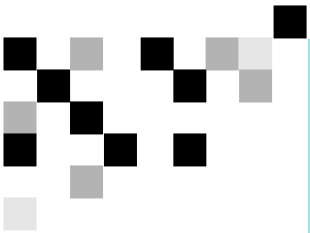


SOLUTIONS TO PROBLEMS

SOLUTIONS TO CHAPTER 1 PROBLEMS

1.1 Computing power increases by a factor of 2 every 18 months, which generalizes to a factor of 2^x every $18x$ months. If we want to figure the time at which computing power increases by a factor of 100, we need to sove $2^x = 100$, which reduces to $x = 6.644$. We thus have $18x = 18\times(6.644 \text{ months}) = 120$ months, which is 10 years.

SOLUTIONS TO CHAPTER 2 PROBLEMS

2.1 (a) [+999.999, –999.999]

(b) .001 (Note that *error* is 1/2 the precision, which would be .001/2 = .0005 for this problem.)

2.2 (a) 101111

(b) 111011

(c) 531

(d) 22.625

(e) 202.22

2.3 (a) 27

(b) 000101

(c) 1B

(d) 110111.111

(e) 1E.8

2.4 $2\times3^{-1} + 0\times3^{-2} + 1\times3^{-3} = 2/3 + 0 + 1/27 = 19/27$

2.5 37.3

2.6 $(17.5)_{10} \cong (122.11)_3 = (17.4)_{10}$

2.7 –8

2.8 0

2.9 0011 0000 0101

2.10 0110 1001 0101

2.11 One's complement has two representations for zero, whereas two's complement has one representation for zero, thus two's complement can represent one more integer.

2.12

	5-bit signed magnitude	5-bit excess 16
Largest number	+15	+15
Smallest number	–15	–16
No. of distinct numbers	31	32

2.13

Base 2 scientific notation	Floating point representation		
	Sign	Exponent	Fraction
-1.0101×2^{-2}	1	001	0101
$+1.1 \times 2^{2}$	0	101	1000
$+1.0 \times 2^{-2}$	0	001	0000
-1.1111×2^{3}	1	110	1111

2.14 (a) -.02734375

(b) $(14.3)_6 = (10.5)_{10} = (A.8)_{16} = .A8 \times 16^1 =$
0 1000001 10101000 00000000 00000000

2.15 (a) decrease; (b) not change; (c) increase; (d) not change

2.16 (a) –.5; (b) decrease; (c) 2^{-5}; (d) 2^{-2}; (e) 33

2.17 $(107.15)_{10}$ = 1101011.00100110011001100

0 1000111 11010110 01001100 11001100

2.18 (a) $+1.011 \times 2^4$

(b) -1.0×2^1

(c) -0

(d) $-\infty$

(e) +NaN

(f) $+1.1001 \times 2^{-126}$

(g) $+1.01101 \times 2^{-124}$

2.19 (a) 0 10000100 1011 0000 0000 0000 0000 000

(b) 0 00000000 0000 0000 0000 0000 0000 000

(c) 1 01111111110 0011 1000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0000 0000

(d) 1 11111111 1011 1000 0000 0000 0000 000

2.20 (a) $(2 - 2^{-23}) \times 2^{127}$

(b) 1.0×2^{-126}

(c) $2^{-23} \times 2^{-126} = 2^{-149}$

(d) $2^{-23} \times 2^{-126} = 2^{-149}$

(e) $2^{-23} \times 2^{127} = 2^{104}$

(f) $2 \times (127 - -126 + 1) \times 1 \times 2^{23} + 1 = 254 \times 2^{24} + 1$

2.21 The distance from zero to the first representable number is greater than the gap size for the same exponent values.

2.22 If we remove the leftmost digit, there is no way to know which value from 1 to 15 should be restored.

2.23 No, because there are no unused bit patterns.

2.24 No. The exponent determines the position of the radix point in the fixed point equivalent representation of a number. This will almost always be different between the orginal and converted numbers, and so the value of the exponent will be different in general.

SOLUTIONS TO CHAPTER 3 PROBLEMS

3.1

```
  1 0 1 1 0        1 1 1 1 0        1 1 1 1 1
+ 1 0 1 1 1      + 1 1 1 0 1      + 0 1 1 1 1
-----------      -----------      -----------
  0 1 1 0 1        1 1 0 1 1        0 1 1 1 0
```

Overflow | No overflow | No overflow

3.2

```
  1 1 0 0 <-- borrows
  0 1 0 1
- 0 1 1 0
-----------

1 1 1 1 1 (No overflow)
^
|__ borrow is discarded in a two's complement representation
```

3.3

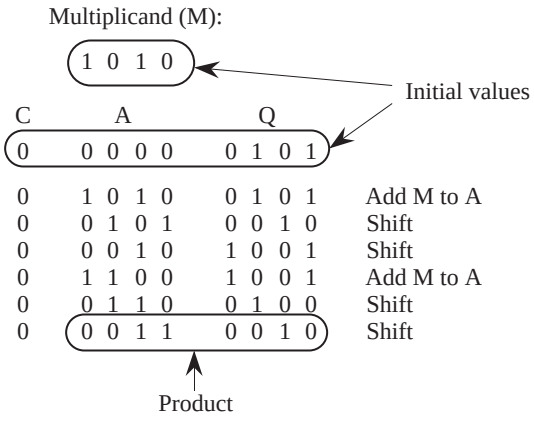
```
Two's complement                One's complement
  1 0 1 1.1 0 1                   1 0 1 1.1 0 1
+ 0 1 1 1.0 1 1                 + 0 1 1 1.0 1 1
-----------------               -----------------

  0 0 1 1.0 0 0 (no overflow)     0 1 0 0.0 0 0 (no overflow)
```

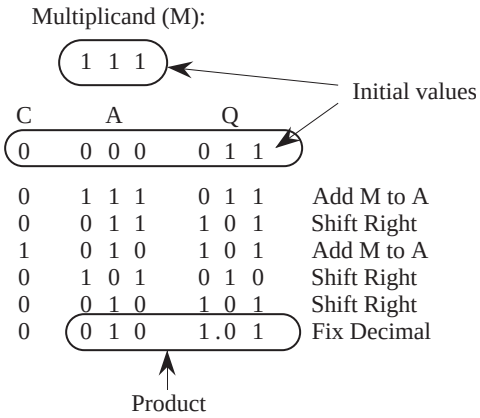
Note that for the one's complement solution, that the end-around carry is added into the 1's posi-

tion.

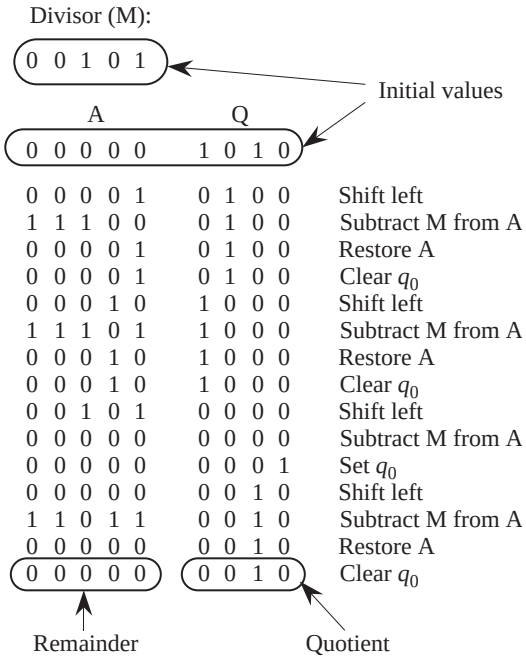
3.4


Multiplicand (M):
1 0 1 0
Initial values
C A Q
0 0 0 0 0 0 1 0 1
0 1 0 1 0 1 0 1 Add M to A
0 0 1 0 1 0 0 1 0 Shift
0 0 0 1 0 1 0 0 1 Shift
0 1 1 0 0 1 0 0 1 Add M to A
0 0 1 1 0 0 1 0 0 Shift
0 0 0 1 1 0 0 1 0 Shift
Product

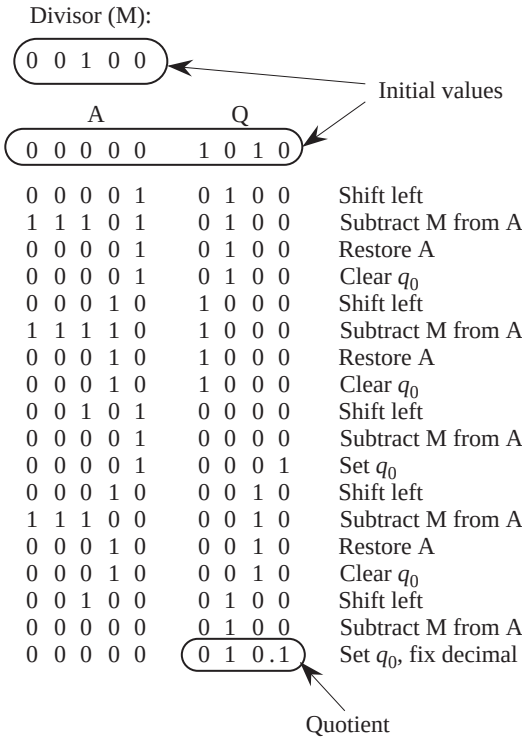
3.5


Multiplicand (M):
1 1 1
Initial values
C A Q
0 0 0 0 0 1 1
0 1 1 1 0 1 1 Add M to A
0 0 1 1 1 0 1 Shift Right
1 0 1 0 1 0 1 Add M to A
0 1 0 1 0 1 0 Shift Right
0 0 1 0 1 0 1 Shift Right
0 0 1 0 1.0 1 Fix Decimal
Product

3.6


Divisor (M):
0 0 1 0 1
Initial values
A
Q
0 0 0 0 0 1 0 1 0
0 0 0 0 1 0 1 0 0 Shift left
1 1 1 0 0 0 1 0 0 Subtract M from A
0 0 0 0 1 0 1 0 0 Restore A
0 0 0 0 1 0 1 0 0 Clear q_0
0 0 0 1 0 1 0 0 0 Shift left
1 1 1 0 1 1 0 0 0 Subtract M from A
0 0 0 1 0 1 0 0 0 Restore A
0 0 0 1 0 1 0 0 0 Clear q_0
0 0 1 0 1 0 0 0 0 Shift left
0 0 0 0 0 0 0 0 0 Subtract M from A
0 0 0 0 0 0 0 0 1 Set q_0
0 0 0 0 0 0 0 1 0 Shift left
1 1 0 1 1 0 0 1 0 Subtract M from A
0 0 0 0 0 0 0 1 0 Restore A
0 0 0 0 0 0 0 1 0 Clear q_0
Remainder
Quotient

3.7



3.8 $c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0$

3.9 (a) The carry out of each CLA is generated in just three gate delays after the inputs settle. The longest path through a CLA is five gate delays. The longest path through the 16-bit CLA/ripple adder is 14 (nine to generate c_{12}, plus five to generate s_{15}).

