# Intro to Assembly Language

## CISC 3310 — CUNY Brooklyn College
## Lecture Notes

# Intro to Assembly Language

In this chapter, we will:

1. Briefly introduce NASM, Netwide Assembly language.
2. List useful tutorials for learning NASM.
3. Learn how to use ChatGPT to assist writing NASM programs.
4. Bring a link to an online NASM compiler, and show how to use it.

# NASM

**NASM**, short for the **Netwide Assembler**, is an assembly language compiler designed for the `x86` architecture.

It was invented by Simon Tatham and Julian Hall in 1996.

NASM is an open-source assembler, meaning its source code is freely available for modification and distribution.

It supports two main versions: `x86-64`, which is designed for 64-bit systems, and a 32-bit version for compatibility with older systems.

The `x86-64` version allows programmers to take advantage of modern 64-bit processors and their extended registers.

The 32-bit version is ideal for developing software for legacy hardware or operating systems.

# NASM

One of the key strengths of NASM is its simplicity and ease of use compared to other assembly languages. NASM uses an intuitive syntax, which reduces the learning curve for new developers.

It is known for its high portability, allowing developers to use it across different operating systems and platforms.

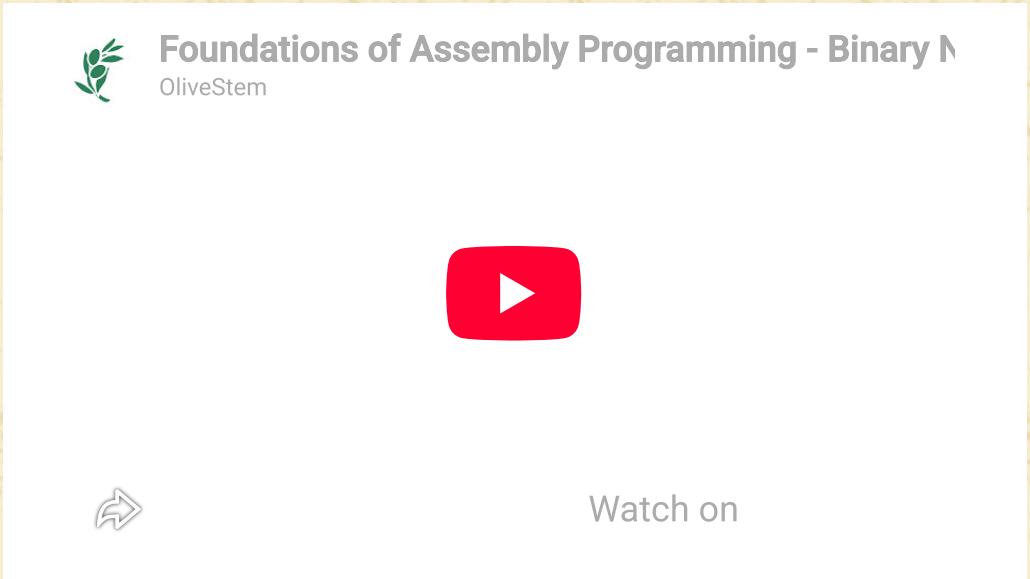
Unlike some other assemblers, NASM provides detailed error reporting, which helps programmers debug their code more effectively. It also supports macros (sort of functions), enabling developers to write more readable and reusable assembly code. Another advantage of NASM is its support for modular programming, allowing code to be organized into smaller, manageable pieces.

Overall, NASM's flexibility and robust feature set make it a popular choice for assembly language programming on x86 platforms.

# NASM: Tutorials

Following are several good tutorials and a YouTube playlist for learning how to code in NASM:

- NASM Tutorial by Professor Ray Toal
- https://www.tutorialspoint.com/assembly_programming/index.htm
- Linux Assembly Tutorial: Step-by-Step Guide by Derick Swanepoel




https://www.youtube.com/playlist?list=PL2EF13wm-hWCoj6tUBGUmrkJmH1972dBB

# NASM: Tutorials

Any questions?

# ChatGPT and NASM

As many of you are already aware, ChatGPT is capable of generating code in various languages according to the prompt you provide to it.

This feature can be used to write NASM code without knowing the NASM language thoroughly (which is true in the case of our course.)

A couple of the HW questions you are assigned this semester ask you to use ChatGPT to write NASM programs according to the provided descriptions.

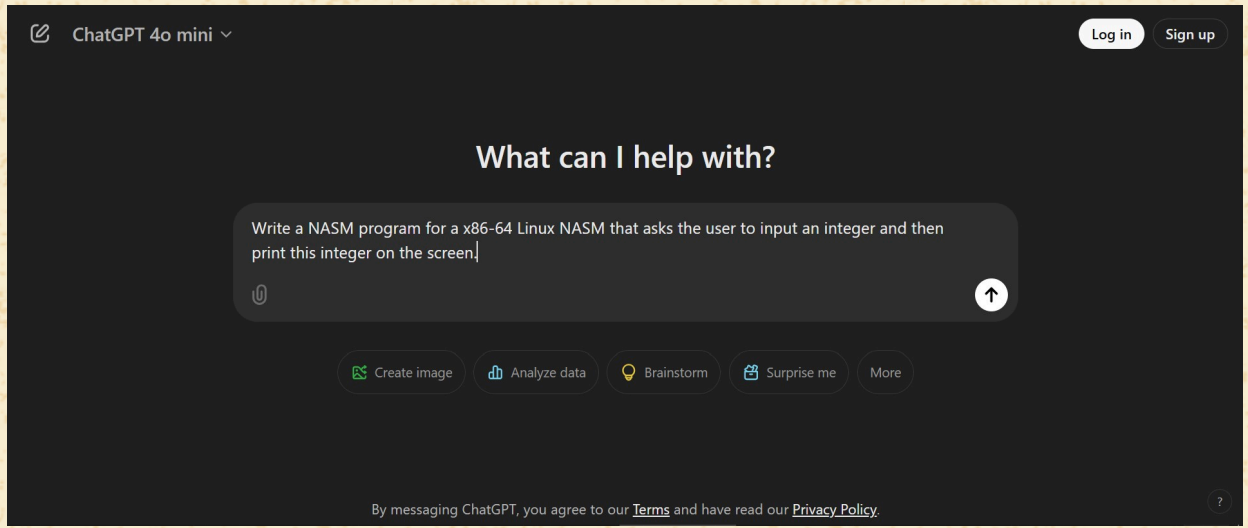
Here is an example of a program (different from the HW questions:) Write a NASM program for an `x86-64` Linux NASM that asks the user to type an integer. The program should then print this integer on the screen.

