6th laboratory

Rodrigo Castillo

September 14, 2020

1 operations :addition, substraction , increment, decrement and modulo

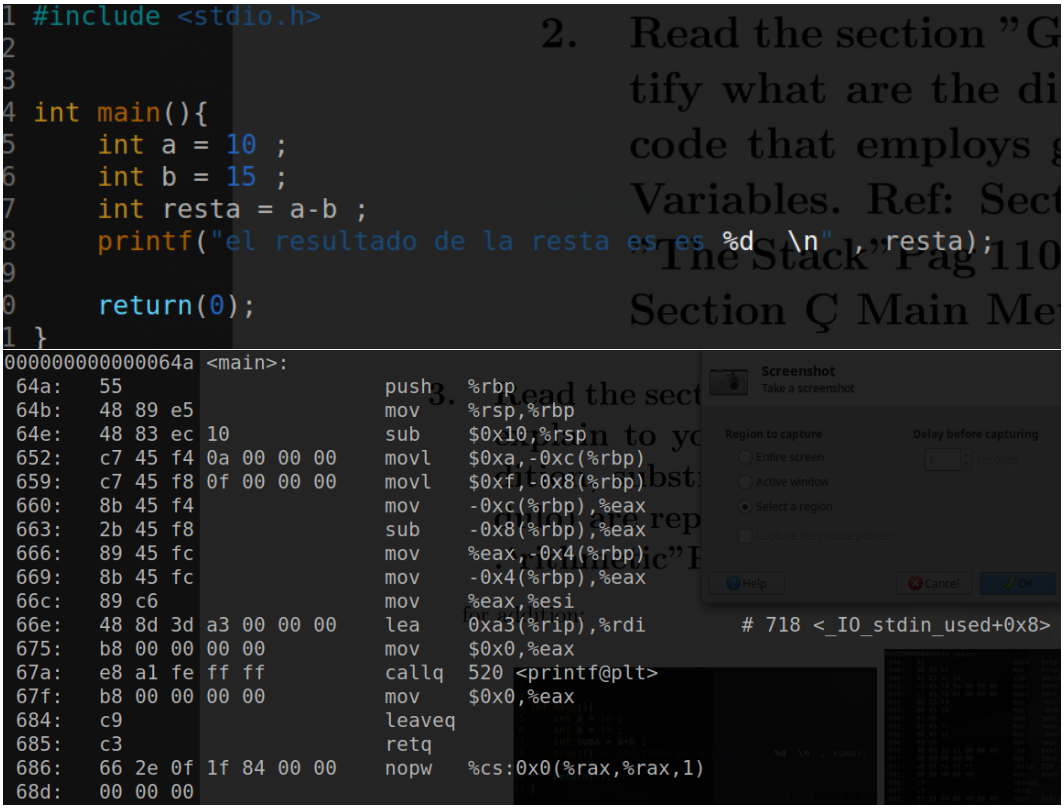
for addition: we can see that the addition operation is in line 666 , in operation *add* , as i defined two variables before, assembly code is locating hex values into the variables i defined and then, he es adding the values into the new variable

```c
#include <stdio.h>


int main(){
    int a = 10 ;
    int b = 15 ;
    int suma = a+b ;
    printf("el resultado de la suma es es %d  \n" , suma);

    return(0);
}
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 f4 0a 00 00 00    movl   $0xa,-0xc(%rbp)
 659:   c7 45 f8 0f 00 00 00    movl   $0xf,-0x8(%rbp)
 660:   8b 55 f4                mov    -0xc(%rbp),%edx
 663:   8b 45 f8                mov    -0x8(%rbp),%eax
 666:   01 d0                   add    %edx,%eax
 668:   89 45 fc                mov    %eax,-0x4(%rbp)
 66b:   8b 45 fc                mov    -0x4(%rbp),%eax
 66e:   89 c6                   mov    %eax,%esi
 670:   48 8d 3d a1 00 00 00    lea    0xa1(%rip),%rdi        # 718 <_IO_stdin_used+0x8>
 677:   b8 00 00 00 00          mov    $0x0,%eax
 67c:   e8 9f fe ff ff          callq  520 <printf@plt>
 681:   b8 00 00 00 00          mov    $0x0,%eax
 686:   c9                      leaveq
 687:   c3                      retq
 688:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
 68f:   00
```

