THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

CS330
Assembly Intro

Spring 2022

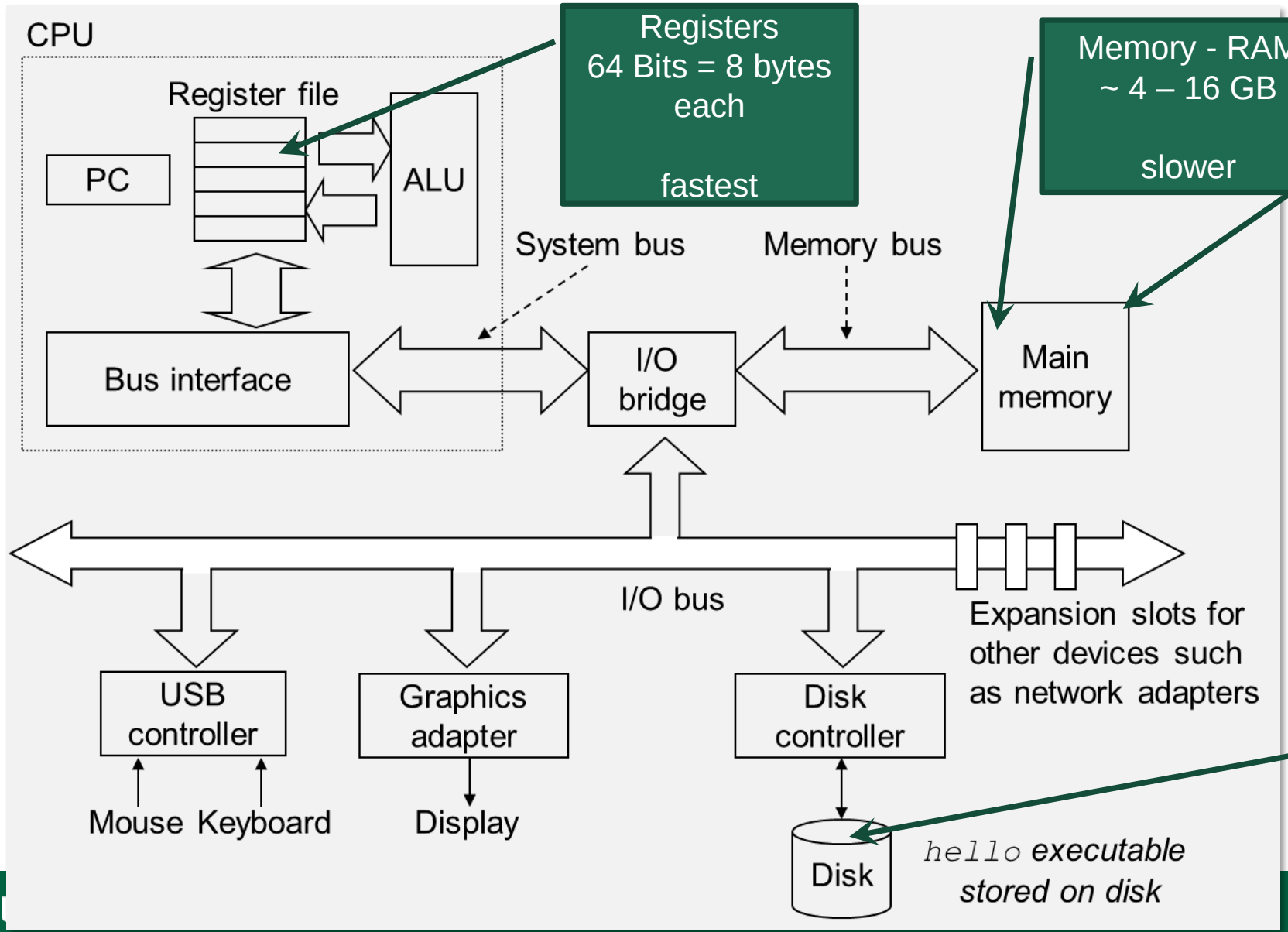
Lab 10

```
5    .text
6    .global main
7    main:
8    # preamble
9    pushq %rbp
10   movq %rsp, %rbp
11
12   # code here
13
14
15
16   # return
17   movq $0, %rax
18   leave
19   ret
```

