RIT

جامعة روتشستر الأمريكية للتكنولوجيا في دبي

A Global American University in Dubai

CSEC 202 Reverse Engineering Fundamentals

Module 0x05
Advanced Static Analysis
(1/2) & (2/2)

Eng. Emad Abu Khousa

Sections: 600 | 601 | 602

March 04, 2024

Lesson Summary

Some of the topics that are covered in this Module are:

- Understand how advanced static analysis is performed.
- Exploring IDA's disassembler functionality.
- Understanding and identifying different C constructs in assembly.
  - Recognizing C code constructs in assembly language involves understanding how high-level structures like loops, conditional statements, and function calls are translated into low-level assembly instructions.

Advanced Static Analysis

Advanced Static Analysis (ASA)

Advanced static analysis is a technique used to analyze the code and structure of malware without executing it.

- **Identifying Malware Behavior and Weaknesses for Detection:**
  - ASA can help us identify the malware's behavior and weaknesses and develop signatures for antivirus software to detect it.

- **Understanding Malware to Develop New Defense Techniques**
  - By analyzing the code and structure of malware, researchers can also better understand how it works and develop new techniques for defending against it.

Good News:

- You do not need to be a programmer or understand every assembly instruction to perform effective code analysis

Global Vs. Local Variables

Global vs. Local Variables

- **Global Variables:** can be accessed and used by any function in a program.

- **Local Variables:** can be accessed only by the function in which they are defined.

Both global and local variables are declared similarly in C, but they look completely different in assembly.

Global vs. Local Variables

```
#include <stdio.h>

// Global variables
int x = 1;
int y = 2;

void main() {
    // Add y to x
    x = x + y;

    // Print the result
    printf("Total = %d\n", x);
}
```

Global vs. Local Variables

```
#include <stdio.h>

// Global variables
int x = 1;
int y = 2;

void main() {
    // Add y to x
    x = x + y;

    // Print the result
    printf("Total = %d\n", x)
}
```

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

The global variable x is signified by dword_40CF60, a memory location at 0x40CF60. Notice that x is changed in memory when eax is moved into dword_40CF60 at 1.

Global vs. Local Variables

```
#include <stdio.h>

// Global variables
int x = 1;
int y = 2;

void main() {
    // Add y to x
    x = x + y;

    // Print the result
    printf("Total = %d\n", x);
}
```

```
.data
        x: .long 1
        y: .long 2

.section    .rodata
        .LC0:
            .string "Total = %d\n“

.text
        main:
                1. mov eax, DWORD PTR x
                2. add eax, DWORD PTR y
                3. mov DWORD PTR x, eax
                4. push eax
                5. push OFFSET FLAT:.LC0
                6. call printf
                7. add esp, 8
                8. ret
```

Global vs. Local Variables

```c
#include <stdio.h>

// Global variables
int x = 1;
int y = 2;

void main() {
    // Add y to x
    x = x + y;

    // Print the result
    printf("Total = %d\n", x);
}
```

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const cha
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
mov     edx, cs:x
mov     eax, cs:y
add     eax, edx
mov     cs:x, eax
mov     eax, cs:x
mov     esi, eax
lea     rax, format       ; "Total = %d\n"
mov     rdi, rax          ; format
mov     eax, 0
call    _printf
nop
pop     rbp
retn
; } // starts at 1139
main endp

_text ends
```

Global vs. Local Variables

```
#include <stdio.h>

int main() {

// local variables
    int x = 0;
    int y = 1;
    x = x + y;
    printf("Total = %d\n", x);
    return 0;
}
```

```
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
```

The local variable x is located on the stack at a constant **offset relative to ebp**. The memory location [ebp-4] is used consistently throughout this function to reference the local variable x. This tells us that ebp-4 is a stack-based local variable that is referenced only in the function in which it is defined.

Global vs. Local 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
```

Here's a description of what each line is doing:

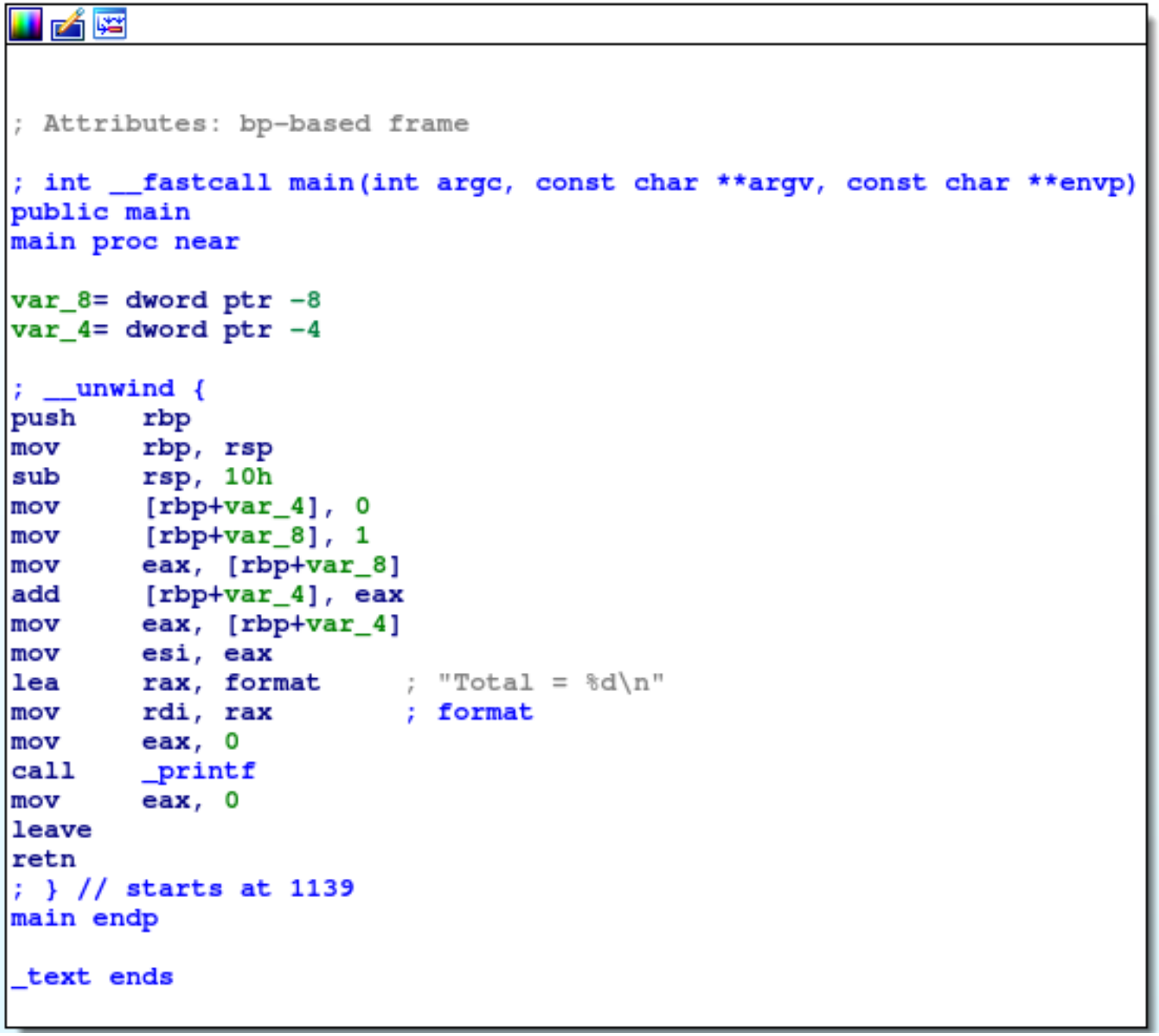
- **mov dword ptr [ebp-4], 0:** Sets the value of the local variable x (which is at [ebp-4] on the stack) to 0.
- **mov dword ptr [ebp-8], 1**: Sets the value of the local variable y (which is at [ebp-8] on the stack) to 1.
- **mov eax, [ebp-4]:** Loads the value of x from the stack into the eax register.
- **add eax, [ebp-8]:** Adds the value of y to eax, which now contains the sum x + y.
- **mov [ebp-4], eax:** Stores the result back into the location of x on the stack, effectively updating the value of x to x + y.
- **mov ecx, [ebp-4]:** Moves the updated value of x into the ecx register in preparation for the printf call.
- **push ecx**: Pushes the value of ecx (the sum of x and y) onto the stack as the argument to printf.
- **push offset aTotalID:** Pushes the address of the format string "Total = %d\n" onto the stack.
- **call printf:** Calls the printf function, which will use the format string and the argument to print "Total = 1" to the standard output.

Global vs. Local Variables

```
section .data
    format db "Total = %d", 10, 0          ; The format string for printf +newline and null terminator

section .text
    global main
    extern printf

main:
    push ebp                               ; Start of function prologue
    mov ebp, esp
    sub esp, 8                             ; Allocate stack space for x and y

    mov dword [ebp-4], 0                   ; int x = 0; (local variable x at [ebp-4])
    mov dword [ebp-8], 1                   ; int y = 1; (local variable y at [ebp-8])

    mov eax, [ebp-4]                       ; Load x into eax
    add eax, [ebp-8]                       ; Add y to eax
    mov [ebp-4], eax                       ; Store the result back in x

    push eax                               ; Push the sum (new x value) as the argument for printf
    push dword format                      ; Push the address of the format string
    call printf                            ; Call printf function
    add esp, 8                             ; Clean up the stack (two 4-byte arguments)

    mov eax, 0                             ; Return 0
    leave                                  ; Equivalent to 'mov esp, ebp' followed by 'pop ebp'
    ret                                     ; Return from main
```

Global vs. Local Variables

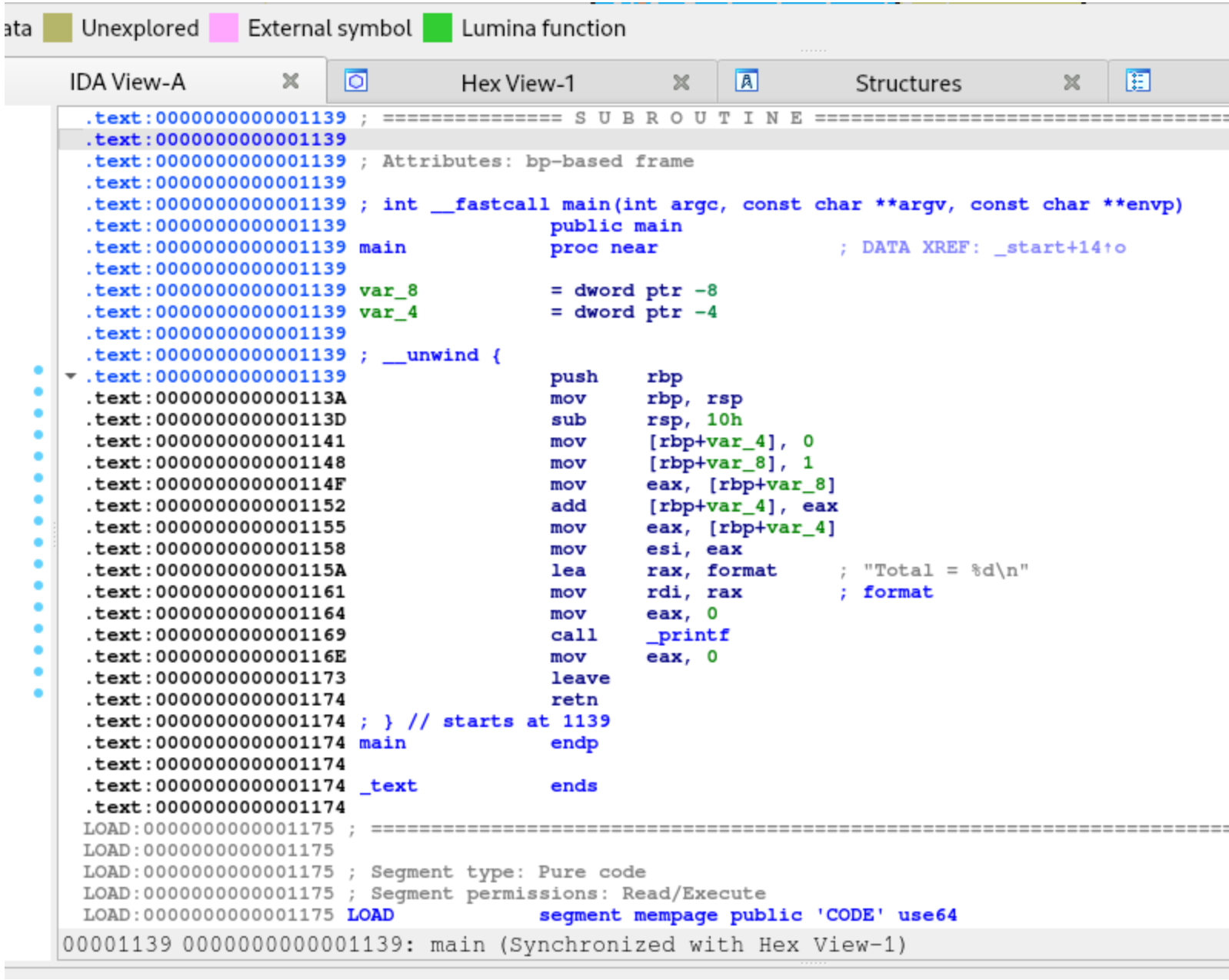
```
#include <stdio.h>

int main() {

// local variables
    int x = 0;
    int y = 1;
    x = x + y;
    printf("Total = %d\n", x);
    return 0;
}
```



```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], 0
mov     [rbp+var_8], 1
mov     eax, [rbp+var_8]
add     [rbp+var_4], eax
mov     eax, [rbp+var_4]
mov     esi, eax
lea     rax, format     ; "Total = %d\n"
mov     rdi, rax        ; format
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 1139
main endp

_text ends
```

Global vs. Local Variables

```
#include <stdio.h>

int main() {

// local variables
    int x = 0;
    int y = 1;
    x = x + y;
    printf("Total = %d\n", x);
    return 0;
}
```



Global vs. Local Variables

"; Attributes: bp-based frame": This comment indicates that the function uses a frame pointer ("bp" stands for base pointer, which is "rbp" in x86-64) to reference local variables and parameters. This is a traditional method for managing the stack frame, which can make the debugging process easier.

"; int __fastcall main(int argc, const char **argv, const char **envp)": This is a comment describing the prototype of the "main" function in C, which is the entry point of a C program. **The "__fastcall"** keyword specifies a calling convention where arguments **are typically passed in registers instead** of being pushed onto the stack.

common calling conventions

CDECL: Short for "C declaration," this is the default calling convention for C and C++ programs on x86 architectures. **The caller cleans up the stack**, and **parameters are pushed onto the stack in right-to-left** order.

STDCALL: This convention is often used by Win32 API functions. It is similar to CDECL in that **parameters are pushed right-to-left**, but the **callee is responsible for cleaning up the stack**. This is useful for functions with a fixed number of parameters.

FASTCALL: This calling convention attempts to minimize the number of accesses to the stack by **passing the first two (on x86) or four (on x86-64 Microsoft) or six (on x86-64 Linux) on integer or pointer arguments in registers,** which is generally faster than using the stack. **The callee is responsible for cleaning up the stack**.

THISCALL: This calling convention is used by C++ for non-static member functions. The this pointer is passed in a register (usually ECX on x86), and other arguments are passed on the stack. This is a Microsoft-specific calling convention not used in System V AMD64 ABI which is followed by Unix-like systems.

Global vs. Local Variables

"public main": This directive makes the "main" label public, meaning it can be accessed from other files or modules. In the context of an executable, it's not typically necessary, but this would be relevant if the symbol needed to be exposed for linkage.

"main proc near": This declares the start of the "main" procedure. The "near" keyword refers to a type of call within the same code segment, as opposed to "far", which would be used for calls between different segments.

"; DATA XREF: _start+14↑o": This comment is likely generated by a disassembler, indicating a cross-reference (**XREF**) from the "_start" label, which is usually the entry point of the program set by the linker. The "+14" suggests an offset from "_start", and the arrow and "o" indicate the direction and type of the reference.

Global vs. Local Variables

```
.text:0000000000001050 ; Attributes: noreturn fuzzy-sp
.text:0000000000001050
.text:0000000000001050                 public _start
.text:0000000000001050 _start          proc near               ; DATA XREF: LOAD:0000000000000018↑o
.text:0000000000001050 ; __unwind {
.text:0000000000001050                 xor     ebp, ebp
.text:0000000000001052                 mov     r9, rdx         ; rtld_fini
.text:0000000000001055                 pop     rsi             ; argc
.text:0000000000001056                 mov     rdx, rsp        ; ubp_av
.text:0000000000001059                 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:000000000000105D                 push    rax
.text:000000000000105E                 push    rsp             ; stack_end
.text:000000000000105F                 xor     r8d, r8d        ; fini
.text:0000000000001062                 xor     ecx, ecx        ; init
.text:0000000000001064                 lea     rdi, main       ; main
.text:000000000000106B                 call    cs:__libc_start_main_ptr
.text:0000000000001071                 hlt
.text:0000000000001071 ; } // starts at 1050
.text:0000000000001071 _start          endp
.text:0000000000001071
.text:0000000000001071 ; ---------------------------------------------------------------------------
.text:0000000000001072                 align 20h
```

Global vs. Local Variables

"var_8 = dword ptr -8": **This defines a local variable at 8 bytes below the current base pointer ("rbp") on the stack. The "dword ptr" indicates that this variable is a double word (32 bits or 4 bytes) in size.**

"var_4 = dword ptr -4": **Similarly, this defines another local variable at 4 bytes below the base pointer on the stack, which is also a double word in size.**

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], 0
mov     [rbp+var_8], 1
mov     eax, [rbp+var_8]
add     [rbp+var_4], eax
mov     eax, [rbp+var_4]
mov     esi, eax
lea     rax, format     ; "Total = %d\n"
mov     rdi, rax        ; format
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 1139
main endp

_text ends
```

Disassembly Challenge

Challenge: Reverse Engineering from Assembly to Pseudocode

Objective:
You are presented with a disassembled output of a simple C code snippet. Your task is to deduce the functionality of this code snippet and translate it back into a high-level language (pseudocode). This challenge requires you to apply your understanding of assembly language and reverse engineering principles.

Disassembled Code Snippet:

```
mov dword ptr [ebp-4],1
mov eax, dword ptr [ebp-4]
mov dword ptr [ebp-8],eax
```

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

Phase 1: Understanding Assembly Instructions

Analyze Each Instruction: Begin by understanding what each assembly instruction does. Identify operations like moving data between registers and memory locations.

Identify Variables: Recognize that memory locations (e.g., [ebp-4], [ebp-8]) are used to store variables in high-level languages. Determine the role of these variables based on their usage.

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

Phase 2: Mapping to High-Level Constructs

Variable Declaration and Initialization: Understand that mov dword ptr [ebp-4],1 is likely initializing a variable. Map this to a variable declaration and initialization in C.

Assignment and Manipulation: For instructions that move data between locations, determine how they correspond to variable assignments or manipulations in C.

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

1. mov dword ptr [**x**],1
2. mov eax, dword ptr [**x**]
3. mov dword ptr [**y**],eax

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

1. mov dword ptr [**x**],1
2. mov eax, dword ptr [**x**]
3. mov dword ptr [**y**],eax

x = 1

eax = x

y= eax

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

Phase 3: Reconstructing the Logic

Build the Logical Flow: With an understanding of what each assembly instruction represents, start piecing together the logical flow of the original C code. Consider the sequence of operations and their implications on the variables' states.

Draft Pseudocode: Begin drafting pseudocode that reflects the logic you've deduced. Ensure your pseudocode accurately represents the operations and sequence found in the assembly..

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

1. mov dword ptr [**x**],1
2. mov eax, dword ptr [**x**]
3. mov dword ptr [**y**],eax

x = 1

eax = x

y= eax

x = 1

y= x

Disassembly Challenge

1. mov dword ptr [ebp-4],1
2. mov eax, dword ptr [ebp-4]
3. mov dword ptr [ebp-8],eax

```
int main() {
    int x = 1;
    int y;
    y = x
    return 0;
}
```

One more? Disassembly Challenge

somefunc:

```
1.    push ebp
2.    mov ebp, esp
3.    sub esp, 4
4.    mov eax, [ebp+8]
5.    add eax, [ebp+12]
6.    add eax, [ebp+16]
7.    mov [ebp-4], eax
8.    mov eax, [ebp-4]
9.    mov esp, ebp
10.   pop ebp
11.   ret
```

Disassembly Challenge

somefunc:

```
1.    push ebp                  ; set up stack frame
2.    mov ebp, esp              ;
3.    sub esp, 4                ; allocate space for local variables
4.    mov eax, [ebp+8]          ; load a into eax
5.    add eax, [ebp+12]         ; add b to eax
6.    add eax, [ebp+16]         ; add c to eax
7.    mov [ebp-4], eax          ; store result in d
8.    mov eax, [ebp-4]          ; prepare return value
9.    mov esp, ebp              ; clean up stack frame
10.   pop ebp                   ;
11.   ret                       ; return
```

Disassembly Challenge

```
somefunc:
1.      push ebp
2.      mov ebp, esp
3.      sub esp, 4
4.      mov eax, [ebp+8]
5.      add eax, [ebp+12]
6.      add eax, [ebp+16]
7.      mov [ebp-4], eax
8.      mov eax, [ebp-4]
9.      mov esp, ebp
10.     pop ebp
11.     ret
```

```
1.  int somefunc(int a, int b, int c) {
2.      int d;

3.      d = a + b + c;

4.      return(d);
5.  }
```

Arithmetic Operations

Arithmetic Operations

- **add eax,42** ; same as eax = eax+42
- **add eax,ebx** ; same as eax = eax+ebx
- **add [ebx],42** ; adds 42 to the value in address specified by ebx
- **sub eax, 64h** ; subtracts hex value 0x64 from eax, same as eax = eax-0x64
- **inc eax** ; same as eax = eax+1
- **dec ebx** ; same as ebx = ebx-1
- **mul ebx** ;ebx is multiplied with eax and result is stored in EDX and EAX
- **mul bx** ;bx is multiplied with ax and the result is stored in DX and AX
- **div ebx** ; divides the value in EDX:EAX by EBX

Arithmetic Operations - MUL

http://carlosrafaelgn.com.br/Asm86/

Multiplicand	Multiplier	Product
al	reg/mem8	ax
ax	reg/mem16	dx:ax
eax	reg/mem32	edx:eax

The **mul** instruction in multiplies unsigned integers. The operand of mul can be an 8-bit, 16-bit, 32-bit, or 64-bit register or memory location, depending on the operation size. The **mul** instruction implicitly uses the AX, EAX, or RAX register (for 16-bit, 32-bit, and 64-bit operations, respectively) as one of its operands.

