MS-CIS-78-22

UNIVERSITY OF PENNSYLVANIA

MOORE SCHOOL

THE DESIGN OF THE PARALLEL ARITHMETIC UNIT IN PEPE

MARK CAMILLO DIVECCHIO

Presented to the Faculty of the College of Engineering and Applied Science (Department of Computer and Information Sciences) in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

Philadelphia, Pennsylvania
August 1978

John W Carr III
Dr. John W. Carr III

Dr. A. K. Joshi

1.0 The Parallel Element Processing Ensemble (PEPE)

1.1 Overview

As the need for solutions to high speed processing problems increases, new and novel architectures will be developed. PEPE is an attempt to solve a specific real-time problem with use of special purpose hardware.

The Ballistic Missile Defense Advanced Technology Center in Huntsville, Alabama started development studies in the late 1960s for a processor to offload the ever increasing demands on a serial processor operating in a ballistic missile defense (BMD) environment.

PEPE's design specifically addresses the BMD problem. It was designed to perform three basic functions previously allocated to a CDC 7700 system. They are:

- Correlation of new radar returns with current tracks
- Track prediction on all current tracks
- Scheduling on a radar time/power line, pulses to acquire more data on current or new tracks.

To solve these three problems, a unique multiprocessor parallel ensemble was proposed. Shown in Figure 1.1, PEPE consists of three independent control processors each commanding an ensemble of up to 288 Processing Elements (PE). The three control processors, housed in a single cabinet called a Control Console (CC), are identical with minor exceptions. Each contains:

- Sequential Control Logic (SCL)
- Program and Data Memories
- I/O Units
- Parallel Instruction Control Unit (PICU)

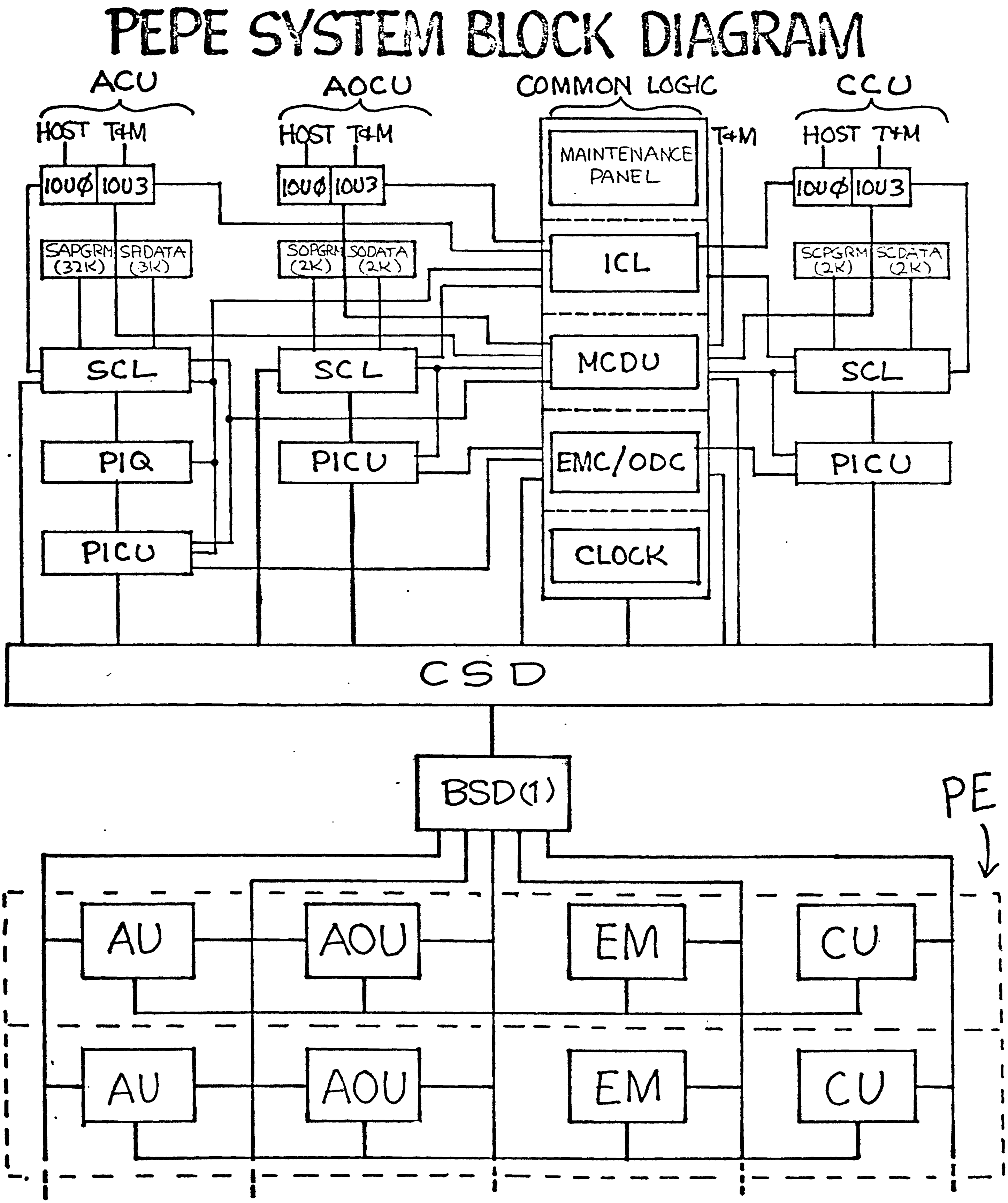


Figure 1.1 PEPE Block Diagram

The SCL is a stand alone processor with all facilities for program execution from memory. It performs typical arithmetic, logical, comparison, and branching functions of a sequential machine. The I/O units perform high speed input and output to the CDC 7700 using a direct channel to both program and data memories.

The PICU receives parallel instructions from the SCL. As the SCL fetches an instruction from memory, it determines if that instruction is targeted for execution in the SCL or in the parallel ensemble. Instructions for the ensemble are sent to the PICU. Described more fully in Section 4.2, the PICU decodes each parallel instruction and transmits control information to all 288 PE.

The three control units are named for their major function. The Correlation Control Unit (CCU) controls the Correlation Unit (CU) portion of the PE. Second, the Arithmetic Control Unit (ACU) handles the Arithmetic Unit (AU) and lastly, the Associative Output Control Unit (AOCU) controls the Associative Output Unit (AOU).

Other major units in the CC are the Intercommunication Logic (ICL), the Output Data Control (ODC) and the Element Memory Control (EMC). The ICL handles communication between the three SCL's and handles interrupt vectoring and masking. The ODC controls access to the main data bus used for outputting data from the ensemble. This bus is a single path shared between all AU and AOU of the ensemble. EMC controls the granting of cycles of the Element Memory (EM) which is shared between processors of the PE.

Figure 1.2 shows the floor layout of a PEPE installation. The CC is centrally located since it must communicate with each of the 8 Element Bays (EB). A Burroughs B1714 computer is the Test and Maintenance (T&M) unit for PEPE. Figure 1.3 is a picture of PEPE installed in Huntsville, Alabama.

1.1.1 The Element Bay

Each EB contains 36 PE as shown in Figure 1.4. The PE are laid out in 4 rows of 9 PE. Each row also contains a clock distribution card. Figure 1.5 is a close up picture of the Element Bay with 11 PE.

One PE consists of 6 large 300 DIP boards, allocated as follows:

• AU	2 boards
• EM	1 board
• CU	1 board
• AOU	2 boards

The Bay Signal Distributor (BSD) occupies the left end of the cabinet. The BSD provides connectors and gates to relay control and data signals from the CC to each PE and to gather output from the PE on the Output Data Bus (ODB) and relay that back to the CC. A simplified layout is shown symbolically in Figure 1.6.

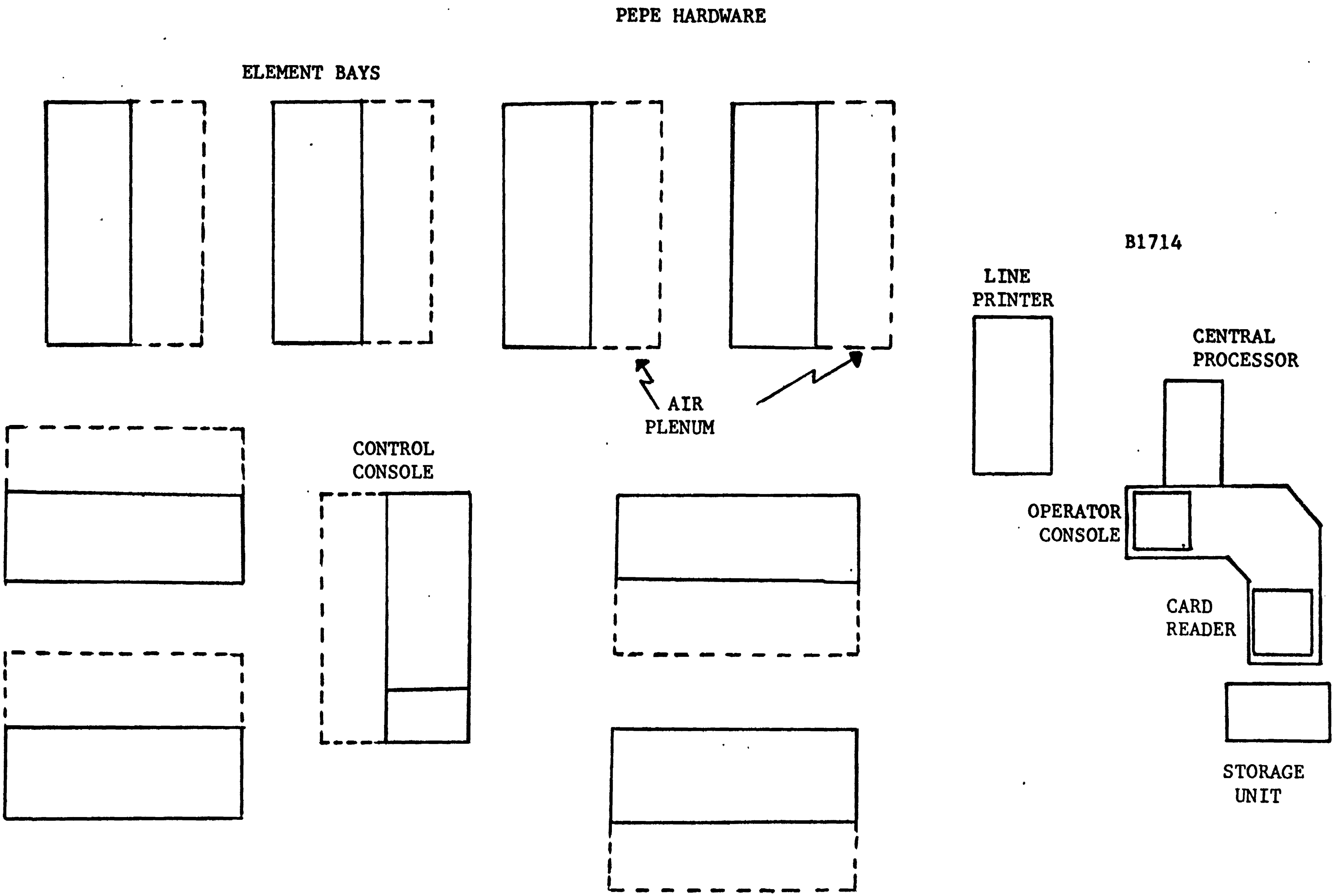


Figure 1.2 PEPE Floor Layout

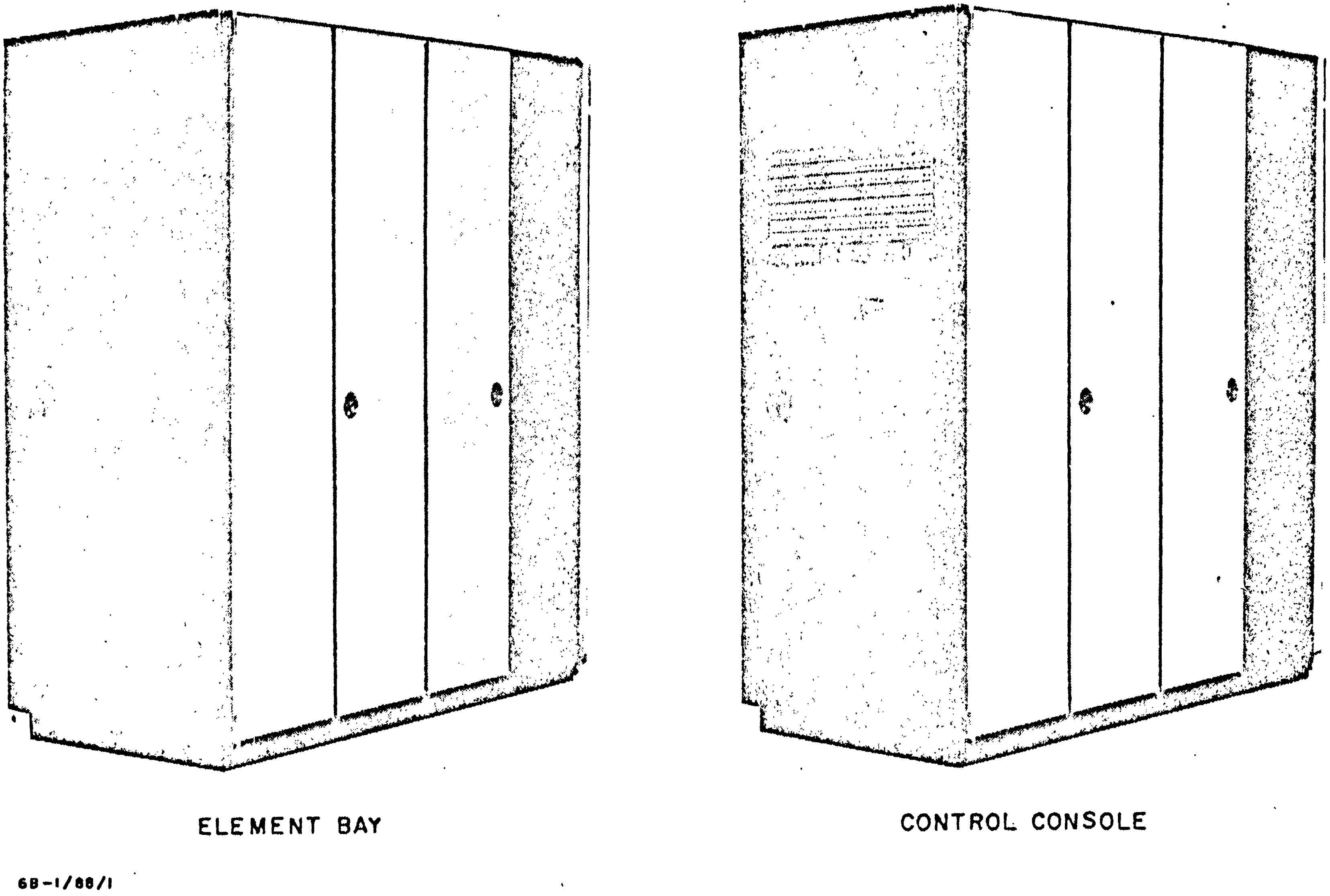


Figure 1.3 PEPE

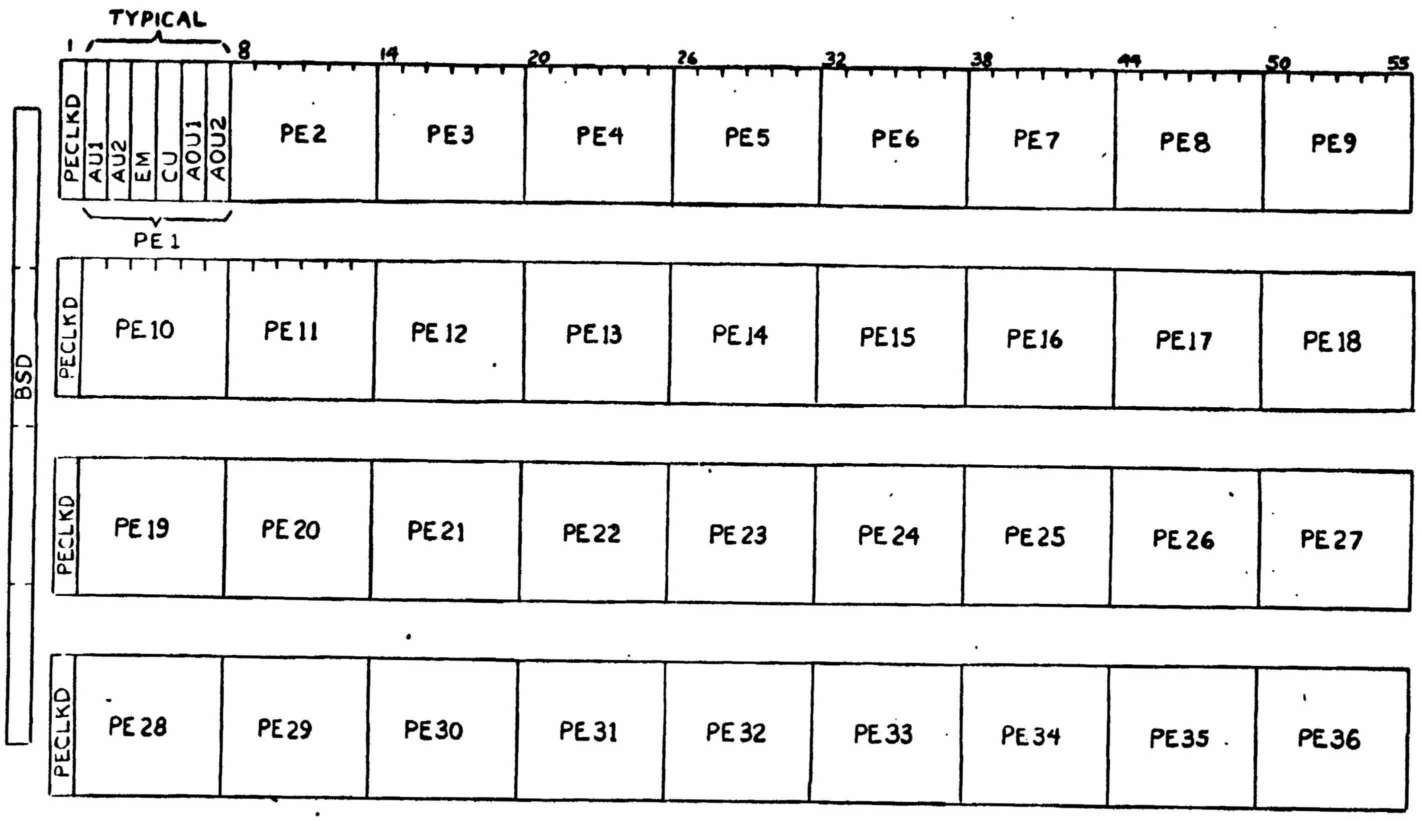


Figure 1.4 PE Bay Layout

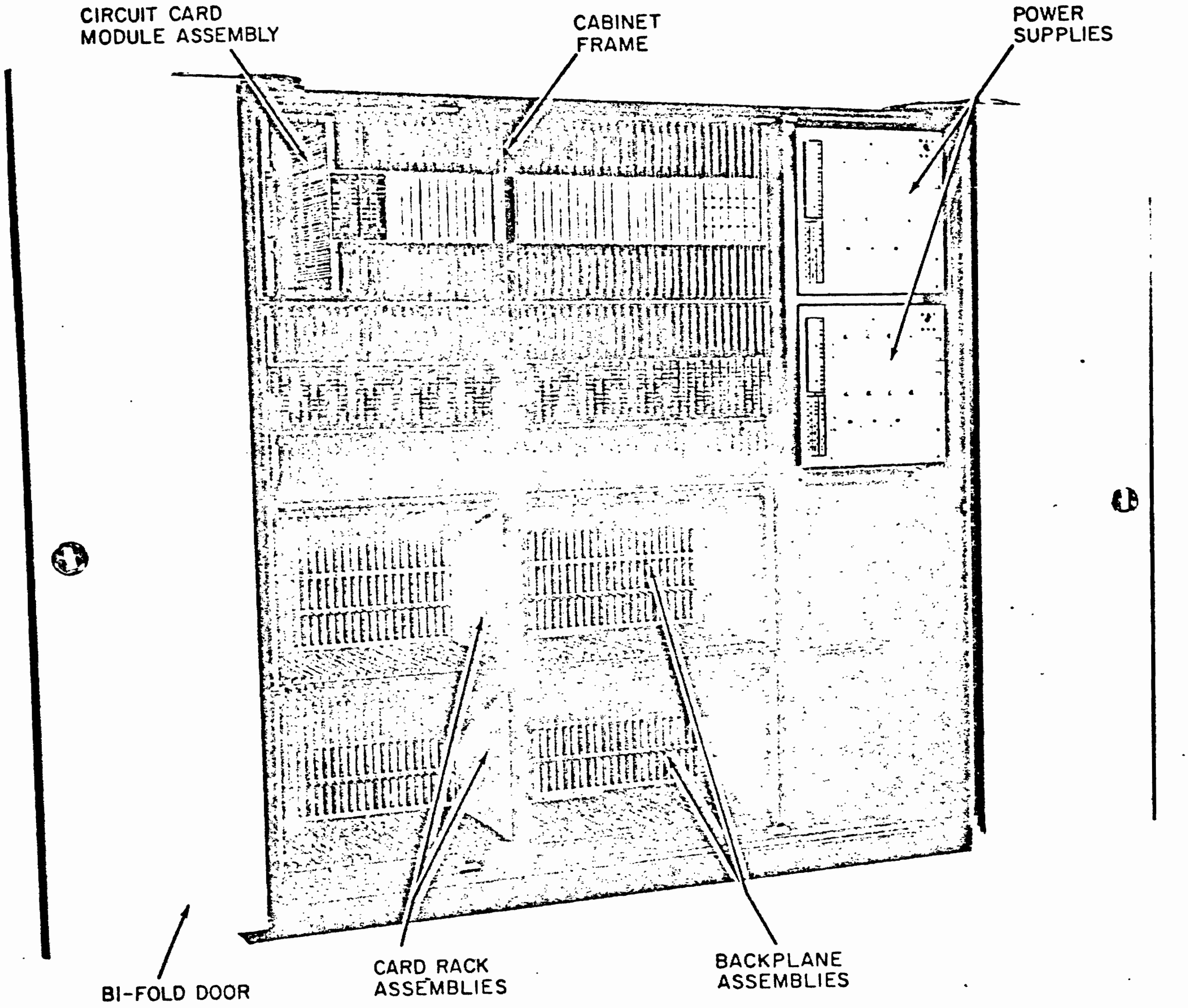


Figure 1.5 PE Bay with 11 PE Installed

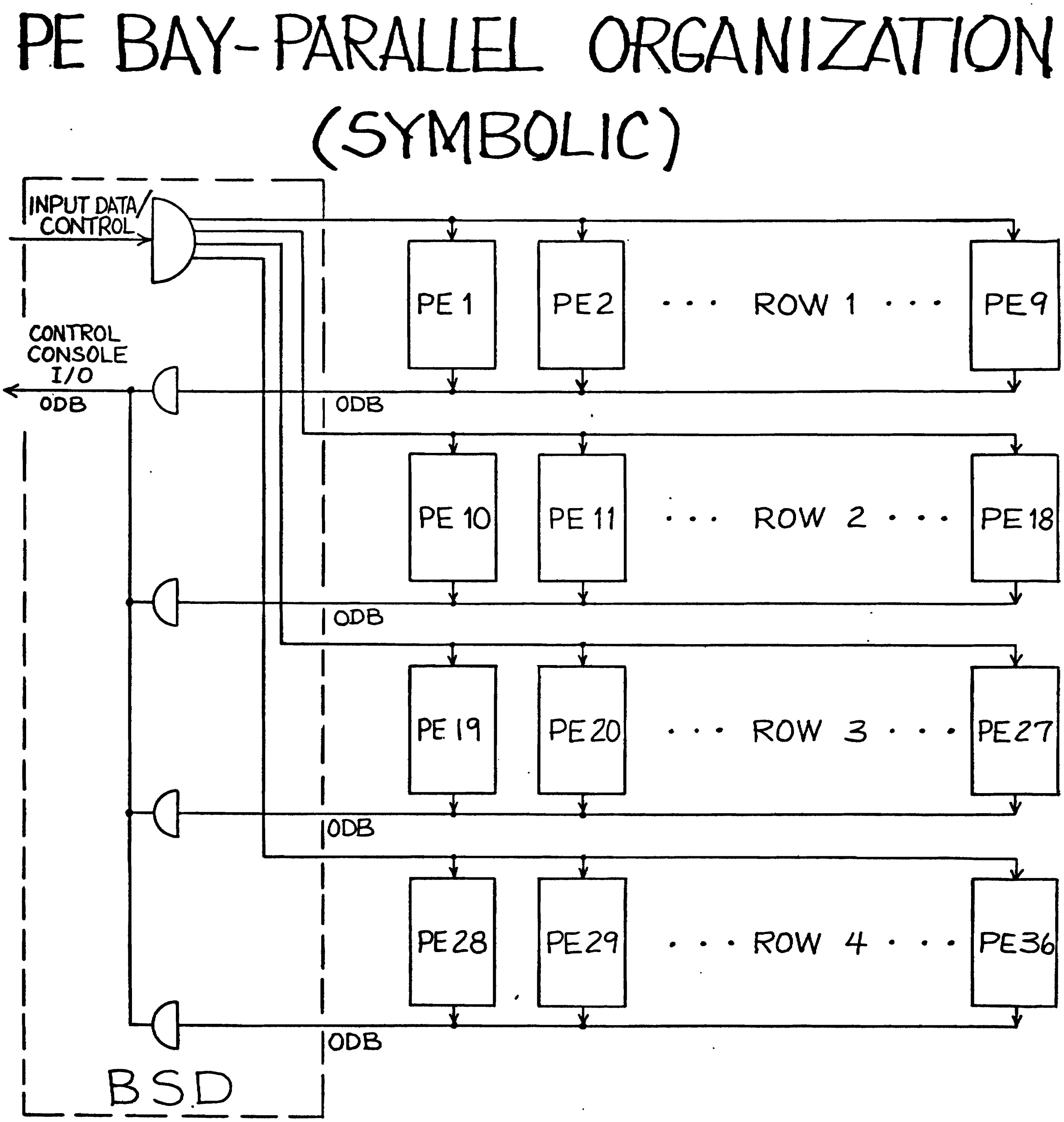
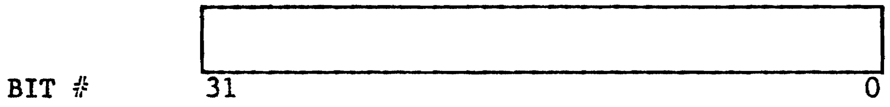


Figure 1.6 Busing Diagram

1.2 Number Formats

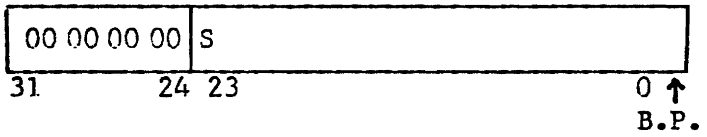
Crucial to the understanding of the operation of a complex floating point processor such as the AU, is a detailed knowledge of the number formats processed by the AU.

The basic PEPE word is 32 bits in length. Bits are numbered right to left with bit 31 being the most significant bit (MSB).



1.2.1 Integer Format

PEPE integers are stored in the lower 24 bits of a word. The upper right bits are always zero.



Bit 23 is the sign bit for the two's complement representation of the number in bits 22-0. The Range of values for integers is

$$+2^{23}-1 \geqslant \text{integer} \geqslant -2^{23}$$

A sign of "1" indicates a negative number and the binary point is considered to be to the right of bit zero.

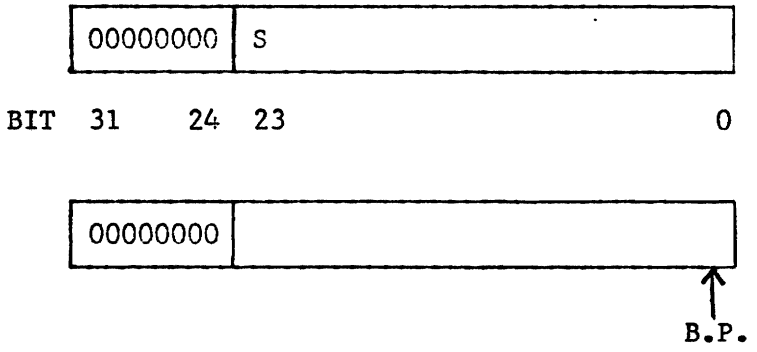
In a two's complement number system, one more negative number may be represented than positive numbers. Consider the following for a four bit system:

BINARY	DECIMAL
0 1 1 1	+7
0 1 1 0	+6
⋮	⋮
0 0 0 0	0
1 1 1 1	-1
⋮	⋮
1 0 0 1	-7
1 0 0 0	-8

Plus seven is the largest positive integer and -8 is the largest negative integer. Since this -8 value does not have a positive "equivalent" we will see later that it requires special handling during the execution of certain PEPE instructions.

1.2.2 Double Integer Format

PEPE double integers are stored in the lower 24 bits of two PEPE words. The upper 8 bits of both words are always zero. Bit 23 of the lower word is always the same as bit 0 of the upper word. Bit 23 of the upper word is the sign bit.



The range of values for double integers is

$$+2^{47}-1 \geq \text{double integer} \geq -2^{47}$$

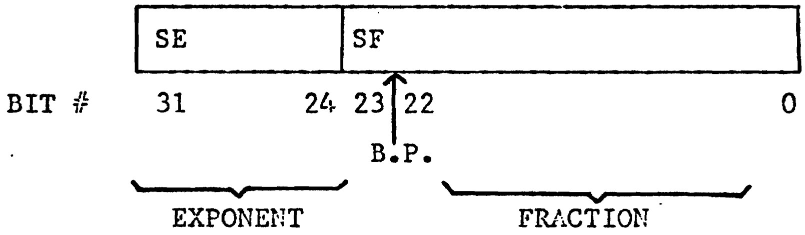
The binary point (B.P.) is to right of bit 0 of the lower word. Double integers less than $+2^{23}-1$ or greater than -2^{23} are completely represented in the lower word as standard PEPE integers.

1.2.3 Logical Format

A PEPE logical word is 32 bits long with no restriction as to its contents.

1.2.4 Floating Point Format

PEPE floating point numbers are represented in one 32 bit word. The fraction (or mantissa) part is stored in bits 23-0. The binary point is to the left of bit 22 and bit 23 is the sign bit of the fraction. The exponent (base 2) is stored in bits 31-24 with bit 31 being the sign bit. Both the fraction and exponent are carried in two's complement notation.



SE - Sign Exponent
SF - Sign Fraction

All floating point numbers are normalized and this is maintained by all PEPE floating point instructions. The number is considered normalized if the most significant bit of the fraction (bit 22) is different from the sign of the fraction (bit 23).

The exponent can be considered as an 8 bit integer field. Range of the exponent field is:

$$127 \geq \text{exponent} \geq -128$$

which can represent decimal values from (approximately):

$$10^{38} \geq \text{floating point} \geq 10^{-38}$$

and

$$-10^{-38} \geq \text{floating point} \geq -10^{38}.$$

The range of the fraction is:

$$2^{0} - 2^{-23} \geq \text{fraction} \geq -2^{0}$$

which can represent decimal values from (approximately):

$$0.9999998^{+} \geq \text{fraction} \geq -1.0000000.$$

The smallest magnitude fraction is $.1192 \times 10^{-6}$. The fraction is always less than magnitude one except for one special case of -1.0000000. This results from the same effect as discussed under integer format. PEPE floating point format allows seven significant decimal digits in the fraction.

1.2.4.1 Floating Point Zero

In any floating point system, a mantissa of zero with any exponent has a value of zero. Thus:

$$0.0X2^{127} = 0.0X2^{o} = 0.0X2^{-128}$$

PEPE hardware must, though, be able to determine equality of numbers that may both be zero. For the hardware to know that $0.0X2^{o} = 0X2^{10}$, poses difficulties that can be avoided. In PEPE, we define one floating point zero. It is:

$$0.0X2^{-128}$$

In a 32 bit word, this is a one in bit 31 and zeros in bits 30-0. All PEPE floating point instructions expect operands of zero in this form and will produce results of zero in this form. This representation can be considered the "smallest" zero in floating point format.

2.0 AU Architecture at the Register Level

The Arithmetic Unit is designed to perform floating point and integer arithmetic, boolean operations, shift operations, single-bit memory data packing, overflow detection and normalize operations. Two's complement arithmetic is used. Double precision integer add and subtract instructions can be accomplished through software.

The AU also is designed to perform element activity instructions. Activity instructions are used for selecting a set or subset of PEPE elements. The selected set is said to be active and will have its activity flip-flop set. Elements whose activity flip-flop is not set will be inactive and will not participate in selected parallel instructions until such time as they are set active.

The ACU controls each AU operation through the use of control lines from the PICU. Global operands are transmitted to the AU on 32 of these lines in a time-sharing mode. In addition, element memory data transfer to and from the AU is done on 32 bidirectional lines. All data output for global use is available on 32 output data lines which are time-shared with AØU data output under global command from Output Data Control (ØDC).

Each AU also provides two outputs for the Select Highest/Lowest logic. For observation of unit faults and as a programming aid, two overflow indicators are sent to the ICL.

The AU contains a fault flip-flop which can be set by the ACU, conditional on element activity. When this flip-flop is set it will not allow the AU, AØU, and CU of the element to participate in any instruction other than "clear fault flip-flop", which will re-enable the failed elements.

Shown in block diagram form in Figures 2.1 and 2.2, the AU was built on two boards named AU1 and AU2. With the exception of the input receivers, output drivers, and a data alignment network, the AU can be functionally described as being composed of an arithmetic and an element activity section.

The arithmetic section contains three working registers (A, B, and Q), an adder, and data input switches. The B register is not programmer accessible. The activity section contains a tag register, a stack (shift) register, and an element activity flip-flop. Also included is the necessary logic and switching which operate in conjunction with these components.

The register level diagram of the AU shows all of the main AU data paths and control. Not shown is the detail logic of functions such as the normalize decode network, zero and overflow detection, and the element activity. These items will be discussed in Section 4. The design is based on specific arithmetic algorithms described in Section 3.

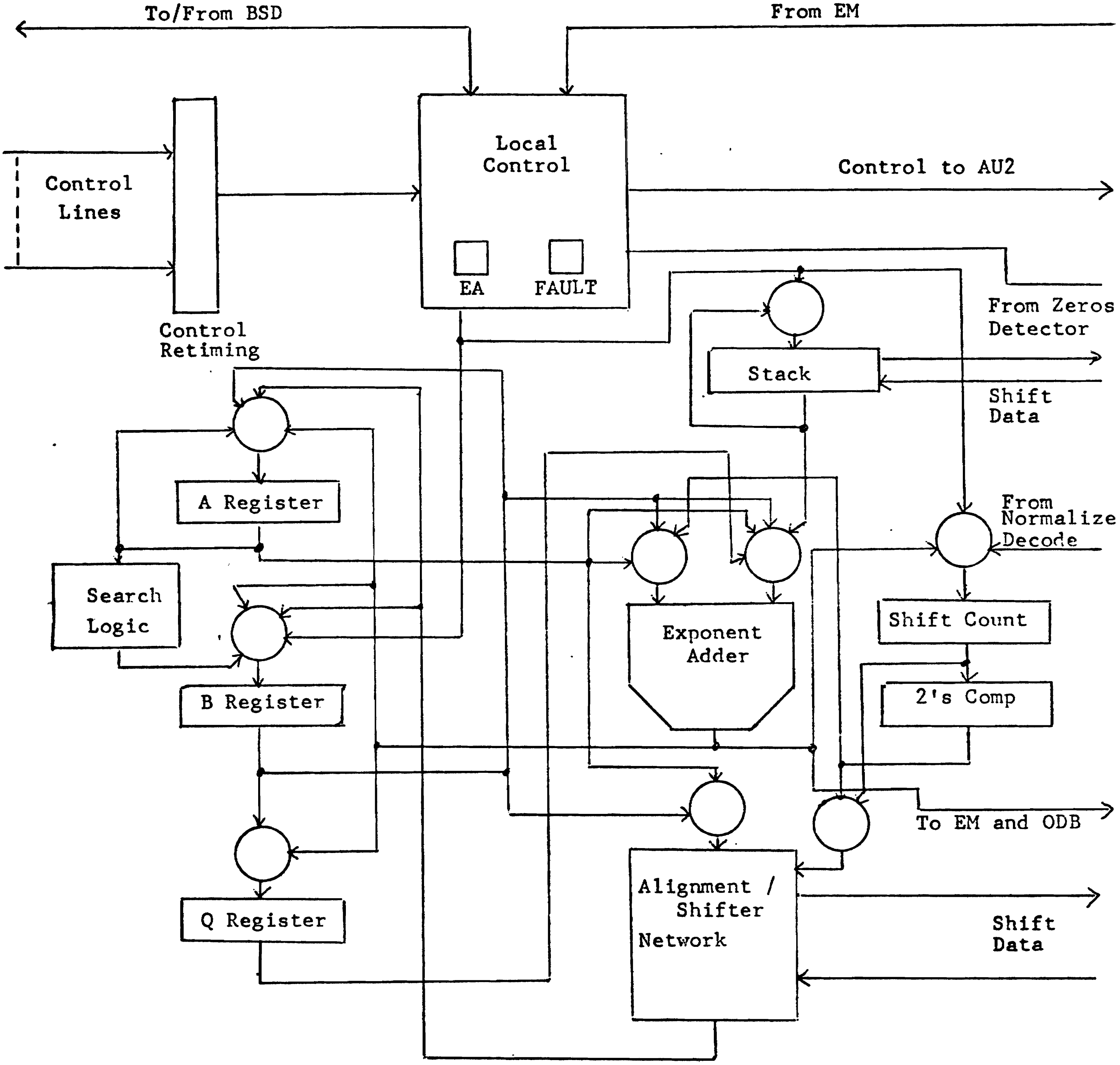


Figure 2.1 AU Card 1

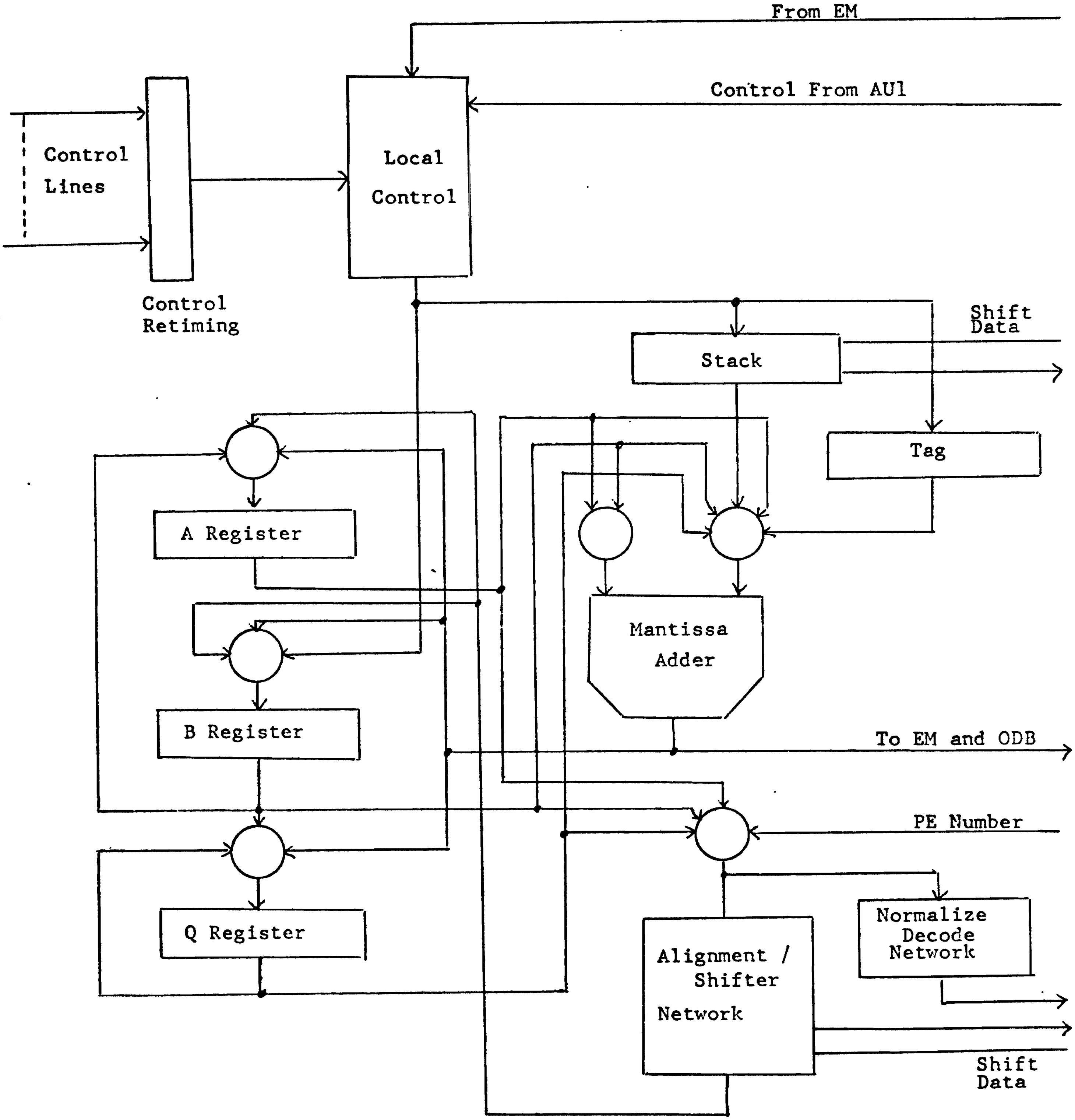


Figure 2.2 AU Card 2

2.1 AU Registers

The AU contains the following registers and flip-flops visible to the programmer:

A Register (PAAREG) - Implicit operand in most instructions. 32 bits in length.

