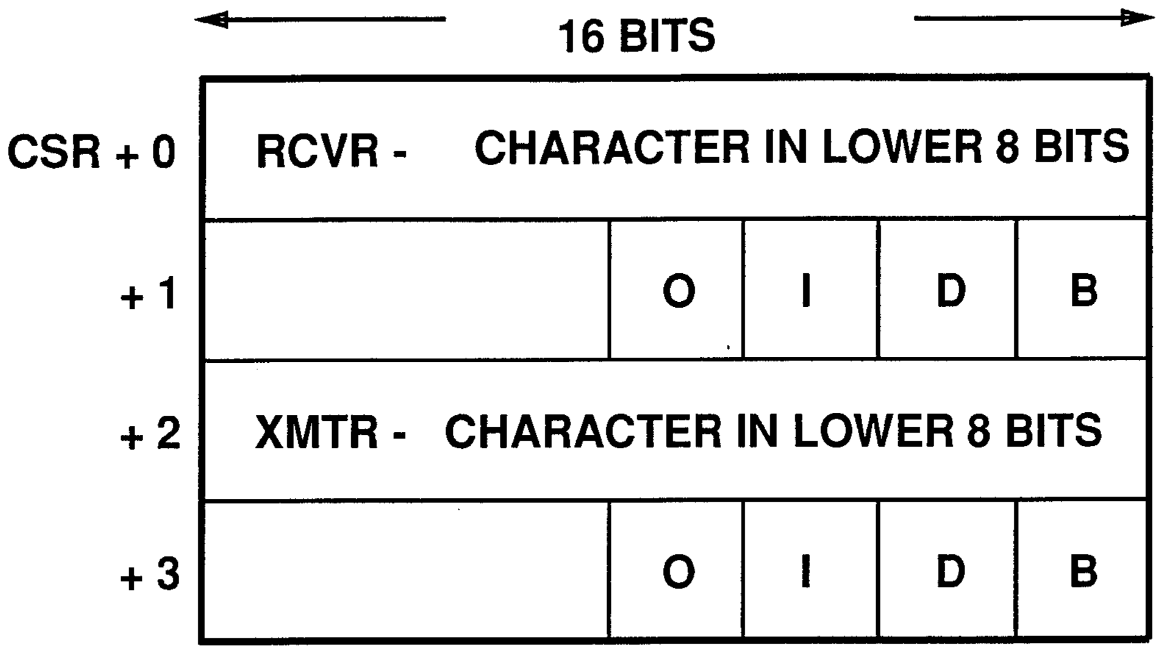

16 BITS
CSR + 0
RCVR - CHARACTER IN LOWER 8 BITS
+ 1
O
I
D
B
+ 2
XMTR - CHARACTER IN LOWER 8 BITS
+ 3
O
I
D
B

NOTE: The assembler, masm, accepts strings of the form:

```
str:    "hi mom"
str1:   "hi moma"
```

and will generate output into the assembly file as:

```
str:    0110100101101000     <==  left byte: i       right byte: h
        0110110100100000     <==  left byte: m       right byte: space
        0110110101101111     <==  left byte: m       right byte: o
        0000000000000000     <==  left byte: \0      right byte: \0
str1:   0110100101101000     <==  left byte: i       right byte: h
        0110110100100000     <==  left byte: m       right byte: space
        0110110101101111     <==  left byte: m       right byte: o
        0000000001100001     <==  left byte: \0      right byte: a
```

This means that bytes are packed in two per word with the last word either all full or half full of zeros as shown in the two different cases above. Since we use strings in this exam it is important to understand how they are deployed in memory.

1. The following assembly program (which has line reference numbers attached to help you answer the questions below) will begin executing at location 0:

```
0            lodd     start:
1            stod     4093
2            stod     4095
3 top:       lodd     4095
4            subd     mask:
5            jzer     print:
6            jump     top:
7 print:     lodd     char:
8            stod     4094
9            subd     c2:
10           stod     char:
11           subd     chA:
12           jpos     cont:
13           lodd     chZ:
14           stod     char:
15 cont:     lodd     4093
16           subd     mask:
17           jzer     done:
18           jump     top:
19 done:     halt
20 start:    8
21 mask:     10
22 chA:      "A"
23 chZ:      "Z"
24 char:     “Z”
25 c2:       2
```

