Malicious Code Analysis

Fangtian Zhong
CSCI 591

Gianforte School of Computing

Norm Asbjornson College of Engineering

E-mail: fangtian.zhong@montana.edu

Overview

1 **Registers**

2 **Basic Syntax**

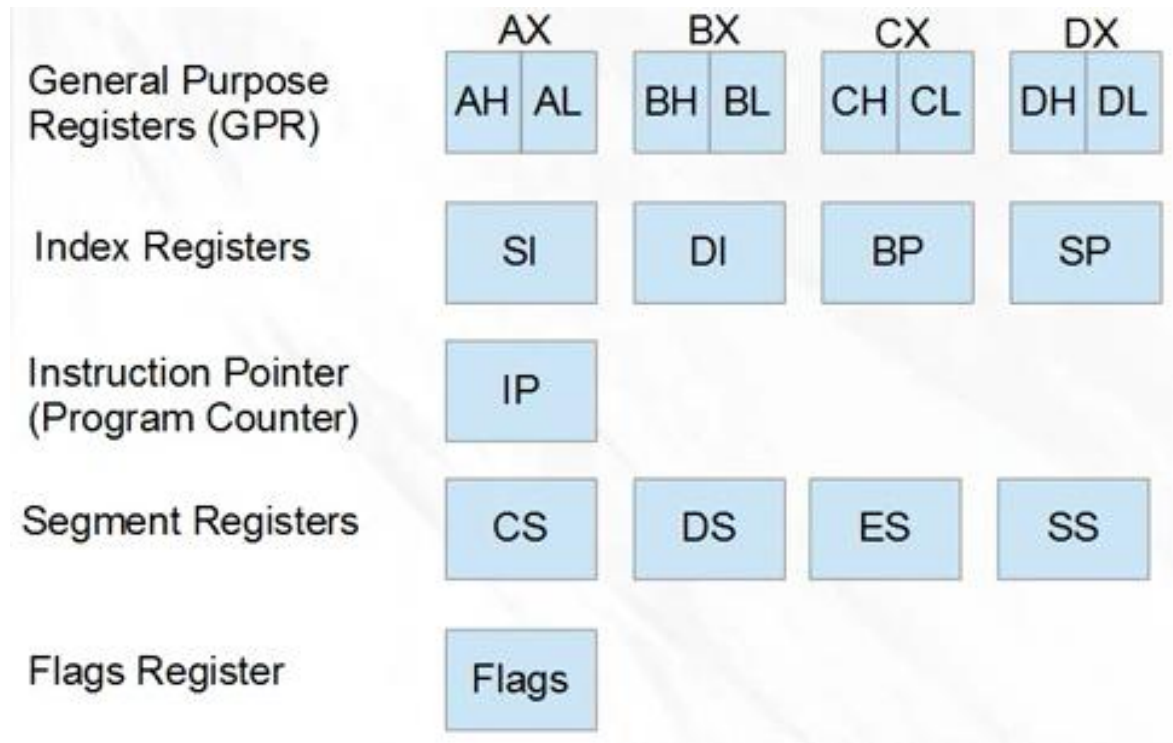3 **Calling Conv.**

Part Two

02

Registers

8086 Registers

TThe 8086 hit the world in 1978 and was incredibly popular. It was a 16 bit processor, which meas most of its general purpose registers are 16 bits, and most of its instructions operate on 16 bits.

|	AX	BX	CX	DX
General Purpose Registers (GPR)	AH AL	BH BL	CH CL	DH DL
Index Registers	SI	DI	BP	SP
Instruction Pointer (Program Counter)	IP			
Segment Registers	CS	DS	ES	SS
Flags Register	Flags			

32 bit 80386:1

🛡 In 1985, the 386 processors came out, they included 32 bit instructions and registers.

