The 80x87 Family

Contents

When we speak of the 80x87 mathematical co-processor family, we include the original Intel chips, the Cyrix D387 and IIT C287 and C387 chips, and the AMD 80287 and 80387 clones, as well as the on-chip floating point unit found on the Intel 80486 chips.

We now describe some features of the 80x87 floating point co-processors (FPUs) that affect scientific programming on IBM-PC compatible machines[1]. The 8087 chip complements the Intel 8088/8086 CPU family, the 80287 works with the 80286 series, the 80387 with the 80386, and the 80486 includes an on-chip FPU. The 8087 chip is connected pin-for-pin to the 8088/8086. (The details are given in *The 8087 Primer*[2].) The interface and instruc-

1. Although we confine ourselves to the 80x87 chip, the Motorola 68881/2 coprocessors can be programmed in the same general manner to achieve floating point capabilities rivalling VAX® minicomputers.
2. J. Palmer and S. Morse, *The 8087 Primer* (John Wiley & Sons, NY (1984), hereafter referred to as 8087P).

tion set automatically take care of bus arbitration (that is, which chip has access to the memory) and interrupts, in order to be sure that the CPU and 80x87 do not perform conflicting operations.

Instructions for the 80x87 are always appended to a code called "escape" (ESC, D8h) that alerts the coprocessor and diverts control to it. The MS-DOS assembler MASM and debugger DEBUG will automatically assemble this code with 80x87 instructions, so the user does not need to worry about including ESC except to be aware that it is happening. (Of course, a FORTH system that lacks 80x87 assembler extensions will need to include ESC explicitly to generate 80x87 codes.)

We shall see in Ch. 4 §1.1 how to use the FORTH assembler, and in Ch. 4 §1.2 how to use DEBUG[3], to extend a FORTH system for 80x87 operations if they are not already included as a floating point lexicon.

The 80x87 machine code instruction set includes instructions for moving numbers to the registers from memory and *vice-versa*, as well as from one 80x87 register to another. The internal moves are of course much faster than those to or from external memory.

Advanced programming methods —such as recursive algorithms— require an fstack of unlimited depth. The designers of the 8087 anticipated the need for fstack extension and included instructions for this purpose. Unfortunately, the instructions were not well thought out[4] so a moderately complex software fstack extension manager is needed to augment them. We design such a manager in Ch. 4 §7.

3. A treasure included with MS-DOS
4. see **8087P**, p. 93ff.

§1 Internal 80x87 stack manipulation

The 80x87 is organized around a stack of 8 80-bit registers (the 87stack). The 8-deep stack can be subdivided into smaller stacks for special purposes, but this is only useful when coding in assembler for speed[5].

We begin with words for performing 80x87 stack manipulation analogous to those defined for parameter stack. These are **FDUP FSWAP FDROP FROT FOVER .**

How are we to define them in terms of machine code primitives?

§§1 The FORTH assembler

Every FORTH worthy of the name includes an assembler, usually set up as an alternate **vocabulary**[6] called **ASSEMBLER**. The assembler allows direct definition of a new word in terms of machine codes, which are referred to using standard mnemonics. A typical FORTH assembly language definition (we now specialize to HS/FORTH) for @ would have the form[7]

```
CODE   @   BX [BX] MOV. END-CODE
```

A **CODE** definition is a machine-coded subroutine somewhere in memory. To use it, the compiler has to know where it is and insert appropriate unconditional JUMP (JMP) and RETURN (RET) instructions in the machine code representation of the calling program.

Here is what happened in the **CODE** version of @ above:

- The defining word **CODE** set up a dictionary entry with the name @, with an appropriate pointer and JUMP instructions to make the newly defined word run the code sequence comprising the definition.

5. see **8087P**, p. 87ff.
6. Vocabularies are a method for subdividing the dictionary.
7. BX is a CPU register, and [BX] means "the memory location whose address is in BX". HS/FORTH uses a naming convention in which assembler mnemonics end with a period, *e.g.* **MOV.** . Also, HS/FORTH makes the TOS the BX register, to reduce the number of pushes and pops needed to execute simple words.

- The word **END-CODE** cleans up the loose ends by adding the obligatory RET instruction sequence and turning off the compiler. (That is, **END-CODE** installs "**NEXT**".)
- **END-CODE** is thus the analog of **;** as **CODE** is of **:**.

Consider now the word **FROT** whose Intel assembly language definition would be

```
FROT:          ; entry point
FWAIT          ; hold 8086 operations
FXCH ST(1)     ; swap TOS and NOS
FWAIT          ; hold 8086 operations
FXCH ST(2)     ; swap TOS and ST(2)
RET            ; return
```

In HS/FORTH assembler, the definition becomes

```
CODE  FROT   FWAIT.  1 FXCH.  FWAIT.  2 FXCH.
   END-CODE
```

Note: the definition includes FWAIT (the same as WAIT), an instruction that makes the 8086 CPU wait for the FPU to complete its work before attempting to access the memory. If WAIT were omitted and the CPU accessed the memory, it could store incomplete results[8].

