1111011sw oo000mmm disp data

Format	Examples	Microprocessor	Clocks
TEST reg,imm	**TEST BX,3** **TEST DI,1AH** **TEST DH,44H** TEST EDX,1AB345H TEST SI,1834H		
		80286	3
		80386	2
		80486	1
		Pentium	1 or 2
TEST mem,imm	**TEST DATAS,3** TEST BYTE PTR[EDI],1AH **TEST DADDY,34H** **TEST LIST,'A'** **TEST TOAD,1834H**		
		80286	6
		80386	5
		80486	2
		Pentium	1 or 2

1010100w data

Format	Examples	Microprocessor	Clocks
TEST acc,imm	**TEST AL,3** **TEST AX,1AH** TEST EAX,34H		
		80286	3
		80386	2
		80486	1
		Pentium	1

VERR/VERW Verify read/write

00001111 00000000 oo100mmm disp

O	D	I	T	S	Z	A	P	C
					*			

Format	Examples	Microprocessor	Clocks
VERR reg	VERR CX VERR DX VERR DI	8086	—
		8088	—
		80286	14
		80386	10
		80486	11
		Pentium	7
VERR mem	VERR DATAJ VERR TESTB	8086	—
		8088	—
		80286	16
		80386	11
		80486	11
		Pentium	7

00001111 00000000 oo101mmm disp

Format	Examples	Microprocessor	Clocks
VERW reg	VERW CX VERW DX VERW DI	8086	—
		8088	—
		80286	14
		80386	15
		80486	11
		Pentium	7
VERW mem	VERW DATAJ VERW TESTB	8086	—
		8088	—
		80286	16
		80386	16
		80486	11
		Pentium	7

WAIT	Wait for coprocessor	[illegible]	[illegible]

10011011

Example	Microprocessor	Clocks
WAIT FWAIT	8086	4
	8088	4
	80286	3
	80386	6
	80486	6
	Pentium	1

WBINVD	Write-back cache invalidate data cache

00001111 00001001

Example	Microprocessor	Clocks
WBINVD	8086	—
	8088	—
	80286	—
	80386	—
	80486	5
	Pentium	2000+

WRMSR	Write to model specific register

00001111 00110000

Example	Microprocessor	Clocks
WRMSR	8086	—
	8088	—
	80286	—
	80386	—
	80486	—
	Pentium	30–45

XADD Exchange and add

00001111 1100000w 11rrrrrr

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Format	Examples	Microprocessor	Clocks
XADD reg,reg	XADD EBX,ECX XADD EDX,EAX XADD EDI,EBP	8086	—
		8088	—
		80286	—
		80386	—
		80486	3
		Pentium	3 or 4

00001111 1100000w oorrrmmm disp

Format	Examples	Microprocessor	Clocks
XADD mem,reg	XADD DATA5,ECX XADD [EBX],EAX XADD [ECX+4],EBP	8086	—
		8088	—
		80286	—
		80386	—
		80486	4
		Pentium	3 or 4

[illegible]

1000011w oorrrmmm

Format	Examples	Microprocessor	Clocks
XCHG reg,reg	**XCHG CL,DL** **XCHG BX,DX** **XCHG DH,CL** XCHG EBP,EBX XCHG EAX,EDI	[illegible]	[illegible]
		[illegible]	[illegible]
		80286	3
		80386	3
		80486	3
		Pentium	3

XCHG mem,reg **reg,mem**	**XCHG DATAJ,CL** **XCHG BYTES,CX** XCHG NUMBS,ECX XCHG [EAX],CX XCHG CL,POPS	[illegible]	[illegible]
		80286	5
		80386	5
		80486	5
		Pentium	3

10010reg Format	Examples	Microprocessor	Clocks
XCHG acc,reg **reg,acc**	**XCHG BX,AX** **XCHG AX,DI** **XCHG DH,AL** XCHG EDX,EAX XCHG SI,AX	[illegible]	[illegible]
		[illegible]	[illegible]
		80286	3
		80386	3
		80486	3
		Pentium	2

XLAT Translate

11010111 Example	Microprocessor	Clocks
XLAT	[illegible]	[illegible]
	[illegible]	[illegible]
	80286	5
	80386	3
	80486	4
	Pentium	4

000110dw oorrrmmm disp

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Format	Examples	Microprocessor	Clocks
XOR reg,reg	**XOR CL,DL** **XOR AX,DX** **XOR CH,CL** XOR EAX,EBX XOR ESI,EDI	8086	3
		8088	3
		80286	2
		80386	2
		80486	1
		Pentium	1 or 2
XOR mem,reg	**XOR DATAJ,CL** **XOR BYTES,CX** XOR NUMBS,ECX XOR [EAX],CX	8086	16 + ea
		8088	24 + ea
		80286	7
		80386	6
		80486	3
		Pentium	1 or 3
XOR reg,mem	**XOR CL,DATAL** **XOR CX,BYTES** XOR ECX,NUMBS XOR DX,[EBX+EDI]	8086	9 + ea
		8088	13 + ea
		80286	7
		80386	7
		80486	2
		Pentium	1 or 2

100000sw oo110mmm disp data

Format	Examples	Microprocessor	Clocks
XOR reg,imm	**XOR CX,3** **XOR DI,1AH** **XOR DL,34H** XOR EDX,1345H **XOR CX,1834H**	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium	1 or 3

XOR mem,imm	**XOR DATAS,3** XOR BYTE PTR[EDI],1AH **XOR DADDY,34H** **XOR LIST,'A'** **XOR TOAD,1834H**	8086	17 + ea
		8088	25 + ea
		80286	7
		80386	7
		80486	3
		Pentium	1 or 3
0010101w data			
Format	Examples	Microprocessor	Clocks
XOR acc,imm	**XOR AL,3** **XOR AX,1AH** XOR EAX,34H	8086	4
		8088	4
		80286	3
		80386	2
		80486	1
		Pentium	1

APPENDIX D

The Arithmetic Coprocessor: Data Formats, Instructions, and Programming

DATA FORMATS FOR THE ARITHMETIC COPROCESSOR

This section of the text presents the types of data used with all arithmetic coprocessor family-members. (See Table D–1 for a listing of all Intel microprocessors and their companion coprocessors.) These data types include signed-integer, BCD, and floating-point. Each has a specific use in a system, and many systems require all three data types. Note that assembly language programming with the coprocessor is often limited to modifying the coding generated by a high-level language such as C/C++. In order to accomplish any such modification, the instruction set and some basic programming concepts are required, which are presented in this appendix.

Signed Integers

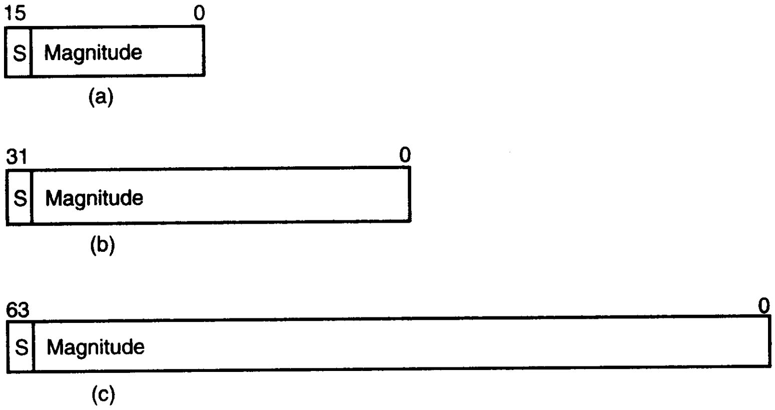
The signed integers used with the coprocessor are the same as those described in Chapter 1. When used with the arithmetic coprocessor, signed integers are 16- (word), 32- (short integer), or 64-bits (long integer) wide. The long integer is new to the coprocessor and is not described in Chapter 1, but the principles are the same. Conversion between decimal and signed-integer format is handled in exactly the same manner as for the signed integers described in Chapter 1. As you will recall, positive numbers are stored in true form with a leftmost sign-bit of 0, and negative numbers are stored in two's complement form with a leftmost sign-bit of 1.

The word integers range in value from –32,768 to +32,767, the short integer range is $\pm 2 \times 10^{+9}$, and the long integer range is $\pm 9 \times 10^{+18}$. Integer data types are found in some applications that use the arithmetic coprocessor. See Figure D–1, which shows these three forms of signed-integer data.

Data are stored in memory using the same assembler directives described and used in earlier chapters. The DW directive defines words, DD defines short integers, and DQ defines long

TABLE D–1 Microprocessor and Intel coprocessor compatibility.

Microprocessor	*Coprocessor*
8086	8087
8088	8087
80186	80187
80188	80187
80286	80287
80386SX	80387SX
80386DX	80387DX
80486SX	80487SX
80486DX	Built into microprocessor
Pentium	Built into microprocessor
Pentium Pro	Built into microprocessor
Pentium II	Built into microprocessor
Pentium III	Built into microprocessor
Pentium 4	Built into microprocessor



Note: S = sign-bit

FIGURE D–1 Integer formats for the 80X87 family of arithmetic coprocessors: (a) word, (b) short, and (c) long.

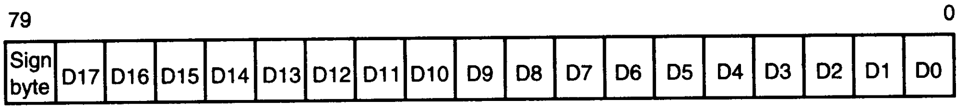


FIGURE D–2 BCD data format for the 80X87 family of arithmetic coprocessors.

integers. Example D–1 shows how several different sizes of signed integers are defined for use by the assembler and arithmetic coprocessor.

EXAMPLE D–1

```
0000  0002               DATA1  DW   +2         ;16-bit integer
0002  FFDE               DATA2  DW   -34        ;16-bit integer
0004  000004D2           DATA3  DD   +1234      ;short integer
0008  FFFFFF9C           DATA4  DD   -100       ;short integer
000C  00000000000005BA0  DATA5  DQ   +23456     ;long integer
0014  FFFFFFFFFFFFFF86   DATA6  DQ   -122       ;long integer
```

Binary-coded Decimal (BCD)

The binary-coded decimal (BCD) form requires 80 bits of memory. Each number is stored as an 18-digit packed integer in nine bytes of memory as two digits per byte. The tenth byte contains only a sign bit for the 18-digit signed BCD number. Figure D–2 shows the format of the BCD number used with the arithmetic coprocessor. Note that both positive and negative numbers are stored in true form and never in 10's complement form. The DT directive stores BCD data in the memory as illustrated in Example D–2.

EXAMPLE D–2

```
0000                 DATA1  DT     200        ;200 decimal stored as BCD
     00000000000000000200
000A                 DATA2  DT     -10        ;-10 decimal stored as BCD
     80000000000000000010
0014                 DATA3  DT     10020      ;10,020 decimal stored as BCD
     00000000000000010020
```

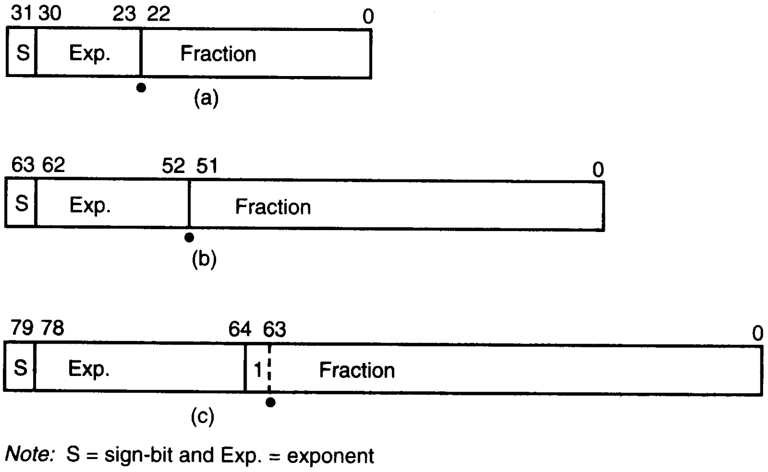
Floating-point

Floating-point numbers are often called *real numbers* because they hold signed integers, fractions, and mixed numbers. A floating-point number has three parts: a sign-bit, a biased exponent, and a significand. Floating-point numbers are written in scientific binary notation. The Intel family of arithmetic coprocessors supports three types of floating-point numbers: short (32 bits), long (64 bits), and temporary (80 bits). See Figure D–3 for the three forms of the floating-point number. Please note that the short form is also called a single-precision number and the long form is called a double-precision number. Sometimes the 80-bit temporary form is called an extended-precision number. The floating-point numbers and the operations performed by the arithmetic coprocessor conform to the IEEE-754 standard, as adopted by all major personal computer software producers. This includes Microsoft, which in 1995 stopped supporting the Microsoft floating-point format and also the ANSI floating-point standard that is popular in mainframe computer systems.

Converting to Floating-point Form. Converting from decimal to the floating-point form is a simple task that is accomplished through the following steps:

1. Convert the decimal number into binary.
2. Normalize the binary number.
3. Calculate the biased exponent.
4. Store the number in the floating-point format.

These four steps are illustrated for the decimal number 100.25_{10} in Example D–3. Here, the decimal number is converted to a single-precision (32-bit) floating-point number.



Note: S = sign-bit and Exp. = exponent

FIGURE D–3 Floating-point (real) format for the 80X87 family of arithmetic coprocessors. (a) Short (single-precision) with a bias of 7FH, (b) long (double-precision) with a bias of 3FFH, and (c) temporary (extended-precision) with a bias of 3FFFH.

EXAMPLE D–3

Step	Result
1	100.25 = 1100100.01
2	1100100.01 = 1.10010001 x 26
3	110 + 01111111 = 10000101

4	Sign = 0
	Exponent = 10000101
	Significand = 10010001000000000000000

In step three of Example D–3, the biased exponent is the exponent, a 26 or 110, plus a bias of 01111111 (7FH) or 10000101 (85H). All single-precision numbers use a bias of 7FH, double-precision numbers use a bias of 3FFH, and extended-precision numbers use a bias of 3FFFH.

In step 4 of Example D–3, the information found in the prior steps is combined to form the floating-point number. The leftmost bit is the sign-bit of the number. In this case, it is a 0 because the number is $+100.25_{10}$. The biased exponent follows the sign-bit. The significand is a 23-bit number with an implied one-bit. Note that the significand of a number 1.XXXX is the XXXX portion. The 1. is an **implied one-bit** that is only stored in the extended precision form of the floating-point number as an explicit one-bit.

Some special rules apply to a few numbers. The number 0, for example, is stored as all zeros except for the sign-bit, which can be a logic 1 to represent a negative zero. The plus and minus infinity is stored as logic 1s in the exponent with a significand of all zeros and the sign-bit that represents plus or minus. A NAN (not-a-number) is an invalid floating-point result that has all ones in the exponent with a significand that is not all zeros.

Converting from Floating-point Form. Conversion to a decimal number from a floating-point number is summarized in the following steps:

1. Separate the sign-bit, biased exponent, and significand.
2. Convert the biased exponent into a true exponent by subtracting the bias.
3. Write the number as a normalized binary number.
4. Convert it to a de-normalized binary number.
5. Convert the de-normalized binary number to decimal.

These five steps convert a single-precision floating-point number to decimal, as shown in Example D–4. Notice how the sign-bit of 1 makes the decimal result negative. Also notice that the implied one-bit is added to the normalized binary result in step 3.

EXAMPLE D–4

Step	Result

1	Sign = 1
	Exponent = 10000011
	Significand = 10010010000000000000000

2	100 = 10000011 - 01111111
3	1.1001001 x 2^4
4	11001.001
5	-25.125

Storing Floating-point Data in Memory. Floating-point numbers are stored with the assembler using the DD directive for single-precision, DQ for double-precision, and DT for extended-precision. Some examples of floating-point data storage are shown in Example D–5. The author discovered that the Microsoft version 6.0 macro assembler contains an error that does not allow a plus sign to be used with positive floating-point numbers. A +92.45 must be defined as 92.45 for the assembler to function correctly. Microsoft has assured the author that this error has been corrected in version 6.11 of MASM if the REAL4, REAL8, or REAL10 directives are used in place of DD, DQ, and DT to specify floating-point data. The assembler provides access 8087 emulator if your system does not contain a microprocessor with a coprocessor. The emulator comes with all Microsoft high-level languages or as shareware programs such as EM87. The emulator is accessed by including the OPTION EMULATOR statement immediately following the .MODEL statement in a program. Be aware that the emulator does not emulate some of the coprocessor instructions. Do not use this option if your system contains a coprocessor. In all cases, you must include the .8087, .80187, .80287, .80387, .80487, or .80587 switch to enable the generation of coprocessor instructions. Note that there is currently no switch for the Pentium 4 through Pentium Pro, but it will most likely be .80687 when Microsoft produces the next version of the assembler program.

