

INTERNSHIP EXPERIENCE AT
TEXAS INSTRUMENTS

The Internship Report

by

KERRY CLOYCE GLOVER

Submitted to the College of Engineering
of Texas A&M University
in partial fulfillment
for the degree of

DOCTOR OF ENGINEERING

May 1982

Major Subject: Electrical Engineering


INTERNSHIP EXPERIENCE AT
TEXAS INSTRUMENTS

The Internship Report


by

KERRY CLOYCE GLOVER


Approved as to style and content by:




Dr. V. T. Rhyne
(Chairman of Committee)




Dr. C. A. Rodenberger
(Committee Member)




Dr. W. B. Jones
(Head of Department)



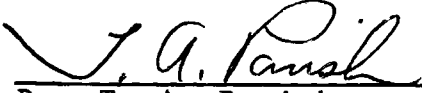
Dr. N. C. Griswold
(D.E. Coordinator)



Dr. N. R. Strader
(Committee Member)



Dr. R. Ohlendorf
(Internship Supervisor)



Dr. T. A. Parish
(D.E. College Rep.)

Dr. J. R. Boone
(Graduate College Rep.)

MAY 1982

ABSTRACT

Internship Experience with Texas Instruments
Kerry Cloyce Glover, B.S., Texas A&M University
M.S., Texas A&M University
Chairman of Advisory Committee: Dr. V.T. Rhyne

This report presents a survey of the author's internship experience with Texas Instruments from November 1980 to November 1981. The internship was spent in the Advanced Research and Development Division of the Digital Systems Group. The report's intent is to demonstrate that this experience fulfills the requirements of the Doctor of Engineering internship.

The author's internship activities involved the design of a special purpose processor. The internship was divided into two parts during which different versions of the processor were developed. The first design was of a slow processor which was to be simple to build and debug. The second design was a high performance, and therefore complex, processor. Several areas of expertise were gained during the internship including simulation, caching techniques and pipelining. Communication skills were developed because of the interaction with software designers.

TABLE OF CONTENTS

	PAGE
INTERNSHIP REPORT.....	1
Introduction.....	1
PROJECT OVERVIEW.....	4
GENERAL PURPOSE PROCESSOR.....	16
Software Defined Architecture.....	16
Architectural Implications.....	20
Processor Evolution.....	23
Microcode.....	27
ALL Field.....	29
Specialized Fields.....	31
Detailed Design.....	34
Arithmetic and Logical Unit (ALU).....	34
Large Register File (LRF).....	39
Sequencer (SEQ).....	41
Testability and Debug Features.....	43
Simulation.....	47
BASIC Simulation.....	47
HDL Simulation.....	50
Communications.....	53
Change in Direction.....	54

FAST PROCESSOR.....56

 Objectives.....56

 Conventional Caching Methods.....57

 Cache Evolution.....61

 Pipeline Evolution.....65

 Performance Improvements.....68

 Detailed Design.....72

 Fifo Buffer.....73

 Operand Specifier Decoder.....76

 Address Developer.....80

 End of the Pipe.....81

 Simulation.....86

 Project Termination.....89

SUMMARY.....91

BIBLIOGRAPHY.....95

APPENDICES..... 96

VITA.....161

LIST OF FIGURES

FIGURES		PAGE
1	Hardware Speed-Cost Curve.....	2
2	System Configuration.....	4
3A	Long Narrow/Slow Machine.....	6
3B	Short Wide/Fast Machine.....	6
4	Project Schedule.....	7
5	Microcode Processor W/Test Features.....	8
6	Debug Configuration.....	9
7	HDL Block Orientation.....	11
8	Slow Processor Block Diagram.....	12
9	Fast Processor Block Diagram.....	14
10	Software/Hardware Interface.....	17
11	Software/Hardware Cost Vs Time Curve.....	18
12	Instruction Stream.....	21
13	Hardware Implemented Operating System.....	22
14	Von Neumann Architecture.....	23
15	Combined Address/Data Buss.....	24
16	Slow Processor Block Diagram.....	26
17	Microcode Word Bit Field.....	29
18	Arithmetic and Logical Unit.....	35
19	Byte Rotater/Selector.....	37
20	Large Register File.....	40

FIGURE		PAGE
21	Sequencer.....	42
22	Test and Debug Data Paths.....	44
23	Sample Screen Display.....	48
24	Hardware Speed-Cost Curve.....	56
25	Workspace Cache.....	58
26	Content Addressable Memory.....	59
27	Associative Cache.....	60
28	Cache Evolution I.....	61
29	Cache Evolution II.....	62
30	Cache Evolution III.....	63
31	Final Cache Configuration.....	64
32	Address Translation.....	65
33	Pipeline.....	67
34	Pipeline Improvements/Calls.....	70
35	Pipeline Improvements/Jumps.....	72
36	FIFO.....	73
37	Address Decoder.....	76
38	Displacement Length Decoding.....	79
39	Address Developer.....	80
40	Fast ALU.....	82
41	Memory Data Paths.....	84
42	Processor Block Diagram.....	85

LIST OF APPEDICES

APPENDIX	PAGE
A	The Internship Objectives.....97
B	Definition of Microcode.....104
C	Sample Microcode Routine.....114
D	Processor/Sequencer Interface.....118
E	Debug Interface Signals.....120
F	Software Debug Routines.....125
G	PASCAL Emulator.....127
H	HDL Simulation of MEM.....141
J	Cache Placement Study.....149
K	Pipeline Data Flow Study.....152
L	Displacement Decode Table.....154
M	HDL Simulation of FIFO.....159

INTERNSHIP REPORT

Introduction

Working on a team effort to design and develop a new computer system is an exciting and challenging opportunity. Such has been my intern experience with Texas Instruments for the past year. My assignment was to design the Central Processing Unit (CPU) of an experimental computer system. This required close work with the research group which was defining the architecture. The successful implementation of the system would require a full understanding of computer hardware, software and system architecture.

The initial objectives of the project are listed in Appendix A. Since this list of objectives was submitted, many things have changed on the project. Many of the objectives were not attainable, however, additional goals were achieved.

Throughout the internship project, the major goal has been to design a processor. During that time, performance requirements for the processor have been changed. The first processor was a low-end implementation of the architecture. This had several

attributes as discussed in the objectives. These were to develop rapidly a machine which had built-in debug and test features, allowing it to be easily microcoded. The goals changed half-way through the project. The new goal was to develop a high performance/high speed version of the processor.

The two versions of the processor were of two speed/performance ranges. A relationship exists between hardware complexity and speed. This project started at one point on that curve (A) and latter was redirected to another point on that curve (B). As the speed of the processor increased the complexity also increased.

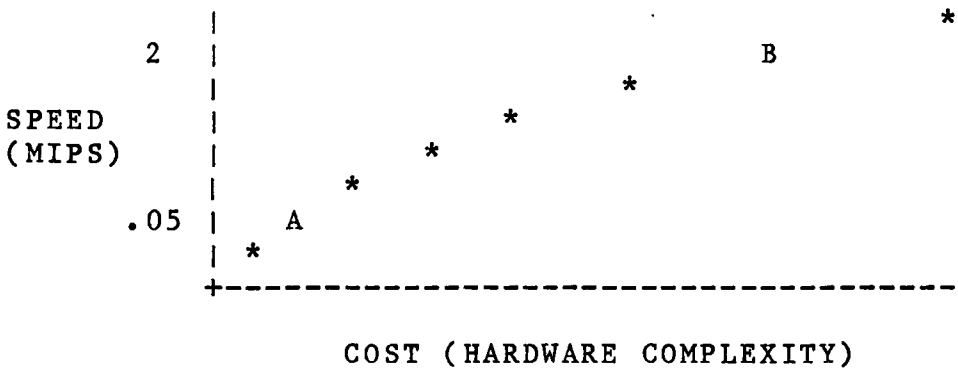


FIGURE 1 - HARDWARE SPEED-COST CURVE

This report will be divided into three sections. The first section will be a short overview of the entire project. This is a condensed form of the report which will familiarize the reader with the project and the various areas which were investigated. The second section will discuss a slow machine. It explains details about the data paths, microcode, and debug features which were designed. The third section will describe several aspects of high performance design of computers and how it applies to this project in developing a fast machine.

PROJECT OVERVIEW

The first several months on the project were spent studying a research oriented computer architecture. This architecture required that several tasks traditionally done in software to be implemented in hardware. Understanding and envisioning an implementation for this experimental machine proved to be a learning experience.

The project had been divided into several different hardware implementation areas. These were the processor, memory management and bus interface. The processor section was developed during the internship. The processor interfaced directly to the memory manager which then interfaced to other parts of the system.

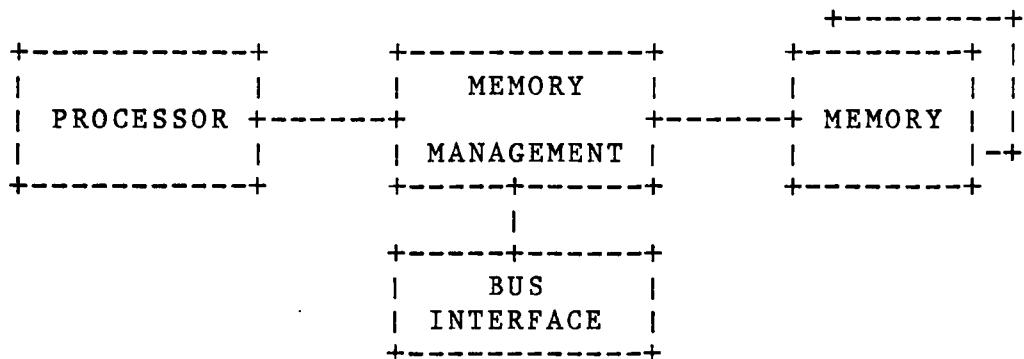


FIGURE 2 - SYSTEM CONFIGURATION

The first step in the design of the machine was to define the data paths. This involved the study of how data would flow inside of the machine and be manipulated to perform the functions needed. Several methods of implementing the architecture were envisioned and each new revision allowed different functions to be implemented at various speeds and efficiencies.

The first architecture was a very basic Von Neumann architecture with separate address and data lines. This proved to be very inefficient and slow. The design went through several variations. It was found that the hardware needed to implement a fast version of the architecture would take a long time to design and would be hard to microcode (Ucode) and debug. The final implementation decision was between a slow vertically microcoded machine and a horizontal fast machine. Figure 3A shows a rough representation of the vertical slow machine and Figure 3B shows the horizontal fast machine.

A project schedule was made to reflect the different phases of the design and to estimate the amount of time each phase would require. These steps included studying and understanding the architecture, defining the data paths, doing the detailed design, the simulation, board layout, hardware and microcode debug. The schedule from initial design to final implementation would require approximately one year to complete. This is how the initial schedule appeared at the beginning of the project.

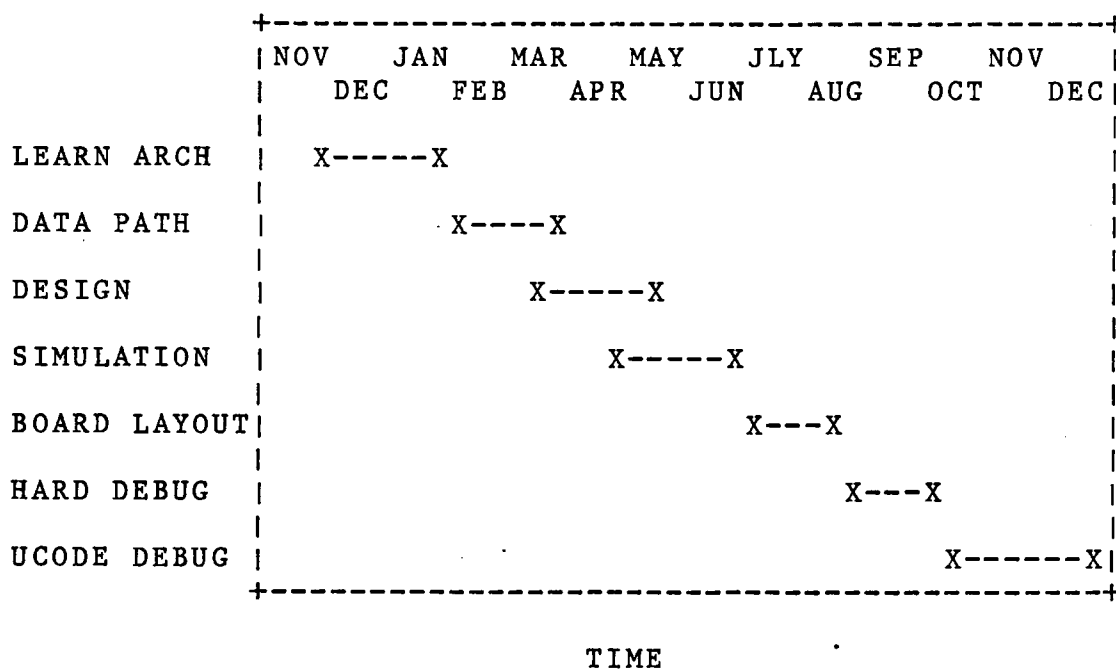


FIGURE 4 - PROJECT SCHEDULE

The detailed design was built on top of the data paths. In traditional system design, schematics would be drawn from which data bases would be created for computerized layout of PC or wirewrap boards. This design was to be fully simulated in software using a Hardware Description Language (HDL) internally available in TI. Simulation is radically different from traditional schematic drawing. The hardware is described in a special computer language where each signal is described as a variable. These variables are programmed to change in the same manner as the hardware signals would change.

One simulation consisted of a register-to-register transfer program written in BASIC to simulate flow of data along the data paths. This was used to exercise the machine to insure the data paths were adequate. It would also be used as a debug tool by microcode writers before an actual machine was available. After this, the detailed design was simulated using HDL.

HDL hardware simulation is written in "blocks". The lowest levels of design represent the lowest levels of hardware. These low levels are then put together to form higher level descriptions. This block orientation was very versatile and allowed rapid programming of many hardware functions which were linked together to form a high level description of the hardware.

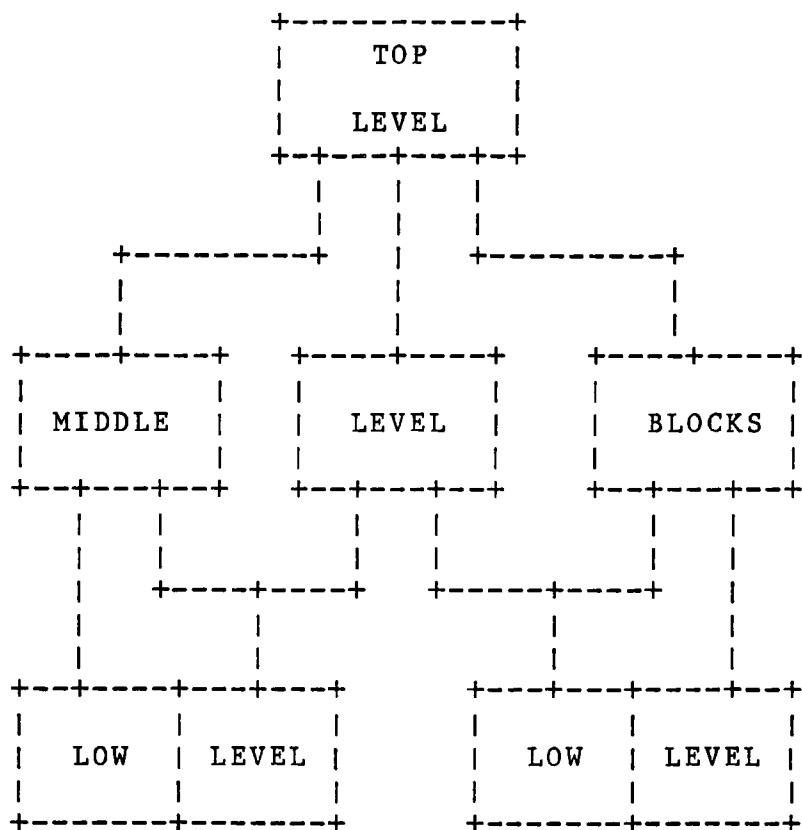


FIGURE 7 - HDL BLOCK ORIENTATION

The actual processor contains three major parts; the Large Register File (LRF), the Arithmetic and Logical Unit (ALU), and the Sequencer (SEQ). The LRF contains a large number of registers used by the ALU and provides an interface with the memory mapper. The ALU is the workhorse of the processor and does all data manipulation. The SEQ controls the system. The microcode is stored in Writable Control Store (WCS) in this portion of the processor.

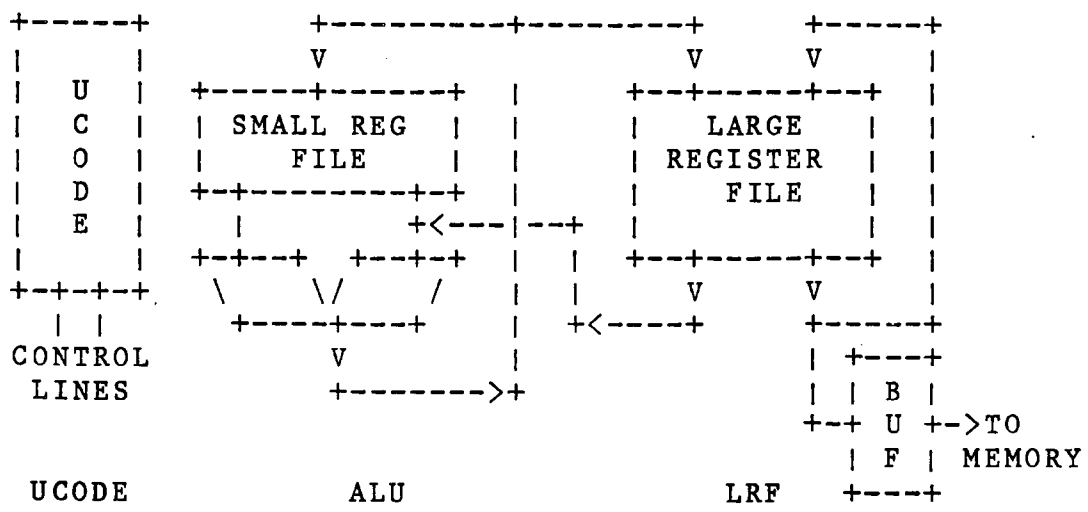


FIGURE 8 - SLOW PROCESSOR BLOCK DIAGRAM

The LRF and ALU were simulated separately using HDL. The design was first described in terms of low level blocks consisting of registers, counters and memory elements. These blocks were then linked together to form a high level representation of the design. The high level blocks of the ALU and LRF were then interconnected to form an overall simulation.

About halfway through the slow processor design, a decision was made to change the implementation to a high performance version. Since this was to be an experimental processor intended for a learning tool, it was decided that the hardware implementation should be designed to learn as much as possible. The major area which was necessary to investigate was performance. Therefore the second version would have its main emphasis on speed.

The new machine would be fast and utilize advanced techniques for improving performance. This included using various caches and pipelining the architecture. This added complexity would result in a longer design period.

Several types of caches were used in the machine. Workspace caches, Content Addressable Memory (CAM), and associative caches were all used in various places. These caches were mandatory to increase the performance to the levels needed.

The fast machine was also pipelined. Five levels of pipelining were used to simplify and speed the machine.

1. Instruction Prefetch
2. Operand Specifier Decode
3. Register Prefetch and Address Development
4. Arithmetic and Logical Unit
5. Buffered Memory Write

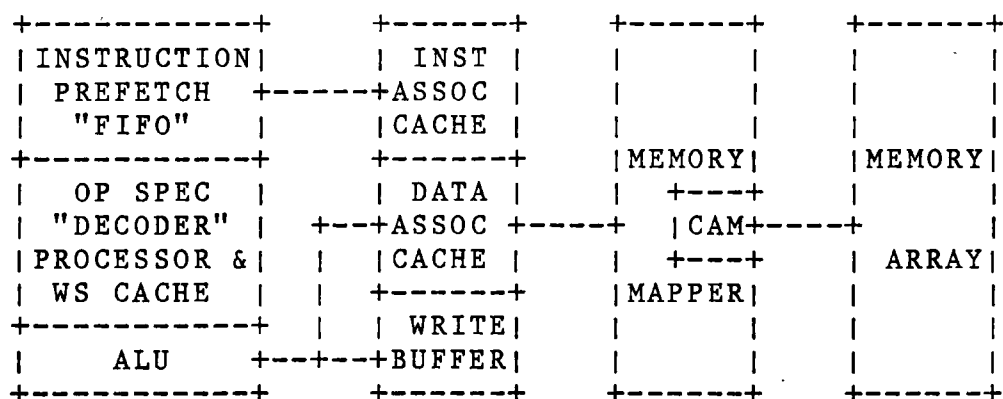


FIGURE 9 - FAST PROCESSOR BLOCK DIAGRAM

The fast processor was simulated using a top down approach to HDL. Top down programming means the high level blocks are simulated before the low level ones. Blocks such as caches, levels of the pipeline or the ALU are simulated before low levels. These high levels are later broken down into registers, latches, counters, etc. Care was taken to reflect how the lower blocks of the design would be connected in the actual hardware.

The first block to be simulated was the First In First Out buffer (FIFO). This part was very straight forward to simulate since the major elements were registers. The "Decoder" was different from the FIFO since it consisted of Programmable Logic Arrays (PLAs) and Read Only Memorys (ROMs). A deliberate effort had to be made to insure the high level code represented the actual hardware. It would be very easy to build a system which could never be implemented.

The FIFO and Decoder were simulated before the internship's termination. Many things were learned and the overall experience was very educational.

GENERAL PURPOSE PROCESSOR

Software Defined Architecture

Upon arrival at the job site, I was introduced to the group and given a document describing the architecture to be implemented. Many advanced concepts were being implemented in hardware rather than software and it took some time to understand their benefit.

Architecture has been defined as what one sees when he looks at something. This could be a building or a computer. The architecture of the internal workings of a computer system depends upon what level is being addressed. The programmer looks at a computer from a different perspective than a hardware designer. There is a difficult task in designing hardware to match the view from the software perspective.

The architecture was based on requirements of both operating systems and programming languages. Data structures, memory management requirements, time sharing functions, and the instruction set were specified and the hardware designer's job was to use this information to develop a system that would reflect this structure.

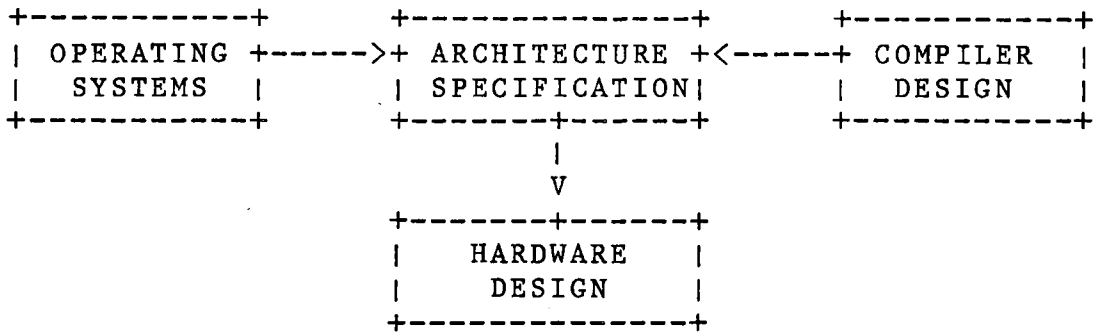


FIGURE 10 - SOFTWARE/HARDWARE INTERFACE

The software designer and the hardware designer are similar to the artist and the builder. The artist can sketch a grand building that looks beautiful. The builder in turn must take the sketch and construct the building. If the artist does not fully understand building principles he could sketch an impossible design to construct; the same is true of computer design. The artist or software designer developing the architecture must understand the constraints of the builder or hardware designer who will implement the final structure.

A major effort was being made in this architecture to put a large amount of the software complexity into the hardware. Software is becoming more expensive to develop while hardware is declining in cost. Therefore

it seems reasonable that one solution to the problem would be to increase the amount of hardware complexity in a manner which would reduce the software development cost.

