Gordon Bell

0309

Confidential

TITLE: The PDP-11 Floating Point Unit

PDP-11/40 Technical Memo #31 35 pages

Authors: Ad van de Goor, Len Hughes

Date: February 1971

Revision: No. 2 Obsolete: 23 and 23A

Index Key: Floating Point Instructions
Virtual Address Space
Physical Address Space
Bus Option
Internal Option

Distribution: PDP-11/40 Working List

ABSTRACT

The purpose of this memo is to describe the nature of the Floating Point Unit "FPU". The FPU is designed to be a Unibus option for the 11/05 and the 11/20. For the 11/40, the FPU is planned to be an internal option.

The FPU is capable of executing single and double precision (i.e. 32 and 64-bit) floating point instructions and is capable of reading and writing its own operands from and into memory. Once an FPU instruction has been started, it can continue without CPU intervention, leaving the CPU free to execute other (i.e. non-FPU) instructions.

0.0 INTRODUCTION

The position of the PDP-11 in the market is such that some floating point arithmetic capabilities are very desirable, if not necessary. Considering the complexity, and therefore the price of a floating point unit "FPU", it should be available as an option only.

Some questions to be answered concerning the FPU option are listed below and elaborated on in the following sections.

1) Internal versus Bus Option
2) FPU - CPU interaction
3) The FPU's Instruction and Data Formats
4) The FPU's Instruction Set

1.0 INTERNAL VERSUS BUS OPTION

The FPU is thought of as a fairly independent processor, i.e. when started it is supposed to finish the instruction independent of the CPU. This includes reading and writing data from and into memory. Therefore, the FPU has to be connected to the bus.

Because the 11/40 will have two memory buses (i.e. a fast synchronous one and the asynchronous Unibus) it should at least be connectable to the Unibus in order to make it acceptable for the 11/50 and 11/20.

The next question to be solved in this section is whether the FPU should operate in virtual or physical address space. Figure 1-1 shows the configuration with the FPU operating in virtual address space. In Figure 1-2 the FPU operates in physical address space.

The virtual address space is defined as the address space the user runs in; the physical address space is defined as the set of core locations actually addressed. For a machine which does not have address mapping (e.g. relocate protect) the virtual address space is identical to the physical address space.

Looking at the solution of Figure 1-2, the following comments can be made:

1) The addresses of the operands have to be passed to the FPU as physical addresses. Looking at the Relocate Protect option, this means that it should recognize certain FPU addresses and not relocate them. Instead, it should take the data (on the data lines) which contain the virtual address and relocate it as if it were an address. This requires special controls and data paths in the relocate protect option.

2) The Relocate/Protect option might have Read/Write protect bits which might have to be duplicated in the FPU, or the Relocate/Protect option has to have knowledge relevant to the use of the virtual addresses it relocates.

3) In case of the modes (R)+ or -(R) address violations can occur which can be detected with difficulty by the Relocate/Protect option.

The solution of Figure 1-1, i.e. let the FPU operate in virtual memory, has none of the above disadvantages. In Figure 1-1 it is treated in the same way as the CPU for which the Relocate/Protect option is designed. Clearly, from the above it can be stated that the FPU should operate in virtual space.

Because of mechanical limitations and restrictions on the fast bus, it is most desirable that the FPU take up no more than one system unit if the solution of Figure 1-1 is implemented.

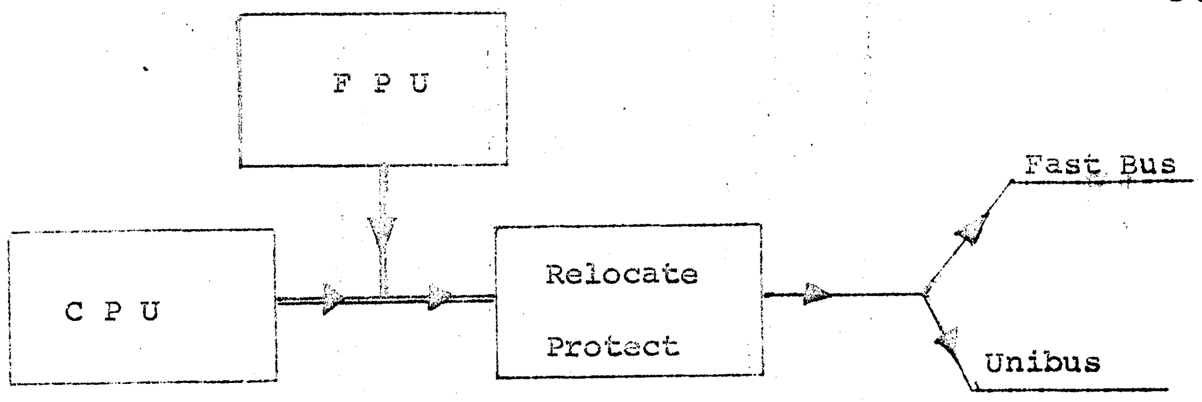


Figure 1-1 Configuration with FPU
Operating in VIRTUAL Address Space
(Presuming an open-collector internal bus.)

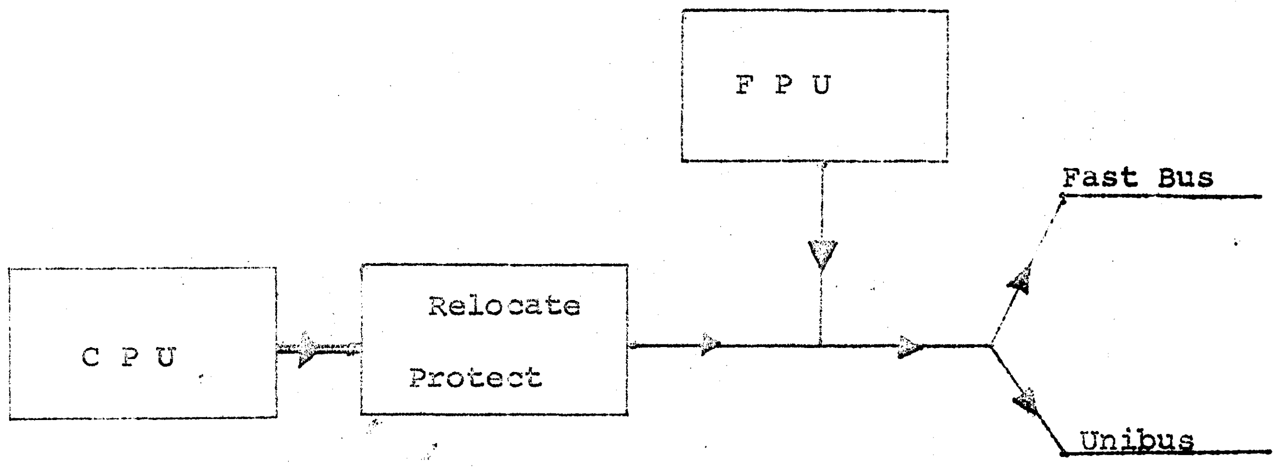


Figure 1-2 Configuration with FPU Operating in
PHYSICAL Address Space

NOTE: ⇉ = Virtual Address Lines
→ = Physical Address Lines

2.0 FPU-CPU INTERACTION

Because the FPU will be an independent processor, it is possible to allow it to finish a started instruction independent of the CPU. This allows the CPU to be handled in either of two ways.