§§2 Using MS-DOS DEBUG

The FORTH assembler, with its 80x87 extension, lets us develop machine-coded words while retaining Intel mnemonics for documentation, at the price of loading and compiling the entire **ASSEMBLER** lexicon. But if we know the actual machine code bytes we can bypass the assembler by entering the (hexadecimal) codes directly into a **CODE** definition. Most FORTH

8. The design of the 80286, 80386 and 80486 eliminates this problem, consequently FWAIT is not required when assembling 80287/80387/80487 machine code. See, *e.g.*, John H. Crawford and Patrick P. Gelsinger, *Programming the 80386* (Sybex, San Francisco, 1987). HS/FORTH allows the user to choose which class of machine to assemble for, when loading the 80x87 assembler extension. The 80287 + option simply defines **FWAIT.** as a null word.

systems, in addition to an assembler, provide a way to insert machine codes – hex numbers – directly into the code field of a word. HS/FORTH, *e.g.*, uses the words **<%** and **%>** to enclose the hex codes being inserted.

The problem is, how do we find out what these hex codes are?

The simplest way is to use DEBUG[9] to generate (HEX) code sequences. Rather than try to explain, we shall illustrate by recording and annotating the DEBUG session for the word **FROT**:

```
C>DEBUG                            starts DEBUG
-A100                              Assemble from 100h
3B01:0100 FWAIT                    enter asembler
3B01:0101 FXCH ST(1)               mnemonics
3B01:0103 FWAIT
3B01:0104 FEXCH ST(2)
          ^ Error                  DEBUG notes a typo
 3B01:0104 FXCH ST(2)
 3B01:0106                   no more, <cr>stops assembly

-U 100 105                         Unassemble to check
3B01:0100 9B          WAIT         Hold CPU
3B01:0101 D9C9        FXCH       ST(1)
                                   (87: a b c -- a c b)
3B01:0103 9B          WAIT         Hold up CPU
3B01:0104 D9CA        FXCH       ST(2)
                                   (87: a c b - - b c a)

-D 100 105                         Dump to get hex codes
3B01:0100  9B D9 C9 9B D9 CA
-Q                                 Quit session
```

From the Dump (or Unsassembly) we find code bytes 9B D9 F7 D9 C9 F6 D9 C9 which can be inserted directly into the definition of **FROT**:

9. see, *e.g.*, R. Lafore, *Assembly Language Primer for the IBM PC & XT* (Plume/Waite -New American Library, New York, 1984). Complete operating systems often include a code debugger that permits assembly; disassembly; modifying the contents of selected memory locations; setting breakpoints; and running programs under debugger control.

```
CODE  FROT <% 9B D9 F7 D9 C9 F6 D9 C9 %>
   END-CODE   ( :: a b c - - b c a )
```

The rest of the 80x87 stack words,

```
FDUP FSWAP FDROP FOVER F-ROT
```

whose assembler definitions are:

```
CODE    FDUP        FWAIT .  0   FLD.   END-CODE
CODE    FSWAP       FWAIT.   1  FXCH.   END-CODE
CODE    FDROP       FWAIT.   0  FSTP.   END-CODE
CODE    F-ROT    FWAIT.   2  FXCH.
        FWAIT.   1  FXCH.               END-CODE
```

can be defined similarly in 8086/8087 machine code using the DEBUG program or a reference manual for the chip (*e.g.* **8087P**) to determine the hex codes.

§2 Memory usage (storage and retrieval)

The 80x87 instruction set includes codes for loading ST(0) from memory and storing ST(0) to memory. The former involves a "push" and the latter may or may not involve a "pop", from the 87stack.

For the moment we need words to retrieve and store 16-bit integers, short reals (32 bit) and temporary reals (80 bit). These have mnemonics FILD ("load integer to ST(0)"), FISTP ("store integer and pop ST(1) into ST(0)"); FLD, FSTP respectively. Typical (HS/)FORTH assembler definitions are

```
CODE   I16@
   FWAIT.   [BX] WORD-PTR  FILD.
   [BX] POP.
   FWAIT.
END-CODE
CODE  I16!
   FWAIT.  [BX] WORD-PTR   FISTP.
   [BX] POP.
   FWAIT.
END-CODE
```

Similarly, **I32@** and **I32!** can be defined by replacing **WORD-PTR** with **DWORD-PTR**. To define **I64@** and **I64!** -assuming these are needed- replace **DWORD-PTR** with **QWORD-PTR**. The 32-, 64- and 80-bit floating point analogues **R32@**, **R32!**, **R64@**, **R64!**, **R80@** and **R80!** are defined by replacing **FILD** by **FLD** and **FISTP** by **FSTP**, and using **DWORD-PTR**, **QWORD-PTR** or **TBYTE-PTR**[10] as appropriate.

We also need words to load the 87stack from the stack and *vice versa*. In HS/FORTH, the top of the parameter stack is actually the BX register on the CPU. There is no machine instruction for loading the 87stack directly from a CPU register. Thus, we must first transfer the contents of BX to memory and thence to the 87stack. The inverse operation also must proceed through a memory location. The data-transfer words are named in an obvious way **S->F** and **F->S**. HS/FORTH defines them directly in machine code, manipulating the CPU register BP that points to the top of the CPU stack. That is, HS/FORTH uses two bytes immediately below NOS as the intermediate memory cell.

Here we define **S->F** and **F->S** directly in high-level FORTH by wasting a little memory for a (hidden) temporary variable:

```
VARIABLE TEMP
: S->F ( n --  :: -- float[n])
   TEMP  !            \ TOS -> TEMP
   TEMP  I16@  ;      \ TEMP -> ST(0)
: F->S ( :: x --  -- int[x])
   TEMP  I16!         \ ST(0) -> TEMP
   TEMP  @ ;          \ TEMP -> TOS
BEHEAD'  TEMP         \ hide address of TEMP
```

Faster machine code versions of **S->F** and **F->S** are[11]

```
CODE   S->F   TEMP  +[ ]  BX  MOV.  BX  POP.
   FWAIT.  TEMP  +[ ]  I16  FILD.   END-CODE

CODE   F->S    BX  PUSH.   TEMP  +[ ]  FISTP.
   BX  TEMP  +[ ]  MOV.   END-CODE
```

10. "Ten-byte pointer". Note HS/FORTH appends a period "." to most Intel mnemonics.
11. **TEMP +[ ]** is HS/FORTH's phrase to assemble a named memory address.

