COMPUTER ARCHITECTURE AND ORGANIZATION: An Integrated Approach

PRACTICE PROBLEMS

(Solutions are at the end)

FEBRUARY 2007

CHAPTER 1 PROBLEMS

(1-1) Place the computing technologies in the proper chronological order corresponding to when they were first introduced, from earliest to most recent:

- integrated circuits
- mechanical - machine powered
- mechanical - manually powered
- transistors
- vacuum tubes

CHAPTER 2 PROBLEMS

(2-1) The binary representation of the hexadecimal number 3B7F is (choose one):

(A) 0100 1001 1110 1101 (B) 0011 1011 0111 1111 (C) 0010 0100 0000 1010

(D) 0110 0011 1011 1100 (E) 1101 1100 1011 0101

(2-2) Convert the following numbers as indicated.

(a) $(110101)_2$ to unsigned base 10.

(b) $(-29)_{10}$ to two's complement (use 8 bits in the result).

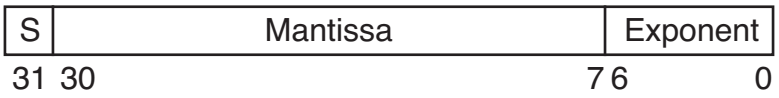
(c) $(61543)_8$ to unsigned base 16 (use four base 16 digits in the result).

(d) $(37)_{10}$ to unsigned base 3 (use four base 3 digits in the result).

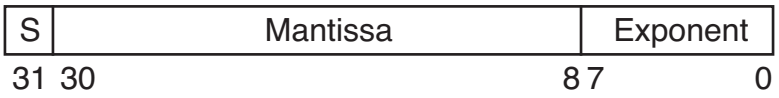
(2-3) A computer with a 32-bit word size uses two's complement to represent numbers. The range of integers that can be represented by this computer is:

(A) -2^{32} to 2^{32} (B) -2^{31} to 2^{32} (C) -2^{31} to $2^{31} - 1$ (D) -2^{32} to 2^3 (E) $-2^{32} - 1$ to 2^{32}

(2-4) Computer A uses the following 32-bit floating-point representation of real numbers:



Computer B uses the following floating point representation scheme:



Which of the following statements is true with regard to Computer B's method of representing floating-point numbers over Computer A's method?

(A) both the range and precision are increased

(B) the range is increased but the precision is decreased

(C) the range is decreased but the precision is increased

(D) both the range and precision are decreased

(E) both the range and precision remain the same

(2-5) (ref: Stallings, 1999) A normalized floating point representation has an exponent e with a representation that lies in the range $0 \le e \le X$, in excess q, with a base b and a p digit fraction. Note the emphasis on *representation*, as opposed to *value*.

(a) What are the largest and smallest positive values that can be represented?
(b) What are the largest and smallest gaps?
(c) What is the number of representable numbers?

(2-6) Express -1/32 in the IEEE 754 single precision format.

(2-7) For parts (a) through (d), use a floating point representation with a sign bit in the leftmost position, followed by a three-bit excess 4 exponent, followed by a normalized six-bit fraction in base 4. Zero is represented by the bit pattern 0 000 000000.

(a) What decimal number is represented by the bit pattern: 1 100 010000?

(b) Show the bit pattern for the smallest non-zero positive representable number.

(c) Show the bit pattern for the largest positive representable number.

(d) There are a total of 10 bits in this floating point representation, so there are $2^{10} = 1024$ unique bit patterns. How many of these bit patterns are valid? (Remember: 0 = 0 000 000000).

(2-8) Represent $(107.875)_{10}$ in the IEEE-754 single precision floating point representation which has a sign bit, an eight-bit excess 127 exponent, and a normalized 23-bit significand in base 2 with a hidden 1 to the left of the radix point. Truncate the fraction if necessary by chopping bits as necessary. Show your work.

CHAPTER 3 PROBLEMS

(3-1) Show the results of adding/subtracting the following pairs of six-bit (*i.e.* one sign bit and five data bits) two's complement numbers **and** indicate whether or not overflow/underflow occurs for each case:

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

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

(3-2) (From Hamacher *et al*, 1990) Perform A×B and A/B on unsigned A=10101 and B=00101 manually. (Just use pen and paper; do not show the serial multiplication or division menthods.)

(3-3) Show how the unsigned serial multiplication method would compute M × Q where M = 10110 and Q = 01101. M and Q are unsigned numbers. For each step, describe in words what is happening (shift left, shift right, add/subtract M or Q into product, set a bit, *etc*.), and show the product (or partial product) for that step. (Note: Q is the multiplier and M is the multiplicand.)

```
Step #              Product                    Action

  0         C=0   M=10110    Q=01101         Initial values

            C        A           Q
```

(3-4) Use the Booth algorithm (not bit pair recoding) to multiply 0110 by 0110.

(3-5) Use the modified Booth algorithm (that is, use bit pair recoding) to multiply signed numbers 010101 (multiplicand) by 110110 (multiplier).

(3-6) Use the Booth and bit-pair recoding techniques to multiply $(-10 \times -10 = 100)_{10}$.

(3-7) Boolean expressions are shown below for the difference $diff_i = (x_i - y_i)$ and borrow b_{i+1} outputs of a full subtractor. The subscripts denote the relative position of a full subtractor in a ripple-borrow subtractor:

$$diff_i = \bar{x}_i\bar{y}_i b_i + \bar{x}_i y_i \bar{b}_i + x_i\bar{y}_i\bar{b}_i + x_i y_i b_i$$
$$b_{i+1} = \bar{x}_i b_i + \bar{x}_i y_i + y_i b_i$$

We can factor the second equation and obtain:

$$b_{i+1} = \bar{x}_i y_i + b_i(\bar{x}_i + y_i)$$

which can be rewritten as:

$$b_{i+1} = G_i + P_i b_i$$

where: $G_i = \bar{x}_i y_i$ and $P_i = \bar{x}_i + y_i$.

The G_i and P_i terms are referred to as **generate** and **propagate** functions, respectively, for the effect they have on the borrow. When $G_i = 1$, a borrow is generated at stage i. When $P_i = 1$, then a borrow is propagated through stage i if either x_i is 0 or y_i is 1. Create Boolean equations for b_0, b_1, b_2, and b_3 for a borrow-lookahead subtractor in terms of P_i and G_i. Hint: Assume $b_0 = 0$.

CHAPTER 4 PROBLEMS

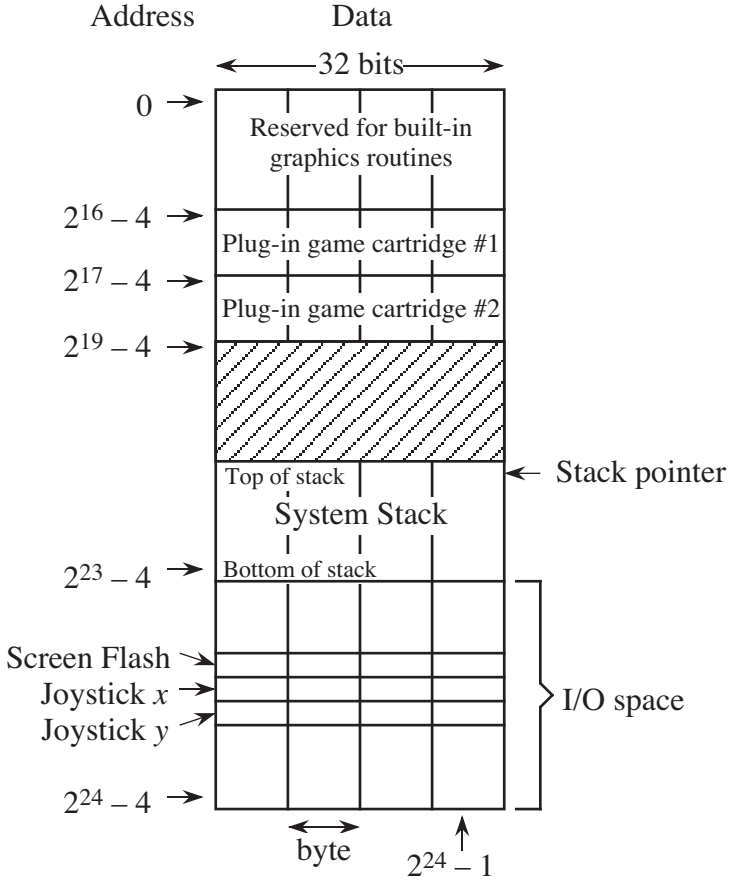
(4-1) Which of the following operations does function `baz` in the ARC program shown below carry out? The parameters *A* and *B* are passed to the function and returned from it are passed via the stack. *B* is closer to the stack pointer than *A*. Circle one of these operations:

(A) *min* (*A* , *B*) (B) *max*(*A* , *B*) (C) *addcc*(*A* , *B*) (D) *subcc*(*A* , *B*)

(E) (*A* == *B*) <--- in this case, the result is a boolean, indicated by a nonzero result.

```
baz:    ld      %sp, 4, %r1
        ld      %sp, 0, %r2
        orncc   %r2, %r0, %r3
        addcc   %r3, 1, %r3
        addcc   %r1, %r3, %r4
        bneg    foo
        st      %r2, 4, %sp
        ba      DONE
foo:    st      %r1, 4, %sp
        addcc   %r14, 4, %r14
DONE:   jmpl    %r15, 4, %r0
```

(4-2) The memory map for a video game that can accept two game cartridges is shown below. Each 32-bit word is composed of four 8-bit bytes in a big endian format, just like the ARC.



