EECE.3170: Microprocessor Systems Design I

Fall 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. If a file register, `x`, holds the value `0x03`, which of the following instruction sequences will set `x = 0xFD`?

i. `comf x, F`

ii. `comf x, W`

iii. `comf x, F`
`incf x, W`

iv. ***`comf x, F`***
`incf x, F`

v. None of the above

1 (continued)

b. Which of the following code snippets <u>will not</u> jump to the label L if `x = 0xF0`?

A.
```
btfss    x, 4
goto     L
```

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

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

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

i. <u>Only A</u>

ii. Only B

iii. A and C

iv. B and D

v. A, C, and D

1 (continued)

c. If a file register, `x`, is equal to `0x37`, and the working register, `W`, is equal to `0x91`, what values do those two registers hold after executing the instruction `swapf x, W`?

i. `x = 0x91, W = 0x37`

ii. `x = 0x73, W = 0x91`

iii. `x = 0x19, W = 0x91`

iv. `x = 0x37, W = 0x19`

v. ***`x = 0x37, W = 0x73`***

d. If a file register, `z`, is equal to `0x03`, and the working register, `W`, is equal to `0x05`, which of the following instructions will set `W = 0x02`?

i. ***`sublw   0x07`***

ii. `subwf   z, W`

iii. `addlw   0x02`

iv. `decf    z, F`

v. None of the above

2. (16 points) ***<u>Reading PIC assembly</u>***

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.

```
cblock 0x7F
   z
endc
```

`clrf`	`z`	***z = <u>0x00</u>***
`decf`	`z, W`	***W = z – 1 = 0x00 – 1 = <u>0xFF</u>***
`incf`	`z, F`	***z = z + 1 = 0x00 + 1 = 0x01***
`andlw`	`0xC3`	***W = W AND 0xC3 = 0xFF AND 0xC3 = <u>0xC3</u>***
`iorwf`	`z, F`	***z = z OR W = 0x01 OR 0xC3 = <u>0xC3</u>***
`asrf`	`z, F`	***z = z >> 1 (keep sign) = 0xC3 >> 1 = <u>0xE1</u>*** ***C = bit shifted out = <u>1</u>***
`btfsc`	`z, 7`	***If bit 7 of z = 0, skip next instruction*** ***→ z = 0xE1 = <u>1</u>110 0001*** $_2$ ***→ <u>do not skip</u>***
`bsf`	`z, 1`	***Set bit 1 of z = 1 → z = 1110 00<u>1</u>1 = <u>0xE3</u>***

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

a. (8 points) The code below represents the body of a function (the code after the function prologue and before the function epilogue). Given this function body, explain what registers should be saved on the stack in the function prologue and why they should be saved:

```
mov  ecx, [ebp+8]
mov  esi, [ebp+12]
lea  esi, [esi+4*ecx]
imul DWORD PTR [esi]
```

Solution: *In the calling convention discussed in class, a function should save any registers it overwrites with the exception of `eax`, `ecx`, and `edx`. The instructions in this function body overwrite `ecx` (in the first `mov`), `esi` (in the second `mov` and `lea` instructions), and `eax/edx` (in the `imul` instruction). Therefore, `esi` should be saved.*

Also, as many of you noted, ebp will be saved on the stack in the function prologue.

b. (6 points) Explain why function arguments and variables are typically accessed using base pointer-relative addressing (for example, `-4[ebp]` or `[ebp+4]`) as opposed to stack pointer-relative addressing.

Solution: *The base pointer provides a fixed point of reference within any stack frame—the argument list for a function always starts 8 bytes below the base pointer, while the first local variable is directly above the base pointer (4 bytes above, if you assume all variables are integers as we've done in our examples).*

3 (continued)

c. (6 points) What is the minimum size for a stack frame in an x86 subroutine using the calling convention we discussed in class? (Recall that a calling convention determines how arguments are passed, values are returned, local variables are declared, and registers are saved in a function.) Explain your answer for full credit.

Solution: *It's possible for a function to take no arguments, use no local variables, and save no registers, meaning that the only things that will absolutely be saved on the stack are (1) the function return address, and (2) the base pointer from the previous stack frame. Each of those registers takes up 4 bytes, making the minimum stack frame size 8 bytes.*

If you want to make the argument that saving the base pointer isn't necessary if you're not using arguments or local variables (which is something that aggressively optimizing compilers do), then the minimum stack frame would only hold the return address, which is just 4 bytes.

d. (8 points) You are writing a function that takes four signed 32-bit integer arguments. The function should return the average of the four values. Write only the body of the function—do not write any code that adds data to or removes data from the stack frame.

For full credit, write the function body without using any local variables or any registers that would have to be saved on the stack in the function prologue.

Solution: *In order to get full credit, all work must be done using eax, which will hold the function return value at the end. To average four values, add them up and divide by 4, using a shift instruction to perform the division. As noted during the exam, you don't have to worry about any remainder, thus freeing you to use a shift instruction rather than a divide:*

```
mov     eax, 8[ebp]     ; eax = first argument
add     eax, 12[ebp]    ; eax = sum of first 2 arguments
add     eax, 16[ebp]    ; eax = sum of first 3 arguments
add     eax, 20[ebp]    ; eax = sum of all 4 arguments
sar     eax, 2          ; eax = sum >> 2 = sum / 4 = average
```

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 loop. You may assume "X" and "Y" are 16-bit variables in memory that can be accessed by name (for example, `MOV AX, X` would move the contents of variable "X" to the register AX). 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)

b. Implement the following conditional statement. As in part (a), assume that "`X`", "`Y`", and "`Z`" 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`). <u>Your solution should not modify AX or BX.</u>

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

Solution: *Other solutions may be correct. I've tried to directly reflect the structure of the code above in my solution (primarily to make grading easier!). However, please note that a more optimal solution would do the following when handling the inner if/else if statement:*

- *Check if Z == Y, and exit the if statement if that condition is true.*
- *Before checking if Z > Y or Z < Y, copy X to Y (since Y will equal some combination of X and AX)*
- *Then, test Z > Y or Z < Y to determine how Y and X will be changed.*

```
     CMP  AX, X               ; If AX < X, continue to body of
     JGE  DONE                ;   statement; exit otherwise
                              ;   (if AX >= X)
     MOV  Z, AX               ; Z = AX
     ADD  Z, BX               ; Z = Z + BX = AX + BX
     MOV  CX, Y               ; CX = Y (so we can compare Z & Y)
     CMP  Z, CX               ; Compare Z to Y (Z to CX)
     JG   L1                  ; If Z > Y, go to inner if case
     JL   L2                  ; If Z < Y, go to else if case
     JMP  DONE                ; Otherwise (Z==Y), exit statement
L1:  MOV  CX, X               ; CX = X
     MOV  Y, CX               ; Y = CX = X
     SUB  Y, AX               ; Y = X - AX
     ADD  X, BX               ; X = X + BX
     JMP  DONE                ; Exit statement
L2:  MOV  CX, X               ; CX = X
     MOV  Y, CX               ; Y = CX = X
     ADD  Y, AX               ; Y = X + AX
     SUB  X, BX               ; X = X - BX
DONE:                         ; Label for exiting statement
```

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 (a)—a while loop simply repeats the loop body as long as the condition tested at the beginning of the loop is true. <u>Your solution should not modify AX or BX.</u>

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

<u>Solution:</u> *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
```