

x86-64 Programming II

CSE 351 Summer 2020

Instructor:

Mara Kirdani-Ryan

Teaching Assistants:

Kashish Aggarwal

Nick Durand

Colton Jobs

Tim Mandzyuk



<http://xkcd.com/99/>



Gentle, Loving Reminders

- Lab 1b due tonight! 8pm
 - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`
 - Can still use late days until 7/12
- hw6, hw7 due tonight! 8pm
- Unit Summary 1 due Monday (7/12) – 8pm
 - Can still use late days until 7/14
- hw8 due Monday (7/12) – 8pm

Gentle, Loving Reminders

- Lab1a grades released today or Monday
 - Talk to us about any questions you have!
 - Regrades open 24 hours after grades are released, stay open usually for about a week
- Lab 2 released later today!
 - Debugging x86-64 assembly using gdb

Guest Lectures incoming!

Guest Lectures

- My partner's getting surgery on Wednesday 7/14
 - Justin Hsia (351 instructor in Autumn) is filling in
 - I probably won't respond to anything that day
- Some TAs are going to step in and lecture!
 - More details as we go forward

Learning Objectives

Understanding this lecture means you can...

- Explain the difference between `mov` and `lea`
- Explain condition codes!
 - What are they? Where are they stored?
 - How are they used to implement control flow?
 - How are they modified by `cmp`, `test`, `set` and arithmetic operations?
 - How can they be read with `movz` and `movs`?
- Translate assembly functions with arithmetic and control flow to C, and vice versa
- Explain the difference between RISC and CISC architectures, from an ideological perspective

x86-64 Introduction

- Data transfer instruction (`mov`)
- Arithmetic operations
- **Memory addressing modes**
- **Address computation instruction (`lea`)**

A note on register widths & Overflow

- Registers are only so wide (64b/8B in x86-64)
- This is the physical limitation that we talked about with integers & floats!
 - “We only have so many bits” → registers are only so wide

Memory Addressing Modes: Basic

- **Indirect:** $(R) \quad \text{Mem}[\text{Reg}[R]]$
 - Data in register R specifies the memory address
 - Like pointer dereference in C
 - Example: `movq (%rcx), %rax`
- **Displacement:** $D (R) \quad \text{Mem}[\text{Reg}[R]+D]$
 - Data in register R specifies the *start* of some memory region
 - Constant displacement D specifies the offset from that address
 - Example: `movq 8(%rbp), %rdx`

Complete Memory Addressing Modes

○ General:

- $D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$
 - Rb : Base register (any register)
 - Ri : Index register (any register except `%rsp`)
 - S : Scale factor (1, 2, 4, 8) – *why these numbers?*
 - D : Constant displacement value (a.k.a. immediate)

○ Special cases (see CSPP Figure 3.3 on p.181)

- $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)$
- $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S] \quad (D=0)$
- $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)$
- $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] * S] \quad (Rb=0, D=0)$

Address Computation Examples





<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$D(Rb, Ri, S) \rightarrow$

$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$

Mix and match expressions to addresses!

<code>0x8(%rdx)</code>
<code>(%rdx,%rcx)</code>
<code>(%rdx,%rcx,4)</code>
<code>0x80(,%rdx,2)</code>

 <code>0x1e080</code>
 <code>0xf400</code>
 <code>0xf008</code>
 <code>0xf100</code>



if you're stuck!

Address Computation Instruction

- `leaq src, dst;` *load effective address*
 - `src` is address expression (in any 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 memory reference
 - e.g. translation of `p = &x[i];`
 - Computing arithmetic expressions of the form $x+k*i+d$
 - Though `k` can only be 1, 2, 4, or 8

Example: lea vs. mov

Registers		Memory	Word Address
%rax		0x400	0x120
%rbx		0xF	0x118
%rcx	0x4	0x8	0x110
%rdx	0x100	0x10	0x108
%rdi		0x1	0x100
%rsi			

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

Example: lea vs. mov

Registers		Memory	Word Address
%rax	0x110	0x400	0x120
%rbx		0xF	0x118
%rcx	0x4	0x8	0x110
%rdx	0x100	0x10	0x108
%rdi		0x1	0x100
%rsi			