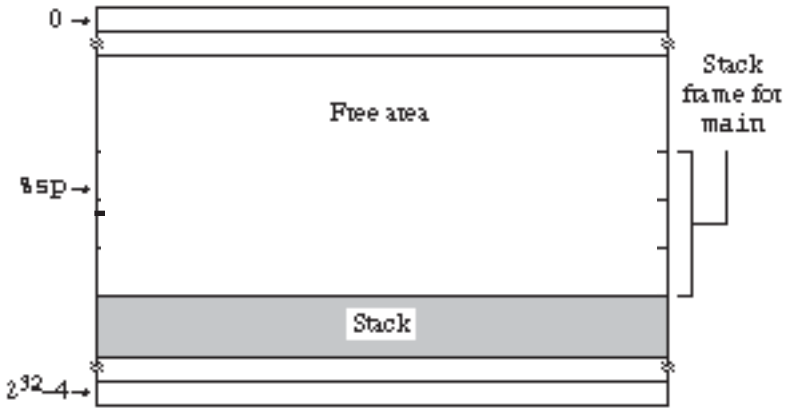
(a) How large can the stack grow, in bytes? (Leave your answer as an equation in powers of two, *e.g.* $8 + 2^{10}$.)

(b) When a joystick is moved, the horizontal (`joy_x`) and vertical (`joy_y`) positions of the joystick are updated in memory locations $(FFFFF0)_{16}$ and $(FFFFF4)_{16}$, respectively. When the number '1' is written to memory location $(FFFFEC)_{16}$ the screen flashes, and then location $(FFFFEC)_{16}$ is automatically cleared to zero by the hardware (the software does not have to clear it). Write an ARC program that flashes the screen every time the joystick moves. Use the skeleton program shown below. The ARC instruction set is summarized at the end of the exam.

```
                .begin
        ld      [joy_x], %r7    ! %r7 and %r8 now point to the
        ld      [joy_y], %r8    !   joystick x and y locations
        ld      [flash], %r9    ! %r9 points to the flash location
loop:   ld      %r7, %r1        ! Load current joystick position
        ld      %r8, %r2        !   in %r1=x and %r2=y
        ld      [old_x], %r3    ! Load old joystick position
        ld      [old_y], %r4    !   in %r3=x and %r4=y
        orncc   %r3, %r0, %r3   ! Form one's complement of old_x
        addcc   %r3, 1, %r3     ! Form two's complement of old_x
        addcc   %r1, %r3, %r3   ! %r3 <- joy_x - old_x
        be      x_not_moved     ! Branch if x did not change
        ba      moved           ! x changed, so no need to check y
x_not_moved:                    ! Your code starts here, about four lines.
        orncc   %r4, %r0, %r4   ! Form one's complement of old_y
        addcc   %r4, 1, %r4     ! Form two's complement of old_y
        addcc   %r2, %r4, %r4   ! %r4 <- joy_y - old_y
        be      loop            ! Repeat
! This portion of the code is entered only if joystick is moved.
! Flash screen; store new x,y values; repeat.

moved:
                                                    <— YOUR CODE GOES HERE

flash:  #FFFFEC                 ! Location of flash register
joy_x:  #FFFFF0                 ! Location of joystick x register
joy_y:  #FFFFF4                 ! Location of joystick y register
old_x:  0                       ! Previous x position
old_y:  0                       ! Previous y position
                .end
```

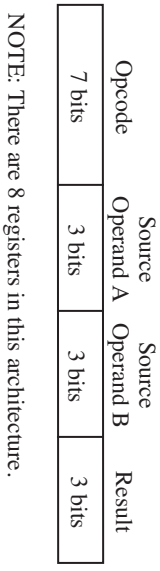
(4-3) With regard to stack frames, a main routine (or method, for Java) is treated just like any other routine or method. Show the stack frame for `main()` in the diagram below. Fill in the blanks for the `main()` stack frame with one-word descriptions of what is stored in each location. Note that there are many correct solutions (and many incorrect solutions too.)

```
main() /* Note: main() has no incoming parameters */
{
    int w, z;          /* Local variables */
  /***** Show stack frame for this location *****/
    w = func 1(1,2);   /* Call subroutine func 1 */
    z = func 2(10);    /* Call subroutine func 2 */
}                      /* End of main routine */
```



(4-4) The instruction set shown below should be new to you. The ZERO instruction has been removed in this version although its description is left in place so that you can understand its function. How can the remaining instructions be used to implement the ZERO function on register 5? No memory storage is to be used. Show the 16-bit object code instruction(s).

MNEMONIC	OPCODE	OPERAND FIELDS Operand A	Operand B	Result	DESCRIPTION
ZERO	0 0 0 0 0 0	—	—	X	Set contents of Result register to zero.
COPY	0 0 0 0 0 1	X	—	X	Copy Operand A register to Result register.
ADD	0 0 0 0 1 0	X	X	X	Add Operand A and Operand B registers, placing result in Result register.
SUB	0 0 0 0 1 1	X	X	X	Subtract Operand B register from Operand A register, placing result in Result register.
MULTIPLY	0 0 0 1 0 0	X	X	X	Multiply the Operand A and Operand B registers, placing the result in Result register.
COMP	0 0 0 1 0 1	X	—	X	Copy the complement of the Operand A register into the Result register.
INC	0 0 0 1 1 0	X	—	X	Increment Operand A register by 1, placing result in Result register.
DEC	0 0 0 1 1 1	X	—	X	Decrement Operand A register by 1, placing result in Result register.
HALT	0 0 0 1 0 0 0	—	—	—	Halt the processor.
LOAD	1 0	Destination	—	Address	Load Destination register with contents of memory at location Address.
STORE	1 1	Source	—	Address	Store Source register in memory at location Address.



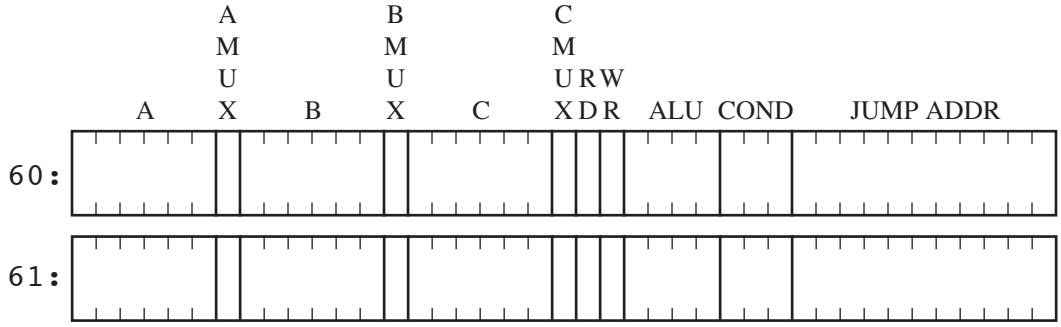
NOTE: There are 8 registers in this architecture.

CHAPTER 5 PROBLEMS

(5-1) List the microinstructions that are executed in interpreting the ARC `"st %r1, %r2, %r3"` instruction. Start with microinstruction 0. Just list the microinstruction numbers, not the code.