Q Register (PAQREG) - Quotient register for the divide instruction. Used as a scratch register in the multiply instruction. 32 bits in length.

Element Activity Flip-Flop (PAEACT) - A one bit register used to indicate the state of the AU. If reset, the AU will not participate in the execution of most instructions.

Activity Stack (PASTAK) - A 21 bit FILO stack used to store the PAEACT bit. An attempt to push a bit out of the bottom of stack will send an error indication to the ICL.

Tag Register (PATAGR) - An Eight bit register used to hold a tag loaded by the control unit. The tag can be used to permit or inhibit the execution of instructions in the AU. Different AU can contain different tags.

Overflow Flip-Flop (PAOVFF) - A one bit register used to indicate if an arithmetic overflow occurred within the AU. This flip-flop is sent as an error indicator to the ICL.

Fault Flip-Flop (PAFALT) - A one bit register used to completely disable the entire Processing Element. Programmatically removes PE from the Ensemble.

Other registers not visible to the programmer are:

B Register (PABREG) - Used to hold the explicit operand during most instructions. It holds the operand after receiving it from the control unit or the Element Memory. 32 bits in length.

Shift Count Register (PASHCR) - A six bit register used to hold the shift amount. Possible values are +31 to -32 with positive numbers indicating a right shift and negative numbers a left shift.

2.2 AU Functional Units

Major functional units of the AU are:

Adder (PAADMN bits 23-0 and PAADEX for bits 31-24) - A twos complement adder divided into a mantissa section and an exponent section. It is capable of both arithmetic and logical operations. Output is PAAOUT.

Alignment Network (PAALNO) - A 32 bit barrel shift network capable of 32 bit left and right logical shifts and 24 bit left and right arithmetic shifts. Input is PAALNI.

Shift Count Two's Complement (PA2SCO) - A six bit subtractor which subtracts PASHCR from zero. It is used to take a positive shift count and negate it to permit left shifts.

Normalize Decode Network (PANRMD) - A combinatorial network which counts the number of leading zeros or ones in a floating point fraction. Used to produce a count of 0 to 23 to left shift the fraction for normalization during floating point instructions. Shift count output is PANRMD, a 5 bit positive value.

Search Conversion Logic (PASRCH) - Logic used in the conversion micro-step of the Select Highest/Lowest instructions.

3.0 Instruction Set of the AU

The PEPE Arithmetic Unit can execute 71 Instructions that can be classified into six basic types:

1. Activity - those instructions which affect the Element Activity Flip-Flop or Activity Stack.
2. Integer - those instructions involved with the processing of integer data.
3. Logical/Data Transfer - those instructions involved with the processing of logical data.
4. Floating Point - those instructions involved with the processing of floating point data.
5. Output - those instructions which cause the AU to transmit data to the EM or the ACU.
6. Distributed - those instructions which operate over a set of AU's rather than within a single AU.

Appendix C lists the instruction set in several different formats and classes each into one of the six types. The following sections will describe each type in general and will describe in detail the more interesting instructions of each type.

3.1 Activity Instructions

As mentioned in Section 2.0, the Element Activity (EA) Flip-Flop controls whether its particular AU will participate in the currently broadcast parallel instruction. Maximum use of this flip-flop is made to facilitate PEPE's associative nature. The select instructions cause the EA to set or reset depending on the result of an internal comparison between two pieces of data. For example, during Select on Not Equal Global (SNG-107), a comparison is made between a global operand and the AU A Register. The EA is reset in all AU performing an equal comparison.

3.2 Integer Instructions

3.2.1 Single Integer

The AU integer instructions perform addition, subtraction and multiplication on the 24 bit integers described in Section 1.2.1.

The implicit operand is always the A Register. The explicit operand can be global or it can be fetched from the local EM. The Arithmetic Logic Unit (ALU) performs only addition. Subtraction is accomplished by complementing the subtrahend, introducing a carry into the lower order bit and then adding. Integer multiply is described in Section 3.4.3.

Instructions are: ADI

SBI

MLI

All AU integer instructions check for overflow. During addition, overflow is generated if the result sign is different than the two operand signs if they are the same. During subtraction, overflow is generated if the result sign is different than the minuend sign if the minuend sign is different from the subtrahend sign. The overflow signal is sent as an error interrupt to the ICL.

3.2.2 Double Integer

The AU can process double integer values using a sequence of instructions. First, the lower words must be fetched. The operation can then be performed and the result stored. Then the upper words are fetched, operated on and stored. A flip-flop is used to store a borrow or carry between the lower and upper words.

Instructions are: LADI
LSBI
UADI
USBI

Overflow is checked when processing upper words.

3.3 Logical/Data Transfer Instructions

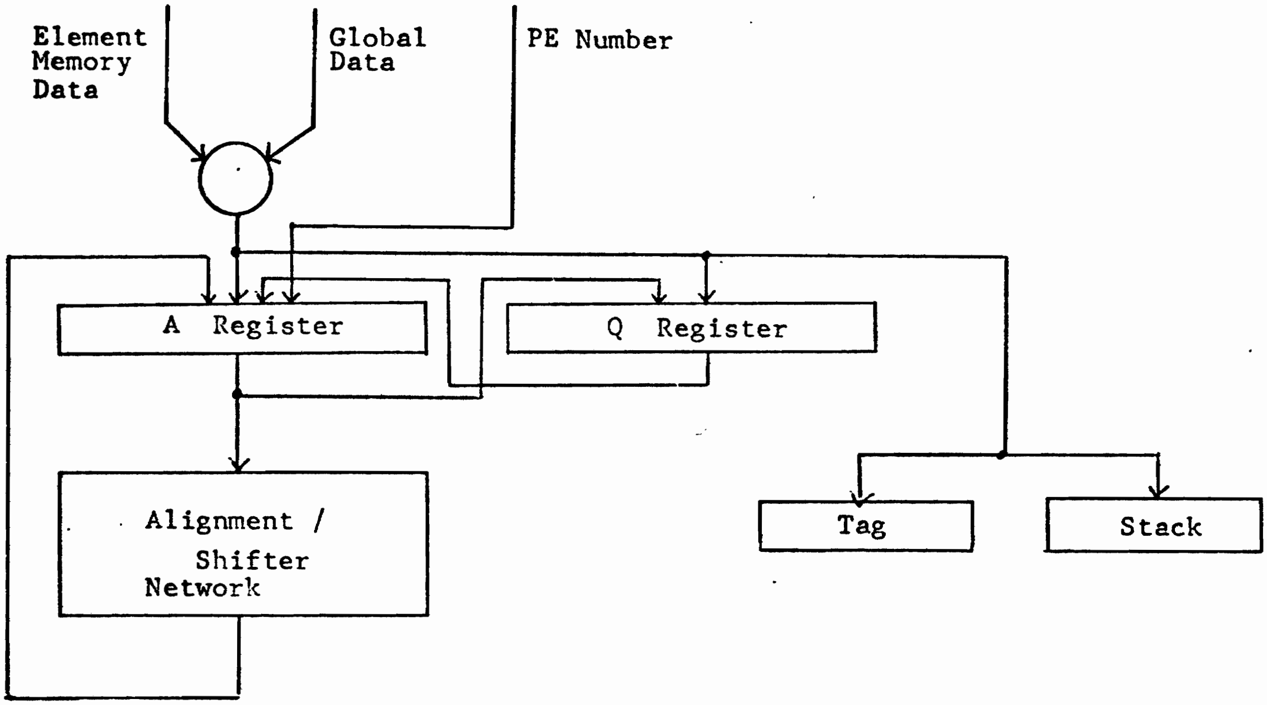
This class of instructions includes both those that perform logical (AND, OR) operations and those that cause the movement of data.

3.3.1 Logical Instructions

The AU performs logical instructions through the ALU. The A Register is one operand and the B Register is loaded with the other operand. As in most instructions the B data can be global or it can be fetched from the local EM. The result is placed in the A Register.

3.3.2 Data Transfer Instructions

These instructions are classed as logical instructions because they handle data as 32 bit words without regard to their internal format. Data transfer can occur over the paths shown in Figure 3.1.



Instructions:

LDA
LDE
LDQ
LTAG
SHL
STG
STGA
STGI
TAQ
TIDA
TQA

Figure 3.1 User Controlled Data Paths

3.4 Floating Point Instructions

This class of instructions make up the more interesting operations possible on PEPE. They operate on the PEPE floating point (FP) data representation described in section 1.2.4.

3.4.1 Float/Fix

The Float instruction (FLOT) takes a 24 bit PEPE integer and converts it into a floating point number. This is done by normalizing and adding an exponent.

FLOT is performed in three steps as follows:

1. Count leading "zeros" for positive numbers (leading "ones" for negative) starting at bit 22 down.
2. Shift the 24 bit integer the counted number of places left (filling with "zeros") thus normalizing it.
3. Subtract the count from 23 to form the exponent of the new FP number.

The subtraction from 23 is required because in the conversion, the binary point is moved 23 places from the right of bit 0 to the right of bit 23. All integers can be converted to FP.

The Fix (FIX) instruction is more complicated because not all FP numbers can be represented as integers. FIX takes a FP number and converts it to an integer. Consider that the PEPE FP number can be mathematically represented by:

$$\pm M + X/Y$$

where M is a whole part and X/Y is the fraction part. FIX is defined to extract the whole part. For example:

5.3	when FIXed becomes	5
5.8	"	5
-5.3	"	-6
-5.8	"	-6

The last two examples are so because -5.3 in two's complement form is really -6 + .7.

The FIX algorithm executes as follows:

1. First check the exponent. If it is greater than 23, we cannot fix the number. An overflow indication is sent to the ICL.
2. Subtract the exponent from 23. Save this number as a shift count.
3. Shift the fraction right extending bit 23, the sign bit. Zero out the exponent.
4. If any significant bits were shifted out, add one to the new integer if it was negative.

The fourth step handles the two's complement negative numbers.

3.4.2 Add/Subtract

FP add (ADD) and subtract (SB) are identical except for the actual arithmetic operation performed.

The ADD algorithm is:

1. The two exponents are subtracted and the result is saved as a shift count.
2. The fraction with the smaller exponent is shifted in order to make the exponents equal.
3. The add or subtract is performed.
4. If the fraction sum overflows, an attempt is made to right shift the fraction and increment the exponent to correct the overflow.
5. If the exponent overflows, an error indication is sent to the ICL.

If the exponent difference is greater than 23, one of the fractions will be shifted completely out of the register. This is not an error condition but must be handled specially (because the shifter is not defined to operate with a greater than 23 arithmetic shift). The result of the exponent subtraction is held in two flip-flops which cause the A or B input to the ALU to be zeroed during step 3.

An ADD example:

$$\begin{array}{rl} & .1 \text{ X } 2^{30} \\ + & \underline{.1 \text{ X } 2^{28}} \end{array} \quad \text{becomes:} \quad \begin{array}{rl} & .1 \text{ X } 2^{30} \\ + & \underline{.001 \text{ X } 2^{30}} \\ & .101 \text{ X } 2^{30} \end{array}$$

also:

$$\begin{array}{rl} & .1 \text{ X } 2^{30} \\ + & \underline{.1 \text{ X } 2^{1}} \end{array} \quad \text{becomes:} \quad \begin{array}{l} .1 \text{ X } 2^{30} \\ \underline{.0 \text{ X } 2^{30}} \\ .1 \text{ X } 2^{30} \end{array}$$

3.4.3 Multiply

Several multiply algorithms were considered for the AU.

The simplest is a bit by bit algorithm. That is, if a bit is "one" in the multiplier, add the multiplicand to a partial sum, shift it left one place and proceed to the next bit.

For example:

```
         00101    = 5
      X  00101    = 5
 1       add multiplicand, shift            00101
 0       add       0     , shift           00000
 1       add multiplicand, shift          00101
 0       add       0     , shift         00000
 0       add       0     , shift        00000
                                        _________
                                        000011001
                                          = 25
```

This method requires one step per bit in the multiplier. In the AU it would have required 24 steps.

A second method, as an extension of the first, is to take 2 bits of the multiplier at a time.

multiplier bits	action	
00	add 0 ,	shift two
01	add 1 X multiplicand,	shift two
10	add 2 X multiplicand,	shift two
11	add 3 X multiplicand,	shift two

Repeat this for each pair of bits in the multiplier.

For example:

$$\begin{array}{rl} & 00101 = 5 \\ \text{X} & \underline{010\underbrace{01}_{}} = 9 \end{array}$$

01	add multiplicand	, shift two	00101
10	add 2 X multiplicand,	shift two	01010
00	add 0	, shift two	00000

$$\begin{array}{r} 000101101 \\ = 45 \end{array}$$

This method needs one step per two bits in the multiplier and in the AU could operate in 12 steps. The major difficulty with this algorithm is the generation of the 3X multiplicand. 0 and 1X are easy to generate. 2X is just the multiplicand shifted left one place. We can, of course, get 3X by adding 1X and 2X but this adds a step to the execution.

The algorithm used in the AU is a modified version of the two bit multiply without the need for a 3X operand. The AU takes the two bits plus a "look ahead", one bit of the next pair. From these three bits add to the partial sum:

000	0	, shift two
001	1X multiplicand,	shift two
010	1X multiplicand,	shift two
011	2X multiplicand,	shift two
100	-2X multiplicand,	shift two
101	-1X multiplicand,	shift two
110	-1X multiplicand,	shift two
111	0	, shift two

Thus, if we had a pair of bit groups as follows:

multiplier 0110 = 6

First, extend it right one zero bit and extend it to the left to complete the last group of three

$$\begin{array}{l} 0001\underline{1\underbrace{00}_{}} \\ \quad\ \ \underbrace{}_{}1 \\ \underbrace{}_{}2 \\ \ \ 3 \end{array}$$

Group 1	100	$-2 \times 2^{0} = -2X$
Group 2	011	$2 \times 2^{2} = 8X$
Group 3	000	$0 \times 2^{4} = 0X$
		$6X$

The result is 6 X the multiplicand. The first group served to correct by subtraction, what it knew was an average added by the second group. The 3 X operand becomes:

0011

extend:

$$\underbrace{0\,0\,\underbrace{1\,1\,0}_{1}}_{2}$$

Group 1	110	-1×2^0 =	$-1X$
Group 2	001	1×2^2 =	$4X$
			$3X$

For our earlier example of 5 times 5:

$$\begin{array}{rl} & 00101 = 5 \\ X & 00101 = 5 \end{array}$$

Group 1	010	1×2^0 =	$1X$
Group 2	010	1×2^2 =	$4X$
Group 3	000	0×2^4 =	$0X$
			$5X$

Another example:

$$\begin{array}{rl} & 00101 = 5 \\ X & 01110 = 14 \end{array}$$

Group 1	100	-2×2^0 =	$-2X$
Group 2	111	0×2^2 =	$0X$
Group 3	001	1×2^4 =	$16X$
			$14X$

This method, used in the AU, requires only 12 steps to perform the actual multiplication. The same algorithm operates correctly on both integer and floating point numbers.

3.4.4 Divide

The AU divide (DV) algorithm resembles standard longhand division. The algorithm divides positive numbers by positive numbers only. Therefore, the first thing the AU does is check if one or the other operand (but not both) are negative. If so, it sets a flip-flop to remember and then takes the two's complement of any negative operands. The AU also requires that the quotient be less than +1 (definition of normalized positive fractions). The AU checks this by subtracting the divisor from the dividend and examining the sign of the result.

For example:

$$0.11010 \div 0.10000$$

$$(13/16 \div \tfrac{1}{2} = 13/8)$$

$$\begin{array}{rlr} 0.11010 & \searrow \text{2's comp} & 0.11010 \\ -\ \underline{0.10000} & \nearrow \text{subtraction:} & +\ \underline{1.10000} \\ & & 0.01010 \end{array}$$

A negative result would indicate that the quotient will have a positive sign and a "one" in the MSB of the fraction (thereby less than +1). A positive result would indicate that the quotient will be greater than one (as in the example). To prevent this, the AU will shift the dividend one place right sign extended and increment the exponent by one. Then:

$$0.11010 \text{ X } 2^{0} \text{ becomes } 0.01101 \text{ X } 2^{1}$$

It is no longer normalized but the algorithm can handle that.

For the divide, the AU subtracts the corrected divisor from the dividend and follows these two rules: (by subtracting the divisor, the AU guesses that the quotient bit for this position is a "one").

1. If a "one" appears as the sign bit of the result of the subtraction, that means the guess of "one" as the quotient bit is incorrect. Therefore, the quotient bit is set to "zero", the result of the subtraction is discarded, and the dividend is shifted left zero fill one place.
2. If a "zero" appears as the sign bit of the result of the subtraction, the guess of "one" as the quotient bit is correct. Therefore, the quotient bit is set to "one" and the result of the subtraction is shift left zero fill one place to become the new dividend.

Continuing our example:

```
                 0.01101
  new         +  1.10000          Quotient
dividend         -------            bit
                 1.11101
                 └──────────────→  0.        (First result bit is the
                 0.11010                      sign and will always be
              +  1.10000                      "zero")
                 -------
                 0.01010
                 └──────────────→  1
                 0.10100
              +  1.10000
                 -------
                 0.00100
                 └──────────────→  1
                 0.01000
              +  1.10000
                 -------
                 1.11000
                 └──────────────→  0
                 0.10000
              +  1.10000
                 -------
                 0.00000
                 └──────────────→  1
                 0.00000
              +  1.10000
                 -------
                 1.10000
                 └──────────────→  0
```

Quotient is 0.11010. The result exponent is the dividend exponent minus the divisor exponent. For the example it is 1 - 0 = 1. The final quotient is

$$0.11010 \times 2^1 = 13/16 \times 2^1 = 13/8$$

The AU executes the iterative portion of the DV instruction in 24 steps.

3.4.5 Square Root

In base ten, a square root operation is a set of guesses and subtractions. The guess can be 0 to 9. In longhand decimal

```
find        √104692

                3     2     3
              √10    46    92
   3 X 3       -9
                1    46
  62 X 2       -1    24
                     22    92
 643 X 3            -19    39
                      3    53
```

If the guess of the root digit was too high as evidenced by a negative difference, the digit is decremented by one to become a new guess. The AU also guesses at the result, but in binary only two guesses are possible. A longhand example in binary (Example A):

```
                 1     1     0     1     1     1       A12
               √10    11    11    01    10    10
      1 X 1     -1
                 1    11                               A1
    101 X 1     -1    01                               A2
                      10    11                         A3
   1100 X 0           -0    00                         A4
                      10    11    01                   A5
  11001 X 1           -1    10    01                   A6
                       1    01    00    10             A7
 110101 X 1           -0    11    01    01             A8
                            01    11    01    10       A9
1101101 X 1                 -1    10    11    01       A10
                             0     0    10    01       A11
                                                       [Reference
                                                        Line #   ]
```

The AU guesses that each root bit is a "one", then performs a subtraction. If the guess turns out to be incorrect, the subtraction result is still used but a correction factor is added during the next subtraction. The AU proceeds as follows:

> First, the exponent must be even. If it is not, the fraction is shifted right one place sign extended and one is subtracted from the exponent.

The first two bits of the square are taken. A guess is made that the first bit of the result will be a "one". The guess is subtracted from the operand This result becomes the new operand and is concatenated with the next two bits of the square. The guess is then shifted left one place and the lower three bits are replaced by:

011	if the subtraction is negative
101	if the subtraction is positive.

This becomes the new guess. If the result of the last subtraction was negative, we add the new guess else we still subtract it. The process continues until all bits of the square are used. The root is contained in the last guess word. It is normalized and the exponent is shifted right one place sign extended to divide it by two and the instruction is complete.

For our earlier example:

```
                                                  [Reference]
                                                  [Line #    ]
      ____________________________________
     √10    11    11    01    10    10
     -01 <- first guess
     ---------
      01    11                                     B1
     -01    01                                     B2
            -----------
            10    11                               B3^1
           -11    01                               B4^1
            -----------------
            11    10    01                         B5^1
          + 01    10    11                         B6^1
            -----------------------
            01    01    00    10                   B7
           -00    11    01    01                   B8
            -----------------------------
                  01    11    01    10             B9
                 -01    10    11    01             B10
                  -----------------------
                        00    10    01             B11
   last guess --->      11    01    11             B12
                        \_________________/
        ________________________|
       |
       v
   normalize

   0.110111
```

In line A4 of the longhand method, zero was subtracted because a guess of zero was used as the next bit of the answer. In line B4^1 of the algorithm, a guess of "one" was subtracted (since the algorithm always guesses "one") and as seen in line B5^1, a negative result indicates too large of a guess. Line B6^1 applies the following insertion factor which replace the last three bits of the guess:

```
011
| \_/
|  |
|  Two's complement of the new guess
|
└Correct bit of the root.
```

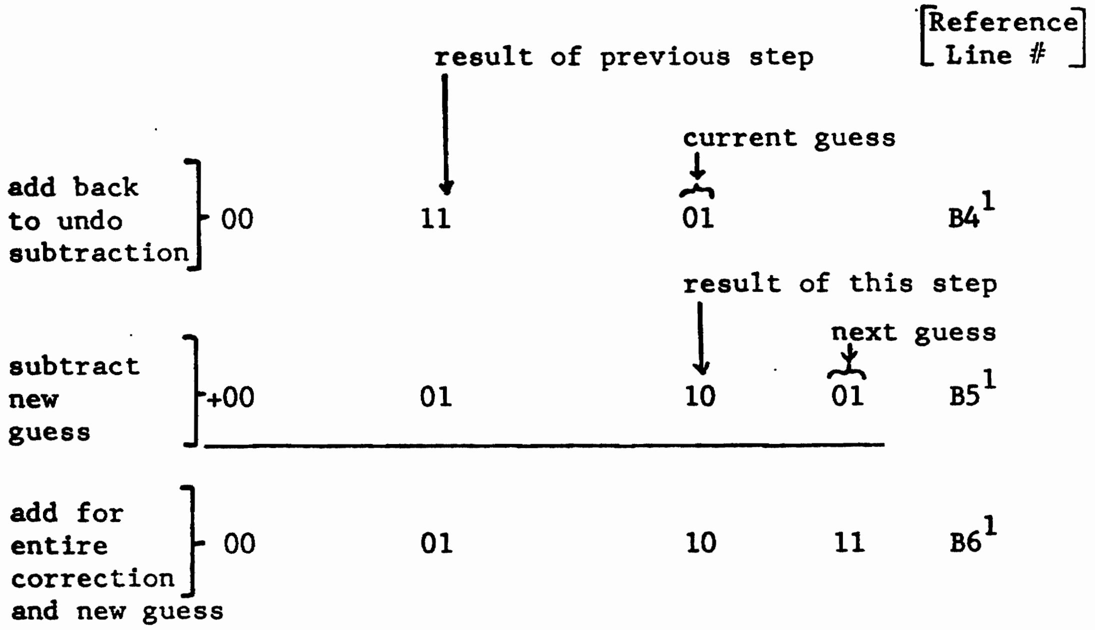
The two's complement of the new guess serves to correct the over subtraction of this step during the next step. The "correction" field used as a result of a correct guess is:

```
101
| \_/
|  |
|  New guess
|
└Correct bit of the root.
```

In the example, the algorithm subtracted:

00 11 01 [B4[1]]

and produced a negative result. Therefore it must add this back during the next step while at the same time subtracting the new guess word.



The AU executes the iterative portion of the square root instruction in 24 steps.

3.5 Output Instructions

3.5.1 Store Instructions (STA)

All of the AU visible registers can be stored in Element Memory. During a single micro-step, the selected data is gated to the EM and written. Storable register are:

A Register

Q Register

Activity Stack

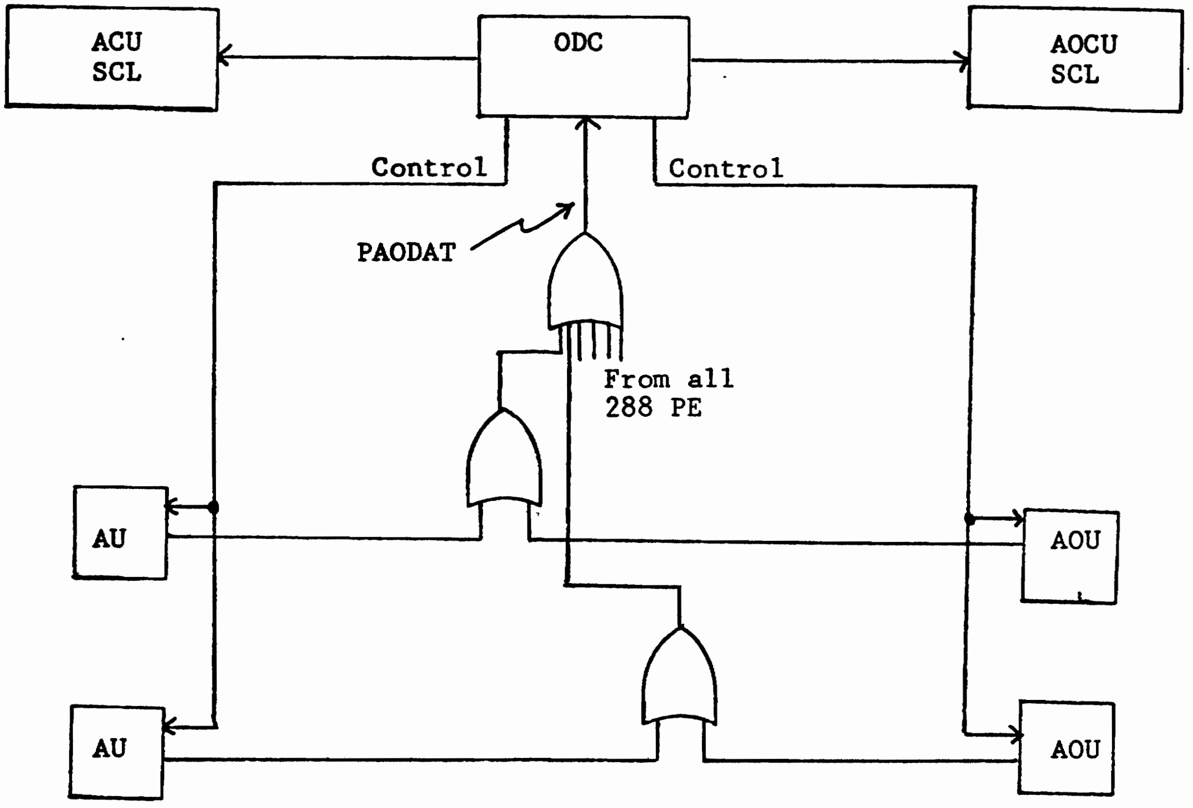
Tag Register

EA Flip-Flop

Overflow Flip-Flop

3.5.2 Output A Register Instruction (OTA)

A special instruction which executes both in the SCL and AU is used to retrieve a 32 bit word from one AU and place it in the SCL A Register. In preparation for the execution of an OTA instruction, appropriate select instructions should have been executed to leave only one AU active. When the SCL decodes an OTA, it sends a copy to the PICU while at the same time requesting use of the Output Data Bus (ODB) from ODC, Figure 3.2. When a bus cycle is granted, the AU gates its A Register onto the bus. After some signal propagation delay, the ODC responds to the SCL with a data ready line and the correct data.



The PAODAT Bus is 32 bits wide

Figure 3.2 Output Data Bus/OTA Instruction Block Diagram

3.6 Distributed Logic Instructions

PEPE also includes a set of logic called distributed logic. This logic is not contained in any one PEPE unit but is spread over the control console and the element bays.

3.6.1 Count AU

The SCL may at any time, request a count of the number of AU is with their EA Flip-Flops on. Special logic exists within each EB and the CC to perform this count. Shown in Figure 3.3, each AU in the ensemble presents the state of its EA to the count logic. The count logic presents a nine bit (0-288) count to the SCL which can be loaded into the SCL A Register during the execution of a Read Activity (RDA) instruction. The AU does not directly participate in this instruction.

Two other signals are generated by the AU Count logic. They are SAMANY and SAACTV. The signal SAMANY is a detection of activity count greater than one or "many" active. The signal SAACTV is also a detection of activity count, here for a count greater than zero. Both of these signals are transmitted to the SCL and SAMANY is also sent to the PICU.

3.6.1 Select Highest/Lowest Instructions

A need was identified in PEPE for a high speed algorithm to perform a maximum or minimum search over a set of data values in the element ensemble. The two search instructions are Select Highest and Select Lowest (SH/SL). The SH/SL instructions execute as follows: in the set of active AU's, reset element activity in each AU whose "A" Register contains a nonmaximal/minimal value relative to the set of active AU. This value comparison is done for the set of PEPE integer or normalized floating point numbers but never for a mix of both types.

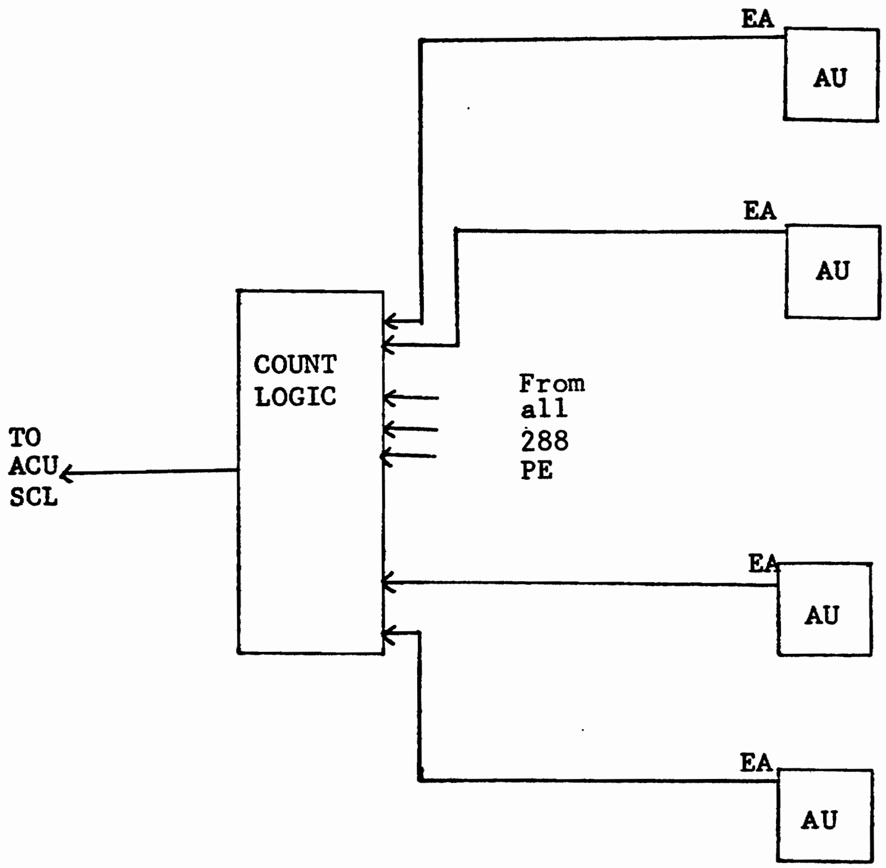


Figure 3.3 Count AU Block Diagram

A discussion of valid operands for the SH/SL may be appropriate here. For simplicity, consider a machine with eight bit registers instead of 32 bit registers. Let four bits represent the fraction or integer and four bits represent the exponent. See figure 3.4. Note that the entire discussion following can be directly extrapolated to PEPE 24 bits fraction/integer and eight bit exponent.

When doing a search over a set of integers, SH/SL requires that the exponent and the sign of the exponent be zero. The integer value is searched for the largest/smallest value.

A search over a set of floating point numbers requires that the fraction be normalized. Normalized numbers are those whose SF and most significant bit of the fraction are different. As shown in figure 3.4, + 1/8 is not a valid floating point number for this reason. If + 1/8 were permitted, the numbers 0010 0001 and 0000 0100 would both have the value of + ½. The search algorithm as described later will report that the first binary pattern is larger than the second when they are actually equal.

Also note that the value of floating point zero can be represented in 16 different ways, that is with any exponent as long as the fraction and SF are all zero. In effect 0111 0000 = 0000 0000 = 1000 0000. Therefore, we define floating point zero to be the "smallest" zero representable or 1000 0000 which is 0 X 2^{-8}. Note that floating point zero is not normalized.

7	6	5	4	3	2	1	0	
0	0	0	0	SI	VALUE			INTEGER
SE	EXP			SF	FRACTION			FLOATING POINT

TWO'S COMPLEMENT NOTATION

SIGN	MAGNITUDE	INTERPRETATION		
		INTEGER	EXPONENT	FRACTION
0	1 1 1	+7	$2^7=128$	+7/8
0	1 1 0	+6	$2^6=64$	+3/4
0	1 0 1	+5	$2^5=32$	+5/8
0	1 0 0	+4	$2^4=16$	+1/2
0	0 1 1	+3	$2^3=8$	+3/8 *
0	0 1 0	+2	$2^2=4$	+1/4 *
0	0 0 1	+1	$2^1=2$	+1/8 *
0	0 0 0	0	$2^0=1$	0
1	1 1 1	-1	$2^{-1}=1/2$	-1/8 *
1	1 1 0	-2	$2^{-2}=1/4$	-1/4 *
1	1 0 1	-3	$2^{-3}=1/8$	-3/8 *
1	1 0 0	-4	$2^{-4}=1/16$	-1/2 *
1	0 1 1	-5	$2^{-5}=1/32$	-5/8
1	0 1 0	-6	$2^{-6}=1/64$	-3/4
1	0 0 1	-7	$2^{-7}=1/128$	-7/8
1	0 0 0	-8	$2^{-8}=1/256$	-1

* Not Valid Normalized Floating Point Patterns

Figure 3.4 A Four Bit Number System

3.6.2.1 Algorithm

The algorithm to be described executes in the AU and requires a maximum of 35 100 ns micro-steps. Execution consists of two parts, Conversion and Search.

3.6.2.2 Conversion

The first micro-step of SH/SL converts the set of values to be compared from the valid set of PEPE 2's complement number system into an ordered set of operands. This conversion is a mapping of PEPE numbers onto a 32 bit pure magnitude number line. In the case of Select Highest, the largest (most positive) PEPE number is converted to all ones and the smallest (most negative) PEPE number is converted to all zeros. For Select Lowest the mapping is reversed.

For the purposes of this explanation, the set of PEPE floating numbers is broken into 4 classes. Refer to figure 3.5. Column 1 represents the ordered set of 2's complement normalized PEPE operands. Top-most positive, bottom-least positive. Fraction sign (bit 23) is first and exponent sign bit 31 next, for both, 0 is positive. The exponent field (bits 30 to 24) is next. Fraction field is last (bits 22 to 0). The classes are:

Class 1 contains all operands with positive fraction and positive exponent.

Class 2 contains all operands with positive fraction and negative exponent.

Note the zero operand is the least class 2 element. Class 3 contains all operands with negative fraction and negative exponent. Here the exponent field is ordered all zeros to all ones. The fraction field is ordered by normalized negative values (zero with all ones to all zeros). This puts the operands in order (most to least positive) but not in decreasing binary

Original Floating Point Number				Select Highest Mapping				Select Lowest Mapping				
SF	SE	EXP	FRAC	SF	SE	EXP	FRAC	SF	SE	EXP	FRAC	
0	0	1 - 1	11 - 1	1	1	1 - 1	11 - 1	0	0	0 - 0	00 - 0	
		:				:				:		
0	0	1 - 1	10 - 1	1	1	1 - 1	10 - 0	0	0	0 - 0	01 - 1	
		:				:				:		CLASS 1
0	0	0 - 0	11 - 1	1	1	0 - 0	11 - 1	0	0	1 - 1	00 - 0	
		:				:				:		
0	0	0 - 0	10 - 0	1	1	0 - 0	10 - 0	0	0	1 - 1	01 - 1	
0	1	1 - 1	11 - 1	1	0	1 - 1	11 - 1	0	1	0 - 0	00 - 0	
		:				:				:		
0	1	1 - 1	10 - 0	1	0	1 - 1	10 - 0	0	1	0 - 0	01 - 1	
		:				:				:		CLASS 2
0	1	0 - 0	11 - 1	1	0	0 - 0	11 - 1	0	1	1 - 1	00 - 0	
		:				:				:		
0	1	0 - 0	10 - 0	1	0	0 - 0	10 - 0	0	1	1 - 1	01 - 1	
0	1	0 - 0	00 - 0	1	0	0 - 0	00 - 0	0	1	1 - 1	11 - 1	(ZERO)
1	1	0 - 0	01 - 1	0	1	1 - 1	01 - 1	1	0	0 - 0	10 - 0	
		:				:				:		
1	1	0 - 0	00 - 0	0	1	1 - 1	00 - 0	1	0	0 - 0	11 - 1	
		:				:				:		CLASS 3
1	1	1 - 1	01 - 1	0	1	0 - 0	01 - 1	1	0	1 - 1	10 - 0	
		:				:				:		
1	1	1 - 1	00 - 0	0	1	0 - 0	00 - 0	1	0	1 - 1	11 - 1	
1	0	0 - 0	01 - 1	0	0	1 - 1	01 - 1	1	1	0 - 0	10 - 0	
		:				:				:		
1	0	0 - 0	00 - 0	0	0	1 - 1	00 - 0	1	1	0 - 0	11 - 1	
		:				:				:		CLASS 4
1	0	1 - 1	01 - 1	0	0	0 - 0	01 - 1	1	1	1 - 1	10 - 0	
		:				:				:		
1	0	1 - 1	00 - 0	0	0	0 - 0	00 - 0	1	1	1 - 1	11 - 1	

Figure 3.5 Number Classes and Conversion
Floating Point

sequence. Class 4 contains all operands with negative fraction and positive exponent. It is ordered similarly to class 3. Column 2 shows the converted values for select highest and Column 3 shows the converted values for select lowest.

The set of PEPE integers is broken into 2 classes. Refer to figure 3.6. Column 1 represents the ordered set 2's complement PEPE integers: top-most positive, bottom-least positive. Fraction sign (bit 23) is first, zero is positive. Exponent sign and exponent are all zero. Fraction field is last (bits 22 to 0). The classes are:

Class 1 contains all positive integers.

Class 2 contains all negative integers.

Column 2 shows converted values for SH, and column 3 shows the converted values for SL. Operation of SH/SL on integer values can be considered a subset of floating point operation when the exponent is zero.

Rules for either floating point or integer conversion are:

<u>Select Highest</u> (SH)

1. Complement bit 31 if bit 23 equals zero
2. Complement bits (30-24) if bit 23 equals one
3. Bits (22-0) remain unchanged
4. Complement bit 23

<u>Select Lowest</u> (SL)

1. Complement bit 31 if bit 23 equals one
2. Complement bits (30-24) if bit 23 equals zero
3. Complement bits (22-0)
4. Bit 23 remains unchanged

Original Integer Number				Select Highest Mapping				Select Lowest Mapping				
SI	XX	XXX	INT	SI	XX	XXX	INT	SI	XX	XXX	INT	
0	0	0 - 0 11	- 1	1	1	0 - 0 11	- 1	0	0	1 - 1 00	- 0	
0	0	0 - 0 11	- 0	1	1	0 - 0 11	- 0	0	0	1 - 1 00	- 1	
		⋮				⋮				⋮		CLASS 1
0	0	0 - 0 00	- 1	1	1	0 - 0 00	- 1	0	0	1 - 1 11	- 0	
0	0	0 - 0 00	- 0	1	1	0 - 0 00	- 0	0	0	1 - 1 11	- 1	(ZERO)
1	0	0 - 0 11	- 1	0	0	1 - 1 11	- 1	1	1	0 - 0 00	- 0	
1	0	0 - 0 11	- 0	0	0	1 - 1 11	- 0	1	1	0 - 0 00	- 1	
		⋮				⋮				⋮		CLASS 2
1	0	0 - 0 00	- 1	0	0	1 - 1 00	- 1	1	1	0 - 0 11	- 0	
1	0	0 - 0 00	- 0	0	0	1 - 1 00	- 0	1	1	0 - 0 11	- 1	

Figure 3.6 Number Classes and Conversion Integer

The conversion is done in the ALU of the element. Following this mapping, each element contains a value in pure 32 bit magnitude representation with the same relative "value" as the original PEPE number. All that remains is to compare this value in all active elements and leave the element(s) with the maximum/minimum value active.

3.6.2.3 Search

Figure 3.7 illustrates the basic principle behind the search function. One bit of the converted value in each active element is gated out of the element and into the Signal Distribution System (SDS). The SDS in PEPE is that portion of logic that enables the control console and the element bays to communicate. The SDS can be considered to encompass the distributed logic. In the SDS, they are "OR"ed together and transmitted back to each element. The line sent back to each element indicates that at least one of the active elements has a "1" in the bit position being examined. A decision is then made in each element to reset the element activity if that element has a "0" in the bit position and at least one element has a "1". It is clear that any element with a "1" in a particular bit position has a number of greater magnitude than an element with a "0" in that position.

Due to PEPE layout constraints, the propagation delay encountered in the "OR" circuit will be on the order of 200 ns. Therefore one bit each 200 ns is the maximum rate in the "OR" circuit and a SH/SL operation over the full 32 bit word would take 6400 ns. In order to reduce this time, the method in Figure 3.8 was used. Two bits each 200 ns are output by the element for a search time of 3200 ns.

