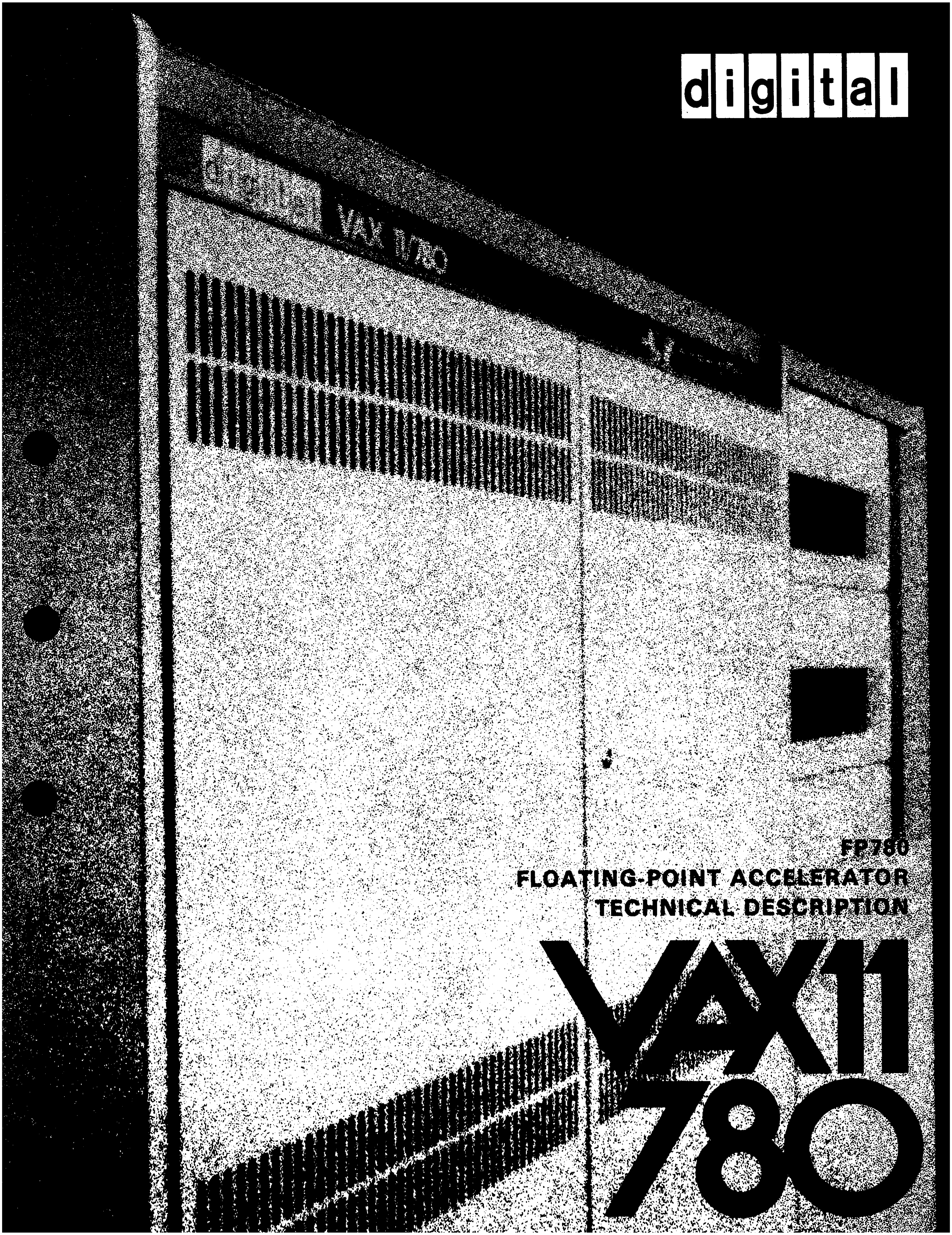
digital



FP780 FLOATING-POINT ACCELERATOR TECHNICAL DESCRIPTION

VAX11 780

EK-FP780-TD-001

FP780 Floating-Point Accelerator
Technical Description

digital equipment corporation • maynard, massachusetts

1st Edition, December 1978

The material in this manual is for informational purposes and is subject to change without notice. Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

Printed in U.S.A.

This document was set on DIGITAL's DECset-8000 computerized typesetting system.

CONTENTS

Page

CONTENTS (Cont)

FIGURES

TABLES

The FPA is a microprogrammed device operating as a synchronous extension of the CPU data path. Both the FPA and CPU operate using a 200 ns microcycle; FPA T0 coincides with CPU T0. As an extension of the CPU, the FPA does not access memory data. The CPU must do memory address calculations, access the calculated address, and transmit the accessed data to the FPA. The CPU is also responsible for fetching and storing the FPA results. The FPA performs only the required floating-point or integer operation on the properly formatted operands transmitted to it.

The FPA can do floating-point addition, subtraction, multiplication, and division instructions. It receives a packed, normalized floating-point number containing a sign bit, fraction bits, and exponent bits. The FPA breaks the number into parts and FPA data manipulation sections perform the operations required to carry out the instructions on each part. Once the result is completed, it normalizes and packs the result for return to the CPU. Refer to Figure 1-1, a simplified diagram of the FPA.

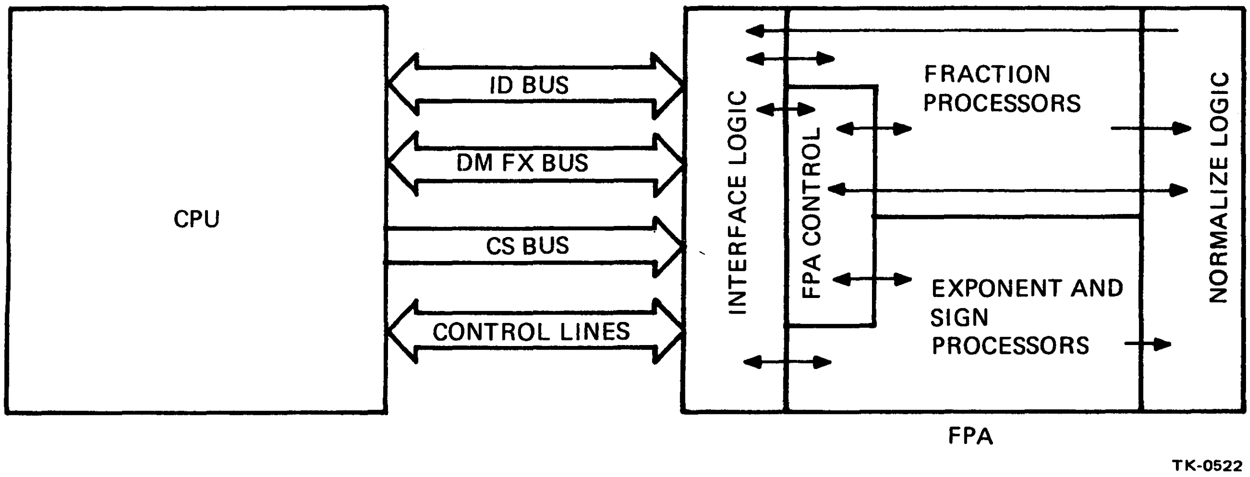


Figure 1-1 The FPA

1.1.1 Accelerator Interface

The FPA is an optional hardware extension of the VAX CPU data path. It is the first of a series of optional accelerators that can be plugged into slots 24 through 28 of the CPU backplane. To facilitate design of these optional accelerators, a set of standard interface signals and buses is used to transfer data and control information.

Two copies of the CPU general register set are kept in the FPA. These are read-only memory to the FPA and provide rapid access to register operands when used in instructions. Every time the CPU general registers are updated, a copy of the update data is transmitted via the DFMX bus to the FPA copies and changes them.

All other data (memory and literal) is transmitted to the accelerator via the ID bus. Memory data is transferred into the CPU D register and then onto the ID bus. Literal data is transferred from the instruction buffer via the ID bus.

All op codes are received from the instruction buffer. The FPA uses dedicated hardware to handle certain op codes. The op codes are decoded and, if part of the FPA implemented set, processing is started.

FPA results are returned to the CPU via the DFMX bus. Any transfer of data (either operands or results) between the CPU and FPA is controlled by the CPSYNC and FPSYNC. CPSYNC is transmitted via the CS bus. When an operand is transferred to the FPA, CPSYNC asserted (by the CPU) indicates that data is available on the ID bus and FPSYNC is asserted (by the FPA) to indicate data has been received. When the FPA is returning a result, FPSYNC indicates result available and CPSYNC indicates result received. When a result is transferred, the FPA also transmits the proper condition codes to the CPU.

Traps and errors are handled with three signals: ACC ERROR (from FPA to CPU), FP TRAP (CPU to FPA), and ACC TRAP (CPU to FPA). ACC ERROR (also called ERRSYNC) is asserted when the FPA detects an internal error and is input to the CPU BEN mux. FP TRAP is used by the CPU to initiate microdiagnostics stored in the FPA. ACC TRAP selects either the power-up trap or the abort trap (both stored in the FPA microcode).

1.2 FPA INSTRUCTION SET

The FPA handles only a limited number of instructions (refer to Table 1-2). No floating-point instructions are available in VAX's PDP-11 compatibility mode. As shown in the table, the FPA handles single and double precision instructions in both 2 and 3-operand formats. The FPA handles the single and double precision instruction variations internally. However, as stated before, the FPA does no memory accessing. This means the CPU must do all address calculations and accessing for any input operands stored in memory. Also, the FPA does not store any final results; it merely makes the results available to the DFMX bus. The'CPU must enable the result onto the DFMX bus, determine the result destination, and put it into the destination. In a 3-operand instruction, the FPA begins computing as soon as it has the 2 source operands while the CPU is computing the third, or destination, address.

Table 1-2 FPA Instruction Set

Mnemonic	Description
ADDF*	Add single-precision floating-point
ADDD*	Add double-precision floating-point
SUBF*	Subtract single-precision floating-point
SUBD*	Subtract double-precision floating-point
MULF*	Multiply single-precision floating-point
MULD*	Multiply double-precision floating-point
DIVF*	Divide single-precision floating-point
DIVD*	Divide double-precision floating-point
POLYF	Evaluate polynomial single-precision floating-point
POLYD	Evaluate polynomial double-precision floating-point
EMODF	Extended single-precision floating-point
EMODD	Extended double-precision floating-point
MULL*	Multiply integer longword

*The FPA instruction set includes both the 2-operand and 3-operand format of these instructions

1.3 PHYSICAL DESCRIPTION

The FPA consists of 5 hex-height, extended-length modules containing mostly Schottky TTL logic. They replace blank modules 7014103 in slots 24 through 28 of the KA780 backplane. These slots are designated as the accelerator option slots. The FPA is powered by an H7100 installed in power supply position 1. When viewed from the rear, position 1 is the rightmost location in the VAX CPU cabinet. Position 1 is left empty if an accelerator is not installed. The H7100 is a 5 V, 100 A supply. Refer to Figure 1-2 for the location of backplane slots and power supply. Refer to Table 1-3 for module designations and locations.

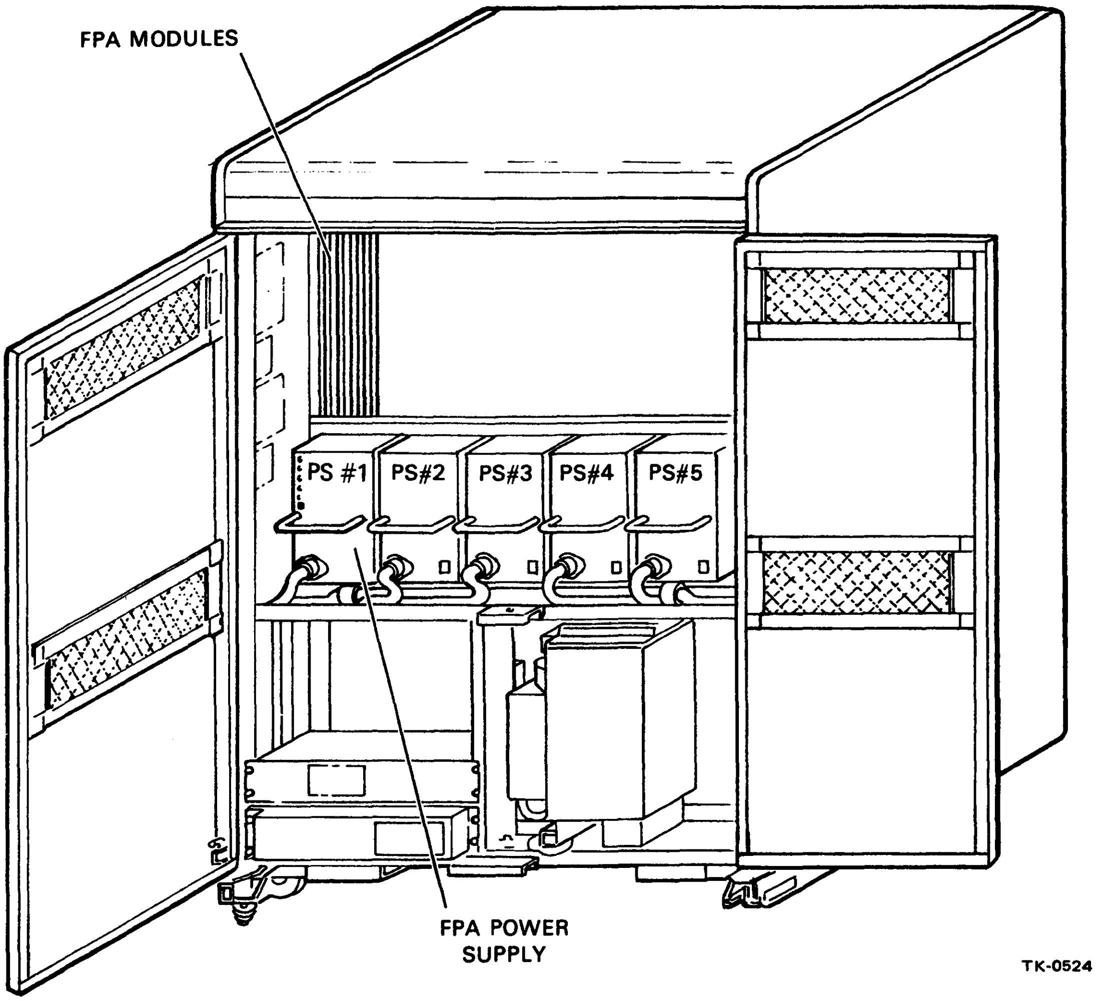


Figure 1-2 FPA Physical Locatıon

Table 1-3 FPA Modules

Module No.	Slot	Module Name	Module Function
M8285	24	FNM	Normalization and fraction division
M8286	25	FMH	Fraction multiplication (most significant bits)
M8287	26	FML	Fraction multiplication (least significant bits)
M8288	27	FAD	Fraction addition and subtraction
M8289	28	FCT	Exponent manipulation and FPA control

1.4 FLOATING-POINT NUMBERS AND ARITHMETIC

1.4.1 Introduction

This section discusses some fundamentals of floating-point numbers and arithmetic. It provides useful background for more advanced topics in later sections. The reader already familiar with floating-point may skip this section.

1.4.2 Integers

All data within a computer system could be represented in integer form. The numbers that could be represented in a 32-bit machine range in magnitude from 00000000_{16} to $FFFFFFFF_{16}$ (or from 0_{10} to 4,294,967,295). However, integer form imposes some limitations. Only whole numbers can be represented, i.e., no fraction or decimal parts; this imposes an accuracy limitation. Furthermore, numbers greater than 4,294,967,295 cannot be represented; this imposes a range limitation.

These limitations are imposed by the stationary position of the radix point (e.g., the decimal point in base 10 notation or the binary point in base 2 notation). An integer's radix point is usually omitted in integer representation because it always marks the integer's least significant place. That is, there are never any digits to the right of an integer's radix point. For this reason, an integer is sometimes called a fixed-point number.

Integer notation, however, can be modified to overcome the range and accuracy limitations imposed by the fixed radix point. This is done through the use of floating-point notation.

1.4.3 Floating-Point Numbers

Floating-point numbers, unlike integers, have no position restrictions imposed on their radix points. A popular type of floating-point representation is called scientific notation. With scientific notation, a floating-point number is represented by some basic value multiplied by the radix raised to some power.

Example

$$1{,}000{,}000 = \overset{\substack{\text{basic}\\ \text{value}\\ \downarrow}}{1.} \times \underset{\nwarrow\ \text{radix}}{10}^{\overset{\text{exponent}}{\swarrow\ 6}}$$

There are many ways to represent the same number in scientific notation, as shown in the following example.

Right shifts				Left shifts			
512 =	512.	X	10^0	512 =	512	X	10^0
=	51.2	X	10^1	=	5120	X	10^{-1}
=	5.12	X	10^2	=	51200	X	10^{-2}
=	.512	X	10^3	=	512000	X	10^{-3}

The convention chosen for representing floating-point numbers with scientific notation in the FPA requires the radix point to always be to the left of the most significant digit in the basic value (e.g., $.512 \times 10^3$ in the above example). This modified basic value is called a fraction.

Notice that for each right shift of the basic value, the exponent is incremented and for each left shift the exponent is decremented. The value of the number remains constant if the exponent is adjusted for each shift of the basic value.

More examples of scientific notation are as follows.

Decimal Notation	**Decimal Scient. No.**	**Binary Notation**	**Hex Notation**	**Hex Scient. No.**
64	$.64 \times 10^2$	1000000.	40_{16}	$.4 \times 16^{-2}$
33	$.33 \times 10^2$	100001.	21_{16}	$.21 \times 16^{-2}$
1/2(.5)	$.5 \times 10^0$	0.1	$.8_{16}$	$.8 \times 16^0$
3/32(.09375)	$.9375 \times 10^{-1}$	0.00011	$.18_{16}$	$.18 \times 16^0$

1.4.4 Decimal/Binary/Hexadecimal Conversion

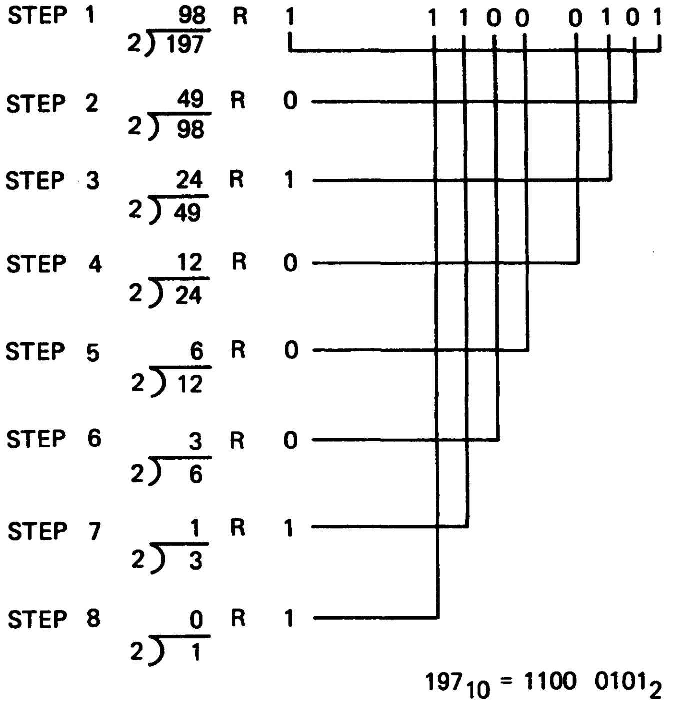
There are standard routines to convert from decimal notation to hexadecimal (also called hex) and back. When converting from either decimal-to-hex or hex-to-decimal it is convenient to first convert to binary notation and then to the final notation.

Decimal to Hex Conversion:

To convert a decimal number with both integer and fraction portion to a hex number, the integer and fraction are separated and converted individually. The integer is converted to binary by a repeated division technique, the fraction by a repeated multiplication technique.

To convert an integer to binary representation, the integer is divided by two. The remainder of this division (either 1 or 0) becomes the LSB of the binary representation. The result of this division is again divided by two. The remainder of this division goes to the left of the LSB, becoming "next to LSB." The result is divided again. This process is continued until the result is zero. Refer to Example 1.

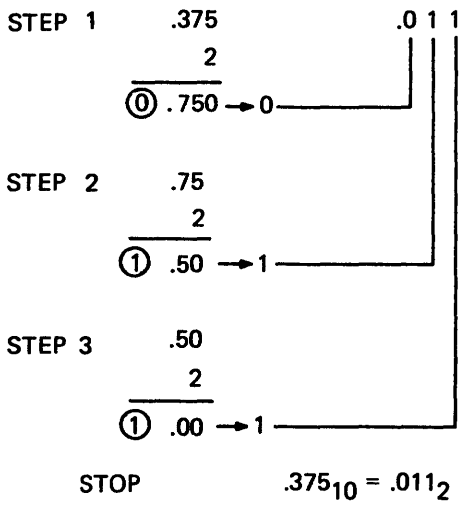
Example 1 Convert 197_{10} to binary


STEP 1 $2\overline{)197}$ 98 R 1
STEP 2 $2\overline{)98}$ 49 R 0
STEP 3 $2\overline{)49}$ 24 R 1
STEP 4 $2\overline{)24}$ 12 R 0
STEP 5 $2\overline{)12}$ 6 R 0
STEP 6 $2\overline{)6}$ 3 R 0
STEP 7 $2\overline{)3}$ 1 R 1
STEP 8 $2\overline{)1}$ 0 R 1
1 1 0 0 0 1 0 1
197_{10} = $1100\ 0101_2$

TK-0654

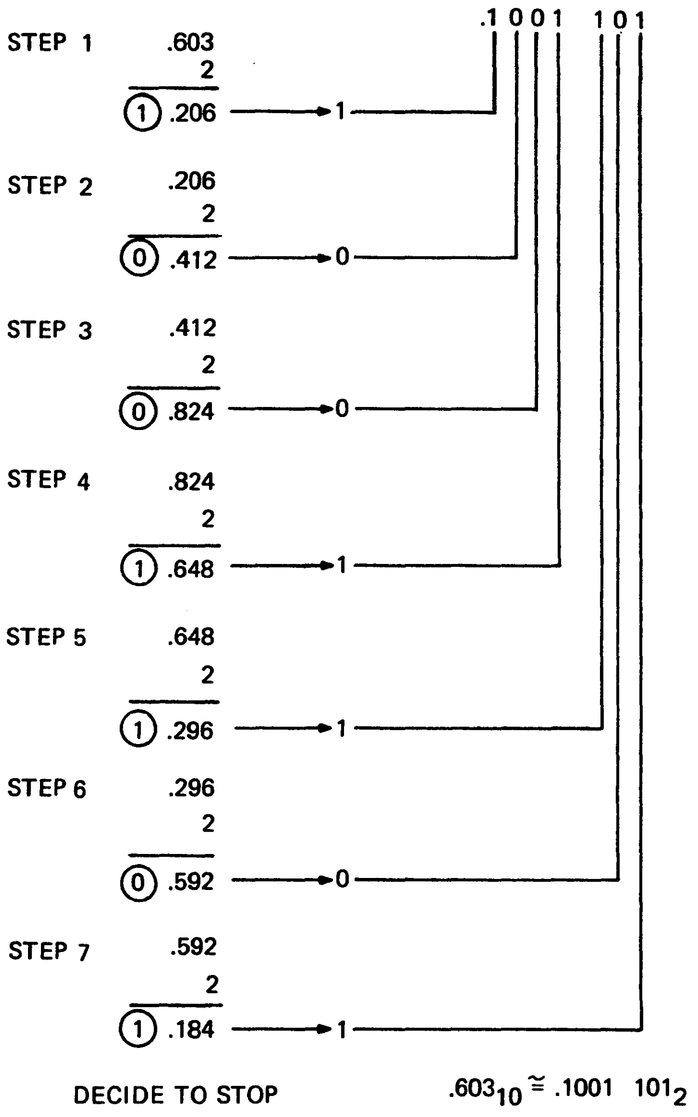
A repeated multiply-by-2 converts a decimal fraction to a binary fraction. The decimal fraction is multiplied by two. If the result is 1.0 or more, a 1 is placed in the MSB of the fraction (directly to the right of the binary point); if less than 1.0, a zero is placed there. The fraction portion only of this result is again multiplied by two, if the result is 1.0 or more, a 1 goes to the right of the MSB, less than 1.0, a zero. This continues until the fraction portion of the result is all zeros (refer to Example 2) or until enough binary fraction bits have been generated to represent the decimal accurately enough (refer to Example 3). Note that finite length decimal fractions can become repeating fractions in binary (Example 3).

Example 2   Convert 3/8 (.375) to binary



TK-0655

Example 3   Convert $.603_{10}$ to binary



TK-0656

The conversion from binary to hex is very simple. Starting at the binary point, break the binary number into groups of 4 digits each. (Zero fill at both right and left ends to complete groups of 4.) Then replace each group of 4 with its hex equivalent. Refer to Table 1-4, and Example 4.

Table 1-4 Binary-Hex Equivalents

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Example 4 Convert 110010110.101101_2 to Hex

1. Break into groups of four and zero-fill left and right ends.

$$\underbrace{\overbrace{000}^{\text{Zeros Added}}1}_{4}\ \underbrace{1001}_{4}\ \underbrace{0110}_{4}.\underbrace{1011}_{4}\ \underbrace{01\overbrace{00}^{\text{Zeros Added}}}_{4}$$

2. Replace four digit groups with hex equivalents. Refer to Table 1-4.

$$\begin{array}{ccccc} 0001 & 1001 & 0110. & 1011 & 0100 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 9 & 6 & B & 8 \end{array}$$

$$196.B8_{16}$$

$1\ 1001\ 0110.1011\ 01_2 = 196.B8_{16}$

To convert from hex back to decimal, first replace each hex digit with its 4-bit binary equivalent (refe to Table 1-4). Each position in a binary number has a positional value based on which side of the binary point it is and its distance from the binary point. The positional values are based on powers of two. The bit in the unit column has a positional value of one. The positional value doubles each time you move from right to left, and halves as you move from left to right. Refer to Figure 1-3 for a summary of binary positional values in both powers of two and decimal value.

... 2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	. 2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6} ...
128	64	32	16	8	4	2	1	½	¼	1/8	1/16	1/32	1/64
								.5	.25	.125	.0625	.03125	.015625

TK-0657

Figure 1-3 Positional Value of Binary Number

To convert from binary notation to decimal notation, add the decimal positional value of each bit that is a one. This sum will be the decimal equivalent of the binary number.

1.4.5 Normalization

As discussed previously, there are many ways to represent a particular floating-point number using scientific notation and the convention chosen for representing floating-point numbers in VAX and the FPA requires the radix point to be to the left of the most significant bit in the basic value. Refer to Example 5.

Example 5 Floating-Point Form

29_{10} = 11101_2 =		1 1101.	X	2^0	=	1 1101.	X	2^0
		1110.1	X	2^1	=	11 1010.	X	2^{-1}
		111.01	X	2^2	=	111 0100.	X	2^{-2}
.11101		11.101	X	2^3	=	1110 1000.	X	2^{-3}
Fraction		1.1101	X	2^4	=	1 1101 0000.	X	2^{-4}
5	Chosen Form →	.1110 1	X	2^5	=	11 1010 0000.	X	2^{-5}
Exponent		.0111 01	X	2^6	=	111 0100 0000.	X	2^{-6}
		.0011 101	X	2^7	=	1110 1000 0000.	X	2^{-7}

The process of ensuring that the first significant bit is directly to the right of the binary point is called normalization. If the number is one or larger it involves right-shifting the basic value and incrementing the exponent until the MSB (a one) is directly to the right of the binary point. If the number is a fraction with leading zeros the basic value is left-shifted and the exponent is decremented. Examples 6 and 7 show conversion of numbers to VAX normalized form.

Example 6 Convert 75_{10} to a normalized binary number

1. Integer conversion
   $75_{10} = 100\ 1011_2$

2. Floating-point form
   $100\ 1011_2 = 100\ 1011_2 \times 2^0$

3. Normalized form
   Right shift fraction 7 times
   Increment exponent by 7

   $100\ 1011_2 \times 2^0 = .100\ 1011 \times 2^7$

   Fraction = .100 1011
   Exponent = 7

Example 7 Convert 3/16 (.01875) to a normalized binary number.

1. Integer conversion
   $.01875_{10} = .0011_2$

2. Floating-point form
   $.0011_2 = .0011_2 \times 2^0$

3. Normalized form
   Left shift fraction 2 times
   Decrement exponent by 2

   $.0011_2 \times 2^0 = .11 \times 2^{-2}$

   Fraction = .11
   Exponent = −2

1.4.6 VAX Floating-Point Notation

Two conventions are used in the FPA to conserve memory space without losing accuracy and to aid in hardware manipulation. The first convention is called the hidden bit. All numbers transferred between the CPU and FPA are normalized floating-point numbers. This means the first significant bit (always a 1) is always directly to the right of the binary point. To conserve memory space and data lines, the first significant bit is not stored or transmitted to the FPA. For example, the fraction part of the normalized binary number .11000... $\times 2^{-2}$ will be stored and transmitted to the FPA as 100.... The normalized fraction of 1/2 (.100... $\times 2^0$) will be stored and transmitted as 000.... In both cases the first 1 (the hidden bit), will be added by hardware in the FPA. When the FPA transfers a normalized answer back to the CPU the hidden bit is not sent.

The 8-bit exponent portion of a floating-point number is stored using excess 80_{16} notation. This notation simplifies the hardware that manipulates the exponent during floating-point arithmetic operation. Excess 80_{16} exponent notation is obtained by adding 10000000_2 (200_8, 80_{16}, or 128_{10}) to 2's complement notation.

Refer to Paragraph 1.5 for a further discussion of excess 80 notation.

1.4.7 Floating-Point Addition and Subtraction

In order to perform floating-point addition or subtraction, the exponents of the two floating-point numbers involved must be aligned or equal. If they are not aligned, the fraction with the smaller exponent is shifted right until they are. Each shift to the right is accompanied by an increment of the associated exponent. When the exponents are aligned, the fractions can then be added or subtracted. The exponent value indicates the number of places the binary point is to be moved to obtain the integer representation of the number.

In example 8, the number 7_{10} is added to the number 40_{10} using floating-point representation. Note that the exponents are first aligned and then the fractions are added; the exponent value dictates the final location of the binary points.

Example 8 Floating-Point Addition

$$0.1010\ 0000\ 0000\ 000 \times 2^6 = 28_{16} = 40_{10}$$

$$+0.1110\ 0000\ 0000\ 000 \times 2^3 = 7_{16} = 7_{10}$$

1. To align exponents, shift the fraction with one smaller exponent three places to the right and increment the exponent by 3, and then add the two fractions.

$$\begin{array}{l} 0.1010\ 0000\ 0000\ 000 \times 2^6 = 28_{16} = 40_{10} \\ +0.\underset{\curvearrowleft}{000}1\ 1100\ 0000\ 000 \times 2^6 = 7_{16} = 7_{10} \\ \hline 0.1011\ 1100\ 0000\ 000 \times 2^6 = 2F_{16} = 47_{10} \end{array}$$

2. To find the integer value of the answer, move the binary point six places to the right.

$$0\underset{\curvearrowright}{10\ 1111}.0000\ 0000\ 0$$

1.4.8 Floating-Point Multiplication and Division

In floating-point multiplication, the fractions are multiplied and the exponents are added. For floating-point division, the fractions are divided and the exponents are subtracted. There is no requirement to align the binary point in the floating-point multiplication or division. Example 9 shows floating-point multiplication. Example 10 shows division.

Example 9:

Multiply 7_{10} by 40_{10}.

1.
$$\begin{array}{r} 0.1110000 \times 2^3 \\ \underline{\times 0.1010000 \times 2^6} \\ 1110000 \\ 0000 \\ 11100 \\ \hline .1000110000 \times 2^9 \end{array} \begin{array}{l} = 7 = 7_{10} \\ = 28_{16} = 40_{10} \\ \\ \\ \\ \\ \text{(Result already in normalized form.)} \end{array}$$

2. Move the binary point nine places to the right.

$\underset{\curvearrowright}{.100011000}.00000 = 118_{16} = 280_{10}$

Example 10:

Divide 15_{10} by 5_{10}.

1.
$$\frac{.1111000 \times 2^4}{.1010000 \times 2^3}$$

$$\begin{array}{r} 1.100000 \\ 1010000 \overline{)1111000.000000} \\ \underline{1010000} \\ 101000 \\ \underline{101000} \\ 0 \end{array}$$

2. Exponent: 4–3 = 1

3. Result: 1.100000×2^1

Normalized Result: $.1100000 \times 2^2$

Normalized Fraction $\nearrow$ Normalized Exponent $\nwarrow$

Move binary point two places to the right.

$\underset{\curvearrowright}{.11}.00000 = 3_{16} = 3_{10}$

1.5 EXCESS 80 NOTATION

The VAX and, consequently, the FPA use excess 80 notation to store and handle the exponent portion of floating-point numbers. Excess 80 notation is the 2's complement of exponent plus 128_{10} or 80_{16}.

It is convenient to handle the exponent portion of the floating-point number in 2's complement notation. This allows a wide range of both positive and negative exponents to be represented. However, in 2's complement notation an overflow must occur to go from the least negative number to zero. To avoid this the bias of 128_{10} is added to the 2's complement number.

Historically, minicomputers have been discussed and explained using octal notation. In octal, the bias of 128_{10} is 200_8. In previous manuals this exponent notation has been discussed using octal form. As a result, it is called excess 200_8 or excess 200. However, the VAX is discussed using hexadecimal notation. Unfortunately, when discussing the excess 80 bias in VAX documentation, it has been called 80_{16}, 128_{10}, 200_8, and 10000000_2 (sometimes the base is indicated, sometimes it isn't). When studying the FPA print sets, technical manuals, and microcode listings, be aware of this variation in terminology. In this manual hex notation is used and the exponent bias is called excess 80.

When multiply and divide operations are performed using floating-point numbers with excess 80 exponent notation the resulting exponent must be adjusted by the bias to return the result to excess 80 notation. When a multiplication is performed exponents are added, 80_{16} must be subtracted from the result to return it to excess 80 notation. To understand why 80 must be subtracted from the exponent calculation during multiplication, consider the following.

$$\begin{array}{l} \text{Exponent A} + \underbrace{80}_{} \\ \qquad\qquad\qquad \text{Excess 80 notation} \\ \text{Exponent B} + \overbrace{80}^{} \\ \hline \text{Exponent A} + \text{Exponent B} + 100 \end{array}$$

Both exponent A and exponent B are biased by 80, yielding a bias of 100. However, only a bias of 80 is desired in excess 80 notation.

Multiplication Example

2 × 3 = 6

Fraction		Exponent
2 = 0.100	X	82
3 = 0.110	X	82

Fraction Calculation		Exponent Calculation
2 = 0.100		82
3 = 0.110		+82
1000		104
100		-80
6 = 0.011000	X	84

Normalize the fraction by left-shifting one place and decreasing the exponent by 1.

$$\overset{\overset{\text{Fraction}}{\downarrow}}{0.11000} \times \overset{\overset{\text{Exponent}}{\swarrow}}{83} = 6$$

When a division is performed, exponents are subtracted and 80_{16} must be added to the result to return it to excess 80 notation. To understand why 80 must be added to the exponent calculation during division, consider the following:

$$\begin{array}{l} \text{Exponent A} + 80 \\ \underline{- \text{Exponent B} + 80} \\ \text{Exponent A} - \text{Exponent B} + 80 - 80 = \text{Exponent A} - \text{Exponent B} + 0 \end{array}$$

However, since the result is to be in excess 80 notation, 80_{16} must be added to the exponent, yielding Exponent A - Exponent B + 80.

Division Example

16/4 = 4

	Fraction		Exponent
16 =	.10000	X	85
4 =	.10000	X	83

Fraction Calculation	Exponent Calculation
$$\begin{array}{r} 1.000 \\ \underbrace{0\,10000.}\overline{)0\underbrace{10000}.000} \end{array}$$	$$\begin{array}{r} 85 \\ \underline{-83} \\ 2 \\ \underline{+80} \\ 82 \end{array}$$

Normalize the fraction by right-shifting one place and incrementing the exponent.

$$\overset{\overset{\text{Fraction}}{\downarrow}}{.10000} \times \overset{\overset{\text{Exponent}}{\swarrow}}{83} = 4$$

CHAPTER 2
FUNCTIONAL DESCRIPTION

This chapter explains the operation of the FPA. The chapter can be divided into four areas: introduction, algorithms, hardware operation, and microcode. The introduction (Paragraph 2.1) discusses the various types of data formats that may be handled by the FPA. The algorithms (Paragraph 2.2) lists the various instructions the FPA can do and explains the FPA operations required to perform each operation. This section discusses the FPA operation based on instruction flow. Hardware operation (Paragraph 2.3) breaks the FPA into hardware blocks and discusses the operation of each. Both the algorithm section and the hardware operation section should be read to get a thorough understanding of the FPA operation. They discuss the same equipment from different viewpoints. Microcode (Paragraphs 2.4 through 2.6) summarizes both the FPA microcode and the FPA specific microcode in the CPU. This discussion focuses on the generation and monitoring of the various control signals passed between the units.

