17

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 Binary Representation
- 17.3.1 IEEE Binary Floating-Point Representation
- 17.3.2 The Exponent
- 17.3.3 Normalizing and Denormalizing
- 17.3.4 Creating the IEEE Bit Representation
- 17.3.5 Converting Decimal Fractions to Binary Reals
- 17.3.6 Rounding
- 17.3.7 Section Review

17.4 Floating-Point Unit
- 17.4.1 IA-32 Floating Point Architecture
- 17.4.2 Instruction Formats
- 17.4.3 Simple 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 / 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:

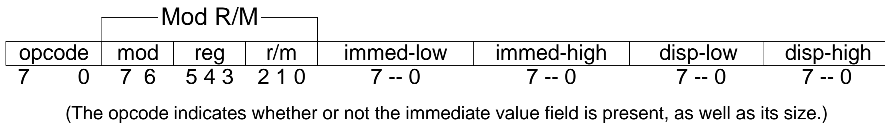
```
    pop  ax                ; get original status
    and  al,11111100b      ; clear lowest 2 bits
    out  speaker,al        ; turn speaker off
```

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, running in Real-address mode.

Figure 17-1 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.



(The opcode indicates whether or not the immediate value field is present, as well as its size.)

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 instructions require only the opcode field, the value of which is predetermined by the processor's instruction set. The following table lists a few common single-byte instructions.:

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/DL = 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 value is **51**.

Other register-based instructions, particularly those with two operands, are a bit more complicated. For example, the machine language for **MOV AX,BX** 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 (mod, reg, and r/m). A ModR/M value of D8, for example, contains the following 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, BX is register 011.
- Bits 0-2 are the *r/m* field, which indicates the destination operand. In our example, AX is register 000.

The following table lists a few more examples that use 8-bit and 16-bit register operands:

Table 17-4 Sample MOV Instruction Encodings, Register Operands.

Instruction	Opcode	mod	reg	r/m
`mov ax,dx`	8B	11	000	010
`mov al,dl`	8A	11	000	010
`mov cx,dx`	8B	11	001	010
`mov cl,dl`	8A	11	001	010

17.2.3.1 IA-32 Processor Operand-Size Prefix

Code generated for an IA-32 processor must often prepend an operand-size prefix (66h), which overrides the default segment attribute for the instruction it modifies. We can see how this works by assembling the same MOV instructions that were listed in Table 17-4. The .286 directive indicates the target processor for the compiled code, assuring (for one thing) that no 32-bit registers will be used. Alongside each MOV instruction, we show its instruction encoding:

```
.model small
.286
.stack 100h
.code
main PROC
    mov ax,dx           ; 8B C2
    mov al,dl           ; 8A C2

    mov cx,dx           ; 8B CA
    mov cl,dl           ; 8A CA
    . . .
```

(Notice that we did not INCLUDE Irvine16.inc because it automatically targets the .386 processor.)

Let's assemble the same instructions for a 386 processor, in which the default size operand is 32 bits. We'll also include some 32-bit moves. The first move (EAX, EDX) needs no prefix, but the second move (AX, DX) does:

```
.model small
.386
.stack 100h
.code
main PROC
    mov eax,edx         ; 8B C2
    mov ax,dx           ; 66 8B C2
    mov al,dl           ; 8A C2
```

```
mov ecx,edx              ; 8B CA
mov cx,dx                ; 66 8B CA
mov cl,dl                ; 8A CA
...
```

Note that 8-bit operands need no prefix. Finally, we must point out that the code generated for this example is the same for both Real-address mode and Protected-mode applications.

17.2.4 Memory-Mode Instructions

If the ModR/M byte were only used for identifying register operands, Intel instruction encoding would be realatively simple. In fact, Intel assembly language has a wide variety of memory addressing modes, causing the encoding of the ModR/M byte to be fairly complex. (This, in fact, is a common criticism of Intel machine language by proponents of reduced instruction-set designs.)