Figure 1: Code and disassembly for addition

for substraction: this is the same as the addition operation, but, the code is replacing addition operation with substraction operation in line 663 .



```
1 #include <stdio.h>
2
3
4 int main(){
5     int a = 10 ;
6     int b = 15 ;
7     int resta = a-b ;
8     printf("el resultado de la resta es es %d  \n" ,resta);
9
0     return(0);
1 }
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 f4 0a 00 00 00    movl   $0xa,-0xc(%rbp)
 659:   c7 45 f8 0f 00 00 00    movl   $0xf,-0x8(%rbp)
 660:   8b 45 f4                mov    -0xc(%rbp),%eax
 663:   2b 45 f8                sub    -0x8(%rbp),%eax
 666:   89 45 fc                mov    %eax,-0x4(%rbp)
 669:   8b 45 fc                mov    -0x4(%rbp),%eax
 66c:   89 c6                   mov    %eax,%esi
 66e:   48 8d 3d a3 00 00 00    lea    0xa3(%rip),%rdi        # 718 <_IO_stdin_used+0x8>
 675:   b8 00 00 00 00          mov    $0x0,%eax
 67a:   e8 a1 fe ff ff          callq  520 <printf@plt>
 67f:   b8 00 00 00 00          mov    $0x0,%eax
 684:   c9                      leaveq
 685:   c3                      retq
 686:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 68d:   00 00 00
```

Figure 2: Code and disassembly for substraction

for increment: first, in line 652 we can see that is adding the value $0xa$ into a register, this value, is 10 , then its adding $0x1$ into the same register, this means that is adding 1 and now the value of this register is 11 .