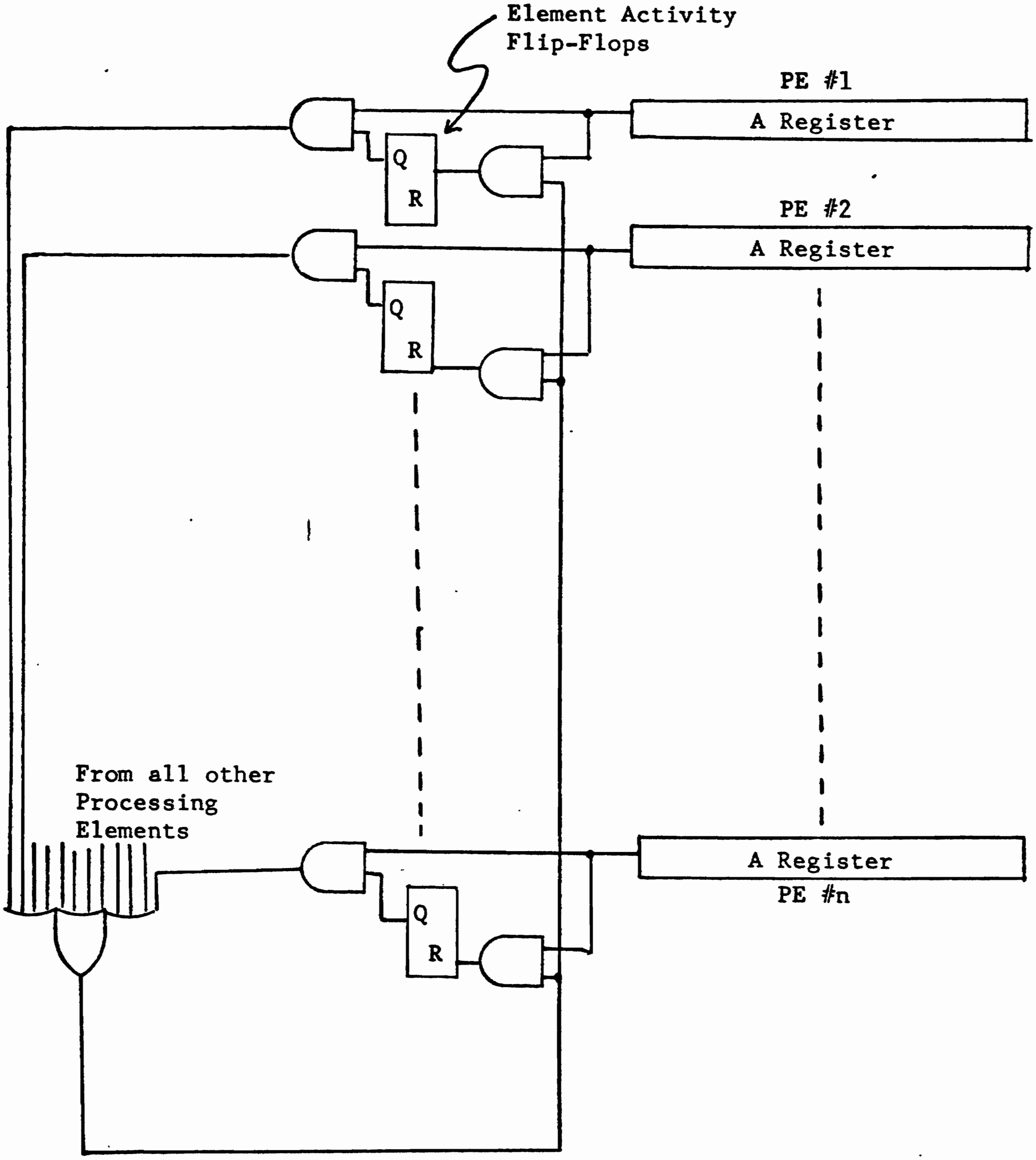


Figure 3.7 SH/SL One Bit Per Cycle

Figure 3.8 shows that 2 bits are decoded and sent out of the element on 3 lines. Each of the 3 lines from all of the elements are "OR"ed and returned to the element. If the element has 2 bits that are less than indicated by the 3 reset lines, the element resets its element activity. Figure 3.9 shows the truth table for this reset function. α and β are the 2 bits being examined, A, B and C are the 3 output lines of the element, X, Y and Z are the 3 "OR"s of all the elements A, B and C lines, and RESET is the signal to reset the element activity flip-flop. Line A equals "1" when both bits being examined are binary "1". Line B equals "1" when the high order bit is a "1" and the low order bit is a "0". Line C equals 1 when the low order bit is a "1" and the high order bit is a "0". All 3 lines are "0" when both bits are zero. These 3 lines are "OR"ed in the CSD/BSD with the corresponding lines from all the other active elements and these 3 signals are returned to every element. The element then compares its local A, B, C lines with the global X, Y, Z lines and any elements whose ABC lines indicate its 2 bits are less than the XYZ lines show for the entire system, will reset its element activity.

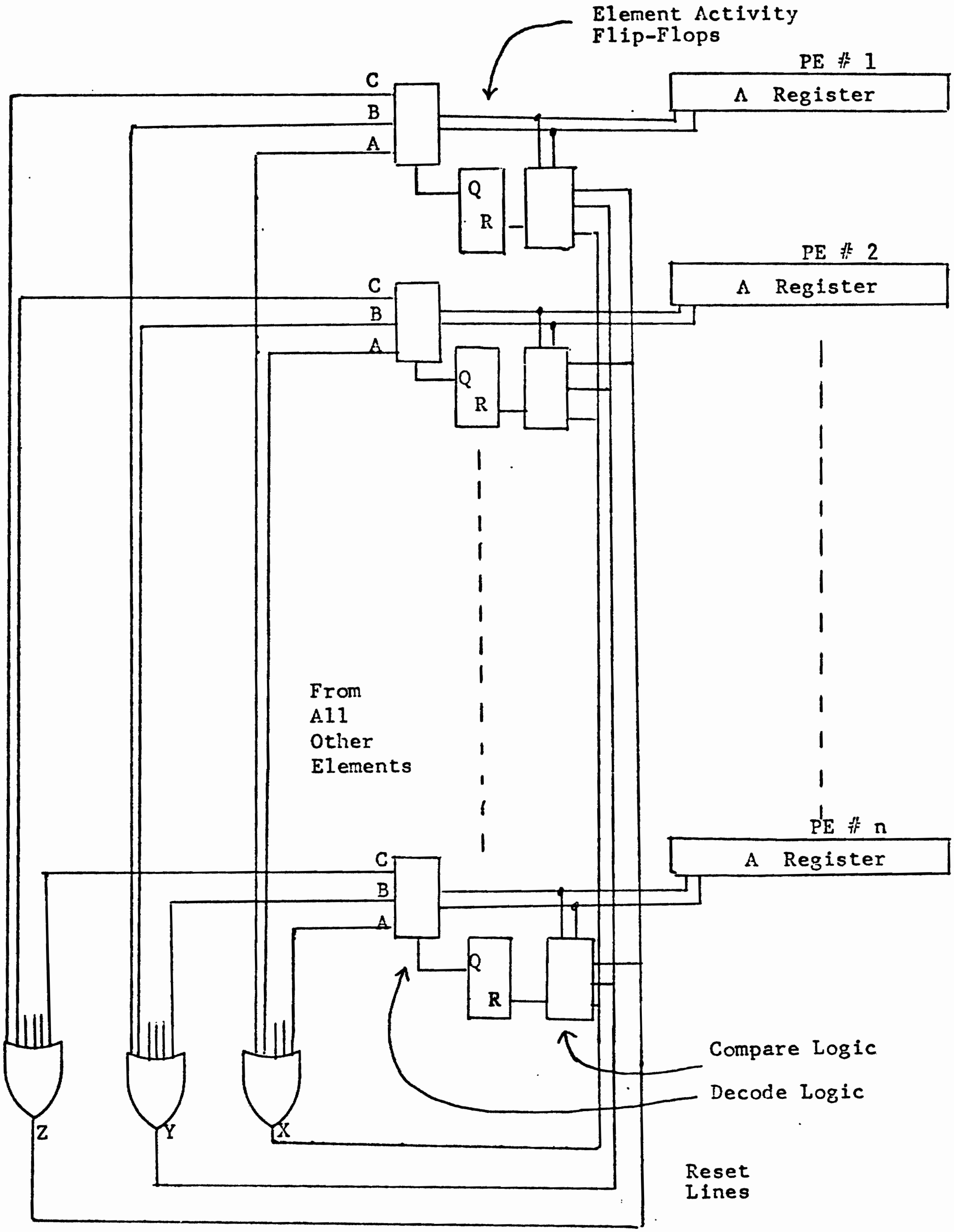


Figure 3.8 SH/SL Two Bits Per Cycle

A,B,C Lines Truth Table

ALPHA	BETA	A	B	C
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

$A = ALPHA \cdot BETA$
$B = ALPHA \cdot \overline{BETA}$
$C = \overline{ALPHA} \cdot BETA$

Reset Element Activity Truth Table

ALPHA	BETA	A	B	C	X	Y	Z	RESET
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	0	0	1
0	0	0	0	0	1	0	1	1
0	0	0	0	0	1	1	0	1
0	0	0	0	0	1	1	1	1
0	1	0	0	1	0	0	1	0
0	1	0	0	1	0	1	1	1
0	1	0	0	1	1	0	1	1
0	1	0	0	1	1	1	1	1
1	0	0	1	0	0	1	0	0
1	0	0	1	0	0	1	1	0
1	0	0	1	0	1	1	0	1
1	0	0	1	0	1	1	1	1
1	1	1	0	0	1	0	0	0
1	1	1	0	0	1	0	1	0
1	1	1	0	0	1	1	0	0
1	1	1	0	0	1	1	1	0

$$RESET = \bar{A}X + \bar{A}\bar{B}Y + \bar{A}\bar{B}\bar{C}Z$$

Figure 3.9 Search:Reset Element Activity Function

The flip-flops α and β contents are as follows during execution of SH/SL:

u-step	α	β	
1	0	0	conversion micro-step
2	A23*	B31	
3			
4	B30	B29	
5			
6	B28	B27	
7			
8	B26	B25	
9			
10	B24	B23*	
11			
12	B22	B21	
13			
14	B20	B19	
15			
16	B18	B17	
17			
18	B16	B15	
19			
20	B14	B13	
21			
22	B12	B11	
23			
24	B10	B09	
25			
26	B08	B07	
27			
28	B06	B05	
29			
30	B04	B03	
31			
32	B02	B01	
33			
34	B00	0	
35			
**			

*Note that the effective bit 23 is examined twice. This was done to simplify the SH/SL logic. The second pass of bit 23 through the compare logic is a NO-OP.

**at the end of micro-step 35, α and β reset to 0.

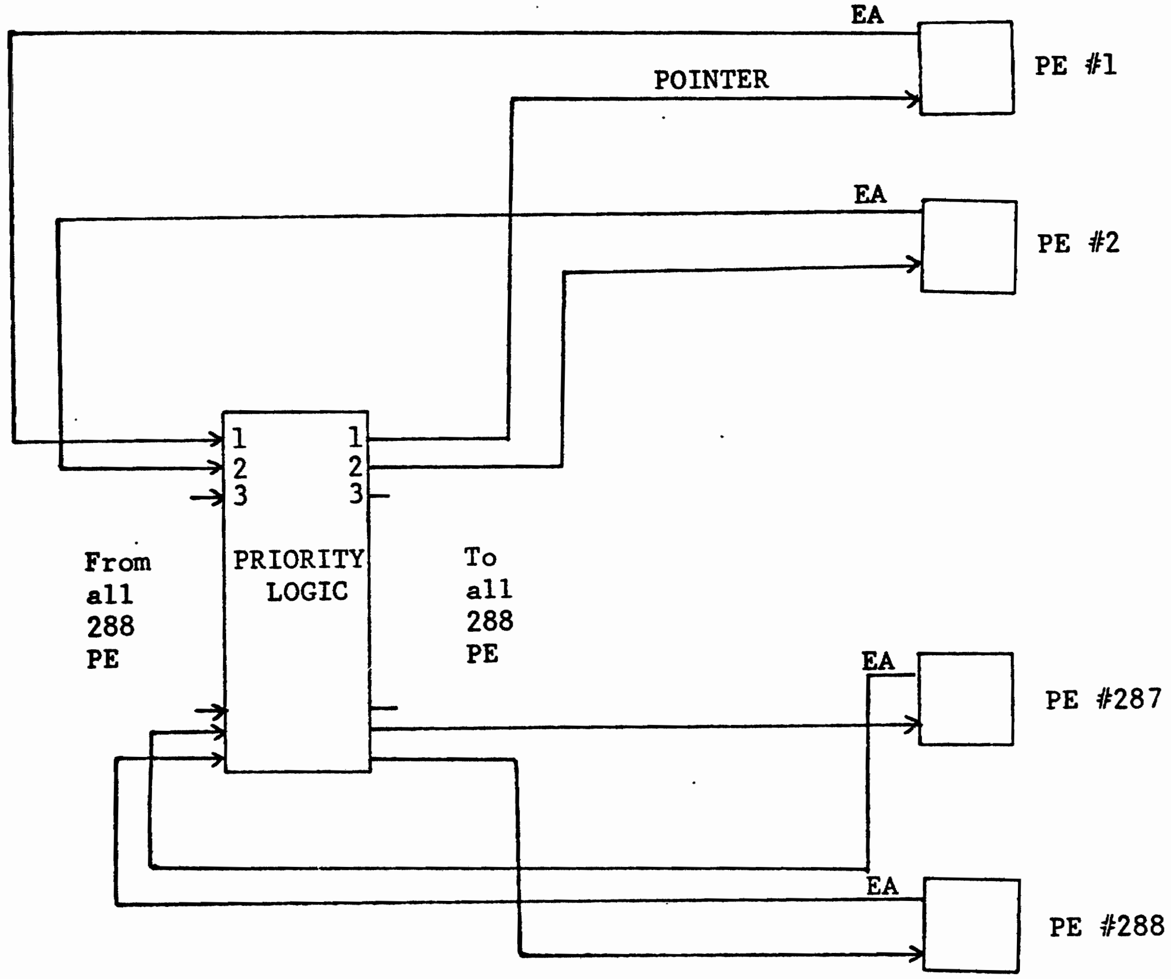
The PICU does not directly participate in the actual search. It sends controls to the elements directing the element to generate the A, B and C lines, to do the mapping, to compare the X, Y and Z lines and to reset the element activity if the compare so indicates. The PICU can end the search early. If during the execution of the search instruction, the element activity count goes to one, the PICU terminates execution of the instruction. This early end is possible because, if there is only one element left, it has the highest or lowest valued A register in the set of active elements. This logic is described in detail in section 4.2.

3.6.3 Select First

Following the execution of a SH, SL or other select instruction, many AU may still remain active. If we wish to extract a data item via the ODB, we require only one active AU. The Select First (SF) instruction, will cause all but one AU to reset its EA.

The one AU selected to remain active will be the "first" one found. "First" is defined as the AU within the lowest numbered PE. As shown in Figure 3.10, each AU supplies its EA to the priority logic. This logic then determines the lowest numbered AU and returns a pointer to this AU. The priority logic is always active. During the execution of a SF instruction, the AU is commanded to load the pointer ("one" or "zero") into its EA flip-flop thus selecting one AU.

The priority logic is contained within each Bay and the CC.



Processing Elements are Numbered on the Basis of Physical Location

Figure 3.10 Select First Block Diagram

4.0 Detail Design of the AU

In this section I will describe in detail the implementation of the algorithms explained in Section 3. After a description of the physical hardware, I will show how the PICU controls the AU. Finally, the special hardware which enables the AU to perform as an efficient parallel associative processor is described.

4.1 Hardware

The selection of the physical and electrical parameters of PEPE played an important role in its success.

4.1.1 Circuits - ECL

The AU is designed using readily available 10K series Emitter Coupled Logic (ECL). Appendix D lists the package types used along with the logic symbols and truth tables for the more complex Medium Scale Integration (MSI) chips.

The use of 10K series logic with its two nanosecond typical delay times permits the AU to execute at a speed of 100 ns per micro-step. With the exception of the 10181 each of these chips is mounted in a 16 pin Dual-In-Line Package (DIP). The 10181 is a 24 pin DIP.

In the remainder of this section, the five digit 10K gate type will generally be abbreviated to the last three digits.

4.1.2 Boards

The AU was fabricated on two large printed circuit boards. The boards, which were specially designed for PEPE, contain eight printed circuit layers. Four of these layers carry ECL signals. Each printed line is controlled during manufacture to maintain a 55 ohm impedence. Two of the remaining layers distribute voltages to the DIPs. Voltages used are a

VCC of +2 and a VEE of -3.2. Finally, the last two layers are ground, used both for DC reference and as a ground plane for the signal lines.

In addition to printed circuit, the board can be wirewrapped with open wire and miniature coaxial cable. All of the signal carrying media are controlled impedance (55 ohm) lines. Since these lines are similar to high frequency transmission cables, they are properly terminated with a 55 ohm resistive load to minimize reflections.

Shown in figure 4.1, the board can hold up to 300 16-pin DIPS. Since the ALU (181) is a 24-pin DIP , special provision was made to permit up to 10 24-pin sockets in the center of the board. Each 24 pin socket replaces two 16 pin sockets. In the final AU design, board AU1 contains 201 16-pin ECL DIPS, 3 24-pin 181 DIPS and 73 16 pin 55Ω terminator modules. Board AU2 contains 219 16-pin ECL DIPS, 6 24-pin 181 DIPS and 67 16 pin terminator modules.

Interconnection into the backplane for signals, power and ground is through four 100 pin connectors. 256 of the 400 pins are available for signals.

4.1.3 Backplane

Each AU board plugs into a 55 slot backplane/card rack assembly. The backplane can accommodate nine processing elements. It is a three layer laminated assembly used not only for physical support but also for power distribution. Power enters the board from the backplane via the two center 100 pin connectors which are mounted on the backplane.

Figure 4.2 shows a rear view of the Element Bay. The Bay houses four backplane/card rack assemblies and power supply for each row. The power bussing scheme is designed as a low inductance distribution network to provide power to the row with less than a 50 milli-volt drop along the five foot backplane.

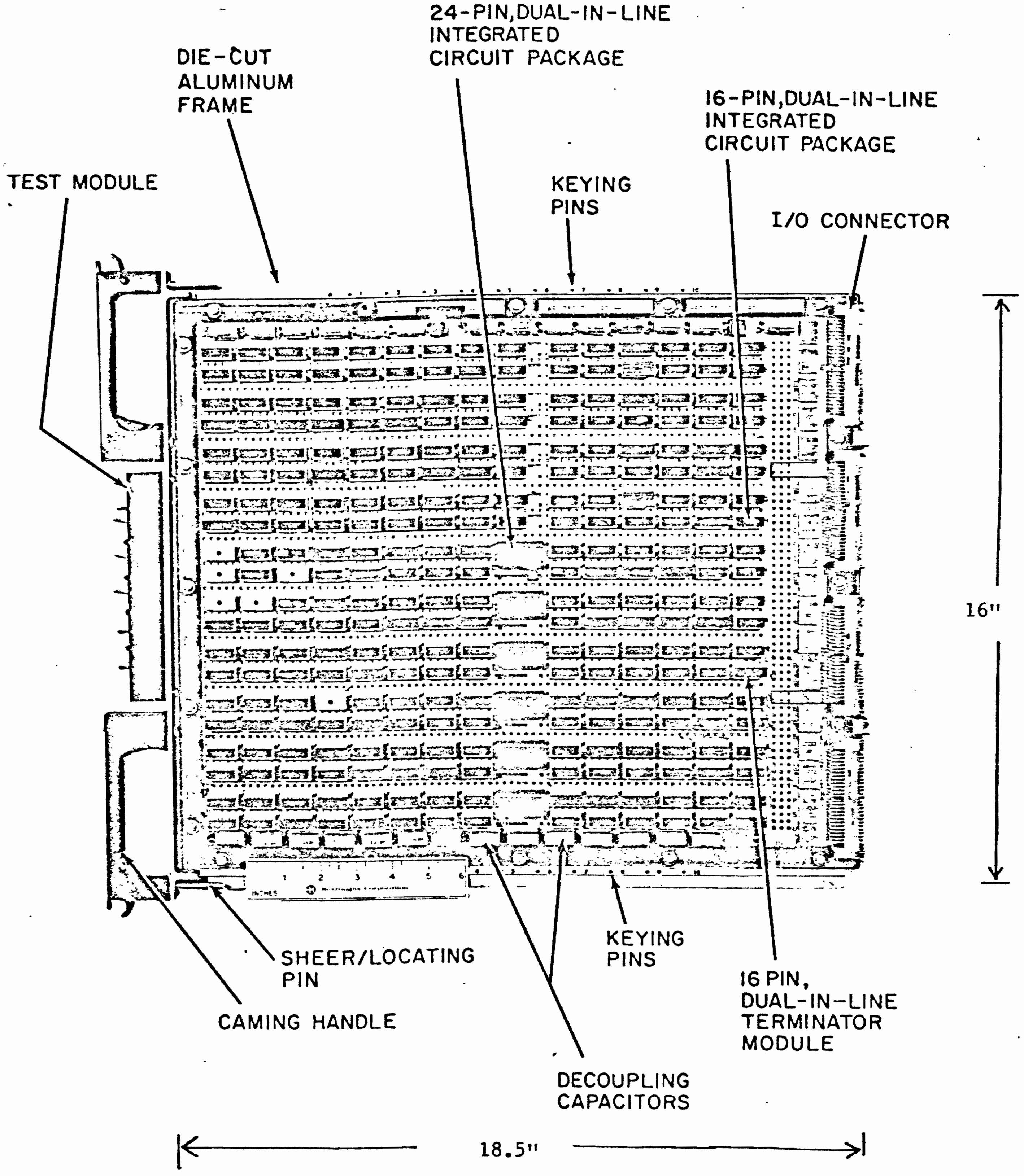

24-PIN,DUAL-IN-LINE INTEGRATED CIRCUIT PACKAGE
DIE-ĈUT ALUMINUM FRAME
16-PIN,DUAL-IN-LINE INTEGRATED CIRCUIT PACKAGE
TEST MODULE
KEYING PINS
I/O CONNECTOR
16"
SHEER/LOCATING PIN
CAMING HANDLE
KEYING PINS
16 PIN, DUAL-IN-LINE TERMINATOR MODULE
DECOUPLING CAPACITORS
18.5"

Figure 4.1 PEPE Printed Circuit Board

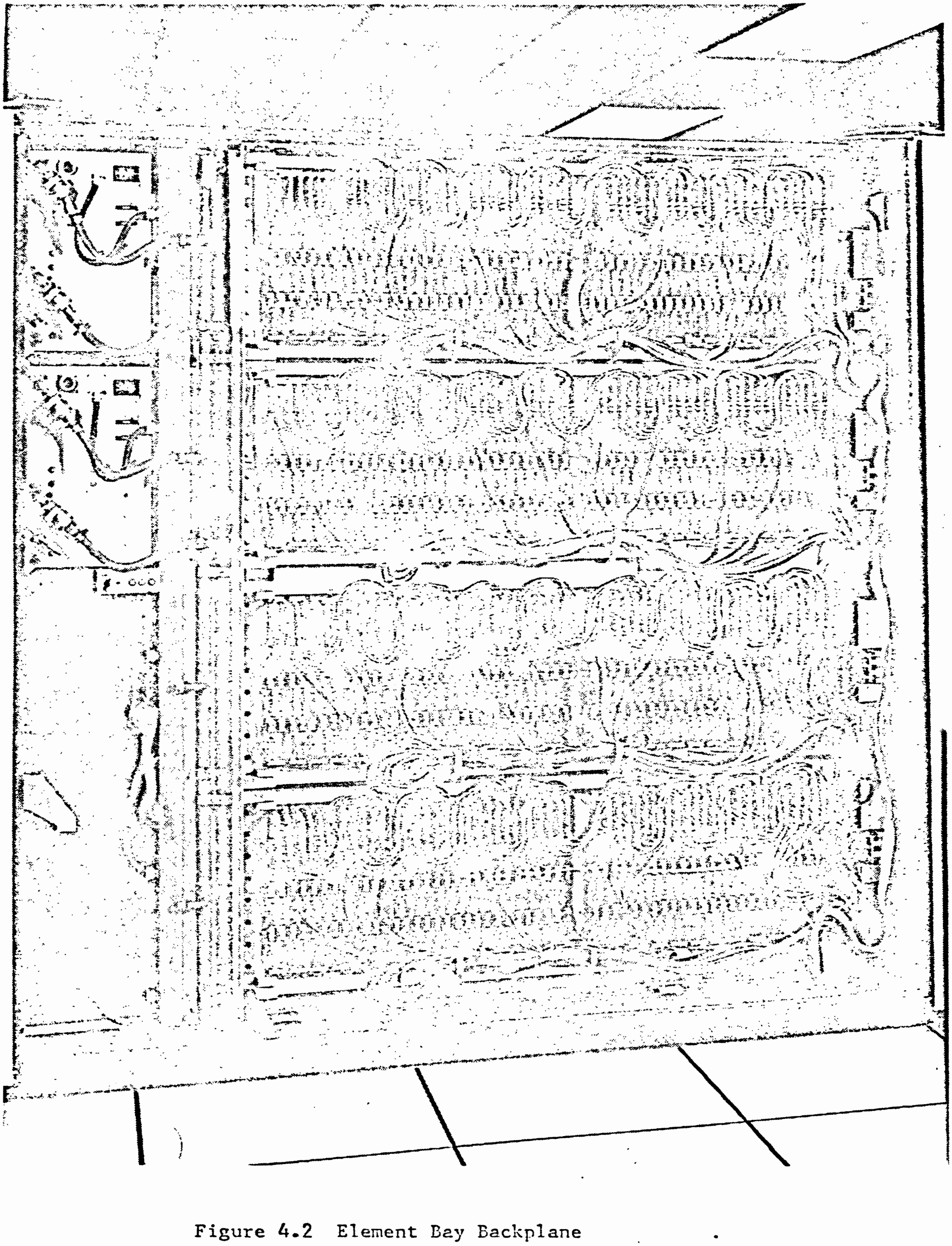


Figure 4.2 Element Bay Backplane

All signal interconnections between boards of the PE are via the three bottom connectors of the boards. Miniature coaxial cable is used as the media. The top connector (P1) uses a specially designed signal distribution scheme using belted cable. The belted cable carries control signals from the BSD to the PE in the row. The regular pattern of cables seen in Figure 4.2 serve to outline each PE. The signalling method used on these cables is differential balanced pair. This provides immunity to both resistive voltage losses and common mode injected noise. The Output Data Bus is also carried on these belted cables.

The control signals reach these belts after originating in the PICU and travelling through the Signal Distribution System. Upon reaching the AU, they cause registers, selectors and functional units to execute the instruction decoded in the PICU. These control lines will be described in the remainder of this section. Appendix E will serve as a quick reference for signal names used in this section.

4.2 Parallel Instruction Control Unit

The control lines for the AU are generated by the Parallel Instruction Control Unit (PICU) in the Control Console. The PICU contains a Micro-Program Memory (MPM) that stores the control bits for each micro-step. The PICU also contains control logic for correct sequencing of each instruction.

4.2.1 Purpose/MPM

The ACU-PICU is part of the Arithmetic Control Unit and is supplied parallel instructions and data from the ACU-SCL. The purpose of the PICU is to cause parallel element AU's to properly perform parallel instructions by transmitting, through the Signal Distribution System, global control and data. The PICU must also interface with the Element Memory Control (EMC) to assure proper operation in the event of Element Memory access conflicts with the two other processors in each PE.

4.2.2 Operation

Figure 4.3 shows a block diagram of the ACU-PICU. The following paragraphs describe the operation of each block.

4.2.2.1 PICU Interfaces

4.2.2.1.1 ACU-SCL/PIQ

The PICU receives instructions and data from the SCL/PIQ. These data are presented to the PICU on 47 lines:

SAIINP(31-0)	32 bit operand
SAIINP(39-32)	8 bit opcode
SAIINO(40)	R (EM reference) bit
SAIINP(45-41)	not used in ACU-PICU
SAIINP(46)	EM look-ahead bit

The PICU signals with SAIEMT when it has completed execution of its parallel instruction. The SCL/PIQ signals with SAIREQ to the PICU when a new parallel instruction is available. The PICU acknowledges receipt of the instruction with SAIACK.

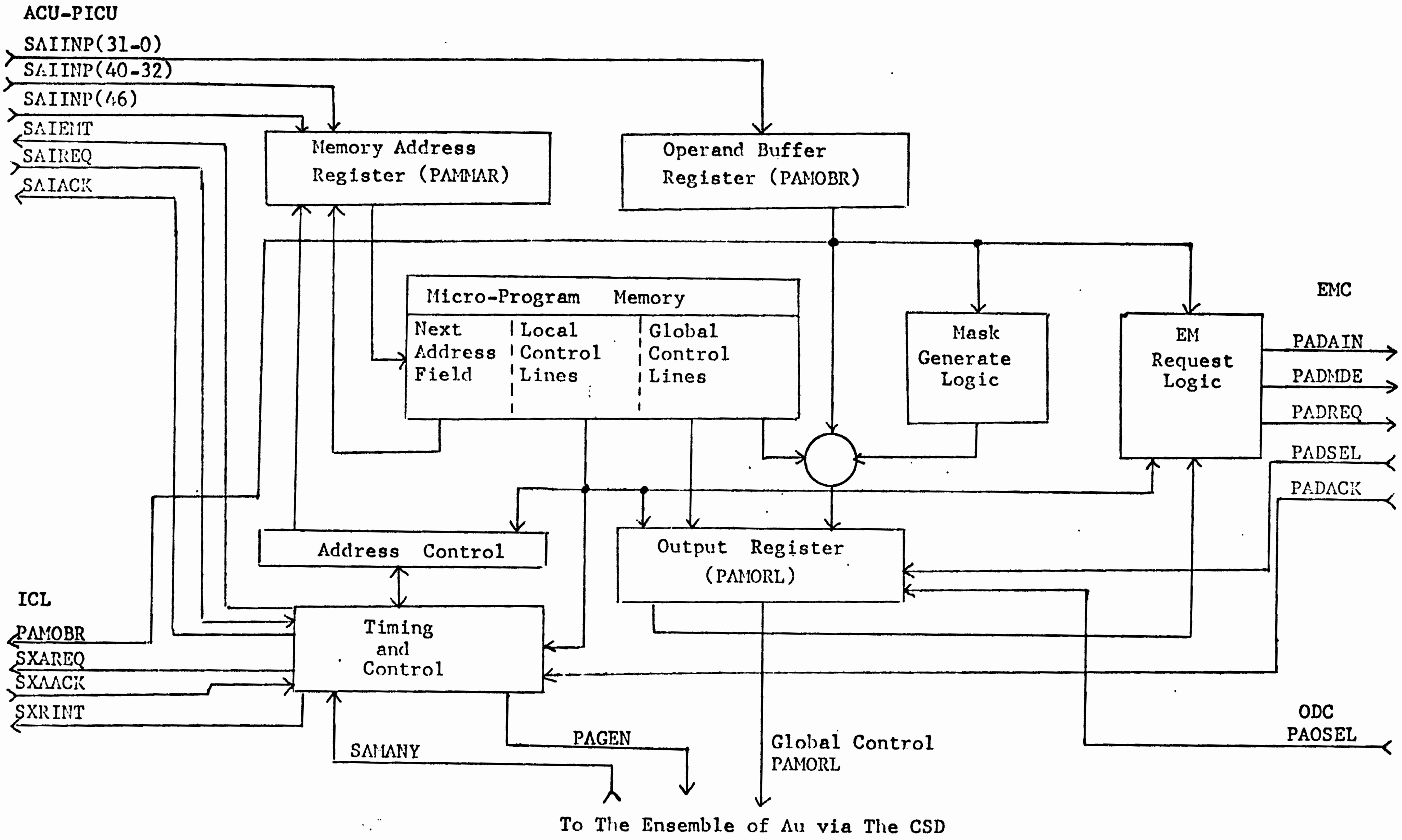


Figure 4.3 ACU-PICU Block Diagram

4.2.2.1.2 Element Memory Control (EMC)

When the PICU requires EM access during the course of a parallel instruction execution, it signals the EMC by:

1. Placing the EM address on address bus PADAIN
2. Signaling read or write with line PADMDE
3. Requesting access by raising line PADREQ

When the EMC has selected the ACU-PICU for service, it responds with PADSEL. The PICU then utilizes this signal to gate the EM data bus to or from the AU.

4.2.2.1.3 Signal Distribution System (SDS)

The PICU receives one signal from the SDS. Named, SAMANY, it is derived from the AU count logic (see section 3.6.1) and indicates that more than one AU in the ensemble are active.

Since the AU contain little execution control logic (they are composed of registers and selectors), all switching control and data strobe signals are generated by the PICU and transmitted to the ensemble of AU through the SDS.

4.2.2.1.4 Output Data Control (ODC)

The ODC transmits PAOSEL to the ACU-PICU during the execution of an OTA instruction. The PICU then causes the active AU to place the contents of its A Register on the Output Data Bus.

4.2.2.1.5 Intercommunication Logic (ICL)

A special purpose interface exists to the ICL. It does not affect operation of the AU and will not be discussed here.

4.2.2.2 Micro-Program Memory

The MPM consists of 1024 words each 80 bits wide. The memory is addressed with 9 bits, the 8 bit opcode and the R bit. The next address field contains the address of the next micro-step. This field allows jumps within the MPM. The global control line field bits are used to define steering gates and register strobes in the ensemble AU which allow data to pass from one register to another. These lines are discussed in detail in Section 4.3.

The local PICU control line field is used to control operations within the PICU. Some of the local control bits are listed:

- PADREQ Request for EM cycle to EMC.
- PADMDE R/W indicator to EMC.
- PAMEOI End of instruction bit. Causes the control to switch the next parallel instruction opcode from the SCL/PIQ into the memory address register. Otherwise the MPM next address field is used.
- PAMGDT A two bit field used to select one of the following as input to bits 31-0 of the PAMORL Output Register:
  1. Operand from the SCL/PIQ (PAMOBR(31-0))
  2. Mask Generation Logic
  3. MPM Global Control Bits (31-0)
- PAMEXT Set to "1" for each micro-step of a Select Highest/Lowest instruction. It is used to affect an early exit from these instruction when only one AU remains active. Section 3.6.2 describes its use in more detail.

The MPM is constructed of 1024 X 1 bit RAM and is loaded from the Test and Maintenance Computer.

4.2.2.3 Memory Address Register (PAMMAR)

PAMMAR supplies address lines to the MPM. It is loaded from the MPM next address field or from SCL/PIQ lines 40-32.

4.2.2.4 Operand Buffer Register (PAMOBR)

Thirty-two bits from the SCL/PIQ are gated into the PAMOBR when the parallel instruction is transferred into the PICU. The data represents the parallel instruction operand. It may be required during any, or none, of the micro-steps of the instruction. The Output Register selector will determine, based on PAMGDT, if and when the PAMOBR will be selected onto the PAMORL lines.

When a request is made to EMC, the lower ten bits are sent as the EM address.

Bits 15-11 are transferred to the Mask Generate Logic during execution of the instructions RBIT, SBIT, PSEL and SELB.

4.2.2.5 Output Register (PAMORL)

The PAMORL holds the current micro-step data and control being transmitted to the ensemble AU. Input to the register is controlled by PAMGDT (paragraph 4.2.2.2).

4.2.2.6 Mask Generate Logic

In order to make efficient use of EM, instructions are provided which operate on a single bit of memory at a time. The instructions are SBIT, RBIT, PSEL and SELB. They are implemented in the AU by performing logical operations over the entire 32 bit data word. To protect the 31 bits that are not involved in the instruction, the PICU generates a mask.

The mask for RBIT will contain "ones" in every bit position except for the bit indicated by PAMOBR(15-11). The mask for SBIT, PSEL and SELB will be all "zeros" with a single "one".

4.2.2.7 Timing and Control

This block performs all of the control functions of the PICU and handles all of the interfaces.

It generates PAGEN, the general clock-strobe, to the AU, causing the AU to execute the current micro-step.

4.2.2.8 The R bit

The R or routing bit is generated by the SCL when decoding the instruction. If the programmer specified a global operand, the R bit is "zero" and the PAMOBR is passed to the AU. If an EM operand was specified, the R bit is a "one". The PICU requests an EM cycle and causes the AU to use its local EM data bus as the source of the operand. The PICU implements this by using the R bit as an address bit in accessing MPM. The AU is then caused to execute a different micro-sequence based on the operand source.

4.3 Detail Block Diagram

Figure 4.4 shows a detail block diagram of the AU. Depicted are all of the registers, flip-flops, selectors, functional units and data paths. Each of these receives a control line originating in the PICU MPM. That control line or lines causes actions which enable the AU to perform instructions. Table 4.1 relates the bit numbers of the PICU Output Register (PAMORL) to the AU control line names.

4.3.1 Registers/Flip-Flops

Table 4.2 lists the control lines for the registers in the AU and Table 4.3 lists the lines for the flip-flops. A "one" on the control line causes the storage element to load the contents of its input lines under the following conditions:

- Since the instruction stream to the AU may not be continuous, signal PAGEN is sent to all AU by the PICU. If PAGEN is "zero" no storage elements are permitted to change state in the AU.
- If the AU is inactive (PAEACT is reset), no user visible registers are permitted to change state with minor exceptions. The instructions which execute in an inactive AU are only those which can activate the AU.
- If the entire PE is faulted (PAFALT is set), the AU will not respond to any instruction except "clear fault" (CF). No user visible storage element is permitted to change state.

