KOM206 Laboratory (Dept. of Computer Science, IPB) Module 06 - Mar 18, 2015

The first-half review

Lecturers: ARD, SWJ *Lab tutors: VDE, WUL*

Objectives

1. able to utilize nested loops
2. able to make a sorting program
3. able to put together all topics in the first half

1 Theoretical Review

In the first half, we have learned the followings: data definitions, stack operations, arithmetic and boolean instructions, conditional and unconditional jumps, conditional and unconditional looping, and debugging. This lab session essentially focusses on putting together those subjects. Nevertheless, we have some additional topics as follows.

1.1 Local labels

A local label is a form of label with a dot prefix. Local labels are defined within the range of enclosing regular global labels. This would allow using the same local labels in multiple global labels. The local labels can be used for all labels inside its parent global label. We use "main.case0" outside of a global label named "main" to refer to the ".case0" label inside "main". Refer to task01 for a concrete use of such local labels.

1.2 Shift operations

Shifting is an excellent tool for extracting bit fields and for building values with bit fields. Additionally, it just happens that translating $N \times 2^M$ into binary becomes shift N by M places. If we are doing something that is not a power of 2 in binary, we have to go back to the old fashioned multiply and add. For example, $i \times 2 = i \ll 1$; $i \times 3 = (i \ll 1) + i$;$i \times 10 = (i \ll 3) + (i \ll 1)$.

There are 4 varieties of shift instructions: shift left (shl), shift arithmetic left (sal), shift right (shr), and shift arithmetic right (sar). The shl and sal left instructions are actually the same instruction. The sar instruction propagates the sign bit into the newly vacated positions on the left which preserves the sign of the number, while shr introduces 0 bits from the left.

There are also rotate left (rol) and rotate right (ror) instructions. These could be used to shift particular parts of a bit string into proper position for testing while preserving the bits. After rotating the proper number of bits in the opposite direction, the original bit string will be left in the register or memory location. Refer to listing 1 for some usage examples.

Listing 1: Usage examples of shift and rotate bit operations

```
1         segment .text
2         global  main
3 main:
4         mov     eax, 0x12345678
5         shr     eax, 8          ; I want bits 8-15
```

```
6       and     eax, 0xff       ; eax now holds 0x56
7       mov     eax, 0x12345678 ; I want to replace bits 8-15
8       mov     edx, 0xaa       ; edx holds replacement field
9       mov     ebx, 0xff       ; I need an 8 bit mask
10      shl     ebx, 8          ; Shift mask to align @ bit 8
11      not     ebx             ; ebx is the inverted mask
12      and     eax, ebx        ; Now bits 8-15 are all 0
13      shl     edx, 8          ; Shift the new bits to align
14      or      eax, edx        ; eax now has 0x1234aa78
15      xor     eax, eax
```

1.3 Single bit handling

To extract a bit field from a word, you first shift the word right until the right most bit of the field is in the least significant bit position (bit0). Then AND the word with a value having a string of 1 bits in bit 0 through $n-1$, where n is the number of bits in the field to extract. For example to extract bits 4 to 7, shift right four bits, and then and with 0xF.

To place some bits into position, you first need to clear the bits. Then OR the new field into the value. The first step is to build the mask with the proper number of 1's for the field width starting at bit 0. Then shift the mask left to align the mask with the value to hold the new field. Negate the mask to form an inverted mask. Then AND the value with the inverted mask to clear out the bits. Then shift the new value left the proper number of bits and or this with the value.

To access a single flag bit in EFLAGS, we can use the PUSHFD instruction to push the flags onto the stack. Read and modify them on the stack using shifting and masking operation mentioned above. Then use the POPF instruction to store them back into the flags register, if necessary. In addition, for reading and writing the sign, zero, auxiliary carry, parity, and carry flags, we can use LAHF to load the lower 8 bits (those 5 flags plus 3 indeterminate bits) into the AH register. We can use SAHF to store those values from AH back into the flags register. Notice that some flags can be set or cleared directly with specific instructions: CLC, STC, and CMC: clear, set, and complement the carry flag; CLI and STI: clear and set the interrupt flag (which should be done atomically); CLD and STD: clear and set the direction flag. Figure 1 shows the bit formation of EFLAGS.

1.4 Two-Dimensional Arrays

From an assembly language programmer's perspective, a two-dimensional array is a high-level abstraction of a one-dimensional array. High-level languages select one of two methods of arranging the rows and columns in memory: row-major order and column-major order, as shown in figure 2.

When row-major order (most common) is used, the first row appears at the beginning of the memory block. The last element in the first row is followed in memory by the first element of the second row. When column-major order is used, the elements in the first column appear at the beginning of the memory block. The last element in the first column is followed in memory by the first element of the second column. If you implement a two-dimensional array in assembly language, you can choose either ordering method.