(b) s_0 is generated in just two gate delays.

(c) s_{12} is generated in 11 gate delays. It takes 3 gate delays to generate c_4, which is needed to generate c 3 gate delays later, which is needed to generate c_{12} 3 gate delays after that, for a total of 9 gate delays before c_{12} can be used in the leftmost CLA. The s_{12} output is generated 2 gate delays after that, for a total of 11 gate delays.

3.10

Booth algorithm:
Scan multiplier from right to left.
use –1 for a 0 to 1 transition;
use –1 for the rightmost 1;
use +1 for a 1 to 0 transition;
use 0 for no change.

0	1	0	0	1	1	Multiplicand
0	1	1	0	1	1	Multiplier
+1	0	–1	+1	0	–1	Booth coded multiplier

					0	1	0	0	1	1	Multiplicand
				×	+1	0	–1	+1	0	–1	Booth coded multiplier
	1	1	1	1	1	1	0	1	1	0	1	Negative multiplicand
	0	0	0	0	1	0	0	1	1	0	0	Multiplicand shifted left by 2
	1	1	1	0	1	1	0	1	0	0	0	Negative multiplicand shifted left by 3
+	0	1	0	0	1	1	0	0	0	0	0	Multiplicand shifted left by 5
	0	1	0	0	0	0	0	0	0	0	1	Product

3.11

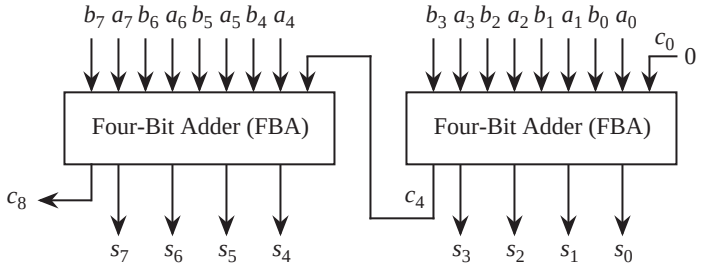
0	1	0	0	1	1	Multiplicand
0	1	1	0	1	1	Multiplier
+1	0	–1	+1	0	–1	Booth coded multiplier
+2		–1		–1		Bit-pair recoded multiplier

						0	1	0	0	1	1	Multiplicand
					×		+2		–1		–1	Bit-pair recoded multiplier
	1	1	1	1	1	1	1	0	1	1	0	1	$(-1 \times 19 \times 1)$
	1	1	1	1	1	0	1	1	0	1	0	0	$(-1 \times 19 \times 4)$
+	0	0	1	0	0	1	1	0	0	0	0	0	$(+2 \times 19 \times 16)$
	0	0	1	0	0	0	0	0	0	0	0	1	Product

3.12 The carry bit generated by the *ith* full adder is: $c_i = G_i + P_iG_{i-1} + ...+ P_iP_1G_0$. The G_i and P_i bits are computed in one gate delay. The c_i bit is computed in two additional gate delays. Once we have c_i, the sum outputs are computed in two more gate delays. There are 1 + 2 + 2 = 5 gate delays in any carry lookahead adder regardless of the word width, assuming arbitrary fan-in and fan-out.

3.13 Refer to Figure 3-21. The OR gate for each c_i has i inputs. The OR gate for c_{32} has 32 inputs. No other logic gate has more inputs.

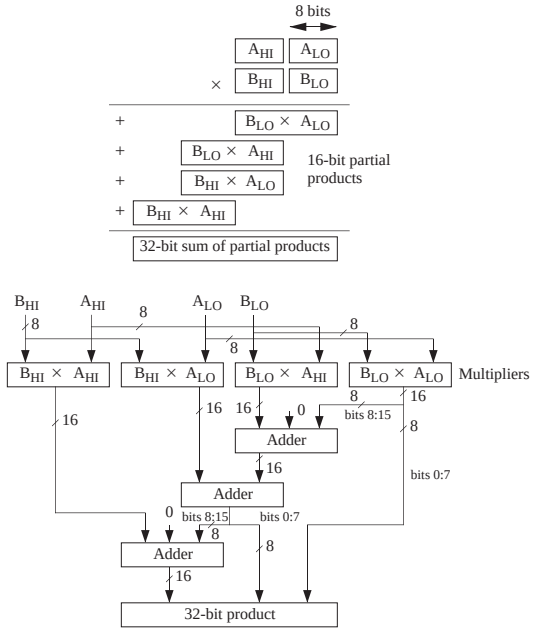
3.14 (a)



(b) Assume that a MUX introduces two gate delays as presented in Chapter 3. The number of gate delays for the carry lookahead approach is 8 (c_4 is generated in three gate delays, and s_7 is generated in five more gate delays). For the carry-select configuration, there are five gate delays for the FBAs, and two gate delays for the MUX, resulting in a total of 5 + 2 = 7 gate delays.

3.15 $3p$

3.16 There is more than one solution. Here is one: The basic idea is to treat each 16-bit operand as if it is made up of two 8-bit digits, and then perform the multiplication as we would normally do it by hand. So, $A_{0:15} = A_{8:15}A_{0:7} = A_{HI}A_{LO}$ and $B_{0:15} = B_{8:15}B_{0:7} = B_{HI}B_{LO}$, and the problem can then be represented as:



3.17 Two iterations.

3.18

```
  0110 0100 0001
+ 0010 0101 1001
-----------------
  1001 0000 0000
```

3.19

```
  0000 0001 0010 0011
+ 1001 1000 0010 0010
---------------------
  1001 1001 0100 0101
```

SOLUTIONS TO CHAPTER 4 PROBLEMS

4.1 24

4.2 Lowest: 0; Highest: $2^{18} - 1$

This is not a byte addressable architecture and so all addresses are in units of words, even though we

might think of words in terms of 4-byte units.

4.3 (a) Cartridge #1: 2^{16} bytes; cartridge#2: $2^{19} - 2^{17}$ bytes.

(b) [The following code is inserted where indicated in Problem 4.3.]

```
    orncc     %r4, %r0, %r4    ! Form 1's complement of old_y
    addcc     %r4, 1, %r4      ! Form 2's complement of old_y
    addcc     %r2, %r4, %r4    ! %r4 <- y - old_y
    be        loop
```

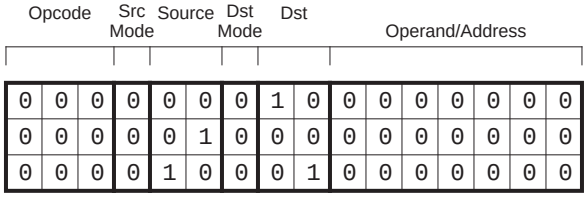
4.4

```
          .begin
          .org  2048
swap:     ld    [x], %r1
          ld    [y], %r2
          st    %r1, [y]
          st    %r2, [x]
          jmpl  %r15 + 4, %r0
x:        25
y:        50
          .end
```

4.5 (a) The code adds 10 array elements stored at `a` and 10 array elements stored at `b`, and places the result in the array that starts at `c`.

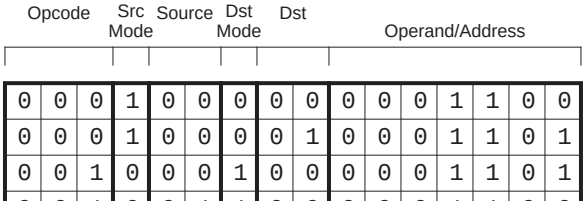
For this type of problem, study the logical flow starting from the first instruction. The first line loads `k`=40 into `%r1`. The next line subtracts 4 from that, leaving 36 in `%r1`, and the next line stores that back into `k`. If the result (+36 at this point) is negative, then `bneg` branches to `X` which returns to the calling procedure via `jmpl`. Otherwise, the code that follows `bneg` executes, which adds corresponding elements of arrays `a` and `b`, placing the results in array `c`.

4.6



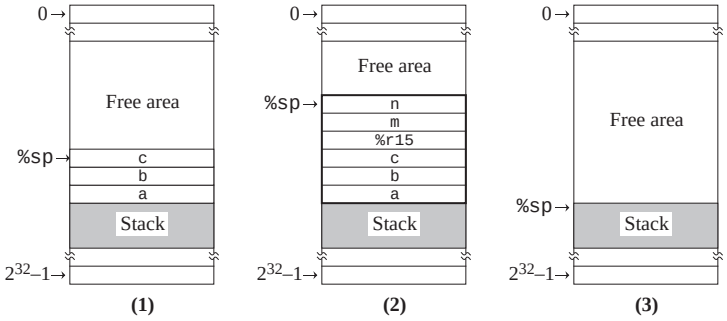
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0

(b) Note: There is more than one correct solution.



0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0	1	1	0	1
0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	1
0	0	1	0	0	1	1	0	0	0	0	0	1	1	0	0

4.7 The code adds 10 array elements stored at `a` and 10 array elements stored at `b`, and places the



result in the array that starts at `c`.

4.8 All instructions are 32 bits wide. 10 of those bits need to be used for the opcode and destination register, which leaves only 22 bits for the `imm22` field.

4.9 The convention used in this example uses a "hardwired" data link area that begins at location 3000. This is a variation to passing the address of the data link area in a register, which is done in the example shown in Figure 4-16.

4.10 The SPARC is big-endian, but the Pentium is little-endian. The file needs to be "byte-swapped" before using it on the other architecture (or equivalently, the program needs to know the format of the file and work with it as appropriate for the big/little-endian format.)

4.11

```
aload_0
invokespecial 3       // Same as: invokespecial <init>
return
```

4.12 Notice that the first line of the bytecode program in Figure 4-24 begins with the hexadecimal word `cafe` at address 0 and `babe` at address 2. The text describes the "magic number 0x`cafebabe`" used to identify a Java program. Since the most significant bits (`cafe`) of the 32-bit magic number are stored at the lower address, the machine must be big-endian. (Note that on a little-endian machine the number would display as `bebafeca`. Not nearly so interesting a word.)