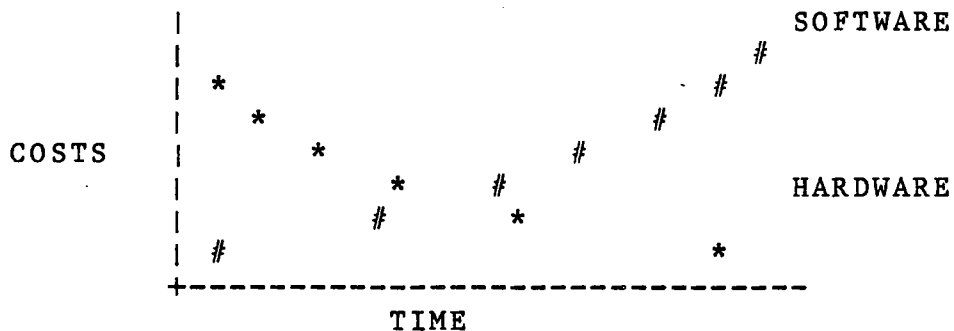


FIGURE 11 - SOFTWARE/HARDWARE COST VS TIME CURVE

Changing the software/hardware boundary leads to the problem of cost/efficiency trade-offs. The typical software programmer has little feeling for the cost of the hardware and cannot make tradeoffs based upon that cost. Therefore it is necessary to identify the most costly aspects of the architecture and make sure that the appropriate tradeoffs is made.

Interfacing with software groups provides a very challenging opportunity to learn many new things. Understanding why a new feature is needed and at the same time analyzing the cost is very difficult. To

communicate the problems to the software designer introduces a distinct communications problem. This requires knowing most members of the group, knowing which person to talk to and approaching them at the right time. If you approach them on a part of the architecture they have designed, you must be careful and not criticize them while evaluating whether it is actually needed or not.

An example of a difficult instruction to argue against is the Reverse Bit instruction. This instruction would take the operand and reverse the most significant and least significant bits in a criss-cross pattern. This is very difficult to do in software yet only takes one data path in hardware. The cost of that hardware is finite and it is questionable as to how much the instruction will be used. It was decided that the architecture would support any needed software feature and therefore the instruction would be implemented.

Architectural Implications

Many features of the architecture had various implications upon the hardware. These areas were investigated in more detail as different parts of the architecture were understood. The areas under investigation included the instruction opcodes, the operand specifiers, the processor data structures and memory management.

Conventional architectures have machine level instructions which are a specific length. One objective of this architecture was to frequency encode the instructions. The most often used instructions would take the smallest amount of room in the code stream. Less frequently used instructions would take more room. This is accomplished by encoding the instruction to be a variable number of bytes in length.

In the instruction, a distinct division is made between the operation code and the operand specifiers. The operation code is normally one byte long, but escape codes are allowed for expansion of the instruction set. The operand specifiers vary from one to seven bytes in length and there can be from one to

five operand specifiers per operation code. This provides two levels of frequency encoding; operation code and operand specifiers.

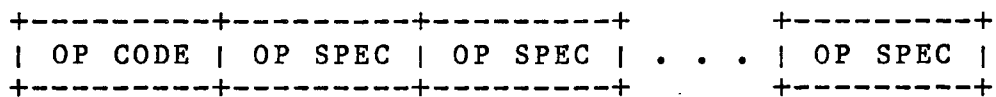


FIGURE 12 - INSTRUCTION STREAM

The instruction fetch was not precise as in most machines. The instructions could be from one byte to many bytes in length depending on the addressing mode. This meant the whole instruction could not be fetched from memory at one time. Rather, blocks of code would be fetched and instructions decoded from these blocks.

The operand specifiers were divided into various lengths and types. The type of operand specifier had to be checked with the opcode to insure that it was valid for that operation. The address of the operand specifier had to be developed from the information given in the instruction. This included the many addressing modes the processor had to support. Once this was done, the data was fetched from memory and stored into the processor. After all operand specifiers were fetched, the execution could begin.

There were other data structures in the architecture which the processor must understand. These were time-slicing, page faulting and complex message primitive instructions. The processor had to time-slice between many different tasks being run on the machine. The processor also had to support a paging system and must handle page faults in the middle of an instruction. The processor also had to deal with very complex message primitive instructions. These three things, page fault, time slicing and messages are the complex OS primitives being implemented in the hardware.

```

HARDWARE      * * * * *
OS            * * * * *
              * PAGE * TIME * MESSAGES*
              * FAULT * SLICING *
              * * * * *
TRADITIONAL   *
HARDWARE      *          INSTRUCTION
              *          SET
              * * * * *

```

FIGURE 13 - HARDWARE IMPLEMENTED OPERATING SYSTEM

Processor Evolution

The initial design was a basic Von Neumann architecture with one data/instruction space and separate address and data lines. This was not effective because it was communicating with the memory management board and not with memory. The memory management board in turn manipulated the data in such a way that it could access actual memory and return a value to the processor.

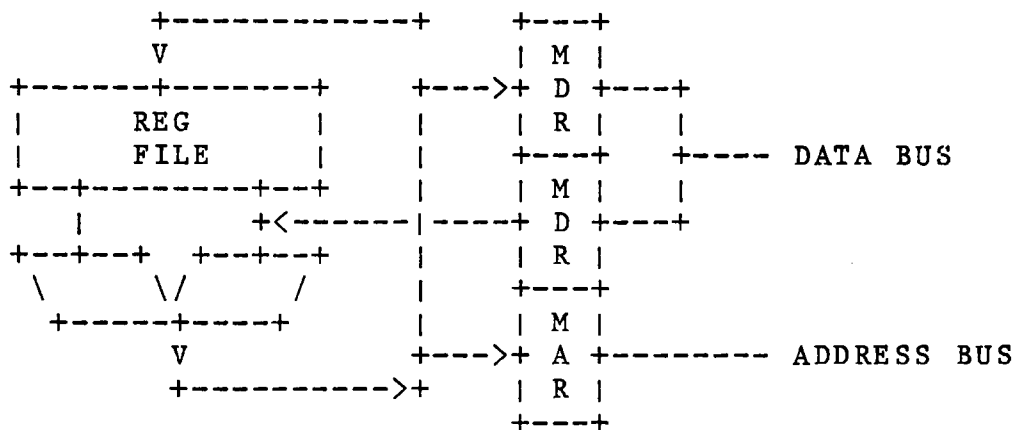


FIGURE 14 - VON NEUMANN ARCHITECTURE

The next type of architecture had a combined data and address bus. This was efficient since the memory management board had to manipulate the address before he received the data. Other data paths were added to speed various operations such as a barrel byte shifter

hardware to speed the execution. Rather than more hardware, there would be more microcode written. This would have the added benefit of hardware being available to the the software group sooner.

With so much functionality being put into microcode, it was imperative that the microcode be easily written. Past experience with microcoded machines shows that a large amount of time is spent writing and debugging the microcode. Therefore the machine must have the necessary support to help write, modify and debug the microcode.

Since the machine would be used extensively to develop the system software, there was a need to make the machine easily testable and maintainable. There should be no hidden data paths or registers that could not be directly tested. This design attitude would allow diagnostics to be written to support the machine during software checkout.

The data paths of the final machine provided a maximum of efficiency with a minimum of complexity. The final data paths were broken down into three parts. The microcode was the heart of the machine and in

Microcode

The microcode of the machine evolved as different functions were added or deleted. The first microcode control word allowed control of all fields in parallel. This word had several fields or distinct groups of bits. These fields are as follows:

1. Arithmetic and Logic Unit (ALU)
2. CONstant field low or high (CON)
3. SHiFt control (SHF)
4. Condition Code and Carry (CCC)
5. MEMory field (MEM)
6. Memory MAP request field (MAP)
7. SEQuencer field (SEQ)

This microcode word was very wide but contained many independent fields. Each field controlled a separate part of the hardware. This wide word could control every state of the machine during each microcode state.

The major problem with the wide microcode word was the size of the microcode control store memory. Additional hardware could be added to reduce the width but this would take some versatility out of the microcode. In most scenarios, not all fields in the microcode could be utilized during each machine state,

yet the functionality of the wide field and all the bits still needed to be controlled.

The best solution to the problem was to reduce the number of bits affected during each cycle. The microcode still controlled the full width, but only one section could be affected at a time. This would be done by loading only a portion of the wide microcode latch. Each of the eight sections of the wide microcode would still be controlled, one section at a time. Appendix B details the definition of the microcode.

The microcode was vertically oriented and narrow such that each machine cycle executed one microcode instruction. These instructions were very similar to assembly language instructions. Appendix C lists a sample add instruction. It can be seen from this example how similar this is to assembly language.

The microinstructions were assigned assembly language memonics such as INC, MOV, ADD and JMP. A microcode assembler was then used to translate this microcode into machine level ones and zeros. There were several people helping write the microcode. They

were familiar with assembly language and could begin programming microcode very easily.

The 32-bit microcode field is subdivided into two sections, a eight bit field and 24-bit field. The eight bit "ALL" field is subdivided into five other control fields. The control field acts as a mode control for the other 24 bits.

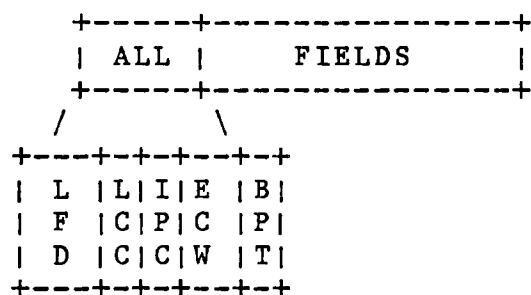


FIGURE 17 - MICROCODE WORD BIT FIELD

ALL Field

The first eight bits are referred to as the ALL field. The first subdivision of this field is into the Load Field (LFD). The LFD is used to determine which portion of the machine the lower 24 bits will control. The other portions of this field controls signals

affected during all modes of operation. These are things such as Load Condition Code (LCC), Increment Program Counter (IPC), Enable Constant Word (ECW), and BreakPoint (BPT).

The LFD field has three bits which mode select between eight different fields. The bit assignments are shown below:

001	LCON	LOAD CON UCODE REG LOW OR HIGH
010	LSHF	LOAD SHIFT/CONTROL
011	LCCC	LOAD COND CODE AND CARRY
100	LALU	LOAD ALU UCODE REG
101	LMEM	LOAD MEM UCODE REG
110	LMAP	LOAD MAP UCODE REG
111	LSEQ	LOAD SEQ UCODE REG

These portions of the wide microcode field have been restructured in the vertical microcoded machine. The details of each will be discussed later.

The Load Condition Code (LCC) is used by several fields which have status inputs to the condition code register. The condition code register is wide enough to test the many status condition. The IPC field is used to advance the program counter at any time. The ECW is used to select a portion of the microcode from

the normal WCS or from a special constant register. This is needed if an internal address is calculated, then used by the microcode. The BPT was a bit used to signal external hardware that a breakpoint is to occur. This could be used to stop the machine, sample some data or increment a loop counter. This would be very useful in data collection.

Specialized Fields

The ALU field controls the functions of the ALU. The control fields include two source register addresses, a destination address, the shift controls, logic and arithmetic function selection. Also there is the ability to read and write special purpose registers such as the Constant, timer, byte pipe, program counter and the LRF.

The CONstant (CON) fields are needed when the microcode uses a constant in some operation or on a computed branch. The Constant In (CI) field allows the microcode to contain constants needed in the execution of an instruction. Another Constant Out (CO) field exists and is the destination of the ALU field and used

by the ALL field. There are many applications for this such as the table branch into the microcode.

The SHiFt control (SHF) field sets the direction and parameters of a shift operation. Care must be taken to set this field to a known value before an ALU operation is performed.

The next field is the CCC field. This sets the condition code parameters. This field is used to select the proper output from the condition code and status multiplexer. This is then used by the sequencer to control conditional branches in the microcode.

The MEM field controls the reading from and writing to the LRF. This includes writing into one of several address counters, incrementing these counters then reading and writing to the data registers. There is an additional pointer into the register file used by the memory mapper. The MAP section will describe this in detail.

The MAP section controls when the memory system is to read and write data into the LRF. The interface is a request system, that is where address and data are

read or written as a group. In this system a pointer is set to the LRF which points to the address of the data followed by a tag word which defines whether it is a read/write and the length number or words. The MAP field tells the memory to run and it then reads the address and tag, does the operation, then reads or writes the results in the locations following the tag. The memory cannot address the LRF but can only sequentially access the data to which the address pointer, set up by the MEM field, is pointing to. The pointer is automatically incremented after every memory mapper initiated LRF read.

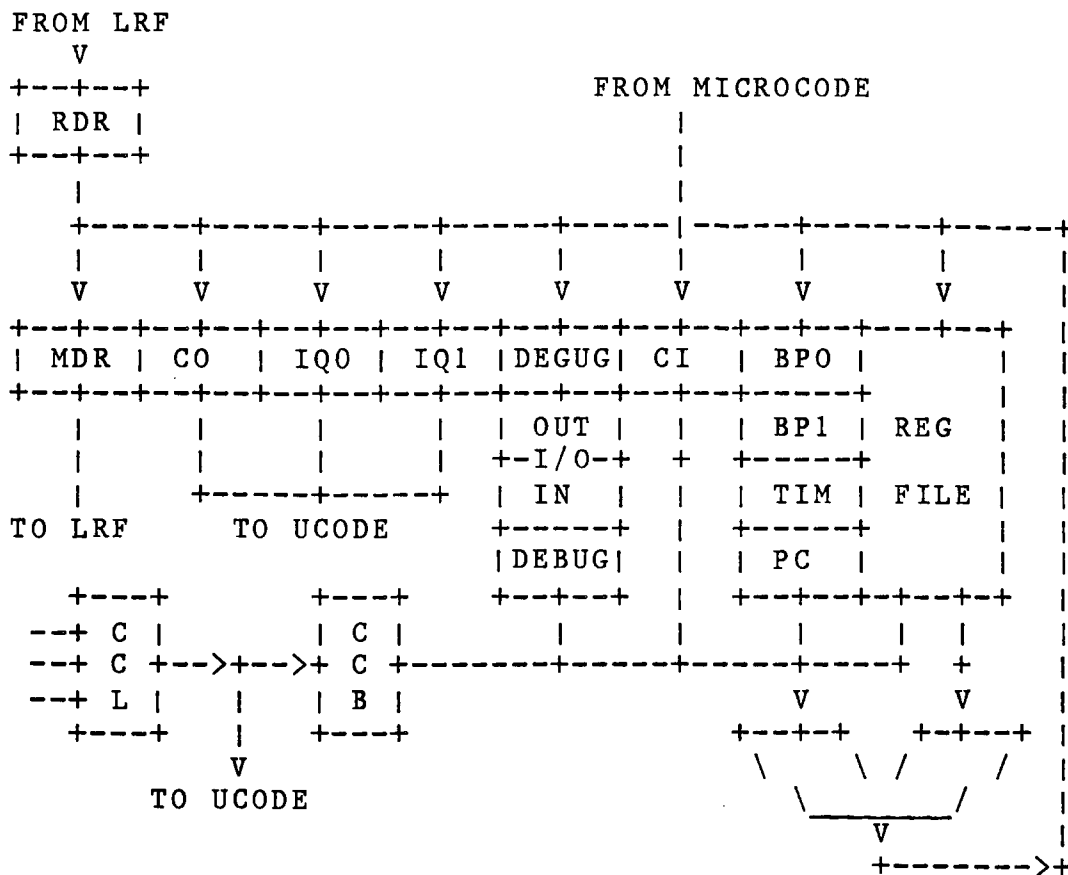
The SEQuencer controls the flow of microcode and microcode subroutine calls. The next microcode word executed could be the old address plus one, a new address given by the microcode, or the address on the Call stack. The Call stack is a push pop stack which enables the microcode to do subroutine calls. This allows efficient use of the microcode with common subroutines. The sequencer is relatively independent of the other parts of the system although a jump to a computed address is possible through the CO field.

Detailed Design

Once the microcode and the data paths had been developed, the detailed design of the hardware could begin. The detailed design consisted of determining the actual chips needed and interconnecting those chips. In addition to interconnecting these parts, the system would be simulated to insure the system was designed correctly. This would be done before any hardware existed and all debugging could be done during the simulation.

Arithmetic and Logical Unit

The ALU manipulated data within the machine. This is the workhorse of the processor and executed the guts of the instructions. This includes a 2900 bit slice machine, a "Byte Pipe"(BP), a Program Counter (PC) and a Condition Code register (CC), a TIMer (TIM), and a Instruction Queue (IQ).



ALU

FIGURE 18 - ARITHMETIC AND LOGICAL UNIT

The 2900 series are standard devices used extensively in bit-slice design. The chip family contains several parts including the ALU, register file, condition code select and shift control. The ALU can have inputs from the register file or an external data path. This external data path busses the output of the IQ, PC, TIM, CI and BP. The ALU has the standard arithmetic and logical functions needed. The

shift controller has the needed functions to accomplish arithmetic and logical shifts. The CC has the necessary functions to set overflow, underflow and carry bits.

The PC, and IQ are related special purpose registers. The PC is the program counter and is incremented when a byte from the IQ is used. Whenever the PC counts over a word boundary, the processor fetches the next word in the code stream. The IQ feeds the sequencer to select where the microcode will execute. This is in the form of a table branch where the lower 8 bits of the branch address comes from the instruction stream and the upper bits from the microcode.

There can be some pipelining done in the processor. When code is fetched from memory it is stored in the LRF. This could be one word or several. When the ALU uses the last word from this cache, it starts a memory request. This memory request is then executed in parallel with the execution within the processor.

The TIM is a simple counter and is incremented every tick of the system clock. This counter is used to allocate time slices to different tasks. An overflow latch must be polled at the end of each instruction to check for the end of a time slice.

The BP is a byte rotater/selector. It has two words as inputs and one word as the output. The output word contains consecutive bytes of the two input words pointed to by the rotate select. This allows many versatile applications. One very useful application is to align words not on word boundaries. Another would be to speed a bit shift operation.

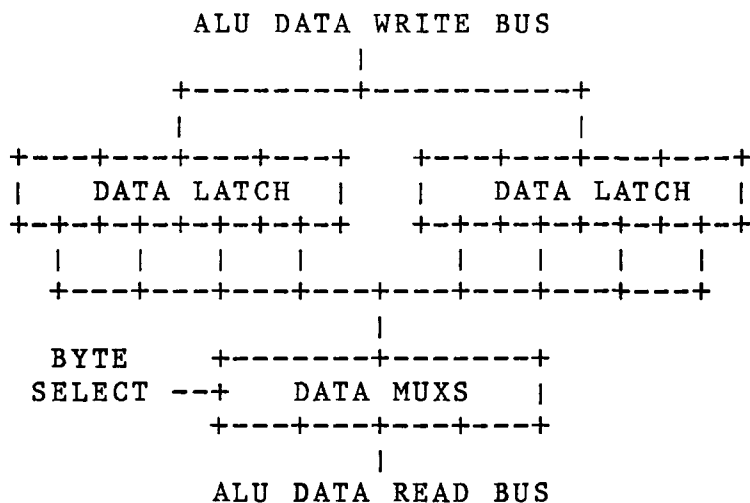


FIGURE 19 - BYTE ROTATER/SELECTOR

The CI and CO are the Constant In and Constant Out registers. The CI latches output from the microcode to the ALU and the CO latches output from the ALU to the microcode. The CI register is controlled by the CON field in the microcode. This field can load the register with a constant value of data. The ALU can load the CO register as a destination of the ALU. This register is then enabled to the microcode by the ECW field.

The SHF field is used to set up parameters of the ALU shifter. This function is not often used but takes many bits of microcode because of the many various options allowed. The shifting is accomplished using the AMD 2900 series parts including the 2904 status and shift control unit. This is a very versatile unit but difficult to control.

The CCC field is used to select a status bit or condition code bit. This selected bit is then used by the SEQ field in a conditional branch. If a branch is taken the condition was true, otherwise sequential execution continues. There are 32 bits of status and condition codes but only one of them can be tested. All of the error condition codes are ORed together and

tested at the end of each instruction. If an error exists, the microcode branches to a routine which resolve the error condition. This reduced the number of tests to check for interrupts and errors.

Large Register File (LRF)

The LRF is a two-port memory accessed by both the memory map and the processor. There are several counters used as pointers into the LRF. One is switched between the memory map and the processor. This limits the memory map to accesses only in the area where the processor has pointed. A memory fetch is accomplished by setting up a group of words in the LRF, pointing a counter to the group and telling the memory map that a memory cycle is to be done. The memory mapper then access the data in the LRF to determine what to fetch.

FROM MICROCODE

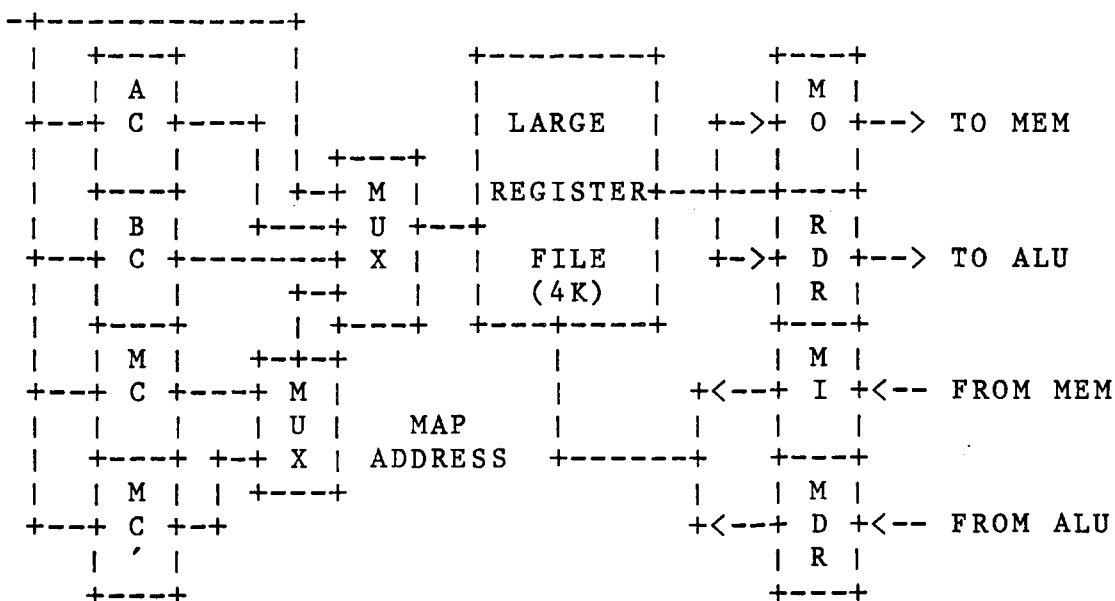


FIGURE 20 - LARGE REGISTER FILE

The LRF is 4k words of very fast (45 ns) memory. The memory is double cycled to form a two-port memory. An address multiplexer selects either the MAP address or the processor address. The processor can address the LRF either directly or through two counters, the AC and BC. Another counter can be swapped with the MAP address counter, the MC and MC' counter. In this way the processor can store data into the LRF, an address into the MC and then swap registers MC and MC'. This will indicate to the memory mapper that a memory cycle is to be started.

The ALU reads and writes the LRF through two registers called the MDR and RDR. Reading and writing occur when in the LRF mode. The MDR is a buffer of data from the ALU to the LRF. The MEM specifies a write operation and an address in the microcode field. The data in the MDR is then written into the LRF at that address. Reading data is somewhat different. The LRF read data through the RDR and directly into one of the ALU registers. This way a second cycle is not needed to transfer data to an ALU register.

Sequencer (SEQ)

The SEQuencer consists of another chip from the 2900 family. This has the controls for subroutine calls, a micro-program counter, a pipelining register and a direct input. Additional hardware was added for conditional branches, table jumps and downloading of microcode.

The table branch hardware is a simple masking operation. In normal operation, the microcode can branch to any location in the microcode memory. The table branch allows part of the branch address to come

from the microcode and part from the IQ register. The upper bits of the address are concatenated to the eight bits from the IQ and this address passed to the sequencer.

