THE MICROCODE LEVEL TIMESLICING
PROCESSOR ARCHITECTURE

by

DANIEL CURTIS MCCRACKIN, B.Eng., M.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Doctor of Philosophy

McMaster University

(c) Copyright by Daniel Curtis McCrackin, August 1988
Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilm cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

THE MICROCODE LEVEL TIMESLICING
PROCESSOR ARCHITECTURE
DOCTOR OF PHILOSOPHY (1988)  McMaster University
(Electrical Engineering)  Hamilton, Ontario

TITLE: The Microcode Level Timeslicing Processor Architecture

AUTHOR: Daniel Curtis McCrackin, B.Eng., M.Eng. (McMaster University)

SUPERVISOR: Dr. Barna Szabados, Department of Electrical and Computer Engineering

NUMBER OF PAGES: xvi, 172
The von Neumann computer model, upon which most of our modern computers are based, is lacking in both parallelism and support for multitasking. One convenient measure of processor-parallelism is the degree to which a processor utilizes its available memory bandwidth for useful work; its Memory Bandwidth Efficiency (MBE). A processor which exhibits an MBE of 100% is operating as fast as possible for its given memory speed.

Conventional processor accelerators like prefetching and pipelining increase parallelism but suffer from the jump problem, in which a taken jump may cause incorrect prefetching. These accelerators are not well adapted to multitasking, although enhancements like selectable register files may be used to reduce context switching overhead. Pipelined multistreaming achieves a high degree of parallelism, avoids the jump problem and supports efficient context switching, but its performance is load dependent and it is awkward to implement. Furthermore, none of these architectures support efficient process resource sharing.

Microcode Level Timeslicing (MLT) is a multistream processor architecture that achieves very high processor MBE, has no-overhead context switching and provides support for resource sharing. Within the processor, process state information is replicated N-fold. Prefetching occurs horizontally across streams, allowing the jump problem to be
circumvented. Context switching occurs at microinstruction boundaries, giving no overhead for up to \( N \) streams. The fetching and executing mechanisms are controlled by a Stream Control Unit (SCU), which contains task status information for each stream. Efficient process control and resource sharing operations are readily supported in the SCU.

The hardware and software design of a prototype processor demonstrating the MLT principle for up to 16 streams is presented. Process control and synchronization operations are implemented at the microcode level. Two high-level language benchmarks, the Sieve of Eratosthenes and the Dhrystone, are used to evaluate the prototype's performance.
ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Barna Szabados, whose encouragement, open-mindedness and insight made this work possible. I also thank my committee members, Dr. Skip Poehlman and Dr. David Capson, for their guidance and help.

I would especially like to thank my beloved wife, Maureen. Without her encouragement, this work could not have been undertaken; without her love, it could not have been endured; without her help, it could not have been completed. I also thank her for preparing many of the diagrams in the text, and for her proofreading of this document. For her help in wiring the prototype's memory board, in debugging the system and in taking measurements, again, thanks. I am truly blessed to have so dear, so supportive and so loving a wife.

I thank my parents, Larry and Betty, for their love, prayers and support. They have helped me far more than they know. To all of my friends and family (you too, Doreen!), who have been very understanding and supportive, thanks.

The support of the National Engineering and Science Research Council is also gratefully acknowledged.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1 - INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Memory Bandwidth Efficiency and Processor Performance</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Multitasking Overhead</td>
<td>5</td>
</tr>
<tr>
<td>1.3.1 Context Switching Overhead</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2 Resource Sharing Overhead</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Scope of This Work</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 2 - RELATED TECHNIQUES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Related Single-Stream Techniques</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Prefetching</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Pipelining</td>
<td>13</td>
</tr>
<tr>
<td>2.2.3 Prefetching with Branch Prediction</td>
<td>15</td>
</tr>
<tr>
<td>2.2.4 Selectable Register Files</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Related Multistream Techniques:</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Pipelined Multistreaming</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Summary</td>
<td>20</td>
</tr>
</tbody>
</table>
CHAPTER 3 - THE MICROCODE LEVEL TIMESLICING PRINCIPLE

3.1 Introduction

3.2 The MLT Principle:
   3.2.1 Operating Restrictions
   3.2.2 Support of Process Synchronization

3.3 Summary and Applications

CHAPTER 4 - PROTOTYPE ARCHITECTURE

4.1 Introduction

4.2 Programmer's Model
   4.2.1 Memory Management and Organization
   4.2.2 Instruction Format

4.3 Architectural Overview
   4.3.1 Overall Processor Organization
   4.3.2 Execution Unit Data Paths
   4.3.3 Bus Interface and Instruction Unit Data Paths
   4.3.4 Stream Control Unit Status Flags

4.4 Control and Operation
   4.4.1 EU / BIU / SCU Interconnections
   4.4.2 EU Microsequencer
   4.4.3 BIU Control

4.5 Microcode Word

4.6 Summary
CHAPTER 5 - PROTOTYPE STREAM CONTROL UNIT

5.1 Introduction 63
5.2 SCU Priority Rotation System Design 63
5.3 Overall SCU Structure 66
5.4 Status Element Design 68
5.5 SCU Control 71
5.6 Overall Processor Operation 73

CHAPTER 6 - PROTOTYPE SOFTWARE 77

6.1 Introduction 77
6.2 Prototype Instruction Set 77
   6.2.1 Design Considerations 77
   6.2.2 Basic Instruction Set 80
   6.2.3 Task Control Instructions 83
   6.2.4 Semaphore Instructions 86
6.3 High-Level Language Compiler 88
   6.3.1 The MPL Language Compiler 88
   6.3.2 Code Generation Phase Optimizations 92
      6.3.2.1 Argument-to-Register and Temporary-to-Register Binding 92
      6.3.2.2 Peephole Optimization 96
6.4 Summary 96
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Simple von Neumann Processor</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>A Processor with Prefetching</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of a Sequential Processor and an Autonomous EU / BIU Processor</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Simplified Pipelined Processor</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Pipelined Multistream Processor</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Microcode Level Timesliced Processor</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>User Visible Registers</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory Management Registers</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Physical Address Generation</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Instruction Encoding Formats</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Overall Prototype System Organization</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Execution Unit Data Paths</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Bus Interface and Instruction Unit Data Paths</td>
<td>42</td>
</tr>
<tr>
<td>4.8</td>
<td>Memory and I/O Access Timing</td>
<td>44</td>
</tr>
<tr>
<td>4.9</td>
<td>SCU Status Flags</td>
<td>46</td>
</tr>
<tr>
<td>4.10</td>
<td>EU / BIU / SCU Interconnections</td>
<td>48</td>
</tr>
<tr>
<td>4.11</td>
<td>Execution Unit Microsequencer</td>
<td>51</td>
</tr>
<tr>
<td>4.12</td>
<td>Bus Interface and Instruction Unit Control</td>
<td>55</td>
</tr>
<tr>
<td>4.13</td>
<td>Prototype Microcode Word</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Status Element Stream Available and Prefetch Request Priority</td>
<td>65</td>
</tr>
</tbody>
</table>
LIST OF TABLES

4.1 Processor Microcode Fields .............. 58

6.1 Basic Instruction Set (Partial Listing) ... 81

6.2 Task Control Instructions ............... 84

8.1 Instruction Set and Benchmark Statistics .... 126

8.2 Comparative Processor Performance ....... 129

A1 Instruction Set .......................... 137

A2 Representative Microcode .............. 147

A4a Sieve of Eratosthenes Results .......... 167

A4b Dhrystone Results ....................... 168

A4c Sieve Execution–Skew Test ........... 169
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSP</td>
<td>Auxiliary and Common Scratch Pad</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic and Logical Unit</td>
</tr>
<tr>
<td>ASP</td>
<td>Auxiliary Scratch Pad</td>
</tr>
<tr>
<td>BIU</td>
<td>Bus Interface and Instruction Unit</td>
</tr>
<tr>
<td>BTB</td>
<td>Branch Target Buffer</td>
</tr>
<tr>
<td>CASTREAM</td>
<td>Current Access Stream</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CS</td>
<td>Code Segment</td>
</tr>
<tr>
<td>CSP</td>
<td>Common Scratch Pad</td>
</tr>
<tr>
<td>CSTREAM</td>
<td>Current STREAM number</td>
</tr>
<tr>
<td>DETIDLE</td>
<td>DETerminate if in IDLE loop</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DS</td>
<td>Data Segment</td>
</tr>
<tr>
<td>EOI</td>
<td>End Of Instruction</td>
</tr>
<tr>
<td>EU</td>
<td>Execution Unit</td>
</tr>
<tr>
<td>EUWAIT</td>
<td>Execution Unit Wait control</td>
</tr>
<tr>
<td>FCOMPL</td>
<td>Fetch COMPLETE</td>
</tr>
<tr>
<td>FCW</td>
<td>Flag and Control Word</td>
</tr>
<tr>
<td>FSP</td>
<td>Flag Scratch Pad</td>
</tr>
<tr>
<td>FSTREAM</td>
<td>Fetch STREAM request(from SCU)</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>IDBUS</td>
<td>Internal Data BUS</td>
</tr>
<tr>
<td>IPS</td>
<td>Instructions Per Second</td>
</tr>
<tr>
<td>IR</td>
<td>Instruction Register</td>
</tr>
<tr>
<td>IRWE</td>
<td>Instruction Register Write Enable</td>
</tr>
<tr>
<td>KROM</td>
<td>Constant ROM</td>
</tr>
<tr>
<td>MAR</td>
<td>Memory Address Register</td>
</tr>
<tr>
<td>MBE</td>
<td>Memory Bandwidth Efficiency</td>
</tr>
<tr>
<td>MDR</td>
<td>Memory Data Register</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction stream and Multiple Data stream</td>
</tr>
<tr>
<td>MIPS</td>
<td>Millions of Instructions Per Seconds</td>
</tr>
<tr>
<td>MLT</td>
<td>Microcode Level Timeslicing</td>
</tr>
<tr>
<td>MOMUX</td>
<td>Microcode Output Multiplexor</td>
</tr>
<tr>
<td>MPL</td>
<td>Modular Programming Language</td>
</tr>
<tr>
<td>MSI</td>
<td>Medium Scale Integration</td>
</tr>
<tr>
<td>MSR</td>
<td>Memory Segment Register</td>
</tr>
<tr>
<td>MUSE</td>
<td>MultiStream Executive</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>NSTREAM</td>
<td>Next STREAM number</td>
</tr>
<tr>
<td>NXT</td>
<td>NeXT stream execute request (from SCU)</td>
</tr>
<tr>
<td>PAL</td>
<td>Programmable Array Logic</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PCALL</td>
<td>Parallel CALL</td>
</tr>
<tr>
<td>PCSP</td>
<td>Program Counter Scratch Pad</td>
</tr>
<tr>
<td>PRET</td>
<td>Parallel RETurn</td>
</tr>
</tbody>
</table>
PRQ  Prefetch ReQuest
PT   Page Table
RISC Reduced Instruction Set Computer
ROM  Read Only Memory
RSP  Register Scratch Pad
RUNNP RUN with No Prefetch mode
SAV  Stream Available
SCON System CONTroller
SCU  Stream Control Unit
SEL  Status Element
SISD Single Instruction stream and Single Data stream
SP   Stack Pointer
WCS Writeable Control Store
XA   Extended Address
XS   Extended Segment
CHAPTER 1

INTRODUCTION

1.1 Introduction

In the computer model proposed by John von Neumann, a program is seen as a single stream of instructions which are for the most part executed in serial order[1]. The instructions and data are stored in the same central memory, and the instructions act upon only a single stream of operand data. Thus one may classify a simple von Neumann processor as a Single Instruction stream and Single Data stream machine (SISD)[2]. This processor model has the advantage of simplicity; it is both simple to build and easy to program. It is because of this model's simplicity that most modern computers are still based upon it.

One problem with this simple processor model is that it lacks parallelism. To execute an instruction, the processor might fetch the instruction, decode it, retrieve operands, perform calculations and store results. Since these operations are sequential in a simple machine, the processor's resources will be under-utilized.

Processor accelerators are architectural changes that improve the efficiency of processor operation by introducing parallelism and optimizing time-consuming operations. This thesis examines the principle and operation of a novel processor accelerator, Microcode Level Timeslicing (MLT), which improves both processor memory utilization and processor multitasking performance. To this end, this
chapter details the nature of these two problems.

1.2 Memory Bandwidth Efficiency and Processor Performance

Figure 1.1 shows the block diagram of a simple von Neumann processor\[1\]. The computer is divided into four main sections. The memory unit holds the data that the computer operates upon and the instructions that it executes. The control unit fetches and decodes instructions from memory and directs the operation of the other units. The data operator performs whatever data manipulations are required. Collectively, the control unit and the data operator are referred to as the central processing unit or CPU. The input and output unit is similar to the memory unit in that data is read from it and written to it, but the data comes from input devices and goes to output devices, linking the computer to the outside world.

Instruction execution involves several steps or cycles. A typical sequence might involve fetching the instruction from memory, decoding the instruction, fetching the instruction's operands, operating on these data and writing the results.

Clearly, the rate at which instructions are executed is proportional to the rate at which data moves across the CPU-memory boundary. Thus if two processors have the same instruction set and the same maximum CPU-memory transfer rate, then the processor which makes better use of the available memory bandwidth will be the faster of the two.

One may define a processor's Memory Bandwidth Efficiency (MBE)
Figure 1.1: Simple von Neumann Processor
to be given by:

\[
\text{Memory Bandwidth Efficiency} = \frac{\text{Average CPU - Memory Transfer Rate}}{\text{Maximum CPU - Memory Transfer Rate}}
\]

Clearly the average CPU-Memory transfer rate must include only the transfer of information that is required for the execution of the program. Unused information, like instructions that are not executed, must not be included. Given this, the MBE may be used as a measure of how well a given CPU utilizes its memory resource.

In general, it is desirable for a computer to exhibit high MBE. One should note that some processor implementations are intrinsically more memory intensive than others; memory-to-memory architectures, in which several memory references are required per instruction, may yield an artificially high MBE. A similar processor with internal registers could be expected to execute faster for a given maximum memory bandwidth, even though its MBE may be lower.

It is interesting that many processor accelerators function by increasing the processor MBE. Prefetching, for example, uses times when the CPU - memory link would have been idle to fetch the next few instructions. Likewise, the use of several functional units for overlapping the execution of instructions may be viewed as reducing the average time to execute an instruction, thus decreasing CPU - memory link idle time and increasing MBE.

Conventional processors typically exhibit poor memory bandwidth efficiencies. For example, in the DEC VAX-11/780 superminicomputer,
about 20% of all memory transfers are for incorrectly prefetched instructions[3]. The recently introduced Intel 80386 utilizes only about 73-79% of its available memory bandwidth, disregarding the effect of incorrect fetching[4].

1.3 Multitasking Overhead

There are two main sources of overhead in multitasking systems: context switching and resource sharing. The former results from the sharing of the processor’s time; the latter results from the sharing of its data structures and I/O devices.

1.3.1 Context Switching Overhead

Context switching involves saving all user-visible CPU registers and flags, selecting a new process to execute, and re-loading that process' registers and flags. This switch requires a number of memory accesses, but these memory accesses do not actually contribute to the execution of any process. The proportion of CPU time consumed by this overhead increases as the context switch rate is increased, until all the CPU's time is consumed by overhead and the machine is said to be thrashing.
1.3.2 Resource Sharing Overhead

Unpredictable operation will result if multiple processes are allowed uncontrolled access to common resources[5]. For example, if two processes try to update a common data structure at the same time, the data structure could well be corrupted. A mutual exclusion system is needed so that only one process may access this common resource at one time.

One way of implementing a mutual exclusion mechanism is to associate a special control variable, called a semaphore, with each shareable resource[5]. The value of the semaphore indicates whether the resource is free or in use. To acquire access to a resource, a task must wait until that resource's semaphore is clear and then set the semaphore, preventing others from accessing the resource. When finished with the resource, the task clears the semaphore again.

This mechanism is difficult to implement with conventional computer architectures. The problem lies in what is done if a process needs a resource but finds that it is already in use. A tight waiting loop or "busy wait" would waste an unacceptable amount of processor time. A better mechanism would be to switch to another task and prevent the waiting task from being invoked until the resource is ready. Unfortunately, this mechanism is fairly expensive in terms of programming complexity and overhead time, as detailed status information must be maintained for each task.
1.4 Scope of This Work

In Chapter 2 we examine several conventional processor accelerators that improve processor MBE and that reduce multitasking overhead. Chapter 3 presents the principle of Microcode Level Timeslicing (MLT), which is a novel Multiple Instruction stream and Multiple Data stream (MIMD) architecture that achieves processor MBEs approaching 100%, context switching with no overhead and very efficient resource sharing. Chapters 4 and 5 detail the hardware design of a prototype minicomputer which utilizes this principle. Chapter 6 examines the design of this prototype's instruction set and of a high-level language compiler for benchmarking. The processor performance tests are discussed in Chapter 7, with the results presented in Chapter 8. Chapter 9 presents our conclusions and recommendations for further work to be carried out.
CHAPTER 2

RELATED TECHNIQUES

2.1 Introduction

The Microcode Level Timeslicing (MLT) architecture has three main advantages: it is capable of very high processor MBE, it supports limited multitasking with no context switching overhead and it supports low overhead process synchronization. This chapter presents a survey of single-stream and multistream architectures that are related to the MLT architecture and share some of these desirable properties.

2.2 Related Single-Stream Techniques

2.2.1 Prefetching

Prefetching is a processor accelerator that improves memory bandwidth efficiency by allowing fetching and execution to occur concurrently. This increased parallelism reduces memory idle time, increasing the processor’s MBE and its speed.

Figure 2.1 shows an implementation of a processor with prefetching, architecturally similar to the IBM 370 model 165 [6] and the Intel 8086 processors [7]. The processor may be divided into two autonomous but synchronized sections: a Bus Interface and Instruction Unit (BIU) and an Execution Unit (EU).
Figure 2.1: A Processor with Prefetching
The EU contains all the hardware required to perform each instruction once the BIU has fetched the instruction and its operands. The BIU handles all memory accesses, and uses times when the memory is not otherwise needed to fetch the next instructions. In a typical EU/BIU processor, the EU will contain the processor ALU and general purpose registers while the BIU will contain the Program Counter (PC) and a small prefetch queue.

Some insight into how this arrangement increases the processor MBE may be gained by comparing the operation of a simple sequential machine with this EU/BIU machine. Figure 2.2 compares the operation of these two processors executing three instructions that take 2, 4 and 2 cycles, respectively. We assume that memory takes 3 cycles to access, as in the Zilog Z8000 [8] and the Motorola 68020 [9]. The prefetch queue depth is taken to be 1, for simplicity. Note that each machine performs exactly the same amount of work, but the simple sequential processor takes 7 cycles more. Alternatively, we observe that the MBE of the simple sequential machine is 9/19 or 47%, while the MBE of the EU/BIU machine is 9/11 or 82%.

If the queue depth were increased, we would observe that the MBE of the EU/BIU machine approaches 100% as long as the average non-memory access time taken to execute each instruction is less than the time taken to fetch the next instruction. This restriction insures that the queue will tend to empty, allowing the effects of longer instructions to be absorbed by the queue.

Clearly, prefetching does serve to increase the MBE and hence the speed of a processor. That the improvement may be had with
Figure 2.2: Comparison of a Simple Sequential Processor and an Autonomous EU/BIU Processor
relatively simple hardware is one reason why this processor accelerator is used in almost all modern von Neumann computers.

The problem with the prefetching processor accelerator is that it is not always possible to predict what instruction should be prefetched next \([10,11,12]\). One cannot predict the outcome of a conditional jump when the jump depends on the results of the preceding instructions. Even unconditional jumps present problems; by the time an unconditional jump has been decoded and recognized, the next instruction fetch may be under way. If the processor continues to fetch, there is a chance that the wrong path will be fetched, requiring the incorrectly fetched instructions to be discarded. This event carries a double performance penalty as not only is there a delay in fetching the correct instruction, but also several memory accesses are wasted. If instruction decoding is fast enough, one may simply suppress prefetching until the jump destination is known, a strategy which avoids wasting memory accesses.

It is because of this jump problem that prefetching cannot yield 100% processor MBE. Since 15-25% of all executed instructions are taken jumps \([13]\), one would expect that a simple prefetching system would achieve at best 80-90% MBE. In the VAX-11/780, about 20% of the memory bandwidth is lost because of taken jumps \([3]\).
2.2.2 Pipelining

In pipelining, the fetching and execution of two or more instructions is overlapped by breaking the fetch and execute process up into equal time phases. For example, executing an instruction might involve fetching, calculation, and accessing memory. This arrangement is shown in Figure 2.3. Each instruction is executed in assembly-line fashion as it "travels" through the pipeline. Since each stage takes equal time, the execution of three instructions may be interleaved, in this simplified example. This interleave has an effect similar to that of prefetching N instructions ahead.

Pipelined processors may be much faster than non-pipelined prefetching implementations, since several instructions are actually in execution at once. However, the maximum speed of pipelined systems is not realizable because of the same jump problem that affects non-pipelined prefetching [11,12]. A jump instruction may require flushing the pipeline or holding up previous stages of the pipeline, reducing throughput and increasing processor complexity.

The simplest way of avoiding the flushing or holding up of pipeline stages is the delayed jump, as demonstrated in the IBM 801[14] and the Berkely RISC I processors[15]. The definition of the jump instruction is changed so that the jump does not take effect until all of the partially executed instructions in the pipeline have been completed. In the IBM 801 and the RISC I, this means that the one instruction immediately following a jump is executed whether the jump is
Figure 2.3: Simplified Pipelined Processor
taken or not. This simplifies the processor hardware, as there is no need to clear or retard pipeline stages.

A simple way of handling delayed jumps is to follow each jump instruction with no-operation (NOP) instructions. This allows code to be generated in a conventional fashion but is inefficient; on both the IBM 801 and the RISC I, 1 instruction would be wasted per jump. A better approach is to re-arrange the machine code to eliminate NOPs, but complete NOP elimination is difficult to achieve. The compiler for the IBM 801 successfully eliminated about 60% of the NOPs in this way[12]. In unoptimized RISC I code, NOPs occur about 18% of the time, statically; approximately 75% of these NOPs were eliminated by code optimization[15]. One would expect worse NOP elimination statistics for machines with more delay slots (i.e., longer pipelines).

Since NOPs do not contribute to the execution of the program, their execution must be considered to waste a memory access. For processors with longer pipelines than that of the IBM 801 and the RISC I, the presence of NOPs will represent a major loss of processor MBE. Although the delayed jump mechanism reduces the jump problem, it is still impossible for a pipelined processor to achieve 100% MBE.

2.2.3 Prefetching with Branch Prediction

Branch Prediction with Branch Target Buffers is one of the more interesting methods of reducing the conditional jump problem encountered in prefetching [10,11]. A Branch Target Buffer (BTB) is a cache memory that holds the addresses and destinations of recently taken jump
instructions along with information for predicting the outcome of each jump the next time it is encountered. This cache, together with a statistical prediction mechanism, allows the prefetching system to recognize an impending branch and alter the prefetching path accordingly.

