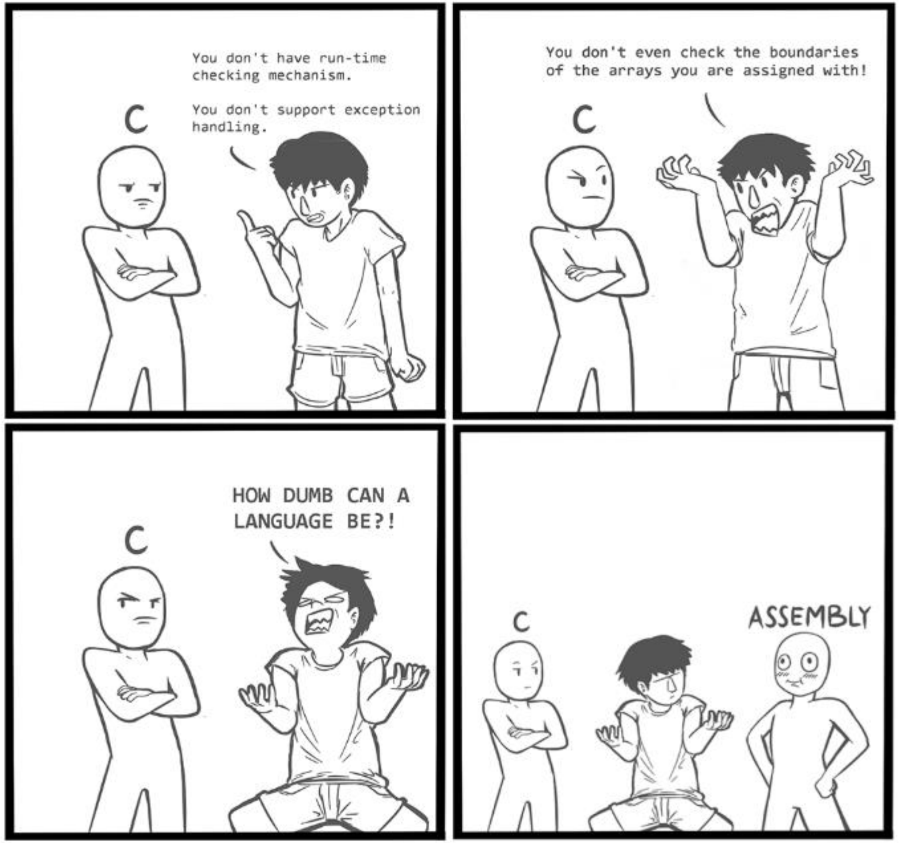
x86-64 Programming II

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- Today:
  - HW6 due (11:59pm)
  - Late deadline for Lab 1a (11:59pm)
  - Quiz 1 out (11:59pm)
    - Reminder: Abbot Elementary policy (see course website for details)
- Monday 7/8:
  - RD9 due (1pm)
  - HW7 due (11:59pm)
- Wednesday 7/10:
  - RD 10 due (1pm)
  - HW8 due (11:59pm)
  - **Lab1b due (11:59pm)**

Aside: C Macros

- Basic syntax is of the form: `#define NAME expression`
- NOT the same as a Java constant, though they're used similarly
  - Performs a *naive copy* before compilation
  - Any time the characters in NAME appear, will be replaced with the characters in expression
  - Expression doesn't have to be a constant
    - Order of operations may get mixed up if you're not careful!
  - Remember that integer constants are *signed ints* by default, need to cast if you plan on using them as anything else
- Useful in Lab 1b
  - Please don't use magic numbers in your code!

Lecture Topics

- **Address Computation Part 2: Electric Boogaloo**
  - **Memory addressing mode examples**
  - **lea instruction**
- Move extension
- Control flow
  - Processor state
  - Condition codes
  - jmp and set instructions

Recap: Memory Addressing Modes

- General format: `D(Rb,Ri,S)` = Mem[Reg(`Rb`) + Reg(`Ri`)*`S` + `D`]
  - `Rb` = base register (any register)
  - `Ri` = index register (any register except `%rsp`)
  - `S` = scale factor (1, 2, 4, 8) - Why these numbers?
  - `D` = displacement value (immediate)
- Can leave any of these out:
  - `D(Rb,Ri)` - `S`=1
  - `(Rb,Ri,S)` - `D`=0
  - `(Rb,Ri)` - `D`=0, `S`=1
  - `(,Ri,S)` - `D`=0, `Rb`=0
  - etc…

