Georgia Tech

INTRO. TO ASSEMBLY LANGUAGE

PROF. BRENDAN SALTAFORMAGGIO

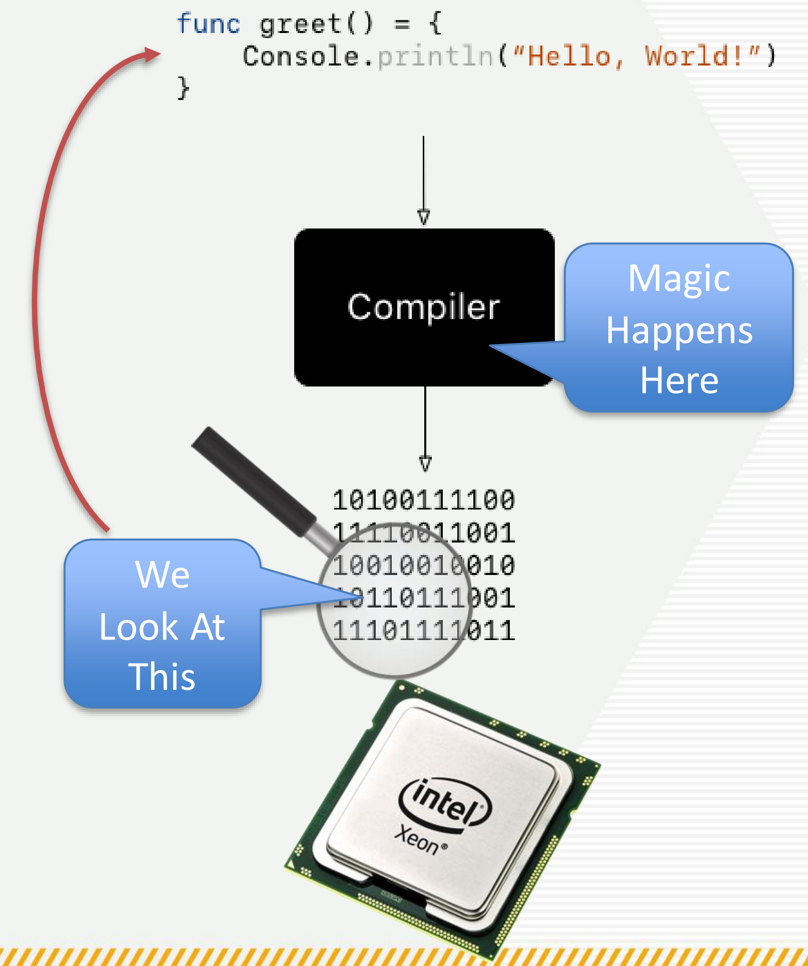
SCHOOL OF ECE

CREATING THE NEXT®

PLEASE CONSIDER THE ENVIRONMENT, AVOID PRINTING SLIDES!

MANY THANKS TO GOLDEN RICHARD AND MATHIAS PAYER FOR THEIR CONTRIBUTIONS TO THESE SLIDES

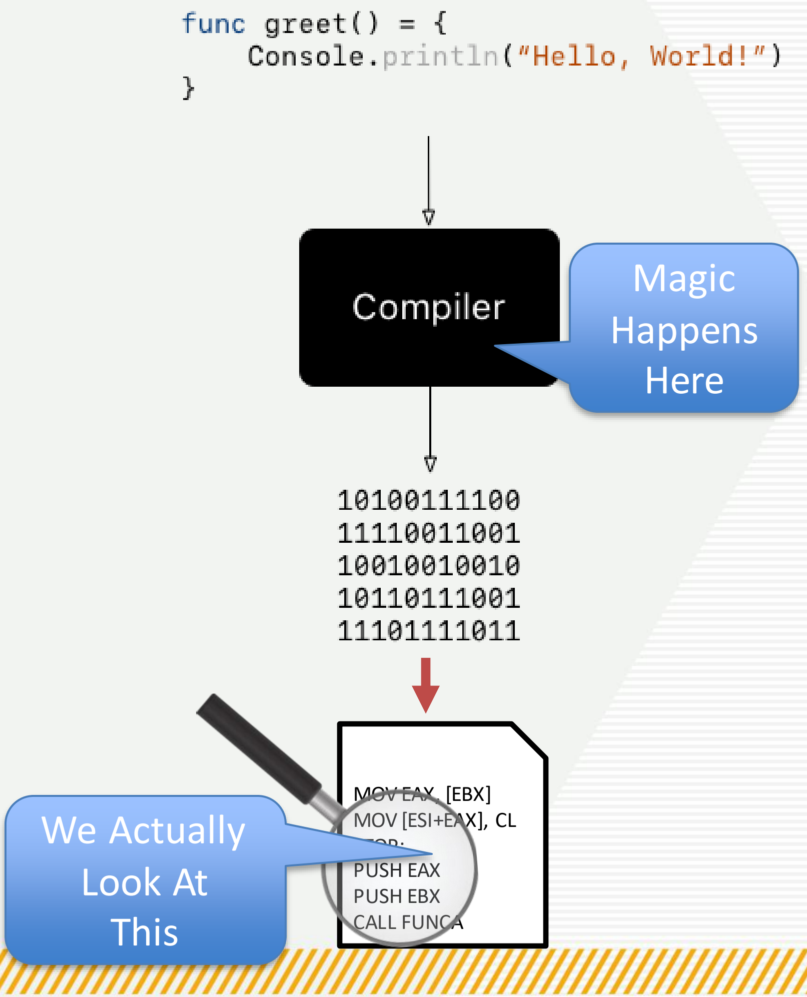
BINARY EXECUTABLE ANALYSIS



```
func greet() = {
    Console.println("Hello, World!")
}
```

- Malware does not come with source code
- So we are left analyzing only malware executables (a.k.a. binaries)
  - Is it hard? Yes
- An executable program is just a sequence of 1’s and 0’s which the CPU understands as instructions
- Reverse Engineering: The process of analyzing a subject binary program to create representations of the program’s logic at a higher level of abstraction

YOUR NEW HOBBY: ASSEMBLY LANGUAGE



```
func greet() = {
    Console.println("Hello, World!")
}
```

```
10100111100
11110011001
10010010010
10110111001
11101111011
```

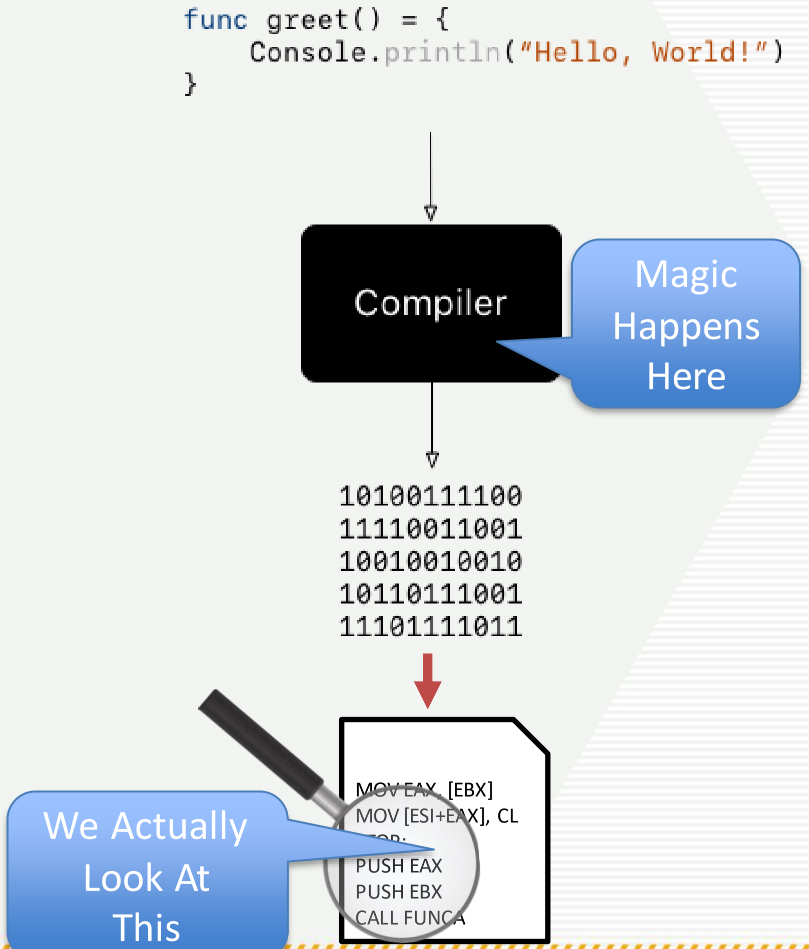
```
MOV EAX, [EBX]
MOV [ESI+EAX], CL
STOP:
PUSH EAX
PUSH EBX
CALL FUNCA
```

- Lucky for us, binary programs can be disassembled back to Assembly Language
- "An executable program is just a sequence of 1's and 0's which the CPU understands as instructions"
- Assembly code is just a sequence of **mnemonics** which represent each processor instruction (called an **opcode**)
- Opcode Disassembly Example:

0000 0100 0000 1010 = 040Ah = **add al,10**

CREATING THE NEXT®

YOUR NEW HOBBY: ASSEMBLY LANGUAGE



```
func greet() = {
    Console.println("Hello, World!")
}
```

```
10100111100
11110011001
10010010010
10110111001
11101111011
```

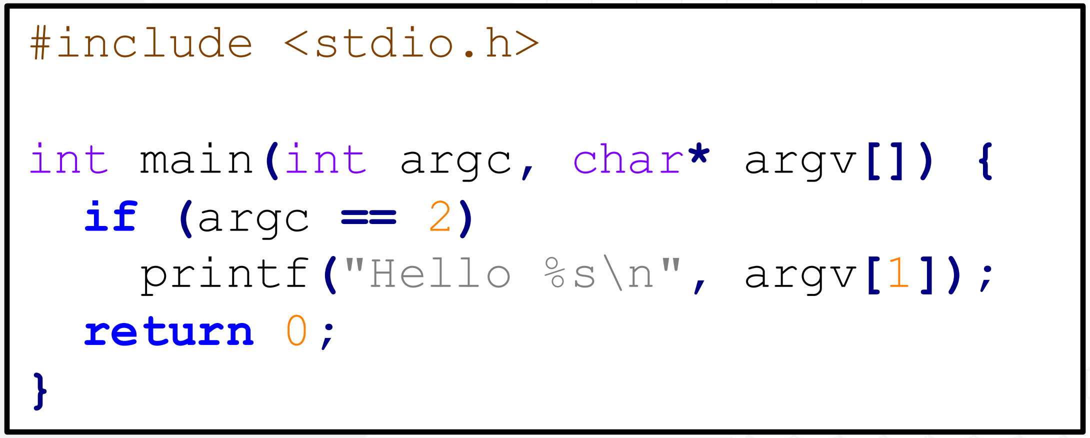
```
MOV EAX, [EBX]
MOV [ESI+EAX], CL
LOOP:
PUSH EAX
PUSH EBX
CALL FUNCA
```

- Now you need to know **everything** about Intel assembly

- Recommendations for success:
  - Read everything you can find about assembler
  - Read the Intel architecture manuals
  - Write native assembler applications and verify that they work properly
  - Compile programs that you're familiar with and examine the resulting assembler to understand what's going on
  - Read, read, read!