4.3.1.1 A Register Mantissa PAAREG(23-0)

STROBE=PAEACT $\cdot$ $\overline{\text{PAFALT}}$ $\cdot$ PAMORL19

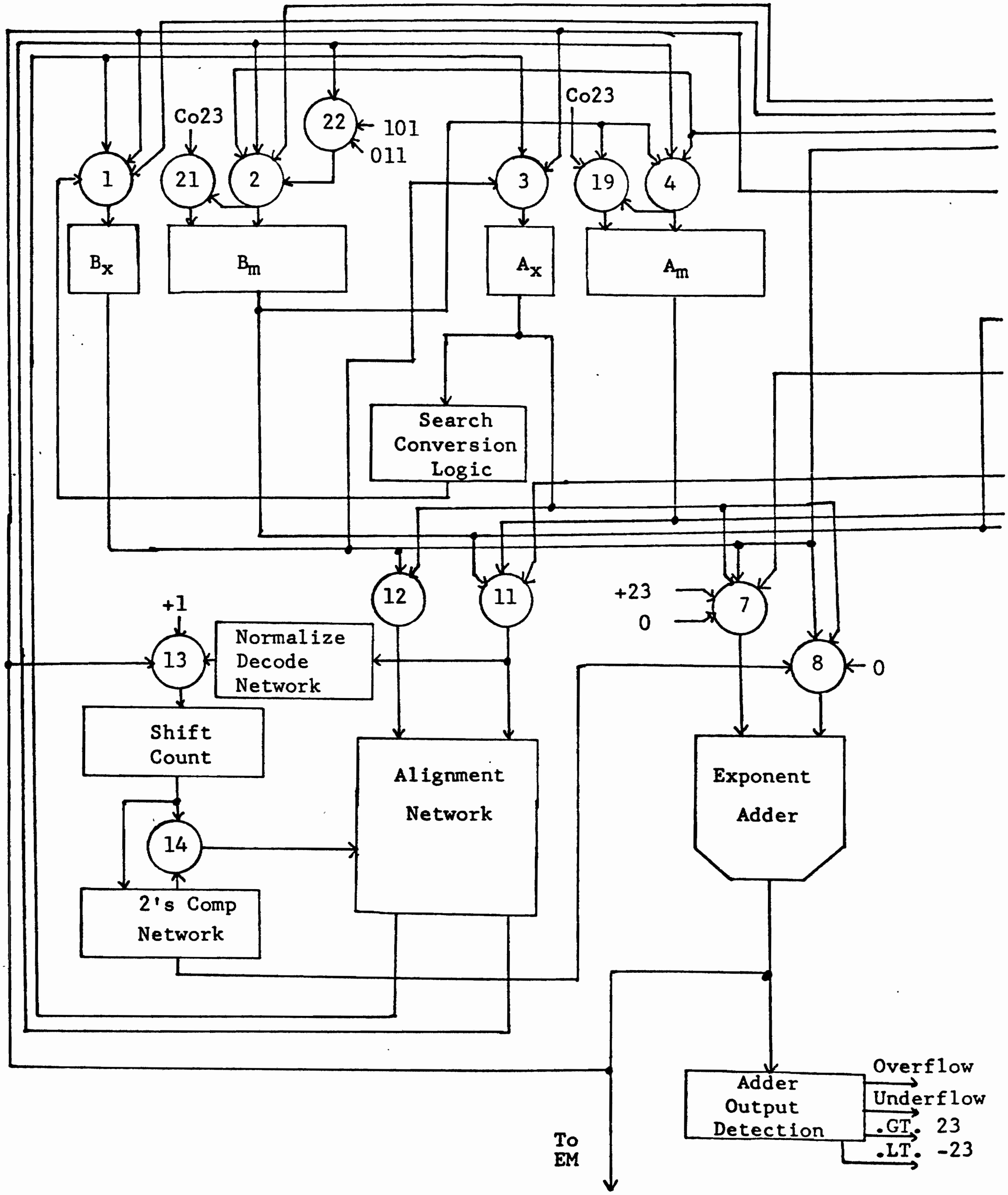


Figure 4.4a PEPE AU Detailed Block Diagram

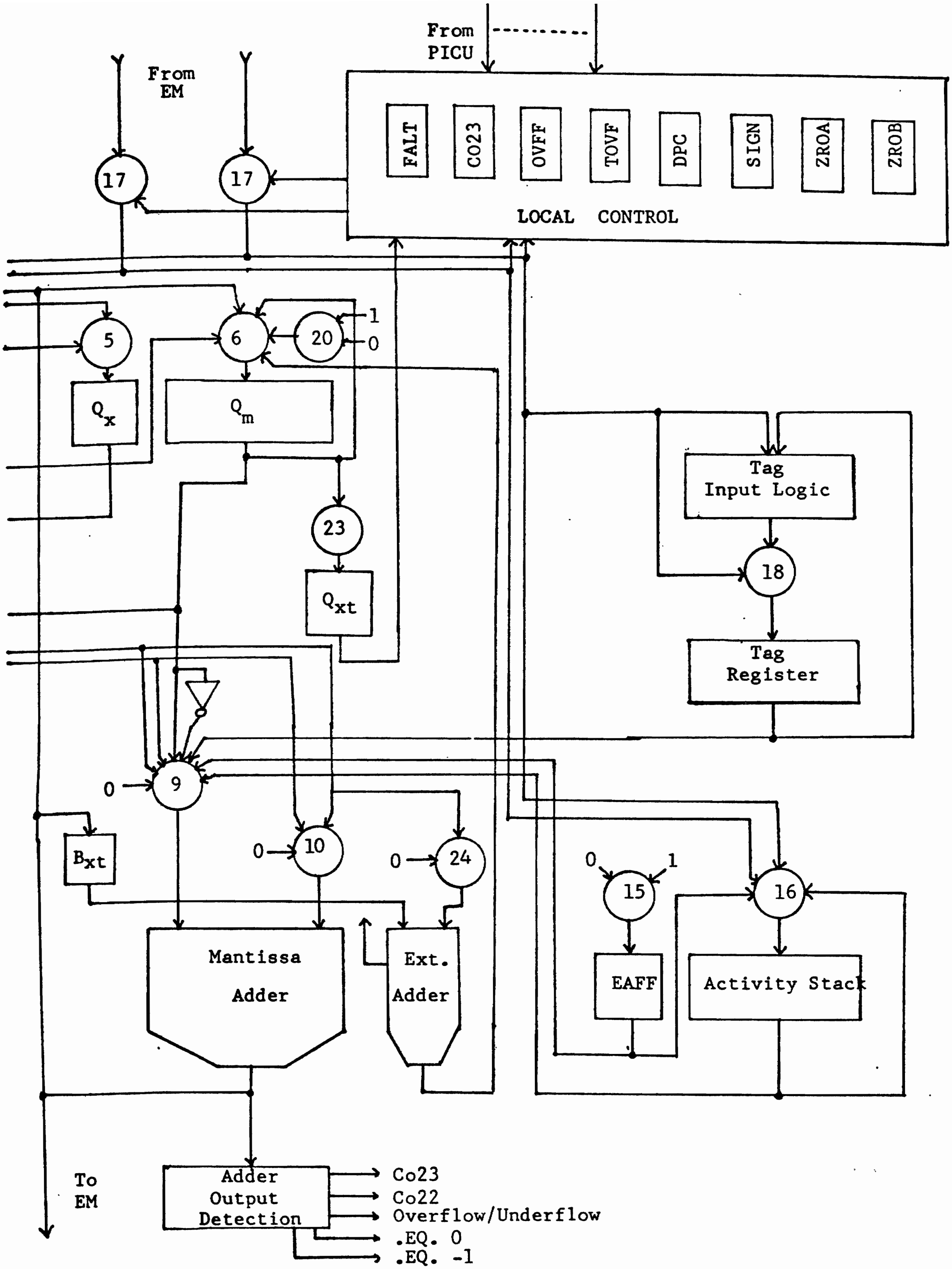


Figure 4.4b PEPE AU Detail Block Diagram

PAMORL BIT NO.	SIGNAL NAME	DEFINITION
	FOR PAMORL 48=0	
00	#PAADMOO	Adder Control (ALU)
01	#PAADMO1	↓
02	#PADDMO2	
03	#PAADMO3	Adder Control
04	#PAAMAOO	Adder mantissa, A input select
05	#PAAMAO1	↓
06	#PAAMAO2	Adder mantissa, A input select
07	#PAAMBOO	Adder mantissa, B input select
08	#PAAMBO1	Adder mantissa, B input select
09	#PAAEBOO	Adder exponent, B input select
10	#PAAEBO1	Adder exponent, B input select
11	#PAAEAOO	Adder exponent, A input select
12	#PAAEAO1	↓
13	#PAAEAO2	Adder exponent, A input select
14	#PAAMSOO	A register, mantissa input select
15	#PAAMSO1	A register, mantissa input select
16	#PAAESOO	A register, exponent input select
17	#PAAESO1	↓
18	#PAAESO2	A register, exponent input select
19	#PAAMM	Clock enable, A mantissa
20	#PAAEM	Clock enable, A exponent
21	#PAQMSOO	Q register, mantissa input select
22	#PAQMSO1	Q register, mantissa input select
23	#PAQESOO	Q register, exponent input select
24	#PAINSOO	Local control decode
25	#PAINSO1	↓
26	#PAINSO2	
27	#PAINSO3	
28	#PAINSO4	Local control decode
29	#PAALO--	Alignment network control
30	#PAALNOO	Alignment network input select
31	#PAALNO1	Alignment network input select
	FOR PAMORL 48=1	
31-00	PAXDIN	Global Data From PICU

TABLE 4.1, CONTROL SIGNALS, AU

PAMORL BIT NO.	SIGNAL NAME FOR 49=1	DEFINITION
32	#PASTK	Clock enable, stack register
33	#PAEAZ00	Element activity control decoder input
34	#PAEAZ01	Element activity control decoder input
35	#PAEAZ02	Element activity control decoder input
36	#PAEAZ03	Element activity control decoder input
37	#PAEAZ04	Element activity control decoder input
38	#PATGS00	Tag register, input select
39	#PATGS01	Tag register, input select
40	#PAEAC	Clock enable, element activity flip-flop
41	#PADPC	Clock enable, double precision carry
42	#PATAG	Clock enable, tag register
43	#PASTK00	Stack register, input select
44	#PASTK01	Stack register, input select
	SIGNAL NAME FOR 49=0	
32	#PANOR	Alignment network control
33	#PAFIX	Special control inhibit A reg mantissa Clock during microstep 5 of FIX instruction
34	#PATOV	Clock enable, temporary overflow hold FF
35	#PACO-23	Clock enable, carry out 23 hold FF
36	#PAQMM	Clock enable, Q mantissa
37	#PAQEM	Clock enable, Q exponent
38	#PABMS00	B register, mantissa input select
39	#PABMS01	B register, mantissa input select
40	#PABES00	B register, exponent input select
41	#PABES01	B register, exponent input select
42	#PABMM01	Clock enable, B mantissa bits 7-0
43	#PABMM02	Clock enable, B mantissa bits 23-8
44	#PABEM	Clock enable, B exponent
45	#PAASCS00	Shift control register, input select
46	#PASCS01	Shift control register, input select
47	#PAOV	Clock enable, PAOVFF flip-flop
48	#PADTC	Control signal, data time shared with control
49	#PACTC	Control signal, select control group
50	#PASCR	Clock enable, shift count register
51	#PARVM	PAXDIN input select

a) The meaning of PAMORL (31-0) lines is dependent on PAMORL 48.
b) The meaning of PAMORL (44-32) lines is dependent on PAMORL 49.

TABLE 4.1, CONTROL SIGNALS, AU

REGISTER (Bit #)	CONTROL LINE
A_m(23-0)	PAMORL19
A_x(31-24)	PAMORL20
B_m(7-0)	PAMORL42
B_m(23-8)	PAMORL43
B_x(31-24)	PAMORL44
B_{xt}(Extension bit)	None
Q_m(23-0)	PAMORL36
Q_x(31-24)	PAMORL37
Q_{xt}(Extension bit)	None
Shift Count Register	PAMORL50
Tag Register	PAMORL42
Activity Stack	PAMORL32

Table 4.2 Register Control

CONTROL FLIP-FLOP	CONTROL LINE
PAEACT	PAMORL40
PAFALT	Local Control
PAZROA	Lbcal Control
PAZROB	Local Control
PASIGN	Local Control
PADPC	PAMORL41
PATOVF	PAMORL34
PAOVFF	PAMORL47
PACO23FF	PAMORL35

Table 4.3 Flip-Flop Control

4.3.1.2 A Register Exponent PAAREG(31-24)

STROBE=PAMORL20 • PAEACT • $\overline{\text{PAFALT}}$

4.3.1.3 B Register Mantissa 1 PABREG(7-0)

STROBE = PAMORL42 • $\overline{\text{PAFALT}}$

4.3.1.4 B Register Mantissa 2 PABREG(23-8)

STROBE = PAMORL43 • $\overline{\text{PAFALT}}$

4.3.1.5 B Register Exponent PABREG(31-24)

STROBE = PAMORL44 • $\overline{\text{PAFALT}}$

4.3.1.6 Q Register Mantissa PAQREG(23-0)

STROBE = PAMORL36 • $\overline{\text{PAFALT}}$ • PAEACT

4.3.1.7 Q Register Exponent PAQREG(31-24)

STROBE = PAMORL37 • $\overline{\text{PAFALT}}$ • PAEACT

4.3.1.8 Shift Count Register PASHCR(5-0)

STROBE = PAMORL50 • PAEACT • $\overline{\text{PAFALT}}$

4.3.1.9 Tag Register PATAGR(7-0)

STROBE = PAMORL42 • $\overline{\text{PAFALT}}$

4.3.1.10 Activity Stack PASTAK(21-0)

STROBE = PAMORL32 • $\overline{\text{PAFALT}}$

4.3.1.11 B Register Extension Bit PABEXT
Q Register Extension Bit PAQEXT

These two single bit registers are used during the multiply instructions. They are continuously clocked. Input to PABEXT is always bit 1 of the adder. Input to PAQEXT is the result of Control Point 24. Used during the ML instruction, see 4.8.4.3.

4.3.1.12 Element Activity Flip-Flop PAEACT

STROBE = (PAMORL40+PAMORL(28-24)=29) • $\overline{\text{PAFALT}}$

Note: This is a special use of PAMORL(28-24). PAMORL40 is the control bit for the flip-flop only when PAMORL49 = 1. When PAMORL40 = 0, PAMORL(28-24) = 29 can be used in place of the control bit.

4.3.1.13 F.P. Add Zero A Input PAZROA
F.P. Add Zero B Input PAZROB

Set and reset by Local Control, see section 4.7.1. Used during the ADD/SB instruction, see 4.8.4.2.

4.3.1.14 DV Sign Flip-Flop PASIGN

Flip-Flop used during the DV instruction to remember to complement the result following the divide. Set and reset by Local Control, see Section 4.5.

4.3.1.15 Double Precision Carry FF PADPC

STROBE = PAMORL41 • PAEACT • $\overline{\text{PAFALT}}$

4.3.1.16 Temporary Overflow FF PATOVF

STROBE = PAMORL34

4.3.1.17 Overflow FF PAOVFF

STROBE = PAMORL47 • PAEACT • $\overline{\text{PAFALT}}$

4.3.1.18 Carry Out of Bit 23 FF PACOFF23

STROBE = PAMORL35

4.3.1.19 Fault Flip-Flop PAFALT

STROBE = (PAMORL(28-24)=30)+
(PAMORL(28-24)=31)

4.3.2 Selectors

Selectors or data multiplexers are numbered 1 through 24. Each selector, which may have several control lines, picks one of many possible inputs and presents that input to the register, flip-flop or functional unit attached.

4.3.2.1 B Register Exponent Input CP1

PAMORL		Function
41	40	
0	0	31-24 = PAALNO(31-24)
0	1	31-24 = PAXDIN(31-24)
1	0	31-24 = PAAOUT(31-24)
1	1	31-24 = PASRCH(31-24)

4.3.2.2 B Register Mantissa Input CP2

PAMORL		Function
39	38	
0	0	23-0 = PAAOUT(23-0)
0	1	23-0 = PAALNO(23-0)
1	0	23-0 = PAAOUT(23, 23, 23-2)
1	1	23-0 = PAXDIN(23-0)

4.3.2.3 A Register Exponent Input CP3

PAMORL			Function
18	17	16	
0	0	0	31-24=exponent of -128
0	0	1	31-24=exponent of -128
0	1	0	31-24=ZERO
0	1	1	31-24=ZERO
1	0	0	31-24=PAAOUT(31-24)
1	0	1	31-24=PAALNO(31-24)
1	1	0	31-24=PAAREG(31, 31-25)
1	1	1	31-24=PABREG(31-24)

4.3.2.4 A Register Mantissa Input CP4

BIT 22-0 truth table

15	14	
0	0	22-0 = PAAOUT(21-0), ZERO
0	1	22-0 = PAALNO(22-0)
1	0	22-0 = PAAOUT(22-0)
1	1	22-0 = PABREG(22-0)

4.3.2.5 Q Register Exponent Input CP5

PAMORL23	Function
0	31-24=PAAOUT(31-24)
1	31-24=PABREG(31-24)

4.3.2.6 Q Register Mantissa Input CP6

PAMORL		Function
22	21	
0	0	23-0=PAQREG(22-0), "bit 0 input"
0	1	23-0=PAAOUT(0), PASEXT, PAQREG(23-2)
1	0	23-0=PAAOUT(23-0)
1	1	23-0=PABREG(23-0)

Q register, mantissa bit 0 input

bit 0 = PAMORL(28-24)≠18 • $\overline{\text{PAAOUT23}}$

4.3.2.7 Input A Exponent Adder CP7

PAMORL			Function
13	12	11	
0	0	0	31-24=PABREG(312-4)
0	0	1	31-24=ZERO
0	1	0	31-24=+23=00010111
0	1	1	31-24=PASTAK(20-13)
1	0	0	31-24=PAQREG(31-24)
1	0	1	31-24=PAAREG(31-24)
1	1	0	not defined
1	1	1	not defined

4.3.2.8 Input B Exponent Adder CP8

PAMORL		Function
10	09	
0	0	31-24 = PA2SCO(5,5,5-0)
0	1	31-24 = PAAREG(31-24)
1	0	31-24 = PABREG(31-24)
1	1	31-24 = ZERO

4.3.2.9 Input A Mantissa Adder CP9

PAMORL			Function
06	05	04	
0	0	0	23-0 = PASTAK(12-0), PAEACT, PAOVFF, PAXDPC, PATAGR(7-0)
0	0	1	23-0 = PAQREG(23-0)
0	1	0	23-0 = PABREG(23-0)
0	1	1	23-0 = ZERO(23-16), PABREG(7-0), ZERO(7-0)
1	0	0	23-0 = PAQREG(23-0)
1	0	1	23-0 = PAAREG(23-0)
1	1	0	23-0 = ZERO
1	1	1	23-0 = PAAREG(21-0), PAQREG(22-21)

4.3.2.10 Input B Mantissa Adder CP10

PAMORL		Function
08	07	
0	0	23-0 = PAAREG(23-0)
0	1	23-0 = PAAREG23, PAAREG23, PAAREG(22-1)
1	0	23-0 = PABREG(23-0)
1	1	23-0 = ZERO

4.3.2.11 Alignment Network/Shifter Mantissa CP11

PAMORL		Function
31	30	
0	0	23-0 = PABREG(23-0)
0	1	23-0 = PAAREG(23-0)
1	0	23-0 = PAQREG(23-0)
1	1	23-0 = Undefined

Inputs (23-0) to alignment network also are gated to a leading zeros/ones counter. Output is a binary count PANRMD(5-0). See Section 4.4.4.1.

4.3.2.12 Alignment Network/Shifter Exponent CP12

Alignment network inputs (31-24) input select truth table

PAMORL 30	Function
0	31-24 = PABREG(31-24)
1	31-24 = PAAREG(31-24)

4.3.2.13 Shift Count Register Input CP13

PAMORL		Function
46	45	
0	0	5-0 = POSITIVE ONE
0	1	5-0 = PAAOUT(29-24)
1	0	5-0 = PANRMD(5-0) See Section 4.4.4.1
1	1	5-0 = PAXDIN(5-0)

4.3.2.14 Alignment Network/Shifter, Shift Distance Selector CP14

PAMORL 32	Function
0	Shift amount = PASHCR(5-0)
1	Shift amount = PA2SCO(5-0) (Two's complement of PASCHR)

4.3.2.15 Element Activity Input CP15

PAMORL(37-33) are used to select the correct input to the EA FF. Table 4.4 lists the 32 possible input selections and the instruction during which each is used. The function is the equation which is evaluated and used to set or reset the flip-flop. "EA STATE" is used to indicate that the EA FF has to be on in order for the instruction to execute.

DECODE	PAMORL 37	 36	 35	 34	 33	USED ON INST	 STEP	FUNCTION	EA STATE
0	0	0	0	0	0	-		NONE	D.C.
1	0	0	0	0	1	SNOV	1	IF PAOVFF, EA← 0	1
2	0	0	0	1	0	SEG SZL	2 1	IF PAAOUT(31-0) ≠ EA← 0	1
3	0	0	0	1	1	SNG SNZL PSEL SELB	2 1 5 4	IF PAAOUT(31-0) = 0, EA← 0	1
4	0	0	1	0	0	SGZ	1	IF PAAOUT23 OR IF PAAOUT(23-0) = zero, EA← 0	1
5	0	0	1	0	1	SGE	1	IF PAAOUT23, EA← 0	1
6	0	0	1	1	0	SNZ	1	IF PAAOUT(23-0) = zero, EA← 0	1
7	0	0	1	1	1	SZR	1	IF PAAOUT(23-0) ≠ zero, EA← 0	1
8	0	1	0	0	0	SLE	1	IF PAAOUT23 AND IF PAAOUT(23-0) ≠ zero, EA← 0	1
9	0	1	0	0	1	SLZ	1	IF PAAOUT23, EA← 0	1
10-11		---				---		NONE	
12	0	1	1	0	0	SF	1	IF $\overline{\text{PASFPT}}$, EA← 0	1
13	0	1	1	0	1	SOV	1	IF $\overline{\text{PAOVFF}}$, EA← 0	1
14-15		---				---		NONE	
16	1	0	0	0	0	LTAG	1	EA← PAXDIN10	D.C.
17	1	0	0	0	1	ACT	3	IF PAAOUT(23-0) = zero, EA← 1	D.C.
18	1	0	0	1	0	CACT	3	EA← (PAAOUT(23-0) = zero)	D.C.
19		---				---		NONE	
20	1	0	1	0	0	COPY POP	1 1	EA← PASTAKOO	D.C.
21	1	0	1	0	1	ORS	1	IF PASTAKOO, EA← 1	D.C.
22	1	0	1	1	0	ANS	1	IF $\overline{\text{PASTAKOO}}$, EA← 0	D.C.
23	1	0	1	1	1	CA	1	IF PASTAKOO AND IF $\overline{\text{PAEACT}}$, EA← 1. Also IF PAEACT, EA← 0	D.C.
24-31		---				---		NONE	

D.C. = Don't Care

TABLE 4.4, ELEMENT ACTIVITY CONTROL

4.3.2.16 Activity Stack Input CP16

PAMORL		Function
44	43	
0	0	20-0 = PAXDIN(31-11)
0	1	20-0 = PASTAK(20-1), 0
1	0	20-0 = PASTAK(19-0), PAEACT
1	1	PASTAK(20-1) : NO CHANGE PASTAK(0) ←0 IF PAEACT ELSE NO CHANGE

Bit 0 is the top of stack.

4.3.2.17 Data Input Selector CP17

PAMORL 51	Function
0	31-0 = PAMORL(31-0) i.e. global
1	31-0 = PXEMOT(31-0) i.e. element memory

4.3.2.18 Tag Register Input CP18

PAMORL		Function
39	38	
0	0	7-0 = PATGLG(7-0) See Note
0	1	7-0 = PATGLG(7-0)
1	0	7-0 = PATGLG(7-0)
1	1	7-0 = PAXDIN(7-0)

Note:

$$\left.\begin{aligned} \text{PATGLGn} &= \text{PAXDIN (n plus 8)} \cdot \text{PAXDINn} + \\ &\overline{\text{PAXDIN (n plus 8)}} \cdot \text{PATAGRn} + \\ &\text{PAXDINn} \cdot \text{PATAGRn} \end{aligned}\right\} \text{for n = 0, 7}$$

PATGLG is the Tag Register input logic shown in Figure 4.4.

4.3.2.19 A Register Bit 23 Input CP19

PAMORL		Function
15	14	
0	0	IF PAMORL(28-24)≠5, 23=PAAOUT(22)
0	1	IF PAMORL(28-24)≠5, 23=PAALNO(23)
1	0	IF PAMORL(28-24)≠5, 23=PAAOUT(23)
1	1	IF PAMORL(28-24)≠5, 23=PABREG(23)

Note: IF PAMORL(28-24)=5, 23=PACOFF23

4.3.2.20 DV/SQ Result Selector CP20

See equation for bit 0 in Section 4.3.2.6

4.3.2.21 B Register Bits 23-22 Input CP21

Control is generated locally, see Section 4.5.13

PAMOVB		Function
01	00	
0	0	Bits (23-22) of CP2
0	1	Adder Carry out signal for both bits
1	0	PACOFF23, bit 22 of CP2
1	1	don't care

4.3.2.22 SQ Result Selector CP22

Control is generated locally, see Section 4.5.14

PASQRT		Function
01	00	
0	0	PAALNO(2-0)
0	1	don't care
1	0	"101"
1	1	"011"

4.3.2.23 Q Extension Bit Input CP23

Control is generated locally, see Section 4.5.15.

PAQSEL	Function
0	"0"
1	PAQREG(1)

4.3.2.24 Extension Adder B Input CP24

Control is generated locally, see Section 4.5.16.

PAAXB	Function
0	PAAREG(0)
1	"0"

4.3.2.25 Double Precision Carry Input CP25

Control is generated locally, see Section 4.5.17.

PADPIN	Function
0	PAXDIN(8)
1	Carry out of Adder bit 22

4.3.3 Functional Units

4.3.3.1 Arithmetic Logic Unit

The ALU performs 32 bit logical operations and 24 bit and 8 bit arithmetic operations. The lower 24 bits, the mantissa adder PAADMN, performs all eight functions shown in the following table. The upper eight bits, or exponent adder PAADEX, perform all functions except decrement.

PAMORL				Function
03	02	01	00	
0	0	0	0	exclusive OR
0	0	0	1	logical AND-NOT
0	0	1	0	logical OR
0	0	1	1	logical AND
0	1	0	0	decrement
0	1	0	1	*
0	1	1	0	*
0	1	1	1	add
1	0	0	0	*
1	0	0	1	*
1	0	1	0	*
1	0	1	1	*
1	1	0	0	subtract
1	1	0	1	*
1	1	1	0	*
1	1	1	1	increment

Notes: * = undefined

4.3.3.2 Alignment Network/Shifter PAALNO

The alignment network can perform four different shifts of up to 32 places. The shift type is selected by:

PAMORL	Function
29	
0	Arithmetic Shift
1	Logical Shift

Shift amount is controlled by the output of control point 14. The most significant bit (the sign) of CP14 output also acts as a control line.

PAMORL	PASCR(5)	Function
	(Output of CP14)	
29		
0	0	24 bit arithmetic right shift sign extended
0	1	24 bit arithmetic left shift zero fill to bit 0
1	0	32 bit logical right shift end around
1	1	32 bit logical left shift zero fill to bit 0

4.4 Micro-Level Implementation

4.4.1 Selectors/Registers

The AU data selectors or multiplexers are constructed out of standard ECL 10K MSI packages. They are:

10164	8 Input Multiplexer
10173	Quad 2 Input Multiplexer
10174	Dual 4 Input Multiplexer

The Registers are also standard MSI packages. They are:

10135	Dual JK Flip-Flop
10141	4 bit Shift Register
10176	Hex D Flip-Flop

4.4.2 Element Activity Flip-Flop

The EA FF is JK with separate selectors for the J and K inputs. Section 4.3.2.15 shows the functions performed by the selectors.

4.4.3 Arithmetic Logic Unit

4.4.3.1 Logical Unit

The ALU directly performs AND, OR, XOR, and ANDNOT operations. The equations are:

AND: $\mathrm{PAAOUT} \leftarrow \mathrm{PAADA} \cdot \mathrm{PAADB}$

OR: $\mathrm{PAAOUT} \leftarrow \mathrm{PAADA} + \mathrm{PAADB}$

XOR: $\mathrm{PAAOUT} \leftarrow \mathrm{PAADA} \cdot \overline{\mathrm{PAADB}} + \overline{\mathrm{PAADA}} \cdot \mathrm{PAADB}$

ANDNOT: $\mathrm{PAAOUT} \leftarrow \mathrm{PAADA} \cdot \overline{\mathrm{PAADB}}$

where: PAADA - A input, both exponent and mantissa
PAADB - B input, both exponent and mantissa

4.4.3.2 Arithmetic Unit

The ALU performs two basic arithmetic functions: (using the 10181 chip)

1 - PAAOUT← PAADMA plus PAADMB plus PACIN

2 - PAAOUT← PAADMA plus $\overline{\text{PAADMB}}$ plus PACIN

where PACIN - Carry in to bit 0

The four required functions are generated as follows:

ADD - Implicit operand to PAADMA
Explicit operand to PAADMB
"0" to PACIN
perform function 1

SUBTRACT - Implicit operand to PAADMA
Explicit operand to PAADMB
"1" to PACIN
perform function 2
This takes the subtrahend and converts it to its two's complement form and performs an addition.

INCREMENT - Implicit operand to PAADMA
Explicit operand to PAADMB
"1" to PACIN
perform function 1

DECREMENT - Implicit operand to PAADMA
Explicit operand to PAADMB
"0" to PACIN
perform function 2

4.4.3.3 Carry Look Ahead

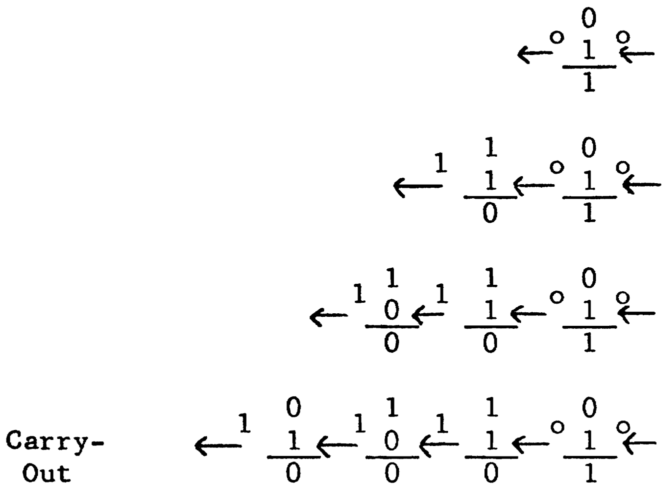
The basic ALU circuit in the AU performs a 24 bit add. The input to each bit position are two 1 bit operands and 1 bit carry-in. The result at each bit position is a 1 bit sum and a 1 bit carry out. It follows the truth table.

A	B	Carry-In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Take, as an example two 4 bit numbers:

$$\begin{array}{rl} & 0110 \\ + & \underline{1011} \leftarrow C_{IN}=0 \end{array}$$

The ALU takes them one bit position at a time with the C_{IN} to that position and generates a sum and C_{IN} for the next bit position:



This effectively takes four operations which must be done sequentially before the final sum and carry-out are known. This is the common ripple carry adder. When implemented in logic, the delay time required is that to let each bit add and to let the carry move down each bit position until it is available at the end of the carry chain.

As an improvement to this adder, consider that much can be inferred about the carry out of a particular stage if the bit patterns of the input to that stage are known. Consider:

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline \end{array}$$ Carry-out must be "0"
no matter what carry-in is.

$$\begin{array}{r} 1 \\ +\ 1 \\ \hline \end{array}$$ Carry-out must be "1"
no matter what carry-in is.

$$\begin{array}{r} 0 \\ +\ 1 \\ \hline \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline \end{array}$$ Carry-out is the same
as carry-in.

So, adding two bits together requires a carry-in. Look at the previous bit position inputs and from above,the carry-in can or cannot be determined. If it is not determined, look at the next previous bit position and continue until carry-in is known. This could be carry-in to the very first bit position - C_{IN}. Since all of the required inputs are available at the start of the add, the logic can proceed immediately not having to wait for any ripple.

Define:

Generate: $Gi = Ai \cdot Bi$

Transmit: $Ti = Ai \oplus Bi$ (Exclusive OR)

where Ai and Bi are the ith bits of the two operands.

Generate indicates that carry-out of this pair is a 1 no matter what carry-in is (Carry is "generated").

Transmit indicates that carry-out of this pair is the same as the carry-out of the last pair (Carry is "transmitted").

So we have:

$$\text{CARRY-IN}i = Gi_{-1} + Ti_{-1} (Gi_{-2} + Ti_{-2} (Gi_{-3} + Ti_{-3} \cdots$$
$$\cdots\cdot (G_o + T_o (C_{IN})) \cdots)$$

From this equation we can get any Carry-in by only looking at the 2 operands and the carry-in bit 0. This is called a carry look ahead adder.

The truth table for carry into stage i-from stage i-1 in a four bit ALU is:

T_{i-1}	G_{i-1}	T_{i-2}	G_{i-2}	T_{i-3}	G_{i-3}	C_{INo}	C_{INi}
Ø	1	Ø	Ø	Ø	Ø	Ø	1
1	0	0	1	Ø	Ø	Ø	1
1	0	1	0	0	1	Ø	1
1	0	1	0	1	0	1	1
1	0	1	0	1	0	0	0
0	0	Ø	Ø	Ø	Ø	Ø	0

Ø = don't care

Actual implementation of this equation can vary drastically. In the AU a 24-bit look ahead adder takes less than 30 ns and uses only 8 ECL chips (6 10181s and 2 10179s).

For simplicity, transmit is often reduced to a simple OR circuit instead of an XOR. It is then referred to as:

$$\text{Propagate: } P_i = A_i + B_i$$

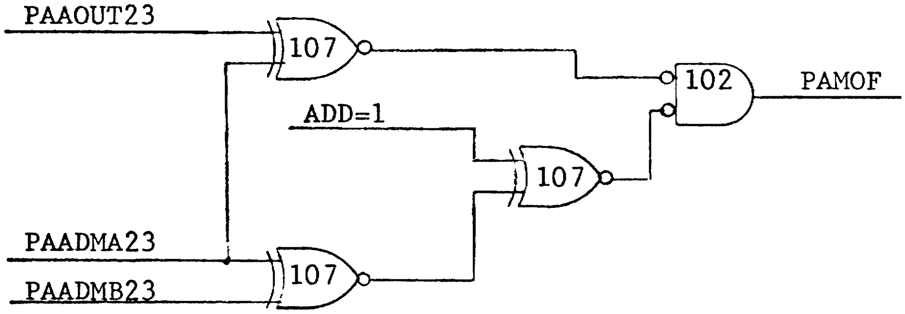
The $A_i = B_i = 1$ case is a don't care, as generate is a higher priority term.

4.4.3.4 Overflow Detectors

4.4.3.4.1 Mantissa Overflow

To detect the overflow conditions described in Section 3.2.1, this circuit is used:

Add = 1, Subtract = 0	PAADMA23	PAADMB23	PAAOUT23	PAMOF (Mantissa Overflow)
1	1	1	1	0
1	1	1	0	1
1	1	0	1	0
1	1	0	0	0
1	0	1	1	0
1	0	1	0	0
1	0	0	1	1
1	0	0	0	0
0	1	1	1	0
0	1	1	0	0
0	1	0	1	0
0	1	0	0	1
0	0	1	1	1
0	0	1	0	0
0	0	0	1	0
0	0	0	0	0



4.4.3.4.2 Exponent Overflow

During FP instructions, the four different types of overflow are detected separately from the exponent adder.

- Add underflow - addition of two negative numbers with a positive result

$$PAEXUF = PAADEA31 \cdot PAADEB31 \cdot \overline{PAAOUT31}$$

- Add overflow - addition of two positive numbers with a negative result

$$PAEXOF = \overline{PAADEA31} \cdot \overline{PAADEB31} \cdot PAAOUT31$$

- Subtract underflow - subtraction of a positive number from a negative number with a positive result

$$PASBUF = PAADEA31 \cdot \overline{PAADEB31} \cdot \overline{PAAOUT31}$$

- Subtract overflow - subtraction of a negative number from a positive number with a negative result

$$PASBOF = \overline{PAADEA31} \cdot PAADEB31 \cdot PAAOUT31$$

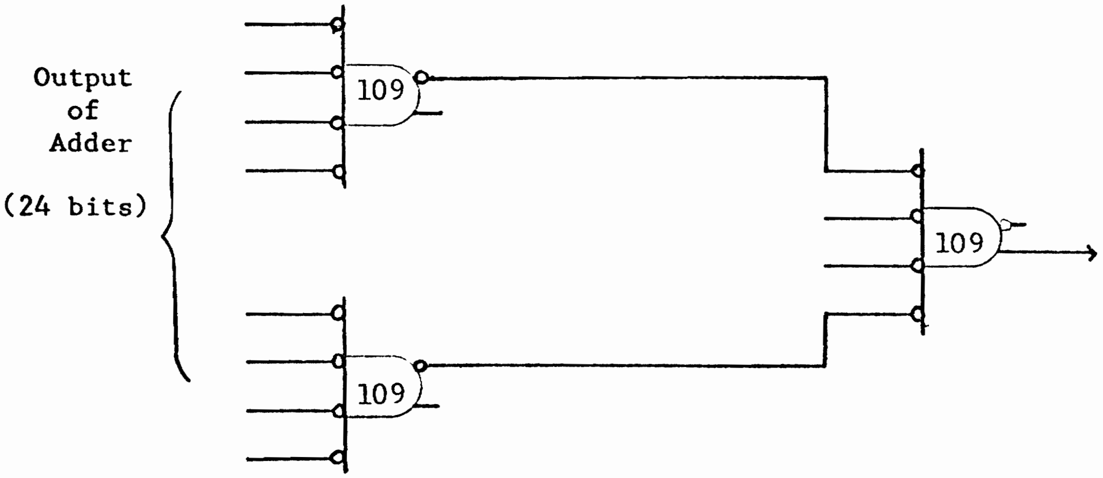
4.4.3.4.3 Overflow Flip-Flop

The overflow FF is set as a result of the above overflow conditions. See Section 4.5.8.

4.4.3.5 Adder Output Detector

4.4.3.5.1 Equal to Zero

All zeros are detected with a tree of negative input AND gates (10109):



4.4.3.5.2 Equal to Minus One

Minus One in two's complement notation is a word of all "1". It is detected by first inverting the adder output and then passing that through a circuit similar to the zeros detector.

4.4.3.5.3 Less Than Zero

Less than zero is detected by examining the sign bit. Negative two's complement numbers always have a "1" sign bit.

4.4.3.5.4 Greater Than Zero

Greater than zero is detected by ANDing "not equal to zero" and "not less than zero".

4.4.4 Alignment Network/Shifter

4.4.4.1 Normalize Decode Network (PANRMD)

A circuit designed to produce a count of the number of leading ones or zeros for the normalize function is attached to the output of CP11. The circuit compares the sign bit with each bit in the fraction starting at bit 22. The first mis-match found causes the count circuit to produce a five bit shift count.

The comparison is done with 10107 exclusive OR gates. The count is generated by selecting the first mis-match with 10165 Priority Encoders and converting that output to a count.

Figure 4.5 shows the circuit.

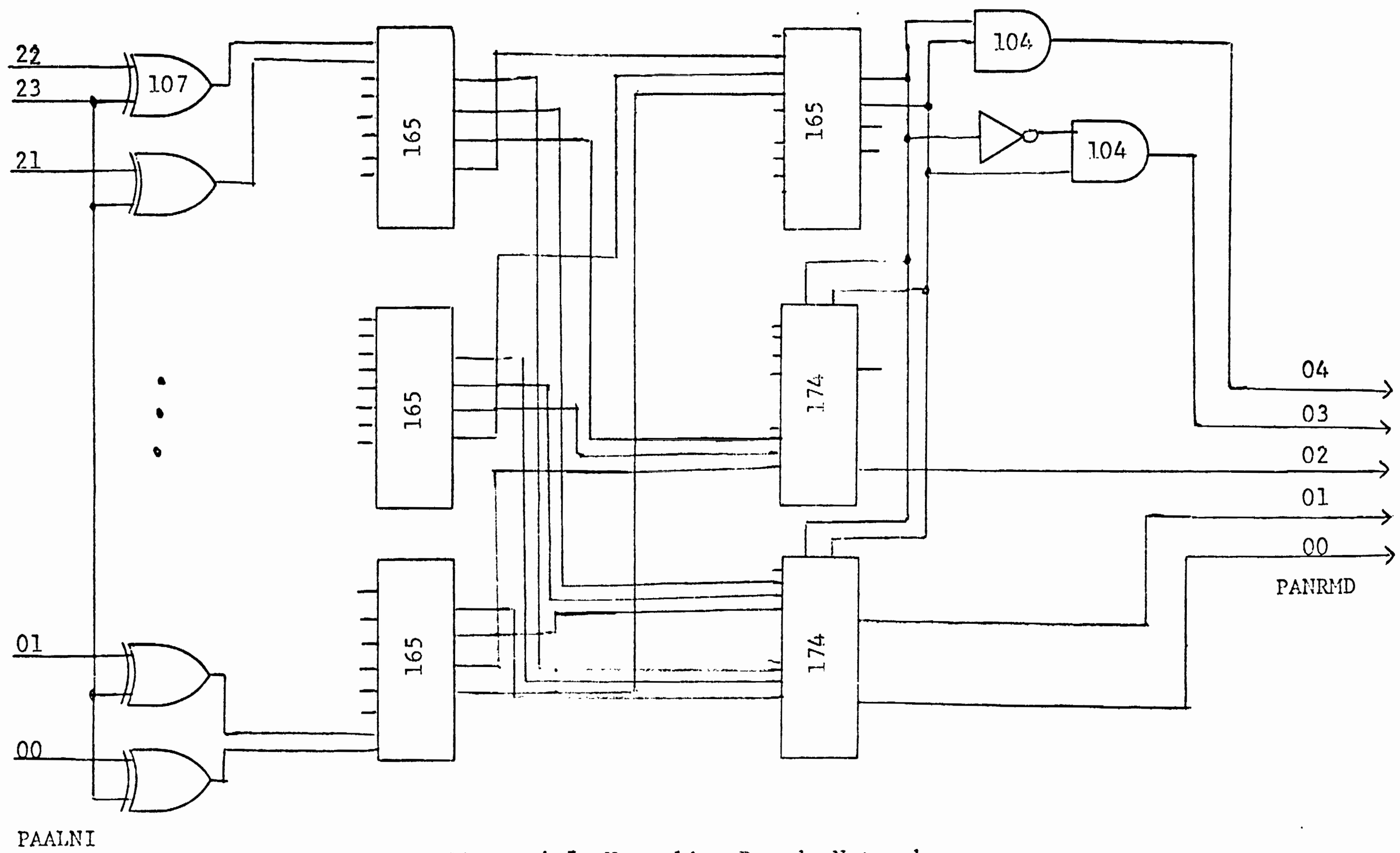


Figure 4.5 Normalize Decode Network

4.4.4.2 Two's Complement of Shift Count (PA2SCO)

The alignment network/shifter performs left shifts when presented with a negative shift count. The PANRMD produces a positive count, but for normalization, a left shift is required. This circuit, which is merely a 10181 ALU, subtracts the Shift Count Register from zero to produce its negative value.

4.4.4.3 The Shifter

The shifter is designed as a multi-level network capable of any shift in one clock cycle.

4.4.4.3.1 Stage One

Stage one generates a 63 bit shift operand. Input is the result of CP11 and CP12, PAALNI(31-0). Output is PAALNI and PAALNA(31-0). Control for this stage is:

PAMORL 29	PASCR(5) (Output of CP14)	FUNCTION
0	0	PAALNI23 duplicated 31 times (for sign extension of arithmetic right shift)
0	1	PAALNI(31-0)
1	0	
1	1	

4.4.4.3.2 Stage Two

Stage two receives the 63 bit shift operand and performs right shifts and left shifts with zero fill. Figure 4.6 shows stages one and two. Control for stage two are bits PASCR(5-3) which are output from CP14.

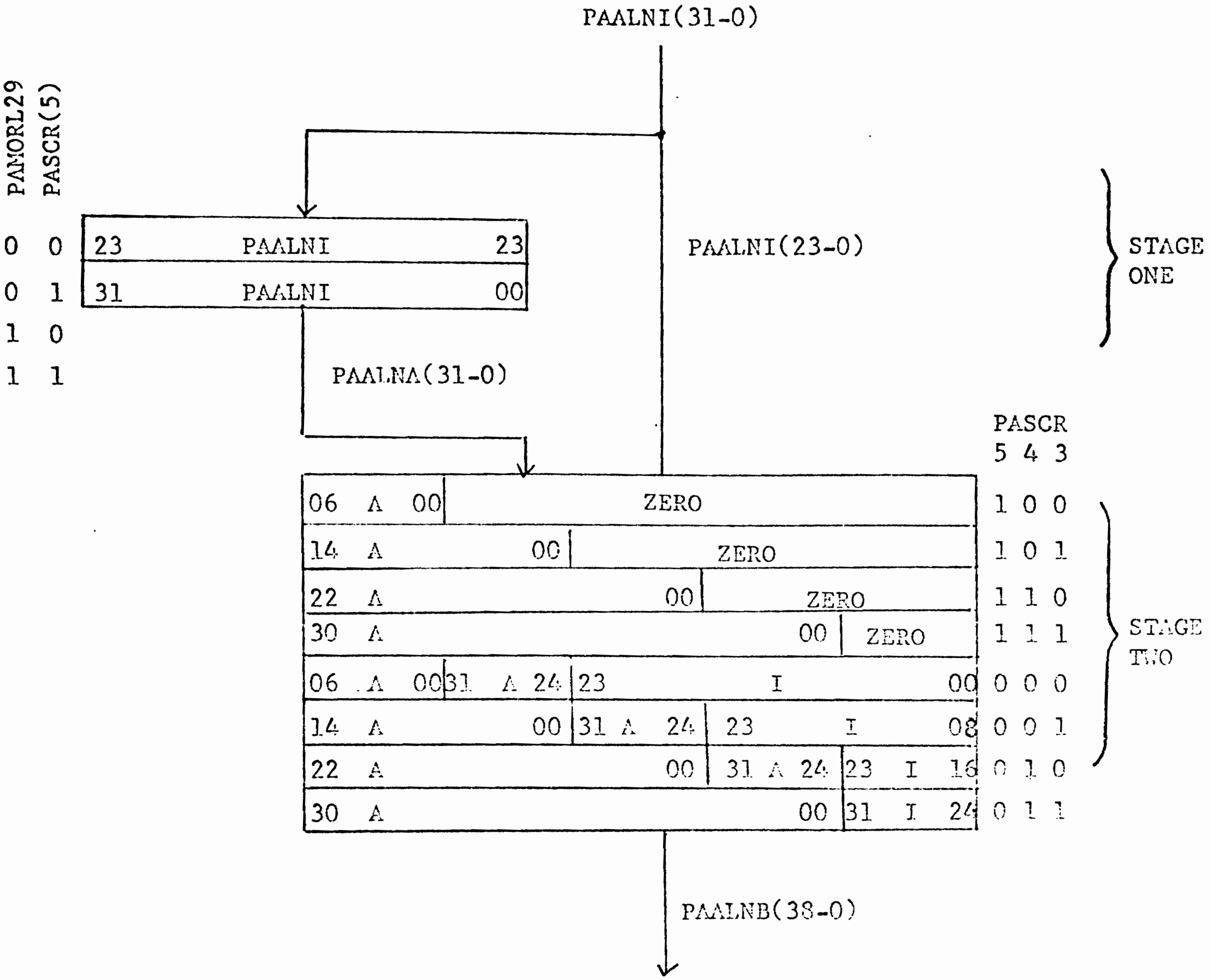


Figure 4.6 Stages One and Two of Alignment Network

PASCR 5 4 3	EFFECTIVE SHIFT (DECIMAL) POSITIVE COUNT	 NEGATIVE COUNT
1 0 0	-	32 left ZF
1 0 1	-	24 left ZF
1 1 0	-	16 left ZF
1 1 1	-	8 left ZF
0 0 0	0 right	-
0 0 1	8 right	-
0 1 0	16 right	-
0 1 1	24 right	-

ZF = Zero Fill

4.4.4.3.3 Stage Three

Stage three receives the 39 bit operand from stage two. Figure 4.7 shows stages three and four. Control for this stage are bits PASCR(2-1) which are output from CP14.