The prompt that one would enter to ChatGPT, therefore, could be: "Write a NASM program for an `x86-64` Linux NASM that asks the user to input an integer and then print this integer on the screen" or something similar.

# ChatGPT and NASM

A suggested ChatGPT server for this task is ChatGPT by OpenAI, which you can access at https://chatgpt.com/. It is recommended to sign up for a free account, but you can still use it without signing up or logging in.

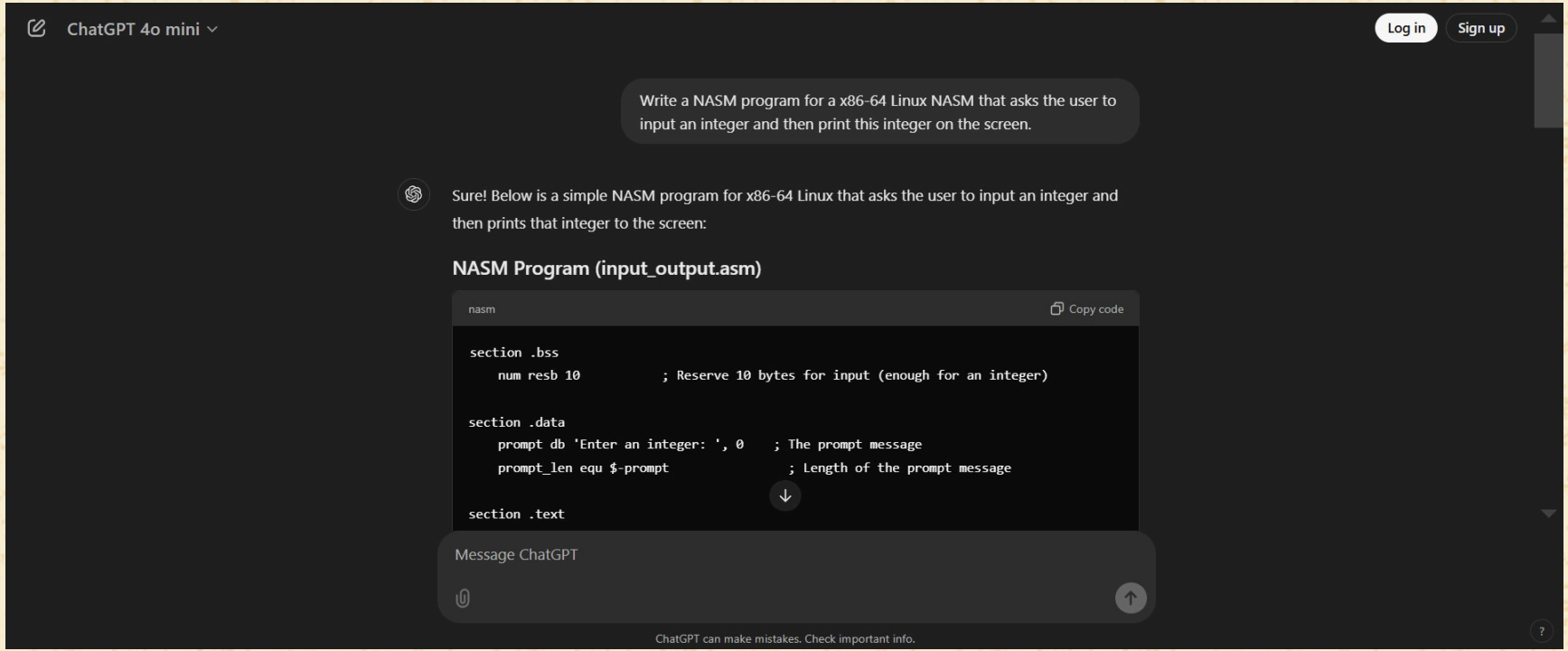
Here is a screenshot of what the ChatGPT window looks like after we typed the prompt from the previous slide into it:




Entering the prompt into ChatGPT. Miriam Briskman, CC BY-NC 4.0.

# ChatGPT and NASM

After pressing ENTER, ChatGPT will respond to you by writing the program's code (and perhaps explaining parts of it:)




ChatGPT replies to the prompt by writing code. Miriam Briskman, CC BY-NC 4.0.

# ChatGPT and NASM



```
section .bss
    num resb 10             ; Reserve 10 bytes for input (enough for an integer)

section .data
    prompt db 'Enter an integer: ', 0    ; The prompt message
    prompt_len equ $-prompt              ; Length of the prompt message

section .text
    global _start

_start:
    ; Print the prompt message
    mov rax, 1              ; Syscall number for sys_write
    mov rdi, 1              ; File descriptor (stdout)
```

# Compiling NASM

ChatGPT can write code, but it doesn't compile it: we need to use different software for compilation.

One way is to download and install a NASM compiler to your device. However, since we won't extensively program this semester in NASM, we will instead rely on an online compiler.