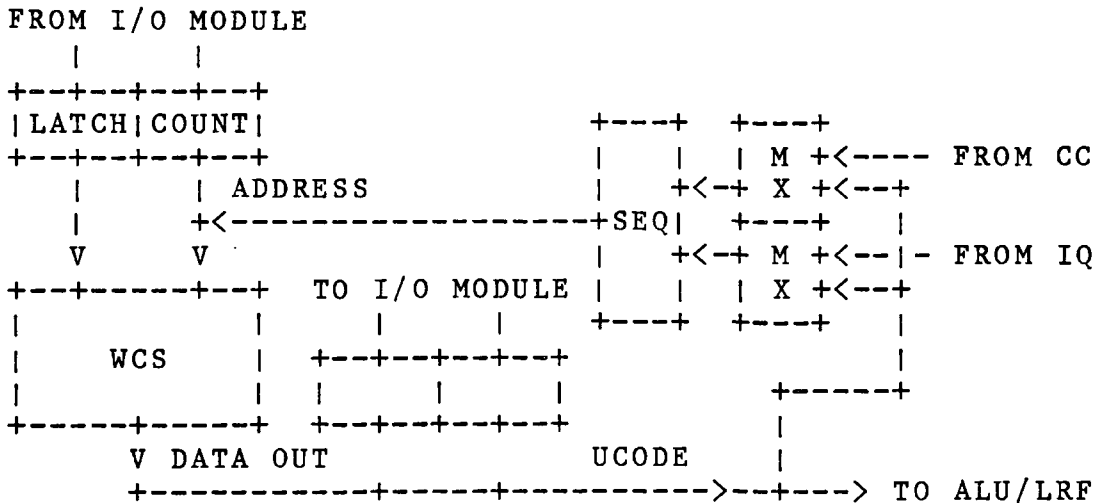


FIGURE 21 - SEQUENCER

A conditional branch uses the condition bit to select the next address in the microcode sequence. First the proper condition code must be selected. If the condition is not true, the next sequential address is selected. If the condition is true, then a branch to another location is executed. This can be a table branch or an absolute branch.

The microcode is stored in fast memory. This is called a Writable Control Store (WCS). Data paths have

been designed into the system to allow simple microcoding of the machine. This is done with parallel I/O ports and control lines. These are used to write, read and test the WCS.

The WCS was to be implemented on a separate board from the rest of the processor. Therefore an interface between the two parts of the processor had to be defined. Appendix D shows a memo defining the interface between the two boards. This division point was selected such that only 32 microcode lines and a few control lines were needed to be passed between the two boards.

Testability and Debug Features

There are several data paths in the processor to increase the ease of testing and debugging the machine. The testing and debugging is done over several I/O ports and through the use of a Trace module. Connectors have been designed into the system to allow ease in connecting trace modules to all important data paths and I/O modules connected to the necessary control points.

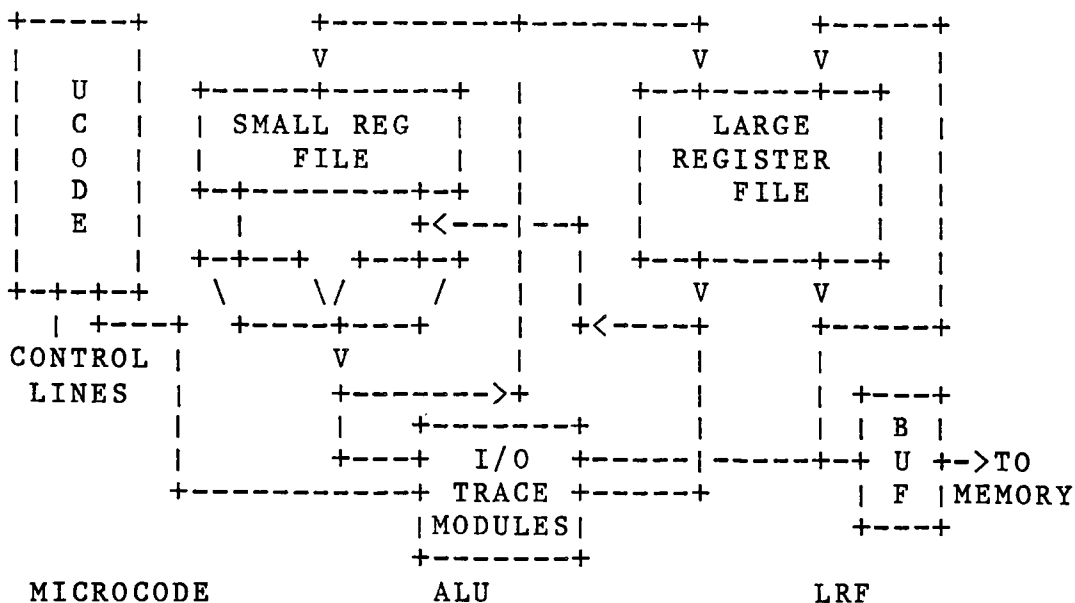


FIGURE 22 - TEST AND DEBUG DATA PATHS

The major concern is the ease of microcoding and debugging. Therefore special data paths have been added to the microcode sequencer to allow ease in downloading the microcode. The address and data lines into the microcode have an interface to several I/O modules and the sequencer output enables are controlled by the I/O modules. Appendix E is a memo describing the debug interface signal assignment and the functions of those signals.

The first I/O module is a control module. It selects the enabling onto control busses from either

the debug module or the microcode. The second I/O module outputs only data. The first module selects the destination for the second and strobes the data into the proper latches. The first module has other various control signals which allow the microcode to be written into, read from, single stepped, multiple stepped, and several other features. This allows full control over the execution of the microcode.

The I/O modules are controlled by AMPL which is a software package TI uses to control the trace modules. AMPL is used to setup and control the trace modules and to read and write the I/O modules.

Connections to the trace module from data busses in the ALU and LRF allow information to be gathered during execution of the microcode. An I/O module connected to the ALU section allows testing of the ALU's registers and the LRF. In addition to the I/O modules, several seven-segment LEDs mounted on the processor board are used during initial self test. This would allow a user to visually check for any error condition with a simple go/no go indication.

A microcode bit was allocated to enhance the tracing of information. This bit could be used to trigger the trace of a sequence, trace a state or stop the machine and wait for a single step. This versatile bit, coupled with the trace module, would provide for an easily debugged system.

Several software interface routines were to be written to provide a simple interface to the debug data. A memo describing showing the definition of these signals is shown in Appendix F. These routines were to be written by two software diagnostic engineers. They would do things from simple to complex. For example one routine would simply reset the processor. Another would allow the processor to be single stepped. These routines were defined in AMPL as subroutines. This would allow more complex programs to be written. These would do complex functions such as reset the processor, download a new set of microcode, execute until a specific point and then begin single stepping and tracing data at that point.

Simulation

Two different types of simulation were utilized in the development of this machine. First a register level simulator was written to define, test and debug the data paths and microcode. The next simulation effort was in conjunction with the detailed design and analyzed detailed delay through all parts of the system.

BASIC Simulation

The initial simulation was written in the BASIC language. This simulator can be described as a register-to-register transfer program. Figure 23 shows how the display looked to the user. This showed the important internal registers and data paths.

```

+-----+ AC 0000 0000
+-RDR- 0000 ---|MEMORY|--- BC 0000 0000
+-MDR- 0000 | | MC 0000 STK 0000
| | +-----+ DIR 0000 0000
+-----+-----+-----+ RE 0000
| | | | | ISB 0000
| R( )0000 R( )0000 CO 0000 |
| IQ0- 0000 BP0 0000 | SA 0000
| IQ1- 0000 BP1 0000 +----+ |
| DEBUG 0000 TIM 0000 | | +-----+
| | PC 0000 CI 0000 | SB |0000|
| | | | | | +-----+
| | +-----+-----+ | |
| | 0000\|00000000 | | ALU 0000
| | ALU- 0 -----|---+CC 0000 | MEM 0000
| | 0000 | | CCC 0000
| | +-----+QR+--+ | MAP 0000
| | 0000 0000 | | SEQ 0000
| | | | | |
+-----+ +---+ +-----+

```

FIGURE 23 - SAMPLE SCREEN DISPLAY

The main purpose of this simulator was to be an interactive tool used to assist in the detailing of the data paths, writing and debugging of the microcode. The register-to-register transfer allowed the data paths to be exercised with several sample programs. The actual microcode controlled the register transfer and data manipulation. These bits were assigned and developed during the writing of the simulator.

After the simulator was finished, it was used to test and debug microcode routines. This proved to be a great benefit since the execution could be visualized

on the CRT screen. No detailed hardware timing considerations other than machine clock cycles were taken into account. Each loop through the program is one state in the microcode. This would allow instruction timing to be accurately defined. Writing and debugging of microcode could be done before the hardware existed.

The user interface was the most important aspect of the simulator. The data paths were graphically displayed upon the screen. As variables changed in various parts of the system, the changes were displayed. The microcoder could actually see what the microcode was doing and visually trace the execution of his program.

One problem with the simulator was it was slow. BASIC is inherently a slow language and therefore the program was rewritten in PASCAL. This version ran much faster but the I/O was much more difficult and harder to modify. Yet both the BASIC and PASCAL versions proved to be beneficial to the microcoders. A listing of the Pascal version of the emulator is shown in Appendix G.

HDL Simulation

The detailed design was initially done in the traditional sense of drawing schematics. The several pages of schematics were drawn fast and rough to be a visual reference during the simulation.

The first step in the simulation was to learn HDL. HDL is a Hardware Description Language written for use in simulating a chip design. No one had used this language to simulate a complete design from the gate level to chip level then to board level and finally complete system level. Since the project was so complex, it was believed that simulation would reduce the time it takes to debug the breadboard. The debug would be done during simulation before hardware existed.

HDL is a block description language. The design is divided into many different blocks and each of the blocks simulated. The low level blocks were then connected by upper level blocks to form a tree structure. Each higher level reflected the functioning of larger pieces of hardware. For example a latch chip could be simulated at the lowest level with gates.

Several latches could be put together at a higher level to form a register. Several registers could be combined to form a register file. The register file could be combined at a higher level with the ALU to form the CPU. This type of simulation is very flexible since the lowest level is only written once for a latch and duplicated whenever a latch is used by a higher level. This is similar to subroutines in software design.

HDL is also easy to modify. Errors are found by tracing the program execution. This is similar to tracing hardware signals with a logic analyzer. Once an error is found a change is made and the hardware resimulated. Once a lower level is fully simulated and debugged it can be used over and over with a high level of confidence. This is better than with schematics which must be redrawn when an error is found.

Before simulation of the processor could begin, low level blocks of memory, counters, latches and multiplexers were simulated. Another engineer had the responsibility of writing these low level blocks and debugging them. I took these low level blocks and made a macro-library out of them. The macro-library

consisted of registers, a memory file, wide multiplexers and large counters.

The macro-library was then used to simulate the LRF, ALU and SEQ portions of the design. The LRF used several of the macro-blocks and combined them together to form this upper level block. This was simulated with selected data being input and internal signals traced. This would then be compared against the expected results. The ALU was also simulated in a similar way but the SEQ was handed off to another engineer to simulate. This could be done since the sequencer was on another board and the interface between the boards was well defined. The data path and block level design had been done and only the detailing of the blocks was left.

The simulation for the LRF can be found in Appendix H. The first portion of the block defines all input and output signals to that block. Next the structure of the sub-blocks is defined. This is done by simply defining the block's type then its inputs and outputs. All blocks are connected in this way until the lowest level is reached.

Communications

During this project, communication with other engineers was very important. There were many key portions of the system which were developed through close communication of ideas and possible solutions.

The architectural specifications was constantly changing while the hardware was being designed. While the data path were general purpose, it was still critical that all changes be fully understood to insure they could be implemented. All changes were to be taken into account by modifying the microcode and not the hardware. For the most part, additional data paths were not needed although some microcode sequences became rather long to perform.

The debugging of the system was done with AMPL. Other software engineers were to write the procedures to accomplish the needed tasks. This again involved communications problems since I had to convey how the hardware would work to people who did not fully understand hardware. Yet with several memos and many informal meetings they were able to understand and write the procedures which were needed.

Change in Direction

Because of the slow speed of the breadboard, it was decided that the development of a slow breadboard be halted and a high performance version be built. It was deemed that a higher risk should be taken at an earlier date so that more could be learned from this experimental design.

The project was then redirected from building a slow vertical machine to a parallel fast machine. This posed many new problems and changed the view of the architecture drastically. Many things which were taken for granted in the slow machine and being done through many cycles must now be done in one. Therefore every data path and every microcode state had to be changed. This change in direction was very frustrating since several months of work was thrown away.

Simulation was the most important thing which was learned from the breadboard. This was one area where what was learned on the slow machine could be applied to the fast processor. A bottom up approach was used on the breadboard in writing the blocks and hooking them together. A top down method could have also been

used and in retrospect would have been the best way to go. The fast machine would be much more difficult to simulate and therefore the top down approach would be more important to use.

With this decision, all work on the slow breadboard was stopped and the new direction towards a faster, improved model was begun.

FAST PROCESSOR

Objectives

Performance is the sole objective of the fast version of the processor. While Millions of Instructions Per Second (MIPS) is not a good measure of performance, the speed goal of the machine was 2 MIPS.

The slow breadboard was at the low-speed/low-cost end of the speed/cost curve (A). This machine is high performance and therefore a high cost implementation. This puts the second processor at the other end of the curve previously discussed (B).

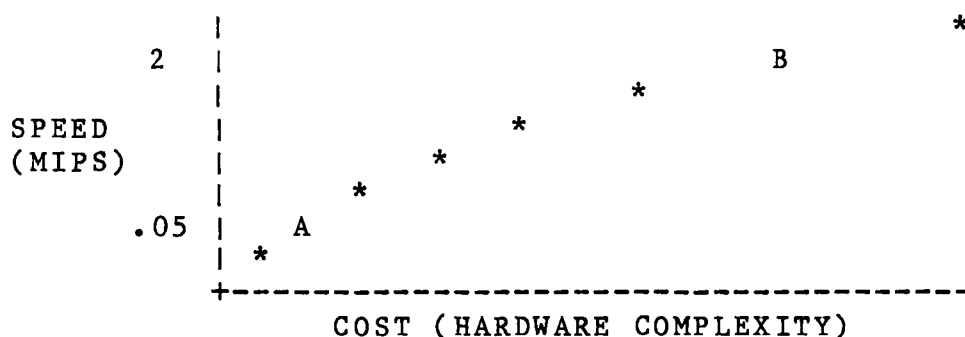


FIGURE 24 - HARDWARE SPEED-COST CURVE

The first problem facing the hardware designer is to sort out the software architecture. This results in defining what will require extensive hardware versus

what remains in software. A good understanding of this was gained from the first breadboard. Architectural implications of caching and pipelining also must be studied. Successful cache placement is very important yet architecturally dependent. Pipelining requires an understanding of data flows and how that flow could be layered in this architecture.

Data paths for the fast processor will be broken down into several areas. First, existing methods of speeding a processor will be discussed including caching and pipelining. Next, specific hardware developed to speed this processor will be described. This will include a decoder for the operand specifiers and a method of speeding the jump and call instructions.

Conventional Caching Methods

One method commonly used to speed execution is through the use of caches. In this report, caches are defined as any fast memory added to the system to improve performance. There are several types of caches and many methods and positions for the caches to be

placed.

The workspace is an area of memory which contains the most frequently used variables. This would be similar to the 990 workspace and is used like a register file. A workspace cache can be used to store the data in the processor such that access to this area do not actually go to memory. This cache is bulk stored and loaded whenever a context switch occurs.

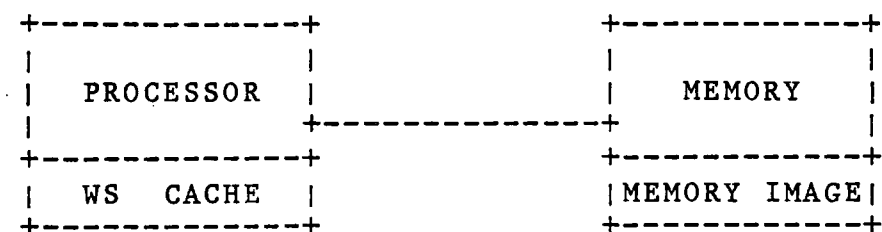


FIGURE 25 - WORKSPACE CACHE

Another type of cache is a Content Addressable Memory (CAM). This cache stores both data and the full address of that data. An address is presented to the cache and it is compared in parallel with every address in the cache. If the compare is true, a cache "hit" occurs and the data corresponding with that address is presented on the output. A large amount of dedicated hardware can make this cache very fast.

A CAM is used frequently to speed address translation. This is important in various memory mapping schemes where an address is mapped from one address space to another. Most mapping schemes require the logical address output by the processor to be translated into a virtual or physical address. This address would be the input to the cache. If the address had been previously mapped and stored in the CAM, data read out of the cache would be the translated address. While being very fast, this cache is also very expensive and usually only has four to eight entries.

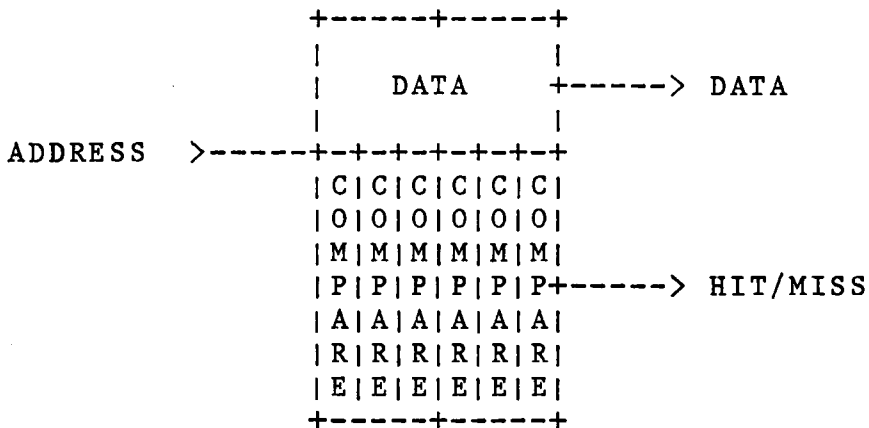


FIGURE 26 - CONTENT ADDRESSABLE MEMORY

Another type of cache is the Associative Cache which stores both data and the group associative address of that data. When a memory address is presented to the cache, data and address are read from the cache. After the read, the address stored in the cache is compared to the associative portion of the memory address presented to the cache which determines if the cache contains valid data. This is slower than a CAM since the compare occurs serially after the data is read but can result in a less expensive and larger cache. This cache is typically between 2k and 16k in length.

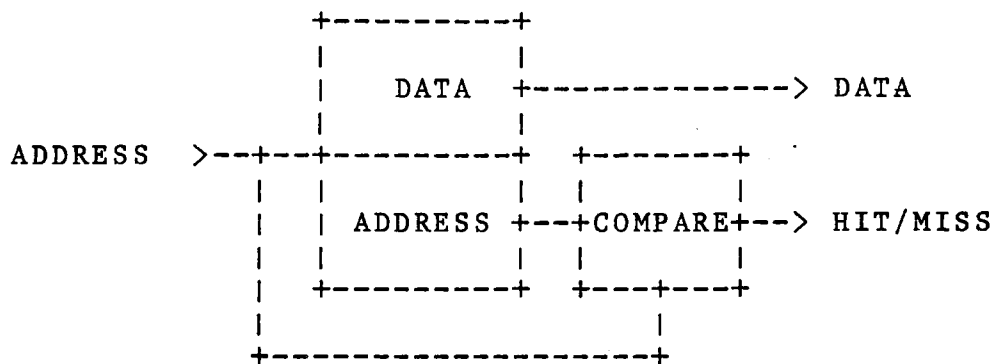


FIGURE 27 - ASSOCIATIVE CACHE

Caching Evolution

Data cache placement is very important to system performance and must be studied carefully. Appendix J shows various cache placements and the resultant speed of the processor. These are estimates based upon the number of memory accesses and the percentage of those accesses which hit the cache. From this study, it was determined that the processor should contain several types of data caches at various locations.

The workspace is an architectural feature which is preferred by software designers. The workspace is an area of memory used similar to a register file. When a subroutine call is made, the workspace pointer is changed. This results in the processor writing the current contents of the cache cache into memory and loading data from the new workspace.

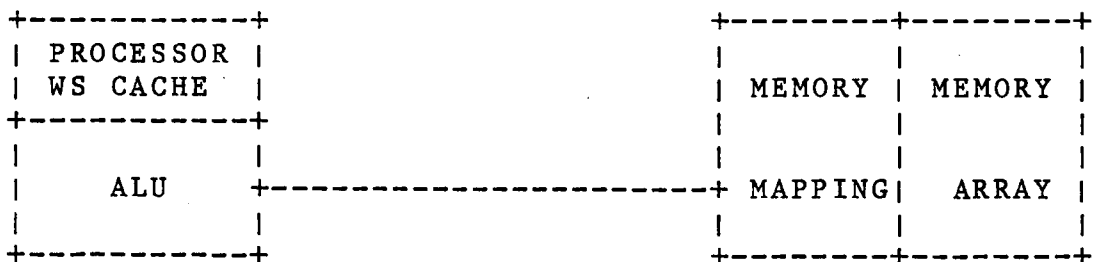


FIGURE 28 - CACHE EVOLUTION I

Another general data cache is needed for frequently accessed data not found in the workspace. This is an associative type cache which is demand loaded with "write through". Demand loading occurs on memory reads when a cache does not contain the correct data. This is referred to as a cache miss and when this happens a full memory cycle is started. When valid data is retrieved from the memory system, the data along with the proper address would be written back into the cache. A "write through" occurs when every cache write also results in a memory write. This causes the memory to always contain the current data thereby avoiding the need to ever "flush" the cache into memory.

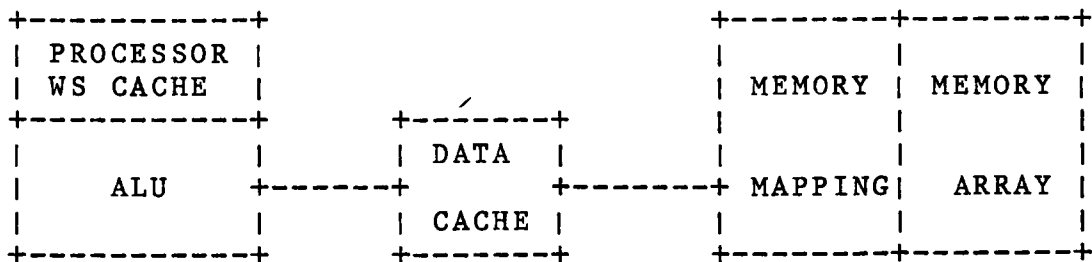


FIGURE 29 - CACHE EVOLUTION II

An instruction cache is in parallel with the data cache and is very similar. In fact, the same cache subsystem could be duplicated for both. This cache is also demand loaded but never written into. Since the data cache and instruction cache are in parallel, both accesses can occur in parallel. This allows for a higher memory bandwidth. The back side of the data and instruction caches are interfaced to the memory system and must arbitrate for the memory accesses.

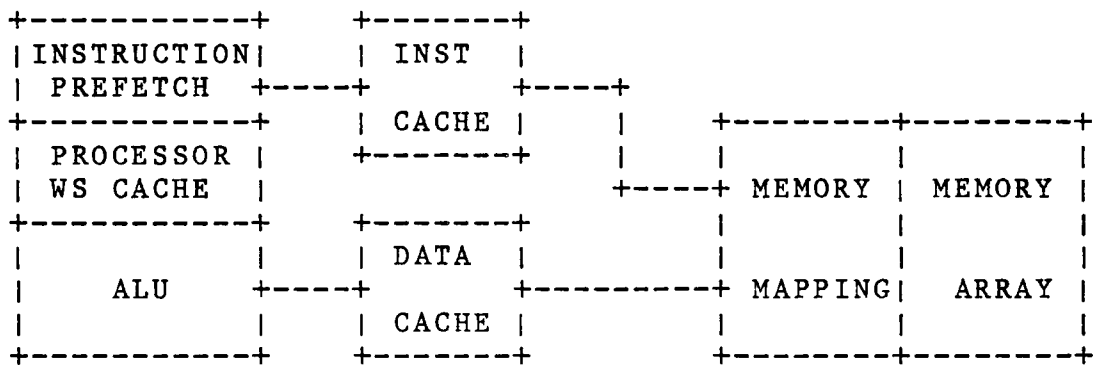


FIGURE 30 - CACHE EVOLUTION III

When a write occurs to the memory system, it is immediately written into the data cache and presented to the memory mapping. The write buffer will queue the write request and the memory mapper will write the data when it is not busy. This allows the processor to continue to execute from the data cache while the memory system is completing the write.