PASCR 2 1	EFFECTIVE SHIFT POSITIVE COUNT	 NEGATIVE COUNT
0 0	0	0
0 1	2 right	2 right
1 0	4 right	4 right
1 1	6 right	6 right

4.4.4.3.3 Stage Four

Stage four receives the 33 bit operand from stage three. Figure 4.7 shows this stage. Control is bit PASCR(0) from CP14.

PASCR 0	EFFECTIVE SHIFT POSITIVE COUNT	 NEGATIVE COUNT
0	0	0
1	1 right	1 right

Output from stage four is PAALNO(31-0) which is the final shifted output.

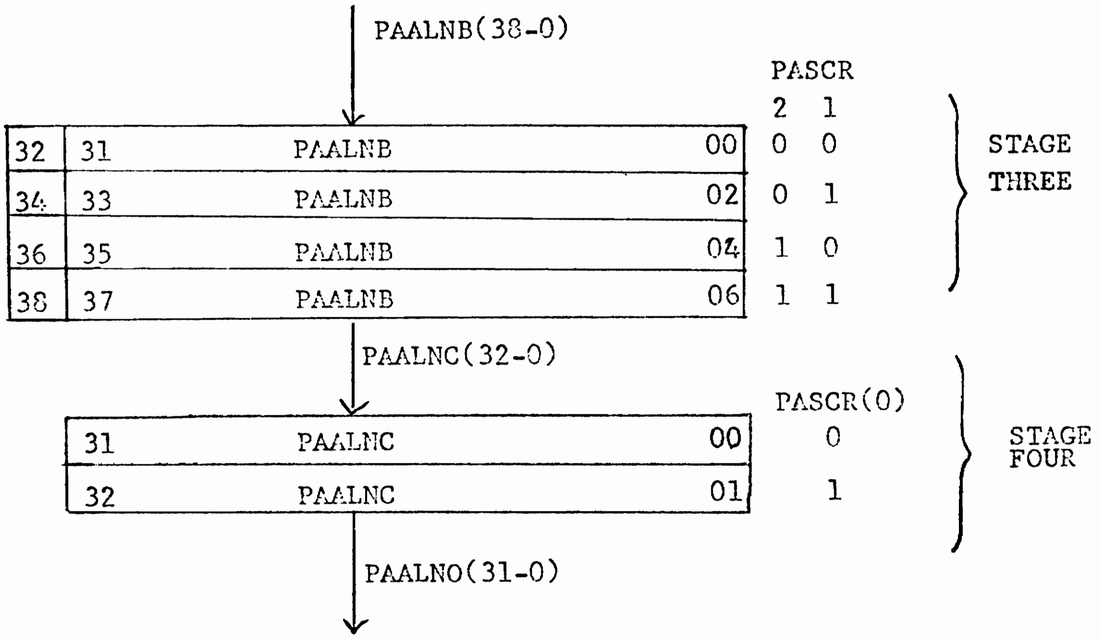


Figure 4.7 Stages Three and Four of Alignment Netowrk

4.4.4.4 Shifter Operation

The shift is the logical sum of the shifts performed by stages two, three and four. For right shifts, stage two shifts right an amount equal to the multiple of eight nearest but not less than the shift amount. Stages three and four then complete the shift.

Examples:

Shift Amount	Stage 2	Stage 3	Stage 4
0	0	0	0
+8	+8	0	0
+12	+8	+4	0
+29	+24	+4	+1
-4	-8	+4	0
-16	-16	0	0
-25	-32	+6	+1

Where "+" is a right shift and "-" is a left shift.

4.5 Local Control

Global control signals PAMORL(28-24) enable special logic in the AU. This is required because the data in each PE can cause slightly different execution of some instructions. Table 4.5 lists the 32 possible decodings of these lines, the instructions affected by each and the function carried out.

Following is a description of the local control modification produced by the PAMORL(28-24) lines.

4.5.1 ALU Control

IF PAMORL(28-24)=06 • PAQREG01, select subtract
IF PAMORL(28-24)=06 • $\overline{\text{PAQREG01}}$, select add
IF PAMORL(28-24)=18 • PAAREG23, select add
IF PAMORL(28-24)=18 • $\overline{\text{PAAREG23}}$, select subtract

C_{in} = PAMORL03 • PAMORL(28-24) ≠ (06+18+26) +
PAMORL(28-24)=06 • PACOXT +
PAMORL(28-24)=18 • $\overline{\text{PAAREG23}}$ +
PAMORL(28-24)=26 • PADPC
IF PAMORL(28-24)=6 or 18, PAMORL(3-0) must be 0.
IF PAMORL(28-24)=26
PAMORL(03) must be 0

4.5.2 Input A Mantissa Adder CP9

PAMORL04 = PAMORL04 + PAMORL(28-24)=02 • $\overline{\text{PAZROA}}$
PAMORL05 = PAMORL05 + PAMORL(28-24)=02 • PAZROA
PAMORL06 = PAMORL06 + PAMORL(28-24)=02

IF PAMORL(28-24)=2, PAMORL(6-4) must be 0.

4.5.3 Input B Mantissa Adder CP10

PAMORL07 = PAMORL07 + PAMORL(28-24)=02 • PAZROB +
PAMORL(28-24) = 06 (PAQREG00=PAQREG01 +
PAQREG00 ≠ PAQEXT)
PAMORL08 = PAMORL08 + PAMQRL(28-24)=02 +
PAMORL(28-24)=06 (PAQREG01 • PAQREG00 •
PAQEXT+PAQREG01 • PAQREG00 • PAQEXT)

IF PAMORL(28-24)=2 or 6, PAMORL(8-7) must be 0.

	PAMORL					USED ON		
DECODE	28	27	26	25	24	INST.	STEP	FUNCTION
00	0	0	0	0	0	-	-	Rest State
01	0	0	0	0	1	ADD SB ML ADE	7 7 18 2	A Register mantissa and exponent select as a function of PAEXUF. Also IF PAEXOF, PAOVFF ← 1
02	0	0	0	1	0	ADD SB	4 4	Adder mantissa A input select as a function of PAZROA Adder mantissa B input select as a function of PAZROB
03	0	0	0	1	1	FIX	1	Clock enable, A mantissa as a function of $\overline{\text{PAAREG31}}$. Also IF PAAOUT31 • $\overline{\text{PAAREG31}}$, PAOVFF ← 1
04	0	0	1	0	0	ADI SBI LINA LDEA	2 2 2 2	IF PAMOF, PAOVFF ← 1
05	0	0	1	0	1	ADD SB ML DV	5 5 16 7	A register bit 23 select as a function of PACOFF23. Also IF PATOVF • PAEXOF, PAOVFF ← 1

06	0	0	1	1	0	ML
						MLI
							3-14
							3-14
								Add/subtract control and adder mantissa control. Also
								IF DECODE=1,2,5 or 6,
								PAADXB ← PAAREG(00)
								IF DECODE=0,3,4 or 7,
								PAADXB ← zero. Also
								IF PAMOF, PABREG(23,22) ← PACOUT(23,23)
								IF PAMOF, PABREG(23,22) ← PAAOUT(23,23).
								Also enable PAQEXT ← PAQREG(01)

						ML	14	IF PAMOF, PATOVF ← 1 Also
								IF PAMOF, PACOFF23 ← PACOUT23

						MLI	14	IF PAMLOF, PAOVFF ← 1
07	0	0	1	1	1	MLI	15	Clock enable, A mantissa as a function of PAOVFF
08	0	1	0	0	0	DV	32	A register, mantissa input select as a function of PAAOUT23. Also Q register bit 0 input as a function of PAAOUT23.

TABLE 4.5, PAMORL(28-24) DECODE TABLE

	PAMORL					USED	ON	
DECODE	28	27	26	25	24	INST.	STEP	FUNCTION
09	0	1	0	0	1	DV	33 34	Clock enable, A mantissa and clock enable, Q mantissa as a function of $\overline{\text{PASIGN}}$
10	0	1	0	1	0	DV	2	IF PAAOUT(23-0) = zero, PAOVFF ← 1
11	0	1	0	1	1	DV	3	Enable PASIGN ← PAAOUT23
12	0	1	1	0	0	DV	4 6	Clock enable, A mantissa as a function of $\overline{\text{PABREG23}}$
13	0	1	1	0	1	DV	5	Clock enable, B mantissa and exponent. Also IF PATOVF • PAEXOF, PAOVFF ← 1.
14	0	1	1	1	0	DV	8	Clock enable, A mantissa and exponent as a function of PAAOUT23. Also IF $\overline{\text{PAAOUT23}}$ • PASBOF, PAOVFF ← 1
15	0	1	1	1	1	DV	9-31	A register input select as a function of PAAOUT23. Also Q register, mantissa bit 0 input as a function of PAAOUT23
16	1	0	0	0	0	SQ	1	IF PAAREG23, PAOVFF ← 1
17	1	0	0	0	1	SQ	2	Clock enable, A mantissa and exponent as a function of PAAREG24. Also IF PAAREG24 • PAEXOF, PAOVFF ← 1

18	1	0	0	1	0	SQ	5-28	Adder control as follows:
								IF PAAREG23 select SUBTRACT
								IF PAAREG23 select ADD.
								Also B register input select as a function of PAAREG23

19	1	0	0	1	1	ADD SB ML DV SQ FLOT	8 8 19 38 32 3	Clock enable, A exponent as a function of PAAOUT(23-0) ≠ zero
20	1	0	1	0	0	DV SBE	36 2	Clock enable, A mantissa and A mantissa input select as a function of PASBUF. Also IF PASBOF, PAOVFF ← 1

TABLE 4.5, CONTINUED

DECODE	PAMORL 28	 27	 26	 25	 24	USED ON INST.	 STEP	FUNCTION
21	1	0	1	0	1	ML	15	Clock enable and A mantissa input select as a function of PAEXUF. Also IF PAEXOF, PAOVFF← 1
22	1	0	1	1	0	ADD SB	3 3	Alignment network control, Alignment network input select, clock enable A exponent and mantissa and clock enable B mantissa
23	1	0	1	1	1	STAG	1	Enable PAXDOT(31-0)← PAAOUT(31-0) Also PADWSL ← 1
24	1	1	0	0	0	SL	1	B register, exponent input select as a function of PAAREG23. Also PAALFA← $\overline{\text{PAAOUT23}}$ PABETA← PAAOUT31
25	1	1	0	0	1	LADI LSBI	2 2	IF PACOUT22, PADPC← 1 OTHERWISE PADPC← 0
26	1	1	0	1	0	UADI USBI	2 2	ADDER C IN← PADPC. Also IF PAMOF, PAOVFF ← 1
27	1	1	0	1	1	SH	1	B register, exponent input select as a function of PAAREG23. Also PAALFA← $\overline{\text{PAAOUT23}}$ PABETA← PAAOUT31
28	1	1	1	0	0	ADD SB ROV	2 2 1	IF PAAOUT(31-24) > 23 OR IF PASBOF, PAZROB ←1. Also IF $\overline{\text{PAAOUT(31-24)}}$ > 23 AND IF $\overline{\text{PASBOF}}$, PAZROB← 0. Also IF PAAOUT(31-24) < -23 OR IF PASBUF, PAZROA← 1. Also IF PAAOUT(31-24) < -23 AND IF $\overline{\text{PASBUF}}$, PAZROA←0. Also PAOVFF← 0
29	1	1	1	0	1	SH SL	3-35 (ODD)	IF ($\overline{\text{PABITA}}$ • PABITZ + $\overline{\text{PABITA}}$ • $\overline{\text{PABITB}}$ PABITY + $\overline{\text{PABITA}}$ • $\overline{\text{PABITB}}$ • $\overline{\text{PABITC}}$ • PABITZ) Then PAEACT← 0. Also PAALFA← PABREG31. Also PABETA← PABREG30.
30	1	1	1	1	0	SFF	1	PAFALT← 1
31	1	1	1	1	1	CF	1	PAFALT← 0

TABLE 4.5, CONTINUED

4.5.4 A Register Mantissa Input CP4

$$\begin{aligned}
\mathrm{PAMORL14} &= \mathrm{PAMORL14} + \mathrm{PAMORL(28\text{-}24)=08} \cdot \mathrm{PAAOUT23}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=15} \cdot \mathrm{PAAOUT23}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=01} \cdot \overline{\mathrm{PAEXUF}}\\
\mathrm{PAMORL15} &= \mathrm{PAMORL15}+\mathrm{PAMORL(28\text{-}24)=08} \cdot \overline{\mathrm{PAAOUT23}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=01} \cdot \mathrm{PAEXUF}
\end{aligned}$$

IF PAMORL(28-24)=1, 8, or 15, PAMORL(15-14) must be 0.

4.5.5 A Register Exponent Input CP3

$$\begin{aligned}
\mathrm{PAMORL18} &= \mathrm{PAMORL18}+\mathrm{PAMORL(28\text{-}24)=01} \cdot \overline{\mathrm{PAEXUF}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=20} \cdot \overline{\mathrm{PASBUF}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=21} \cdot \overline{\mathrm{PAEXUF}}
\end{aligned}$$

IF PAMORL(28-24)=1, 20, or 21, PAMORL(18-16) must be 0.

4.5.6 Strobe A Register Mantissa

Enable Strobe if X where

$$\begin{aligned}
X &= \mathrm{PAMORL(28\text{-}24)=03} \cdot \overline{\mathrm{PAAREG31}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=05} \cdot \overline{\mathrm{PATOVF}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=07} \cdot \mathrm{PAOVFF}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=09} \cdot \overline{\mathrm{PASIGN}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=12} \cdot \overline{\mathrm{PAAREG23}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=14} \cdot \mathrm{PAAOUT23}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=17} \cdot \overline{\mathrm{PAAREG24}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=20} \cdot \overline{\mathrm{PASBUF}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=21} \cdot \overline{\mathrm{PAEXUF}}+\\
&\quad \mathrm{PAMORL33} \cdot (\overline{\mathrm{PAAREG23}}+\overline{\mathrm{PABREG23}})+\\
&\quad \mathrm{PAMORL(28\text{-}24)=22} \cdot (\overline{\mathrm{PASHCR05} \cdot \overline{\mathrm{PAZROA}} \cdot \overline{\mathrm{PAZROB}}})
\end{aligned}$$

IF PAMORL(28-24) equals any of the above values or if

PAMORL33, PAMORL19 must be = 1

4.5.6 Strobe A Register Exponent

Enable strobe of X

$$\begin{aligned}
\text{where } X &= \mathrm{PAMORL(28\text{-}24)=05} \cdot \overline{\mathrm{PATOVF}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=14} \cdot \mathrm{PAAOUT23}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=17} \cdot \overline{\mathrm{PAAREG24}}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=19} \cdot \mathrm{PAAOUT(23\text{-}0)} \neq \mathrm{ZERO}+\\
&\quad \mathrm{PAMORL(28\text{-}24)=22} \cdot \mathrm{PASBOF} + (\overline{\mathrm{PASBUF}} \cdot \overline{\mathrm{PAAOUT31}})
\end{aligned}$$

IF PAMORL(28-24) equals any of the above values, PAMORL20 must be = 1.

4.5.7 Alignment Network/Shifter CP11 and CP12

PAMORL30 = PAMORL30 + PAMORL(28-24) = 22 · PASCHRO5

IF PAMORL(28-24) = 22, PAMORL(31-30) must be 0.

4.5.8 Alignment Network/Shifter, Shift Distance Selector CP14

PAMORL32 = PAMORL32+PAMORL(28-24)=22 · PASHCRO5

IF PAMORL(28-24)=22, PAMORL32 must be 0.

4.5.9 Strobe Overflow FF

PAOVFF 0 IF STROBE · PAMORL(28-24) = 28
PAOVFF 1 IF STROBE · PAMORL(28-24) =

01 · PAEXOF +
03 · PAAOUT31 +
04 · PAMOF +
05 · PATOVF · PAEXOF +
06 · PAMLOF +
10 · PAAOUT(23-0) = zero +
13 · PATOVF · PAEXOF +
14 · PASBOF · $\overline{\text{PAAOUT23}}$ +
16 · PAAREG23 +
17 · PAEXOF · PAAREG24 +
20 · PASBOF +
21 · PAEXOF +
26 · PAMOF

IF PAMORL(28-26) equals any of the above values, PAMORL47 must be set to effect the change in PAOVFF

4.5.10 Strobe Q Mantissa

Enable Strobe if X

where X = (PAMORL(38-24) = 09) · $\overline{\text{PASIGN}}$

IF PAMORL(28-24) = 9, PAMORL36 MUST BE 1.

4.5.11 Strobe B Exponent

Enable strobe if X

where X = (PAMORL(28-24) ≠ 13) + PATOVF

4.5.12 Strobe B Mantissa

Enable strobe if X

where X = PAMORL(28-24) = 12 • $\overline{\text{PABREG23}}$ +
PAMORL(28-24) = 13 • $\overline{\text{PATOVF}}$ +
PAMORL(28-24) = 22 • (PASHCR05 + PAXROA + PAZROB)

IF PAMORL(28-24) = 12, 13 or 22, PAMORL42 and 43 must be 1

4.5.13 B Register Bits 23-22 Input CP21

Control lines locally generated.

PAMOVB00 = (PAMORL(28-24) = 6) • PAMOF

PAMOVB01 = (PAMORL(28-24) = 13)

4.5.14 SQ Result Selector CP22

Control lines locally generated.

PASQRT01 = (PAMORL(28-24) = 18)

PASQRT00 = (PAMORL(28-24) = 18) • PAAREG(23)

4.5.15 Q Extension Bit Input CP23

Control is locally generated.

PAQSEL = (PAMORL(28-24) = 6)

4.5.16 Extension Adder B Input CP24

Control is locally generated.

PAAXB = PAQXT (Q Extension FF)

4.5.17 DPC Input CP25

Control is locally generated.

PADPIN = (PAMORL(28-24) = 25)

4.5.18 Strobe Tag Register

Enable strobe if X

where X = PAMORL(39-38) = 1 • $\overline{\text{PAEACT}}$ +

PAMORL(39-38) = 2 • PAEACT +

4.5.19 Control Line Select Signals

4.5.19.1 Control signal, data time shared with control

PAMORL 48	Function
1	Control lines PAMORL(31-0) contain data
0	Control lines PAMORL(31-0) contain control

4.5.19.2 Control signal, select control group

PAMORL 49	Function
0	Control lines PAMORL(44-32) contain group 2 control
1	Control lines PAMORL(44-32) contain group 1 control

4.6 Double Integer Hardware

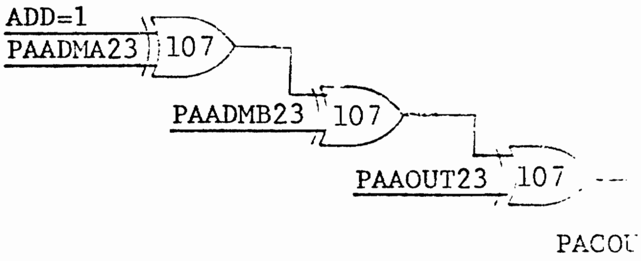
4.6.1 Double Precision Carry FF

Used to hold the carry or borrow between double integer instructions. Input is from a circuit which detects the carry from bit position 22 of the ALU. See Section 4.3.2.25.

4.6.2 Carry-out Bit 22

Since the ALU is constructed of 4 bit wide 10181 ALU chips, the carry between bits 22 and 23 is completely within the chip. The following circuit was designed to reconstruct that line outside of the chip.

ADD = 1, SUBTRACT = 0	PAADMA23	PAADMB23	PAAOUT23	PACOUT 22
1	1	1	1	1
1	1	1	0	0
1	1	0	1	0
1	1	0	0	1
1	0	1	1	0
1	0	1	0	1
1	0	0	1	1
1	0	0	0	0
0	1	1	1	0
0	1	1	0	1
0	1	0	1	1
0	1	0	0	0
0	0	1	1	1
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1



4.7 Floating Point Hardware

4.7.1 PAZROA and PAZROB

These FF are used during the FP ADD and SB instruction. As explained in Section 3.4.2, when an addition of two numbers where one is more than 2^{23} greater or less than the other causes a shift of more than 23 places, special handling is required.

In micro-step two, after the exponents are subtracted, the result is compared to 23. If the result is greater than 23, PAZROB is set. If the result is less than -23, PAZROA is set. Then in micro-step four, the appropriate input to the adder is zeroed out.

4.7.1.1 Greater Than 23 Detector

The circuit for the > 23 detector is shown in Figure 4.8.

4.7.1.2 Less Than -23 Detector

The circuit for the < -23 detector is shown in Figure 4.9.

4.7.2 Extension Hardware

The multiply algorithm described in Section 3.4.3 requires a 25 bit adder. The additional bit is implemented by extending the B and Q Register one bit each and extending the adder one position. The extension is on the least significant bit end.

4.7.3 Sign FF PASIGN

This FF is used during the DV instruction to save the sign of the result. As described in Section 3.4.4, the DV algorithm only divides two positive numbers. If the hardware must take the two's complement of one of the operands, it sets the sign FF. Then following the divide, if the sign FF is set, the two's complement of the quotient is taken.

PAAOUT									
31	30	29	28	27	26	25	24	DECIMAL	FUNCTION
1	0	0	0	0	0	0	0	-128	0
1	1	1	1	1	1	1	1	-1	0
0	0	0	0	0	0	0	0	0	0
				⋮				⋮	⋮
0	0	0	1	0	1	1	0	22	0
0	0	0	1	0	1	1	1	23	0
0	0	0	1	1	0	0	0	24	1
0	0	0	1	1	0	0	1	25	1
				⋮				⋮	⋮
0	0	1	0	0	0	0	0	32	1
				⋮				⋮	⋮
0	1	0	0	0	0	0	0	64	1
				⋮				⋮	⋮
0	1	1	1	1	1	1	1	127	1

Function = $\overline{31} \cdot (30 + 29 + (28 \cdot 27))$

TESTS .GE. 24

TESTS .GE. 32

TESTS .GE. 0

Figure 4.8 Adder Output Greater Than 23 Detector

PAAOUT									
31	30	29	28	27	26	25	24	DECIMAL	FUNCTION
0	1	1	1	1	1	1	1	127	0
			⋮					⋮	⋮
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	-1	0
			⋮					⋮	⋮
1	1	1	0	1	1	0	0	-20	0
1	1	1	0	1	0	1	1	-21	0
1	1	1	0	1	0	1	0	-22	0
1	1	1	0	1	0	0	1	-23	0
1	1	1	0	1	0	0	0	-24	1
1	1	1	0	0	1	1	1	-25	1
1	1	1	0	0	1	1	0	-26	1
			⋮					⋮	⋮
1	0	0	0	0	0	0	0	-128	1

$$\text{Function} = 31 \cdot (\overline{30} + \overline{29} + (\overline{28} \cdot \overline{27}) + (\overline{28} \cdot 27 \cdot \overline{26} \cdot \overline{25} \cdot \overline{24}))$$

TESTS .EQ. -24

TESTS .LT. -24

TESTS .LT. -32

TESTS .LT. 0

Figure 4.9 Adder Output Less Than -23 Detector

Input to the FF is bit 23 of the ALU output (the sign bit). The FF is clocked when local control senses PAMORL(28-24) = 11.

4.7.4 Temporary Overflow Hold FF PATOVF

This FF is used to temporarily store an overflow indication during ML, ADD, SB and DV instructions. When an overflow is detected during an FP instruction, the AU attempts to correct it by right shifting the fraction and incrementing the exponent. Only if the exponent then overflows, is the overflow made permanent and PAOVFF is set.

Input to the FF is the mantissa overflow detector PAMOF (Section 4.4.3.4.1).

4.7.5 Carry Out of Bit 23 FF PACOFF23

This FF holds the carry-out of the ALU during ML, ADD, SB and DV instructions. As described in the previous section, when the fraction overflows, a correction is attempted. This FF becomes the new bit 23 of the right shifted fraction.

Input is the carry out of the last ALU chip only if there is a mantissa overflow (PAMOF).

4.8 Instruction Implementation

Each of the following sections is centered around the micro-step sequence of some of the more interesting AU instructions of each of the six types.

Each sequence contains the following information:

Opcode - 8 bits, in octal

Instruction Name

Mnemonic

Description of the operation

Number of micro-steps in execution

Micro-step Sequence

In the sequence, this symbology is used:

← indicates that the register or data lines on the left receive the register or data lines on the right.

"Select" indicates the operation being performed by the ALU.

"iff" If and only if.

; Qualifier. The operation on the left is not performed unless the equation on the right is true.

, Concatenate. The bits on the left are combined with the bits on the right to produce a single value.

(XX-YY) Bits XX through YY of the named register or data path.

4.8.1 Activity Instructions

Instructions chosen as examples are SNZL, SNG, ANS, CAS, and PSEL.

4.8.1.1 SNZL

The A Register is compared to zero. If it is, the EA FF is reset. Figure 4.10.

4.8.1.2 SNG

An operand is formed in the B Register during micro-step one. During step 2, the A and B are exclusive "OR"ed by the ALU. The result is checked for zero and if it is, the EA FF is reset. Figure 4.11.

4.8.1.3 ANS

The top of the stack (PASTAK(0)) is "AND"ed with the EA FF and the result placed in the EA FF. The stack is then popped. Figure 4.12.

4.8.1.4 CAS

The top of the stack is set to "0". Figure 4.13.

4.8.1.5 PSEL

The EA is pushed into the activity stack. A 32 bit mask of zeros with a single "1" is generated in the PICU and sent to the AU B Register. A word is fetched from Element Memory and "AND"ed with the mask by the ALU. The output of the ALU is checked for all zeros, if so, the EA FF is reset. Figure 4.14.

4.8.2 Integer Instructions

Instructions chosen as examples are ADI, LSBI and USBI.

4.8.2.1 ADI

During the first micro-step the operand is formed in the B Register. In the second step, the A and B Registers are added together with the result placed in the A Register. Overflow is checked and the Overflow FF is set if necessary. Figure 4.15 shows the micro-step sequence.

OPCODE	INSTRUCTION	MNEMONIC
166	Select on Non-Zero Logical	SNZL

DESCRIPTION

Set EA to zero in those active AUs in which A-register (31-0) equal zero

Execution: 1 step

MICRO STEP SEQUENCE

```
1)  PAADEA(31-24) ←— PAAREG(31-24)
    PAADEB(31-24) ←— 0
    PAADMA(23-0)  ←— PAAREG(23-0)
    PAADMB(23-0)  ←— 0
    Select OR
    PAEACT ←——       0; iff PAAOUT(31-0)=0
```

FIGURE 4.10, SNZL

OPCODE	INSTRUCTION	MNEMONIC
167	Select on Not Equal Global	SNG

DESCRIPTION

Set EA to zero in those active AUs in which A-Register (31-0) are equal to Operand (31-0)

Execution: 2 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
        PADMDE ←            0
        PADAIN(10-0) ←      PAMOBR(10-0)
        Transmit PADREQ to EMC
        PAXDIN(31-0) ←      PXEMOT(31-0); when PADSEL=1
    If R=0:
        PAMORL(31-0) ←      PAMOBR(31-0)
        PAXDIN(31-0) ←      PAMORL(31-0)
    PABREG(31-0) ←          PAXDIN(31-0);
2)  PAADEA(31-24) ←         PAAREG(31-24)
    PAADEB(31-24) ←         PABREG(31-24)
    PAADMB(23-0) ←          PABREG(23-0)
    PAADMA(23-0) ←          PAAREG(23-0)
    Select EXCLUSIVE OR
    PAEACT ←                0; iff PAAOUT(31-0)=0
```

FIGURE 4.11, SNG

OPCODE	INSTRUCTION	MNEMONIC
252	AND of STACK	ANS

DESCRIPTION

In all ensemble AUs:

PAEACT ← PAEACT · AND · PASTAK(0)

PASTAK(I) ← PASTAK(I + 1) ; I = 0, ---, 19

PASTAK(20) ← 0

Execution: 1 step

MICRO STEP SEQUENCE

1) PAEACT ← (PAEACT) · (PASTAK(0))

PASTAK(20-0) ← (0, PASTAK(20-1))

FIGURE 4.12, ANS

OPCODE	INSTRUCTION	MNEMONIC
253	Clear Active Stack	CAS

DESCRIPTION

In all active AUs

PASTAK(0) ← 0

Execution: 1 step

MICRO STEP SEQUENCE

1) PASTAK(20-0) ← PASTAK(20-1), 0; iff PAEACT = 1

FIGURE 4.13, CAS

OPCODE	INSTRUCTION	MNEMONIC
013	Push and Select on Bit	PSEL

DESCRIPTION

In all ensemble AUs, EA is pushed into the activity stack as follows:

```
    PASTAK(20-1)←  PASTAK(19-0)

    PASTAK(0)←  PAEACT
```

If PASTAK(19)=1, prior to "Push", the overflow error is generated to the ICL.

After PUSH is complete, select on bit (where PAEACT = 1) as follows:

> Reset PAEACT in AUs where the bit specified by Operand (15-11) in memory word specified by Operand (10-0) is zero.

Execution: 5 steps

MICRO STEP SEQUENCE

```
1)  PASTAK(20-1)←  PASTAK(19-0)
    PASTAK(0)←  PAEACT
    If PASTAK(19)=1; generate overflow error to ICL
2)  PAMORL(31-0)←  Mask Generate Logic (31-0)
    PAXDIN(31-0)←  PAMORL(31-0)
    PABREG(31-0)←  PAXDIN(31-0)
3)  PAAREG(31-0)←  PABREG(31-0)
4)  PADAIN(10-0)←  PAMOBR(10-0)
    PADMDE←  0
    Transmit PADREQ to EMC
    PAXDIN(31-0)←  PXEMOT(31-0) ; when PADSEL = 1
    PABREG(31-0)←  PAXDIN(31-0)
5)  PAADEA(31-24)← PAAREG(31-24)
    PAADEB(31-24)← PABREG(31-24)
    PAADMA(23-0)←  PAAREG(23-0)
    PAADMB(23-0)←  PABREG(23-0)
    Select AND
    PAEACT←  0 ; iff (PAEACT = 1) • (PAAOUT(31-0) = 0)
    PAEACT ←  PAEACT ; iff (PAEACT = 0) + (PAAOUT(31-0) ≠ 0)
```

FIGURE 4.14, PSEL

OPCODE	INSTRUCTION	MNEMONIC
220	Add Integer	ADI

DESCRIPTION

```
In all active AUs'
PAAREG(23-0)  ←    PAAREG(23-0) + OPERAND(23-0)
PAAREG(31-24) ←    0
```

Execution: 2 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
       PADMDE ←            0
       PADAIN(10-0) ←      PAMOBR(10-0)
       Transmit PADREQ to EMC
       PAXDIN(31-0) ←      PXEMOT(31-0); when PADSEL = 1
    If R=0:
       PAMORL(31-0) ←      PAMOBR(31-0)
       PAXDIN(31-0) ←      PAMORL(31-0)
    PABREG(31-0) ←         PAXDIN(31-0); iff PAEACT=1

2)  PAADMA(23-0) ←         PAAREG(23-0)
    PAADMB(23-0) ←         PABREG(23-0)
    Select ADD
    PAAREG(23-0) ←         PAAOUT(23-0); iff PAEACT=1
    PAOVFF ←               1; iff (PAEACT=1) • (overflow)
    PAAREG(31-24) ←        0; iff PAEACT=1
```

FIGURE 4.15, ADI

4.8.2.2 LSBI and USBI

During the first micro-step of both instructions the operand is formed. The second micro-step of the LSBI subtracts the two operands and saves the borrow in the DPC FF. The second step of the USBI subtracts its two operands using the DPC FF as a borrow into the low order bit of the ALU. Overflow is checked and the Overflow FF is set if necessary. Figures 4.16 and 4.17 show the micro-step sequences.

4.8.3 Logical/Data Transfer Instructions

Instructions chosen as examples are ORA and TAQ.

4.8.3.1 ORA

During the first micro-step, the explicit operand is loaded into the B Register. Then, the A and B Registers are "OR"ed in the ALU and returned to the A Register. Figure 4.18 shows the micro-step sequence.

4.8.3.2 TAQ

In this instruction, the A Register is routed to the input of the Q Register which is then clocked. Figure 4.19 shows the micro-step sequence.

4.8.4 Floating Point Instructions

Instructions chosen as examples are FIX, FLOT, ADD, ML, DV and SQ. These instructions were described in detail in Section 3.4. The micro-steps in this section follow those algorithms.

4.8.4.1 FIX/FLOT

In the FIX instruction, the exponent is compared to 23. If it is greater than 23, the overflow flip-flop is set. The A Register is then shifted until the exponent is zero and placed in the B Register. The B Register is then shifted right to its original position. The A and B Register are compared and the difference is placed in B. The A Register is then shifted again until the exponent is zero. If the difference in

OPCODE	INSTRUCTION	MNEMONIC
226	Lower Subtract Integer	LSBI

DESCRIPTION

```
In all active AUs;
PAAREG(23-0) ←    PAAREG(23-0) - OPERAND(23-0)
PAAREG(31-24)←    0
PAXDPC ←          1; iff borrow out
```

Execution: 2 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
       PADMDE ←             0
       PADAIN(10-0)←        PAMOBR(10-0)
       Transmit PADREQ to EMC
       PAXDIN(31-0)←        PXEMOT(31-0); when PADSEL=1
    If R=0:
       PAMORL(31-0)←        PAMOBR(31-0)
       PAXDIN(31-0) ←       PAMORL(31-0)
    PABREG(31-0) ←          PAXDIN(31-0)

2)  PAADMA(23-0) ←          PAAREG(23-0)
    PAADMB(23-0) ←          PAAREG(23-0)
    Select SUBTRACT
    PAAREG(23-0) ←          PAAOUT(23-0); iff PAEACT=1
    PAXDPC ←                1 iff (borrow out) • (PAEACT=1)
    PAAREG(31-24) ←         0; iff PAEACT=1
```

FIGURE 4.16, LSBI

OPCODE	INSTRUCTION	MNEMONIC
216	Upper Subtract Integer	USBI

DESCRIPTION

In all active AUs;

```
PAAREG(23-0) ←  PAAREG(23-0) - OPERAND(23-0) - PAXDPC
PAAREG(31-24) ←  0
```

Execution: 2 steps

MICRO STEP SEQUENCE

```
1)  If R=3:
        PADMDE ←           0
        PADAIN(10-0) ←     PAMOBR(10-0)
        Transmit PADREQ to EMC
        PAXDIN(31-0) ←     PXEMOT(31-0); when PADSEL=1
    If R≠3:
        PAMORL(31-0) ←     PAMOBR(31-0)
        PAXDIN(31-0) ←     PAMORL(31-0)
    PABREG(31-0) ←         PAXDIN(31-0)

2)  PAADMA(23-0) ←         PAAREG(23-0)
    PAADMB(23-0) ←         PABREG(23-0)
    Select SUBTRACT, carry-in = PAXDPC
    PAAREG(23-0) ←         PAAOUT(23-0); iff PAEACT=1
    PAOVFF ←               1 if overflow and PAEACT=1
    PAAREG(31-24) ←        0; iff PAEACT=1
```

FIGURE 4.17, USBI

OPCODE	INSTRUCTION	MNEMONIC
244	Logical OR	ORA

DESCRIPTION

In all active AUs;

PAAREG(31-0)← PAAREG(31-0).OR.OPERAND(31-0)

Execution: 2 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
       PADMDE←                  0
       PADAIN(10-0)←            PAMOBR(10-0)
       Transmit PADREQ to EMC
       PAXDIN(31-0)←            PXEMOT(31-0); when PADSEL=1
    If R=0:
       PAMORL(31-0)←            PAMOBR(31-0)
       PAXDIN(31-0)←            PAMORL(31-0)
    PABREG(31-0)←               PAXDIN(31-0)

2)  PAADEA(31-24)←              PAAREG(31-24)
    PAADEB(31-24)←              PABREG(31-24)
    PAADMA(23-0)←               PAAREG(23-0)
    PAADMB(23-0)←               PABREG(23-0)
    Select OR
    PAAREG(31-0)←               PAAOUT(31-0); iff PAEACT=1
```

FIGURE 4.18, ORA

OPCODE	INSTRUCTION	MNEMONIC
043	Transfer A to Q	TAQ

DESCRIPTION

AU A-Register 31-0 is moved to AU Q-Register 31-0 in those AUs which are active.

Execution: 1 step

MICRO STEP SEQUENCE

```
1)  PAADEA(31-24)←   PAAREG(31-24)
    PAADEB(31-24)←   0
    PAADMA(23-0) ←   PAAREG(23-0)
    PAADMB(23-0) ←   0
    Select OR
    PAQREG(31-0)←    PAAOUT(31-0); iff PAEACT = 1
```

FIGURE 4.19, TAQ

B is less than zero and the A Register is negative, the A Register is incremented by one to become the final integer value.

In the FLOT instruction, the number of leading ones/zeros is counted and placed in the Shift Count Register. This value is subtracted from 23 and the result becomes the exponent of the FP number. The fraction is then shifted right by the amount determined in step one to become the final fraction.

Figure 4.20 shows the micro-step sequence for FIX and 4.21 shows it for FLOT.

4.8.4.2 FP ADD/SB

The floating point add and subtract instructions require a binary point alignment step before the fractions of the two operands can be added or subtracted. During this alignment step, the exponents of the operands are differenced and the fraction with the smallest exponent is enabled, via local control flip-flops, into the alignment network. See Figure 4.22.

After the binary points are aligned and the fractions added (subtracted), the result is normalized and the exponent updated. If the result of the add (subtract) operation is zero, -128 is gated into the exponent of the result.

The floating point add and subtract instructions are identical except for micro-step 4 as shown in the flow chart (Figure 4.23).

OPCODE	INSTRUCTION	MNEMONIC
205	Fix	FIX

DESCRIPTION

In all active AU's, the floating point number contained in the A-Register (31-0) is converted to integer format in A-Register (23-0) with PAAREG(31-24) set to zero. It is assumed that the floating point mantissa has no overflow and is normalized. Truncation is performed as follows:

$$\pm m + \frac{x}{y} \rightarrow \pm m.$$

Execution: 5 steps

MICRO STEP SEQUENCE

```
1)  PAADEA(31-24)←    $23_{10}$
    PAADEB(31-24)←    PAAREG(31-24)
    PAADMA(23-0)←     0
    PAADMB(23-0)←     0
    Select SUBTRACT
    PASHCR(5-0) ←     PAAOUT(29-24); iff PAEACT=1
    PAOVFF←        1; iff (PAEACT=1) • (PAAOUT(31-24)<0)
    PAAREG(23-0)←     PAAOUT(23-0);
                      iff (PAEACT=1) • (PAAREG(31-24)<0)

2)  PAALNI(23-0)←     PAAREG(23-0)
    PASCR(5-0) ←      PASHCR(5-0)
    PAAREG(31-24) ←   0; iff PAEACT=1
    PABREG(23-0) ←    PAALNO(23-0)
```

FIGURE 4.20, FIX

```
3)  PAALNI(23-0)←      PABREG(23-0)
    PASCR(5-0)←        PA2SCO(5-0)
    PABREG(23-0)←      PAALNO(23-0)

4)  PAADMA(23-0)←      PABREG(23-0)
    PAADMB(23-0)←      PAAREG(23-0)
    Select SUBTRACT
    PABREG(23-0)←      PAAOUT(23-0)
    PASCR(5-0)←        PASHCR(5-0)
    PAALNI(23-0)←      PAAREG(23-0)
    PAAREG(23-0)←      PAALNO(23-0); iff PAEACT=1

5)  PAADMA(23-0)←      PAAREG(23-0)
    PAADMB(23-0)←      0
    Select INCREMENT
    PAAREG(23-0)←      PAAOUT(23-0); iff
            (PAEACT=1) • (PABREG(23-0) <0) • (PAAREG(23-0) <0)
```

OPCODE	INSTRUCTION	MNEMONIC
206	Float	FLOT

DESCRIPTION

In all active AUs, the integer number contained in the A-Register (31-0) is converted to normalized floating point format in A-Register (31-0).

