CHAPTER 10
INSTRUCTION SETS: CHARACTERISTICS AND FUNCTIONS

ANSWERS TO QUESTIONS

10.1 The essential elements of a computer instruction are the opcode, which specifies the operation to be performed, the source and destination operand references, which specify the input and output locations for the operation, and a next instruction reference, which is usually implicit.

10.2 Registers and memory.

10.3 Two operands, one result, and the address of the next instruction.

10.4 **Operation repertoire:** How many and which operations to provide, and how complex operations should be. **Data types:** The various types of data upon which operations are performed. **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on. **Registers:** Number of CPU registers that can be referenced by instructions, and their use. **Addressing:** The mode or modes by which the address of an operand is specified.

10.5 Addresses, numbers, characters, logical data.

10.6 For the IRA bit pattern 011XXXX, the digits 0 through 9 are represented by their binary equivalents, 0000 through 1001, in the right-most 4 bits. This is the same code as packed decimal.

10.7 With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out is lost. On the other end, a 0 is shifted in. The **arithmetic shift** operation treats the data as a signed integer and does not shift the sign bit. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained.

10.8 **1.** In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data. **2.** Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds. **3.** To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

10.9 First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction.

10.10 The term refers to the occurrence of a procedure call inside a procedure.

10.11 Register, start of procedure, top of stack.

10.12 A reentrant procedure is one in which it is possible to have several calls open to it at the same time.

10.13 An assembly language uses symbolic names for addresses that are not fixed to specific physical addresses; this is not the case with machine language.

10.14 In this notation, the operator follows its two operands.

10.15 A multibyte numerical value stored with the most significant byte in the lowest numerical address is stored in **big-endian** fashion. A multibyte numerical value stored with the most significant byte in the highest numerical address is stored in **little-endian** fashion.

Answers to Problems

10.1 **a.** 23
b. 32 33

10.2 **a.** 7309
b. 582
c. 1010 is not a valid packed decimal number, so there is an error

10.3 **a.** 0; 255
b. –127; 127
c. –127; 127
d. –128; 127
e. 0; 99
f. –9; +9

10.4 Perform the addition four bits at a time. If the 4-bit digit of the result of binary addition is greater then 9 (binary 1001), then add 6 to get the correct result.

```
  1698          0001        0110        1001        1000
+ 1798          0001        0111        1000        0110
                0010        1100      1 0001        1110
                   1           1           1        0110
                0011        1110        0110      1 0100
                            0110        1000
                          1 0100

  3484          0011        0100        1000        0100
```

10.5 The tens complement of a number is formed by subtracting each digit from 9, and adding 1 to the result, in a manner similar to twos complement. To subtract, simply take the tens complement and add:

$$\begin{array}{rl} 0736 & \\ \underline{9674} & \\ 1 & \quad 0410 \end{array}$$

10.6

PUSH A	LOAD E	MOV R0, E	MUL R0, E, F
PUSH B	MUL F	MUL RO, F	SUB R0, D, R0
PUSH C	STORE T	MOV R1, D	MUL R1, B, C
MUL	LOAD D	SUB R1, R0	ADD R1, A, R1
ADD	SUB T	MOV R0, B	DIV X, R0, R1
PUSH D	STORE T	MOV R0, C	
PUSH E	LOAD B	ADD R0, A	
PUSH F	MUL C	DIV R0, R1	
MUL	ADD A	MOV X, R0	
SUB	DIV T		
DIV	STO X		
POP X			

Source: [TANE90]

10.7 **a.** A memory location whose initial contents are zero is needed for both X → AC and AC → X. The program for X → AC, and its effects are shown below. Assume AC initially contains the value a.

Instruction	AC	Effect on M(0)	M(X)
SUBS 0	a	a	x
SUBS 0	0	0	x
SUBS X	–x	0	–x
SUBS 0	–x	–x	–x
SUBS 0	0	0	–x
SUBS X	x	0	x

b. For addition, we again need a location, M(0), whose initial value is 0. We also need destination location, M(1). Assume the initial value in M(1) is y.

Instruction	AC	M(0)	M(1)	M(X)
SUBS 0	a	a	y	x
SUBS 1	a – y	a	a – y	x
SUBS 1	0	a	0	x
SUBS X	–x	a	0	–x
SUBS 0	–x – a	–x – a	0	–x
SUBS 1	–x – a	–x – a	–x – a	–x
SUBS 0	0	0	–x – a	–x
SUBS X	x	0	–x – a	x
SUBS 0	x	x	–x – a	x
SUBS 0	0	0	–x – a	x
SUBS 1	a + x	0	a + x	x

10.8 **1.** A NOOP can be useful for debugging. When it is desired to interrupt a program at a particular point, the NOOP is replaced with a jump to a debug routine. When temporarily patching or altering a program, instructions may be replaced with NOOPs. **2.** A NOOP introduces known delay into a program, equal to the instruction cycle time for the NOOP. This can be used for measuring time or introducing time delays. **3.** NOOPs can be used to pad out portions of a program to align instructions on word boundaries or subroutines on page boundaries. **4.** NOOPs are useful in RISC pipelining, examined in Chapter 13.

10.9

Bit pattern	Value	Arithmetic left shift	Value	Logical left shift	Value
00000	0	00000	0	00000	0
00001	1	00010	2	00010	2
00010	2	00100	4	00100	4
00011	3	00110	6	00110	6
00100	4	01000	8	01000	8
00101	5	01010	10	01010	10
00110	6	01100	12	01100	12
00111	7	01110	14	01110	14
01000	8	00000	overflow	10000	overflow
01001	9	00010	overflow	10010	overflow
01010	10	00100	overflow	10100	overflow
01011	11	00110	overflow	10110	overflow
01100	12	01000	overflow	11000	overflow
01101	13	01010	overflow	11010	overflow
01110	14	01100	overflow	11100	overflow
01111	15	01110	overflow	11110	overflow
10000	–16	10000	overflow	00000	overflow
10001	–15	00010	overflow	00010	overflow
10010	–14	10100	overflow	00100	overflow
10011	–13	10110	overflow	00110	overflow
10100	–12	11000	overflow	01000	overflow
10101	–11	11010	overflow	01010	overflow
10110	–10	11100	overflow	01100	overflow
10111	–9	11110	overflow	01110	overflow
11000	–8	10000	–16	10000	–16
11001	–7	10010	–14	10010	–14
11010	–6	10100	–12	10100	–12
11011	–5	10110	–10	10110	–10
11100	–4	11000	–8	11000	–8
11101	–3	11010	–6	11010	–6
11110	–2	11100	–4	11100	–4
11111	–1	11110	–2	11110	–2

10.10 Round toward –∞.

10.11 Yes, if the stack is only used to hold the return address. If the stack is also used to pass parameters, then the scheme will work only if it is the control unit that removes parameters, rather than machine instructions. In the latter case, the CPU would need both a parameter and the PC on top of the stack at the same time.

10.12 The DAA instruction can be used following an ADD instruction to enable using the add instruction on two 8-bit words that hold packed decimal digits. If there is a decimal carry (i.e., result greater than 9) in the rightmost digit, then it shows up either as the result digit being greater than 9, or by setting AF. If there is such a carry, then adding 6 corrects the result. For example:

$$\begin{array}{r} 2\,7 \\ \underline{+\ 4\,6} \\ 6\,D \\ \underline{+\ 0\,6} \\ 7\,3 \end{array}$$

The second test similarly corrects a carry from the left digit of an 8-bit byte. A multiple-digit packed decimal addition can thus be programmed using the normal add-with-carry (ADC) instruction in a loop, with the insertion of a single DAA instruction after each addition.

10.13 **a.**

CMP result	Z	C
destination < source	0	1
destination > source	0	0
destination = source	1	0

b.

CMP result	Flags
destination < source	S ≠ O
destination > source	S = O
destination = source	ZF = 1

c.
- **Equal:** The two operands are equal, so subtraction produces a zero result (Z = 1).
- **Greater than:** If A is greater than B, and A and B are both positive or both negative, then the twos complement operation (A – B) will produce a positive result (S = 0) with no overflow (O = 0). If A is greater than B with A positive and B negative, then the result is either positive with no overflow or negative (S = 1) with overflow (O = 1). In all these cases, the result is nonzero (Z = 0)
- **Greater than or equal:** The same reasoning as for "Greater than" applies, except that the result may be zero or nonzero.
- **Less than:** This condition is the opposite of "Greater than or equal" and so the opposite set of conditions apply.
- **Less than or equal:** This condition is the opposite of "Greater than" and so the opposite set of conditions apply.
- **Not equal:** The two operands are unequal, so subtraction produces a nonzero result (Z = 0).

10.14 **a.** sign bit in the most significant position, then exponent, then significand
b. sign, exponent, and significand are all zero; that is, all 32 bits are zero
c. biased representation of the exponent
d. yes. However, note that the IEEE has a representation for minus zero, which would yield results indicating that –0 < 0.

10.15 **a.** It might be convenient to have a word-length result for passing as a parameter via a stack, to make it consistent with typical parameter passing. This is an advantage of Scond. There doesn't seem to be any particular advantage to the result value for true being integer one versus all binary ones.
b. The case for setting the flags: In general, instructions that operate on data values will, as a side effect, set the condition codes according to the result of the operation. Thus, the condition code should reflect the state of the machine

after the execution of each instruction that has altered a data value in some way. These instructions violate this principle and are therefore inconsistent with the remainder of the architecture.

The case against: These instructions are similar to branch on condition instructions in that they operate on the result of another operation, which is reflected in the condition codes. Because a branch on condition code instruction does not itself set the condition codes, the fact that these other instructions do not is not inconsistent.

For a further discussion, see "Should Scc Set Condition Codes?" by F. Williams, *Computer Architecture News*, September 1988.

c.

```
        SUB     CX, CX  ;set register CX to 0
        MOV     AX, B   ;move contents of location B to register AX
        CMP     AX, A   ;compare contents of register AX and location A
        SETGT   CX      ;CX = (a GT b)
TEST    JCXZ    OUT     ;jump if contents of CX equal 0
        THEN

        OUT
```

d.

```
        MOV     EAX, B  ; move from location B to register EAX
        CMP     EAX, C
        SETG    BL      ; BL = 0/1 depending on result
        MOV     EAX, D
        CMP     EAX, F
        MOV     BH, 0
        SETE    BH
        OR      BL, BH
```

10.16 **a.** Add one byte at a time:

```
 AB 08 90 C2
+45 98 EE 50
 F0 A0 7E 12
```

b. Add 16 bits at a time:

```
 AB08 90C2
+4598 EE50
 F0A0 7F12
```

10.17 If the processor makes use of a stack for subroutine handling, it only uses the stack while executing CALL and RETURN instructions. No explicit stack-oriented instructions are needed.

