

Copyright © 1984, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A FLOATING POINT COPROCESSOR FOR THE  
BUTTERFLY MULTIPROCESSOR SYSTEM

by

D. Y. Cheng

Memorandum No. UCB/ERL M84/55

6 July 1984

(cover)

A FLOATING POINT COPROCESSOR FOR THE  
BUTTERFLY MULTIPROCESSOR SYSTEM

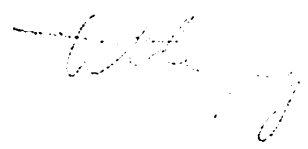
by

D. Y. Cheng

Memorandum No. UCB/ERL M84/55

6 July 1984

ELECTRONICS RESEARCH LABORATORY  
College of Engineering  
University of California, Berkeley  
94720

A handwritten signature in dark ink, appearing to be 'D. Y. Cheng', is located in the lower right quadrant of the page.

## ACKNOWLEDGEMENTS

The author would like to thank Professor A. R. Newton for the support, encouragement, and guidance he has given during this project.

She would like to thank Jeffrey T. Deutsch for his ideas and suggestions for this project, for his help in changing the Crysalis operating system and the Butterfly C compiler, and for being joint designer of the second generation floating point coprocessor for the Butterfly.

She would like to thank Glenn Simpson, Ward Harriman, John Rokosz, especially Walter Milliken of Bolt Beranek and Newman Inc. for their suggestions and help during this project.

She would like to thank Bolt Beranek and Newman Inc. for their generous support of this project, and thank Randy Rettberg, John Goodhue, and Frank Heart for providing her a pleasant working and living environment while at BBN.

This project was supported in part by the Semiconductor Research Corporation and their support is gratefully acknowledged.

Finally, the author would like to thank her parents for the chance they gave to her to continue her education in the United States.

## TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION .....	2
CHAPTER 2: THE BUTTERFLY MULTIPROCESSOR .....	4
2.1 Introduction .....	4
2.2 Architecture Of The Butterfly .....	4
2.2.1 Overview .....	4
2.2.2 The MC68000 .....	5
2.2.3 The Processor Node Controller .....	6
2.2.4 Memory Management And The Memory Management Unit .....	8
2.2.5 The I/O Co-processor .....	13
2.2.6 The Switch Interface And The Interconnection Network .....	13
2.3 Butterfly Operating System .....	19
2.3.1 Object Management System .....	19
2.3.2 Processes .....	22
2.3.3 Events .....	23
2.3.4 Dual Queues .....	24
2.3.5 Buffer Management .....	24
CHAPTER 3: THE PROTOTYPE FLOATING POINT CO-PROCESSOR .....	26
3.1 Introduction .....	26
3.2 The Interface To The Processor Node .....	26
3.3 Tradeoffs In The Architecture Design .....	28
3.3.1 The Memory To Memory Architecture .....	29
3.3.2 The Stack Pointer Architecture .....	29
3.3.3 Frame Pointer Architecture .....	31
3.3.4 On Board Stack Architecture .....	33

3.4 The Prototype .....	34
3.4.1 The Hardware .....	34
3.4.2 Performance .....	35
CHAPTER 4: PROPOSALS FOR THE NEXT GENERATION CO-PROCESSOR .....	37
4.1 Introduction .....	37
4.2 Tradeoffs .....	38
4.3 Hardware Support For Evaluating Elementary Functions .....	39
4.4 The FPP2 .....	42
4.5 The FPP3 .....	44
4.5.1 Introduction .....	44
4.5.2 The Control .....	44
4.5.3 The ALU .....	46
4.5.4 The Bus Interface And Dual Static RAM .....	47
4.5.5 The Table ROM .....	48
4.5.6 The Diagnostic System .....	48
4.5.7 The Micro Instruction Format .....	49
CHAPTER 5: SUMMARY .....	50
APPENDIX 1: Schematics of the FPP1 .....	A1.1
APPENDIX 2: PNC Microcode for Floating Point .....	A.2.1
APPENDIX 3: Prototype Microcode .....	A3.1
APPENDIX 4: Modified LA11 .....	A4.1
APPENDIX 5: Schematics For The FPP3 .....	A5.1
APPENDIX 6: exp.c and log.c .....	A6.1

## CHAPTER 1

### INTRODUCTION

For the past several years, the complexity of the VLSI chip designs has been increasing rapidly. Many current circuits would be impossible to design without computer aided design (CAD) tools. This trend is likely to continue as a result of the rapid development of the VLSI technology.

Circuit simulation is an important part of computer aided design. It allows users to find many design flaws without building the chip. However, there is a large distance between the computing power required to simulate large circuits and that which current computers can provide. Although there have been attempts to create circuit simulation programs which can exploit the vector processing capabilities of machines such as the Cray-1 [VLA82], the result has only been approximately an order of magnitude improvement over scalar performance, for practical circuits. One reason for this is that the extensive sparse matrix operations in circuit simulation cause a "gather-scatter" problem [CAL79], where the time needed to read data from the sparse matrix and write it back limits the machine to a small percentage of its peak performance.

In general, most CAD problems are computationally intensive and contain high degree of concurrency. More and more CAD programs need more computing power than today's sequential machine can provide. A multiprocessor system with a reasonably high bandwidth and low latency, such as the BBN Butterfly [RET79], gives a nearly linear increase in performance for a small number of processors [DEU84].

The Butterfly is the first multiprocessor system with a reasonable software environment available commercially. The current version of the Butterfly has sixteen processors in the system. A 128 processor version is now under design. The high bandwidth, low latency, and

easily-expanded interconnection network of the Butterfly makes it suitable as a first generation CAD machine. Many architectural and algorithmic ideas can be evaluated by experimenting with them on multiprocessor hardware. This can lead to better insight about what the best architectures and algorithms for computationally-intensive scientific applications such as CAD are likely to be. However, there is no hardware support for the floating-point operations on the Butterfly system. Efficient floating-point is necessary for scientific applications and this requirement motivated the project of building a floating-point co-processor for the Butterfly.

In Chapter 2 of this report the architecture and the operating system of the BBN Butterfly Multiprocessor System is introduced. In Chapter 3 the prototype of the first generation floating-point co-processor FPP1 is described. Comparisons among the memory to memory architecture, the architecture using the stack pointer, the architecture using the frame pointer, and the on-board stack architecture are described, and tradeoffs in designing the FPP1 are presented. The FPP1 is microprogrammable. Its speed is 2-4 times the floating-point operation than the fast Motorola floating-point software on the 68000 employed as the main general purpose processor in the Butterfly machine. By changing the microcode to the architecture in which the 68000 passes the frame pointer to the processor node controller on a Butterfly node at the beginning of a floating-point operation, another 2.6 times speed up can be obtained.

In Chapter 4 two new architectures for the second generation floating-point co-processor are proposed, the FPP2 and the FPP3. The FPP2 is a low cost but relatively low performance version of the co-processor. Its hardware supports fast evaluation of elementary functions. The FPP3 provides the highest possible floating-point speed in the Butterfly system. Its WEITEK chip set [WEI83] gives 5 million floating-point operations per second performance if 4K Static RAM is used, 2.5 million floating-point operations per second performance if 16K static RAM is used. An optional NS16081 chip provides 64-bit precision.



## CHAPTER 2

### THE BUTTERFLY MULTIPROCESSOR

#### 2.1. Introduction

In this chapter, The main features of the Butterfly architecture and the Chrysalis operating system are described. Only those features that are the basis of the floating-point co-processor are described in detail; others are reviewed briefly for reference. Additional information can be found in the Butterfly Quarterly Reports from Bolt Beranek and Newman (BBN) [RET79].

#### 2.2. Architecture Of The Butterfly

**2.2.1. Overview** The Butterfly is a 68000-based tightly coupled multiprocessor system built by BBN as part of the DARPA voice-funnel project, designed for interfacing high speed digitized voice signals to a packet-switched communications network.

The Butterfly multiprocessors can be configured with from 1 to 128 processor nodes. All processor nodes communicate through an interconnection network as shown in Figure 2.1. This network is topologically equivalent to the Omega network [LAW75].

Each Butterfly processor node is itself a multiprocessor. It contains the following functional units:

- A MC68000 microprocessor
- An AMD2901-based processor node controller.
- A memory management unit which controls the segment-based virtual memory system of the Butterfly.
- 256K bytes of on board dynamic memory.
- Up to four 2901-based I/O co-processors.

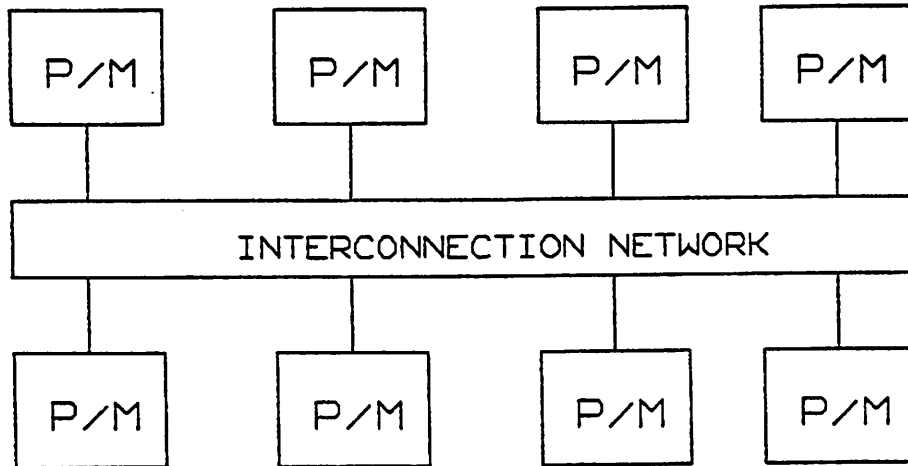


Figure 2.1 - Butterfly Multiprocess System

---

- Two special purpose finite state machines for interfacing the switch.
- 4K bytes of Erasable Programmable ROM for bootstrapping and diagnosis

A block diagram of these functional units and their interconnections is shown in Figure 2.2. The rest of this section describes some of these functional units in detail.

**2.2.2. The MC68000** The Motorola MC68000 is a 16-bit microprocessor, with 32-bit internal data and address registers, and a 16 bit ALU. The MC68000 communicates with the rest of the functional units in the processor node through a 24-bit CPU address bus (CPUA bus), a 16-bit CPU data bus (CPUD bus) and the CPU control bus. The MC68000 takes 4 clock cycles to access memory. If CPU does not receive an acknowledge signal from the memory before the fourth cycle, a wait state is automatically inserted. In the Butterfly multiprocessor system, the memory access is more complicated than a uniprocessor system. The fact that many decisions have to be made during the fourth cycle lengthens the local memory access to five cycles.

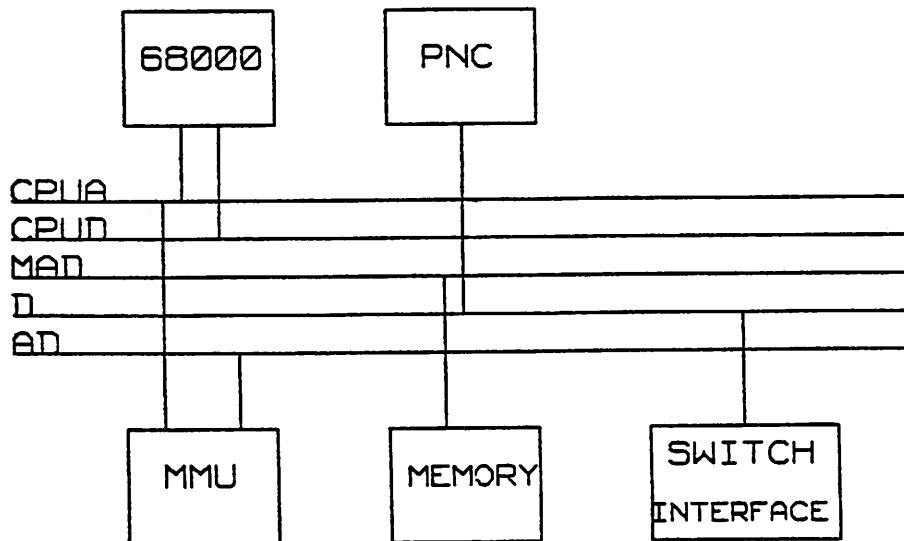


Figure 2.2 - The Butterfly Processor Node

---

When the CPU receives a bus error signal together with a halt signal, it will put the CPUTA bus and the CPUD bus in the high impedance state and remain halted as long as the halt signal remains asserted; after the halt signal is removed it reruns the bus cycle.

**2.2.3. The Processor Node Controller** The processor node controller (PNC) is the key component in a processor node. The PNC handles the following functions:

- Local memory accesses and local memory refresh
- Switch interface and remote memory access
- I/O co-processor interface
- Time-of-day clock and timer handling
- Multiprocessor system synchronization and communication primitives
- Interrupt controlling

- Initializing the memory management unit registers

The PNC is a 16-bit AMD 2901-based microprogrammed machine. As shown in Figure 2.3, it contains a 16-bit 2901 ALU, a 2911 microprogram sequencer, a 1K x64 bit micro control ROM and a PLA which generates the address of a microinterrupt service routine.

The way 68000 is serviced by the PNC is by the use of microinterrupts. In the local memory subspace 0, there are some "magic" locations which will be described in detail in the next section. Whenever 68000 reads from-or writes into-these locations, CPUA bits 15 through 8 and some control information cause the PLA to generate the address of a proper microinterrupt service routine. The PNC then executes the desired microcode. The 68000 waits while the PNC is performing the desired function. After the PNC finishes the function, it sends an acknowledgement to the 68000. The 68000 then ends the memory cycle and

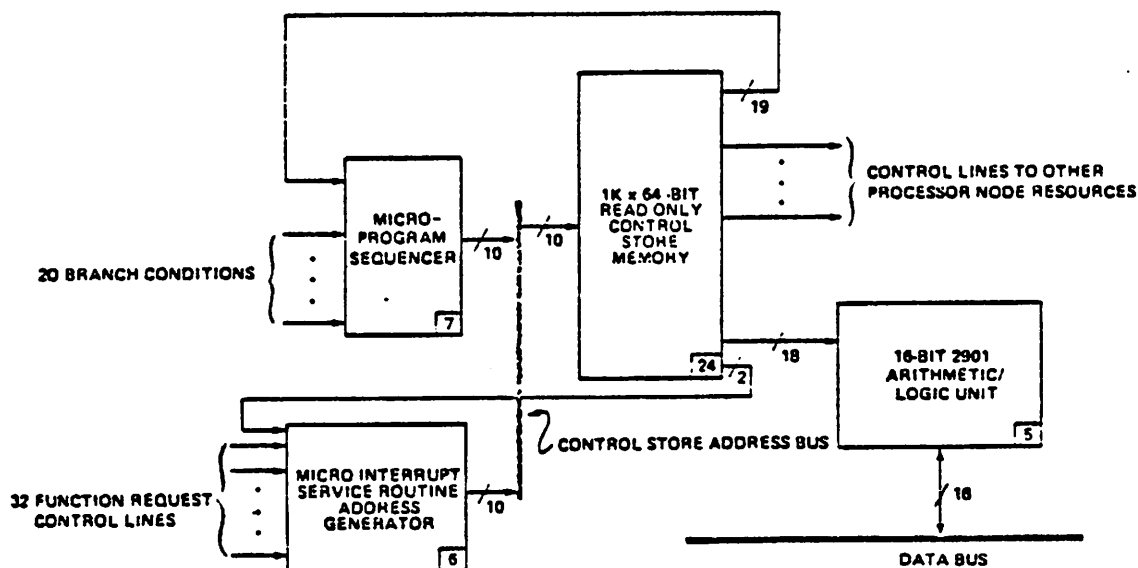


Figure 2.3 - The Processor Node Controller



proceeds [RET80].

2.2.4. Memory Management And The Memory Management Unit The Butterfly provides segmentation-based virtual memory. The virtual memory system in the Butterfly

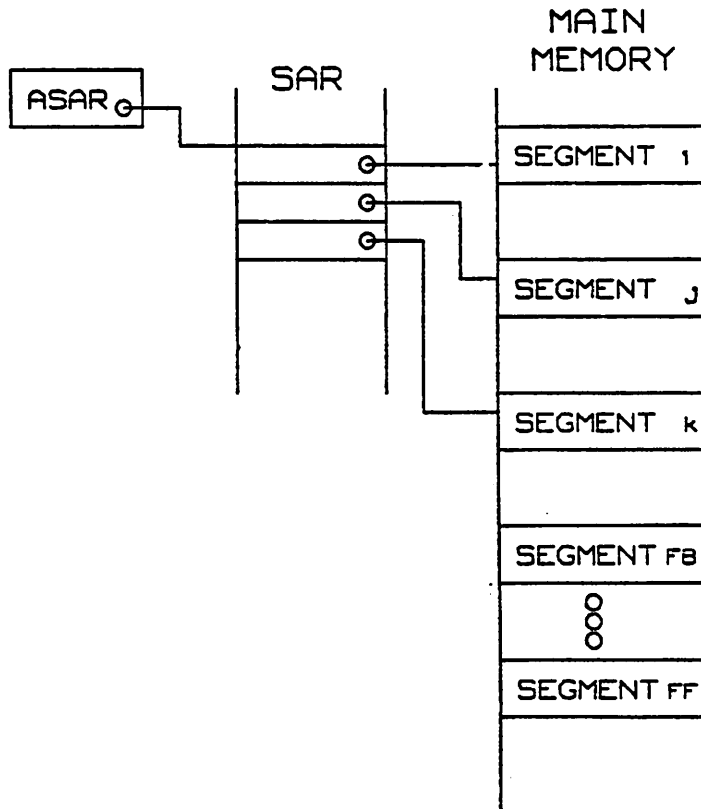


Figure 2.4 - The Address Space For A Process

provides each process with up to 256 memory segments. Each segment can be 256 to 64K bytes long which may either be in the local memory or a remote memory. Segments F8 through FF are shared among all processes in the system. Figure 2.4 shows the virtual address space of a process. The ASAR and SAR will be explained later in this section. The 32 bit virtual address format is shown in Figure 2.5. The upper 8 bits are unused. The next 8 bits are the segment number. The last 16 bits are the segment offset.

The physical address format is shown in Figure 2.5. This 32-bit physical address space is the concatenation of the physical address spaces of all the processor nodes in the system. The highest 8 bits are thus the processor node number. The physical space of each processor node is divided into four subspaces. Two bit are used to select the subspace. The last 22 bits are the subspace offset.

Each subspace of a processor node has a special purpose.

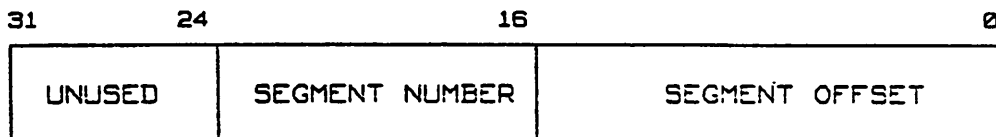


Figure 2.5 - The Virtual Address Format

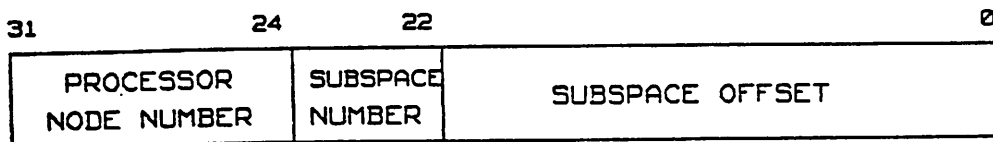


Figure 2.6 - The Physical Address Format

---

- Subspace 0 contains EPROM, Segment Attribute Registers, 68000 interrupt vectors, interrupt handling routines, operating system kernel, library routines and all PNC control registers which are used to generate required microinterrupts.
- Subspace 1 contains the I/O control registers on the I/O boards. It is divided into four equal parts, one for each possible I/O board.
- Subspace 2 contains the local memory.
- Subspace 3 indicates that the access should be made through the switch. This is mainly for remote memory access. But it is also possible to map the local memory address in this subspace.

The current Butterfly system does not have a file system or secondary storage. This will be changed in a later version of the system.

