Escuela de Ingeniería, Ciencia y Tecnología.

Code Constructs

September 14th, 2020

Daniel Forero Corredor

1 Read the introduction of the section 6 "Recognizing C Code Constructs in Assembly" and explain what means a "Code Construct". What aspects may impact the way as assembly code is generated?

Code construct makes reference to the abstraction of a code implementation. Showing properties and structures but no details. Assembly code abstracted from a C file may change depending on the compiler and OS used by the user. It will be noticed during this lab, the code obtained is in some small aspects different to the one on the book.

2 Read the section "Global vs Local Variables" and identify what are the differences in the compilation of a code that employs global vs one that employs local Variables.

Consider the following two *C* codes:

```c
#include<stdio.h>

int x=1;
int y=2;

void main(){
  x = x+y;
  printf("Total = %d\n", x);
}
```

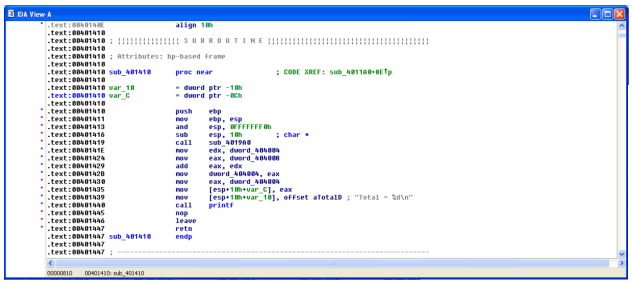
```c
#include<stdio.h>

void main(){
  int x=1;
  int y=2;

  x = x+y;
  printf("Total = %d\n", x);
}
```

Figure 1: C code for local and global variables.

Now It's possible to analyze the difference between the **assembly code** obtained on *IDA pro* and the one book. Let's analyze **global variable declaration** first:

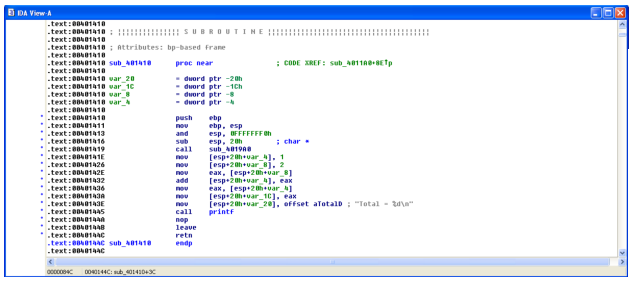


```
00401003        mov     eax, dword_40CF60
00401008        add     eax, dword_40C000
0040100E        mov     dword_40CF60, eax
00401013        mov     ecx, dword_40CF60
00401019        push    ecx
0040101A        push    offset aTotalD  ;"total = %d\n"
0040101F        call    printf
```

Listing 6-3: Assembly code for the global variable example in Listing 6-1

Figure 2: Comparison between own assembly code and book's for global variables.

Now the **global variables**:



```
00401006        mov     dword ptr [ebp-4], 0
0040100D        mov     dword ptr [ebp-8], 1
00401014        mov     eax, [ebp-4]
00401017        add     eax, [ebp-8]
0040101A        mov     [ebp-4], eax
0040101D        mov     ecx, [ebp-4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

Listing 6-4: Assembly code for the local variable example in Listing 6-2, without labeling

Figure 3: Comparison between own assembly code and book's for local variables.

The difference on the compilation of a code with *global* and *local* variables is the way they are referenced.Global variables are referenced by *memory address*, the local ones are referenced by *stack address*.It means stacks are used to store all the variables and values created inside a function. Those stacks are used only during the execution of the function that's why this variables doesn't get an actual memory space.

3 Read the section "Disassembling Arithmetic Operations" and explain to your classmates how the operations (addition, subtraction, increment, decrement and modulo) are represented in assembly code.

Consider the following C code.

