EECE.3170: Microprocessor Systems Design I

Spring 2016

Exam 2 Solution

1. (16 points, 4 points per part) ***Multiple choice***

For each of the multiple choice questions below, clearly indicate your response by circling or underlining the single choice you think best answers the question.

Please note that all of the multiple choice questions deal with PIC 16F1829 instructions.

a. Which of the following instructions can always be used to complement the working register, W? (Note: remember that a complement operation simply flips all the bits of a register—it is not the same as negating a register.)

i. `comf x, W`

ii. `addlw -1`

iii. `sublw 0`

iv. ***`xorlw 0xFF`***

v. `iorwf x, W`

1 (continued)

b. Which of the following code snippets will jump to the label L if `x = 0x01`?

A.
```
btfss     x, 0
goto      L
```

B.
```
btfsc     x, 7
goto      L
```

C.
```
decfsz    x, F
goto      L
```

D.
```
incfsz    x, F
goto      L
```

i. Only A

ii. ***Only D*** *(all other choices skip the goto instruction)*

iii. A and B

iv. B and C

v. A, B, and C

1 (continued)

c. Which of the following instructions will set the carry bit (C) to 1 if the file register `x` is equal to 0xF0, the working register is equal to 0x20, and the carry bit is initially 0?

A. `subwf x, F` ***C = 1 because x > W → no borrow required***

B. `lslf x, F` ***C = 1 because MSB shifted into carry***

C. `rrf x, F` ***C = 0 because LSB rotated into carry***

D. `addwf x, F` ***C = 1 because 0xF0 + 0x20 = 0x110 → only 8 bits in sum, so underlined bit is carry***

i. A and B

ii. B and C

iii. A, B, and C

iv. A, B, and D

v. A, B, C, and D

d. Which of the following instructions has the same effect as rotating a file register, `x`, by four bits, without including the carry?

i. `rrf x, F`

ii. `rlf x, F`

iii. `lslf x, F`

iv. `asrf x, F`

v. `swapf x, F`

2. (16 points) ***Reading PIC assembly***
Show the result of each PIC 16F1829 instruction in the sequences below. Be sure to show the state of the carry (C) bit for any shift or rotate operations. You may assume C is initially 0.

```
a. cblock 0x70
     x
   endc

   clrf    x         x = 0x00

   comf    x, W      W = ~x = ~0x00 = 0xFF

   sublw   0x10      W = 0x10 - W = 0x10 - 0xFF = 0x11, C = 0

   incf    x, F      x = x + 1 = 0x00 + 1 = 0x01

   lslf    x, F      x = x << 1 = 0x01 << 1 = 0000 0001 << 1

                       = 0000 0010 = 0x02

                     C = bit shifted out = 0

   iorwf   x, F      x = x OR W = 0x02 OR 0x11 = 0x13

   xorlw   0x3C      W = W XOR 0x3C = 0x11 XOR 0x3C

                       = 0001 0001 XOR 0011 1100

                       = 0010 1101 = 0x2D

   addwf   x, W      x = x + W = 0x13 + 0x2D = 0x40, C = 0
```

3. (28 points) ***Subroutines; HLL → assembly***

The following questions (parts a-c) deal with the register and memory contents shown below. Note that:

- These values represent the state of some registers and memory locations immediately after the stack frame has been set up for the current function.
- The entire stack frame for the current function is shown, but there may be some additional data stored in the given address range—do not assume that the values shown in memory represent only the contents of the current stack frame.
- For parts a-c of this problem, you can assume that the stack frame for the current function starts at address 0x12580020.

EAX: 0x0000ABBA
EBX: 0x00001400
ECX: 0x09090909
EDX: 0xFF000000
ESI: 0x11340550
EDI: 0x11340590
ESP: 0x12580008
EBP: 0x12580014

Address	
0x12580000	0x00000005
0x12580004	0xCAE11600
0x12580008	0x09090909
0x1258000C	0x00001400
0x12580010	0x00000000
0x12580014	0x12580040
0x12580018	0x31700050
0x1258001C	0xFF000000
0x12580020	0x0000ABBA

a. (5 points) Assuming each argument uses 4 bytes, how many arguments does this function take? Explain your answer.

Solution: *To solve this problem, consider that (1) the bottom of the stack frame is 0x12580020, and (2) EBP is equal to 0x12580014. That means address 0x12580014 holds the saved base pointer, address 0x12580018 holds the function's return address, and everything at higher addresses within the stack frame represents the function arguments. Since there are 8 bytes remaining in the stack frame, this function takes two arguments.*

b. (4 points) Can you determine how many bytes of data the function called before the current function uses for local variables and saved registers? If so, explain how many bytes that function uses; if not, explain why not.

Solution: *Recall that local variables and saved registers are stored above (i.e., at lower addresses than) the saved base pointer in a stack frame. Therefore, all data between the previous frame's base pointer and the start of the current frame is used for local variables and saved registers. Since the previous function's base pointer is saved in the current frame, we can figure out how much space that function uses for local variables and saved data.*

The saved base pointer is 0x12580040; the start of the current stack frame is 0x12580020, but we need to account for the fact that 4 bytes of data are stored at that address. The top of the previous function's stack frame is therefore 0x12580024, giving 0x12580040 – 0x12580024 = 0x1C = 28 bytes of data used for local variables and saved registers in the previous stack frame.

3 (continued)

c. (4 points) If we assume that the function uses the stack to save every register it overwrites, what registers does this function overwrite? Explain your answer.