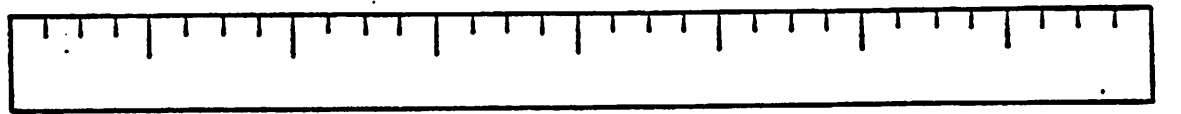
The memory management unit (MMU) provides hardware support for memory relocation, memory protection, and address translation. The architecture of the MMU is shown in Figure 2.7. There is an Address Space Attribute Register (ASAR) which points to the beginning of the Segment Attribute Register (SAR) block of the currently executing process. Figure 2.8 shows the format of the ASAR and SAR. The highest two bits of the ASAR are the control and mode bits. The size code indicates how many segments are in the virtual address space of the current process. The least significant nine bits are the pointer to the first SAR of the current process. Upon context switching, the operating system only needs to save the value of the ASAR and loads it with a new value. The 1K ×16 segment attribute RAMs contain 512 SARs each of which points to a segment in the virtual memory. Each SAR also contains the protection information of the segment. An 8 bit adder is used in translating the virtual address to physical address. A PLA detects the protection violation of a memory access.

The process of translation from the virtual address to the physical address is picturized in Figure 2.9. Bits 8 to 0 of the ASAR is ored with the segment number in the virtual address to get the required SAR. Bits 31 to 24 of the physical address come directly from the processor node number field of the SAR; Bit 23 to 16 of the physical address is formed by the subspace number field and the least significant bits 5 to 0 of the SAR. The page offset in the



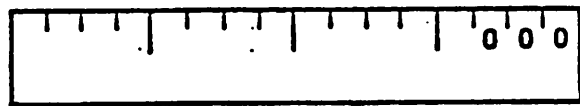


**SEGMENT ATTRIBUTE REGISTER**



<b>Processor Node Number</b>		<b>S</b>	<b>Page Offset</b>		<b>S S</b>	<b>Bits 19:16 of the Physical Address</b>
<b>Access Code</b>		<b>Segment Size</b>		<b>Subspace</b>		
0	R__r_x	0	0	8	16	0 Subspace Zero
2	R_Xr_x	1	1	9	24	4 I/O
4	RWXrwx	2	2	A	32	8 Local Memory
6	RW_rw_	3	3	B	48	C Remote Memory
8	R__r__	4	4	C	64	
A	RW____	5	6	D	96	
C	R_____	6	8	E	128	
E	RW_r__	7	12	F	256	

**ADDRESS SPACE ATTRIBUTE REGISTER**



<b>Kernel Inhibit</b>	<b>S S</b>	<b>SAR Pointer</b>	
	<b>Size Code</b>		
	0	8	
	2	16	
	4	32	
	6	64	
	8	128	
	A	256	

Figure 2.8 - The SAR And The ASAR

SAR is added with bits 15 to 8 of the virtual address to form the bits 16 to 8 of the physical address. The physical address bits 7 to 0 come directly from the bits 7 to 0 of the virtual address.

The PNC must be able to access the MMU in order to perform some functions such as block transfer through the switch. The Address Register is there for this purpose. To do this, the PNC first asks the 68000 to back off the CPUA bus and the CPU control bus. The PNC then loads the SAR address into both Address Register and the ASAR. The MMU then proceeds to perform relocation and protection violation detection as usual [RET82].

**2.2.5. The I/O Co-processor** Up to four I/O processors can be configured in each processor node. Each I/O co-processor supports four synchronous and four asynchronous channels with a data rate of four million bits per second. As shown in Figure 2.10, the processor node communicates with the I/O co-processors through an I/O bus called BIOLINK. Figure 2.11 shows that an I/O co-processor consists of four parts interconnected by a 16-bit internal bus.

- An I/O controller which provides the means for data manipulations.
- An interface to the BIOLINK, in which a finite state machine deals with the synchronous protocol between the I/O co-processor and the processor node.
- Four Signetics 2661 Enhanced Programmable Communications interface chips with RS232C line driver/receivers,
- Four Signetics 2652 Multiprotocol Communications Controllers chips with RS422 line driver/receivers.

The I/O controller is composed of four parts: An AMD2901-based microprocessor, a 512 word 64-bit read-only control store in which the microprogram resides, a 2911-based microprogram sequencer, and a 1K ×16 scratch-pad memory which provides control variable and data storage for all the channels [RET82].

**2.2.6. The Switch Interface And The Interconnection Network** The interface of a processor node to the interconnection network is composed of two parts. One is the receiver and the other is the transmitter. Figure 2.12 shows the block diagram of the receiver. The

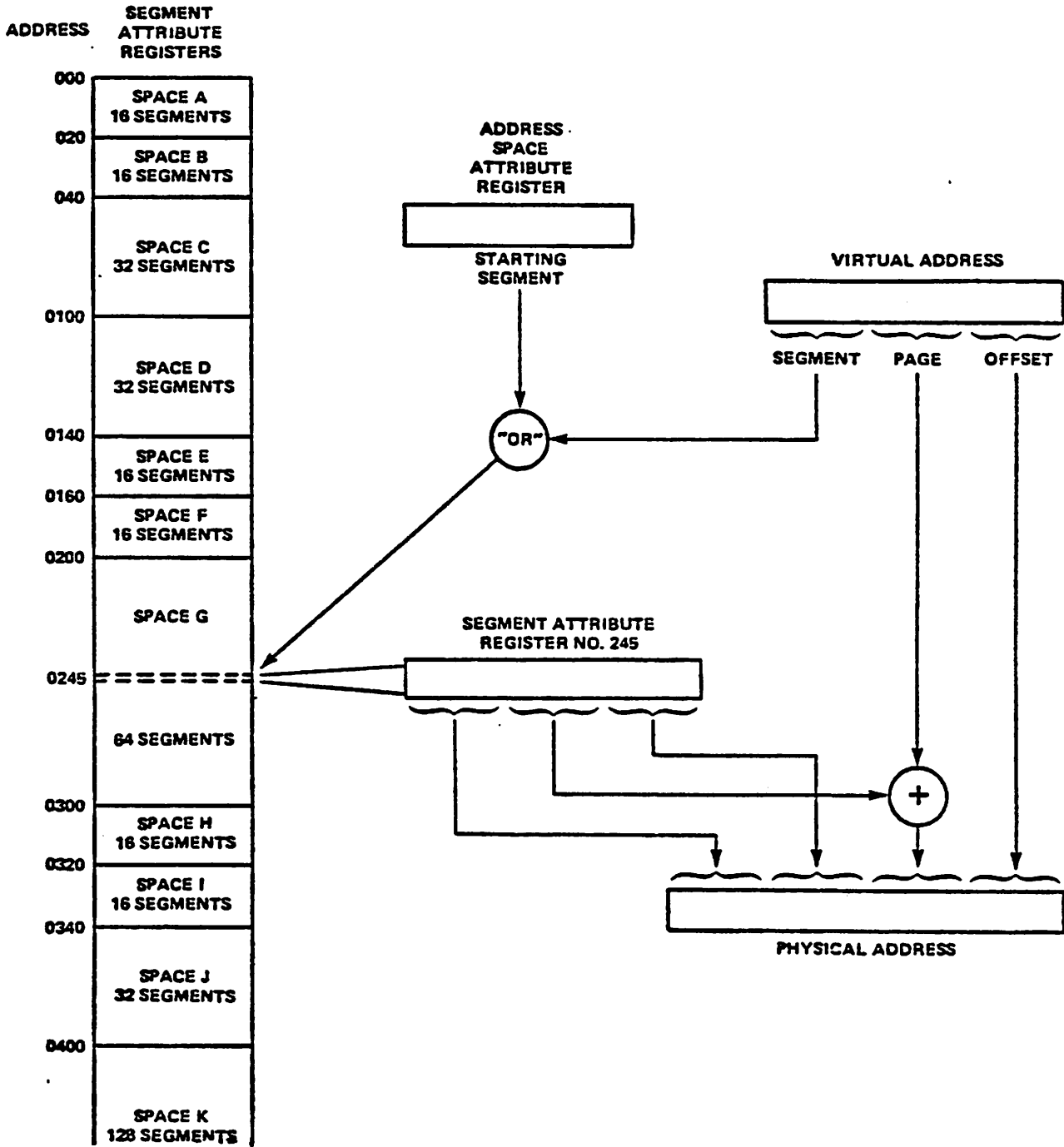


Figure 2.9 - Address Translation

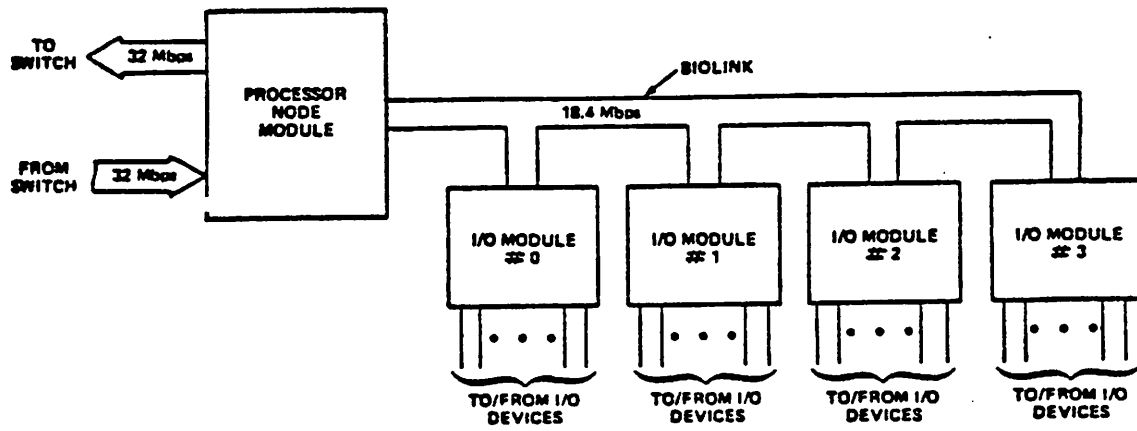


Figure 2.10 - The BIOLINK

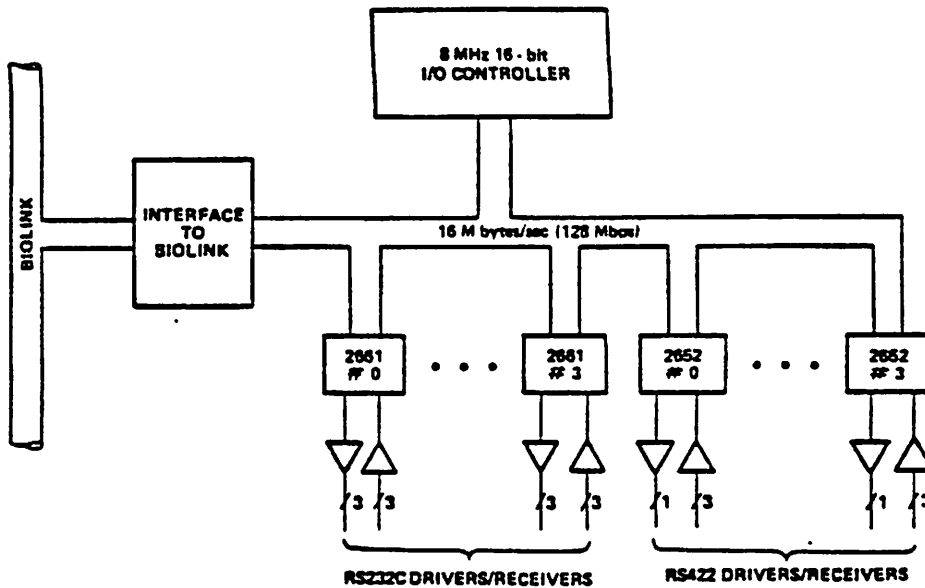


Figure 2.11 - The I/O Co-processor

receiver has a ROM/PLA controlled finite state machine, a checksum generator, a 16-word dual-port memory with 16-bit per word, and a set of ECL to TTL line drivers and receivers. The finite state machine controls the interaction between the processor node and the switch. It also controls the registers and line drivers and line receivers. The dual-port memory supports two input buffers. Two input buffers are needed in order to avoid deadlock situations. The transmitter shown in Figure 2.13 has a symmetrical architecture to the receiver. Although there is no deadlock problem in the transmitter side, it also provides two output buffers.

The interconnection network is implemented by a set of switches and wires. Each switch is a 4x4 crossbar plus the routing logic. The interswitch wiring is similar to the wiring rule of the perfect shuffle network. Figure 2.14 shows 16x16 such a network. For N processor nodes, the number of switches needed is

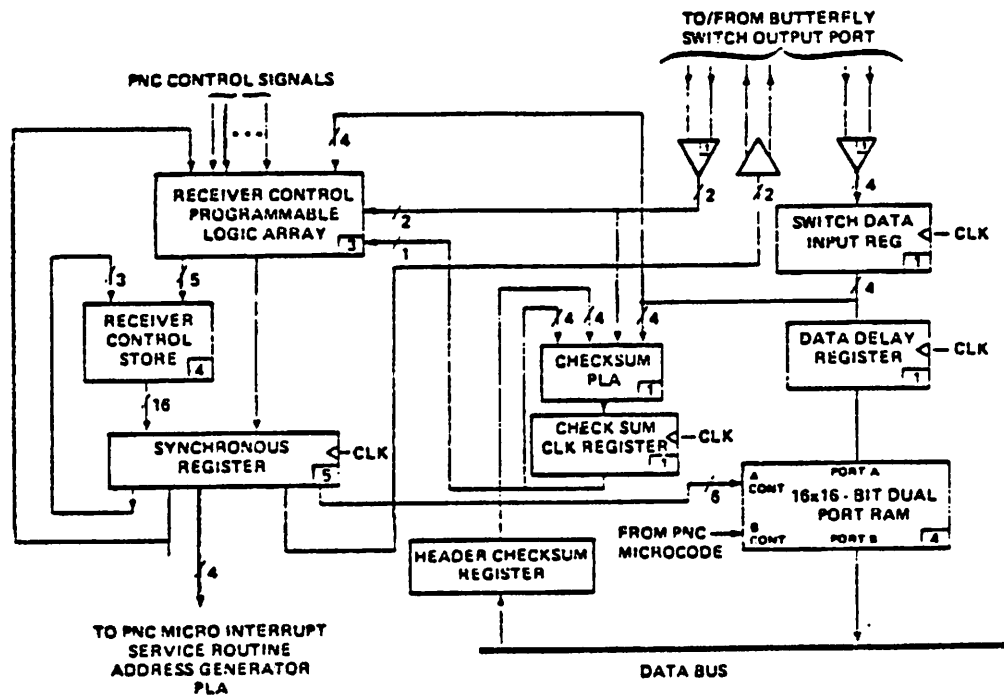


Figure 2.12 - The Receiver

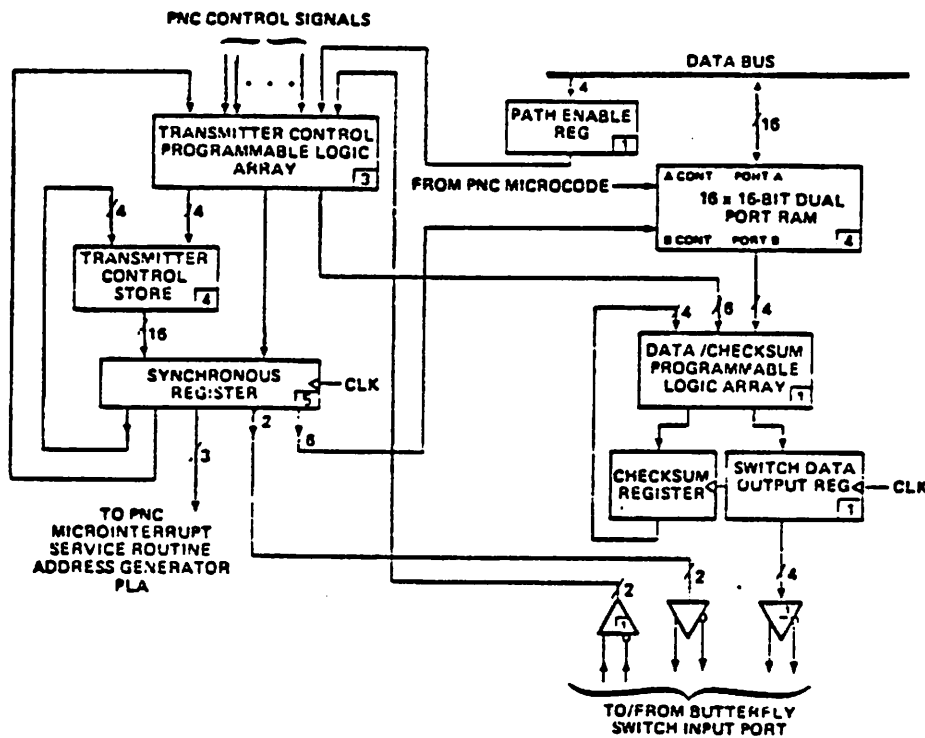


Figure 2.13 - The Transmitter

$$\frac{N}{4} \times \log_4 N$$

This network has several interesting characteristics:

- (1) Uniform latency across the system,  
The latency from any node to any other node is :  $\log_4 N$
- (2) The latency increases very slowly when the system scales up.
- (3) The bandwidth of the network is linearly proportional to the number of the switch chips. No changes in the switch design are required to expand the network.

These characteristics make this kind of interconnection network very suitable for a multiprocessor system with large number of processors (more than hundreds) in which low latency and high throughput are essential [RET79a].

### 2.3. Butterfly Operating System

Chrysalis, the Butterfly operating system, is a process-based real-time operating system. The system services are provided by a collection of system calls. All system resources are conceptualized as objects and are managed by the Object Management System. The synchronization, scheduling and interprocess communication are implemented through events, locks and queues. The storage is allocated using a Buddy algorithm [KNU73], and is reclaimed by the Chrysalis garbage collector. The garbage collector scans all objects every few seconds. If the owner field of an object claims that its owner has been deleted, this object will be deleted by the garbage collector. In the following sections some important components of the Chrysalis operating system are described.

**2.3.1. Object Management System** The Object Management System is designed to manage the segmentation-based virtual memory in a tightly-coupled multiprocessor system. It provides the services for memory allocation, memory protection, and controlled memory sharing. All the resources including all data structures in the system are organized as objects. The Object Management System manipulates each object through a 32-bit Object Handle or Object Identifier (OID). The Object Handle is unique across the system. Each object has an Object Attribute Block (OAB) which contains the type information, the status, protection, and debugging information of the object, and the pointers to the object representation in the memory. Sharing an object among processes is implemented by mapping the Object Handle into their own address spaces. The format of the Object Handle is shown in Figure 2.15. The processor number describes which processor contains the header data structure of the object. The sequence number is for debugging and error detection purposes. The Offset field is the physical address of the object due to the special usage of segment F8. As mentioned before, all processes in the system can access segments F8 through FF in addition to their own segments. Physical memory starting at location 0 is mapped into segment F8 in each processor node and is used for the operating system data structures.



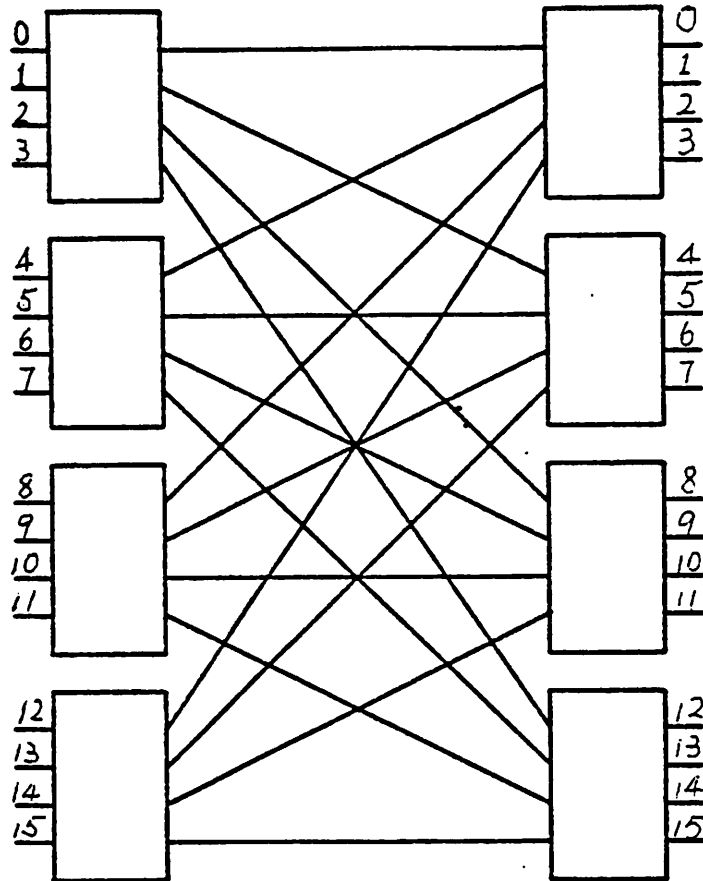


Figure 2.14 - Butterfly Network

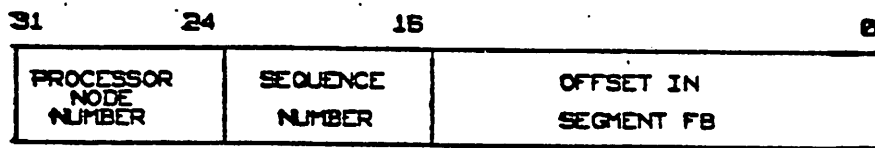


Figure 2.15 - The Object Handle

---

The Object Management System has four layers:

- The physical Memory Manager
- The Central Object Utility
- The Memory Mapping Manager
- Special Purpose Managers

The Physical Memory Manager allocates the physical memory on the node using Buddy algorithm. This algorithm divides the memory into blocks of size of power of two and allocates memory with a block of size of the nearest power of two. If there are no such blocks left, it breaks a block with the next larger size. The Central Object Utility provides the type-independent operations on objects, such as creation, deletion etc. The Memory Mapping Manager defines and maintains the virtual address spaces for the processes in the system. The Special Purpose Managers are a collection of managers for special objects such as processes, events and dual queues

The Object Management system provides routines to manipulate some special objects such as processes. To access these objects, a user must ask the Object Management system to give him the OID and map it into his address space. If an access to an object does not have the right access mode the access is rejected and an error is generated [RET83].

