EK-FP11E-UG-001

FP11-E
floating-point processor
user's guide

digital equipment corporation • maynard, massachusetts

Preliminary Edition, October 1977
1st Edition, December 1977

Printed in U.S.A.

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

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

FIGURES

FIGURES (CONT)

TABLES

PREFACE

This user's guide is intended to familiarize the reader with the operation of the FP11-E Floating-Point Processor. It contains information relating to the FP11-E's hardware and software and can thus be used as a reference by both programmers and technicians. The manual is divided into the following chapters.

Chapter 1	General characteristics of the FP11-E
Chapter 2	Basics of floating-point numbers and floating-point arithmetic
Chapter 3	Formats in which data must enter and leave the FP11-E
Chapter 4	FP11-E instruction set
Chapter 5	Relationship and interaction between the FP11-E and the PDP-11/60 CPU
Chapter 6	Data paths and components that manipulate data in the FP11-E
Chapter 7	Operation of the FP11-E control logic
Chapter 8	FP11-E installation information

It is emphasized that this user's guide is intended only as an overview of the FP11-E. For more detailed information (primarily for the use of maintenance personnel), refer to the *FP11-E Floating-Point Processor Technical Manual.*

The following documents should be used in conjunction with this manual:

Title	**Document No.**	**Media**
PDP-11/60 Processor Handbook	EB06498	Hard copy only
FP11-E Microcode Listing	DQMCB-A-D	Hard copy
	DQMCB-A-FA	Microfiche

CHAPTER 1
INTRODUCTION

1.1 GENERAL

The FP11-E Floating-Point Processor is a hardware option used with the PDP-11/60 central processor. The FP11-E, which is an extension of the CPU data paths, allows rapid execution of floating-point instructions.

The primary advantage of the FP11-E is speed. Although a floating-point instruction set is an integral part of the PDP-11/60 CPU (contained in the CPU's firmware), the FP11-E can execute the same instructions at much greater speed. Furthermore, the operation of the CPU and the FP11-E overlap; some floating-point instructions allow the CPU to proceed to other tasks while the FP11-E completes the instruction execution. This increases system efficiency by reducing CPU idle time.

1.2 FEATURES

The following paragraphs summarize the features of the PDP-11/60 floating-point instruction set and the FP11-E.

1.2.1 Floating-Point Instruction Set Features

- 32-bit (single-precision) and 64-bit (double-precision) data modes
- Addressing modes compatible with existing PDP-11 addressing modes
- Special instructions that can improve input/output routines and mathematical subroutines
- Allows execution of in-line code (i.e., floating-point instructions and other instructions can appear in any sequence desired)
- Multiple accumulators for ease of data handling
- Can convert 32- or 64-bit floating-point numbers to 16- or 32-bit integers during the Store class of instructions
- Can convert 32-bit floating-point numbers to 64-bit floating-point numbers and vice-versa during the Load or Store class of instructions.

1.2.2 FP11-E Features

- Performs high-speed, floating-point operations on single- and double-precision data
- Has 17 (decimal) digit accuracy
- Performs overlapped operation with CPU
- Contains its own microprogrammed control store
- Contains six 64-bit floating-point accumulators
- Contains error recovery aids
- Responds to maintenance instructions for ease of maintenance. (Refer to *FP11-E Floating-Point Processor Technical Manual*.)

1.3 FP11-E PHYSICAL DESCRIPTION

The FP11-E is used with the PDP-11/60 CPU (KD11-K) and consists of four multilayer hex modules (Table 1-1). These modules are circuit boards which are plugged into slots 8, 9, 10, and 11 of the prewired KD11-K backpanel. Since the modules are hex modules, they occupy rows A through F in all four slots (Figure 1-1).

The FP11-E is powered solely by its own power supply. This supply provides the FP11-E with +5 Vdc and is mounted in the rear of the PDP-11/60 chassis (Chapter 8).

Table 1-1 FP11-E Modules

Module No.	Module Name	Slot	Rows
M7881	FALU	11	A–F
M7880	MULNET	10	A–F
M7879	EXPONENT	9	A–F
M7878	FNUA	8	A–F

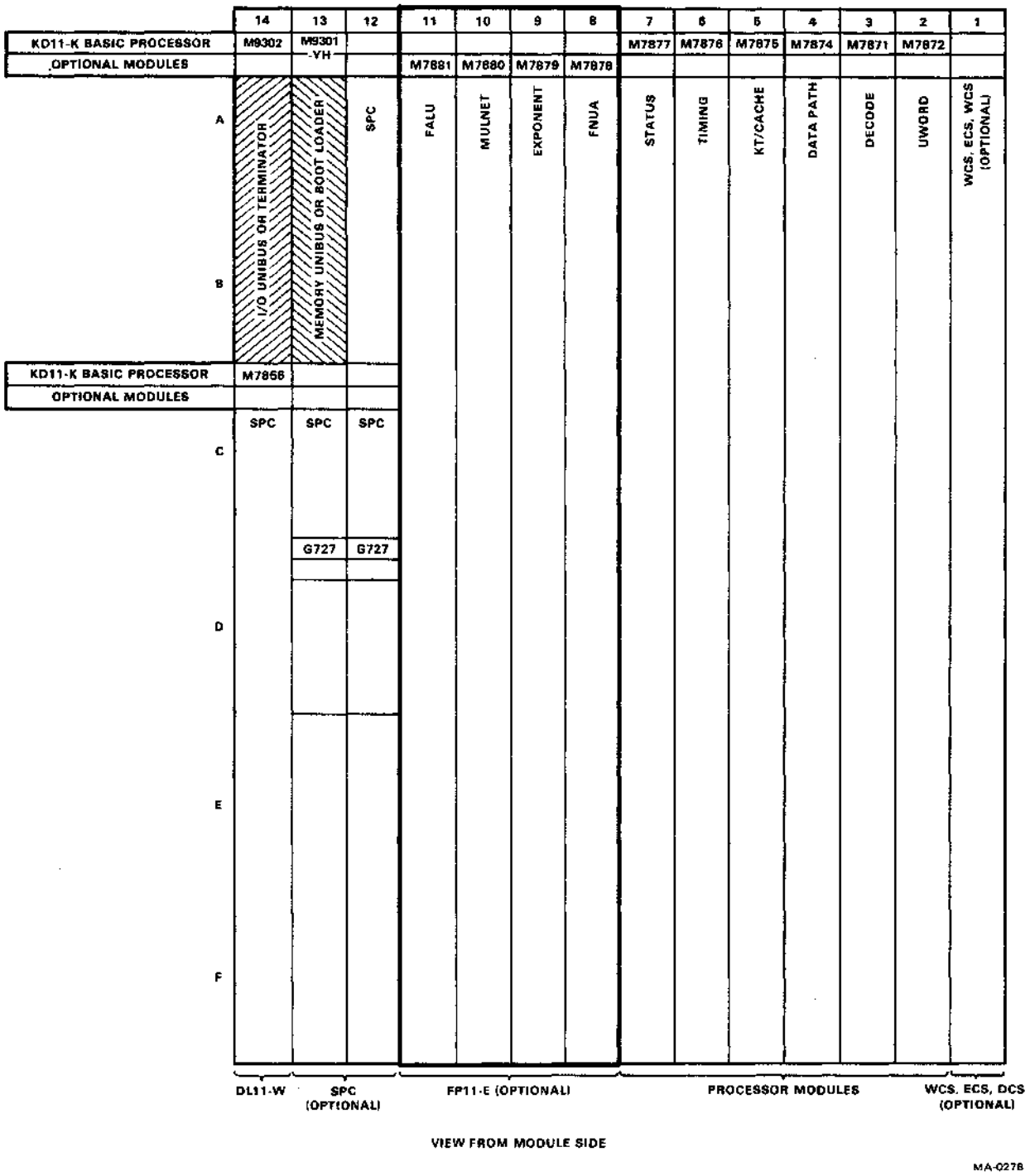


Figure 1-1 FP11-E Module Layout

CHAPTER 2
REVIEW OF FLOATING-POINT NUMBERS

2.1 INTRODUCTION

This chapter briefly outlines some fundamentals of floating-point arithmetic. It provides useful background for more advanced topics in later chapters. The reader already familiar with floating-point numbers and arithmetic may skip to Chapter 3 for a discussion of FP11-E data formats.

2.2 INTEGERS

All data within a computer system could be represented in integer form. The numbers that could be represented in a 16-bit machine range in magnitude from 000000_8 to 177777_8 (or from 0_{10} to $65{,}536_{10}$). However, there would be problems with integer representation. A number between 1 and 2 (for example) could not be represented. Thus, integer representation imposes an *accuracy* limitation. Furthermore, numbers greater than $65{,}536_{10}$ could not be represented. This imposes a *range* limitation.

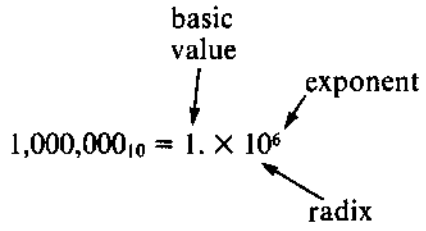
These limitations are imposed by the stationary position of the *radix point* (e.g., the decimal point in base 10 notation or the binary point in base 2 notation). An integer's radix point is usually omitted in integer representation because it always marks the integer's least significant place. That 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.

2.3 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



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

$$
\begin{aligned}
512. &= 51200. \times 10^{-2} \\
&= 5120. \times 10^{-1} \\
&= 512. \times 10^{0} \\
&= 51.2 \times 10^{1} \\
&= 5.12 \times 10^{2} \\
&= .512 \times 10^{3}
\end{aligned}
$$

The convention chosen for representing floating-point numbers with scientific notation in the FP11-E 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.

More examples of scientific notation are shown below.

Decimal No.	**Decimal Scient. No.**	**Octal Scient. No.**	**Binary Scient. No.**
64	0.64×10^{2}	0.1×8^{3}	0.1×2^{7}
33	0.33×10^{2}	0.41×8^{2}	0.100001×2^{6}
1/2	0.5×10^{0}	0.4×8^{0}	0.1×2^{0}
1/16	0.625×10^{-1}	0.4×8^{-1}	0.1×2^{3}

Note that in each of the examples above, only significant digits are retained in the final result and the radix point is always (by convention) to the left of the most significant digit. Establishing the radix point in a number whose basic value is greater than (or equal to) 1 is accomplished by shifting the number to the right until the most significant digit is to the right of the radix point. Each right shift causes the exponent to be incremented by 1. Similarly, establishing the radix point in a number whose basic value is between 1 and 0 (i.e., a fraction) is accomplished by shifting the number to the left until all leading 0s are eliminated. Each left shift causes the exponent to be decremented by 1.

To summarize, the value of the number remains constant if its exponent is incremented for each right shift of the basic value and decremented for each left shift. The representation for floating-point fractions in the PDP-11/60 is one in which all nonsignificant leading zeros have been removed. The process used to obtain this representation is called *normalization*, which is explained in more detail in Paragraph 2.4.

2.4 NORMALIZATION

In digital computers, the number of bits in a fraction is limited. Retention of nonsignificant leading zeros decreases accuracy by taking places which could be filled by significant digits. For this reason, a process called normalization is used in the FP11-E. The normalization process consists of testing the fraction for leading zeros and left-shifting it until it is in the form 0.1 . . . . The exponent is accordingly decremented by the number of left shifts of the fraction. This ensures that the normalized number retains equivalence with the original number. Since digits to the right of the binary point are weighted with inverse powers of two (i.e., 1/2, 1/4, 1/8 . . .), the smallest normalized fraction is 1/2 (0.10000 . . .). The largest normalized fraction is 0.11111 . . . . Figure 2-1 shows an unnormalized fraction which must be left-shifted six places to be normalized. The exponent is decremented by six to maintain equivalence with the original number.

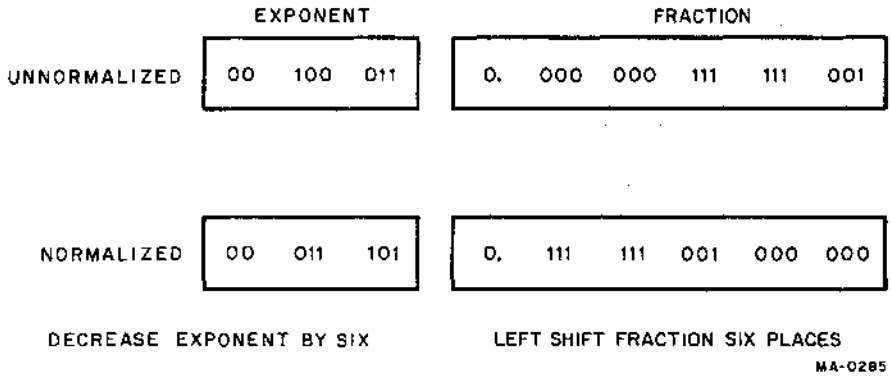


Figure 2-1 Normalization

Problem A – Represent the number 75_{10} as a binary normalized floating-point number.

1. Integer conversion
   $75_{10} = 1001011_2$

2. Convert to floating-point form
   $1001011.0 \times 2^0 = 0.1001011 \times 2^7$

   Fraction = 0.1001011
   Exponent = 111

Problem B – Represent the number 0.25_{10} as a binary normalized floating-point number.

1. Integer conversion
   $0.25_{10} = 0.01_2$

2. Convert to floating-point form
   $0.01 \times 2^0 = 0.1 \times 2^{-1}$

   Fraction = 0.1
   Exponent = –1

2.5 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 incrementation of the associated exponent. When the exponents are aligned or equal, 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 the example below, 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.

$$0.101\ 000\ 000\ 000\ 000 \times 2^6 = 50_8 = 40_{10}$$

$$+0.111\ 000\ 000\ 000\ 000 \times 2^3 = 7_8 = 7_{10}$$

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.

$$\begin{array}{r} 0.101\ 000\ 000\ 000\ 000 \times 2^6 = 50_8 = 40_{10} \\ +0.000\ 111\ 000\ 000\ 000 \times 2^6 = 7_8 = 7_{10} \\ \hline 0.101\ 111\ 000\ 000\ 000 \times 2^6 = 57_8 = 47_{10} \end{array}$$

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

$$\begin{array}{l} \underset{\sim}{5}\ \ \underset{\sim}{7} \\ 0.\underset{\smile}{101}\ \underset{\smile}{111}.000\ 000\ 000 \end{array}$$

2.6 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:

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

1. $$\begin{array}{r} 0.1110000 \times 2^3 = 7_8 = 7_{10} \\ \times 0.1010000 \times 2^6 = 50_8 = 40_{10} \\ \hline 111 \\ 0000 \\ 11100 \\ \hline .10001100000000 \times 2^9 \end{array}$$ (Result already in normalized form.)

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

$$\begin{array}{l} \underset{\sim}{4}\ \underset{\sim}{3}\ \underset{\sim}{0} \\ .\underset{\smile}{100011000}.00000 = 430_8 = 280_{10} \end{array}$$

Example:

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

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

$$1.010000\overline{).1111000} =$$

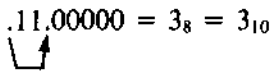
$$\begin{array}{r} 1.100000 \\ 1010000\overline{)1111000.000000} \\ 1010000 \\ \hline 101000 \\ 101000 \\ \hline 0 \end{array}$$

2. Exponent: $4 - 3 = 1$

3. Result: 1.100000×2

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

Move binary point two places to the right.



CHAPTER 3
DATA FORMATS

3.1 INTRODUCTION

The FP11-E requires its input data (operands) to be formatted. Formatting allows the FP11-E to process operands in a meaningful way and produce correct results. There are four different formats for operands input to the FP11-E: short integer format (I), long integer format (L), single-precision format (F), and double-precision format (D).

Output data from the FP11-E is also formatted. This output data is in the form of:

1. FP11-E status information and FP11-E exception information required by the CPU
2. Data sent to memory (via the CPU). Data sent to memory must be in I, L, F, or D format.

This chapter describes the FP11-E data formats. It is assumed that the reader is familiar with 2's complement notation.

3.2 FP11-E INTEGER FORMATS

There are two integer formats, short (I) and long (L). The short integer format is 16 bits long and the long integer format is 32 bits long. Data words (operands) in integer format are represented in 2's complement notation. In both I and L formats, the most significant bit of the data word is the sign bit. Figure 3-1 shows the integers 5 and -5 in both I and L formats.

Figure 3-1 illustrates the formats in which integers are arranged *in memory*. Integers sent to memory must be in one of these formats. Integers received by the FP11-E are arranged and manipulated according to the type of instruction being executed. Refer to Paragraphs 4.3.11 and 4.3.12 for descriptions of the ways in which incoming integers are manipulated during the Load Exponent and Load Convert Integer-to-Floating instructions, respectively.

3.3 FP11-E FLOATING-POINT FORMATS

There are two floating-point formats, single-precision (F) and double-precision (D). The single-precision format is 32 bits long and the double-precision format is 64 bits long. Referring to Figure 3-2, it can be seen that the most significant bit is the sign of the fraction (and the floating-point number being represented). The next 8 bits contain the value of the exponent, expressed in excess 200 notation (Paragraph 3.3.1.2). The remaining bits (23 for single-precision, 55 for double-precision) contain the fraction. The fraction and its associated sign bit are expressed in sign and magnitude notation (Paragraph 3.3.1.1).

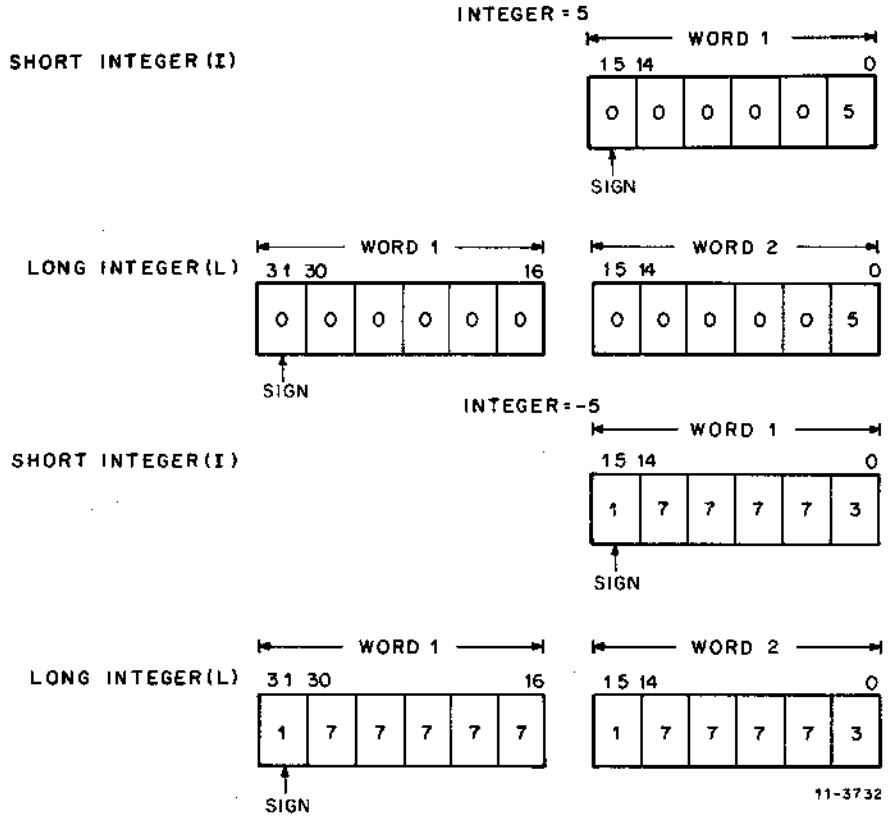
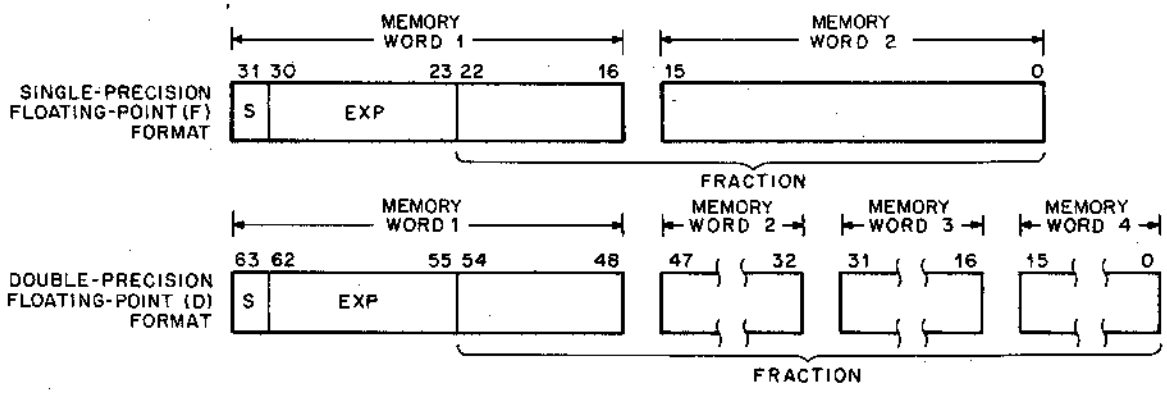


Figure 3-1 Integer Formats



S = Sign
EXP = Exponent in excess 200 notation (refer to paragraph 3.3.1.2.).
Fraction = 23 or 55 bit fraction in sign and magnitude format.

MA-0280

Figure 3-2 Floating-Point Data Formats

3.3.1 FP11-E Floating-Point Data Word

Figure 3-2 illustrates the formats in which floating-point numbers are arranged *in memory*. Floating-point numbers sent to memory must be in one of these formats. Floating-point numbers received by the FP11-E are arranged as illustrated in Figure 3-3.

The sign bit, exponent bits, and fraction bits in the FP11-E data word have the same values as the data word in memory. Note, however, that the FP11-E data word has more bits than its counterpart in memory. This is because the FP11-E has provisions for generating an overflow bit and a "hidden" bit; for double-precision data words, there are guard bits.

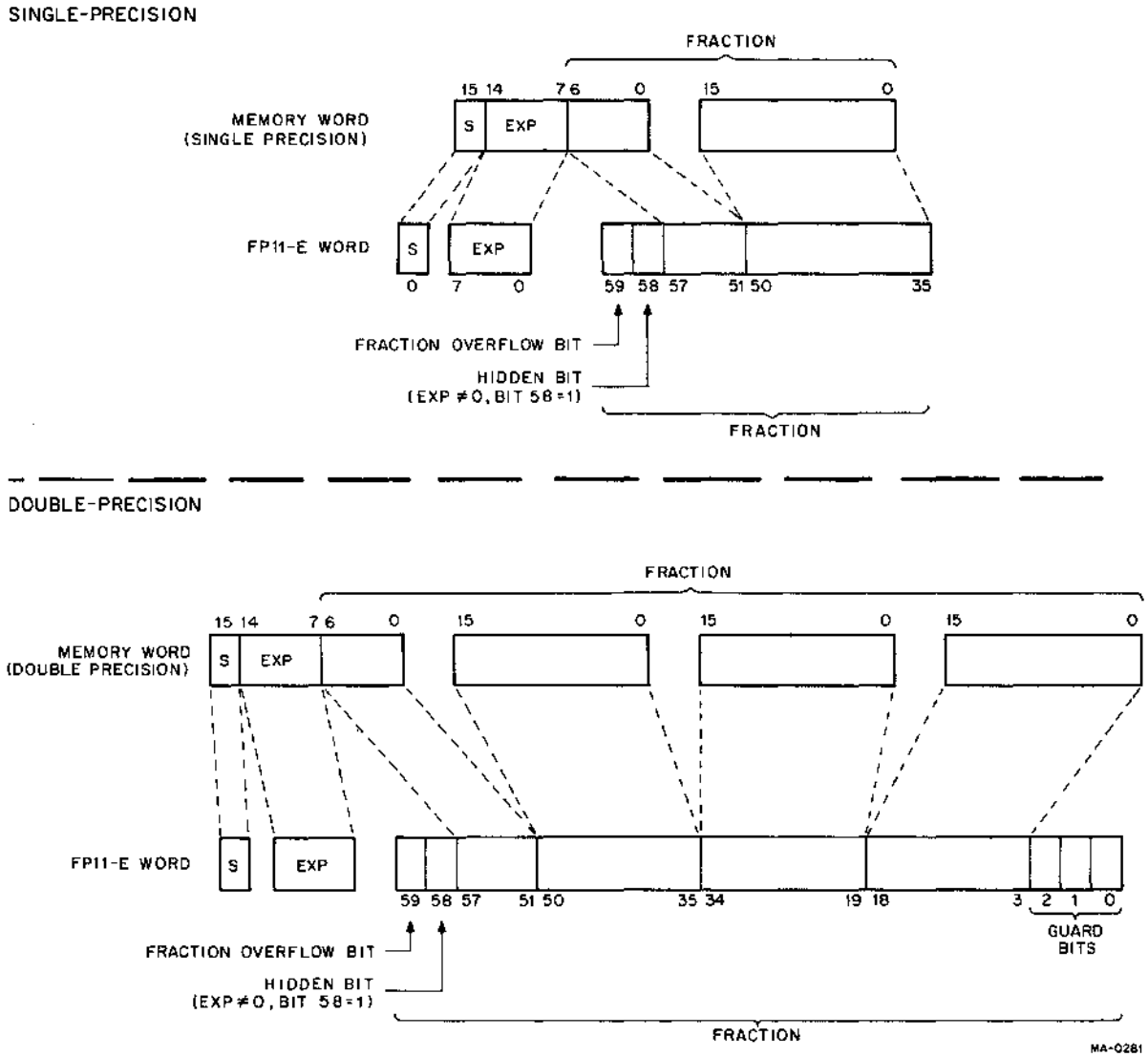


Figure 3-3 FP11-E Floating-Point Data Words

For purposes of discussion, the FP11-E data word can be thought of as being divided into two major parts:

1. A fraction, with its associated sign bit, hidden bit, overflow bit, and (for double-precision) guard bits

2. An exponent.

3.3.1.1 Floating-Point Fraction – The fraction is expressed in sign and magnitude notation. The following simple example illustrates the idea behind sign and magnitude notation.

