

Advanced Topics

- 17.1 Hardware Control Using I/O Ports
 - 17.1.1 Input-Output Ports
- 17.2 Intel Instruction Encoding
 - 17.2.1 Single-Byte Instructions
 - 17.2.2 Immediate Operands
 - 17.2.3 Register-Mode Instructions
 - 17.2.4 Memory-Mode Instructions
 - 17.2.5 Section Review
- 17.3 Floating-Point Arithmetic
 - 17.3.1 IEEE Binary Floating-Point Representation
 - 17.3.2 The Exponent
 - 17.3.3 Normalizing the Mantissa
 - 17.3.4 Creating the IEEE Bit Representation
 - 17.3.5 Converting Decimal Fractions to Binary Reals
 - 17.3.6 IA-32 Floating Point Architecture
 - 17.3.7 Instruction Formats
 - 17.3.8 Floating-Point Code Examples

17.1 Hardware Control Using I/O Ports

IA-32 systems offer two types of hardware input-output: *memory-mapped*, and *port-based*. When memory-mapped output is used, a program can write data to a particular memory address, and the data will be transferred to the output device. A good example is the memory-mapped video display. When you place characters in the video segment, they immediately appear on the display. Port-based I/O requires the IN and OUT instructions to read and write data to specific numbered locations called ports. Ports are connections, or gateways, between the CPU and other devices, such as the keyboard, speaker, modem, and sound card.

17.1.1 Input-Output Ports

Each input-output port has a specific number between 0 and FFFFh. A port is used when controlling the speaker, for example, by turning the sound on and off. You can communicate directly with the asynchronous adapter through a serial port by setting the port parameters (baud rate,

parity, and so on) and by sending data through the port.

The keyboard port is a good example of an input-output port. When a key is pressed, the keyboard controller chip sends an 8-bit scan code to port 60h. The keystroke triggers a hardware interrupt, which prompts the CPU to call INT 9 in the ROM BIOS. INT 9 inputs the scan code from the port, looks up the key's ASCII code, and stores both values in the keyboard input buffer. In fact, it would be possible to bypass the operating system completely and read characters directly from port 60h.

In addition to ports that transfer data, most hardware devices have ports that let you monitor the device status and control the device behavior.

IN and OUT Instructions The IN instruction inputs a byte or word from a port. Conversely, the OUT instruction outputs a byte or word to a port. The syntax for both instructions are:

```
IN accumulator, port
OUT port, accumulator
```

Port may be a constant in the range 0-FFh, or it may be a value in DX between 0 and FFFFh. *Accumulator* must be AL for 8-bit transfers, AX for 16-bit transfers, and EAX for 32-bit transfers. Examples are:

```
in  al, 3Ch          ; input byte from port 3Ch
out 3Ch, al          ; output byte to port 3Ch
mov dx, portNumber   ; DX can contain a port number
in  ax, dx           ; input word from port named in DX
out dx, ax           ; output word to the same port
in  eax, dx          ; input doubleword from port
out dx, eax          ; output doubleword to same port
```

17.1.1.1 PC Sound Program

We can write a program that uses the IN and OUT instructions to generate sound through the PC's built-in speaker. The speaker control port (number 61h) turns the speaker on and off by manipulating the Intel 8255 *Programmable Peripheral Interface* chip. To turn the speaker on, input the current value in port 61h, set the lowest 2 bits, and output the byte back through the port. To turn off the speaker, clear bits 0 and 1 and output the status again.

The Intel 8253 Timer chip controls the frequency (pitch) of the sound being generated. To use it, we send a value between 0 and 255 to port 42h. The Speaker Demo program shows how to generate sound by playing a series of ascending notes:

```
TITLE Speaker Demo Program                                (Spkr.asm)

; This program plays a series of ascending notes on
; an IBM-PC or compatible computer.
INCLUDE Irvine16.inc
speaker EQU 61h      ; address of speaker port
```

```

timer    EQU 42h           ; address of timer port
delay1   EQU 500
delay2   EQU 0D000h       ; delay between notes

.code
main PROC
    in    al,speaker        ; get speaker status
    push ax                 ; save status
    or    al,00000011b      ; set lowest 2 bits
    out   speaker,al        ; turn speaker on
    mov   al,60             ; starting pitch
L2: out   timer,al          ; timer port: pulses speaker

    ; Create a delay loop between pitches:
    mov   cx,delay1
L3: push  cx                 ; outer loop
    mov   cx,delay2
L3a:                      ; inner loop
    loop  L3a
    pop   cx
    loop  L3
    sub   al,1              ; raise pitch
    jnz   L2                ; play another note

    pop   ax                ; get original status
    and   al,11111100b      ; clear lowest 2 bits
    out   speaker,al        ; turn speaker off
    exit
main ENDP
END main