These definitions will satisfy our present needs for storing and retrieving from the 87stack.

<u>Note</u>: a substantial gain in speed can be achieved with the 80386/80387 and 80486 families, by using instructions that effect 32-bit wide transfers[12].

§3 Arithmetic words

FPU (80x87) arithmetic is generally performed with the maximum precision allowed by the (80-bit) size of the registers[13].

As noted in §4.2, the 80x87 allows 3 floating point representations for storage and retrieval in external memory: 32 bit (single precision), 64 bit (double precision) and 80 bit ("temporary real").

To conserve memory we generally use 32 bit floating point numbers (REAL*4 in the old FORTRAN parlance)unless the nature of the calculation demands retention of more significant figures to prevent roundoff errors.

The FORTH arithmetic words we shall need are

```
F+  F-  FR-  F*  F/  FR/  FNEGATE  FABS  FSGN
```

whose definitions are (CODE and END-CODE are assumed)

```
F+      FWAIT.      FADDP.      ( 87:  a b -- a+b)
F-      FWAIT.      FSUBP.      ( 87:  a b -- a-b)
FR-     FWAIT.      FSUBRP.     ( 87:  a b -- b-a)
F*      FWAIT.      FMULP.      ( 87:  a b -- a*b)
F/      FWAIT.      FDIVP.      ( 87:  a b -- a/b)
FR/     FWAIT.      FDIVRP.     ( 87:  a b -- b/a)
```

12. See, *e.g.*, John H. Crawford and Patrick P. Gelsinger, *Programming the 80386* (Sybex, San Francisco, 1987).

13. Although it *is* possible to force artificially 80x87 precision to 24 mantissa bits to simulate arithmetic performed on other machines (that is, to compare results while debugging), I see no virtue in a mode that slows up calculations while *diminishing* precision and refer the reader to refs. 2 or 8.

```
FNEGATE  FWAIT.     FCHS.       ( 87: a -- -a)
FABS     FWAIT.     FABS.       ( 87: a -- |a|)

: FSGN          ( n -- 87: x -- |x|*sgn[n] )
    FABS  0<  IF  FNEGATE  THEN  ;
```

§4 Special constants

For convenience the designers of the 8087 chip have arranged fast loading of certain constants into ST(0) of the fstack (TOS). The words that place these constants on the fstack, and the corresponding assembler mnemonics and (hex) codes are shown in Table 4-1 below.

Table 4-1 *Special Constants*

word	const.	mnemonic	codes
F=0	0	FLDZ	9B D9 E5
F=1	1	FLD1	9B D9 E8
F=PI	pi=3.14...	FLDPI	9B D9 E5
F=L2(10)	$\log_2 10$	FLDL2T	9B D9 E9
F=L2(E)	$\log_2 e$	FLDL2E	9B D9 EA
F=L10(2)	$\log_{10} 2$	FLDLG2	9B D9 EC
F=LN(2)	$\log_e 2$	FLDLN2	9B D9 ED

§5 Test words

We need to be able to determine the algebraic sign of a floating point number, as well as whether one is larger than another. The 80x87 chip has 4 instructions for this purpose, whose mnemonics are FTST, FCOM, FCOMP and FCOMPP; they are shown below in Table 4-2.

Table 4-2 *Machine language floating point tests*

mnemonic	comparison	pop?
FTST	ST(0) to 0	no
FCOM	ST(1) to ST(0)	no
FCOMP	ST(1) to ST(0)	pop once
FCOMPP	ST(1) to ST(0)	pop twice

The results of these comparisons are encoded as bits C3 (14) and C0 (8) of the 16-bit 80x87 STATUS register. In order to get these bits by bit-masking techniques, the STATUS register must be moved to the parameter stack[14]. This is done with the the 80x87 instruction FSTSW *via* the assembler sequence

```
VARIABLE  F.STATUS
CODE  FSTSW   BX  PUSH.
   F.STATUS  +[ ]   FSTSW.
   BX  F.STATUS  +[ ]  MOV.
END-CODE
BEHEAD'  F.STATUS
```

14. Note: the 80387 includes a new instruction whereby the status control word can be moved *directly* into the AX register of the 80386 CPU. The hex codes for FSTSW AX are DF E0.

Now we have to consider how to bit-mask the status integer left on the stack by **FSTSW**. We use the logical **AND** with 4000h or 0100h (Exercise: Why?) to pick out C3 and C0. Now from **8087P**[15] we have the truth table 4-3 below:

Table 4-3 *Truth table of test status bits (x is the operand – 0 or ST(1))*

Condition	C3	C0
ST(0) > x	0	0
ST(0) = x	1	0
ST(0) < x	0	1

Now we are in a position to define the test words **F0>**, **F0=**, **F0<**, as well as **F>**, **F=**, **F<**. We have[16]

```
CODE   FTST   FWAIT.  FTST.  0 FSTP.  END-CODE
CODE   FCOMPP   FWAIT.  FCOMPP.   END-CODE

HEX
: FTSTP   FTST   FSTSW   ;
: F0>     FTSTP  4100  AND  NOT  0> ;
: F0=     FTSTP  4000  AND  0 > ;
: F0<     FTSTP  0100  AND  0>  ;

: F<    FCOMPP   FSTSW   4100   AND  NOT  0> ;
: F=    FCOMPP   FSTSW   4000   AND  0> ;
: F>    FCOMPP   FSTSW   0100   AND  0> ;
DECIMAL
```

15. see, *e.g.*, Table 4.1
16. Note: the test words **F<** and **F>** are defined opposite to **F0>** and **F0<**. This reversal of directions is *not* a typographical error: it is *demanded* by the operation of **FCOMPP** – see **8087P**.

