16.317: Microprocessor Systems Design I

Spring 2015

Homework 2 Solution

1. (45 points) Assume the state of the x86 registers and memory are:

EAX: 00000010H
EBX: 00000020H
ECX: 00000030H
EDX: 00000040H
CF: 1
ESI: 00020100H
EDI: 00020100H

Address				
20100H	10	00	08	00
20104H	10	10	FF	FF
20108H	08	00	19	91
2010CH	20	40	60	80
20110H	02	00	AB	0F
20114H	30	00	11	55
20118H	40	00	7C	EE
2011CH	FF	00	42	D2
20120H	30	00	30	90

What is the result produced in the destination operand by each of the instructions listed below? Assume that the instructions execute in sequence.

ADD AX, 00FFH

Solution: AX = AX + 00FFH = 0010H + 00FFH = **010FH, CF = 0**

ADC CX, AX

Solution: CX = CX + AX + CF = 0030H + 010FH + 0 = **013FH, CF = 0**

INC BYTE PTR [20100H]

Solution: Increment byte at address 20100

→ (byte at 20100h) = 10 + 1 = **11H**

SUB DL, BL

Solution: DL = DL – BL = 40H – 20H = **20H, CF = 0**

SBB DL, [20114H]

Solution: DL = DL – (byte at 20114h) – CF = 20H – 30H – 0 = **F0H**

1 (cont.)

DEC BYTE PTR [EDI+EBX]

Solution: Decrement byte at address EDI+EBX = 00020100h+00000020h = 00020120h

→ (byte at 20120h) = 30H – 1 = **2FH**

NEG BYTE PTR [EDI+0018H]

Solution: Negate byte at address 00020100h + 0018h = 00020118h

→ (byte at 20118h) = -40H = -0100 0000_2 = 1011 1111_2 + 1 = 1100 0000_2 = **C0H**

MUL DX

Solution: (DX,AX) = DX * AX (unsigned multiplication of 16-bit values)

→ (DX,AX) = 00F0 * 010F = 240 * 271 = 65040 = 0000FE10H

→ **DX = upper 16 bits of result = 0000H; AX = lower 16 bits of result = FE10H**

IMUL BYTE PTR [ESI+0006h]

Solution: AX = AL * byte at address ESI+0006h (signed multiplication of 8-bit values)

→ Address = 00020100h + 0006h = 20106H; byte = FF

→AX = 10H * FFH = 16 * -1 (must account for signs!) = -16

→-16 = -(10H) = -(0000 0000 0001 0000_2) = 1111 1111 1110 1111_2 + 1

= 1111 1111 1111 0000_2 = **FFF0H**

DIV BYTE PTR [ESI+0008h]

Solution: Divide AX by byte at address ESI+0008h; AL = quotient, AH = remainder (unsigned integer division of 8-bit values)

→ Address = 00020100h + 0008h = 20108h; byte = 08H

→ Dividing FFF0 / 08H = 65520 / 8 = 8190 R0 = 1FFEH R0 (keep only lowest byte)

→ **AL = FEH, AH = 00H**

IDIV BYTE PTR[ESI+0010H]

Solution: Divide AX by byte at address ESI+0010h; AL = quotient, AH = remainder (signed integer division of 8-bit values)

→ Address = 00020100 + 0010h = 20110h; byte = 02

→ Dividing 00FEH / 02H = 254 / 2 = 127 R0

→ **AL = 7FH, AH = 00H**

2. (25 points) Assume the state of the 80386DX's registers and memory are:

EAX: 00005555H
EBX: 00045010H
ECX: 00000010H
EDX: 0000AAAAH
ESI: 000000F2H
EDI: 00000200H

Address				
45100H	0F	F0	00	FF
	…			
45200H	30	00	19	91
	…			
45210H	AA	AA	AB	0F
	…			
45220H	55	55	7C	EE
	…			
45300H	AA	55	30	90

What is the result produced in the destination operand by each of the instructions listed below? Assume that the instructions execute in sequence.

AND BYTE PTR [45300H], 0FH

Solution: Byte at address 45300h = Byte at 45300H AND 0FH

→ (45300h) = AAH AND 0FH = **0AH**

SAR DX, 8

Solution: DX = DX >> 8 (arithmetic shift)

→ DX = AAAAH >> 8 = **FFAAH**; CF = last bit shifted out = **1**

OR [EBX+EDI], AX

Solution: Word at address EBX+EDI = Word at EBX+EDI OR AX

→ EBX+EDI = 00045010h+00000200h = 45210H

→ (45210H) = AAAAH OR 5555H = **FFFFH**

SHL AX, 2

Solution: AX = AX << 2

→ AX = 5555H << 2 bits = 0101 0101 0101 0101 << 2

→ AX = 0101 0101 0101 0100 = **5554H**, CF = last bit rotated out = **1**

2 (cont.)