2's Complement Notation		Sign and Magnitude Notation
+2	000010	Sign 000010 Magnitude
−2	111110	Sign 100010 Magnitude

Only a change of sign bit is required to change the sign of a number in sign and magnitude notation. Note that a positive number is the same in both notations.

Unnormalized floating-point fractions have a range from approximately 0 through 2 as shown in Figure 3-4. The FP11-E, however, normalizes all unnormalized fractions. That is, the fractions are adjusted such that there is a 0 to the left of the binary point (bit 59) and a 1 to the right of the binary point (bit 58). Thus, normalized fractions range in magnitude from 0.1000 . . . to 0.1111 or from 1/2 to approximately 1.

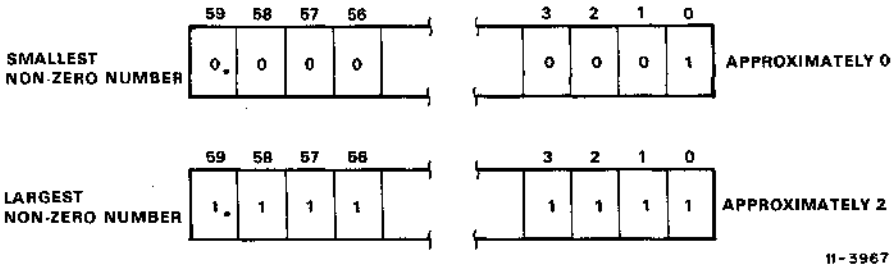


Figure 3-4 Unnormalized Floating-Point Fraction

The fraction overflow bit (bit 59) is set during certain arithmetic operations. For example, during addition, certain sums will produce an overflow such as 0.1000 . . . + 0.100 . . . which yields 1.000 . . . . This result must be normalized, so the FP11-E right-shifts the fraction one place and increases the exponent by one.

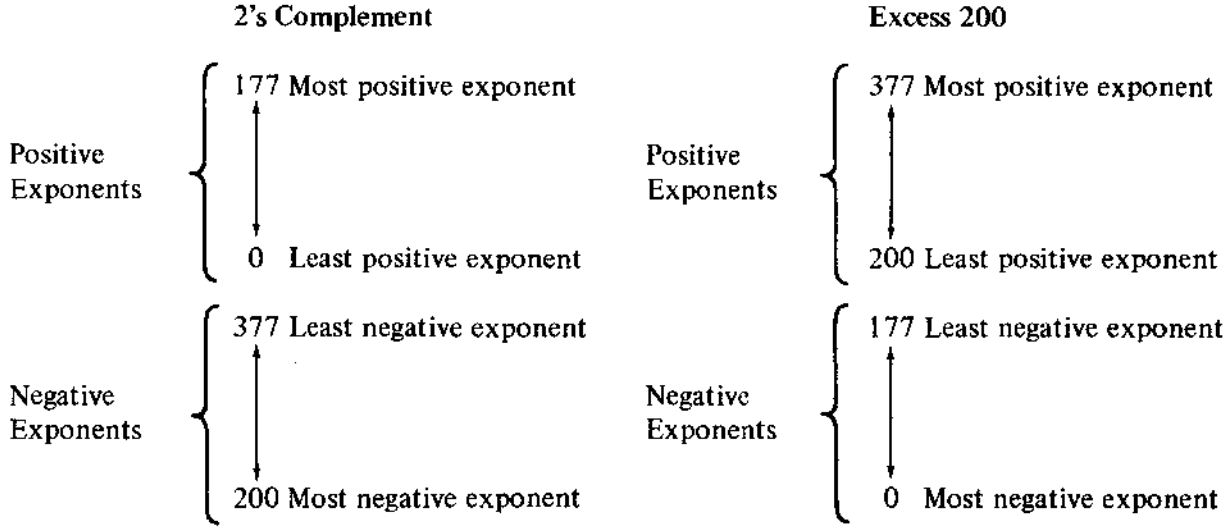
Bit 58 is called the hidden bit. Recall that since fractions are normalized by the FP11-E, the bit immediately to the right of the binary point (bit 58) is always a 1. This bit is dropped when a fraction is sent to memory and appended when a fraction is received from memory. This procedure allows one extra bit of significance in floating-point fraction representation.

The guard bits serve three functions.

1. They allow rounding of a double-precision fraction.
2. They maintain the accuracy of a double-precision fraction that has been shifted right as many as three places.
3. They allow storage of a 59-bit constant during a MOD instruction.

Guard bits are discussed in more detail in the *FP11-E Floating-Point Processor Technical Manual.*

3.3.1.2 Floating-Point Exponent – The 8-bit floating-point exponent is expressed in excess 200 notation. The chart below illustrates the relationship between exponents in 2's complement notation and exponents in excess 200 notation.



Note that an exponent in excess 200 notation is obtained by simply adding 200 to the exponent in 2's complement notation. Thus, 8-bit exponents in excess 200 notation range from 0 to 377 (or from –200 to +177). A number with an exponent in excess of –200 is treated by the FP11-E as 0.

For example, the number 0.1_2 is actually 0.1×2^0, and the exponent is represented as $10\ 000\ 000_2$ because 200_8 represents an exponent of zero. Figure 3-5 illustrates the range of floating-point numbers that can be handled by the FP11-E. For simplicity, a fraction length of only three bits is shown.

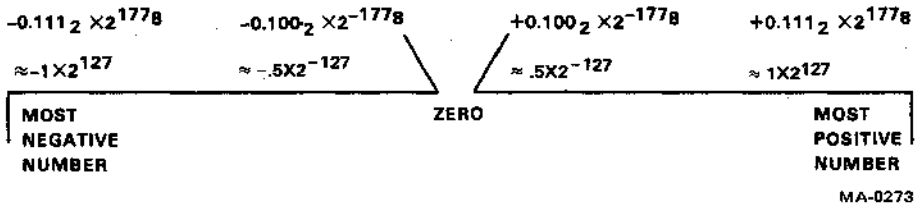


Figure 3-5 Floating-Point Range

A number with an exponent of less than 0 causes an underflow condition, which means that the number is too small to be represented. A number with an exponent of more than 377_8 causes an overflow condition, which means that the number is too large to be represented.

3.3.2 Interpretation of a Floating-Point Number

Floating-point operands or arguments stored in memory are assumed normalized and in sign and magnitude format.

Figure 3-6 shows the decimal number 32 represented in memory in sign and magnitude format. The FP11-E interprets the number as a floating-point number with sign, exponent, and fraction. Only one memory word is shown, which contains the sign, exponent, and upper bits of the fraction. An additional word from memory would be transferred to bits 50 through 35 for single-precision mode. For double-precision mode, two additional words from memory would be transferred to bits 34 through 19 and bits 28 through 03.

The lower half of Figure 3-6 represents the decimal number 7/16 in memory and how it is interpreted by the FP11-E.

3.4 FP11-E STATUS FORMAT

The CPU contains the FP11-E's status register. This register contains FP11-E condition codes (carry, overflow, zero, and negative) that can be copied into the CPU's status register. Specifically, FC, FV, FZ, and FN can be copied into C, V, Z, and N, respectively. The FP11-E status register also contains four mode bits and additional bits to enable various interrupt conditions. Figure 3-7 illustrates the organization of the FP11-E status register and Table 3-1 describes each bit.

3.5 FLOATING-POINT EXCEPTION FORMAT

A total of seven possible interrupts (exceptions) can occur. The interrupt vector used to handle all floating-point exceptions is in location 244_8. When they occur, exceptions are encoded in the FP11-E exception code (FEC) register located in the CPU. The interrupt exception codes represent an offset into a dispatch table, which routes the program to the correct error handling routine. The dispatch table is determined by the software. The code for each exception is shown in Table 3-2, with a brief description.

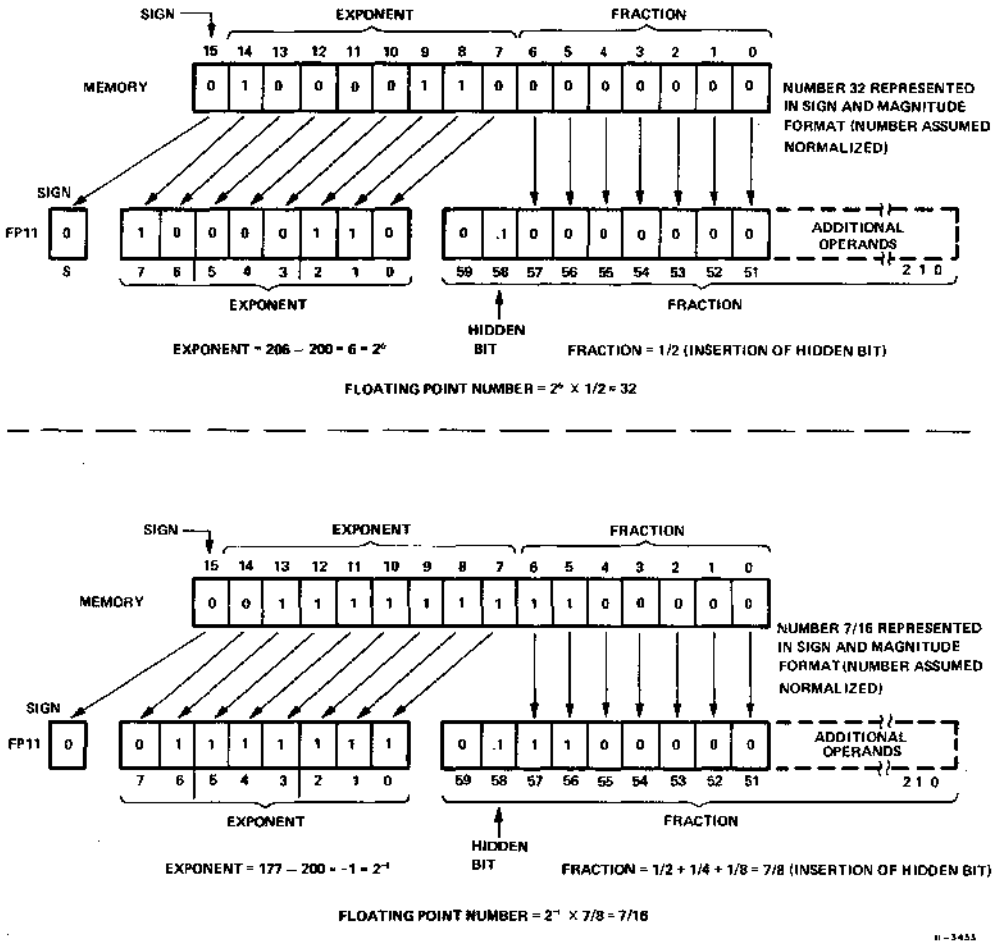


Figure 3-6 Interpretation of Floating-Point Numbers

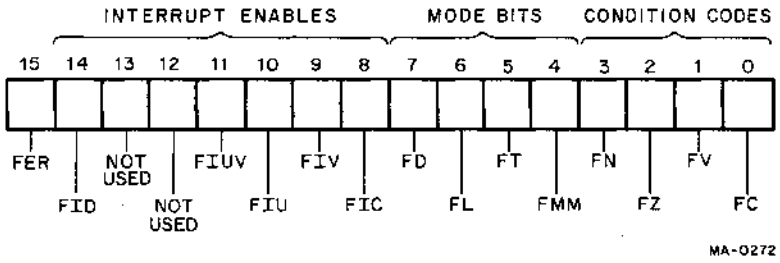


Figure 3-7 FP11-E Status Register Format

Table 3-1 FP11-E Status Register

Bit	Function
FER	This bit indicates an error condition of the FP11-E.
FID	Floating Interrupt Disable – All interrupts by the FP11-E are disabled when this bit is on. Primarily for maintenance use. Normally clear.
FIUV	Floating Interrupt on Undefined Variable – When this bit is set and a –0 is obtained from memory, an interrupt occurs. If the bit is not set, –0 can be loaded and stored; however, any arithmetic operation treats it as if it were a positive 0.
FIU	Floating Interrupt on Underflow – When this bit is set, an underflow condition causes a floating underflow interrupt. The result of the operation causing the interrupt is correct except for the exponent, which is off by 400_8. If the FIU is not set and underflow occurs, the result is set to zero.
FIV	Floating Interrupt on Overflow – When this bit is set, floating overflow causes an interrupt. The result of the operation causing the interrupt is correct except for the exponent, which is off by 400_8. If the FIV bit is not set, the result of the operation is the same; the only difference is that the interrupt does not occur.
FIC	Floating Interrupt on Integer Conversion Error – When this bit is set and the Store Convert Floating-to-Integer instruction causes FC to be set (indicating a conversion error), an interrupt occurs. When a conversion occurs, the destination register is cleared and the source register is untouched. When FIC is reset, the result of the operation is the same; however, no interrupt occurs.
FD	Double-Precision Mode Bit – This bit, when set, specifies double-precision format and, when reset, specifies single-precision format.
FL	Long Precision Integer Mode Bit – This bit is employed during conversion between integer and floating-point format. If set, double-precision 2's complement integer format of 32 bits is specified; if reset, single-precision 2's complement integer format of 16 bits is specified.
FT	Truncate Bit – This bit, when set, causes the result of any floating-point operation to be truncated rather than rounded.
FMM	Maintenance Mode Bit – This bit is used to enable special maintenance logic and is described in the *FP11-E Floating-Point Processor Technical Manual.*
FC, FV, FZ, and FN	These bits are the four floating-point condition codes, which can be loaded in the CPU's C, V, Z, and N condition codes, respectively. This is accomplished by the Copy Floating Condition Codes (CFCC) instruction. To determine how each instruction affects the condition codes, refer to Table 4-1. For the Store Convert Floating-to-Integer instruction (which converts a floating-point number to an integer), the FC bit is set if the resulting integer is too large to be stored in the specified register.

Table 3-2 FP11-E Exception Codes

FP11-E Exception Code	Definition
2	Floating Op Code Error – The FP11-E causes an interrupt for an erroneous op code
4	Floating Divide by Zero – Division by zero causes an interrupt if FID is not set
6	Floating (or Double) Integer Conversion Error
10	Floating Overflow
12	Floating Underflow
14	Floating Undefined Variable
16	Maintenance Trap

NOTE

The traps for exception codes 6, 10, 12, 14, and 16 can be enabled in the FP11-E program status register. All traps are disabled if FID is set.

Refer to the *PDP-11/60 Processor Handbook* for further details concerning FP11-E exceptions.

In addition to the FEC register, the CPU contains a 16-bit floating exception address (FEA) register, which stores the address of the last floating-point instruction that caused a floating-point exception.

CHAPTER 4
FLOATING-POINT INSTRUCTIONS

4.1 FLOATING-POINT ACCUMULATORS

The FP11-E contains six accumulators (AC0–AC5). These accumulators are 64-bit read/write scratchpad memories with non-destructive readout.

Each accumulator is interpreted as being either 64 or 32 bits long, depending on the instruction and the FP11-E status (Chapter 3). If an accumulator is interpreted as being 64 bits long, 64 bits of data occupy the entire accumulator. If an accumulator is interpreted as being 32 bits long, 32 bits of data occupy only the leftmost 32 bits of an accumulator as shown in Figure 4-1.

The floating-point accumulators are used in numeric calculations and interaccumulator data transfers. AC0–AC3 are used for all data transfers between the FP11-E and the CPU or memory.

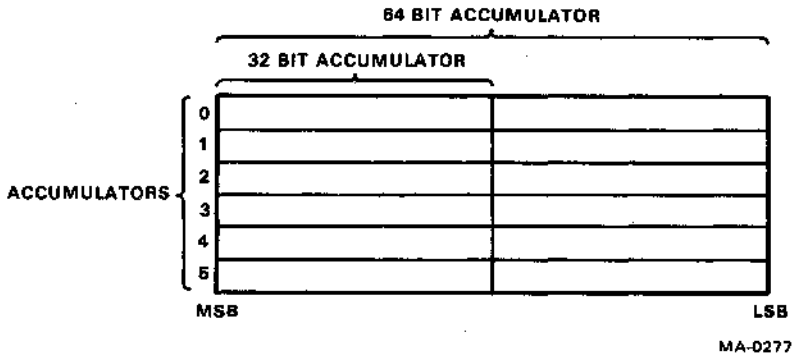


Figure 4-1 Floating-Point Accumulators

4.2 INSTRUCTION FORMATS

An FP11-E instruction must be in one of five formats. These formats are summarized in Figure 4-2.

The 2-bit AC field (bits 6 and 7) allows selection of scratchpad accumulators 0 through 3 only.

If address mode 0 is specified with formats F1 or F2, bits 2 through 0 are used to select a floating-point accumulator. Only accumulators 5 through 0 can be specified in mode 0. If 6 or 7 is specified in bits 2 through 0 in mode 0, the FP11-E traps if floating-point interrupts are enabled (FID = 0). The FEC will indicate an illegal op code error (exception code 2).

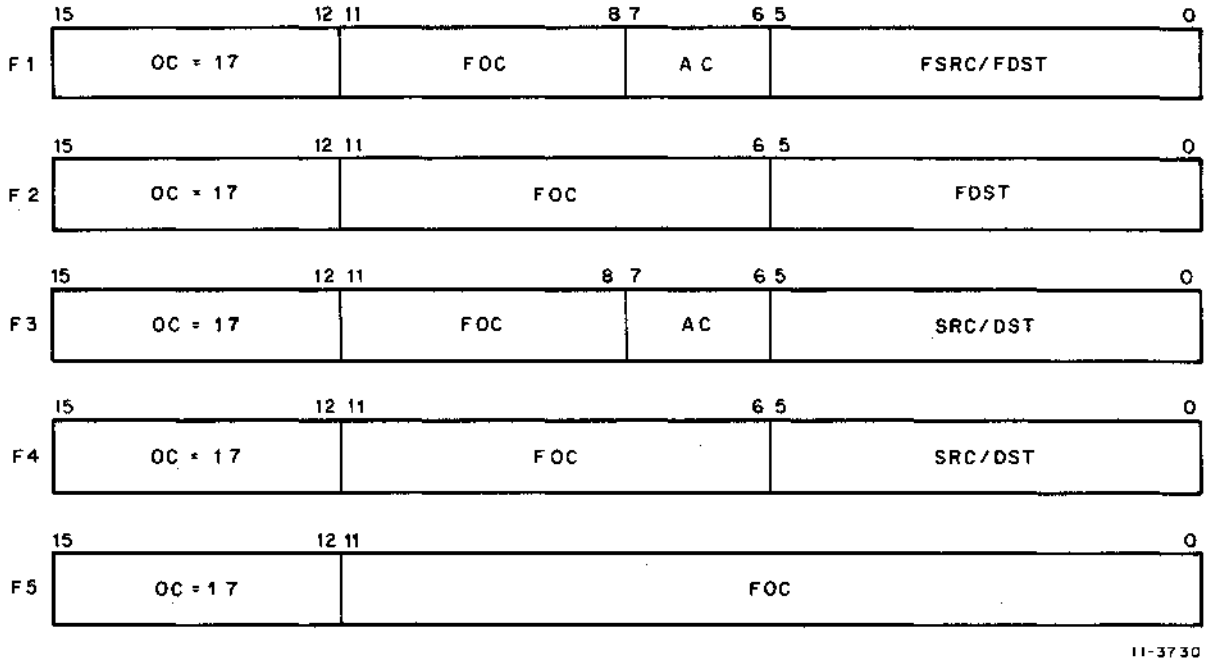


Figure 4-2 Instruction Formats

The fields of the various instruction formats (as summarized in Table 4-1) are interpreted as follows:

Mnemonic	Description
OC	Operation Code – All floating-point instructions are designated by a 4-bit op code of 17_8.
FOC	Floating Operating Code – The number of bits in this field varies with the format; the code is used to specify the actual floating-point operation.
SRC	Source – A 6-bit source field identical to that in the PDP-11 instruction.
DST	Destination – A 6-bit destination field identical to that in a PDP-11 instruction.
FSRC	Floating Source – A 6-bit field used only in format F1. It is identical to SRC, except in mode 0 when it references a floating-point accumulator rather than a CPU general register.
FDST	Floating Destination – A 6-bit field used in formats F1 and F2. It is identical to DST, except in mode 0 when it references a floating-point accumulator instead of a CPU general register.
AC	Accumulator – A 2-bit field used in formats F1 and F3 to specify FP11-E scratchpad accumulators 0 through 3.

Table 4-1 Format of FP11-E Instructions

Instruction Format	Instruction	Mnemonic
F2	ABSOLUTE	ABSF FDST ABSD FDST
F1	ADD	ADDF FSRC, AC ADDD FSRC, AC
F2	CLEAR	CLRF FDST CLRD FDST
F1	COMPARE	CMPF FSRC, AC CMPD FSRC, AC
F5	COPY FLOATING CONDITION CODES	CFCC
F1	DIVIDE	DIVF FSRC, AC DIVD FSRC, AC
F1	LOAD	LDF FSRC, AC LDD FSRC, AC
F1	LOAD CONVERT	LDCFD FSRC, AC FDCDF FSRC, AC
F3	LOAD CONVERT INTEGER TO FLOATING	LDCIF SRC, AC LDCID SRC, AC LDCLF SRC, AC LDCLD SRC, AC
F3	LOAD EXPONENT	LDEXP SRC, AC
F4	LOAD FP11'S PROGRAM STATUS	LDFPS SRC
F1	MODULO	MODF FSRC, AC MODD FSRC, AC
F1	MULTIPLY	MULF FSRC, AC MULD FSRC, AC
F2	NEGATE	NEGF FDST NEGD FDST
F5	SET DOUBLE MODE	SETD
F5	SET FLOATING MODE	SETF
F5	SET INTEGER MODE	SETI
F5	SET LONG INTEGER MODE	SETL
F1	STORE	STF AC, FDST STD AC, FDST
F1	STORE CONVERT	STCFD AC, FDST STCDF AC, FDST
F3	STORE CONVERT FLOATING TO INTEGER	STCFI AC, DST STCFL AC, DST STCDI AC, DST STCDL AC, DST
F3	STORE EXPONENT	STEXP AC, DST
F4	STORE FP11'S PROGRAM STATUS	STFPS DST
F4	STORE FP11'S STATUS	STST DST
F1	SUBTRACT	SUBF FSRC, AC SUBD FSRC, AC
F2	TEST	TSTF FDST TSTD FDST

4.3 INSTRUCTION SET

Table 4-2 contains the instruction set of the FP11-E. Some of the symbology may not be familiar. Therefore, a brief description follows.

1. A floating-point flip-flop, designated FD, determines whether single- or double-precision floating-point format is specified. If the flip-flop is cleared, single-precision is specified and is designated by F. If the flip-flop is set, double-precision is specified and is designated by D. Examples are NEGF, NEGD, and SUBD.

2. An integer flip-flop, designated FL, determines whether short integer or long integer format is specified. If the flip-flop is cleared, short integer format is specified and is designated by I. If the flip-flop is set, long integer format is specified and is designated by L. Examples are SETI and SETL.

3. Several convert type instructions use the symbology defined below.

   $C_{IL,FD}$ – Convert integer to floating

   $C_{FD,IL}$ – Convert floating to integer

   $C_{F,D}$ or $C_{D,F}$ – Convert single-floating to double-floating or convert double-floating to single-floating

4. UPLIM is defined as the largest possible number that can be represented in floating-point format. This number has an exponent of 377 (excess 200 notation) and a fraction of all 1s. Note the UPLIM is dependent on the format specified. LOLIM is defined as the smallest possible number that is not identically zero. This number has an exponent of 001 and a fraction of all 0s except for the hidden bit.