4.13

```
main:
        mov     15,%r16
        mov     9,%r17
        st      %r16,[%sp+68]    ! 15 on stack
        st      %r17,[%sp+72]    ! 9 on stack
        call    addtwoints
        nop
ld      [%sp-4],%r9     ! result loaded from stack to %r9
!       printf("ans is %d \n", result);
!       printf expects value in %r9 and format string in %r8.
        sethi   %hi(.L17),%r8
        or      %r8,%lo(.L17),%r8
        call    printf
        nop
        jmp     %r31+8

        ...

addtwoints:
        ld      [%fp+68],%r16    ! first parameter in %r16
        ld      [%fp+72],%r17    ! second parameter in %10

        add     %r16,%r17,%r16
        st      %r16,[%fp-4]     ! result on stack
        nop
        jmp     %r31+8
```

4.14 a) 13 (instruction fetches) + 5 (stack pushes) + 4 (stack pops) + 3 (stack add) = 25 bytes.

b) [Placeholder for missing solution.]

4.15 It is doubtful that a bytecode program will ever run as fast as the equivalent program written in the native language. Even if the program is run using a just-in-time (JIT) compiler, it still will use stack-based operations, and will thus not be able to take advantage of the register-based operations of the native machine.

4.16

	3-address:	2-address	1-address
	SUB B, C, A	MOV B, A	LOAD B
	SUB D, E, Tmp	SUB C, A	SUB C
	MPY A, Tmp, A	MOV D, Tmp	STO Tmp
		SUB E, Tmp	LOAD D
		MPY Tmp, A	SUB E
			MPY Tmp
			STO A
Size:	7×3=21 bytes	5×5 = 25 bytes	3×7 = 21 bytes.
Traffic:	3×4+3×3 = 21 wds.	5×3 + 2×2 + 3×3 = 28 wds.	7×2 + 7 = 21 wds.

4.17

```
ld      [B], %r1
ld      [C], %r2
ld      [D], %r3
ld      [E], %r4
subcc   %r1, %r2, %r2
subcc   %r3, %r4, %r4
smul    %r2, %r4, %r4
st      %r4, [A]
```

Size: 8 32-bit words. Traffic: 8 + 4 +1 = 13 32-bit words. Note how the load-store, general register architecture leads to less memory traffic than either of the architectures in the exercise above.

SOLUTIONS TO CHAPTER 5 PROBLEMS

5.1 The symbol table is shown below. The basic approach is to create an entry in the table for each symbol that appears in the assembly language program. The symbols can appear in any order, and a simple way to collect up all of the symbols is to simply read the program from top to bottom, and from left to right within each line. The symbols will then be encountered in the order: `x`, `main`, `lab_4`, `k`, `foo`, `lab_5`, and `cons`. Of these labels, `x`, `main`, `lab_4`, `foo`, and `cons` are defined in the program. `k` and `lab_5` are not defined and are marked with a U. Excluded from the symbol table are mnemonics (like `addcc`), constants, pseudo-ops, and register names.

`x` has the value 4000 because `.equ` defines that. `main` is at location 2072, and so it has that value in the symbol table. `lab_4` is 8 bytes past main (because each instruction is exactly 4 bytes in size) and so `lab_4` is at location 2800, etc.

Symbol	Value
`x`	4000
`main`	2072
`lab_4`	2080
`k`	U
`foo`	2088
`lab_5`	U
`cons`	2104

5.2 Notice that the `rd` field for the `st` instruction in the last line is used for the source register.

```
10001000 10000001 00100100 00000000
11001010 00000011 10000000 00000000
10011100 10000011 10111111 11111111
11001010 00100000 00110000 00000000
```

5.3 [Placeholder for missing solution.]

5.4

```
3072: 10001100 10000000 10000000 00000100
3076: 00001010 10000000 00000000 00000011
3080: 10001010 10000000 01000000 00000011
```

```
3084: 10000001 11000011 11100000 00000100
3088: 10001010 10000000 01000000 00000011
3092: 00001010 10000000 00000000 00000011
3096: 10001010 10000001 01100000 00000001
3100: 10000001 11000011 11100000 00000100
3104: 10001010 10000001 01100000 00000001
3108: 00001111 00111111 11111111 11111111
3112: 10000000 10000001 11000000 00000111
3116: 10000001 11000011 11100000 00000100
3120: 00000000 00000000 00000000 00000000
3124: 00000000 00000000 00000000 00011001
3128: 11111111 11111111 11111111 11111111
3132: 11111111 11111111 11111111 11111111
3136: 00000000 00000000 00000000 00000000
3140: 00000000 00000000 00000000 00000000
```

5.5

```
b:  addcc      %r1, 1, %r1
    orcc       %r5, %r6, %r0
    be         a
    srl        %r6, 10, %r6
    ba         b
a:  jmpl       %r15 + 4, %r0
```

5.6

```
    ld         %r14, %r1
    addcc      %r14, 4, %r14    ! This line can be deleted
    addcc      %r14, -4, %r14   ! This line can be deleted
    st    %    r2, %r14
```

5.7

```
     .macro     return
     jmpl       %r15 + 4, %r0
     .endmacro
```

5.8

```
          .
          .
          .
     st   %r1, [x]
     st   %r2, [x+4]
     sethi .high22(x), %r5
     addcc %r5, .low10(x), %r5
     call  add_2
     ld    [x+8], %r3
          .
          .
          .
x:   .dwb  3
```

5.9

```
          .begin
          .org  2048
add_128:  ld    [x+8], %r1       ! Load bits 32-63 of x
          ld    [x+12], %r2      ! Load bits 0 - 31 of x
          ld    [y+8], %r3       ! Load bits 32 - 63 of y
          ld    [y + 12], %r4    ! Load bits 0 -31 of y
          call  add_64           ! Add lower 64 bits
          st    %r5, [z + 8]     ! Store bits 32 - 63 of result
          st    %r6, [z + 12]    ! Store bits 0 - 31 of result
```

```
          bcs   lo_64_carry
          addcc %r0, %r0, %r8    ! Clear carry
          ba    hi_words
lo_64_carry: addcc %r0, 1, %r8 ! Set carry
hi_words: ld    [x], %r1         ! Load bits 96 - 127 of x
          ld    [x + 4], %r2     ! Load bits 64-95 of x
          ld    [y], %r3         ! Load bits 96 - 127 of y
          ld    [y + 4], %r4     ! Load bits 64 - 95 of y
          call  add_64           ! Add upper 64 bits
          bcs   set_carry
          addcc %r6, %r8, %r6    ! Add in low carry
          st    %r5, [z]         ! Store bits 96 - 127 of result
          st    %r6, [z + 4]     ! Store bits 64 - 95 of result
          jmpl  %r15 + 4, %r0    ! Return
set_carry:addcc%r6, %r8, %r6   ! Add in low carry
          st    %r5, [z]         ! Store bits 96 - 127 of result
          st    %r6, [z + 4]     ! Store bits 64 - 95 of result
          sethi #3FFFFF, %r8
          addcc %r8, %r8, %r0    ! Restore carry bit
          jmpl  %r15 + 4, %r0    ! Return
x:        .dwb  4
y:        .dwb  4
z:        .dwb  4
          .end
```

5.10 Note: In the code below, `arg2` must be a register (it cannot be an immediate).

```
    .macro     subcc arg1, arg2, arg3
    orncc      arg2, %r0, arg2
    addcc      arg2, 1, arg2
    addcc      arg1, arg2, arg3
```

```
.endmacro
```

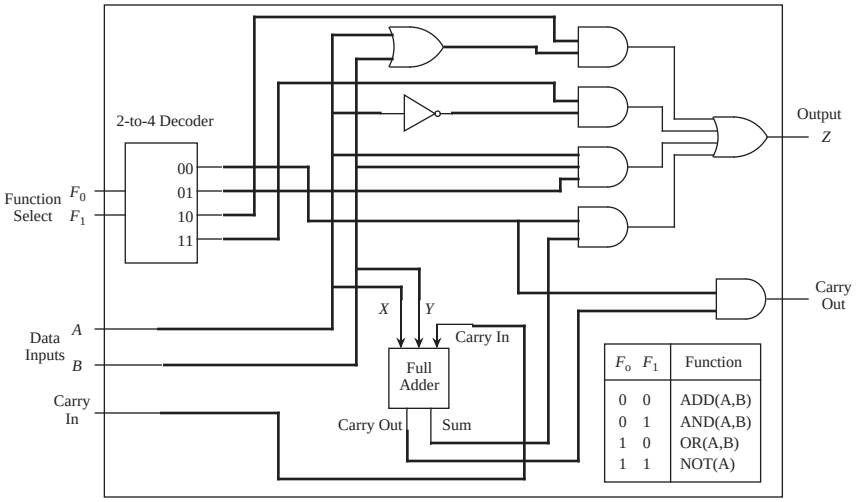
Note that this coding has a side effect of complementing `arg2`.

5.11 All macro expansion happens at assembly time.

5.12 The approach allows an arbitrary register to be used as a stack, rather than just %r14. The danger is that an unwitting programmer might try to invoke the macro with a statement such as `push X, Y`. That is, instantiating a stack at memory location Y. The pitfall is that this will result in an attempt to define the assembly language statement `addcc Y, -4, Y`, which is illegal in ARC assembly language.

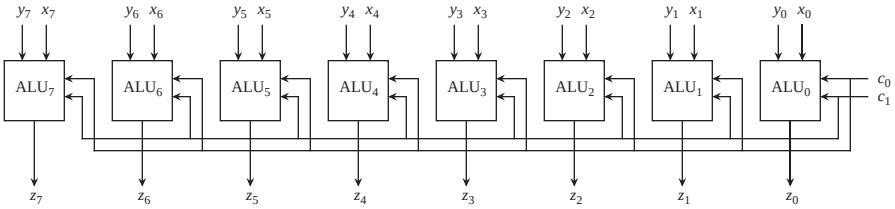
SOLUTIONS TO CHAPTER 6 PROBLEMS

6.1

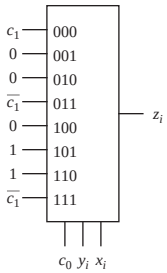


6.2 There is more than one solution, especially with respect to the choice of labels at the MUX

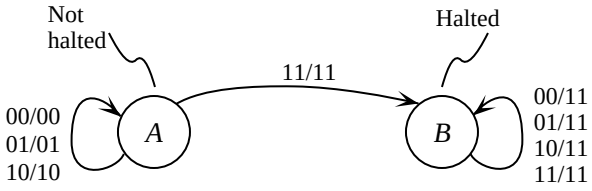
inputs. Here is one solution:



c_1	c_0	y_i	x_i	z_i
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0



6.3



6.4

Register `%r0` cannot be changed, and so there is no need for it to have a write enable line.

