

Lecture 07

Registers and x86 Assembly

Comp_Sci 213
Introduction to Computer Systems
Wednesday, Oct. 14, 2026
Kate Smith – Fall 2026

Slides adapted from:

Ghena, St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Notes

- HW 1 due Thursday, Oct. 15
- Bomb Lab out
 - Due Oct 29
- Midterm on Monday, Oct. 26
 - If you need accommodations, please contact AccessibleNU

Midterm exam

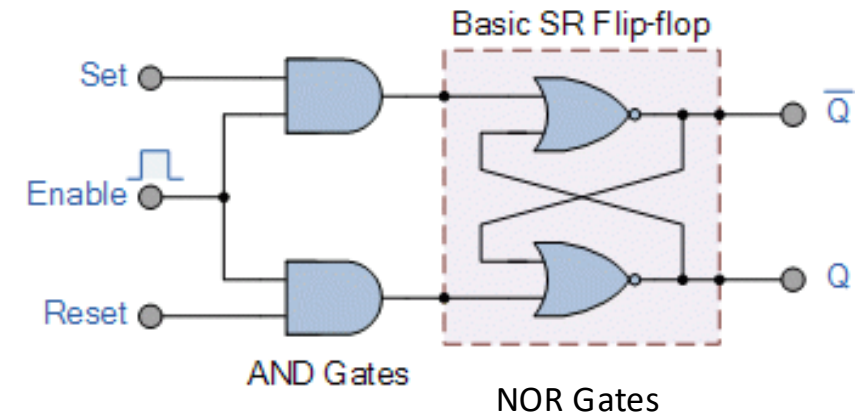
- In class, pencil and paper
- You can have 1 cheat sheet, double sided
 - Handwritten or typed!
- Our exam will be different from the other 213 section
- I will save some time during Wed. Oct 21 class (~20 minutes) for exam review

Topics

- **Registers**
- x86-64 Assembly
 - Introduction
 - Move Instruction
 - Memory Addressing Modules

Hardware uses registers for variables

- Unlike C, assembly doesn't have variables as you know them
 - Instead, assembly uses *registers* to store values
- Registers are:
 - Small memory chunks of a fixed size
 - Can be read or written
 - Limited in number
 - Very fast and low power to access
- Registers are different from variables or RAM:
 - They don't have an address
 - We can only access them in assembly
 - They don't have types (just bits)
 - The operation performed determines how contents are treated



How many registers?

- Tradeoff between speed and availability
 - More registers can hold more variables
 - Simultaneously; all registers are slower
 - Also registers take physical space within the chip
- x86-64 has 16 registers (for integer operations)
 - Historically only 8 registers
 - Added 8 more with 64-bit extensions

How big should each register be?

- Registers are usually the size of a *word*
 - The natural unit of data for a processor
 - Width of the data type that a CPU can process in one instruction
 - Likely the size of its registers
 - Imprecise term that will inevitably slip into explanations
- x86 processors started with 16-bit words
- IA32 upgraded to 32-bit “double word” registers
- x86-64 upgraded again 64-bit “quad word” registers

Why did we change the register size??

- Registers need to be able to hold memory addresses
- So, the size of a register limits the total size of memory!!
 - Each address is a single byte of memory
- Memory limitations based on register size
 - **16-bit registers:** 2^{16} addresses = 2^{16} bytes = **64 KB of RAM**
 - **32-bit registers:** 2^{32} addresses = 2^{32} bytes = **4 GB of RAM**
 - **64-bit registers:** 2^{64} addresses = 2^{64} bytes = **16 EB of RAM**
 - Kilobyte, Megabyte, Gigabyte, Terabyte, Petabyte, Exabyte
- We moved to 32-bit in the early 90s and 64-bit in the late-00s
 - No computer has 128-bit words, although many have special-purpose registers that big

x86-64 64-bit Integer Registers

<code>%rax</code>
<code>%rbx</code>
<code>%rcx</code>
<code>%rdx</code>
<code>%rsi</code>
<code>%rdi</code>
<code>%rsp</code>
<code>%rbp</code>

 original x86 registers

<code>%r8</code>
<code>%r9</code>
<code>%r10</code>
<code>%r11</code>
<code>%r12</code>
<code>%r13</code>
<code>%r14</code>
<code>%r15</code>