1) The CPU is declared busy while the FPU carries out its operation. This has the advantage that it would deteriorate the interrupt response time (because floating point operations tend to take a relatively long execution time). It also prohibits the CPU from executing other, i.e. non-floating, instructions.

2) Allow the CPU to continue executing non-floating instructions once the FPU is set up (i.e. ready to start executing). This allows the CPU to carry out subscript computation, etc., in parallel with the execution of the floating point instruction thus improving the overall execution time. This method is selected because of its described advantages.

   Because the FPU is a bus option on the 11/05 and 11/20, the FPU OP code and the source address have to be transferred to the FPU. The FPU-CPU interaction takes place for the 11/05, 11/20, and 11/40 as described in the next sections.

2.1 FPU-11/20 INTERACTION

In previous memos the FPU was activated by the 11/20 through a sequence of MOV instructions, as described in Technical Memos 23 and 23-A. A typical sequence looked like the one given below for the case of the instruction MULF A(Rx), AC1

```
ADD #A, Rx          ; compute address

SUB FBR, PC         ; test for FPU busy (FBR)=4 when FPU busy else Ø

MOV PC, FPC         ; save PC

MOV Rx, FIR+FOC+AC ; move operand address and start FPU
```

The above sequence takes 8 words and has to be repeated for every FPU instruction of the above type.

The new scheme requires that every FPU instruction (like: MULF A(R2), AC1) is preceded by a JSR. The JSR allows the FPU to take control over the CPU. The FPU uses the CPU for address computation, stack pointer adjustments, etc., and acts like a hardwired interpreter. The JSR instruction has to be the following, "JSR R7, FPU" where FPU is an address in the I/O area. An example of this is given below. (next page)

```
        JSR  R7,  FPU                ; typical call sequence in
        MULF  A(R2),  AC1            ; user's program
        ---------------------------------------------------------------------
       /BR.                          ; the I/O address FPU contains
FPU<
       \MOV  (R6) + , FRA*           ; "BR.      when the FPU is busy
                                     ; otherwise it contains "MOV (R6) + , FRA"
                                     ; When the latter instruction is executed
                                     ; the return address is popped off
                                     ; the stack into the FPU's FRA register
        @(FRA) ---> FIR*             ; The instruction is fetched, under
        (FRA) +2 ---> FRA            ; hardware control, and loaded in the
                                     ; FIR register and FRA is incremented
        MOV  R2,  FDA                ; The CPU's register R2 is read
        (FRA) + (FRA) ---> FDA*              ; The index computation "A+(R2)"
        (FRA) + 2 ---> FRA           ; is done under hardware control
                                     ; and FRA is incremented
        2 or 4 data fetches          ; Depends on the mode of the FPU
        MOV  FRA,  PC                ; Control is transferred back to the
                                     ; CPU while the FPU does the
                                     ; required operation
```

* FRA means Floating Return Address

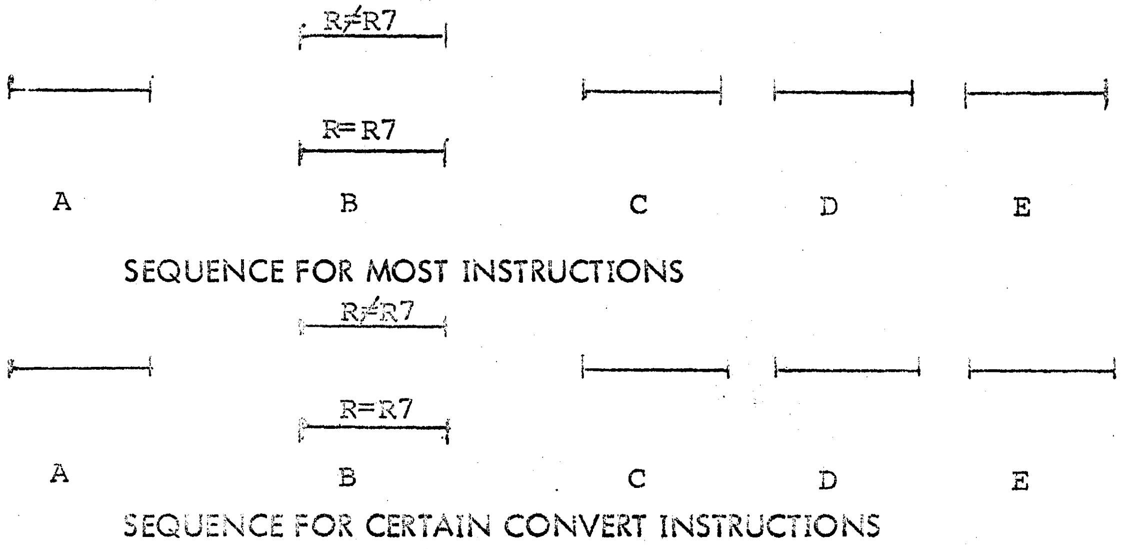
* FIR means Floating Instruction Register

* FDA means Floating Data Address

A summary of the different CPU instructions issued by the FPU is given below.

0314

This sequence of CPU instructions is interleaved with FPU data fetches/stores done under control of the FPU hardware for greater efficiency. A complete FPU instruction execution cycle can be divided into 6 sub-cycles as shown in Figure 2-1 below.


R≠R7
R=R7
A
B
C
D
E
SEQUENCE FOR MOST INSTRUCTIONS
R≠R7
R=R7
A
B
C
D
E
SEQUENCE FOR CERTAIN CONVERT INSTRUCTIONS

A = Instruction Fetch

B = Operand Address Computation, two paths depending on R=R7/R≠R7

C = Data Fetches/Stores

D = Transfer Control from FPU back to CPU

E = Execution

FIGURE 2-1, FPU instruction Subsequences

Below is the sequence of FPU issued instructions and FPU actions which are required for the address computation and final execution of most FPU instructions. The capital letters preceding the sections correspond to the subsequences of Figure 2-1.

```
A:    FPU: BR .                  ; FPU busy loop

           MOV (R6) +,FRA        ; Get return address

            (FRA)→FIR            ; Get instruction, both parts

           (FRA)  +2→FRA         ; done by FPU hardware
```

```
5  @ -(R)  *(FRA)-2 ⟶ FRA
           *@(FRA) ⟶ FDA

6  A(R)    *@ (FRA) ⟶ FDA
           *(FRA) +2 ⟶ FRA
           *(FRA) +(FDA) ⟶ FDA

7  @ A(R) *@(FRA) ⟶ FDA
          *(FRA) +2 ⟶ FRA
          *(FRA) + (FDA) ⟶ FDA
        *@(FDA) ⟶ FDA
```

C. Possible data fetches/stores

These happen at priority level 7

D. MOV FRA, PC ; transfer control back to CPU
; to execute non FPU initiated
; instructions

E. Execute the FPU instruction, i.e. perform the actual multiplication, etc.

2.1.1 CPU CONDITION CODES

The CPU condition code bits C, N, Z, and V which existed just prior to the FPU instruction, are destroyed. This is caused by the instructions the FPU issues to the CPU.