6.5

(a)

Write Enables				A-bus enables				B-bus enables						
0	1	2	3	0	1	2	3	0	1	2	3	F_0	F_1	Time
0	1	0	0	0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	1	0	0	0	0	1	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	1	0	0	2

(b)

$$r_0 \oplus r_1 = r_0\bar{r}_1 + \bar{r}_0 r_1 = \overline{\overline{r_0\bar{r}_1} + \overline{\bar{r}_0 r_1}} = \overline{\overline{r_0\bar{r}_1}}\;\overline{\overline{\bar{r}_0 r_1}}$$

Write Enables 0	1	2	3	A-bus enables 0	1	2	3	B-bus enables 0	1	2	3	F_0	F_1	Time	
0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	Save r_0
1	0	0	0	0	1	0	0	0	0	0	0	1	1	1	Compute $\bar{r}_1$
1	0	0	0	1	0	0	0	0	0	1	0	0	1	2	Compute $r_0\bar{r}_1$
1	0	0	0	1	0	0	0	0	0	0	0	1	1	3	Compute $\overline{r_0\bar{r}_1}$
0	0	1	0	0	0	1	0	0	0	0	0	1	1	4	Compute $\bar{r}_0$
0	0	1	0	0	0	1	0	0	1	0	0	0	1	5	Compute $\bar{r}_0 r_1$
0	0	1	0	0	0	1	0	0	0	0	0	1	1	6	Compute $\overline{\bar{r}_0 r_1}$
1	0	0	0	1	0	0	0	0	0	1	0	0	1	7	Compute $\overline{r_0\bar{r}_1}\;\overline{\bar{r}_0 r_1}$
1	0	0	0	1	0	0	0	0	0	0	0	1	1	8	Compute $\overline{\overline{r_0\bar{r}_1}}\;\overline{\overline{\bar{r}_0 r_1}}$

6.6

	A	AMUX	B	BMUX	C	CMUX	DR	RW	ALU	COND	JUMP ADDR
60	000000	0	100001	0	100001	00	0	0	111	010	0000100000
61	000000	1	000000	0	000000	10	0	1	101	000	0000000000

6.7

```
R[temp0] ← SEXT13(R[ir]);
R[temp0] ← ADD(R[rs1],R[temp0]); GOTO 1793;
R[temp0] ← ADD(R[rs1],R[rs2]); IF IR[13] THEN GOTO 1810;
```

6.8

```
1280: R[15] <- AND(R[pc], R[pc]);       / Save %pc in %r15
1281: R[temp0] <- LSHIFT2(R[ir]);       / Shift disp30 left two bits
1282: R[pc] <- ADD(R[pc], R[temp0]);    / Jump to subroutine
      GOTO 0;
```

6.9 Either seven or eight microinstructions are executed, depending on the value of `IR[13]`:

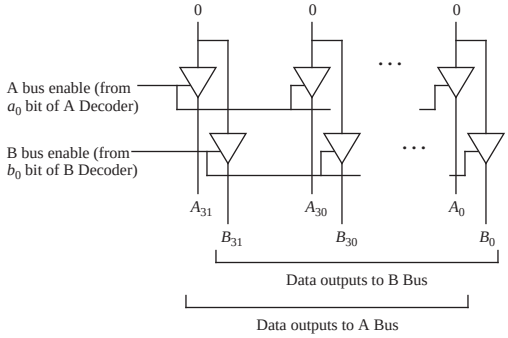
`IR[13] = 0`: (Eight microinstructions) 0, 1, 1584, 1585, 1586, 1587, 1603, 2047.

`IR[13] = 1`: (Seven microinstructions) 0, 1, 1584, 1586, 1587, 1603, 2047.

6.10 (a) (11 microinstructions): 0, 1, 1088, 2, 3, 4, 5, 6, 7, 8, 12.

(b) 0, 1, 2, 19

6.11



6.12

000000, or any bit pattern that is greater than 37_{10}.

6.13

There is more than one solution. Here is one:

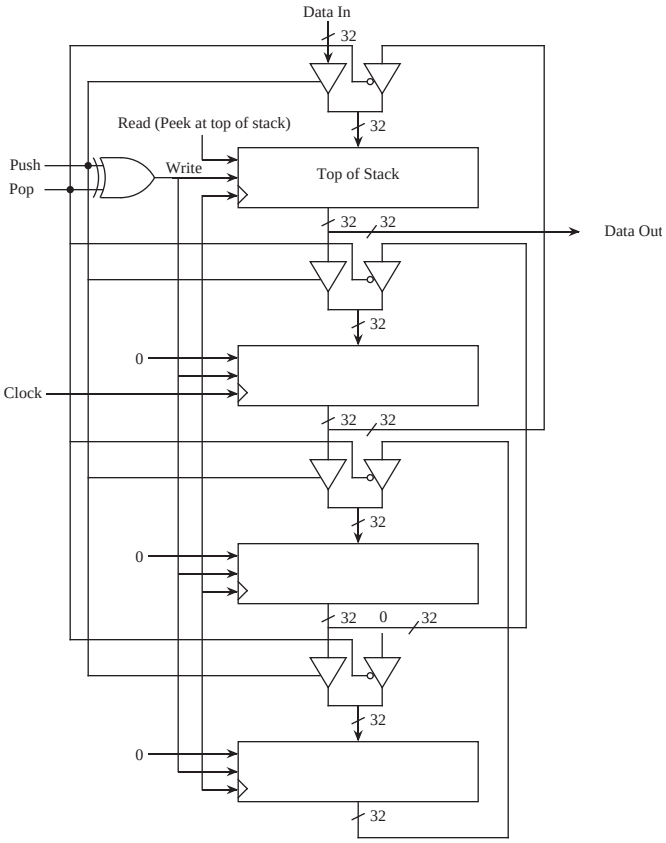
Address A B C D	V	W	X	S
0 0 0 0	1	0	0	0
0 0 0 1	0	0	1	1
0 0 1 0	0	1	0	0
0 0 1 1	0	0	1	1
0 1 0 0	0	1	0	0
0 1 0 1	0	0	1	1
0 1 1 0	0	1	0	0
0 1 1 1	0	0	1	1
1 0 0 0	0	0	1	1
1 0 0 1	0	0	1	1
1 0 1 0	0	0	1	1
1 0 1 1	0	0	1	1
1 1 0 0	0	0	1	1
1 1 0 1	0	0	1	1
1 1 1 0	0	0	1	1
1 1 1 1	0	0	1	1

6.14

```
1612: IF IR[13] THEN GOTO 1614;   / Is second source operand immediate?
1613: R[temp0] <- R[rs2];  / Get B operand in temp0
      GOTO 1615;
1614: R[temp1] <- SIMM13(R[ir]);  / Get A operand in temp1
      GOTO 21;
1615: R[temp1] <- R[rs1]; GOTO 21;
21: R[temp2] ← NOR(R[temp0], R[temp0]); / Get complement of B in temp2
22: R[temp3] ← NOR(R[temp1], R[temp1]); / Get complement of A in temp3
```

```
23: R[temp1] ← NOR(R[temp1], R[temp2]); / temp1 gets AND(A, B')
24: R[temp2] ← NOR(R[temp0], R[temp3]); / temp2 gets AND(A', B)
25: R[rd] ← ORCC(R[temp1],R[temp2]); GOTO 2047; / Extract rs2 operand
```

6.15



6.16

No.

6.17

	ALU	A-Bus	B-Bus	C-Bus	Cond	Jump Address	Next Address
0	00	0100	1000	001000		000 000 0000	00 00 00 0001
1	00	0000	0000	000011		000 000 1111	00 00 00 0010
2	01	0010	0100	100000		000 000 0000	00 00 00 0011
3	00	0000	0000	000001		000 001 0100	00 00 00 0000

6.18 No. After adding 1 to 2047, the 11-bit address wraps around to 0.

6.19 (a) 137 bits

(b) (2^{11} words × 137 bits) / (2^{11} words × 41 bits) = 334%

6.20

ALU LUT_1		ALU LUT_0	
z_i	*Carry Out*	z_i	*Carry Out*
0	0	1	0
0	0	1	0
1	0	0	1
1	0	0	1
1	0	d	d
1	0	d	d
0	1	d	d
0	1	d	d

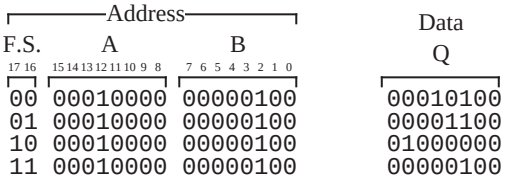
6.21 `temp0` should be multiplied by 4 (by shifting `temp0` to the left by two bits) before adding `temp0` to the PC in line 12.

SOLUTIONS TO CHAPTER 7 PROBLEMS

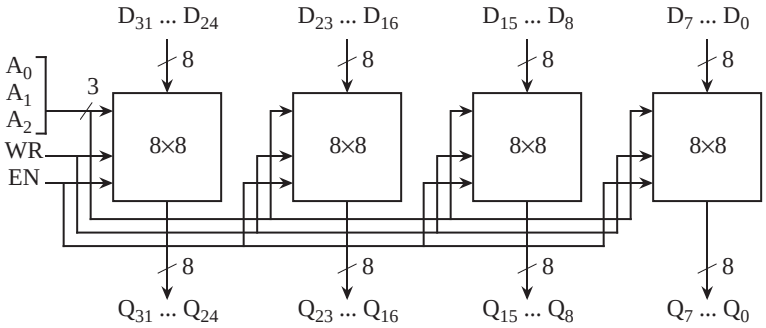
7.1

Input / P.S.	$X(A_0)$ 0	$X(A_0)$ 1
A_1 A_2		
A: 00	*B*/0	*B*/1
B: 01	*C*/1	*D*/1
C: 10	*A*/1	*B*/0
D: 11	*A*/0	*A*/0

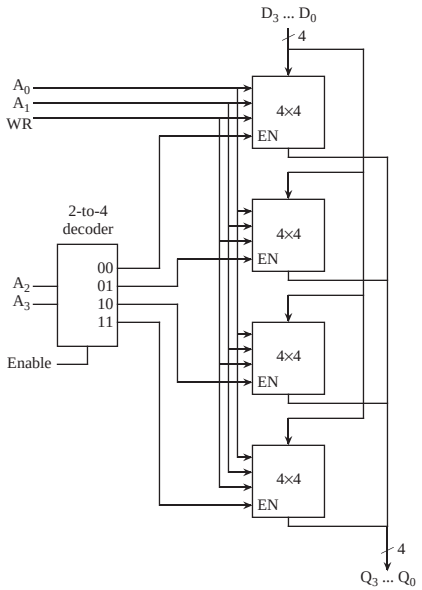
7.2


Address
F.S.
A
B
Data
Q
17 16
15 14 13 12 11 10 9 8
7 6 5 4 3 2 1 0
00 00010000 00000100 00010100
01 00010000 00000100 00001100
10 00010000 00000100 01000000
11 00010000 00000100 00000100