```

First, the program turns the speaker on using port 61h, by setting the lowest 2 bits in the speaker status byte:

```

    or    al,00000011b      ; set lowest 2 bits
    out   speaker,al        ; turn speaker on

```

Then it sets the pitch by sending 60 to the timer chip:

```

    mov   al,60             ; starting pitch
L2: out   timer,al          ; timer port: pulses speaker

```

A delay loop makes the program pause before changing the pitch again:

```

    mov   cx,delay1
L3: push  cx                 ; outer loop
    mov   cx,delay2
L3a:                      ; inner loop
    loop  L3a
    pop   cx

```

```
loop L3
```

After the delay, the program subtracts 1 from the period ($1 / \text{frequency}$), which raises the pitch. The new frequency is output to the timer when the loop repeats. This process continues until the frequency counter in AL equals 0. Finally, the program pops the original status byte from the speaker port and turns the speaker off by clearing the lowest two bits:

```
pop    ax                ; get original status
and    al,11111100b     ; clear lowest 2 bits
out    speaker,al        ; turn speaker off
```

17.2 Intel Instruction Encoding

One of the interesting aspects of assembly language is the way assembly instructions are translated into machine language. The topic is quite complex because of the rich variety of instructions and addressing modes available in the Intel instruction set. We will use the 8086/8088 processor as an illustrative example.

Figure 2 shows the general machine instruction format, and Table 17-1 and Table 17-2 describe the instruction fields. The opcode (operation code) field is stored in the lowest byte (at the lowest address). All remaining bytes are optional: the ModR/M field identifies the addressing mode and operands; the immed-low and immed-high fields are for immediate operands (constants); the disp-low and disp-high fields are for displacements added to base and index registers in the more complex addressing modes (e.g. $[BX+SI+2]$). Few instructions contain all of these fields; on average, most instructions are only 2-3 bytes long. (Throughout our discussions of instruction encoding, all numbers are assumed to be in hexadecimal.)

Figure 17-1 Intel 8086/8088 Instruction Format.

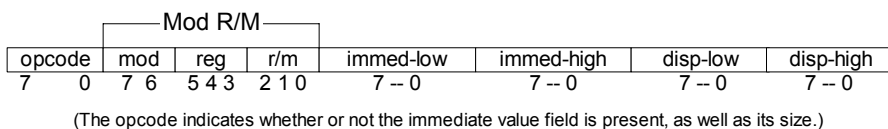


Table 17-1 Mod Field Values.

Mod	Displacement
00	DISP = 0, disp-low and disp-high are absent (unless r/m = 110).
01	DISP = disp-low sign-extended to 16 bits; disp-high is absent.
10	DISP = disp-high and disp-low are used.
11	r/m field contains a register number.

Table 17-2 R/M Field Values.

r/m	Operand
000	[BX + SI] + DISP
001	[BX + DI] + DISP
010	[BP + SI] + DISP
011	[BP + DI] + DISP
100	[SI] + DISP
101	[DI] + DISP
110	[BP] + DISP DISP-16 (for mod = 00 only)
111	[BX] + DISP

Opcode The opcode field identifies the general instruction type (MOV, ADD, SUB, and so on) and contains a general description of the operands. For example, a **MOV AL,BL** instruction has a different opcode from **MOV AX,BX**:

```
mov  al,bl           ; opcode = 88h
mov  ax,bx           ; opcode = 89h
```

Many instructions have a second byte, called the *modR/M byte*, which identifies the type of addressing mode being used. Using our sample register move instructions again, the ModR/M byte is the same for both moves because they use equivalent registers:

```
mov  al,bl           ; mod R/M = D8
mov  ax,bx           ; mod R/M = D8
```

17.2.1 Single-Byte Instructions

The simplest type of instruction is one with either no operand or an implied operand, such as AAA, AAS, CBW, LODSB, or XLAT. These instructions require only the opcode field, the value of which is predetermined by the processor's instruction set:

Instruction	Opcode
AAA	37
AAS	3F
CBW	98

LODSB	AC
XLAT	D7
INC DX	42

It might appear that the INC DX instruction slipped into this table by mistake, but the designers of the Intel instruction set decided to supply unique opcodes for certain commonly used instructions. Because of this, incrementing a register is optimized for both code size and execution speed.

