16.216: ECE Application Programming

Solution to Practice Problems for Exam 2

1. Assume the state of the 80386DX's registers and memory are:
   - (EAX) = 00005555H
   - (EBX) = 00000010H
   - (ECX) = 00000010H
   - (EDX) = 0000AAAAH
   - (ESI) = 00000100H
   - (EDI) = 00000200H
   - (DS:100H) = 0FH
   - (DS:101H) = F0H
   - (DS:110H) = 00H
   - (DS:111H) = FFH
   - (DS:200H) = 30H
   - (DS:201H) = 00H
   - (DS:210H) = AAH
   - (DS:211H) = AAH
   - (DS:220H) = 55H
   - (DS:221H) = 55H
   - (DS:300H) = AAH
   - (DS:301H) = 55H

Also, assume all flags (ZF, CF, SF, PF, OF) are initialized to 0.

For each instruction sequence shown below, list <u>all</u> changed registers and/or memory locations and their new values, as well as all changed flags from the list above. Note that the registers and memory have the same starting values at the beginning of each sequence, but a value changed by one instruction in a sequence can affect the results of all other instructions in the same sequence.

All work shown, but final answer (locations/flags changed by instruction) in *blue italics*.

a. `BT AX, 4` → **AX = 5555H = $0101\ 0101\ 010\underline{1}\ 0101_2$**
   CF = bit 4 of AX = 1

`SETC [100H]` → ***(DS:100H) = FFH,*** **since CF == 1**

`BTS AX, 5` → ***CF = bit 5 of AX = 0***
→ **Bit 5 of AX set to 1**
   AX = $0101\ 0101\ 01\underline{1}1\ 0101_2$ = 5575H

`SETC [101H]` → ***(DS:101H) = 00H,*** **since CF == 0**

`BTR AX, 6` → ***CF = bit 6 of AX = 1***
→ **Bit 6 of AX reset to 0**
   AX = $0101\ 0101\ 0\underline{0}11\ 0101_2$ = 5535H

`SETC [110H]` → ***(DS:110H) = FFH,*** **since CF == 1**

`BTC AX, 7` → ***CF = bit 7 of AX = 0***
→ **Bit 7 of AX complemented**
   AX = $0101\ 0101\ \underline{1}011\ 0101_2$ = 55B5H

`SETC [111H]` → ***(DS:111H) = 00H,*** **since CF == 0**

```
b. BSF      AL, WORD PTR [BX+SI]     → Scan word (DS:BX+SI),
                                          starting with bit 0
                                        → (DS:110H) = FF00H
                                          First non-zero bit = bit 8
                                        → AL = 08H, ZF = 1

   BSR      AH, WORD PTR [BX+SI]     → Scan word (DS:BX+SI),
                                          starting with bit 15
                                        → (DS:110H) = FF00H
                                         First non-zero bit = bit 15
                                        → AL = 0FH, ZF = 1

   CMP      AL, AH        → Compare AL to AH by subtracting AL – AH
                              AL - AH = 08H – 0FH = F9H
                            → SF = 1        (result negative)
                              ZF = 0        (result non-zero)
                              OF = 0        (no overflow)
                              CF = 1        (borrow out of MSB)
                              PF = 1        (even parity)

   JG       S             → Jump is not taken, since (AL > AH) not
                              true ((SF XOR OF) must be 0 for
                              condition code "G" to be true)

   MOV      DX, [200H]→ DX = word at (DS:200H) = 0030H

   JMP      E             → Unconditionally jump to label E, skip
                              next instruction

S: MOV      DX, [210H]

E: MOV      [BX+DI+10H],DX → (DS:BX+DI+10H) = DX
                                 (DS:220H) = 0030H →(DS:220H)= 30H
                                                    (DS:221H)= 00H
```

```
c.   CMP  AL, 56H     → Compare AL to 56H by subtracting AL – 56H
                          AL - 56H = 55H – 56H = FFH
                      → SF = 1        (result negative)
                        ZF = 0        (result non-zero)
                        OF = 0        (no overflow)
                        CF = 1        (borrow out of MSB)
                        PF = 1        (even parity)

     JL   L1          → Jump is taken, since AL < 56H
                          Next three instructions are skipped

     JG   L2
     MOV  AH, BL
     JMP  E
L1:  MOV  AH, CH      → AH = CH = 00H

     JMP  E           → Unconditionally jump to label E, skip
                          next instruction
L2:  MOV  AH, DL
E:   SETL [DI]        → If flags indicate "less than", set byte at
                          (DS:DI) = FFH, otherwise (DS:DI) = 0
                          Remember, move and jump instructions
                          don't change flags—still have same
                          values from compare instruction!
                      → (DS:0200H) = FFH
```

```
d.   MOV  AX, 0001H → AX = 0001H
     MOV  CX, 0004H → CX = 0004H
```

The following two instructions comprise a loop—the SHL instruction is the loop body, while the LOOP instruction will decrement CX and then jump back to label ST if CX is not 0.