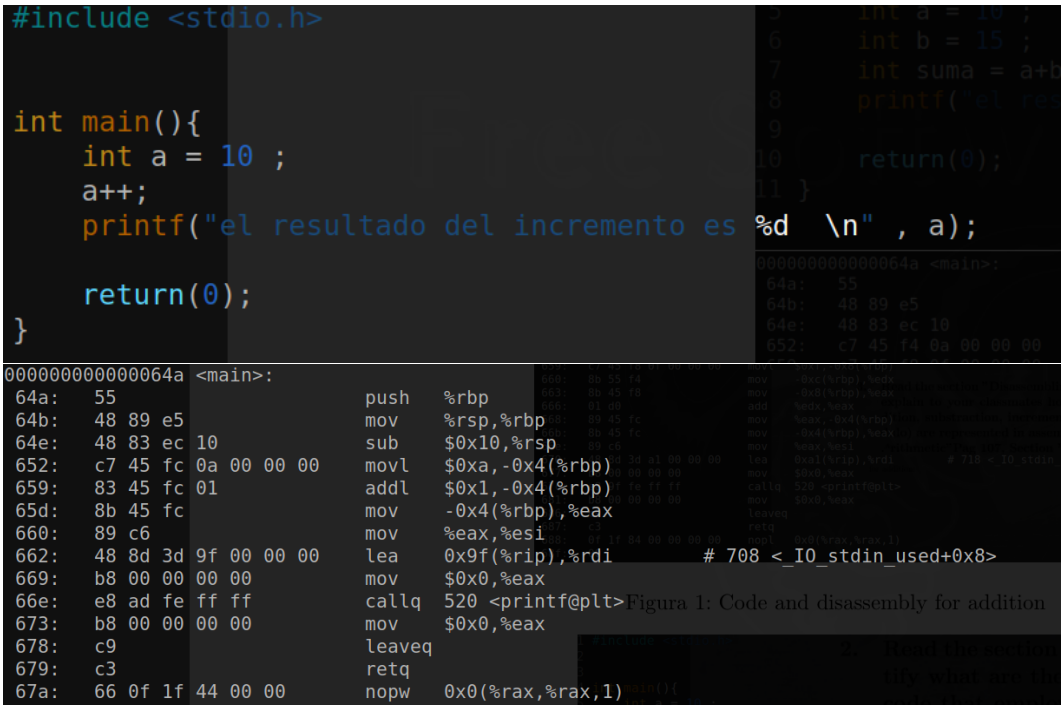
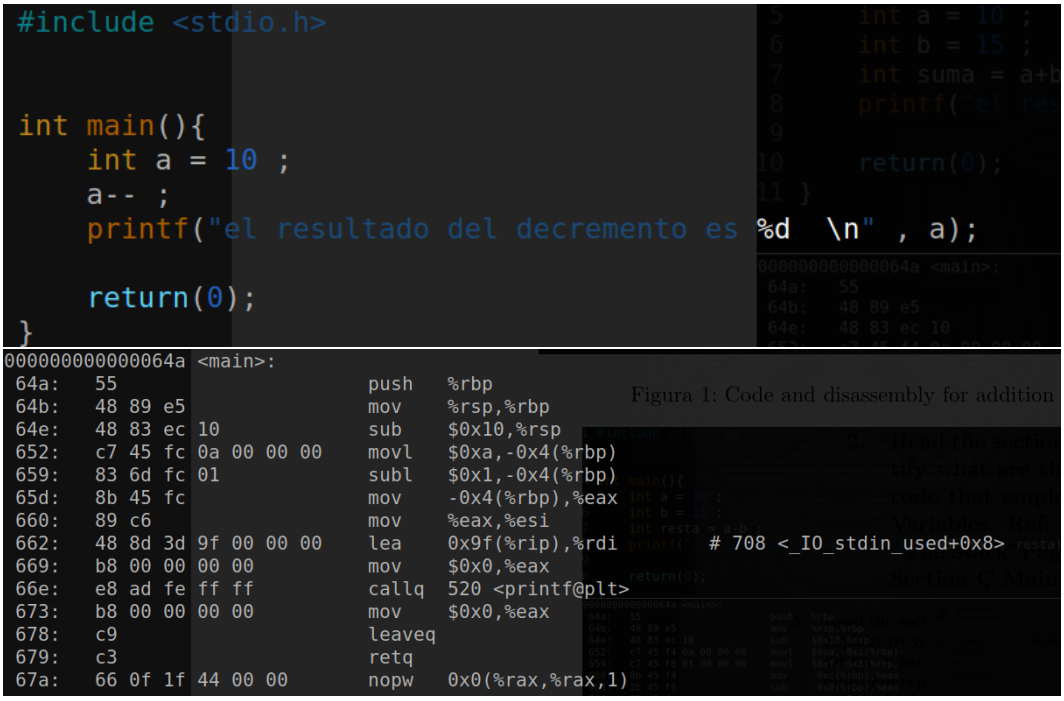