5. The following conventions are used when referring to address locations.

   (xxxx) = the contents of the location specified by xxxx
   ABS (address) = absolute value of (address)
   EXP (address) = exponent of (address) in excess 200 notation

6. Some of the octal codes listed in Table 4-2 are in the form of mathematical expressions. These octal codes can be calculated as shown in the following examples.

   Example 1: LDFPS Instruction

   Mode 3, register 7 specified (F instruction format).

   170100 + SRC
   SRC field is equal to 37
   Basic op code is 170100
   SRC and basic op code are added to yield 170137.

Example 2: LDF Instruction

AC2, mode 2, and register 6 specified (F1 instruction format).

172400 + C * 100 + FSRC

AC = 2

2 * 100 = 200

172400 + 200 = 172600
FSRC is equal to 26

172600 + 26 + 172626

7. AC v 1 means that the accumulator field (bits 6 and 7 in formats F1 and F3) is logically ORed with 01.

Example:

Accumulator field = bits 6 and 7 = AC2 = 10. AC v 1 = 11.

The information in Table 4-2 is expressed in symbolic notation to provide the reader with a quick reference to the function of each instruction. The following paragraphs supplement the information in Table 4-2.

Table 4-2 FP11-E Instruction Set

Mnemonic	Instruction Description	Octal Code
ABSF FDST ABSD FDST	Absolute FDST ← minus (FDST) if FDST ⩽ 0; otherwise FDST ← (FDST) FC ← 0 FV ← 0 FZ ← 1 if exp (FDST) = 0; otherwise FZ ← 0 FN ← 0	170600+FDST F2 Format
ADDF FSRC, AC ADDD FSRC, AC	Floating Add AC ← (AC) + (FSRC) if |AC| + (FSRC) ⩽ LOLIM; otherwise AC ← 0 FC ← 0 FV ← 1 if |AC| ⩾ UPLIM; otherwise FV ← 0 FZ ← if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0	172000+AC*100+FSRC F1 Format

Table 4-2 FP11-E Instruction (Cont)

Mnemonic	Instruction Description	Octal Code
CLRF FDST CLRD FDST	Clear FDST ← 0 FC ← 0 FV ← 0 FZ ← 1 FN ← 0	170400+FDST F2 Format
CMPF FSRC, AC CMPD FSRC, AC	Floating Compare FC ← 0 FV ← 0 FZ ← 1 if (FSRC) – (AC) = 0; otherwise FZ ← 0 FN ← 1 if (FSRC) – (AC) < 0; otherwise FN ← 0	173400+AC*100+FSRC F1 Format
CFCC	Copy Floating Condition Codes C ← FC V ← FV Z ← FZ N ← FN	170000 F5 Format
DIVF FSRC, AC DIVD FSRC, AC	Floating Divide AC ← (AC)/(FSRC) if |(AC)/(FSRC)| ⩾ LOLIM; otherwise AC ← 0 FC ← 0 FV ← 1 if |AC| ⩾ UPLIM; otherwise FV ← 0 FZ ← 1 if EXP (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0	174400+AC*100+FSRC F1 Format

LDF FSRC, AC
or
LDD FSRC, AC
	Floating Load
	AC ← (FSRC)
	FC ← 0
	FV ← 0
	FZ ← 1 if (AC) = 0; otherwise FZ ← 0
	FN ← 1 if (AC) < 0; otherwise FN ← 0
		172400+AC*100+FSRC
		F1 Format

Table 4-2 FP11-E Instruction (Cont)

Mnemonic	Instruction Description	Octal Code
LDCDF FSRC, AC LDCFD FSRC, AC	Load Convert Double-to-Floating or Floating-to-Double AC ← $C_{F,D}$ or $C_{D,F}$ (FSRC) FC ← 0 FV ← 1 if |AC| ⩾ UPLIM; otherwise FV ← 0 FZ ← 1 if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0 If the current format is single-precision floating-point (FD = 0), the source is assumed to be a double-precision number and is converted to single-precision. If the floating truncate bit is set, the number is truncated; otherwise, it is rounded. If the current format is double-precision (FD = 1), the source is assumed to be a single-precision number and loaded left-justified in the AC. The lower half of the AC is cleared.	177400+AC*100+FSRC F1 Format F, D-single-precision to double-precision floating D, F-double-precision to single-precision floating
LDCIF SRC, AC LDCID SRC, AC LDCLF SRC, AD LDCLD SRC, AC LDCIF = Single Integer to Single Float LDCID = Single Integer to Double Float LDCLF = Long Integer to Single Float LDCLD = Long Integer to Double Float	Load and Convert from Integer to Floating AC ← $C_{IL,FD}$ (SRC) FC ← 0 FV ← 0 FZ ← 1 if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0 $C_{IL,FD}$ specifies conversion from a 2's complement integer with precision I or L to a floating-point number of precision F or D. If integer flip-flop IL = 0, a 16-bit integer (I) is double specified, and if IL = 1, a 32-bit integer (L) is specified. If floating-point flip-flop FD = 0, a 32-bit floating-point number (F) is specified, and if FD = 1, a 64-bit floating point number (D) is specified. If a 32-bit integer is specified and addressing mode 0 or immediate mode is used, the 16 bits of the source register are left justified, and the remaining 16 bits are zeroed before the conversion.	177000+AC*100+SRC F3 Format

Table 4-2 FP11-E Instruction (Cont)

Mnemonic	Instruction Description	Octal Code
LDEXP SRC, AC	Load Exponent AC SIGN ← (AC SIGN) AC EXP ← (SRC) + 200 only if ABS (SRC) < 177 AC FRACTION ← (AC FRACTION) FC ← 0 FV ← 1 if (SRC) > 177; otherwise FV ← 0 FZ ← 1 if EXP (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0	176400+AC*100+SRC F3 Format
LDFPS SRC	Load FP11-E's Program Status Word FPS ← (SRC)	170100+SRC F4 Format
MODF FSRC, AC MODD FSRC, AC	Floating Modulo AC v 1 ← integer part of (AC) * (FSRC) AC ← fractional part of (AC) * (FSRC) − (AC v 1) if |(AC) * (FSRC)| ⩾ LOLIM or FIU = 1; otherwise AC ← 0 FC ← 0 FV ← 1 if |AC| ⩾ UPLIM; otherwise FV ← 0 FZ ← 1 if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0 The product of (AC) and (FSRC) is 48 bits in single-precision floating-point format or 59 bits in double-precision floating-point format. The integer part of the product [(AC) * (FSRC)] is found and stored in AC v 1. The fractional part is then obtained and stored in AC. Note that multiplication by 10 can be done with zero error, allowing decimal digits to be stripped off with no loss in precision.	171400+AC*100+FSRC F1 Format
MULF FSRC, AC MULD FSRC, AC	Floating Multiply AC ← (AC) * (FSRC) if |(AC) * (FSRC)| ⩾ LOLIM; otherwise AC ← 0 FC ← 0 FV ← 1 if |AC| ⩾ UPLIM; otherwise FV ← 0 FZ ← 1 if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0	171000+AC*100FSRC F1 Format
NEGF FDST NEGD FDST	Negate FDST ← minus (FDST) if EXP (FDST) ≠ 0; otherwise FDST ← 0 FC ← 0 FV ← 0 FZ ← 1 if if EXP (FDST) = 0; otherwise FZ ← 0 FN ← 1 if (FDST) < 0; otherwise FN ← 0	170700+FDST F2 Format

Table 4-2 FP11-E Instruction (Cont)

Mnemonic	Instruction Description	Octal Code
SETD	Set Floating Double Mode FD ← 1	170011 F5 Format
SETF	Set Floating Mode FD ← 0	170001 F5 Format
SETI	Set Integer Mode FL ← 0	170002 F5 Format
SETL	Set Long Integer Mode FL ← 1	170012 F5 Format
STF AC, FDST STD AC, FDST	Floating Store FDST ← (AC) FC ← FC FV ← FV FZ ← FZ FN ← FN	174000+AC*100+FDST F1 Format
STCFD AC, FDST STCDF AC, FDST	Store convert from Floating to Double or Double to Floating FDST ← $C_{F,D}$ or $C_{D,F}$ (AC) FC ← 0 FV ← 1 if |AC| ⩾ UPLIM; otherwise FV ← 0 FZ ← 1 if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0	176000+AC*100+FDST F1 Format F, D-single-precision to double-precision floating D, F-double-precision to single-precision floating
STCFI AC, DST STCFL AC, DST STCDI AC, DST STCDL AC, DST	Store Convert from Floating to Integer Destination receives converted AC if the resulting integer number can be represented in 16 bits (short integer) or 32 bits (long integer). Otherwise, destination is zeroed and C bit is set.	175400+AC*100+DST F3 Format
STCFI = Single Float to Single Integer STCFL = Single Float to Long Integer STCDI = Double Float to Single Integer STCDL = Double Float to Long Integer	FV ← 0 FZ ← 1 if (DST) = 0; otherwise FZ ← 0 FN ← 1 if (DST) < 0; otherwise FN ← 0 C ← FC V ← FV Z ← FZ N ← FN When the conversion is to long integer (32 bits) and address mode 0 or immediate mode is specified, only the most significant 16 bits are stored in the destination register.	

Table 4-2 FP11-E Instruction (Cont)

Mnemonic	Instruction Description	Octal Code
STEXP AC, DST	Store Exponent DST ← AC EXPONENT – 200_8 FC ← 0 FV ← 0 FZ ← 1 if (DST) = 0; otherwise FZ ← 0 FN ← 1 if (DST) < 0; otherwise FN ← 0 C ← FC V ← FV Z ← FZ N ← FN	175000+C*100+DST F3 Format
STFPS DST	Store FP11-E's Program Status Word DST ← (FPS)	170200+DST F4 Format
STST DST	Store FP11-E's Status DST ← (FEC) DST + 2 ← (FEA) if not mode 0 or not immediate mode	170300+DST F4 Format
SUBF FSRC, AC SUBD FSRC, AC	Floating Subtract AC ← (AC) – (FSRC) if |(AC) – (FSRC)| ⩾ LOLIM; otherwise AC ← 0 FC ← 0 FV ← 1 if AC UPLIM; otherwise FV ← 0 FZ ← 1 if (AC) = 0; otherwise FZ ← 0 FN ← 1 if (AC) < 0; otherwise FN ← 0	173000+AC*100+FSRC F1 Format
TSTF FDST TSTD FDST	Test FDST ← (FDST) FC ← 0 FV ← 0 FZ ← 1 if EXP (FDST) = 0; otherwise FZ ← 0 FN ← 1 if (FDST) < 0; otherwise FN ← 0	170500+FDST F2 Format

4.3.1 Arithmetic Instructions

The arithmetic instructions (Add, Subtract, Multiply, Divide) require one operand in a source (a floating-point accumulator in mode 0, a memory location otherwise) and one operand in a destination accumulator. The instruction is executed by the FP11-E and the result is stored in the destination accumulator.

The Compare instruction also requires one operand in a source and one operand in a destination accumulator. However, the two operands remain in their respective locations after the instruction is executed by the FP11-E, and there is no transfer of the result.

4.3.2 Floating Modulo Instruction

The Floating Modulo (MOD) instruction causes the FP11-E to multiply two floating-point operands, separate the product into integer and fractional parts, and store one or both parts as floating-point numbers. The whole number portion goes into an odd-numbered accumulator and the fraction goes into an even-numbered accumulator.

The whole number portion of the number, when expressed as a floating-point number, contains an exponent greater than 201 in excess 200 notation, which means that the whole number has a decimal value of some number greater than one and less than UPLIM, where UPLIM is the greatest possible number that can be represented by the FP11-E.

The fractional portion of the number, when expressed as a floating-point number, contains an exponent less than or equal to 201 in excess 200 notation. This means that the fraction has a value less than one and greater than LOLIM, where LOLIM is the smallest possible number that can be represented by the FP11-E.

4.3.3 Load Instruction

The Load instruction causes the FP11-E (and the CPU, if not in mode 0) to take an operand from a source and copy it into a destination accumulator. The source is a floating-point accumulator in mode 0 and a memory location otherwise.

4.3.4 Store Instruction

The Store instruction causes the FP11-E (and the CPU, if not in mode 0) to take an operand from a source accumulator and transfer it to a destination. This destination is a floating-point accumulator in mode 0 and a memory location otherwise.

4.3.5 Load Convert (Double-to-Floating, Floating-to-Double) Instructions

The Load Convert Double-to-Floating (LDCDF) instruction causes the FP11-E to assume that the source specifies a double-precision floating-point number. The FP11-E then converts that number to single-precision, and places this result in the destination accumulator. If the floating truncate status bit (FT) is set, the number is truncated. If FT is not set, the number is rounded by adding a 1 to the single-precision segment if the MSB of the double-precision segment is a 1, as shown in Figure 4-3. If the MSB of the double-precision segment is 0, the single-precision word remains unchanged after rounding.

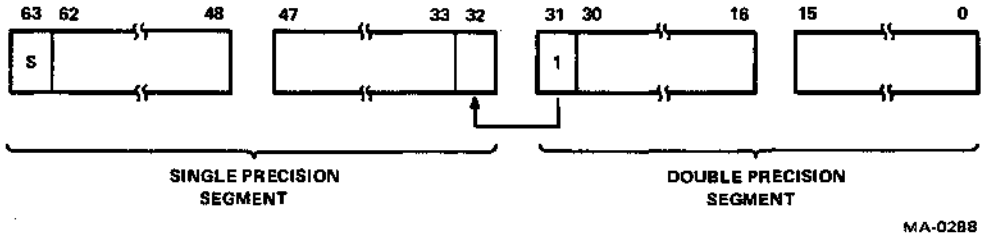


Figure 4-3 Double-to-Single Precision Rounding

The Load Convert Floating-to-Double (LDCFD) instruction causes the FP11-E to assume that the source specifies a single-precision number. The FP11-E then converts that number to double-precision by appending 32 zeros to the single-precision word, and places this result in the destination accumulator.

Note that for both Load Convert instructions, the number to be converted is originally in the source (a floating-point accumulator in mode 0, a memory location otherwise) and is transferred to the destination accumulator after conversion.

4.3.6 Store Convert (Double-to-Floating, Floating-to-Double) Instructions

The Store Convert Double-to-Floating (STCDF) instruction causes the FP11-E to convert a double-precision number located in the source accumulator to a single-precision number. The FP11-E then transfers this result to the specified destination. If the floating truncate (FT) bit is set, the floating-point number is truncated. If FT is not set, the number is rounded. If the MSB (bit 31) of the double-precision segment of the word is a 1, 1 is added to the single-precision segment of the word (Figure 4-3); otherwise, the single-precision segment remains unchanged.

The Store Convert Floating-to-Double (STCFD) instruction causes the FP11-E to convert a single-precision number located in the source accumulator to a double-precision number. The FP11-E then transfers this result to the specified destination. The single-to-double precision is obtained by appending zeros equivalent to the double-precision segment of the word as shown in Figure 4-4.

Note that for both Store Convert instructions, the number to be converted is originally in the source accumulator and is transferred to the destination (a floating-point accumulator in mode 0, a memory location otherwise) after conversion.

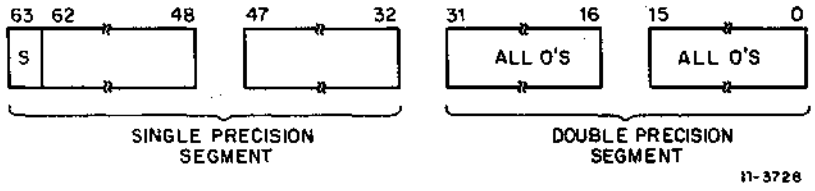


Figure 4-4 Single-to-Double Precision Appending

4.3.7 Clear Instruction

The Clear instruction causes the CPU (or the FP11-E, in mode 0) to clear a floating-point number by setting all its bits to 0. If in mode 0, the FP11-E microcode writes zeros into the source accumulator, which is then transferred to a floating-point accumulator. If not in mode 0, the CPU writes 0s into the source accumulator, which is then transferred to a memory location.

4.3.8 Test Instruction

The Test instruction causes the CPU (or the FP11-E, in mode 0) to test the sign and exponent of a floating-point number and update the FP11-E status accordingly. The number tested is obtained from the destination (a floating-point accumulator in mode 0, a memory location otherwise). The FC and FV bits are cleared. The FN bit is set only if the destination is negative. The FZ bit is set only if the exponent of the destination is zero. If the FIUV status bit is set, a trap occurs (after the test instruction is executed) if a minus zero is encountered.

4.3.9 Absolute Instruction

The Absolute instruction causes the CPU (or the FP11-E, in mode 0) to take the absolute value of a floating-point number by forcing its sign bit to 0. If mode 0 is specified, the sign of the number in the floating-point destination accumulator is forced to 0. The exponent of the number is tested, and if it is 0, zeros are written into the accumulator. If the exponent is non-zero, the accumulator is unaffected.

If mode 0 is not specified, the sign bit of the specified data word in memory is zeroed. This word is then transferred from memory to a floating-point accumulator. The exponent of this word is tested, and if it is 0, the entire data word is zeroed and transferred back to memory. If the exponent is non-zero, the original fraction and exponent are restored to memory.

Absolute and Negate instructions are the only instructions that can read and write a memory location.

4.3.10 Negate Instruction

The Negate instruction causes the CPU (or the FP11-E, in mode 0) to complement the sign of an operand. If mode 0 is specified, the sign of the number in the floating-point destination accumulator is complemented. The exponent of the number is tested, and if it is 0, zeros are written into the accumulator. If the exponent is non-zero, the accumulator is unaffected.

If mode 0 is not specified, the sign bit of the specified data word in memory is complemented. This word is then transferred from memory to a floating-point accumulator. The exponent of this word is tested, and if it is 0, the entire data word is zeroed and transferred back to memory. If the exponent is non-zero, the original fraction and exponent are restored to memory.

4.3.11 Load Exponent Instruction

The Load Exponent instruction causes the CPU to load an exponent from the source (a floating-point accumulator in mode 0, a memory location otherwise) into the exponent field of the destination accumulator. In order to do this, the 16-bit, 2's complement exponent from the source must be converted (by the CPU) to an 8-bit number in excess 200 notation. This process is described further below.

Assume that the 16-bit, 2's complement exponent is coming from memory. The possible legal range of 16-bit numbers in memory is from 000000 to 177777_8. On the other hand, the possible legal range of exponents in the FP11-E falls into two classes:

1. Positive exponents (0 through 177) – When 200 is added to any of these numbers, the sum stays within the legal 8-bit exponent field (i.e., from 200 through 377).

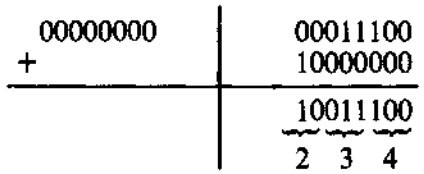
2. Negative exponents (177601 through 177777) – When 200 is added to any of these numbers, the sum stays within the legal 8-bit exponent field (i.e., from 1 through 177).

Notice that all legal positive exponents coming from memory have something in common: their nine high-order bits are all 0s. Similarly, all legal negative exponents from memory have their nine high-order bits equal to 1. Therefore, to detect a legal exponent, only the nine high-order bits need be examined for all 1s or all 0s.

Any number from memory outside these ranges is illegal and will result in either an overflow or an underflow trap condition.

Example 1: LDEXP 000034

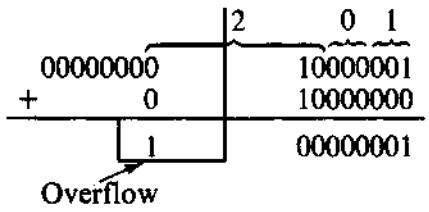
Exponent of 34
200



The upper nine bits all equal 0, so this is a legal positive exponent. The number 234 is sent to the 8-bit exponent field of the specified accumulator.

Example 2: LDEXP 201

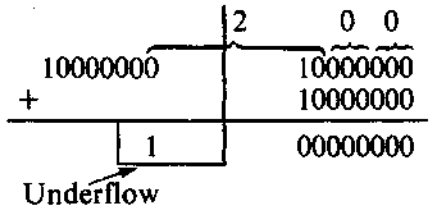
Exponent of 201
200



This is an illegal positive exponent. Notice that when 200 is added to the exponent, an overflow occurs.

Example 3: LDEXP 100200

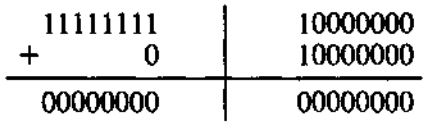
Exponent of 100200
200



This is an illegal negative exponent. Notice that when 200 is added to the exponent, a result is produced that is more negative than can be expressed by the 8-bit exponent field. Thus, an underflow occurs.

Example 4: Special Case – Exponent of 0: LDEXP 177600

Exponent of 177600



This is the one case where the nine high-order bits are all equal, but the exponent is illegal. This is because 177600 represents an exponent of 0. This exponent causes an underflow condition to exist; that is, it is treated as an illegal negative exponent.

4.3.12 Load Convert Integer-to-Floating Instruction

The Load Convert Integer-to-Floating instruction causes the FP11-E to take a 2's complement integer from a source and convert it to a floating-point number. If short integer mode is specified, the number from the source is 16 bits and is converted to a 24-bit fraction (single-precision) or a 56-bit fraction (double-precision), depending on whether floating- or double-precision mode is specified. If long integer mode is specified, the number from the source is 32 bits and is converted to a single-precision or double-precision number, depending on whether floating or double mode is specified.

The integer is loaded into FP11-E bits 34 through 19 if short integer mode is specified or into FP11-E bits 34 through 03 if long integer mode is specified (Figure 4-5). The other bits of the FP11-E word are loaded with zeros.

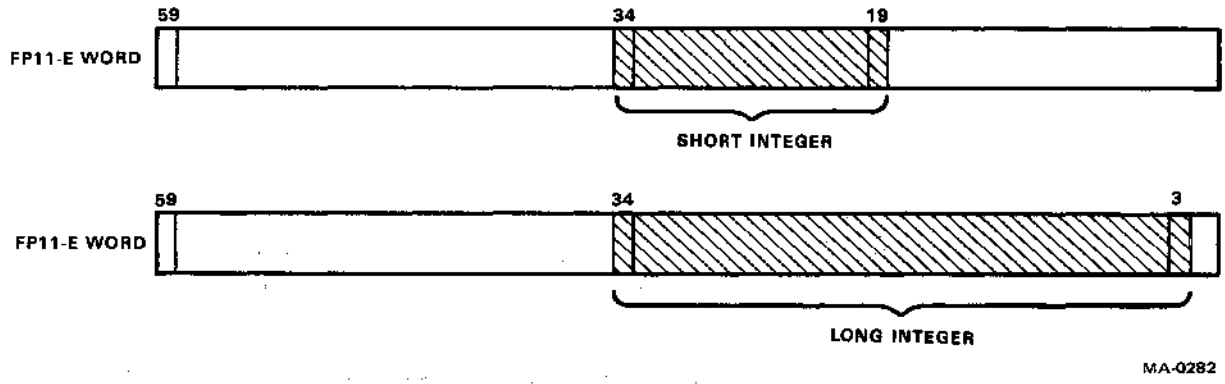
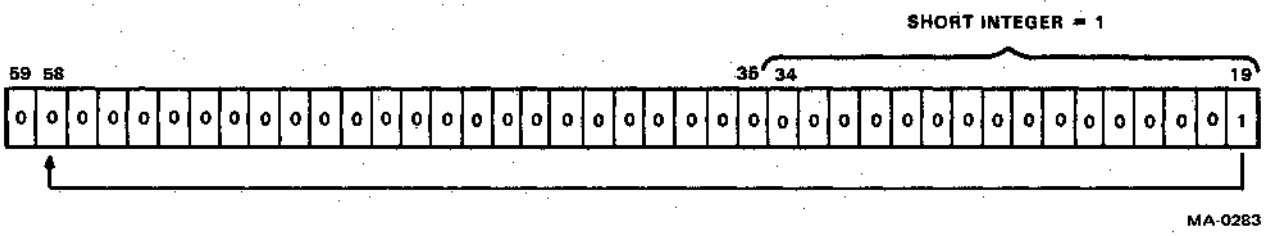


Figure 4-5 Integer Loading

The most significant bit (MSB) of the integer (bit 34) is examined. If the MSB is zero, the integer is either positive or zero. To determine if the integer equals zero, the 2's complement of bits 59 through 19 (short) or 59 through 03 (long) is taken. If bit 59 is zero as a result of this operation, the integer was zero (since the 2's complement of any other positive integer will yield a negative result, setting bit 59) and zero is stored in the specified accumulator. If the (uncomplemented) integer is positive, 250_8 (short) or 270_8 (long) is temporarily assigned as the integer's exponent. The integer is then normalized by shifting it left until bit 59 = 0 and bit 58 = 1. The exponent is decreased by the number of left shifts. A simple example illustrates this process.