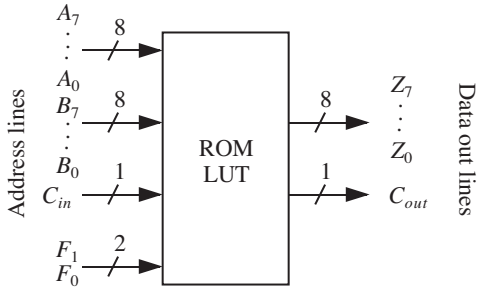
(5-2) Write the binary form for the microinstructions shown below. Use the value 0 for any fields that are not needed.

```
1615: R[temp1] ← R[rs1]; GOTO 21;
  21: R[temp2] ← NOR(R[temp0], R[temp0]);
```



(5-3) Microcode: An ALU that operates on 8-bit operands can be implemented with a read-only memory (ROM) that behaves as a lookup table (LUT), in which the data and control bits are treated alike. The block diagram shown below illustrates the external behavior of the ALU that we would like to implement with a ROM LUT. The *A* and *B* lines are for the operand inputs, and the *F* lines (that is, the control inputs) select the ALU function according to the truth table shown below. The C_{in} and C_{out} lines are for the carry in and the carry out, respectively. The carry out is always 0 when an addition is not taking place. The 8-bit output appears on the *Z* lines. Assume that a two's complement representation is used. Fill in the missing entries in the truth table below.
[HINT: The solution for the first row is all 1's for Z_7 - Z_0.]

F_1	F_0	Function
0	0	NOR
0	1	Multiply
1	0	AND
1	1	Add



A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	C_{in}	F_1	F_0	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0	C_{out}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	_	_	_	_	_	_	_	_	0
_	_	_	_	_	_	_	_	0	0	0	0	0	0	1	1	0	0	1	0	0	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	_	_	_	_	_	_	_	_	_

CHAPTER 6 PROBLEMS

(6-1) Translate the following Pascal code into SPARC (note: this is a variation of a problem from the Tanenbaum *Structured Computer Organization* textbook).

```
/* Compute min of i and j */
function min (i, j: Int): Int     /* i and j are parameters, min returns Int*/
var m: Int;
begin
        if i < j then m := i else m := j;
        min := m        /* This is how a Pascal f        retur        ue */
end;
```

(6-2) Translate the single line indicated in the following AR[illegible] into obj[illegible] Remember: branch displacements are in terms of 4-byte words, not bytes. (That is, an offset of 8 bytes is represented as 2, because it is equivalent to an offset of 2 word

```
      .begin
      .org    0

      srl     %r2, 10, %r2      ans:  10   00010   100110   00010   1   0000000001010
```

(6-3) Disassemble the following ARC object code into a source statement. Since there is not enough information in the object code to determine symbol names, use numbers (offsets, like +5 or –12) instead, if needed.

```
   00000010 10000000 00000000 00000011
```

(6-4) Given the assembly language program shown on the next page, which uses an assembly language syntax that should be new to you, construct a symbol table. As with the ARC assembly language, each instruction in this assembly language occupies four bytes. Not all of the lines in the table will be needed. Do not try to figure out what the program does – it is not helpful to know this. Mark any undefined symbols with an "X." All values are given in base 10. Registers are named R0, R1, and R3

```
/ All numbers are given in base 10
K           EQU     77
P           EQU     1

            ORG     32

            JUMP    MAIN
PMUL:       SUB     2,R0
            LOAD    A,R1
            BZERO   ANOTZ
            LOAD    0,R1
            JUMP    DONE
ANOTZ:      LOAD    B,R3
            BZERO   BNOTZ
            LOAD    0,R3
            JUMP    DONE
BNOTZ:      LOAD    0,R1
            STORE   R1,A
            BZERO   PMUL
DONE:       JUMP    DONE
A:          12
B:          15
```

Symbol	Value
K	77
P	1
MAIN	X
PMUL	36
A	88
ANOTZ	56
DONE	84
B	92
BNOTZ	72

(6-5) Define a SPARC macro `mov` that copies its first argument (a literal) into its second argument (a register). For example, the code

```
mov 10, %r24
```

copies 10 into register `%r24`.

(6-6) In the compiled SPARC code used in the book, both `call` and `jmpl` need two instruction cycles to complete. Both instructions are followed by a `nop` if there is no nearby instruction that can replace the `nop`. The `ret` instruction is actually a pseudo-instruction that translates to:

```
jmpl %r15+8, %r0
```

In the other code examples, however, the following code is used for returning from a subroutine call:

```
jmpl %r15+4, %r0
```

What would happen if we used the `%r15+4` version for the `ret` in the compiled code? In what way(s) would the program behave differently?

ans: The program would still work correctly, but the return would go to the `nop` instruction that follows the `call`, thus wasting an instruction cycle.

(6-7) Given an architecture that implements the following instructions:

Mnemonic	Operand(s)	Meaning
`PUSH`	`arg`	Push contents of memory location `arg` onto stack
`POP`	`arg`	Pop stack into memory location `arg`
`BA`	`label`	Branch to `label`
`ADD`	`arg1, arg2, arg3`	`arg3 <- arg1 + arg2`

What is the result of executing the following program?

```
PUSH A
ADD  A, 1, A
PUSH A
ADD  A, 1, A
PUSH A
POP  B
ADD  B, -1, B
POP  B
ADD  B, -1, B
POP  B
```

(6-8) A program is running on a pipelined computer in which every fourth instruction is a jump (or a branch), and there is a 20% probability that each jump is taken. When a jump is taken, the pipeline is flushed, which has a branch penalty of 3. Compute the average instruction time in terms of instruction cycles.

(6-9) Consider the following code segment, which is for an architecture that many be new to you (the MIPS):

```
      LW R1, 0(R2)          Load a word from memory
      SUB R1, R1, R3        Subtract
      BEQZ R1, L            Branch if equal to zero
      OR R4, R5, R6         or
            .
            .
            .
L:    ADD R7, R8, R9        Add
```

There is a load delay slot after the load from memory instruction: LW. The dependence of the subsequent SUB and BEQZ instructions on the LW instruction means that we need a stall after LW. Suppose we know that the branch is almost always taken, and that the value of R7 is not needed on the fall-through path. Suggest a rescheduling of the code segment which would increase the speed of the program.

CHAPTER 7 PROBLEMS

(7-1) A cache has a 95% hit ratio, an access time of 100ns on a cache hit, and an access time of 800ns on a cache miss. Compute the effective access time.

(7-2) (From Stallings, 1999) A set associative cache consists of 64 slots divided into 4-slot sets. Main memory contains 4K blocks of 128 words each. Show the format of the main memory address.

(7-3) Using the page table shown below, translate virtual address 2050 into a physical address, and translate physical address 25 into a virtual address. Address length is 16 bits, page size is 2048 words. Physical memory has 4 page frames.

Page	Present (1-in/0-out)	Page Frame
0	1	3
1	1	2
2	1	0
3	0	--
	...	

(7-4) A computer has 16 pages of virtual address space but only 4 page frames. Initially the memory is empty. A program references the virtual pages in the order: 0 2 4 5 2 4 3 11 2 10.

(a) Which references cause a page fault with the LRU page replacement policy?

(b) Which references cause a page fault with the FIFO page replacement policy?

(7-5) When running a particular program with N memory accesses, a computer with a cache and paged virtual memory generates a total of M cache misses and F page faults. T1 is the time for a cache hit; T2 is the time for a main memory hit; and T3 is the time to load a page into main memory from the disk.

(a) What is the cache hit ratio?

(b) What is the main memory hit ratio? That is, what percentage of main memory accesses do not generate a page fault?

(c) What is the overall effective access time for the system?

(7-6) Four 256-word × 8-bit PROM chips are used to produce a total capacity of 1024-word × 8-bits. How many address bus lines are required? (Circle one.)

(A) 4 (B) 8 (C) 10 (D) 16 (E) 32

(7-7) A direct mapped cache consists of 4 blocks of 16 words per block. Main memory contains 32K blocks of 16 words each. The hit time for a cache access is 10 ns, and the miss time is 200 ns, which includes the time to transfer the missed block from the main memory to the cache. Note: When referring to memory, 1K = 1024. Compute the hit ratio for a program that loops 10 times from locations 0 – 64.

(7-8) A set associative cache consists of 64 slots divided into 4-slot sets. Main memory contains 4K blocks of 128 bytes each. Show the format of the main memory address. (Tag, Slot or Set, and Byte fields.)

(7-9) An operating system uses a Least Recently Used (LRU) page replacement algorithm. Consider the following page reference ordering (pages are referenced from left to right):

1, 8, 1, 7, 8, 2, 7, 2, 1, 8, 3

Which of the following is the number of page faults that are generated for this particular LRU case assuming that the process has been allocated four page frames, and that initially, none of the pages are in the main memory? (Circle one.)

(A) 6 (B) 5 (C) 4 (D) 3 (E) 7

(7-10) True or False (choose one): The purpose of virtual memory is to increase the speed of main memory, and the purpose of cache memory is to increase the size of main memory.

