Lecture 05
Intro to x86-64 Assembly

CS213 – Intro to Computer Systems

Branden Ghena – Fall 2023

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Pack Lab
  - Due this Tuesday (10/10) at 11:59 pm

  - If you haven't yet, **get started right away**!
    - Lots of conceptually difficult parts to understand
    - Lots of C code to write, which you may have forgotten

  - Especially make sure you don't have issues logging into Moore
    - Takes ~24 hours to fix and we won't be giving extensions for it

  - This assignment is 12.5% of your overall course grade, and takes at least that much effort
    - Not an "easier first assignment"

Administrivia

- Homework 2
  - Ready to be worked on (except the last question)
  - Not due for 1.5 weeks (Tuesday, October 17th)

- Midterm exam
  - Exactly two weeks from today
  - Taken here in the classroom
  - Details next week

Today's Goals

- Introduce assembly and the x86-64 Instruction Set Architecture
  - Discuss background of the factors that affected its evolution

- Understand registers: the analogy to variables in assembly

- Explore our first assembly instruction: **`mov`**

Outline

- **Assembly Languages**

- Registers

- x86-64 Assembly
  - Introduction
  - Move Instruction
  - Memory Addressing Modes

Assembly (Also known as: Assembly Language, ASM)

- Purpose of a CPU: execute instructions

- High-level programs (like in C) are split into many small instructions

- Assembly is a low-level programming language where the program instructions match a particular architecture's operations
  - Assembly is a human-readable text representation of machine code
  - Each assembly instruction is one machine instruction (usually)

Programs can be written in assembly or machine instructions

C Program (source code)

```
a = (b+c)-(d+e);
```

Assembly Program

```
addq %rdi, %rsi
addq %rdx, %rcx
subq %rcx, %rsi
movq %rsi, %rax
```

Machine Instructions

```
0x4889D3
0x488903
0x53
0x5B
```

There are many assembly languages

- Instruction Set Architecture: All programmer-visible components of a processor needed to write software for it
  - Operations the processor can execute
  - The system’s state (registers, memory, program counter)
  - The effect operations have on system state

- Each assembly language has instructions that match a particular processor’s Instruction Set Architecture (ISA)

- Assembly is not portable to other architectures (like C is)

Which instructions should an assembly include?

Each assembly language has its own operations

There are some obviously useful instructions:

- Add, subtract, and bit shift
- Read and write memory

But what about:

- Only run the next instruction if these two values are equal
- Perform four pairwise multiplications simultaneously
- Add two ascii numbers together (‘2’ + ‘3’ = 5)

Instruction Set Philosophies

Early trend: add more instructions to do elaborate operations
Complex Instruction Set Computing (CISC)

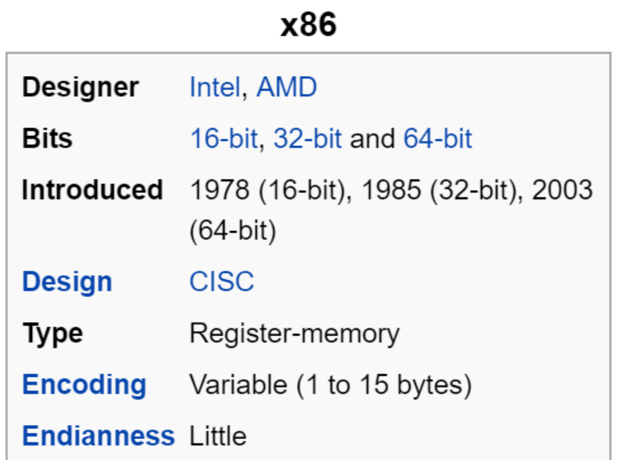
- Handle many different types of operations
- More options for the compiler
- Complicated hardware runs more slowly

Opposite philosophy later began to dominate:
Reduced Instruction Set Computing (RISC)

- Simpler (and smaller) instruction set makes it easier to build fast hardware
- Let software do the complicated operations by composing simpler ones

Modern reality is somewhere between these two

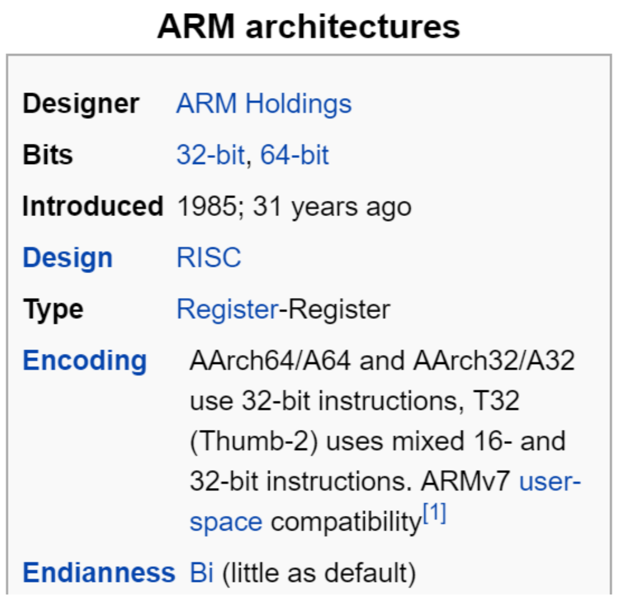
Mainstream Instruction Set Architectures



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

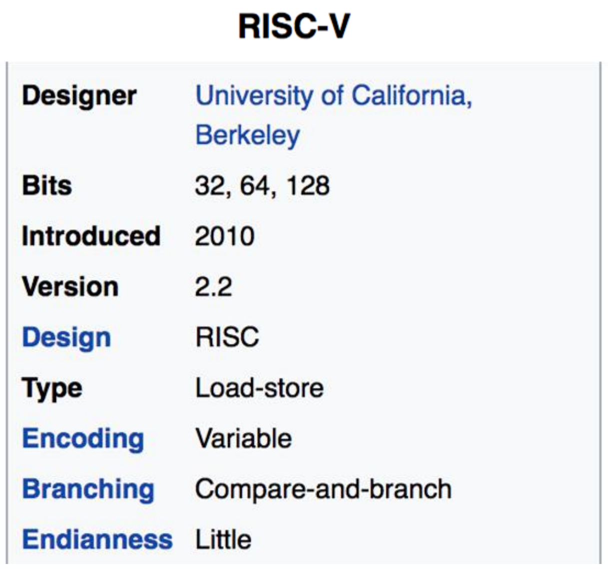
Macbooks & PCs
(Core i3, i5, i7, M)
[x86 Instruction Set]



ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphones (iPhone, Android),
M1 Macbooks, Raspberry Pi,
Embedded systems
[ARM Instruction Set]