10.18 **a.** (A + B + C) * D
b. (A/B) + (C/D)
c. A/(B * C * (D + E))
d. A + (B * ((C + (D * E)/F) – G)/H)

10.19 **a.** AB + C + D + E +
b. AB + CD + * E +
c. AB * CD * + E +
d. AB - CDE * – F/G/ * H *

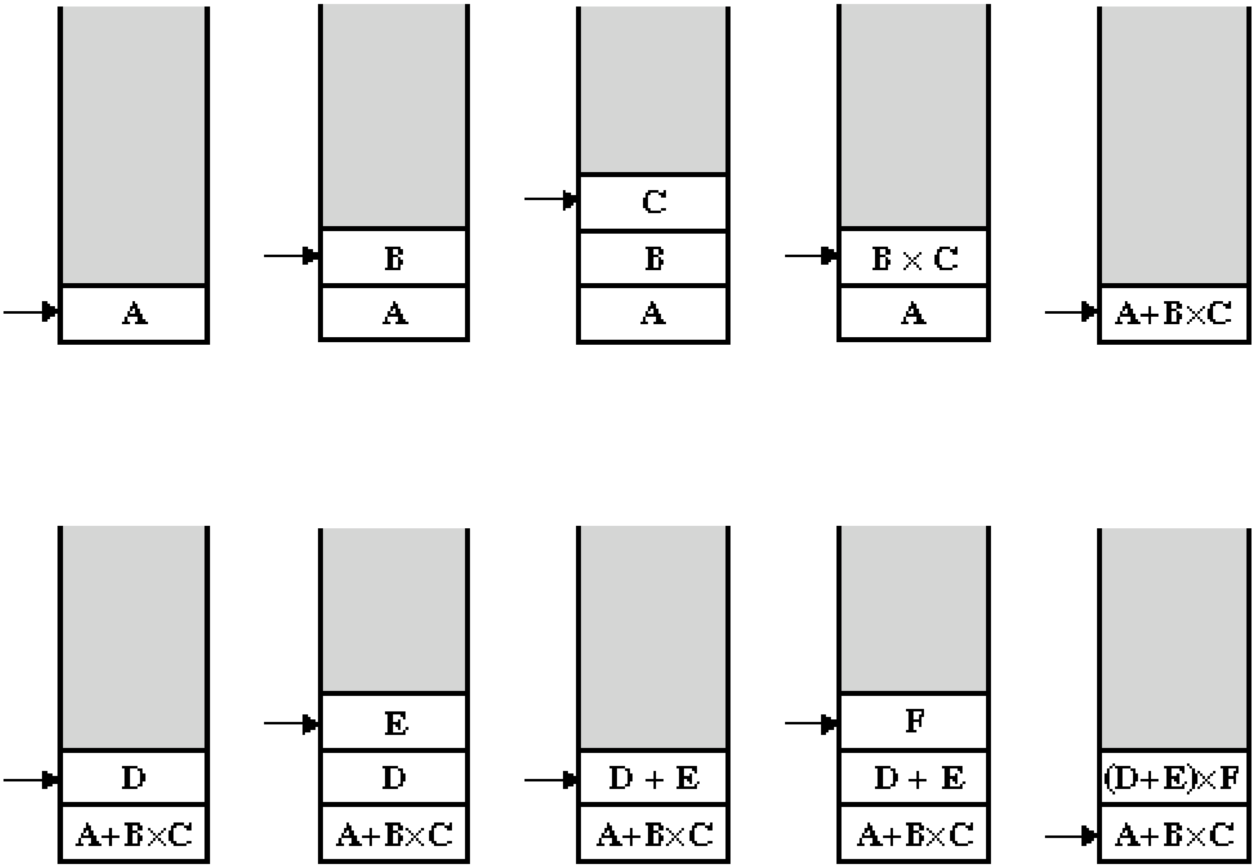
10.20 Postfix Notation: AB + C –
Equivalent to (A + B) – C

It matters because of rounding and truncation effects.

10.21

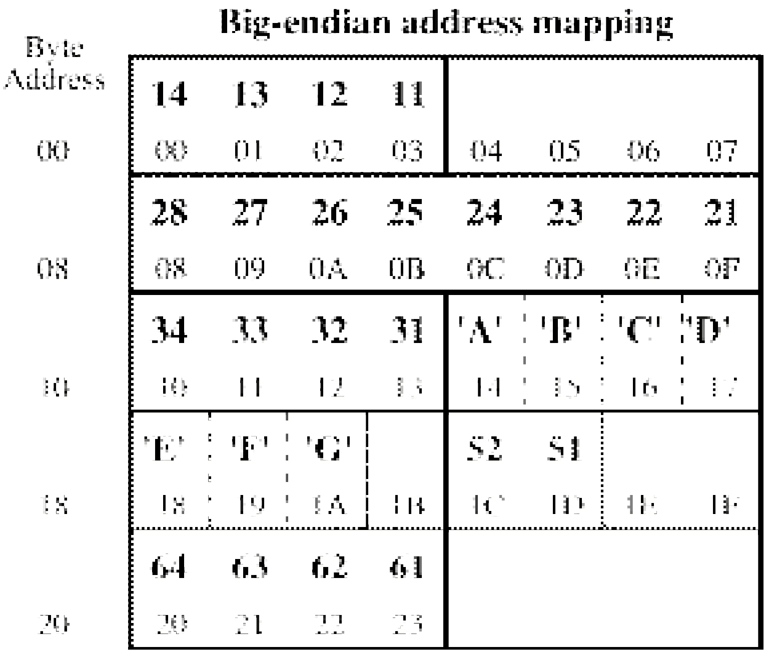
Input	Output	Stack (top on right)
(A – B) / (C + D × E)	empty	empty
A – B) / (C + D × E)	empty	(
– B) / (C + D × E)	A	(
B) / (C + D × E)	A	(–
) / (C + D × E)	A B	(–
/ (C + D × E)	A B –	empty
(C + D × E)	A B –	/
C + D × E)	A B –	/ (
+ D × E)	A B – C	/ (
D × E)	A B – C	/ (+
× E)	A B – C D	/ (+
E)	A B – C D	/ (+ ×
)	A B – C D E	/ (+ ×
empty	A B – C D E × +	/
empty	A B – C D E × + /	empty

10.22



The final step combines the top two stack elements using the + operator.

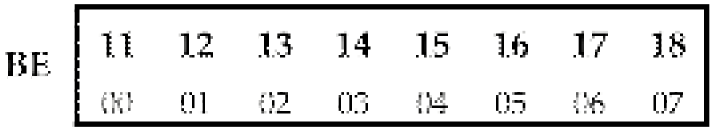
10.23



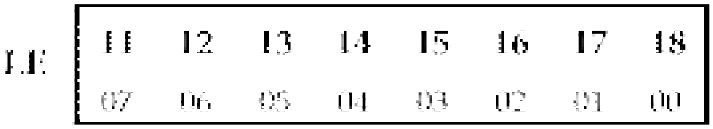
Big-endian address mapping

Byte Address								
	14	13	12	11				
00	00	01	02	03	04	05	06	07
	28	27	26	25	24	23	22	21
08	08	09	0A	0B	0C	0D	0E	0F
	34	33	32	31	'A'	'B'	'C'	'D'
10	10	11	12	13	14	15	16	17
	'E'	'F'	'G'		S2	S1		
18	18	19	1A	1B	1C	1D	1E	1F
	64	63	62	61				
20	20	21	22	23				

10.24 a.



BE	11	12	13	14	15	16	17	18
	00	01	02	03	04	05	06	07



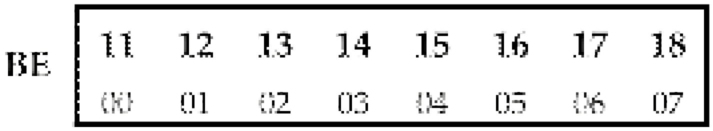
LE	11	12	13	14	15	16	17	18
	07	06	05	04	03	02	01	00

b.

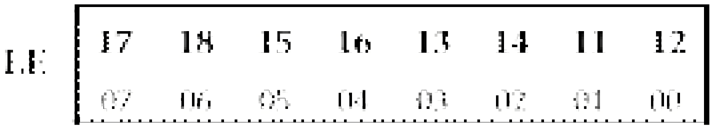
BE	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
	00	01	02	03	04	05	06	07

LE	**15**	**16**	**17**	**18**	**11**	**12**	**13**	**14**
	07	06	05	04	03	02	01	00

c.



BE	11	12	13	14	15	16	17	18
	00	01	02	03	04	05	06	07



LE	17	18	15	16	13	14	11	12
	07	06	05	04	03	02	01	00

The purpose of this question is to compare halfword, word, and doubleword integers as members of a data structure in Big- and Little-Endian form.

10.25 Figure 10.12 is not a "true" Little-Endian organization as usually defined. Rather, it is designed to minimize the data manipulation required to convert from one Endian to another. Note that 64-byte scalars are stored the same in both formats on the PowerPC. To accommodate smaller scalars, a technique known as address munging is used.

When the PowerPC is in Little-Endian mode, it transforms the three low-order bits of an effective address for a memory access. These three bits are XORed with a value that depends on the transfer size: 100b for 4-byte transfers; 110 for 2-byte transfers; and 111 for 1-byte transfers. The following are the possible combinations:

4-Byte Transfers (XOR with 100)		**2-Byte Transfers (XOR with 110)**		**1-Byte Transfers (XOR with 111)**	
Original Address	**Munged Address**	**Original Address**	**Munged Address**	**Original Address**	**Munged Address**
000	100	000	110	000	111
001	101	001	111	001	110
010	110	010	100	010	101
011	111	011	101	011	100
100	000	100	010	100	011
101	001	101	011	101	010
110	010	110	000	110	001
111	011	111	001	111	000

For example, the two-byte value 5152h is stored at location 1C in Big-Endian mode. In Little-Endian mode, it is viewed by the processor as still being stored in location 1C but in Little-Endian mode. In fact, the value is still stored in Big-Endian mode, but at location 1A. When a transfer occurs, the system must do an address unmunging and a byte transfer to convert data to the form expected by the processor. The processor generates effective addresses of 1C and 1D for the two bytes. These addresses are munged (XOR with 110) to 1A and 1B. The data bytes are retrieved, swapped, and presented as if found in the unmunged addresses 1D and 1C.

10.26 There are a number of ways to do this. Here is one way that will work:

```
#include <stdio.h>
main()
{
  int integer;
  char *p;

  integer = 0x30313233;  /* ASCII for chars '0', '1', '2', '3' */
  p = (char *)&integer

  if (*p=='0' && *(p+1)=='1' && *(p+2)=='2' && *(p+3)=='3')
     printf("This is a big endian machine.\n");
  else if (*p=='3' && *(p+1)=='2' && *(p+2)=='1' && *(p+3)=='0')
     printf("This is a little endian machine.\n");
  else
     printf("Error in logic to determine machine endian-ness.\n");
}
```

10.27 BigEndian

10.28 The documentation uses little-endian bit ordering, stating that the most significant bit of a byte (leftmost bit) is bit 7. However, the instructions that operate on bit fields operate in a big-endian manner. Thus, the leftmost bit of a byte is bit 7 but has a bit offset of 0, and the rightmost bit of a byte is bit 0 but has a bit offset of 7.

Chapter 11
Instruction Sets: Addressing Modes and Formats

Answers to Questions

11.1 Immediate addressing: The value of the operand is in the instruction.

11.2 Direct addressing: The address field contents the effective address of the operand.

11.3 Indirect addressing: The address field refers to the address of a word in memory, which in turn contains the effective address of the operand.

11.4 Register addressing: The address field refers to a register that contains the operand.

11.5 Register indirect addressing: The address field refers to a register, which in turn contains the effective address of the operand.

11.6 Displacement addressing: The instruction has two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field refers to a register whose contents are added to A to produce the effective address.

11.7 Relative addressing: The implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the EA.

11.8 It is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some systems will automatically do this as part of the same instruction cycle, using autoindexing.

11.9 These are two forms of addressing, both of which involve indirect addressing and indexing. With **preindexing**, the indexing is performed before the indirection. With **postindexing**, the indexing is performed after the indirection.

11.10 **Number of addressing modes:** Sometimes an addressing mode can be indicated implicitly. In other cases, the addressing modes must be explicit, and one or more mode bits will be needed. **Number of operands:** Typical instructions on today's machines provide for two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address fields. **Register versus memory:** The more that registers can be used for operand references, the fewer bits are needed. **Number of register sets:** One advantage of using multiple register sets is that, for a fixed number of registers, a functional split requires fewer bits to be used in the instruction. **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used. With displacement addressing, the range is opened up to the length of the address register. **Address granularity:** In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed-size memory, more address bits.

11.11 **Advantages:** It easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes. **Disadvantages:** an increase in the complexity of the CPU.

