Introduction to x86-64 Assembly

Branden Ghena

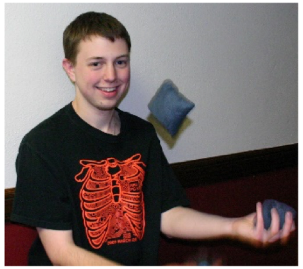
UC Berkeley

Carleton College – CS208
Northfield, Minnesota
January 22, 2020

Slides gratefully adapted from:
- University of Washington
- University of California-Berkeley
- Carnegie Mellon University

Branden Ghena (he/him)

- Education
  - Undergrad: Michigan Tech
  - Master's: University of Michigan
  - PhD: University of California, Berkeley
- Research
  - Low-power sensing systems
  - Embedded systems generalist
    - Platforms
    - Networks
- Teaching
  - Intro to computer systems
  - Embedded systems



GAMES DONE QUICK

Goals of this seminar:

1. What is an Instruction Set Architecture?
   - RISC versus CISC Ideas
2. Understand how CPUs hold data and interact with it.
3. Learn move and arithmetic x86-64 instructions.
4. Practice translating C statements to x86-64 assembly.

Please ask questions!

Introduction

Levels of Representation

Higher-Level Language Program (e.g. C)

Compiler

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Assembly Language Program (e.g. x86-64)

Assembler

```
pushq %rbx
movq %rdx, %rbx
movq %rax, (%rbx)
popq %rbx
```

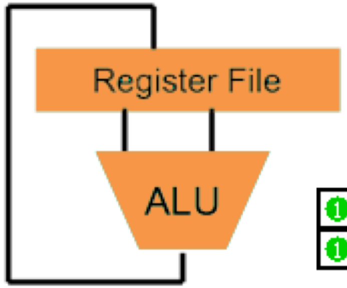
We are here

Machine Language Program (x86-64)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

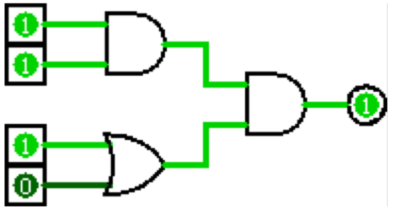
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



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

```
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
```

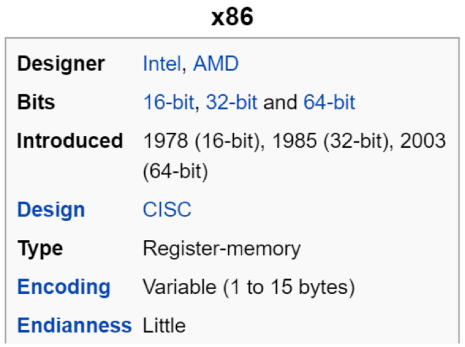
There are many assembly languages

- 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

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

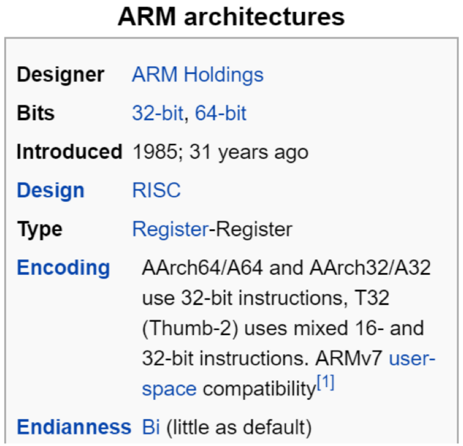
Mainstream Instruction Set Architectures



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

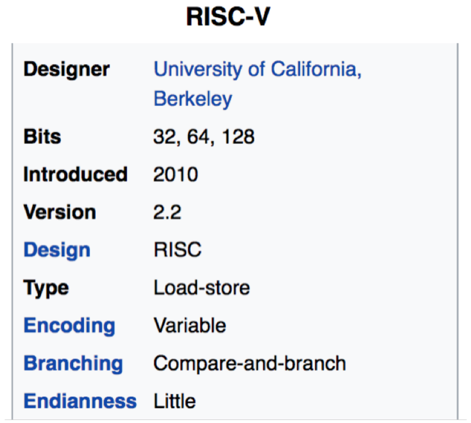
Macbooks & PCs
(Core i3, i5, i7, M)
x86 Instruction Set



