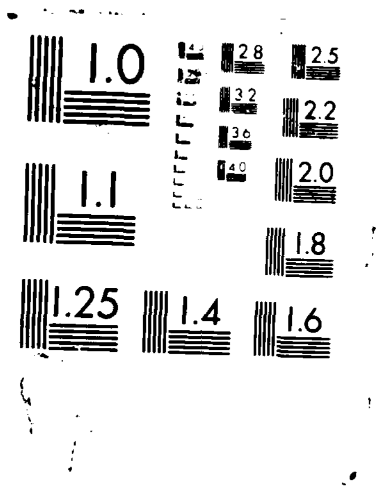
AD-A190 483 DESIGN AND IMPLEMENTATION OF VLSI PRIME FACTOR ALGORITHM PROCESSOR(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 1/2

UNCLASSIFIED R S HAUSER DEC 87 AFIT/GCE/ENG/87D-5 F/G 12/6 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

①

DESIGN AND IMPLEMENTATION OF A VLSI PRIME FACTOR ALGORITHM PROCESSOR

THESIS

Robert S. Hauser
Second Lieutenant, USAF

AFIT/GCE/ENG/87D-5

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCE/ENG/87D-5

DESIGN AND IMPLEMENTATION OF A VLSI PRIME FACTOR ALGORITHM PROCESSOR

THESIS

Robert S. Hauser
Second Lieutenant, USAF

AFIT/GCE/ENG/87D-5

AFIT/GCE/ENG/87D-5

DESIGN AND IMPLEMENTATION OF A VLSI PRIME FACTOR ALGORITHM PROCESSOR

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Robert S. Hauser, B.S.

Second Lieutenant, USAF

December 1987

Approved for public release; distribution unlimited.

Acknowledgements

I would like to thank my thesis advisor, Captain Richard W. Linderman, for his continued support and dedication. Without his help, this thesis would not have been possible. I would also like to thank Major Joe DeGroat and Captain Nathaniel Davis for reading this thesis and providing their input.

Special thanks to Juanita M. Blackford for her endless work in keeping the computer systems up and functioning and preventing me from becoming a hermit. And a late night thank you to everybody that worked in the VLSI trailer.

Finally, thanks to Mom and Dad for continually providing constant support and encouragement, always.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

List of Figures

List of Figures (continued)

List of Tables

AFIT/GCE/ENG/87D-5

Abstract

High-speed digital signal processing has a wide range of applications including, radar, sonar, image processing, and target acquisition. The calculation of the Discrete Fourier Transform (DFT) used in these applications has long been a significant bottleneck for high-speed processing. Previous AFIT students have adopted a Prime Factor Algorithm (PFA) method using Winograd Fourier Transform (WFT) processors. Three WFT processors are pipelined into a system capable of computing a 4080-point DFT on complex data approximately every 120 microseconds when operating with a 70 MHz clock.

This thesis effort addressed the design and implementation of PFA controller chip and interconnecting memory modules between the WFT processors. The PFA controller is an application specific processor to control the flow of information in the pipeline, interface to the WFT processors, monitor pipeline status, and take corrective action in the presence of faults. The interconnecting memory modules buffer the data coming out of a WFT processor and going into another allowing concurrent reading and writing.

The PFA controller chip was designed, simulated, and submitted for fabrication through MOSIS. Twenty-eight 16-bit registers store the pipeline information. An arithmetic/logic unit (ALU) computes data transformations. A read only memory stores the microcode. A control sequencer sequences through the proper code segments. Finally, special circuitry interprets the fault information and reconfigures the pipeline.

This thesis effort included writing a microcode assembler to to raise the user interface to the AFIT-XROM silicon compiler. Raising the user's level of abstraction to mnemonic microcode, while still providing an error free path to silicon layout, reduces chances for error in the microcode specification. A generic microcode assembler tool was created as an extension for use with other application specific processors. This tools generates a microcode assembler from a word format and a translation file. The assembler will output a file compatible with the XROM compiler, a VHDL description of the XROM, a listing file, a reference file, and a reverse assembly. This tool was tested on two other AFIT theses and a computer architecture class.

A prototype memory chip was designed and fabricated in 3 micron CMOS through MOSIS to test the 1-transistor memory cell, the wordline selectors, and the sense amplifiers. Simulations predict an access time of 10ns. A larger memory was designed, simulated, and submitted for fabrication through MOSIS. It contains storage for 272 words of 32 bits each. It is dual ported and permits concurrent reading and writing of 24 bit data. The memory also includes error control circuitry for single error correction and double error detection.

DESIGN AND IMPLEMENTATION OF A VLSI PRIME FACTOR ALGORITHM PROCESSOR

CHAPTER 1

Introduction

1.1. Background

The military has a demonstrated interest in high-speed digital signal processing (DSP). Digital signal processing is used in a wide range of applications including radar, sonar, image processing, voice processing, artificial intelligence, and target acquisition. Applications such as these require data from sample points to be processed as near to real-time as possible with a reasonable amount of resolution. A reasonable amount of resolution is determined by the importance of the application and how accurate the results must be. Since resolution increases with the number of points sampled, it is logical to increase the number of sample points. Unfortunately, as the number of sample points increases, the number of operations increases as the square of the number of sample points (i.e., $O(N^2)$).

Many current DSP applications involve computation of the Discrete Fourier Transform (DFT) which uses only a finite set of sampled signal values instead of the original analog signal. Using a finite set of points allows the signal to be processed and stored on a digital computer. The DFT, in summation form, is as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n)\ W_N^{kn} \qquad k = 0, 1, \ldots, N-1 \tag{1.1}$$

where W sub N ~=~ e sup {-j(2 pi /N)}. Since the number of computations grows as $O(N^2)$ when computed as a sequence of inner products, the DFT is not usually computed directly. Instead, a class of algorithms developed in 1965 by Cooley and Tukey [Coo65] is often used which capitalize on the symmetry in the DFT to reduce to complexity of computation. This class of algorithms, known as Fast Fourier Transforms (FFTs), reduces the number of operations from being proportional to the square of the number of sample points, as in the DFT, to the number of sample point times the log of the number of sample points (i.e., $O(\frac{N}{2}\log_2 N)$). The introduction of the FFT made many DSP applications possible on digital computers.

Prior to the introduction of very large scale integration (VLSI), most DSP computations were performed on-line by general purpose computers, requiring large amounts of time, or off-line by special purpose banks of circuit boards using medium scale, or small scale integration (MSI or LSI respectively). Because the constraints of throughput and blocklengths were so high, real-time computation was not possible. The increased speed and density now available in VLSI, will allow certain DSP functions to be implemented on a single chip.

One way to further increase the speed of the DFT is to reduce the number of multiplications since these are the most consuming. In 1978, Winograd showed a way to reduce the number of multiplications of a DFT while keeping the number of additions approximately equal. He also proved that his class of algorithms, known as Winograd Fourier Transform Algorithms (WFTs), contain the minimum number of multiplication required for computation of a DFT [Win78]. Winograd's large algorithms, however, do not easily map into VLSI because of their size. But, when the small algorithms are combined with the Good-Thomas Prime Factor Algorithm (PFA), they easily map into VLSI due to their small size and regularity [Lin84].

In 1985, a 4080-point transform was chosen as a representative system by the AFIT VLSI Design Group. The system consists of three WFTs of lengths 16, 15, and 17 pipelined together with interconnecting memory modules and a PFA controller. The block diagram of the system is shown in Figure 1. The goal of the WFT-PFA project is to produce a real-time signal processing system. With the PFA operating at 20MHz and each of the WFTs operating at 80MHz, one 4080-point transform will be computed every 120 seconds.

1.2. Problem Statement

The first problem of this thesis effort will be to design the memory modules and implement one in 3-micron CMOS. The second problem will be to design the Prime Factor Algorithm controller chip and implement it in 3-micron CMOS.

1.3. Scope

This thesis will include the design and implementation of the interconnecting memory modules and the PFA controller. First, the memory module will be architecturally specified, layed out in VLSI, and sent for fabrication; second, the PFA controller will be architecturally specified, laid out in VLSI, and sent for fabrication; third, both chips will be tested to ensure proper operation and validation.

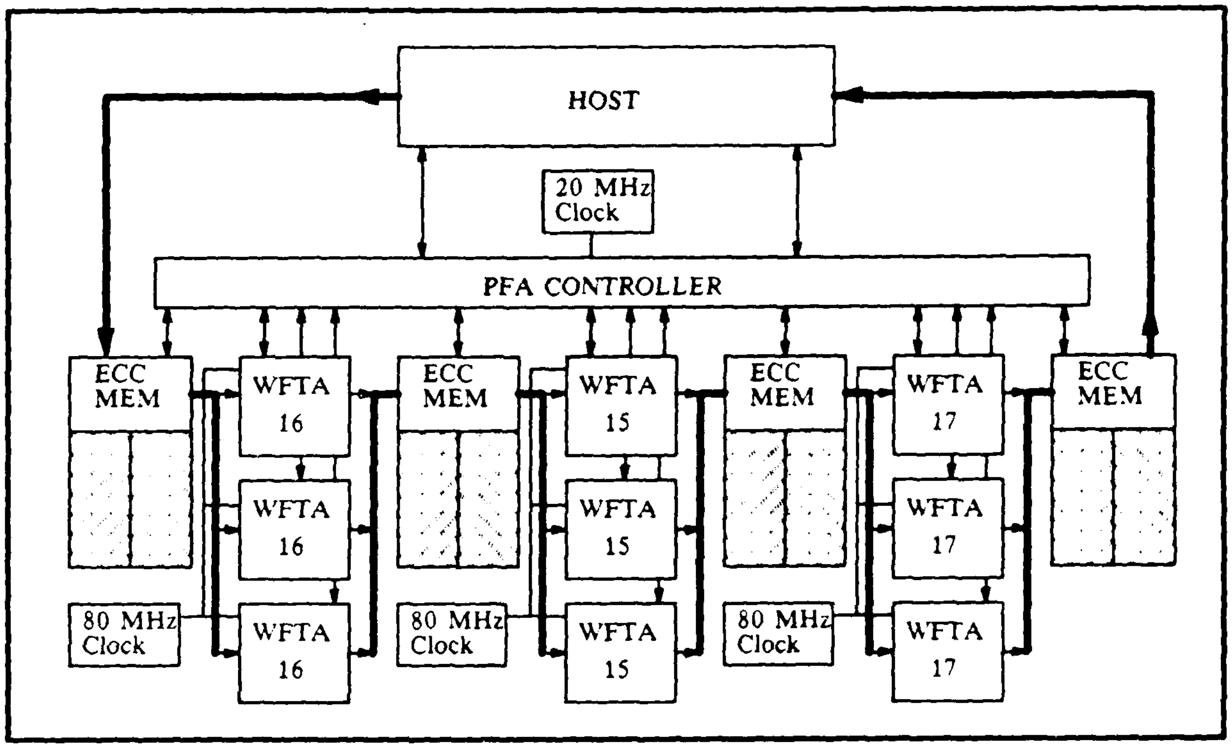


Figure 1 4080-Point Processor

1.4. Approach

This thesis will include the design and implementation of the interconnecting memory modules and the PFA controller. First, the problem requirements and their impact on the design will be analyzed. Second, the architectural descriptions for both chip will be developed. Third, the architectural description will be translated into gate-level descriptions and from gate-level descriptions into VLSI. Fourth, the completed designs will be simulated to verify the design. Fifth, the simulated designs will be sent for fabrication through MOSIS.

1.5. Summary of Current Knowledge

1.5.1. Digital Signal Processing. The current state of fast signal processing algorithms was forged in 1965 by Cooley and Tukey [Coo65]. They demonstrated a method for computing n-point discrete Fourier transforms that required on the order of O(NlogN) computations instead of $O(N^2)$. Their algorithm took advantage of the symmetry and periodicity inherent within DFTs to reduce the number of operations. This method, known as the Fast Fourier Transform or FFT, brought signal processing to digital computer computation.

Another important contribution to the field was Winograd's work published in 1978 [Win78]. Winograd presented algorithms that significantly reduced the number of multiplications required while keeping the number of additions approximately equal. Winograd also showed that his method required a minimal number of multiplications. Because multiplications were much more computationally intensive than additions, his method significantly reduced the computation time.

For a VLSI implementation, however, Winograd's algorithm lacked the modularity needed for effective VLSI design and the number of additions grew quickly as the transform size increased [Lin84]. One solution to this problem was given by Burrus who combined the Good-Thomas Algorithm (PFA) [Goo71] with small Winograd transforms [Bur83]. This combination took one-dimensional transforms and broke them into smaller multi-dimensional transforms. Finally, Linderman [Lin84] presented a way to embed the PFA into VLSI using Winograd Fourier Transform Algorithm (WFT) processors. His solution consisted of decomposing a 4080-point transform into three Winograd processors of lengths 15, 16, and 17.

In 1985, four theses were dedicated to the implementation of Linderman's solution. Taylor presented the PFA and WFT theory, the overall signal processing architecture, and the numerical precision results [Tay85]. He showed that architecture would indeed compute a DFT and remain within acceptable numerical accuracy. Coutee described the arithmetic circuitry for the WFT 16-point transform processor [Cou85]. Rossbach presented the control circuitry for the WFT 16-point [Ros85]. He was able to demonstrate that the control sequencer operated correctly at speeds in excess of 60 MHz in 3-CMOS. Finally, Collins presented a description and validation of the WFT 16-point in the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, VHDL [Col85].

In 1986, two theses were involved in the PFA-WFT project [She86]. Shephard completed the VLSI design and implementation of the WFT16 chip and Hedrick discussed the memory modules and the PFA controller. Hedrick established the foundations of the PFA-ASP by describing the major functional blocks and their interfaces.

1.5.2. Memory Techniques. Weste and Eshragian classified memories into three types, random access, serial access, and content addressable [Wes85]. Random access memory, at the chip level, was described as having an access time independent of location. This contrasts with serial access and content addressable access where the time needed to read or write a value was variable depending on location. Random access memories (RAMs) may be further classified into read only (ROM) or read and write (usually referred to as RAM). ROMs are generally denser than RAMs but they are not as flexible due to the permanency of the data.

RAMs and ROMs can be further be divided according to whether they are static or dynamic. In a static memory, the value of the data is stored with some type of latch, whereas in a dynamic memory, the value is stored with some type of charge on a capacitor. Because the dynamic memory is stored on a capacitor, its charge will degrade and must be refreshed within a certain time interval. The advantage of dynamic memories is that they are much smaller than their static counterparts and use less power [Muk86]. Dynamic memories are smaller because there is no feedback circuitry within the memory cell to keep the location refreshed. The disadvantages are that they are hard to design, somewhat slower than static memories, and more suspectable to soft errors (errors caused by transient radiation).

Several authors described a dynamic memory consisting of a single transistor [Gla85, Muk86, Wes85]. In this type of memory cell, the charge was stored on a capacitor and a single transistor acted as a gate between the bit line and the capacitor. The value was written and read via the bit line. Various techniques for making the capacitor have been implemented including a double polysilicon method and a trench method [Rid79]. These methods tried to increase the amount of capacitance in the memory cell by exploiting non-standard techniques.

The most difficult part of a one-transistor memory cell is the sense amplifier. When a cell is read, the resultant value on the bit line is determined by charge sharing between the bit line and the storage capacitor. Because the storage capacitor is so small compared to the bit line, when the cell is read, there is little voltage change in the bit line. To detect this small change, a very sensitive amplifier is needed. A sense amplifier will amplify the change to levels that correspond to digital logic values.

In 1985, Shinn designed a double stage differential amplifier with current mirror active loading [Shi85]. He detected a differential of 0.01 volts when reading the value and its complement. Grebene fully described the transfer characteristics and gain of such an amplifier in 1984 [Gre84]. He found that the gain was directly proportional to the current loading and the width to length ratio of the gate on the nMOS devices to which the voltages to be sensed were connected.

1.5.3. Error Control Coding. In the transmission or storage of data, noise or other factors may cause erroneous results. In 1949, Shannon demonstrated that a proper encoding of the information could reduce the number of errors induced by a channel [Sha49]. Peterson and Weldon pointed out that as early as 1956, systems were being built that demonstrated error correction and detection [Pet72].

A typical storage system with error correction and detection consists of encoding the data before storage or transmission, the storing or transmission media, and decoding to output the data [Hed86]. With this kind of system, two different types of codes are popular, convolution codes and block codes. In a convolution code, the encoded data is based on the current data as well as previous data, thus requiring a storage media associated the encoder. In a block code, the encoded data only depends on the current information [Hed86].

Since digital computers deal with information coded in binary digits, the discussion of block codes can be limited to those with two symbols [Lin83]. Lin and Costello developed algorithms for block codes of this type using syndromes and standard arrays. Their algorithms will accomplish single error correction and double error detection.

1.6. Materials and Equipment

The materials and equipment needed for this thesis are available at AFIT. All the computer aided design (CAD) tools require a UNIX environment. The tools from the AFIT/VLSI CAD system will include CSTAT, a tool that determines whether nodes can be set to logic 1, set to logic 0, affect the outputs, or can be affected by the inputs, STOVE, a circuit extraction tool, NOFEED, a tool which removes feedback paths for simulation, and FIXROM, a tool that alters the XROM for simulation. The other CAD tools are from the University of California at Berkeley. These tools are distributed each year to various academic institutions. The tools necessary for this thesis include MAGIC, a VLSI layout tool, MEXTRA, a circuit extraction tool, ESIM, a switch level simulator, CRYSTAL, a performance analyzer for VLSI circuits, SPICE, a timing analysis tool, and CIFPLOT, a tool to plot VLSI circuits. MAGIC requires either a SUN Workstation or an AED 767 graphics terminal, both of which are currently available.

This thesis will also require the use of the VHDL language. This language is supported on AFIT's Classroom Support Computer (CSC) operating under the VAX/VMS operating system.

A high-speed VLSI chip tester will also be needed. The tester is located in building 125 Area B.

1.7. Sequence of Presentation

Chapter 2 gives a detailed analysis of the requirements needed for the two problems of this thesis. First the memory is discussed and then the PFA controller.

Chapter 3 presents the architectural descriptions for the memory chip and the PFA controller chip. It also discusses the algorithms involved in the error control coding for the memory and the algorithms involved in the operation of the PFA controller.

Chapter 4 discusses the AFIT CAD environment and the VLSI design methodology. This Chapter also discusses development and operation of the Generic Microcode Assembler Tool.

Chapter 5 presents the VLSI description of the circuits used to implement the memory and the PFA controller. The development of the microcode is also presented.

Chapter 6 discusses the results of the thesis effort The features of the memory chip as well as the PFA controller will be discussed. This chapter will also discuss the results of the microcode assembler on this thesis effort and two other applications.

Chapter 7 discusses the conclusions from this work and presents recommendations for future work. The future work will focus on testing the parts of the pipeline and implementing the prototype.

CHAPTER 2

Detailed Analysis of the Problem

2.1. Overview

The scope of this thesis effort is to design the memory modules and the PFA controller chip. The memory modules are used as a buffer between the different stages of the WFT-PFA pipeline. They allow concurrent reading and writing so that the WFT processors may operate asynchronously with respect to each other and keep data flowing through the pipeline as it is needed and generated. The memory also encorporates single error correction and double error detection based on an (n,k) linear systematic block code to correct some simple errors and provide some error monitoring for the PFA controller.

The PFA controller operates and monitors the pipeline. It is responsible for the pipeline data flow, fault monitoring and reconfiguration, if necessary, and communication with the output host. The PFA controller is considered to be an application specific processor containing elements to store and manipulate data, sequence through a set of predetermined states, and communicate with outside activities.

2.2. Memory

There are six major areas in the memory design. These are the interfaces, the data flow, the storage cell, reading and writing, address selection, and error control coding. The memory act as buffers for the data as it travels through the pipeline. Thus, there are predetermined interfaces which the memories must conform to as set by the previously designed WFT processors.

2.2.1. Interfaces. The first constraints imposed on the memory chip are those of the external interfaces. The PFA controller interfaces with the memory via a *LEFT(RIGHT)* signal. This signal determines which side of the memory is written to and which side is read from. Additionally, the memory is required to send two signals to the PFA controller for error monitoring. The first signal is the *Error Control Code Correctable (ECCC)* and the second is the *Error Control Code Uncorrectable (ECCU)*. The *ECCC* signal flags the PFA controller that a single error occurred and was corrected. The *ECCU* signal flags the PFA controller that a double error was detected and therefore the data could be flawed. These signals will be discussed more in Chapter 5. The WFT processors interface with several groups of signals, the address select lines, the input data lines, the output data lines, and *PRECHARGE*. There are 12 address select lines capable of addressing up to 4096 words. Each word is 24 bits long, giving 24 input data lines and 24 output data lines. The WFT processor on the left, or host in the case of the first memory, feeds the 12 write select addresses and the input data to the chip and the WFT processor on the right, or host in the case of the last memory, feeds the 12 read select addresses and receives the output data.

2.2.2. Data Flow. The basic data flow through the chip is as follows; for the input data, the inputs come into the chip, pass through the ECC encoding and are written into the memory; for the output data, the values from the memory are passed through the ECC decoding and then sent off-chip to the WFT processor. To include concurrent reading and writing by two processors, two banks of the memory must exist so that while one bank is written the other bank is read. To accomplish this, the memory chip must be able to route data from the encoding circuitry to either side of the memory and from either side of the memory to the decoding circuitry. Additionally, the

PRECHARGE signal from the processors must also be routed to the side of the memory the respective processor is using.

2.2.3. Memory Cell. The memory cell holds the value of the input data for later retrieval. Ideally, the node should be able to store data for more than a millisecond. Two types of memory cells can be used, static or dynamic. In a static memory cell, loss of data due to leakage is not a problem. Static memories, however, are large compared to dynamic memories. Additionally, dynamic memories with no refresh are much more dense and less complicated that static memories. Data must be read within a certain time interval or it is lost without refreshing. One of the main results of this thesis will be to determine whether the memory cell will indeed hold the value within the time requirements.

2.2.4. Reading and Writing. The WFT processors operate at a clock frequency of 80MHz outputting a new word every other clock cycle. This means that a complete read or write must be accomplished at 40 MHz, or one operation every 25ns, for an input/output bandwidth of $9.6\text{x}10^8$ bits per second or total bandwidth of $1.92\text{x}10^9$ bits per second.

A complete write includes inputting, encoding, address selection, and value storage. A complete read includes address selection, value sensing, decoding, and outputting.

2.2.5. Address Selection. Each read or write must be able to access any one of the 4096 words in a non-linear fashion. The order of accesses is determined by the Prime Factor algorithms used to compute the Winograd Fourier Transform. Address selection is also included in the 25ns access time. Several different approaches to address selection will be discussed in Chapter 5.

2.2.6. Error Control Coding. As described in Chapter 1, the Error Control Coding is used to provide error correction and detection. The ECC included on the memory chip must be able to provide single error correction double error detection for the 24 bits of data. The ECC to implement this consists of four functional blocks. The first block is used to encode the data before it is written into the storage array. The input data and the parity bits are then written in to the memory. The second block is used to decode the output data into the syndrome bits. The third block takes the syndrome bits, computes the error vectors and generates the *ECCC* and *ECCU* signal discussed earlier. The fourth block takes the error vectors and the output data and generates the data output to the WFT processors.

2.3. PFA Controller

As stated earlier, the PFA controller must be able to process instructions and take appropriate action on conditional data to accomplish its three major tasks. The first of these tasks is the ability to sequence through a set of predetermined control states. the second is the ability to store and manipulate data, and the third is the ability to communicate with off-chip activities. The requirements for each of these major tasks will be described below.

2.3.1. Control State Sequencing. To control the total state of the system, the controller must be able to sequence through a set of predetermined states. These states control the functioning of the pipeline, the storing and manipulation of the data, and the pipeline configuration. These states are expressed through the use of microinstructions. The microinstructions are stored in a read-only memory for execution. The control sequencer determines which of the microinstructions will be executed next based on the current microinstruction. The source of the next microinstruction can either be an external address, the next sequential address, a field from the current instruction, or from the top of the stack. The sequencer determines the source from a set of input conditions and certain fields from the current microinstruction.

2.3.2. Storing and Manipulating Data. Subject to the control states, the PFA controller must be able to store and manipulate data. The controller must be able to store data which is passed from one stage of the pipeline to another after each DFT computation. It must also store information regarding pipeline configuration, catastrophic pipeline failure, memory faults, WFT processor faults, error location, and individual problem status.

The information that must pass from one stage of the pipeline to the other is the scale factor. The scale factor is a 3-bit number indicating the number of scale bit on the data words. A 3-bit scale factor is input along with the data to each WFT processor. After the WFT has completed the DFT it passes a new 3-bit scale factor. This scale factor is passed along to the next WFT processor in the pipeline. The total number of scale bits is accumulated in the PFA controller for each problem as it travels through the pipeline. Thus, the PFA controller must be able to store a 5 bit number for each problem at worst case $(111 + 111 + 111 = 10101)$.

In addition to storing the data, the controller must also have the ability to manipulate it as well. Information from one source may need to be transferred to other sources or used in determining future controller states. This includes loading values from the pads, driving values to the pads, shifting a storage location, comparing two storage locations, and adding storage locations. For example, it will be necessary to increment the number of faults stored for a particular processor each time it fails.

2.3.3. Communication. The controller must be able to interface with the pipeline it operates and the host which submits DFTs for computation and receives the results. The following is a list of those signals which the PFA controller must be able to input or output:

Memory Chips:

1. *FLIP* (*from controller*) - Used to determine which side of the memory is read from or written to. One signal for all memory chips.

2. *ECCC* (*to controller*), Error Control Code Corrected - Used to indicate that a single error occurred in a read operation from the memory and it was corrected. Each memory has its own *ECCC*.

3. *ECCU* (*to controller*), Error Control Code Uncorrected - Used to indicate that an error occurred on a read that could not be corrected. Each memory has its own *ECCU*.

WFT processors:

1. *WFTop* (*from controller*), WFT Operate - Used to put the WFTs processors into computation mode. All WFT processors share one *WFTop* signal.

2. *WD* (*from controller*) - Used to put the WFTs in the operational or watchdog mode. Each pipeline stage has its own *WD* signal. (The signals described here in 2, 3, and 4 share the state/scale bus. This will be explained in Chapter 5)

3. *SZ0* and *SZ1* (*from controller*), Size0 and Size1 - Used to send the DFT size to the WFTs. Each pipeline stage has its own *SZ0* and *SZ1*.

4. *SC0, SC1, SC2* (*bi-directional*), Scale Factor 0,1,2 - Used to send the input data scaling factors and receive the output data scaling factors. Each pipeline stage has its own *SC0, SC1, SC2*.

5. *PE* (*to controller*), Parity Error - Used by the WFTs to signal that a parity error exists in the input data. Each pipeline stage has one *PE* signal.

6. *WDerr* (*to controller*), Watchdog Error - Used by the WFTs to signal that a Watchdog Error has occurred. Each WFT processor has a *WDerr* signal.

7. *WFTdone* (*to controller*), WFT Done - Used to signal that the WFT has finished a DFT. Each pipeline stage has one *WFTdone* signal.

8. *LOAD* (*from controller*), WFT Load - Used to signal a WFT processor to receive information on operation, scaling, and DFT size. Each WFT processor has its own *LOAD* signal.

HOST (these signals will be explained in Chapter 5):

1. *PFAop* (*to controller*), PFA Operate
2. *PFAdone* (*from controller*), PFA Done
3. *LOADSTUFF* (*to controller*), Load/Read PFA
4. *Hs4,Hs3,Hs2,Hs1,Hs0* (*to controller*), Storage Select
5. *H15-H0* (*from controller*), Internal Data Output Lines

CHAPTER 3

Architecture and Algorithms

3.1. Overview

This chapter discuss the architectures and algorithms used in the memory and the PFA controller. First, the memory architectures will be presented and then algorithms for the error control coding will be presented. Second, the general architecture for the PFA controller will be discussed then the algorithms used to operate the controller and the WFT processors will be developed, and finally, the resulting architecture will be shown.

3.2. Memory

The memory can be broken into three main sections, data flow, storage, and error control coding. The data flow section deals with controlling the flow of data through the chip. The storage section discusses the actual storage cells; how they are selected, written to, or read from. Finally, the error control coding explains how the single error correction/double error detection is accomplished.

3.2.1. Data Flow. As previously mentioned, the memory must be partitioned into two halves to allow concurrent reading and writing. This partitioning is shown in Fig. 2. The data initially enters into the memory chip from the input pads, from there the data is passed through the input ECC circuitry and then to either memory side. A control signal determines to which side of the memory the data is written. After a side of the memory is read, also determined by the control signal, the values output by the

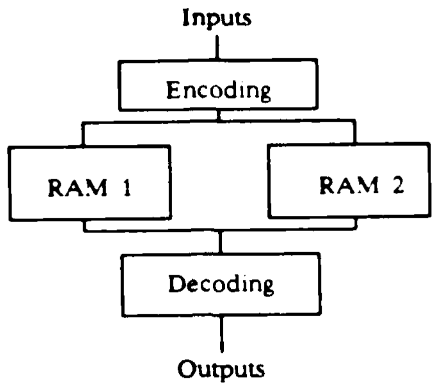


Figure 2 Memory Chip Partioning

memory arrays pass through the output ECC circuitry and then off-chip through the output pads.

3.2.2. Storage. The storage area consists of several units, the word select, the memory cell, and the sense amplifier The word select unit inputs the address lines and determines which of the 4096 words will be selected for the read or write operation A memory word consists of 32 storage cells activated for reading or writing by a single word select line When the value is read, the sense unit will detect the stored memory value and amplify it for further use in the chip

3.2.3. Error Control Coding. Error Control Coding is accomplished in two steps The first step to encode the input data and the second step is to decode the data from the memory before sending off-chip The next two sections outline the development of a systemetic block code described by Lin and Costello Lin83 The matrices for the code were developed by Major Prescot in 1985

3.2.3.1. Encoding For this memory, a *message* is defined as an input word of 24 bits and denoted by **u** where $\mathbf{u}=(u_0, u_2, \ldots, u_{23})$. With 24 bits, the maximum number of distinct messages is 2^{24} or 1.68×10^7. The encoder transforms **u** into an n-tuple, **v**, where $n>24$. The mapping of **u** to **v** is one to one and the set of all vectors **v** is the block code. Additionally, the 2^k code words ($k = 24$) form a k-dimensional subspace of the vector space of all the n-tuples over the Galois field GF(2). Twenty-four linearly independent code words, $\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{23}$, can be found from the block code such that every code word in the block code is a linear combination of the 24 code words. The resulting relationship between **u** and **v** is,

$$\mathbf{v}=u_0\mathbf{g}_0+u_1\mathbf{g}_1+\cdots+u_{23}\mathbf{g}_{23}. \tag{3.1}$$

The 24 linearly independent code words can be arranged into a (24 x n) matrix, **G**, so that,

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{g}_{23} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ g_{23,0} & g_{23,1} & \cdots & g_{23,n-1} \end{bmatrix}. \tag{3.2}$$

G is considered to be the generator matrix for the linear block code. If $\mathbf{u} = (u_0, u_1, \ldots, u_{23})$ is the word to be encoded, then the code word, **v**, is given as:

$$\mathbf{v} = \mathbf{u}\,\mathbf{G}$$

$$= (u_0, u_1, \ldots, u_{23}) \cdot \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{g}_{23} \end{bmatrix} \tag{3.3}$$

$$= u_0\mathbf{g}_0 + u_1\mathbf{g}_1 + \cdots u_{23}\mathbf{g}_{23}.$$

A variation on the linear block code is the linear systematic block code. A linear systematic block code divides the code word into two parts, the message part and the redundant part. The message part consists of the original input bits and the redundant part consists of parity-check digits which are linear sums of the input bits. For a linear systematic (n, 24) code, the 24xn **G** matrix has the following form:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{g}_{23} \end{bmatrix} = \begin{bmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,n-23} & 1 & 0 & \cdots & 0 \\ p_{1,0} & p_{1,1} & \cdots & p_{1,n-23} & 0 & 1 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ p_{23,0} & p_{23,1} & \cdots & p_{23,n-23} & 0 & 0 & \cdots & 1 \end{bmatrix}, \tag{3.4}$$

where p_{ij} is a binary digit. Let **P** denote the first part of the generator matrix and $\mathbf{I}_{24}$ denote the 24x24 identity matrix. Then $\mathbf{G} = \left[\mathbf{PI}_{24}\right]$. The code word now becomes

$$\mathbf{v} = (v_0, v_1, \ldots, v_{n-1}) \tag{3.5}$$
$$= (u_0, u_1, \ldots, u_{23}) \cdot \mathbf{G}.$$

The last 24 bits of the code word correspond exactly to the original 24 in bits the input word and the first n-24 bits are the redundant parity-check bits. The equations used to compute the first n-24 bits are called parity-check equations and are of the form:

$$v_j = u_0 p_{0,j} + u_1 p_{1,j} + \cdots + u_{23} p_{23,j}, \quad (3.6)$$

for $j = 0$ to $n-23$. For our case we use 8 parity check bits making n=32. Appendix A shows the resulting **P** matrix. Using Eq. 3.6, the corresponding parity-check equations from matrix **P** are:

$$v_0 = u_0+u_7+u_8+u_{14}+u_{16}+u_{21} \quad (3.7a)$$

$$v_1 = u_0+u_1+u_9+u_{15}+u_{17}+u_{22} \quad (3.7b)$$

$$v_2 = u_1+u_2+u_8+u_{10}+u_{18}+u_{23} \quad (3.7c)$$

$$v_3 = u_2+u_3+u_9+u_{11}+u_{16}+u_{19} \quad (3.7d)$$

$$v_4 = u_3+u_4+u_{10}+u_{12}+u_{17}+u_{20} \quad (3.7e)$$

$$v_5 = u_4+u_5+u_{11}+u_{13}+u_{18}+u_{21} \quad (3.7f)$$

$$v_6 = u_5+u_6+u_{12}+u_{14}+u_{19}+u_{22} \quad (3.7g)$$

$$v_7 = u_6+u_7+u_{13}+u_{15}+u_{20}+u_{23} \quad (3.7h)$$

The code word **v** can now be expressed as:

$$\mathbf{v} = (v_0, \ldots, v_7, u_0, \ldots u_{23}),$$

where $v_0, \ldots, v_7$ are from Eqs 3.7a-h and $u_0, \ldots, u_{23}$ are the original input word **u**.

3.2.3.2. Decoding Once **v** has been stored in the memory it may be exposed to conditions which cause errors to appear. Because of the extra parity bits, the decoding scheme will correct single errors and detect double errors.