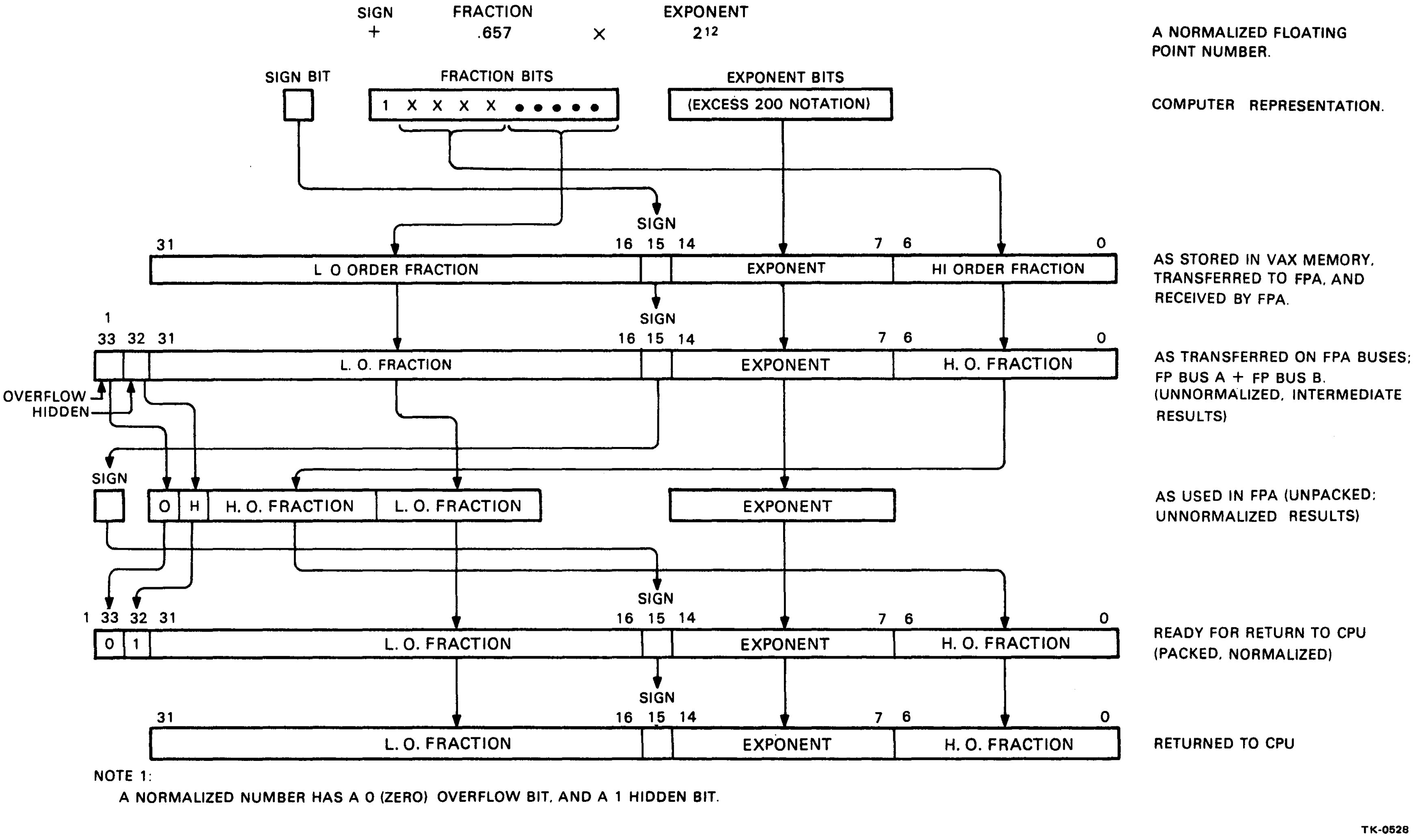
2.1 DATA FORMATS

The FPA handles single (float) and double precision floating-point data and signed integer longwords. It receives normalized, packed data from the CPU and returns normalized, packed results to the CPU over 32-bit wide buses. Within the FPA, intermediate data is transmitted over two 34-bit wide buses. The data formats used by the FPA are compatible with these bus structures as well as the input and output formats of the various data manipulation units within the FPA.

2.1.1 Floating-Point Numbers

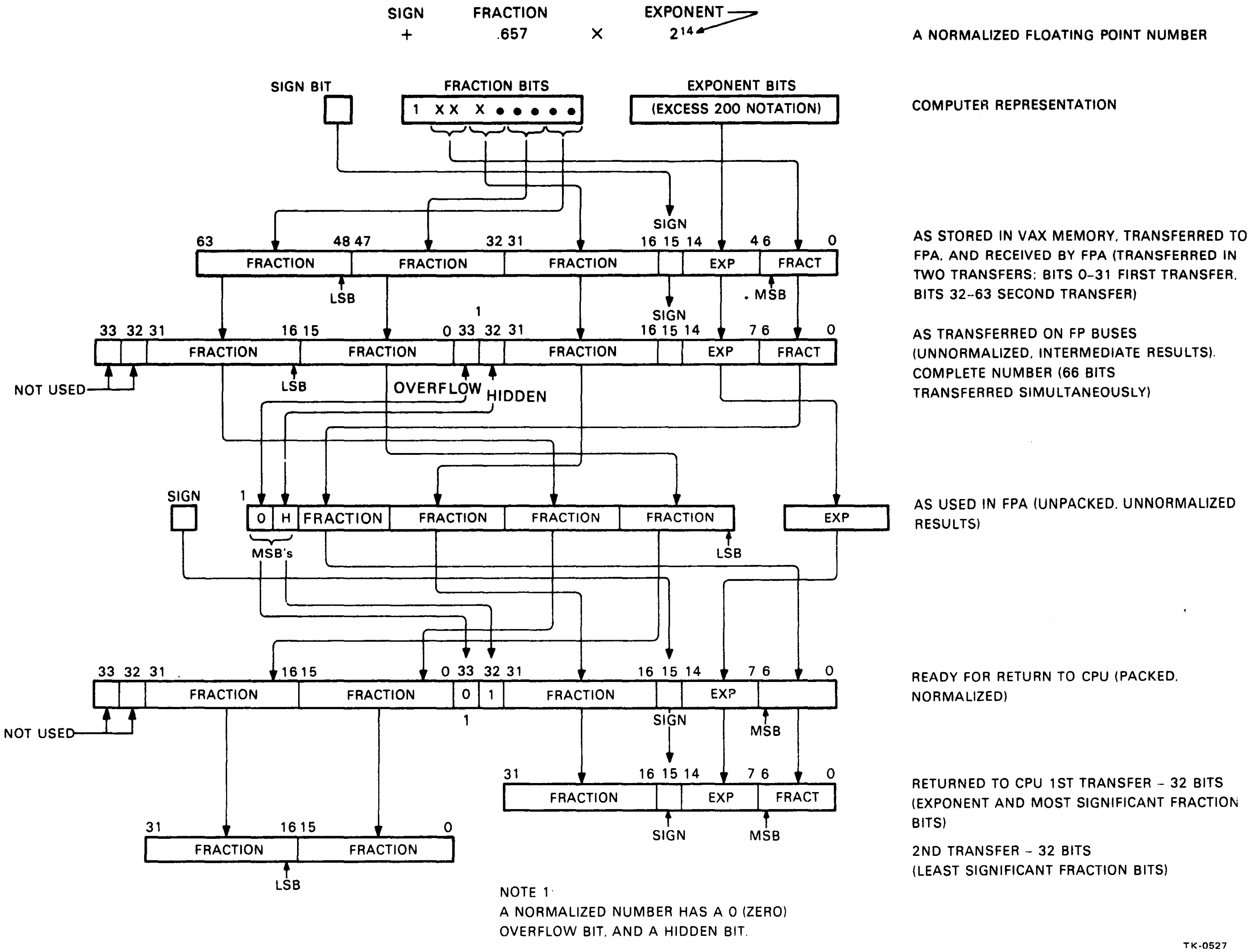
Floating-point numbers consist of sign bit, exponent bits, and fraction bits. A single precision floating-point number is stored in CPU memory as 4 contiguous bytes starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 31. The number is specified by its address A, the address of the byte containing bit 0 (Figure 2-1). The range of a single precision floating-point number is approximately $.29 \times 10^{-38}$ through 1.7×10^{38}. The precision is typically 7 decimal digits.

A double precision floating-point number is stored as 8 contiguous bytes. Bit labeling and addressing is similar to a single precision floating-point number. A double precision number has a range similar to a single precision, but its precision is about 16 decimal digits (Figure 2-1).



a. Single Precision

Figure 2-1 Floating-Point Format (Sheet 1 of 2)


SIGN
FRACTION
EXPONENT
+
.657
×
2^14
A NORMALIZED FLOATING POINT NUMBER
SIGN BIT
FRACTION BITS
1 XX X ● ● ● ● ●
EXPONENT BITS
(EXCESS 200 NOTATION)
COMPUTER REPRESENTATION
SIGN
63
48 47
32 31
16 15 14
4 6
0
FRACTION
FRACTION
FRACTION
EXP
FRACT
LSB
MSB
AS STORED IN VAX MEMORY, TRANSFERRED TO FPA, AND RECEIVED BY FPA (TRANSFERRED IN TWO TRANSFERS; BITS 0-31 FIRST TRANSFER, BITS 32-63 SECOND TRANSFER)
1
SIGN
33 32 31
16 15
0 33 32 31
16 15 14
7 6
0
FRACTION
FRACTION
FRACTION
EXP
FRACT
NOT USED
LSB
OVERFLOW
HIDDEN
AS TRANSFERRED ON FP BUSES (UNNORMALIZED, INTERMEDIATE RESULTS). COMPLETE NUMBER (66 BITS TRANSFERRED SIMULTANEOUSLY)
SIGN
1
0
H
FRACTION
FRACTION
FRACTION
FRACTION
EXP
MSB's
LSB
AS USED IN FPA (UNPACKED, UNNORMALIZED RESULTS)
33 32 31
16 15
0 33 32 31
16 15 14
7 6
0
FRACTION
FRACTION
0
1
FRACTION
EXP
NOT USED
1
SIGN
MSB
READY FOR RETURN TO CPU (PACKED, NORMALIZED)
31
16 15 14
7 6
0
FRACTION
EXP
FRACT
SIGN
MSB
RETURNED TO CPU 1ST TRANSFER - 32 BITS (EXPONENT AND MOST SIGNIFICANT FRACTION BITS)
31
16 15
0
FRACTION
FRACTION
LSB
2ND TRANSFER - 32 BITS (LEAST SIGNIFICANT FRACTION BITS)
NOTE 1
A NORMALIZED NUMBER HAS A 0 (ZERO) OVERFLOW BIT, AND A HIDDEN BIT.
TK-0527

b. Double Precision

Figure 2-1 Floating-Point Format (Sheet 2 of 2)

Floating-point numbers are transmitted to the FPA as packed, normalized numbers without a hidden or overflow bit. A single precision (float) number will have 24 fraction bits and a double precision number will have 56 fraction bits. Hardware in the FPA inserts and handles both the hidden and overflow bits. The number is split apart and used in various data manipulation units in the FPA. Although all operations begin with normalized operands, the intermediate results produced by the FPA data manipulation units can vary widely. Subtraction of nearly equal numbers can produce a number very close to zero. Addition and division can produce numbers close to 2. As a result intermediate results are transferred between data manipulation units as unnormalized numbers with both hidden and overflow bits. After the result is normalized, it is ready to return to the CPU. When the result is transmitted, it is transmitted as a packed, binary normalized number without hidden or overflow bits.

POLY uses specialized floating-point notation for intermediate results. In POLY, 7 additional bits are used for fraction addition. POLY execution consists of multiply, add, multiply, etc. To maintain maximum accuracy while functioning within the limitations of the FPA hardware, 7 additional LSBs are transferred from the fraction multiply (FMH + FML) hardware to the fraction add hardware (FAD). The 7 additional bits come from LSH <11:5> along FP bus A <14:08> into AR <06:00> (also called ARX). The FPA performs the add on the extended precision number, then transfers the addition result to the normalizer logic (FNM) where it is rounded, normalized, and held for the next part of the POLY instruction.

The EMOD instruction causes a 32 × 24 (64 × 56 for double) bit fraction multiplication to be performed in the FMH and FML. The extra 8 bits in the multiplicand are transferred over the ID bus to FP bus B line <07:00> to MCINT (also called MCX). MCINT <07:00> drives MCAND bus <07:00> for the fraction multiply. MPLIER is handled in the usual fashion. The result of the extended precision multiply is transferred to the CPU in one 32-bit transfer (F) or two 32-bit transfers (D).

2.1.2 Integer Numbers

The FPA handles a single integer format instruction, MULL (multiply longword). A longword is stored in CPU memory as 4 contiguous bytes starting on an arbitrary byte boundary. The FPA receives two 32-bit signed integers and multiplies them as unsigned integers to form a 64-bit product. The product, a 64-bit number, is returned to the CPU in two 32-bit transfers (low half first) for further processing. Refer to Figure 2-2 for summary of integer format.

2.1.3 Literals

The FPA handles float and double precision literal data. It receives the data from the CPU IB. Float literal data is transferred from the IB to the FPA's Literal Register (LR) using the ID bus. The FPA then loads the LR data into FPA internal registers and begins processing. The first half of double precision literal data is handled similarly. The second half comes from the CPU D-register via the ID bus and is loaded directly from the ID bus into the FPA internal registers.

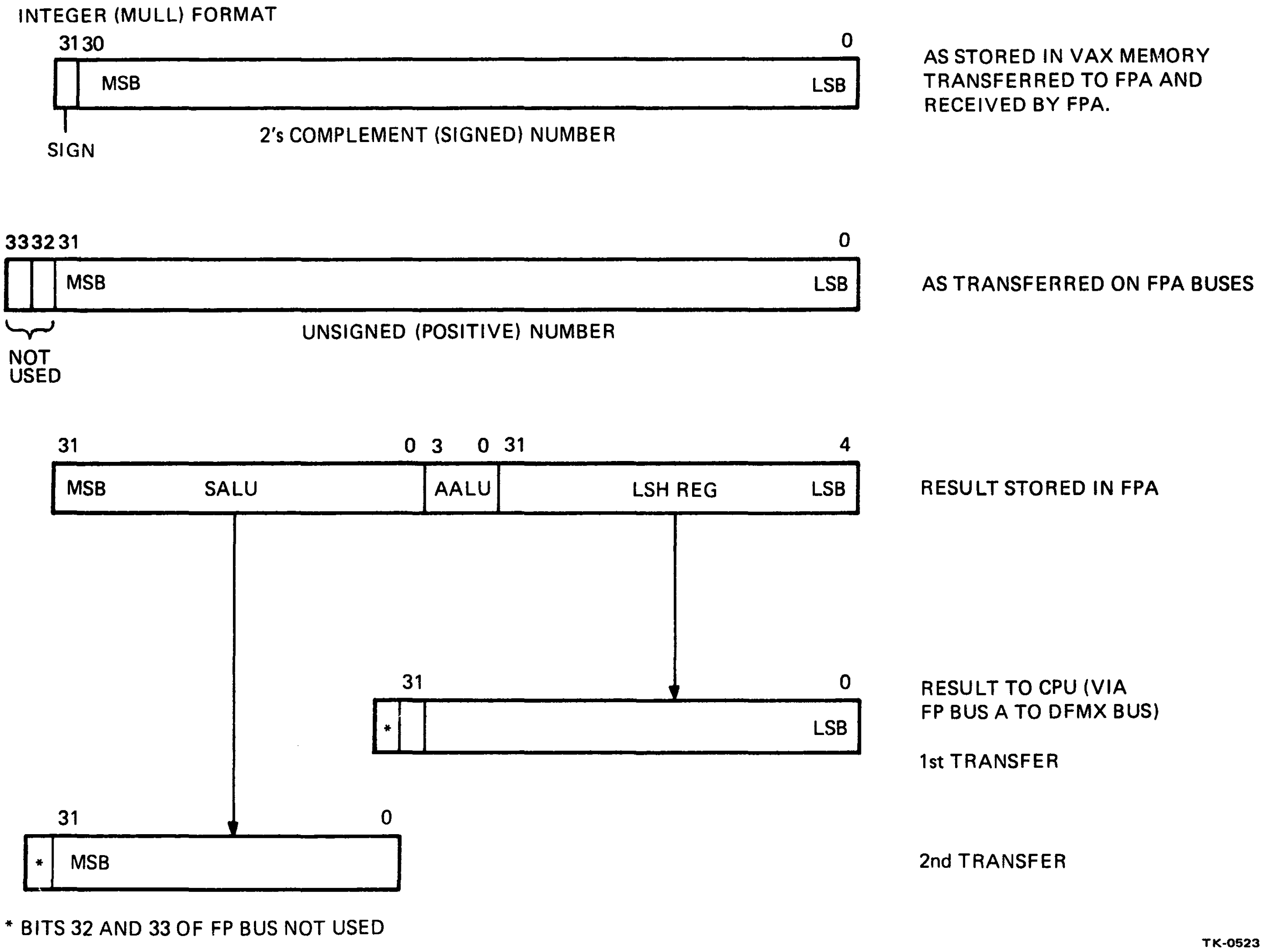
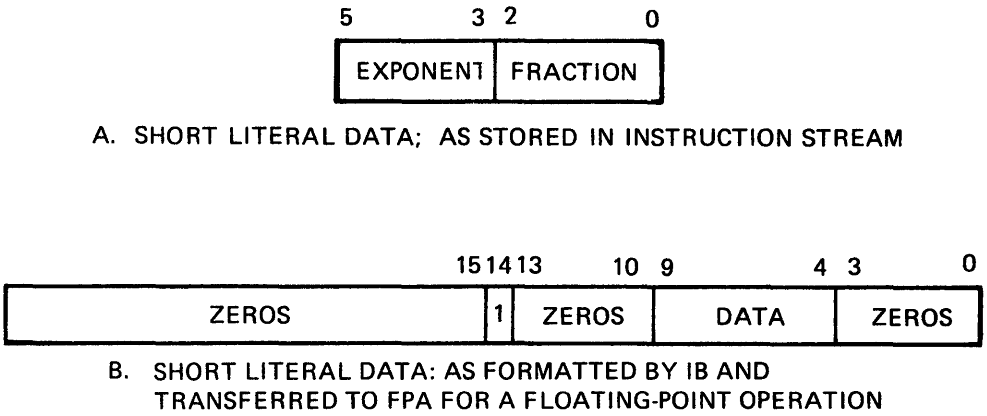

INTEGER (MULL) FORMAT
31
30
0
MSB
LSB
SIGN
2's COMPLEMENT (SIGNED) NUMBER
AS STORED IN VAX MEMORY TRANSFERRED TO FPA AND RECEIVED BY FPA.
33
32
31
0
MSB
LSB
NOT USED
UNSIGNED (POSITIVE) NUMBER
AS TRANSFERRED ON FPA BUSES
31
0
3
0
31
4
MSB
SALU
AALU
LSH REG
LSB
RESULT STORED IN FPA
31
0
*
LSB
RESULT TO CPU (VIA FP BUS A TO DFMX BUS)
1st TRANSFER
31
0
*
MSB
2nd TRANSFER
* BITS 32 AND 33 OF FP BUS NOT USED
TK-0523

Figure 2-2 Integer Format

The FPA handles short literals. Short literals contain only six data bits and are part of the instruction. The CPU formats the six data bits within the 32-bit data longword based on instruction type (floating-point or integer instruction.) If it is an integer instruction (the FPA handles only MULL), the six data bits are zero extended (26 zeros are added.) Any integer between 0 and 63_{10} can be written using a short literal. If it is a floating-point instruction, the short literal is assumed to contain three exponent bits and three fraction bits. The IB packs the data into standard FP format. This includes excess 80 notation for the exponent, a positive sign bit and a normalized fraction with a one hidden bit that is not stored. Refer to Figure 2-3 for FPA short literal format, and Table 2-1 for data that can be transferred using floating-point short literal form. Notice only positive numbers can be transferred.

If a double precision short literal is specified, the FPA accepts the first half and manufactures zeros to fill the second half.



A. SHORT LITERAL DATA; AS STORED IN INSTRUCTION STREAM

B. SHORT LITERAL DATA: AS FORMATTED BY IB AND TRANSFERRED TO FPA FOR A FLOATING-POINT OPERATION

TK-0519

Figure 2-3 Short Literal Format

Table 2-1 Floating Literals

Exponent	Fraction							
	0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1-1/8	1-1/4	1-3/8	1-1/2	1-5/8	1-3/4	1-7/8
2	2	2-1/4	2-1/2	2-3/4	3	3-1/4	3-1/2	3-3/4
3	4	4-1/2	5	5-1/2	6	6-1/2	7	7-1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

The FPA also handles long literals (32 or 64 data bits). Thirty-two bits, either a complete single precision transfer or the first half of a double precision, are transferred from the IB to the FPA LR. The second half of the double precision number is taken directly from the ID bus. Float and double precision floating-point data can be transferred using long literal format. The FPA also receives 32-bit integer data using the long literal format. (The FPA does not handle any 64-bit integer operands.)

2.1.4 Zero and Reserved Operand Codes

The FPA checks all data received for zeros and reserved operands during the fraction processing. Both zero and reserved operand function as codes transmitting special information. As discussed in Paragraph 1.4, the FPA assumes all floating-point numbers to be normalized numbers (between 1/2 and 1) with a hidden bit that is not stored. The hidden bit is normally inserted by data manipulation hardware. A zero cannot be represented as a normalized number and the hardware that inserts the hidden bit only increases the problem of representing and using zero. As a result, zero is represented by a code with zeros in the exponent bits (no excess 200 notation) and a clear sign bit. The fraction bits do not matter. Whenever this combination of bits is sensed, the FPA accesses special microcode that simulates the special properties of addition, subtraction, multiplication, and division with zero. Refer to Table 2-2 for the result of an operation with zero, and Figure 2-4 for the zero code.

Table 2-2 Zero Operand Microcode

Operation	Operand(s)	Operation Result
Add	0+X, X+0 0+0	X operand returned Zero returned*
Subtract	0–X X–0 0–0	–X returned X operand returned Zero returned
Multiply	0×0, X×0, 0×X	Zero returned*
Divide	0÷X (dividend is zero) X÷0 (divisor is zero; divide by zero)	Zero returned* Error condition†

* Zero code is returned, 0 in sign and exponent.

† FPA informs CPU that division by zero was attempted by asserting FPA error and PSL V bit and not asserting FP SYNC.

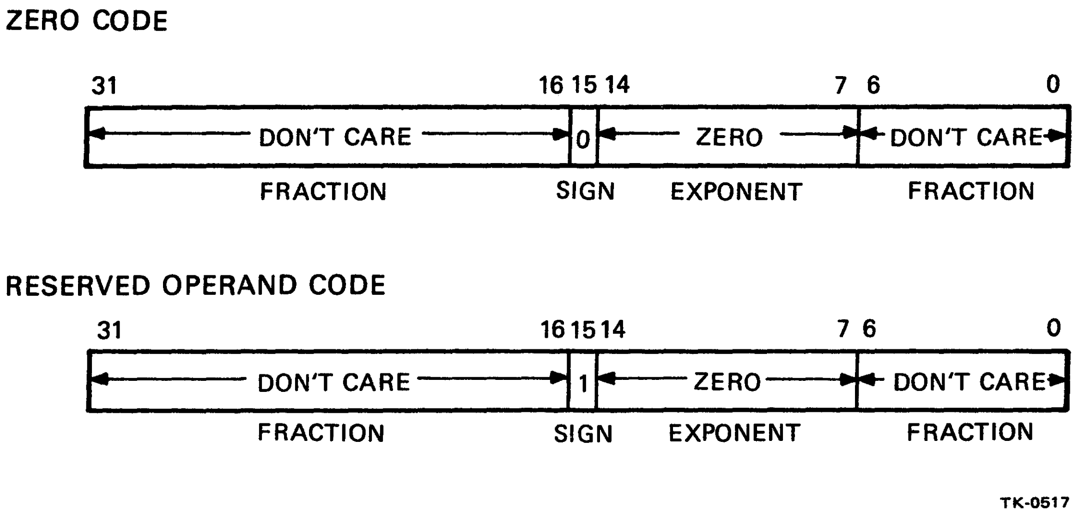


Figure 2-4 Zero and Reserved Operand Code

The code for reserved operand is zeros (cleared) in the exponent bits and a one (set) in the sign bit. One in the sign bit normally indicates a minus number so this sometimes called minus zero. A reserved operand indicates invalid data. It indicates data was accessed from a location that had not had data loaded into it, or a previous exception. Refer to Figure 2-4 for reserved operand code.

2.1.5 Hidden, Overflow and Guard Bits

The FPA uses extra fraction data bits during fraction manipulation to completely represent the fraction data, to handle result overflow, and to ensure accuracy of fraction result. Refer to Figure 2-5 for location of hidden, overflow, and guard bits.

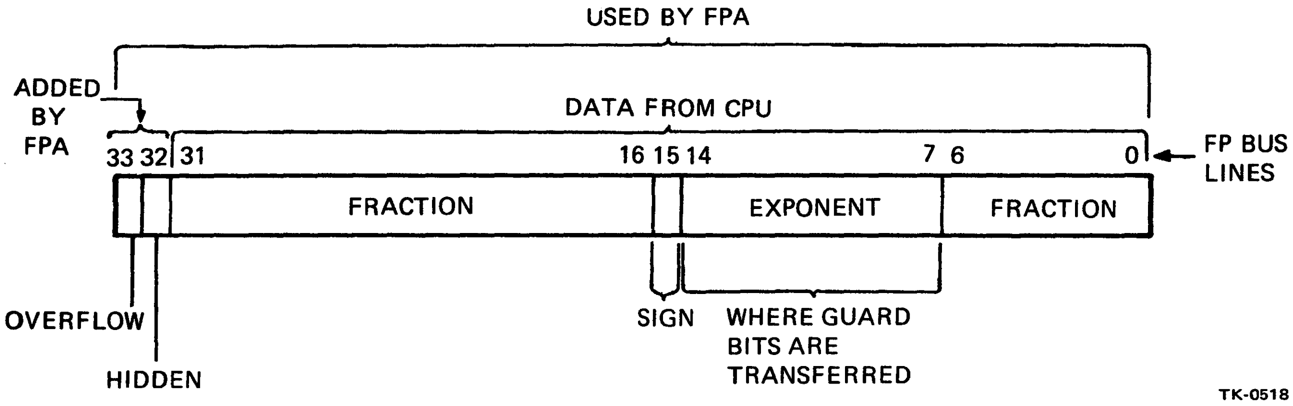


Figure 2-5 Hidden, Overflow, and Guard Bits

As discussed previously, the CPU stores floating-point numbers in a packed normalized form with the MSB of the fraction (called the hidden bit) not stored (since it is always a 1). The FPA receives the floating-point numbers in this form. To facilitate fraction calculation, logic on FNM adds the hidden bit to all CPU fraction data as it transported over the FP buses. The hidden bit is transmitted on FP bus (32). This means that all fraction data received by FPA fraction manipulation units have correct hidden bits.

The FPA also transmits an overflow bit between fraction manipulation units using FP bus (33). The overflow bit handles unnormalized intermediate fraction results. The combination (addition, subtraction, or division) of two normalized fractions can create a result greater than 1. The overflow bit enables the FPA to transmit this unnormalized result from the fraction computation units to the fraction normalizer logic (FNM).

To ensure accuracy of fractional results, the FPA data manipulation units add seven zeros called guard bits to the low order end of the fraction data they receive. This means a float fraction is 32-bits wide; a double, 64-bits wide. The POLY instruction loads extra data bits rather than zeros at the low order end of each coefficient fraction. The instruction also transfers additional low order data bits from the fraction multiply logic to the fraction add logic. These guard bits are dropped each time the POLY accumulation is normalized and rounded but they do ensure that the final answer is accurate. Without the guard bits, the right-shifting of a FP fraction to align radix points for addition and subtraction, or to normalize the result would lose the least significant bits off the right end of the shifted fraction. In some cases this loss would cause the last bit of the normalized result to be wrong. The guard bits prevent this. Guard bits are transmitted between FP data manipulation units using FP bus A (14:08). These lines normally transmit exponent data. This arrangement allows the FPA to maximize accuracy without additional hardware overhead.

2.1.6 Overflow, Underflow, Zero, and Reserved Operands

The FPA monitors all operands and results for exceptional conditions. When the FPA senses one or more of these conditions it informs the CPU via various bits and combinations of bits. Either one or both units begin special operations designed to minimize the effect of the condition. In some cases it stops the FPA's current operation and returns the FPA to the IRD state where all logic and registers are cleared in anticipation of a new FP instruction. The following paragraphs discuss these various unusual conditions. Table 2-3 summarizes the FPA and CPU operations caused by the unusual conditions.

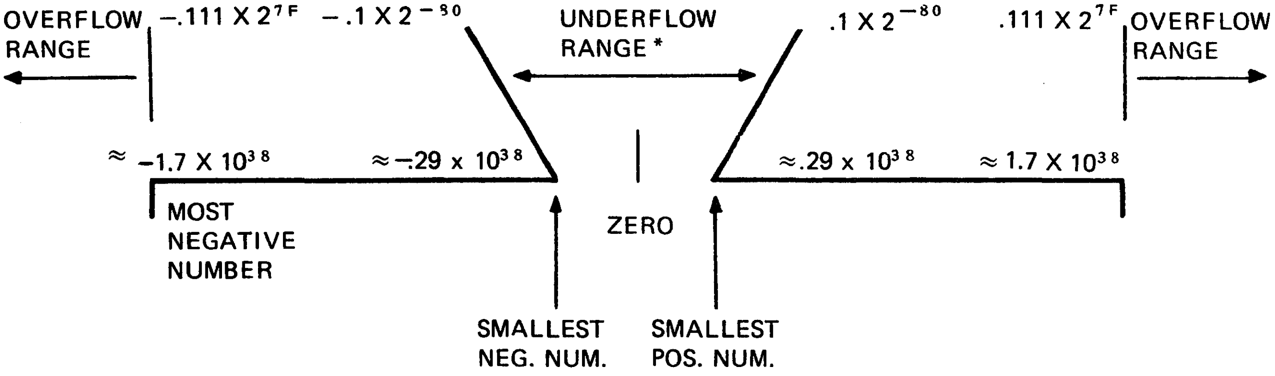
Table 2-3 Exception Conditions

Op Code	Exceptions Encountered		Result
	Zero Operand	Reserved Operand	
ADD, SUBT, MULT, EMOD	Microcode simulates arithmetic operation with zero (Table 2-2).	FPSYNC (ACC0) clear ERRSYNC (ACC1) set CPU traps FPA to IRD	All operations handle the occurrence of zero, underflow, and overflow results similarly.*
DIVIDE	ZERO DIVIDEND – Microcode returns zero as result	FPSYNC (ACC0) clear ERRSYNC (ACC1) set PSL V bit clear	ZERO – The zero code and FPSYNC are sent. PSL Z bit is set.
	ZERO DIVISOR – Divide by zero ERROR – FPSYNC (ACC0) clear ERRSYNC (ACC1) set PSL V bit set		UNDERFLOW – Zero code, FPSYNC, and ERRSYNC are sent. PSL Z is set. If PSL U (underflow) is set underflow causes a trap, otherwise operations continue.
	CPU differentiates between ZERO DIVISOR and RESERVED OPERAND by examining PSL V bit. In both cases, CPU traps FPA to IRD.		OVERFLOW – Reserved code, FPSYNC, and ERR SYNC are sent. PSL V is set. CPU traps FPA to IRD.
POLY*	POLY microcode simulates POLY operations with zero. (Table 2-2 and Paragraph 2.2.6).	FPSYNC (ACC0) set ERRSYNC (ACC1) set In STATUS REGISTER, minus ZERO ERROR bit set. CPU checks argument = RESERVED OPERAND. FPA checks coefficient = RESERVED OPERAND.	
MULL	No checking of MULL operands or results is performed by FPA software or hardware. Any combination of bits can be interpreted as an acceptable integer.		

* When POLY flows note a RESERVED OPERAND, UNDERFLOW, or OVERFLOW, both FPSYNC (ACC0) and ERRSYNC (ACC1) are set. CPU examines PSL and FPA STATUS REGISTER to determine exception condition. RESERVED OPERAND sets the MINUS ZERO ERROR bit. OVERFLOW sets the PSL V bit. UNDERFLOW sets PSL Z bit.

Overflow and Underflow

The FPA can handle a very large but bounded, range of numbers. Numbers too large (overflow) or too small (underflow) cannot be accurately handled (Figure 2-6). Special hardware monitors the results of all FPA operations for overflow and underflow conditions. The FPA checks for overflow and underflow by monitoring the exponent results. The monitoring is straightforward because of the excess 80 notation used. If the exponent with its excess 80 bias exceeds FF_{16} an overflow has occurred. If the exponent is less than 0, an underflow has occurred.



* EXACT ZERO DOES NOT CAUSE UNDERFLOW

TK-0521

Figure 2-6 Overflow and Underflow Ranges

If an overflow condition is sensed, the overflowed number is useless. The FPA manufactures a reserved operand and informs the CPU that an overflow occurred. The CPU notes the overflow and stores the reserved operand. The FPA returns to IRD.

Underflow is not as serious a problem. It merely indicates that the number is so small and so close to zero that the FPA cannot accurately represent it. If an underflow occurs the FPA sets the underflowed number to zero and informs the CPU that an underflow has occurred by asserting both FP SYNC and ERR SYN. It is important to inform the CPU that a zero has been returned because the CPU may at some later time attempt a division by the result (division by zero results in an error).

Zero

If a zero code is encountered in an operand transmitted to the FPA from the CPU, FPA microcode simulates the special properties of addition, subtraction, multiplication, and division with zero. Refer to Table 2-2 for the result of an operation with zero. If an exact zero is generated as a result of an FPA operation, the zero code is returned to the CPU and the condition code bits are set for a zero result. Zero can be generated in a normal arithmetic add or subtract operation (equal or equal-opposite operands) or in a microcode simulated arithmetic operation with a zero operand. An operation that generates an exact zero does not assert ERR SYN like an underflow operation (although both return a zero code).

Reserved Operand

Refer to Table 2-3 for the condition codes returned to the CPU when a reserved operand is encountered by the FPA.

2.2 INSTRUCTIONS AND ALGORITHMS

This section concentrates on the microcontrol used to carry out each FPA instruction. Each instruction accesses different microcontrol addresses to correctly move and load operands, compute intermediate results, and ready the final result for return to the CPU. Special instructions check for and handle errors and exceptional conditions.

This section details the data flow between hardware required to carry out the selected instruction. It only summarizes the hardware actions started once the data has been loaded by the microcontrol. Paragraph 2.3 contains a complete and detailed description of the hardware in each FPA section. Paragraph 2.2 and 2.3 complement each other and both should be read to thoroughly understand how the hardware implements each FPA instruction.

As stated before this section concentrates on data flow. Figure 2-7, FPA block diagram, shows the data bus interconnections and the various register in the FPA. Although this figure is not specifically referenced in the discussion it will help in understanding the data flow and should be referred to frequently.

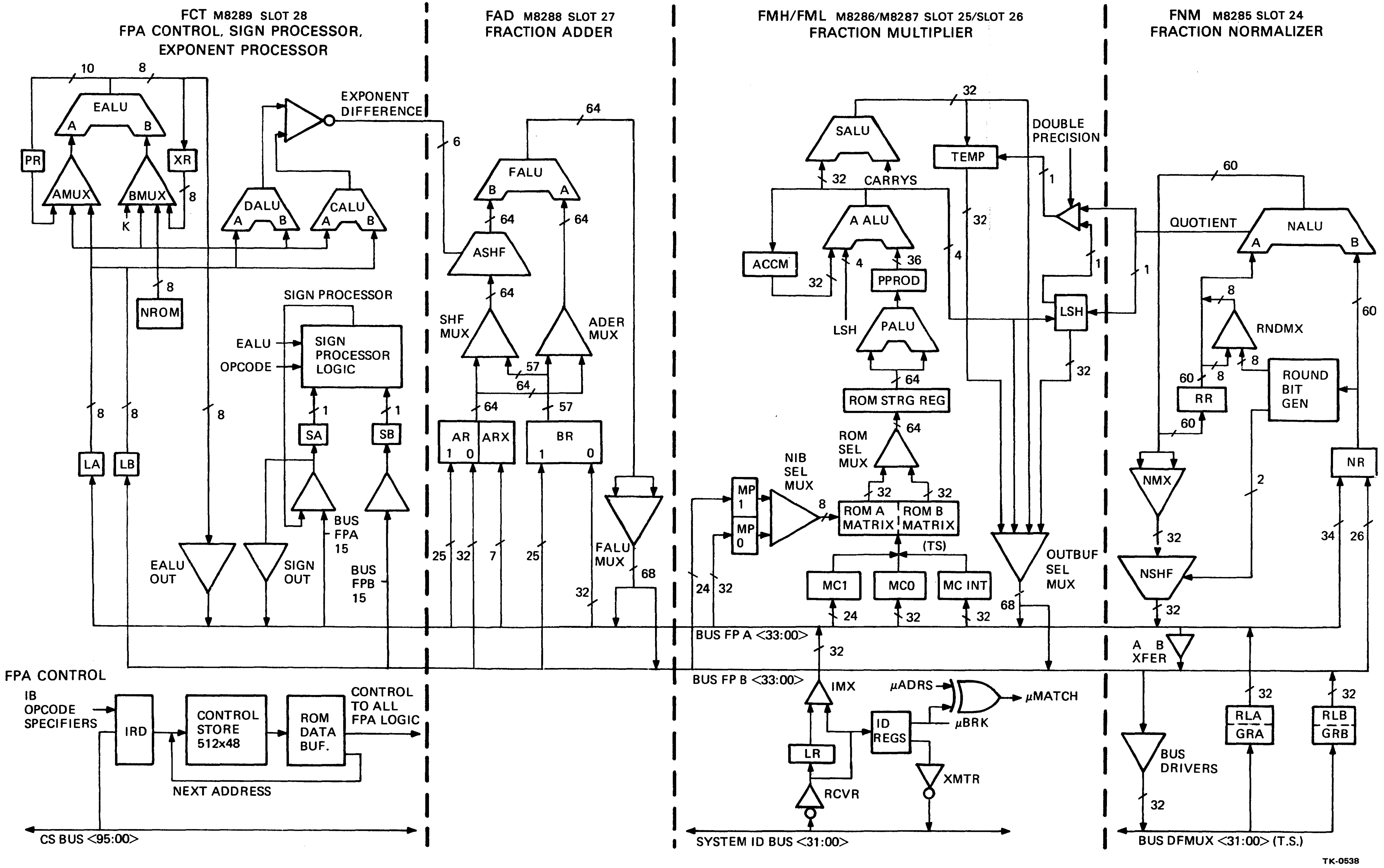


Figure 2-7 FPA Block Diagram

During IRD (instruction decode) the FPA performs some operations that are prerequisites to many FPA instructions. The FPA assumes a R-R float instruction and begins FPA register loading. The FPA has two copies of the CPU general registers. During IRD, it receives specifier information from the IB and accesses the register addresses contained. The contents of the first specifier is placed on FPA bus A, the content of the second on bus B.