Answers to Problems

11.1 **a.** 20 **b.** 40 **c,** 60 **d.** 30 **e.** 50 **f.**70

11.2 **a.** X3 = X2
b. X3 = (X2)
c. X3 = X1 + X2 + 1
d. X3 = X2 + X4

11.3 **a.** the address field
b. memory location 14
c. the memory location whose address is in memory location 14
d. register 14
e. the memory location whose address is in register 14

11.4

	EA	Operand
a.	500	1100
b.	201	500
c.	1100	1700
d.	702	1302

	EA	Operand
e.	600	1200
f.	R1	400
g.	400	1000
h.	400	1000

The autoindexing with increment is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.

11.5 Recall that relative addressing uses the contents of the program counter, which points to the next instruction after the current instruction. In this case, the current instruction is at decimal address 256028 and is 3 bytes long, so the PC contains 256031. With the displacement of –31, the effective address is 256000.

11.6 (PC + 1) + Relative Address = Effective Address
Relative Address = –621 + 530 = –91
Converting to twos-complement representation, we have: 1110100101.

11.7 **a.** 3 times: fetch instruction; fetch operand reference; fetch operand.
b. 2 times: fetch instruction; fetch operand reference and load into PC.

11.8 Load the address into a register. Then use displacement addressing with a displacement of 0.

11.9 The PC-relative mode is attractive because it allows for the use of a relatively small address field in the instruction format. For most instruction references, and many data references, the desired address will be within a reasonably short distance from the current PC address.

11.10 This is an example) of a special-purpose CISC instruction, designed to simplify the compiler. Consider the case of indexing an array, where the elements of the array are 32 bytes long. The following instruction is just what is needed:

```
IMUL     EBX, I, 32
```

EBX is a 32-bit register that now contains the byte offset into the array whose subscript is 1.

11.11 The three values are added together: 1970 + 48022 + 8 = 50000.

11.12 **a.** No, because the source operand is the contents of X, rather than the top of the stack, which is in the location pointed to by X.
b. No, because address of the top of the stack is not changed until after the fetching of the destination operand.
c. Yes. The stack grows away from memory location 0.
d. No, because the second element of the stack is fetched twice.
e. No, because the second element of the stack is fetched twice.
f. No, because the stack pointer is incremented twice, so that the result is thrown away.
g. Yes. The stack grows toward memory location 0.

11.13

Instruction	**Stack (top on left)**
PUSH 4	4
PUSH 7	7, 4
PUSH 8	8, 7, 4
ADD	15, 4
PUSH 10	10, 15, 4
SUB	5, 4
MUL	20

11.14 The 32-bit instruction length yields incremental improvements. The 16-bit length can already include the most useful operations and addressing modes. Thus, relatively speaking, we don't have twice as much "utility".

11.15 With a different word length, programs written for older IBM models would not execute on the newer models. Thus the huge investment in existing software was lost by converting to the newer model. Bad for existing IBM customers, and therefore bad for IBM.

11.16 Let X be the number of one-address instructions. The feasibility of having K two-address, X one-address, and L zero-address instructions, all in a 16-bit instruction word, requires that:

$$(K \times 2^6 \times 2^6) + (X \times 2^6) + L = 2^{16}$$

Solving for X:

$$X = (2^{16} - (K \times 2^6 \times 2^6) - L)/2^6$$

To verify this result, consider the case of no zero-address and no two-address instructions; that is, L = K = 0. In this case, we have

$$X = 2^{16}/2^6 = 2^{10}$$

This is what it should be when 10 bits are used for opcodes and 6 bits for addresses.

11.17 The scheme is similar to that for problem 11.16. Divide the 36-bit instruction into 4 fields: A, B, C, D. Field A is the first 3 bits; field B is the next 15 bits; field C is the next 15 bits, and field D is the last 3 bits. The 7 instructions with three operands use B, C, and D for operands and A for opcode. Let 000 through 110 be opcodes and 111 be a code indicating that there are less than three operands. The 500 instructions with two operands are specified with 111 in field A and an opcode in field B, with operands in D and C. The opcodes for the 50 instructions with no operands can also be accommodated in B.
Source: [TANE90]

11.18 **a.** The zero-address instruction format consists of an 8-bit opcode and an optional 16-bit address. The program has 12 instructions, 7 of which have an address. Thus:

$$N_0 = 12 \times 8 + 7 \times 16 = 208 \text{ bits}$$

b. The one-address instruction format consists of an 8-bit opcode and a 16-bit address. The program has 11 instructions.

$$N_1 = 24 \times 11 = 264 \text{ bits}$$

c. For two-address instructions, there is an 8-bit opcode and two operands, each of which is 4 bits (register) or 16 bits (memory).

$$N_2 = 9 \times 8 + 7 \times 16 + 11 \times 4 = 228 \text{ bits}$$

d. For three-address instructions

$$N_3 = 5 \times 8 + 7 \times 16 + 8 \times 4 = 184 \text{ bits}$$

11.19 No. If the two opcodes conflict, the instruction is meaningless. If one opcode modifies the other or adds additional information, this can be viewed as a single opcode with a bit length equal to that of the two opcode fields. However, instruction bundles, such as seen on the IA-64 Itanium architecture, have multiple opcodes.

11.20 **a.** The opcode field can take on one of $2^5 = 32$ different values. Each value can be interpreted to ways, depending on whether the Operand 2 field is all zeros, for a total of 64 different opcodes.

b. We could gain an additional 32 opcodes by assigning another Operand 2 pattern to that purpose. For example, the pattern 0001 could be used to specify more opcodes. The tradeoff is to limit programming flexibility, because now Operand 2 cannot specify register R1. Source: [PROT88].

CHAPTER 12
PROCESSOR STRUCTURE AND FUNCTION

ANSWERS TO QUESTIONS

12.1 **User-visible registers:** These enable the machine- or assembly language programmer to minimize main-memory references by optimizing use of registers. **Control and status registers:** These are used by the control unit to control the operation of the CPU and by privileged, operating system programs to control the execution of programs.

12.2 General purpose; Data; Address; Condition codes

12.3 Condition codes are bits set by the CPU hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

12.4 All CPU designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information. The PSW typically contains condition codes plus other status information.

12.5 (**1)**The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer. (**2)** A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

12.6 **Multiple streams:** A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. **Prefetch branch target:** When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched. **Loop buffer:** A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the *n* most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. **Branch prediction:** A prediction is made whether a conditional branch will be taken when executed, and subsequent instructions are fetched accordingly. **Delayed branch:** It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

12.7 One or more bits that reflect the recent history of the instruction can be associated with each conditional branch instruction. These bits are referred to as a taken/not taken switch that directs the processor to make a particular decision the next time the instruction is encountered.

Answers to Problems

12.1 **a.**

```
 00000010
 00000011
 00000101
```

Carry = 0; **Zero** = 0; **Overflow** = 0; **Sign** = 0; **Even parity** = 1; **Half-carry**= 0.

Even parity indicates that there is an even number of 1s in the result. The Half-Carry flag is used in the addition of packed decimal numbers. When a carry takes place out of the lower-order digit (lower-order 4 bits), this flag is set. See problem 10.1.

b.

```
  11111111
  00000001
 100000000
```

Carry = 1; **Zero** = 1; **Overflow** = 1; **Sign** = 0; **Even Parity** = 1; **Half-Carry** = 1.

12.2 To perform A – B, the ALU takes the twos complement of B and adds it to A:

$$\begin{array}{rl} \text{A:} & 11110000 \\ \overline{\text{B}} + 1\text{:} & +\underline{11101100} \\ \text{A} - \text{B:} & 11011100 \end{array}$$

Carry = 1; **Zero** = 0; **Overflow** = 0; **Sign** = 1; **Even parity** = 0; **Half-carry**= 0.

12.3 **a.** 0.2 ns
b. 0.6 ns

12.4 **a.** The length of a clock cycle is 0.1 ns. The length of the instruction cycle for this case is [10 + (15 × 64)] × 0.1 = 960 ns.
b. The worst-case delay is when the interrupt occurs just after the start of the instruction, which is 960 ns.
c. In this case, the instruction can be interrupted after the instruction fetch, which takes 10 clock cycles, so the delay is 1 ns. The instruction can be interrupted between byte transfers, which results in a delay of no more than 15 clock cycles = 1.5 ns. Therefore, the worst-case delay is 1.5 ns.

12.5 **a.** A factor of 2.
b. A factor of 1.5. Source: [PROT88].

12.6 **a.** The occurrence of a program jump wastes up to 4 bus cycles (corresponding to the 4 bytes in the instruction queue when the jump is encountered). For 100 instructions, the number of nonwasted bus cycles is, on average, 90 × 2 = 180.

The number wasted is as high as 10 × 4 = 40. Therefore the fraction of wasted cycles is 40/(180 + 40) = 0.18.

b. If the capacity of the instruction queue is 8, then the fraction of wasted cycles is 80/(180 + 80) = 0.3. Source: [PROT88].

12.7

Time →

	1	2	3	4	5
Instruction 1	F	E			
Instruction 2		F	E		
Instruction 3			F	E	
Instruction 4				F	E

This diagram distorts the true picture. The execute stage will be much longer than the fetch stage.

12.8

	1	2	3	4	5	6	7	8	9	10
I1	FI	DA	FO	EX						
I2		FI	DA	FO	EX					
I3			FI	DA	FO	EX				
I4				FI	DA	FO				
I5					FI	DA				
I6						FI				
I15							FI	DA	FO	EX

12.9 **a.** We can ignore the initial filling up of the pipeline and the final emptying of the pipeline, because this involves only a few instructions out of 1.5 million instructions. Therefore the speedup is a factor of five.

b. One instruction is completed per clock cycle, for an throughput of 2500 MIPS.

12.10 **a.** Using Equation (12.2), we can calculate the speedup of the pipelined 2-GHz processor versus a comparable 2-GHz processor without pipelining:

$$S = (nk)/[k + (n - 1) = 500/104 = 4.8$$

However, the unpipelined 2-GHz processor will have a reduced speed of a factor of 0.8 compared to the 2.5-GHz processor. So the overall speedup is 4.8 × 0.8 = 3.8.

b. For the first processor, each instruction takes 4 clock cycle, so the MIPS rate is 2500 MHz/4 = 625 MIPS. For the second processor, instructions are completed at the rate of one per clock cycle, so that the MIPS rate is 2000 MIPs.

12.11 The number of instructions causing branches to take place is pqn, and the number that do not cause a branch is (1 – pq)n. As a good approximation, we can replace Equation (12.1) with:

$$T_k = pqnk\tau + (1 - pq)[k + (n - 1)]\tau$$

Equation (12.2) then becomes

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{(pq)nk\tau + (1 - pq)[k + (n - 1)]\tau} = \frac{nk}{(pq)nk + (1 - pq)[k + (n - 1)]}$$

12.12 **(1)** The branch target cannot be fetched until its address is determined, which may require an address computation, depending on the addressing mode. This causes a delay in loading one of the streams. The delay may be increased if a component of the address calculation is a value that is not yet available, such as a displacement value in a register that has not yet been stored in the register. Other delays relate to contention for the register file and main memory. **(2)** The cost of replicating significant parts of the pipeline is substantial, making this mechanism of questionable cost-effectiveness.

12.13 **a.** Call the first state diagram Strategy A. Strategy A corresponds to the following behavior. If both of the last two branches of the given instruction have not taken the branch, then predict that the branch will not be taken; otherwise, predict that the branch will be taken.

Call the second state diagram Strategy B. Strategy B corresponds to the following behavior. Two errors are required to change a prediction. That is, when the current prediction is Not Taken, and the last two branches were not taken, then two taken branches are required to change the prediction to Taken. Similarly, if the current prediction is Taken, and the last two branches were taken, then two not-taken branches are required to change the prediction to Not Taken. However, if there is a change in prediction followed by an error, the previous prediction is restored.

- **b.** Strategy A works best when it is usually the case that branches are taken. In both Figure 12.17 and Strategy B, two wrong guesses are required to change the prediction. Thus, for both a loop exit will not serve to change the prediction. When most branches are part of a loop, these two strategies are superior to Strategy A. The difference between Figure 12.17 and Strategy B is that in the case of Figure 12.17, two wrong are also required to return to the previous prediction, whereas in Strategy B, only one wrong guess is required to return to the previous prediction. It is unlikely that either strategy is superior to the other for most programs.