Since CX = 0004H at the start of the loop, *the loop will execute 4 times*. Each time through the loop, AX will be shifted left by CX bits—4, then 3, then 2, then 1.

`ST:  SHL  AX, CX` → **1st iteration: AX << 4 =**
$0000\ 0000\ 0000\ 0001_2$ << 4 =
$0000\ 0000\ 0001\ 0000_2$
→ **2nd iteration: AX << 3 =**
$0000\ 0000\ 0001\ 0000_2$ << 3 =
$0000\ 0000\ 1000\ 0000_2$
→ **3rd iteration: AX << 2 =**
$0000\ 0000\ 1000\ 0000_2$ << 2 =
$0000\ 0010\ 0000\ 0000_2$
→ **3rd iteration: AX << 1 =**
$0000\ 0010\ 0000\ 0000_2$ << 1 =
$0000\ 0100\ 0000\ 0000_2$ = final value of AX

`     LOOP ST` → ***After last iteration, CX = 0***

e. `MOV AX, 8000H` → ***AX = 8000H***

The following three instructions comprise a loop—the SAR/CMP instructions are the loop body, while the LOOPNE instruction will decrement CX (which starts as 0010H), then jump back to label ST if CX is not 0 AND the result of the CMP is "not equal".

The loop has a maximum of 16 iterations, but will exit early if the value of AX == (DS:BX+SI) == (DS:110) == FF00H. As you can see below, *this early exit condition will occur after 7 loop iterations*.

`ST: SAR AX, 1`

→ **Remember, SAR maintains the sign of the original value**

→ **1st iteration: AX >> 1 =**
$1000\ 0000\ 0000\ 0000_2$ >> 1 =
$1100\ 0000\ 0000\ 0000_2$

→ **2nd iteration: AX >> 1 =**
$1100\ 0000\ 0000\ 0000_2$ >> 1 =
$1110\ 0000\ 0000\ 0000_2$

.
.
.

→ **6th iteration: AX >> 1 =**
$1111\ 1100\ 0000\ 0000_2$ >> 1 =
$1111\ 1110\ 0000\ 0000_2$

→ **7th iteration: AX >> 1 =**
$1111\ 1110\ 0000\ 0000_2$ >> 1 =
$1111\ 1111\ 0000\ 0000_2$ = FF00H
AX = FF00H = (DS:110H) → NE condition will be false; loop will end

`CMP AX, [BX+SI]` → **Compare AX to FF00H by subtracting AX – FF00H**

→ **In first 6 iterations:**

SF = 1	**(result negative)**
ZF = 0	**(result non-zero)**
OF = 0	**(no overflow)**
CF = 1	**(borrow out of MSB)**
PF depends on result	

→ **In last iteration:**

SF = 0	***(result positive)***
ZF = 1	***(result is zero)***
OF = 0	***(no overflow)***
CF = 0	***(no borrow out of MSB)***
PF = 1	***(even parity)***

`LOOPNE ST` → ***After last iteration, CX = 0009H***

2. As noted in class, the SETcc instruction can be used to combine multiple conditions together to create a compound conditional test. For example, the code below tests the condition `((A < B) && (C < D))`, storing the result in DL:

```
MOV     AX, A
CMP     AX, B
SETL    DL
MOV     AX, C
CMP     AX, D
SETL    DH
AND     DL, DH
```

For each part of this problem, assume A, B, C, D, E, and F refer to signed integers stored in memory.