```
void main(){
    int a = 0;
    int b = 1;
    a = a + 11;
    a = a - b;
    b = a % 3;
}
```

Now Analyze and compare the own *assembly code* vs. the one provided by the book.

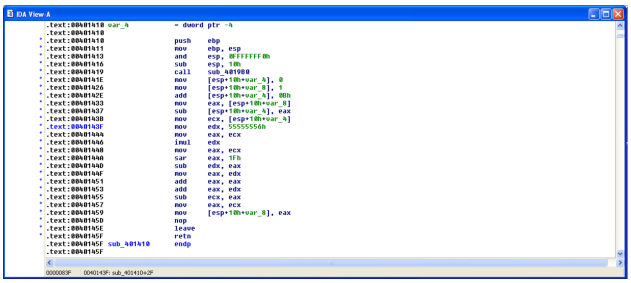


Figure 4: Assembly code obtained on *IDA pro*.

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4] ❶
00401017        add     eax, 0Bh
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        sub     ecx, [ebp+var_8] ❷
00401023        mov     [ebp+var_4], ecx
00401026        mov     edx, [ebp+var_4]
00401029        sub     edx, 1 ❸
0040102C        mov     [ebp+var_4], edx
0040102F        mov     eax, [ebp+var_8]
00401032        add     eax, 1 ❹
00401035        mov     [ebp+var_8], eax
00401038        mov     eax, [ebp+var_4]
0040103B        cdq
0040103C        mov     ecx, 3
```

```
00401041        idiv    ecx
00401043        mov     [ebp+var_8], edx ❺
```

Listing 6-7: Assembly code for the arithmetic example in Listing 6-6

Figure 5: Assembly code provided by the book.

Notice this code has **variable creation, addition, subtraction and module**. Variable creation is visible on *text: 0040141E* and *text:00401426* of the *assembly* code. According to what was mentioned on the last questions It's possible to tell those variables are created as **local** ones.

Addition is placed on the following 2 lines, doing the addition first, then reorganizing the memory, associating the value obtained to the stack address of the variable a. The process is analog for the **subtraction** on the lines 5 and 6.

Modulo function (found from text:0040143F) is a more complicated process according to the *mov's, add's, sar's, etc.* involved on it.

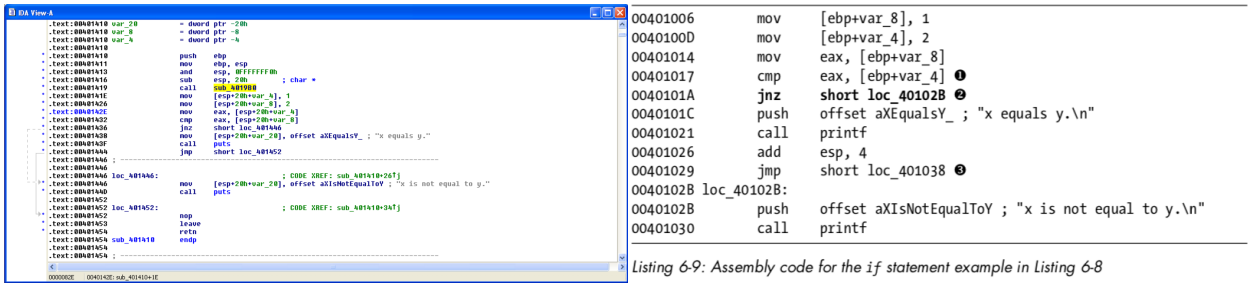
4 Read the section "Recognizing if Statements" and explain to your classmates how to recognize an if/else structure in assembly code.

Consider the following *C* code:

```c
#include<stdio.h>

void main(){
    int x = 1;
    int y = 2;
    if(x==y){
        printf("x equals y.\n");
    }else{
        printf("x is not equal to y.\n");
    }
}
```

Now let's analyze and compare It's *assembly codes* with the one provided by the book:



```
00401006        mov     [ebp+var_8], 1
0040100D        mov     [ebp+var_4], 2
00401014        mov     eax, [ebp+var_8]
00401017        cmp     eax, [ebp+var_4] ❶
0040101A        jnz     short loc_40102B ❷
0040101C        push    offset aXEqualsY_ ; "x equals y.\n"
00401021        call    printf
00401026        add     esp, 4
00401029        jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B        push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030        call    printf
```

Listing 6-9: Assembly code for the if statement example in Listing 6-8

Figure 6: Assembly codes.