Figura 1: Code and disassembly for addition

Figure 3: Code and dissasembly for increment

for decrement: is exactly the same as increment, but now, he es substracting instead of adding $0x1$ value into the register.



```c
#include <stdio.h>


int main(){
    int a = 10 ;
    a-- ;
    printf("el resultado del decremento es %d  \n" , a);

    return(0);
}
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
 659:   83 6d fc 01             subl   $0x1,-0x4(%rbp)
 65d:   8b 45 fc                mov    -0x4(%rbp),%eax
 660:   89 c6                   mov    %eax,%esi
 662:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi        # 708 <_IO_stdin_used+0x8>
 669:   b8 00 00 00 00          mov    $0x0,%eax
 66e:   e8 ad fe ff ff          callq  520 <printf@plt>
 673:   b8 00 00 00 00          mov    $0x0,%eax
 678:   c9                      leaveq
 679:   c3                      retq
 67a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```

Figura 1: Code and disassembly for addition

Figure 4: Code and disassembly for decrement opperation

for modulo: is actually the same as addition and substraction, but the opperation *idivl* stands for division, the result of this divion is going to store the residue in the designer register.

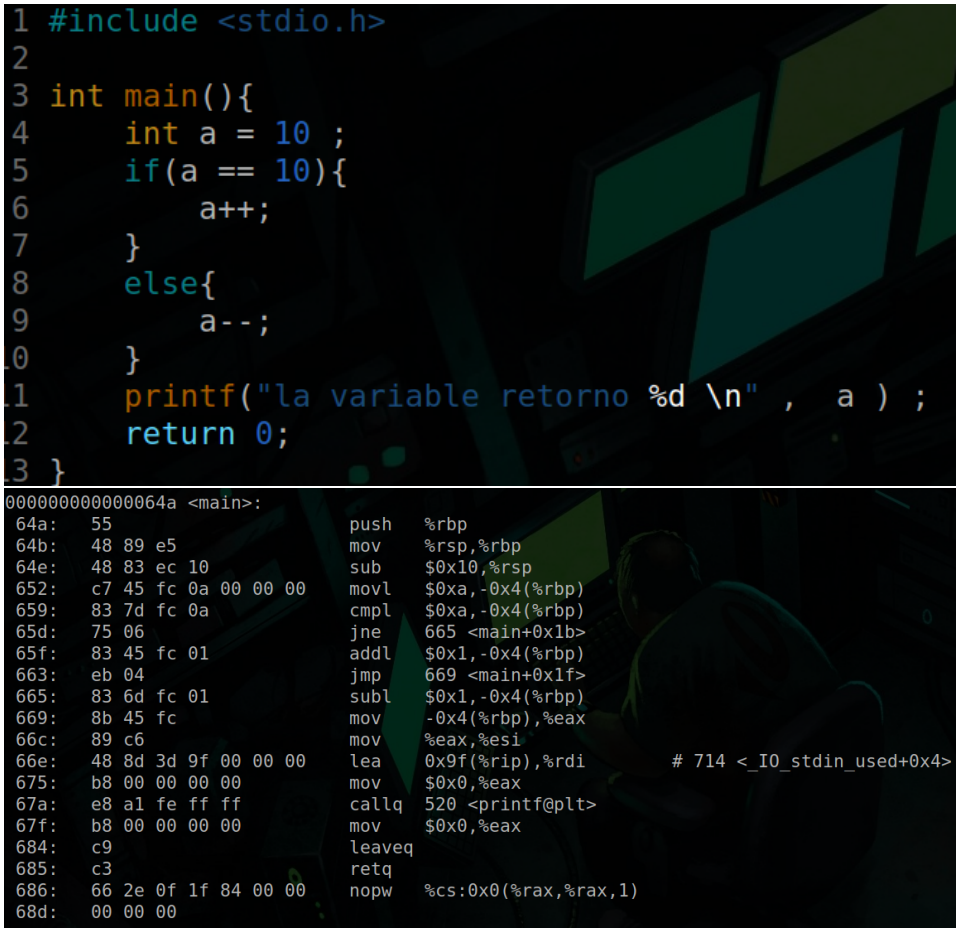
```c
#include <stdio.h>


int main(){
    int a = 25 ;
    int b = 15 ;
    int mod = a%b ;
    printf("el resultado del modulo es es %d  \n" , mod);

    return(0);
}
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 f4 19 00 00 00    movl   $0x19,-0xc(%rbp)
 659:   c7 45 f8 0f 00 00 00    movl   $0xf,-0x8(%rbp)
 660:   8b 45 f4                mov    -0xc(%rbp),%eax
 663:   99                      cltd
 664:   f7 7d f8                idivl  -0x8(%rbp)
 667:   89 55 fc                mov    %edx,-0x4(%rbp)
 66a:   8b 45 fc                mov    -0x4(%rbp),%eax
 66d:   89 c6                   mov    %eax,%esi
 66f:   48 8d 3d a2 00 00 00    lea    0xa2(%rip),%rdi        # 718 <_IO_stdin_used+0x8>
 676:   b8 00 00 00 00          mov    $0x0,%eax
 67b:   e8 a0 fe ff ff          callq  520 <printf@plt>
 680:   b8 00 00 00 00          mov    $0x0,%eax
 685:   c9                      leaveq
 686:   c3                      retq
 687:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
 68e:   00 00
```

Figure 5: Code for modulo

2 Q4: Read the section "Recognizing if Statements" and explain to your classmates how to recognize an if/else structure in assembly code. Ref: Section "Conditionals" Pag 113, Section "Branching" Pag 113

if statement in c: this disassemble, first, saves the variable 10 into the register, next, the compare instruction checks if the value in the register is equal to 10, if is not equal , the instruction pointer is going to follow instruction 665 and substract, but if is equal, $\$eip$ is going to point at next instruction $65f$ and add 1 to the register.