12.14 **a.** The comparison of memory addressed by A0 and A1 renders the BNE condition false, because the data strings are the same. The program loops between the first two lines until the contents of D1 are decremented below 0 (to -1). At that point, the DBNE loop is terminated. D1 is decremented from 255 ($FF) to -1; thus the loop runs a total of 256 times. Due to the longword access and the postincrement addressing, the A0 and A1 registers are incremented by 4 × $100 = $400, to $4400 and $5400, respectively.

b. The first comparison renders the BNE condition true, because the compared data patterns are different. Therefore the DBNE loop is terminated at the first comparison. However, the A0 and A1 registers are incremented to $4004 and $5004, respectively. D1 still contains $FF.

12.15

Fetch	**D1**	**D2**	**EX**	**WB**		**CMP Reg1, Imm**
	Fetch	**D1**	**D2**	**EX**		**Jcc Target**
		Fetch	**D1**	**D2**	**EX**	**Target**

12.16 We need to add the results for the three types of branches, weighted by the fraction of each type that go to the target. For the scientific environment, the result is:

$$[0.725 \times (0.2 + 0.432)] + [0.098 \times 0.91] + 0.177 = 0.724$$

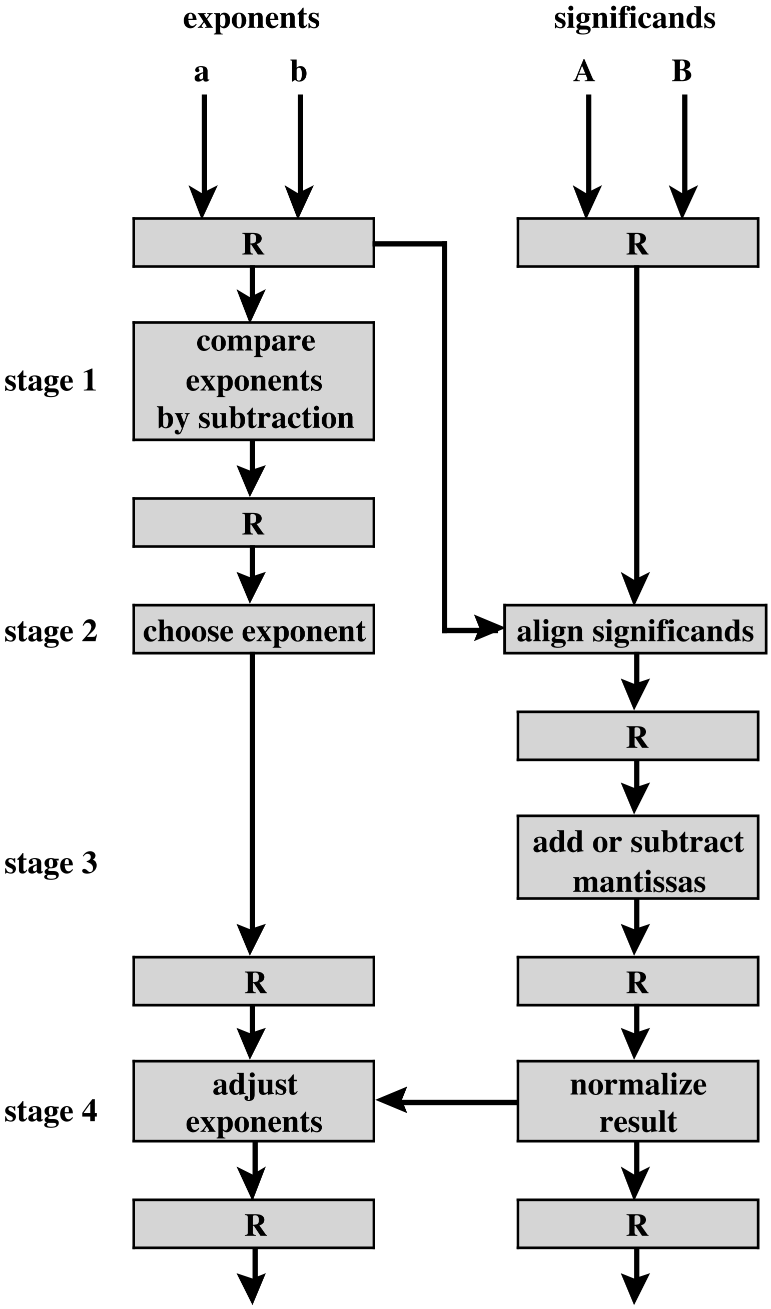
For the commercial environment, the result is:

$$[0.725 \times (0.4 + 0.243)] + [0.098 \times 0.91] + 0.177 = 0.732$$

For the systems environment, the result is:

$$[0.725 \times (0.35 + 0.325)] + [0.098 \times 0.91] + 0.177 = 0.756$$

12.17


exponents
significands
a
b
A
B
R
R
stage 1
compare exponents by subtraction
R
stage 2
choose exponent
align significands
R
stage 3
add or subtract mantissas
R
R
stage 4
adjust exponents
normalize result
R
R

CHAPTER 13
REDUCED INSTRUCTION SET COMPUTERS

ANSWERS TO QUESTIONS

13.1 **(**1) a limited instruction set with a fixed format, (2) a large number of registers or the use of a compiler that optimizes register usage, and (3) an emphasis on optimizing the instruction pipeline.

13.2 Two basic approaches are possible, one based on software and the other on hardware. The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms. The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time.

13.3 **(1)** Variables declared as global in an HLL can be assigned memory locations by the compiler, and all machine instructions that reference these variables will use memory-reference operands. **(2)** Incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures

13.4 One instruction per cycle. Register-to-register operations. Simple addressing modes. Simple instruction formats.

13.5 Delayed branch, a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after execution of the following instruction.

ANSWERS TO PROBLEMS

13.1 **a.** Figure 4.16 shows the movement of the window for a size of five. Each movement is an underflow or an overflow. Total = 18.

b. The results for W = 8 can easily be read from Figure 4.16. Each movement of a window in the figure is by an increment of 1. Initially, the window covers 1 through 5, then 2 through 6, and so on. Only when the window reaches 5 through 9 have we reached a point at which a window of size 8 would have to move. Total = 8.

c. The greatest call depth in the figure is 15, hence for W = 16, Total = 0.

13.2 The temporary registers of level J are the parameter registers of level J + 1. Hence, those registers are saved and restored as part of the window for J + 1.

13.3 **Two-way pipeline:** The I and E phases can overlap; thus we use N rather than 2N. Each D phase adds delay, so that term still must be included. Finally, each jump wastes the next instruction fetch opportunity. Hence

$$\text{2-Way:} \quad N + D + J$$

Three-way pipeline: Because the D phase can overlap with the subsequent E phase, it would appear that we can eliminate the D term. However, as can be seen in Figure 13.6, the data fetch is not completed prior to the execution of the following instruction. If this following instruction utilizes the fetched data as one of its operands, it must wait one phase. If this data dependency occurs a fraction α of the time, then:

$$\text{3-Way:} \quad N + \alpha D + J$$

Four-way pipeline: In this case, each jump causes a loss of two phases, and a data-dependent D causes a delay of one phase. However, the phases may be shorter.

$$\text{4-Way:} \quad N + \alpha D + 2J$$

13.4

Load	rA ← M	I	E_1	E_2	D							
Load	rB ← M		I	E_1	E_2	D						
NOOP				I	E_1	E_2						
Branch	X				I	E_1	E_2					
Add	rC ← rA + rB					I	E_1	E_2				
Store	M ← rC						I	E_1	E_2	D		

13.5 If we replace I by 32 × I, we can generate the following code:

```
      MOV    ECX, 32         ; use register ECX to hold 32 × I
LP:   MOV    EBX, Q[ECX]     ; load VAL field
      ADD    S, EBX          ; add to S
      ADD    ECX, 32         ; add 32 to 32 × I
      CMP    ECX, 3200       ; test against adjusted limit
      JNE    LP              ; loop until I × 32 = 100 × 32
```

13.6

```
      LD     R1, 0           ; keep value of S in R1
      LD     R2,1            ; keep value of K in R2
LP    SUB    R1, R1, R2      ; S := S – 1
LP1   BEQ    R2, 100, EXIT   ; done if K = 100
      NOP
      ADD    R2, R2, 1       ; else increment K
      JMP    LP1             ; back to start of loop
      SUB    R1, R1, R2      ; execute SUB in JMP delay slot
```

13.7 **a.**

```
LD     MR1, A             ;load A into machine register 1
LD     MR2, B             ;load B into machine register 2
ADD    MR1, MR1, MR2      ;add contents of MR1 and MR2 and store in MR3
LD     MR2, C
LD     MR3, D
ADD    MR2, MR2, MR3
```

A total of 3 machine registers are used, but now that the two additions use the same register, we no longer have the opportunity to interleave the calculations for scheduling purposes.

b. First we do instruction reordering from the original program:

```
LD      SR1, A
LD      SR2, B
LD      SR4, C
LD      SR5, D
ADD     SR3, SR1, SR2
ADD     SR6, SR4, SR5
```

This avoids the pipeline conflicts caused by immediately referencing loaded data. Now we do the register assignment:

```
LD      MR1, A
LD      MR2, B
LD      MR3, C
LD      MR4, D
ADD     MR5, MR1, MR2
ADD     MR1, MR3, MR4
```

Five machine registers are used instead of three, but the scheduling is improved.

13.8

	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic
Pentium II	12	12	15	no	yes
PowerPC	1	4	1	no	no

	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
Pentium II	2	yes	2	2	4
PowerPC	1	no	1	5	5

13.9

Register-to-Register Move	$R_d \leftarrow R_s + R_0$
Increment, Decrement	Use ADD with immediate constant of 1, –1
Complement	R_S XOR (–1)
Negate	$R_0 - R_s$
Clear	$R_d \leftarrow R_0 + R_0$

13.10 $N = 8 + (16 \times K)$

13.11 **a.** OR rsc with Go and store the result in dst

b. SUBCC src2 from src1 and store the result in G0

c. ORCC src1 with G0 and store the result in G0

d. XNOR dst with G0
e. SUB dst from G0 and store in dst
f. ADD 1 to dst (immediate operand)
g. SUB 1 from dst (immediate operand)
h. OR G0 with G0 and store in dst
i. SETHI G0 with 0
j. JMPL %I7+8m %G0
Source: [TANE99]

13.12 a.

```
      sethi  %hi(K), %r8              ;load high-order 22 bits of address of location
                                      ;K into register r8
      ld     [%r8 + %lo(K)], %r8      ;load contents of location K into r8
      cmp    %r8, 10                  ;compare contents of r8 with 10
      ble    L1                       ;branch if (r8) ≤ 10
      nop
      inc    %r8                      ;add 1 to (r8)
      b      L2
      nop
L1:   dec    %r8                      ;subtract 1 from (r8)
L2:   sethi  %hi(L), %r10
      st     %r8, [%r10 + %lo(L)]     ;store (r8) into location L
```

b.

```
      sethi  %hi(K), %r8              ;load high-order 22 bits of address of location
                                      ;K into register r8
      ld     [%r8 + %lo(K)], %r8      ;load contents of location K into r8
      cmp    %r8, 10                  ;compare contents of r8 with 10
      ble.a  L1                       ;branch if (r8) ≤ 10
      dec    %r8                      ;subtract 1 from (r8)
      inc    %r8                      ;add 1 to (r8)
      b      L2
      nop
L1:
L2:   sethi  %hi(L), %r10
      st     %r8, [%r10 + %lo(L)]     ;store (r8) into location L
```

c.

```
      sethi  %hi(K), %r8              ;load high-order 22 bits of address of location
                                      ;K into register r8
      ld     [%r8 + %lo(K)], %r8      ;load contents of location K into r8
      cmp    %r8, 10                  ;compare contents of r8 with 10
      ble.a  L1                       ;branch if (r8) ≤ 10
      dec    %r8                      ;subtract 1 from (r8)
      inc    %r8                      ;add 1 to (r8)
L2:   sethi  %hi(L), %r10
      st     %r8, [%r10 + %lo(L)]     ;store (r8) into location L
```

Chapter 14
Instruction-Level parallelism and Superscalar Processors

Answers to Questions

14.1 A superscalar processor is one in which multiple independent instruction pipelines are used. Each pipeline consists of multiple stages, so that each pipeline can handle multiple instructions at a time. Multiple pipelines introduce a new level of parallelism, enabling multiple streams of instructions to be processed at a time.