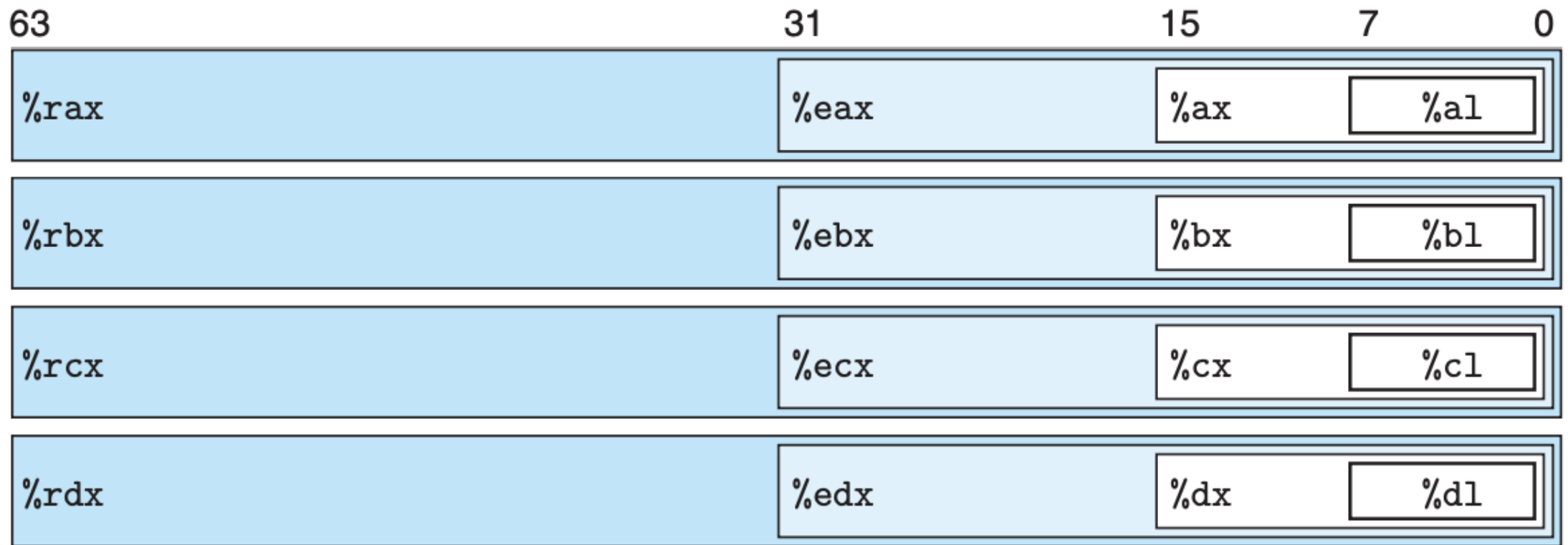
 additional x86-64 registers

Historical Register Purposes

Name Origin (mostly obsolete)

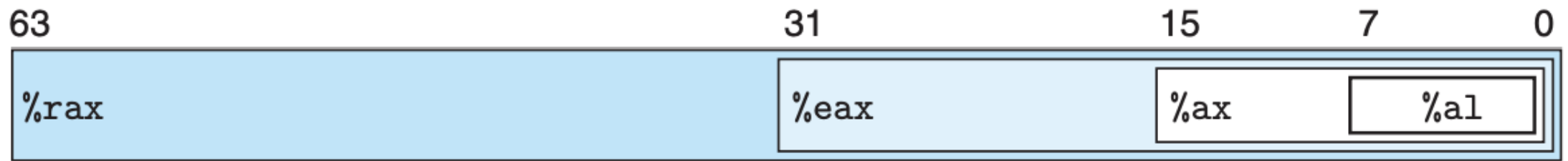
<code>%rax</code>	Accumulate
<code>%rbx</code>	Base
<code>%rcx</code>	Counter
<code>%rdx</code>	Data
<code>%rsi</code>	Source Index
<code>%rdi</code>	Destination Index
<code>%rsp</code>	Stack Pointer (still important)
<code>%rbp</code>	Base Pointer

x86-64 Register Access Options



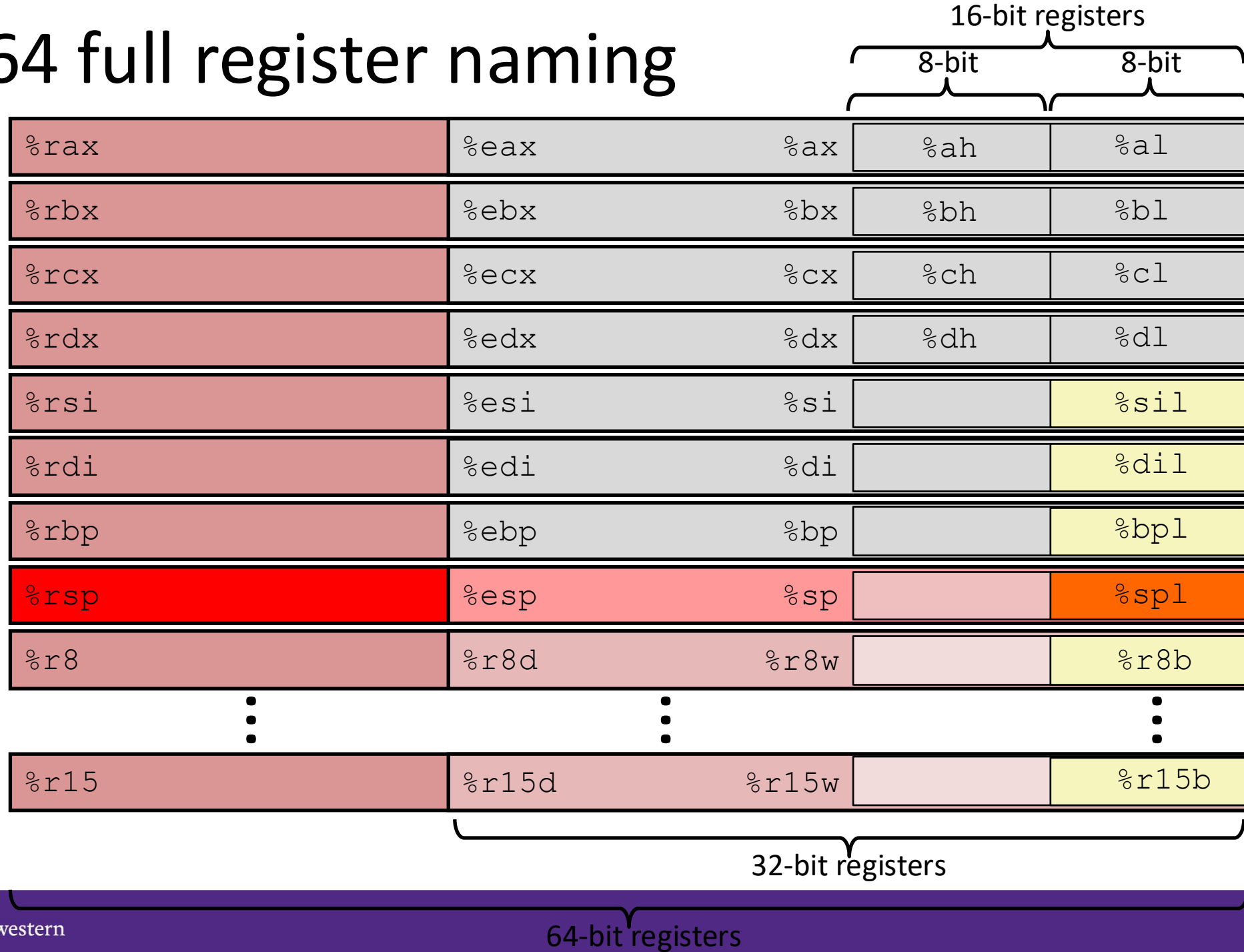
Registers can be accessed by any of these names to work with 8-byte, 4-byte, 2-byte, or 1-byte data

