \times 1 = 0.13$ cycles per instruction. Similarly, loads incur a penalty $0.25 \times 1 = 0.25$ cycles per instruction

Sustained performance assuming all memory references are satisfied by the caches ("infinite cache performance") is therefore $1 + 0.13 + 0.25 = 1.38$ cycles per instruction. If the cache latencies are I for the I-cache and D for the D-cache, the performance is given by $1 + 0.13I + 0.25D$. For $I=D=2$, the performance degrades to 1.76 cycles per instruction. Cache latencies are thus critical.

2.4 Instruction Scheduling

Performance can be improved by scheduling instructions to avoid penalties. Such scheduling need be done only on a local basis and does not involve global analysis. The first technique is to move loads back and introduce instructions between a load and the instruction that needs the data fetched by the load. "Branch and Execute" (BEX) instructions can be used to reduce branch penalties. The semantics of branch and execute are: perform the branch test, generate the target address, and execute the next sequential instruction (Subject). If the branch is taken, execute the target; else execute the instruction following the subject instruction. Load and branch scheduling allow useful work to be done during otherwise empty cycles. We estimate the following statistics: 25% of the load instructions cannot be scheduled, 65% can be moved back one or two instructions and 10% can be moved back only one instruction. Almost all the unconditionally taken branches can be scheduled as branch and execute, as can 50% of the conditional branches. If I-cache and D-cache latencies are each one cycle then only 25% of the loads will incur a penalty (of one cycle) for a total contribution of 0.25 * 0.25 = 0.0625 cycles per instruction. Only half of the conditionally taken branches will incur a penalty. The total branch delay is therefore 0.20 * 0.5 * (1/3) = 0.0333 cycles per instruction. The sustained performance is then 1 + 0.0625 + 0.0333 = 1.1 cycles per instruction.

More generally, the sustained performance is:

$$1 + 0.25\ (.65\ \langle D-2\rangle + .1\ (D-1) + .25D)$$
$$+\ 0.2\ (\ (1/3) * (I-1) + (.5/3) * (I-1) + (.5/3) * (I)\ )$$

where I and D are the instruction and data cache latencies, and <n>=0 if n is negative.

With a single instruction decoded per cycle, the best performance that can be obtained is 1 cycle per instruction. With scheduling, the simple machine achieves a performance of 1.1 cycles per instruction. However, the cycle consists of the time required to do two basic functions. In what follows, we will describe approaches that achieve close to one cycle per instruction at a "lean" cycle time; i.e., the cycle consists of the time required to perform a single basic function.

3. A Pipelined Processor

The performance of the simple machine can be improved if the cycle time is reduced by pipelining the simple data flow. Figure 2 is a variation on Figure 1 where the processing of RR instructions is divided into 2 stages:

```
| Decode        | ALU/ROTATE → FR|
```

The pipeline for loads is

```
| Decode        | ALU → DADDR    | D-CACHE → CREG |
|               | ALU → FR       |                |
```

The pipeline for stores is

```
|               | ALU → DADDR    | D-CACHE → CREG |
| Decode        | ROTATE → DATA  |                |
|               | ALU → FR       |                |
```

The pipeline for branches is

```
| Decode        | ALU → IADDR    | I-CACHE → IREG |
```

The first cycle is now called RD. In this cycle, general purpose registers are accessed and staged in AREG , BREG , and DREG. In the next cycle, address generation or execution takes place; the result is staged in FR and can also be "bypassed" to AREG, BREG, or DREG. We assume for now that the caches remain at 1 cycle latency (possibly by using faster arrays) and will discuss the effect of increased latency later. Data from the D-cache is staged in CREG and can be bypassed to AREG, BREG, or DREG.

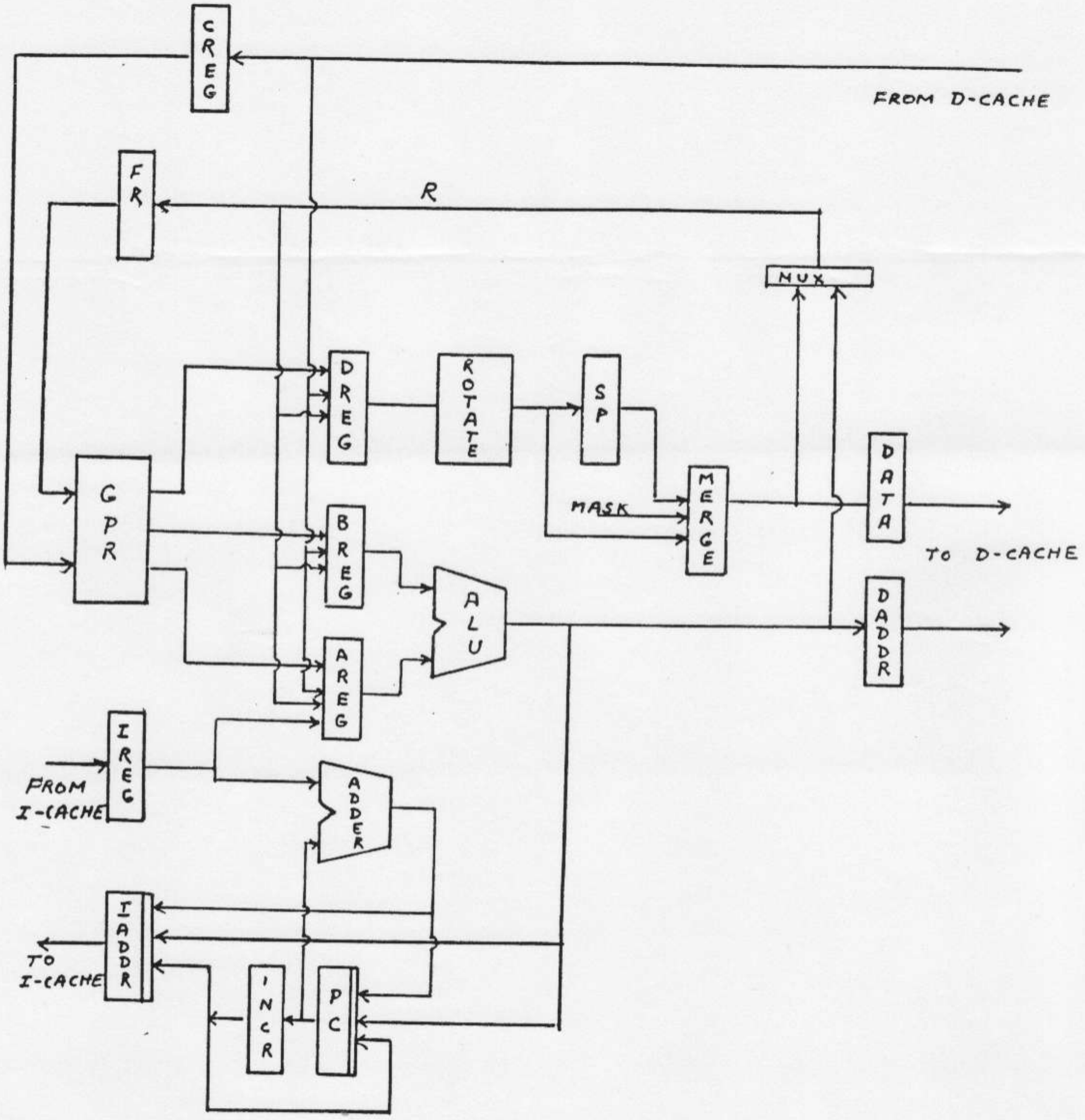


FIGURE 2. DATA PATHS OF A PIPELINED PROCESSOR

Both FR and CREG registers "write through" the GPR file as before. The reason for the separate adder with the IREG will be explained later.

The result and data bypasses improve performance. Consider the sequence

$R_1 + R_2 \rightarrow R_3$ [A]

$R_3 + R_4 \rightarrow R_5$ [B]

$R_3 + R_6 \rightarrow R_7$ [C]

The timing on the pipelined processor without bypasses is:

```
RD      | A |     | B |     | C |
AE            | A |     | B |     | C |
```

The timing on the pipelined processor with bypasses is:

```
RD      | A | B | C |
AE          | A | B | C |
          1   2   3   4
```

In cycle 1, registers R_1 and R_2 are accessed and staged in AREG and BREG. In cycle 2, the add takes place; simultaneously, the access of R_3 and R_4 occurs. During the decode of instruction B in cycle 2, it is determined that the correct value for R_3 will be on bus R late in cycle 2. At the end of cycle 2, bus R is gated into AREG (instead of R_3). During cycle 3, B executes and operands are accessed for instruction C. The correct value of R_3 is now in FR. At the start of cycle 3, R_3 is being written from FR and also being read. This will cause the contents of FR to appear at the output port of the register file. This example illustrates that whether an operand is in the register file, being computed or waiting in FR for put away, it is available for use. The bypass from the cache is required for the same reason. Consider the sequence:

Load R_3, Address [A]

$R_3 + R_4 \rightarrow R_5$ [B]

$R_3 + R_6 \rightarrow R_7$ [C]

The timing with the cache bypass is:

RD	A		B	C	
AE		A		B	C
DC			A		
	1	2	3	4	5

At the end of cycle 3, data on the cache bus is bypassed into AREG and also latched in CREG. Without the bypass an additional cycle would be lost. Ignoring the IREG adder, the timing for branches is given below.

CR R_1, R_2 [A]

Branch [B]

IC	A	B	C		T		
RD		A	B	C		T	
AE			A	B			T
	1	2	3	4	5	6	7

In cycle 4, address generation of the branch takes place. In cycle 5, the target is fetched and a 2 cycle delay occurs. The next sequential instruction, C, is conditionally decoded in cycle 4. (If this is not done, a 1-cycle delay will occur even if the branch is not taken).