2 Tasks

1. Implement a switch-case selection mechanism in assembly language. Your implementation should utilize local labels, unconditional jumps, an array of locations to jump to. Do not use conditional jumps.

2. Implement a program that returns the array value at a given row and column in a two-dimensional array of *doublewords*. Define the matrix M of equation 1 in a variable named *mat*. Define the i-th

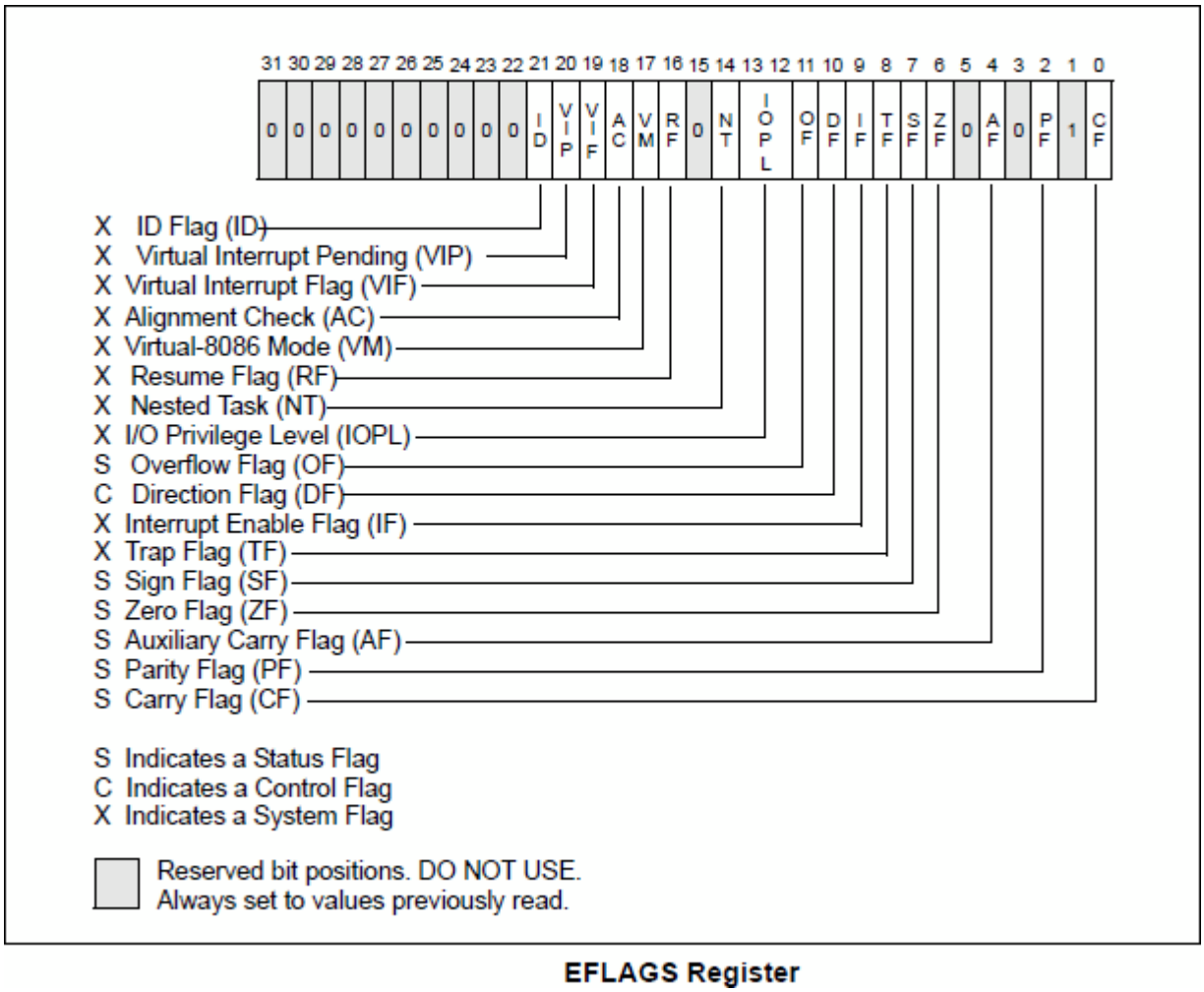

Figure 1: The EFLAGS register, taken from the Wikipedia

row and j-th column in variables i and j, respectively. Store the value of target element in a variable named val.

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \tag{1}$$

3. Implement a program that calculates the sum of a selected row in a matrix of integers: Define the matrix of equation 1 in a variable named mat. Define the i-th row in variables i and j. Store the summation value of target row i in a variable named sum.