14.2 Superpipelining exploits the fact that many pipeline stages perform tasks that require less than half a clock cycle. Thus, a doubled internal clock speed allows the performance of two tasks in one external clock cycle

14.3 Instruction-level parallelism refers to the degree to which the instructions of a program can be executed in parallel.

14.4 **True data dependency:** A second instruction needs data produced by the first instruction. **Procedural dependency:** The instructions following a branch (taken or not taken) have a procedural dependency on the branch and cannot be executed until the branch is executed. **Resource conflicts:** A resource conflict is a competition of two or more instructions for the same resource at the same time. **Output dependency:** Two instructions update the same register, so the later instruction must update later. **Antidependency:** A second instruction destroys a value that the first instruction uses.

14.5 **Instruction-level parallelism** exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping. **Machine parallelism** is a measure of the ability of the processor to take advantage of instruction-level parallelism. Machine parallelism is determined by the number of instructions that can be fetched and executed at the same time (the number of parallel pipelines) and by the speed and sophistication of the mechanisms that the processor uses to find independent instructions.

14.6 **In-order issue with in-order completion:** Issue instructions in the exact order that would be achieved by sequential execution and to write results in that same order. **In-order issue with out-of-order completion:** Issue instructions in the exact order that would be achieved by sequential execution but allow instructions to run to completion out of order. **Out-of-order issue with out-of-order completion:** The processor has a lookahead capability, allowing it to identify independent instructions that can be brought into the execute stage. Instructions are issued with little regard for their original program order. Instructions may also run to completion out of order.

14.7 For an out-of-order issue policy, the instruction window is a buffer that holds decoded instructions. These may be issued from the instruction window in the most convenient order.

14.8 Registers are allocated dynamically by the processor hardware, and they are associated with the values needed by instructions at various points in time. When a new register value is created (i.e., when an instruction executes that has a register as a destination operand), a new register is allocated for that value.

14.9 **(1)** Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions. These functions require the use of multiple pipeline fetch and decode stages, and branch prediction logic. **(2)** Logic for determining true dependencies involving register values, and mechanisms for communicating these values to where they are needed during execution. **(3)** Mechanisms for initiating, or issuing, multiple instructions in parallel. **(4)** Resources for parallel execution of multiple instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references. **(5)** Mechanisms for committing the process state in correct order.

Answers to Problems

14.1 This problem is discussed in [JOHN91]. One approach to restarting after an interrupt relies on processor hardware to maintain a simple, well-defined restart state that is identical to the state of a processor having in-order completion. A processor providing this form of restart state is said to support *precise interrupts*. With precise interrupts, the interrupt return address indicates both the location of the instruction that caused the interrupt and the location where the program should be restarted. Without precise interrupts, the processor needs a mechanism to indicate the exceptional instruction and another to indicate where the program should be restarted. With out-of-order completion, providing precise interrupts is harder than not providing them, because of the hardware required to give the appearance of in-order completion.

14.2 **a.**

Instruction	Fetch	Decode	Execute	Writeback
0 ADD r3, r1, r2	0	1	2	3
1 LOAD r6, [r3]	1	2	4	9
2 AND r7, r5, 3	2	3	5	6
3 ADD r1, r6, r0	3	4	10	11
4 SRL r7, r0, 8	4	5	6	7
5 OR r2, r4, r7	5	6	8	10
6 SUB r5, r3, r4	6	7	9	12
7 ADD r0, r1, 10	7	8	12	13
8 LOAD r6, [r5]	8	9	13	18
9 SUB r2, r1, r6	9	10	19	20
10 AND r3, r7, 15	10	11	14	15

b.

Instruction	Fetch	Decode	Execute	Writeback
0 ADD r3, r1, r2	0	1	2	3
1 LOAD r6, [r3]	1	2	4	9
2 AND r7, r5, 3	2	3	5	10
3 ADD r1, r6, r0	3	4	11	12
4 SRL r7, r0, 8	4	5	12	13
5 OR r2, r4, r7	5	6	14	15
6 SUB r5, r3, r4	6	7	15	16
7 ADD r0, r1, 10	7	8	17	18
8 LOAD r6, [r5]	8	9	19	24
9 SUB r2, r1, r6	9	10	25	26
10 AND r3, r7, 15	10	11	26	27

c.

Instruction	Fetch	Decode	Execute	Writeback
0 ADD r3, r1, r2	0	1	2	3
1 LOAD r6, [r3]	0	1	4	9
2 AND r7, r5, 3	1	2	3	4
3 ADD r1, r6, r0	1	2	10	11
4 SRL r7, r0, 8	2	3	4	5
5 OR r2, r4, r7	2	3	6	7
6 SUB r5, r3, r4	3	4	5	6
7 ADD r0, r1, 10	3	4	12	13
8 LOAD r6, [r5]	4	5	11	16
9 SUB r2, r1, r6	4	5	17	18
10 AND r3, r7, 15	5	6	7	8

14.3 Because integer decoding is done in the same pipeline stage as dispatching. Forcing the integer instruction to dispatch from the bottom of the queue eliminates the need for buffer selection logic prior to the integer dispatch logic; in fact, the integer dispatch/decode logic can be merged with the logic forming the head of the instruction buffer. If the integer pipeline should be blocked, there is a decode buffer following the bottom of the queue that the instruction moves into, freeing up the queue slot for another instruction. The instruction in the decode buffer moves into the integer unit when the unit becomes free.

14.4 **a.**

```
                           1  2  3  4  5  6  7  8  9  1  1  1  1  1  1  1
                                                      0  1  2  3  4  5  6
     lwz   r8=a(r1)        F  D  E  C  W
     lwz   r12=b(r1,4)     F  ·  D  E  C  W
     lwz   r9=c(r1,8)      F  ·  ·  D  E  C  W
     lwz   r10=d(r1,12)    F  ·  ·  ·  D  E  C  W
     lwz   r11=e(r1,16)    F  ·  ·  ·  ·  D  E  C  W
     cmpi  cr0=r8,0        F  ·  ·  ·  ·  ·  D  E
     bc ELSE,cr0/gt=false  F  ·  ·  ·  S
IF:  add   r12=r8,r12      F  ·  ·  ·  ·  ·  ·  ·
     add   r12=r12,r9            F  ·  ·  ·  ·  ·
     add   r12=r12,r10           F  ·  ·  ·  ·  ·
     add   r4=r12,r11
     stw   a(r1)=r4
     b     OUT
ELSE: subf r12=r8,r12                                  F  D  E  W
     subf  r12=r12,r9                                  F  ·  D  E  W
     subf  r12=r12,r10                                 F  ·  ·  D  E  W
     subf  r4=r12,r11                                  F  ·  ·  ·  D  E  W
     stw   a(r1)=r4                                    F  ·  ·  ·  ·  D  E  C
OUT:
```

b.

```
                           1  2  3  4  5  6  7  8  9  1  1  1  1  1  1  1
                                                      0  1  2  3  4  5  6
     lwz   r8=a(r1)        F  D  E  C  W
     lwz   r12=b(r1,4)     F  ·  D  E  C  W
     lwz   r9=c(r1,8)      F  ·  ·  D  E  C  W
     lwz   r10=d(r1,12)    F  ·  ·  ·  D  E  C  W
     lwz   r11=e(r1,16)    F  ·  ·  ·  ·  D  E  C  W
     cmpi  cr0=r8,0        F  ·  ·  ·  ·  ·  D  E
     bc                    F  ·  ·  ·  S
ELSE,cr0/gt=false
IF:  add   r12=r8,r12      F  ·  ·  ·  ·  ·  ·  D  E  W
     add   r12=r12,r9            F  ·  ·  ·  ·  ·  D  E  W
     add   r12=r12,r10           F  ·  ·  ·  ·  ·  ·  D  E  W
     add   r4=r12,r11                                F  ·  ·  D  E  W
     stw   a(r1)=r4                                  F  ·  ·  ·  D  E  C
     b     OUT
ELSE: subf r12=r8,r12
     subf  r12=r12,r9
     subf  r12=r12,r10
     subf  r4=r12,r11
     stw   a(r1)=r4
OUT:
```

14.5
- write-write: I1, I3
- read-write: I2, I3

- write-read: I1. I2

14.6 **a.** True data dependency: I1, I2; I5, I6
Antidependency: I3, I4
Output dependency: I5, I6

b.

I1	f1	d1	e2	s1					
I2	f2	d2	▨	a1	a2	s2			
I3		f1	d1	▨	a1	a2	s1		
I4		f2	d2	m1	m2	m3	s2		
I5			f1	d1	e1	▨	▨	s1	
I6			f2	d2	▨	m1	m2	m3	s2

c.

I1	f1	d1	e2	s1					
I2	f2	d2	▨	a1	a2	s2			
I3		f1	d1	▨	a1	a2	s1		
I4		f2	d2	m1	m2	m3	s2		
I5			f1	d1	e1	s1			
I6			f2	d2	▨	m1	m2	m3	s2

d.

I3	f1	d1	a1	a2	s1		
I4	f2	d2	m1	m2	m3	s2	
lookahead window I5	f3	d3	e1	s1			
I6		f1	d1	m1	m2	m3	s2
I1		f2	d2	e2	s2		
I2			f1	d1	a1	a2	s1

14.7 The figure is from [SMIT95]. w = instruction dispatch; x = load/store units; y = integer units; z = floating-point units. Part a is the single-queue method, with no out of order issuing. Part b is a multiple-queue method; instructions issue from each queue in order, but the queues may issue out of order with respect to one another. Part c is a reservation station scheme; instructions may issue out of order.

14.8 **a.** **Figure 14.17d** is equivalent to Figure 12.17
Figure 14.17b is equivalent to Figure 12.25a
Figure 14.17c is equivalent to Figure 12.25b
Figure 14.17a: If the last branch was taken, predict that this branch will be taken; if the last branch was not taken, predict that this branch will not be taken.
Figure 14.7e: This is very close to Figure 14.7c. The difference is as follows. For Figure 14.7c, if there is a change in prediction followed by an error, the previous prediction is restored; this is true for either type of error. For Figure 14.7c, if there is a change in prediction from taken to not taken followed by an error, the prediction of taken is restored. However if there is a change in prediction from not taken to taken followed by an error, the taken prediction is retained.

b. The rationale is summarized in [OMON99, page 114]: "Whereas in loop-closing branches, the past history of an individual branch instruction is usually a good guide to future behavior, with more complex control-flow structures, such as sequences of IF-ELSE constructs or nestings of similar constructs, the direction of a branch is frequently affected by the directions taken by related branches. If we consider each of the possible paths that lead to a given nested branch, then it is clear that prediction in such a case should be based on the subhistories determined by such paths, i.e., how a particular branch is arrived at, rather than just on the individual history of a branch instruction. And in sequences of conditionals, there will be instances when the outcome of one condition-test depends on that of a preceding condition if the conditions are related in some way — for example, if part of the conditions are common."

CHAPTER 15
THE IA-64 ARCHITECTURE

ANSWERS TO QUESTIONS

15.1 **I-unit:** For integer arithmetic, shift-and-add, logical, compare, and integer multimedia instructions. **M-unit:** Load and store between register and memory plus some integer ALU operations. **B-unit:** Branch instructions. **F-unit:** Floating-point instructions.

15.2 The template field contains information that indicates which instructions can be executed in parallel.

15.3 A stop indicates to the hardware that one or more instructions before the stop may have certain kinds of resource dependencies with one or more instructions after the stop.