a. What compound condition is tested by each of the code sequences below?

```
i.  MOV     AX, A
    CMP     AX, B
    SETLE   BL          (A <= B)
    CMP     AX, E
    SETGE   BH          (A >= E)
    OR      BL, BH      ((A <= B) || (A >= E)

ii. MOV     AX, C
    CMP     AX, A
    SETE    BL          (C == A)
    MOV     AX, B
    CMP     AX, A
    SETNE   BH          (B != A)
    AND     BL, BH      ((C == A) && (B != A))
    CMP     AX, C
    SETL    BH          (B < C)
    AND     BL, BH      ((C == A) && (B != A) && (B < C))
    CMP     AX, A
    SETZ    BH          (B – A == 0) → (B == A)
    OR      BL, BH      (((C == A)&&(B != A)&&(B < C)) || (B == A))
```

iii.
```
MOV     AX, A
SUB     AX, B       AX == A - B
CMP     AX, C
SETGE   BL          ((A – B) >= C)
MOV     AX, D
ADD     AX, E
SUB     AX, F
SETNZ   BH          ((D + E) – F != 0) → ((D + E) != F)
OR      BL, BH      (((A – B) >= C) || ((D + E) != F))
```

b. Write a sequence of instructions that tests each of the following compound conditions.

i. `((A > B) || (A < C)) && ((A != D) || (A == E))`

```
MOV     AX, A
CMP     AX, B
SETG    BL
CMP     AX, C
SETL    BH
OR      BL, BH
CMP     AX, D
SETNE   BH
CMP     AX, E
SETE    DL
OR      BH, DL
AND     BL, BH
```

ii. `((A – B > 0) && !C)`

```
MOV     AX, A
SUB     AX, B
SETG    BL          → Note that you don't have to explicitly
                       compare AX to 0 (although you can)—if
                       an operation that sets the 80386 flags
                       generates a positive result, the
                       condition "G" (greater than) is true
MOV     AX, C
CMP     AX, 0       → You do have to explicitly compare C to 0
SETE    BH             (condition !C is the same as (C == 0))
                       because the MOV operation does not set
                       the flags
AND     BL, BH
```

iii. `((B >= A + C) || (D <= C + A))`

```
MOV     AX, A
ADD     AX, C
CMP     B, AX
SETGE   BL
CMP     D, AX      → AX == A + C == C + A
SETLE   BH
OR      BL, BH
```

3. Assume CS = 1010H, IP = 1A00, and EBX = 20AAFE00. What is the starting address of each subroutine accessed by the CALL instructions below? (In other words, what is the target address of the CALL?)

i. CALL 0100H

Solution: If the target address is a 16-bit immediate, as shown here, that value is added to IP to generate the new address.

→ ***IP = 1A00 + 0100H = 1B00H***
→ ***CS is unchanged = 1010H***
→ ***Target address is CS:IP = 1010H:1B00H***

If you assume the processor is in real mode (which is usually a safe assumption), then the physical target address is 10100+1B00 = 11C00H

ii. CALL FFF0H

Solution: The target address is again a 16-bit immediate to be added to IP. Note that this offset is negative—this CALL goes to a lower address than the instruction that calls it.

→ ***IP = 1A00 + FFF0H = 19F0H***
→ ***CS is unchanged = 1010H***
→ ***Target address is CS:IP = 1010H:19F0H***

In real mode, the physical target address is 10100+19F0 = 11AF0H

iii. CALL 411ABE00

Solution: With a 32-bit immediate as the target, both CS and IP are overwritten, with the upper 16 bits of the immediate going to CS and the lower 16 bits going to IP.

→ ***IP = BE00H***
→ ***CS = 411AH***
→ ***Target address is CS:IP = 411AH:BE00H***

In real mode, the physical target address is 411A0+BE00 = 4CFA0H

iv. CALL BX

Solution: With a 16-bit register as the target, IP is overwritten by the register value.

→ ***IP = BX = FE00H***
→ ***CS is unchanged = 1010H***
→ ***Target address is CS:IP = 1010H:FE00H***

In real mode, the physical target address is 10100+FE00 = 1FF00H

v. CALL EBX

Solution: With a 32-bit register as the target, both CS and IP are overwritten, with the upper 16 bits of the register going to CS and the lower 16 bits going to IP.