x86-64 Register Access Options



- The same data can be accessed under different names and widths
- Example: store 64-bit value 0x00000000000010A0B in %rax
 - %rax: 0x00000000000010A0B (64-bit)
 - %eax: 0x00010A0B (32-bit)
 - %ax: 0x0A0B (16-bit)
 - %al: 0x0B (8-bit)

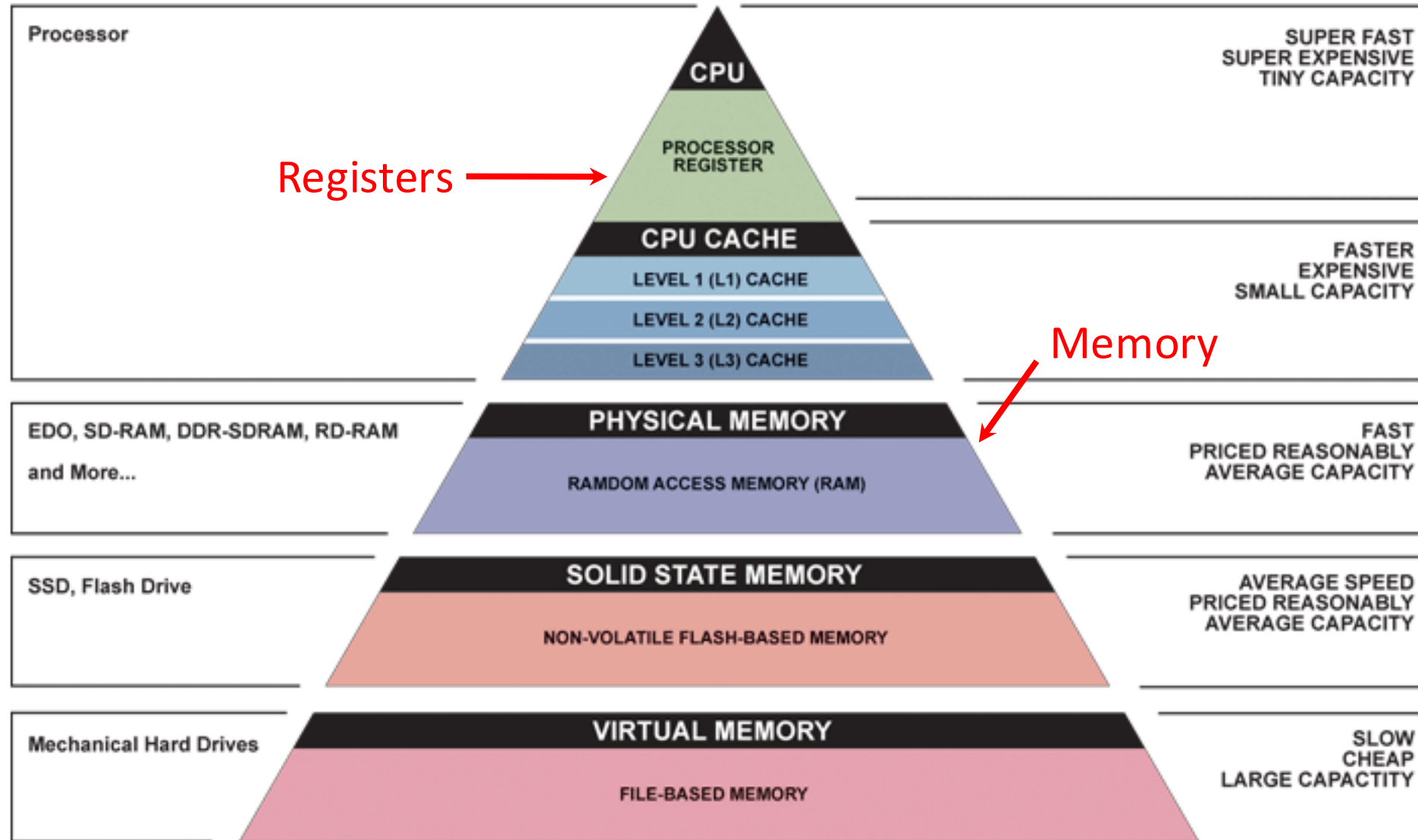
x86-64 full register naming



Registers versus Memory

- What if there are more variables than registers?
 - Keep most frequently used in registers and move the rest to memory (called *spilling* to memory)
- Why not put all variables in memory?
 - Smaller is faster: registers 100-500 times faster
 - Memory Hierarchy
 - Registers: 16 registers * 64 bits = 128 Bytes
 - RAM: 4-32 GB (4-32 billion bytes)
 - SSD: 100-1000 GB (0.1-1 trillion bytes)

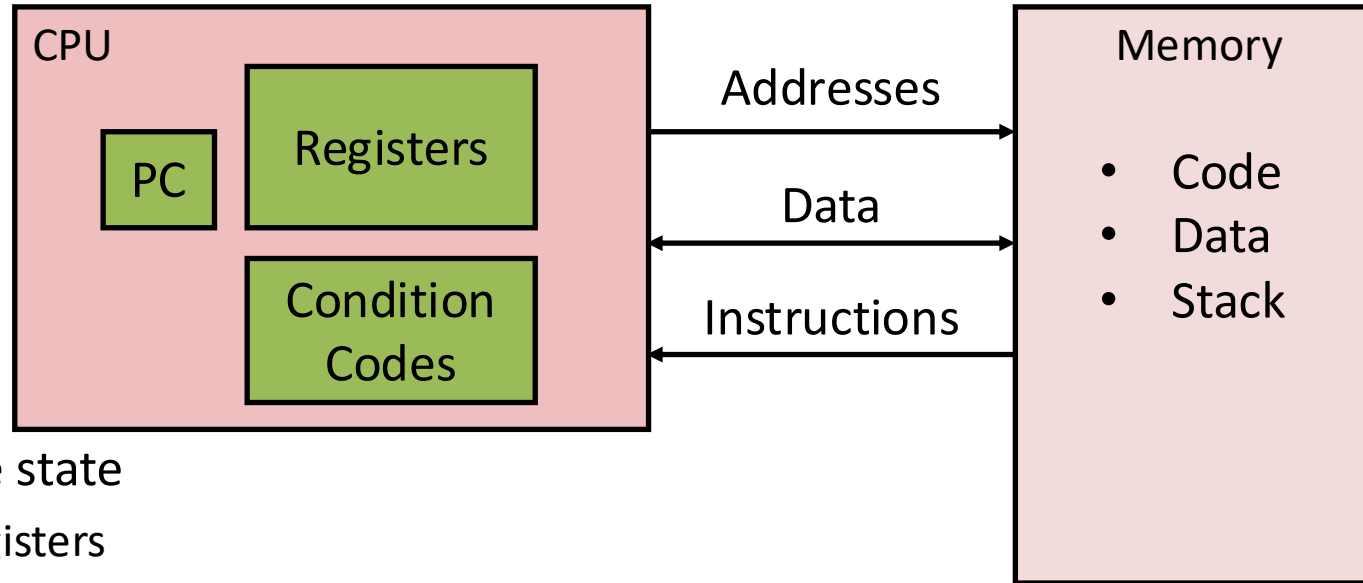
Memory Hierarchy



Special-purpose register: Instruction Pointer

- Instruction Pointer: `%rip` (not listed in prior tables)
 - Contains the address of the currently executing instruction
 - Special-purpose: only used for this one thing
- Processor hardware uses `%rip` when loading instructions
 - Load instruction from memory pointed to by `%rip`
 - Advance `%rip` to the next instruction
 - Repeat
 - Note: hardware does this automatically, you don't (usually) interact with this at all

Assembly Programmer's View of System State



Programmer-visible state

- Named registers
 - Together in “register file”
 - Heavily used program data
- The Instruction Pointer (`%rip` in x86-64)
 - Address of next instruction
- Condition codes
 - Store status information about most recent arithmetic operation
 - Used for conditional branching

Memory

- Byte-addressable array
- Code and user data
- Includes *the Stack* (for supporting procedures and variables)

Question

Which of these about registers is FALSE?

- [A] Registers are faster to access than memory
- [B] Registers do not have a type
- [C] Registers can have special purposes
- [D] Registers are dynamically created as needed

Question

Which of these about registers is FALSE?

[A] Registers are faster to access than memory

[B] Registers do not have a type

*There are a fixed number
of registers for a given
architecture!!*

[C] Registers can have special purposes

[D] Registers are dynamically created as needed

Topics

- Registers
- **x86-64 Assembly**
 - **Introduction**
 - Move Instruction
 - Memory Addressing Modules

Writing Assembly Code? In 2026???