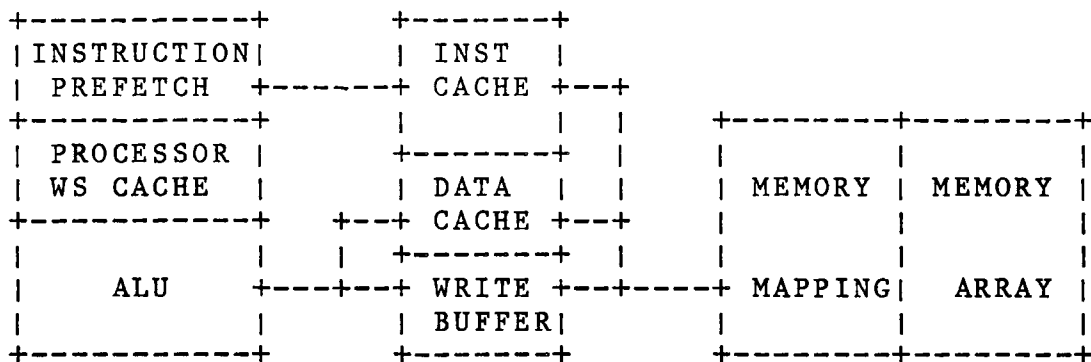


FIGURE 31 - FINAL CACHE CONFIGURATION

There are two address translation in the system. A logical-to-virtual and virtual-to-physical. Address caches are mandatory in this system. The logical-to-virtual translation occurs in the processor while the virtual-to-physical occurs in the memory mapper.

The address register file is a defined set of base addresses which must be used when addressing memory. When a base address register is loaded into the processor, the address is resolved from a logical address to a virtual address and this virtual address is cached in the address register file. When a base register is specified, the previously translated virtual address is then used in address development.

The virtual-to-physical translation occurs in the memory mapping portion of the system. This address translation utilizes a CAM to improve speed.

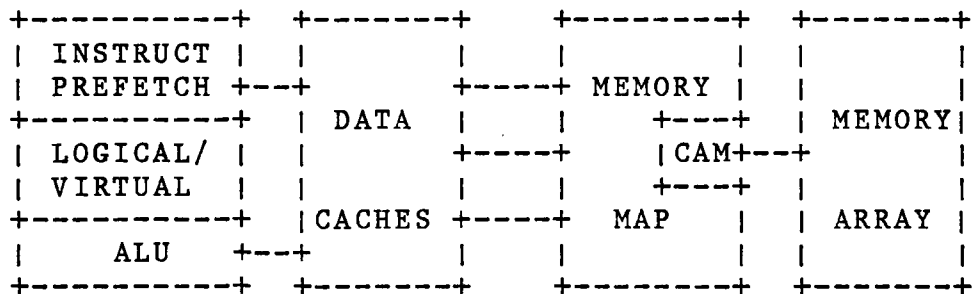


FIGURE 32 - ADDRESS TRANSLATION

Pipelining Evolution

Pipelining is another widely used method of speeding the execution of a machine. Pipelining is a method of paralleling various portions of the instruction cycle such as the fetch, decode and execution phases.

Several studies were made of data flowing through the pipeline. Appendix K shows an example of one study. An instruction is traced through the pipeline and the diagram keeps track of how the data flows. The

pipeline needs to be balanced such that each level stays equally busy. This diagram can show holes in the pipeline when one level waits on another. This is a useful tool in developing the appropriate division in the pipeline and the hardware to put into each level.

The processor is pipelined in several layers. The first is the instruction prefetch. The second is the operand specifier decoding. The third level fetches data from internal registers and develops the addresses. The fourth level contained the ALU which does data fetches and manipulation. The last portion of the pipeline was a memory write buffer.

The instruction prefetch is the first level of the pipeline. It looks ahead of the current program counter and retrieves the addition words of code from the instruction cache. This is fed into a FIFO which buffers several code words and performs code alignment and extraction.

The speed of the instruction execution is dependent upon the bandwidth to memory. With small data paths, the instruction fetches occur more often than data fetches. As the paths get wider more and

more of the instruction stream is fetched on each memory access. This will improve performance of data manipulation by speeding the handling of larger data types with a major benefit coming from not making as many instruction fetches. This is very important especially when instruction prefetch is heavily used.

The next level of the pipeline is the operand specifier DECODER. Instructions are aligned on byte boundaries and can be many bytes in length. This section takes these instructions and decodes their operand specifiers. The DECODER extracts the base register designator, workspace register designator, displacement and/or an immediate value then passes the pointers and data to the next level of the pipeline.

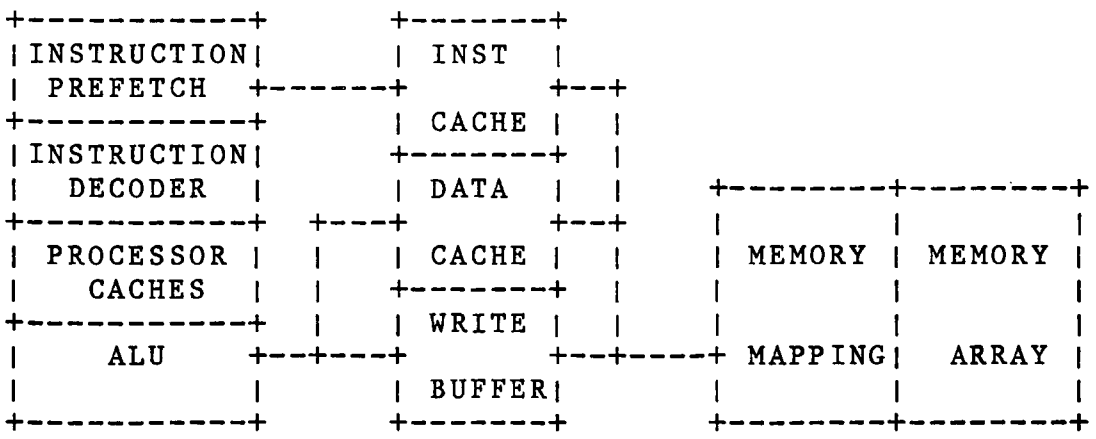


FIGURE 33 - PIPELINE

The register fetch and address development is the next level of the pipeline. At this level the base address, limits, and access privileges are read from the internal address register file and the index is read from the workspace cache. The base is then added to the index and displacement and compared to the bound. The results of this stage are then passed to the ALU.

The ALU does all memory fetches, data manipulation and writes into memory. After all address, register or immediate data has been developed, this leaves a relatively small amount of work for the ALU. The ALU can do a memory fetch during every cycle. There is no delay when the cache is "hit" or contains the correct data. If there is a cache "miss" a longer memory cycle is started. Once the ALU has all the data needed for the operation, it proceeds with the calculation.

Performance Improvements

One factor which is of key importance to software implementers is the speed and functionality of the CALL

instruction. In high level languages, the CALL instruction occurs frequently. Yet most architectures do not have any assemble language instructions which resembles the high level language CALL. In order to accomplish a CALL, many assemble language instructions must be executed to branch to the routine, pass parameters and save the current context. In this machine one single instruction does all of this. It saves the current context, loads the new context, gives the branch address and sets up parameters to be passed.

Fast execution of the CALL instruction requires special purpose hardware. The context changes when a CALL instruction is executed. This means the old context must be stored into memory and a new context read into the processor. Addresses must be developed from this old context and data fetched. One method of speeding the Call is to cache the last context in the processor. This is done by simply saving a copy of the old context within the processor so that the last context is not fetched from memory on a return.

In Figure 34 the internal processor cache #1 could contain the old context. The #2 bank contains the current task. When a return is executed, the old

context is still in the processor and a switch is simply made from bank #2 to bank #1. This also improves speed when task #2 wants to get a parameter from the old context. All registers are still in the processor and it is simple to calculate the passed operand specifiers address from the old context.

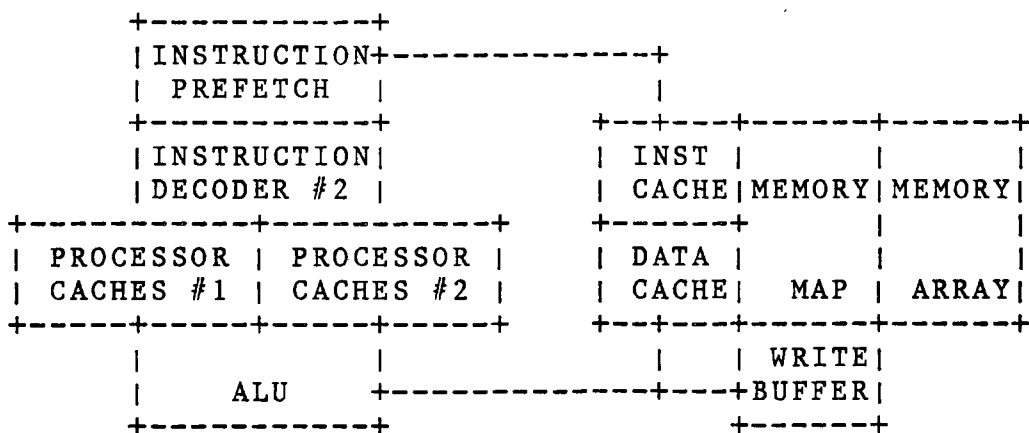


FIGURE 34 - PIPELINE IMPROVEMENTS/CALLS

There are problems which arise from this structure. If more than one call is made, the first Call's context must be destroyed. This results in only the last context being saved. It is believed that most programs execute CALLs and RETURNs in groups which are only one level deep. If this proved false, several levels of context caching could be added to speed the CALL.

Another speed problem arises from JUMP instructions. Since the machine is pipelined, a JUMP instruction causes the pipeline to be flushed and the new instruction stream fetched from the branch address. One way to speed this is to have a dual prefetch and decode system such that both branches of the JUMP instruction are prefetched and decoded ahead of time. This way both directions of the branch execute at the same speed assuming that the prefetch can keep the queues full.

This same method can be used to speed the CALL. If the old prefetched instruction stream were kept in the processor, the old context would be preserved along with the old instruction stream, and no penalty would be paid upon a return. This requires three parallel paths within the machine from instruction prefetch through decoding and register prefetching. Instruction decoder #2 and #3 would now hold the current context. #3 would contain the instruction stream if a jump were taken while #2 would have the currently executing instruction stream.

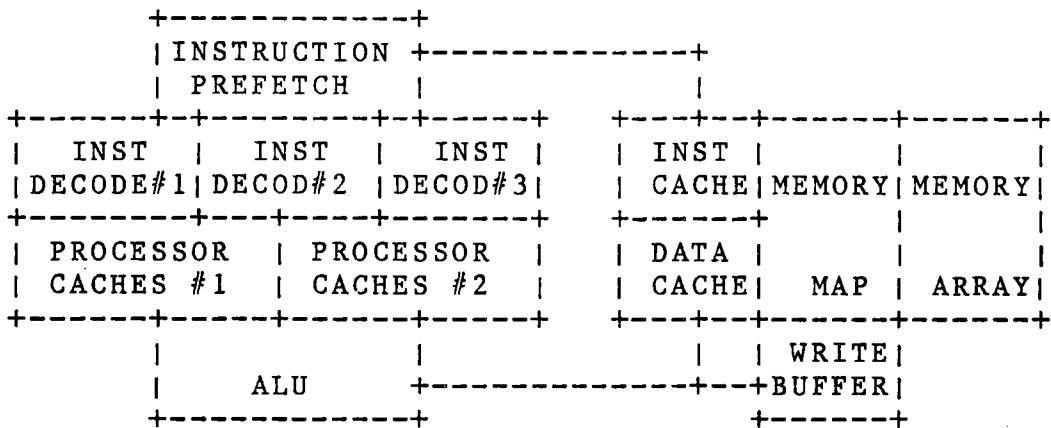


FIGURE 35 - PIPELINE IMPROVEMENTS/JUMPS

The CALL, Jump and current context portions of the pipeline are all dynamic. The assignment depends upon where the processor is currently executing and which bank is an available prefetch queue.

Detailed Design

The operand specifier decoder was the first part of the machine to be detailed. This includes the FIFO buffer and the operand specifier DECODER. It is assumed that an instruction prefetch feeds code into this portion of the processor and the output is the base register addresses, cache register addresses,

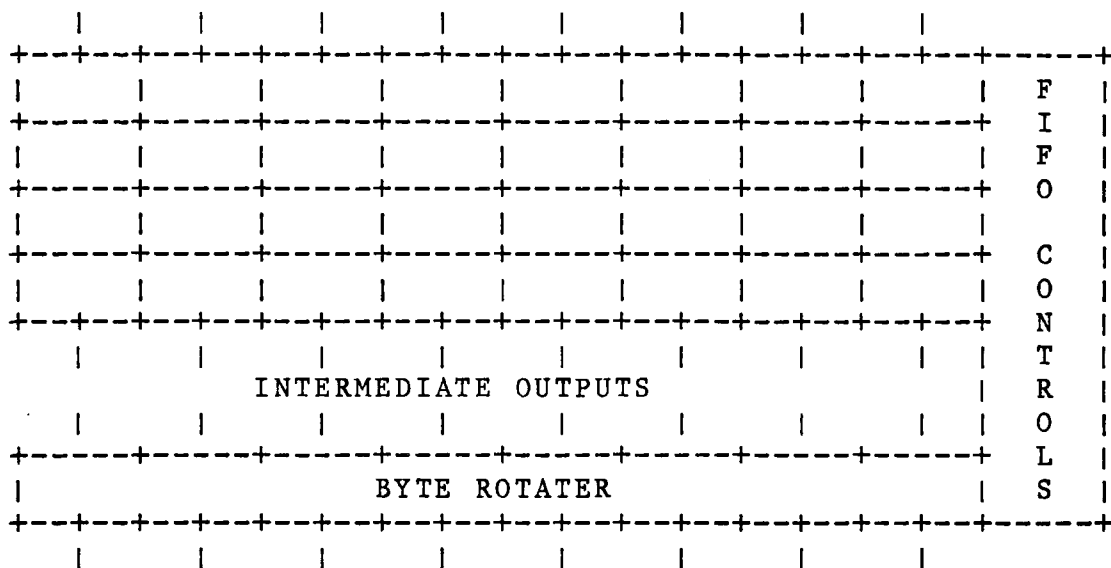
displacements and literals.

FIFO Buffer

The first part of the detailed design consisted of a FIFO buffer. This buffer is a queue for the prefetched instructions and a code stream aligner for the decoder. The aligner extracts any eight sequential bytes from the FIFO. The select input determines which byte is the first position on the output.

Up to seven bytes of the code stream are needed for decoding the operand specifier. Therefore it was decided that each level of the FIFO would be eight bytes wide. The number of levels or depth of the FIFO depends upon how far ahead of the current program counter the FIFO needs to buffer the code stream. Two levels, each eight bytes wide, are mandatory to be able to properly utilize the FIFO. Four levels were selected as the number of total levels in the FIFO.

INSTRUCTION PREFETCH INPUTS



DATA OUTPUTS

FIGURE 36 - FIFO

The FIFO is constructed from byte-wide register latches, eight per row with four rows. The outputs of the four rows are tied together and the controller determines which output is enabled. Following this intermediate output is a byte rotater.

The FIFO read is a two step process. First, the byte pointed to by the select input and the next eight consecutive bytes of code stream are enabled, one per column. Secondly, these eight intermediate bytes are

rotated until they were properly aligned.

The FIFO controller has an internal counter which keeps track of the next write location. From the next write location and the select input, the controller can determine when a level in the FIFO becomes empty. When this happens, a handshaking signal with the instruction prefetch is toggled to begin the next code word prefetch.

The amount of code stream prefetched depends upon the bandwidth between the processor and the memory. For a 64-bit data path, one of four writes would be possible. For narrower data paths, the inputs would be arranged differently and the controller would cause a different combination of registers to be loaded. An output control signal tells the following portions of the pipeline whether the FIFO has valid data or whether it must wait until the prefetch catches up with valid data.

Operand Specifier Decoder

The second level of the pipeline is the DECODER. This part of the pipeline decodes the operand specifiers into a form that the processor could easily understand. The rotated and aligned output from the FIFO feeds several PLAs and ROMS. These decode the necessary information from the operand specifier such as displacement, literal value, type, length and a pointer to the base address register and workspace cache. These outputs are then fed to the address developer.

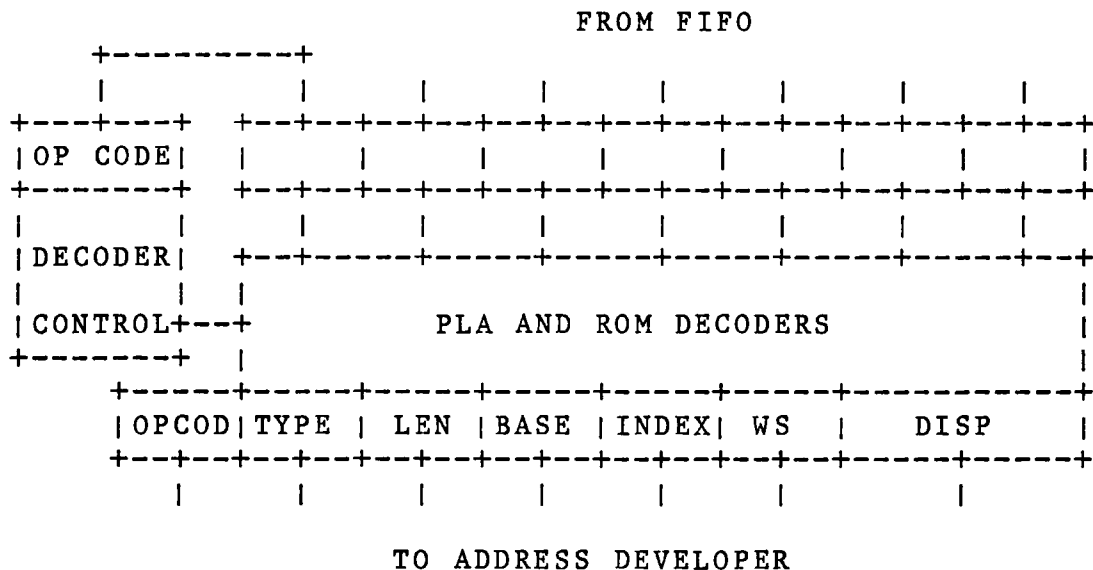


FIGURE 37 - ADDRESS DECODER

The first byte in an instruction is always the opcode. The number of operand specifiers is decoded from the opcode and passed along to the ALU. The decoder expands the following operand specifiers into a form needed by the address developer.

There are several fields the decoder must recognize and decode. First, the length of the operand specifier is decoded into a three bit field. Second, the base register field is decoded to determine the base address register used in the address development. This is encoded in a four-bit field. The next field is the workspace cache pointer. If the access is to a workspace data location, this four-bit field will point to that register. The last field is the displacement field. This is the offset from the base register to be added to that base. This field can also contain immediate data to be used by the ALU.

The DECODER controller is a small microcoded engine which has all possible combinations of operand specifiers stored in its control store. This controller counts operand specifiers and provides additional information about which operand specifiers are valid for the particular instruction. The operand

specifier for each instruction must be type checked to insure their validity.

Many PLA and ROM decoders are used to decode the operand specifiers. The operand specifier may consist of several bytes of information. Each byte may depend upon the previous byte to determine what its function will be. Therefore several widths of decoders are needed for one operand specifier.

The first section of the DECODER decodes the position of various fields within the operand specifier. The base register, workspace cache pointer, and displacement all have distinct places that they can occur in the code stream. The next portion of the DECODER takes these pointers and decodes the information starting at the specified location.

The displacement is always the last thing in the operand specifier stream. This displacement could be the first, second, third or fourth byte in the operand specifier and all four bytes as well as the type must be looked at to determine the correct position from which to pull the displacement. A displacement decode table is shown in Appendix L.

The previous decoder tells where the displacement is to start. The first byte of the displacement has encoded in it the length. If the first bit is a zero, then the displacement is seven bits long. If the first two bits are binary 10 then the displacement is fourteen bits long. If the first two bits are binary 11, the the displacement is 30 bits long. Several multiplexers are used to select every possible combination. A large PLA decodes and selects the proper combination. This large amount of circuitry is needed to get the displacement decoded in one cycle.

```

0XXXXXXXX          X - LITERAL DATA BITS
10XXXXXXXX XXXXXXXX
11XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX

```

FIGURE 38 - DISPLACEMENT LENGTH DECODING

The final output of the DEVELOPER section is a set of well defined pointers, information fields, and data. This includes the type, length, base register, workspace cache, and displacement. This information is then passed to the address developer.

The output of the adder, the fully developed operand specifier, is fed to the ALU. If the operand specifies the workspace, then the address register and the displacement are selected as null and set to zero. This would result in the output of the adder being the actual workspace data. If the operand specifies a base register address then the address is accessed while the workspace pointer and displacement would be null and again set to zero. This would result in the output of the adder being the address register value. If the operand specifies an indexed address with a displacement then all three are added together to provide the proper address.

End of Pipe

The fourth level of the pipeline is the ALU. This portion is used to actually execute the instruction. If the operand specifier is immediate, workspace cache, or address register direct, the data is read and used immediately. If the operand specifies an address, then the address would be presented to the data cache. If the data cache contained the data item, it would be returned on that cycle. If the data cache were

The fifth and last level of the pipeline is the memory buffer. This portion of the design would not occur in the processor but in the memory controller. When data is written into memory, it is stored both in the data cache and the memory buffer. This pseudo-level of the pipeline is necessary to write data into the main memory without slowing the processor down. The buffer allows the write-through to occur fast with the processor not having to wait for a full memory cycle to be completed.

The overall machine has three data paths from the processor to main memory. One is from the instruction cache to memory. A second is from the data cache to memory. The last is from the memory buffer to memory. There are priorities associate with these data paths. The data cache access is the most important and has the highest priority. The instruction cache is the next most important and lastly the memory write buffer. When the memory buffer is full it immediately becomes the highest priority and must be serviced before anything else is written into memory.

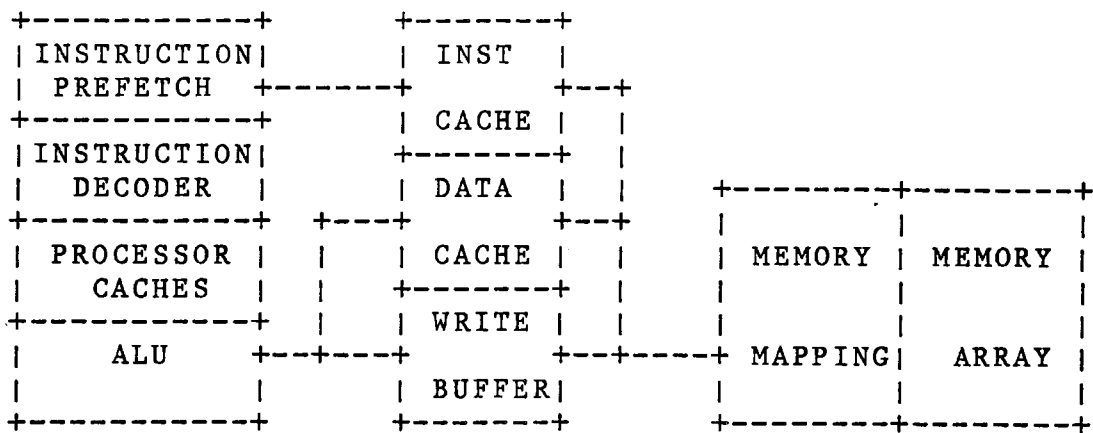


FIGURE 41 - MEMORY DATA PATHS

Some possible modification to the pipeline could result in a reduced cost. One question is, "how busy will the main memory be?". If the memory is not kept busy, an instruction cache may not be necessary. A wide path from main memory to the instruction prefetch may be all that is necessary. Another way to reduce cost is to eliminate all but one of the FIFOs. The three FIFOs can be viewed as performance improvements while only one is necessary.