🛡 For backwards compatibility, all of the 8086's original 16 bit registers were maintained. But most registers also got a 32 bit version;
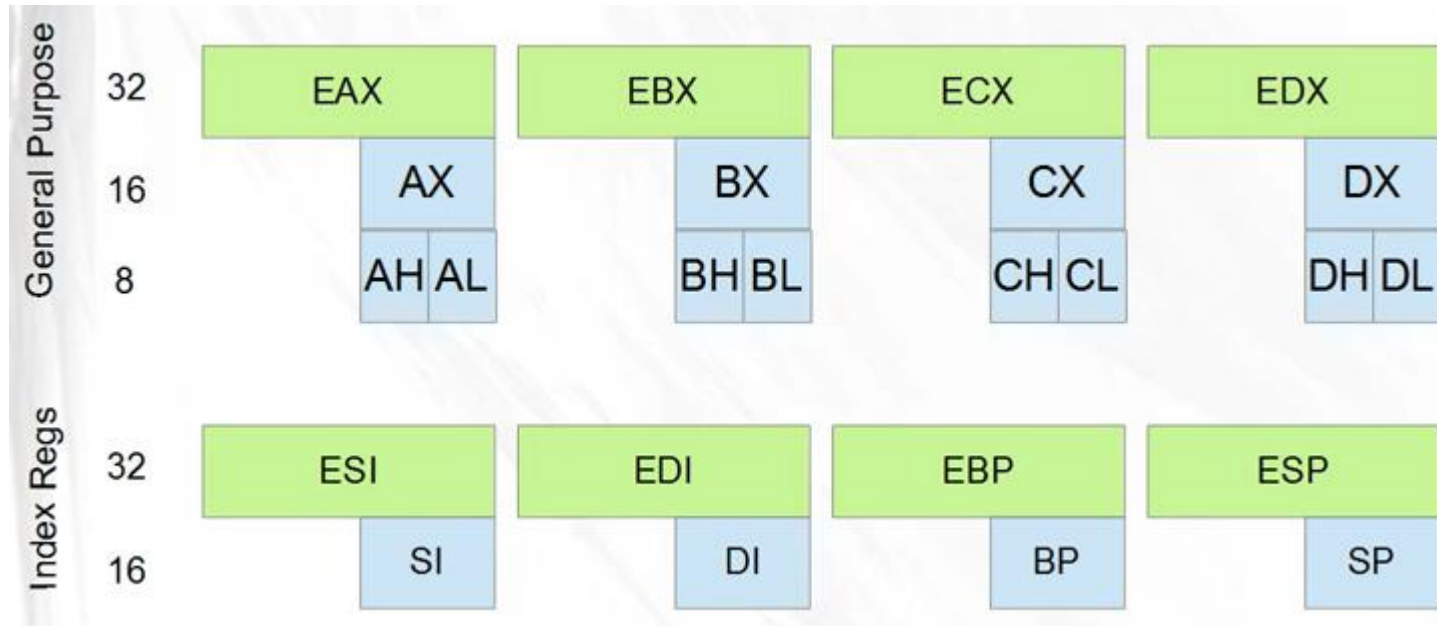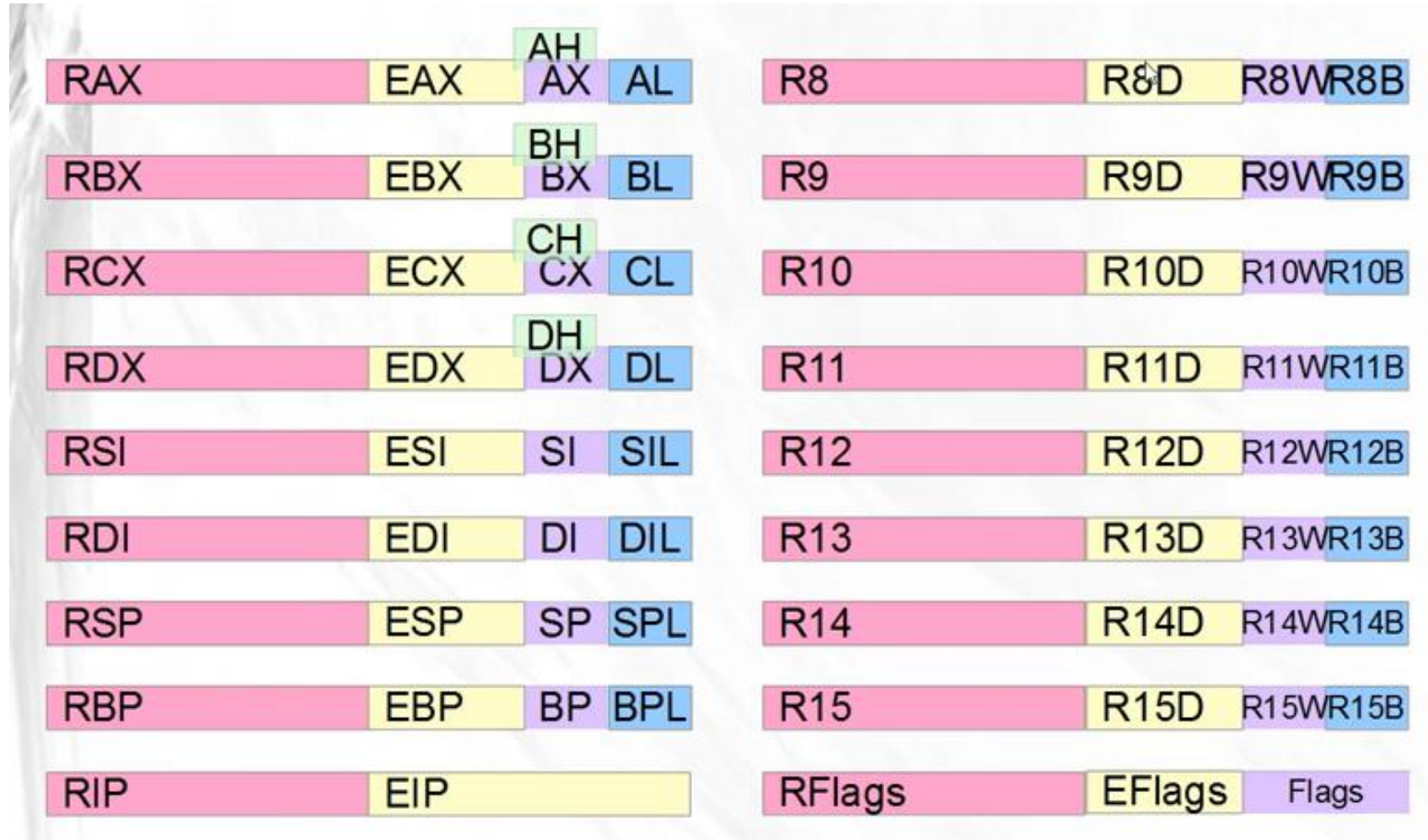
32 bit 80386:1