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

Example: lea vs. mov

Registers		Memory	Word Address
%rax	0x110	0x400	0x120
%rbx	0x8	0xF	0x118
%rcx	0x4	0x8	0x110
%rdx	0x100	0x10	0x108
%rdi		0x1	0x100
%rsi			

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

Example: lea vs. mov

Registers		Memory	Word Address
%rax	0x110	0x400	0x120
%rbx	0x8	0xF	0x118
%rcx	0x4	0x8	0x110
%rdx	0x100	0x10	0x108
%rdi	0x100	0x1	0x100
%rsi			

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


Example: lea vs. mov

Registers		Memory	Word Address
%rax	0x110	0x400	0x120
%rbx	0x8	0xF	0x118
%rcx	0x4	0x8	0x110
%rdx	0x100	0x10	0x108
%rdi	0x100	0x1	0x100
%rsi	0x1		

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

lea – “It just does math”

Arithmetic Example

```

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

```

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rdx	3 rd argument (z)

Interesting Instructions

- leaq: “address” computation
- salq: shift
- imulq: multiplication
- Only used once!

Arithmetic Example

```

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;
}

```

Register	Use(s)
%rdi	x
%rsi	y
%rdx	z, t4
%rax	t1, t2, rval
%rcx	t5

```

arith:
    leaq    (%rdi,%rsi), %rax    # rax/t1    = x + y
    addq    %rdx, %rax          # rax/t2    = t1 + z
    leaq    (%rsi,%rsi,2), %rdx  # rdx       = 3 * y
    salq    $4, %rdx           # rdx/t4    = (3*y) * 16
    leaq    4(%rdi,%rdx), %rcx   # rcx/t5    = x + t4 + 4
    imulq   %rcx, %rax          # rax/rval  = t5 * t2
    ret

```

Polling Question [Asm II – a]

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

 `leaq (,%rdi,9), %rax`

 `movq (,%rdi,9), %rax`

 `leaq (%rdi,%rdi,8), %rax`

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

 **We're lost...**

Feelings Check: LEA & MOV?

Control Flow

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rax	return value

```
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

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rax	return value