Lee and Smith suggest that a 512 element 4-way set-associative BTB and a simple prediction mechanism could yield a prediction accuracy of about 80% [10]. Assuming that jumps occur about 20% of the time dynamically, this should reduce the prefetch miss rate to about 4%. However, context switching presents a problem, since it is necessary to flush or correct the BTB when the processor address space mapping is changed. Lee and Smith note that this factor may significantly affect performance.

2.2.4 Selectable Register Files

A common method for improving context switching performance is to incorporate selectable alternate register files into the processor. Context switching may then be accomplished by altering a select register, rather than saving the register file to memory and re-loading it. This speeds up the context switching process, although some overhead remains.

Selectable register files have been incorporated into a number of computers. Intel's 8048 and 8051 family of microcontrollers have two and four selectable register banks, respectively, for implementing low latency interrupts [16]. The small scale of these particular
implementations limits their usefulness for multitasking, however.

The Texas Instruments 9900 family uses a very interesting implementation of selectable register files that does support multitasking [17]. The mechanism is simple: within the processor is a Workspace Pointer register WP, which points to 16 contiguous 16-bit words in memory. These 16 words are called the Workspace Registers, WRO-WR15, and are used by the processor as general purpose registers. Context switching is easily accomplished by saving the current PC and WP and loading the new pair. The problem with the TMS9900 is that it is actually a memory-to-memory architecture; a simple "register-to-register" add instruction requires four memory accesses. The advantage gained with a moveable register file is largely outweighed by this implicit inefficiency.

Clearly, selectable register files do reduce the overhead involved in context switching. One must note that only context switching is improved; the processor MBE and process synchronization overhead are not affected.

2.3 Related Multistream Techniques

Several multistream processors appear in the literature [18] to [22]. We will briefly examine one technique, Pipelined Multistreaming, which shares several of the characteristics of Microcode Level Timeslicing [20,21].
2.3.1 Pipelined Multistreaming

Pipelined multistreaming, in which an N-stage circular pipeline contains N offset processes, is capable of achieving both 100% MBE and limited multitasking with no context switching overhead [20,21]. Figure 2.4 shows a diagram of this arrangement. In each cycle, the state information for each process (flags, program counter, registers) moves one stage around the circular pipeline. The processor is designed so that memory accesses for each task occur at only one section of the ring and will occur each time a process rotates through this section, ensuring complete memory utilization.

This architecture is conceptually simple, and achieves very high memory utilization. There are, however, three major problems. First, implementation is difficult, as all the state information of each stream must travel around the pipeline; a very wide internal data path is needed. For this reason it is impractical to implement more than a few registers in the pipeline, making register-to-register architectures inappropriate. If a slower memory-to-memory structure is used then memory accesses are needed for all operands, reducing the execution speed. Second, the performance of this architecture is very load dependent; all processes must be executing for 100% MBE. For example, if only 2 out of 4 processes are in use, the MBE drops to a mere 50%. The third drawback arises from this, as there is no obvious, efficient way to implement process synchronization mechanisms; if a process is blocked, Nth of the memory bandwidth is wasted.
Figure 2.4: Pipelined Multistream Processor
2.4 Summary

Prefetching, pipelining, selectable register files and pipelined multistreaming each act to improve some aspect of processor performance. Prefetching improves processor MBE by allowing concurrent instruction fetching and execution. Pipelining overlaps the execution of several instructions. Selectable register files reduce context switching overhead. Pipelined multistreaming achieves both high MBE and low context switching overhead.

Each of these accelerators has serious disadvantages. Prefetching and pipelining cannot achieve 100% MBE because of the conditional jump problem. Selectable register files improve context switching but have no effect on processor MBE. Pipelined multistreaming achieves efficient multitasking and high MBE, but is too complex for practical implementation and yields good performance only under full process load. None of these techniques support efficient process synchronization.

Clearly a "perfect" accelerator would produce 100% MBE, exhibit zero overhead context switching, be simple to construct and would support process synchronization. It will be seen that the Microcode Level Timeslicing (MLT) architecture comes very close to realizing this ideal.
CHAPTER 3

THE MICROCODE LEVEL TIMESLICING PRINCIPLE

3.1 Introduction

In Chapter 2, several processor architectures related to Microcode Level Timeslicing (MLT) were presented. This chapter presents the MLT principle itself and discusses its applications.

3.2 The MLT Principle:

Recall that in an autonomous EU/BIU processor, the Bus Interface and Instruction Unit (BIU) handles all memory accesses and prefetches instructions into an instruction queue. The Execution Unit (EU) accepts these prefetched instructions and performs all the data operations required to carry them out. A simple MLT processor uses a modification of this arrangement, as shown in Figure 3.1.

The first major modification involves replicating the state information of the EU and the BIU N-fold, so that their contexts may be individually selected. This is similar to having selectable register files. In the EU, the registers, flags and micro-program counter are replicated; in the BIU, the program counter and the prefetch registers are replicated. In hardware, this replication corresponds to increasing the size of the register file, replacing the PC with a small scratch pad, and so forth. This state replication is considerably cheaper than
Figure 3.1: Microcode Level Timesliced Processor
constructing the wide pipeline required by pipelined multistreaming; wide pipelines require many global interconnections whereas scratchpads use mainly nearest-neighbor interconnections.

In itself, this N-fold replication of state information may be used to eliminate context switching overhead. Multitasking for at most N tasks requires only the appropriate manipulation of the select lines. Since context switching occurs at microinstruction boundaries, there is no context switching overhead.

The next change involves limiting the prefetching depth for each stream to (at most) 1 instruction. We now eliminate the jump problem by adding the restriction that a stream's next instruction may not be fetched until enough time has elapsed to determine if the previous instruction is a taken jump, and if so, its destination. This allows enough time to choose the correct fetching path.

If only one stream were executing, this restriction on prefetching would have a deleterious effect upon the processor's MBE, as several cycles would elapse from the time that the next instruction could be prefetched to the time that it actually is prefetched. During this interval the memory would be idle.

The last requirement for the operation of the MLT principle is the addition of a Stream Control Unit (SCU), which coordinates the EU and BIU context selection. The purpose of the SCU is simple: it selects the next stream for which the BIU will prefetch and selects the next stream for which the EU will execute an instruction. The SCU ensures that only streams with empty prefetch registers will be prefetched and that only streams with full prefetch registers will be executed, and
enforces the prefetch delay required to circumvent the jump problem. Implicitly, the SCU is also responsible for the correct distribution of processing time to each of the tasks.

In order for the SCU to carry out this selection function, it is necessary that it contain some status information for each stream. The simplest SCU implementation might have 1 bit per stream to indicate the status of the stream's prefetch register. If fewer than the maximum N streams are to be executed, additional status bits are required to handle the presence of halted streams.

The operation of the principle is straightforward. Streams compete for the BIU resource (for prefetching) and the EU resource (for execution). If only one stream is running, the prefetch depth is 1, although the enforced prefetch delay reduces the beneficial effect of this prefetching. If two streams are running, the prefetch depth is two, and the prefetch delay effect is reduced; during one stream's delay, the other may be prefetched. As more tasks are added, the prefetch depth grows to a maximum depth of N instructions ahead for N streams. Since the prefetch depth is great, processor MBE may be very high (100%), but because prefetching is done across independent streams, no prefetch misses occur. It is this "horizontal prefetching" that allows MLT to achieve very a high processor MBE.

A unique property of this processor is now apparent. In a conventional processor executing multiple processes, performance generally decreases, or at best remains constant, as more processes are added. With the MLT accelerator, performance actually increases until 100% MBE is reached. Since there is no context switching overhead,
context switching rates do not affect processor performance. Naturally, more than N processes would require that less efficient context switching techniques be used also, degrading performance somewhat.

It is important to note that although the performance of an MLT machine is dependent on the number of executing processes, the relationship is not linear. In pipelined multistreaming, removing a process disables 1/Nth of the processor; in Microcode Level Timeslicing, one level of prefetch is lost. Clearly the latter will have much less effect on processor MBE.

3.2.1 Operating Restrictions

It is necessary to define the conditions under which Microcode Level Timeslicing improves processor MBE. The performance of an MLT processor executing Ns streams will be similar to that of a processor with error-free prefetching into an Ns level prefetch queue. For the purpose of this discussion we define Tf to be the time required to prefetch the next instruction word; this is usually a constant. We also define Te to be the amount of time required to execute an instruction; this is a random variable. Te must not include time spent for operand access, since the memory cannot be idle at these times.

The three parameters, Ns, Tf and Te are the most important in determining the efficacy of the MLT scheme. First consider the processor's operation if Te is a constant:

Case 1: Tf < Te. The prefetching system is faster than the execution system. The prefetching queue will tend to fill, causing
the memory to idle part of the time. Since the queue tends to stay full, increased queue depth increases EU utilization, as the EU need not wait for the next instruction to be fetched.

Case 2: $T_f > T_e$. As the execution system is faster than the prefetching system, the queue will tend to empty, causing the EU to idle part of the time. Increasing the queue depth will reduce memory idle time, increasing processor MBE. In effect we trade EU utilization for better MBE.

Since the main tenet of MLT is that Memory Bandwidth Efficiency should be maximized, we must have $T_f > T_e$. Since $T_e$ is actually a random variable, this requires that $T_f > \text{mean}(T_e)$. Given this, the MBE performance of MLT for a given number of running streams depends on the instruction mix characteristics. If two instruction mixes have the same $\text{mean}(T_e)$ then the one with the wider $T_e$ distribution will require deeper prefetching for the MBE to reach 100%.

3.2.2 Support of Process Synchronization

Extra stream status information in the SCU may be used to implement sophisticated mechanisms for process synchronization. Bits in the SCU could inhibit the selection of a stream until an awaited event occurs. Very efficient semaphore operations would then allow a process to "sleep" until the desired resource is free. These operations may be implemented as single machine instructions, reducing system software overhead.

Note that the task trigger event may also originate from outside
of the CPU. This external trigger would allow efficient interrupt-like mechanisms to be implemented. Very low latencies are possible, as context switching takes no processor time.

3.3 Summary and Applications

The Microcode Level Timeslicing architecture has many of the advantages of other processor architectures with few of their problems. It can achieve very high Memory Bandwidth Efficiencies through deep prefetching, but does not suffer from the jump problem. Like systems with selectable register files, MLT supports very efficient context switching. In these two respects the MLT architecture is similar to pipelined multistreaming. Unlike pipelined multistreaming, however, MLT's operation is not very load dependent. Furthermore, MLT does not require the wide internal pipeline needed by pipelined multistreaming. Lastly, MLT supports resource sharing, which is essential to multitasking systems.

MLT's main limitation is that the maximum number of tasks is limited by the number of hardware contexts, N. Reasonable values for N likely range from 4 to 32 steams, with 8 being an economical compromise. Of course, more tasks than N may be supported with the aid of other context switching techniques. With clever design, the penalty for using these auxiliary techniques could be minimized.

Despite its limitation to N tasks, MLT may be the architecture of choice for low-cost multitasking systems. Lower context switching rates are usually acceptable in this application, reducing the penalty
of having auxiliary context switching mechanisms. If fast response to some events is required, timing-critical tasks could be left resident in the processor. The overall system cost/ performance ratio should be lower than for conventional processors, as the higher MBE allows slower memory to be used for similar processor performance. In low-end systems, the need for separate direct memory access (DMA) devices could be eliminated by having one or two streams handle DMA as required. Since the processor MBE approaches 100%, one may argue that performance in a cacheless system with stream DMA will be the same as that of a cacheless system with external DMA.

The MLT architecture is perhaps best suited to real-time control applications. With an appropriately designed SCU, MLT can easily achieve event response times on the order of one instruction time. At an average of 6 clocks per instruction, with a clock speed of 10MHz, one might expect latencies on the order of 2 uS. A comparable Motorola 68020 based system, however, may take on the order of 60 clocks to respond to an interrupt[23]. Including another 30 clocks for partial context switching, one would expect a latency of about 10 uS, 5 times slower than with MLT. Even better latencies are possible if streams are dedicated to servicing specific events.

Clearly, although the MLT architecture is limited by the number of hardware contexts, N, it is still very versatile. It may be the method of choice for low-cost high-performance multitasking systems, and has great potential in real-time control applications.
CHAPTER 4

PROTOTYPE ARCHITECTURE

4.1 Introduction

The purpose for constructing a Microcode Level Timeslicing prototype is primarily to illustrate that the principle is indeed workable. The prototype must illustrate the three main features of the MLT architecture: high memory bandwidth utilization, no context switching overhead for limited multitasking, and low resource sharing overhead.

This prototype design is a compromise between complexity of function and ease of construction with MSI logic. It is desirable that a high-level language be supported to facilitate testing of the processor.

For these considerations, the prototype is designed to have a 16-bit word length (with 32-bit operations supported in microcode). A maximum of 16 streams may run concurrently in a MLT fashion on this machine. Each stream has 32 general purpose registers. A simplified load-store instruction set is implemented, but some complex operations, like multiplication and multiple moves, are included to improve high-level language performance. The simplicity of the instruction set is essential, as a 16-bit word size provides only a small number of instruction encodings.

This chapter presents an overview of the prototype computer's
architecture and design. The details of the Stream Control Unit will be discussed in Chapter 5, while Chapter 6 will examine the prototype's instruction set and its implementation.

4.2 Programmers Model

The stream-local user-visible registers are shown in Figure 4.1. There are 32 general purpose registers, R0 to R31, which may be used singly for 16-bit operations or in pairs for 32-bit operations. Thus R0 is a 16-bit register while R1R0 and R2R1 are 32-bit. Note that multiple word data are stored least significant word first in both the register file and in the memory, following the "little-endian" convention.

The program counter (PC) has the usual function of pointing to the next instruction to be fetched, and is accessed primarily by jump instructions. Two other special registers, Extended Address (XA) and Extended Segment (XS), allow access to more memory than the 64K bytes that can be addressed with 16 bits. They are also used for accessing the memory management registers.

The Flag and Control Word (FCW), in addition to the usual sign, carry, overflow and zero flags, contains the identification number of the current stream, which is used for maintaining stream specific data structures. For example, if the instruction "move r3,ds" (move the contents of register 3 into the current data segment register) is executed, this stream number is used to determine which of the 16 data segment registers must be modified.
General Purpose Registers:

15

\[
\begin{array}{c}
R0 \\
R1 \\
\vdots \\
R30 \\
R31 (SP) \\
\end{array}
\]

Special Purpose Registers:

15

\[
\begin{array}{c}
\text{Extended Segment (XS)} \\
\text{Extended Address (XA)} \\
\text{Program Counter (PC)} \\
\end{array}
\]

Flag and Control Word (FCW):

15

\[
\begin{array}{cccccccccc}
\times & \times & \times & \times & \text{stream \#} & \times & \times & \times & S & CY & OV & Z \\
\end{array}
\]

- S = sign flag
- CY = carry flag
- OV = signed overflow
- Z = zero flag
- x = undefined (always 1)

Figure 4.1: User Visible Registers
4.2.1 Memory Management and Organization

In addition to the stream-local registers, the streams share two other register files: the segment table and the page table. Figure 4.2 shows the organization of these two tables. The segment table contains 16 code segment registers, CS0 to CS15, and 16 data segment registers, DS0 to DS15. All stream 0 data references refer to the segment named by DS0, all stream 0 code references use CS0, and so on. There is one segment register pair for each stream.

The calculation of the physical address from the stream number, access type and logical address is shown in Figure 4.3. When memory is accessed, the chosen segment register's contents are concatenated onto the 4 most significant bits of the logical address, forming an offset into the page table. The selected page table element then gives the physical address of the selected page, while the 12 least significant bits of the logical address give the offset within the 4K word page.

It should be noted that the logical address itself is 16 bits long and addresses 64K words or 128K bytes. All CPU-memory transfers are 1 word long; byte operations are not well supported in hardware. Byte move operations are implemented only at the microcode level, giving the illusion that each data segment consists of 64K bytes, with words having even addresses. This word-even addressing scheme (reminiscent of that used by the DEC PDP-11 [24]) is used only when accessing the data segment. All accesses to the code segment use word addresses, allowing a total of 128K bytes of code and 64K bytes of
Segment Table:

- **Code Segment**
  - CS0
  - CS1
  - ...
  - CS14
  - CS15

- **Data Segment**
  - DS0
  - DS1
  - ...
  - DS14
  - DS15

Page Table:

- **P0**
- **P1**
- ...
- **P2046**
- **P2047**

*Figure 4.2: Memory Management Registers*
Figure 4.3: Physical Address Generation
data per stream.

4.2.2 Instruction Format

There are two simple instruction encoding formats, as shown in Figure 4.4. There are 60 instruction encodings having two 5-bit select fields and 128 encodings having one 5-bit select field. Note that a select field may specify a register number or a short 5-bit unsigned constant, and field 1 may additionally specify a 5-bit condition code for conditional instructions. The short constants allow considerable code savings, as most constants in high-level language programs fall within this small (0..31) range[13]. If an instruction requires constants outside of this range, extra words are appended to the opcode, least significant word first.

4.3 Architectural Overview

4.3.1 Overall Processor Organization

Figure 4.5 shows the overall prototype system organization. There are three sections: the System Controller (SCON), the Memory and the Central Processing Unit (CPU).

The System Controller is an 8-bit microprocessor system (Zilog Z80 based) which has three functions. The SCON is responsible for initializing the CPU's hardware and for loading the microcode into its writeable control store. Facilities for single stepping the main
(I) Double Select Field: (60 combinations)

\[
\begin{array}{c|c|c}
15 & 10 & 9 \\
6 \text{ bit instr. code} & \text{field 2} & \text{field 1}
\end{array}
\]

eg., move Rs,Rd -- move the contents of Rs to Rd

(II) Single Select Field: (128 combinations)

\[
\begin{array}{c|c|c}
15 & 12 & 11 \\
1 & 1 & 1 \\
7 \text{ bit instr. code} & \text{field 1}
\end{array}
\]

eg., srl Rd -- logical shift Rd right

Notes:
Field 1 may contain: (1) a register #
(2) a 5 bit constant, 0..31
or (3) a 5 bit condition code

Field 2 may contain: (1) a register #
or (2) a 5 bit constant, 0..31

If immediate operands are required by an instruction,
(i.e., move #n,Rd), the operands are placed in the
locations following the opcode, least significant
word first.

Figure 4.4: Instruction Encoding Formats
Figure 4.5: Overall Prototype System Organization
processor's clock and for monitoring its internal buses permit easier system debugging. Lastly, the prototype's I/O is handled by the SCON via an 8-bit communication channel.

The prototype's memory consists of 1M byte of static RAM organized as 512K 16-bit words. The use of static memory reduces the complexity of the hardware since refresh circuitry and processor-memory synchronization circuitry are not needed.

The prototype CPU is itself organized into three sections: a Bus Interface and Instruction Unit (BIU), an Execution Unit (EU) and a Stream Control Unit (SCU).

As its name would imply, the Bus Interface and Instruction Unit handles all memory access required by the EU and all instruction prefetches required by the SCU. The EU performs all the actual data processing required by each instruction. The SCU coordinates the activity of the EU and the BIU by choosing the next stream for which the BIU will fetch and the next stream for which the EU will execute. Thus the EU and the BIU perform all memory and data operations, while the SCU controls the selection of the various streams for the EU and the BIU.

4.3.2 Execution Unit Data Paths

Figure 4.6 shows the organization of the prototype's Execution Unit (EU) data paths.

Central to the Execution Unit are the Arithmetic and Logical Unit (ALU) and the shifter, which perform all simple data operations. The ALU obtains its input from two select circuits, the Source A
Figure 4.6: Execution Unit Data Paths
multiplexor and the Source B multiplexor, AMUX and BMUX. The high byte and the low byte of each multiplexor may be separately selected, simplifying the read/modify/write implementation of byte store operations. The ALU's output is passed through a shifter for shift, rotate and byte swap operations to the Internal Data Bus (IDBUS). The operation of the ALU and shifter may affect the flags in the Flag Scratch Pad (FSP), which is a 16 * 4-bit register file with extra gating. The FSP and the adjacent current stream number buffer form the processor's Flag and Control Word (FCW).

Source A of the ALU may come from a 32 * 16-bit Constant ROM (KROM) or from the Register Scratch Pad (RSP). The RSP, which has 16 banks of 32 16-bit registers each, contains the user-visible general purpose registers, one bank for each stream. A register select circuit allows register[X+offset] to be chosen, where X is either 0, the contents of the P register, or the contents of a select field in the instruction register (IR).

Source B of the ALU may come from the Constant ROM or from the Auxiliary and Common Scratch Pad (ACSP). The ACSP is subdivided into two sections: a 16 element Common Scratch Pad (CSP) which is shared by all 16 streams, and a 16 bank * 16 element Auxiliary Scratch Pad (ASP), of which each stream has a private bank. The CSP portion is used for interstream communication, while the ASP portion is used for holding temporary values during instruction execution.

Both the Register and the Auxiliary and Common Scratch pads are constructed so that one of their registers may be read at the start of a clock cycle and written at the end of it. Thus two address code
instructions like "add r0,r1" (add the contents of register 0 into register 1) take two clock cycles to execute. In this example, the first clock is used to move r0 to an auxiliary register a0, and the second is used to add a0 to r1, storing the result in r1.

The P register is a special purpose loadable up/down counter that is shared by all streams. It has three uses: as an index into the register file for multiple register move instructions, as an index into the segment table in the BIU memory manager and as a microcode loop index. The highest bit of the P register (bit 7) may be tested to facilitate the latter use.

4.3.3 Bus Interface and Instruction Unit Data Paths

The main data paths of the Bus Interface and Instruction Unit are shown in Figure 4.7. Note that the BIU really consists of two separate sections: an Address Generation Section and a Data Handling Section.

The BIU Data Handling Section contains a Memory Data Register (MDR) which latches data being written to main memory, an input buffer for gating memory data onto the Internal Data Bus (IDBUS), and a dual-port prefetch register array. At the end of each prefetch operation, data is written into the prefetch register corresponding to the current access stream number (CASTREAM). When the next stream (NSTREAM) begins execution, its prefetched word is copied into the Instruction Register (IR) and is decoded to give the instruction's start address in the EU microcode store.
Figure 4.7: Bus Interface and Instruction Unit Data Paths
The BIU Address Handling Section generates the logical addresses and performs address translation for all memory and input/output accesses. During prefetching, the instruction address is taken from the appropriate Program Counter Scratch Pad (PCSP) element. A controlled incrementer allows a PCSP element to be incremented during fetching or simply loaded during jumps. Note that during fetching the CASTREAM element of the PCSP must be used; during jumps the Currently executing Stream (CSTREAM) element is used. For all other memory or I/O accesses, the address is generated by the ALU in the EU and transferred over the Internal Data Bus (IDBUS).

The Memory Address Register (MAR) holds the logical address for each of these references. Concurrent with the logical address generation, the segment number for the access is obtained from the segment table (see section 4.2.1 above) and is stored in the Memory Segment Register (MSR). The final logical-to-physical address translation is performed using the Page Table (PT).