→ ***IP = lower 16 bits of EBX = FE00H***
→ ***CS = upper 16 bits of EBX = 20AAH***
→ ***Target address is CS:IP = 20AAH:FE00H***

In real mode, the physical target address is 20AA0 + FE00 = 308A0H

4. Assume the 80386 is running in protected mode with the state given below (all values in hex); note that each memory location shown contains a descriptor about a particular segment:

GDTR = 00200000001F
LDTR = 000B

DS = 0017
SS = 0018
ESI = 00001000
EBX = 0001120

Memory	Address

Base = 030010F0
Limit = 020F
	00200000

Base = 00200020
Limit = 0017
	00200008

Base = 00200038
Limit = 0010
	00200010

Base = 1200C000
Limit = FFFF
	00200018

Base = 12340000
Limit = 00FF
	00200020

Memory	Address

Base = 01000010
Limit = 1127
	00200028

Base = 03170200
Limit = 03F7
	00200030

Base = 1A000000
Limit = 01FF
	00200038

Base = 06B01000
Limit = 0F07
	00200040

Base = 05000120
Limit = 000F
	00200048

a. What is the base address and limit of the global descriptor table? How many descriptors does this table contain?

Solution: The base address and limit of the GDT are stored in the GDTR—the upper 4 bytes contain the base address (***00200000H***); the lower 2 bytes contain the limit (***001FH***).

To determine the number of descriptors, recall that:

- Each descriptor uses 8 bytes
- The size of the table, in bytes, is (limit + 1) = 001FH + 1 = 0020H = 32 bytes

Therefore, this table contains 32 / 8 = ***4 descriptors***

b. What is the base address and limit of the current local descriptor table? How many descriptors does this table contain?

Solution: The base address and limit of the current LDT are stored in the LDT cache, which must be loaded from the appropriate descriptor in the GDT. The LDTR is a selector that points to the correct descriptor. Recall that, in a selector:

- The lowest 2 bits give the requested priority level
- The next bit (table indicator) indicates either global (0) or local (1) memory access
- The upper 13 bits index into the appropriate descriptor table to choose a descriptor.

LDTR = 000BH = 0000 0000 0000 1011_2
$\rightarrow$ Priority = 11_2, table indicator = 0, index = 0000 0000 0000 1_2 = 1
$\rightarrow$ GDT descriptor 1 (the second descriptor in the GDT) describes current LDT

Therefore, the ***LDT base address = 00200020H,*** its ***limit = 0017H,*** and the number of descriptors = (0017H+1) / 8 = 0018H / 8 = 24 / 8 ***= 3 descriptors.***

c. What are the starting and ending addresses for the current data and stack segments?

Solution: In protected mode, the segment registers are selectors pointing either to the GDT or current LDT, as shown in (b). Therefore, the starting (base) and ending (base + limit) addresses for each segment can be determined after finding the right descriptor.

DS = 0017H = 0000 0000 0001 0111_2

→ Priority = 11_2, table indicator = 1, index = 0000 0000 0001 0 = 2
→ Descriptor #2 (3rd descriptor) in LDT describes data segment

→ ***DS base address = 03170200H, ending address = 03170200 + 03F7 = 031705F7H***

SS = 0018H = 0000 0000 0001 1000_2

→ Priority = 00_2, table indicator = 0, index = 0000 0000 0001 1 = 3
→ Descriptor #3 (4th descriptor) in GDT describes stack segment

→ ***SS base address = 1200C000H, ending address = 1200C000 + FFFF = 1201BFFFH***

d. What address is accessed by each of the following instructions?

Recall that protected mode addresses are calculated by adding the base address of the requested segment to the effective address calculated from the instruction. Part (c) of this problem helped you determine the starting address of each segment used.

i. `MOV AX, [0100H]`

Solution: Address = DS:0100H = 03170200H + 0100H = ***03170300H***

ii. `ADD DX, [SI]`

Solution: Address = DS:SI = DS:1000H = 03170200H + 1000H = ***03171200H***

iii. `MOV AX, SS:[SI+EF00]`

Solution: Address = SS:SI+EF00 = SS:1000H+EF00H
= 1200C000H + 1000H + EF00H = ***1201BF00H***

iv. `SUB SS:[A200], CX`

Solution: Address = SS:A200 = 1200C000H + A200H = ***12016200H***

v. `MOV DX, [BX+SI]`

Solution: Address = DS:BX+SI = DS:1120H+1000H
= 03170200H + 1120H + 1000H = ***03172320H***

vi. `MOV CX, [BX+SI+1EH]`

Solution: Address = DS:BX+SI+1EH = DS:1120H+1000H +1EH
= 03170200H + 1120H + 1000H + 1EH = ***0317233EH***