```
mov al, 2
mov ah, 3
mul al

mov al, 2
mov bh, 3
mul bh

mov al, 2
mov ah, 3
mul ax
```

Arithmetic Operations -MUL

```
mov edx, 0x98765432
mov eax, 12345h
mov ebx, 1000h
mul ebx      ; EDX:EAX = 00000000 12345000h

mov edx, 0x98765432
mov eax, 12345h
mov ebx, 100000h
mul ebx      ; EDX:EAX = 00000012 34500000h
```

http://carlosrafaelgn.com.br/Asm86/

Arithmetic Operations - MUL

What will be the hexadecimal values of DX & AX, after the following instructions execute?

```
mov ax, 1234h
mov bx, 100h
mul bx                                  ; dx:ax=ax*bx
; 0012:3400=1234*00100
```

http://carlosrafaelgn.com.br/Asm86/

Arithmetic Operations -MUL

What will be the hexadecimal values of DX & AX, after the following instructions execute?

```
mov eax, 00128765h
mov ecx, 10000h
mul ecx
```

http://carlosrafaelgn.com.br/Asm86/

Arithmetic Operations - MUL

What will be the hexadecimal values of DX & AX, after the following instructions execute?

```
mov eax, 0xffffffff
mov ebx, 0x10
mul ebx
```

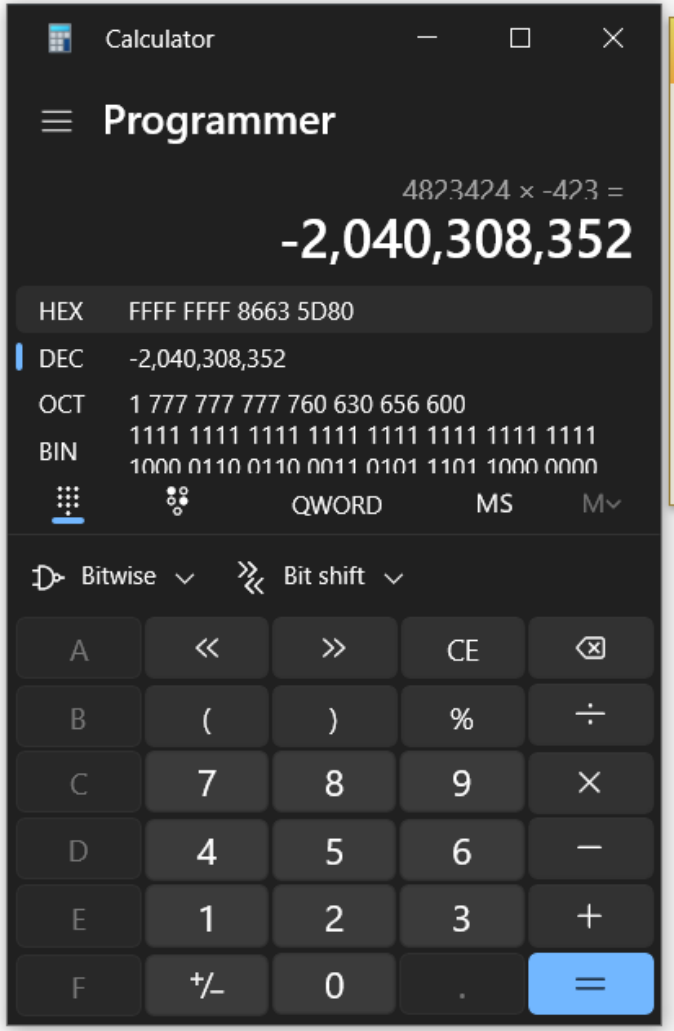
http://carlosrafaelgn.com.br/Asm86/

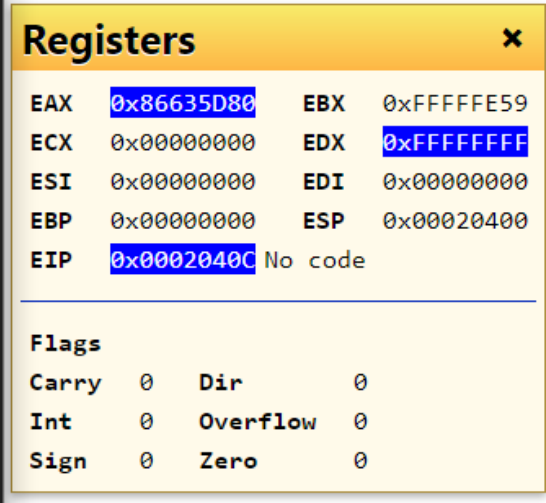
Arithmetic Operations - iMUL

IMUL (signed integer multiply) multiplies an 8-, 16-, or 32 bit signed operand by either AL, AX, or EAX
Preserves the sign of the product by sign-extending it into the upper half of the destination register

Arithmetic Operations - iMUL

```
;Multiply 4,823,424 *  -423:
mov eax, 4823424
mov ebx, -423
imul ebx
; EDX:EAX = FFFFFFFF86635D80h
```



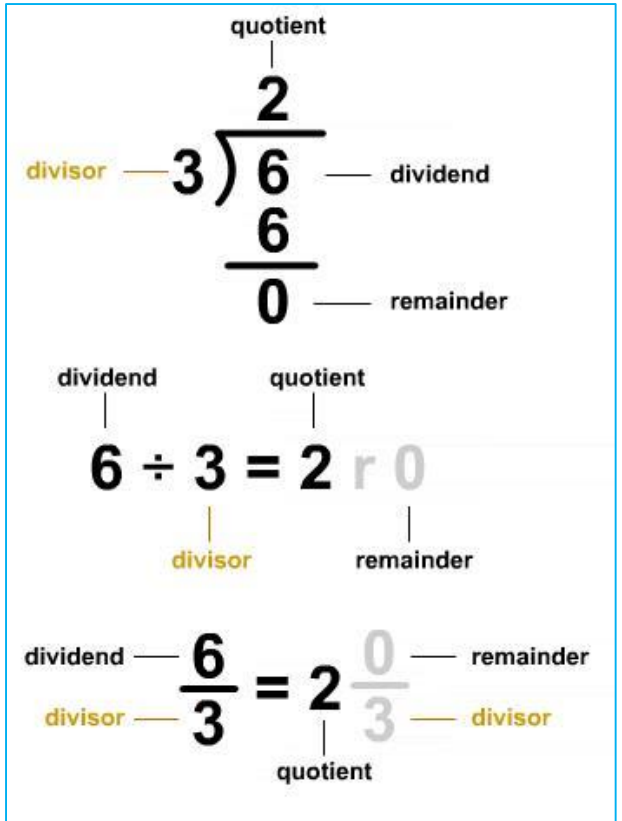


http://carlosrafaelgn.com.br/Asm86/

Arithmetic Operations - DIV Instruction

The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
A single operand is supplied (register or memory operand), which is assumed to be the divisor

Dividend	Divisor	Quotient	Remainder
ax	reg/mem8	al	ah
dx:ax	reg/mem16	ax	dx
edx:eax	reg/mem32	eax	edx



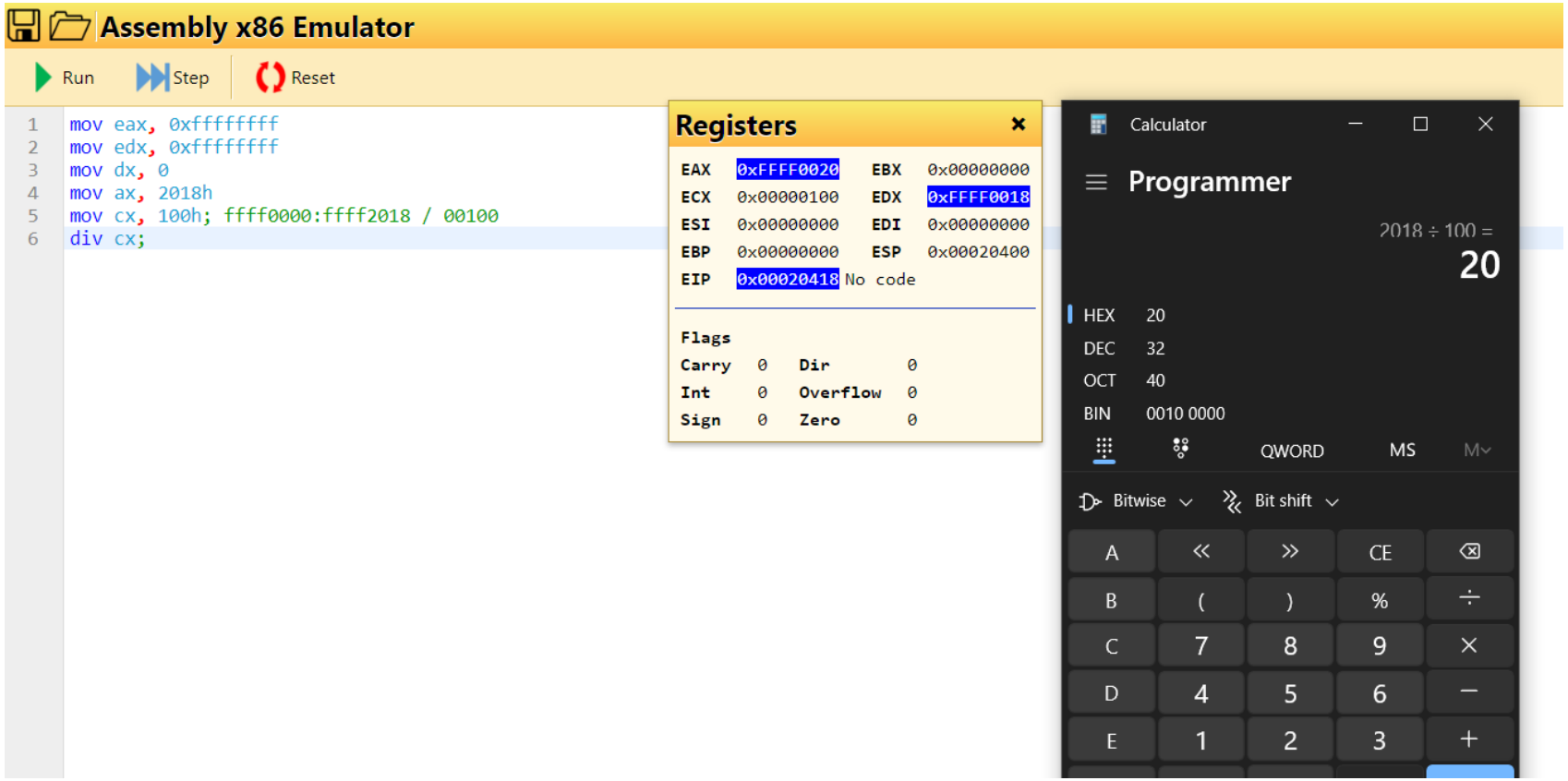
Arithmetic Operations - DIV Instruction

What will be the hexadecimal values of DX & AX, after the following instructions execute?

```
mov eax, 0xffffffff
mov edx, 0xffffffff
mov dx, 0
mov ax, 2018h
mov cx, 100h        ; ffff0000:ffff2018 / 00100
div cx              ; 0000:2018 / 00100 -> dx:ax / cx; q->ax, R->dx
```

http://carlosrafaelgn.com.br/Asm86/

Arithmetic Operations - DIV Instruction



http://carlosrafaelgn.com.br/Asm86/

Arithmetic Operations - DIV Instruction

What will be the hexadecimal values of DX and AX after the following instructions execute?

```
mov dx,0087h
mov ax,6000h
mov bx,100h
div bx;
```

0087:6000 / 0100 -> ax=8760

Dividend	Divisor	Quotient	Remainder
ax	reg/mem**8**	al	ah
dx:ax	reg/mem**16**	ax	dx
edx:eax	reg/mem**32**	eax	edx

http://carlosrafaelgn.com.br/Asm86/

Disassembly Challenge

Challenge: Reverse Engineering from Assembly to Pseudocode

The following is a disassembled output of a simple C program. Can you figure out what this program does, and can you translate it back to a pseudocode?

.

Disassembled Code Snippet:

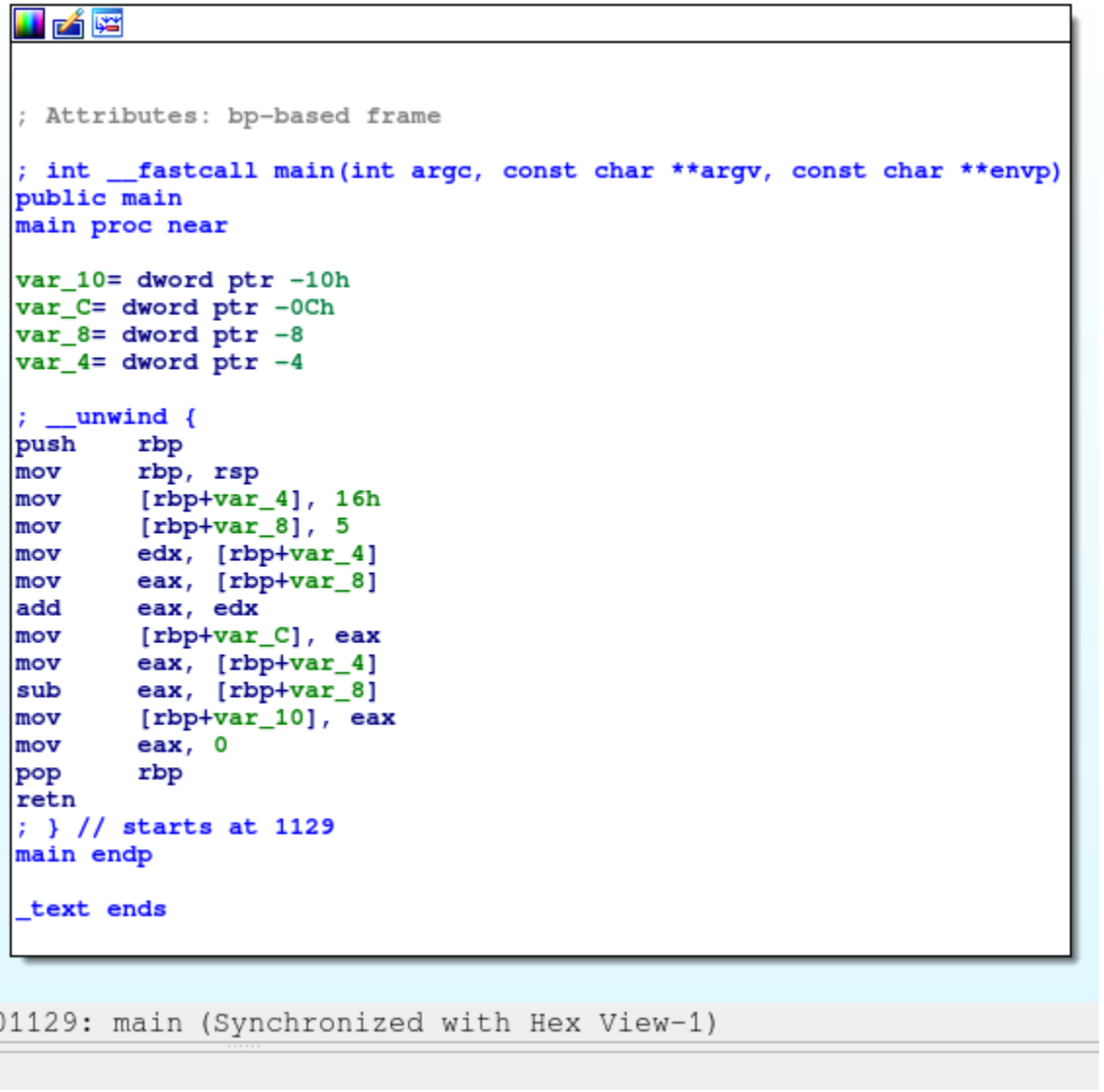
```
mov dword ptr [ebp-4], 16h
mov dword ptr [ebp-8], 5
mov eax, [ebp-4]
add eax, [ebp-8]
mov [ebp-0Ch], eax
mov ecx, [ebp-4]
sub ecx, [ebp-8]
mov [ebp-10h], ecx
```

Disassembly Challenge

```
0000117d <main>:
    117d:       55                          push   ebp
    117e:       89 e5                       mov    ebp,esp
    1180:       83 ec 10                    sub    esp,0x10
    118d:       c7 45 fc 16 00 00 00        mov    DWORD PTR [ebp-0x4],0x16
    1194:       c7 45 f8 05 00 00 00        mov    DWORD PTR [ebp-0x8],0x5
    119b:       8b 55 fc                    mov    edx,DWORD PTR [ebp-0x4]
    119e:       8b 45 f8                    mov    eax,DWORD PTR [ebp-0x8]
    11a1:       01 d0                        add    eax,edx
    11a3:       89 45 f4                    mov    DWORD PTR [ebp-0xc],eax
    11a6:       8b 45 fc                     mov    eax,DWORD PTR [ebp-0x4]
    11a9:       2b 45 f8                    sub    eax,DWORD PTR [ebp-0x8]
    11ac:       89 45 f0                    mov    DWORD PTR [ebp-0x10],eax
    11af:      b8 00 00 00 00               mov    eax,0x0
    11b4:       c9                          leave
    11b5:       c3                          ret
```

Disassembly Challenge

mov dword ptr [ebp-4], 16h
mov dword ptr [ebp-8], 5
mov eax, [ebp-4]
add eax, [ebp-8]
mov [ebp-0Ch], eax
mov ecx, [ebp-4]
sub ecx, [ebp-8]
mov [ebp-10h], ecx



```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], 16h
mov     [rbp+var_8], 5
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
mov     [rbp+var_C], eax
mov     eax, [rbp+var_4]
sub     eax, [rbp+var_8]
mov     [rbp+var_10], eax
mov     eax, 0
pop     rbp
retn
; } // starts at 1129
main endp

_text ends
```

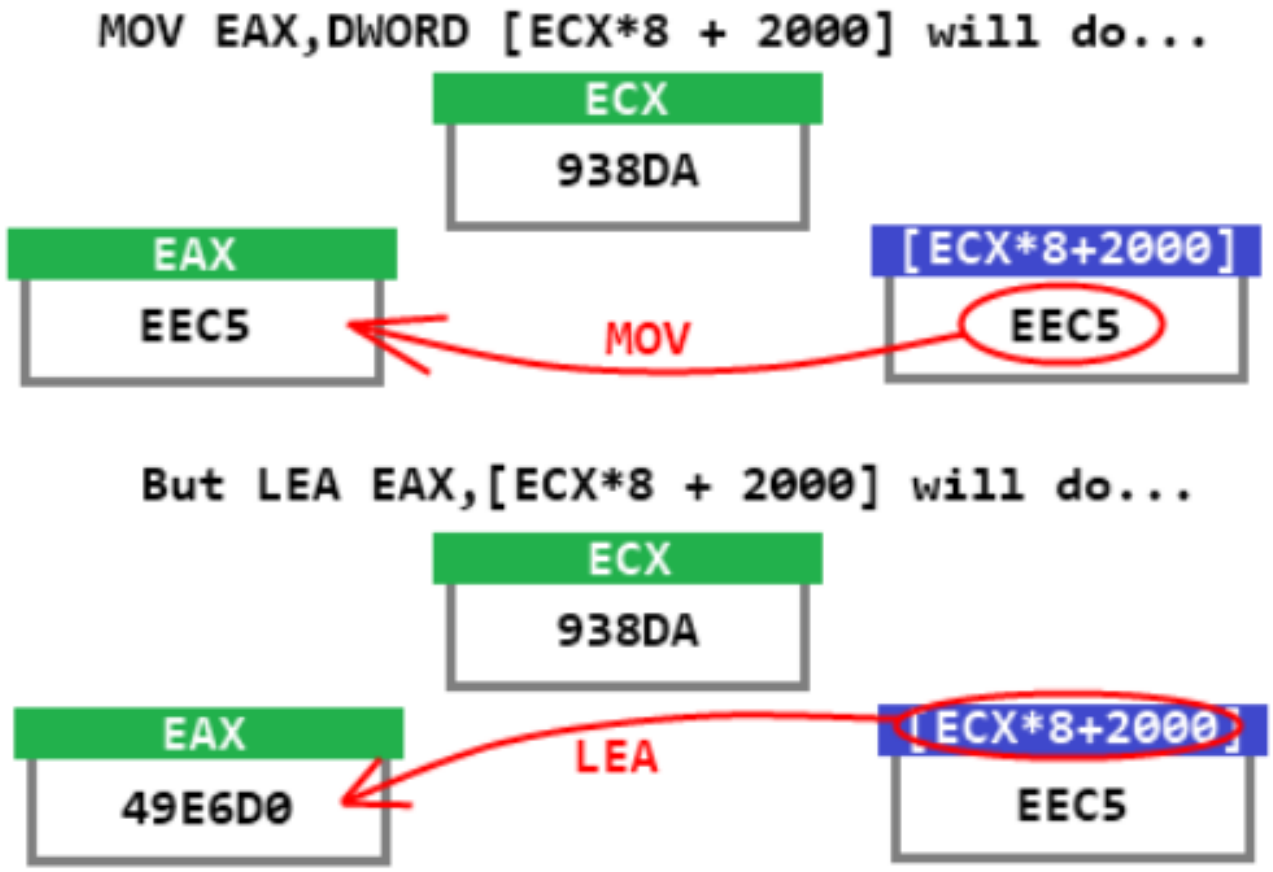
)001129: main (Synchronized with Hex View-1)

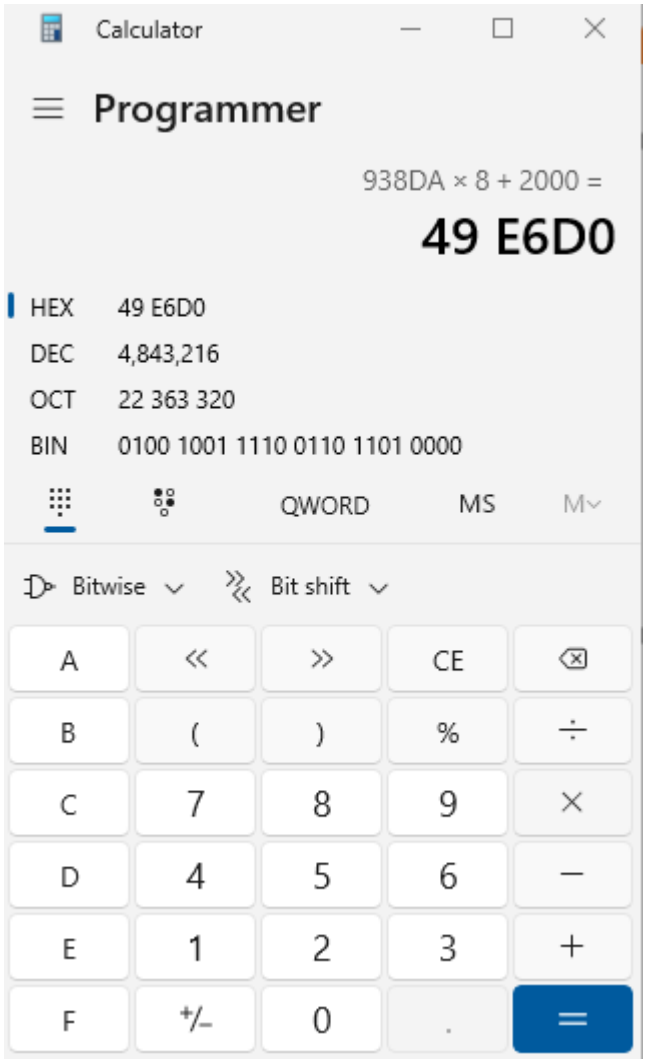
Disassembly Challenge