There are two main types of external BIU accesses: memory and input/output. Memory accesses address up to 256M words of external storage. Input/Output accesses use only the unmapped 12 bits of the logical address, giving a maximum of 4096 I/O registers. These external accesses are controlled with four signals: mreq (memory request) enables memory devices, iorq (I/O request) enables I/O devices, rd (read) gates data from the external devices onto the memory data bus and wr (write) signifies that data is available on the memory data bus. The timing of these three-cycle accesses is shown in Figure 4.8.
Memory or I/O Read Cycle:

<table>
<thead>
<tr>
<th>Bus Cycle</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IORQ, MREQ</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 23-12:</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 11-00:</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory or I/O Write Cycle:

<table>
<thead>
<tr>
<th>Bus Cycle</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IORQ, MREQ</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 23-12:</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 11-00:</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data out</td>
<td>valid</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8: Memory and I/O Access Timing
4.3.4 Stream Control Unit Status Flags

It is useful at this point to examine the nature of the status information within the Stream Control Unit. Each stream has one set of status flags, shown in Figure 4.9. Flags $S_2$ and $S_1$ specify what mode the stream is in: HALT, WAIT, RUN or RUNNP (run with no prefetch). Flag $S_0$ indicates if the prefetch register for the stream is full. In addition, there is one shared flag, CREATE, that indicates if the SCU is in process creation mode.

If a stream is in the RUN mode, it may be selected for prefetching if $S_0$ is clear (i.e. the prefetch register is empty) and may be selected for execution if $S_0$ is set. This is the normal mode for a running process.

In the RUNNP (run with no prefetch) mode, a stream will never be selected for prefetching but may be selected for execution at any time. This mode is useful for implementing instructions in which it is necessary to perform calculations before a conditional jump (e.g. DJNZ "decrement and jump if not zero"). Switching a stream into RUNNP prevents the next instruction for that stream from being fetched, avoiding a possible prefetch miss.

The WAIT mode may be used to implement a simple process blocking/unblocking scheme. In the WAIT mode, a process will neither be prefetched nor executed. A stream in the WAIT mode will be switched to the RUNNP mode under two conditions: if another process executes a SIGNAL microoperation or if an external hardware AWAKE signal is
Stream Status

<table>
<thead>
<tr>
<th>$S_2$</th>
<th>$S_1$</th>
<th>$S_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shared

<table>
<thead>
<tr>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Stream State

<table>
<thead>
<tr>
<th>$S_2$</th>
<th>$S_1$</th>
<th>Stream State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>HALT ($S_0$ cleared)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>WAIT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RUN</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RUNNP (no prefetch)</td>
</tr>
</tbody>
</table>

Prefetch Status

<table>
<thead>
<tr>
<th>$S_0$</th>
<th>Prefetch Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Prefetch Register Empty</td>
</tr>
<tr>
<td>1</td>
<td>Prefetch Register Full</td>
</tr>
</tbody>
</table>

Creation Mode

<table>
<thead>
<tr>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Creation Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Normal Mode</td>
</tr>
<tr>
<td>1</td>
<td>Process Creation Mode</td>
</tr>
</tbody>
</table>

Figure 4.9: SCU Status Flags
received. If a stream is waiting for a resource it may WAIT. When any other stream releases any resource it will SIGNAL, allowing WAITing streams to re-test their blocking conditions. Thus the synchronization overhead is reduced to a few clocks per stream at each SIGNAL.

A stream in the HALT mode will be neither prefetched nor executed under normal conditions. However, when a new process is to be created, the creating process places the new process' startup parameters (start address, etc.) in the Common Scratch Pad and executes a CREATION micro-operation, setting the CREATE flag in the SCU. When the CREATE flag is set, only HALTTeď streams may be selected for execution. The first HALTTeď stream to execute then picks up the startup parameters, clears the CREATE flag, restoring the system to normal operation, and enters the RUN mode.

It is important to note that except for the effect on processor prefetching behavior, no mode is final until a process initiates a micro-context switch. Thus a process may enter the WAIT mode for a few cycles, and then switch itself back to the RUN mode with no effect on its execution.

4.4 Control and Operation

4.4.1 EU / BIU / SCU Interconnections

Figure 4.10 shows the signals that are used to coordinate the operation of the Execution Unit, the Bus Interface and Instruction Unit and the Stream Control Unit.
Figure 4.10: EU / BIU / SCU Interconnections
The EU/SCU interconnection allows the SCU to specify which stream the EU is to execute next and allows the EU to indicate when the execution of the selected stream actually begins. The Stream Available (SAV) signal indicates that at least one stream is ready for execution, while the NXT signals indicate the number of the highest priority ready stream. The CREATE signal indicates if this stream had been halted and is to be initialized. The Begin Execution (BEGINEX) signal from the EU acknowledges that the specified stream is about to be executed, allowing the SCU to modify that stream's status information accordingly.

A similar handshake exists between the SCU and the BIU for the control of prefetching. The SCU signals a prefetch request (PRQ) and specifies the stream to be fetched for (FSTREAM). One clock cycle before the fetch is completed, the BIU signals Fetch Complete (FCOMPL) and identifies the stream that the current access was for (CASTREAM). This allows the SCU to adjust its status information early enough so that the same stream will not be prefetched twice.

The synchronization of the EU and the BIU is accomplished in a simpler fashion. If the EU tries to initiate an access or access the program counter at the wrong time, the BIU asserts the EUWAIT signal. This signal disables the EU clock, preventing the current microinstruction from completing. After 1 or 2 cycles, when the conflict has ended, the EUWAIT signal is released, allowing the EU to continue. Thus the BIU may operate continuously; it is the EU that is forced into synchrony as required.
4.4.2 EU Microsequencer

The control of the execution unit is accomplished by a specially modified microsequencer, shown in Figure 4.11. The main adaptations for the MLT architecture involve the 16-fold replication of the microprogram counter (micro PC) and the addition of a Next Stream (NSTREAM) register and a Current stream (CSTREAM) register.

Since each stream has its own micro PC, stream to stream context switching involves only changing the contents of the Next Stream (NSTREAM) register, which selects the correct element of the micro PC. The context switch is completed one cycle later, when the CSTREAM register, from which the EU register bank selects are derived, is updated. Thus the new stream's first microinstruction begins execution one cycle after the NSTREAM select changes.

The output of the micro PC is fed into the Microcode Output Multiplexor (MOMUX), which selects the source of the microcode address presented to the 2048 word Writeable Control Store (WCS). In normal operation the MOMUX selects the micro PC. At the start of a new instruction it selects the instruction decoder and when a new stream is created it selects the special process startup address OBC_{16}.

The WCS generates the 72-bit microcode word. 50 bits are routed to the Microinstruction Register and from there are distributed to the EU data paths and the SCU. "Early" signals, controlling microcode jumps, context switching and BIU operations, are taken directly from the WCS, as they are required one cycle before the microinstruction actually
Figure 4.11: Execution Unit Microsequencer
Microcode jumps are controlled by the early signals CC_Source and C_Code. CC_Source selects, via the Condition Code Multiplexor (CCMUX), the source of the condition code for the present instruction. Either the C_Code field or instruction register field 1 may be chosen. The resulting condition code is evaluated by the Condition Logic. If true, the Jump Multiplexor (JMUX) selects the early Jump Address field as the value to be clocked into the micro PC, effecting the jump. If false, the micro PC receives the incremented MOMUX output, which points to the next microinstruction.

The use of "early" microinstruction signals has two interesting consequences. First, if a microinstruction contains a jump, that jump appears to occur just after the microinstruction is executed. This contrasts with the "delayed jump" mechanism used in many other microsequencers, and is necessary for single execution cycle instructions. Second, the condition flags used to control the jump were produced two microinstructions before the jump micro-instruction. Thus if one wishes to test a register and jump if it contains zero, three microinstructions are required: the test, a "no op" and a conditional jump.

The context switching process itself is accomplished with the early End of Instruction signal (EOI), which is also produced one cycle before the microinstruction containing it executes. If EOI is active and there is a stream available, then the BEGINEX signal is returned to the SCU and the SCU Next Stream number (NXT) is written into the Next Stream Register (NSTREAM). Note that if no stream is available (SAV
is inactive) no context switch occurs. \textbf{BEGINEX} is not generated and the current stream continues to execute.

Complications arise as there are actually three types of context switch that must occur under different circumstances. Clearly, one type occurs at the end of an instruction when a new instruction for a ready stream is to be executed. There must also be a method of starting a new stream at \texttt{OBC}_{16} for process creation. Thirdly, certain instructions (like semaphore test-and-set with waiting) must be able to switch contexts and later resume where they left off. Each of these three circumstances corresponds to one setting of the Microcode Output Multiplexor (MOMUX).

\textbf{BEGINEX} triggers the Microcode Output Multiplexor (MOMUX) select logic to select the correct address source in the next clock cycle. If the SCU CREATE flag was set, then MOMUX input 2 is selected, causing a jump to location \texttt{OBC}_{16} in the Writeable Control Store for new process initialization. If CREATE is not set, then the DETIDLE signal, which is simply the (unused) high bit of the micro PC, is used to determine if a new instruction is to start. In the cycle after \textbf{BEGINEX}, if DETIDLE is set, the instruction decoder (input 1) is selected and Instruction Register Write Enable (IRWE) is generated, causing a new instruction to be written into the instruction register from the prefetch register file. If DETIDLE is clear, then the NSTREAM's microcode sequence simply resumes where it left off.

The key to this mechanism is that all instructions end with a microinstruction containing "eoi / jump idle", where "idle" is location
OFFF<sub>16</sub>. This causes the instruction decoder to be selected the next time the stream is executed. At location OFFF<sub>16</sub> in the WCS (which is the same as location OFFF<sub>16</sub> since bit 11 is not connected) is a single microinstruction "eoi / jump idle". If no stream is available, the microsequencer will simply wait in this idle loop until one is ready.

4.4.3 BIU Control

The control of the Bus Interface and Instruction unit is accomplished with a small state machine and extra logic to keep track of stream numbers, as shown in Figure 4.12.

The sequencer's operation is straightforward. The sequencer recognizes access requests whenever it is either idle or in the last cycle of an access. The EU requests memory or I/O accesses via the early BIUOP signals while the SCU uses the PRQ (prefetch request) line to initiate a fetch for FSTREAM. EU access requests always have priority over fetch requests since instructions requiring operands cannot proceed until the operands have been fetched.

Since the EU and the BIU operate autonomously, it is possible for the EU to issue a request at a time when the BIU cannot honour it. In this situation, the BIU asserts the EUWAIT signal (not shown), which prevents the present EU microinstruction from completing. Thus the EU may be retarded for one or two cycles to allow the BIU to catch up.

A special register, the Current Access Stream Register (CASTREAM) keeps track of which stream the current fetch or other access is for. This information is needed to select the correct program
Figure 4.12: Bus Interface and Instruction Unit Control
counter and prefetch register for fetching and to select the correct segment register for each memory access. The CASTREAM register may be loaded either from SCU FSTREAM (Fetch Stream) signals for prefetching or from the microsequencer NSTREAM (Next Stream) signals for EU requested accesses.

An additional multiplexor allows the program counter (PC) to be selected from either CASTREAM for prefetching or from CSTREAM (current stream) to permit updating the current stream's PC for jumping.

4.5 Microcode Word

The layout of the machine microcode word is shown in Figure 4.13. Table 4.1 details the meaning of each of these microcode fields.

4.6 Summary

This chapter has presented a detailed overview of the prototype computer's architecture and design. Chapter 5 presents the design details of the Stream Control Unit, which coordinates the operation of the Execution Unit and Bus Interface and Instruction Unit.
<table>
<thead>
<tr>
<th>Bit:</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0</td>
<td>Flagop</td>
<td>Ccode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ccsrc</td>
<td>Eoi</td>
<td>Ccf</td>
<td>Maddr[11..8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maddr[7..0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 1</td>
<td>Sin</td>
<td>Shift</td>
<td>Cin</td>
<td>Src</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Aluop</td>
<td>Const</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pmode</td>
<td>Acwr</td>
<td>Acsel</td>
<td>Acad</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rwr</td>
<td>Ras</td>
<td>Roffset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M3</td>
<td>M2</td>
<td>M1</td>
<td>M0</td>
<td>Merge</td>
<td>Idsrc</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pcwr</td>
<td>Biuop</td>
<td>Segtwr</td>
<td>Streamop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.13: Prototype Microcode Word
### Flagop: Flag Scratch Pad Operation

<table>
<thead>
<tr>
<th>Code</th>
<th>Operation</th>
<th>Mnemonic</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>s, cy, ov, z &lt;= s, cy, ov, z</td>
<td>(default)</td>
<td>no change</td>
</tr>
<tr>
<td>001</td>
<td>s, cy, ov, z &lt;= idbus[3..0]</td>
<td>fcw := ?????</td>
<td>load</td>
</tr>
<tr>
<td>010</td>
<td>s, cy, ov, z &lt;= s', c_s', ov', z'</td>
<td>flags arith</td>
<td>norm arith</td>
</tr>
<tr>
<td>011</td>
<td>s, cy, ov, z &lt;= s', c_s', ov', z'</td>
<td>flags shift</td>
<td>norm shift</td>
</tr>
<tr>
<td>100</td>
<td>s, cy, ov, z &lt;= s', c_a', ov', z'&amp;z'</td>
<td>flags arith_extend</td>
<td>extend z</td>
</tr>
<tr>
<td>101</td>
<td>s, cy, ov, z &lt;= s', c_s', ov', z'&amp;z'</td>
<td>flags shift_extend</td>
<td>extend z</td>
</tr>
<tr>
<td>110</td>
<td>s, cy, ov, z &lt;= s', cy, ov, z'</td>
<td>flags logical</td>
<td>only s, z</td>
</tr>
<tr>
<td>111</td>
<td>s, cy, ov, z &lt;= s', cy, ov, z'</td>
<td>flags logical_extend</td>
<td>extend z</td>
</tr>
</tbody>
</table>

**Special Symbols:**
- $c_a$ = alu carry out
- $c_s$ = shifter output
- $s$ = sign flag
- $s'$ = idbus sign bit
- $z$ = 'zero flag
- $z'$ = idbus zero detect
- ov = overflow flag
- ov' = alu overflow output
- cy = carry flag

### Ccf: Complement Carry Flag

- O' => normal carry flag hold or update
- 1 => complement carry flag

---

Table 4.1a: Microcode Fields -- Flag Operations
### Ccode: Condition Code Select

**Note:** `c = 1` complements the condition

<table>
<thead>
<tr>
<th>Code</th>
<th>Equation</th>
<th>Mnemonic</th>
<th>Complement Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0000</td>
<td>false</td>
<td>(default)</td>
<td></td>
</tr>
<tr>
<td>c0001</td>
<td>cy</td>
<td>cy, ugt (unsigned &gt;)</td>
<td>ncy, ule (unsigned &lt;=)</td>
</tr>
<tr>
<td>c0010</td>
<td>v</td>
<td>ov (overflow)</td>
<td>nov (no overflow)</td>
</tr>
<tr>
<td>c0011</td>
<td>s</td>
<td>minus</td>
<td>plus</td>
</tr>
<tr>
<td>c0100</td>
<td>z</td>
<td>z, eq (equal)</td>
<td>nz, neq (not equal)</td>
</tr>
<tr>
<td>c0101</td>
<td>s xor v</td>
<td>gt (signed &gt;)</td>
<td>lte (signed &lt;=)</td>
</tr>
<tr>
<td>c0110</td>
<td>(s xor v) or z</td>
<td>gte (signed &gt;=)</td>
<td>lt (signed &lt;)</td>
</tr>
<tr>
<td>c0111</td>
<td>c or z</td>
<td>uge (unsigned &gt;=)</td>
<td>ult (unsigned &lt;)</td>
</tr>
<tr>
<td>c1000</td>
<td>p reg. bit 7</td>
<td>pb7</td>
<td>npb7</td>
</tr>
<tr>
<td>c1001</td>
<td>xf1</td>
<td>xfl (external flag 1)</td>
<td>nxf1</td>
</tr>
<tr>
<td>c1001</td>
<td>xf2</td>
<td>xfl (external flag 2)</td>
<td>nxf1</td>
</tr>
<tr>
<td>c1001</td>
<td>xf3</td>
<td>xfl (external flag 3)</td>
<td>nxf1</td>
</tr>
<tr>
<td>c1001</td>
<td>xf4</td>
<td>xfl (external flag 4)</td>
<td>nxf1</td>
</tr>
<tr>
<td>c1001</td>
<td>xf5</td>
<td>xfl (external flag 5)</td>
<td>nxf1</td>
</tr>
<tr>
<td>c1001</td>
<td>xf6</td>
<td>xfl (external flag 6)</td>
<td>nxf1</td>
</tr>
<tr>
<td>c1001</td>
<td>xf7</td>
<td>xfl (external flag 7)</td>
<td>nxf1</td>
</tr>
</tbody>
</table>

### Csrc: Condition Code Source

0 => source is Ccode field  
1 => source is instruction register field 1

### Eol: End of Instruction

0 => no action  
1 => trigger microcode context switch

### Maddr: Microcode Jump Address

Table 4.1b: Microcode Fields -- Condition Codes
**Shift: Shift Operation**

- 00: no shift
- 01: shift right
- 10: shift left
- 11: byte swap

**Clk: ALU Carry Input**

- 00: 0
- 01: 1
- 10: cy flag
- 11: not cy flag

**Aluop: ALU Operation**

- 000: 0
- 001: B - A
- 010: A - B
- 011: A + B
- 100: A xor B
- 101: A or B
- 110: A and B
- 111: -1

**Merge and Src: Byte Merge and ALU Source Select**

<table>
<thead>
<tr>
<th>Merge Src</th>
<th>ALU Source A</th>
<th>ALU Source B</th>
<th>Special Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>high byte</td>
<td>low byte</td>
<td>high byte</td>
</tr>
<tr>
<td>0 0 0:</td>
<td>rsp</td>
<td>rsp</td>
<td>acsp</td>
</tr>
<tr>
<td>0 0 1:</td>
<td>rsp</td>
<td>rsk</td>
<td>acsp</td>
</tr>
<tr>
<td>0 1 0:</td>
<td>krom</td>
<td>krom</td>
<td>acsp</td>
</tr>
<tr>
<td>0 1 1:</td>
<td>krom</td>
<td>krom</td>
<td>acsp</td>
</tr>
<tr>
<td>1 0 0:</td>
<td>krom</td>
<td>rsk</td>
<td>krom</td>
</tr>
<tr>
<td>1 0 1:</td>
<td>krom</td>
<td>rsk</td>
<td>acsp</td>
</tr>
<tr>
<td>1 1 0:</td>
<td>rsk</td>
<td>krom</td>
<td>acsp</td>
</tr>
<tr>
<td>1 1 1:</td>
<td>rsk</td>
<td>krom</td>
<td>acsp</td>
</tr>
</tbody>
</table>

**Ids src: Internal Data Bus Source**

- 000: ALU / Shifter
- 001: Flags
- 010: --
- 011: segment table[p]
- 100: pc
- 101: mdr
- 110: IR field 1
- 111: IR field 2

Table 4.1c: Microcode Fields -- ALU, Shifter and Internal Bus Control
### Const: KROM constant Select
(Note: only "0" is currently available)

<table>
<thead>
<tr>
<th>00000:</th>
<th>0</th>
<th>01000:</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001:</td>
<td>--</td>
<td>01001:</td>
<td>--</td>
</tr>
<tr>
<td>00010:</td>
<td>--</td>
<td>01010:</td>
<td>--</td>
</tr>
<tr>
<td>00011:</td>
<td>--</td>
<td>01011:</td>
<td>--</td>
</tr>
<tr>
<td>00100:</td>
<td>--</td>
<td>01100:</td>
<td>--</td>
</tr>
<tr>
<td>00101:</td>
<td>--</td>
<td>01101:</td>
<td>--</td>
</tr>
<tr>
<td>00110:</td>
<td>--</td>
<td>01110:</td>
<td>--</td>
</tr>
<tr>
<td>00111:</td>
<td>--</td>
<td>01111:</td>
<td>--</td>
</tr>
<tr>
<td>10000:</td>
<td>--</td>
<td>11000:</td>
<td>--</td>
</tr>
<tr>
<td>10001:</td>
<td>--</td>
<td>11001:</td>
<td>--</td>
</tr>
<tr>
<td>10010:</td>
<td>--</td>
<td>11010:</td>
<td>--</td>
</tr>
<tr>
<td>10011:</td>
<td>--</td>
<td>11011:</td>
<td>--</td>
</tr>
<tr>
<td>10100:</td>
<td>--</td>
<td>11100:</td>
<td>--</td>
</tr>
<tr>
<td>10101:</td>
<td>--</td>
<td>11101:</td>
<td>--</td>
</tr>
<tr>
<td>10110:</td>
<td>--</td>
<td>11110:</td>
<td>--</td>
</tr>
<tr>
<td>10111:</td>
<td>--</td>
<td>11111:</td>
<td>--</td>
</tr>
</tbody>
</table>

### Ras: Register Addr Base Select

| 00: | 0 |
| 01: | p |
| 10: | IR field 1 |
| 11: | IR field 2 |

### Acsel: Aux or Common Select

| 0: | Common Scratch Pad |
| 1: | Auxiliary Scratch Pad |

### Rwr: Register Scratch Pad Write

| 0: | write |
| 1: | no write |

### Acrw: Aux/Common Scratch Pad Write

| 0: | write |
| 1: | no write |

### Roffset: Register Select Offset

### Acad: Aux/Common Address

- **Pmode**: P Register Mode
  - 00: decrement
  - 01: increment
  - 10: hold
  - 11: load

Table 4.1d: Microcode Fields -- Register and Constant Control
**Pcwr: Program Counter Write**

- 0: write  
- 1: no write

**Segtwri: Segment Table Write**

- 0: write segment table[p]  
- 1: no write

**Busop: BIU Operation**

- 000: no operation  
- 001: immed opcode fetch  
- 010: data read  
- 011: data write  
- 100: mmu page table read  
- 101: mmu page table write  
- 110: i/o read  
- 111: i/o write

**Streamop: Stream Operation**

- 000: no operation  
- 001: signal sleeping streams  
- 010: create at next eol  
- 011: --  
- 100: enter HALT  
- 101: enter WAIT  
- 110: enter RUN  
- 111: enter RUNNP

**M3 to M0: Event Marker Bits**

- Connect to external test points

Note: * marked commands may cause 1..2 cycle EU wait

---

Table 4.1e: Microcode Fields -- Miscellaneous Control
CHAPTER 5

PROTOTYPE STREAM CONTROL UNIT

5.1 Introduction

In the previous chapter, the architecture of a prototype Microcode Level Timeslicing computer was presented. This chapter deals with the design of the prototype's Stream Control Unit (SCU), which is perhaps the most unusual part of the processor.

The SCU selects which stream is to be fetched for by the BIU and which stream is to be executed by the EU. Status information must be maintained for each stream, as described in section 4.3.4. In the prototype, each stream has a Status Element (SEL) which contains that stream's status bits and the logic for updating them. Thus there are 16 independently operating status elements in this machine.

