جامعة روتشستر الأمريكية للتكنولوجيا في دبي
A Global American University in Dubai

CSEC 202 Reverse Engineering Fundamentals

Module 0x04
Basics of assembly
{a lot of x86-32} & {some of x86-64}
Hands-On 0x01

Eng. Emad Abu Khousa

Sections: 600 | 601 | 602

February 19, 2024

Learning Objectives*

1. Recognize and define x86 assembly language
2. Distinguish the difference between 16/32/64-bit assembly code
3. Explore sections of executable file
4. Differentiate Intel and AT&T syntax
5. Explain Stack and Heap memory allocations.

* Adopted from: Saltaformaggio, B. (2022). *Introduction to assembly language*. Georgia Tech.

Native – vs. - Compiled

```
section .text
   global _start     ;must be declared for linker (ld)

_start:             ;tells linker entry point
   mov  edx,len     ;message length
   mov  ecx,msg     ;message to write
   mov  ebx,1       ;file descriptor (stdout)
   mov  eax,4       ;system call number (sys_write)
   int  0x80        ;call kernel

   mov  eax,1       ;system call number (sys_exit)
   int  0x80        ;call kernel

section .data
msg db 'Hello, world!', 0xa  ;string to be printed
len equ $ - msg     ;length of the string
```

```
segment .text       ;code segment
   global _start    ;must be declared for linker

_start:             ;tell linker entry point
   mov edx,len      ;message length
   mov ecx,msg      ;message to write
   mov ebx,1        ;file descriptor (stdout)
   mov eax,4        ;system call number (sys_write)
   int 0x80         ;call kernel

   mov eax,1        ;system call number (sys_exit)
   int 0x80         ;call kernel

segment .data       ;data segment
msg     db 'Hello, world!',0xa   ;our dear string
len     equ    $ - msg           ;length of our dear string
```

Live Demo

https://www.tutorialspoint.com/assembly_programming/index.htm

Native – vs. - Compiled

```
section .text
    global _start     ;must be declared for linker (ld)

_start:               ;tells linker entry point
    mov  edx,len      ;message length
    mov  ecx,msg      ;message to write
    mov  ebx,1        ;file descriptor (stdout)
    mov  eax,4        ;system call number (sys_write)
    int  0x80         ;call kernel

    mov  eax,1        ;system call number (sys_exit)
    int  0x80         ;call kernel

section .data
msg db 'Hello, world!', 0xa  ;string to be printed
len equ $ - msg       ;length of the string
```

```c
#include <stdio.h>

int main() {
    printf("Hello, world!");
    return 0;
}
```

```
    .file   "hellow.c"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "Hello, world!"
    .text
    .globl  main
    .type   main, @function
main:
    push    rbp
    mov rbp, rsp
    lea rax, .LC0[rip]
    mov rdi, rax
    mov eax, 0
    call    printf@PLT
    mov eax, 0
    pop rbp
    ret
```

Hands-On 0x01

Target:

Write an assembly code that prompts the user to enter multiple positive grades. Whenever the user enters a number, the main function checks if that grade is less than 100. If the grade is not withing that range it print an error message. The main function should also check if the grade is equal to, or greater than 50 (pass course) and then calls another function to increase the counter of passed courses, otherwise it increases the counter of failed courses. The iteration stops when the user enters a negative grade (-1). Before exiting the program, the main function calls another function to print the result of the counters for passed and failed courses provided by the user.

Analyzing the requirements of the program:

1. **Input Handling**:
    1. The program needs to prompt the user to enter multiple positive grades.
    2. It should handle the input validation to ensure that the entered grade is within the range of 0 to 100.
2. **Grade Evaluation**:
    1. Once a valid grade is entered, the main function should evaluate if the grade is within the passing range (greater than or equal to 50) or not.
    2. Based on the evaluation, it should update counters for passed and failed courses accordingly.
3. **Looping and Termination**:
    1. The iteration should continue until the user enters a negative grade (-1).
4. **Output**:
    1. Before exiting the program, it should print the results of the counters for passed and failed courses.