The data on bus A is loaded in AR1, LA, SA, MC1, and MPO; bus B loads BR1, LB, SB, MP1, and MCI. AR1 and BR1 are fraction registers used for the addition and subtraction of floating-point numbers. LA and LB are loaded with the exponents of the numbers and immediately the hardware begins an exponent difference calculation. The exponent difference and/or which exponent is larger is needed for floating-point additions, subtractions, and multiplications. SA and SB are input registers for the sign-processing hardware. Fraction data from specifier 1 (on bus A) is loaded into multiply registers, MC1 (multiplicand) and MPO (multiplier). Fraction data from specifier 2 (on bus B) is loaded into MP1 (multiplier) and MCI (multiplicand-integer). MC1 and MP1 hold operand data for MULF and EMODF instructions. The hardware multiply begins the MULF or EMODF fraction multiply operation during IRD using MC1 and MP1. MCI and MPO contain the operand for a MULL instruction.

During IRD, numerous FPA instructions have been started. If the instruction is a float register-to-register, both operands are already loaded and ready in the FPA. Exponent manipulations needed for add, subtract, and multiply operations have started. MULF and EMODF fraction multiplication have started. If the instruction decoded is a MULL, the multiplier and multiplicand have already been loaded into the proper registers.

2.2.1 Add/Subtract

The FPA add/subtract operations can be broken into three states:

1. Load
2. Add/Subtract
3. Normalize.

2.2.1.1 Load – While the FPA is in IRD, it is setting up for a float, R-R operation. This means that specifiers 1 and 2 from the instruction buffer are being placed on FP buses A and B, respectively. Bus A loads AR1 (fraction register), LA (exponent register) and SA (sign latch). Bus B loads BR1, LB, and SB.

When the FPA decodes a floating-point instruction, it enters A-Fork and selects a microword address based on op code and specifier types. If the instruction is a float R-R A/S, the FPA enters the optimized add/subtract execution state immediately. If, however, it is not, the FPA, under-control of the selected microword, receives and stores the required data during A-Fork and possibly B-Fork flows. If it is double-precision, 32 additional fraction bits are loaded into both AR0 (extension of AR1) and BR0 (extension of BR1.) If it is not an R-R operation, the new data from the correct source is loaded into AR1, LA, SA, BR1, LB, and SB.

As tne final correct operands are loaded, whether during IRD (in the case of float R-R operations) or during some following microcontrol state in A-Fork or B-Fork, the exponent difference of the two operands is determined by comparing LA and LB in DALU and CALU. Based on the exponent difference, the fraction associated with the smaller exponent is loaded into SHMX and right-shifted by ASHR until the radix points align. This happens before entering the add/subtract state.

2.2.1.2 Add/Subtract – In this state, the fractional result is computed. Based on the op codes, signs of the operands, and exponent difference, FALU operation is selected. Normally, the FALU adds or subtracts the already aligned fractions for the fractional result. Refer to Table 2-4 for normal FALU operation, and Table 2-5 for special FAD operation criterion.

Table 2-4 FALU Operation

Op Code	Operand Sign	FALU Operation
ADD	Same	Add
ADD	Diff	Subtract
SUBT	Same	Subtract
SUBT	Diff	Add

Table 2-5 Combination of Conditions Initializing Special FAD Operation

FALU Subtract	Exponent Diff	Op Code	Precision
Yes	Greater than 7	X	D
Yes	Greater than 1	POLY	D
Yes	Less than 2	POLY	X

X = Don't care

The special FAD operation is used to ensure maximum accuracy in the result while operating within the FPA hardware constraints. The special FAD operation involves complementing the fraction associated with the smaller exponent by subtracting the fraction from zero in the FAD, returning the complemented number to the fraction register (either AR or BR) it was in originally, and then loading it into SHFMX and right-shifting and sign-extending based on exponent difference until the radix points align. This special operation takes an extra microstep but ensures maximum accuracy. As a result, the actual fraction subtraction to produce the result does not take place until this third state.

During the add/subtract state, the larger exponent is transferred to the PR.

2.2.1.3 Normalize - In this state, the answer is readied for return to the main machine. This involves final normalization of the fraction, adjustment of the exponent and determination of the resultant sign. If the calculation involved special FAD operations as discussed in the previous paragraph, the fraction subtraction will first be carried out and then the result will be readied for return to the main machine.

When entering the normalization flows, the FPA checks three conditions:

1. Exponents equal zero
2. FALU subtract with exponent difference less than two
3. Subtract, exponent difference less than 7, and DP.

If a zero operand is noted, the other (non-zero) operand is transferred to the output and if it is the subtrahend in a FALU subtraction, the sign is complemented (minuend - subtrahend = remainder; 0 - X = -X). A FALU subtraction with exponent difference of 1 or 0 initiates special flows because the subtraction of two nearly equal numbers can result in a very small fraction (numerous leading zeros) which might require many shifts before the first significant bit is located. The special flow initiated can shift the result up to sixty places to find the first signficant bit before it is transferred to the standard normalize routine. If a first significant bit is not found after 60 bits have been shifted, a zero is readied as a result. If the third branch is taken, the addition state described in Paragraph 2.2.1.2 results, then flow reenters the normalization routine.

Usually, the unnormalized result requires a shift of four places or less. If this is the case, the four MSBs are examined to locate the first significant bit. Based on the location of the first significant bit, a rounding byte is added to the fraction. If the result from a FALU subtraction is negative, the FALU result is subtracted from the rounding byte to return the number to sign magnitude notation and round it in a single step. Once the FALU result is added to or subtracted from the rounding byte, the fraction is shifted and least significant bits are dropped.

In all cases, the number of shifts required to ready the fraction for return to the CPU is computed and is used to adjust the exponent in the PR. Once completed, the exponent, the normalized fraction, and the sign of the result are placed on the FP bus A. When the complete result is on the bus, standard routines handle the actual transfer to the main machine.

2.2.2 Multiply (Floating-Point)

The FPA multiply operation can be broken into three operations: load, multiply, and normalize. In the process of carrying out a FP multiply, the FPA receives the operands (each consisting of an exponent, fraction, and sign bits), checks for zeros and reserved operands; loads the exponent, fraction, and sign bits into the appropriate registers; starts the hardware to carry out the required calculations; and assembles and readies the result for return to the CPU when notified that the hardware calculation is finished.

2.2.2.1 Load - To maximize speed, the FPA is continuously setting up for a float R-R operation. This means that in IRD specifiers, 1 and 2 from the instruction buffer are addressing the GPRs (general-purpose register) in the CPU, and the register data is being placed on FP buses A and B, respectively. Bus A loads MC1 (multiplicand register), LA (exponent register) and SA (sign latch.) Bus B loads MP1 (multiplier register), LB, and SB.

When the FPA decodes a floating-point instruction, it enters A-Fork and branches to a specific microword based on op code and specifier types. If the instruction is a float R-R multiply, the operands are already loaded and the FPA enters the multiply state immediately. If, however, it is not, the FPA, under control of the selected microword receives and stores the required data during A-Fork and possibly B-Fork flows. If it is a double-precision multiply, 32 additional fraction bits are loaded into both MC0 (extension of MC1) and MP0 (extension of MP1.) If one or both of the specifiers are not registers, all new data will be loaded into MC1, LA, SA, MP1, LB, and SB.

As the final correct operands are loaded, whether during IRD (in the case of float R-R operations) or during some following microcontrol state, the fraction multiplier begins the fraction multiply by breaking the fractions into nibbles and beginning the hardware multiplication using the first multiplier nibble.

2.2.2.2 Multiply - In the multiply state, the fraction multiplication continues until a final fraction (as yet unnormalized) is computed, the exponents are added, and the sign of the result is computed. The fraction multiplication is initiated when the multiply flows issue MCONT (multiply continue.)

As MCONT is issued, the FPA checks for operands equal to zero or minus zero (reserved operand.) If a zero operand is found, computation stops and the FPA immediately returns a zero to the base machine. If a reserved operand is found, the operation aborts. If neither are found, computation continues. In the case of a float (single-precision) multiply, the fraction multiplication is completed as the exponent calculation is completed. The product is transferred to the NR. In a double-precision multiply, the microcontrol enters a wait state. While waiting during a double-precision multiply, the FPA continually transfers the output of the fraction multiplier to the normalizer. This enables the FPA to begin normalizing the fraction result as soon as the multiplication is complete. It remains in the wait state until a hardware counter in the fraction multiply logic asserts MUL/DIV DONE indicating the fraction multiply is complete.

While the fraction multiply and the check for zeros and reserved operands is taking place, the exponents are added If no zeros or reserved operands are found, the fraction multiply and exponent processing continues. After the exponents are added, a bias of 200_8 or 80_{16} is subtracted from the exponent result to return the exponent to excess 80 notation (refer to Paragraph 1.5).

In a multiply operation, the sign of the result is the exclusive-OR of the operand signs.

By the time the fraction multiply is complete, the exponents have been added, and exponent bias subtracted, and the sign of the result has been calculated. The result of the fraction multiply is moved to NR.

2.2.2.3 Normalize – The normalize state of a floating-point multiply is very simple. Since the input operands are always between 1/2 and 1, the result is always between 1/4 and 1. This means that the result can be normalized with a single shift of four bits, or less. In the normalize state, the fraction is rounded and shifted, and the exponent is adjusted to reflect the normalization shift. The normalized fraction, adjusted exponent, and sign bit are placed on the FP bus A. Once the complete result is on the bus, standard routines handle the actual data transfers to the main machine.

2.2.3 MULL (Multiply Integer Longword)

The FPA's MULL algorithm is the simplest and most straightforward of all the operation flows. The FPA receives two 32-bit signed integers, performs an unsigned multiplication, and returns the 64-bit answer to the base machine. The FPA performs no result normalization, no checks for reserved operands, zero operands, or other error conditions. Microcode in the base machine generates the condition codes and handles all the checks and manipulations required to ensure a correct result.

2.2.3.1 Load – As discussed in introductory Paragraph 2.2, the FPA during IRD loads MP0 and MCI (the two registers used in MULL operations) with the register contents of specifier 1 and 2, respectively. If the instruction decoded in the A-Fork flows is a R-R MULL, the FPA can begin the multiply immediately. If it is a MULL but not an R-R, the FPA will, under the control of the selected microaddress, load data from the correct source into either or both MP0 and MCI.

2.2.3.2 Multiply and Return – The decoding of a MULL causes the fraction multiply hardware to abandon set-up of a MULF and begin accessing the registers used for MULL (MCI and MP0.) When the proper data has been loaded, MCONT is issued by the FPA. This indicates to the fraction multiply hardware that the correct data is in MP0 and MCI, and that the data accesses started previously were accessing correct data.

MCONT enables the fraction multiply hardware to continue multiplying. The multiply continues, controlled by a hardware sequencer within fraction multiply hardware, while the FPA waits two machine cycles. The answer accumulates in ACCM and LSH. After two wait cycles, the multiply is finished. The hardware stops and the FPA makes the 32 low-order bits (from LSH) available to the CPU. When the CPU responds with CPSYNC, indicating the low-order bits have been stored, the FPA readies the high 32 bits from SALU for transmission to the CPU.

2.2.4 Divide

The FPA divide operation can be broken into three steps: load, divide, and normalize. To do a floating-point divide, the FPA receives the operands (each consisting of sign, fraction, and exponent bits), loads the operands into holding registers, tranfers the operands from the holding registers into the correct division registers, starts the hardware to do the fraction division, checks for zero and reserved operands, starts the hardware to store the result, and normalizes and packs the result for return to the CPU.

2.2.4.1 Load - The loading of division operands takes place in two substeps: data fetch, and division register load. Unlike the FPA add/subtract, multiply, and MULL operations, the FPA does not load division operands into the proper division registers during IRD (Table 2-6).

Table 2-6 The Division Load

	Specifier 1	**Specifier 2**
IRD	Register and float assumed (divisor) Register data to AR1, LA, SB	Register and float assumed (dividend). Register data to BR1, LB, SB
Data Fetch Substep	Op code decoded, specifiers and precision known	
	New data loaded into AR1 and AR0*, LA, and SA, if needed.	New data loaded into BR1 and BR0*, LA, and SB, if needed.
Division Register Load Substep 2 microwords	1st Microword - move LA (divisor exponent) to XR.	Move BR (divident fraction) to NR.
	2nd Microword - move AR (divisor fraction) to just vacated NR.	Move NR (dividend fraction) to RR and right shifts the just loaded divident fraction to compensate for RR's hard wired left shift. This right shift ensures initial dividend is properly represented.
	Subtract XR (divisor exponent) from LB (divident exponent).	

*AR0 and BR0 are fraction extension registers for double precision operations.

During IRD a R-R float operand is assumed. This means that both specifier 1 and 2 are assumed to be registers. The contents of the first register named is placed in AR, LA, and SA, the content of the second in BR, LB, and SB. If the operation decode is a R-R float divide, the data fetch substep is done and division register load may begin.

However, if it is not an R-R float, divide microcode waits for data from the correct specifier and loads it into either AR1, LA, and SA; and/or BR, LB, and SB. When the divisor is in AR, LA, and SA, and the dividend is in BR, LB, and SB; the data fetch substep is finished.

The division register load substep loads the divisor's and the dividend's fraction and exponent components into the registers required to do a division. The loading of the proper registers takes two microcode steps. The first microcode step loads the divisor exponent into XR and loads the dividend fraction into the NR. The second microcode step finishes the register loading by moving dividend fraction (in the NR) to the RR and loading the just vacated NR with the divisor fraction from the AR. It also starts the fraction division hardware, checks for zeros and reserved operands, and subtracts the divisor exponent (XR) from the dividend exponent (LB) (LB - XR).

2.2.4.2 Divide - The divide operation continues unless a zero, or reserved operand is found. If a zero dividend is found, operations cease and a zero is readied for return to the CPU. Finding a zero divisor or a reserved operand initiates error states. The FPA will remain in these error states until returned to IRD by a CPU signal.

If no zeros or reserved operands are found, the division continues. A bias 80 is added to the result of the exponent subtraction to return it to excess 80 notation (Paragraph 1.5.) The fraction multiply hardware is started. This hardware is used to store the result of the fraction division as it is generated. The division continues under hardware control as the FPA microcode remains in a divide wait loop.

The hardware uses the restoring, repeated subtraction technique to divide. The dividend is initially loaded into the RR and the divisor is stored in the NR. The divisor (contents of NR) is subtracted from the dividend (contents of RR). If the result is negative, a zero is left-shifted into result register in the fraction multiply hardware and the contents of the RR is left-shifted by one. If the result is positive or zero, a 1 is left-shifted into the result register, and the result is loaded into the remainder register left shifted by one. The divisor (contents of NR) is continually subtracted from the contents of the RR until 26 bits (58 bits for double precision) of quotient are generated. MUL/DIV DONE is now asserted.

Asserting MUL/DIV DONE stops the division and ends the divide wait loop. The divide result is transferred from the fraction multiply hardware where it was stored during generation to the normalize register (NR) in the normalize hardware.

2.2.4.3 Normalize - Since the two initial operands are normalized (between 1/2 and 1), the result is always positive and between 1/2 and 2. This means the normalize and round operation is simple and will take only one microstep. The result is examined, a round byte is selected and added, and the data is shifted as needed to produce a normalized result. The exponent result is adjusted to reflect the direction and amount of the fraction shift. The normalized fraction, adjusted exponent, and sign bit are placed on the FP bus(es). Once the result is on the bus(es), standard storage routines handle the actual transfer to the CPU.

2.2.5 EMOD (Extended Precision Multiply and Integerize)

The EMOD operation is partially done in the FPA. The FPA performs an unsigned 32 × 24-bit (64 × 56-bit for double precision) multiplication and returns the fraction result to the main machine. The main machine does all further processing. The FPA EMOD operation can be broken into two steps: operand load, and result calculation and return.

2.2.5.1 Operand Load - Loading the EMOD operands involves loading the multiplicand, an 8-bit multiplicand extension, and the multiplier into proper registers. The multiplicand (either single or double precision) is loaded into MC during A-Fork. In B-Fork, EMOD flows are started. These flows wait for the CPU to fetch the multiplicand extension (8 bits) and transmit it to the FPA via the ID bus. The FPA loads the extension into MCX which is part of the MCI register. The second operand is then transmitted to the FPA and loaded into appropriate multiplier register MP0 and MP1. The multiplier is not extended. The FPA receives and stores the exponent and sign associated with both operands but does not use them.

2.2.5.2 Result Calculation and Return - Once the operands are loaded, MCONT is asserted and the FMOD multiply begins. The operands are tested for zeros or reserved operands. If zeros are found, special flows stop the multiply and return a zero to the CPU. Finding reserved operands initiates error flows. If no exceptions are found, the multiply sequencer, started by MCONT asserted, continues multiplying. A single precision (float) multiply is finished in one microstep after the exponent test. A double precision multiply causes the FPA to enter a wait loop. It remains in the wait loop until the multiply sequencer asserts MUL/DIV DONE indicating the result is computed.

When the result computation is finished, the fraction (32-bit float, 64-bits double) is transmitted to the CPU. The CPU does all further processing including sign computation, removal of the integer part, normalization, and exponent calculation.

2.2.6 POLY (Polynomial Evaluation)

2.2.6.1 Introduction - POLY is an FPA implemented instruction. The FPA does the majority of calculations required to evaluate a polynomial expression. This involves storing a constant, and an accumulation; receiving coefficients; repeated additions and multiplications using the constant, the accumulation, and the new coefficient, and the readying of a final result to be returned to the CPU. It also uses specialized operations (both hardware and microcode) to ensure maximum accuracy within the FPA hardware limits.

The following paragraphs explain POLY flows, polynomial expression and define various terms, and POLY exceptions in detail. Also discussed are the numerous flows required to handle errors, underflows, overflows, and zeros.

2.2.6.2 The Polynomial Expression - The generalized polynomial may be written:

$$f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n.$$

The x, a constant within each polynomial, is called the argument and is raised to various powers: x^1, x^2, x^3, ..., x^n. The highest power represented here by n superscript is called the degree of the equation. The $a_0, a_1, a_2, ..., a_n$ are the coefficients. Rearrangement and factoring produces $f(x) = a_0 + x(a_1 + x (a_2 + ... + x(a_{n-1} + xa_n)))$. The result, f(x), may be computed: a_n times x then add a_{n-1} ; the resultant answer times x and then add a_{n-2} ... The generalized form is: (accumulation times x) plus the new coefficient, a_i, equals the new accumulation.

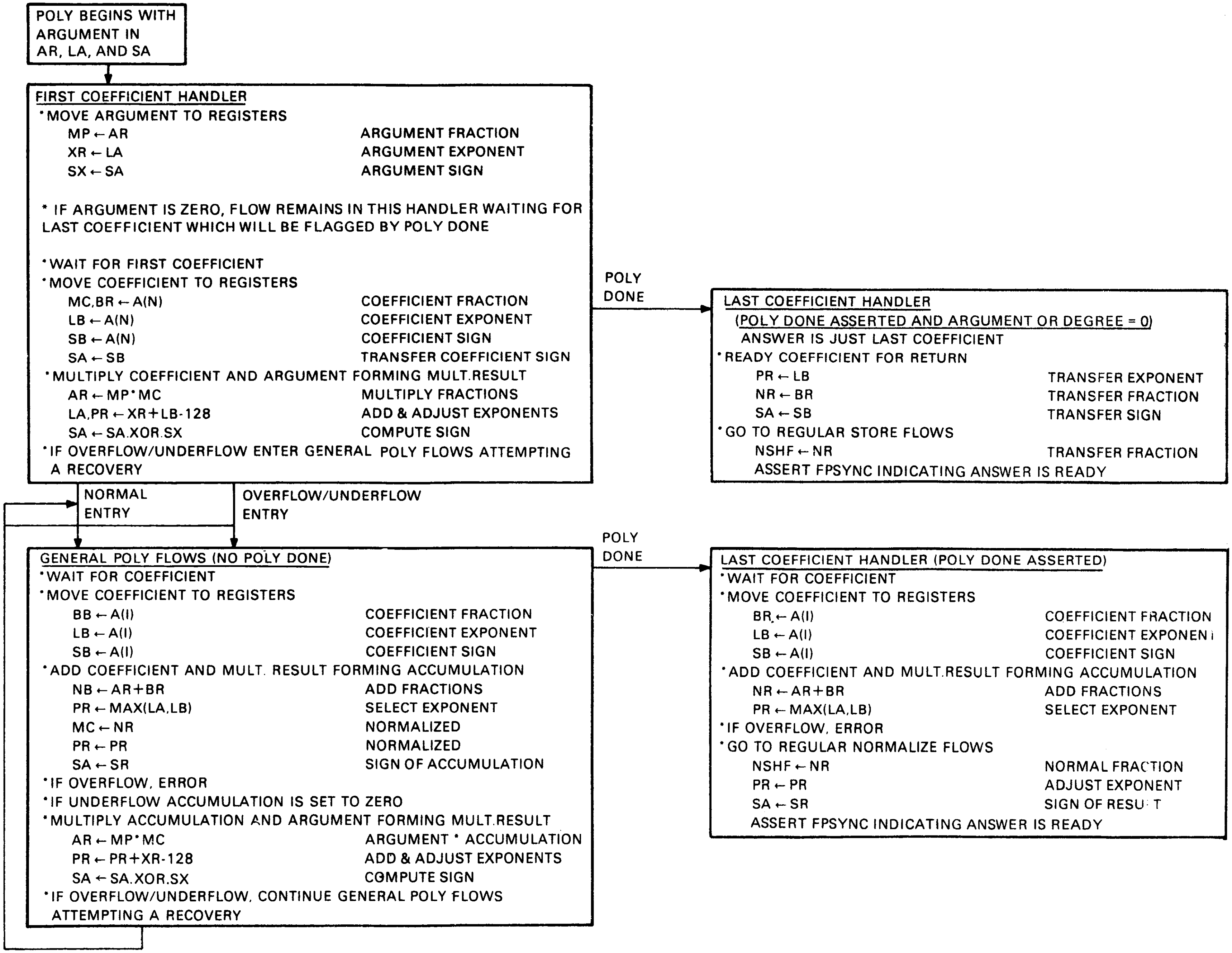
The POLY instruction format is POLY argument, degree, coefficients table. The FPA receives and stores the argument. The CPU uses the degree operand to determine when it has accessed the last coefficient of the table so it may inform the FPA that the POLY calculation is done. The coefficient table is arranged in a_n, a_{n-1}, a_{n-2}, ..., a_1, and a_0 order. The CPU transmits the coefficients to the FPA as needed: a_n first, a_{n-1} next, ...

2.2.6.3 Normal POLY Flows - The FPA begins special POLY flows in B-Fork. The POLY argument is transferred to the FPA during A-Fork and then loaded into the argument registers. The argument fraction is loaded into MP, the exponent into XR, and the sign is SX. The argument remains in these registers throughout POLY execution. The FPA waits for the first coefficient to be sent so the POLY computation can begin.

POLY computation can be divided into three large categories:

1. Argument and First Coefficient Handler
2. Generalized POLY Computation (neither first term or last term)
3. POLY DONE Handler (handles A_0, the last coefficient).

This section will discuss the flow by these three categories. Within each category, microcode controls the normal operations, checks for exceptional conditions, and attempts to recover from any exceptional conditions. Refer to Figure 2-8 for a summary of the POLY flow.


POLY BEGINS WITH ARGUMENT IN AR, LA, AND SA
FIRST COEFFICIENT HANDLER
•MOVE ARGUMENT TO REGISTERS
MP ← AR ARGUMENT FRACTION
XR ← LA ARGUMENT EXPONENT
SX ← SA ARGUMENT SIGN
* IF ARGUMENT IS ZERO, FLOW REMAINS IN THIS HANDLER WAITING FOR LAST COEFFICIENT WHICH WILL BE FLAGGED BY POLY DONE
•WAIT FOR FIRST COEFFICIENT
•MOVE COEFFICIENT TO REGISTERS
MC,BR ← A(N) COEFFICIENT FRACTION
LB ← A(N) COEFFICIENT EXPONENT
SB ← A(N) COEFFICIENT SIGN
SA ← SB TRANSFER COEFFICIENT SIGN
•MULTIPLY COEFFICIENT AND ARGUMENT FORMING MULT.RESULT
AR ← MP*MC MULTIPLY FRACTIONS
LA,PR ← XR+LB-128 ADD & ADJUST EXPONENTS
SA ← SA.XOR.SX COMPUTE SIGN
•IF OVERFLOW/UNDERFLOW ENTER GENERAL POLY FLOWS ATTEMPTING A RECOVERY
POLY DONE
LAST COEFFICIENT HANDLER
(POLY DONE ASSERTED AND ARGUMENT OR DEGREE = 0)
ANSWER IS JUST LAST COEFFICIENT
•READY COEFFICIENT FOR RETURN
PR ← LB TRANSFER EXPONENT
NR ← BR TRANSFER FRACTION
SA ← SB TRANSFER SIGN
•GO TO REGULAR STORE FLOWS
NSHF ← NR TRANSFER FRACTION
ASSERT FPSYNC INDICATING ANSWER IS READY
NORMAL ENTRY
OVERFLOW/UNDERFLOW ENTRY
GENERAL POLY FLOWS (NO POLY DONE)
•WAIT FOR COEFFICIENT
•MOVE COEFFICIENT TO REGISTERS
BB ← A(I) COEFFICIENT FRACTION
LB ← A(I) COEFFICIENT EXPONENT
SB ← A(I) COEFFICIENT SIGN
•ADD COEFFICIENT AND MULT. RESULT FORMING ACCUMULATION
NB ← AR+BR ADD FRACTIONS
PR ← MAX(LA,LB) SELECT EXPONENT
MC ← NR NORMALIZED
PR ← PR NORMALIZED
SA ← SR SIGN OF ACCUMULATION
•IF OVERFLOW, ERROR
•IF UNDERFLOW ACCUMULATION IS SET TO ZERO
•MULTIPLY ACCUMULATION AND ARGUMENT FORMING MULT.RESULT
AR ← MP*MC ARGUMENT * ACCUMULATION
PR ← PR+XR-128 ADD & ADJUST EXPONENTS
SA ← SA.XOR.SX COMPUTE SIGN
•IF OVERFLOW/UNDERFLOW, CONTINUE GENERAL POLY FLOWS ATTEMPTING A RECOVERY
POLY DONE
LAST COEFFICIENT HANDLER (POLY DONE ASSERTED)
•WAIT FOR COEFFICIENT
•MOVE COEFFICIENT TO REGISTERS
BR ← A(I) COEFFICIENT FRACTION
LB ← A(I) COEFFICIENT EXPONENT
SB ← A(I) COEFFICIENT SIGN
•ADD COEFFICIENT AND MULT.RESULT FORMING ACCUMULATION
NR ← AR+BR ADD FRACTIONS
PR ← MAX(LA,LB) SELECT EXPONENT
•IF OVERFLOW, ERROR
•GO TO REGULAR NORMALIZE FLOWS
NSHF ← NR NORMAL FRACTION
PR ← PR ADJUST EXPONENT
SA ← SR SIGN OF RESULT
ASSERT FPSYNC INDICATING ANSWER IS READY

TK-0526

Figure 2-8 The POLY Flow

Within the flows different microcode handles float and double precision operation. In POLY double coefficient, argument, and accumulation fractions each use an additional 32 low-order bits. The differences between float and double precision are not discussed in each operation because it is normally limited to longer fraction multiply times and slower fraction transfers. These come about because there are more bits to be multiplied and moved.

When the first coefficient, A_0, is sent, it is loaded in MC, LB, and SB. Since the argument has not yet been checked, both the argument and the coefficient are checked for exception conditions and POLY DONE is checked. If any exception condition is noted, special flows are accessed. POLY DONE asserted indicates that the coefficient just sent was the final coefficient (in this case, the first coefficient is also the last coefficient). If the argument (x) is zero, all terms except the A_0 term of the polynomial will be zero. Either POLY DONE asserted or x equals zero causes the FPA to access a special last coefficient routine in the argument and first coefficient handler that returns A_0 to the CPU as the result of the polynomial calculation.

After both the argument and the coefficient are checked and no exception conditions are found, the first multiply takes place. While the fractions are multiplied in the fraction multiply logic (FML and FMH), the exponents are added and adjusted to return the excess 80 notation (FCT) and the sign of the result is computed (FCT). When the multiply is done, the fraction is moved to AR for the addition operation. To maximize calculation accuracy, no normalization is performed after the multiplication and 8 additional low-order fraction bits are transferred to the AR register and stored in ARX. These 8 bits are used when the new coefficient is added to the multiplication result to produce the new accumulation.

While the multiplication fraction result is being transferred to AR, the exponent result is checked for exponent overflow or underflow. If no overflow or underflow is found, the addition will begin as soon as the new coefficient data is ready. If, however, overflow or underflow are sensed, special flows that attempt to recover from the over/underflow are accessed (Paragraph 2.2.6.4).

While the new coefficient data is checked for zero and/or reserved operands, the addition/subtraction begins on the assumption that the coefficient data will be valid. The exponent difference hardware selects the larger exponent for processing by the FCT and loads it into PR. It also shifts and loads the fraction associated with the smaller exponent into the B-input of FALU. FALU then adds or subtracts the fraction. When the coefficient data proves valid, the computed fraction result is transferred to NR where it can be normalized.

The fraction normalization takes place in the FNM logic. A rounding byte is added and the result is shifted until normalized. The exponent is adjusted based on both the rounding byte and the number of shifts required to normalize the fraction. The normalized fraction is moved to MC and a multiply with the stored argument (x) begins.

Once the first multiply is completed, the POLY calculation is in the general POLY flow. These flows multiply by the result of the last add and normalize by the argument (x), receive a new coefficient from the CPU, check it for exceptional condition, then add it to the result of the multiply operation, normalize the result of the addition, and ready it for the next multiply. The general POLY flows check the intermediate results for overflow, underflow, and zeros, and access special flows if an exception is found.

The general POLY flow continues until the CPU sends a coefficient flagged with POLY DONE rather than CP SYNC. This indicates that the coefficient just transmitted is the final coefficient in the table. The POLY DONE flow adds the final coefficient and then accesses the normalization flows in the FPA addition flows. These flows round and normalize the fraction and adjust the exponent based on the-rounding byte and normalization shift. Once the result is complete, it is placed on the FP bus A and standard routines handle the transfer to the CPU.

2.2.6.4 POLY Exception Flows - The POLY flows have many special sections to check for and handle exceptional conditions. Each coefficient is checked for zeros and reserved operands. The POLY argument is checked for zero. The CPU checks the argument and degree for reserved operand. The FPA also checks the intermediate results for underflow, zero, and overflow. If an underflow or overflow is detected, special flows attempt to recover from the condition without a loss of accuracy.

The exception flows (zero, reserved operand, overflow, and underflow) can be divided into three categories to handle exceptions discovered during:

1. First coefficient and argument handling
2. General coefficient handling
3. POLY DONE (final coefficient) handling.

Within each category, different microcode handles float and double precision operation. However, there is little difference between the exception procedures used in each category and only minor differences in the microcode. As a result, each individual exception flow is not discussed, rather the microcode procedure for each type of exception is explained.

Zeros
The argument and each coefficient are checked for zeros. The argument and first coefficient are checked for zeros at the start of the POLY flow. If the argument (x) is zero, all the terms of the polynomial will be zero except A_0, the last coefficient. With the argument equal to 0, the FPA will remain in the first coefficient loop waiting for the last coefficient (flagged by POLY DONE). When it is received, it will be tested for reserved operand and, if not reserved, will be returned to the CPU as the result of the polynomial. If the first coefficient is zero, the accumulation registers will be set to zero and the FPA will wait for the next coefficient.

If a zero is found as a subsequent coefficient (when the current accumulation is not zero), the current accumulation which is unnormalized will be rounded and normalized, and the FPA will wait for the next coefficient.

Reserved Operand
Each coefficient is checked by FPA hardware for reserved operand. If a reserved operand is found, the POLY operation is immediately aborted and the accelerator error bit is set. The argument is not checked for reserved operand by the FPA because it is checked in the CPU and, if found to be reserved, the POLY operation never starts in the FPA.

Overflow
The FPA checks for overflow by examining the exponent bits PR8 and PR9 in the PR register. If PR8 (the overflow bit) is high and PR9 is low, an overflow has occurred.

The FPA checks each current accumulation two times per cycle for an overflow condition – once when the unnormalized multiplication result is readied for adding the new coefficient and once after the addition result has been rounded and normalized. If an overflow is detected in the second instance (normalized addition result overflow) the FPA will abort. The FPA will set the PSL V (overflow) bit and wait until the CPU traps it back to IRD.

If the unnormalized multiplication result overflows, the FPA accesses overflow routines in an attempt to recover an accurate result from the overflow. The FPA microcode is written based on the assumption that if the new coefficient exponent is subtracted from the current overflow, the result may be small enough that the exponent will no longer overflow (PR8 will be low.) As stated before, PR8 is high. This means the exponent in PR is 10XXXXXXX (9 bits long.) Since the exponent difference taker EALU is only 8 bits long, the overflowed exponent must be scaled down. The FPA subtracts 80_{16} to scale it down.

The new coefficient is first checked for zero or reserved operands. A reserved operand causes an abort. If the coefficient is zero, it will not change the overflow. The FPA will attempt to recover from the overflow by first adding back the 80_{16} to return the exponent to correct value, then normalizing and rounding. If this fails the FPA will set the overflow bit and abort.

If the new coefficient is not zero or reserved, the operation continues. The FPA subtracts 80_{16} from the exponent of the coefficient to scale it down. The reduced exponent coefficient is checked for underflow. If an underflow is sensed, the coefficient is effectively zero when compared with the accumulation. Since the coefficient is effectively zero, the FPA will attempt to recover from the overflow by first adding back the 80_{16} to return the exponent to correct value, then normalizing and rounding. If this fails, the FPA will set the overflow bit and abort.

If the reduced coefficient did not underflow, it shows that the coefficient can effect the accumulation and possibly recover it from the overflow condition. In the case of accumulation overflow flows, we know the accumulation is the larger number. Therefore, no checks are performed on the exponent to find the larger number. The exponent difference taker then subtracts the two scaled down exponents to determine how many times the coefficient must be shifted to align the radix points. The POLY add/subtract will take place. The accumulation fraction is moved through ADER MUX to FALU and the restored (80_{16} added) accumulation exponent is moved to PR for processing.

The POLY add/subtract takes place. The fraction result is moved to NR where it is normalized and rounded. The result exponent (formerly the accumulation exponent), is adjusted based on the fraction normalization and rounding. The result is checked for overflow and underflow. As stated at the beginning of this overflow section, an overflow after the normalization and rounding operation will cause the FPA to assert the overflow V bit and abort.

Underflow

The FPA can handle numbers as small as $.29 \times 10^{-38}$. A number smaller than this causes an underflow. The FPA checks for underflow by examining the exponent register PR. PR9 will be high or PR<8:0> will be low in an underflow.

Underflow is not as serious a fault as overflow. An underflow means the result just checked is so close to zero that the FPA cannot accurately represent it. When encountered, the FPA sets the ACC ZDATA bit and special flows attempt to recover the number. If the underflow result cannot be recovered, the number is set to zero and FPA operation continues. After the POLY operation is completed, the CPU will trap on underflow if bit 6 (floating underflow) of the PSL is set.

The FPA checks for accumulation underflow twice per POLY cycle, once as the unnormalized multiplication result is readied for the following addition and once after the result of the addition has been normalized and rounded. If an underflow is detected in the normalized addition result, no result recovery is possible. The FPA merely sets the accumulation to zero, informs the CPU of the underflow, and continues the operation.

If an underflow is detected after the multiplication, special flows are accessed to save the result. In an underflow the exponent of both the accumulation and the coefficient must be scaled up so the exponent difference can be taken with an 8-bit exponent processor. The scale factor is 80_{16}.