17.2.2 Immediate Operands

Many instructions contain an immediate (constant) operand. For example, the machine code for **MOV AX,1** is **B8 01 00** (hexadecimal). How would the assembler build the machine language for this? First, in the Intel documentation, the encoding of the MOV instruction that moves an immediate word into a register is **B8 +rw dw**, where +rw indicates that a register code (0-7) is to be added to B8, and dw indicates that an immediate word operand follows (low byte first). The register code for AX is 0, so (rw = 0) is added to B8; the immediate value is 0001, so the bytes are inserted in reversed order. This is how the assembler generates **B8 01 00**.

What about the instruction **MOV BX,1234h**? BX is register number 3, so we add 3 to B8; we then reverse the bytes in 1234h. The machine code is generated as **BB 34 12**. Try hand-assembling a few such MOV instructions to get the hang of it, and then check your results by inspecting the listing file (.LST). The register numbers are as follows: AX/AL = 0, CX/CL = 1, DX/DI = 2, BX/BL = 3, SP/AH = 4, BP/CH = 5, SI/DH = 6, and DI/BH = 7.

17.2.3 Register-Mode Instructions

If you write an instruction that uses only the register addressing mode, the ModR/M byte identifies the register name(s). Table 17-3 identifies register numbers in the r/m field. The choice of 8-bit or 16-bit register depends upon bit 0 of the opcode field; it equals 1 for a 16-bit register and 0 for an 8-bit register.

Table 17-3 Identifying Registers in the Mod R/M Field

R/M	Register	R/M	Register
000	AX or AL	100	SP or AH
001	CX or CL	101	BP or CH
010	DX or DL	110	SI or DH
011	BX or BL	111	DI or BH

For example, let's assemble the instruction **PUSH CX**. The Intel encoding of a 16-bit register push is **50 +rw**, where +rw indicates that a register number (0-7) is added to 50h. Because CX is register number 1, the machine language would be **51**. But other register-based instructions are more complicated, particularly those with two operands. For example, the machine language for **MOV AX,DX** is **89 D8**. The Intel encoding of a 16-bit MOV from a register to any other operand is **89 /r**, where /r indicates that a ModR/M byte follows the opcode. The ModR/M byte is made up of three fields. D8, for example, contains the following bit fields:

mod	reg	r/m
11	011	000

- Bits 6-7 are the *mod* field, which tells us the addressing mode. The current operands are registers, so this field equals 11.
- Bits 3-5 are the *reg* field, which indicates the source operand. In our example, DX is register number 011.
- Bits 0-2 are the *r/m* field, which indicates the destination operand. In our example, AX is register number 000.

17.2.4 Memory-Mode Instructions

The real purpose of having the ModR/M byte is for addressing memory. Because of the rich variety of addressing modes, the ModR/M byte is a model of economy: Exactly 256 different combinations of operands may be specified by this byte. The rules for generating the bit patterns in the ModR/M byte are a trifle complex, so the Intel manuals conveniently supply a table that makes it easy to look up the values (see Table 17-4)

Table 17-4 Mod R/M Byte Values (16-Bit Segments).

Byte: Word:		AL AX 0	CL CX 1	DL DX 2	BL BX 3	AH SP 4	CH BP 5	DH SI 6	BH DI 7	
Mod	R/M	ModR/M Value								Effective Address
00	000	00	08	10	18	20	28	30	38	[BX + SI]
	001	01	09	11	19	21	29	31	39	[BX + DI]
	010	02	0A	12	1A	22	2A	32	3A	[BP + SI]
	011	03	0B	13	1B	23	2B	33	3B	[BP + DI]
	100	04	0C	14	1C	24	2C	34	3C	[SI]
	101	05	0D	15	1D	25	2D	35	3D	[DI]
	110	06	0E	16	1E	26	2E	36	3E	D16

Table 17-4 Mod R/M Byte Values (16-Bit Segments).