Let $\mathbf{r} = (r_0, r_1, \ldots, r_{31})$ be the word read out of the memory. This word may or may not be different from the word initially stored. Now let **e** be the vector sum of the code word, **v**, and **r** such that,

$$\mathbf{e} = \mathbf{v}+\mathbf{r} = (e_0, e_1, \ldots, e_{31}), \quad (3.8)$$

where e_i is the boolean exclusive-or of v_i and r_i. Alternately, **r** may be represented as the vector sum of **e** and **v**.

It is necessary to introduce another matrix associated with the block code. This $(n-k)\mathrm{x}n$ matrix $\mathbf{H}$, commonly referred to as the parity-check matrix, is defined so that any vector in the row space of $\mathbf{G}$ is orthogonal to the rows of $\mathbf{H}$ and any vector that is orthogonal to the rows of $\mathbf{H}$ is in the row space of $\mathbf{G}$. Thus, a codeword generated by $\mathbf{G}$ solves the equation $\mathbf{v}\cdot\mathbf{H}^T = 0$. The parity-check equation may be written as follows:

$$\mathbf{H} = \left[\mathbf{I}_{n-k}\ \mathbf{P}^T\right] \tag{3.9}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & p_{0,0} & p_{0,0} & p_{1,0} & \cdots & p_{23,0} \\ 0 & 1 & 0 & \cdots & 0 & p_{0,1} & p_{0,1} & p_{1,1} & \cdots & p_{23,1} \\ 0 & 0 & 1 & \cdots & 0 & p_{0,2} & p_{0,2} & p_{1,2} & \cdots & p_{23,2} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdots & 1 & p_{0,7} & p_{0,7} & p_{1,7} & \cdots & p_{23,7} \end{bmatrix},$$

where $\mathbf{P}^T$ represents the transpose of the matrix $\mathbf{P}$.

When $\mathbf{r}$ is read, the decoder computes the following:

$$\mathbf{s} = \mathbf{r}\cdot\mathbf{H}^T = (s_0, s_1, \ldots, s_7). \tag{3.10}$$

This equation defines the syndrome, $\mathbf{s}$, of $\mathbf{r}$. Because $\mathbf{v}\cdot\mathbf{H}^T = 0$, $\mathbf{r}$ is a valid codeword if and only if $\mathbf{s} = 0$. If $\mathbf{s} \neq 0$ then the word read has been corrupted by errors. If the error vector $\mathbf{e}$ is identical to a codeword then $\mathbf{e}+\mathbf{v}$ represents a valid code word and $\mathbf{r}\cdot\mathbf{H}^T = 0$. This condition is considered an undetectable error. Since there exist 2^{24}-1 nonzero codewords, there exist the same number of undetectable error patterns. The $\mathbf{H}^T$ matrix for our (32,24) block code is shown in appendix A.

The corresponding syndrome digit equations are:

$$s_0 = r_0+r_8+r_{15}+r_{16}+r_{22}+r_{24}+r_{29} \tag{3.11a}$$

$$s_1 = r_1+r_8+r_9+r_{17}+r_{23}+r_{25}+r_{30} \tag{3.11b}$$

$$s_2 = r_2+r_9+r_{10}+r_{16}+r_{18}+r_{26}+r_{31} \tag{3.11c}$$

$$s_3 = r_3+r_{10}+r_{11}+r_{17}+r_{19}+r_{24}+r_{27} \tag{3.11d}$$

$$s_4 = r_4+r_{11}+r_{12}+r_{18}+r_{20}+r_{25}+r_{28} \tag{3.11e}$$

$$s_5 = r_5+r_{12}+r_{13}+r_{19}+r_{21}+r_{26}+r_{29} \tag{3.11f}$$

$$s_6 = r_6+r_{13}+r_{14}+r_{20}+r_{22}+r_{27}+r_{30} \tag{3.11g}$$

$$s_7 = r_7+r_{14}+r_{15}+r_{21}+r_{23}+r_{28}+r_{31} \tag{3.11h}$$

Thus, we have computed the syndrome of $\mathbf{r}$ which is equal to $\mathbf{r}\cdot\mathbf{H}^T$. But, $\mathbf{r} = (\mathbf{v}+\mathbf{e})$ so that $\mathbf{s} = (\mathbf{v}+\mathbf{e})\mathbf{H}^T = \mathbf{v}\cdot\mathbf{H}^T+\mathbf{e}\cdot\mathbf{H}^T$. However, $\mathbf{v}\cdot\mathbf{H}^T = 0$ so that the following relationship is established: $\mathbf{s} = \mathbf{e}\cdot\mathbf{H}^T$. Because of the above relation it is possible to compute $\mathbf{e}$ from $\mathbf{s}$. The reader is referred to Chapter 3 of Lin and Costello [Lin83] for the proof. The error vector, $\mathbf{e}$, is computed from the matrix in Appendix A. Appendix A also shows the 32 e_j equations. Each e_j is a combination of the 8 syndrome digits so that

$$e_j = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7 \ ,$$

where $s_m \cdot s_n$ represents the boolean AND function and s_i may be boolean 1 or 0. For example, the syndrome decoding of $e_8 = s_0 \cdot s_1 \cdot \overline{s}_2 \cdot \overline{s}_3 \cdot \overline{s}_4 \cdot \overline{s}_5 \cdot \overline{s}_6 \cdot \overline{s}_7$, where s_i represents boolean 1 and $\overline{s}_i$ represents boolean 0. From Eq. 3.8 it follows that $\mathbf{v} = \mathbf{e}+\mathbf{r}$. Therefore, the approximation of the original codeword is obtained by EXCLUSIVE-ORing the word read out of memory, $\mathbf{r}$, and the computed error vector, $\mathbf{e}$.

3.3. PFA Controller

The PFA controller chip consists two main architectural parts, the microprogrammed control unit and the processor unit. The microprogrammed control unit is responsible for sequencing through predetermined states for the controller and initiating processor actions as well as data flow through the chip. The processor unit performs all the data manipulations as well as storing the data and sending signals to the microprogrammed control unit. In order to make the architectural description more meaningful, it is necessary to discuss the algorithms associated with the PFA controller. In an application specific processor, the algorithm drives the architecture.

3.3.1. Host Algorithm. The host must operate and communicate with the PFA controller. Handshaking is kept to a minimum, optimally only an operate signal to the PFA controller and a done signal from the controller. The host algorithm is shown in Fig. 3. Initially, the host must reset the controller. This initializes the storage areas and resets all counters and addresses lines. Now, the host lowers the operate signal and then lowers the reset signal. The host can proceed to inspect any storage locations and update if needed. Specifically, the host must specify the initial configuration, scale factors, size, and timeout information. Once the proper data has been set, the operate signal to the PFA controller is raised. Since the pipeline is set with data entering from one side and exiting from the other, there may be two different hosts, an input host and an output host. The input host must load the new data into the first memory and send a done signal to the PFA controller when all the data is loaded. The output host, at this time, must unload the DFT results from the last memory in the pipeline. From the register inspection, the output host knows whether the pipeline output data is valid or not. If the data is not valid, the host ignores the current output data. The next event is for the current DFT to be computed, this is signaled by the done signal from the PFA controller. The host may then continue to operate the PFA as long as needed.

At any time during the pipeline operation while the PFA is not computing, the host may change the configuration, change the size of the DFT to be computed, implying a hardware reconfiguration, or change the scale factors of the input data.

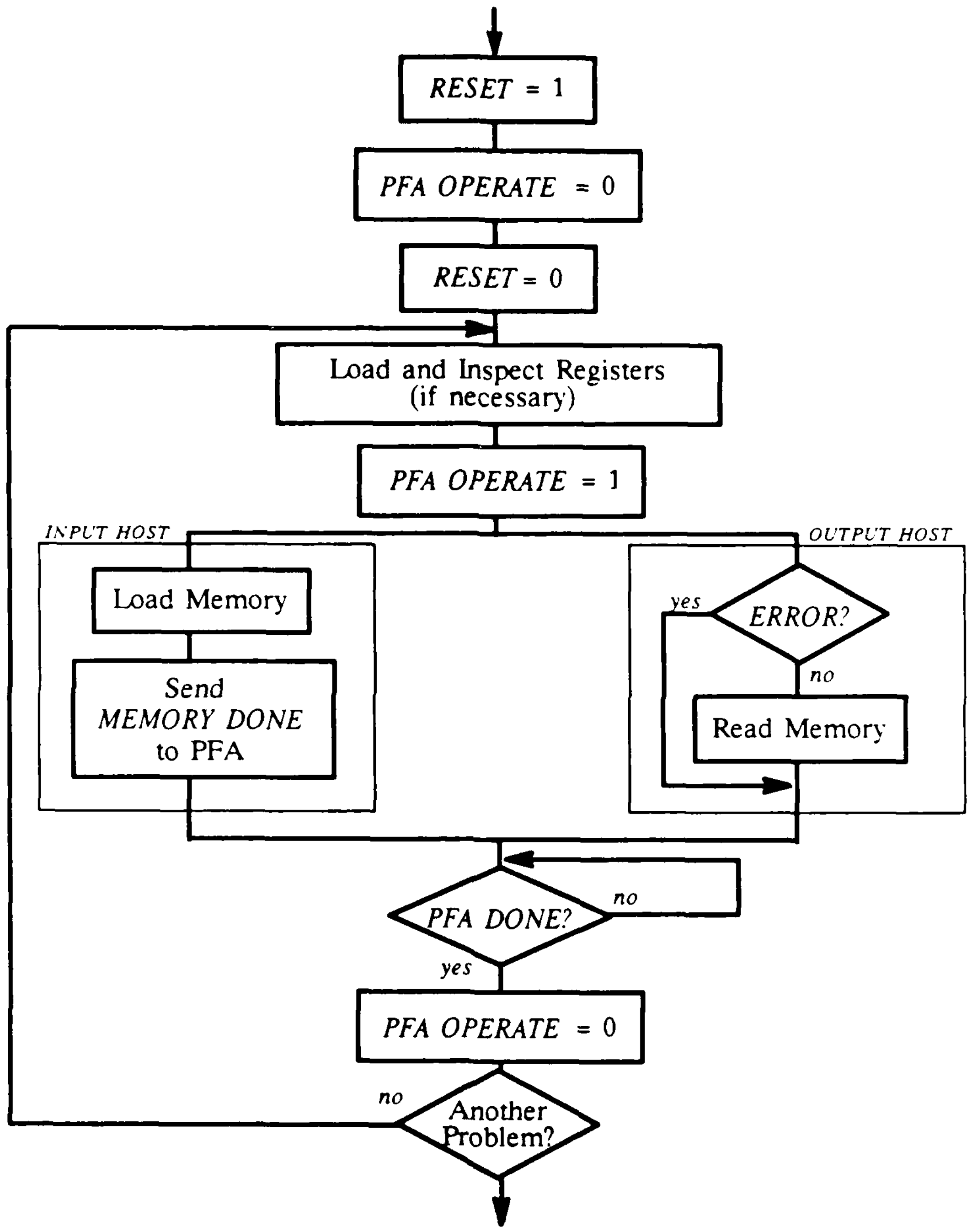


Figure 3 Host Microcode Flow Chart

3.3.2. PFA Controller Algorithm. The high level algorithm for the PFA controller is shown in Fig. 4. The first section deals with initializing values and data on the chip during power up. The controller then waits for an operate signal from the host. While the PFA controller is waiting for the operate signal, the host has access to the internal data stored on the chip. Once the controller has received the operate signal, it

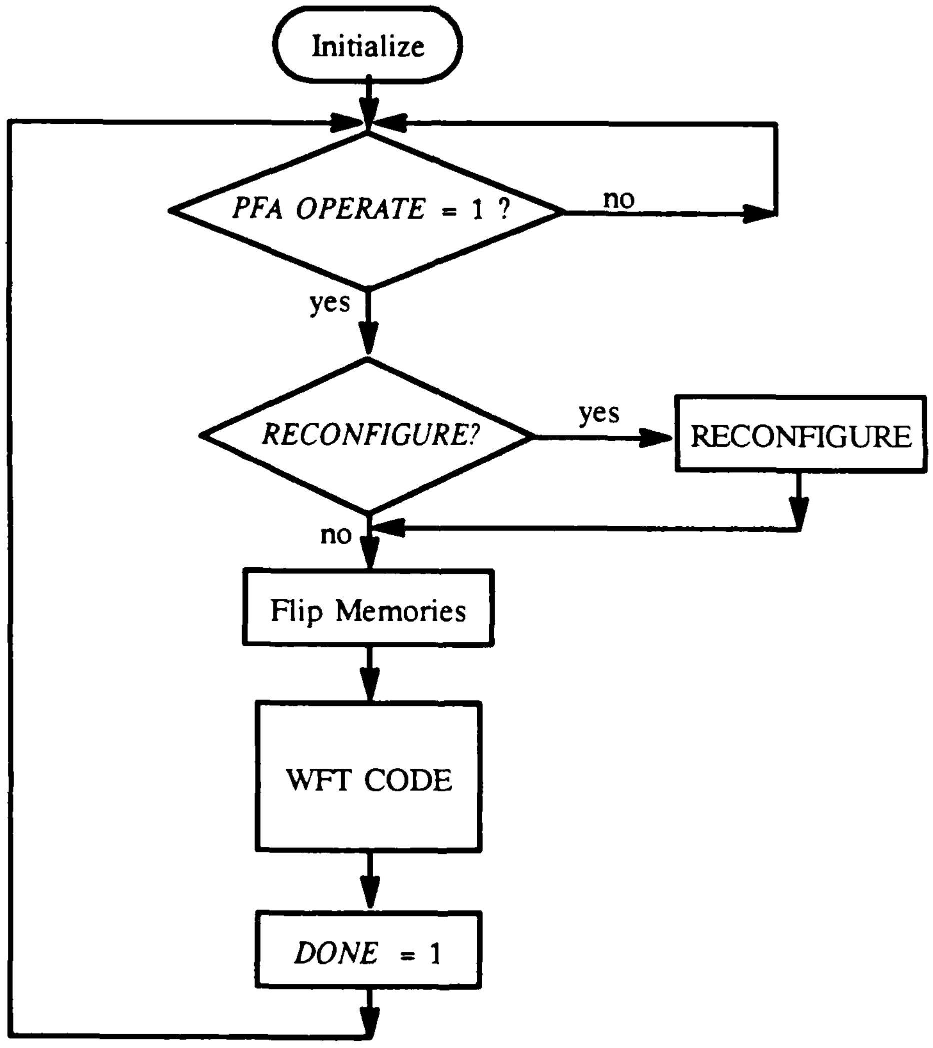


Figure 4 Controller Microcode Flow Chart

checks to see if the pipeline must be reconfigured. Reconfiguration is discussed later in this chapter. The PFA controller then toggles the flip signal sent to the memories. The controller then communicates with the WFT processors. Once the WFT computations are done, the PFA sends a done signal to the host.

The WFT interface algorithm is shown in Fig. 5. The PFA controller must first send the scale factors to the WFT, then send the operate signal, and then float the scale factor lines so that the output scale factors may be returned by the WFT processors. The controller then waits for four done signals: one from the input host and one from each stage of WFT processors. Once all have finished, the PFA controller latches the new scale factors from each stage and stores them internally. The controller then lowers the WFT operate signal. The controller looks at information received from the pipeline to determine if any errors have occurred and updates certain storage locations if need be. If an error has occurred in the active WFT processor, the controller sets up a new configuration.

3.3.3. Microprogrammed Control Unit. The microprogrammed control unit (MCU) controls the operation of the chip. It generates the control signals that operate the pipeline, communicate with the host, or manipulate the internal data. This unit can be thought of as two separate sections: the control memory and the microprogrammed sequencer [Man82]. The control memory is a read only memory (ROM) where words, read one at a time, represent a microinstruction. Each microinstruction contains information needed for the chip to operate. The instruction format is broken down into logical fields. Each of these fields defines a certain set of operations in the chip. These fields can be horizontally or vertically coded [Man82]. To achieve a compromise of the benefits and tradeoffs of the two approaches, the PFA microinstruction contains both.

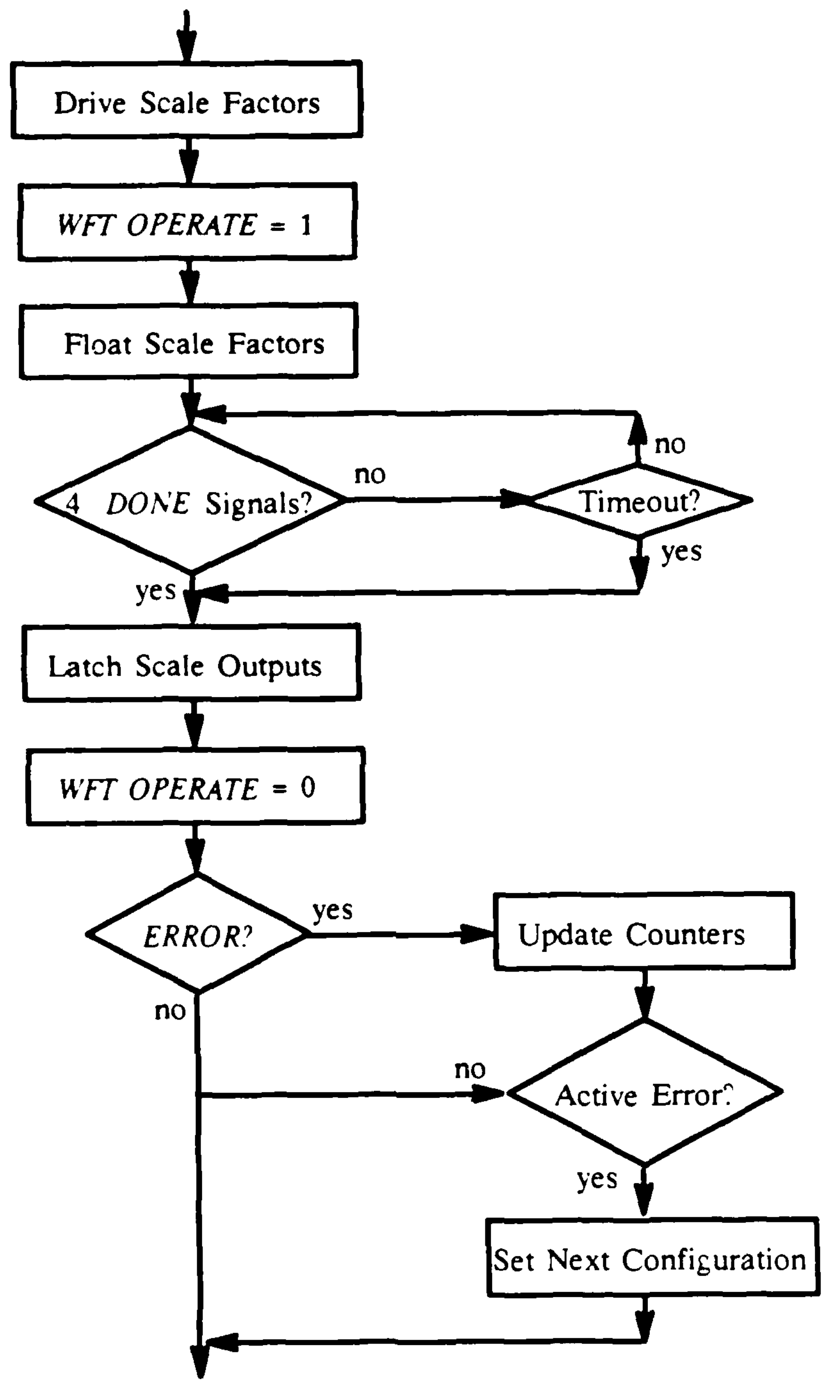


Figure 5 WFT Interface Microcode Flow Chart

The microprogrammed sequencer determines the next address to be read out of the ROM. The next address can come from several sources including the microinstruction itself, an external bus, a stack used for subroutines, or simply be the next address in sequence. The microprogrammed sequencer determines the address from a number of conditions selected by the microcode instruction.

Figure 6 shows a typical MCU. The control memory contains several main fields. Two of these fields determine how the next address is selected, the condition and the branch fields. The condition field selects one of the conditions used for branching and control that exist in the chip. The branch field selects either *T* or *F* for an unconditional branch or call, or positive or negative sense of the selected condition. The branch

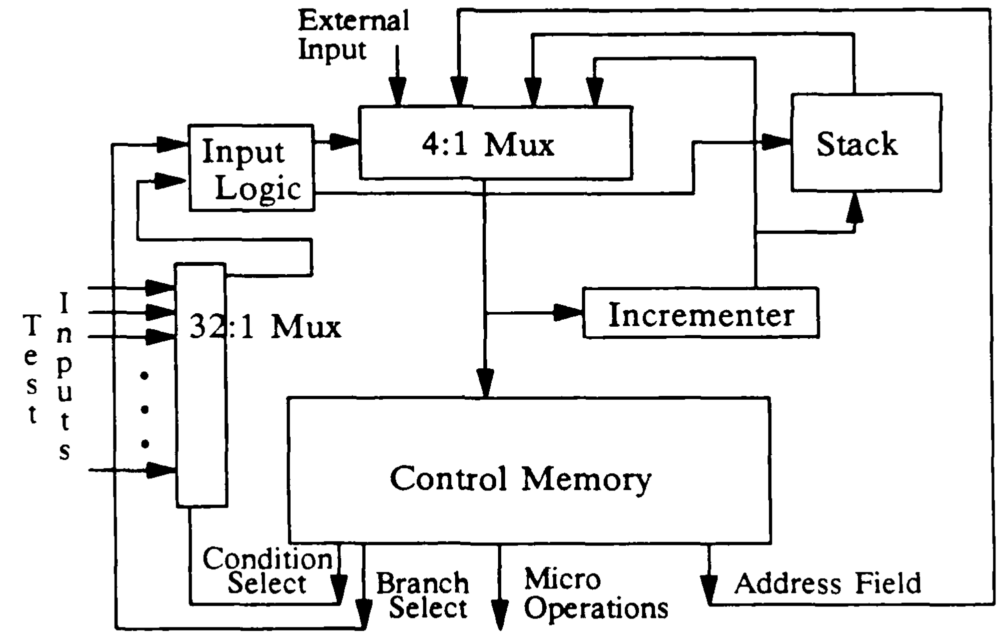


Figure 6 Microprogrammed Control Unit

address may come from three different places, the instruction itself, the stack, or an external source. The stack is used as a temporary storage location in the event of subroutine call for the return address. When a call is executed, the address of the next instruction is pushed onto the top of the stack. On a return, the branch address is popped off the stack and selected as the next address. The MCU also contains an incrementer for sequential addressing.

3.3.4. Processor Unit. The processor unit contains the internal storage for the chip and the circuitry necessary to manipulate the data. The PFA controller architecture is organized around a bus structure and is shown in Fig. 7. The data is stored in a set of registers that can be loaded or read. The registers are loaded through the C bus or from an XROM field. The particular register is selected via a decoder. The data may be read out on either the A bus or the B bus. The A bus and the B bus are fed into the arithmetic logic unit (ALU). The ALU can perform 15 functions on the data and set condition bits accordingly. For a detailed description of the ALU, the reader is referred to the thesis of Capt. David Gallagher [Gal87].

3.3.5. Registers. The registers store the data for the controller. There are 28 registers in the controller. Seven of the registers have special purposes requiring such options as being loaded from sources other than the load bus, driving the contents to destinations other than the data busses, and shifting. The following defines the registers used in the PFA controller and any special requirements:

ECC1-ECC3, Error Control Corrected registers.

These three registers are used as counters for the number of Error Control Code

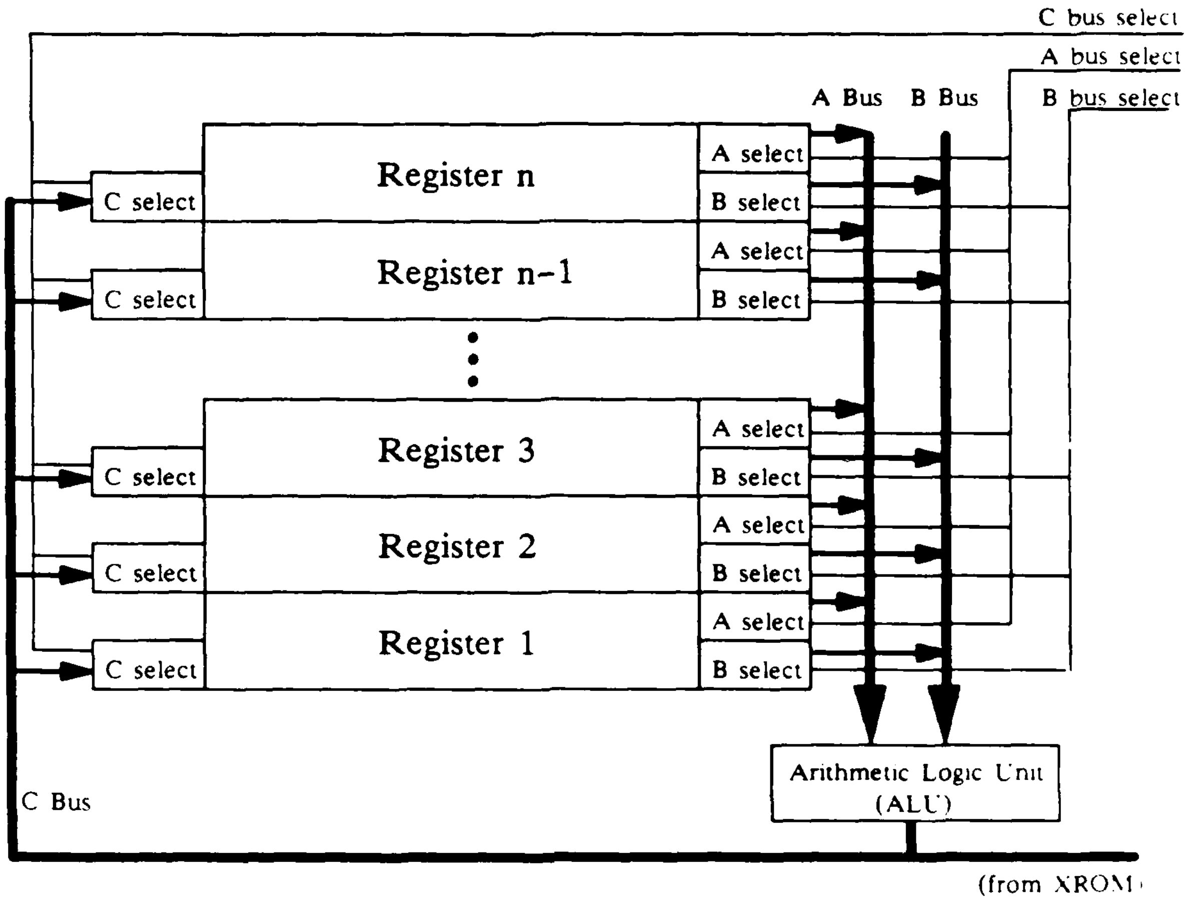


Figure 7 Processor Unit

corrected from the first three memories. These registers can be used for fault monitoring.

ECU1-ECU3, Error Control Uncorrected registers.

These three registers are used as counters for the Error Control Code *uncorrected* from the first three memories. These registers can also be used for fault monitoring.

PE1-PE3, Parity Error registers.

These three registers are used as counters for the number of Parity Errors signaled by each stage of the pipeline.

WD11-WD13,WD21-WD23,WD31-WD33, Watch Dog registers.

These nine registers are used as counters for the number of times the WFT processor was assigned as having a fault.

TSR, Temporary Scale Register.

This register is used to store the scale factors to be given to each stage of the pipeline. Nine of the bits (3 sets of 3) must be able to be sent to the WFT processors. The register must also be able to receive the scale factors from the WFT processors and then shift those by 5 bit for the next stage of the pipeline.

PSR, Permanent Scale Register.

This register is used to store the accumulating scale factors for each of the three problems in the pipeline. This register must be able to transfer the problem's scale factors to the respective problem's Problem Status register. This register must also be able to shift the scale factor bit by 5 positions.

PS1-PS3, Problem Status register.

These three registers store information about each of the three problems in the pipeline. They store the current accumulated scale factor, and whether there was a fault due to an active error, a memory uncorrectable error, or a parity error and which stage of the pipeline the error occurred. The register also contains a bit signifying whether the data was validated by the processor finishing.

CCR, Current Configuration Register and NCR, Next Configuration Register.

These two registers store the current and next configurations respectively.

TOUT, Timeout Register.

This register contains a time out value to determine whether a pipeline catastrophic pipeline failure has occurred, such as a processor not finishing.

TEMP, Temporary register.

This register is used as a scratch pad for many of the data manipulations.

3.3.6. Microcode Word Format. Now that all of the major architectural components have been identified, the fields in the word format can be identified. Figure 8 shows the architecture of the PFA controller. Each field in the microcode will control the operation of an architectural block, peripheral circuit, or direct output. The first field controls the addresses sequencer for the XROM. It determines where the next address originates. The second field specifies the operation of the ALU. The third field is used to indicate an insertion of the literal field onto the destination register bus. This is used to load a constant from the microword into a register. The fourth, fifth, and sixth fields select the two source registers and the destination register, respectively, for the ALU. The seventh field is horizontally coded and specifies certain control signals for the chip. The eighth field signals the WFT processors to operate. The ninth field signals the host that the DFT is done. Finally, the tenth field contains the address of the branch or a constant to be inserted onto the destination bus. The bit fields will be described in Chapter 4.

3.3.7. Pipeline Fault Tolerance. The pipeline is set up for fault tolerance using triple redundancy and voting for the WFT processors [Hed86]. In each stage, one of the three processors is considered to be active, the other two are considered to be in watchdog mode. When a processor operates in watchdog mode, it receives the same data as the active processor, computes the same transform and compares its results with the results of the active at the output pads without driving the pipeline data bus. If there is a discrepancy, a WatchDog error bit is signaled to the PFA controller. The PFA controller looks at the three WatchDog error bits from each of the processors in each stage

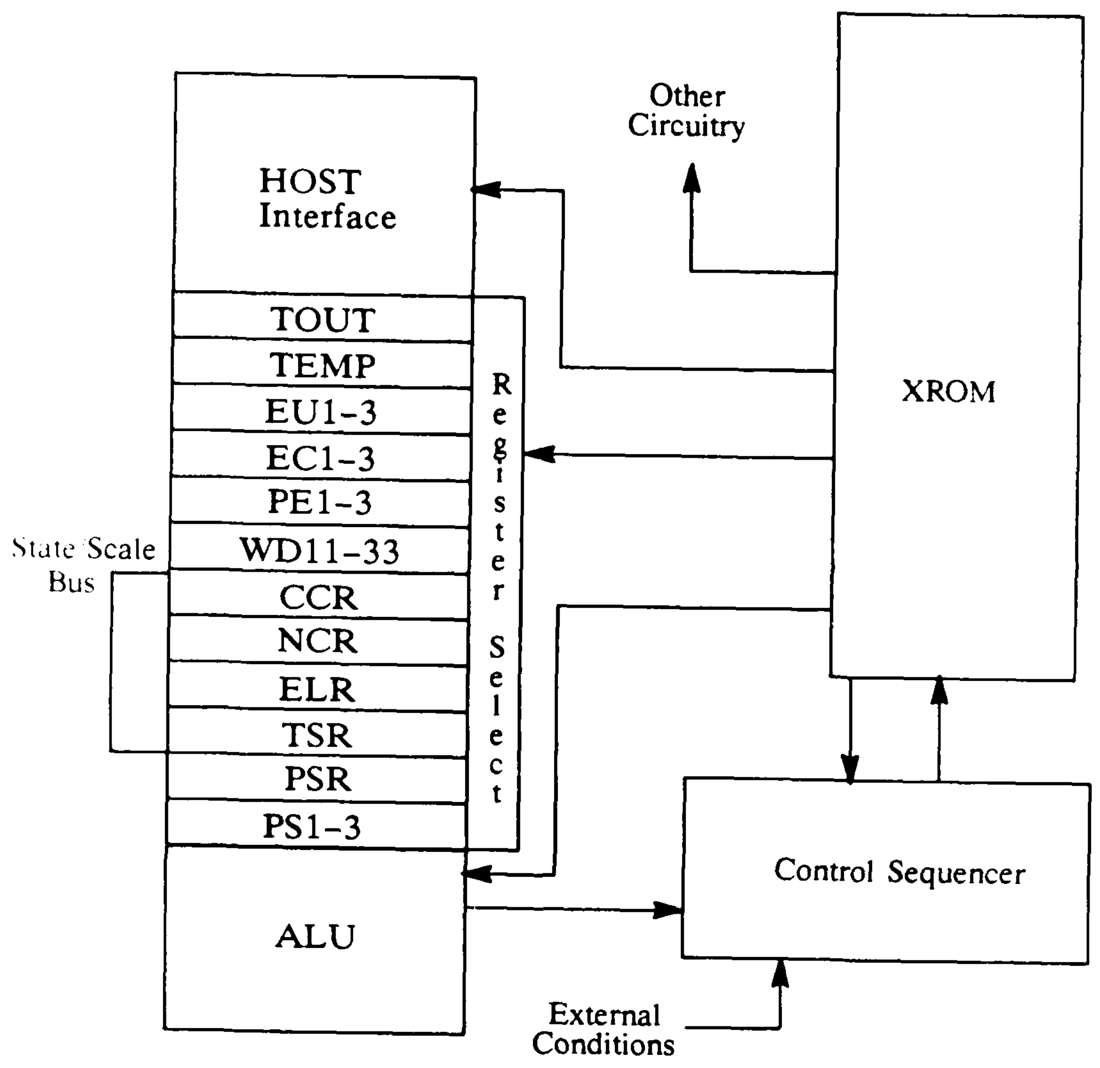


Figure 8 PFA Controller Architecture

and uses a voting strategy to assign the error, if any one of the WatchDog error bit is raised. The voting strategy is as follows:

Table 1: Voting Strategy

Active	Watchdog1	Watchdog2	Result
0	0	0	No Error Assigned
0	0	1	Error Assigned to Watchdog2
0	1	0	Error Assigned to Watchdog1
0	1	1	Error Assigned to Active
1	0	0	Error Assigned to Active
1	0	1	Error Assigned to Watchdog1
1	1	0	Error Assigned to Watchdog2
1	1	1	No Error Assigned

The (1,X,X) cases are conditions that are treated as normal because the possibility of the active signaling a watchdog error is too small to warrant the amount of microcode and hardware necessary for handling this condition. The conditions to cause a (1,X,X) case would be for the active processor to signal that it had a watchdog error. This can only happen when the WFT has suffered a major error, the pipeline has some kind of bus error in which the line is set high, or a transient fault. If the active processor signals a watchdog error and the others do not, this error will be assigned to the active as it should be. The problem arises when the the active signals as watchdog error and one watchdog also signals an error. In this case, the error will be assigned to the watchdog processor that did not signal the error. It is difficult to determine what was the true cause of the error was. The error could be in the active or either watchdog processor and this voting strategy will not find the fault. In the event that all three signal a watchdog error, no error is assigned. The probability of this occurring is extremely small except in a high radiation zone. And, should it occur, the data would probably be corrupted enough so that the next WFT in the pipeline would most likely detect a parity error. If it is the last stage in the pipeline the data will be bad with no error reported.