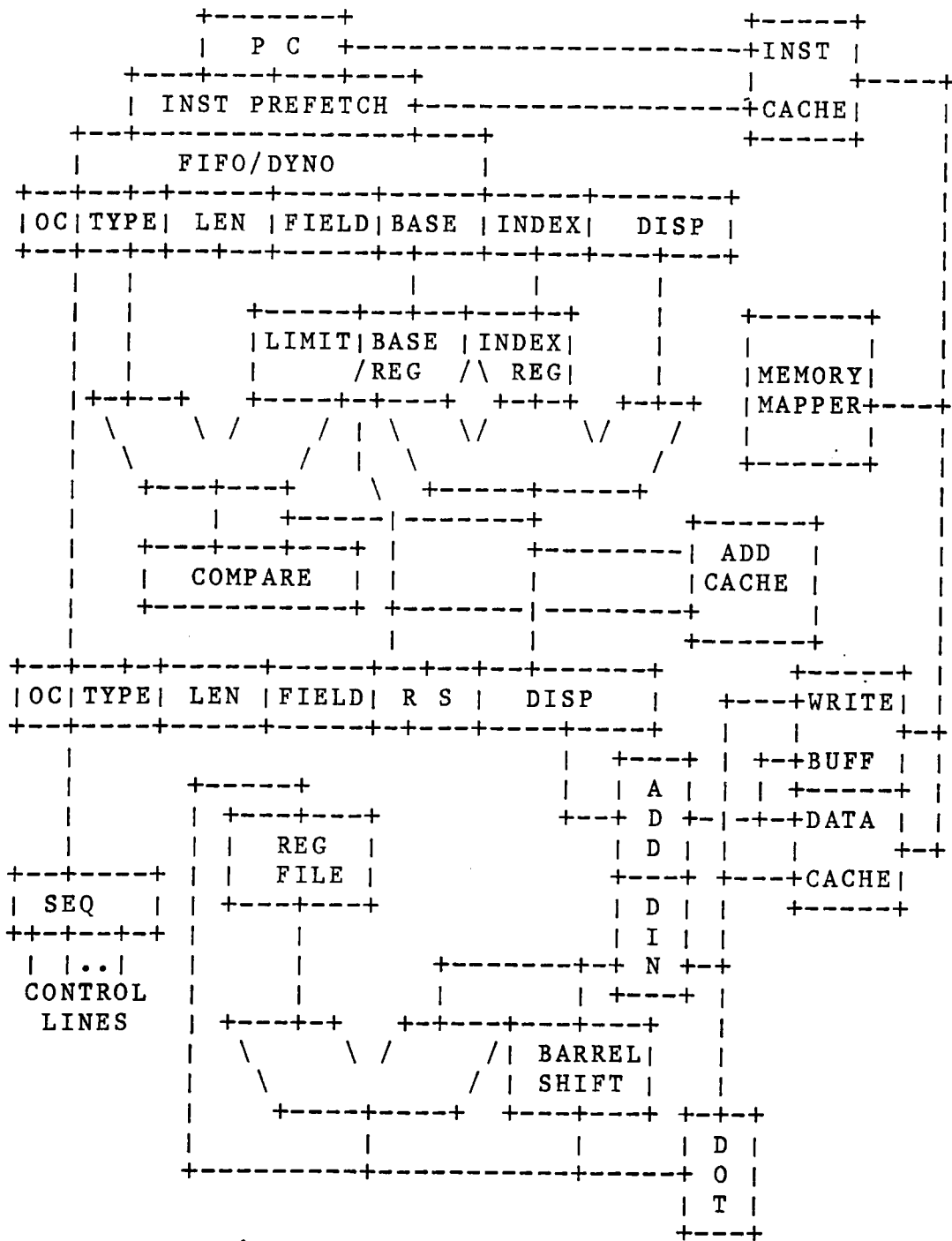


FIGURE 42 - PROCESSOR BLOCK DIAGRAM

A rough idea of the details of the design were now understood. Figure 42 shows a detailed processor block diagram. To proceed with the detailed design would conventionally consisted of drawing schematics. However since HDL had been used on the previous project and seem to be a very beneficial tool, it was decided that the design was to be detailed by simulation. This was viewed as the best approach especially considering the magnitude and complexity of the project. The simulation started with the least understood part of the design which was the FIFO and DECODER.

Simulation

The simulation was done in a top down manner rather than a bottom up. With this complicated processor, the design would be very complex. The top down approach allows each level of the pipeline to be simulated at a high level using functional descriptions. The functional description describes, in software, how the hardware will work. Once this top level of the design is completed, each block will be broken down into lower level and those levels simulated.

The first level to be simulated was the FIFO. The FIFO consists of four levels of eight byte wide registers. The output from the FIFO is eight bytes of contiguous data which includes all information needed for developing an operand specifier. Writing the functional simulation consisted of defining several variables which represent the signals and registers. Input and output variables are defined in a similar manner to reflect the actual hardware signals into and out of that block.

The hardest part of the simulation was to reflect in the functional description how the hardware should actually function. It is relatively simple to simulate a function in software. The difficult task is to write the software such that the simulation represents closely the hardware. This high level description will be expanded into lower levels until it is detailed enough to convert into hardware. Care must be taken to define the blocks and signals properly or else the top-down design could result in a functionally described parts that cannot be implemented.

The top down approach allowed a very high level description of the FIFO to be written with loops, IF

and CASE statements. The FIFO was described as an array of 32 bytes. To read eight consecutive bytes would require looping from one to eight equating the output buffer variables with the bytes in the FIFO. The output was then scheduled to occur after some time delay. HDL is similar to PASCAL in organization and syntax. Appendix M shows the HDL program which accomplishes the FIFO function.

The next step in the simulation is to break this high level description down into smaller blocks. This would consist of registers, decoders, and multiplexers. The intermediate level would again be simulated using a functional description. For an edge-triggered latch the routine would test for the proper clock edge and equate the input data with the latched data. Then the output would be scheduled to be equal to the input after some delay through the device.

The last step of the design is to break the functions down into the lowest possible level. This could be TTL packages or STL inverters for gate arrays. However, for now the top level of simulation is the only level being written.

The second level of the pipeline to be simulated was the DECODER. This was simulated with several case statements and functionally ANDing and ORing inputs to produce outputs. The HDL language then allows scheduling of the outputs from a block so that the actual timing of the hardware can be simulated. Again the top-level design must reflect the future implementation and the timings should be as close as possible to the functionality of the actual part.

The DECODER is simulated in several blocks. The first block is the state machine controller. This decodes the opcode to determine the number and type of operand specifiers which follow. It then controls the decoding circuitry to point at the proper byte in the code stream. This is designed to decode one operand specifier in every machine cycle. It has control outputs to other blocks of the DECODER as well as to the FIFO.

Project Termination

With the second block of the design finished and beginning to interconnect the FIFO and the DECODER together, the internship came to an end. The time period allocated for completion of the internship had

expired.

There was a extremely large amount of work left to be done. Yet most of the concepts of building a high-level machine had been investigated and most of the complex engineering design had been done. What was left was the "nitty gritty" work of detailing the machine and simulating it to insure its functionality.

SUMMARY

The internship experience was a very excellent learning opportunity. The experimental nature of this machine has allowed me to learn about some advanced methods of designing computer systems. All of the objectives that were possible to meet were met while others were changed due to existing circumstances of ongoing research.

During the project, a slow processor was designed which is believed to have met the goals of being easily microcoded, testable and maintainable. The data paths were designed, detailed and partially simulated. Through the simulation it was shown that part of the system would theoretically work. Yet the whole design was not completely simulated and the processor was not implemented. This resulted in not being able to get the hardware running or being able to fully microcode the machine. Instead a fast version of the same processor was designed.

The fast processor required many different areas to be investigated. A lot was learned about caching, pipelining, and instruction stream decoding. The data

paths of the fast machine were developed and some of the detailed design was finished with parts of the machine being simulated. The project did not go far enough to determine if the processor would actually meet the speed goal but from rough estimations of execution speed those goals would have been surpassed.

While all of the goals were not met, the two designs lead to a broader design experience. In the first design the goals were different from the second.

The first design resulted in understanding the problems with debugging testing a machine. Many features were designed into the machine to ensure these goals were met. Special data paths and additional hardware were used in conjunction with external test equipment to simplify the overall checkout. While designing the slow machine, a necessary understanding of the architecture was gained which was needed before starting to design the fast version of the machine.

The fast machine provided a opportunity to learn investigate the operation of fast computers. Many high performance techniques were used to speed the processor. Several different methods of implementing

caches, pipelining the processor, modifying memory bandwidths, decoding instructions operation codes and operand specifiers were investigated and designed into the high performance machine.

It is believed that more was learned by doing the two designs than if only one would have be finished. Most of the engineering research had been done on both projects. What remained was taking the design from the detailed stage to a final implementation. While this is very important, the learning opportunity afforded by doing two designs is believed to outweigh what would have been learned otherwise.

Describing the system in HDL proved to be a good method of designing the system and has many benefits over drawing schematics. There was confidence the design was correct in addition to the ease in changing and debugging the design before the hardware ever existed. This ability to assure the accuracy of the design through simulation and the ease of changing an error since it was written in software seemed to compensate for any delay due to learning to use this new system.

There are several other areas which I would liked to have had the opportunity to investigate. Marketing is very important in any project. This project had opportunities to get exposure with the marketing department, yet no such opportunity was afforded me. The same is true of other management areas I am interested in, but was never given the opportunity to get into.

In conclusion, the internship experience was very positive and beneficial. Many things were learned about engineering within TI and about how TI's management operated. It is disappointing that the projects did not result in hardware being built. Yet a lot of experience was gained in computer engineering and a understanding of how management works was gained. This will help a great deal in future work situations.

BIBLIOGRAPHY

Sources consulted during the internship.

- Habermann, A.N. Computer Hardware and Organization. Chicago: Science Research Associates, Inc., 1976.
- Hill, Fredrick J. and Peterson, Gerald R. Digital Systems: Hardware Organization and Design. New York: John Wiley and Sons, Inc., 1973,
- Rhyne, V.T. Fundamentals of Digital Systems Design. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973.
- Sloan, M.E. Computer Hardware and Organizaton. Chicago: Science Research Associates, Inc., 1976.
- Ullman, Jeffrey D. Fundamental Concepts of Programming Systems. Menlo Park, California: Addison-Wesley Publishing Company, Inc., 1976.
- The Am2900 Family Data Book. Sunnyvale, California: Advanced Micro Devices, Inc., 1979.
- The Bipolar Microprocessor Component Data Book. Dallas, Texas: Texas Instruments, Inc., 1977.
- The Bipolar Microprocessor Logic and Interface Data Book. Sunnyvale, California: Advanced Micro Devices, Inc., 1981.
- The TTL Data Book, 2nd Edition. Dallas, Texas: Texas Instruments, Inc., 1977.

APPENDICES

APPENDIX A
THE INTERNSHIP OBJECTIVES

INTERNSHIP OBJECTIVES

Introduction

A list of objectives is being submitted to the Doctor of Engineering committee members for approval in partial fulfillment of the degree requirements. It will define the scope and goals of the project and some of the problems related to them. The specific objectives will then be defined.

The Internship is with Texas Instruments, in the Digital Systems Group (DSG) located in Austin, Texas. The intern position is as a Digital Design Engineer. The job description includes the design, debug and checkout of a digital system. This will require the studying of many problems, making the proper tradeoffs, and implementing a design.

Project Definition

The basic project is to design a experimental processor. This processor is to be a very general purpose machine with several basic properties. First,

it is to be a microcode driven machine with writable control store. The machine must be easily and quickly microcoded and then be easily debugged. The microcode is the lowest level of programming a processor. Another key factor behind the design is the testability and maintainability of the processor. The system must be highly testable for the initial checkout. After the processor is running, it must be maintainable and diagnosable. This is necessary to check that all parts are working correctly with high reliability.

Project Objectives

The first objective is to design a machine that is easily microcoded. The general philosophy behind this is to do as much as possible in software. All of the microcoding is to be done from a remote terminal to the system. This means that a host computer will be used with an umbilical cord into the processor. This umbilical cord will have full control over the machine. After the initial checkout, all testing should be done from the host computer with a minimal hardware probing needed. There should be no hardware manipulation needed to load, debug and modify the microcode.

The second objective is testability. microcode diagnostics are important to allow a major part of the testing to be done in software. This will require a minimal of hardware probing to locate a problem. This test philosophy requires a minimal amount of technician support and a maximum amount of self test and self diagnosis. The driving philosophy is for the diagnostics to isolate the problem down to the smallest replaceable part. A fully testable machine is very expensive and the cost-benefits must be weighed to determine what level of testability is to be implemented.

There are several smaller, short term design objectives which will be accomplished. These objectives will lead to the the implementation of the project objectives. The following could be viewed as a step by step method of accomplishing the project objective.

Design Objectives

There are several milestones which must be reached on the way to a complete, working processor. First the problem must be fully understood. This is necessary

before the objectives can be properly defined. This has been the major task of the last months work. With the problem well understood, a set of specific objectives can be formulated.

The first design objective is to develop the data paths for the processor. This means that all paths for the flow of data through the processor must be determined and checked out such that any data manipulation needed can be done. Since this is to be a general purpose microcoded machine, those data paths cannot be special purpose but rather very general to handle a wide variety of applications. This is the first step in the processor design.

The second design objective is the detailed digital design. This part will take the data paths and put them onto paper in the form of an actual hardware design. This will require the selecting of appropriate parts, studying the timings and interconnecting the parts appropriately. The output of this phase will be a data base which will define how the system is interconnected.

The third design objective is to simulate the design. The simulation will show the detailed timing and confirm the design before it is committed to hardware. This will result in working hardware with few or no iteration since the major checkout is done in the simulation.

The fourth design objective is to get the hardware running. This will require initial power-up and debug of the system. This will utilize the appropriate equipment- scopes, data analyzers, etc. A major portion of the debug is to be done with a host computer. This host computer will be able to read all busses and trace various signal within the processor.

The last objective is to begin the microcode of the machine. It is an objective to verify that the execution of any microcode state caused the expected results. This is intended to support the actual microcoding of the machine and insure that the hardware is functioning properly.

Summary

The objectives of the Doctor of Engineering Internship of Kerry C. Glover with Texas Instruments have been presented. The major overall objective is the design of a experimental processor. This includes the development of a system which is inherently testable, easily microcoded and supported. The design objectives are to specify the data paths, do the detailed digital design, simulate that design, debug the hardware and support the initial microcode effort.

APPENDIX B
DEFINITION OF MICROCODE

MICROCODE DEFINITION

The following is a definition and summary of the microcode

1. Arithmetic and Logic Unit (ALU)
2. CONStant field low or high(CON)
3. SHiFt control (SHF)
4. Condition Code and Carry (CCC)
5. MEMory field (MEM)
6. Memory MAP request field (MAP)
7. SEQuencer field (SEQ)

SUMMARY

ALL

```

332  2   2   22  2
109  8   7   65  4
XXX  X   X   XX  X
LFD  LCC IPC ECW BPT

```

ALU

```

      22  1   1   1   1   1111 1100 0000 0000
      10  9   8   7   6   5432 1098 7654 3210
      XX  X   X   X   X   XXXX  XXXX XXXX XXXX
SCY XXX AIN AIX L/A  ALU  BREG AREG XREG

```

CON

```

      2   2   11  11 1111111000000000
      1   0   98  76 5432109876543210
      X   X   XX  XX XXXXXXXXXXXXXXXXXXXX
LCH LCL SCH  SCL  CONSTANT

```

SHF

```

      0  0   000 0000
      8  7   654 3210
      X  X   XXX XXXX
SE SLR SALU SSC

```

CCC

1	111100	0	0	0	00000
4	321098	7	6	5	43210
X	XXXXXX	X	X	X	XXXXX
ECT	SRS	CEU	CEM	OEY	SCC

MEM

2	22	1	111	1	111	1100	00000000
2	10	9	876	5	432	1098	76543210
X	XX	X	XXX	X	XXX	XXXX	XXXXXXXXXX
WMF	SC	MIX	LCT	I/D	ECT	RMD	AMF

MAP

1	1	111111000000	00	0	0
7	6	543210987654	32	1	0
X	X	XXXXXXXXXXXXXX	XX	X	X
ESEX	STM		WLI	BSEX	HSEX

SEQ

2	2	2	2	1	1	1	1	1111110000000000
3	2	1	0	9	8	7	6	5432109876543210
X	X	X	X	X	X	X	X	XXXXXXXXXXXXXXXXXX
DLC1	DLC0	OCB	BCC	SI	SO	FE	PUP	SADD

ALL

332	2	2	22	2
109	8	7	65	4
XXX	X	X	XX	X
LFD	LCC IPC ECW BPT			

LFD	LOAD FIELD			
001	LCON	LOAD CON UCODE REG LOW OR HIGH		
010	LSHF	LOAD SHIFT/CONTROL		
011	LCCC	LOAD COND CODE AND CARRY		
100	LALU	LOAD ALU UCODE REG		
101	LMEM	LOAD MEM UCODE REG		
110	LMAP	LOAD MAP UCODE REG		
111	LSEQ	LOAD SEQ UCODE REG		

LCC	LOAD CONDITION CODE REGISTER			
IPC	INCREMENT PROGRAM COUNTER			

ECW	ENABLE CONSTANT OR WCS			
00	E00	ENABLE 00 BITS OF CONSTANT		
01	E08	ENABLE 08 BITS OF CONSTANT		
10	E16	ENABLE 16 BITS OF CONSTANT		
11	E24	ENABLE 24 BITS OF CONSTANT		

BPT	BREAKPOINT			
-----	------------	--	--	--

ALU

22	1	1	1	1	1111	1100	0000	0000
10	9	8	7	6	5432	1098	7654	3210
XX	X	X	X	X	XXXX	XXXX	XXXX	XXXX
SCY	XXX	AIN	AIX	L/A	ALU	BREG	AREG	XREG

SCY	SELECT CARRY INPUT (I12 AND I11 ON 2904)							
AIN	A INPUT IS INTERNAL REGISTER FILE							
AIX	ALU WRITE INTERNAL OR EXTERNAL							
L/A	LOGICAL OR ARITHMETIC FUNCTION							
IALU	ALU INSTRUCTION							
BREG	B REGISTER FILE ADDRESS							
AREG	A REGISTER FILE ADDRESS							
XREG	X REGISTER FILE ADDRESS							

ALU DESTINATION

REG#	AIX=0	AIX=1	
XXXX	BREG=XXXX	XREG=XXXX	
0000	BREG 0	LBUG	LOAD DEBUG
0001	BREG 1	LTIM	LOAD TIME
0010	BREG 2	LPC	LOAD PC
0011	BREG 3	LCO	LOAD CON
0100	BREG 4	LMDR	LOAD MDR
0101	BREG 5	LIQ	LOAD IQ
0110	BREG 6	LBPO	LOAD BPO
0111	BREG 7	LBP1	LOAD BP1
1000	BREG 8	LLOP1	LOAD LOOP COUNTER 1
1001	BREG 9	LLOP2	LOAD LOOP COUNTER 2
1010	BREG A		
1011	BREG B		
1100	BREG C		
1101	BREG D		
1110	BREG E		
1111	BREG F	DUMMY	LOAD NOTHING

ALU A SIDE SOURCE SELECT

AREG	AIN=0	AIN=1	
XXXX			
0000	AREG 0	EBUG	ENABLE BUG
0001	AREG 1	ETIM	ENABLE TIME
0010	AREG 2	EPC	ENABLE PC
0011	AREG 3	ECI	ENABLE CONSTANT
0100	AREG 4	ECC	ENABLE CONDITION CODE
0101	AREG 5	EIQ	ENABLE INSTRUCTION QUEUE
0110	AREG 6	EMC	ENABLE MEMORY COUNTER
0111	AREG 7		
1000	AREG 8	ROT0	ENABLE ROT 0
1001	AREG 9	ROT1	ENABLE ROT 1
1010	AREG A	ROT2	ENABLE ROT 2
1011	AREG B	ROT3	ENABLE ROT 3
1100	AREG C	ROT4	ENABLE ROT 4
1101	AREG D	ROT5	ENABLE ROT 5
1110	AREG E	ROT6	ENABLE ROT 6
1111	AREG F	ROT7	ENABLE ROT 7

CON

```

1 1 11 11 1111110000000000
1 0 98 76 5432109876543210

```

```

X X XX XX XXXXXXXXXXXXXXXX
LCH LCL SCH SCL CONSTANT

```

```

LCH LOAD CONSTANT HIGH
LCL LOAD CONSTANT LOW
SCH SELECT CONSTANT HIGH
SCL SELECT CONSTANT LOW

```

```

00 FILL LOWER/UPPER BITS WITH ZERO'S
01 FILL LOWER/UPPER BITS WITH ONE'S
10 SELECT CONTROL STORE
11 SELECT CONTROL STORE

```

SHF

```

0 0 000 0000
8 7 654 3210
X X XXX XXXX
SE SLR SALU SSC

```

```

SE SHIFT ENABLE
SLR SELECT LEFT OR RIGHT (I10 OF 2904 AND I8 OF 2903)
SALU INSTRUCTION FOR ALU I5-I7 OF 2903
SSC SELECT SHIFT CONTROL I6-I9 OF 2904

```

CCC

1	111100	0	0	0	00000
4	321098	7	6	5	43210
X	XXXXXX	X	X	X	XXXXX
ECT	SRS	CEU	CEW	CEY	SCC/SM

ECT ENABLE CONDITION TEST
 SRS STATUS REGISTER SELECT I0-I5 OF 2904
 CEU CONTROL ENABLE FOR MICRO STATUS REGISTER
 CEM CONTROL ENABLE FOR MACHINE STATUS REGISTER
 CEY CONTROL ENABLE FOR STATUS READ OR WRITE
 SCC/SM SELECT CONDITION CODE AND STATUS MASK

SCC

00000	ONE	+5 VDC
00001	MBUSY	MEMORY BUSY (0=ACTIVE,1=TERMINATED)(INT)
00010	PCERR	PROGRAM COUNTER ERROR
00011	MERR	MEMORY ERROR (MISC1)
00100	UFL	UNDERFLOW OCCURRED
00101	TRNC	TRUNCATION OCCURRED
00110	AUX	AUXILARY
00111	ROUND	ROUNDING OCCURRED
01000	LCZRO	LOOP COUNTER ZERO

OTHER CONDITIONS ARE CHECHED VIA 2904
 ECT AND SRS FIELDS. THESE SAME CONDITIONS
 MAY HAVE THREE(3) DIFFERENT SOURCES:

- 1) MICRO STATUS REG (USR)
- 2) MACHINE STATUS REG (MSR)
- 3) ALU STATUS (ASR)

CONDITIONS POSSIBLE ARE:

EQU	EQUAL	:
NEQU	NOT EQUAL	:
GTE	GREATER THAN OR EQUAL	:THESE CONDITIONS
LTE	LESS THAN OR EQUAL	:RESULT FROM COMPARE
LT	LESS THAN	:
GT	GREATER THAN	:

ZRO	ZERO
NZRO	NOT ZERO
CAR	CARRY
NCAR	NOT CARRY
OVR	OVERFLOW
NOVR	NOT OVERFLOW
SIGN	SIGN
NSIGN	NOT SIGN

MEM

```

2  22  1  111  1  111  1100  00000000
2  10  9  876  5  432  1098  76543210

X  XX  X  XXX  X  XXX  XXXX  XXXXXXXX
WMF SC MIX LCT I/D ECT RMD      AMF