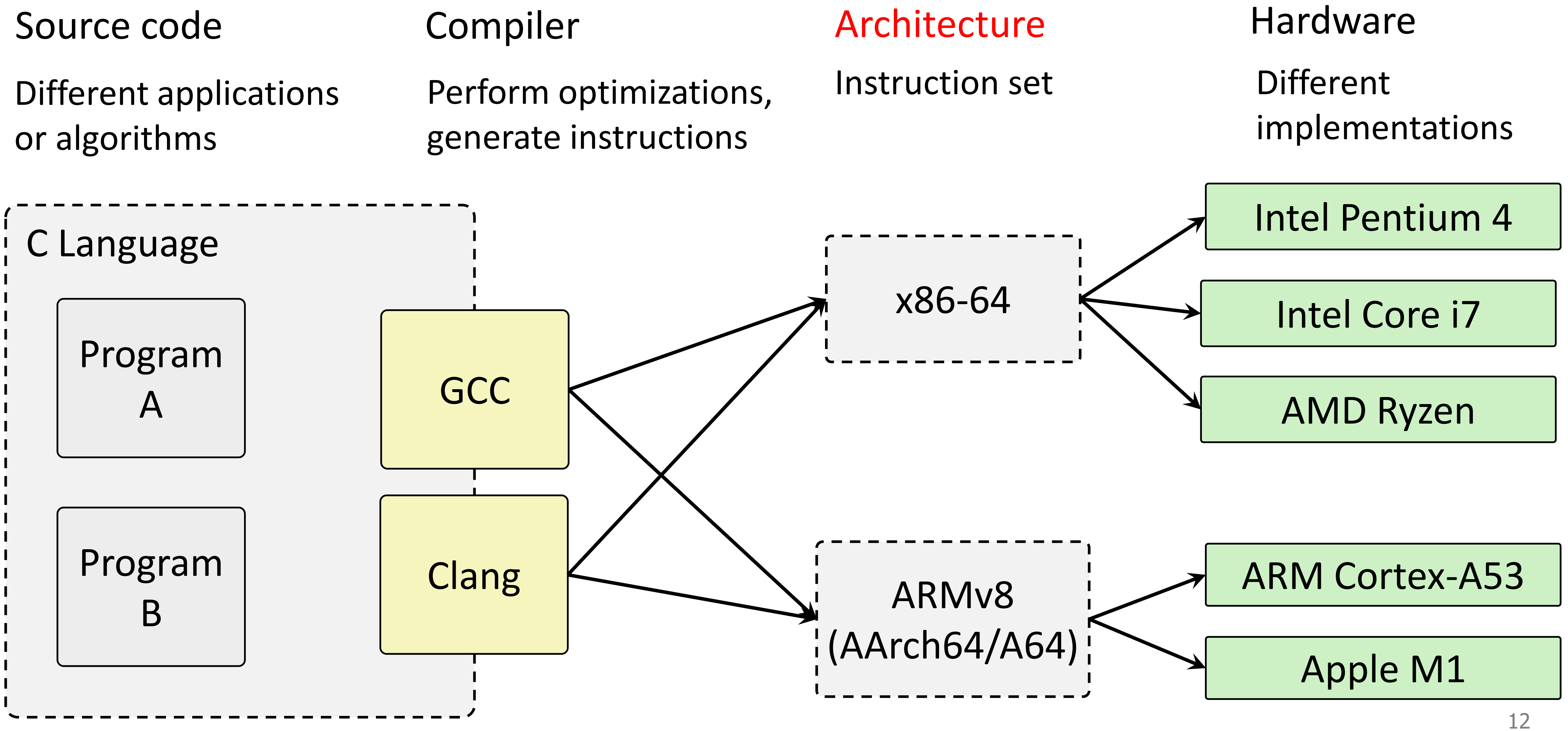


RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

Open-source
Relatively new, designed for cloud computing, embedded systems, academic use
[RISCV Instruction Set]

Instruction Set Architecture sits at software/hardware interface



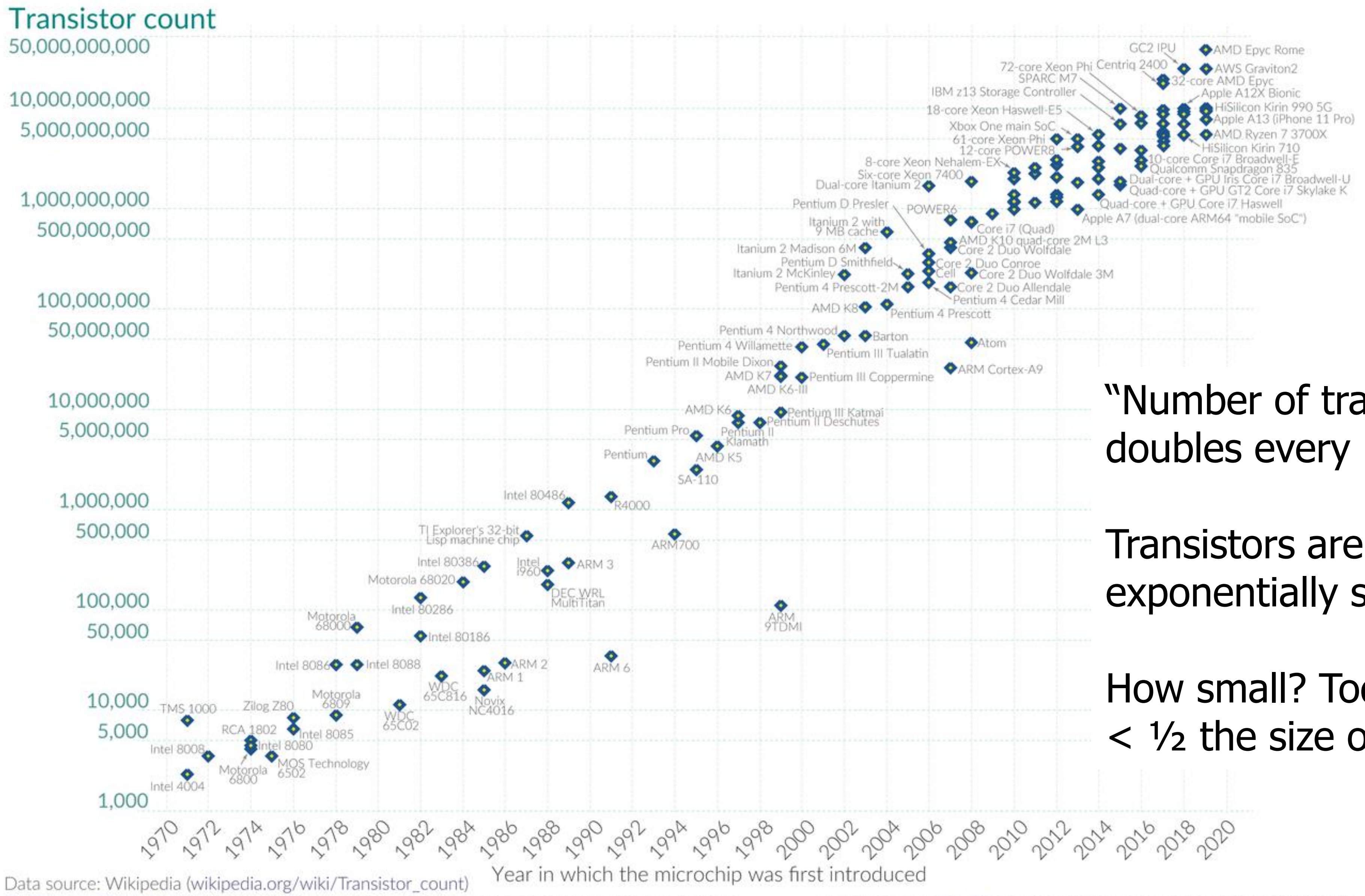
Intel x86 Processors

- Dominate laptop/desktop/server market
  - No longer completely dominant in laptops though

- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
  - But, only small subset encountered by normal programs

- Design evolved over time
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
  - Historical legacy has **large** impact on architecture

Moore’s Law – CPU transistors counts



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world’s largest problems.

“Number of transistors in a chip doubles every 18 months”

Transistors are getting exponentially smaller!

How small? Today: 3nm!
< ½ the size of most viruses!