Byte: Word:		AL AX 0	CL CX 1	DL DX 2	BL BX 3	AH SP 4	CH BP 5	DH SI 6	BH DI 7	
01	111	07	0F	17	1F	27	2F	37	3F	[BX]
	000	40	48	50	58	60	68	70	78	[BX + SI] + D8 ^a
	001	41	49	51	59	61	69	71	79	[BX + DI] + D8
	010	42	4A	52	5A	62	6A	72	7A	[BP + SI] + D8
	011	43	4B	53	5B	63	6B	73	7B	[BP + DI] + D8
	100	44	4C	54	5C	64	6C	74	7C	[SI] + D8
	101	45	4D	55	5D	65	6D	75	7D	[DI] + D8
	110	46	4E	56	5E	66	6E	76	7E	[BP] + D8
10	111	47	4F	57	5F	67	6F	77	7F	[BX] + D8
	000	80	88	90	98	A0	A8	B0	B8	[BX + SI] + D16
	001	81	89	91	99	A1	A9	B1	B9	[BX + DI] + D16
	010	82	8A	92	9A	A2	AA	B2	BA	[BP + SI] + D16
	011	83	8B	93	9B	A3	AB	B3	BB	[BP + DI] + D16
	100	84	8C	94	9C	A4	AC	B4	BC	[SI] + D16
	101	85	8D	95	9D	A5	AD	B5	BD	[DI] + D16
	110	86	8E	96	9E	A6	AE	B6	BE	[BP] + D16
11	111	87	8F	97	9F	A7	AF	B7	BF	[BX] + D16
	000	C0	C8	D0	D8	E0	E8	F0	F8	w = AX, b = AL
	001	C1	C9	D1	D9	E1	E9	F1	F9	w = CX, b = CL
	010	C2	CA	D2	DA	E2	EA	F2	FA	w = DX, b = DL
	011	C3	CB	D3	DB	E3	EB	F3	FB	w = BX, b = BL
	100	C4	CC	D4	DC	E4	EC	F4	FC	w = SP, b = AH
	101	C5	CD	D5	DD	E5	ED	F5	FD	w = BP, b = CH
	110	C6	CE	D6	DE	E6	EE	F6	FE	w = SI, b = DH
	111	C7	CF	D7	DF	E7	EF	F7	FF	w = DI, b = BH

- a. D8 is an 8-bit displacement following the Mod R/M byte that is sign-extended and added to the effective address.

For example, let's encode the instruction **MOV [SI],AX**. Earlier, it was shown that the encoding format for a move from a 16-bit register was **89 /r**. All we have to do is look along the top of Table 5 for the AX register, and then along the right side for the effective address **[SI]**. The ModR/M byte found at the intersection of these two values in the table is **04**. Therefore, the machine instruction is **89 04**.

Let's try to figure out why that ModR/M value was chosen. Looking back at the diagram of the Intel instruction format in Figure 2, the mod field assignments are listed in a table: For memory operands having no displacement, the mod field is 00. This is true in the instruction **MOV [SI],AX**. In the same figure, the instruction format diagram shows that bits 3-5 in the ModR/M byte are the reg field (register number). AX is register 000. Finally, the r/m field value for either [SI] or [SI] + DISP is **100**. Let's put all of this together, creating a ModR/M byte value of **04**:

mod	n	r/m
00	000	100

What about the instruction **MOV [SI],AL**? The opcode for a move from an 8-bit register is **88**. The ModR/M byte, on the other hand, would be exactly the same, because AL also happens to be register number 000. The machine instruction would be **88 04**.

17.2.4.1 MOV Instruction Examples

Let's take a look at the 8-bit and 16-bit MOV instruction opcodes, shown in Table 17-5. Table 17-6 and Table 17-7 both provide supplemental information about abbreviations used in Table 17-5. Use these tables as references when hand-assembling your own MOV instructions. (If you would like to see more details such as these, refer to the IA-32 Intel Architecture Software Developer's Manual, which can be downloaded from www.intel.com.)

Finally, Table 17-8 contains a few additional examples of MOV instructions that you can assemble by hand and compare to the resulting machine code shown in the table.

Table 17-5 MOV Instruction Opcodes.

Opcode	Instruction	Description
88 /r	MOV eb,rb	Move byte register into EA byte
89 /r	MOV ew,rw	Move word register into EA word
8A /r	MOV rb,eb	Move EA byte into byte register
8B /r	MOV rw,ew	Move EA word into word register
8C /0	MOV ew,ES	Move ES into EA word
8C /1	MOV ew,CS	Move CS into EA word
8C /2	MOV ew,SS	Move SS into EA word
8C /3	MOV DS,ew	Move DS into EA word
8E /0	MOV ES,mw	Move memory word into ES
8E /0	MOV ES,rw	Move word register into ES

Table 17-5 MOV Instruction Opcodes.

Opcode	Instruction	Description
8E /2	MOV SS,mw	Move memory word into SS
8E /2	MOV SS,rw	Move register word into SS
8E /3	MOV DS,mw	Move memory word into DS
8E /3	MOV DS,rw	Move word register into DS
A0 dw	MOV AL,xb	Move byte variable (offset dw) into AL
A1 dw	MOV AX,xw	Move word variable (offset dw) into AX
A2 dw	MOV xb,AL	Move AL into byte variable (offset dw)
A3 dw	MOV xw,AX	Move AX into word register (offset dw)
B0 +rb db	MOV rb,db	Move immediate byte into byte register
B8 +rw dw	MOV rw,dw	Move immediate word into word register
C6 /0 db	MOV eb,db	Move immediate byte into EA byte
C7 /0 dw	MOV ew,dw	Move immediate word into EA word

Table 17-6 Key to Instruction Opcodes.