Analyzing the requirements of the program:

- **Input/Output Operations:** The program must prompt the user for input and display messages.
- **Conditional Statements:** It should use conditions to check:
  - If a grade is within a certain range (less than 100).
  - If a grade is a passing grade (greater than or equal to 50).
- **Loops:** Implement a loop to continuously prompt for grades until a negative grade is entered.
- **Function Calls: Use functions to:**
  - Increase the counter of passed or failed courses.
  - Print the result of the counters.
- **Arithmetic Operations:** Perform basic increment operations on counters.
- **Data Storage:** Manage storage for grades, and counters for passed and failed courses.

Hands-on Task Breakdown:

1. **Prompt for Grade Input:** Use system calls or platform-specific methods to read user input.
2. **Validate Grade:** Check if the grade is less than 100. If not, output an error message.
3. **Check Pass/Fail Condition:** Determine if the grade constitutes a pass or fail and increment the appropriate counter.
4. **Implement a Loop:** Ensure the program continues to prompt for grades until a negative grade is entered.
5. **Function to Increase Counters:** Implement functions to handle incrementing the pass and fail counters.
6. **Function to Print Results:** Create a function that outputs the counts of passed and failed courses.
7. **Negative Grade to Terminate:** Use a negative grade as a sentinel value to exit the loop.
8. **Test and Debug**: Encourage testing with various inputs to ensure the program behaves as expected.

Hands-on Training Tasks:

1. Task 1: Hello World Program
2. Task 2: Reading User Input
3. Task 3: Basic Arithmetic Operations
4. Task 4: Using Conditional Statements
5. Task 5: Loop Constructs
6. Task 6: Function Calls
7. Task 7: Implementing a Counter
8. Task 8: Combining Loops and Conditionals
9. Task 9: Advanced Input Validation
10. Task 10: Complete Program Assembly

Hands-on Training Tasks:

Guidance for Completion

- **Start Simple:** Begin with basic programs to build your foundational skills.
- **Incremental Learning:** Each task builds upon the previous, so complete them in order.
- **Test and Debug:** Regularly test your programs for different scenarios to ensure they behave as expected.
- **Consult Resources:** Use course materials, textbooks, and online resources to aid in solving tasks.
- **Seek Feedback:** Discuss your solutions with peers or instructors to gain insights and improve your code.

Hands-on Training Tasks:

Task 1: Hello World Program

- **Objective:** Learn the basic syntax of assembly language and how to output text.
- **Task:** Write an assembly/C program that outputs "Hello, World!" to the console
- **Languages:** C and Native Assembly
- **Reverse Engineering:** Compare the compiled C with the native code.

Hands-on Training Tasks:

Task 1: Hello World Program

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
 return 0;
}
```

$ gcc hello.c -o hello

Or

$gcc –m32 hello.c -o hello

Or

$ gcc -m32 -W -Wall -Wextra -Wpedantic -fno-asynchronous-unwind-tables -no-pie -O0 -fno-pie -fno-pic hello.c -o hello_32_clean

Structuring the Hands-on Training:

```
section .data
    helloMessage db 'Hello, World!',0xA  ; Define the message with a newline character at the end
    helloLen equ $-helloMessage          ; Calculate the length of the message

section .text
    global _start                        ; The entry point for the program

_start:
                                         ; Write the message to stdout
    mov eax, 4                           ; The system call number for sys_write (4)
    mov ebx, 1                           ; File descriptor 1 is stdout
    mov ecx, helloMessage                ; The message to write
    mov edx, helloLen                    ; The length of the message
    int 0x80                             ; Make the kernel call

    ; Exit the program
    mov eax, 1                           ; The system call number for sys_exit (1)
    mov ebx, 0                           ; Exit code 0
    int 0x80                             ; Make the kernel call
```

Structuring the Hands-on Training:

```
section .data

section .text

_start:
```

Structuring the Hands-on Training:

```
section .data
    helloMessage db 'Hello, World!',0xA  ; Define the message with a newline character at the end
    helloLen equ $-helloMessage           ; Calculate the length of the message
```

- **Data Section:** This is where your program's data (constants, strings, etc.) is defined. It's not executable but is accessed by the executable sections of your program.

- **helloMessage**: A label pointing to a byte array containing the "Hello, World!" string followed by a newline character (0xA, which is ASCII for newline).

- **helloLen:** Uses the equ directive to calculate the length of helloMessage by subtracting the current address (**$**) from the address where helloMessage starts. This effectively gives the size of the helloMessage string in bytes.

Structuring the Hands-on Training:

```
section .text
    global _start                         ; The entry point for the program

_start:
```

Section Directive

section .text: This directive tells the assembler to place the following instructions in the .text section of the output file. The .text section is where executable code resides.

Global Directive

global _start: The global directive makes the symbol _start visible outside of this file, which is necessary for the linker to recognize _start as the entry point of the program.

Label

_start:: This line defines a label named _start. Labels in assembly language serve as markers or placeholders that represent addresses in memory. When the program is assembled and linked, _start will be associated with the memory address where the following instructions begin.

Structuring the Hands-on Training:

```
mov eax, 4              ; The system call number for sys_write (4)
mov ebx, 1              ; File descriptor 1 is stdout
mov ecx, helloMessage   ; The message to write
mov edx, helloLen       ; The length of the message
int 0x80                ; Make the kernel call
```

What Does int 0x80 Do?

- int: This stands for "interrupt," a powerful feature of x86 CPUs.
- interrupt number 0x80 is designated for **system calls.**

Before int 0x80 is Executed:

The program sets up specific registers with the necessary information for the system call:

- EAX: Contains the system call number (e.g., 4 for sys_write, 1 for sys_exit).
- EBX, ECX, EDX, ...: These registers are used to pass arguments to the system call.

Structuring the Hands-on Training:

General System Call Number Summary (for x86 Linux)

1: sys_exit - Terminates the current process.
2: sys_fork - Creates a new process (child process).
3: sys_read - Reads data from a file descriptor into a buffer.
4: sys_write - Writes data from a buffer to a file descriptor.
5: sys_open - Opens a file and returns a file descriptor.
6: sys_close - Closes a file descriptor.
7: sys_waitpid - Waits for a child process to change state.
8: sys_creat - Creates a new file or rewrites an existing one.
9: sys_link - Creates a new link (hard link) to an existing file.
10: sys_unlink - Removes a directory entry (deletes a file name).
11: sys_execve - Executes a program.
12: sys_chdir - Changes the current working directory.
13: sys_time - Gets the current time.
45: sys_brk - Changes the space allocated for the calling process's data segment.
85: sys_readlink - Reads the value of a symbolic link.
91: sys_munmap - Unmaps a file or device from memory.
122: sys_uname - Gets system information.
145: sys_readv - Read vectors from a file descriptor.
146: sys_writev - Write vectors to a file descriptor.
252: sys_exit_group - Exits all threads in a process.

Structuring the Hands-on Training:

sys_write: System Call Number: 4

Purpose: Writes data to a file descriptor from a buffer.

sys_write (fd, buf, count)

Parameters:

1. File Descriptor (**fd**): Think of it as a special number that your program gets when it opens a file or starts up. It tells the system where you want to write your data. For example, 1 always means "**write to the terminal window**.“
2. Buffer (**buf**): This is where your data is stored before it gets written. If you want to write "Hello", your buffer contains the letters of "Hello".
3. Count (**count**): This tells how many letters (or bytes) you want to write from your buffer.

Usage:

```
mov eax, 4;
mov ebx, fd;
mov ecx, buf;
mov edx, count;
int 0x80
```

Structuring the Hands-on Training:

sys_read: System Call Number: 3

Purpose: Reads data from a file descriptor into a buffer.
Parameters:

1. unsigned int fd: File descriptor to read from.
2. char *buf: Buffer to read the data into.
3. size_t count: Number of bytes to read.

Usage:

```
mov eax, 3;
mov ebx, fd;
mov ecx, buf;
mov edx, count;
int 0x80
```

Structuring the Hands-on Training:

```
; Exit the program
mov eax, 1                    ; The system call number for sys_exit (1)
mov ebx, 0                    ; Exit code 0
int 0x80                      ; Make the kernel call
```

mov eax, 1
The number 1 corresponds to the sys_exit system call, which is used to terminate a process.

mov ebx, 0
When making the sys_exit system call, ebx holds the exit status of the process. An exit status of 0 typically indicates that the program terminated successfully without any errors.

int 0x80
When this interrupt is triggered, the kernel looks at the value in the eax register to determine which system call to execute, and then it uses the values in other registers (ebx in this case) as arguments to that system call.

Resources:

Setup the environment on your VM:

https://www.tutorialspoint.com/assembly_programming/assembly_environment_setup.htm

Use online Emulators:

https://www.tutorialspoint.com/compile_asm_online.php

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

NASM website:

https://www.nasm.us/

Structuring the Hands-on Training:

Steps to Compile and Run

To compile and run this program on a Linux system, you would typically follow these steps:

Compile the Program: Use NASM to compile the assembly code. Open a terminal and run the following command:

$nasm -f elf hello.asm

This command compiles the hello.asm file into an object file hello.o in ELF (Executable and Linkable Format).

Link the Object File: Link the object file to create an executable. You can use the ld linker for this step:

$ld -m elf_i386 -s -o hello hello.o

This creates an executable named hello. The -m elf_i386 option specifies the target architecture, which is important for compatibility reasons.

Run the Program: Finally, run the executable by typing:

$./hello

This should display "Hello, World!" in the terminal.

Structuring the Hands-on Training:

```
section .data
    helloMessage db 'Hello, World!',0xA  ; Define the message with a newline character at the end
    helloLen equ $-helloMessage          ; Calculate the length of the message

section .text
    global _start                        ; The entry point for the program

_start:

        ; Write the message to stdout
    mov eax, 4                           ; The system call number for sys_write (4)
    mov ebx, 1                           ; File descriptor 1 is stdout
    mov ecx, helloMessage                ; The message to write
    mov edx, helloLen                    ; The length of the message
    int 0x80                             ; Make the kernel call

        ; Exit the program
    mov eax, 1                           ; The system call number for sys_exit (1)
    mov ebx, 0                           ; Exit code 0
    int 0x80                             ; Make the kernel call
```