5.2 SCU Priority Rotation System Design

The major SCU design problem is that of ensuring that each running stream has an equal opportunity to execute. If there are 16 status elements, each with a PRQ (prefetch request) and a SAV (stream available) output, then two 16-to-4 priority encoders are sufficient for implementing a stream selection mechanism. However, the distribution of time would be skewed in favour of the highest priority streams, a situation which would be unacceptable.
A better time distribution may be achieved by periodically rotating the stream priorities, ensuring that each stream has equal time as the highest priority stream. The problem with this simple approach is that the rotatable encoder must respond within 100 ns (for a 5 MHz clock rate) and must be fairly simple. Unfortunately, no compact implementation of a fast rotatable priority encoder exists.

A novel way of solving this encoder problem is to use fixed priority encoders and rotate the location of the data instead. This is done by arranging the SCU status elements in a ring. In each clock cycle the status elements may either operate normally or may shift their status bits one position around the ring. A 4-bit counter counts the number of rotations and 4-bit adders are used to adjust the encoder outputs accordingly. This mechanism achieves the effect of a high-speed rotatable priority encoder with simple hardware.

This mechanism is not in itself sufficient to ensure that equal time is given to all streams; if fewer than 16 streams are executing then the time distribution may be skewed. Consider Figure 5.1, which shows the SAV and PRQ priority arrangement for the ring of SELs. The SAV priority lags the PRQ priority by one position, so that the stream with the highest SAV (stream available) priority has the lowest PRQ (prefetch request) priority. Since the processor is designed so that its execution speed is limited by the memory access time, it is really the PRQ priority distribution that determines the distribution of processing time to the streams.

Consider what happens when only two streams with status bits
Figure 5.1: Status Element Stream Available and Prefetch Request Priority
in adjacent SELs are running. One stream will have the higher prefetch priority for 15/16 of the time; the other will have higher priority only 1/16 of the time. With three adjacent streams, one stream will have higher priority for 14/16 of the time, the other two only for 1/16 each. Thus, the presence of halted SELs in the ring causes asymmetry in the time distribution. A mechanism is needed to compensate for this effect.

The rotate compensation algorithm may be simply stated: "if SEL 14 (which has the highest PRQ priority) holds the status bits of a halted stream, then rotate one position." In the prototype, priority rotation may occur at most every other clock cycle or every 400 ns. Thus this algorithm ensures that there will be a running stream in SEL 14 in at most \(14 \times 2 = 28\) clock cycles (5.6 uS at 5 MHz). If the rotation time quantum is chosen to be much greater than this minimum time, the algorithm will yield a very even priority time distribution. For example, with a time quantum of 1 mS, at most 0.6% of the time quantum is needed for this rotate compensation.

5.3 Overall SCU Structure

Figure 5.2 shows the overall structure of the stream control unit.

Central to the design is the ring of 16 status elements, SEL 0 to SEL 15. Priority encoders, in conjunction with 4-bit adders, are used to generate the SAV (stream available for execution) and PRQ (prefetch request) signals. Demultiplexors distribute the FCOMPL (fetch
Figure 5.2: Overall SCU Structure
complete) signal from the BIU and the BEGINEX (begin execution) signal from the EU to the SELs. One additional demultiplexor drives the CURRENT input of the SEL which contains the currently executing stream, allowing status-altering microinstructions to change only that stream's status.

The 4-bit ROTATE counter tracks the number of priority rotations, modulo 16. This positive offset, ROTATE, is added to the CSTREAM number for CURRENT stream selection and to the CASTREAM number for FCOMPL (fetch complete) stream selection. The priority encoder compensation requires a negative offset which is generated from the complement of the ROTATE count. Since \(-x = \overline{x} + 1\) (2's complement), the SAV compensation adder carry input must be set to 1. Since the PRQ encoder's inputs are already offset one position, its adder carry input is set to zero.

Note that the BEGINEX (beginning execution) demultiplexor requires no, compensating adder since its select input is derived directly from the SAV encoder's output. This allows the EU to send the BEGINEX signal to the NXT stream's SEL with minimum delay and hardware. Note also that if the EU is waiting then the BEGINEX signal is delayed until the EU can continue.

5.4 Status Element Design

A block diagram of a SEL is shown in Figure 5.3. Each SEL is actually a simple state machine and is fabricated from two 74PL16R4
Figure 5.3: Status Element Block Diagram
Programmable Array Logic ICs[25].

Each SEL's status outputs (\(S_3 - S_0\)) connect to the status inputs of the next SEL (\(S_3 - S_0\)). This ring structure permits the shifting of status information from each SEL to the next. RESET causes all status flip-flops to be synchronously cleared for system initialization, forcing all streams to HALT. Priority rotation is triggered by the SHIFT input, which overrides all other inputs except RESET. Care must be taken in performing a shift cycle, as the PAL program is not sufficiently complex to handle simultaneous shifting and status updating.

Three demultiplexor driven signals allow the updating of status information in specific SELs. The CURRENT input selects the SEL with the currently executing stream's status, allowing it to respond to STREAMOP (stream operation) commands from the EU. The FCOMPL signal indicates that a prefetch is completing on the selected stream, allowing status bit \(S_0\) (prefetched) to be set accordingly. Similarly, BEGINEX allows \(S_0\) to be cleared when the stream begins execution. An extra state bit (not shown) is used to delay the positive edge of the \(S_0\) bit by one cycle. This is necessary, since the BIU sequencer sends FCOMPL one cycle before the fetch is actually finished.

Each SEL has three outputs: PRQ, SAV and HALTED. PRQ (prefetch request) and SAV (stream available for execution) request selection for prefetching and execution, respectively. HALTED indicates that the stream is in the HALT state. A simple NOR gate (not shown) combines all SEL HALTED signals to produce a NONE_FREE signal that may be tested with a conditional microcode jump. This allows instructions like PCALL
(parallel call) to create a subprocess if NONE_FREE is false, or conventionally call the subprocess if NONE_FREE is true.

5.5 SCU Control

Figure 5.4 shows a simplified diagram of the SCU control logic. Most of the control circuitry is implemented with a single 74PL16R6 PAL IC, which is logically divided into two sections: one that handles external process awake requests and one that handles rotation requests.

During normal operation, the awake handling circuitry simply passes the EU STREAMOP signals to the SEL array. When a positive edge is detected on the AWRI (awake request in) input, an internal flip-flop is set. When a null operation is detected in the EU STREAMOP microcode field, the circuit inserts an AWAKE stream operation code and clears its internal request flip-flop. Since stream operations are infrequent in the microcode, awake requests are typically honoured within 5 clock cycles (1 us at 5 MHz).

The rotation control circuitry is somewhat more complex since a rotation cycle must not coincide with any other status update (see 5.4 above). There are three events that should inhibit rotation: STREAMOP events, FCOMPL events and BEGINEX events. The first is handled within the control PAL itself, but the latter two are processed with external high speed circuitry in an effort to reduce delay in this critical signal path. The SAV signal (from which the BEGINEX signal is
Figure 5.4: SCU Control Logic
derived) is used in lieu of BEGINEX itself for a similar reason.

The rotation mechanism operates as follows: When a positive edge is detected on the SHRI (shift request in) input, or when SEL 14's HALTED output is active, an internal shift request flip-flop becomes set. Then when no STREAMOP is being issued, the SHRQ signal is asserted, causing a SHIFT request to the SELs if FCOMPL and SAV are both inactive. This SHIFT signal also serves to enable the rotation counter and synchronously clear the original request flip-flop, completing the rotation cycle.

5.6 Overall Processor Operation

Figure 5.5 shows a simplified execution sequence for one stream on the prototype. Note that there is a two cycle decoding delay between the fetching of an instruction and it actually beginning execution. If the processor were executing one stream of "no ops", its memory bandwidth efficiency would be only 3/5 or 60%. Of particular importance in this diagram is the operation of Instruction 2, which requires an operand to be fetched. The operand fetch request is shown at the start of the instruction, but a look-ahead system actually places the request to the BIU one cycle earlier. This allows the BIU operand fetch to start at the same cycle as the instruction execution. For immediate operands (following the instruction) the BIU computes the operand address; for other accesses the EU provides this address at the first cycle.
<table>
<thead>
<tr>
<th>Cycle</th>
<th>BIU State</th>
<th>EU State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>2</td>
<td>fetch instr. 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Idle</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>execute instr. 1</td>
</tr>
<tr>
<td>7</td>
<td>fetch instr. 2,</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>idle</td>
</tr>
<tr>
<td>9</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>request</td>
</tr>
<tr>
<td>12</td>
<td>instr. 2 operand</td>
<td>execute instr. 2</td>
</tr>
<tr>
<td></td>
<td>fetch</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>fetch instr. 3</td>
<td>idle</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5: Prototype Execution Sequence for 1 Stream
Figure 5.6 shows an execution sequence for two streams. Times at which the BIU idled in the previous example are "filled in" by the operation of the second stream in an interleaved fashion. Note that now when a stream places an operand fetch request, it may have to wait for up to two cycles for the BIU to complete its current operation. In the prototype a simple synchronizing circuit extends the length of the EU's clock cycle to achieve this synchronization.

Clearly the MBE of the processor is increased over the one stream example. Naturally, if very long instructions (like multiplication in the prototype) execute, then the prefetching registers for all running streams will fill, and the memory will go idle. If more streams are running, this effect is reduced by the greater effective prefetch depth. Thus the effect of long instructions can be "drowned out" by shorter ones.
Figure 5.6: Prototype Execution Sequence for 2 Streams
CHAPTER 6

PROTOTYPE SOFTWARE

6.1 Introduction

This chapter discusses the two main software components needed for meaningful evaluation of the prototype architecture: the machine instruction set and the MPL (Modular Programming Language) high-level language compiler.

6.2 Prototype Instruction Set

6.2.1 Design Considerations

The main design goal for the prototype's instruction set is that it be both symmetric and simple, to facilitate the construction of an efficient high-level language compiler. The limited number of instruction encodings in the prototype (60 single field and 128 double field) precludes a fully composable instruction set like that of the YAX-11[13]. For this reason, a simple load-store instruction set similar to that of the Berkeley RISC I was chosen[15].

The philosophy of this load-store instruction set is that data may be moved to and from the register file using several addressing modes, but most data operations are performed on the register file. The large number of general purpose registers somewhat compensates for
this restriction. For example, the prototype compiler uses these registers for temporary values and for all subroutine parameter passing; a more clever compiler might also perform variable-to-register binding. Efficient utilization of the CPU registers has the effect of increasing processor speed by reducing data fetches from memory. Reducing data fetches is also beneficial to the performance of cached memory, as data references do not exhibit the same locality of reference as code references[3].

For high-level language support, an instruction set should handle multiple data types in a uniform fashion. In the prototype, memory move instructions may refer to byte, word or long-word operands, with most addressing modes supported for each data type. However, data operations may take only word or long-word operands, forcing bytes to be treated as unsigned 8-bit integers. Byte load instructions facilitate this by automatically zeroing the upper 8 bits of the destination register.

Figure 6.1 illustrates the operation of the nine major processor addressing modes. Of particular interest is the short immediate mode "#K" which allows an unsigned 5-bit constant to be embedded in the instruction word itself, similar to the "quick" immediate mode of the Motorola 68000 family[26]. Since most constants in high-level language programs fall within a very small range, this mode saves considerable space and time[13].

Also of interest are the postincrement"(Rn)+" and predecrement "-(Rn)" modes which may be used to implement stack pop and push operations, in PDP-11 fashion[24]. The postincrement and postdecrement
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>In Instruction</th>
<th>In Registers</th>
<th>In Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Register Direct</td>
<td>reg. #</td>
<td>operand</td>
<td></td>
</tr>
<tr>
<td>(Rn)</td>
<td>Indirect</td>
<td>reg. #</td>
<td>address</td>
<td>operand</td>
</tr>
<tr>
<td>#A</td>
<td>Immediate</td>
<td>16-b const</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#K</td>
<td>Short Immediate</td>
<td>5-b const</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Direct Address</td>
<td>16-b addr</td>
<td>operand</td>
<td></td>
</tr>
<tr>
<td>(Rn,A)</td>
<td>Indexed</td>
<td>reg. #</td>
<td>base addr.</td>
<td>operand</td>
</tr>
<tr>
<td>(Rn)+</td>
<td>Postincrement</td>
<td>16-b const</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Rn)-</td>
<td>Postdecrement</td>
<td>reg. #</td>
<td>address</td>
<td>operand</td>
</tr>
<tr>
<td>-(Rn)</td>
<td>Predecrement</td>
<td>reg. #</td>
<td>address</td>
<td>operand</td>
</tr>
</tbody>
</table>

Note: l = length of operand, in bytes

Figure 6.1: Major Addressing Modes
"(Rn)-" mode are also useful for performing the block move operations required for string manipulations.

The indexed "(Rn,A)" addressing mode is useful for data structure access where "A" is the base of the structure and "Rn" contains a computed offset. An equivalent sequence of instructions would require an immediate add instruction in addition to an indirect "(Rn)" move. The prototype compiler emits indexed mode instructions instead of add/move sequences whenever possible.

The following sections present a brief overview of the processor's basic instruction set and of the specific process control instructions which are unique to this machine. A complete listing of the machine's instruction set may be found in Appendix 1. Appendix 2 gives the microcode for several of the instruction types. This instruction set represents only a first approximation to an efficient instruction set; several changes could be made improve the machine's high-level language performance.

6.2.2 Basic Instruction Set

Table 6.1 presents a partial listing of the machine's basic instruction set, giving the mnemonic, function, number of operands, data types and allowed addressing modes for several representative instructions.

The assembly syntax used by the compiler and assembler follows the Motorola convention[23,26]. Thus "move r0,r1" copies the contents of register 0 into register 1. The byte move instruction,
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Opers.</th>
<th>Types</th>
<th>Src Modes</th>
<th>Dst Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>data movement</td>
<td>2</td>
<td>w,l</td>
<td>Rn,#A,#K,(Rn)+(, (Rn), (Rn), (Rn))</td>
<td>Rn,#A,#K,(Rn)+(, (Rn), (Rn))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>w,l</td>
<td>Rn</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b,w,l</td>
<td>l</td>
<td>A,(Rn),(Rn,A)</td>
<td>A,(Rn),(Rn,A)</td>
</tr>
<tr>
<td>pushm</td>
<td>push multiple</td>
<td>2</td>
<td>w</td>
<td>Rn</td>
<td>Rx,Ry</td>
</tr>
<tr>
<td>pØpm</td>
<td>pop multiple</td>
<td>2</td>
<td>w</td>
<td>Rx</td>
<td>Rn</td>
</tr>
<tr>
<td>add</td>
<td>addition</td>
<td>2</td>
<td>w,l</td>
<td>Rn,#A,#K</td>
<td>Rn</td>
</tr>
<tr>
<td>sub</td>
<td>subtraction</td>
<td>2</td>
<td>w,l</td>
<td>Rn,#A,#K</td>
<td>Rn</td>
</tr>
<tr>
<td>cp</td>
<td>compare</td>
<td>2</td>
<td>w,l</td>
<td>Rn,#A,#K</td>
<td>Rn</td>
</tr>
<tr>
<td>neg</td>
<td>2's complement</td>
<td>1</td>
<td>w,l</td>
<td>Rn</td>
<td>Rn</td>
</tr>
<tr>
<td>and</td>
<td>bitwise AND</td>
<td>2</td>
<td>w</td>
<td>Rn,#A</td>
<td>Rn</td>
</tr>
<tr>
<td>or</td>
<td>bitwise OR</td>
<td>2</td>
<td>w</td>
<td>Rn,#A</td>
<td>Rn</td>
</tr>
<tr>
<td>not</td>
<td>bitwise NOT</td>
<td>1</td>
<td>w</td>
<td>Rn</td>
<td>Rn</td>
</tr>
<tr>
<td>tc</td>
<td>test condition.</td>
<td>2</td>
<td>w</td>
<td>&lt;cond code&gt;</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dst &lt;= 0 if false, -1 if true.</td>
<td></td>
</tr>
<tr>
<td>jp</td>
<td>conditional</td>
<td>2</td>
<td>--</td>
<td>&lt;cond code&gt;</td>
<td>Rn,A</td>
</tr>
<tr>
<td></td>
<td>jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>call subr.</td>
<td>1</td>
<td>--</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ret addr on stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>return from subroutine</td>
<td>0</td>
<td>--</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>djnz</td>
<td>decrement src</td>
<td>2</td>
<td>w</td>
<td>Rn</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>and jump if not zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Basic Instruction Set (Partial Listing)
"move.b (r0),r1", copies the contents of the byte whose address is in r0 into register r1.

One will note that the move instruction is not completely symmetric with respect to byte operands. This arises because byte operands are implicitly treated as short unsigned integers; the word-sized immediate and register direct modes are equally useful for byte operands. The postincrement, postdecrement and predecrement modes are not available in byte-size as they are used for stack manipulation and the stack pointer must stay word-aligned for correct word and long-word operation.

A particularly interesting form of the move instruction is "move (Rs)+,(Rd)+" which is used to move blocks of memory, one word at a time. Combined with the DJNZ instruction (Decrement a register and jump if not zero), this instruction allows very fast block moves to be performed. Since the MPL compiler forces all structures to be word aligned, the sequence "move (Rs)+,(Rd)+ / djnz Rc,.l" may be used to implement all structure assignments, even those involving arrays of bytes. This is much more efficient than using byte moves which require read-mask operations for reading and read-modify-write operations for writing.

In addition to "move (Rs)+,(Rd)+" and "djnz", several other instructions are adapted to high-level language support. The "pushm" and "popm" instructions allow groups of registers to be saved to the stack and restored from the stack. These instructions are similar to the Motorola 68000's MOVEM (move multiple)[23,26] and the Zilog Z8000's LDM (load multiple)[8]. These instructions are useful for saving
temporary values in preparation for procedure calls.

The TC (test condition) instruction, which is similar to the Motorola 68000's SC instruction, simplifies logical expression evaluation[23,26]. For example, "tc gte, r0" will set r0 to all ones (true) if the flags indicate a signed greater-than-or-equal condition, or to all zeros (false) otherwise. This eliminates the need for inefficient sequences like "move #0, r0 / jump if..+3 / sub #1, r0".

6.2.3. Task Control Instructions

One of the unique features of the Microcode Level Timeslicing architecture is that complex task control functions may be implemented as machine language instructions. Table 6.2 summarizes the main task control instructions implemented in the prototype.

The CREATE instruction is the main instrument for the creation of new tasks. Four consecutive registers contain the initial program counter value (PC), code segment number (CS), stack pointer value (SP) and data segment number (DS) for the new process. This function would be handled by the operating system software in other computers, taking many machine instructions to accomplish. On the prototype, the function is completed in 8 clock cycles, or in a mere 1.6 μS at 5 MHz.

The other method of creating tasks uses the parallel call instruction, PCALL. In combination with the parallel return instruction, PRET, PCALL allows a sub-task to be executed in parallel if there are free streams or in sequence if not. Structures like the
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Halt this process</td>
<td>8/3 cr/fail</td>
<td>0</td>
</tr>
<tr>
<td>pcall Rs</td>
<td>Create or Call a subprocess / subroutine pc &lt;= R[s] cs &lt;= same cs sp &lt;= R[s+2] ds &lt;= same ds if create: new sp =&gt; 0 if call: new sp =&gt; old pc, old sp</td>
<td>6/8 halt/ret</td>
<td>1/2</td>
</tr>
<tr>
<td>pret</td>
<td>Halt or Return if (sp) = 0 =&gt; halt &lt;&gt; 0 =&gt; pop pc, sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kill Rs</td>
<td>Force stream #Rs to halt</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>sset A</td>
<td>Indivisible test-and-set memory to -1</td>
<td>10/9 set/not</td>
<td>3/2</td>
</tr>
<tr>
<td>sset (Rd)</td>
<td></td>
<td>7/6</td>
<td>2/1</td>
</tr>
<tr>
<td>ssetw A</td>
<td>Indivisible test-and-set-with-wait</td>
<td>11 + BN 3+N</td>
<td></td>
</tr>
<tr>
<td>ssetw (Rd)</td>
<td></td>
<td>9 + BN 2+N</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if dst &lt;= 0 =&gt; set to -1 &lt;&gt; 0 =&gt; WAIT until SIGNAL then repeat</td>
<td></td>
<td>where N = # of loops</td>
</tr>
<tr>
<td>sclr A</td>
<td>Clear semaphore and signal WAITing</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>sclr (Rd)</td>
<td>processes</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.2: Task Control Instructions
PAR construct of concurrent Pascal, where statements may be executed in any order including concurrently, may be directly implemented with these instructions[5]. Because the PCALL/PRET execution sequence adapts to the loading of the processor, they may also be used to maximize the number of streams executing on the processor. This has the effect of maximizing processor MBE by increasing the effective prefetch depth.

The PCALL/PRET mechanism is simple: if there are no free streams, PCALL simply pushes the current SP and PC onto the new task's stack. The processor SP and PC are given their new values, causing a control transfer to the sub-task. When PRET is encountered, the old SP and PC are restored from the sub-task stack. However, if PCALL occurs when there are sufficient resources for task creation, a 0 marker is pushed onto the sub-task stack and a new process is created. Now when PRET is executed, it finds the 0 marker (which is an impossible return address) and simply halts the sub-task. Overhead for this adaptive execution is very low; PCALL and PRET require at most 2.6μS and 1.6μS, respectively, at 5 MHz.

Once a process is created with PCALL or CREATE it may be halted in three ways: with PRET (as described above), with HALT or with KILL. HALT allows a stream to halt itself, as at the termination of a task. KILL allows one stream to force another to halt, which is useful for aborting runaway tasks.

The KILL mechanism itself is quite interesting. KILL simply sets the victim process' code segment number to OFF_{16}, which is a segment that maps into an unused portion of the memory space. Read
operations on this unused area always return $0\text{FFFF}_{16}$ because of passive pull-up devices on the external memory bus. Because the code for the HALT instruction was chosen to be $0\text{FFFF}_{16}$, the next instruction the victim stream executes will be a HALT.

6.2.4 Semaphore Instructions

In any multitasking system, it is necessary to have some form of controlled resource sharing. One way of implementing the required resource mutual exclusion is to have a special control variable, called a semaphore, for each sharable resource[5]. The value of the semaphore indicates if the resource is free or is in use. To acquire access to a resource, a task must wait until that resource's semaphore is clear and then set the semaphore, preventing others from accessing the resource. The task clears the semaphore when it is finished with the resource, allowing other tasks to acquire it. Because instructions of several tasks may be interleaved in unpredictable order, it is essential that the initial test-and-set operation be indivisible.

The prototype's SSET (semaphore set), SSETW (semaphore set with wait) and SCLR (semaphore clear) instructions form the basis for an efficient semaphore system. SSET implements an indivisible test-and-set similar to the TAS (test and set) instruction of the Motorola 68000[23,26]. However, if the SSET instruction is used to implement semaphores, there remains the problem of what to do with a
a waiting task. A simple tight waiting loop or "busy wait" would waste a great deal of the processor's time.

The SSETW instruction solves this "busy wait" problem by causing the task to enter the WAIT state if the semaphore was not clear. No processor time is wasted as a WAITING stream is never executed. When a SIGNAL microoperation is performed by another stream, all WAITing processes switch to RUNNP (run with no prefetch), and may be selected for execution. When the original task is again executed, it repeats the test-with-wait until the semaphore is finally found to be clear. The semaphore is then set to -1 and execution continues at the next instruction.