```
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 then jump to else
    movq    %rdi, %rax
    jump to done
else:
    movq    %rsi, %rax
done:
    ret
```


Conditionals and Control Flow

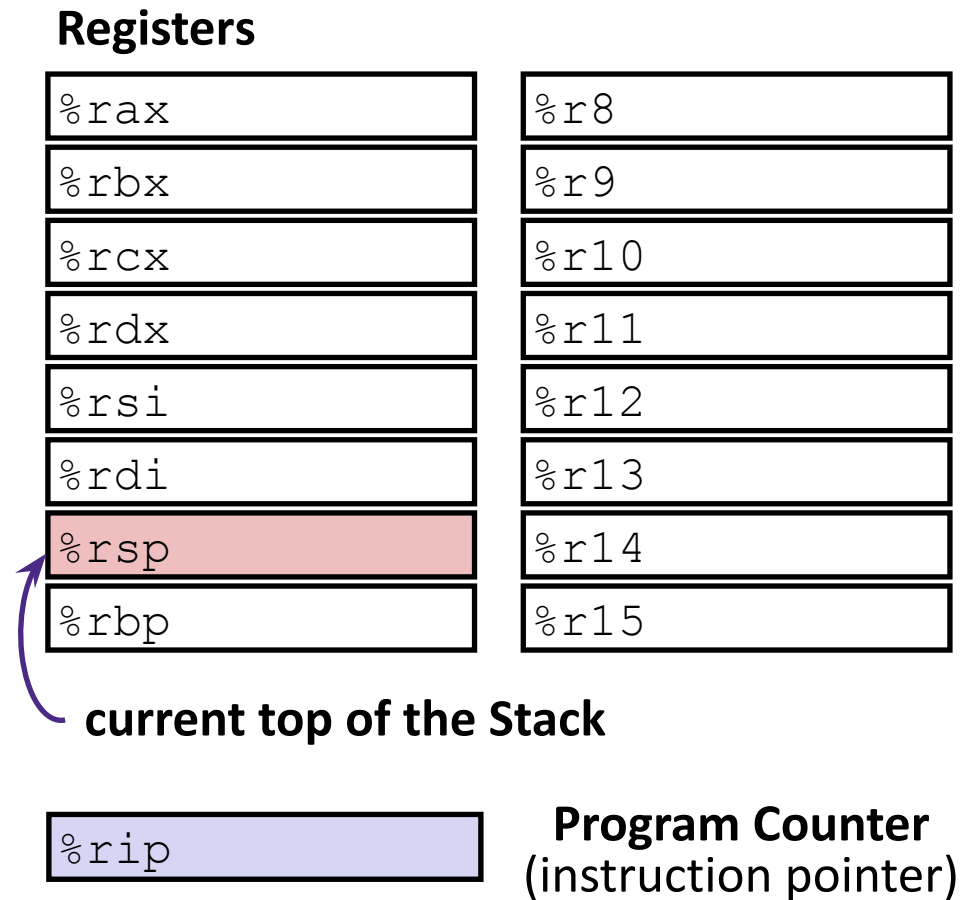
- Conditional branch/*jump*
 - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
 - *Always* jump when you get to this instruction
- Together, they can implement most control flow constructs in high-level languages:
 - **if** (*condition*)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

x86 Control Flow

- **Condition codes**
- **Conditional and unconditional branches**
- Loops
- Switches

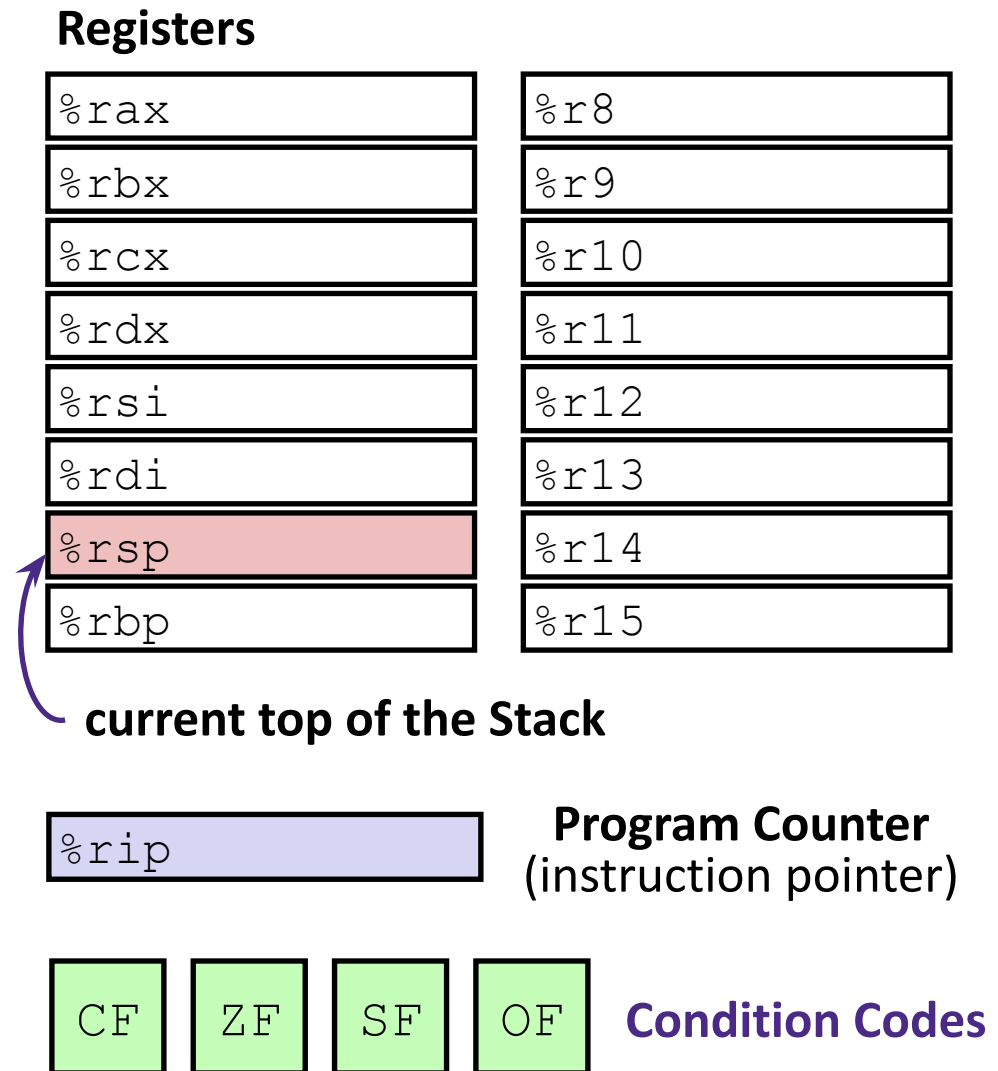
Processor State (x86-64, partial)

- Information about currently executing program
 - Temporary data (`%rax`, ...)
 - Location of runtime stack (`%rsp`)
 - Location of current code control point (`%rip`, ...)



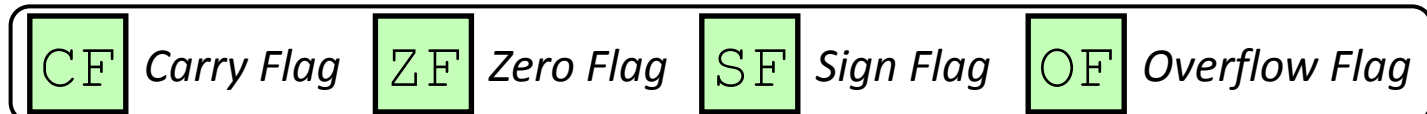
Processor State (x86-64, partial)

- Information about currently executing program
 - Temporary data (`%rax`, ...)
 - Location of runtime stack (`%rsp`)
 - Location of current code control point (`%rip`, ...)
 - Status of recent tests (**CF**, **ZF**, **SF**, **OF**)
 - Single bit registers:



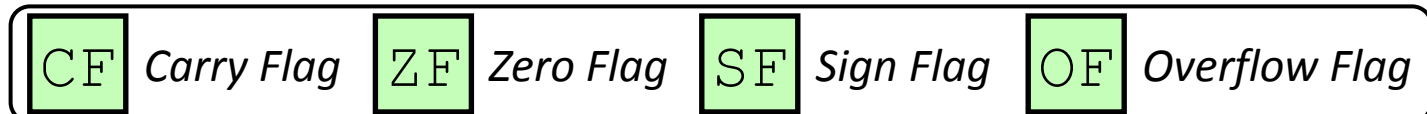
Condition Codes (Implicit Setting)

- ❖ *Implicitly* set by **arithmetic** operations
 - (think of it as side effects)
 - Example: **addq** src, dst \leftrightarrow $r = d+s$
 - **CF=1** if carry out from MSB (*unsigned* overflow)
 - **ZF=1** if $r==0$
 - **SF=1** if $r<0$ (if MSB is 1)
 - **OF=1** if *signed* overflow
($s>0 \ \&\& \ d>0 \ \&\& \ r<0$) || ($s<0 \ \&\& \ d<0 \ \&\& \ r>=0$)
 - **Not set by lea instruction (beware!)**



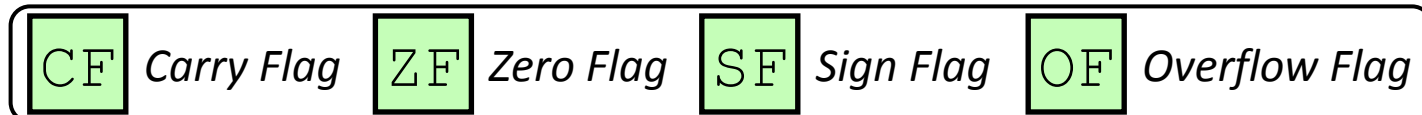
Condition Codes (Explicit Setting: Compare)

- *Explicitly* set by **Compare** instruction
 - **cmpq** *s, d* sets flags based on $d-s$, but doesn't store $d-s$
 - **CF=1** if carry out from MSB (i.e *unsigned* comparison)
 - **ZF=1** if $a==b$
 - **SF=1** if $(b-a) < 0$ (if MSB is 1)
 - **OF=1** if *signed* overflow
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (b-a) > 0) \ ||$
 $(a < 0 \ \&\& \ b > 0 \ \&\& \ (b-a) < 0)$



Condition Codes (Explicit Setting: Test)

- *Explicitly* set by **Test** instruction
 - **testq** src2, src1
 - **testq** a, b sets flags based on a&b, but doesn't store a&b
 - Useful to have one of the operands be a *mask*
 - Can't have carry out (**CF**) or overflow (**OF**)
 - **ZF=1** if $a \& b == 0$
 - **SF=1** if $a \& b < 0$ (signed)



Using Condition Codes: Jumping