Pentium IV and x64

Pentium IV and x64

- Streaming SIMD Extensions (SSE)
- Advanced Vector Extensions (AVX)
- AVX-512
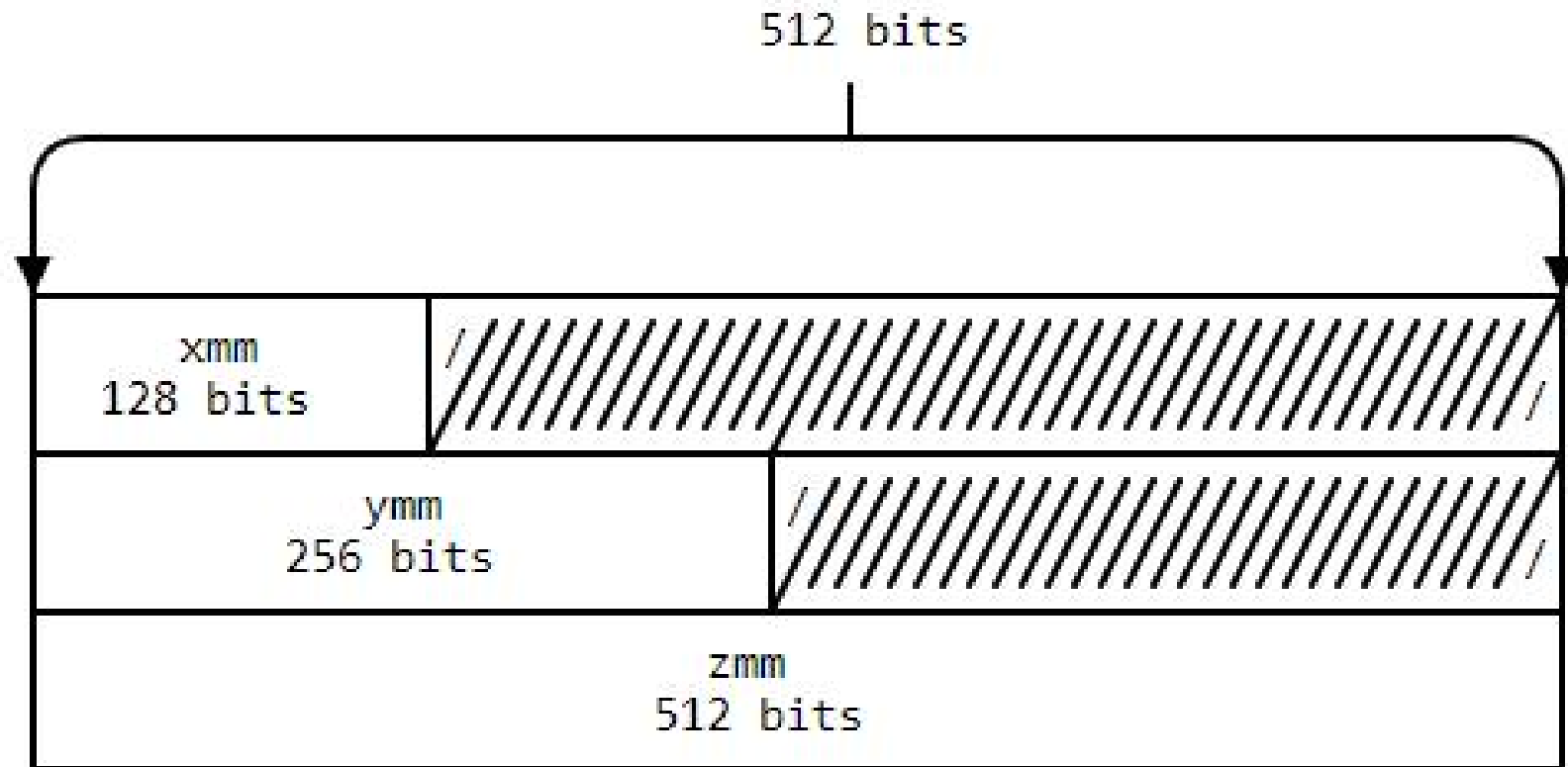
02

Basic Syntax

The data Section

🏅 The data section is used for declaring initialized data or constants.

🏅 You can declare various constant values, file names, or buffer size, etc., in this section.

🏅 This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

🏅 The syntax for declaring data section is −

- **section .data**

The bss Section

🏅 A dynamic memory section that contains buffers for data to be declared later in the program.

🏅 zero-filled.

🏅 The syntax –

- **section .bss**

The text section

🏅 A section for keeping the actual code.

🏅 Begin with the declaration "global", which tells the kernel where the program execution begins.

🏅 The syntax −

section .text

   global main

main:

Constants

🏅 We will particularly discuss three directives –

- equ
- %assign
- %define

Constants Examples: equ

```
bits 64
default rel

section .data
    myConstant equ 42
    message db "The constant value is: ", 0xd, 0xa, 0; byte
    format db "%d", 0xd, 0xa, 0; printing format for numbers
    dwchars dw "ninechars", 0xd, 0xa, 0; double word

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
;Print the string
lea rcx, [message]
call printf

; Print the constant value
mov rcx, format
mov rdx, myConstant
call printf

;Print the word chars
lea rcx, [dwchars]
call printf

; Exit the program
xor rax, rax
call ExitProcess
```

Constants Examples: %assign

```asm
bits 64
default rel

section .data
    %assign myConstant 42
    message db "The constant value is: ", 0xd, 0xa, 0; byte
    format db "%d", 0xd, 0xa, 0; printing format for numbers

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```asm
;Print the string
lea rcx, [message]
call printf

; Print the constant value
mov rcx, format
mov rdx, myConstant
call printf

;Print the word chars
lea rcx, [dwchars]
call printf

; Exit the program
xor rax, rax
call ExitProcess
```

Constants Examples: %define

```
bits 64
default rel

section .data
    %define myConstant 42
    message db "The constant value is: ", 0xd, 0xa, 0; byte
    format db "%d", 0xd, 0xa, 0; printing format for numbers

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
;Print the string
lea rcx, [message]
call printf

; Print the constant value
mov rcx, format
mov rdx, myConstant
call printf

; Exit the program
xor rax, rax
call ExitProcess
```