The SIGNAL microoperation is generated by the SCLR (semaphore clear) instruction, which implements the resource release operation. The net effect of SSETW and SCLR is that when any process releases any semaphore, all WAITing processes awake and re-test their semaphores. This wastes at most one bus cycle per waiting process at each SCLR, which is clearly preferable to a "busy wait". As long as SCLR instructions are relatively infrequent the overhead is negligible.

Although this particular semaphore implementation does waste bus cycles, it is straightforward to design a method that does not. For example, a small (16 element for the prototype) associative memory could hold the address of the semaphore that each task is waiting for. When SIGNAL occurs, only one task whose address matches the semaphore's would be awakened, eliminating wasted memory cycles.
6.3 High-Level Language Compiler

The use of hand-coded benchmarks for processor testing is a questionable practice at best, as most computer programs are now written in high-level languages like C, Modula-2 and Pascal. It has been observed that on CISC machines like the VAX-11, hand coded programs may run considerably faster than compiler generated code[13]. This implies that hand-coded benchmarks may yield unrealistic results.

For this reason it was considered essential that the testing of the prototype be performed with compiler generated code. The following sections detail the design and implementation of the prototype's high-level language compiler. The types of optimizations attempted by the compiler are described, as this does much to determine the size and speed of the compiled code.

6.3.1 The MPL Language Compiler

MPL (Modular Programming Language) was originally developed by the author as a tool for compiler and operating system development. For the latter purpose the compiler was made easily re-targetable. It consists of two sections: the compiler, which produces symbols tables and intermediate code (i-code), and the code generator (coder), which translates i-code to assembly language for the target processor. Only the coder need be re-written to adapt the compiler to other processors.
The MPL language is based largely upon Pascal and C, and incorporates the data and control structures of Pascal with the pointer and separate compilation capabilities of C. MPL programs appear very like Pascal programs except that all structures have specific terminating symbols. For example, the "if-then-else" structure of Pascal becomes the "if-then-else-endif" structure of MPL. This inclusion of "endif", "endwh", "endcase" and so on has two advantages: it makes the code more readable and it removes ambiguities like the "dangling else" from the syntax[27]. Figure 6.2 shows the Sieve of Eratosthenes benchmark[28], which counts prime numbers, coded in MPL:

The present version of MPL allows user-defined array and record data types. The base types "byte" (8-bit unsigned), "integer" (16-bit signed) and "long_int" (32-bit signed) are supported, as is simple string manipulation. Floating point data types are not implemented at present.

The MPL i-code code model is based upon a stack machine. Figure 6.3 shows hand-annotated i-code output for the inner if-then-endif of the Sieve. "Ldd.w _j" (load direct word) pushes the variable j onto the top of stack. "Ld.b _flag" retrieves the byte pointed to by the top of stack plus the offset "_flag", which is flag[j]. "Jf.b L54" implements the test for the if statement. "Ldk.w N" pushes a constant N onto the stack, and so on. This stack machine intermediate model was chosen for three reasons: (1) the i-code is simple to generate, (2) the model makes minimal assumptions about the target machine and (3) the model permits efficient temporary-to-register binding, as will be seen below.
module sieve;

[ Count prime numbers using the Sieve of Eratosthenes. ]

#include lib:stdio.def

cost
    size := 8190; { sieve array size }
    ntimes := 10; { # of iterations }

var
    flag : array[0..size] of byte; { "is prime" flags for numbers from 3 to size*2+3 }
    i : byte; { loop index }
    j,k, count, prime : integer; { primes count }

begin
    print(newln,newln,dec(ntimes)," iterations;",bell,newln)

    { Main loop: }
    for i := 1 to ntimes do
        { Compute the sieve: }
        { Initialize: }
        count := 0 { clear the count }
        for j := 0 to size do { clear the flags }
            flag[j] := true
        endfor

        { Computation: }
        for j := 0 to size do
            if flag[j] then
                { is prime — mark off all odd multiples }
                prime := j shl 1 + 3 { the prime to eliminate }
                k := j + prime { start position }
                while k <= size do
                    flag[k] := false { eliminate odd multiples }
                    k := k + prime
                endwh
                count := count + 1 { have found 1 more prime }
            endif
        endfor

        endfor
    print(newln,bell,dec(count)," primes found.",newln,newln)
end
endmodule

Figure 6.2: MPL Sieve of Eratosthenes
ldd.w _j
ld.b _flag
jf.b L54+0.w

ldd.w _j
ldk.w l.w
shl.w
ldk.w 3.w
add.w
std.w _prime

ldd.w _j
ldd.w _prime
add.w
std.w _k

L55:
ldd.w _k
ldk.w 8390.w
jgt.w L56+0.w

ldd.w _k
ldk.b 0.b
st.b _flag

ldd.w _k
ldd.w _prime
add.w
std.w _k

jump L55+0.w

L56:
ldd.w _count
ldk.w l.w
add.w
std.w _count

L54:

if flag[j] then
prime := j shl 1 + 3
k := j + prime
while k <= size do
flag[k] := false
k := k + prime
endwh
count := count + 1
endif

Figure 6.3: Intermediate Code for the Sieve If-Then-Else
One will note that some "optimizations" have already been performed at the i-code stage. Most notably, the indexed "ld.? offset" and "st.? offset" instructions simplify the use of target processor indexed addressing modes. Also, all constant expressions are evaluated by the compiler to reduce i-code emission where possible. In this version of the compiler no other optimizations, like common sub-expression elimination, are attempted. Some local optimizations are carried out during code generation, as detailed below.

Figure 6.4 shows the final compiled assembly code for the inner for-do loop of the Sieve.

6.3.2 Code Generation Phase Opti3mizations

The coder performs two specific types of local optimization: argument-to-register and temporary-to-register binding, and peephole optimization.

6.3.2.1 Argument-to-Register and Temporary-to-Register Binding

The calling convention used by the prototype compiler is simple: arguments are passed entirely within the register file, with R0 holding the first word of the first argument and other registers allocated in ascending order. R31 is reserved for use as the stack pointer, allowing at most 31 words of arguments to be passed. Since structures are passed as pointers to subprograms, this limit presents few problems. Experience indicates that only rarely would even 10 parameter words be
move.w #$0000,r0 ; for j = 0 to size do
    cp.w #$1FFE,r0
    jp 1t,L53
    move.w r0, _j

    move.b (r0, _flag), r0 ; if flag[j] then
    tst.w r0
    jp eq,L54

    move.w _j, r0 ; prime := j shl l + 3
    sl.w r0
    add.w #$0003, r0
    move.w r0, _prime

    move.w _j, r0
    move.w _prime, r1
    add.w r1, r0
    move.w r0, _k

L55:
    move.w _k, r0 ; while k <= size do
    cp.w #$1FFE, r0
    jp 1t, L56

    move.w _k, r0
    move.w #$0000, r1
    move.b r1, (r0, _flag)

    move.w _k, r0
    move.w _prime, r1
    add.w r1, r0
    move.w r0, _k

    jp L55 ; endif

L56:
    move.w _count, r0 ; count := count + 1
    inc.w r0
    move.w r0, _count

L54:
    move.w _j, r0
    inc.w r0
    jp L52

L53:

Figure 6.4: Assembly Code for the Sieve Inner Loop
required for most conceivable applications.

The argument-to-register and temporary-to-register binding algorithm is simple to implement. The coder treats the register file as a virtual stack and uses the actual machine stack only when it is necessary to preserve temporary values during function calls. The compiler provides some semantic information in the i-code to indicate to the coder when parameters for a procedure or function are being generated. Figure 6.5 illustrates this process for an MPL assignment statement.

At the start of the expression the register file from r0 to r2 is used as a "stack" to hold the values of i, j and k. The "params" i-code signals the coder that parameters for a subprogram are to be generated, requiring that temporary values be saved on the machine stack. The "pushm" instruction is used where more than one register must be saved, otherwise the simpler "move.w r0,-(sp)" instruction is used.

Parameter generation into r0, r1 and r2 follows. The "call.w" i-code causes the call to be made and indicates that a word value is returned. This value is returned in r0 and is moved by the coder into r3, the next register tagged for temporary binding. The values of the other temporaries remain on the stack until needed.

One should note that this is a "worst case" example; a simple re-arrangement of the equation to "((f(i,j,k) + k) * j) * 2 + i" completely eliminates the need for saving temporaries onto the stack. Inspection of compiler output indicates that save/restore sequences are infrequent in actual programs.
MPL Source Statement:
\[ l := (i + (j \times (k + f(i,j,k))) \times 2) \]

<table>
<thead>
<tr>
<th>Intermediate Code</th>
<th>Prototype Assembly Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldd.w _l</td>
<td>move.w _l,r0</td>
<td>; i,j,k retrieval</td>
</tr>
<tr>
<td>ldd.w _j</td>
<td>move.w _j,r1</td>
<td></td>
</tr>
<tr>
<td>ldd.w _k</td>
<td>move.w _k,r2</td>
<td></td>
</tr>
<tr>
<td>params</td>
<td>pushm r0,r2</td>
<td>; param. generation;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; must save temps r0..r2</td>
</tr>
<tr>
<td>ldd.w _l</td>
<td>move.w _l,r0</td>
<td>; generate parameters</td>
</tr>
<tr>
<td>ldd.w _j</td>
<td>move.w _j,r1</td>
<td></td>
</tr>
<tr>
<td>ldd.w _k</td>
<td>move.w _k,r2</td>
<td></td>
</tr>
<tr>
<td>call.w f##</td>
<td>call _f##</td>
<td>; call the subroutine (returns a word)</td>
</tr>
<tr>
<td></td>
<td>move.w r0,r3</td>
<td>; reposition the return value</td>
</tr>
<tr>
<td>add.w</td>
<td>move.w (r31)+,r2</td>
<td>; pop _k</td>
</tr>
<tr>
<td></td>
<td>add.w r3,r2</td>
<td></td>
</tr>
<tr>
<td>mul.w</td>
<td>move.w (r31)+,r1</td>
<td>; pop _j</td>
</tr>
<tr>
<td></td>
<td>mul.w r1</td>
<td>; r2r1 &lt;= r2 \times r1</td>
</tr>
<tr>
<td>ldk.w 2</td>
<td>sl.w r1</td>
<td>; multiply * 2 =&gt; left shift</td>
</tr>
<tr>
<td>mul.w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.w</td>
<td>move.w (r31)+,r0</td>
<td>; pop _l</td>
</tr>
<tr>
<td></td>
<td>add.w r1,r0</td>
<td></td>
</tr>
<tr>
<td>st.w _l</td>
<td>move.w r0,_l</td>
<td>; store result in _l</td>
</tr>
</tbody>
</table>

Figure 6.5: Temporary Binding and Parameter Passing
6.3.2.2 Peephole Optimization

Peephole optimization is performed by the coder in two "passes". The first pass operates on the incoming i-codes, converting sequences like "ldk.w 4 / add.w" to simple add immediate or increment assembly instructions. The second pass operates directly on the outgoing assembly code, eliminating redundant instructions as in "move.w r0, _i / move.w _i, r0", where the second instruction is clearly redundant.

6.4 Summary

For the evaluation of the prototype's performance to be meaningful, it is necessary to perform the evaluation using high-level language benchmarks. For this reason the prototype's instruction set is designed to support high-level language data types and constructs. A compiler for the author's Pascal-like high-level language MPL has been developed for benchmarking purposes. The prototype version attempts only simple local optimizations and performs argument-to-register and temporary-to-register binding. Although this is sufficient for evaluation purposes, a more complex compiler will be needed if the full potential of this processor is to be exploited.
CHAPTER 7

PERFORMANCE MEASUREMENT

7.1 Introduction

The evaluation of any computer architecture is a difficult undertaking. Ideally, a complete system should be available so that actual applications programs may be run. As the prototype system has little more than the CPU and memory, and input/output must be done over the slow 8-bit CPU to System Controller link, this type of test is difficult.

At present, the best that can be done to test the prototype processor is to run well-known high-level language processor-intensive benchmark programs upon it. Since Microcode Level Timeslicing is a multistream architecture, these benchmarks must be parallelized so that the processor's multitasking performance may be evaluated. These parallel benchmarks may then be used to perform elementary performance measurements under different task loads.

This chapter discusses the chosen processor-intensive benchmark programs and describes the method used to run them in a multistream fashion. The measurement techniques for the performance-related parameters are then presented, and the significance and inter-relationships of these quantities are discussed.
7.2 Benchmark Software

The benchmark programs must be processor-intensive because no high-speed input/output is presently available on the prototype. Beyond this, the benchmarks should be well-known and portable; it is desirable that results for many computers be available. Since floating point is not supported in the prototype or in the compiler, the benchmarks are limited to integer arithmetic only. These restrictions led to the choice of two well-known benchmarks: the Sieve of Eratosthenes[28] and the Dhrystone[29].

7.2.1 The Sieve of Eratosthenes

The Sieve of Eratosthenes, which counts prime numbers, was proposed as a high-level language benchmark in 1981[28]. It is a simple, processor-intensive program that tests the speed of looping constructs and of 1-dimensional array access. Only simple operations like addition are required by the Sieve; multiplication and division are not needed. The source code for a non-parallel Sieve may be found in Figure 6.2. Although the scope of the Sieve benchmark is very narrow, a great deal of performance data is available for it[28,30,31].

The main reason for using the Sieve is that it is simple enough to be hand compiled. Since the result of the Sieve is one number (1899), its correct operation may be verified by inspecting CPU registers. The main section of the Sieve requires no subroutine calls.
eliminating the need for linking. These simple requirements allowed
testing of the prototype in the very early debugging stages, long before
the MPL compiler, linker and library were completed.

The Sieve represents one instruction mix extreme. It is a
program that performs a great deal of data movement with very little
computation. Operations like multiplication, which are very time
consuming on the prototype, are not used. For this reason, as the
number of parallel Sieve tasks increases, the performance should
increase rapidly to maximum at 100% MBE.

7.2.2 The Dhrystone Benchmark

The Dhrystone benchmark program was proposed by Weicker in
1984[29]. The Dhrystone is designed so that statement, operand and
operator frequencies reflect the weighted averages of those found in
several high-level language studies, with the most weight placed upon
studies using systems programs. Thus one would expect the Dhrystone
performance of a computer system to correlate with its performance on
compilers, editors and similar programs. Since its introduction, the
Dhrystone has gained wide acceptance, becoming one of the more popular
computer benchmarks; results for hundreds of computer systems are
available. A listing of a non-parallel MPL version of the Dhrystone
(adapted from the Pascal version), may be found in Appendix 3.

Like the Sieve, the Dhrystone does not use floating point
arithmetic. However, the program does use 5 multiplications per main
loop (4 of which are implicit in array references). Because of the
presence of these time-consuming instructions, one would expect the processor MBE to increase more slowly to 100% than that of the Sieve, as processes are added. This effect is further examined in Chapter 8.

The Dhrystone benchmark serves two purposes: it provides a reasonable means of comparing the prototype’s performance to that of other computer systems, and it allows assessment of the effect of the prototype’s inefficient multiply instruction.

7.2.3 Memory Organization and Support Software

It is useful to examine the software support developed for system benchmarking before the method for running parallel benchmarks is discussed. Of particular importance are the process memory organization and the monitor software.

Figure 7.1 shows the memory map of each process’ data and code space. Each task has 24K words (48K bytes) of data and 32K words of code available. In the data area there are two additional 4K word pages. One of these is reserved for monitor use, and the other is mapped into the same 4K page for all tasks. This common data page is where shared variables, like semaphores, are kept. Note that the data area from 8000H to FFFFH (words) is unusable, since the processor’s word-even addressing scheme allows access only to the first 64K bytes or 32K words.

The simple monitor program is called the MultiStream Executive, or MUSE. The MUSE kernel resides in a shared page at E000H (words) in each process’ code space. MUSE provides simple system services like
Data Space

(.words)

$FFFF

Unusable and Unallocated (32K words)

$8000
$7FFF

Common to all tasks (4K words)

$7000
$6FFF

Monitor Related Variables (4K words)

$6000

Task-Specific Data Area (24K words)

$0000

Code Space

(.words)

$FFFF

Monitor Kernel (Common to all tasks) (4K words)

$E000

Unallocated (28K words)

$8000
$7FFF

Task-Specific Code Area (32K words)

$0000

Totals:

1 Shared data page (4K words)
1 Shared code page (4K words)
7 Data pages per task (28K words)
8 Code pages per task (32K words)

Figure 7.1: System Memory Map for Testing
file handling to executing processes. In the initial version of MUSE, almost none of these services are implemented in the monitor itself. Instead, the monitor relays service requests to the "real" operating system on the System Controller (SCON), via the prototype-to-SCON.8-bit link. After each request is processed, the SCON relays the results back to the prototype. Having the SCON handle all I/O requests in this fashion greatly simplified the design of MUSE, and greatly reduced its implementation time.

Semaphore interlocks allow multiple streams to use MUSE simultaneously, although input/output requests are typically handled one at a time. The MUSE interlocks are implemented with SSETW (semaphore-set-with-wait) instructions, eliminating busy-waiting while a stream awaits MUSE access. However, because the external AWAKE system is not presently used in the prototype-SCON dialogue, a task with a request in service by the SCON will busy-wait until service is complete. The effect on benchmark performance is negligible, as only the GTICKS (get tick count) facility is used by the benchmarks (twice per stream during timing runs), and this facility executes very rapidly (<1mS).

MUSE greatly simplifies the benchmarking process, as programs may use normal operating system services during their execution. This allows benchmarks to be executed on command from the SCON console, and allows their results to be displayed in readable form.
7.2.4 Multistream Benchmarking

A mechanism for running benchmarks on several streams simultaneously is required in order to characterize the multistream performance of Microcode Level Timeslicing. The main requirement is that all of the created processes must start together and be individually timed. It is also desirable that the most of the process control be done with the MPL high-level language. To facilitate this, a set of interlock variables and MPL callable machine-language subroutines were developed. The definitions of the synchronizing variables are shown in Figure 7.2.

The structured variable, Synch, contains all of the semaphores and counters needed to track the operation of the benchmark processes. The address of Synch is forced to be at "synch_base", which is in the the data page that is shared between processes. This allows all of the processes to access the variable. The Synch structure contains both synchronizing variables and an array, Ticks, which holds the start and stop tick counts for each process. The tick count is obtained from the System Controller (SCON), and is the number of 10 mS intervals since power-up.

The operation of the synchronizing mechanism is as follows. The main program records its stream number in orig_stream, locks the start_gate and the finish_gate by setting them to -, and clears the active_lock semaphore. It also sets the active_counter to the number of streams to be run in parallel. The main program initializes the code
const
synch_base := $e010;  // shared data area location

type
{ timing array }
t_array := array[0..15] of record
  start,          // start time
  finish : long_int  // finish time
endrec;

{ shared record structure }
s_record := record
  orig_stream,     // originating stream #
  active_counter,  // active stream count
  active_lock,     // lock semaphore for active
  start_gate,      // start synch gate
  finish_gate : integer;  // final synch gate
  ticks : t_array   // elapsed time tracking array
endrec;

var
{ the shared variable itself }
synch : s_record location synch_base;  // force variable location

Figure 7.2: Multistream Benchmark Synchronizing Variables
and data spaces of each stream to be run by copying its own code and
data space into theirs. Each new process is then created, one after
another, and each begins executing a procedure called Process. When all
have been created, the main task itself calls Process. The interlocks
are arranged so that all of the other tasks are held until the main task
begins, allowing all of the streams to start together.

Figure 7.3 shows a skeleton of the Process procedure. When
each task begins, it first checks its own stream number, via function
cstream(), to see if it is the original stream. If not, the new process
attempts a semaphore-set-with-wait on start_gate using procedure
ssieze(), and since start_gate was initially set by the main program,
the new process waits. Eventually, the main task itself begins
Process, and releases the start_gate using procedure srelease(). Each
of the waiting processes now awake in turn, finish setting start_gate,
and then clear start_gate, permitting another waiting process to
proceed. This causes the processes to start virtually simultaneously.

It is interesting to note that the cstream() function and the
ssieze() and srelease() procedures require very few machine instructions
to implement. Szieze(), for instance, which performs the semaphore
set-with-wait, requires only two instructions -- an SSETW and a RET
(return).

After passing the start_gate, each process records its starting
time in the ticks array, executes the benchmark and records its ending
time. After this, all that remains is to ensure that the sub-tasks are
properly halted and that the main task simply returns from the Process
subroutine. Together, the active_counter and the finish_gate accomplish
global procedure process();
{
  The process to be run in parallel
}
begin
{
  Synchronization preamble:
}
if cstream() = synch.orig_stream then
{
  start the other streams
  srelease("synch.start_gate")  { charge!!! }
}else
{
  must wait until the start_gate is clear
  ssieze("synch.start_gate");  { wait until start }
  srelease("synch.start_gate")  { allow the next to proceed }
endif

synch.ticks[cstream()].start := gticks();  { start time }
{ the process itself }
synch.ticks[cstream()].finish := gticks();  { finish time }
{
  Synchronization postamble:
}
ssieze("synch.active_lock");  { lock the active_counter }
synch.active_counter := synch.active_counter - 1;
if synch.active_counter = 0 then
{
  this is the last process through
  srelease("synch.finish_gate")  { release the main task }
}endif
srelease("synch.active_lock");  { release the active_counter }
if cstream() <> synch.orig_stream then
  halt();  { if not the main task }
endif;
ssieze("synch.finish_gate");  { forces the main task to wait }
end ( process );

Figure 7.3: Parallel Process Skeleton
this. Each terminating task obtains exclusive access to the shared active_counter by locking the active_lock semaphore. The counter is then decremented. If the counter goes to zero, then the current task must be the last one to complete (active_counter is initialized to the number of tasks to run). If so, then finish_gate is cleared. The task proceeds to check its stream number against the orig_stream number; if they do not match, then it is not the main process, and it halts. Only the main process will execute the final ssize(synch.finish_gate) statement, which has the effect of forcing it to wait until finish_gate is cleared, i.e., when all tasks have completed. When the main task finally returns to the main program, all other tasks will have halted.

The main program proceeds to compute and report starting times, ending times, elapsed times, et cetera. The measurement process is completely automated, and timing accuracy on the order of 10mS is achieved.

7.3 Parameters and Measurement Techniques

Since the main features of Microcode Level Timeslicing are high processor MBE, low overhead context switching and efficient resource sharing, processor characterization should deal with each of these three properties. It is also desirable to gather data to aid in comparing MLT with other architectures. For example, jump statistics are useful in estimating what the MBE of a similar computer with conventional prefetching would be. The next sections consider the measurement of these facets of the prototype’s performance.
7.3.1 Basic Measurement Technique

The method used to measure the aspects of processor performance is very simple. Processor test points provide simple l-pulse-per-event markers, and the frequency of these event pulses is measured with a frequency counter. To measure a parameter like memory bandwidth, for example, the counter is connected to a l-pulse-per-memory-access test point. The test program is then run. A frequency counter sample window of 1 second allows several readings to be recorded for each test run; these readings are averaged after discarding any anomalous initial and final readings.

After acquisition, consistency checks are applied to the data, where possible. For example, the Millions of Instructions per Second (MIPS) to Memory Bandwidth Efficiency ratio should be constant for a given benchmark, independent of the number of parallel copies running. Readings that appear inconsistent are re-checked.