EXAMPLE D–5

```
0000  C377999A     DATA7     DD      -247.6      ;define single-precision
0004  40000000     DATA8     DD      2.0         ;define single-precision
0008  486F4200     DATA9     REAL4   2.45E+5     ;define single-precision
000C               DATA10    DQ      100.25      ;define double-precision
      4059100000000000
00E               DATA11    REAL8   0.001235    ;define double-precision
      3F543BF727136A40
001C               DATA12    REAL10  33.9876     ;define extended-precision
      400487F34D6A161E4F76
```

Coprocessor Instructions

Although the microprocessor circuitry has not been discussed, the instruction sets of these coprocessors and their differences from the other versions of the coprocessor can be discussed. These newer coprocessors contain the same basic instructions provided by the earlier versions, with a few additional instructions.

The 80387, 80486, 80487SX, and Pentium through the Pentium 4 contain the following additional instructions: FCOS (cosine), FPREM1 (partial remainder), FSIN (sine), FSINCOS (sine and cosine), and FUCOM/FUCOMP/FUCOMPP (unordered compare). The sine and cosine instructions are the most-significant addition to the instruction set. In the earlier versions of the coprocessor, the sine and cosine is calculated from the tangent. The Pentium Pro through the Pentium 4 contain two new floating-point instructions: FCMOV (a conditional move) and FCOMI (a compare and move to flags).

Table D–2 lists the instruction sets for all versions of the coprocessor. It also lists the number of clocking periods required to execute each instruction. Execution times are listed for the 8087, 80287, 80387, 80486, 80487, and Pentium–Pentium 4. (The timings for the Pentium through the Pentium 4 are the same because the coprocessor is identical in each of these microprocessors.) To determine the execution time of an instruction, the clock time is multiplied times the listed execution time. The FADD instruction requires 70–E3 clocks for the 80287. Suppose that an 8 MHz clock is used with the 80287. The clocking period is 1/8 MHz, or 125 ns. The FADD instruction requires between 8.75 Q and 17.875 μs to execute. Using a 33 MHz (33 ns) 80486DX2, this instruction requires between 0.264 μs and 0.66 Q to execute. On the Pentium the FADD instruction requires from 1–7 clocks, so if operated at 133 MHz (7.52 ns), the FADD requires between 0.00752 Q and 0.05264 Q. The Pentium Pro through the Pentium 4 are even faster than the Pentium.

Table D–2 uses some shorthand notations to represent the displacement that may or may not be required for an instruction that uses a memory-addressing mode. It also uses the abbreviation *mmm,* to represent a register/memory addressing mode, and uses *rrr* to represent one of the floating-point coprocessor registers ST(0)–ST(7). The d (destination) bit that appears in some instruction opcodes defines the direction of the data flow, as in FADD ST,ST(2) or FADD ST(2),ST. The d bit is a logic 0 for flow toward ST, as in FADD ST,ST(2), where ST holds the sum after the addition; and a logic 1 for FADD ST(2),ST, where ST(2) holds the sum.

Also note that some instructions allow a choice of whether a wait is inserted. For example, the FSTSW AX instruction copies the status register into AX. The FNSTSW AX instruction also copies the status register to AX, but without a wait.

TABLE D–2 The instruction set of the arithmetic coprocessor (pp. 713–728).

F2XM1 $2^{ST} - 1$

11011001 11110000

Example		Clocks
F2XM1	8087	310–630
	80287	310–630
	80387	211–476
	80486/7	140–279
	Pentium–Pentium 4	13–57

FABS Absolute value of ST

11011001 11100001

Example		Clocks
FABS	8087	10–17
	80287	10–17
	80387	22
	80486/7	3
	Pentium–Pentium 4	1

FADD/FADDP/FIADD Addition

11011000 oo000mmm disp	32-bit memory (FADD)
11011100 oo000mmm disp	64-bit memory (FADD)
11011d00 11000rrr	FADD ST,ST(rrr)
11011110 11000rrr	FADDP ST,ST(rrr)
11011110 oo000mmm disp	16-bit memory (FIADD)
11011010 oo000mmm disp	32-bit memory (FIADD)

Format	Examples		Clocks
FADD FADDP FIADD	FADD DATA FADD ST,ST(1) FADDP FIADD NUMBER FADD ST,ST(3) FADDP ST,ST(2) FADD ST(2),ST	8087	70–143
		80287	70–143
		80387	23–72
		80486/7	8–20
		Pentium–Pentium 4	1–7

FCLEX/FNCLEX Clear errors

11011011 11100010

Example		Clocks
FCLEX FNCLEX	8087	2–8
	80287	2–8
	80387	11
	80486/7	7
	Pentium–Pentium 4	9

FCOM/FCOMP/FCOMPP/FICOM/FICOMP Compare

11011000 oo010mmm disp	32-bit memory (FCOM)
11011100 oo010mmm disp	64-bit memory (FCOM)
11011000 11010rrr	FCOM ST(rrr)
11011000 oo011mmm disp	32-bit memory (FCOMP)
11011100 oo011mmm disp	64-bit memory (FCOMP)
11011000 11011rrr	FCOMP ST(rrr)
11011110 11011001	FCOMPP
11011110 oo010mmm disp	16-bit memory (FICOM)
11011010 oo010mmm disp	32-bit memory (FICOM)
11011110 oo011mmm disp	16-bit memory (FICOMP)
11011010 oo011mmm disp	32-bit memory (FICOMP)

Format	Examples		Clocks
FCOM FCOMP FCOMPP FICOM FICOMP	FCOM ST(2) FCOMP DATA FCOMPP FICOM NUMBER FICOMP DATA3	8087	40–93
		80287	40–93
		80387	24–63
		80486/7	15–20
		Pentium–Pentium 4	1–8

FCOMI/FUCOMI/COMIP/FUCOMIP Compare and Load Flags

11011011 11110rrr	FCOMI ST(rrr)
11011011 11101rrr	FUCOMI ST(rrr)
11011111 11110rrr	FCOMIP ST(rrr)
11011111 11101rrr	FUCOMIP ST(rrr)

Format	Examples		Clocks
FCOM FUCOMI FCOMIP FUCOMIP	FCOMI ST(2) FUCOMI ST(4) FCOMIP ST(0) FUCOMIP ST(1)	8087	—
		80287	—
		80387	—
		80486/7	—
		Pentium–Pentium 4	—

FCMOVcc Conditional Move

11011010 11000rrr	FCMOVB ST(rrr)
11011010 11001rrr	FCMOVE ST(rrr)
11011010 11010rrr	FCMOVBE ST(rrr)
11011010 11011rrr	FCMOVU ST(rrr)
11011011 11000rrr	FCMOVNB ST(rrr)
11011011 11001rrr	FCMOVNE ST(rrr)
11011011 11010rrr	FCMOVENBE ST(rrr)
11011011 11011rrr	FCMOVNU ST(rrr)

Format	Examples	Clocks	
FCMOVB	FCMOVB ST(2)	8087	—
FCMOVE	FCMOVE ST(3)	80287	—
		80387	—
		80486/7	—
		Pentium–Pentium 4	—

FCOS Cosine of ST

11011001 11111111

Example	Clocks	
FCOS	8087	—
	80287	—
	80387	123–772
	80486/7	193–279
	Pentium–Pentium 4	18–124

FDECSTP Decrement stack pointer

11011001 11110110

Example	Clocks	
FDECSTP	8087	6–12
	80287	6–12
	80387	22
	80486/7	3
	Pentium–Pentium 4	1

FDISI/FNDISI Disable interrupts

11011011 11100001

(Ignored on the 80287, 80387, 80486/7, Pentium–Pentium 4)

Example	Clocks	
FDISI FNDISI	8087	2–8
	80287	—
	80387	—
	80486/7	—
	Pentium–Pentium 4	—

FDIV/FDIVP/FIDIV Division

11011000 oo110mmm disp	32-bit memory (FDIV)
11011100 oo100mmm disp	64-bit memory (FDIV)
11011d00 11111rrr	FDIV ST,ST(rrr)
11011110 11111rrr	FDIVP ST,ST(rrr)
11011110 oo110mmm disp	16-bit memory (FIDIV)
11011010 oo110mmm disp	32-bit memory (FIDIV)

Format	Examples	Clocks	
FDIV FDIVP FIDIV	FDIV DATA FDIV ST,ST(3) FDIVP FIDIV NUMBER FDIV ST,ST(5) FDIVP ST,ST(2) FDIV ST(2),ST	8087	191–243
		80287	191–243
		80387	88–140
		80486/7	8–89
		Pentium–Pentium 4	39–42

FDIVR/FDIVRP/FIDIVR Division reversed

11011000 oo111mmm disp	32-bit memory (FDIVR)
11011100 oo111mmm disp	64-bit memory (FDIVR)
11011d00 11110rrr	FDIVR ST,ST(rrr)
11011110 11110rrr	FDIVRP ST,ST(rrr)
11011110 oo111mmm disp	16-bit memory (FIDIVR)
11011010 oo111mmm disp	32-bit memory (FIDIVR)

Format	Examples	Clocks	
FDIVR FDIVRP FIDIVR	FDIVR DATA FDIVR ST,ST(3) FDIVRP FIDIVR NUMBER FDIVR ST,ST(5) FDIVRP ST,ST(2) FDIVR ST(2),ST	8087	191–243
		80287	191–243
		80387	88–140
		80486/7	8–89
		Pentium–Pentium 4	39–42

FENI/FNENI Disable interrupts

11011011 11100000

(Ignored on the 80287, 80387, 80486/7, Pentium–Pentium 4)

Example		Clocks
FENI FNENI	8087	2–8
	80287	—
	80387	—
	80486/7	—
	Pentium–Pentium 4	—

FFREE Free register

11011101 11000rrr

Format	Examples		Clocks
FFREE	FFREE FFREE ST(1) FFREE ST(2)	8087	9–16
		80287	9–16
		80387	18
		80486/7	3
		Pentium–Pentium 4	1

FINCSTP Increment stack pointer

11011001 11110111

Example		Clocks
FINCSTP	8087	6–12
	80287	6–12
	80387	21
	80486/7	3
	Pentium–Pentium 4	1

FINIT/FNINIT Initialize coprocessor

11011001 11110110

Example		Clocks
FINIT FNINIT	8087	2–8
	80287	2–8
	80387	33
	80486/7	17
	Pentium–Pentium 4	12–16

FLD/FILD/FBLD Load data to ST(0)

11011001 oo000mmm disp	32-bit memory (FLD)
11011101 oo000mmm disp	64-bit memory (FLD)
11011011 oo101mmm disp	80-bit memory (FLD)
11011111 oo000mmm disp	16-bit memory (FILD)
11011011 oo000mmm disp	32-bit memory (FILD)
11011111 oo101mmm disp	64-bit memory (FILD)
11011111 oo100mmm disp	80-bit memory (FBLD)

Format	Examples		Clocks
FLD FILD FBLD	FLD DATA FILD DATA1 FBLD DEC_DATA	8087	17–310
		80287	17–310
		80387	14–275
		80486/7	3–103
		Pentium–Pentium 4	1–3

FLD1 Load +1.0 to ST(0)

11011001 11101000

Example		Clocks
FLD1	8087	15–21
	80287	15–21
	80387	24
	80486/7	4
	Pentium–Pentium 4	2

FLDZ Load +0.0 to ST(0)

11011001 11101110

Example		Clocks
FLDZ	8087	11–17
	80287	11–17
	80387	20
	80486/7	4
	Pentium–Pentium 4	2

FLDPI Load ❐to ST(0)

11011001 11101011

Example		Clocks
FLDPI	8087	16–22
	80287	16–22
	80387	40
	80486/7	8
	Pentium–Pentium 4	3–5

FLDL2E Load $\log_2 e$ to ST(0)

11011001 11101010

Example		Clocks
FLDL2E	8087	15–21
	80287	15–21
	80387	40
	80486/7	8
	Pentium–Pentium 4	3–5

FLDL2T Load $\log_2 10$ to ST(0)

11011001 11101001

Example		Clocks
FLDL2T	8087	16–22
	80287	16–22
	80387	40
	80486/7	8
	Pentium–Pentium 4	3–5

FLDLG2 Load $\log_{10}2$ to ST(0)

11011001 11101000

Example		Clocks
FLDLG2	8087	18–24
	80287	18–24
	80387	41
	80486/7	8
	Pentium–Pentium 4	3–5

FLDLN2 Load $\log_e2$ to ST(0)

11011001 11101101

Example		Clocks
FLDLN2	8087	17–23
	80287	17–23
	80387	41
	80486/7	8
	Pentium–Pentium 4	3–5

FLDCW Load control register

11011001 oo101mmm disp

Format	Examples		Clocks
FLDCW	FLDCW DATA FLDCW STATUS	8087	7–14
		80287	7–14
		80387	19
		80486/7	4
		Pentium–Pentium 4	7

FLDENV Load environment

11011001 oo100mmm disp

Format	Examples		Clocks
FLDENV	FLDENV ENVIRON FLDENV DATA	8087	35–45
		80287	25–45
		80387	71
		80486/7	34–44
		Pentium–Pentium 4	32–37

FMUL/FMULP/FIMUL Multiplication

11011000 oo001mmm disp	32-bit memory (FMUL)
11011100 oo001mmm disp	64-bit memory (FMUL)
11011d00 11001rrr	FMUL ST,ST(rrr)
11011110 11001rrr	FMULP ST,ST(rrr)
11011110 oo001mmm disp	16-bit memory (FIMUL)
11011010 oo001mmm disp	32-bit memory (FIMUL)

Format	Examples	Clocks	
FMUL	FMUL DATA	8087	110–168
FMULP	FMUL ST,ST(2)	80287	110–168
FIMUL	FMUL ST(2),ST	80387	29–82
	FMULP	80486/7	11–27
	FIMUL DATA3	Pentium–Pentium 4	1–7

FNOP No operation

11011001 11010000

Example	Clocks	
FNOP	8087	10–16
	80287	10–16
	80387	12
	80486/7	3
	Pentium–Pentium 4	1

FPATAN Partial arctangent of ST(0)

11011001 11110011

Example	Clocks	
FPATAN	8087	250–800
	80287	250–800
	80387	314–487
	80486/7	218–303
	Pentium–Pentium 4	17–173

FPREM Partial remainder

11011001 11111000

Example		Clocks
FPREM	8087	15–190
	80287	15–190
	80387	74–155
	80486/7	70–138
	Pentium–Pentium 4	16–64

FPREM1 Partial remainder (IEEE)

11011001 11110101

Example		Clocks
FPREM1	8087	—
	80287	—
	80387	95–185
	80486/7	72–167
	Pentium–Pentium 4	20–70

FPTAN Partial tangent of ST(0)

11011001 11110010

Example		Clocks
FPTAN	8087	30–450
	80287	30–450
	80387	191–497
	80486/7	200–273
	Pentium–Pentium 4	17–173

FRNDINT Round ST(0) to an integer

11011001 11111100

Example		Clocks
FRNDINT	8087	16–50
	80287	16–50
	80387	66–80
	80486/7	21–30
	Pentium–Pentium 4	9–20

FRSTOR Restore state

11011101 oo110mmm disp

Format	Examples	Clocks	
FRSTOR	FRSTOR DATA FRSTOR STATE FRSTOR MACHINE	8087	197–207
		80287	197–207
		80387	308
		80486/7	120–131
		Pentium–Pentium 4	70–95

FSAVE/FNSAVE Save machine state

11011101 oo110mmm disp

Format	Examples	Clocks	
FSAVE FNSAVE	FSAVE STATE FNSAVE STATUS FSAVE MACHINE	8087	197–207
		80287	197–207
		80387	375
		80486/7	143–154
		Pentium–Pentium 4	124–151

FSCALE Scale ST(0) by ST(1)

11011001 11111101

Example	Clocks	
FSCALE	8087	32–38
	80287	32–38
	80387	67–86
	80486/7	30–32
	Pentium–Pentium 4	20–31

FSETPM Set protected mode

11011011 11100100

Example	Clocks	
FSETPM	8087	—
	80287	2–18
	80387	12
	80486/7	—
	Pentium–Pentium 4	—

FSIN Sine of ST(0)

11011001 11111110

Example		Clocks
FSIN	8087	—
	80287	—
	80387	122–771
	80486/7	193–279
	Pentium–Pentium 4	16–126

FSINCOS Find sine and cosine of ST(0)

11011001 11111011

Example		Clocks
FSINCOS	8087	—
	80287	—
	80387	194–809
	80486/7	243–329
	Pentium–Pentium 4	17–137

FSQRT Square root of ST(0)