(7-11) A direct mapped cache consists of 256 slots. Main memory contains 32K blocks of 16 words each. Access time of the cache is 10 ns, and the time required to fill a cache slot is 200 ns. Load-through is **not** used; that is, when an accessed word is not found in the cache, the entire block is brought into the cache, and the word is then accessed through the cache. Initially, the cache is empty. Note: When referring to memory, 1K = 1024.

(a) Show the format of the memory address (show the number of bits in the Tag, Block or Set (as appropriate), and Byte fields, and their relative position from left to right.)

(b) Compute the hit ratio and effective access time for a program that loops 10 times from locations 15 – 52 (there is no need to divide it out, without a calculator). Note that although the memory is accessed twice during a miss (once for the miss, and once again to satisfy the reference), a hit does not occur for this case. To a running program, only a single memory reference is observed.

$$\text{Hit ratio} = \frac{\text{No. times referenced words are in cache}}{\text{Total number of memory accesses}}$$

$$\text{Eff. access time} = \frac{\text{(No. hits) (Time per hit)} + \text{(No. misses) (Time per miss)}}{\text{Total number of memory accesses}}$$

(7-12) A virtual memory system has a page size of 512 bytes, eight virtual pages, and four physical page frames. The page table is as follows:

Page #	Present bit	Disk address	Page frame field
0	0	01001011100	xx
1	0	11101110010	xx
2	1	10110010111	00
3	0	00001001111	xx
4	1	01011100101	01
5	0	10100111001	xx
6	1	00110101100	11
7	0	01010001011	xx

(a) What is the main memory address for virtual address 1024?

(b) What is the virtual address for main memory address 512?

(7-13) A memory has 2^{24} addressable locations. What is the smallest width in bits that the address can be while still being able to address all 2^{24} locations?

(7-14) If a virtual memory system has 4 pages in real memory and the rest must be swapped to disk, determine the hit ratio for the following page address system. Assume memory starts empty. Use the First In First Out (FIFO) page replacement policy. Choose the closest answer.

PAGE REQUESTS: 2 5 3 4 1 4 7 2 1 3 1 7 4 5 4 6

(A) 10%

(B) 15%

(C) 25%

(D) 31%

(E) 50%

(7-15) A memory system has a two-level cache in which Level 1 is closer to the CPU than Level 2. The hit time for the Level 1 cache is T1 and the hit time for the Level 2 cache is T2. The miss time for the Level 2 cache is T3. On a cache miss at either level, the miss time includes the time to read in a block and deliver the requested word. What is the effective access time of the memory system if the hit ratios of both caches are 90%? An equation for the effective access time of a two-level cache is shown below:

$$T_{EFF} = \frac{\begin{array}{l}\text{(No. on±chip cache hits) (Time per on±chip cache hit) +} \\ \text{(No. off±chip cache hits) (Time per off±chip cache hit) +} \\ \text{(No. off±chip cache misses) (Time per off±chip cache miss)}\end{array}}{\text{Total number of memory accesses}}$$

CHAPTER 8 PROBLEMS

(8-1) A hard magnetic disk with a single platter rotates once every 16 ms. There are 8 sectors on each of 1000 tracks. An interleave factor of 1:2 is used. What is the fastest possible time to copy a track from the top side of the platter to the corresponding track on the bottom side of the platter? Assume that the sectors must be read in numerical order starting from 0, that the top and bottom

tracks must be mirror images, that any number of sectors can be read from the top track before writing them to the bottom track, and that simultaneous reading and writing is not allowed (even on different tracks.)

(8-2) Consider a disk drive with the following characteristics:

7200 revolutions per minute rotation speed
7 msec average seek time
256 sectors per track, with 512 bytes per sector
2048 tracks per surface
16 surfaces
1 head per surface, all heads move together as a group
reading and writing cannot be done at the same time

(a) What is the total capacity of the disk drive?

(b) What is the data transfer rate in bytes per second for this drive? That is, once the head is positioned over the sector to be read, what is the data transfer rate?

(c) What is the average time to transfer a whole sector from one track to another track?

(8-3) A number of disks, a CPU, and the main memory are all connected to the same 10 MHz 32-bit bus. The disk has a transfer rate of 2 MBytes/sec. The CPU and main memory can both keep pace with the bus. How many disks can be simultaneously transmitting?

CHAPTER 9 PROBLEMS

(9-1) Construct an SEC code for EBCDIC 'T'. When constructing the code, number the bits from right to left starting with 1 (the same way it is done in the book).

(9-2) Construct the SEC code for the Unicode character '1/4' = $00BC_{16}$ using even parity. Hint: check bits go in positions that correspond to powers of 2.

(9-3) If 7-digit telephone numbers are assigned so that misdialing one digit results in an unassigned number, how many numbers can be assigned?

(9-4) For SEC of 7-digit phone numbers, we need to add check digits such that the Hamming distance(H.D.) is 3 (for error correction of p digits, we need H.D. = $2p+1$. For SEC, p=1 so H.D. is 3). How many check digits should we add?

(9-5) In alphabetic encryption (AE), a string of characters is mapped into a different, longer string of characters consisting of only upper case letters. The AE code is shown below:

Letter	Code	Letter	Code	Letter	Code
A	00000	J	01001	S	10010
B	00001	K	01010	T	10011
C	00010	L	01011	U	10100
D	00011	M	01100	V	10101
E	00100	N	01101	W	10110
F	00101	O	01110	X	10111
G	00110	P	01111	Y	11000
H	00111	Q	10000	Z	11001
I	01000	R	10001		

(a) Compute the checksum word for the AE characters C, D, E, F and G. Use both longitudinal and vertical redundancy checking.

(b) A receiver sees the following bit pattern, which is Hamming encoded using even parity (the way it is done in the book). What AE character was transmitted?

(9-6) A single-user workstation is attached to a local network. The workstation accesses files over the network from a file server. The average access time is 0.09 seconds per page. A similar stand-alone workstation accesses files from its own local disk with an average access time of 0.03 seconds per page. A particular program accesses and processes a 300-page file. The time to process the file once the data is in memory is 45 seconds. What is the ratio of the total time to access and process the file for the local network workstation to the total time for the stand-alone workstation?

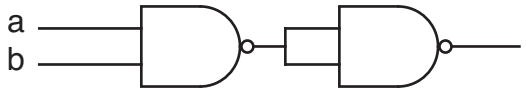
(A) 3/1 (D) 5/2

(B) 4/3 (E) 1/1

(C) 8/5

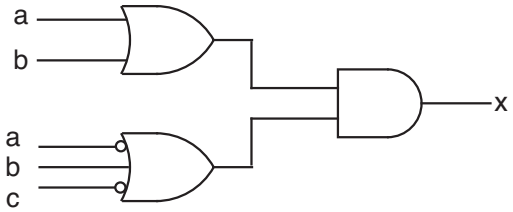
APPENDIX A PROBLEMS

(A-1) The combinational circuit given below is implemented with two NAND gates. To which of the following individual gates is it equivalent?



(A) NOT (B) OR (C) AND (D) XOR (E) NOR

(A-2) Derive function x represented by the following circuit. The prime symbol (') has the same meaning as overbar.



(A) ab + a'bc' (B) (a + b)(a' + b + c') (C) ab + a'bc' (D) (ab)(a'bc') (E) bc'

(A-3) Design a binary-to-Gray code converter using an 8-to-1 MUX, a 4-to-1MUX, and a 16-to-1 MUX. Use the truth table shown below:

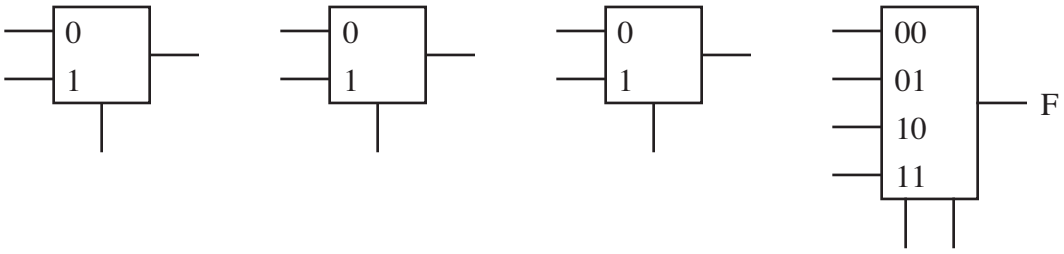
x2	x1	x0	z2	z1	z0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

(A-4) Design a circuit that implements function *p* below using AND, OR, and NOT gates. DO NOT change the form of the equation. The circuit should implement this function exactly.

$$f(i_0, i_1, i_2) = i_2(i_0 i_1 + \overline{i_0}\,\overline{i_1})$$

(A-5) Label and connect the MUXes shown below to implement function F:

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



(A-6) The black box in the following figure consists of a combinational logic unit that uses only AND, OR, and NOT gates.