**2.3.2. Processes** All the tasks in the Butterfly system are represented by a set of communicating processes which share the main memory. Each process is represented as an object and is managed by the Object Management System. Each process has its own data structure called the Process Control Block (PCB) which contains the addressing and scheduling informations. When the process is switched out of the CPU, its state is also saved in its PCB. Each process also has a data structure called Process Template which contains a pointer to the code and the initialized data of the process.

All the Butterfly program are developed and compiled in a host computer, VAX. Object code with the extension ".68" is down loaded from the VAX to the Butterfly. A process is created by first asking the Object Management System for the Process Template. The execution of the Make\_Template system call allocates the Process Template and calls the loader demon. The loader demon cooperates with a program "bld" running on the VAX to load and create the process. Using the existing Template, a process can be created on a different node at run time by executing a Make\_Process system call. By specifying the argument block of the process being creating, the user can specify the name, the space information, protection information, the type of the process and some additional information. After having the Process Template, the user stack size has to be specified. If the specified stack size is too small for the process, a fatal error message of "stack overflow" will be generated at run time.

A process can be terminated by executing the "exit" system call or by another process through the "Del\_Obj" system call. When a process terminates, other processes can be informed by the event mechanism. If some of the objects of the terminating process are still needed by some other processes, the ownership of these objects should be transferred either to these processes or to the operating system. The Object Management System will invalidate the owner fields of all the other objects which will then be garbage collected by the garbage collector [RET83].

**2.3.3. Events** Events are an important mechanism for synchronization, scheduling and interprocess communication. An event is an object. Each event is owned by a process and is represented by a unique event handle. An object called "Event Block" is used to record the data of the event. An Event Block contains several fields:

- A link field for putting it in a queue
- Two fields containing type information
- A sequence number field
- A field of flag bits
- A protection field
- An Owner's Processor Handle field
- A data field
- An owner's data field

The sequence number field refers to the number of times this block of memory has been allocated. It matches the sequence number field in its Event Handle if this Event Handle is valid. The flags are for the error detection purposes. The owner's Process Handle is used to put this event into the right queue when the event is posted. It is also used by the garbage collector to reclaim this block of memory. Two data fields are pointers to the data block. In the Butterfly system, the Event Block is allocated explicitly before the event is posted. Because of this preallocation of the Event Block, it is better for a process to allocate all its Event Blocks when it starts to run.

When a process needs service from another process, it creates an event by making a `Make_Event` system call. The Object Management System allocates the Event Block in segment F8, and returns an Event Handle to the process. The process may then do some other jobs and check whether the event has occurred periodically, or it may go to sleep by executing a "wait" system call. When the event occurs, it will be posted, if the owner process is waiting for this event, the process will be waken up. A user can also specify a specific time at which an event should be posted by executing a `Set_Timer` system call.

**2.3.4. Dual Queues** The dual queue is the data structure used by the Chrysalis to implement the event and lock mechanism. The reason why it is called the dual queue is that it can contain either Event Handles of the waiting processes or queued data elements. A flag in the header tells whether it is a data queue or an event queue. Each dual queue has a header portion and a ring buffer. The ring buffer must have a length of a multiple of four bytes and it must be allocated within a 64K block. A library routine is used to initialize a dual queue. Locks are used for synchronization. When a dual queue is used as locks, a "lock" flag bit is set in the header, and it is limited to have only one element. An empty queue represents a locked lock. A process needing the lock can execute de-queue to wait for it or poll to test it. To unlock the lock, the process which is holding the lock uses enqueue to store its Process Handle on the queue [RET83].

**2.3.5. Buffer Management** A Buffer Management System has been developed for managing buffers more efficiently. There are two shortcomings in the Object Management System.

(1) When the Object Management System does a mapping, it checks the access privileges of the process, checks the validity of the Object Handle, and sets up a SAR pointing to the object. This is too time consuming for the buffer management where the mapping is frequently done.

(2) The object size allocated by the Object Management System is limited to be one segment. When the number of the objects gets large, there may not be enough SARs for the process.

The Buffer System uses the Object Management System to allocate a large amount of memory space as a buffer pool object. A buffer pool is obtained by executing a Make\_BFpool system call. It then suballocates buffers from this space by supporting a set of simpler and faster operations. A dual queue is used to keep all the legal buffer pool identifiers. Each buffer pool is given a unique ID from the dual queue at creation time. When the pool is deleted its ID is put back to the queue for reuse. Each buffer also has an ID. Figure 2.16 shows the format of the ID. The pool ID is the identifier assigned to the buffer pool at creation time. The offset is the offset from the buffer pool pointer. All the buffer IDs are put into a dual queue called Free Queue. A buffer is not an object. It is composed of a header and

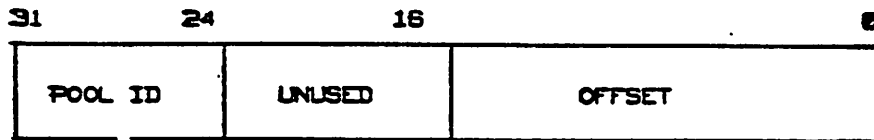


Figure 2.16 - The Buffer Identifier

---

a data area. A lock is used for each buffer pool for the mutual exclusive accesses among several processes to a buffer in the pool. A mechanism similar to "use count" is used to free a buffer shared by several processes. It is possible for a buffer pool to span more than one processor node.

Process can access buffers by submitting the Object Handle of the Free Queue. After a process selects the buffer pools and maps them into its address space, it maintains a table which transfers the buffer ID to the virtual address of the buffer. When a process finishes its work with a buffer, it may free the buffer by calling the BFree\_buf service routine; It may pass the buffer ID to another process without keeping a copy of the ID for itself; It may also retain the ID after it passes the buffer to another process [RET81].

## CHAPTER 3

### THE PROTOTYPE FLOATING-POINT CO-PROCESSOR

#### 3.1. Introduction

Although the Motorola 68000 has integer performance in the range of a super-minicomputer, it has no hardware or microcode to support floating-point operations. In Appendix 7, tables of floating-point performance on different processors are listed. It shows that the speed of floating-point operations on the MC68000 is 5-10% of that of their integer counterparts. The resulting performance degradation may be unacceptable when trying to use the Butterfly for scientific applications such as circuit simulation. The tradeoffs involved in co-processor design for the Butterfly are described first, then several architectures which may be used with the Butterfly processor node are presented. Finally, the architecture used in the prototype floating-point co-processor and associated experimental results are presented.

#### 3.2. The Interface To The Processor Node

A co-processor is a processor which offloads specific tasks from the main CPU. Traditionally, there are two ways to design a co-processor. The first way is to have the co-processor sit on the CPU address and data bus. The co-processor watches all bus activities and decodes all the instructions appearing on the bus. If the instruction is in its instruction repertoire, it executes the instruction and returns the result to the CPU. The second way is that instructions in the co-processor repertoire cause the main CPU to trap. The trap service routine then sends the information to the co-processor and starts the co-processor. Both these two methods are not suitable for our purpose. The first method is impossible since it is impossible to tell whether the 68000 is fetching an instruction or a data in a memory access. The second method is too slow to meet our performance requirements. The Butterfly

floating-point co-processor must have its own way to interface to the processor node.

The Butterfly floating-point co-processor interacts with the Processor Node Controller (PNC) instead of the 68000 CPU. It takes advantage of two features of the Butterfly processor node. One is that the PNC performs all the memory and switch functions efficiently. The other is that on the processor node there is a memory daughter board connector, which provides all the memory control signals and the signals coming out from the Memory Address Data bus (MAD bus). The floating-point co-processor is thus connected to the processor node by this daughter board connector. Figure 3.1 shows the Butterfly system with the floating-point co-processor. The 68000 CPU is not aware of this co-processor; it treats the co-processor as a local memory. Every floating-point instruction in the source code is compiled into a memory reference instruction referencing a "magic" location in subspace 0. This reference causes a microinterrupt to the PNC. The microinterrupt service routine then fetches the operands for the floating-point operation, passes them to the floating-point co-processor, and starts the micro engine on the co-processor. The co-processor works in lock step with the

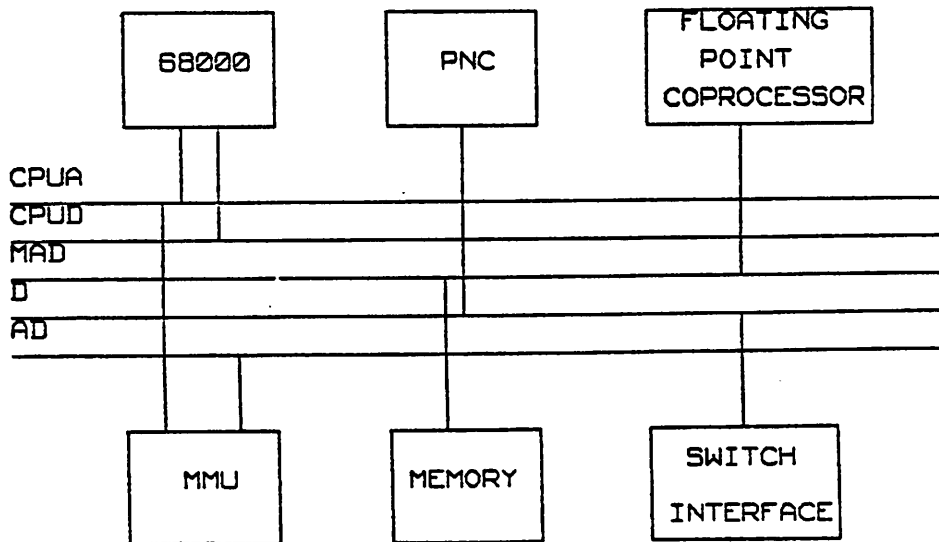


Figure 3.1 - Butterfly With The Floating-Point Co-processor

---



PNC. After several clock cycles, The PNC reads the result from the MAD bus, stores it to the right memory location, and informs the 68000 to end its memory cycle and to proceed. The 68000 waits for the result in the extended memory cycle until it receives an acknowledge signal from the PNC.

There are two ways to pass the opcode to the co-processor. One is to pass it as data; the other is to pass it as part of the address. To pass the opcode as a data, the 68000 can either put the opcode in segment F8 or put it in the same place with the operands, and let the PNC fetch it. This requires several cycles for the PNC to get the opcode and is thus a time consuming approach. When the 68000 references the subspace 0 "magic" location, only the virtual address bits 15 through 8 are used by the PLA to generate the address of the microinterrupt service routine. The physical address bits 7 to 0 are the same as the virtual address bits 7 to 0. These bits can be used to pass the opcode from the 68000 to the co-processor. No extra works are needed for this approach, so it was chosen as the mechanism for passing the opcode.

There are two ways to pass operands to the co-processor. One way is to pass them through segment F8. This is also slow because the process must change its access mode from user mode to kernel mode and then change back to the user mode afterwards. The other way is to use the run time stack. This way is better because most operations occur between data items on the stack, such as parameters and locals.

### **3.3. Tradeoffs In The Architecture Design**

Four different architectures were considered during the architecture design phase: the memory to memory architecture, the architecture using the stack pointer, the architecture using the frame pointer, and the on-board stack architecture. The tradeoff criteria are the performance and the complexity of implementation and debugging. The performance is the primary goal for the whole design. The complexity was limited in this case by the fact that the whole project was to be completed in two months. In this section, each of the four architectures is examined and the tradeoffs are compared.

**3.3.1. The Memory To Memory Architecture** The simplest architecture for implementing floating-point is to let the 68000 do everything. In order to allow multiprogramming, a floating-point operation has to be atomic, that is, no context switch is allowed during floating-point operation. As shown in Figure 3.2, the 68000 first turns the interrupts off. It then wakes up the floating-point co-processor and passes the operands to it. After waiting for the computation to finish, it then moves the result to the correct location. Finally, it turns the interrupts back on. There are three microinterrupts and several MOV macro instructions involved. It is therefore quite time consuming. Also, since the data movement is done in 68000 assembler code, it is hard to improve the performance further.

---

```

MOVEQ #n, magic-location-1      ;turn off interrupts
MOVEQ #n, magic-location-2      ;wake up co-processor
MOVL a6@(-8), d0
MOVL d0, special-location-1     ;pass operand 1
MOVL a6@(-4), d0
MOVL d0, special-location-2     ;pass operand 2
NOOP                             ;wait for result
NOOP
.
.
.
NOOP
MOVL special-location-3, d0     ;store result
MOVL d0, a6@(-12)
MOVEQ #n, magic-location-3      ;turn interrupt back on

```

Figure 3.2 - The Simplest Architecture

---

The advantage of this architecture is that it requires no changes in the PNC microcode, requires the least changes in the compiler and in the operating system.

**3.3.2. The Stack Pointer Architecture** In the Butterfly system, a run-time stack is used to hold the activation records. An activation record or a frame contains the parameters for the subroutine, its local variables, and some linkage information. One register, called the stack pointer (SP), is used to point to the top of the stack. Another register, called the frame pointer (FP), is used to point to the current activation record.

In the stack pointer architecture, the 68000 first moves the operands to the top of the stack. This takes two MOVL macro instructions. It then passes the stack pointer to the PNC by writing into a "magic" location in subspace 0, and waits for the result. After it gets the result, it puts the result in the frame. When the 68000 passes the stack pointer to the PNC, the PNC gets a micro interrupt. The PNC then translates the SP from a virtual address to a physical address, and wakes up the co-processor. The co-processor starts to work in lock step with the PNC. The PNC puts the physical address of the operand out to the memory. Two cycles later the operand is valid on the MAD bus. The co-processor then latches the operand into an internal register. After the co-processor gets all the operands, it starts the computation; the PNC waits for the result. After 6 clock cycles the result is available; if the PNC wants the result right way, the result is also available on the MAD bus. The PNC saves the result in its registers and initiates the memory cycle to write it to the top of the stack. Finally the PNC informs the 68000 to terminate the MOVL instruction and to proceed. The 68000 then moves the result from the top of the stack to the right place in the activation record, and adjust the stack pointer to where it was before the floating-point operation. Figure 3.3 shows the assembler code for this architecture.

---

MOVL a6@(-8), d0	;move operand 1 to top of stack
MOVL d0, sp@-	
MOVL a6@(-4), d0	;move operand 2 to top of stack
MOVL d0, sp@-	
MOVL sp, 0XFFC804	;pass the stack pointer
MOVL sp@(+4), d0	;save the result
MOVL d0, a6@(-12)	
ADDQ #8, sp	;adjust the stack pointer

Figure 3.3 - Code For The Stack Pointer Architecture

---

The performance of this architecture is better than the memory to memory architecture. It takes 8 macro instructions and 55 micro instructions, and the total execution time is 26  $\mu$ s. This architecture is quite flexible for performance improvement so that it is easier to build a

prototype in a conservative way and improve the performance later. This architecture does not require as many changes in the operating system and in the compiler as does the next architecture. It is clear that in this architecture the 68000 performs some unnecessary moves which slow down the overall operation. Also, the architecture requires a complicated PNC microinterrupt service code.

**3.3.3. The Frame Pointer Architecture** Figure 3.4 shows the stack sections in the stack pointer architecture and in the frame pointer architecture. In the stack architecture, the 68000 and the PNC communicates through the top of the stack. In the frame pointer architecture, the frame pointer is passed to the PNC and the PNC can fetch the operands directly from the frame and return the result into the right place in the frame. In addition of the frame pointer, the offsets of the operands and the offset of the result from the frame pointer must also be passed to the PNC. This can be accomplished in two ways. One is to pass the frame pointer in segment F8. Every time a subroutine containing the floating-point operations is called, a microinterrupt is generated to translate the FP from a virtual address to a physical address and save this physical address in a fixed location in segment F8. This fixed location is known by the PNC. Since this operation is performed once each subroutine call, and there are usually several floating-point operations in a subroutine requiring floating-point, the extra overhead per floating-point operation is thus small. The other way is to generate two separate microinterrupts. In one of the microinterrupt service routine, the FP is saved in the internal registers of the PNC. The other routine serves all the rest of the operations. The microcode in this case must make sure that the registers containing the FP is not rewritten by some other interrupts between the two floating-point microinterrupts. This method is more complicated and more time consuming since it is required for each floating-point operation. Figure 3.5 shows the assembler code for the frame pointer architecture.

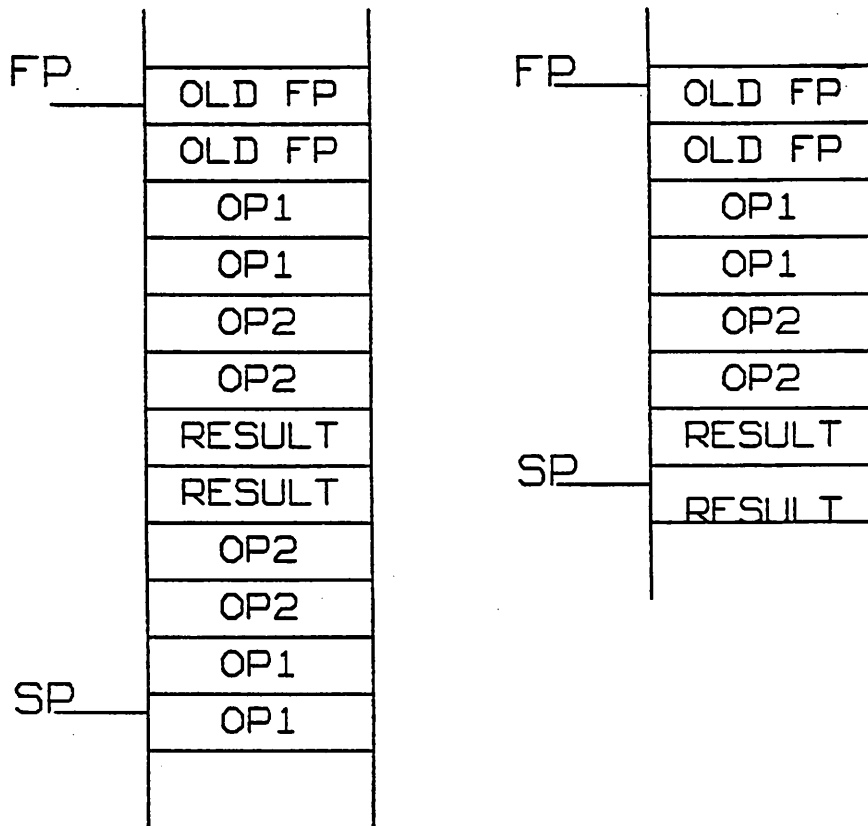


Figure 3.4 - Stack Segments From The SP Architecture And The FP Architecture

MOVI 0XFFC804, immediate-data ;pass offsets to the PNC

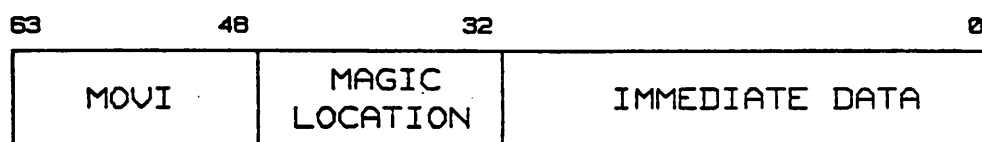
Figure 3.5 - The Frame Pointer Architecture

Figure 3.6 shows the instruction format for the frame pointer architecture. The 68000 assembler instruction MOVI writes the 32-bit immediate data into a "magic" location. The immediate data has four fields: offset of the operand 1, offset of the operand 2, an unused field, and the offset of the result. Since the PNC hardware does not support byte-swap operation, this format is so designed that the PNC can get the offsets of the operand 1 and the operand 2 from this 32-bit word immediately and, while waiting for the result, the PNC can shift the

offset of the result to the lower byte position. The PNC then calculates the effective physical address of the operands, fetches them, and passes them to the co-processor as in the SP architecture.