Example: Integer of 1 (short)



Shift integer 39 places to the left to normalize.

Bit 59 = 0; bit 58 = 1.

Decrease temporary exponent by $39_{10} = 47_8$.

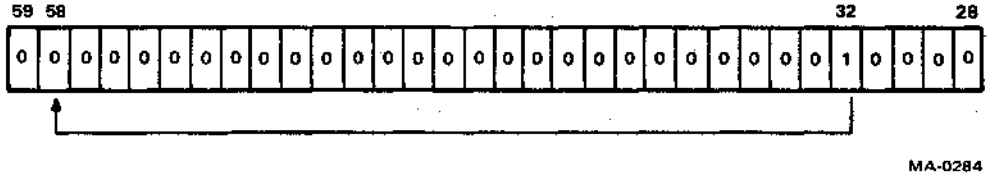
$$\begin{array}{r} 250_8 \\ -\ 47_8 \\ \hline 201_8 \end{array}$$

The integer has been converted to a floating-point number with a fraction of 0.1_2 and an exponent of 201_8 (excess 200 notation). The sign has already been determined; it is positive.

If the MSB is one, the integer is negative. The integer is shifted 25 places left so that its MSB is bit 59. The 2's complement of the integer is taken and 217_8 (short) or 236_8 (long) is temporarily assigned as the integer's exponent. The integer is normalized and its temporary exponent is decreased by the number of left shifts.

Example: Integer of -16 (Long)

NOTE
Long integer of -16 has already been shifted 25 places and 2's complemented.



Shift integer 26 places to the left to normalize.

Bit 59 = 0; bit 58 = 1

Decrease temporary exponent by $26_{20} = 32_8$.

$$\begin{array}{r} 237_8 \\ -\ 32_8 \\ \hline 205_8 \end{array}$$

The integer has been converted to a floating-point number with a fraction of 0.1_2 and an exponent of 205_8 (excess 200 notation). The sign has already been determined; it is negative.

4.3.13 Store Exponent Instruction

The Store Exponent (STEXP) instruction causes the CPU to access a floating-point number in the FP11-E, extract the 8-bit exponent field from this number, and subtract a constant of 200 (since the exponent is expressed in excess 200 notation). The exponent is then stored in the destination as a 16-bit, 2's complement, right-justified number with the sign of the exponent (bit 07) extended through the eight high-order bits.

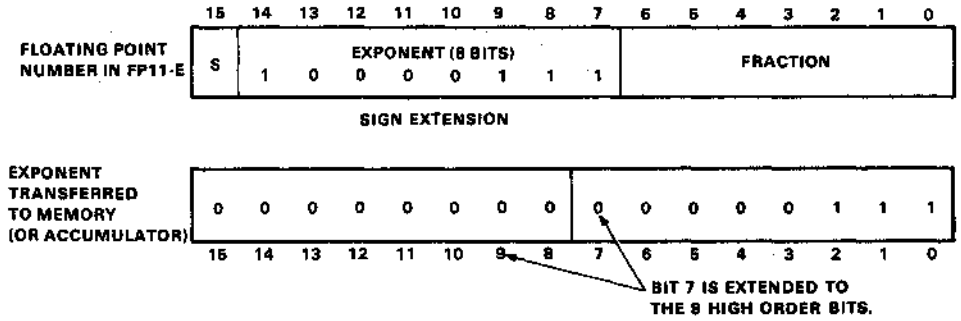
The legal range of exponents is from 0 to 377, expressed in excess 200 notation. This means that the number stored ranges from -200 to 177 after the constant of 200 has been subtracted. The subtraction of 200 is accomplished by taking the 2's complement of 200 and adding it to the exponent field.

Two examples that illustrate the process follow: one using an exponent greater than 200 and the next using an exponent less than 200.

Example 1: Exponent = 207

Exponent of 207	10000111
2's Complement of 200	+10000000
Result = 7	00000111

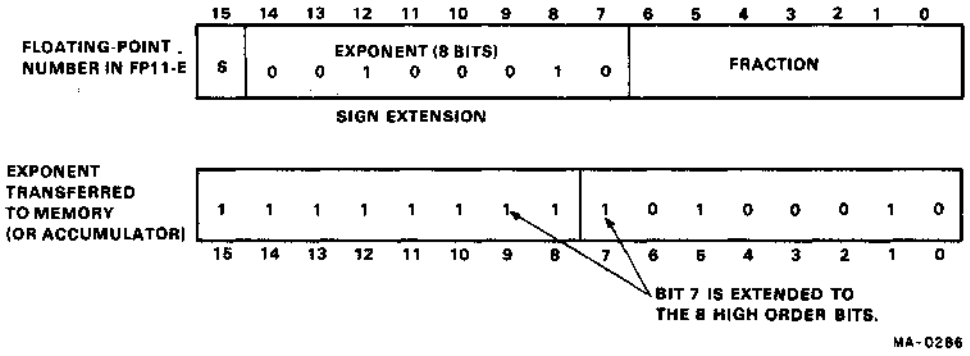
Sign Bit → 7



Example 2: Exponent = 42

Exponent of 42 — 00100010
2's Complement of 200 — +10000000
Result = –42 — 10100010

Sign Bit → 4 2



4.3.14 Store Convert Floating-to-Integer Instruction

The Store Convert Floating-to-Integer instruction causes the CPU to take a floating-point number and convert it to an integer for transfer to a destination.

The four classes of this instruction are:

1. STCFI – Convert single-precision, 24-bit fraction to a 16-bit integer (short integer mode).
2. STCFL – Convert single-precision, 24-bit fraction to a 32-bit integer (long integer mode).
3. STCDI – Convert double-precision, 56-bit fraction to a 16-bit integer (short integer mode).
4. STCDL – Convert double-precision, 56-bit fraction to a 32-bit integer (long integer mode).

The (normalized) floating-point number to be converted is transferred to the CPU. The CPU works with the sign bit and one of the following.

1. The 15 MSBs of the fraction for F-to-I and D-to-I conversion
2. The 31 MSBs of the fraction for D-to-L conversion
3. The entire fraction for F-to-L conversion.

The CPU subtracts 201 from the exponent to determine if the floating-point number is a fraction. If the result of the subtraction is negative, the exponent is less than 201, and the absolute value of the floating-point number is less than 1. When converted to an integer, the value of this number is 0; a conversion error occurs, the FZ bit is set, and 0s are sent to the destination. If the result of the subtraction is positive (or zero), it indicates that the exponent is greater than (or equal to) 201, and the floating-point number can be converted to a non-zero integer.

A second test is made by the CPU to determine if the floating-point number to be converted is within the range of numbers which can be represented by a 16-bit integer (I format) or 32-bit integer (L format).

Consider the range of integers that can be represented in I and L formats and their floating-point equivalents.

	I Format (16 bits)	**Floating-Point Equivalent**	**L Format (32 bits)**	**Floating-Point Equivalent**
Most Positive Integer	077777	$+.1111...X2^{15}$	17777777777	$+.1111...X2^{31}$
Least Positive Integer	000001	$+.100...X2^{1}$	00000000001	$+.100...X2^{1}$
Least Negative Integer	177777	$-.1111...X2^{16}$	37777777777	$-.1111...X2^{32}$
Most Negative Integer	100000	$-.1000...X2^{16}$	20000000000	$-.100...X2^{32}$

NOTE
MSB of integer = sign of integer.

Thus, the exponent of a positive floating-point number to be converted must be less than 16_{10} (220 in excess 200 notation) to convert to I format or 32_{10} (240 in excess 200 notation) to convert to L format. The exponent of a negative number to be converted must be less than or equal to 16_{10} or 32_{10} to convert to I or L formats, respectively.

The CPU tests whether the floating-point number to be converted is within the range of integers that can be represented in I or L format by subtracting a constant of 20_8 (for short integers) or 40_8 (for long integers) from the result of the first test (result of first test = biased exponent – 201_8 = unbiased exponent – 1). If the result of the subtraction is positive or zero, it indicates that the floating-point number is too large to be represented as an integer. In that case, a conversion error occurs and 0s are sent to the destination. If the result of the subtraction is a negative number other than –1, the floating-point number can be represented as an integer without causing an overflow condition. If the result of the subtraction is –1, the exponent of the floating-point number is either 220 (short) or 240 (long), and conversion proceeds. However, the floating-point number is within range only if its sign is negative and its fraction is .100 . . . (i.e., if it is the most negative integer; see table above). If, in this case, the number is not the most negative integer, it will be detected by a third conversion error test (see below) after conversion.

To convert the fraction to an integer, the CPU shifts it right a number of places as specified by the following algorithms:

Short integer: No. of right shifts = $20_8 + 201_8$ – biased exponent – 1

Long integer: No. of right shifts = $40_8 + 201_8$ – biased exponent – 1

Regardless of the condition of the FT bit, the fractional part of the number is always truncated during this shifting process.

If the floating-point number is positive, the integer conversion is complete after shifting, and the number is transferred to the appropriate destination. If, however, the floating-point number is negative, the integer must be 2's complemented before being sent to its destination.

After conversion, the CPU performs a third test for a conversion error by comparing the MSB of the (converted) integer with the sign bit of the original (unconverted) number. If these signs are not equal, there has been a conversion error and the CPU traps if the FIC bit is set. This test is performed to detect a floating-point number with an exponent of 220 (short) or 240 (long) that has not been converted to the most negative integer.

Example 1: Store Convert Floating-to-Integer (STCFI)

Exponent = 203
Sign = 0
Fraction (24 bits) = .100000000000000000000000
15 MSBs of fraction = .100000000000000
203 (excess 200) = 2
Fraction = 1/2 Integer to be stored = $1/2 \times 2 = 4$

1. Test 1: Is the number to be converted a fraction?

Exponent:	203_8 -201	
No	2	Since this result is positive, the given floating-point number is not a fraction and conversion may proceed without error.

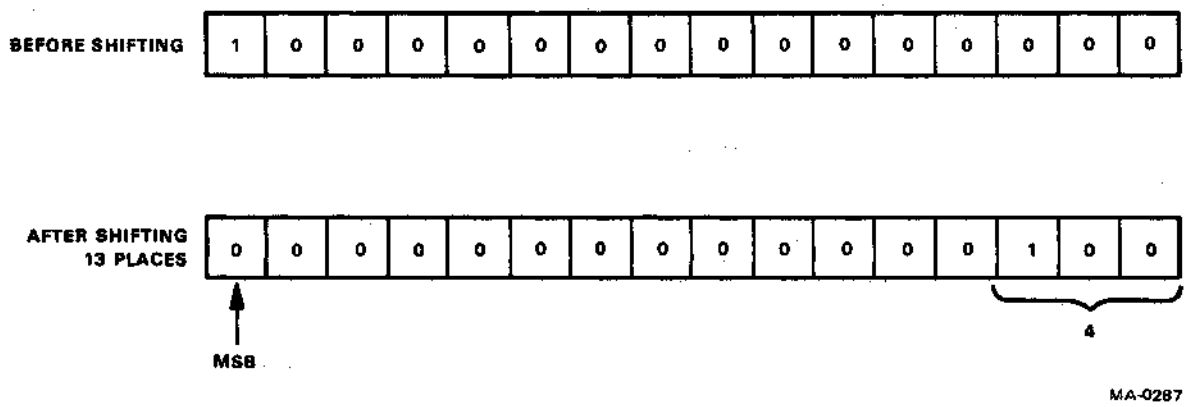
2. Test 2: Is the floating-point number to be converted within range? (We are working with a positive short integer.)

Result of Test 1:	2 −20	
Yes	−16	Indicates that the number to be converted is within range and can be represented as a 16-bit integer. No conversion error occurs.

How many right shifts? Use algorithm:

$$20_8 + 201_8 - 203_8 - 1 = 20_8 - 3_8 = 15_8 = 13_{10}$$

$$= 13 \text{ right shifts}$$



This example involved a positive number, so conversion is complete after 13 right shifts. If the number had been negative, the integer would have been 2's complemented.

3. Test 3: The MSB of the converted integer and the sign bit of the original floating-point number are compared. Since they are equal, no conversion error occurs.

Example 2: Store Convert Floating-to-Integer (STCDL)

Exponent = 240_8
Sign = 0
31 MSBs of fraction = .10000000000000000000000000000000

1. Test 1: Is the number to be converted a fraction?

Exponent:	240_8 -201	
No	37_8	Since this result is positive, the given floating-point number is not a fraction, and conversion may proceed (i.e., no conversion error occurs).

2. Test 2: Is the floating-point number to be converted within range? (We are working with a positive long integer.)

Result of Test 1:	37 -40	
	-1	We know the number is out of range by examining the sign bit (in fact, this number is one greater than the most positive integer that can be represented). However, the CPU does not know this yet, and conversion proceeds without error at this point.

How many right shifts? Use algorithm:

$40_8 + 201_8 - 240_8 - 1 = 0$

= no right shifts

Converted 32-bit integer = 20000000000_8

Since the number is positive, conversion is now complete (i.e., no need for 2's complementing).

3. Test 3: The most significant bit of the converted integer (which is 1) and the sign bit of the original floating-point number (which is 0) are compared. Since they are not equal, a conversion error occurs, which we predicted in Step 2.

4.3.15 Load FP11's Program Status
This instruction causes the CPU to transfer 16 bits from the location specified by the source to the floating-point status (FPS) register. These 16 bits contain status information for use by the FP11-E in order to enable and disable interrupts, set and clear mode bits, and set condition codes (Paragraph 3.4).

4.3.16 Store FP11's Program Status
This instruction causes the CPU to transfer the 16 bits of the FPS register to the specified destination (a floating-point accumulator in mode 0, a memory location otherwise).

4.3.17 Store FP11's Status
The Store FP11's Status (STST) instruction causes the CPU to read the contents of the floating exception code (FEC) and floating exception address (FEA) registers when a floating-point exception (error) occurs.

If mode 0 addressing is enabled, only the FEC is sent to the destination accumulator. If mode 0 addressing is not enabled, the FEC is stored in memory followed by the FEA. In memory, the FEC data occupies all 16 bits of its memory location, while the FEA data occupies only the lower four bits of its location.

When an error occurs and the interrupt trap in the CPU is enabled, the CPU traps to interrupt vector 244 and issues the STST instruction to determine the type of error.

NOTE

The STST instruction should be used only after an error has occurred, since in all other cases the instruction contains irrelevant data or contains the conditions that occurred after the last error.

4.3.18 Copy Floating Condition Codes
The Copy Floating Condition Codes (CFCC) instruction causes the CPU to copy the four floating condition codes (FC, FZ, FV, FN) into the CPU condition codes (C, Z, V, N).

4.3.19 Set Floating Mode
The Set Floating Mode (SETF) instruction causes the CPU to clear the FD bit (bit 07 of the FPS register) and indicate single-precision operation.

4.3.20 Set Double Mode
The Set Double Mode (SETD) instruction causes the CPU to set the FD bit (bit 07 of the FPS register) and indicate double-precision operation.

4.3.21 Set Integer Mode
The Set Integer Mode (SETI) instruction causes the CPU to clear the IL bit (bit 06 of the FPS) and indicate that short integer mode (16 bits) is specified.

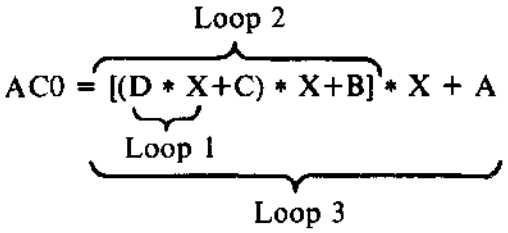
4.3.22 Set Long Integer Mode
The Set Long Integer Mode (SETL) instruction causes the CPU to set the IL bit (bit 06 of the FPS) and indicate that long integer mode (32 bits) is specified.

4.4 FP11-E PROGRAMMING EXAMPLES
This section contains two programming examples using the FP11-E instruction set. In example 1, A is added to B, D is subtracted from C, the quantity (A + B) is multiplied by (C – D), and the product of this multiplication is divided by X and the result stored. Example 2 calculates $DX^3 + CX^2 + BX + A$, which involves a 3-pass loop.

Example 1 $[(A + B) * (C - D)] * X$

```
LDF     A,AC0     ;LOAD AC0 FROM A
ADDF    B,AC0     ;AC0 HAS (A + B)
LDF     C,AC1     ;LOAD AC1 FROM C
SUBF    D,AC1     ;AC1 HAS (C - D)
MULF    AC1,AC0   ;AC0 HAS (A + D)*(C - D)
DIVF    X,AC0     ;AC0 HAS (A + D)*(C - D)/X
STF     AC0,Y     ;STORE (A + D)*(C - D)/X IN Y
```

Example 2: $DX^3 + CX^2 + BX + A$



$AC0 = [DX^2 + CX + B] * X + A$

$AC0 = DX^3 + CX^2 + BX + A$

```
        MOV #3,%0       ;SET UP LOOP COUNTER
        MOV #D+4,%1     ;SET UP POINTER TO COEFFICIENTS
        LDF (6)+,AC1    ;POP X FROM STACK
        CLRF AC0        ;CLEAR OUT AC0
LOOP;   ADDF -(4),AC0   ;ADD NEXT COEFFICIENT
                        ;TO PARTIAL RESULT
        MULF AC1,AC0    ;MULTIPLY PARTIAL RESULT BY X
        SOB %0,LOOP     ;DO LOOP 3 TIMES
        ADDF -(4),AC0   ;ADD X TO GET RESULT
        STF AC0,-(6)    ;PUSH RESULT ON STACK
```

CHAPTER 5
FP11-E/CPU RELATIONSHIP

5.1 INTRODUCTION

As shown in Figure 5-1, the CPU and FP11-E communicate with each other via data lines and control lines. Since the FP11-E connects directly to the PDP-11/60 CPU and not to the Unibus, the FP11-E uses the CPU's Unibus control facility (for Unibus data transfers) and memory management facilities.

The FP11-E depends on the CPU to fetch instructions and data via the CPU-to-FP11-E data lines. The FP11-E can also supply data to the CPU via the FP11-E-to-CPU data lines.

The CPU controls the operation of the FP11-E via the CPU-to-FP11-E control lines and monitors the operation of the FP11-E via the FP11-E-to-CPU control lines. The control lines are also used to ensure that the CPU and FP11-E are synchronized with each other at the proper time when floating-point instructions are being performed.

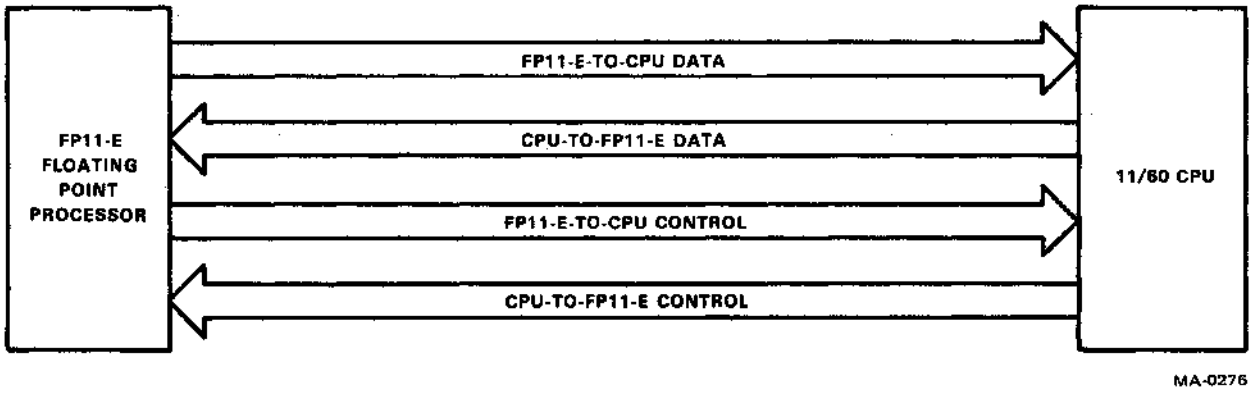


Figure 5-1 FP11-E/CPU Relationship

5.2 FP11-E/CPU SIGNALS

The signals via which the FP11-E and the CPU communicate are shown in Figure 5-2 and described in Table 5-1.

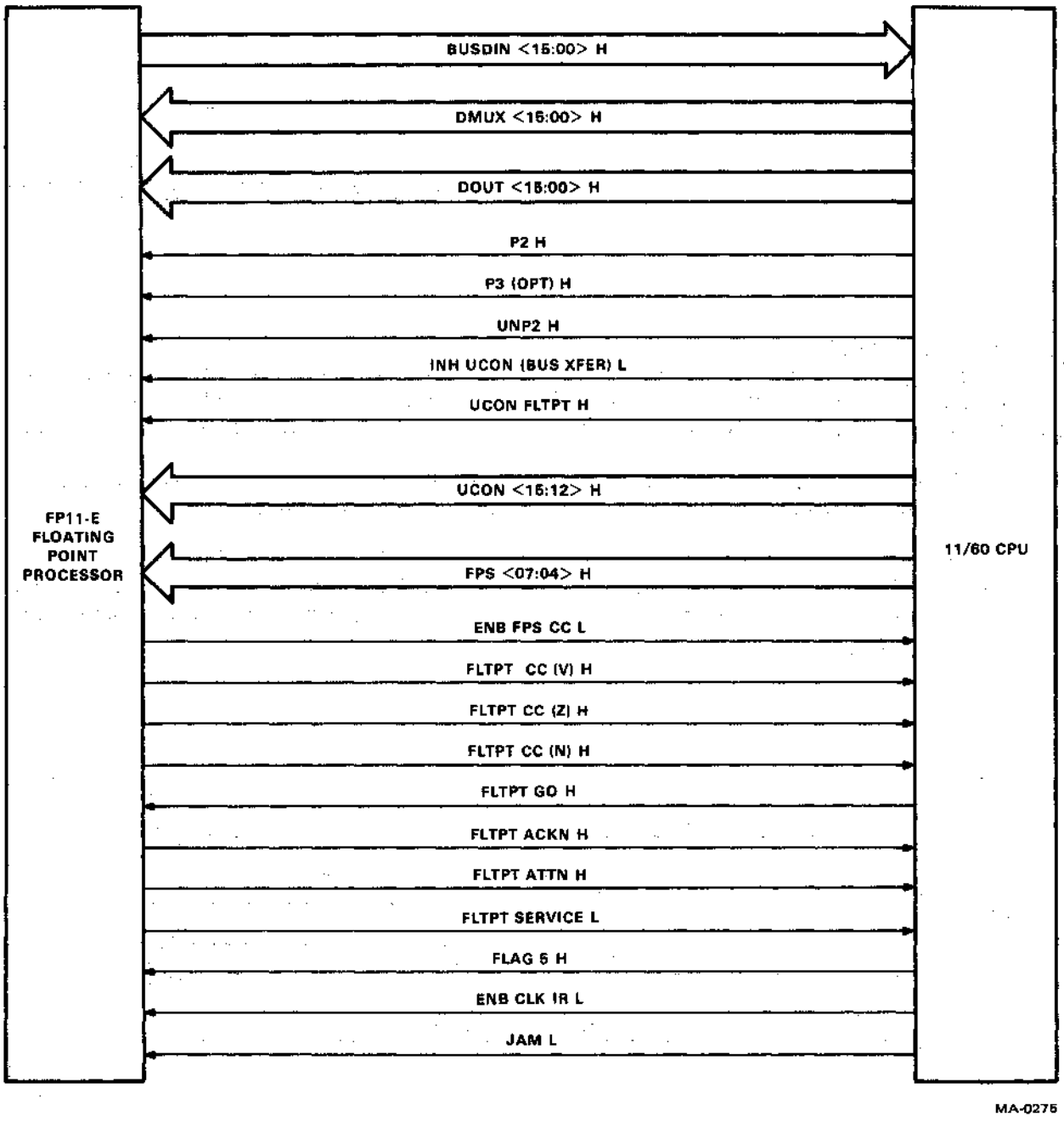


Figure 5-2 FP11-E/CPU Signals

Table 5-1 FP11-E/CPU Signals

Signal Name	Direction	Description
Data Signals		
BUSDIN (15:00) H	FP11-E to CPU	Sixteen lines that the FP11-E uses to send data to the CPU.
DMUX (15:00) H	CPU to FP11-E	Sixteen lines over which the CPU sends instructions and operands to the FP11-E.
DOUT (15:00) H	CPU to FP11-E	Sixteen lines that the CPU uses to send data to the FP11-E.
Clocks		
P2 H P3 (OPT) H UNP2 H	CPU to FP11-E CPU to FP11-E CPU to FP11-E	P2 and P3 (OPT) are suppressed clocks, while UNP2 is an unsuppressed clock. A suppressed clock can be "frozen" by the CPU at certain times.
Microcontrol Signals		
INH UCON (BUS XFER) L	CPU to FP11-E	Disables FP11-E input to BUSDIN. Allows data from memory to enter CPU via BUSDIN.
UCON FLTPT H	CPU to FP11-E	Selects the FP11-E for the purpose of transferring data or control. It causes the FP11-E output MUX (through which all FP11-E output data is routed) to route data to BUSDIN if INH UCON is not asserted; allows a service request to be granted in conjunction with UCON 12; can abort execution of a floating-point instruction in conjunction with UCON 13; allows the CPU to determine whether or not the FP11-E is installed in conjunction with UCON 14, if INH UCON is not asserted; and initializes the FP11-E in conjunction with UCON 15.
UCON 12 H	CPU to FP11-E	Issued in response to an FP11-E service request. In order for the request to be granted, UCON FLTPT must also be asserted at this time.
UCON 13 H	CPU to FP11-E	Issued by the CPU if it terminates a floating-point instruction to honor a service request before the CPU has completed its interfacing operation with that of the FP11-E. Executed if UCON FLTPT is asserted.