```
#include <stdio.h>

int main() {
    int var1 = 0x16;
    int var2 = 5;
    int sum = var1 + var2;
    int difference = var1 - var2;

    return 0;
}
```

MOV vs LEA

MOV vs LEA


MOV EAX,DWORD [ECX*8 + 2000] will do...
ECX
938DA
EAX
EEC5
MOV
[ECX*8+2000]
EEC5
But LEA EAX,[ECX*8 + 2000] will do...
ECX
938DA
EAX
49E6D0
LEA
[ECX*8+2000]
EEC5


Calculator
Programmer
938DA × 8 + 2000 =
49 E6D0
HEX 49 E6D0
DEC 4,843,216
OCT 22 363 320
BIN 0100 1001 1110 0110 1101 0000
QWORD
MS
Bitwise
Bit shift
A
CE
B
(
)
%
C
7
8
9
D
4
5
6
E
1
2
3
F
+/-
0
.

MOV vs LEA

mov Instruction

Purpose: The mov instruction is used to copy data from one location to another. The source remains unchanged.

Operation: It can move immediate values into registers, copy data between registers, or move data between registers and memory (both directions).

Example:

```
mov eax, 5      ; Copies the value 5 into the EAX register.
mov ebx, eax    ; Copies the value from EAX into EBX. Now EBX = 5.
```

Note: mov does not affect the flags register.

Restrictions on MOV in Assembly Language

1. **Segment Registers in Protected Mode:** Cannot directly manipulate the **CS register** with **mov** for safety and security reasons. **Others? Yes = mov can transfer data to DS, ES, FS, GS, and SS registers.**

2. **Instruction Pointer (EIP/RIP):** Direct movement into **EIP/RIP** using **mov is not permitted.**

3. **Flags Register (EFLAGS/RFLAGS):**Cannot use **mov** directly with flags register.

4. **Control and Debug Registers:** Requires specialized instructions for movement. Regular **mov** syntax not applicable for CR0, CR2, CR3, CR4, etc., and debug registers

5. **…. What else?**

MOV vs LEA

lea Instruction

- Purpose: The lea instruction is used to load the effective address of a memory operand into a register.
- **While often used for address calculation, it can also perform certain arithmetic operations.**
- Operation: It calculates the address of a memory location specified by its operand and stores this address in a register. The memory is not accessed.

Example:

lea eax, [ebx+4*ecx] ; Calculates the address ebx + 4*ecx and stores it in EAX.

In this example, if ebx is 1000 and ecx is 2, eax will hold 1008 after the instruction. It essentially performs the calculation without accessing the memory at that calculated address.

MOV vs LEA

Key Differences

1. **Data vs. Address:** mov is used for moving data, whereas lea is used for loading the address of the data (i.e., the memory location) into a register.
2. **Arithmetic Operations:** lea can be cleverly used to perform certain arithmetic operations (like addition or scaled addition) without affecting the flags register, whereas mov is strictly for data transfer.
3. **Flags Register:** mov does not affect the flags register, and similarly, lea also does not affect the flags register, but its ability to perform arithmetic without altering flags can be particularly useful in specific contexts.

Disassembly Challenge

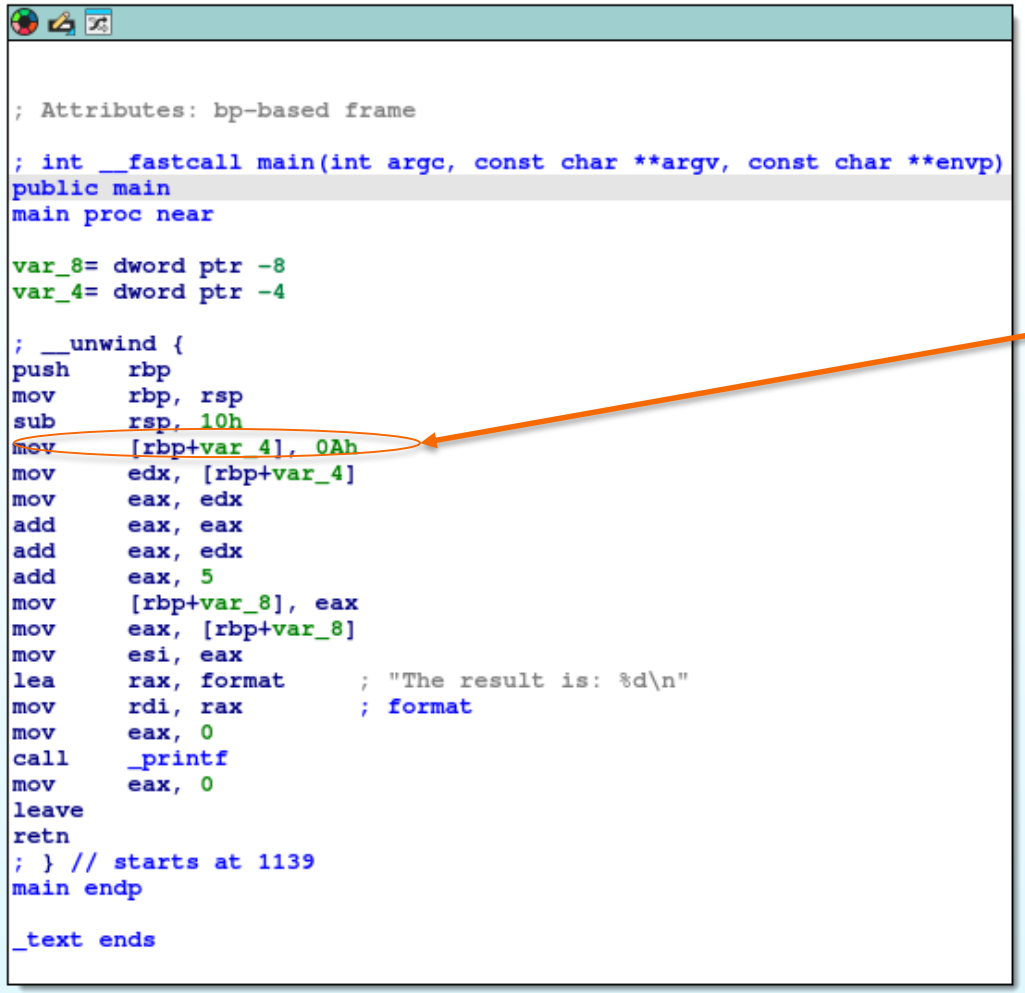
```
section .data
    x dd 10
    format db "The result is: %d", 10, 0

section .text
    global main
    extern printf

main:
    push ebp
    mov ebp, esp
    mov eax, [x]
    lea ebx, [eax*2 + eax + 4]
    push ebx
    push dword format
    call printf
    add esp, 8
    mov esp, ebp
    pop ebp
    ret
```

Disassembly Challenge

```
section .data
    x dd 10
    format db "The result is: %d", 10, 0

section .text
    global main
    extern printf

main:
    push ebp
    mov ebp, esp
    mov eax, [x]
    lea ebx, [eax*2 + eax + 4]
    push ebx
    push dword format
    call printf
    add esp, 8
    mov esp, ebp
    pop ebp
    ret
```

```
#include <stdio.h>

int x = 10;
int main() {
        int y;
        y = 3 * x + 4;

    printf("The result is: %d\n", y);

    return 0;
}
```

Disassembly Challenge (IDA → C)! Same?



```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], 0Ah
mov     edx, [rbp+var_4]
mov     eax, edx
add     eax, eax
add     eax, edx
add     eax, 5
mov     [rbp+var_8], eax
mov     eax, [rbp+var_8]
mov     esi, eax
lea     rax, format     ; "The result is: %d\n"
mov     rdi, rax        ; format
mov     eax, 0
call    _printf
mov     eax, 0
leave
retn
; } // starts at 1139
main endp

_text ends
```

```
#include <stdio.h>

int main() {
        int x = 10;
        int y;
        y = 3 * x + 4;

        printf("The result is: %d\n", y);

        return 0;
}
```

Is it global?

Bitwise Operations

Bitwise Operations = Boolean/Logical Operations

- **and eax,ebx**
  - Performs a bitwise AND operation on the values stored in EAX and EBX.
    The result is saved in EAX.

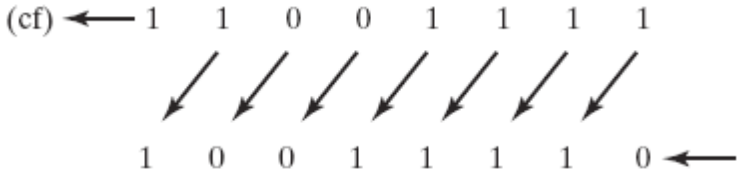
- **or eax,ebx**
  - Performs a bitwise OR operation on the values stored in EAX and EBX.
    The result is saved in EAX.

Bitwise Operations = Boolean/Logical Operations

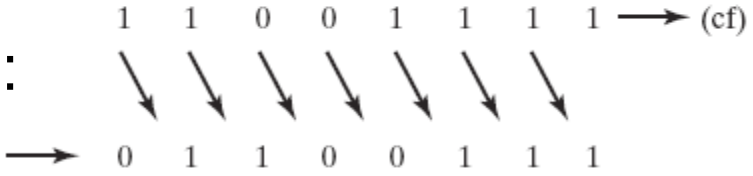
- **xor eax,ebx**
  - Performs a bitwise XOR operation on the values stored in EAX and EBX.
    The result is saved in EAX
- **test eax, edx**
  - Performs a **non-destructive** AND operation on the values in EAX and EDX.
    If the result is zero, sets the zero flag to 1
- **not**
  - **Flips a 1 to a 0 and a 0 to a 1.**

Logical: Shift operation

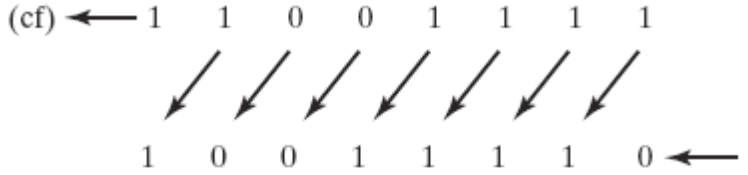
- Logical Shift Left (SHL) or Shift Left (SAL):
  - SHL AL, 1



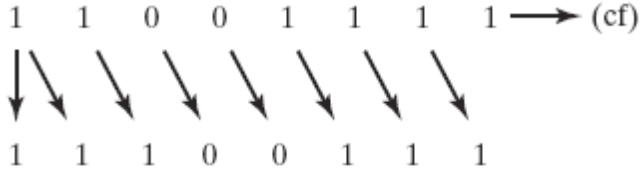
- Logical Shift Right (SHR):
  - SHR AL, 1



- Arithmetic Shift Left (SAL):
  - SAL AL, 1 (Identical to SHL)

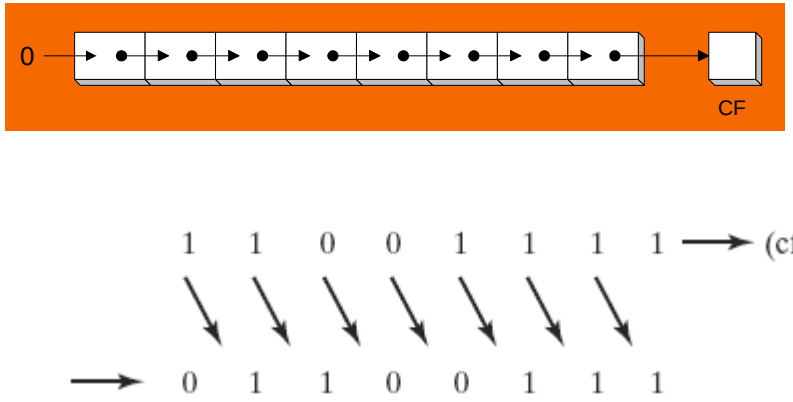


- Arithmetic Shift Right (SAR):
  - SAR AL, 1

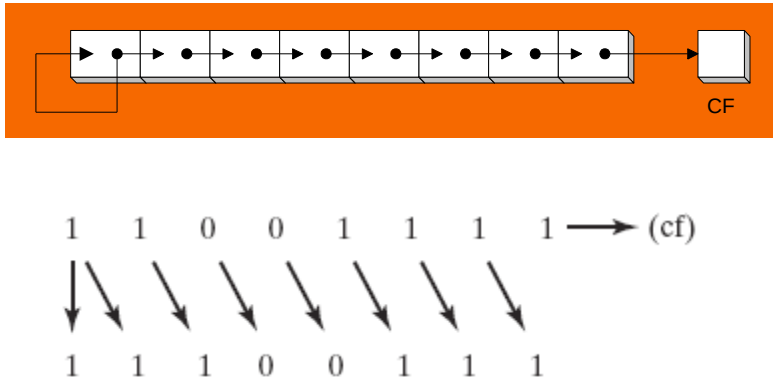


Logical: Shift operation

- A logical shift (unsigned shift) fills the newly created bit position with a zero:



- An arithmetic shift (signed shift) fills the newly created bit position with a copy of the number's sign bit:



Disassembly Challenge

Challenge: Reverse Engineering from Assembly to Pseudocode

The following is a disassembled output of a simple C program. Can you figure out what this program does, and can you translate it back to a pseudocode?

What is the result?

```
section .data
        format db "Result: %d", 10, 0  ;
section .text
        global _start
        extern printf

_start:
        mov eax, 5
        shl eax, 1
        shr eax, 1
        shl eax, 3
        shr eax, 2
        push eax
        push
        call printf
        add esp, 8
```

Disassembly Challenge

Shift right 1 (>>1) = divide by 2
Shift left 1(<<1) = multiply by 2

Before:	0 0 0 0 0 1 0 1	= 5
After:	0 0 0 0 1 0 1 0	= 10

```
section .data
        format db "Result: %d", 10, 0  ;
section .text
        global _start
        extern printf

_start:
        mov eax, 5
        shl eax, 1
        shr eax, 1
        shl eax, 3
        shr eax, 2
        push eax
        push
        call printf
        add esp, 8
```

Disassembly Challenge

Shift right 1 (>>1) = divide by 2
Shift left 1(<<1) = multiply by 2

```
int main() {
    int x = 5;
    x = x * 2;
    x = x / 2;
    x = x * 8;
    x = x / 4;
    printf(" Result: %d\n", x);
    return 0;
}
```

```
section .data
        format db "Result: %d", 10, 0  ;
section .text
        global _start
        extern printf

_start:
        mov eax, 5
        shl eax, 1
        shr eax, 1
        shl eax, 3
        shr eax, 2
        push eax
        push
        call printf
        add esp, 8
```

Functions

Disassembly Challenge

```
section .data
output_format db "the function returned the number %d", 10, 0

section .text
global _main
extern _printf
```

```
_adder:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    mov edx, [ebp+12]
    add eax, edx
    pop ebp
    ret
```

```
_main:
    push ebp
    mov ebp, esp
    sub esp, 8

    mov dword [ebp-4], 1
    mov dword [ebp-8], 2
    push dword [ebp-8]
    push dword [ebp-4]
    call _adder
    add esp, 8

    push eax
    push output_format
    call _printf
    add esp, 8

    mov esp, ebp
    pop ebp
    ret
```

Disassembly Challenge

```
int adder(int a, int b)
{
        return a+b;
}

void main()
{
        int x = 1;
        int y = 2;
        printf("the function returned the number %d\n", adder(x,y));
}
```

```
section .data
output_format db "the function returned the number %d", 10, 0

section .text
global _main
extern _printf

_adder:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]
    mov edx, [ebp+12]
    add eax, edx
    pop ebp
    ret

_main:
    push ebp
    mov ebp, esp
    sub esp, 8

    mov dword [ebp-4], 1
    mov dword [ebp-8], 2
    push dword [ebp-8]
    push dword [ebp-4]
    call _adder
    add esp, 8

    push eax
    push output_format
    call _printf
    add esp, 8

    mov esp, ebp
    pop ebp
    ret
```

Branching And Conditionals

Branching And Conditionals

- Branching instructions **transfer the control of execution** to a different memory address. {**change EIP**}
- To perform branching, **jump instructions** are typically used in the assembly language.
- There are two kinds of jumps:
  1. **Unconditional**: **jmp <jump address>**
  2. **Conditional: jcc < jump address >**
     - If a certain condition is met, one path is followed; if not, another path is followed.{The cc in the preceding format represents conditions.}

Branching And Conditionals

Conditional Jumps:

In conditional jumps, control is transferred to a memory address based on a specific condition.

- **Flag Alteration:** These jumps require instructions that can set or clear flags, achievable through arithmetic or bitwise operations.
- The two most popular conditional instructions are **test** and **cmp**.

Branching And Conditionals

TEST Instruction:

- Performs a bitwise **AND** operation on the operands.
- Alters the flags based on the operation outcome without storing the result.
- Example:

test eax, eax

sets the zero flag (**zf=1**) if eax is zero, as ANDing 0 with 0 yields 0.

Branching And Conditionals

CMP Instruction:

- Subtracts the second operand (source) from the first operand (destination).
- Alters the flags based on the outcome but does not store the result.
- Example: **cmp eax, 5** will set the zero flag (zf=1) if eax is 5, indicating a subtraction result of zero.

Branching And Conditionals

CMP Instruction:

- The cmp instruction is identical to the sub instruction; however, the operands are not affected.
- The cmp instruction is used only to set the flags.
- The **zero flag** and **carry flag** (CF) may be changed as a result of the cmp instruction.

CMP= dst - src

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

Conditional Jumps

Instruction	Description	Aliases	Flags
`jz`	jump if zero	`je`	`zf=1`
`jnz`	jump if not zero	`jne`	`zf=0`
`jl`	jump if less	`jnge`	`sf=1`
`jle`	jump if less or equal	`jng`	`zf=1 or sf=1`
`jg`	jump if greater	`jnle`	`zf=0 and sf=0`
`jge`	jump if greater or equal	`jnl`	`sf=0`
`jc`	jump if carry	`jb, jnae`	`cf=1`
`jnc`	jump if not carry	`jnb, jae`	.

Conditional Jumps

Instruction	Description
`jz loc`	Jump to specified location if ZF = 1.
`jnz loc`	Jump to specified location if ZF = 0.
`je loc`	Same as `jz`, but commonly used after a `cmp` instruction. Jump will occur if the destination operand equals the source operand.
`jne loc`	Same as `jnz`, but commonly used after a `cmp`. Jump will occur if the destination operand is not equal to the source operand.
`jg loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than the source operand.
`jge loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than or equal to the source operand.
`ja loc`	Same as `jg`, but an unsigned comparison is performed.
`jae loc`	Same as `jge`, but an unsigned comparison is performed.
`jl loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than the source operand.
`jle loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than or equal to the source operand.
`jb loc`	Same as `jl`, but an unsigned comparison is performed.
`jbe loc`	Same as `jle`, but an unsigned comparison is performed.
`jo loc`	Jump if the previous instruction set the overflow flag (OF = 1).
`js loc`	Jump if the sign flag is set (SF = 1).
`jecxz loc`	Jump to location if ECX = 0.

Branching And Conditionals

if/then/else

Typically an assembly sequence that:

1. **Computes the expression in the if portion**
2. **Performs a test or cmp** to ensure that CPU **flags are set**
3. **Performs** a jnz/jge/etc. and a **jmp**

```
    cmp [z], 5              ; if z < 5 then
    jge Else
    call func_if            ;   func_if()
    jmp After
Else:                       ; else
    call func_else ;        func_else()
After:                      ; end if
```

Branching And Conditionals

If statement:

```
if (x == 0) {
        x = 5;
}
        x = 2;
```

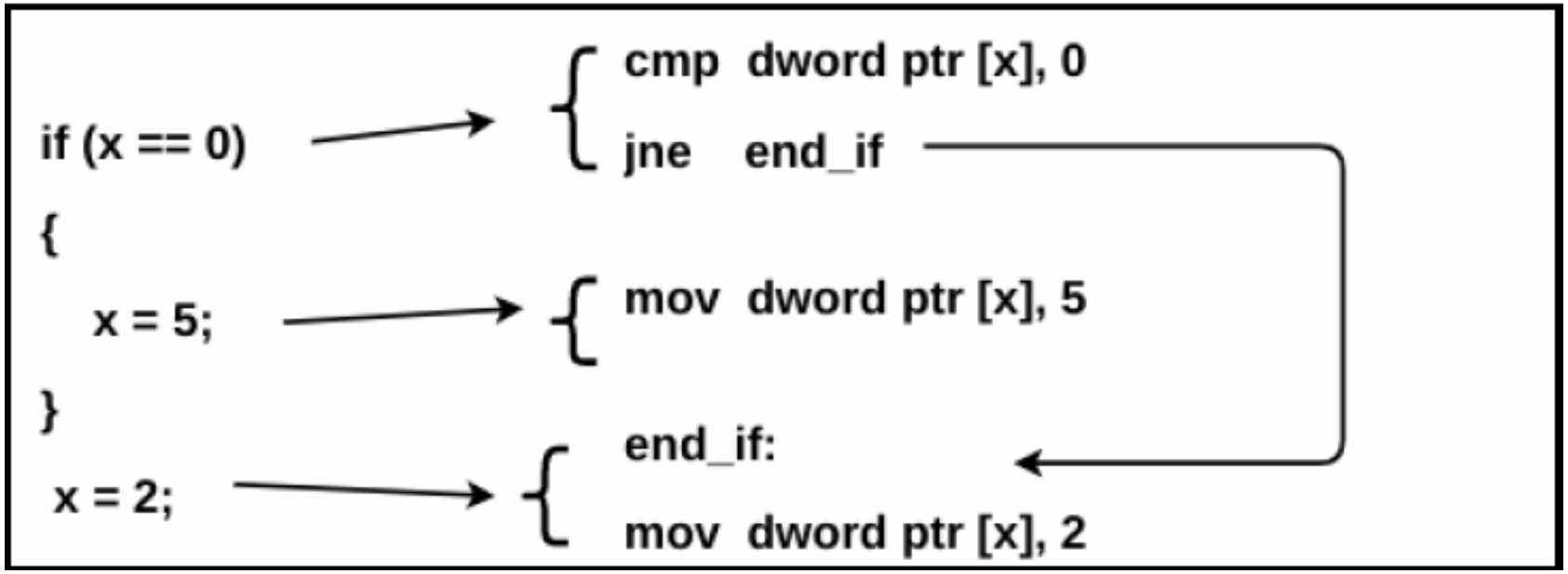
- **cmp** x with 0
- Conditionally: if x == 0, then x = 5;
- Unconditionally (in all cases): x = 2;
- …. Quit
- **What is x now?**

Branching And Conditionals

If statement:

```
if (x == 0) {
        x = 5;
}
        x = 2;
```

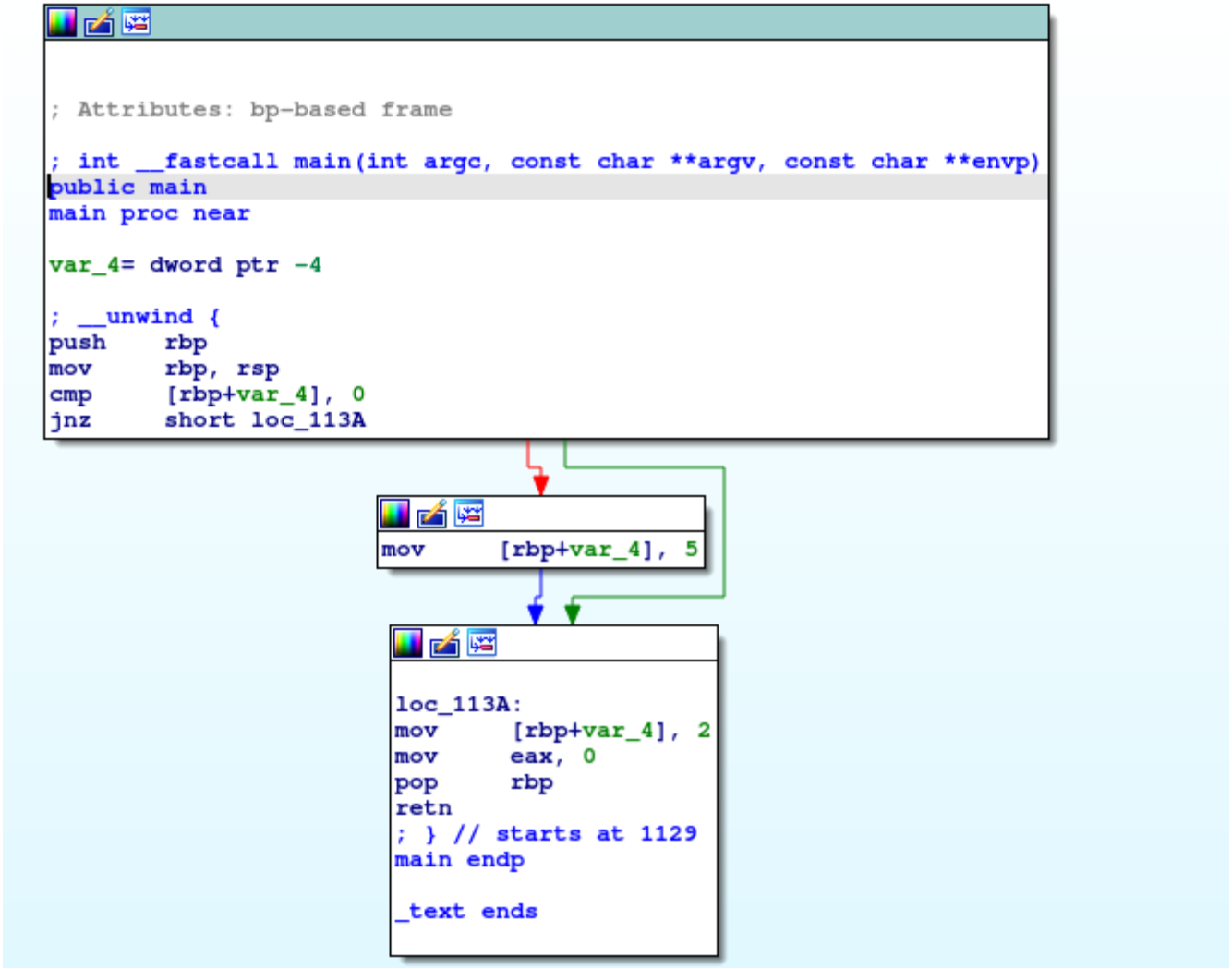
```
; Assuming x is stored at ebp-4
mov     eax, [ebp-4]                ; Load the value of x into eax
        cmp     eax, 0              ; Compare x with 0
jne     end_if                      ; Jump to end_if if x is not equal to 0
        mov     dword ptr [ebp-4], 5    ; Set x to 5 if the above condition is true

end_if:
        mov     dword ptr [ebp-4], 2    ; Set x to 2, unconditionally
```



Branching And Conditionals

If statement:

```
if (x == 0) {
        x = 5;
}
        x = 2;
```



Branching And Conditionals

If-Else Statement:

```
if (x == 0) {
        x = 5;
}
else {
        x = 1;
};
```

```
; Assume x is at ebp-4
mov     eax, [ebp-4]            ; Load the value of x into eax
cmp     eax, 0                  ; Compare x with 0
je      is_zero                 ; If x is 0, jump to is_zero
mov     dword ptr [ebp-4], 1    ; Set x to 1 (the else part)
jmp     end_if                  ; Jump to the end to avoid executing the
                                ; is_zero block

is_zero:
mov     dword ptr [ebp-4], 5    ; Set x to 5

end_if:
; Continue with the rest of the code...
```

Branching And Conditionals

If-Else Statement:

```
; Assume x is at ebp-4
mov     eax, [ebp-4]            ; Load the value of x into eax
cmp     eax, 0                  ; Compare x with 0
je      is_zero                 ; If x is 0, jump to is_zero
mov     dword ptr [ebp-4], 1    ; Set x to 1 (the else part)
jmp     end_if                  ; Jump to the end to avoid executing the
                                ; is_zero block

is_zero:
mov     dword ptr [ebp-4], 5    ; Set x to 5

end_if:
; Continue with the rest of the code...
```

Branching And Conditionals

If-Else Statement:

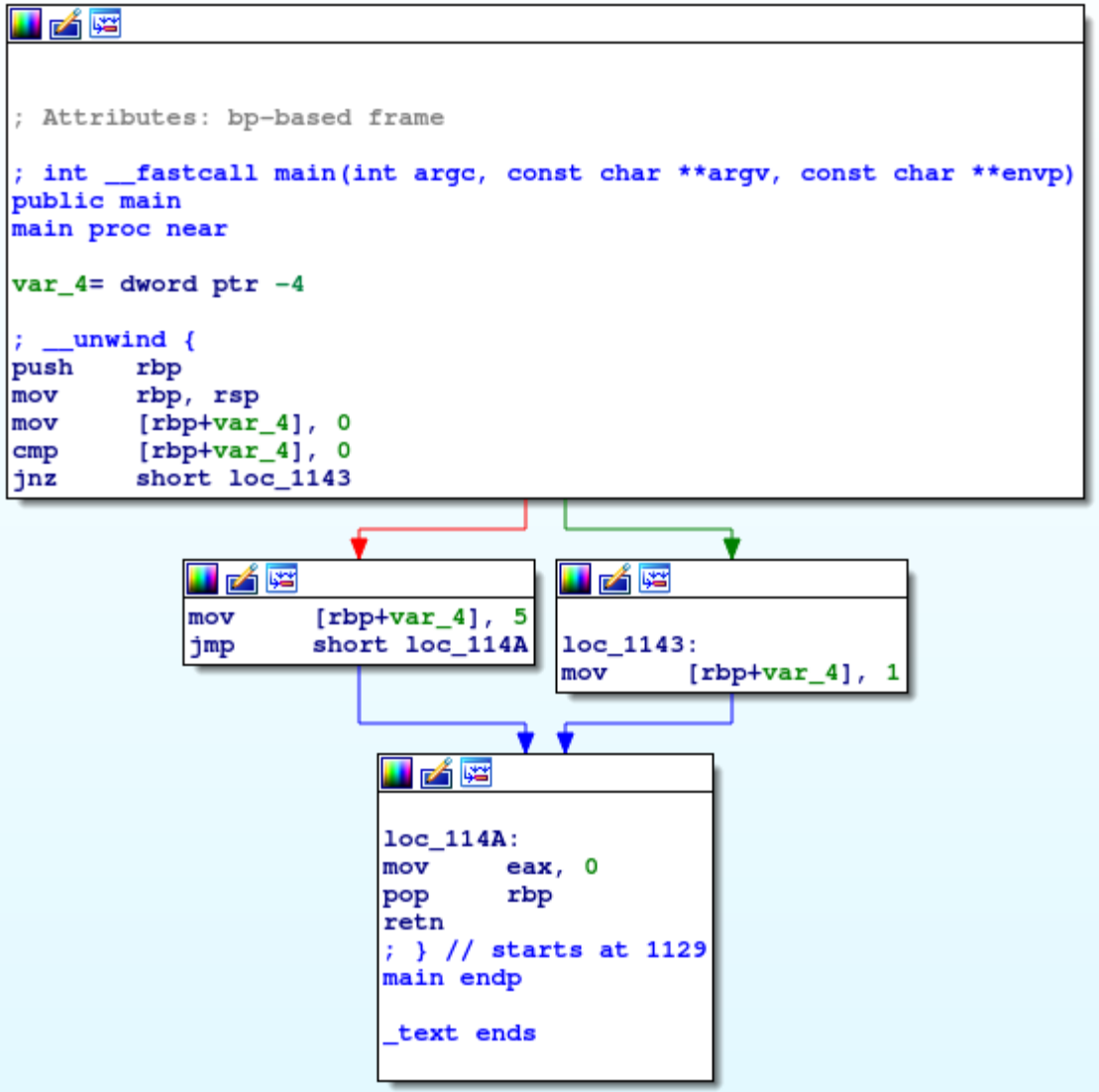
```
if (x == 0) {
        x = 5;
}
else {
        x = 1;
};
```

```
.text:0000000000001129 ; =============== S U B R O U T I N E =======================================
.text:0000000000001129
.text:0000000000001129 ; Attributes: bp-based frame
.text:0000000000001129
.text:0000000000001129 ; int __fastcall main(int argc, const char **argv, const char **envp)
.text:0000000000001129                 public main
.text:0000000000001129 main            proc near               ; DATA XREF: _start+14↑o
.text:0000000000001129
.text:0000000000001129 var_4           = dword ptr -4
.text:0000000000001129
.text:0000000000001129 ; __unwind {
.text:0000000000001129                 push    rbp
.text:000000000000112A                 mov     rbp, rsp
.text:000000000000112D                 mov     [rbp+var_4], 0
.text:0000000000001134                 cmp     [rbp+var_4], 0
.text:0000000000001138                 jnz     short loc_1143
.text:000000000000113A                 mov     [rbp+var_4], 5
.text:0000000000001141                 jmp     short loc_114A
.text:0000000000001143 ; ---------------------------------------------------------------------------
.text:0000000000001143
.text:0000000000001143 loc_1143:                               ; CODE XREF: main+F↑j
.text:0000000000001143                 mov     [rbp+var_4], 1
.text:000000000000114A
.text:000000000000114A loc_114A:                               ; CODE XREF: main+18↑j
.text:000000000000114A                 mov     eax, 0
.text:000000000000114F                 pop     rbp
.text:0000000000001150                 retn
.text:0000000000001150 ; } // starts at 1129
.text:0000000000001150 main            endp
.text:0000000000001150
.text:0000000000001150 _text           ends
.text:0000000000001150
LOAD:0000000000001151 ; ===========================================================================
```

Branching And Conditionals

If-Else Statement:

```
if (x == 0) {
        x = 5;
}
else {
        x = 1;
};
```



Branching And Conditionals

If-Elseif-Else Statement:

```
if (x == 0) {
        x = 5;
}
else if (x == 1) {
        x = 6;
}
else {
        x = 7;
}
```

Branching And Conditionals

If-Elseif-Else Statement:

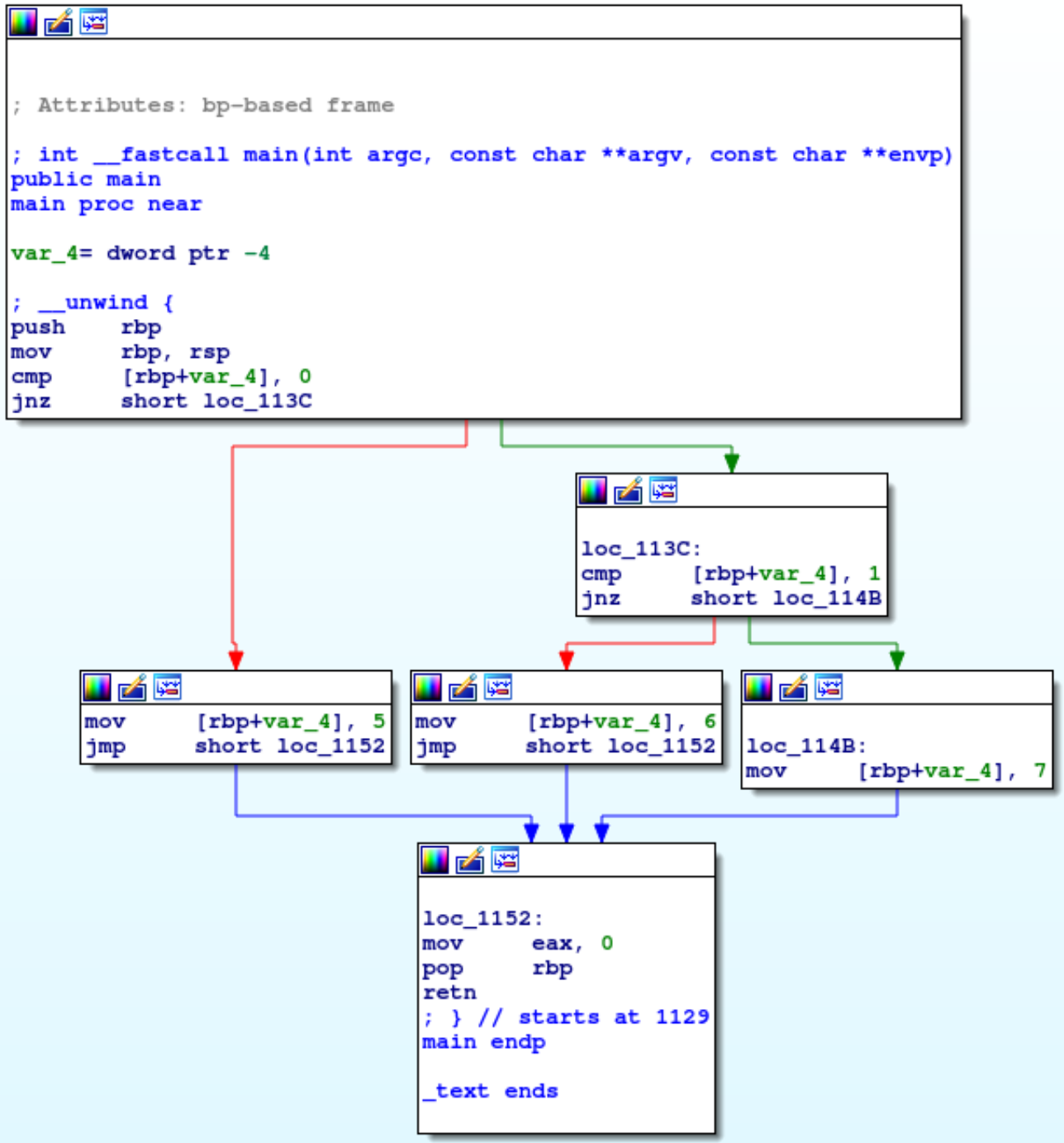
```
if (x == 0) {
        x = 5;
}
else if (x == 1) {
        x = 6;
}
else {
        x = 7;
}
```

```
; Assume x is stored at ebp-4
mov     eax, [ebp-4]            ; Load the value of x into eax
cmp     eax, 0                          ; Compare x with 0
je      x_is_zero               ; Jump if x is equal to 0
cmp     eax, 1                   ; Compare x with 1
je      x_is_one                ; Jump if x is equal to 1
mov     dword ptr [ebp-4], 7    ; Set x to 7 if it's neither 0 nor 1
jmp     end_if                  ; Jump to the end of the conditional block

x_is_zero:
mov     dword ptr [ebp-4], 5    ; Set x to 5 if it's 0
jmp     end_if                  ; Jump to the end of the conditional block

x_is_one:
mov     dword ptr [ebp-4], 6 ; Set x to 6 if it's 1

end_if:
; Continue with the rest of the code...
```

Branching And Conditionals

If-Elseif-Else Statement:

```
if (x == 0) {
        x = 5;
}
else if (x == 1) {
        x = 6;
}
else {
        x = 7;
}
```



Disassembly Challenge

Challenge: Reverse Engineering from Assembly to Pseudocode

The shown assembly is the disassembled output of a program; let's translate the following code to its high-level equivalent. Use the techniques and the concepts that you learned previously to solve this challenge:

```
        mov dword ptr [ebp-4], 1
        cmp dword ptr [ebp-4], 0

        jnz loc_40101C
        mov eax, [ebp-4]
        xor eax, 2
        mov [ebp-4], eax
        jmp loc_401025

loc_40101C:
        mov ecx, [ebp-4]
        xor ecx, 3
        mov [ebp-4], ecx

loc_401025:
```

Disassembly Challenge

```
        mov dword ptr [ebp-4], 1
        cmp dword ptr [ebp-4], 0

        jnz loc_40101C
        mov eax, [ebp-4]
        xor eax, 2
        mov [ebp-4], eax
        jmp loc_401025

loc_40101C:
        mov ecx, [ebp-4]
        xor ecx, 3
        mov [ebp-4], ecx

loc_401025:
```

Disassembly Challenge

```
int x = 1;
        if (x == 0) {
                x ^= 2;
        }

        else {
                x ^= 3;
                }
```

```
int x = 1;
        if (x != 0) {
                x ^= 3;
        }

        else {
                x ^= 2;
                }
```

```
        mov dword ptr [ebp-4], 1
        cmp dword ptr [ebp-4], 0

        jnz loc_40101C
        mov eax, [ebp-4]
        xor eax, 2
        mov [ebp-4], eax
        jmp loc_401025

loc_40101C:
        mov ecx, [ebp-4]
        xor ecx, 3
        mov [ebp-4], ecx

loc_401025:
```

Disassembly Challenge

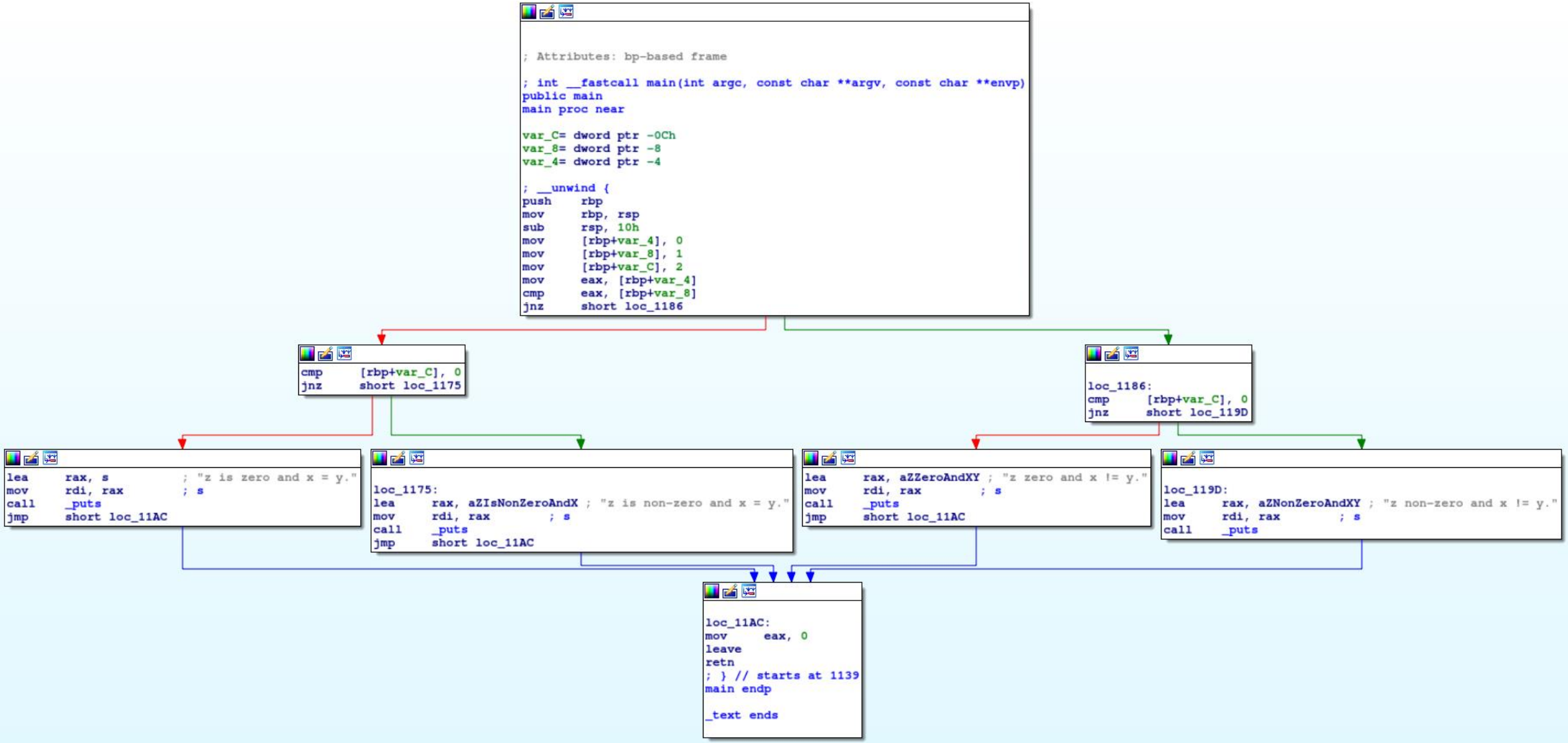
1. 00401006 mov [ebp+var_8], 1
2. 0040100D mov [ebp+var_4], 2
3. 00401014 mov eax, [ebp+var_8]
4. 00401017 cmp eax, [ebp+var_4]
5. 0040101A jnz short **loc_40102B**
6. 0040101C push offset aXEqualsY_ ; "x equals y.\n"
7. 00401021 call printf
8. 00401026 add esp, 4
9. 00401029 jmp short loc_401038
10. **0040102B** **loc_40102B:**
11. 0040102B push offset aXIsNotEqualToY ; "x is not equal to y.\n"
12. 00401030 call printf

Disassembly Challenge

1. 00401006 mov [ebp+var_8], 1
2. 0040100D mov [ebp+var_4], 2
3. 00401014 mov eax, [ebp+var_8]
4. 00401017 cmp eax, [ebp+var_4]
5. 0040101A jnz short loc_40102B
6. 0040101C push offset aXEqualsY_ ; "x = y.\n"
7. 00401021 call printf
8. 00401026 add esp, 4
9. 00401029 jmp short loc_401038
10. **0040102B loc_40102B:**
11. 0040102B push offset aXIsNotEqualToY ; "x != y.\n"
12. 00401030 call printf