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)

Smartphone-like devices (iPhone, Android), Raspberry Pi, Embedded systems
ARM Instruction Set

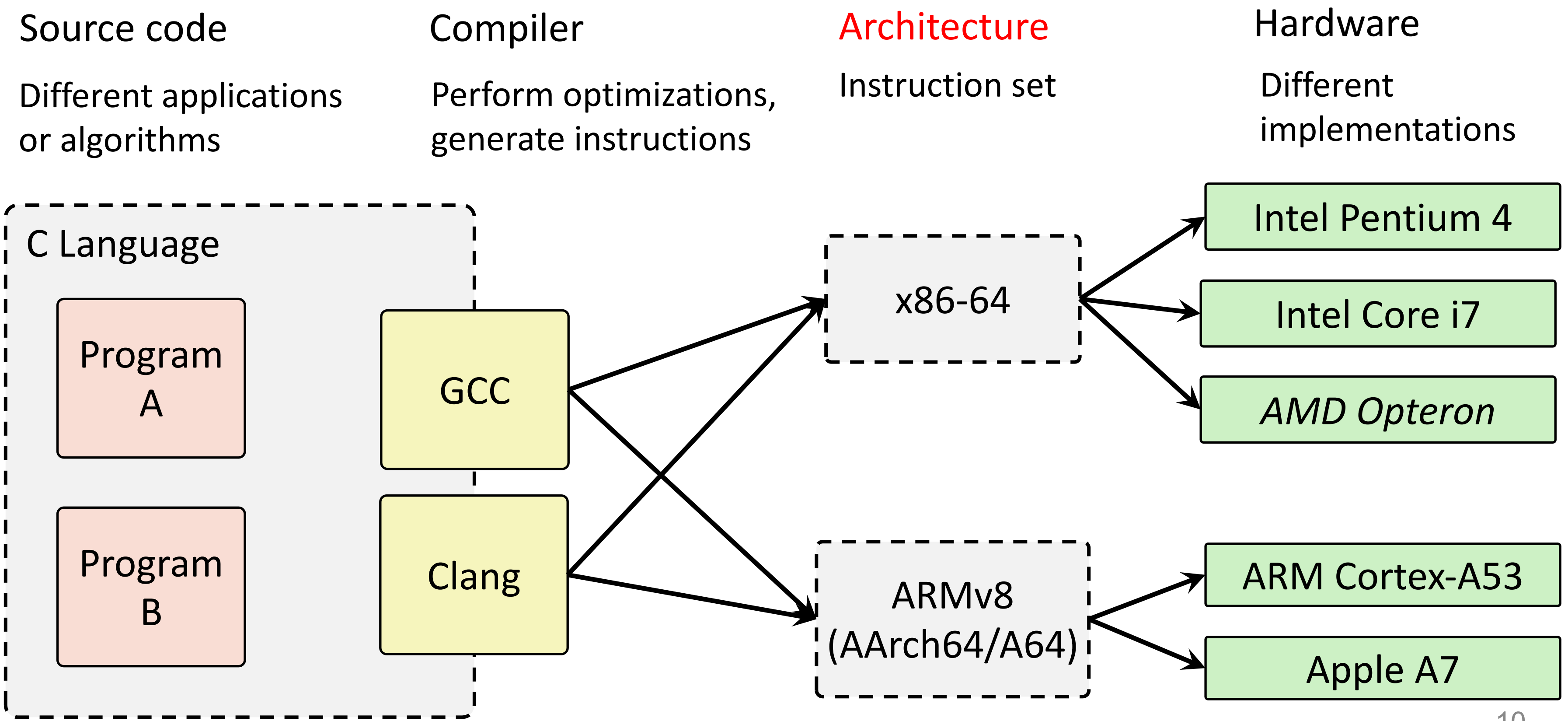


RISC-V

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



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 and 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

Intel x86 Processors

- Dominate laptop/desktop/server market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
  - Historical legacy has large impact on architecture