Variables

🏅 The syntax for storage allocation statement for initialized data is −

- [variable-name] define-directive initial-value [, initial-value]

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Allocating Storage Space for initialized Data

choice	db	'y'	
number	dw	12345	
neg_number	dw	-12345	
big_number	dq	123456789	; eight byte constant
real_number1	dd	1.234	; floating-point constant
real_number2	dq	123.456	; double-precision float
real_number_3	dt	1.234567e20	; extended-precision float

Examples

```
bits 64
default rel

section .data
    neg_number dw -12345
    message db "The constant value is: ", 0xd, 0xa, 0; byte
    format db "%hd", 0xd, 0xa, 0; printing format for numbers

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
;Print the string
lea rcx, [message]
call printf

; Print the constant value
mov rcx, format
mov edx, dword [neg_number]
call printf


; Exit the program
xor rax, rax
call ExitProcess
```

Allocating Storage Space for Uninitialized Data

🎖 The reserve directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved.

🎖 There are five basic forms of the reserve directive –

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Examples

```
bits 64
default rel

section .data
    message db "Hello, World!\n", 0
    format db "%d", 0xd, 0xa, 0; printing format for numbers

section .bss
    uninitializedData resb 4

section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
;Print the string
lea rcx, [message]
call printf

; Print the constant value
mov byte [uninitializedData], 42
mov rcx, format
mov dl, byte [uninitializedData]
call printf


; Exit the program
xor rax, rax
call ExitProcess
```

Multiple Definitions and Initializations

🏅 You can have multiple data definition statements in a program. For example −

choice	dd	'Y'	; ASCII of Y = 79H
number1	dw	12345	; 12345D = 3039H
number2	dd	12345679	; 123456789D = 75BCD15H

🏅 The assembler allocates contiguous memory for multiple variable definitions.

Multiple Definitions and Initializations

🏅 The TIMES directive allows multiple initializations to the same value.

🏅 For example, an array named marks of size 9 can be defined and initialized to zero using the following statement −

- **marks TIMES 9 DW 0**

Multiple Definitions and Initializations

🎗 The TIMES directive is useful in defining arrays and tables. The following program displays 9 asterisks on the screen −

```
bits 64
default rel

section .data
    stars   times 9 db '*'


section .text
    extern printf
    global main
    extern ExitProcess
    extern _CRT_INIT
```

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ;Print the string
    lea rcx, [stars]
    call printf