7.3.2 Memory Bandwidth Efficiency and Processing Speed

Maximizing memory bandwidth efficiency is one of the major goals of the MLT architecture. For this reason, it is of particular interest to examine how processor MBE is affected by the number of streams executing; ideally the MBE should go to 100% as more streams are added. A simple l-pulse-per-memory-access test point is used to find the actual memory bandwidth utilization. Since the system clock runs at 5 MHz, and
3 cycles are required per access, maximum memory bandwidth reads 5/3 MHz. The ratio of the measured value to this maximum value gives the processor MBE.

Relative processing speed in MIPS (millions of instructions per second) is measured in a similar fashion, with a 1-pulse-per-instruction-decode test point. As was previously stated, MBE and MIPS readings are related by a constant for a given application, allowing the MIPS/MBE ratio to be used for result consistency checking. The change in MIPS and MBE as more streams are run indicates how much processing speed improvement is achieved.

7.3.3 CPU Hardware Utilization

The utilization of the Execution Unit (EU) is also of interest, as the MLT architecture trades EU utilization for processor MBE. Recall that the EU may be inactive if it is waiting for EU-BIU synchronization or if it is waiting for an instruction to be fetched. If neither of these events occur, then the EU is executing useful microcode.

A pulse generator in the microsequencer permits measurement of the EU utilization by generating 1-pulse-per-clock-cycle whenever EU WAIT is not active and the EU is not executing the idle loop at OFFFH (see Section 4.4.2). The average pulse frequency, divided by the system clock frequency, gives the proportion of time that the EU spends performing useful work. Another generator produces 1-pulse-per-EU WAIT, permitting the amount of EU time lost in EU-BIU synchronization to be measured also.
7.3.4 Rotation Compensation Effect

Of particular interest is the effectiveness of the rotation compensation system, which ensures equal time distribution to each task (Section 5.2). A switch on the SCU circuit board allows the system to operate normally or with compensation disabled. To test the effectiveness of the rotation system in distributing time fairly, runs are made in both modes with different numbers of tasks.

The benchmarking software records start and stop times for each process, allowing checking of the process time distribution. For ease of discussion, we make the following definitions: Let Tmax be the longest time taken by any process to execute, and Tmin be the minimum time taken. Then the process execution skew is given by:

\[
\text{Execution Skew} = \frac{(T_{\text{max}} - T_{\text{min}})}{T_{\text{max}}}
\]

If the system works perfectly, there should be almost no skew with compensation enabled. Without compensation, one would expect the skew to peak at about 1/2 the maximum stream count.

7.3.5 Semaphore Overhead

Measuring Semaphore overhead involves determining the amount of time spent by the processor in testing and re-testing semaphores. Recall that the re-testing overhead arises because there is only one
process awake mechanism; when a semaphore is cleared, all waiting processes must awake and re-test their semaphores. Since the prototype at present has very little input/output capability and does not execute an actual operating system, this overhead is very difficult to measure. At best the operation of the semaphore mechanism may be checked, and estimates of the semaphore overhead made.

7.3.6 Benchmark Characteristics

The basic method for measuring benchmark characteristics is very simple. In the microcode word (Figure 4.13) are 4 unused microcode bits designated M3..M0. These Marker bits are used to indicate the occurrence of specific events in the instruction stream. The two main events marked are Jump Encountered and Jump Taken. A Jump Encountered event indicates the decoding of any jump, call or return instruction. Jump Taken indicates that a jump, call or return caused a change in program flow. The frequency of each of these events is proportional to the processor MIPS (and MBE), for a given program.

One may use the frequency of Jump Encountered events to compute the proportion of jump instructions in the instruction stream, by simply dividing by the average number of instructions per second. For example, if the resulting value is 0.2, then on the average 1 in 5 instructions is a jump instruction. Similarly, one may use the frequency of Jump Taken events to compute the proportion of taken jumps in the instruction stream. These two statistics should be constant for a given program, regardless of the number of running streams.
These two figures -- the proportion of jumps and the proportion of jumps taken -- may be used to estimate how well a processor with a similar instruction set would perform. For example, if a processor with prefetching executes a program where the taken jump proportion is 0.2, then, assuming that 1 instruction is always prefetched, one would expect an MBE figure of about 80%. Likewise, estimates for pipelined processors and pipelined processors with delayed jumping may be made. This gives some basis for comparing the MBE performance of the MLT architecture with other architectures.

7.4 Summary

Two processor-intensive benchmarks -- the Sieve of Eratosthenes and the Dhrystone -- are used for testing the prototype system. The former measures processor performance on looping constructs and array access, while the latter is based on systems program statistics. These benchmarks are supported by a simple monitor, program, MUSE. Parallel execution is controlled in the MPL high-level language with the aid of shared data structures and special machine-language subroutines.

The use of test-point circuitry allows the measurement of all major performance parameters. The processor MBE and MIPS rate, EU hardware utilization and rotation compensation effect are easily measured, as are jump statistics on the benchmark programs. Only the semaphore overhead is not easily measureable with the present system.

Chapter 8 presents the actual results obtained from benchmark runs on the prototype, and discusses the significance of these
findings.
CHAPTER 8
PERFORMANCE

8.1 Introduction

This chapter briefly outlines the problems encountered in building the prototype processor and presents the results of the benchmark tests described in Chapter 7. These results are discussed, and the prototype’s performance is compared to that of more conventional computer systems.

8.2 Problems with the Prototype

Some difficulties were encountered in making the prototype MLT processor operational. The main problem was clock skew, where the system clock did not occur simultaneously on all circuit boards. Since advanced Schottky transistor-transistor logic (ASTTL) components were used throughout, timing deviations as small as 4 ns may cause problems. Since a logic analyzer was not available, debugging was quite difficult.

At present, the prototype works perfectly at clock speeds up to 6 MHz, as long as the priority rotation system is disabled. With rotation enabled, processor operation is reliable for single-stream programs, but is less stable for multi-stream programs. Fortunately, the system works well enough for short multistream benchmark runs.
In order to reduce the chance of erroneous results, checks are performed in each benchmark program to verify that the computed results are correct. Multiple test runs are made to further reduce the chance of error. Despite the problems with the prototype, we have confidence in the validity of the testing results.

8.3 Experimental Results and Analysis

The following sections examine each facet of the prototype's performance, as outlined in chapter 7. The measured results themselves may be found in the tables in Appendix 4.

8.3.1 Memory Bandwidth Efficiency and Processing Speed

Figure 8.1 shows the prototype processor's memory bandwidth efficiency for the Sieve and the Dhrystone benchmarks as a function of the number of running streams. As expected, the Sieve performance increases to unity faster than the Dhrystone performance, as streams are added.

For the Dhrystone, the average MIPS/MBE ratio was found to be 0.851 (S.D. = 0.0006); for the Sieve it was 0.704 (S.D. = 0.008). These figures depend only on the instruction mix; they are constant for a given benchmark. Using the MIPS/MBE ratio and the maximum memory bandwidth of 1.6667 M references/second, it is simple to deduce \( N_t \), the average number of (word) memory transfers per instruction for each program. For the Sieve, \( N_t \) is 2.4 transfers per instruction; for the
Figure 8.1: Memory Bandwidth Efficiency as a Function of N
Dhrystone, $N_t$ is 2.0 transfers each. The higher figure for the Sieve indicates that it performs a greater proportion of data movement than the Dhrystone. The fact that the 1-stream MBE for the Dhrystone is less than the 1-stream MBE for the Sieve is consistent with this observation.

As we observed in Sections 7.2.1 and 7.2.2, this difference in the amount of data movement should mean that the Sieve's MBE will increase faster to unity as streams are added. This is indeed what happens; the Sieve reaches 100% performance at $N = 2$ while the Dhrystone's performance increases rapidly to 96.2% at $N = 3$ and then slowly increases towards 100%.

It is interesting that the performance increase for the Dhrystone for $N > 2$ is nearly linear. This linear increase may be attributed to the prototype processor's inefficient multiplication instruction, which takes an average of 46 clocks to execute. Naturally, instructions requiring many clock cycles to execute allow the prefetch "queue" to fill, causing the BIU to go idle, reducing processor MBE. We may estimate the effect of the processor's multiply instruction as follows:

The multiply instruction takes an average of 46 cycles to execute. This is sufficient time to fetch $46/3 = 15.3$ instructions. Therefore, for the processor MBE to go to 100%, we need about 16 streams executing, assuming that the multiplication delay is the main factor degrading processor MBE, and that multiplications do not occur together. With $N$ streams running, the number of BIU clock cycles wasted per multiply is approximately $46 - 3N$, $N < 15$. If multiplications occur
with frequency \( F_m \), and the processor clock speed is \( F_c \), then the expected processor MBE is:

\[
MBE = \left[ 1 - \frac{F_m}{F_c} \left( 46 - 3N \right) \right] \times 100\% \quad N < 15 \quad (1)
\]

The problem now becomes one of computing \( F_m \), since \( F_c \) is 5 MHz. In the Dhrystone benchmark, 5 multiplications occur per main loop (4 of which are implicit in array references). If \( F_d \) is the number of Dhrystone loops per second, then \( F_m = 5 \times F_d \). Now let the \( F_{d_{max}} \) represent the ratio \( F_d/MBE \), which is a constant. We now have:

\[
F_m = 5 \times F_{d_{max}} \times MBE \quad (2)
\]

Substituting (2) into (1) and solving for \( MBE \) gives a simplified estimate for the multiply effect on processor MBE:

\[
MBE = \left[ \frac{1}{1 + \frac{5 \times F_{d_{max}}}{F_c} \left( 46 - 3N \right)} \right] \times 100\% \quad N < 15 \quad (3)
\]

This equation estimates the processor MBE, assuming that the inefficient multiplication instruction is the only factor governing MBE, and that multiplications are infrequent enough to allow the queue to empty between them. \( F_{d_{max}} \) is found from the measured data to be 984.6 Hz/100\% (S.D. = 4.6), and \( F_c \) is simply 5 MHz. Figure 8.2 compares
Figure 8.2: Dhrystone MBE Performance and the Multiply-based MBE Estimate
the estimate in (3) with the measured processor performance. The match is excellent for \( N \geq 3 \), indicating that for 3 or more streams, the multiplication instruction's effect on processor MBE is dominant.

The implication of this is obvious: were the inefficient multiply instruction corrected, the processor MBE would reach 100% at \( N = 3 \). This implies that a small number of streams suffices to realize full MBE improvement; in the prototype, 4 would suffice if there were no anomalous time-consuming instructions.

8.3.2 **EU Utilization**

The measurements support the intuitive expectation that the EU utilization / MBE ratio is constant for a given program. For the Sieve, the EU utilization / MBE ratio was found to be 0.603 (S.D. = 0.003); for the Dhrystone, it was found to be 0.662 (S.D. = 0.0007). Thus at 100% MBE, the processor's EU is 60% utilized by the Sieve program and should be 66% utilized by the Dhrystone. The rest of the time, the EU is waiting for EU-BIU synchronization or for instruction fetching.

Figure 8.3 graphs the amount of time the EU spends waiting for EU-BIU synchronization. The relationship of this parameter to the other performance parameters is not a simple ratio. For 1 stream running, very little EU time (1-2%) is spent waiting for synchronization because each instruction begins execution before the next fetch can begin. The only time an EU WAIT occurs is when a long instruction (like a byte move) allows the next instruction to begin fetching before the current
Figure 8.3: EU Time Taken for EU - BIU Synchronization
instruction's completion. When an instruction accesses memory after the next fetch begins, an EU WAIT may occur. For more than 1 stream, prefetching may easily conflict with operand fetches, increasing the EU WAIT time. However, the exact proportion of EU WAIT time partly depends on the sequence in which instructions are executed, making the EU WAIT proportion difficult to estimate.

One simple estimate of EU WAIT time may be obtained from knowledge about the average number of word transfers per instruction, $N_t$, computed above. Since the first word transferred in the prototype is the instruction opcode, the average number of EU requested memory transfers per instruction is simply $N_t - 1$. If $N_{wait}$ is the average number of cycles the EU must wait at the start of these transfers, $F_c$ is the processor clock speed and $IPS$ is the number of instructions executed per second, then:

$$EU\ WAIT = \left[ (N_t - 1) \frac{N_{wait}}{F_c} \frac{IPS}{IPS} \right] \times 100\% \quad (4)$$

Since the number of wait cycles ranges from 0 to 2, we might take $N_{wait} = 1$ as a reasonable estimate. From the Dhrystone results, $N_t$ is approximately 2, and $IPS$ is about 0.8 MIPS. From (4), one would expect an EU WAIT proportion of about 16%. The actual EU WAIT figures for the Dhrystone are about half this, at about 8.6% for $N > 3$. For the Sieve, the EU Wait time plateaus at about 10%. It is likely that this discrepancy between the simple estimate and the actual data arises from instructions like the indexed move in the prototype's
Instruction set. An indexed move fetches an immediate base value and then performs a memory access. Because the second access immediately follows the first, only the first access can cause the EU to wait. Thus *N*wait may actually be less than 1.

The typical EU utilization of 65% and the typical EU WAIT proportion of 10% imply that the EU waits for instruction fetching about 25% of the time. Thus, in the prototype, this 25% of the EU's performance is traded for 100% processor MBE.

8.3.3 Rotation Compensation Effect

The operation of the Stream Control Unit's rotation compensation system is particularly spectacular. Recall from Section 7.3.4 that the process Execution Skew is defined as \((T_{\text{max}} - T_{\text{min}}) / T_{\text{max}}\), where \(T_{\text{max}}\) and \(T_{\text{min}}\) are the minimum and maximum times for processes to complete execution, respectively. Figure 8.4 compares the Sieve benchmark's execution skew with compensation enabled to its skew with compensation disabled.

With compensation enabled, the Sieve execution skew is very small \((< 0.004)\). However, if compensation is disabled, the skew peaks at \(N = 5\) streams to 0.42, a value over 100 times greater than that found with compensation enabled. To place this in perspective, a skew of 0.004 would correspond to a 14 second time distribution inequality over 1 hour; a skew of 0.42 would correspond to a 25 minute inequality. The difference in measured skew indicates that the rotation compensation system does indeed work, and that its presence is essential to ensuring
Figure 8.4: Execution Skew as a Function of N
a fair time distribution. Furthermore, it appears that distributing fetch and execute priority has a similar effect to distributing actual execution time.

8.3.4 Semaphore Operation

The lack of a complete operating system prevents complete evaluation of the overhead associated with the present semaphore mechanism, although the correct operation of instructions like SSETW (semaphore set with wait) and SCLR (semaphore clear) has been verified. From the microcode, the overhead involved in the SSETW instruction is known to be 8 clock cycles per re-test. Thus if 8 streams were waiting to access semaphores, and another stream cleared another semaphore, the 8 streams would waste 64 clock cycles re-testing their semaphores. If this occurred 1000 times per second, a 1.2% processing time overhead would be incurred. However, without actual semaphore use statistics, it is difficult to make an overhead estimate with any great confidence.

8.3.5 Instruction Set and Benchmark Characteristics

Table 8.1 summarizes the measured characteristics of the prototype's benchmarks. There are four main parameters: Pj, Pjt, Na and Nw. Pj is simply the proportion of instructions that are jump, call or return instructions. Pjt is the proportion of instructions that are taken jumps, calls or returns. Pj and Pjt are measured using the event marker bits in the microcode word (see Section 7.3.6). Na is the
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Sieve</th>
<th>Dhrystone</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pj</td>
<td>Proportion of jump, call or return instructions (dynamic)</td>
<td>22.3%</td>
<td>23.4%</td>
<td>22.9%</td>
</tr>
<tr>
<td>Pjt</td>
<td>Proportion of taken jump, call or return instructions (dynamic)</td>
<td>12.1%</td>
<td>16.6%</td>
<td>14.4%</td>
</tr>
<tr>
<td>Na</td>
<td>Average number of (word) memory accesses per instruction (dynamic)</td>
<td>2.4</td>
<td>2.0</td>
<td>2.2</td>
</tr>
<tr>
<td>Nw</td>
<td>Average number of words per instruction opcode (static)</td>
<td>1.78</td>
<td>1.74</td>
<td>1.76</td>
</tr>
<tr>
<td>Nw/Na</td>
<td>Proportion of memory accesses for code words</td>
<td>0.74</td>
<td>0.87</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Table 8.1: Instruction Set and Benchmark Statistics
average number of (word) memory accesses per instruction, determined above. NW is the average number of words per instruction opcode (i.e., the opcode and any immediate operands), and is determined by static measurements on compiler generated code. PJ and Pjt in particular are useful for estimating the performance of comparable non-MLT processors.

It is interesting to note that PJ and Pjt (average values 22.9% and 14.4%, respectively), are comparable to those reported by Lee and Smith in their study of branch probabilities on IBM, DEC and CDC computer programs. They found average PJ and Pjt to have values of 24% and 16% respectively[10].

8.4 Comparison With Other Systems

It is necessary to compare the performance of the MLT prototype with other similar computer systems in order to evaluate this architecture's effectiveness. The following briefly compares the prototype's performance with other modern mini and microcomputers.

8.4.1 Memory Bandwidth Efficiency

The MLT prototype does achieve very high memory bandwidth efficiencies -- 100% for the Sieve \( N > 1 \) and 96.5% for the Dhrystone \( N = 4 \). Were the multiply instruction made more efficient, 100% MBE would be expected on the Dhrystone benchmark for \( N > 2 \). Thus the prototype does achieve MBEs near 100%.

For comparison, let us consider a system identical to the
prototype, but with deep conventional prefetching instead of horizontal prefetching. We know that the probability of a taken jump instruction, \( P_{jt} \), is about 14\% for the prototype instruction set. Assuming that the other processor manages, on average, to fetch 1 word ahead, one would expect an MBE of about 100\% - \( P_{jt} = 86\% \). If the average number of prefetched words were greater, the MBE would be correspondingly lower. Hence the MLT prefetching mechanism yields about a 14\% increase in MBE and processing speed over conventional prefetching.

Conventional processors also exhibit low MBE figures. In the 32-bit VAX 11/780 processor, about 25\% of all instructions are taken jumps, causing about 20\% of all memory access to be wasted[3]. With the MLT architecture, no memory access are wasted (except for computed jumps in the prototype), a significant improvement over normal prefetching.

8.4.2 Processing Speed

Table 8:2 summarizes Sieve and Dhrystone benchmark conditions and results for the prototype, the DEC VAX 11/780, the Motorola 68020 (in an Apple Macintosh II)[31], and the Intel 80386 (in an IBM PS/2 Model 80)[31]. For fair comparison between systems, the Sieve and Dhrystone results are also scaled to a processor clock speed of 5 MHz.

The scaled results for the Sieve benchmark indicate that the prototype performs the Sieve at 0.32 times the speed of the M68020, 0.41 times the speed of the 80386 and 0.47 times the speed of the VAX. However, the prototype's relative performance on the Dhrystone is much better, being 0.61, 0.84 and 1.0 times the Dhrystone performance of the
<table>
<thead>
<tr>
<th>Machine</th>
<th>Prototype</th>
<th>DEC VAX 11/780</th>
<th>Macintosh 11</th>
<th>IBM PS/2 Model 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>MPL experimental</td>
<td>BDS C</td>
<td>MPW C for 68020 v 2.0</td>
<td>High C 80386 v 1.3</td>
</tr>
<tr>
<td>Special Notes</td>
<td></td>
<td></td>
<td>automatic reg. var allocation</td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td>VAX 11/780</td>
<td>Motorola 68020</td>
<td>Intel 80386</td>
<td></td>
</tr>
<tr>
<td>Clock speed (MHz)</td>
<td>5</td>
<td>5</td>
<td>15.67</td>
<td>16</td>
</tr>
<tr>
<td>Clocks per mem. access</td>
<td>3</td>
<td>--</td>
<td>2-3 (on chip cache)</td>
<td>2+1 wait</td>
</tr>
<tr>
<td>Sieve loops/second</td>
<td>2.14 (N = 4)</td>
<td>4.57</td>
<td>20.8</td>
<td>16.7</td>
</tr>
<tr>
<td>Dhrystones/second</td>
<td>952 (N = 4)</td>
<td>1560</td>
<td>2061</td>
<td>3626</td>
</tr>
<tr>
<td>Sieve loops/second scaled to 5 MHz clock</td>
<td>2.14</td>
<td>4.57</td>
<td>6.63</td>
<td>5.22</td>
</tr>
<tr>
<td>Dhrystones/second scaled to 5 MHz clock</td>
<td>952</td>
<td>1560</td>
<td>913</td>
<td>1133</td>
</tr>
</tbody>
</table>

Table 8.2: Comparative Processor Performance
VAX, 80286 and M68020, respectively. The difference in relative performance suggests that the other compilers are better at optimizing optimizing the looping constructs and array indexing of the Sieve than the MPL compiler. Their advantage in this respect is reduced by the more complex nature of the Dhrystone benchmark.

Considering that the other processors have 32-bit data paths (allowing them to fetch more than one instruction at once) and optimizing compilers, the prototype performs very well indeed. However, the fact that the prototype’s relative performance varies widely between the Sieve and the Dhrystone indicates that these two benchmarks are perhaps too limited in scope. Further benchmark tests will be required in order to completely gauge the performance of the MLT architecture.

8.4.3 Context Switching Overhead

The prototype switches contexts at microinstruction boundaries, requiring "no time" per context switch. The Motorola 68020 also has support for rapid context switching, in the form of the MOVEM (move multiple) instruction. The time required to save and then re-load the entire register file is approximately 124 clock cycles, not counting time required to locate the new context[23]. For the 16-bit Intel 80286, a context switch is accomplished via the Task Gate mechanism in about 180 clocks[32]. If high speed context switching is required, the prototype has a clear advantage. At 1000 context switches per second and 5 MHz clock rates, > 2.5% of the M68020’s time and > 3.6% of the Intel 80286’s time would be consumed by context switching. The
prototype could switch contexts at every instruction boundary (700,000 – 800,000 times per second), with no performance reduction at all.

8.5 Performance Summary

The prototype processor performs very much as expected. For both the Sieve and the Dhrystone benchmarks, processor MBE is very high -- 100% for the Sieve (N > 1) and >96% for the Dhrystone (N > 2). If the effect of the inefficient multiply instruction were eliminated, the Dhrystone MBE performance should reach 100% with N > 2. The fact that processor MBE increases most dramatically as N is increased from 1 to 2 suggests that MLT systems with a small number of contexts may still exhibit very high processor memory bandwidth efficiencies.

The stream execution skew measurements show that the simple rotation and compensation system proposed in Chapter 5 works very well. This indicates that the even distribution of fetching and execution priorities yields an even distribution of processing time. The proposed simple WAIT / AWAKE semaphore system also functions well, although its efficiency is difficult to estimate with the present test system.

Measurements of the dynamic frequency of taken jumps during execution suggest that a similar computer with simple prefetching would be at least 14% slower and less efficient than the prototype. The fact that the prototype's performance is comparable to several modern 32-bit computers despite compiler quality and word size differences suggests that Microcode Level Timeslicing is an effective processor accelerator.
CHAPTER 9

SUMMARY AND CONCLUSIONS

9.1 Summary

The von Neumann computer model, upon which most of our modern computers are based, is lacking in both parallelism and support for multitasking. One convenient measure of processor parallelism is the degree to which a processor utilizes its available memory bandwidth for useful work: its Memory Bandwidth Efficiency (MBE). A processor which exhibits an MBE of 100% is operating as fast as possible for its given memory speed.