11011001 11111010

Example		Clocks
FSQRT	8087	180–186
	80287	180–186
	80387	122–129
	80486/7	83–87
	Pentium–Pentium 4	70

FST/FSTP/FIST/FISTP/FBSTP Store

11011001 oo010mmm disp	32-bit memory (FST)
11011101 oo010mmm disp	64-bit memory (FST)
11011101 11010rrr	FST ST(rrr)
11011011 oo011mmm disp	32-bit memory (FSTP)
11011101 oo011mmm disp	64-bit memory (FSTP)
11011011 oo111mmm disp	80-bit memory (FSTP)
11011101 11001rrr	FSTP ST(rrr)
11011111 oo010mmm disp	16-bit memory (FIST)
11011011 oo010mmm disp	32-bit memory (FIST)
11011111 oo011mmm disp	16-bit memory (FISTP)
11011011 oo011mmm disp	32-bit memory (FISTP)
11011111 oo111mmm disp	64-bit memory (FISTP)
11011111 oo110mmm disp	80-bit memory (FBSTP

Format	Examples	Clocks	
FST FSTP FIST FISTP FBSTP	FST DATA FST ST(3) FST FSTP FIST DATA2 FBSTP DATA6 FISTP DATA9	8087	15–540
		80287	15–540
		80387	11–534
		80486/7	3–176
		Pentium–Pentium 4	1–3

FSTCW/FNSTCW Store control register

11011001 oo111mmm disp

Format	Examples	Clocks	
FSTCW FNSTCW	FSTCW CONTROL FNSTCW STATUS FSTCW MACHINE	8087	12–18
		80287	12–18
		80387	15
		80486/7	3
		Pentium–Pentium 4	2

FSTENV/FNSTENV Store environment

11011001 oo110mmm disp

Format	Examples	Clocks	
FSTENV FNSTENV	FSTENV CONTROL FNSTENV STATUS FSTENV MACHINE	8087	40–50
		80287	40–50
		80387	103–104
		80486/7	58–67
		Pentium–Pentium 4	48–50

FSTSW/FNSTSW Store status register

11011101 oo111mmm disp

Format	Examples	Clocks	
FSTSW FNSTSW	FSTSW CONTROL FNSTSW STATUS FSTSW MACHINE FSTSW AX	8087	12–18
		80287	12–18
		80387	15
		80486/7	3
		Pentium–Pentium 4	2–5

FSUB/FSUBP/FISUB Subtraction

11011000 oo100mmm disp	32-bit memory (FSUB)
11011100 oo100mmm disp	64-bit memory (FSUB)
11011d00 11101rrr	FSUB ST,ST(rrr)
11011110 11101rrr	FSUBP ST,ST(rrr)
11011110 oo100mmm disp	16-bit memory (FISUB)
11011010 oo100mmm disp	32-bit memory (FISUB)

Format	Examples	Clocks	
FSUB FSUBP FISUB	FSUB DATA FSUB ST,ST(2) FSUB ST(2),ST FSUBP FISUB DATA3	8087	70–143
		80287	70–143
		80387	29–82
		80486/7	8–35
		Pentium–Pentium 4	1–7

FSUBR/FSUBRP/FISUBR Reverse subtraction

11011000 oo101mmm disp	32-bit memory (FSUBR)
11011100 oo101mmm disp	64-bit memory (FSUBR)
11011d00 11100rrr	FSUBR ST,ST(rrr)
11011110 11100rrr	FSUBRP ST,ST(rrr)
11011110 oo101mmm disp	16-bit memory (FISUBR)
11011010 oo101mmm disp	32-bit memory (FISUBR)

Format	Examples	Clocks	
FSUBR FSUBRP FISUBR	FSUBR DATA FSUBR ST,ST(2) FSUBR ST(2),ST FSUBRP FISUBR DATA3	8087	70–143
		80287	70–143
		80387	29–82
		80486/7	8–35
		Pentium–Pentium 4	1–7

FTST Compare ST(0) with + 0.0

11011001 11100100

Example		Clocks
FTST	8087	38–48
	80287	38–48
	80387	28
	80486/7	4
	Pentium–Pentium 4	1–4

FUCOM/FUCOMP/FUCOMPP Unordered compare

11011101 11100rrr	FUCOM ST,ST(rrr)
11011101 11101rrr	FUCOMP ST,ST(rrr)
11011101 11101001	FUCOMPP

Format	Examples		Clocks
FUCOM FUCOMP FUCOMPP	FUCOM ST,ST(2) FUCOM FUCOMP ST,ST(3) FUCOMP FUCOMPP	8087	—
		80287	—
		80387	24–26
		80486/7	4–5
		Pentium–Pentium 4	1–4

FWAIT Wait

10011011

Example		Clocks
FWAIT	8087	4
	80287	3
	80387	6
	80486/7	1-3
	Pentium–Pentium 4	1-3

FXAM Examine ST(0)

11011001 11100101

Example		Clocks
FXAM	8087	12–23
	80287	12–23
	80387	30–38
	80486/7	8
	Pentium–Pentium 4	21

FXCH Exchange ST(0) with another register

11011001 11001rrr FXCH ST,ST(rrr)

Format	Examples		Clocks
FXCH	FXCH ST,ST(1) FXCH FXCH ST,ST(4)	8087	10–15
		80287	10–15
		80387	18
		80486/7	4
		Pentium–Pentium 4	1

FXTRACT Extract components of ST(0)

11011001 11110100

Example		Clocks
FXTRACT	8087	27–55
	80287	27–55
	80387	70–76
	80486/7	16–20
	Pentium–Pentium 4	13

FYL2X ST(1) x $\log_2$ ST(0)

11011001 11110001

Example		Clocks
FYL2X	8087	900–1100
	80287	900–1100
	80387	120–538
	80486/7	196–329
	Pentium–Pentium 4	22–111

FXL2XP1 ST(1) x $\log_2$ [ST(0) + 1.0]

11011001 11111001

Example		Clocks
FXL2XP1	8087	700–1000
	80287	700–1000
	80387	257–547
	80486/7	171–326
	Pentium–Pentium 4	22–103

Notes: d = direction, where d = 0 for ST as the destination, and d = 1 for ST as the source; rrr = floating-point register number; oo = mode; mmm = r/m field; and disp = displacement

PROGRAMMING WITH THE ARITHMETIC COPROCESSOR

This section of the chapter provides programming examples for the arithmetic coprocessor. Each example is chosen to illustrate a programming technique for the coprocessor.

Calculating the Area of a Circle

This first programming example illustrates a simple method of addressing the coprocessor stack. First, recall that the equation for calculating the area of a circle is A = □R^2. A program that performs this calculation is listed in Example D–6. Note that this program takes test data from array RAD that contains five sample radii. The five areas are stored in a second array called AREA. No attempt is made in this program to use the data from the AREA array.

EXAMPLE D–6

```
                    ;A short program that finds the area of five circles whose
                    ;radii are stored in array RAD.
                    ;
                          .MODEL SMALL
                          .386                    ;select 80386
                          .387                    ;select 80387
0000                      .DATA
0000  4015C28F      RAD   DD   2.34,5.66,9.33,234.5,23.4
        40B51EB8
        411547AE
        436A8000
        41BB3333
0014  0005 [        AREA  DD   5 DUP (?)
            00000000
                    ]
0000                      .CODE
                          .STARTUP
0010  BE 0000             MOV  SI,0               ;source element 0
0013  BF 0000             MOV  DI,0               ;destination element 0
0016  B9 0005             MOV  CX,5               ;count of 5
0019                MAIN1:
0019  D9 84 0000 R        FLD  RAD [SI]           ;radius to ST
001D D8 C8                FMUL ST;ST(0)           ;square radius
001F D9 EB                FLDPI                   ;□ to ST
0021 DE C9                FMUL                    ;multiply ST = ST x ST(1)
0023 D9 9D 0014 R         FSTP AREA [DI]          ;save area
0027 46                   INC  SI
0028 47                   INC  DI
0029 E2 EE                LOOP MAIN1
                          .EXIT
                          END
```

Although this is a simple program, it does illustrate the operation of the stack. To provide a better understanding of the operation of the stack, Figure D–4 shows that the contents of the stack after each instruction of Example D–6 executes. Note only one pass through the loop is illustrated because the program calculates five areas and each pass is identical.

The first instruction loads the contents of memory location RAD [SI], one of the elements of the array, to the top of the stack. Next, the FMUL ST, ST(0) instruction squares the radius on the top of the stack. The FLDPI instruction loads n to the stack top. The FMUL instructions use the classic stack addressing mode to multiply ST by ST(1). After the multiplication, the prior values of ST and ST(1) are removed from the stack and the product replaces them at the top of the stack. Finally, the FSTP [DI] instruction copies the top of the stack, the area, to an array memory location AREA and clears the stack.

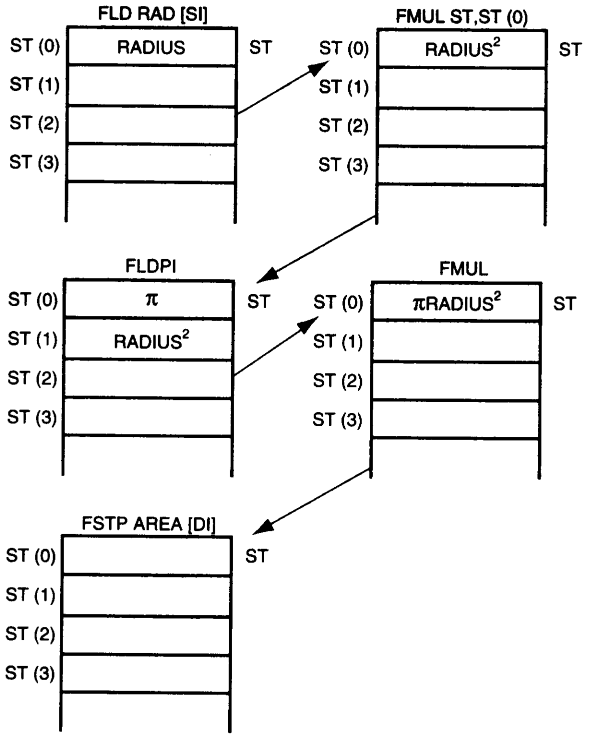


FIGURE D–4 Operation of the stack for Example D–4. Note that the stack is shown after the execution of the indicated instruction.

Notice how care is taken to always remove all stack data. This is important because if data remain on the stack at the end of the procedure, the stack top will no longer be register 0. This could cause problems because software assumes that the top of the stack is register 0. Another way of ensuring that the coprocessor is initialized is to place the FINIT (initialization) instruction at the start of the program.

Finding the Resonant Frequency

An equation commonly used in electronics is the formula for determining the resonant frequency of an LC circuit. The equation solved by the program illustrated in Example D–7.

$$Fr \quad \frac{1}{2 \boxed{} \rightarrow \overline{LC}}$$

This example uses L1 for the inductance L, C1 for the capacitor C, and RESO for the resultant resonant frequency.

EXAMPLE D–7

```
                    ;A sample program that finds the resonant frequency of an LC
                    ;tank circuit.
                    ;
                          .MODEL SMALL
                          .386
                          .387
0000                      .DATA
```

```
0000  00000000    RESO   DD    ?                    ;resonant frequency
0004  358637BD    L1     DD    0.000001             ;inductance
0008  358637BD    C1     DD    0.000001             ;capacitance
000C  40000000    TWO    DD    2.0                  ;constant
0000                     .CODE
                         .STARTUP
0010  D9 06 0004 R       FLD L1                     ;get L
0014  D8 0E 0008 R       FMUL C1                    ;find LC

0018  D9 FA              FSQRT                      ;find ←LC

001A  D8 0E 000C R       FMUL TWO                   ;find 2←LC

001E  D9 EB              FLDPI                      ;get □
0020  DE C9              FMUL                       ;get 2□←LC

0022  D9 E8              FLD1                       ;get 1
0024  DE F1              FDIVR                      ;form 1/(2□←LC)

0026  D9 1E 0000 R       FSTP RESO                  ;save frequency
                         .EXIT
                         END
```

Notice the straightforward manner in which the program solves this equation. Very little extra data manipulation is required because of the stack inside the coprocessor. Notice how the constant TWO is defined for the program and how the DIVRP, using classic stack addressing, is used to form the reciprocal. If you own a reverse-polish entry calculator, such as those produced by Hewlett-Packard, you are familiar with stack addressing. If not, using the coprocessor will increase your experience with this type of entry.

Finding the Roots Using the Quadratic Equation

This example illustrates how to find the roots of a polynomial expression ($ax^2 + bx + c = 0$) by using the quadratic equation. The quadratic equation is

$$b\pm \frac{\overline{\leftarrow b^2 - 4ac}}{2a}$$

Example D–8 illustrates a program that finds the roots (R1 and R2) for the quadratic equation. The constants are stored in memory locations A1, B1, and C1. Note that no attempt is made to determine the roots if they are imaginary. This example tests for imaginary roots and exits to DOS with a zero in the roots (R1 and R2), if it finds them. In practice, imaginary roots could be solved for and stored in a separate set of result memory locations.

EXAMPLE D–8

```
                         ;A program that finds the roots of a polynomial equation using
                         ;the quadratic equation. Note imaginary roots are indicated if
                         ;both root 1 (R1) and root 2 (R2) are zero.
                         ;
                                .MODEL SMALL
                                .386
                                .387
0000                            .DATA
0000  40000000           TWO    DD   2.0
0004  40800000           FOUR   DD   4.0
0008  3F800000           A1     DD   1.0
000C  00000000           B1     DD   0.0
0010  C1100000           C1     DD   -9.0
0014  00000000           R1     DD   ?
0018  00000000           R2     DD   ?
```

```
0000                           .CODE
                               .STARTUP
0010  D9 EE                    FLDZ
0012  D9 16 0014 R             FST  R1               ;clear roots
0016  D9 1E 0018 R             FSTP R2
001A  D9 06 0000 R             FLD  TWO
001E  D8 0E 0008 R             FMUL A1               ;form 2a
0022  D9 06 0004 R             FLD  FOUR
0026  D8 0E 0008 R             FMUL A1
002A  D8 0E 0010 R             FMUL C1               ;form 4ac
002E  D9 06 000C R             FLD  B1
0032  D8 0E 000C R             FMUL B1               ;form b^2
0036  DE E1                    FSUBR                 ;form b^2-4ac
0038  D9 E4                    FTST                  ;test b^2-4ac for zero
003A  9B DF E0                 FSTSW AX              ;copy status register to AX
003D  9E                       SAHF                  ;move to flags
003E  74 0E                    JZ   ROOTS1           ;if b^2-4ac is zero
0040  D9 FA                    FSQRT                 ;find square root of b^2-4ac
0042  9B DF E0                 FSTSW AX
0045  A9 0001                  TEST AX,1             ;test for invalid error (negative)
0048  74 04                    JZ  ROOTS1
004A  DE D9                    FCOMPP                ;clear stack
004C  EB 18                    JMP  ROOTS2 ;end
004E                    ROOTS1:
004E  D9 06 000C R             FLD  B1
0052  D8 E1                    FSUB ST,ST(1)
0054  D8 F2                    FDIV ST,ST(2)
0056  D9 1E 0014 R             FSTP R1               ;save root 1
005A  D9 06 000C R             FLD  B1
005E  DE C1                    FADD
0060  DE F1                    FDIVR
0062  D9 1E 0018 R             FSTP R2               ;save root 2
0066                    ROOTS2:
                               .EXIT
                               END
```

Using a Memory Array to Store Results

The next programming example illustrates the use of a memory array and the scaled-indexed addressing mode to access the array. Example D–9 shows a program that calculates 100 values of inductive reactance. The equation for inductive reactance is XL = 2□FL. In this example, the frequency range is from 10 Hz to 1000 Hz for F and an inductance of 4H. Notice how the instruction FSTP DWORD PTR CS:[EDI+4*ECX] is used to store the reactance for each frequency, beginning with the last at 1000 Hz and ending with the first at 10 Hz. Also notice how the FCOMP instruction is used to clear the stack just before the RET instruction.