15.4 **Predication** is a technique whereby the compiler determines which instructions may execute in parallel. With **predicated execution**, every IA-64 instruction includes a reference to a 1-bit predicate register, and only executes if the predicate value is 1 (true).

15.5 Predicates enable the processor to speculatively execute both branches of an if statement and only commit after the condition is determined.

15.6 With control speculation, a load instruction is moved earlier in the program and its original position replaced by a check instruction. The early load saves cycle time; if the load produces an exception, the exception is not activated until the check instruction determines if the load should have been taken.

15.7 Associated with each register is a NaT bit used to track deferred speculative exceptions. If a ld.s detects an exception, it sets the NaT bit associated with the target register. If the corresponding chk.s instruction is executed, and if the NaT bit is set, the chk.s instruction branches to an exception-handling routine.

15.8 With **data speculation**, a load is moved before a store instruction that might alter the memory location that is the source of the load. A subsequent check is made to assure that the load receives the proper memory value.

15.9 S**oftware pipelining** is a technique in which instructions from multiple iterations of a loop are enabled to execute in parallel. Parallelism is achieved by grouping together instructions from different iterations. Hardware pipelining refers to the use of a physical pipeline as part of the hardware

15.10 **Rotating registers** are used for software pipelining. During each iteration of a software-pipeline loop, register references within these ranges are automatically incremented. **Stacked registers** implement a stack.

Answers to Problems

15.1 Eight. The operands and result require 7 bits each, and the controlling predicate 6. A major opcode is specified by 4 bits; 38 bits of the 41-bit syllable are committed, leaving 3 bits to specify a suboperation. Source: [MARK00]

15.2 Table 15.3 reveals that any opcode can be interpreted as referring to on of 6 different execution units (M, B, I, L, F, X). So, the potential maximum number of different major opcodes is $2^4 \times 6 = 96$.

15.3 16

15.4 **a.** Six cycles. The single floating-point unit is the limiting factor.
b. Three cycles.

15.5 The pairing must not exceed a sum of two M or two I slots with the two bundles. For example, two bundles, both with template 00, or two bundles with templates 00 and 01 could not be paired because they require 4 I-units. Source: [EVAN03]

15.6 Yes. On IA-64s with fewer floating-point units, more cycles are needed to dispatch each group. On an IA-64 with two FPUs, each group requires two cycles to dispatch. A machine with three FPUs will dispatch the first three floating-point instructions within a group in one cycle, and the remaining instruction in the next. Source: [MARK00]

15.7

p1	comparison	p2	p3
not present	0	0	1
not present	1	1	0
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

15.8 **a.** (3) and (4); (5) and (6)
b. The IA-64 template field gives a great deal of flexibility, so that many combinations are possible. One obvious combination would be (1), (2), and (3) in the first instruction; (4), (5), and (6) in the second instruction; and (7) in the third instruction.

15.9 Branching to label `error` should occur if and only if at least one of the 8 bytes in register `r16` contains a non-digit ASCII code. So the comments are not inaccurate but are not as helpful as they could be. Source: [EVAN03]

15.10 a.

```
      mov     r1, 0
      mov     r2, 0
      ld      r3, addr(A)
L1:   ld      r4, mem(r3+r2)
      bge     r4, 50, L2
      add     r5, r5, 1
      jump    L3
L2:   add     r6, r6, 1
L3:   add     r1, r1, 1
      add     r2, r2, 4
      blt     r1, 100, L1
```

b.

```
      mov     r1, 0
      mov     r2, 0
      ld      r3, addr(A)
L1:   ld      r4, mem(r3+r2)
      cmp.ge  p1, p2 = r4. 50
      (p2)    add r5 = 1, r5
      (p1)    add r6 = 1, r6
      add     r1 = 1, r1
      add     r2 = 4, r2
      blt     r1, 100, L1
```

15.11 a.

```
fmpy   t = p, q     //floating-point multiply
ldf.a  c = [rj];;   //advanced floating point load
                    //load value stored in location specified by address
                    //in register rj; place value in floating-point register c
                    //assume rj points to a[j]
stf    [ri] = t;;   //store value in floating-point register t in location
                    //specified by address in register ri
                    //assume ri points to a[i]
ldf.c  c = [rj];;   //executes only if ri = rj
```

If the advanced load succeeded, the ldf.c will complete in one cycle, and c can be used in the following instruction. The effective latency of the ldf.a instruction has been reduced by the latency of the floating-point multiplication. The stf and ldf.c cannot be in the same instruction group, because there may be a read-after -write dependency.

b.

```
        fmpy       t = p, q
        cmp.ne     p8, p9 = ri, rj;;
(p8)    ldf        c = [rj];;          //p8 ⇒ no conflict
        stf        [ri] = t;;          //if ri = rj, then c = t
(p9)    mov        c = t;;
```

c. In the predicated version, the load begins one cycle later than with the advanced load. Also, two predicated registers are required. Source: [MARK00]

15.12 **a.** The number of output registers is

SOO = SOF – SOL = 48 – 16 = 32

b. Because the stacked register group starts at r32, the local register and output register groups consist of:

Local register group: r32 through r47
Output register group: r48 through r63
Source: [TRIE01]

Chapter 16
Control Unit Operation

Answers to Questions

16.1 The operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle. This sequence of instruction cycles is not necessarily the same as the **written sequence** of instructions that make up the program, because of the existence of branching instructions. The actual execution of instructions follows a **time sequence** of instructions.

16.2 A micro-operation is an elementary CPU operation, performed during one clock pulse. An instruction consists of a sequence of micro-operations.

16.3 The control unit of a processor performs two tasks: (1) It causes the processor to execute micro-operations in the proper sequence, determined by the program being executed, and (2) it generates the control signals that cause each micro-operation to be executed.

16.4 **1.** Define the basic elements of the processor. **2.** Describe the micro-operations that the processor performs. **3.** Determine the functions that the control unit must perform to cause the micro-operations to be performed.

16.5 **Sequencing:** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed. **Execution:** The control unit causes each micro-operation to be performed.

16.6 The **inputs** are: **Clock:** This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time, or the clock cycle time. **Instruction register:** The opcode of the current instruction is used to determine which micro-operations to perform during the execute cycle. **Flags:** These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. **Control signals from control bus:** The control bus portion of the system bus provides signals to the control unit, such as interrupt signals and acknowledgments. The **outputs** are: **Control signals within the processor:** These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions. **Control signals to control bus:** These are also of two types: control signals to memory, and control signals to the I/O modules.

16.7 **(1)** Those that activate an ALU function. **(2)** those that activate a data path. **(3)** Those that are signals on the external system bus or other external interface

16.8 In a hardwired implementation, the control unit is essentially a combinatorial circuit. Its input logic signals are transformed into a set of output logic signals, which are the control signals.

Answers to Problems

16.1 Consider the instruction SUB R1, X, which subtracts the contents of location X from the contents of register R1, and places the result in R1.

t_1:	MAR	←	(IR(address))
t_2:	MBR	←	Memory
t_3:	MBR	←	Complement(MBR)
t_4:	MBR	←	Increment(MBR)
t_5:	R1	←	(R1) + (MBR)

16.2

LOAD AC:	t_1:	MAR	←	(IR(address))	C_8
	t_2:	MBR	←	Memory	C_5,C_R
	t_3:	AC	←	(MBR)	C_{10}
STORE AC	t_1:	MAR	←	(IR(address))	C_8
	t_2:	MBR	←	(AC)	C_{11}
	t_3:	Memory	←	(MBR)	C_{12},C_W
ADD AC	t_1:	MAR	←	(IR(address))	C_8
	t_2:	MBR	←	Memory	C_5,C_R
	t_3:	AC	←	(AC) + (MBR)	C_{ALU},C_7,C_9

Note: There must be a delay between the activation of C_8 and C_9, and one or more control signals must be sent to the ALU. All of this would be done during one or more clock pulses, depending on control unit design.

AND AC	t_1:	MAR	←	(IR(address))	C_8
	t_2:	MBR	←	Memory	C_5,C_R
	t_3:	AC	←	(AC) AND (MBR)	C_{ALU},C_7,C_9
JUMP	t_1:	PC	←	IR(address)	C_3
JUMP if AC= 0	Test AC and activate C_3 if AC = 0				
Complement AC	t_1:	AC	←	($\overline{\text{AC}}$)	C_{ALU},C_7,C_9

16.3 **a.** Time required = propagation time + copy time
= 30 ns

b. Incrementing the program counter involves two steps:

(1)	Z	←	(PC) + 1
(2)	PC	←	(Z)

The first step requires 20 + 100 + 10 = 130 ns.
The second step requires 30 ns.
Total time = 160 ns.

16.4 **a.** t_1: Y ← (IR(address))
t_2: Z ← (AC) + (Y)
t_3: AC ← (Z)

b. t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: Y ← (MBR)
t_4: Z ← (AC) + (Y)
t_5: AC ← (Z)

c. t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: MAR ← (MBR)
t_4: MBR ← Memory
t_5: Y ← (MBR)
t_6: Z ← (AC) + (Y)
t_7: AC ← (Z)

16.5 Assume configuration of Figure 10.14a. For the push operation, assume value to be pushed is in register R1.

POP: t_1: SP ← (SP) + 1

PUSH: t_1: SP ← (SP) – 1
MBR ← (R1)
t_2: MAR ← (SP)
t_3: Memory ← (MBR)

Chapter 17
Microprogrammed Control

Answers to Questions

17.1 A **hardwired control unit** is a combinatorial circuit, in which input logic signals are transformed into a set of output logic signals that function as the control signals. In a **microprogrammed control unit**, the logic is specified by a microprogram. A microprogram consists of a sequence of instructions in a microprogramming language. These are very simple instructions that specify micro-operations.

17.2 **1.** To execute a microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed. **2.** If the condition indicated by the condition bits is false, execute the next microinstruction in sequence. **3.** If the condition indicated by the condition bits is true, the next microinstruction to be executed is indicated in the address field.

17.3 The control memory contains the set of microinstructions that define the functionality of the control unit.

17.4 The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating where to go next.

17.5 In a **horizontal microinstruction** every bit in the control field attaches to a control line. In a **vertical microinstruction**, a code is used for each action to be performed and the decoder translates this code into individual control signals.

17.6 **Microinstruction sequencing:** Get the next microinstruction from the control memory. **Microinstruction execution:** Generate the control signals needed to execute the microinstruction.

17.7 The degree of packing relates to the degree of identification between a given control task and specific microinstruction bits. As the bits become more **packed**, a given number of bits contains more information. An unpacked microinstruction has no coding beyond assignment of individual functions to individual bits.

17.8 **Hard microprograms** are generally fixed and committed to read-only memory. **Soft microprograms** are more changeable and are suggestive of user microprogramming.

17.9 Two approaches can be taken to organizing the encoded microinstruction into fields: functional and resource. The **functional encoding** method identifies functions within the machine and designates fields by function type. For example, if various sources can be used for transferring data to the accumulator, one field can be designated for this purpose, with each code specifying a different source.

Resource encoding views the machine as consisting of a set of independent resources and devotes one field to each (e.g., I/O, memory, ALU).

17.10 Realization of computers. Emulation. Operating system support. Realization of special-purpose devices. High-level language support. Microdiagnostics. User Tailoring.

Answers to Problems

17.1 The multiply instruction is implemented by locations 27 through 37 of the microprogram in Table 17.2. It involves repeated additions.

17.2 Assume that the microprogram includes a fetch routine that starts at location 0 and a BRM macroinstruction that starts at location 40.

```
40: IF ($AC_0$ = 1) THEN CAR ← 42; ELSE CAR ← (CAR) + 1
41: CAR ← 43; PC ← (PC) + 1
42: PC ← (IR(address))
43: CAR ← 0
```

17.3 **a.** These flags represent Boolean variables that are input to the control unit logic. Together with the time input and other flags, they determine control unit output.

b. The phase of the instruction cycle is implicit in the organization of the microprogram. Certain locations in the microprogram memory correspond to each of the four phases.