Address Computation Examples

%rdx	0xF000
%rcx	0x0100

- 8-bit addresses
- Reminder: `D(Rb,Ri,S)` →Mem[Reg[`Rb`]+Reg[`Ri`]*`S`+`D`]
  - (ignoring memory addressing portion for this exercise, just calculating address)

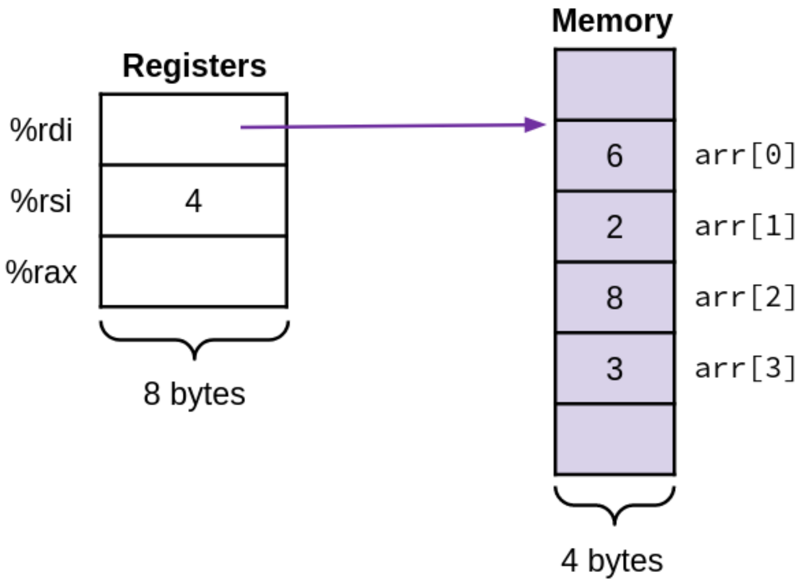
Expression	Address Computation	Address
`0x8(%rdx)`	0x8 + rdx	0xF000
`(%rdx, %rcx)`	rdx + rcx	0xF100
`(%rdx, %rcx, 4)`	rdx + rcx · 4	0xF400
`0x80(, %rcx, 2)`	0x80 + rcx · 2	0x0280

Address Computation Code Example

%rdi	arr
%rsi	len
%eax	Return value

```c
int last_elem(int* arr, long len)
{
    return arr[len-1];
}
```

```
last_elem:
    movl  -4(%rdi,%rsi,4), %eax
    ret
```



Review Question

Which of the following x86-64 instructions correctly calculates **`%rax=9*%rdi`**?

A) `leaq (,%rdi,9), %rax`
B) `movq (,%rdi,9), %rax`
C) `leaq (%rdi,%rdi,8), %rax`
D) `movq (%rdi,%rdi,8), %rax`

A and B ruled out - 9 is invalid. S must be 1, 2, 4, or 8

C and D both compute rdi + rdi*8 = rdi*9, but lea just stores that result into %rax, while mov treats the result as an address, gets data at that location in memory, and stores that into %rax

Address Computation Instruction

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (using memory addressing format), `dst` is a register
  - Sets `dst` to the **address** computed by the `src` expression
    - Does not go to memory! – **it just does math**
  - Example: `leaq (%rdx,%rcx,4), %rax`
- Uses:
  - Computing addresses without a going to memory
    - Ex: assembly equivalent of `p = &x[i]`
  - Computing arithmetic expressions of the form `x+k*i+d` (k must be 1, 2, 4, or 8)
    - Can use to compute anything! Doesn't have to be an address

Example: `lea` vs `mov`

rdx + rcx·4 = 0x100 + 4·4 = 0x110

get data at address 0x110

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

store data at address rdx points to

store value of %rdx

Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	0x1

Memory

Memory	Word Address
0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

note: leaq (%rdx), %rdi == movq %rdx, %rdi

lea Arithmetic Example

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    leaq (%rsi,%rsi,2), %rdx
    salq $4, %rdx
    leaq 4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

%rdi	x
%rsi	y
%rax	t1, t2, rval
%rcx	t5
%rdx	z, t4

- Notice `imulq` is only used once!
  - Uses `leaq` and `salq` instead

lea Arithmetic Example (pt 2)

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    leaq (%rsi,%rsi,2), %rdx
    salq $4, %rdx
    leaq 4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

%rdi	x
%rsi	y
%rax	**t1**
%rcx	
%rdx	z

rax= rdi + rsi

t1 = x + y

lea Arithmetic Example (pt 3)

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    leaq (%rsi,%rsi,2), %rdx
    salq $4, %rdx
    leaq 4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

%rdi	x
%rsi	y
%rax	t1, **t2**
%rcx	
%rdx	z

rax = rdx + rax
t2 = z + t1

lea Arithmetic Example (pt 4)

reminder: x<<4 = $x \cdot 2^4 = x \cdot 16$

```
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

skips line 3, we'll come back to it!