Table 5-1 FP11-E/CPU Signals

Signal Name	Direction	Description
Microcontrol Signals (Cont)		
UCON 14 H	CPU to FP11-E	Asserted by the CPU during its power-up sequence. If UCON FLTPT is asserted and INH UCON is not asserted at this time, the FP11-E will assert BUSDIN (14). This informs the CPU that the FP11-E is installed.
UCON 15 H	CPU to FP11-E	Initializes the FP11-E if UCON FLTPT is asserted.
Status Signals		
FPS (07:04) H	CPU to FP11-E	FP11-E status bits from the FPS register located in the CPU. Refer to Paragraph 3.4 for a description of the significance of these bits.
ENB FPS CC L	FP11-E to CPU	Enable which allows condition codes from the FP11-E to be clocked into the FPS register.
Condition Codes from FP11-E		
FLTPT CC(V) H	FP11-E to CPU	Indicates that the result of the last FP11-E operation resulted in an arithmetic overflow.
FLTPT CC(Z) H	FP11-E to CPU	Indicates that the result of the FP11-E operation was zero.
FLTPT CC(N) H	FP11-E to CPU	Indicates that the result of the last FP11-E operation was negative.
Synchronization		
FLTPT GO H	CPU to FP11-E	Initiates synchronization of the operation of the FP11-E with that of the CPU.
FLTPT ACKN H	FP11-E to CPU	FP11-E's response to FLTPT GO. Issued when the FP11-E and the CPU are in synchronization.
FLTPT ATT H	FP11-E to CPU	FP11-E ready to proceed. At certain times, the CPU will not perform further operations until it has received this signal.

Table 5-1 FP11-E/CPU Signals

Signal Name	Direction	Description
Other Signals		
FLTPT SERVICE L	FP11-E to CPU	FP11-E's request for service.
FLAG 5 H	CPU to FP11-E	Issued by the CPU to enable the FP11-E when the FP11-E is present in the system.
ENB CLK IR L	CPU to FP11-E	Allows an instruction to be loaded into the floating-point instruction register (FIRA) by the unsuppressed clock signal UNP2.
JAM L	CPU to FP11-E	Can abort the current FP11-E instruction when certain error conditions (i.e., illegal address, stack overflow, MM abort, parity error, microbreak) are detected.

5.3 FP11-E/CPU INTERACTION

Figure 5-3 illustrates the relationship between the FP11-E and the CPU in more detail.

The CPU fetches an instruction from main memory (via BUSDIN and DMUX) or from cache memory (via DMUX) and loads it into both its own instruction register (IR) and an instruction register in the FP11-E (FIRA).

The CPU then decodes the instruction. If the four high-order bits are set, the instruction has an op code of 17xxxx and is a floating-point instruction. The CPU then executes microinstructions associated with the operation of the FP11-E.

NOTE

The CPU verifies the presence of the FP11-E by monitoring the signal FLAG 5. If FLAG 5 is asserted, the CPU assumes that the FP11-E is present in the system and executes microinstructions associated with the operation of the FP11-E. If FLAG 5 is not asserted, the CPU executes microinstructions associated with the integral floating-point instruction set contained in its control store.

The FP11-E also decodes the instruction. At the beginning of floating-point instruction execution, the contents of FIRA are loaded into FIRB (by the FP11-E). The contents of FIRB are then decoded to determine the type of floating-point instruction to be executed.

NOTE

The FIRB contains the instruction that the FP11-E is currently executing, while the FIRA contains the instruction most recently fetched by the CPU.

If the instruction to be executed is a Load class instruction, the FP11-E goes into an idle state while the CPU fetches the required operands. After calculating the addresses of the operands, the CPU loads (or loads and converts) the operands and transfers them to the FP11-E. Once the operands have been received by the FP11-E (via DOUT) and loaded via INBUFA and/or INBUFB, the FP11-E can complete execution of the instruction. Meanwhile, the CPU fetches the next instruction and loads it into the IR and FIRA. If this instruction is a floating-point instruction, it cannot be executed until the FP11-E is finished with its current operation. If this instruction is not a floating-point instruction, the CPU executes it immediately.

If the instruction to be executed is a Store class instruction, the FP11-E decodes the instruction and begins executing it. The CPU calculates the addresses of the operands and waits until the FP11-E is completed with its instruction execution. The FP11-E then passes its result to the CPU via BUSDIN which stores (or converts and stores) this result. After this, the CPU is free to fetch another instruction.

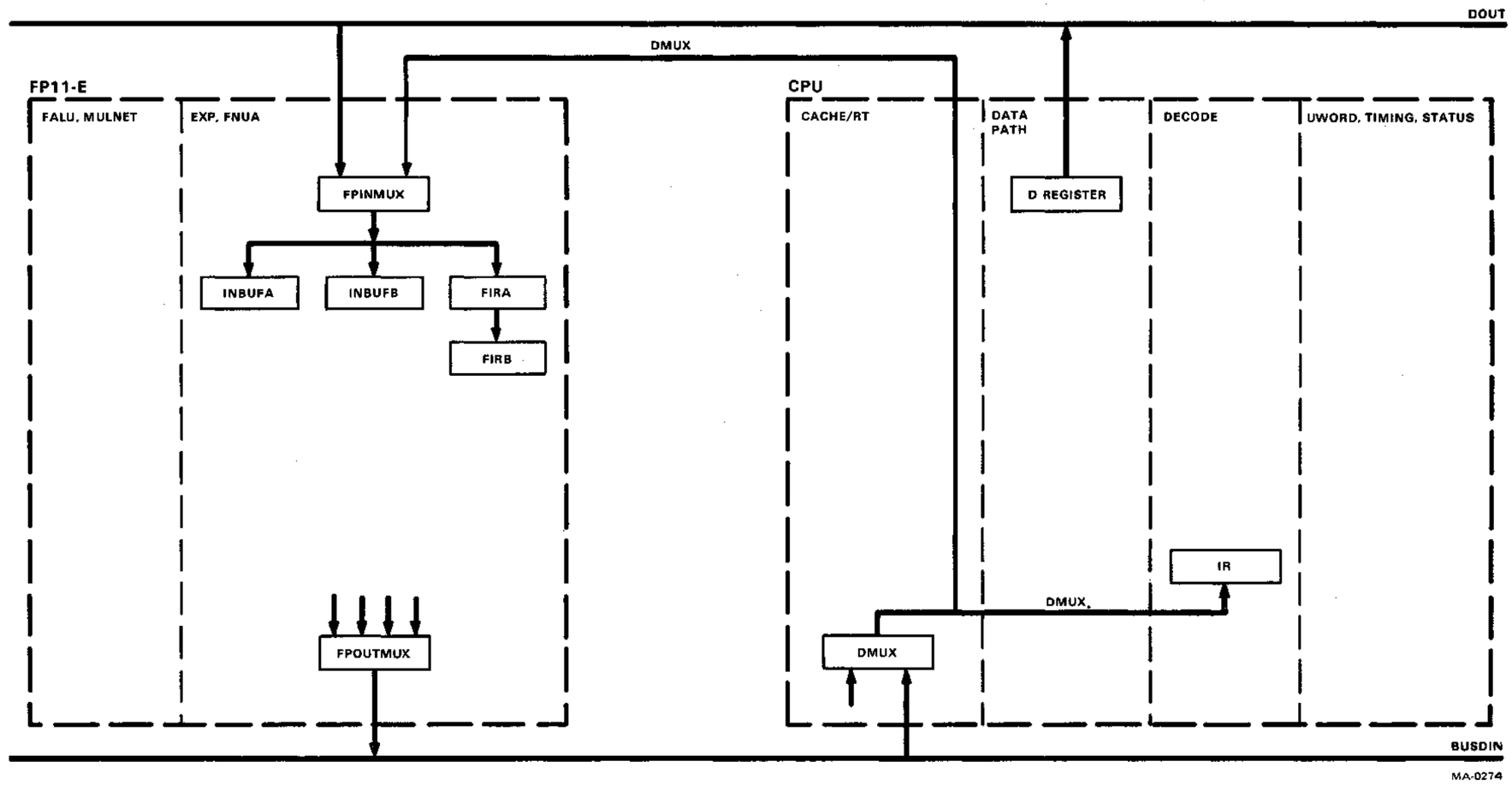


Figure 5-3 FP11-E/CPU Interconnection

CHAPTER 6
FP11-E DATA MANIPULATION COMPONENTS

6.1 INTRODUCTION

With the aid of Figure 6-1, this chapter briefly describes the functions and organization of the components that manipulate data in the FP11-E. The block diagram is partitioned into four sections (FNUA, FALU, EXPONENT, MULNET), each of which represents an FP11-E circuit board. The following descriptions of the FP11-E data manipulation components are also grouped in this way.

6.2 FNUA BOARD

The main purpose of the FNUA circuit board is to determine the (micro) address of the next microinstruction to be executed. This is necessary because execution of FP11-E floating-point instructions involves addressing sequences of microinstructions. However, the FNUA board also contains components that receive and manipulate instructions and data. These components are the FPINMUX, FIRA, FIRB, INBUFA, and INBUFB.

6.2.1 FPINMUX

All input data to the FP11-E is routed through the FPINMUX. The FPINMUX selects either the DMUX lines (which carry instructions and operands) or the DOUT lines (which carry status information and other data from the CPU) for input into the FP11-E.

6.2.2 FIRA and FIRB

All instructions which are loaded into the CPU's instruction register (IR) are also loaded into the FP11-E's FIRA. After an instruction has been loaded into FIRA, the FP11-E microcode will load it into FIRB only if the FP11-E has completed executing another instruction and the CPU has not decided to abort the instruction. If the FP11-E has completed executing another instruction, the loading of FIRB is inhibited until the FP11-E is done. If the CPU aborts the instruction, it must be fetched again and reloaded into the FIRA.

Once FIRB is loaded, the FP11-E determines whether or not FIRB contains a floating-point instruction. If it does, a sequence of microinstructions is executed; the sequence depends on the type of floating-point instruction in FIRB.

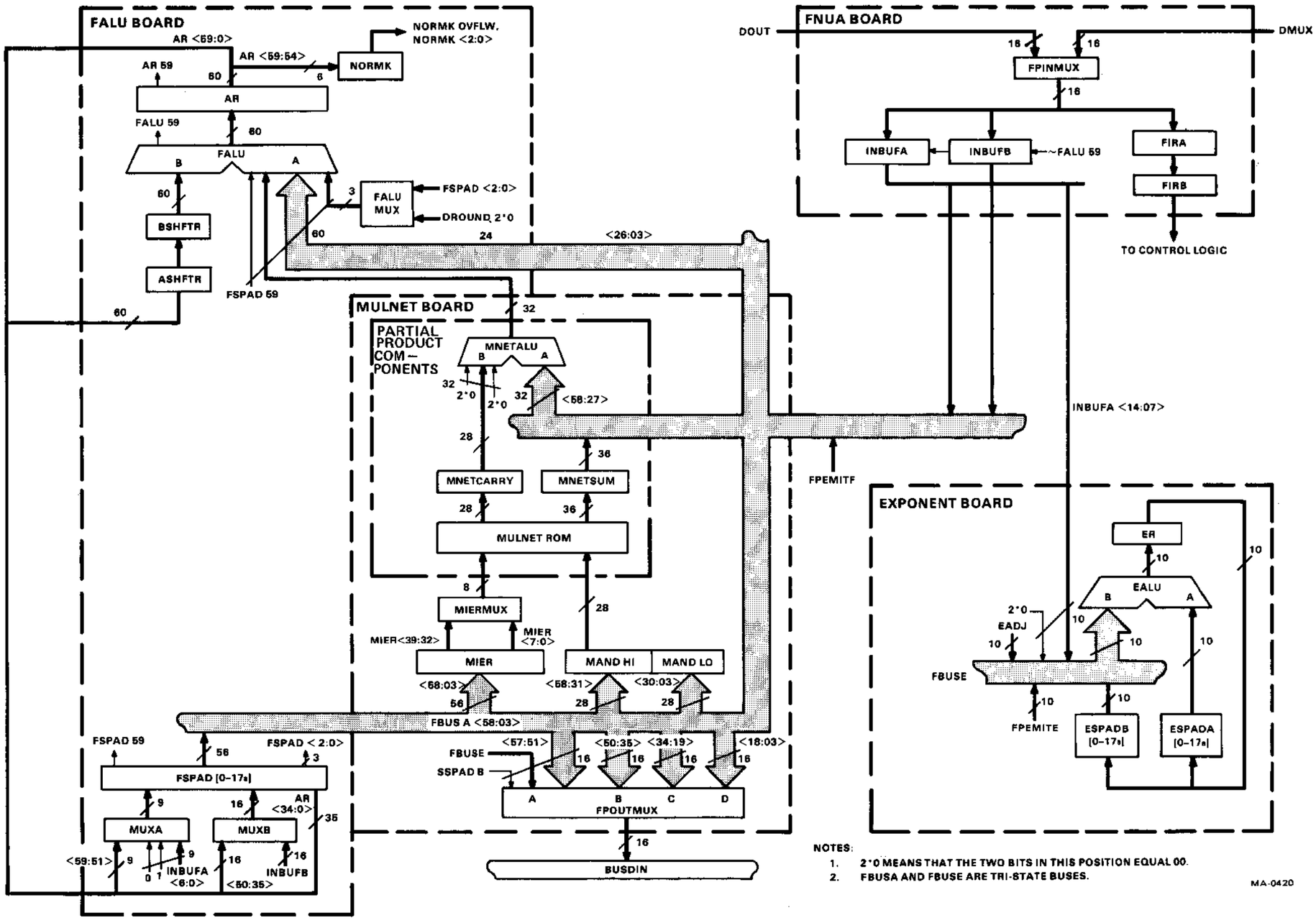


Figure 6-1 FP11-E Data Manipulation Components

6.2.3 INBUFA and INBUFB

INBUFA and INBUFB are input buffers for operand shift registers entering the FP11-E in D, F, L, or I format and also serve as shift registers for the accumulation of the quotient during a Divide instruction. INBUFB can also be loaded with the high byte of the floating-point status (FPS) word from the CPU.

If the operand to be loaded into the FP11-E is in D format, INBUFA and INBUFB are affected as follows. The first word (word A), which contains the sign bit, the exponent, and the high-order seven bits of the fraction, is fetched and loaded into INBUFA. Meanwhile, the second word (word B), which contains the next 16 most significant bits of the fraction, is being fetched for loading into INBUFB. At the time INBUFB is loaded, the sign and exponent in INBUFA are sent to appropriate parts of the FP11-E [i.e., the sign is clocked into the SD flip-flop and the exponent is clocked into the exponent register (ER)]. The upper 23 bits of the fraction, now contained in INBUFA and INBUFB, are then clocked into bits 57–35 of an FP11-E data word register contained in the fraction scratchpad (FSPAD). The next word (word C), which contains the next 16 bits of the fraction, is fetched and loaded into INBUFA, followed by the fetching and loading of the final word (word D), which contains the least significant 16 bits of the fraction, into INBUFB. The lower 32 bits of the fraction, now contained in INBUFA and INBUFB, are then clocked into bits 34–03 of the register in FSPAD (via the FALU data path). Loading of the double-precision operand into the FP11-E is then complete.

An operand in F format (two words) is loaded into the FP11-E in exactly the same way that the first two words of an operand in D format are loaded.

If the operand to be loaded into the FP11-E is in L format, the first word, which contains the high-order 16 bits of the integer, is loaded into INBUFA. Then the second word, which contains the low-order 16 bits of the integer, is loaded into INBUFB. The 32-bit operand is then loaded into bits 34–03 of a register in FSPAD (via the MULNET and FALU data paths).

An operand in I format is loaded into INBUFA and sent to bits 34–19 of a register in FSPAD (via the MULNET and FALU data paths).

During a Divide instruction, the quotient is serially left-shifted into INBUFA and INBUFB. After the left-shifting is completed, the quotient is sent to a floating-point accumulator.

As mentioned before, INBUFB can also be loaded with the high byte of the floating-point status (FPS) word. The FPS word is right-justified in INBUFB, with 0s occupying the upper byte of INBUFB.

6.3 FALU BOARD

The FALU components FSPAD, MUXA and MUXB, ASHFTR and BSHFTR, FALU, FALUMUX, AR, and NORMK are shown in Figure 6-1. The main purpose of these components is to process floating-point fractions, integers, and MULNET partial products during floating-point instructions. Operands and constants are input to the FALU board (and to the MULNET board) via a 56-bit tristate data bus internal to the FP11-E called FBUSA. FBUSA also carries data out of the FALU board to the FPOUTMUX on the MULNET board.

This section briefly describes the functions of the FALU components and describes the sources and destinations of FBUSA.

6.3.1 FSPAD

FSPAD is a scratchpad consisting of sixteen 60-bit registers. The FSPAD receives data from AR (34:00) and from the outputs of MUXA and MUXB and produces a 60-bit output, FSPAD (59:00).

Each 60-bit register in FSPAD is partitioned into six fields: bit 59 = overflow bit field; bits 58–51 = A field; bits 50–35 = B field; bits 34–19 = C field; bits 18–03 = D field; bits 03–00 = guard bit field.

The 16 registers are numbered 0–17_8. Only registers 0–5 are accessible to the user. These six registers can contain floating-point fractions or integers.

NOTE

The six floating-point accumulators referred to in Paragraph 4.1 consist of: twelve (effectively six) 1-bit sign registers contained in the sign scratchpads SSPADA and SSPADB (numbered 0–5 in both scratchpads); twelve (effectively six) 8-bit exponent registers contained in the exponent scratchpads ESPADA and ESPADB (numbered 0–5 in both scratchpads); and the six fraction registers contained in FSPAD (numbered 0–5) just mentioned. For example, floating-point accumulator AC2 has its sign bit in SSPADA 2 and SSPADB 2, its exponent in ESPADA 2 and ESPADB[2], and its fraction in FSPAD 2. The other registers in FSPAD are related to the sign and exponent scratchpads in the same way.

FSPAD register 6 contains the fraction or integer that has been most recently fetched. Register 7 contains only the FEC (Paragraph 3.5) in its B field. Registers 10_8–16_8 are simply general-purpose registers for the FP11-E's internal use during floating-point operations. Registers 16_8 and 17_8 contain all zeros.

The various fields of FSPAD can be read and written as follows.

Read	**Write**
1. A and B or	1. A and B or
2. C and D or	2. C and D or
3. A, B, C, and D or	3. A, B, C, and D
4. C(L) and D where C(L) = the lower 4 bits of C	

For example, numbers 1 or 3 above are selected when a single- or double-precision floating-point word (respectively) is read or written. Number 2 is selected when an integer is read or written. Number 2 or 4 is selected for reading when FSPAD[17] must be driven onto FBUSA (Paragraph 6.3.8).

6.3.2 MUXA and MUXB

MUXA and MUXB are two 2:1 multiplexers that input either INBUFA and INBUFB (respectively) or AR (59:35) to the upper 25 bits (i.e., the overflow bit, A, and B fields) of a selected register in FSPAD.

MUXA and MUXB select INBUFA and INBUFB for input to FSPAD when a single- or double-precision floating-point operand enters the FP11-E (Paragraph 6.2.3). When INBUFA is selected as an input to MUXA, the overflow bit field is set to zero, the hidden bit is inserted in the most significant bit of the A field, and the upper seven bits of the floating-point operand are routed to the remainder of the A field. Simultaneously, INBUFB is selected as an input to MUXB and is routed to the B field.

At all other times, MUXA and MUXB select AR (59:35) for input to FSPAD.

6.3.3 ASHFTR and BSHFTR

The ASHFTR and BSHFTR are two series-connected 4:1 multiplexers that, as a unit, behave like a shifter. In one pass, the ASHFTR/BSHFTR combination can perform as many as 4 left shifts or as many as 11 right shifts on the contents of AR (59:00). The shifted version of the AR is then sent to the "B" input of the FALU.

NOTE

It is emphasized that ASHFTR and BSHFTR are considered as a unit. ASHFTR alone does not right- or left-shift AR (59:00). That is, the relationship between the input and output of the ASHFTR is not one of a shifter. The same is true for BSHFTR. If considered collectively, however, it is seen that shifting does occur between the ASHFTR input and the BSHFTR output.

The most common applications of the ASHFTR and BSHFTR are shifting the contents of the AR left, during normalization; right, during fraction alignment (as in floating-point addition or subtraction); and right, for correct alignment of partial products during floating-point multiplications.

6.3.4 FALU

The FALU performs arithmetic and logical operations on its two 60-bit inputs and produces a 60-bit result for loading into the AR. One input to the FALU (the "B" input) is the output of the ASHFTR/BSHFTR combination, BSHFTR (59:00). FALU's other input (the "A" input) consists of FSPAD 59 (the overflow bit), MNETALU (58:27), FBUSA (26:03), and FALU MUX (02:00).

The FALU can also pass data unaltered from one input to its output.

6.3.5 AR

The AR is a 60-bit register that, when clocked, temporarily holds the output of the FALU. The output of the AR can be shifted by the ASHFTR and BSHFTR or it can be clocked into a register in FSPAD. The six most significant bits of the AR drive the NORMK component.

6.3.6 NORMK

NORMK is an encoder used during fraction normalization. It has a 6-bit input [AR (59:54)] and a 4-bit output [NORMK OVFLW, NORMK (02:00)].

During fraction normalization, the contents of the AR may require left or right shifting. As mentioned previously, this function is performed by the ASHFTR/BSHFTR combination. NORMK produces a code which causes ASHFTR/BSHFTR to shift the AR either one place right or as many as four places left at a time. If more than four left shifts must be performed, NORMK OVFLW is asserted, four leftshifts are performed, and the current microstate is repeated until the number of remaining left shifts is less than four.

The EXPONENT components also rely on information from NORMK. NORMK provides its code to EADJ, which then produces a constant that is input to the EALU. This constant is added to or subtracted from the exponent associated with the unnormalized fraction (Paragraph 6.4.4).

6.3.7 FALU MUX

The FALU MUX is a 2:1 multiplexer which routes one of its 3-bit inputs [FSPAD (02:00) or DROUND, 0, 0] to its 3-bit output [FALU MUX (02:00)] for use by the FALU.

The FALU MUX usually routes the value of DROUND and two 0s to the FALU. If a double-precision fraction is to be rounded, DROUND = 1 and thus FALU MUX (02:00) = 100. The fraction to be rounded is first normalized in the FALU data path. Rounding is then accomplished by adding FALU's A input (which contains 57 zeros in bits 59–03 and 100 in bits 02–00) with its B input (which contains the normalized fraction). If the normalized fraction contains a 1 in bit 02, adding DROUND will produce a carry-out and will increment bit 03. If the normalized fraction contains a 0 in bit 02, adding DROUND will have no effect on bit 03. If the normalized fraction is all 1s, adding DROUND will cause a fraction overflow. In this one special case, the fraction must be normalized a second time. If DROUND = 0, FALU MUX (02:00) = 000, and no rounding occurs. Bits 59–00 of the normalized double-precision fraction are unchanged and, in fact, bits 02–00 are later effectively truncated.

6.3.8 FBUSA – Sources and Destinations

As mentioned before, FBUSA is a 56-bit tristate internal data bus. FBUSA carries data from one of four sources (FSPAD, FPEMITF, MNETSUM, INBUF) to one or more of five destinations (MAND, MIER, FALU, FPOUTMUX, MNETALU).

Figure 6-2 and Table 6-1 define which parts of FBUSA the various sources drive. Figure 6-3 and Table 6-2 define the parts of FBUSA from which the various destinations receive data.