First we make sure that the beginning and the end of the program is clear. It starts on *text:0040141E*(line 1) and ends on *00401454*.

After the variable declarations, there is a *cmp* instruction that compares the values (the if condition). This instruction is followed by a *jnz* (jump if not zero) instruction that analyzed the result provided by the comparison and if it is 0 (false) redirects us to the *else* section of the code (prints *x equals y*). If the *jnz*

doesn't occur the program continues with the *if* section of the code(prints *x is not equal to y*).

Both of those options end with *jmp* instructions that redirect the process to the code after the if statement.

5 Read the section "Recognizing Nested if Statements" and explain to your classmates how to recognize a "Nested IF" structure in assembly code.

Consider the following *C* code:

```c
#include<stdio.h>

void main (){
    int x = 1;
    int y = 1;
    int z = 2;
    if(x==y){
        if(z==0){
            printf("x equals y.\n");
        }else{
            printf("x is not equal to y.\n");
        }
    }else{
        if(z==0){
            printf("z zero and x != y.\n");
        }else{
            printf("z non-zero and x != y.\n");
        }
    }
}
```

Now Analyze and compare the own *assembly code* vs. the one provided by the book. item[R).] Consider the following C code.

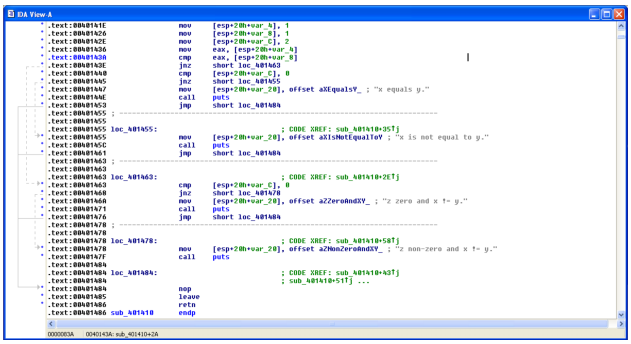


Figure 7: Assembly code obtained on *IDA pro*.

```
00401006        mov     [ebp+var_8], 0
0040100D        mov     [ebp+var_4], 1
00401014        mov     [ebp+var_C], 2
0040101B        mov     eax, [ebp+var_8]
0040101E        cmp     eax, [ebp+var_4]
00401021        jnz     short loc_401047 ❶
00401023        cmp     [ebp+var_C], 0
00401027        jnz     short loc_401038 ❷
00401029        push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E        call    printf
00401033        add     esp, 4
00401036        jmp     short loc_401045
00401038 loc_401038:
00401038        push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D        call    printf
00401042        add     esp, 4
```

```
00401045 loc_401045:
00401045        jmp     short loc_401069
00401047 loc_401047:
00401047        cmp     [ebp+var_C], 0
0040104B        jnz     short loc_40105C ❸
0040104D        push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052        call    printf
00401057        add     esp, 4
0040105A        jmp     short loc_401069
0040105C loc_40105C:
0040105C        push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061        call    printf00401061
```

Listing 6-11: Assembly code for the nested if statement example shown in Listing 6-10

Figure 8: Assembly code provided by the book.

The behavior of this assembly is similar to the one on the "single if" code. The difference is the use of the *jnz*'s one inside the others according to the intersection of the conditions.

Just like on the exercise before, all of the functions produced according to the conditions end with *jmp* instructions that lead the code to what's after the nested if's.