Obviously, this architecture has higher performance than the stack pointer architecture since it uses fewer macro instructions. The disadvantage of this architecture is that it requires complicated changes in the operating system, the compiler, and the PNC microcode. This disadvantage makes the debugging phase more complicated, especially because the Chrysalis operating system is not designed for easy debugging.

**3.3.4. On Board Stack Architecture** The memory access initialized by the PNC requires 3 clock cycles. Putting the run time stack on the co-processor can reduce the memory access time to one clock cycle. More importantly, all the intermediate results are kept on the stack so that the performance is increased significantly. However, this architecture requires more hardware than the previous architectures. Different memory interfaces to the co-processor and to the rest of the system must also be provided. More changes are required in the operating system. Obviously, this architecture should not be considered as the first prototype.



Immediate Data Format:

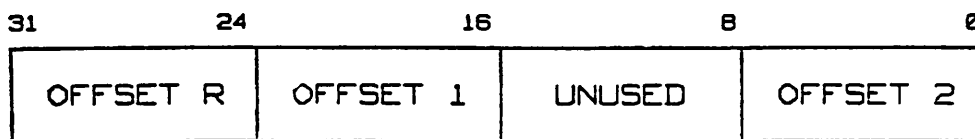


Figure 3.6 - Instruction Format For The FP Architecture

### 3.4. The Prototype

To achieve reasonable performance in a short period of design time and to have enough flexibility to improve the performance without changing the hardware later on, the prototype was designed to be microprogrammable for either the SP or FP architecture and the microcode for the SP architecture was written.

**3.4.1. The Hardware** Figure 3.7 shows the block diagram of the prototype floating-point co-processor the FPP1. The FPP1 was designed and built using TTL. The detailed design is included in Appendix 1. Since the 68000 and the PNC have a cycle time of 125 ns, the co-

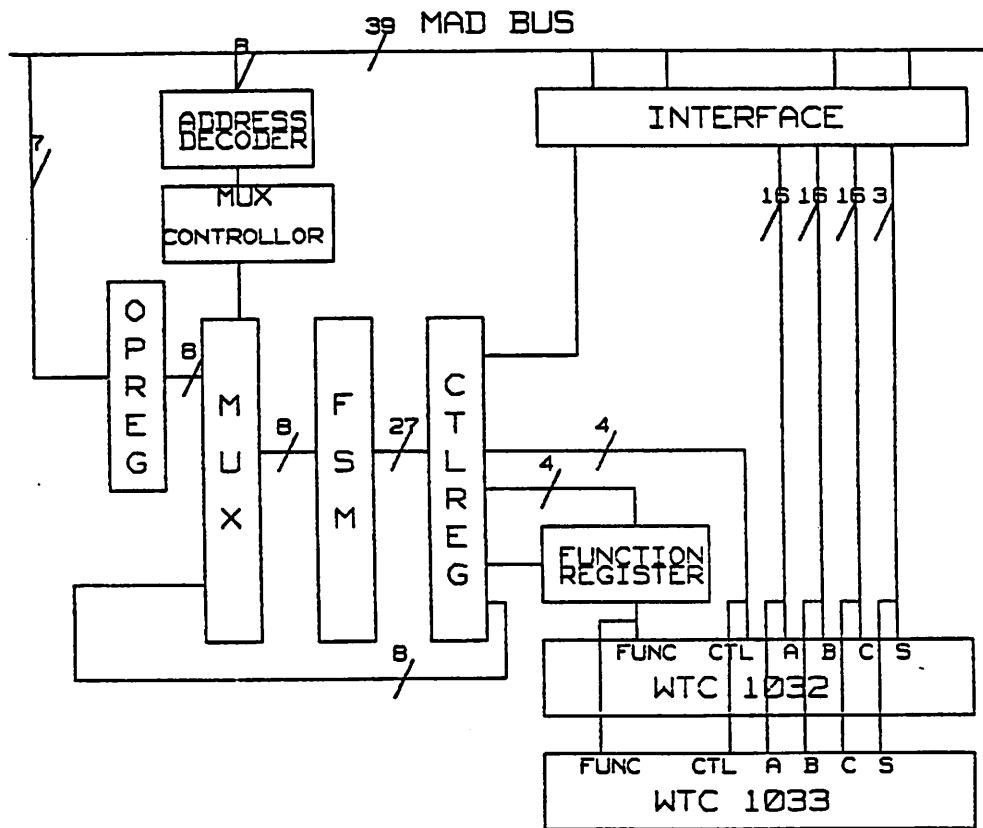


Figure 3.7 - The Prototype Floating-Point Co-processor FPP1

processor also uses the 8 MHz clock provided by the processor node.

The key components of the FPP1 are the 32-bit WEITEK 1032 floating-point multiplier and 1033 floating-point ALU [WEI83]. These WTC chips have 100ns cycle time and they support IEEE floating-point format. In the pipeline operating mode, they have the peak performance of 5 million floating-point operations per second. In the "flow" mode, 9 clock cycles are required for each floating-point operation. A finite state machine (FSM) implemented by microcode in a 512x28 PROM controls all the operations on the board in lock step with the PNC. The FSM normally sits in an idle loop; the multiplexer (MUX) normally selects the address coming out from the control register (CTLREG) which comes from the next address field of the microcode. When the address decoder recognizes a special pattern on the MAD bus in the address phase, it forces the MUX controller to select the next microcode address from the opcode register (OPREG). The opcode register in this cycle contains the pseudo opcode for the floating-point operation. This pseudo code is the entrance address of the microcode in the FSM for the required floating-point operation. From the very next cycle on, the MUX selects the next address field of the FSM microcode again. The FSM then sets up the function code, and mode code for the WTC chips, latches the operands and feeds them into the chips to start the floating-point operation. Only one of the WTC chips is activated in each operation. When the WTC chip is ready to output its result, the FSM latches the result into the output register. If the PNC is ready to read the result, the FSM also puts it on the MAD bus. The interface to the MAD bus contains a collection of registers, and TTL line drivers. The PNC microcode for the floating-point operation is included in Appendix 2; the microcode in the FSM and the program which generates the microcode of the FSM can be found in Appendix 3.

**3.4.2. Performance** The FPP1 is not optimized for the performance. It is now two to four times faster than the fast Motorola floating-point software. The performance seen by a high level application program is 26  $\mu$ s for each floating-point operation supported by the WEITEK chips. Table 3.1 compares the time required for a floating-point addition or multiplication by



the MIT software, Motorola software, and the prototype FPP1. The corresponding speed-up is listed in Table 3.2. A factor of 2.6 speed up can be obtained by using the frame pointer architecture without changing the FPP1 hardware.

OPERATION	MIT	MOTOROLA	FPP
ADD	298	69.21	26
MUL	453	97.9	26

Table 3.1 - Time Required For Addition And Multiplication

OPERATION	MIT	MOTOROLA
ADD	11.48	2.66
MUL	17.42	3.77

Table 3.2 - Speed Up

The performance of FPP1 is limited by the long time required for the PNC to pass operands from the processor-node memory to the WEITEK chips. In next chapter, some alternative co-processor architectures which have higher performance will be presented.

## CHAPTER 4

### PROPOSALS FOR THE NEXT GENERATION CO-PROCESSOR

#### 4.1. Introduction

The first generation floating-point co-processor demonstrates that it is possible to build a floating-point co-processor for the 68000-based Butterfly system but it does not provide optimal performance on the machine. Two architectures for the next generation floating-point co-processor which promise to improve the floating-point performance are presented in this chapter.

FPP2 is proposed for those applications requiring moderate performance, low cost, and fast development. By adding the capability of carrying out fast evaluation of frequently used elementary functions in hardware, the FPP2 increases the performance of the co-processor. Since it is also microprogrammable, it is flexible enough that more power can be added to the co-processor when needed. It requires the least changes in hardware from FPP1 of all the architectures under consideration.

The FPP3 provides the best possible performance and can be modified to use 64-bit floating-point chips. A 2910-based micro sequencer and a dual-port, fast, on-board memory make it possible to carry out large blocks of calculations at very high speed. A shift register chain is designed to provide easy debugging and diagnosis.

In the first section the tradeoffs in the design are described. In the second section the algorithms for evaluating elementary functions are presented. The design of FPP2 is described in detail in section three and in the last section the design of FPP3 is described.

## 4.2. Tradeoffs

The second generation floating-point co-processor design involves tradeoffs among performance, accuracy of the result, and the complexity of hardware implementation. The floating-point calculations in more accurate circuit models for circuit simulation require 64-bit precision. The choice of what kind of chips to use is made by considering that the most important goal of the second generation co-processor is high speed with an acceptable precision.

The best choice of all 64-bit floating-point chips is the HP 64-bit Monolithic floating-point processor [WAR82]. It provides very high speed as well as 64-bit precision. However, it is not commercially available. To build a 64 bit floating-point processor from off-the-shelf components, bit-sliced components, such as Am2903-based processor [AMD83], the Am29116 [AMD83], the NS16081 processor [NSD82], or Intel 80287 co-processor [INT84] may be used.

Using a 16-bit processor such as Am29116 to perform 64-bit arithmetic, operations such as partitioning the 64 bits into 16-bit parts and then putting them back together according to the IEEE floating-point format, must be performed in microcode. Therefore, the performance of this approach may not be acceptable. A 64-bit processor can be built using 16 Am2903 processors and 5 Am2959 carry-look-ahead parts. The large parts count of the ALU alone reduces the board area available for memory, which is critical for achieving the best performance. Besides, the need to implement the floating-point operations in the microcode also degrades the performance of the system. These disadvantages make it inappropriate to use either bit-sliced parts or the Am29116 processor.

The Intel 80287 numeric data co-processor provides both arithmetic and elementary functions, such as exponential and logarithm. It also provides 64-bit precision. The drawback is its slow speed. Using a 5MHz clock, it takes 14  $\mu$ s to perform an addition, 27  $\mu$ s for multiplication, 39  $\mu$ s for division, 36  $\mu$ s for square root, and 100  $\mu$ s for exponential [INT84]. Its single 16-bit I/O port makes it slower considering moving the 64-bit data in and out of the chip. It cannot meet our performance requirements.

The NS16081 processor provides 64-bit floating-point operations. It also provides addition, multiplication and division on the same chip. The only disadvantage of this processor is that its speed does not meet our requirement. It takes 7.4  $\mu$ s to perform an addition, 6.2  $\mu$ s for multiplication, and 11.8  $\mu$ s for division after it receives all the operands [NSD82]. It may be considered the main processor on the FPP3. However, it has only one 16-bit I/O port. It takes 12 clock cycles to get two 64-bit operands into-and one 64-bit result out of-the processor. This limits the potential performance of the FPP3 greatly.

Since the WEITEK 32-bit floating-point chips are the fastest floating-point chips available commercially and it is expected that the 64-bit WEITEK chips will be available soon, the second generation floating-point co-processor for the Butterfly is designed based on the WEITEK chips with an NS16081 chip on board optionally. It is expected that 32-bit precision can satisfy most CAD applications. Occasional calculations requiring 64-bit precision can be performed in software. Those applications requiring 64-bit precision can use the NS16081 chip. The design should consider the flexibility of upgrade the 32-bit chip to 64-bit chip, when the WEITEK 64-bit chips or other compatible chips are available.

#### **4.3. Hardware Support For Evaluating Elementary Functions**

The prototype FPP1 hardware supports only the functions available in the WTC chips, and the software does the rest. In many applications, divide, square root, exponential, logarithm, and other elementary functions are used frequently. The FPP2 is designed to support these functions by adding minimum amount of hardware.

Using table look-up method to evaluate division and square root using IEEE floating-point format is quite time consuming. In order to perform fast evaluation for  $A/B$  and  $\sqrt{A}$ , the table look-up and iterative method suggested by the WEITEK application note are used [WEI83a]. Issues of accuracy and IEEE compatibility are discussed in the application note. Algorithms for calculating  $\log A$  and  $A^x$  are included in Appendix 6, which may be implemented in microcode in the FPP2.

Before describing these algorithms, it is necessary to introduce the single precision IEEE floating-point format. Figure 4.1 shows the IEEE floating-point format. The highest bit is the sign bit. The next eight bits are the exponent and the last 23 bits are the fraction.

Figure 4.2 illustrates the table look-up for divide (A/B). The upper 32-bit word is the divisor B. The lower 32-bit word is the quotient Q. The sign bit of Q is the sign bit of B. The 8-bit exponent of B is used as the address of an 8 x 256 ROM. The output of the ROM forms the exponent of the quotient. The highest 12 bits of the fraction addresses a 12x4K ROM. The output of this ROM is the highest 12 bits of the fraction of the quotient. The content of the exponent ROM is calculated by the equation  $G = 253 - E$  which is the inverse of the exponent of B. The content of the fraction ROM is calculated by the equation

$$H = \frac{4096 \times 8192}{4097 + F} - 4096$$

H is an approximate reciprocal of B. This table look-up gives the initial accuracy to about 12 bits. The accuracy is extended to about 24 bits using the following function:

$$H_1 = H \times (2 - B \times H)$$

The final quotient is obtained by multiplying A with H1.

$$Q = A \times H_1$$

In this calculation three multiplications and one subtraction are required [WEI83a].

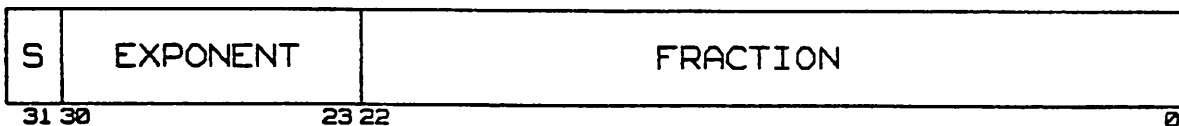


Figure 4.1 - IEEE Floating-Point Format

---

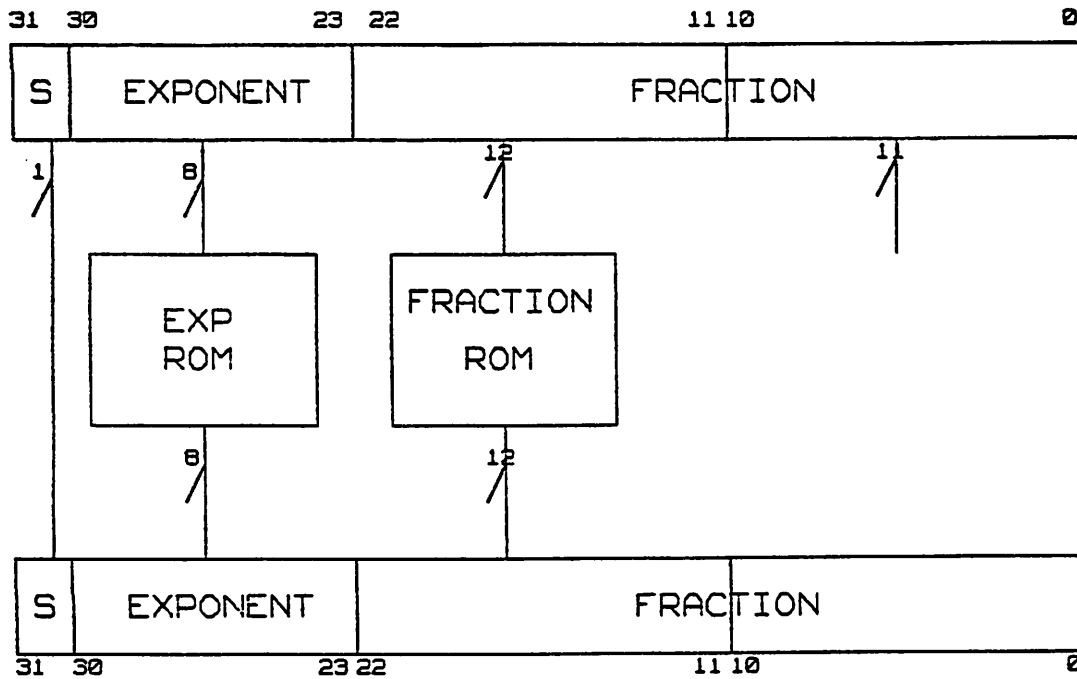


Figure 4.2 - Mapping Of Table Look-Up For A/B

Table look-up for  $\sqrt{A}$  is shown in Figure 4.3. The least significant bit of the exponent is used to distinguish the cases between A has an even exponent and A has an odd exponent. G is calculated by the function:

$$G = \frac{379 - E}{2}$$

H is calculated by the function:

If  $E(8) = 1$  then

$$H = \frac{2^{19}}{4097 + F} - 4096$$

IF  $E(8) = 0$  then

$$H = \frac{2^{19}}{8192 + 2F} - 4096$$

H is approximately equal to  $\frac{1}{\sqrt{A}}$ . The initial table look-up gives about 12 bits of accuracy

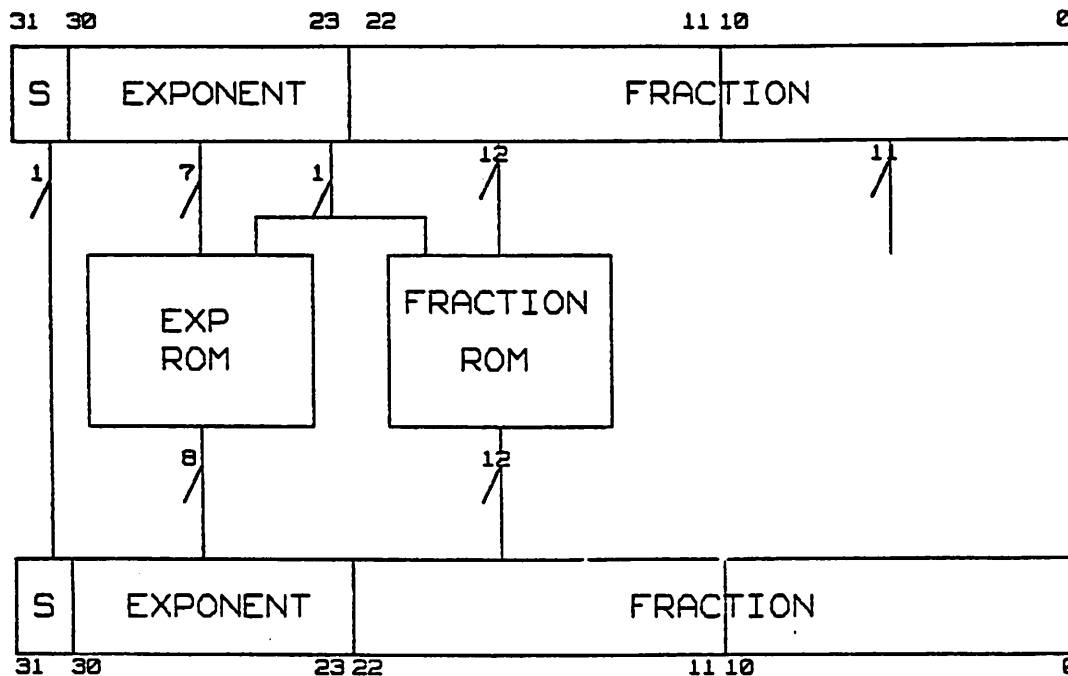


Figure 4.3 - Mapping Of Table Look-Up For Square Root

which is improved to about 24 bits by the following function:

$$H_1 = 0.5 \times H \times (3.0 - A \times H \times H)$$

In the calculation five multiplications and one subtraction are performed [WEI83a].

#### 4.4. The FPP2

The frame pointer architecture is used to interface to the floating-point co-processor and to the PNC in FPP2. That is that the 68000 passes the frame pointer to the PNC; the PNC fetches the operands and passes them to the co-processor. After the FPP2 finishes the calculation, it passes the result to the PNC. The PNC then stores the result in the preallocated locations in the active frame.