3.1 Performance of the Pipelined Processor

Performance of pipelined processors can be estimated by examining any one stage and accounting for empty cycles in that stage. This is particularly important in the presence of instructions which execute for more than one cycle in a given stage. For example, a branch delay may not show up in the AE stage if the latter is executing a multicycle operation when the branch is encountered. The analysis in this section is based on activity in the AE stage. To determine the benefits of various mechanisms being proposed, three cases are discussed below. At the end of this section, additional techniques to improve performance will be presented. The code and scheduling statistics given earlier will be used here.

CASE 1 (No cache bypass and no scheduling of loads or branches)

The penalty for a load followed by a dependent operation is 2 cycles. The total penalty due to loads is therefore $0.25 \times 2 = 0.5$ cycles per instruction. The penalty for a taken branch is 2 cycles. With one conditional decode, there is no penalty for a branch which is not taken. Since two thirds of the branches are taken, the branch penalty is 0.2 *

(2/3) * 2 = 0.27 cycles per instruction. The sustained performance is 1 + 0.5 + 0.27 = 1.77 cycles per instruction.

<u>CASE 2</u> (Bypasses but no scheduling of loads or branches)

With both bypasses, the penalty due to loads is reduced to 1 cycle. The performance is: 1 + 0.25 + 0.27 = 1.52 cycles per instruction.

<u>CASE 3</u> (Bypasses and scheduling of loads and branches)

Seventy five percent of the loads can be moved back 1 or 2 instructions, and in these cases there is no penalty. For the remaining 25%, the penalty is 1 cycle. The penalty due to load instructions is 0.25 × 0.25 × 1 = 0.0625 cycles per instruction. A third of the branches (unconditionally taken) can be scheduled as branch and execute and the penalty is 1 cycle; a third are conditionally not taken and these do not incur any penalty; of the remaining third, 50% can be scheduled and the penalty is 1 cycle. The remaining branches cause a 2 cycle penalty. The additional delay due to branches is therefore 0.2 * [(1/3) * 1 + 0.5 × (1/3) * 1 + 0.5 * (1/3) × 2] = 0.167 cycles per instruction. The performance is: 1 + 0.063 + 0.167 = 1.23 cycles per instruction.

3.2 Relative Branches

The branch penalty (14% of the total cycles per instruction) can be reduced further by introducing branches relative to the program counter.

We estimate that 90% of all branches can be coded relative to the program counter (with a range of ± 32K words). This is important because target address generation for relative branches can be done without GPR access. The IREG adder is used to compute the target address for relative branches. In the RD cycle of every instruction, while register access and decode are proceeding, the displacement field of the IREG is added to the Program Counter (PC). The PC is also incremented. Late in this cycle, a decision is made to gate the next sequential instruction address or the "guessed" target address. If the branch is not relative, a full address generation occurs in the next cycle. This is best illustrated by an example:

Compare R_1, R_2 [A]

Branch Relative [B]

The timing on the pipelined machine is:

IC	A	B	C	T		
RD		A	B		T	
AE			A			T
	1	2	3	4	5	6

Several things occur simultaneously in cycle 3. During the RD cycle, the address adder always adds the displacement field to the PC; PC+4 is also computed. During cycle 3, it will be determined whether instruction B is a branch, and if so, whether it is relative. Late in cycle 3, the compare will complete and the condition will be resolved. If instruction B is not a branch, or if it is a branch but is not taken, PC+4 is gated to the I-cache at the end of cycle 3. If B is a relative taken branch, PC+displacement is gated. Otherwise, B is a taken non-relative branch and a full AE cycle is required. This would occur in cycle 4. The penalty for a taken branch can thus be reduced to 1 cycle for a majority of the branches. We estimate that 90% of the branches in each category (unconditional, conditional taken, conditional not taken) are relative. The branch penalty is then reduced to:

$$.2\ [(1/3) \times 1 \times 0.1 + (1/3) \times .5 \times .1 + (1/3) \times .5 \times 1 \times .9 + (1/3) \times .5 \times 2 \times .1\ ] = 0.047 \text{ cycles per instruction.}$$

The overall performance with special handling of relative branches is: $1 + 0.063 + 0.047 \approx 1.1$ cycles per instruction.

The various techniques described in this section bring the performance of the pipelined machine down to 1.1 cycles per instruction at a cycle much leaner than that of the simple machine. A single basic function is performed in each cycle.

3.3 Increased Cache Latencies

The effect of increased cache latencies is presented next for the sake of completeness . Again, let I and D be the latencies of the instruction and data caches, respectively. Load penalties are given by:

$$0.25\ [\ 0.65 \langle D-2\rangle + 0.1\ (D-1) + 0.25\ (D)\ ] = (D-1.4)/4 \text{ for } D > 1$$

Branch penalties can be calculated from the table below:

CASE				Freq.	Penalty	
Conditional	Taken	Relative	BEX		1~Cache	I~Cache
Y	N	-	-	(1/3)	0	0
N	Y	Y	Y	(1/3)×.9	0	I-1
N	Y	N	Y	(1/3)×.1	1	I
Y	Y	Y	Y	(1/3)×.9×.5	0	I-1
Y	Y	N	Y	(1/3)×.1×.5	1	I
Y	Y	Y	N	(1/3)×.9×.5	1	I
Y	Y	N	N	(1/3)×.1×.5	2	I+1

The branch penalty is:

$$(0.2) * (1/3) [0.9 * (I-1) + 0.1 * I + 0.9 \times 0.5 * (I-1)$$
$$+ 0.1 * 0.5 * I + 0.9 * 0.5 * I + 0.1 * 0.5 * (I+1)]$$
$$= (0.4I - 0.26)/3 \text{ cycles per instruction}$$

For I=D=2, the performance is 1.33 cycles per instruction and goes up to 1.71 cycles per instruction if the latencies are 3 cycles.

3.4 Additional Enhancements

Several additional techniques can be utilized to improve performance by reducing branch and load delays. The total delay due to branches depends on the number of branches dynamically encountered, the delay due to condition resolution and the delay in fetching the target.

1. The branch and execute instruction can reduce the effect of the delay in fetching the target by scheduling an instruction (which occurs prior to the branch) as a subject instruction. Branch and execute can be extended to allow more than one subject instruction. This is useful if the latency of the I-cache is greater than 1 cycle. For additional flexibility, instructions from the target stream can be scheduled as subjects by introducing a "branch and execute N or skip" instruction. If the branch is taken, the N subject instructions are executed while the remainder of the target is being fetched. If the branch is not taken, the N subject instructions are skipped. If a skip can be implemented faster than a target fetch, branch penalties can be reduced.

2. A standard loop closing instruction, "DBR COUNT, Address" has the following semantics: "Decrement COUNT. If the result is not equal to zero then branch." The delay in condition resolution can be improved by providing a more efficient instruction: "BRD COUNT, Address" which specifies that "if COUNT is not equal to zero then branch and decrement COUNT". In this case, the branch is immediately resolved after register access and does not have to wait for a full addition. The BRD instruction

can be extended to allow subject instructions. The example below illustrates the code generation changes.

Program

```
       DO  10  I  =  1 , N
       A(I)  =  B(I)
10     CONTINUE
```

Pseudo Code Using Standard Loop Closing

```
       COUNT  =  N
LOOP:  A(N - COUNT + 1)  =  B(N - COUNT + 1)
       DBR COUNT,LOOP
```

Pseudo Code Using New Loop Closing

```
       COUNT  =  N - 1
LOOP:  A(N - COUNT)  =  B(N - COUNT)
       BRD COUNT LOOP
```

Finally, a generalization of "auto increment" index addressing can be used to reduce load penalties. In this case, "Load R1, R2, R3" specifies that the effective operand address is in R2. While the D-cache is being accessed, R2 and R3 are added and the result is placed in R2. The advantage over progressive indexing is that the operand address is available immediately after register access and a full address generation cycle is not required. When this instruction can be used, the load penalty is reduced by 1 cycle without scheduling.

3.5 Comments

It should be clear from the analysis in the previous sections that compiler scheduling is important. Like other processors, the pipelined machine described here benefits from compiler optimizations such as register allocation, common subexpression elimination, reduction in strength, constant propagation, etc. Except for "branch and execute N or skip", the additional scheduling discussed in this paper is all done locally within a basic block. Simple minded algorithms for load scheduling and branch and execute take $O(n^2)$ and $O(n)$ time, respectively, when n is the number of instructions in a basic block. "Branch and execute N or skip" requires global flow analysis.

In the remainder of this paper, we present techniques to execute long operations efficiently and discuss the penalties incurred due to finite caches together with approaches that reduce these penalties.

4. Caches

The performance of the pipelined machine described in section 3 is 1.1 cycles per instruction with a cache latency of one cycle. Increasing the cache latency can have a large impact on performance. Moreover, all performance estimates up to now have been "infinite cache", i.e. assuming all references are satisfied by the cache. The finite cache penalty can be calculated as:

$$FCP = m * P \text{ cycles per instruction}$$

where m is the miss ratio (in misses per instruction) and P is the average penalty per miss. If a miss occurs every 20 instructions and P is 20 cycles, performance decreases from 1.1 cycle per instruction to 2.1 cycles per instruction. The finite cache penalty can thus be a first order determinant of performance. In what follows, the constituents of this penalty are analyzed in more detail. Architecture and machine organization approaches to reduce the penalty are discussed.

The penalty per miss can be divided into two main parts: the leading edge delay (LED) and the trailing edge delay (TED). The leading edge delay is the number of cycles the processor is delayed until the requested word is supplied by the memory system. Included in this is the access time of the next level in the memory hierarchy and transmission time between the two levels. The leading edge delay can be greater when the processing of a miss is delayed because a previous miss is in progress ("clustering"). For a store-in-cache, a modified line must be written back to memory ("cast-out"), before a line can be brought in ("put away"). Depending on the cache system design, this can increase the leading edge delay. After the processor receives the requested word (which caused a miss), it competes with the putaway and cast-out for cache cycles and can be delayed further. This is the trailing edge delay.