```

```

WMF  WRITE MAR INTO MEMORY
MIX  MEMORY REGISTER WRITE INTERNAL OR EXTERNAL

```

```

LCT  LOAD COUNTER

```

```

100  LOAD A-COUNTER
010  LOAD B-COUNTER
001  LOAD M-COUNTER

```

```

I/D  INCREMENT OR DECREMENT

```

```

ECT  ENABLE COUNTER TO COUNT

```

```

100  ENABLE A-COUNTER
010  ENABLE B-COUNTER
001  ENABLE M-COUNTER

```

```

SC   SELECT MEMORY COUNTER

```

```

00   SAC      SELECT A COUNTER
01   SBC      SELECT B COUNTER
10   SMC      SELECT MAP COUNTER
11   SAD      SELECT ADDRESS DIRECT

```

READ MEMORY DESTINATION

REG#	MIX=0	MIX=1
XXXX	RMD=XXXX	RMD=XXXX
0000	BREG 0	LBUG LOAD DEBUG
0001	BREG 1	LTIM LOAD TIME
0010	BREG 2	LPC LOAD PC
0011	BREG 3	LCON LOAD CON
0100	BREG 4	LMDR LOAD MDR
0101	BREG 5	LIQ LOAD IQ
0110	BREG 6	LBPO LOAD BP 0
0111	BREG 7	LBP1 LOAD BP 1
1000	BREG 8	LLOP1 LOAD LOOP COUNTER 1
1001	BREG 9	LLOP2 LOAD LOOP COUNTER 2
1010	BREG A	
1011	BREG B	
1100	BREG C	
1101	BREG D	
1110	BREG E	
1111	BREG F	DUMMY LOAD NOTHING

AMF ADDRESS MEMORY FILE

MAP

1	1	111111000000	00	0	0
7	6	543210987654	32	1	0
X	X	XXXXXXXXXXXXX	XX	X	X
ESEX	STM		WLI	BSEX	HSEX

ESEX	ENABLE SIGN EXTENSION
STM	START MEMORY CYCLE
WLI	WS, LS, INST ACCESS
BSEX	BYTE SIGN EXTEND
HSEX	HALFWORD SIGN EXTEND

WLI	FETCH TYPE
-----	------------

00	FINS	FETCH INSTRUCTION
01	FWSD	FETCH WORKSPACE DATA
10	FLSD	FETCH LINKAGE SECTION DATA
11	FMIS	FETCH MISC

SEQ

2	2	2	2	1	1	1	1	1111110000000000
3	2	1	0	9	8	7	6	5432109876543210
X	X	X	X	X	X	X	X	XXXXXXXXXXXXXXXXXX
DLC1	DLC0	OCB	BCC	S1	SO	FE	PUP	SADD

DLC1	DECREMENT LOOP COUNTER 1
DLC0	DECREMENT LOOP COUNTER 0
OCB	OPCODE BRANCH
BCC	BRANCH ON CONDITION CODE
LRE	LOAD SEQUENCER REGISTER
SO	SELECT 0
S1	SELECT 1
PUP	PUSH/POP STACK
FE	FILE ENABLE

SADD	SEQUENCER NEXT ADDRESS
------	------------------------

APPENDIX C
SAMPLE MICROCODE ROUTINE

* MAIN

```

      INC      PC          ; INCREMENT PC
      TEST    PCER        ; TEST FOR IQ EMPTY
      CEQ     PCHDLR      ; CALL PC HANDLER IF EMPTY
      JREL    OCTBL,IQ    ; JUMP RELATIVE OCTBL + IQ
      :
      :

```

* TABLE OF OPCODE SPECIFIER JUMP VECTORS

```

OCTBL  :
      :
+IQ    JMP     ADD2       ; JUMP TO ADD S,SD ROUTINE
      :
      :

```

* ROUTINE TO ADD S,SD

```

ADD2   MOV     OPA0,MC    ; MOVE OPERAND SPECIFIER
      INC     PC          ; ADDRESS TO MEMORY COUNTER
      TEST    PCER
      CEQ     PCHDLR
      CALL    DOPS        ; CALL DECODE OPERAND SPEC
      MOV     OPA1,MC    ; MOVE OPERAND SPECIFIER
      INC     PC          ; ADDRESS TO MEMORY COUNTER
      TEST    PCER
      CEQ     PCHDLR
      CALL    DOPS
      MOV     OPA0+2,MRA ; MOVE OPS DATA 0 TO MRA
      MOV     OPA1+2,MRB ; MOVE OPS DATA 1 TO MRB
      ADD     MRA,MRB,MAR ; ADD MRA TO MRB SAVE TO MAR
      MOV     MAR,OPA1+2 ; MOVE RESULTS TO OPS 1 DATA
      MOV     OPA1,AC,MC ; MOVE OPS TO A CTR, MEM CTR
      CALL    TRUN        ; CALL TEST FOR TRUNCATE
      WRITE
      JMP     MAIN

```

* ROUTINE TO DECODE AND JUMP TO OP SPECIFIER ROUTINE

```

DOPS   JREL    OSTBL,IQ  ; JUMP RELATIVE TO OSTBL + IQ
      :
      :

```

* TABLE FOR OPERAND SPECIFIER JUMP VECTOR

```

OSTBL  :
      :
+IQ    JMP     SHWS       ; JUMP TO SHORT WORKSPACE OPS
      :

```

* ROUTINE TO FETCH SHORT WORKSPACE

```

SHWS   MVI     00,CHI    ; MOVE 00 TO CONSTANT HI FLD

```

```

MVI      OF,CLO      ; MOVE OF TO CONSTANT LO FLD
AND      IDR,CON,RO  ; MASK IDR/CON STORE IN RO
ADD      RO,WSP,MAR  ; ADD RO TO WS PTR STORE MAR
MOV      MAR,MCI     ; MOV MAR TO MEM CTR LOC
MVI      TYP,CLO    ; MOV IMM THE TYPE TO CLO
MOV      CON,MAR     ; MOV TYPE TO MAR
MOV      MAR,MCD     ; MOV TYPE TO MEM CNTR LOC
TEST     MBZ        ; TEST MAP BUZY BIT
JEQ      *-1        ; HOLD UNTIL READY
READ     ; START READ INSTRUCTION
RETURN   ; RETURN

```

* PC HANDLER

```

PCHDLR  MOV      PCN+2,MRA ; MOV NEXT PC WORD ADD TO MRA
        MOV      MRA,IQ   ; LOAD THE INSTRUCTION QUEUE
        ADD      4,PC,MAR ; ADD FOUR TO PC STORE MAR
        MOV      MAR,PCN+1 ; STORE ADD OF NEXT PC WORD
        MOV      PCN,MC   ; MOV PC NEXT ADD-MEM CNTR
        TEST     MBZ      ; TEST MEM BUZY BIT
        JEQ      *-1     ; HOLD UNTIL MEM NOT BUZY
        READ     ; START READ CYCLE
        RETURN

```

* TRUNCATION TESTER

```

TRUN    MOV      ACI,MRA   ; MOVE A CNTR AND INC TO MRA
        MOV      03F,CLO  ; MOVE 03F TO CLO
        AND      MRA,CON,RO ; AND MRA WITH CON STORE RO
        MOV      TRTBL,CLO ; MOVE TRUN TABLE BASE TO CLO
        ADD      RO,CON,CON ; ADD TYPE/TBL BASE STORE CON
        JMP      E16      ; JUMP TO TRTBL + TYPE

```

* TABLE FOR TRUNCATION JUMP VECTOR

```

TRTBL   :
        :
        +TYPE JUMP      TRHALF
                                ; JUMP TO TRUN HALFWORD CHECK

```

* TRUNCATE HALFWORD RESULTS

```

TRHALF  MOV      ACI,MRA   ; INCREMENT A COUNTER
        MOV      AC,MRA   ; MOVE A CNTR TO MRE
        MOV      0,CLO
        MOV      0,CHI
        CMP      MRA,CON   ; CMP MRA TO ZERO
        TEST     ZERO     ; TEST FOR ZERO
        JEQ      NTR      ; JUMP IF ZERO TO NO TRUNCATE
        MOV      FFFF,CHI  ; MOV FFFF TO CON
        CMP      MRA,CON   ; CMP TO NEGATIVE RESULTS
        TEST     ZERO     ; TEST FOR ZERO
        JEQ      NTR      ; JUMP IF ZERO TO NO TRUNCATE
        MOV      TRMK,CLO  ; MOVE TRUNCATE MASK TO CLO
        MOV      0,CHI

```

```
AND      STAT,CON,STAT
RETURN   ; SET TRUN BIT IN STAT REG
NTR      MOV      NTRMK,CLO ; MOVE NO TRUN MASK TO CLO
MOV      0,CHI
AND      STAT,CON,STAT
RETURN   ; CLEAR TRUN BIT IN STAT REG
```

APPENDIX D
PROCESSOR/SEQUENCER INTERFACE

THE FOLLOWING WILL DEFINE THE INTERFACE BETWEEN THE PROCESSOR AND SEQUENCER BOARD. SOME SIGNALS WILL BE GLOBAL WHILE OTHERS WILL BE ONLY BETWEEN BOARDS.

****GLOBAL SIGNALS****

CLKA	1	CLOCK PHASE A	
CLKB	1	CLOCK PHASE B	
SRS	1	SYSTEM RUN STOP	(MUST BE SYNCRONIZED)
SSS	1	SYSTEM SINGLE STEP	(MUST BE SYNCRONIZED)

****LOCAL SIGNALS FROM PROCESSOR TO SEQUENCER****

CC	1	CONDITION CODE BIT	
INST	8	INSTRUCTION	
CS	32	CONTROL STORE BUS	

****LOCAL SIGNALS FROM SEQUENCER TO PROCESSOR****

CLK1	1	CLOCK PHASE 1	
CLK2	1	CLOCK PHASE 2	
CLK3	1	CLOCK PHASE 3	
CLK4	1	CLOCK PHASE 4	
SDIR	1	SELECT DIRECTION OF CONTROL STORE BUS	
RBHL	1	READ BUG HIGH OR LOW (SAME BIT AS BELOW)	
WPDH	1	WRITE PROCESSOR DATA LATCH HIGH	
WPDH	1	WRITE PROCESSOR DATA LATCH LOW	

APPENDIX E
DEBUG INTERFACE SIGNALS

MEMO DESCRIBING THE DEBUG FEATURES AND INTERFACE TO THE BREADBOARD SEQUENCER.

DESIGN OF THE SEQUENCER TO 16 I/O INTERFACE

THE FOLLOWING WILL SHOW THE DETAILS OF HOW TO DESIGN THE INTERFACE OF THE SEQUENCER BOARD WITH THE 16 I/O MODULE. THIS IS OF SIMILAR FORMAT AND REFERS TO THE MEMO OF MAY 27.

FIRST 16 I/O MODULE

SIN I 16 DATA INPUT LINES FROM WCS BOARD TO I/O MODULE

THESE 16 LINES ARE DEFINED AS OUTPUT LINES FROM THE WCS BOARD AND INPUT LINES TO AMPL I/O MODULE. THE VIKING NUMBERING SYSTEM IS LITTLE INDIAN BUT AMPL IS BIG INDIAN. THEREFORE ON THE AMPL END OF THE CABLE THE MSB WILL BE BIT 0 AND ON THE VIKING END OF THE CABLE THE MSB WILL BE BIT 15. LIKEWISE ON THE AMPL END OF THE CABLE THE LSB WILL BE NUMBERED BIT 15 AND ON THE VIKING END OF THE CABLE THE LSB WILL BE NUMBERED BIT 0. THIS IS SO THAT WHEN A VALUE IS DISPLAYED ON EITHER SYSTEM IT WILL APPEAR THE SAME.

SOUT O 16 DATA OUTPUT LINES TO WCS BOARD FROM I/O MODULE

THESES 16 LINES ARE DEFINED AS INPUT LINES TO THE WCS BOARD ANT OUTPUT LINES FROM THE AMPL I/O MOD. LIKEWISE THE NUMBERING SYSTEM WILL BE ON AMPL BIT 0 AND 15 AS MSB AND LSB. ON THE VIKING END OF THE CABLE THE SAME BITS WILL BE NUMBERED 15 AND 0 AS MSB AND LSB.

SECOND I/O MODULE

PIN I 16 DATA INPUT LINES FROM PROCESSOR BOARD TO I/O MODULE. REFER TO ABOVE

POUT O 16 DATA OUTPUT LINES TO PROCESSOR BOARD FROM I/O MODULE. REFER TO ABOVE

THIRD I/O MODULE

VIK AMPL

NAME	I/O	PIN	PIN	DESCRIPTION
SWA	0	0	15	SELECT WRITABLE CONTROL STORE ADDRESS SOURCE.

THIS SELECTS THE SOURCE ADDRESS OF THE WCS AS COMING FROM A COUNTER OR THE 2911. A VALUE OF ZERO SELECTS THE COUNTER AND A ONE SELECTS THE 2911.

SDIR	0	1	14	SELECT DIRECTION OF CONTROL STORE BUSS.
------	---	---	----	---

THIS SELECTS THE DIRECTION OF THE BUFFER FROM THE SEQUENCER BOARD TO THE PROCESSOR BOARD. SDIR=1 DRIVES DATA FROM SEQUENCER TO PROCESSOR. SDIR=0 DIRVES DATA FROM PROCESSOR TO SEQUENCER. THIS ALLOWS DATA FROM THE PROCESSOR TO BE READ OR MANIPULATED ON THE SEQUENCER BOARD. NOTE THAT THIS IS IN COMBINATION WITH THE ECW FIELD OF THE UCODE WHICH ENABLES DIFFERENT BYTES FROM THE PROCESSOR TO THE SEQUENCER BOARD.

SHS	0	2	13	SELECT HARD OR SOFT BREAKPOINT
-----	---	---	----	--------------------------------

THIS BIT IS USED TO DETERMINE WHETHER THE BREAKPOINTS ARE TO BE HARD IE AN INPUT TO A AMPL TRACE MODULE WHICH WILL RESULT IN THE SYSTEM BEING HALTED OR WHETHER IT IS A SOFT BREAKPOINT FOR USE AS A TRIGGER IN A SCOPE OR BIOMATION TRACE LOOP. SHS=1 SELECTS A HARD BREAKPOINT AND SHS=0 SELECTS A SOFT BREAKPOINT.

SRS	0	3	12	SYSTEM RUN/STOP
-----	---	---	----	-----------------

THIS SIGNAL IS THE RUN/STOP SIGNAL FOR THE SYSTEM. THIS GOES TO ALL BOARDS AND WILL CAUSE ALL BOARDS TO BE HALTED AND READY FOR A SYSTEM SINGLE STEP. SRS=1 SELECTS RUN STATE AND SRS=0 SELECTS A STOP STATE.

SSS	0	4	11	SYSTEM SINGLE STEP
-----	---	---	----	--------------------

CAUSES A SINGLE STEP ON THE SYSTEM. THIS HAS BEEN DEFINED AS A CYCLE ON ALL FOUR PHASES OF THE CLOCK BOTH RISING AND FALLING EDGES. THE CYCLE STOPS BETWEEN PHASE 4 AND PHASE 1 WITH ALL TRUE CLOCKS LOW. A SINGLE STEP OCCURS ON THE RISING EDGE OF SSS.

PRS 0 5 10 PROCESSOR RUN/STOP

SIMILAR TO SYSTEM RUN/STOP EXCEPT THAT IT STOPS ONLY THE PROCESSOR BOARD. PRS=1 SELECTS THE RUN STATE AND PRS=0 SELECTS THE STOP STATE.

PSS 0 6 9 PROCESSOR SINGLE STEP

SIMILAR TO SYSTEM SINGLE STEP EXCEPT THAT IT SINGLE STEPS ONLY THE PROCESSOR BOARD. A SINGLE STEP OCCURS ON THE RISING EDGE OF SSS.

ZERO 0 7 8 FORCE ZERO TO 2911 OUTPUT (RESET)

ZERO INPUT ON THE 2911. FORCES A ZERO ON THE OUTPUTS OF THE 2911 ADDRESS LINES. THE FIRST INSTRUCTION SHOULD BE A JUMP TO SOME LOCATION TO GET THE SEQUENCER STARTED AT A KNOWN LOCATION. ZERO=0 FORCES THE OUTPUT TO A ZERO.

RBHL 0 8 7 READ BUG HIGH OR LOW

ENABLES AMPL TO READ OR WRITE THE HIGH LOW HALFWORD OF THE DEBUG LATCH. A RBHL=1 ENABLES THE HIGH HALFWORD AND A RBHL=0 ENABLES THE LOW HALFWORD.

RAMP 0 98 67 READ AMPL
RMS 00 READ MISC STATUS

ENABLES THE MISC STATUS TO BE READ FROM THE SEQUENCER BOARD OVER THE AMPL MODULE.

RAB 01 READ WCS ADDRESS BUSS

ENABLES THE WCS ADDRESS TO BE READ FROM THE SEQUENCER BOARD OVER THE AMPL MODULE.

RCH 10 READ CS HIGH BUSS

ENABLES THE CS HIGH DATA TO BE READ FROM THE SEQUENCER BOARD OVER THE AMPL MODULE.

RCL 11 READ CS LOW BUSS

ENABLES THE CS LOW DATA TO BE READ FROM THE SEQUENCER BOARD OVER THE AMPL MODULE.

WAMP 012-10 3-5 WRITE AMPL

DEFINES WHICH REGISTER THE DATA WHICH IS
ON THE AMPLE OUTPUT LINES IS TO BE WRITTEN.

WCL	000	WRITE COMPARE LATCH
WAC	001	WRITE WCS ADDRESS COUNTER
WWDH	010	WRITE WCS DATA HIGH LATCH
WDDL	011	WRITE WCS DATA LOW LATCH
WPDH	100	WRITE PROCESSOR DATA LATCH HIGH
WDDL	101	WRITE PROCESSOR DATA LATCH LOW
WWCS	110	WRITE WRITABLE CONTROL STORE
NWRT	111	NO WRITE (MUST END HERE)
EWRT	0 13 2	ENABLE WRITE

THIS LINE SHOULD BE TAKEN LOW AND THEN
HIGH TO WRITE DATA INTO THE APPROPRIATE
REGISTER AS DESCRIBED ABOVE.

IWC 0 14 1 INCREMENT WRITABLE CONTROL COUNTER

INCREMENT COUNTER USED WHEN DOWN-LOADING
THE WCS FROM AMPL. THIS LINE SHOULD BE
TAKEN LOW AND THEN HIGH TO COUNT.

- * O INDICATED OUTPUT LINES
- * I INDICATED INPUT LINES

APPENDIX F
SOFTWARE DEBUG ROUTINES

SOFTWARE ROUTINES FOR PROCESSOR

THE PROCESSOR BOARD WILL REQUIRE SEVERAL SOFTWARE ROUTINES TO HELP IN THE DEBUG AND UCODE CHECKOUT PROCEDURES THE FOLLOWING IS A LIST OF SOME OF THE NEEDED ROUTINES.

RST	RESET
RUN	SYSTEM RUN
STOP	SYSTEM STOP
SSS	SYSTEM SINGLE STEP
PRUN	PROCESSOR RUN
PSTP	PROCESSOR STOP
PSS	PROCESSOR SINGLE STEP
DREG	DISPLAY CPU REGISTERS
MREG	MODIFY A CPU REGISTER
DMEM	DISPLAY A MEM LOCATION
MMEM	MODIFY A MEM LOCATION
DWCS	DISPLAY WCS ADDRESS AND DATA
MWCS	MODIFY WCS ADDRESS AND/OR DATA
LWCS	LOAD WCS
VWCS	VERIFY WCS
SBPT	SET A BREAKPOINT
BPMS	BREAKPOINT HARD OR SOFT
DALL	DISPLAY CPU STATE BETWEEN EVERY INSTRUCTION
DBPT	DISPLAY CPU STATE AT EACH BREAKPOINT

THE PREVIOUS PROCS MAY HAVE SEVERAL OPTIONS SUCH AS SAVE DISPLAYED DATA ONTO DISK FILE FOR LATTER REFERENCE.

IN ADDITION SEVERAL TEST ROUTINES ARE NEEDED TO CHECK THE SYSTEM OUT TO INSURE IT IS OPERATION CORRECTLY.

TMEM	TEST MEMORY
TALU	TEST ALU
TCCC	TEST CCC
TSEQ	TEST SEQ
TWCS	TEST WCS

APPENDIX G
PASCAL EMULATOR

```

PROGRAM EMULATE;
  TYPE BYTE=0..#FF;
        F30=0..#3FFFFFFF;
        F24=0..#FFFFFFF;
        F15=0..#3FFF;
        F12=0..#FFF;
        F10=0..#3FF;
        F9=0..#1FF;
        F7=0..#7F;
        F6=0..#3F;
        F5=0..#1F;
        F4=0..#F;
        F3=0..7;
        F2=0..3;
        F1=0..1;

WORD=PACKED RECORD
CASE INTEGER OF
  1:(CS:LONGINT);
  2:(LFD:F3;LCC:F1;IPC:F1;ECW:F2;BPT:F1;
    CASE INTEGER OF
      1:(DTC1:F2;SCY:F2;DTC11:F1;AIN:F1;AIX:F1;LA:F1;
        IALU:F4;BREG:F4;AREG:F4;XREG:F4;);
      2:(DTC2:F4;LHI:F1;LLO:F1;SHI:F1;SLO:F1;
        FHI:F1;FLO:F1;CON:INTEGER;);
      3:(DTC3:F15;SE:F1;SLR:F1;SALU:F3;SSC:F4;);
      4:(DTC4:F9;ECT:F1;SRS:F6;CEU:F1;CEM:F1;CEY:F1;
        SCC:F5;);
      5:(DTC5:F10;SRSEL:F2;SR:F4;DTC88:F4;YOVR:F1;
        YN:F1;YC:F1;YZ:F1;);
      6:(DTC6:F1;WMF:F1;SC:F2;MIX:F1;LCT:F3;ID:F1;
        EC:F3;RMD:F4;AMF:BYTE;);
      7:(DTC7:F6;ESEX:F1;STM:F1;TAG:F12;WLI:F2;
        BSEX:F1;HSEX:F1;);
      8:(DLC1:F1;DLC0:F1;OCB:F1;BCC:F1;S1:F1;S0:F1;
        FE:F1;PUP:F1;SADD:INTEGER;););
  3:(W:LONGINT);
  4:(H:PACKED ARRAY [0..1] OF INTEGER);
  5:(B:PACKED ARRAY [0..3] OF BYTE);
  6:(N:PACKED ARRAY [0..7] OF F4);
  7:(T:PACKED ARRAY [0..15] OF F2);
  8:(I:PACKED ARRAY [0..31] OF F1);
  9:(S:SET OF 0..31);
  10:(SEG:BYTE;OFFSET:F24);
  11:(DTC8:F24;DTC9:F1;SELPC:F3);
  12:(ALL:BYTE;FCN:F24);
END;

```



```

PIPE=PACKED RECORD
  CASE INTEGER OF
    1:(B:PACKED ARRAY [0..11] OF BYTE);
    2:(W:PACKED ARRAY [0..2] OF LONGINT);
    3:(ROTO:LONGINT;ROT4:LONGINT);
    4:(DMY0:BYTE;ROT1:LONGINT;ROT5:LONGINT);
    5:(DMY1:INTEGER;ROT2:LONGINT;ROT6:LONGINT);
    6:(DMY2:F24;ROT3:LONGINT;ROT7:LONGINT);
  END;

```

```

HALF=PACKED RECORD
  CASE INTEGER OF
    1:(H:INTEGER);
    2:(B:PACKED ARRAY [0..1] OF BYTE);
  END;

```

```

STR8=PACKED ARRAY [1..8] OF CHAR;

```

```

LINE=PACKED RECORD
  CASE INTEGER OF
    1:(L:PACKED ARRAY [1..80] OF CHAR);
    2:(W:PACKED ARRAY [1..10] OF STR8);
  END;