The function f(x, y, z) = 1 whenever x, y are different and 0 otherwise. Which of the following equations leads to the correct design for the combinational logic unit? The prime symbol (') in X' has the same meaning as an overbar.

(A) x'y + xy' (B) x + y'z (C) x'y'z' + xy'z (D) xy + y'z + z' (E) x'z + xy + y'z'

(A-7) How many distinct Boolean functions of 3 variables are there? This question is not asking how many possible unique combinations of 3 variables there can be, but rather, how many unique functions of those 3 variables can there be?

(A) 16 (B) 64 (C) 256 (D) 512 (E) 1024

(A-8) Create a state transition diagram that outputs a 1 whenever the sequence of two-bit inputs 00→01*→11 is detected. The asterisk means that the bit-pair 01 can appear any number of times at that position in the sequence, including not at all (zero times). Show the state transition diagram only (do not create a state table or draw a circuit).

(A-9) Design a finite state machine (FSM) that has two control lines C_0 and C_1. The FSM has an output Z that is a 1 if C_1C_0=00, a 0 ifC_1C_0=01, the value of its current output if C_1C_0=10, and the complement of its current output ifC_1C_0=11. Just draw a state transition diagram that describes the behavior of the FSM.

(A-10) For the FSM described by the state table shown below,

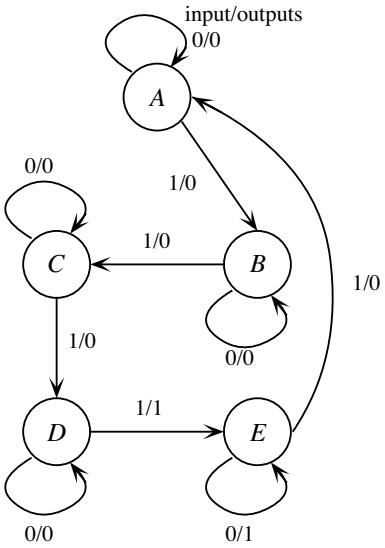
(a) What is the smallest number of flip-flops needed to implement the FSM? (Do not apply any state reduction techniques.)

(b) How many Boolean functions need to be created to implement this FSM? That is, how many next-state and output functions are there?

Input / P.S.	X = 0	X = 1
$s_2s_1s_0$	$s_2s_1s_0/z$	$s_2s_1s_0/z$
A:000	001/0	010/0
B:001	001/0	011/0
C:010	001/0	100/0
D:011	001/0	100/1
E:100	001/1	100/0

(A-11) Design a sequential machine that outputs a 1 when the last three inputs are 011 or 110. Note that sequences can overlap, so that an input sequence of 0110 will produce an output sequence of 0011. Just show the state transition diagram. Do not reduce the diagram or draw a circuit.

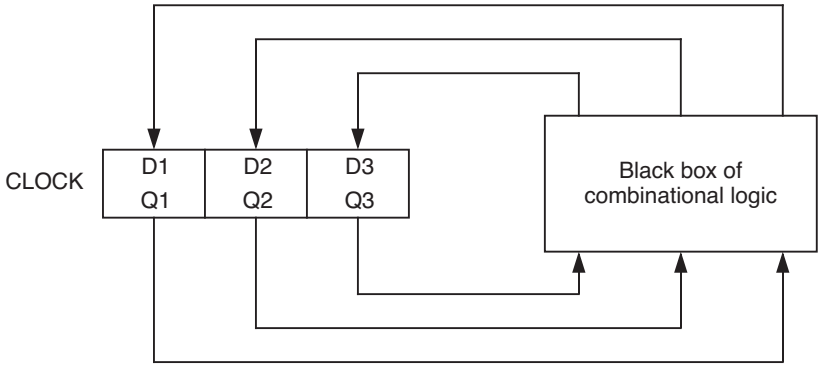
(A-12) Given the state transition diagram shown below, design a circuit for this state machine using D flip-flop(s) and one 4-to-16 decoder with OR gates. For the state assignment, use the bit pattern that corresponds to the position of each letter in the alphabet, starting from 0 (A = 000, B = 001, C = 010, *etc*.).



Present State	Input (x) 0	1
A		
B		
C		
D		
E		

(A-13) Design an FSM using D flip-flops and the smallest possible ROM for a remote control that operates a TV:0, a VCR:1, and a CATV:2 device. The user selects a device by pressing the corresponding key. Once a device is selected, the user can increase the channel number (3) or decrease the channel number (4). The input (0, 1, 2, 3, or 4) is repeated on the corresponding output port (one for each device), except for 0, 1, and 2 which produce 0's at all outputs. (Hint: The STD is easier to draw if you use base 10.)

(A-14) Consider the following circuit which contains a 3-bit register and a black box with some combinational logic:



The initial state of the circuit is $Q_1Q_2Q_3 = 000$. The circuit generates the sequence

$$(010) \rightarrow (110) \rightarrow (001) \rightarrow (001) \rightarrow \ldots \rightarrow (001)$$

on successive clock cycles. Which of the following sets of equations are implemented by the combinational logic in the black box? The prime symbol (') in X' has the same meaning as an overbar.

(A) $D_1 = Q_1'Q_2'Q_3'$ $D_2 = Q_1'$ $D_3 = Q_2$

(B) $D_1 = Q_2Q_3'$ $D_2 = Q_1Q_2'$ $D_3 = Q_1Q_2Q_3'$

(C) $D_1 = Q_1 + Q_2$ $D_2 = Q_1'$ $D_3 = Q_1Q_2$

(D) $D_1 = Q_1'Q_2Q_3'$ $D_2 = Q_1'Q_2'Q_3' + Q_1'Q_2Q_3'$ $D_3 = Q_1Q_2Q_3' + Q_1'Q_2'Q_3$

(E) $D_1 = Q_1Q_2Q_3'$ $D_2 = Q_1'Q_2'Q_3' + Q_1'Q_2'Q_3'$ $D_3 = Q_1'Q_2Q_3'$

(A-15) Design a control unit for a simple hand-held video game in which a juggler on the display catches objects. Treat this as an FSM problem, in which you only show the state transition diagram. Do not show a state table, state assignment, or a circuit. The input to the control unit is a two-bit vector in which 00 means "Move Left," 01 means "Move Right," 10 means "Do Not Move," and 11 means "Halt." The two-bit output *Z* is 11 if the machine is currently halted, and is 00, 01, or 10 otherwise, corresponding to the input patterns 00, 01, and 11. That is, the two-bit output is identical to the two-bit input, except when the machine is halted, in which case the output is 11 regardless of the input. Once the machine is halted, it must remain in the halted state indefinitely. Show only the state transition diagram.

SOLUTIONS

CHAPTER 1 SOLUTIONS

(1-1)

(1) mechanical - manually operated
(2) mechanical - machine powered
(3) vacuum tubes
(4) transistors
(5) integrated circuits

CHAPTER 2 SOLUTIONS

(2-1) (B) 0011 1011 0111 1111

(2-2)

(a) 53_{10}

(b) 11100011_2

(c) 6363_{16}

(d) 1101_3

(2-3) (C) -2^{31} to $2^{31} - 1$

(2-4) (B) the range is increased but the precision is decreased

(2-5)

(a) Smallest = $.1 \times b^{-q}$
Largest = $(1 - b^{-p}) \times b^{(X - q)}$

(b) Smallest gap = $b^{-p} \times b^{-q}$
Largest gap = $b^{-p} \times b^{(X - q)}$

(c) $2 \times (X + 1) \times (b - 1) \times b^{p-1} + 1$

(2-6)

```
1   0 1 1 1 1 0 1 0  .  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
S    Exponent                    Fraction
```

$1/32 = (.00001)_2 = 1.0 \times 2^{-5}$. The 1 to the left of the radix point is dropped, which is why the fraction is all 0's. The exponent is in excess 127 (not 128), so -5+127 = 122 for the exponent.