§6 Mathematical functions

We now proceed to develop a suite of special functions for the 80x87 FPU. These will include the usual trigonometric functions, logarithms, and exponentials. We retain initial **F**s in the names to remind us the FPUis being used. The functions are given in Table 4-4 below:

Table 4-4 *Mathematical function primitives*

name	action	code(s)	mnemonic
FSQRT	(87: x - - $\sqrt{x}$)	9B D9 FA	FWAIT FSQRT
FY*LG2X	(87: y x - - y*log2[x])	9B D9 F1	FWAIT FYL2X
FY*LG2XP1	(87: y x - - y*log2[x+1])	9B D9 F9	FWAIT FYL2XP1
F2XM1	(87: x - - $2^x - 1$)	9B D9 F0	FWAIT F2XM1

These primitive functions allow us to define the logarithms and exponentials. To get $\log_2(x)$, for example, we need to decide whether x lies between 0 and 2: this can best be accomplished with the sequence (we assume x is already on the 87stack)

```
: F=2  F=1 FDUP FSCALE FPLUCK ;  ( 87: - - 2)[17]

: LOG.TST   FDUP   F0>   NOT
   ABORT" Can't take log(-|x|) !!"  ;

: FLG          ( 87:  y x - -  y*lg[x] )
   LOG.TST   FDUP    F=2  F<
   IF        F=1  F-   FY*LG2XP1
   ELSE      FY*LG2X   THEN  ;
```

17. See below for a discussion of **FSCALE**.

```
: FLN    F=LN(2)    FSWAP  FLG ;
: FLOG   F=L10(2)   FSWAP  FLG ;
```

Now we can use the fundamental definition of exponentiation to define both the operation of raising an arbitrary positive number to a real power, as well as the standard mathematical function e^x:

```
:  F2**  ( 87: x -- 2**x )   F2XM1  F=1  F+ ;
:  F**   ( 87: x y -- y**x ) FLG    F2** ;
:  FEXP  ( 87: x -- exp[x] ) F=L2(E) F*  F2** ;
```

We now have only to implement the trigonometric and inverse-trigonometric functions. We need (TREAL) 10-byte constants:

```
: F,    HERE   10 ALLOT  R80!  ;
: FCONSTANT    CREATE  F,   DOES>  R80@  ;
```

Also, for simplicity, we define FORTH functions for degree/radian conversions and conversely:

```
FINIT   F=PI   180 S->F  F/      ( 87: -- p/180 )
FCONSTANT      PI/180            \ make constant
: DEG->RAD     PI/180   F* ;
: RAD->DEG     PI/180   F/ ;
```

The 80x87 chip has a fast way to multiply or divide by powers of 2, called **FSCALE**. The **CODE** definition is

```
CODE  FSCALE   <% 9B D9 FC % >  END-CODE
```

FSCALE adds ST(1) to the (powers-of-2) exponent of ST(0). Thus, *e.g.*, we can write fast divide- and multiply-by-2 instructions:

```
: F2*  F=1  FSWAP  FSCALE  FPLUCK  ;
: F2/  F=1  FNEGATE  FSWAP  FSCALE  FPLUCK ;
```

Now, according to **8087P**, we may evaluate trigonometric and inverse-trigonometric functions using the instructions

```
CODE  FPTAN   FWAIT.  <% D9 F2 %>  END-CODE
CODE  FPATAN  FWAIT.  <% D9 F3 %>  END-CODE
```

The 8087 and 80287 implement an ***unnormalized*** tangent function, whose effect is (87: z - - y x), with tan(z)=y/x. Let us define

$$\frac{y}{x} = \tan(z/2) \qquad (1)$$

That is, we obtain the tangent of half the angle. The other trigonometric functions can be computed in software using the identities

$$\sin(z) = \frac{2(y/x)}{1 + (y/x)^2} \qquad (2)$$

$$\cos(z) = \frac{1 - (y/x)^2}{1 + (y/x)^2} \qquad (3)$$

$$\tan(z) = \frac{\sin(z)}{\cos(z)} \qquad (4)$$

A further problem created by the 8087/80287 instruction set is that the argument z must lie in the range $0 < z < \pi/4$. Thus we must shift the argument to this range, using a special instruction **FPREM** ("exact" partial remainder[18]) that can be used to extract multiples of π :

```
CODE  FPREM  FWAIT.  <% D9 F8 %> END-CODE
: XDUP  FOVER  FOVER  ;
: ENUF?   FSTSW   1024 AND  0= ;     \ bit C2 =0?
: FNORM   ( 87:  x k - - x mod k )  FSWAP
    BEGIN   FPREM  ENUF?  UNTIL   FPLUCK  ;
    \ extract multiples of k
```

Here is how we code the tangent in high-level FORTH:

<u>**Pseudocode version**</u>:

```
x=0 is an exception  –  set tan = 0 and exit.
x <  0  ? Save sign as a flag on stack.
reduce by multiples of π .
\ cont'd ...
```

18. see 8087P, p. 100ff

x in 1st quadrant ($\pi/2 < x < \pi$) ? Flag, reduce by $\pi/2$
x in 1st octant ($\pi/4 < x < \pi/2$) ? Flag, reduce by $\pi/4$

```
\ HIGH LEVEL FORTH VERSION
: REDUCE        ( :87: x k --x mod k  --f )
   XDUP  F> DUP IF F-  ELSE  FDROP  THEN ;
: FTAN          ( 87: x - - tan[x] )  FDUP  F0=
   IF  EXIT  THEN                      \ tan=0
   FDUP  F0<  FABS                     ( - - fsgn 87: - - |x|)
   F=PI   FNORM                        \ 0 < x < π
   F=PI  F2/   REDUCE                  ( - - fsgn f1q )
   F=PI  F2/  F2/ REDUCE               ( - - fsgn f1q  f1o)
   FPTAN   F/                          ( 87: |x|- - tan[x] )
   IF  F=1  XDUP   F+  F-ROT  F-  F/  THEN
                                  \ adjust for octant
   IF  1/F  FNEGATE  THEN              \ adjust for quadrant
   IF  FNEGATE  THEN  ;                \ adjust sign
```

The remaining trigonometric functions (sine, cosine, secant, cosecant) can easily be defined in terms of tan(z /2) . For example, here are **FSIN** and **FCOS**:

```
: FSIN  F2/  FTAN  FDUP  FDUP  F*  F=1  F+
   FR/  F2*  ;
: FCOS  F2/  FTAN  FDUP  F*  F=1  FOVER  F-
   FSWAP  F=1  F+  F/  ;
```

Note: The 80387 improves on the 8087/80287 by eliminating the need to adjust the argument in software. Further, the tangent produced by the 80387 is normalized (that is, x = 1.0 in Eq. 4.1 above). Finally, the 80387 has instructions FSIN, FCOS and FSINCOS built in, so all the software emulation is unnecessary.

We define inverse-trigonometric functions using **FPATAN** defined previously, whose action is (87: y x - - arctan[y/x]). The 80387 has no additional instructions for inverse trig functions, relative to the 8087/80287, so the same code fits all.

To calculate the inverse functions, we make use of standard identities (the forms chosen minimize roundoff error). Thus,

$$\text{Arcsin}(z) = \text{Arctan}\left(\frac{z}{\sqrt{(1-z)(1+z)}}\right) \qquad (5)$$

$$\mathrm{Arccos}(z) = 2\,\mathrm{Arctan}\left\{\left(\frac{1-z}{1+z}\right)^{1/2}\right\} \qquad (6)$$

```
: FATAN   F=1  FPATAN  ;
: FASIN   FDUP  FABS  F=1   F>
   ABORT" argument of arcsin > 1"
   F=1   FOVER  FDUP  F*  F-   FSQRT
   F/  FPATAN  ;

: FACOS   FDUP  FABS  F=1   F>
   ABORT" argument of arccos > 1"
   FDUP  F=1    FR-   FSWAP   F=1  F+  F/
    FSQRT   FPATAN  ;
```

Note: the argument of arcsin(x) or arccos(x) must be smaller than 1 in absolute value – hence we include a bounds check to avoid taking the square root of a negative number.

§7 Extending the intrinsic 80x87 stack

As promised in the beginning of the chapter, we now design a fstack manager in software that allows more than 8 cells. First we examine the structure of the 80x87 stack (87stack).

From 8087P we see that the 8 registers in the 87stack are organized as a circular stack. A 3-bit pointer, ST, records which physical register is actually TOS. The instruction set allows ST to be incremented (FINCSTP) or decremented (FDECSTP) modulo 8. The 87stack is shown below in Fig. 4-1 on page 81.

A FLD instruction decrements ST (mod 8) before storing whereas a FSTP instruction increments ST. To build our software fstack we need to do the following things:

- When the 87stack gets full, put ST(7) on the memory extension, and *vice-versa.*
- Keep track of how many numbers are on the 87stack.
- Keep track of where we have stored the last number removed from the 87stack.