Execution: 3 steps

MICRO STEP SEQUENCE

```
1)  PAALNI(23-0) ←     PAAREG(23-0)
    PASHCR(5-0) ←      PANRMD(5-0); iff PAEACT = 1

2)  PASCR(5-0) ←       PA2SCO(5-0)
    PAALNI(23-0) ←     PAAREG(23-0)
    PAAREG(23-0) ←     PAALNO(23-0); iff PAEACT=1
    PAADEA(31-24) ←    $+ 23_{10}$
    PAADEB(31-24) ←    PA2SCO(5,5,5-0)
    Select ADD
    PAAREG(31-24) ←    PAAOUT(31-24); iff PAEACT=1

3)  PAADMA(23-0) ←     PAAREG(23-0)
    PAADMB(23-0) ←     0
    PAADMB(23-0) ←     0
    Select ADD
    Iff PAAOUT(23-0) = 0:
       PAAREG(31-24) ← $- 128_{10}$; iff PAEACT=1
```

FIGURE 4.21, FLOT

OPCODE	INSTRUCTION	MNEMONIC
200	Add Floating Point	ADD

DESCRIPTION

In all active AUs, A-Register (31-0) is added to operand (31-0) and the sum placed in A-Register (31-0).

Execution: 8 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
       PADMDE ← 0
       PADAIN(10-0) ← PAMOBR(10-0)
       Transmit PADREQ to EMC
       PAXDIN(31-0) ← PXEMOT(31-0) ; when PADSEL = 1
    If R=0:
       PAMORL(31-0) ← PAMOBR(31-0)
       PAXDIN(31-0) ← PAMORL(31-0)
    PABREG(31-0) ← PAXDIN(31-0)

2)  PAADEA(31-24) ←   PAAREG(31-24)
    PAADEB(31-24) ←   PABREG(31-24)
    Select SUBTRACT
    PASCHR(5-0) ← PAAOUT(29-24); iff PAEACT=1
    PAZROB ← 1 ; if (PAAOUT(29-24) > 24) • (PAEACT = 1)
                 or exponent overflow : Else 0
    PAZROA ← 1 ; if (PAAOUT(29-24) < -23) • (PAEACT = 1)
                 or exponent underflow : Else 0

3)  If (PASHCR(5-0) > 0) • ($\overline{\text{PAZROA}}$) • ($\overline{\text{PAZROB}}$):
       PASCR ← PASHCR(5-0)
       PAALNI(23-0) ← PABREG(23-0)
       PABREG(23-0) ← PAALNO(23-0); iff PAEACT = 1
```

FIGURE 4.22

```
     If (PASHCR(5-0) < 0) • ($\overline{\text{PAZROA}}$) • ($\overline{\text{PAZROB}}$) = 1 :
        PASCR ←  PA2SCO
        PAALNI(23-0) ←  PAAREG(23-0)
        PAAREG(23-0) ←  PAALNO(23-0) ; iff PAEACT = 1
     If (PASHCR(5-0) < 0) • (PAEACT = 1) :
        PAAREG(31-24) ← PABREG(31-24); iff PAEACT=1

4)   PAADMA(23-0)←  ($\overline{\text{PAZROA}}$) • (PAAREG(23-0)
     PAADMB(23-0)←  ($\overline{\text{PAZROB}}$) • (PABREG(23-0))
     Select ADD
     PAAREG(23-0)←  PAAOUT(23-0) ; iff PAEACT = 1
     PATOVF ←  0  if  (no overflow) • (PAEACT = 1)
     If (overflow) • (PAEACT = 1):
        PATOVF ←  1
        PACO23FF ← CO23 (carry out of bit 23)
        PASHCR(5-0) ← 1 ; iff PAEACT - 1

5)   If (PATOVF = 1) • (PAEACT = 1):
        PASCR(5-0) ←  PASHCR(5-0)
        PAALNI(23-0) ←  PAAREG(23-0)
        PAADEA(31-24) ←  PAAREG(31-24)
        PAADEB(31-24) ←  0
        Select INCREMENT
        PAAREG(22-0) ←  PAALNO(22-0)
        PAAREG(23) ←  PACO23FF
        PAAREG(31-24) ←  PAAOUT(31-24)
        PAOVFF ← 1 ; iff overflow from exponent
     If  (PATOVF = 0) :
        No operation

6)   PAALNI(23-) ← PAAREG(23-0)
     PASHCR(5-0) ← PANRMD(5-0) ; iff PAEACT = 1
```

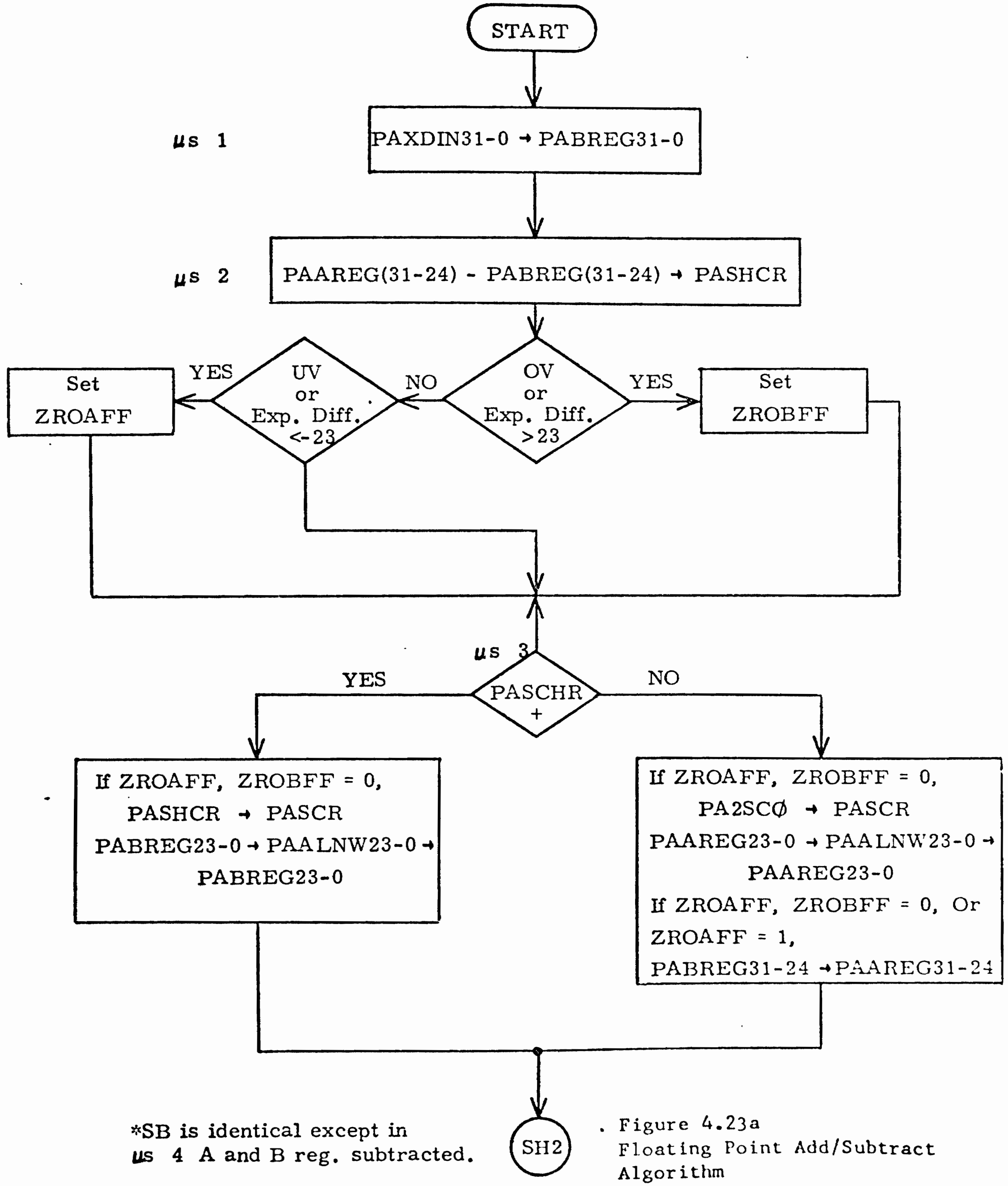
FIGURE 4.22, CONTINUED

```
7)  PASCR(5-0) ← PA2SCO(5-0)
    PAALNI(23-0) ← PAAREG(23-0)
    PAADEA(31-24) ← PAAREG(31-24)
    PAADEB(31-24) ← PA2SCO(5,5,5-0)
    PAADMA ← 0
    PAADMB ← 0
    Select ADD
    If no underflow from exponent:
       PAAREG(23-0) ← PAALNO(23-0) ; iff PAEACT = 1
       PAAREG(31-24) ← PAAOUT(31-24) ; iff PAEACT = 1
    If exponent(31) underflow:
       PAAREG(31-24) ← $-128_{10}$
       PAAREG(23-0) ← PAAOUT(23-0)

8)  PAADMA(23-0) ← PAAREG(23-0)
    PAADMB(23-0) ← 0
    Select ADD
    If  PAAOUT(23-0) = 0:
       PAAREG(31-24) ← -128 ; iff PAEACT = 1
```

FIGURE 4.22, CONTINUED

ADD*



*SB is identical except in µs 4 A and B reg. subtracted.

Figure 4.23a
Floating Point Add/Subtract
Algorithm

ADD

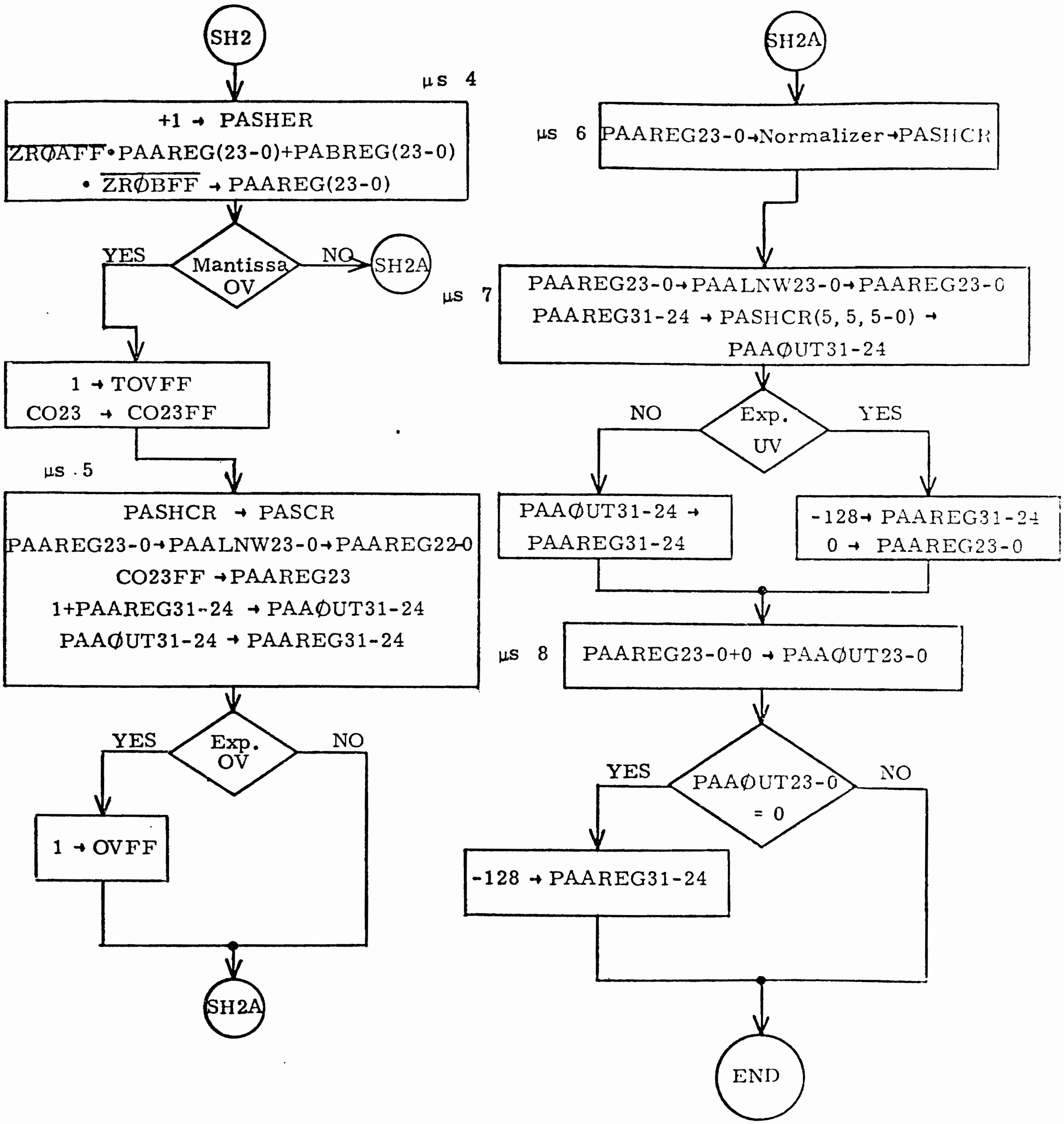


Figure 4.23b
Floating Point Add/Subtract
Algorithm

4.8.4.3 Multiply Floating Point

The method used in the execution of the FP multiply instruction is comprised of three functional categories. They are initialize, iterate, and terminate. See Figure 4.24.

During the initialization phase, the multiplier is received and placed into the Q register. The partial product register is zero filled, and the lowest three multiplier bits are decoded as shown below.

Multiplier Bits	Operation
0 0 0	+0X
0 0 1	+1X
0 1 0	+1X
0 1 1	+2X
1 0 0	-2X
1 0 1	-1X
1 1 0	-1X
1 1 1	-0X

During the iterate phase, two final product bits are calculated for each micro-step. Therefore, twelve steps are performed to generate the 24 bit final product.

In the terminate phase, the 24 bits of the fraction are normalized, the addition of the multiplicand and multiplier exponents is performed, and the result (of exponent addition) is corrected for any normalization shifts. In addition, the 24 bit fraction is checked and corrected for the one overflow fault resulting from the multiplication of two -1.0 operands. After all operand corrections have been accomplished, the final exponent and fraction are checked for exponent overflow and fraction equal to zero conditions to set the appropriate indicator. Figure 4.25 illustrates the ML instruction processing.

OPCODE	INSTRUCTION	MNEMONIC
202	Multiply Floating Point	ML

DESCRIPTION

In all active AU, operand (31-0) is multiplied by A-Register (31-0) and the product is placed in the A-Register (23-0). A-Register is corrected for (1) mantissa overflow during adds, (2) result of multiplication equal to zero, (3) exponent underflow, and (4) A not normalized.

If A-Register of product mantissa equal to 0, then the
A-exponent set to -128 .

If A-exponent greater than or equal to +128 set overflow
flip-flop to 1.

Q register contents are destroyed.

Execution: 19 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
        PADMDE ←  0
        PADAIN(10-0) ←    PAMOBR(10-0)
        Transmit PADREQ to EMC
        PAXDIN(31-0) ←    PXEMOT(31-0) when PADSEL=1
    If R=0:
        PAMORL(31-0) ←    PAMOBR(31-0)
        PAXDIN(31-0) ←    PAMORL(31-0)
        PABREG(31-0) ←    PAXDIN(31-0)
```

FIGURE 4.24, ML

```
2)    PAQREG(23-0)←          PABREG(23-0), iff PAEACT=1
      PAQEXT← 0
      PAADMA(23-0)←          0
      PAADMB(23-0)←          0
      Select add (PAADMN)
      PABREG(23-0)←          PAAOUT(23-0)
      PABEXT ← PAAOUT1
3-13) Decode PAQREG1, PAQREG0, PAQEXT into octal 0-7.
      PAADMA(23-0)←    PABREG(23-0)
      If decode = 0,7      :  PAADMB(23-0)←  0
      If decode = 3,4      :  PAADMB(23-0)←  PAAREG(23-0)
      If decode = 1,2,5,6:  PAADMB(23-0)←  PAAREG(23,23-1)
      If decode = 0,1,2,3:  Select Add (PAADMN)
      If decode = 4,5,6,7:  Select subtract (PAADMN)
      PABREG(21-0) ←    PAAOUT(23-2)
      If decode = 1,2:
         PAQREG(22)←  PABEXT + PAAREG0; iff PAEACT=1
      If decode = 5,6:
         PAQREG(22)←  PABEXT - PAAREG0; iff PAEACT=1
      If decode = 0,3,4,7:
         PAQREG(22)←  PABEXT; iff PAEACT=1
      PAQREG(23) ←  PAAOUT0; iff PAEACT=1
      PABEXT←  PAAOUT1
      PAQREG(21-0)←  PAQREG(23-2); iff PAEACT=1
      PAQEXT ← PAQREG1
```

FIGURE 4.24 (CONTINUED)

```
     If PAMOF:  PABREG(23)←  PACO23FF

                PABREG(22)←  PACO23FF

     If PAMOF:  PABREG(23)←  PAAOUT(23)

                PABREG(22)←  PAAOUT(23)

14)  Decode PAQREG(1), PAQREG(0), PAQEXT

     into octal 0-7

     PAADMA(23-0)←  PABREG(23-0)

     If decode = 0,7     :  PAADMB(23-0)←  0

     If decode = 3,4     :  PAADMB(23-0)←  PAAREG(23-0)

     If decode = 1,2,5,6:  PAADMB(23-0)←  PAAREG(23,23-1)

     If decode = 0,1,2,3:  Select Add (PAADMN)

     If decode = 4,5,6,7:  Select Subtract (PAADMN)

     PAAREG(23-0)←  PAAOUT(23-0); iff PAEACT=1

     If decode = 1,2:

        PAQREG(22)←  PABEXT + PAAREGO; iff PAEACT=1

     If decode = 5,6:

        PAQREG(22)←  PABEXT - PAAREGO; iff PAEACT=1

     If decode = 0,3,4,7:

        PAQREG(22)←  PABEXT; iff PAEACT=1

     PAQREG(23)←  PAAOUTO; iff PAEACT=1

     PAQREG(21-0)←  PAQREG(23-2); iff PAEACT=1

     PASHCR(5-0)←+1; iff PAEACT=1

     If PAMOF:  PATOVF←  1

        PACO23FF←  CARRY OUT BIT 23

     Esle PATOVF←  0
```

FIGURE 4.24 (CONTINUED)

```
15) PAADEA(31-24)← PAAREG(31-24)
                                          [ADD EXPONENTS]
    PAADEB(31-24)← PABREG(31-24)

    Select Add (PAADEX)

    PAADMA(23-0)← 0

    PAADMB(23-0)← 0

    Select Add (PAADMN)

    If PAEXOF: PAOVFF← 1

    If PAEXUF  PAAREG(31-24)← -128; iff PAEACT=1

               PAAREG(23-0)← PAAOUT(23-0); iff PAEACT=1

    If $\overline{\text{PAEXUF}}$: PAAREG(31-24)← PAAOUT(31-24); iff PAEACT=1

16) If PATOVF = 1:

        PASCR(5-0)← PASHCR(5-0)

        PAALNI(23-0)← PAAREG(23-0)

        PAAREG(22-0)← PAALNO(22-0); iff PAEACT

        PAAREG(23)← PACO23FF; iff PAEACT

        PAADEA(31-24)← PAAREG(31-24)

        PAADEB(31-24)← 0                   [ADD ONE TO EXPONENT IF

        Select INCREMENT                    MANTISSA OVERFLOWS]

        PAAREG(31-24)← PAAOUT(31-24); iff PAEACT

        If PAEXOF: PAOVFF← 1; iff PAEACT

17) PAALNI(23-)← PAAREG(23-0)

    PASHCR(5-0)← PANRMD(5-0); iff PAEACT=1
```

FIGURE 4.24 (CONTINUED)

```
18) PAALNI(23-0)←  PAAREG(23-0)

    PAADMA(23-0)←  0

    PAADMB(23-0)←  0

    Select add (PAADMN)

    PAADEA(31-24)←  PAAREG(31-24)

    PAADEB(31-24)←  PA2SCO(5,5,5-0)

    Select add (PAADEX)

    If PAEXUF:  PAAREG(31-24)← -128

                PAAREG(23-0) ←  PAAOUT(23-0)

    If PAEXUF:  PAAREG(31-24)← PAAOUT(31-24)
                                                 iff PAEACT=1
                PAAREG(23-0) ←  PAALNO(23-0)

19) PAADMA(23-0) ←  PAAREG(23-0)

    PAADMB(23-0)←  0

    Select Add (PAADMN)

    If PAAOUT(23-0) = 0:

       PAAREG(31-24)←  -128; iff PAEACT=1
```

FIGURE 4.24 (CONTINUED)

MULTIPLY FLOATING POINT (1)

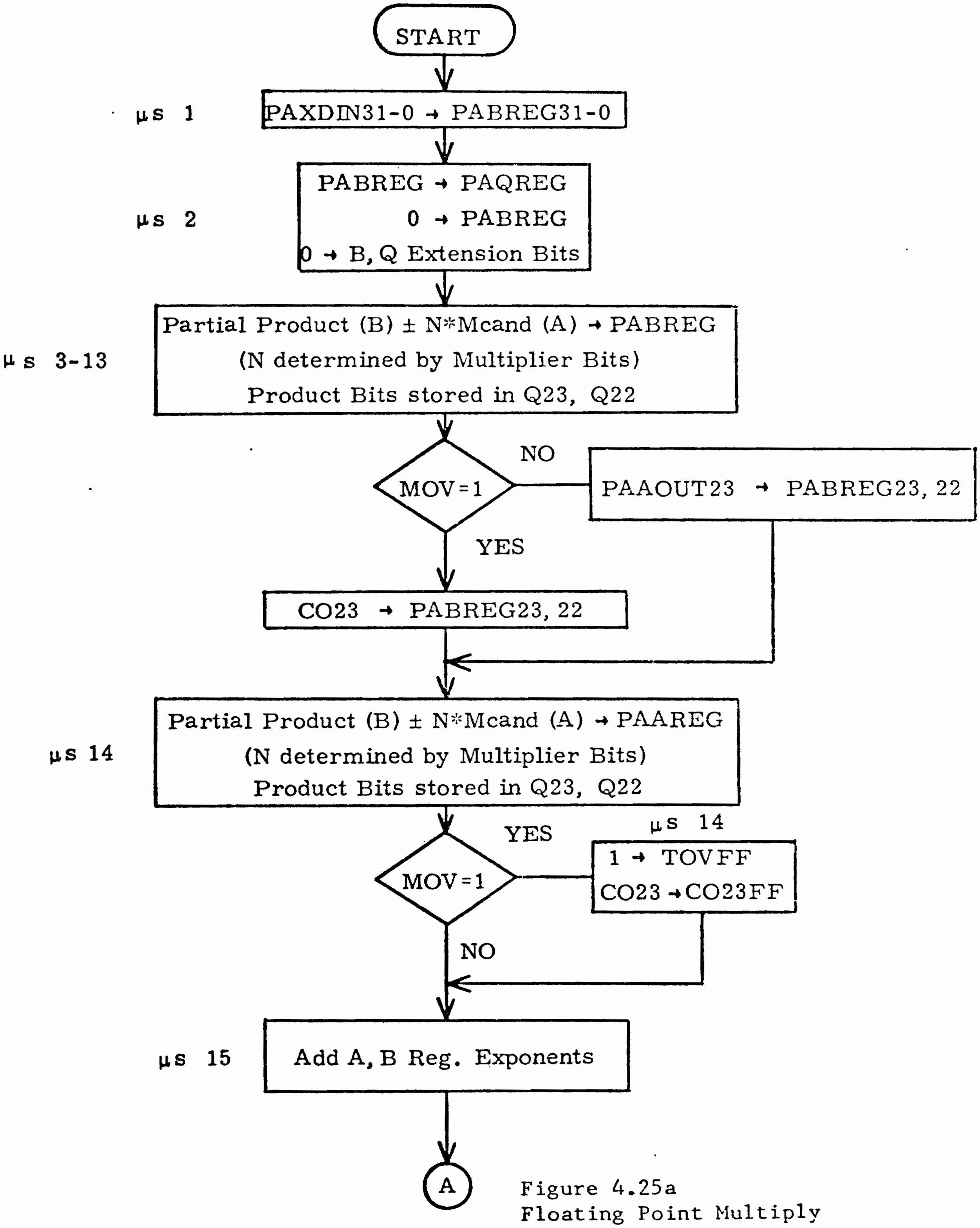


Figure 4.25a
Floating Point Multiply
Algorithm

MULTIPLY FLOATING POINT (2)

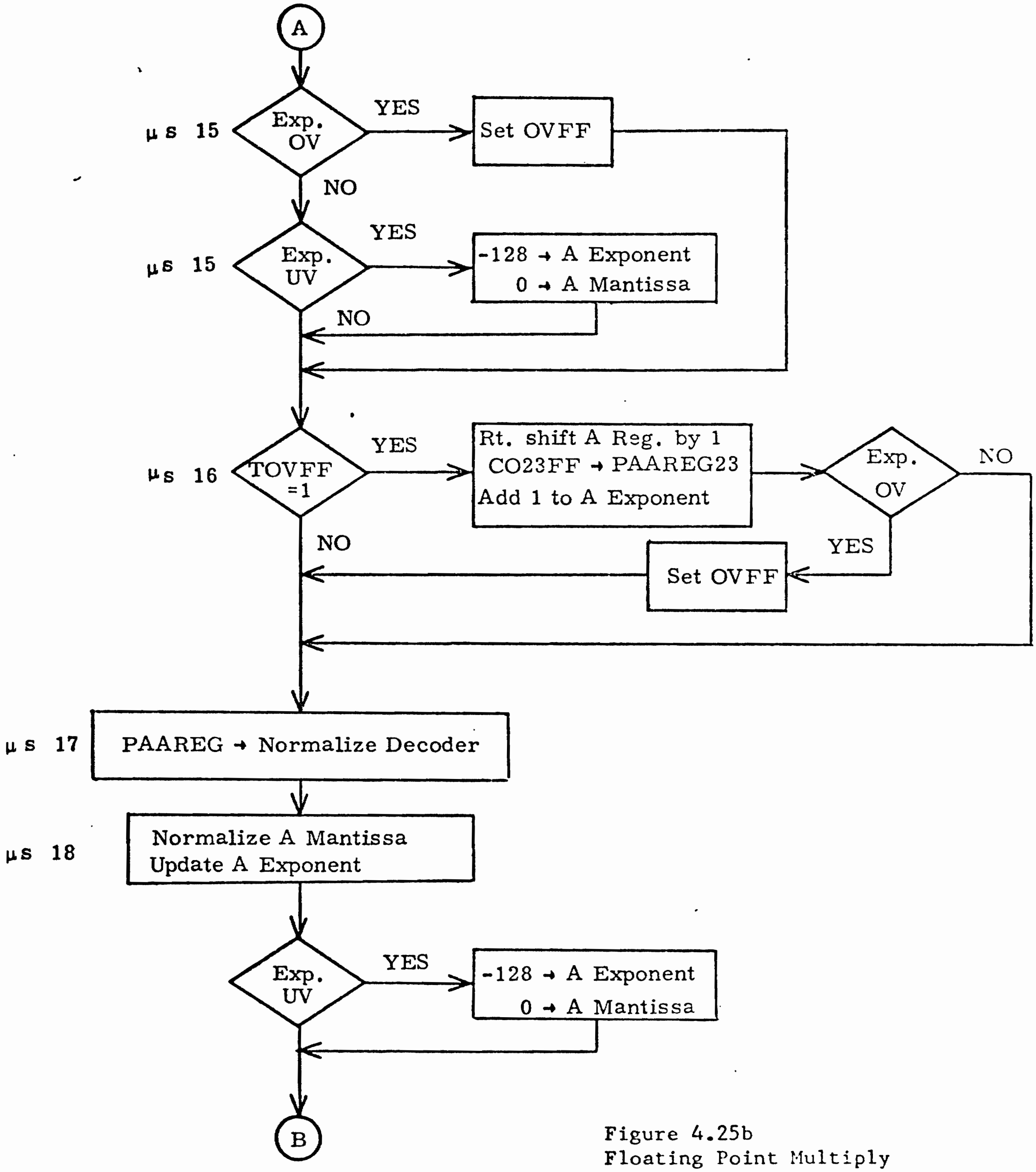


Figure 4.25b
Floating Point Multiply
Algorithm

MULTIPLY FLOATING POINT (3)

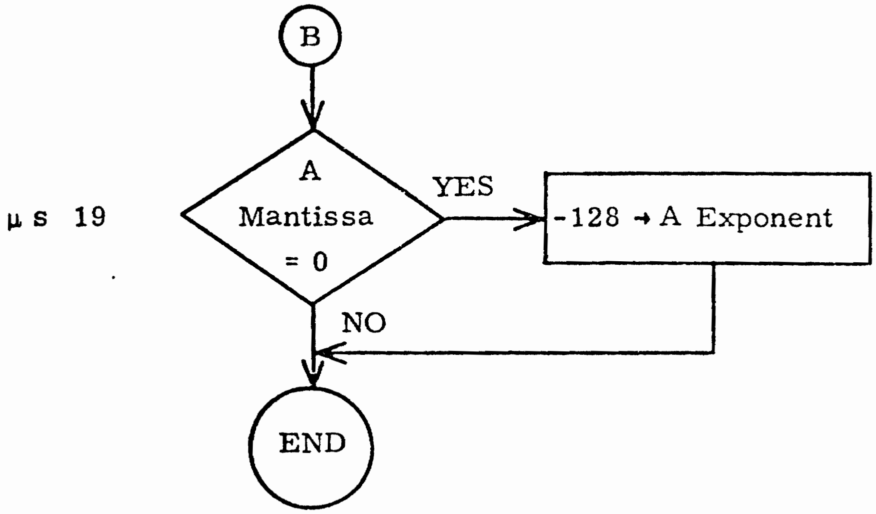


Figure 4.25c
Floating Point Multiply
Algorithm

4.8.4.4 FP Divide

The execution of the floating point divide instruction is comprised of three functional steps; initialize, iterate and terminate. Figure 4.26 shows the micro-step sequence and Figure 4.27 is a flow chart.

The main function performed during the initialize phase is to check the divisor and set the overflow flip-flop if the divisor is zero. When the divisor is not zero, the divisor and dividend fractions are made positive (if negative). It is assumed that the dividend and divisor are normalized initially. In this algorithm, the developed quotient must be less than one. This is accomplished by performing a trial subtraction of the divisor from the dividend. If the result is positive (dividend > divisor), then the dividend is shifted right one bit position and its exponent is corrected for this shift. This procedure thus insures the first quotient bit to be zero and that the quotient will have a value less than one.

In the iterate phase, the divisor is subtracted from the dividend. If the result is positive, a zero bit is entered in the quotient and the dividend is shifted left one place. If the result is negative, a one bit is entered in the quotient and the difference becomes the new dividend.

During the terminate cycle, the quotient and remainder are sign corrected and placed into their respective final registers. The divisor exponent is subtracted from the dividend exponent. The quotient is then normalized and the normalization shift is added to the quotient exponent. The remainder is not normalized. Finally, the exponent is checked for overflow and the quotient for zero to set the appropriate indicator.

OPCODE	INSTRUCTION	MNEMONIC
203	Divide Floating Point	DV

DESCRIPTION

In all active AUs, the floating point A-Register (31-0) is divided by the floating point operand (31-0) with the quotient going to A-Register (31-0) and the remainder to Q-Register (23-0). The divisor and dividend are assumed to be normalized. The quotient shall be normalized and the remainder not.

If the divisor equals zero or if exponent overflow occurs, then the overflow indicator shall be set and the overflow error transmitted to the ICL.

Execution: 38 steps

MICRO STEP SEQUENCE

```
1)  If R=1:
        PADMDE ← 0
        PADAIN(10-0) ← PAMOBR(10-0)
        Transmit PADREQ to EMC
        PAXDIN(31-0) ← PXEMOT(31-0); when PADSEL=1
    If R=0:
        PAMORL(31-0) ← PAMOBR(31-0)
        PAXDIN(31-0) ← PAMORL(31-0)
    PABREG(31-0) ← PAXDIN(31-0)
```

FIGURE 4.26, DV

```
2)  PAADMB(23-0) ←  PABREG(23-0)

    PAADMA(23-0) ←  0

    Select ADD

    PAOVFF ←  1; iff PAAOUT(23-0) = 0 and PAEACT=1

3)  PAADMA(23-0) ←  PAAREG(23-0)

    PAADMB(23-0) ←  PABREG(23-0)

    Select EXCLUSIVE OR

    PASIGN ←  PAAOUT(23)

4)  PAADMA(23-0) ←  0

    PAADMB(23-0) ←  PABREG(23-0)

    Select SUBTRACT

    If  PABREG(23) = 1:

        PABREG(23-0) ←  PAAOUT(23-0)

    If  overflow:

        PATOVF ←  1

        PACO23FF ←  CO23

        PASHCR(5-0) ←  (000001); iff PAEACT=1

5)  If PATOVF = 0:

        Do nothing

    If PATOVF = 1:

        PASCR(5-0) ←  PASHCR(5-0)

        PAALNI(31-0) ←  PABREG(31-0)

        PAADEA(31-24) ←  0

        PAADEB(31-24) ←  PABREG(31-24)

        Select INCREMENT
```

FIGURE 4.26 (CONTINUED)

```
          PABREG(31-24) ← PAAOUT(31-24)
          PAOVFF ← 1 ; iff exponent overflow PAEACT=1
          PABREG(23) ← PACO23FF
          PABREG(22-0) ← PAALNO(22-0)
 6)  PAADMA(23-0) ← 0
     PAADMB(23-0) ← PAAREG(23-0)
     Select SUBTRACT
     If PAAREG(23) = 1:
          PAAREG(23-0) ← PAAOUT(23-0); iff PAEACT=1
     If overflow:
          PATOVF ← 1
          PACO23FF ← CO23
     If not overflow:
          PATOVF ← 0
 7)  If PATOVF = 0:
          Do nothing
     If PATOVF = 1:
          PASCR(5-0) ← PASHCR(5-0)
          PAALNI(31-0) ← PAAREG(31-0)
          PAADEA(31-24) ← PAAREG(31-24)
          PAADEB(31-24) ← 0
          Select INCREMENT
          PAAREG(31-24) ← PAAOUT(31-24)
          PAOVFF ← 1 ; iff overflow • PAEACT=1
          PAAREG(23) ← PACO23FF ; iff PAEACT=1
          PAAREG(22-0) ← PAALNO(23-0) ; iff PAEACT=1
```

FIGURE 4.26 (CONTINUED)

```
 8)   PAADMA(23-0) ←  PAAREG(23-0)
      PAADBM(23-0) ←  PABREG(23-0)
      PAADEA(31-24) ←  PAAREG(31-24)
      PAADEB(31-24) ←  PA2SCO(5,5,5-0)
      Select SUBTRACT
      PASCR(5-0) ←  PASHCR(5-0)
      PAALNI(23-0) ←  PAAREG(23-0)
      If PAAOUT(23) = 0: (A ≥ B)
         PAAREG(23-0) ←  PAALNO(23-0)
         PAAREG(31-24) ←  PAAOUT(31-24)        ;iff PAEACT=1
         PAOVFF ← 1 ; iff overflow (PAADEX)
      If PAAOUT(23) = 1 : (A < B)
         Do nothing
9-30) PAADMA(23-0) ←  PAAREG(23-0)
      PAADMB(23-0) ←  PABREG(23-0)
      Select SUBTRACT
      PASCR(5-0) ←  PA2SCO(5-0)
      PAALNI(23-0) ←  PAAREG(23-0)
      PAQREG(23-1) ←  PAQREG(22-0); iff PAEACT=1
      If PAAOUT(23) = 1:
         PAAREG(23-0) ←  PAALNO(23-0)
         PAQREG(0) ←  0                         ;iff PAEACT=1
      If PAAOUT(23) = 0:
         PAAREG(23-1) ←  PAAOUT(22-0)
         PAAREG(0) ←  0
         PAQREG(0) ←  1
```

FIGURE 4.26 (CONTINUED)

```
31)  PAADMA(23-0) ←  PAAREG(23-0)

     PAADMB(23-0) ←  PABREG(23-0)

     PAADEA(31-24) ← 0

     PAADEB(31-24) ← 0

     Select SUBTRACT

     PASCR(5-0) ←  PA2SCO(5-0)

     PAALNI(23-0) ←  PAAREG(23-0)

     PAQREG(23-1) ←  PAQREG(22-0); iff PAEACT=1

     If PAAOUT(23) = 1:

        PAAREG(23-0) ←  PAALNO(23-0)

        PAQREG(0) ←  0

     If PAAOUT(23) = 0:                          if PAEACT=1

        PAAREG(23-1) ←  PAAOUT(22-0)

        PAAREG(0) ←  0

        PAQREG(0) ←  1

     PASHCR(5-0) ←  PAAOUT(29-24)

32)  PAADMA(23-0) ←  PAAREG(23-0)

     PAADMB(23-0) ←  PABREG(23-0)

     Select SUBTRACT

     PASCR(5-0) ←  PA2SCO(5-0)

     PAALNI(23-0) ←  PAAREG(23-0)

     PAQREG(23-1) ←  PAQREG(22-0) ; iff PAEACT=1

     If PAAOUT(23) = 1:

        PAAREG(23-0) ←  PAALNO(23-0)

        PAQREG(0) ←  0                       iff PAEACT=1

     If PAAOUT(23) = 0:

        PAAREG(23-0) ←  PAAOUT(23-0)

        PAQREG(0) ← 1
```

FIGURE 4.26 (CONTINUED)

```
33)  PAADMA(23-0) ←  $\overline{PAQREG(23-0)}$
     PAADMB(23-0) ←  0
     Select INCREMENT
     If PASIGN = 0:
        Do nothing
     If PASIGN = 1:
        PAQREG(23-0) ←  PAAOUT(23-0) ; iff PAEACT=1
34)  PAALNI(23-0) ←  PAQREG(23-0)
     PASHCR(5-0) ←  PANRMD(5-0)
     PAADMA(23-0) ←  0
     PAADMB(23-0) ←  PAAREG(23-0)
     Select SUBTRACT
     Iff PASIGN = 1:
        PAAREG(23-0) ←  PAAOUT(23-0) ; iff PAEACT=1
35)  PASCR(5-0) ←  PA2SCO(5-0)
     PAALNI(23-0) ←  PAQREG(23-0)
     PAADMA(23-0) ←  PAAREG(23-0)
     PAADMB(23-0) ←  0
     Select ADD
     PAAREG(23-0) ←  PAALNO(23-0)
                                          iff PAEACT=1
     PAQREG(23-0) ←  PAAOUT(23-0)
```

FIGURE 4.26 (CONTINUED)

```
36) PAADEA(31-24) ← PAAREG(31-24)
    PAADEB(31-24) ← PABREG(31-24)
    PAADMA(23-0) ←  0
    PAADMB(23-0) ←  0
    Select SUBTRACT
    If not underflow:
       PAAREG(31-24) ←  PAAOUT(31-24) ; iff PAEACT=1
    If  overflow (exponent):
       PAOVFF ← 1 ; iff PAEACT=1
    If  underflow (exponent):
       PAAREG(31-24) ←  $-128_{10}$
                                             ;iff PAEACT=1
       PAAREG(23-0) ←  PAAOUT(23-0)=0
37) PAADEA(31-24) ← PAAREG(31-24)
    PAADEB(31-24) ← PA2SCO(5,5,5-0)
    PAADMA(23-0) ←  0
    PAADMB(23-0) ←  0
    Select ADD
    If no underflow (exponent):
       PAAREG(31-24) ← PAAOUT(31-24)
                                                ; iff PAEACT=1
    If underflow: PAAREG(31-24) ← $-128_{10}$
       PAAREG(23-0) ← PAAOUT(23-0)
38) PAADMA(23-0) ← PAAREG(23-0)
    PAADMB(23-0) ← 0
    Select ADD
    If  PAAOUT(23-0) = 0:
       PAAREG(31-24) ← $-128_{10}$ ; iff PAEACT=1
```

FIGURE 4.26 (CONTINUED)

DIVIDE FLOATING POINT (1)

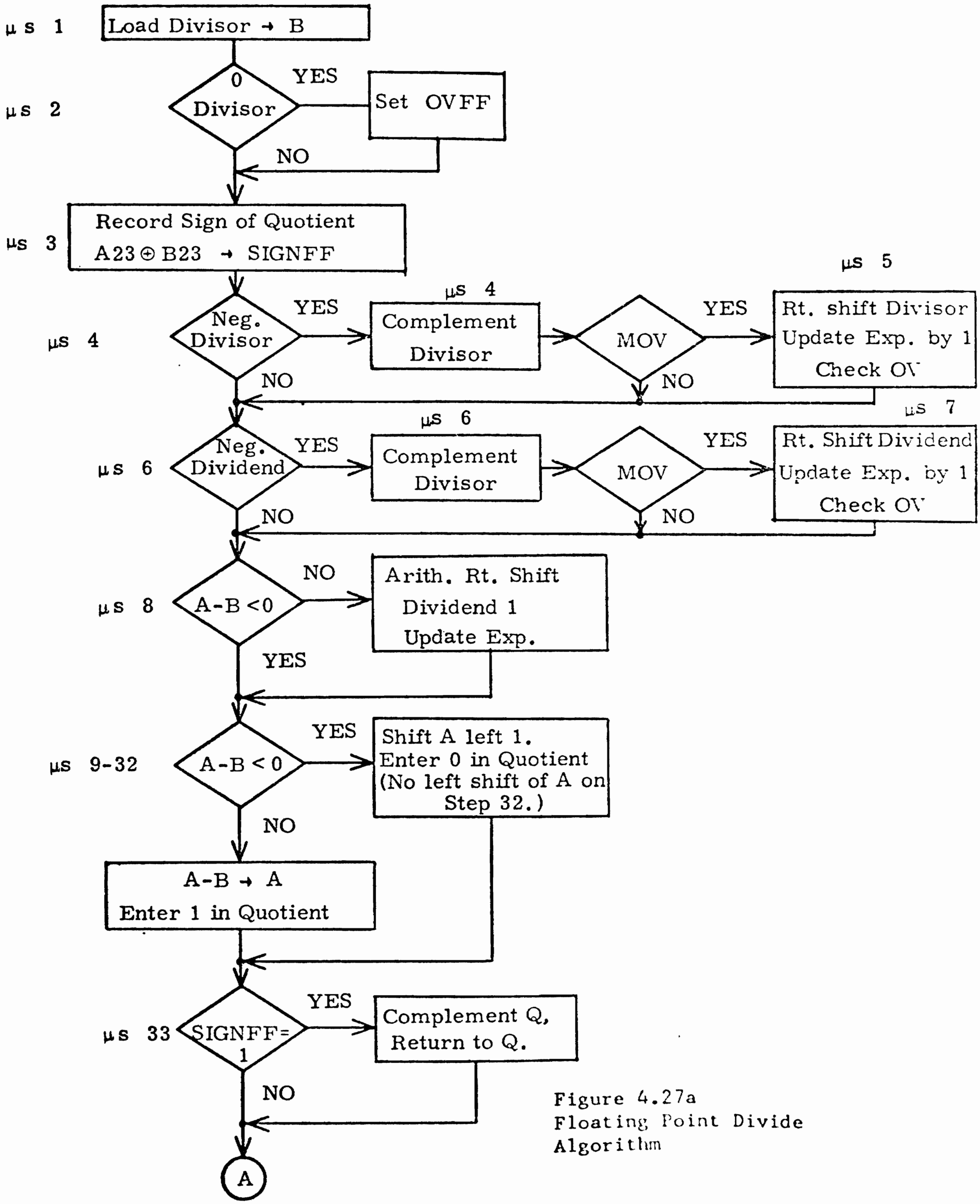


Figure 4.27a
Floating Point Divide
Algorithm

DIVIDE FLOATING POINT (2)

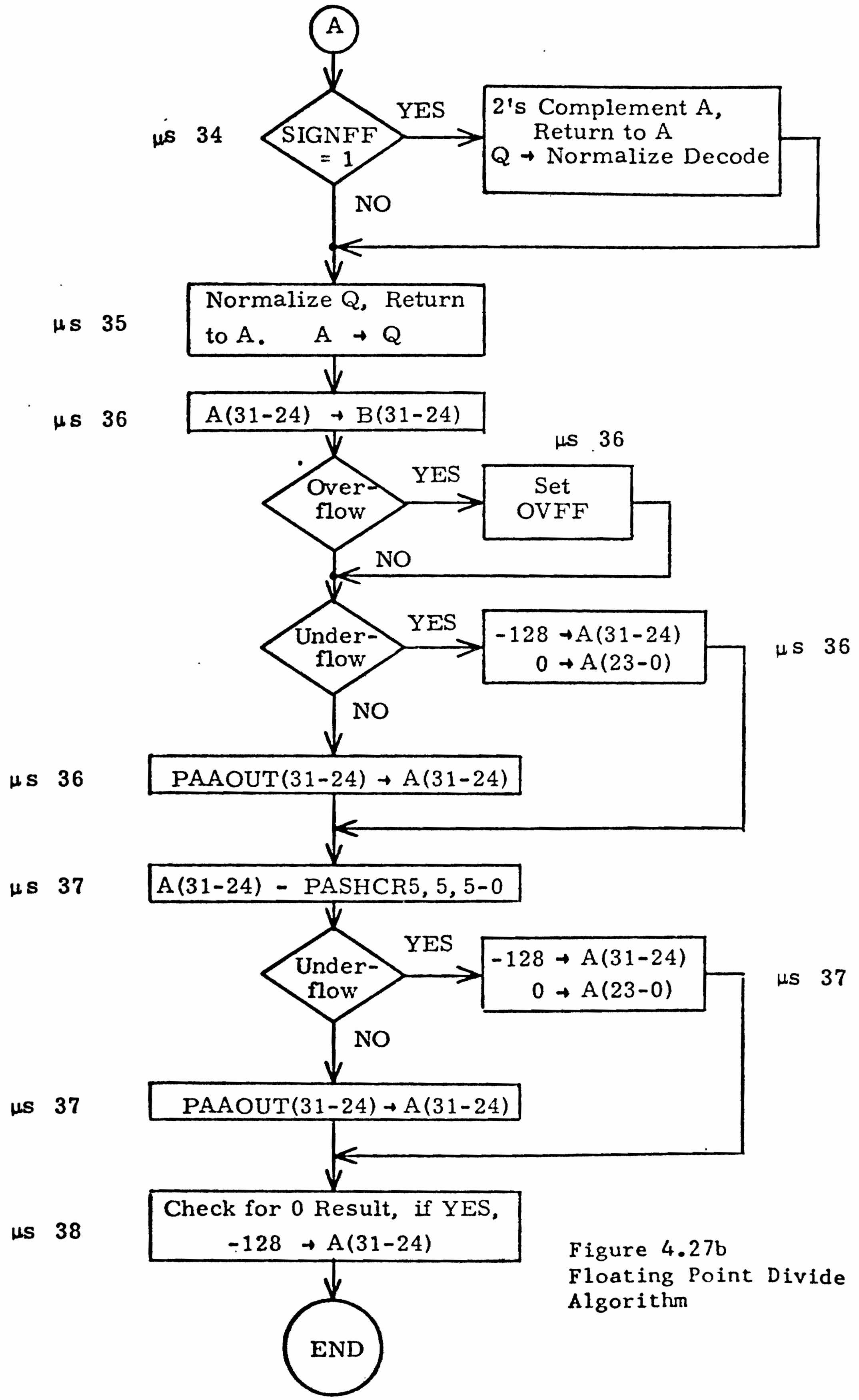


Figure 4.27b
Floating Point Divide
Algorithm

4.8.4.5 F.P. Square Root

The execution of the floating point square root instruction is comprised of two functional steps; initialize and iterate. Figure 4.28 shows the micro-step sequence for SQ and Figure 4.29 is the flow chart.