17.4 **a.** Three bits are needed to specify one of 8 flags.
b. 24 – 13 – 3 = 8
c. 2^8 = 256 words × 24 bits/word = 6144 bits.

17.5 Two of the codes in the address selection field must be dedicated to that purpose. For example, a value of 000 could correspond to no branch, a value of 111 could correspond to unconditional branch.

17.6 An address for control memory requires 10 bits (2^{10} = 1024). A very simple mapping would be this:

opcode	XXXXX
control address	00XXXXX000

This allows 8 words between successive addresses.

17.7 A field of 5 bits yields $2^5 - 1 = 31$ different combinations of control signals. A field of 4 bits yields $2^4 - 1 = 15$ different combinations, for a total of 46.

17.8 A 20-bit format consisting of the following fields:

A1 (4 bits): specify register to act as one of the inputs to ALU
A2 (4 bits): specifies other ALU input
A3 (4 bits): specifies register to store ALU result
AF (5 bits): specifies ALU function
SH (3 bits): specifies shift function

In addition, an address field for sequencing is needed.

CHAPTER 18
PARALLEL PROCESSING

ANSWERS TO QUESTIONS

18.1 **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets.

18.2 **1.** There are two or more similar processors of comparable capability. **2.**These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor. **3.** All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device. **4.** All processors can perform the same functions (hence the term *symmetric*). **5.** The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

18.3 **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type. **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance. **Incremental growth:** A user can enhance the performance of a system by adding an additional processor. **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

18.4 **Simultaneous concurrent processes:** OS routines need to be reentrant to allow several processors to execute the same IS code simultaneously. With multiple processors executing the same or different parts of the OS, OS tables and management structures must be managed properly to avoid deadlock or invalid operations. **Scheduling:** Any processor may perform scheduling, so conflicts must be avoided. The scheduler must assign ready processes to available processors. **Synchronization:** With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering. **Memory management:** Memory management on a multiprocessor must deal with all of the issues found on uniprocessor machines, as is discussed in Chapter 8. In addition, the operating system needs to exploit the available hardware parallelism, such as multiported memories, to achieve the best performance. The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement. **Reliability and fault tolerance:** The operating system should provide graceful degradation in the face of processor failure. The scheduler and other portions of the operating system must recognize the loss of a processor and restructure management tables accordingly.

18.5 Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem. In hardware schemes, the cache coherence logic is implemented in hardware.

18.6 **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache. **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache. **Shared:** The line in the cache is the same as that in main memory and may be present in another cache. **Invalid:** The line in the cache does not contain valid data.

18.7 **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest standalone machines. **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system. **High availability:** Because each node in a cluster is a standalone computer, the failure of one node does not mean loss of service. **Superior price/performance:** By using commodity building blocks, it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

18.8 The function of switching an applications and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**. A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**.

18.9 **Uniform memory access (UMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor to all regions of memory is the same. The access times experienced by different processors are the same. **Nonuniform memory access (NUMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ. **Cache-coherent NUMA (CC-NUMA):** A NUMA system in which cache coherence is maintained among the caches of the various processors.

Answers to Problems

18.1 **a.** MIPS rate = $[n\alpha + (1 - \alpha)]\,x = (n\alpha - \alpha + 1)x$

b. $\alpha = 0.6$

Source: [HWAN93]

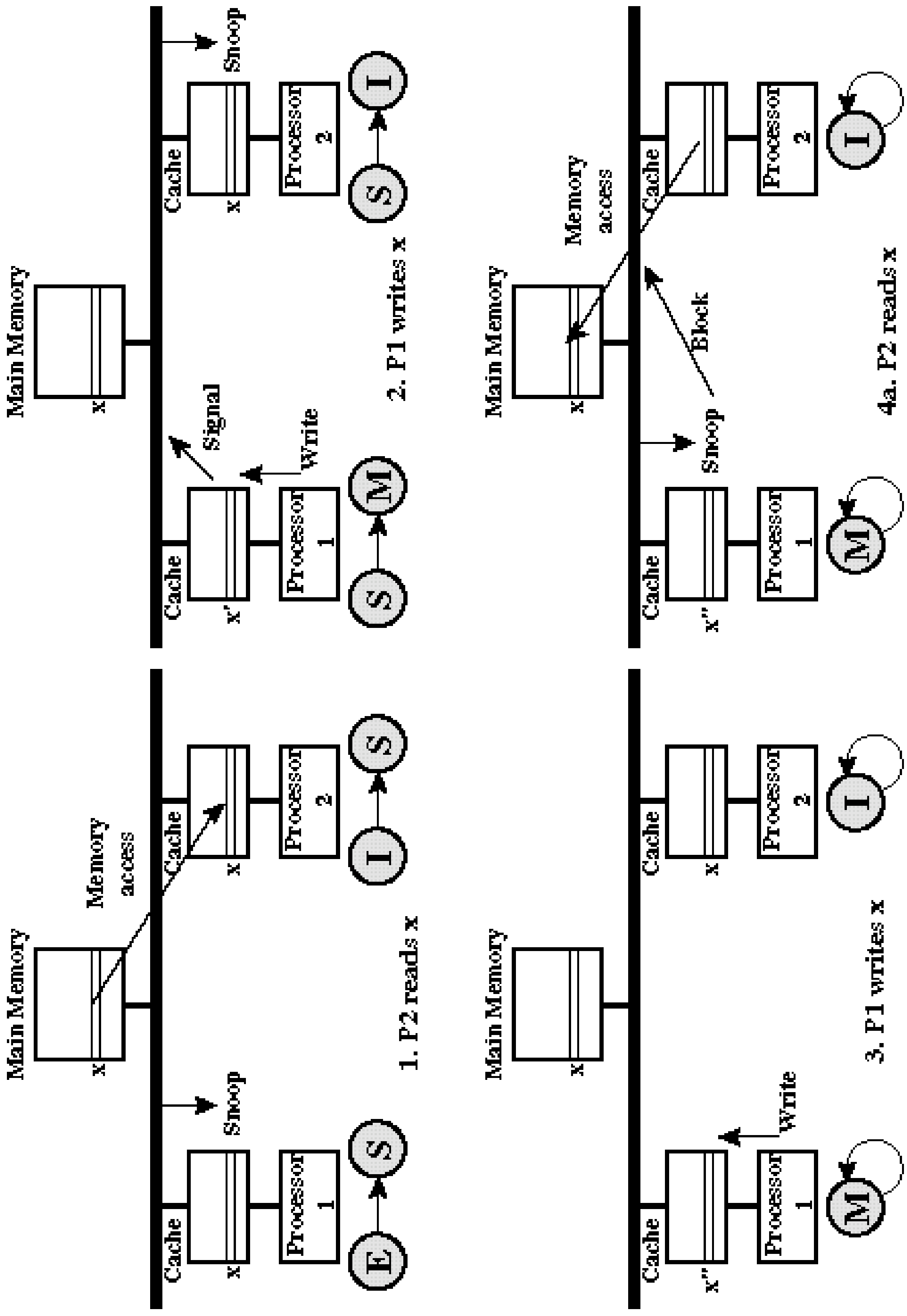
18.2 **a.** If this conservative policy is used, at most 20/4 = 5 processes can be active simultaneously. Because one of the drives allocated to each process can be idle most of the time, at most 5 drives will be idle at a time. In the best case, none of the drives will be idle.

b. To improve drive utilization, each process can be initially allocated with three tape drives, with the fourth drive allocated on demand. With this policy, at most $\lfloor 20/3 \rfloor = 6$ processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum number is 2.

Source: [HWAN93]

18.3 Processor A has a block of memory in its cache. When A writes to the block the first time, it updates main memory. This is a signal to other processors to invalidate their own copy (if they have one) of that block of main memory. Subsequent writes by A to that block only affect A's cache. If another processor attempts to read the block from main memory, the block is invalid. Solution: If A makes a second update, it must somehow tag that block in main memory as being invalid. If another processor wants the block, it must request that A write the latest version from its cache to main memory. All of this requires complex circuitry.

18.4

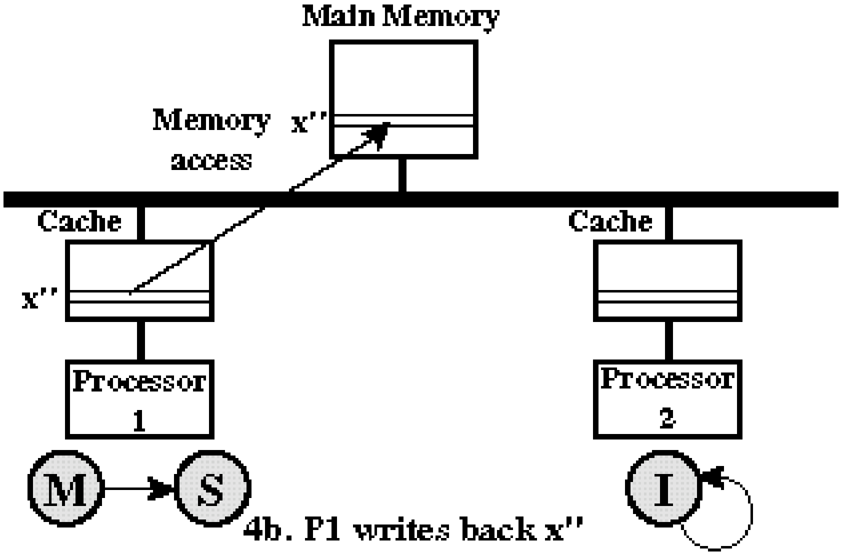


1. P2 reads x

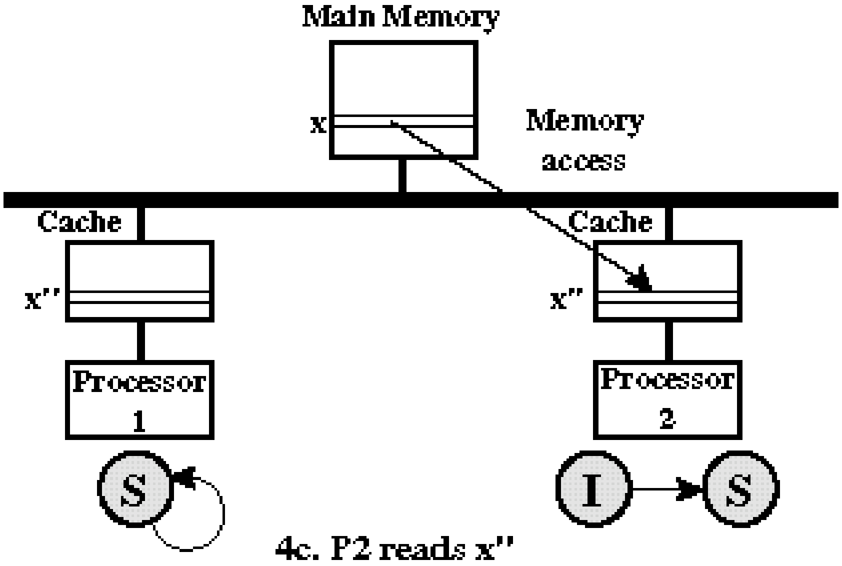
2. P1 writes x

3. P1 writes x

4a. P2 reads x



4b. P1 writes back x''



4c. P2 reads x''

18.5 a. This is the simplest possible cache coherence protocol. It requires that all processors use a write-through policy. If a write is made to a location cached in remote caches, then the copies of the line in remote caches are invalidated. This approach is easy to implement but requires more bus and memory traffic because of the write-through policy.

b. This protocol makes a distinction between shared and exclusive states. When a cache first loads a line, it puts it in the shared state. If the line is already in the modified state in another cache, that cache must block the read until the line is updated back to main memory, similar to the MESI protocol. The difference between the two is that the shared state is split into the shared and exclusive states for MESI. This reduces the number of write-invalidate operations on the bus.