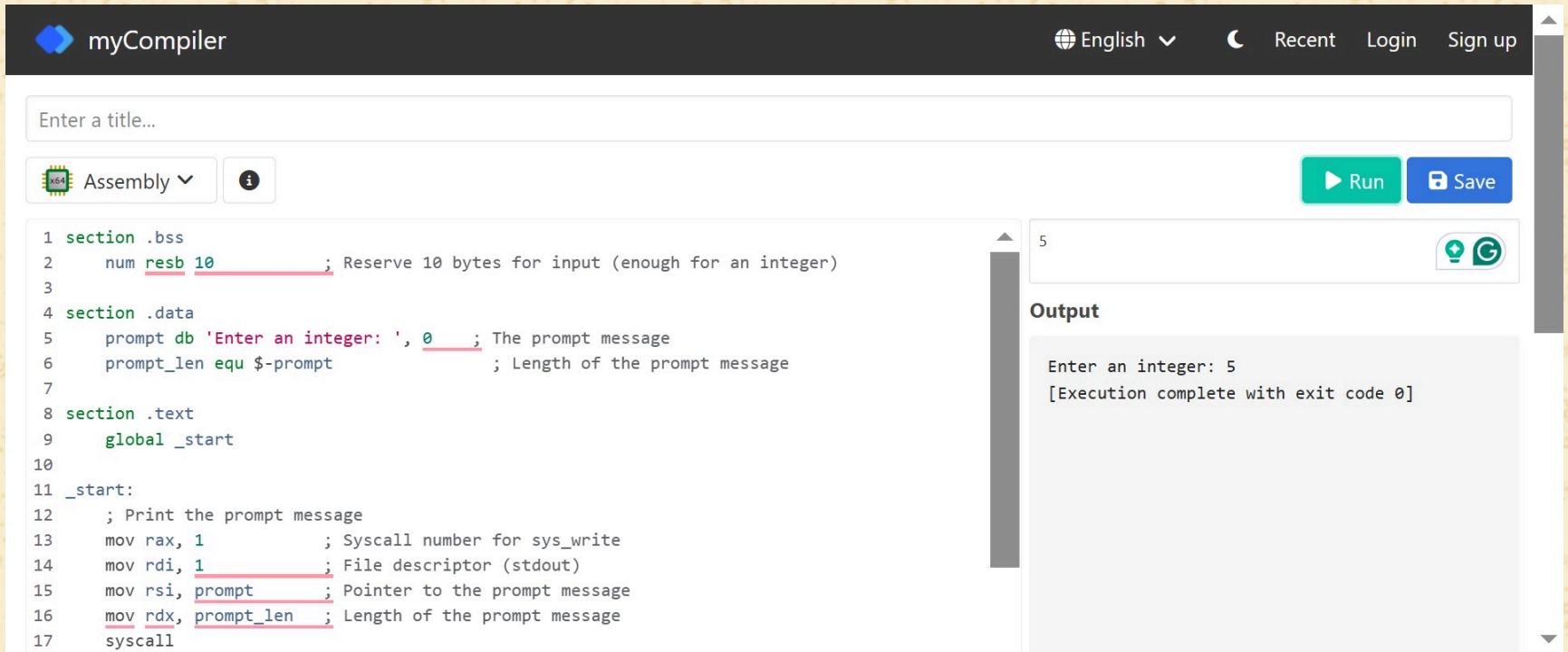
Specifically, we will use the NASM online compiler at **mycompiler.io**: https://www.mycompiler.io/new/asm-x86_64.

As we'll see in the following screenshots, this compiler's window contains two input boxes: one for the source code (on the left) and the other, smaller one for the input (top-right.) The output window (= console) at the bottom-right will display the results of running a program after you hit the Run button.

We will enter the code that ChatGPT produced for us into this compiler to assemble (= compile) and run the program we got.

# Compiling NASM

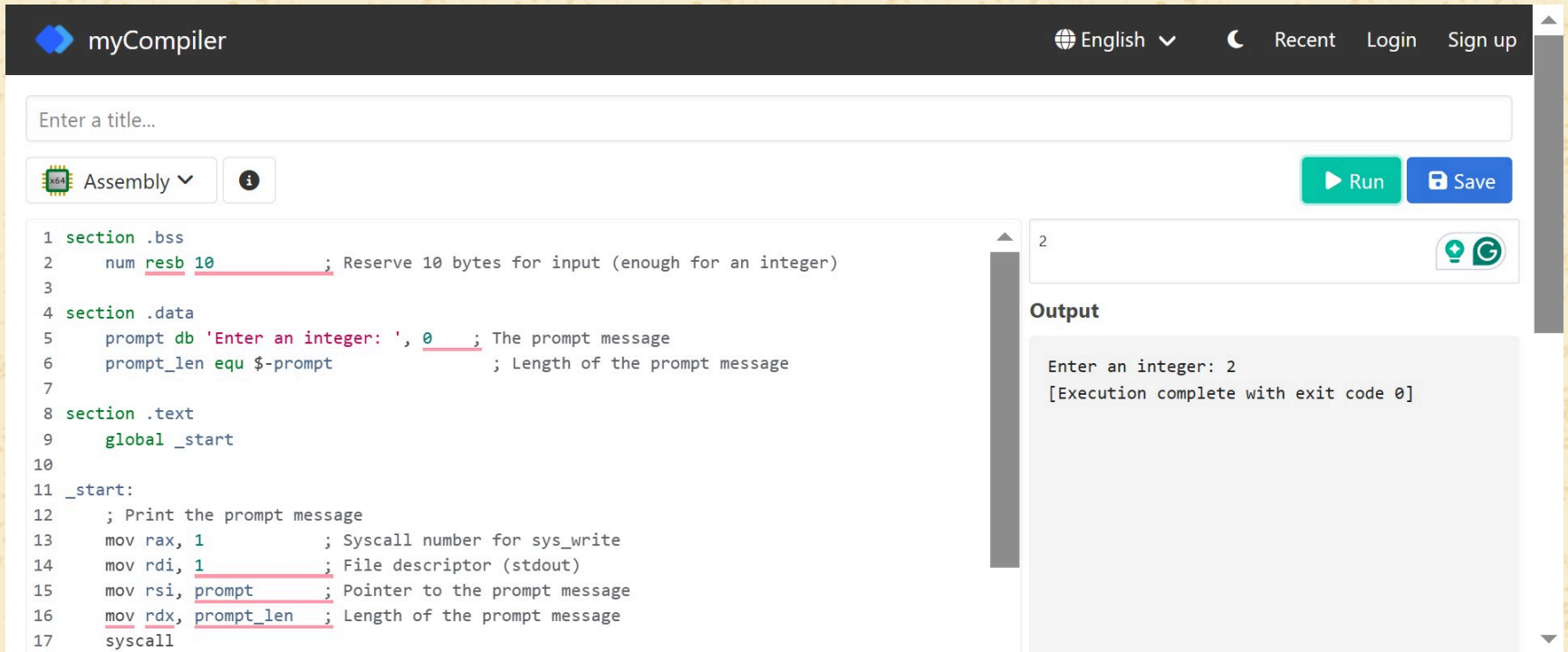
A screenshot of running the program with the input $5$:




Entering the code of the program into the compiler, and running it with the input 5. Miriam Briskman, CC BY-NC 4.0.