(2-7) (a) $-.1_4 \times 4^{4-4} = -.25_{10}$

(b) 0 000 010000

(c) 0 111 111111

(d) 2 (sign bit) $\times$ 2^3 (exponent) $\times$ 3 (first digit) $\times$ 4^2 (remaining digits) + 1 (zero) = 769

(2-8)

```
0   1 0 0 0 0 1 0 1  .  1 0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

CHAPTER 3 SOLUTIONS

(3-1)

```
  0 1 0 1 1 0       1 0 1 0 1 1       1 1 1 1 1 1
+ 0 0 1 0 0 1     + 1 0 0 1 0 1     + 0 0 0 1 1 1
-------------     -------------     -------------
  0 1 1 1 1 1       0 1 0 0 0 0       0 0 0 1 1 0
 No overflow        Overflow          No overflow


  0 1 0 1 1 0       1 1 1 1 1 0       1 0 0 0 0 1
- 0 1 1 1 1 1     - 1 0 0 1 0 1     - 0 1 1 1 0 1
-------------     -------------     -------------
  1 1 0 1 1 1       0 1 1 0 0 0       0 0 0 1 0 0
 No underflow      No underflow       Underflow
```

(3-2)

```
          1 0 1 0 1   21
        x 0 0 1 0 1    5
        -----------
          1 0 1 0 1
        0 0 0 0 0
      1 0 1 0 1
    0 0 0 0 0 0
0 0 0 0 0 0
-----------------
0 0 1 1 0 1 0 0 1  105

          00100 R1
        +--------
00101   | 10101
        00101
        -------
          00001
```

(3-3)

Step #	Product			Action
0	C=0	M=10110	Q=01101	Initial values
	C	A	Q	
1	0	10110	01101	Add M to A
2	0	01011	00110	Shift

3	0	00101	10011	Shift (no add)
4	0	11011	10011	Add M to A
5	0	01101	11001	Shift
6	1	00011	11001	Add M to A
7	0	10001	11100	Shift
8	0	01000	11110	Shift

(3-4)

```
          0 1 1 0    Multiplicand
          0 1 1 0    Multiplier
x         1 0 1 0    Booth recoded multiplier
-------------------
  1 1 1 1 0 1 0 0   -1 x 0 1 1 0
+ 0 0 1 1 0          1 x 0 1 1 0
-------------------
  0 0 1 0 0 1 0 0
```

(3-5)

```
             0 1 0 1 0 0   (Multiplicand)
             1 1 0 1 1 0   (Multiplier)
             0-1 1 0-1 0   Booth recoded multiplier
  x           -1   2  -2   Bit-Pair recoded multiplier
  -------------------------------
   1 1 1 1 1 1 0 1 0 1 1
 + 0 0 0 0 1 0 1 0 1
 + 1 1 1 0 1 0 1 1
  -------------------------------
   1 1 1 1 0 0 1 0 1 1 1
```

(3-6)

Booth:

```
                  1 1 1 1 0 1 1 0  -10 (Multiplicand)
                  1 1 1 1 0 1 1 0  -10 (Multiplier)
  x               0 0 0-1 1 0-1 0  Booth recoded multiplier
  -------------------------------
  0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0  -10 x -2
+ 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0  -10 x 8
+ 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0  -10 x -16
  -------------------------------
  0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 = 100
```

Bit-pair:

```
                  1 1 1 1 0 1 1 0  -10 (Multiplicand)
                  1 1 1 1 0 1 1 0  -10 (Multiplier)
```

```
                    0  0  0-1  1  0-1  0   Booth recoded multiplier
  x                   0   -1    2   -2   Bit-Pair recoded multiplier
  -------------------------------
  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  0   -10 x -2
+ 1  1  1  1  1  1  1  1  1  0  1  1  0  0  0  0   -10 x 8
+ 0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0   -10 x -16
  -------------------------------
  0  0  0  0  0  0  0  0  0  1  1  0  0  1  0  0 = 100
```

(3-7)

$b_0 = 0$

$b_1 = G_0 + P_0 b_0 = G_0$

$b_2 = G_1 + P_1 b_1 = G_1 + P_1 G_0$

$b_3 = G_2 + P_2 b_2 = G_2 + P_2 G_1 + P_2 P_1 G_0$

$b_4 = G_3 + P_3 b_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$

CHAPTER 4 SOLUTIONS

(4-1) (B) $max(A , B)$

(4-2)

(a) $2^{23} - 2^{19} + 4$

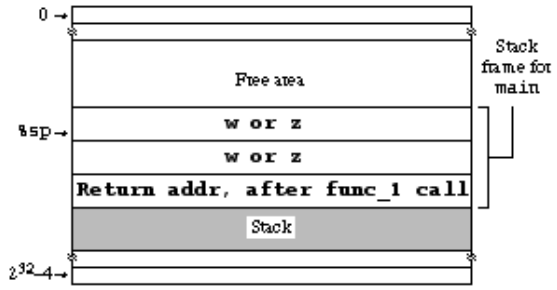
(b)

```
                .begin

        ld      [joy_x], %r7    ! %r7 and %r8 now point to the
        ld      [joy_y], %r8    !   joystick x and y locations
        ld      [flash], %r9    ! %r9 points to the flash location
loop:   ld      %r7, %r1        ! Load current joystick position
        ld      %r8, %r2        !   in %r1=x and %r2=y
        ld      [old_x], %r3    ! Load old joystick position
        ld      [old_y], %r4    !   in %r3=x and %r4=y
        orncc   %r3, %r0, %r3   ! Form one's complement of old_x
        addcc   %r3, 1, %r3     ! Form two's complement of old_x
        addcc   %r1, %r3, %r3   ! %r3 <- joy_x - old_x
        be      x_not_moved     ! Branch if x did not change
        ba      moved           ! x changed, so no need to check y
x_not_moved:                    ! Your code starts here, about four lines.
        orncc   %r4, %r0, %r4   ! Form one's complement of old_y
        addcc   %r4, 1, %r4     ! Form two's complement of old_y
        addcc   %r2, %r4, %r4   ! %r4 <- joy_y - old_y
        be      loop            ! Repeat
! This portion of the code is entered only if joystick is moved.
! Flash screen; store new x,y values; repeat.
```

```
moved:
        addcc   %r0, 1, %r5
        st      %r5, %r9
        st      %r1, [old_x]
        st      %r2, [old_y]
        ba      loop

flash:  #FFFFEC                 ! Location of flash register
joy_x:  #FFFFF0                 ! Location of joystick x register
joy_y:  #FFFFF4                 ! Location of joystick y register
old_x:  0                       ! Previous x position
old_y:  0                       ! Previous y position
                .end
```

(4-3)

```
main() /* Note: main() has no incoming parameters */
{
    int w, z;          /* Local variables */
  /***** Show stack frame for this location *****/
    w = func1(1,2);   /* Call subroutine func 1 */
    z = func2(10);    /* Call subroutine func 2 */
}                     /* End of main routine */
```



(4-4)

There is more than one solution. Here is one:

0000011 001 001 101

CHAPTER 5 SOLUTIONS

(5-1) Note that `%r1` goes in the `rd` field for an `st` instruction:

```
11  00001  000100  00010  0  00000000  00011
op    rd     op3    rs1   i             rs2
```

`DECODE: 1 11 000100 00 = 1808`

Microinstructions: 0, 1, 1808, 1809, 40, 41, 42, 43, 44, 2047

(5-2)

```
1615:  000000  1  000000  0  100010  0  0  0  0001  110  00000010101
  21:  100001  0  100001  0  100011  0  0  0  0010  000  00000000000
```

(5-3)

```
00000000  00000000  0  00  00000000  0
00000111  00000011  0  01  00010101  0
11111111  11111111  1  11  11111110  1
```

CHAPTER 6 SOLUTIONS

(6-1) The important point in making the translation to SPARC is how to manipulate the stack frame. Prior to the function call, the calling routine places the parameters on the stack. When the called routine (min) is invoked, its first task should be to save the return address on the stack. There is no need to do it here since min makes no nested calls. So, the stack pointer currently points to j, and i is one word deeper into the stack:

```
        addcc   %r14, -4, %r14        ! Push m onto stack frame
        ld      %r14, 8, %r1          ! r1 <- i
        ld      %r14, 4, %r2          ! r2 <- j
        orncc   %r2, %r0, %r3         ! r3 <- ~r2
        addcc   %r3, 1, %r3           ! r3 <- r3+1 (2's comp of j)
        addcc   %r1, %r3, %r4         ! r4 <- i - j
        bneg    J_GT_I                ! branch if i<j (j>i actually)
        st      %r2, %r14             ! m <- j
        ba      DONE
J_GT_I: st      %r1, %r14             ! m <- i
DONE:   jmpl    %r15, 4, %r0          ! Return value m is on stack.
```

(6-2) `10  00010  100110  00010  1  0000000001010`

(6-3) `be 3`

(6-4)

