The Intel 8087 Numerics Processor Extension

This chip lets you perform mathematical operations with 18-decimal digits of accuracy.

R. B. Simington
Intel Corporation
27 Industrial Ave.
Chelmsford, MA 01824

So you have a shiny new 8086/8088-based microcomputer and you want to write some accounting software that will meet the requirements of the fussiest CPA. Or perhaps you want to write some code to control your latest robotics experiment and you need to do some critical number-crunching with very accurate results. Let me introduce you to the Intel 8087 Numerics Processor Extension (NPX).

The 8087 NPX is a chip that will give you, regardless of your mathematical skill, the ability to write code that will perform mathematical operations with guaranteed accuracy of 18-decimal digits. To make things even easier, the NPX, known as a coprocessor, is easy to interface to a host processor (8086 or 8088). When you add an NPX to one of its host processors, you have a Numeric Data Processor (NDP).

Registers

The 8087, due to its special status as a coprocessor, combines with the architecture of the host processor to produce a processor that contains all the capabilities of an 8086 or 8088 plus a register stack of eight 80-bit registers, as shown in figure 1. The NDP also brings the ability to execute a set of 68 new floating-point-arithmetic instructions in addition to the extensive instruction set of the host alone. These 80-bit registers hold all the operands for the floating-point operations in a format called Temporary Real, which is capable of representing any whole number up to 2^{64} exactly. This 80-bit format is accurate enough to guarantee 18-decimal-digit accuracy.

You can think of the eight registers of the 8087 as a standard register set or use them as a classical stack. This arrangement allows you a great amount of flexibility in using these storage areas. You can use the registers as a stack, for example, to

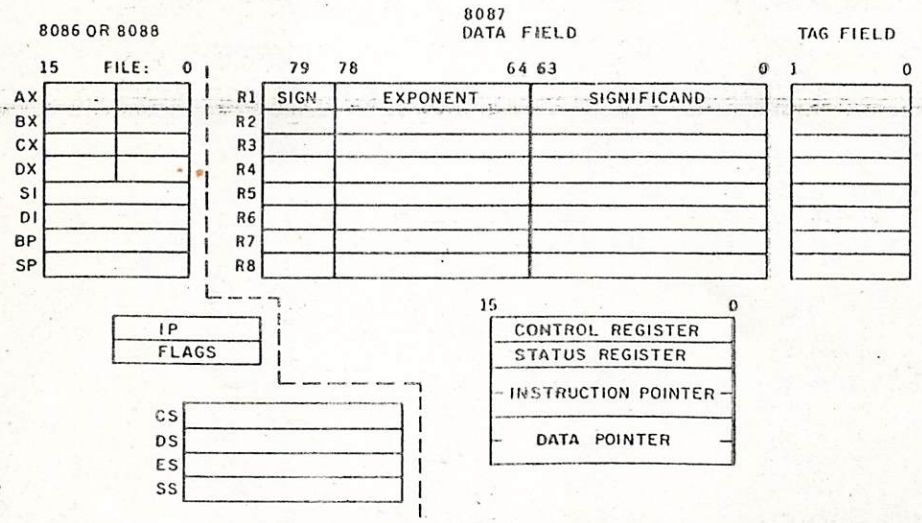


Figure 1: *The NDP register set of eight 80-bit registers.*

About the Author

Bob Simington is a curriculum developer for Intel's customer training department. He is responsible for the development of all microprocessor training curricula. His most recent assignment was to develop a course based on 16-bit personal computers.

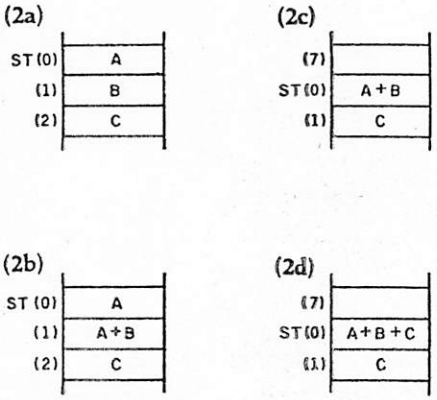


Figure 2: *The effects of instructions on the register stack.*

pass operands to a floating-point subroutine that expects to find its parameters in the 8087 register stack. The instruction set of the 8087 exploits this capability by allowing assembly-language instructions that use the stack top as one of the operands implicitly.