Evolution of x86 ISA

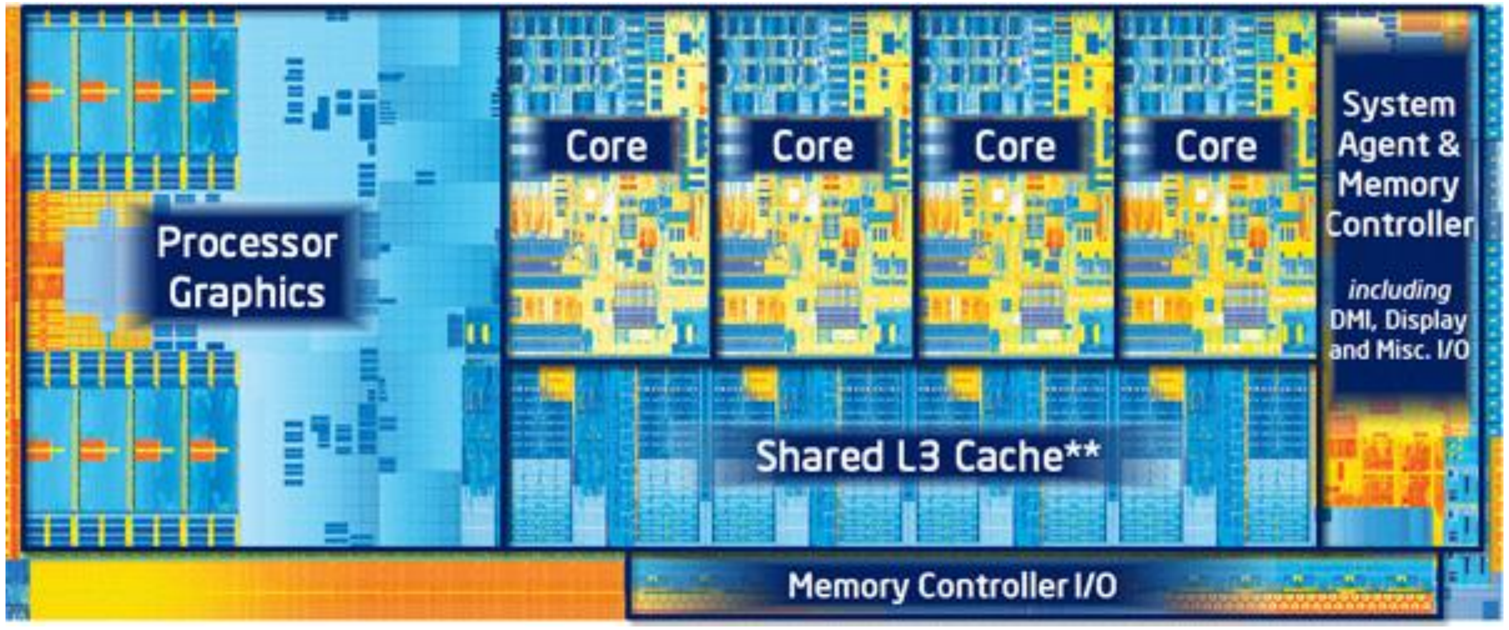
Name	Date	Transistors	Comments
8086	1978	29k	**16b** processor, basis for IBM PC & DOS; 1MB address space
80286	1982	134K	Elaborate (!useful) addressing; basis for IBM PC and Windows
386	1985	275K	Extended to **32b**, added "flat addressing" that Linux/gcc uses
486	1989	1.9M	Improved performance; **integrated FP unit into chip**
Pentium	1993	3.1M	Improved performance
PentiumPro	1995	6.5M	**Conditional move instructions**; big change in microarch. (P6)
Pentium II	1997	7M	Merged Pentium/MMZ + PentiumPro, **MMX** instructions within P6
Pentium III	1999	8.2M	Integer and floating point vector instructions (**SSE**); Level2 cache
Pentium 4	2001	42M	8B ints and floating point formats to vector instructions
Pentium 4E	2004	125M	**Hyperthreading** (able to run 2 programs simultaneously), **64b**
Core 2	2006	291M	P6-like, **multicore**, no hyperthreading
Core i7 (Nehalem)	2008	781M	Hyperthreading + multicore, TurboBoost (run fewer cores faster)
Core i3 (Nehalem)	2010	383M+177M	GPU on second silicon die within package (at 2010 version)
Core i3, i5, i7 (Sandy Bridge)	2011	997M (i7 – 4 cores)	**Cores and GPU within the same processor die**
Core i3, i5, i7 (Ivy Bridge)	2012	1400M (i7 – 4 cores)	Tri-gate transistors, much lower power consumption
Xeon E7 8800 V4 (Broadwell-EX)	2016	>5690M (22 cores)	14nm technology

Backwards Compatibility The cause of, and solution to, all of life’s problems.

- Programs that worked on one x86 processor should keep working on the next one
  - Old programs work on new processors, which makes upgrading possible
  - Even today’s x86-64 processors boot thinking they are 8086s!
- Adding powerful new features while keeping backwards compatibility is a careful balancing act
  - Backwards compatibility introduces a lot of constraints
  - May rule out “cleaner” designs that would break existing programs
  - The cause of some “surprising” aspects of the design of x86-64
  - “The x86 really isn't all that complex—it just doesn't make a lot of sense.”
    — Mike Johnson (AMD's x86 architect), 1994
- Not just a hardware thing!

In this class

- x86-64/EMT64: the current standard
  - Some asides on IA32: The traditional x86
- Presentation
  - Book covers x86-64; web aside on IA32
  - Labs will be based on x86-64

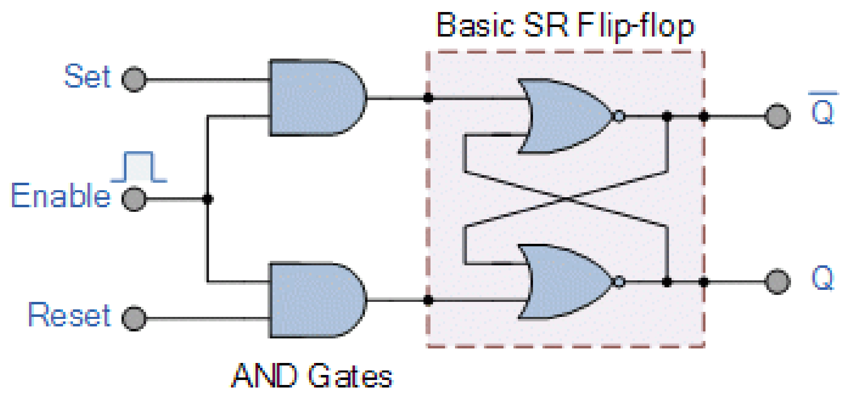


Outline

- Assembly Languages
- **Registers**
- x86-64 Assembly
    - Introduction
    - Move Instruction
    - Memory Addressing Modes

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
    - 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