As shown in Figure 4.4, the control part and the interface to the WEITEK chips of the FPP2 are exactly the same as that of the FPP1. Two 32-bit registers, AOP and BOP are used to

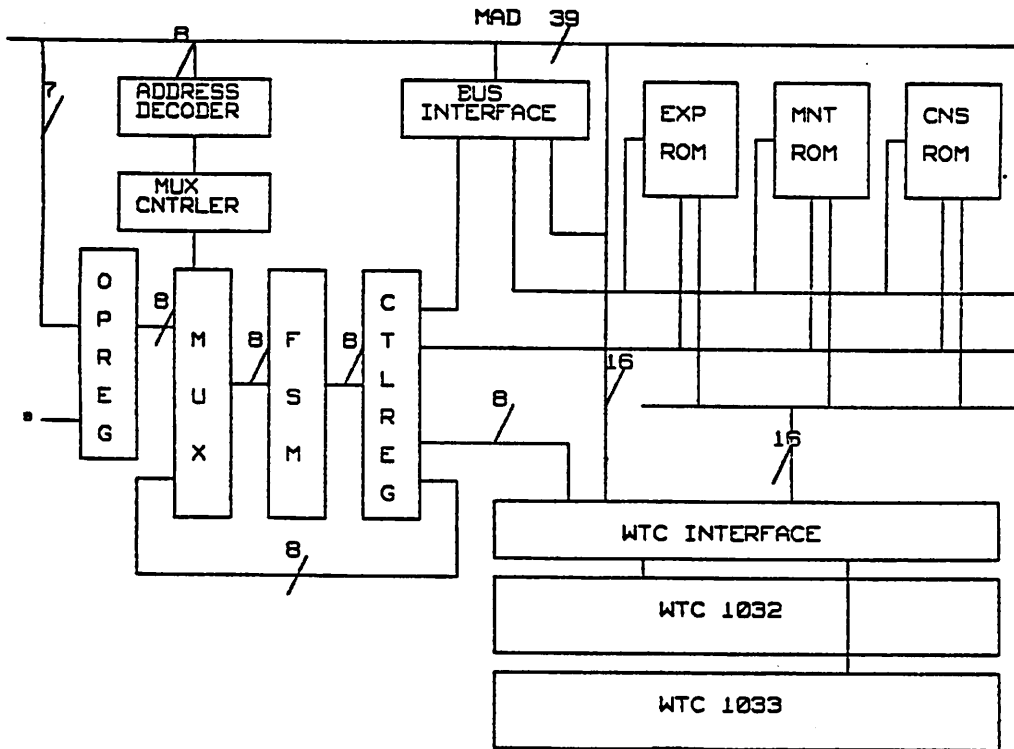


Figure 4.4 - The FPP2

save the two operands passed by the PNC. BOP register contains the operand which is used for table look-up. A constant ROM holds the constants needed by the computations. A  $4K \times 8$  read-only memory, EXP ROM, contains the required approximation of the exponent part. Three 12-bit ROMs contain the required approximation of the fraction part. The delay register is provided to multiplex the output of the ROMs onto the 16-bit ROM bus. A flip-flop is used to put the sign bit of operand B on to the ROM bus at the right time. A multiplexer in front of the BM register allows the B operand to be selected either directly from the MAD bus or from the table ROM. The path directly from MAD bus to the BM register allows the simple operations such as addition and multiplication to bypass the table ROM. A path from



the CL register to the AM register provides the fast path for recurrence calculations.

The microprogram can be written very easily for FPP2. The estimated time for carrying out division is 15  $\mu$ s: 10  $\mu$ s to pass the operands to the board and 5  $\mu$ s for the execution. The estimated time for square root is 15.5  $\mu$ s: 9.5  $\mu$ s for passing the operand and 6  $\mu$ s for execution.

#### 4.5. The FPP3

**4.5.1. Introduction** The fact that three clock cycles are needed for the PNC to access a 16-bit data from the local memory limits the speed of the floating-point calculations on the Butterfly. The only way to achieve the highest performance in floating-point operations is to keep as many data items on-board as possible. The FPP3 is designed to provide the highest floating-point speed possible in the Butterfly system. In this section the design of the FPP3 is described. I designed FPP3 jointly with Jeffrey T. Deutsch.

The FPP3 provides two ways to interface the PNC and the co-processor. The frame pointer architecture is used to carry out the single floating-point operations such as multiplication and  $A^x$ . If a complicated block such as the model evaluation is called by the high level program, blocks of parameters are passed to the co-processor through segment F8.

The FPP3 consists of four major parts: the control, the bus interface and the dual-port static RAM, the table ROM and the functional unit. As shown in Figure 4.5, they communicate through the MAD bus and three on board busses A, B, and C. A chain of on board shift registers is used to load the microcode into the writable control store at the reset time. It also provides a good diagnostic and debugging facility.

**4.5.2. The Control** Drawing 2 in Appendix 5 shows the detailed design of the FPP3 control unit. A 2910-1 microsequencer, a 4K  $\times$ 80 bit writable control store, and a 80-bit pipeline register generate all the control signals for the board. A condition PLA collects the informations about the operands and generates the condition code. The sequencer then selects the condition for the next microinstruction. A bootstrap PROM initializes the micro sequencer,

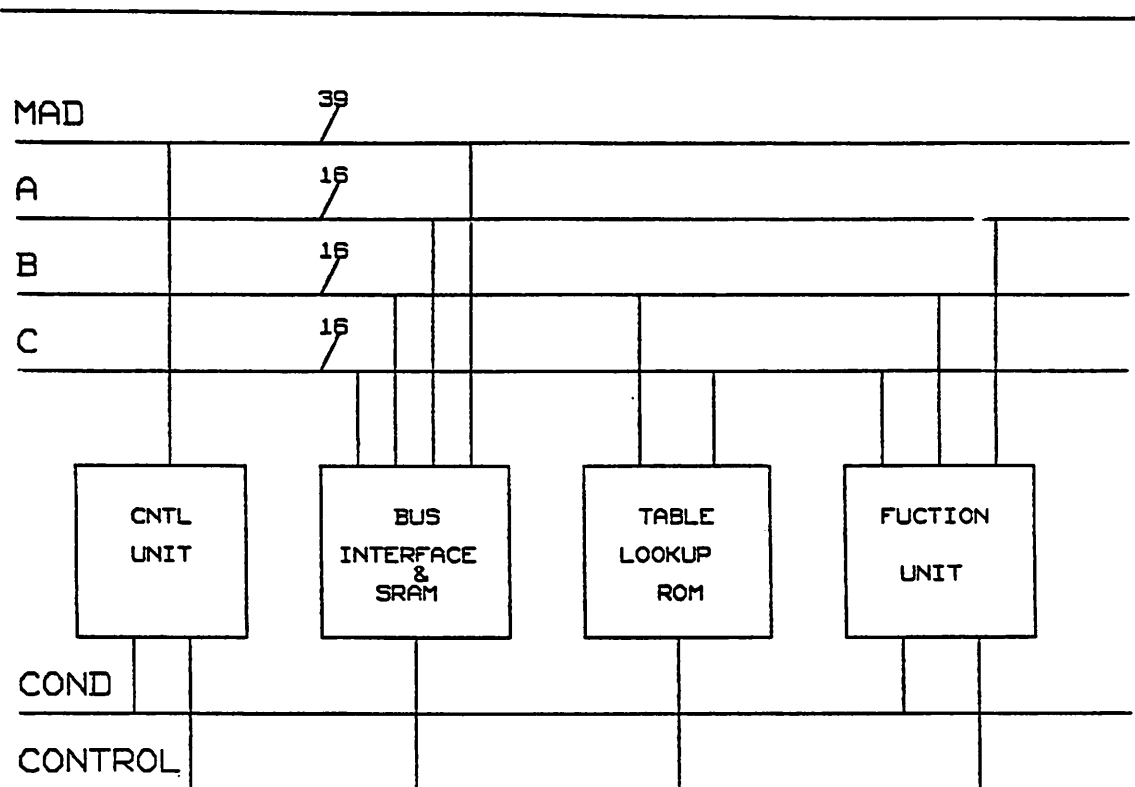


Figure 4.5 - The FPP3

loads the micro control program into the writable control store, and starts the normal operation of the micro engine. A multiplexer (MUX) selects the instruction for the 2910-1 sequencer from multiple sources.

The writable control store (WCS) is loaded from the MAD bus under the control of both the PNC and the bootstrap finite state machine (BSFSM). In order to keep the part count low, the WCS is loaded serially through a shift register. Considering the fact that the PNC has to take care of some other tasks such as memory refresh, the microcode loading is broken into blocks. Each block contains one 80-bit microinstruction. If the shifting is not overlapped with loading each 16-bit data from the PNC, it takes 19 cycles to load each 16 bits data from

the MAD bus and shift it to the right place. The loading time is thus about  $4K \times 19 \times 5$  clock cycles which is about 50 milliseconds.

Three commands are provided for the PNC to control the BSFSM to start the FPP3 in the normal operation mode. They are "LOAD MICROINSTRUCTION", "END OF LOADING", and "BEGIN". All of them are implemented as special patterns on the MAD bus during the address phase. When the BSFSM receives the reset signal, it forces the control unit to execute the instruction at location 0. The control unit loops at location 0 and waits for further instructions. When the BSFSM receives the LOAD MICROINSTRUCTION command, it loads one microinstruction into the WCS. During loading, the address of the WCS is generated by the 2910-1 under the control of the BSFSM. When it receives the "END OF LOADING" command, it sets the registers in the register chain in normal operating mode and goes into a state waiting for the floating-point instructions. When the BSRAM receives the "BEGIN" command, it forces the 2910-1 to accept instructions from the MAP register which contains the address of the beginning of the microcode for the desired function.

**4.5.3. The ALU** Drawing 3 in Appendix 5 illustrates the design of the ALU of the FPP3. A WEITEK 1033 ALU chip, and a WEITEK 1032 multiplier provides the high speed floating-point operations. A set of comparators generates the condition signals for the sign, the exponent, and the fraction parts of the operand which are fed into the condition PLA. A register MSW\_REG is there to hold the most significant word of the operand and to form a 32-bit input for the comparator. Necessary paths are provided for operating the WEITEK chips at a peak speed of 5 million floating-point operations per second which is the full speed at which the WEITEK chips can run.

Optionally, an NS16081 can be put on the B bus and the C bus. The FPP3 control unit controls the interface of the NS16081 chip. The microcode fields needed for this chip may be overlapped with those of the WEITEK chips to reduce the parts count.

**4.5.4. The Bus Interface And Dual Static RAM** Drawing 4 in Appendix 5 shows the bus interface and the dual-port static RAM of FPP3. All functional units are connected by three on board busses. The A bus and the B bus are for the operands; and the C bus are for the result. The MAD bus transfers the information between the PNC and the FPP3. The board communicates with the MAD bus through C bus. A path between the A bus and the C bus provides the path from the dual-port static RAM outputs to the MAD bus. A path between the C bus and the B bus provides a bypass path from the table ROM and the output ports of the WEITEK chips to the input ports of the WEITEK chips.

Two 16K  $\times$ 16 fast static RAMs provides the high speed on board memory. These two memories are organized as a dual-port memory system. They contain exactly the same contents. Thus the micro engine can address two operands in the same cycle. These two operands are passed to the WEITEK chips through the A bus and through the B bus. When data is written into the on-board memory system, both of these memories are written. This 16K dual-port RAM system may be partitioned into 256 segments with 64 words each segment or 64 segments with 256 words each segment. Segment 0 contains constants. One segment is allocated to each block of computations, such as a model evaluation in circuit simulation, for its parameters and temporary variables. The address to this dual-port memory is formed by concatenating 8 bits segment number and 6 bits offset provided by the microcode. A multiplexer (MUX) selects the 8-bit segment number from a constant address 0, the MAD bus, and the C bus. It thus allows both the PNC and the FPP3 control unit to select a segment.

In order to run the WEITEK chips at its full speed we needed both a large amount of on-board memory and the ability to read two 16-bit words and write one 16-bit word in the same clock cycle. This requires a RAM with less than 35ns access time. The largest existing 35ns access time RAM is a 4K static RAM which may not be able to hold parameters for all the simulation models on board. The FPP3 is designed to use 16K RAM at present but it is easy to convert this memory system to one with two read ports and one write port when the 35ns large static RAMs are available. This dual-port memory organization is the key factor of

the high speed of the FPP3.

**4.5.5. The Table ROM** Drawing 5 in Appendix 5 shows the ROM for the table look-up for functions  $1/B$  and  $\sqrt{A}$ . This part of the design is the same as that in the FPP2. The sign bit of the B operand is passed to the C bus in the right cycle. The  $4K \times 8$  EXP\_ROM contains the exponent of the resulting functions. Up to 16 functions can be put in this ROM. The function is selected by the microcode. Its address is formed by concatenating 4-bit microcode and 8-bit exponent from the B operand. Its output is the approximation of the exponent of  $1/B$ . The two DFRA\_ROMs form the table of the fraction part of  $1/B$ . It is addressed by the fraction part of B. The SFRA\_ROM1 contains the approximate value of  $\sqrt{B}$  when B has an even exponent; the SFRA\_ROM2 contains the approximate value of  $\sqrt{B}$  when B has an odd exponent. They are addressed by the fraction part of B and are selected by the least significant bit of the exponent of B. Three registers are used to hold and form the addresses to these tables. One register is used for time multiplexing the fraction part to the C bus. A latch whose value is always zero is used to form the lower bits of the fraction of the approximation of the result.

**4.5.6. The Diagnostic System** The pipeline register in the control unit and all the registers interfacing the busses have both parallel and serial I/O ports. The serial I/O ports are all chained together. They can be operated in two modes. The normal operating mode is the parallel I/O mode; they behave just as normal registers. In the diagnostic mode, these registers behave like a single shift register chain.

There are five commands for the PNC to run the diagnosis of the board. They are "BEGIN DIAGNOSTICS", "SCAN IN", "SCAN OUT", "SINGLE STEP" and "END DIAGNOSTICS". When the "BEGIN DIAGNOSTICS" command is received by the bootstrap finite state machine, it puts the board in the diagnostic mode. In this mode, the PNC can write any pattern into any parts on the board, or read the content of any on-board memory. The PNC can also single-step the on-board operation. The ability of read-modifying-write any registers on board and single-step the executions are very similar to those provided by the

software debug system. It is possible to develop a good debugging software on top of this system.

**4.5.7. The Micro Instruction Format** The FPP3 contains horizontal microcode with 80 bits per micro word. Figure 4.6 shows the microcode format of the FPP3. It contains four fields each of which controls one of the four parts. Bits 0 to 18 form the control field for the control unit. Bits 19 to 35 are the control field for the ALU functional unit. Bits 36 to 50 are the control field for the table ROM unit. Bits 51 to 76 are the control for the bus interface and the dual-port static RAM unit. Bits 77 to 80 are unused.

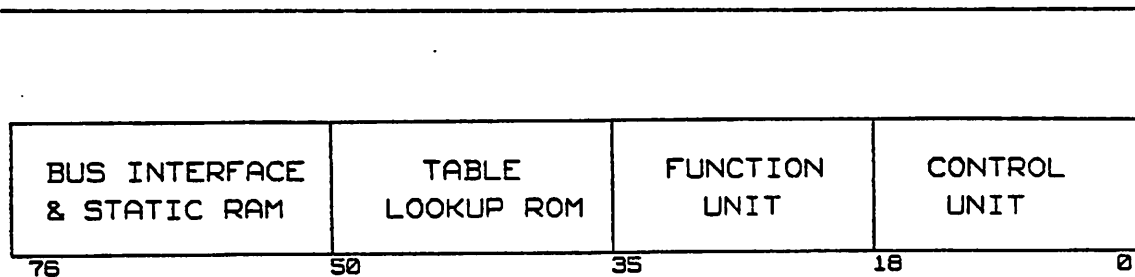


Figure 4.6 - FPP3 Microcode Format

---

## CHAPTER 5

### SUMMARY

The BBN Butterfly Multiprocessor System has been used as a test-bed for ideas of how to build a high-speed CAD machine. A prototype of the first generation floating-point co-processor, the FPP1, has been built. The FPP1 is a WEITEK 1032/1033 based, micro-programmable machine. The communication protocol between the FPP1 and the Butterfly processor node is the same protocol as between the memory daughter board and the processor node. In the current implementation of the FPP1, the 68000 passes operands and result between the processor node and the FPP1 through the run time stack by passing the stack pointer to the processor node controller (PNC). It takes 26  $\mu$ s to perform a 32-bit floating-point addition or a 32-bit floating-point multiplication seen by a high level application program, which is two to four times faster than the Motorola fast software. By changing the architecture to using the frame pointer instead of the stack pointer, the speed of the FPP1 will be 2.6 times faster.

Two proposals, the FPP2 and the FPP3, are made for the next generation floating-point co-processor. The FPP2 is a proposal for the applications requiring low cost and moderate performance. It provides the ability of fast evaluation of elementary functions in the hardware.

The FPP3 provides the highest possible speed of floating-point operations on the Butterfly. It may be built to run the WEITEK chips at their full speed with 5 million floating-point operations per second using a 4K static RAM, or to run them at half the full speed using a 16K static RAM. The on-board fast static RAM is organized to be able to hold all the parameters and temporary variables and to support the data flow of the WEITEK chips at their full speed. A writable control store is designed to make the microcode writing easier.

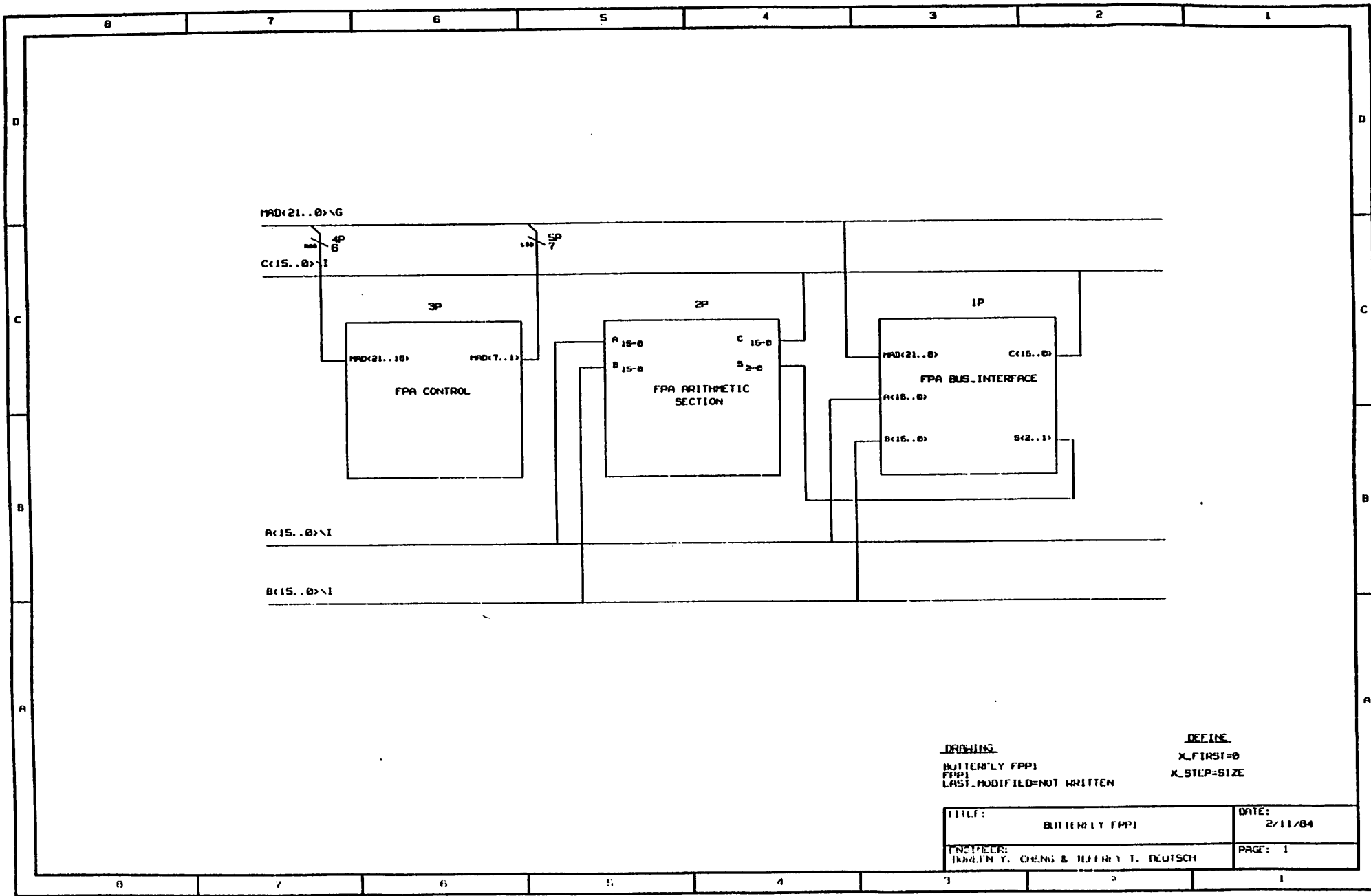
Hardware support for diagnostics is provided by a shift register chain which connects all the key registers together. This system makes the diagnostics and debugging much easier.