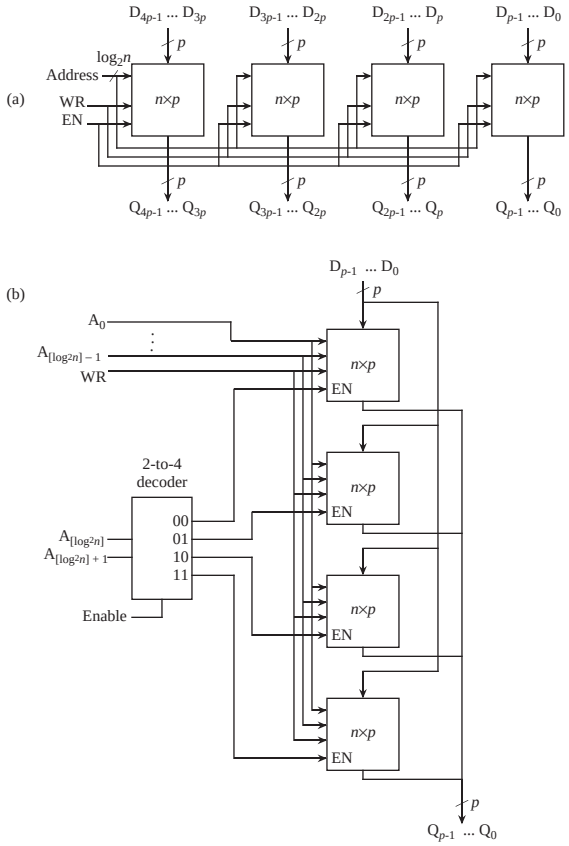
7.3


D31 ... D24
D23 ... D16
D15 ... D8
D7 ... D0
8
A0
A1
A2
3
WR
EN
8×8
Q31 ... Q24
Q23 ... Q16
Q15 ... Q8
Q7 ... Q0

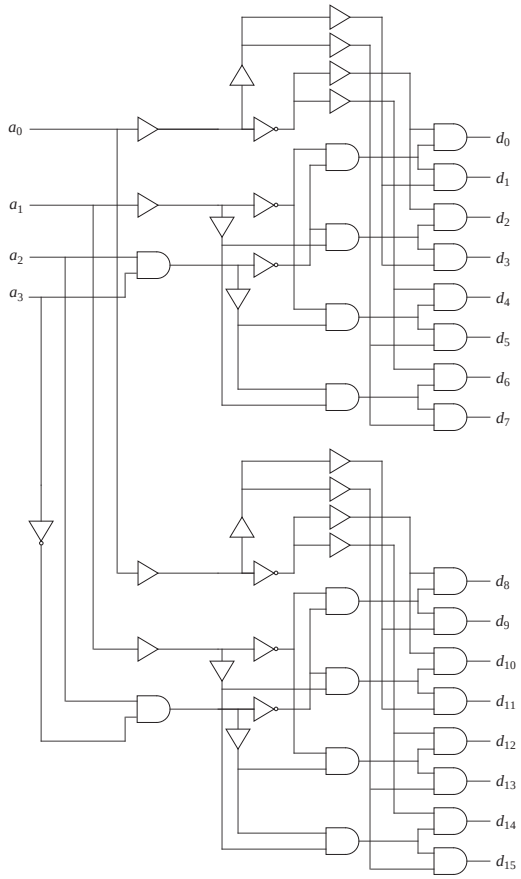
7.4


D3 ... D0
4
A0
A1
WR
4×4
EN
2-to-4
decoder
00
01
10
11
A2
A3
Enable
Q3 ... Q0

7.5


(a)
D_{4p-1} ... D_{3p}
D_{3p-1} ... D_{2p}
D_{2p-1} ... D_p
D_{p-1} ... D_0
p
$\log_2 n$
Address
WR
EN
n×p
Q_{4p-1} ... Q_{3p}
Q_{3p-1} ... Q_{2p}
Q_{2p-1} ... Q_p
Q_{p-1} ... Q_0
(b)
A_0
$A_{[\log_2 n]-1}$
WR
2-to-4
decoder
$A_{[\log_2 n]}$
$A_{[\log_2 n]+1}$
00
01
10
11
Enable
EN

7.6



7.7 (a)

Tag	Slot	Word
7	7	4

(b)

misses: 13 (on first loop iteration)

hits: 173 (first loop)

hits after first loop $9 \times 186 = 1674$

hits total = 173 + 1674 = 1847

accesses = hits + misses = 1860

hit ratio = 1847/1860 = 99.3%

(c) Avg. access time = [(1847)(10 ns) + (13)(210 ns)]/1860

= 11.4 ns

7.8 (a) The number of blocks in main memory is $2^{16}/8 = 2^{13}$. The tag field is 13 bits wide and the word field is three bits wide.

Tag	Word
13	3

(b) Addresses 20-45 lie in four different main memory blocks. When the program executes, there will be four misses and 94 hits. Hit ratio = 94/98 = 95.9%.

(c) The total access time for the program is 4(1040 ns) + 94(40 ns) = 7920 ns. Average access time = 7920/98 = 80.82 ns.

7.9 Associative = $(8 \times 32 + 29) \times 2^{14}$; Direct = $(8 \times 32 + 15) \times 2^{14}$

7.10 (a) The size of the cache is 2^{14} slots $\times$ 2^5 words/block = 2^{19} words, which is the minimum spacing needed to suffer a miss on every cache access.

(b) Every cache access causes a miss, and so the effective access time is 1000 ns.

7.11 (a) LRU

	0	2	4	5	2	4	3	11	2	10
Misses:	↑	↑	↑	↑			↑	↑		↑

(b) FIFO

	0	2	4	5	2	4	3	11	2	10
Misses:	↑	↑	↑	↑			↑	↑	↑	↑

7.12 If the page table is swapped out to disk, then the next memory address will cause a fault because the memory manager needs to know the physical location of that virtual address, and since the page table is on disk, the memory manager does not know where to get the page that contains the page table. An infinite sequence of page faults will occur. Using the LRU page replacement policy will prevent this from happening since the page containing the page table will always be in memory, but this will only work if the page table is small enough to fit into a single page (which it usually is not).

7.13 (a) 1024

(b) It is not in main memory.

(c) Page frame 2

7.14 (a) $(N - M) / N$

(b) $(M - F) / M$

(c) $[(N - M)T_1 + (M - F)T_2 + FT_3]/N$

7.15 Virtual addresses tend to be large (typically 32 bits). If virtual addresses are cached, then the tag fields will be correspondingly large as well as the hardware that looks at the tags. If physical addresses are cached, then every memory reference will go through the page table. This slows down the cache, but the tag fields are smaller, which simplifies the hardware.

If we cluster the virtual memory and cache memory into a single memory management unit (MMU), then we can cache physical addresses and simultaneously search the cache and the page table, using the lower order bits of the address (which are identical for physical and virtual addresses). If the page table search is successful, then that means the corresponding cache block (if we found a block) is the block we want. Thus, we can get the benefits of small size in caching physical addresses while not being forced to access main memory to look at the page table, because the page table is now in hardware. Stated more simply: this is the purpose of a translation lookaside buffer.

7.16 There are 2^{32} bytes / 2^{12} bytes/page = 2^{20} pages. There is a page table entry for each page, and so the size of the page table is $2^{20} \times 8$ bytes = 2^{23} bytes.

7.17 For the 2D case, each AND gate of the decoder needs a fan-in of 6, assuming the decoder has a form similar to Figure 7-4. There are 2^6 AND gates and 6 inverters, giving a total gate input count of $2^6 \times 6 + 6 = 390$ for the 2D case. For the 2-1/2D case, there are two decoders, each with 2^3 AND gates and 3 inverters, and a fan-in of 3 to the AND gates. The total gate input count is then $2 \times (2^3 \times 3 + 3) = 54$ for the 2-1/2D case.

7.18 $\log_4(2^{20}) = 10$

7.19 (a) 2^{20}

(b) 2^{11}

7.20 Words F1A0028 and DFA0502D

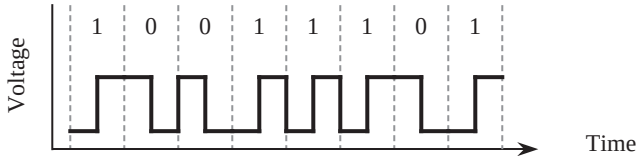
7.21 32

SOLUTIONS TO CHAPTER 8 PROBLEMS

8.1 The slowest bus along the path from the Audio device to the Pentium processors is the 16.7 MB/sec ISA bus. The minimum transfer time is thus 100 MB/(16.7 MB/sec) = 6 sec.

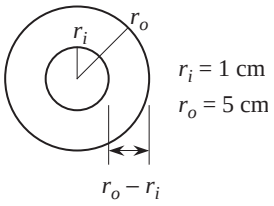
8.2 Otherwise, a pending interrupt would be serviced before the ISR has a chance to disable interrupts.

8.3



8.4 4.

8.5 (a)



Width of storage area = 5 cm – 1 cm = 4 cm.

Number of tracks in storage = 4 cm × 10 mm/cm × 1/.1 tracks/mm = 400 tracks.

The innermost track has the smallest storage capacity, so all tracks will store no more data than the innermost track. The number of bits stored on the innermost track is: 10,000 bits/cm × 2π × 1 cm = 62,832 bits.

The storage per surface is: 62,832 bits/track × 400 tracks/surface = 25.13×10^6 bits/surface.

The storage on the disk is: 2 surfaces/disk × 25.13×10^6 bits/surface = 50.26 Mbits/disk.

(b) 62,832 bits/track × 1 track/rev × 3600 rev/min × 1/60 min/s = 3.77 Mbits/sec

8.6 In the worst case, the head will have to move between the two extreme tracks, at which point an entire revolution must be made to line up the beginning of the sector with the position of the head.

The entire sector must then move under the head. The worst case access time for a sector is thus composed of three parts:

Seek time

15 ms/head movement × 127 head movements +

(1/3600 min/rev × 60,000 ms/min)(1 + 1/32) = 1922 ms

Rotational delay Sector read time

8.7 (a) The time to read a track is the same as the rotational delay, which is:

1/3600 min/rev × 1 rev/track × 60,000 ms/min = 16.67 ms.