The leading edge delay can be reduced by (1) accessing the requested word first and bypassing it directly to the processor and (2) by placing another (larger and slower) cache between the first level and the main memory. The delay due to clustering can be reduced by providing multiple miss facilities and increased memory bandwidth, but this is expensive. The trailing edge delay can be reduced by (1) reducing the size of the cache line, but this can increase the miss ratio, (2) by increasing the size of the unit that is putaway in the cache every cycle, or (3) by buffering the requested line and storing it in the cache during free cycles.

4.1 An Example of a Cache System

A particular cache system is presented and analyzed in this section as an example.

	I-CACHE	D-CACHE	LEVEL 2	MAIN MEM(L3)
Size (KB)	32	32	1024	1M
Line Size (B)	32	32	1024	4096
Associativity	4	4	4	Full
Latency (CYCLES)	1	1	4	20
Access rate (BYTES/CYCLES)	4	4	16	16
Putaway rate (BYTES/CYCLES)	4	4	16	16
Store-in ?	No	No	Yes	

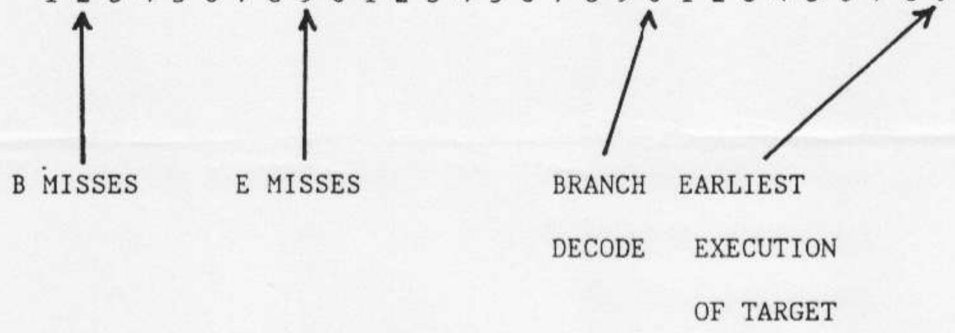
Comments

On an I-cache miss, words are continuously bypassed to the processor. On a D-cache miss only the requested word is bypassed. The L2 cache has a 1KB cast-out buffer. A modified line in L2 is first read out into this buffer and then stored in main memory. The requested line from main memory is placed in a 1KB read buffer and written into L2 after the cast out, if any, is completed. For this system, various delays are estimated below.