```
$nasm -f elf hello.asm
$ld -m elf_i386 -s -o hello hello.o
$./hello
```

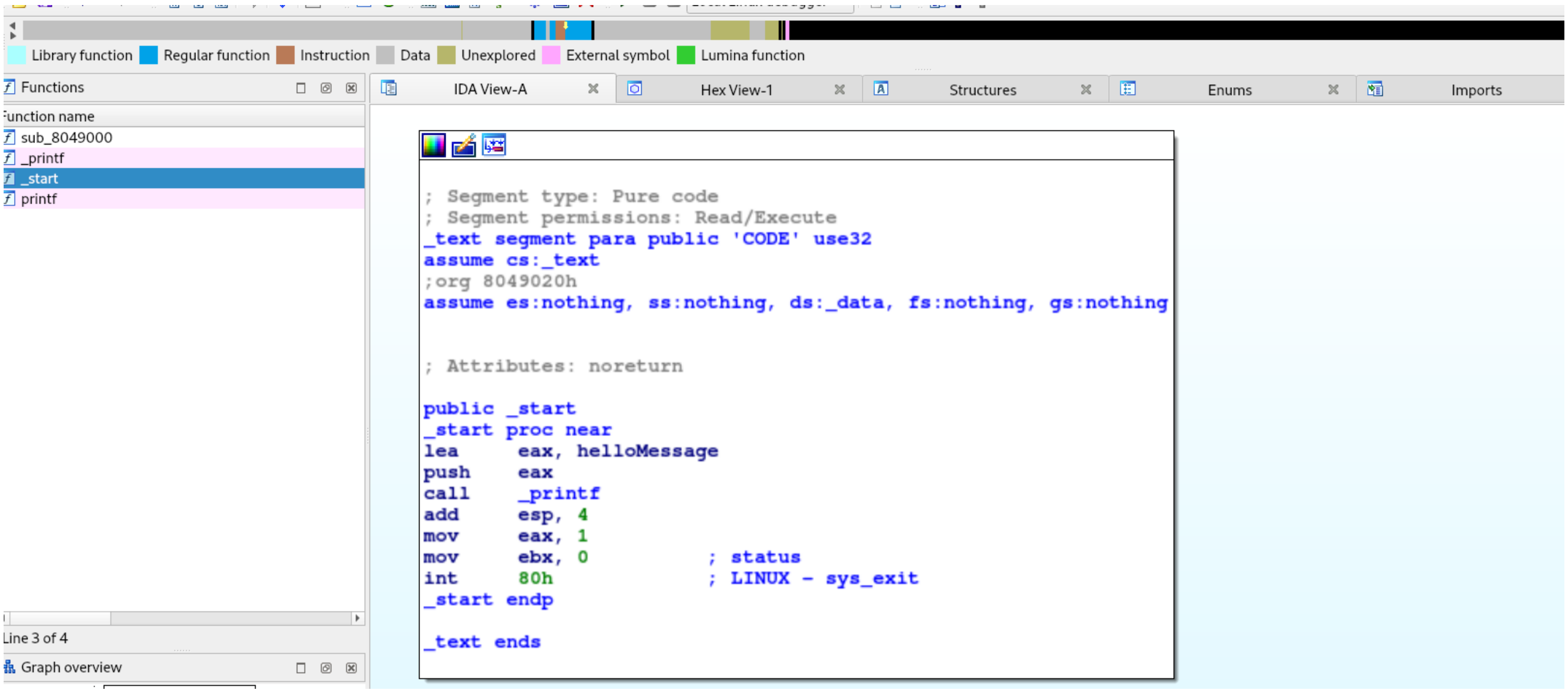
Can we mix?:

```
section .data
    helloMessage db 'Hello, World!', 0xA  ; Null-terminated string with a newline

section .text
    global _start                       ; The entry point for the program
    extern printf                       ; External declaration of printf

_start:

    lea eax, [helloMessage]             ; Load the effective address of helloMessage into EAX
    push eax                            ; Push the value in EAX (the address) onto the stack
    call printf                         ; Call the printf function
    add esp, 4                          ; Clean up the stack (pop the argument)

    ; Exit the program using sys_exit
    mov eax, 1                          ; The system call number for sys_exit (1)
    mov ebx, 0                          ; Exit code 0
    int 0x80                            ; Make the kernel call
```

```
$ nasm -f elf32 -o hello.o hello.asm
$ gcc -m32 -nostartfiles -no-pie -o hello hello.o
```

```
$ sudo apt update
$  sudo apt install gcc-multilib g++-multilib
```

Can we mix?:


Library function
Regular function
Instruction
Data
Unexplored
External symbol
Lumina function
Functions
IDA View-A
Hex View-1
Structures
Enums
Imports
Function name
sub_8049000
_printf
_start
printf
Line 3 of 4
Graph overview
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
;org 8049020h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
; Attributes: noreturn
public _start
_start proc near
lea eax, helloMessage
push eax
call _printf
add esp, 4
mov eax, 1
mov ebx, 0 ; status
int 80h ; LINUX - sys_exit
_start endp
_text ends

Hands-on Training Tasks:

Task 2: Reading User Input

- **Objective:** Understand how to read a single number input from the user.
- **Task:** Create an assembly/C program that prompts the user for a number and then displays that number back to the user.
- **Reverse Engineering:** Compare the compiled C with the native code.

Hands-on Training Tasks:

Task 2: Reading User Input

C task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}