# Compiling NASM

Another screenshot of running the program, this time with the input $2$:




Entering the code of the program into the compiler, and running it with the input 2. Miriam Briskman, CC BY-NC 4.0.

# ChatGPT and NASM: Wrapping Up



In case any errors are printed while you run the program, this means that ChatGPT made some mistakes while running the code. To fix this, go back to the ChatGPT window, type what kind of an error you got, and ask ChatGPT to fix that error. After a couple of back-and-forth discussions with ChatGPT, it should generate an error-less code.

---

The final exam of CISC 3310 won't contain any questions on this chapter: the only tasks this semester that involve writing NASM code are those couple of homework questions that you are given at the end of the semester.

The goal of this chapter is to introduce you to how to use available online tools, including ChatGPT and compilers, to code efficiently.

Specifically, we showed that, even without deeply understanding a language, one can create useful code in a short amount of time and put it to use (by running it.)

# ChatGPT and NASM: Wrapping Up



Any questions?

# NASM Coding: Basics

The rest of these lecture notes will teach us some basic coding skills in NASM. Specifically, since the online NASM compiler that we introduced on slide 11 supports NASM `x86_64` on Linux, our slides will focus on code examples that run on a `x86_64` Linux device only.

In Linux `x86_64` NASM programs:

- We directly interact with CPU registers: an instruction will include the name of a register (or registers) and what data we want to put inside it (or read from it).
- We request the operating system to perform services for us, such as reading or writing from files, using the `syscall` instruction ("system call").
- We don't import any libraries and don't call any library functions. If needed, we write our repetitively-called code.

In short, a NASM program consists of (1) data, (2) instructions, and a defined entry point into the program (`_start`).

# NASM Coding: Basics



```
; A program printing "Hello world!\n" to the screen
; 10 is the ASCII code of a newline character

section .data
    msg db "Hello world!", 10

section .text
    global _start

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, 13
```

# NASM Coding: Basics

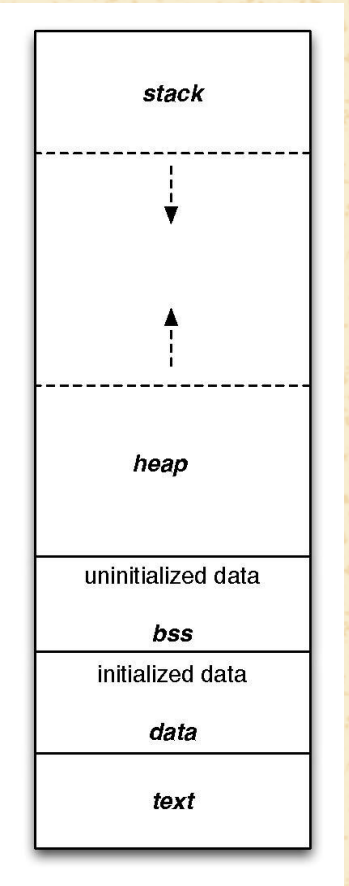


A NASM program is divided into sections:

```
section .data       ; initialized data
    ...
section .bss        ; uninitialized data
    ...
section .text       ; code
    global _start
    ...
```

- The **data** section contains variables that are initialized (= were assigned some initial values.)
- The **bss** (= **b**lock **s**tarted by **s**ymbol) section contains variables that *weren't yet* initialized; this is basically just memory reservation for later use.
- The **text** section contains the instructions of the program. The computer starts running the code at a label named `_start`.

# NASM Coding: Basics




Memory sections of a process. Taken from Dougct, CC BY-SA 3.0, via Wikipedia.

# NASM Coding: Basics - Examples



- `msg db "Hello", 0`

  In the code line above, we define a variable named `msg` whose content is the string literal `Hello` followed by the null character (`0`) whose purpose is to act as the string's endpoint. The directive (= pseudo-instruction) `db` tells the computer to store that data as raw 8-bit bytes in the **.data** section of the executable.

- `buffer resb 64`

  This line in the **.bss** section declares a variable with the name `buffer`, and tells the computer to reserve `64` bytes of space for it using the `resb` directive, whose purpose is to "Reserve Byte(s)".

- `mov rax, 1`

  Here, we copy the number `1` into the `rax` 64-bit register in the CPU using the `mov` instruction, which you will find in the **.text** section.

# NASM Coding: Basics

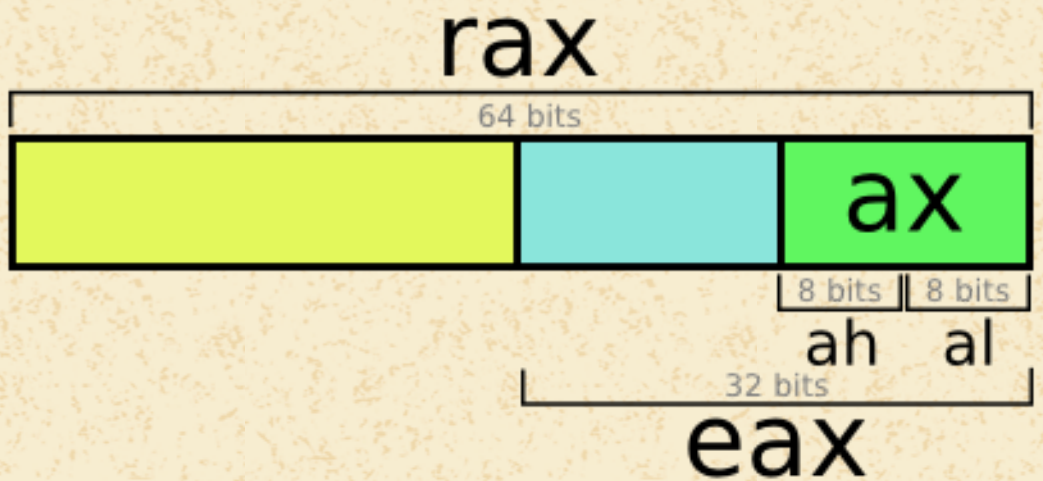
Here are the General-Purpose CPU registers that we'll work with and their descriptions:

| Register | Size | Typical Purpose (Convention) |
|---|---|---|
| `rax` | 64-bit | Accumulator; return value; syscall number |
| `rbx` | 64-bit | General-purpose |
| `rcx` | 64-bit | Counter (loops, shifts) |
| `rdx` | 64-bit | Data register; I/O length; syscall arguments |
| `rsi` | 64-bit | Source index; pointer to input data |
| `rdi` | 64-bit | Destination index; pointer to output data |
| `rbp` | 64-bit | Base pointer (stack frame reference) |
| `rsp` | 64-bit | Stack pointer (top of stack) |
| `r8` | 64-bit | General-purpose; syscall arguments |
| `r9` to `r15` | 64-bit | General-purpose |

# NASM Coding: Basics

We can also access smaller portions of each of the registers that we listed on the previous slide. For example, for the register `rax`, the following parts have names:

| Register | Size | What Part |
|---|---|---|
| `eax` | 32-bit | Lower (rightmost) 32 bits of `rax` |
| `ax` | 16-bit | Lower (rightmost) 16 bits of `rax` |
| `ah` | 8-bit | Higher (leftmost) 8 bits of `ax` |
| `al` | 8-bit | Lower (rightmost) 8 bits of `ax` |




Named parts of the `rax` register. Taken from Raw Linux Threads via System Calls, Public Domain.

# NASM Coding: Basics

Linux programs interact with the operating system and its resources (specifically, by asking the OS to do some service for the program), such as writing content to a file or checking the current time, using the `syscall` NASM instruction.

Before calling `syscall`, we first prepare the necessary data (service/syscall number, and arguments) for the call to run correctly: we put the syscall number inside the `rax` register, and the arguments inside the following registers: `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` (in this order).

This means that we can call any OS service (function/syscall) that needs up to $6$ arguments. Most OS syscalls, however, usually use much fewer than $6$ argument (the average number of arguments is $1$ to $3$ arguments.)

NASM Linux syscalls are easy to understand using code of the C language. On the next slides, we will see examples of Linux syscalls written in C, and their easy translation into NASM.

# NASM Coding: Basics - Examples

1. `ret = write (myfile, arr, 13);`

   This is a C syscall for writing the first `13` characters of string `arr` into a file (or device, like the screen) whose file ID number is in `myfile`. The `write` syscall returns the number of successfully written characters, or `-1` in case of an error.

   Since `write` takes only $3$ arguments, we'll use only the registers `rdi`, `rsi`, and `rdx` to put arguments in, and `rax` for the syscall number. After the syscall returns, `rax` will include the return value from `write`.

```
mov       rax, 1                ; syscall code 1 is for 'write'
mov       rdi, 1                ; file ID 1 is for stdout (= screen)
mov       rsi, arr              ; String variable 'arr' to output
mov       rdx, 13               ; number of bytes / characters to output
syscall                         ; invoke operating system to do the writing
```

# NASM Coding: Basics - Examples

More info on the `write()` syscall in C (this is not required for NASM programming: this is for your information only.)

<table>
<tr><td>Name: <code>write()</code></td><td colspan="2">System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: writes data from an array into a file, and returns the # of written bytes.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines write() and the size_t, ssize_t types.</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t write (int fd, const void *buf, size_t count);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of a file that was open for writing<br><code>buf</code>: The array name (or address of a variable, e.g., <code>&var</code>) from which data is taken.<br><code>count</code>: Number of bytes we wish to write into the file.</td></tr>
<tr><td rowspan="2">What type it returns: <code>ssize_t</code></td><td colspan="3">On success: A non-negative integer indicating how many bytes we wrote. This number will be equal to or smaller than <code>count</code>, and could also be <code>0</code> (in cases of some special files.)</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="4">Quick example: <code>ssize_t number_of_chars_written = write (fd, arr, strlen(arr));</code></td></tr>
</table>

# NASM Coding: Basics - Examples

2. `ret = read (myfile, arr, 64);`

This is a C syscall for reading up to `64` characters from a file (or device, like the keyboard) whose file ID number is in `myfile` into string `arr`. The `read` syscall returns the number of successfully read characters, or `-1` in case of an error.

As it was with `write`, the `read` syscall also takes only $3$ arguments and uses only the registers `rdi`, `rsi`, and `rdx` to put arguments in, and `rax` for the syscall number. After the syscall returns, `rax` will include the return value from `read`.

```
mov       rax, 0                ; syscall code 0 is for 'read'
mov       rdi, 0                ; file ID 0 is for stdin (= keyboard)
mov       rsi, arr              ; String variable 'arr' to read into
mov       rdx, 64               ; max number of bytes / characters to read
syscall                         ; invoke operating system to do the reading
```

# NASM Coding: Basics - Examples

More info on the `read()` syscall in C:

<table>
<tr><td>Name: <code>read()</code></td><td colspan="2">System call or function: System call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: reads data from a file into an array, and returns the # of read bytes.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines read() and the size_t, ssize_t types.</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t read (int fd, void *buf, size_t len); // ssize_t = signed size_t</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>fd</code>: file descriptor of an open file<br><code>buf</code>: The array name (or address of a variable, e.g., <code>&var</code>) into which data is read.<br><code>len</code>: Maximum number of bytes we can or wish read.</td></tr>
<tr><td rowspan="2">What type it returns: <code>ssize_t</code></td><td colspan="3">On success: A non-negative integer indicating how many bytes we read in. This number will be equal to or smaller than <code>len</code>, and could also be <code>0</code> (end-of-file case.)</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="4">Quick example: <code>ssize_t number_of_chars_read_in = read (fd, arr, 100);</code></td></tr>
</table>

# NASM Coding: Basics - Examples

3. `_exit(num);`

   This is a C syscall for terminating the program's execution. The argument `num` contains a number representing the exit status (also called exit code): `0` means "Success" (no issues happened during the execution), while any non-zero number represents some error. In particular, every non-zero exit code stands for a different type of error (examples: `1` is a generic error code; `11` is for segmentation fault; and `13` is for permission denied errors - e.g., trying to open a file that you don't own.)