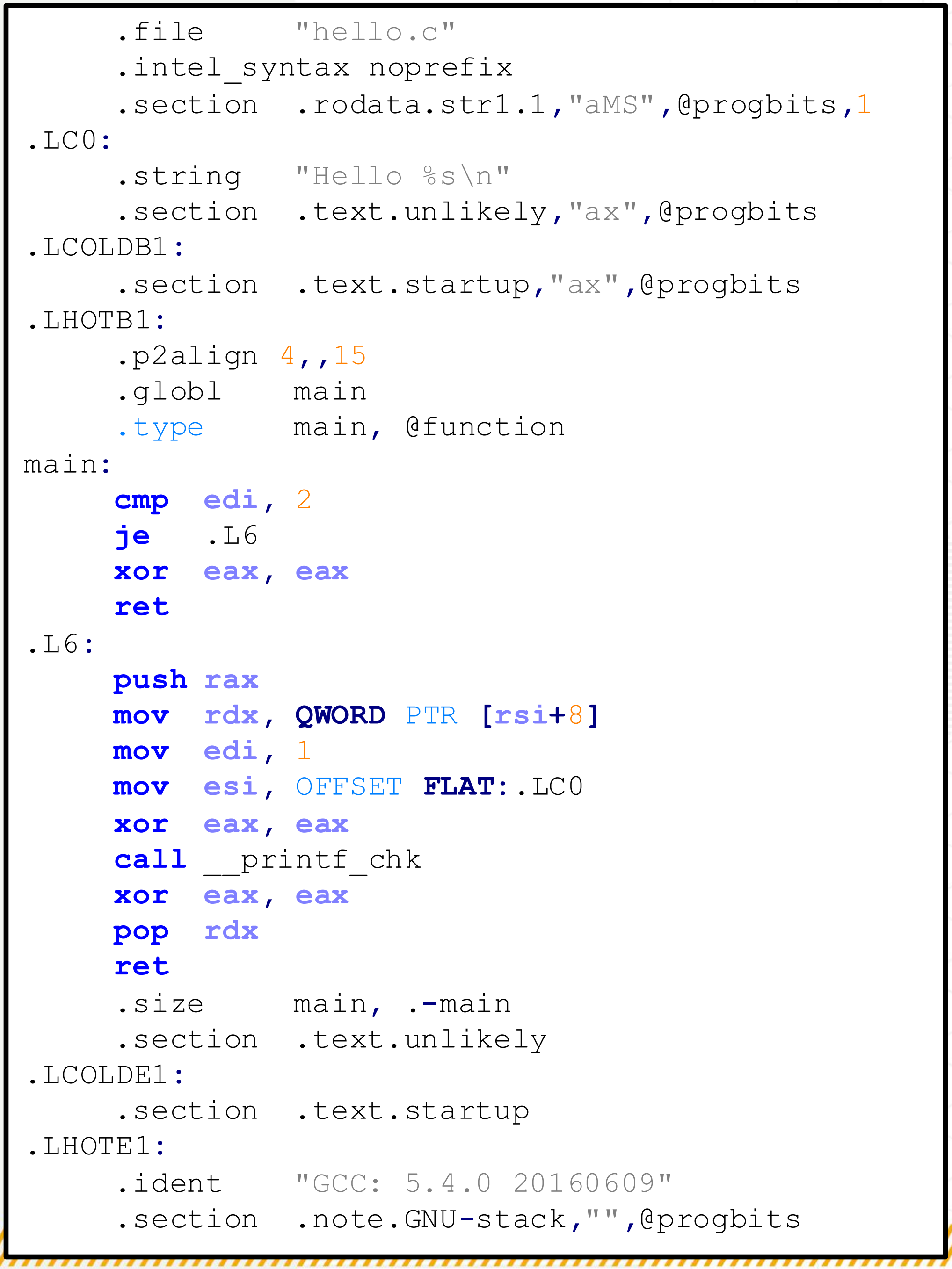
- In class: a quick tutorial on Intel assembly

COMPILING C CODE



```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc == 2)
    printf("Hello %s\n", argv[1]);
  return 0;
}
```

- How much code is generated?
- How complex is the executable?
- gcc -W -Wall -Wextra -Wpedantic -O0 -S -masm=intel hello.c -o hello.s



```
     .file     "hello.c"
     .intel_syntax noprefix
     .section  .rodata.str1.1,"aMS",@progbits,1
.LC0:
     .string   "Hello %s\n"
     .section  .text.unlikely,"ax",@progbits
.LCOLDB1:
     .section  .text.startup,"ax",@progbits
.LHOTB1:
     .p2align 4,,15
     .globl    main
     .type     main, @function
main:
     cmp  edi, 2
     je   .L6
     xor  eax, eax
     ret
.L6:
     push rax
     mov  rdx, QWORD PTR [rsi+8]
     mov  edi, 1
     mov  esi, OFFSET FLAT:.LC0
     xor  eax, eax
     call __printf_chk
     xor  eax, eax
     pop  rdx
     ret
     .size     main, .-main
     .section  .text.unlikely
.LCOLDE1:
     .section  .text.startup
.LHOTE1:
     .ident    "GCC: 5.4.0 20160609"
     .section  .note.GNU-stack,"",@progbits
```

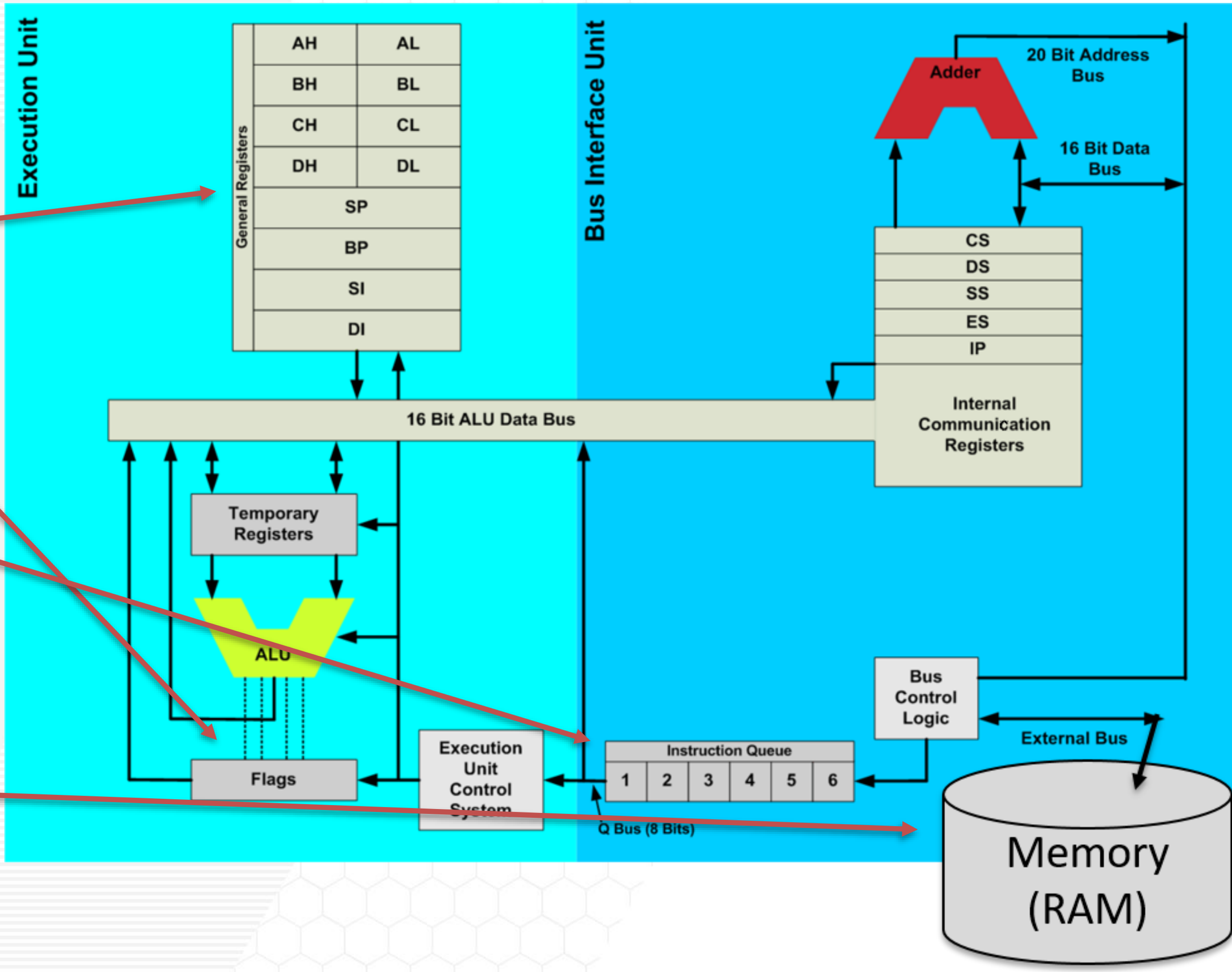
By the end of these slides, you will understand every line of this ☺

INTEL ASSEMBLER: NEED TO KNOW

You must begin thinking like a processor…

- Registers
  - Store "Live" Data
- Flags
  - Control the CPU's decisions
- Instructions
  - Tell the CPU what to do
- Data Formats
  - How data is stored
- Stack & Heap
  - How memory is accessed
- Intel vs. AT&T Syntax
- Executable Formats

Basic 16-bit CPU Architecture



CREATING THE NEXT®

X86 ASSEMBLER: REGISTERS

- Maximum register size depends on the CPU:
- 8088 / 8086 / 80186 / 80188 / 80286:
  - ➢ 16-bit registers
- 80386 / 80486 / Pentium / Pentium Pro / Pentium MMX / Pentium II / Pentium M / Pentium III / Pentium 4:
  - ➢ 32-bit registers
- Pentium 4 [later] / Pentium D / Pentium Extreme / Core2 (i3 / i5 / i7):
  - ➢ 64-bit registers
- You need to understand the differences between 16/32/64-bit Intel processors because of register naming and feature sets

BASIC REGISTER LIST (32-BIT NAMES)

Let’s start with 32-bit registers:

- EAX: The Accumulator
- EBX: The Base Register
- ECX: The Count Register
- EDX: The Data Register
- EDI: The Destination Index
- ESI: The Source Index
- EBP: Base Pointer
- ESP: Stack Pointer
- EIP: Instruction Pointer
- EFLAGS: processor flags

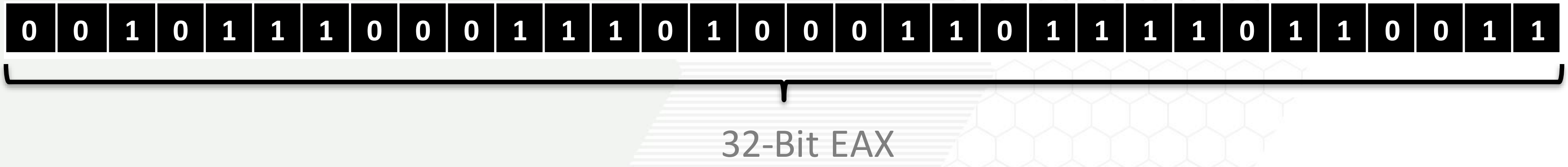
Mostly used as general-purpose storage, with some restrictions / special behaviors

Used to keep track of the stack (more on this later…)

“GENERAL PURPOSE” REGISTERS

- On a 32-bit processor, each register holds exactly 32 bits
  - This is used to store any binary value the program needs

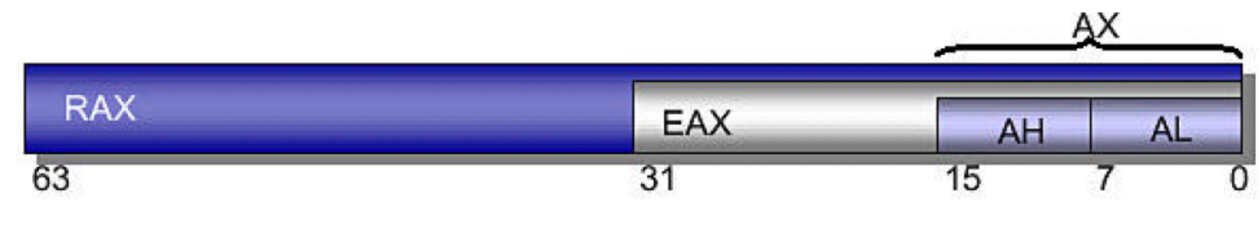
- Consider storing the number 775,567,283 in the EAX register:



0	0	1	0	1	1	1	0	0	0	1	1	1	0	1	0	0	0	1	1	0	1	1	1	1	0	1	1	0	0	1	1

“GENERAL PURPOSE” REGISTERS

- On 64-bit processors, you have 64-bit registers: rax, rbx, rcx, rdx
- 32-bit registers: eax, ebx, ecx, edx
  - Are actually the low order 32 bits of the 64-bit registers
- 16-bit registers: ax, bx, cx, dx
  - Are the low order 16 bits of the 32-bit registers eax, ebx, ecx, edx
- 8-bit registers: ah, al, bh, bl, ch, cl, dh, dl
  - Are bits 8-15 and 0-7 of the 16-bit registers ax, bx, cx, dx



- 64-bit mode introduces an additional 8 new 64-bit general purpose registers, r8 – r15
- R8=64 bits, r8d=32 bits, r8w = 16 bits, r8b = 8 bits
- No “h” mode for these (i.e., no direct access to bits 8-15)

8/16/32/64-BIT MADNESS

- Consider storing the number 775,567,283 in the EAX register:

0 0 1 0 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0 1 1

32-Bit EAX

- If you read this from RAX, it still equals 775,567,283:

… 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0 1 1

64-Bit RAX

- If you read this from AX, it equals 14,259:

0 0 1 0 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0 1 1

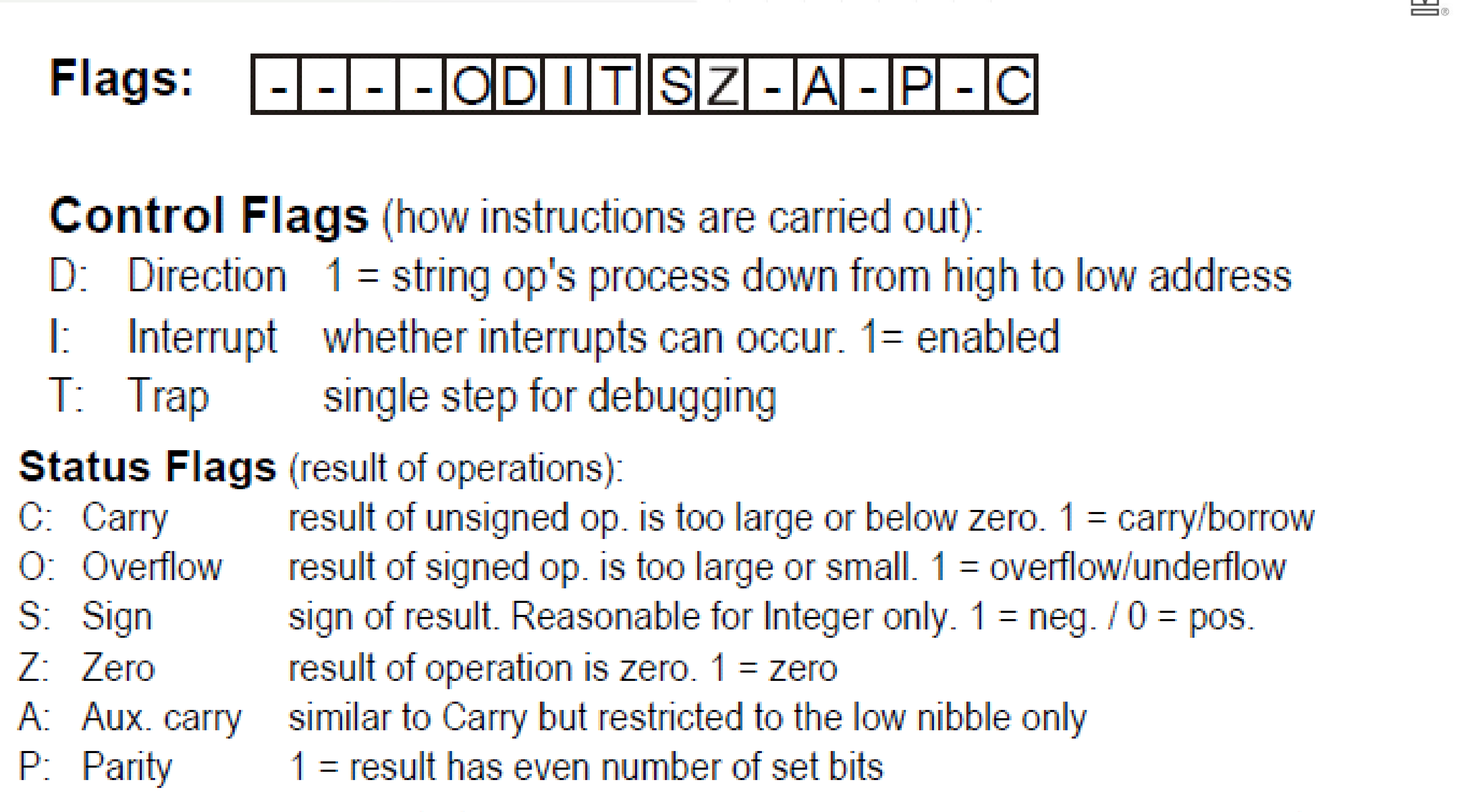
16-Bit AX

- If you read this from AL, it equals 179 or -77 **(signed integer, read up on 2's compliment)**

0 0 1 0 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0 1 1

8-Bit AL

EFLAGS (OR RFLAGS ON 64-BIT)



Flags:

-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C

Control Flags (how instructions are carried out):

D: Direction 1 = string op's process down from high to low address

I: Interrupt whether interrupts can occur. 1= enabled

T: Trap single step for debugging

Status Flags (result of operations):

C: Carry result of unsigned op. is too large or below zero. 1 = carry/borrow

O: Overflow result of signed op. is too large or small. 1 = overflow/underflow

S: Sign sign of result. Reasonable for Integer only. 1 = neg. / 0 = pos.

Z: Zero result of operation is zero. 1 = zero

A: Aux. carry similar to Carry but restricted to the low nibble only

P: Parity 1 = result has even number of set bits

- We will see this again when we talk about the CMP instruction

CREATING THE NEXT®

YES, MORE

- Floating point registers
  - 80 bit ST(0) – ST(7)
  - Organized as a stack + control registers
- MMX (stands for nothing, but used for multimedia):
  - MM0-MM7
  - Aliases for lower 64 bits of existing FP registers ST(0) – ST(7)
- SSE (Streaming SIMD Extensions)
  - XMM0-XMM15 (only 0-7 for 32-bit)
    - Independent, 128 bit
- We'll look at these only as needed in the case studies
- Also control registers that support, e.g., processor features, the debugging and virtualization architectures
  - CR0 – CR8 (see Intel manuals, debugging architecture)
  - More on these when we look at anti-analysis techniques