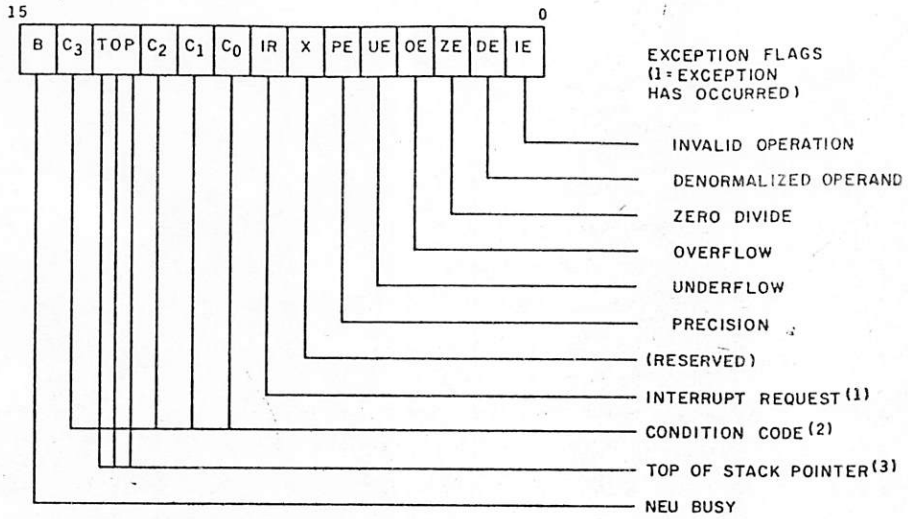
You can also treat these registers as a standard register set; the instruction set supports this use of the registers by allowing both operands to be specified explicitly for an operation. Look at the example in figure 2.

Your NPX stack contains the values A, B, and C, as shown in figure 2a. You can cause the value in the stack top to be added to the value in the next stack element (ST(1)), as shown in figure 2b, and then have the stack popped, so that the sum is the new stack top, as shown in figure 2c, with the one instruction FADD. This is the classical stack operation where none of the operands are specified. The same instruction could have been specified explicitly with the instruction FADDP ST(1),ST.

As another example, you could add the value that is in ST(1) to the value in the stack top and retain both values, as shown in figure 2d, with the instruction FADD ST,ST(1). All registers are addressed with respect to the current stack top. The Status Word of the 8087 contains a field that identifies the current stack-top status at all times.

Status Word

The Status Word, as its name implies, reflects the status of the NDP at



(1) IR IS SET IF ANY UNMASKED EXCEPTION BIT IS SET, CLEARED OTHERWISE.
(2) SEE TABLE 3 FOR CONDITION CODE INTERPRETATION.
(3) TOP VALUES:
000 = REGISTER 0 IS TOP OF STACK.
001 = REGISTER 1 IS TOP OF STACK.
•
•
•
111 = REGISTER 7 IS TOP OF STACK.

Figure 3: *The various fields of the Status Word.*

all times. The various fields of the Status Word are shown in figure 3.

The Busy bit indicates whether the 8087 is in the process of performing an operation at any given time. The 4 condition bits (C3 through C0) denote the result of a previous operation, and the host processor can examine them to determine if a branch should take place in the program. This is accomplished by storing the 8087's Status Word in memory and then causing the host processor to fetch and examine the condition codes.

Three of the bits of the Status Word, ST2 through ST0, specify which register is the current top of the stack. Notice that if the ST field contains 000, for example, and a value is pushed onto the NDP's stack, the ST field would decrement to 111. By the same token, if the ST field contains 000 and a value is popped from the stack, the ST field would increment to 001.

As it turns out, you really don't need to know which register is the current stack top, but this gives you some idea of what the bits mean. You *do* need to know the relative position of your various data elements on the stack, however, and keeping track of them is one of the jobs of the programmer.

All the remaining bits in the Status Word apply to exceptions (errors) and the way to handle them. Bits 0 through 5 signify whether the NDP detects an exception condition. Table 1 indicates the significance of these exceptions. Bit 7 is set to indicate whether the 8087 requests the host central processing unit to execute an interrupt-service routine in response to one of the exception conditions. It is possible for an exception condition to take place without an interrupt request if the programmer masks a particular exception (or group of exceptions). Generally, however, the NDP uses a collection of built-in default routines to handle most exceptions.

If you really want to avoid the messy error-condition-handling business, you can leave those problems to the nice people who wrote the

Invalid Operation
- Attempt to load a register that is not empty
- Attempt to pop an operand from a register
- Operand is a NAN (not a number)
- Operands cause operation to be indeterminate (%, $\sqrt{}$ – number

Denormalized Operand
- Attempt to use an operand that is not normalized

Zerodivide
- Attempt to divide by 0

Overflow
- Result too large for destination format

Underflow
- Result too small for destination format

Precision
- Result not exactly representable in destination format
- 8087 rounds result

Note: Exception bits are "sticky" and can be cleared only by the FCLEX (clear exceptions) instruction.