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

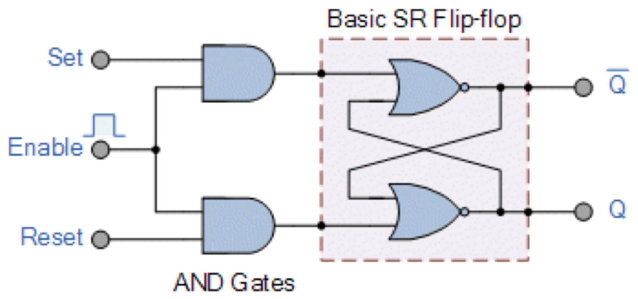
Intel x86 Evolution

Name	*Date*	*Transistors*	*MHz*
• 8086	1978	29K	5-10
– First 16-bit Intel processor. Basis for IBM PC & DOS. 1MB address space			
• 386	1985	275K	16-33
– First 32-bit Intel processor, referred to as IA32			
• Pentium 4E	2004	125M	2800-3800
– First 64-bit Intel x86 processor, referred to as x86-64			
• Core 2	2006	291M	1060-3500
– First multi-core Intel processor			
• Core i7	2008	731M	1700-3900

Registers

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 memories of a fixed size
  - Can be read or written
  - Limited in number
  - Very fast and low power to access
  - not typed like C
    - 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

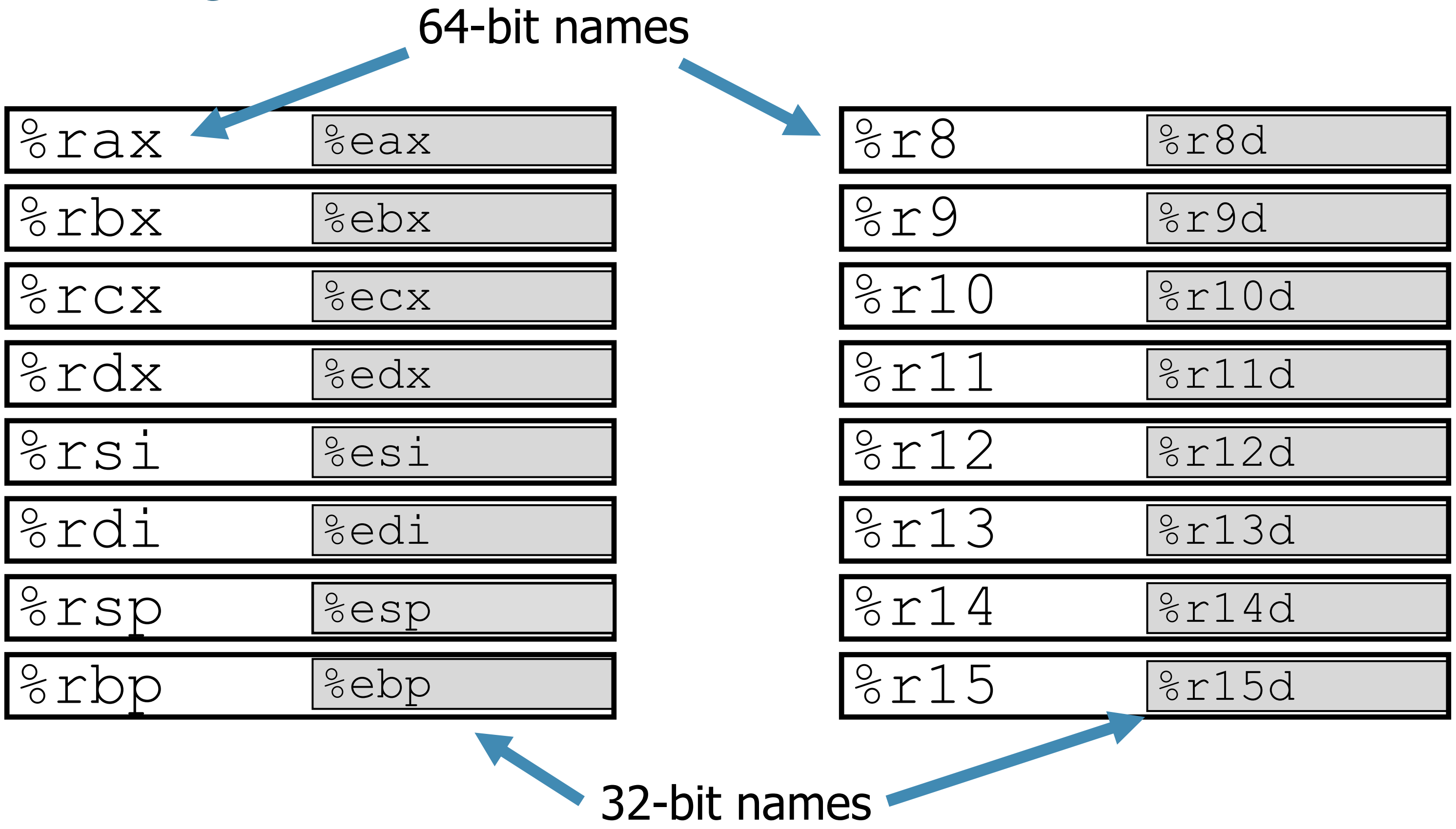
- x86-64 has 16 registers
  - Historically only 8 registers
  - Added 8 more with 64-bit extensions