- j^* Instructions
 - Jumps to **target** (an address) based on condition codes

Instruction	Condition	Description
<code>jmp target</code>	1	Unconditional
<code>je target</code>	ZF	Equal / Zero
<code>jne target</code>	$\sim ZF$	Not Equal / Not Zero
<code>js target</code>	SF	Negative
<code>jns target</code>	$\sim SF$	Nonnegative
<code>jg target</code>	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
<code>jge target</code>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<code>j1 target</code>	$(SF \wedge OF)$	Less (Signed)
<code>jle target</code>	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
<code>ja target</code>	$\sim CF \ \& \ \sim ZF$	Above (unsigned ">")
<code>jb target</code>	CF	Below (unsigned "<")

Using Condition Codes: Setting

- `set*` Instructions
 - Set low-order byte of `dst` to 0 or 1 based on condition codes
 - Does not alter remaining 7 bytes

Instruction	Condition	Description
<code>sete dst</code>	ZF	Equal / Zero
<code>setne dst</code>	\sim ZF	Not Equal / Not Zero
<code>sets dst</code>	SF	Negative
<code>setns dst</code>	\sim SF	Nonnegative
<code>setg dst</code>	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
<code>setge dst</code>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<code>setl dst</code>	$(SF \wedge OF)$	Less (Signed)
<code>setle dst</code>	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
<code>seta dst</code>	$\sim CF \ \& \ \sim ZF$	Above (unsigned ">")
<code>setb dst</code>	CF	Below (unsigned "<")

Reminder: x86-64 Integer Registers

- Accessing the low-order byte:

<code>%rax</code>	<code>%al</code>
<code>%rbx</code>	<code>%bl</code>
<code>%rcx</code>	<code>%cl</code>
<code>%rdx</code>	<code>%dl</code>
<code>%rsi</code>	<code>%sil</code>
<code>%rdi</code>	<code>%dil</code>
<code>%rsp</code>	<code>%spl</code>
<code>%rbp</code>	<code>%bpl</code>

<code>%r8</code>	<code>%r8b</code>
<code>%r9</code>	<code>%r9b</code>
<code>%r10</code>	<code>%r10b</code>
<code>%r11</code>	<code>%r11b</code>
<code>%r12</code>	<code>%r12b</code>
<code>%r13</code>	<code>%r13b</code>
<code>%r14</code>	<code>%r14b</code>
<code>%r15</code>	<code>%r15b</code>

Reading Condition Codes

Register	Use(s)
<code>%rdi</code>	1 st argument (x)
<code>%rsi</code>	2 nd argument (y)
<code>%rax</code>	return value

o `set*` Instructions

- Set a low-order byte to 0 or 1 based on condition codes
- Operand: byte register (e.g. `al`, `dl`) or byte in memory
- Do not alter remaining bytes in register
 - Use `movzbq` (zero-extended `mov`) to fill register

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