Exactly 256 different combinations of operands can be specified by the ModR/M byte, shown in Table 17-5. Here's how it works: The two bits **Mod** column indicate groups of addressing modes. In the group labeled "00" for example, there are eight possible **R/M** values (000 to 111) that identify the operands shown in the **Effective Address** column. Suppose we wished to encode **MOV AX,[SI]**; then the Mod value would be 00, and the R/M value would be 100. We know from Table 17-2 that register AX is numbered 000, so the complete ModR/M byte is **00 000 100**, or 04h:

mod	n	r/m
00	000	100

Note that the value 04h appears in the column marked AX, in row 5 (lines up with [si]). As it happens, the ModR/M byte for **MOV [SI],AL** is the same because register AL is also identified as register number 000.

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

Table 17-5 Mod R/M Byte Values (16-Bit Segments).

Byte:		AL	CL	DL	BL	AH	CH	DH	BH	
Word:		AX	CX	DX	BX	SP	BP	SI	DI	
		0	1	2	3	4	5	6	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]

Table 17-5 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	
	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
	111	07	0F	17	1F	27	2F	37	3F	[BX]
01	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
	111	47	4F	57	5F	67	6F	77	7F	[BX] + D8
10	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
	111	87	8F	97	9F	A7	AF	B7	BF	[BX] + D16
11	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.

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 17-6. Table 17-7 and Table 17-8 both provide supplemental information about abbreviations used in Table 17-6. Use these tables as references when hand-assembling your own MOV instructions. (If 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.)

Table 17-6 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
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 17-7 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 17-8 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.
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 17-9 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. We assume that **myWord** begins at offset 0102h.

Table 17-9 Sample MOV Instructions, with Machine Code.

Instruction	Machine Code	Addressing Mode
mov ax,myWord	A1 20 01	direct (optimized for AX)
mov myWord,bx	89 1E 20 01	direct
mov [di],bx	89 1D	indexed
mov [bx+2],ax	89 47 02	base-disp

Table 17-9 Sample MOV Instructions, with Machine Code.

Instruction	Machine Code	Addressing Mode
mov [bx+si],ax	89 00	base-indxed
mov word ptr [bx+di+2],1234h	C7 41 02 34 12	base-indexed-disp

17.2.5 Section Review

1. Provide op codes for the following MOV instructions:

```
.data
myByte BYTE ?
myWord WORD ?
.code
mov ax,@data
mov ds,ax                 ; a.
mov ax,bx                 ; b.
mov bl,al                 ; c.
mov al,[si]               ; d.
mov myByte,al             ; e.
mov myWord,ax             ; f.
```

2. Provide op codes for the following MOV instructions:

```
.data
myByte BYTE ?
myWord WORD ?
.code
mov ax,@data
mov ds,ax
mov es,ax                 ; a.
mov dl,bl                 ; b.
mov bl,[di]               ; c.
mov ax,[si+2]             ; d.
mov al,myByte             ; e.
mov dx,myWord             ; f.
```

3. Provide ModR/M bytes for the following MOV instructions:

```
.data
array WORD 5 DUP(?)
.code
mov ax,@data
mov ds,ax                 ; a.
mov dl,bl                 ; b.
mov bl,[di]               ; c.
mov ax,[si+2]             ; d.
mov ax,array[si]          ; e.
```

```
mov array[di],ax            ; f.
```

4. Provide ModR/M bytes for the following MOV instructions:

```
.data
array WORD 5 DUP(?)
.code
mov ax,@data
mov ds,ax
mov BYTE PTR array,5        ; a.
mov dx,[bp+5]               ; b.
mov [di],bx                 ; c.
mov [di+2],dx               ; d.
mov array[si+2],ax          ; e.
mov array[bx+di],ax         ; f.
```

5. Assemble the following instructions by hand and write the hexadecimal machine language bytes for each labeled instruction. Assume that **val1** is located at offset 0. Where 16-bit values are used, the bytes must appear in little-endian order:

```
.data
val1  BYTE  5
val2  WORD  256
.code
mov ax,@data
mov ds,ax                   ; a.
mov al,val1                 ; b.
mov cx,val2                 ; c.
mov dx,OFFSET val1          ; d.
mov dl,2                    ; e.
mov bx,1000h                ; f.
```

17.3 Floating-Point Binary Representation

Before we begin a specific discussion of floating-point binary numbers, let's be clear on a few important terms: In the decimal number -1.23154×10^5, the *sign* is negative, the *significand* is 1.23154, and the *exponent* is 5.

