

Assembly developers be like



TODO

- Assignment 2 assigned due October 7

Overview for today

- Advanced memory operands
- Arithmetic operations

Exam 1

- First third (Exam 1):
 - Data Representation
 - Memory
- Second third (Exam 2):
 - Machine Code
 - Memory exploits
- Final third (Final exam):
 - Memory hierarchy
 - Processes

What to study?

- Primarily:
 - Lab 0, Lab 1, Lab 2
 - All in-class exercises on data representation and memory
- Other topics lightly covered from:
 - The slides, your notes, etc.

Operand types

- Three operand types:
 - Immediate
 - Register
 - Memory

Allowed MOV operand combinations

- **movq immediate, register**
- **movq immediate, memory**
- **movq register, register**
- **movq register, memory**
- **movq memory, register**

x86-64 registers

	<code>%rax</code>	<code>%r8</code>
	<code>%rbx</code>	<code>%r9</code>
	<code>%rcx</code>	<code>%r10</code>
	<code>%rdx</code>	<code>%r11</code>
	<code>%rsi</code>	<code>%r12</code>
	<code>%rdi</code>	<code>%r13</code>
Stack Pointer	<code>%rsp</code>	<code>%r14</code>
Frame/Base Pointer	<code>%rbp</code>	<code>%r15</code>
Program Counter (Instruction Pointer)	<code>%rip</code>	

x86 virtual registers

- Each general purpose register has variants
 - 64-bit (**%rax**, the physical register)
 - 32-bit (**%rax** → **%eax**)
 - 16-bit (**%rax** → **%ax**)
 - 8-bit (**%rax** → **%al**)
- Naming is, of course, inconsistent...

64-Bit Register	Lowest 32 Bits	Lowest 16 Bits	Lowest 8 Bits
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

x86 instruction suffixes

- So, for example:
- `int *rcx = ...;`
- `int rax = *rcx;` → `movl (%rcx), %eax`
- `short *rcx = ...;`
- `short rax = *rcx;` → `movw (%rcx), %ax`
- `char *rcx = ...;`
- `char rax = *rcx;` → `movb (%rcx), %al`

What happens to other bits?

- **movl (%rcx), %eax**
 - Upper 32 bits of %rax are what now?
- **movw (%rcx), %ax**
 - Upper 48 bits of %rax are what now?
- **movb (%rcx), %al**
 - Upper 56 bits of %rax are what now?

What happens to other bits?

- **movl (%rcx), %eax**
 - Upper 32 bits of %rax are what now? Zero
- **movw (%rcx), %ax**
 - Upper 48 bits of %rax are what now? Unchanged
- **movb (%rcx), %al**
 - Upper 56 bits of %rax are what now? Unchanged

General principle

- Always use 64-bit *memory* operands on a 64-bit machine
- Change the size of the *register* operands when changing the instruction suffix
- Using the wrong register size is illegal

Practice translating to English

- **movq %rax, %rbx** →
- “Copy 8 bytes from the register **%rax** to the register **%rbx**”

Practice translating to English

- `movq %rax, (%rbx)` →
- ???

Practice translating to English

- `movq %rax, (%rbx)` →
- “Copy 8 bytes from the register `%rax` to memory at the address `%rbx`”

Practice translating to English

- `movq (%rax), %rbx` →
- ???

Practice translating to English

- `movq (%rax), %rbx` →
- “Copy 8 bytes from memory at the address (%rax) to the register %rbx”

Practice translating to English

- `movl (%rax), %ebx` →
- ???

Practice translating to English

- `movl (%rax), %ebx` →
- “Copy 4 bytes from memory at the address (%rax) to the register %ebx”

Practice translating to English

- `movl 1234, %ebx` →
- ???

Practice translating to English

- `movl 1234, %ebx` →
- “Copy 4 bytes from memory at the address **1234** to the register **%ebx**”

Practice translating to English

- `movl $1234, %ebx` →
- ???

Practice translating to English

- `movl $1234, %ebx` →
- “Copy the value **1234** to the register **%ebx**”

mov variants

- movzbw, movzbl, movzbq
- movzwl, movzwq
- movzlq
- movsbw, movsbl, movsbq
- movswl, movswq
- movslq

mov variants

- `movzbw, movzbl, movzbq`
- `movzwl, movzwq`
- `movzlq`
- `movsbw, movsbl, movsbq`
- `movswl, movswq`
- `movslq`
- These all move from a smaller size to larger

mov variants

- `movzbw, movzbl, movzbq`
- `movzwl, movzwq`
- `movzlq`
- `movsbw, movsbl, movsbq`
- `movswl, movswq`
- `movslq`

mov variants

- movzbw, movzbl, movzbq
- movzwl, movzwq
- movzlq
- movsbw, movsbl, movsbq
- movswl, movswq
- movslq

movz* vs. movs*

- z: zero extend
- s: sign extend

movz* vs. movs*

- z: zero extend: useful for promoting unsigned
- s: sign extend: useful for promoting signed

Memory operands

- Remember our three operand types:
 - Immediate
 - Register
 - Memory
- We need to say more about the latter!