   The `_exit` syscall takes only 1 argument, so it uses only the register `rdi` to put the argument in, and `rax` for the syscall number. Calling `_exit` is pretty similar to calling `return` inside the `main()` function in C.

```
mov       rax, 60                 ; syscall code 60 is for 'exit'
mov       rdi, 0                  ; exit code 0 is for 'Success'
syscall                           ; invoke operating system to exit the program
```

# NASM Coding: Basics - Examples

More info on the `_exit()` syscall in C:

<table>
<tr><td colspan="2">Name: <code>_exit()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: lets the kernel terminate the process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines _exit().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void _exit (int status);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>status</code>: the status with which we want to exit the program</td></tr>
<tr><td rowspan="2">What type it returns: <code>void</code></td><td colspan="3">On success: – This function doesn't return anything –</td></tr>
<tr><td colspan="3">On failure: – This function doesn't return anything –</td></tr>
<tr><td colspan="4">Quick example: <code>_exit (0); // Success!</code></td></tr>
</table>

# NASM Coding: Basics - Examples



"Success!" - Borat. Source: @primevideo, via GIPHY.

# NASM Coding: Basics

The **Linux kernel** (= the main component of the Linux OS) keeps a list of all the files currently open by every running program. This list is called the **file table**, and its indexes are **file descriptors** (`fd`s), each uniquely representing a file.

- Most file system calls accept an `fd` as an argument (since they need to know which file(s) we want to change.)
- Each file table entry contains a pointer to the file's metadata (inode), including the location of the file's content on disk.
- Legit `fd`s are non-negative integers (`0, 1, 2, ..., n`), so `fd == -1` indicates an error. Each program has an upper limit of how many files can be opened at the same time (by default: `n = 1000`), but we can change this limit if wanted.
- Conventionally, `fd == 0` represents **standard input** (= `stdin` = keyboard), `fd == 1` is **standard output** (= `stdout` = display), and `fd == 2` represents **standard error output** (= `stderr` = also printed to display). All 3 are open by default.
- `fd`s can also represent special files (devices, disks, sockets, etc.) since Linux treats everything as a file.

In the scope of our study of NASM coding, the only files that we will interact with are standard input (= `stdin` = keyboard) and standard output (= `stdout` = screen), even though NASM has the tools for reading from and writing to other files (e.g., text files stored on the computer.)

# NASM Coding: Basics

A quick summary of the files we will work with and common syscalls:

| File | ID | Description | Use |
|---|---|---|---|
| `stdin` | `0` | Keyboard | Input |
| `stdout` | `1` | Screen | Output |
| `stderr` | `2` | Screen (for printing error messages) | Output |

| Syscall | ID | # of Args | Description |
|---|---|---|---|
| `read` | `0` | 3 | Reading from files |
| `write` | `1` | 3 | Writing to files |
| `_exit` | `60` | 1 | Exiting the program |

The full list of all the supported Linux syscalls (there are slightly over 300 such functions) and their syscall numbers are listed in this article. In the scope of this class, however, we will use only the 3 syscalls we mentioned in the summary table above.

# NASM Coding: Common Instructions

Here are many of the instructions that we will used in our NASM programs, and their description:

| Instruction | Description | Operands |
|---|---|---|
| `add x, y` | `x += y;` | Numbers as registers or variables (`y` can also be a literal) |
| `and x, y` | `x &= y;` | Registers or variables (`y` can be a literal); they will be bitwise ANDed |
| `cmp x, y` | Is `x == y`? | Registers, variables, or literals; the OS sets the sign, zero, carry, and overflow flags |
| `dec x` | `x -= 1;` | Register or variable |
| `div x` | `rax:rdx /= x;` | Register, variable, or literal; unsigned integer division; quotient is put in `rax` and rem |
| `idiv x` | `rax:rdx /= x;` | Register, variable, or literal; signed integer division; quotient is put in `rax` and rema |
| `imul x` | `rax:rdx = x * rax;` | Register, variable, or literal; signed multiplication |
| `imul x, y` | `x *= y;` | `x` is a register and `y` is a register or variable; signed multiplication |
| `imul x, y, z` | `x = y * z;` | `x` is a register; `y` is a register or variable; `z` is a numeric literal (e.g., `5`); signed multi |

# NASM Coding: Common Instructions

| Instruction | Description | Operands |
|---|---|---|
| `inc x` | `x += 1;` | Register or variable |
| `jmp x` | Jump to other code | Name of a label that we set in our program |
| `je x` | Jump if equal | Name of a label that we set in our program |
| `jg x` | Jump if greater | Name of a label that we set in our program |
| `jge x` | Jump if greater or equal | Name of a label that we set in our program |
| `jl x` | Jump if less | Name of a label that we set in our program |
| `jle x` | Jump if less or equal | Name of a label that we set in our program |
| `jne x` | Jump if not equal | Name of a label that we set in our program |
| `jnz x` | Jump if not zero | Name of a label that we set in our program |
| `mov x, y` | `x = y;` | Registers or variables (y can be a literal) |
| `mul x` | `rax:rdx *= x;` | Register, variable, or literal; unsigned multiplication |
| `neg x` | `x = -x;` | Register or variable |

# NASM Coding: Common Instructions



| Instruction | Description | Operands |
|---|---|---|
| `not x` | `x = !x;` | Register or variable; bitwise NOT |
| `or x, y` | `x = x \| y;` | Registers or variables (`y` can be a literal); bitwise OR |
| `pop x` | `x = stack.pop();` | Register or variable; pop the top value from the stack |
| `push x` | `stack.push(y);` | Register, variable, or literal; push `y` to stack |
| `sub x, y` | `x -= y;` | Numbers as registers or variables (`y` can also be a literal) |
| `syscall` | Run a syscall | The arguments are in `rax`, `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` |
| `test x, y` | Compute `x && y` | Registers, variables, or literals; the OS sets the sign, zero, carry, and overflow flags |
| `xor x, y` | `x = x ^ y;` | Registers or variables (`y` can be a literal); bitwise XOR |

The full list of all the supported NASM instructions (in the `x86_64` architecture and others) can be found in Appendix F of the NASM manual.