The coefficient is first checked for zero or reserved operands. A reserved operand causes an abort. A zero coefficient will not change the underflow so the FPA will try to recover by normalizing and rounding. If this fails, the accumulation will be cleared (set to zero) and the FPA operation continues.

If the new coefficient is not zero or reserved, the operation continues. The FPA adds 80_{16} to both exponents to scale them up. If the coefficient exponent overflows when it is scaled up, the coefficient is so much larger than the accumulation that the accumulation will not effect the coefficient. The FPA will disregard the accumulation and make the new coefficient the accumulation by subtracting the 80_{16} just added to the coefficient exponent and moving the coefficient to the registers formerly holding the underflow accumulation.

If the new coefficient does not overflow, it shows that the coefficient can effect the accumulation and the exponent difference taker determines the exponent difference. Since the coefficient is the larger number, the coefficient fraction is moved through the ADER MUX to the FALU and the coefficient exponent is stored in PR after the bias previously added is removed. The accumulation fraction is shifted based on the exponent difference until the radix points align, and then added/subtracted. The result is rounded and normalized in the normalize logic. The coefficient exponent (stored in PR) is adjusted based on the fraction normalization and rounding, and becomes the accumulation exponent. The rounded result is checked for underflow. If underflow is detected, the ACCZ bit is set and a zero is stored. The FPA informs the CPU that an underflow has occurred by asserting both FP SYNC and ERR SYNC. In any case, the polynomial operation continues.

2.3 BLOCK DIAGRAM AND UNIT DESCRIPTION

This section provides a functional description of each area of the FPA with relation to the control store and instruction execution. Discussions of logic unit operations are included for areas that require further clarification.

The FPA can be divided into three areas. The first area contains two interface sections: the CPU-FPA interface and the FPA internal buses (which interface between the various sections of the data manipulation area). The second area, data manipulation, contains five sections: Fraction Adder/Subtractor, Fraction Normalizer/Divider, Fraction Multiplier, Exponent Processor, and Sign Processor. Each section in this area operates as an independent unit, capable of processing data in parallel with operations being performed in other sections. The third area contains only the Control Store and Logic which controls both interfacing and data manipulation. Refer to Figure 2-9, the FPA Block Diagram.

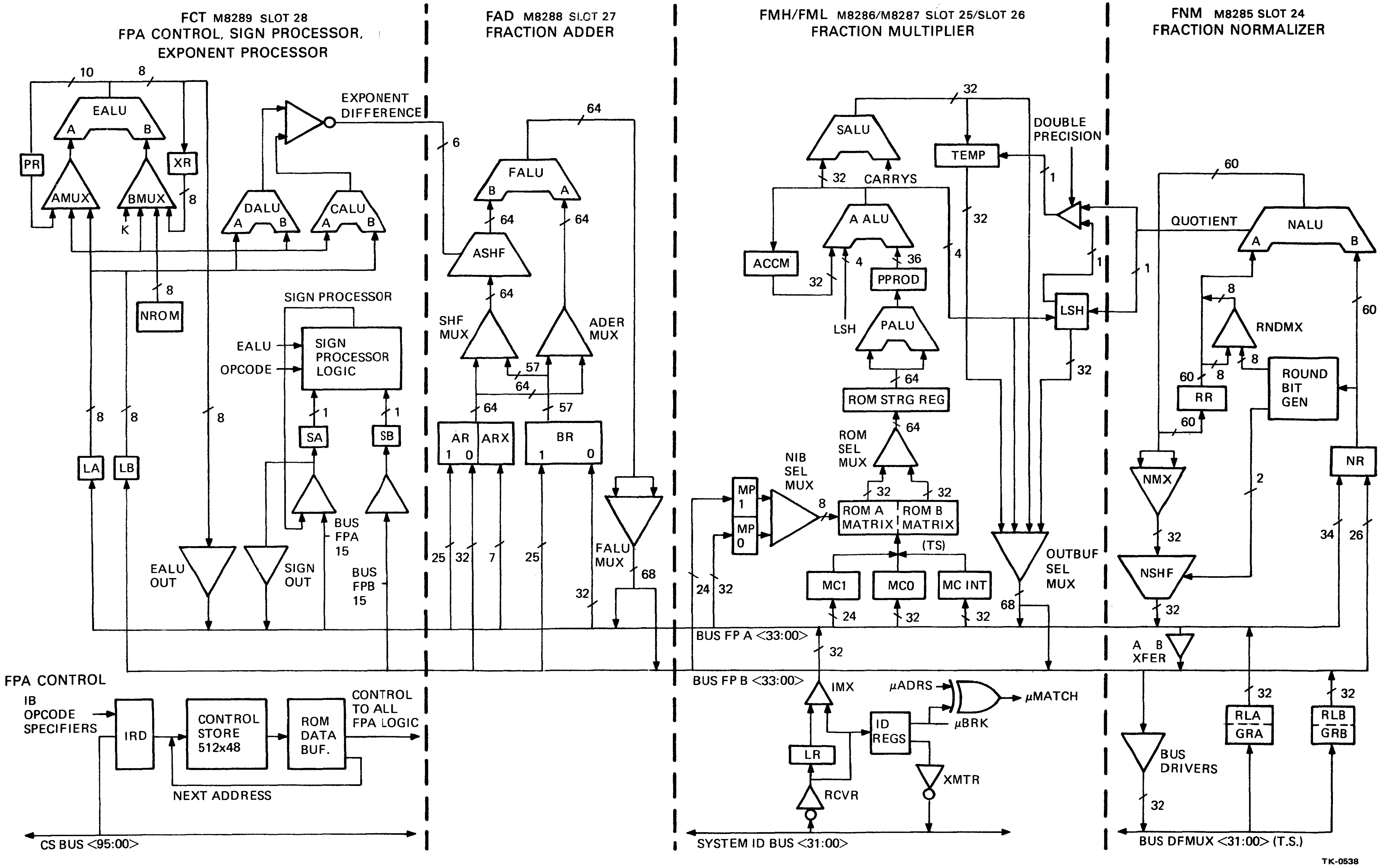


Figure 2-9 FPA Block Diagram

The CPU transmits both data and instructions to the FPA. The instructions are decoded in the Control Store and Logic and access an FPA control store word. The FPA control store word controls the transfer of the data on the FPA internal buses and the operation of the various data manipulation sections. The various data manipulation sections perform the required operations. The resulting answer is formatted and sent to the CPU-FPA interface. A signal from the FPA informs the CPU that the answer is available at the interface.

Each of the eight sections mentioned in this introduction are discussed individually in the following paragraphs. Each discussion includes an explanation of pertinent control store fields and a description of the hardware operation as controlled by the control store, CPU instruction, data characteristics, and both internal and external flags.

2.3.1 CPU-FPA Interface

The CPU and FPA have numerous interconnections. They exchange data, instruction information, device control signals, and status information over buses and individual signal lines. There are three types of information transferred via the CPU-FPA interface.

1. CPU-FPA control and status
2. Data
3. Trap and diagnostic information.

They will be discussed in this order in the following paragraphs. Refer to Figure 2-10 for a summary of the CPU-FPA interface.

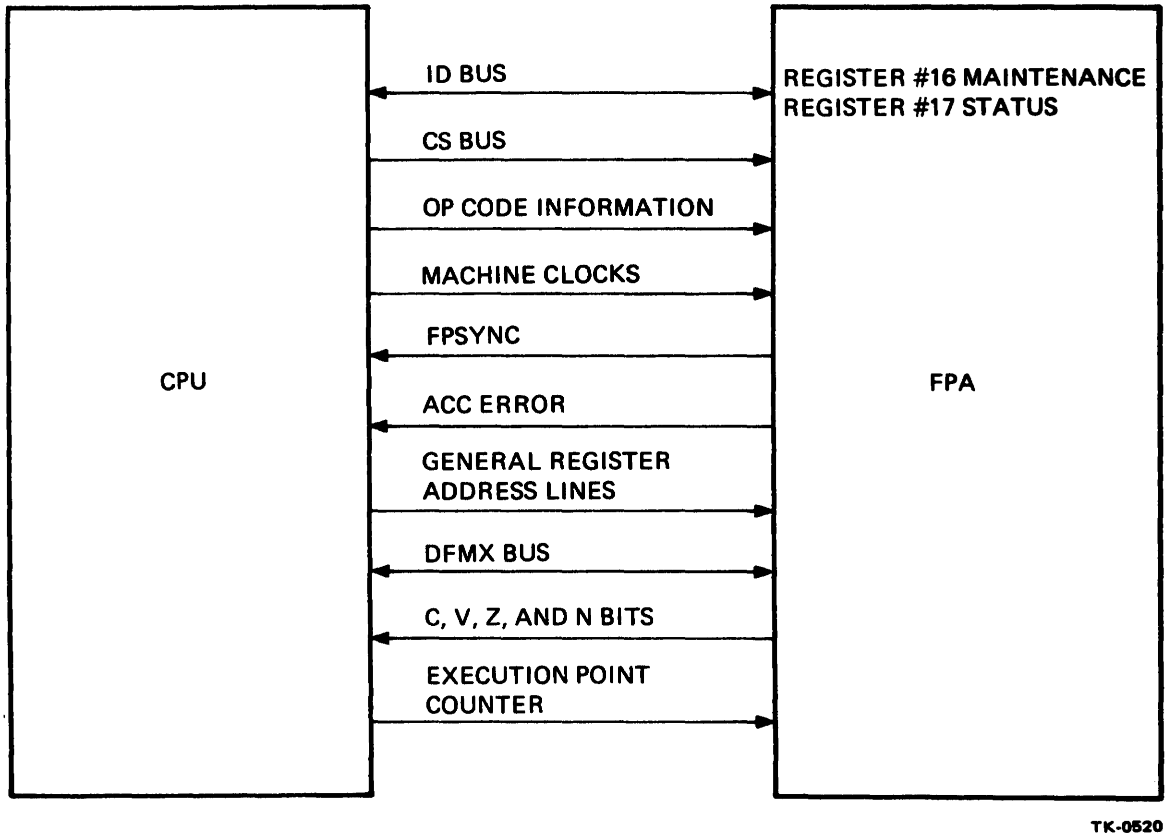


Figure 2-10 CPU-FPA Interface

2.3.1.1 CPU-FPA Status and Control Interface - The FPA and CPU work interactively. This means they are constantly exchanging status and control information, and that operations in one unit can and do effect operations in the other unit. The status register (ID register 17) provides some CPU control of the FPA. Bit 15 of the status register is used by the CPU to enable the FPA. The CPU can disable all FPA outputs and effectively remove the FPA from the computing system by clearing bit 15. Refer to Figure 2-11 and Table 2-7 for a complete description of this register.

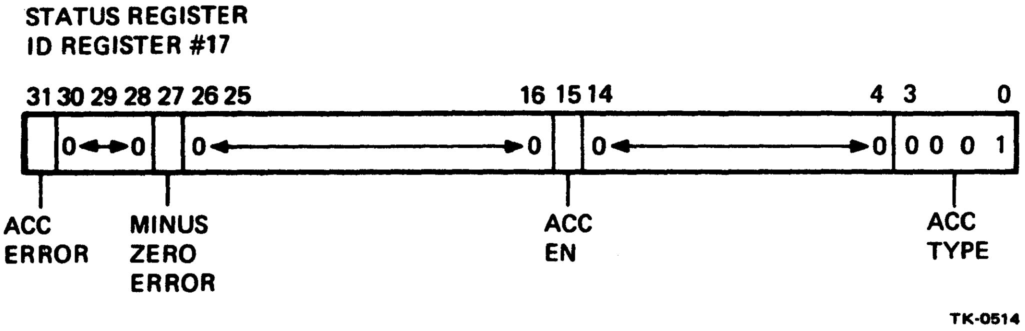


Figure 2-11 Status Register

Table 2-7 The Status Register

Bit No.	Name	Bit Access	Function
31	Accelerator Error Also called ACC Also called Error Sync	Write by FPA Read by CPU	Set when FPA detects an exception condition.
30–28	Not Used-Set to zero		
27	Minus Zero Error	Write by FPA Read by CPU	Set when FPA encounters a reserved operand or generates an overflow. Setting this bit sets Accelerator Error.
26–16	Not Used–Set to zero		
15	Accelerator Enable	Write by CPU Read by FPA	When clear all FPA outputs are disabled. This removes the FPA from the computing system. Must be set for normal FPA outputs.
14–4	Not Used–Set to zero		
3–0	Accelerator Type	Read by CPU Hardwired in FPA	A hardwired code identifies the type of accelerator installed in the backplane slots. The FPA code is 0001.

The FPA also receives control and status information from the CS bus. The functions of these lines are summarized in Table 2-8.

Table 2-8 CS Lines

CS BUS 71	70		Name	Function
0	0		NOP	
1	0		ACC TRAP	Initiates an Accelerator trap. Refer to Paragraph 2.3.1.3
0	1		CPSYNC	Indicates CPU has received FPA data or CPU is presenting valid data to FPA.
1	1		Redefine μSI	Decodes CS lines 57, 56, and 55 for more information.
CS BUS 57	**56**	**55**		
1	1	0	Poly End	Indicates last term of polynomial has been transmitted from CPU.
1	1	1	FP TRAP	Initiates an FPA trap. Refer to Paragraph 2.3.1.3.

Op code information (operation and precision) is transmitted to the FPA from the instruction buffer via IRC OPC lines 7 to 0. These lines, from byte 0 of the instruction buffer, are used by the A-Fork/B-Fork logic and BEN logic for FPA control store next address generation (refer to Figure 2-34). A few other lines from the instruction buffer and decode logic provide specifier source information to the FPA. The possible sources of data are as follows:

1. Memory
2. Register
3. Short literal
4. Long literal.

The CPU-FPA interface includes clock signals from the CPU to the FPA. The units operate synchronously on a 200 ns cycle. The T0 of both units coincide.

The FPA transmits two status signals to the CPU: FP SYNC and ACC ERROR. These signals are input to the CPU for branch control. FP SYNC is normally asserted when an FPA result is available to the CPU. ACC ERROR is set during an FPA error condition.

2.3.1.2 CPU-FPA Data Interface - The FPA receives operand data from the CPU and, after performing the required operation, returns the answer to the CPU. The data is transmitted to the FPA via the ID bus and is returned to the CPU via the DF mux bus. As mentioned previously the FPA does not do any memory accessing. The CPU must calculate the data memory address, access the address, and place the data on the ID bus to the FPA.

The FPA is optimized to use CPU scratchpad register data. It stores two copies of the 16 CPU scratchpad registers. To ensure that the FPA copies are exact copies, the FPA copies are addressed and written by the same lines that address and write the CPU general registers. The address lines are from the DAP board and the data is transmitted via the DF mux bus. To ensure that a changing register is never read, the CPU updates the general register and the FPA copies between T100 and T200 (T0) and the FPA reads the copies between T0 and T100. Note that the FPA general register copies are write-only memory to the CPU and read-only memory to the FPA. This means that results of FPA operations that are destined for the general register set are transmitted back to the CPU via the DF mux bus and then written into the general register set under CPU control rather than written directly into the general register copies by the FPA.

The data stored in the FPA general register copies is read by the FPA using address lines from the instruction buffer operand source logic. This scheme enables the FPA to access register data and begin the operation as soon as the general register address/addresses is/are in the instruction buffer.

All operands other than register operands are transmitted to the FPA via the ID bus. This includes memory data, and long and short literals. When memory data is specified in an instruction, the CPU fetches it and places it in the CPU D-register. The contents of the D-register is placed on the ID bus and, in the FPA, is transferred from the ID bus directly onto the FP buses. Since the D-register and ID bus are only 32 bits wide each, it takes two transfers to transmit a double precision number. Single precision (float) literal data, part of the instruction stream is transferred from the instruction buffer onto the ID bus. In the FPA, single precision literal data is latched into the literal register (LR) and then placed on the FP bus. The most significant part of double precision literal data is handled similiarly, i.e., IB → ID bus → LR → FP buses. The least significant part of a double precision literal is transferred from the instruction buffer over the ID bus to the CPU D-register, then back on the ID bus and onto the FP buses. Note that no ID bus addresses are required for data transfers over the ID bus. The FPA simply accepts the current ID bus data.

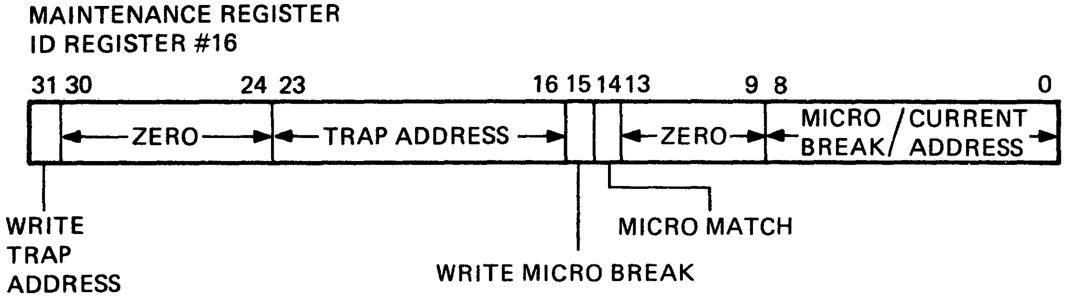
When the FPA operation result is ready to be transmitted to the CPU, FP SYNC is asserted and the single precision result or the most significant part of a double precision result is on FP bus A. The CPU responds to FP SYNC by enabling the FPA DF mux bus drivers which place the FP bus A contents on the DF multiplexer bus. The FPA result is transferred to the CPU D-register via the DF mux bus. When the CPU has the data, it asserts CP SYNC. This ends a single precision (float) transfer or enables the second part of a double precision transfer. For a double precision transfer, the second part is placed on FP bus A and remains there until the CPU responds to the newly asserted FP SYNC by enabling the DF mux bus drivers, accepting the data, and asserting CP SYNC to indicate it has the data.

While the FPA is transmitting the result back to the CPU, valid condition codes are also being transmitted to CPU condition code latches. These latches are read during the next machine cycle. The N, V, and Z bits are set based on the status of the result. The C-bit is always cleared by the FPA.

2.3.1.3 Trap and Diagnostic Information – The FPA contains several features to facilitate error diagnosis and troubleshooting. These include programmable traps, and microdiagnostics, special maintenance features, and the visibility bus.

The CPU can initiate 2 types of traps: ACC TRAP and FP TRAP. CS 71 high and CS 70 low initiate an ACC TRAP. This causes the FPA to access one of the FPA microcode addresses 0 through 7 as selected by CS lines 57, 56, and 55. Currently only 2 of these traps are used: Accelerator Power-Up Trap (address 0) and Accelerator Abort Trap (address 2). The FP TRAP (used for FP microdiagnostics), is selected by CS lines 71, 70, 57, 56, and 55 high. When FP TRAP is asserted, the FPAmicrocode address is selected by bits 23 through 16 of the maintenance register. The trap address (0 through 255 in the microcode) is selected by the data previously loaded into the maintenance register.

The maintenance register is a CPU-FPA readable/writeable register located on the ID bus. The CPU accesses this register as ID bus register 16. The register is designed to facilitate maintenance. As discussed previously it contains the FP trap diagnostic address. Using the trap address the CPU can exercise various sections of FPA logic. Bit 14 of this register provides a synch pulse that can be used for troubleshooting with an oscilloscope. This bit will go high each time the FPA accesses the microcode address stored in bits 8 through 0. Refer to Figure 2-12 and Table 2-9 for summary of this address.



TK-0515

Figure 2-12 Maintenance Register

Table 2-9 The Maintenance Register

Bit No.	Name	Bit Access	Function
31	Write Trap Address	Write by CPU Read by FPA	When set (by CPU) enables CPU to write trap address (bits <23:16>).
30–24	Not Used–Set to zero		
23–16	Trap Address	Write/Read by CPU Read by FPA	Selects FPA microcode address for FPA microdiagnostics.
15	Write Microbreak	Write by CPU Read by FPA	When set (by CPU) enables CPU to write microbreak (bits <8:0>).
14	Micromatch	Write by FPA Read by CPU	Set by FPA when currently accessed by FPA microcode address equals address stored in microbreak (bits<8:0>).
13–9	Not Used–Set to Zero		
8–0	Micro-break/Current Address	CPU writes microbreak. FPA reads microbreak. FPA writes current FPA microcode address. CPU reads current FPA microcode address.	These bits serve two functions: 1. The microbreak selects the FPA microcode address to be monitored for micromatch (bit 14). 2. The current address provides CPU monitoring of FPA microcode activity.

Forty-three FPA signals are accessed by the Visibility Bus (V bus). The V bus is a diagnostic tool, designed to allow polling of stable internal CPU (in this case, FPA) signals. The console can issue commands which load the V bus latches with the signals monitored and then shift the loaded latches one bit at a time to a control word located in the console interface. At the console, the data shifted in will be examined by diagnostic software. There are 8 data input channels on the V bus, channel 6 is devoted to the FPA. Refer to Table 2-10 for listing of the FPA signals that are available to the V bus.

Table 2-10 Signals Monitored by Visibility Bus

FCTE SHF COUNT 5 H	FCTD EALU 0 L
FCTE SHF COUNT 4 H	FCTE COMPL L
FCTE SHF COUNT 3 H	FADR SPC (0) H
FCTE SHF COUNT 2 H	FNMS EALU CIN L
FCTE SHF COUNT 1 H	FCTC SEL NORM H
FCTE SHF COUNT 0 H	FCTP RA ADRS 3 L
FCTN FALU CARRY IN H	FCTP RA ADRS 2 H
FCTN FAMX SEL 0 H	FCTP RA ADRS 1 L
FCTN FAMX EN 0 L	FCTP RA ADRS 0 L
FCTA A GT B J	FCTP RB ADRS 3 L
FCTN SHF MUX EN 1 L	FCTP RB ADRS 2 L
FCTN SHF MUX EN 0 L	FCTP RB ADRSS 1 L
FCTN FALU FUNC SEL 2 H	FCTP RB ADRS 0 L
FCTN FALU FUNC SEL 1 H	DAPL ACC CONTEXT 0 H
FCTN FALU FUNC SEL 0 H	DAPL ACC CONTEXT 1 H
FCTN FAMX SEL 1 H	FCTC CLR RR L
FCTN LOAD AR1 H	FCTH CP SYNC H
FCTN LOAD AR0 H	FNME BUS → EXP L
FCTN LOAD ARX H	FCTJ ACC NDATA H
FCTN LOAD BR1 H	FCTC ACC ZDATA H
FCTN LOAD BR0 H	FCTC ACC VDATA H
FADS BUS → FAD L	

2.3.2 FPA Internal Buses

As discussed in Paragraph 2.3, the FPA internal buses transmit data between the various data manipulation units. These units are arranged along two parallel 34-bit tristate buses called FP bus A and FP bus B. These buses transmit data from the CPU-FPA interface to the various data manipulation units, transfer intermediate results between units, and return the result to the FPA-CPU interface. The buses can transfer a complete 64-bit double-precision word or two 32-bit float words simultaneously.

The BSC field of the microword controls a majority of the bus activity. The available sources include all FPA data manipulation units and the CPU-FPA interface. Refer to Table 2-11 for a summary of BSC bus control operations. Note that the BSC field controls only the data source. The destination is enabled via other control fields and accepts the data available onthe FP buses.

Table 2-11 BSC Control Store Field

Hex	BSC Field				Microcode Mnemonic	Function
	3	2	1	0		
	μCS 15	μCS 14	μCS 13	μCS 12		
0	0	0	0	0		
1	0	0	0	1		
2	0	0	1	0		
3	0	0	1	1	INTH	Bus A ← SALU
4	0	1	0	0	NL	Bus B* ← Bus A*← NSHF LO
5	0	1	0	1	NH	Bus B* ← Bus A* ← NSHF HI EXP SGN (Packed result)
6	0	1	1	0	PQ	Buses ← SALU and LSH if MUL TEMP and LSH if DIV (LSH is accessed differently if MUL or DIV)
7	0	1	1	1	INT L	Bus A ← LSH
8	1	0	0	0	ID	Bus B* ← Bus A* ← ID Bus
9	1	0	0	1	LR	Bus B* ← Bus A* ←LR
A	1	0	1	0	ID.RB	Bus A ← ID bus Bus B ← RB
B	1	0	1	1	R	Bus A ← RA Bus B ← RB
C	1	1	0	0	FAL.X	Bus A ← FALU HI/LO Bus B ← FALU LO/HI OR
D	1	1	0	1	FAL.LH	Bus A ← FALU LO Bus B ← FALU HI
E	1	1	1	0	FAL.HL	Bus A ← FALU LO Bus B ← FALU HI
F	1	1	1	1		

*The same data is placed on both buses.

The buses handle both floating-point and integer numbers. The buses can handle intermediate, unpacked, and unnormalized data as well as final packed and normalized results. Since the buses must handle intermediate data each bus contains two extra lines to handle the overflow and hidden bits. Refer to Figure 2-13 for summary of data formats used on FP buses.

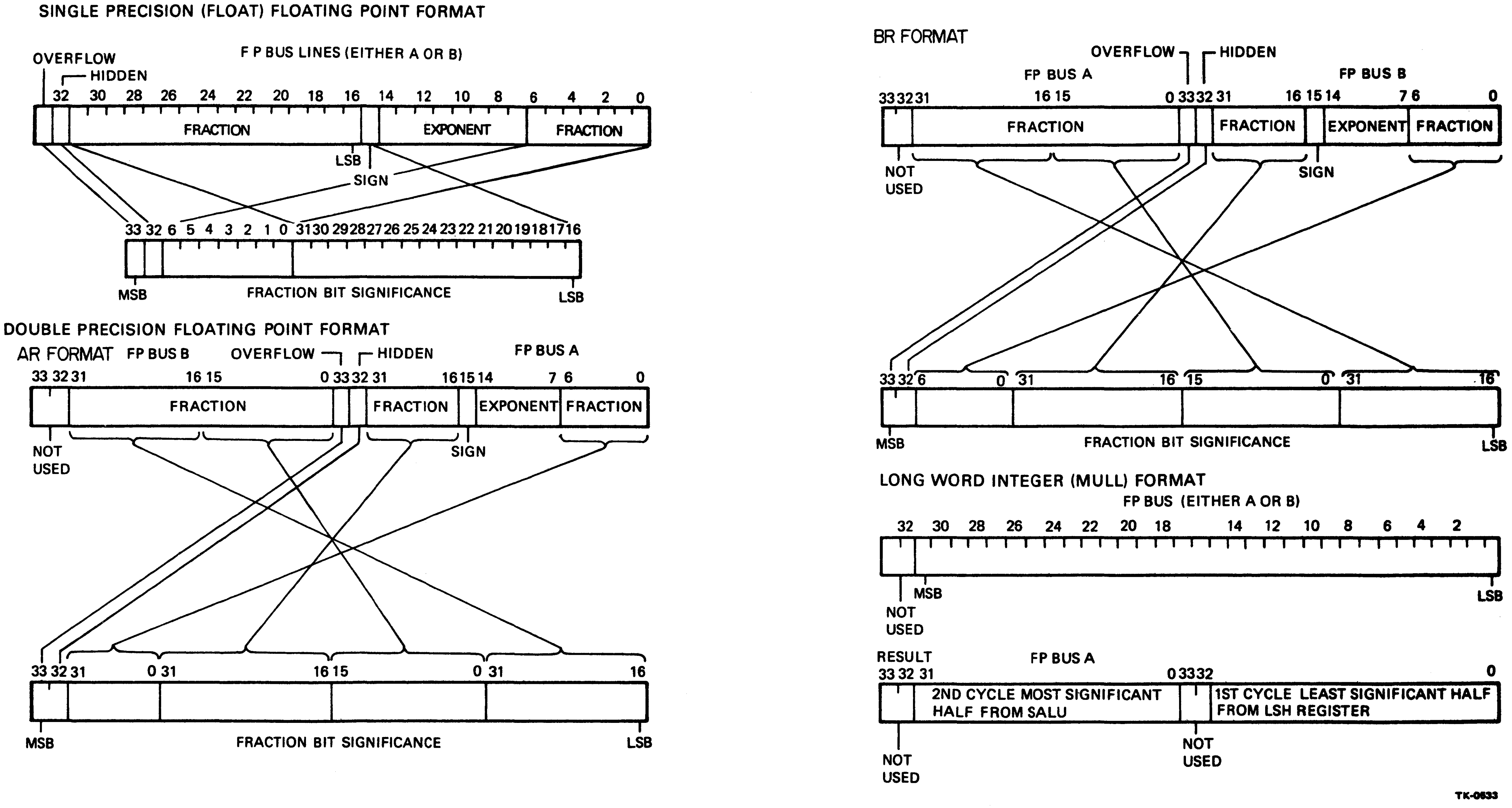


Figure 2-13 FP Bus Formats

2.3.3 Fraction Adder (FAD)

The fraction adder aligns and adds or subtracts the fraction portions of two FPNs. The module contains 2 registers that receive data from the FP buses, 2 multiplexers that manipulate the register data, a shifter to align register contents before an add or subtract, an ALU to add or subtract the data, and bus drivers to place the result on the FP buses (Figure 2-14). Certain FAD signals are interfaced to the V-bus for maintenance and diagnostic purposes. Refer to Paragraph 2.3.1 for a discussion of the V-bus.

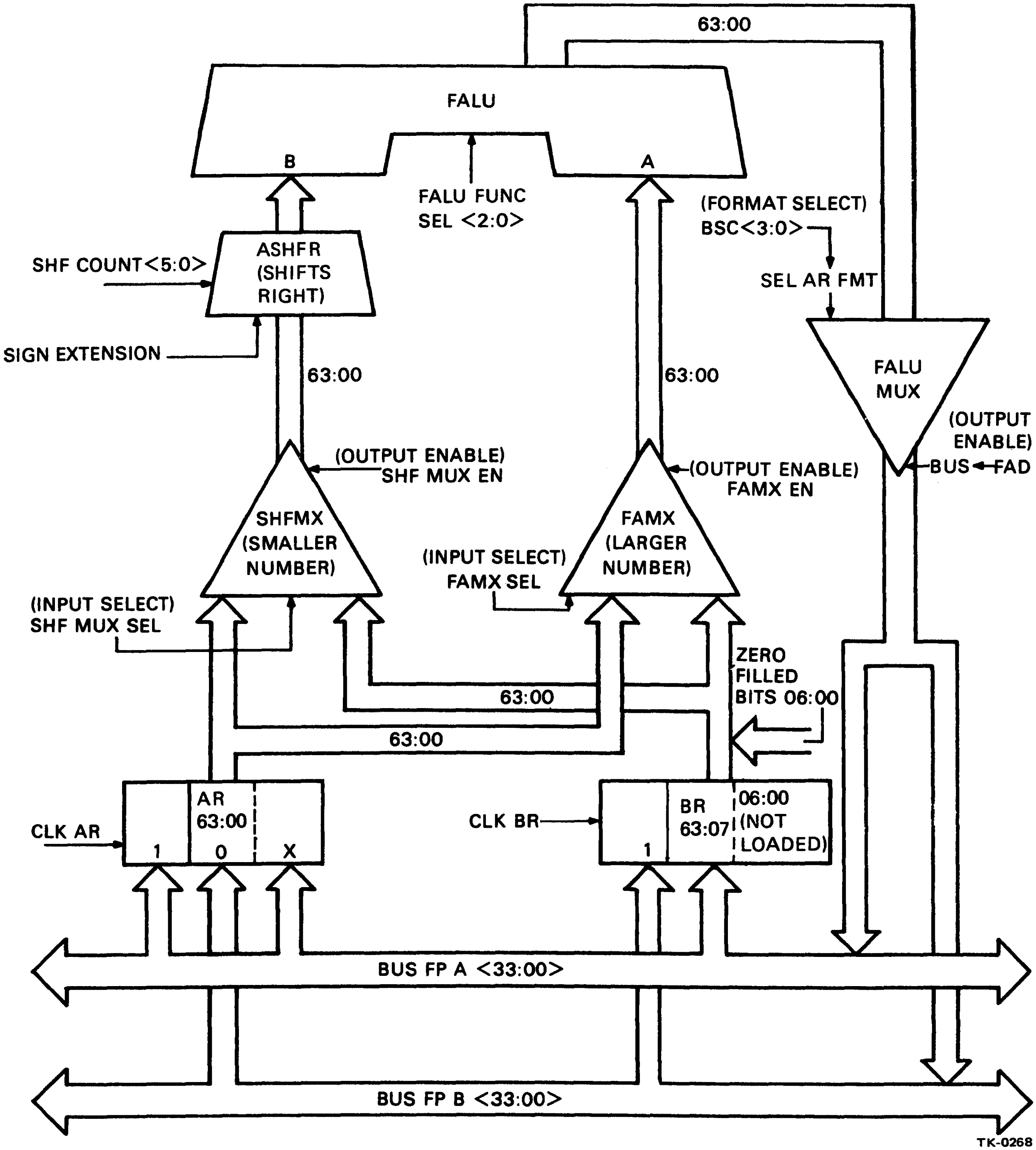


Figure 2-14 Fraction Adder Block Diagram

The fraction parts of the FPNs are loaded into the AR and BR registers. The data entry is controlled by the FADC (Fraction Processor Controls) control store field as shown in Table 2-12. Both registers are loaded with the MSB in bit 63. The execution of the POLY instruction causes an additional 7 LSBs to be transmitted via FP bus A lines <14:08> (where the FPE is normally) and placed in AR <6:0> by loading ARX.

Table 2-12 Fraction Data Entry

	FADC Fields				Operation				
	3	2	1	0	LOAD				
Hex	μCS 11	μCS 10	μCS 9	μCS 8	AR1	AR0	ARX	BR1	BR0
0	0	0	0	0	1	1	1	0	0
1	0	0	0	1	0	0	0	1	0
2	0	0	1	0	1	0	0	0	0
3	0	0	1	1	1	0	0	1	0
4	0	1	0	0	0	0	0	1	1
5	0	1	0	1	0	0	0	0	1
6	0	1	1	0	0	1	0	0	0
7	0	1	1	1	0	1	0	0	1
8	1	0	0	0	1	1	1	1	1

Select lines controlled by both microcode and hardware normally load the FPF associated with the smaller exponent into the SHFMX and the other fractional part into FAMX.

The contents of SHFMX is then right-shifted up to 63 bits to ensure that the radix points align. The magnitude of the exponent difference determines the amount of the shift. The shifted number is padded on the left with its sign. In most cases, the fraction is positive (Figure 2-15).

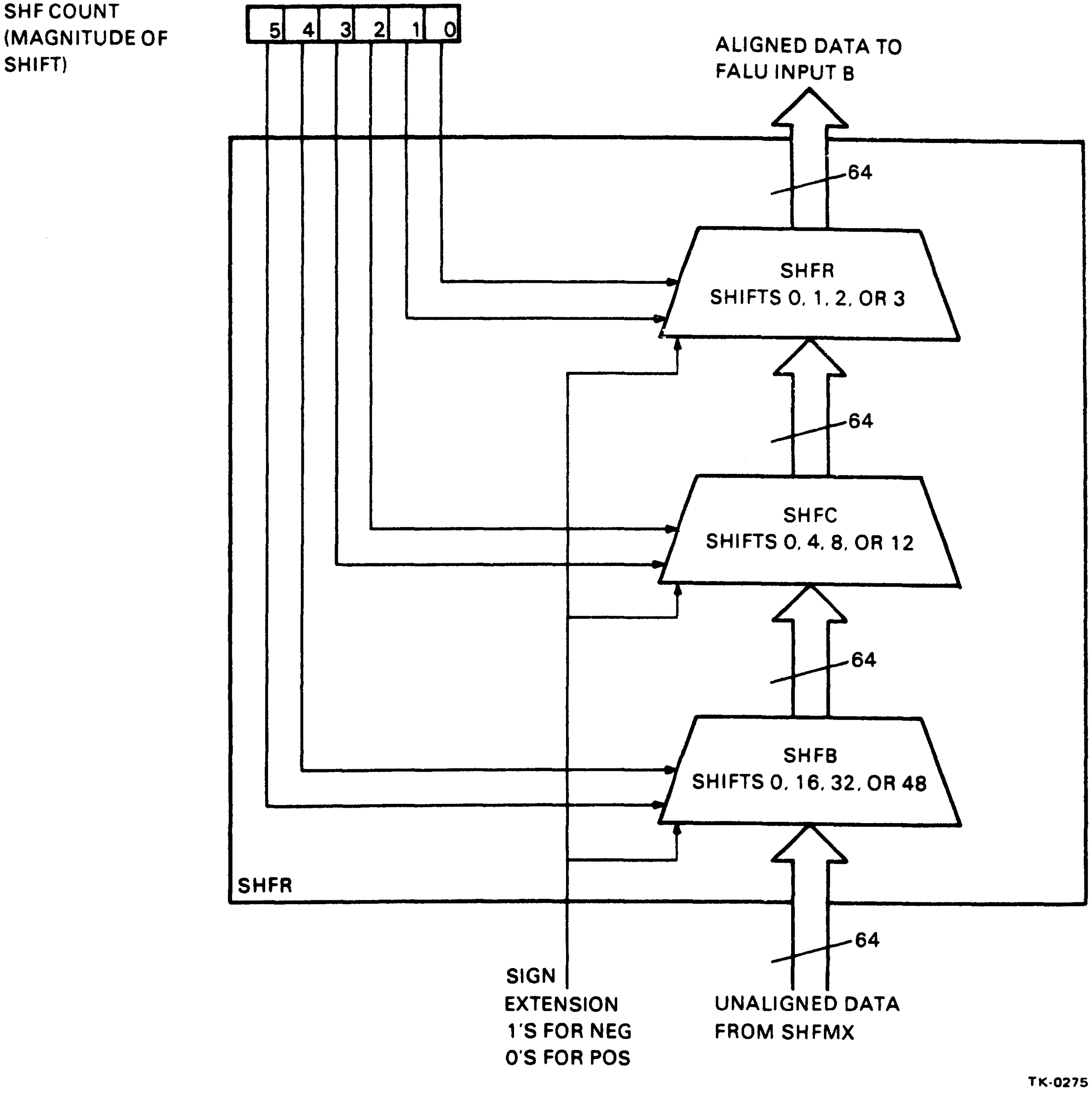

SHF COUNT (MAGNITUDE OF SHIFT)
5
4
3
2
1
0
ALIGNED DATA TO FALU INPUT B
64
SHFR
SHIFTS 0, 1, 2, OR 3
64
SHFC
SHIFTS 0, 4, 8, OR 12
64
SHFB
SHIFTS 0, 16, 32, OR 48
SHFR
64
SIGN EXTENSION 1'S FOR NEG 0'S FOR POS
UNALIGNED DATA FROM SHFMX
TK-0275

Figure 2-15 SHFR Operation

When the two FPFs are aligned, the FALU operates on the two fractions. The FALU operation is determined by the op code and the sign of the two numbers. Refer to Table 2-13.