18.6 If the L1 cache uses a write-through policy, as is done on the S/390 described in Section 18.2, then the L1 cache does not need to know the M state. If the L1 cache uses a write-back policy, then a full MESI protocol is needed between L1 and L2.

18.7 If only the L1 cache is used, then 89% of the accesses are to L1 and the remaining 11% of the accesses are to main memory. Therefore, the average penalty is (1 × 0.89) + (32 × 0.11) = 4.41. If both L1 and L2 are present, the average penalty is (1 × 0.89) + (5 × 0.05) + (32 × 0.06) = 3.06. This normalizes to 3.06/4.41 = 0.69. Thus, with the addition of the L2 cache, the average penalty is reduced to 69% of that with only one cache. If all three caches are present, the average penalty is (1 × 0.89) + (5 × 0.05) + (14 × 0.03) + (32 × 0.03) = 2.52, and normalized average penalty is 2.52/4.41 = 0.57. The reduction of the average penalty from 0.69 to 0.57 would seem to justify the inclusion of the L3 cache.

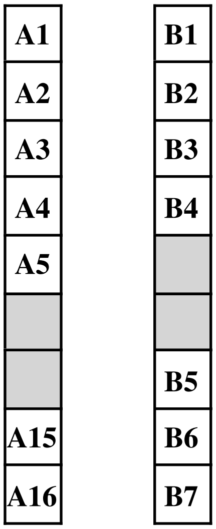
18.8 **a.** $t_a = f_i[H_i c + (1 - H_i)(b + c) + (1 - f_i)(H_d c) + (1 - H_d)((b + c)(1 - f_d) + (2b + c)f_d)]$

b. $t'_a = \mathrm{t_a} + (1 - f_i)f_{inv}i$ Source: [HWAN93]

18.9 **a.** chip multiprocessor
b. interleaved multithreading superscalar
c. blocked multithreading superscalar
d. simultaneous multithreading

18.10 [UNGE03] refers to these as horizontal losses and vertical losses, respectively. With a horizontal loss, full parallelism is not achieved; that is, fewer instructions are dispatched than the hardware would allow. With a vertical loss, the dispatching mechanism is stalled because no new instructions can be accommodated due to latency issues.

18.11 **a.**



b. The two pipelines are operating independently on two separate processors on the same chip. Therefore, the diagrams of Figure 18.24 and part (a) of this solution apply.

c. We assume that the A thread requires a latency of two clock cycles before it is able to execute instruction A15, and we assume that the interleaving mechanism is able to use the same thread on two successive clock cycles if necessary.

A1	A2
B1	B2
A3	A4
B3	B4
A5	
B5	B6
B7	
A15	A16

instruction issue diagram

CO	F0	EI	WO	CO	F0	EI	WO
A1				A2			
B1	A1			B2	A2		
A3	B1	A1		A4	B2	A2	
B3	A3	B1	A1	B4	A4	B2	A2
A5	B3	A3	B1		B4	A4	B2
B5	A5	B3	A3	B6		B4	A4
B7	B5	A5	B3		B6		B4
A15	B7	B5	A5	A16		B6	
	A15	B7	B5		A16		B6
		A15	B7			A16	
			A15				A16

pipeline execution diagram

d.

A1	A2
A3	A4
A5	
B1	B2
B3	B4
A15	A16
B5	B6
B7	

instruction issue diagram

CO	F0	EI	WO	CO	F0	EI	WO
A1				A2			
A3	A1			A4	A2		
A5	A3	A1			A4	A2	
	A5	A3	A1			A4	A2
B1		A5	A3	B2			A4
B3	B1		A5	B4	B2		
	B3	B1			B4	B2	
A15		B3	B1	A16		B4	B2
B5	A15		B3	B6	A16		B4
B7	B5	A15			B6	A16	
	B7	B5	A15			B6	A16
		B7	B5				B6
			B7				

pipeline execution diagram

e.

A1	A2	B1	B2
A3	A4	B3	B4
A5			
B5	B6	B7	
A15	A16		

instruction issue diagram

CO	F0	EI	WO	CO	F0	EI	WO	CO	F0	EI	WO	CO	F0	EI	WO
A1				A2				B1				B2			
A3	A1			A4	A2			B3	B1			B4	B2		
A5	A3	A1			A4	A2			B3	B1			B4	B2	
	A5	A3	A1			A4	A2			B3	B1			B4	B2
B5		A5	A3	B6			A4	B7			B3				B4
A15	B5		A5	A16	B6				B7						
	A15	B5			A16	B6				B7					
		A15	B5			A16	B6				B7				
			A15				A16								

pipeline execution diagram

18.12 **a.** Sequential execution time = 1664 processor cycles.
b. SIMD execution time = 26 cycles.
c. Speedup factor = 64. Source: [HWAN93]

18.13 To begin, we can distribute the outer loop without affecting the computation.

```
        DO 20A  I = 1, N
        B (I,1) = 0
20A     CONTINUE
        DO 20B  I = 1, N
        DO 10    J = 1, M
        A(I) = A(I) + B (I, J) * C (I, J)
10      CONTINUE
20B     CONTINUE
        DO 20C   I = 1, N
        D (I) = E (I) + A (I)
20C     CONTINUE
```

Using vectorized instructions:

```
        B (I,1) = 0  (I = 1, N)
        DO 20B  I= 1, N
        A(I) = A(I) + B(I, J) * C(I, J)    (J = 1, M)
20B     CONTINUE
        D(I) = E(I0 + A(I)   (I = 1, N)
```

18.14 **a.** One computer executes for a time T. Eight computers execute for a time T/4, which would take a time 2T on a single computer. Thus the total required time on a single computer is 3T. Effective speedup = 3. α = 0.75.
b. New speedup = 3.43

18.15 **a.** Sequential execution time = 1,051,628 cycles

b. Speedup = 16.28

c. Each computer is assigned 32 iterations balanced between the beginning and end of the I-loop.

d. The ideal speedup of 32 is achieved.

Source: [HWAN93]

18.16 **a.** The I loop requires *N* cycles, as does the J loop. With the L4 statement, the total is $2N + 1$.

b. The sectioned I loop can be done in *L* cycles. The sectioned J loop produces *M* partial sums in *L* cycles. Total = $2L + l(k + 1)$.

c. Sequential execution of the original program takes $2N = 2^{21}$ cycles. Parallel execution requires $2^{13} + 1608 = 9800$ cycles. This is a speedup factor of approximately 214 ($2^{21}/9800$). Therefore, an efficiency of 214/256 = 83.6% is achieved.

Appendix A
Number Systems

Answers to Problems

A.1 **a.** 12 **b.** 3 **c.** 28 **d.** 60 **e.** 42

A.2 **a.** 28.375 **b.** 51.59375 **c.** 682.5

A.3 **a.** 1000000 **b.** 1100100 **c.** 1101111 **d.** 10010001 **e.** 11111111

A.4 **a.** 100010.11 **b.** 11001.01 **c.** 11011.0011

A.5 A BAD ADOBE FACADE FADED (Source: [KNUT98])

A.6 **a.** 12 **b.** 159 **c.** 3410 **d.** 1662 **e.** 43981

A.7 **a.** 15.25 **b.** 211.875 **c.** 4369.0625 **d.** 2184.5 **e.** 3770.75

A.8 **a.** 10 **b.** 50 **c.** A00 **d.** BB8 **e.** F424

A.9 **a.** CC.2 **b.** FF.E **c.** 277.4 **d.** 2710.01

A.10 **a.** 1110 **b.** 11100 **c.** 101001100100 **d.** 11111.11 **e.** 1000111001.01

A.11 **a.** 9.F **b.** 35.64 **c.** A7.EC

A.12 $1/2^k = 5^k/10^k$

Appendix B
Digital Logic

Answers to Problems

B.1

A	B	C	**a**	**b**	**c**	**d**
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	1
1	0	0	0	1	0	1
1	0	1	0	0	1	1
1	1	0	0	0	1	0
1	1	1	1	1	0	0

B.2 Recall the commutative law: $AB = BA; \quad A + B = B + A$

a. $A\overline{B} + CDE + \overline{C}DE$

b. $AB + AC$

c. $(LMN)(AB)(CDE)$

d. $F(K + R) + SV + W\overline{X}$

B.3 **a.** $F = \overline{V} \bullet \overline{A} \bullet \overline{L}$. This is just a generalization of DeMorgan's Theorem, and is easily proved.

b. $F = \overline{ABCD}$. Again, a generalization of DeMorgan's Theorem.

B.4 **a.** $A = ST + VW$

b. $A = TUV + Y$

c. $A = F$

d. $A = ST$

e. $A = D + \overline{E}$

f. $A = YZ(W + X + YZ) = YZ$

g. $A = C$

B.5 $A \text{ XOR } B = A\overline{B} + \overline{A}B$

B.6 $ABC = \text{NOR}(\overline{A}, \overline{B}, \overline{C})$

B.7 $Y = \text{NAND}(A, B, C, D) = \overline{ABCD}$

B.8 **a.**

X_1	X_2	X_3	X_4	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7
0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0	1
0	0	1	1	1	0	1	1	0	1	1
0	1	0	0	0	1	1	1	0	1	0
0	1	0	1	1	1	0	1	0	1	1
0	1	1	0	0	1	0	1	1	1	1
0	1	1	1	1	0	1	0	0	1	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	0
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. All of the terms have the form illustrated as follows:

$$Z_5 = \overline{X_1X_2X_3X_4} + \overline{X_1X_2}\, X_3\, \overline{X_4} + \overline{X_1}\, X_2X_3\, \overline{X_4} + \overline{X_1}\, X_2X_3X_4$$

c. Whereas the SOP form lists all combinations that produce an output of 1, the POS lists all combinations that produce an output of 0.

For example,

$$Z_3 = \overline{(\overline{X_1}\, X_2\, \overline{X_3}\, X_4)}\ \overline{(\overline{X_1}\, X_2X_3\, \overline{X_4})}$$

$$= (X_1\, \overline{X_2}\, X_3\, \overline{X_4})\, (X_1\, \overline{X_2}\, \overline{X_3}\, X_4)$$

B.9 Label the 8 inputs I_0, ... , I_7 and the select lines S_0, S_1, S_2.

$$\overline{S_1}F = I_0 + I_1\, \overline{S_0S_1S_2} + I_2\, \overline{S_0}\, S_1\, \overline{S2} + I_3\, \overline{S_0}\, S_1\, S_2$$
$$+ I_4\, S_0\overline{S_1S_2} + I_5\, S_0\overline{S_1}S_2 + I_6\, S_0\overline{S_1}S_2 + I_7\, S_0\, S_1\, S_2$$

B.10 Add a data input line and connect it to the input side of each AND gate.

B.11 Define the input leads as B_2, B_1, B_0 and the output leads as G_2, G_1, G_0. Then

$$G_2 = B_2$$
$$G_1 = B_2\overline{B_1} + \overline{B_2}B_1$$
$$G_0 = B_1\overline{B_0} + \overline{B_1}B_0$$

B.12 The Input is $A_4A_3A_2A_1A_0$. Use $A_2A_1A_0$ as the input to each of the 3 × 8 decoders. There are a total of 32 outputs from these four 3 × 8 decoders. Use A_4A_3 as input to a 2 × 4 decoder and have the four outputs go to the enable leads of the four 3 × 8 decoders. The result is that one and only one of the 32 outputs will have a value of 1.

B.13 SUM $= A \oplus B \oplus C$
CARRY $= AB \oplus AC \oplus BC$

B.14 **a.** The carry to the second stage is available after 20 ns; the carry to the third stage is available 20 ns after that, and so on. When the carry reaches the 32nd stage, another 30 ns are needed to produce the final sum. Thus

$$T = 31 \times 20 + 30 = 650 \text{ ns}$$

b. Each 8-bit adder produces a sum in 30 ns and a carry in 20 ns. Therefore,

$$T = 3 \times 20 + 30 = 90 \text{ ns}$$