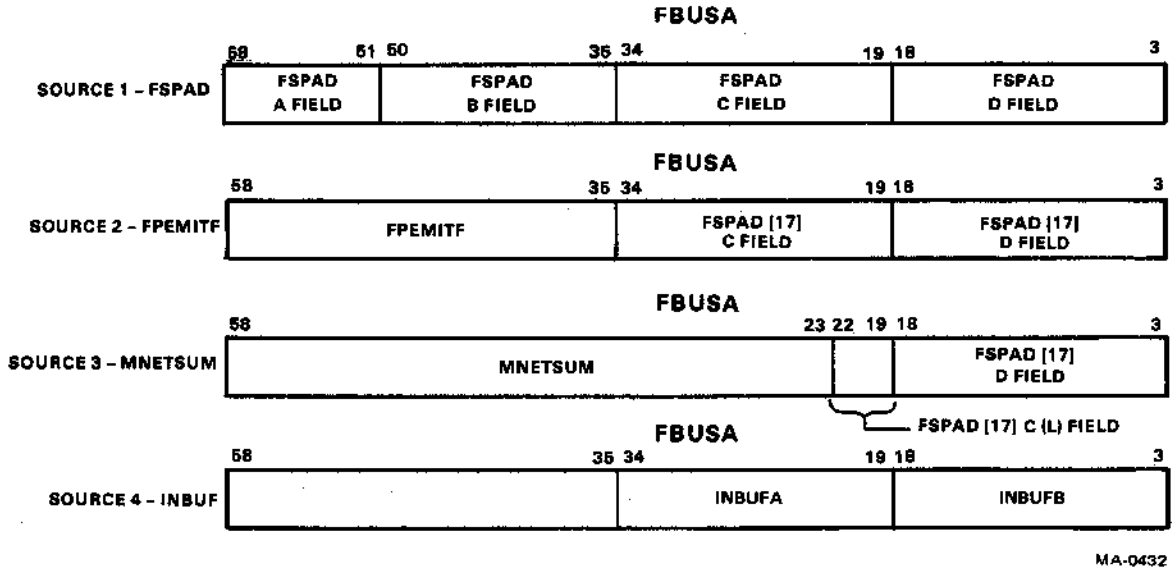


Figure 6-2 Sources of FBUSA

Table 6-1 Sources of FBUSA

Source	Description
FSPAD	All or part of a register in FSPAD can be read from FSPAD (Paragraph 6.3.1) and driven onto FBUSA. If a single-precision fraction is read from FSPAD, the C and D fields of FSPAD 17 are also read to ensure that the 32 lower bits of FBUSA are zeros.
FPEMITF	FPEMITF is a 24-bit source that generates all constants required by the fraction data path. When it is selected for reading onto FBUSA, it drives bits 58–35, while the remainder of FBUSA is driven with zeros from FSPAD 17.
MNETSUM	MNETSUM (Paragraph 6.5.3.1) generates a 36-bit number which, when read onto FBUSA, drives bits 58–23. The remainder of FBUSA is driven with zeros from FSPAD 17.
INBUF	INBUFA and INBUFB, when read, drive bits 34–03 of FBUSA, while FBUSA bits 58–35 are left floating and are not used.

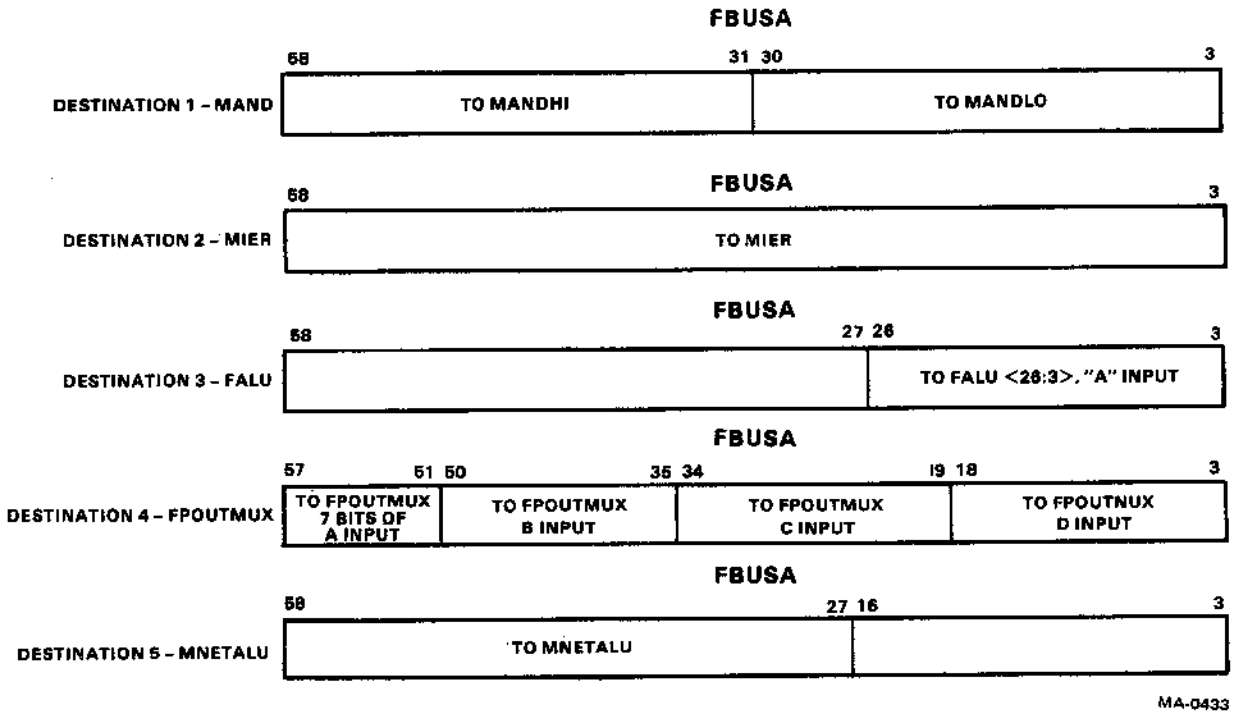


Figure 6-3 Destinations of FBUSA

Table 6-2 Destinations of FBUSA

Destination	Description
MAND and MIER	The MAND and MIER are multiplicand and multiplier registers (respectively) used during multiplication instructions. When these registers load data from FBUSA, FBUSA is being driven by either the FPEMITF or FSPAD sources.
FALU	When FALU receives data from FBUSA (26:03), FBUSA (26:03) are being driven by the lower 24 bits of the MNETSUM source, bits 26–03 of the INBUF source, the lower 24 bits of the FSPAD source, or the lower 24 bits of the FPEMITF source.
MNETALU	When the MNETALU (Paragraph 6.5.3.2) receives data from FBUSA (58:27), FBUSA (58:27) are being driven by the upper 32 bits of the MNETSUM source, bits 34–27 of the INBUF source, the upper 32 bits of the FSPAD source, or the upper 32 bits of the FPEMITF source. **NOTE** **The FALU-MNETALU combination can receive any of the four sources of FBUSA in its entirety.**
FPOUTMUX	The FPOUTMUX routes data from FBUSA to BUSDIN, 16 bits at a time. It performs this function when either the FSPAD source or FPEMITF (50:35) are driving FBUSA.

6.4 EXPONENT BOARD

The main purpose of the EXPONENT components shown in Figure 6-1 (i.e., the EALU, the ER register, and the ESPADA and ESPADB scratchpads) is to process the exponents of floating-point numbers during floating-point instructions. These exponents and other data are input to the EXPONENT board via a 10-bit tristate data bus internal to the FP11-E called FBUSE. (FBUSE also carries exponents out of the EXPONENT board to the FPOUTMUX on the MULNET board.)

NOTE

Although the EXPONENT components accommodate 10-bit data words and are interconnected with 10-bit data lines, only the lower eight bits of EXPONENT's data path are used by the exponents themselves. The upper two bits of EXPONENT's data path occasionally provide information for the FP11-E control logic relating to overflow and underflow as the result of an operation performed by the EALU on two exponents. A complete description of these two upper bits is found in the FP11-E Floating-Point Processor Technical Manual. For the purposes of this manual, however, it is convenient to think of the EXPONENT data path as being 8 bits wide and to think of the EXPONENT components as accommodating 8-bit exponents.

This section briefly describes the functions of the EXPONENT components and describes the sources of FBUSE, from which EXPONENT receives input.

6.4.1 EALU

The EALU performs arithmetic and logical operations on its two inputs, FBUSE and ESPADA, and produces a result for loading into the ER. For example, during a floating-point multiplication, EALU adds two exponents. In this case, one exponent would be retrieved from ESPADA and one exponent from ESPADB (a source of FBUSE).

The EALU can also pass data from one input directly to its output, for eventual storage in the exponent scratchpads.

6.4.2 ER

The ER is a register that, when clocked, temporarily holds the output of the EALU. The output of the ER is subsequently loaded into ESPADA and ESPADB.

6.4.3 ESPADA and ESPADB

ESPADA and ESPADB are the exponent scratchpads. They are loaded with data from the ER. ESPADA feeds one input to the EALU, while ESPADB feeds the other input to the EALU, via FBUSE.

Each scratchpad contains 16 registers, numbered $0–17_8$. If the register specified for writing is in the range 0–7, both ESPADA and ESPADB are loaded simultaneously. If the register specified for writing is in the range $10_8–13_8$. ESPADA and ESPADB are loaded either simultaneously or individually. For example, if register 3 is specified, both ESPADA 3 and ESPADB 3 are loaded with the contents of the ER. If register 10_8 is specified, however, ESPADA 10 or ESPADB 10 or both ESPADA 10 and ESPADB 10 can be loaded with the contents of the ER.

Both ESPADA and ESPADB registers $14_8–17_8$ contain the constants 0, 1, 200, and 201, respectively. Also, ESPADB 12 and ESPADB 13 contain 231 and 271, respectively. These constants are loaded into the scratchpads once, during the FP11-E power-up sequence, from the FPEMITE input (Paragraph 6.4.4).

6.4.4 FBUSE and Its Sources

As mentioned before, FBUSE is a 10-bit (effectively 8-bit) tristate internal data bus that carries data from one of four sources (INBUFA, EADJ, FPEMITE, or ESPADB) for use by the EALU (or alternatively by the FPOUTMUX, if the source is ESPADB). Only one source at a time can be enabled onto FBUSE.

INBUFA (14:07) is used as a source when the first word of a floating-point operand is loaded into the FP11-E. [INBUF (14:07) contains the exponent of the operand.] After passing through the EALU, this exponent is loaded into ESPADA and ESPADB via the ER.

EADJ is used as a source when it is necessary to increase or decrease an exponent as a result of a fraction normalization that has occurred in the FALU data path. EADJ specifies a constant, generated on the basis of data from the FALU data path, which indicates the number of times a fraction in the AR has been shifted left or right to be normalized. This constant is applied to one input of the EALU and is added or subtracted from the exponent (retrieved from ESPADA) associated with the normalized fraction.

FPEMITE is used as a source when a constant must be loaded into the exponent scratchpads. FPEMITE specifies a constant which is passed through the EALU and is loaded into ESPADA and/or ESPADB, via the ER.

ESPADB is used as a source when an operation must be performed on an exponent contained in ESPADB and an exponent contained in ESPADA. The EALU performs the operation and the result is loaded into ESPADA and ESPADB via the ER. ESPADB is also used as a source of FBUSE when an exponent must be retrieved from the EXPONENT board. In that case, the FPOUTMUX receives the exponent rather than the EALU.

6.5 MULNET BOARD

The MULNET components (Figure 6-1) multiply the fractions of two single- or double-precision floating-point numbers. Recall that both fractions are either 24 bits (single-precision) or 56 bits (double-precision).

NOTE

Fraction multiplication requires the use of both the MULNET and FALU data paths.

6.5.1 Single-Precision Multiplication

To multiply two single-precision fractions, the FP11-E performs the steps outlined in Figure 6-4. At the time this fraction calculation is occurring, the two exponents of the floating-point numbers are being added in the EXPONENT data path and the sign of the result is being determined by the control logic. The fraction obtained by step 3 must be normalized. If the fraction is shifted to the left, the exponent must be decreased accordingly.

To see how steps 1, 2, and 3 in Figure 6-4 produce the correct result, refer to Figure 6-5. To see how this is implemented in the FP11-E, refer to Figure 6-1.

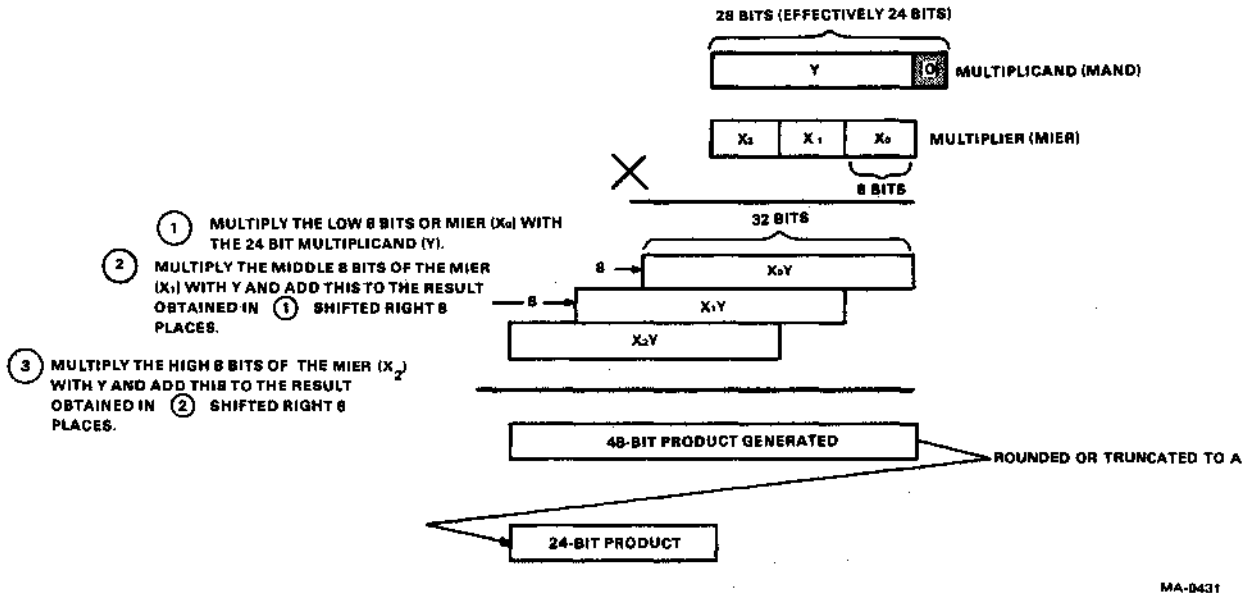


Figure 6-4 FP11-E Single-Precision Multiplication

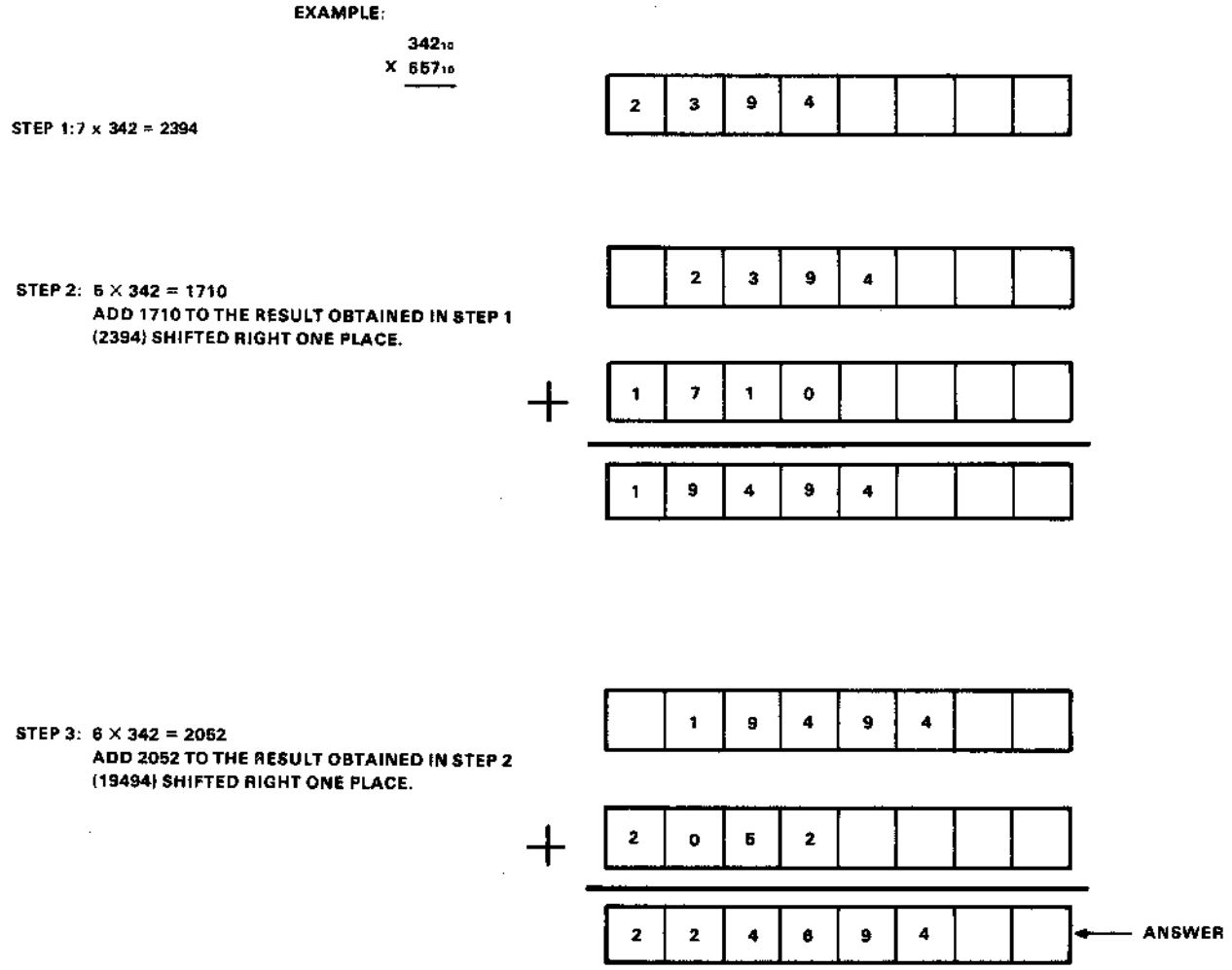


Figure 6-5 Example of Single-Precision Multiplication

The upper half of a 56-bit register called MAND (MANDHI) is loaded with the multiplicand. For a single-precision multiplication, the 24-bit multiplicand fraction is loaded into MAND (55:32), while 0s are loaded into MAND (31:28). The upper half of MAND is then swapped with the lower half. Thus, after the swap, the fraction is contained in MAND (27:04), while 0s occupy MAND (03:00).

NOTE

In the block diagram, the components labeled MULNET, MNETCARRY, MNETSUM, and MNETALU are all contained in a 2-input, 1-output box labeled "partial product components." More details about these components are given in Paragraph 6.5.3. In this case, they simply multiply an 8-bit portion of MIER (see below) with the 28-bit (effectively 24-bit) MAND and produce a 36-bit (effectively 32-bit) partial product for input to the FALU.

A 56-bit register called MIER is then loaded with the multiplier. For a single-precision multiplication, the multiplier fraction is loaded into the upper 24 bits of MIER [MIER (55:32)]. A multiplexer called MIERMUX routes MIER (39:32) to the partial product components.

Let's go through the steps again.

1. MIER (39:32), which contain the low eight bits of the multiplier, and MAND (27:00), which contain the multiplicand, are multiplied by the partial product components. The resultant partial product is passed through FALU (unaffected) and stored in the upper 32 bits of a 60-bit register called the AR.

2. The MIER is right-shifted eight places. MIER (39:32) now contain the middle eight bits of the multiplier and are multiplied with MAND. The partial product obtained by this multiplication is input to the FALU. The partial product obtained in step 1 (in the AR) is right-shifted eight places (by ASHFTR and BSHFTR) and is also applied to the FALU. The FALU adds its 2 inputs and the result is stored in the upper 40 bits of the AR.

3. The MIER is right-shifted eight places; MIER (39:32) now contain the high eight bits of the multiplier and are multiplied with MAND. The partial product obtained by this multiplication is input to the FALU. The partial product obtained in step 2 (in the AR) is right-shifted eight places (by ASHFTR and BSHFTR) and is also applied to the FALU. The FALU adds its two inputs, and the result is stored in the upper 48 bits of the AR. This fraction is then normalized. Note that the resultant product occupies the upper 48 bits of the AR and must be rounded or truncated before being stored in a destination accumulator in FSPAD.

6.5.2 Double-Precision Multiplication

Multiplication of two double-precision fractions is accomplished in 14 steps, as illustrated in Figure 6-6. At the time this fraction calculation is occurring, the two exponents of the floating-point numbers are being added in the EXPONENT data path and the sign of the result is being determined. The fraction obtained by step 14 must be normalized. If the fraction is shifted to the left, the exponent must be decreased accordingly.

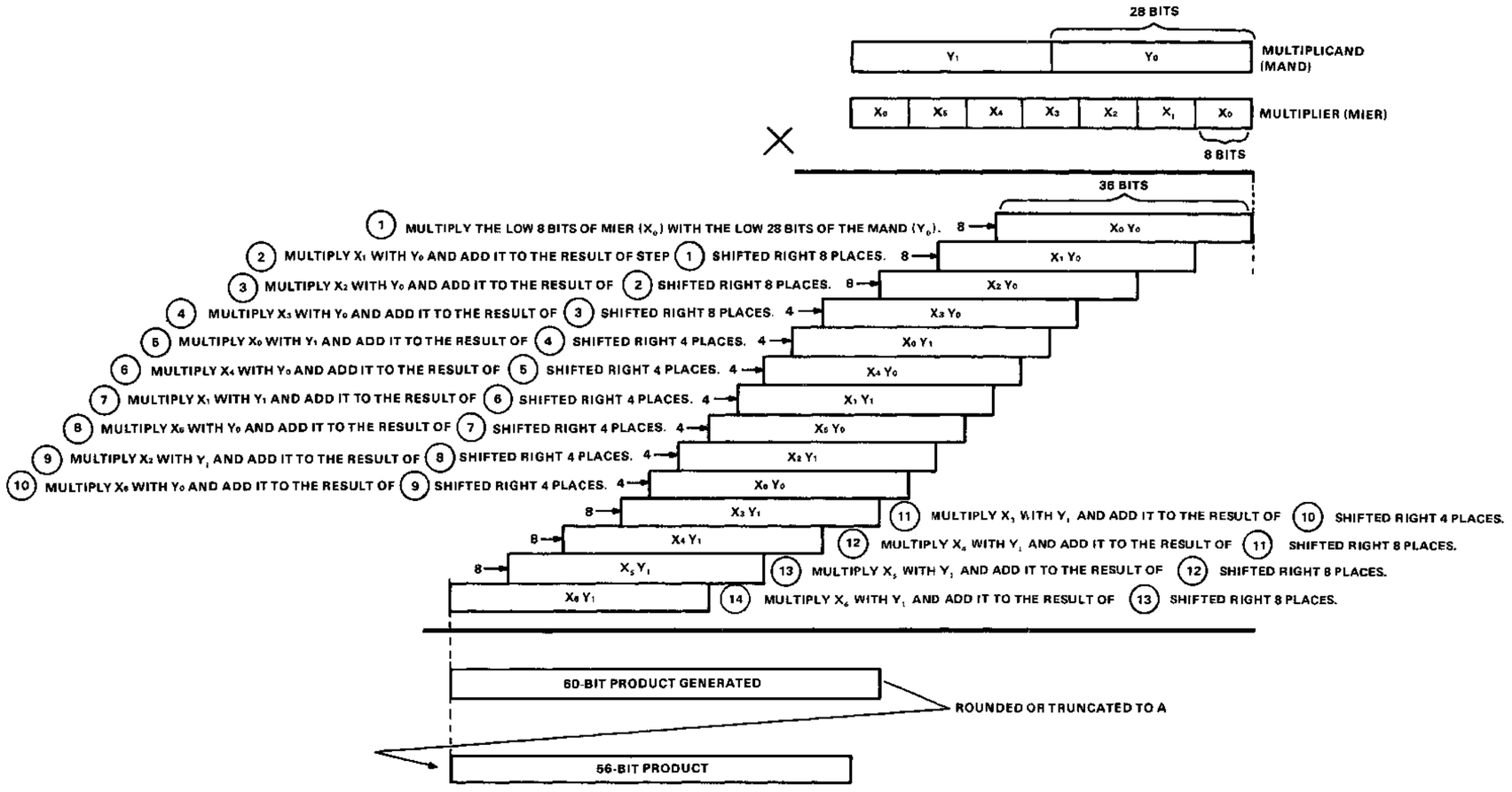


Figure 6-6 FP11-E Double-Precision Multiplication

A brief summary of the behavior of the FP11-E components during the 14 steps of a double-precision multiplication is given below.