Table 2-13 FALU Operation

Instruction	Sign of Numbers	FALU Operation
Add	Like(Both + or -)	Add
Add	Unlike	Subtract
Subtract	Like	Subtract
Subtract	Unlike	Add

FALU Operations Selected

S_2	S_1	S_0	Function	Comment
0	0	0	Clear	
0	0	1	B-A	B = 0. Used for complementing number when Shift/Subtract D.P. would lose bits off end. Used when SUBD and exponent difference is greater than 7 or POLYD.
0	1	0	A-B	Normal Subtract
0	1	1	A+B	Normal Add
1	0	0	Not Used	
1	0	1	A or B	Used to get A out or B out. Other side is zero.
1	1	0	Not Used	
1	1	1	Not Used	

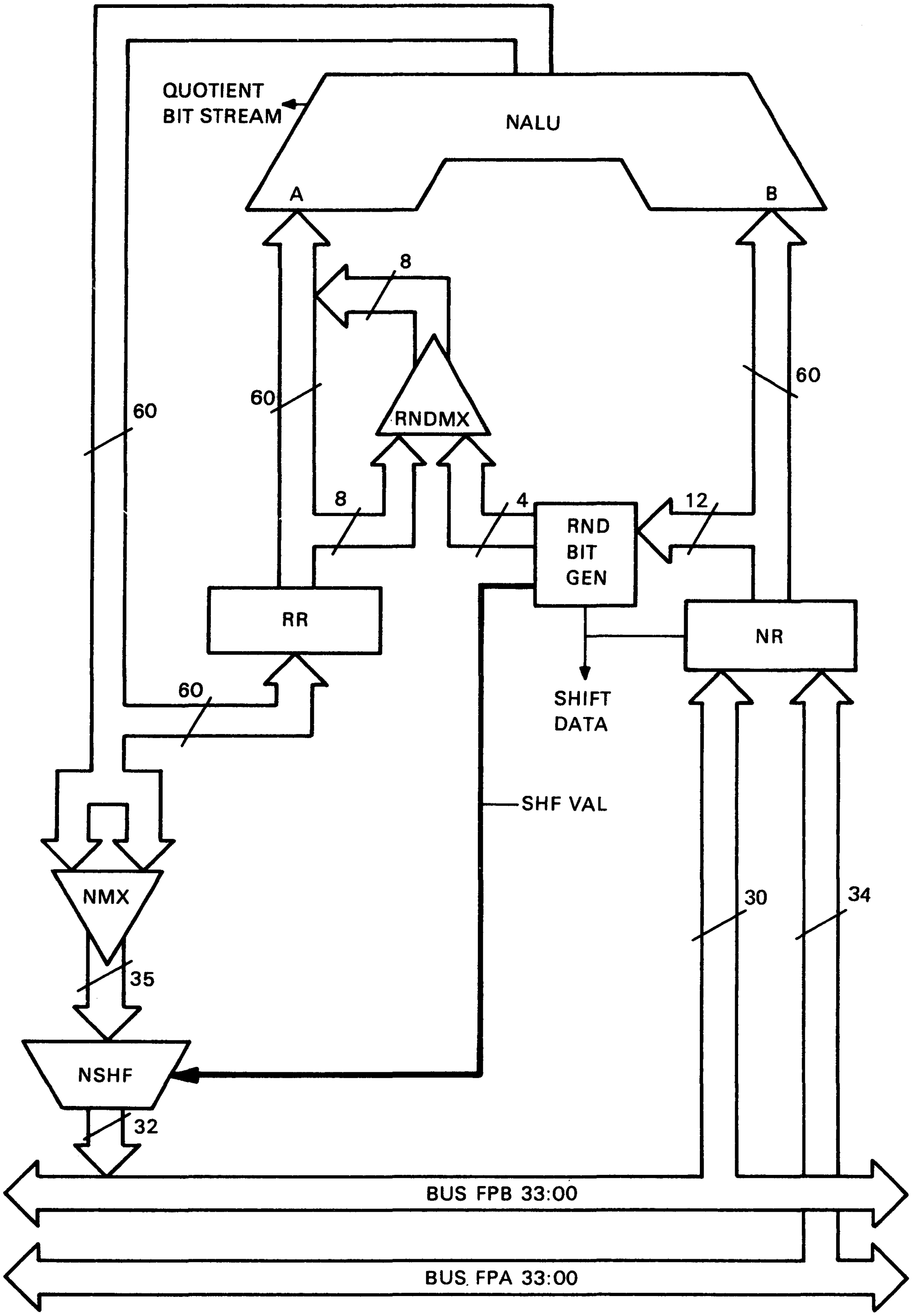
The output of the FALU is loaded onto the FP buses under control of hardware and the BSC microcontrol field. Refer to Table 2-14. The result is in unnormalized form. When a double precision ALU subtraction is done (either as the result of an ADDD, SUBD, or a POLY instruction), the exponent difference is examined. If it is less than or equal to 7, operation continues as usual. However, if the difference is 8 or more, error will be introduced into the LSB if a shift, then subtract is done. To prevent this error, special control hardware is enabled. It disables the output of SHFMX, forcing zeros into the shifter. The smaller operand is routed through FAMX to the A side of the ALU. A B-A (B = all zeros) is done, complementing the operand. The larger operand remains stored in its original register. The result of the ALU operation is output to the FP buses and reloaded into the AR or BR depending upon where it was before complementing. During the next machine state the complemented operand is aligned, sign-extended and added to the other operand. The result is loaded onto the FP buses and is normalized.

Table 2-14 FALU MUX Control

	BSC Field				**FALU Function**
	3	**2**	**1**	**0**	
HEX	μCS 11	μCS 10	μCS 9	μCS 8	
0-B	Not used for FALU MUX Control				
C	1	1	0	0	Hardware determined. NOTE During double precision add/subtract and poly; If EXP A<EXP B, AR format is used. If EXP B<EXP A, BR format is used.
D	1	1	0	1	FP A FALU L (BR Format) FP FALU H
E	1	1	1	0	FP A FALU H (AR Format) FP B FALU L

2.3.4 Fraction Normalize/Divide (FNM)

The normalize/divide logic located on FNM performs the two functions indicated by its title. Refer to Figure 2-16. The hardware can either normalize the fractional result of an add, subtract, multiply or divide, generate the quotient given a divisor and dividend. The quotient is generated bit by bit and stored elsewhere. When the quotient is complete, it is returned to the same hardware to be normalized as any other fraction result. Both functions receive data based on microcontrol words, but once started, operate relatively free of microcode control until they are ready to transmit the answer.



TK-0274

Figure 2-16 Fraction Normalizer/Divide Block Diagram

2.3.4.1 Normalize Operation - Before a normalize operation can take place, the Remainder Register must be cleared. A 3 in the 3-bit MSC field of the microstore word clears it during IRD. Since the divide operations use the RR, it is also cleared during the end of the divide flows before the normalization of the quotient.

The add, subtract, multiply, and divide operations produce results with varying characteristics. The add/subtract operation has the widest variability in result. Operand size (both fraction and exponent), operand sign, and desired operation, all contribute to this variation. The subtraction of two very nearly equal operands can result in a very small number, i.e., a number that must be shifted left many times before it is in final normalized form. Addition of two operands with equal exponents will produce a result between 1 and 2, necessitating a right-shift. Since the add/subtract operations do produce a wide variability of results, special firmware in the control store is accessed and the normalizations proceed under firmware and hardware control.

A divide operation produces results between 1/2 and 2. A multiply produces results between 1/4 and 1. Both divide and multiply normalizations proceed under hardware-only control.

All normalizations begin with NRC equal to 0, parallel-loading the result to be normalized into the NR. If the operation was an A/S, BEN 5 selects special firmware based on exponent differences. If the special firmware is enabled, an NRC equal to 2 enables the NR to shift left in 4-bit steps, 3 steps per machine cycle.

Once the NR shift left is enabled, hardware looks at the top 12 bits of the NR for the first significant bit as the leading bits are left shifted away. In a positive number, leading zeros are disregarded and the first significant bit is a 1. In negative numbers (2's complement notation), leading 1s are disregarded and the first significant bit is a 0 (refer to Figure 2-17). MSN NE SIGN becomes true as the data is parallel-loaded into NR. If the first significant bit is in NR<63:60>. This stops any left shifts. STOP SHF goes high whenever NR <59:56> contain the first significant bit and will cause the NR to stop shifting after one more 4-bit shift (i.e., when first significant bit is in NR <63:60>). If NR <63;52> does not contain the first significant bit, SWR will remain low, shifting all 12 bits out and enabling a new microstore control word via BEN 2. It continues monitoring for the first significant bit. If the NR is left-shifted 60 bits (counted by the control store), and the first significant bit is not found, firmware returns a result of zero by forcing the output of the NMX to zero via FORCE ZERO.

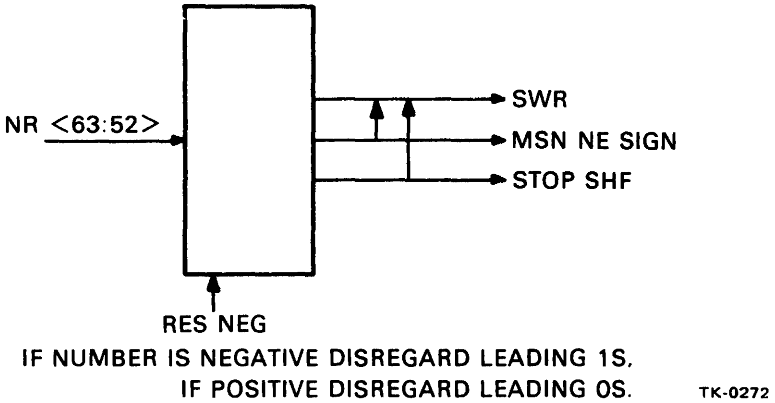

NR <63:52>
SWR
MSN NE SIGN
STOP SHF
RES NEG
IF NUMBER IS NEGATIVE DISREGARD LEADING 1S.
IF POSITIVE DISREGARD LEADING 0S.
TK-0272

Figure 2-17 Normalize Shift Enable Control Hardware

When the first significant bit is in NR <63:60>, the number can be rounded and normalized by the remaining FNM logic.

The round byte contents, NALU operation, and final normalization shift is controlled by the round bit generator. The round bit generator controls these functions based on NR 63, NR 62, NR 61 and RES NEG. The round byte is combined with NR lines 39 through 36 (float or single precision) or lines 7 through 4 (double precision). This is selected via the FLOAT line. Since the final normalization shift takes place after the round byte is added and the first significant bit can be in NR 63, NR 62, NR 61, or NR 60 (it must be in one of these four positions), the position of the round bit (1) in the round byte varies (refer to Table 2-15). As summarized in the table, decode logic divides the 16 possible input cases into 4 cases, corresponding to the FSB in bit 63, 62, 61, and 60. Note that the RBG does not monitor NR bit 63, but, since the logic is only enabled when the FSB is in bits 63 through 60 the RBG logic can sense the contents NR bit 63 even though it does not monitor it. RES NEG L enabled means that the number being shifted and normalized is negative. This means that leading 1s (Hs) should be disregarded in the search for FSB and that the FSB will be a 0 (L). RES NEG L high indicates a positive number, disregard of leading 0s (Ls), and FSB will be a 1 (H). The contents of the rounding byte is based on the location of the FSB. The rounding byte is designed to place a one 24 bits (56 bits for double precision) behind the FSB.

Table 2-15 Round Byte and Normalize Control

1. The logic decodes the four signals and locates the FSB.

RES NEG L*	NR63	NR62	NR61	First Significant Bit (FSB)
L	L	L	L	63
L	L	L	H	63
L	L	H	L	63
L	L	H	H	63
L	H	L	L	62
L	H	L	H	62
L	H	H	L	61
L	H	H	H	60
H	L	L	L	60
H	L	L	H	61
H	L	H	L	62
H	L	H	H	62
H	H	L	L	63
H	H	L	H	63
H	H	H	L	63
H	H	H	H	63

*RES NEG L high indicates a positive number. This means a 1 (H) is the FSB. RES NEG L low indicates a negative number. This means a 0 (L) is the FSB. RES NEG L asserted also causes a NALU subtract thereby rounding and complementing the number in a single step.

Table 2-15 Round Byte and Normalize Control (Cont)

2. Based on location of FSB, an appropriate rounding byte is generated.

FSB	Rounding Byte Selected Bit 3	Bit 2	Bit 1	Bit 0
63	1	0	0	0
62	0	1	0	0
61	0	0	1	0
60	0	0	0	1

3. Also based on location of FSB, the final shift required to normalize and ready the result for the CPU is selected.

FSB	Shift Selected	SHF VAL 1	SHF VAL 0
63	Right 1 place	L	L
62	No shift	L	H
61	Left 1 place	H	L
60	Left 2 places	H	H

If the FSB is not in NR <63:60>, the NR is left-shifted and a binary counter counts each 4-bit shift. This count, RES NEG line, and NR bits 63, 62, and 61 (magnitude of final shift) determine the NORM ROM location to be addressed. The content of this location is added to the exponent of the result in the FALU and corrects it for all shifts that take place in the FNM. If however, the number to be rounded is all 1s, the addition of the rounding byte will ripple through all bits and cause a fraction overflow. This is sensed by comparing the round byte location (indicating where the logic decoded the current MSB of the number to be rounded) and location of the MSB of the rounded result. If this comparison asserts NORM ERR and thus EALU CIN (indicating there was a ripple and subsequent overflow), a one will be added to the EALU (the exponent adder on FCT) to correct the exponent for the overflow. NR <63:04> goes to the NALU B side and round byte (4-bit) goes to the A side. Normally the NR is added to the rounding byte. However, if RES NEG L is asserted, indicating a negative (2's complement) number, the content of the NR is subtracted from the rounding byte. This operation rounds and complements (return to positive notation) in one step.

The 60-bit result <63:04> of the NALU operation (rounded and ready to be normalized) is transmitted to the NMX. The high part (and only part, if float or single precision) is transmitted through to the NSHF for final normalization shift. The NSHF shift control bits select a 0 to 3-bit shift for final normalization.

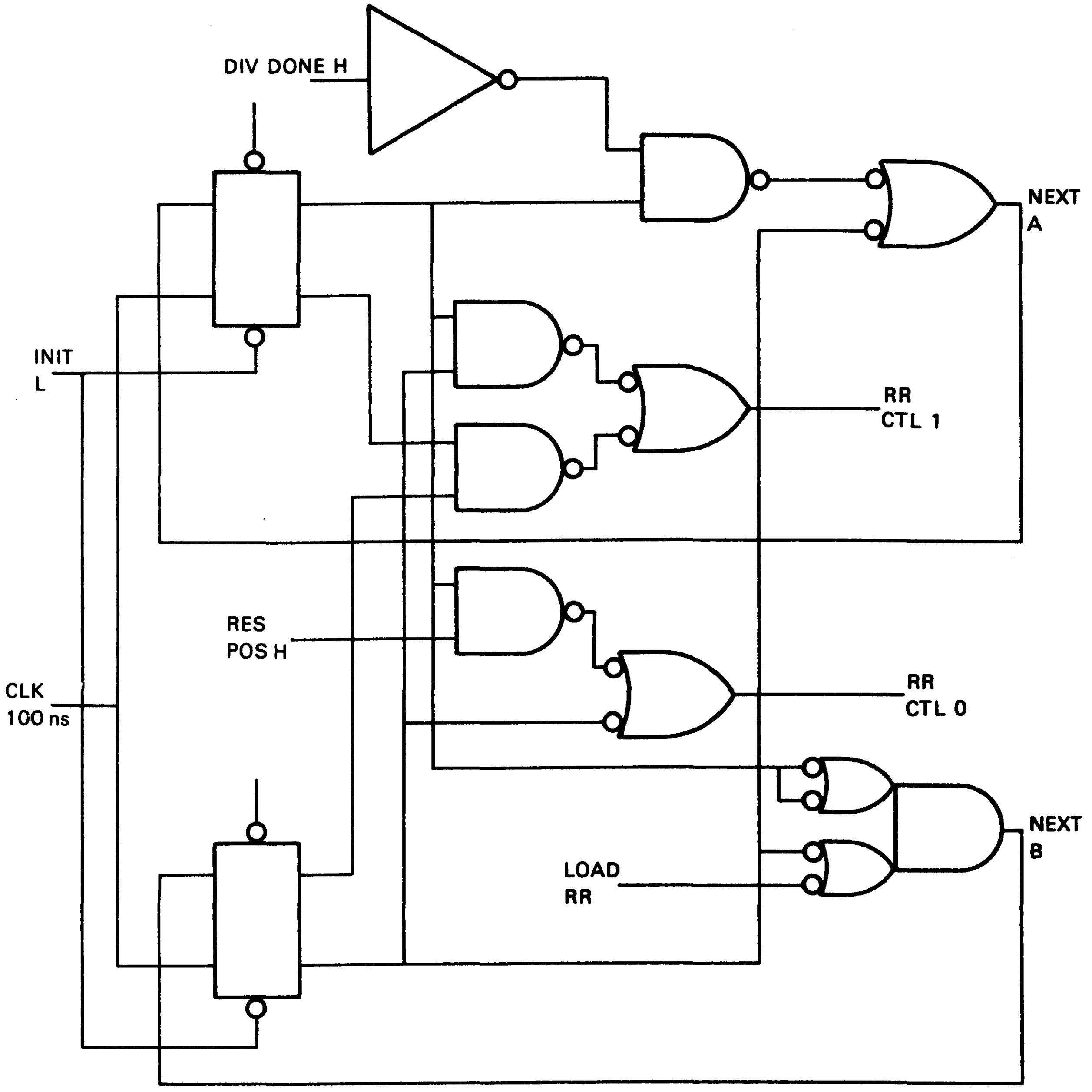
Final normalization moves the MSB to the equivalent of the NR 62 position. When the data is placed on the FP buses, NR 62 (always a one since the fraction is now normalized) is the hidden bit and is placed on the FP bus A bit 32. When the data is transferred to the CPU, the hidden bit is not transferred and the data in NR 61 (bus A bit 6) is the MSB to be transferred.

2.3.4.2 Divide Operation – This logic also performs the fraction part of the divide operation for the FPA. Once the dividend and divisor are loaded into the FNM logic and the quotient storage on the multiplier boards is enabled for either a float (single) or double precision result, the divide operation runs under hardware control until the answer has been computed to the required precision. Once the answer has been computed, microcontrol takes over and transmits the unnormalized quotient back to the FNM logic where it is normalized and rounded like any other fraction.

The hardware uses the restoring, repeated subtraction technique to divide. The dividend is initially loaded into the RR and the divisor is stored in the NR. The divisor (contents of NR) is subtracted from the dividend (contents of RR). If the result is negative, a 0 is left-shifted into the answer (quotient) register and the contents of the RR is left-shifted by one. If the result is positive or 0, a 1 is left-shifted into the answer (quotient) register; and the result is loaded into the remainder register left shifted by one. The divisor (contents of NR) is continually subtracted from the contents of the RR until 26 bits (58 bits for double precision) of quotient are generated. The quotient is then rounded and normalized.

The division operands are loaded under microstore control. The first microstore state loads the dividend into the NR. The second state causes the NALU to OR the contents of the NR with the contents of the RR (currently clear) and load the result of the operation into the RR. In the same state the divisor is loaded into the NR. At the end of the second state the division operands are in their correct register and the divide sequencer hardware takes over.

The divide sequencer hardware generates the RR control signals (refer to Figures 2-18 and 2-19). The RR CTL signals either load the NALU result into the RR or left-shift the RR contents based on the result being negative or positive. The input of the RR is hardwired to automatically produce a left shift when loading NALU result. This means that during the initial loading of the RR, the dividend is left-shifted by 1. The 11 state in Table 2-16 right shifts the dividend by one to adjust for this before beginning the divide operation.



REFER TO TABLE 2-16 DIVIDE SEQUENCE STATES

TK-0270

Figure 2-18 Divide Sequence Hardware

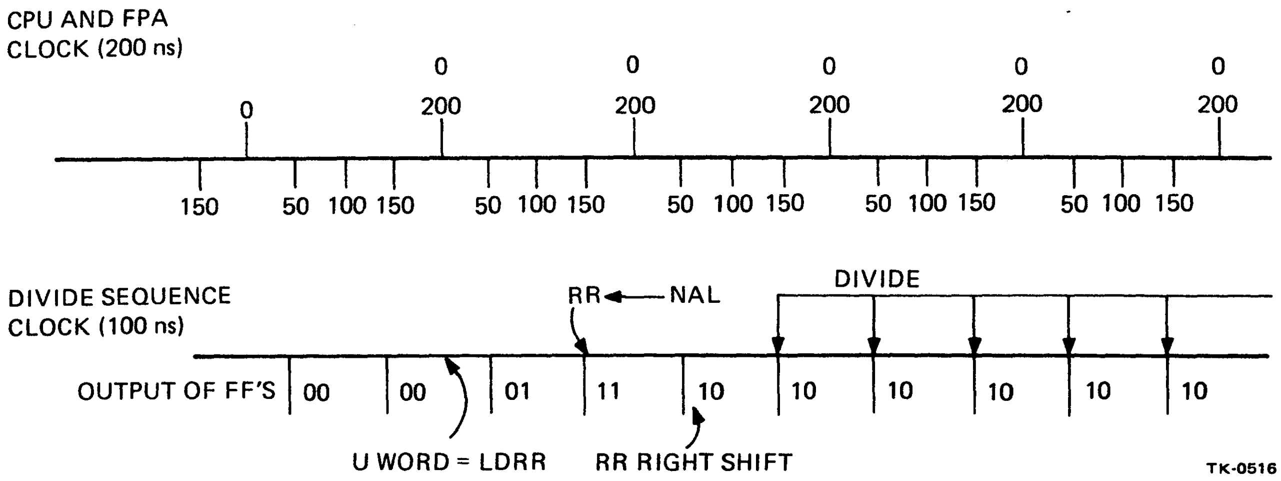


Figure 2-19 Divide Sequence Timing

Table 2-16 Divide Sequence States

State		Input	Next		FNM Function	RR CTL		RR Function
A	B		A	B		1	0	
0	0	LD RR	0	0	NOP	L	L	NOP
0	0	LD RR	0	1	NOP	L	L	
0	1	X	1	1	LD NALU TO RR	H	H	Parallel LD**
1	1	X	1	0	Shift R*	L	H	Shift R*
1	0	DIV DONE	1	0	Divide	H	H†	Parallel LD Result**
						H	L†	Shift L RR Contents
1	0	DIV DONE	0	0	Divide			Refer to PREVIOUS STATE

*Used only once at the beginning of each divide.

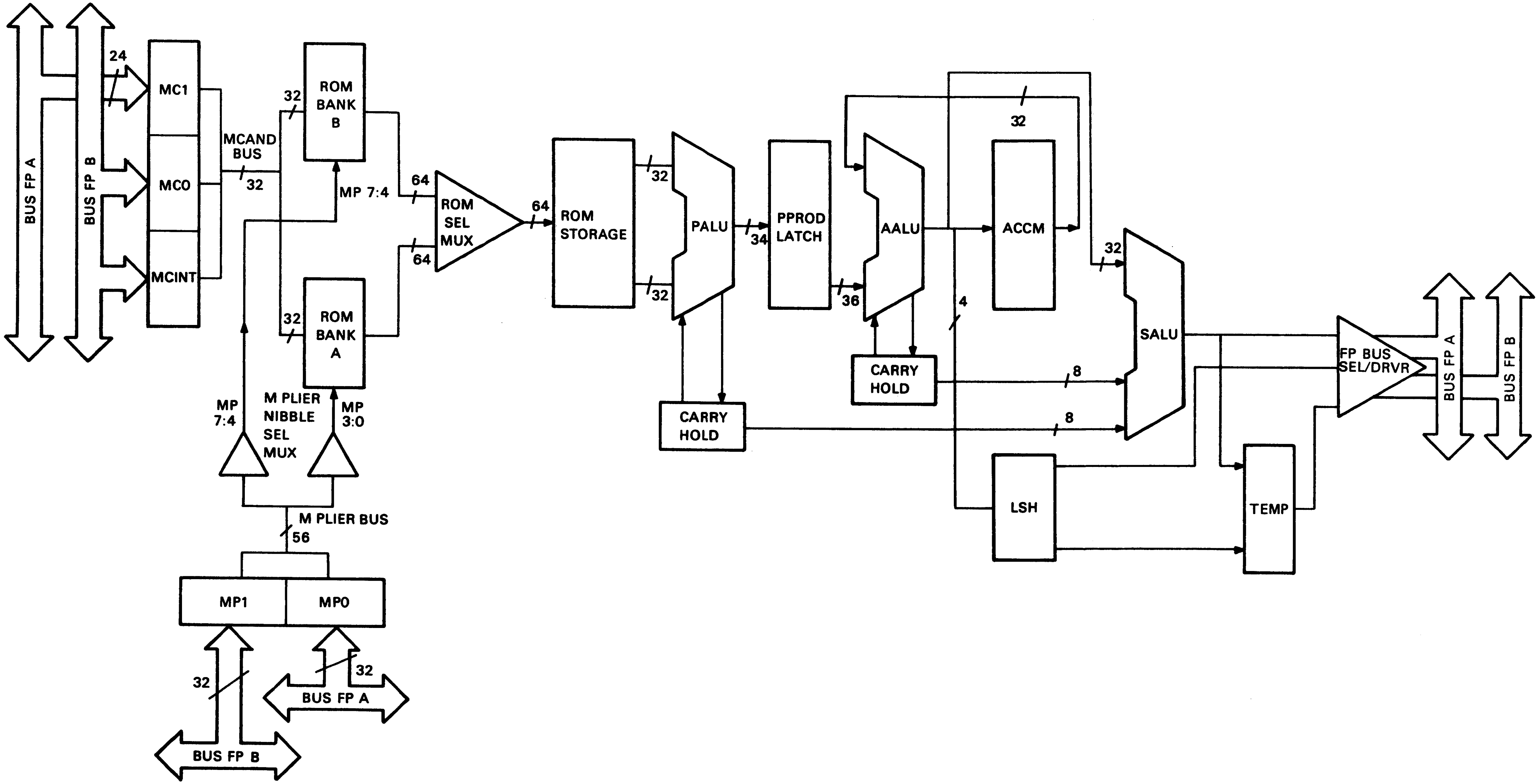
† Control bit 0 is controlled by RES POS H.

**Since the RR is hardwired for a left shift, a parallel load shifts the data one place left.

The answer is generated at the rate of one bit per 100 ns. If the result of the NALU subtract is positive or zero, a 1 is left-shifted into the quotient register. A negative NALU result causes a 0 to be shifted into the quotient register. The quotient register is made of two multiplier registers (TEMP and LSH). In single (float) precision the quotient bit stream is shifted into TEMP (use only TEMP <29:4>. Indouble precision the bit stream shifts into LSH <31:4> then to TEMP <29:00>. When a 1 is left-shifted into TEMP 29 or 28 on the proper time phase in the multiplier logic, DIV DONE is asserted. This stops the division and accesses a new microstore word that normalizes and rounds the quotient.

2.3.5 Fraction Multiplier (FML and FMH)

The fraction multiplier hardware in the FPA is located on two modules, FMH (Fraction Multiplier High) and FML (Fraction Multiplier Low). They handle all fraction multiply functions, part of the EMOD function, and also store the division quotient as it is generated. It accepts data from the FP buses, performs the required unsigned multiplication, and gates the results back on the FP buses. Refer to Figure 2-20.



TK-0278

Figure 2-20 Fraction Multiplier Block Diagram

The FPA microcontrol controls the loading of both the multiplicand and multiplier into the appropriate FM (fraction multiplier) registers. In both float and double the complete multiplier is stored on the FMH. During the single precision (float) function, the FMH handles the upper 16 bits of the multiplicand, FML the lower 8 bits and the answer is completed after one pass through the logic. For double precision (56 bits) the upper half of multiplicand fraction is handled in the FMH and the lower half is handled in the FML. Two passes are required to compute the final answer.

The FM multiplies under its own control logic. After the operands are loaded, the MCTL field in the FPA microcontrol is asserted; this starts the multiplication. A float multiply is stopped by the microcode two states (400 ns) after it starts. For a double multiply, control goes to a wait state and remains at that location until MUL/DIV DONE is enabled, indicating that the FM logic has finished the operation. At this point microstore control takes over and the answer is transmitted to the normalize logic or, in the case of EMOD or MULL, transmitted to the CPU as an unnormalized number.

In order to obtain fast multiplication, a pipeline technique is used (Figure 2-21). The multiplier is divided into 4-bit nibbles. The nibbles are then accessed consecutively by a counter-multiplexer combination (least significant nibble first) and each nibble operates on up to 32 bits of multiplicand. The MCAND bus and MPLIER nibbles are used to address the ROMs. The banks of ROMs provide a 4 × 4 primitive with 2-way interleaving. The data is latched (ROM STORE) and applied to the inputs of 4-bit adders (PALU). These adders combine the ROM data to form a partial product, storing the carry-out of each 4-bit section, to be added in on the next cycle. The partial product is latched in PPROD and passed to another row of adders (AALU) which accumulate the final product, again, saving the carries. Thus, when the pipeline is operating, there are four processes cycling at the same time:

1. Select ROM addresses
2. Latch ROM data
3. Form partial product
4. Accumulate final product.

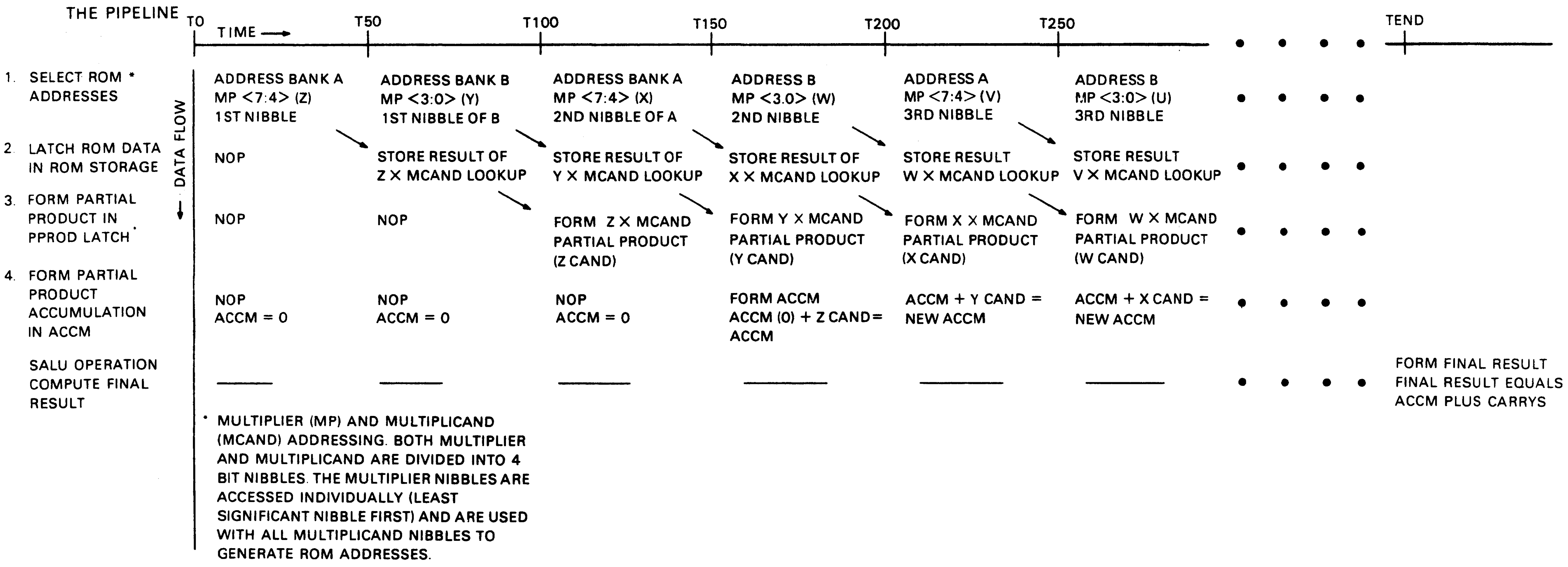
After the final product is calculated, the stored carriers from both stages are combined with the accumulated product using full carry look-ahead to produce the final answer in a single precision (float) operation. In double precision, this result is stored and used during the generation of the final answer during the second pass.

Each of the pipeline processes, with the exception of accessing ROM data (which occurs in each bank of ROMs on 100 ns) occurs at 50 ns intervals.

The operation of the FM hardware is discussed in three sections. The first section explains the operation of the pipeline, concentrating on operand loading and manipulation of partial products, partial results, and carries to produce the final answer. The second section concentrates on the control logic and how the signals that control the pipeline are generated. The third, and shortest section, explains how the FM registers are used to accumulate the quotient during a divide operation.

2.3.5.1 The Pipeline

Loading the Operands
The multiplication process begins with the loading of the operands. As discussed in Paragraphs 2.1 and 2.3.2, data is transferred along the FPA buses in several formats. The multiplicand loading logic sorts out these formats and loads the multiplicand register (MC0, MC1, and MC I) so that when the MCAND bus does a parallel access of the MCAND, the MSB of the multiplicand is always in MCAND bus bit 31, and each following bit is progressively less significant (Figures 2-22 and 2-23).



THE PIPELINE	T0 TIME →	T50	T100	T150	T200	T250	• • • •	TEND
1. SELECT ROM * ADDRESSES	ADDRESS BANK A MP <7:4> (Z) 1ST NIBBLE	ADDRESS BANK B MP <3:0> (Y) 1ST NIBBLE OF B	ADDRESS BANK A MP <7:4> (X) 2ND NIBBLE OF A	ADDRESS B MP <3.0> (W) 2ND NIBBLE	ADDRESS A MP <7:4> (V) 3RD NIBBLE	ADDRESS B MP <3:0> (U) 3RD NIBBLE	• • • •	
2. LATCH ROM DATA IN ROM STORAGE	NOP	STORE RESULT OF Z X MCAND LOOKUP	STORE RESULT OF Y X MCAND LOOKUP	STORE RESULT OF X X MCAND LOOKUP	STORE RESULT W X MCAND LOOKUP	STORE RESULT V X MCAND LOOKUP	• • • •	
3. FORM PARTIAL PRODUCT IN PPROD LATCH*	NOP	NOP	FORM Z X MCAND PARTIAL PRODUCT (Z CAND)	FORM Y X MCAND PARTIAL PRODUCT (Y CAND)	FORM X X MCAND PARTIAL PRODUCT (X CAND)	FORM W X MCAND PARTIAL PRODUCT (W CAND)	• • • •	
4. FORM PARTIAL PRODUCT ACCUMULATION IN ACCM	NOP ACCM = 0	NOP ACCM = 0	NOP ACCM = 0	FORM ACCM ACCM (0) + Z CAND= ACCM	ACCM + Y CAND = NEW ACCM	ACCM + X CAND = NEW ACCM	• • • •	
SALU OPERATION COMPUTE FINAL RESULT	———	———	———	———	———	———	• • • •	FORM FINAL RESULT FINAL RESULT EQUALS ACCM PLUS CARRYS

* MULTIPLIER (MP) AND MULTIPLICAND (MCAND) ADDRESSING. BOTH MULTIPLIER AND MULTIPLICAND ARE DIVIDED INTO 4 BIT NIBBLES. THE MULTIPLIER NIBBLES ARE ACCESSED INDIVIDUALLY (LEAST SIGNIFICANT NIBBLE FIRST) AND ARE USED WITH ALL MULTIPLICAND NIBBLES TO GENERATE ROM ADDRESSES.