- Chances are, you'll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:
 - Behavior of programs in the presence of bugs
 - When high-level language model breaks down
 - Tuning program performance
 - Understanding compiler optimizations and sources of program inefficiency
 - Implementing systems software
 - What are the “states” of processes that the OS must manage
 - Using special units (timers, I/O co-processors, etc.) inside processor!
 - Fighting malicious software
 - Distributed software is in binary form

Inspecting assembly code

```
gcc -Og -S hello.c
```

- Optimize, but keep things easy to debug. Middle ground between `-O0` and `-O1`
 - **Compared to `-O0`:** values are kept in registers instead of being stored to and reloaded from the stack around every statement. The assembly is much shorter and closer to what a person would write
 - **Compared to `-O1`/`-O2`:** passes that reorder or restructure code heavily (aggressive inlining, loop transformations, instruction scheduling, and so on) are left off. The result still follows the structure of your C source almost line by line, so stepping through it in `gdb` makes sense.
- `-S` stops after compilation and writes `hello.s`

```
C hello.c > ...
1  /*
2  * hello world
3  */
4  #include <stdio.h>
5
6  int main()
7  {
8      float var_1, var_2;
9      float var_3;
10     var_1 = 5;
11     var_2 = 2;
12
13     var_3 = var_1 / var_2;
14     printf("%6.3f \n", var_3);
15
16     printf("hello, world!\t!\n");
17 }
18
```

```
dhcp-10-105-113-158:fall2026-213 katesmith$ gcc -o hello hello.c
dhcp-10-105-113-158:fall2026-213 katesmith$ ./hello
 2.500
hello, world!  !
```

Inspecting assembly code

```
gcc -Og -S hello.c
```

- Optimize, but keep things easy to debug. Middle ground between `-O0` and `-O1`
 - **Compared to `-O0`:** values are kept in registers instead of being stored to and reloaded from the stack around every statement. The assembly is much shorter and closer to what a person would write
 - **Compared to `-O1` / `-O2`:** passes that reorder or restructure code heavily (aggressive inlining, loop transformations, instruction scheduling, and so on) are left off. The result still follows the structure of your C source almost line by line, so stepping through it in gdb makes sense.
- `-S` stops after compilation and writes `hello.s`
- **NOTE! This is x86-64 assembly in AT&T syntax, but on a Mac, `gcc` actually runs Clang**
 - Output doesn't look like GNU GCC
 - `_main` and `_printf` have a leading underscore, which is the macOS naming convention for C symbols
 - Comments also start with `##`, among other subtle differences

ASM hello.s × C hello.c ×

```

1      .section      __TEXT,__text,regular,pure_instructions
2      .build_version macos, 10, 15, 4 sdk_version 10, 15, 4
3      .section      __TEXT,__literal8,8byte_literals
4      .p2align      3                      ## -- Begin function main
5      LCPI0_0:
6      .quad          4612811918334230528      ## double 2.5
7      .section      __TEXT,__text,regular,pure_instructions
8      .globl         _main
9      .p2align      4, 0x90
10     _main:                      ## @main
11     .cfi_startproc
12     ## %bb.0:
13     pushq           %rbp
14     .cfi_def_cfa_offset 16
15     .cfi_offset %rbp, -16
16     movq            %rsp, %rbp
17     .cfi_def_cfa_register %rbp
18     leaq            L_.str(%rip), %rdi
19     movsd           LCPI0_0(%rip), %xmm0      ## xmm0 = mem[0],zero
20     movb            $1, %al
21     callq           _printf
22     leaq            L_.str(%rip), %rdi
23     callq           _puts
24     xorl            %eax, %eax
25     popq            %rbp
26     retq
27     .cfi_endproc
28     |               |               |               |               |               |               ## -- End function
29     .section      __TEXT,__cstring,cstring_literals
30     L_.str:                      ## @.str
31     .asciz          "%6.3f \n"
32
33     L_str:                      ## @str
34     .asciz          "hello, world!\t!"
35
36
37     .subsections_via_symbols
38

```

Example x86-64 Assembly

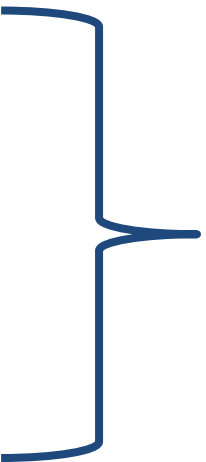
```
.text
.globl multstore
.type multstore, @function

# multiply and store to memory
multstore:
    pushq %rbx # save to stack
    movq %rdx, %rbx
    call mult2
    movq %rax, (%rbx)
    popq # restore from stack
    ret
```

Example x86-64 Assembly

```
.text
.globl multstore
.type multstore, @function

# multiply and store to memory
multstore:
    pushq %rbx # save to stack
    movq %rdx, %rbx
    call mult2
    movq %rax, (%rbx)
    popq # restore from stack
    ret
```