17.3.1 IEEE Binary Floating-Point Representation

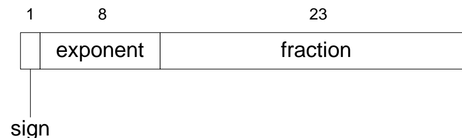
Intel processors use three floating-point binary storage formats specified in the *Standard 754-1985 for Binary Floating-Point Arithmetic* produced by the IEEE organization. The following

table describes their characteristics[1]:

Single Precision	32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional part of the significand. Approximate normalized range: 2^{-126} to 2^{127}. Also called a *short real*.
Double Precision	64 bits: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fractional part of the significand. Approximate normalized range: 2^{-1022} to 2^{1023}. Also called a *long real*.
Double Extended Precision	80 bits: 1 bit for the sign, 16 bits for the exponent, and 63 bits for the fractional part of the significand. Approximate normalized range: 2^{-16382} to 2^{16383}. Also called an *extended real*.

All three formats use essentially the same method to represent floating-point binary numbers, so we will focus on the single precision format to keep the discussion simple, shown in Figure 17-2. The 32 bits in a single precision value are arranged with the most significant bit (MSB) on the left. The segment marked *fraction* indicates the fractional part of the significand. As you might expect, the individual bytes are stored in memory in little endian order (LSB at the starting address).

Figure 17-2 Single-Precision Format.



17.3.1.1 The Sign

If the sign bit is 1, the number is negative; if the bit is 0, the number is positive. Zero is considered positive.

17.3.1.2 The Significand

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 floating-point number. For example, the decimal value 123.154 is represented by the following sum:

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

1.IA-32 Intel Architecture Software Developer's Manual, Volume 1, Chapter 4. See also: www.grouper.ieee.org/groups/754/

All digits to the left of the decimal point have positive exponents, and all digits to the right side have negative exponents.

As we found out in Chapter 1, binary floating-point numbers also use weighted positional notation. The floating-point binary value 11.1011 is 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 binary point in this number is to list them as a sum of fractions whose denominators are powers of 2, which, of course, is 11/16 (or 0.6875):

$$.1011 = 1/2 + 0/4 + 1/8 + 1/16 = 11/16$$

You can easily format the numerator (11) from the binary bit pattern 1011. The denominator is 2^4, or 16, because there are 4 significant bits to the right of the binary point. Following are additional examples that translate binary floating-point notation to base 10 fractions:

Binary Floating-Point	Base 10 Fraction
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.00000000000000000000001	1/8388608

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

For quick reference, 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

17.3.1.3 The Significand's Precision

The entire continuum of real numbers cannot be represented by floating-point numbers in a computer, because there are only a finite number of available bits in each storage format. For example, a single-precision real cannot represent real number values between binary 1.11111111111111111111111 and 10.00000000000000000000000. One such value that cannot be represented is 1.111111111111111111111111. The result of this is that not all decimal fractions can be accurately represented by IEEE real-number formats.

17.3.2 The Exponent

Single-precision exponents are stored as 8-bit unsigned integers with a bias of 127. The number's actual exponent must be added to 127. For example, the exponent of binary 1.101 x 2^5 is added to 127, producing the biased exponent 132. Here are some examples of exponents, first shown in decimal, then biased, and finally in unsigned binary:

Exponent (E)	Biased (E + 127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

The biased exponent is always positive, between 1and 254. As stated earlier, the actual exponent range is from –126 to +127. The range was chosen so that the smallest possible exponent's reciprocal will not cause an overflow.