I-Cache

On an I-cache miss, the missing instruction is fetched from the L2 cache. Assuming a hit in L2, the leading edge delay is 4 cycles. Subsequent instructions are bypassed to the processor. If no branch instructions are encountered during the putaway, there is no additional delay. A second miss can be caused by a taken branch or sequential instruction fetch across a cache line boundary. Figure 3 illustrates a specific situation. Instructions B and E miss in the I-cache and F is a branch. The instruction fetch of B misses in the I-cache in cycle 2. B is available 4 cycles later. From cycle 7 to 14 the line containing B is put away in the I-cache. The instruction E also misses in the I-cache because it happens to be in a new line. The access of this line from L2 does not start until the previous put away is complete. E is available for decode in cycle 18. The next sequential instruction, F, is a branch whose target is in the I-cache. However, it cannot be fetched until cycle 27 because the I-cache is busy with a put away. In this example, the penalty is 19 cycles for 2 misses or 9.5 cycles per miss.

```
IC  A B X X X X P P P P P P P P X X X X P P P P P P P P T
RD    A         B C D                 E F               T
AE      A - - - - B C D - - - - - - - - - E F   - - - - - - T
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```



X : L2 ACCESS

P : LINE PUTAWAY

- : CYCLES LOST DUE TO I-CACHE MISSES

19 CYCLE PENALTY FOR TWO CLUSTERED MISSES

FIGURE 3: I-CACHE MISS PENALTY

D-Cache

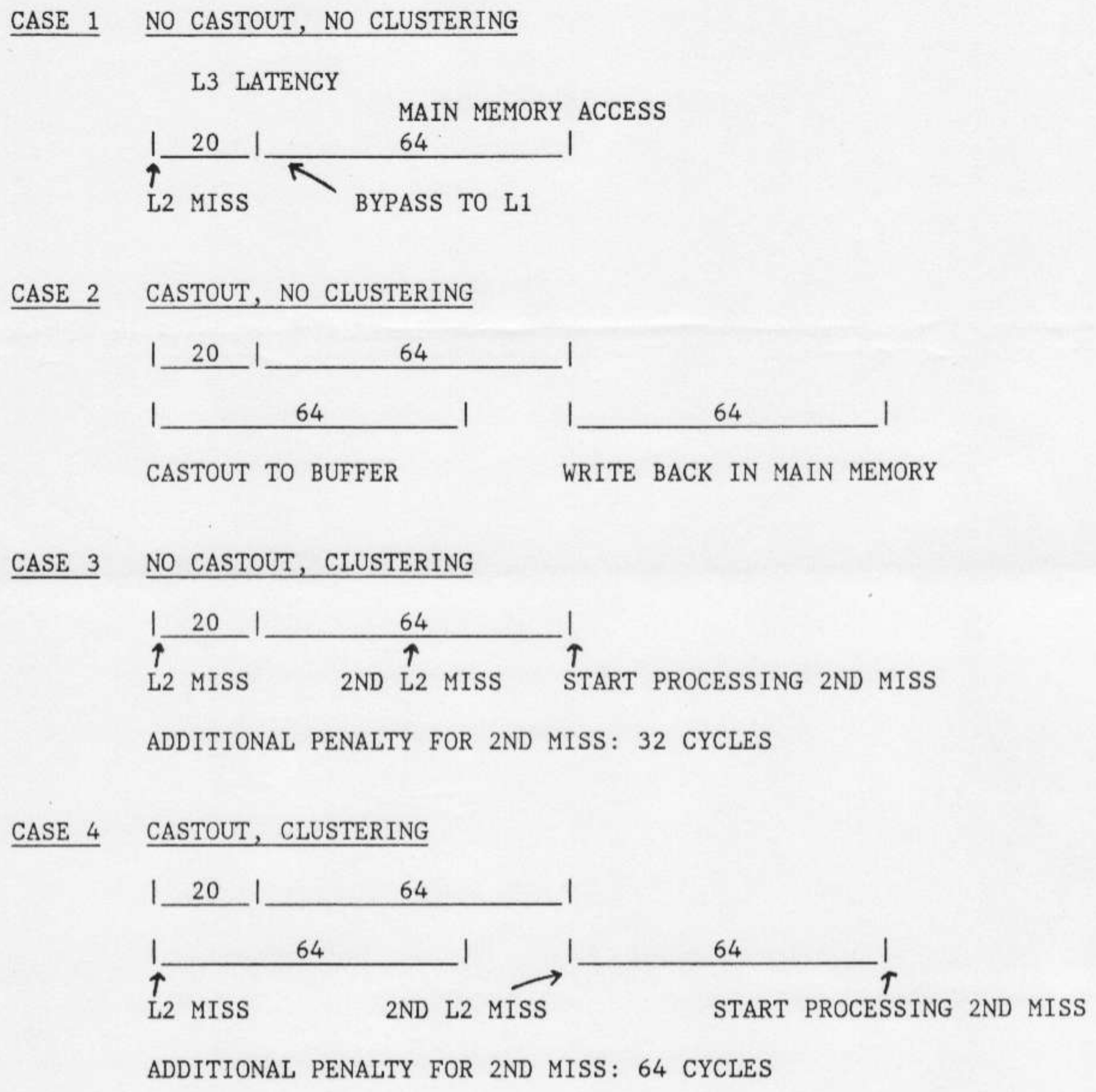
We will assume the worst case here. For example, the load which misses is followed by another load (which requires a cache access). Since the cache is busy for 8 cycles during the line putaway, the full 4+8=12 cycle penalty is taken for the miss. There are no additional penalties due to clustering.

L2 - Cache

The different L2 miss cases are illustrated in Figure 4.

In Case 1, there is no cast-out from L2 and there is no clustering (no L2 miss occurs until the previous one is completely processed). Twenty cycles after the L1 miss, the quadword (16 bytes) containing the requested word (4 bytes) is available at L2 and is bypassed to L1. Complete access of the 1KB line takes 64 cycles. Because of the bypass to L1 these 64 cycles do not show up as a delay in the processor. The L2 delay, seen at the processor is 20 cycles.

In Case 2, the line being replaced in L2 has been modified and must be cast out (written back to main memory). There is however, no clustering of L2 misses. As soon as the main memory access is initiated, the replaced line is read out of L2, into a cast out buffer. This takes 64 cycles. When the read out from main memory is complete, the line buffered in the cast out buffer is written into memory. Twenty cycles after the



ADDITIONAL PENALTY FOR 2ND MISS: 32 CYCLES

ADDITIONAL PENALTY FOR 2ND MISS: 64 CYCLES

FIGURE 4: L2 MISS PENALTIES

L2 miss occurred, the appropriate quadword returns to L2 and is bypassed to L1. The L2 delay as seen by the processor is still 20 cycles.

In Case 3, there is no cast out but a second L2 miss occurs while the first one is being processed. Assume that the second miss occurs midway between the 64 cycle readout of the previous L2 line. The processing of the second miss waits until the first has completed and is delayed for an additional 32 cycles.

In Case 4, both cast out and clustering occur. We assume that the miss occurs midway between the period when the main memory is busy with the first request, i.e., just after the readout from main memory and just before the write back of the line cast out of L2. In this case, the processing of the second L2 miss is delayed an additional 64 cycles.

To estimate the finite cache penalty, the frequency with which various events occur is needed. We assume that an I-cache miss occurs every 60 instructions, a D-cache miss occurs every 50 instructions, and an L2 miss occurs every 300 instructions on the average. We further assume that due to clustering, an instruction miss incurs a 9.5 cycle penalty as in the example. We assume the probability of cast out from L2 is 0.25; when there is no cast out, the second miss occurs half way through the main memory access with probability 0.5 and after the main memory access with probability 0.5; when there is a cast out, the second miss occurs immediately after the main memory access with probability 0.8 and after the main memory writeback with probability 0.2. We can now estimate the total finite cache penalty. The instruction cache miss penalty is (1/60) * 9.5 = 0.16 cycles per instruction. The data cache miss penalty is (1/50) * 12 = 0.24 cycles per instruction. For L2 misses without castout the average penalty is 0.5 * 20 + 0.5 * 52 = 36 cycles. In the presence of castout, the average penalty is 0.8 * 84 + 0.2 * 20 = 71 cycles. Since the castout occurs with probability 0.25, the average L2 miss penalty is 0.25 * 71 + 0.75 * 36 = 45 cycles. The degradation due to L2 misses is then (1/300) * 45 = 0.15 cycles per instruction. The total cache miss penalty is the sum of the contributions from the I-cache, the D-cache and L2 and is 0.16 + 0.24 + 0.15 = 0.55 cycles per instruction. Without the Level 2 cache, the penalty would be approximately 1 cycle per instruction with simple controls.

We stress that this exercise is for illustrative purposes only to identify the various constituents of the finite cache penalty and a very crude method of estimating this penalty. For a given cache system, accurate estimates must be obtained through detailed trace driven simulation. It should be clear from this example that finite cache effects are a first order determinant of performance and that clustering, cast outs, and line putaway can increase delays significantly, beyond basic memory latencies.

4.2 Architectural Support for Caches

Cache penalties can be reduced by providing explicit cache management instructions [2]. This approach is very important for high performance systems. Examples of such instructions are:

(1) Establish a line in the cache without moving any data from the next level in the hierarchy.

(2) Invalidate a line.

(3) Write back a line if modified.

The first two can be used to reduce the traffic between the cache and the next level. If a new temporary storage area is being created, instruction (1) can be used. If computed values have been used and are no longer required, instruction (2) can be used. With (2) and (3), I/O can proceed directly to main memory; cache lines can be flushed and invalidated only when necessary. If this software approach is not used, hardware designers have two options for handling I/O:

1. Move the I/O data physically through the cache. This reduces the hit ratio.

2. Interrogate the cache directories on every access and invalidate or flush as necessary. This slows down the processor due to cache interference created by the I/O.

With 1MB of L2 and 1GB of main memory (as in the example cache), a large amount of mapping information must be maintained to control the caches. Efficient mapping mechanisms can be designed if it is guaranteed, through software conventions, that there is no aliasing: i.e. the same physical data will never be referred to by two different virtual addresses. One such mapping mechanism is described below.

Let the virtual address be 32 bits V(0-31) where bit 31 is the least significant bit and let the virtual page size be 4KB. Each of the I and D-caches is 32KB, 4-way set associative and has a line size of 32B. V(17-24) is used to access one of the 256 congruence classes. All 4 sets in the directory and cache arrays are read out simultaneously in one cycle. Late in the cycle, the information in the directory is compared against the 17 high order bits of V to determine whether the requested word is in the cache and if it is, one of the four words read out of the cache arrays is transmitted to the processor. Virtual to real translation is done on 4KB pages. The directory was accessed using 8 bits, of which V(17-19) are subject to translation. If aliasing were allowed, 8 congruence classes would have to be searched to determine a cache miss. In the absence of aliasing, a single search is sufficient. When the directory is organized as above, the cache is called a "virtual" cache since it is accessed using a virtual address. If a virtual cache is used, cache management instructions can specify virtual addresses and need not be privileged operations. Program execution can then be optimized without operating system calls.

L2 is 1MB, 4-way associative, and has a line size of 1KB. The same mechanism can be used for L2 as well. A congruence class is accessed using V(14-21). V(0-13) is compared against the directory entry contents to determine a match and one of 4 lines is selected after the L2 arrays are accessed. If L2 is not busy, the L1 and L2 directory accesses can be done in parallel to reduce the leading edge delay on an L1 miss. The corresponding main memory address can be kept in the L2 directory entries to locate a line in main memory rapidly. In addition, a full inverted page table for L3 can be maintained in memory and accessed by a small finite state machine to provide full coverage of main memory [4]

5. Long Operations

Consider a reduced instruction set processor with added instructions for frequently performed complex operations such as "Move Characters" and Multiply. Assume that the frequencies are: Branches 20%, Loads 25%, Stores 15%, Multiplies 2%, and one cycle register to register instructions 35.95%. Assume further that short character string moves (< 8 bytes) are 2% and long moves (1000 bytes are 0.05%).

Assume that a multiply takes 4 cycles, a 1000 character move 500 cycles, and the short move 5 cycles. The performance of the pipelined machine of Section 3 can be calculated by adding the extra cycles for the long operations (appropriately scaled by frequency) to the base performance of 1.1 cycles per instruction. The performance is $1.1 + 0.02 \times 3 + 0.0005 \times 499 + 0.02 * 4 = 1.5$ cycles per instruction where the instruction set has both simple and complex instructions. If the complex functions are coded using a sequence of simple instructions, the performance remains close to 1 cycle per instruction, but the number of instructions executed increases. In either case, the complex functions must be executed as efficiently as possible. In this section we discuss four categories of complex instructions: Storage to register (RX) operations, character moves, fixed point multiply, and floating point arithmetic. Architectural support for character string moves and fixed point multiply was defined in the original 801 architecture [2].

5.1 RX Instructions

RX instructions can be included in an architecture to improve performance, reduce register usage and reduce code space. Since our focus is on performance, we will consider the first point in more detail. The computation $J = I_1 + I_2 + \ldots\ldots + I_n$ requires n+2 instructions if RX adds are used and 2n+2 if they are not. On some processors, this can result in improved performance; on the processors discussed in this paper, it does not. The RX add requires two additions, one for address generation and the other for execution. Our pipelined machine has a single adder, and an RX add will therefore take at least two cycles. The execution of sequence of RX adds (A, B, C, D, ...) on the pipelined machine is given below:

IC	A	B	C			D				
RD		A	B			C	D			
AE			A	B	A	B	C	D	C	D
DC				A	B			C	D	
	1	2	3	4	5	6	7	8	9	10

To achieve this speed, control information about A must be maintained in cycles 4 and 5. The same performance can be obtained on the pipelined machine without RX instructions and simple controls by scheduling loads.

To obtain a speed of 1 cycle per instruction on RX adds, a processor with 2 fixed point adders is required. A straightforward approach is to pipeline the processing into 5 stages:

```
| I FETCH | DECODE  |  AGEN   | D FETCH | EXECUTE |
```

Though many variations are possible, one generic approach is discussed in more detail. Two copies of the general purpose registers are provided, one for the Address Generation adder and the other for the Execute adder. The copies are kept identical by simultaneously updating both of them. The instruction is fetched in the first stage. The decode and register access for address generation takes place in the second stage. The address is computed in the third stage. In the fourth stage, data is fetched from the cache and the registers are accessed. Execution takes place in the fifth stage. A simple RR add could execute in stage 2, but this complicates the hardware. To keep the controls and data paths simple, all instructions go through the 5-stage pipeline, and execute in the last stage. We will call this machine the two adder machine. The performance of the pipelined machine and the two adder machine is compared on two sequences to determine the advantages of RX operations.

Sequence 1

Add R_1, ADDRESS	[A]
$R_2 + R_3 \rightarrow R_4$.	[B]
COMPARE R_4 R_1	[C]
BRANCH T	[D]

(Branch is taken)

Timing on the two adder machine

IC	A	B	C	D			T				
RD		A	-	-	D			T			
AG			A	-	-	D			T		
DC				A	-	-	-			T	
EX					A	B	C	D			T
	1	2	3	4	5	6	7	8	9	10	11

Cycle 7 is the earliest point at which the target T can be fetched (without branch prediction mechanisms). The target is fetched as soon as the address is available but prior to branch resolution. This complicates the controls. Execution completes in cycle 11.

Code for the pipelined machine after scheduling

Load R_5, ADDRESS	[A]
$R_2 + R_3 \rightarrow R_4$	[B]
$R_1 + R_5 \rightarrow R_1$	[C]
COMPARE R_4, R_1	[D]
BRANCH T	[E]

Timing on the pipelined machine

IC	A	B	C	D	E			T		
RD		A	B	C	D	E			T	
AE			A	B	C	D	E			T
DC				A						
	1	2	3	4	5	6	7	8	9	10

The target is fetched after branch resolution and is initiated one cycle later. However, the sequence completes one cycle earlier.

Sequence 2

Add R_1, ADDRESS	[A]
$R_1 + R_2 \rightarrow R_3$	[B]
Add R_1, (R_3)	[C]
$R_3 + R_4 \rightarrow R_3$	[D]

Timing on the two adder machine

IC	A	B	C			D				
RD		A	-			C	-			
AG			A	-			C	-		
DC				A	-			C	-	
EX					A	B			C	D
	1	2	3	4	5	6	7	8	9	10

Code for the pipelined machine after scheduling

Load R_4, ADDRESS	[A]
$R_1 + R_4 \rightarrow R_1$	[B]
$R_1 + R_2 \rightarrow R_3$	[C]
Load R_5, (R_3)	[D]
$R_3 + R_4 \rightarrow R_3$	[E]
$R_5 + R_1 \rightarrow R_1$	[F]

Timing on the pipelined machine

IC	A	B		C	D	E	F		
RD		A		B	C	D	E	F	
AE			A		B	C	D	E	F
DC				A				D	
	1	2	3	4	5	6	7	8	9

Though two more instructions are executed, the final add completes one cycle earlier.

RX operations require additional code points, and this can be significant if the full cross product of {loads}×{RR operations} is supported. For processors like the pipelined RISC machine, RX instructions can be included to save code space if the code points are available. They should be used only when nothing can be scheduled between a LOAD R_2, ADDRESS and R_1+R_2→R_1 sequence. For machines like the two adder machine, significant performance improvement can be obtained on some sequences; on others the utility is questionable.

5.2 Character String Moves

Moving an arbitrary string of bytes from one location to another is a common operation. The preferred semantics are described below assuming that the source and target have the same number of bytes, B. Let the SOURCE string be in addresses S, S+1, ..., S+B-1 and let the TARGET string be in addresses T, T+1, ..., T+B-1. The effect of the move, TARGET ← SOURCE, should be