```
1 #include <stdio.h>
2
3 int main(){
4     int a = 10 ;
5     if(a == 10){
6         a++;
7     }
8     else{
9         a--;
10    }
11    printf("la variable retorno %d \n" ,  a ) ;
12    return 0;
13 }
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
 659:   83 7d fc 0a             cmpl   $0xa,-0x4(%rbp)
 65d:   75 06                   jne    665 <main+0x1b>
 65f:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
 663:   eb 04                   jmp    669 <main+0x1f>
 665:   83 6d fc 01             subl   $0x1,-0x4(%rbp)
 669:   8b 45 fc                mov    -0x4(%rbp),%eax
 66c:   89 c6                   mov    %eax,%esi
 66e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi        # 714 <_IO_stdin_used+0x4>
 675:   b8 00 00 00 00          mov    $0x0,%eax
 67a:   e8 a1 fe ff ff          callq  520 <printf@plt>
 67f:   b8 00 00 00 00          mov    $0x0,%eax
 684:   c9                      leaveq
 685:   c3                      retq
 686:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 68d:   00 00 00
```

Figure 6: if statement and disassemble

3 Q5: Read the section "Recognizing Nested if Statements" and explain to your classmates how to recognize a "Nested IF" structure in assembly code.Ref: Section "Conditionals" and "Branching" Pag 113

nested: inside the memory spaces of the first *jmp* instruction there are more *jmp* instruction .