When the PFA controller executes the code for error recording, it looks at the watchdog errors and determines if one of the three active processors has errored, if so it sets up the Next Configuration Register to make the next active processor to be the current watchdog with the smallest error count. If the two watchdogs have the same number of errors, it selects the processor with the lowest number name (ie. WD*i*1 < WD*i*2 < WD*i*3; where *i* represents the column). The names assigned to the processors are shown in Fig. 9.

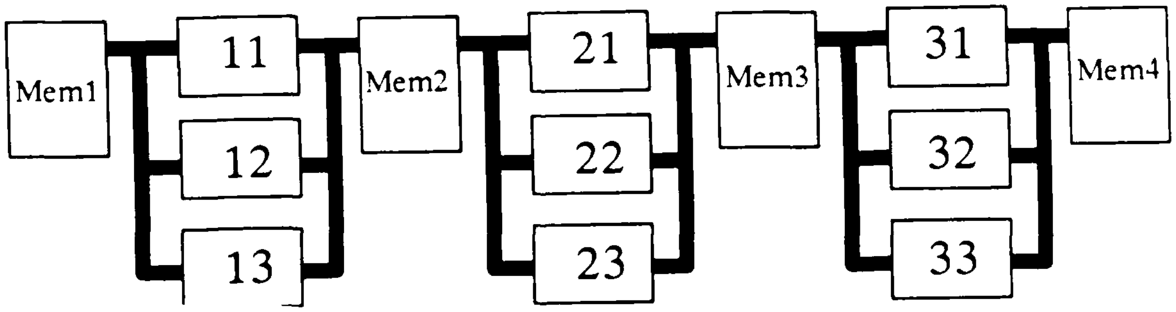


Figure 9 Processor Names

CHAPTER 4

Computer Aided Design Environment Tools

4.1. Overview

The computer aided design (CAD) environment has a great impact on the quality and timeliness of what can be produced. A well-integrated set of tools from the design stage through the implementation stage to the simulation stage allows easier transitions between stages and quicker execution. AFIT currently supports a limited number of tools for the design and implementation phases. In the following sections, the methodology for designing a VLSI chip within the AFIT CAD environment is discussed, descriptions of the tools needed are given, and a description and the development of a CAD tool created as a result of this thesis effort.

4.2. Design Methodology

The first step in producing a VLSI chip to decide what problem to solve. A detailed problem specification limits the scope of what is to be done and thus limits the amount of extra design that might not be needed. The next step is to develop an architectural description from the problem specification. For the design parts of this thesis effort, the architectural descriptions are extremely different. The memory chip architecture supports a data flow architecture and the PFA controller chip supports a finite state machine architecture. Once the architecture has been described, the logic and circuit design follow. At this stage, the interfaces between the macrocells are defined as well as the incorporation of testiblility. The next step is the VLSI layout. Once the layout is complete, the simulation phase begins. Simulation and verification start at the lowest level of design and proceed up the hierarchy. When discrepancies are noted between the simulation and the descriptions, the chip design loops back to various levels depending on where the discrepancy occurred. Once the chip has met the simulation requirements, it is ready for fabrication.

4.2.1. AFIT CAD Tools. AFIT currently supports the CAD environment near the lower design levels. These tools currently support VLSI layout and switch-level simulation. In addition to the software tools, AFIT has considerable computing power and other hardware support for CAD. The hardware used in this thesis effort included:

1. Two ELXSI 6400s,2-CPU(12MIP,6MIP) running UNIX 4.2BSD
3. SUN2 and SUN3 Workstations running UNIX 4.3BSD
4. Two VAX 11/785s running UNIX 4.3BSD
5. A VAX 11/785 running VMS 4.5
6. A Versatec Plotter, and an assortment of printers.

The current mainstay of the AFIT VLSI CAD toolset is Magic [Ost86]. Magic is an interactive VLSI layout tool that allows creation and modification of VLSI circuits using Manhattan circuit design geometries. For this thesis effort, Magic was run primarily on a SUN 3 because of the window environment and interactive speed for cell editing. Magic was run on the ELXSI 6400 when executing several features of Magic that are computationally intensive. This included the design rule checking of a silicon compiled XROM (described later) and hierarchal cell "flattening" of the same XROM. The XROM contains thousands of cells and was found to be easier, and faster, to work with the XROM "flattened."

Mextra [Fit83] was used to translate from the Cal-Tech Intermediate Form (CIF) to a switch-level format suitable for other simulations. Mextra takes the "cif" file, which is a mask level description of the circuit produced by Magic, and makes a "sim" file which is a listing of all the transistors in the circuit and associated capacitances. In addition to the translation, Mextra outputs several other files. These include the alias file, the log file, and the nodes file. The alias file is very useful in finding nodes that are "shorted" together that should not be. The log file gives information about the number of occurrences of labels that are not connected together by stating that a certain label has X number of occurrences. The file provides information that usually lead to finding unwanted "open circuits" between nodes. Finally, the nodes file is a list of the node numbers assigned by Mextra and their location on the chip. This is useful reference when running other tools which refer to node numbers.

Cstat, a CMOS version of stat [Ter86], was run on the output of Mextra. Cstat provides information about nodes that cannot be affected by the inputs, cannot affect the outputs, and nodes that cannot be set to either logic-1 or logic-0. This tool is useful for finding nodes that are not connected or shorted to either Vdd(logic-1) or GND(logic-0). By the time the circuit is ready for fabrication, every node, if any, signaled by Cstat should be accounted for.

Nofeed and Fixrom are two tools needed to allow Esim, described below, to work properly. Nofeed scans the Mextra output to identify and remove the feedback loops from Master-Slave Flip-Flops (MSFFs) making them dynamic. A more complete discussion of the operation of the MSFF will be given in Chapter 5. Fixrom modifies two portions of the XROM for Esim compatibility. The first modification is to replace the shared drain in the XROM storage array and the second is to replace an inverter in the XROM sense amplifier. A thorough discussion of the operation of the XROM will be given in Chapter 5.

Esim [Ter86], is an event-driven switch simulator for nMOS or CMOS transistor circuits. Esim is used to exercise the switch level description from FIXROM. This tool was used to verify the logic created from the Magic layout for correct operation. Simulation can be preformed as if the chip is under test by stimulating only the inputs and observing only the outputs, or as a diagnostic tool by stimulating any node in the circuit and observing any node in the circuit. Once a circuit performs as expected under Esim, it is considered ready for fabrication.

Two tools developed at AFIT were used to create the XROM used in the control section. The first tool was the Generic Microcode Assembler Tool (GMAT). GMAT was developed as part of this thesis effort and is described in the following section. The second tool was an optimizing XROM silicon compiler. This tool, given a list of integer values describing the binary contents of the ROM, will minimize the transistor count and the number of drains. The compiler will also generate the Magic layout for the optimized ROM including the word selectors, column drivers, sense amplifiers, precharge circuitry, and cell arrays

4.2.2. Generic Microcode Assembler Tool (GMAT).

When designing microcode, it is desirable to describe the code in a structured language representation using mnemonics. Describing microcode in these terms is helpful for two reasons; first, the code is more readable and second, the code is less prone to errors. These factors motivate the need for a microcode assembler. The CAD environment needs tools that can be applied to different projects so the assembler must be generic.

To achieve the above needs, a generic microcode assembler tool was developed This tool takes a microcode word format description and mnemonic translation file and builds a microcode assembler for that particular microcode format The resulting assembler reads in the translation file and the microcode to produce outputs that are compatible with other CAD tools and are useful in debugging the integration of the microcode into chip designs. Specifically, it generates an address stream used by the optimizing XROM compiler, a VHDL description of the ROM, a reference file, and a reverse assembly file

The assembler supports several important programming features including labels, literals, and default fields. By supporting the use of labels, jump fields may be specified by a label rather than an absolute address that must be put into the microcode by hand Supporting literals is useful when loading a location with a specified value from the microinstruction. Supporting default values greatly improves code readability The microinstruction need only specify those fields where some action is to take place

The assembler also supports several other features that are useful in the design process. The assembler can output a file that can be fed into the XROM compiler This eliminates a step previously needed and subject to human error The assembler also outputs a reference file that shows the instruction and its translation. This is particularly helpful in debugging the XROM connections in the chip

4.2.3. GMAT Assembler. The assembler created by GMAT is essentially a two-pass assembler as described by Beck [Bec85 On the first pass unsed parts are stripped and branch labels are stored for access on the second pass As the assembler scans the input file on the first pass, it writes out an intermediate "stripped file" and a listing file containing the input file and associated line numbers

On the second pass the translation is accomplished First the assembler reads the translation file into internal data structures The translation file created by the user defines the microword format the fields and the values they represent The assembler now reads the "stripped" microprogram and uses the translation tables for the symbol substitution. When the assembler parses a line, it starts with the first symbol encountered and searches through the list of names in the first field table. If the symbol is encountered, it makes the translation and outputs the result to a data file then gets the next symbol. If a symbol is not found, the assembler puts in the default value for the field and gets the next symbol. This way the writer need not specify all fields in the microcode. This enhances readability and decreases the chances of leaving a field out if they all must be declared.

A label is treated like any other field. When the label is encountered in the word, the assembler puts in the value of the line number at which the label appeared in the first pass. It also translates the line number to a 1 0 representation with the same number of bits as the field where it is being placed. This allows the microcode to use labels for the branch fields instead of having to figure out the absolute location of the branch and manually insert it into the microcode word. The assembler also allows insertion of literals into the translation. When a pound sign is encountered, the field allowing literals will be filled with exactly what follows the pound sign. This allows constants to be used in the microcode. If the assembler does not find a symbol after parsing the entire line, two conditions may have occurred. First, the symbol could have been a "nop", or second, the symbol is an error. The "nop" represents a "no operation" instruction. The assembler checks for a "nop" at this point. If the symbol is not a "nop" then it reports this back to the user as an error and indicates the symbol that was not found.

After the second pass, the assembler has created a data file that holds the translation for the entire microcode. The assembler now creates a reference file for the user

This file lists the original instruction and following it, the translation by fields. This reference file is extremely useful in debugging the microcode and the chip once the ROM is in place.

The remaining features of the assembler are invoked as a command line switch when running the assembler. The user may generate a reverse compile, an XROM address file, or a VHDL description of the ROM. The reverse compile takes the file of translated symbols and reverse assembles it. If the code is correct, the reverse compile will produce the original microcode with labels removed and line numbers inserted. The assembler may also generate the XROM addresses. In this section, the assembler transforms the translated file into a form compatible with the XROM optimizing compiler. The assembler separates the word into four parts and computes the integer value of the resulting binary number. These four integers are then put into the XROM addresses file. The assembler does this for all words in the microcode. The final option currently supported is a VHDL description of the ROM. The assembler generates a VHDL package that holds the ROM and defines how the XROM is interfaced. The interface allows words to be read from the ROM. The ROM is represented as an array of bit strings.

4.2.3.1. Translation File The translation file contains the microcode word format and the field definitions, as well as their translation. The following is a representation section of the translation file used for the PFA controller:

```
BR_SEL ALU LOAD_FD REG REG REG SPEC_FUNCT WFTOP PFADONE NXT_ADDR # .

BR_SEL                                              000000000
RET                                                 000000001
CALL                                                000000010
JMP                                                 000000011
CALLCR                                              000010110
JnPE3                                               111111111
```

ALU	0000
AND	0010
XOR	0011
OR	0100
LOAD_FD	0
LOAD	1
REG	00000
ECCC1	00001
ECCC2	00010

The first line, it must be the first line, contains the microword format ended with a semi-colon. The end-of-line delimiter is used in case a format description is longer that one line. Fields in the microword format must appear below in the translation file. If a '#' follows a field in the format description, this field might contain a literal. If a ':' follows a field, this field might contain a label. In the above example, the NXT_ADDR field contains both. For the PFA controller this field is the jump field as well as the literal field.

The field definitions that follow the format description are separated by one blank line and end with the last field. The first line of a field definition consists of the field name followed by its default value. Blank spaces are used to separate the values, not tabs. The remaining lines in a field definition specify the sub fields and their translation. In the above example, the REG field default value is 0000 and the value of register ECCC1 is 0001. When the assembler encounters the symbol ECCC1, it will place 0001 in the translated file.

4.2.4. PREG Operation. PREG is the interface to the user's microcode. PREG reads the microword format and scans the translation file to build those sections of code for the assembler that are user specific. The code segments generated are then written to files that are added into the assembler when it is compiled.

PREG first scans the translation file for the microword format and which fields may contain labels or literals. It then scans the translation file and records the names of the fields and the lengths of the bit translation fields. The bit lengths are needed when making the code that generates the reference file where the translated microinstruction fields are separated by a space for readability. The length of the label field is also needed so the translation from the line number to the binary representation of the proper bit length can be done.

After having scanned the translation file, PREG builds the user-specific code. The first file created, *assem.h*, is a header file containing definitions needed to implement the code. These definitions include the word length in bits, the label field length in bits, structures for the fields found in the translation file, and defines integers to record the number of subfields for each field definition. Because the assembler dynamically assigns these value when it reads the translation file, the number of subfields for each field may change without needing to rerun PREG.

The next file created by PREG is the *assem.tailored* file. This contains the routines to read the translations file, translate the microcode, make the reference file, and preform the reverse assembly. The routine to read the translation file reads the fields in the order found when PREG scanned the translation file. The routine to translate the microcode uses the microword format and the names included in the translation file. The reference file routine uses the bit lengths to separate the fields when the reference file is created.

The reverse assembly routine parses the translated file into fields and then translates the fields backwards into mnemonics, except for the jump field which is replaced with a line number.

To make a tailored assembler, the user runs *gmat* a shell script that runs PREG and compiles the resultant assembler. Appendix B shows the gmat shell script. Appendix C shows the code for PREG. Appendix D shows the two files created when GMAT was run for the PFA controller. Appendix E shows the code for the assembler skeleton. translation file for the PFA controller is shown in Appendix F.

CHAPTER 5

VLSI Design

5.1. Design Techniques

Before describing the the VLSI implementation, it is necessary to explain several different VLSI design techniques used. These are 2-phase clocking, design with transmission gates (t-gates), and master-slave flip-flops (MSFFs).

Two-phase clocking employs the use of two non-overlapping clocks to synchronize operations on the chip. [Wes85]. A timing diagram is shown in Fig. 10. The two clocks, PQ1 and PQ1, each have separate operations. The inputs to all logic units that are synchronized with the clocks become valid on the rising edge of PQ1, falling edge of precharge if precharging is used, starting all computations. The outputs are latched on the falling edge of PQ2, ending all computations. Two-phase clocking is useful for synchronization in the circuit. It also prevents some signals from racing through flip-flops destroying the intended sequencing.

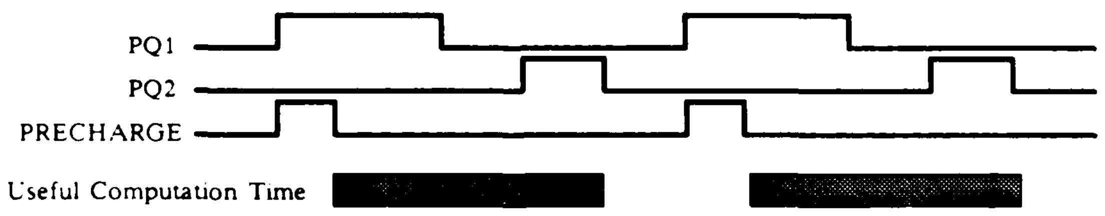
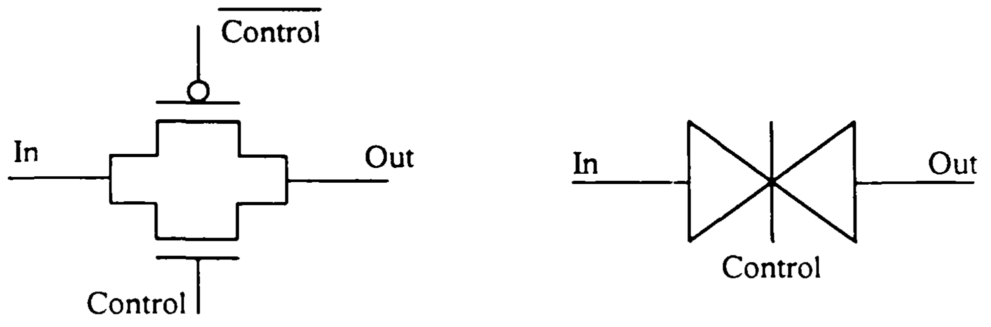


Figure 10 2-Phase Clocking Timing Diagram

The second design style is the use of transmission-gates (t-gates). T-gates are the CMOS switch, the equivalent to pass transistor in nMOS logic. A t-gate is shown in Fig. 11(a). The transmission gate is made up of a p-transistor and an n-transistor in parallel. Both types of transistors are because a p-transistor will not pass a strong logic-0, that is, it passes no lower then approximately 1.7v, and a n-transistor will not pass a strong logic-1, that is, it passes no higher than approximately 3.3v. By using both, a good switch with both a strong logic-1 and logic-0 is designed. The control for the t-gate is supplied to the n-transistor and its complement is supplied to the p-transistor so that both transistors are on when control is high on the n-transistor and its complement is low on the p-transistor. When using the t-gate symbol, only the signal to the n-transistor is shown in Fig. 11(b). The complemented input to the p-transistor is still needed, but not shown.

A MSFF is shown in Fig. 12. The MSFF is the basic storage unit. The input is latched on the falling edge of PQ2 and remains in the first feedback loop until the rising edge of PQ1, at which time it moves into the storage area on the right where it is latched



(a) Transmission Gate (b) Transmission Gate Symbol

Figure 11 Transmission gate

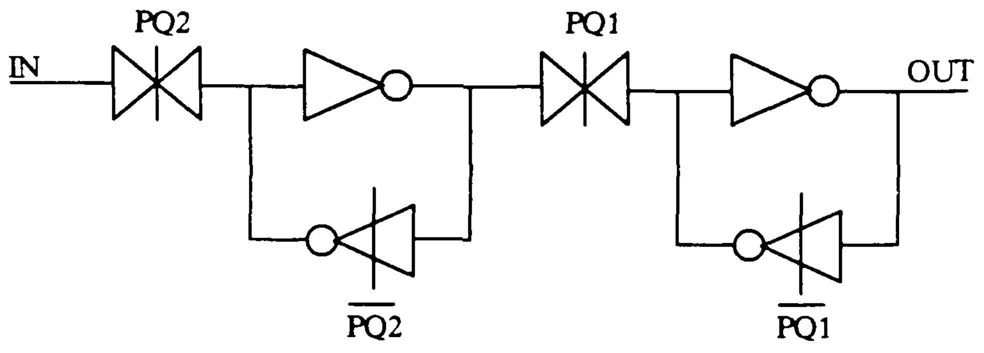


Figure 12 MSFF Description

on the falling edge of PQ1. The feedback loops keep the value of the nodes as long as power is supplied to the circuit or until a new value is loaded. When PQ1 rises, the value is output to the circuitry using the stored signal.

5.2. Memory Chip

The memory chip includes the storage cells for the bits, the encoding circuitry for the error correcting, the decoding circuitry for the error correcting, as well as other circuits for bitline control, word selection, bitline detection, and one-shot generation. Figure 13 shows the chip architecture including all the major components.

5.2.1. Memory Cell. The memory cell is based on a one transistor cell design shown in Fig. 14(a). The one-transistor design was chosen to increase the density of the memory. The cifplot of the memory cell is shown in Fig. 14(b). Each memory cell actually holds two bits, one associated with the wordline above and the other with the wordline below. This was done for several reasons including modularity, density, and capacitance reduction. Modularity is obtained by designing the cell to be arrayed in any direction. The density is obtained because the cell is so tightly packed. The final reason, and

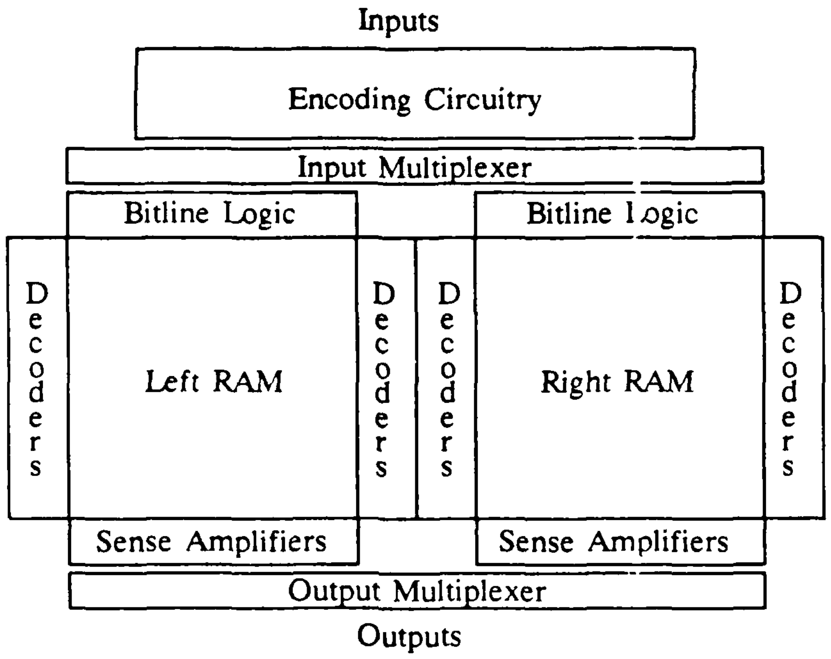
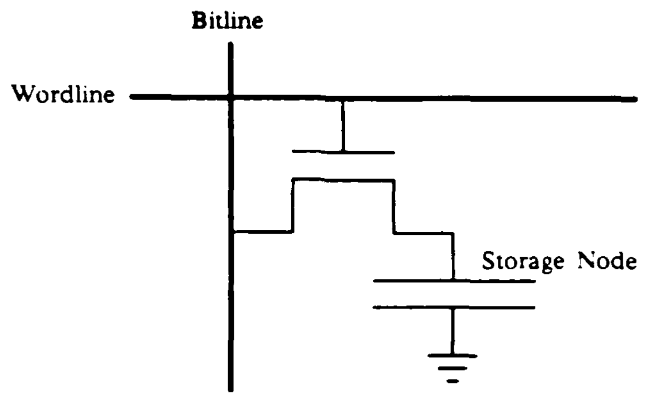


Figure 13 Memory Chip Architecture



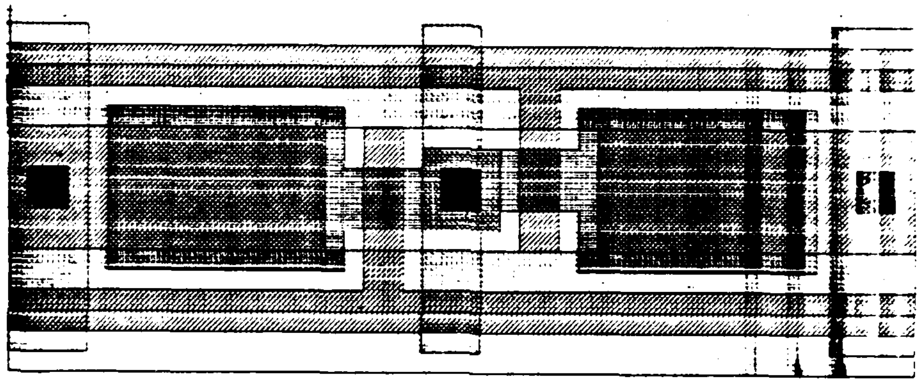
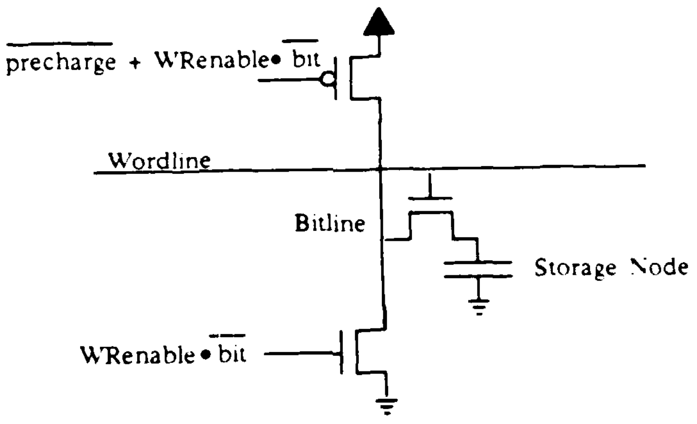


Figure 14   One-Transistor Memory Cell Logic and Cifplot

the most important, is the reduction of the bitline capacitance. The sources of all the memory storage cell transistors are attached to the bitline. If transistor source sharing was not used, there would n sources on the bitline where n is the number of words in the array. In this implementation, there are only $\frac{n}{2}$ sources. The reason for keeping the capacitance of the bitline as small as possible is related to reading a memory value. Reading is based on charge sharing between the bitline and the storage node. The wordlines run horizontally across the cell in polysilicon and second metal. The polysilicon and the second metal are shunted together at both sides of the array to decrease the effective resistance of the polysilicon thereby decreasing the access time. The bitlines run vertically in first metal. The actual memory storage area is n diffusion area connected to the drain of the pass transistor on the bitline. A grounded polysilicon plate sits above the area of n-diffusion. The charge for the cell is stored on the capacitor created between the polysilicon and the n-diffusion. When the wordline is raised, the precharged bitline will either maintain its value because the charge stored on the capacitor was high and no charge sharing takes place, or experience a decrease in charge because the value on the polysilicon plate was low and charge sharing between the bitline and the cell takes place. The decrease in charge will be less than 0.02 volts because the capacitance on the bitline is so much greater then the capacitance of the memory cell.

5.2.2. Bitline Control. The bitline is pulled up and down for reading and writing. Figure 15 shows the circuitry for the bitline control. The bitline is precharged before every read and before writing a logic-1, and pulled down before and during writing a logic-0. The three signals that determine the state of the bitline are



$\overline{\text{precharge}} + \text{WRenable} \bullet \overline{\text{bit}}$
Wordline
Bitline
Storage Node
$\text{WRenable} \bullet \overline{\text{bit}}$

Figure 15 Bitline Control Circuitry

Write_enable(Left from PFA), $\overline{precharge}$ (PQ1 from WFT), and bit (data to be written from WFT). These three signals control the $\overline{bitline\_precharge}$ (active low) and *bitline_pulldown*. The above is summarized below:

Table 2: Bitline Control Logic

WRenable	$\overline{precharge}$	bit	$\overline{bitline\_precharge}$	*bitline_pulldown*	commment
0	0	X	0	0	precharge before read
0	1	X	1	0	read
1	0	0	1	1	pull dn before write 0
1	0	1	0	0	precharge before write 1
1	1	0	1	1	pull dn during write 0
1	1	1	1	0	write 1

5.2.3. Sense Amplifiers. The sense amplifier design is shown in Fig. 16. The sense amplifier is used to detect a slight drop in the bitline voltage if the memory cell stored a logic-0. The sense amplifier is based on a differential voltage amplifier. A dummy voltage reference is used for the comparison. The dummy bitline is connected to a

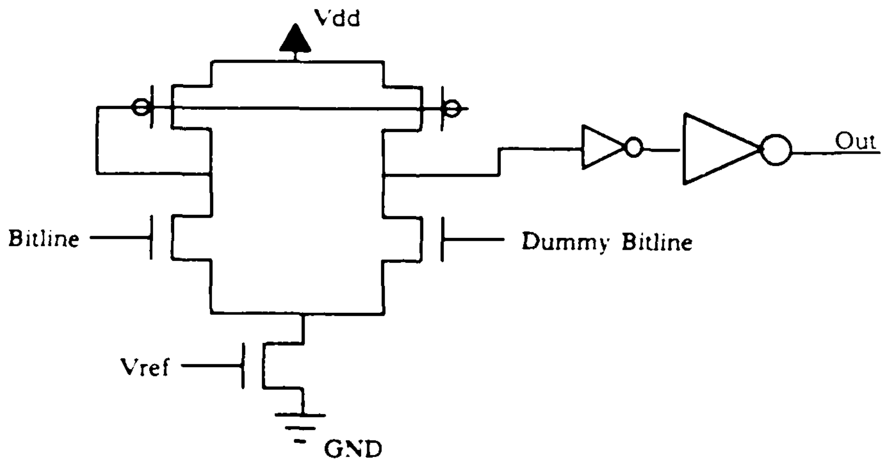


Figure 16 Sense Amplifier

column of memory cells that always have a stored value of 2.5 volts from a voltage divider. Vref is used to maintain a constant current in the amplifier. When a logic-1 is read and the bitline voltage does not change, it will be higher than the dummy bitline. This will cause its transistor to be fully on, causing the above node's voltage to drop. The p-transistor above the dummy is turned on, pulling the output toward logic-1. When the bitline is lower than the dummy, more current will be drawn through the dummy transistor pulling the output to logic-0.

5.2.4. Word Selection. The word selector is based on a NOR of the address bits. The design is shown in Fig. 17. The NOR approach was chosen over the NAND approach because the resistance of the consecutive gates in the NAND design increase the time needed for the output to change. The inputs are selected for the NOR as either *address* or $\overline{\textit{address}}$ in such a way as to prevent pulldown when the word is selected. The NOR had previously been pulled up by $\overline{\textit{precharge\_shot}}$ and the address select lines are gated with $\overline{\textit{precharge\_shot}}$ to prevent inadvertent pulldown of a NOR output while it is being pulled up. If the NOR output is logic-1 when $\overline{\textit{precharge}}$ transitions to logic-1, the value of the NOR passes through the NAND gate. This output is inverted to become the

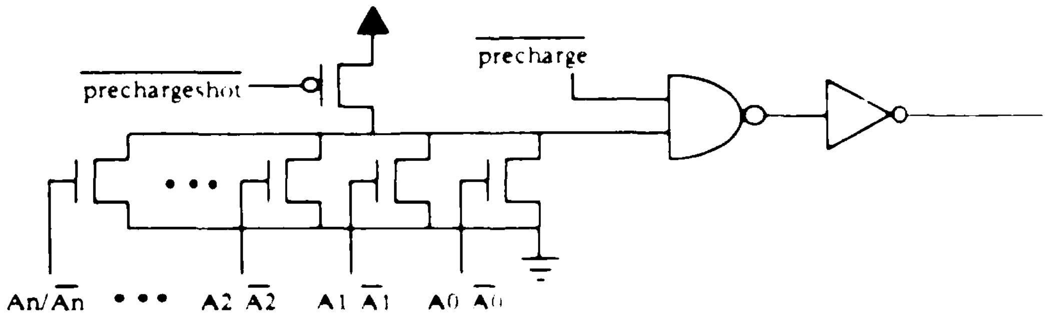

$\overline{\text{precharge}}$
$\overline{\text{prechargeshot}}$
An/$\overline{\text{An}}$
• • •
A2 $\overline{\text{A2}}$
A1 $\overline{\text{A1}}$
A0 $\overline{\text{A0}}$

Figure 17. Wordline Selector Decoder

wordline select. If any one of the word select lines pulls the NOR [illegible] low, the wordline is not selected.

5.2.5. One-Shot. The [illegible] for the word [illegible] is [illegible] by *precharge_shot*. This signal is generated by a one-shot [illegible]. The [illegible] of the one-shot signal is to allow the NOR circuit to be [illegible] state [illegible] before precharge transitions to logic-1 thereby decreasing the time needed for wordline select. [illegible] the precharging and the wordline selection are overlapped. [illegible] described below.

Table 3. One-Shot Logic Table

precharge	*precharge*_δ	*precharge_shot*
1	1	1
0	1	0
0	0	1
1	0	1

Where the two inputs are *precharge* and *precharge*_δ, the [illegible] signal being *precharge* delayed by δ. The length of the one-shot is [illegible] NAND gate delay. [illegible] achieved by feeding the *precharge* through five inverters.

5.2.6. ECC. The ECC consists of two major sections: the Encoding section and the Decoding section.

5.2.6.1. Encoding. The encoder circuit computes the eight parity bits added to the word for error correction and detection. Each of the eight parity bits is derived from a logic function based on six different bits of the input word. Each parity bit is a unique combination of six input bits. The equations were described in Chapter 3. In boolean logic the addition operation is the same as the XOR logic function. Thus each

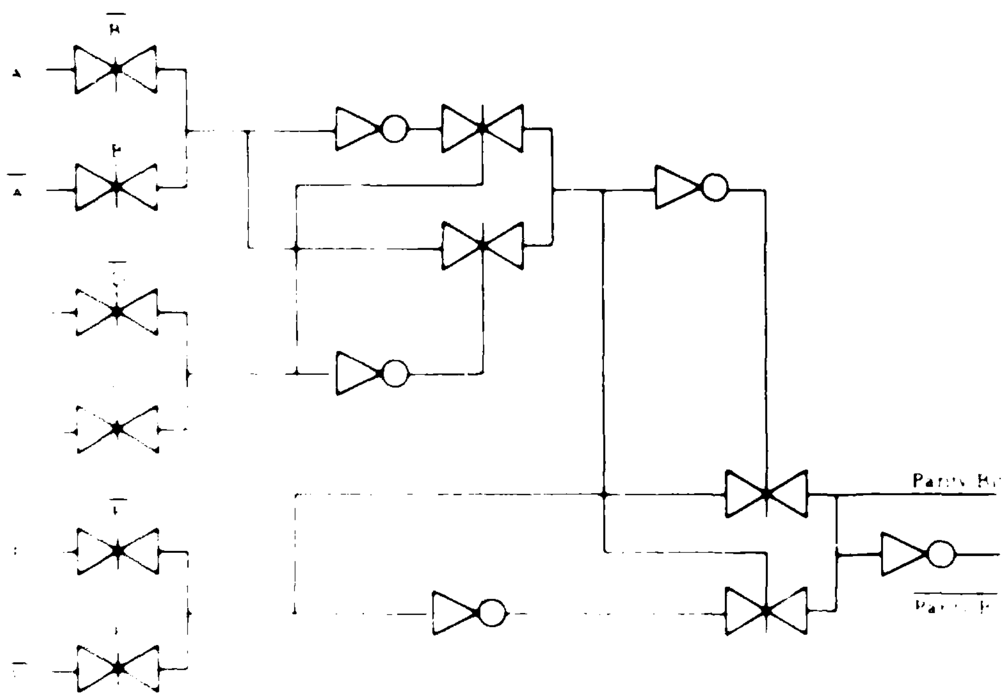


Figure 18. Parity Bit Generator

parity bit is the XOR of six inputs as shown in Fig. 18. The inputs A-F represent the six different inputs for the parity digit. This circuit is duplicated eight times, once for each of the parity bits. The parity bit and its complement are then fed directly to the [illegible] control logic.