```
TEMP   ←  SOURCE
TARGET ←  TEMP
```

The 370 architecture has MVC and MVCL instructions for moving character strings, neither of which has these semantics. The operation of MVC is:

```
FOR I = 1 to B DO
   T(I-1) ← S(I-1)
```

By setting S=T-1, the character in location S can be propagated throughout memory. In the case of MVCL, no move takes place if $S \le T \le S+B-1$ (Destructive overlap). Otherwise, the move takes place according to the preferred semantics. (MVCL is actually much more complex and general than described here). In this section we will discuss approaches to executing character moves on the pipelined RISC machine.

The move can be implemented without use of a temporary string as follows:

```
IF T≤S THEN
     FOR  I = 1 TO B DO
          T(I-1) ← S(I-1)
     ELSE
     FOR  I = 1 TO B DO
          T(B-I) ← S(B-I)
END.
```

Thus, if T < S, the move starts from the beginning of the string and proceeds forward. If T >S the move starts from the end of the string and goes backward.

Figure 5 shows a character string move of 8 bytes. To accomplish this on a register machine, two functions are required: "extended shift" and "partial store". The extended shift function allows the entire source string to be shifted left or right across word boundaries. In Figure 5, the source and target strings are not aligned with respect to each other or on word boundaries. The source string must be shifted left one position before it can be stored at the target address. This is accomplished by fetching a word, rotating it, leaving the result in a special register, merging the rotated word with the previous contents of the register and

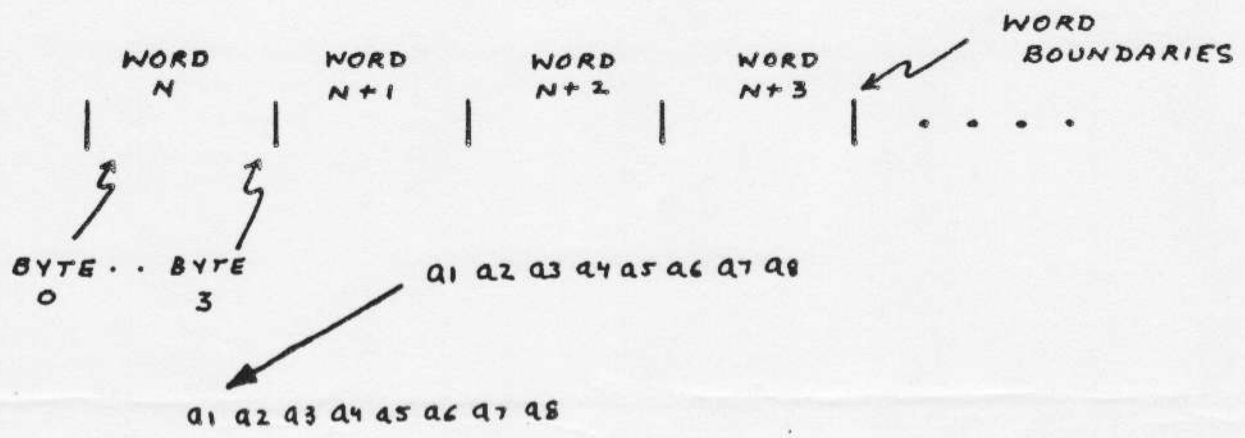

WORD N
WORD N+1
WORD N+2
WORD N+3
WORD BOUNDARIES
BYTE 0
BYTE 3
a1 a2 a3 a4 a5 a6 a7 a8
a1 a2 a3 a4 a5 a6 a7 a8

FIGURE 5

storing the result. For the first and last words stored, only selected bytes are changed (refer to figure 5). A partial store function is required to achieve this. Statistics indicate that most character moves are quite long or very short. The approach for long moves is described first. An instruction ROTATE AND STORE RC, RT, RN where register RC contains the data (D), RT contains the target byte address (T), and register RN specifies the rotation (N) is used. |N| is always a multiple of 8 and is the amount of the rotation (0, 1, 2 or 3 bytes). If N is negative a left rotation is specified else a right rotation. A mask M1 is generated as specified below:

Generation of Mask M1

ROTATE AMOUNT	DIRECTION	MASK
0	Left	1111
8	Left	1110
16	Left	1100
24	Left	1000
0	Right	1111
8	Right	0111
16	Right	0011
24	Right	0001

The Rotate and Store instruction executes on the data paths of the pipelined machine as specified below:

RD CYCLE

Access general registers RC, RT, and RN

AE CYCLE

Rotate D as specified by N

Place the result Y in register SP

```
        Generate M1

        Merge Y with the previous contents of SP to get X
          (For each byte, select from Y if M1 is 1 else select from SP)

        Increment T by 4

        Place this value in FR for updating RT

DC   CYCLE

        Store X at the new value of T
```

Storage is addressed as shown in Figure 5. Let S and T be the source and target addresses. Assume that T≤ S, i.e., the move starts from the beginning of the strings. Let s = S mod 4 and t = T mod 4. Let N = (t - s) * 8. Let S, T - 4, and N be in general registers RS, RT, and RN, respectively. Ignoring boundary conditions, the innermost loop for a long move is given below:

```
LOOP: Load RC from address in RS; RS ← RS + 4
      ROTATE AND STORE RC, RT, RN
      COMPARE RT, RLIMIT
      BRANCH LT, LOOP
```

The loop can be optimized:

```
LOOP: Load RC from address in RS; RS ← RS + 4        [A]
      COMPARE RT, RLIMIT'                             [B]
      BRANCH LT AND EXECUTE RELATIVE, LOOP            [C]
      ROTATE AND STORE RC, RT, RN                     [D]
```

Where RLIMIT' = RLIMIT - 4 the timing on the pipelined machine is

IC	A1	B1	C1	D1	A2						
RD		A1	B1	C1	D1	A2					
AE			A1	B1	C1	D1	A2				
DC				A1			D1	A2			

The move occurs at one byte per cycle. This is half the maximum possible speed on the pipelined machine since a load a store of a word each takes one cycle. Data can be moved at close to two bytes per cycle by unrolling the loop.

We now discuss an approach for moving short character strings (≤ 9 bytes) rapidly. Figure 6 describes the basic hardware algorithm. The move is achieved by loading (at most) three general registers, rotating the data appropriately and storing in the target location. Let B = number of bytes to be moved, S = the source address, T = the target address, s = S mod 4, t = T mod 4, s' = 4 - s, and t' = 4 - t. t and t' are recalculated from T each time they are used. ROTATE LEFT/RIGHT RO AND STORE RT is used in the flowchart to specify hardware controls similar to the ROTATE instruction described previously except that the direction of rotation is explicit and the amount is calculated separately. In addition to M1, a 4-bit mask M2 consisting of 1s surrounded by 0s is generated as follows: The number of left hand zero bits = t; the number of right hand 0 bits = 4 - (t + "number of bytes stored"). In the DC cycle data is selectively stored in the cache, i.e., only those bytes in the word are stored for which M2 = 1. The reader should simulate the algorithm in Figure 6 for

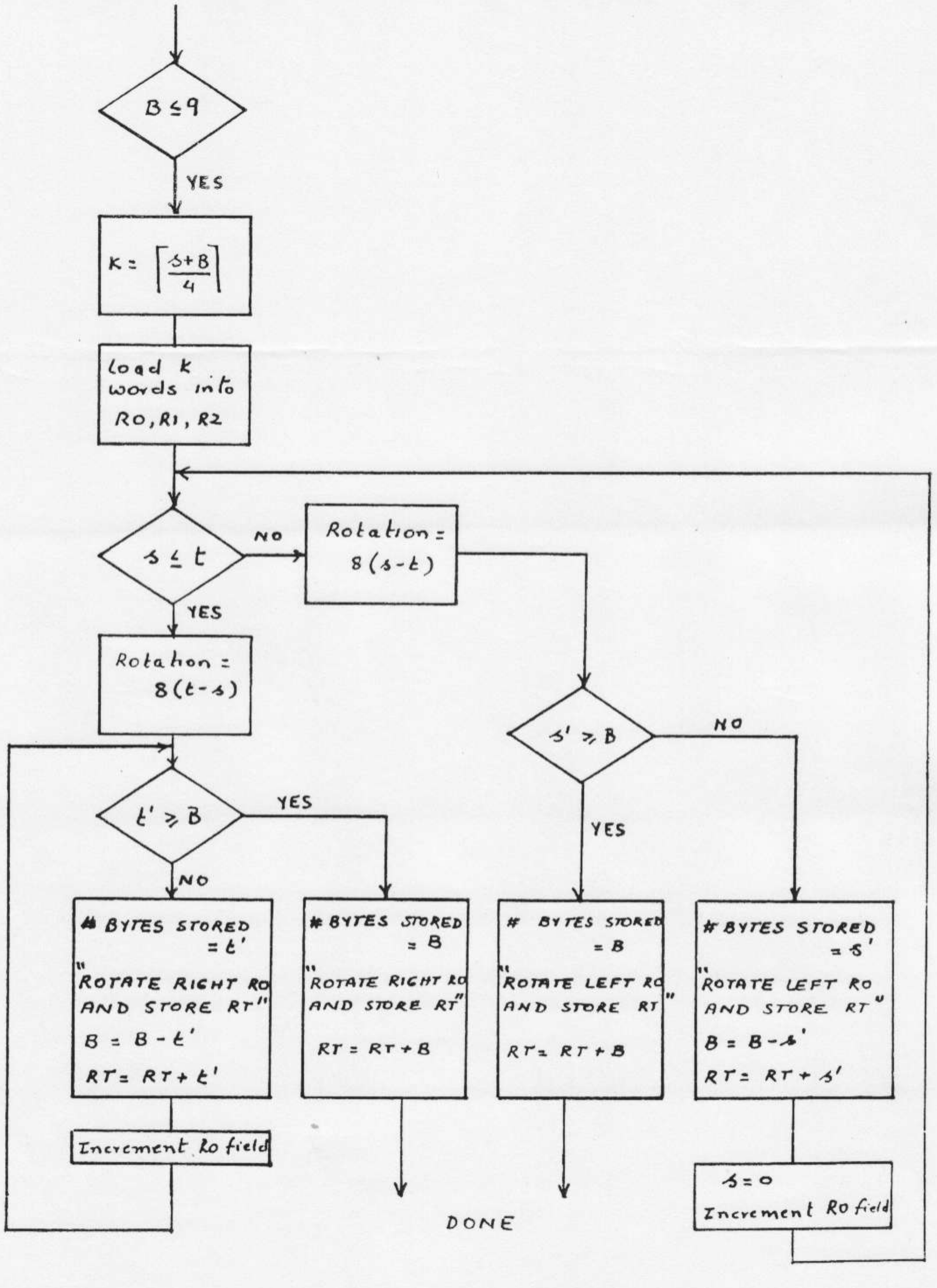

B ≤ 9
YES
K = ⌈(s+B)/4⌉
load k words into R0, R1, R2
s ≤ t
NO
Rotation = 8(s-t)
YES
Rotation = 8(t-s)
s' ≥ B
NO
YES
t' ≥ B
YES
NO
# BYTES STORED = t' "ROTATE RIGHT R0 AND STORE RT" B = B - t' RT = RT + t'
Increment R0 field
# BYTES STORED = B "ROTATE RIGHT R0 AND STORE RT" RT = RT + B
# BYTES STORED = B "ROTATE LEFT R0 AND STORE RT" RT = RT + B
# BYTES STORED = s' "ROTATE LEFT R0 AND STORE RT" B = B - s' RT = RT + s'
s = 0 Increment R0 field
DONE

FIGURE 6

an example move ignoring hardware considerations. With extra state (for B and s), some controls and an additional bypass, the algorithm for short moves can be implemented efficiently on the pipelined machine.

Character moves can be initiated by defining a special instruction

OP RS RT RB

RS contains S, RT contains T, and RB contains B.

The semantics are:

> If RB ≤ 9 then execute the (hardwired) algorithm for short moves after setting the required state else NOP.

The sequence of instructions following this would compare R_S and R_T and branch to the appropriate code for the long moves.

5.3 Fixed Point Multiplication

One approach to fixed point multiplication is to use the modified Booth's algorithm [5]. The multiplier Y is divided into 3-bit groups, with adjacent groups sharing a common bit. X is the multiplicand, 0, X, -X, 2X or -2X is added to the partial product in accordance with the table below; (bit 0 is most significant). After each action, the partial accumulation is shifted right two positions with respect to the new partial product.

BIT			OPERATION
Y_{i-1}	Y_i	Y_{i+1}	
0	0	0	+0
0	0	1	+X
0	1	0	+X
0	1	1	+2X
1	0	0	-2X
1	0	1	-X
1	1	0	-X
1	1	1	-0

Special instructions can be defined to implement multiplication on the pipelined machine using this algorithm. Register positions are numbered from left to right, 0 to 31. The SP register is loaded with Y the multiplier. An instruction, MULT R_1, R_2 executes one step of the modified Booth algorithm.

The semantics of MULT R_1, R_2 (where R_2 contains the multiplicand) are: Form PARTIAL PROD from R_2 as indicated in table (with Y_i-1, Y_i, Y_i+1 replaced by SP30, SP31, and C, where C is a flag).

```
SUM ← $R_1$ + PARTIAL PROD        {34 bit sum}
$R_1$ ← 32 high order bits of SUM
C ← SP30
SP(2-31) ← SP(0-29)
SP(0-1) ← 2 low order bits of SUM
```

With a one bit shift at the input of the ALU, the above steps can be done in the AE cycle.

The code for a 32-bit multiply, with R_3 containing the multiplier and R_2 the multiplicand, is:

```
    SP ← R3                         {Multiplier}
    INIT MULT R1                    {Clear R1 and C}
    MULT R1, R2
       .
       .
                  16 times
       .
       .
    MULT R1, R2
```

These 18 instructions execute a 32-bit x 32-bit multiply in 18 cycles. On completion, the low order bits of the result are in the SP register and the high order bits are in R_1.

It should be stressed that for applications where multiply time is important, high speed multiply should be implemented in hardware.

5.4 Floating Point Operations

Several compute intensive engineering/scientific applications contain a large fraction of floating point operations. Obtaining very high performance in such environments requires that floating point arithmetic be executed in special, dedicated hardware. Software simulation on RISC machines, even with special support, is quite inadequate. In this section we will present an approach to achieving very high performance on engineering/scientific applications by using a RISC machine enhanced with pipelined floating point hardware. We will show how vector speeds can be attained without a vector architecture. The approach is an extension of the concepts presented thus far in the paper: architectural support for concurrency, careful branch and interlock handling, and a strong emphasis on compiler optimizations.

To motivate the approach, we will identify an important bottleneck in conventional scalar machines and illustrate how vector architectures can alleviate this bottleneck in special circumstances. This will also highlight a fundamental difference between RISC and vector architectures.

An important bottleneck in conventional machines is a control bottleneck. Each instruction must be decoded in the specified order to ensure that interlocks are placed so that the computation occurs as specified. The overall execution time can be no less than the time to decode all the instructions. This limitation can be clearly illustrated with a simple example. Consider the computation shown below:

```
DO 10 I = 1, 100
A(I) = C * D(I)
10 CONTINUE
```

The primary operation is a floating point multiply and the design of pipelined hardware to execute this rapidly is well understood. The control problem becomes evident when a pipelined multiply unit is part of a scalar processor. The computation could be coded as below:

```
LOOP:  Load
       Mult
       Store
       Compute Index
       Test Index
       Branch Loop
```

The peak rate at which this computation can execute assuming one instruction initiation per cycle is six cycles. Advanced pipeline processors can actually sustain this rate during the execution of the loop. The pipelined multiplier can accept a pair of operands and deliver a result every cycle. The execution rate is therefore one-sixth the peak multiplier rate. Clearly, RISC machines suffer from this basic limitation. In a vector architecture the computation would be coded using instructions that operate on entire vectors. When a single vector instruction is decoded, the hardware takes over and can initiate several operations automatically. The instruction decode bottleneck is thus removed. The previous computation would be coded as:

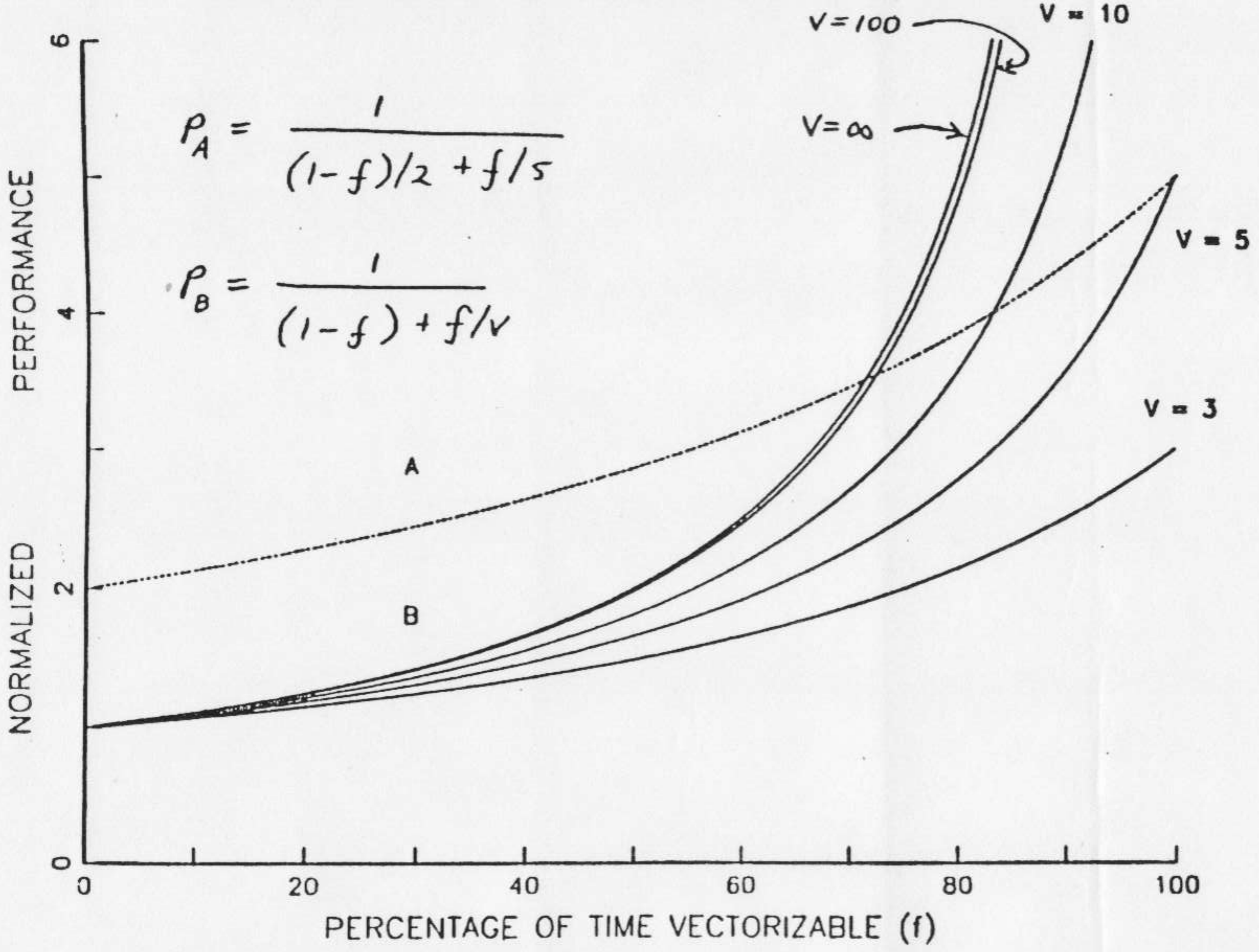
```
Load Vector D
Multiply Vector D by C
Store Vector A
```

Using "chaining", vector processors can execute the computation at a rate of one result per cycle, or 6 times faster than the scalar processor. The real speed-up on the loop is less because the peak speed is attained only after various pipelines are full and is a complex function of start-up time, vector length, and main memory bandwidth.

Assume that the computation under discussion constitutes the main part of an application which takes 100 seconds on a pipelined scalar processor. 90 seconds are spent in the loop and 10 seconds on overhead tasks such as setting up the arrays, input, and output. The application is now "vectorized", a vector unit is added to the scalar machine, and the vector instructions execute in this unit. For the example under consideration, the speed-up, V, is 6 (ignoring start up) and the 90 seconds are reduced to 15. The net speed-up is 100/25 = 4. We will define percent vectorization, f, as 90%, since 90% of the initial workload was affected by vectorization.

Some important points can now be made about vector processors. The reader is referred to Figure 7. The family of curves B represent a simplified

PERFORMANCE OF NON-OVERLAPPED SCALAR/VECTOR



TKMA

view of the performance of vector machines at several values of V (the speed up, while executing vector instructions): If **f** consistently lies in the 80 - 100% range, the design point is fairly evident: a very fast vector unit, a high bandwidth memory system, and an adequate scalar processor. The sophistication of the hardware is directly dependent on cost considerations. Thus, moving from V=5, to V=10 may require doubling the number of arithmetic units and memory bandwidth. At 80-100% vectorization, this added hardware can be well utilized; at 40-60% it will idle most of the time.

Percent vectorization, though critical to performance, is difficult to pin down. It is a very complex function of the application, algorithm, compiler sophistication, and architecture. Let f_1 be the fraction of parallelism in an application. Only a fraction f_2 of this may be vectorizable depending on the algorithms chosen. Furthermore, a compiler may only detect a fraction f_3 of this even assuming an infinitely robust architecture. Finally, a fraction $(1 - f_4)$ may be lost due to the architectures inability to support all forms of vectorization. If each **f** is 90%, only 60% of the parallelism may be exploitable through vectorization. For some very large, structured engineering/scientific problems each **f** may be arbitrarily close to 1 making vector processing an extremely attractive solution. New inventions in algorithms and problem reformulation can also cause percent vectorization to go up. However, except for very special computations, vectorization in the 80% to 100% range over a long period of time is the exception rather than the rule and sustained performance is often 1 to 2 orders of magnitude lower than peak performance. In our opinion, vector architectures are an elegant approach to high performance on some computations. Employing a vector approach more generally increases the complexity of algorithm design, compiling, the architecture, and the hardware and much of the elegance is lost.

One approach to obtaining high performance without vector instructions is to dispatch multiple instructions to the execution units every cycle. This corresponds to a more general though also more limited form of concurrency than vector instructions. By allowing out-of-sequence execution, special handling of branch instructions, and providing pipelined execution hardware, very high performance can be obtained on unstructured code. We call such machines superscalar processors. (Unstructured code typically does not vectorize because of numerous data dependencies and branches.) Since superscalars exploit concurrency even on unstructured computations the performance can be expected to be higher than a conventional scalar machine. In Figure 7, the performance is shown to be a factor of 2 better than a conventional scalar processor (see curve A). Once the instruction issue rate is improved and high performance floating point hardware is added, the processor executes structured computations (the kind that normally get vectorized) at very high speeds. In fact, the performance on structured code is better than the performance on unstructured code. The performance of the machine increases with the percentage of structured code as shown in curve A, Figure 7. Scalar machines exhibiting such behavior have been built. An example is the IBM 360/91.

Curve A crosses curve B, V = 10 at approximately 80% vectorization. If f lies in the range of 40-60%, clearly the superscalar approach has merit.

To explain the overall superscalar approach, we will start with the machine organization shown in Figure 8. On the surface, this is a fairly common block diagram. There are several units: branch, fixed point, floating point, etc. Each unit has a queue to hold incoming instructions, a decoder, registers, and pipelined execution hardware. The goal is to fetch and dispatch N instructions every cycle, one to each unit. If this is attempted in a straightforward manner for an existing architecture (ex. IBM System 370), the dispatcher quickly becomes the bottleneck. It must detect and set interlocks between several instructions simultaneously. In the superscalar machine, this bottleneck is alleviated by moving some of the work from run time to compile time in a fairly straightforward manner.

Some approaches (example MIPS [3]) move all interlock handling to compile time. It is the compiler's responsibility to schedule instructions in such a manner that when a particular instruction is seen by the hardware, it is guaranteed that all operands are available. This is quite constraining. It is often possible to get significant improvements in performance by designing execution units that compute an answer very rapidly in 90% of the cases and once in a while take longer. Such designs cannot be used in the MIPS approach. In environments with caches and asynchronous memory references it is not possible to predict access times and using the worst case is not meaningful. Interlocks must exist for these

MACHINE ORGANIZATION

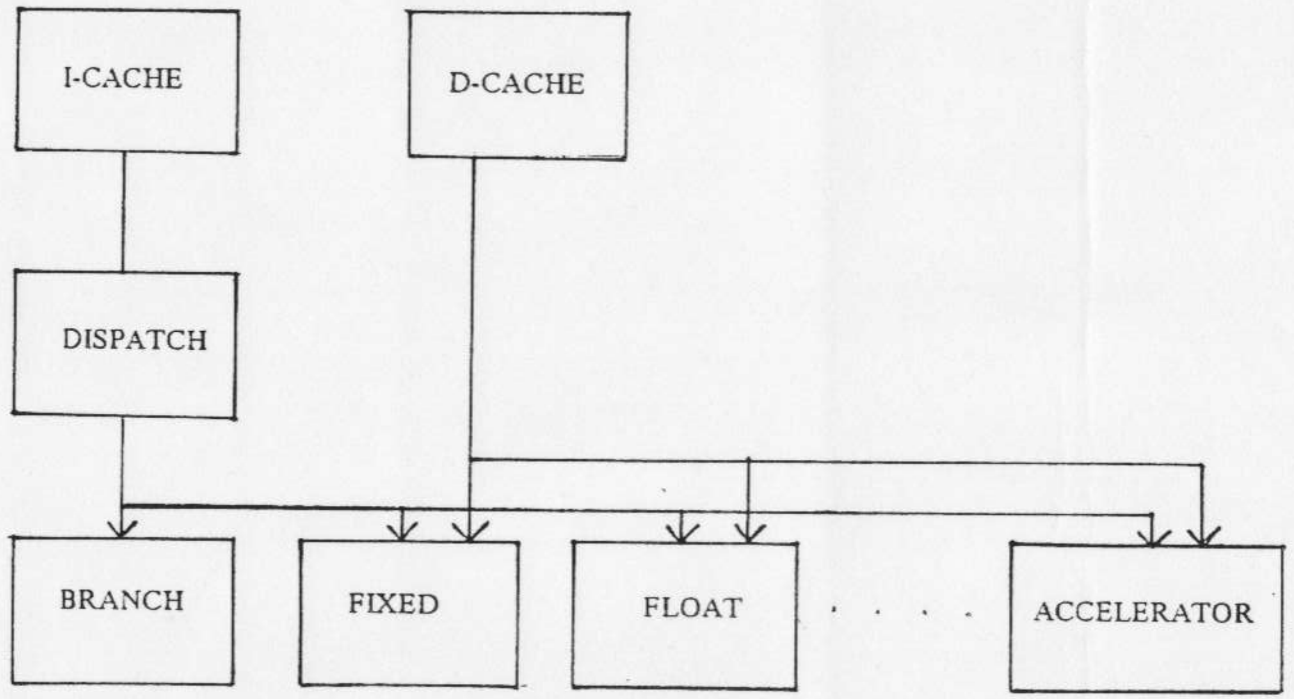


FIGURE 8

reasons. In the superscalar approach, interunit interlocks are set at compile time and tested at run time. Within each unit, decoding and interlock handling is done in the usual manner. This simple innovation allows the dispatcher to issue instructions without being concerned about interlocks. In effect the original single instruction stream is broken into multiple streams at run time. These streams execute asynchronously with hand-shaking on an as needed basis. Two forms of concurrency are thus exploited. The decoders in the functional units are active simultaneously and within each unit, instructions execute in a pipelined manner.

The superscalar uses a reduced instruction set approach augmented with full fixed and floating point arithmetic and provides architectural support for concurrency.

In contrast to simple implementations of reduced instruction set architectures, the superscalar has a lot of concurrency and instructions are processed in a pipeline fashion. The main extension to the 801 architecture lies in the fact that this concurrency is exposed to the compiler in several ways. Instructions are separated into well defined "architectural classes". Each class has its own set of registers. Instructions in one class do not refer directly to registers of another class. Special instructions are provided to move information from one class to another. Architectural support is provided for compile time interlock setting. The opcode assignments are carefully made so that it is trivial for the dispatcher to recognize the architectural class of an instruction. These features which provide explicit architectural support for multiple execution units are key to obtaining very high scalar performance and a lean cycle.

In addition to the branch optimizations discussed in section 3, techniques such as loop unrolling, loop jamming, and loop splitting [6] can be used to further reduce the number of branches executed. An example of jamming is given below.

<u>Before Jamming</u>