/n:	A ModR/M byte follows the opcode, possibly followed by immediate and displacement fields. The digit n (0-7) is the value of the reg field of the ModR/M byte.
/r:	A ModR/M byte follows the opcode, possibly followed by immediate and displacement fields.
db:	An immediate byte operand follows the opcode and ModR/M bytes.
dw:	An immediate word operand follows the opcode and ModR/M bytes.
+rb:	A register code (0-7) for an 8-bit register, which is added to the preceding hexadecimal byte to form an 8-bit opcode.
+rw:	A register code (0-7) for a 16-bit register, which is added to the preceding hexadecimal byte to form an 8-bit opcode.

Table 17-7 Key to Instruction Operands.

db	A signed value between –128 and +127. If combined with a word operand, this value is sign-extended.
dw	An immediate word value that is an operand of the instruction.
eb	A byte-sized operand, either register or memory.
ew	A word-sized operand, either register or memory.

Table 17-7 Key to Instruction Operands.

rb	An 8-bit register identified by the value (0-7).
rw	A 16-bit register identified by the value (0-7).
xb	A simple byte memory variable without a base or index register.
xw	A simple word memory variable without a base or index register.

Table 17-8 Sample MOV Instructions, with Machine Code.

Instruction	Machine Code	Addressing Mode
mov ax,[0120]	A1 20 01	direct (optimized for AX)
mov [0120],bx	89 1E 20 01	direct
mov ax,bx	89 D8	register
mov [di],bx	89 1D	indexed
mov [bx+2],ax	89 47 02	base-disp
mov [bx+si],ax	89 00	base-indexed
mov word ptr [bx+di+2],1234	C7 41 02 34 12	base-indexed-disp

17.2.5 Section Review

1. Assemble the following MOV instructions by hand, using Table 6 to obtain the opcode. We have restricted these to immediate, register, and direct operands. Write the machine language for each instruction. When you are finished, type these instructions into an ASM source file, assemble it and inspect the listing file (.LST). Check your machine code values with those generated by the assembler:

```
.data
val1  BYTE  5
val2  WORD  256
.code
mov  al,val1
mov  cx,val2
mov  dx,OFFSET val1
mov  dl,2
mov  bx,1000h
```

17.3 Floating-Point Arithmetic

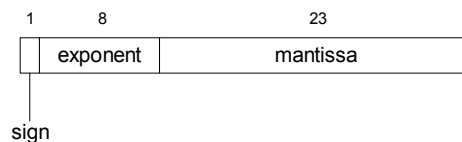
Before we begin a specific discussion of floating-point binary numbers, let's be clear on a few important terms: In the decimal number -123.154×10^5 , the **sign** is negative, the **mantissa** is 123.154, and the **exponent** is 5.

17.3.1 IEEE Binary Floating-Point Representation

The two most common floating-point binary storage formats used by Intel processors were created for Intel and later standardized by the IEEE organization:

IEEE Short Real: 32 bits	1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. Also called <i>single precision</i> .
IEEE Long Real: 64 bits	1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. Also called <i>double precision</i> .

Both formats use essentially the same method for storing floating-point binary numbers, so we will use the Short Real format as an example in this tutorial. The bits in an IEEE Short Real are arranged as follows, with the most significant bit (MSB) on the left:



17.3.1.1 The Sign

The sign of a binary floating-point number is represented by a single bit. A 1 bit indicates a negative number, and a 0 bit indicates a positive number.

17.3.1.2 The Mantissa

In Chapter 1 we introduced the concept of weighted positional notation when explaining the binary, decimal, and hexadecimal numbering systems. The same concept can be extended now to include the fractional part of a number. For example, the decimal value 123.154 can be represented by the following sum:

$$123.154 = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$$

A binary floating-point number is similar, except that we use base 2 to calculate its positional values. The floating-point binary value 11.1011 can be expressed as:

$$11.1011 = (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$$

Another way to express the values to the right of the decimal point in this number is to list them

as a sum of fractions whose denominators are powers of 2:

$$.1011 = 1/2 + 0/4 + 1/8 + 1/16$$

Which, of course, is $11/16$ (or 0.6875). A quick way to calculate this fraction is to realize that binary 1011 is the numerator and 2^4 is the denominator. Returning to our original value, binary 11.1011 is equal to decimal 3.6875. Here are additional examples:

Binary Floating-Point	Base 10 Fraction	Base 10 Decimal
11.11	$3 \frac{3}{4}$	3.75
101.0011	$5 \frac{3}{16}$	5.1875
1101.100101	$13 \frac{37}{64}$	13.578125
0.000000000000000000000001	$1/8388608$	0.00000011920928955078125

The last entry in this table is the smallest fraction that can be stored in a 23-bit mantissa.