## APPENDIX 1

### SCHEMATICS OF THE FPP1

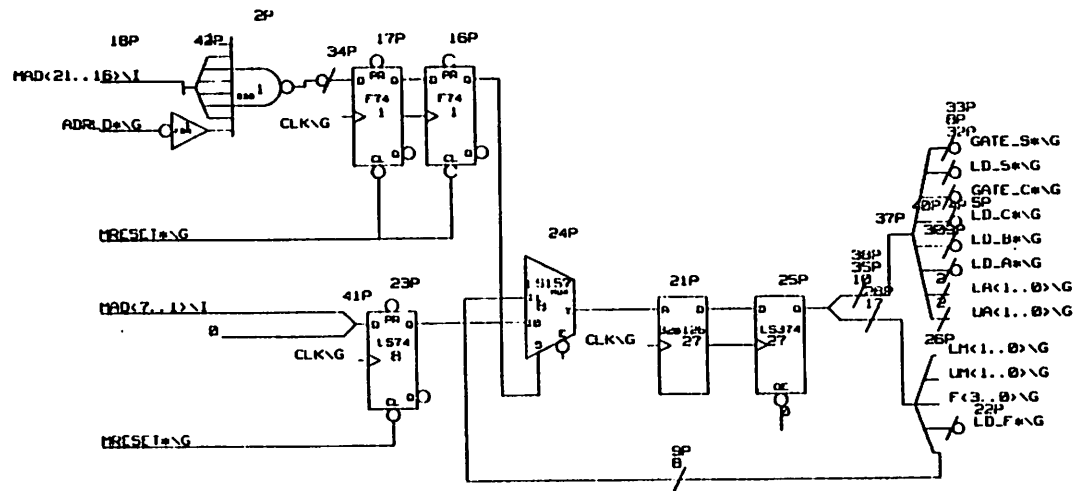
Included in this appendix are the schematics of the prototype of the first generation floating point coprocessor. Drawing 1 is the overall block diagram of the FPP1. Drawing 2 is the FPP1 control unit. Drawing 3 is the FPP1 arithmetic unit. Drawing 4 is the FPP1 bus interface. Drawing 5 is the interface to the Butterfly processor node.



DRAWING  
 BUTTERFLY FPP1  
 FPP1  
 LAST MODIFIED=NOT WRITTEN

DEFINE  
 X.FIRST=0  
 X.STEP=SIZE

TITLE:	BUTTERFLY FPP1	DATE:	2/11/84
DESIGNER:	IRVING Y. CHENG & JERRY T. DEUTSCH	PAGE:	1

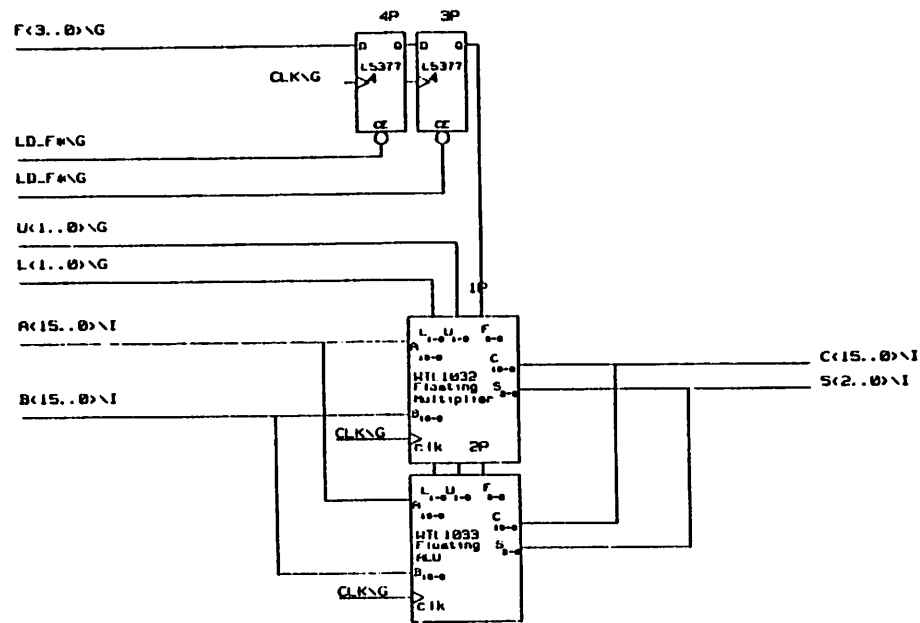


DRAWING

DEFINE

XREF=51-0  
 X\_STEP=51Z  
 FPA CONTROL  
 LAST\_MODIFIED=Sun May 13 14:47:10 1984

TITLE:	FPA CONTROL	DATE:	1/12/84
DESIGNER:	DORLIN Y. CHENG & JEFFREY T. DEUTSCH	PAGE:	1



DRAWING

FPA ARITHMETIC SECTION  
 1 PAPER 11

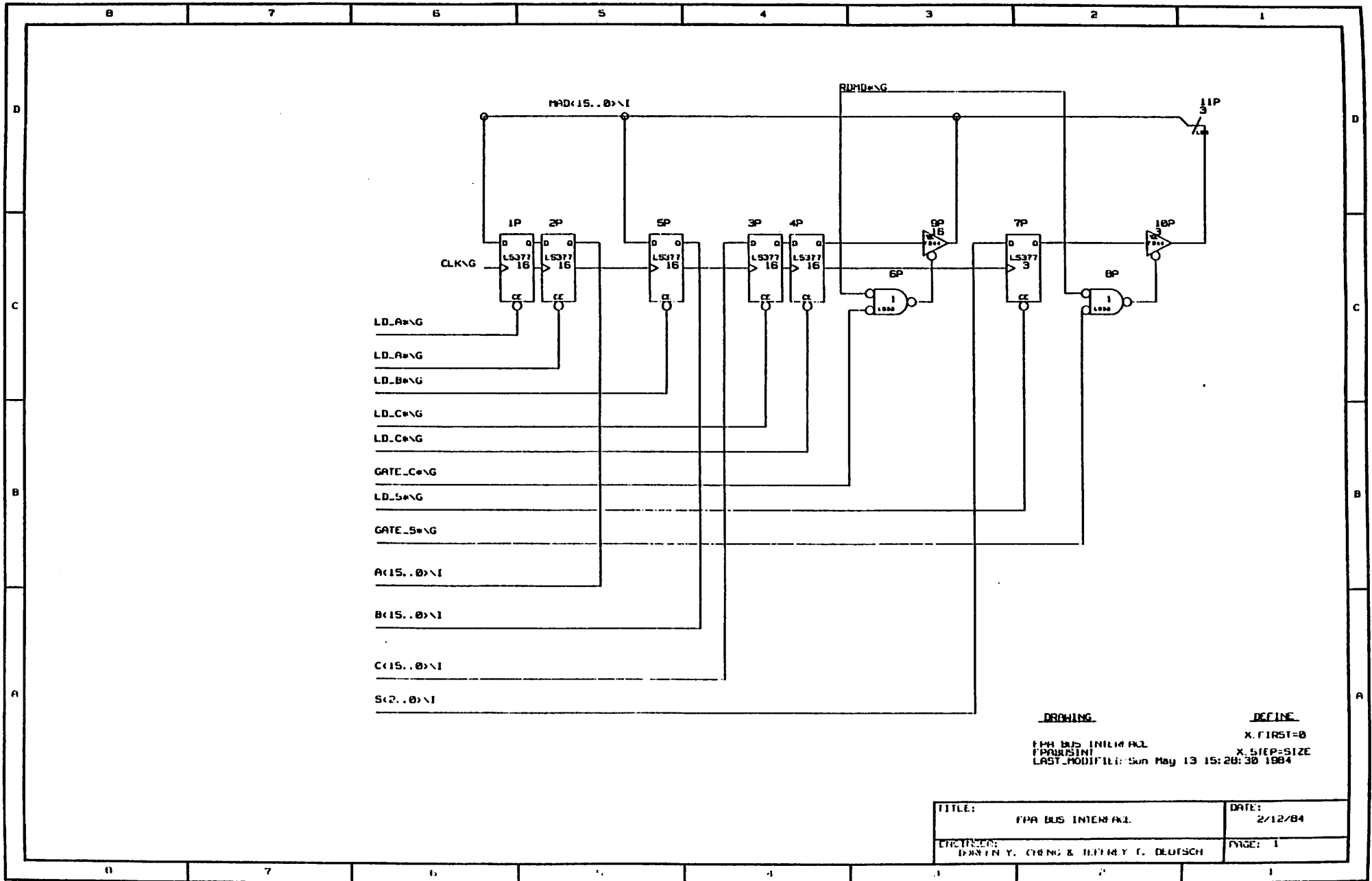
LAST\_MODIFIED=Sun May 13 13:29:14 1984

DEFINE

X\_FIRST=0

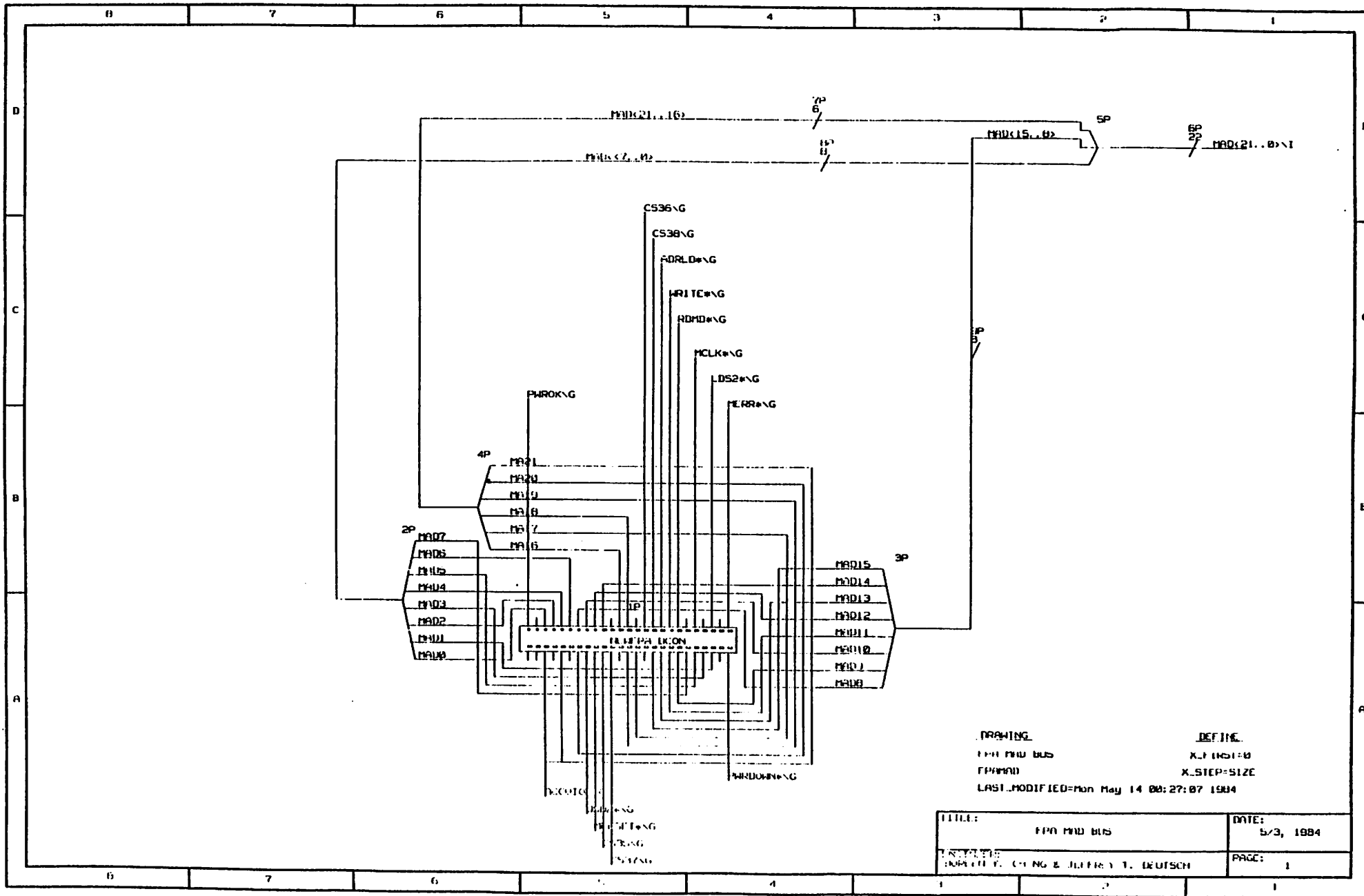
X\_SIZE=512L

TITLE: FPA ARITHMETIC SECTION	DATE: 2/12/84
DESIGNED BY: GOREN Y. CHENG & JEFFREY T. DEUSCH	PAGE: 1



DRAWING DEFINE  
 X.FIRST=0  
 X.SIZE=SIZE  
 LAST\_MODIFIED: Sun May 13 15:28:30 1984

TITLE:	FPA BUS INTERNAL	DATE:	2/12/84
DESIGNER:	DWAYN Y. CHENG & JEFFREY T. DEUTSCH	PAGE:	1



DRAWING: FPD MMD BUS  
 K.F. KOSTER  
 FPMMD  
 LAST MODIFIED: Mon May 14 08:27:07 1984

DEFINE:  
 K.F. KOSTER  
 K. STEP: SIZE  
 1

TITLE:	FPD MMD BUS	DATE:	5/3, 1984
DESIGNER:	DAVID Y. CHING & JEFFREY T. DEUTSCH	PAGE:	1

## APPENDIX 2

### PNC MICROCODE FOR FLOATING POINT

Included in this appendix is the program which generates the PNC microcode for the floating point functions. In order to make this program work, some changes are also made in pnc.h and pnc.m which can be found in BBN microcode files.

pnc10.m

; This version reads the result in to PNC registers and then writes it to the  
; memory. It treats FPP as a memory daughter board.

; The floating point function entries are in the range 0xFFC800 to 0xFFC820.  
; OP code for functions in binary:

; UNIT	FUNCTION	OP CODE	VIRTUAL ADDR
; WTC 1033:	A+B	0b00001000	0xFFC804
;	A-B	0b00001010	0xFFC808
;	-A+B	0b00001100	0xFFC80C
;	WRAP A	0b00000000	0xFFC810
;	UNWRAP A	0b00000010	0xFFC814
;	FLOAT	0b00000100	0xFFC818
;	FIX A	0b00000110	0xFFC81C
;	ABS A	0b00001110	0xFFC820
;	+ ABS B		
;	ABS(A+B)	0b00010010	0xFFC824
;	ABS(A-B)	0b00010000	0xFFC828
;			
; WTC 1032:	A*B	0b00100000	0xFFC82C
;	WA*B	0b00100010	0xFFC830
;	A*WB	0b00100100	0xFFC834
;	WA*WB	0b00100110	0xFFC838
;			
; ADDED FUNCTIONS:			
;	A/B	0b01000000	0xFFC83C
;	SQRT A	0b01000010	0xFFC840
;	1/A	0b01000110	0xFFC844
;			
; Virtual address 0xFFC845 to 0xFFC8FF are reserved.			

float.1:

```

    fixed_at(0x150),
    interrupt,
    lemi,
    call(float.2,Ar1,),
    cpuD -> D -> Tx2,
    D -> r1,
    setaux(fHALT, mmR -> aD)
;test for low word vs high word
;if cycle 2, virtual SP bits 15.0
;if cycle 1, save virtual SP 31.16
;for float.2f

float.2f:
    at(0x3a2),
    case(!Ar1,),
    br(float.3f),
    move(r0,r2)
;save function code in r2

float.3f:
    at(0x312),
    br(idle_ret),
    mmu9,
    assert(mmR1 -> aD1),
    cpuACK,
    move(r1, ),
    alu -> D -> Tx1,
    setaux(aux_dflt)
;select memory protection word
;access violation?
;set up alu output
;MSW of virtual SP in Tx1
;turn 68000 back on

float.2:
    at(float.2f-2),
    case(Ar1,),
    br(float.3),
    Tx2 -> D -> r1
;68000 is writing the low word
;r1 contains MSW of virtual SP
;set up interface to rdpbk

float.3:
    at(float.2f+1),
    call(rdpbk.1),
    setBerr
;Have to save Tx2 away
;read first 2 words from stack
;request 68000 rerun the bus cycle

float.4:
    at(float.3+1),
    br(float.5),
    setaux(aux_dflt),
    eras
;turn 68000 back on
;rdpbk already started reading the
;third word

float.5:
    at(0x316),
    br(float.6),
    ADr -> mAD -> D -> Tx3,
    eras
;MSW of operand 2 in Tx3

float.6:
    at(0x3a5),
    next,
    0x003f -> D -> mAD,
    mAD -> Ah1 -> mA,
    fpp
;Wake up FPP, float address portion
;use the constant field in CS15.0
;Address 21.16 = 111111
;see pnc.h p.11, This asserts ADrLD

```



```

float.6a:
    at(float.6+1),
    next,
    Ah1 -> mA,           ;Address 21.16 = 111111
    eras

float.7:
    at(float.6a+1),
    br(float.7a),
    move(r2, ),
    alu -> D -> mAD     ;gate the function part
                        ;FPP FSM float entry

float.7a:
    at(0x322),
    br(float.8),
    move(r2, ),
    alu -> D -> mAD     ;wait for FPP to be set up
                        ;gate the function part
                        ;FPP FSM float entry

float.8:
    at(0x327),
    br(float.9),
    Tx1 -> D -> mAD     ;MSW of operand 1 to FPP
                        ;rdpbk returns MSW of operand 1

float.9:
    at(0x33c),
    br(float.10a),
    Tx2 -> D -> mAD     ;LSW of operand 1 to FPP

float.10a:
    at(0x357),
    br(float.10),
    move(r3, ),
    alu -> D -> mAD -> Ah1 ;Ah1 is trashed by float.6

float.10:
    at(0x36e),
    br(float.11),
    Ah1 -> mA,
    add(r0,r10,r0),
    alu -> D -> mAD -> ADr,
    mR
                        ;read MSW of operand 2
                        ;rdpbk returns physical address
                        ;of the MSW of operand 2 in r0
                        ;start reading MSW of operand 2

float.11:
    at(0x36f),
    br(float.12),
    Tx3 -> D -> mAD,
    eras
                        ;MSW of operand 2 to FPP

float.12:
    at(0x377),
    br(float.13),
    ADr -> mAD -> D -> Tx4,
                        ;LSW of operand 2 in Tx4
                        ;FPP should get this too!