XOR AX, [ESI+EBX]

Solution: AX = AX XOR (Word at address ESI+EBX)

→ ESI+EBX = 000000F2h + 00045010h = 45102H

→ AX = 5554H XOR FF00H = **AA54H**

NOT BYTE PTR [45300H]

Solution: Flip all bits of byte at 45300h

→ (45300H) = NOT 0AH = **F5H**

SHR AX, 4

Solution: AX = AX >> 4 (logical shift)

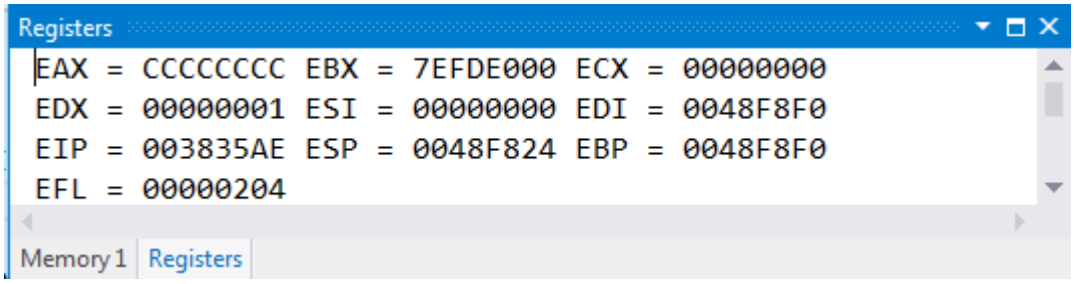
→ AX = AA54H >> 4 = 1010 1010 0101 0100 >> 4

→ AX = 0000 1010 1010 0101 = **0AA5H**, CF = last bit rotated out = **0**

1. (30 points) This exercise will give you some familiarity with debugging assembly programs in Visual Studio. See the actual assignment for a full description of the exercise.

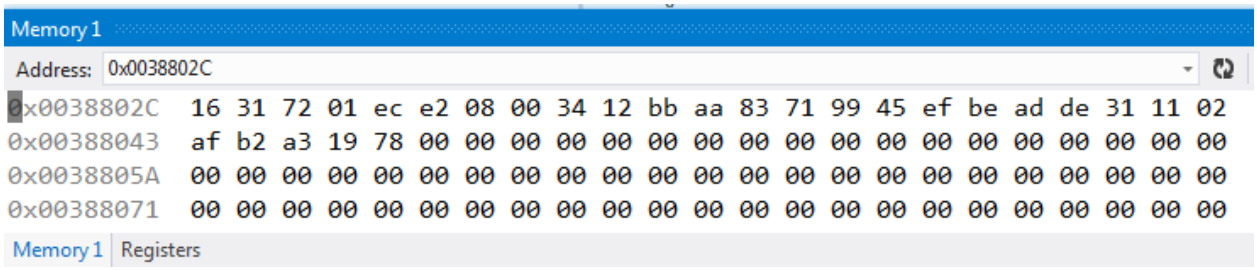
a. (3 points) Generate and submit a screenshot of the register contents at the start of the program.

Solution: A screenshot of my registers window, which I resized to better show the register contents, is shown below. Yours should show the same registers, but may have different values.



b. (4 points) Type the array name "`blist`" into the address field to see the contents of memory starting at the beginning of that array. Generate and submit a screenshot of those memory contents. Describe the data shown in that screenshot and how it corresponds to any of the arrays declared at the start of the program (`blist`, `wlist`, and `dlist`).

Solution: My screenshot is below. You may have different addresses, but should see the same data *(unless you're using Visual C++ Express 2010 and viewing `blist` in the Watch window).*