17.3.3 Normalizing and Denormalizing

Most floating-point binary numbers are stored in *normalized* form, so as to maximize the precision of the significand. Given any floating-point binary number, you can normalize it by shifting the binary point until a single "1" appears to the left of the binary point. Following are examples:

```
1110.1      -->    1.1101
.000101     -->    1.01
1010001.    -->    1.010001
```

The exponent expresses the number of positions the binary point is moved left (positive exponent) or moved right (negative exponent). Using the previous three examples, the normalized values are:

```
1110.1      -->    1.1101 x 2^3
.000101     -->    1.01 x 2^-4
1010001.    -->    1.010001 x 2^6
```

Denormalizing a Number Denormalizing a floating-point binary number reverses the normalizing process. Shift the binary point until the exponent is zero. If the exponent is positive *n*, shift the binary point *n* positions to the right; if the exponent is negative *n*, shift the binary point *n* positions to the left, filling leading zeros if necessary. The following examples demonstrate the process:

```
1.1101 x 2^3      -->   1110.1
1.01 x 2^-4       -->   .000101
1.010001 x 2^6    -->   1010001.
```

17.3.4 Creating the IEEE Representation

Now that we understand how the sign bit, exponent bits, and significand bits are encoded, it's easy to generate a complete binary IEEE short real. Using Figure 17-2 as a reference, we place the sign bit first, the exponent bits next, and the significand bits last. For example, binary 1.101 x 2^0 is represented as:

- sign bit: 0
- exponent: 01111111
- significand: 10100000000000000000000

The biased exponent (01111111) is the binary representation of 127. All normalized significands have a 1 to the left of the binary point, so there is no need to explicitly encode the bit. Additional examples are shown in Table 17-10.

Table 17-10 Examples of Single-Precision Bit Encodings.

Binary Value	Biased Exponent	Sign, Exponent, Significand
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

17.3.4.1 Real Number Encodings

The IEEE specification includes several real-number and non-number encodings.

- Positive and negative zero
- Denormalized finite numbers
- Normalized finite numbers
- Positive and negative infinity
- Non-numeric values (NaN, known as *Not a Number*)
- Indefinite numbers

Normalized and Denormalized *Normalized finite numbers* are all the non-zero finite values that can be encoded in a normalized real number between zero and infinity.

Although it would seem that all finite non-zero floating-point numbers should be normalized, it is not possible when their values are close to zero. This happens when the FPU cannot shift the binary point to a normalized position, given the limitation posed by the range of the exponent. Suppose, for example, that the FPU computes a result of 1.0101111 X 2^{-129}, which has an exponent that is too small to be stored in a single-precision number. An underflow exception condition is generated, and the number is gradually denormalized by shifting the binary point to the right one bit at a time until the exponent reaches a valid range:

```
1.01011110000000000001111 X 2^-129
0.10101111000000000000111 X 2^-128
0.01010111100000000000011 X 2^-127
0.00101011110000000000001 X 2^-126
```

Note that some loss of precision has occurred in the significand as a result of the shifting of the binary point.

Positive and Negative Infinities Positive infinity (+∞) represents the maximum positive real number, and negative infinity (–∞) represents the maximum negative real number. You can compare infinities to to each other, so that –∞ is less than any finite number, and +∞ is greater than any finite number. The two infinities may represent a floating-point overflow condition. The result of a computation cannot be normalized because its exponent would be too large to be represented by the available number of exponent bits.