TK-0529

Figure 2-21 The Pipeline

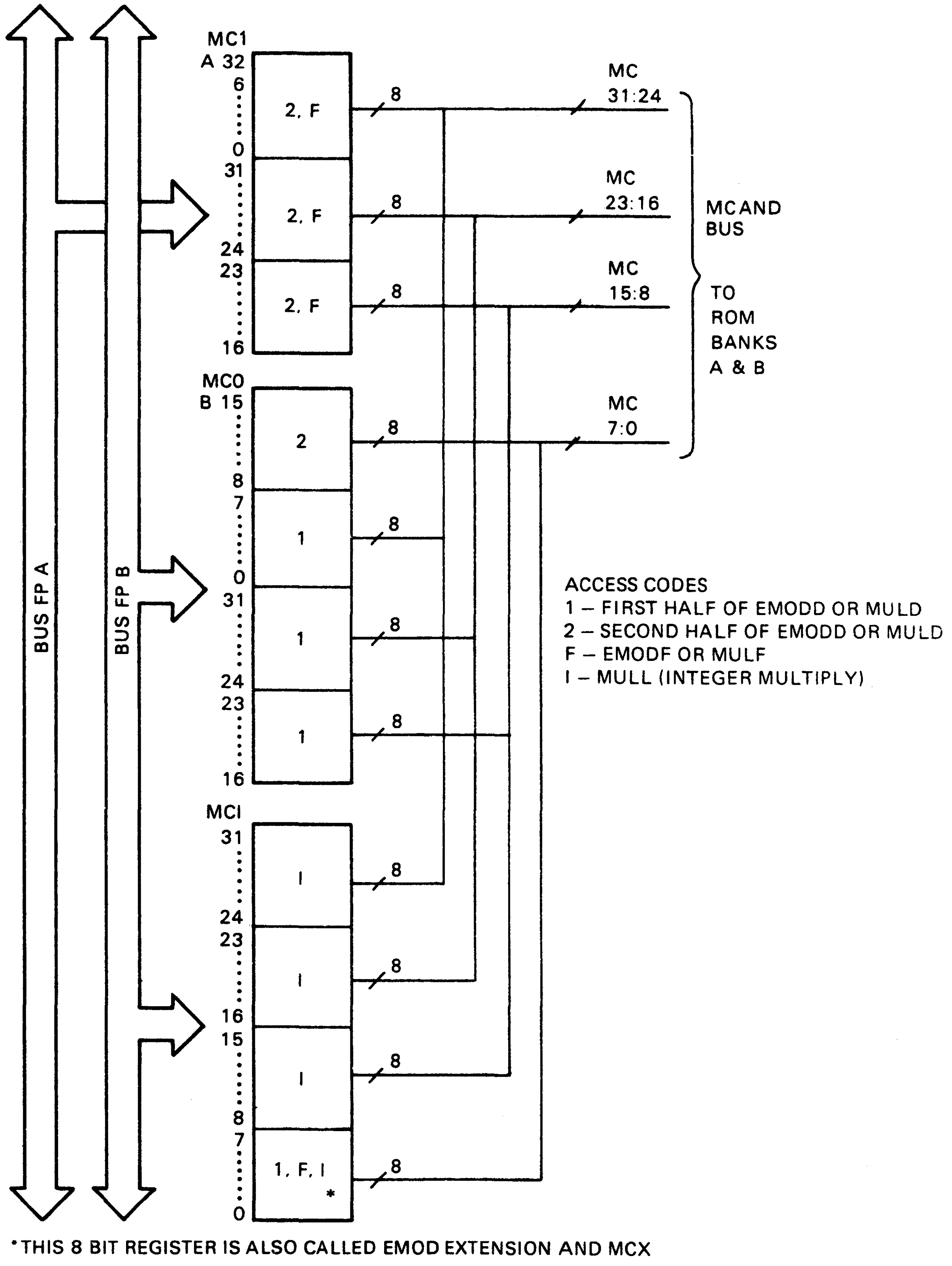


Figure 2-22 Loading and Accessing the Multiplicand

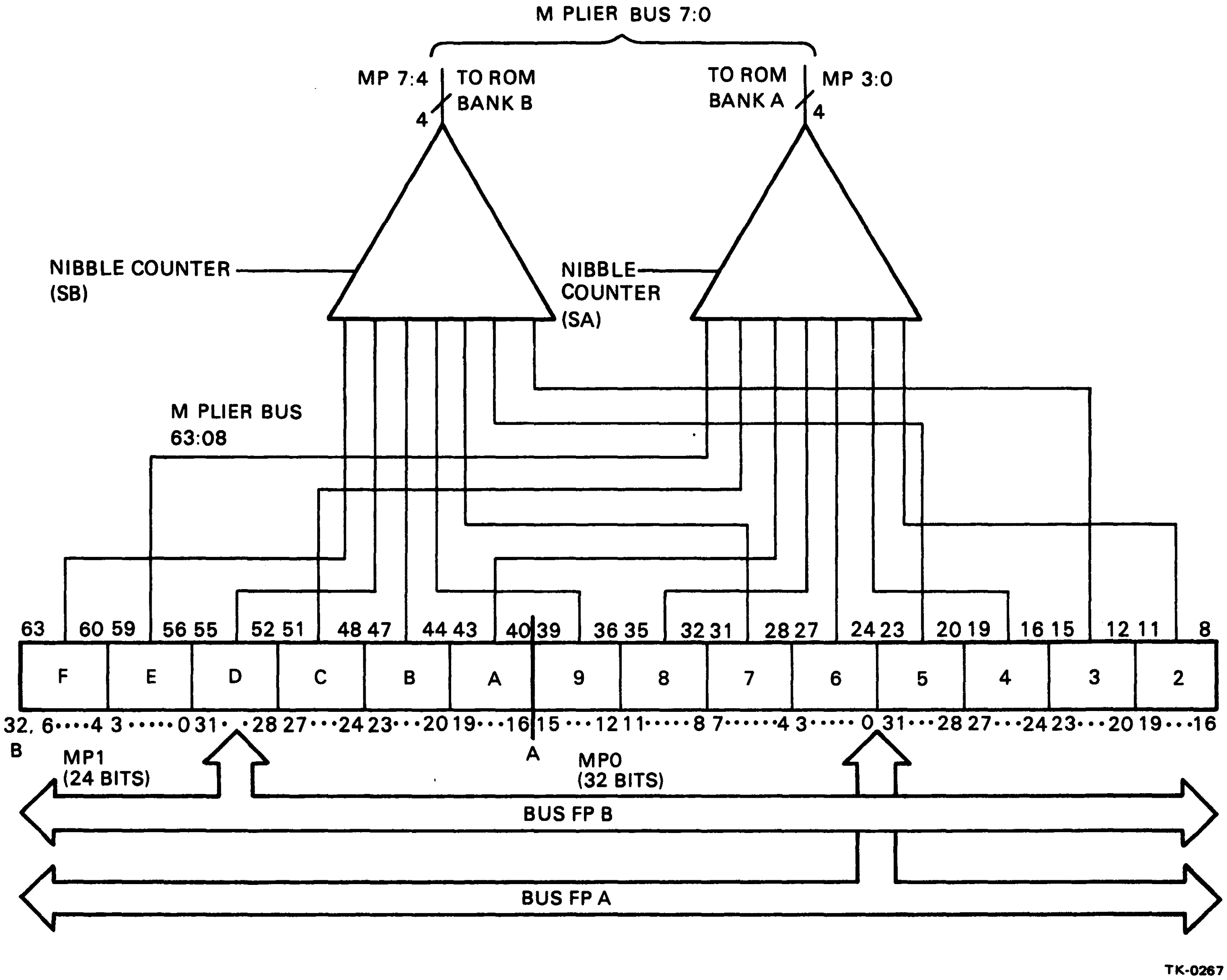


Figure 2-23 Loading and Accessing The Multiplier

The multiplier up to 56 bits (14 nibbles) long, is loaded into MP1 and MP0 on FMH. MP1 is 24 bits (6 nibbles) long and MP0 is 32 bits (8 nibbles) long. Unlike the multiplicand, the multiplier is loaded in one format only (Figure 2-23). The MSB is in MP1–23 and each following bit is progressively less significant. The LSB is MP1–00 for single precision (float) or MP0–00 for double precision. The single format is possible because, as stated before, the multiplier is used consecutively, the various formats are sorted out by the counter as the nibbles are used during the multiplication.

Selecting the Multiplicand
The operands, multiplicand and multiplier, are enabled onto their respective buses, MCAND BUS and MPLIER BUS, under control of operand bus source logic. Refer to Figures 2-22 and 2-23 and Table 2-17. All 32 lines of the MCAND bus are enabled every time. During a MULF and EMOD and for the first pass of a MULD and EMODD, the MCAND bus accesses MCX. Both MULF and MULD (first pass) use only the top 24 bits, as the lower 8 are discarded later in the pipeline.

The MPLIER BUS multiplexer begins by selecting the least significant byte of the multiplier. Interleaving hardware later selects the high or low nibble of the bus. The mux then selects a new, progressively more significant byte each 100 ns.

Selecting ROM Address – The Interleave Hardware
Both the MCAND and MPLIER buses are divided into 4-bit nibbles for ROM addressing. Each MCAND nibble (8 nibbles) is combined with a MPLIER nibble to provide address bits for 16 4X4 look-up ROMs. Rather than compute the product of the two 4-bit nibbles, the fraction multiply hardware uses look-up ROMs. The multiply results are stored in the ROMs. The data is stored within the ROMs such that the content of the address accessed by the two nibbles is the 8-bit result of a multiply with the same two nibbles. Since the ROMs are relatively slow the 16 ROMs are divided into two interleaved 8 ROM banks. One bank is accessed by the low MPLIER nibble (MP 3:00) theotherby the high MPLIER nibble (MP 7:4). Both ROMs are addressed on 100 ns cycles; the MP low ROM is first, and the MP high is second, trailing by 50 ns. The addressing of a ROM bank ends the first part of the pipe.

Latch the ROM Data
The second part of the pipe selects the outputs from either of the ROM banks, using the ROM SEL MUX, and latches the data (64 bits) in ROM STRG. It alternately selects data from the low and high ROM banks on a 50 ns cycle.

While the ROM data selected is being latched, the first part of the pipe is selecting a new address for the ROM bank just selected. The output of the other ROM bank will be selected during the next cycle (50 ns in the future). The address lines of this ROM bank were changed 50 ns ago and the outputs are settling.

Form Partial Product
The outputs of ROM STRG and any carrys from the previous PALU add are added to form the partial product. The PALU is eight 4-bit adders. The outputs of the ROM STRG are wired to the PALU adder inputs such that bits of equal significance are combined. The outputs of the PALU without carrys are stored in the PPROD LATCH. The carrys are stored in CARRY-HOLD registers to be added in on the next PALU add. The latching of the partial products in the PPROD LATCH ends the third part of the pipeline.

As indicated previously each multiply cycle selects 4 new bits from the multiplier register and each 4 new bits are 4 positions more significant. This means that the input of the PALU add becomes 4 bits more significant each multiply cycle. Because of the increase in significance the stored carry-out of each PALU adder is input, on the next cycle, to the carry-in of the same PALU adder rather than the carry-in of the next PALU adder.

Table 2-17 Operand Bus Source

	Input Signals			MCAND Bus Load Enable*					MPLIER BUS
Operation	DOUBLE	TTH	OPC7	MC1	MC1L	MC0	MCINT	MCX	Nibble Select
EMODF or MULF	L	X	L	L				L	Start at A, do 6 nibbles
MULL (INTEGER MUL)	X	X	H				L	L	Start at 6, do 4, then start at 2, do 4.
EMODD or MULD									
1st Pass	H	H	L			L		L	Start at 2, do 14
2nd Pass	H	L	L	L	L				Start at 2, do 14
	MCAND Bus lines fed			31–8	7–0	31–8	31–8	7–0	

*MCAND Bus lines are low enabled.

Note that while the third part of the pipeline is operating, new ROM data is being placed in ROM STRG to be presented to the PALU inputs on the next cycle, and new ROM addresses are being generated to access new data.

Accumulate Result

The fourth and final section, the AALU and associated accumulator (ACCM) adds the partial products computed in the previous pipeline section to the result stored in the ACCM including stored carries from the previous AALU cycle and latches the result into the ACCM and LSH register.

The AALU, ACCM, and ALU carry-hold interconnections automatically shift the ACCM content and ALU carry-hold content to adjust for the 4-bit increase of each new partial product. Because each partial product input to the AALU is 4 bits more significant than the previously stored ACCM content, the outputs of the ACCM are wired to shift the ACCM content 4 bits right (a decrease in significance) before being added to the PPROD LATCH content. The lower 4 bits of the AALU output are always right-shifted into the LSH register. In double precision operations, the content of this register is the least significant half of the result.

As with the PALU carrys, the carry-out of each AALU is stored and added in on the next cycle. Also similar to the PALU logic, the stored carrys are added to the AALU adder that generated them because the content of the AALU is now 4 bits more significant than when the stored carrys were generated.

The latching of the accumulating final result in the ACCM ends the fourth pipeline section.

The 4 sections of the pipeline continue to operate until stopped by the FM control logic. The stopping point is selected based on both function and precision.

SALU OPERATION

When stop is initiated, the whole pipeline stops and new logic, the SALU, is accessed which adds the two sets of stored carrys still in the pipeline to the total product on the output of AALU. When a pipeline stop is initiated, the AALU output (SALU input) is the contents of ACCM plus the current PPROD. Both the ACCM plus PPROD addition (the AALU operation) and the PPROD forming addition (the PALU operation) form stored carrys.

The hard-wired 2-bit shift in the PPROD LATCH input is not part of the several 4-bit shifts that take place throughout the FM logic, but rather format the stored carrys so they may be easily combined for a final answer in the SALU. Both the PALU and AALU are composed of 4-bit adders with carry-outs. This means that the carry-outs are generated every 4 bits and that the PALU and AALU stored carry-outs can be treated as numbers of the following format:

X000X000X	X is a stored carry (data bit)
	0 is a zero (non-significant bit)

Conventional wiring (output of a 4-bit PALU adder to input of a 4-bit PPROD LATCH to a 4-bit AALU adder) would cause the data bits of the PALU stored-carry to line up (be of equal significance) with the AALU stored-carry. This would prevent PALU stored-carrys, the AALU stored-carrys, and the ACCM result from being combined in one operation in one adder (the SALU). However, wiring the PPROD LATCH input and outputs with a 2-place shift, generates a PALU stored-carry number with data bits of significance between the AALU stored-carry data bits. This shift allows both AALU and PALU stored-carry numbers to be input to one side of the SALU, since the data bit of the PALU stored-carry is always a non-significant bit of the AALU stored-carry and vice versa. Refer to Figure 2-24.

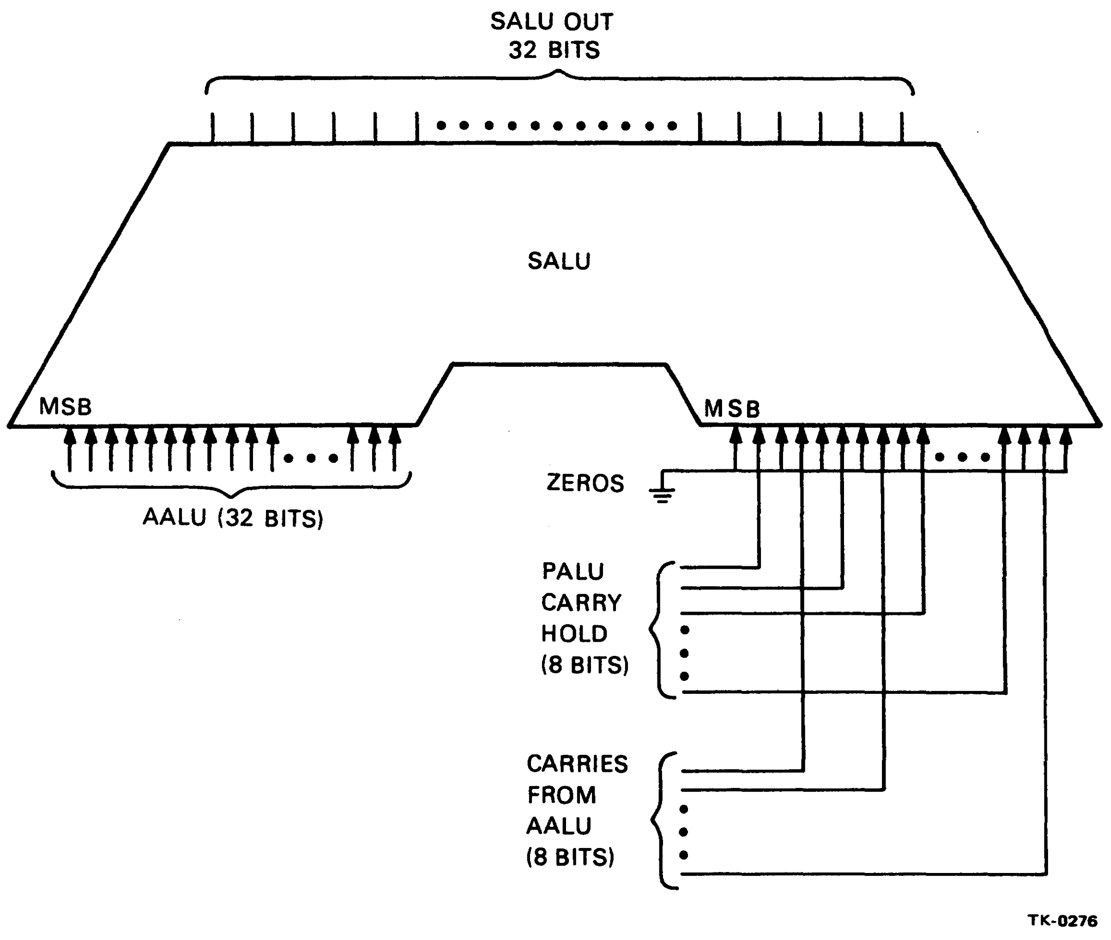


Figure 2-24 SALU Operation - Adding the Stored Carrys

The use of the SALU result is determined by operation and the operation precision. If the SALU result is the final answer, the result is transferred to the FP buses under both op code control and FPA microcontrol. If however, the operation is double precision, the result is stored, and then, shifted to format it for later operations under FM logic control. Before the shift, the most significant half of the operation is in TEMP, the least significant half in LSH. The shift transfers the contents of LSH (the least significant half) to the ACCM register which is designated ACCM 14 at this time, and transfers the most significant half from TEMP to (just vacated) LSH.

For the second pass, the second half (the more significant half) of the multiplicand is accessed from register MC1 and MC1L, and logic enabled only during the second pass, combines the data transferred to LSH from TEMP with the new result being accumulated. Otherwise, the operation of the pipeline during the second pass is the same as during the first pass.

2.3.5.2 FM Control – The fraction multiplier logic is hardware rather than firmware controlled. Four state bits select one of 13 function states that control the FM logic. Within each state, the state bits, various internal flags, and various flags from other FPA logic are combined to provide the control signals needed to implement the selected state's functions (Figure 2-25 and Table 2-18).

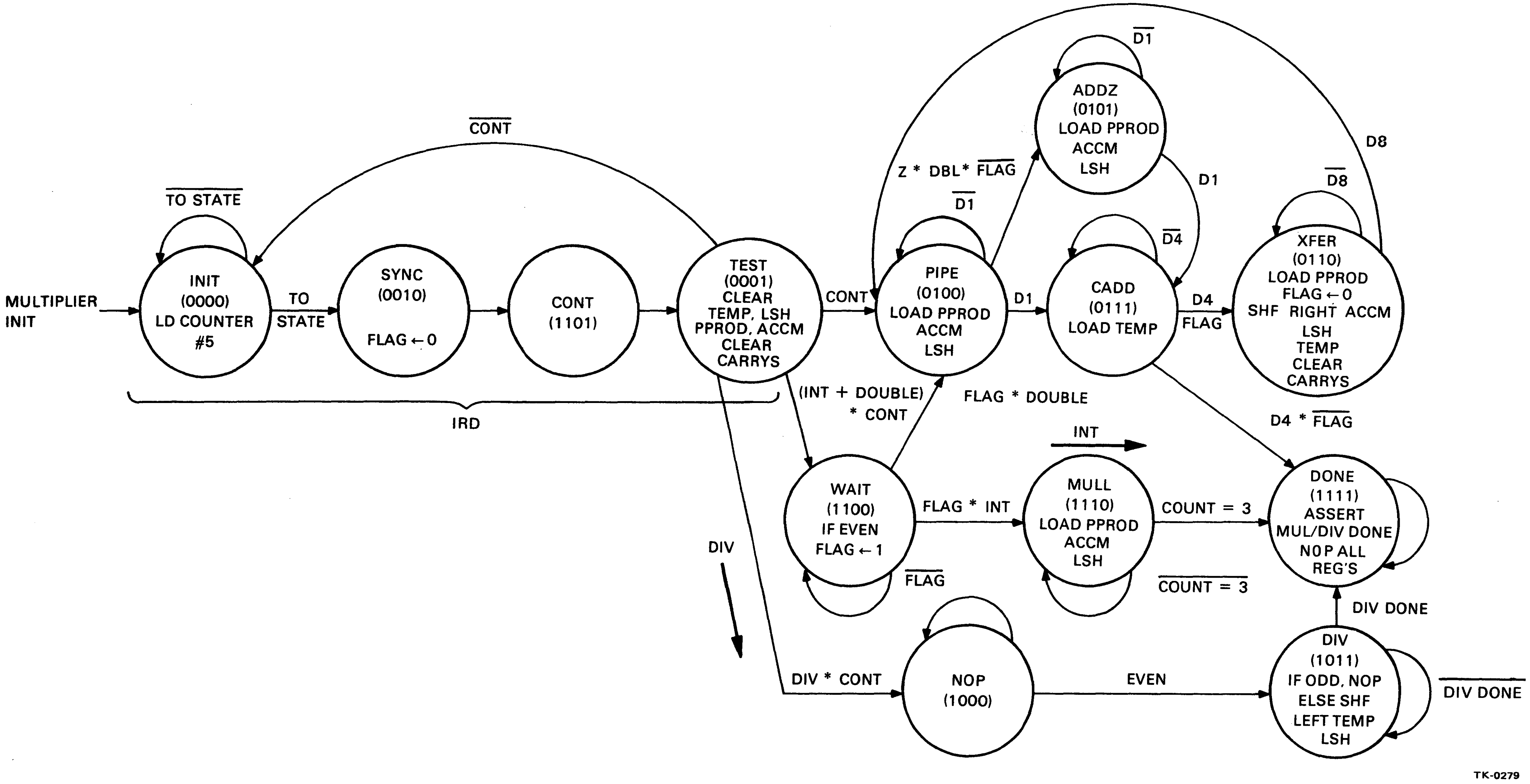

MULTIPLIER INIT
INIT (0000) LD COUNTER #5
$\overline{\text{TO STATE}}$
TO STATE
SYNC (0010) FLAG ← 0
CONT (1101)
$\overline{\text{CONT}}$
TEST (0001) CLEAR TEMP, LSH PPROD, ACCM CLEAR CARRYS
IRD
CONT
PIPE (0100) LOAD PPROD ACCM LSH
$\overline{\text{D1}}$
Z * DBL* $\overline{\text{FLAG}}$
ADDZ (0101) LOAD PPROD ACCM LSH
$\overline{\text{D1}}$
D1
D1
CADD (0111) LOAD TEMP
$\overline{\text{D4}}$
D4
FLAG
XFER (0110) LOAD PPROD FLAG ← 0 SHF RIGHT ACCM LSH TEMP CLEAR CARRYS
$\overline{\text{D8}}$
D8
(INT + DOUBLE) * CONT
FLAG * DOUBLE
D4 * $\overline{\text{FLAG}}$
INT
WAIT (1100) IF EVEN FLAG ← 1
$\overline{\text{FLAG}}$
FLAG * INT
MULL (1110) LOAD PPROD ACCM LSH
COUNT = 3
$\overline{\text{COUNT = 3}}$
DONE (1111) ASSERT MUL/DIV DONE NOP ALL REG'S
DIV DONE
DIV
DIV * CONT
NOP (1000)
EVEN
DIV (1011) IF ODD, NOP ELSE SHF LEFT TEMP LSH
$\overline{\text{DIV DONE}}$
TK-0279

Figure 2-25 FM Control States

Table 2-18 FM Control States

STATE VARIABLES	NAME	NEXT STATE	DEFINITION	OUTPUT CONTROL												
X3 X2 X1 X0				LD CNTR	CNTR CONSTANT	NEXT TTH	NEXT ALU ADD	NEXT FLAG	NEXT CLR ALL	MUL/DIV DONE	PPROD	ACCM	LSH	TEMP
0 0 0 0	INIT	IF $\overline{\text{T0}}$, THEN 0000; ELSE 0010	RESULT OF MINIT SIGNAL FROM MICROCODE. PREPARES MPLIER NIBBLE SELECT COUNTER FOR MULF SEQUENCE.	1	1010	0 *	1 IF $\overline{\text{FLAG}}$ AND DOUBLE	PREV FLAG *	1	0	NOP *	NOP *	NOP *	NOP *
0 0 1 0	SYNC	1101	ENTRY FROM STATE 0000 AT T50 TO PROVIDE SYNCRONIZATION BETWEEN MULTIPLIERS 50ns. CLOCK AND MICROCODES 200ns. CLOCK	0	1010 *	0	1 IF $\overline{\text{FLAG}}$ AND DOUBLE *	0	1	0	LD *	SR *	SR *	SR *
1 1 0 1	CONT	0001	NOP IF MULF OR EMODF; LOAD MPLIER NIBBLE SELECT COUNTER IF MUL, MULD, OR EMODD.	1 IF DOUBLE OR INT	0110 IF INT ELSE 0010	1 IF DOUBLE ELSE 0	0 *	PREV FLAG	0	1 *	LD *	LD *	NOP *	NOP *
0 0 0 1	TEST	IF $\overline{\text{CONT.}}$, THEN 0000 ELSE IF DIV, THEN 1000; ELSE IF DBL. OR INT., 1100; ELSE 0100	TESTS OPCODE FOR FIRST EXECUTION STATE CALCULATION; CLEARS THE MULTIPLIER DATA PATH,	0	1010 *	PREV TTH	0 *	PREV FLAG	1	0	NOP *	NOP *	SL IF EVEN LD IF EVEN AND FLAG	LD *
1 0 0 0	NOP	IF $\overline{\text{EVEN}}$ THEN 1000; ELSE 1011	WAITS FOR FIRST QUOTIENT BIT TO BE FORMED IN THE NALU.	1 *	1010	0 *	0	PREV FLAG	0	0	NOP	NOP	NOP	NOP *
1 0 1 1	DIV	IF $\overline{\text{DIV DONE}}$, THEN 1011; ELSE 1111	SHIFTS LSH AND TEMP LEFT ONE BIT TO ACCEPT QUOTIENT BITS IN DIVIDE	1 IF D3 AND DBL OR INT *	1110 IF INT ELSE 1010	1 IF DBL ELSE PREV TTH	0	PREV FLAG *	0	0	NOP *	NOP *	SL IF EVEN	SL IF EVEN
1 1 0 0	WAIT	IF $\overline{\text{FLAG}}$, THEN 1100 ELSE IF DBL, THEN 0100; ELSE 1110	CLEAR DATA PATH AND CARRY REGISTERS FOR MULD, EMODD, AND MULL. WAITS FOR FIRST ROM LOOK UP.	1 IF INT AND FLAG	0010	PREV TTH	0	1 IF EVEN ELSE PREV FLAG	1 IF EVEN	0	LD	LD	LD	NOP *
1 1 1 0	MULL	IF $\overline{\text{COUNT}}$=3, THEN 1110 ELSE 1111	RUNS MULTIPLIER PIPE FOR MULL.	0	0010 *	PREV TTH *	0	0	0	0	LD	LD	LD	SR *
0 1 0 0	PIPE	IF SHF ZEROES, AND DBL. AND $\overline{\text{FLAG}}$ THEN 0101 ELSE IF D1, THEN 0111; ELSE 0100	RUNS MULTIPLIER PIPE FOR FLOATING POINT MULTIPLY OPERATIONS. LSH'S 4 LSB'S TO ACCM'S 4 MSB'S IF SECOND TIME THROUGH DBL MULTIPLY.	1 IF INT AND FLAG	0010 *	PREV TTH	1 IF $\overline{\text{FLAG}}$ AND DOUBLE	PREV FLAG	0	0	LD	LD	LD	NOP *
0 1 1 1	CADO	IF $\overline{\text{D4}}$, THEN 0111 ELSE IF FLAG, THEN 0110 ELSE 1111	STOPS PIPE TO ADD FINAL STORED CARRYS TO FINAL ACCUMULATION. LOADS TEMP.	1 IF D3	0010	0	0	PREV FLAG	0	1 IF $\overline{\text{FLAG}}$	NOP	NOP	LD IF EVEN AND FLAG	LD
0 1 1 0	XFER	IF $\overline{\text{D8}}$, THEN 0110 ELSE 0100	SHIFTS ACCM, TEMP, AND LSH RIGHT TO TRANSFER.	0	0010 *	PREV TTH	1 IF DOUBLE	0	0	0	LD	SR	SR	SR
0 1 0 1	ADDZ	IF $\overline{\text{D1}}$, THEN 0101 ELSE 0111	ADDS ZEROES TO ACCM'S 4 MSB'S.	0	0010	0	0	PREV FLAG	0	0	LD	LD	LD	LD *
1 1 1 1	DONE	1111	STOPS ALL REGISTERS FROM CHANGING TO ALLOW NR OR CPU D REG. TO ACCEPT FINAL RESULT.	1 IF D3 AND DBL OR INT *	0110 IF INT ELSE 0010 *	1 IF DOUBLE *	0	PREV FLAG	0	1	NOP	NOP	NOP	NOP

* DON'T CARE

TK-0735

The states can be roughly divided into four groups:

1. IRD
2. Integer Multiply
3. Fraction Multiply
4. Divide.

This section will discuss the states by groups and in the previously shown order. Within each discussion, the states will be discussed in the order they are accessed within the group. This is important because the function of some states is partially dependent on the previous state.

The state of the logic is defined by the output of the PRESENT STATE register which is clocked on a 50 ns cycle. The inputs to this register (the next state) are based on the current state and internal and external flags. A majority of the internal flags provide sequence information and are generated in the logic shown in Figure 2-26.

IRD Group (Instruction Register Decode)
When the FM logic is not performing a multiply or divide operation, it is in IRD. While waiting, the logic is continually cycling through the 4 states in this group preparing the FM logic for a multiply. In this IRD group the op codes in the instruction buffer are monitored. Initially, (in INIT), the FM logic is set up for a MULF, but if the op codes indicate either a MULL, MULD, or EMODD, new information is loaded into the FM logic in the CONT state. The FPA microcontrol will be loading the MPLIER and MCAND register during IRD if the op codes indicate a multiply operation.

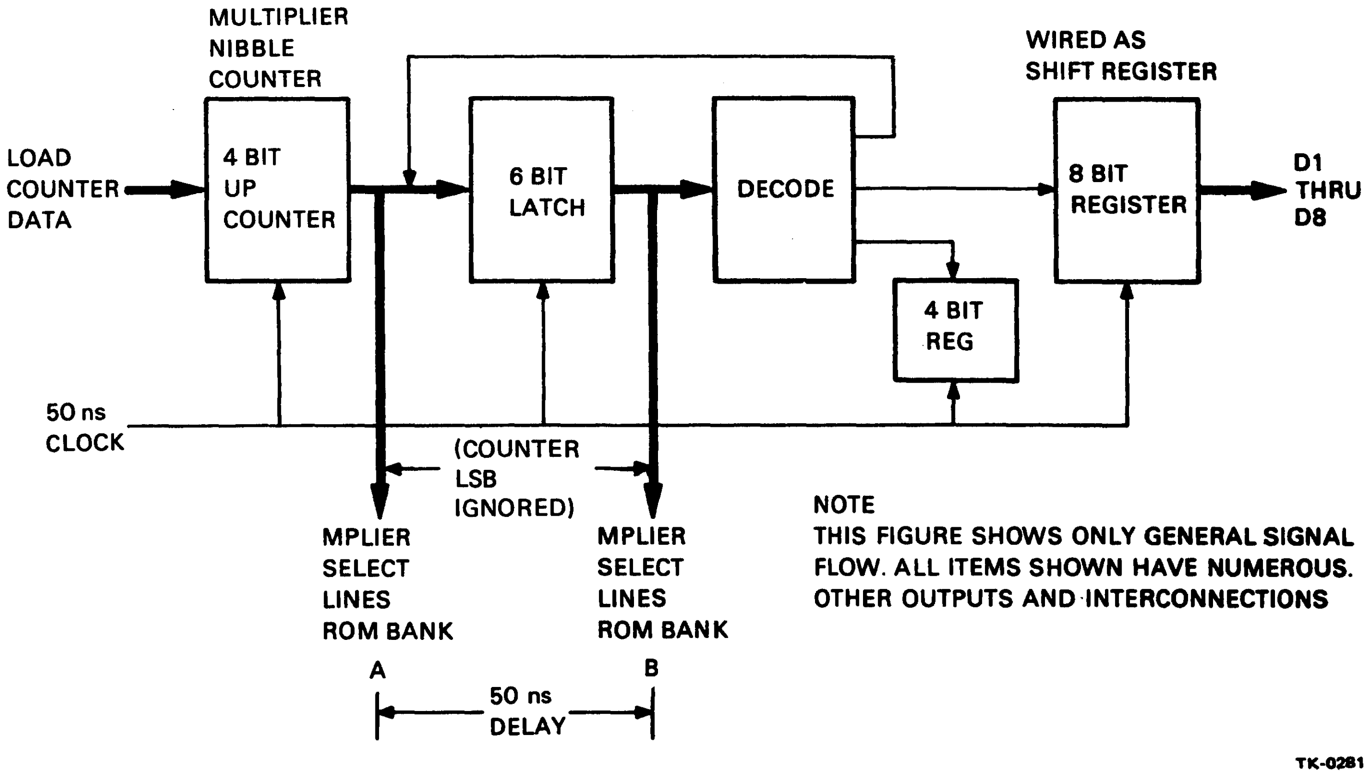
The control logic enters INIT whenever the Multiplier Operand Control (OPLD) field in the FPA microcontrol store is F. This normally happens during the FPA IRD or when a multiply operation is finished. The SYNC state is entered at CPU T50 and synchronizes the FM clock with the CPU clock. It also clears FLAG. CONT is entered at T100 and loads new information if the op codes indicate a MULL, MULD or EMODD. TEST is entered at T150. In TEST, if the MCNT bit in the FPA microcode is not asserted, indicating that the FPA does not want the multiply pipeline to begin, the FM returns to the INIT state and continues waiting. If however, MCNT is asserted, indicating that the multiplier operands are loaded and the FPA wants a multiply to start, the correct execution state is selected based on the op code. Refer to Table 2-18 for summary of IRD group functions.

Multiply Float Path
If the op code indicates a MULF, the PIPE state is selected and the multiplier pipe can continue. Note that during INIT the nibble counter was loaded with MULF control data for ROM look-up to begin based on that data. Since a MULF is being done, the data in the beginning of the pipe is correct.

The logic remains this state (PIPE), running the pipe and accumulating the answer, until D1, a timing signal, is asserted. When D1 is asserted the current content of the PPROD plus ACCM plus the stored-carrys is the final correct answer.

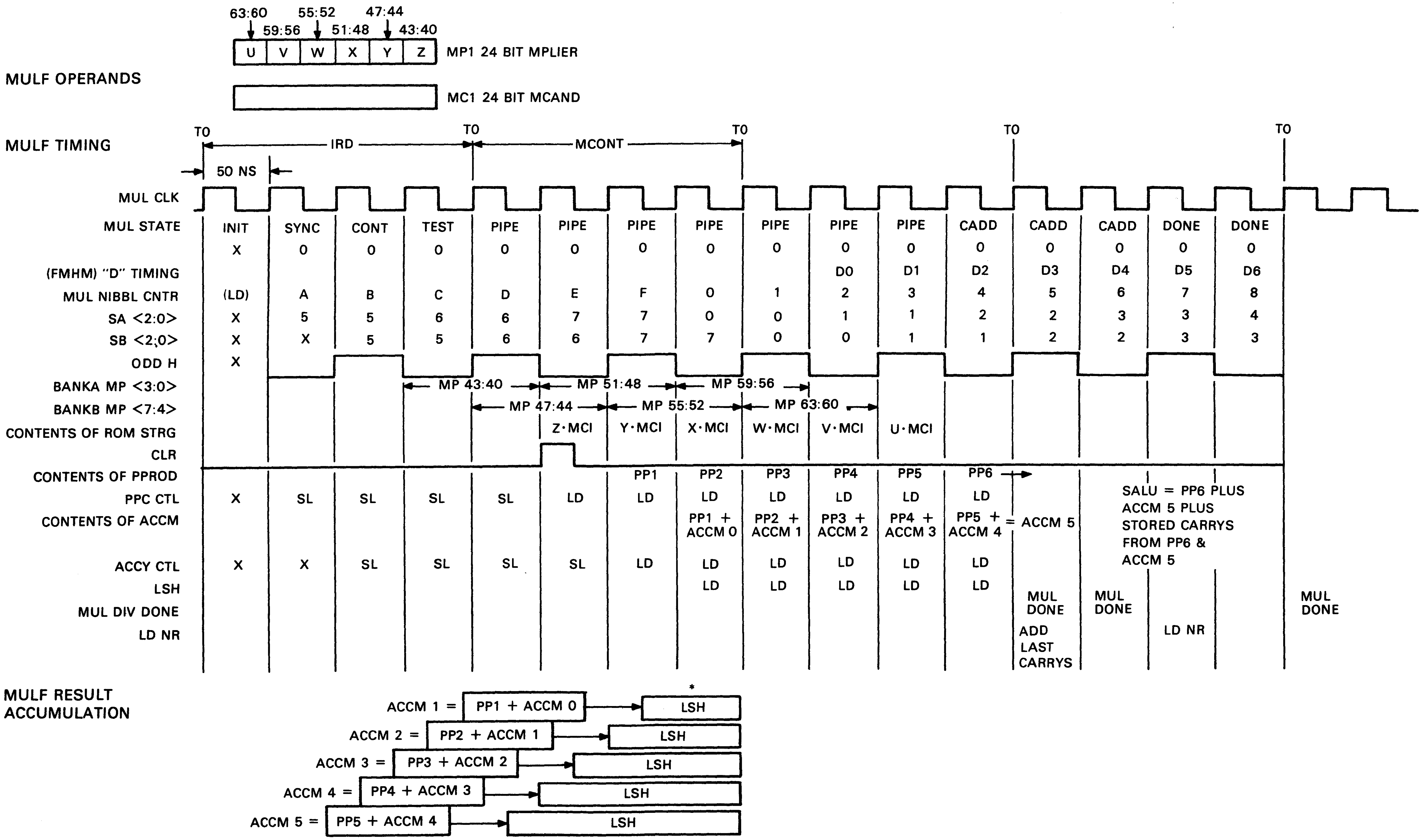
Asserting D1 selects the CADD state. This state NOPS most of the FM registers and enables the SALU add of stored-carrys to the AALU content. CADD also latches the SALU result into TEMP. The FM logic remains in CADD 150 ns (until D4 is asserted.)