Table 1: *The significance of the exceptions.*

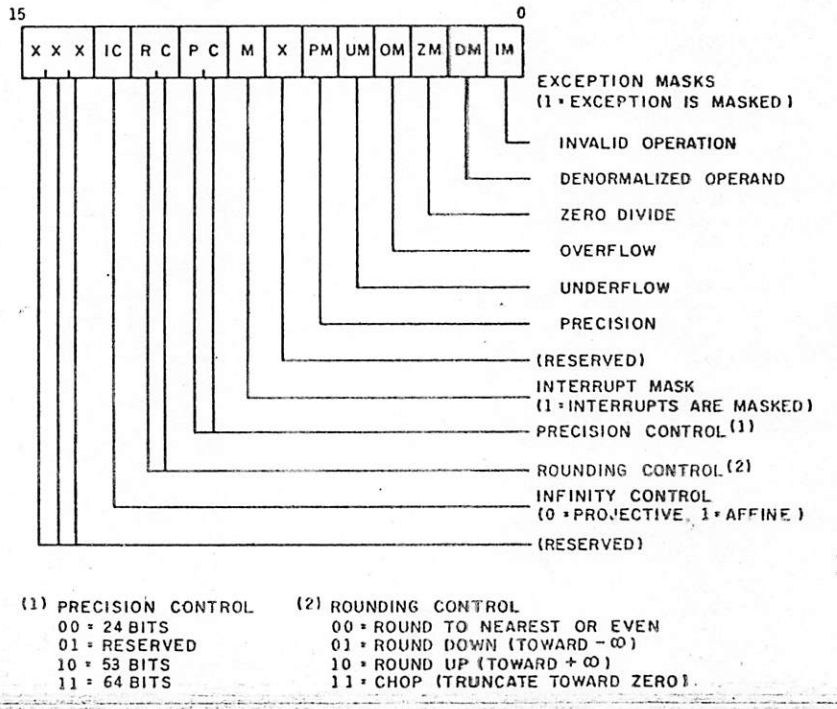


Figure 4: *The Control Word.*

IEEE standard that specifies the NDP default routines. (See the text box on pages 174-175.) In other words, you can take responsibility for handling an exception condition or you can let the chip do it. To inform the NDP which exceptions it will handle and which ones you will handle, you use the Control Word.

Tag Word

Before we move to the Control Word, let's look at one other thing. How would the NDP know if we had tried to pop something from an empty register so that it could set the invalid operation flag? For that matter, how would it know if one of the operands was not a number? Well, the NDP stores a Tag Word that contains a 2-bit field for each of the eight registers, and which keeps track of the contents of each register and marks the contents as being Valid, Zero, Special (not a number, infinity, etc.), or Empty.

Control Word

The Control Word is shown in figure 4. The least significant portion of the Control Word coincides with the least significant portion of the Status Word in that you set or clear these bits to mask or unmask (respectively) the exception conditions and to disable or enable the ability of the 8087 to interrupt the host processor. The Control Word also has bits that allow you to set the precision of the NDP to various levels. You should realize that there is no advantage to lowering the precision of the NDP in terms of execution speed. This capability exists only to allow the 8087 to more accurately emulate previous floating-point products that might be replaced by the 8087.

The next 2 bits in the Control Word set the rounding convention that the NPX uses. You can set the NPX to round up, round down, round to the nearest or even number, or to round toward zero.

Lastly, a bit in the Control Word selects the model of infinity for the NPX. Just think, philosophers argue whether infinity even exists and you get to define it in one of two different ways!

Simply put, the first method, known as the Affine closure mode, considers that there are numbers that are positively infinite and numbers that are negatively infinite. While this may seem reasonable, it can cause unexpected problems in calculations. For example, if we divide a number by the number X, and X is very, very small and positive, the result could be positive infinity. If X is very, very small but negative, the result could be negative infinity. In other words, you have two very different results.

The Projective closure mode uses only one infinity. Although it is very large, it has no sign. This mode is the easiest to use and therefore is recommended.

Exception Pointers

If an exception does take place, and you have unmasked that particular exception flag, it would be nice to know a few things before you jump in and try to recover from your error. You would probably like to know

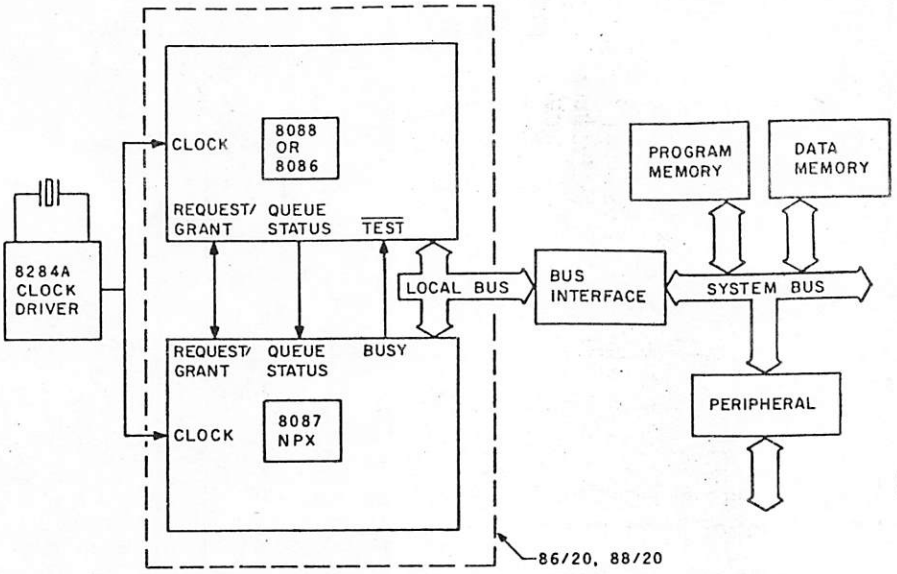


Figure 5: *The interface and communication lines.*

which instruction or op code caused your problem. It would also be nice if you could determine this instruction's address so that you might resume at that point when you have eliminated the cause of the problem. Also, if the instruction operated on data in memory, you might want to know where this data came from. The 8087 contains all this information in three registers that are collectively called the Exception Pointers. The operational details of these registers are beyond the scope of this article, but can be found in Intel technical documents.