The FPU has its own set of condition code bits, "FC, FN, FZ, and FV". These can be transferred to the CPU's condition code bits under control of a special instruction Copy Floating Condition "CFCC".

2.1.2 PRIORITY LEVEL OF THE FPU

On the 11/05 and the 11/20 the FPU will be a Unibus option. The priority level of the FPU will be 7.

In order not to increase NPR latency, the FPU will monitor the NPR line and give up the bus between memory cycles.

The 11/20 bus priority arbitrator requires a MSYN signal to transfer Bus Mastership between peripherals. In certain special cases this could lead to the execution of an instruction before the bus would be rearbitrated to another requesting device (e.g. the FPU). The execution of an out of sequence instruction would be in conflict with the correct operation of the FPU, as will be clear from Section 3.0. This is prevented by "feeding" the CPU a "BR." instruction when the above condition occurs and the FPU is in control.

When the CPU wants to make use of the FPU, the CPU's priority level should be less than 7 (i.e. PR<7). When PR=7, the FPU will not be able to become bus master, because the CPU's PR=7 is considered to be higher. This will cause the CPU to be in an infinite loop executing the "BR ." instruction as described above, once it tries to execute an FPU instruction.

2.1.3 ALTERNATIVE FPU-11/20 INTERACTION

The method of Section 2.1 requires every FPU instruction to be preceded by a JSR. An alternative method is to issue the FPU instruction "as is" and have a trap service routine to transfer control to the FPU. An example of such a routine is given below. (It should be noted that all FPU OP codes start with a "17".)

```
        ; Trap service routine to handle FPU instructions

        SUB  #2, @R6           ; decrement saved PC
        CMP #1700000,@0 (R6)   ; test for FPU OP code
        BLO    NOTFPU
        MOV  @R6, -(R6)        ; make 3 top words of stack
        MOV  4(R6), 2(R6)      ; FPU, PS and PC
        MOV   R6,  4(R6)
        MOV  #FPU, @R6
        RTI                    ; end of FPU trap handler

NOT FPU:   ADD #2, @R6
```

2.1.4 INTERRUPTABILITY

A special deadlock condition can arise when the CPU is executing FPU supplied instructions and an interrupt occurs by a device which also wants to make use of the FPU. At the time the CPU was executing FPU supplied instructions, it was considered "busy". The CPU is interruptable at that point because it is running at a priority level lower than 7. If the interrupting device would go off and use the FPU without testing, the CPU would start an infinite loop of "BR ." instructions because the FPU was busy.

This loop is executed at the priority level of the interrupting device. In order for the FPU to become free, it has to continue supplying CPU instructions until subsequence D of Figure 2-1 has been completed, i.e. when the FPU dismisses the CPU.

A special hardware aid is built into the FPU to discover this state. The FPU has a register called the Floating Interrupt Vector "FINTV" and a bit called the Floating Interrupt CPU Dismissed "FICD" in the Floating Program Status "FPS" word. The FICD bit is set when ever a non-zero value is loaded into FINTV. The operation is as follows: Whenever the subsequence D of Figure 2-1 is executed, and the FICD bit of the FPS is set, the FPU will cause an interrupt using as interrupt vector (FINTV).

A possible routine preventing the deadlock making use of the above hardware, is shown on the next page. This code is part of the interrupt service routine of the interrupting device which wants to use the FPU.

```
        CMP  R6, #172           ;  did PC point to FPU
        BLO  FPUFREE

        MOV  FINTV,  TEMP       ;  save old FINTV
        MOV  NEW.INTV, FINTV ;  set up new interrupt vector

       *MOV  2(R6), TEMP1       ;  save old PS
       *MOV  NEW.PS,2(R6)       ;  install new PS

        RTI                     ;  dismiss current interrupt
                                ;  and start FPU

FPU FREE:  SAVE FPU STATUS
           USE THE FPU

        MOVE TEMP, FINTV        ;  restore old FINTV

       *MOVE TEMP1, 2(R6)       ;  restore old PS

        RTI                     ;  dismiss interrupt
```

It should be noted that any interrupt vector can be loaded into FINTV. If, for example, the interrupt vector of the interrupting device is loaded into FINTV, then upon the first RTI in the above code, the interrupt will be dismissed until the FPU has dismissed the CPU. At that point, the FPU will request an interrupt with the interrupt vector of the original interrupting device, thus simulating the old interrupt.

2.2 FPU-11/05 INTERACTION

The use of the FPU with the 11/05 is essentially the same as with the 11/20 except for the JSR preceding an FPU instruction, which is not required with the 11/05. The 11/05 will execute code making use of the JSR, however, for compatibility reasons.

When the 11/05 fetches an instruction which starts with a "17" (i.e. an FPU OP code) it will not trap, but execute the following sequence.

```
        TST  FPU05              ;  test is FPU is busy
        BEQ  .-4                ;  loop is busy

        MOV  PC, FPU05+2
        MOV  IR, FPU05+4
        MOV  FPU05+6,  PC       ;  start fetching instructions from the FPU
```

The above sequence is not executed with PDP-11 instructions as shown above, but in 11/05 micro code which is done at a much greater speed. This allows the FPU instructions to be given without a JSR, thus eliminating the space and time consuming JSR and the Instruction Fetch subsequence "A" of Figure 2-1.

*These instructions are only necessary when the FPU has to proceed with the interrupted instruction at a different priority level.

2.3 FPU-11/40 INTERACTION

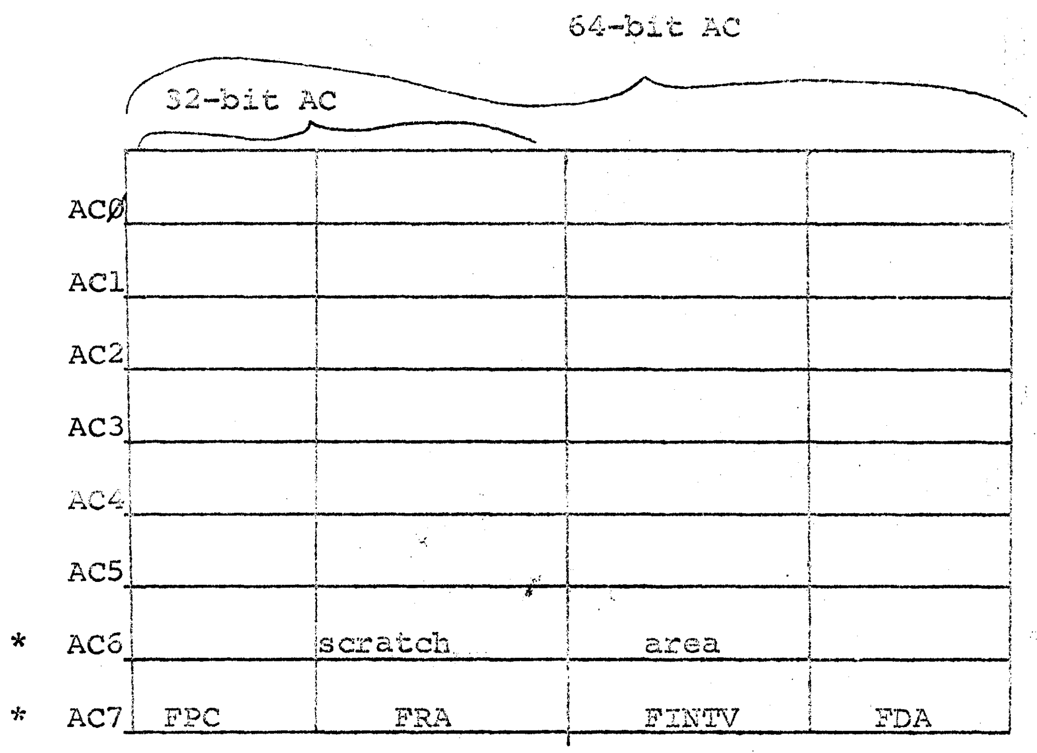
The FPU will be connected to the 11/40 via a direct set of wires rather than via the Unibus. This is required for the proper operation of the segmentation option, see Section 1.0.