Solution: *The top of the current stack frame is ESP = 0x12580008. Saved registers are saved at the top of the stack. So, starting at that address, we see two values that match current registers: ECX (0x09090909) and EBX (0x00001400).*

d. (15 points) A partially completed x86 function is written below. Complete the function by writing the appropriate instructions in the blank spaces provided. The comments next to each blank or instruction describe the purpose of that instruction. Assume that the function takes two arguments (`v1` and `v2`, in that order) and contains a single local integer variable, `x`.

```
f PROC                                  ; Start of function f
  push   ebp                            ; Save ebp
  mov    ebp, esp                       ; Copy ebp to esp

  sub    esp, 4                         ; Create space on stack for x

  mov    ebx, DWORD PTR 8[ebp]          ; ebx = v1


  add    ebx, DWORD PTR 12[ebp]         ; ebx = v1 + v2



  mov    DWORD PTR -4[ebp], ebx         ; x = ebx = v1 + v2 (copy ebx
                                        ;    to memory location for x)
  sar    ebx, 2                         ; ebx = ebx >> 2 = x >> 2


  add    ebx, DWORD PTR -4[ebp]         ; ebx = ebx + x = (x >> 2) + x


  mov    esp, ebp                       ; Clear space allocated for
                                        ;     local variable
  pop    ebp                            ; Restore ebp

  ret                                   ; Return from subroutine
f ENDP
```

4. (40 points) ***Conditional instructions***

For each part of this problem, write a short x86 code sequence that performs the specified operation. **CHOOSE ANY TWO OF THE THREE PARTS** and fill in the space provided with appropriate code. **You may complete all three parts for up to 10 points of extra credit, but must clearly indicate which part is the extra one—I will assume it is part (c) if you mark none of them.**

Note also that your solutions to this question will be short sequences of code, not subroutines. **You do not have to write any code to deal with the stack when solving these problems.**

a. Implement the following conditional statement. You may assume that "`X`" and "`Y`" refer to 16-bit variables stored in memory, which can be directly accessed using those names (for example, `MOV AX, X` would move the contents of variable "`X`" to the register `AX`). Your solution should not modify AX or BX.

```
if (X < 10) {
   Y = X + AX;
}
else if (Y > BX) {
   X = Y – AX;
}
else {
   X = Y * 4;
   Y = X / 4;
}
```

Solution: *Other solutions may be valid. Note that, in the else case, Y doesn't actually change, so instructions that modify it are really unnecessary.*

```
     CMP  X, 10
     JGE  L1            ; Jump to second comparison if X >= 10
     MOV  DX, X         ; if case: start by setting DX = X
     MOV  Y, DX         ; Y = DX = X
     ADD  Y, AX         ; Y = Y + AX = X + AX
     JMP  end           ; Skip else if, else cases
L1:  CMP  Y, BX
     JLE  L2            ; Jump to else case if Y <= BX
     MOV  DX, Y         ; else if case: set DX = Y
     MOV  X, DX         ; X = DX = Y
     SUB  X, AX         ; X = X – AX = Y – AX
     JMP  end           ; Skip else case
L2:  MOV  DX, Y         ; else case: set DX = Y
     SHL  DX, 2         ; DX = DX << 2 = Y << 2 = Y * 4
     MOV  X, DX         ; X = DX = Y * 4
     SHR  DX, 2         ; DX = DX >> 2 = X >> 2 = X / 4
     MOV  Y, DX         ; Y = DX = X / 4
end:                    ; Target to skip else if, else cases
```

4 (continued)

b. Implement the following loop. As in part (a), assume "`X`" and "`Y`" are 16-bit variables in memory that can be accessed by name. Assume that `ARR` is an array of 32-bit values, and that the loop does not go outside the bounds of the array. The starting address of this array is in the register `SI` when the loop starts—you can use that register to help you access values within the array. Your solution should not modify X, Y, or EAX.

```
for (i = X; i < Y; i = i + 3) {
   ARR[i+1] = ARR[i] + ARR[i+2];
   ARR[i] = ARR[i+2] - EAX;
}
```

Solution: *Other solutions may be correct.*

```
     MOV  ECX, X               ; Let ECX = i; initialize i = X
L:   LEA  EDX, [SI+4*ECX]      ; EDX = address of ARR[i]
     MOV  EBX, [EDX]           ; EBX = ARR[i]
     ADD  EBX, [EDX+8]         ; EBX = ARR[i] + ARR[i+2]
     MOV  [EDX+4], EBX         ; ARR[i+1] = ARR[i] + ARR[i+2]
     MOV  EBX, [EDX+8]         ; EBX = ARR[i+2]
     SUB  EBX, EAX             ; EBX = ARR[i+2] - EAX
     MOV  [EDX], EBX           ; ARR[i] = ARR[i+2] - EAX
     ADD  ECX, 3               ; i = i + 3
     CMP  ECX, Y               ; Compare i to Y and return to
     JL   L                    ;   start of loop if i < Y
```

4 (continued)

c. Implement the following loop. As in part (a), assume "`X`", "`Y`", and "`Z`" are 16-bit variables in memory that can be accessed by name. Recall that a while loop is a more general type of loop than the for loop seen in part (b)—a while loop simply repeats the loop body as long as the condition tested at the beginning of the loop is true. Your solution should not modify AX or BX.

```
while ((X < AX) || (Y > BX)) {
   X = X - Z;
   Y = Y + X;
}
```

Solution: *Other solutions may be correct.*

```
ST:  CMP  X, AX                ; If X < AX, goto loop body (LB),
     JL   LB                   ;   since only part of condition
                               ;   must be true to stay in loop
     CMP  Y, BX                ; If Y <= BX, exit loop
     JLE  DONE
LB:  MOV  DX, Z                ; DX = Z
     SUB  X, DX                ; X = X - DX = X - Z
     MOV  DX, X                ; DX = X
     ADD  Y, DX                ; Y = Y + DX = Y + X
     JMP  ST                   ; Return to conditional tests at
                               ;   start of loop
DONE:                          ; Label for loop exit
```