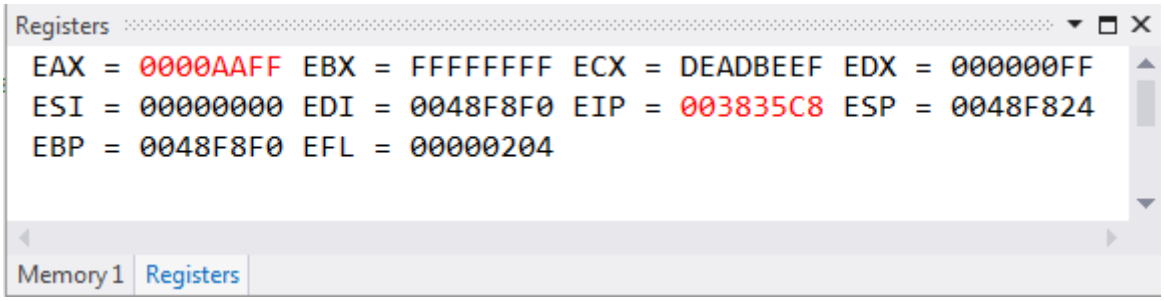
Note that, in this screenshot, the contents of all three arrays are shown. The first 7 bytes show the contents of `blist` in order. After the byte `00` (which is there to ensure the next array is aligned), the next 8 bytes (starting with `34`) show the contents of `wlist` in order. Since that array is composed of four words, the individual bytes in each word are shown in little endian format (as we'd expect), with the least significant byte listed first. The last 12 bytes (starting with `ef`) show the contents of the double word array `dlist`, again in little endian format.

c. *(4 points) To execute the first four instructions, press the "Continue" button (which has replaced the "Local Windows Debugger" button) or use the "Continue" option from the DEBUG window. Look at the Registers window and describe how the register contents have changed after running those first four instructions.*

Solution: Those four instructions change `EAX`, `EBX`, `ECX`, and `EDX`, assigning the values 0, -1 (which is `0xFFFFFFFF` in hexadecimal), `0xDEADBEEF`, and `0x000000FF`, respectively. You may also notice that the instruction pointer `EIP` has also changed to reflect that the address of the current instruction has changed after executing those instructions.

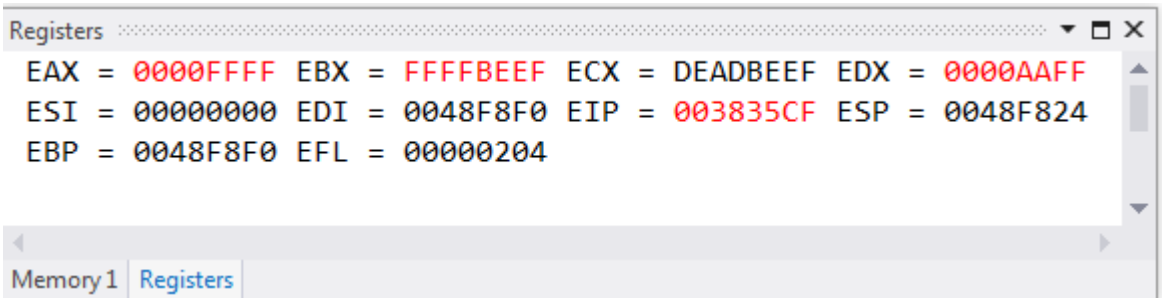
d. *(3 points) Use the "Continue" command to execute the next two instructions and run to the third breakpoint. Explain the changes these two instructions generate, and submit another screenshot of the register state at this point.*

Solution: The next two instructions change parts of register `EAX`, which was set to 0 prior to those instructions. The first (`mov ax, 0xFFFF`) changes the lowest 16 bits of `EAX` to `0xFFFF`, leaving the upper 16 bits unchanged. The second (`mov ah, 0xAA`) changes the second least significant byte of `EAX` to the value `0xAA`, giving `EAX` the value `0x0000AAFF`. These changes are reflected in the screenshot below.



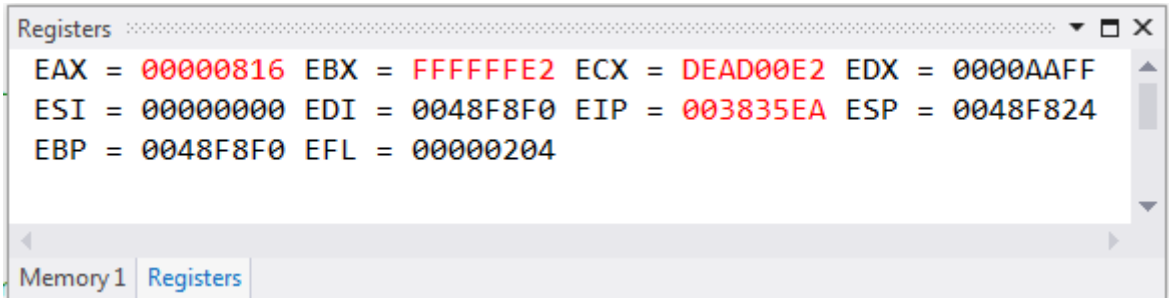
e. *(4 points) Set a breakpoint on the instruction `mov al, [blist]`. Use the "Continue" command to execute the next three instructions, describe their operation, and generate a screenshot showing the register state after these instructions complete.*