NaNs *NaNs* are bit patterns that do not represent any valid real number. The IA-32 architecture includes two types of NaNs: A *quiet NaN* can propogate through most arithmetic operations without causing an exception. A *signalling NaN* can be used to generate a floating-point invalid operation exception. A compiler might fill an uninitialized array with signalling NaN values so that any attempts to perform calculations on the array will generate an exception. The exception can then be used to execute an exception handler function. A quiet NaN can be used to hold diagnostic information created during debugging sessions. A program is free to encode any information in a NaN that it wishes. The floating-point unit does not attempt to perform any

operations on NaNs. The Intel IA-32 manual details a set of rules that determine instruction results when combinations of the two types of NaNs are used as source operands.[2]

Specific Encodings There are several specific encodings for values often encountered in floating-point operations, listed in Table 17-11. Bit positions marked with the letter x can be either 1 or 0. QNaN is a quiet NaN, and SNaN is a signalling NaN.

Table 17-11 Specific Single-Precision Encodings.

Value	Sign, Exponent, Significand
Positive Zero	0 00000000 00000000000000000000000
Negative Zero	1 00000000 00000000000000000000000
Positive Infinity	0 11111111 00000000000000000000000
Negative Infinity	1 11111111 00000000000000000000000
QNaN	x 11111111 1xxxxxxxxxxxxxxxxxxxxxx
SNaN	x 11111111 0xxxxxxxxxxxxxxxxxxxxxx[a]

a. SNaN significand field begins with 0, but at least one of the remaining bits must be 1.

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 + ...), it is fairly easy to discover the corresponding binary real. Table 17-12 a few simple examples.

Table 17-12 Examples of Decimal Fractions and Binary Reals.

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

2.IA-32 Intel Architecture Software Developer’s Manual, Vol. 1, Section 4.8.3.5.

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 exactly translate to a finite number of binary digits. 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.

Alternate Method, Using Binary Division When small decimal values are involved, an easy way to convert decimal fractions into binary is to convert both the numerator and denominator to binary, and then perform long division. For example, decimal 0.5 may be represented as the fraction 5/10. The 5 converts to binary 0101, and decimal 10 converts to binary 1010. Performing the binary long division, the quotient is .1 binary:

```
                .1
1 0 1 0 | 0 1 0 1.0
            1 0 1 0
          ---------
                  0
```

After 1010 is subtracted from the dividend, the remainder is zero, and the division stops. We will call this approach the *binary long division method*.[3]

Representing 0.2 in Binary Let's look at the output from a program that subtracts each successive fraction from 0.2 and shows each remainder. An exact value is not found after filling in all 23 bits of the significand. 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.Harvey Nice of DePaul University was kind enough to point out this method to me.

```
 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   subtracting 0.000015258789
     remainder = 0.000003051758
17
18
19   subtracting 0.000001907349
     remainder = 0.000001144409
20   subtracting 0.000000953674
     remainder = 0.000000190735
21
22
23   subtracting 0.000000119209
     remainder = 0.000000071526
significand: .00110011001100110011001
```

The bit pattern in the significand 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 bits.

Because decimal 0.2 is 2/10, we can use the binary long division method to translate into binary floating-point. We divide binary 10 by binary 1010:

```
              .0 0 1 1 0 0 1 1 (etc.)
           ___________________________
1 0 1 0  | 1 0.0 0 0 0 0 0 0 0 0
             1 0 1 0
           ---------
               1 1 0 0
               1 0 1 0
             ---------
                   1 0 0 0 0
                     1 0 1 0
                   ---------
                       1 1 0 0
                       1 0 1 0
                     ---------
                           etc.
```

The first quotient large enough to use is 10000. After dividing 1010 into 10000, the remainder is 110. Appending another zero, the new dividend is 1100. After dividing 1010 into 1100, the remainder is 10. After appending two zeros, the new dividend is 10000. This is the same dividend we started with. From this point on, the sequence of the bits in the quotient repeats (1100. . .), so we know that an exact quotient will not be found.