```c
#include <stdio.h>

int main() {
    int var_8 = 1;
    int var_4 = 2;

    if (var_8 != var_4) {
        printf("x is not equal to y.\n");
    } else {
        printf("x equals y.\n");
    }

    return 0;
}
```

Disassembly Challenge 4U! Try it…



Switch

```
switch(i)
{
case 1:
        printf("i = %d", i+1);
        break;
case 2:
        printf("i = %d", i+2);
        break;
case 3:
        printf("i = %d", i+3);
        break;
default:
        break;
}
```

00401013 **cmp** [ebp+var_8], 1
00401017 **jz** short loc_401027 →

00401027 loc_401027:
00401027 mov ecx, [ebp+var_4]
0040102A add ecx, 1
0040102D push ecx
0040102E push offset unk_40C000 ; i = %d
00401033 call printf
00401038 add esp, 8
0040103B jmp short loc_401067

00401019 **cmp** [ebp+var_8], 2
0040101D **jz** short loc_40103D →

0040103D loc_40103D:
0040103D mov edx, [ebp+var_4]
00401040 add edx, 2
00401043 push edx
00401044 push offset unk_40C004 ; i = %d
00401049 call printf
0040104E add esp, 8
00401051 jmp short loc_401067

0040101F **cmp** [ebp+var_8], 3
00401023 **jz** short loc_401053 →

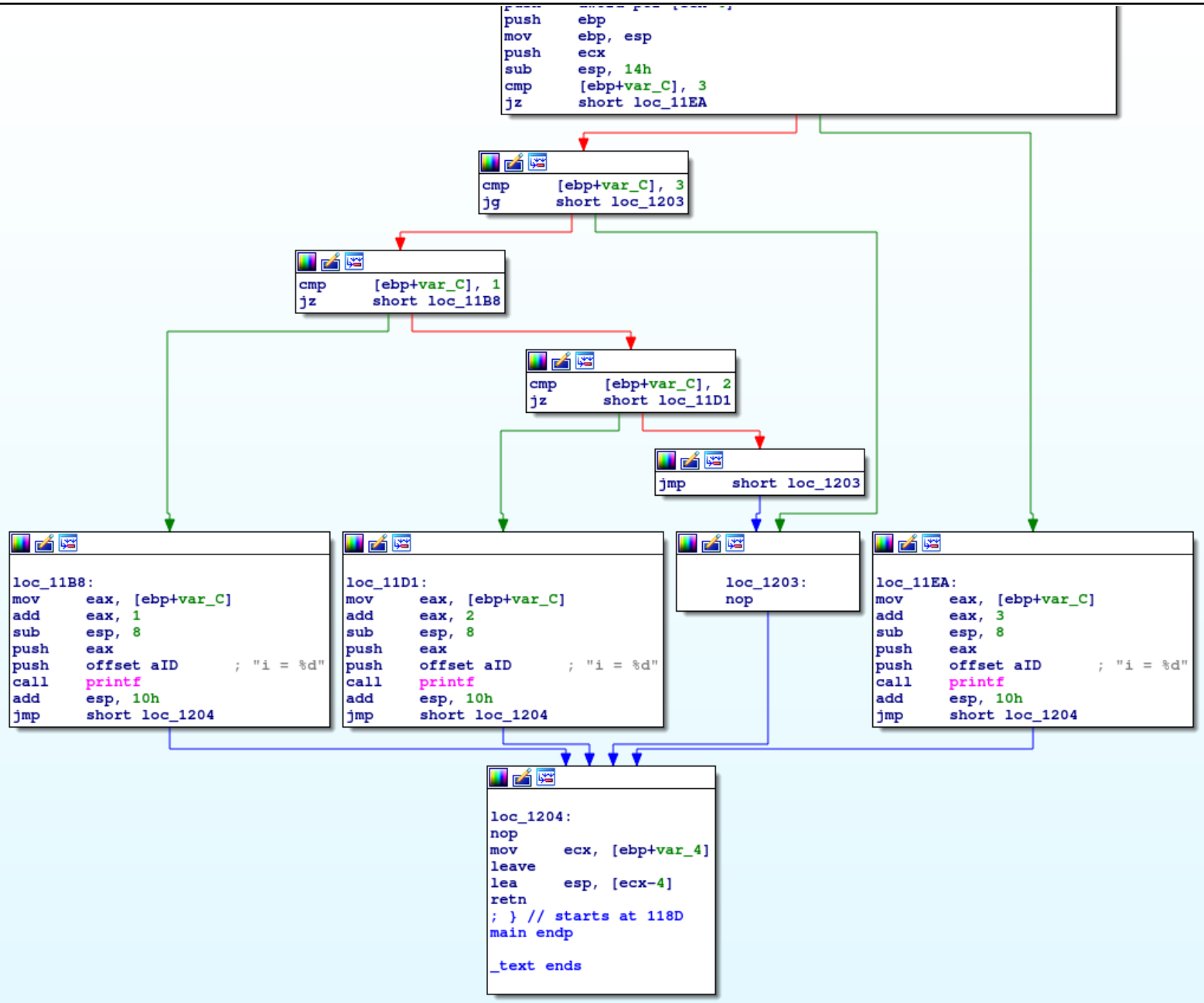
00401053 loc_401053:
00401053 mov eax, [ebp+var_4]
00401056 add eax, 3
00401059 push eax
0040105A push offset unk_40C008 ; i = %d
0040105F call printf
00401064 add esp, 8

00401025 **jmp** short loc_401067 →

loc_401067:

Switch

```
switch(i)
{
case 1:
        printf("i = %d", i+1);
        break;
case 2:
        printf("i = %d", i+2);
        break;
case 3:
        printf("i = %d", i+3);
        break;
default:
        break;
}
```



Looping

Loops

The loops jump backward

For Loops

The general form of **a for** loop is :

```
for (initialization; condition; update_statement )
{
      block of code
}
```

```
int i;
for (i = 0; i < 5; i++)
{
Do something...
}
```

For Loops

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

```
for (initialization; condition; update_statement )
{
        block of code
}
```

```
00401004 mov     [ebp+var_4], 0
0040100B jmp short loc_401016
0040100D loc_40100D:
0040100D mov eax, [ebp+var_4]
00401010 add eax, 1
00401013 mov [ebp+var_4], eax
00401016 loc_401016:
00401016 cmp [ebp+var_4], 64h
0040101A jge short loc_40102F
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
```

For Loops

```
                mov         [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

```
for (initialization; condition; update_statement )
{
        block of code
}
```

For Loops

```
                mov         [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

```
for (initialization; condition; update_statement )
{
        block of code
}
```

1) Initialization:

mov [ebp+var_4], 0:
Initializes the loop counter (stored at [ebp+var_4]) to 0. This corresponds to int i = 0; in the C code.

For Loops

```
                mov         [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

2) Jump to Condition Check (before loop starts):

jmp short loc_401016: Jumps to the loop condition check before entering the loop. This ensures that the loop condition is checked before the loop body is executed for the first time.

This jump takes us directly to the comparison instruction to ensure the loop condition (i < 100 or **64h**) is evaluated before entering the loop body.

For Loops

```
                mov         [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

3) Executing the Loop Body:

This is where the printf function is called to print the current value of i.

- mov ecx, [ebp+var_4]: Prepares the current value of i for printf.
- push ecx and push offset aID: Pushes the arguments for printf onto the stack.
- call printf: Calls the printf function using the arguments.
- add esp, 8: Cleans up the stack by removing the printf arguments.

For Loops

```
                mov         [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

4) Loop Iteration:

jmp short loc_40100D
This jump sends execution back to the point where i is incremented, effectively continuing the loop until the condition is no longer met.

For Loops

```
                mov         [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

5) Incrementing the Loop Counter:

After executing the loop body, the counter needs to be incremented:

- mov eax, [ebp+var_4]: Moves the value of i into the eax register.
- add eax, 1: Adds 1 to eax, incrementing i.
- mov [ebp+var_4], eax: Stores the incremented value back in the memory location of i.

For Loops

```
        mov     [ebp+var_4], 0
        jmp short loc_401016

0040100D loc_40100D:
        mov eax, [ebp+var_4]
        add eax, 1
        mov [ebp+var_4], eax

00401016 loc_401016:
        cmp [ebp+var_4], 64h
        jge short loc_40102F
        mov ecx, [ebp+var_4]
        push ecx
        push offset aID ; "i equals %d\n"
        call printf
        add esp, 8
0040102D jmp short loc_40100D
```

6) Condition Check:

cmp [ebp+var_4], 64h

Compares the value of i to 100 (64 in hexadecimal) to decide whether to continue the loop.

jge short loc_40102F

If i is greater than or equal to 100, the jump to loc_40102F exits the loop. Otherwise, execution continues with the loop body.

For Loops

```
                mov     [ebp+var_4], 0
                jmp short loc_401016

0040100D loc_40100D:
                mov eax, [ebp+var_4]
                add eax, 1
                mov [ebp+var_4], eax

00401016 loc_401016:
                cmp [ebp+var_4], 64h
                 jge short loc_40102F
                mov ecx, [ebp+var_4]
                push ecx
                push offset aID ; "i equals %d\n"
                call printf
                add esp, 8
0040102D jmp short loc_40100D
```

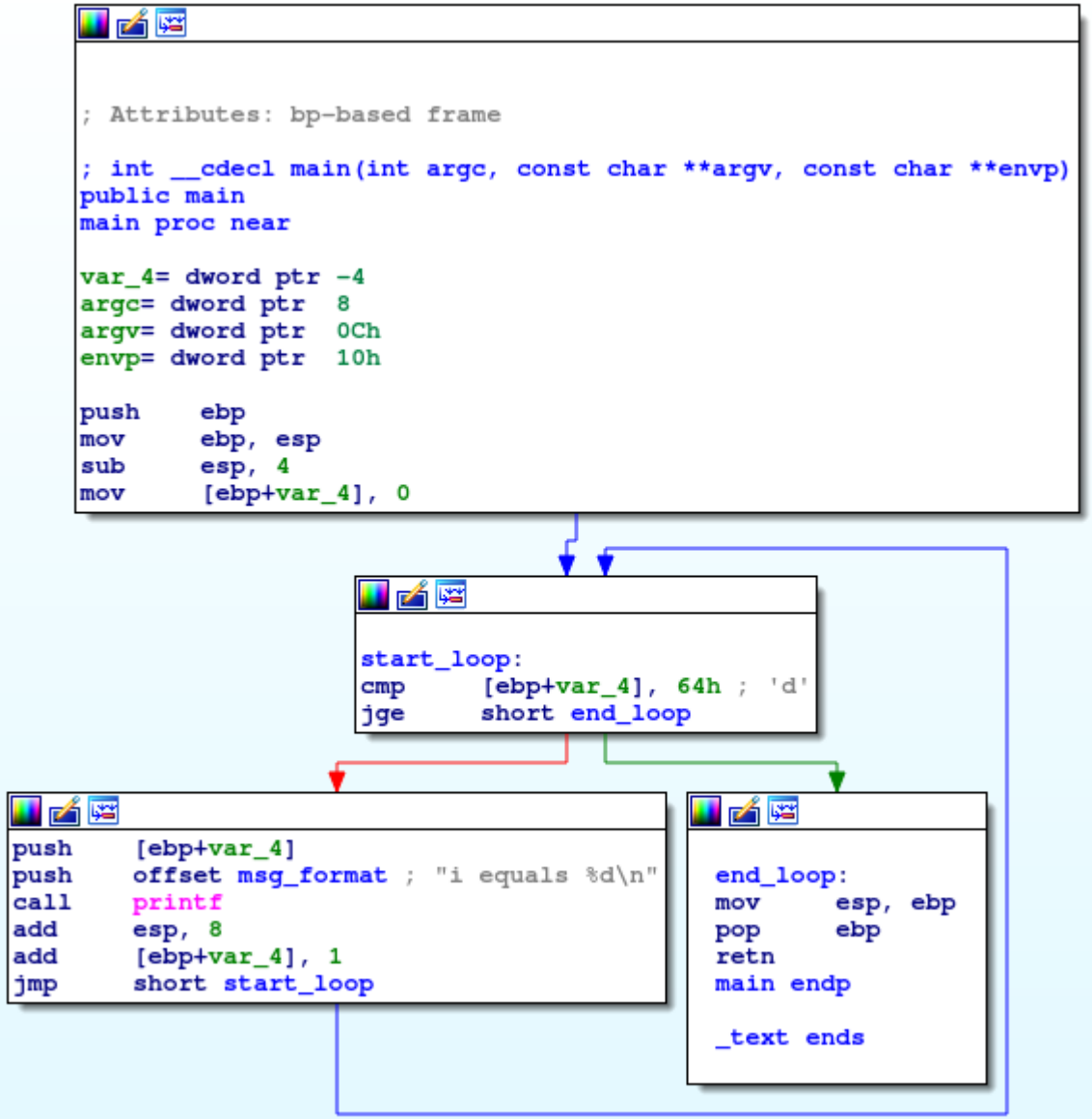
1) Initialization
2) Jump to Condition Check (before loop starts)
3) Executing the Loop Body
4) Loop Iteration
5) Incrementing the Loop Counter
6) Condition Check

For Loops – 32 Bit semi native

```
section .data
msg_format db 'i equals %d', 10, 0

section .text
global main
extern printf

main:
    push ebp
    mov ebp, esp
    sub esp, 4
    mov dword [ebp-4], 0

start_loop:
    cmp dword [ebp-4], 100
    jge end_loop

    push dword [ebp-4]
    push msg_format
    call printf
    add esp, 8

    add dword [ebp-4], 1
    jmp start_loop

end_loop:
    mov esp, ebp
    pop ebp
    ret
```

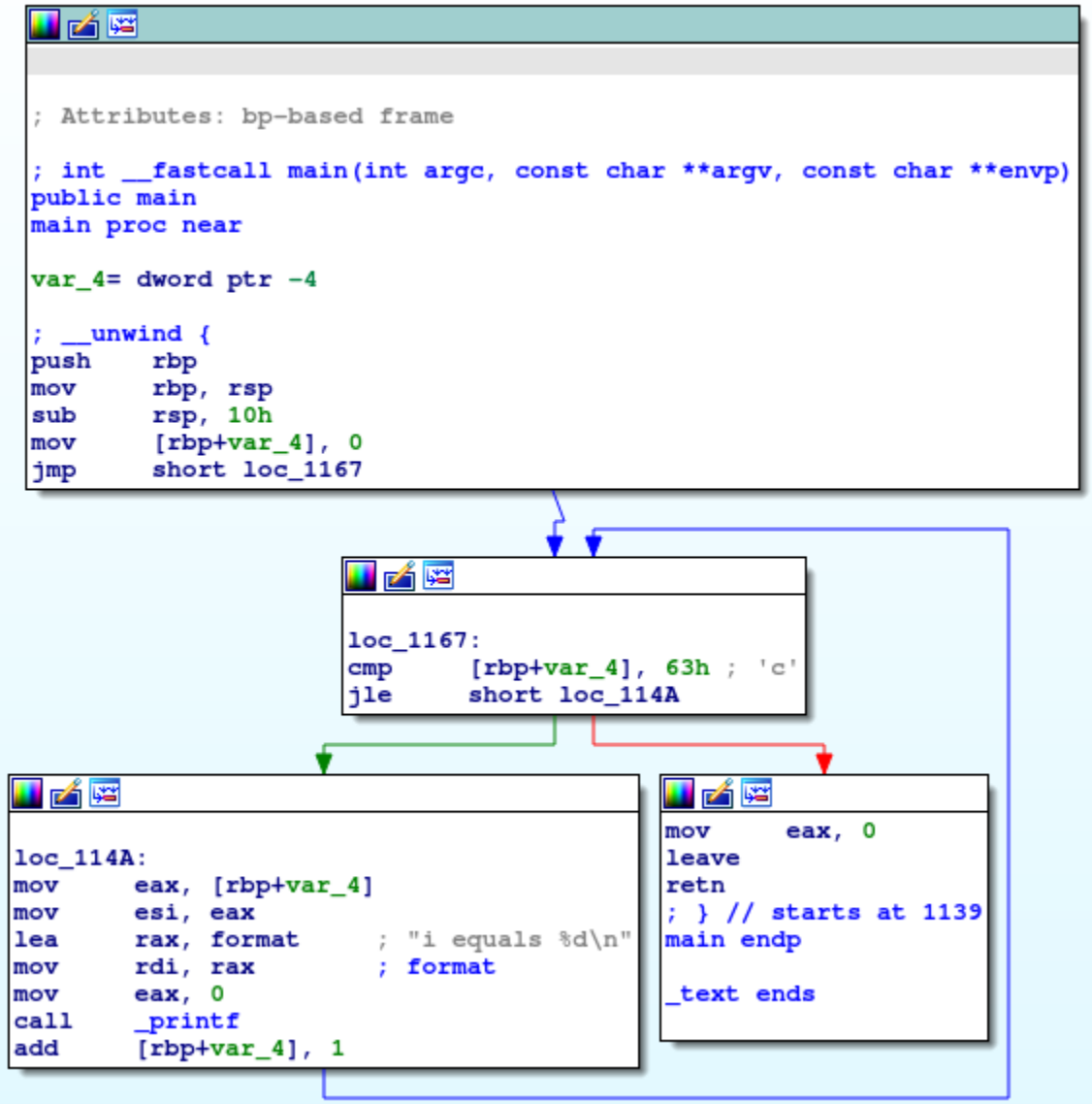
For Loops – IDA 32 bit

```
section .data
msg_format db 'i equals %d', 10, 0

section .text
global main
extern printf

main:
    push ebp
    mov ebp, esp
    sub esp, 4
 mov dword [ebp-4], 0

start_loop:
    cmp dword [ebp-4], 100
    jge end_loop

    push dword [ebp-4]
    push msg_format
    call printf
    add esp, 8

    add dword [ebp-4], 1
    jmp start_loop

end_loop:
    mov esp, ebp
    pop ebp
    ret
```



For Loops – IDA 64 bit

```
.text:0000000000001139
.text:0000000000001139 ; =============== S U B R O U T I N E =======================
.text:0000000000001139
.text:0000000000001139 ; Attributes: bp-based frame
.text:0000000000001139
.text:0000000000001139 ; int __fastcall main(int argc, const char **argv, const c
.text:0000000000001139                 public main
.text:0000000000001139 main            proc near               ; DATA XREF: _sta
.text:0000000000001139
.text:0000000000001139 var_4           = dword ptr -4
.text:0000000000001139
.text:0000000000001139 ; __unwind {
.text:0000000000001139                 push    rbp
.text:000000000000113A                 mov     rbp, rsp
.text:000000000000113D                 sub     rsp, 10h
.text:0000000000001141                 mov     [rbp+var_4], 0
.text:0000000000001148                 jmp     short loc_1167
.text:000000000000114A ; ---------------------------------------------------------
.text:000000000000114A
.text:000000000000114A loc_114A:                               ; CODE XREF: main+
.text:000000000000114A                 mov     eax, [rbp+var_4]
.text:000000000000114D                 mov     esi, eax
.text:000000000000114F                 lea     rax, format     ; "i equals %d\n"
.text:0000000000001156                 mov     rdi, rax        ; format
.text:0000000000001159                 mov     eax, 0
.text:000000000000115E                 call    _printf
.text:0000000000001163                 add     [rbp+var_4], 1
.text:0000000000001167
.text:0000000000001167 loc_1167:                               ; CODE XREF: main+
.text:0000000000001167                 cmp     [rbp+var_4], 63h ; 'c'
.text:000000000000116B                 jle     short loc_114A
.text:000000000000116D                 mov     eax, 0
.text:0000000000001172                 leave
.text:0000000000001173                 retn
.text:0000000000001173 ; } // starts at 1139
.text:0000000000001173 main            endp
.text:0000000000001173
.text:0000000000001173 _text           ends
.text:0000000000001173
.fini:0000000000001174 ; ---------------------------------------------------------
```



For Loops – textbook example 32 bit