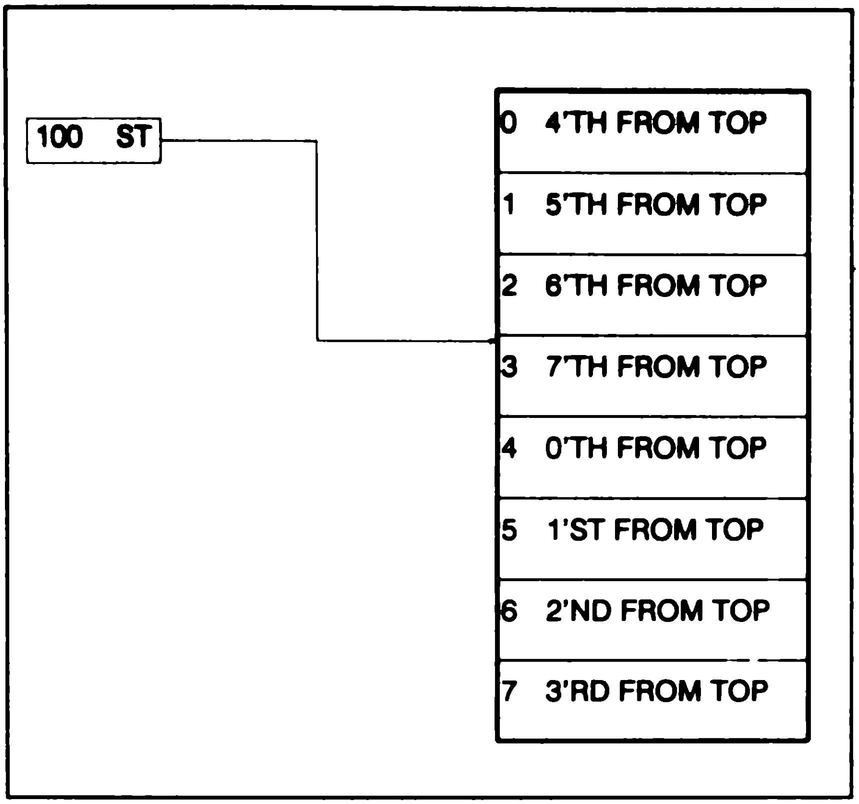

Fig. 4-1 *The 80x87 stack, from 8087P*

The algorithm has the following expression in pseudo-FORTH

```
Redefine operations that put #s on 87stack:
    increment stack_pointer
    if 87stack full put st(7) on fstack
    push onto st(0)

Redefine operations that take #s off 87stack:
    pop st(0)
    decrement stack_pointer
    if fstack not empty, put tofs onto st(7)
```

We begin by defining the data structure (the fstack proper) where we will stash and retrieve the numbers coming off the 87stack.

- We need to decide how deep the fstack will be, in 80-bit (TREAL) wide cells, and then **ALLOT** 10* that number of bytes of storage.
- We need a word that will initialize the 80x87 and fstack.
- We need a fast way to increment (or decrement) the address of the next available space in the fstack by 10.
- We need to test whether the 87stack is full (or empty).
- Finally, what do we do if the memory we set aside gets full? The solution chosen below makes the extension circular using modulo arithmetic to compute the addresses within it.

We now exhibit the fstack manager program in Fig. 4-2 on page 83 below. By now the reader should be familiar enough with FORTH style to understand the program logic with only moderate commenting and explanation. Remember – the program reads from the bottom up!

The key words are **FPUSH** and **FPOP** – they do the work. Every word that changes the 87stack has to be redefined to include either **FPUSH** or **FPOP** as appropriate.

As the test results in Table 4-5 below make clear, the high-level stack extension is too slow. Some optimization is necessary.

Table 4-5 *fstack manager timings*†,‡

†8086 machine @ 4.77 MHz

‡40,000 FPUSH's and FPOP's

High-level	Optimized	Hand-coded
29 sec	6 sec	4 sec

HS/FORTH comes with a very helpful utility: a recursive-descent optimizer. The optimizer replaces the subroutine calls of ordinary high-level threaded FORTH code by in-line machine code,

```
\ FLOATING POINT STACK MANAGER
DECIMAL
VARIABLE FS-SIZE                 ( size of fstack in TREAL's)
40 FS-SIZE !                     ( the fstack is 40 deep)
VARIABLE FLGTH                   ( length of fstack in bytes)
VARIABLE FSP                     ( current offset into fstack)
VARIABLE FDEPTH                  ( current size of stack)
                                 ( -8 fdepth fs-length)
CREATE FSTACK FS-SIZE @ 10 * ALLOT OKLW
: FSINIT FSP 0! -8 FDEPTH ! FS-SIZE @ 10 *
      FLGTH ! FINIT ;            ( initialize 8087 and stacks)
CODE 10+      <% 83 C3 0A %>         END-CODE
CODE 10-      <% 83 EB 0A % >        END-CODE
        ( fast way to add or subtract 10)
FSINIT
: WRAP( fsp - - fsp mod flgth )     \make fstack circular
      [ FLGTH @ ]  LITERAL  UNDER  +  SWAP  MOD ;

: AWAY!  DUP  ROT  ! ; ( adr n - - n)       \ useful factored word
: INC-FSP   ( - - fsp')  FSP  DUP@   10+  WRAP  AWAY!  ;
: DEC-FSP   ( - - fsp')  FSP  DUP@   10-   WRAP  SWAP  ! ;

: INC-FDEPTH   ( - - fdepth')
      FDEPTH  DUP@    1+     FS-LENGTH  @   MIN
      AWAY! ;

: DEC-FDEPTH                     ( - - fdepth')
      FDEPTH  DUP@  1-     DUP  -8  <
      ABORT" FSTACK UNDERFLOW"  AWAY!   ;

: FPUSH  INC-FDEPTH   0 <  NOT
      IF  INC-FSP   FSTACK   +
          FDECSTP  R80!       ( st[7] - - fsp )
      THEN  ;

: FPOP  DEC-FDEPTH  -1  <   NOT
        IF   FSP @  FSTACK  +
             R80@   FINCSTP   ( fsp - - st[7] )
              DEC-FSP  THEN  ;
```

Fig. 4-2 *Hi-level fstack manager for 80x87*

stripping out redundant pushes and pops of the parameter stack. It is fairly straightforward to construct a similar optimizer for any FORTH dialect. Alternatively, one can machine code the time-critical words.