X86: BASIC DATA TYPES

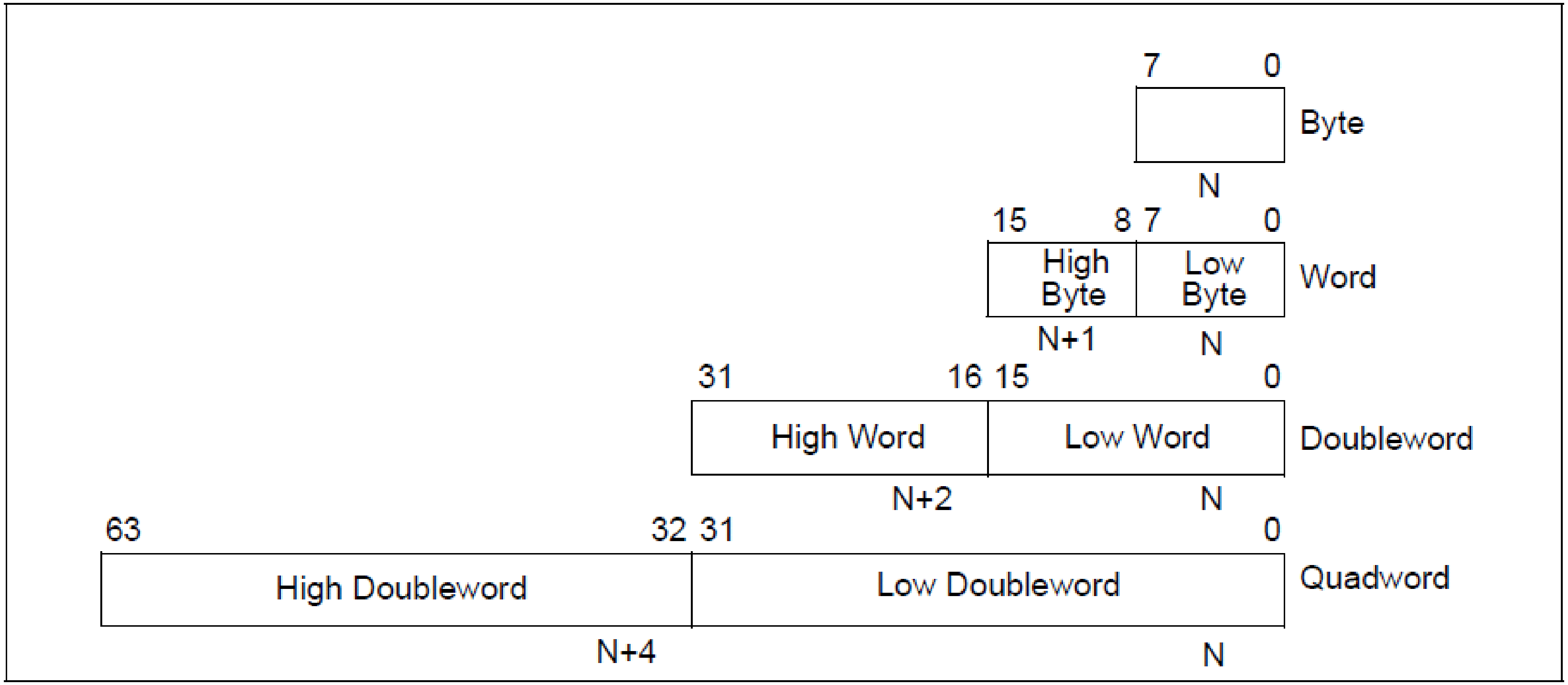


Figure 5-1. Fundamental Data Types

INSTRUCTION FORMATS

[LABEL:] INSTRUCTION destop [, sourceop] [; comment]

e.g.:

```
HERE: cmp ebx, 0xBEEF   ; does ebx contain magic #?
push ebx
push eax
xor ebx, ebx     ; ebx = 0
xor eax, eax     ; eax = 0
```

CREATING THE NEXT®

DATA ADDRESSING MODES

- Immediate
  - Value is a constant

```
mov rax, 0xBEEF  ; store 0xBEEF in rax
```

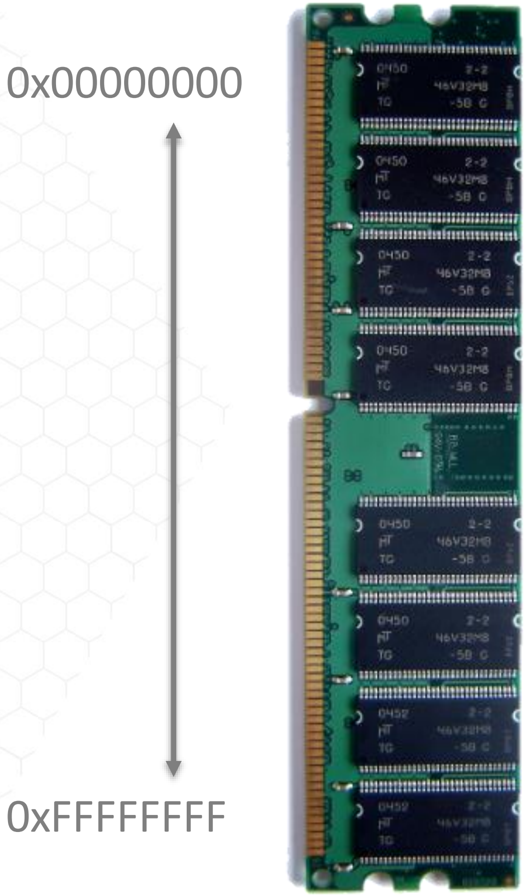
- Register addressing
  - Use value in register

```
mov eax, ebx  ; store the value held in ebx into eax
```

- Indexed / Memory operands (next slide)

MEMORY OPERANDS

- To the CPU: Memory looks like a huge contiguous bank of **bytes**
- **Notice:** Memory is **byte** addressable!



- The first byte starts at address 0x00000000
- The second byte starts at address 0x00000001

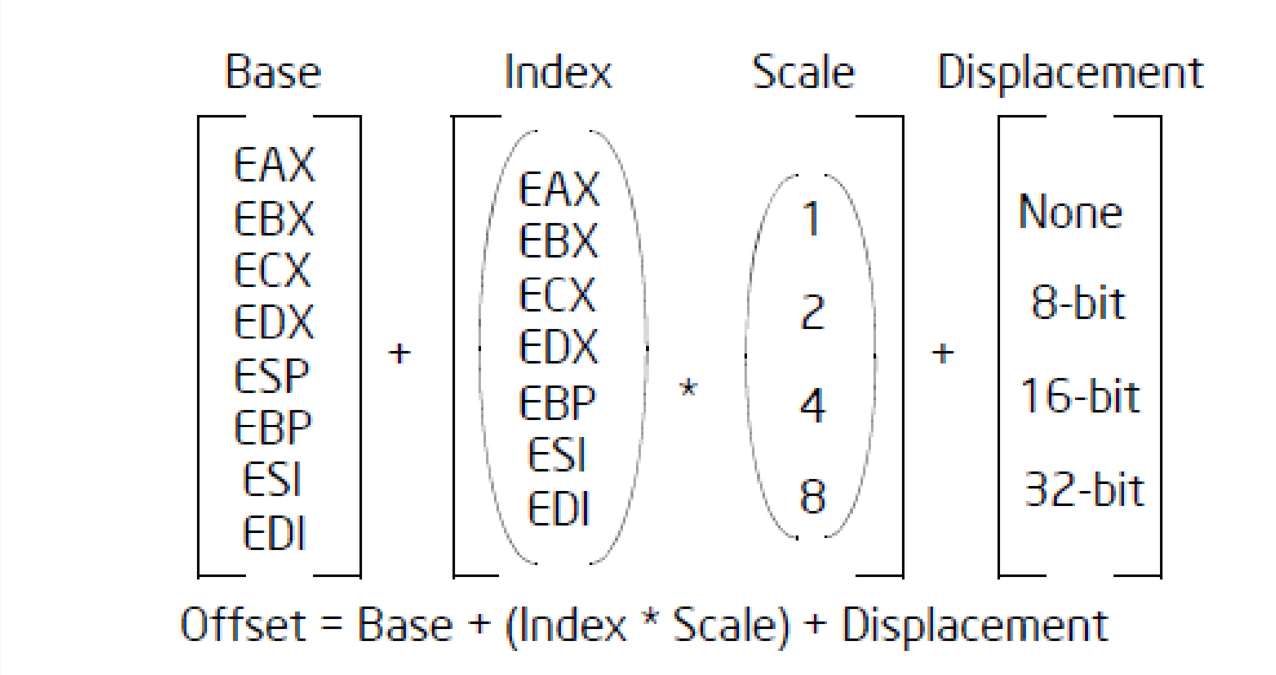
- But special instructions have to tell the CPU “go to memory!”
- Write:

```
mov ebx, 0x00000001      ; set the byte at
mov BYTE PTR [ebx], 0    ; 0x00000001 to 0
```

- Read:

```
mov ebx, 0x00000001      ; set al equal to the
mov al, BYTE PTR [ebx]   ; byte at 0x00000001
```

INDEXED & MEMORY OPERANDS



Offset = Base + (Index * Scale) + Displacement

```
mov BYTE PTR [ebx], 0    ; address in ebx (as a byte) <- 0
mov DWORD PTR [ebx], 0   ; address in ebx (as a 32-bit) <- 0
mov al, BYTE PTR 0x10003564 ; set al to the byte at 0x10003564
mov [ecx+8*ebx], eax     ; address in ecx + 8*ebx <- eax
mov 100[ecx+4*ebx], eax  ; address in ecx + 4*ebx + 100 <- eax
```

DATA SECTION: ALLOCATING STORAGE

```
friend: BYTE “joe”
friend: BYTE 'j', 'o', 'e'
```

Same Effect

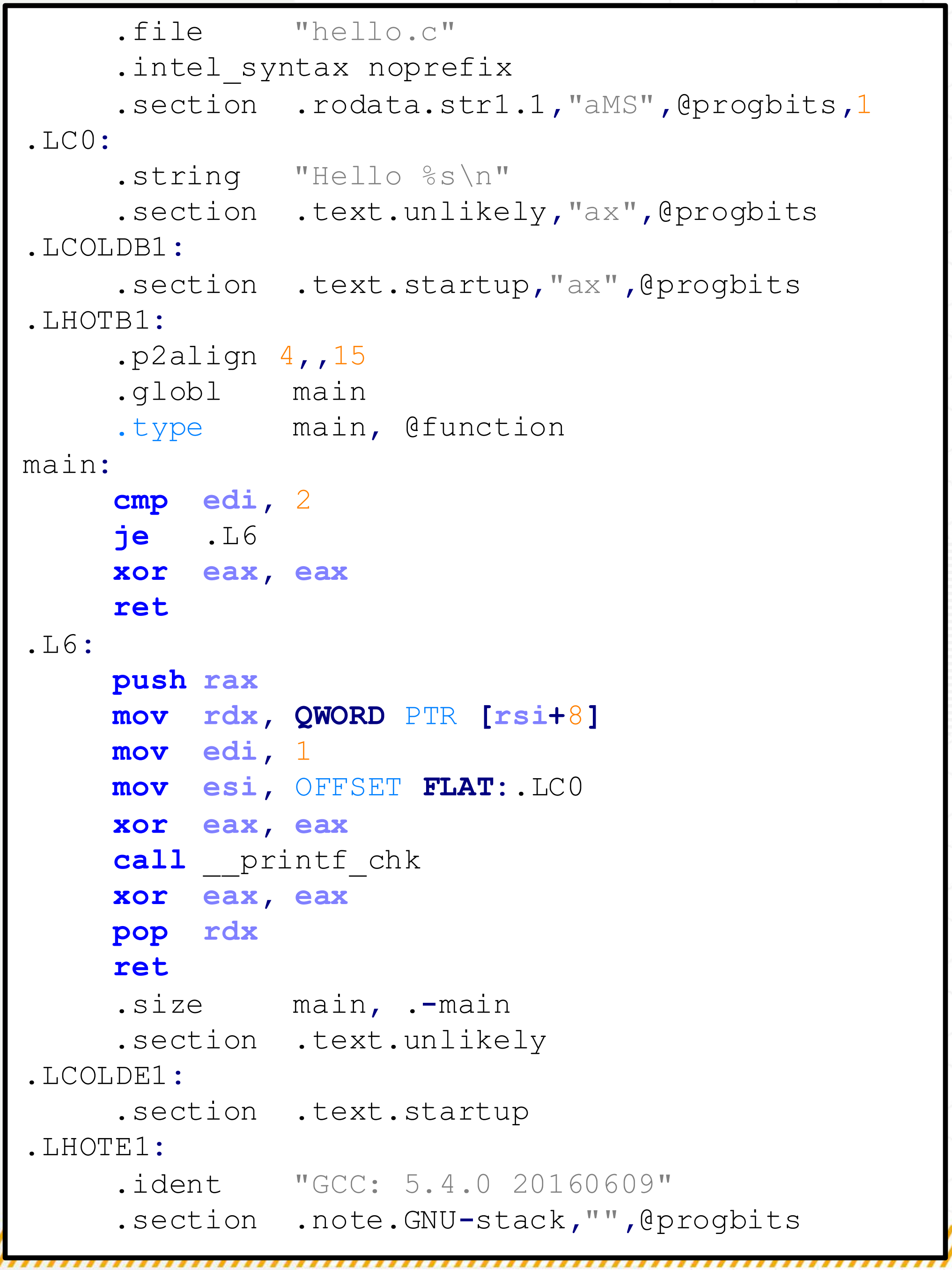
```
gross:  DWORD 144
gross:  DWORD 12*12
gross:  DWORD 10*15-7+1
gross:  DWORD 90h         ; hex
gross:  DWORD 10010000b   ; binary
gross:  DWORD 220o      ; octal
```