Solution: Each of the next three instructions copies a value from one register to another—`EDX` gets a copy of the value in `EAX` (`0x0000AAFF`), `BX` (the lowest 16 bits of `EBX`) gets a copy of `CX` (`0xBEEF`), and `AH` (the second least significant byte of `EAX`, originally holding `0xAA`) gets a copy of `AL` (`0xFF`). These changes are shown in the screenshot below.



f. (4 points) Set a breakpoint on the instruction `mov bx, [wlist]`*. Use the "Continue" command to execute the next four instructions, describe their operation, and generate a screenshot showing the register state after these instructions complete. Be sure to discuss the differences between the* `movsx` *and* `movzx` *instructions used in this step.*

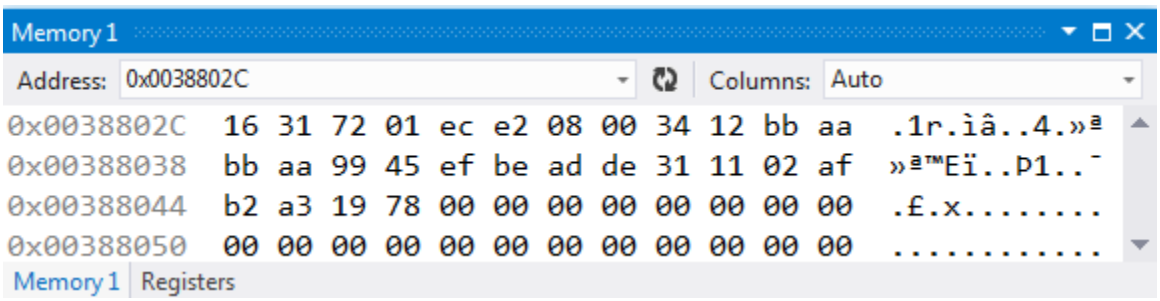
Solution: These instructions transfer data from the byte array `blist` to various registers. The first two instructions transfer the first byte (`0x16`) and last byte (`0x08`) of that array into `AL` and `AH`, respectively, thus changing the lowest 16 bits of `EAX`. The next two instructions take the same byte (`0xE2`) and transfer it to two different 16-bit registers. `BX` receives a sign-extended version of the byte, setting the lowest 16 bits of `EBX` to `0xFFE2`. `CX` receives a zero-extended version of the byte, setting the lowest 16 bits of `ECX` to `0x00E2`.



g. (4 points) Set a breakpoint on the instruction `mov esi, 0`*. Use the "Continue" command to execute the next three instructions. How are the memory accesses performed in these instructions different than those performed in part (f)? Generate a screenshot showing the memory state that has been changed within this sequence of instructions.*

Solution: These instructions transfer words of data to and from the array `wlist`, rather than a single byte for each instruction as in part (f). Note that, to move from one array element to the next, the program must go two bytes further into the array. Therefore, the address `[wlist+2]` accesses the second array element, while `[wlist+4]` accesses the third.

The first instruction is a simple data transfer from the first array element to `BX`. The second takes the second word in the array (`0xAABB`) and copies a sign-extended version into `ECX`. And the third takes that word and copies it into the third location in the array. That change is seen on the second line of memory shown below (starting with address `0x00388038`). The first two bytes of that line—the third element of `wlist`—are the same as the last two bytes of the previous line.



h. (4 points) The last group of instructions in this program comprises a loop, which will repeatedly execute the same instructions until the end condition is reached. To see the loop operation, set one breakpoint on the instruction `mov eax, [dlist + 4 * esi]`*, and another one on the curly brace* `}` *at the end of the program.*

Using "Continue" once will bring you to the start of the loop. Each time you use "Continue" after that will execute one iteration of the loop. When the program reaches the final breakpoint, the loop will be done.

How many iterations does the loop execute? What happens in each loop iteration? How is the memory address accessed in each loop iteration changed?

Solution: The loop executes three iterations. (You didn't have to explain this point, but the number of loop iterations is controlled by the register `ESI`, which starts at 0 and is incremented in every iteration. The program will return to the start of the loop as long as `ESI` is less than 3.)

In each iteration, a double word is copied from the array `dlist` into the register `EAX`. In addition to controlling the number of loop iterations, `ESI` controls which array element is accessed. The `mov` instruction that accesses memory uses scaled-index addressing to access each array element—since a double word is four bytes, the scaling factor is 4. So, although `ESI` only increases by one in each iteration, the memory address being used increases by 4.