```

```

VAR  PWR,MM,ROM
      :ARRAY [0..31] OF LONGINT;
REG
      :ARRAY [0..15] OF LONGINT;
STK
      :ARRAY [0..3] OF INTEGER;
CS,PC,CO,CI,CC,IAB,IBB
      :WORD;
IYB,IQR,F
      :WORD;
SADD,SB
      :HALF;
OPC,PCE,AC,BC,MC,MA,LEV
      :INTEGER;
CN,CCB,CYCLE,CHI,CLO,YB,QR
      :INTEGER;
DAB,DBB,BUG,TIM,MDR,RDR
      :LONGINT;
ALUL,MEML,SEQL
      :F24;
IQ,BP
      :PIPE;
KEYBD
      :TEXT;
INKEY
      :CHAR;

```

```

SCREEN
  :ARRAY [1..24] OF LINE;
CCB,MBUSY,PCERR,MERR,UFL,TRNC,AUX
  :BOOLEAN;
LC1,LC2,Z,OV,N,C,TB,FB
  :BOOLEAN;
UZ,UOVR,UN,UC,MZ,MOVR,MN,MCB
  :BOOLEAN;
IZ,IOVR,INB,IC,QIO0,QIO31,SIO0,SIO31
  :BOOLEAN;

PROCEDURE READCPU;
  VAR   CPUA:TEXT;
BEGIN
  RESET (CPUA);
  FOR I:=1 TO 24 DO
    READLN (CPUA,SCREEN[I].L);
END;

PROCEDURE DISHEX(DAT:LONGINT;X,Y:INTEGER);
  TYPE  NIB=0..15;
        WIN=PACKED RECORD
          CASE INTEGER OF
            1:(W:LONGINT);
            2:(N:PACKED ARRAY [0..7] OF NIB);
          END;
        STR8=PACKED ARRAY [1..8] OF CHAR;
        STR16=PACKED ARRAY [1..16] OF CHAR;
        ASTR=PACKED RECORD
          CASE INTEGER OF
            1:(S:STR8);
            2:(N:PACKED ARRAY [0..7] OF CHAR);
          END;
        BSTR=PACKED RECORD
          CASE INTEGER OF
            1:(S:STR16);
            2:(N:PACKED ARRAY [0..15] OF CHAR);
          END;
  VAR   DAT1:WIN;
        HEXS1:ASTR;
        ONE:CHAR;
        TWO:BSTR;
BEGIN
  DAT1.W:=DAT;
  TWO.S:='0123456789ABCDEF';
  FOR I:=0 TO 7 DO
    BEGIN
      HEXS1.N[I]:=TWO.N[DAT1.N[I]];
    END;
  SCREEN [X].W[Y]:=HEXS1.S;

```

END;

PROCEDURE DISCPUA;

BEGIN

```

    DISHEX (RDR,4,2);
    DISHEX (MDR,5,2);
    DISHEX (REG[CS.AREG],10,2);
    DISHEX (IQ.W[0],11,2);
    DISHEX (IQ.W[1],12,2);
    DISHEX (BUG,13,2);
    DISHEX (DAB,17,2);
    DISHEX (YB,19,3);
    DISHEX (YB,21,3);
    DISHEX (REG[CS.BREG],10,4);
    DISHEX (BP.W[0],11,4);
    DISHEX (BP.W[1],12,4);
    DISHEX (TIM,13,4);
    DISHEX (PC.W,14,4);
    DISHEX (DBB,17,4);
    DISHEX (QR,22,4);
    DISHEX (AC,3,6);
    DISHEX (BC,4,6);
    DISHEX (MC,5,6);
    DISHEX (CS.AMF,6,6);
    DISHEX (CO.W,10,6);
    DISHEX (CI.W,14,6);
    DISHEX (CC.W,18,6);
    DISHEX (STK[0],3,8);
    DISHEX (STK[1],4,8);
    DISHEX (STK[2],5,8);
    DISHEX (STK[3],6,8);
    (*DISHEX (RE,8,8);*)
    (*DISHEX (ISB,9,8);*)
    DISHEX (SB.H,11,8);
    DISHEX (CS.W,14,8);
    DISHEX (ALUL,17,8);
    DISHEX (MEML,18,8);
    (*DISHEX (CCC,19,8);*)
    (*DISHEX (MAP,20,8);*)
    DISHEX (SEQL,21,8);
    FOR I:=1 TO 24 DO
        WRITELN (SCREEN[I].L);

```

END;

PROCEDURE ROMINIT;

BEGIN

```

    ROM[0] := #A0012001;
    ROM[1] := #A0023502;
    ROM[2] := #A0011012;
    ROM[3] := #A0010013;

```

```

ROM[4] := #A000C100;
ROM[5] := #E00C0000;
ROM[6] := 0;
ROM[7] := 0;
ROM[8] := 0;
ROM[9] := 0;
ROM[11] := 0;
ROM[12] := 0;
ROM[13] := 0;
ROM[14] := 0;
ROM[15] := 0;

```

```
END;
```

```
PROCEDURE POWER;
```

```
  BEGIN
```

```
    PWR[0] := 1;
```

```
    FOR I:=1 TO 31 DO PWR[I] := PWR[I-1]*2;
```

```
  END;
```

```
PROCEDURE PROGCNT;
```

```
  VAR OPC:INTEGER;
```

```
  BEGIN
```

```
    IF CS.IPC=1 THEN
```

```
      BEGIN
```

```
        OPC:=CS.SELPC;
```

```
        PC.W:=PC.W+CS.IPC;
```

```
        IF OPC=3 OR OPC=7 THEN PCE:=1;
```

```
      END;
```

```
  END;
```

```
PROCEDURE ECW;
```

```
  BEGIN
```

```
    IF CS.ECW=1 THEN CS.B[3] := CO.B[3];
```

```
    IF CS.ECW=2 THEN CS.H[1] := CO.H[1];
```

```
    IF CS.ECW=3 THEN CS.FCN:=CO.FCN;
```

```
  END;
```

```
PROCEDURE CON;
```

```
  BEGIN
```

```
    IF CS.SHI=1 THEN CHI:=CS.CON ELSE CHI:=CS.FHI;
```

```
    IF CS.SLO=1 THEN CLO:=CS.CON ELSE CLO:=CS.FLO;
```

```
    IF CS.LHI=1 THEN CI.H[0] := CHI;
```

```
    IF CS.LLO=1 THEN CI.H[1] := CLO;
```

```
  END;
```

```
PROCEDURE SHF;
```

```
  BEGIN
```

```
    TB:=TRUE;
```

```
    FB:=FALSE;
```

```
    IF F.I[0]=1 THEN SIOO:=TB ELSE SIOO:=FB;
```

```

IF F.I[31]=1 THEN SIO31:=TB ELSE SIO31:=FB;
IF F.I[0]=1 THEN QIO0:=TB ELSE QIO0:=FB;
IF F.I[31]=1 THEN QIO31:=TB ELSE QIO31:=FB;
IF CS.SLR=0 THEN
  CASE CS.SSC OF
    0:BEGIN SIO31:=FB; QIO31:=FB; END;
    1:BEGIN SIO31:=TB; QIO31:=TB; MCB:=SIO0; END;
    2:BEGIN SIO31:=FB; QIO31:=MN; END;
    3:BEGIN SIO31:=TB; QIO31:=SIO0; END;
    4:BEGIN SIO31:=MCB; QIO31:=SIO0; END;
    5:BEGIN SIO31:=MN; QIO31:=SIO0; END;
    6:BEGIN SIO31:=FB; QIO31:=SIO0; END;
    7:BEGIN SIO31:=FB; QIO31:=SIO0; END;
    8:BEGIN SIO31:=SIO0; QIO31:=QIO0; END;
    9:BEGIN SIO31:=MCB; QIO31:=QIO0; MCB:=SIO0; END;
    10:BEGIN SIO31:=SIO0; QIO31:=QIO0; END;
    11:BEGIN SIO31:=IC; QIO31:=SIO0; END;
    12:BEGIN SIO31:=MCB; QIO31:=SIO0; MCB:=QIO0; END;
    13:BEGIN SIO31:=QIO0; QIO31:=SIO0; MCB:=QIO0; END;
    14:BEGIN SIO31:=(INB AND (NOT IOVR))OR(NOT(INB)
      QIO31:=SIO0; END; AND IOVR);
    15:BEGIN SIO31:=QIO0; END;
  END

```

```
ELSE
```

```
BEGIN
```

```
  CASE CS.SSC OF
```

```

    0:BEGIN SIO0:=FB; QIO0:=FB; MCB:=SIO31; END;
    1:BEGIN SIO0:=TB; QIO0:=TB; MCB:=SIO31; END;
    2:BEGIN SIO0:=FB; QIO0:=FB; END;
    3:BEGIN SIO0:=TB; QIO0:=TB; END;
    4:BEGIN SIO0:=QIO31; QIO0:=FB; MCB:=SIO31; END;
    5:BEGIN SIO0:=QIO31; QIO0:=TB; MCB:=SIO31; END;
    6:BEGIN SIO0:=QIO31; QIO0:=FB; END;
    7:BEGIN SIO0:=QIO31; QIO0:=TB; END;
    8:BEGIN SIO0:=SIO31; QIO0:=QIO31; MCB:=SIO31; END;
    9:BEGIN SIO0:=MCB; QIO0:=QIO31; MCB:=SIO31; END;
    10:BEGIN SIO0:=SIO31; QIO0:=QIO31; END;
    11:BEGIN SIO0:=MCB; QIO0:=FB; END;
    12:BEGIN SIO0:=QIO31; QIO0:=MCB; MCB:=SIO31; END;
    13:BEGIN SIO0:=QIO31; QIO0:=MCB; MCB:=SIO31; END;
    14:BEGIN SIO0:=QIO31; QIO0:=MCB; END;
    15:BEGIN SIO0:=QIO31; QIO0:=SIO31; END;

```

```
  END;
```

```
END;
```

```
END;
```

```
PROCEDURE CCC; (* NOT IMPLEMENTED YET *)
```

```
BEGIN
```

```
  IF CS.ECT=0 THEN
```

```
  BEGIN
```

```

CASE CS.SCC OF
  0:CCB:=TB;
  1:CCB:=MBUSY;
  2:CCB:=PCERR;
  3:CCB:=MERR;
  4:CCB:=UFL;
  5:CCB:=TRNC;
  6:CCB:=AUX;
  8:CCB:=LC1;
  9:CCB:=LC2;
  END;
END
ELSE
BEGIN
CASE CS.SRSEL OF
  0:BEGIN Z:=UZ;OVR:=UOVR;N:=UN;C:=UC;END;
  1:BEGIN Z:=UZ;OVR:=UOVR;N:=UN;C:=UC;END;
  2:BEGIN Z:=MZ;OVR:=MOVR;N:=MN;C:=MCB;END;
  3:BEGIN Z:=IZ;OVR:=IOVR;N:=INB;C:=IC;END;
  END;
CASE CS.SR OF
  0:CCB:=N<>OVR;
  1:CCB:=NOT(N<>OVR);
  2:CCB:=N<>OVR;
  3:CCB:=NOT(N<>OVR);
  4:CCB:=Z;
  5:CCB:=NOT(Z);
  6:CCB:=OVR;
  7:CCB:=NOT(OVR);
  8:CCB:=C OR Z;
  9:CCB:=Z AND (NOT C);
  10:CCB:=C;
  11:CCB:=NOT C;
  12:CCB:=(NOT C) OR Z;
  13:CCB:=C AND (NOT Z);
  14:CASE CS.SRSEL OF
    0:CCB:=INB <> MN;
    1:CCB:=UN;
    2:CCB:=MN;
    3:CCB:=INB;
    END;
  15:CASE CS.SRSEL OF
    0:CCB:=NOT(INB <>MN);
    1:CCB:=NOT UN;
    2:CCB:=NOT MN;
    3:CCB:=NOT INB;
    END;
  END;
END;
END;
END;

```

```

PROCEDURE CCCU;
BEGIN
  TZ:=UZ; TC:=UC; TN:=UN; TOVR:=UOVR;
  IF CEU=0 THEN
  BEGIN
    CASE SRS OF
      0:BEGIN
        UZ:=MZ; UC:=MCI; UN:=MN; UOVR:=MOVR;
      END;
      1:BEGIN
        UZ:=TB; UC:=TB; UN:=TB; UOVR:=TB;
      END;
      2:BEGIN
        UZ:=MZ; UC:=MCI; UN:=MN; UOVR:=MOVR;
      END;
      3:BEGIN
        UZ:=FB; UC:=FB; UN:=FB; UOVR:=FB;
      END;
      6:BEGIN
        UZ:=IZ; UC:=IC; UN:=INB; UOVR:=IOVR+UOVR;
      END;
      7:BEGIN
        UZ:=IZ; UC:=IC; UN:=INB; UOVR:=IOVR+UOVR;
      END;
      8:UZ:=FB;
      9:UZ:=TB;
      10:UC:=FB;
      11:UC:=TB;
      12:UN:=FB;
      13:UN:=TB;
      14:UOVR:=FB;
      15:UOVR:=TB;
      24:UZ:=IZ;
      25:UZ:=IZ;
      40:UC:=NOT IC;
      41:UC:=NOT IC;
      44:BEGIN
        UN:=IN;
        UOVR:=IOVR;
      END;
      45:BEGIN
        UN:=IN;
        UOVR:=IOVR;
      END;
    OTHERWISE
      UZ:=IZ;
      UC:=IC;
      UN:=INB;
      UOVR:=IOVR;
  END;

```

```

END;
END;
IF CEM=0 THEN
BEGIN
CASE SRS OF
0:BEGIN
MZ:=YZ; MCI:=YC; MN:=YN; MOVR:=YOVR;
END;
1:BEGIN
MZ:=TB; MC:=TB; MN:=TB; MOVR:=TB;
END;
2:BEGIN
MZ:=TZ; MC:=TCI; MN:=TN; MOVR:=TOVR;
END;
3:BEGIN
MZ:=FB; MC:=FB; MN:=FB; MOVR:=FB;
END;
4:BEGIN
MZ:=IZ; TC:=MC; MC:=MOVR; MN:=INB; MOVR:=TC;
END;
6:BEGIN
UZ:=IZ; UC:=IC; UN:=INB; UOVR:=IOVR+UOVR;
END;
7:BEGIN
UZ:=IZ; UC:=IC; UN:=INB; UOVR:=IOVR+UOVR;
END;
8:UZ:=FB;
9:UZ:=TB;
10:UC:=FB;
11:UC:=TB;
12:UN:=FB;
13:UN:=TB;
14:UOVR:=FB;
15:UOVR:=TB;
24:UZ:=IZ;
25:UZ:=IZ;
40:UC:=NOT IC;
41:UC:=NOT IC;
44:BEGIN
UN:=IN;
UOVR:=IOVR;
END;
45:BEGIN
UN:=IN;
UOVR:=IOVR;
END;
OTHERWISE
UZ:=IZ;
UC:=IC;
UN:=INB;

```



```
    UOVR:=IOVR;
  END;
END;
```

```
FUNCTION IQW:LONGINT;
BEGIN
```

```
  CASE CS.SELPC OF
    0: IQW:=IQ.ROT0;
    1: IQW:=IQ.ROT1;
    2: IQW:=IQ.ROT2;
    3: IQW:=IQ.ROT3;
    4: IQW:=IQ.ROT4;
    5: IQW:=IQ.ROT5;
    6: IQW:=IQ.ROT6;
    7: IQW:=IQ.ROT7;
  END;
```

```
END;
```

```
PROCEDURE ALU;
```

```
BEGIN
```

```
  ALUL:=CS.FCN;
  DAB:=REG[CS.AREG];
  IF CS.AIN=0 THEN
    CASE CS.XREG OF
      1: DAB:=TIM;
      2: DAB:=PC.W;
      3: DAB:=CI.W;
      4: DAB:=CC.W;
      5: DAB:=IQW;
      8: DAB:=BP.ROT0;
      9: DAB:=BP.ROT1;
     10: DAB:=BP.ROT2;
     11: DAB:=BP.ROT3;
     12: DAB:=BP.ROT4;
     13: DAB:=BP.ROT5;
     14: DAB:=BP.ROT6;
     15: DAB:=BP.ROT7;
```

```
  OTHERWISE;
```

```
  END;
```

```
  DBB:=REG[CS.BREG];
```

```
  IF CS.LA=0 THEN
```

```
    CASE CS.IALU OF
      1: YB:=0;
      2: YB:=DBB-DAB-1+CN;
      3: YB:=DAB-DBB-1+CN;
      4: YB:=DBB+DAB+CN;
      5: YB:=DBB+CN;
      6: YB:=DAB+CN;
      7: YB:=DAB+CN;
```

```
  END;
```

```

IF CS.LA=1 THEN
  BEGIN
    IAB.W:=DAB;
    IBB.W:=DBB;
    CASE CS.IALU OF
      0:IYB.S:=[ ];
      1:IYB.S:=IAB.S*IBB.S;
      2:IYB.S:=[ ];
      3:IYB.S:=[ ];
      4:IYB.S:=IAB.S*IBB.S;
      5:IYB.S:=[ ]-(IAB.S+IBB.S);
      6:IYB.S:=[ ]-(IAB.S*IBB.S);
      7:IYB.S:=IAB.S+IBB.S;
    END;
  END;
YB:=IYB.W;
IF CS.AIX=0 THEN REG[CS.BREG]:=YB;
IF CS.AIX=1 THEN
CASE CS.XREG OF
  0:BUG:=YB;
  1:TIM:=YB;
  2:PC.W:=YB;
  3:CO.W:=YB;
  4:MDR:=YB;
  5:BEGIN
    IF PC.I[31]=0 THEN
      BEGIN
        IQ.W[0]:=YB;
        IQ.W[2]:=YB;
      END;
    IF PC.I[31]=1 THEN IQ.W[1]:=YB;
    END;
  6:BEGIN
    BP.W[0]:=YB;
    BP.W[2]:=YB;
  END;
  7:BP.W[1]:=YB;
  OTHERWISE;
END;
END;

PROCEDURE MEM;
BEGIN
  MEML:=CS.FCN;
  IF CS.LMC=1 THEN
    BEGIN
      CASE CS.SMC OF
        0:AC:=CS.AMF;
        1:BC:=CS.AMF;
        2:MC:=CS.AMF;

```

```

        OTHERWISE;
        END;
    END;
CASE CS.SMC OF
    0:MA:=AC;
    1:MA:=BC;
    2:MA:=MC;
    3:MA:=CS.AMF;
    END;
IF CS.WMF=1 THEN
    BEGIN
    CASE CS.SMC OF
        0:MM[AC]:=MDR;
        1:MM[BC]:=MDR;
        2:MM[MC]:=MDR;
        3:MM[CS.AMF]:=MDR;
        END;
    END;
    CASE CS.SMC OF
        0:RDR:=MM[AC];
        1:RDR:=MM[BC];
        2:RDR:=MM[MC];
        3:RDR:=MM[CS.AMF];
        END;
    IF CS.MIX=0 THEN REG[CS.RMD]:=RDR;
    IF CS.MIX=1 THEN
        BEGIN
        CASE CS.SMC OF
            0:BUG:=RDR;
            1:TIM:=RDR;
            2:PC.W:=RDR;
            3:CO.W:=RDR;
            4:MDR:=RDR;
            5:BEGIN
                IF PC.I[31]=0 THEN
                    BEGIN
                    IQ.W[0]:=RDR;
                    IQ.W[2]:=RDR;
                    END;
                IF PC.I[31]=1 THEN IQ.W[1]:=RDR;
                END;
            6:BEGIN
                BP.W[0]:=RDR;
                BP.W[2]:=RDR;
                END;
            7:BP.W[1]:=RDR;
            END;
        END;
    END;
END;
END;
END;

```

```

PROCEDURE SEQ;
BEGIN
  SEQL:=CS.FCN;
  IF CS.OCB=1 THEN SADD.B[1]:=IQ.B[CS.SELPC];
  IF CS.BCC=1 AND CCB=0 THEN CS.S10:=0;
  IF CS.FE=1 AND CS.PUP=0 THEN LEV:=-LEV-1;
  IF CS.FE=1 AND CS.PUP=1 THEN
    BEGIN
      LEV:=LEV+1;
      STK[LEV]:=SB.H+1;
    END;
  CASE CS.S10 OF
    0:SB.H:=SB.H+1;
    1:SB.H:=SADD.H;
    2:SB.H:=STK[LEV];
    3:SB.H:=SADD.H;
  END;
END;

BEGIN
  READCPU;
  ROMINIT;
  POWER;
  RESET (KEYBD);
  READ (KEYBD,INKEY);
  WHILE INKEY<>' 'S' ' DO
  BEGIN
    CYCLE:=CYCLE+1;
    DISCPUA;
    CS.W:=ROM[CYCLE];
    PROGCNT;
    ECW;
    CASE CS.LFD OF
      1:CON;
      2:SHF;
      3:CCC;
      4:ALU;
      5:MEM;
      (* 6:MAP; *)
      7:SEQ;
    END;
    (* UPDATE; *)
    READ (KEYBD,INKEY);
  END;
END.