5.2.6.2. Decoding. Decoding consists of three steps: first, the eight syndrome bits are generated; second, the syndrome bits are used to compute the error vector bits; and finally, the error vector bits are XORed with the values read out of the memory.

The syndrome bit generation is similar to the parity bit generation except the syndrome bits are the XOR of 7 signals as shown in Fig. 19.

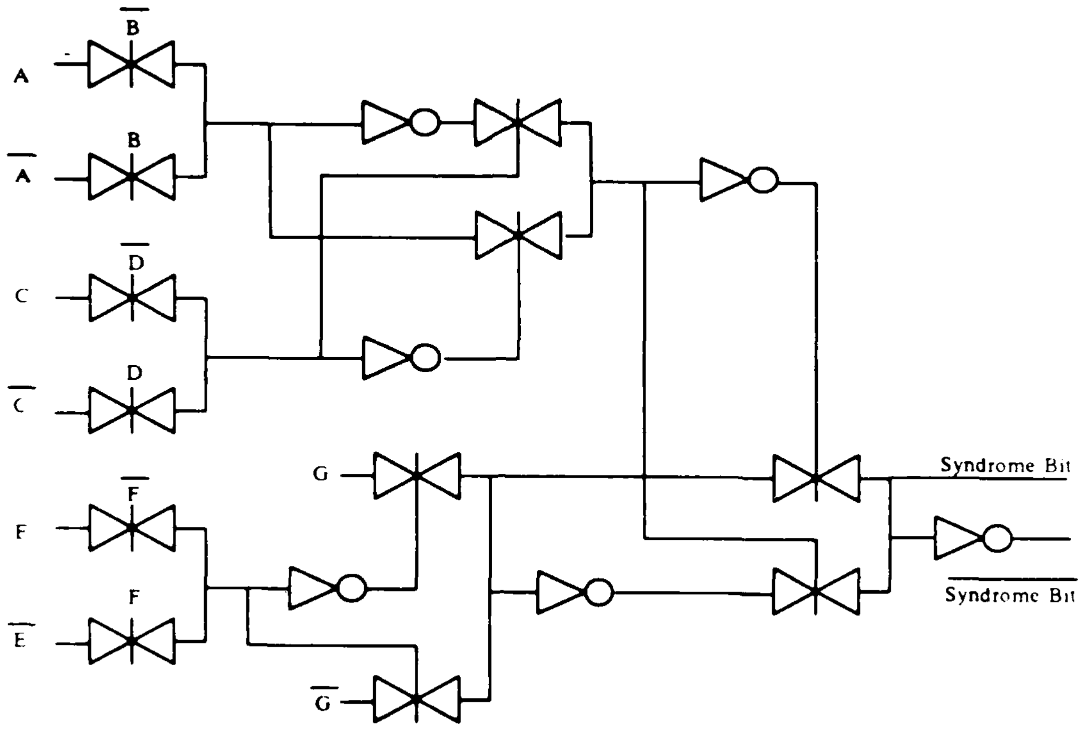


Figure 19 Syndrome Bit Generator

The computation of the error vector bits may be done two different ways, the eight syndrome bits could be fed into a Programmed Logic Array (PLA) or each error vector bit could be computed using custom logic. The PLA method is not the most efficient because each error vector bit is the sum of only one product. This would leave a great deal of area not being utilized. Custom logic is thus the implementation choice. To increase the ease of design, a basic cell is used and then personalized with smaller cells to select the desired logic configuration. The logic for the error bit generator is similar to the address decoder except that no gating is done with $\overline{\textit{precharge}}$. Finally, the error vector bits are fed into a 2-input XOR gate to produce the final output as shown in Fig. 20. The other inputs are the bits read out of the memory. The ECC is designed to produce error control for all 32 bits (24 data, 8 parity) but the 8 parity bits are not used outside

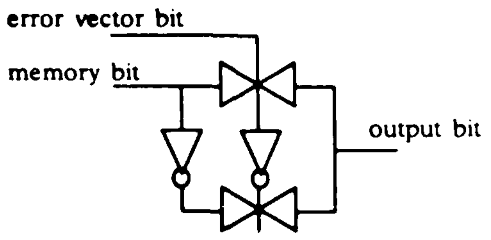


Figure 20 Out Bit Generator

of the chip, so there is no need to correct them. This reduces the number of XOR gates to 24.

5.2.6.3. ECCC and ECCU. The Error Correcting Code Corrected signal sent to the PFA is determined based on the syndrome bits. If the number of syndrome bits at logic-1 is one or two, then a single error was found and can be corrected. Thus the ECCC signal can be implemented using a PLA with 24 product terms of 8 bits each. The Error Correcting Code Uncorrected signal sent to the PFA is also based on the syndrome bits. If the number of error vectors bit at logic-1 is greater than two, then a double error was detected and may be corrected. The correction cannot be guaranteed, however. Thus the ECCU signal could be implemented using a PLA with $2^7 - 24 = 104$ product terms of eight bits each. A PLA with 104 product terms is prohibitively large for any application. The solution for these signals can be implemented using analog circuit techniques very easily. The circuit is shown in Fig 21. When one or two of the error vector bits is high, the input to the inverters will drop no lower than 2.3 volts. This will trigger the top inverter, designed to switch at 2.5 volts. When more than two of the error vector bits is raised, the inputs to the inverters will drop below 2 volts. The bottom inverter is designed to trigger below at 2 volts. The logic is summarized as follows

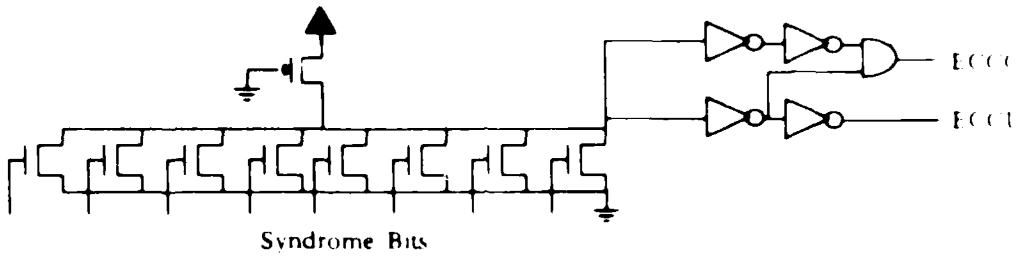


Figure 21 ECCC ECCU Circuitry

Table 4 ECCC and ECCU Determination

A	B	Result
0	0	no errors
0	1	not possible
1	0	ECCC
1	1	ECCU

5.2.7. Switching Circuitry. The switching circuitry is used to control the flow of the data to the memory arrays The data flow is shown in Fig 22 The word select lines must come from two different sources, one source for reading and the other for writing The input data is channeled into the memory side being written and the output data is extracted from the side of the memory being read from The signal to accomplish all of the necessary multiplexing is the *Write_enable* signal generated by the PFA controller as *LEFT*

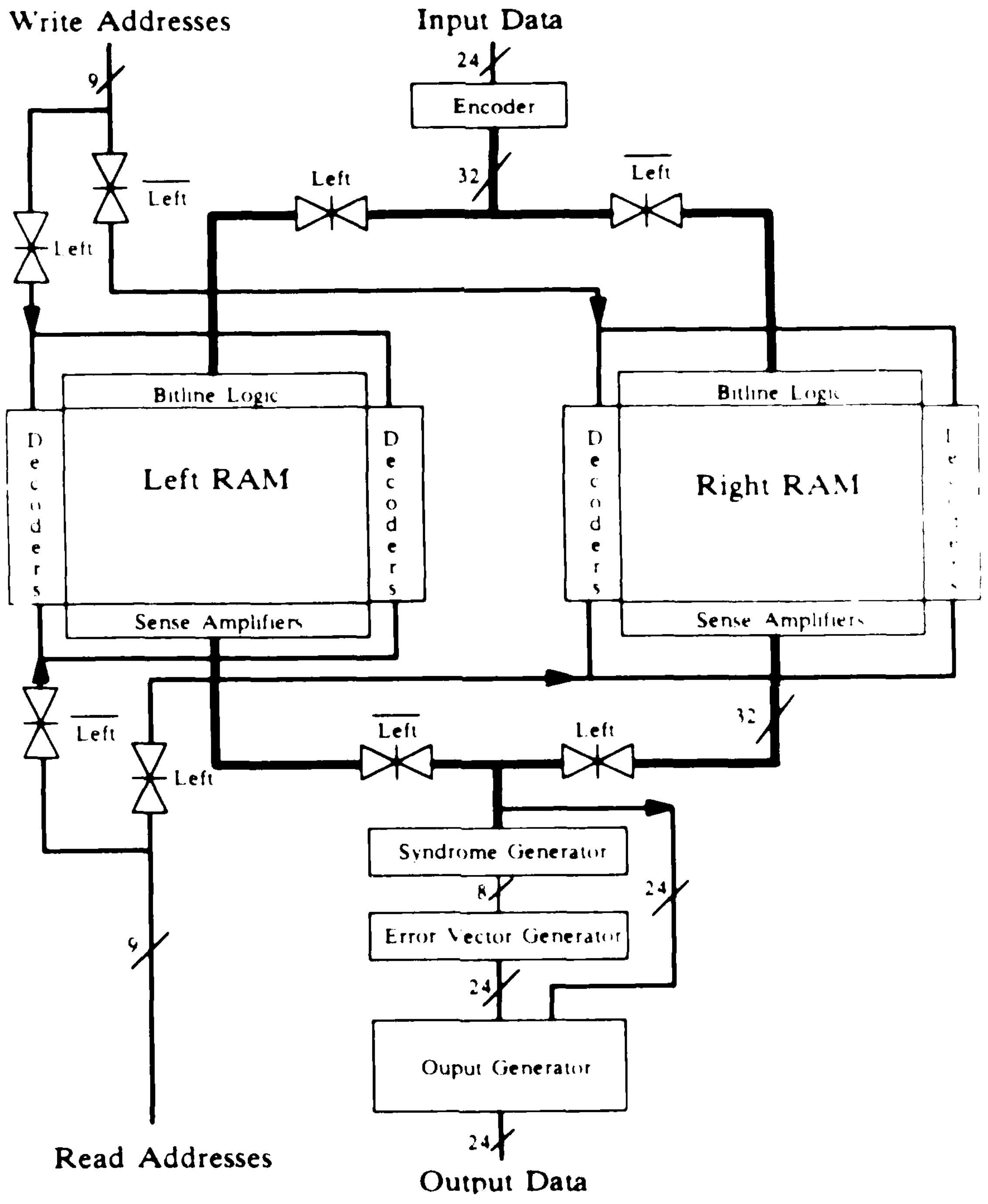


Figure 22 Memory Chip Data Flow

5.3. PFA Chip

The PFA controller has three major sections, the control section, the data section, and the periphery These are shown in Fig 23

5.3.1. Control Side. The control side of the PFA controller is responsible for generating the control signals used to operate the PFA controller and the pipeline It does this by sequencing through a set of microinstructions stored in a read only memory The address of the instruction to be executed is generated by the control sequencer based on various control signals from the current instruction and certain state variables generated by the environment The control section was described in Chapter 3 and shown again in Fig 4 The major sections include the control memory, the next address generator the condition select the stack, and the incrementer

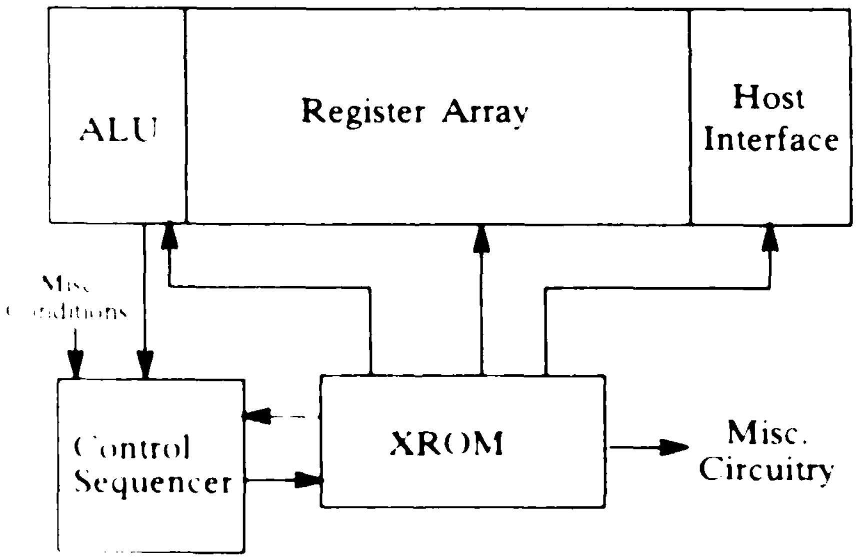


Figure 23 PFA Major Components

5.3.2. XROM. The control memory is implemented using a read-only memory (ROM) developed at AFIT. This ROM, the AFIT XROM was designed by Paul Rossbach in 1985 [Ros85] and its general structure is shown in Fig. 24. The vertical pitch of the memory cells is such that decoders are needed on both sides of the ROM to access all of the words. The wordlines run horizontally in polysilicon and second metal. The horizontal pitch of a memory cell is so small compared to the sense amplifier that some column decoding is needed. The sense amplifier is four times the horizontal pitch to run a single bitline. Therefore, two of the address lines are fed into the sense amplifiers to select one of four bitlines. Additionally, the LSB of the address lines is fed into the column drivers at the bottom to select which bit is "A0."

5.3.2.1. XROM Memory Cell. The AFIT XROM memory cell is shown in Fig 25 The name XROM is derived from the 'X' shape of the transistors around a common drain [Ros85]. Before a read, each of the bitlines is precharged to logic-1 through an n-transistor resulting in a voltage around 3.3 volts. When a wordline is selected the

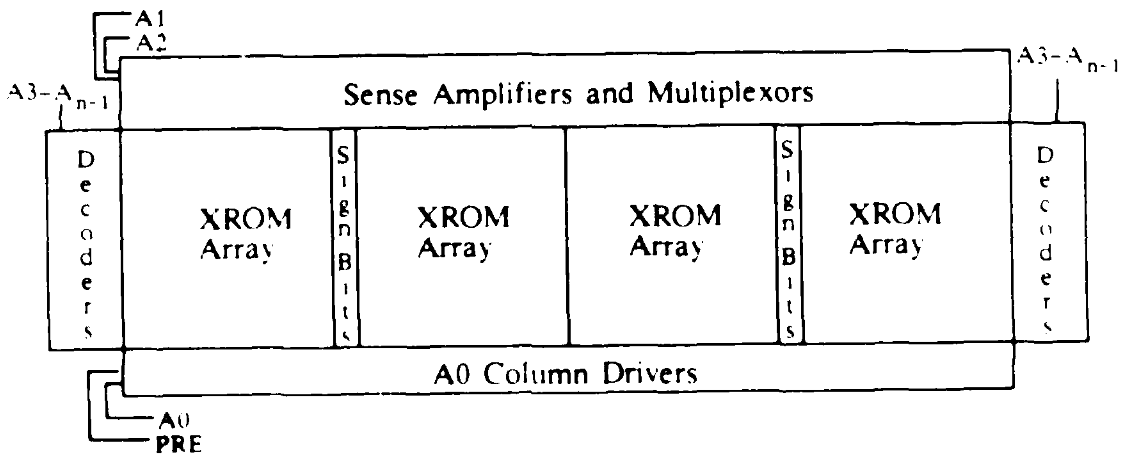


Figure 24. AFIT XROM Structure

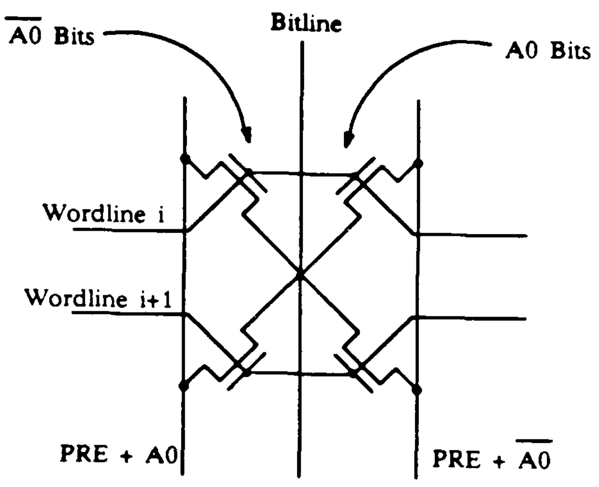


Figure 25 XROM Memory Cell

transistors connecting the bitline with the $A0$ or $\overline{A0}$ address line turn on allowing the bitline to discharge through whichever of the two is tied to logic-0. When the bitline is discharged to logic-0 this indicates the presence of a transistor and thus, a stored value of logic-1. If no transistor is present, the bitline will not discharge and the sense amplifier recognizes this as a stored logic-0. If both $A0$ and $\overline{A0}$ are connected to the bitline, "fighting" will occur and the bitline voltage will settle to 1.5 volts. Since the sense amplifier is set to trigger above the n-transistor at 4 volts, it will correctly recognize this as a logic-0.

5.3.3. XROM Sense Amplifier. The sense amplifier is used to detect the value on the bitline and amplify it to a full 5 volts or 0 volts. The implementation is shown in Fig. 26. Initially, the bitline is precharged through the p-transistor at the top of the cell. The two address lines which select one of the four bitlines are already stable. Thus at the end of precharge, the bitline below the selected n-transistor is at 3.3 volts. If a transistor is present at the word selected, the bitline will discharge to approximately 0

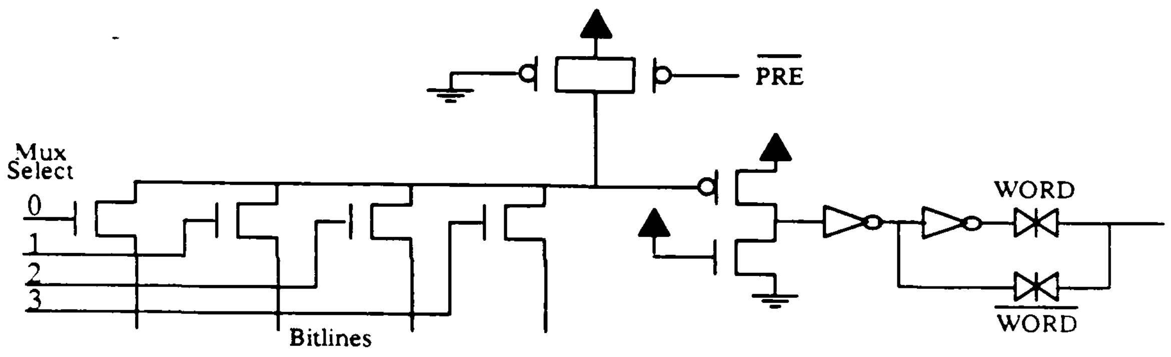


Figure 26 XROM Sense Amplifier

volts, this causes the gate connected above the bitlines to turn the p-transistor on, raising the output to logic-1. The two transistors before the two inverters are designed so that they become an inverter triggered a 4 volts. The n-transistor gate is not connected to the p-transistor as in regular inverters to minimize the capacitance on the sense line. When no transistor is present, the bitline does not discharge, the n-transistor pulls the output of the pseudo-inverter down, and a logic-0 is output. If two transistors were on the bitline, it will settle at 2.5 volts below the n-transistor. This will drop the sense line enough to turn the p-transistor on the pseudo-inverter output a logic-1 as expected.

5.3.4. XROM Pipeline Register. The XROM pipeline register sits above the XROM sense amplifiers and was developed by Capt. David Gallagher [Gal87]. Using a pipeline increases the utilization of the XROM. When a pipeline is used, the last word out of the XROM is being executed while the next word is being fetched. By overlapping the execution and fetch operations, the effective speed of the controller is doubled. Without a pipeline, it would take one complete clock cycle to fetch the word and another to execute, then the next word would be fetched in one clock cycle and executed in another and so on. With a pipeline, however, the clock cycle needed for fetching is hidden by the execution of the last word fetched.

The pipeline works by using a modified MSFF to store the instruction from the XROM output and isolate it while the XROM fetches the next word. Additionally, the pipeline may be logically separated from the XROM for testing purposes. In this mode, the pipeline becomes a shift register controllable from several chip pads.

5.3.5. Control Sequencer.

The control sequencer, designed by Larry French [Fre86] and modified by David Gallagher [Gal87], determines the addressing for the XROM. It consists of five main blocks, address selection, condition select, branch select, incrementer, and stack.

5.3.5.1. Address Selection.

The address select block selects the next address for the XROM from four sources. The first is the next address field of the current instruction. Allowing the next address to come from an instruction enables the microcode to branch. The second source is the top of the stack. Using a stack allows the use of subroutines in the microcode. The stack will store the address of the instruction following the call so that program control can return to that point. The third source is the incrementer allowing sequential addressing. The fourth source is an external source. This external source is used to manually control the sequencer for testing of the controller. In a more general processor, this source can be used to map functions from a register

5.3.5.2. Condition Select.

The condition select block is used to select one of thirty-two possible conditions. These conditions are used to determine conditional branches, calls, and returns. The original 32:1 mux routing was slightly modified for this thesis effort to allow access to all 32 condition inputs. French gives a detailed discussion on the construction of the 32:1 mux in his thesis [Fre86]. The condition selected depends on the value of the 5 conditional mux select (CMS) lines generated from the XROM. The conditional input, along with the values needed to select it appear below.

Table 5: Condition Selects

Condition Mux Select CMS					Condition	Condition Mux Select CMS					Condition
0	1	2	3	4		0	1	2	3	4	
0	0	0	0	0	not defined	0	0	0	0	1	not defined
0	0	0	1	0	ERROR	0	0	0	1	1	Negative Flag
0	0	1	0	0	Zero Flag	0	0	1	0	1	PFA Operate
0	0	1	1	0	4 Done	0	0	1	1	1	Watch Dog Error
0	1	0	0	0	WD Err Col 1	0	1	0	0	1	WD Err Col 2
0	1	0	1	0	WD Err Col 3	0	1	0	1	1	WD Err 11
0	1	1	0	0	WD Err 12	0	1	1	0	1	WD Err 13
0	1	1	1	0	WD Err 21	0	1	1	1	1	WD Err 22
1	0	0	0	0	WD Err 23	1	0	0	0	1	WD Err 31
1	0	0	1	0	WD Err 32	1	0	0	1	1	WD Err 33
1	0	1	0	0	Error Uncorrected	1	0	1	0	1	EU Mem.1
1	0	1	1	0	EU Mem.2	1	0	1	1	1	EU Mem.3
1	1	0	0	0	Error Corrected	1	1	0	0	1	EC Mem 1
1	1	0	1	0	EC Mem 2	1	1	0	1	1	EC Mem 3
1	1	1	0	0	Parity Error	1	1	1	0	1	PE Mem 1
1	1	1	1	0	PE Mem 2	1	1	1	1	1	PE Mem 3

The reasons for selecting these particular conditions will be explained when the microcode development is discussed.

5.3.5.3. Incrementer. The incrementer can be implemented by placing a full adder on each bit, setting one of the inputs to zero and the other to the previous address, setting the carry in of the least significant bit to 1, and letting the carry propagate through all 7 bits. Since the second input to the adders is always zero the equations can be simplified from

$$Sum = A \text{ } xor \text{ } B \text{ } xor \text{ } Cin \text{ } and$$

$$Cout = A \; (A \text{ } xnor \text{ } B) + Cin \; (A \text{ } xor \text{ } B)$$

to

$$Sum = A \text{ } xor \text{ } Cin \text{ } and$$

$$Cout = A \text{ and } Cin.$$

Additionally, since Cin to the LSB is always 1, the equations for this bit simplify to $Sum = \overline{A}$ and $Cout = A$. Thus, all bits but the LSB can be implemented with half-adders as shown in Fig. 27. The LSB can be implemented simply using a single inverter

5.3.6. Subroutine Stack. The stack is used to store the return address for a call. On a cell, the address of the next instruction is pushed on the top of the stack. On a return, the top of the stack is poped, and this becomes the address of the next instruction. For the PFA controller microcode, there will only be one call active at a time meaning that the stack only needs to store one address. Figure 28 shows the stack for more than one stage to illustrate the stack operation. When *Push* is activated, the input is fed into the first MSFF and the values of the following MSFFs are fed into the next MSFF. The 2-phase clocking keeps the values from overwriting each other until the proper time. When the *Pop* is activated, the output comes from the top of the stack and all other

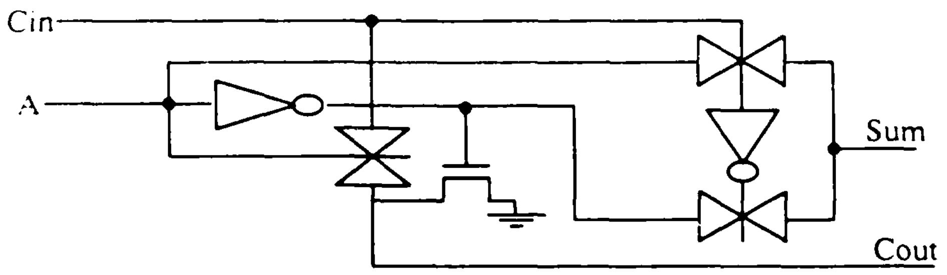
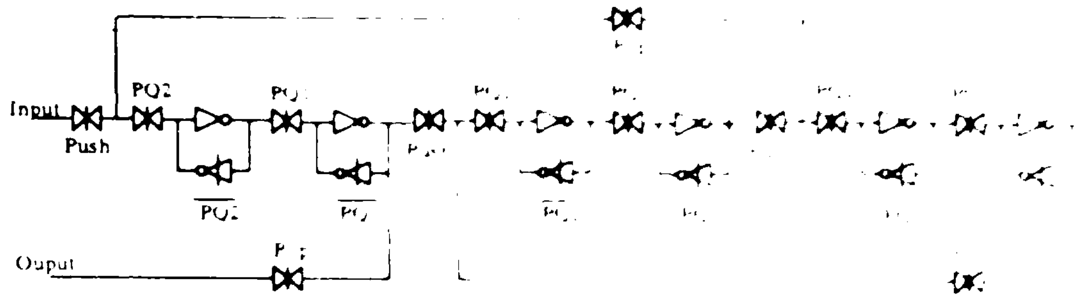


Figure 27 Incrementer Half-Adder



Fig[illegible] 2[illegible]

values are fed into the inputs [illegible]

5.3.6.1. Branch Selection [illegible] the next address and generate the [illegible] sense is determined by *branch_on*. If *branch*[illegible] *branch_on*=1 the complement of the [illegible] mined based on the next address field NAF as follows.

Table 6 Next Address Field NAF [illegible]

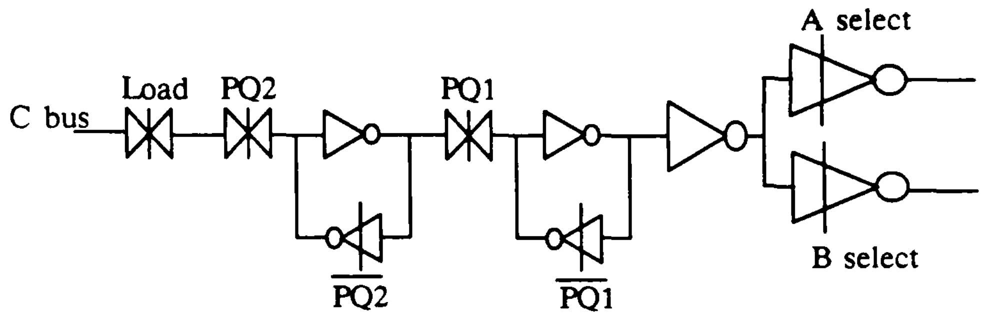
NAF0	NAF1	NAF2	Function	[illegible]
0	0	0	Continue	NA* ← Incremented [illegible]
0	0	1	Return	NA ← Stack Top
0	1	0	Call	NA ← Nxt Addr Field; Stack Top ← [illegible]
0	1	1	Branch	NA ← Nxt Addr Field
1	0	0	Cond External Load	NA ← External Addr
1	0	1	Cond Return	NA ← Stack Top (if cond)
1	1	0	Cond Call	NA ← Nxt Addr Field (if cond); Stack Top ← [illegible] Addr
1	1	1	Cond Branch	NA ← Nxt Addr Field (if condition)

* NA=Next Address

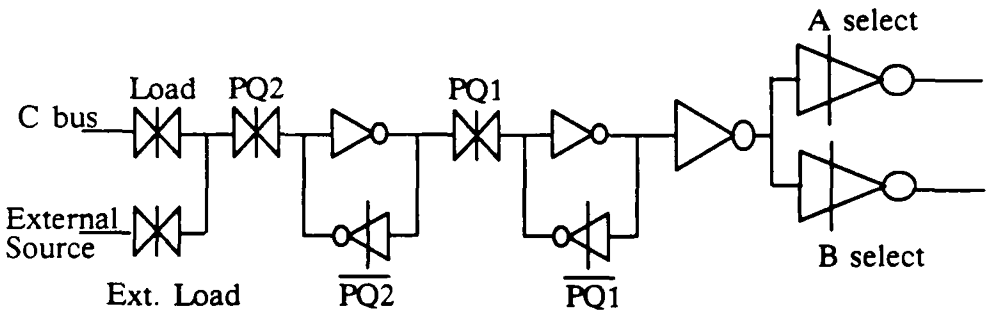
The conditional functions are executed when the selected condition is activated from the condition mux field described earlier.

5.3.7. Data Side. The data side if the PFA controller includes the registers, the arithmetic/logic unit, and any special configurations for data handling including special register interfaces and data data path insertion.

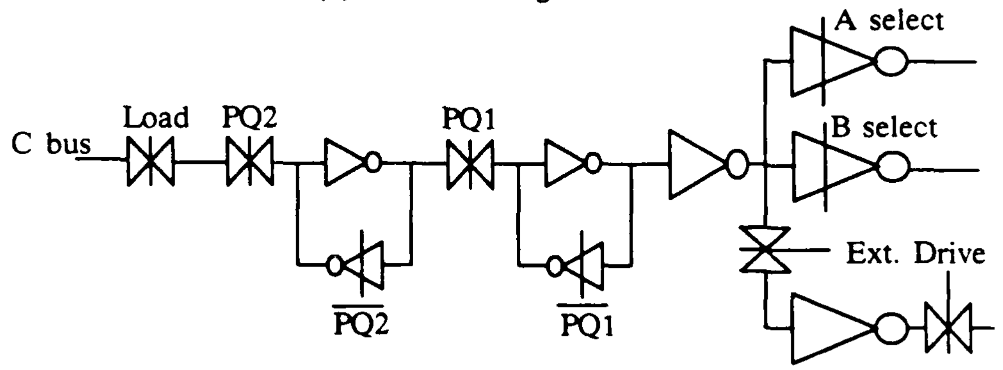
5.3.7.1. Register Cells. The data in the PFA controller is stored mainly in registers. This gives a uniform method of access and increases modularity and regularity in the layout. The basic register cell is shown in Fig. 29(a). The cell is based on the MSFF described earlier. The input is loaded into the first part of the cell when Load is high and the rising edge of PQ2 occurs. The data is latched on the falling edge of PQ2. On the rising edge of PQ1 the data is loaded into the main part of the cell. When the value is to be driven on the A or B bus, the signal lets the value pass through the t-gate inverter to the bus. The three inverters from the PQ1 t-gate to the bus are staged to provide maximum current drive to the bus line. Two other register cells are needed by the PFA controller. The first allows the cell to be driven from external sources other than the bus, and the second allows the cell to drive its value to a destination other than the bus. The externally loadable cell is shown if Fig. 29(b). In the cell, the load t-gate is bypassed and the input is driven in when the other load signal is raised. The driveable cell is shown is Fig. 29(c). In this cell, the value is tapped off right before the t-gate inverters into a t-gate followed by an inverter. In all, four cells are needed to implement all the registers: a basic cell, a loadable cell, a driveable cell, and a loadable and driveable cell. The loadable and driveable cell is just the extra cell circuitry for both options added to a single cell.



(a) Register Cell



(b) Loadable Register Cell



(c) Driveable Register Cell

Figure 29 Register Cells

The A and B busses are precharged for two reasons. First the time to pullup the line can be incorporated into areas of the clocking where no useful computation is taking place (i.e., register selection), and second a decrease in the register cell size. The standard inverter contains a p device for pulling up to logic-1 and an n device for pulling down to logic-0. In using a precharged bus, if the inverter output is a logic-1 no action is taken, however if the output is a logic-0, then the bus line is pulled to logic-0. Therefore by using precharged buses, the p-device pullup is not needed. This significantly reduces the area needed for the register cell. This size directly impacts the register arrays. If a cell is decreased by one lambda in the vertical direction, this equates to decrease of 28 lambda for the entire array, one lamba per register. The removal of the p-device reduced the size of the register cell by approximately 12 lambda in the vertical direction.

5.3.7.2. Registers. This section describes all the registers and how the data is mapped in them. The previous section described the three types of registers cells that are used. Each register is an array of the type of cell needed according to the register's function as described in chapter 3, Section 3.2.5. The *register* cell is an array of 16 basic register cells. This cell is used to implement the error count registers (WD, EC, EU, and PE), the TOUT register (Timeout), and the TEMP register. The *sregister* cell is an array of 16 loadable/driveable register cells used to implement the TSR and PSR registers. The *dregister* is a register using 16 of the driveable register cells and is used to implement the CCR register. Finally, the ELR is made up of 16 loadable register cells. The interfaces with the registers is described in the following section.

The least complicated registers are the error count registers. These registers, including the 9 watchdog counters (WD11-WD33), the 3 parity error counters (PE1-PE3), the 3 error corrected counters (EC1-EC3), and the 3 error uncorrectable counters (EU1-EU3), store a 16-bit count indicating how many errors have occurred for the event the register counts. The WD registers are used not only for fault monitoring, but to determine which WD will become the next active processor should that become necessary.

The CCR and the NCR store the current configuration and the next configuration respectively. Each of the nine MSBs are associated with a WFT processor and indicate, by a logic-1, if the processor is active. Accordingly, a logic-0 indicates that the processor is in watchdog mode. The bit to processor translation is shown in Table 7.