Traditional processor accelerators like prefetching and pipelining increase parallelism but suffer from the jump problem, in which a taken jump may cause incorrect prefetching. These accelerators are not well adapted to multitasking, although enhancements like selectable register files may be used to reduce context switching overhead. The pipelined multistream architecture achieves a high degree of parallelism while avoiding the jump problem and supporting efficient context switching. Unfortunately, its performance is load dependent and the architecture is awkward to implement. Furthermore, none of these architectures support efficient task resource sharing.

Microcode Level Timeslicing (MLT) is a multistream processor architecture that achieves very high processor MBE, has no-overhead context switching and provides support for resource sharing. A simple
processor implementing this principle consists of an Execution Unit (EU), a Bus Interface and Instruction unit (BIU) and a Stream Control Unit (SCU). The BIU handles all memory accesses including instruction fetching, while the EU performs all data operations required by each instruction. The state information (registers, flags and so forth) in the EU and BIU are replicated N-fold, and the SCU selects which of the N possible streams will be fetched for by the BIU and which of the streams will be executed by the EU. Because prefetching is carried out only one level deep for each stream, the jump problem may be eliminated by enforcing an execute-to-prefetch delay on each individual stream. Because context switching occurs on clock cycle boundaries, there is no context switching overhead for up to N tasks. Synchronizing operations like semaphore-set-with-wait may be supported in the SCU by including extra status information for each stream.

The hardware and software design of a prototype processor demonstrating the MLT principle is presented. This 16-bit prototype implements MLT for at most 16 streams. The prototype's SCU uses a rotating priority system to ensure even time distribution between running tasks, and also supports a simple WAIT / AWAKE mechanism. Process control and synchronization is implemented in the microcode, allowing complex functions like process creation and semaphore-set-with-wait to be performed with single instructions. A high-level language compiler for the author's Pascal-like language, MPL, is implemented, allowing standard high-level language benchmarks to be run.

Two limited benchmarks, the Sieve of Eratosthenes and the
Dhrystone, are used to evaluate the prototype processor's single and multistream performance. The processor's MBE, processing speed, EU utilization and SCU time distribution are evaluated. As expected, processing MBE and speed increase as more streams are run. For the Sieve benchmark, 100% MBE is achieved for \( N > 1 \) stream. The Dhrystone performance increases more slowly because the benchmark is more computation intensive than the Sieve, but MBE performance in excess of 96% (\( N > 2 \)) is achieved. This MBE is considerably better than that obtainable with conventional prefetching. Furthermore, results suggest that if the prototype's inefficient multiply instruction were improved, Dhrystone performance could reach 100% for \( N > 2 \).

The prototype's performance on the Sieve and Dhrystone benchmarks compares well on a clock-for-clock basis with that of other processors like the VAX 11/780, the Motorola 68020 and the Intel 80386, despite differences in word size and compiler technology.

9.2 Conclusions

Microcode Level Timeslicing is a viable mechanism for improving processor memory bandwidth efficiency and reducing multitasking overhead. The fact that the most MBE increase is achieved between \( N = 1 \) and \( N = 2 \) in the prototype suggests that a processor implemented with as few as 4 streams could achieve great MBE improvement with minimal hardware. MLT's maximum use of available memory bandwidth, efficient context switching, process synchronization support and relatively low complexity may make it the architecture of choice for low-cost
multitasking systems. Because context switching occurs between clock cycles, very fast response times to external events are possible for dedicated service streams. This makes MLT particularly attractive for real-time control applications, where a versatile MLT processor could replace dedicated hardwired controllers.

9.3 Recommendations for Further Work

There remains much that may be done to explore the potentials of the Microcode Level Timeslicing processor architecture. Of particular interest will be to evaluate the overall system performance of an MLT processor. A prototype with full input/output and mass storage capabilities will be required to assess this. Hardware support for complex operations like multiplication should be included. With an improved compiler and with an operating system it will be possible to determine the appropriateness of the proposed task and semaphore handling mechanisms. Methods of exploiting parallelism with the PCALL/PRET (parallel call and return) instructions in high-level language code must also be developed.

If MLT processors are to be used in large computer systems, it is essential that auxiliary context switching techniques be developed. This will allow more processes to run than the number of hardware contexts would otherwise permit. Mechanisms for handling virtual memory must also be investigated. If MLT machines are to be used in cache-based multiprocessors, the cache performance of this architecture must be evaluated. Since several independent instruction streams run
simultaneously, caches with several sets may be required to achieve reasonable cache hits rates. It may also be necessary to restrict the task priority rotation speed.

The structure of the SCU will require optimization if the MLT architecture is to be used for real-time control. Facilities for fast response to external events using dedicated streams must be designed. Also, if only a small number of streams (perhaps 4) is to be implemented, the SCU design may be considerably simplified.

Although the prototype MLT processor was not implemented in a fully pipelined fashion, it is possible to construct a fully pipelined MLT processor. In addition to reducing the jump problem, MLT may be be useful in eliminating operand conflicts within the pipeline.
APPENDIX 1

INSTRUCTION SET SUMMARY
### Basic Data Movement:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>move rs,rd</td>
<td>move word data</td>
<td>----</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>move.l rs,rd</td>
<td>move long-word data</td>
<td>----</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>move #k,rd</td>
<td>move short const word</td>
<td>----</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>move #data,rd</td>
<td>move word data</td>
<td>----</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>move.l #k,rd</td>
<td>move shrt const l-wrd</td>
<td>----</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>move.l #data,rd</td>
<td>move long-word data</td>
<td>----</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>move.b addr,rd</td>
<td>move byte data</td>
<td>----</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>move addr,rd</td>
<td>move word data</td>
<td>----</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>move.l addr,rd</td>
<td>move long-word data</td>
<td>----</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>move.b rs,addr</td>
<td>move byte data</td>
<td>----</td>
<td>8/9</td>
<td>10/13</td>
</tr>
<tr>
<td>move rs,addr</td>
<td>move word data</td>
<td>----</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>move.l rs,addr</td>
<td>move long-word data</td>
<td>----</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>move.b (rs),rd</td>
<td>move byte data</td>
<td>----</td>
<td>5</td>
<td>.7</td>
</tr>
<tr>
<td>move (rsg),rd</td>
<td>move word data</td>
<td>----</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>move.l (rs),rd</td>
<td>move long_word data</td>
<td>----</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

### Special Symbols:
- `addr` = direct address (in extra 16 bit word)
- `base` = base address (in extra 16 bit word)
- `disp` = displacement (in extra 16 bit word)
- `rs` = source register r0..r3
- `rd` = destination register r0..r3
- `sp` = r3
- `cc` = 5-bit condition code (in opcode)
- `#data` = immediate data (in 1 or 2 16 bit data words, lsw first)
- `#k` = 5 bit constant (in opcode)
- `#f` = 6 bit constant (in opcode)
- `list` = 5 bit flag map XSCVZ (in opcode)
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>move.b rd,(rs)</td>
<td>move byte data</td>
<td>----</td>
<td>6/7 8/11</td>
<td>3</td>
</tr>
<tr>
<td>move rs,(rd)</td>
<td>move word data</td>
<td>----</td>
<td>2 4</td>
<td>2</td>
</tr>
<tr>
<td>move.l rs,(rd)</td>
<td>move long_word data</td>
<td>----</td>
<td>5 7</td>
<td>3</td>
</tr>
<tr>
<td>move.b (rs,base),rd</td>
<td>move byte data</td>
<td>----</td>
<td>7 9</td>
<td>3</td>
</tr>
<tr>
<td>move (rs,base),rd</td>
<td>move word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move.l (rs,base),rd</td>
<td>move l-word data</td>
<td>----</td>
<td>5 11</td>
<td>4</td>
</tr>
<tr>
<td>move.b rs,(rd;base)</td>
<td>move byte data</td>
<td>----</td>
<td>8/9 10/13</td>
<td>4</td>
</tr>
<tr>
<td>move rs,(rd;base)</td>
<td>move word data</td>
<td>----</td>
<td>5 7</td>
<td>3</td>
</tr>
<tr>
<td>move.l rs,(rd;base)</td>
<td>move l-word data</td>
<td>----</td>
<td>8 10</td>
<td>4</td>
</tr>
<tr>
<td>move (rs)+,rd</td>
<td>move word data</td>
<td>----</td>
<td>3 5</td>
<td>2</td>
</tr>
<tr>
<td>move.l (rs)+,rd</td>
<td>move long-word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move rs,(rd)+</td>
<td>move word data</td>
<td>----</td>
<td>3 5</td>
<td>2</td>
</tr>
<tr>
<td>move.l rs,(rd)+</td>
<td>move long-word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move -(rs),rd</td>
<td>move word data</td>
<td>----</td>
<td>3 5</td>
<td>2</td>
</tr>
<tr>
<td>move.l -(rs),rd</td>
<td>move long-word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move rs,-(rd)</td>
<td>move word data</td>
<td>----</td>
<td>3 5</td>
<td>2</td>
</tr>
<tr>
<td>move.l rs,-(rd)</td>
<td>move long-word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move (rs)+,(rd)+</td>
<td>move word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move (rs)-,(rd)-</td>
<td>move word data</td>
<td>----</td>
<td>6 8</td>
<td>3</td>
</tr>
<tr>
<td>move fcw,rd</td>
<td>move from flags</td>
<td>----</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>move rs,fcw</td>
<td>move to flags</td>
<td>???</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>move fcw,-(rd)</td>
<td>push flags</td>
<td>----</td>
<td>3 5</td>
<td>2</td>
</tr>
<tr>
<td>move (rs)+,fcw</td>
<td>pop flags</td>
<td>???</td>
<td>3 5</td>
<td>2</td>
</tr>
<tr>
<td>move pc,rd</td>
<td>move from pc</td>
<td>----</td>
<td>2 4</td>
<td>2</td>
</tr>
</tbody>
</table>

*(1 wasted)*
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks min</th>
<th>EU clocks max</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex rs, rd</td>
<td>exchange words</td>
<td>----</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>ex.1 rs, rd</td>
<td>exchange long-words</td>
<td>----</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ex addr, rd</td>
<td>exchange word data</td>
<td>----</td>
<td>9</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>ex.1 addr, rd</td>
<td>exchg. long-word data</td>
<td>----</td>
<td>15</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>ex.1 (rs), rd</td>
<td>exchange words</td>
<td>----</td>
<td>5</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>ex.1 (rs), rd</td>
<td>exchange long-words</td>
<td>----</td>
<td>11</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>pushm ra, rb</td>
<td>push from ra to rb</td>
<td>0--1</td>
<td>6</td>
<td>8</td>
<td>b-a+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+3(b-a)</td>
<td>+3(b-a)</td>
<td></td>
</tr>
<tr>
<td>popm ra, rb</td>
<td>pop from rb to ra</td>
<td>0--1</td>
<td>8</td>
<td>10</td>
<td>a-b+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+3(a-b)</td>
<td>+3(a-b)</td>
<td></td>
</tr>
</tbody>
</table>

**Special Data Movement:**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks min</th>
<th>EU clocks max</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>move ds, rd</td>
<td>move current ds reg.</td>
<td>----</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>move rs, ds</td>
<td>move current ds reg.</td>
<td>----</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>move cs, rd</td>
<td>move current cs reg.</td>
<td>----</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>move xa, rd</td>
<td>move from xa</td>
<td>----</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>move rs, xa</td>
<td>move to xa</td>
<td>----</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>move xs, rd</td>
<td>move from xs</td>
<td>----</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>move rs, xs</td>
<td>move to xs</td>
<td>----</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>move.1 xa, rs</td>
<td>move xsaxa long-word</td>
<td>----</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>move.1 rs, xa</td>
<td>move to xsaxa long_word</td>
<td>----</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>move (xa)+, rd</td>
<td>move extended memory</td>
<td>----</td>
<td>7</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>move rs, (xa)+</td>
<td>move extended memory</td>
<td>----</td>
<td>7</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>movep (xa)+, rd</td>
<td>move from page table</td>
<td>----</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>movep rs, (xa)+</td>
<td>move to page table</td>
<td>----</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Function</td>
<td>Flags affected</td>
<td>EU clocks</td>
<td>Memory Accesses</td>
<td>Flags affected</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>moves (xa)+,rd</td>
<td>move from segment tab.</td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>moves rs,(xa)+</td>
<td>move to segment table</td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Arithmetic Operations:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type of Operation</th>
<th>Flags</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>add rs,rd</td>
<td>add words</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>add.1 rs,rd</td>
<td>add long-words</td>
<td>SCVZ</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add #k,rd</td>
<td>add words</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>add.1 #k,rd</td>
<td>add long-words</td>
<td>SCVZ</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>add #data,rd</td>
<td>add words</td>
<td>SCVZ</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>add.1 #data,rd</td>
<td>add long-words</td>
<td>SCVZ</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>adc rs,rd</td>
<td>add words with cy</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>adc.1 rs,rd</td>
<td>add l-words with cy</td>
<td>SCVZ</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>adc #k,rd</td>
<td>add words with cy</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>adc.1 #k,rd</td>
<td>add l-words with cy</td>
<td>SCVZ</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>adc #data,rd</td>
<td>add words with carry</td>
<td>SCVZ</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>adc.1 #data,rd</td>
<td>add long-words with cy</td>
<td>SCVZ</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>sub rs,rd</td>
<td>subtract words</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sub.1 rs,rd</td>
<td>subtract long-words</td>
<td>SCVZ</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>sub #k,rd</td>
<td>subtract words</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sub.1 #k,rd</td>
<td>subtract long-words</td>
<td>SCVZ</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>sub #data,rd</td>
<td>subtract words</td>
<td>SCVZ</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>sub.1 #data,rd</td>
<td>subtract long-word</td>
<td>SCVZ</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>sbc rs,rd</td>
<td>sub words with cy</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Function</td>
<td>Flags affected</td>
<td>EU clocks</td>
<td>Memory Accesses</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>sbc.1 rs, rd</td>
<td>sub l-words with cy</td>
<td>SCVZ</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sbc #k, rd</td>
<td>sub words with cy</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sbc.1 #k, rd</td>
<td>sub l-words with cy</td>
<td>SCVZ</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sbc #data, rd</td>
<td>sub. words with carry</td>
<td>SCVZ</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sbc.1 #data, rd</td>
<td>sub. l-words with cy</td>
<td>SCVZ</td>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>cp rs, rd</td>
<td>compare words</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cp.1 rs, rd</td>
<td>compare long-words</td>
<td>SCVZ</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cp #k, rd</td>
<td>compare words</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cp.1 #k, rd</td>
<td>compare long-words</td>
<td>SCVZ</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cp #data, rd</td>
<td>compare: form dst-src</td>
<td>SCVZ</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>cp.1 #data, rd</td>
<td>compare long_words</td>
<td>SCVZ</td>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>mul rd</td>
<td>multiply adjacent regs</td>
<td>S00Z</td>
<td>37</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>mul.1 rd</td>
<td>_mul. adj. long regs</td>
<td>S00Z</td>
<td>118</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>mul #data, rd</td>
<td>multiply words</td>
<td>S00Z</td>
<td>38</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mul.1 #data, rd</td>
<td>multiply long_words</td>
<td>S00Z</td>
<td>121</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>neg rd</td>
<td>negate word</td>
<td>SCVZ</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>neg.1 rd</td>
<td>negate long_word</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>inc rd</td>
<td>increment word</td>
<td>SCVZ</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>inc.1 rd</td>
<td>increment long-word</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>inc2 rd</td>
<td>inc word by 2</td>
<td>SCVZ</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>inc2.1 rd</td>
<td>inc long-word by 2</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dec rd</td>
<td>decrement word</td>
<td>SCVZ</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dec.1 rd</td>
<td>decrement long-word</td>
<td>SCVZ</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dec2 rd</td>
<td>dec word by 2</td>
<td>SCVZ</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Function</td>
<td>Flags affected</td>
<td>EU clocks</td>
<td>Memory Accesses</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>dec2.1 rd</td>
<td>dec long-word by 2</td>
<td>SCVZ</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ext rd</td>
<td>sign extend b to w</td>
<td>S--Z</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>ext.1 rd</td>
<td>sign extend w to 1</td>
<td>S--Z</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

**Logical Operations:**

<table>
<thead>
<tr>
<th></th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>and rs,rd</td>
<td>logical AND words</td>
<td>S--Z</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>and #data,rd</td>
<td>logical AND words</td>
<td>S--Z</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>or rs,rd</td>
<td>logical OR words</td>
<td>S--Z</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>or #data,rd</td>
<td>logical OR words</td>
<td>S--Z</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>xor rs,rd</td>
<td>logical XOR words</td>
<td>S--Z</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>xor #data,rd</td>
<td>logical XOR words</td>
<td>S--Z</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>not rd</td>
<td>logical NOT</td>
<td>S--Z</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>tst rd</td>
<td>test word</td>
<td>S--Z</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>tst.1 rd</td>
<td>test long word</td>
<td>S--Z</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>tc cc,rd</td>
<td>test condition code &amp; set rd accordingly</td>
<td>----</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>fset list</td>
<td>set flags by bit map</td>
<td>????</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>fres list</td>
<td>clr flags by bit map</td>
<td>????</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>fcom list</td>
<td>cpl flags by bit map</td>
<td>????</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
### Rotate and Shift Operations:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>rr rd</td>
<td>rotate right word</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rr.l rd</td>
<td>rotate right long-word</td>
<td>SCOZ</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>rrc rd</td>
<td>rr word through cy</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rrc.i rd</td>
<td>rr l-word through cy</td>
<td>SCOZ</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>rl rd</td>
<td>rotate right word</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rl.l rd</td>
<td>rotate right long-word</td>
<td>SCOZ</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>rlc rd</td>
<td>rr word through cy</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rlc.i rd</td>
<td>rr l-word through cy</td>
<td>SCOZ</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>sr rd</td>
<td>shift w right logical</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sr.l rd</td>
<td>shr l-word logical</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sra rd</td>
<td>shr word arithmetic</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sra.l rd</td>
<td>shr l-word arithmetic</td>
<td>SCOZ</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>sl rd</td>
<td>shift word left</td>
<td>SCOZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sl.l rd</td>
<td>shift l-word left</td>
<td>SCOZ</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>swap rd</td>
<td>swap bytes</td>
<td>S--Z</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Input / Output:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>in addr,rd</td>
<td>Input from port</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>out rs,addr</td>
<td>Output to port</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>in (rs),rd</td>
<td>Port input indirect</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>out (rs),rd</td>
<td>Port output indirect</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Function</td>
<td>Flags affected</td>
<td>EU clocks</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------</td>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>Control Transfer:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jp cc,rd</td>
<td>jump indirect</td>
<td>----</td>
<td>3 5</td>
</tr>
<tr>
<td>jp cc,adrs</td>
<td>absolute cond. jump</td>
<td>----</td>
<td>3 5</td>
</tr>
<tr>
<td>jr cc,disp</td>
<td>relative cond. jump</td>
<td>----</td>
<td>6/5 8/7</td>
</tr>
<tr>
<td>djnz rd,adrs</td>
<td>decrement rd and jump</td>
<td>S--Z</td>
<td>6/7 8/9</td>
</tr>
<tr>
<td></td>
<td>if result not zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>djnzr rd,disp</td>
<td>djnz relative</td>
<td>S--Z</td>
<td>6/7 8/9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call rd,adrs</td>
<td>call subroutine using rd as stack pointer</td>
<td>----</td>
<td>6 8</td>
</tr>
<tr>
<td>call rd,adrs</td>
<td>call relative</td>
<td>----</td>
<td>7 9</td>
</tr>
<tr>
<td>ret rd</td>
<td>return from subr</td>
<td>----</td>
<td>3 5</td>
</tr>
<tr>
<td>ret #data,rd</td>
<td>return from subr, with sp adjust</td>
<td>----</td>
<td>6 8</td>
</tr>
<tr>
<td>svc #f</td>
<td>supervisor call</td>
<td>----</td>
<td>10 14</td>
</tr>
<tr>
<td>svfet</td>
<td>return from svc</td>
<td>----</td>
<td>3 5</td>
</tr>
<tr>
<td><strong>Semaphore Handling:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sset (rd)</td>
<td>semaphore tst and set</td>
<td>S--Z</td>
<td>7/6 11/8</td>
</tr>
<tr>
<td></td>
<td>(set/not)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sset adrs</td>
<td>semaphore tst and set</td>
<td>S--Z</td>
<td>10/9 14/11</td>
</tr>
<tr>
<td></td>
<td>(set/not)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssetw (rd)</td>
<td>semaphore tst and set</td>
<td>0--1</td>
<td>9 13</td>
</tr>
<tr>
<td></td>
<td>with wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssetw adrs</td>
<td>semaphore tst and set</td>
<td>0--1</td>
<td>12 16</td>
</tr>
<tr>
<td></td>
<td>with wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sclr (rd)</td>
<td>semaphore clear</td>
<td>----</td>
<td>3 5</td>
</tr>
<tr>
<td>sclr adrs</td>
<td>semaphore clear</td>
<td>----</td>
<td>6 8</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Function</td>
<td>Flags affected</td>
<td>EU clocks</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------</td>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>sleep</td>
<td>enter WAIT mode for testing</td>
<td>----</td>
<td>11+</td>
</tr>
<tr>
<td>signal</td>
<td>signal AWAKE for testing</td>
<td>----</td>
<td>1</td>
</tr>
</tbody>
</table>

**Task Control:**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Flags affected</th>
<th>EU clocks</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>create rs</td>
<td>create process cr: 0000 r[s] = ds fail: 0100 (create/fail)</td>
<td>8/3 8/3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>kill rd</td>
<td>kill process rd</td>
<td>----</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>pc all rs</td>
<td>parallel call r[s] = new sp r[s+1] = new pc</td>
<td>13/8 15/10 2/3</td>
<td>2/3</td>
<td></td>
</tr>
<tr>
<td>pret</td>
<td>parallel return</td>
<td>unde</td>
<td>6/8</td>
<td>2/3</td>
</tr>
<tr>
<td>halt</td>
<td>halt current process</td>
<td>unde</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>monitor</td>
<td>enter microcode monitor</td>
<td>????</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>l nop</td>
<td>long no-op for test purposes</td>
<td>----</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
APPENDIX 2

REPRESENTATIVE MICROCODE
Add Words
add Rs,Rd

\[ a[0] := r[ir2] \]
\[ r[irl] := r[irl] + a[0] / \]
flags arith
flags arith /
eoi / jump idle

Assembly mnemonic

Encoding

100000ssssssdddd

\[ t = 2, m = 0 \]
Flags: SCVZ

\[ \text{Flags affected} \]
\[ (- \Rightarrow \text{no change}) \]

Add Long Words
add.1 Rs,Rd

\[ a[0] := r[ir2] \]
\[ r[irl] := r[irl] + a[0] / \text{flags arith} \]
\[ a[0] := r[ir2+1] \]
\[ r[irl+1] := r[irl+1] + a[0] + cy / \]
flags arith_extend /
eoi / jump idle

A2a: Representative Microcode
Move Byte Indirect To Register

move.b (Rs),Rd	011000ssssssdddd	$t = 5$, $m = 1$

Flags: ----

a[0] := fcw
mar := shr r[ir2] /
blu read /
flags shift
fcw := a[0]
r[ir1] := mdr /
jump ugt,..+2

r[ir1] := r[ir1] & c[00ff] /
eoi / jump idle

r[ir1] := swap r[ir1] & c[ff00] /
eoi / jump idle

; just mask for even addresses
; mask and swap for odd addresses

Move Byte Indirect From Register

move.b Rs,(Rd)	011001ssssssdddd
t = 6/7, $m = 2$
even/offd

Flags: ----

a[0] := fcw 
mar := shr r[ir1] /
blu read /
flags shift
fcw := a[0]
a[0] := mdr /
jump ugt,..+3

; even addresses
mar := shr r[ir1] / blu write
mdr := merge a[0], r[ir2] /
eoi / jump idle

; initiate the write cycle
; write out the modified data
; and done.