```
00401004 mov    [ebp+var_4], 0
0040100B jmp short loc_401016
0040100D loc_40100D:
0040100D mov eax, [ebp+var_4]
00401010 add eax, 1
00401013 mov [ebp+var_4], eax
00401016 loc_401016:
00401016 cmp [ebp+var_4], 64h
0040101A jge short loc_40102F
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
```

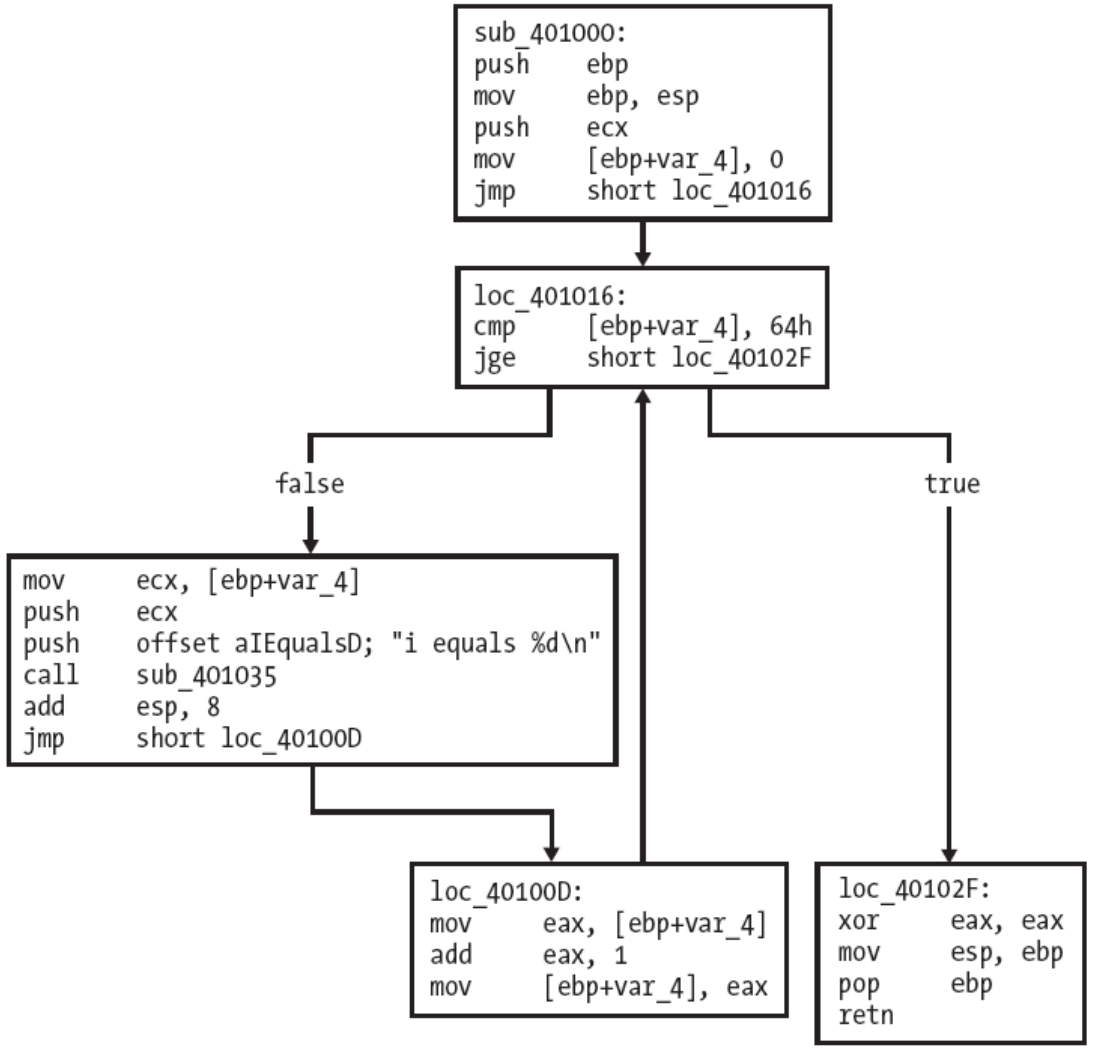


Figure 6-2: Disassembly graph for the for loop example in Listing 6-13

While Loops

The general form of a **while** loop is:

initialization
while (**condition**)
{
block of code
update_statement }

Example:

```
int i = 0;
while (i < 5) {
DoSomething …
i++;
}
```

```
mov [i],0
while_start:
        cmp [i], 5
        jge end

        DoSomething …

        mov eax, [i]
        add eax, 1
        mov [i], eax

jmp while_start
end:
```

Disassembly Challenge

```
mov dword ptr [ebp-8], 1
mov dword ptr [ebp-4], 0
loc_401014:
cmp dword ptr [ebp-4], 4
jge short loc_40102E
mov eax, [ebp-8]
add eax, [ebp-4]
mov [ebp-8], eax
mov ecx, [ebp-4]
add ecx, 1
mov [ebp-4], ecx
jmp short loc_401014
loc_40102E:
```

Disassembly Challenge

```
mov dword ptr [ebp-8], 1
mov dword ptr [ebp-4], 0
loc_401014:
cmp dword ptr [ebp-4], 4
jge short loc_40102E
mov eax, [ebp-8]
add eax, [ebp-4]
mov [ebp-8], eax
mov ecx, [ebp-4]
add ecx, 1
mov [ebp-4], ecx
jmp short loc_401014
loc_40102E:
```

```
int sum = 1;
for (int i = 0; i < 4; i++) {
    sum += i;
}
```

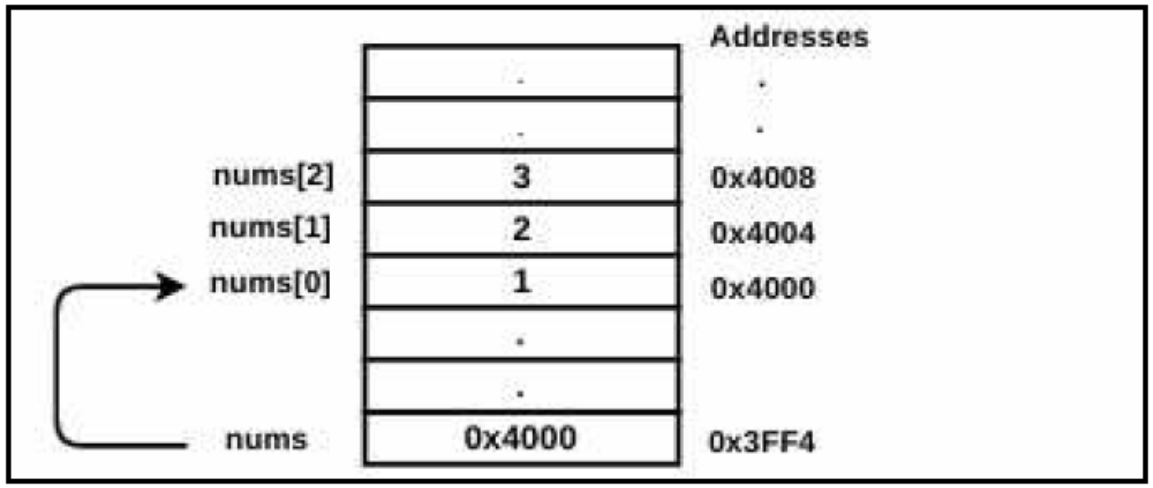
```
int sum = 1;
int i = 0;
while (i < 4) {
        sum += i;
         i++;
}
```

Arrays

Arrays:

An array is a structured **collection of elements** that are of the same data type. It is characterized by its ability to store these elements in contiguous memory locations, thereby facilitating efficient access to each element.

int nums[3] = {1, 2, 3};

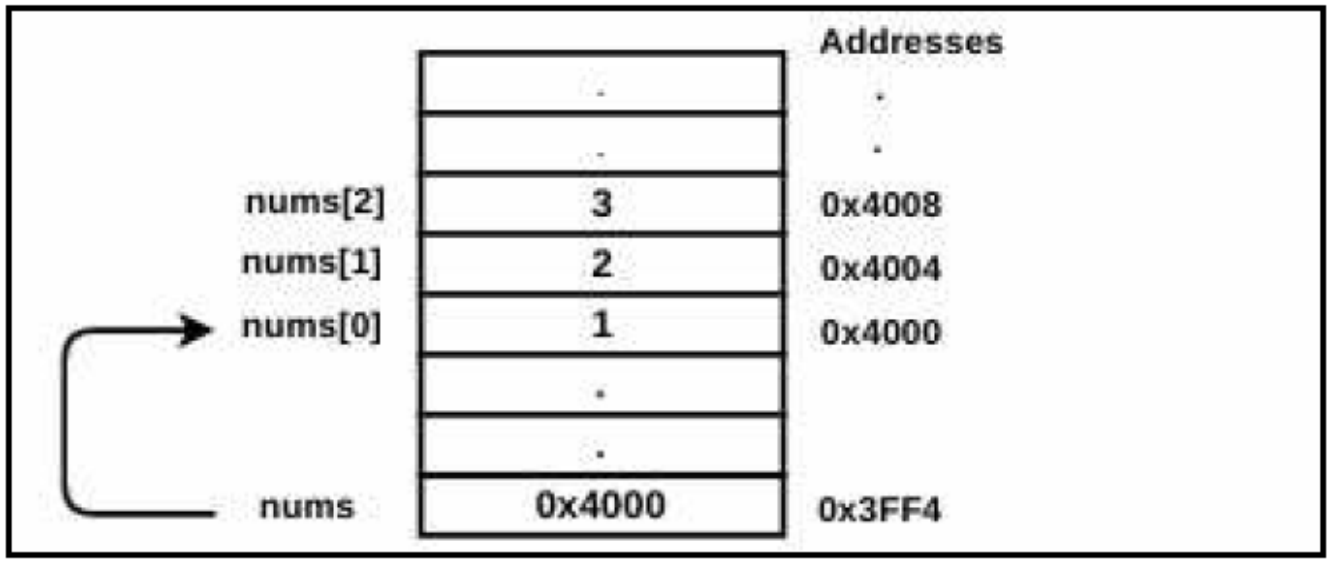


Arrays:

[base_address + index * size of element]

In assembly language, the address of any element in the array is computed using three things:

1. The base address
2. The index of the element
3. The size of each element in the array



When you use nums[0] in a high-level language, it is translated to [nums+0*<size_of_each_element_in_bytes>], where 0 is the index and nums represents the base address of the array

nums[0] = [nums+0*4] = [0x4000+0*4] = [0x4000] = 1
nums[1] = [nums+1*4] = [0x4000+1*4] = [0x4004] = 2
nums[2] = [nums+2*4] = [0x4000+2*4] = [0x4008] = 3

Disassembly Challenge

Translate the following code to its high-level equivalent. Use the techniques and the concepts that you have learned so far to solve this challenge:

```
        push ebp
        mov ebp, esp
        sub esp, 14h
        mov dword ptr [ebp-14h], 1
        mov dword ptr [ebp-10h], 2
        mov dword ptr [ebp-0Ch], 3
        mov dword ptr [ebp-4], 0
loc_401022:
        cmp dword ptr [ebp-4], 3
        jge loc_40103D
        mov eax, [ebp-4]
        mov ecx, [ebp+eax*4-14h]
        mov [ebp-8], ecx
        mov edx, [ebp-4]
        add edx, 1
        mov [ebp-4], edx
        jmp loc_401022
loc_40103D:
        xor eax, eax
        mov esp, ebp
        pop ebp
        ret
```

[base_address + index * size of element]

Disassembly Challenge

```
int main()
{
	int a[3] = { 1, 2, 3 };
	int b, i;
	i = 0;

	while (i < 3)
	{
		b = a[i];
		i++;
	}
return 0;
}
```

```
	push ebp
	mov ebp, esp
	sub esp, 14h
	mov dword ptr [ebp-14h], 1
	mov dword ptr [ebp-10h], 2
	mov dword ptr [ebp-0Ch], 3
	mov dword ptr [ebp-4], 0
loc_401022:
	cmp dword ptr [ebp-4], 3
	jge loc_40103D
	mov eax, [ebp-4]
	mov ecx, [ebp+eax*4-14h]
	mov [ebp-8], ecx
	mov edx, [ebp-4]
	add edx, 1
	mov [ebp-4], edx
	jmp loc_401022
loc_40103D:
	xor eax, eax
	mov esp, ebp
	pop ebp
	ret
```

From C to Assembly

From C to Assembly

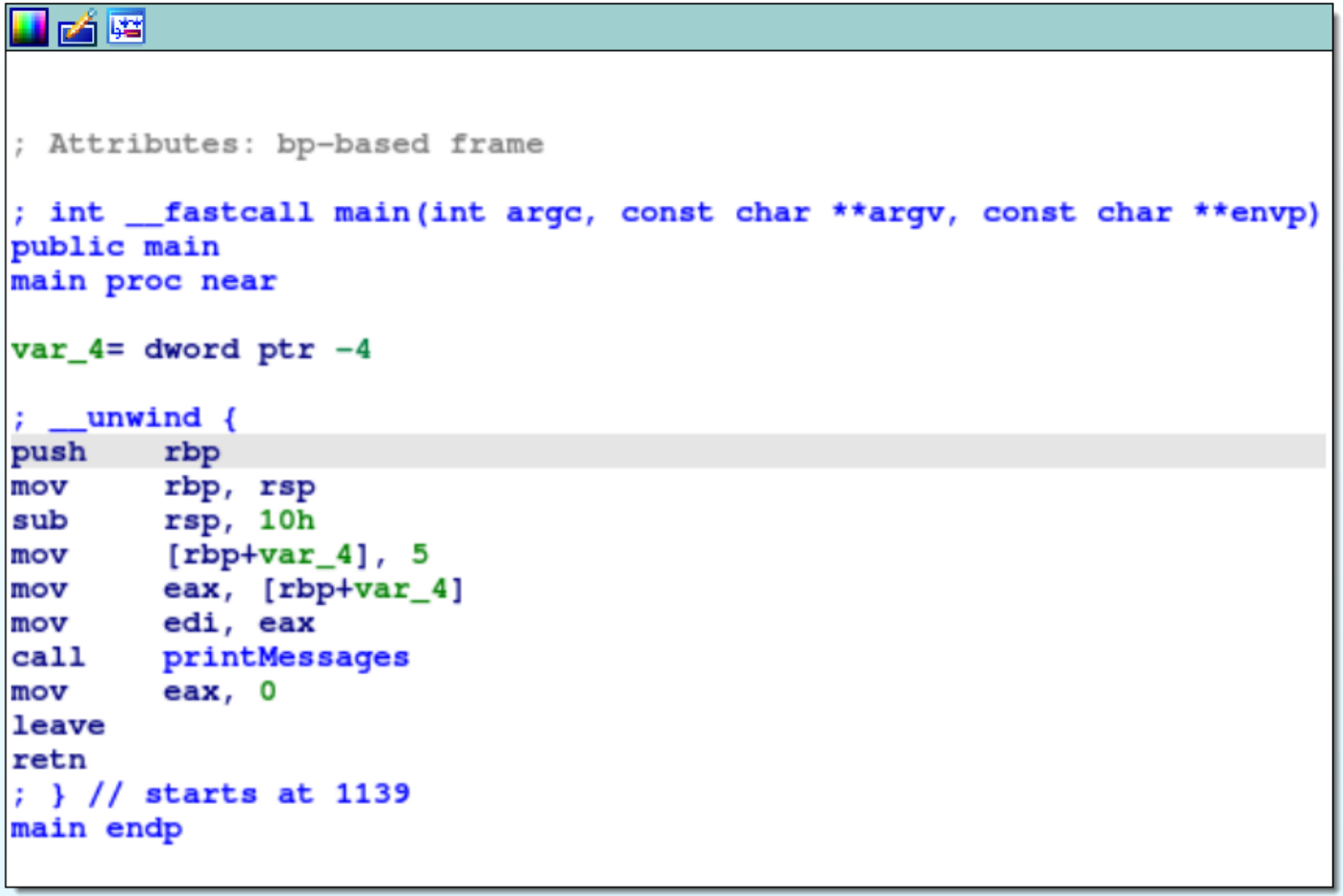
hellorit.c hellorit2.c

HelloRITDubai > hellorit2.c

```c
#include <stdio.h>

// Function declaration
void printMessages(int numberOfMessages);

int main() {
    int numberOfMessages = 5; // You can change this to any number you like

    // Calling the printMessages function with the number of messages
    printMessages(numberOfMessages);

    return 0;
}

// Function definition
void printMessages(int numberOfMessages) {
    for (int i = 0; i < numberOfMessages; i++) {
        printf("Hello RIT Dubai!\n");
    }
}
```

⚠ 0 0

HelloRIT V2.0 main()



```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], 5
mov     eax, [rbp+var_4]
mov     edi, eax
call    printMessages
mov     eax, 0
leave
retn
; } // starts at 1139
main endp
```

HelloRIT V2.0 printMessages()