```
cmpq    %rsi, %rdi    #
setg    %al          #
movzbq  %al, %rax    #
ret
```

Reading Condition Codes

Register	Use(s)
<code>%rdi</code>	1 st argument (x)
<code>%rsi</code>	2 nd argument (y)
<code>%rax</code>	return value

o `set*` Instructions

- Set a low-order byte to 0 or 1 based on condition codes
- Operand: byte register (e.g. `al`, `dl`) or byte in memory
- Do not alter remaining bytes in register
 - Use `movzbq` (zero-extended `mov`) to fill register

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

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al          # Set when >
movzbq  %al, %eax    # Zero rest of %rax
ret
```

Aside: movz and movs

`movz __ __ src, regDest` # Move with zero extension

`movs __ __ src, regDest` # Move with sign extension

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

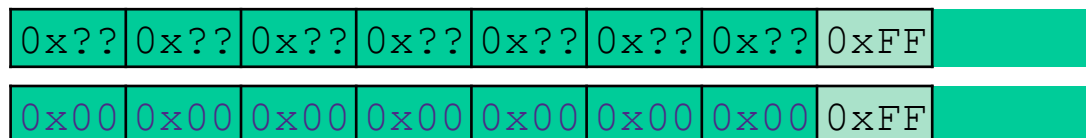
`movzSD` / `movsSD`:

S – size of source (**b** = 1B, **w** = 2)

D – size of dest (**w** = 2B, **l** = 4, **q** = 8)

Example:

`movzbq %al, %rbx`



Aside: movz and movs

`movz __ __ src, regDest` # Move with zero extension

`movs __ __ src, regDest` # Move with sign extension

- Copy from a *smaller* source value to a *larger* destination
- Source can be memory or register; Destination *must* be a register
- Fill remaining bits of dest with **zero** (`movz`) or **sign bit** (`movs`)

`movzSD` / `movsSD`:

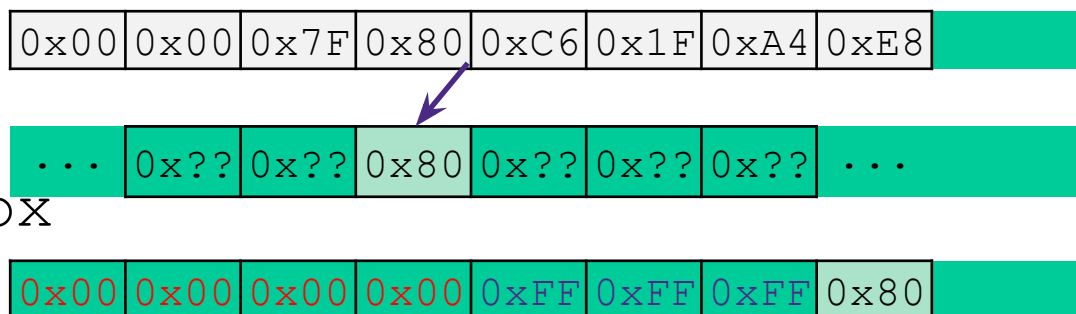
S – size of source (**b** = 1B, **w** = 2)

D – size of dest (**w** = 2B, **l** = 4, **q** = 8)

Note: In x86-64, any instruction that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.

Copy 1 byte from memory into 8-byte register & sign extend it

`movsbl (%rax), %ebx`



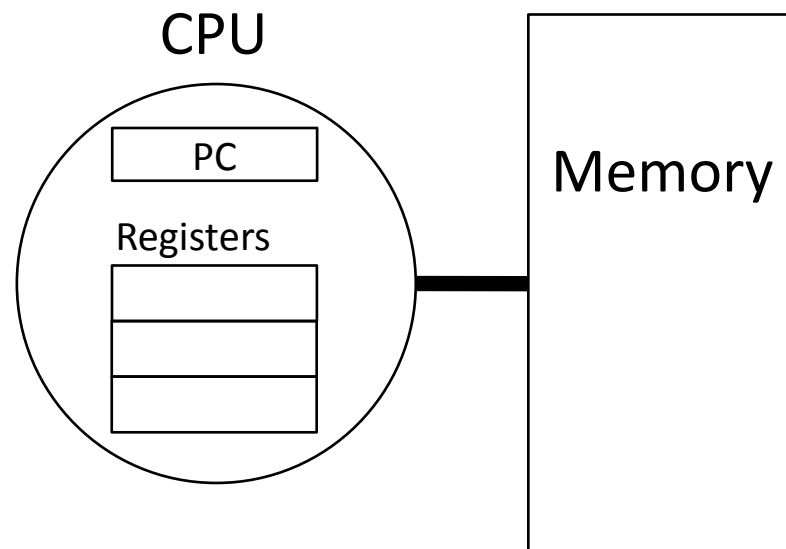
**How do we feel about
condition codes?**

Technical Summary

- **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
 - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations
- Control flow in x86 determined by status of Condition Codes
 - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, others exist
 - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit), read with `set` instruction
 - Jump instructions use flag values to determine next instruction to execute

Instruction Set Architectures

- The ISA defines:
 - The system's **state** (e.g. registers, memory, program counter)
 - The **instructions** the CPU can execute
 - The **effect** that each of these instructions will have on the system state



Instruction Set Philosophies

- *Complex Instruction Set Computing (CISC)*: Add more and more elaborate and specialized instructions as needed
 - Lots of tools for programmers to use, but hardware must be able to handle all instructions
 - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs
- *Reduced Instruction Set Computing (RISC)*: Keep instruction set small and regular
 - Easier to build fast hardware
 - Let software do the complicated operations by composing simpler ones

Design Goals

- *Complex Instruction Set Computing (CISC):*
Complete a task in as few instructions as possible
 - Fetch instructions from memory as part of other instructions
 - Minimize memory access (which takes a while)
- *Reduced Instruction Set Computing (RISC):*
All instructions should complete in a single clock cycle (FP mult can be ~30 cycles)
 - Easier to build fast hardware
 - Minimize complexity, maximize performance

What's the ideology?