Memory operands

- So far we have learned how to do `*rax`
- But how would we do `rax[4]`?
- One way is using another instruction to do some arithmetic on the address before we use it (we will learn arithmetic instructions coming soon!)
- But we can also do this directly with x86 memory operands

Memory displacement

- `unsigned char *rax = ...;`
- `*rax` $\rightarrow$ `(%rax)`
- `rax[3]` $\rightarrow$ `3(%rax)`
- `rax[5]` $\rightarrow$ `5(%rax)`
- `rax[-1]` $\rightarrow$ `-1(%rax)`

Memory displacement

- `unsigned char *rax = ...;`
- `*rax → (%rax)`
- `rax[rbx + 3] → 3(%rax,%rbx)`
- `rax[rcx + 5] → 5(%rax,%rcx)`
- `rax[rdx + -1] → -1(%rax,%rdx)`

Memory displacement

- unsigned **short** *rax = ...;
- *rax → (%rax)
- rax[rbx + 3] → **6**(%rax,%rbx,**2**)
- rax[rcx + 5] → **10**(%rax,%rcx,**2**)
- rax[rdx + -1] → **-2**(%rax,%rdx,**2**)

Memory displacement

- unsigned **int** *rax = ...;
- *rax → (%rax)
- rax[rbx + 3] → **12**(%rax,%rbx,**4**)
- rax[rcx + 5] → **20**(%rax,%rcx,**4**)
- rax[rdx + -1] → **-4**(%rax,%rdx,**4**)

Memory displacement

- unsigned **long** *rax = ...;
- *rax → (%rax)
- rax[rbx + 3] → **24**(%rax,%rbx,**8**)
- rax[rcx + 5] → **40**(%rax,%rcx,**8**)
- rax[rdx + -1] → **-8**(%rax,%rdx,**8**)

Scale

- `24(%rax,%rbx,8)`
- The 8 is called the *scale*
- x86 only supports scales of 1, 2, 4, 8

Memory operand summary

General Form:

$D(Rb, Ri, S) \quad Mem[D + Reg[Rb] + S * Reg[Ri]]$

D Constant "displacement"

Rb Base register

Ri Index register

s Scale constant: 1, 2, 4, or 8

$5(\%rax, \%rbx, 8) \quad Mem[5 + \%rax + 8 * \%rbx]$

Special Cases:

(Rb)	(%rax)	Mem[Reg[Rb]]
D(Rb)	5(%rax)	Mem[D + Reg[Rb]]
(Rb, Ri)	(%rax, %rbx)	Mem[Reg[Rb] + Reg[Ri]]
D(Rb, Ri)	5(%rax, %rbx)	Mem[D + Reg[Rb] + Reg[Ri]]
(Rb, Ri, S)	(%rax, %rbx, 8)	Mem[Reg[Rb] + S * Reg[Ri]]
(, Ri, S)	(, %rbx, 8)	Mem[S * Reg[Ri]]
D(, Ri, S)	5(, %rbx, 8)	Mem[D + S * Reg[Ri]]

Summary: when x86 is illegal

- Instruction suffix doesn't match register operand size
- Destination operand is an immediate value
- Source and destination operands are both memory operands
- Invalid scale values

Three categories of instructions

1. Data movement – moving data between registers and memory
 - E.g., load data from memory address 0x20 into register `%rax`
2. **Arithmetic** – perform arithmetic on data in registers or in memory
 - E.g., take the value in `%rax` and add it to the value in `%rbx`
3. Control flow – choosing next instruction
 - Calling a function
 - Conditional branches, loops, etc.

addq	<i>Src, Dest</i>	Dest = Dest + Src	
subq	<i>Src, Dest</i>	Dest = Dest - Src	
imulq	<i>Src, Dest</i>	Dest = Dest * Src	
sarq	<i>Src, Dest</i>	Dest = Dest >> Src	<i>Arithmetic RShift</i>
shrq	<i>Src, Dest</i>	Dest = Dest >> Src	<i>Logical RShift</i>
salq	<i>Src, Dest</i>	Dest = Dest << Src	<i>Also called shlq</i>
xorq	<i>Src, Dest</i>	Dest = Dest ^ Src	
andq	<i>Src, Dest</i>	Dest = Dest & Src	
orq	<i>Src, Dest</i>	Dest = Dest Src	
incq	<i>Dest</i>	Dest = Dest + 1	
decq	<i>Dest</i>	Dest = Dest - 1	
negq	<i>Dest</i>	Dest = -Dest	
notq	<i>Dest</i>	Dest = ~Dest	

leaq	<i>Src, Dest</i>	Dest = Src (as expr)	<i>No memory access!</i>
------	------------------	----------------------	--------------------------

Arithmetic instructions

- Addition example
- **`addq %rax, %rbx`**
- Read as “add rax to rbx”
- $rbx = rbx + rax$ (in C)
- Remember: destination is second argument 

Arithmetic instructions

- Addition example
- **subq %rax, %rbx**
- Read as “subtract rax from rbx”
- $rbx = rbx - rax$ (in C)
- Remember: destination is second argument 