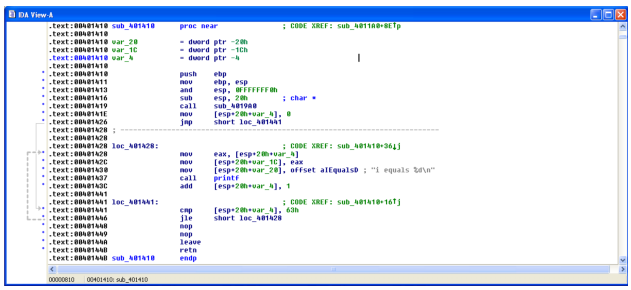
6 Read the section "Recognizing Loops" and explain to your classmates how to recognize a FOR structure in assembly code.

Consider the following *C* code:

```c
#include<stdio.h>

void main(){
    int i;
    for(i=0; i<100;i++){
        printf("i equals %d\n", i);
    }
}
```

Now Analyze and compare the own *assembly code* vs. the one provided by the book.



```
00401004        mov     [ebp+var_4], 0 ❶
0040100B        jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D        mov     eax, [ebp+var_4] ❸
00401010        add     eax, 1
00401013        mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016        cmp     [ebp+var_4], 64h ❺
0040101A        jge     short loc_40102F ❻
0040101C        mov     ecx, [ebp+var_4]
0040101F        push    ecx
00401020        push    offset aID  ; "i equals %d\n"
00401025        call    printf
0040102A        add     esp, 8
0040102D        jmp     short loc_40100D ❼
```

Listing 6-13: Assembly code for the for loop example in Listing 6-12

Figure 9: Assembly code provided by the book.

Code starts creating the local variable *i*. After it it executes a *jmp* instruction that leads to a comparison, in this case it's a *jle*instruction. This checks if the first value is less or equal that the second. This condition determines if the for loop executes or not. If it does the *jle* instructions redirects the code to the section in which prints the value of *i* and increases it by one.

Then it goes back to the "condition checking" section to determine if this sequence of instructions should be repeated.

7 Read the section "Recognizing Loops" and explain to your classmates how to recognize a WHILE structure in assembly code.

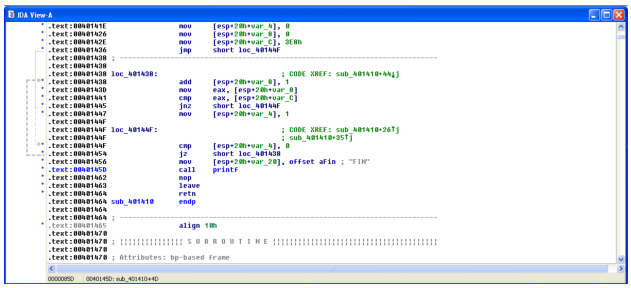
Consider the following *C* code:

```c
#include<stdio.h>

void main(){
    int status = 0;
    int result = 0;

    while(status == 0){
        result = performAction();
        status = chechResult(result);
    }
}
```

Now let's analyze and compare It's *assembly codes* with the one provided by the book:



```
00401036        mov     [ebp+var_4], 0
0040103D        mov     [ebp+var_8], 0
00401044 loc_401044:
00401044        cmp     [ebp+var_4], 0
00401048        jnz     short loc_401063 ❶
0040104A        call    performAction
0040104F        mov     [ebp+var_8], eax
00401052        mov     eax, [ebp+var_8]
00401055        push    eax
00401056        call    checkResult
0040105B        add     esp, 4
0040105E        mov     [ebp+var_4], eax
00401061        jmp     short loc_401044 ❷
```

Listing 6-15: Assembly code for the while loop example in Listing 6-14

Figure 10: Assembly codes.

The while loop is a sort of for loop with out an **explicit** iteration variable. This loops has almost the same behavior as the for loop. The difference is the existence of an accessible iteration variable and the breaking point of the loop. Breaking point is also given by a comparison, in this case a **Boolean** one.

8 BIBLIOGRAPHY:

- Siroski, M.,(2012).*Practical Malware Analysis. The hands-on guide to dissecting malicious software*, San Francisco, USA: no starch press.
- https://stackoverflow.com/questions/3527026/assembly-language-more-than-one-type/3527083
- Intel® 64 and IA-32 Architectures Software Developer's Manual
- X86/WIN32 reverse engineering cheat-sheet