Table 7: Bit to Processor Translation

Processor	WD11	WD12	WD13	WD21	WD22	WD23	WD31	WD32	WD33	not defined						
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The ELR, Error Location Register, indicates where errors, if any, have occurred during the DFT computation just completed. The first nine bits are similar to those for the CCR and NCR, but the following six bits are associated with parity errors and memory uncorrected errors. There is one bit position for each of the three columns for *Parity Error* and *ECCU*. The translation is shown in Table 8.

Table 8: ELR Translation

Error	WD 11	WD 12	WD 13	WD 21	WD 22	WD 23	WD 31	WD 32	WD 33	PE 1	PE 2	PE 3	EU 1	EU 2	EU 3	nd
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The TSR and the PSR store the temporary and permanent scale factors, respectively. The TSR is used to store the scale factors to be driven to the WFT processors and the receive the output scale factors from the processors after computation. Once the scale factors are received after a computation, their values are added to the PSR for scale accumulation. The TSR is then shifted so that the scale factors can be passed to the next stage. Each problem uses five bits to store the accumulated scale factor. Five bit are at worst case when all the scale factors are 7 (i.e., 111 + 111 + 111 +111 = 11100). Once the contents of the TSR are added to the PSR, it too is shifted so that the least significant set of five store the total scaling of the next DFT to complete. The bit translation is shown Table 9.

Table 9: TSR and PSR Translation

Register	Scale Factors (MSB-LSB)															nd
	Problem 1					Problem 2					Problem 3					
TSR	0	0	s2	s1	s0	0	0	s2	s1	s0	0	0	s2	s1	s0	-
PSR	s4	s3	s2	s1	s0	s4	s3	s2	s1	s0	s4	s3	s2	s1	s0	-
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Problem Status registers (PS1, PS2, PS3) are used to store information about each of the three problems in the pipeline. The five MSBs store the accumulated scale factor for the problem, the remaining bits are used for error identification. In this way, the host can determine, in the event of a failure, which problem is bad and where the error occurred. The LSB indicates whether the problem finished. This identifies which processor column did not finish in the invent a timeout failure occurs. The information in the PS*i* registers duplicates the information stored in other registers, but consolidates it by problems for faster identification by the host. The bit translation is shown in Table 10.

Table 10: PSi Translation

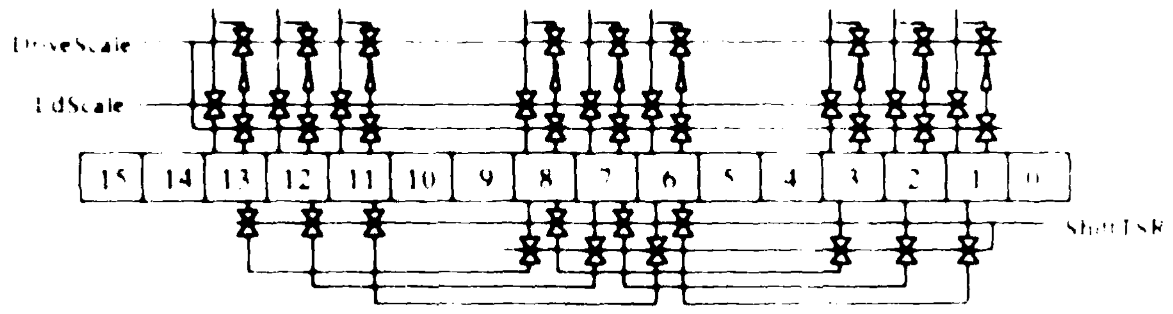
Reg	Scale Factors					Active Error			nd	Parity Error			Memory Error			Done
PS*i*	s4	s3	s2	s1	s0	C1	C2	C3	-	C1	C2	C3	M1	M2	M3	Dn
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

5.3.7.3. Special Register Interfaces. Some registers need special register interfaces to perform either a shift, a load from external sources, a drive to external destinations, or a combination of the three. The special requirements are summarized in Table 11.

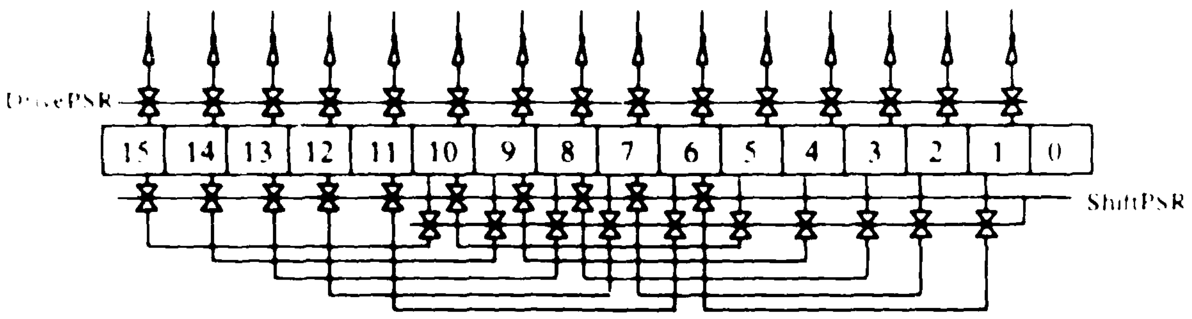
Table 11. Special Register Requirements

Register	Special Requirements
TSR	5-bit, 5-position shiftable Drive State Scale Bus Load from State Scale Bus
PSR	5-bit, 5-position shiftable Drive sets of 5-bits to PS1, PS2, PS3
ELR	Ld 9 MSBs from Error Assignment
PS$_i$	Load 5 MSBs from PSR Load LSB from Done Input Pads
CCR	Drive 9 MSBs to State Scale Bus

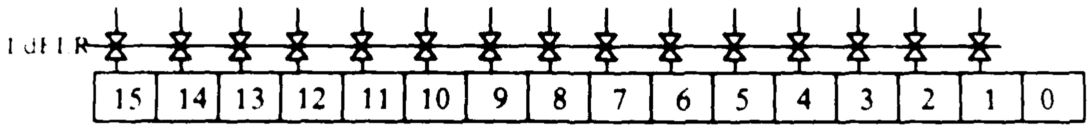
Each of the interfaces is shown in Fig. 30. The shift is implemented as follows: the shift signal (*ShiftPSR* for the PSR and *ShiftTSR* for the TSR) allows the tapped value to pass through the t-gate and directly into the load by-pass in the destination cell. For the load, a t-gate is put before the load by-pass. The signal on the input line is fed directly into the cell when the t-gate is turned on (*LdScale* for TSR, *LdELR* for ELR, *LdScale* for PS*i*). For the drive, a t-gate is attached to the tap point before the drive inverters of the register cell. The t-gate, when on, allows the tapped value to pass through and onto a staged inverter to drive the line. The t-gate prevents unnecessary capacitance when the cell is not driving a line. For the TSR, another t-gate must be placed before the connection to the state/scale bus. This prevents the previous scale values from driving the bus while the state values are being driven to the state/scale bus from the CCR. The state/scale bus and its interfaces are discussed in a later section. For the CCR, this is not needed because of the extra circuitry needed to select the state information.



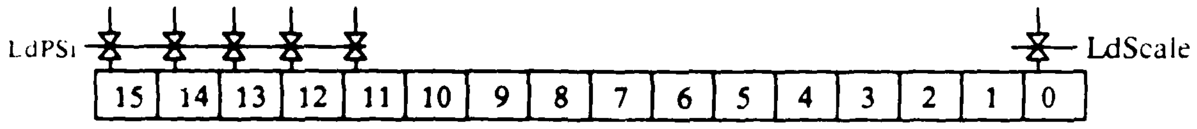
(a) TSR



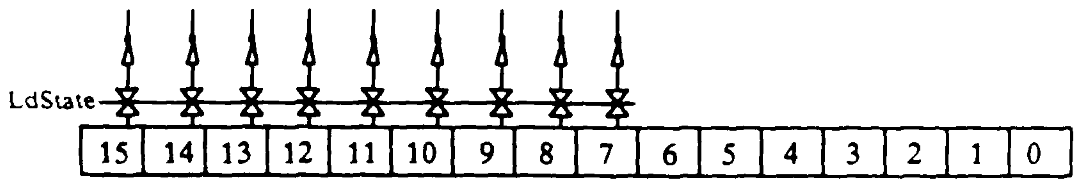
(b) PSR



(c) ELR



(d) PS1,PS2,PS3



(e) CCR

Figure 30 Special Register Interfaces

The extra circuitry is needed for the CCR when it drives the state information onto the state scale bus. This circuitry selects which of the configuration bit is driven. Since it is possible to load the state of three WFT processors at once, one in each column, the circuitry selects which three bits to drive. This circuitry is shown in Fig 31. The LR*i* (*i* = 1 2 3) selects which of the rows to drive and the *LdState* allows the information to drive the state scale bus only when loading the state thus, preventing fighting with the scale information.

5.3.7.4. Register Selection. The register selection cell determines whether the register is selected to drive the A bus, drive the B bus, or be loaded from the C bus. The cell has the same vertical space as a register cell. This allows the register selects to array right along side the registers. Inside the cell are the three selection circuits. The selection circuit is based on ANDing combinations of the five selection bits and then ANDing this result with $\overline{precharge}$. A gate-level description is shown in Fig. 32.

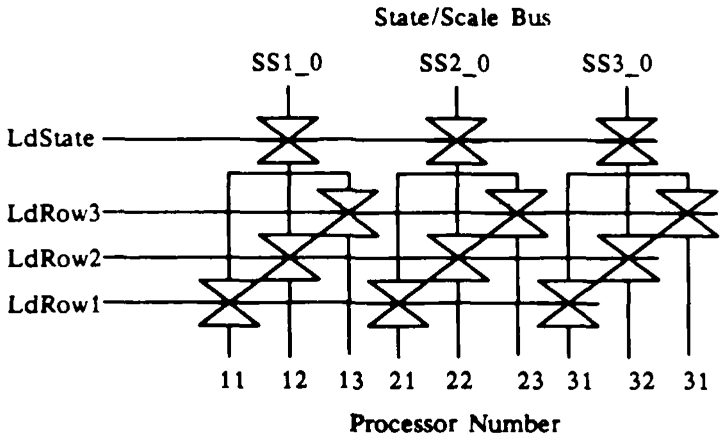


Figure 31 Extra Circuitry for Selecting Configuration Bits

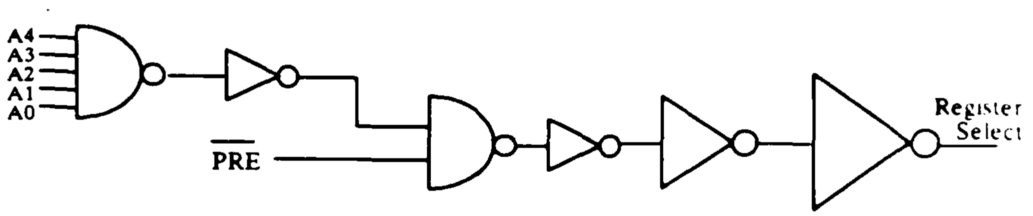
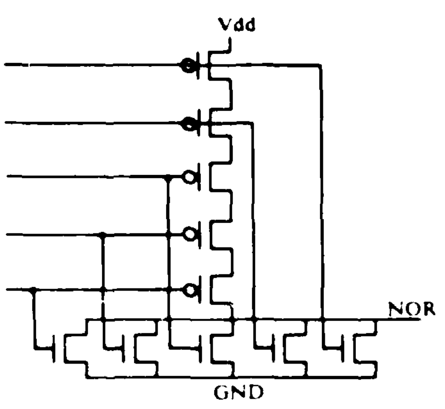


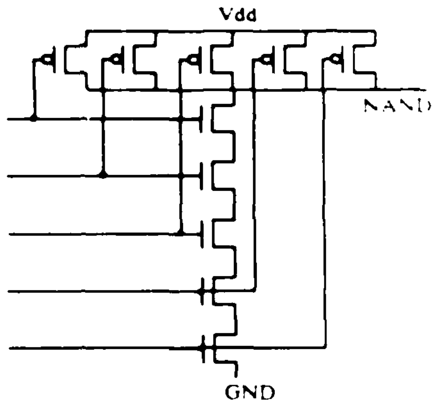
Figure 32 Gate-Level Description of a Register Select

The inputs for each select (A bus drive, B bus drive, C bus load) are fed into a fully CMOS NAND gate. The NAND was chosen over the NOR to make the transition delay more equal in both directions for the least amount of area. If a NOR was used, the delay for a 0-1 transition would be slower than the delay for a 1-0 transition because of the mobility due to p-diffusion since the '1' must pass through 5 p-transistors. This could be offset by increasing the length of the gates on the p-diffusion, but this would greatly increase the area. Instead, the NAND is used where the delay for a '1' is through only one p-transistor. The difference between the two types of implementation is shown in Fig. 33. The output of the NAND gate is inverted to form the AND and this is gated to a NAND along with $\overline{\textit{precharge}}$ followed by an inverter. The purpose of these two gates, making an AND, is to prevent the selection circuit from activating the selection lines during $\overline{\textit{precharge}}$. If either of the drive lines were to be activated during $\overline{\textit{precharge}}$, the A or B bus might not be properly precharged. If the load line was activated during $\overline{\textit{precharge}}$, the precharged lines would be loaded into the register destroying all previous data. The output of the inverted NAND is then staged-up to drive all the selection lines for the register cells.

The VLSI implementation of the cell allows them to be stacked vertically for modularity in the design. The selection bits, along with their complements, run vertically



(a) NOR GATE



(b) NAND GATE

Figure 33 NAND and NOR VLSI Gates

through the array of register selects. To personalize a register select, the select signal or its complement is fed into the NAND gate through the use of a select bit cell. This style of implementation allows easy change from one personalization to another so that only the configuration of the select bit cells is different for each register select. To select register 00111, for example, the personalization would be (sel_0,sel_0,sel_1,sel_1,sel1) where sel_0 selects $\overline{signal}$ for the n-diffusion 'and' and *signal* for the p-diffusion 'or', and sel_1 selects *signal* for the n-diffusion 'and' and the $\overline{signal}$ for the p-diffusion 'or'. A cifplot of a register select cell is shown in Fig. 34 that shows the personalization for 00111.

5.3.8. Data Path Insertion. The XROM must be able to insert a literal onto the C bus. This allows constants to be loaded from the XROM microinstruction to the register. This is accomplished by using an array of t-gates controlled by *LdCbus* which drives the XROM field onto the C bus.

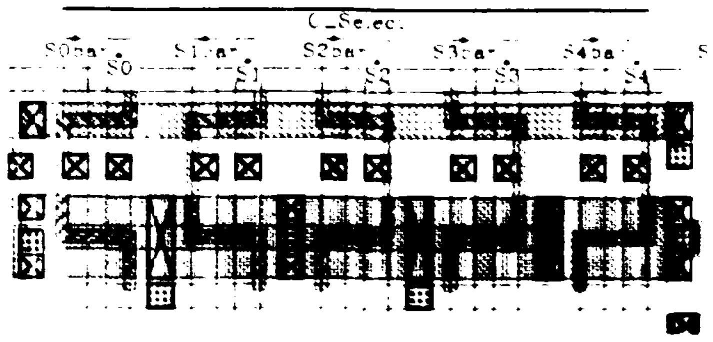
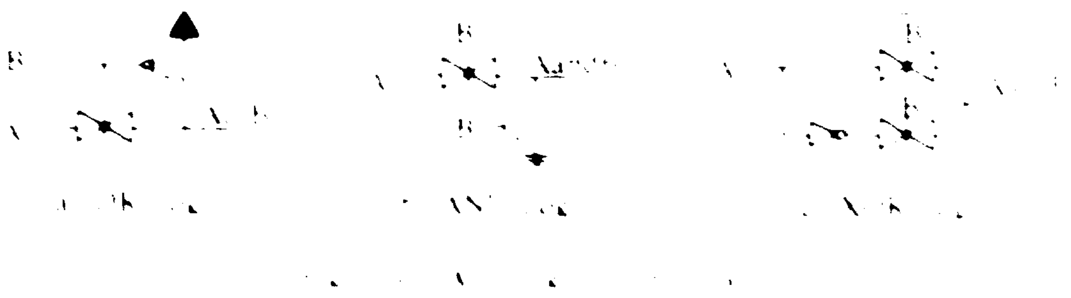


Figure 34   Register Select Cell with 00111 Personalization

5.3.9. Arithmetic/Logic Unit. The Arithmetic Logic Unit (ALU) computes the data manipulations for the controller and was designed by Capt Dave Gallagher [Gal87]. It uses four functional units and one passive unit for each stage. The functional units include logic to implement addition, AND, OR and XOR; the passive unit computes a MOV by passing the data through unchanged. The A bus feeds directly into all five units. The B bus, however, is fed into a selection unit that selects either B, $\overline{B}$, 0, or 1 depending on the function desired. A 5:1 multiplexer selects which unit is output to the C bus. Figure 35 shows the implementations of the three logic functions. Additionally, the ALU will compute four flags: *overflow*, *zero*, *negative*, and *carryout*. The PFA controller uses only the *zero* and the *negative* flags.

5.3.9.1. Integer Adder. The integer adder works with two's complement arithmetic based on the carry-select method [Wes85]. With the carry-select method, the sum and carry out is computed for both a carry-in of zero and a carry-in of one. When



[illegible]

[illegible] this can be [illegible] stages. This is shown in Fig. 36. Now the critical path is the time needed for the carry to propagate through 4 adder cells and 4 additional gates. This reduces the time for the ALU to slightly longer then four alu cells for the entire 16-bit addition. The adder equations are

$Sum = A \ xor \ B \ xor \ C$

$Carry = AB + AC + BC$

However the carry sum may also be represented as

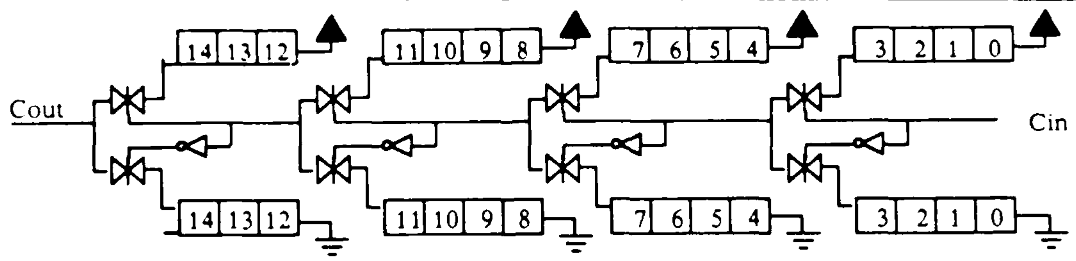


Figure 36 Carry Select Adder Blocking

$Carry = (A\ xnor\ B)A + (A\ xor\ B)C$

This implementation for the carry allows the use of the *A xor B* signal generated for the sum to be used reducing the amount of circuitry per cell. The implementation is shown in Fig. 37.

5.3.9.2. Functions. The functions computed by the ALU as well as the signals needed to generate are described as follows:

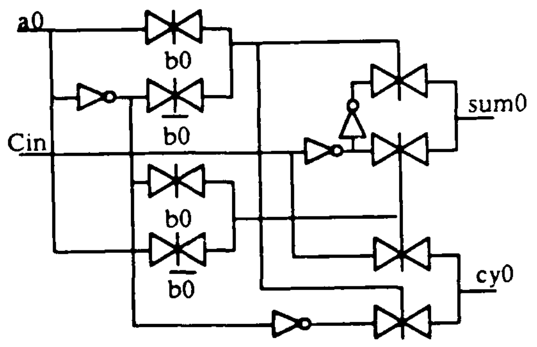
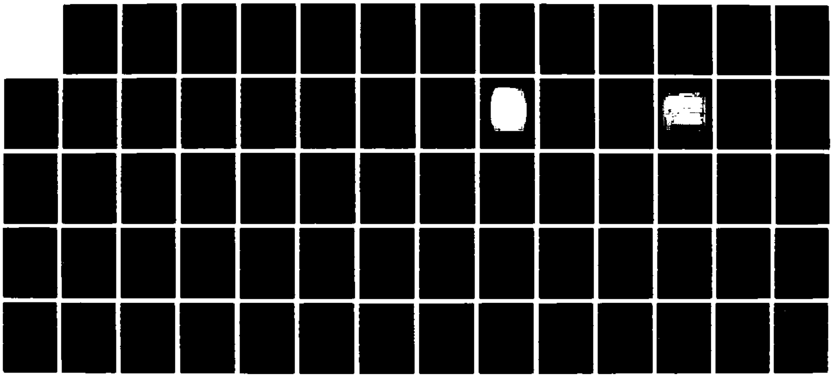


Figure 37 ALU Adder Cell

AD-A190 403 DESIGN AND IMPLEMENTATION OF VLSI PRIME FACTOR ALGORITHM PROCESSOR(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 2/2

UNCLASSIFIED R S HAUSER DEC 87 AFIT/GCE/ENG/87D-5 F/G 12/6 NL



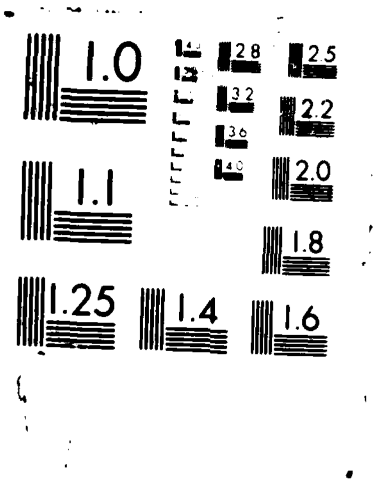

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

Table 12: ALU Control and Functions

Control Signals				Function	Operation	Implementation	Carry In	Flags
a3	a2	a1	a0					
0	0	0	0	nop	-	-	-	-
0	0	0	1	INV	$\overline{A}$	C ← A xor 1	-	Z
0	0	1	0	AND	A and B	C ← A and B	-	Z
0	0	1	1	XOR	A xor B	C ← A xor B	-	Z
0	1	0	0	OR	A or B	C ← A or B	-	Z
0	1	0	1	MOVE	A	C ← A	-	-
0	1	1	0	Set Carry	Cin = 1	Cin ← 1	Cin ←1	Cout
0	1	1	1	Reset Carry	Cin = 0	Cin = 0	Cin ← 0	Cout
1	0	0	0	INC	A + 1	C ← A + 1	1	all
1	0	1	1	DEC	A - 1	C ← A + 1	0	all
1	0	1	0	ADC	A + B + Cin	C ← A + B + Cin	Cin	all
1	0	1	1	ADD	A + B	C ← A + B	0	all
1	1	0	0	Not Defined	-	-	-	-
1	1	0	1	SUB	A - B	C ← A + $\overline{B}$ + 1	1	all
1	1	1	0	SUBB	A - B - Bin	C ← A + $\overline{B}$ + $\overline{Cin}$	$\overline{Cin}$	all
1	1	1	1	CMP	A - B	C ← A + $\overline{B}$ + 1	1	all

5.3.10. Host Control Interface. This section describes the host interface with the PFA controller during the WAITGO loop. The WAITGO loop in the microcode is used so the host can examine the PFA registers and change, if necessary. For the host to examine any register it must be able to select a register to drive onto the data bus. For loading, it must be able to put the input data onto the data bus and select the register to load. The signal *HOSTCONTROL* determines whether the inputs to the register selects come from the XROM or from the host and the signal *LOADSTUFF* determines the data flow direction. This allows the same set of host register selection signals for both reading and writing. It also allows the data pads to be used for reading and loading of data. Thus, the number of pads is 21 (16 data, 5 register selection) instead of 42. Figure 38 shows how the determination is made for the register selection. The host register

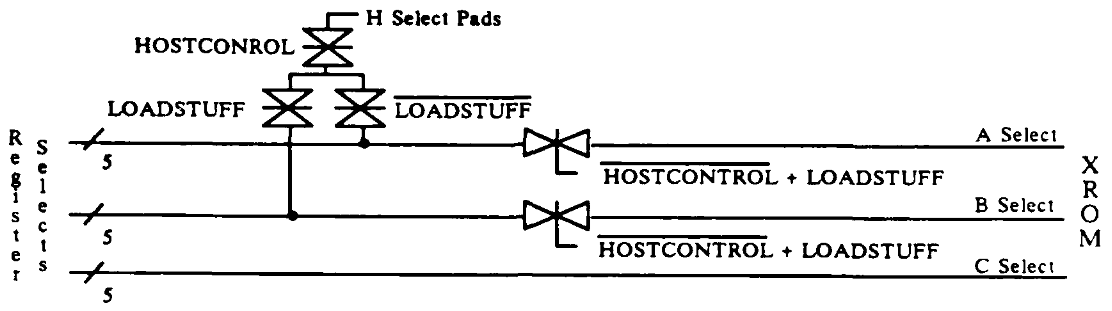

H Select Pads
HOSTCONROL
LOADSTUFF
$\overline{\text{LOADSTUFF}}$
Register
Selects
5
5
5
A Select
$\overline{\text{HOSTCONTROL}}$ + LOADSTUFF
B Select
$\overline{\text{HOSTCONTROL}}$ + LOADSTUFF
C Select
XROM

Figure 38 Source Determination for Register Selection

selection source for the A bus will be the XROM for $\overline{HOSTCONTROL}+LOADSTUFF$ and the source for the C bus selection will be the XROM for $\overline{HOSTCONTROL}+\overline{LOADSTUFF}$ otherwise, it will be the host. This ensures that when the host is in control, but not using one the register selects, the inputs will be set to all zeros from the XROM preventing nondeterministic results from floating lines. Figure 39 shows the source and destination determination for the data busses. For the data bus, data will flow from the pads to the C bus for $HOSTCONTROL \cdot LOADSTUFF$ and from

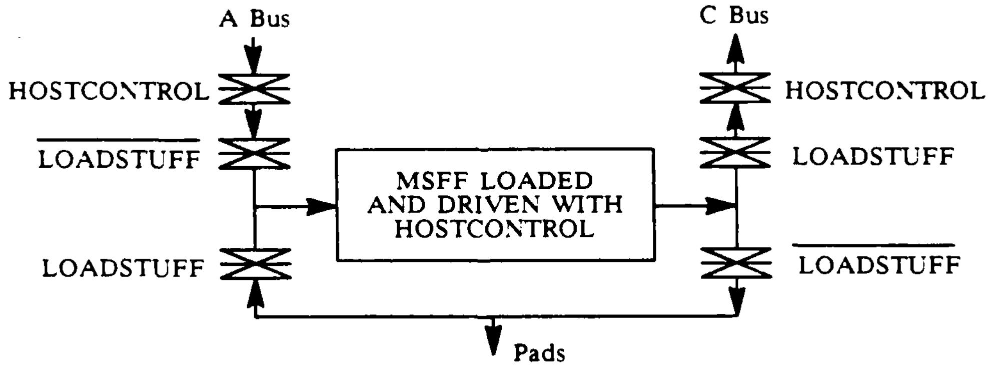

A Bus
C Bus
HOSTCONTROL
HOSTCONTROL
$\overline{\text{LOADSTUFF}}$
LOADSTUFF
MSFF LOADED AND DRIVEN WITH HOSTCONTROL
LOADSTUFF
$\overline{\text{LOADSTUFF}}$
Pads

Figure 39 Source and Destination Determination for Data Busses

the A bus to the pads for $HOSTCONTROL \cdot \overline{LOADSTUFF}$. Using the *LOADSTUFF* signal prevents the A bus and the C bus from being shorted together. The signals and their effects are summarized in the following:

Table 13: Register Selection and Bus Determination

HOSTCONTROL	*LOADSTUFF*	A bus select	A bus destination	C bus select	C bus source
0	0	XROM	internal	XROM	internal
0	1	HOST	internal	XROM	internal
1	0	XROM	PADS	XROM	internal
1	1	XROM	internal	HOST	PADS

5.3.11. Periphery. The peripheral circuitry contains that which does not fit into either of the two previous major sections. This circuitry includes the interfaces to the WFTs, the state/scale bus, the voting circuitry, the load circuitry, the scale factor handling, the DFT size handling, the 4 Done signal generation, the ERROR signal generation and associted signals, and the toggle flip-flop.

5.3.11.1. State/Scale Bus. The state/scale bus is used to transmit data to the WFTs about state and scale information as well as receive the new scale information. The state/scale bus consists of nine lines with a group of three representing the state/scale bus for a particular WFT pipeline column. The data flow for the bus is shown on Fig. 40. The three main signals that control the data flow are *LdState*, *LdScale*, and *DriveScale*. The scale information flows into or out of the TSR. *DriveScale* controls the t-gates above the TSR register that allow the register to drive the signals, and *LdScale* controls the t-gates above the register to load in values by-passing the C bus. The state information is generated from two places; the size information comes from the size storage cells and the WFT processor watchdog configuration information comes

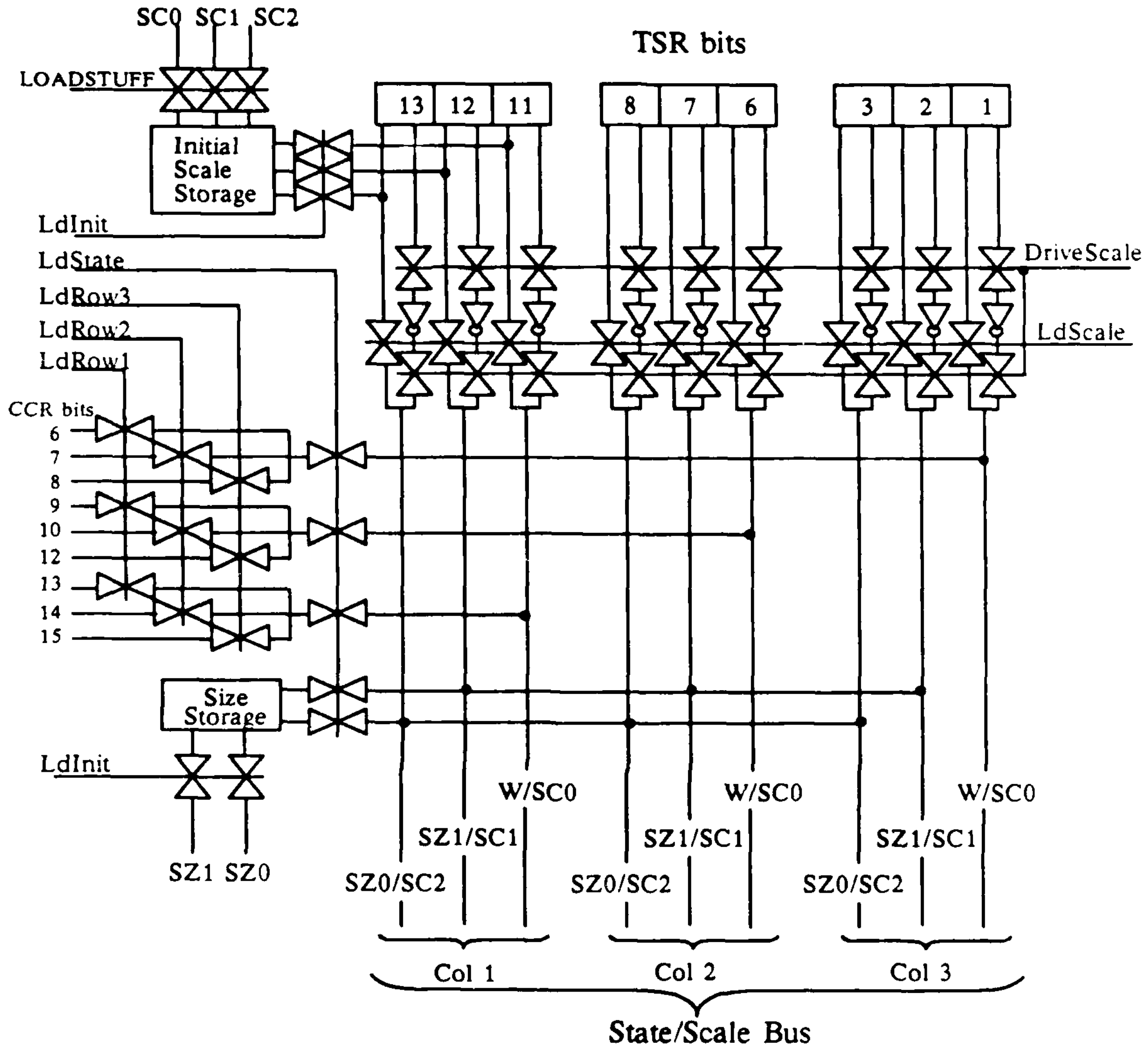


Figure 40 Data Flow for State/Scale Bus

from the CCR. A logic-1 tells the WFT to be active, and a logic-0 tells it to be a watchdog. Since the columns have independent state/scale busses, they can be loaded concurrently. The *LR*1, *LR*2, and *LR*3 signals from the XROM select which bit from the CCR to drive onto the state/scale bus. These signals are further gated by *LdState* so as not to interfere with loading or receiving of the scale information. The translation of the state/scale bus to the WFT interface for each column is as follows:

Table 14: PFA-WFT Interface Translation

PFA	ST1	ST2	ST3
WFT	WD/SC0	SIZE1/SC1	SIZE2/SC2

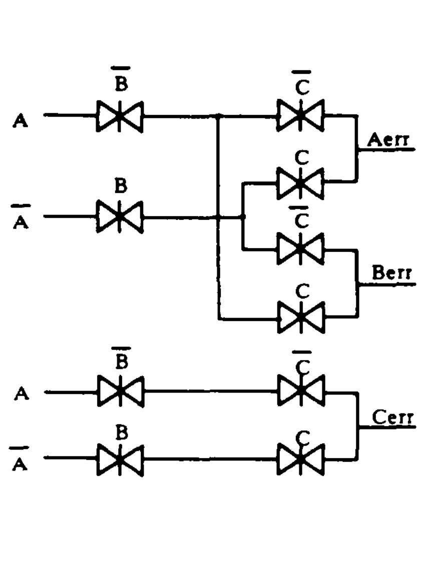
5.3.11.2. Voting Circuitry. The voting cell is used to assign an error to one of three inputs according to the voting strategy described in chapter 3. The voting cell has several components. The first component is the cell called *3vote*. This cell actually implements the voting strategy. From Karnaugh maps with inputs i,j,k, the following logic equation result for each of the three input to determine error assignment:

$$error_i = i\cdot\overline{j}\cdot\overline{k} + \overline{i}\cdot j\cdot k,$$

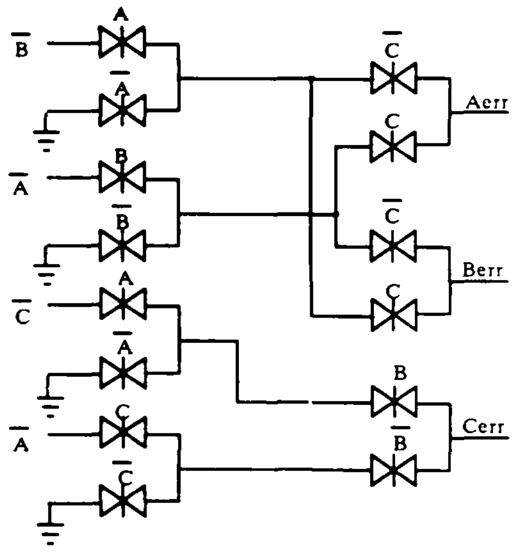
$$error_j = \overline{i}\cdot j\cdot\overline{k} + i\cdot\overline{j}\cdot k,$$

$$error_k = i\cdot\overline{j}\cdot k + i\cdot j\cdot\overline{k}.$$

One particular implementation is shown in Fig. 41(a). The problem with this implementation is uneven capacitive loading. The capacitance loading on the c input is 1.5 times that of b and much larger than a. Also the a signal must travel through two t-gates, whereas b ar do not. To reduce this imbalance, thus increasing speed, the solution shown in Fig. 41(b) was chosen. This distributes the load while still maintaining the mutual exclusion needed for the multiplexers. It does, however, increase the number of t-gates needed because of the need to prohibit floating nodes. The cell is repeated three times, one for each set of three WFT processors. The next component in the voting cell is the column error generator. This cell looks at all of the nine error lines output by the 3vote cells and determines which columns, if any, contain errors. The column generator unit is made up of three 3-input OR gates. The OR gates are made by a 3-input NOR gate followed by an inverter. The column error signals go to the branch circuitry. The



(a) Voting Circuit



(b) Modified Voting Circuit

Figure 41 Voting Implementation

column error signals are used to reduce the number of lines of microcode by narrowing the location of the error to a column. The third component is a set of t-gates, which gates the signals from the 3vote cells to the stageup cell using the *LdELR* signal. The stageup cell inverts each of the nine signals four times with increasing gate size in a ratio of 1:2:4:8 to stage up the signals for greater current drive due to the load on these nine lines. The stageup is necessary because the nine error lines go to the *ERROR?* cell, (described later) to the ELR, and the branch logic.