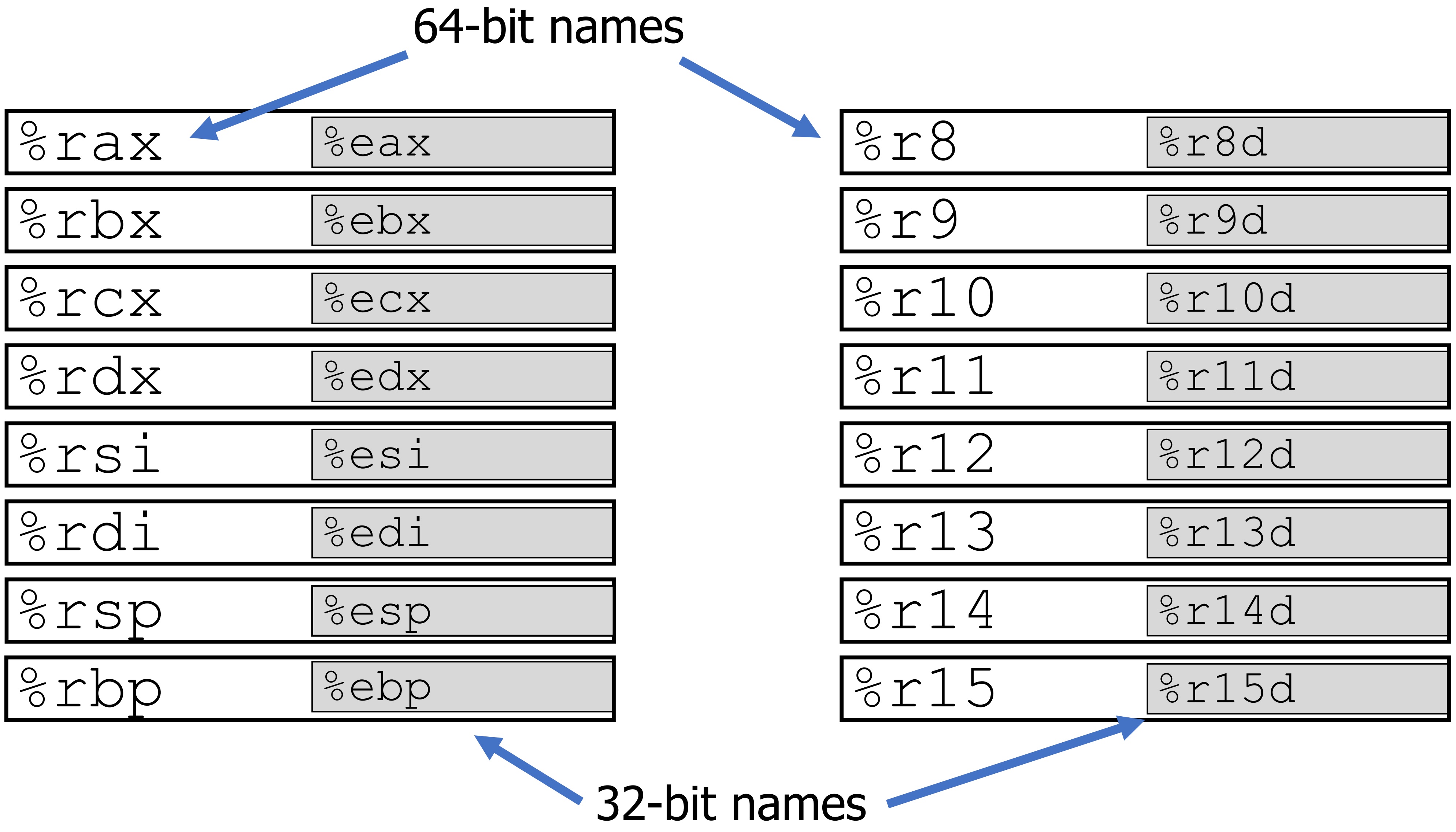
- x86-64 has 16 registers (for integer operations)
  - Historically only 8 registers 😱
  - Added 8 more with 64-bit extensions

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

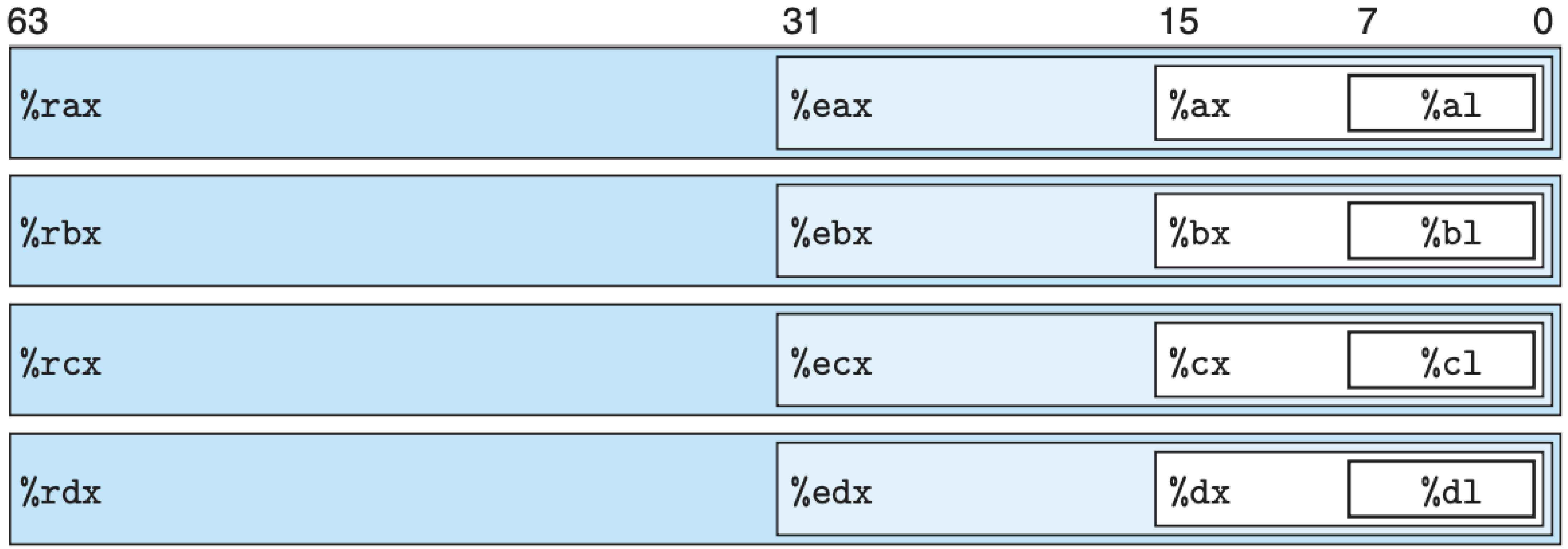
x86-64 Registers



Historical Register Purposes

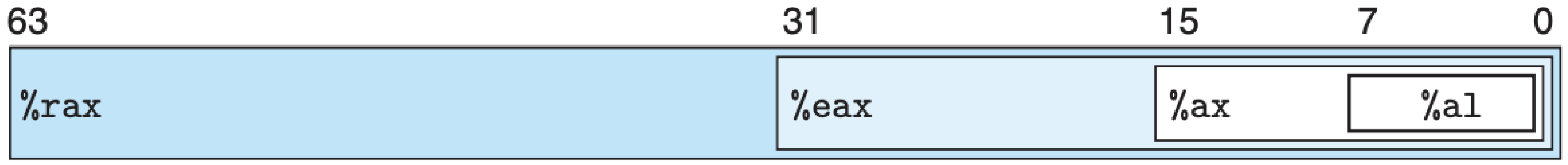
		Name Origin (mostly obsolete)
`%rax`	`%eax`	Accumulate
`%rbx`	`%ebx`	Base
`%rcx`	`%ecx`	Counter
`%rdx`	`%edx`	Data
`%rsi`	`%esi`	Source Index
`%rdi`	`%edi`	Destination Index
`%rsp`	`%esp`	Stack Pointer (still important)
`%rbp`	`%ebp`	Base Pointer