```
arith:
    leaq  (%rdi,%rsi), %rax
    addq  %rdx, %rax
    leaq  (%rsi,%rsi,2), %rdx
    salq  $4, %rdx
    leaq  4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

%rdi	x
%rsi	y
%rax	t1, **t2**
%rcx	
%rdx	z, **t4**

1st line! rdx = rsi + rsi · 2
= y + y·2 = y·3

2nd line rdx = rdx · 16
t4 = y·3·16 = y·48

compiler chose lea + shift because it's faster than a single multiply

lea Arithmetic Example (pt 5)

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

back to line 3

```
arith:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    leaq (%rsi,%rsi,2), %rdx
    salq $4, %rdx
    leaq 4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

2 C lines in 1 x86 instruction

%rdi	x
%rsi	y
%rax	t1, **t2**
%rcx	**t5**
%rdx	z, **t4**

rcx = 4+rdi + rdx

t5 = 4+ x + t4 (4+ x = t3)

lea Arithmetic Example (pt 6)

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq  (%rdi,%rsi), %rax
    addq  %rdx, %rax
    leaq  (%rsi,%rsi,2), %rdx
    salq  $4, %rdx
    leaq  4(%rdi,%rdx), %rcx
    imulq %rcx, %rax
    ret
```

%rdi	x
%rsi	y
%rax	t1, t2, **rval**
%rcx	**t5**
%rdx	z, **t4**

rax = rax · rcx
rval = t2 · t5

Lecture Topics

- Address Computation Part 2: Electric Boogaloo
  - Memory addressing mode examples
  - lea instruction
- **Move extension**
- Control flow
  - Processor state
  - Condition codes
  - `jmp` and `set` instructions

Move extension: `movz` and `movs`

- `movz__ src, dst` = Move with zero extension
- `movs__ src, dst` = Move with sign extension
  - `src` can me memory or register. `dst` *must* be a register
  - Copy from a *smaller* source value to a *larger* destination
    - Takes two width specifiers: 1st is source, 2nd is destination
  - Fill remaining bits of dest with zero (mov**z**) or sign bit (mov**s**)

<u>Example</u>:

`movzbw %al, %bx`

%rax	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0xFF
%rbx	0x??	0x??	0x??	0x??	0x??	0x??	0x00	0xFF

lowest B of %rax
lowest 2B of %rbx
zero-extend
1B src
2B dst
copy
extend

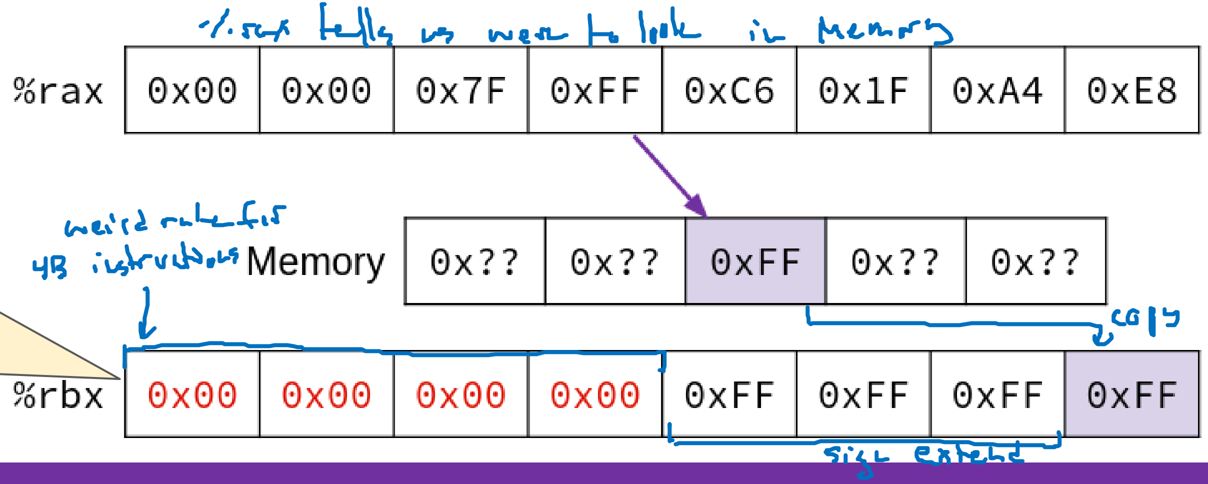
Move extension: `movz` and `movs` (pt 2)

- `movz__ src, dst =` Move with zero extension
- `movs__ src, dst =` Move with sign extension

Example:

```
movsbl (%rax), %ebx
```

In x86_64, *any instruction* that writes a 4-byte value to a register **automatically sets the highest 4 bytes of that register to 0!**



Lecture Topics

- Address Computation Part 2: Electric Boogaloo
  - Memory addressing mode examples
  - lea instruction
- Move extension
- **Control flow**
  - **Processor state**
  - **Condition codes**
  - **jmp and set instructions**

Control Flow

- How do we alter the flow of execution?
  - ex: if/else ladders, loops, etc.