Since FLAG was cleared during the IRD group and never set, it is clear and asserting D4 initiates the DONE state. This state asserts MUL/DIV DN and NOPs all other FM logic. MUL/DIV DN, monitored by the FPA control logic, returns control to the FPA microcontrol. It is the FPA control store that selects the MULF result, via a multiplexer, directly from the SALU outputs rather than from TEMP. The FM logic will remain in DONE until returned to INIT by the multiplier INIT code in the multiplier operand control field of the FPA microcontrol store. Refer to Figure 2-27 for a summary of MULF control.



NOTE
THIS FIGURE SHOWS ONLY GENERAL SIGNAL FLOW. ALL ITEMS SHOWN HAVE NUMEROUS. OTHER OUTPUTS AND INTERCONNECTIONS

Figure 2-26 FM Control Logic



* AFTER EACH ADDITION OF THE PARTIAL PRODUCT AND ACCUMULATOR CONTENTS, THE 4 LEAST SIGNIFICANT BITS OF THE RESULT ARE LOADED INTO THE LSH REGISTER.

TK-0512

Figure 2-27 MULF Control

MULD Path

If, when the FM control logic is in TEST, the op codes indicate a double precision multiply (DOUBLE set), the WAIT state will be entered. Initially (in INIT) the nibble counter was loaded for MULF and ROM lookup began, then in CONT (100 ns later) when a MULD was decoded, new data was loaded into the nibble counter. The WAIT state waits for the data loaded in CONT to settle and access new ROM locations before beginning the pipe. After 100 ns in this state FLAG is set. In this context, FLAG set indicates the first pass in a double precision multiply. After 150 ns, since both DOUBLE and FLAG are set, PIPE is entered.

The logic remains in the PIPE state, running the pipe and accumulating the answer until D1, a timing signal, is asserted. When D1 is asserted the current content of ACCM plus the two sets of stored-carrys are the first half of the MULD partial product.

Asserting D1 selects the CADD state. This state NOPs most of the FM registers and enables the SALU add of stored-carrys and the ACCM content. CADD latches the upper 32 bits of the first half of the MULD partial product in TEMP. The lower 32 bits have been accumulating in LSH during the pipeline operation. The FM logic remains in CADD 150 ns (until D4 is asserted).

Since FLAG is asserted, indicating first pass, asserting D4 selects the XFER state. Four cycles in the XFER state transfer the content of TEMP and LSH to LSH and ACCM (refer to Figure 2-28), clear FLAG, and clear the stored-carry registers.

The assertion of D8 returns the FM logic to PIPE. The FLAG bit now cleared and DOUBLE set asserts ALU ADD. This signal causes the data stored in LSH during the XFER state to be added in (4 bits per cycle) to the final product being developed. Six cycles transfer all 24 bits stored during XFER. While these bits are being right-shifted from the right end of LSH into the MSBs of the developing final product, the LSB of the developing final product are being right-shifted into the left end of the LSH.

When 20 bits have been transferred in from LSH, SHF ZERO is enabled. This causes the logic to enter the ADDZ state. The final 4-bit transfer of LSH data takes place during the first ADDZ state. After that the ALU that added LSH to the ACCM is disabled. During this state, the pipe continues functioning and the LSBs of the accumulating final product are still shifted into the left end of LSH. The only difference between PIPE and ADDZ during this second pass is, in PIPE, LSH data bits are added into the MSB of the ACCM, and, in ADDZ, zeros are added. Note this state even has the same ending criterion as PIPE, namely D1 asserted.

D1 asserted transfers control to the CADD state. As discussed in MULF path, CADD is entered when the ACCM plus the two sets of stored-carrys is the final answer. In CADD the stored-carrys are added to the AALU content by SALU and the result is latched into TEMP. Since FLAG is now clear the assertion of D4 causes a transfer to DONE.

In DONE, MUL/DIV DONE is asserted. This causes the FPA microcode to select and transfer, via multiplexers, the upper 32 bits of the double precision result from the SALU onto FP bus A and the lower 32 bits from the LSH register onto FP bus B. Refer to Figure 2-29 for a summary of MULD control.

MULL Path

If the op code being monitored during CONT decodes as MULL, new data is loaded into the nibble counter. The logic proceeds to TEST and, in TEST, selects the WAIT as the first execution state because INT (meaning integer) is set.

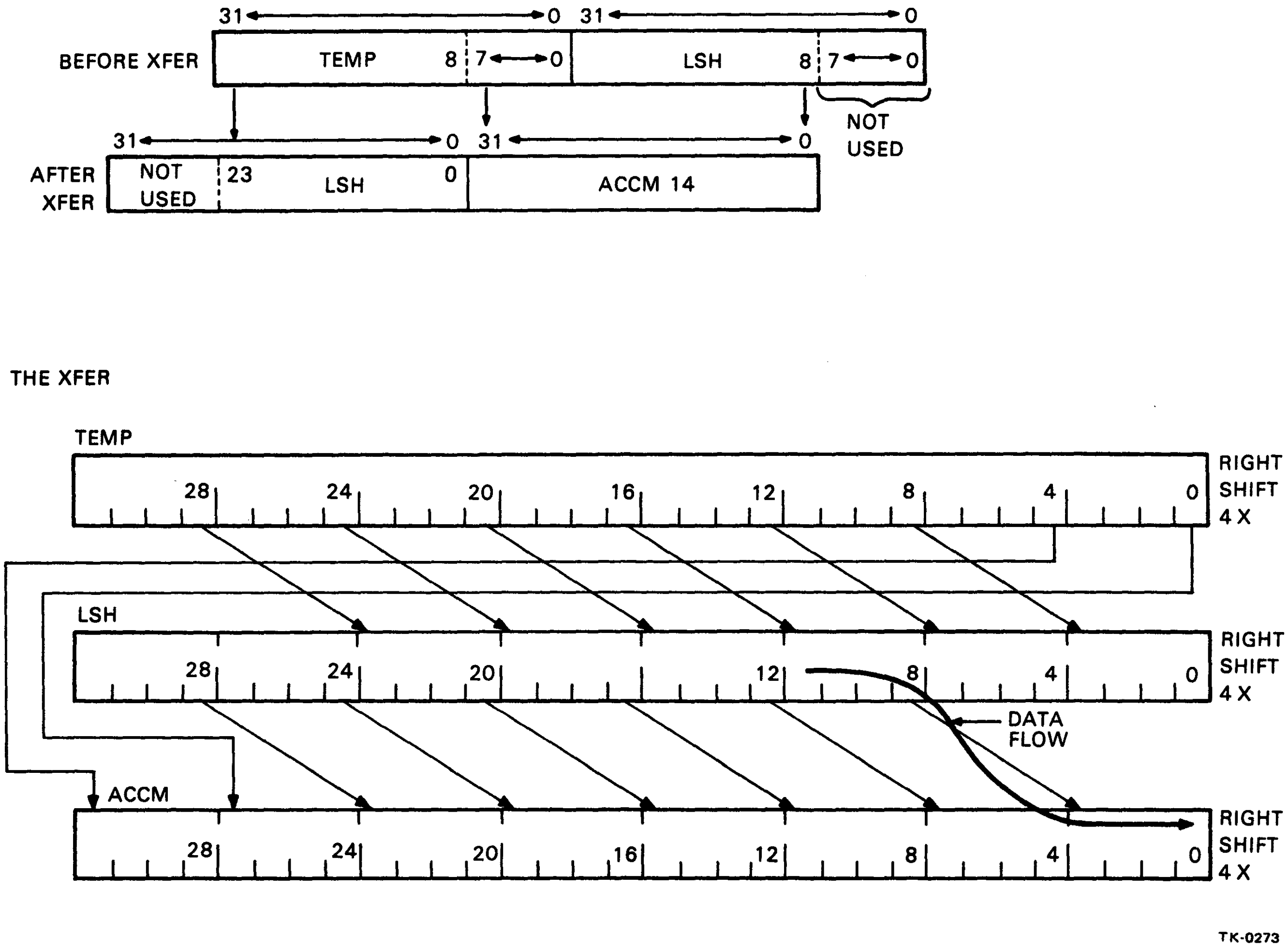
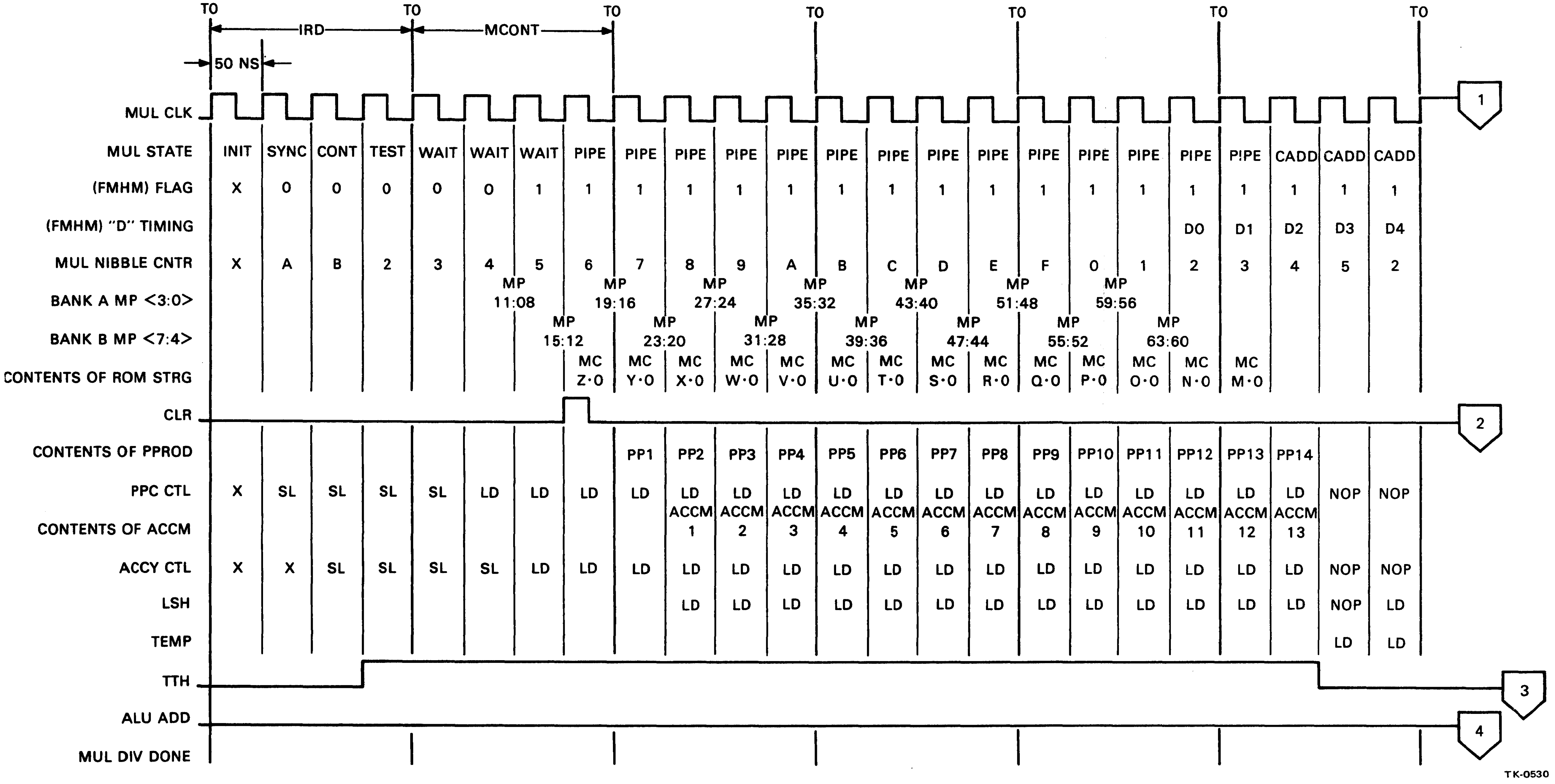


Figure 2-28 The XFER State

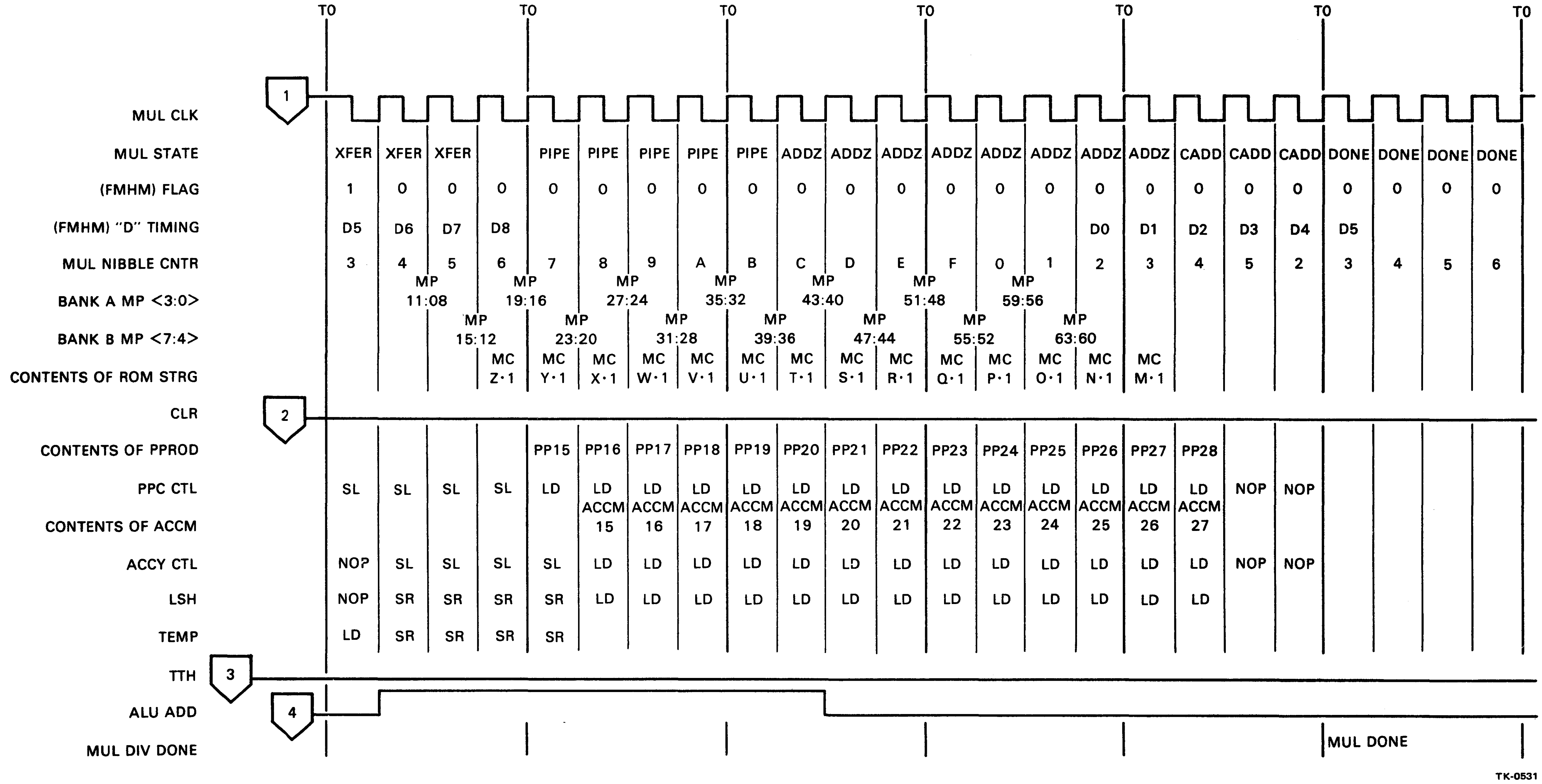
MULD TIMING



T0 T0 T0 T0 T0 T0 T0
IRD
MCONT
50 NS
MUL CLK
MUL STATE INIT SYNC CONT TEST WAIT WAIT WAIT PIPE PIPE PIPE PIPE PIPE PIPE PIPE PIPE PIPE PIPE PIPE PIPE PIPE P!PE CADD CADD CADD
(FMHM) FLAG X 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
(FMHM) "D" TIMING DO D1 D2 D3 D4
MUL NIBBLE CNTR X A B 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 2
BANK A MP <3:0> MP 11:08 MP 19:16 MP 27:24 MP 35:32 MP 43:40 MP 51:48 MP 59:56
BANK B MP <7:4> MP 15:12 MP 23:20 MP 31:28 MP 39:36 MP 47:44 MP 55:52 MP 63:60
CONTENTS OF ROM STRG MC Z·0 MC Y·0 MC X·0 MC W·0 MC V·0 MC U·0 MC T·0 MC S·0 MC R·0 MC Q·0 MC P·0 MC O·0 MC N·0 MC M·0
CLR
CONTENTS OF PPROD PP1 PP2 PP3 PP4 PP5 PP6 PP7 PP8 PP9 PP10 PP11 PP12 PP13 PP14
PPC CTL X SL SL SL SL LD LD LD LD LD LD LD LD LD LD LD LD LD LD LD LD LD NOP NOP
CONTENTS OF ACCM ACCM 1 ACCM 2 ACCM 3 ACCM 4 ACCM 5 ACCM 6 ACCM 7 ACCM 8 ACCM 9 ACCM 10 ACCM 11 ACCM 12 ACCM 13
ACCY CTL X X SL SL SL SL LD LD LD LD LD LD LD LD LD LD LD LD LD LD LD LD NOP NOP
LSH LD LD LD LD LD LD LD LD LD LD LD LD LD NOP LD
TEMP LD LD
TTH
ALU ADD
MUL DIV DONE
1
2
3
4
TK-0530

Figure 2-29 MULD Control (Sheet 1 of 3)

MULD TIMING



MUL STATE	XFER	XFER	XFER		PIPE	PIPE	PIPE	PIPE	PIPE	ADDZ	ADDZ	ADDZ	ADDZ	ADDZ	ADDZ	ADDZ	ADDZ	CADD	CADD	CADD	DONE	DONE	DONE	DONE
(FMHM) FLAG	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(FMHM) "D" TIMING	D5	D6	D7	D8												D0	D1	D2	D3	D4	D5			
MUL NIBBLE CNTR	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	2	3	4	5	6
BANK A MP <3:0>		MP 11:08		MP 19:16		MP 27:24		MP 35:32		MP 43:40		MP 51:48		MP 59:56										
BANK B MP <7:4>			MP 15:12		MP 23:20		MP 31:28		MP 39:36		MP 47:44		MP 55:52		MP 63:60									
CONTENTS OF ROM STRG				MC Z·1	MC Y·1	MC X·1	MC W·1	MC V·1	MC U·1	MC T·1	MC S·1	MC R·1	MC Q·1	MC P·1	MC O·1	MC N·1	MC M·1							
CONTENTS OF PPROD					PP15	PP16	PP17	PP18	PP19	PP20	PP21	PP22	PP23	PP24	PP25	PP26	PP27	PP28						
PPC CTL	SL	SL	SL	SL	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	NOP	NOP				
CONTENTS OF ACCM						ACCM 15	ACCM 16	ACCM 17	ACCM 18	ACCM 19	ACCM 20	ACCM 21	ACCM 22	ACCM 23	ACCM 24	ACCM 25	ACCM 26	ACCM 27						
ACCY CTL	NOP	SL	SL	SL	SL	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	NOP	NOP				
LSH	NOP	SR	SR	SR	SR	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD	LD						
TEMP	LD	SR	SR	SR	SR																			

Figure 2-29 MULD Control (Sheet 2 of 3)

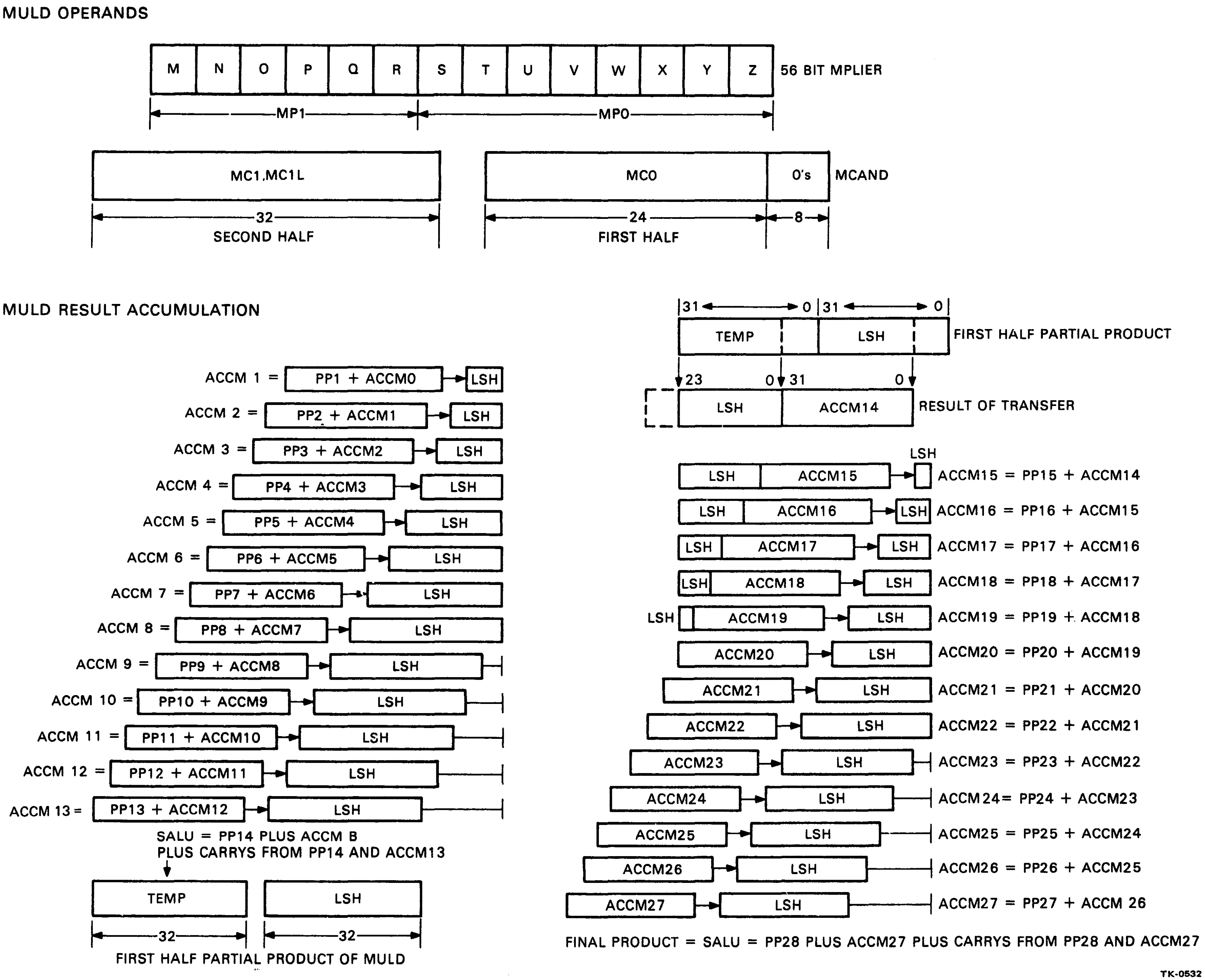


Figure 2-29 MULD Control (Sheet 3 of 3)

In WAIT, the new ROM data selected by the new ROM address accessed as a result of the new data loaded into the nibble counter during CONT is given time to settle before entering the pipeline. When FLAG is set, the data has settled and the integer multiply pipeline state (MULL) is entered.

The FM logic remains in the MULL state as the pipeline accumulates the final product (the least significant half accumulates in LSH). When COUNT = 3 is set, the AALU plus the two sets of stored-carrys is the final product. COUNT = 3 asserted selects DONE.

In DONE, MUL/DIV DONE is asserted and the final product is available. The FPA microcode loads the upper half from the SALU onto FP bus A during one machine cycle. On the following cycle the lower half is loaded from LSH onto FP bus A. Refer to Figure 2-30 for a summary of MULL control.

2.3.5.3 Division - The TEMP and LSH register in the fraction multiplier logic are used to store the quotient generated during floating-point division. The registers are concatenated with the MSB of LSH shifting into the LSB of TEMP.

During a divide operation the FPA asserts DIV and loads the divisor and dividend into the FNM. In the FM logic, the nibble counter is loaded for a MULF and clocks through until TEST. To initiate quotient storage the multiply control field (MCNT) of the FPA microcode must be asserted. The combination of MCNT and DIV asserted selects the NOP state in the division path.

The FM logic enters NOP with the nibble counter odd and exits when the nibble counter is even. The 2 cycles (100 ns) allows the first quotient bit to be formed.

From NOP, the FM logic enters DIV. In DIV, the logic left-shifts LSH and TEMP one bit every even cycle. When doing a single precision division the single quotient bit is input to both LSH bit 4 and TEMP bit 4. The data input to LSH is never accessed in single precision. In double precision the TEMP bit 4 quotient input is blocked and the TEMP bit 3 is input to TEMP bit 4 on the left shifts.

DIV DONE is asserted when quotient bits are left-shifted in TEMP bits 28 and 29. This condition is tested at T100 of each state and transfers control to DONE if true.

In DONE, MUL/DIV DONE is asserted, stopping the division process in the FNM and causing the FPA microcode to access TEMP for a single precision quotient and TEMP and LSH for a double precision quotient.

2.3.6 Exponent Processor

The exponent processor, part of the FCT, processes the FP exponent during FP operations. During FP multiply/divide, the processor adds/subtracts the exponents as needed. During add/subtracts, the processor stores the larger exponent and determines the final exponent by taking into account the operation, fraction right-shifts, and left-shifts during normalization. By comparing the exponent magnitudes the exponent processor also controls the FPF addition and subtraction in the FAD. Refer to Figure 2-31.

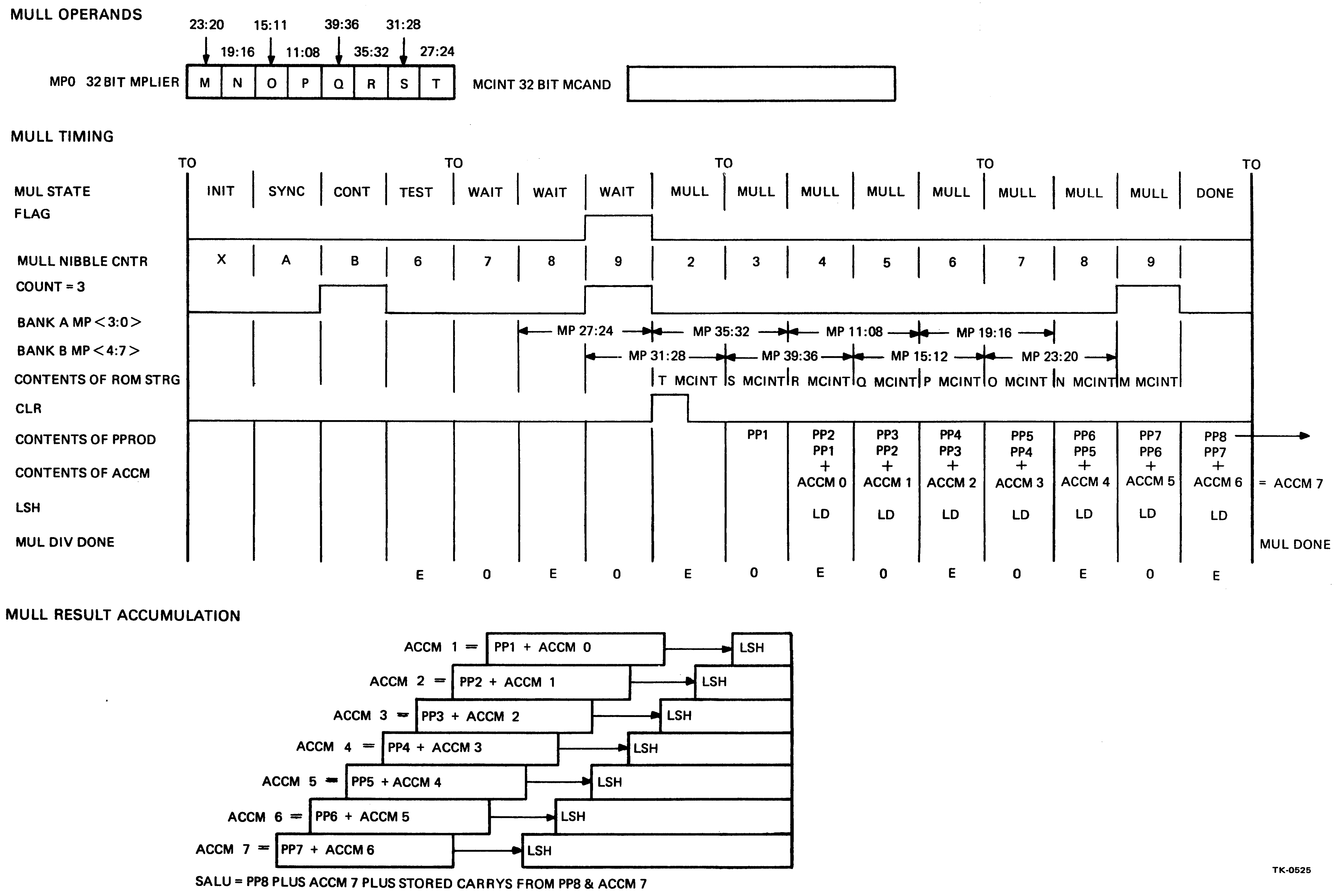


Figure 2-30 MULL Control

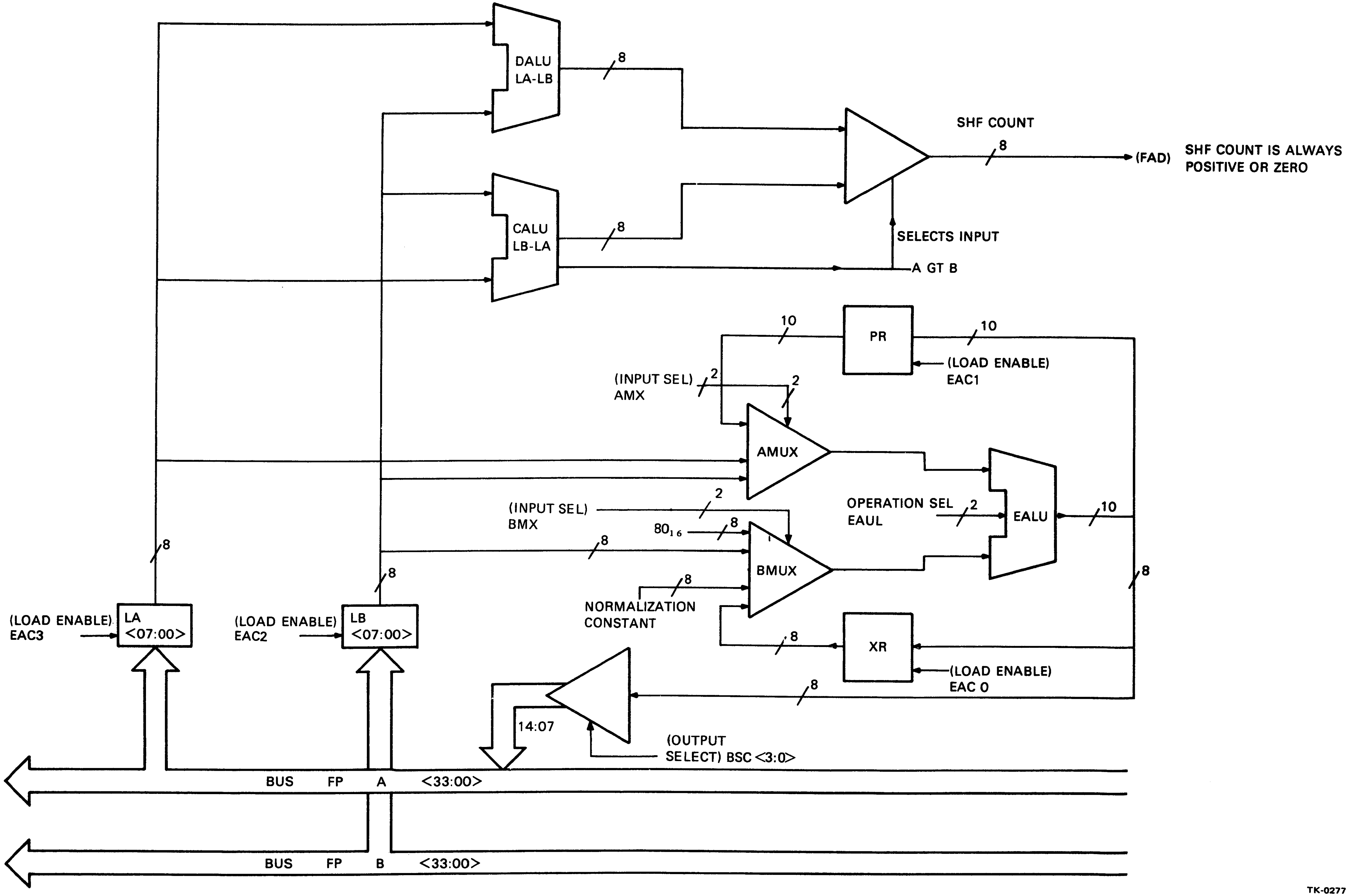


Figure 2-31 Exponent Processor Block Diagram

The FPEs are loaded from FP buses A plus B into LA and LB under control of the EAC field in the microcontrol (Table 2-19). The contents of LA and LB are loaded into CALU and DALU. CALU computes LA - LB and DALU computes LB - LA. The carry-out signal from DALU selects either CALU or DALU as the positive exponent difference (SHF COUNT) to provide FPF control in the FAD.

Table 2-19 EAC Control Store Field

	EAC Fields				
	3	2	1	0	
	µCs 27	µCs 26	µCs 25	µCs 24	
Operation **Hex**	**Controls LA → Bus A Transfers**	**Controls LB → Bus B Transfers**	**Controls PR → EALU Transfers**	**Controls XR → EALU Transfers**	
0	0	0	0	0	NOP
1	0	0	0	1	
2	0	0	1	0	
3	0	0	1	1	
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	
A	1	0	1	0	
B	1	0	1	1	
C	1	1	0	0	
D	1	1	0	1	
E	1	1	1	0	
F	1	1	1	1	

NOTE

Although the control field appears to be a 4-bit field, each bit of the 4 bits actually controls a single, independent function.

The contents of LA and LB, as well as XR (poly register), PR (product register), a normalization constant, and 80_{16} are possible inputs to EALU. Input selection is controlled by both microcontrol and hardware. Refer to Table 2-20 for input selection summary.

Table 2-20 EALU Input Control

AMXC Fields		
1	0	
μCs 35	μCs 34	**Operation**
0	0	LA to EALU A input
0	1	LB to EALU A input
1	0	PR to EALU A input
1	1	Hardware select: For FP Add/Subtract, larger exponent (LA or LB) to EALU A
BMXC Fields		
1	0	
μCs 33	μCs 32	**Operation**
0	0	Normalization constant to EALU B input
0	1	XR to EALU B input
1	0	80_{16} to EALU B input
1	1	LB to EALU B input

The EALU operation is controlled by the microcontrol field EALUC. Refer to Table 2-21. The output of the EALU can be loaded into XR or PR for further processing, or loaded onto the FPA bus as a final answer. The XR and/or PR are loaded under control of the EAC microcontrol field. Refer to Table 2-19 (bits 0 and 1). The EALU output to FP bus A <14:07> is controlled by BSC microcontrol field (Bus A EXP). Refer to the discussion of BSC field in Paragraph 2.3.2. The partial answers in XR and PR are reloaded into the EALU via AMUX and BMUX, and are combined with either a normalization constant or $\pm 80_{16}$ before they are loaded onto FPA <14:7>. Refer to Table 2-20. The normalization constant, a variable quantity, adjusts the exponent for shifts required to normalize the FPF in the FAD. (The actual normalization constant is read from a ROM rather than computed. The ROM is on the FNM.) The 80_{16} corrects for the offset that results in FPE add/subtract during exponent processing in MUL/DIV. Refer to Paragraphs 1.4 and 1.5.