```

APPENDIX H
HDL SIMULATION OF MEM

```

BLOCK MEM DESIGN;
(* GLOBALS CLKA,CLK1,CLK3,CLK4_,VCC,GND *)
  (CLKA,CLK1,CLK3,CLK4_,MR_)@INPUT;

(* INTERFACE TO UCODE *)
(* CS IS THE WCS WORD TO BE LATCHED IN UCODE REG *)
  CS(23 TO 0) @INPUT;

(* LPMEM IS THE LOAD PULSE FOR THE MEM UCODE REG *)
  LPMEM @INPUT;

(* MEMCY MEMORY CYCLE *)
  MEMCY @INPUT;

(* INTERFACE TO MAP BOARD *)
(* PD BUS IS FROM THE PROCESSOR TO THE MAP BOARD *)
  PD(31 TO 0) @INOUT;

(* ENPRD MEANS MAP READ, ENPWT MEANS MAP WRITE *)
  (ENPRD,ENPWT) @INPUT;

(* MAPREQ REQUEST TO THE MAP FOR A MEMORY CYCLE *)
  MAPREQ @INPUT;

(* BSEX BYTE SIGN EXTEND *)
  BSEX @INPUT;

(* HSEX HALF SIGN EXTEND *)
  HSEX @INPUT;

(* INTERFACE TO ALU *)
(* YO IS THE OUTPUT BUS FROM THE 2903 *)
  YO(31 TO 0) @INOUT;

(* LPMDR LOAD PULSE FOR MEMORY DATA REGISTER *)
  LPMDR @INPUT;

(* RMD READ MEMORY DESTINATION INTO THE ALU *)
  RMD(3 TO 0) @OUTPUT;

(* MIX MEMORY DESTINATION INTERNAL OR EXTERNAL *)
  MIX @OUTPUT;

(* MD IS THE MEMORY COUNTER OR DIRECT BUSS *)
  MD(7 TO 0) @OUTPUT;

```

```

(* LOCAL ARRAYS *)
(* AM DIRECT ADDRESS LINES FROM UCODE TO MEMORY *)
  AM(7 TO 0)           @LOCAL;

(* AC A MEMORY COUNTER OUTPUT BUSS *)
  AC(7 TO 0)          @LOCAL;

(* BC B MEMORY COUNTER OUTPUT BUSS *)
  BC(7 TO 0)          @LOCAL;

(* MC M MEMORY COUNTER OUTPUT BUSS *)
  MC(7 TO 0)          @LOCAL;

(* M1 MEMORY COUNTER 1 OUTPUT BUSS *)
  M1(7 TO 0)          @LOCAL;

(* MM MEMORY COUNTERS MUXED BUSS *)
  MM(7 TO 0)          @LOCAL;

(* MA MEMORY ADDRESS BUSS *)
  MA(7 TO 0)          @LOCAL;

(* MBR MEMORY READ BUSS *)
  MBR(31 TO 0)        @LOCAL;

(* MBW MEMORY WRITE BUSS *)
  MBW(31 TO 0)        @LOCAL;

```

STRUCTURE

```

U00:  LAT24
      (* CS 24 BITS INPUT TO UCODE LATCH *)
      CS(23 TO 0),
      (* NCO DONT CARE *)
      NCO,
      (* WMF WRITE MEMORY FILE *)
      WMF,
      (* SMC SELECT MEMORY COUNTER *)
      SMC1, SMC0,
      (* MIX MEMORY DESTINATION INTERNAL OR EXTERNAL *)
      MIX,
      (* LCA LOAD COUNTER A *)
      LCA,
      (* LCB LOAD COUNTER B *)
      LCB,
      (* LCM LOAD COUNTER M *)
      LCM,
      (* ID INCREMENT OR DECREMENT COUNTER *)
      ID,
      (* ECA ENABLE COUNTER A *)

```

```

    ECA,
(* ECB ENABLE COUNTER B *)
    ECB,
(* ECM ENABLE COUNTER M *)
    ECM,
(* RMD READ MEMORY DESTINATION *)
    RMD(3),RMD(2),RMD(1),RMD(0),
(* AM 8 BITS FOR MEMORY ADDRESS *)
    AM(7),AM(6),AM(5),AM(4),AM(3),AM(2),AM(1),AM(0),
    GND,
(* LPMEM LOAD PULSE TO LOAD MEM UCODE REGISTER *)
    LPMEM;

(* MUX MEMORY OR MEMORY1 COUNT AND LOADS *)
U010: LS257A
    EMMC_,ECM_,EMCO_,
    ECM_,EMMC_,EMC1_,
    VCC_,LCM_,LMCO_,
    LCM_,VCC_,LMC1_,
    MM1,GND;

(* FLIP FLOP TO DETERMINE PHASE OF MEMORY AND MEMORY1 *)
U020: LS109A
    VCC,GND,MM1,MM1_,MAPREQ,VCC,MR_,
    NC1,NC2,NC3,NC4,NC5,NC6,NC7;

(* XOR TO CHANGE MM BUSS WITH CLOCK *)
(* GENERATE INCREMENT FROM MAP *)
U030: LS86
    CLKA,MM1,SMM1,
    ENPWT,ENPRD,EMMC,
    CLKA,VCC,CLKA_,
    EMMC,VCC,EMMC_;

(* A MEMORY COUNTER *)
U040: CNT08
    AM(7 TO 0),
    AC(7 TO 0),
    ID,CLK4_,ECA_,LCA_;

(* B MEMORY COUNTER *)
U050: CNT08
    AM(7 TO 0),
    BC(7 TO 0),
    ID,CLK4_,ECB_,LCB_;

(* M MEMORY COUNTER *)
U060: CNT08
    AM(7 TO 0),

```



```

        MC(7 TO 0),
        ID, CLK4_, EMC0_, LMC0_;

(* M1 MEMORY COUNTER *)
U070:  CNT08
        AM(7 TO 0),
        M1(7 TO 0),
        ID, CLK4_, EMC1_, LMC1_;

(* M OR M1 MUX *)
U080:  MUX16_08
        MC(7 TO 0),
        M1(7 TO 0),
        MM(7 TO 0),
        SMM1;

(* MM, AM MUX *)
U090:  MUX32_08
        AM(7 TO 0),
        AC(7 TO 0),
        BC(7 TO 0),
        MM(7 TO 0),
        MD(7 TO 0),
        SMC1, SMC0;

(* MD, MM MUX *)
U100:  MUX16_08
        MM(7 TO 0),
        MD(7 TO 0),
        MA(7 TO 0),
        CLKA;

(* DELAY CLOCK 1 THEN AND WITH MEMORY WRITE FOR WE_ *)
U110:  LS00
        CLK1, CLK1, CLK1_,
        CLK1_, CLK1_, CLK1D,
        CLK3, CLK3, CLK3_,
        CLK3_, CLK3_, CLK3D;

(* AND MEMCY WITH WMF, LMC, EMC *)
U120:  LS00
        LCM, MEMCY, LCM_,
        ECM, MEMCY, ECM_,
        WMF, MEMCY, WMF_,
        WMF_, WMF_, WMFD;

(* AND MEMCY WITH A AND B COUNTER SIGNALS *)
U130:  LS00
        LCA, MEMCY, LCA_,
        ECA, MEMCY, ECA_,

```

```
LCB, MEMCY, LCB_,
ECB, MEMCY, ECB_;
```

```
(* FORM MEMORY WRITE PULSES *)
```

```
U140: LS00
      CLK1D, WMFD, WME_,
      CLK3D, ENPWT, WMME_,
      WME_, WMME_, WE,
      WE, WE, WE_;
```

```
(* MEMORY FILE *)
```

```
U150: MEM256
      MBW(31 TO 0),
      MBR(31 TO 0),
      MA(7 TO 0),
      GND, GND, VCC, WE_;
```

```
(* RDR REGISTER *)
```

```
U160: GAT32
      MBR(31 TO 0),
      YO(31 TO 0),
      VCC, CLKA;
```

```
(* MDR REGISTER *)
```

```
U170: LAT32
      YO(31 TO 0),
      MBW(31 TO 0),
      CLKA_, LPMDR;
```

```
(* MO REGISTER *)
```

```
U180: GAT32
      MBR(31 TO 0),
      PD(31 TO 0),
      ENPRD_, CLKA_;
```

```
(* SIGN EXTEND INVERTER *)
```

```
(* PD BYTE 1 ENABLE=PD1 *)
```

```
(* SE BYTE 1 ENABLE=SE1 *)
```

```
(* SE1=CLKA*BSEX*PD7 *)
```

```
(* SE2=CLKA*(BSEX*PD7+HSEX*PD15) *)
```

```
U190: LS00
      BSEX, PD(7), INT1_,
      INT1_, INT1_, INT1_,
      CLKA_, INT1_, SE1_,
      CLKA_, INT1_, PD1_;
```

```
U195: LS00
      HSEX, PD(15), INT2_,
      INT2_, INT2_, INT2_;
```

```

    INT1_,INT2_,INT3,
    INT1,INT2,INT3_;

U200: LS00
    CLKA_,INT3,SE2_,
    CLKA_,INT3_,PD2_,
    ENPRD,ENPRD,ENPRD_,
    NC8,NC9,NC10;

U210: LS373
    VCC,VCC,VCC,VCC,VCC,VCC,VCC,VCC,
    MBW(31),MBW(30),MBW(29),MBW(28),
    MBW(27),MBW(26),MBW(25),MBW(24),
    SE2_,GND;

U220: LS373
    VCC,VCC,VCC,VCC,VCC,VCC,VCC,VCC,
    MBW(23),MBW(22),MBW(21),MBW(20),
    MBW(19),MBW(18),MBW(17),MBW(16),
    SE2_,GND;

U230: LS373
    VCC,VCC,VCC,VCC,VCC,VCC,VCC,VCC,
    MBW(15),MBW(14),MBW(13),MBW(12),
    MBW(11),MBW(10),MBW(09),MBW(08),
    SE1_,GND;

(* MI REGISTER *)
U240: GAT32S
    PD(31 TO 0),
    MBW(31 TO 0),
    PD2_,PD2_,PD1_,CLKA_,ENPWT;

ENVIRONMENT
    GLOBAL VCC,GND;

END MEM;

BLOCK LAT24;
    DI(23 TO 0) @INPUT;
    (DO23,DO22,DO21,DO20,DO19,DO18,DO17,DO16,
    DO15,DO14,DO13,DO12,DO11,DO10,DO09,DO08,
    DO07,DO06,DO05,DO04,DO03,DO02,DO01,DO00) @OUTPUT;
    (OE_,CLK) @INPUT;

STRUCTURE
U00: LS374
    DI(7),DI(6),DI(5),DI(4),DI(3),DI(2),DI(1),DI(0),

```

```
D007,D006,D005,D004,D003,D002,D001,D000,  
OE_,CLK;
```

```
U01: LS374  
DI(15),DI(14),DI(13),DI(12),  
DI(11),DI(10),DI(9),DI(8),  
D015,D014,D013,D012,D011,D010,D009,D008,  
OE_,CLK;
```

```
U02: LS374  
DI(23),DI(22),DI(21),DI(20),  
DI(19),DI(18),DI(17),DI(16),  
D023,D022,D021,D020,D019,D018,D017,D016,  
OE_,CLK;
```

```
END LAT24;
```

APPENDIX J
CACHE PLACEMENT STUDY

NO CACHE

$$30 * M / 9 = 3.3 * M$$

$$M = 600 \text{ NS}$$

$$MIP = 1 / (3.3 * .6) = .5$$

WORST CASE

$$M = 600 \text{ NS} + 400 \text{ NS}$$

$$MIP = 1 / 3.3 = .3$$

```

+-----+
| DYN0 +-----+
+-----+ |
| DEV | |
+-----+ |
| FETCH+-----+-----
+-----+

```

INST CACHE

NOT OVERLAPPED

$$(18 * M + 12 * PC + 9 * MC) / 9 =$$

$$= 2.0 * M + 1.33 * MC + 1 * PC$$

$$M = 600 \text{ NS} \quad MC = 150 \text{ NS} \quad PC = 150 \text{ NS}$$

$$MIP = 1 / (2.0 * .6 + 1.33 * .15 + .15)$$

$$= .65$$

```

+-----+ +-----+
| DYN0 +-----+ | INST |
+-----+ | | CACHE |
| DEV | | +---+---+
+-----+ | |
| FETCH+-----+-----+-----
+-----+

```

OVERLAPPED

$$18 * M / 9 = 2.0 * M$$

$$M = 600 \text{ NS}$$

$$MIP = 1 / (2.0 * .6) = .83$$

```

+-----+ +-----+
| DYN0 +---+CACHE+---+
+-----+ +-----+ |
| DEV | | |
+-----+ | |
| FETCH+-----+-----+-----
+-----+

```

INSTRUCTION AND DATA CACHE

NOT OVERLAPPED

$$(30 * MC + 9 * PC) / 9 = 3.3 * MC + 1 * PC$$

$$MC = 150 \text{ NS} \quad PC = 150 \text{ NS}$$

$$MIP = 1 / (3.3 * .15 + .15) = 1.5$$

```

+-----+ +-----+
| DYN0 +---+ | INST |
+-----+ | | DATA |
| DEV | | | CACHE |
+-----+ | +---+---+
| FETCH+---+-----+-----
+-----+

```

OVERLAPPED

$$(18 * MC + 9 * PC) / 9 = 2.0 * MC + 1 * PC$$

$$MC = 150 \text{ NS} \quad PC = 150 \text{ NS}$$

$$MIP = 1 / (2.0 * .15 + .15) = 2.22$$

```

+-----+ +-----+
| DYN0 +---+CACHE+---+
+-----+ +-----+ |
| DEV | | |
+-----+ +-----+ |
| FETCH+---+CACHE+---+-----
+-----+ +-----+

```

INSTRUCTION AND DATA CACHE 80%

NOT OVERLAPPED

$$(30*MC*.8+30*M*.2+9*PC)/9=2.66*MC+.66*M+1*PC$$

$$MC = 150 \text{ NS} \quad M = 600 \text{ NS} \quad PC = 150 \text{ NS}$$

$$MIP = 1/(2.66*.15+.66*.6+.15)=1.05$$

OVERLAPPED

$$(18*.8*MC+18*.2*M+9*PC)/9=1.6*MC+.4*M+1*PC$$

$$MC = 150 \text{ NS} \quad M = 750 \text{ NS} \quad PC = 150 \text{ NS}$$

$$MIP = 1/(1.6*.15+.4*.75+.15)=1.45$$

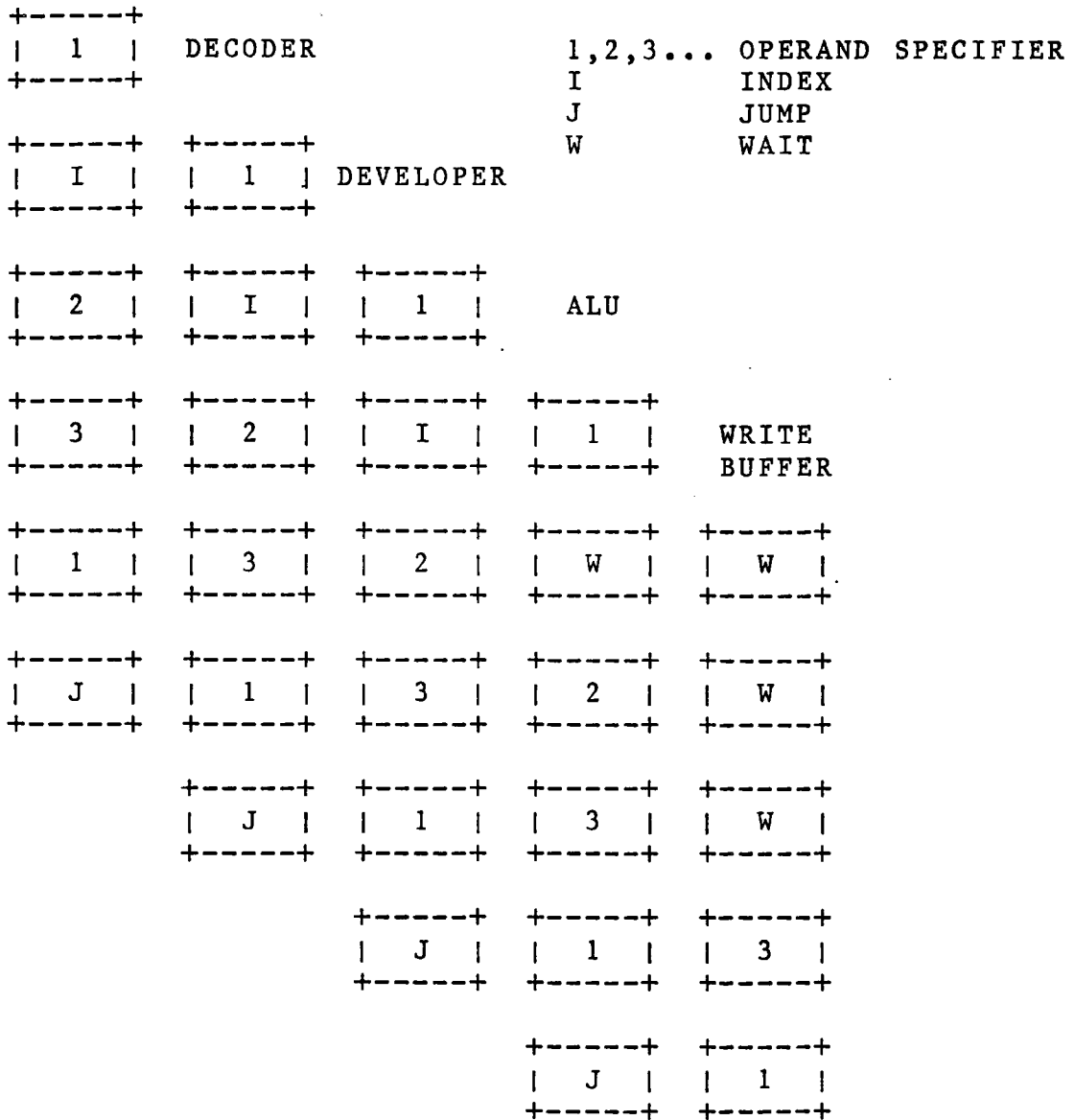
APPENDIX K
PIPELINE DATA FLOW STUDY

PIPELINE EVALUATION FOR THE FOLLOWING ROUTINE;

```

BEGIN   ADD A,B[C],D
        INC C
        JUMP BEGIN
    
```

FIFO



APPENDIX L
DISPLACEMENT DECODE TABLE

OPTIONS

OP CODE ONLY
 WS SHORT
 ADDRESS DIRECT
 DISPLACEMENT
 ADDRESS BASE

INDEX	WS SHORT
INDEX	ADDRESS BASE
LITERAL	#
LS BASE	DISPLACEMENT
WS BASE	DISPLACEMENT
ADDRESS BASE	DISPLACEMENT

INDEX	LS LONG	DISPLACEMENT
INDEX	WS LONG	DISPLACEMENT
INDEX	ADDRESS BASE	DISPLACEMENT
LS LONG	FIELD	DISPLACEMENT
WS LONG	FIELD	DISPLACEMENT
ADDRESS LONG	FIELD	DISPLACEMENT

INDEX	LS LONG	FIELD	DISPLACEMENT
INDEX	WS LONG	FIELD	DISPLACEMENT
INDEX	ADDRESS LONG	FIELD	DISPLACEMENT

DISPLACEMENT

000XXXXX		00000
0010XXXX		00000
0011XXXX	000XXXXX	00102
	0010XXXX	00202
	0011XXXX	10000
	010XXXXX	10000
	0110XXXX	10000
	0111XLLL 0XXXXXXX	01003
	0111XLLL 10XXXXXX	02003
	0111XLLL 110XXXXX	03003
	0111XLLL 1110XXXX	04003
	0111X110 XXXXXXXX 0XXXXXXX	01004
	0111X110 XXXXXXXX 10XXXXXX	02004
	0111X110 XXXXXXXX 110XXXXX	03004
	0111X110 XXXXXXXX 1110XXXX	04004
	1XXX0XXX	00000
	1XXX1LLL 0XXXXXXX	01003
	1XXX1LLL 10XXXXXX	02003
	1XXX1LLL 110XXXXX	03003
	1XXX1LLL 1110XXXX	04003

	1XXX1110	XXXXXXXX	XXXXXXXX	01004
	1XXX1110	XXXXXXXX	10XXXXXX	02004
	1XXX1110	XXXXXXXX	110XXXXX	03004
	1XXX1110	XXXXXXXX	1110XXXX	04004
010XXXXX	000XXXXX			00102
	0010XXXX			00202
	0011XXXX			10000
	010XXXXX			10000
	0110XXXX			10000
	0111XLLL	OXXXXXXXX		01003
	0111XLLL	10XXXXXX		02003
	0111XLLL	110XXXXX		03003
	0111XLLL	1110XXXX		04003
	0111X110	XXXXXXXX	OXXXXXXXX	01004
	0111X110	XXXXXXXX	10XXXXXX	02004
	0111X110	XXXXXXXX	110XXXXX	03004
	0111X110	XXXXXXXX	1110XXXX	04004
	1XXX0XXX			00000
	1XXX1LLL	OXXXXXXXX		01003
	1XXX1LLL	10XXXXXX		02003
	1XXX1LLL	110XXXXX		03003
	1XXX1LLL	1110XXXX		04003
	1XXX1110	XXXXXXXX	OXXXXXXXX	01004
	1XXX1110	XXXXXXXX	10XXXXXX	02004
	1XXX1110	XXXXXXXX	110XXXXX	03004
	1XXX1110	XXXXXXXX	1110XXXX	04004
01100XXX				00000
01101000				01012
01101001				02012
01101010				03013
01101RRR				10000
0111XLLL	OXXXXXXXX			01002
0111XLLL	10XXXXXX			02002
0111XLLL	110XXXXX			03002
0111XLLL	1110XXXX			04002
0111X110	XXXXXXXX	OXXXXXXXX		01003
0111X110	XXXXXXXX	10XXXXXX		02003
0111X110	XXXXXXXX	110XXXXX		03003
0111X110	XXXXXXXX	1110XXXX		04003
11110XXX				00000
11111LLL	OXXXXXXXX			01002
11111LLL	10XXXXXX			02002
11111LLL	110XXXXX			03002
11111LLL	1110XXXX			04002
11111110	XXXXXXXX	OXXXXXXXX		01003
11111110	XXXXXXXX	10XXXXXX		02003
11111110	XXXXXXXX	110XXXXX		03003
11111110	XXXXXXXX	1110XXXX		04003
PREVIOUS	STATE	JUMP/CALL		
OXXXXXXXX				01001

10XXXXXX	XXXXXXXX		02001
110XXXXX	XXXXXXXX	XXXXXXXX	03001
1110XXXX	XXXXXXXX	XXXXXXXX	04001

FIELD

000XXXXX		0
0010XXXX		0
0011XXXX	0111X110	3
0011XXXX	1XXXX110	3
010XXXXX	0111X110	3
010XXXXX	1XXXX110	3
0110XXXX		0
0111XLLL		0
0111X110		2
1XXXXLLL		0
1XXXX110		2

BASE

000XXXXX		WS	1
0010XXXX		WS	1
0011XXXX	000XXXXX	WS	2
0011XXXX	0010XXXX	WS	2
0011XXXX	0110XXXX	0	0
0011XXXX	01110XXX	WS	2
0011XXXX	01111XXX	LS	2
0011XXXX	1AAAXXXX	AAA	2
010XXXXX	000XXXXX	WS	2
010XXXXX	0010XXXX	WS	2
010XXXXX	0110XXXX	0	0
010XXXXX	01110XXX	WS	2
010XXXXX	01111XXX	LS	2
010XXXXX	1AAAXXXX	AAA	2
0110XXXX		0	0
01110XXX		WS	1
01111XXX		LS	1
1AAAXXXX		AAA	1

INDEX/SHORT

000XXXXX	1
0010XXXX	1
0011XXXX	1
010XXXXX	1
011XXXXX	0
1XXXXXXX	0

LENGTH

000XXXXX	H
0010XXXX	W
0011XXXX	W
010XXXXX	H
011XXLLL	LLL
1XXXXLLL	LLL

APPENDIX M
HDL SIMULATION OF FIFO

BLOCK FIFO DESIGN;

```

CLK                @INPUT;
INST_CACHE(63 TO 0) @INPUT;
BYTE_SKIP(2 TO 0)  @INPUT;
FIFO_LOAD          @INPUT;

```

```

FIFO_VALID        @OUTPUT;
FIFO_OUT(63 TO 0) @OUTPUT;

```

BEHAVIOR FUNCTIONAL PROGRAM;

```

BOOLEAN I(4 TO 0), NEXT_FIL(4 TO 0), LAST_SEL(4 TO 0),
        NEXT_SEL(4 TO 0), FIFO(0 TO 31, 7 TO 0),
        J(4 TO 0), FIFO_VALIDI;

```

```

IF VCHANGE(CLK) #*(CLK=1) THEN
  NEXT_SEL:=LAST_SEL@+BYTE_SKIP;
  I:=NEXT_SEL@-NEXT_FIL;
  IF (FIFO_LOAD=1) #*((I>7) #+FIFO_VALIDI=0) THEN
    FIFO(NEXT_FIL):=INST_CACHE(63 TO 56);
    FIFO(NEXT_FIL@+1):=INST_CACHE(55 TO 48);
    FIFO(NEXT_FIL@+2):=INST_CACHE(47 TO 40);
    FIFO(NEXT_FIL@+3):=INST_CACHE(39 TO 32);
    FIFO(NEXT_FIL@+4):=INST_CACHE(31 TO 24);
    FIFO(NEXT_FIL@+5):=INST_CACHE(23 TO 16);
    FIFO(NEXT_FIL@+6):=INST_CACHE(15 TO 8);
    FIFO(NEXT_FIL@+7):=INST_CACHE(7 TO 0);
    NEXT_FIL:=NEXT_FIL@+8;
    I:=NEXT_SEL@-NEXT_FIL;
  END IF;
  J:=NEXT_SEL;
  FIFO_OUT(63 TO 0):=FIFO(J)::FIFO(J@+1)::FIFO(J@+2)::
                    FIFO(J@+3)::FIFO(J@+4)::
                    FIFO(J@+5)::FIFO(J@+6)::FIFO(J@+7);
  IF I>23 THEN FIFO_VALIDI:=0;
  ELSE LAST_SEL:=NEXT_SEL;
        FIFO_VALIDI:=1;
  END IF;
  FIFO_VALID:=FIFO_VALIDI;
  SCHEDULE FIFO_OUT, FIFO_VALID AT 35;
END IF;
END FIFO;

```


VITA

Kerry Cloyce Glover
3802 Hawkshead Dr.
Austin, Texas 78759

Birthplace: Lubbock, Texas

Birthdate: May 21, 1954

Parents: Erwin and Dorthy Glover

Spouse: Joann Adlin Wright Glover

Children: Robert Erwin Glover

Education: B.S. Electrical Engineering
Texas A&M, 1976.

M.S. Electrical Engineering
Texas A7M, 1977.

Experience: November 1980 - Present
Internship Position
Texas Instruments
Austin, Texas

Nov. 1978 - Aug. 1980
Design Engineer
A.R.C.
College Sta., Texas

May 1978 - Sep. 1978
Design Engineer
S.D. Sales
Dallas, Texas

Jan. 1978 - April 1978
Design Engineer
IBM
Austin, Texas

The typist for this report was a TI 990/12 - PDWS.