5.3.11.3. WFT Processor Loads. The load signals indicate to a WFT processor that state information is being loaded. This information is loaded when configuring the pipeline. When the *LOAD* line to a WFT processor is high, the information is loaded into flip-flops that store the information until the *LOAD* line becomes high again. Each WFT processor must have its own *LOAD* signal since each must be configured individually. However, each of the three columns of WFT processors may be done at the same time since each has its own 3-bit state/scale bus. Therefore, loading is done one logical row at a time for the three rows. The implementation is shown if Fig. 42. Since each row is loaded at a time only three lines are needed, one for each row.

5.3.11.4. Scale Factors. The initial scale factors for the WFT16 processors must be given by the host. The host inputs these while *LOADSTUFF* is high. Therefore, every time the host raises *LOADSTUFF* when changing the register values, it must also ensure the correct scale factors. The storage for the scale factors are modified register cells. They are modified because the lines that the cells drive are not precharged as in the register array. The modification is to put in the p-device that was not needed in the basic register cell. The input to the cells are gated with *LOADSTUFF* so that when *LOADSTUFF* is high, the signals on the input scale pads are loaded into the storage cells. The scale factors are loaded into the TSR when *LdInit* is raised. This occurs before the TSR drives the scale factors onto the state/scale bus to be output to the WFT processor columns.

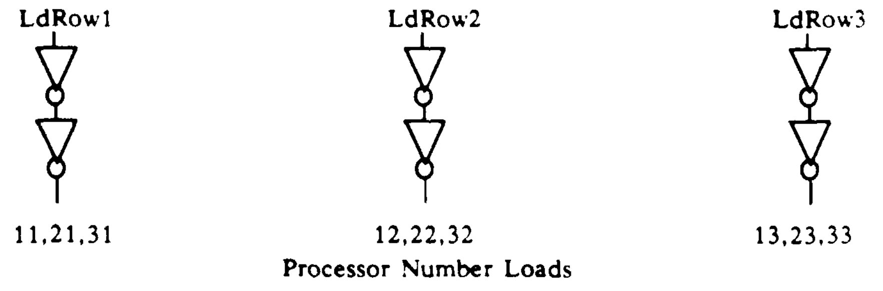


Figure 42 WFT Processor Load Determination

Table 15: WFT DFT Size Determination

SIZE0	SIZE2	Size Translation
0	0	4096
0	1	768
1	0	276
1	1	16

5.3.11.6. Done?. The DONE signal generated by this cell is used as a condition input for branching until the input host and the three WFTs have completed their operations. Each of the input DONE signals are fed into a NAND gate, the output of this gate is inverted three times to produce the desired AND product and stage up the signal to travel across the chip to the branch logic.

5.3.11.7. Error?. The ERROR? cell determines if an error occurred during a DFT computation and also generates signals isolating the error to a set of input bits. Figure 44 show the gate-level description of the circuit. The three sets of inputs come

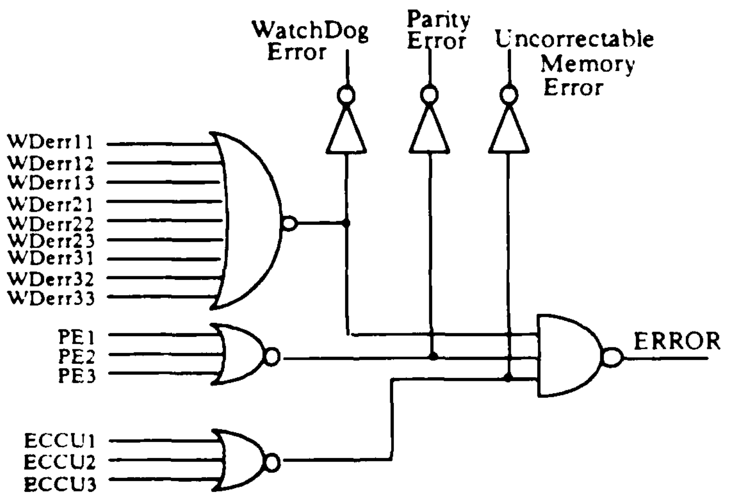


Figure 44 Gate-Level Description of ERROR? Cell

from the Voting Circuitry, the Parity Error pads, and the Error Code Uncorrected pads. Any one of these bits being high invalidates the data where the error occurred. The logic 'goal' is to implement an OR of all the signals. At the same time, the area of the error needs to be identified, therefore a simple 15-input OR gate cannot be used. Instead a NOR gate is used for each set of inputs. The inverted NOR, making an OR, is used to generate to error flags for each input set. The outputs of the NOR are fed into a NAND gate to produce the ERROR flag. By using the boolean equations, the use of an OR gate in CMOS would have produced two levels of gates since an OR gate is produced by inverting the output of a NOR gate. The boolean transformation from the input to the ERROR signal can be seen as follows:

each of the NORgates produces:

$$\overline{WDerr} = \overline{WD11err+WD12err+\cdots+WD32err+WD33err}$$

$$\overline{PEerr} = \overline{PE1err+PE2err+PE3err}$$

$$\overline{EUerr} = \overline{ECCU1err+ECCU2err+ECCU3err}$$

the NAND gate produces:

$$ERROR = \overline{\overline{WDerr}\cdot\overline{PEerr}\cdot\overline{EUerr}}$$

by deMorgan's Law$(\overline{\overline{a}\cdot\overline{b}} = a+b)$

so,

$$ERROR = \overline{\overline{WDerr}}+\overline{\overline{PEerr}}+\overline{\overline{EUerr}} = WDerr+PEerr+EUerr.$$

Thus producing the same logic output but using one less gate. The output ERROR is used as a condition flag for calling the error routine and the three other outputs are used to narrow down the error location to save time in the error routine.

5.3.11.8. Toggle F/F. The toggle flip/flop is used to generate the *FLIP* for the memories. The XROM simply indicates to the circuitry that the signal needs to be changed. This way, the microcode need not test what the value was before and then change flip it. The circuit needed to implement this must toggle its output every time the input is pulsed from the XROM This kind of flip-flop is a toggle flip-flop (TF/F). The TF/F chosen for implementation is described in Glasser and Dobberpohl [Gla85]. This circuit, shown in Fig. 45, operates with a 2-phase clock and a reset. The reset signal comes from the global reset signal for the PFA controller. The reset signal is needed to put the TF/F into a deterministic starting state, otherwise the feedback loops are undefined. The input is the *FLIP* bit from the XROM, this bit is raised for one clock cycle before each DFT computation is started. The *FLIP* determines which side of the memory is written to (read from).

5.3.11.9. Column Done Storing. If a timeout occurs, it will be necessary to indicate which of the column failed. This is done by loading the DONE signal from the three WFT columns into the PS*i* registers at the same time the output scale factors are loaded into the TSR. This way, the PS*i* will contain information as to whether the

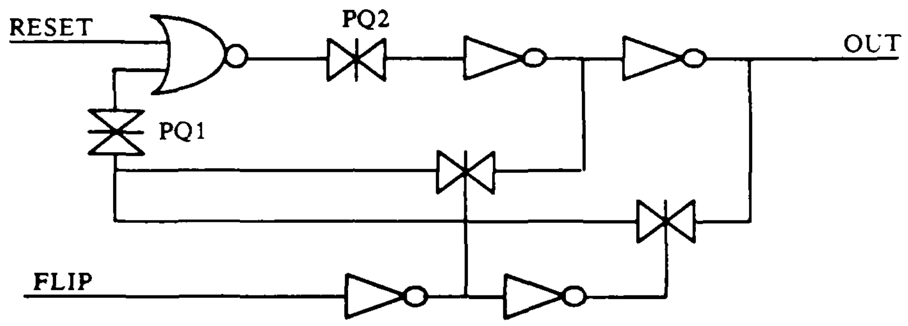


Figure 45 Toggle Flip-Flop Gate-Level Representation

5.3.11.5. Size. The DFT size bits are handled in an manner similar to the initial scale factor bits. When *LOADSTUFF* is high, the input at the two SIZE pads are loaded into two register cells. The DFT size bits are driven to the state scale bus when the state is being loaded into each WFT processor with the *LdState* signal. Each WFT processor, 15, 16 and 17 each receive the same DFT size. The size determines how many words the WFT processor will use in the DFT computation. The interface to the state/scale bus is shown in Fig. 43. The reason for gating each of the size bits through *LdState* even though this same signal drives the output of the storage cells is so that the state/scale lines will not be shorted when loading scale information. The following describes the meaning of the size bits for the WFT16:

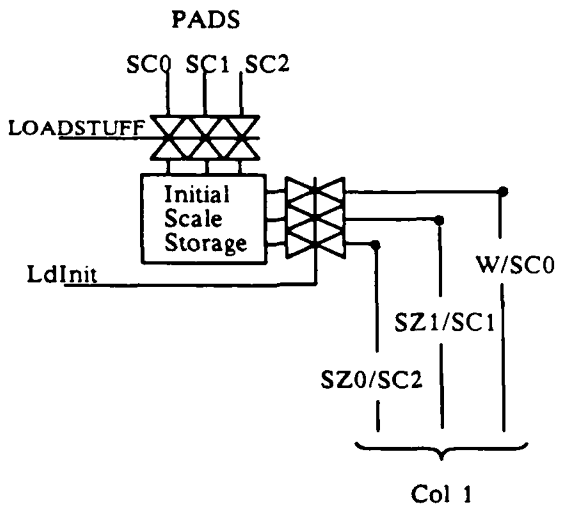


Figure 43 Size Interface to State/Scale Bus

problem is still valid as well as which column did not finish. The WFT *done*'s are directly loaded into the PS*i* registers with the *LdScale* signal from the XROM.

5.3.12. Microcode Development.

The microcode is developed along side the VLSI design. Design tradeoffs are made between the microcode and the hardware. Routines in the microcode can be made simpler by increasing the hardware complexity. Tradeoffs occur when the complexity of the hardware increases more rapidly than the microcode simplification.

The first step in the microcode development defines the microinstruction fields. The PFA microinstruction contains 10 fields as shown in Table 16.

Table 16: Microword Format

BR_SEL	ALU	LOAD_FD	ABUS	BBUS	CBUS	SPEC_FUNCT	WFTOP	PFADONE	NXT_ADDR
0-8	9-12	13	14-18	19-23	24-28	29-41	42	43	44-59

The BR_SEL (branch selection) determines the branching conditions and selections. The ALU field determines the operation of the ALU. The LOAD_FD (load field) field determines whether the NXT_ADDR field is inserted into the datapath or not. The ABUS and BBUS fields determine which register is driven onto the A bus and the B bus respectively. The CBUS field determines which register is loaded from the Cbus. The SPEC_FUNCT (special functions) field is a horizontally encoded field to control certain operations on the chip. The WFTOP field is used to start the WFT processors. The PFADONE fields is used to signal the Host that a DFT computation has been completed. The WFTOP and PFADONE fields were not included as part of the SPEC_FUNCT field to increase code readability and emphasize their importance. NXT_ADDR (next address) is the final field in the word. This field is used to both specify the branch location or a literal to be placed on the C bus.

The BR_SEL field is connected to the control sequencer and broken down into three subfields as shown in Table 17.

Table 17: BR_SEL Field

BR_SEL Field								
CMS					BR_ON	NAF		
0	1	2	3	4	5	6	7	8

The CMS (conditional mux selector) subfield, enumerated in earlier in Table 5, selects the condition bit for a branch. The BR_ON field selects the condition bit when this field is a 1, and the complement when it is a 0. This way, the microcode can branch on the presense or absence of the condition. The NAF (next address field) subfield selects the source for the next address. The NAF field is enumerated in Table 6.

The ALU field consists of the four signals a3-a0 that are connected to the ALU. The ALU field is enumerated in Table 18 and shown below.

Table 18: ALU Field

ALU Field			
a3	a2	a1	a0
9	10	11	12

The LOAD_FD is bit 13 in the microinstruction. When this bit is a 1, the contents of the NXT_ADDR field is driven onto the C bus. From the C bus, it can be loaded into any register. Accordingly, when the bit is 0, the field does not affect the C bus.

The ABUS, BBUS, and CBUS select one of the 28 registers to be driven or loaded. Five bits are needed to select the 28 registers. The format of these fields is shown in Table 19.

Table 19: Bus Fields

ABUS					BBUS					CBUS				
0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

The select of a register is a combination of the 5 address bits. Each fields uses the same decoding scheme enumerated in Table 20.

Table 20: Register Select Translation

Select 4 3 2 1 0	Register	Select 4 3 2 1 0	Register
00000	unused	10000	WD31
00001	EC1	10001	WD32
00010	EC2	10010	WD33
00011	EC3	10011	TSR
00100	EU1	10100	PSR
00101	EU2	10101	unused
00110	EU3	10110	PS1
00111	PE1	10111	PS2
01000	PE2	11000	PS3
01001	PE3	11001	ELR
01010	WD11	11010	NCR
01011	WD12	11011	CCR
01100	WD13	11100	TEMP
01101	WD21	11101	TOUT
01110	WD22	11110	unused
01111	WD23	11111	unused

The SPEC_FUNCT (special functions) field is the seventh on the microinstruction This field contains bits to control certain operations on the chip and in shown in Table 21.

Table 21: SPEC_FUNCT Field

SPEC_FUNCT Field	
Bit	Control Signal
29	Flip
30	LdInit
31	LdScale
32	LdPSi
33	ShiftTSR
34	ShiftPSR
35	LdELR
36	HOSTCONTROL
37	LdR1
38	LdR2
39	LdR3
40	LdState
41	DriveScale

The WFTOP field is bit 42. When this bit is high the WFT processors are allowed to compute. Bit 43 is the PFADONE field. This bit is raised when the PFA has completed a DFT computation.

Bits 44-59 specify the NXT_ADDR (next address) field. Sixteen bits are needed because this field interfaces with the 16-bit C bus. Bits 51-59 are connected to the source multiplexer in the control sequencer and represents the branch address.

The microcode word format is summarized in Appendix G. The second step in microcode development takes the algorithm flow charts developed in chapter 3 and translates them into microcode routines. The first task according to Fig. 3-3 is initialization. This is done by loading all the registers with 0s as shown below:

```
RESET:      LOAD REG REG  TEMP  #0000000000000000;
            LOAD REG REG  WD11  #0000000000000000;
            LOAD REG REG  WD12  #0000000000000000;
            LOAD REG REG  WD13  #0000000000000000;
            LOAD REG REG  WD21  #0000000000000000;
            LOAD REG REG  WD22  #0000000000000000;
            LOAD REG REG  WD23  #0000000000000000;
```

When consecutive microword fields use the same field from the translation file, the defaults must be explicitly defined for the fields before the one being used. In this example, the third register field was used but not the first two. Therefore, the defaults for the first two must be set.

The next step is to wait for the Host to give the PFA the *PFAOP* signal. During this time, the Host is allowed to examine and alter the register contents. The loop for this is shown below.

```
WAITGO:     JnOP  HostCntl WAITGO;
            HostCntl;
```

This loop illustrates several important points about the microcode and the PFA controller. First, the use of label is shown. WAITGO is the name of the loop and the line number is substituted for WAITGO in the microinstruction. Second, the use of default values makes the microcode more readable. If default values were not permitted the same two lines would be coded as shown below.

```
WAITGO      JnOP ALU LOAD_FD REG REG REG HostCntl WFTOP PFADONE WAITGO;
            BR_SEL ALU LOAD_FD REG REG REG HostCntl WFTOP PFADONE NXT_ADDR;
```

Third, the pipeline requires the instruction following a branch to cause no undesirable side effects. Since the fetching of the XROM microinstruction is pipelined, there is a one instruction delay before a branch occurs. In this instance, a nop is not needed because HostCntl should be high until the branch occurs.

After *PFAoperate* is received the controller compares the CCR and the NCR. If these are different, a new configuration was requested. The controller then moves the contents of the NCR to the CCR and loads the configuration data via the state/scale bus to the WFT processors. The controller then toggles the *LEFT* signal to the WFT processors, drives the scale factors from the TSR to the WFT processors via the state/scale bus, and raises *WFTOP*.

The controller now waits for all four *DONE* signals from the input host and the three WFT processors. The controller will only wait a predetermined time for the processors to finish. This time is stored in the TOUT register. While the controller waits, it increments the TEMP register and compares it to the TOUT register. When the two register are equal, a time out has occurred. The host will detect this in the PSi registers because the *done* bit will not be set. The code implementing this is shown below.

```
WAITDONE:   INC TEMP REG TEMP WFTop;
            CMP TEMP TOUT WFTop;
            JZ WFTop SCALE;
            WFTop;
            Jn4DN WFTop WAITDONE;
            WFTop;
```

When the controller exits the WAITDONE loop, it latches the scale factors from the processors into the TSR, drops the *WFTOP* signal, and checks for errors. The reasons for many of the condition inputs will now be explained.

To save time in the error routine, the controller isolates the error(s) to a specific set of inputs. A sample of the error routine below illustrates this.

```
            JnWD  ErrPE;
            OR TEMP PS3 PS3;
WDREGS:     JnECo1 INCOL2;
            nop;
WD_11:      Jn11 WD_12;
            nop;
            INC WD11 REG WD11;
WD_12:      Jn12 WD_13;
            nop;
            INC WD12 REG WD12;
WD_13:      Jn13 INCOL2;
            nop;
            INC WD13 REG WD13;
```

```
INCOL2:     JnECo2 INCOL3;
            .
            .
            .
ErrPE:      JnPE ECCU;
            nop;
```

The controller first checks if the error occurred in the watchdog processors. If the error was not in the watchdogs then it skips to the segment for Parity Error. Within the segment for watchdog errors, the controller first narrows the error to a column and then to a specific processor. This same approach is used for all the error signals.

The controller then determines if an active processor faulted by comparing the CCR and the ELR. If an active was at fault, the pipeline must be reconfigured. The controller looks at the watchdog error counts for the two current watchdogs and assigns the one with the lowest error count to be the next active. Once the controller has set up the new configuration in the NCR, it will be different than the CCR and RECONFIGURE will be called when the controller starts the next problem. The controller now sends the *PFA-DONE* signal to the host and waits for *PFAOP*

5.3.13. PFA Controller Summary.

This chapter has described the VLSI design for the memory and the VLSI design and microcode development for the PFA controller. The PFA controller consists of several major functional units and interfaces. The high level interaction of these units can be better appreciated in Fig. 46. This figure shows the major parts of the controller and their approximate location on the chip.

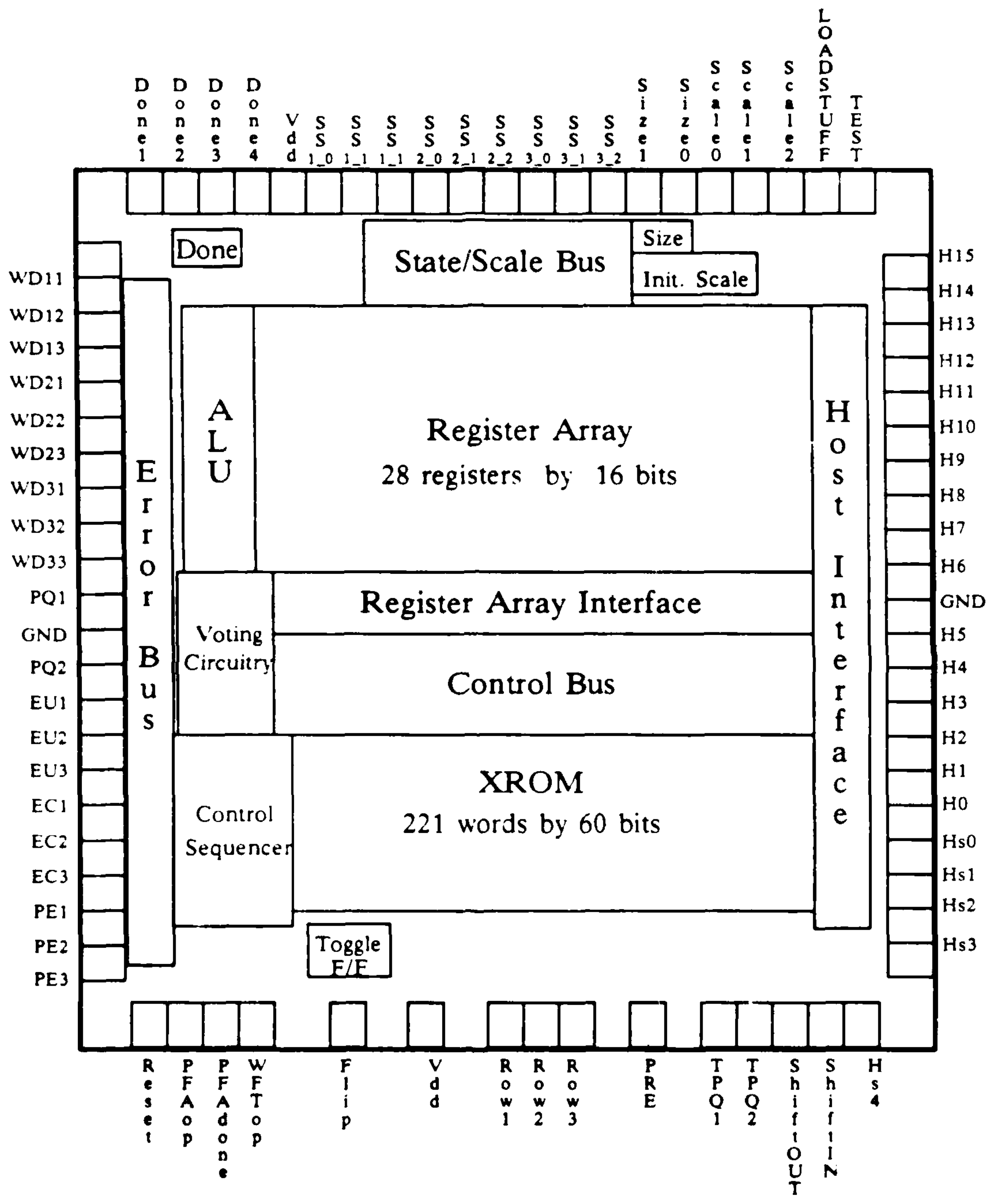


Figure 46 PFA Controller Floorplan

CHAPTER 6

Results

6.1. Results

This thesis effort has produced a prototype memory chip, the layout for the full memory chip and the PFA contoller, and a generic microcode assembler.

6.1.1. Memory Chips. A prototype memory chip was design and fabricated. It was designed to test the address decoders, the memory cell, the sense amplifiers, and the bitline logic. The chip was fabricated in 28-pin package using 3 micron CMOS process through MOSIS (MOS Implementation Service). A photomicrograph of the fabricated chip is shown in Fig. 47. The chip contains 32 words with 10 bits each. One of the bits in each word was used for the dummy bitline and one other bit was unused.

A larger memory chip was also designed and submitted for fabrication. This chip was designed to store 272 words by 24 bits. This is the size memory needed for the a prototype PFA pipeline using a WFT16 and a WFT17 processor. The larger chip also contains all the circuitry to support the error correction and detection. A cifplot of this chip is shown in Fig.48. The chip is 7900 microns by 9200 microns

6.1.2. PFA Chip. A fully functional PFA controller was designed and submitted for fabrication. The chip will be 7900 microns by 9200 micron and sits in an 84-pin package. The chip contains over 23900 transistors. A cifplot of the chip is shown in Fig. 49. Prior to submission, the chip was fully simulated using Esim and the design was verified.

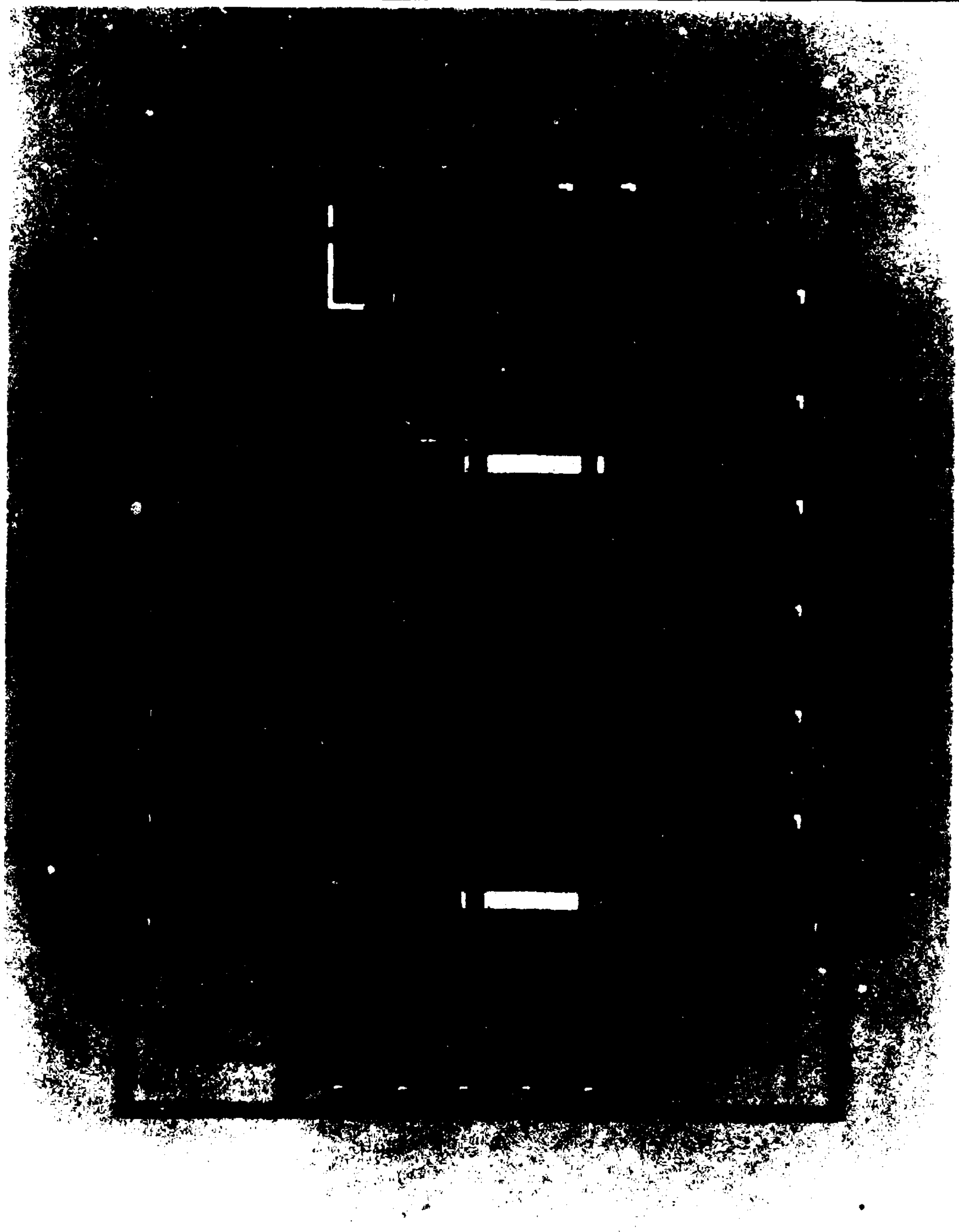


Figure 47 Prototype Memory Photomicrograph

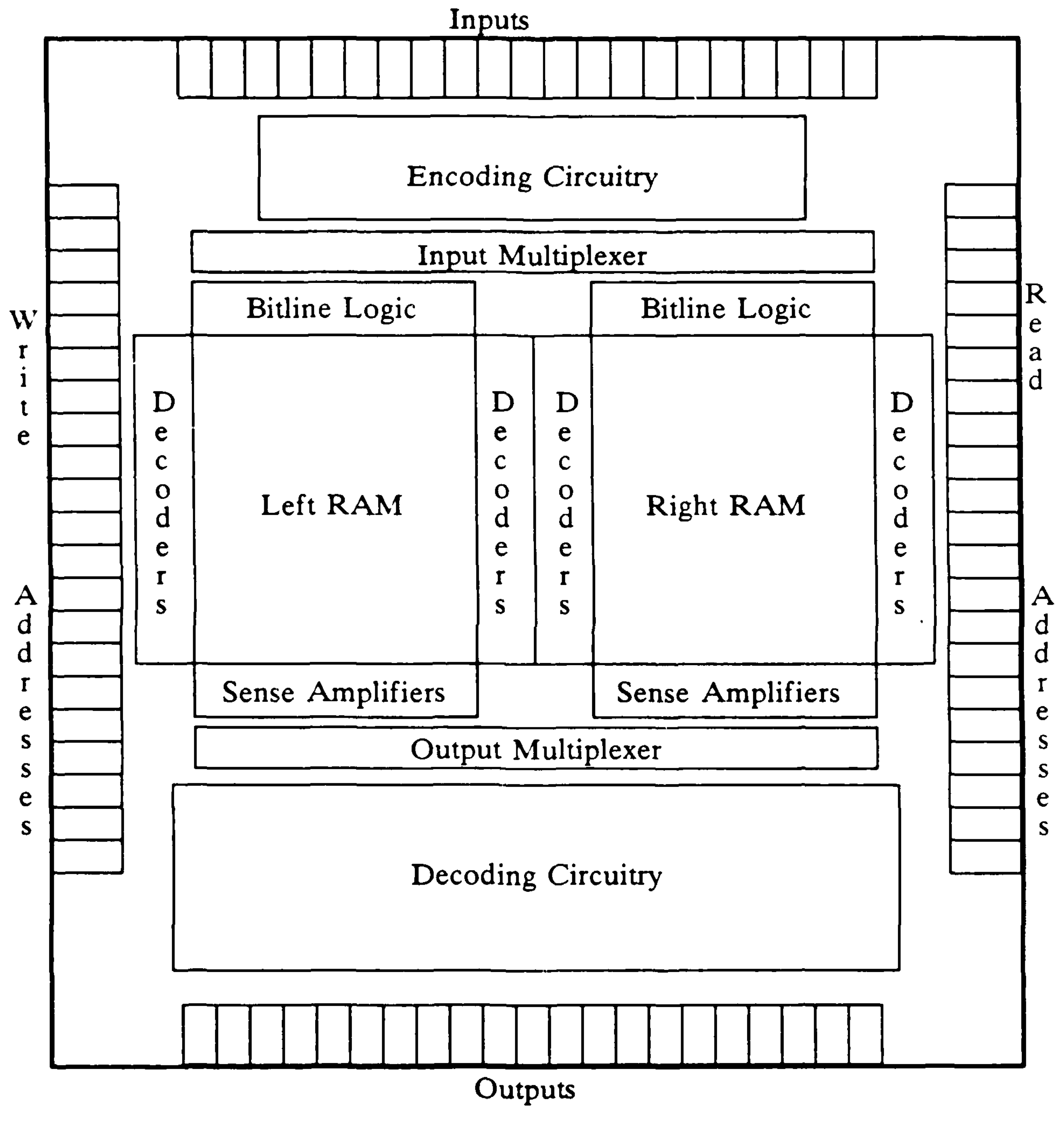


Figure 48 Memory Floorplan

6.1.3. Generic Microcode Assembler Tool. A CAD tool was developed that takes a microcode word format and a mnemonic translation file and builds a customized microcode assembler. The assembler uses the translation file to generate a listing file, a reference file, and a file of the translated microcode. Optionally, the assembler will produce an output of the microcode suitable for input to the optimizing XROM compiler, a file reverse compiled form the translated microcode, or a VHDL description of the XROM.

GMAT was also used in two other thesis efforts and in a class taught af AFIT in the Fall term. Capt. Dave Gallagher used GMAT for his microcode for application specific processors [Gal87] and Capt. Larry Shand used GMAT on a microcode description on a Kalman filter chip to generate a VHDL description [Sha87].

Capt. Gallagh used a preliminary version of GMAT where much of the information needed by GMAT was entered interactively. This process was tedious and very error prone. After the initial assembler was created, GMAT was no longer used. Instead, alterations to the assembler were manually inserted by the author.

Capt. Shand used the final version of GMAT for the Kalman filter application. With this version, GMAT extracted all information from the translation file and no data was entered interactively.

The students in the Introduction to Computer Architecture class used GMAT in the completion of their group projects.

