ICS 51: Introductory Computer Organization **[Winter 2015]**

Lab#2 (lab2.zip)

(Due: Tuesday Feb. 17 - 11:55pm)

The primary goals of this lab is to teach you how to use the low-level x86 assembly instructions to:

1. Implement FOR and WHILE loop constructs
2. Access data stored on the memory

1. Implementing loop constructs in x86 assembly:

Program loops consist of three components: an optional initialization component, a loop termination test, and the body of the loop. The order with which these components are assembled can dramatically change the way the loop operates. Two permutations of these components appear over and over again. Because of their frequency, these loop structures are given special names in high-level languages: WHILE loops and FOR loops.

1.1. WHILE loop:

The most general loop is the WHILE loop which takes the following form in C/C++:

```
while ( condition )  {

     <<loop body>>

}
```

There are two important points to note about the WHILE loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of the position of the termination test, the body of the loop may never execute. If the termination condition is always true, the loop body will never execute.

The WHILE loop can be converted to x86 assembly using the following pseudo-code template :

```
BeginningOfWhile:
                if( condition ) {
                    <<loop body>>
                    jmp BeginningOfWhile
                }
```

```
     EndOfWhile:
```

Therefore, you can use the technique from earlier lab to convert IF statements to assembly language along with a single JMP instruction to produce a WHILE loop:

```
BeginninggOfWhile:
               BeginningOfIf:
                           test for negated condition
                           jcc EndOfIf
                              <<loop body>>
                           jmp BeginningOfWhile
                     EndOfIf:
     EndOfWhile:
```

For example:

```
while ( c > 0 ) {
     a = b + 1;
     c--;
}
```

can be converted to x86 assembly, using the following code:

```
               mov eax, a
               mov ebx, b
               mov ecx, c

BeginningOfWhile:

               cmp ecx, 0
               jle EndOfWhile
                   mov eax, ebx
                   inc eax
                   dex ecx
               jmp BeginningOfWhile

     EndOfWhile:

               mov a, ebx
               mov c, ecx
```

1.2. FOR loop:

What is a FOR Loop? In essence, it's a WHILE Loop with an initial state, a condition, and an iterative instruction. For instance, the following generic FOR Loop in C/C++:

```
for(initialization; condition; increment/decrement) {

     <<loop body>>

}
```

gets translated into the following pseudo-code WHILE loop:

```
initialization
while(condition) {

     <<loop body>>
     increment/decrement

}
```

2. x86 Memory Addressing Modes:

The memory addressing mode determines, for an instruction that accesses a memory location, how the address for the memory location is specified.

To summarize, the following diagram shows the possible ways an address could be specified:

$$\left[\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{Bmatrix}\right] + \left[\begin{Bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{Bmatrix} * \begin{Bmatrix} 1 \\ 2 \\ 4 \\ 8 \end{Bmatrix}\right] + [\text{displacement}]$$

Each square bracket in the above diagram indicates an optional part of the address specification. These parts (from left to right) are: A register used as a base address, a register used as an index, a width (or scale) value to multiply the register by, and an displacement (aka offset) which is an integer. The address is computed as the sum of: the base register, the index times the width, and the displacement. Here are couple of common modes with examples:

Mode	Example
Direct (or Absolute)	`mov eax, [0100] - mov ebx, [var] ≡ mov ebx, var`
Register indirect	`mov eax, [esi]`
Register indirect with offset	`mov eax, [ebp-8]`
Scaled-index with offset	`mov eax, [ebx*4 + 100]`
Base with scaled-index	`mov eax, [edx + ebx*4]`

Note that, When accessing memory, the operand will always be surrounded by square brackets ("[" and "]"). The square brackets mean that the operand points to a memory location and that data should be read from or written to that memory location. So in general, MOV XXX, [YYY] means: “copy anything that YYY points to and put it in the XXX”.

2.1. Direct (or Absolute)

The direct memory addressing mode is the most straightforward way to access memory. However, it is not always the most useful. This mode is used when you know the exact address of the memory you wish to access.

2.2. Register Indirect:

Register indirect addressing mode allows you to specify the address of the memory location that you wish to access in a register. This mode is especially useful when the address of a variable is passed to a procedure as a pointer argument.

2.3. Register indirect with offset:

This mode allows you to use a base register and add that to a constant displacement to access memory. It is possible to emulate this mode by adding the displacement to the base register using the add instruction, however using the based mode will save instruction space as well as CPU cycles.