Same Effect

```
values: DWORD 10, 20, 30, 40  ; (4) 32-bit values

bigval: QWORD 99999999999     ; 64 bit
```

MORE ALLOCATING

```
negint: SDWORD -32          ; signed 32-bit int

bigger: TBYTE  0              ; ten byte int

stars: BYTE    50 DUP('*')    ; 50 asterisks

fl1:  REAL4  3.14           ; 32-bit float

fl2:  REAL8  3.1415         ; 64-bit float

fl3:  REAL10 3.1415926535   ; 80-bit float
```



```
     .file     "hello.c"
     .intel_syntax noprefix
     .section  .rodata.str1.1,"aMS",@progbits,1
.LC0:
     .string   "Hello %s\n"
     .section  .text.unlikely,"ax",@progbits
.LCOLDB1:
     .section  .text.startup,"ax",@progbits
.LHOTB1:
     .p2align 4,,15
     .globl    main
     .type     main, @function
main:
     cmp  edi, 2
     je   .L6
     xor  eax, eax
     ret
.L6:
     push rax
     mov  rdx, QWORD PTR [rsi+8]
     mov  edi, 1
     mov  esi, OFFSET FLAT:.LC0
     xor  eax, eax
     call __printf_chk
     xor  eax, eax
     pop  rdx
     ret
     .size     main, .-main
     .section  .text.unlikely
.LCOLDE1:
     .section  .text.startup
.LHOTE1:
     .ident    "GCC: 5.4.0 20160609"
     .section  .note.GNU-stack,"",@progbits
```

Let’s see how much we understand now

EXECUTABLE FILES HAVE SECTIONS

The .section directive is used like this:

```
.section name [, "flags"[, @type[,flag_specific_arguments]]]
```

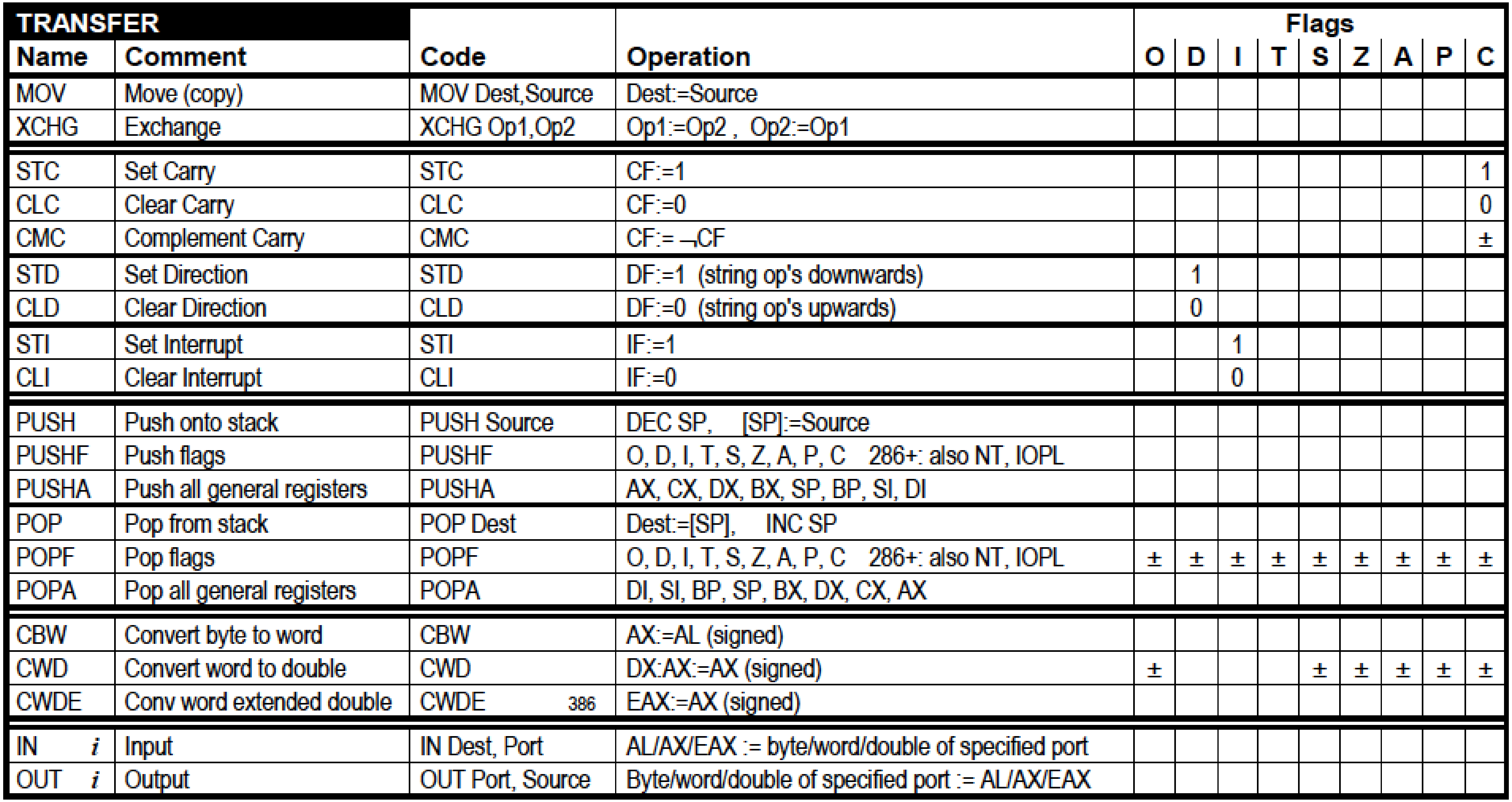
- a - section is allocatable
- e - section is excluded from executable and shared library
- w - section is writable
- x - section is executable
- M - section is mergeable
- S - contains zero terminated strings
- G - section is a member of a section group
- T - section is used for thread-local-storage
- ? - section is a member of the previously-current section's group, if any

- @progbits section contains data
- @nobits section w/o data (i.e., only occupies space)
- @note section contains non-program data
- @init_array section contains an array of ptrs to init functions
- @fini_array section contains an array of ptrs to finish functions
- @preinit_array section contains an array of ptrs to pre-init functions

```
        .file       "hello.c"
        .intel_syntax noprefix
        .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string     "Hello %s\n"
        .section    .text.unlikely,"ax",@progbits
.LCOLDB1:
        .section    .text.startup,"ax",@progbits
.LHOTB1:
        .p2align 4,,15
        .globl      main
        .type       main, @function
main:
        cmp  edi, 2
        je   .L6
        xor  eax, eax
        ret
.L6:
        push rax
        mov  rdx, QWORD PTR [rsi+8]
        mov  edi, 1
        mov  esi, OFFSET FLAT:.LC0
        xor  eax, eax
        call __printf_chk
        xor  eax, eax
        pop  rdx
        ret
        .size       main, .-main
        .section    .text.unlikely
.LCOLDE1:
        .section    .text.startup
.LHOTE1:
        .ident      "GCC: 5.4.0 20160609"
        .section    .note.GNU-stack,"",@progbits
```

Notice the sections?

COMMON INTEL INSTRUCTIONS



TRANSFER				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:= ¬CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±				±	±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN *i*	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port									
OUT *i*	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

Source: IntelCodeTable, Roger Jegerlehner

BE VERY CAREFUL!

- These slides are NOT meant to exhaustively teach you everything about assembly!
- You **must** read the Intel manuals & online references when reverse engineering