1.	Load MIER and MAND with the multiplier and multiplicand, respectively. Multiply the lower eight bits of the MIER with MANDCO (via the partial product components) and store the result in the upper 36 bits of the AR.
2–4	Shift the MIER right eight places and multiply byte 0 of the MIER [MIER (07:00)] with MANDLO (via the partial product components). Shift the contents of the AR eight places right and add the partial product just obtained.
5.	Reload the MIER with the multiplier. Swap the MAND. Multiply byte 0 of the MIER with MANDLO (which now contains the contents of MANDHI because of the swap). Shift the contents of the AR right four places and add the partial product just obtained.
6, 8, 10	Swap the MAND and multiply the 4th byte of the MIER [MIER (39:32)] with MANDLO. Shift the contents of the AR right four places and add the partial product just obtained.
7, 9, 11	Swap the MAND and shift the MIER eight places right. Multiply byte 0 of the MIER with MANDLO (which now contains the contents of MANDHI because of the swap). Shift the contents of the AR right four places and add the partial product just obtained. After step 11, swap the MAND again.
12–14	Shift the MIER eight places right and multiply byte 0 of the MIER with MANDLO (which contains the contents of MANDHI because of the swap in step 11). Shift the contents of the AR right eight places and add the partial product just obtained.

After step 14, the fraction must be normalized. Note that the resultant product occupies all 60 bits of the AR and must be rounded or truncated before being stored in a destination accumulator in FSPAD.

It may not be immediately obvious how the FP11-E obtained the correct final product from the sequence in which the partial products were calculated and summed. However, we can arrange the partial products in any order as long as they are aligned correctly. By doing this, a more familiar arrangement of partial products is obtained (Figure 6-7). By comparing Figures 6-6 and 6-7, it can be seen that the two arrangements are equivalent.

6.5.3 Partial Product Components

From the previous paragraphs, it is evident that we do not directly multiply two 24-bit fractions during a single-precision multiplication, nor do we directly multiply two 56-bit fractions during a double-precision multiplication. Instead, we break up the multiplier fraction (and the multiplicand fraction, in double-precision multiplication) into pieces and multiply these pieces together to produce a number of partial products. These partial products are subsequently aligned and summed by components on the FALU board to produce our final product.

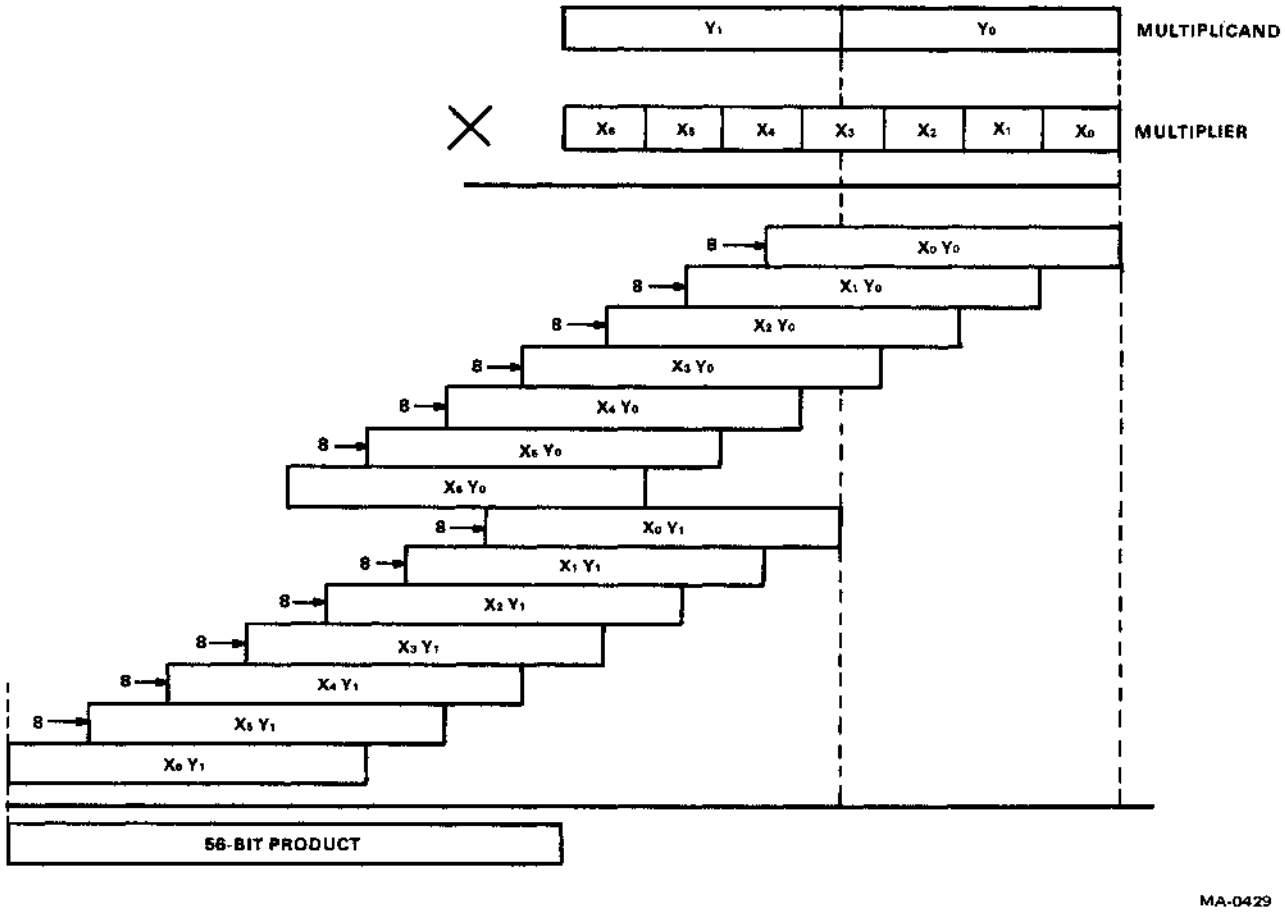


Figure 6-7 Rearrangement of Partial Products, Alignment Maintained

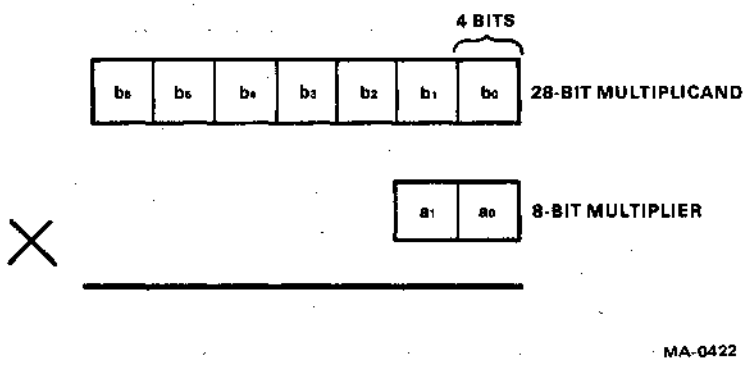
The components that produce the partial products are called the partial product components and are labeled MULNET ROM, MNETCARRY, MNETSUM, and MNETALU in Figure 6-1. Specifically, the components multiply an 8-bit piece of the multiplier with a 28-bit piece of the multiplicand to produce a 36-bit partial product that is input to the FALU board for further processing. This is accomplished in two steps.

1. MULNET ROM multiplies the 8-bit piece of the multiplier with the 28-bit piece of the multiplicand and produces 36 "sum" bits (which are loaded into the MNETSUM register) and 28 "carry" bits (which are loaded into the MNETCARRY register).

2. The lower four bits of MNETSUM are directly input to the FALU as the lower four bits of the 36-bit partial product. The other 32 bits of the partial product are obtained by adding the upper 32 bits of MNETSUM with the (aligned) contents of MNETCARRY, via the MNETALU.

The following paragraphs describe the operation of the partial product components in more detail.

6.5.3.1 MULNET ROM, MNETSUM, and MNETCARRY – The box labeled MULNET ROM consists of 28 ROMs. Fourteen of these ROMs (called "multiplier" ROMs) multiply an 8-bit piece of the multiplier with a 28-bit piece of the multiplicand. Consider the following illustration.



Notice that the multiplicand and multiplier are broken up into 4-bit segments. This is because each ROM has two 4-bit inputs. For example, a_0 and b_0 are applied to the two 4-bit inputs to one of the ROMs. This ROM (which has already been programmed with all the possible products of two 4-bit numbers) produces the 8-bit product of $a_0 \times b_0$. Similarly, there is a ROM dedicated to finding the product of a_0 and b_1, a ROM dedicated to finding the product of a_0 and b_2, and so on. Note again that there are 14 multiplier ROMs in all.

Now that we have all these *a* and *b* products, we align and sum them in a special way to give us a number of "sum" and "carry" bits. The 14 other ROMs in MULNET ROM (called "counter" or "adder" ROMs) are used for this purpose. Figure 6-8 shows the proper alignment of the *a* and *b* products.

Figure 6-8 also shows the aligned *a* and *b* products divided into fourteen 2-bit central columns and two 4-bit columns at either end. The purpose of the adder ROMs is to sum the contents of each 2-bit central column.

NOTE

The two 4-bit columns are the upper four bits of a_1b_6 and the lower four bits of a_0b_0. These bypass the adder ROMs and are sent directly to the upper four bits and lower four bits of the MNETSUM register, respectively.

Each 2-bit column has an adder ROM dedicated to it. Each ROM sums the contents of its column and produces two "sum" bits and two "carry" bits.

For example, examine column 14 in Figure 6-8. The two least significant bits of a_1b_0 are summed with the two least significant bits of a_0b_1 and the second most significant 2-bit piece of a_0b_0. The ROM which performs this summation produces two sum bits which are sent to the 5th and 6th least significant bits of the MNETSUM register. (Recall that the four least significant bits of MNETSUM are being fed directly with the four least significant bits of a_0b_0.) This ROM also produces two carry bits which are sent to the two least significant bits of the MNETCARRY register.

In all, the 14 adder ROMs produce 28 sum bits (sent to the central 28 bits of MNETSUM) and 28 carry bits (sent to the MNETCARRY register).

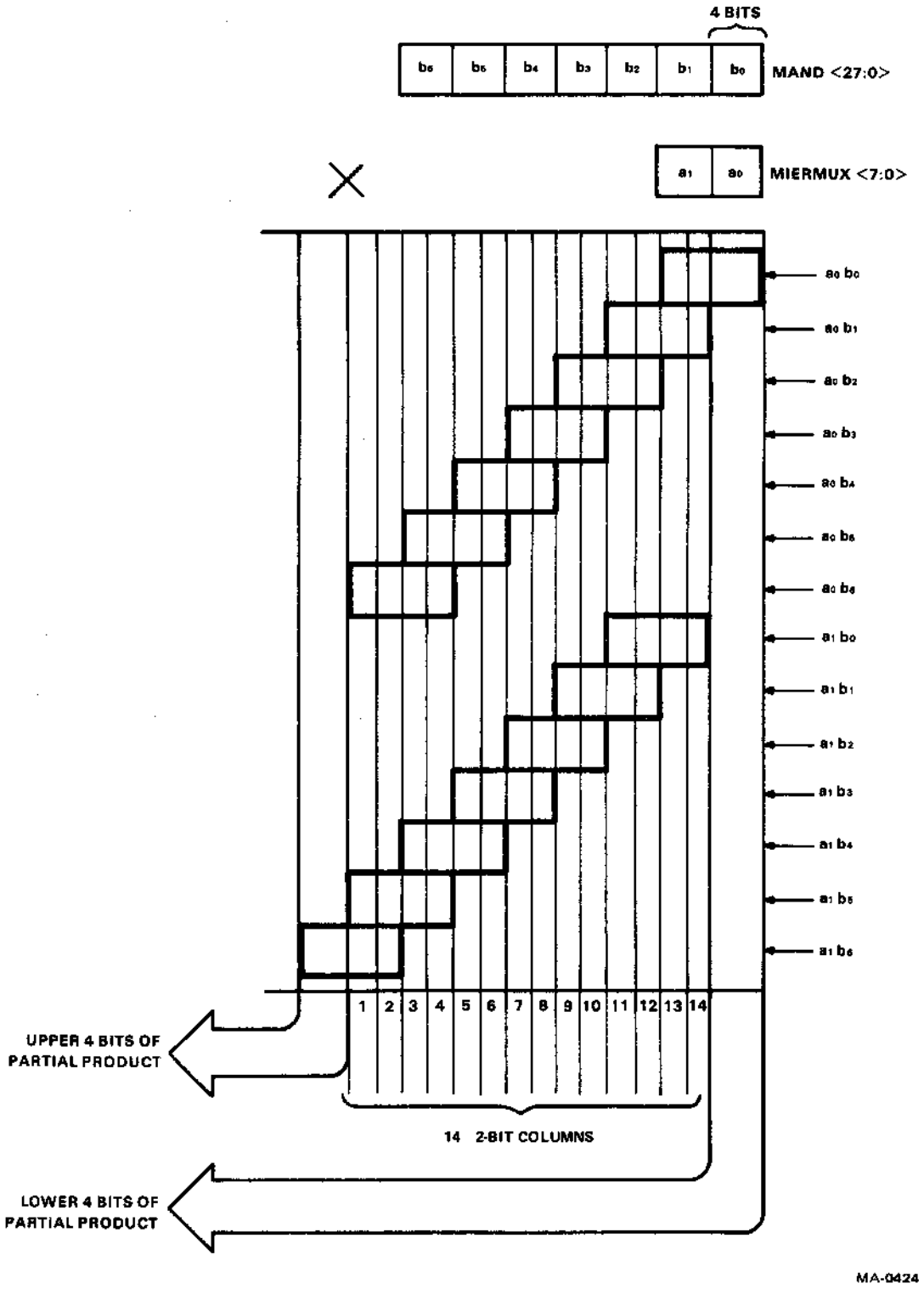


Figure 6-8 Division of Aligned A and B Products into Columns

6.5.3.2 MNETALU - The MNETALU aligns and adds the 32 most significant bits of MNETSUM with the 28 MNETCARRY bits. It produces a 32-bit output as illustrated in Figure 6-9.

The four least significant bits of MNETSUM are appended to the low end of this result to yield our final 36-bit partial product. This partial product is then input to the FALU board for further manipulation.

It should be realized that to produce our final partial product, we effectively added together all of the *a* and *b* products produced by the multiplier ROMs in MULNET ROM.

The MNETALU also acts as a gate for the FBUSA bus. When required, it allows data on FBUSA (58:27) to be routed to the FALU.

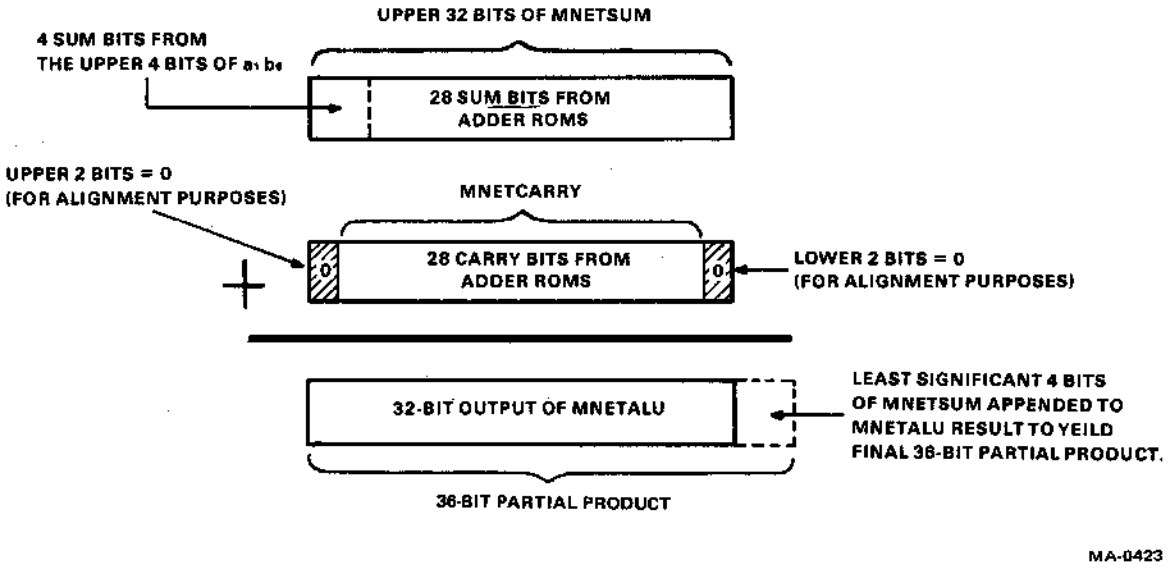


Figure 6-9 MNETALU Operation and Final Partial Product

6.5.3.3 MIER - The MIER is a 56-bit shift register which loads and shifts the multiplier during floating-point multiplication instructions. It has one data input (FBUSA) and two data outputs [MIER (39:32) and MIER (07:00)]. The MIER receives its contents via FBUSA from a register contained in FSPAD.

During a single-precision multiplication, the multiplier is loaded into the upper 24 bits of the MIER. During a double-precision multiplication, the multiplier is loaded into the entire 56 bits of the MIER.

The inputs to the MIER are wired such that an 8-bit right shift (or left shift) can occur in one jump when the register is clocked. Furthermore, the MIER is a rotative shift register, i.e., MIER's least significant bit and most significant bit are linked (Figure 6-10).

Specifics of MIER's operation are found in the paragraphs describing single- and double-precision multiplication.

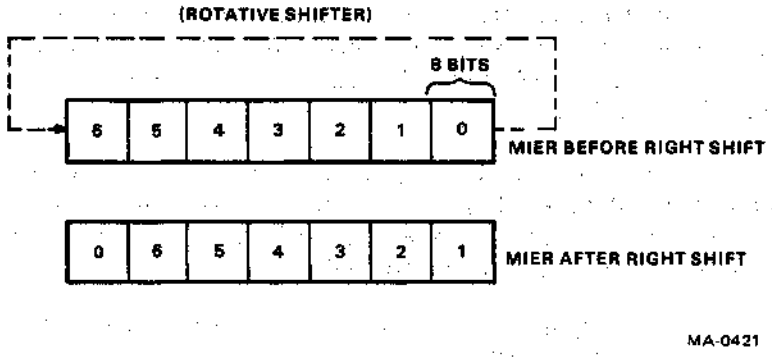


Figure 6-10 MIER – Right Shift

6.5.3.4 MIERMUX – The MIERMUX is a 2:1 multiplexer that performs one of two functions. It either routes MIER (39:32) to the MULNET ROMs during a single-precision multiplication or routes MIER (07:00) to the MULNET ROMs during a double-precision multiplication.

6.5.3.5 MAND – The MAND is a 56-bit shift register which loads and shifts the multiplicand during floating-point multiplication instructions. It has two data inputs from FBUSA. One data input [FBUSA (58:31)] is fed to the upper half of the MAND (called MANDHI) and the other data input [FBUSA (30:37)] is fed to the lower half of the MAND (called MANDLO). The MAND has one 28-bit output [MAND (27:00)] which transfers the contents of MANDLO to the MULNET ROMs.

The inputs to the MAND are wired such that a 28-bit right shift (or "swap") can occur in one jump when the register is clocked. Like the MIER, the MAND is also a rotative shifter. Thus, when the MAND performs a right shift, the net effect is that MANDHI is swapped with MANDLO (Figure 6-11).

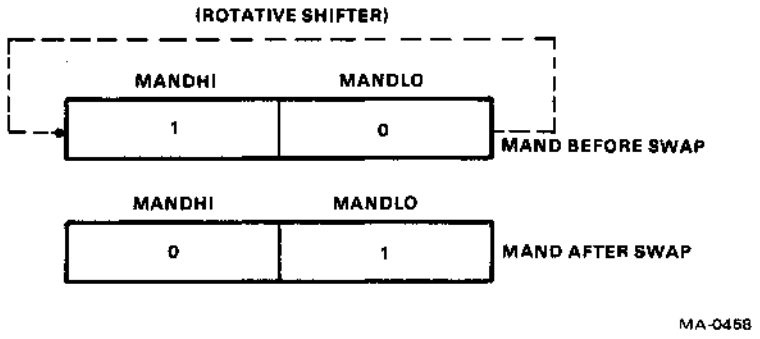


Figure 6-11 MAND – Swap (28-Bit Right Shift)

6.5.3.6 FPOUTMUX – The FPOUTMUX is a 4:1 multiplexer located on the MULNET board. The FPOUTMUX routes one of its four 16-bit data inputs to BUSDIN, for use by the CPU. As shown in Figure 6-1, these four inputs are: SSPADB, FBUSE (07:00) and FBUSA (57:51); FBUSA (50:35); FBUSA (34:19); and FBUSA (18:03).

Formatted data words leaving the FP11-E are routed through FPOUTMUX 16 bits at a time. The most significant 16 bits of the data word are routed out first, followed by the next 16 significant bits, and so on. Note that for a floating-point data word, the 16 most significant bits also include the sign bit (SSPADB) and an 8-bit exponent [FBUSE (07:00)].

Floating exception code (FEC) information, which appears on FBUSA (34:19), is also routed to BUSDIN and the CPU via the FPOUTMUX.

CHAPTER 7
FP11-E MICROINSTRUCTION

This chapter presents an introduction to the operation of the FP11-E control logic by examining the fields of an 88-bit FP11-E microinstruction.

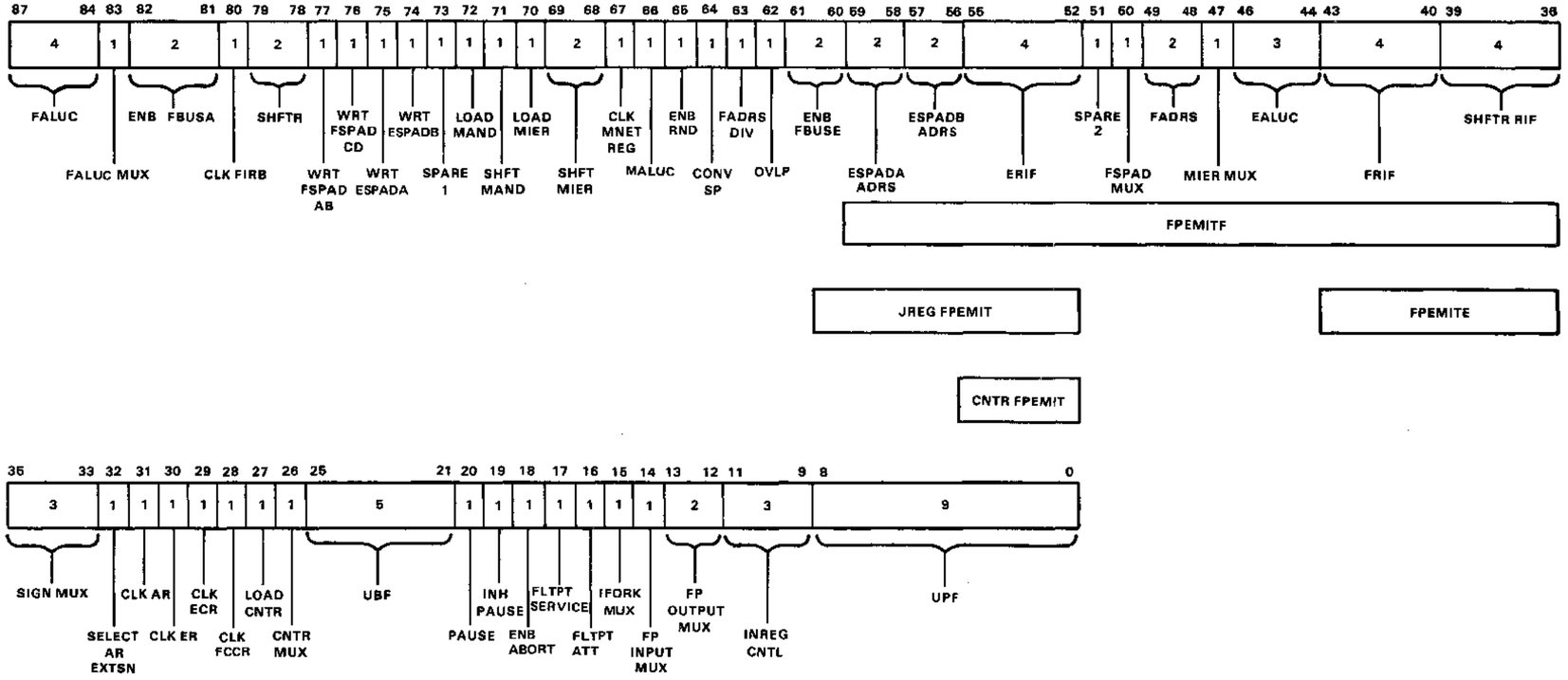
As mentioned in Chapter 6, the FP11-E performs floating-point instructions and other operations by addressing sequences of microinstructions. These microinstructions are contained in a part of the FP11-E control logic called the ROM control store. In general, these microinstructions:

1. Control the operation of the FP11-E data manipulation components (see Chapter 6 for background information)

2. Help coordinate the operation of the FP11-E with the operation of the CPU (see Chapter 5 for background information)

3. Control the operation of parts of the FP11-E control logic (see *FP11-E Floating-Point Processor Technical Manual* for more detailed information).