```

```
┌──(kali㉿kali)-[~]
└─$ gcc -m32 task2scan.c -o task2scan
```

Task2 – Scan - Native:

```
section .data

section .bss

section .text
        global _start

_start:
        ; Display the prompt to the user
        ; Read the grade from user input
        ; Write the output message
        ; Echo the grade back to stdout
        ; Exit the program
```

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

Task2 – Scan - Native:

section .data

```
msg db 'Please enter your grade (0-100): ', 0
lenMsg equ $ - msg

outMsg db 'You entered this grade: '
lenOutMsg equ $ - outMsg
```

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

RIT | Rochester Institute of Technology of Dubai

Task2 – Scan - Native:

```
section .data
msg db 'Please enter your grade (0-100): ', 0
lenMsg equ $ - msg
outMsg db 'You entered this grade: '
lenOutMsg equ $ - outMsg

section .bss
grade resb 5 ; Reserve space for grade input, including '\n' and null terminator
```

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

resb: This stands for "**reserve byte(s)**" in NASM syntax. It's an assembler directive used to allocate space in memory without initializing it. The space allocated is intended for byte-sized elements. Other similar directives include resw for word-sized elements (2 bytes), resd for double wo

Task2 – Scan - Native:

```
section .data
section .bss
section .text
        global _start

_start:
        ; Display the prompt to the user

mov eax, 4                 ; sys_write system call
mov ebx, 1                 ; File descriptor 1 (stdout)
mov ecx, msg               ; Pointer to the message
mov edx, lenMsg            ; Message length
int 0x80                   ; Call kernel
```

task2scan.c

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

Task2 – Scan - Native:

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

```
section .data

section .bss

section .text
        global _start

_start:
        ; Read the grade from user input
mov eax, 3          ; sys_read system call
mov ebx, 0          ; File descriptor 0 (stdin)
mov ecx             , grade ; Pointer to the buffer where the input will be stored
mov edx, 5          ; Max number of bytes to read, accommodating up to 3 digits, newline, and null terminator
int 0x80            ; Call kernel
```

Task2 – Scan - Native:

```
section .data

section .bss

section .text
        global _start

_start:
        ; Display the prompt to the user
        ; Read the grade from user input
        ; Write the output message
        ; Echo the grade back to stdout
        ; Exit the program
```

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

Task2 – Scan - Native:

```
section .data

section .bss

section .text
	global _start

_start:
	; Write the output message
mov eax, 4			; sys_write system call
mov ebx, 1			; File descriptor 1 (stdout)
mov ecx, outMsg		; Pointer to the output message
mov edx, lenOutMsg		; Output message length
int 0x80 ; Call kernel
```

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

RIT Rochester Institute of Technology of Dubai

Task2 – Scan - Native:

```
section .data

section .bss

section .text
        global _start

_start:
        ; Echo the grade back to stdout
mov eax, 4              ; sys_write system call
mov ebx, 1              ; File descriptor 1 (stdout)
mov ecx, grade          ; Pointer to the buffer with the grade
mov edx, 5              ; Using 5 bytes as a maximum length to echo back
int 0x80 ; Call kernel
```

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

Task2 – Scan - Native:

```
section .data

section .bss

section .text
	global _start

_start:
	; Exit the program
mov eax, 1		; sys_exit system call
xor ebx, ebx		; Exit code 0
int 0x80	; Call kernel
```

task2scan.c

task2scan.c

```c
#include <stdio.h>

int main() {
    int grade;
    printf("Please enter your grade (0-100): ");
    scanf("%d", &grade);
    printf("You entered this grade: %d\n", grade);
    return 0;
}
```