EXAMPLE D–9

```
                      ;A program that calculates the inductive reactance of L
                      ;at a frequency range of 10Hz to 1000Hz and stores them
                      ;in array XL. Note that the increment is 10Hz.
                           .MODEL SMALL
                           .386
                           .387
0000                       .DATA
0000  40800000    L        DD  4.0                ;4.0H test value
0004  0064 [      XL       DD  100 DUP (?)
            00000000
                  ]
0194  447A0000    F        DD  1000.0             ;start at 1000Hz
0198  41200000    TEN      DD  10.0               ;increment of 10Hz
```

```
0000                            .CODE
                                .STARTUP
0010   66| B9 00000064          MOV   ECX,100              ;load count
0016   66| BF 00000000 R        MOV   EDI,OFFSET XL-4      ;address result
001C   D9 EB                    FLDPI                      ;get ❐
001E   D8 C0                    FADD  ST,ST(0)             ;form 2❐
0020   D8 0E 0000 R             FMUL  L                    ;form 2❐L
0024                      L1:
0024   D9 06 0194 R             FLD   F                    ;get F
0028   D8 C9                    FMUL  ST,ST(1)
002A   67& D9 1C 8F             FSTP  DWORD PTR [EDI+4*ECX]
002E   D9 06 0194 R             FLD   F
0032   D8 26 0198 R             FSUB  TEN                  ;change frequency
0036   D9 1E 0194 R             FSTP  F
003A   E2 E8                    LOOP  L1
003C   D8 D9                    FCOMP
                                .EXIT
                                END
```

Displaying a Single-precision Floating-point Number

This section of the text shows how to take the floating-point contents of a 32-bit single-precision floating-point number and display it on the video display. The procedure displays the floating-point number as a mixed number with an integer part and a fractional part, separated by a decimal point. In order to simplify the procedure, a limit is placed on the display size of the mixed number so the integer portion is a 32-bit binary number and the fraction is a 24-bit binary number. The procedure will not function properly for larger or smaller numbers.

Example D–10 lists a program that calls a procedure for displaying the contents of memory location NUMB on the video display at the current cursor position. The procedure first tests the sign of the number and displays a minus sign for a negative number. After displaying the minus sign, if needed, the number is made positive by the FABS instruction. Next, it is divided into an integer and fractional part and stored at WHOLE and FRACT. Notice how the FRNDINT instruction is used to round (using the chop mode) the top of the stack to form the whole number part of NUMB. The whole number part is then subtracted from the original number to generate the fractional part. This is accomplished with the FSUB instruction that subtracts the contents of ST(1) from ST.

EXAMPLE D–10

```
                     ;A program that displays the floating-point contents of NUMB
                     ;as a mixed decimal number.
                           .MODEL SMALL
                           .386
                           .387
0000                       .DATA
0000   C50B0200      NUMB  DD    -2224.125             ;test data
0004   0000          TEMP  DW    ?
0006   00000000      WHOLE DD    ?
000A   00000000      FRACT DD    ?
0000                       .CODE
                             .STARTUP
0010   E8 000B                CALL DISP              ;display NUMB
                             .EXIT
                     ;
                     ;A procedure that displays the ASCII code from AL.
                     ;
0017                 DISPS PROC NEAR

0017   B4 06                 MOV   AH,6             ;display AL
0019   8A D0                 MOV   DL,AL
001B   CD 21                 INT   21H
001D   C3                    RET
```

```
001E                    DISPS ENDP
                        ;
                        ;A procedure that displays the floating-point contents of NUMB
                        ;in decimal form.
                        ;
001E                    DISP  PROC NEAR

001E  9B D9 3E 0004 R        FSTCW TEMP                   ;save current control word
0023  81 0E 0004 R 0C00      OR    TEMP,0C00H             ;set rounding to chop
0029  D9 2E 0004 R           FLDCW TEMP
002D  D9 06 0000 R           FLD   NUMB ;get NUMB
0031  D9 E4                  FTST                         ;test NUMB
0033  9B DF E0               FSTSW AX                     ;status to AX
0036  25 4500                AND   AX,4500H               ;get C3, C2, and C0
                             .IF AX == 0100H
003E  B0 2D                     MOV AL,'-'
0040  E8 FFD4                   CALL DISPS
0043  D9 E1                     FABS
                             .ENDIF
0045  D9 C0                  FLD ST
0047  D9 FC                  FRNDINT                      ;get integer part
0049  DB 16 0006 R           FIST WHOLE
004D  DE E1                  FSUBR
004F  D9 E1                  FABS
0051  D9 1E 000A R           FSTP FRACT                   ;save fraction
0055  66| A1 0006 R          MOV  EAX,WHOLE
0059  66| BB 0000000A        MOV  EBX,10
005F  B9 0000                MOV  CX,0
0062  53                     PUSH BX
                             .WHILE 1                     ;divide until quotient = 0
0063  66| BA 00000000           MOV    EDX,0
0069  66| F7 F3                 DIV    EBX
006C  80 C2 30                  ADD    DL,30H
006F  52                        PUSH   DX
                             .BREAK     .IF EAX == 0
0075  41                        INC    CX
                                .IF    CX == 3
007B  6A 2C                        PUSH    ','
007D  B9 0000                      MOV     CX,0
                                   .ENDIF
                             .ENDW
                             .WHILE  1                    ;display whole number part
0082  5A                        POP    DX
                             .BREAK  .IF DX == BX
0087  8A C2                     MOV    AL,DL
0089  E8 FF8B                   CALL   DISPS
                             .ENDW
008E  B0 2E                  MOV AL,'.'                   ;display decimal point
0090  E8 FF84                CALL DISPS
0093  66| A1 000A R          MOV EAX,FRACT
0097  9B D9 3E 0004 R        FSTCW TEMP                   ;save current control word
009C  81 36 0004 R 0C00      XOR TEMP,0C00H               ;set rounding to nearest
00A2  D9 2E 0004 R           FLDCW TEMP
00A6  D9 06 000A R           FLD FRACT
00AA  D9 F4                  FXTRACT
00AC  D9 1E 000A R           FSTP FRACT
00B0  D9 E1                  FABS
00B2  DB 1E 0006 R           FISTP WHOLE
00B6  66| 8B 0E 0006 R       MOV  ECX,WHOLE
00BB  66| A1 000A R          MOV  EAX,FRACT
00BF  66| C1 E0 09           SHL  EAX,9
00C3  66| D3 D8              RCR  EAX,CL
                             .REPEAT
00C6  66| F7 E3                 MUL  EBX
```

```
00C9  66| 50                   PUSH EAX
00CB  66| 92                   XCHG EAX,EDX
00CD  04 30                    ADD  AL,30H
00CF  E8 FF45                  CALL DISPS
00D2  66| 58                   POP  EAX
                            .UNTIL EAX == 0
00D9  C3                    RET

00DA               DISP   ENDP
                   END
```

The last part of the procedure displays the whole number part, followed by the fractional part. The techniques are the same as introduced earlier in the text—dividing a number by ten and displaying the remainders in reverse order converts and displays an integer. A multiplication by 10 converts a fraction to decimal for displaying. Note that the fractional part may contain a rounding error for certain values. This occurs because the number has not been adjusted to remove the rounding error that is inherent in floating-point fractional numbers.

Reading a Mixed Number from the Keyboard

If floating-point arithmetic is used in a program, a method of reading the number from the keyboard and converting it to a floating-point number must be developed. The procedure listed in Example D–11 reads a signed mixed number from the keyboard and converts it to a floating-point number located at memory location NUMB.

EXAMPLE D–11

```
                   ;A program that reads a mixed number from the keyboard.
                   ;The result is stored at memory location NUMB as a
                   ;double-precision floating-point number.
                   ;
                           .MODEL SMALL
                           .386
                           .387
0000                       .DATA
0000  00           SIGN    DB   ?                   ;sign indicator
0001  0000         TEMP1   DW   ?                   ;temporary storage
0003  41200000     TEN     DD   10.0 ;10.0
0007  00000000     NUMB    DD   ?                   ;result
0000                       .CODE
                   GET     MACRO                    ;;read key macro
                           MOV AH,1
                           INT 21H
                           ENDM
                           .STARTUP
0010  D9 EE                FLDZ                     ;clear ST
                           GET                      ;read a character
                           .IF  AL == '+'           ;test for +
001A  C6 06 0000 R 00        MOV  SIGN,0            ;clear sign indicator
                           GET
                           .ENDIF
                           .IF  AL == '-'           ;test for -
0027  C6 06 0000 R 01        MOV  SIGN,1            ;set sign indicator
                             GET
                           .ENDIF
                           .REPEAT
0030  D8 0E 0003 R           FMUL  TEN              ;multiply result by 10
0034  B4 00                  MOV   AH,0
0036  2C 30                  SUB   AL,30H           ;convert from ASCII
0038  A3 0001 R              MOV   TEMP1,AX
003B  DE 06 0001 R           FIADD TEMP1            ;add it to result
                             GET                    ;get next character
                             .UNTIL  AL < '0' || AL > '9'
                             .IF AL == '.'          ;do if -
004F  D9 E8                    FLD1                 ;get one
                               .WHILE 1
0051  D8 36 0003 R                 FDIV  TEN
                                   GET
                               .BREAK .IF AL < '0' || AL > '9'
0061  B4 00                        MOV   AH,0
0063  2C 30                        SUB   AL,30H     ;convert from ASCII
0065  A3 0001 R                    MOV   TEMP1,AX
0068  DF 06 0001 R                 FILD  TEMP1
006C  D8 C9                        FMUL ST,ST(1)
006E  DC C2                        FADD ST(2),ST
0070  D8 D9                        FCOMP
                               .ENDW
0074  D8 D9                    FCOMP                ;clear stack
                             .ENDIF
                             .IF SIGN == 1
007D  D9 E0                    FCHS                 ;make negative
                             .ENDIF
007F  D9 1E 0007 R           FSTP NUMB              ;save result
                             .EXIT
                             END
```

Unlike other examples in this chapter, Example D–11 uses some of the high-level language constructs presented in earlier chapters to reduce its size. Here, the sign is first read from the keyboard, if present, and saved for later use as a 0 for positive and a 1 for negative, in adjusting the sign of the resultant floating-point number. Next, the integer portion of the number is read. The .REPEAT–.UNTIL loop is used to read the number until something other than a number (0–9) is typed. This portion terminates with a period, space, or carriage return. If a period is typed, then the procedure continues and reads a fractional part by using an .IF–.ENDIF construct. If a space or carriage return is entered, the number is converted to floating-point form and stored at NUMB. The .WHILE–.ENDW loop converts the fractional part of the number. The whole number portion is converted with a multiply by 10, and the fractional portion is converted with a divide by 10.

QUESTIONS AND PROBLEMS

1. List the three types of data that are loaded or stored in memory by the coprocessor.
2. List the three integer data types, the range of the integers stored in them, and the number of bits allotted to each.
3. Explain how a BCD number is stored in memory by the coprocessor.
4. List the three types of floating-point numbers used with the coprocessor and the number of binary bits assigned to each.
5. Convert the following decimal numbers into single-precision floating-point numbers:
   (a) 28.75
   (b) 624
   (c) –0.615
   (d) +0.0
   (e) –1000.5
6. Convert the following single-precision floating-point numbers into decimal:
   (a) 11000000 11110000 00000000 00000000
   (b) 00111111 00010000 00000000 00000000
   (c) 01000011 10011001 00000000 00000000
   (d) 01000000 00000000 00000000 00000000

(e) 01000001 00100000 00000000 00000000

(f) 00000000 00000000 00000000 00000000

7. Explain what the coprocessor does when a normal microprocessor instruction executes.
8. Describe how the FST DATA instruction functions. Assume that DATA is defined as a 64-bit memory location.
9. What does the FILD DATA instruction accomplish?
10. Form an instruction that adds the contents of register 3 to the top of the stack.
11. Describe the operation of the FADD instruction.
12. Choose an instruction that subtracts the contents of register 2 from the top of the stack and stores the result in register 2.
13. What is the function of the FBSTP DATA instruction?
14. What is the difference between the FTST instruction and FXAM?
15. Which instruction stores the environment?
16. What does the FSAVE instruction save?
17. Develop a procedure that finds the area of a rectangle ($A = L \times W$). Memory locations for this procedure are single-precision floating-point locations *A*, *L*, and W.
18. Write a procedure that finds the capacitive reactance (*XC* = 1/2□*FC* 1,). Memory locations for this procedure are single-precision floating-point locations *XC, F*, and *C1*.
19. Develop a procedure that generates a table of square roots for the integers 2 through 10. The results must be stored as single-precision floating-point numbers in an array called ROOTS.
20. Develop a procedure that finds the cosine of a single-precision floating-point number. The angle, in degrees, is passed to the procedure in EAX and the cosine is returned in EAX. Recall that FCOS finds the cosine of an angle expressed in radians.
21. Develop a procedure that takes the single-precision contents of register EBX times n and stores the result in register EBX as a single-precision floating-point number. You must use memory to accomplish this task.
22. Write a procedure that raises a single-precision floating-point number X to the power Y. Parameters are passed to the procedure with *EAX* = X and *EBX* = Y. The result is passed back to the calling sequence in ECX.

APPENDIX E

RISC MICROPROCESSORS

THE PERENNIAL ISSUE

The microprocessor, which is the CPU of the Personal Computer, is widely being used and is expected to perform computation at a rapid speed. In order to do this, the microprocessors have to execute the instructions very fast. These devices are sequential devices and they may be speeded up by increasing their clock rate. On the other side, the microprocessor chips are becoming more and more smaller in size due to advanced nano-meter VLSI technology. The faster a microprocessor is operated the more power it consumes. This is because the rise time of the digital signal is affected by the stray capacitances which develop across the interconnects in the IC. The close interconnects have a potential difference which introduces a capacitance between them. The digital signals are supposed to have a very short rise time but these capacitances increase the rise time as the clock frequency increases. To make the capacitor charge fast, it should be charged through at a higher current. This increases the power consumption of the microprocessor. Microprocessors which operate faster than 100 MHz need a heat sink to dissipate their heat. As the clock frequency further goes up, the heating will also increase and the cooling has to be more elaborate. Further, there will be a final limit at which the VLSI realization becomes difficult, if not impossible. This could happen when interconnect distances become closer lesser and closer in the IC, as could be encountered in nano/pico meter VLSI design. What is the other way to make microprocessors compute faster? This is a question which has many answers. However the following sections provide a solution.

EVOLUTION OF RISC MICROPROCESSORS

Sometime in the early 1970s a study was conducted on the instruction set used in computers. This brought to light that more than 50 percent of the instructions used in machine code/assembly language programs were of data transfer. Further, about 20-25 percent of the entire instruction set of the computer was used in the programs. This left 75-80 percent of the instruction set as an appendage which could be removed. This concept was initiated at IBM Research by John Cocke, and led to the view that a computer based on this philosophy would certainly be less complex. The reduced complexity would result in a smaller number of electronic elements to build the CPU. This would in turn result in enhancement of the speed of execution of the programs as the instructions are less complex.

By the early 1980s this philosophy was reasonably understood and it was called Reduced Instruction Set Computer. This was usually referred by the acronym RISC and was coined by a computer architecture researcher and Professor at University of California Berkley, David Patterson. This was during a study of the architecture of Instruction Set termed as the Instruction Set Architecture (ISA). The concept behind ISA was to reduce the control logic and enhance the CPU performance by improved Instruction processing, pipelining, and faster clock rates. The ISA concept was to reduce the burden of the hardware which was to be shared by the compiler.

Simpler Decoder

RISC philosophy meant that the Instruction Set is reduced to the most commonly used instructions. Further, the memory accesses were reduced to only two types: LOAD and STORE, which are instructions fetching data from memory directly, avoiding complex addressing modes. This also reduced the execution time as most data transfer was between the CPU registers.

Single Cycle Instruction Execution

Most of the instructions were made of equal length and preferably the word size of the microprocessor. Many of them were made to execute in one clock instruction cycle. This made the execution unit simpler and it was fully used without wait states.

The instruction decoder is a complex piece of combinational logic and sometimes a stored program component is also present. This reduction in complex instructions followed by making most of the instructions of same width and single cycle execution, the decoder became less complex and occupied lesser space on the chip. Further in the RISC philosophy the stored program component (micro-program ROM) was eliminated making it hard wired control. The RISC architecture brought about a change in CPU design and there was now more space on the microprocessor chip.

Registers, Memory and Cache's (Larger Number of Registers than the CISC)

To further utilize this advantage of more free area more registers were added. The registers were used by the data movement instructions and there were very few memory references needed. The registers are also memory elements but with a difference. The registers are directly accessed by the microprocessor without having to generate effective address as in the case of memory. So access to the register takes far less time than a memory access, thus contributing to faster data movement. Accessing memory takes extra time as the address lines are to be activated through a clock sequence.

Cache is also memory with a difference; it is faster and matches the microprocessor speed as it is made of the same material. Further, there is a faster separate bus used for accessing it. Registers are used by the microprocessor to facilitate the instructions performing their operations, whereas cache stores information as user data or program instruction which is under immediate use. RAM is the slowest and cheapest of the other two and is used like the cache but the instructions and data it contains need not be in immediate use. Registers are usually small in number and so cannot be used for user data. This makes the caches the next best choice as compared to the slower RAM and ROM. Registers are directly accessed by the microprocessor almost at the speed of the microprocessor whereas cache needs a bus. That is the reason why in classification of memory registers are L0; cache, if on chip, is L1; off chip cache is L2; and so on. The L0 memory is faster than L1, and as the sequence goes the speeds become lower and so do the costs and quantity.