```
/ All numbers are given in base 10
K           EQU     77
P           EQU     1

            ORG     32

            JUMP    MAIN
PMUL:       SUB     2,R0
            LOAD    A,R1
            BZERO   ANOTZ
            LOAD    0,R1
            JUMP    DONE
ANOTZ:      LOAD    B,R3
            BZERO   BNOTZ
            LOAD    0,R3
            JUMP    DONE
BNOTZ:      LOAD    0,R1
            STORE   R1,A
            BZERO   PMUL
DONE:       JUMP    DONE
A:          12
B:          15
```

Symbol	Value
K	77
P	1
MAIN	X
PMUL	36
A	88
ANOTZ	56
DONE	84
B	92
BNOTZ	72

(6-5)

```
.macro  mov  arg1,arg2
orcc    %r0, arg1, arg2
.endmacro
```

(6-6) The program would still work correctly, but the return would go to the `nop` instruction that follows the `call`, thus wasting an instruction cycle.

(6-7) When the code finishes execution:

(1) A is incremented by 2;

(2) B has the original value of A;

(3) The stack is restored to its original state.

(6-8) $1 + P_b P_t b = 1 + (.25)(.20) \times 3 = 1.15$

(6-9) Move the `Add R7, R8, R9` line after the LW instruction

CHAPTER 7 SOLUTIONS

(7-1) $T_{eff} = .95 \times 100\text{ns} + .05 \times 800\text{ns} = 135\text{ns}$

(7-2) Tag: 8 bits Set: 4 bits Word: 7 bits

(7-3)

```
       < page# ><     offset          >
2050 = 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 (virtual)
4096 =       1 0 0 0 0 0 0 0 0 0 0 1 0 (physical)

  25 =       0 0 0 0 0 0 0 0 1 1 0 0 1 (physical)
4121 = 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 (virtual)
```

(7-4) (a) 0, 2, 4, 5, 3, 11, 10

(b) 0, 2, 4, 5, 3, 11, 2, 10

(7-5) (a) $(N - M) / N$

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

(c) $T1 \times (N - M) / N + T2 \times (M - F) / M + T3 \times F$

(7-6) (C) 10

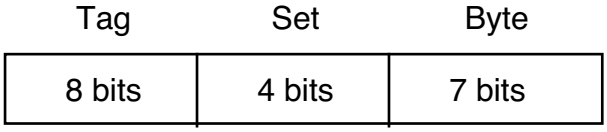
(7-7)

Iteration #1: 5 misses and 60 hits

Iterations #2 - #10: 2 misses and 63 hits per iteration

Hit ratio = hits/misses = (60 + 9 x 63) / (65 x 10) = 96.56%

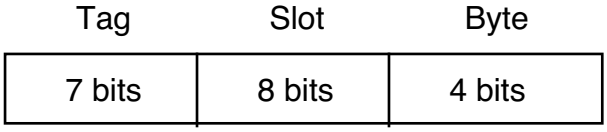
(7-8)



(7-9) (B) 5; page faults: 1, 8, 7, 2, 3

(7-10) False

(7-11) (a)



(b) Hit ratio = (34 + 9 x 38) / (10 x 38) = 376 / 380 = 98.94%

T_{eff} = [(4 misses x 210ns/miss) + (34 + 9 x 38) hits x 10 ns/hit] / 380 accesses = 12.1 ns

(7-12) (a) 00000000000

(b) This address is in Page #1 which is not present, so there is no virtual address.

(7-13) 24

(7-14) (D) 31%

(7-15) $T_{eff} = .9 \times T1 + .9 \times (1 - .9) \times T2 + [1 - .9 \times (1 - .9)] \times T3$

CHAPTER 8 SOLUTIONS

(8-1)

(1) Read 0, 1, 2, 3 on first rotation.

(2) Write 0, 1, 2, 3; read 4, 5, 6, 7 on second rotation

(3) Write 4, 5, 6, 7 on third rotation.

Total time = 3.0 rotations x 16 ms/rotation = 48 ms.

(8-2) **(a)** 16 surfaces x 2048 tracks/surface x 256 sectors/track x 512 bytes/sector = 2^{32} bytes

(b) 7200 rev/min x 1/60 min/sec x 1 track/rev x 256 sectors/track x 512 bytes/sector = 15.7 MB/sec

(c) Avg seek time = 7ms.

Avg rotational delay = [1/7200 min/rev x 60 sec/min]/2 = .00417s = 4.17ms

Sector read/write time = 1/8 x [1/7200 min/rev x 60 sec/min] = .00104s = 1.04ms

Avg transfer time = [7ms + 4.17ms + 1.04ms read] + [7ms + 4.17ms + 1.04ms write] = 24.42ms

(8-3) 10^6 words/s per bus × 1/(.5 × 10^6) s/word per disk = 20 disks/bus

CHAPTER 9 SOLUTIONS

(9-1)

```
 1  1  1  0  1  0  0  1  0  1  1  1
-- -- -- -- -- -- -- -- -- -- -- --
12 11 10  9 C8  7  6  5 C4  3 C2 C1
```

(9-2)

```
 0  0  0  0  0  0  0  0  0  1  0  1  1  1  1  1  0  1  0  1  0
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
21 20 19 18 17C16 15 14 13 12 11 10  9 C8  7  6  5 C4  3 C2 C1
```

(9-3) We want a Hamming distance of 2 between valid numbers, in base 10. In base 2, we looked for even/odd parity, which can be generalized to mod2(sum of digits in number) = 0 for even parity. We want mod10(sum of digits in number) = 0 for this problem. So, mod_10(4+4+5+2+6+5+4) = 0 is a valid telephone number (445-2654) whereas mod_10(4+4+5+3+5+2+3) = 6 is not (445-3523).

Since only 1/10 of the numbers will come out as mod_10() = 0, only 10% of the possible numbers can be assigned.

(9-4) For each of the 10^7 valid phone numbers, there is an uncorrupted version, and 9×7 ways to corrupt each of the 7 original digits, plus 9*r* ways to corrupt each of the r check digits.

The following relationship must hold:

$10^7(9\times7 + 9r + 1) <= 10^{(7 + r)}$

which simplifies to

$64 + 9r <= 10^r$

for which $r = 2$ is the smallest value that satisfies the relation.

(9-5) (a)