```
.text:0000000000001159 ; =============== S U B R O U T I N E =======================================
.text:0000000000001159
.text:0000000000001159 ; Attributes: bp-based frame
.text:0000000000001159
.text:0000000000001159                 public printMessages
.text:0000000000001159 printMessages   proc near               ; CODE XREF: main+14↑p
.text:0000000000001159
.text:0000000000001159 var_14          = dword ptr -14h
.text:0000000000001159 var_4           = dword ptr -4
.text:0000000000001159
.text:0000000000001159 ; __unwind {
.text:0000000000001159                 push    rbp
.text:000000000000115A                 mov     rbp, rsp
.text:000000000000115D                 sub     rsp, 20h
.text:0000000000001161                 mov     [rbp+var_14], edi
.text:0000000000001164                 mov     [rbp+var_4], 0
.text:000000000000116B                 jmp     short loc_1180
.text:000000000000116D ; ---------------------------------------------------------------------------
.text:000000000000116D
.text:000000000000116D loc_116D:                               ; CODE XREF: printMessages+2D↓j
.text:000000000000116D                 lea     rax, s          ; "Hello RIT Dubai!"
.text:0000000000001174                 mov     rdi, rax        ; s
.text:0000000000001177                 call    _puts
.text:000000000000117C                 add     [rbp+var_4], 1
.text:0000000000001180
.text:0000000000001180 loc_1180:                               ; CODE XREF: printMessages+12↑j
.text:0000000000001180                 mov     eax, [rbp+var_4]
.text:0000000000001183                 cmp     eax, [rbp+var_14]
.text:0000000000001186                 jl      short loc_116D
.text:0000000000001188                 nop
.text:0000000000001189                 nop
.text:000000000000118A                 leave
.text:000000000000118B                 retn
.text:000000000000118B ; } // starts at 1159
.text:000000000000118B printMessages   endp
.text:000000000000118B
.text:000000000000118B _text           ends
.text:000000000000118B
.fini:000000000000118C ; ===========================================================================
```

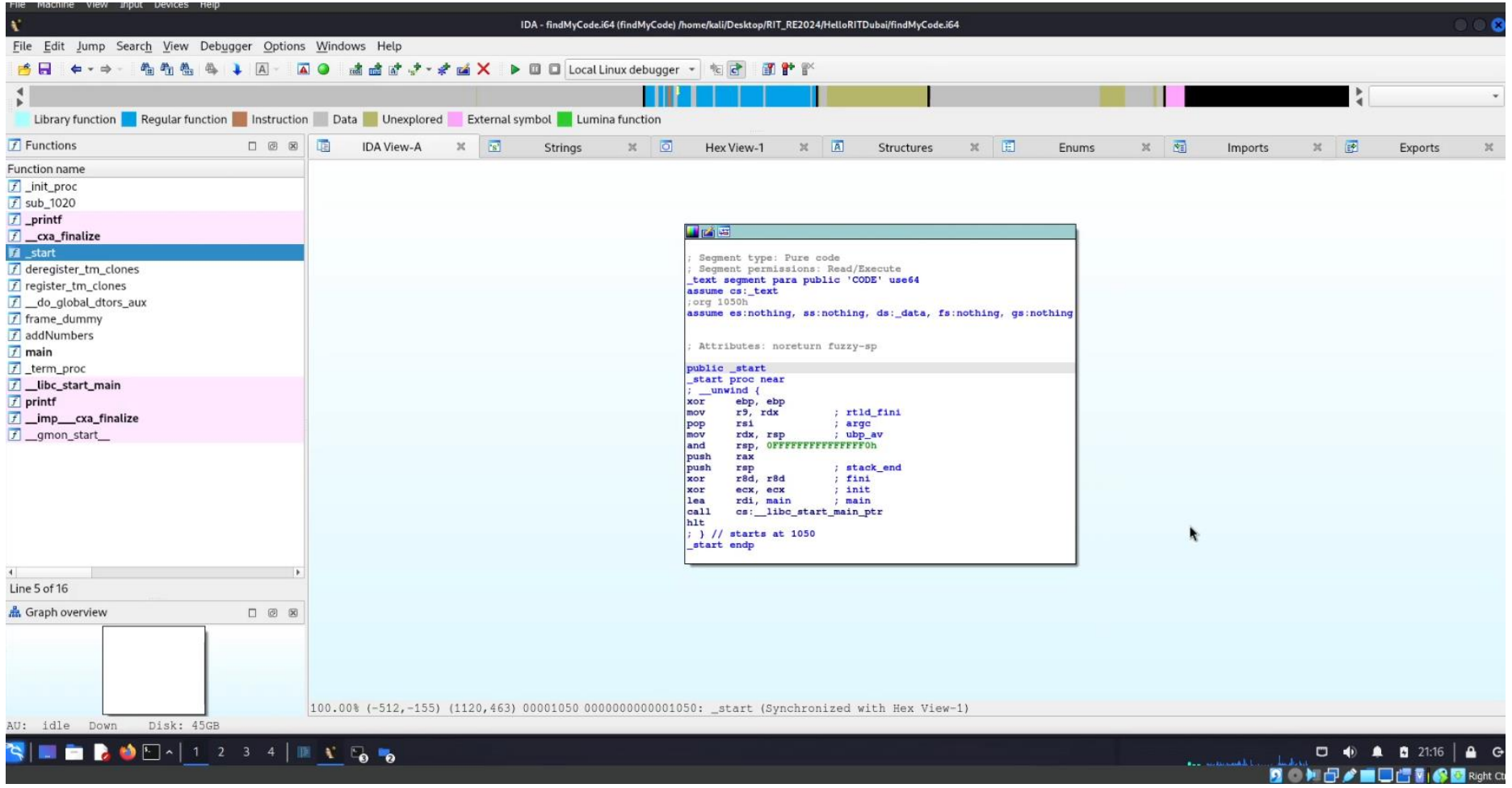
HelloRIT V2.0
printMessages()

All together … From Assembly to C

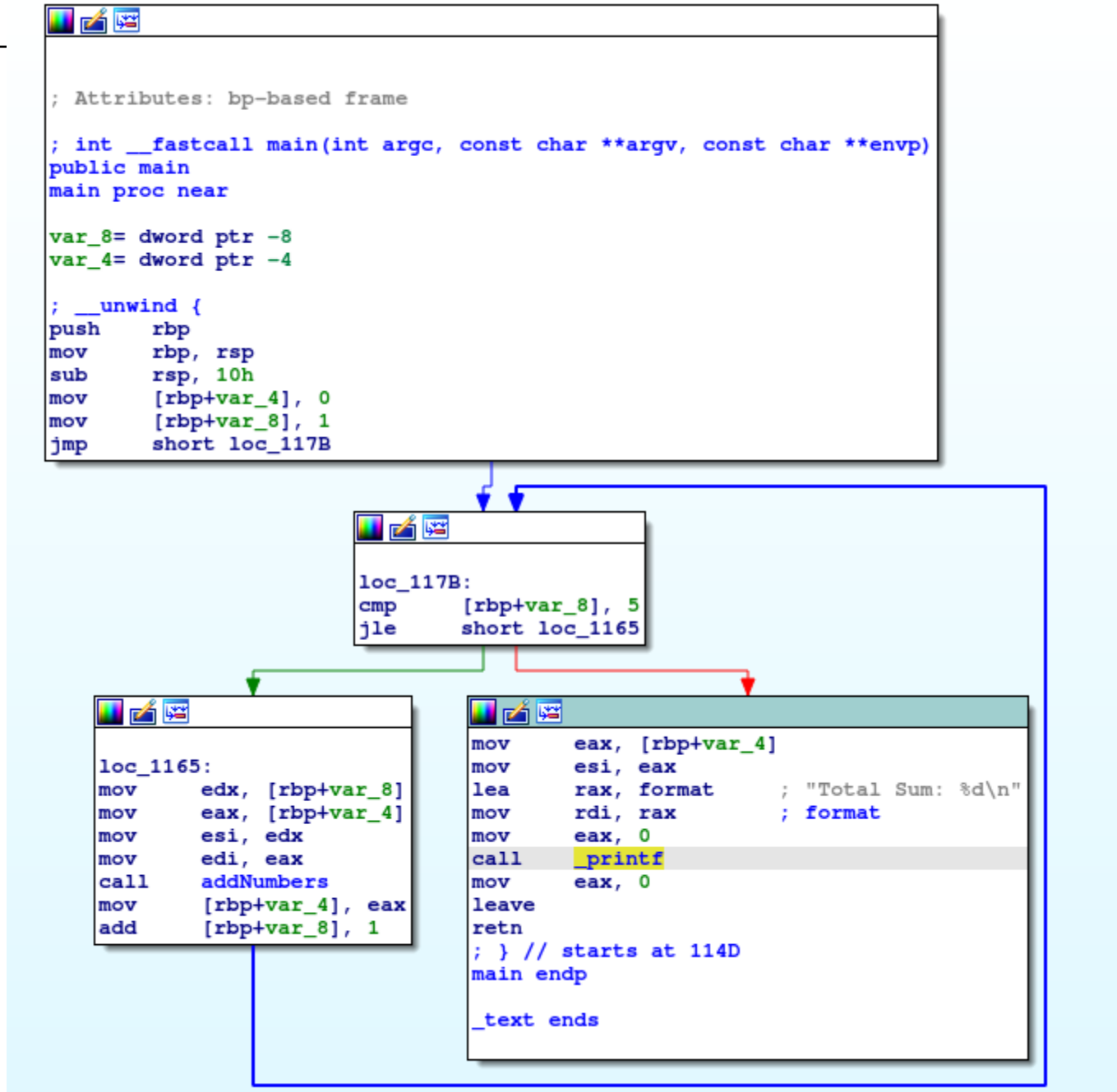
Reverse Engineering Strategy (Our Roadmap)

- **Step 1: String Hunting**
  - Note down interesting strings that suggest functionality.
- **Step 2: Identifying Entry or Main Function**
- **Step 3: Scaffolding**
  - Create a C code structure based on IDA's assembly observations.
  - Sketch out the main function with noted strings.
- **Step 4: Reconstructing Logic**
  - Translate assembly instructions into C constructs. Account for loops, conditionals, and function calls.
- **Step 5: Iterative Refinement**

Tigers! Can you “FindMyCode”?



FindMyCode main()



FindMyCode
main()

- In the provided assembly for the main function, we observe a structured sequence of operations:
- **Initialization:** The function begins by establishing a stack frame, a common practice in assembly to manage variables and function calls. It uses push rbp and mov rbp, rsp to do this.
- **Variable Assignment:** Two local variables are initialized. mov [rbp+var_4], 0 sets a local variable, likely our accumulator or sum, to zero. mov [rbp+var_8], 1 initializes another variable, probably a counter, to one.
- **Loop Mechanics**: The code then enters a loop, indicated by the jmp instruction which jumps to loc_117B. This loop is where the main computational work of the function occurs.

FindMyCode - main()

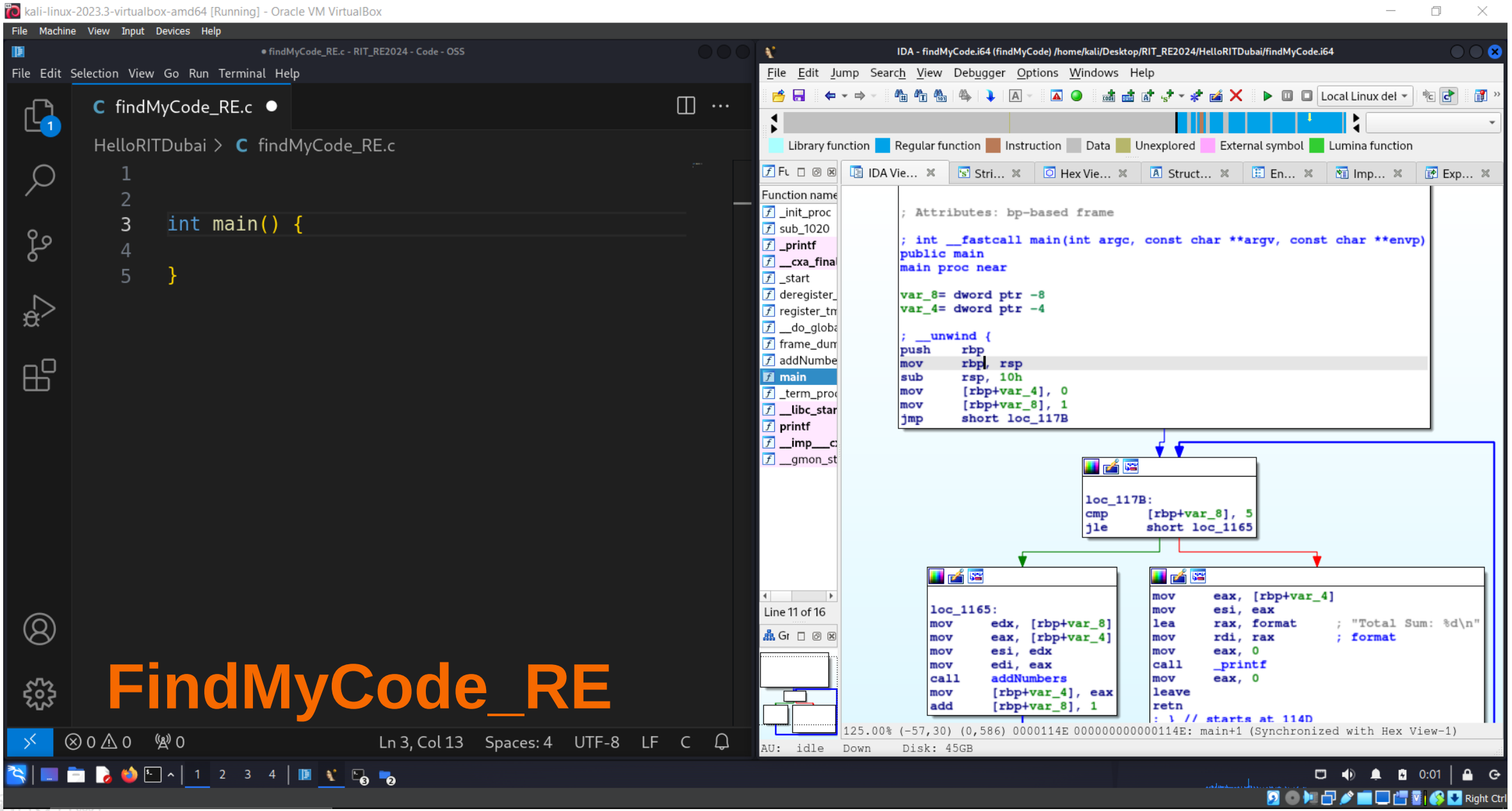
- **Function Call:** Within this loop, the function addNumbers is called with call addNumbers. The arguments are passed via registers: edx is loaded with the counter value, and eax with the sum, which are then moved to esi and edi respectively, adhering to the calling convention.
- Loop Continuation: The loop increments the counter with add [rbp+var_8], 1 and continues until the counter exceeds 5, as shown by cmp [rbp+var_8], 5 and jle short loc_1165.
- **Final Operation:** Upon loop termination, the main function prepares to print the total sum. It moves the sum into esi, sets up the format string, and calls printf to display the result.
- **Clean-up:** The function concludes by setting the return value to zero with mov eax, 0 and restoring the stack frame with leave, before finally returning with retn.

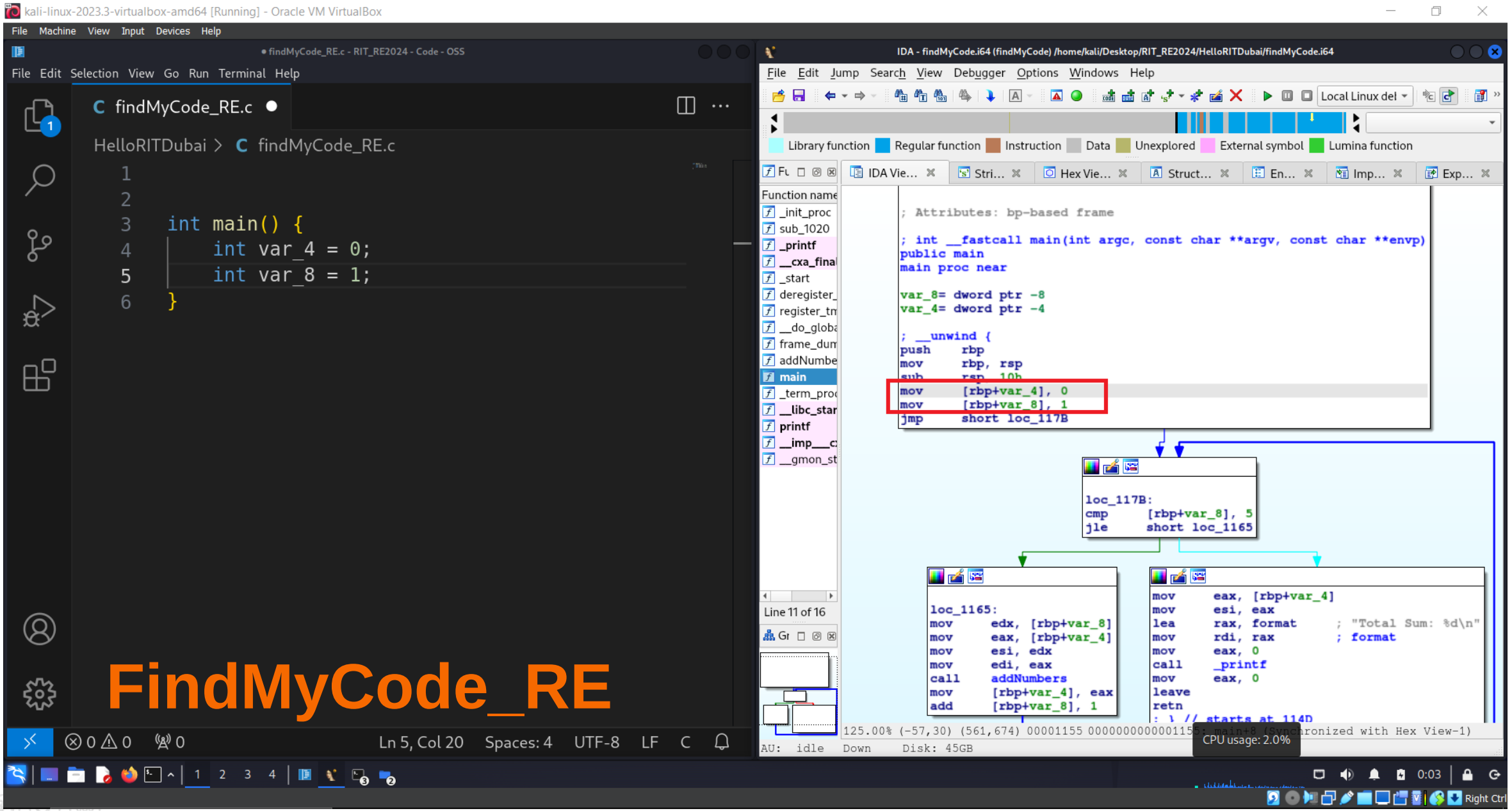
FindMyCode
addNumbers function

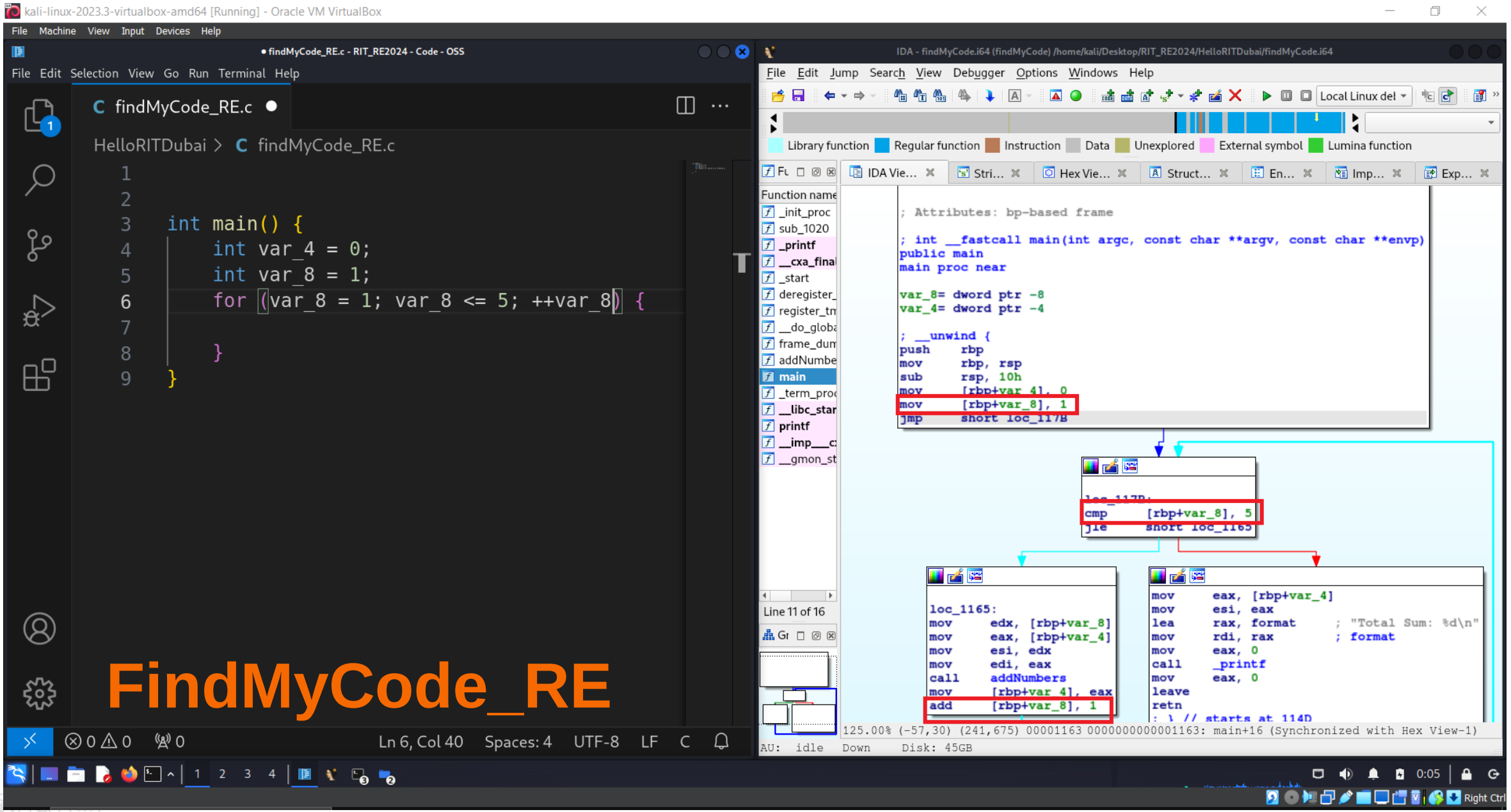
For the addNumbers function:

- **Parameter Reception**: addNumbers starts by receiving two parameters from main. These parameters are placed into the local stack space: mov [rbp+var_4], edi and mov [rbp+var_8], esi.

- **Addition and Return:** It then performs an addition with add eax, edx, where eax and edx hold the values of the two parameters. The result is left in eax, which is the register used for return values in this calling convention.


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
findMyCode_RE.c - RIT_RE2024 - Code - OSS
File Edit Selection View Go Run Terminal Help
findMyCode_RE.c
HelloRITDubai > findMyCode_RE.c
1
2
3 int main() {
4
5 }
Ln 3, Col 13 Spaces: 4 UTF-8 LF C
IDA - findMyCode.i64 (findMyCode) /home/kali/Desktop/RIT_RE2024/HelloRITDubai/findMyCode.i64
File Edit Jump Search View Debugger Options Windows Help
Local Linux del
Library function Regular function Instruction Data Unexplored External symbol Lumina function
IDA Vie... Stri... Hex Vie... Struct... En... Imp... Exp...
Function name
_init_proc
sub_1020
_printf
__cxa_fina
_start
deregister_
register_tm
__do_globa
frame_dum
addNumbe
main
_term_proc
__libc_star
printf
__imp___c
__gmon_st
Line 11 of 16
; Attributes: bp-based frame
; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near
var_8= dword ptr -8
var_4= dword ptr -4
; __unwind {
push rbp
mov rbp, rsp
sub rsp, 10h
mov [rbp+var_4], 0
mov [rbp+var_8], 1
jmp short loc_117B
loc_117B:
cmp [rbp+var_8], 5
jle short loc_1165
loc_1165:
mov edx, [rbp+var_8]
mov eax, [rbp+var_4]
mov esi, edx
mov edi, eax
call addNumbers
mov [rbp+var_4], eax
add [rbp+var_8], 1
mov eax, [rbp+var_4]
mov esi, eax
lea rax, format ; "Total Sum: %d\n"
mov rdi, rax ; format
mov eax, 0
call _printf
mov eax, 0
leave
retn
125.00% (-57,30) (0,586) 0000114E 000000000000114E: main+1 (Synchronized with Hex View-1)
AU: idle Down Disk: 45GB
0:01
Right Ctrl
FindMyCode_RE


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
findMyCode_RE.c - RIT_RE2024 - Code - OSS
File Edit Selection View Go Run Terminal Help
findMyCode_RE.c
HelloRITDubai > findMyCode_RE.c
1
2
3 int main() {
4 int var_4 = 0;
5 int var_8 = 1;
6 }
Ln 5, Col 20 Spaces: 4 UTF-8 LF C
IDA - findMyCode.i64 (findMyCode) /home/kali/Desktop/RIT_RE2024/HelloRITDubai/findMyCode.i64
File Edit Jump Search View Debugger Options Windows Help
Local Linux del
Library function Regular function Instruction Data Unexplored External symbol Lumina function
IDA Vie... Stri... Hex Vie... Struct... En... Imp... Exp...
Function name
_init_proc
sub_1020
_printf
__cxa_final
_start
deregister_
register_tm
__do_globa
frame_dum
addNumbe
main
_term_proc
__libc_star
printf
__imp___c
__gmon_st
Line 11 of 16
; Attributes: bp-based frame
; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near
var_8= dword ptr -8
var_4= dword ptr -4
; __unwind {
push rbp
mov rbp, rsp
sub rsp, 10h
mov [rbp+var_4], 0
mov [rbp+var_8], 1
jmp short loc_117B
loc_117B:
cmp [rbp+var_8], 5
jle short loc_1165
loc_1165:
mov edx, [rbp+var_8]
mov eax, [rbp+var_4]
mov esi, edx
mov edi, eax
call addNumbers
mov [rbp+var_4], eax
add [rbp+var_8], 1
mov eax, [rbp+var_4]
mov esi, eax
lea rax, format ; "Total Sum: %d\n"
mov rdi, rax ; format
mov eax, 0
call _printf
mov eax, 0
leave
retn
125.00% (-57,30) (561,674) 00001155 0000000000001155: main+8 (Synchronized with Hex View-1)
AU: idle Down Disk: 45GB
CPU usage: 2.0%
0:03
Right Ctrl
FindMyCode_RE


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
• findMyCode_RE.c - RIT_RE2024 - Code - OSS
File Edit Selection View Go Run Terminal Help
findMyCode_RE.c
HelloRITDubai > findMyCode_RE.c
int main() {
int var_4 = 0;
int var_8 = 1;
for (var_8 = 1; var_8 <= 5; ++var_8) {
}
}
Ln 6, Col 40 Spaces: 4 UTF-8 LF C
IDA - findMyCode.i64 (findMyCode) /home/kali/Desktop/RIT_RE2024/HelloRITDubai/findMyCode.i64
File Edit Jump Search View Debugger Options Windows Help
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Function name
_init_proc
sub_1020
_printf
__cxa_final
_start
deregister_
register_tm
__do_globa
frame_dum
addNumbe
main
_term_proc
__libc_star
printf
__imp___c
__gmon_st
Line 11 of 16
; Attributes: bp-based frame
; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near
var_8= dword ptr -8
var_4= dword ptr -4
; __unwind {
push rbp
mov rbp, rsp
sub rsp, 10h
mov [rbp+var_4], 0
mov [rbp+var_8], 1
jmp short loc_117B
cmp [rbp+var_8], 5
jle short loc_1165
loc_1165:
mov edx, [rbp+var_8]
mov eax, [rbp+var_4]
mov esi, edx
mov edi, eax
call addNumbers
mov [rbp+var_4], eax
add [rbp+var_8], 1
mov eax, [rbp+var_4]
mov esi, eax
lea rax, format ; "Total Sum: %d\n"
mov rdi, rax ; format
mov eax, 0
call _printf
mov eax, 0
leave
retn
125.00% (-57,30) (241,675) 00001163 0000000000001163: main+16 (Synchronized with Hex View-1)
AU: idle Down Disk: 45GB
FindMyCode_RE

kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

• findMyCode_RE.c - RIT_RE2024 - Code - OSS

File Edit Selection View Go Run Terminal Help

findMyCode_RE.c

HelloRITDubai > findMyCode_RE.c