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
  - Imprecise term that will inevitably slip in to 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

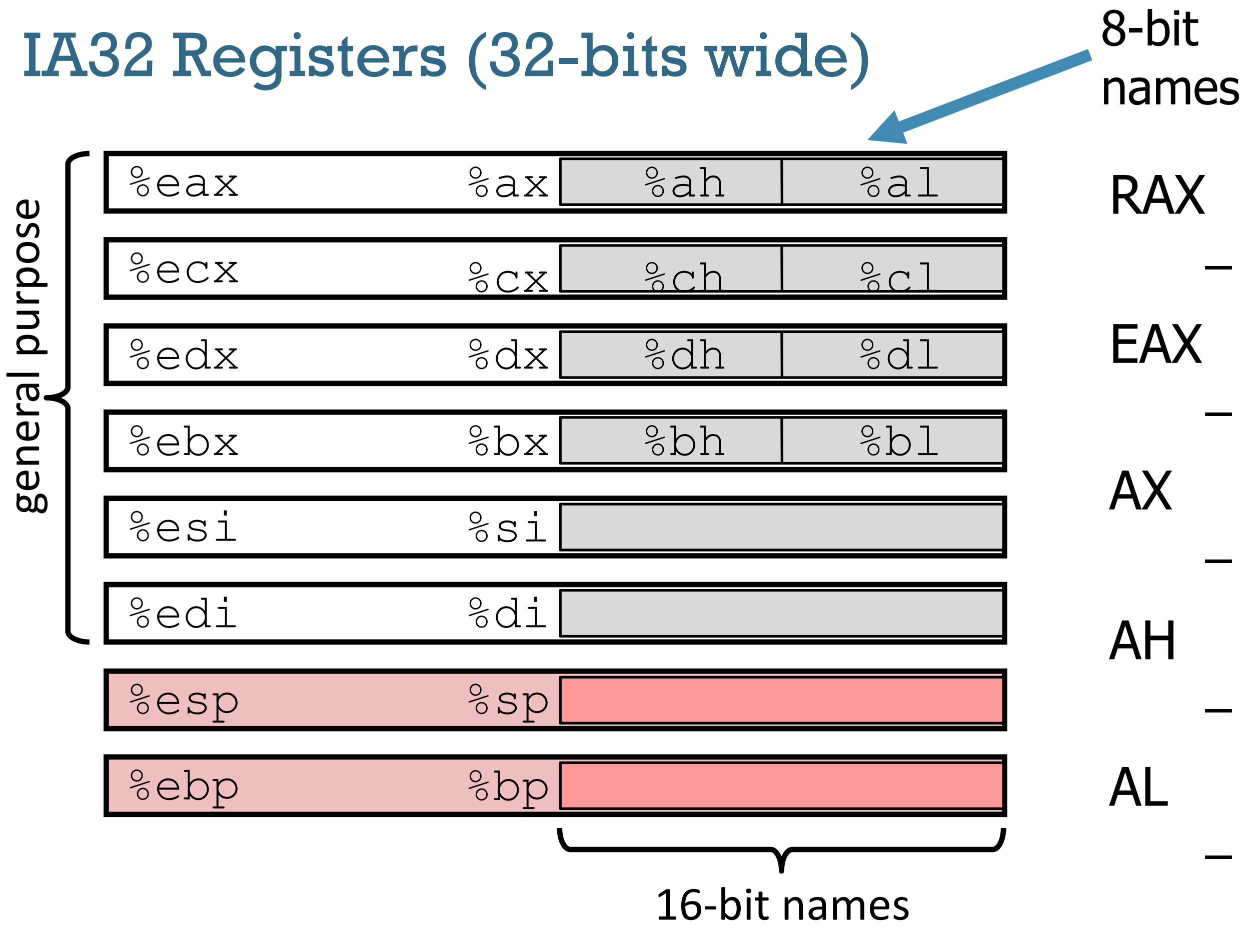
x86-64 Registers



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