The required address computation will be done by the 11/40 hardware. The need to have the FPU supply the 11/40 with the instructions is thereby eliminated. The 11/40 condition codes will not be affected by the FPU unless the instruction is a CFCC.

When the 11/40 fetches an FPU instruction, it tests if the FPU is busy while it allows for higher priority bus requests. Once the FPU is free, the 11/40 will do the required address computation and notify the FPU. The FPU will then strobe in the required data from the 11/40's internal registers (like the PC, IR, etc.), do the required data fetches (stores from) into memory and allows the 11/40 CPU to proceed while it executes the FPU instruction.

3.0 THE FPU's INSTRUCTION & DATA FORMATS

The FPU has, except for its scratch, address and status registers, 6 general purpose data registers called Accumulators "AC's". They are named ACØ through AC5 and are interpreted to be 32 or 64-bits long depending on the instruction. In case of a 32-bit instruction, only the top (i.e. left most) 32-bits are used, while the remaining (i.e. right 32-bits) of the AC remain unaffected. See Figure 3-1.



* AC6 and AC7 are reserved for internal use.

AC7 is used to contain the following status registers:

1) FPC "Floating PC" - points to the word following the first word of the FPU instruction.

2) FRA "Floating Return Address" - points to the next instruction to be executed.

3) FINTV "Floating Interrupt Vector" - a 16 bit interrupt vector used by the FPU upon completion of the address computation part of an instruction when (FINTV)≠0. This is only used on the 11/05 and 11/20.

4) FEC "Floating Exception Code" - A number which identifies the cause of the interrupt.

FIGURE 3-1 Accumulator Layout

The FPU instruction set is divided in five formats as shown in Figure 3-2. Format F1 is used by the binary floating instructions. Format F2 is used by the unary floating instructions. Format F3 is used by the load and store convert to and from Integer instructions. Format F5 is used by some special instructions like Copy Floating Condition Code.

The fields of the formats of Figure 3-2 are interpreted in the following way.

OC	"Operation Code" The OC field of all FPU instructions is 4 bits long and contains a "17".
FOC	"Floating Operation Code" This field of the format specifies the specific floating point operation.
FSRC	"Floating Source" The floating source specifies the source operand of the instruction. The interpretation of the addressing modes is as shown below:

MODE	INTERPRETATION
0	ACØ-AC5 contain the data. The "data" is considered 32 or 64 bits depending on the mode of the FPU (i.e. Floating or Extended). When AC6 or AC7 are specified, an OP code error will be given unless the instruction is a STX instruction.
1	RØ-R7 contain the address of the data. When R=R7 the data is considered to be only 1 word long (i.e. 16 bits).
2	RØ-R7 contain the address of the data. After the data has been fetched RØ-R6 are incremented with 4 or 8 depending on the mode of the FPU. When R=R7, the data is considered to be 1 word long and therefore, R7 will be incremented with 2.
3	RØ-R7 contains the address of the address of the data. RØ-R7 are incremented by 2.
4	RØ-R6 are decremented by 4 or 8, depending on the FPU mode. After that they contain the address of the data. When R=R7, R7 is decremented by 2 and contains the address of a 1 word data item.

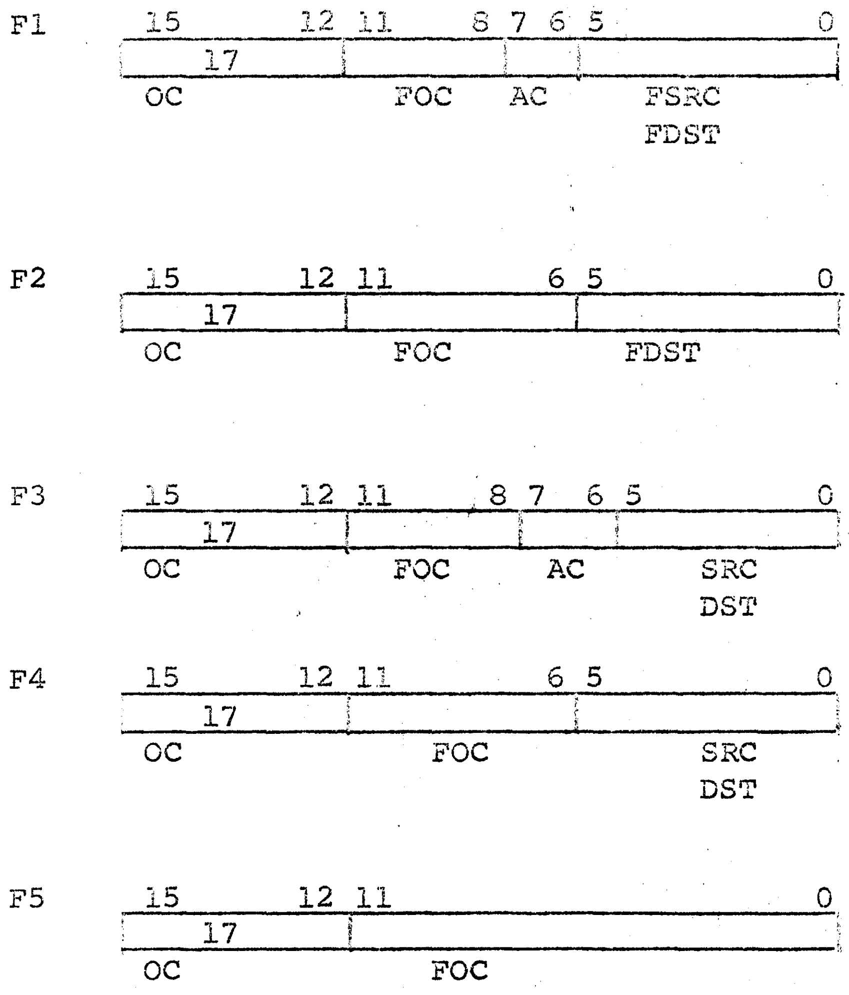
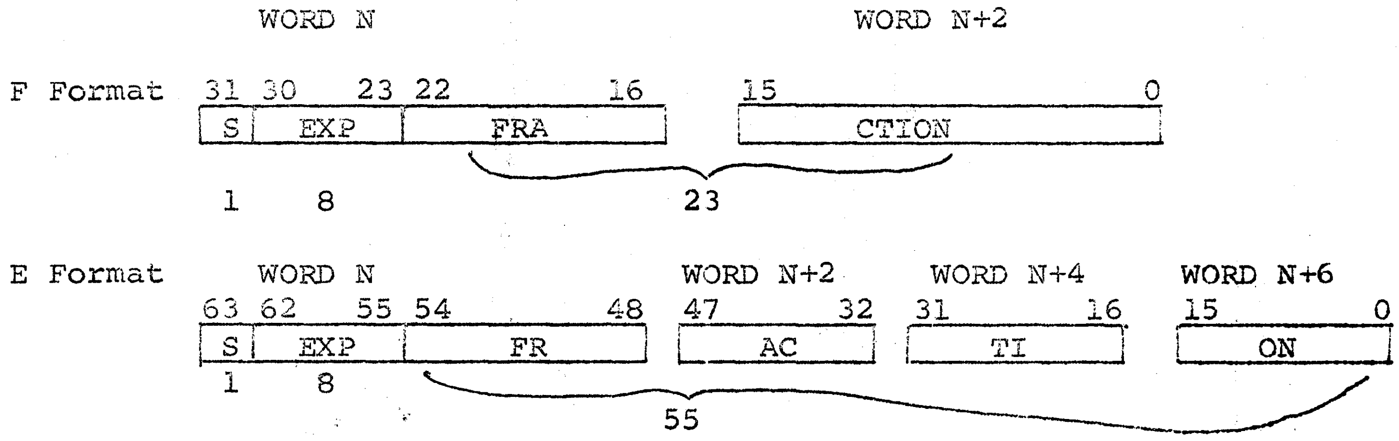