    ; Exit the program
    xor rax, rax
    call ExitProcess
```

Efficiency

🏅 To tell NASM to compile our program with rip-relative addressing, the following directive is used:

- **default rel**

🏅 Thus, we get what we want; the loader does less work, and the program still runs, and we get our position-independent code.

Addressing Modes

🏅 Addressing modes, put simply, are the different conventions available by which an assembly instruction can access registers or other memory. For example, the instruction:

- **mov   rax, 0**

🏅 Is what's known as an immediate addressing mode. This is because the operand has a constant value.

Addressing Modes

🛡️ In contrast, the instruction:

- **mov   rax, rbx**

🛡️ Is simply known as register addressing, for obvious reasons.

🛡️ We can also do what is known as indirect register addressing:

- **mov   rax, [rbx]**

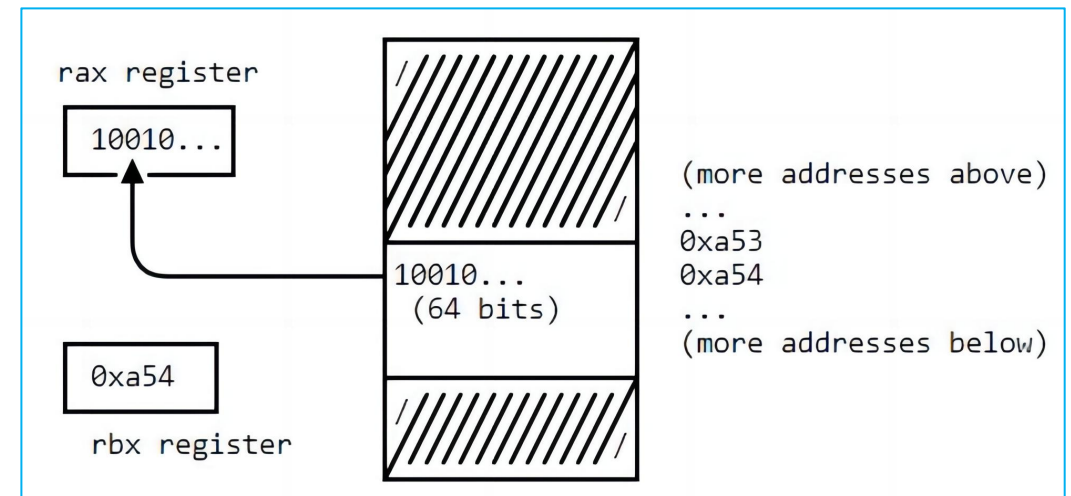**Which basically results in the following operation.**

Figure: Illustration of what happens after an indirect address operation.
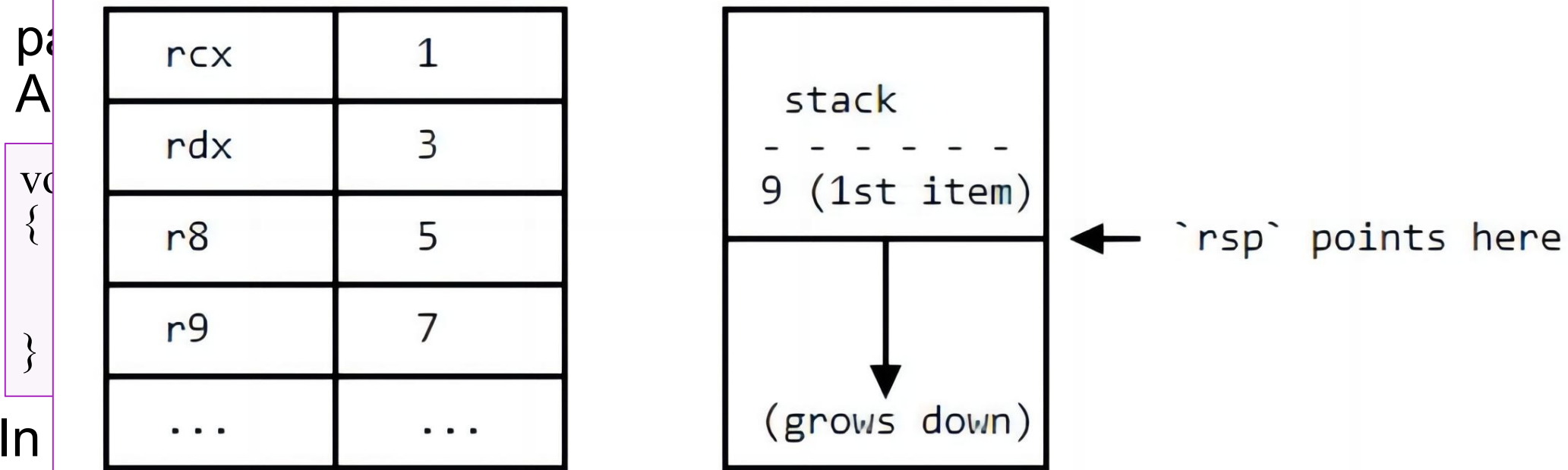
Part Three

03

Calling Convention

Calling Convention

★ Simply put, it's a set of strict guidelines that our code must adhere to in order for the operating system to be able to run our assembly code.

The Microsoft x64 Calling Convention

★ Function parameters and return values

★ The first four integer arguments are passed in registers. Integer values are pa... ...es are
A...

v...
{

}

rcx	1
rdx	3
r8	5
r9	7
...	...