; odd addresses
a[0] := swap a[0]
m = shr r[ir1] / blu write
mdr := swap merge a[0], r[ir2] /
eoi / jump idle

; must pre-swap
; initiate the write cycle
; write out the modified data
; and done.

A2b: Representative Microcode
Push Multiple Registers

pushm #k,Rs

011110kkkkkssss

(Push from Rs..Rs+k on the stack)

a[0] := ir2 /
flags logical
p := ir1
r[31] := shr r[31]

; Transfe r loop:

r[31],mar := r[31] - 0 - 1 /
bu write
mdr := r[p] /
inc p /
jum p eq,.+2
a[0] := a[0] - 0 - 1
flags logical /
jum p -.2

; Finalize:

r[31] := shr r[31] /
eoi / jump idle

; ; ;

Rop Multiple Registers

; popm #k,Rd

011111kkkkkddd

(Pop Rs..Rs-k from the stack)

a[0] := ir2 / flags logical
p := ir1
r[31] := shr r[31]
r[31] := r[31] - 0 - 1
r[31],mar := r[31] + 0 + 1 / biu read
a[0] := a[0] - 0 - 1 / flags logical
.r[p] := mdr / dec p / jum p neq,.-2

r[31] := shr r[31] + 0 + 1 / eoi / jump idle

; ; ;

A2c: Representative Microcode
Decrement Register and Jump if Non-zero:

djnz rd, adrs 1111010010ddddd  

Note: adrs is the WORD address
t = 6/6, m = 1

taken/not

Flags: S--Z

\[
\begin{align*}
\text{Flags} & : S--Z \\

t & : 6/6, m = 1 \\
\text{Taken/not} & : \\
\text{A2d: Representative Microcode} \\
\end{align*}
\]
; Semaphore Set with Wait
; ssetw X 111101101111111

biu opfetch
no_op
a[0] := mdr

; Main test-and-set-with-wait loop:
; swalp:
mar := rr a[0] / biu read / stream wait
p := a[csn]
:= mdr / flags logical
a[1] := seg[p]
:= swap a[1] / flags logical /
    jump eq, swdone
no_op
jump neg,stop
eoi / jump swalp

swdone:
mar := rr a[0] / biu write / stream run
mdr := 0 -- 1
:= 0 ++ 1 / flags logical /
    eoi / jump idle

; Semaphore Clear
; sclr X 111110111101111

biu opfetch / stream signal
no_op
a[0] := mdr
mar := rr a[0] / biu write
mdr := 0 / eoi / jump idle

; t = 6, m = 1
; Flags: ----
; fetch the semaphore's address
; and signal waiting tasks
; store the address in a[0]
; write out 0
; done.
Create New Process

create rs 111111110ssss s

create/fail

Flags: 0000 if success
else 0100 if failed (cy set)

reg file contains:
r[s+3] = data segment
r[s+2] = stack address (BYTE address)
r[stl] = code segment
r[as] = code start address (WORD address)

fcw := 0 ; clear all flags

Set up data in the common scratch pad:

c[0] := r[irl] /
jump nfree,crfail

; move code start and
; skip if no streams free

; move code segment

; and set creation mode

; move stack addr

; move data segment

; move creator #

; done!!

c[4] := r[irl]
r[irl] := c[0]
eoi / jump idle

Failure -- no streams free:

; Failure -- no streams free:

crfail: fcw := shl c[0002] /
stream run / jump idle ; cy set if failure
; "stream run" resets the
; creation mode.

Halt current process

halt 111111111xxxxx

stop:
stream halt ; enter HALT state
jump Idle ; done.

A2f: Representative Microcode
Kill Selected Process

\[ \text{kill } rd \quad \text{flags: ----} \quad t = 7, \ m = 0 \]

\text{Note: -- all semaphore operations should check their cs regs each test cycle. $ff => must halt -- the process might not die for up to 1 priority rotation cycle.}

\begin{align*}
a[0] & := r[i1r] / \text{stream signal} \quad \text{save } r[i1r] \text{ in } a[0] \\
r[i1r] & := c[\_0010] - 0 - 1 \quad \text{stream # mask = $0f} \\
p, r[i1r] & := r[i1r] \& a[0] \quad r[i1r] = 0..15; p' => \text{victim CS} \\
seg[p] & := c[\_00ff] \quad \text{change the victim CS} \\
p := r[i1r]; c[\_0010] & \quad p => \text{victim DS} \\
seg[p] & := c[\_00ff] \quad \text{change the victim DS} \\
r[i1r] & := a[0] / \quad \text{restore } r[i1r] \\
eoi / \text{jump idle} & \quad \text{and done.}
\end{align*}

A2g: Representative Microcode
APPENDIX 3

THE DHRYSTONE BENCHMARK
module dhrystone;
{
  The Dhrystone 2.0 Benchmark -- Adapted from the Pascal Version
  Original by R.P. Weicker, March 3, 1988
}

#include lib:stdio.def { standard i/o library }
#include lib:utility.def { utility i/o library }

( Global type definitions )

type
  boolean := byte;
  char := byte;

Enumeration := (Ident1, Ident2, Ident3, Ident4, Ident5);

OneToThirty := integer;
OneToFifty := integer;
CapitalLetter := byte;
String30 := string[30];
Array1DimInteger := array[1..50] of integer;
Array2DimInteger := array[1..50,1..50] of integer;

RecordType := record
  PointerComp : @;
  Discr : Enumeration;
  EnumComp : Enumeration;
  IntComp : OneToFifty;
  StringComp : String30
endrec;

RecordPointer := @RecordType;

var
  [ Ada version: Variables local in Proc_0 ]

Int1Glob,
Int2Glob,
Int3Glob := OneToFifty;
CharIndex := char;
EnumGlob : Enumeration;
String1Glob,
String2Glob : String30;

{ Ada version: Variables global in Pack_1 }

PointerGlob,
NextPointerGlob : RecordPointer;

IntGlob : integer;
BoolGlob : boolean;
Char1Glob,
Char2Glob : char;
Array1Glob : Array1DimInteger;
Array2Glob : Array2DimInteger;

{ Variables for Measurement }

LineBuf : string[22];
NumberOfRuns,
RunIndex : integer;
TicCount : long_int;
Dhrystones : integer;

const
TicsPerSecond := 100;

{ End of Variable for Measurement }

function Func1( Char1ParVal, Char2ParVal : CapitalLetter) : Enumeration;
{ Executed 3 times, always returns Ident1 }
var
Char1Loc, Char2Loc : CapitalLetter;
begin
Char1Loc := Char1ParVal;
Char2Loc := Char1Loc;
if Char2Loc <> Char2ParVal then
  { executed }
  Func1 := Ident1
else
  { not executed }
  Char1Glob := Char1Loc;
  Func1 := Ident2;
function Func2( String1ParRef, String2ParRef : @String30 ) : boolean;
{ Executed once, returns false }  
var
  IntLoc : OneToThirty;
  CharLoc : CapitalLetter;
begin
  IntLoc := 2;
  while IntLoc <= 2 do
    { loop body executed once }
    if Func1(@String1ParRef[IntLoc], @String2ParRef[IntLoc+1]) = Ident1 then
      { Executed }
      CharLoc := 'A';
      IntLoc := IntLoc + 1
    endif
  endwh;

  if (CharLoc >='W') and (CharLoc < 'Z') then
    { not executed }
    Func2 := true
  else
    { executed }
    if @String1ParRef > String2ParRef then
      { not executed }
      IntLoc := IntLoc + 7;
      IntGlob := Intloc;
      Func2 := true
    else
      { executed },
      Func2 := false
    endif
  endif
end;  { Func2 }

function Func3( EnumParVal : Enumeration) : boolean;
{ Execute once, returns true }
( EnumParVal := Ident3 )
var
  EnumLoc : Enumeration;
begin
EnumLoc := EnumParVal;
if EnumLoc = Ident3 then
  { executed }
  Func3 := true
endif
end;

procedure Proc6( EnumParVal : Enumeration; EnumParRef : @Enumeration );
{ Executed once }
{ EnumParVal = Ident3, EnumParRef becomes Ident2 }
begin
  @EnumParRef := EnumParVal;
  if not Func3(EnumParVal) then
    { not executed }
    @EnumParRef := Ident4
  endif;
  case EnumParVal
  of Ident1:
     @EnumParRef := Ident1
  of Ident2:
    if IntGlob > 100 then
      @EnumParRef := Ident1
    else
      @EnumParRef := Ident4
    endif
  of Ident3:
    @EnumParRef := Ident2 { executed }
  of Ident4:
  of Ident5:
    @EnumParRef := Ident5
  endcase
end;

procedure Proc7( Int1ParVal, Int2ParVal : OneToFifty;
                 IntParRef : @OneToFifty );
{ Executed three times }
var
  IntLoc : OneToFifty;
begin
  IntLoc := Int1ParVal + 2;
  @IntParRef := Int2ParVal + IntLoc
end;
procedure Proc3( PointerParRef : @RecordPointer );
{ executed once }
{ PointerParRef becomes PointerGlob }
begin
    if mknz(PointerGlob) <> 0 then
        { executed }
        PointerParRef := PointerGlob.PointerComp
    else
        { not executed }
        IntGlob := 100;
    endif;
    Proc7(10, IntGlob, ~PointerGlob.IntComp)
end;

procedure Proc1( PointerParVal : RecordPointer );
{ executed once }
{ PointerParVal = PointerGlob }
var
    WithPtr : RecordPointer; { for WITH simulation }
begin
    WithPtr := @(@PointerParVal.PointerComp);

        @(@PointerParVal.PointerComp) := @PointerGlob;
        PointerParVal.IntComp := 5;
        WithPtr.IntComp := @PointerParVal.IntComp;
        WithPtr.PointerComp := @PointerParVal.PointerComp;
        Proc3(@mkptr(mknz(~WithPtr.PointerComp)))
        if WithPtr.Discr = Ident1 then
            { executed }
            WithPtr.Intcomp := 6;
            Proc6(@PointerParVal.EnumComp, ~WithPtr.EnumComp);
            WithPtr.PointerComp := @PointerGlob.PointerComp;
            Proc7(@WithPtr.IntComp, 10, ~WithPtr.IntComp)
        else
            { not executed }
            @PointerParVal := @(@PointerParVal.PointerComp)
        endif
end;
procedure Proc2( IntParRef : @OneToFifty );
{ executed once }
{ IntParRef = 3, becomes 7 }
var
  IntLoc : OneToFifty;
  EnumLoc : Enumeration;
begin
  IntLoc := @IntParRef + 10;
  repeat
    { executed once }
    if Char1Glob = 'A' then
      { executed }
      IntLoc := IntLoc - 1;
      @IntParRef := IntLoc - IntGlob;
      EnumLoc := Ident1
      endif
    until EnumLoc = Ident1; { true }
end;

procedure Proc4();
{ Executed once }
var
  BoolLoc : boolean;
begin
  BoolLoc := Char1Glob = 'A';
  BoolGlob := BoolLoc or BoolGlob;
  Char2Glob := 'B'
end;

procedure Proc5();
{ Execute once }.
begin
  Char1Glob := 'A';
  BoolGlob := false
end;

procedure Proc8( Array1ParRef : @Array1DimInteger;
  Array2ParRef : @Array2DimInteger;


```

    Int1ParVal, Int2ParVal : integer;

    ( Executed once )
    ( Int1ParVal = 3, Int2ParVal = 7 )

    var
        IntIndex,
        IntLoc : OneToFifty;

    begin
        IntLoc := Int1ParVal + 5;
        @Array1ParRef[IntLoc] := Int2ParVal;
        @Array1ParRef[IntLoc+1] := @Array1ParRef[IntLoc];
        @Array1ParRef[IntLoc+30] := IntLoc;
        for IntIndex := IntLoc to IntLoc + 1 do
            @Array2ParRef[IntLoc, IntIndex] := IntLoc
        endfor;
        @Array2ParRef[IntLoc, IntLoc-1] := @Array2ParRef[IntLoc, IntLoc-1] + 1;
        @Array2ParRef[IntLoc+20, IntLoc] := @Array1ParRef[IntLoc];
        IntGlob := 5
    end;

    function chr_2_s( c : char ) : string[1];
    ( For displaying characters )
    begin
        chr_2_s[0] := 1;
        chr_2_s[1] := c
    end;

    begin
        (_Main Program )

        ( Initializations: )

        NextPointerGlob := alloc(sizeof(RecordType));
        PointerGlob    := alloc(sizeof(RecordType));

        @PointerGlob.PointerComp := NextPointerGlob;
        @PointerGlob.Discr := Ident1;
        @PointerGlob.EnumComp := Ident3;
        @PointerGlob.IntComp := 40;
        @PointerGlob.StringComp := "DHRYSTONE PROGRAM, SOME STRING";
        StringGlob := "DHRYSTONE PROGRAM, 1'ST STRING";
        Array2Glob[8,7] := 10;
```
print(newln,"Dhrystone Benchmark, Version MPL / 2.0", newln, newln);
print(newln,"Please give the number of runs through the benchmark: ");
readln(LineBuf,10,md_norm);
NumberOfRuns := val(LineBuf);

print(newln,newln,"Execution starts.", dec(NumberOfRuns),
" runs through Dhrystone",newln);

{ Start Timer }
TicCount := gticks();

for RunIndex := 1 to NumberOfRuns do
  Proc5();
  Proc4();
  Int1Glob := 2;
  Int2Glob := 3;
  String2Glob := "DHRYSTONE PROGRAM, 2'ND STRING";
  EnumGlob := Ident2;
  BoolGlob := not Func2(String1Glob, String2Glob);
    { BoolGlob is true }
  while Int1Glob < Int2Glob do
    { loop body executed once }
    Int3Glob := 5 * Int1Glob - Int2Glob;
    Proc7(Int1Glob,Int2Glob,Int3Glob);
    Int1Glob := Int1Glob + 1
  endwh;
  Proc8(Array1Glob, Array2Glob, Int1Glob, Int3Glob);
  Proc1(PointerGlob);
  for CharIndex := 'A' to Char2Glob do
    { loop body executed twice }
    if EnumGlob = Func1(CharIndex, 'C') then
      { not executed }
      Proc6(Ident1, EnumGlob);
      String2Glob := "DHRYSTONE PROGRAM, 3'RD STRING";
      Int2Glob := RunIndex;
      IntGlob := RunIndex;
    endif
  endfor;
  Int2Glob := Int2Glob * Int1Glob;
  Int1Glob := Int2Glob div Int3Glob;
  Int2Glob := 7 * (Int2Glob - Int3Glob) + Int1Glob;
  Proc2(Int1Glob)
endfor;

{ Stop Timer }
TicCount := gticks() - TicCount; { compute elapsed time }
print(newln,"Execution ends",newln,newln);
print("Final values of the variables used in the benchmark:",newln,
    newln);

print("IntGlob:
    should be: ",dec(IntGlob),newln); 5",newln);

print("BoolGlob:
    if BoolGlob then print("TRUE",newln) else print("FALSE",newln) endif;
    TRUE",newln);

print("Char1Glob:
    print(" should be:

print("Char2Glob:
    print(" should be:

print("Array1Glob[8]:
    print(" should be:

print("Array2Glob[8,7]:
    print(" should be:

print("@PointerGlob.Discr:
    print(" should be:

print("@PointerGlob.EnumComp:
    print(" should be:

print("@PointerGlob.IntComp:
    newln);
    print(" should be:

print("@PointerGlob.StringComp:
    print(" should be:

print("@NextPointerGlob.Discr:
    newln);
    print(" should be:

print("@NextPointerGlob.EnumComp:
    newln);
    print(" should be:

print("@NextPointerGlob.IntComp:
    newln);
    print(" should be:

print("@NextPointerGlob.StringComp,newln);
"DHRYSTONE PROGRAM, SOME STRING", newln);

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:

print(" should be:
print("@NextPointerGlob.StringComp:
    newln);
print("    should be:
    newln);

print("Int1Glob:
print("    should be:

print("Int2Glob:
print("    should be:

print("Int3Glob:
print("    should be:

print("EnumGlob:
print("    should be:

print("String1Glob:
print("    should be:
    newln);

print("String2Glob:
print("    should be:
    newln,newln);

print("Ticks taken:",idec(TicCount),newln);

Dhrystones := lowd( (long(NumberOfRuns) * long(TicsPerSecond))
                 div TicCount );

print("Dhrystones:",dec(Dhrystones),newln,newln);

end

endmodule
APPENDIX 4

RESULTS
<table>
<thead>
<tr>
<th>N</th>
<th># of iter</th>
<th>Total Time</th>
<th>Iter /Sec</th>
<th>MIPS</th>
<th>MBE(%)</th>
<th>EU Util(%)</th>
<th>EU Wait(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>11.89</td>
<td>1.68</td>
<td>0.561</td>
<td>78.4</td>
<td>47.2</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>9.38</td>
<td>2.13</td>
<td>0.707</td>
<td>100.0</td>
<td>60.6</td>
<td>10.0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9.80</td>
<td>2.14</td>
<td>0.707</td>
<td>100.0</td>
<td>60.2</td>
<td>10.1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9.33</td>
<td>2.14</td>
<td>0.695</td>
<td>100.0</td>
<td>60.4</td>
<td>10.1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>14.00</td>
<td>2.14</td>
<td>0.697</td>
<td>100.0</td>
<td>59.8</td>
<td>9.8</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>18.67</td>
<td>2.14</td>
<td>0.702</td>
<td>100.0</td>
<td>60.6</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Frequency of jump / call / return instructions: 22.3%
Frequency of taken jump / call / returns : 12.1%

Operating Conditions:
System Clock: 5 MHz
Priority Rotation: 10mS period

A4a: Sieve of Eratosthenes Results
<table>
<thead>
<tr>
<th>N</th>
<th># of iter</th>
<th>Total Time</th>
<th>Dhryst /Sec</th>
<th>MIPS</th>
<th>MBE(%)</th>
<th>EU Util(%)</th>
<th>EU Wait(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5040</td>
<td>7.12</td>
<td>707</td>
<td>0.610</td>
<td>71.7</td>
<td>47.4</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>2520</td>
<td>5.52</td>
<td>913</td>
<td>0.798</td>
<td>93.8</td>
<td>62.0</td>
<td>9.12</td>
</tr>
<tr>
<td>3</td>
<td>1680</td>
<td>5.31</td>
<td>949</td>
<td>0.818</td>
<td>96.2</td>
<td>63.7</td>
<td>8.50</td>
</tr>
<tr>
<td>4</td>
<td>1260</td>
<td>5.29</td>
<td>952</td>
<td>0.821</td>
<td>96.5</td>
<td>63.9</td>
<td>8.52</td>
</tr>
<tr>
<td>5</td>
<td>1008</td>
<td>5.27</td>
<td>956</td>
<td>0.824</td>
<td>96.8</td>
<td>64.1</td>
<td>8.54</td>
</tr>
<tr>
<td>6</td>
<td>840</td>
<td>5.26</td>
<td>958</td>
<td>0.826</td>
<td>97.2</td>
<td>64.3</td>
<td>8.56</td>
</tr>
<tr>
<td>7</td>
<td>720</td>
<td>5.24</td>
<td>961</td>
<td>0.829</td>
<td>97.6</td>
<td>64.5</td>
<td>8.58</td>
</tr>
<tr>
<td>8</td>
<td>630</td>
<td>5.22</td>
<td>965</td>
<td>0.832</td>
<td>97.8</td>
<td>64.8</td>
<td>8.64</td>
</tr>
</tbody>
</table>

Frequency of jump / call / return instructions: 23.4%
Frequency of taken jump / call / return instructions: 16.6%

Operating Conditions:
System Clock: 5 MHz
Priority Rotation: 10mS period
<table>
<thead>
<tr>
<th>N</th>
<th># of iter</th>
<th>Rotation Compensation Enabled</th>
<th>Rotation Compensation Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total Time</td>
<td>Iter /Sec</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>59.43</td>
<td>1.68</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>46.73</td>
<td>2.14</td>
</tr>
<tr>
<td>3</td>
<td>35</td>
<td>49.00</td>
<td>2.14</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>46.66</td>
<td>2.14</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>46.67</td>
<td>2.14</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>50.40</td>
<td>2.14</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>52.26</td>
<td>2.14</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>52.26</td>
<td>2.14</td>
</tr>
</tbody>
</table>

Operating Conditions:
System Clock: 5 MHz
Priority Rotation: 50mS period

A4c: Sieve-Execution Skew Test
ADDENDUM

OTHER RELATED ARCHITECTURES

After the completion and submission of this thesis, references to three related multistream processor architectures were discovered in the literature [33,34,35].

The Lincoln TX-2 Computer[33] utilizes an efficient context switching mechanism that is similar to the prototype's. The system dedicates one context to each family of input/output devices and one context to the machine's main program. Device service request signals cause the machine's centralized sequence selection logic to select device service routines for execution. Because context switching is achieved by register bank switching, very low latency device service is achieved.

The Xerox Alto[34] and the Xerox Dorado[35] processors achieve efficient input/output service in a similar fashion. However, whereas context switching in the TX-2 occurs between instructions, context switching in the Alto and Dorado occurs on microinstruction boundaries. In each of these processors, the sole purpose of this rapid interrupt mechanism is to improve CPU handled I/O performance and thus reduce system cost.

There are several differences between these systems and Microcode Level Timeslicing (MLT). None of the three systems perform multistream prefetching: the TX-2 and the Alto do not prefetch at all,
while the Dorado prefetches only for the main task. No attempt is made to circumvent the jump problem in any of the systems. Furthermore, the stream selection mechanisms in all three processors are very limited in capability, having no support for interprocess communication and fixed stream priority selection. Because of these limitations, the processors may be only loosely classed as MIMD machines. Rather than supporting true hardware assisted multitasking, they support a form of hardware enhanced interrupt.

Microcode Level Timeslicing goes beyond this concept by using multistream horizontal prefetching to eliminate the jump problem and to improve processor memory bandwidth efficiency. True multitasking is supported in hardware by the rotateable priority Stream Control Unit. Hence the Microcode Level Timesliced architecture is in fact much more general and more versatile than these other processor architectures.
REFERENCES