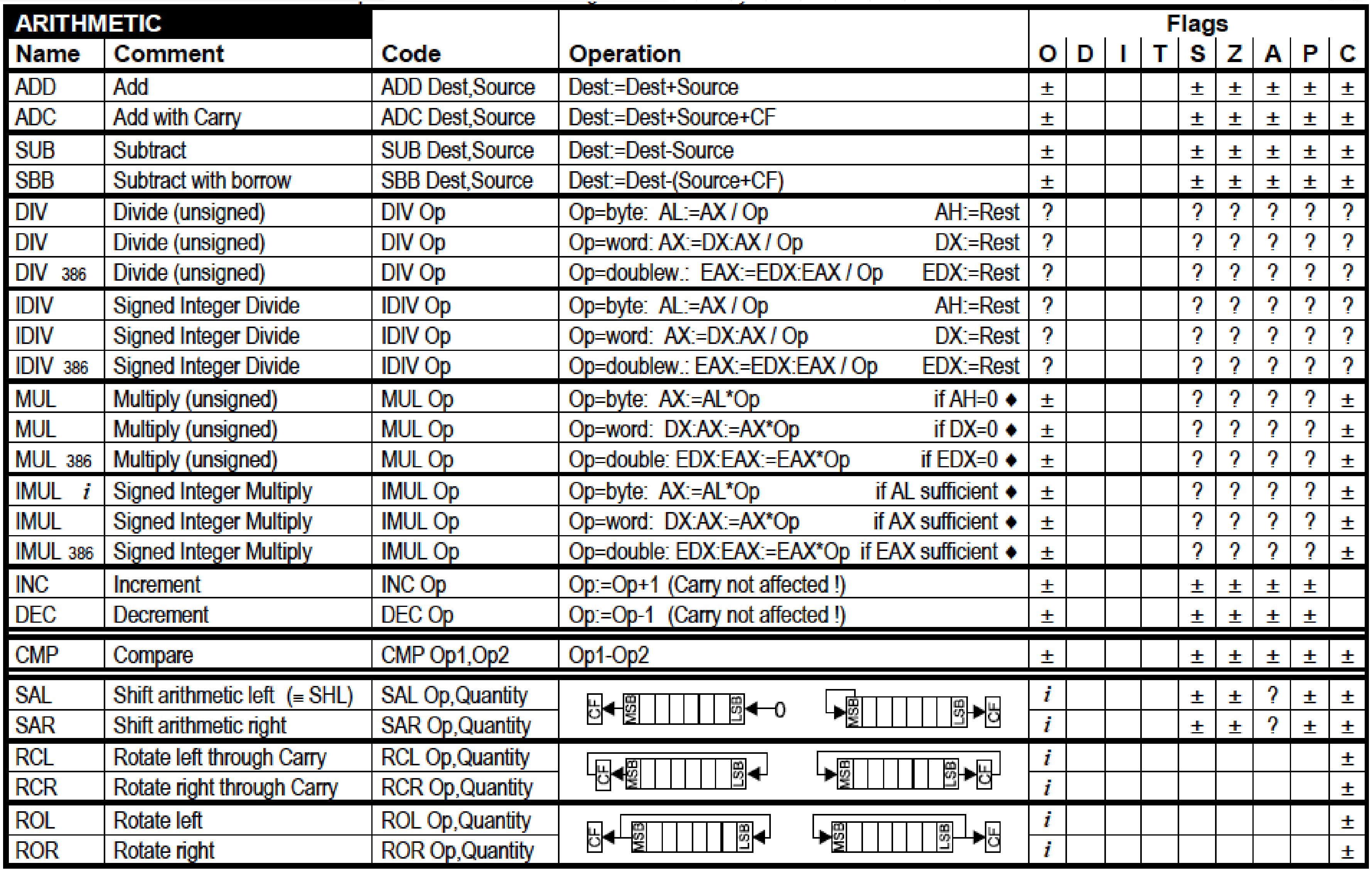
- For example:

```
mov eax, ebx
```

- On a 64-bit CPU?
- … automatically zeroes the upper 32 bits of RAX!

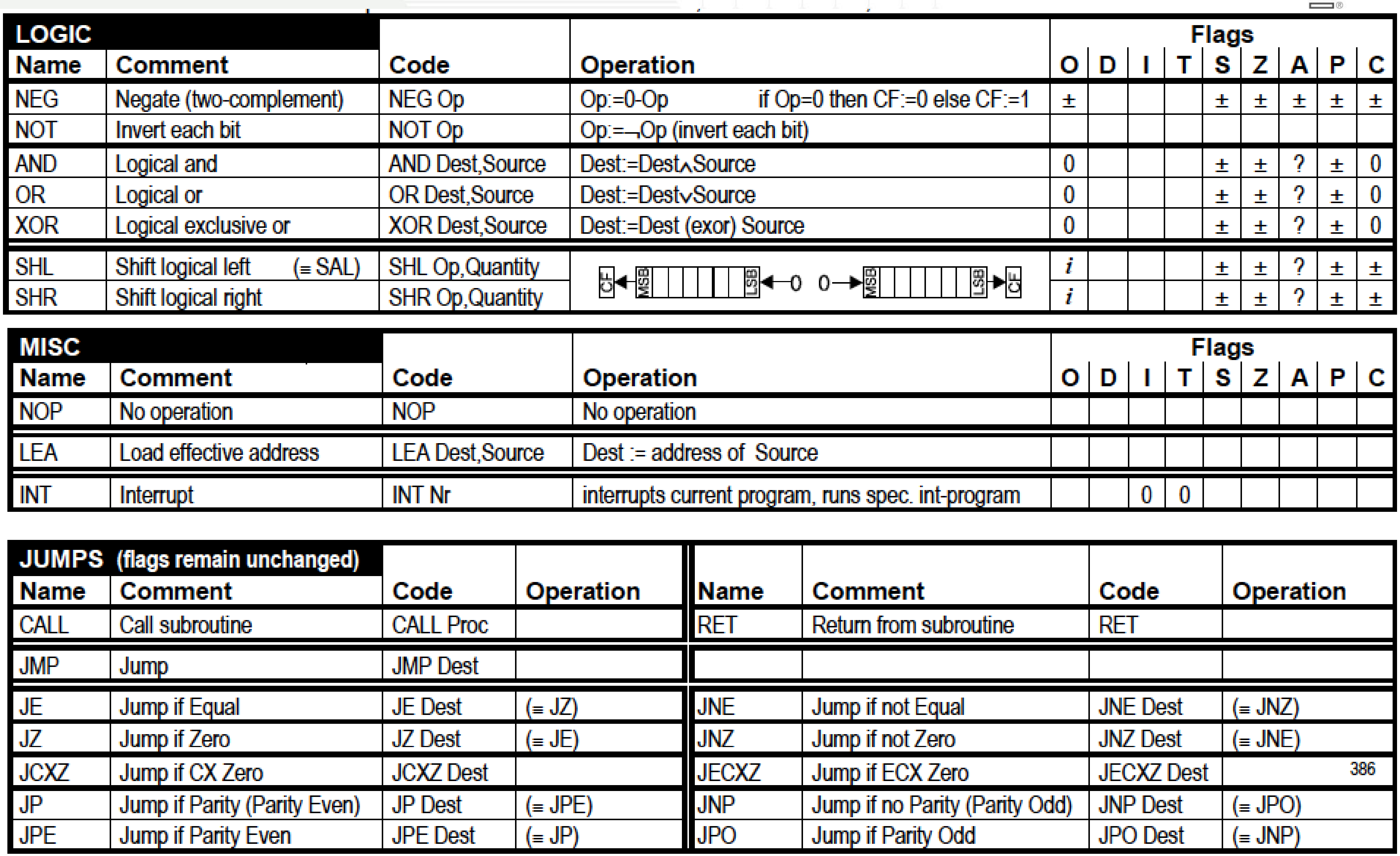
- The manual says:
  - 64-bit operands generate a 64-bit result in the destination 64-bit register
  - 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination register
  - 8-bit and 16-bit operands generate an 8-bit or 16-bit result in the destination 64-bit register
    - Upper 56 bits or 48 bits (respectively) of the destination 64-bit register **are left intact!**

COMMON INTEL INSTRUCTIONS (2)



ARITHMETIC				Flags								
Name	**Comment**	**Code**	**Operation**	**O**	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	
CMP	Compare	CMP Op1,Op2	Op1-Op2	±				±	±	±	±	±
SAL	Shift arithmetic left (≡ SHL)	SAL Op,Quantity	CF ← MSB … LSB ← 0 MSB … LSB → CF	*i*				±	±	?	±	±
SAR	Shift arithmetic right	SAR Op,Quantity		*i*				±	±	?	±	±
RCL	Rotate left through Carry	RCL Op,Quantity	CF ← MSB … LSB ← MSB … LSB → CF	*i*								±
RCR	Rotate right through Carry	RCR Op,Quantity		*i*								±
ROL	Rotate left	ROL Op,Quantity	CF ← MSB … LSB ← MSB … LSB → CF	*i*								±
ROR	Rotate right	ROR Op,Quantity		*i*								±

COMMON INTEL INSTRUCTIONS (3)

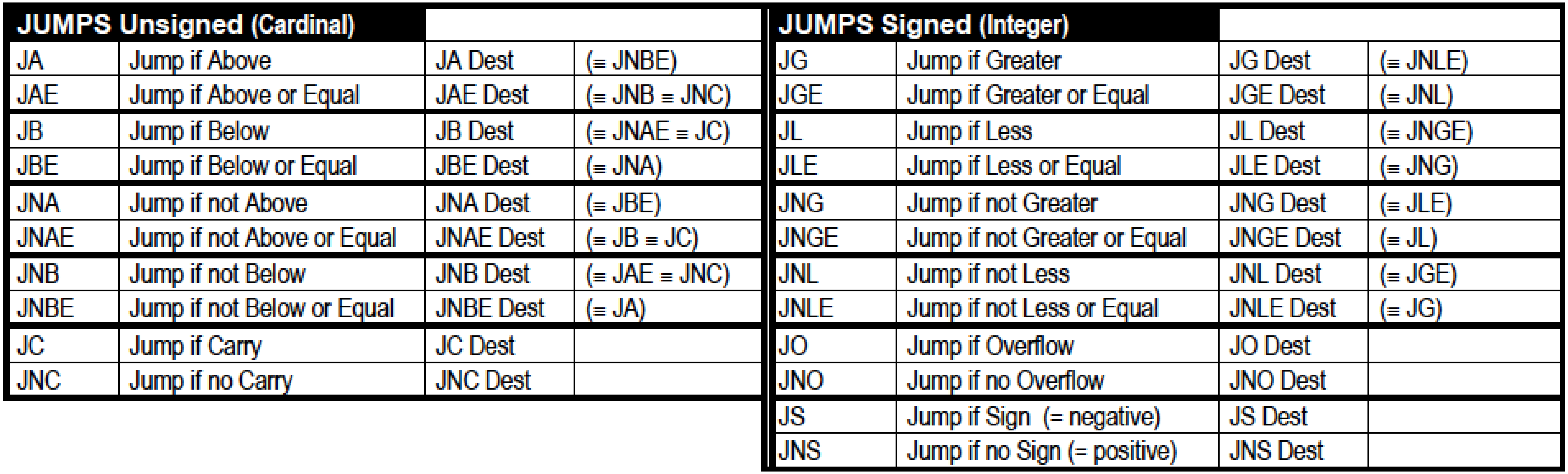


LOGIC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=¬Op (invert each bit)									
AND	Logical and	AND Dest,Source	Dest:=Dest∧Source	0				±	±	?	±	0
OR	Logical or	OR Dest,Source	Dest:=Dest∨Source	0				±	±	?	±	0
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (exor) Source	0				±	±	?	±	0
SHL	Shift logical left (≡ SAL)	SHL Op,Quantity	CF ← MSB LSB ← 0 0 → MSB LSB → CF	i				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity		i				±	±	?	±	±

MISC				Flags								
Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NOP	No operation	NOP	No operation									
LEA	Load effective address	LEA Dest,Source	Dest := address of Source									
INT	Interrupt	INT Nr	interrupts current program, runs spec. int-program			0	0					

JUMPS (flags remain unchanged)							
Name	Comment	Code	Operation	Name	Comment	Code	Operation
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest	(≡ JZ)	JNE	Jump if not Equal	JNE Dest	(≡ JNZ)
JZ	Jump if Zero	JZ Dest	(≡ JE)	JNZ	Jump if not Zero	JNZ Dest	(≡ JNE)
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	386
JP	Jump if Parity (Parity Even)	JP Dest	(≡ JPE)	JNP	Jump if no Parity (Parity Odd)	JNP Dest	(≡ JPO)
JPE	Jump if Parity Even	JPE Dest	(≡ JP)	JPO	Jump if Parity Odd	JPO Dest	(≡ JNP)