FIGURE 3-2. FPU INSTRUCTION FORMATS

MODE	
5	RØ-R7 are decremented by 2. After that they contain the address of the address of the data.
6	The address of the data is determined by the regular index computation.
7	The address of the data is determined by the regular deferred index computation.
FDST	"Floating Destination" The interpretation of this field is identical to that of the source.
AC	"Accumulator" This is a 2 bit field specifying ACØ-AC3.
SRC	"Source" Regular PDP-11 source field.
DST	"Destination" Regular PDP-11 destination field.

3.1 THE FPU'S DATA FORMATS

The FPU handles two types of floating point data: Floating "F" which is 32 bits long, and Extended "E" which is 64 bits long. Both formats assume normalized numbers only. The fraction is represented in sign-magnitude notation with the binary radix point to the left. The most significant bit of the fraction is not stored because it is redundant. This bit is always a 1 except when the exponent is 0, then the number is declared to be zero. The F and E format are shown in Figure 3-3 below.



S=Sign of fraction

EXP=8 bit exponent, in excess 200_8 notation, radix =2

FRACTION=23 or 55 bit fraction in sign-magnitude notation, radix point to the left

FIGURE 3-3 Floating Point Data Format

4 THE FPU'S INSTRUCTION SET

Appendix A lists the complete FPU instruction set, a description of which is given below. Appendix B lists some maximum and minimum execution times.

4.1 THE FPU PROGRAM STATUS REGISTER

The FPU's program status register in shown in Figure 4-1. It has four mode bits:

1) FT, the FPU's Truncate Mode Bit. This bit, when set, causes the result of any floating point operation to be truncated rather than rounded.

2) FD, the FPU's Double Precision Integer Mode Bit. This bit is active in conversion between integer and floating point format. When on, the integer format assumed is double precision 2's complement (i.e. 32 bits). When off, the integer format that is assumed is single precision 2' complement (i.e. 16 bits).

3) FE, the FPU's Extended Precision Mode Bit. This bit determines the precision that is used for floating point calculations. When set, extended precision is assumed - when reset, normal precision is used.

4) FMM, the FPU's Maintenance Mode Bit. The FMM enables special maintenance logic. The exact nature of this logic will be detailed in a later memo.

Along with the four mode bits, the status register contains four condition codes, FC, FV, FZ and FN. These are loaded into the CPU's C, V, Z, and N condition codes by the Copy Floating Condition Codes instruction.

The way in which each instruction affects the floating condition codes is detailed in the instruction definitions. The FC condition code bit has two meanings:

1) For the STCXJ 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.

2) In all other cases, the FC bit indicates that the absolute value of the floating point result was larger than the largest integer that can be represented in M bits, where M is the width of the fraction. In the

extended mode, M = 56 bits and in floating mode M = 24. This allows sign-magnitude integer arithmetic with 24 and 56 bits of precision, not including the sign bit, to be performed with the FPU.

The FPU's Program Status Register also contains six interrupt enable bits. The FPU interrupt vector is at core location 240_8.

1) FIC FLOATING INTERRUPT ON INTEGER CONVERSION ERROR

When FIC is set, and the STCXJ instruction causes FC to be set, a trap will occur. If the interrupt occurs, the instruction is aborted leaving the contents of all the registers untouched.

2) FIV FLOATING INTERRUPT ON OVERFLOW

When this bit is set, floating overflows will cause an interrupt. The result of the operation causing the interrupt will be correct except for the exponent which will be off by 400 (octal). If the bit is off, the result of the operation will be the same as detailed above and no interrupt will occur.

3) FIU FLOATING INTERRUPT ON UNDERFLOW

When this bit is on, floating underflow will cause an interrupt. The result of the operation, causing the interrupt, will be correct except for the exponent which will be off by 400 (octal). If the bit is off and underflow occurs, the result will be set to zero.

4) FIOR FLOATING INTERRUPT ON OUT OF RANGE

When this bit is on, and the FC bit is set because the result is out of integer range, an interrupt occurs. Out of integer range means that the absolute value of the result is greater than or equal to 2^{XL} where XL=24 if floating mode, or 56 if extended mode.

5) FIUV FLOATING INTERRUPT ON UNDEFINED

When this bit is on and a -Ø is obtained from memory, an interrupt will occur. When this bit is off -Ø can be loaded and used in any arithmetic operation. The result of such operation is undefined.

6) <u>FICD FLOATING INTERRUPT ON CPU DISMISSED</u>

The FICD bit, when on, will cause an interrupt to occur when the address computation performed by the 11/20 and 11/05 is done. On the 11/40 this bit will be ignored. For a complete description of the use of this enable, see Section 2.1.4.

7) <u>FIE FLOATING INTERRUPT ENABLE</u>

All interrupts by the FPU are disabled when this bit is off.