(b) The time to read a track is 16.67ms (from 8.5a). The time to read a cylinder is 19 × 16.67 ms = 316.67 ms. The time to move the arm between cylinders is:

.25 mm × 1/7.5 s/m × 1000 ms/s × 1/1000 m/mm = 1/7.5 ms = .033 ms.

The storage per cylinder is 300/815 MB/cyl = .37 MB/cyl.

The time to transfer the buffer to the host is:

1/300 s/KB × .37 MB/cyl × 1024 KB/MB = 1.26 seconds/cylinder.

We are looking for the minimum time to transfer the entire disk to the host, and so we can assume that after the buffer is emptied, that the head is exactly positioned at the starting sector of the next cylinder. The entire transfer time is then (.317s/cyl + 1.26 s/cyl) × 815 cyl = 1285 s, or 21.4 min. Notice that the head movement time does not contribute to the transfer time because it overlaps with the 1.26 buffer transfer time.

8.8 A sector can be read into the buffer in .1 revolutions (rev). The disk must then continue for .9 rev in order to align the corresponding sector on the target surface with its head. The disk then continues through another .1 rev to write the sector, at which point the next sector to be read is lined up with its head, which is true regardless of which track the next sector is on. The time to transfer each sector is thus 1.1 rev. There are 10,000 sectors per surface, and so the time to copy one surface to another is:

10,000 sectors × 1.1 rev/sector × 1/3000 min/rev = 3.67 min.

8.9 The size of a record is:

2048 bytes × 1/6250 in/byte = .327 in.

There are x records and $x - 1$ inter-record gaps in 600 ft, and so we have the relation:

(.327 in)(x) + (.5 in) ($x - 1$) = 600 ft × 12 in/ft = 7200 in.

Solving for x, we have x = 8706 (whole) records, which translates to: 8706 records × 2048 bytes/record = 17.8 MB.

8.10 The number of bits written to the display each second is 1024 × 1024 × 60 = 62,914,560 bits/s. The maximum time allowed to write each pixel is then: 1/62,914,560 s/bit × 10^9 ns/s = 15.9 ns.

8.11 The LUT size = 2^8 words × 24 bits/word = 6144 bits, and the RAM size is 2^{19} words × 8 bits/word = 4,194,304. Without a LUT, the RAM size is 2^{19} words × 24 bits/word = 12,582,912 bits. The increase in RAM size is 8,388,608 bits, which is much larger than the size of the LUT.

8.12 (a) We no longer have random access to sectors, and must look at all intervening sectors before reaching the target sector.

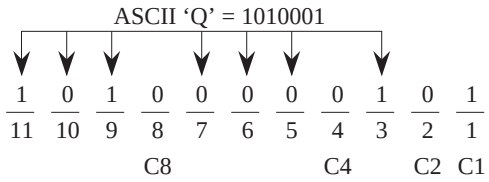
(b) Disk recovery would be easier if the MCB is badly damaged, because the sector lists are distributed throughout the disk. An extra block is needed at the beginning of each file for this, but now the MCB can have a fixed size.

8.13 The problem is that the data was written with the heads in a particular alignment, and that the head alignment was changed after the data was written. This means that the beginning of each track no longer corresponds to the relative positioning of each track prior to realignment. The use of a timing track will not fix the problem, unless a separate timing track is used for each surface (which is not the usual case).

SOLUTIONS TO CHAPTER 9 PROBLEMS

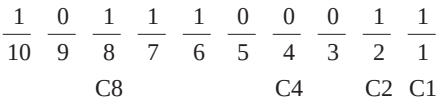
9.1 Hamming distance = 3.

9.2



9.3 (a) $k + r + 1 \leq 2^r$ for $k = 6$. Simplifying yields: $7 + r \leq 2^r$ for which $r = 4$ is the smallest value that satisfies the relation.

(b)



(c) 11

(d) 101111011001

9.4 $k + r + 1 \leq 2^r$ for $k = 1024$. Simplifying yields $1025 + r \leq 2^r$ for which $r = 11$ is the smallest value that satisfies the relation.

9.5

Code	*Character*
1 1 1 0 0 1 0 1	V
1 1 1 0 0 1 1 0	W
1 1 1 0 0 1 1 1	X
1 1 1 0 1 0 0 0	Y
1 1 1 0 1 0 0 1	Z
1 1 1 0 0 1 0 1	Checksum

9.6 (a) 4096

(b) 8

9.7 First, we look for errors in the check bit computations for the SEC code. If all of the check bit computations are correct, then there are no single bit errors nor any double bit errors. If any of the SEC check bit computations are wrong, then there is either a single bit error or a double bit error.

The DED parity bit creates even parity over the entire word when there are no errors. If the DED parity computation is in error, then there is an odd number of errors, which we can assume is a single bit

error for this problem. If the DED parity computation is even, and the SEC computation indicates there is an error, then there must be at least a double bit error.

9.8 CRC = 101. The entire frame to be transmitted is 101100110 101.

9.9 32 bits

9.10 Class B

9.11 $2^7 + 2^{14} + 2^{21}$

9.12 63 ns for bit-serial transfer, 32 ns for word-parallel. The word-parallel version requires 32 times as much hardware, but only halves the transmission time, as a result of the time-of-flight delay through the connection.

9.13 [Placeholder for solution.]

SOLUTIONS TO CHAPTER 10 PROBLEMS

10.1 $CPI_{AVG} = 1 + 5(.25)(.5) = 1.625 \times 10 = 16.25$ ns

Execution efficiency = 1/1.625 = 62%

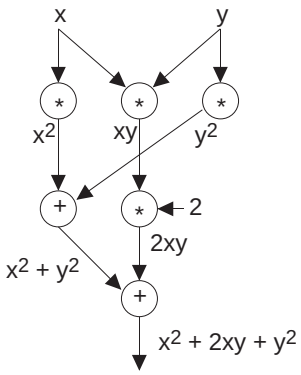
10.2

No, for the ARC architecture, we cannot use `%r0` immediately after the `st` because `st` needs two cycles to finish execution. If we did reuse `%r0` in the second line, then a `nop` (or some other instruction that does not produce a register conflict) would have to be inserted between the `st` and `sethi` lines. However, for the SPARC architecture, the pipeline stalls when a conflict is detected, so `nop`s are never actually needed for delayed loads in the SPARC. The `nop` instruction is needed for delayed branches in the SPARC, however.

10.3 [Placeholder for solution.]

10.4 [Placeholder for solution.]

10.5



10.6

$$S = \frac{1}{0.05 + \frac{1 - 0.05}{100}} = 16.8$$

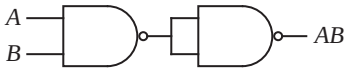
$$\frac{16.8}{100} = .168, \text{ or } 16.8\%$$

10.7 n=p=6: complexity = 306. n=p=2: complexity = 180. n=p=4: complexity = 231. n=p=3: complexity = 200.

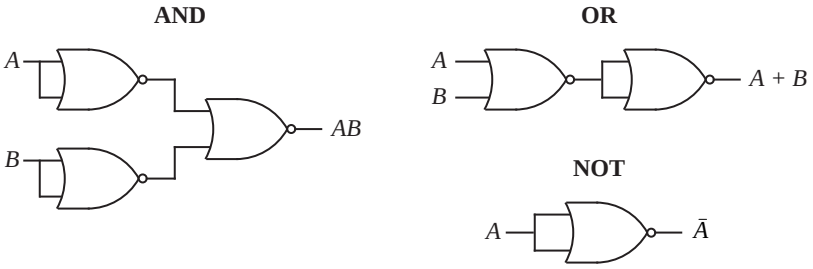
10.8 15

SOLUTIONS TO APPENDIX A PROBLEMS

A.1



A.2



A.3

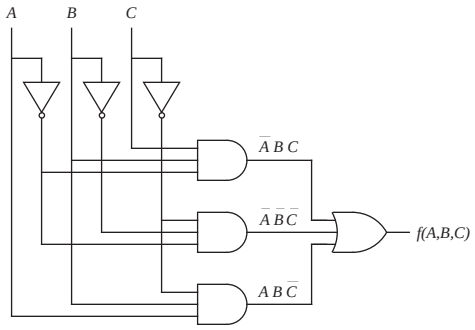
A	B	C	F	G
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	0

A.4

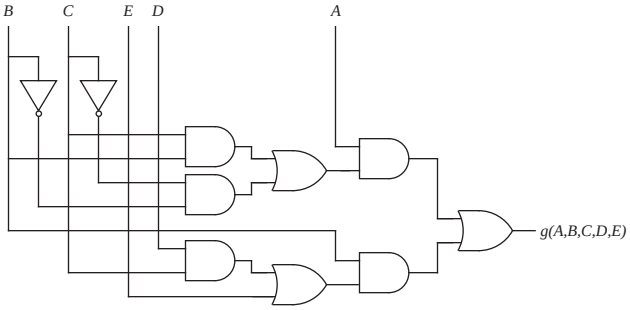
A	B	C	XOR
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A.5 18.

A.6



A.7



A.8

Not equivalent:

A	B	C	F	G
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

Alternatively, we can also disprove equivalence algebraically:

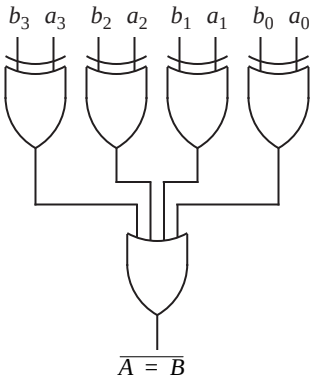
$$\begin{aligned} g(A, B, C) &= (A \oplus C)B \\ &= (A\overline{C} + \overline{A}C)B \\ &= AB\overline{C} + \overline{A}BC \\ &\neq f(A, B, C) \end{aligned}$$

A.9

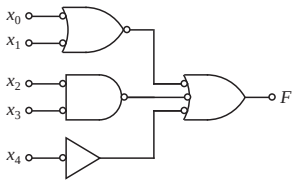
A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$F(A,B,C) = \overline{A}\overline{B}C$$