Arithmetic instructions

- Addition example
- **`imulq %rax, %rbx`**
- Read as “multiply rbx by rax”
- `rbx = rbx * rax` (in C)
- Remember: destination is second argument 

addq	<i>Src, Dest</i>	Dest = Dest + Src	
subq	<i>Src, Dest</i>	Dest = Dest - Src	
imulq	<i>Src, Dest</i>	Dest = Dest * Src	
sarq	<i>Src, Dest</i>	Dest = Dest >> Src	<i>Arithmetic RShift</i>
shrq	<i>Src, Dest</i>	Dest = Dest >> Src	<i>Logical RShift</i>
salq	<i>Src, Dest</i>	Dest = Dest << Src	<i>Also called shlq</i>
xorq	<i>Src, Dest</i>	Dest = Dest ^ Src	
andq	<i>Src, Dest</i>	Dest = Dest & Src	
orq	<i>Src, Dest</i>	Dest = Dest Src	
incq	<i>Dest</i>	Dest = Dest + 1	
decq	<i>Dest</i>	Dest = Dest - 1	
negq	<i>Dest</i>	Dest = -Dest	
notq	<i>Dest</i>	Dest = ~Dest	

leaq	<i>Src, Dest</i>	Dest = Src (as expr)	<i>No memory access!</i>
------	------------------	----------------------	--------------------------

Arithmetic instructions

- Addition example
- `addq %rax, %rbx`
- `addl %eax, %ebx`
- `addw %ax, %bx`
- `addb %al, %bl`

Memory displacement

- unsigned **int** *rax = ...;
- rax[rbx + 3] → **12**(%rax,%rbx,**4**)
- rax[rcx + 5] → **20**(%rax,%rcx,**4**)
- rax[rdx + -1] → **-4**(%rax,%rdx,**4**)

Memory displacement

- unsigned **int** *rax = ...;
- &rax[rbx + 3] → ??? **12**(%rax,%rbx,**4**)
- &rax[rcx + 5] → ??? **20**(%rax,%rcx,**4**)
- &rax[rdx + -1] → ??? **-4**(%rax,%rdx,**4**)

Memory displacement

- unsigned **int** *rax = ...;
- &rax[rbx + 3] → leaq **12**(%rax,%rbx,**4**), %rsi
- &rax[rcx + 5] → leaq **20**(%rax,%rcx,**4**), %rsi
- &rax[rdx + -1] → leaq **-4**(%rax,%rdx,**4**), %rsi

Load effective address

- `leaq i(%rax,%rbx,j), %rsi`
- $rsi = rax + rbx * j + i$
- We will see `leaq` in other surprising places though...

Load effective address

- `leaq (%rax,%rax,4), %rbx`
- What does this compute?

Load effective address

- `leaq (%rax,%rax,4), %rbx`
- What does this compute?
- $rbx = rax * 5$
- But why not just...
- `imulq $5, %rax, %rbx?`

Load effective address

- `leaq (%rax,%rax,4), %rbx`
- What does this compute?
- $rbx = rax * 5$
- But why not just...
- `imulq $5, %rax, %rbx?`
- Because that doesn't exist!
- LEA effectively allows us a third operand!

Compiler optimizations

- Compilers love doing tricks like this
- They are called *compiler optimizations*
- (We saw them earlier talking about undefined behavior in C)
- They are trying to make our code faster

Compiler optimizations

- Other optimizations we commonly see:
- `rax = rax * 8` $\rightarrow$ `shlq $3, %rax`
- `rax = rax / 8` $\rightarrow$ `shrq $3, %rax`
- *Strength reduction*: using cheaper operations when possible

Load effective address

- LEA's first operand must always be a memory operand
 - But that operand never performs an actual memory operation

Operands Exercise

General Form:

$D(Rb, Ri, S) \quad Mem[D + Reg[Rb] + S * Reg[Ri]]$

D Constant "displacement"

Rb Base register

Ri Index register

s Scale constant: 1, 2, 4, or 8

$5(\%rax, \%rbx, 8) \quad Mem[5 + \%rax + 8 * \%rbx]$

Special Cases:

(Rb)	(%rax)	Mem[Reg[Rb]]
D(Rb)	5(%rax)	Mem[D + Reg[Rb]]
(Rb, Ri)	(%rax, %rbx)	Mem[Reg[Rb] + Reg[Ri]]
D(Rb, Ri)	5(%rax, %rbx)	Mem[D + Reg[Rb] + Reg[Ri]]
(Rb, Ri, S)	(%rax, %rbx, 8)	Mem[Reg[Rb] + S * Reg[Ri]]
(, Ri, S)	(, %rbx, 8)	Mem[S * Reg[Ri]]
D(, Ri, S)	5(, %rbx, 8)	Mem[D + S * Reg[Ri]]

Three categories of instructions

1. Data movement – moving data between registers and memory
 - E.g., load data from memory address 0x20 into register `%rax`
2. Arithmetic – perform arithmetic on data in registers or in memory
 - E.g., take the value in `%rax` and add it to the value in `%rbx`
3. **Control flow** – choosing next instruction
 - Calling a function
 - Conditional branches, loops, etc.

TODO

- Assignment 2 assigned due October 7