Helpful Hints

- Assembly is just like procedural programming in other languages, but the building blocks are smaller, and there are a lot less helper tools
  - The process is the same - Break the problem down into smaller pieces
  - Use the tools in our toolkit to solve the problem, build up the tools we need
- It's a manual drive, nothing is automatic -- We need to keep a lot of program state info in our head, or better, write it down, e.g. what's in memory, and where. Or "where did I leave my variable?"
  - A quick sketch of the Registers and Stack can be extremely helpful
- The computer is a shared space (shared resources), following the rules / contract is key to harmony
  - Register management (caller and callee save responsibilities)
  - Stack management: leave it like you found it, check stack alignment
- Code documentation is key. It's not uncommon to have more comment lines than code.
  - `# single line comment`
  - `/* block comment */`
- We'll use x86 AT&T syntax in this course
  - Be careful, there is a lot of Intel syntax (and other syntax, and bad info) on the interwebs

High Level (just for today) Computer Architecture

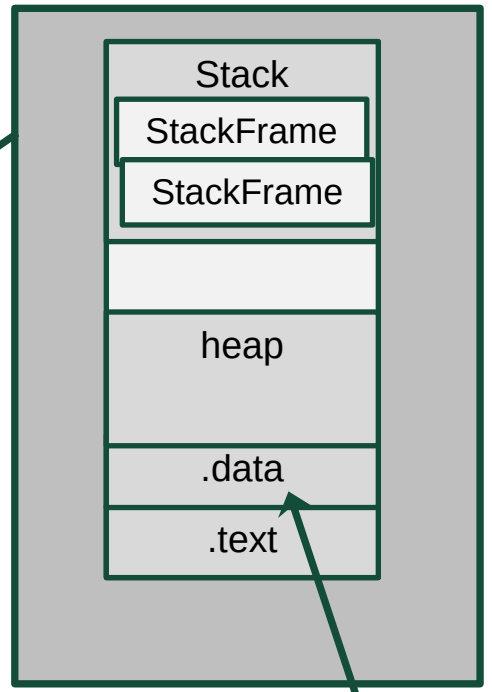


Registers
64 Bits = 8 bytes each

fastest

Memory - RAM
~ 4 – 16 GB

slower



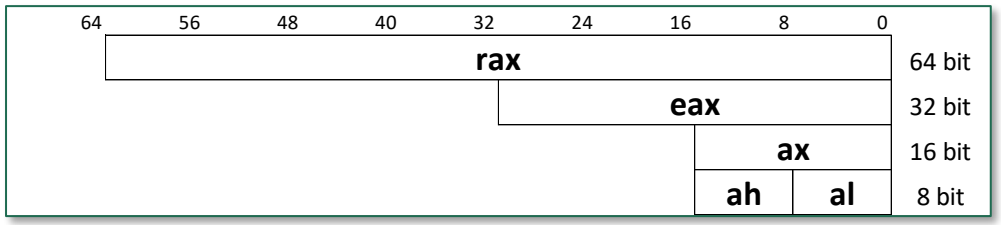
.data is really multiple segments:
.data
.rodata
.bss

Disk – Hard Drive
~ 250 GB – 2 TB

slowest

Registers

- 16 General Purpose Registers
- Register names per AT&T syntax
- Will not use floating, vector registers in this course
- Can also access subsets



Register	Usage	Old Names	Args	Saved by	Preserved Across Function Calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; **1st return register**	accumulator		**Caller**	No
%rbx	callee-saved register; optionally used as base pointer	base		**Callee**	Yes
%rcx	used to pass 4th integer argument to functions	counter, loop counter	4	**Caller**	No
%rdx	used to pass 3rd argument to functions; **2nd return register**	data	3	**Caller**	No
%rsp	stack pointer	stack pointer		**Callee**	Yes
%rbp	callee-aved register, optionally used as frame pointer	base pointer		**Callee**	Yes
%rsi	used to pass 2nd argument to functions	source index	**2**	**Caller**	No
%rdi	used to pass 1st argument to functions	destination index	**1**	**Caller**	No
%r8	used to pass 5th argument to functions		5	**Caller**	No
%r9	used to pass 6th argument to functions		6	**Caller**	No
%r10	temporary register, used for passing a function's static chain pointer			**Caller**	No
%r11	temporary register			**Caller**	No
%r12 - r15	callee-saved registers			**Callee**	Yes

GNU Assembler (AS) Manual: https://sourceware.org/binutils/docs/as/index.html#SEC_Contents

Contracts we'll need to honor

1. Which registers to use (and in which order) to pass arguments into functions (rdi, rsi, rdx, rcx, r8, r9), and which register holds the return value (rax)
2. The caller-callee saved registers – or which registers remain unchanged across function calls
3. Stack management – or "leave it like we found it"

Register	Usage	Old Names	Args	Saved by	Preserved Across Function Calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; **1st return register**	accumulator		**Caller**	No
%rbx	callee-saved register; optionally used as base pointer	base		**Callee**	Yes
%rcx	used to pass 4th integer argument to functions	counter, loop counter	4	**Caller**	No
%rdx	used to pass 3rd argument to functions; **2nd return register**	data	3	**Caller**	No
%rsp	stack pointer	stack pointer		**Callee**	Yes
%rbp	callee-aved register, optionally used as frame pointer	base pointer		**Callee**	Yes
%rsi	used to pass 2nd argument to functions	source index	**2**	**Caller**	No
%rdi	used to pass 1st argument to functions	destination index	**1**	**Caller**	No
%r8	used to pass 5th argument to functions		5	**Caller**	No
%r9	used to pass 6th argument to functions		6	**Caller**	No
%r10	temporary register, used for passing a function's static chain pointer			**Caller**	No
%r11	temporary register			**Caller**	No
%r12 - r15	callee-saved registers			**Callee**	Yes

eflags
(and how to view registers)

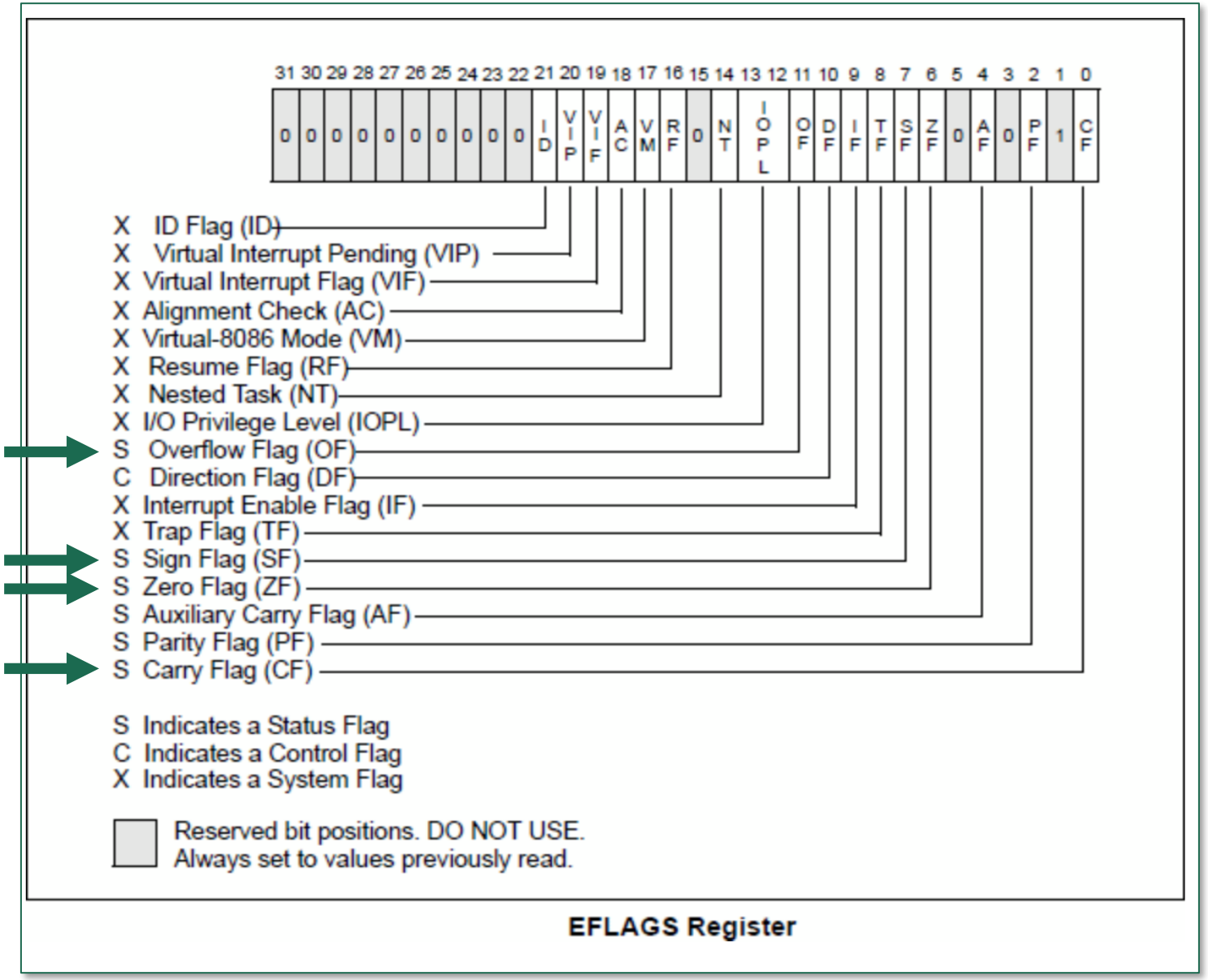
- In GDB
  (i)nfo (r)egisters eflags

```
(gdb) info registers eflags
eflags         0x202    [ IF ]
```

- To show all general purpose registers, including %rip (instruction pointer), eflags
  (i)nfo (r)egisters all

  or individually via
  (i)nfo (r)egisters $<name>
  e.g. (i)nfo (r)egisters $rax

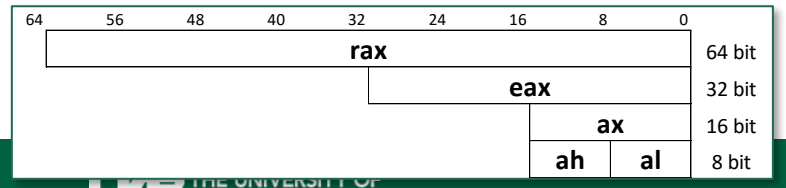
  or
  tui reg general



How to read / interpret the syntax

- Typical AT&T mnemonics use three letter instructions with a one letter suffix to represent the size

Suffix		
b	byte	1 byte
w	word	2 bytes
l	doubleword	4 bytes
q	quadword	8 bytes



Instruction		Effect	Description	pg
Data Movement				
mov	S, D	$D \leftarrow S$	Move source to destination (movslq, sign extend l to q, pg 222)	183
push	S	$R[\%rsp] \leftarrow R[\%rsp] - 8$ $M[R[\%rsp]] \leftarrow S$	push source onto stack	189
pop	D	D $\leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	pop top of stack into destination	189
Arithmetic				
lea	S, D	D $\leftarrow \&S$	load effective address	191
add	S, D	D $\leftarrow D + S$	add	192
sub	S, D	D $\leftarrow D - S$	subtract	192
mul	S, D	D $\leftarrow D * S$	multiply	192
imulq	S	R[%rdx]:R[%rax] $\leftarrow S * R[\%rax]$	multiply (2 64 bit numbers)	198
xor	S, D	D $\leftarrow D \hat{} S$	exclusive-or	192
cqto		R[%rdx]:R[%rax] $\leftarrow$ SignExtend($R[\%rax]$)		
idivq	S	$R[\%rdx]$ $\leftarrow$ R[%rdx]:R[%rax] mod S $R[\%rax]$ $\leftarrow$ R[%rdx]:R[%rax] / S	signed divide	198
Control				
cmp	S_1, S_2	S_2 - S_1	compare	202
jmp	label		direct jump	205
jmp	*Operand		indirect jump	205
je	label		jump if equal / zero (Zero Flag set)	205

S = Source, D = Destination

Operands take one of these three forms

1. Immediate / Literal: $4
2. Register: %rax
3. Memory

Type	From	Operand Value	Name
Immediate	$Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	M[Imm]	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + (R[r_i] * s)]$	Scaled Indexed

(see Book, pg 181 for more)

- Imm refers to a constant value, e.g. 0x8048d8e, 48
- r_a refers to a register
- $R[r_a]$ refers to the value stored in register r_a
- M[x] refers to the value stored at memory address x

Note: can't move (mov) from Memory to Memory

More min requirements (potentially)

- You may also need
  - Preamble
  - Return

  - To compile with special flags:
    - -fno-pie -no-pie

```
5   .text
6   .global main
7   main:
8   # preamble
9   pushq %rbp
10  movq %rsp, %rbp
11
12  # code here
13
14
15
16  # return
17  movq $0, %rax
18  leave
19  ret
```

Recommend you create a template that includes the minimums

Exercise to work through

- Write an assembly language program to add two numbers.
- We will help you through this!
- Let's start with adding a+b = c, where
  - a == 2
  - b == 3
- Initially:
  - no printing
  - no user input, just hard code the numbers, e.g. "$2"
  - we'll add these capabilities later