4. Implement a program that finds the minimum value of a matrix M in equation 1.

5. Use an array to translate the decimal numbers 0 through 100 to their corresponding squares. Hint: Fill in the array with some looping mechanism.

6. Write an assembly program to determine if a string stored in memory is a palindrome. A palindrome is a string which is the same after being reversed, like "refer". Use at least one repeat instruction.

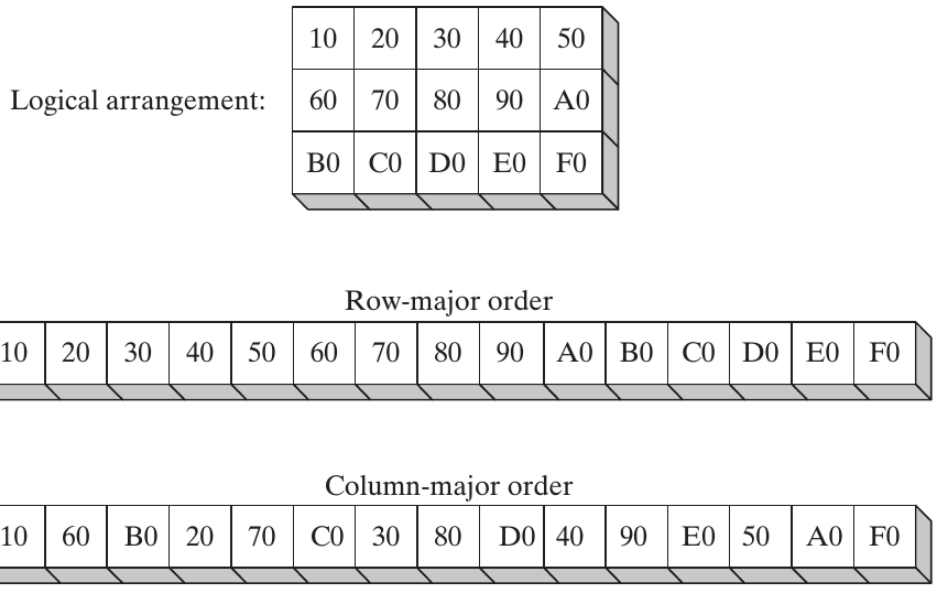

Figure 2: Row-major and column-major ordering, taken from [1]

7. Write an assembly program to compute the dot product of 2 arrays, i.e.

$$p = \sum_{i=0}^{n-1} a_i \cdot b_i$$

   Your arrays should be double word arrays in memory and the dot product should be stored in memory.

8. Write an assembly program to count all the 1 bits in a byte stored in memory. Use repeated code rather than a loop.

9. Implement a program to compute the greatest common divisor (GCD) of two integers. The GCD algorithm involves integer division in a loop, described by the following pseudocode:

Listing 2: The GCD pseudocode

```
1  int GCD(int x, int y) {
2    x = abs(x)
3    y = abs(y)
4
5    do {
6      int n = x % y
7      x = y
8      y = n
9    } while (y > 0)
10
11   return x
12 }
```

10. Implement the following C++ expression in assembly language, using 32-bit signed operands: $val1 = (val2/val3) \times (val1 + val2)$

11. Using a loop and indexed addressing, write code that rotates the members of a 32-bit integer array forward one position. The value at the end of the array must wrap around to the first position. For example, the array [10,20,30,40] would be transformed into [40,10,20,30].

12. Implement the following pseudocode in assembly language.

```
1 if( ebx > ecx ) OR ( ebx > val1 )
2     X = 1
3 else
4     X = 2
```

13. Implement the following pseudocode in assembly language.

```
1 if( ebx > ecx AND ebx > edx) OR ( edx > eax )
2   X = 1
3 else
4   X = 2
```

14. Use a loop with indirect or indexed addressing to reverse the elements of an integer array in place. Do not copy the elements to any other array.

3 Homework

1. Why will a PUSH AL instruction cause an error message to be displayed?

2. Determine the number of times that the following program segment executes the body of the loop:
   MOV CX, -1
   LP1:
   ...
   LOOP LP1

3. Generate a listing file for the AddTwoSum program and write a description of the machine code bytes generated for each instruction. You might have to guess at some of the meanings of the byte values.

4 Miscellany

The content is mainly based on the previous lab module and the following resources: [2], of [1] and [3]. (compiled on 14/03/2015 at 2:16pm)

References

[1] K. Irvine, *Assembly Language for X86 Processors.* Pearson Education, Limited, 2011. [Online]. Available: http://books.google.co.id/books?id=0k20RAAACAAJ

[2] R. Seyfarth, *Introduction to 64 Bit Intel Assembly Language Programming for Linux.* CreateSpace, 2012.

[3] J. Cavanagh, *X86 Assembly Language and C Fundamentals.* CRC Press, 2013.