

# Laboratory 6

Daniel Felipe Rambaut Lemus

September 2020

1. The mean of the a code construct is a code abstraction level that defines a functional property but not the details of its implementation. The impacts that can be generated when analyzing the assembly code are caused by the versions of the computers that are used to generate it.
2. One difference that occurs in the use of global variables is the fact that in memory it does not need to be assigned to within the execution of a function, if not by default it is in a global context. It can be seen in the mov of more that is used in the compilation of the code with local variables.

```
*.text:00401410      push    ebp
*.text:00401411      mov     ebp, esp
*.text:00401413      and     esp, 0FFFFFFF0h
*.text:00401416      sub     esp, 10h          ; char *
*.text:00401419      call    sub_4019A0
*.text:0040141E      mov     edx, dword_404004
*.text:00401424      mov     eax, dword_404008
*.text:00401429      add     eax, edx
*.text:0040142B      mov     dword_404004, eax
*.text:00401430      mov     eax, dword_404004
*.text:00401435      mov     [esp+10h+var_C], eax
*.text:00401439      mov     [esp+10h+var_10], offset aTotalD ; "Total = %d\n"
*.text:00401440      call    printf
*.text:00401445      nop
*.text:00401446      leave
*.text:00401447      retn
*.text:00401447      sub_401410      endp
```

Figure 1: Global variable

```
*.text:00401410      push    ebp
*.text:00401411      mov     ebp, esp
*.text:00401413      and     esp, 0FFFFFFF0h
*.text:00401416      sub     esp, 20h          ; char *
*.text:00401419      call    sub_4019A0
*.text:0040141E      mov     [esp+20h+var_4], 1
*.text:00401426      mov     [esp+20h+var_8], 2
*.text:0040142E      mov     eax, [esp+20h+var_8]
*.text:00401432      add     [esp+20h+var_4], eax
*.text:00401436      mov     eax, [esp+20h+var_4]
*.text:00401438      mov     [esp+20h+var_10], eax
*.text:0040143E      mov     [esp+20h+var_20], offset aTotalD ; "Total = %d\n"
*.text:00401445      call    printf
```

Figure 2: Local variable

```

> .text:0040141E      mov     [esp+20h+var_4], 0
> .text:00401426      mov     [esp+20h+var_8], 1
> .text:0040142E      add     [esp+20h+var_4], 0Bh
> .text:00401433      mov     eax, [esp+20h+var_8]
> .text:00401437      sub     [esp+20h+var_4], eax
> .text:0040143B      sub     [esp+20h+var_4], 1
> .text:00401440      add     [esp+20h+var_8], 1
> .text:00401445      mov     ecx, [esp+20h+var_4]
> .text:00401449      mov     edx, 55555556h
> .text:0040144E      mov     eax, ecx
> .text:00401450      imul    edx
> .text:00401452      mov     eax, ecx
> .text:00401454      sar     eax, 1Fh
> .text:00401457      sub     edx, eax
> .text:00401459      mov     eax, edx
> .text:0040145B      add     eax, eax
> .text:0040145D      add     eax, edx
> .text:0040145F      sub     ecx, eax
> .text:00401461      mov     eax, ecx
> .text:00401463      mov     [esp+20h+var_8], eax

```

Figure 3: Operations

3. In the previous image we can see to perform the addition, what the add function is performed, which takes the variable a and adds  $0Bh$  to it. Now to perform the subtraction, the variable of the value of b is moved and saved in eax, then we proceed to perform the sub of the variable a and eax. To perform the  $a--$  and  $b++$  operations, it is similar to the above, since the add and sub command is used to update the value of the variables. Finally we have the operation of the module, for this case the imul and sar functions are presented, which allow us to perform the multiplicative inverse and the respective subtraction to obtain the module.
4. In order to identify the if conditional, we can see it with assembly instructions such as *jnz*, which allows us to make a comparison between the values, which in our case are numbers. The *jnz* function makes a jump when the values are not equal and in this way we can identify when they are not equal.

```

> .text:00401410      push    ebp
> .text:00401411      mov     ebp, esp
> .text:00401413      and     esp, 0FFFFFFFh
> .text:00401416      sub     esp, 20h
> .text:00401419      call    sub_4019B0
> .text:0040141E      mov     [esp+20h+var_4], 1
> .text:00401426      mov     [esp+20h+var_8], 2
> .text:0040142E      mov     eax, [esp+20h+var_4]
> .text:00401432      cmp     eax, [esp+20h+var_8]
> .text:00401436      jnz     short loc_401446
> .text:00401438      mov     [esp+20h+var_20], offset aXEqualsY_ ; "x equals y."
> .text:0040143F      call    puts
> .text:00401444      jmp     short loc_401452
> .text:00401446      ;
> .text:00401446      loc_401446:
> .text:00401446      mov     [esp+20h+var_20], offset aXIsNotEqualY_ ; "x is not equal to y."
> .text:00401446      call    puts
> .text:00401452