Synchronization

The 8087 works in close harmony with its host. In order for this to take place, there must be some communication and synchronization between the two devices. The interface and communication lines are shown in figure 5.

When you add an 8087 to a host microprocessor, the host retains the responsibility of fetching the instructions from system memory. As the host fetches these instructions, the 8087 also captures them and puts them in a queue—just as the host processor does. The answer to how the 8087 knows whether to emulate the 6-byte queue of the 8086 or the 4-byte queue of the 8088 is quite simple.

Whenever the processor is reset, the host processor goes to absolute address 0FFFF0 hexadecimal to begin fetching instructions. The 8088 always fetches a byte from this location because the 8088's data bus is a byte wide. The 8086, however, can fetch its instructions one word at a time. So when it goes to address 0FFFF0 to fetch its first instruction, it fetches a word instead of a byte. To accomplish this, the 8086 issues a signal called Byte High Enable (BHE/). The 8087 simply looks to see if its host generates a BHE/ signal when it performs its first op-code fetch after a reset. The presence or absence of the signal determines if it is connected to an 8088 or an 8086, and it then emulates the correct queue.

In order for the 8087 to work in close harmony with its host, there must be some synchronization between the two devices.

All 8087 instructions have a unique 5-bit prefix called the Escape prefix (11011). When the host fetches an op code with this prefix, it is placed in the queue as any other instruction would be. The 8087 captures the same op code as it is fetched by the host and places the op code in its own queue. As this op code moves through the queue of the host processor, its movement must be emulated by the NPX queue as well. The mechanism that is set up to do this is the first of the three types of synchronization signals that are shared by the host and the NPX. The host has two lines called the Queue Status lines (QS0 and QS1). They allow an external device such as a coprocessor to emulate its queue. When this floating-point instruction finally makes its way through the queue, the host determines that it is an instruction for the NPX rather than for itself.

Once the host has determined that the instruction is an NPX instruction, it looks at the instruction to see if the NPX will need to access memory in the execution of the instruction either to fetch (read) an operand from memory or to store (write) an operand to memory. If it sees that this is the case, the host will calculate the address of the operand and perform a *dummy read*. By performing this dummy-read operation, the host can supply the NPX with the address of the memory operand. The host generates the address and even issues the strobe signals, but it ignores the data that is read from the addressed location.

The NPX, on the other hand, captures the address that the host generates. Then, depending on whether the NPX needs the data that is read or whether it needs the address so that it can write a result in memory after it finishes its instruction, the NPX captures the data or ignores it. Even if the data were required by the NPX to execute the instruction at hand, the dummy read will result in only one byte or one word of data. Typically, the NPX will be working with data types that are 4, 8, or even 10 bytes in length. This means that the 8087 will have to perform additional memory-read cycles in order to access the remainder of its operand. Because the 8087 shares the host processor's buses, it must have a way of informing the host that it wishes to use them.

The mechanism to do this is the

Text continued on page 170

same used by any potential bus master to borrow bus cycles from the host. This line is the Request/Grant line (RQ/GT), and it is the method that the host uses to implement the HOLD and HLDA (hold and hold acknowledge) functions. When the NPX wants to use the buses, it toggles the RQ/GT line low. The host latches this request and, when it is done with the buses, three-states its buses and signifies to the requesting device that it can have them by toggling the same line low itself.

The NPX takes over the buses at this point and generates all the signals (address and control) to access the operand. It signifies when it is finished by once again toggling the RQ/GT line low. In the case where the NPX needs the address generated during the dummy read to store a result, it simply executes its instruction, saving the address until it has completed the execution and then requests the bus so that it can perform the memory-write cycles required to put the result in the proper memory location.

In some cases, the host can continue to fetch and execute instructions (with the NPX emulating its queue) while the 8087 is busily crunching numbers. There may be a situation, however, when this cannot take place. For example, we have two instructions in a program, the first of which causes the NPX to operate on a variable and to store the result in memory. If the second instruction causes the host to use this result, we must not allow the host to execute the second instruction and to access the variable until the 8087 has written the proper value in the variable location.

To accomplish this, we precede the instruction intended for the host with a WAIT instruction. The WAIT instruction causes the host to look at its TEST pin. The TEST pin, as you can see in figure 5, ties to the BUSY line of the NPX. Whenever the NPX is busy, the BUSY line is high and the TEST pin on the host is high. When the host executes the WAIT instruction, it cannot go on to the next instruction until its TEST pin goes low.

Another case where you would not want the host to go off blindly is when you have more than one floating-point instruction back to back. In this situation, the host would like to be more than obliging by performing all the dummy reads in the world just so that it can get to an instruction that it can execute. The problem is that the 8087 queue has to be in sync with the host, and if the host just gobbles its way through a series of floating-point instructions, ignoring them as it is designed to, the devices will never be able to regain sync. You can solve this problem simply by adding a WAIT instruction in front of all floating-point instructions.