```
stack
- - - - - -
9 (1st item)
                        ← `rsp` points here


(grows down)
```

★ In ... ...a, b, c and d in the registers rcx, rdx, r8 and r9 respectively, with e being pushed onto the stack before calling the function foo(). The ... if I wanted to call foo() like so:

*Figure: How the memory should be laid out before calling **foo()**.*

- foo(1, 3, 5, 7, 9);
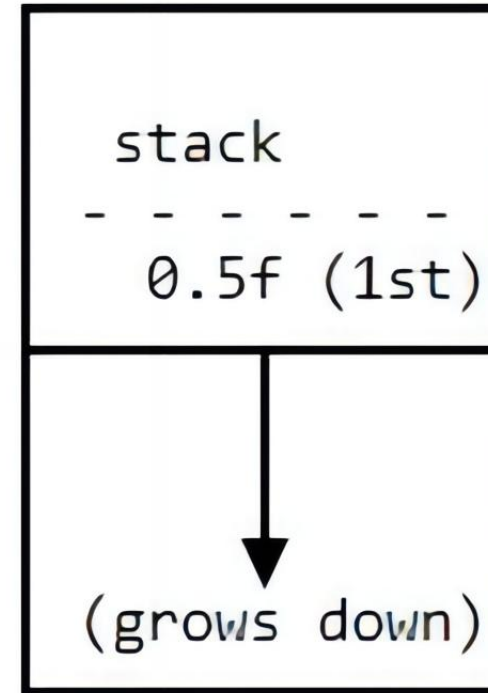
Floating-point arguments

★ Any floating-point and double-precision arguments in the first four parameters are passed in XMM0 - XMM3, depending on position.

★ For floati   3 are used,
for a tota   th the rest
being pu

★ Let's see

xmm0	0.1f
xmm1	0.2f
xmm2	0.3f
xmm3	0.4f
...	...

```
void foo_fp
{
    // Do son
    return;
}
```

```
stack
- - - - - -
0.5f (1st)

(grows down)
```

★ *Figure: How the memory should be laid out before calling **foo_fp()**.*

• foo_fp(0.1f, 0.2f, 0.3f, 0.4f, 0.5f);

Questions?

★ As we can see, the concept applies much like it for integers. But what if we have something like this?

```
void foo_mixed(int a, int b, float c, int d, float e)
{
    // Variable types of arguments...now what?
    return;
}
```
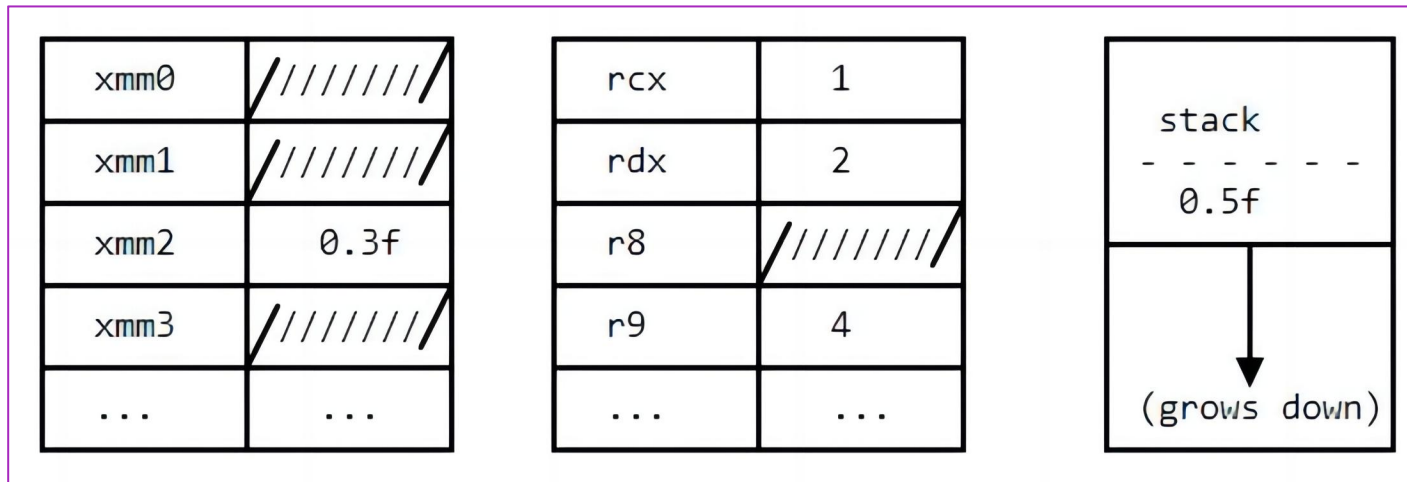
Which values go in which registers now?

Mixing parameter types

★ The answer is that the position of the argument dictates which register it goes in. Therefore, if we called foo_mixed() like so:

★ foo_mixed(1, 2, 0.3f, 4, 0.5f);

Figure: How the memory should be laid out before calling **foo_mixed().**

Return Values

- The Microsoft x64 calling convention, thankfully, has simpler rules when it comes to return values from functions.
  - Any scalar return value 64 bits or less in size is returned in rax.
  - Any floating-point value is returned in xmm0.
- Thus, a function int foo() would return its value in the rax register, and a function float foo() or double foo() would return its value in the xmm0 register.

The Microsoft C Runtime Library

★ What's the odd extern _CRT_INIT symbol we're importing? What's that for?

★ If you're experienced enough with C/C++, you'll no doubt have guess that the CRT refers to the C standard run-time library.

★ _CRT_INIT: _CRT_INIT is a CRT initialization function that sets up various aspects of the C Runtime Library, including initializing global variables, setting up memory allocation, and performing other necessary setup tasks.

WinMain and main

★ let's look further down the code, to the first label in the .text section of our assembly program:

main:

All C/C++ programmers should be familiar by now with the concept of their program's entry point, that is, the initial point where code starts to execute when their program is loaded. The signature usually appears as
       int main(int argc, char *argv[]).

★ However, if you're familiar with Win32 API programming, you'll know that things are not that simple. Technically, the entry point for Windows programs is defined as WinMain, not main; specifying the subsystem to the MSVC linker determines which entry point symbol is chosen by default. Additionally, the MSVCRT's implementation of main actually calls WinMain, which means that it's essentially a wrapper function. More than being just a simple wrapper, however, there's one other important thing that MSVCRT's main function does, and that is to also perform any static initialization of variables required.

Making a stack

★ Let's continue looking at the example code.