```

```

eras

float.13:
    at(0x379),
    br(float.14)                                ;wait for result

float.14:
    at(0x37d),
    br(float.15)

float.15:
    at(0x382),
    br(float.16)

float.16:
    at(0x389),
    br(float.17)

float.17:
    at(0x393),
    br(float.18)

float.18:
    at(0x397),
    br(float.19)

float.19:
    at(0x3a1),
    br(float.20)

float.20:
    at(0x3a8),
    br(float.21)

float.21:
    at(0x3a9),
    br(float.22)

float.22:
    at(0x3aa),
    br(float.22a)

float.22a:
    at(0x3ab),
    br(float.23)

float.23:
    at(0x3ac),
    br(float.24),
    ADr -> mAD -> D -> Tx3,                    ;MSW of result in, Tx3 ADr = FPP
    fpp

float.24:
    at(0x3ad),
    br(float.25),

```

```

    ADr -> mAD -> D -> Tx4,      ;LSW of result in Tx4, ADr = FPP
    fpp

float.25:
    at(0x3ae),
    br(float.26),
    Ah1 -> mA,
    sub(r0,r10,r0),
    alu -> D -> mAD -> ADr,      ;Write MSW of result
    mWwE

float.26:
    at(0x3af),
    next,
    Tx3 -> D -> mAD -> ADr,      ;MSW of result to memory
    eras

float.27:
    at(float.26+1),
    br(float.28),
    eras                          ;2nd eras to memory

float.28:
    at(0x3b1),
    br(float.29),
    Ah1 -> mA,                    ;write LSW of result
    add(r0,r10,r0),
    alu -> D -> mAD -> ADr,
    mWwE

float.29:
    at(0x3b2),
    next,
    Tx4 -> D -> mAD -> ADr,      ;LSW of result to memory
    eras

float.30:
    at(float.29+1),
    br(proc_ret),
    eras                          ;2nd eras

```

## APPENDIX 3

### PROTOTYPE MICROCODE

In this appendix, the microcode in the finite state machine on the floating point board, both the overall microcode and the content in each PROM, are included. The microcode is not generated by the BBN micro assembler, the program which generates this microcode is also included.

fpp.h

```
/*
This file is required for generating microcode for the FSM on FPP.
It contains the definitions for the fields of the microword.
```

Microword definition:

BIT #	SIGNAL NAME	
0	GATE_S	
1	LD_S	
2	GATE_C	
3	LD_C	
4	LD_B	
5	LD_A	
6	L1_2	for 1032
7	L0_2	
8	U1_2	
9	U0_2	
10	L1_3	for 1033
11	L0_3	
12	U1_3	
13	U0_3	
14	F0	for both
15	F1	
16	F2	
17	F3	
18	LD_F	
19.26	GOTO	next address
27	UNUSED	

```
*/
```

```
/* The following fields are defined to be ANDed */
```

```
#define NOOP          0x0007d13f
#define GATE_S        0x0007fffe
#define LD_S          0x0007fffd
#define GATE_C        0x0007fffb
#define LD_C          0x0007fff7
#define LD_B          0x0007ffef
#define LD_A          0x0007ffdf
#define LD_F          0x0003ffff
#define MODE          0x00061dff          /* for AND */

/* control code for 1032 */
```

/\* The following fields are defined to be ORed \*/

```
#define NLD_2      NOOP                /* L1,L0 = 00 */
#define LAB_2      0x00000080          /* L1_2 = 0, L0_2 = 1 */
#define LA_2       0x00000040          /* L1_2 = 1, L0_2 = 0 */
#define LMODE_2    0x000000c0          /* L1_2 = 1, L0_2 = 1 */
```

/\* The following fields are individually defined \*/

```
#define DAB_2      NOOP                /* U1_2 = 1 */
#define ENB_2      0x0007d03f          /* U1_2 = 0 */
#define UMS_2      ENB_2              /* U1_2 = 0, U0_2 = 0 */
#define ULS_2      0x0007d23f          /* U1_2 = 0, U0_2 = 1 */
```

/\* control code for 1033 \*/

/\* The following fields are defined to be ORed \*/

```
#define NLD_3      NOOP                /* L1_3 = 0, L0_3 = 1 */
#define LAB_3      0x00000800          /* L1_3 = 1, L0_3 = 0 */
#define LA_3       0x00000400          /* L1_3 = 1, L0_3 = 1 */
#define LMODE_3    0x00000c00
```

/\* The following fields are individually defined \*/

```
#define DAB_3      NOOP                /* U1_3 = 1 */
#define ENB_3      0x0007c13f          /* U1_3 = 0 */
#define UMS_3      0x0007c13f          /* U1_3 = 0, U0_3 = 0 */
#define ULS_3      0x0007e13f          /* U1_3 = 0, U0_3 = 1 */
```

/\* OPcode for 1033 \*/

/\* The following fields are defined to be ANDed \*/

```
#define WRAP      0x0004113f
#define UNWRAP    0x0004513f
#define FLOAT     0x0004913f
#define FIX       0x0004d13f
#define ADD       0x0005113f
#define SUB       0x0005513f
#define NSUB      0x0005913f          /* -A+B */
#define AADD      0x0005d13f          /* ABS A + ABS B */
#define SUBA      0x0006113f          /* ABS(A-B) */
#define ADDA      0x0006513f          /* ABS(A+B) */
```

/\* OPcode for 1032 \*/

```
#define MUL      0x0004113f
#define WMUL     0x0004513f          /* WRAP A * B */
#define MULW     0x0004913f          /* A * WRAP B */
#define WMULW    0x0004d13f          /* WRAP A * WRAP B */
```

fpp.c

```
#include <stdio.h>
#include <ctype.h>

#include "fpp.h"

#define WOP          19

/* This one is for the first testing only. Later on will change for single
   operand operations. It is also needed to change the lall */
#define COMMON_1    0x90
#define COMMON_2    0x70
#define COMMON_3    0x50

#define ADD_E       0x4
#define SUB_E       0x8
#define NSUB_E      0xc
#define WRAP_E      0x10
#define UNWRAP_E    0x14
#define FLOAT_E     0x18
#define FIX_E       0x1c
#define AADD_E      0x20
#define ADDA_E      0x24
#define SUBA_E      0x28
#define MUL_E       0x2c
#define WMUL_E      0x30
#define MULW_E      0x34
#define WMULW_E     0x38

#define IDLE_LOOP   0
#define CODE_SIZE   512

unsigned long      ucode[ CODE_SIZE ];

main() {
    register int i;

    set_ucose();

    for( i = 0; i < CODE_SIZE; i++ ) {
        printf( "%07x0, ucode[ i ] );
    }
}

set_ucose() {

    register int i, next;

    /* Zero everyone out */

    for( i = 0; i < CODE_SIZE; i++ ) {
        ucode[ i ] = (IDLE_LOOP << WOP | NOOP);
    }
}
```

```

/* IDLE_LOOP */

next = IDLE_LOOP;
ucode[next] = (next << WOP | NOOP);

/* A+B entry */

next = ADD_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_3 << WOP | ADD & LD_F & NOOP );

/* A-B entry */

next = SUB_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_3 << WOP | SUB & LD_F & NOOP );

/* -A+B entry */

next = NSUB_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_3 << WOP | NSUB & LD_F & NOOP );

/* WRAP entry */

next = WRAP_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_1 << WOP | WRAP & LD_F & NOOP );

/* UNWRAP entry */

next = UNWRAP_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_1 << WOP | UNWRAP & LD_F & NOOP );

/* FLOAT entry */

next = FLOAT_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_1 << WOP | FLOAT & LD_F & NOOP );

/* FIX entry */

```



```

next = FIX_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_J << WOP | FIX & LD_F & NOOP );

/* ABS A + ABS B entry */

next = AADD_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_3 << WOP | AADD & LD_F & NOOP );

/* ABS(A+B) entry */

next = ADDA_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_3 << WOP | ADDA & LD_F & NOOP );

/* ABS(A-B) entry */

next = SUBA_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_3 << WOP | SUBA & LD_F & NOOP );

/* A*B entry */

next = MUL_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_2 << WOP | MUL & LD_F & NOOP );

/* WA*B entry */

next = WMUL_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_2 << WOP | WMUL & LD_F & NOOP );

/* A*WB entry */

next = MULW_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_2 << WOP | MULW & LD_F & NOOP );

/* WA*WB entry */

```

```

next = WMULW_E;
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | MODE & LD_F & NOOP);
ucode[next++] = (COMMON_2 << WOP | WMULW & LD_F & NOOP );

```

```

/* COMMON_3 for WTC 1033 */

```

```

next = COMMON_3;
ucode[next++] = (next+1 << WOP | LMODE_3 | LD_F & LD_B & NOOP);
ucode[next++] = (next+1 << WOP | LAB_3 | LD_A & LD_B & NOOP);
ucode[next++] = (next+1 << WOP | LAB_3 | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | UMS_3);
ucode[next++] = (next+1 << WOP | ULS_3 & LD_S); /* this may be wrong! */
ucode[next++] = (next+1 << WOP | LD_C & NOOP);
ucode[next++] = (next+1 << WOP | LD_C & GATE_C & NOOP);
ucode[next++] = (next+1 << WOP | GATE_C & NOOP);
ucode[next++] = (IDLE_LOOP << WOP | LD_C & GATE_C & NOOP);

```

```

/* COMMON_2 for WTC 1032 */

```

```

next = COMMON_2;
ucode[next++] = (next+1 << WOP | LMODE_2 | LD_B & LD_F & NOOP);
ucode[next++] = (next+1 << WOP | LAB_2 | LD_A & LD_B & NOOP);
ucode[next++] = (next+1 << WOP | LAB_2 | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | UMS_2);
ucode[next++] = (next+1 << WOP | ULS_2 & LD_S);
ucode[next++] = (next+1 << WOP | LD_C & NOOP);
ucode[next++] = (next+1 << WOP | LD_C & GATE_C & NOOP);
ucode[next++] = (next+1 << WOP | GATE_C & NOOP);
ucode[next++] = (IDLE_LOOP << WOP | LD_C & GATE_C & NOOP);

```

```

/* COMMON_1 for single operand */

```

```

next = COMMON_1;
ucode[next++] = (next+1 << WOP | LMODE_3 | LD_F & NOOP);
ucode[next++] = (next+1 << WOP | LA_3 | LD_A & NOOP);
ucode[next++] = (next+1 << WOP | LA_3 | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | NOOP);

```

```
ucode[next++] = (next+1 << WOP | NOOP);
ucode[next++] = (next+1 << WOP | UMS_3);
ucode[next++] = (next+1 << WOP | ULS_3 & LD_S);
ucode[next++] = (next+1 << WOP | LD_C & NOOP);
ucode[next++] = (next+1 << WOP | LD_C & GATE_C & NOOP);
ucode[next++] = (next+1 << WOP | GATE_C & NOOP);
ucode[next++] = (IDLE_LOOP << WOP | LD_C & GATE_C & NOOP);
}
```

The Microcode

02fd11f  
037d11f  
03a113f  
281113f  
04fd11f  
057d11f  
05a113f  
281513f  
06fd11f  
077d11f  
07a113f  
281913f  
08fd11f  
097d11f  
09a113f  
280113f  
0afd11f  
0b7d11f  
0ba113f  
280513f  
0cfd11f  
0d7d11f  
0da113f  
280913f  
0efd11f  
0f7d11f  
0fa113f  
280d13f  
10fd11f  
117d11f  
11a113f  
281d13f  
12fd11f  
137d11f  
13a113f  
282513f  
14fd11f  
157d11f  
15a113f  
282113f  
16fd11f  
177d11f  
17a113f  
380113f  
18fd11f  
197d11f  
19a113f  
380513f  
1afd11f  
1b7d11f

1ba113f  
380913f  
1cfd11f  
1d7d11f  
1da113f  
380d13f  
28bdd2f  
297d90f  
29fd93f  
2a7d13f  
2afd13f  
2b7d13f  
2bfd13f  
2c7d13f  
2cfd13f  
2d7c13f  
2dfe13d  
2e7d137  
2efd133  
007d133  
38bdief  
397d18f  
39fd1bf  
3a7d13f  
3afd13f  
3b7d13f  
3bfd13f  
3c7d13f  
3cfd13f  
3d7d03f  
3dfd23d  
3e7d137  
3efd133  
007d133

## APPENDIX 4

### Modified LA11

In this appendix the boolean equations which define the entrance address of the PNC microinterrupt service routine for the floating point functions, and the current contents of this PLA are presented.

floating point functions are implemented by writing to group A registers  
!150 for two operands floating point functions  
!151 for single operand floating point functions  
! 01 0101 0000 if v1.0=00 & wr=1 & a15.8 = 11001000  
! 01 0101 0001 if v1.0=00 & wr=1 & a15.8 = 11001001

csa0= /v1\*/v0\*wr\*a15\*a14\*/a13\*/a12\*a11\*/a10\*/a9\*a8;

csa4= /v1\*/v0\*wr\*a15\*a14\*/a13\*/a12\*a11\*/a10\*/a9\*/a8;

csa6= /v1\*/v0\*wr\*a15\*a14\*/a13\*/a12\*a11\*/a10\*/a9\*/a8;

la11 content

\$A000,  
00

\$A100,  
ff ff ff f0 ff f0 ff f0 ff 30 ff 30 ff 30 3f f0  
ff 00 f3 3c c0 bf ff ff d5 2a 2a d5 3f c0 ff 00  
ff 7f f3 bc ff ff 4f 4f

\$A200,  
ff ff dd ff bb ff 77 ff ff 01 e1 1f 1f e1 01 ff  
fe 01 ff ff 1f e1 ff ff ff 01 01 ff 01 ff ff 00  
ff 77 ba dc ef ef ee fe

\$A300,  
ff ff bd ff 7b ff f7 ff ef 3e 3f c0 00 ff 00 ff  
ff 00 ff 01 fe c1 ff ff ff 00 00 ff 00 ff ff 00

ef f7 7b bd ff de df de

\$A400,

ff ff 7f ff 7f ff 7e ff 7f f8 78 ff 78 ff 78 e7  
ff 80 ff 78 67 dd ff ff 6f f0 70 ef 60 ff 9f e0  
ff 66 27 5f fb fb 65 23

\$A500,

1f 1e 17 0f 1f 07 1f 07 07 1f 1f 07 1f 07 07 1f  
18 07 1f 1f 1f 05 1f 1f 1f 1f 1f 07 1f 18 07  
1f 1d 1f 16 0f 1f 0f 1b

\$A600,

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00

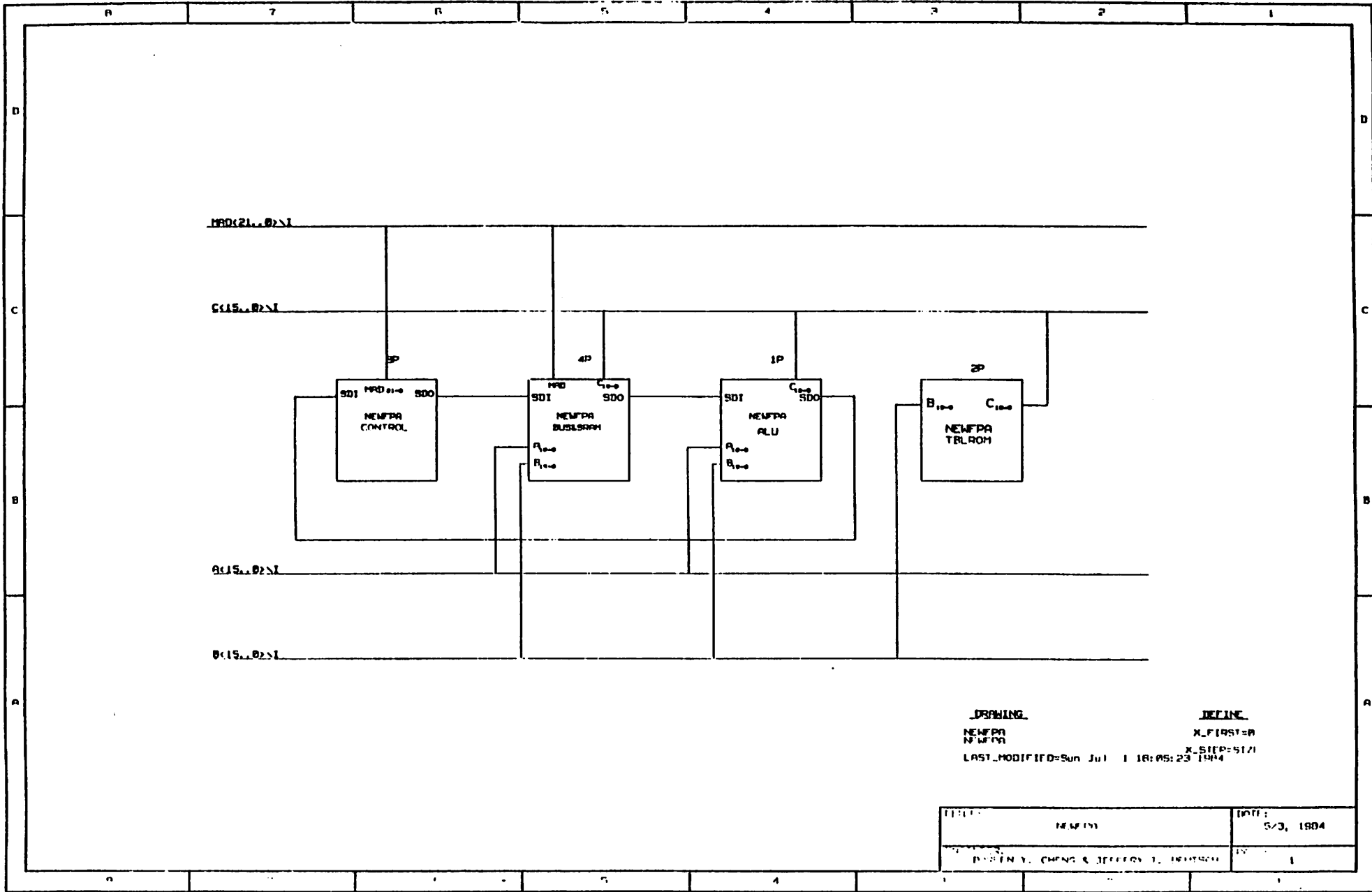
\$S739d,

## **APPENDIX 5**

### **SCHEMATICS FOR THE FPP3**

This appendix contains the drawings for the FPP3. Drawing 1 is the high level block diagram of the FPP3. Drawing 2 is the control unit. Drawing 3 is the ALU and functional unit. Drawing 4 is the bus interface and the dual-port static RAM. Drawing 5 is the table ROM and its interface. Drawing 6 is the interface to the Butterfly processor node.