The main function of the initialize phase is to determine if the fraction is negative. If so, the overflow flip-flop is set. If not, the exponent portion of the operand is examined to determine if it is even or odd. If it is odd, the fraction is shifted right one position. Then, the exponent is shifted right one place to divide it by two. The first trial divisor is then taken.

The iterative phase is composed of pairs of micro-steps wherein one square root bit is generated per pair. Within each pair, a trial divisor is generated, a subtraction of this divisor from the dividend (or previous remainder) is performed, a square root bit is generated based on the sign of the result of the subtraction, and a new trial divisor is formed.

The result of the iterative phase is a square root of 12 bits.

4.8.5 Output Instructions

Instructions chosen as examples are STA and OTA.

4.8.5.1 STA

During the STA instruction, the A Register is gated unchanged through the ALU to EM. The micro-step sequence is shown in Figure 4.30. The actual EM write cycle is handled by the PICU and EMC as described in Section 4.2.2.1.2.

4.8.5.2 OTA

During the OTA instruction, the A Register is gated unchanged through the ALU onto the Output Data Bus (PAODAT). The micro-step sequence is shown in Figure 4.31. The AU should have been selected such that only one AU in the ensemble has its EA flip-flop set.

OPCODE	INSTRUCTION	MNEMONIC
204	Square Root Floating Point	SQ

DESCRIPTION

In all active AUs, the square root of the floating point contents of A-Register (31-0) is placed in A-Register (31-0). Overflow error is generated and PAOVFF set in the case of negative operands. Q-Register contents are destroyed.

Execution: 32 steps

MICRO STEP SEQUENCE

```
1)  PAADEA ← 0
    PAADEB ← 0
    PAADMA ← 0
    PAADMB ← 0
    Select INCREMENT
    PABREG(23-0) ← PAAOUT(23-0) = (0...01)
    PASHCR(5-0) ← PAAOUT(29-24) = +1 iff PAEACT=1
    PAOVFF — 1; iff PAAREG(23) = 1 • PAEACT=1
2)  PAALNI(23-0) ← PAAREG(23-0)
    If PAAREG(24) = 1:
       PAADEA(31-24) ← PAAREG(31-24)
       PAADEB(31-24) ← 0
       Select INCREMENT
       PAOVFF ← 1 iff overflow
       PAAREG(31-24) ← PAAOUT(31-24)      ; iff PAEACT=1
       PAAREG(23-0) ← PAALNO(23-0)
```

FIGURE 4.28, SQ

```
          3)  PAAREG(31-24)←  PAAREG(31,31-25)
              PAADMA(23-0)←  PAAREG(23-0)
              PAADMB(23-0)←  0
              Select ADD
              PAQREG(23-0)← PAAOUT(23-0) ; iff PAEACT=1
          4)  PAADMA(23-0)← 0
              PAADMB(23-0)← 0
              Select ADD
              PAAREG(23-0)← PAAOUT(23-0) ; iff PAEACT=1
   5,7,9-27)  PAQREG(23-0)← (PAQREG(22-0, 0) ; iff PAEACT=1
              PAADMA(23-0)← (PAAREG(21-0), PAQREG(22,21)
              PAADMB(23-0)← PABREG(23-0)
              If PAAREG(23) = 0:
                 Select SUBTRACT
                 PAAREG(23-0)← PAAOUT(23-0) ; iff PAEACT=1
              If PAAREG(23) = 1:
                 Select ADD
                 PAAREG(23-0)← PAAOUT(23-0) ; iff PAEACT=1
 6,8,10,-28)  PAQREG(23-0)← (PAQREG(22-0),0) ; iff PAEACT=1
              PASCR(5-0)← PA2SCO(5-0)
              PAALNI(23-0)← PABREG(23-0)
              PABREG(23-0)← PAALNO(23-0)
              If PAAREG(23) = 0 (correct guess)
                 PABREG(2-0)← (101) ; iff PAEACT=1
              If PAAREG(23) = 1 (wrong guess)
                 PABREG(2-0)← (011) ; iff PAEACT=1
```

FIGURE 4.28 (CONTINUED)

SQUARE ROOT (1)

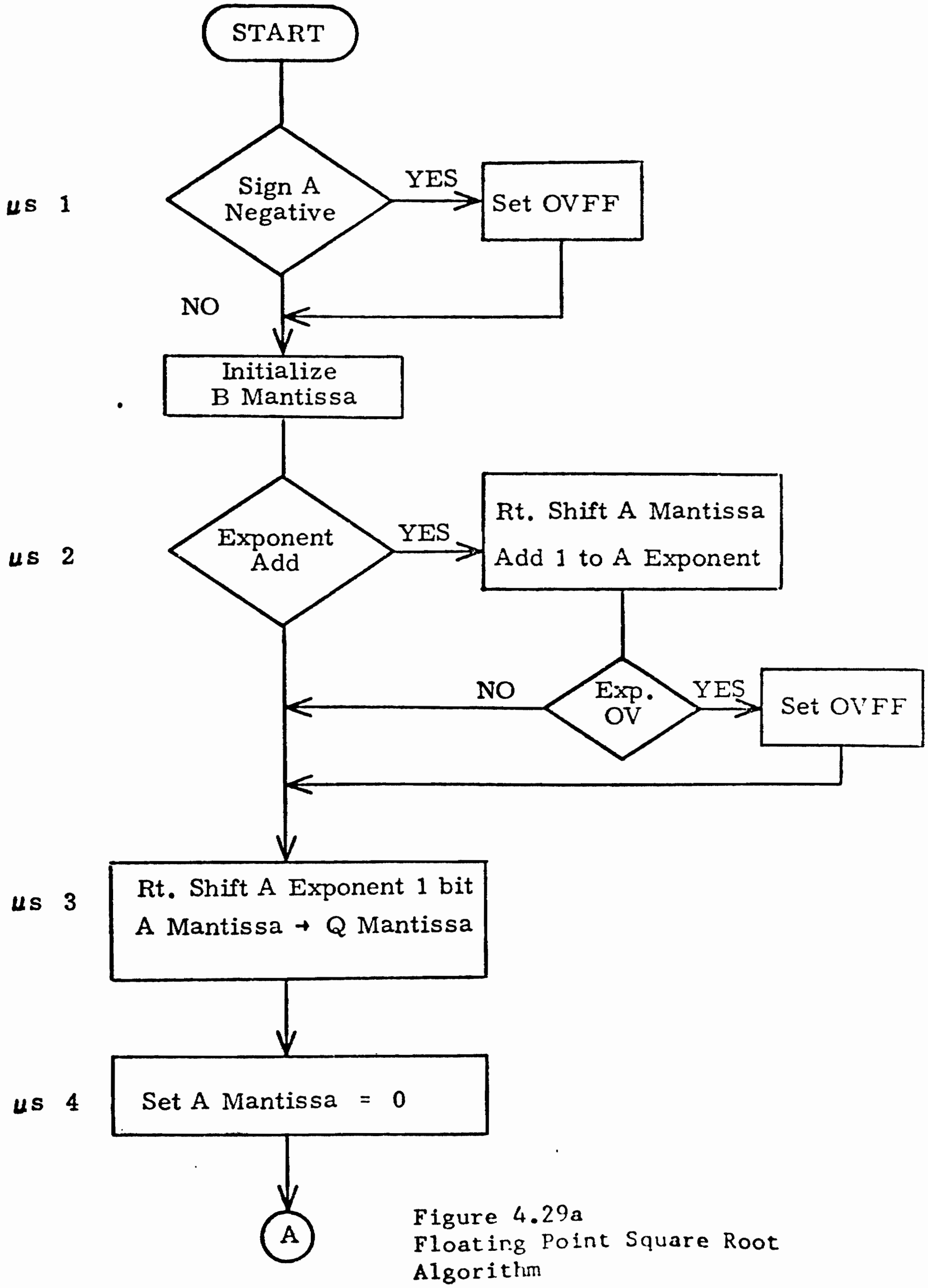


Figure 4.29a
Floating Point Square Root
Algorithm

SQUARE ROOT (2)

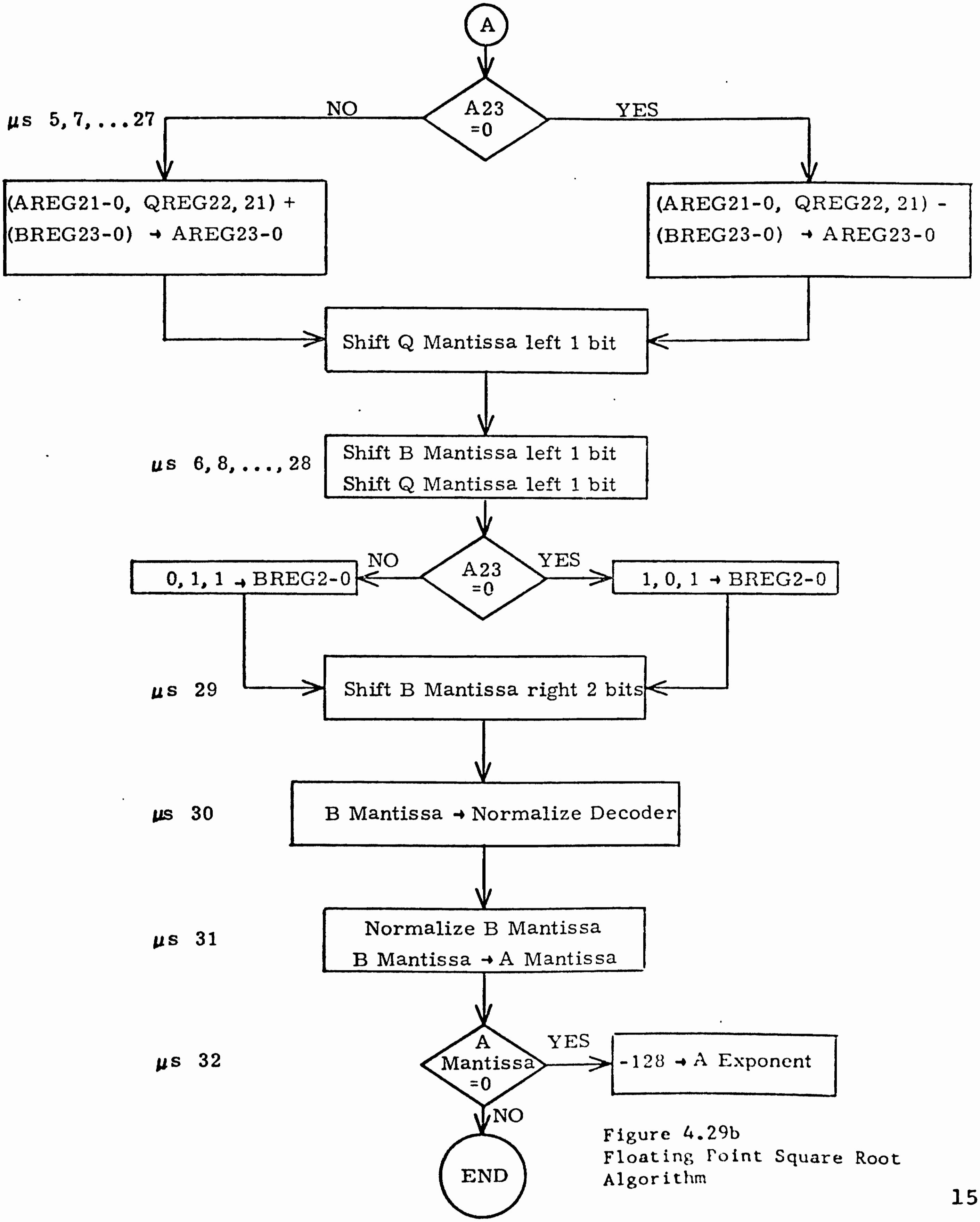

A
A23 =0
NO
YES
μs 5, 7, ...27
(AREG21-0, QREG22, 21) + (BREG23-0) → AREG23-0
(AREG21-0, QREG22, 21) - (BREG23-0) → AREG23-0
Shift Q Mantissa left 1 bit
μs 6, 8, ..., 28
Shift B Mantissa left 1 bit
Shift Q Mantissa left 1 bit
A23 =0
NO
YES
0, 1, 1 → BREG2-0
1, 0, 1 → BREG2-0
μs 29
Shift B Mantissa right 2 bits
μs 30
B Mantissa → Normalize Decoder
μs 31
Normalize B Mantissa
B Mantissa → A Mantissa
μs 32
A Mantissa =0
YES
-128 → A Exponent
NO
END

Figure 4.29b
Floating Point Square Root Algorithm

OPCODE	INSTRUCTION	MNEMONIC
031	Store A-Register	STA

DESCRIPTION

AU A-Register (31-0) is stored in element memory (word specified by operand (bits 10-0)) from those AUs which are active.

Execution: 1 step

MICRO STEP SEQUENCE

```
1)  PADMDE ← 1
    PADAIN(10-0) ← PAMOBR(10-0)
    Transmit PADREQ to EMC
    PAADEA(31-24) ← PAAREG(31-24)
    PAADEB(31-24) ← 0
    PAADMA(23-0) ← PAAREG(23-0)
    PAADMB(23-0) ← 0
    Select OR
    PXEMOT(31-0) ← PAAOUT(31-0); iff PADSEL· (PAEACT=1)
    PADWSL ← 1; iff PADSEL·(PAEACT=1)
```

FIGURE 4.30, STA

OPCODE	INSTRUCTION	MNEMONIC
047	Output to A	OTA

DESCRIPTION

AU A Register (31-0) is gated to the SCL A Register via the Adder Output to the Output Data Bus if Element Activity is 1.

Execution: 1 step

MICRO STEP SEQUENCE

```
1)  PAADEA(31-24)←  PAAREG(31-24)
    PAADEB(31-24) ←  0
    PAADMA(23-0) ←   PAAREG(23-0)
    PAADMB(23-0) ←   0
    Select OR
    PAODAT(31-0)← PAAOUT(31-0); iff PAEACT=1
```

FIGURE 4.31, OTA

4.8.6 Distributed Logic Instructions

Instructions chosen as examples are RDA (executed in the SCL), SH, SL and SF.

4.8.6.1 Count AU/RDA

AU Count logic is based on a design by Drs. C.C. Foster (FO76) and K. E. Batcher at Goodyear Aerospace. It is an exact counter which requires no more than one full adder per PE.

Each row of nine AU contains one set of the logic shown in Figure 4.32. The AU supplies the output of its EA FF. The sum is generated and sent to the CC where standard adders generate a count for the entire system.

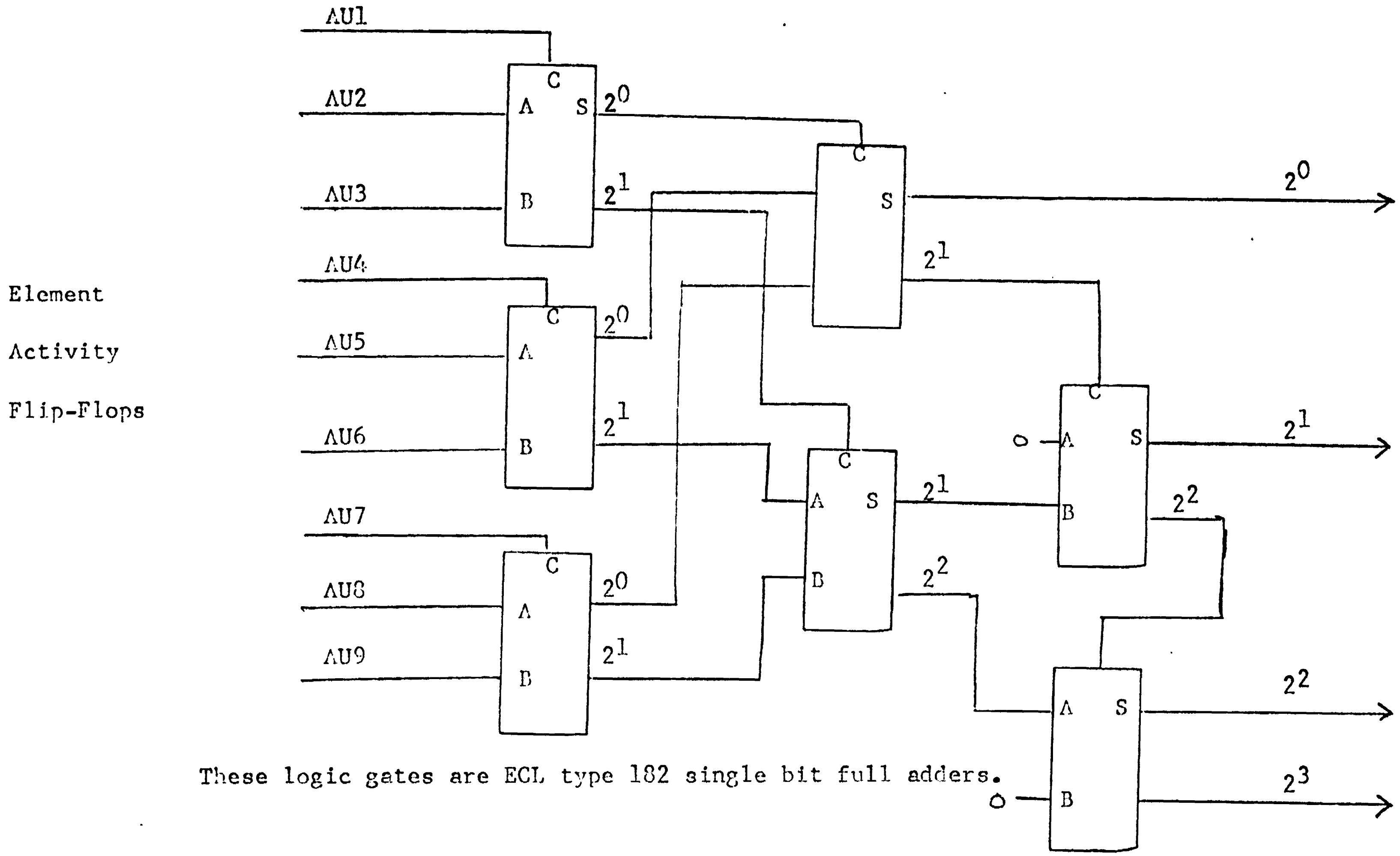
The nine EA signals are fed to a group of full adders. The outputs are labeled with a power of two weighting. The remaining adder stages gather all like weighted signals together to produce a single four bit count.

In the CC, the AU count is available to the SCL, when, during the execution of a RDA, the count of active AU is loaded into the SCL A Register.

4.8.6.2 Select Highest/Lowest

The SH/SL algorithm was described in detail in Section 3.6.2. Figures 4.33 shows the micro-step sequence for SH. In the following micro-step description

PAALFA -	Flip-Flop
PABETA -	Flip-Flop
SAMANY -	When low, indicates to the PICU that zero or one AU remains active.
PABITA	A, B, C lines
PABITB	
PABITC	
PABITX	X, Y, Z lines
PABITY	
PABITZ	



These logic gates are ECL type 182 single bit full adders.

Figure 4.32 Row Count Logic

OPCODE	INSTRUCTION	MNEMONIC
164	Select Highest	SH

DESCRIPTION

Set EA to zero in all active AU's whose A-Registers contain non-maximal values relative to set of active elements (Integer or Floating Point).

Execution: 35 Steps (Maximum)

MICRO STEP SEQUENCE

```
1)  PAADMA(23-0) ← 0
    PAADMB(23-0) ← PAAREG(23-0)
    SELECT OR
    PABREG(23-0) ← PAAOUT(23-0)
    If PAAREG(23) = 1
       PABREG(31) ← PAAREG(31)
       PABREG(30-24) ← $\overline{\text{PAAREG(30-24)}}$
    If PAAREG(23) = 0
       PABREG(31) ← $\overline{\text{PAAREG(31)}}$
       PABREG(30-24) ← PAAREG(30-24)
    PAALFA ← $\overline{\text{PAAOUT(23)}}$
    If PAAREG(23) = 1
       PABETA ← PAAREG(31)
    If PAAREG(23) = 0
       PABETA ← $\overline{\text{PAAREG(31)}}$
    PASHCR(5-0) ← $01_8$; Iff PAEACT = 1
```

FIGURE 4.33, SH

```
         2)  PABITA ←  PAAFLA·PABETA·PAEACT

             PABITB ←  PAALFA·$\overline{PABETA}$·PAEACT

             PABITC ←  $\overline{PAALFA}$·PABETA·PAEACT

         3)  PABITA ←  PAALFA·PABETA·PAEACT

             PABITB ←  PAALFA·$\overline{PABETA}$·PAEACT

             PABITC ←  $\overline{PAALFA}$·PABETA·PAEACT

             PASCR(5-0) ←  PA2SCO(5-0)

             PAALNI(31-0) ←  PABREG(31-0)

             PABREG(31-0) ←  PAALNO(31-0)

             If ($\overline{PABITA}$·PABITX) +

                ($\overline{PABITA}$·$\overline{PABITB}$·PABITY) +

                ($\overline{PABITA}$·$\overline{PABITB}$·$\overline{PABITC}$·PABITZ)

                Then PAEACT ← 0

             Else PAEACT not changed.

             PAALFA ←  PABREG(31)

             PABETA ←  PABREG(30)

even, 4-34)  If SAMANY = 0:

                EXIT

             If SAMANY = 1:

                PABITA ←  PAALFA·FABETA·PAEACT

                PABITB ←  PAALFA·$\overline{PABETA}$·PAEACT

                PABITC ←  $\overline{PAALFA}$·PABETA·PAEACT

                PASCR(5-0) ←  PA2SCO(5-0)

                PAALNI(31-0) ←  PABREG(31-0)

                PABREG(31-0) ←  PAALNO(31-0)
```

FIGURE 4.33 (CONTINUED)

```
odd, 5-35)  If SAMANY = 0:
               EXIT
            If SAMANY = 1;
               PABITA ← PAALFA·PABETA·PAEACT
               PABITB ← PAALFA·$\overline{\text{PABETA}}$·PAEACT
               PABITC ← $\overline{\text{PAALFA}}$·PABETA·PAEACT
               PASCR(5-0) ← PA2SCO(5-0)
               PAALNI(31-0) ← PABREG(31-0)
               PABREG(31-0) ← PAALNO(31-0)
               IF ($\overline{\text{PABITA}}$·PABITX) +
                  ($\overline{\text{PABITA}}$·$\overline{\text{PABITB}}$·PABITY) +
                  ($\overline{\text{PABITA}}$·$\overline{\text{PABITB}}$·$\overline{\text{PABITC}}$·PABITZ)
                  Then PAEACT ← 0
               Else PAEACT not changed
               PAALFA ← PABREG(31)
               PABETA ← PABREG(30)
```

FIGURE 4.33 (CONTINUED)

The conversion micro-step uses both normal ALU hardware and special logic. The fraction part of the conversion is performed by the mantissa adder. The exponent part, converted by the Search Conversion Logic, PASRCH(31-24), is as follows:

```
(IF  $(\text{PAMORL}(28\text{-}24) = 27 \cdot \text{PAAREG23})$ OR
     $(\text{PAMORL}(28\text{-}24) = 24 \cdot \overline{\text{PAAREG23}})$ THEN
         $\text{PASRCH31} \leftarrow \text{PAAREG31},$
         $\text{PASRCH}(30\text{-}24) \leftarrow \overline{\text{PAAREG}(30\text{-}24)})$ OR
(IF  $(\text{PAMORL}(28\text{-}24) = 27 \cdot \overline{\text{PAAREG23}})$ OR
     $(\text{PAMORL}(28\text{-}24) = 24 \cdot \text{PAAREG23})$ THEN
         $\text{PASRCH31} \leftarrow \overline{\text{PAAREG31}},$
         $\text{PASRCH}(30\text{-}24) \leftarrow \text{PAAREG}(30\text{-}24)$ )
```

The conversion logic is shown in block diagram form in Figure 4.34.

The two bit interative logic is shown in Figure 4.35. The A, B, C output lines are sent to the BSD where they are "OR"ed with all other like lines with the Bay. Then they are sent to the CSD where all eight Bays are "OR"ed. The result, now called X, Y, Z lines are then broadcast back to all AU. The global X, Y, Z lines are compared to the local A, B, C lines and the EA FF reset if necessary.

4.8.6.3 Select First

The SF instruction encompasses very little logic in the AU. The AU sends its EA FF to the BSD and receives a pointer, PASFPT, back. The BSD sends a row activity signal to the CSD. The row activity is an "OR" of the EA of each AU in the row. The CSD sends a pointer to the first BSD in the first Bay with an active AU. The BSD then sends the pointer to the first active AU in the row.

The AU, upon execution of an SF instruction, waits one clock for the propagation delay of the distributed logic to settle and then the AU loads the pointer into its EA Flip-flop. Figure 4.36 shows these two micro-steps.

The SF logic on the BSD is shown in Figure 4.37. The nine EA lines enter at the left and the nine pointers exit on the right.

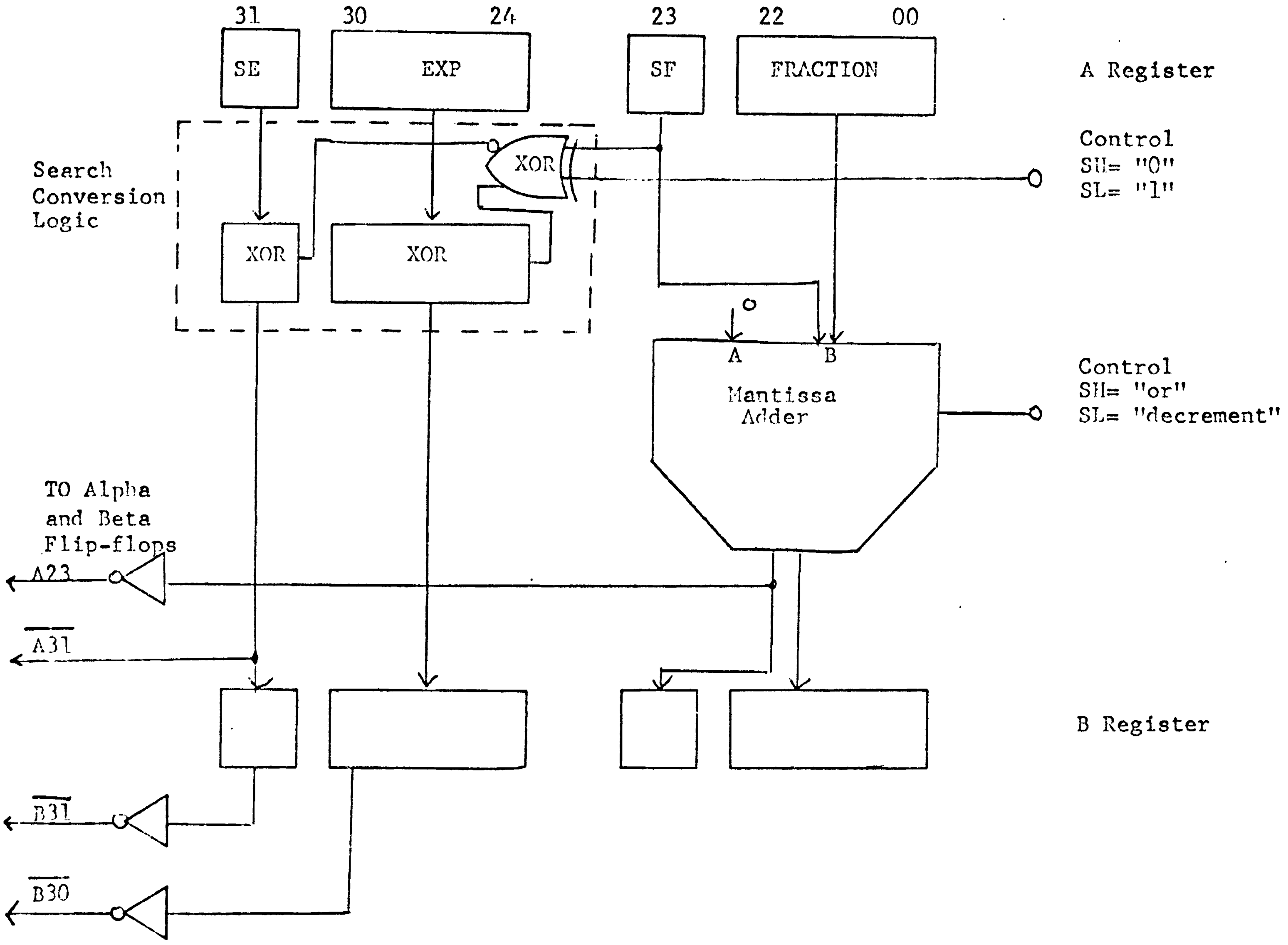


Figure 4.34 SH/SL Conversion Micro-Step Logic

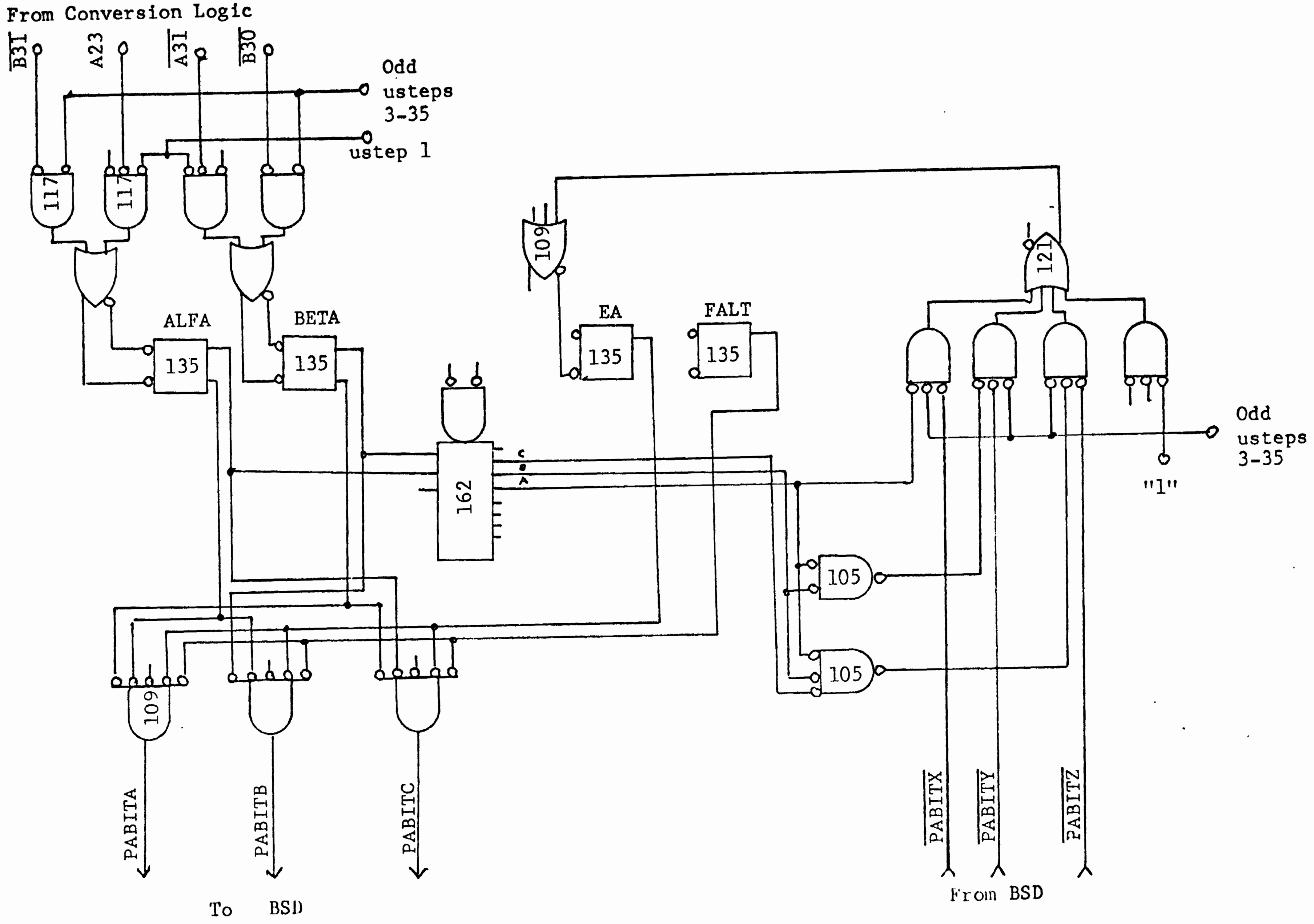


Figure 4.35 AU Search Logic

OPCODE	INSTRUCTION	MNEMONIC
150	Select First	SF

DESCRIPTION

Set EA to zero in all active AUs except one. The one remaining active AU is chosen on the basis of physical hardware location.

Execution: 2 steps

MICRO STEP SEQUENCE

```
1)  NOP
2)  PAEACT ← 0; if PASFPT = 0
        Else PAEACT not changed
```

FIGURE 4.36, SF

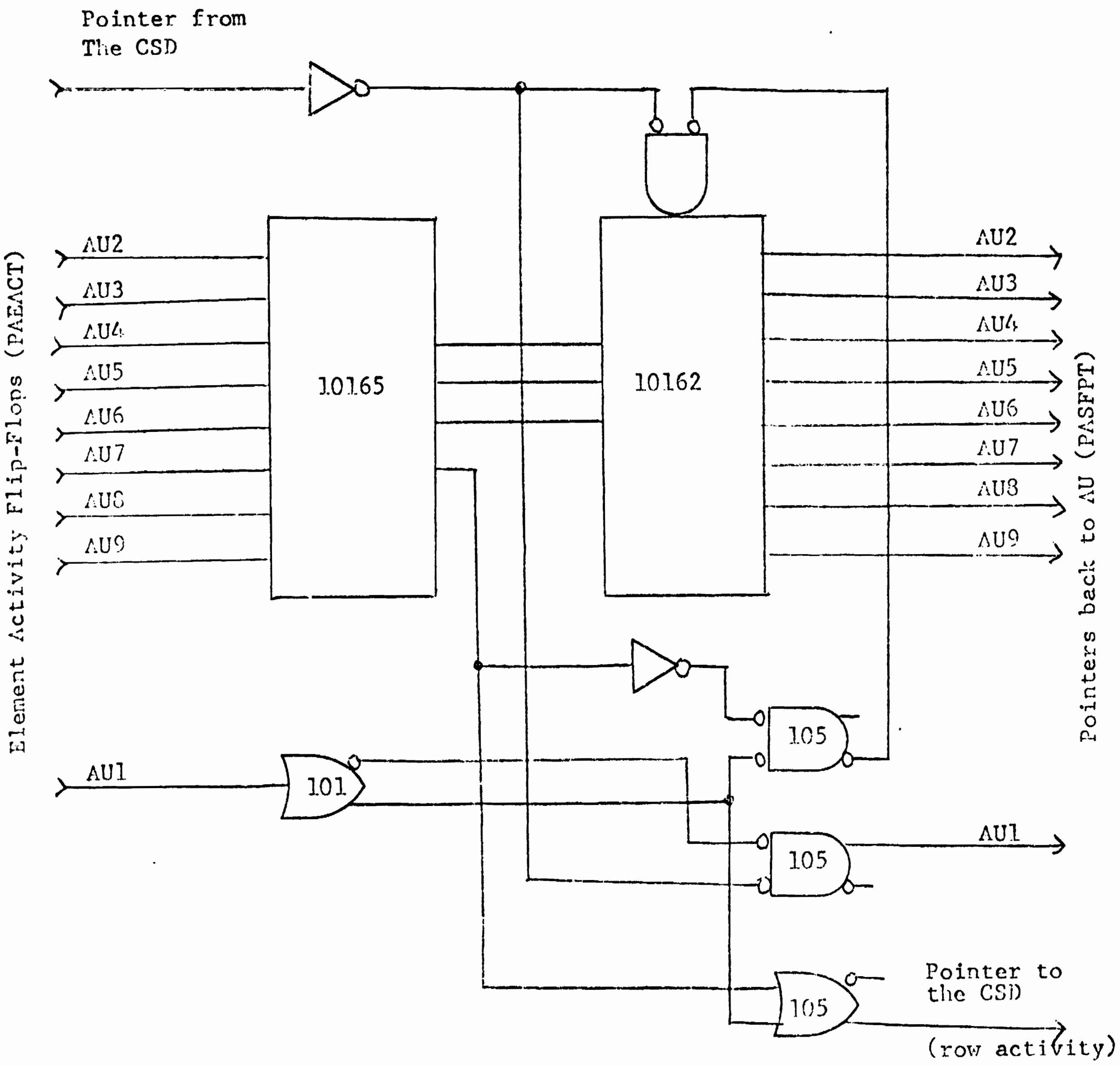


Figure 4.37 Row Select First Logic

5.0 Conclusion

The AU design successfully met all design and operational goals set by the specification document. It and the remainder of PEPE was delivered on budget and on schedule to the BMD Advanced Technology Center in Huntsville, Alabama.