Various assembly instructions

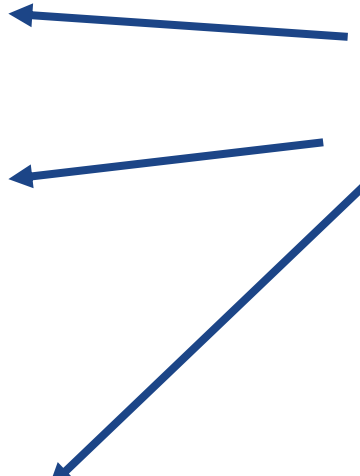
Example x86-64 Assembly

```
.text  
.globl multstore  
.type multstore, @function
```

```
# multiply and store to memory  
multstore:
```

```
    pushq %rbx # save to stack  
    movq %rdx, %rbx  
    call mult2  
    movq %rax, (%rbx)  
    popq # restore from stack  
    ret
```

Comments use the
symbol



Example x86-64 Assembly

```
.text
.globl multstore
.type multstore, @function

# multiply and store to memory
multstore:
    pushq %rbx # save to stack
    movq %rdx, %rbx
    call mult2
    movq %rax, (%rbx)
    popq # restore from stack
    ret
```

Labels are arbitrary names that mark a section of code

We'll get back to these later

Example x86-64 Assembly

```
.text
.globl multstore
.type multstore, @function
```



Assembler directives
(mostly ignore these)

```
# multiply and store to memory
multstore:
```

```
    pushq %rbx # save to stack
    movq %rdx, %rbx
    call mult2
    movq %rax, (%rbx)
    popq # restore from stack
    ret
```

Can be used to
specify data versus
code regions, make
functions linkable
with other code,
and many other
tasks.

x86-64 Instructions

- General Instruction Syntax:

`op src, dst`

- 1 operator, 2 operands
 - `op` = operation name (“operator”)
 - `src` = source location (“source”)
 - `dst` = destination location (“destination”)
- Keep hardware simple via regularity

Careful! Two Syntaxes for Assembly

Intel/Microsoft Format

```
lea    eax,[ecx+ecx*2]
sub     esp,8
cmp     dword ptr [ebp-8],0
mov     eax,dword ptr [eax*4+100h]
```

ATT Format

```
leal    (%ecx,%ecx,2),%eax
subl    $8,%esp
cmpl    $0,-8(%ebp)
movl    $0x100(,%eax,4),%eax
```

- Intel/Microsoft mnemonics vs. ATT
 - Operands listed in opposite order: `mov Dest, Src` vs. `movl Src, Dest`
 - Constants not preceded by '\$', Denote hex with 'h' at end: `100h` vs. `$0x100`
 - Operand size indicated by operands rather than operator suffix: `sub` vs. `subq`
 - Addressing format shows effective address computation: `[eax*4+100h]` vs. `$0x100(,%rax,4)`
- `gcc (gas), gdb, objdump` work on the ATT format
 - We will **always** use the ATT format as well

Example x86-64 Assembly

```
.text
.globl multstore
.type multstore, @function

# multiply and store to memory
multstore:
    pushq %rbx # save to stack
    movq %rdx, %rbx
    call mult2
    movq %rax, (%rbx)
    popq # restore from stack
    ret
```

← What might this instruction do?

(op src, dst)

Topics

- Registers
- **x86-64 Assembly**
 - Introduction
 - **Move Instruction**
 - Memory Addressing Modules

Three Basic Kinds of Instructions

1. Transfer data between memory and register

- *Load* data from memory into register

- `%reg = Mem[address]`

- *Store* register data into memory

- `Mem[address] = %reg`

Remember: Memory is indexed just like an array of bytes!

2. Perform arithmetic operation on register or memory data

- `c = a + b;` `z = x << y;` `i = h & g;`

3. Control flow: what instruction to execute next

- Unconditional jumps to/from procedures
- Conditional branches

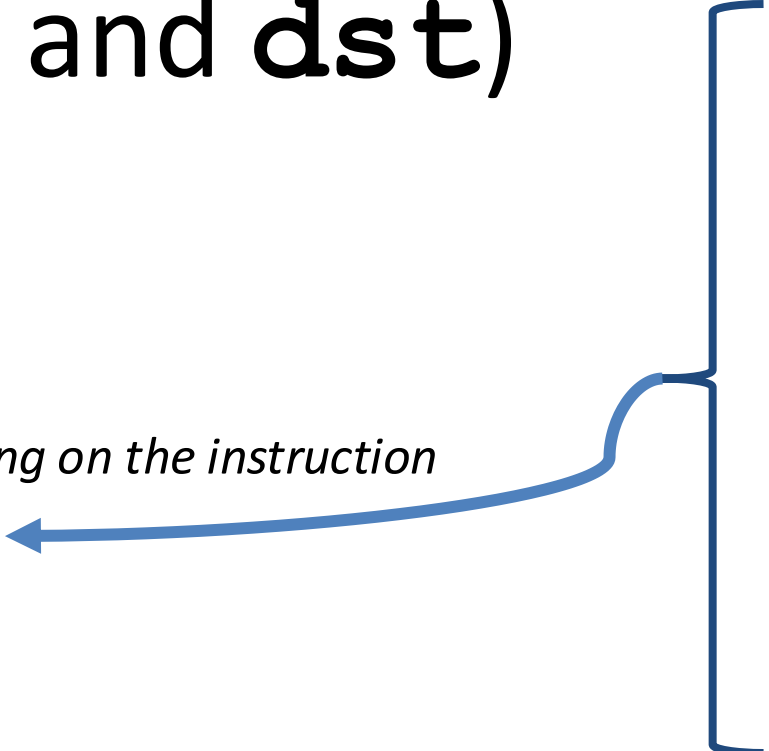
Copying Data with the “move” instruction

