# Assembly Language II: Addressing Modes & Control Flow

- Learning Objectives
  - Interpret different modes of addressing in x86 assembly
  - Move data seamlessly between registers and memory
  - Map data structures in C to different addressing modes in assembly.
  - Interpret the Intel flag values.
  - Interpret cmp, test, and jump instructions correctly.
  - Map common C control flow constructs into assembly language.
  - Recognize common C control flow patterns in assembly language
  - But first ...

## Note 1: Passing structs

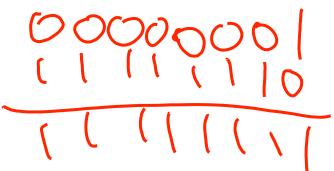
- We know that the first six arguments to a function go in registers: rdi, rsi, rdx, rcx, r8, and r9.
- At the end of class, someone asked what happens if you pass structs as arguments – we're glad you asked!
- 1. If the struct has fields that all fit in registers, the compiler will pass them in registers.
- 2. If the struct does not fit in registers, it is placed on the stack.
- 3. Other parameters are still placed in registers.

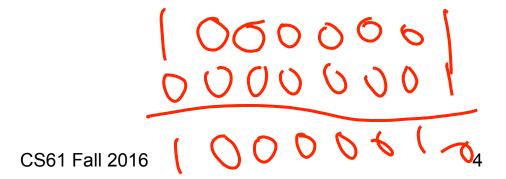
## Note 2: Binary Arithmetic

- We've talked about how we represent numbers in a computer in binary:
  - The number decimal 10 can be represented by combining powers of 2:
  - $8 + 2 = 2^3 + 2^2 = 10$
  - So, in binary: 1010
- But how do we represent -10?

## (cont) 2's Complement Arithmetic

- We represent —n by flipping all the bits in n (~n) and adding 1 to it:
  - $\bullet \quad -n = (\sim n) + 1$
- Example (in 8 bits):
  - N = 5; 5 = 00000101
  - $\sim N = 111111010$
  - $\sim N + 1 = 111111011 = 0xfb$





## (cont) Implications of 2's Complement

- Adding signed and unsigned numbers is identical!
  - That's why you could generate some identical assembly last class using signed and unsigned C types.
- Negative numbers will always have a 1 in the high order bit.
- Positive numbers will always have a 0 in the high order bit.

#### Confused? Check out:

https://mix.office.com/watch/mgjeezhtni45?lcid=

## **Accessing Memory**

- So far we've seen that you can move data around using mov instructions:
  - mov %rax, %edi
- Now it's time to dig into that in a bit more detail and find out how to access data in memory.
- Fundamentally there are three ways to access memory:
  - Via labels (names)
  - Via registers (including %rip-relative addressing)
  - Via constants

## Register based addressing

- Recall that simply using %reg (e.g., %rax) accesses the value stored in the register.
- Indirect addressing: Addressing that uses the contents of a register to produce an address:
  - (%reg) treats the value in reg as an address and refers to the value stored at that address.
    - Example: (%rsp) Refers to the value stored on the stack at the location referenced by the stack pointer.
  - N(%reg) refers to the data at the address produced by adding N to the contents of reg.
    - Example 4 (%rsp) refers to the value stored at the address 4 greater than the stack pointer.

• Sum.c (unoptimized)

## Indirect Addressing Continued

- Indirect addressing: Addressing that uses the contents of two registers to produce an address:
  - (%reg1, %reg2) Refers to the value stored at the address formed by adding the contents of reg1 and reg2.
  - Example: (%rax, %rdx): If %rax contains the starting address of an array of chars, and %rdx is the index, this accesses the value at that index.
- Extended Indirect addressing:
  - (%reg1, %reg2, S) refers to the value stored at the address formed by adding "%reg2 \* S" to %reg1. S must be one of 1, 2, 4, 8 (when missing, 1 is implied).
  - Example: (%rax, %rdx, 4) Now the example above works on an array of integers.