COMMON INTEL INSTRUCTIONS (4)



JUMPS Unsigned (Cardinal)			
JA	Jump if Above	JA Dest	(≡ JNBE)
JAE	Jump if Above or Equal	JAE Dest	(≡ JNB ≡ JNC)
JB	Jump if Below	JB Dest	(≡ JNAE ≡ JC)
JBE	Jump if Below or Equal	JBE Dest	(≡ JNA)
JNA	Jump if not Above	JNA Dest	(≡ JBE)
JNAE	Jump if not Above or Equal	JNAE Dest	(≡ JB ≡ JC)
JNB	Jump if not Below	JNB Dest	(≡ JAE ≡ JNC)
JNBE	Jump if not Below or Equal	JNBE Dest	(≡ JA)
JC	Jump if Carry	JC Dest	
JNC	Jump if no Carry	JNC Dest	

JUMPS Signed (Integer)			
JG	Jump if Greater	JG Dest	(≡ JNLE)
JGE	Jump if Greater or Equal	JGE Dest	(≡ JNL)
JL	Jump if Less	JL Dest	(≡ JNGE)
JLE	Jump if Less or Equal	JLE Dest	(≡ JNG)
JNG	Jump if not Greater	JNG Dest	(≡ JLE)
JNGE	Jump if not Greater or Equal	JNGE Dest	(≡ JL)
JNL	Jump if not Less	JNL Dest	(≡ JGE)
JNLE	Jump if not Less or Equal	JNLE Dest	(≡ JG)
JO	Jump if Overflow	JO Dest	
JNO	Jump if no Overflow	JNO Dest	
JS	Jump if Sign (= negative)	JS Dest	
JNS	Jump if no Sign (= positive)	JNS Dest	

Source: IntelCodeTable, Roger Jegerlehner

INTEL VS. AT&T SYNTAX

- There are actually two accepted representations for x86 assembly language
  - Intel Syntax and AT&T Syntax

- Virtually every tool for Windows uses Intel syntax
- gcc traditionally used AT&T syntax
  - Thus, virtually every tool for Linux uses AT&T syntax

- Can be overridden with appropriate switches
- WE WILL USE BOTH
  - So you need to be able to read either

- Here's a quick one slide cheat sheet…

INTEL VS. AT&T SYNTAX

- GENERAL INSTRUCTIONS:

Intel

```
push 4
add eax, 4
mov al, byte ptr FOO
call
jmp
ret
```

AT&T

```
pushl $4
addl $4, %eax
movb FOO, %al
lcall
ljmp
lret
```

- b = byte
- s = short (16 bit int or 32-bit float)
- w = word
- l = long (32 bit int or 64-bit float)
- q = quad
- t = ten bytes

- MEMORY REFERENCES:

Intel

```
displacement[base+index*scale]

mov eax, BYTE PTR [ebp-4]
mov ebx, BYTE PTR [foo+eax*4]
```

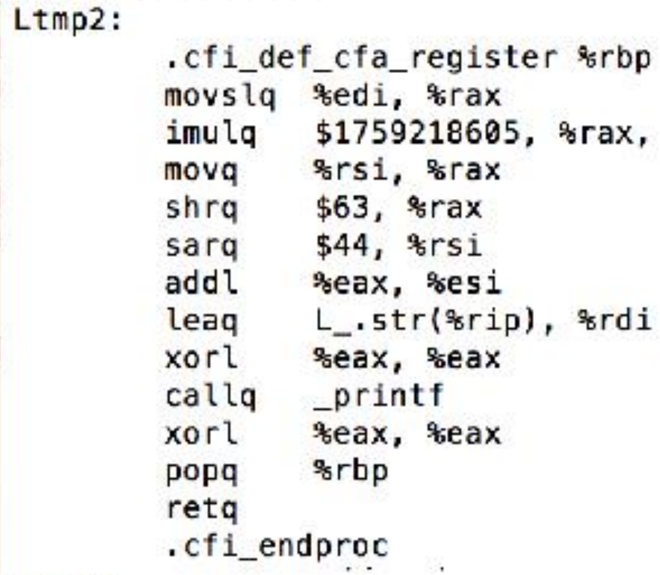
AT&T

```
displacement(base,index,scale)

movb -4(%ebp), %eax
movb foo(,%eax,4), %ebx
```

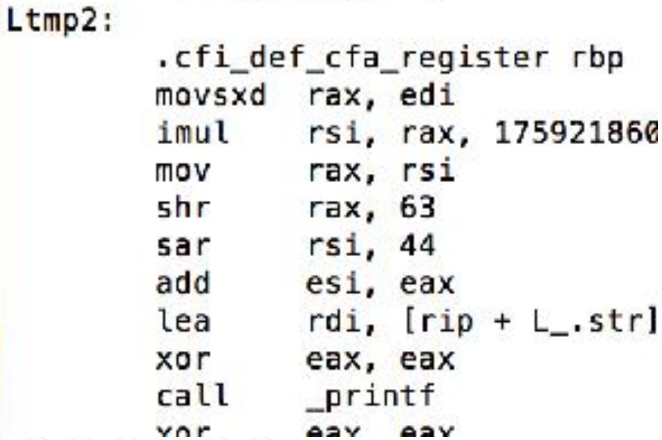
Note: constants in memory references do not need a “$”

CREATING THE NEXT®

LISTEN TO DRAKE



```
Ltmp2:
        .cfi_def_cfa_register %rbp
        movslq  %edi, %rax
        imulq   $1759218605, %rax,
        movq    %rsi, %rax
        shrq    $63, %rax
        sarq    $44, %rsi
        addl    %eax, %esi
        leaq    L_.str(%rip), %rdi
        xorl    %eax, %eax
        callq   _printf
        xorl    %eax, %eax
        popq    %rbp
        retq
        .cfi_endproc
```



```
Ltmp2:
        .cfi_def_cfa_register rbp
        movsxd  rax, edi
        imul    rsi, rax, 175921860
        mov     rax, rsi
        shr     rax, 63
        sar     rsi, 44
        add     esi, eax
        lea     rdi, [rip + L_.str]
        xor     eax, eax
        call    _printf
        xor     eax, eax
```

Credit To: Reddit

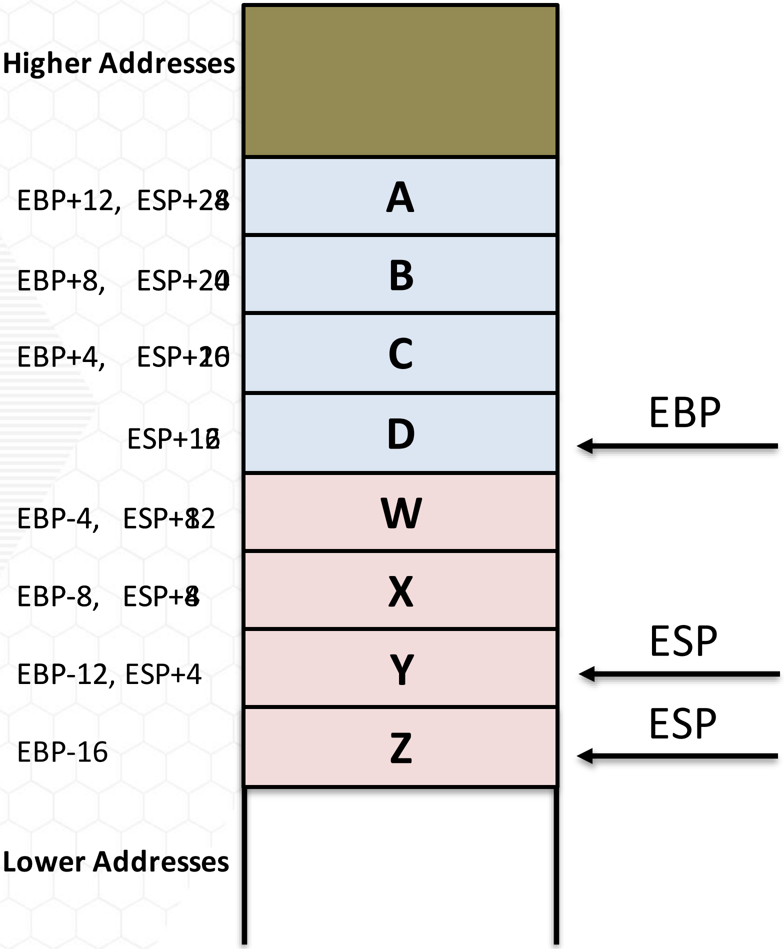
MEMORY STORAGE: STACK

- The stack is considered an endless sequence of memory "slots"
  - "Slots" are allocated in **descending** order!
  - The **EBP** register points to the most recent "Stack Base Address"
  - Stack base address is usually updated for each function call
  - The **ESP** register points to the **bottom-most used** "slot"
  - Data (the size of a register) is **pushed** into a free "slot"

```
push eax        sub esp, 4
                mov [esp], eax
```

- Data (the size of a register) is **popped** from the "slot" pointed to by ESP

```
pop eax         mov eax, [esp]
                add esp, 4
```

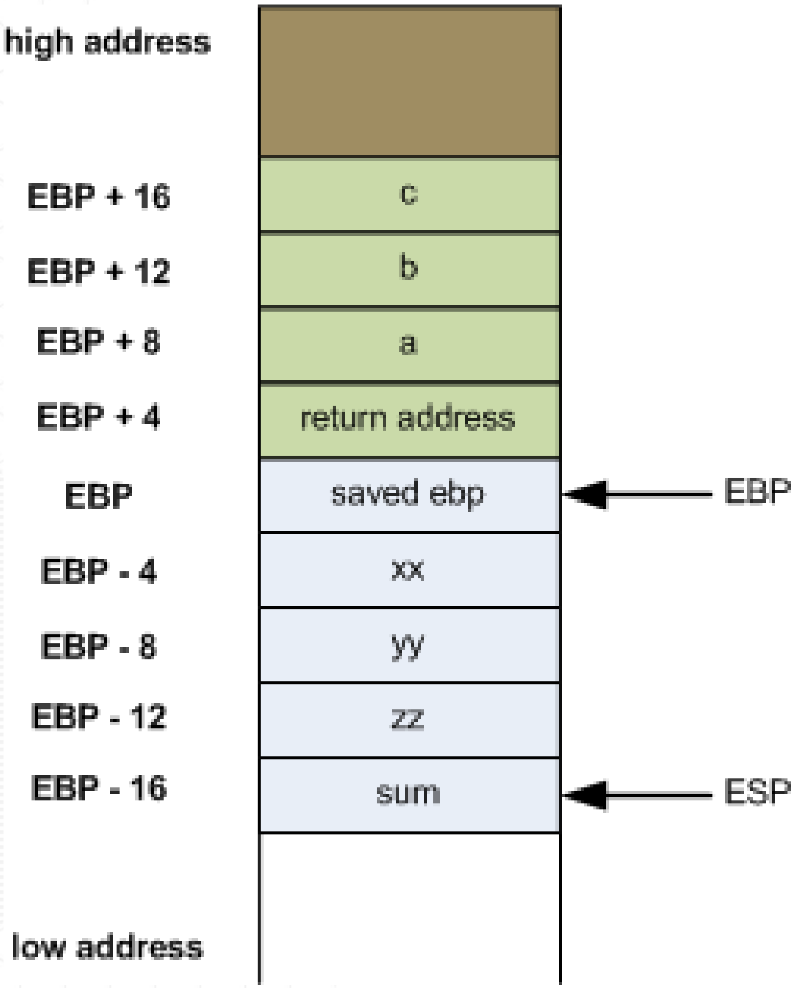


STACK AND FUNCTION CALL (16 & 32-BIT)

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}
```

```
_foobar:
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    …

_main:
    push    ecx
    push    ebx
    push    eax
    call    _foobar
```

Referred to as the "function prologue"



Credit To: Eli Bendersky

```
call _foobar  ==  push  ***
                  jmp   _foobar
```

*** Push the address of the next instruction (call _foobar is 5 bytes long)

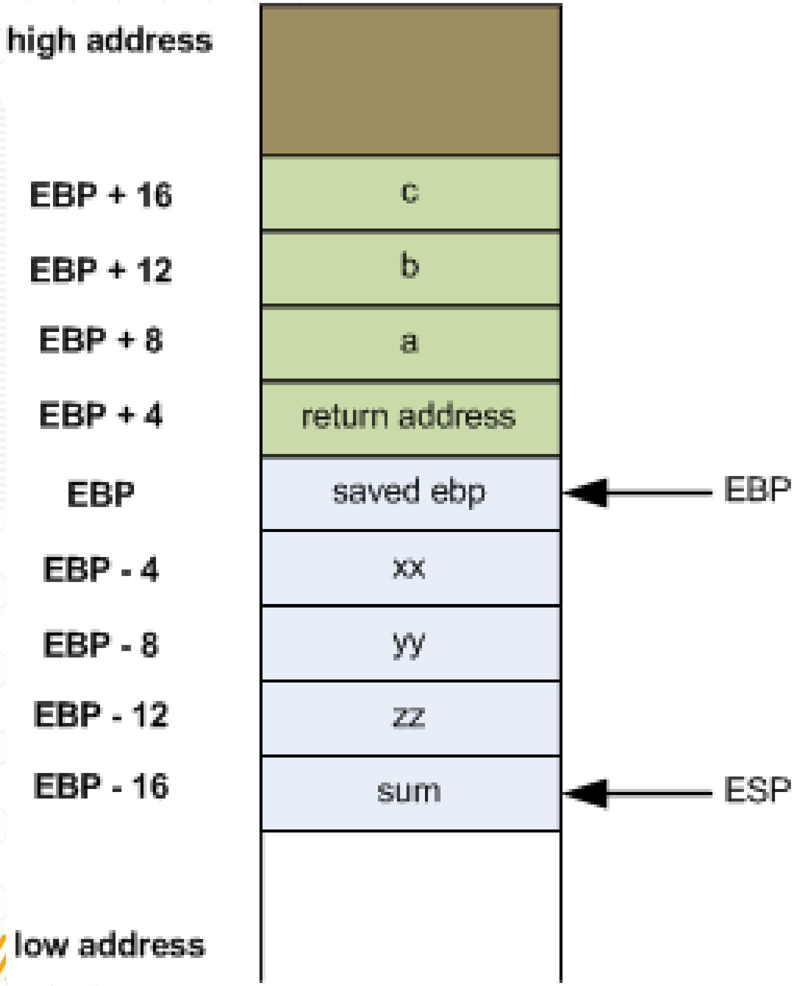
FUNCTION RETURN (SAME ON BOTH 32- & 64-BIT)

- The **ret** instruction performs a function return
- Prior to executing a RET instruction:
  1. The return value must be stored in EAX
  2. The stack must be cleaned up!

```
ret = pop eip
```

```
int foobar(int a, int b, int c)
{    …
     return xx * yy * zz + sum;
}
```

```
_foobar:
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    …
    add    eax, edx
    add    esp, 16
    pop    ebp
    ret
```



high address

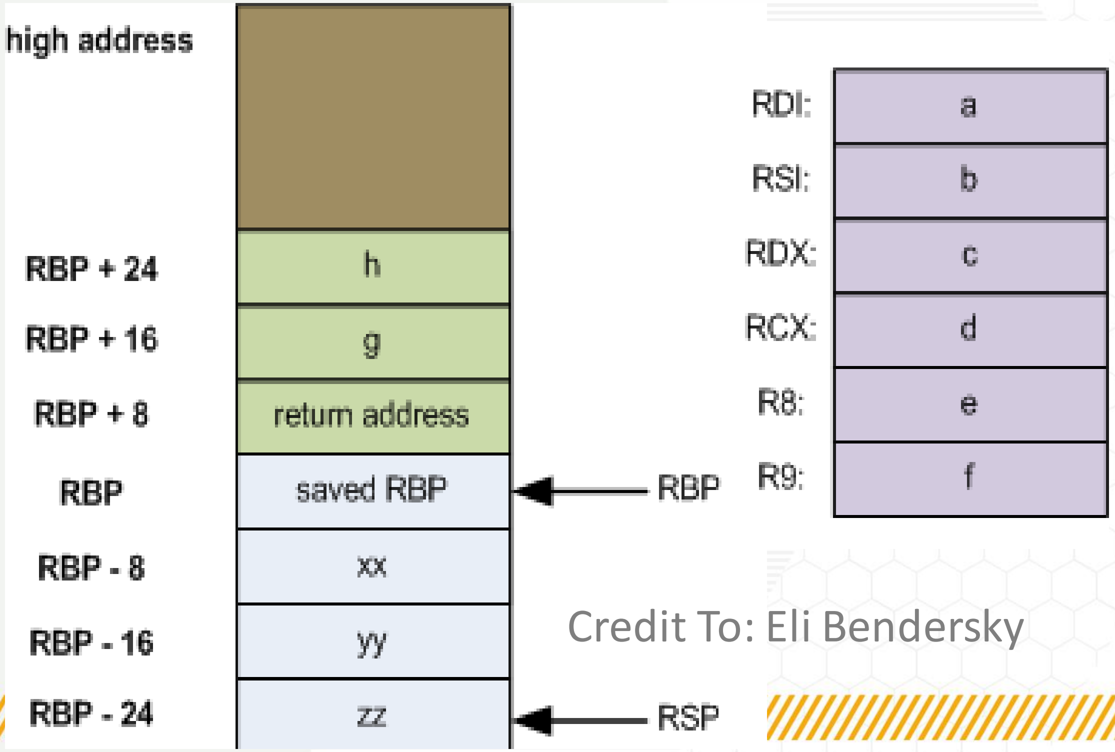
EBP + 16	c
EBP + 12	b
EBP + 8	a
EBP + 4	return address
EBP	saved ebp
EBP - 4	xx
EBP - 8	yy
EBP - 12	zz
EBP - 16	sum

low address

STACK AND FUNCTION CALL (64-BIT LINUX)

```c
long foobar(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = a - b - c - d - e - f - g - h;
    return zz * xx * yy;
}
```

```
foobar:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 24
    …

main:
    push    8
    push    7
    mov     r9, 6
    mov     r8, 5
    mov     rcx, 4
    mov     rdx, 3
    mov     rsi, 2
    mov     rdi, 1
    call    foobar
```



high address

RBP + 24 h

RBP + 16 g

RBP + 8 return address

RBP saved RBP ← RBP

RBP - 8 xx

RBP - 16 yy

RBP - 24 zz ← RSP

RDI: a

RSI: b

RDX: c

RCX: d

R8: e

R9: f

Credit To: Eli Bendersky

- Windows is similar, except only uses 4 registers! ☹
  - a in RCX
  - b in RDX
  - c in R8
  - d in R9

MIND THE CALLING CONVENTIONS!

- These "calling conventions" have grown up over the years
- More formally called Application Binary Interface (ABI)

- But you can imagine how many different ABIs were proposed and used
- Compiler authors, OS developers, library maintainers....

- Finally, most OSs mandated a few that they would use
- Read up on them here: https://en.wikipedia.org/wiki/X86_calling_conventions
  - https://en.wikipedia.org/wiki/X86_calling_conventions#List_of_x86_calling_conventions
- This will be important in future labs!
  - Who is responsible for cleaning arguments off the stack? The caller? The callee?
  - How does ellipse (" … ") type arguments work?
  - On and on and on...
  - Read, Read Read! It's all documented!

MEMORY STORAGE: HEAP

- Heap is much simpler! ☺

- The program will:
  1. Call malloc (or other heap allocation functions)
  2. Use the address that returns to access that malloc returns (recall: in EAX)

```
int foobar()
{
    int* x = malloc(sizeof(int));
    *x = 3;
    return *x;
}
```

```
foobar:
    push    ebp
    mov     ebp, esp
    push    4
    call    malloc
    mov     DWORD PTR [eax], 3
    mov     eax, [eax]
    add     esp, 4
    pop     ebp
    ret
```

EXTRA CREDIT PROJECT! ☺

Get your assembly skillz up to speed!

Instructions:

(1) Compile the hello world C code we saw in class into assembly code

(2) For each line with an assembly instruction, add a comment explaining what that instruction is doing

(3) Be smart about it! No “moves 2 into eax” Instead say: “the number of args must be 2”

Submission: Your commented assembly code file (called hello.s)

Grade: 5 extra credit points (fills in lost points on real assignments)

Submission Instructions: Upload your solution to the “Extra Credit #1” Assignment in Canvas

Due Date: The beginning of next class!

Here is the gcc command and the code:

```
gcc -W -Wall -Wextra -Wpedantic
 -fno-asynchronous-unwind-tables
 -O0 -S -masm=intel hello.c -o hello.s
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc == 2)
    printf("Hello %s\n", argv[1]);
  return 0;
}
```

ADDITIONAL READINGS (OPTIONAL)

- RE4B (Reverse Engineering 4 Beginners)
  - PDF is on Canvas. Read it!
  - Covers Intel, ARM, MIPS assembler with concrete examples
  - Focus isn't on malware, but still a great reference

- Intel architecture manuals (Volumes 1-3)
  - https://software.intel.com/en-us/articles/intel-sdm

- https://en.wikipedia.org/wiki/X86_calling_conventions
- 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/

Georgia Tech

QUESTIONS?

CREATING THE NEXT®