# NASM Coding: Common Instructions



Any questions so far?

# NASM Coding: Basics - Program Examples



1.
```
; The only thing that this program does is terminating.
; This is the same as running the C program:
; int main()
; {
;     return 0;
; }

section .text
    global _start

_start:
    mov rax, 60
    mov rdi, 0
    syscall
```

# NASM Coding: Basics - Program Examples



2.
```
; A program that adds two constants and exits by returning the addition result.

section .text
    global _start

_start:
    mov rax, 5     ; Put 5 in rax
    mov rbx, 3     ; Put 3 in rbx
    add rax, rbx   ; Compute: rax += rbx

    mov rdi, rax   ; Put rax (= 3 + 5) in rdi
    mov rax, 60    ; Put the 'exit' syscall code in rax
    syscall        ; Call _exit(rdi), which is _exit(8)
```

# NASM Coding: Basics - Program Examples



3.
```
; The string "6-7" will be printed 5 times on the screen (using a loop):

section .data
    message db "6-7", 10    ; message = "6-7\n"
    msg_len equ $ - message ; msg_len = len(message)

section .text
    global _start

_start:
    mov rbx, 5          ; Put 5 in rbx

loop_start:             ; The loop_start label
    dec rbx             ; Decrement rbx -= 1
```

# NASM Coding: Basics - Program Examples

4.
```
; A program that takes an input (integer) from the user, and checks if it is even or odd.
; Output example:
; The number you entered is:
; 8
; and it is even.

section .data
    even_msg db "and it is even.", 10
    even_len equ $ - even_msg

    odd_msg db "and it is odd.", 10
    odd_len equ $ - odd_msg

    newline db 10
```

# NASM Coding: Basics - Program Examples



Any questions?

# NASM Coding: Volatile Registers

Throughout the execution of a NASM program, the contents of some registers will be 'wiped out' by the operating system.

Specifically, when we use the instruction `syscall`, the contents of the following registers will be erased (or replaced by 'garbage' values:) `rcx` and `r11`.

This means that we should avoid keeping values in a long-term fashion in these two registers.

Moreover, since the registers `rax`, `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` are used by us to store arguments to syscalls, we shouldn't keep important values in these registers in a long-term fashion, either.

***In which registers can we keep values long-term?***

Answer: the registers `rbx`, `r12`, `r13`, `r14`, and `r15` are guaranteed to never be modified by the operating system, so if we wish to store values for long duration throughout the execution of our NASM program, these $5$ registers are the right place.

# NASM Coding: Numeric Operand Formats



We can format our number literals in NASM as follows:

```
291           ; decimal
0291          ; decimal - the leading 0 does not make it octal
0291d         ; decimal - d suffix
0d291         ; decimal - 0d prefix
04bh          ; hex - h suffix, and the leading 0 is required
0x4b          ; hex - 0x prefix
0h4b          ; hex - 0h prefix
725q          ; octal - q suffix
0q725         ; octal - 0q prefix
10010111b     ; binary - b suffix
0b1001_0111   ; binary - 0b prefix (we can use underscores!)
```

# NASM Coding: Numeric Operand Formats



We can also use numbers to index into RAM as follows:

```
[750]                   ; offset
[rbp]                   ; base register
[rcx + rsi*4]           ; base + index * scale
[rbp + rdx]             ; scale is 1
[rbx - 8]               ; offset is -8
[rax + rdi*8 + 500]     ; all four components
[rbx + counter]         ; offset is the address of the variable 'counter'
```

Example:

```
mov rbx, [rbp] ; rbx = RAM[rbp]
```

# NASM Coding: Declaring Variables & Reserving Data



Declaring and setting variables (in the `.data` section:)

```
db    0x55                ; (db = 'define byte(s)') just the byte 0x55
db    0x55, 0x56, 0x57    ; three bytes in succession
db    'hello', 13, 10, '$' ; characters and string literals also work
dw    0x1234              ; ('define word(s)' = 2*n bytes) 0x34 0x12
dw    'a'                 ; 0x61 0x00 (character followed by null)
dw    'ab'                ; 0x61 0x62 (2 characters)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (string: 3 chars + null)
dd    0x12345678          ; ('define dword(s)' = 4*n bytes) 0x78 0x56 0x34 0x12
dq    0x123456789         ; ('define qword(s)' = 8*n bytes) 0x89 0x67 0x45 0x23 0x01 0x00 0x00
0x00
```

Example: `mystr db 'hello', ' ', 'world', 10, '$'`

# NASM Coding: Declaring Variables & Reserving Data



Reserving space for variables (in the `.bss` section:)

```
myarr resb 10  ; Reserve space for 10 bytes
var1 resw 2    ; Reserve space for 2 words (= 2 * 2 = 4 bytes)
large resd 5   ; Reserve space for 5 dwords (= 5 * 4 = 20 bytes)
wow resq 1     ; Reserve space for 1 qword (= 8 bytes)
```

NASM has greater data types (e.g., `reso` = 16 bytes, `resy` = 32 bytes, etc., but we won't need these for our course.) Also:

```
mov rbx, byte [loc]   ; Copies a single byte at RAM[loc] to rbx
mov rbx, word [loc]   ; Copies a single word (= 2 bytes) starting at RAM[loc] to rbx
mov rbx, dword [loc]  ; Copies a single dword (= 4 bytes) starting at RAM[loc] to rbx
mov rbx, qword [loc]  ; Copies a single qword (= 8 bytes) starting at RAM[loc] to rbx
```

# NASM Coding: Constants and String Length

We can declare constant values in several ways in NASM:

1. Using the `equ` pseudo-instruction:

```
name equ expression
```

Example:

```
msg db "Hello", 10
len equ $ - msg
```

**Explanation**: we set `msg` to be the string `"Hello\n"`. Then, we set `len` to be the size of the string `msg`, which is `6`.
**Fun fact**: the expression `$ - msg` is actually address subtraction: we subtract the start address of `msg` from the most recent address (which is stored at `$`), which happens to be the end address of `msg` after running the previous instruction. This results in the length of the string `msg`.