- General form: `mov_ source, destination`
 - Very common instruction in typical code. Copies value into destination location
 - Missing letter `_` specifies size of operands
 - Reminder: backwards compatibility means “word” = 16 bits
 - `movl src, dst`
- `movb src, dst`
 - Move 1-byte “byte”
 - `movq src, dst`
 - Move 4-byte “long word”
- `movw src, dst`
 - Move 2-byte “word”
 - `movq src, dst`
 - Move 8-byte “quad word”
 - Native size for x86-64

Note: Instructions *must* be used with properly-sized register names
And they’re also a helpful hint about the sizes being used

Operand Types (**src** and **dst**)

- **Immediate:** Constant integer data
 - Examples: **\$0x400**, **\$-533**
 - Like C literal, but prefixed with **`\$'**
 - Encoded with 1, 2, 4, or 8 bytes *depending on the instruction*
- **Register:** 1 of 16 integer registers
 - Examples: **%rax**, **%r13**
 - But **%rsp** reserved for special use
- **Memory:** Consecutive bytes of memory at a computed address
 - Simplest example: **(%rax)** treats value of **%rax** as an address → access memory
 - Various other “address modes” we’ll talk about later



%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN (r8-r15)

MOV Operand Combinations

***immediate can
also never be a
destination, since you
can't write into a
constant!*

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	var_a = 0x4;
		Mem	movq \$-147, (%rax)	*p_a = -147;
	Reg	Reg	movq %rax, %rdx	var_d = var_a;
		Mem	movq %rax, (%rdx)	*p_d = var_a;
	Mem	Reg	movq (%rax), %rdx	var_d = *p_a;

Cannot do memory-memory transfer with a single instruction

- **How would you do it?**

MOV Operand Combinations

***immediate can
also never be a
destination, since you
can't write into a
constant!*

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	var_a = 0x4;
		Mem	movq \$-147, (%rax)	*p_a = -147;
	Reg	Reg	movq %rax, %rdx	var_d = var_a;
		Mem	movq %rax, (%rdx)	*p_d = var_a;
	Mem	Reg	movq (%rax), %rdx	var_d = *p_a;

Cannot do memory-memory transfer with a single instruction

- **How would you do it?**

```
movq (%rax), %rdx #load
movq %rdx, (%rbx) #store
```

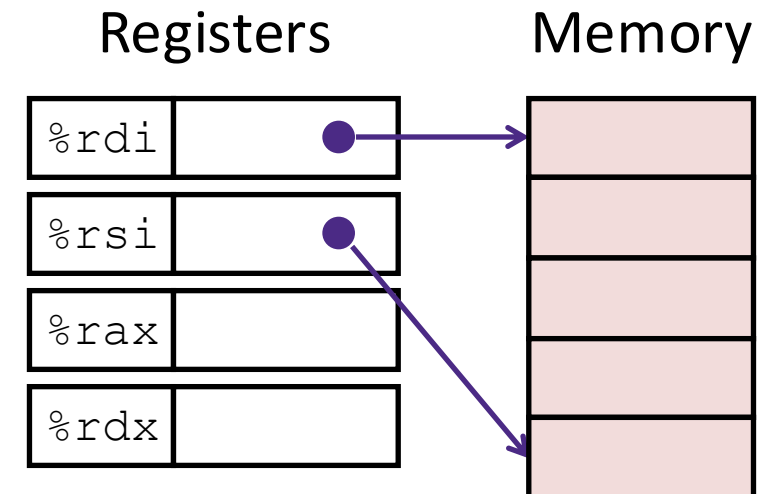
Example of Move Instructions: swap()

```
void swap(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

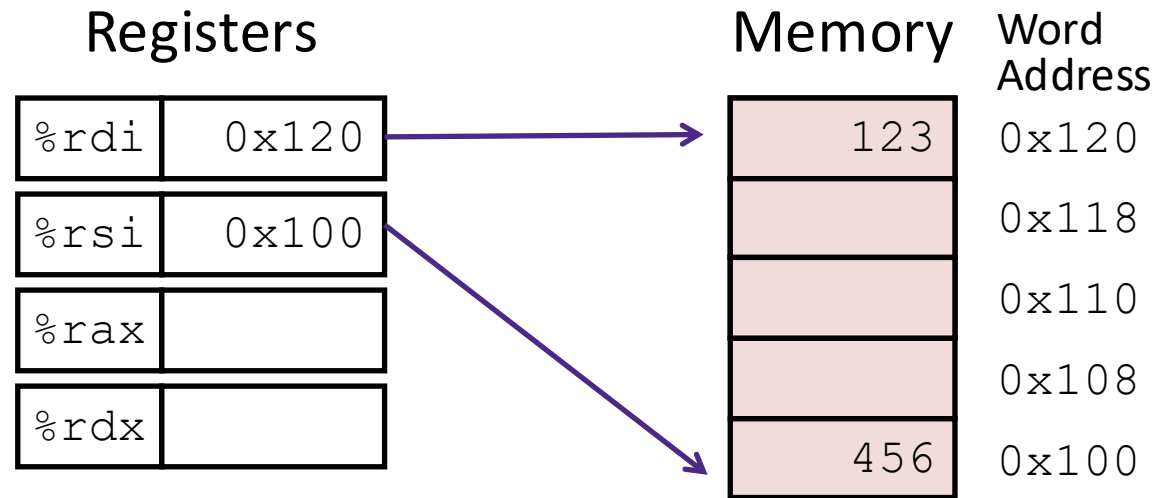
<u>Register</u>		<u>Variable</u>
%rdi	↔	xp
%rsi	↔	yp
%rax	↔	t0
%rdx	↔	t1

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Initial state:



Example of Move Instructions: swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Example of Move Instructions: swap()

Registers		Memory	Word Address
%rdi	0x120	123	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx			0x108
		456	0x100

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Example of Move Instructions: swap()

Registers		Memory	Word Address
%rdi	0x120	123	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx	456		0x108
		456	0x100