Listing 1: *An application of the 8087 using an Intellec Series III Microcomputer Development System and Intel's 8087 software emulator.*

```
SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE PYTHAGORUS
OBJECT MODULE PLACED IN :F1:EUCLID.OBJ
NO INVOCATION LINE CONTROLS

LOC  OBJ                  LINE     SOURCE

                             1 +1  $SYMBOLS DEBUG
                             2
                             3             NAME PYTHAGORUS
                             4
                             5             EXTRN   IN1T87:FAR
                             6
----                         7     TRIANGLE        STRUC
0000                         8             BASE    DD      ?       ; The DD memory allocation allows
0004                         9             ALT     DD      ?       ; enough space for the variables
0008                        10             HYP     DD      ?       ; to be defined in the SHORT REAL
000C                        11             AREA    DD      ?       ; format.
----                        12     TRIANGLE        ENDS
                            13
----                        14     DATA    SEGMENT PUBLIC  'DATA'
                            15
0000 ????????               16     RIGHT   TRIANGLE        <>      ;We will assume that these
0004 ????????
0008 ????????
000C ????????
                            17                                     ;variables are initialized by
                            18                                     ;another program.
0010 3E03                   19     CONTROL_87      DW      033EH
                            20
                            21     ; This is the default control word which masks all exception except
                            22     ; INVALID OP, leaves the INTERRUPTS UNMASKED, ROUNDS TO NEAREST OR EVEN,
                            23     ; and sets the INFINITY CONTROL to PROJECTIVE.
                            24
                            25
----                        26     DATA    ENDS
                            27
                            28
----                        29     STACK   SEGMENT STACK   'STACK'
                            30
0000 (100                   31             DW      100 DUP(?)
     ????
     )
```

Application

The following application of the 8087 uses an Intellec Series III Microcomputer Development System and Intel's 8087 software emulator (see listing 1). The program takes the base and altitude of a right triangle, which are stored in memory as two fields of a structure called TRIANGLE, and uses this data to calculate the area of the triangle and the length of its hypotenuse. Notice that I define the fields of the structure as double words (DD). This allows me to define the variables in the Short Real format. I will discuss the program using both the listing and a chart in figure 6 that tracks the status of the NPX's stack as references. When I refer to the chart, I use lowercase letters in parentheses.

The procedure INIT87, which I reference on line 42, is in the 8087 emulator and prepares the environment. The FLDCW instruction on line 46 causes the Control Word, which is stored in memory at a location symbolically called CONTROL-87, to be loaded into the 8087. The function of the Control Word is explained beginning with line 21.

Notice that all the floating-point instructions are preceded by a hexadecimal 9B op code. This is the WAIT op code that is inserted automatically by the assembler. The first thing that I must do is set up the stack so that I can perform my calculations. The instructions on lines 50 through 53 perform this. FLD1 puts the number 1.0 on the top of my stack (figure 6a). FADD ST,ST(0) causes the value at ST(0), which is my stack top, to be added to the stack top.

The result of this instruction is to multiply the stack top by 2 (figure 6b). The FLD instructions cause the 8087 to load the contents of the specified memory locations, which are considered to be in the Real format, on the stack (figure 6c). In the process of loading these values, they are transformed into the Temporary Real format. Notice that because the host processor calculates all the operand addresses, I can use any of the addressing modes of the host. The 8086 and 8088 processors are capable of 24 powerful addressing modes. In this case, I am accessing a field of a structure.

Next, I calculate the area of the triangle. The instructions that do this are on lines 57 through 60. The FLD ST(1) instruction causes the contents of the current ST(1) register to be duplicated on the top of my stack (figure 6d), and the FMUL ST,ST(1) instruction puts the BASE X ALT in the stack top (figure 6e). FDIV ST,ST(3) divides the stack top by the 2.0 that I have saved on the stack (figure 6f), and the FSTP causes the result, which is equal to the area of the triangle, to be popped into the ap-