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

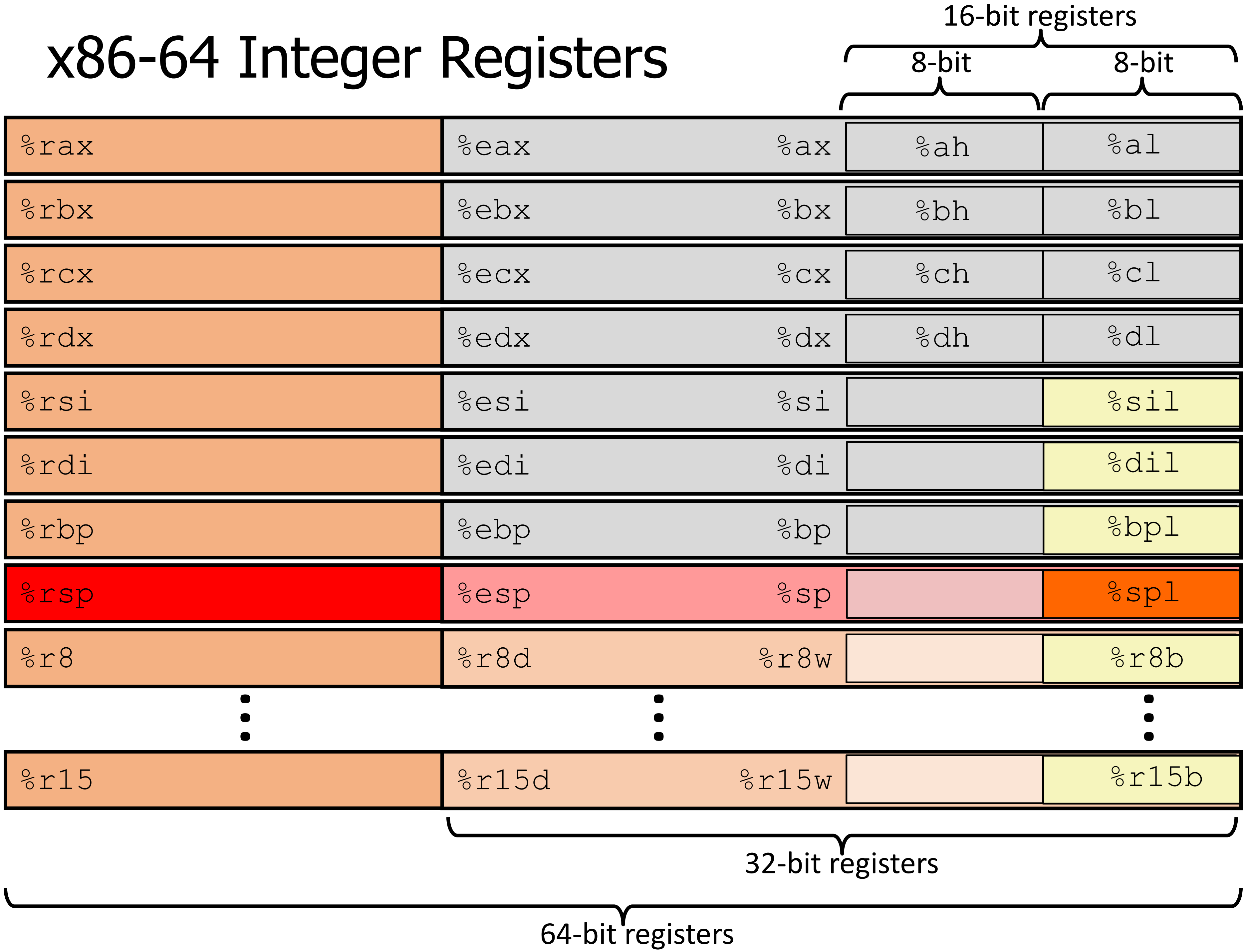
x86-64 Register Access Options



- The same data can be accessed under different names and widths

- Example: store 64-bit value 0x000000000010A0B in %rax
  - %rax: 0x000000000010A0B (64-bit)
  - %eax: 0x00010A0B (32-bit)
  - %ax: 0x0A0B (16-bit)
  - %al: 0x0B (8-bit)

x86-64 Integer Registers

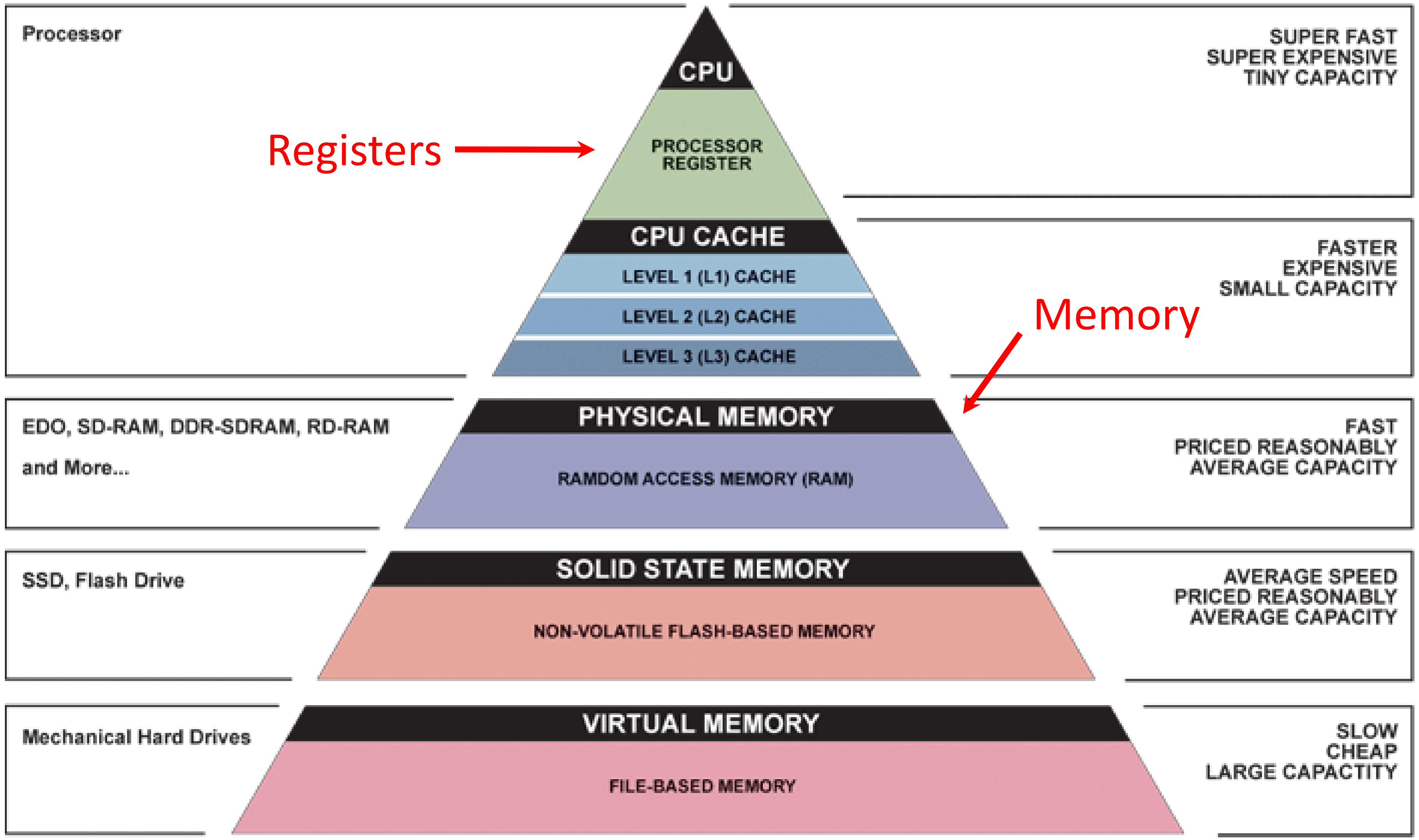


Registers versus Memory

- What if more variables than registers?
  - Keep most frequently used in registers and move the rest to memory (called *spilling* to memory)

- Why not 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
    - SSD: 100-1000 GB

Memory Hierarchy



Special-purpose register: Instruction Pointer

- Instruction Pointer: `%rip`
  - Contains the address of the currently executing instruction
  - Actually 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

Break + Question

Which of these 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

Break + Question

Which of these 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~~]

There are a fixed number of registers for a given architecture

Outline

- Assembly Languages

- Registers

- **x86-64 Assembly**
  - **Introduction**
  - Move Instruction
  - Memory Addressing Modes

Writing Assembly Code? In 2023???