As shown in Figure 7-1, an FP11-E microinstruction is divided into a number of fields (called microfields), each of which performs unique functions. Table 7-1 describes the functions of these microfields.



MA-0419

Figure 7-1 FP11-E Microinstruction

Table 7-1 FP11-E Microfield Description

Field	Mnemonic	Function
87:84	FALUC	Specifies the function of the FALU. 0000 ~A 1011 A·B 0101 ~B 1110 A+B 0110 A minus B 1111 A 1001 A plus B 0011 0 1010 B
83	FALUC MUX (Not the same as FALU MUX, mentioned in Chapter 6).	When set, causes the FALU to receive its function control from a "prep 2 table" (*FP11-E Floating-Point Processor Technical Manual*), rather than from the FALUC microfield.
82:81	ENB FBUSA	Specifies the source of FBUSA. 00 FSPAD 10 MNETREG 01 FPEMITF 11 INBUF
80	CLK FIRB	When set, enables clocking of FIRB.
79:78	SHFTR	Specifies the source of function control for the ASHFTR/BSHFTR combination. 00 Shifter receives its function control from SHFTR RIF microfield. 01 Shifter receives its function control from PRESHFK logic. (This logic produces an output based on the difference between the two exponents of a floating-point addition or subtraction.) Used for prealignment of fractions. 10 Shifter receives its function control from NORMK.
77	WRT FSPAD AB	When set, enables the writing of data into the A and B fields of a selected register in FSPAD.
76	WRT FSPAD CD	When set, enables the writing of data into the C and D fields of a selected register in FSPAD.
75	WRT ESPADA	When set, enables the writing of data into ESPADA.
74	WRT ESPADB	When set, enables the writing of data into ESPADB.
73	SPARE 1	Reserved for future use.

Table 7-1 FP11-E Microfield Description (Cont)

Field	Mnemonic	Function
72	LOAD MAND	When set, enables loading of the MAND register.
71	SHFT MAND	When set, enables swapping of upper half of MAND with lower half of MAND.
70	LOAD MIER	When set, enables loading of the MIER register.
69:68	SHFT MIER	When set, enables MIER shifting. 01 Right shift MIER 10 Left shift MIER
67	CLK MNETREG	When set, enables clocking of the MNETSUM and MNETCARRY registers.
66	MALUC	Specifies the function of the MNETALU. 0 A plus B 1 A
65	ENB RND	When set, enables single- or double-precision fraction rounding. Used in conjunction with the CONV SP microfield, FD, and FT.
64	CONV SP	When set, enables reinterpretation of the FD bit. If the FD bit is set when CONV SP is set, the FP11-E can behave as if it were in F mode. If the FD bit is cleared when CONV SP is set, the FP11-E can behave as if it were in D mode.
63	FADRS DIV	When set, enables FALU 59 control over FSPAD address. If FALU 59 is cleared when FADRS DIV is set, the least significant bit of the FRIF field is set. This allows access to either the actual or 2's complement divisor during Divide instructions.
62	OVLP	When set, the operation of the last microinstruction of the current (macro) instruction is overlapped with the first microinstruction of the next (macro) instruction, thus saving one microinstruction.
61:60	ENB FBUSE	Specifies source of FBUSE. 00 ESPADB 10 FPEMITE 01 INBUFA 11 EADJ

Table 7-1 FP11-E Microfield Description (Cont)

Field	Mnemonic	Function
59:58	ESPADA ADRS	Specifies source of ESPADA address. 00 DF – Destination field – FIRB (07:06) or FIRA (07:06) 01 SF – Source field – FIRB (02:00) or FIRA (02:00) 10 RIF – ERIF field 11 DF v 1 – Destination field with its LSB forced to 1
57:56	ESPADB ADRS	Specifies source of ESPADB address. 00 SF – Source field – FIRB (02:00) or FIRA (02:00) 10 RIF – ERIF field 11 DF – Destination field – FIRB (07:06) or FIRA (07:06)
55:52	ERIF	Specifies exponent and sign scratchpad registers. These registers are numbered from 0 to 17. Used in conjunction with the ESPADA ADRS and ESPADB ADRS microfields.
51	SPARE 2	Reserved for future use.
50	FSPAD MUX	Selects source of data to be written into the A and B fields of FSPAD. 00 AR 01 INBUF
49:48	FADRS	Specifies the source of FSPAD address. 00 DF – Destination field – FIRB (07:06) 01 SF – Source field – FIRB (02:00) or FIRA (02:00) 10 RIF – FRIF field 11 DF v 1 – Destination field with its LSB forced to 1.
47	MIER MUX	Specifies the MIER byte that is input to MULNET ROM. 00 BYTE 0 – MIER (07:00) 01 BYTE 4 – MIER (39:32)

Table 7-1 FP11-E Microfield Description (Cont)

Field	Mnemonic	Function
46:44	EALUC	Specifies the function of the EALU. 001A plus B 110 A+B 010A minus B 111 B
43:40	FRIF	Specifies a fraction scratchpad register. These registers are numbered from 0 to 17_8. Used in conjunction with the FADRS microfield.
39:36	SHFTR RIF	Specifies the number of shifts (and the shift direction) that the ASHFTR/BSHFTR combination must perform in one pass. 0000 Left 4 1000 Right 4 0001 Left 3 1001 Right 5 0010 Left 2 1010 Right 6 0011 Left 1 1011 Right 7 0100 No shift 1100 Right 8 0101 Right 1 1101 Right 9 0110 Right 2 1110 Right 10 0111 Right 3 1111 Right 11
35:33	SIGN MUX	Specifies the value of the SD (destination sign) bit. The SS (source sign) bit is affected by the SIGN MUX field only when (35:33) = 111. 000 SD←SD 100 SD←SS XOR SD 001 SD←~SD 101 SD←INBUFA 15 010 SD←SS 110 SD←0 011 SD←~SS 111 SD←SSPADA SS SSPADB
32	SELECT AR EXTSN	This field controls the FALU MUX mentioned in Chapter 6. If set, FSPAD (02:00) (containing the guard bits) are routed to FALU MUX (02:00). If cleared, DROUND, 0, 0 are routed to FALU-MUX (02:00).
31	CLK AR	When set, enables clocking of AR.
30	CLK ER	When set, enables clocking of ER.
29	CLK ECR	When set, enables clocking of the ECR, a register on the EXPONENT board which indicates various conditions of the EXPONENT data path. The ECR is used in FP11-E microinstruction branching.

Table 7-1 FP11-E Microfield Description (Cont)

Field	Mnemonic	Function
28	CLK FCCR	When set, enables clocking of the FCCR, a register on the EXPONENT board which indicates whether the EALU has performed an operation that has resulted in an overflow, a negative number, or zero.
27	LOAD CNTR	When set, enables loading of counter. This counter is loaded with a 4-bit constant and is incremented by one each time it is enabled and clocked.
26	CNTR MUX	This field determines the source of the counter's 4-bit input. 0 CNTR FPEMIT field (see end of section) 1 PRESFHK quotient field (*FP11-E Floating-Point Processor Technical Manual*
25:21	UBF	Microbranch field. This field, in conjunction with the UPF (micropointer field) determines the FNUA, or next FP11-E microaddress. Each UPF combination specifies from one to three conditions that may cause the FNUA board to specify an FP11-E microaddress other than the one specified by the UPF as the next microaddress (*FP11-E Floating-Point Processor Technical Manual*). There is also a "null" UBF combination that unconditionally allows the UPF field to specify the FNUA.
20	PAUSE	When set, disables further sequencing of the FNUA logic until FLTPT GO (Chapter 5) is received from the CPU.
19	INH PAUSE	When set in the current microinstruction, the PAUSE microfield of the next microinstruction will be treated as if it were cleared.
18	ABORT	When set, enables the CPU to abort the current FP11-E instruction for a variety of abort conditions (Chapter 5).
17	FLTPT SERVICE	When set, is a request to the CPU for FP11-E service (Chapter 5).
16	FLTPT ATT	When set, informs the CPU that the FP11-E is ready to proceed with an operation. This allows the CPU to leave an idle state and resume its activity.

Table 7-1 FP11-E Microfield Description (Cont)

Field	Mnemonic	Function
15	IFORK MUX	Selects the input to a piece of FP11-E control logic called the IFORKMUX. Refer to the *FP11-E Floating-Point Processor Technical Manual.*
14	FPINPUT MUX	Selects source of data input to the FP11-E. 0 DMUX 1 DOUT
13:12	FPOUTPUT MUX	Selects source of data output from FP11-E. 00 SD, FBUSE (07:00), FBUSA (57:51) 01 FBUSA (50:35) 10 FBUSA (34:19) 11 FBUSA (18:03)
11:09	INREG CNTL	Specifies operations of INBUFA, INBUFB, and a register called the microbreak register. 001 Enables loading of INBUFA 010 Enables loading of INBUFB at time P2 (*FP11-E Floating-Point Processor Technical Manual*) 011 Enables loading of INBUFB at time P3 (*FP11-E Floating-Point Processor Technical Manual*) 100 Left-shifts INBUFA and INBUFB (for storage of quotient during Divide instruction) 101 Enables loading of microbreak register (*FP11-E Floating-Point Processor Technical Manual*)
08:00	UPF	Micropointer field. A 9-bit address which, in conjunction with the UBF field, is used by the FNUA board to determine the next microaddress.

NOTE

Four fields of a microinstruction can be interpreted as constants when so specified by the microinstruction. These fields are (59:36) (also called the FPEMITF field), (43:36) (also called the FPEMITE field), (60:52) (also called the JREG FPEMIT field), and (55:52) (also called the CNTR FPEMIT field).

The FPEMITF and FPEMITE fields are discussed in Chapter 6. The JREG FPEMIT and CNTR FPEMIT fields are associated with the FP11-E control logic and are explained in the FP11-E Floating-Point Processor Technical Manual.

CHAPTER 8
FP11-E INSTALLATION PROCEDURE

1. Swing open the doors on the back of the PDP-11/60 cabinet.

2. Disconnect primary power from the PDP-11/60 by switching main power breakers (located on the back of the power controller box) to their OFF position.

3. Secure the FP11-E power supply box (H7421A) to rails with four screws as shown in Figure 8-1. As viewed from the back of the PDP-11/60, these rails are located on either side of the PDP-11/60's lower right quadrant. When mounted, the top of the power supply box should be close to the top of the quadrant.

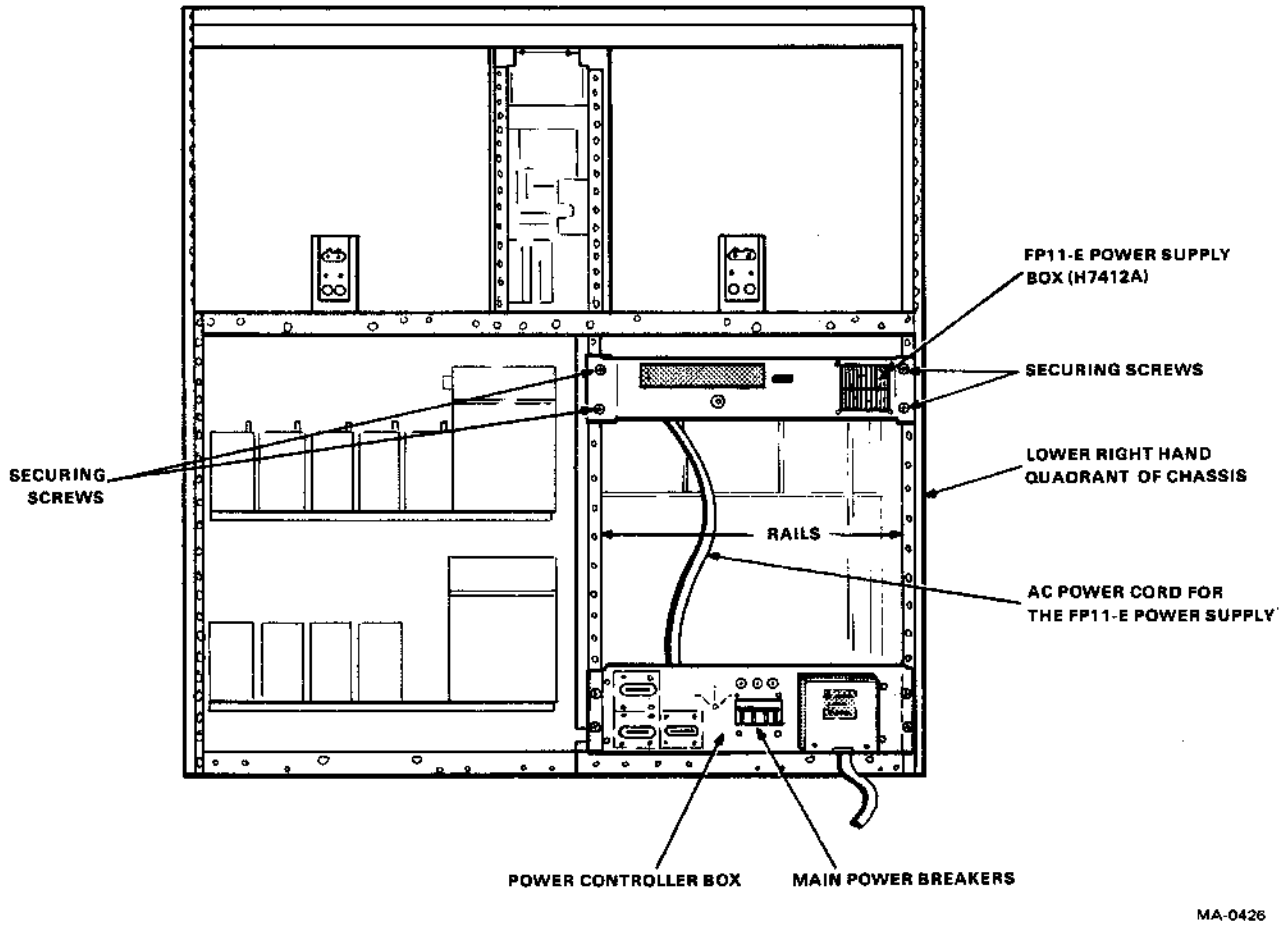


Figure 8-1 PDP-11/60 Chassis (Rear View)

4. Insert the ac power cord of the FP11-E's power supply into a socket labeled PHASE 2 SWITCHED located on the front of the power controller box as shown in Figure 8-2.

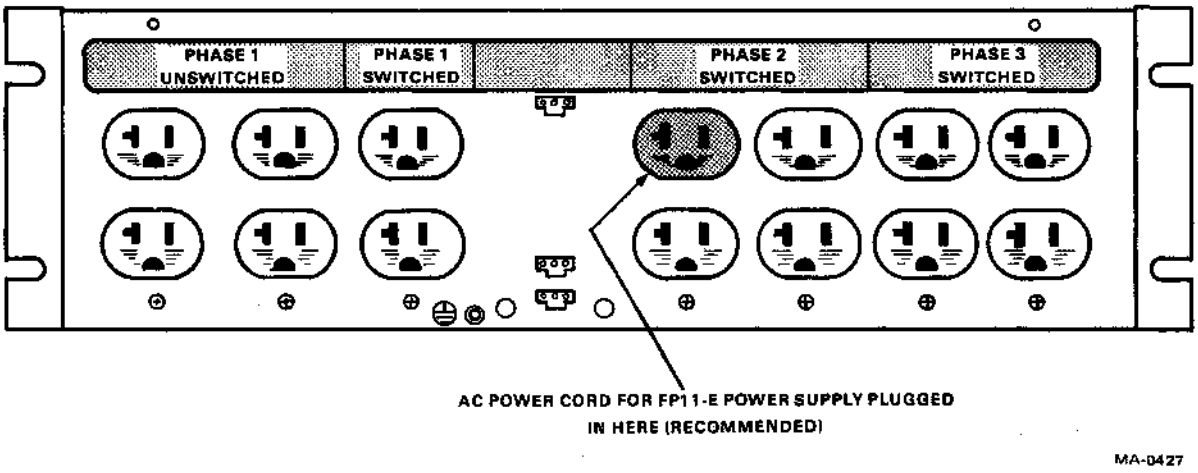


Figure 8-2 Power Controller Box (Front View)

5. Remove the front cover of the PDP-11/60 cabinet.

6. Swing the BA11-PA box out of the cabinet. This box contains circuit boards and, when viewing the PDP-11/60 from the front, is located in the PDP-11/60's lower left quadrant (Figure 8-3).

7. Remove the top cover of the BA11-PA box by removing four screws, as shown in Figure 8-3.

8. Locate socket J4 on the power distribution board inside the BA11-PA box.

9. Connect the +5 Vdc cable from the FP11-E power supply to socket J4 (Figure 8-4).

10. Switch main power breakers ON and turn the console rotary switch to POWER position.

11. Test for +5 Vdc at backpanel pins A8A2, A9A2, A10A2, and A11A2.

12. After ensuring that +5 Vdc is present at all four pins, disconnect primary power by switching main power breakers OFF.

13. Replace top cover of the BA11-PA box by replacing the four screws.

14. Swing BA11-PA box back into cabinet.

15. Install the FALU board (M7881) into slot 11 of the BA11-PA box.

16. Install the MULNET board (M7880) into slot 10 of the BA11-PA box.

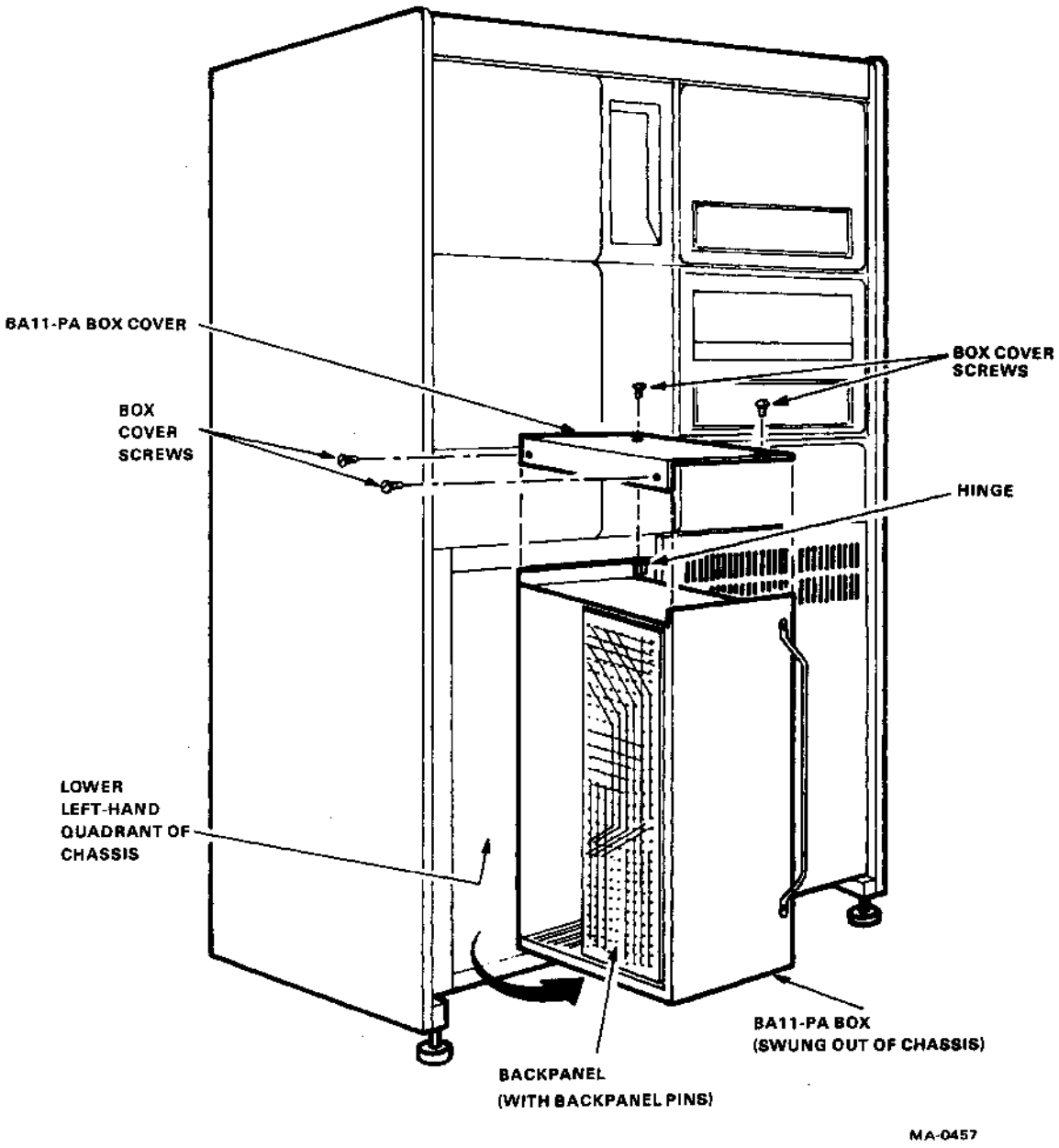


Figure 8-3 PDP-11/60 Chassis (Front View)

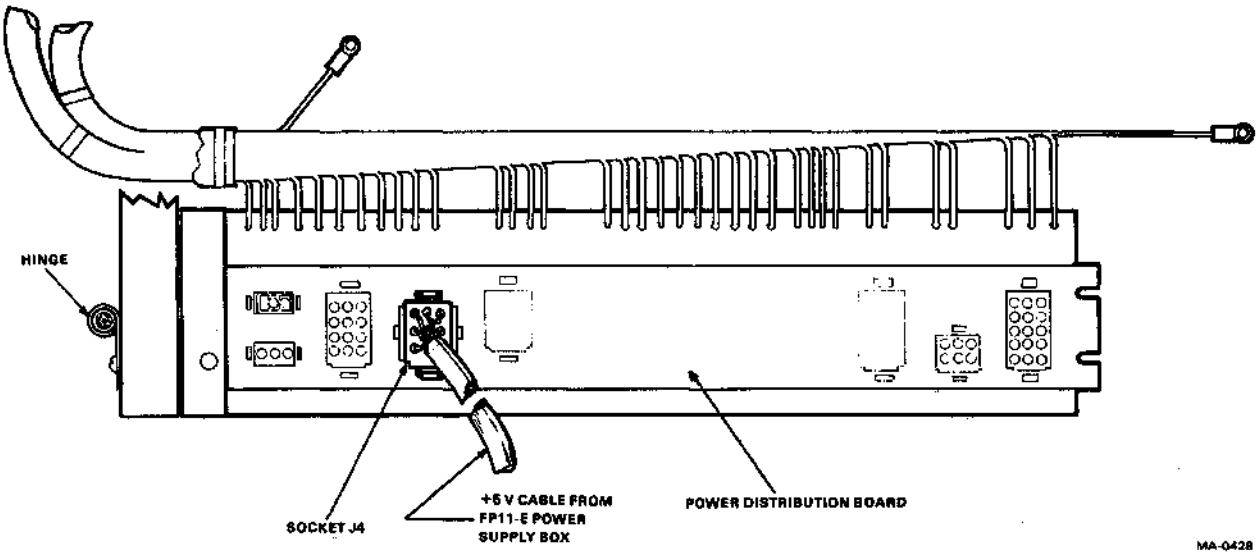


Figure 8-4 Top of BA11-PA Box
(Cover Off)

17. Install the EXPONENT board (M7879) into slot 9 of the BA11-PA box.
18. Install the FNUA board (M7878) into slot 8 of the BA11-PA box.
19. Replace front of PDP-11/60 cabinet.
20. Switch main power breakers ON.
21. Close doors in the back of the PDP-11/60 cabinet.
22. Run the following FP11-E diagnostics to ensure correct FP11-E operation.

    MD-11-DQFPA-B
    MD-11-DQFPB-B
    MD-11-DQFPC-B
    MD-11-DQFPD-B
    MD-11-DQFPE-A

Reader's Comments

FP11-E FLOATING-POINT PROCESSOR
USER'S GUIDE
EK-FP11E-UG-001

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-FP11E-UG-001

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

Digital Equipment Corporation
Technical Documentation Department
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.