Examining Ideology: Critical Reading

1. Go through descriptions, sales pitches, comparisons, really, any content
2. Highlight:
 - a. What's assumed to be important?
 - b. What's valued?
 - c. What's emphasized? What's given space?
3. Read through highlights, examine for themes
4. (ideally) Ask for how others see it! You're always limited by your own perspective.

Queering: Queer Reading

- Reading literature (texts, broadly) for heteronormative or identity binaries
 - Ideological assumptions! Assumed without question.
 - Are folks assumed to be straight? How much are gender norms enforced?
 - e.g. look at letters from celibate monks, what happens if we assumed *homonormativity*? What happens if we assume other ideologies?
- More popular in 1980s/1990s, though still in use
 - Surfacing ideology's much older than "queering", this is just especially prevalent for me

Design Goals

- *Complex Instruction Set Computing (CISC):*
Complete a task in as few instructions as possible
 - Fetch instructions from memory as part of other instructions
 - Minimize memory access (which takes a while)
- *Reduced Instruction Set Computing (RISC):*
All instructions should complete in a single clock cycle (FP mult can be ~30 cycles)
 - Easier to build fast hardware
 - Minimize complexity, maximize performance

Design Goals

- *Complex Instruction Set Computing (CISC):*
Complete a task in **as few instructions as possible**
 - Fetch instructions from memory as part of other instructions
 - **Minimize** memory access (which takes a while)
- *Reduced Instruction Set Computing (RISC):*
All instructions should complete in a single clock cycle (FP mult can be ~30 cycles)
 - **Easier to build fast** hardware
 - **Minimize complexity, maximize performance**

Design Goals

- “as few instructions as possible” -- minimalism
- “Minimize memory access” -- minimalism, efficiency
- Easy to build -- minimalism
- Emphasis on “fast” hardware -- efficiency, production, performance
- “Minimize complexity” -- minimalism
- “Maximize performance” -- I mean, it’s right there

Ideology through Design Goals

- *Minimalism, Efficiency, Performance*
 - We've seen this before! We've seen this throughout!
- CISC & RISC are usually viewed as “different design philosophies”
 - How are they actually different?

Ideology through Design Goals

- *Minimalism, Efficiency, Performance*
 - We've seen this before! We've seen this throughout!
- CISC & RISC are usually viewed as “different design philosophies”
 - How are they actually different?
 - Different goals, but both pursue efficiency, minimalism
 - Really, just this is just efficiency; minimalism is a means to an end

Different implementations, same ideology!

Most of “traditional” CS follows both of these

The Battle of the 80's

Think of your next microcomputer as a weapon against horrendous inefficiencies, outrageous costs and antiquated speeds. We invite you to peruse this chart.

Features:	8080A	Z80-CPU	Features:	8080A	Z80-CPU
Power Supplies	+5,-5,+12	+5	Instructions	78	158*
Clock	2 ϕ , +12 Volt	1 ϕ , 5 Volt	OP Codes	244	696
Standard Clock Speed	500 ns	400 ns	Addressing Modes	7	11
Interface	Requires 8222, 8228 & 8224	Requires no other logic and includes dynamic RAM Refresh	Working Registers	8	17
Interrupt	1 mode	3 modes; up to 6X faster	Throughput	Up to 5 times greater than the 8080A	
Non-maskable Interrupt	No	Yes	Program Memory Space	Generally 50% less than the 8080A	

*Including all of the 8080A's instructions.



Announcing Zilog Z-80 microcomputer products.
With the next generation, the battle is joined.

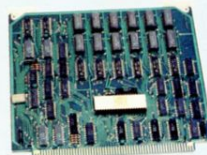
The Z-80: A new generation LSI component set including CPU and I/O Controllers.

The Z-80: Full software support with emphasis on high-level languages.

The Z-80: A floppy disc-based development system with advanced real-time debug and in-circuit emulation capabilities.