- 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

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
```

$\Bigg\}$ 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

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

$\}$ Assembler directives (mostly ignore these)

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”)
    - `src1` = 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

Short Break + 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`)

Outline

- Assembly Languages
- Registers
- **x86-64 Assembly**
  - Introduction
  - **Move Instruction**
  - Memory Addressing Modes

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

Moving Data

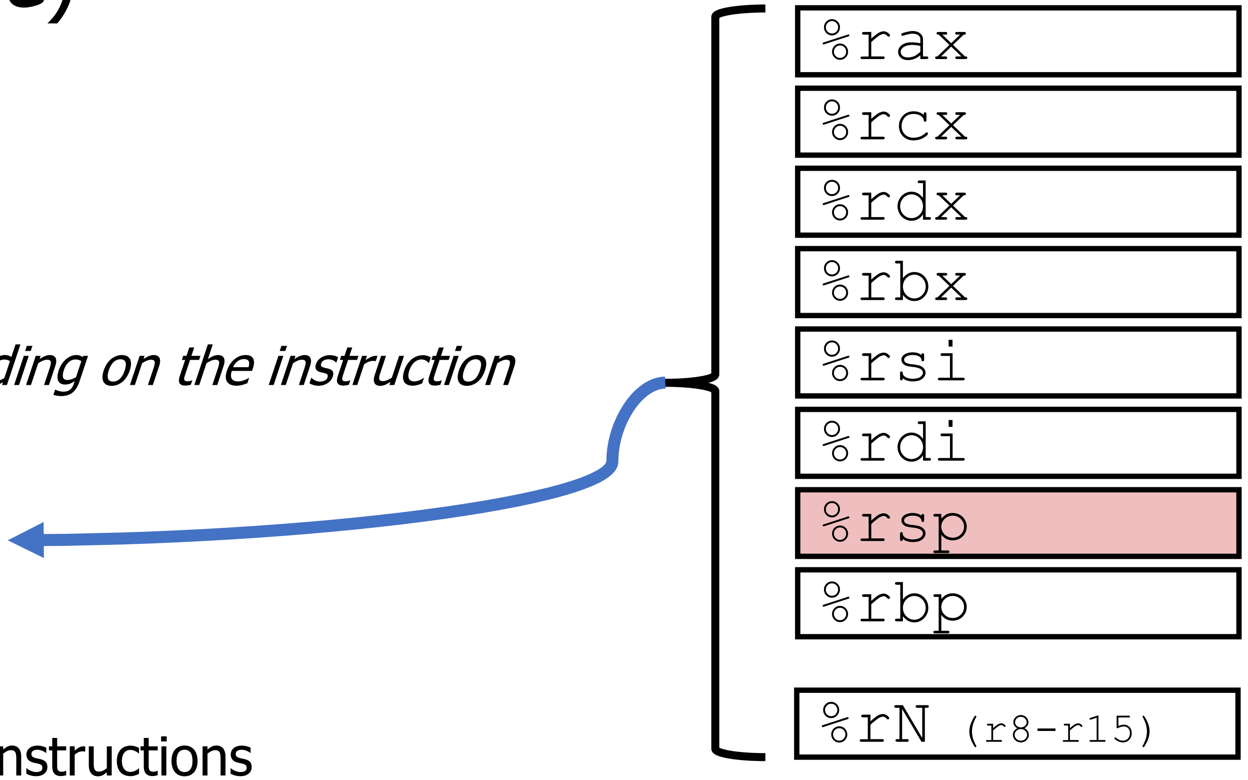
- General form: `mov_ source, destination`
  - Missing letter _ specifies size of operands
  - Reminder: backwards compatibility means “word” = 16 bits
  - Lots of these in typical code

- `movb src, dst`
  - Move 1-byte “**b**yte”
- `movw src, dst`
  - Move 2-byte “**w**ord”
- `movl src, dst`
  - Move 4-byte “**l**ong word”
- `movq src, dst`
  - Move 8-byte “**q**uad word”
  - Native size for x86-64

Note: Instructions *must* be used with properly-sized register names

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
  - Others have special uses for particular instructions
- ***Memory:*** Consecutive bytes of memory at a computed address
  - Simplest example: **`(%rax)`** treats value of %rax as an address $\longrightarrow$ access memory
  - Various other "address modes" we'll talk about later



MOV Operand Combinations

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

	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

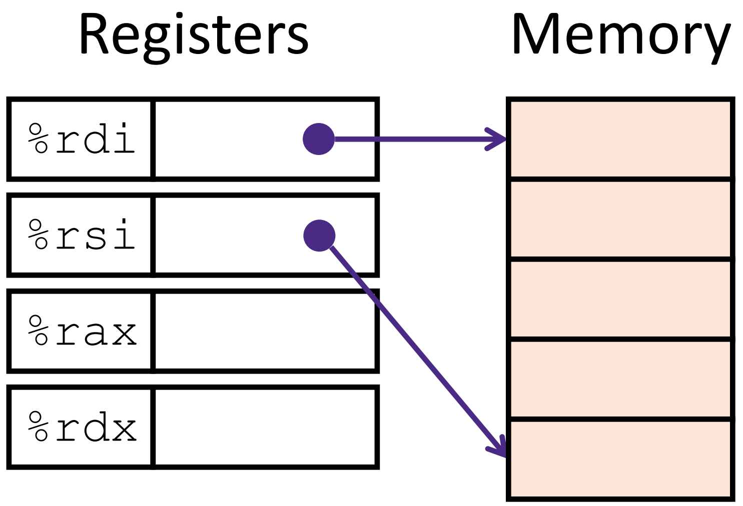
- **How would you do it?** 1) Mem->Reg, 2) Reg->Mem

Example of Move Instructions: swap()

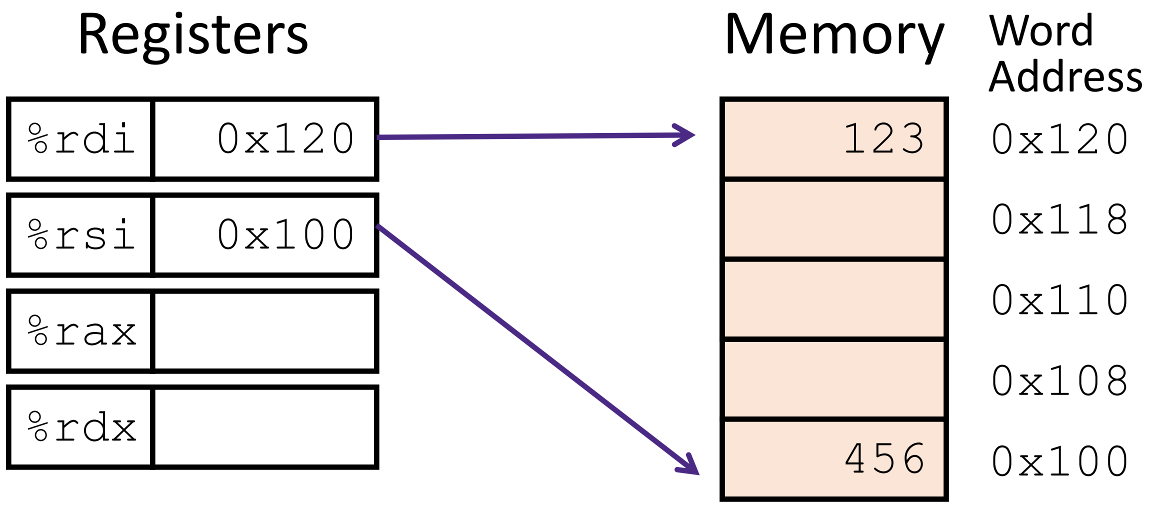
```
void swap(long* xp, long* yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