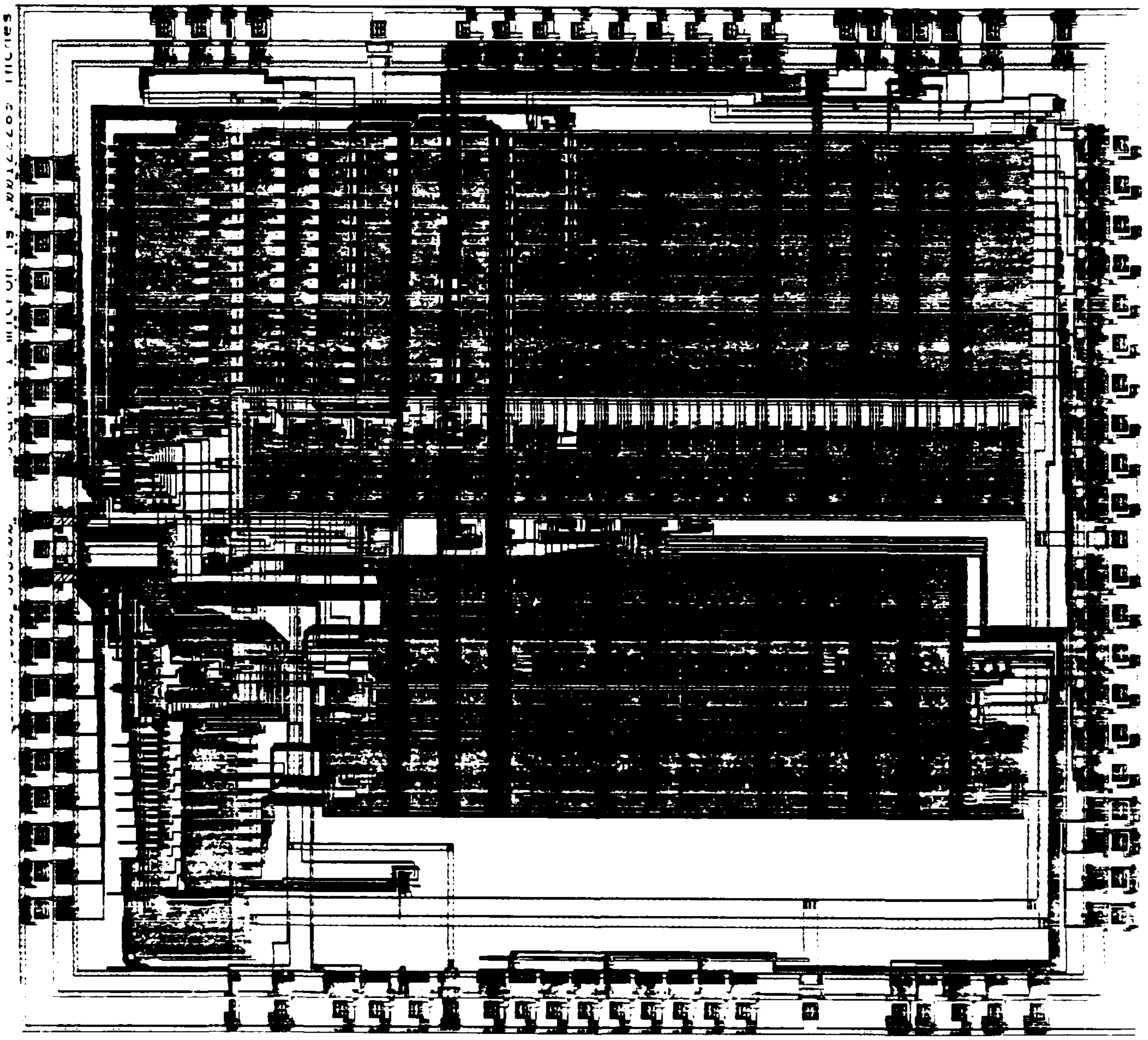


Figure 49 PFA Controller Cifplot

CHAPTER 7

Conclusions and Recommendations

7.1. Conclusions

This thesis has shown that an application specific processor can be designed within four months. With the state of the CAD tool set at the current time, it should be possible to design, submit, and receive application specific processors within one thesis cycle. Efforts due to this thesis and the thesis by Capt. David Gallagher have increased the CAD tool set such that this can be done. It is now possible to have an ALU that will be useful in almost any processor, a control section that is correctly designed, an optimized XROM, and a microcode assembler. Each of these improvements has special importance and decreases the layout time. This thesis used the predesigned ALU, control sequencer, and XROM. However, the design of the ALU was still being done and the control sequencer had never been completely debugged. The XROM functionality had already been proven. By using pre-designed cells, the designer of the processor, can spend more time on other areas of design including testability, controlability, and observability. The designer will also be able to spend more time simulating the circuits in both spice, for timing analysis, and Esim, for functional analysis. All of this increases the probability that the chip will function properly when fabricated.

The microcode assembler is a very useful tool in several areas. First, the microcode can be written in a form that is easy to read and less prone to errors. Once this microcode has been written and debugged, it is no longer necessary for the writer to translate the microcode into an integer format for the XROM optimizer. This greatly reduces the time needed to generate an XROM and the opportunity for human error. Secondly, the reference file containing the microword along with its translations is a useful debugging tool. One can observe the intended instruction, the XROM outputs, and the connections to the XROM. This lets the designer verify all connections to the XROM and the XROM itself. Thirdly, the VHDL output of the XROM will be useful when chip level verification can be done using VHDL. The VHDL environment at the current time does not support easy simulation of VLSI chips. However, when it does this tool will already support an XROM description.

7.2. Recommendations

Several areas in the Prime Factor Algorithm and in the CAD arena still need to be addressed. In the PFA project, the fully functional memory chip and the PFA controller must be tested. Additionally, the chips used for clocking need to be developed. These chips need to be carefully designed to meet the requirements of the pipeline. The chips should be very powerful and able to drive the large currents needed by the WFTs and capacitances associated with it. The WFT15 and WFT17 need to be designed. Although they are just modifications of the WFT16, the time needed for layout and simulation will take approximately one man year for both. The work involved in these design will be intensive, but will not be suitable thesis material. The design and implementation of these chips could be done by a staff engineer. Finally, the prototype PFA-WFT pipeline will need to be implemented and tested. The prototype includes a WFT16 followed by a WFT15 in a two-stage pipeline.

In the CAD arena more areas still need to be developed for streamlined design of application processors. The area most lacking in tools is simulation. Esim is the only tool really used to verify chip design. This will simulate at a switch level but higher level simulation is needed. Two areas of research are currently being developed at AFIT. These are the STOVE (sim to VHDL extraction) project and the ongoing VHDL theses. The STOVE project is attempting to extract chip at a gate-level representation. Currently, it extracts inverters, clocked inverters, and t-gates. The PFA controller was extracted at this level and produced approximately 12000 lines of VHDL.

The current state of the VHDL environment does not support chip level simulation. When it does, however, this will become an important step is VLSI design. Chips will be designed first at the VHDL level, the layout will be done, then the chip will be extracted back to a VHDL description and then compared to the original VHDL description. The tool to complete the design loop will be able to take the original VHDL description and compile it into silicon. This decreases design time and eliminates human error at the layout design level.

This thesis effort, along with the thesis effort of Capt. David Gallagher [Gal87], has shown that it is possible to generate a complete application specific processor within one thesis cycle. This could be very important to the Air Force and the DoD, as well as AFIT. The rapid development of VLSI chips will decrease the time need to insert VLSI technology into existing systems. All the design methodologies associated with application specific processors can be applied to the design to VHSIC systems as well. The AFIT VLSI environment could be developed so that high quality, fast turnaround application specific processors could be produced and tested within one year. To encourage this development several areas could be explored. First, AFIT could be designated as an Air Force "center of excellence." This would establish AFIT as an identified program and allow more resources to be dedicated to VLSI design. Resources are the second area. A set of hardware could be dedicated just for VLSI design. This should include at least one

Sun Workstation and a superminicomputer such as an ELXSI 6400. Third, a civilian staff of at least two people could be dedicated to the VLSI design teams. One person would be responsible for maintaining the CAD tools, systems, and general configuration management. The second person could be a design engineer acquainted with the CAD tools and the cell libraries able to integrate the design into silicon rapidly. With these recommendations, AFIT could be a leader in the field of VLSI/VHSIC insertion.

Bibliography

[Bec85] Beck, Leland L., *System Software: An Introduction to Systems Programming*, Addison-Wesley, Reading, MA (1985).

[Bur83] Burrus, Charles S., "Comments on 'Selection Criteria for Efficient Implementation of FFT Algorithms'," *IEEE Transactions on Acoustics, Speech, and Signal Processing* **1**(31) pp. 206-207 (February 1983).

[Col85] Collins, James M., "Simulation and Modeling of a VLSI Winograd Fourier Transform Processor," *MS Thesis, GE/ENG/85D-9*, School of Engineering, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, (December 1985).

[Coo65] Cooley, J and J. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series," *Mathematics of Computation*, (19) pp. 297-301 (1965).

[Cou85] Coutee, Paul W., "Arithmetic Circuitry for High Speed VLSI Winograd Fourier Transform Processors," *MS Thesis, GE/ENG/85D-11*, School of Engineering, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, (December 1985).

[Fit83] Fitzpatrick, Dan, "Mextra: A Swith Level Simulator ," in *1983 Berkeley CAD Tools User's Manual*, , Berkeley (1983).

[Fre86] French, L. E., "A RISC Controller for the CAM-PUTER System," *M.S. Thesis, AFIT/GE/ENG/86D-53*, School of Engineering, Air Force Institute of Technology (AU), (December 1986).

[Gal87] Gallagher, David M., "Rapid of Prototyping of Application Specific Processors," *MS Thesis, AFIT/GE/ENG/87D*, School of Engineering, Air Force Institute of Technology (AU), (to be published December 1987).

[Gla85] Glasser, L. A. and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley Publishing Company, Reading (1985).

[Goo71] Good, I. J., "The Relationship between Two Fast Fourier Transforms," *IEEE Transactions on Acoustics, Speech, and Signal Processing* **C-20** pp. 310-317 (March 1971).

[Gre84] Grebene, Alan B., *Bipolar and MOS Analog Integrated Circuit Design*, John Wiley and Sons, New York (1984).

[Hed86] Hedrick, Gary D., "Design of Fault Tolerant Prime Factor Algorithm Array Elements," *M.S. Thesis, AFIT/GE/ENG/86D-45*, School of Engineering, Air Force Institute of Technology (AU), (December 1986).

[Lin83] Lin, S. and D. Costello, *Error Control Codes: Fundamentals and Applications*, Prentice Hall, Englewood Cliffs (1983).

[Lin84] Linderman, Richard W., "High Performance VLSI Technologies, Integrated Circuits, and Architectures for Digital Signal Processing," *PhD Thesis*, Cornell University Ithaca, NY, (August 1984).

[Man82] Mano, M. Morris, *Computer System Architecture,* Prentice-Hall, Englewood Cliffs, N.J. (1982).

[Muk86] Mukherjee, Amar, *Introduction to nMOS and CMOS VLSI Systems Design,* Prentice-Hall, Englewood Cliffs, N.J. (1986).

[Ost86] Osterhout, John K., "Magic: a VLSI Layout Editor," in *1986 Berkeley CAD Tools User's Manual*, , Berkeley (1986).

[Pet72] Peterson, William W. and E. J. Weldon, Jr., *Error-Correcting Codes,* MIT Press, Cambridge, MA (1972).

[Rid79] Rideout, Leo V., "One-Device Cells for Dynamic Random Access Memories: A Tutorial," *Transactions on Electron Devices* **26**(6) pp. 839-852 (June 1979).

[Ros85] Rossbach, Paul C., "Control Circuitry for High Speed VLSI Winograd Fourier Transform Processors," *MS Thesis,GE/ENG/85D-35,* School of Engineering, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, (December 1985).

[Sha87] Shand, Larry J., "The VHDL Simulation of a Space Surveillance Signal Processor," *MS Thesis, AFIT/GCE/ENG/87D,* School of Engineering, Air Force Institute of Technology (AU), (to be published December 1987).

[Sha49] Shannon, Claude E. and Warren Weaver, *The Mathematical Theory of Communication,* University of Illinois Press, Urbana (1949).

[She86] Shephard, Carl G., "Integration and Design for Testability of a High Speed Winograd Fourier Transform Processor," *M.S. Thesis, AFIT/GE/ENG/86D-46,* School of Engineering, Air Force Institute of Technology (AU), (December 1986).

[Shi85] Shinn, Wook H., "Design and Implementation of High Performance Content-Addressable Memories," *MS Thesis,GE/ENG/85D-39,* School of Engineering, Air Force Institute of Technology (AU) Wright-Patterson AFB, OH, (December 1985).

[Tay85] Taylor, Kent, "Architecture and Numerical Accuracy of High-Speed DFT Processors," *MS Thesis, AFIT/GE/ENG/85D-47,* School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (December 1985).

[Ter86] Terman, Chris, "Esim: An Event Driven Switch Level Simulator," in *1986 Berkeley CAD Tools User's Manual*, , Berkeley (1986).

[Wes85] Weste, N. H. E. and K. Eshraghian, *Principles of CMOS VLSI Design,* Addison-Wesley Publishing Company, Reading (1985).

[Win78] Winograd, S., "On Computing the Discrete Fourier Transform," *Math Comput.* **32** pp. 175-199 (Jan. 1978).

Vita

Lieutenant Robert S. Hauser was born on 5 January 1964 in Wood River, Illinois. He graduated from the University of Illinois in 1986, and was commissioned in the United States Air Force as a Second Lieutenant. In January 1988, he will begin an assignment at Arnold Engineering Development Center, Tennessee as a data acquisition engineer.

APPENDIX A

Error Correcting Code Matricies

This appendix contains the matricies used for the Error Control Coding in the PFA controller chip.

Table A-1: Parity Generator Matrix

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
u_0	1	1	0	0	0	0	0	0
u_1	0	1	1	0	0	0	0	0
u_2	0	0	1	1	0	0	0	0
u_3	0	0	0	1	1	0	0	0
u_4	0	0	0	0	1	1	0	0
u_5	0	0	0	0	0	1	1	0
u_6	0	0	0	0	0	0	1	1
u_7	1	0	0	0	0	0	0	1
u_8	1	0	1	0	0	0	0	0
u_9	0	1	0	1	0	0	0	0
u_{10}	0	0	1	0	1	0	0	0
u_{11}	0	0	0	1	0	1	0	0
u_{12}	0	0	0	0	1	0	1	0
u_{13}	0	0	0	0	0	1	0	1
u_{14}	1	0	0	0	0	0	1	0
u_{15}	0	1	0	0	0	0	0	1
u_{16}	1	0	0	1	0	0	0	0
u_{17}	0	1	0	0	1	0	0	0
u_{18}	0	0	1	0	0	1	0	0
u_{19}	0	0	0	1	0	0	1	0
u_{20}	0	0	0	0	1	0	0	1
u_{21}	1	0	0	0	0	1	0	0
u_{22}	0	1	0	0	0	0	1	0
u_{23}	0	0	1	0	0	0	0	1

Corresponding parity generator matrix equations:

$v_0 = u_0+u_7+u_8+u_{14}+u_{16}+u_{21}$ $\qquad$ $v_4 = u_3+u_4+u_{10}+u_{12}+u_{17}+u_{20}$

$v_1 = u_0+u_1+u_9+u_{15}+u_{17}+u_{22}$ $\qquad$ $v_5 = u_4+u_5+u_{11}+u_{13}+u_{18}+u_{21}$

$v_2 = u_1+u_2+u_8+u_{10}+u_{18}+u_{23}$ $\qquad$ $v_6 = u_5+u_6+u_{12}+u_{14}+u_{19}+u_{22}$

$v_3 = u_2+u_3+u_9+u_{11}+u_{16}+u_{19}$ $\qquad$ $v_7 = u_6+u_7+u_{13}+u_{15}+u_{20}+u_{23}$

Table A-2: Syndrome Generator Matrix

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
r_0	1	0	0	0	0	0	0	0
r_1	0	1	0	0	0	0	0	0
r_2	0	0	1	0	0	0	0	0
r_3	0	0	0	1	0	0	0	0
r_4	0	0	0	0	1	0	0	0
r_5	0	0	0	0	0	1	0	0
r_6	0	0	0	0	0	0	1	0
r_7	0	0	0	0	0	0	0	1
r_8	1	1	0	0	0	0	0	0
r_9	0	1	1	0	0	0	0	0
r_{10}	0	0	1	1	0	0	0	0
r_{11}	0	0	0	1	1	0	0	0
r_{12}	0	0	0	0	1	1	0	0
r_{13}	0	0	0	0	0	1	1	0
r_{14}	0	0	0	0	0	0	1	1
r_{15}	1	0	0	0	0	0	0	1
r_{16}	1	0	1	0	0	0	0	0
r_{17}	0	1	0	1	0	0	0	0
r_{18}	0	0	1	0	1	0	0	0
r_{19}	0	0	0	1	0	1	0	0
r_{20}	0	0	0	0	1	0	1	0
r_{21}	0	0	0	0	0	1	0	1
r_{22}	1	0	0	0	0	0	1	0
r_{23}	0	1	0	0	0	0	0	1
r_{24}	1	0	0	1	0	0	0	0
r_{25}	0	1	0	0	1	0	0	0
r_{26}	0	0	1	0	0	1	0	0
r_{27}	0	0	0	1	0	0	1	0
r_{28}	0	0	0	0	1	0	0	1
r_{29}	1	0	0	0	0	1	0	0
r_{30}	0	1	0	0	0	0	1	0
r_{31}	0	0	1	0	0	0	0	1

Corresponding syndrome generator equations:

$s_0 = r_0+r_8+r_{15}+r_{16}+r_{22}+r_{24}+r_{29}$ $\quad$ $s_4 = r_4+r_{11}+r_{12}+r_{18}+r_{20}+r_{25}+r_{28}$

$s_1 = r_1+r_8+r_9+r_{17}+r_{23}+r_{25}+r_{30}$ $\quad$ $s_5 = r_5+r_{12}+r_{13}+r_{19}+r_{21}+r_{26}+r_{29}$

$s_2 = r_2+r_9+r_{10}+r_{16}+r_{18}+r_{26}+r_{31}$ $\quad$ $s_6 = r_6+r_{13}+r_{14}+r_{20}+r_{22}+r_{27}+r_{30}$

$s_3 = r_3+r_{10}+r_{11}+r_{17}+r_{19}+r_{24}+r_{27}$ $\quad$ $s_7 = r_7+r_{14}+r_{15}+r_{21}+r_{23}+r_{28}+r_{31}$ $sp3$

Table A-3: Error Bit Generator Matrix

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
e_0	1	0	0	0	0	0	0	0
e_1	0	1	0	0	0	0	0	0
e_2	0	0	1	0	0	0	0	0
e_3	0	0	0	1	0	0	0	0
e_4	0	0	0	0	1	0	0	0
e_5	0	0	0	0	0	1	0	0
e_6	0	0	0	0	0	0	1	0
e_7	0	0	0	0	0	0	0	1
e_8	1	1	0	0	0	0	0	0
e_9	0	1	1	0	0	0	0	0
e_{10}	0	0	1	1	0	0	0	0
e_{11}	0	0	0	1	1	0	0	0
e_{12}	0	0	0	0	1	1	0	0
e_{13}	0	0	0	0	0	1	1	0
e_{14}	0	0	0	0	0	0	1	1
e_{15}	1	0	0	0	0	0	0	1
e_{16}	1	0	1	0	0	0	0	0
e_{17}	0	1	0	1	0	0	0	0
e_{18}	0	0	1	0	1	0	0	0
e_{19}	0	0	0	1	0	1	0	0
e_{20}	0	0	0	0	1	0	1	0
e_{21}	0	0	0	0	0	1	0	1
e_{22}	1	0	0	0	0	0	1	0
e_{23}	0	1	0	0	0	0	0	1
e_{24}	1	0	0	1	0	0	0	0
e_{25}	0	1	0	0	1	0	0	0
e_{26}	0	0	1	0	0	1	0	0
e_{27}	0	0	0	1	0	0	1	0
e_{28}	0	0	0	0	1	0	0	1
e_{29}	1	0	0	0	0	1	0	0
e_{30}	0	1	0	0	0	0	1	0
e_{31}	0	0	1	0	0	0	0	1

Corresponding error bit equations:

$$e_0 = \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_1 = s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_2 = s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_3 = s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_4 = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_5 = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot s_6 \cdot s_7$$

$$e_6 = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot \overline{s_6} \cdot s_7$$

$$e_7 = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot \overline{s_7}$$

$$e_8 = \overline{s_0} \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_9 = s_0 \cdot \overline{s_1} \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{10} = s_0 \cdot s_1 \cdot \overline{s_2} \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{11} = s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot \overline{s_4} \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{12} = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot \overline{s_5} \cdot s_6 \cdot s_7$$

$$e_{13} = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot \overline{s_6} \cdot s_7$$

$$e_{14} = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot \overline{s_6} \cdot \overline{s_7}$$

$$e_{15} = \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot \overline{s_7}$$

$$e_{16} = \overline{s_0} \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{17} = s_0 \cdot \overline{s_1} \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{18} = s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{19} = s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot \overline{s_5} \cdot s_6 \cdot s_7$$

$$e_{20} = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot \overline{s_6} \cdot s_7$$

$$e_{21} = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot s_6 \cdot \overline{s_7}$$

$$e_{22} = \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot \overline{s_6} \cdot s_7$$

$$e_{23} = s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot \overline{s_7}$$

$$e_{24} = \overline{s_0} \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{25} = s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot s_6 \cdot s_7$$

$$e_{26} = s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot s_6 \cdot s_7$$

$$e_{27} = s_0 \cdot s_1 \cdot s_2 \cdot \overline{s_3} \cdot s_4 \cdot s_5 \cdot \overline{s_6} \cdot s_7$$

$$e_{28} = s_0 \cdot s_1 \cdot s_2 \cdot s_3 \cdot \overline{s_4} \cdot s_5 \cdot s_6 \cdot \overline{s_7}$$

$$e_{29} = \overline{s_0} \cdot s_1 \cdot s_2 \cdot s_3 \cdot s_4 \cdot \overline{s_5} \cdot s_6 \cdot s_7$$

$$e_{30} = s_0 \cdot \overline{s_1} \cdot s_2 \cdot s_3 \cdot s_4 \cdot s_5 \cdot \overline{s_6} \cdot s_7$$

$$e_{31} = s_0 \cdot s_1 \cdot \overline{s_2} \cdot s_3 \cdot s_4 \cdot s_5 \cdot s_6 \cdot \overline{s_7}$$

APPENDIX B

GMAT Shell Script

```
#! /bin/sh
#
# DATE: 1 DEC 1987
# Version: 1.0
#
# NAME:   gmat
# DESCRIPTION:
#    This shell script run the Generic Microcode Assembler Tool (GMAT).
#    It first calls preg to parse the translation file and write out
#    the tailored C code. If the call to preg was successful, gmat
#    copies assem.c into the directory and compiles the assembler.
#    After compilation, gmat removes assem.c.
#
# AUTHOR: Robert S. Hauser
# HISTORY:
#
echo Running preg
if /eng/87d/rhauser/bin/preg $1
then
   echo Copying library source code into this directory
   cp /eng/87d/rhauser/bin/assem.c assem.c;
   echo Compiling your assembler
   cc -O -o assem assem.c -lm;
   echo Removing library source code
   rm assem.c
else
   echo Error in running preg.
   echo
fi
```

APPENDIX C

PREG C Code

```
/*********************************************************************
*
* DATE: 1 DEC 1987
* Version: 1.0
*
* TITLE: Pre Gmat routine
* FILENAME: PREG.C
* COORDINATOR: Capt R W Linderman
* PROJECT: Generic Microcode Assembler Tool (GMAT)
* OPERATING SYSTEM: UNIX 4.3BSD
* LANGUAGE: C
* CONTENTS:
*                get_answer()
*                get_names()
*                build_assem()
*                get_micro_format()
*                scan_t_file()
*                update_blengths()
*
* AUTHOR: Robert S. Hauser
* HISTORY:
********************************************************************/
#include <stdio.h>
#define TRUE 1
#define FALSE 0

struct WD_FIELD_ENTRY{
       char name[1000];
       int  literal;
       int  label;
       int  blength;
       };
struct T_ENTRY{
       char name[1000];
       int  blength;
       };
struct WD_FIELD_ENTRY fields[1000];
struct T_ENTRY t_fields[1000];

FILE *tranfile,*gmathead;
char word[1000],answer[2];
int  num_sub_fields;
int  num_wd_fields,num_t_fields;
int  bits_in_word;
int  ok;
int  i,k,j;
char wordend[1000];
char field_name_tmp[1000];
char empty[]= " ";
char fill[1000];
int  ch;

main(argc,argv) int argc;char **argv;
{
  if (argc != 2)
    {
    printf("\nUsage:  preg   translation_file_name\n\n");
    exit(1);
    }
  if ((tranfile = fopen(argv[1],"r")) == NULL)


     {
        printf("\nFile %s could not be found.\n\n",argv[1]);
        exit(1);
     }

  get_micro_format();
  printf("\n");
  scan_t_file();
  update_blengths();
  build_assem(argv[1]);
}/*end main*/

/*******************************************************************
* DATE: 1 DEC 1987
* Version: 1.0
* PROCEDURE: get_names()
* DESCRIPTION:
*               Reads translation file and pulls out the field name
*
* PASSED VARIABLES:
* RETURNS:
*               t_field_name : name of field
*               t_field_blength : field length
* GLOBAL VARIABLES USED:
*               EOF
*               ch
*               tranfile
*               num_sub_fields
* GLOBAL VARIABLES CHANGED:
*               ch
*               tranfile
*               num_sub_fields
* FILES READ:
*               tranfile
* FILES WRITTEN:
* MODULES CALLED:
* CALLING MODULES:
*               scan_t_file()
*
* AUTHOR: Robert S. Hauser
* HISTORY:
*******************************************************************/
```

```
get_names(t_field_name,t_field_blength)
               char t_field_name[];
               int  *t_field_blength;
{
char t_field_value[100];
num_sub_fields = 0;
for(;;)
 {
 k = j = 0;
 ch = fgetc(tranfile);
 while (ch == ' ') ch = fgetc(tranfile); /* ignore leading blanks */
 if (ch == '\n'|| ch == EOF) break;
          /* if at end of blank line or file quit*/
 while (ch != '\n' && ch != ' ')/* until EOL or blank */
   {
   if (num_sub_fields == 0) t_field_name[k++] = ch;
   ch = fgetc(tranfile);
   }

   if (num_sub_fields == 0) t_field_name[k] = '\0';
   num_sub_fields++;
   while (ch == ' ') ch = fgetc(tranfile); /* skip blanks */
   while (ch != '\n' && ch != ' ')
     {
     if (num_sub_fields == 1) t_field_value[j++] = ch;
     ch = fgetc(tranfile);
     }
   if (num_sub_fields == 1) *t_field_blength = j;
       /* ignore trailing blanks */
   while (ch == ' ') ch = fgetc(tranfile); /* ignore trailing blanks */
   }
  }
  /*******************************************************************
  * DATE: 1 DEC 1987
  * Version: 1.0
  * PROCEDURE: build_assem()
  * DESCRIPTION:
  *              Makes read_trans_table(), translate(), make_reffile()
  *
  * PASSED VARIABLES:
  *              filename : name of translation file
  * RETURNS:
  * GLOBAL VARIABLES USED:
  *              bits_in_word
  *              num_wd_fields
  * GLOBAL VARIABLES CHANGED:
  * FILES READ:
  * FILES WRITTEN:
  *              assem.h
  *              assem.tailored
  * MODULES CALLED:
  * CALLING MODULES:
  *              main()
  *
  * AUTHOR: Robert S. Hauser
  * HISTORY:
  *******************************************************************/
```

```
build_assem(filename) char filename[];
{
int i;
FILE *aheader,*atailored;
aheader = fopen("assem.h","w");
fprintf(aheader,"#define BitsInWord %d\n",bits_in_word);
for(i=0;i<num_t_fields;i++)
    fprintf(aheader,"struct SYMBOL_ENTRY %stbl[MaxSubFields];\n",
                                               t_fields[i].name);

for(i=0;i<num_t_fields;i++)
    fprintf(aheader,"int num_fields%s;\n",t_fields[i].name);

atailored = fopen("assem.tailored","w");

/******************** make read_trans_table() *****************/
fprintf(atailored,"/*******************************/\n");
fprintf(atailored,"read_trans_table()\n{\n");
fprintf(atailored,"     char   fill[1000],filll[1000];\n");
fprintf(atailored,"     symbolfile = fopen(\"%s\",\"r\");
                                              \n",filename);
fprintf(atailored,"     fscanf(symbolfile,\"%%[^\\n]%%


                                    [\\ \\n]\",fill,filll);\n");
for(i=0;i<num_t_fields;i++)
  fprintf(atailored,"     readin(%stbl,&num_fields%s);\n"
                             ,t_fields[i].name,t_fields[i].name);
fprintf(atailored,"     fclose(symbolfile);\n}
                                  /* end read_trans_table */");

/************************ make translate() ******************/
fprintf(atailored,"\ntranslate()\n{\n     ");
fprintf(atailored,"stripped = fopen(strip_file,\"r\");\n");
fprintf(atailored,"transfile = fopen(trans_file,\"w\");\n");
fprintf(atailored,"fscanf(stripped,\"%%\s\",input);\n");
fprintf(atailored,"while(strcmp(END,input)!=0)\n{\n");


for(i=0;i<num_wd_fields;i++)
  {
   if (fields[i].literal == TRUE)
      {
       fprintf(atailored,"    if (literal(input)==TRUE) \n");
       fprintf(atailored,"       fscanf(stripped,\"%%\s\"
                                           ,input);\n");
       fprintf(atailored,"    else\n      ");
      }
   if (fields[i].label == TRUE)
      {
       fprintf(aheader,"int   lab_b_length = %d;\n",fields[i].blength);
       fprintf(atailored,"    if (symtrans(Labeltbl,
                                input,index_to_labels)==TRUE) \n");
       fprintf(atailored,"       fscanf(stripped,\"%%\s\",input);\n");
       fprintf(atailored,"    else\n            ");
      }
   fprintf(atailored,"if(symtrans(%stbl,input,num_fields%s)==TRUE)
   fscanf(stripped,\"%%\s\",input);\n",fields[i].name,fields[i].name);
  } /* end for num_wd_fields */
fprintf(atailored,"\nif(strcmp(NOP,input) == FALSE)
                    fscanf(stripped,\"%%\s\",input);\n");
fprintf(atailored,"\nif(input[0]=='+') fprintf(transfile,\" +\\n\");\n");
fprintf(atailored,"\nelse\nprintf(\"\\nERROR: symbol
                                >%%\s< not defined\\n\",input);\n");
fprintf(atailored," fscanf(stripped,\"%%\s\",input);\n");
fprintf(atailored,"\n}\nfclose(transfile);
                    \nfclose(stripped);\n}/* end translate*/\n");


/************* make make_reffile() *********************/
fprintf(atailored,"\nmake_reffile()\n{\n");
fprintf(atailored,"    char ");
for(i=0;i<num_wd_fields;i++)
  {
  if (i!=0) fprintf(atailored,",");
  fprintf(atailored,"t%d[%d+1]",i,fields[i].blength);
  }

fprintf(atailored,";\nlistingfile = fopen(l_file,\"r\");");
fprintf(atailored,"\ntransfile = fopen(trans_file,\"r\");");
fprintf(atailored,"\nreffile = fopen(r_file,\"w\");");
fprintf(atailored,"\nfscanf(listingfile,\"%%\[\^;\];\\n\",line);");
fprintf(atailored,"\nwhile(strcmp(END,line)!=0)\n{\n");
fprintf(atailored,"\n  strcat(line,EOL2);");

fprintf(atailored,"\n  fprintf(reffile,\"%%-50s\",line);");
fprintf(atailored,"\n  fscanf(transfile,\"%%\[\^+\]+\\n\",line1);");
fprintf(atailored,"\n  sscanf(line1,\"");
for(i=0;i<num_wd_fields;i++)
  {
  fprintf(atailored,"%%%ds" ,fields[i].blength);
  }
fprintf(atailored,"\",");
for(i=0;i<num_wd_fields;i++)
  {
  if (i!=0) fprintf(atailored,",");
  fprintf(atailored,"t%d ",i);
  }
fprintf(atailored,");");
fprintf(atailored,"\n  fprintf(reffile,\"\\n");
for(i=0;i<num_wd_fields;i++)
  {
  fprintf(atailored,"%%s ");
  }
fprintf(atailored,"\\n\",");
for(i=0;i<num_wd_fields;i++)
  {
  if (i!=0) fprintf(atailored,",");
  fprintf(atailored,"t%d",i);
  }
fprintf(atailored,");");
fprintf(atailored,"\nfscanf(listingfile,\"%%\
                         [\^;\];\\n\",line);\n}\n");
fprintf(atailored,"fprintf(reffile,\"end;\\n\");");
fprintf(atailored,"\nfclose(transfile);");
fprintf(atailored,"\nfclose(reffile);");
fprintf(atailored,"\nfclose(listingfile);");
fprintf(atailored,"\n}\/*** end make_reffile ****\/\n");
```

```
/*********  make reverse_comp()  ***********/
fprintf(atailored,"\nreverse_comp()\n{\n     ");
fprintf(atailored,"revfile = fopen(r_file,\"w\");\n");
fprintf(atailored,"transfile = fopen(trans_file,\"r\");\n");

fprintf(atailored,"for(i=0;i<line_num-1;i++)\n  {\n");
fprintf(atailored,"fprintf(revfile,\"%%0.6d   \",i);");
for(i=0;i<num_wd_fields;i++)
  {
   fprintf(atailored,"fscanf(transfile,\"%%%ds\",input);
                                       \n",fields[i].blength);
   if (fields[i].label == TRUE)
     {
      fprintf(atailored,"if (revtrans(%stbl,input,num_fields%s)
                            ==FALSE);\n",fields[i].name,fields[i].name);
      fprintf(atailored,"{\nif (convert(input) != 0)\n");
      fprintf(atailored,"fprintf(revfile,\"   %%\d(%%\s)\"
                                     ,convert(input),input);\n}\n");
     }
  else
     {
      fprintf(atailored,"revtrans(%stbl,input,num_fields%s);\n"
                                     ,fields[i].name,fields[i].name);
     }
  } /* end for num_wd_fields */
fprintf(atailored," fscanf(transfile,\"%%\s\",input);\n");

fprintf(atailored," fprintf(revfile,\"\\n\");");
fprintf(atailored,"\n}\nfclose(transfile);
                         \nfclose(revfile);\n}/* end reverse comp*/\n");


}/****** end build assembly () ******/
/*********************************************************************
* DATE: 1 DEC 1987
* Version: 1.0
* PROCEDURE: get_micro_format()
* DESCRIPTION:
*               Reads the first line of the translation file and
*               pulls out the word format
*
* PASSED VARIABLES:
* RETURNS:
* GLOBAL VARIABLES USED:
*               word
*               fill
*               wordend
*               field_name_tmp
*               fields
*               empty
* GLOBAL VARIABLES CHANGED:
*               word
*               fill
*               wordend
*               field_name_tmp
*               fields
*               empty
* FILES READ:
*                tranfile
* FILES WRITTEN:
* MODULES CALLED:
* CALLING MODULES:
*                main()
*
* AUTHOR: Robert S. Hauser
* HISTORY:
*******************************************************************/
get_micro_format()
{
  fscanf(tranfile,"%[^;];%[\ \n]",word,fill);
  printf("Microword format: %s ",word);
  num_wd_fields=0;
  do
   {
     strcpy(wordend,empty);
     sscanf(word,"%s%[\ ]%[^;]",field_name_tmp,fill,wordend);
     strcpy(word,wordend);
     if (index(field_name_tmp,'#')) fields[num_wd_fields].literal=TRUE;
     if (index(field_name_tmp,':')) fields[num_wd_fields].label=TRUE;
     sscanf(field_name_tmp,"%[^#:]",fields[num_wd_fields++].name);
   }/* for num fields */
  while (strcmp(empty,word)!=0 );
}/**** end get_micro_format ******/
/*******************************************************************
* DATE: 1 DEC 1987
* Version: 1.0
* PROCEDURE: scan_t_file()

* DESCRIPTION:
*                Reads the translation file and pulls out the
*                field names and put them in t_fields
*
* PASSED VARIABLES:
* RETURNS:
* GLOBAL VARIABLES USED:
*                t_fields
* GLOBAL VARIABLES CHANGED:
*                t_fields
* FILES READ:
* FILES WRITTEN:
* MODULES CALLED:
*                get_names()
* CALLING MODULES:
*                main()
*
* AUTHOR: Robert S. Hauser
* HISTORY:
*******************************************************************/
scan_t_file()
{
  num_t_fields=0;
  do
   {
     get_names(t_fields[num_t_fields].name,
                              &t_fields[num_t_fields].blength);
     num_t_fields++;
   }
  while(ch != EOF);
} /****** end scanf_t_file  *****/
```

```
/*********************************************************************
* DATE: 1 DEC 1987
* Version: 1.0
* PROCEDURE: update_blengths()
* DESCRIPTION:
*               Determines the number of bits in each field and
*               the total number of bits in the word.
*
* PASSED VARIABLES:
* RETURNS:
* GLOBAL VARIABLES USED:
*               num_t_fields
*               num_wd_fields
*               t_fields
*               bits_in_word
* GLOBAL VARIABLES CHANGED:
*               t_fields
*               bits_in_word
* FILES READ:
* FILES WRITTEN:
* MODULES CALLED:
* CALLING MODULES:
*               main()
*
* AUTHOR: Robert S. Hauser
* HISTORY:
*********************************************************************/
update_blengths()
{



  bits_in_word = 0;
      for(i=0;i<num_t_fields;i++)
          for(j=0;j<num_wd_fields;j++)
             if (strcmp(t_fields[i].name,fields[j].name)==0)
                {
                  fields[j].blength = t_f
```