Listing 1 continued:

```
00C8                            32      T_O_S   LABEL   WORD
                                33
----                            34      STACK   ENDS
                                35
                                36 +1   $EJECT
----                            37      CODE    SEGMENT PUBLIC  'CODE'
                                38              ASSUME  CS:CODE,DS:DATA,SS:STACK
                                39
                                40      ; INITIALIZE 8087
                                41
0000 9A0000----      E          42      INIT:   CALL    INIT87          ; This routine is in a library.
                                43                                      ; It sets up the environment
                                44                                      ; for the 8087.
                                45
0005 9BD92E1000      R          46              FLDCW   CONTROL_87      ; Load control word from memory
                                47
                                48      ; SET UP 8086 REGS AND CONSTANT STACK IN 8087
                                49
000A 9BD9E8                     50      SETUP:  FLD1                    ; PUT 1.0 IN STACK TOP (ST)
000D 9BDCC0                     51              FADD    ST,ST(0)        ; ST=ST X 2
0010 9BD9060000      R          52              FLD     RIGHT.BASE      ; ST <-- BASE
0015 9BD9060400      R          53              FLD     RIGHT.ALT       ; ST <-- ALT
                                54      ;
                                55      ; CALCULATE AREA=(BASE X ALT)/2 AND STORE IN MEMORY
                                56      ;
001A 9BD9C1                     57      CALC:   FLD     ST(1)           ; Duplicate BASE in ST
001D 9BD8C9                     58              FMUL    ST,ST(1)        ; ST <-- BASE X ALT
0020 9BD8F3                     59              FDIV    ST,ST(3)        ; ST <-- ST/2
0023 9BD91E0C00      R          60              FSTP    RIGHT.AREA      ; Store ST in AREA and discard
                                61      ;
                                62      ; CALCULATE HYPOTENUSE = ((BASE**2)+(ALT**2))** 1/2
                                63      ;
0028 9BDCC8                     64              FMUL    ST,ST(0)        ; Square ALT
002B 9BD9C9                     65              FXCH    ST(1)           ; Exchange ALT**2 and BASE
002E 9BDCC8                     66              FMUL    ST,ST(0)        ; Square BASE
0031 9BDEC1                     67              FADD                    ; ST <-- BASE**2 + ALT**2
0034 9BD9FA                     68              FSQRT                   ; ST <-- ST** 1/2
0037 9BD91E0800      R          69              FSTP    RIGHT.HYP       ; Store ST in HYP and discard
                                70
003C F4                         71      HALT:   HLT
                                72
                                73
                                74
----                            75      CODE    ENDS
                                76              END     INIT,DS:DATA,SS:STACK:T_O_S
```

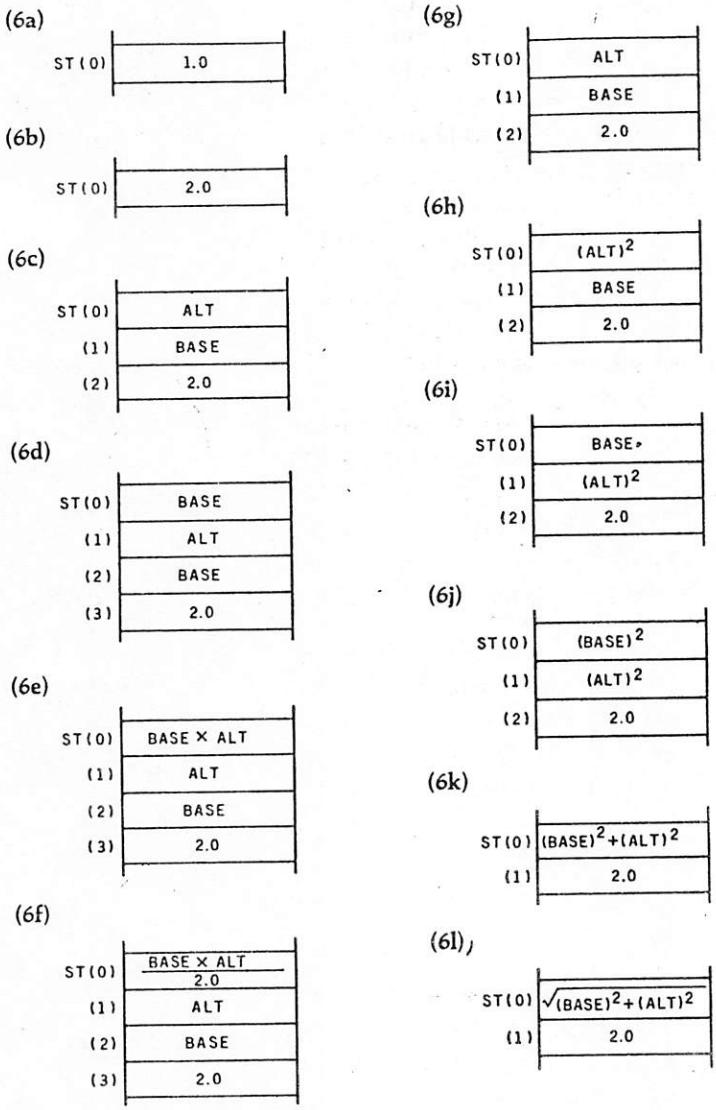


Figure 6: *The status of the NPX stack during operation of the program in listing 1.*

propriate memory location in the Short Real format (figure 6g).

My next task is to calculate the length of the hypotenuse of the triangle. The instructions to do this are on lines 64 through 69. First, I multiply the element on the top of the stack by itself (figure 6h). Next, I exchange this element with the element below it (figure 6i), and once again I square the stack top by multiplying it by itself (figure 6j). All that remains is to add the stack top to the register below it, pop the old stack top (figure 6k), and find the square root of the stack top (figure 6l). The value in the stack top now equals the length of the hypotenuse. I next pop this value into the appropriate location in memory and I am finished.