# NASM Coding: Constants and String Length



2. Using the `%define` directive:

```
%define name expression
```

Example:

```
%define WRITE_SYSCALL_NUM 1
%define PRETTY_NUM 12345
%define EXIT_SYSCALL mov rax, 60 \ syscall ; We can put entire code segments in the macro!!!
%define MYREG rax                          ; We can even create register aliases!
```

**Explanation**: we can turn any piece of NASM code into a constant. This is similar to creating a macro, whose name you just need to write somewhere in your program: you can write (repeat) it as many times as wanted! During compilation, a special software called the **pre-processor** will replace every instance of your constant name with the actual code behind it before converting the program into machine code (e.g., every `WRITE_SYSCALL_NUM` will be replaced with a `1`.)

# NASM Coding: Constants and String Length



3. Using literals in your code. Example:

```
mov rax, 10
mov rbx, 0xA
mov rcx, 'A'
```

**Explanation**: This is probably the most straightforward way to work with constants, as any number or character literal is considered a constant: just write the literal(s) directly in the code (= `.text` section) inside the instructions you call.

Quick summary for NASM x86_64 constants:

1. `equ` defines fixed values, such as a string's length (in bytes/chars).
2. `%define` can be used as a macro: code substitution.
3. Number and character literals like `0ah` or `'z'` can be directly embedded in the code.

# NASM Coding: Constants and String Length



Any questions?

# NASM Coding: Trace Tables

One way to keep track of what is happening with a program that we are working with is by creating a **trace table** for it: a table what keep track of the input, output, register, variable, and exit code values as the program keeps executing.

- The table will feature one row per each instruction that we run (if we have a loop, we might run the same instruction more than once, so there will be multiple rows per that instruction.)
- You may create this table using any convenient app, e.g., Microsoft Excel or Google Sheets.

Here is the trace table for the short program on slide 37:

| Instruction | stdin | rax | rdi | stdout | Exit Code | Notes |
|---|---|---|---|---|---|---|
| `mov rax, 60` | | `60d` | | | | |
| `mov rdi, 0` | | `60d` | `0` | | | |
| `syscall` | | `60d` | `0` | | `0` | `exit(0);` |

# NASM Coding: Trace Tables

Here is the trace table for the program on slide 38:

| Instruction | stdin | rax | rbx | rdi | stdout | Exit Code | Notes |
|---|---|---|---|---|---|---|---|
| `mov rax, 5` | | 5d | | | | | |
| `mov rbx, 3` | | 5d | 3d | | | | |
| `add rax, rbx` | | 8d | 3d | | | | |
| `mov rdi, rax` | | 8d | 3d | 8d | | | |
| `mov rax, 60` | | 60d | 3d | 8d | | | |
| `syscall` | | 60d | 3d | 8d | | 8 | `exit(8);` |

# NASM Coding: Trace Tables

Here is the trace table for the program on slide 39:

| Instruction | stdin | message | msg_len | rax | rbx | rdi | rsi | rdx | (rbx != 0) | stdout | Ex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `message db "6-7", 10` | | `"6-7\n"` | | | | | | | | | |
| `msg_len equ $ - message` | | `"6-7\n"` | `4d` | | | | | | | | |
| `mov rbx, 5` | | `"6-7\n"` | `4d` | | `5d` | | | | | | |
| `dec rbx` | | `"6-7\n"` | `4d` | | `4d` | | | | | | |
| `mov rax, 1` | | `"6-7\n"` | `4d` | `1d` | `4d` | | | | | | |
| `mov rdi, 1` | | `"6-7\n"` | `4d` | `1d` | `4d` | `1d` | | | | | |
| `mov rsi, message` | | `"6-7\n"` | `4d` | `1d` | `4d` | `1d` | `"6-7\n"` | | | | |
| `mov rdx, msg_len` | | `"6-7\n"` | `4d` | `1d` | `4d` | `1d` | `"6-7\n"` | `4d` | | | |
| `syscall` | | `"6-7\n"` | `4d` | `4d` | `4d` | `1d` | `"6-7\n"` | `4d` | | `"6-7\n"` | |
| `cmp rbx, 0` | | `"6-7\n"` | `4d` | `4d` | `4d` | `1d` | `"6-7\n"` | `4d` | True | | |

# NASM Coding: Trace Tables

***Fun Class Activity***:

1. Create trace tables for the programs on:
    1. Slide 10
    2. Slide 17
    3. Slide 40
    4. Slide 56
    5. Slide 57

# NASM Coding: Trace Tables



Any questions?

# NASM Coding: More Programs

5.
```
; A program that takes an input (integer) from the user, and prints its square to stdout.
; Example: If the user entered 3 as the input, the output will be:
; 9

section .data
    newline db 10

section .bss
    buffer resb 32  ; input buffer
    outbuf resb 32  ; output buffer

section .text
    global _start
```

# NASM Coding: More Programs

6.
```
; A program that prints a simple pyramid of stars ('*') to stdout.

section .bss
    maxlines  equ 8
    dataSize  equ 44
    output:   resb dataSize

section .text
    global _start

_start:
    mov rdx, output             ; rdx = pointer to buffer
    mov r8, 1                   ; current line length
    xor r9, r9                  ; stars written in current line
```

# NASM Coding: More Programs

Any questions?

# Intro to Assembly Language: Sources


"Create a New Assembly Program - Mycompiler." *myCompiler*, www.mycompiler.io/new/asm-x86_64.

Puiu, Rares Gabriel, et al. *A Gentle Introduction to the Central Processing Unit and Assembly Language*, 2021, *GitHub*. License: CC BY-NC-SA: Attribution-NonCommercial-ShareAlike.

Tatham, Simon, et al. *NASM - The Netwide Assembler*. 2.16.03 version, 2024, *nasm.us*. License: The 2-Clause BSD License.