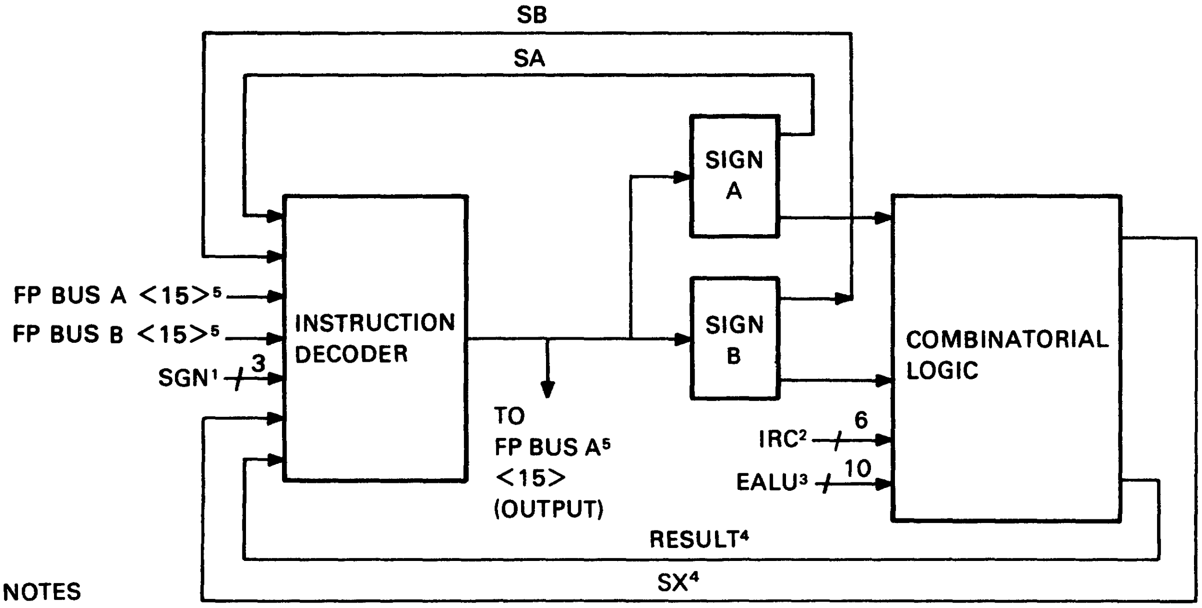
Table 2-21 EALU Control Store Field

EALU Fields								
1	0			Control Signals Generated				EALU Operation
μCs 31	μCs 30	Required Carry	Req Mode Control	S_3	S_2	S_1	S_0	
0	0	X	H (logic)	H	H	H	H	Pass A INPUT
0	1	0	L (arith)	L	H	H	L	A - B
1	0	1	L (arith)	H	L	L	H	A + B
1	1	X	H (logic)	H	H	L	L	Force 1's out (interpreted as underflow. This function is used to generate zeros on the buses.

X = Don't care

2.3.7 Sign Processor

The sign processor, a section of the FCT, determines the sign of the FP operation result using both hardware and the microcontrol field SGNC (sign latch controls). Refer to Figure 2-32 and Tables 2-22 and 2-23. This section receives information indicating the sign and magnitude of each operand, the desired operation (add, subtract, multiply, divide, poly) and the magnitude of the result. The resulting sign is placed on FP bus A 15.



NOTES

1. FROM μCS SGN FIELD
2. FROM IB DETERMINES INSTRUCTION TYPE
3. DETERMINES IF RESULT IS ZERO OR NEGATIVE
4. INTERMEDIATE RESULTS
5. SIGN OF OPERANDS

TK-0280

Figure 2-32 Sign Processor Block Diagram

Table 2-22 SGNC Control Store Field

SGNC Field			Operation	
SGN C2	SGN C1	SGN C0		
μCs 07	μCs 06	μCs 05	Load into SA	Load into SB
0	0	0	SA (NOP)	SB (NOP)
0	0	1	FP bus A 15	SB (NOP)
0	1	0	SA + Op Code = SUB	SB (NOP)
0	1	1	Result*	SB (NOP)
1	0	0	SA (NOP)	FP bus B 15
1	0	1	FP bus A 15	FP bus B 15
1	1	0	SB	SB (NOP)
1	1	1	SA + SX	SB (NOP)

* This is the resultant sign, determined by the op code, signs of the operands, the relative magnitude of the exponents, and the signs of the FALU. It can also be forced if a floating underflow or overflow occur.

Table 2-23 Sign Processor Operation

Op Code	Relative Size of Exponents	Sign of Result (FALU sign)	Result*
MULX	X	X	SA ⊕ SB
DIVX	X	X	SA ⊕ SB
ADDX	LA>LB	X	$\underline{SA}$
SUBX	LA>LB	X	$\overline{SA}$
ADDX	LA<LB	X	SB
SUBX	LA<LB	X	SB
ADDX	LA = LB	Positive	$\underline{SB}$
ADDX	LA = LB	Negative	$\overline{SB}$
SUBX	LA = LB	Positive	$\underline{SB}$
SUBX	LA = LB	Negative	$\overline{SB}$

X = Don't Care

*Except for error – in case of overflow, the sign is forced to a 1 while underflow forces a 0.

2.3.8 Control Store and Logic

As indicated in previous sections, the control store and logic, located on the FCT, provides the control signals for all FPA operations. These include both FPA internal operations: the transfer and manipulation of FP data, and external operations (interface between the FPA and CPU). Refer to Figure 2-33.

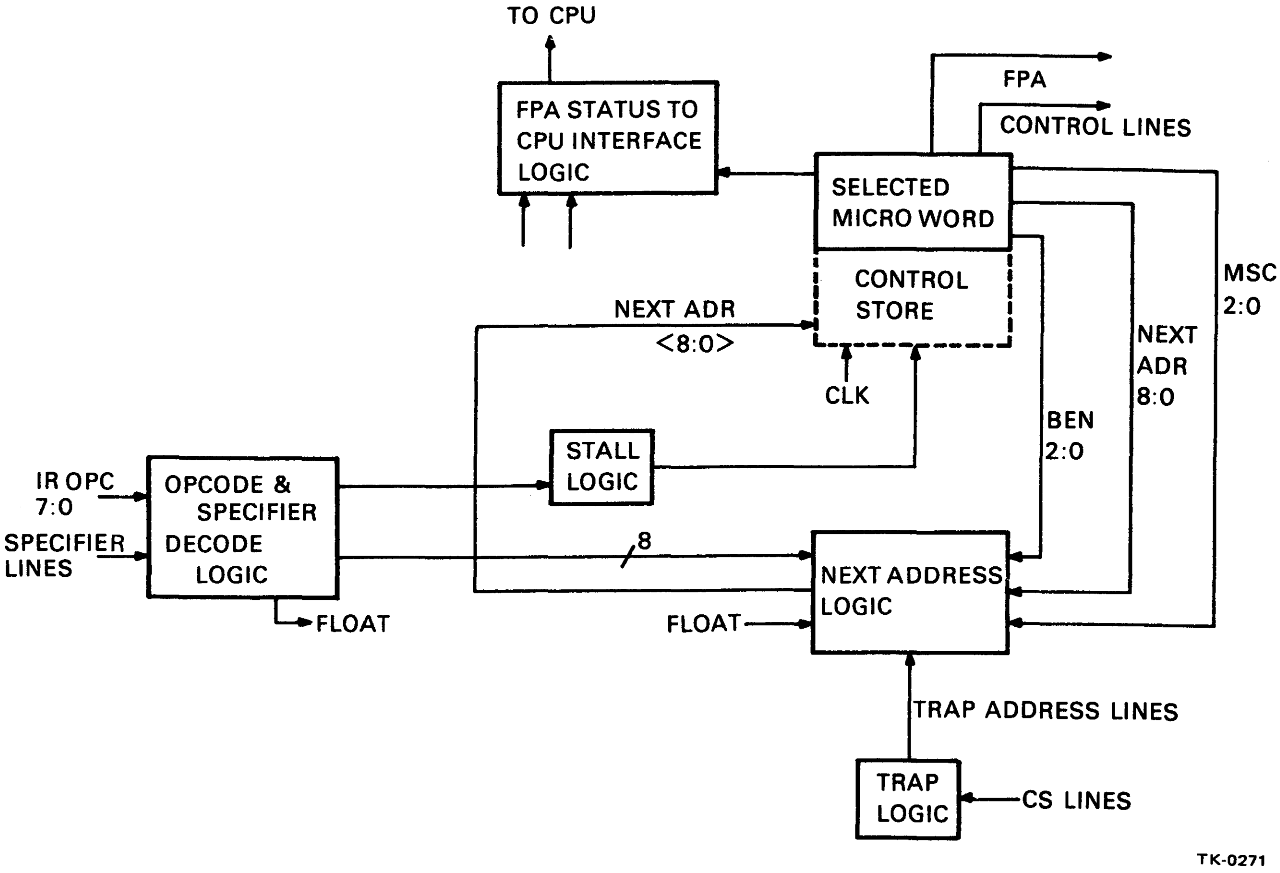


Figure 2-33 Control Store and Logic Block Diagram

The FPA has two normal operating functions: instruction register decode (IRD), and performing an FPA instruction. The FPA normally alternates between these two functions. A third function, exceptional conditions, handles error conditions, traps, and interrupts. The FPA executes the third function whenever an exceptional condition is sensed.

The FPA and the CPU run synchronously, i.e., both have 200 ns microcycles divided into 4 time states (CPT0, CPUT50, CPT100, CPT150) and T0 CPU is simultaneous with T0 FPA. Both load a new microword only at T0.

The FPA always keeps two updated copies of the 16 CPU general (scratchpad) registers. These copies are used by the FPA to optimize register-mode instructions. These register copies are accessed and updated by the same lines that access and update the CPU registers themselves. To ensure that the FPA never reads a changing register the CPU updates the general register set (and FPA copies) between T100 and T200 (T0) and the FPA reads the copies only between T0 and T100.

The FPA as a whole is directly controlled by the CPU. The CPU can enable and disable the FPA via bit 15 of the FPA status register (ID bus register 17). The FPA is normally enabled by the CPU.

The FPA is a microcontrolled unit containing a 512 words by 48 bits of control store in ROM. Each word is divided into various length control fields, each field providing independent control of a particular section of the FPA. In general, these fields: control the operation of the FPA data manipulation components; coordinate the operation of the FPA with the operation of the CPU; and initiate the operation of parts of the FPA control logic. Control of FPA operations is handled by accessing specific ROM words causing a particular set of FPA actions.

2.3.8.1 IRD – The IRD state is controlled by location IRD.1 in the control ROM. In this state a new microword is not read until STALL is disabled. ACC INSTR H and IB CALL from the CPU microword disables the STALL condition. When the FPA leaves IRD, the ACC ERROR bit in the status register is cleared if it was set during a previous cycle. The op code and specifier decode logic is monitoring the IRC OPC 7:0 and specifier lines. The OPC lines enable ACC INSTR H when a FPA instruction is in the IB and are decoded to determine instruction type. The specifier decode lines determine specifier type. The output of this decode logic is transmitted to the next address logic.

Location IRD.1 controls all FPA operations in the IRD state. The operation assumed is a register to register operation. The FPA continually begins this operation without any indication that the next operation will be an R to R because it has both operands in its register set and, if the next FPA operation is an R to R, both operands will already be loaded. Location IRD.1 has MSC = 6 and the next address = 180. This information is transmitted to the next address logic and along with the outputs of the op code and specifier decode logic determines the correct next microaddress.

In the next address logic (refer to Figure 2-34 and Table 2-24), the MSC = 6, and op code and specifier decode logic lines select the address offset to be ORed with next address (= 180) to select the next microaddress. MSC = 6 selects the A-fork inputs from op code and specifier decode logic lines and transmits them through the A–B fork mux. This selects the correct offset based on instruction type, float or double, and specifiers 1 and 2.

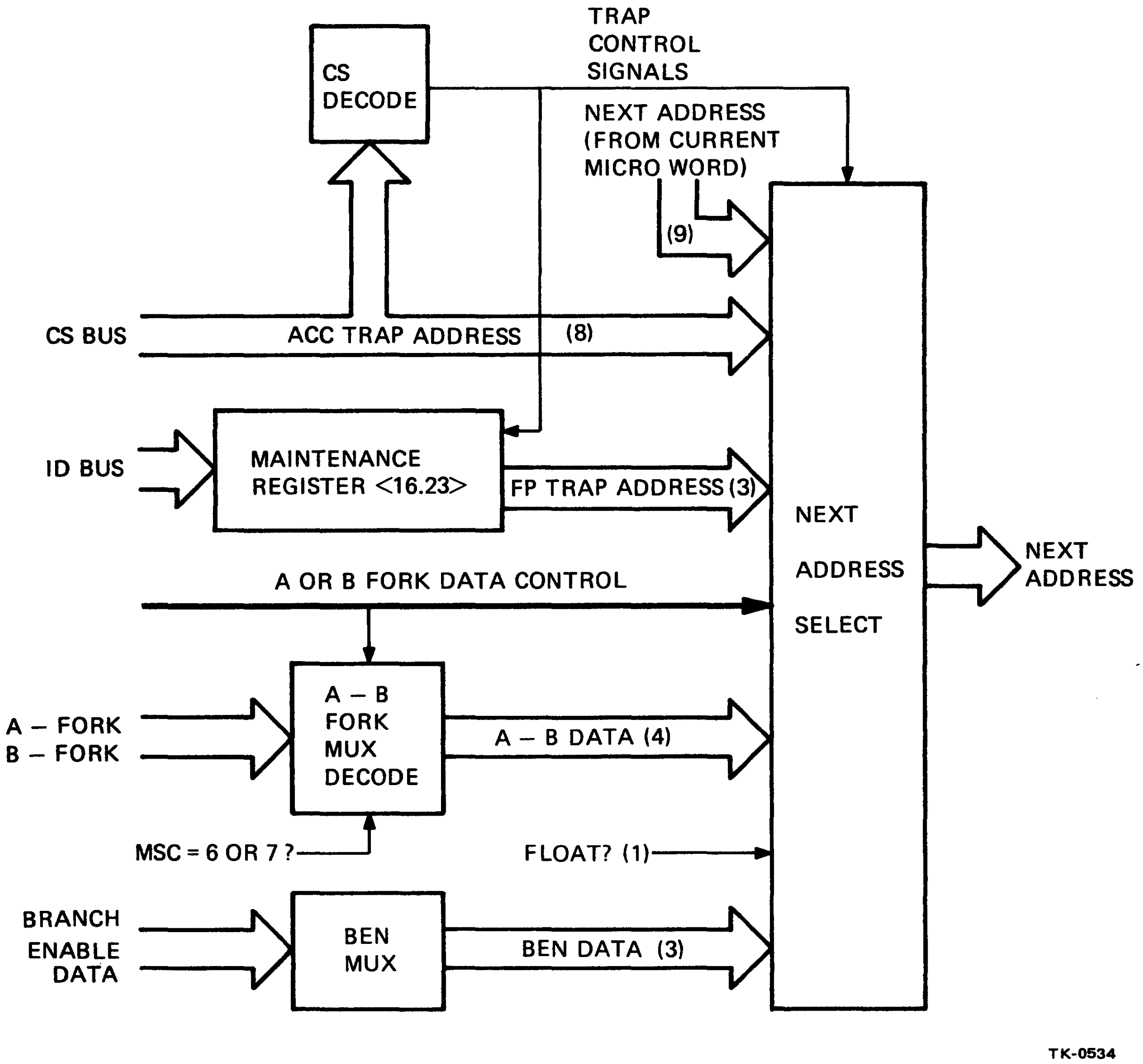


Figure 2-34 Next Address Logic

Table 2-24 Next Address Lines

Address	Description
Next Address Control Lines	
FCTK BEN 2:0 H	From FPA control store selects lines to be monitored during execution flows.
CS 71, 70	CPU accelerator control field 00 - NOP 01 - CPSYNC 10 - ACC TRAP - To 3-bit address specified by CPU USI field 11 - REDEFINE USI

Table 2-24 Next Address Lines (Cont)

Address	Description
Next Address Control Lines (Cont)	
CS 57, 56, 55	If CS71 and CS70 are high enabling DEC USI, a 6 on these lines enables POLY DONE, a 7 FP TRAP.
FCTH ACC TRAP H	High during accelerator trap, low otherwise.
FCTH FP TRAP L	Low during FP trap, high otherwise.
FCTH TRAP DIS L	Low during either FP trap or accelerator trap, high otherwise.
Next Address Selector Controls	
DEC μSI	FCTH DEC μSI L enabled and CS 57, 56, and 55 high enable FCTH FP TRAP, otherwise it is high.
A-FORK B-FORK SELECT MUX	Enable H causes all highs out and doesn't affect next address. Enable L enables select input to select A–B data.
NEXT ADDRESS MUX	Enable H causes all highs out. If enable is low, S low selects A input.
BEN MUX	Enable high causes all highs out.
Address Lines	
FCTR CRADR 08:00 H	To control store selects address. Also can be transmitted to CPU via Reg 16 as current ADR.
FCTK NEXT ADR 08:00	From control store next address from microword.
FCTH TRAP A 07:00 L to FCTF	Contains either trap address or next address.
FMHR TRAP A 7:00H	FP trap address from MAINT REG ID BUS.
FCTH BRC 2:0 L	From branch enable MUX (BEN) monitors various FPA conditions and modifies the next address during execution flows based on BEN field in FPA microcode.
A–B FORK ADR	(Not a signal name on prints) From A-FORK B-FORK select Mux. Monitors op code and specifier type from IB and modifies address in A–B forks.
FCTF FLOAT H	Based on op code. Used during A–B forks and by branch enable logic (BEN).
CS 57, 56, 55	Select trap address during ACC trap. Also refer to CS 57, 56, 55 in control lines.

The offset is ORed with 180 and since STALL is no longer enabled (ACC INSR H is high) the next CPT 0 will select the correct microword to control the next FPA cycle. If the data is already in the FPA, an optimized routine will be selected.

2.3.8.2 Performing an FPA Instruction - Once an FPA instruction is sensed, the microcontrol words and the order they are selected is based on the operation desired, float or double, location of the operands, and relative size of the operands and/or result.

The FPA first ensures that it has all the required data. If both operands are in registers, or one is in a register and the other is a short literal, all the data is in the FPA after the A-fork test and the FPA transfers directly to the execution flows. If not, the first operand is fetched during A-fork and then MSC = 7 and next address = 100 is transmitted to the next address logic.

In the next address logic, MSC = 7 selects the B-fork inputs from the op code and specifier decode, and transmits them through the A–B fork mux to be ORed with next address = 100. The offset selected depends on instruction type, double or float, and type of specifier 2. As before, if the data is already in the FPA, an optimized routine is selected; otherwise, the FPA waits for the CPU to fetch data.

In some data transfers (A-fork or B-fork) the FPA must wait for data to be transmitted from the CPU via the ID bus. The microcode has a special WAIT bit to enable STALL for this purpose. The CPU indicates that the required data is on the ID bus by asserting CP SYNC. CP SYNC causes the data to be stored in the FPA and clears STALL; thereby enabling a new microword to be read and FPA operations to continue.

Once the FPA has all required data ACC OVERIDE is asserted. This signal, transmitted to CPU microaddress bit 12, causes the CPU to select microcode from FPA specialized microcode in the writeable control store (WCS) rather than PCS. This prevents the CPU from beginning microcode floating-point routines (used when no FPA is present) to do FP instructions. The enabling of ACC OVERIDE is based on instruction type (IRC lines) and the execution point counter, (IRC EP 2:0). Note that since the FPA cannot fetch data itself, the data-fetch routines (CPU AFORK and BFORK) are allowed to continue until the FPA has all required data.

Once the FPA has all the data the FPA execution flows are entered. These flows perform the manipulation required to A, S, M, and D. This includes unpacking and individually manipulating the FPF and FPE parts of the number, as well as checking the operands and/or results for unusual conditions (zeros, underflow, overflow, etc.). During execution flows the BEN field selects lines to be monitored and used to modify the next address. The 3-bit BEN field of each microword can select 3 of 24 possible lines to be ORed with the next address field of the microword to select the address.

The BEN multiplexer monitors signals from both the CPU and FPA. POLY DONE and CP SYNC are transmitted from the CPU using CS lines 71, 70, 57, 56, and 55. FLOAT, IRBR0 L, and IRBR1 L are generated in the FPA but are summaries of op code information transmitted from the instruction buffer. All other BEN lines monitor FPA internal conditions. Refer to Table 2-25 for a summary of BEN fields. Finally the flows manipulate the result to ensure it is in correct form and inform the CPU via FP SYNC asserted that the answer is available.

Table 2-25 BEN Control Store Field

BEN	Lines Monitored			Operation
Field	BRC2L	BRC1L	BRC0L	Summary
0				NOP
1	FLOAT H*	IRBR1 L*	IRBR0 L*	Op code decode
2	SWR	SWR	SWR	Shift within range
3	RSVH	B H	A=0 H	Operand(s) equal zero Reserved operand
4	POLY DN L*	CP SYNC H*	FLOAT*	
5	(A or B=0) H	SUB*ED<2 H	ED.GE.8 H	Operand(s) equal zero Check exponent difference
6			MUL/DIV DN H	Multiply done Division done
7		UNDF L	PR 8 H	Error Condition

*From the CPU.

The CPU accepts the answer via DFMX bus drivers on the FNM using DAP ENA ACC D (1) and also reads the ACC Z, V, C, and N data lines to determine the condition codes of the answer. Once the CPU has the answer it transmits a CPSYNC and the FPA returns to its IRD state.

2.3.8.3 Exception Conditions – At any time during either IRD or instruction states the CPU can direct the FPA to enter a trap routine for error recovery or microdiagnostics. The trap routines are located in the FPA's own microcode. There are two separate sets of trap routines: ACC traps for CPU and FPA errors and FP traps for microdiagnostics. Both trap routines are initiated via CS lines 71 and 70.

If CS bus 71 is H and CS bus 70 is L, an ACC TRAP is initiated. An ACC TRAP addresses the FPA microcode location selected by CS bus lines 57, 56, and 55 (location 0–7). These traps are normally initiated for power-up and abort sequences.

If CS bus 71, 70, 57, and 56 are high and 55 is low, an FP trap is initiated. The FP trap selects an 8-bit address previously stored in ID register 16, the Status register to access one of 256 addresses in the FPA microcode (location 0–255). These trap locations normally handle FPA microdiagnostics. Refer to Figure 2-34.

2.4 FPA Microcontrol Fields

This section summarizes all the fields in the FPA microcontrol word. Figure 2-35 shows the complete microcontrol word, all the fields, and the microcode mnemonics. Table 2-26 lists the function of each field.

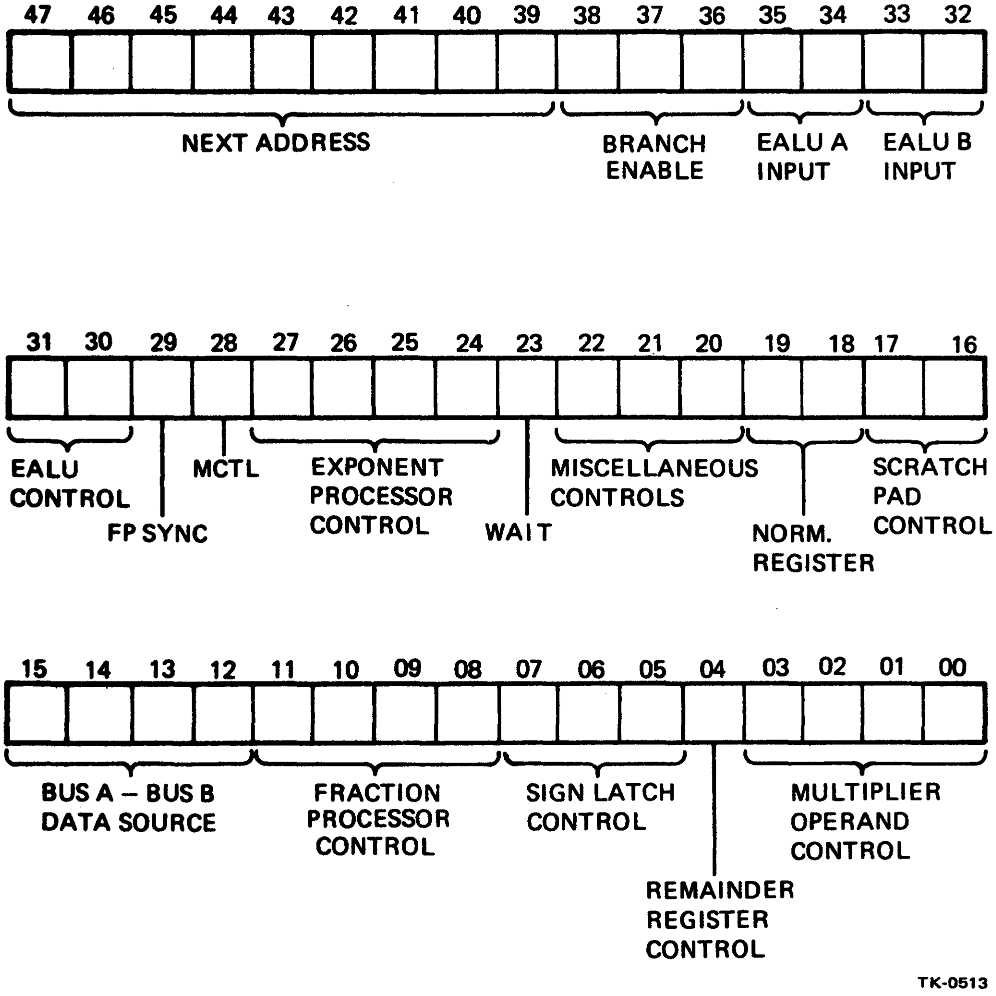


Figure 2-35 FPA Control Word Fields

Table 2-26 FPA Control Word Field Definitions

Microcode Bits	Field	Function
47:39 (9 bits)	NAD – Next Address	Contains the address of the next control word to be accessed.
38:36 (3 bits)	BEN – Branch Enable	Selects signals to be used for next address calculations.
35:34 (2 bits)	AMXC – A Mux Control	Selects A input to FCT exponent ALU.
33:32 (2 bits)	BMXC – B Mux Control	Selects B input to FCT exponent ALU.
31:30 (2 bits)	EALUC – EALU Control	Controls FCT exponent ALU operation.
29 (1 bit)	FPSYNC – Floating-Point Synchronize	Transmits FPSYNC to CPU.
28 (1 bit)	MCTL – Multiply Control	Starts FML and FMH fraction multiply operation.
27:24 (4 bits)	EAC – Exponent Processor Control	Controls FCT (exponent processing).
23 (1 bit)	WAIT – Wait	Controls FPA wait loop operation. Stalls until CPSYNC.
22:20 (3 bits)	MSC – Miscellaneous Control	Controls Miscellaneous FPA operations.
19:18 (2 bits)	NRC – Normalization Register Control	Controls fraction normalize operation in FNM.
17:16 (2 bits)	SCR – Scratchpad Control	Handles FPA General Register copies on FNM.
15:12 (4 bits)	BSC – Bus A – Bus B Data Source	Controls data transmission along FPA buses.
11:8 (4 bits)	FADC – Fraction Processor Controls	Controls FAD fraction processing.
7:5 (3 bits)	SGNC – Sign Latch Controls	Controls sign calculation on FCT.
4 (1 bit)	LRR – Load Remainder Register	Controls remainder register (RR) on FNM.
3:0 (4 bits)	OPLD – Operand Load (Multiplier Control)	Loads fractions for multiplication on FML and FMH.

2.5 FPA MICROCODE STRUCTURE

The FPA contains a 512 word by 48 bits (per word) memory. This memory provides microcontrol of the FPA during normal operation and diagnostic programs for maintenance and troubleshooting. About 225 locations are for normal microcontrol, and 200 locations contain diagnostic programs. The other locations are available for future use.

The microcontrol code has an IRD state (instruction register decode) and three fork points (A, B, and C). The FPA remains in the IRD state until an FPA instruction is decoded. The FPA then enters A-fork, to receive the operands. If both operands are registers or short literals, optimized routines are entered and computation begins. Otherwise, B-fork is entered. If the second operand is not register data, C-fork is entered. Otherwise a B-fork optimization is taken. Figure 2-36 shows the basic microcode structure and indicates the microcode starting addresses of the various routines.

2.6 FPA INTERFACE FIRMWARE

The CPU-FPA interaction is handled by specialized firmware located in the CPU's writeable control store (WCS).

This firmware handles numerous interface tasks. For ADD, SUBT, MUL, and DIV operations it accepts and stores the FPA results and condition codes, and handles any exceptions flagged by the FPA. In 3-operand op codes it calls specifier decoding microcode in the base machine to decode the third operand. It also handles the special requirements of the EMOD, MULL and POLY commands. It is accessed when the FPA overrides the CPU Address by forcing the μPC <12> to 1. This happens when the FPA detects an execution or optimization exit at a CPU A-fork, B-fork, or C-fork for an FPA implemented instruction.

2.6.1 Major Interface Functions

This firmware coordinates the interface between the CP microcode and the FP microcode including the normal transfers of CPU data to the FPA, FPA results back to the proper register in the CPU, and various control signals for both normal and exception control.

Table 2-27 lists important macros and microorders that are used by the FPA interface firmware to generate and/or monitor the signals which are transferred between the CPU and FPA.

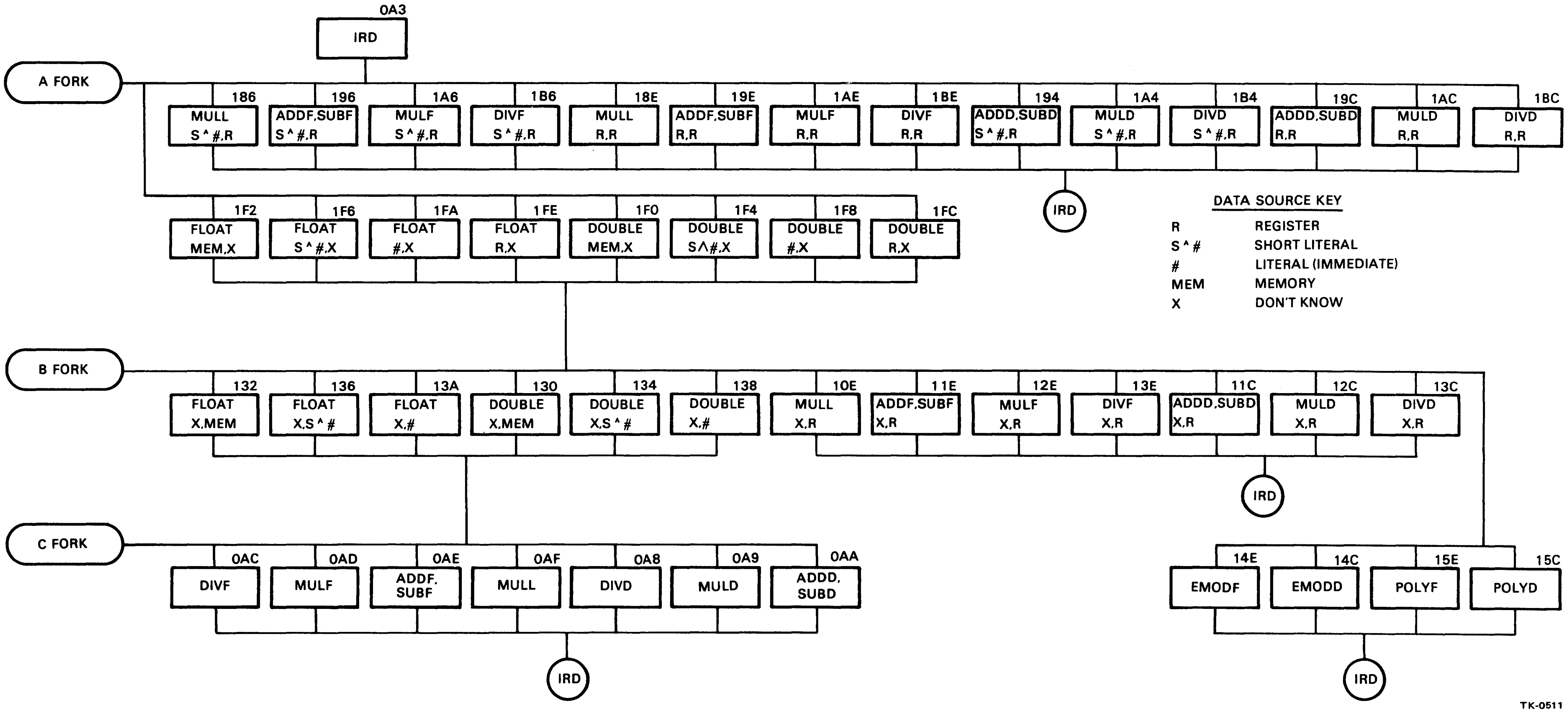


Figure 2-36 FPA Microcode Structure

Table 2-27 Interface Microcode

Name of Macro	Signal Monitored or Generated	Data Transfer	Function
ID-D. SYNC	CP SYNC generated	CPU → FPA	Gates the CPU D-Register's contents onto the ID bus. Generates CP SYNC. CP SYNC indicates that valid data is on bus.
D-ACCEL & SYNC	CP SYNC generated	FPA → CPU	Gates data placed on DFMX Bus by FPA into D-Register. CP SYNC indicates that the FPA's data has been accepted.
Q-ACCEL & SYNC	CP SYNC generated	FPA → CPU	Gates data placed on DFMX Bus by FPA into Q-Register. CP SYNC indicates that the FPA's data has been accepted.
ACCEL?* (BEN/ACC<UB2, UB1, UB0>)†	FP SYNC monitored	FPA → CPU	ACC<UB0> = 1; Result data, on DFMX bus, and condition codes are being transmitted by FPA. If double precision condition codes are passed with first half.
	ERR SYNC monitored	NO	ACC<UB1> = 1; An exception has been detected by the FPA. This initiates specialized routines that handle the exception.
	Not Mull** generated	NO	ACC<UB2> = 1; Separates MULL and MULF
POLY.DONE	POLY.DONE generated	CPU → FPA	Indicates the last coefficient in the POLY operation, it being presented. In POLYD, used while both halves of the last coefficient are transmitted.
TRAP.ACC[1]	Accelerator Trap	NO	Returns FPA microcode to IRD state
MSC/LOAD. ACC.CCT		NO	Loads PSW<N,Z,V,C> with FPA generated condition codes from CPU latches loaded in previous cycle.

* This macro, in combination with the target constraint block, enables the CP microcode to test for various conditions.

† This is a microorder rather than a macro.

** This is a condition rather than a specific signal.

2.6.2 Major Instruction Groups

The FPA firmware can be broken into 4 groups of routines: Generalized instructions handler, POLY handler, MULL handler, and EMOD handler.

Group 1 handles all ADD, SUB, MUL, and DIV instructions as well as FPA exceptions. This group provides optimized flows for operands located in the general register set and literal operands.

The POLY group transmits the polynomial coefficients to the FPA as they are needed and transmits POLY DONE when the last coefficient has been transmitted. It also responds to the FPA detection of overflow, underflow, and coefficient reserved operand. Overflow and reserved operand detections causes a branch to exception conditions routines in the base machine. If an underflow is noted, the firmware notes it and continues execution of the POLY flows.

The MULL routine accepts the result of the longword integer multiplication from the FPA. Since the FPA creates an unsigned 64-bit product using 32-bit signed operands, the firmware must correct the result by subtracting out the effects of the negative signs on the magnitude result. To do this the firmware stores the operands in a form that can later be used as subtrahend operands to correct the product and, based on this stored information, determines the correction sequence to select when the result is transmitted from the FPA. The firmware also creates the proper signed result, sets the condition codes, and tests for overflow.

The FPA handles only the fraction multiply of the EMOD instructions. As a result the EMOD firmware is relatively short. While the FPA is doing the fraction multiply this routine adds the exponents and checks for reserved operands, accepts the fraction multiply result from the FPA, checks for a zero result, and formats the FPA result so control can return to the EMOD routines in the base machine.

FP780
FLOATING-POINT ACCELERATOR
TECHNICAL DESCRIPTION
EK-FP780-TD-001

Reader's Comments

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our publications.

What is your general reaction to this manual? In your judgment is it complete, accurate, well organized, well written, etc.? Is it easy to use? ______

______

______

______

What features are most useful? ______

______

______

______

What faults or errors have you found in the manual? ______

______

______

______

Does this manual satisfy the need you think it was intended to satisfy? ______

Does it satisfy *your* needs? ______ Why? ______

______

______

______

□ Please send me the current copy of the *Technical Documentation Catalog*, which contains information on the remainder of DIGITAL's technical documentation.

Name ______ Street ______
Title ______ City ______
Company ______ State/Country ______
Department ______ Zip ______

Additional copies of this document are available from:

Digital Equipment Corporation
444 Whitney Street
Northboro, Ma 01532
Attention: Communications Services (NR2/M15)
Customer Services Section

Order No. EK-FP780-TD-001

Fold Here

Do Not Tear - Fold Here and Staple

Postage will be paid by:

Digital Equipment Corporation
Technical Documentation Department
Maynard, Massachusetts 01754