Task2 – Scan - Native:

```
section .data
msg db 'Please enter your grade (0-100): ', 0
lenMsg equ $ - msg
outMsg db 'You entered this grade: '
lenOutMsg equ $ - outMsg

section .bss
grade resb 5

section .text
global _start

_start:

; Display the prompt to the user

mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, lenMsg
int 0x80

; Read the grade from user input
mov eax, 3
mov ebx, 0
mov ecx, grade
mov edx, 5
int 0x80

; Write the output message
mov eax, 4
mov ebx, 1
mov ecx, outMsg
mov edx, lenOutMsg
int 0x80

; Echo the grade back to stdout
mov eax, 4
mov ebx, 1
mov ecx, grade
mov edx, 5
int 0x80
; Exit the program
mov eax, 1
xor ebx, ebx
int 0x80
```

```
$ nasm -f elf32 -o task2scanasm.o task2scanasm.asm
$ gcc -m32 -nostartfiles -no-pie -o task2scanasm task2scanasm.o
```

Task 2 C+ native:

```
section .data
msg1 db 'Please enter your grade (0-100): ', 0
inputFormat db '%d', 0
msg2 db 'You entered this grade: %d', 10, 0

section .bss
grade resd 1

section .text
global main
extern printf
extern scanf
```

```
main:
; Print the first message
        push msg1
        call printf
        add esp, 4

; Read the grade
        push grade
        push inputFormat
        call scanf
        add esp, 8

; Print the second message with the grade
        push dword [grade]
        push msg2
        call printf
        add esp, 8

; Exit
        mov eax, 1          ; sys_exit system call number
        xor ebx, ebx        ; Status 0
        int 0x80
```

```
$ nasm -f elf32 -o task2scanasm.o task2scanasm.asm
$ gcc -m32 -nostartfiles -no-pie -o task2scanasm task2scanasm.o
```

Hands-on Training Tasks:

1. Task 1: Hello World Program
2. Task 2: Reading User Input
3. Task 3: Basic Arithmetic Operations
4. Task 4: Using Conditional Statements
5. Task 5: Loop Constructs
6. Task 6: Function Calls
7. Task 7: Implementing a Counter
8. Task 8: Combining Loops and Conditionals
9. Task 9: Advanced Input Validation
10. Task 10: Complete Program Assembly

5) Tasks 3:10

...0x2BCON10U

By Students ...

Installing NASM:

If you select "Development Tools" while installing Linux, you may get NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps −

1. Open a Linux terminal.
2. Type **whereis nasm** and press ENTER.
3. If it is already installed, then a line like, ***nasm: /usr/bin/nasm*** appears. Otherwise, you will see just *nasm:*, then you need to install NASM.

Assembly - Environment Setup (tutorialspoint.com)
https://www.tutorialspoint.com/assembly_programming/assembly_environment_setup.htm

Installing NASM:

To install NASM, take the following steps –

- Check The netwide assembler (NASM) NASM website for the latest version.
- Download the Linux source archive nasm-X.XX.ta.gz, where X.XX is the NASM version number in the archive.
- Unpack the archive into a directory which creates a subdirectory nasm-X. XX.
- cd to nasm-X.XX and type ./configure. This shell script will find the best C compiler to use and set up Makefiles accordingly.
- Type make to build the nasm and ndisasm binaries.
- Type make install to install nasm and ndisasm in /usr/local/bin and to install the man pages.

Additional Reading (Optional):

- RE4B (Reverse Engineering 4 Beginners)
  - Covers Intel, ARM, MIPS assembler with concrete examples
  - Focus isn’t on malware, but still a great reference
- Intel architecture manuals https://software.intel.com/en-us/articles/intel-sdm
- http://ref.x86asm.net/
- http://x86asm.net/articles/x86-64-tour-of-intel-manuals/index.html
- http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
- https://godbolt.org/

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