17.3.5.1 Converting Floating-point Decimal to IEEE Single-Precision

We can summarize the required steps when converting a decimal floating-point number (DFP) to an IEEE single-precision real as:

1. Write the integer portion of the DFP in binary, followed by a binary point.
2. Successively divide the fractional part of the DFP by negative powers of 2, notating a "1" when the fraction is divisible, and "0" when it is not. Continue this process until the fraction divides evenly, or until all 23 fractional bits have been notated. This step produces the significand.
3. Normalize the binary number produced in Steps 1 and 2.
4. Add the binary exponent to 127, producing a biased exponent.
5. Notate a 0 as the MSB if the number is positive, 1 if it is negative. Follow the sign bit with the 8 exponent bits. Follow the exponent bits with all bits to the right of the binary point of the number normalized in Step 3. Fill unused bits on the right with zeros until the significand contains 23 bits.

Example: Convert 10.75 to IEEE single-precision

1. The integer portion of +10.75 is 1010.
2. The significand is (1 x .5) + (1 x .25), which equals binary .11
3. The floating-point binary value is +1010.11. It is normalized to +1.01011 X 2^3
4. Exponent = 130, or 10000010 binary
5. IEEE = 0 10000010 01011000000000000000000

17.3.5.2 Converting IEEE Single-Precision to Decimal

We can summarize the required steps when converting a IEEE single-precision (SP) value to decimal:

1. If the MSB is 1, the number is negative; otherwise, it is positive.
2. The next 8 bits represent the exponent. Subtract binary 01111111 (decimal 127), producing the unbiased exponent. Convert the unbiased exponent to decimal.
3. The next 23 bits represent the significand. Notate a "1.", followed by the significand bits. Trailing zeros can be ignored. Create a floating-point binary number, using the significand, the sign determined in Step 1, and the exponent calculated in Step 2.
4. Denormalize the binary number produced in Step 4.
5. From left to right, use weighted positional notation to form the decimal sum of the powers of 2 represented by the floating-point binary number.

Example: Convert IEEE (0 10000010 01011000000000000000000) to Decimal

1. The number is positive.
2. The unbiased exponent is binary 00000011, or decimal 3.
3. Combining the sign, exponent, and significand, the binary number is +1.01011 X 2^3
4. The denormalized binary number is +1010.11
5. The decimal value is +10 3/4, or +10.75.

17.3.6 Rounding

The FPU always attempts to generate an infinitely accurate result from a floating-point calculation. In many cases this is impossible because the destination operand may not be able to accurately represent the calculated result. For example, suppose that a certain storage format would only permit three fractional bits. It would permit us to store values such as 1.011 or 1.101, but not 1.0101. Suppose that the precise result of a calculation produced +1.0111 (decimal 1.4375). We could either round the number up to the next higher value by adding .0001, or round it down-

ward to by subtracting .0001:

```
(a) 1.0111 --> 1.100
(b) 1.0111 --> 1.011
```

If the precise result were negative, adding –.0001 would move the rounded result closer to –∞. Subtracting –.0001 would move the rounded result closer to both zero and +∞:

```
(a) -1.0111 --> -1.100
(b) -1.0111 --> -1.011
```

The FPU lets you select one of four rounding methods:

- *Round to nearest even*: The rounded result is the closest to the infinitely precise result. If two values are equally close, the result is an even value (least significant bit = 0).
- *Round down toward –∞*: The rounded result is less than or equal to the infinitely precise result.
- *Round up toward +∞*: The rounded result is greater than or equal to the infinitely precise result.
- *Round toward zero*: Also known as *truncation*: The absolute value of the rounded result is less than or equal to the infinitely precise result.

The FPU control register contains two bits called the *RC field* that let you select which rounding method to use. *Round to nearest even* is the default, and is considered to be the most accurate and appropriate for most application programs.