```

Figure 4: Conditional

5. For the realization of the two nested ifs, we can analyze them as we saw

previously that if we use the jnz function and the cmp function, which allow us to compare the values. The cmp function this instruction subtracts the source operand from the destination operand but without it storing the result of the operation, only the state of the flags is affected. With this present we can see in the image that there are several branches for the established conditions.

```

* .text:00401410      push    ebp
* .text:00401411      mov     ebp, esp
* .text:00401413      and     esp, 0FFFFFF0h
* .text:00401416      sub     esp, 20h          ; char *
* .text:00401419      call   sub_4019E0
* .text:0040141E      mov     [esp+20h+var_4], 0
* .text:00401426      mov     [esp+20h+var_8], 1
* .text:0040142E      mov     [esp+20h+var_C], 2
* .text:00401436      mov     eax, [esp+20h+var_4]
* .text:0040143A      cmp     eax, [esp+20h+var_8]
* .text:0040143E      jnz     short loc_401463
* .text:00401440      cmp     [esp+20h+var_8], 0
* .text:00401445      jnz     short loc_401455
* .text:00401447      mov     [esp+20h+var_20], offset a2IsZeroAndXY_ ; "z is zero and x = y."
* .text:0040144E      call   puts
* .text:00401455      jmp     short loc_401484
* .text:0040145C
* .text:0040145C

```

Figure 5: Conditional

```

* .text:00401455 ; -----
* .text:00401455      loc_401455:
* .text:00401455      mov     [esp+20h+var_20], offset a2IsNonZeroAndX_ ; "z is non-zero and x = y."
* .text:0040145C      call   puts
* .text:00401461      jmp     short loc_401484
* .text:00401463 ; -----
* .text:00401463      loc_401463:
* .text:00401463      cmp     [esp+20h+var_C], 0
* .text:00401468      jnz     short loc_401478
* .text:0040146A      mov     [esp+20h+var_20], offset a2IsZeroAndXY_0 ; "z is zero and x != y."
* .text:00401471      call   puts
* .text:00401476      jmp     short loc_401484
* .text:00401478 ; -----
* .text:00401478      loc_401478:
* .text:00401478      mov     [esp+20h+var_20], offset a2IsNonZeroAndX_ ; "z is non-zero and x = y."
* .text:0040147F      call   puts
* .text:00401484

```

Figure 6: Conditional

6. The way to identify the loops is to be able to see how the jmp is used to perform the validations of the variable that is increased in each iteration. Also in the image below we can see how the code prints the variables and also redoes the validation with cmp.

```

* .text:00401419      call   sub_4019A0
* .text:0040141E      mov     [esp+20h+var_4], 0
* .text:00401426      jmp     short loc_401441
* .text:00401428 ; -----
* .text:00401428      loc_401428:
* .text:00401428      mov     eax, [esp+20h+var_4]
* .text:0040142C      mov     [esp+20h+var_1C], eax
* .text:00401430      mov     [esp+20h+var_20], offset a1Equals0 ; "i equals %d \n"
* .text:00401437      call   printf
* .text:0040143C      add     [esp+20h+var_4], 1
* .text:00401441      jmp     short loc_401441
* .text:00401441      loc_401441:
* .text:00401441      cmp     [esp+20h+var_4], 63h
* .text:00401446      jle     short loc_401428
* .text:00401448      nop
* .text:00401449      nop
* .text:0040144A      leave
* .text:0040144B      retn
* .text:0040144B      sub_401410
* .text:0040144B      endp

```

Figure 7: Loop

7. In order to identify the while we can see that the declaration of the comparison variables is presented to stop the while. It is worth mentioning that I have modified the example in the book for my own and in we can see how the result variable is assigned the value plus one in each iteration. The while can be recognized at the time of comparison before entering the while.

```

* .text:0040141E      mov     [esp+20h+var_8], 0
* .text:00401426      mov     [esp+20h+var_4], 0
* .text:0040142E      jmp     short loc_401435
* .text:00401430      ; -----
* .text:00401430      loc_401430:      add     [esp+20h+var_4], 1 ; CODE XREF: sub_401410+204j
* .text:00401435      loc_401435:      cmp     [esp+20h+var_8], 0 ; CODE XREF: sub_401410+1E7j
* .text:00401438      jz      short loc_401430
* .text:0040143C      mov     eax, [esp+20h+var_4]
* .text:00401440      mov     [esp+20h+var_1C], eax
* .text:00401444      mov     [esp+20h+var_20], offset aResultado0 ; "Resultado %d"
* .text:00401448      call    printf
* .text:00401450      nop
* .text:00401451      leave
* .text:00401452      sub_401410      retn
* .text:00401452      sub_401410      endp

```

Figure 8: While