```c
#include <stdio.h>

int main(){
    int a = 10;
    if(a==10){
        if(a == 12){
            //imposible si el flujo de ejecucion
            //no es modificado
            printf("%d es igual a 12", a);
        }
    }
    return 0 ;
}
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
 659:   83 7d fc 0a             cmpl   $0xa,-0x4(%rbp)
 65d:   75 1c                   jne    67b <main+0x31>
 65f:   83 7d fc 0c             cmpl   $0xc,-0x4(%rbp)
 663:   75 16                   jne    67b <main+0x31>
 665:   8b 45 fc                mov    -0x4(%rbp),%eax
 668:   89 c6                   mov    %eax,%esi
 66a:   48 8d 3d a3 00 00 00    lea    0xa3(%rip),%rdi        # 714 <_IO_stdin_used+0x4>
 671:   b8 00 00 00 00          mov    $0x0,%eax
 676:   e8 a5 fe ff ff          callq  520 <printf@plt>
 67b:   b8 00 00 00 00          mov    $0x0,%eax
 680:   c9                      leaveq
 681:   c3                      retq
 682:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 689:   00 00 00
 68c:   0f 1f 40 00             nopl   0x0(%rax)
```

Figure 7: Nested code and disassemble

4 Q6: Read the section "Recognizing Loops" and explain to your classmates how to recognize a FOR structure in assembly code. Ref: Section "Conditionals" and "Branching" Pag 113

for: a for loop, in disassambly language, its the mix between an if and adittion instructions, here, we can see that the assembly code is executing print instruction. at first, he defines variable at 0 because in code, i = 0 , then it executes the instruction and then it compares the variable with 10, if the variable is equal to 10, follows normal flood of execution , but if variable is not equal to 10, it increases it by one and sets *eip* to 65*b* , that means that the program is going to execute this instruction 10 times until the variable $i = 10$.

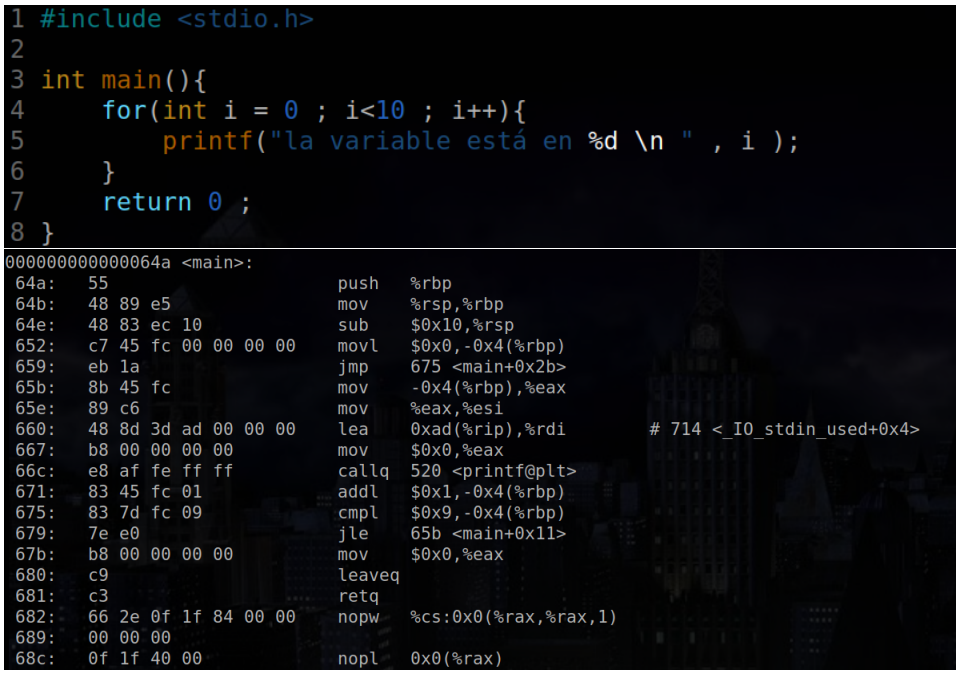


Figure 8: For loop and disassemble

5 Q7: Read the section "Recognizing Loops" and explain to your classmates how to recognize a WHILE structure in assembly code. Ref: Section "Conditionals" and "Branching" Pag 113

while: while loop is similar to for loop, the difference is that its not going to increment anything if the code dont tell it to do , basically, in this example, its going to compare a variable with $99 = 0x63$, if the comparisson returns true, the code is going to continue normal flood of execution, but if not, its going to set $\$eip$ to $0x663$ intruction, this means that its going to execute all the code again till the comparisson between a and $0x63$ returns true, by the way, this code is finit because in the instruction 678 i am adding 1 to variable a , but this is because i put it in the code, but this is not mandatory, this can make infinite loops .

```
1 #include <stdio.h>
2
3 int main(){
4     int a = 35;
5     int b = 10;
6     while(a<100){
7         printf("a es : %d \n" , a);
8         a++ ;
9         b-- ;
10    }
11    return 0 ;
12 }
```