A critique of the final design found two areas of the AU that could be improved. The FIX instruction algorithm could be reduced from 5 steps to 3 steps with the addition of hardware. The rather clumsy algorithm now used, were developed because an error was found in the AU during final tests. The correction had to entail minimum change and therefore was not an elegant solution. Hardware to detect the case where significant bits are shifted out would cut steps 4 and 5 from the current algorithm.

The quotient of the floating point multiply and the dividend of the floating point divide should be 48 bits long to avoid any loss of accuracy. AU hardware does not support the long words but this could possibly be added to enhance the AU performance.

PEPE has been studied by Burroughs and System Development Corporation for use in other high speed processing problems. Areas suggested have been air traffic control, image processing, weather forecasting and wind tunnel simulation. Since PEPE was designed as a processor ensemble, there is no direct communication between AU's. Most problems that are amenable to parallel solutions require at least nearest neighbor communications. The possibility of using the AOCU/AOU parallel ensemble as a programmable routing network has been proposed. Under this scheme, the ACU/AU ensemble could be processing data while the AOCU/AOU ensemble was routing data between the

Element Memories. The major advantage of this software routing is that in a hardwired system, like ILLIAC IV, if a PE failed, the communication network was left with a hole whereas if a PEPE PE failed, software could reestablish the network using another PE. This work has not yet been completed.

The analysis of algorithms for parallelism is a new, largely unexplored field. Some work [Ra76] and [KU76] has been done but a useable parallelism detecting compiler is still in the future.

References and Bibliography

Search Algorithm

[Le 62] Lewin, M. H., "Retrieval of Ordered Lists from a Content Addressed Memory", RCA Review, June 1962.

[Se 77a] "Select High/Low Register Method and Apparatus", US Patent #4,007,439.

[Se 77b] "Integer and Floating Point to Binary Converter", US Patent #4,038,538.

Select First Algorithm

[An 74] Anderson, George A., "Multiple Match Resolvers: A New Design Method", IEEE Transactions on Computers, C-23, No. 12, Dec. 1974.

PEPE Hardware

[Fo 76a] Foster, C. C., Computer Architecture, Van Nostrand Reinhold, 1976.

[Fo 76b] Foster, C. C., Content Addressable Parallel Processors, Van Nostrand Reinhold, 1976.

[Sd 74] PEPE System Functional Design Specification, Volume II Hardware Specification, System Development Corporation, REV. E, Dec. 1975.

[Th 76] Thurber, K. J., Large Scale Computer Architecture, Hyden, 1976.

PEPE Software

[Bl 77] Blakely, C. E., "PEPE Application to BMD Systems", 1977 International Conference on Parallel Processing, 1977.

[Co 72] Cornell, J. A., "Parallel Processing of Ballistic Missile Defense Radar Data with PEPE", IEEE COMP CON 1972 Digest.

[Dl 73] Dingeldine, J. R., et al, "Operating System and Support Software for PEPE", 1973 Sagamore Computer Conference on Parallel Processing, 1973.

[We 77] Welch, H. O., "Numerical Weather Prediction in the PEPE Parallel Processor", 1977 International Conference on Parallel Processing, 1977.

Parallelism Detection

[Go 71] Gonzalaz, M. J. Jr., and C. V. Ramamoorthy, "Program Suitability for Parallel Processing", IEEE Transactions on Computers, C-20, June 1971.

[Ku 74] Kuck, D. J., et al, "Measurements of Parallelism in Ordinary FORTRAN Programs", Computer, Jan. 1974.

[Ku 76] Kuck, D. J., "Parallel Processing of Ordinary Programs", Advances in Computers Volume 15, Edited by M. Rubinoff and M. Yovits, Academic Press, 1976.

[Ra 76] Ramamoorthy, C. V. and W. H. Lering, "A Scheme for the Parallel Execution of Sequential Programs", 1976 International Conference on Parallel Processing, 1976.

APPENDICES

Arithmetic Control Unit (ACU) - Controls the Arithmetic Units (AU) in the PEPE elements so that they can execute the required parallel algorithms in an efficient manner. In addition, it executes sequential instructions retrieved from the program memory (PGRM) similar to that of a conventional processor.

Associative Output Unit (AOU) - Performs computations including, but not limited to, those required in the output of data from the PE to the Control Unit.

Associative Output Control Unit (AOCU) - Similar to ACU except controls the associative output units (AOU) in the PEPE elements.

Arithmetic Unit (AU) - Performs computations including, but not limited to, those required to apply complex arithmetic functions to data contained in element memory (EM).

Bay Signal Distributor (BSD) - The portion of the SDS contained in the Element Bay.

Central Signal Distributor (CSD) - The portion of the SDS contained in the Control Console.

Control Console (CC) - That portion of PEPE consisting of all of the functional units except the PE.

Correlation Control Unit (CCU) - Similar to ACU except controls the correlation units (CU) in the PEPE elements.

Correlation Unit (CU) - Performs computations including, but not limited to, those required to correlate incoming data with data already resident in the element.

Data Memory (DATA) - One of three memories, one in each control unit, used to hold data in transit between the Host computer and the elements.

Element Bay (EB) - That portion of PEPE consisting of up to 36 processing elements (PE). A complete PEPE system consists of eight element bays.

Element Memory (EM) - A word addressed memory of 2048 locations contained in each processing element (PE). It is shared between the AU, AOU, and CU.

Element Memory Control (EMC) - Unit which performs conflict resolution on requests from the three control units for use of element memory.

Intercommunication Logic (ICL) - A unit in the PEPE Control Console which handles:

Inter-control unit communication

Control unit interrupts

PEPE status and maintenance

Data collection and timing

Input-Output Unit (IOU) - Provides communication between the PEPE control units and external computers or another PEPE.

Interval Timer (IT) - A programmable timer in the ICL which can be set to interrupt the ACU after a time-out.

Maintenance Control and Diagnostic Unit (MCDU) - Unit which can perform test and diagnostic operations on each control under either manual or external computer control.

Micro-Program Memory (MPM) - A programmable and alterable memory in each SCL and PICU which controls registers and gating in order to execute the micro-sequences that comprise an instruction.

Output Data Control (ODC) - Unit which performs conflict resolution on requests from the ACU and AOCU for use of the output data bus from the elements.

Parallel Instruction Execution - The process of the PICU receiving instructions from the SCL, translating the instruction into micro-sequence control bits and transmitting these controls to the element for execution.

Processing Element (PE) - One minimum building block of PEPE. Includes one AU, AOU, and CU and one EM.

PE Clock Distributor (PECLKD) - The logic which receives a square wave clock signal from the BSD and generates all clock and write enable pulses required by the PE.

Parallel Element Processing Ensemble (PEPE) - A highly-parallel computer consisting of three control units and 288 content-addressable processing elements (PE). PEPE is capable of executing three independent instruction streams simultaneously.

Program Memory (PGRM) - One of three memories, one in each control unit, used to hold the program to be executed by that control unit. Contains both parallel and sequential instructions.

Parallel Instruction Control Unit (PICU) - Unit which transmits controls and data to the element. One PICU in each control unit.

Parallel Instruction Queue (PIQ) - First-in, first-out buffer between the ACU SCL and the ACU PICU. Used to increase the execution overlap of long arithmetic parallel instructions with shorter arithmetic sequential instructions.

Real Time Clock (RTC) - A clock in the ICL which can follow real-time or simulated real-time. Can cause an interrupt to the ACU when it equals a programmable value.

Routing - The process whereby the SCL fetches an instruction from PGRM, determines if it is a parallel or sequential instruction and transfers it to the sequential execution sub-unit of the SCL or to the PICU for parallel execution.

Select First (SF) - A distributed logic function whereby the first active element is selected to perform a sequence of instructions while non-first elements remain idle.

Select Highest/Select Lowest (SH/SL) - A distributed logic function whereby the active element with the maximal/minimal binary value in its A register is selected to perform a sequence of instruction while non-maximal/minimal elements remain idle.

Sequential Control Logic (SCL) - That part of a control unit that performs instruction fetching, sequential instruction execution and parallel instruction preparation.

Signal Distribution System (SDS) - The logic which carries data and control lines from the PICU to the PE and which returns data and status signals from the PE to the PICU and SCL. Contains the Distributed Logic.

Sub-Processor - One of the three processors in the processing element (PE). The three sub-processors are the AU, AOU and CU.

Test and Maintenance Computer (T & M) - An external computer interfaced with PEPE via the MCDU and IOUs. The T & M can have complete control over PEPE for program loading, execution and debugging and for maintenance and diagnostic functions. The T & M is a Burroughs B1714 system.

1. CLOCK RATE - 10 MHZ, ± 0.01% Crystal Controlled

2. MECL 10K

   - 29 types used
   - 16 and 24 pin Dual-In-Line package (DIP)
   - 2 to 30 gates per 16 pin DIP (62 gates per 24 pin DIP)
   - 2 nanosecond delay and rise time

3. NO. OF CIRCUIT TYPES USED (excluding memory) - 29

   - PE BAY (11 PE system) - 10,000 IC DIPS
   - (36 PE system) - 33,000 IC DIPS
   - (288 PE system) - 262,000 IC DIPS
   - CONTROL CONSOLE - 11,000 IC DIPS
   - TOTAL EQUIVALENT GATES (288 PE system) - 3 million
11 gates/DIP average

4. MEMORY TYPES AND SIZES

- Control Console SCL MPM, 3-1K X 48:	144 IC DIPS
- Control Console DATA & PGM, 5-4K X 32;	640 IC DIPS
- Control Console ACU PGM, 1-32K X 32;	1024 IC DIPS
- Control Console PICU MPM, 3-1K X 80;	240 IC DIPS
- ELEMENT BAY EM, 11-2K X 32;	704 IC DIPS
- ELEMENT BAY CU Reg. 11-16 X 32;	88 IC DIPS

TOTAL MEMORY DIPS = 2840
(2 circuit types)

5. IMPEDANCE CHARACTERISTICS

   - Laminated Copper backplane used for low impedance power distribution
   - 50 ohm transmission lines used for signal distribution throughout the system - including all boards and backplanes.

6. NUMBER OF BOARDS

- Control Console,	6 layer -	105 boards
No. of types,	6 layer -	41
- Element Bay,	8 layer -	66 boards
	6 layer -	6 boards
No. of types,	8 layer -	6
	6 layer -	2

APPENDIX B, PEPE HARDWARE FEATURES

7. COOLING TECHNIQUES

- Cooled Forced Air with Fin and Tube Heat Exchanger
- Direct Liquid Cooling

8. POWER SUPPLY

- High Frequency Power Conversion
- Output - 4300 Watts
  - 600 Amps at 5.2V
  - 600 Amps at 2.0V
- Efficiency ≥ 70%
- Regulation; ± 2%
- Fault Control - over/under voltage
  - over current
  - over temperature
  - air flow
- Output Rectification - Schottky Diodes
- Cooling - Direct Liquid Cooling for Schottky Diodes and Power Switches

		0	1	2	3	4	5	6	7	
	00									
	01	ACT	CACT	POP	PSEL	PUSH				-ACTIVITY
Move Data	02		LDA		LDQ			LDE	LTAG	-LOAD
	03		STA		STQ				STAG	-STORE
	04			TQA	TAQ			TIDA	OTA	-TRANSFER
	05									
	06									
	07									
	10	RBIT	SBIT	STG	STGA	STGI	CF	SFF	ROV	-SET/RESET
Control	11									
	12	SEG	SZL	SGZ	SLZ	SGE	SLE	SNZ	SZR	-SELECT
	13									
	14	SNOV							SOV	
	15	SF							SELB	
	16					SH	SL	SNZL	SNG	
Arithmetic/ Logical	17		SHAI		SHL					-SHIFT
	20	ADD	SB	ML	DV	SQ	FIX	FLOT		-FLOATING POINT
	21	ADE	SBE				UADI	USBI		-SPECIAL
	22	ADI	SBI	MLI	LINA	LDEA	LADI	LSBI		-INTEGER
	23									
	24			ANA	ANNT	ORA	XOA	OCL	TCI	-LOGICAL
	25	COPY	CA	ANS	CAS	ORS				-STACK

Appendix C Instruction Set of the AU

Listed in the following tables are the AU instructions. First is a list ordered by mnemonic and second is a list ordered by opcode with a brief description of the operation of each instruction.

MNEMONIC - Four character code accepted by the PEPE
Assembler

TYPE - Instruction class

1	Activity
2	Integer
3	Logical/Data Transfer
4	Floating Point
5	Output
6	Distributed

OPCODE - 8 bit binary

STEPS - Number of 100 ns clock periods required for execution.

MNEMONIC	TYPE	OPCODE	STEPS	MNEMONIC	TYPE	OPCODE	STEPS
ACT	1	010	3	SBI	2	221	2
ADD	4	200	8	SBIT	1	101	4
ADE	4	210	2	SEG	1	120	2
ADI	2	220	2	SELB	1	157	4
ANA	3	242	2	SF	1	150	2
ANNT	3	243	2	SFF	1	106	1
ANS	1	252	1	SGE	1	124	1
CA	1	251	1	SGZ	1	122	1
CACT	1	011	3	SH	6	164	35
CAS	1	253	1	SHAI	2	171	2
CF	1	105	1	SHL	3	173	2
COPY	1	250	1	SL	6	165	35
DV	4	203	38	SLE	1	125	1
FIX	4	205	5	SLZ	1	123	1
FLOT	4	206	3	SNG	1	167	2
LADI	2	225	2	SNOV	1	140	1
LDA	3	021	2	SNZ	1	126	1
LDE	3	026	2	SNZL	1	166	1
LDEA	2	224	2	SOV	1	147	1
LDQ	3	023	2	SQ	4	204	32
LINA	2	223	2	STA	5	031	1
LSBI	2	226	2	STAG	5	037	1
LTAG	3	027	1	STG	3	102	1
ML	4	202	19	STGA	3	103	1
MLI	2	222	15	STGI	3	104	1
OCL	3	246	1	STQ	5	033	1
ORA	3	224	2	SZL	1	121	1
ORS	1	254	1	SZR	1	127	1
OTA	5	047	1	TAQ	3	043	1
POP	1	012	1	RXI	2	247	1
PSEL	1	013	5	TIDA	3	046	2
PUSH	1	014	1	TQA	3	042	1
RBIT	1	100	4	UADI	2	215	2
ROV	2	107	1	USBI	2	216	2
SB	4	201	8	XOA	3	245	2
SBE	4	211	2				

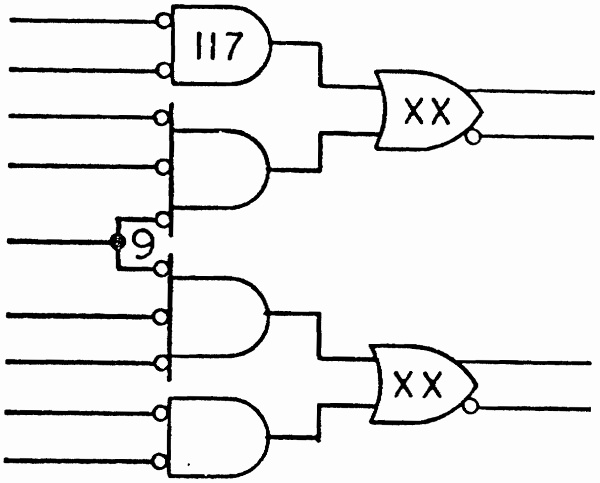
```
**
**
**
**                AU INSTRUCTION SET
**
**
**  ACTIVITY INSTRUCTIONS
**
        010 ACT  ACTIVATE       PAEACT=TAG REGISTER INPUT LOGIC
        011 CACT CLEAR AND      PAEACT=1 IF TAG MATCH ELSE 0
                 ACTIVATE
        012 POP  POP STACK      PAEACT=PASTAK(0)
                                PASTAK(19-0)=PASTAK(20-1)
                                PASTAK(20)=0
        013 PSEL PUSH AND       COMBINATION  OF PUSH AND SELB
                 SELECT ON BIT
        014 PUSH PUSH ACTIVITY  PASTAK(0)=PAEACT
                 STACK
                                PASTAK(20-1)=PASTAK(19-0)
**
**  LOAD INSTRUCTIONS
**
        021 LDA  LOAD A REG     PAAREG=OPERAND
        023 LDQ  LOAD Q REG     PAQREG=OPERAND
        026 LDE  LOAD A         PAAREG(31-24)=OPERAND(31-24)
                 EXPONENT
        027 LTAG LOAD TAGS      PATAGR=OPERAND(7-0)
                                PADPC=OPERAND(8)
                                PAEACT=OPERAND(10)
                                PASTAK=OPERAND(31-11)
**
**  STORE INSTRUCTIONS
**
        031 STA  STORE A        EM(OPERAND(10-0))=PAAREG
        033 STQ  STORE Q        EM(OPERAND(10-0))=PAQREG
        037 STAG STORE TAGS     EM(OPERAND(10-0))=REGISTERS IN THE
                                SAME FORMAT AS LTAG
**
**  TRANSFER INSTRUCTIONS
**
        042 TQA  TRANSFER Q     PAAREG=PAQREG
                 TO A
        043 TAQ  TRANSFER A     PAQREG=PAAREG
                 TO Q
        046 TIDA TRANSFER ID    PAAREG(8-0)=PE NUMBER
                 TO A
        047 OTA  OUTPUT TO A    PAODAT=PAAREG
        100 RBIT RESET BIT      EM(OPERAND(10-0)(OPERAND(15-11))=0
        101 SBIT SET BIT        EM(OPERAND(10-0)(OPERAND(15-11))=1
        102 STG  SET TAG IN     PATAGR=OPERAND(7-0). .AND.
                 ACTIVE AU      OPERAND(15-8).AND.PAEACT=1
        103 STGA SET TAG IN     PATAGR=OPERAND(7-0).AND.
                 ALL AU         OPERAND(15-8)
        104 STGI SET TAG IN     PATAGR=OPERAND(7-0).AND.
                 IN ACTIVE AU   OPERAND(15-8).AND.PAEACT=0
        105 CF   CLEAR FAULT    PAFALT=0
        106 SFF  SET FAULT      PAFALT=1 IF PAEACT=1
        107 ROV  RESET OVERFLOW PAOVFF=0
```

```
**
**  SELECT INSTRUCTIONS
**
          120 SEG  SELECT ON EQUAL GLOBAL
          121 SZL  SELECT ON ZERO LOGICAL
          122 SGZ  SELECT ON GT ZERO
          123 SLZ  SELECT ON LT ZERO  ** SET PAEACT=0 IN THOSE
          124 SGE  SELECT ON GE ZERO  ** ACTIVE AU WHERE THE CONDITION
          125 SLE  SELECT ON LE ZERO  ** IS NOT SATISFIED
          126 SNZ  SELECT ON NE ZERO
          127 SZR  SELECT ON EQ ZERO
          140 SNOV SELECT ON NON-OVERFLOW
          147 SOV  SELECT ON OVERFLOW
          150 SF   SELECT FIRST   SET PAEACT TO ZERO IN ALL
                                  AU EXCEPT THE ONE WITH THE LOWEST
                                  PE NUMBER
          157 SELB SELECT ON BIT  SET PAEACT TO ZERO IN ALL ACTIVE AU
                                  WHERE EM(OPERAND(10-0)(OPERAND(15-11))
                                  IS NOT EQUAL TO ZERO
          164 SH   SELECT HIGEST  ** SET PAEACT TO ZERO IS ALL ACTIVE AU
          165 SL   SELECT LOWEST  ** WHOSE A REGISTER CONTAIN A
                                  ** NON MAXIMAL/MINIMAL VALUE
          166 SNZL SELECT ON NON ZERO LOGICAL
          167 SNG  SELECT ON NOT EQUAL GLOBAL
**
**  ARITHMETIC INSTRUCTIONS
**
          171 SHAI SHIFT ARITHMETIC INTEGER
          173 SHL  SHIFT LOGICAL
          200 ADD  ADD FLOATING POINT
          201 SUB  SUBTRACT FLOATING POINT
          202 ML   MULTIPLY FLOATING POINT
          203 DV   DIVIDE FLOATING POINT
          204 SQ   SQUARE ROOT FLOATING POINT
          205 FIX  FIX
          206 FLOT FLOAT
          210 ADE  ADD EXPONENT    PAAREG(31-24)=PAAREG(31-24) PLUS
                                   OPERAND(31-24)
          211 SBE  SUBTRACT EXPONENT
          215 UADI UPPER ADD INTEGER
          216 USBI UPPER SUBTRACT INTEGER
          220 ADI  ADD INTEGER
          221 SBI  SUBTRACT INTEGER
          222 MLI  MULTIPLY INTEGER
          223 LINA LOAD AND INCREMENT A
          224 LDEA LOAD AND DECREMENT A
          225 LADI LOWER ADD INTEGER
          226 LSBI LOWER SUBTRACT INTEGER
```

```
**
**  LOGICAL INSTRUCTIONS
**
        242 ANA  LOGICAL .AND.
        243 ANNT LOGICAL .ANDNOT.
        244 ORA  LOGICAL .OR.
        245 XOA  LOGICAL .XOR.
        246 OCL  LOGICAL .NOT.
        247 TCI  TWOS COMPLEMENT INTEGER
**
**  ACTIVITY INSTRUCTIONS
**
        250 COPY COPY ACTIVITY  PAEACT=PASTAK(0)
                 STACK
        251 CA   COMPLEMENT     PAEACT= .NOT. PAEACT .AND. PASTAK(0)
                 ACTIVITY
        252 ANS  AND OF STACK   PAEACT=PAEACT .AND. PASTAK(0)
                                PASTAK(20-0)=0,PASTAK(19-1)
        253 CAS  CLEAR ACTIVE   PASTAK(0)=0
                 STACK
        254 ORS  OR OF STACK    PAEACT=PAEACT .OR. PASTAK(0)
                                PASTAK(20-0)=0,PASTAK(19-1)
```

10101	Quad OR/NOR
10102	Quad 2 Input OR
10104	Quad 2 Input AND/NAND
10105	Triple 2-3-2 Input OR/NOR
10106	Triple 4-3-3 Input NOR
10107	Triple XOR
10109	Dual 4-5 Input OR/NOR
10110	Dual 3 Input OR
10111	Dual 3 Input NOR
10115	Quad Line Receiver
10116	Triple Line Receiver
10117	Dual 2 wide 2-3 Input OR/NOR
10121	4 Wide 3 Input OR/AND
10135	Dual JK Flip-Flop
10141	4 bit Shift Register
10161	1 to 8 Decoder
10164	8 Input Multiplexer
10165	8 Input Priority Encoder
10173	Quad 2 Input Multiplexer
10174	Dual 4 Input Multiplexer
10176	Hex D Flip-Flop
10179	Look Ahead Carry Block
10180	Dual Full Adder
10181	4 Bit Arithmetic Logic Unit (ALU)

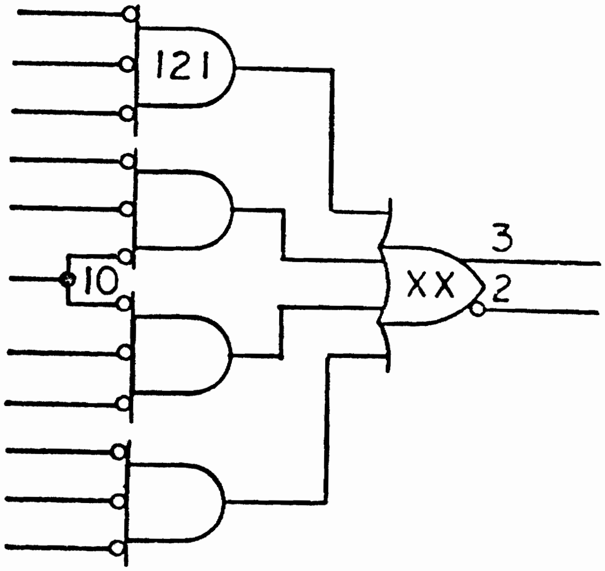
APPENDIX D, MECL 10K Logic Family Used in the AU



ECL 10117

Dual 2 Wide 2-3 Input

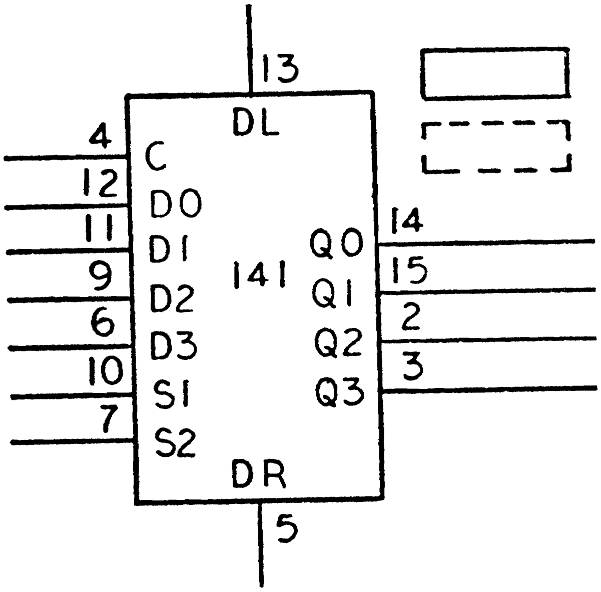
OR/AND



ECL 10121

4 Wide 3 Input

OR/AND



ECL 10141

4 Bit Shift Register

FUNCTION TABLE

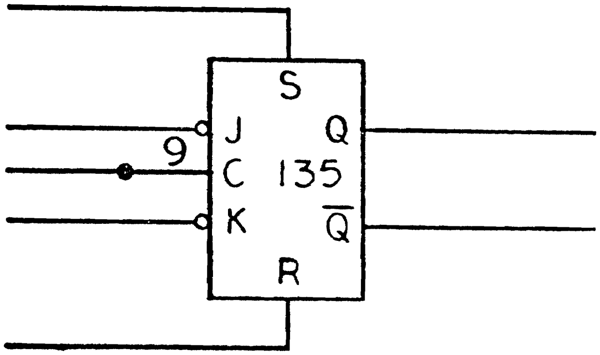
FUNCTION TABLE		
SELECT		OPERATING MODE
S1	S2	
L	L	Parallel Entry
L	H	Shift Right
H	L	Shift Left
H	H	Stop Shift

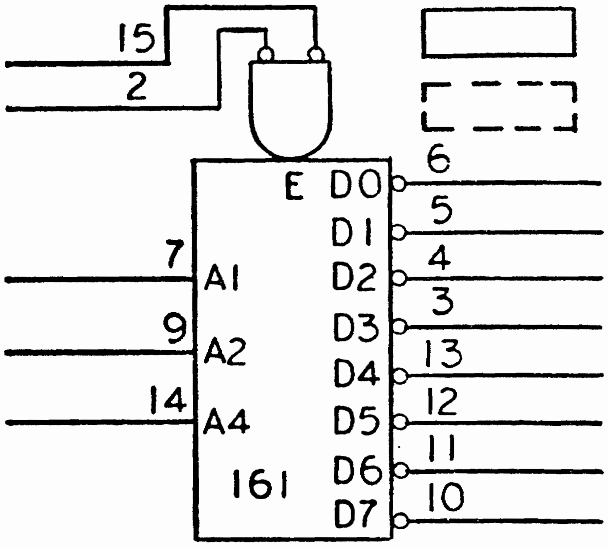
TRUTH TABLE

PULSE	INPUTS								OUTPUTS*			
	S1	S2	D3	D2	D1	D0	DR	DL	Q3	Q2	Q1	Q0
0	L	L	L	H	H	L	X	X	–	–	–	–
1	L	L	L	H	H	L	X	X	L	H	H	L
2	L	L	H	L	L	H	X	X	H	L	L	H
3	L	L	H	H	L	L	X	X	H	H	L	L
4	L	H	X	X	X	X	L	X	L	H	H	L
5	L	H	X	X	X	X	H	X	H	L	H	H
6	L	H	X	X	X	X	L	X	L	H	L	H
7	L	H	X	X	X	X	L	X	L	L	H	L
8	H	L	X	X	X	X	X	L	L	H	L	L
9	H	L	X	X	X	X	X	H	H	L	L	H
10	H	L	X	X	X	X	X	H	L	L	H	H
11	H	L	X	X	X	X	X	L	L	H	H	L
12	H	L	X	X	X	X	X	L	H	H	L	L
13	H	H	X	X	X	X	X	X	H	H	L	L
14	H	H	X	X	X	X	X	X	H	H	L	L

ECL 10135

Dual J-K Flip-Flop



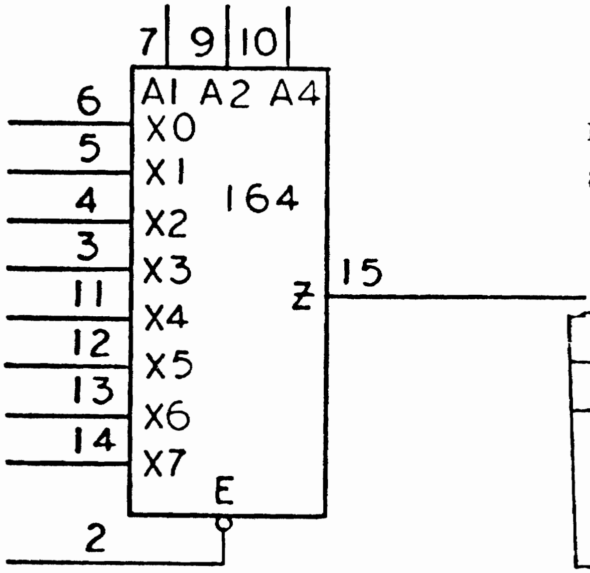


ECL 10161

1 to 8 Decoder

Enable		Inputs			Outputs							
02	15	A4	A2	A1	D0	D1	D2	D3	D4	D5	D6	D7
L	L	L	L	L	L	H	H	H	H	H	H	H
L	L	L	L	H	H	L	H	H	H	H	H	H
L	L	H	H	H	H	H	H	H	H	H	H	L
H	∅	∅	∅	∅	H	H	H	H	H	H	H	H
∅	H	∅	∅	∅	H	H	H	H	H	H	H	H

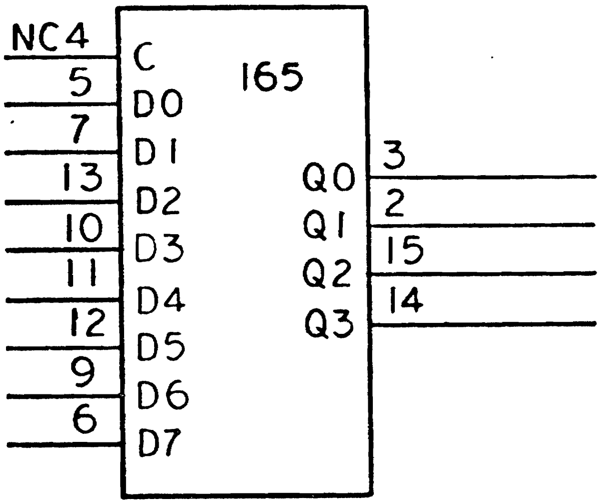
∅ = dont care



ECL 10164

8 Input Multiplexer

Enable	Inputs			Output
02	A4	A2	A1	Z
L	L	L	L	X0
L	L	L	H	X1
L	H	H	H	X7
H	∅	∅	∅	[illegible]

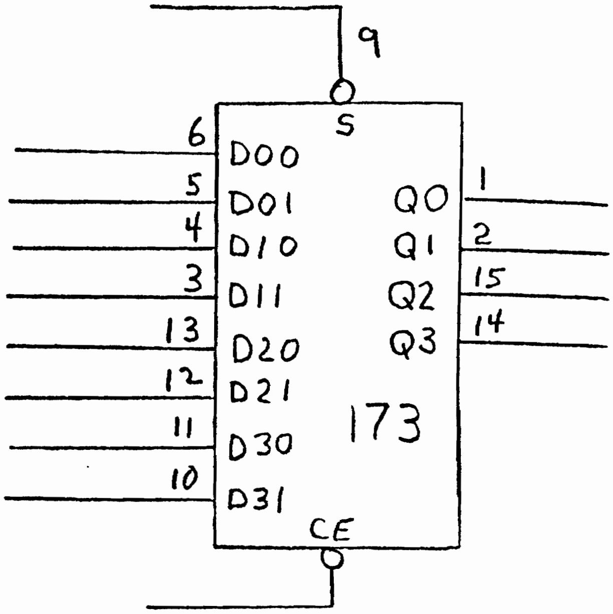


ECL 10165

8 Input Priority Encoder

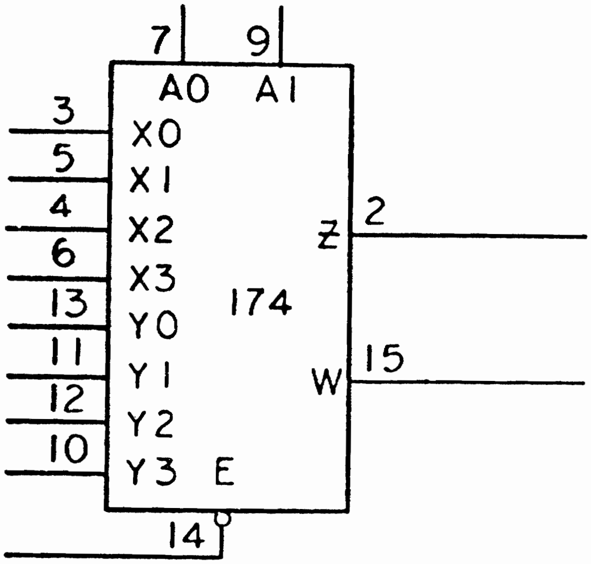
Data Inputs								Outputs			
D0	D1	D2	D3	D4	D5	D6	D7	Q3	Q2	Q1	Q0
H	∅	∅	∅	∅	∅	∅	∅	H	L	L	L
L	H	∅	∅	∅	∅	∅	∅	H	L	L	H
L	L	H	∅	∅	∅	∅	∅	H	L	H	L
			⋮								
L	L	L	L	L	L	L	H	H	H	H	H
L	L	L	L	L	L	L	L	L	L	L	L

∅= dont care



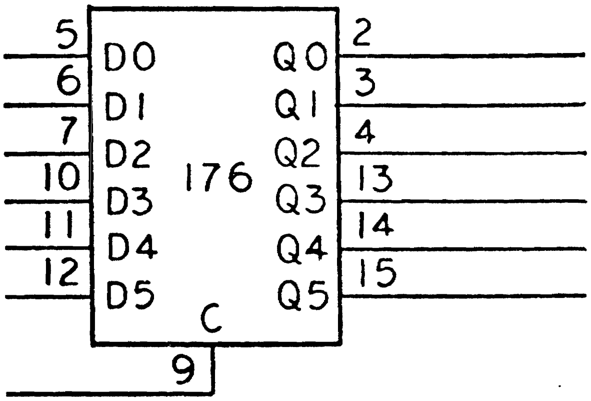
ECL 10173

Quad 2 Input Multiplexer



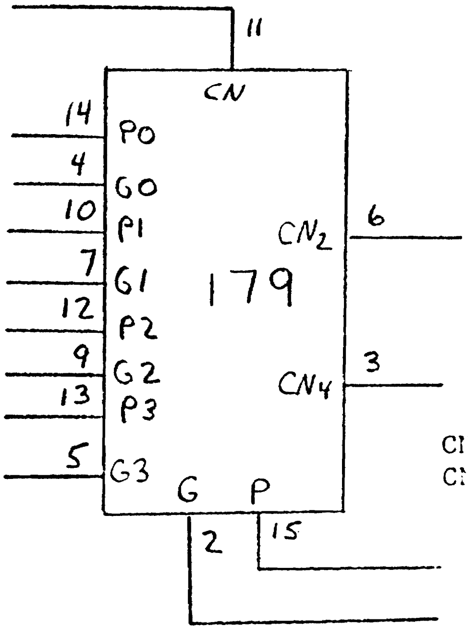
ECL 10174

Dual 4 Input Multiplexer



ECL 10176

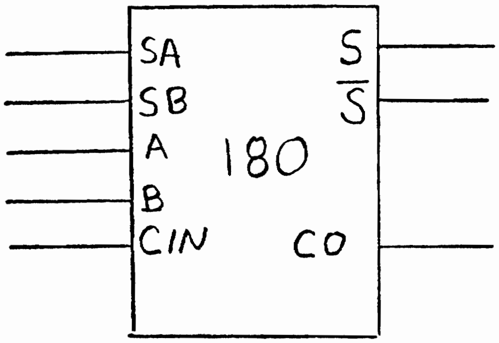
Hex D Flip-Flops



ECL 10179

Look Ahead Carry Block

CN2=CN • P0 • P1 + G0 • P1 + G1
CN4=CN • P0 • P1 • P2 • P3 +
G0 • P1 • P2 • P3 +
G1 • P2 • P3 +
G2 • P3 +
G3

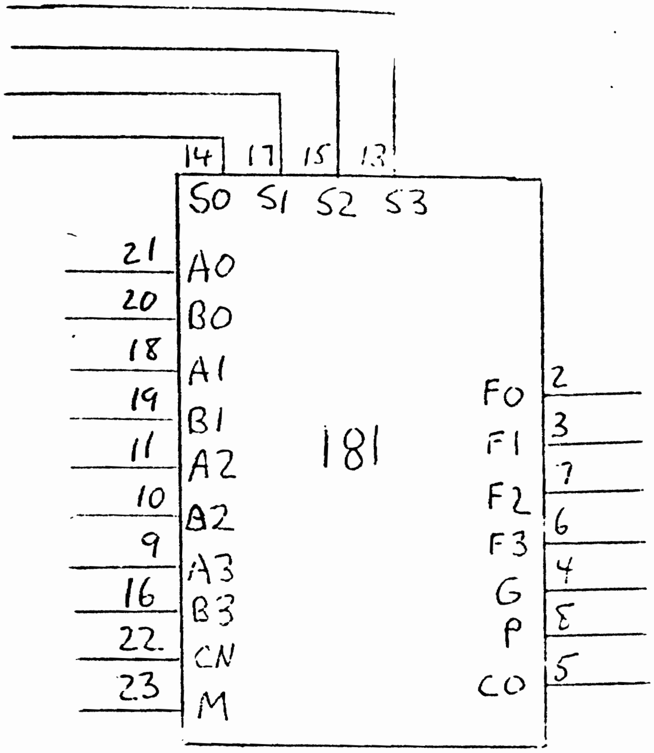


ECL 10180

Dual Full Adders

Function Select Table

SA	SB	FUNCTION
H	H	S=A plus B
H	L	S=A minus B
L	H	S=B minus A
L	L	S=0 minus A minus B



ECL 10181

4 Bit Arithmetic
Logic Unit

Operations of Interest

Select				Mode	Output
S3	S2	S1	S0	M	
L	H	H	L	L	A plus B plus CN (add)
H	L	L	H	L	A plus $\overline{B}$ plus CN (subtract)
L	H	H	L	H	A exclusive OR B
H	L	H	H	H	A OR B
H	H	L	H	H	A AND $\overline{B}$
H	H	H	L	H	A AND B

Signal Mnemonic	Definition
PAALO	Alignment Network Control
PAADEA	A Input to Exponent Adder
PAADEB	B Input to Exponent Adder
PAADMA	A Input to Mantissa Adder
PAADMB	B Input to Mantissa Adder
PAALFA	Search: ALPHA Flip-Flop
PAALNA	Alignment Network Output of Stage One
PAALNB	Alignment Network Output of Stage Two
PAALNC	Alignment Network Output of Stage Three
PAALNI	Input to Alignment Network
PAALNO	Output of Alignment Network
PAAOUT	ALU Output
PAAREG	A Register
PAAXB	Local Control - Extension Adder B Input
PABETA	Search: BETA Flip-Flop
PABEXT	B Extension Flip-Flop
PABITA	Search: Decoded Output
PABITB	
PABITC	
PABITX	Search: Input to AU
PABITY	
PABITZ	
PABREG	B Register

Signal Mnemonic	Definition
PACIN	Carry Input to Mantissa Adder
PACOFF23	Carry Output Flip-Flop
PACOUT23	Carry Out of Position 23
PACOUT22	Carry Out of Position 22
PACOXT	Carry Out of Extension Adder
PADAIN	Global Address to Element Memory
PADPC	Double Precision Carry Flip-Flop
PADPIN	Local Control for DPC Input
PADSEL	Global Control EM Bus Enable
PAEACT	Element Activity Flip-Flop
PAEXOF	Exponent Adder Add Overflow
PAEXUF	Exponent Adder Add Underflow
PAGEN	Global Clock Control for the AU
PAMLOF	Detection of All Ones or Zeros from Mantissa Adder
PAMOBR	PICU Operand Buffer Register
PAMOF	Mantissa Overflow
PAMORL	Global Data/Control Inputs
PAMOVB	Local Control for B Register(23,22)
PANRMD	Normalize Decode Network Output
PAODAT	Output Data Bus
PAOVFF	Overflow Flip-Flop
PAQSEL	Local Control for Q Extension Bit
PASBOF	Exponent Adder Subtract Overflow