BIT		
0	FC	;Floating Carry
1	FV	;Floating Overflow
2	FZ	;Floating Zero
3	FN	;Floating Negative
4	FMM	;Floating Maintenance Mode
5	FT	;Floating Truncate Mode
6	FD	;Floating Double Precision Mode
7	FE	;Floating Extended Mode
8	FIC	;Floating Interrupt on Conversion Error
9	FIV	;Floating Interrupt on Overflow Error
10	FIU	;Floating Interrupt on Underflow Error
11	FIOR	;Floating Interrupt on Out of Range Error
12	FIUV	;Floating Interrupt on Undefined Variable
13	FICD	;Floating Interrupt on CPU Dismissed
14	FIE	;Floating Interrupt Enable
15	RUN	;FPU's Run Status

FIGURE 4-1. Layout of FPU Program Status Register

INSTRUCTION: Set Floating Mode
MNEMONIC: SET F
OPERATION: FE ← Ø
FORMAT:

1	7	0	0	0	1

INSTRUCTION: Set Extended Mode
MNEMONIC: SETE
OPERATION: FE ← 1
FORMAT:

1	7	0	0	0	2

INSTRUCTION: Integerize Floating/Extended
MNEMONIC: INTX FSRC
OPERATION: AC4 ← $\mathcal{J}$(FSRC) ; AC5 ← (FSRC) − $\mathcal{J}$(FSRC)
FC ← 1 if $|$FSRC$| \geq 2^{XL}$ else FC ← Ø*
FV ← Ø
FZ ← 1 if (FSRC) = Ø else FZ ← Ø
FN ← 1 if (FSRC) < Ø else FN ← Ø
FORMAT:

1	7	0	3	FSRC

$\mathcal{J}$(FSRC) is the integer part of (FSRC) i.e. (FSRC) is fixed and then floated. Note that the integer is obtained by truncation i.e. 5.9 becomes 5. If $|$FSRC$| \geq 2^{XL}$, $\mathcal{J}$(FSRC) = (FSRC). Note that the fractional part of (FSRC) is stored in AC5.

INSTRUCTION: Clear Floating/Extended
MNEMONIC: CLRX FDST
OPERATION: FDST ← Ø
FC ← 0
FV ← 0
FZ ← 1
FN ← 0
FORMAT:

1	7	0	4	FDST

* XL = 24 if FE mode = Ø
= 56 if FE mode = 1

INSTRUCTION: Negate Floating/Extended
MNEMONIC: NEGX FDST
OPERATION: FDST $\leftarrow -(\text{FDST})$
FC $\leftarrow$ 1 if $|(\text{FDST})| \geq 2^{XL}$ else FC $\leftarrow$ 0*
FV $\leftarrow$ Ø
FZ $\leftarrow$ 1 if (FDST) = Ø else FZ $\leftarrow$ Ø
FN $\leftarrow$ 1 if (FDST) < Ø else FN $\leftarrow$ Ø
FORMAT:

1	7	0	5	FDST

INSTRUCTION: Make Absolute Floating/Extended
MNEMONIC: ABSX FDST
OPERATION: FDST $\leftarrow -(\text{FDST})$ if (FDST) < 0 else FDST $\leftarrow$ (FDST)
FC $\leftarrow$ 1 if $|(\text{FDST})| \geq 2^{XL}$ else FC $\leftarrow$ Ø*
FV $\leftarrow$ 0
FZ $\leftarrow$ 1 if (FDST) = 0 else FZ $\leftarrow$ Ø
FN $\leftarrow$ Ø
FORMAT:

1	7	0	6	FDST

INSTRUCTION: Test Floating/Extended
MNEMONIC: TSTX FDST
OPERATION: FDST $\leftarrow$ (FDST)
FC $\leftarrow$ 1 if $|(\text{FDST})| \geq 2^{XL}$ else FC $\leftarrow$ 0*
FV $\leftarrow$ 0
FZ $\leftarrow$ 1 if (FDST) = Ø else FZ $\leftarrow$ Ø
FN $\leftarrow$ 1 if (FDST) < 0 else FN $\leftarrow$ 0
FORMAT:

1	7	0	7	FDST

INSTRUCTION: Load Floating/Extended
MNEMONIC: LDX FSRC,AC
OPERATION: AC $\leftarrow$ (FSRC)
FC $\leftarrow$ 1 if $|(\text{FSRC})| \geq 2^{XL}$ else FC $\leftarrow$ 0*
FV $\leftarrow$ 0
FZ $\leftarrow$ 1 if (FSRC) = Ø else FZ $\leftarrow$ Ø
FN $\leftarrow$ 1 if (FSRC) < 0 else FN $\leftarrow$ Ø
FORMAT:

1	7	1	AC	FSRC

*XL=24 if FE mode =1
56 if FE mode=1

INSTRUCTION:	Store Floating/Extended
MNEMONIC:	STX AC, FDST
OPERATION:	FDST$\leftarrow$(AC)
	FC$\leftarrow$FC
	FV$\leftarrow$FV
	FZ$\leftarrow$FZ
	FN$\leftarrow$FN

FORMAT:

1	7	1	4+AC	FDST

INSTRUCTION: Add Floating/Extended

MNEMONIC: ADDX FSRC, AC

OPERATION:

```
AC ← (AC) + (FSRC) if |(AC) + (FSRC)| ≥ XLL OR FIU=1
        else AC ← Ø**
FC ← 1 if |(AC)| ≥ $2^{XL}$ else FC ← Ø*
FV ← 1 if |(AC)| > XUL else FV ← Ø***
FZ ← 1 if (AC) = Ø else FZ ← Ø
FN ← 1 if (AC) < Ø else FN ← 1
```

FORMAT:

1	7	2	AC	FSRC

INSTRUCTION: Subtract Floating/Extended

MNEMONIC: SUBX FSRC, AC

OPERATION:

```
AC ← (AC) - (FSRC) if |(AC) - (FSRC)| ≥ XLL OR FIU=1
        else AC ← Ø**
FC ← 1 if |(AC)| ≥ $2^{XL}$ else FC ← Ø*
FV ← 1 if |(AC)| > XUL else FV ← 0***
FZ ← 1 if (AC)=Ø else FZ ← Ø
FN ← 1 if (AC)<Ø else FN ← 1
```

FORMAT:

1	7	2	4+AC	FSRC

* XL = 24 if FE=Ø
 = 56 if FE=1

**XLL = smallest number that is not identically zero
 $=2^{-128}$

***XUL = largest number that can be represented
 $=2^{127}*(1-2^{-XL-1})$

INSTRUCTION: Multiply Floating/Extended

MNEMONIC: MULX FSRC, AC

OPERATION:

```
AC ← (AC)*(FSRC) if |(AC)*(FSRC)| ≯ XLL OR FIU=1
     else AC ← Ø**
FC ← 1 if |(AC)| ≯ 2^XL else FC ← Ø*
FV ← 1 if |(AC)| > XUL else FV ← Ø***
FZ ← 1 if (AC) = Ø else FZ ← Ø
FN ← 1 if (AC) < Ø else FN ← Ø
```

FORMAT:

1	7	3	AC	FSRC

INSTRUCTION: Divide Floating/Extended

MNEMONIC: DIVX FSRC,AC

OPERATION:

```
Case 1 (FSRC) ≠ Ø
AC ← (AC)/(FSRC) if |(AC)/(FSRC)| ≯ XLL OR FIU=1
     else AC ← Ø**
FC ← 1 if |(AC)| ≯ 2^XL else FC ← Ø*
FV ← 1 if |(AC)| > XUL else FV ← Ø***
FZ ← 1 if (AC)=Ø else FZ ← Ø
FN ← 1 if (AC) < Ø else FN ← Ø
Case 2 (FSRC) = Ø
AC ← (AC)
FC ← (FC)
FV ← (FV)
FZ ← (FZ)
FN ← (FN)
```

FORMAT:

1	7	3	4+AC	FSRC

INSTRUCTION: Reverse Subtract Floating/Extended

MNEMONIC: RSUBX FSRC,AC

OPERATION:

$AC \longleftarrow (FSRC)-(AC)$ if $|(FSRC)-(AC)| \geq XLL$ OR
    FIU=1 else $AC \longleftarrow Ø$**
$FC \longleftarrow 1$ if $|(AC)| \geq 2^{XL}$ else $FC \leftarrow Ø$*
$FV \longleftarrow 1$ if $|(AC)| > XUL$ else $FV \leftarrow Ø$***
$FZ \longleftarrow 1$ if $(AC)=Ø$ else $FZ \leftarrow Ø$

FORMAT:

1	7	4	AC	FSRC

INSTRUCTION: Compare Floating/Extended

MNEMONIC: CMPX AC, FDST

OPERATION:

$AC \longleftarrow (AC)$
$FC \longleftarrow 1$ if $|(AC)| \geq 2^{XL}$ else $FC \leftarrow Ø$*
$FV \longleftarrow 1$ if $|(AC)| > XUL$ else $FV \leftarrow Ø$***
$FZ \longleftarrow 1$ if $(AC)=Ø$ else $FZ \longleftarrow Ø$
$FN \longleftarrow 1$ if $(AC) < Ø$ else $FN \longleftarrow Ø$

FORMAT:

1	7	4	4+AC	FDST

* XL = 24 if FE=Ø
     = 56 if FE=1

**XLL = smallest number that is not identically zero
     $= 2^{-128}$

***XUL = largest number that can be represented
     $= 2^{127} (1-2^{-XL-1})$

INSTRUCTION: Reverse Divide Floating/Extended

MNEMONIC: RDIVX FSRC,AC

OPERATION:

Case 1 (AC)=Ø

$AC \leftarrow (FSRC)/(AC)$ if $|(FSRC) / (AC)| \geq XLL$ OR FIV=1 else $AC \leftarrow Ø$**

$FC \leftarrow 1$ if $|(AC)| \geq 2^{XL}$ else $FC \leftarrow Ø$*

$FV \leftarrow 1$ if $|(AC)| > XUL$ else $FV \leftarrow Ø$***

$FZ \leftarrow 1$ if $(AC) = Ø$ else $FZ \leftarrow Ø$

$FN \leftarrow 1$ if $(AC) < Ø$ else $FN \leftarrow Ø$

Case 2 (AC)=0

$AC \leftarrow (AC)$

$FC \leftarrow (FC)$

$FV \leftarrow (FV)$

$FZ \leftarrow (FZ)$

$FN \leftarrow (FN)$

FORMAT:

1	7	5	AC	FSRC

INSTRUCTION: Load & convert from Extended Floating to Floating/Extended

MNEMONIC: LDCYX FSRC,AC

OPERATION:

$AC \leftarrow C_{YX}(FSRC)$ if $|(FSRC)| \geq XLL$ or FIU=1 else $AC \leftarrow Ø$**

$FC \leftarrow 1$ if $|(AC)| \geq 2^{XL}$ else $FC \leftarrow Ø$*

$FV \leftarrow 1$ if $|(AC)| > XUL$ else $FV \leftarrow Ø$***

$FZ \leftarrow 1$ if $(AC)=Ø$ else $FZ \leftarrow Ø$

$FN \leftarrow 1$ if $(AC) < Ø$ else $FN \leftarrow Ø$

FORMAT:

1	7	5	4+AC	FSRC

* XL = 24 if FE=Ø
 = 56 if FE=1

**XLL = smallest number that is not identically zero
 $= 2^{-128}$

***XUL = largest number that can be represented
 $= 2^{127}(1-2^{-XL-1})$

C_{YX}(FSRC) is defined as (FSRC) converted from Y mode (Y=-X) to the current mode, i.e. Floating or Extended. The source is assumed to be opposite to the current mode. Specifically, if the current mode is F and the FT bit is set, then FSRC ⟨63:32⟩ are loaded into AC ⟨31:0⟩. If the FT bit is zero, the result is rounded using FSRC ⟨31⟩. Note that FSRC ⟨31:0⟩ is unaffected. If the current mode is E, AC ⟨63:32⟩ are loaded from FSRC ⟨31:0⟩ and AC ⟨31:Ø⟩ are cleared. Similarly, C_{XY} (FSRC) converts (FSRC) from X to -X mode by truncating or rounding (FT=1 or Ø when X=E or loading trailing zeros if X=F.

INSTRUCTION:	Store & Convert from Floating/Extended to Extended/Floating
MNEMONIC:	STCXY AC, FDST

OPERATION:

```
FDST ← $C_{XY}$ (AC) if |$C_{XY}$(AC)| ≥ XLL or FIU=1
else FDST ← Ø**
FC ← 1  if |(AC)| ≥ $2^{XL}$ else FC ← Ø*
FV ← 1  if |(AC)| > XUL else FV ← Ø***
FZ ← 1  if (AC)=Ø else FZ ← Ø
FN ← 1  if (AC) < Ø else FN ← Ø
```

FORMAT:

1	7	6	AC	FDST

* XL = 24 if FE = Ø
= 56 if FE = 1

**XLL = smallest number that is not identically zero
= 2^{-128}

***XUL = largest number that can be represented
= $2^{127}(1-2^{-XL-1})$

INSTRUCTION: Load & Convert Integer/Double to Floating/Extended

MNEMONIC: LDCJX SRC,AC

OPERATION:

$AC \leftarrow C_{JX}\ (SRC)$
$FC \leftarrow 1$ if $|C_{JX}\ (SRC)| \not\geq 2^{XL}$ else $FC \leftarrow Ø$**
$FV \leftarrow Ø$
$FZ \leftarrow 1$ if $(AC)=Ø$ else $FZ \leftarrow Ø$
$FN \leftarrow 1$ if $(AC) < Ø$ else $FN \leftarrow Ø$

FORMAT:

1	7	6	4 +AC	SRC

C_{JX}(SRC) specifies a conversion from an integer with precision specified by J to a floating point number with precision specified by X, i.e. if J=I and X=F the source is assumed to be a 16-bit 2's complement integer which is converted to a sign magnitude floating point number with a 24 bit fraction. In the case of C_{DF} (SRC), the fraction is truncated, i.e, only the highest 24 significant digits are used.