2.4. Scaled-index with offset:

In this mode you specify an index register, a scale factor, and a displacement. This mode is extremely useful when accessing elements of an array when the element size is 1, 2, 4, or 8 bytes big. However, if the element size is not one of the scale sizes than you have to manually adjust the index according to the element size.

2.5. Base with scaled-index:

In this mode you specify a base, an index register, and a scale factor. This mode is particularly useful when accessing elements of an array when the element size is 1, 2, 4, or 8 bytes big. In this addressing mode, scale is a constant that represents the element size and could be 1, 2, 4, or 8.

3. Size Directives:

In general, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction in which it is referenced. For example, if we are loading a 32-bit register, the assembler could infer that the region of memory we were referring to was 4 bytes wide. For example:

```
mov eax, [ebp-8]
```

However, in some cases the size of a referred-to memory region is ambiguous. Consider the instruction:

```
mov [ebx], 2
```

Should this instruction move the value 2 into the single byte at address EBX? Perhaps it should move the 32-bit integer representation of 2 into the 4-bytes starting at address EBX. Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct. The size directives BYTE PTR, WORD PTR, DWORD PTR, and QWORD PTR serve this purpose, indicating sizes of 1, 2, 4, and 8 bytes respectively.

For example:

```
mov BYTE PTR [ebx], 2       ; Move 2 into the single byte at the address stored in EBX.

mov WORD PTR [ebx], 2       ; Move the 16-bit integer representation of 2 into the 2
                            bytes starting at the address in EBX.

mov DWORD PTR [ebx], 2      ; Move the 32-bit integer representation of 2 into the 4
                            bytes starting at the address in EBX.
```

4. Putting it all together:

Let's translate the following for-loop, which does an operation on an array, to x86 assembly:

```
int sum=0;
for (int i=0; i < 10; i++)
{
     sum += array[i];
}
```

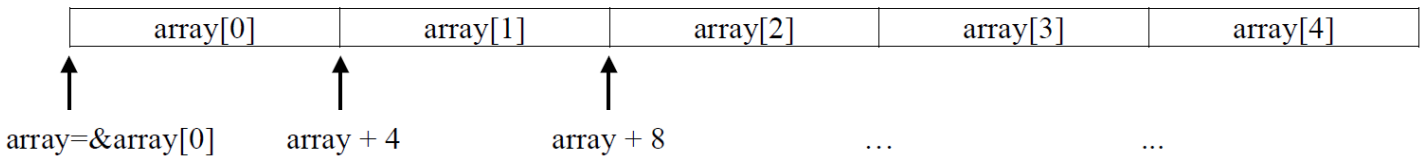
The equivalent x86 assembly code would be:

```
                mov    eax, 0   // sum
                mov    ecx, 0   // i

START_ARRAY_FOR:
                cmp    ecx, 10
                jg     END_ARRAY_FOR

                mov    edx, dword ptr array[ecx*4]
                add    eax, edx
                inc    ecx
                jmp    START_ARRAY_FOR

  END_ARRAY_FOR:
                mov    sum, eax
```

In the above assembly code:

```
mov    edx, dword ptr array[ecx*4]
```

uses “scaled-index with offset” addressing mode. Note that `array[ecx*4]` is the same as `[ecx*4 + array]`.
We’ve scaled the index by a factor of 4. This is because every integer is 4-byte, the memory is byte-addressable, and array elements are stored contiguously in the memory.



We could accomplish the same purpose by replacing this line with these two lines:

```
lea    edi, array;
mov    edx, dword ptr array[edi + ecx*4]
```

The `lea` instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address

is computed and placed into the register. This is useful for obtaining a pointer into a memory region. Then the second line (i.e., `mov edx, dword ptr array[edi + ecx*4])` uses "case with scaled-index" to read from memory.

5. Your task:

You are given an array of integer numbers. You need to implement two functions in x86 assembly to (1) find the maximum number and (2) sort the array ascendingly. The template code (lab2.c) and the tester (lab2-testing.c) are given to you. Download them from here. You need to make a new project and add these two files to it, like what you did for lab#1. However, remember you are only allowed to make modifications in the marked blocks of lab2.c. **For more details, read the comment sections of lab2.c and lab2-testing.c**.

VERY IMPORTANT:

Submit ONLY the completed lab2.c file
and
do NOT zip it!