This concludes the example. I have just performed calculations that involve transcendental functions and that use operands that are very accurate with no concern for either the complexity of the algorithm or the accuracy of the result. If I had attempted to perform the same calculation in software, I would have had to write some very complicated algorithms and the code necessary to trap any errors that may have occurred. Instead, I let the 8087 and the standard to which it complies worry about such things. It puts the world of accurate, high-level mathematics in the realm of every assembly-language programmer. ■

Guaranteed Standard Results with the 8087

In an attempt to standardize both the data types and algorithms used in floating-point-arithmetic problems, the IEEE formed a committee in 1979 to produce a standard to which all floating-point-arithmetic processors and software would comply. This standard not only established standard data formats, but also set rules for rounding and precision control and required that all machines (and software packages) complying with the standard be capable of performing the same algorithms with the same predictable results. The concept of exception handling was expanded and standardized so that all machines (and software) that complied with the standard would monitor their own activity and be capable of signifying when the result of their calculations was suspected of being in error. The 8087 implements this standard fully.

Data Formats

One of the most important aspects of the IEEE standard was the definition and standardization of the seven data formats that would be used in floating-point calculations. These formats belong to three basic types: Integer, Decimal, and Real.

The Integer type should be familiar because it is a standard data type used quite generally. The Integer formats are capable of representing signed whole numbers. The most significant bit denotes the sign (where a 0 indicates a positive number and a 1 indicates a negative number that is represented in standard two's-complement notation). The three Integer formats differ in length. The shortest format, Word Integer, is 16 bits in length and identical to the Integer format used by the host processor in its own instruction set (IMUL, IDIV). The Short Integer is 32 bits in length. Most 8086 assemblers allow you to allocate space for this type of variable using the Define Doubleword (DD) operator. Appropriately, the longest format is the Long Integer, which is 64 bits in length. Most assemblers allow you to allocate space for this type of variable with the Define Quadword (DQ) operator.

The Packed Decimal format, 80 bits long, is similar to the Integer format in that the most significant bit is a sign bit. The similarity ends there, however, because the data is stored in Packed Decimal format where the 18 least significant nybbles (bits 0 through 72) each represent a decimal digit. This means that each group of 4 bits must be equal to a value between 0 (0000) and 9 (1001). If this looks like a lot of data, remember that this processor is capable of delivering results that use all 18 nybbles of this format with no round-off error! You can use the Define Tenbyte (DT) operator to allocate space for this variable.

Up until now I have described only data formats that represent whole numbers. While it would be nice if everything always worked out neat and clean without any fractions, in the real world we have to deal with all types of numbers. The next data type, Real, was created to handle these types of numbers.

The Real data type contains three formats. They are very similar to the scientific notation that represents large decimal numbers. A number represented in scientific notation has three components—sign, mantissa, and exponent. In scientific notation, I would represent the decimal number $-547{,}390$ as -5.4739×10^5. In other words, I denote the sign of the number in the normal manner. Then I display its value or mantissa as a value between 0 and 10. I do this by moving the decimal point to either the right or the left. Whenever I move the decimal point to the left to normalize the number (make it equal to a value between 0 and 10), the exponent increases. When I move the decimal point to the right to normalize the number, the exponent decreases. For example, 0.0075 becomes 7.5×10^{-3}. The Real formats also have a sign, mantissa or significand, and an exponent, but they are binary values that are calculated a little differently.

As an example, I'll convert the decimal number 66 to the Short Real format. The Short Real format uses the 23 least significant bits to denote the mantissa of the number, the next 8 most significant bits to denote the exponent, and the most significant bit to denote the sign of the number. The first thing I must do is to convert the number to binary: decimal 66 = binary 1000010. Because this is a whole number, I have no fractional part to worry about. Now I must normalize the number by moving the decimal point over so that the value has a "1" in the most significant digit. As in scientific notation, I increase the exponent whenever I move the decimal point to the left and decrease the exponent whenever I move the decimal point to the right. In this case, because I move the decimal point to the left, the exponent is positive: 1000010 = 1.000010 E 110.

Now I am just about to finish the conversion, but you must know two things. In the Short and Long Real formats I save a bit by dropping the leading "1" bit of the significand. This allows me to denote 24 or 53 digits in the space required for 23 or 52 in the Short or Long formats, respectively. For some reason, this is not done in the Temporary Real format. The second thing is that the standard format requires that a bias be added to the exponent so that negative exponents appear as positive values. The bias is different for each format and is shown in figure 7. The bias for the Short format is 7F hexadecimal (01111111). This means that the exponent becomes 110 + 01111111 = 10000101. Putting it all together, I get

sign	*biased exponent*
0	*10000101*
significand	
00001000000000000000000	

or

0100 0010 1000 0100 0000 0000 0000 0000

With a number with a fractional part, the conversion looks like that shown in table 2. Table 3 is useful in understanding the power of the various data types.

Guaranteed Accuracy

The key to the 8087's ability to produce results with such a high degree of accuracy lies in the fact that the 8087 performs all its calculations in the Temporary Real data format. Regardless of the format in which an operand is stored in memory, any piece of data

that is loaded into the 8087 is transformed into this highly accurate, 80-bit-long format. The benefit derived by using this format for all internal calculations can be best illustrated by the following analogy.

I have a calculator with a range of 3 decimal digits and I want to use it to solve the equation $F = A \times B / C$. I got a calculator with 3-digit accuracy because I determined that A, B, and C would never be greater than 3 digits in length. If A=100, B=10, and C=10, you can see that I will have trouble solving this problem using my calculator because $A \times B$ is outside of its range. Even though the final result may be well within the range of my calculator, I may have trouble with the intermediate results.