INSTRUCTION: Store Converted from Floating/Extended to Integer/Double

MNEMONIC: STCXJ AC, DST

OPERATION:

$DST \leftarrow C_{XJ}(AC)$ if $-2^{JL} \leq C_{XJ}(AC) \leq 2^{JL}-1$ else $DST \leftarrow (DST)$*
$FC \leftarrow 1$ if $-2^{JL} > C_{XJ}(AC) > 2^{JL}-1$ else $FC \leftarrow Ø$ *
$FV \leftarrow Ø$
$FZ \leftarrow 1$ if $(DST) = Ø$ else $FZ \leftarrow Ø$
$FN \leftarrow 1$ if $(DST) < Ø$ else $FN \leftarrow Ø$

FORMAT:

1	7	7	AC	DST

*JL=15 if FD mode = Ø
   =31 if FD mode = 1

**XL=24 if FE=Ø
   =56 if FE=1

INSTRUCTION:	Load FPU's Program Status
MNEMONIC:	LDFPS SRC
OPERATION:	FPS$\leftarrow$(SRC)
FORMAT:	

1	7	7	4	SRC

INSTRUCTION:	Store FPU's Program Status
MNEMONIC:	STFPS DST
OPERATION:	DST$\leftarrow$(FPS)
FORMAT:	

1	7	7	5	DST

INSTRUCTION:	Store FPU's Exception Code
MNEMONIC:	STFEC DST
OPERATION:	DST$\leftarrow$(FEC)*2
FORMAT:	

1	7	7	6	DST

INSTRUCTION:	LOAD Maintenance Counter
MNEMONIC:	LDMC
OPERATION:	MC $\longleftarrow$ (RØ)

FORMAT:

1	7	Ø	Ø	1	Ø

The ROM cycle counter (RCC) decrements each ROM cycle. In maintenance mode the next ROM word will not be fetched if the RCC=Ø.

INSTRUCTION:	Store A register in ACØ
MNEMONIC:	STAØ
OPERATION:	ACØ $\longleftarrow$ (AR)

FORMAT:

1	7	Ø	Ø	1	1

INSTRUCTION:	Store B register in ACØ
MNEMONIC:	STBØ
OPERATION:	ACØ $\longleftarrow$ (BR)

FORMAT:

1	7	Ø	Ø	1	2

INSTRUCTION: Store Q register in ACØ

MNEMONIC: STQØ

OPERATION: $BR \leftarrow (QR)$
$ACØ \leftarrow (BR)$

FORMAT:

1	7	Ø	Ø	1	3

0338

APPENDIX A

SUMMARY OF FPU INSTRUCTION SET

INSTRUCTION	MNEMONIC	OP CODE	DESCRIPTION
Copy Floating Condition Codes	CFCC	170000	CC ← FCC
Set Floating Mode	SETF	170001	FE←0
Set Extended Mode	SETE	170002	FE←1
Load Maintenance Counter	LDMC	170010	MC ←(R0)
Store AR Register in ACØ	STAØ	170011	ACØ ←(AR)
Store BR Register in ACØ	STBØ	170012	ACØ ←(BR)

Store QR Register ACØ	STQØ	170013	BR ← (QR)
			ACØ ←(BR)

Integerize Floating/ Extended	INTX FSRC)	170300+FSRC	AC4 ←integer part of (FSRC); AC5←fractional part of (FSRC)
Clear Floating/ Extended	CLRX FDST	170400+FDST	FDST ←Ø

APPENDIX A (continued)

INSTRUCTION	MNEMONIC	OP CODE	DESCRIPTION
Negate Floating/ Extended	NEGX FDST	170500+FDST	FDST $\leftarrow$ -(FDST)
Make Absolute Floating/Extended	ABSX FDST	170600 +FDST	FDST $\leftarrow$ |(FDST)|
Test Floating/ Extended	TSTX FDST	170700+FDST	FCC$\leftarrow$condition of (FDST)
Load Floating/ Extended	LDX FSRC,AC	171000+AC*100+FSRC	AC $\leftarrow$(FSRC)
Store Floating/ Extended	STX AC,FDST	171400+AC*100+FDST	FDST $\leftarrow$AC
Add Floating/ Extended	ADDX FSRC,AC	172000+AC*100+FSRC	AC $\leftarrow$(AC)+(FSRC)
Subtract Floating/ Extended	SUBX FSRC,AC	172300+AC*100+FSRC	AC $\leftarrow$(AC)-(FSRC)
Multiply Floating/ Extended	MULX FSRC,AC	173000+AC*100+FSRC	AC $\leftarrow$(AC)*(FSRC)
Divide Floating/ Extended	DIVX FSRC,AC	173400+AC*100+FSRC	AC $\leftarrow$(AC)/(FSRC)
Reverse Subtract Floating/Extended	RSUBX FSRC,AC	174000+AC*100+FSRC	AC $\leftarrow$(FSRC)-(AC)
Compare Floating/ Extended	CMPX AC,FDST	174400+AC*100+FSRC	FCC$\leftarrow$condition of (FDST)-(AC)

APPENDIX A (continued)

INSTRUCTION	MNEMONIC	OP CODE	DESCRIPTION
Reverse Divide Floating/Extended	RDIVX FSRC,AC	175000+AC*100+FSRC	AC ⟵ (FSRC)/(AC)
Load & Convert from Extended/Floating to Floating/Extended	LDCYX FSRC,AC	175400+AC*100+FSRC	AC ⟵ converted (FSRC)
Store & Convert from Floating/Extended to Extended/Floating	STCXY AC,FDST	176000+AC*100+FDST	FDST ⟵ converted (AC)
Load & Convert Integer/Double to Floating/Extended	LDCJX SRC,AC	176400+AC*100+SRC	AC ⟵ converted (SRC)
Store & Convert Floating/Extended to Integer/Double	STCXJ AC,DST	177000+AC*100+SRC	DST ⟵ converted (AC)
Load FPU's Program Status	LDFPS SRC	177400+SRC	FPS⟵(SRC)
Store FPU's Program Status	STFPS DST	177500+DST	DST ⟵FPS
Store FPU's Inception Code	STFEC DST	177600+DST	DST ⟵(FEC)*2

APPENDIX B
FLOATING POINT EXECUTION TIMES

An initial analysis of our floating point algorithms resulted in the execution times of the table below. These times apply to AC to AC operations.

The following approximation can be used to find the execution time for memory referencing operations: Take time of table below and add to it (0.25 μsec + memory access/cycle time) * number of memory references. This time has to be corrected further for possible memory cycles for the address computation (e.g. add 1 memory access time for mode 6 "A(R)").

FLOATING POINT EXECUTION TIMES FOR AC-AC OPERATIONS

INSTRUCTION	EXECUTION TIME IN μSEC			
	SINGLE PRECISION		EXTENDED PRECISION	
	MIN.	MAX.	MIN.	MAX.
ADDX	1.8	3.5	2.4	5.1
SUBX	1.8	3.5	2.4	5.1
MULX	2.7	5.5	4.8	10.0
DIVX	3.0	6.0	5.0	12.0