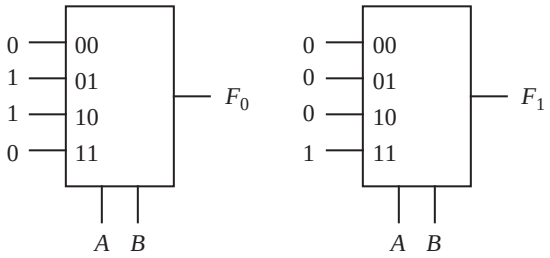
A.10



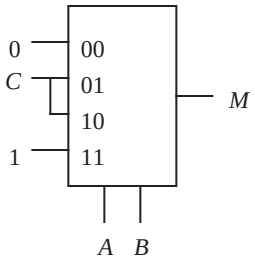
A.11



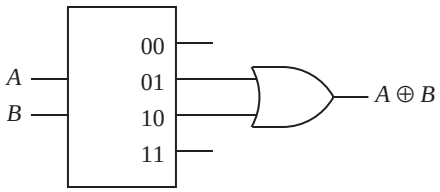
A.12



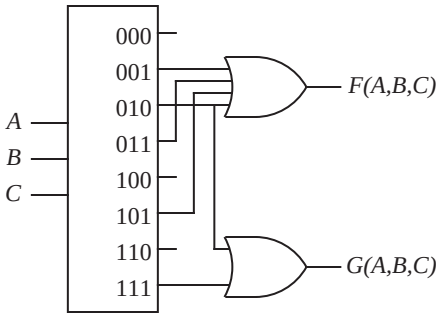
A.13



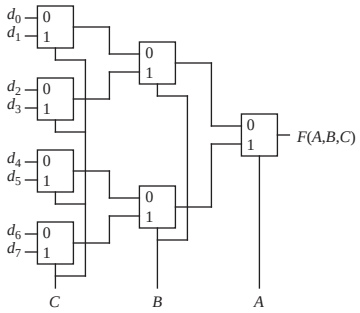
A.14



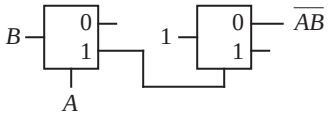
A.15



A.16

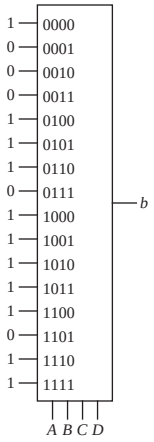


A.17

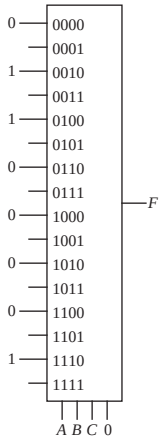


A.18

A	B	C	D	b
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1



A.19 There are a few solutions. Here is one:



A.20

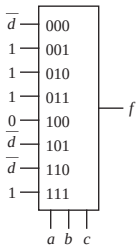
A	*B*	*C*	*D*	*X*	*Y*
0	0	0	0	d	d
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	d	d
0	1	0	0	1	0
0	1	0	1	d	d
0	1	1	0	d	d
0	1	1	1	d	d
1	0	0	0	1	1
1	0	0	1	d	d
1	0	1	0	d	d
1	0	1	1	d	d
1	1	0	0	d	d
1	1	0	1	d	d
1	1	1	0	d	d
1	1	1	1	d	d

A.21

a	b	c	u	v	w	x	y	z
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	1

A.22

a	b	c	d	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1



A.23

A	B	Z
0	0	1
0	1	0
0	2	0
1	0	2
1	1	1
1	2	0
2	0	2
2	1	2
2	2	1

a_1	a_0	b_1	b_0	z_1	z_0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	d	d
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	d	d
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	d	d
1	1	0	0	d	d
1	1	0	1	d	d
1	1	1	0	d	d
1	1	1	1	d	d

A.24

a	b	c	$ab + \bar{a}c + bc$	$ab + \bar{a}c$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

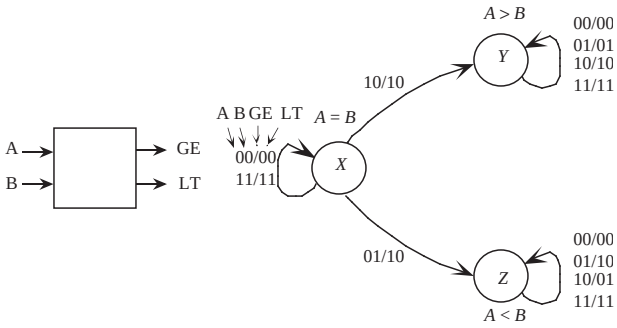
A.25 [Note: This solution is for the wrong problem. This is a proof of DeMorgan's theorem, not the absorption theorem.]

$$\bar{y} + y = 1$$
$$x + \bar{y} + y = 1$$
$$1x + \bar{y} + y = 1$$
$$(x + \bar{x})(x + \bar{y}) + y = 1$$
$$x + \bar{x}\bar{y} + y = 1$$
$$\bar{x}\bar{y} + (x + y) = 1$$
$$\overline{(x + y)} + (x + y) = \bar{x}\bar{y} + (x + y)$$
$$\overline{(x + y)} = \bar{x}\bar{y}$$

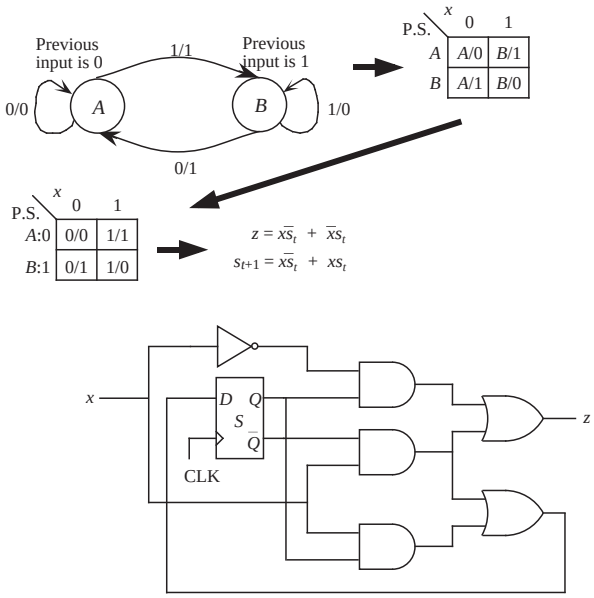
A.26 No, an S-R flip-flop cannot be constructed this way. There is no way to force an output high or low based only on S or R. While the combination of 11 on the inputs provides a quiescent state, the result of applying 00 is undefined and 10 and 01 are unstable.

A.27 [Placeholder for missing solution.]

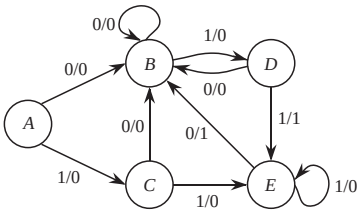
A.28


A
B
GE
LT
A B GE LT
A = B
00/00
11/11
X
10/10
Y
A > B
00/00
01/01
10/10
11/11
01/10
Z
A < B
00/00
01/10
10/01
11/11

A.29


Previous input is 0
Previous input is 1
1/1
0/0
1/0
0/1
A
B
P.S.
x
0
1
A
A/0
B/1
B
A/1
B/0
P.S.
x
0
1
A:0
0/0
1/1
B:1
0/1
1/0
$z = x\bar{s}_t + \bar{x}s_t$
$s_{t+1} = x\bar{s}_t + xs_t$
x
D
Q
S
Q̄
CLK
z

A.30

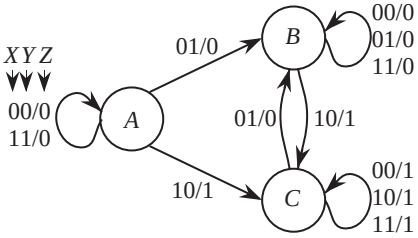


Meanings of States

A	No inputs seen yet
B	Seen '0'
C	Seen '1'
D	Seen '01'
E	Seen '11'

Input / P.S.	*X* = 0	*X* = 1
A	*B*/0	*C*/0
B	*B*/0	*D*/0
C	*B*/0	*E*/0
D	*B*/0	*E*/1
E	*B*/1	*E*/0

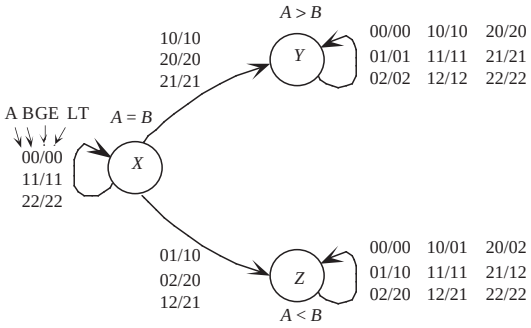
A.31



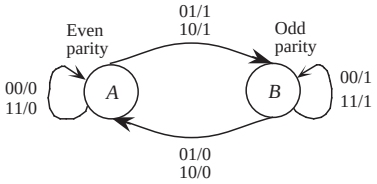
Meanings of States

A	$X = Y$
B	$X < Y$
C	$X > Y$

A.32



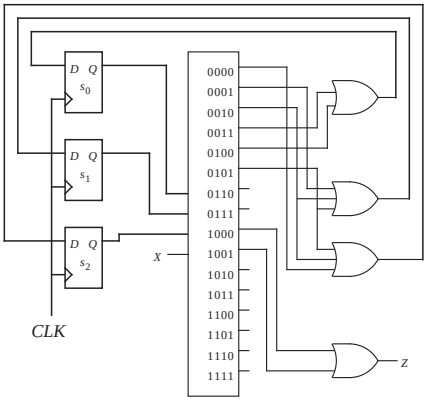
A.33



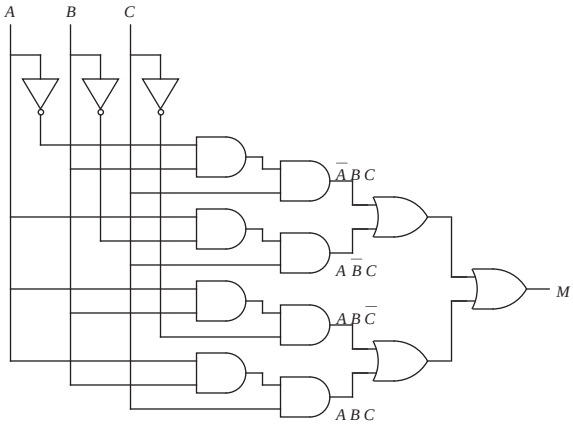
A.34

Input / P.S.	X = 0	X = 1
A	B/0	C/0
B	D/0	E/0
C	E/0	D/0
D	A/0	A/0
E	A/1	A/1

Input / P.S.	X = 0	X = 1
$s_0\ s_1\ s_2$		
A:000	001/0	010/0
B:001	011/0	100/0
C:010	100/0	011/0
D:011	000/0	000/0
E:100	000/1	000/1



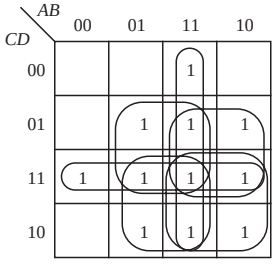
A.35 There are a number of solutions. Here is one:



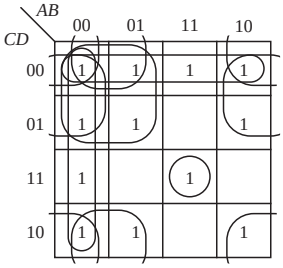
A.36 $\lceil log_3 N \rceil$ delays using a balanced tree.