The following table shows how the four rounding methods would be applied to binary +1.0111:

Method	Precise Result	Rounded
Round to nearest even	1.0111	1.100
Round toward –∞	1.0111	1.011
Round toward +∞	1.0111	1.100
Round toward zero	1.0111	1.011

Similarly, the following table shows the possible roundings of binary –1.0111:

Method	Precise Result	Rounded
Round to nearest (even)	-1.0111	-1.100
Round toward –∞	-1.0111	-1.100

Round toward +∞	-1.0111	-1.011
Round toward zero	-1.0111	-1.011

17.3.7 Section Review

1. Why doesn't the single-precision real format permit an exponent of –127?
2. Why doesn't the single-precision real format permit an exponent of +128?
3. Given a precise result of 1.010101101, round it to an 8-bit significand using the FPU's default rounding method.
4. Given a precise result of –1.010101101, round it to an 8-bit significand using the FPU's default rounding method.

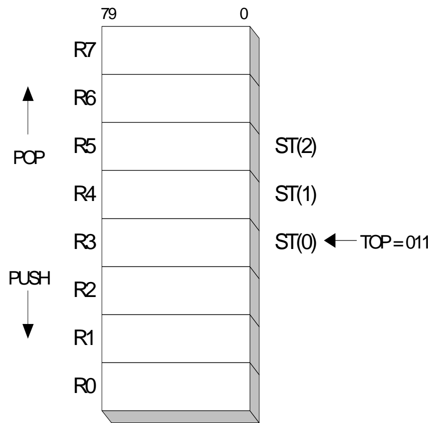
17.4 Floating-Point Unit

17.4.1 IA-32 Floating Point Architecture

The Intel 8086 processor was designed to handle only integer arithmetic. This turned out to be a problem for graphics and calculation-intensive software that primarily used floating-point calculations. It has always been 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 separate floating-point coprocessor chip named the 8087, and upgraded it along with each processor generation. With the advent of the Intel486, it was integrated into the main CPU and renamed as the *Floating-Point Unit* (FPU) .

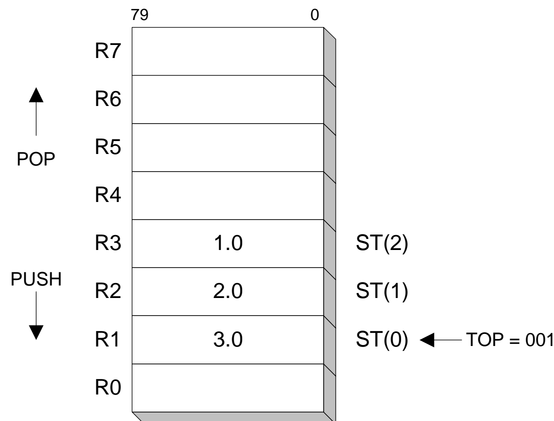
Data Registers The FPU has eight individually addressable 80-bit registers arranged in the form of a register stack, named R0 through R7 (see Figure 17-3). A 3-bit field named **TOP** in the FPU status word marks the register number that is currently the top of the stack. In Figure 17-3, for example, TOP equals binary 011, identifying R3 as the top of the stack. This stack location is also known as ST(0) (or simply ST) when writing floating-point instructions.

Figure 17-3 Floating-Point Data Register Stack



As we might expect, a *push* operation (also called *load*) decrements TOP by 1 and copies an operand into the location now marked by ST(0). If TOP equals 0 before a push, TOP will wrap around to register R7. A *pop* operation (also called *store*) first copies the data at ST(0) into an operand, and then adds 1 to TOP. If TOP equals 7 before the pop, it will wrap around to register R0. If loading a value into the stack would result in overwriting unsaved data in the register stack, a FPU exception is generated. Figure 17-4 shows the same stack after 1.0, 2.0, and 3.0 have been pushed, in that order. Note that ST(0) is now at R1.

Figure 17-4 FPU Stack, After Pushing Three Numbers.



While it is interesting to understand how the FPU implements the stack using a limited set of registers, you only need to focus on the ST(*n*) notation, where ST(0) is always the top of stack. From this point forward, we will only refer to ST(0), ST(1), and so on. Instruction operands

never refer directly to register numbers.