```
swap:
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Example of Move Instructions: swap()

Registers		Memory	Word Address
%rdi	0x120	456	0x120
%rsi	0x100		0x118
%rax	123		0x110
%rdx	456		0x108
		456	0x100

```
swap:
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Example of Move Instructions: swap()

Note: these did not change

Registers	
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory	Word Address
456	0x120
	0x118
	0x110
	0x108
123	0x100

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)   # *yp = t0
ret
```

Question

- How does the number of available registers affect a system?
 - What if x86-64 only had two registers?
 - What if x86-64 instead had 512 registers?

Question

- How does the number of available registers affect a system?
 - What if x86-64 only had two registers?
 - “Register Pressure” becomes a problem
 - Accessing 3+ things at once requires memory
 - Way more memory reads/writes
 - What if x86-64 instead had 512 registers?
 - Most of the registers would never be used
 - For any realistic program
 - Could have spent that silicon on something more important

Topics

- Registers
- **x86-64 Assembly**
 - Introduction
 - Move Instruction
 - Memory Addressing Modules

Memory Addressing Modes: Basic

- Common need: interact with memory
 - Exact address might be made of multiple parts
- **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:**

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

- **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

The scale factors match the sizes in bytes of the common C types!

- **Special cases** (see textbook Figure 3.3 or next slide)

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

Full list of addressing mode forms

Type	Form	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	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$

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

Expression	Address Computation	Address
0x8(%rdx)		
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$

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

Expression	Address Computation	Address
0x8 (%rdx)	%rdx + 0x8	0xf008
(%rdx, %rcx)		
(%rdx, %rcx, 4)		
0x80(, %rdx, 2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$

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

Expression	Address Computation	Address
0x8 (%rdx)	%rdx + 0x8	0xf008
(%rdx, %rcx)	%rdx + %rcx*1	0xf100
(%rdx, %rcx, 4)		
0x80(, %rdx, 2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$

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

Expression	Address Computation	Address
0x8 (%rdx)	%rdx + 0x8	0xf008
(%rdx, %rcx)	%rdx + %rcx*1	0xf100
(%rdx, %rcx, 4)	%rdx + %rcx*4	0xf400
0x80(, %rdx, 2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$

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

Expression	Address Computation	Address
0x8 (%rdx)	%rdx + 0x8	0xf008
(%rdx, %rcx)	%rdx + %rcx*1	0xf100
(%rdx, %rcx, 4)	%rdx + %rcx*4	0xf400
0x80(, %rdx, 2)	%rdx*2 + 0x80	0x1e080

What is the final value in %rdx?

Registers		Memory	Word Address
%rdi	0x100	456	0x120
%rsi	2	378	0x118
%rax	37	568	0x110
%rdx	15	224	0x108
		123	0x100

```
swap:
    movq    (%rdi,%rsi,8), %rax
    movq    %rax, %rdx
```

D (Rb, Ri, S) →

Mem[Reg[Rb]+Reg[Ri]*S+D]

What is the final value in %rdx?

Registers		Memory	Word Address
%rdi	0x100	456	0x120
%rsi	2	378	0x118
%rax	568	568	0x110
%rdx	15	224	0x108
		123	0x100

```
swap:  
    movq    (%rdi,%rsi,8), %rax  
    movq    %rax, %rdx
```

Address: $0x100 + 2 * 8 = 0x110$

Memory at address: 568

Sets %rax to 568

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$

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

What is the final value in %rdx?

Registers		Memory	Word Address
%rdi	0x100	456	0x120
%rsi	2	378	0x118
%rax	568	568	0x110
%rdx	568	224	0x108
		123	0x100

```
swap:  
    movq    (%rdi,%rsi,8), %rax  
    movq    %rax, %rdx
```

Copies value from %rax to %rdx

Sets %rdx to 568

Final value in %rdx: 568

$D(\text{Rb}, \text{Ri}, \text{S}) \rightarrow$
 $\text{Mem}[\text{Reg}[\text{Rb}] + \text{Reg}[\text{Ri}] * \text{S} + D]$