Although changing the order of operations may help in this particular situation, it should be obvious that this solution will not resolve the problem in all cases, for instance, if A=100, B=10, and C=0.1. In this case, I would have a problem if I were to calculate A/C.

Now, if I got a new calculator that had a range of 4 or 5 digits, even though I expected all my results to be 3 digits or less, I could solve my problem. Regardless of the order in which I did my calculations, the intermediate results would probably not overflow the range of the new calculator.

This is precisely how the 8087 can guarantee such accurate results. By using the Temporary Real format for all its calculations, it avoids errors that may appear in the calculation of its intermediate results. The Temporary Real format, as you can see from table 3, has a range so large that we would probably never overflow it in our calculations. Also, because it uses 64 bits for the mantissa, it can provide results that are accurate to 18 significant digits with no round-off errors.

Ease of Use

Another way in which the 8087 can guarantee accurate and standard results is by providing default exception handlers that ensure results even though an error may have occurred during one of the intermediate calculations. For example, if for some reason during the operation of one of my calculations I somehow manage to attempt to divide a number by 0, the 8087, rather than aborting the operation, executes a default response to the zerodivide exception and replaces the quotient with the most reasonable value—infinity. In this manner, the 8087 provides standard results.

$-178.125D = -10110010.001B$

$10110010.001 = 1.0110010001\ E\ 111$

$111 + 01111111 = 10000110$

sign	biased exponent	significand
1	10000110	01100100010000000000000
1100	0011 0011 0010	0010 0000 0000 0000

Table 2: *The conversion of a number with a fractional part.*

Data Type	Bits	Significant Digits	Approximate Range
Word Integer	16	4–5	$-32768 < X < 32767$
Short Integer	32	9	$-2E9 < X < 2E9$
Long Integer	64	18	$-9E18 < X < 9E18$
Packed Decimal	80	18	$-99...9 < X < 99...9$ (18 digits)
Short Real	32	6–7	$0, 1.2E-38 < X < 3.4E38$
Long Real	64	15–16	$0, 2.3E-308 < X < 1.7E308$
Temporary Real	80	19–20	$0, 3.4E-4932 < X < 1.1E4932$

Table 3: *The various data types.*

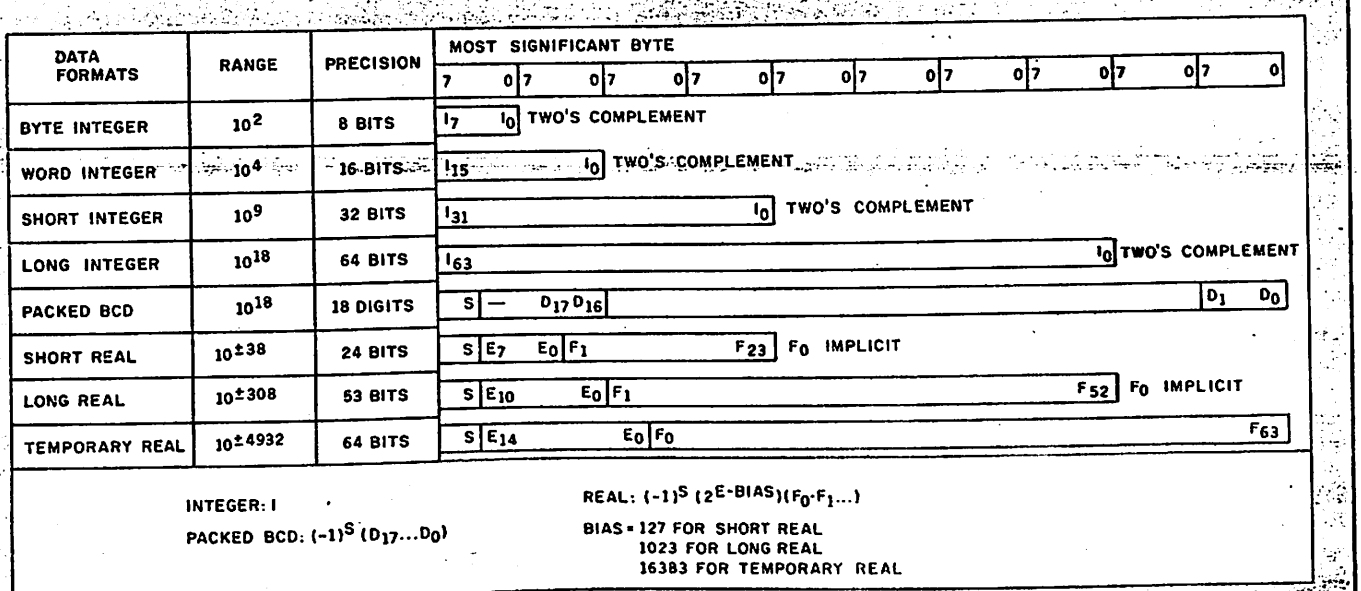


Figure 7: *The seven data formats contained in the three basic types.*