## Complete form of Indirect Addressing

Compute address as follows:

$$N + reg1 + (reg2 * S)$$

- Another handy instruction: lea[lq] = load effective address: computes an address and places the result in a register.
  - Used to put the pointer to something in a register.
  - Also, often a fast way to perform addition.

- Sum1.c (optimized)
- Cindex.c (character array access)

- Cindex2.c
- Cindex3.c

• lindex.c

### Other modes: Immediate

- Immediate:
  - The syntax \$NNN refers to the number NNN.
  - You'll see it used in (at least) two ways:

```
movl $63, %eax # Moves the value 63 into a register
movl $Label, %eax # Move the address to which Label # corresponds into a register.
```

 Note: When you use labels as the destination of a jumpclass instruction, you do not use the \$, e.g., jmp Label

## Other modes: RIP-relative

#### RIP-relative:

- Recall that %rip is the instruction pointer.
- RIP-relative addressing is a special case of indirect addressing, using the instruction pointer as the register.
- Example: you will frequently see global symbols referenced in this manner:

## **Control Flow**

When we write:

What do we really mean?

#### Control Flow

• When we write:

What do we really mean?

#### Control Flow Overview

#### Unconditional:

- Directs the processor to execute at an address other than the next sequential address.
- Examples:
  - jmp (jump)
  - callq (call a function)

#### Conditional:

- Paired instructions:
  - 1. An instruction that sets condition flags (e.g., arithmetic, logical, cmp, test)
  - 2. A conditional jump instruction that changes the sequential execution of a program depending on the state of the condition flags.

## **Condition Flags**

- Special bits stored in a special register: EFLAGS
- SF: Sign Flag
  - The most recent operation yielded a negative value.
  - Equal to MSbit of result; which indicates the sign of a two's complement integer.
  - 0 means result was positive, 1 means negative.
- CF: Carry Flag
  - The most recent operation generated a carry bit out of the MSbit.
  - Indicates overflow when performing unsigned integer arithmetic.
- OF: Overflow Flag
  - The most recent operation caused a 2's complement overflow (either positive or negative).
  - Indicates an overflow when performing signed integer arithmetic.
- ZF: Zero Flag
  - The most recent operation yielded a zero.
- Condition flags are set implicitly by every arithmetic instruction
- Condition flags are set explicitly by comparison and test instructions

## Comparison Instructions

- cmp[bwlq] src1, src2
  - Compares value of src1 and src2
  - *src1*, *src2* can be registers, immediate values, or contents of memory.
  - Computes (src2 src1) without modifying either operand
    - like "subl *src1*, *src2*" without changing *src2*
  - But, sets the condition flags based on the result of the subtraction.
- test[bwlq] src1, src2
  - Like cmp1, but computes (src1 & src2) instead of subtracting them.

## Conditional Jump Instructions

- Used for signed or unsigned operations
  - JE: jump if equal (ZF=1)
  - JNE: jump if not equal (ZF=0)
- Used for signed operations
  - JS: jump if signed/negative (SF = 1)
  - JNS: Jump if not signed/positive (SF = 0)
  - JL: jump if less (SF != OF)
  - JLE: jump if less than or equal to (ZF = 1 or SF != OF)
  - JG: jump if greater than (ZF = 0 and SF = OF)
  - JGE: jump if greater than or equal to (SF = OF)
- Used for unsigned operations
  - JA: jump above (CF = 0 and ZF = 0)
  - JAE: jump above or equal (CF = 0)
  - JB: jump below (CF = 1)
  - JBE: jump below or equal (CF = 1 or ZF = 1)

## **CS Faculty Coffee Chats**

- CS will pay for CS faculty and undergrads to go out for coffee (or other beverage).
- Open to all undergrads, not just CS concentrators
- One or more students can join each coffee chat
- Check my calendar and suggest a time!