```
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   c7 45 f8 23 00 00 00    movl   $0x23,-0x8(%rbp)
 659:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
 660:   eb 1e                   jmp    680 <main+0x36>
 662:   8b 45 f8                mov    -0x8(%rbp),%eax
 665:   89 c6                   mov    %eax,%esi
 667:   48 8d 3d a6 00 00 00    lea    0xa6(%rip),%rdi        # 714 <_IO_stdin_used+0x4>
 66e:   b8 00 00 00 00          mov    $0x0,%eax
 673:   e8 a8 fe ff ff          callq  520 <printf@plt>
 678:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
 67c:   83 6d fc 01             subl   $0x1,-0x4(%rbp)
 680:   83 7d f8 63             cmpl   $0x63,-0x8(%rbp)
 684:   7e dc                   jle    662 <main+0x18>
 686:   b8 00 00 00 00          mov    $0x0,%eax
 68b:   c9                      leaveq
 68c:   c3                      retq
 68d:   0f 1f 00                nopl   (%rax)
```

Figure 9: while code and disassemble

6 Q8: Read the section "Understanding Function Call Convenstions" and explain to your classmates how to recognize a "function call" in assembly code. Ref: Section "Function Calls" Pag 110. Section "Stack Layout" Pag 111.

function: calling a function into disassembly means that compiler is going to create another section for it with the name of the function in the code ,this will let the programmer call a function many times with only one definition. The processor is going to run the *main* function by default, but, inside main function, the *callq* intruction tells to execute the function inside the parameter, in this case we can see that the instruction is being called in $67e$, that means that in this space the pointer *eip* is going to jump to $64a$ where the function $func$ is allocated and its going to execute those intructions, the instructions in $64a$ section. when its done, the *retq* its going to return the *eip* pointer to the original flood of execution in *main* function.

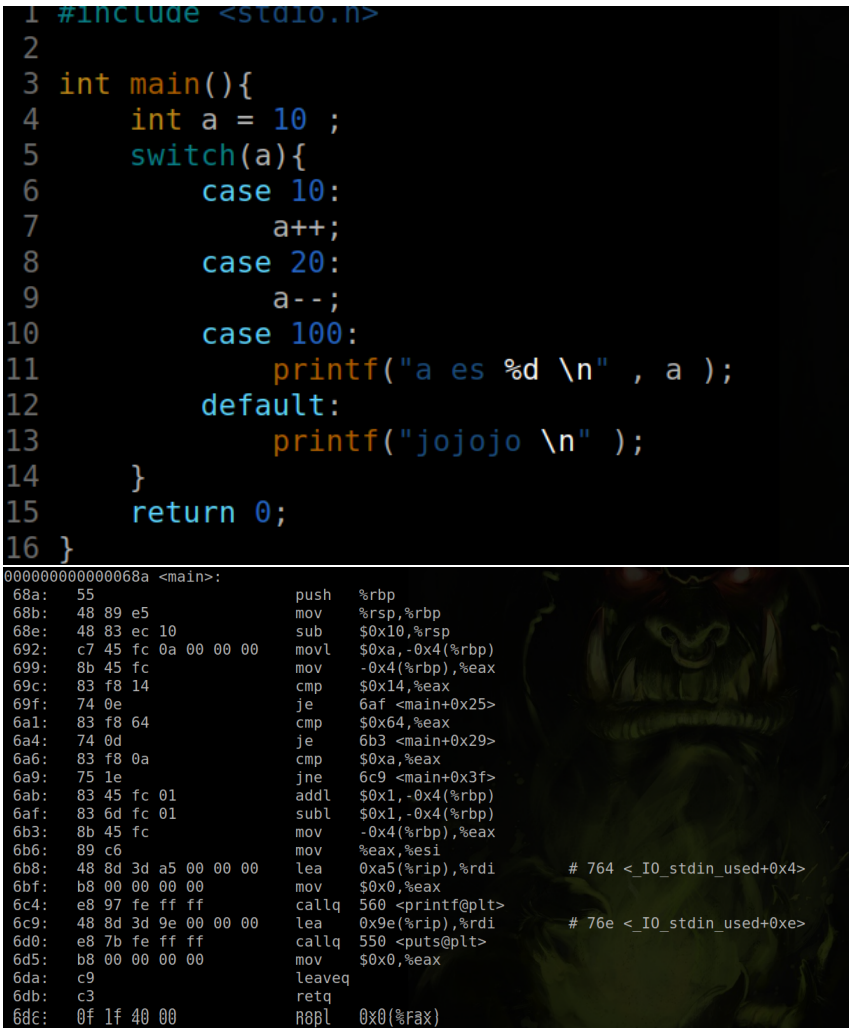
```
1 #include <stdio.h>
2
3
4 int func(int a, int b){
5     return(a+b);
6 }
7
8 int main(){
9     int a = 10;
10    int b = 0x10;
11    int variable = func(a,b);
12    printf("la variable es %d \n" , variable);
13    return 0;
14 }
```

```
000000000000064a <func>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   89 7d fc                mov    %edi,-0x4(%rbp)
 651:   89 75 f8                mov    %esi,-0x8(%rbp)
 654:   8b 55 fc                mov    -0x4(%rbp),%edx
 657:   8b 45 f8                mov    -0x8(%rbp),%eax
 65a:   01 d0                   add    %edx,%eax
 65c:   5d                      pop    %rbp
 65d:   c3                      retq