The following table shows a few simple examples of binary floating-point numbers alongside their equivalent decimal fractions and decimal values:

Binary	Decimal Fraction	Decimal Value
.1	$1/2$.5
.01	$1/4$.25
.001	$1/8$.125
.0001	$1/16$.0625
.00001	$1/32$.03125

17.3.2 The Exponent

IEEE Short Real exponents are stored as 8-bit unsigned integers with a bias of 127. Let's use the number 1.101×2^5 as an example. The exponent (5) is added to 127 and the sum (132) is binary 10100010. Here are some examples of exponents, first shown in decimal, then adjusted, and finally in unsigned binary:

Exponent (E)	Adjusted (E + 127)	Binary
+5	132	10000100

0	127	01111111
-10	117	01110101
+128	255	11111111
-127	0	00000000
-1	126	01111110

The binary exponent is unsigned, and therefore cannot be negative. The largest possible exponent is 128. When added to 127, their sum is 255, the largest unsigned value represented by 8 bits. The approximate range is from 1.0×2^{-127} to $1.0 \times 2^{+128}$.

17.3.3 Normalizing the Mantissa

Before a floating-point binary number can be stored correctly, its mantissa must be normalized. The process is basically the same as when normalizing a floating-point decimal number. For example, decimal 1234.567 is normalized as 1.234567×10^3 by moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent).

Similarly, the floating-point binary value 1101.101 is normalized as 1.101101×2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3 . Here are some examples of normalizations:

Binary Value	Normalized As	Exponent
1101.101	1.101101	3
.00101	1.01	-3
1.0001	1.0001	0
10000011.0	1.0000011	7

You may have noticed that in a normalized mantissa, the digit 1 always appears to the left of the decimal point. In fact, the leading 1 is omitted from the mantissa in the IEEE storage format because it is redundant.

17.3.4 Creating the IEEE Bit Representation

We can now combine the sign, exponent, and normalized mantissa into the binary IEEE short real representation. Using Figure 1 as a reference, the value 1.101×2^0 is stored as sign = 0 (positive), mantissa = 101, and exponent = 01111111 (the exponent value is added to 127). The "1" to the left of the decimal point is dropped from the mantissa. Here are more examples:

Binary Value	Biased Exponent	Sign, Exponent, Mantissa
-1.11	127	1 01111111 1100000000000000000000
+1101.101	130	0 1000010 1011010000000000000000
-.00101	124	1 01111100 0100000000000000000000
+100111.0	132	0 10000100 0011100000000000000000
+.0000001101011	120	0 01111000 1010110000000000000000

17.3.5 Converting Decimal Fractions to Binary Reals

If a decimal fraction can be easily represented as a sum of fractions in the form $(1/2 + 1/4 + 1/8 + \dots)$, it is fairly easy to discover the corresponding binary real. Here are a few simple examples

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

Many real numbers do not turn out to be simple. A fraction such as $1/5$ (0.2), for example, is represented by a sum of fractions whose denominators are powers of 2. This produces a rather complex sum of fractions that is only an approximation of $1/5$.

Example: Represent 0.2 in Binary Here is the output from a program that subtracts each successive fraction from 0.2 and shows each remainder. An exact value is not found after creating the 23 mantissa bits. The result is at least accurate to 7 digits. Blank lines are shown for fractions that were too large to be subtracted from the remaining value of the number. Bit 1, for example, is equal to .5 (1/2), which could not be subtracted from 0.2.

```

starting:    0.200000000000

1
2
3   subtracting 0.125000000000
   remainder = 0.075000000000
4   subtracting 0.062500000000
   remainder = 0.012500000000
5
6
7   subtracting 0.007812500000
   remainder = 0.004687500000
8   subtracting 0.003906250000
   remainder = 0.000781250000
9
10
11  subtracting 0.000488281250
   remainder = 0.000292968750
12  subtracting 0.000244140625
   remainder = 0.000048828125
13
14
15  subtracting 0.000030517578
   remainder = 0.000018310547
16
17  subtracting 0.000015258789
   remainder = 0.000003051758
18
19  subtracting 0.000001907349
   remainder = 0.000001144409
20  subtracting 0.000000953674
   remainder = 0.000000190735
21
22
23  subtracting 0.000000119209
   remainder = 0.000000071526
Mantissa: .00110011001100110011001

```

The bit pattern in the Mantissa follows, from left to right, the progress of our subtracting fractions from the remaining value of the number. Even at step 23, after subtracting 1/23, there is a

remainder of .000000071526 which cannot be calculated. We ran out of mantissa bits!