The Z-80: Multiple sourcing available now.

Your ammunition:
A chip off a new block.



A single chip, N-channel processor arms you with a super-set of 158 instructions that include all of the 8080A's 78 instructions with total software compatibility. The new instructions include 1, 4, 8 and 16-bit operations. And that means less programming time, less paper and less end costs.

And you'll be in command of powerful instructions: Memory-to-memory or memory-to-I/O block transfers and searches, 16-bit arithmetic, 9 types of rotates and shifts, bit manipulation and a legion of addressing modes. Along with this army you'll also get a standard instruction speed of 1.6 μ s and all Z-80 circuits require only a single 5V power supply and a single phase 5V clock. And you should know that a family of Z-80 programmable circuits allow for direct interface to a wide range of both parallel and serial interface peripherals and even dynamic memories without other external logic.

With these features, the Z80-CPU generally requires approximately 50% less memory space for program storage

yet provides up to 500% more throughput than the 8080A. Powerful ammunition at a surprisingly low cost and ready for immediate shipment.

Mighty weapons against an enemy entrenched: The Z-80 development system.

You'll be equipped with performance and versatility unmatched by any other microcomputer development system in the field. Thanks to a floppy disc operating system in alliance with a sophisticated Real-Time Debug Module.

The Zilog battalion includes:

- Z80-CPU Card.
- 16K Bytes of RAM Memory, expandable to 60K Bytes.
- 4K Bytes of ROM/RAM Monitor software.
- Real-Time Debug Module and In-Circuit Emulation Module.
- Dual Floppy Disc System.
- Optional I/O Ports for other High Speed Peripherals are also available.
- Complete Software Package including Z-80 Assembler, Editor, Disc Operating System, File Maintenance and Debug.



On standby: Software support.

All this is supported by a contingent of software including: resident micro-computer software, time sharing programs, libraries and high-level languages such as PL/Z.

On standby: User support.

Zilog conducts a wide range of strategic meetings and design oriented workshops to provide the know-how required to implement the Z-80 Micro-computer Product line into your design. All hardware, software and the development system are thoroughly explained with "hands-on" experience in the classroom. Your Zilog representative can provide you with further details on our user support program.



Reinforcements: A reserve of technological innovations.

The Zilog Z-80 brings to the battle-front new levels of performance and ease of programming not available in second generation systems. And while all the others busy themselves with overtaking the Z-80, we're busy on the next generation—continuing to demonstrate our pledge to stay a generation ahead.

The Z-80's troops are the specialists who were directly responsible for the development of the most successful first and second generation micro-processors. Nowhere in the field is there a corps of seasoned veterans with such a distinguished record of victory.

Signal us for help. We'll dispatch appropriate assistance.



Zilog MICROCOMPUTERS

170 State Street, Los Altos, California 94022
(415) 941-5055/TWX 910-370-7955

Circle 33 on reader service card

AN AFFILIATE OF **EXON** ENTERPRISES INC.



The Case for the Reduced Instruction Set Computer

David A. Patterson

Computer Science Division
University of California
Berkeley, California 94720

David R. Ditzel

Bell Laboratories
Computing Science Research Center
Murray Hill, New Jersey 07974

INTRODUCTION

One of the primary goals of computer architects is to design computers that are more cost-effective than their predecessors. Cost-effectiveness includes the cost of hardware to manufacture the machine, the cost of programming, and costs incurred related to the architecture in debugging both the initial hardware and subsequent programs. If we review the history of computer families we find that the most common architectural change is the trend toward ever more complex machines. Presumably this additional complexity has a positive tradeoff with regard to the cost-effectiveness of newer models. In this paper we propose that this trend is not always cost-effective, and in fact, may even do more harm than good. We shall examine the case for a Reduced Instruction Set Computer (RISC) being as cost-effective as a Complex Instruction Set Computer (CISC). This paper will argue that the next generation of VLSI computers may be more effectively implemented as RISC's than CISC's.

As examples of this increase in complexity, consider the transitions from IBM System/3 to the System/38 [Utley78] and from the DEC PDP-11 to the VAX11. The complexity is indicated quantitatively by the size of the control store; for DEC the size has grown from 256 x 56 in the PDP 11/40 to 5120 x 96 in the VAX 11/780.

**We'll be doing this
much more in 351!
I'm doing this with material for CSE351!**