APPENDIX D

Code Created by PREG

```
#define BitsInWord 60
struct SYMBOL_ENTRY BR_SELtbl[MaxSubFields];
struct SYMBOL_ENTRY ALUtbl[MaxSubFields];
struct SYMBOL_ENTRY LOAD_FDtbl[MaxSubFields];
struct SYMBOL_ENTRY REGtbl[MaxSubFields];
struct SYMBOL_ENTRY SPEC_FUNCTtbl[MaxSubFields];
struct SYMBOL_ENTRY WFTOPtbl[MaxSubFields];
struct SYMBOL_ENTRY PFADONEtbl[MaxSubFields];
struct SYMBOL_ENTRY NXT_ADDRtbl[MaxSubFields];
int num_fieldsBR_SEL;
int num_fieldsALU;
int num_fieldsLOAD_FD;
int num_fieldsREG;
int num_fieldsSPEC_FUNCT;
int num_fieldsWFTOP;
int num_fieldsPFADONE;
int num_fieldsNXT_ADDR;
int   lab_b_length = 16;



read_trans_table()
{
     char   fill[100],filll[100];
     symbolfile = fopen("t_file","r");
     fscanf(symbolfile,"%[^\n]%[\ \n]",fill,filll);
     readin(BR_SELtbl,&num_fieldsBR_SEL);
     readin(ALUtbl,&num_fieldsALU);
     readin(LOAD_FDtbl,&num_fieldsLOAD_FD);
     readin(REGtbl,&num_fieldsREG);
     readin(SPEC_FUNCTtbl,&num_fieldsSPEC_FUNCT);
     readin(WFTOPtbl,&num_fieldsWFTOP);
     readin(PFADONEtbl,&num_fieldsPFADONE);
     readin(NXT_ADDRtbl,&num_fieldsNXT_ADDR);
     fclose(symbolfile);
}/* end read_trans_table */
```

```
translate()
{
     stripped = fopen(strip_file,"r");
     transfile = fopen(trans_file,"w");
     fscanf(stripped,"%s",input);
     while(strcmp(END,input)!=0)
     {
     if(symtrans(BR_SELtbl,input,num_fieldsBR_SEL)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(ALUtbl,input,num_fieldsALU)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(LOAD_FDtbl,input,num_fieldsLOAD_FD)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(REGtbl,input,num_fieldsREG)==TRUE)
                    fscanf(stripped,"%s",input);

     if(symtrans(REGtbl,input,num_fieldsREG)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(REGtbl,input,num_fieldsREG)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(SPEC_FUNCTtbl,input,num_fieldsSPEC_FUNCT)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(WFTOPtbl,input,num_fieldsWFTOP)==TRUE)
                    fscanf(stripped,"%s",input);
     if(symtrans(PFADONEtbl,input,num_fieldsPFADONE)==TRUE)
                    fscanf(stripped,"%s",input);
    if (literal(input)==TRUE)
       fscanf(stripped,"%s",input);
    else
           if (symtrans(Labeltbl,input,index_to_labels)==TRUE)
       fscanf(stripped,"%s",input);
    else
             if(symtrans(NXT_ADDRtbl,input,num_fieldsNXT_ADDR)==TRUE)
                                        fscanf(stripped,"%s",input);

    if(strcmp(NOP,input) == FALSE)  fscanf(stripped,"%s",input);

    if(input[0]=='+') fprintf(transfile," +\n");

    else
    printf("\nERROR: symbol >%s< not defined\n",input);
    fscanf(stripped,"%s",input);

   }
fclose(transfile);
fclose(stripped);
}/* end translate*/

make_reffile()
{
    char t0[9+1],t1[4+1],t2[1+1],t3[5+1],t4[5+1],t5[5+1],
                                t6[13+1],t7[1+1],t8[1+1],t9[16+1];
    listingfile = fopen(l_file,"r");
    transfile = fopen(trans_file,"r");
    reffile = fopen(r_file,"w");
    fscanf(listingfile,"%[^;];\n",line);
    while(strcmp(END,line)!=0)
    {

    strcat(line,EOL2);
    fprintf(reffile,"%-50s",line);
    fscanf(transfile,"%[^+]+\n",line1);
    sscanf(line1,"%9s%4s%1s%5s%5s%5s%13s%1s%1s%16s",t0,t1,t2,t3,
                                                        t4,t5,t6,t7,t8,t9);
    fprintf(reffile,"\n%s %s %s %s %s %s %s %s %s %s \n",t0,t1,
                                                    t2,t3,t4,t5,t6,t7,t8,t9);
    fscanf(listingfile,"%[^;];\n",line);
    }
    fprintf(reffile,"end;\n");
    fclose(transfile);
    fclose(reffile);
    fclose(listingfile);
}/*** end make_reffile ****/

reverse_comp()
{

      revfile = fopen(r_file,"w");
      transfile = fopen(trans_file,"r");
      for(i=0;i<line_num-1;i++)
        {
        fprintf(revfile,"%0.6d   ",i);fscanf(transfile,"%9s",input);
        revtrans(BR_SELtbl,input,num_fieldsBR_SEL);
        fscanf(transfile,"%4s",input);
        revtrans(ALUtbl,input,num_fieldsALU);
        fscanf(transfile,"%1s",input);
        revtrans(LOAD_FDtbl,input,num_fieldsLOAD_FD);
        fscanf(transfile,"%5s",input);
        revtrans(REGtbl,input,num_fieldsREG);
        fscanf(transfile,"%5s",input);
        revtrans(REGtbl,input,num_fieldsREG);
        fscanf(transfile,"%5s",input);
        revtrans(REGtbl,input,num_fieldsREG);
        fscanf(transfile,"%13s",input);
        revtrans(SPEC_FUNCTtbl,input,num_fieldsSPEC_FUNCT);
        fscanf(transfile,"%1s",input);
        revtrans(WFTOPtbl,input,num_fieldsWFTOP);
        fscanf(transfile,"%1s",input);
        revtrans(PFADONEtbl,input,num_fieldsPFADONE);
        fscanf(transfile,"%16s",input);
        if (revtrans(NXT_ADDRtbl,input,num_fieldsNXT_ADDR)==FALSE);
        {
        if (convert(input) != 0)
        fprintf(revfile,"   %d(%s)",convert(input),input);
        }
         fscanf(transfile,"%s",input);
         fprintf(revfile,"\n");
        }
        fclose(transfile);
        fclose(revfile);
 }/* end reverse comp*/
```

APPENDIX E

Assembler Skeleton

```
/*********************************************************************
*
* DATE: 1 DEC 1987
* Version: 1.0
*
* TITLE: Assembler Skeleton
* FILENAME: ASSEM.C
* COORDINATOR: Capt R W Linderman
* PROJECT: Generic Microcode Assembler Tool (GMAT)
* OPERATING SYSTEM: UNIX 4.3BSD
* LANGUAGE: C
* CONTENTS:
*          readin()
*          symtrans()
*          literal()
*          itobs()
*          convert()
*          revtrans()
*          vhdl_out()
*          strip()
*          make_xromaddrs()
*
* AUTHOR: Robert S. Hauser
* HISTORY:
*********************************************************************/
#include <stdio.h>
#include <math.h>
#define EOL                         "+"
#define EOL2                        ";"
#define MaxFieldName                100
#define MaxFileName                 100
#define MaxValueLength              100
#define MaxSubFields                100
#define MaxSubFieldLength           100
#define MaxLineLength               100
#define MaxLabelLength              100
#define MaxLabels                   100
#define TRUE                        1
#define FALSE                       0

struct SYMBOL_ENTRY {
       char SubField[MaxSubFieldLength];
       char Value[MaxValueLength];
};
```

```
FILE *revfile,*symbolfile,*stripped,*transfile;
FILE *reffile,*xromaddrs,*infile,*listingfile;

char strip_file[MaxFileName],i_file[MaxFileName];
char l_file[MaxFileName],r_file[MaxFileName];
char trans_file[MaxFileName],a_file[MaxFileName],v_file[MaxFileName];
char line[MaxLineLength],line_no_lab[MaxLineLength];
char line1[MaxLineLength];
char temp[MaxLineLength],label[MaxLabelLength];
char LABELarray[MaxLabels][MaxLabelLength];
char input[MaxSubFieldLength];
char line_num1[MaxValueLength];
char slice[MaxValueLength];
char tchar;

char remove[50];

char END[]="end";
char END2[]="end;";
char LST[]=".lst";
char MC[]=".mc";
char REF[]=".ref";
char STRIP[]=".strip";
char TRANS[]=".trans";
char VHDL[]=".vhd";
char ADDR[]=".addr";
char NOP[]="nop";
char COL[]=':';

int  LABELaddr[MaxLabels];
int  i,j,k,index_to_labels;
int  line_num,label_num;
int  xrom_length;
int  num,base,indexa;
int  ch;
int  b_length;
int generate_xromaddrs,reverse_compile,generate_vhdl;
int clean;

struct SYMBOL_ENTRY Labeltbl[MaxSubFields];
#include "assem.h"

main(argc,argv) int argc;
char **argv;
{
        line_num =0;
        label_num = 0;
        index_to_labels = 1;
        generate_xromaddrs = FALSE;
        reverse_compile = FALSE;
        generate_vhdl = FALSE;
        clean = TRUE;
        if ((argc < 2)||(argc > 3))
        {
                printf("\n\nUsage:  assem file_name [xvrd]\n\n");
                exit(1);
        }
        if (argc == 3)  /* then options */
        {
        if (index(argv[2],'x') != NULL) generate_xromaddrs = TRUE;
        if (index(argv[2],'v') != NULL) generate_vhdl= TRUE;
        if (index(argv[2],'r') != NULL) reverse_compile = TRUE;
        if (index(argv[2],'d') != NULL) clean = FALSE;
        };

        strcpy(i_file,argv[1]);
        strcpy(l_file,argv[1]);
        strcat(l_file,LST);
        strcpy(r_file,argv[1]);
        strcat(r_file,REF);
        strcpy(strip_file,argv[1]);
        strcat(strip_file,STRIP);
        strcpy(trans_file,argv[1]);
        strcat(trans_file,TRANS);


        strcpy(v_file,argv[1]);
        strcat(v_file,VHDL);
        strcpy(a_file,argv[1]);
        strcat(a_file,ADDR);

        strip();
        read_trans_table();
        translate();
        make_reffile();
        if (generate_xromaddrs)
                make_xromaddrs();
        if (reverse_compile)
                reverse_comp();
        if (generate_vhdl)
                vhdl_out();

        if (clean)
          {
          strcpy(remove,"rm ");
          strcat(remove,trans_file);
          strcat(remove," ");
          strcat(remove,strip_file);
          system(remove);
           }

}/* end main program */
```

```
/*******************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: readin()
*  DESCRIPTION:
*       This procedure reads in the translation file one char
*       at a time to get the field name and the value.
*
*  PASSED VARIABLES: none
*  RETURNS:
*                 t_field : pointer to structure for field
*                 num_sub_fields : number of fields found
*  GLOBAL VARIBLES USED:
*                 symbolfile
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*                 symbolfile : program name for the translation file
*  FILES WRITTEN:
*  MODULES CALLED:
*  CALLING MODULES:
*                 read_trans_table
*  AUTHOR: Robert S. Hauser
*  HISTORY:
*******************************************************************/

readin(t_field,num_sub_fields)
              struct SYMBOL_ENTRY t_field[];
              int *num_sub_fields;
{
i = 0;
for(;;)
 {
 k = j = 0;

 ch = fgetc(symbolfile);
 while (ch == ' ') ch = fgetc(symbolfile); /* ignore leading blanks */
 if (ch == '\n'|| ch == EOF)
   {
    *num_sub_fields = i;
    break; /* if at end of blank line or file quit*/
    }
 while (ch != '\n' && ch != ' ')/* until EOL or blank */
   {
   t_field[i].SubField[k++] = ch;
   ch = fgetc(symbolfile);
   }
 t_field[i].SubField[k] = '\0';
 while (ch == ' ') ch = fgetc(symbolfile); /* skip blanks */
 while (ch != '\n' && ch != ' ')
   {
   t_field[i].Value[j++] = ch;
   ch = fgetc(symbolfile);
   }
 t_field[i++].Value[j] = '\0';
 while (ch == ' ') ch = fgetc(symbolfile); /* ignore trailing blanks */
 }
}/* end readin */
```

```
/**********************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: symtrans()
*  DESCRIPTION:
*       This procedure searches a table for a sysbol. If the symbol
*       is found then the translation is written out, if not the
*       default is written out.
*
*  PASSED VARIABLES:
*               tablename : pointer to table
*               symbol : symbol to look for
*               lentbl : length of the table
*  RETURNS:
*  GLOBAL VARIBLES USED:
*               transfile
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*  FILES WRITTEN:
*               transfile
*  MODULES CALLED:
*  CALLING MODULES:
*               translate()
*
*  AUTHOR: Robert S. Hauser
*  HISTORY:
**********************************************************************/
int symtrans(tablename,symbol,lentbl) struct SYMBOL_ENTRY tablename[];
char symbol[];
int lentbl;
{
        int sindex,found;
        found = FALSE;
        for(sindex=0;(sindex<=lentbl)&&(found==FALSE);)
        {
                if (strcmp(tablename[sindex++].SubField,symbol)==0)

                        found = TRUE;
        }
        if (found==FALSE)
        {
                fprintf(transfile,"%s",tablename[0].Value);
                return(FALSE);
        }
        else
        {
                fprintf(transfile,"%s",tablename[sindex-1].Value);
                return(TRUE);
        }
}/* end symtrans */
```

```
/*****************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: literal()
*  DESCRIPTION:
*       This procedure determines if the symbol is a literal. If so
*       it prints out the value. If not it return FALSE.
*
*  PASSED VARIABLES: symbol
*  RETURNS:
*                TRUE : if symbol was a literal
*                FALSE : if symbol was not a literal
*  GLOBAL VARIBLES USED:
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*  FILES WRITTEN:
*  MODULES CALLED:
*  CALLING MODULES:
*                translate()
*
*  AUTHOR: Robert S. Hauser
*  HISTORY:
*******************************************************************/
int literal(symbol) char symbol[];
{
        int lindex;
        if (symbol[0] == '#')
        {
                for(lindex=1;lindex<(strlen(symbol));lindex++)
                        fprintf(transfile,"%c",symbol[lindex]);
                return(TRUE);
        }
        else
                return(FALSE);
}/* end literal */

/*******************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: itobs()
*  DESCRIPTION:
*       This procedure converts an integer to a binary string.
*
*  PASSED VARIABLES:
*                number : integer value
*                lab_b_length  : label field length
*  RETURNS:
*                 b_string : binary string
*  GLOBAL VARIBLES USED:
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*  FILES WRITTEN:
*  MODULES CALLED:
*  CALLING MODULES:
*                 strip()
*
*  AUTHQR: Robert S. Hauser
*  HISTORY:
*******************************************************************/
itobs(number,b_string,lab_b_length) int number;
char b_string[]; int lab_b_length;
{
        int index8;
        int index9;

        b_string[lab_b_length]='\0';
        for(index9=0,index8=lab_b_length-1;index8>=0;index8--,index9++)
        {
        if ((int)(number/pow((double)2,(double)index8)) >=1)
        {
                b_string[index9]='1';
                number = number - (int)pow((double)2,(double)index8);
        }
        else
        {
                b_string[index9]='0';
        }
        }
}/* end itobs */

/*******************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: convert()
*  DESCRIPTION:
*       This procedure return the integer value of the string input
*
*  PASSED VARIABLES:
*                 slice : binary string of 1/0
*  RETURNS:
*                 num: the integer value
*  GLOBAL VARIBLES USED:
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*  FILES WRITTEN:
*  MODULES CALLED:
*  CALLING MODULES:
*                 revtrans()
*
*  AUTHOR: Robert S. Hauser
*  HISTORY:
*******************************************************************/
long convert(slice)char slice[];
{
        long num;
        int i,j;
        num = 0;
        for(i=strlen(slice)-1,j=0;i>=0;i--,j++)
                if (slice[i]=='1')
                        num = num + (int)pow((double)2,(double)j);
        return(num);
}/* end convert */

/*******************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: revtrans()
*  DESCRIPTION:
*       This procedure takes a value and prints the field name
*
*  PASSED VARIABLES:
*               tablename : pointer to a table
*               symbol : value to translate
*               lentbl : table length
*  RETURNS:
*  GLOBAL VARIBLES USED:
*               revfile
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*               revfile
*  FILES WRITTEN:
*  MODULES CALLED:
*  CALLING MODULES:
*               reverse_comp()
*
*  AUTHOR: Robert S. Hauser
*  HISTORY:
*******************************************************************/
int revtrans(tablename,symbol,lentbl) struct SYMBOL_ENTRY tablename[];
char symbol[];
int lentbl;
{
        int rindex,found;
        found = FALSE;
        for(rindex=0;(rindex<=lentbl)&&(found==FALSE);)
        {
                if (strcmp(tablename[rindex++].Value,symbol)==0)
                        found = TRUE;
        }
        if (rindex==1)
                return(FALSE);
        else
        {
                fprintf(revfile,"%s ",tablename[rindex-1].SubField);
                return(TRUE);
        }
}/*end revtrans */

/*******************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: vhdl_out()
*  DESCRIPTION:
*       This procedure reads the translated file and output a VHDL
*       description of the ROM.
```

```
 *  PASSED VARIABLES: none
 *  RETURNS:
 *  GLOBAL VARIBLES USED:
 *              BitsInWord
 *              v_file
 *  GLOBAL VARIBLES CHANGED:
 *  FILES READ:
 *              trans_file : file of translated mcode
 *  FILES WRITTEN:
 *              vhdl_file : VHDL description
 *  MODULES CALLED:
 *  CALLING MODULES:
 *              main()
 *
 *  AUTHOR: Robert S. Hauser
 *  HISTORY:
*********************************************************************/
vhdl_out()
{
        FILE *data,*vhdl_file;
        int word_num;
        char xrom_word[BitsInWord+1];
        char eol[2];

        data = fopen(trans_file,"r");
        vhdl_file = fopen(v_file,"w");

        fprintf(vhdl_file,"package AN_XROM is\n");
        fprintf(vhdl_file,"   type WORD_%d is array (%d downto 0)
                                 of bit\;\n",BitsInWord,BitsInWord-1);
        fprintf(vhdl_file,"   type ROM_ARRAY is array (0 to %d)
                               of WORD_%d\;\n",line_num-1,BitsInWord);
        fprintf(vhdl_file,"   function GETWORD (WORD_NUMBER : integer)\n");
        fprintf(vhdl_file,"     return WORD_%d is\n",BitsInWord);
        fprintf(vhdl_file,"       variable XROM : ROM_ARRAY \;\n");
        fprintf(vhdl_file,"       variable RETURN_WORD : WORD_%d \;\n",
                                                          BitsInWord);
        fprintf(vhdl_file,"     begin\n");

        word_num = 0;
        while(fscanf(data,"%s %s",xrom_word,eol) != EOF)
        {
                fprintf(vhdl_file,"    XROM(%d) := B\"%s\";\n",
                                           word_num++,xrom_word);
        }

        fprintf(vhdl_file,"      RETURN_WORD := XROM(WORD_NUMBER);\n");
        fprintf(vhdl_file,"      return (RETURN_WORD);\n");
        fprintf(vhdl_file,"    end GETWORD;\n");
        fprintf(vhdl_file,"  end AN_XROM\;\n");

} /*end vhdl_out*/

/***********************************************************************
 *  DATE: 1 DEC 1987
 *  Version: 1.0
 *  PROCEDURE: strip()
 *  DESCRIPTION:
 *      This procedure reads the microcode and strips off the
*       delimiters and converts the labels into integer valued
*       binary strings. It also makes a listing file and check
*       for the exsistence of the input file.
*
*   PASSED VARIABLES: none
*   RETURNS:
*   GLOBAL VARIBLES USED:
*               strip_file
*               l_file
*               i_file
*               line
*               line_num
*               line_no_lab
*               line_numl
*               END2
*               EOL
*               Labeltbl
*   GLOBAL VARIBLES CHANGED:
*               line
*               line_num
*               line_no_lab
*               line_numl
*               Labeltbl
*   FILES READ:
*               infile
*   FILES WRITTEN:
*               listingfile
*               stripped
*   MODULES CALLED:
*   CALLING MODULES:
*               main()
*
*   AUTHOR: Robert S. Hauser
*   HISTORY:
************************************************************************/
strip()
{
        stripped = fopen(strip_file,"w");
        if ((infile = fopen(i_file,"r")) == NULL)
        {
                printf("\nFile %s could not be found.\n\n",i_file);
                exit(1);
        }
        listingfile = fopen(l_file,"w");

        fscanf(infile,"%[^;]",line);
        strcat(line,EOL2);
        while(strcmp(END2,line)!=0){
                if (index(line,':') != NULL)
                {
                sscanf(line,"%[^:]: %[^;]",label,line_no_lab);
                strcpy(Labeltbl[index_to_labels].SubField,label);
                itobs(line_num,line_numl,lab_b_length);
                strcpy(Labeltbl[index_to_labels++].Value,line_numl);
                LABELaddr[label_num] = line_num;
                strcpy(LABELarray[label_num++],label);
                }
                if (index(line,':') != NULL)
                {
                strcat(label,COL);
                fprintf(listingfile,"%0.6d  %-15s%s;\n",line_num++,
                                                    label,line_no_lab);
                fprintf(stripped,"%s %s\n",line_no_lab,EOL);
                }
                else
                {
                fprintf(listingfile,"%0.6d
                                             %s\n",line_num++,line);
                        sscanf(line,"%[^;]",line);
                        fprintf(stripped,"%s %s\n",line,EOL);
                }
                fscanf(infile,"%[^\n]\n",line);
                fscanf(infile,"%[^;]\n",line);
                strcat(line,EOL2);
        }
        fprintf(stripped,"%s\n",END);
        fprintf(listingfile,"%s\n",END2);

        fclose(listingfile);
        fclose(stripped);
}/*end strip */

/*******************************************************************
*  DATE: 1 DEC 1987
*  Version: 1.0
*  PROCEDURE: makexromaddrs()
*  DESCRIPTION:
*       This procedure reads the translated file and build the
*       XROM compiler input file.
*
*  PASSED VARIABLES: none
*  RETURNS:
*  GLOBAL VARIBLES USED:
*               trans_file
*               BitsInWord
*  GLOBAL VARIBLES CHANGED:
*  FILES READ:
*               trans_file
*  FILES WRITTEN:
*               xromaddrs
*  MODULES CALLED:
*               convert()
*  CALLING MODULES:
*               main()
*
*  AUTHOR: Robert S. Hauser
*  HISTORY:
*******************************************************************/
make_xromaddrs()
{
        transfile = fopen(trans_file,"r");
        xromaddrs = fopen(a_file,"w");


        xrom_length =(int)( BitsInWord/4);

        while(fscanf(transfile,"%[^+]+ \n",line) != EOF)
        {
                for(base=0;base<(4*xrom_length);base = base+xrom_length)
                {
                        for(indexa=0;indexa<xrom_length;indexa++)
                        {
                                slice[indexa]=line[base+indexa];
                        }
                        slice[indexa+base]='\0';
                        num = convert(slice);
                        fprintf(xromaddrs,"%d\n",num);
                }
        }
        fclose(xromaddrs);
        fclose(transfile);
}/* end make_xromaddrs */

#include "assem.tailored"
```

APPENDIX F

Microcode Translation File

=

BR_SEL ALU LOAD_FD REG REG REG SPEC_FUNCT WFTOP PFADONE NXT_ADDR # .

BR_SEL	000000000
RET	000000001
CALL	000000010
JMP	000000011
CALLCR	000010110
CALLCE	000100110
CALLnZ	001001110
JGE	000111111
JnZ	001001111
JZ	001000111
JnOP	001011111
Jn4DN	001101111
JnWD	001111111
JnECo1	010001111
JnECo2	010011111
JnECo3	010101111
Jn11	010111111
Jn12	011001111
Jn13	011011111
Jn21	011101111
Jn22	011111111
Jn23	100001111
Jn31	100011111
Jn32	100101111
Jn33	100111111
JnEU	101001111
JnEU1	101011111
JnEU2	101101111
JnEU3	101111111
JnEC	110001111
JnEC1	110011111
JnEC2	110101111
JnEC3	110111111
JnPE	111001111
JnPE1	111011111
JnPE2	111101111
JnPE3	111111111
ALU	0000

COMP	0001
AND	0010
XOR	0011
OR	0100
MOV	0101
S$\overline{\text{C}}$Y	0110
RCY	0111
INC	1000
DEC	1001
ADDC	1010
ADD	1011
SUB	1101
SUBB	1110
CMP	1111

LOAD_FD	0
LOAD	1

REG	00000
ECCC1	00001
ECCC2	00010
ECCC3	00011
ECCU1	00100
ECCU2	00101
ECCU3	00110
PE1	00111
PE2	01000
PE3	01001
WD11	01010
WD12	01011
WD13	01100
WD21	01101
WD22	01110
WD23	01111
WD31	10000
WD32	10001
WD33	10010
TSR	10011
PSR	10100
HE2	10101
PS1	10110
PS2	10111
PS3	11000
ELR	11001
NCR	11010
CCR	11011
TEMP	11100
TOUT	11101

SPEC_FUNCT	0000000000000
Flip	1000000000000
LoadInit	0100000000000
FlipLoadInit	1100000000000
LoadScale	0010000000000
LoadPSi	0001000000000
ShiftTSR	0000100000000
LPSiSTSR	0001100000000
ShiftPSR	0000010000000
LoadELR	0000001000000
HostCntl	0000000100000
LDStateR1	0000000010010
LDStateR2	0000000001010
LDStateR3	0000000000110
LDState	0000000000010
DriveScale	0000000000001
WFTOP	0
WFTop	1
PFADONE	0
PFAdone	1
NXT_ADDR	0000000000000000

APPENDIX G

Microcode Word Format

BITS

0-8 BR_SEL(9) - Branch Select Field

0-4 CMS(5) - Conditional Mux Select
0 None
1 Reconfigure
2 Error
3 Negative
4 Zero
5 PFAoperate
6 4DONE
7 Watch Dog Error (WD_ERR)
8 Error in Col 1 (ErrCol1)
9 Error in Col 2 (ErrCol2)
10 Error in Col 2 (ErrCol2)
11 Watch Dog Error 11 (WD11)
12 Watch Dog Error 12 (WD12)
13 Watch Dog Error 13 (WD13)
14 Watch Dog Error 21 (WD21)
15 Watch Dog Error 22 (WD22)
16 Watch Dog Error 23 (WD23)
17 Watch Dog Error 31 (WD31)
18 Watch Dog Error 32 (WD32)
19 Watch Dog Error 33 (WD33)
20 EU
21 ECCU1
22 ECCU2
23 ECCU3
24 EC
25 ECCC1
26 ECCC2
27 ECCC3
28 PE
29 PE1
30 PE2
31 PE3

5 BR_ON(1) - Branch On
0 Positive Logic
1 Negative Logic

6-8 NAF(3) - Next Address Field
0 Continue
1 Return
2 Call
3 Unconditional Branch
4 Conditional Datapath Load

5 Conditional Return
6 Conditional Call
7 Conditional Branch

9-12 ALU(4) - ALU Function Select
0 nop
1 C = A'
2 C ← A and B
3 C ← A xor B
4 C ← A or B
5 C ← A (mov)
6 Set Carry Flag
7 Reset Carry Flag
8 C ← A + 1
9 C ← A − 1
10 C ← A + B + cy
11 C ← A + B
12 not defined
13 C ← A− B
14 C ← A − B − br
15 A − B

13 LOAD_FIELD(1) - Load Next Address Field to C Bus
0 No Load
1 Load

14-18 ABUS_SEL(5) - A Bus Select
0 none
1-29 Registers 1-19
30-31 not defined

19-23 BBUS_SEL(5) - B Bus Select
0 none
1-29 Registers 1-19
30-31 not defined

24-28 CBUS_SEL(5) - C Bus Select
0 none
1-29 Registers 1-19
30-31 not defined

29-41 SPEC_FUNCT(13) - Special Functions

bit	
0(MSB)	Flip Memories
1	Load Initial Scale Factors
2	Load Scale Factors from WFTs
3	Load Problem Status Registers
4	Shift Temp Scale Register
5	Shift Permanent Scale Register
6	Load Error Location Register
7	Host Control
8	Load State into Row 1
9	Load State into Row 2
10	Load State into Row 3
11	Load State
12	Drive Scale Factors

42 WFTOP(1) - WFT Operate

0: WFTop'
1: WFTop

43 PFADONE(1) - PFA Done Signal
0: PFAdone'
1: PFAdone

44-59 NXT_ADDR9(16) - Next Address Field & Literal for Datapath

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Uclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/87D-5	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering; 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson, AFB, CH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Air Force Office of Scientific Research; 8b. OFFICE SYMBOL (If applicable) AFOSR/XP	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Bolling, AFB, Washington D.C. 20332	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

See Box 19

12. PERSONAL AUTHOR(S)

Robert S. Hauser, B.S. Computer Engineering, 2Lt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
M.S. Thesis	FROM ________ TO ______	December 1987	154

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

VLSI, Discrete Fourier Transform, Winograd Fourier Transform

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: Design and Implementation of a VLSI Prime Factor Algorithm Processor

Thesis Chairman: Richard W. Linderman, Captain, USAF
Assistant Professor of Electrical and Computer Engineering

Approved for public release IAW AFR 190-17.
[illegible] 14 Mar 88
[illegible] Development
[illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard W. Linderman, Captain, USAF	22b. TELEPHONE (Include Area Code) 513-255-3576; 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 Previous editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

(block 18 continued)
Application Specific Processor, Fault Tolerance

(block 19 continued)

Abstract

High-speed digital signal processing has a wide range of applications including, radar, sonar, image processing, and target acquisition. The calculation of the Discrete Fourier Transform (DFT) used in these applications has long been a significant bottleneck for high-speed processing. Previous AFIT students have adopted a Prime Factor Algorithm (PFA) method using Winograd Fourier Transform (WFT) processors. Three WFT processors are pipelined into a system capable of computing a 4080-point DFT on complex data approximately every 120 microseconds when operating with a 70 MHz clock.

This thesis effort addressed the design and implementation of PFA controller chip and interconnecting memory modules between the WFT processors. The PFA controller is an application specific processor to control the flow of information in the pipeline, interface to the WFT processors, monitor pipeline status, and take corrective action in the presence of faults. The interconnecting memory modules buffer the data coming out of a WFT processor and going into another allowing concurrent reading and writing.

The PFA controller chip was designed, simulated, and submitted for fabrication through MOSIS. Twenty-eight 16-bit registers store the pipeline information. An arithmetic/logic unit (ALU) computes data transformations. A read only memory stores the microcode. A control sequencer sequences through the proper code segments. Finally, special circuitry interprets the fault information and reconfigures the pipeline.

This thesis effort included writing a microcode assembler to to raise the user interface to the AFIT-XROM silicon compiler. Raising the user's level of abstraction to mnemonic microcode, while still providing an error free path to silicon layout, reduces chances for error in the microcode specification. A generic microcode assembler tool was created as an extension for use with other application specific processors. This tools generates a microcode assembler from a word format and a translation file. The assembler will output a file compatible with the XROM compiler, a VHDL description of the XROM, a listing file, a reference file, and a reverse assembly. This tool was tested on two other AFIT theses and a computer architecture class.

A prototype memory chip was designed and fabricated in 3 micron CMOS through MOSIS to test the 1-transistor memory cell, the wordline selectors, and the sense amplifiers. Simulations predict an access time of 10ns. A larger memory was designed, simulated, and submitted for fabrication through MOSIS. It contains storage for 272 words of 32 bits each. It is dual ported and permits concurrent reading and writing of 24 bit data. The memory also includes error control circuitry for single error correction and double error detection.