Pipelining

Pipelining, another way to speed up computing, is enhanced by the property of RISC instructions which have equal execution time. Multiple instructions are overlapped during execution and so the execution units are fully utilized. This is somewhat like a mason (execution unit) receiving bricks (instructions) from the line of labourers (machine cycle units). It is a technology used on modern microprocessors to enhance their performance. The computer pipeline is divided in stages (machine cycle). Each stage completes a part of an instruction in parallel. The processor works on different stages of the instruction at the same time; more instructions can be executed in a shorter period of time. Pipelining increases instruction through put and not reduce their execution time.

The processing of any instruction by the microprocessor can be broken down into a series of four simple steps, which each instruction in the code stream goes through in order to be executed:

Fetch : Read the instruction from memory.
Decode : Decode the instruction and fetch the source operand(s).
Execute : Perform the operation specified by the instruction.
Write : Store the result(s) in the destination location.

The above four steps get repeated over and over again until the processor finishes executing the program. An instruction starts out in the fetch phase, moves to the decode phase, then to the execute phase, and finally to the write phase. The sequence of events for the above case is given in the figure below. Four instructions are in progress at any time, assuming that there are four separate hardware units and each are capable of doing their tasks simultaneously. Notice that without the pipelining, it would have taken 8 cycles for 2 instructions to execute. With pipeline, it takes only 5 cycles.

The pipeline size depends on the machine cycles that the instruction is made up of. It is most effective in RISC systems as most instructions execute in same time and single clock.

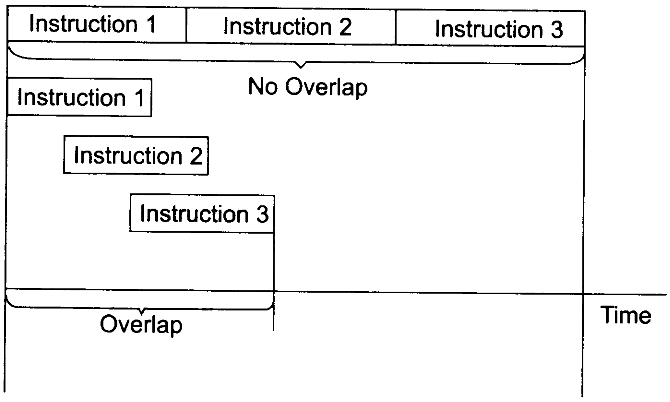


FIGURE E–1 Instruction execution without and with pipelining.

Cycle Number	1	2	3	4	5	6	7	8
Instruction								
i	**F**	**D**	**E**	**W**				
i +1					**F**	**D**	**E**	**W**

(a)

Cycle Number	1	2	3	4	5	6	7	8
Instruction								
i	F	D	E	W				
i +1		F	D	E	W			
i +2			F	D	E	W		
i +3				F	D	E	W	
i +4					F	D	E	W

(b)

Cycle Number	1	2	3	4	5	6	7	8
Instruction								
i	F	D	E	W				
i +1		F	X	X	D	E	W	
i +2					F	D	E	W

(c)

FIGURE E–2 (a) Execution without Pipeline (b) Pipelined Execution (c) Effect of fetch operation taking more than one clock cycle (X is the wait cycle).

Super Scalar RISC design

The advantage of more space on the chip, the fixed instruction size and the single cycle of execution brought about the possibility of having multiple execution units which operate at the same time in parallel, and so execute

multiple instructions in one clock cycle. This was a good development but had an issue. If, for example, there are three execution units E1, E2, and E3 which process three sequential instructions I1, I2, and I3 in one clock cycle, then the CPU would have three times more speed of computation. Let, for example, instruction I2's data (available in a register), after its execution, be used by instruction I3. If, for any reason, instruction I2 gets delayed in execution (one possibility may be that I2 is a floating point instruction which got delayed because the arithmetic coprocessor encountered an error), the instruction I3 will take the un-updated data, thus causing a computational error. This effect is called score-boarding. Such a situation is to be avoided, so sticking to equal instruction lengths and execution times is a must. One way to prevent such error is to ensure that the parallel execution is completed only after the registers are updated.

Branch Prediction Logic

Whenever a microprocessor encounters a conditional jump, a number of associated things happen. First, if the condition is true and a branch is to be taken then the cache, both instruction and data cache, are to be flushed out and new values loaded into the caches. If there was a way to predict a while before the conditional branch instruction is taken then the microprocessor could ignore or flush the cache depending on taking or not taking the branch. This is based on a statistical approach and has been known to give 70-90% accuracy. This saves a lot of computational time.

High Level Language Support

The RISC microprocessors can give good support to high-level languages as they have abundant registers. One such technique is called Register Windowing found from the beginning in the SPARC processor.

Register Window. A large number of registers are available on the SUN SPARC and these registers are divided into various groups. Each group containing four sets of eight registers INREGS, OUTREGS, LOCALREGS, and GLOBALREGS. The operating system assigns a group of registers to processes which are running for facilitating their passing parameters through the INREGS and OUTREGS. Procedures are assigned individual window consisting of these register file sets. The base of this window is pointed at by the Current Window Pointer which is located in the CPU's status register.

If the currently running procedure is assigned the register window RW1 which contains register K, k+, k+2, K+n where n is the n+1th register, a current window pointer (CWP) points to the base of the current register window RW1. The CWP would be incremented for assigning the next procedure's register window. The calling and called procedure have overlapping register sets. The OUTREGS of previous procedure will overlap the INREGS of the current procedure. This makes parameter passing very efficient. The overlapping is done as illustrated in the figure given below.

RISC processors have good regularization factor, so abundant registers can be incorporated. Due to use of register windows no external memory reference was needed, not even the stack, which speeded up the program execution time. This, however, put a burden on the operating system to assign the registers to the various processes while loading them for execution which is done only once so there is no detrimental affect.

Further, the instructions were of three operands; that is, for adding two numbers and storing the result, this could be an instruction- ADD R5, R6, R7. Add the contents of R5 to R6 and put the result in R7. All registers had access to ALU so they could function as the accumulator.

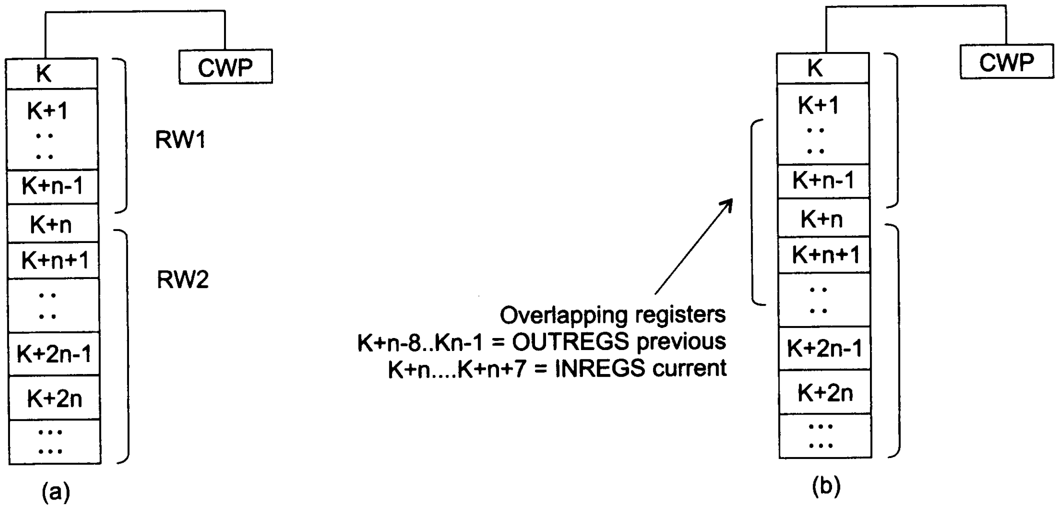


FIGURE E–3 The assignment of registers to windows. (b) Overlapping of INREGS and OUTREGS of two adjacent windows RW1 and RW2.

RISC VERSUS CISC

Realisation

RISC AND VLSI Realization. The decoder of a RISC processor is simpler and so is its control unit. This brings about an advantage in the VLSI realization of the RISC CPU's.

TABLE E–1 Advantages and disadvantages of RISC and CISC

CISC	*RISC*
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
No extra features on CPU chip	Many varieties of peripheral controllers on chip
Hardware is costly and compiler is simpler	Hardware is simpler and compiler is complex
Hardware is repetitive in cost systems are more expensive	Software is not repetitive in cost so systems are less expensive
Takes time to market	Quicker to the market

Control Area. The area occupied by the control unit, which is called the control area is reduced to a great extent, and thus provides more space. It is usually around 10% of the VLSI chip area. Typically CISC processors have about 60 to 70% control area of the VLSI chip. This saving in space may be utilized to provide built-in peripheral controllers and coprocessors and other features like caches, I/O ports , memory management units, timers. In fact, many microcontrollers adopt RISC features as they need lots of space on chip for the peripheral components.

Regularization Factor

VLSI EDA tools provided a large variety of building blocks. These building blocks are drawn out of a library. The library of these blocks consists of registers, arithmetic and logic units, counters, shift registers and many more modules. The ratio of the total number of devices (except ROM) on the chip and the number of drawn devices is called the realization factor. The registers form the maximum number of devices in control area so the regularization factor is very high. Generally a higher realization factor reduces the cost of VLSI design.

Increased Registers. The reduced control area, the increased regularization factor further permit use of larger number of registers.

CURRENT RISC MICROPROCESSORS: AN OVERVIEW

The RISC computer system designers were always faced with an opposition that the advantages of RISC were not commensurate with the sacrifice of the CISC. In spite of this strong controversy, quite a lot of microprocessors have been designed based on RISC philosophy. They are as follows:

IBM	RISC801
Inte	i860
Digital equipment corporation	ALPHA 21064
IBM/Motorola /Apple	POWERPC
MIPS	Rx000
Advanced Risc Microprocessors	ARM7

Most of these, though based on RISC philosophy, do not adhere strictly to the principles of RISC. Their success through RISC is via proven performance. Many of the above mentioned microprocessors are super scalar and have two issue pipelines. The RISC processors are used in workstations and real-time embedded systems, and are universal in every field of application. Many microcontrollers are nowadays RISC based.

Overview of Some RISC Processors

Power PC. This is a microprocessor developed by three companies Apple-IBM-Motorola in 1991 based on the RISC architecture. This alliance of three companies was called the AIM. This is an offshoot of the IBM's Performance Optimization with Enhanced RISC which was an offshoot of IBM's ROMP (Research Office Micro Processor). The features of POWERPC are as follows:

❷ User Instruction and Architecture
❷ Virtual Environment Architecture
❷ Operating Environment Architecture
❷ Operates in USER and SUPERVISOR modes

❷ Has 32 general purpose registers called GPR0 PR32. MSB is numbered as 0 and the LSB is numbered as 31
❷ Supports both BIG and LITTLE endian byte ordering
❷ Has a 64-bit time base register instead of a real-time clock
❷ Has built-in floating point unit
❷ Has an elaborate Condition Code register
❷ POWERPC is quite popular in the embedded systems arena.

SUN SPARC

Manufactured by SUN Microsystems Inc. and kept as open architecture in their web site, it has versions starting from "low cost" to thirty times "low cost". The starting versions have over 50 registers scalable as the name says to 300 or so. The windowing of the registers is the HLL support imparted by SPARC. It is a 32-bit processor and also has certain registers which are hardwired for zero. The instruction formats supported are CALL, BRANCH, and OPERATE (Register to register). It has a built-in floating point processor.

ALPHA 21064

Digital Equipment Corporations contribution to the high-end RISC market is the DEC ALOHA 2106 series. This is 64-bit processor, supports the RISC philosophy, and has features like instruction level parallelism, two issue superscaler, floating point unit, and memory management unit. It has two sets of 64-bit registers, thirty-two in number, one set for FPU and the other for instruction unit. Two registers are hardwired to zero. Support VAX floating point format along with the IEEE format. The instruction formats are five:

❷ Memory Instruction Format
❷ Branch Instruction Format
❷ Operate Instruction Format
❷ Floating Point Instruction Format
❷ PAL Code Instruction Format

PAL is a library of extended processor functions built in and called Privilege Architecture Library.

SUMMARY

The need for speeding up the computation is expressed as a perennial issue. The clock can make the CPU speed up but has a lot of related issues cropping up. Another way is to streamline the organization of the CPU to be able to speed up computation. RISC was a philosophy which started surfacing in early 1970s and matured in the minds of the computer designers by 1980s. First of all the instruction set was reduced making the control unit and the instruction decoder simpler than in the CISC processor. Further, the microprogramming was removed and hardwired logic was used bringing further space on chip. Next it was thought that the execution of the instruction should be done in a single cycle while also keeping the size of the instruction the same as the CPU's width. Avoid memory access as much as possible and have only LOAD STORE instruction for memory accesses and incorporate many CPU registers for data manipulation. This made the control area small on the chip and the registers stepped up the regularization factor. This paved the way for adding peripheral to the chip like floating point units, I/O ports and so on. Instruction pipelining could be most efficient in RISC as the instructions execute in equal times. In some cases, multiple execution units were used and these turned up as super scalar RISC processors. However, the problems in multiple execution units like score-boarding would not affect an RISC CPU. To support the faster execution of compilers and facilitate parameter passing between procedures register windowing was incorporated. The various manufacturers of RISC CPU's were introduced followed by an overview of some typical RISC processors.

QUESTIONS AND PROBLEMS

1. Why does the CPU get hot when the clock frequency is above 100 MHz?
2. What brought about the concept of reduced instruction set?
3. Why is it that complex instructions are more in an instruction set of general purpose microprocessor?
4. Which type of instructions are used the most often?
5. What do the LOAD and STORE instructions perform with respect to the RISC principles?
6. Why do the control unit and the decoder become smaller in RISC machines?
7. Why are many registers used in RISC machines?
8. What is the meaning of Control Area and Regularization Factor?
9. Why does pipelining work effectively in RISC processors?
10. If there are 6 machine cycles, what is the depth of the pipeline?
11. What is score-boarding?
12. Based on question number 10, how many clocks will be needed to complete 6 instructions?
13. What is register windowing?
14. Give the name of the RISC processor which has priviledge architecture library on chip.
15. Which RISC processor supports both BIG ENDIAN and LITTLE ENDIAN?
16. Which RISC processor uses the concept of register windowing?

APPENDIX F

Answers to Selected Even-Numbered Questions and Problems

CHAPTER 1

2. Herman Hollerith
4. Konrad Zuse
6. ENIAC
8. Augusta Ada Bryon
10. A machine that stores its program in the memory system.
12. Over 100,000,000
14. 16M
16. 1993
18. 1999
20. Millions of instructions per second
22. 1 or a 0
24. 1024K
26. About 1,000,000
28. 640K
30. 1M
32. 80386, 80486, and Pentium (Note that the Pentium Pro through Pentium 4 address 64G)
34. (a) 13.25 (b) 57.1875 (c) 43.3125 (d) 7.0625
36. (a) 163.1875 (b) 297.75 (c) 172.859375 (d) 4,011.1875 (e) 3,000.0578125
38. (a) 0.101_2, 0.5_8, and $0.D_{16}$ (b) 0.00000001_2, 0.002_8, and 0.01_{16} (c) 0.10100001_2, 0.502_8, and $0.A1_{16}$ (d) 0.11_2, 0.6_8, and $0.C_{16}$ (e) 0.1111_2, 0.74_8, and $0.F_{16}$
40. (a) C2 (b) 10FD (c) B.C (d) 10 (e) 8BA
42. (a) 0111 1111 (b) 0101 0100 (c) 0101 0001 (d) 1000 0000
44. (a) 46 52 4F 47 (b) 41 72 63 (c) 57 61 74 65 72 (d) 57 65 6C 6C
46. MESS DB ‘What time is it?’
48. (a) 0000 0011 1110 1000 (b) 1111 1111 1000 1000 (c) 0000 0011 0010 0000 (d) 1111 0011 0111 0100
50. (a) 34 12 (b) 22 A1 (c) 00 B1
52. DATA2 DW 123AH
54. (a) –128 (b) +51 (c) 110 (d) –118
56. (a) 0 01111111 10000000000000000000000
    (b) 1 10000010 01010100000000000000000
    (c) 0 10000101 10010001000000000000000
    (d) 1 10001001 00101100000000000000000

CHAPTER 2

2. 16 bits
4. EBX
6. The instruction pointer is used by the microprocessor to locate the next instruction in a program.
8. No
10. The interrupt flag (I)
12. In the real mode, a segment register locates the start of a 64K-byte memory segment.
14. (a) 12000H (b) 21000H (c) 24A00H (d) 25000H (e) 3F12DH
16. ES:DI
18. Stack segment plus the stack offset
20. (a) 12000H (b) 21002H (c) 26200H (d) A1000H (e) 2CA00H
22. Any location between and including 000000H–FFFFFFH
24. The segment register contains a selector that chooses a descriptor from either the local or global descriptor table. It also contains the requested privilege level.
26. A00000H-A01000H
28. Base address = 00280000H and end address = 00290FFFH
30. 3
32. 64K bytes
34.