%rdi	x
%rsi	y
%rax	Return value

Example:

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

```
max:
        ???
        movq %rdi, %rax
        ???
        ???
        movq %rsi, %rax
        ???
        ret
```

Control Flow (pt 2)

- How do we alter the flow of execution?
  - ex: if/else ladders, loops, etc.

%rdi	x
%rsi	y
%rax	Return value

Example:

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

Conditional jump

Unconditional jump

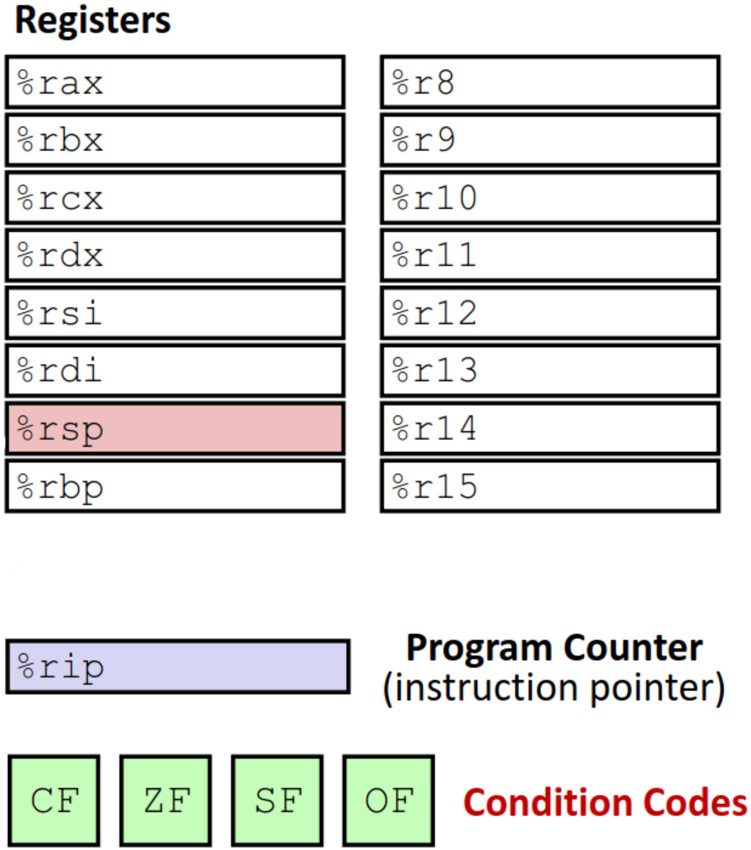
```
max:
    if x<y, jump to else
    movq %rdi, %rax
    jump to done
else:
    movq %rsi, %rax
done:
    ret
```

Conditionals and Control Flow

- Conditional jump
  - Jump to somewhere else if some condition is true, otherwise execute next instruction in order
- Unconditional jump
  - Always jump when you get to this instruction
- Together, they can implement most control flow constructs in high-level languages:
  - `if (condition) {...} else {...}`
  - `while (condition) {...}`
  - `for (initialization; condition; iterative) {...}`
  - `switch {...}`

x86 Processor State (partial)

- Program data
  - General-purpose registers (ex: `%rax`)
- Next instruction
  - **Instruction pointer** (`%rip`)
- Status of recent operations
  - **Condition codes**: single-bit registers (CF, ZF, SF, OF)
- We'll talk about more in the future!



Condition Codes

- Keep track of information about the recent instructions
  - **Carry Flag**: CF
    - 1 if carry-out from MSB (unsigned overflow)
  - **Zero Flag**:
    - 1 if result is 0
  - **Sign Flag**: SF
    - 1 if result < 0 when interpreted as signed (i.e. MSB is 1)
  - **Overflow Flag**: OF
    - 1 if signed overflow

Condition Codes: Implicit Setting

- Any *arithmetic or logical* instruction implicitly sets condition codes
  - (Think of it as a side effect)
  - `lea` doesn't count!