DRAWING

NEWPPA  
NEWPPA

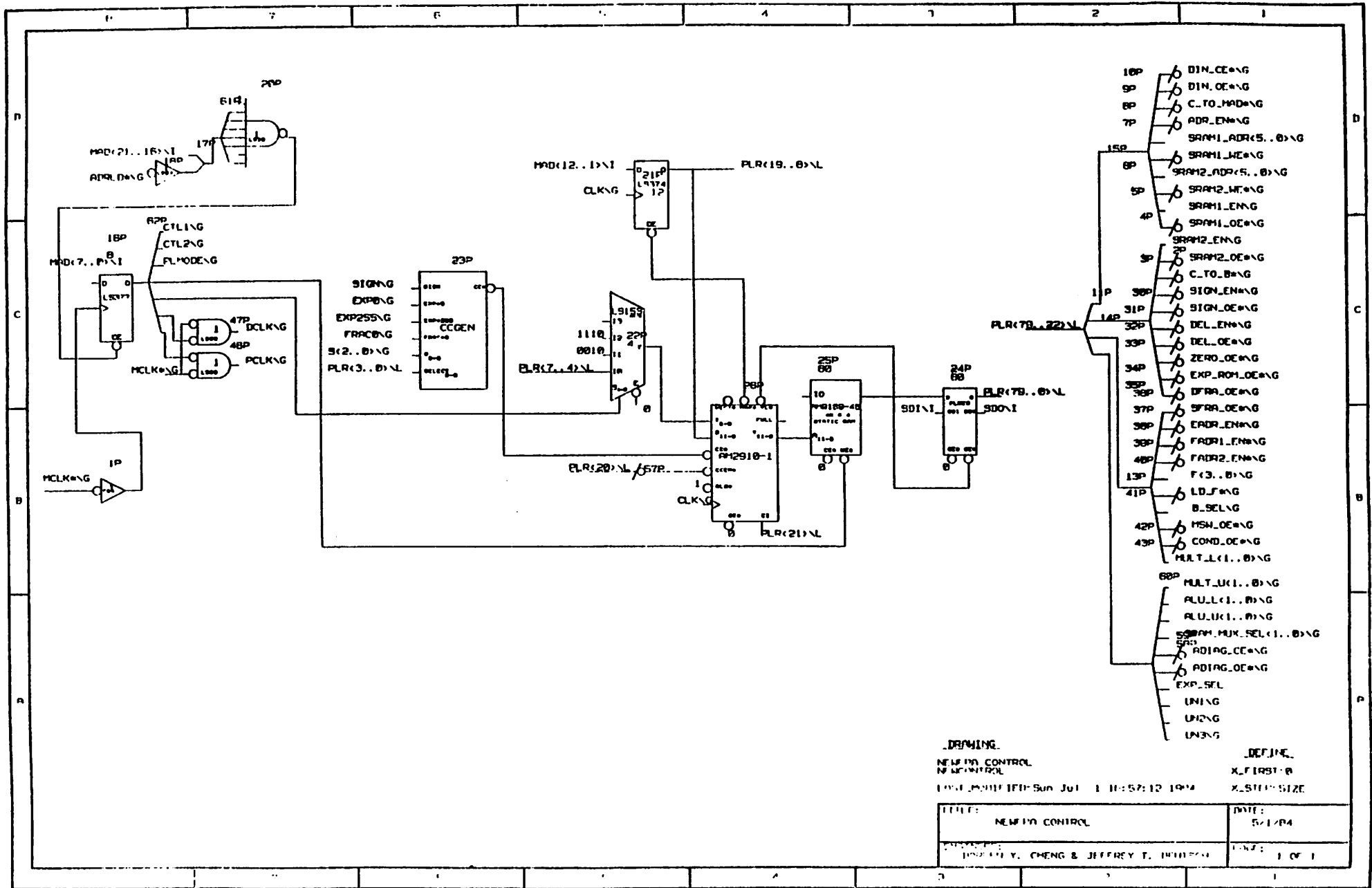
LAST\_MODIFIED=Sun Jul 1 16:05:23 1994

DEFINE

X\_FIRST=0

X\_STEP=512

FILE:	NEWPPA	DATE:	5/3, 1994
DESIGNED BY:	DORFEN Y. CHENG & JEFFREY J. BRITTON	REV:	1



.DRAWING  
 NEWBIT CONTROL  
 NEWBIT CONTROL  
 100000001 IP: Sun Jul 1 11:57:12 1984

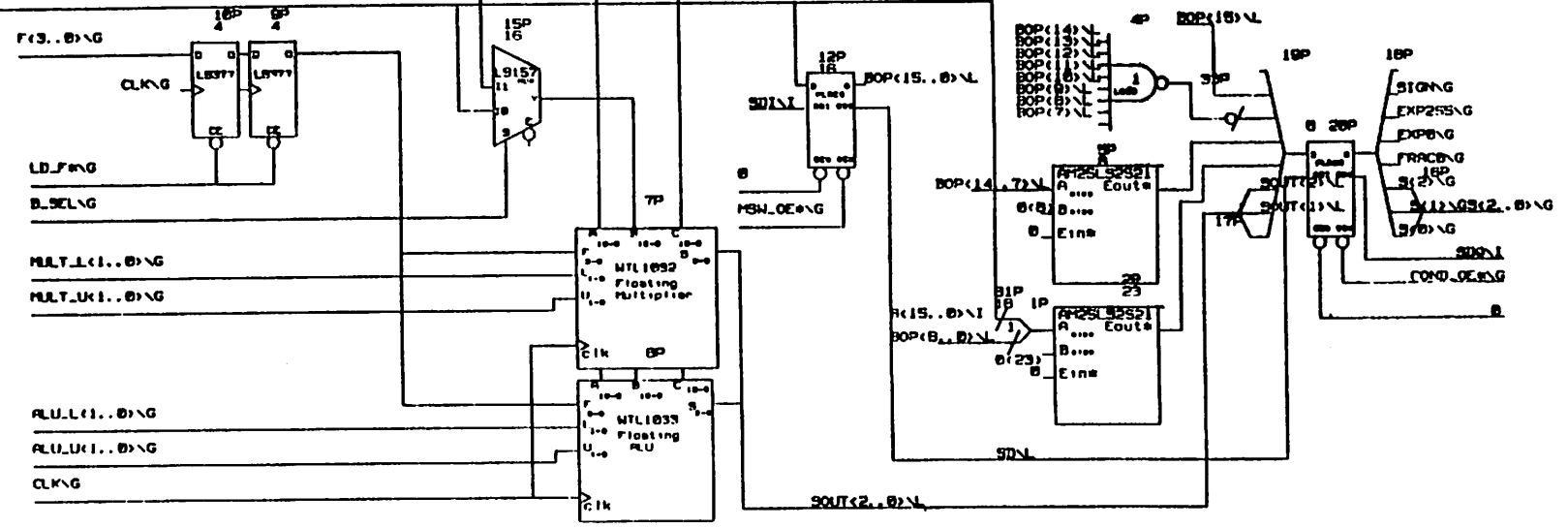
.DEFIN  
 X\_FIRST: B  
 X\_SIZE: 12

TITLE: NEWBIT CONTROL	DATE: 5/1/84
DESIGNED BY: CHENG & JEFFREY T. INTL	SHEET: 1 OF 1

C<15..0>\I

A<15..0>\I

B<15..0>\I



DRAWING

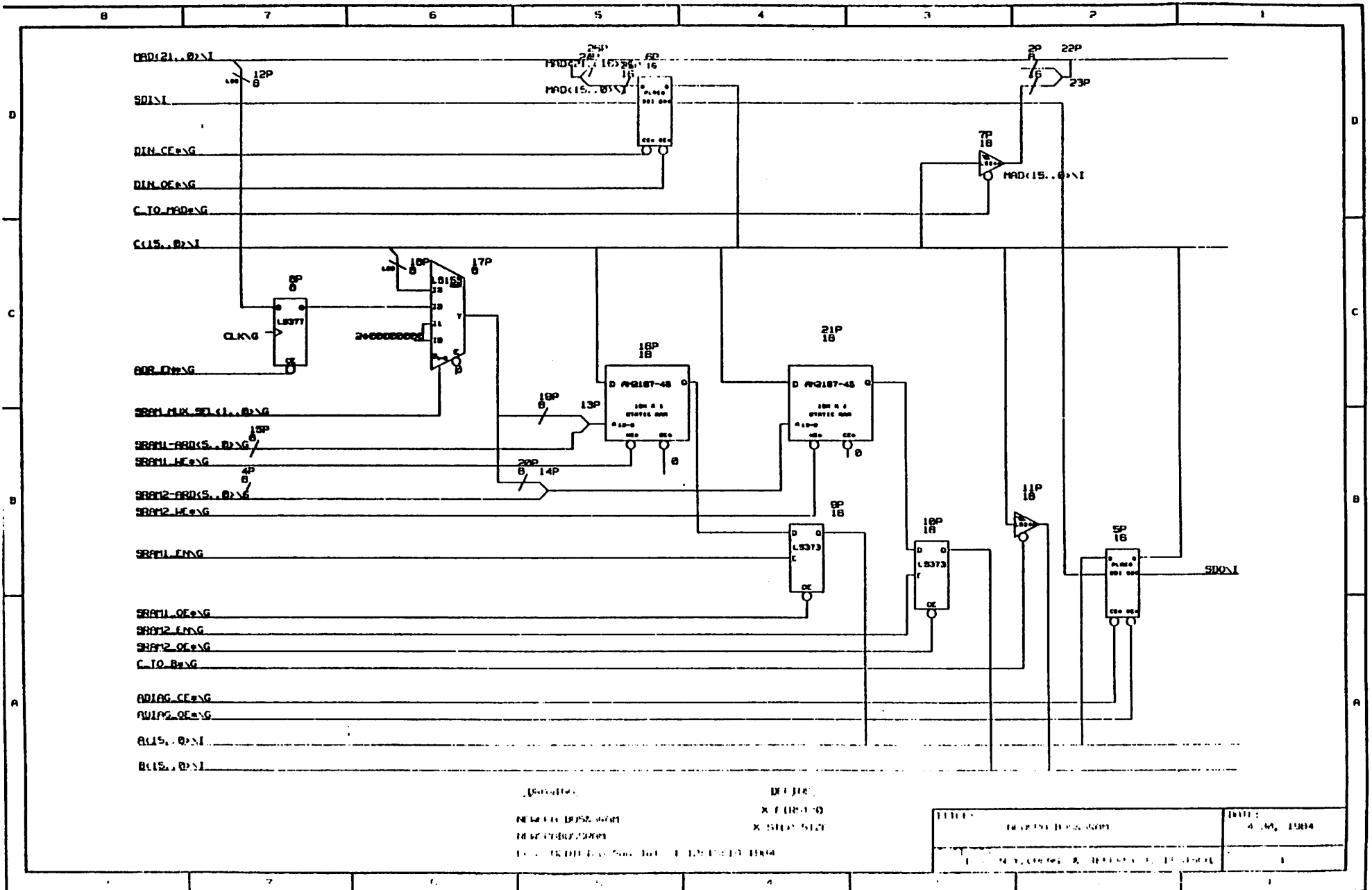
NEWFA ALU  
N.WALU

LAST MODIFIED=Sat Jun 30 17:31:21 1984

DEFINE

X\_FIRST=0  
X\_STEP=SIZE

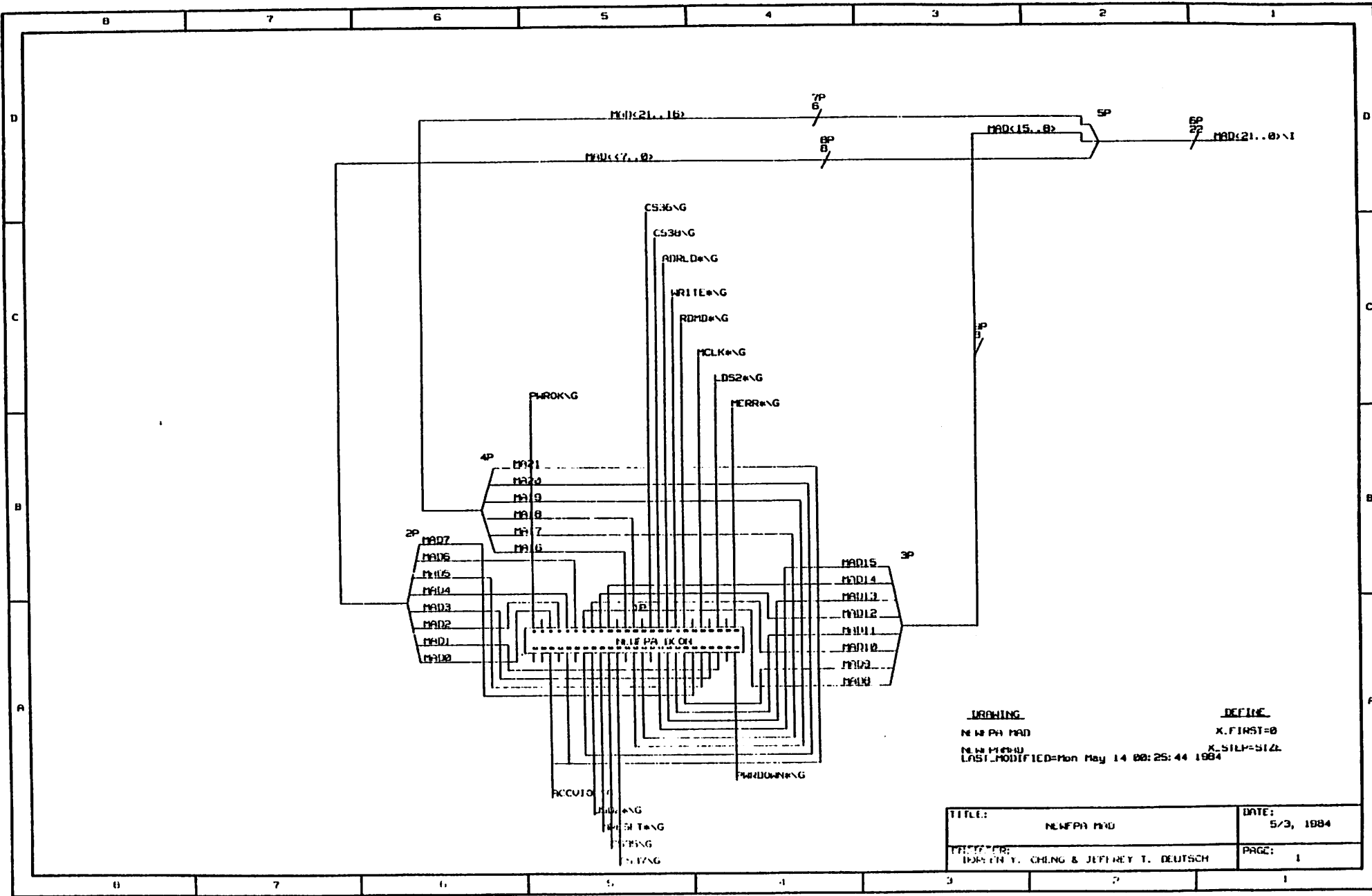
TITLE: NEWFA ALU	DATE: 2/12/84
DESIGNER: MARTIN Y. CHENG & JEREMY T. DEUTSCH	DRAWN: J



(REVISED)      DE 105  
 NEWELL BUS SYSTEM      X 1000 10  
 10000000000000000000      X 1000 10  
 10000000000000000000

TITLE:	NEWELL BUS SYSTEM	DATE:	4.10.1984
DESIGNED BY:	...	CHECKED BY:	...





DRAWING

MEMPA MAD

NEW PCHW

LAST MODIFIED=Mon May 14 00:25:44 1984

DEFINE

X.FIRST=0

K.STEP=SIZE

TITLE:	MEMPA MAD	DATE:	5/3, 1984
DESIGNER:	JOSEPH T. CHENG & JEFFREY T. DEUTSCH	PAGE:	1

## APPENDIX 6

### EXP.C AND LOG.C

This appendix contains the C programs which calculate  $e^x$  and  $\log x$ . These programs are copied from /usr/src/lib.

```

/*  @(#)exp.c      4.1    12/25/82    */

/*
    exp returns the exponential function of its
    floating-point argument.

    The coefficients are #1069 from Hart and Cheney. (22.35D)
*/

```

```

#include <errno.h>
#include <math.h>

```

```

int    errno;
static double  p0    = .2080384346694663001443843411e7;
static double  p1    = .3028697169744036299076048876e5;
static double  p2    = .6061485330061080841615584556e2;
static double  q0    = .6002720360238832528230907598e7;
static double  q1    = .3277251518082914423057964422e6;
static double  q2    = .1749287689093076403844945335e4;
static double  log2e = 1.4426950408889634073599247;
static double  sqrt2 = 1.4142135623730950488016887;
static double  maxf  = 10000;

```

```

double
exp(arg)
double arg;
{
    double fract;
    double temp1, temp2, xsq;
    int ent;

    if(arg == 0.)
        return(1.);
    if(arg < -maxf)
        return(0.);
    if(arg > maxf) {
        errno = ERANGE;
        return(HUGE);
    }
    arg *= log2e;
    ent = floor(arg);
    fract = (arg-ent) - 0.5;
    xsq = fract*fract;
    temp1 = ((p2*xsq+p1)*xsq+p0)*fract;
    temp2 = ((1.0*xsq+q2)*xsq+q1)*xsq + q0;
    return(ldexp(sqrt2*(temp2+temp1)/(temp2-temp1), ent));
}

```

```

/*  @(#)log.c      4.1    12/25/82    */

```

```

/*
    log returns the natural logarithm of its floating

```



point argument.

The coefficients are #2705 from Hart & Cheney. (19.38D)

It calls frexp.

\*/

```
#include <errno.h>
```

```
#include <math.h>
```

```
int    errno;
double frexp();
static double  log2    = 0.693147180559945309e0;
static double  ln10    = 2.302585092994045684;
static double  sqrt2   = 0.707106781186547524e0;
static double  p0      = -.240139179559210510e2;
static double  p1      = 0.309572928215376501e2;
static double  p2      = -.963769093368686593e1;
static double  p3      = 0.421087371217979714e0;
static double  q0      = -.120069589779605255e2;
static double  q1      = 0.194809660700889731e2;
static double  q2      = -.891110902798312337e1;
```

```
double
log(arg)
double arg;
{
    double x,z, zsq, temp;
    int exp;

    if(arg <= 0.) {
        errno = EDOM;
        return(-HUGE);
    }
    x = frexp(arg,&exp);
    while(x < 0.5) {
        x = x*2;
        exp = exp-1;
    }
    if(x < sqrt2) {
        x = 2*x;
        exp = exp-1;
    }

    z = (x-1)/(x+1);
    zsq = z*z;

    temp = ((p3*zsqu + p2)*zsqu + p1)*zsqu + p0;
    temp = temp/(((1.0*zsqu + q2)*zsqu + q1)*zsqu + q0);
    temp = temp*z + exp*log2;
    return(temp);
}
```

```
double
log10(arg)
```

double arg;

{

return(log(arg)/ln10);

}

## APPENDIX 7

### FLOATING POINT SPEED

Included in this appendix is a note by J.T. Deutsch. This note gives floating point times for 68000 software, and several commercial floating point chips.

#### 68000 SOFTWARE

Here are some times for software floating point on 68000 based machines (10mhz 68000, no wait states).

#### MOTOROLA

OP	32-BIT
ADD	18 $\mu$ s
SUB	20 $\mu$ s
MUL	45 $\mu$ s

#### MIT

OP	32-BIT	64-BIT
ADD	206 $\mu$ s	220 $\mu$ s
MUL	328 $\mu$ s	345 $\mu$ s

#### NS 16081 10MHZ

OP	32-BIT	64-BIT
ADD	7.4 $\mu$ s	7.4 $\mu$ s
MUL	4.8 $\mu$ s	4.8 $\mu$ s

## INTEL 80287 5MHZ

OP	32-BIT	64-BIT		
	ADD		14 $\mu$ s	14 $\mu$ s
	SUB		18 $\mu$ s	18 $\mu$ s
	MUL		19 $\mu$ s	27 $\mu$ s
	DIV		39 $\mu$ s	39 $\mu$ s
	SQRT		36 $\mu$ s	36 $\mu$ s
	TAN		90 $\mu$ s	90 $\mu$ s

## FLOATING POINT FUNCTIONAL UNITS

These chips are not designed as co-processors for a particular microprocessor. All have synchronous TTL-level interfaces.

## HP CHIPS

OP	32-BIT	64-BIT	VECTOR
ADD	700ns	1.1 $\mu$ s	(L*300ns)+(400ns/32bits or 800ns/64)
MUL	1 $\mu$ s	1.4 $\mu$ s	(L*600ns)+

## WEITEK CHIPS

OP	32-BIT	VECTOR
ADD	900ns	(L*200ns)+400ns
MUL	900ns	(L*200ns)+400ns

## REFERENCES

- [NAG75]  
L.N. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", University of California Electronics Research Laboratory, Memo ERL-M520, May 1975
- [KNU73]  
D.E. Knuth, "Fundamental Algorithms", Vol. 1, pp.442-445, Addison and Wesley, 1973
- [DEU84]  
J.T. Deutsch, and A.R. Newton, "A Multiprocessor Implementation of Relaxation Based Electrical Circuit Simulation", Proceedings, 21st IEEE Design Automation Conference, Albuquerque, New Mexico, June 1984
- [RET79]  
R.D. Rettberg, C. Wyman, et. al., "Development of a Voice Funnel System: Design Report", BBN Reports #4098, Bolt Beranek and Newman Inc., August, 1979
- [LAW75]  
D.H. Lawrie, "Access and alignment of data in an array processor", IEEE Transactions on Computers, C-24(12), Dec. 1975, pp.1145-1155.
- [RET79a]  
R.D. Rettberg, "Development of a Voice Funnel System: Design Report", Quarterly Technical Report #4143, Bolt Beranek and Newman Inc., June, 1979
- [RET80]  
R.D. Rettberg, "Development of a Voice Funnel System: Design Report", Quarterly Technical Report #4563, Bolt Beranek and Newman Inc., November, 1980
- [RET81]  
R.D. Rettberg, "Development of a Voice Funnel System: Design Report", Quarterly Technical Report #4564, Bolt Beranek and Newman Inc., January, 1981
- [RET82]  
R.D. Rettberg, "Development of a Voice Funnel System: Design Report", BBN Reports #4845, Bolt Beranek and Newman Inc., January, 1982
- [RET83]  
R.D. Rettberg, B. Mann, J. Goodhue, and M. Hoffman, "Chysalis Operating System", Bolt Beranek and Newman Inc., June 1983
- [WAR82]  
F.A. Ware, W.H. McAllister, J.R. Carlson, D.K. Sun, and R.J. Vlach, "64 Bit Monolithic Floating Point Processors", IEEE Journal of Solid-State Circuits, Vol. SC-17, NO. 5, October 1982, pp 898-907.
- [AMD83]  
"Bipolar Microprocessor Logic and Interface", Am 2900 Family 1983 Data Book, Advanced Micro Device Inc., 1983

- [NSD82]  
"NS16081 Floating-Point Unit", Product Review, National Semiconductor Inc., October 1982
- [INT84]  
"80287 80-Bit HMOS Numeric Processor Extension 80287-3", Microsystem Components Handbook, Vol 1, Intel Inc. 1984, pp 4-52 to 4-75.
- [WEI83]  
"WTL 1032/1033 High-Speed 32-bit IEEE Floating-Point Multiplier/ALU", Data Sheet available from Weitek Inc., Sunnyvale, CA., 1983
- [WEI83a]  
"WTL 1032/1033 Floating Point Division/Square Root/IEEE Arithmetic", Application Note, Weitek Inc., Sunnyvale, CA., 1983
- [VLA82]  
A. Vladimirescu and D.O. Pederson, "Performance Limits of the CLASSIE Circuit Simulation Program", Proceedings of Int. Symp. on Circ and Syst., Rome, May 1982
- [CAL79]  
D.A. Calhan and W.G. Ames, "Vector Processors: Models and Applications", IEEE Trans. on Circ. and Syst., Vol. CAS-26, September 1979