03	10
90	00
00	00
2F	FF

36. The LDT is addressed through the local descriptor table register.
38. The program invisible register is the cache portion of the segment register, the task register, and also the descriptor table register.
40. 4K bytes
42. 1024
44. Page directory 000H and page table entry 200H
46. The TLB stores the last 22 linear-to-physical address translation from the paging unit.

CHAPTER 3

2. AH, AL, BH, BL, CH, CL, DH, and DL
4. EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI
6. You may not mix register sizes.
8. (a) MOV EDX,EBX (b) MOV CL,BL (c) MOV BX,SI (d) MOV AX,DS (e) MOV AH,AL
10. #
12. .CODE
14. Opcode field
16. This is an assembly language directive that returns control to DOS.
18. The .STARTUP directive loads the DS register with the segment address of the data segment.
20. The [ ] symbols denote indirect addressing.
22. Memory-to-memory transfers are not allowed.
24. MOV WORD PTR [DI],3

26. The MOV BX,DATA instruction copies the contents of a data segment memory location DATA into BX, while the MOV BX,OFFSET DATA instruction loads BX with the offset address of DATA.
28. Nothing is wrong with this instruction; this is an alternate to MOV AL,[BX+SI].
30. (a) 11750H (b) 11950H (c) 11700H
32. BP/EBP
34.

```
FIELDS STRUC

F1      DW ?
F2      DW ?
F3      DW ?
F4      DW ?
F5      DW ?
FIELDS ENDS
```

36. Direct, indirect, and stack
38. The intrasegment jump is within a segment, while the intersegment jump is to any location in the memory system.
40. 32-bit
42. Short
44. JMP BX
46. Two bytes are stored for a 16-bit PUSH and 4 by a 32-bit PUSH.
48. AX, CX, DX, BX, SP, BP, DI and SI
50. PUSHFD

CHAPTER 4

2. The W bit selects either a byte (W = 0) or a word/doubleword (W = 1). The D bit selects the direction of flow between the register field and the register/memory field.
4. DL
6. DS:[BX+DI]
8. MOV AX,DI
10. 8B 77 02
12. You should never change CS without also changing IP. This instruction would most likely cause the system to crash because only the segment portion of the address of the next instruction is changed.
14. 16-bit
16. AX, CX, DX, BX, SP, BP, SI, and DI
18. (a) The PUSH AX instruction pushes the contents of AX onto the stack. (b) The POP ESI instruction removes a 32-bit number from the stack and places it into ESI. (c) The PUSH [BX] instruction pushes the 16-bit contents of the data segment memory location addressed by BX onto the stack. (d) The PUSHFD instruction pushes the EFLAG register onto the stack. (e) The POP DS instruction removes a 16-bit number from the stack and places it into the DS register. (f) The PUSHD 4 instruction places a 32-bit number 4 onto the stack.
20. The PUSH EAX instruction places bits 31–24 of EAX into memory location 20FFH, bits 23–16 into 20FEH, bits 15–8 into 20FDH, and bits 7–0 into 20FCH. After the data are stored, the contents of SP are decremented to four, which results in 20FCH.
22. One possibility is 200H in both registers.
24. The MOV using the OFFSET is more efficient than the LEA instruction for all microprocessors prior to the Pentium.
26. This instruction loads DS and BX with the 32-bit number stored at memory location NUMB.
28.

```
MOV  BX,NUMB
MOV  DX,BX
MOV  SI,BX
```

30. The CLD instruction clears direction and the STD instruction sets it.
32. The LODSB instruction copies the contents of the data segment memory location addressed by SI into AL. Next, the contents of SI are either incremented or decremented by a 1, depending on the state of the direction flag.
34. The OUTSB instruction outputs the contents of the data segment memory location addressed by SI to the I/O port addressed by DX. Next, the contents of SI are either incremented or decremented by 1, depending on the state of the direction flag.
36.
```
MOV  SI,OFFSET SOURCE
MOV  DI,OFFSET DEST
MOV  CX,12
REP  MOVSB
```
38 The XLAT instruction adds the contents of AL to the contents of BX to form a data segment offset address that loads a byte of data from a table into AL.
40. The IN AL,12H instruction inputs a byte of data from I/O port 0012H into AL.
42. The segment override prefix allows the default segment to be changed to any other segment.
44.
```
XCHG  AX,BX
XCHG  CX, DX
XCHG  SI,DI
```
46. The DB directive is used to define or store bytes, DW defines words, and DD defines doublewords.
48. The EQU directive allows one label to be equated to another or a constant.
50. The .MODEL directive identifies the type of memory model used to generate a program.
52. Full segment definitions
54. The PROC directive indicates the start of a procedure and the ENDP directive indicates the end of a procedure.
56.
```
COPS  PROC  FAR
   MOV   AX,CS:DATA1
   MOV   BX,AX
   MOV   CX,AX
   MOV   DX,AX
   MOV   SI,AX
   RET
COPS  ENDP
```

CHAPTER 5

2. You may not mix register sizes.
4. Sum = 3100H, C = 0, A = 1, S = 0, Z = 0, and O = 0
6.
```
ADD    AX,BX
ADD    AX,CX
ADD    AX,DX
ADD    AX,SP
MOV    DI,AX
```
8. ADC DX,BX
10. The assembler cannot determine whether the memory location is a byte, word, or doubleword.
12. Difference = 81H, C = 0, A = 0, S = 1, Z = 0, and O = 0
14. DEC EBX
16. Both instructions are identical, except that the CMP instruction does not change the destination.
18. The product is found in DX:AX, where DX is the most significant part.
20.
```
MOV    DL,5
MOV    AL,DL
MUL    DL
MUL    DL
```

22. AX
24. If an overflow or divide by zero error occurs, the microprocessor executes a divide error interrupt.
26. AH
38. DAA (BCD addition) and DAS (BCD subtraction)
30. AAM converts AX to BCD by dividing it by a 10. The result (00–99) is found in AH and AL.

```
32. PUSH   AX
    MOV    AL,BL
    ADD    AL,DL
    DAA
    MOV    DL,AL
    MOV    AL,BH
    ADC    AL,DH
    DAA
    MOV    DH,AL
    POP    AX
    ADC    AL,CL
    DAA
    MOV    CL,AL
    MOV    AL,AH
    ADC    AL,CH
    DAA
    MOV    CH,AL
34. MOV    BH,DH
    AND    BH,1FH
36. MOV    SI,DI
    OR     SI,1FH
38. OR     AX,0FH
    AND    AX,1FFFH
    XOR    AX,0E0H
```

40. TEST CH, 4 or BT CH, 2
42. (a) SHR DI, 3 (b) SHL AL, 1 (c) ROL AL, 3 (d) SAR DH, 1
44. Extra
46. The REPE prefix continues to compare while an equal outcome from the comparison occurs or while CX is not equal to a zero.
48. The CMPSB instruction compares the byte contents of two memory locations.
50. The letter C is displayed on the video screen.

CHAPTER 6

2. A near JMP
4. A far JMP
6. (a) near (b) short (c) far
8. EIP/IP
10. The JMP AX instruction is a near jump to the offset address loaded in AX.
12. The JMP [DI] instruction is a near jump that obtains the jump address from the data segment memory location addressed by DI. The other JMP is a far jump.
14. JA is the jump above instruction that jumps if the destination is above the source.
16. JE, JNE, JG, JGE, JL, and JLE
18. JBE and JA

```
20.     MOV     DI,OFFSET DATA
    CLD
    MOV    CX,150H
      MOV    AL,0
    AGAIN:
      STOSB
      LOOP   AGAIN
```

24.
```
          CMP    AL,3
      JNE    ?0000
      ADD    AL,2
  ?0000:
```

26.
```
MOV     SI,OFFSET BLOCKA
MOV     DI,OFFSET BLOCKB
CLD
.REPEAT
LODSB
STOSB
.UNTIL AL==0
```

28
```
MOV     SI,OFFSET BLOCKA
MOV     DI,OFFSET BLOCKB
CLD
MOV     AL,0
.WHILE AL!=12H
LODSB
ADD    AL,ES:[DI]
STOSB
.ENDW
```

30. Both instructions store the return address on the stack, and then jump to the procedure. Note that the return address for a near CALL is EIP/IP, and the return address for the far CALL is EIP/IP and CS.
32. RET
34. By using NEAR or FAR to the right of the PROC directive.
36.
```
CUBE  PROC  NEAR  USES AX DX
   MOV   AX,CX
   MUL   CX
   MUL   CX
   RET
CUBE  ENDP
```
38. INT, INTO, and DIV (if an error occurs)
40. Interrupt type number 0 is used for a divide error.
42. The IRET pops the return address from the stack, as does a RET, but it also pops the flags, which RET does not.
44. INTO interrupts a program if the overflow flag is set.
46. The STI instruction enables the INTR pin and the CLI instruction disables INTR.
48. Interrupt vector number 9

CHAPTER 7

2. The result depends on the options set for the assembler, but the TEST.OBJ file is always generated while the TEST.LST and TEST.CRF files can also be generated.
4. PUBLIC is used to declare that a label or even an entire segment is public to other modules.
6. NEAR PTR, FAR PTR, BYTE, WORD, or DWORD
8. The MACRO directive starts a macro and the ENDM ends it.
10. Parameters are indicated next to the MACRO statement. Parameters are placed in a program next to the name of the macro, as operands.
12. The LOCAL directive must immediately follow the macro statement and must contain all labels used within the macro sequence.
14.
```
ADDM  MACRO  LIST,LENGTH
   MOV     CX,LENGTH
   MOV     SI,OFFSET LIST
   MOV     AX,0
   CLD
```

```
       .REPEAT
          ADD    AX,[SI]
          ADD    SI,2
       .UNTILCXZ
       ENDM
16. RANDOM MACRO
        LOCAL  R1
R1:
        INC    CL
        MOV    AH,6
        MOV    DL,-1
        INT    21H
        JZ     R1
        ENDM
18. DISP  MACRO  PARA
       IFB    <PARA>
          MOV     AH,6
          MOV     DL,13
          INT     21H
          MOV     DL,10
       ENDIF
       IFNB   <PARA>
          MOV     DL,PARA
       ENDIF
       MOV    AH,6
       INT    21H
       ENDM
20. DIPS  PROC   NEAR
       MOV    AH,6
       MOV    DL,-1
       INT    21H
       JZ     DISP
       .IF AL==0
           PUSH   AX
           SHR    AL,4
           ADD    AL,30H
           .IF AL>'9'
               ADD    AL,7
           .ENDIF
           MOV    AH,6
           MOV    DL,AL
           INT    21H
           POP    AX
           AND    AL,0FH
           ADD    AL,30H
           .IF AL>'9'
               ADD    AL,7
           .ENDIF
           MOV  AH,6
           MOV  DL,AL
               INT  21H
           .ENDIF
           RET
    DISP       ENDP
```

22. A large number is converted by repeated divisions by 10.

24. 30H

26. A 30H is first subtracted from each digit, and then the most significant digit is multiplied by 100 and the middle digit is multiplied by 10. The three are then summed to generate a binary value.

```
28. HEXAS  PROC   NEAR       ;AL is converted
        AND    AL,0FH
        MOV    BX,OFFSET LOOK
        XLAT   CS:LOOK
        RET
    HEXAS  ENDP
    LOOK   DB      30H,31H,32H,33H
        DB     34H,35H,36H,37H
        DB     38H,39H,41H,42H
```

```
       DB      43H,44H,45H,46H
30. XLAT SS:LOOK
32. .MODEL TINY
    .CODE
    DISP   MACRO  PARA
       MOV    DL,PARA
       MOV    AH,6
       INT    21H
       ENDP
    .STARTUP
       MOV    CX,8
       .REPEAT
           DISP   13
           DISP   10
           DISP   ' '
           MOV    AL,8
           SUB    AL,CL
           ADD    AL,30H
           DISP   AL
           DISP   13
           DISP   10
           DISP   '2'
           DISP   ' '
           DISP   '='
           MOV    AX,100H
           SHR    AX,CL
           .IF    AL>99
              DISP   '1'
              SUB    AL,100
              AAM
              ADD    AX,3030H
              DISP   AH
              DISP   AL
          .ELSE
              AAM
              .IF    AH!=0
                 ADD    AH,30H
                 DISP   AH
              .ENDIF
              ADD  AL,30H
              DISP AL
          .ENDIF


    UNTILCXZ
    EXIT
    END
```

CHAPTER 8

2. As long as you don't exceed a logic zero current of 2.0 mA, they are TTL-compatible.
4. Address bits A7–A0.
6. A logic 0 on RD indicates that the microprocessor is either reading data from memory or I/O.
8. The CLK input must be a TTL-compatible square-wave with a 33% duty cycle.
10. The WR signal indicates that the microprocessor has placed data on its data bus to be written to the memory or an I/O device.
12. If DT/R is a logic 1, it indicates that the microprocessor's data bus is transmitting data to the memory or I/O.
14. S2–S0
16. The queue tracking status bits indicate the condition of the queue within the microprocessor for the arithmetic coprocessor.

18. Three
20. 2.33 MHz
22. AD19–AD0
24. An eight-bit transparent latch (74LS373).
26. Buffers are required because of the drive current (2.0 mA) available at the output pins of the microprocessor.
28. Four
30. A read or a write
32. (a) State T1 is used by the microprocessor to provide the memory or I/O with the address. (b) State T2 provides access time to the memory and also is where the READY input is sampled. (c) State T3 is where the data are sampled or sent to the memory or I/O. (d) State T4 is used to deactivate the control signals.
34. 400 ns (2 clocks)
36. This input is used to request wait states.
38. Minimum mode is used unless the system must contain the arithmetic coprocessor; then we use the maximum mode.
40. The microprocessor is free to obtain and execute normal microprocessor instructions while the coprocessor executes a coprocessor instruction.
42. The FSTSW AX copies the status register into the AX register.
44. After executing the FCOMP ST(2) and FSTSW AX instructions, the SAHF instruction copies the AH register into the flag register (F7–F0). This allows the condition jump instruction JE to be used to test for equality.
46. FSTSW AX
48. Data are stored in eight 80-bit wide registers that are formed into a stack.
50. ST(0)
52. Affine allows positive or negative infinity, while projective does not.

CHAPTER 9

2. (a) 256 (b) 2048 (c) 4096 (d) 8192
4. The CS or CE pin on a memory device is used to select or enable the device so it can perform a read or a write operation.