```
push    rbp
mov     rbp, rsp
sub     rsp, 32
```

★ The push psuedo-op takes the operand passed to it, decrements the stack pointer, and then stores the operand on top of the stack. We do this to the base pointer so that we can save the current position of the stack. (So that if we need to refer to variables on the stack, we have a base address to refer to, since we could be adding/removing objects from the stack all the time and thus the stack pointer alone would be insufficient.)

Calling functions in assembly

★ **Instruction Pointer (IP):** The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.

★ The next few lines are the real business logic of the example assembly code:

```
lea     rcx, [msg]
call    printf
```

★ The first instruction here is a little confusing.

★ LEA stands for "Load Effective Address," but that name doesn't clearly explain what it does. I like to think of LEA as being similar to MOV — both put a value into the first operand. The key difference is how they get that value. MOV transfers actual data from memory or a register, while LEA calculates the address from the second operand and puts that address into the first operand.

Shutting down the program

★ Now that we've printed our line out to the console, we're good. It's time to shut it down!

```
xor     rax, rax
call    ExitProcess
```

★ Remember that according to the calling convention, the return value for a function goes into rax for integers. Well, main is no different, and so we exclusive-or the rax register with itself, effectively zero-ing it out, before calling the Win32 ExitProcess function, thus ending the application.

39

★ You might be asking, "what on earth am I typing here?" at this point. We'll go over this later. For now, though, let's just make sure that your toolchain is working.

- Let's go ahead and try to assemble this text into an object file. Go to the directory where hello_world.asm is located at and run the following command in a command prompt.
  - nasm -f win64 -o hello_world.obj hello_world.asm

- If it worked, you should see the hello_world.obj file appear in the same directory as your hello_world.asm file. We can then use the linker to create an executable out of this object file.

- Run the following command from a command prompt that has the Visual Studio environment variables set (x64 Native Tools Command Prompt for VS 2022)
  - link hello_world.obj /subsystem:console /out:hello_world_basic.exe kernel32.lib legacy_stdio_definitions.lib msvcrt.lib

40

Run

★ You should now have a hello_world.exe file in the directory. Run it, and if you get the following output, congratulations, you've just wrote an assembly program that runs on Windows!

- hello_world_basic.exe
- Hello world!

THE END

Fangtian Zhong

CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

8/26/2025