17.3.6 IA-32 Floating Point Architecture

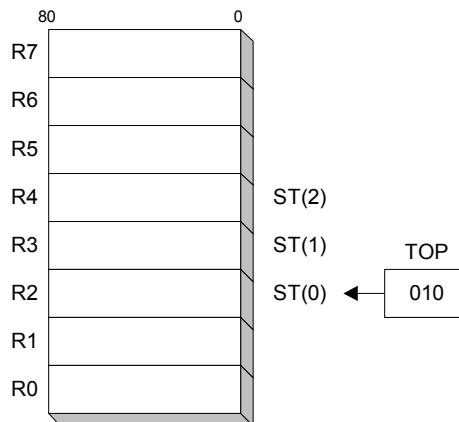
The original Intel 8086 processor was designed for integers only. This turned out to be a problem for graphics and calculation-intensive software that primarily uses floating-point calculations. It is possible to emulate floating-point arithmetic purely through software, but the performance penalty is severe. Programs such as AutoCad (by Autodesk) demanded a more powerful way to perform floating-point math.

Intel sold a floating-point coprocessor named the 8087, and upgraded it along with each processor generation. Later, as it was integrated into the main CPU, it was renamed the *Floating-Point Unit* (FPU). Originally, the FPU was a separate chip; with the introduction to the Intel486, the FPU was integrated into the main CPU.

Data Registers The FPU has eight individually addressable 80-bit registers arranged in the form of a register stack, named R0 through R7 (see Figure 17-2). All references to the registers are relative to the top of the stack, identified by a 3-bit field named **TOP** in the FPU status word.

A *load* operation decrements TOP by 1 and pushes a value on the stack. A *store* operation pops the value from the stack location identified by TOP, and increments TOP by 1. Instructions that access the stack use notation such as ST(0), ST(1), and ST(2). These operands are relative to the location pointed to by TOP. Register ST(0), often referred to as ST, is located at the top of the stack.

Figure 17-2 Floating-Point Data Register Stack



If TOP points at R0 and another value is pushed on the stack, TOP wraps around to R7. If decrementing TOP would result in overwriting unsaved data in the register stack, an exception is generated.

Numbers are held in registers while being used in calculations, in 10-byte temporary real format. When the FPU stores the result of an arithmetic operation in memory, it automatically translates the number from temporary real format to one of the following formats: integer, long integer, short real, or long real.

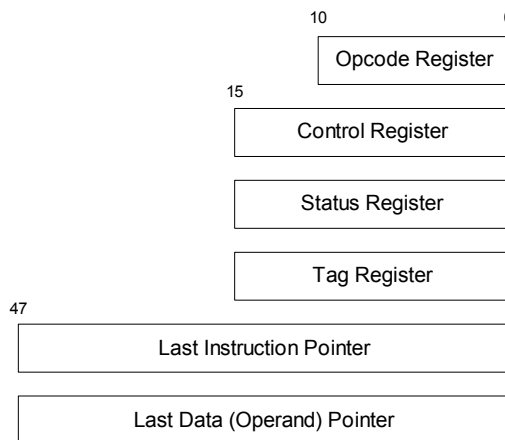
Floating-point values are transferred to and from the main CPU via memory, so you must always store an operand in memory before invoking the FPU. The FPU can load a number from memory into its register stack, perform an arithmetic operation, and store the result in memory.

The FPU has six *special-purpose* registers (see Figure 17-3):

- A 10-bit opcode register
- A 16-bit control register
- A 16-bit status registers
- A 16-bit tag word register
- A 48-bit last instruction pointer register
- A 48-bit last data (operand) pointer register

(IA-32 logical addresses in Protected mode require a total of 48 bits: 16 for the segment selector, and 32 bits for the offset.)

Figure 17-3 FPU General-Purpose Registers.



17.3.7 Instruction Formats

Floating-point instructions always begin with the letter F to distinguish them from CPU instructions. The second letter of an instruction (often B or I) indicates how a memory operand is to be interpreted: B indicates a binary-coded decimal (BCD) operand, and I indicates a binary integer operand. If neither is specified, the memory operand is assumed to be in real-number format. For example, FBLD operates on BCD numbers, FILD operates on integers, and FLD operates on

real numbers.

A floating-point instruction can have up to two operands, as long as one of them is a floating-point register. Immediate operands are not allowed, except for the FSTSW (store status word) instruction. CPU registers such as AX and EBX are not permitted as operands. Memory-to-memory operations are not permitted.