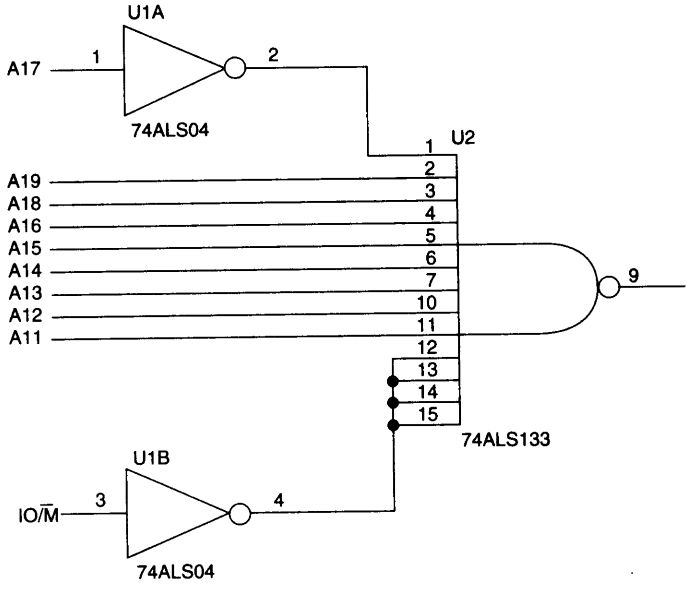


FIGURE D–1

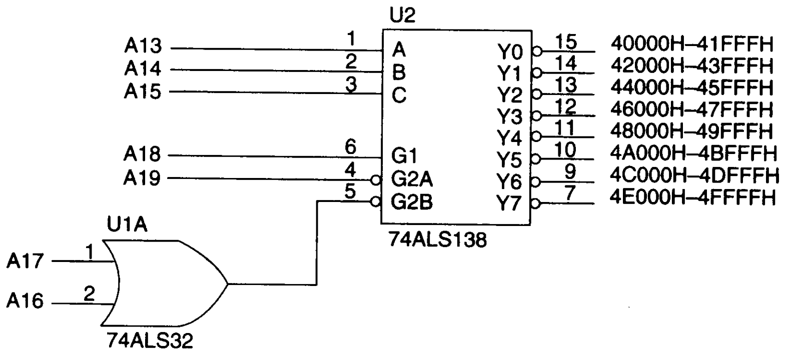


FIGURE D–2

6. The WE pin causes the memory to perform a write operation, provided that the CS or CE pin is also active.
8. The 5 MHz version of the 8088 allows 460 ns of time for the memory to access data. A 450 ns memory device will only function if the amount time required for the address decoder and buffers in a system is less than 10 ns, which is unlikely.
10. The SRAM (static RAM) is a device that retains data for as long as power is applied to the memory device. The SRAM can be read or written.
12. See Figure D-1
14. One of the eight outputs becomes a logic 0, as dictated by the A, B, and C address inputs.
16. See Figure D-2
18. The PROM address decoder is more suited to memory address decoding than the 74LS138 in many cases.
20. See Figure D-3

OE	A8	A7	A6	A5	A4	A3	A2	A1	A0	O0	O1	O2	O3	O4	O5	O6	O7
0	0	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1	1	1
0	0	0	0	0	1	1	0	1	0	1	1	0	1	1	1	1	1
0	0	0	0	0	1	1	0	1	1	1	1	1	0	1	1	1	1
0	0	0	0	0	1	1	1	0	0	1	1	1	1	0	1	1	1
0	0	0	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1
0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	1	0	1
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0

22. EQUATIONS

```
/O1 = A19 * /A18 * A17 * A16 * /A15 * /A14 * /A13
/O2 = A19 * /A18 * A17 * A16 * /A15 * /A14 * A13
/O3 = A19 * /A18 * A17 * A16 * /A15 * A14 * /A13
/O4 = A19 * /A18 * A17 * A16 * /A15 * A14 * A13
/O5 = A19 * /A18 * A17 * A16 * A15 * /A14 * /A13
/O6 = A19 * /A18 * A17 * A16 * A15 * /A14 * A13
/O7 = A19 * /A18 * A17 * A16 * A15 * A14 * /A13
/O8 = A19 * /A18 * A17 * A16 * A15 * A14 * A13
```

24. See Figure D-3.
26. The BHE selects the upper memory bank and the A0 (BLE) signal selects the lower memory bank.
28. Either separate decoders or separate write control signals
30. Low

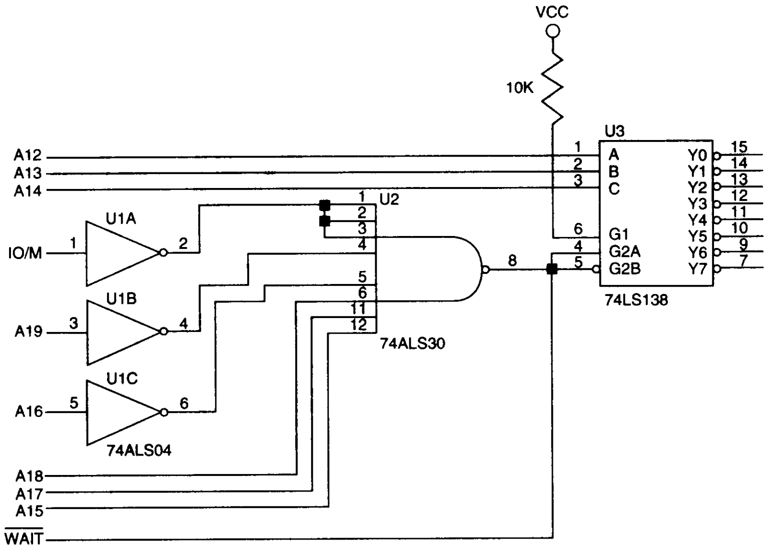


FIGURE D–3

CHAPTER 10

2. The fixed I/O port is stored in the memory immediately following the opcode.
4. Register AL
6. The OUTSB instruction copies the contents of the data segment memory location addressed by SI to the data bus, where it is written to the I/O device addressed by DX. After the transfer, the contents of SI are incremented or decremented, as dictated by the direction flag.
8. The difference between the memory-mapped I/O and the isolated I/O is that with isolated I/O, all memory locations are available to the system.
10. The basic output interface is a latch that holds data output from the microprocessor.
12. Low bank
14. The contact bounce eliminator removes mechanical bounces from a switch by using a latch or flip-flop.
16. See Figure D-4
18. See Figure D-5

```
;Equations for U1

EQUATIONS

/O1 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * /A3 * /A2 * /A1 * /MIO
/O2 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * /A3 * /A2 * A1 * /MIO
/O3 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * /A3 * A2 * /A1 * MIO
/O4 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * /A3 * A2 * A1 * /MIO
/O5 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * A3 * /A2 * /A1 * /MIO
/O6 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * A3 * /A2 * A1 * /MIO
/O7 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * A3 * A2 * /A1 * /MIO
/O8 = /U2 * /A4 * /A5 * /A6 * /A7 * A8 * A3 * A2 * A1 * /MIO
;Equation for U2
```

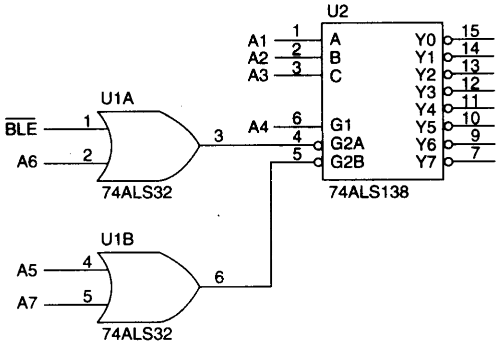


FIGURE D–4

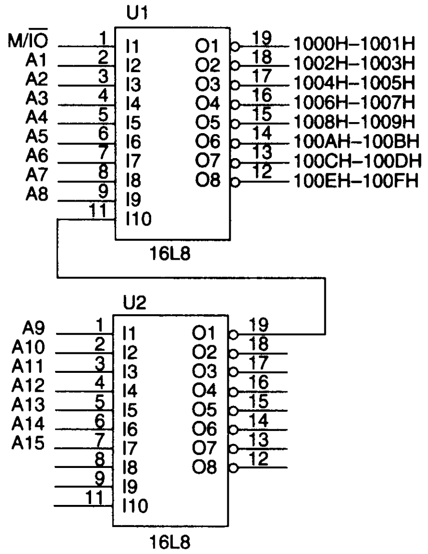


FIGURE D–5

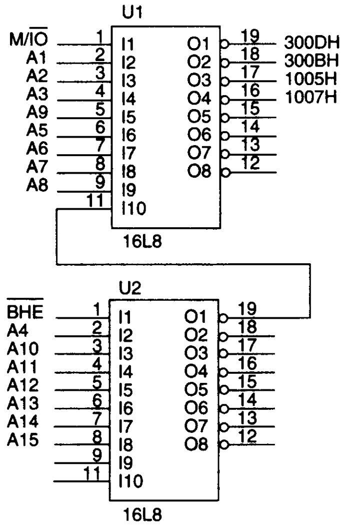


FIGURE D–6

```
EQUATIONS

/U2 = /A9 * /A10 * /A11 * /A12 * /A13 * /A14 * /A15
```

20. See Figure D-6

```
;Equations for U1

EQUATIONS

/O1 = /U2 * /A9 * /A5 * /A6 * /A7 * A8 * A3 * A2 * /A1 * /MIO
/O2 = /U2 * /A9 * /A5 * /A6 * /A7 * A8 * A3 * /A2 * A1 * /MIO
/O3 = /U2 * A9 * /A5 * /A6 * /A7 * A8 * /A3 * A2 * /A1 * MIO
/O4 = /U2 * A9 * /A5 * /A6 * /A7 * A8 * /A3 * A2 * A1 * /MIO
;Equation for U2

EQUATIONS

/U2 = /BHE * /A4 * /A10 * /A11 * /A12 * /A13 * /A14 * /A15
```

22. D7–D0
24. 24
26. A1 and A0
28. See Figure D-7

```
EQUATION

/CS = /A15*/A14*/A13*/A12*/A11*/A10*A9*A8*A7*/A6*/A5*/A4*/A3*/MIO
```

30. Latched I/O (mode 0), strobed I/O (mode 1), and bi-directional I/O (mode 2)
32. Whenever a coil is energized, the current causes the permanent magnet armature to step (move) to the next position.
34.

```
MOV  AL,0FH
OUT  COMMAND,AL
```

36. The ACK signal is an output that signals that the data have been removed from the port.
38.

```
IN    AL,PORTC
TEST  AL,16
JNZ   FOR_A_ONE
```

40. PC0–PC2

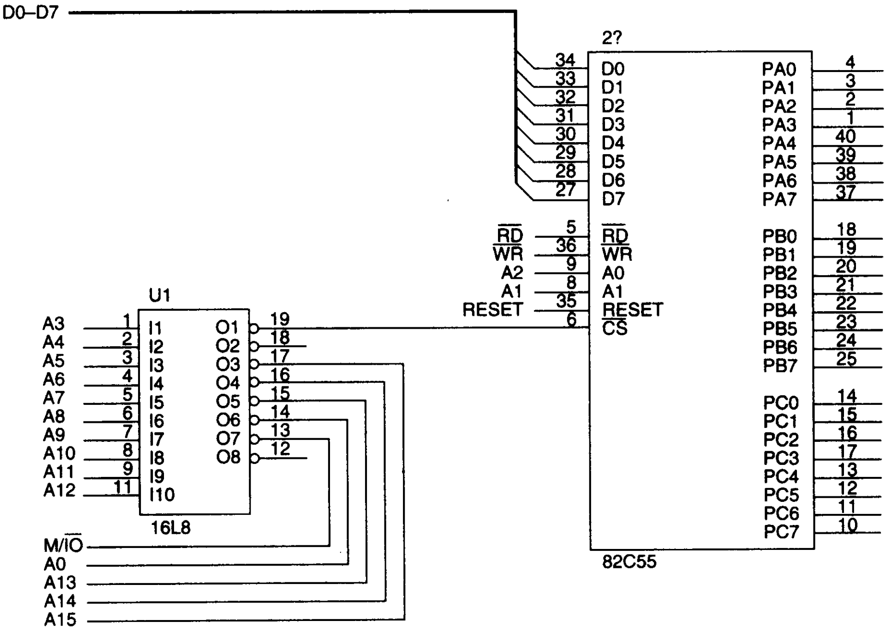


FIGURE D–7

42. A display position is selected by changing the display position address found in the set address command.
44. The busy line is tested by executing the busy flag command and then testing the most-significant bit.
46. 10 ms to 20 ms
48. Two wait states.
50. An overrun error occurs if the internal FIFO fills before the data are input to the microprocessor.
52. See Figure D-8
54. 10 MHz
56. See Figure D-9

```
MOV     AL,0B6H
OUT     16H,AL
MOV     AL,64H
OUT     14H,AL
MOV     AL,00H
OUT     14H,AL
```

58. Least-significant
60.

```
;using a 1 MHz clock
;
MOV     AL,74H
OUT     CONTROL,AL
MOV     AL,65H          ;count of 101
OUT     TIMER1,AL
MOV     AL,0
OUT     TIMER1,AL
```

62. Asynchronous serial data are data sent without a clock pulse.
64.

```
LINE    EQU     023H
LSB     EQU     020H
MSB     EQU     021H
FIFO    EQU     022H

        MOV     AL,10001010B     ;enable Baud divisor
        OUT     LINE,AL
```

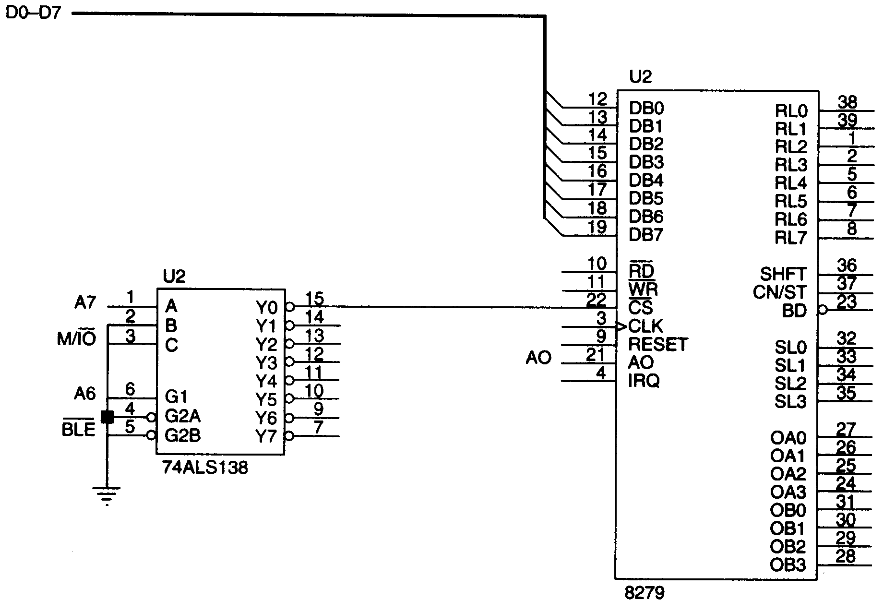


FIGURE D–8

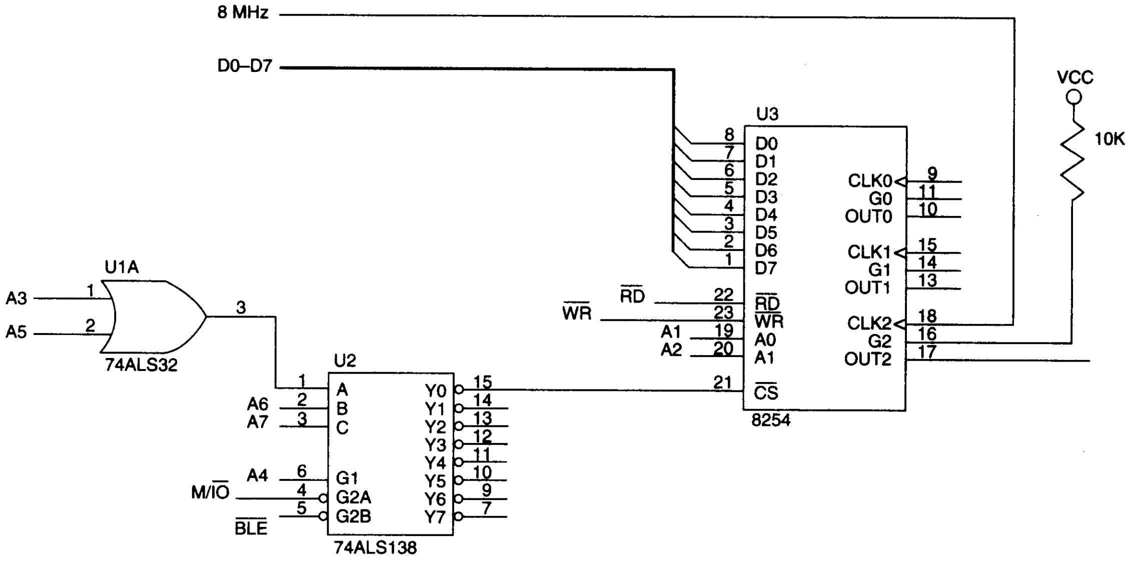


FIGURE D–9

```
        MOV    AL,60           ;program Baud rate
        OUT    LSB,AL
        MOV    AL,0
        OUT    MSB,AL

        MOV    AL,00011001B    ;program 7-data, odd
        OUT    LINE,AL         ;parity, one stop

        MOV    AL,00000111B    ;enable transmitter and
        OUT    FIFO,AL         ;and receiver
```

66. Simplex = sending or receiving, but not both. Half Duplex = sending and receiving, but only one direction at a time. Full Duplex = sending and receiving, both simultaneously.