```
C   1 00010
D   0 00011
E   1 00100
F   0 00101
C   0 00110
Chk 0 00110
```

(b) The letter 'M'

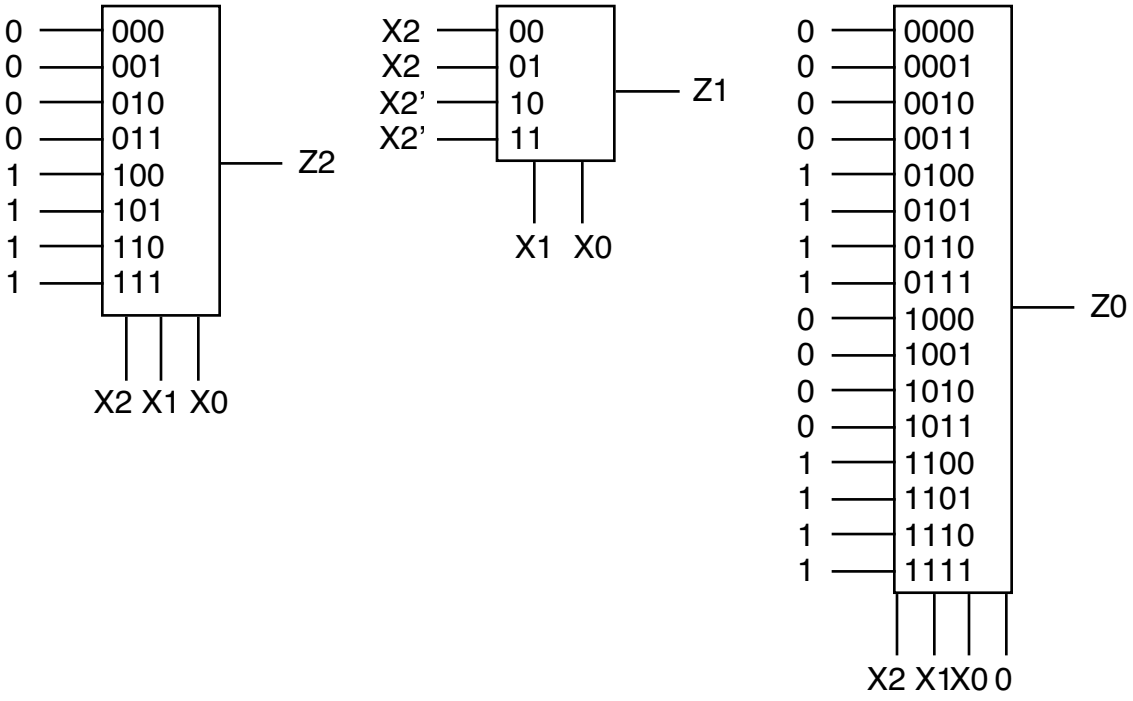
(9-6) (B) 4/3

APPENDIX A SOLUTIONS

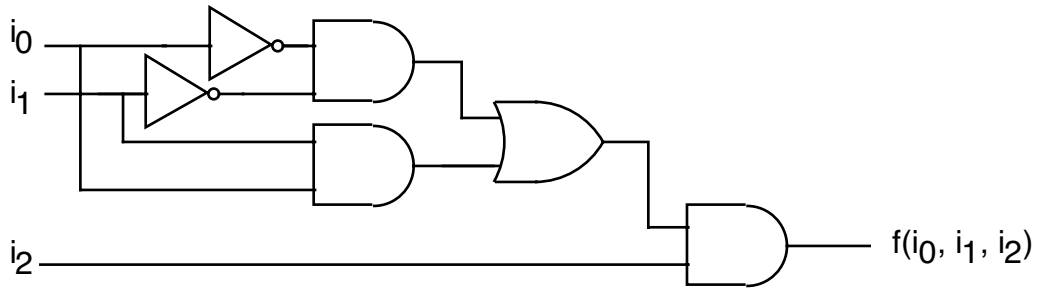
(A-1) (C) AND

(A-2) (B) (a + b)(a' + b + c')

(A-3)

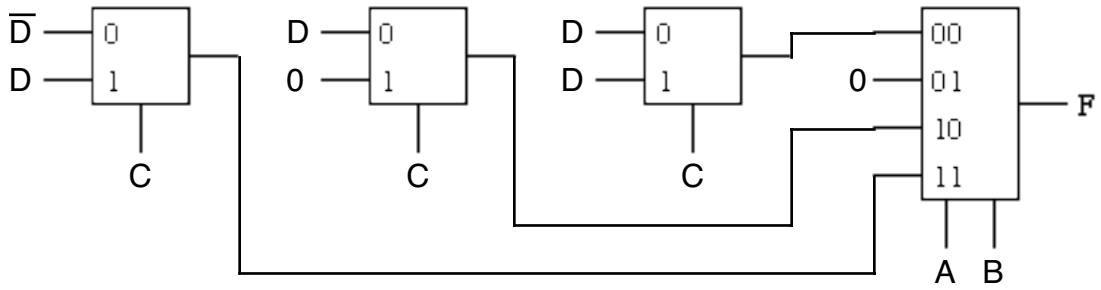


(A-4)



(A-5)

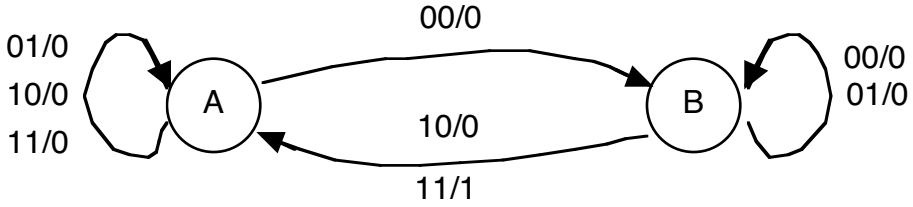
$$F(A,B,C,D) = AB(CD + \overline{C}\overline{D}) + A\overline{B}(\overline{C}D) + \overline{A}\overline{B}(CD + \overline{C}\overline{D})$$



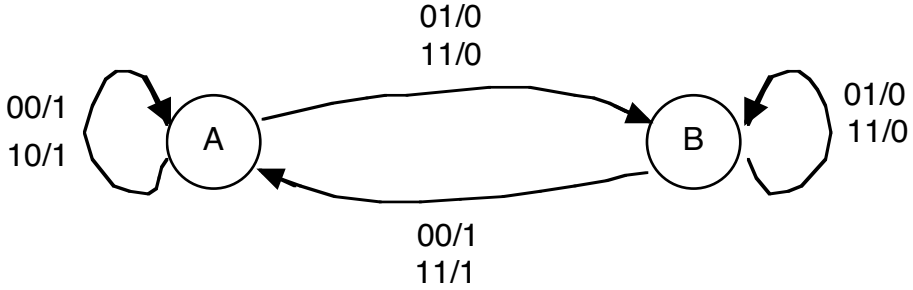
(A-6) (A) x'y + xy'

(A-7) (C) 256

(A-8)



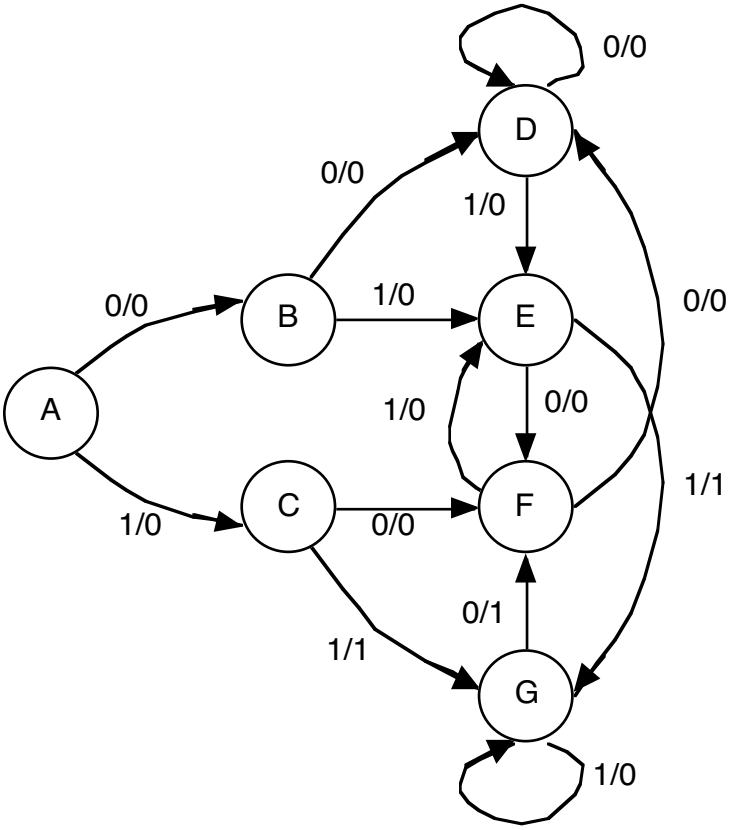
(A-9)



(A-10) **(a)** 3

(b) 4

(A-11) There is more than one solution. Here is one:



(A-12) [Partial solution]

Present State	Input (x) 0	1
A	000/0	001/0
B	001/0	010/0
C	010/0	011/0
D	011/0	100/1
E	100/1	000/0

(A-13) [Partial solution] Here is the state table:

PS	Input (in base 10) 0	1	2	3	4
A	A/000	B/000	C/000	A/300	A/400
B	A/000	B/000	C/000	B/030	B/040

```
C | A/000 | B/000 | C/000 | C/003 | C/004 |
  +-------+-------+-------+-------+-------+
```

State Assignment:

```
PS                                  Input
            000           001           010           011           100
    +-------------+-------------+-------------+-------------+-------------+
A:00|00/000 000 000|01/000 000 000|10/000 000 000|00/011 000 000|00/100 000 000|
    +-------------+-------------+-------------+-------------+-------------+
B:01|00/000 000 000|01/000 000 000|10/000 000 000|01/000 011 000|01/000 100 000|
    +-------------+-------------+-------------+-------------+-------------+
C:10|00/000 000 000|01/000 000 000|10/000 000 000|10/000 000 011|10/000 000 100|
    +-------------+-------------+-------------+-------------+-------------+
```

For an implementation, you can use two D flip flops for the state bits, and label them s1 and s0 (left to right), and a ROM with 5 address lines and 11 data out lines. Then drew a table that shows the ROM contents; here are a few locations:

```
Address          Value
00000            00 000 000 000
00001            01 000 000 000
        ...
10100            10 000 000 000
```

(The rest are don't cares, same for a few in the middle. Only 15 entries in the ROM are needed).

(A-14) (D) $D_1 = Q_1'Q_2Q_3'$ $D_2 = Q_1'Q_2'Q_3' + Q_1'Q_2Q_3'$ $D_3 = Q_1Q_2Q_3' + Q_1'Q_2'Q_3$

(A-15)