Ex: `addb $1, %ax` (`%ax = 0xFF`)

Result = `0x00`

CF=1 **ZF**=1 **SF**=0 **OF**=0

- Reminder: the hardware doesn't keep track of data types!
  - i.e. it will still set the sign and zero flags, even if you're interpreting the data as unsigned!

Condition Codes: Explicit Setting

- **Compare** instruction: `cmp_ src1, src2`
  - Sets flags based on `src2-src1`, but **doesn't store**
  - **CF** = 1 if unsigned overflow (good for unsigned comparison)
  - **ZF** = 1 if `src1==src2`
  - **SF** = 1 if `(src2-src1)<0` (good for signed comparison)
  - **OF** = 1 if signed overflow

Condition Codes: Explicit Setting (pt 2)

- **Test** instruction: `test_ src1, src2`
  - Sets flags based on `src1&src2`, but **doesn't store**
  - Can't have carry out (**CF**) or overflow(**OF**)
  - **ZF** = 1 if `src1&src2==0`
  - **SF** = 1 if `src2&src1<0` (signed, i.e. MSB=1)

in 2's compl

MSB =1 means negative

Polling Question: Condition Code Setting

Assuming that %al = 0x80 and %bl = 0x81, which flags (CF, ZF, SF, OF) are set when we execute **`cmpb %al, %bl`**?

(Hint: instead of computing `%bl - %al`, compute `%bl + (-%al)`)

al = 0b 1000 0000 ← oops!

-al = ~al +1 = 0b 0111 1111 +1 = 0b1000 0000

bl = 0b 1000 0001

1 ← carry bit

bl + (-al) = 0b 1000 0001 + 0b 1000 0000 = 0000 0001

CF = 1 because of carry bit

ZF = 0 result != 0

SF = 0 result not negative

OF = 0 tricky! looks like overflow based on signs of operands and result, but 0x81 - 0x80 = 1, which is our answer!

Using Condition Codes: Setting

- General format: `set* dst`
  - Sets lowest byte of `dst` to 1 if the condition is met, 0 otherwise
  - Does not alter the remaining 7 bytes

dst should be a 1-byte register or memory location

Don't bother memorizing, just use the chart.

don't need to know these!

Instruction	Condition	Description
sete dst	ZF	Equal (to zero)
setne dst	~ZF	Not Equal (to zero)
sets dst	SF	Negative
setns dst	~SF	Nonnegative
setg dst	~(SF^OF)&~ZF	Greater (signed)
setge dst	~(SF^OF)	Greater or Equal (signed)
setl dst	(SF^OF)	Less than (signed)
setle dst	(SF^OF)|ZF	Less or Equal (signed)
seta dst	~CF&~ZF	Above (unsigned “>”)
setb dst	CF	Below (unsigned “<”)

Reading Condition Codes

`%edi`	x
`%esi`	y
`%eax`	Return value

- Use `set*` instructions
  - Remember: only alters the lowest byte of a register
    - Commonly seen with `movz` instructions to fill out remaining bytes

Example:

set %al to 1 if x-y>0 → x>y

set flags based on edi - esi = x - y

(comments)

```
int gt(int x, int y)
{
    return x > y;
}
```

```
gt:
    cmpl %esi, %edi   # Compare x and y
    setg %al          # Set when >
    movzbl %al, %eax  # Zero rest of %rax
    ret
```

extend to 4B int

Using Condition Codes: Jumping

- General format: `j* target`
  - Sets `%rip` to `target` if the condition is met
  - `jmp` is **unconditional** - always jumps
- Used to create if/else statements, loops, etc.
  - More info next lecture

rip points to next instruction to execute

Don't bother memorizing, just use the chart.

Instruction	Condition	Description
jmp target	1	Unconditional
je target	ZF	Equal (to zero)
jne target	~ZF	Not Equal (to zero)
js target	SF	Negative
jns target	~SF	Nonnegative
jg target	~(SF^OF)&~ZF	Greater (signed)
jge target	~(SF^OF)	Greater or Equal (signed)
jl target	(SF^OF)	Less than (signed)
jle target	(SF^OF)|ZF	Less or Equal (signed)
ja target	~CF&~ZF	Above (unsigned “>”)
jb target	CF	Below (unsigned “<”)

Summary:

- **Memory addressing modes** allow us to compute an address in a single instruction
  - `D(Rb, Ri, S)` -> `Rb + Ri*S + D`
  - **`lea`** just performs the computation and stores the result
  - All instructions treat the result as an *address*, then get data at that address in memory
- **Condition codes** store information about previous operation
  - Implicitly set by any arithmetic or logical instruction
  - Explicitly set by **`cmp`** and **`test`** instructions
  - Used with **`jump`** and **`set`** instructions