```
1
2   int addNumbers(int var_4, int var_8);
3
4   int main() {
5       int var_4 = 0;
6       int var_8 = 1;
7       for (var_8 = 1; var_8 <= 5; ++var_8) {
8
9           var_4 = addNumbers(var_4, var_8);
10
11      }
12
13  }
14
15
16  int addNumbers(int var_4, int var_8) {
17
18
19  };
```

FindMyCode_RE

0 0 0 Ln 13, Col 2 Spaces: 4 UTF-8 LF C

IDA - findMyCode.i64 (findMyCode) /home/kali/Desktop/RIT_RE2024/HelloRITDubai/findMyCode.i64

File Edit Jump Search View Debugger Options Windows Help

Local Linux del

Library function Regular function Instruction Data Unexplored External symbol Lumina function

IDA Vie... Stri... Hex Vie... Struct... En... Imp... Exp...

Function name

_init_proc
sub_1020
_printf
__cxa_final
_start
deregister_
register_tm
__do_globa
frame_dum
addNumbe
main
_term_proc
__libc_star
printf
__imp___c
__gmon_st

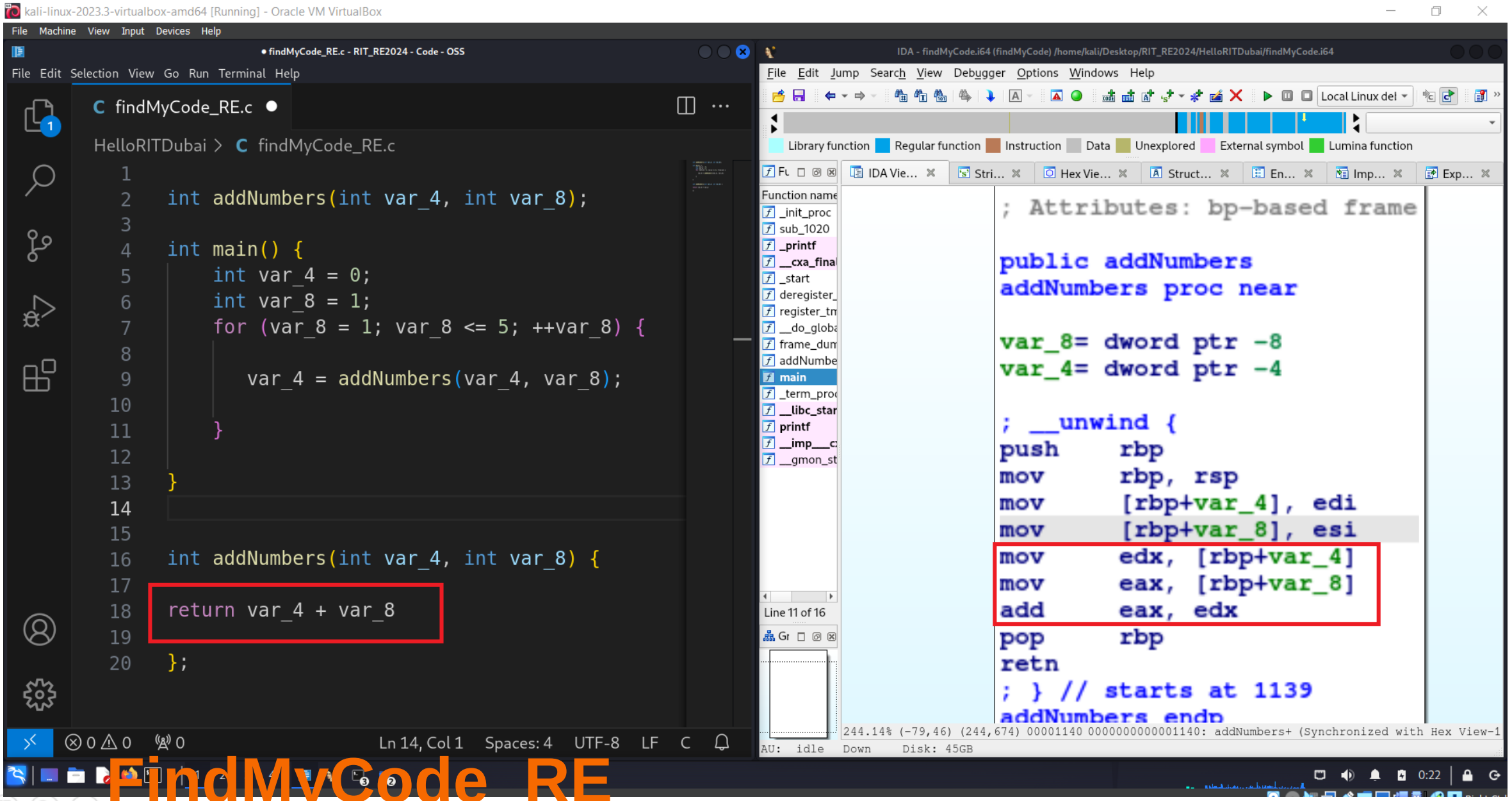
Line 11 of 16

```
; Attributes: bp-based frame

public addNumbers
addNumbers proc near

var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
pop     rbp
retn
; } // starts at 1139
addNumbers endp
```

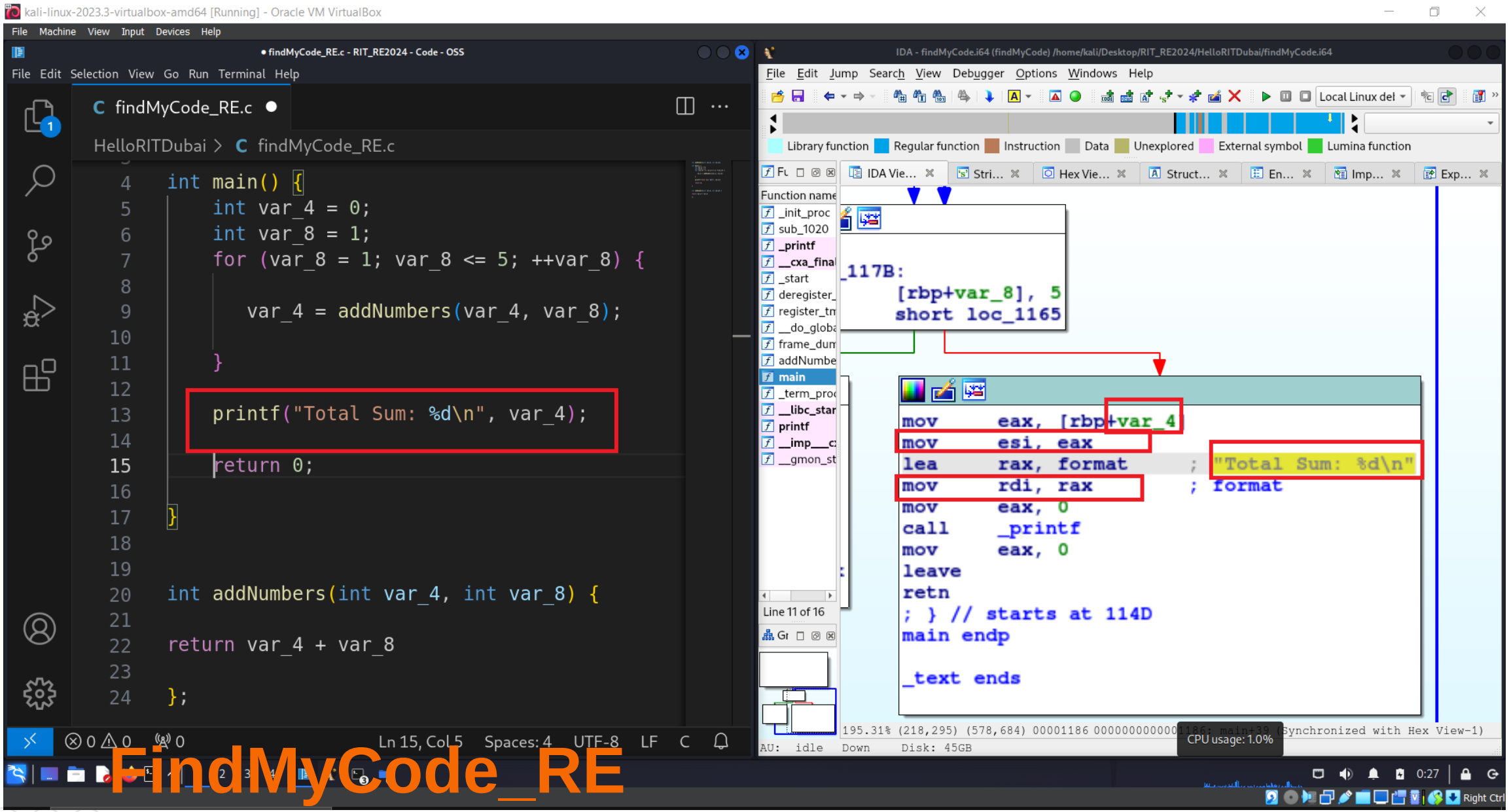
244.14% (-79,46) (175,635) 00001140 0000000000001140: addNumbers+ (Synchronized with Hex View-1

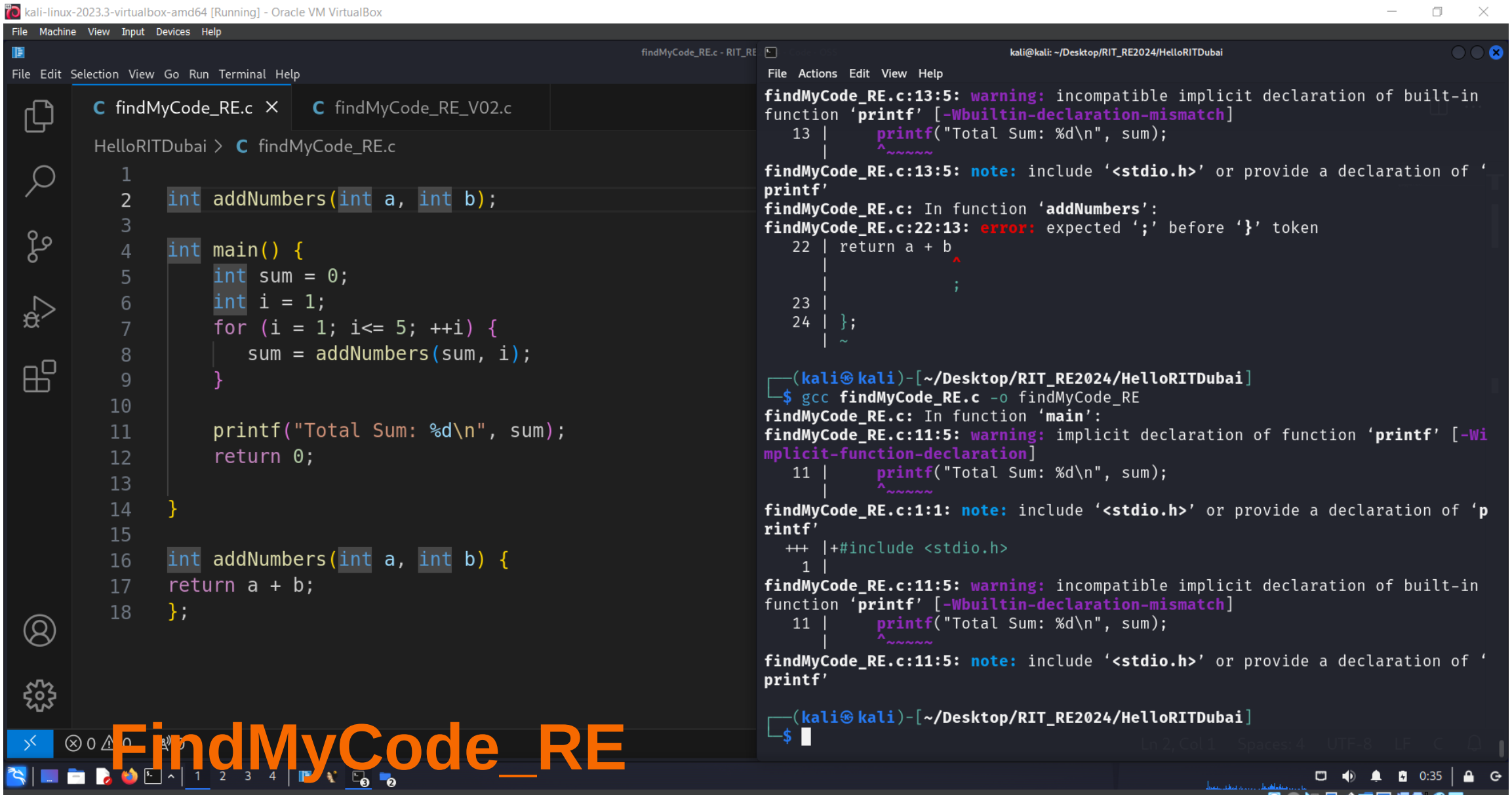
AU: idle Down Disk: 45GB

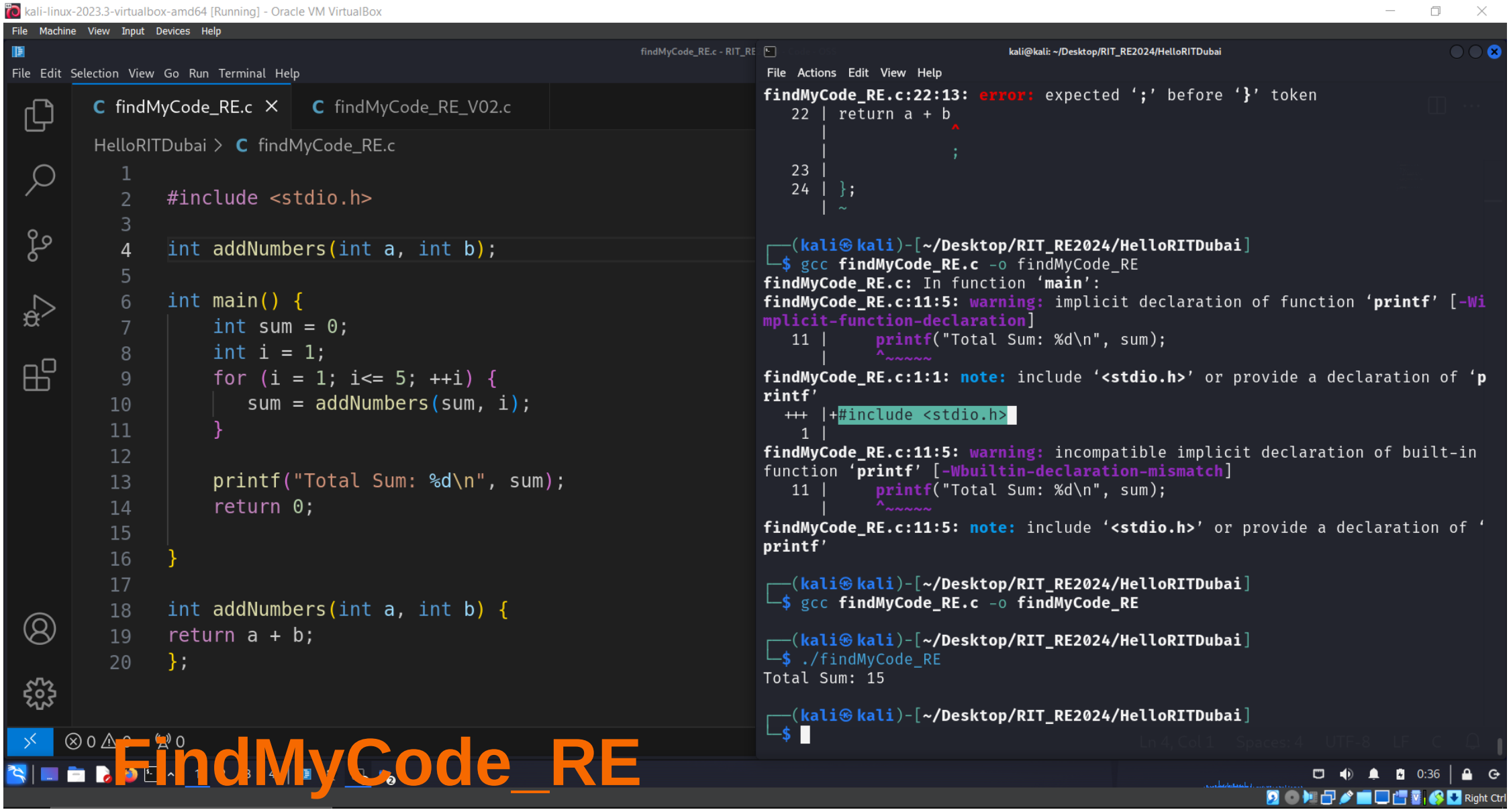
1 2 3 4 0:19



FindMyCode_RE


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
• findMyCode_RE.c - RIT_RE2024 - Code - OSS
File Edit Selection View Go Run Terminal Help
findMyCode_RE.c
HelloRITDubai > findMyCode_RE.c
int main() {
int var_4 = 0;
int var_8 = 1;
for (var_8 = 1; var_8 <= 5; ++var_8) {
var_4 = addNumbers(var_4, var_8);
}
printf("Total Sum: %d\n", var_4);
return 0;
}
int addNumbers(int var_4, int var_8) {
return var_4 + var_8
};
Ln 15, Col 5 Spaces: 4 UTF-8 LF C
IDA - findMyCode.i64 (findMyCode) /home/kali/Desktop/RIT_RE2024/HelloRITDubai/findMyCode.i64
File Edit Jump Search View Debugger Options Windows Help
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Function name
_init_proc
sub_1020
_printf
_start
deregister_
register_tm
__do_globa
frame_dum
addNumbe
main
_term_proc
__libc_star
printf
__gmon_st
Line 11 of 16
_117B:
[rbp+var_8], 5
short loc_1165
mov eax, [rbp+var_4]
mov esi, eax
lea rax, format ; "Total Sum: %d\n"
mov rdi, rax ; format
mov eax, 0
call _printf
mov eax, 0
leave
retn
; } // starts at 114D
main endp
_text ends
CPU usage: 1.0%
AU: idle Down Disk: 45GB
FindMyCode_RE


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
File Edit Selection View Go Run Terminal Help
findMyCode_RE.c
findMyCode_RE_V02.c
HelloRITDubai > findMyCode_RE.c
1 int addNumbers(int a, int b);
2
3 int main() {
4 int sum = 0;
5 int i = 1;
6 for (i = 1; i<= 5; ++i) {
7 sum = addNumbers(sum, i);
8 }
9
10 printf("Total Sum: %d\n", sum);
11 return 0;
12
13 }
14
15 int addNumbers(int a, int b) {
16 return a + b;
17 };
18
kali@kali: ~/Desktop/RIT_RE2024/HelloRITDubai
File Actions Edit View Help
findMyCode_RE.c:13:5: warning: incompatible implicit declaration of built-in function 'printf' [-Wbuiltin-declaration-mismatch]
13 | printf("Total Sum: %d\n", sum);
| ^~~~~~
findMyCode_RE.c:13:5: note: include <stdio.h> or provide a declaration of 'printf'
findMyCode_RE.c: In function 'addNumbers':
findMyCode_RE.c:22:13: error: expected ';' before '}' token
22 | return a + b
| ^
| ;
23 |
24 | };
| ~
(kali㉿kali)-[~/Desktop/RIT_RE2024/HelloRITDubai]
$ gcc findMyCode_RE.c -o findMyCode_RE
findMyCode_RE.c: In function 'main':
findMyCode_RE.c:11:5: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
11 | printf("Total Sum: %d\n", sum);
| ^~~~~~
findMyCode_RE.c:1:1: note: include <stdio.h> or provide a declaration of 'printf'
+++ |+#include <stdio.h>
1 |
findMyCode_RE.c:11:5: warning: incompatible implicit declaration of built-in function 'printf' [-Wbuiltin-declaration-mismatch]
11 | printf("Total Sum: %d\n", sum);
| ^~~~~~
findMyCode_RE.c:11:5: note: include <stdio.h> or provide a declaration of 'printf'
(kali㉿kali)-[~/Desktop/RIT_RE2024/HelloRITDubai]
$
FindMyCode_RE


kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
File Edit Selection View Go Run Terminal Help
findMyCode_RE.c
findMyCode_RE_V02.c
HelloRITDubai > findMyCode_RE.c
#include <stdio.h>
int addNumbers(int a, int b);
int main() {
int sum = 0;
int i = 1;
for (i = 1; i<= 5; ++i) {
sum = addNumbers(sum, i);
}
printf("Total Sum: %d\n", sum);
return 0;
}
int addNumbers(int a, int b) {
return a + b;
};
kali@kali: ~/Desktop/RIT_RE2024/HelloRITDubai
File Actions Edit View Help
findMyCode_RE.c:22:13: error: expected ';' before '}' token
22 | return a + b
| ^
| ;
23 |
24 | };
| ~
(kali㉿kali)-[~/Desktop/RIT_RE2024/HelloRITDubai]
$ gcc findMyCode_RE.c -o findMyCode_RE
findMyCode_RE.c: In function 'main':
findMyCode_RE.c:11:5: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
11 | printf("Total Sum: %d\n", sum);
| ^~~~~~
findMyCode_RE.c:1:1: note: include '<stdio.h>' or provide a declaration of 'printf'
+++ |+#include <stdio.h>
1 |
findMyCode_RE.c:11:5: warning: incompatible implicit declaration of built-in function 'printf' [-Wbuiltin-declaration-mismatch]
11 | printf("Total Sum: %d\n", sum);
| ^~~~~~
findMyCode_RE.c:11:5: note: include '<stdio.h>' or provide a declaration of 'printf'
(kali㉿kali)-[~/Desktop/RIT_RE2024/HelloRITDubai]
$ gcc findMyCode_RE.c -o findMyCode_RE
(kali㉿kali)-[~/Desktop/RIT_RE2024/HelloRITDubai]
$ ./findMyCode_RE
Total Sum: 15
(kali㉿kali)-[~/Desktop/RIT_RE2024/HelloRITDubai]
$
FindMyCode_RE

kali-linux-2023.3-virtualbox-amd64 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

findMyCode.c - RIT_RE2024 - Code - OSS

File Edit Selection View Go Run Terminal Help

findMyCode_RE.c | findMyCode_RE_V02.c

HelloRITDubai > findMyCode_RE.c

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main() {
    int sum = 0;
    int i = 1;
    for (i = 1; i<= 5; ++i) {
        sum = addNumbers(sum, i);
    }

    printf("Total Sum: %d\n", sum);
    return 0;

}

int addNumbers(int a, int b) {
return a + b;
};
```

findMyCode.c

```c
#include <stdio.h>

int addNumbers(int a, int b) {
    return a + b;
}

int main() {
    int totalSum = 0;
    for (int i = 1; i <= 5; i++) {
        totalSum = addNumbers(totalSum, i);
    }
    printf("Total Sum: %d\n", totalSum);
    return 0;
}
```

FindMyCode_RE

FindMyCode

CPU usage: 2.0%

Ln 5 ... es: 4 UTF-8 LF C

0:54

Right Ctrl

Course Overview

- **Title: "CSEC 202 - Reverse Engineering Fundamentals"**

Instructor	Office	Phone	Email	Semester-Year
Emad Abu Khousa	D003		eakcad@rit.edu	Spring-2024
Office Hours:	M: 12:00-01:00 TR: 11:00-12:00			

- **600: TR 12:00-01:20, Room B-107**
- **601: MW 01:05-02:25, Room C-109**
- **602: TR 01:30-02:50, Room D-207**

Thank You and Q&A