SOLUTIONS TO APPENDIX B PROBLEMS

B.1

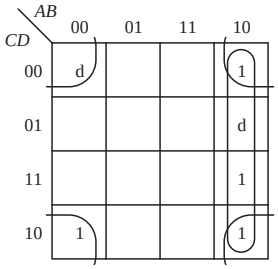


$f(A,B,C,D) = AB + CD + BD + AD + BC + AC$

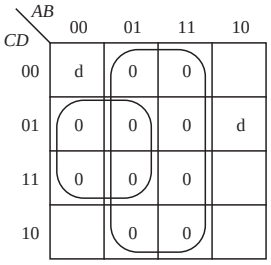


$g(A,B,C,D) = ABCD + \bar{B}\bar{D} + \bar{C}\bar{D} + \bar{A}\bar{B} + \bar{A}\bar{D} + \bar{A}\bar{C} + \bar{B}\bar{C}$

B.2



SOP Form: $f(A,B,C,D) = A\bar{B} + \bar{B}\bar{D}$



POS Form: $\overline{f(A,B,C,D)} = \bar{A}D + B$

$f(A,B,C,D) = \overline{\bar{A}D + B}$

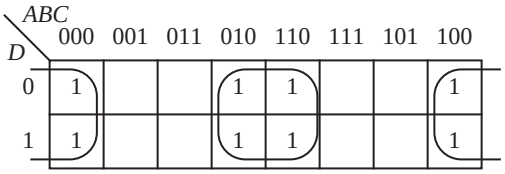
$= (\overline{\bar{A}D})(\bar{B})$

$= (A + \bar{D})(\bar{B})$

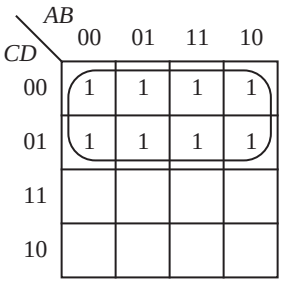
B.3

No. The don't cares are used during the design process. Once the design is fixed, the don't cares assume values of either 0's or 1's.

B.4

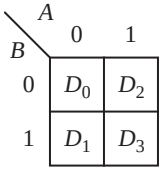


$F(A,B,C,D) = \bar{B}\bar{C} + B\bar{C}$



$F(A,B,C,D) = \bar{C}$

B.5



$F = \bar{A}\bar{B}D_0 + \bar{A}BD_1 + A\bar{B}D_2 + ABD_3$

B.6

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	*d*
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	*d*
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Initial setup

A	B	C	D	
0	0	1	0	√
0	0	1	1	√
0	1	0	1	√
0	1	1	0	√
1	0	1	0	√
0	1	1	1	√
1	1	0	1	√
1	1	1	1	√

After first reduction

A	B	C	D	
0	0	1	_	√
0	_	1	0	√
_	0	1	0	*
0	_	1	1	√
0	1	_	1	√
_	1	0	1	√
0	1	1	_	√
_	1	1	1	√
1	1	_	1	√

After second reduction

A	B	C	D	
0	_	1	_	*
_	1	_	1	*

Prime Implicants	Minterms					
	0011	0101	0111	1010	1101	1111
*_010				√		
*0_1_	√		√			
*_1_1		√	√		√	√

All prime implicants are essential, so we do not construct a reduced table of choice.

$$f(A, B, C, D) = \overline{B}C\overline{D} + \overline{A}C + BD$$

B.7

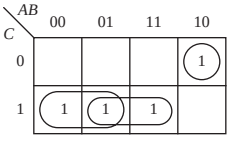
Function	Prime Implicant
F_0	0_11, 111_, _1_1
F_1	1_10, _1_1, 111_
F_2	0_00, 111_, 1_10
$F_{0,1}$	111_, _1_1
$F_{0,2}$	111_
$F_{1,2}$	111_, 1_10
$F_{0,1,2}$	111_

Reduce by eliminating prime implicants covered by higher order functions

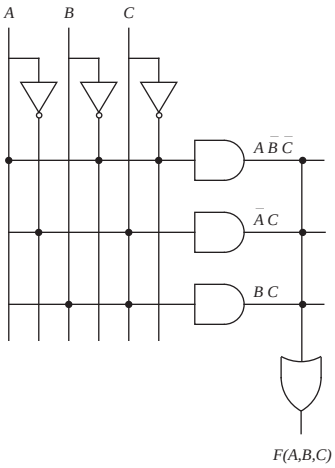
Function	Prime Implicant
F_0	0_11
F_1	none
F_2	0_00
$F_{0,1}$	_1_1
$F_{0,2}$	none
$F_{1,2}$	1_10
$F_{0,1,2}$	111_

Prime Implicants \ Min-terms	$F_0(A,B,C,D)$						$F_1(A,B,C,D)$						$F_2(A,B,C,D)$					
	m_3	m_5	m_7	m_{13}	m_{14}	m_{15}	m_5	m_7	m_{10}	m_{13}	m_{14}	m_{15}	m_0	m_4	m_{10}	m_{14}	m_{15}
* F_0 0_11	√		√															
* F_2 0_00													√	√				
* $F_{0,1}$ _1_1		√	√	√		√	√	√		√		√						
* $F_{1,2}$ 1_10									√		√				√	√		
* $F_{0,1,2}$ 111_					√	√					√	√				√	√	

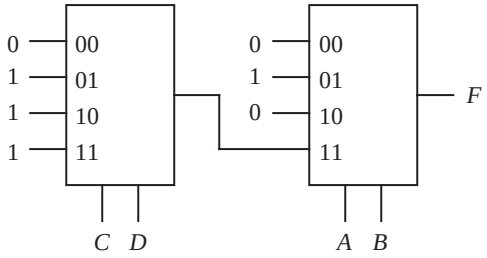
B.8



$F(A, B, C) = A\bar{B}\bar{C} + \bar{A}C + BC$



B.9



$F(A,B,C,D) = AB(CD + \bar{C}D + C\bar{D}) + \bar{A}B$

B.10

$P_0 = (ABCDEFG)$

$P_1 = (AB)(CD)(EF)(G)$

$P_2 = (AB)(CD)(E)(F)(G)$

$P_3 = (AB)(CD)(E)(F)(G)\ \surd$

$A'\quad B'\quad C'\ D'\ E'$

Present state \ Current input	X = 0	X = 1
A'	$B'/0$	$E'/0$
B'	$A'/0$	$B'/1$
C'	$A'/1$	$C'/0$
D'	$B'/1$	$D'/0$
E'	$C'/1$	$E'/1$

B.11

$P_0 = (ABCDE)$

$P_1 = (A)(BD)(CE)$

$P_2 = (A)(BD)(CE)$ √

A' B' C'

Present state \ Current input	XY 00	01	10	11
A'	$A'/0$	$B'/0$	$C'/0$	$B'/0$
B'	$A'/0$	$B'/1$	$B'/0$	$B'/1$
C'	$C'/1$	$B'/0$	$B'/0$	$C'/1$

B.12

$P_0 = (ABCDEFG)$

$P_1 = (AEFG)(BCD)$

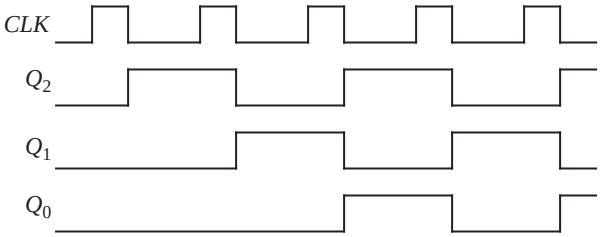
$P_2 = (AFG)(E)(BCD)$

$P_3 = (AFG)(E)(BCD)$ √

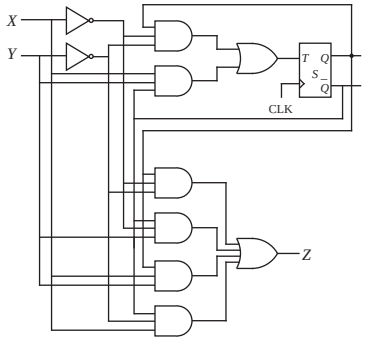
A' B' C'

Present state \ Current input	X 0	1	2
A'	$C'/0$	$B'/2$	$A'/1$
B'	$A'/0$	$B'/2$	$C'/1$
C'	$C'/2$	$A'/1$	$C'/0$

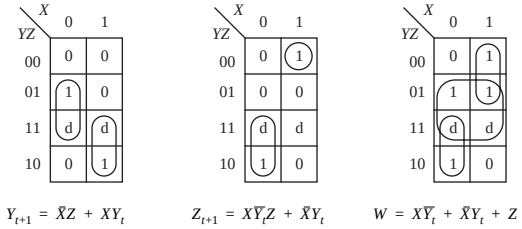
B.13

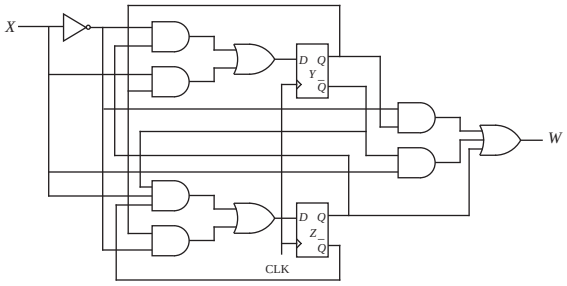


B.14

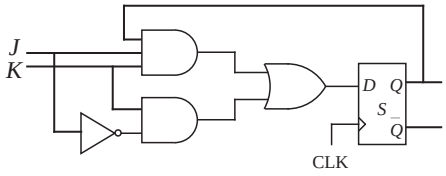


B.15


X
YZ
0 1
00 0 0
01 1 0
11 d d
10 0 1
$Y_{t+1} = \bar{X}Z + XY_t$
X
YZ
0 1
00 0 1
01 0 0
11 d d
10 1 0
$Z_{t+1} = X\bar{Y}_tZ + \bar{X}Y_t$
X
YZ
0 1
00 0 1
01 1 1
11 d d
10 1 0
$W = X\bar{Y}_t + \bar{X}Y_t + Z$


X
D Q
Y
Q
D Q
Z
Q
CLK
W

B.16


J
K
D Q
S
Q
CLK