A. Explain what lines **1 and 2** are used for:

B. As the program runs it begins to write characters on the display. **Describe what the output will look like.**

C. **When** and **how** will this program reach the **halt command ??**

```
0          lodd    start:
1          stod    4093
2          stod    4095
3 top:     lodd    4095
4          subd    mask:
5          jzer    print:
6          jump    top:
7 print:   lodd    char:
8          stod    4094
9          subd    c2:
10         stod    char:
11         subd    chA:
12         jpos    cont:
13         lodd    chZ:
14         stod    char:
15 cont:   lodd    4093
16         subd    mask:
17         jzer    done:
18         jump    top:
19 done:   halt
20 start:  8
21 mask:   10
22 chA:    "A"
23 chZ:    "Z"
24 char:   “Z”
25 c2:     2
```

A. Explain what lines **1 and 2** are used for:

To turn “ON” the rcvr and xmtr

B. As the program runs it begins to write characters on the display. **Describe what the output will look like.**

ZXVT – BZXVT – B……

C. When and **how** will this program reach the **halt command ??**

When some keyboard input is available

The following **busy-wait** function (starting at label xbsywt :) is used with our mic-1 IO interface in order to determine when it is **safe to place another character in the transmitter**. When called, it will not return to the caller **until the transmitter is ready** for another character. While this implementation works correctly, it is inefficient because it contains more instructions than required to perform the busy-wait. You must re-code the function using fewer instructions to achieve the same functionality:

```
xbsywt:     lodd        4095
            subd        xdmask:
            jzer        xrdy:
            jump        xbsywt
xrdy:       retn
xdmask:     10
```

↓ ***<u>YOUR RE-WRIITEN VERSION BELOW:</u>*** ↓

xbsywt:

```
xbsywt:    lodd     4095
           subd     xdmask:
           jzer     xrdy:
           jump     xbsywt:
xrdy:      retn
xdmask:    10
```

```
xbsywt:    lodd     4095
           subd     xdmask:
           jneg     xbsywt:
xrdy:      retn
xdmask:    10
```

. Contemporary random access memories can be broadly characterized as DRAM or SRAM devices.

A. **Explain why** SRAM devices are typically **faster** than DRAM devices.

All transistors, no capacitors to refresh

B. **Explain why** DRAM devices are typically **less expensive** per bit than SRAM devices.

One transistor and a capacitor cost much less than 6 or 7 transistors

C. On die **CPU cache** is generally fabricated as **SRAM** storage that deploys some specific **line size**. If an x86 CPU is trying to complete an instruction like: **`movl (%edx), %eax`** and the 4 bytes needed are **not in cache**, the resulting cache miss will typically cause the cache controller to bring in one or two cache line(s) that contain within them the 4 bytes required to complete the instruction. Since the cache line for the **newest x86 processors** (Nehalem) **is 64 bytes**, a lot more information is brought into the cache than is needed to satisfy the current instruction. **Explain and provide some examples** of the attribute of computer programs that makes this caching strategy a **performance win**.

Temporal and spatial locality (loops, function calls)

The following procedure labeled **sb:** takes one argument in the form of a **memory address** which is pushed onto the stack by the caller before it is called. As we've seen in class, this routine is supposed to do a byte swap on the content of the location passed as an argument. Part of the code is missing beginning at the label sb: and another part is missing beginning at the label **add1:** . You must add the missing code segments to each location which will make **sb:** work correctly.

```
sb:
                                                        ← write missing code

          loco  8
loop:     jzer  finish:
          subd  c1:
          stod  lpcnt:
          lodl  0
          jneg  add1:
          addl  0
          stol  0
          lodd  lpcnt:
          jump  loop:
add1:
                                                        ← write missing code
finish:   lodl  2
          popi
          retn                ; procedure ends here
c1:       1                   ; data locations for the procedure
lpcnt:    0
```

```
sb:
```

```
sb: lodl  1
    pushi
```

← write missing code

```
            loco    8
loop:       jzer    finish:
            subd    c1:
            stod    lpcnt:
            lodl    0
            jneg    add1:
            addl    0
            stol    0
            lodd    lpcnt:
            jump    loop:
add1:
```

```
add1: addl 0
      addd c1:
      stol 0
      lodd lpcnt:
      jump loop:
```

← write missing code

```
finish:     lodl    2
            popi
            retn                  ; procedure ends here
c1:         1                     ; data locations for the procedure
lpcnt:      0
```

A particular computer system provides caching by using a **direct** cache for instructions and data (often called a unified cache) consisting of **8192 sets of a single line of 16 bytes** as shown in the diagram below. The computer system uses 32 bit addressing for loads and stores and fetching instructions (its program counter (PC) is 32 bits).

