THE UNIVERSITY OF
ALABAMA AT BIRMINGHAM

CS330
Arithmetic Ops & Functions

Spring 2022

Lab 10

```
.text
.global main
main:
# preamble
pushq %rbp
movq %rsp, %rbp

# code here



# return
movq $0, %rax
leave
ret
```

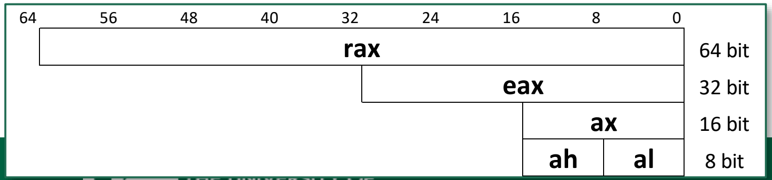
Arithmetic Operations

- Most Assembly arithmetic mnemonics operate as you would expect
- Multiplication and Division are a little unique
  - Multiplication: returns the result across two registers
    - Remember Booth's algorithm for multiplication? Two numbers multiplied together may require twice as many bits as either number.
  - Division:
    - Starts with a number across two registers (use cqto): rax sign extended into rdx
    - Then returns the answer in two registers, rax for the integer division, and rdx for the remainder
- For all the arithmetic operations, note what happens to the eflag register
  - Specifically the Overflow, Sign, Carry, and Zero flags
  - We'll talk more about these next week

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($R[\%rax]$)	Convert to oct word (sign extend)	198
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

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	$D \leftarrow D \,\hat{}\, S$	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`	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word
`idivq` S	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S$; $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
`divq` S	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S$; $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide

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.

`**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.

`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.

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

Exercise to work through

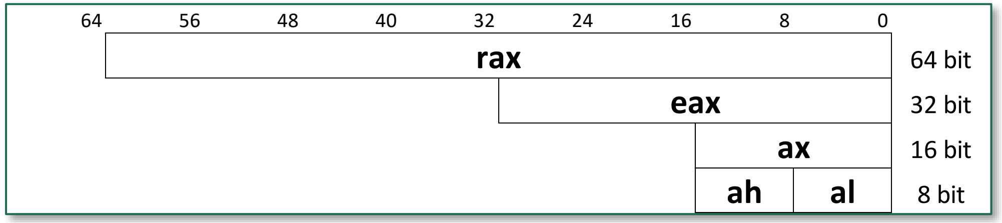
- You can work in teams of 2
  - But everyone needs to submit to Canvas
- Write an assembly language program to:
  - convert Fahrenheit to Celsius $C = (F - 32) * 5 / 9$
  - convert Celsius to Fahrenheit $F = (C * 9 / 5) + 32$
- Ignore anything after the decimal point (no floating point registers)
- Be sure to use at least one function
- You can take user input, or hardcode the values
- Print the result

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)

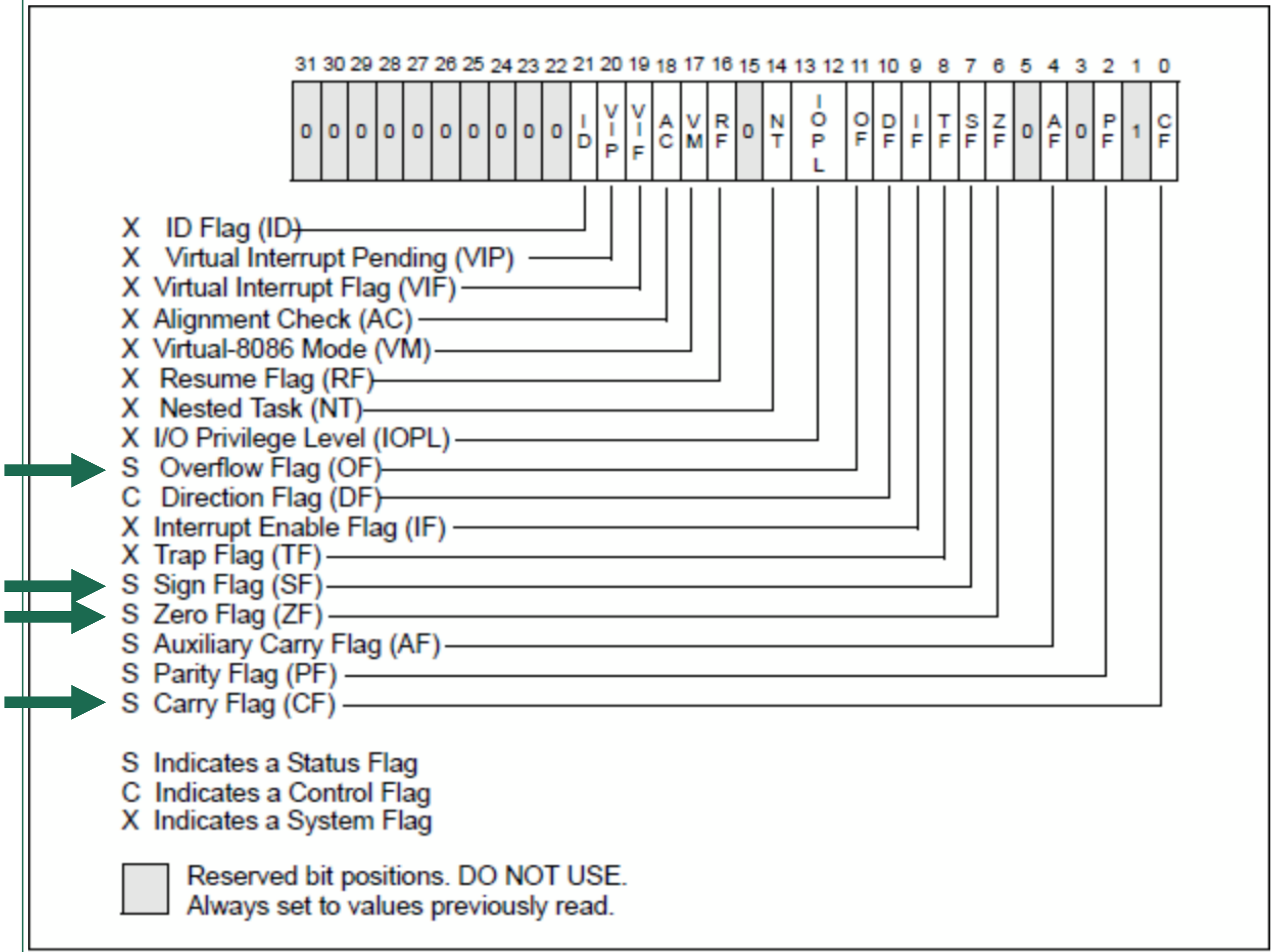
- In GDB
  (i)nfo (r)egisters eflags

```
(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

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

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