Floating-point operands are held in registers while being used in calculations, in 10-byte *extended real* format (also known as *temporary real*). When the FPU stores the result of an arithmetic operation in memory, it automatically translates the number from extended real format to one of the following formats: integer, long integer, single precision (short real) or double precision (long real).

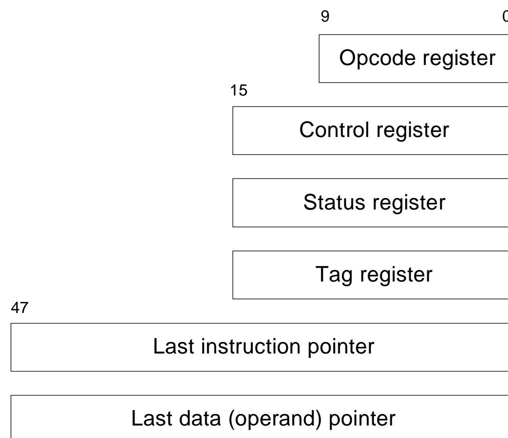
Floating-point values are transferred to and from the main CPU via memory, so you must always store an operand in memory before invoking the FPU. The FPU can load a number from memory into its register stack, perform an arithmetic operation, and store the result in memory.

The FPU has six *special-purpose* registers (see Figure 17-5):

- 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 17-5 FPU General-Purpose Registers.



17.4.2 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 17-13. 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 17-13 Basic FPU Instruction Formats.

Instruction Format	Mnemonic Format	Operands (Dest, Source)	Example
Classical Stack	F*op*	{ST(1),ST}	FADD
Classical Stack, extra pop	F*op*P	{ST(1),ST}	FSUBP
Register	F*op*	ST(n),ST ST, ST(n)	FMUL ST(1),ST FDIV ST,ST(3)
Register, pop	F*op*P	ST(n),ST	FADDP ST(2),ST
Real Memory	F*op*	{ST},memReal	FDIVR payRate
Integer Memory	FI*op*	{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)
```

17.4.3 Simple Code Examples

Let's look at a few short code examples that demonstrate the use of floating-point arithmetic instructions. You can test the examples by typing the code into a program and running it in a debugger. All modern debuggers have the capability of displaying the contents of floating-point registers and variables in a readable format.

Add Three Numbers We want to calculate the sum of three single-precision numbers that are stored in an array. FLD loads from memory into ST(0). FADD adds its operand to the number at ST(0). FSTP stores the number at ST(0) into memory, and pops the value from the stack:

```
.data
sngArray  REAL4  1.5, 3.4, 6.6
sum       REAL4  ?
.code
   fld   sngArray          ; load mem into ST(0)
   fadd [sngArray+4]       ; add mem to ST(0)
   fadd [sngArray+8]       ; add mem to ST(0)
   fstp sum                ; store ST(0) to mem
```

Divide Two Reals The following statements divide 1234.56 by 10.0, producing 123.456:

```
.data
dblOne   REAL8  1234.56
dblTwo   REAL8  10.0
dblQuot  REAL8  ?
.code
   fld  dblOne             ; load into ST(0)
   fdiv dblTwo             ; divide ST(0) by mem
   fstp dblQuot            ; store ST(0) to mem
```

Calculate a Square Root The SQRT instruction replaces the number stored at the top of the floating-point stack with its square root. The following program excerpt shows how this is done:

```
.data
sngVal1   REAL4 25.0
sngResult REAL4 ?
.code
   fld sngVal1             ; load into FP stack
   fsqrt                   ; ST(0) = square root
   fstp sngResult          ; store the result
```

Evaluating an Expression Register pop instructions are well-suited to evaluating postfix arithmetic expressions. For example, to evaluate the following expression 6 2 * 5 +, we multiply 6 by 2 and add 5 to the product. The standard algorithm for evaluating postfix expressions is:

- 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 folloing program excerpt calculates the expression (6.0 * 2.0) + (4.5 * 3.2). This is sometimes referred to as a *dot product*. You can find the complete program in *Expr.asm*:

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

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