Entry	**Valid**	**Tag**	**Data** (16 bytes)
0			
1			
2			
3			
. . .			
8191			

A. If the PC value of the next instruction fetch is (in hex) **0x004C71F5**, which **cache set** in the cache will be searched for the instruction ? (Give the answer in **base 10** between 0 - 8191)

0000 000 0 010 0 110 | 0 0111 0001 1111 | 0101

TAG = 0x26 **13 BIT LINE = 1823** **BYTE OFFSET**

B. What **value** will the **tag bits** be checked for in the tag field associated with the line you selected in part A ? (Give the value in **hex**.)

TAG BITS = 0x26

C. If the same 8192 cache lines were broken up into a 4 way set associative cache (4 lines per set, 2048 sets), **how many lines**, and **which cache set** would be searched for the above address (**0x004C71F5,) ?** Give the answer in **base 10)**

0000 0000 0 100 1 100 0 | 111 0001 1111 | 0101

TAG = 0x98 **11 BIT LINE = 1823** **BYTE OFFSET**

4 LINES SEARCHED FOR THIS TAG IN SET 1823

Suppose a new MACRO instruction was added to the set in the book (you've actually done this in your last assignment). This new instruction is called **BSWAPD** and will use a 4 bit op code and a 12 bit address to specify a location in memory that is to be **byte swapped** directly in its memory location, **leaving the accumulator unchanged**. (This instruction would actually have to replace an existing instruction since all possible 4 bit op-code, 12 bit address instructions are defined.) You must write a series of MIC-1 MAL (Micro Assembly Language) statements to support this new instruction. You do **NOT** have to worry about **DECODING its OPCODE** (we won't even worry about what its macro OPCODE is, but will assume that your code begins **with the first MAL statement needed after the instruction has been successfully decoded** by earlier parts of the microprogram and has jumped to your first MAL statement), and you must make sure that when you finish byte swapping the memory location, your microprogram segment will continue normal machine execution.

```
100  mar := ir; rd;
101  c := smask; rd;
102  a := mbr;
103  c := rshift(c); if z goto 107;
104  a := lshift(a); if n goto 106;
105  goto 103;
106  a := a + 1; goto 103;
107  mbr := a; wr;
108  goto 0; wr;
```

To **begin** **accessing** information on a magnetic disk requires a period of time which is usually expressed in **two additive** components of disk access time known as **average seek time** and **average rotational latency**.

A. If the **average seek time** for a magnetic disk which spins at **15,000 RPM** is **5 ms**. and the disk has **512 sectors of 512 bytes per track**, what is the **expected average time required** to actually transfer an arbitrary **single** 512 byte sector from disk to memory ?

B. Once a data transfer has begun for the disk described above, what is the **maximum transfer rate** (also known as **maximum spindle bandwidth**) in **bytes-per-second** possible on the disk if a continuous sequential transfer is made of all the data sectors on the current track (i.e. assume that a sector can now transfer in the time it takes to pass under the read/write head) ?