There are six basic instruction formats, shown in Table 17-9. In the **operands** column, *n* refers to a register number (0-7), *memReal* refers to a single or double precision real memory operand, *memInt* refers to a 16-bit integer, and *op* refers to an arithmetic operation. Operands surrounded by braces {...} are implied operands and are not explicitly coded. ST is used in place of ST(0), though they refer to the same register.

Table 17-9 Basic FPU Instruction Formats.

Instruction Format	Mnemonic Format	Operands (Dest, Source)	Example
Classical Stack	<i>Fop</i>	{ST(1),ST}	FADD
Classical Stack, extra pop	<i>FopP</i>	{ST(1),ST}	FSUBP
Register	<i>Fop</i>	ST(n),ST ST, ST(n)	FMUL ST(1),ST FDIV ST,ST(3)
Register, pop	<i>FopP</i>	ST(n),ST	FADDP ST(2),ST
Real Memory	<i>Fop</i>	{ST},memReal	FDIVR payRate
Integer Memory	<i>Flop</i>	{ST},memInt	FILD hours

Implied operands are not coded but are understood to be part of the operation. The operation may be one of the following:

ADD	add source to destination
SUB	Subtract source from destination
SUBR	Subtract destination from source
MUL	Multiply source by destination
DIV	Divide destination by source
DIVR	Divide source by destination

A *memReal* operand can be one of the following: a 4-byte short real, an 8-byte long real, a 10-byte packed BCD, a 10-byte temporary real, A *memInt* operand can be a 2-byte word integer, a 4-byte short integer, or an 8-byte long integer.

Classical Stack A classical stack instruction operates on the registers at the top of the stack. No explicit operands are needed. By default, ST(0) is the source operand and ST(1) is the destination. The result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the result on the top of the stack. The FADD instruction, for example, adds ST(0) to ST(1) and leaves the result at the top of the stack:

```
fld op1           ; op1 = 20.0
fld op2           ; op2 = 100.0
fadd
```

	Before		After
ST(0)	100.0	ST(0)	120.0
ST(1)	20.0	ST(1)	

Real Memory and Integer Memory The real memory and integer memory instructions have an implied first operand, ST(0). The second operand, which is explicit, is an integer or real memory operand. Here are a few examples involving real memory operands:

```
FADD mySingle      ; ST(0) = ST(0) + mySingle
FSUB mySingle      ; ST(0) = ST(0) - mySingle
FSUBR mySingle     ; ST(0) = mySingle - ST(0)
```

And here are the same instructions modified for integer operands:

```
FIADD myInteger    ; ST(0) = ST(0) + myInteger
FISUB myInteger    ; ST(0) = ST(0) - myInteger
FISUBR myInteger   ; ST(0) = myInteger - ST(0)
```

Register A register instruction uses floating-point registers as ordinary operands. One of the operands must be ST (or ST(0)). Here are a few examples:

```
FADD   st,st(1)      ; ST(0) = ST(0) + ST(1)
FDIVR  st,st(3)      ; ST(0) = ST(3) / ST(0)
FMUL   st(2),st      ; ST(2) = ST(2) * ST(0)
```

17.3.8 Floating-Point Code Examples

17.3.8.1 Example 1: Evaluating an Expression

Register pop instructions are well-suited to evaluating postfix arithmetic expressions. For example, to evaluate the following expression, we would multiply 6 by 2 and add 5 to the product:

6 2 * 5 +

Many calculators use *reverse-polish* notation, in which operands are keyed in before their operators. The algorithm for evaluating a postfix expressions is as follows:

- When reading an operand from input, push it on the stack.
- When reading an operator from input, pop the two operands located at the top of the stack, perform the selected operation on the operands, and push the result back on the stack.

The following program, for example, calculates the sum of two products:

```
TITLE FPU Expression Evaluation                                (Expr.asm)

; Implementation of the following expression:
;      (6.0 * 2.0) + (4.5 * 3.2)
; FPU instructions used.

INCLUDE Irvine32.inc

.data
array      REAL4 6.0, 2.0, 4.5, 3.2
dotProduct REAL4 ?

.code
main PROC
    finit                ; initialize FPU
    fld array             ; push 6.0 onto the stack
    fmul array+4          ; ST(0) = 6.0 * 2.0
    fld array+8           ; push 4.5 onto the stack
    fmul array+12         ; ST(0) = 4.5 * 3.2
    fadd                  ; ST(0) = ST(0) + ST(1)
    fstp dotProduct       ; pop stack into memory operand
    exit
main ENDP
END main
```

The following illustration shows a picture of the register stack after each instruction executes:

