THE UNIVERSITY OF
ALABAMA AT BIRMINGHAM

CS330
Arrays

Spring 2022

```
.section .data      # start of data section
# === global, static variables here ===

.section   .rodata    # start of read-only data section
# === constants here ===

.text                 # start of text / code
.global main          # tells computer we're starting at main

# === functions here ===

main:                 # start of main, required
# preamble
pushq %rbp
movq %rsp, %rbp

# code here


# return 0
movq $0, %rax         # move 0 into rax to return
leave                 # undo preamble
ret

```

Lab 12

Declaring and Initializing an Array

- Arrays can be declared and initialized in the .data section by listing the values:
  <label>: <element size> <list elements separated by commas>
  myArr: .quad 1, 2, 3
- Declares three 8-byte values, initialized to 1, 2, and 3
- The value at location myArr + 16 will be 3 since quad is 8 bytes in length
  - Recall Pointer Arithmetic
- Uninitialized arrays are defined using the .space/.skip directives:

```
<label>: .skip <number of bytes> [element value]
scores: .skip 40        # arrays with 40 bytes of storage
A: .skip 40, 0          # makes 40 bytes of 0s (4 bytes each)
A: .skip 40, 0x00       # makes 40 bytes of 0s (8 bytes each)
```

One warning, and Addressing Arrays

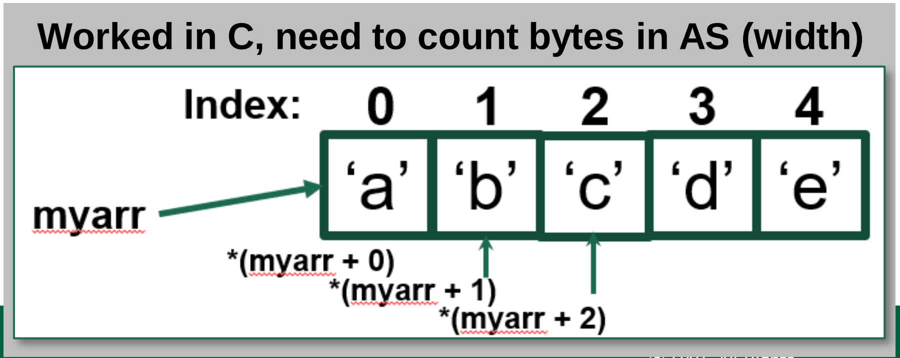
- A common fault when dealing with arrays is not indexing properly
- Unlike C, in Assembly we have to do the math ourselves, Example:
  - Quads are 8 bytes for each element
  - When we want to increase the array index by 1, we must increase the address by 8
- One approach to read from an array:
  movq (%arr_pointer, %index, width), %destination

```
movq (%rax, %rbx, 8), %rdx
```

  so we're indexing the array at: $\text{rax} + 8 * \text{rbx}$

- And to write to an array

```
movq %rbx, (%rax, %rbx, 8)
```



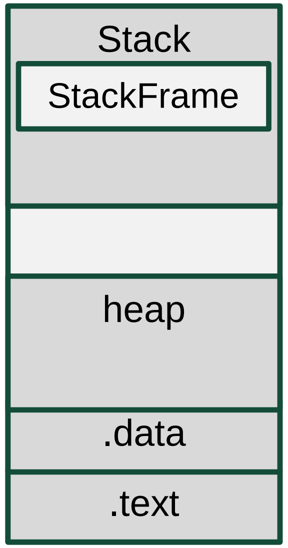
Detail on the mov syntax + examples

- General form: movq offset(%source, %index, WIDTH), %destination
  - Where offset is an int which is summed into the pointer created at the end (usually used with struct, pg 265)
  - Source is a register which we get the data from;   it must be a pointer to an array.
  - Index is a register containing an integer that corresponds to a value in the array like any index does.
  - WIDTH (aka Scale Factor) is multiplied by index to get the byte index of each value. So its 4 for 32 bit value array and 8 for 64 bit
  - == (WIDTH * index) + source + offset
- Some examples of mov instructions using address computations are:
  - movq (%rbx), %rax   #Load 8 bytes from the memory address in RBX into RAX.
  - movq %rbx, var(,1)   #Move the contents of RBX into the 8 bytes at memory address var. (Note, var is a 32-bit constant).
  - movq -4(%rsi), %rax   #Move 8 bytes at memory address and 4 bytes before RSI into RAX.
  - movq %cl, (%rsi,%rax,1)   #Move the contents of CL into the byte at address RSI+RAX.
  - movq (%rsi,%rbx,4), %rdx   #Move the 8 bytes of data at address RSI+4*RBX into RDX.

➢ You won't necessarily use all of these! The simpler statements from the previous two slides are sufficient for our purpose

Two other ways to create arrays / variables

- (a) Recommend the method discussed in the previous charts, but there are two other ways:
- (b) Local Variable on the Stack
  - See first Assembly lab where Compiler created the Assembly code
  - Adjust rsp (stack pointer), use pointer arithmetic based on rbp (base pointer)
- (c) On the heap
  - Using malloc
  - Can also use calloc



Exercise to work – submit for attendance

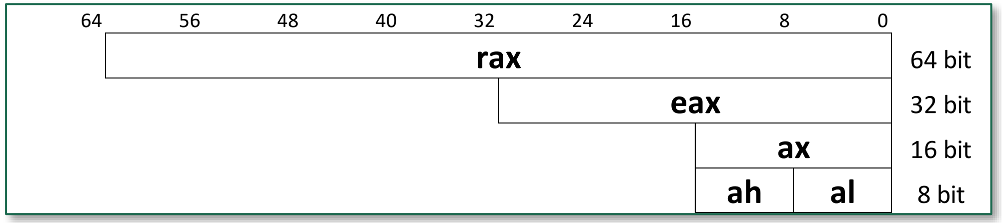
- You can work in teams of 2
  - But everyone needs to submit to Canvas
- Write an assembly language program to sum all the elements of an array:
- {1, 2, 3, 4, 5}
  - Start with hard coding the array
  - If time permits, take user input

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

Reference

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

eflags
(and how to view registers)

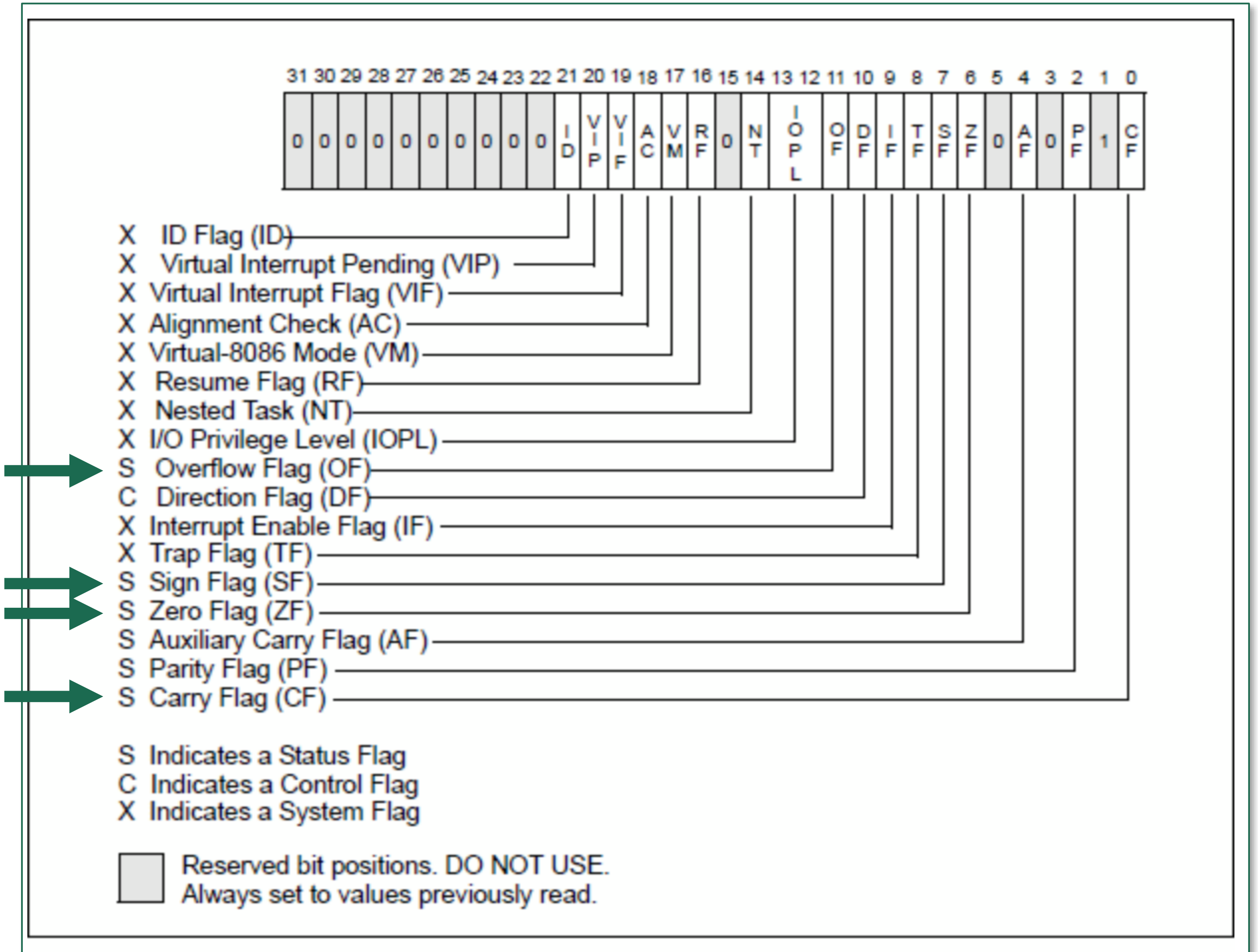
- In GDB
  (i)nfo (r)egisters eflags

```
(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

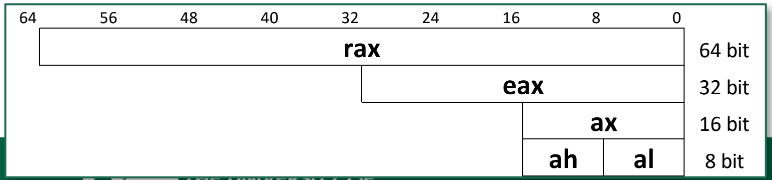


EFLAGS Register

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	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
imul 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 $\leftarrow D$^S	exclusive-or	192
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

Example assembly Instructions

Instruction		Effect	Description
leaq	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \verb!^! S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_{A} k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_{L} k$	Logical right shift

All of the instructions here are used for some kind of mathematical operation.
They show you the name of the instruction as it will be written in your code (but without the size-you may need to add the size suffix, such as q), and the order for the operands. S is Source, D is Destination.

Instruction		Effect	Description
imulq	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Signed full multiply
mulq	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Unsigned full multiply
cqto		R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
idivq	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div$ S	Signed divide
divq	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div$ S	Unsigned divide

`**cqto**` has the purpose of sign-extending an integer in the %rax register to all of %rdx:%rax. For example, if %rax were –1 (11111111...), %rdx:%rax would be the same, but for all 128 bits.

The instructions here are different kinds of Multiplication and Division, except for cqto, which has a special purpose.
You will need to use Signed Multiplication and Division to see the correct results for all inputs on your homework, but positive inputs will behave the same way for both.

Carefully observe the "effect" column: Like Booth's algorithm, the multiplication result takes up twice as much space as the operands took up. Since we cannot know the operands are smaller than the size of the specified registers, the result is always stored across two whole registers.

`cltq` has the purpose of sign-extending an integer in the %eax register to all of %rax. For example, if %eax were –1 (11111111...), %rax would be the same, but for all 64 bits.

Condition Code – Explicit Setting with Compare

- Example: $\mathrm{cmpq\ src1,\ src2}$
- Same as computing src2 **–** src1 without setting a destination
  - Result is **not** stored, but flags are still set
- **CF Set ->** if carry occurs from most significant bit (leftmost)
- **ZF Set ->** if Src1 == Src2
- **OF Set ->** if overflow occurs
- **SF Set ->** if Src2 – Src1 < 0 (negative)

Condition Code – Explicit Setting with Test

- Example: $\mathrm{testq\ src1,\ src2}$
- Same as computing src1 **&** src2 without setting a destination
  - Result is **not** stored, but flags are still set
  - Allows conditional statements on Boolean expressions
- **ZF Set ->** if Src1 & Src2 == 0
- **SF Set ->** if Src1 & Src2 < 0 (negative)

Jump Commands

Syntax:

- **Direct**:

  <jX> <label>

- **Indirect**:

  <jX> <*operand>

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Functions

- Each function begins with a **label**:
  - A label is a name (capitalization matters) followed by a colon ":"
  - e.g. myFunction:
- Each function ends with a return **ret**
- Functions should be placed in the .text section, but above main:
- Don't forget our contract to pass arguments in the appropriate registers
  - Pass in arguments in the order: rdi, rsi, etc
  - Return values in rax
- Don't forget our contract to save appropriate registers
  - Ideally the caller-saved registers are the responsibility of the caller (e.g. main:), and the callee-saved registers are the responsibility of the callee (e.g. myFunction:)
  - Practically, since we're writing both the caller (main:) and the callee (myFunction:) functions, we can do whatever we want
    - And usually we don't waste resources saving registers unnecessarily, just the ones we need / use
- Functions can call other functions …
- **Be sure to document**, describe: what the function does, what it takes as arguments, what it returns

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