The results of the tests, presented in Table 4-5 on page 82 above, are interesting:

- the optimizer does nearly as well as hand-tuned machine code.
- An FPUSH or an FPOP takes about 50 μsec on the average, when coded by hand, @ 4.77 MHz, or about 240 clock cycles.
- The irreducible minimum on an 8086/80286 – since TREAL storage from, or retrieval to the 87stack is demanded – cannot possibly be less than 170 clock cycles: 1 fetch and 1 store[19]. (The main place to save some time would be in moves to or from memory; the depth of the fstack and the pointer must be kept as variables, but **FS-SIZE** and **FLGTH** do not change and could be compiled as literals, thereby saving 30 cycles in **FPUSH** and 10 in **FPOP**.)
- Substantially greater efficiency (at best another 1.5x improvement over the code version discussed above) would require a different algorithm – based, perhaps, on the 87stack overflow or underflow interrupt. But because other error conditions can initiate this interrupt, the testing and decision-making needed to use this method seemed to me likely to produce equivalent overhead to the method employed here.

§8 Clone wars

Several companies have produced non-infringing clones of the Intel 80x87 family of chips. American Micro Devices is a second source for 80287 and 80387 chips. Cyrix Corporation has produced 80287 and 80387 equivalents with significantly faster transcendental functions and moderately faster arithmetic than the Intel originals.

And finally, Integrated Information Technology, Inc. (founded by the designers of the Weitek chips) has produced the most inter-

19. Somewhat less, ≈100 clocks, if a 32-bit bus is utilized with the '386/'387 pair. See, *e.g.*, John H. Crawford and Patrick P. Gelsinger, *Programming the 80386* (Sybex, San Francisco, 1987).

esting of the clones: the 80c287/80c387 not only perform arithmetic significantly faster than the Intel originals, they possess 24 additional 80-bit on-chip registers. Unfortunately these cannot be combined directly with the eight original registers to make a 32-deep stack, since this would have required increasing the 80x87 stack-pointer (see §7 above) from 3 bits to 5. Since there was no place to find the 2 extra bits, one cannot really fault IIT's designers.

But if they cannot be used to extend the 87stack, what are the 24 extra registers good for? Eight (bank 3) are not even accessible, being used to speed up on-chip arithmetic. However, banks 0–2 – 24 registers – *can* be accessed (*e.g.* for on-chip cache memory). Moreover, IIT has provided an on-chip linear transformation: a 4×4 matrix multiplies a 4-dimensional column vector in place. (The original vector is overwritten, but the matrix is unchanged.) It works like this[20]:

First we define the instructions

```
CODE  FSBP0  <%  DB E8  %>   END-CODE
CODE  FSBP1  <%  DB EB  %>   END-CODE
CODE  FSBP2  <%  DB EA  %>   END-CODE
CODE  F4x4   <%  DB F1  %>   END-CODE
```

Then we load the 4×4 matrix into banks 1 and 2, the vector into bank 0, and multiply:

```
0 VAR a{{    0 VAR v{
: VEC->c87  ( adr - - )        \ load vector into 80c387
    IS v{                      \ ! vector address to v{
    FINIT  FSBP0               \ reset ST, select bank 0
    3 0  DO  v{ I }  G@  LOOP  ;
: MAT->c87  ( adr - - )        \ load matrix into 80c387
    IS a{{                     \ ! matrix address to a{{
    FINIT  FSBP2               \ reset ST, select bank 2
    1 0  DO
      3 0  DO  a{{ I J }}  G@  LOOP
    LOOP                       \ cont'd ...
```

20. Note: FINIT operates differently on IIT's coprocessors than on Intel's. It does not place NAN's in all 8 87stack cells.

```
    FINIT  FSBP1              \ reset ST, select bank 1
    3 2  DO
      3 0  DO  a{{ I J }}  G@   LOOP
    LOOP ;

:  c87->VEC  ( adr - - )      \ ! from 80c387 to vector
   IS v{                       \ ! vector address to v{
   FSBP0                       \ select bank 0
   3 0  DO  v{ I }  G!  LOOP ;
\ example:  ANS = A*V
A{{  MAT->87   V{  VEC->87   F4x4   ANS{ c87->VEC
```

The on-chip 4×4 matrix multiply was intended to accelerate 3-dimensional graphics (rotation and translation can be expressed as a single 4-dimensional transformation). However, most scientific programmers spend little time on 3-D graphics. The matrix instructions are more interesting for their potential to accelerate general matrix operations[21]. For example, suppose we need to transform a vector by multiplying with an arbitrary matrix. Normally we would write

$$y_i = \sum_{j=1}^{n} A_{ij} x_j \tag{7}$$

But consider, *e.g.*, an 8×8 matrix operating on an 8-dimensional column vector: we **partition** the matrix and vector into 4-dimensional sub-matrices a_{11}, *etc.* and sub-vectors x_1, *etc.*:

$$\begin{bmatrix}A\end{bmatrix}[x] \equiv \begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} = \begin{bmatrix}a_{11}x_1 + a_{12}x_2\\ a_{21}x_1 + a_{22}x_2\end{bmatrix} \tag{8}$$

From timings on assembly-coded demonstration programs that multiply vectors by constant 4×4 matrices, we estimate an overall speedup of 6–7 -fold on an 80c387 system. The timings for this process are as shown in Table 4-6 below (assume 32-bit REAL*4 matrices). The execution time for a 4-dimensional linear transformation using conventional operations is approximately 1744 clock cycles. The time using the vector operation is some 250-300 clocks, based on measured performance. Since 256 clocks are

21. See Ch. 9 of this book for a fuller discussion of standard matrix algorithms.

needed to load and store a 4-dimensional vector, we therefore estimate the vector operation **F4x4** takes only about 50 clocks, *i.e.* about 1 floating point multiplication time. We can now estimate the time to multiply two 4×4 matrices as about 1200 clocks *vs.* about 7000 for the scalar process, *i.e.* the same speedup factor

Table 4-8 *Timings for 4×4 matrix · vector*

Operation:	×	+	@	!
No. × Clocks:	16*52	12*28	20*20	4*44
Total clocks:	832	336	400	176

as for *matrix·vector*. If one must load the 4×4 matrix each time, the speedup factor is less: about 3.5-fold because of the 16 additional fetches.