```
      DO 10 I = I, N
      S1
10    CONTINUE
      DO 20 I = 1, N
      S2
20    CONTINUE
```

<u>After Jamming</u>

```
      DO 10 I = 1, N
      S1
      S2
10    CONTINUE
```

Let S1 (J) denote the execution of S1 when the loop control variable I has a value J. The execution sequence in the original program fragment is

S1(1), S1(2), ... S1(N), S2(1), S2(2), ... S2(N).

After jamming, the sequence would be

S1(1), S2(1), S1(2), S2(2), ... S1(N), S2(N).

This optimization is legal if and only if, for all values J of the loop control variable I, the inputs of S2(J) do not depend on S1(J + 1) ... S1(N) and the outputs of S2(J) do not change the inputs of S1(J + 1), ... , S1(N).

A very general form of loop unrolling is shown in Figure 9. Though the amount of static code is large, the main loop contains a single branch. If the loop bounds, N1 and N2, are known at compile time, much less code need be generated.

Before unrolling

```
     DO 10 I = N1, N2
     S1
10 CONTINUE
```

Code for M-Way Unrolling

```
I = N1
IF N2 < N1 GO TO L0
P = N2 - N1 + 1
J = P MOD M
K = (P-J)/M
IF J=0 AND K=0 GO TO L0
IF J=1 GO TO L1
IF J=2 GO TO L2
     :
     :
     :
IF J=M-1 GO TO LM-1
L0:  S1
     I=I+1
     GO TO EXIT
L1:  S1
     I=I+1
     IF K=0 GO TO EXIT
     GO TO MAIN
     :
     :
     :
LM-1:S1
     I = I+1
     :                       M-1 times
     :
     :
     S1
     I=I+1
     IF K=0 GO TO EXIT
     GO TO MAIN
MAIN:S1
     I=I+1
     :                       M times
     S1
     I=I+1
     IF I < N2 GO TO MAIN
EXIT:
```

FIGURE 9

An example of using loop splitting is shown in Figure 10. The aim is to remove the branch inside the main loop by handling 4 cases separately. The four cases are:

(1) J = N1. The corresponding execution sequence is

S1(N1), S3(N1), S1(N1 + 1), S2(N1 + 1), S3(N1 + 1), ...,S1(N2), S2(N2), S3(N2)

(2) N1 < J < N2. The corresponding execution sequence is

S1(N1), S2(N1), S3(N1), ..., S1(J - 1), S2(J - 1), S3(J - 1), S1(J), S3(J), S1(J + 1), S2(J + 1), S3(J + 1), ..., S1(N2), S2(N2), S3(N2)

(3) J = N2. The corresponding execution sequence is

S1(N1), S2(N1), S3(N1), ... , S1(N2 - 1), S2(N2 - 1), S3(N2 - 1), S1(N2), S3(N2)

(4) J > N2. The corresponding execution sequence is

S1(N1), S2(N1), S3(N1), ... , S1(N2), S2(N2), S3(N3)

The main loop after splitting is DO 10 I = L1, L2 and it does not contain a branch.

Before splitting

```
      DO 10 I = N1, N2
      S1
      IF I = J GO TO 20
      S2
20    S3
10    CONTINUE
1000
```

After splitting

```
      L1 = N1
      L2 = J-1
      IF J = N1 THEN GO TO 11
      IF J > N2 THEN L2 = N2
101   DO 10 I = L1, L2
      S1
      S2
      S3
10    CONTINUE
      IF L2 = N2 GO TO 1000
11    L2 = L2 + 1
      S1
      S3
      If L2 = N2 GO TO 1000
      L1 = L2 + 1
      L2 = N2
      GO TO 101

1000
```

FIGURE 10

The superscalar approach was extensively evaluated through paper designs and simulation at the register transfer level (including the processor and memory system). The results were extremely high performance on unstructured computations (approximately one cycle per instruction including floating point instructions) and vector speeds on structured computations (one floating point operation per cycle; extension to 2 is fairly straightforward). This is achieved with a lean cycle. In pipelined machines the data flow can be partitioned (within limits) into very lean stages. The controls are more difficult to handle. These are not a bottleneck in the superscalar machine because of the approach to interlocks and partitioning of functions.

The superscalar approach can be summarized as follows. The architecture largely preserves the basic Single Instruction Stream Single Data Stream (SISD) model. As a result, standard optimizations and locality (instruction buffers, data caching, register usage) can be exploited. Fundamental bottlenecks to instruction dispatching are removed. As a result, the machine is flooded with instructions (the number active is a function of the amount of buffering provided). The instructions then execute based on the availability of operands and execution resources. One future challenge is to determine whether this approach can be significantly extended to obtain higher degrees of fine grain parallelism. Multiple dispatchers with super efficient hardware interlock mechanisms is one possibility.

6. Summary

Very generally, the performance of a pipelined scalar processor that decodes a single instruction every cycle is given by

$1 + \Sigma\ (f_i * p_i)$ cycles per instruction

where f_i is the frequency with which certain pipeline disruptions (such as branches, dependencies, cache misses) occur and p_i is the corresponding penalty per disruption. This paper has presented compiler, architecture, and machine organization approaches that reduce the delays caused by the disruptions.

Sections 2 and 3 described a series of techniques to execute simple instructions very rapidly. The results are summarized below where SM and PM denote the simple and pipelined RISC machines and C/I denotes cycles per instruction.

Processor	Cache Latency	Functions/ Cycle(F/C)	Comments	Performance	
				C/I	(C/I)*(F/C)
SM	1	2	No scheduling	1.38	2.76
SM	1	2	Scheduling	1.1	2.2
PM	1	1	No cache bypass, No scheduling	1.77	1.77
PM	1	1	No scheduling	1.52	1.52
PM	1	1	Scheduling	1.23	1.23
PM	1	1	Relative branches	1.1	1.1

By designing the architecture with the machine organization and compiler in mind, infinite cache performance close to one cycle per instruction can be achieved with modest amounts of hardware and a lean cycle (8-10 levels of logic). Whenever a new feature is considered (to reduce cycles per instruction), the impact on cycle time must be analyzed. The mechanism for relative branches reduced the total cycles per instruction from 1.23 cycles per instruction to 1.11 cycles per instruction. Branch resolution and late select in the decode cycle on the branch could cause the cycle time to increase. If the increase is 10%, (one more level of logic in a 10 stage path), the mechanism provides no advantage.

As the processing speed of a pipelined machine is increased, the effects of finite caches become very significant. An analysis of delays and approaches to reducing them were presented in section 4. Some relief can be obtained by proper architecture design.

Efficient execution of long operations was discussed in Section 5. We presented a reduced instruction set approach to character string moves which achieves the moves at the maximum speed possible on the given data paths (two bytes per cycle). Fixed point multiply can be executed reasonably fast (18 cycles) with proper architecture support. For very high speed execution, special hardware must be used. High speed execution of floating point arithmetic is not possible using single cycle instructions. We presented an approach that uses a RISC machine augmented with pipelined floating point hardware. Using an extension of the 801 philosophy, this approach utilizes concurrency to obtain very high performance on unstructured computations and vector speeds on structured computations without a vector architecture.

7. Acknowledgements

We are grateful to George Radin and Marty Hopkins for numerous discussions on RISC architectures. Rudy Rechtschaffen developed the basic performance analysis technique that is used in this paper. Doug DeGroot, Greg Grohoski, and Dan Prener made significant contributions to the development and evaluation of the superscalar approach.

We would like to thank Arvind, Dan Prener, and Harold Stone for their comments on the paper.

8. References

1. Patterson, D. A., "Reduced Instruction Set Computers", CACM, Vol 28, Number 1, January 1985.

2. Radin, G., "The 801 Minicomputer", IBM Journal of Research and Development, Vol. 27, No. 3, May, 1983.

3. Henessey, J., et al, "Hardware/Software Tradeoffs for Increased Performance", Symposium on Architectural Support for Programming Languages and Operating Systems, March 1-3, 1982, Palo Alto, CA.

4. Tick, E. M., "Design and Analysis of a Memory Hierarchy for a Very High Performance Multiprocessor Configuration", M.S. Thesis, MIT, January, 1982.

5. Wasser, S. and Flynn, M. J., "Introduction to Arithmetic for Digital Systems Designers", Holt, Reinhart, and Winston, 1982, pp. 131-135.

6. Allen, F. E. and Cocke, J., "A Catalogue of Optimizing Transformations", Design and Optimization of Computers, Rustin, R. (Ed), Prentice Hall, Englewood Cliffs, N. J., 1972, 1-30.