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

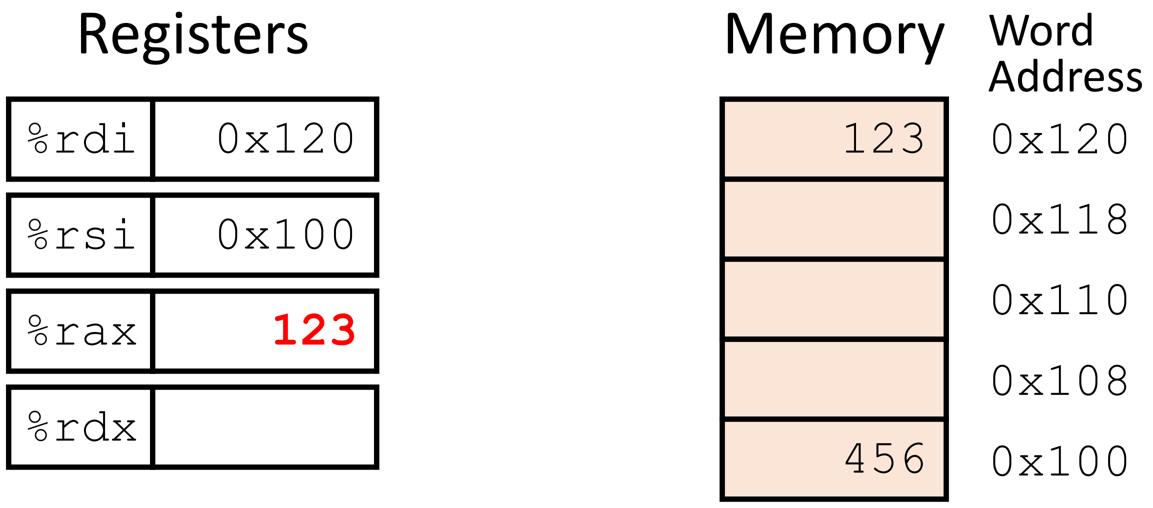


Example of Move Instructions: swap()



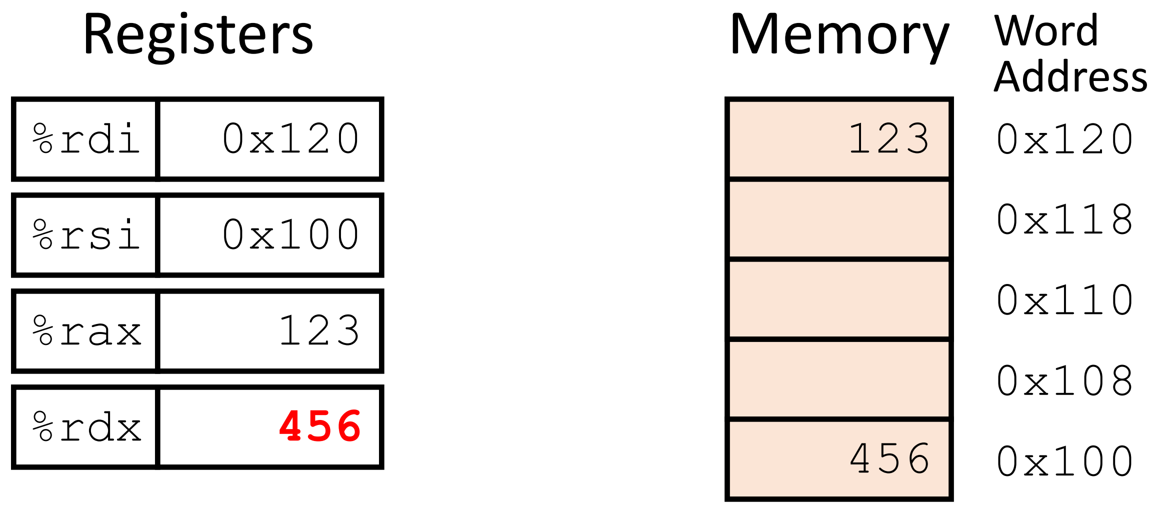
```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Example of Move Instructions: swap()



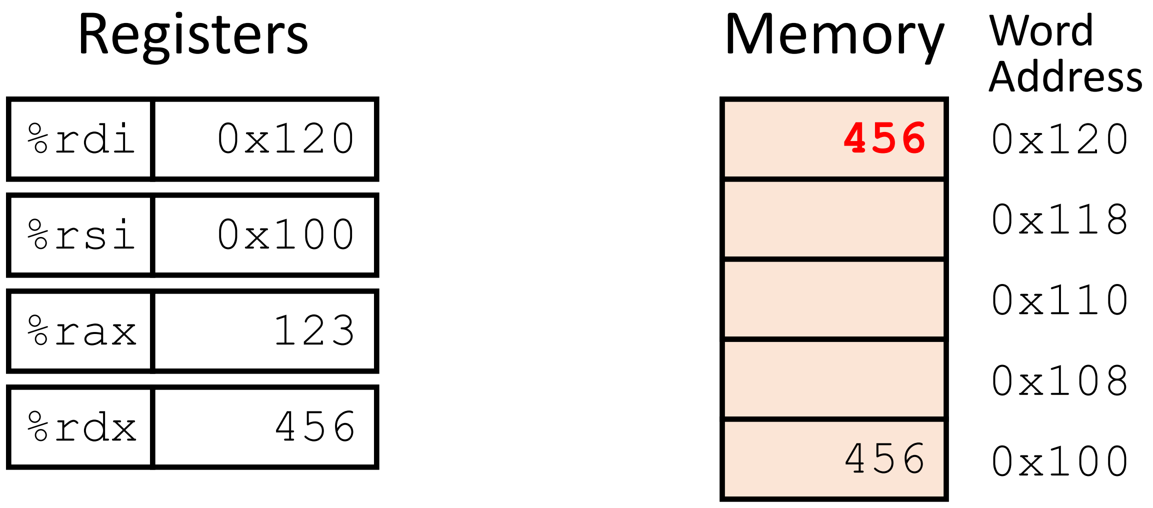
```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Example of Move Instructions: swap()