68.

```
SENDS  PROC NEAR

MOV    CX,16
    .REPEAT
       .REPEAT
           IN     AL,LSTAT       ;get line status register
           TEST   AL,20H         ;test TH bit
       .UNTIL !ZERO?
       LODSB                     ;get data
       OUT  DATA,AL              ;transmit data
    .UNTILCXZ

    RET

SENDS  ENDP
```

70. 0.01 V

72.

```
.MODEL TINY
.CODE
.STARTUP
   MOV     DX,400H
   .WHILE 1
   MOV     CX,256
   MOV     AL,0
   .REPEAT
       OUT     DX,AL
       INC     AL
       CALL    DELAY
   .UNTILCXZ
   MOV  CX,256
   .REPEAT
       OUT     DX,AL
       DEC     AL
       CALL    DELAY
   .UNTILCXZ
   .ENDW
DELAY  PROC    NEAR

;  39 microsecond time delay

DELAY  ENDP
       END
```

74. The INTR pin indicates that the converter has completed a conversion.

76. See Figure D-10

```
;equations for U1
;
EQUATIONS

/O1 = /A15*/A14*/A13*/A12*/A11*/A10*A9*/A8*/A7*A6*A5*/A4*/A3*/A2*/A1*/BLE
/O8 = /A15*/A14*/A13*/A12*/A11*/A10*A9*/A8*/A7*A6*A5*A4*/A3*/A2*/A1*/BLE
```

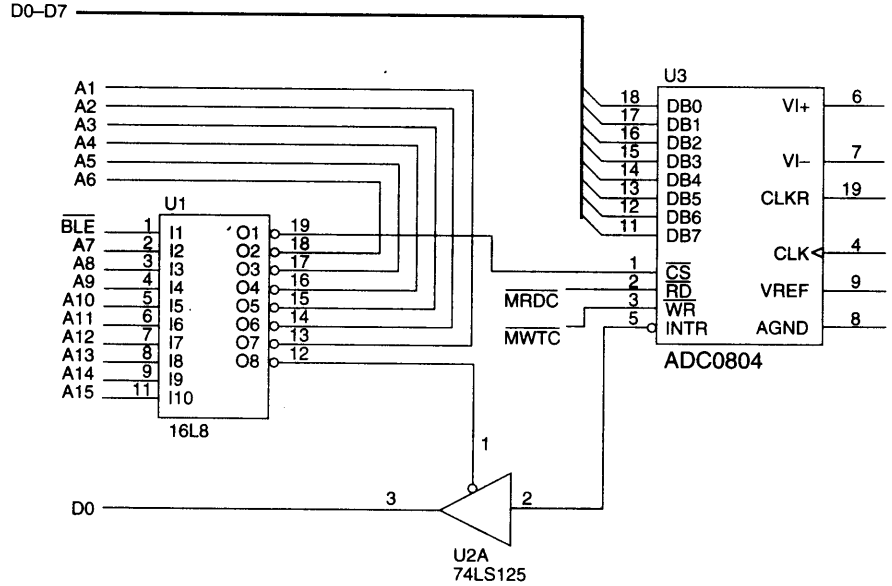


FIGURE D–10

CHAPTER 11

2. An interrupt is a hardware-initiated subroutine call.
4. Interrupts free processing time because the only time the processor is used is when the interrupt is active. This means that no time is wasted polling an I/O device.
6. INT, INTO, IRET/IRETD, CLI, STI
8. In the first 1K-byte at location 00000000H–000003FFH
10. Vectors 00H–1FH are reserved, even though some are used in the personal computer for other purposes.
12. The interrupt descriptor table is located anywhere in the memory system, as addressed by the interrupt descriptor-table register.
14. The main difference is the location of the interrupt vector. The real mode interrupt uses a vector from a table in the bottom 1K byte of memory, while the protected mode interrupt uses a descriptor from any location in the memory system. The other difference is that the protected mode interrupt service procedure can be placed anywhere in the memory system.
16. The INTO instruction interrupts a program only if the overflow flag bit is set.
18. The IRET instruction functions as a far RET, except that before the return occurs, data from the stack are popped into the flag register.
20. The flags are pushed onto the stack, the I and T flag bits are cleared to zero, and the interrupt service procedure is called using a vector from the interrupt vector table.
22. The trace or trap flag is set to enable tracing. Tracing is an interrupt that occurs after each instruction is executed to allow software to trace through a program.
24. The only way to clear or set the trace flag is to obtain an image of the flag register, and then change the trace bit to a zero or a one before returning the value back to the flag register. There is no instruction to set or clear the trace flag.
26. The INTA signal is only active in response to the INTR input.
28. The NMI input is both level and edge-sensitive.
30. A FIFO is a first-in, first-out memory system that stores data in the FIFO order. We sometimes also call a FIFO a *queue*.

32. See Figure D-11
34. A daisy chain is a method of connecting interrupts so that any active interrupt causes a logic 1 to be placed on the INTR input to the microprocessor. The daisy chain interrupt requires software to determine which interrupt is active.
36. The 8259A is a programmable interrupt controller that adds eight interrupt inputs to the microprocessor.
38. The IR inputs are the interrupt request input to the 8259A.
40. The slave INT pin connects to any IR pin on the master 8259A.
42. The OCW is an operational command word used to control the 8259A once it has been initialized by the ICW.
44. ICW2
46. ICW1, among other things, selects the level or edge triggering for the 8259A.
48. The priority rotation algorithm places the most recently serviced interrupt at the lowest priority level.

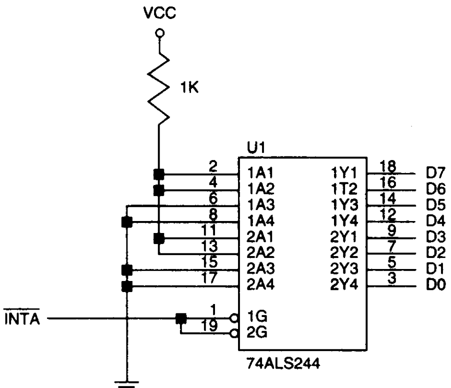


FIGURE D–11

CHAPTER 12

2. Whenever HOLD is placed at a logic 0 level, the microprocessor (a) stops executing a program within a few clocks, (b) places its address, data, and control buses at their high-impedance state, and (c) places a logic 1 on the HLDA to signal that the HOLD is in effect.
4. A DMA write transfers data from I/O to the memory.
6. DACK
8. If both HOLD and HLDA are a logic 1 level, the microprocessor is in its hold state.
10. 4
12. The command register
14.

```
LATCHB  EQU   10H       ;latch B
CLEAR_F EQU   7CH       ;F/L flip flop
CH0_A   EQU   70H       ;channel 0 address
CH1_A   EQU   72H       ;channel 1 address
CH1_C   EQU   73H       ;channel 1 count
MODE    EQU   7BH       ;mode
CMMD    EQU   78H       ;command
MASKS   EQU   7FH       ;masks
REQ     EQU   79H       ;request register
STATUS  EQU   78H       ;status register

TRANS   PROC  FAR USES AX

    MOV   AL,20H
    SHR   AL,4
    OUT   LATCHB,AL

    OUT   CLEAR_F,AL ;clear F/L flip-flop

    MOV   AX,1000H   ;program source address
    OUT   CH0_A,AL
    MOV   AL,AH
    OUT   CH0_A,AL

    MOV   AX,0000H   ;program destination address
```

```
        OUT     CH1_A,AL
        MOV     AL,AH
        OUT     CH1_A,AL

        MOV     AX,00FFH    ;program count
        OUT     CH1_C,AL
        MOV     AL,AH
        OUT     CH1_C,AL

        MOV     AL,88H      ;program mode
        OUT     MODE,AL
        MOV     AL,85H
        OUT     MODE,AL

        MOV     AL,1        ;enable block transfer
        OUT     CMMD,AL

        MOV     AL,0EH      ;unmask channel 0
        OUT     MASKS,AL

        MOV     AL,4        ;start DMA transfer
        OUT     REQ,AL

        .REPEAT             ;wait until DMA complete
           IN      AL,STATUS
        .UNTIL AL &1

    TRANS   ENDP
```

CHAPTER 13

2. The early ISA bus supports only 8-bit transfers, while the newest supports either 8- or 16-bit transfers.
4. 12M bits per second and 1.5M bits per second
6. 5 Meters
8. 127
10. A stuffed bit is a logic 0 placed in a data stream after a fixed number of ones are transmitted.
12. Up to 1024 bytes

CHAPTER 14

2. The 80186/80188 contain an internal clock generator, chip-selection logic, timers, programmable interrupt controller, DMA controller, power-down mode, serial interfaces, and parallel interfaces. Note that not all versions contain all features.
4. 10 MHz
6. 3.0 mA of current for a fan-out 7 TTLS parts or 10 CMOS components
8. The ALE signal appears ½ clock earlier in the 80186/80188.
10. 417 ns
12.
```
MOV   AX,1100H
MOV   DX,0FFFEH
OUT   DX,AL
```
14. Ten on most versions of the 80186/80188, including all the internal interrupts.
16. The interrupt-control registers each control a single interrupt input to the 80186/80188 interrupt controller.
18. The difference is that reading the interrupt poll register acknowledges the interrupt, while reading the interrupt poll-status register does not.
20. 3
22. Timer 2 always connects, and Timers 0 and 1 can be connected to the system clock.
24. The INH bit must be set to allow the EN bit to change.
26. Alternate operation using the two compare or maximum count registers.

28.

```
MOV    AX,    123
MOV    DX,    0FF5AH
    OUT    DX,AL
    MOV    AX,23
    ADD    DX,2
    OUT    DX,AL
    MOV    AX,8007H
    MOV    DX,0FF58H
    OUT    DX,AL
```

30. 2
32. The channel is started by software control (control register) or by hardware control (timer 2 or the DRQ input).
34. 7
36. Base
38. 0 and 15
40. It selects the operation of the PCS5/A0 and PCS6/A1 pins.
42.

```
MOV    DX,0FF90H
MOV    AX,1001H
OUT    DX,AL
MOV    DX,0FF92H
MOV    AX,1008H
OUT    DX,AL
```

44. It verifies that a protected mode segment can be read.

CHAPTER 15

2. 64T
4. See Figure D–12
6. The memory map for the 80386 contains 4G bytes of memory that is physically organized into a 32-bit wide memory system. Each of the four eight-bit wide memory banks is selected by using a bank enable signal labeled BE0–BE3.
8. The pipeline allows the microprocessor to send out an address while the data from a prior operation is being fetched. This allows the memory additional time for accessing the data.
10. 0000H–FFFFH

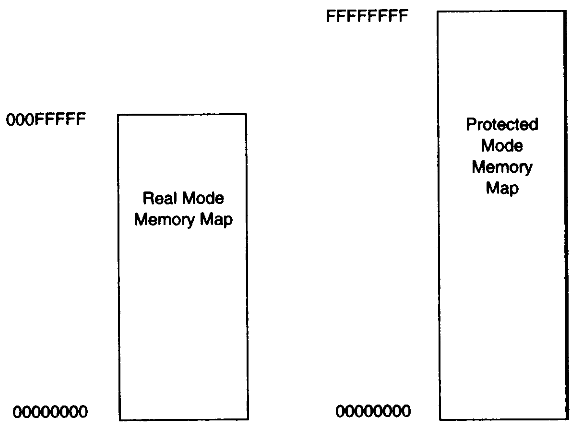


FIGURE D–12

12. The only differences are a wider data bus (32 bits) and a wider address bus (32 bits).
14. The BS16 pin configures the microprocessor to operate with a 16-bit data bus.
16. EAX, EBX, ECX, EDX, WSP, EBP, ESI, EDI, EIP, and EFLAGS.
18. CR0 = selects paging, and enters or leaves the protected mode, CR1 = reserved for the future, CR2 = holds the linear address of any fault, and CR3 = holds the base address of the page directory.
20. Interrupt type 1
22. The BSR instruction scans a number from the right toward the left. If a 1 is found, the zero flag is set and the bit position of the logic one is placed into the destination register.
24. MOV FS:[DI],EAX
26. Yes
28. Interrupt type number 7 is used to emulate the arithmetic coprocessor.
30. A double fault interrupt occurs whenever more than one interrupt occurs within the microprocessor.
32. A descriptor is a sequence of eight bytes that describe the location, length, and attributes of a protected mode memory segment.
34. If the table indicator is a logic 1, the local descriptor table is chosen by the segment register.
36. 8192
38. The segment descriptor describes a data, code, or stack segment, while the system descriptor describes a CALL or interrupt gate or a task.
40. The TSS is addressed by a special descriptor that is accessed by the task register.
42. The 803786 is switched between real and protected mode by setting or clearing the rightmost bit of CR0.
44. CR3 holds the base address of the paging directory.
46. Linear address D0000000H addresses a physical page by accessing page directory entry 1101000000. In this entry, the address of the page table that describes 4M of memory is located. Page table entry 0000000000 holds memory address C0000000H to translate linear address D0000000H to C0000000H.
48. The FLUSH input erases the internal 80486 cache.
50. None, except for the 80486SX, which contains an alignment check flag used by the arithmetic coprocessor (80487).
52. Even parity
54. 16
56. A cache write-through occurs when the microprocessor writes data to the cache and to the memory system.
58. If paging is in effect, caching can be turned on and off for different page translations.
60. This instruction does nothing if AL = CL; if AL ●CL, then CL is copied into AL.
62. If PCD = 0, the cache is enabled for the current memory page.

CHAPTER 16

2. Up to 64G bytes.
4. These pins both generate parity for the ninth bit per byte and also check parity.
6. This pin signals the microprocessor that the bus is ready and is used to insert wait states into the timing.
8. 18.2 ns
10. T2
12. Two 8K-byte caches, one for data and the other for instructions.
14. Yes, as long as they are not dependent on each other.
16. The memory management mode is a special mode accessed through the memory management interrupt input to the Pentium and Pentium Pro.
18. 38000H
20. This instruction compares eight bytes of data stored in memory with EDX:EAX.
22. ID, VIF, and VIP

24. The Pentium and Pentium Pro access 4M-byte pages by using the page directory to store the base page address of a 4M-byte memory page instead of the address of a page table.
26. The Pentium and the Pentium Pro differ in address bus size (32 bits versus 36 bits on the Pentium Pro), the FCMOV and CMOV instructions are added to the Pentium Pro, and the Pentium Pro contains the level 2 cache with a size of either 256K or 512K bytes.
28. A35–A3
30. The access times are essentially the same on both microprocessors if operated with the same frequency bus clock.
32. 72-bit wide SDRAM called ECC SDRAM

CHAPTER 17

2. 512K, 1M, and 2M
4. The Pentium Pro level 2 cache is built into the integrated circuit, while the Pentium II level 2 cache is a separate integrated circuit mounted onto a printed circuit board called the Pentium II cartridge.
6. 64G bytes
8. 242
10. The read and write pins are replaced by request input used by the bus controller to request a read or a write operation.
12. 8 ns
14. SYSENTER_CS, SYSENTER_SS, and SYSENTER_ESP
16. The ECX register passes the address or register number to the RDMSR instruction. After executing the RDMSR instruction, EDX:EAX contains the contents of the register.
18.

```
TESTS  PROC NEAR
    CPUID
    TEST EDX,800H       ;test bit position 11
    .IF ZERO?
       CLC
    .ELSE
       STC
    .ENDIF
    RET
TESTS  ENDP
```

20. The return address is retrieved, placed in EDX, and then stored in EIP by the SYSEXIT instruction.
22. Level or ring 3

APPENDIX D

2. 16-bit (±32K), 32-bit (±2 G), and 64-bit (±9 ∞ 10^{18})
4. Short (32-bits), long (64-bits), and extended (64-bits)
6. (a) –7.5 (b) +0.5625 (c) +320 (d) +2.0 (e) +10 (f) +0.0
8. The FST DATA instruction copies (does not pop) data from ST(0) into memory location data as a 64-bit floating-point number.
10. FADD ST,ST(3)
12. FSUB ST(2),ST
16. The FSAVE instruction saves all of the register in the coprocessor to memory.
18.

```
REAC    PROC    NEAR

    FLDPI
    FADD    ST,ST(0)
    FMUL    F
```

```
        FMUL    C1
        FLD1
        FDIVR
        FSTP    XC
        RET

REAC    ENDP
22.POW    PROC    NEAR

        MOV     TEMP,EBX
        FLD     TEMP
        F2XM1
        FLD1
        FADD
        MOV     TEMP,EAX
        FLD     TEMP
        FYL2X
        FSTP    TEMP
        MOV     ECX,TEMP
        RET

POW     ENDP
```

APPENDIX E

2. A study of the most often used instructions revealed that most of the instructions are not used. This led to the Reduced Instruction Set.
4. The Data Manipulation Type.
6. They become smaller as the complex instructions are removed. Complex instructions need more control signals/machine cycles.
8. Control Area is the area on the chip used by Control Unit. Regularization Factor is the ratio of the modules drawn from a library to the total modules on the chip.
10. For a 6 machine cycle the depth of pipeline is 6.
12. After 6 clocks the 1st instruction would have been completed and after every clock one instruction would be completed. The number of instructions to complete is 5, after the first one came out; so 6+5 clocks would be needed to complete the 6 instruction execution.
14. ALPHA 21064
16. SPARC