The conventionally programmed 8×8 matrix-vector multiply should also be some 3-5 times faster than the scalar operations, *i.e.* there is no obvious speed gain —except being able to employ the built-in vector instruction **F4x4** — from partitioning the 8×8 system into 4×4 sub-units. However, Strassen[22] has pointed out that if one can evaluate matrix products recursively, partitioning can substantially speed the most time-consuming matrix operations, multiplication and inversion. For example, it appears as though the product of two partitioned matrices,

$$\begin{gathered}[A][B] = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \equiv [C] \\ [C] = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}\end{gathered} \tag{9}$$

22. V. Strassen, *Numer. Math.* 13 (1969) 184. See also V. Pan, *SIAM Review* 26 (1984) 393.

requires 8 matrix multiplications and 4 matrix additions to evaluate. Strassen has shown that in fact the evaluation can be performed with 7 matrix multiplications:

$$\begin{aligned}
p_1 &= \left(a_{11} + a_{22}\right)\left(b_{11} + b_{22}\right) \\
p_2 &= \left(a_{21} + a_{22}\right) b_{11} \\
p_3 &= a_{11}\left(b_{12} - b_{22}\right) \\
p_4 &= \left(-a_{11} + a_{21}\right)\left(b_{11} + b_{12}\right) \\
p_5 &= \left(a_{11} + a_{12}\right) b_{22} \\
p_6 &= a_{22}\left(-b_{11} + b_{21}\right) \\
p_7 &= \left(a_{12} - a_{22}\right)\left(b_{21} + b_{22}\right)
\end{aligned} \tag{10}$$

and 18 matrix additions:

$$\begin{aligned}
c_{11} &= p_1 - p_5 + p_6 + p_7 \\
c_{12} &= p_3 + p_5 \\
c_{21} &= p_2 + p_6 \\
c_{22} &= p_1 - p_2 + p_3 + p_4
\end{aligned} \tag{11}$$

Equations 10 and 11 look at first blush half as efficient as 8 multiplications and 4 additions. But let us examine the time to multiply two partitioned matrices, first by the straightforward method and then by Strassen's: clearly,

$$M_{2n} = 8M_n + 4A_n \tag{12}$$

where M_n is the multiplication time and A_n the addition time, for square matrices of order n.

Setting $n = 2^k$ that (note $A_n \cong O(n^2)$) we see the recursion, Eq. 12, is satisfied by an expression of form

$$M_n \approx m\lambda^k + ca\,4^k \tag{13}$$

where m and a are the elementary multiplication and addition times. Substituting 13 in 12 we find $\lambda = 8$ and $c = -1$, *i.e.*,

$$M_n = mn^3 - an^2 \tag{14}$$

Applying the same idea to Strassen's method we obtain

$$\hat{M}_{2n} = 7\hat{M}_n + 18an^2 \tag{15}$$

or

$$\hat{M}_n = mn^{lg\,7} - 6an^2\,, \tag{16}$$

where

$$lg\,7 \equiv \log_2 7 = 2.807\ldots$$

That is, partitioning allows a potentially large reduction in the time to multiply dense matrices.

By writing a partitioned matrix in the form

$$\left[A\right] = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ a_{21}a_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ 0 & z \end{bmatrix} \tag{17}$$

where

$$z = a_{22} - a_{21}a_{11}^{-1}a_{12} \tag{18}$$

we may express the **inverse** of A as

$$\left[A\right]^{-1} \equiv \begin{bmatrix} a_{11}^{-1} & a_{11}^{-1}a_{12}z^{-1} \\ 0 & z^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -a_{21}a_{11}^{-1} & I \end{bmatrix} \tag{19}$$

which leads to the recursion for I_n, the time to invert an n×n matrix:

$$I_{2n} \approx 2I_n + 5M_n \tag{20}$$

whose solution is

$$I_n = mn^{lg\,7} + O(n) \tag{21}$$

i.e., the time needed to invert is comparable with that needed to multiply.

Suppose we merely wish to solve a linear system *without* inverting the matrix: can we gain some speed that way? From 17 we see that the problem

$$Ax = y$$

reduces to three sub-problems:

$$a_{11}u_1 = y_1 \qquad (22a)$$

$$zx_2 = y_2 - a_{21}u_1 \qquad (22b)$$

$$a_{11}x_1 = y_1 - a_{12}x_2 \quad ; \qquad (22c)$$

that is, we have the recursion

$$S_{2n} = 3S_n + mn^{lg\,7} + 2mn^2 \qquad (23)$$

whose solution is dominated by

$$S_n = m\left(\frac{1}{4}n^{lg\,7} + 2n^2\right) + O(n^{lg\,3}) \qquad (24)$$

Linear equation solution *via* recursion thus has the same asymptotic running time as matrix multiplication, except that 4× fewer operations are required than for inversion. That is, it should be about 5× faster to solve a dense system of 1000 linear equations by recursive partitioning than by ordinary Gaussian elimination even on a scalar processor. The vector instruction on the IIT 80c387 chip, together with Strassen's algorithm, offers the possibility of solving very large systems in practical times, on desktop computers.