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

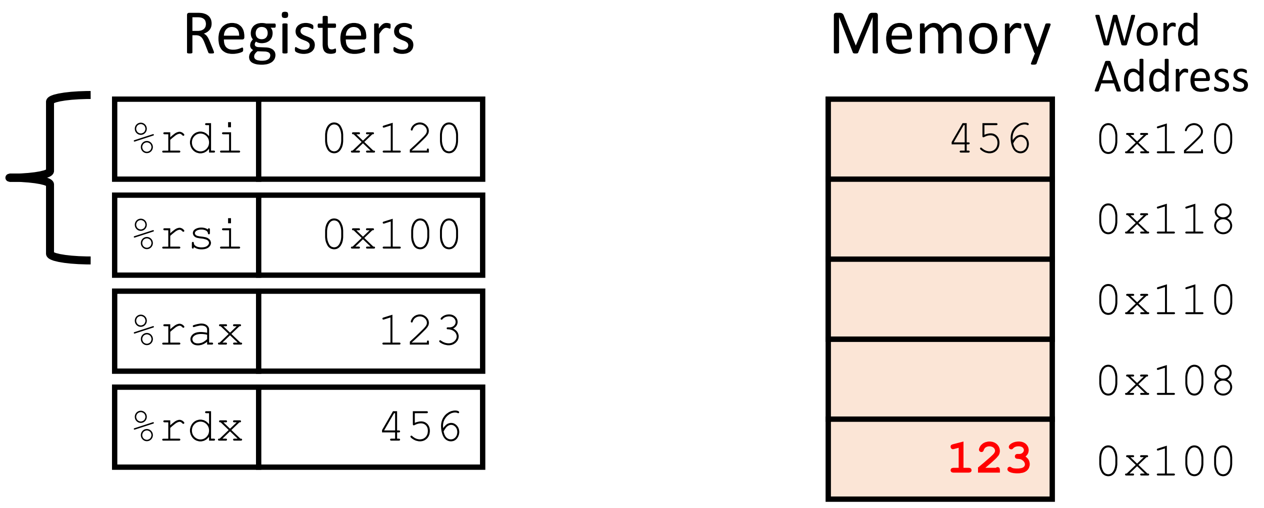
Example of Move Instructions: swap()



```
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



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

Break + Open 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?

Break + Open 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

Outline

- Assembly Languages
- Registers
- **x86-64 Assembly**
    - Introduction
    - Move Instruction
    - **Memory Addressing Modes**

Memory Addressing Modes: Basic

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

- **Indirect:** `(R)` Mem[Reg[`R`]]
  - Data in register `R` specifies the memory address
  - Like pointer dereference in C
  - Example: **`movq`** `(%rcx), %rax`

- **Displacement:** `D(R)` Mem[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

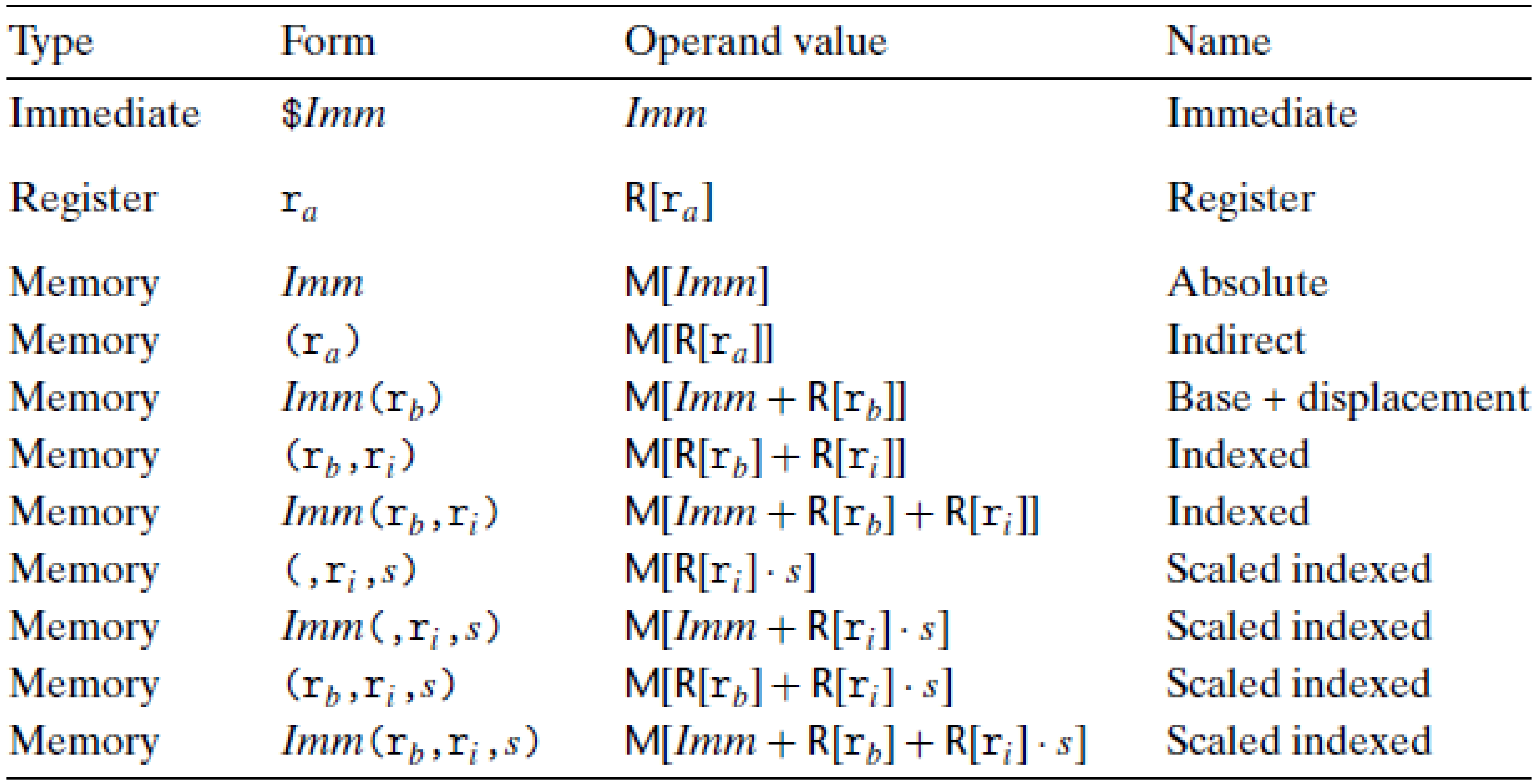
- **General:**
  - `D(Rb,Ri,S)` Mem[Reg[`Rb`]+Reg[`Ri`]`*S+D`]
    - `Rb`: Base register (any register)
    - `Ri`: Index register (any register except `%rsp`)
    - `S`: Scale factor (1, 2, 4, 8) – *why these numbers?*
    - `D`: Constant displacement value (a.k.a. immediate)

Sizes of common C types!

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

`D(Rb,Ri)`	Mem[Reg[`Rb`]+Reg[`Ri`]`+D`]	`(S=1)`
`(Rb,Ri,S)`	Mem[Reg[`Rb`]+Reg[`Ri`]`*S`]	`(D=0)`
`(Rb,Ri)`	Mem[Reg[`Rb`]+Reg[`Ri`]]	`(S=1,D=0)`
`(,Ri,S)`	Mem[Reg[`Ri`]`*S`]	`(Rb=0,D=0)`

Full list of addressing mode forms



Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$R[\mathtt{r}_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$M[R[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$M[Imm + R[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$M[R[\mathtt{r}_b] + R[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$M[Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$M[R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$M[Imm + R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$M[R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$M[Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s]$	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(Rb,Ri,S)` → Mem[Reg[`Rb`]+Reg[`Ri`]`*S+D`]

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

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

`D(Rb,Ri,S)` → Mem[Reg[`Rb`]+Reg[`Ri`]`*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(Rb,Ri,S)` → Mem[Reg[`Rb`]+Reg[`Ri`]`*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(Rb,Ri,S)` →
Mem[Reg[`Rb`]+Reg[`Ri`]`*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(Rb,Ri,S)` →
Mem[Reg[`Rb`]+Reg[`Ri`]`*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`

Outline

- Assembly Languages
- Registers
- x86-64 Assembly
    - Introduction
    - Move Instruction
    - Memory Addressing Modes