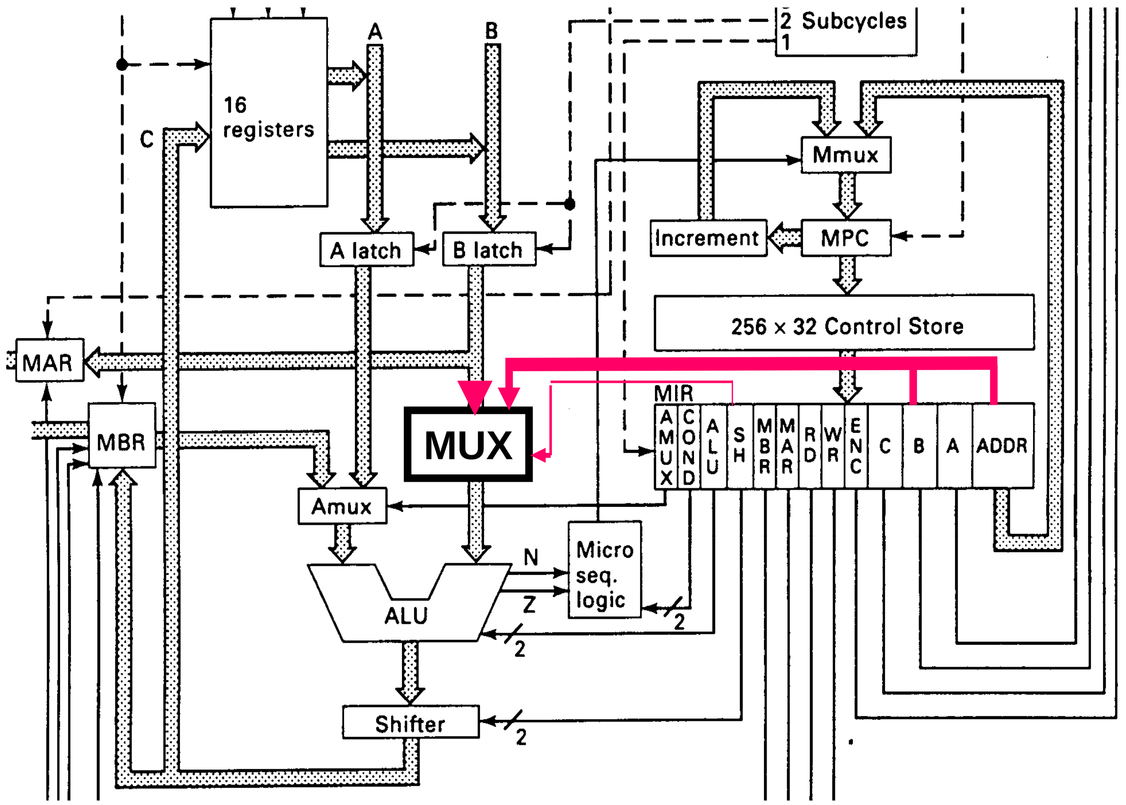
A. If the **average seek time** for a magnetic disk which spins at **15,000 RPM** is **5 ms**. and the disk has **512 sectors of 512 bytes per track**, what is the **expected average time required** to actually transfer an arbitrary **single** 512 byte sector from disk to memory ?

1m/15000R *60S/1M *1000mS/1S = 4mS/Rev
5mS + 2mS + 4mS/512Sectors = 7.0078125mS

B. Once a data transfer has begun for the disk described above, what is the **maximum transfer rate** (also known as **maximum spindle bandwidth**) in **bytes-per-second** possible on the disk if a continuous sequential transfer is made of all the data sectors on the current track (i.e. assume that a sector can now transfer in the time it takes to pass under the read/write head) ?

512 Bytes/.0078125Ms = 65.536 MB/S

The following picture shows a part of the Mic-1 hardware we've been working with (the complete picture is attached to back pages of the exam). **You need to draw on this picture the components and connections** which would have to be added to this circuit to support a new kind of MAL instruction that would somehow provide a 12 bit unsigned immediate value to be used in a way that would allow microcode statements like pc := pc + 1234. The 12 bit immediate value **must be delivered to the ALU via the B – bus access**, such that the example above would have the **pc** value entering from the **A latch,** and the **1234** immediate value somehow (**based on your new circuitry**) making its way **from a part of the MIR to the B – bus access side of the ALU.** You must add this functionality **without changing any existing functionality**, although this microinstruction may not itself have all the functionality possible of the other microinstructions. (i.e. You may need to use fields like the **A, B, C or ADDR** components to build your **12 bit immediate value**, and if used this way, they would not be available in this new MAL for their **original purpose**. This would be a **necessary trade-off** to include this new immediate capability, but only the new MAL would notice this loss of previous functionality.) The idea here is to use **any available unused bit combinations** to define another micro activity.



16 registers
A
B
C
2 Subcycles
1
A latch
B latch
Mmux
Increment
MPC
256 × 32 Control Store
MAR
MBR
MUX
MIR
AMUX
COND
ALU
SH
MBR
MAR
RD
WR
ENC
C
B
A
ADDR
Amux
N
Z
Micro seq. logic
ALU
2
2
Shifter
2