000000000000065e <main>:
 65e:   55                      push   %rbp
 65f:   48 89 e5                mov    %rsp,%rbp
 662:   48 83 ec 10             sub    $0x10,%rsp
 666:   c7 45 f4 0a 00 00 00    movl   $0xa,-0xc(%rbp)
 66d:   c7 45 f8 10 00 00 00    movl   $0x10,-0x8(%rbp)
 674:   8b 55 f8                mov    -0x8(%rbp),%edx
 677:   8b 45 f4                mov    -0xc(%rbp),%eax
 67a:   89 d6                   mov    %edx,%esi
 67c:   89 c7                   mov    %eax,%edi
 67e:   e8 c7 ff ff ff          callq  64a <func>
 683:   89 45 fc                mov    %eax,-0x4(%rbp)
 686:   8b 45 fc                mov    -0x4(%rbp),%eax
 689:   89 c6                   mov    %eax,%esi
 68b:   48 8d 3d a2 00 00 00    lea    0xa2(%rip),%rdi        # 734 <_IO_stdin_used+0x4>
 692:   b8 00 00 00 00          mov    $0x0,%eax
 697:   e8 84 fe ff ff          callq  520 <printf@plt>
 69c:   b8 00 00 00 00          mov    $0x0,%eax
 6a1:   c9                      leaveq
 6a2:   c3                      retq
 6a3:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 6aa:   00 00 00
 6ad:   0f 1f 00                nopl   (%rax)
```

Figure 10: Function call and disassembly

7 Q9: Read the section "Analyzing switch Statements" and explain to your classmates how to recognize a switch structure in assembly code.

switch: switch is just a sequence of if-else statements , what assembly code is doing is concatenating *cmp* - *je*(jump equal) intructions if the conditions are true al false. If the conditions are true, its going to run them, but if not, is going to set $*eip* to next *cmp* instruction, and its going to do this till the *switch* statement is done .



```
 1 #include <stdio.h>
 2
 3 int main(){
 4     int a = 10 ;
 5     switch(a){
 6         case 10:
 7             a++;
 8         case 20:
 9             a--;
10         case 100:
11             printf("a es %d \n" , a );
12         default:
13             printf("jojojo \n" );
14     }
15     return 0;
16 }
```

```
000000000000068a <main>:
 68a:   55                      push   %rbp
 68b:   48 89 e5                mov    %rsp,%rbp
 68e:   48 83 ec 10             sub    $0x10,%rsp
 692:   c7 45 fc 0a 00 00 00    movl   $0xa,-0x4(%rbp)
 699:   8b 45 fc                mov    -0x4(%rbp),%eax
 69c:   83 f8 14                cmp    $0x14,%eax
 69f:   74 0e                   je     6af <main+0x25>
 6a1:   83 f8 64                cmp    $0x64,%eax
 6a4:   74 0d                   je     6b3 <main+0x29>
 6a6:   83 f8 0a                cmp    $0xa,%eax
 6a9:   75 1e                   jne    6c9 <main+0x3f>
 6ab:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
 6af:   83 6d fc 01             subl   $0x1,-0x4(%rbp)
 6b3:   8b 45 fc                mov    -0x4(%rbp),%eax
 6b6:   89 c6                   mov    %eax,%esi
 6b8:   48 8d 3d a5 00 00 00    lea    0xa5(%rip),%rdi        # 764 <_IO_stdin_used+0x4>
 6bf:   b8 00 00 00 00          mov    $0x0,%eax
 6c4:   e8 97 fe ff ff          callq  560 <printf@plt>
 6c9:   48 8d 3d 9e 00 00 00    lea    0x9e(%rip),%rdi        # 76e <_IO_stdin_used+0xe>
 6d0:   e8 7b fe ff ff          callq  550 <puts@plt>
 6d5:   b8 00 00 00 00          mov    $0x0,%eax
 6da:   c9                      leaveq
 6db:   c3                      retq
 6dc:   0f 1f 40 00             nopl   0x0(%rax)
```

Figure 11: Switch disassemble