IA32 Registers (32-bits wide)



RAX
- Register Accumulator

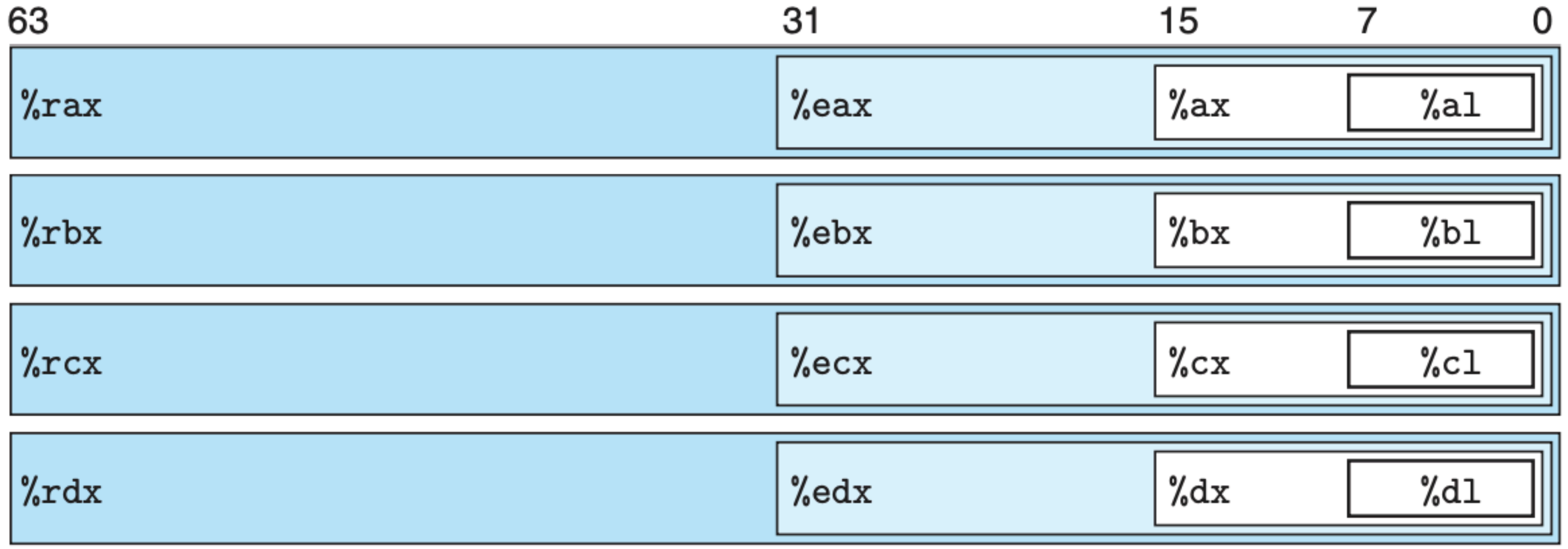
EAX
- Extended Accumulator

AX
- Accumulator

AH
- Accumulator Higher

AL
- Accumulator Lower

x86-64 Register Access Options



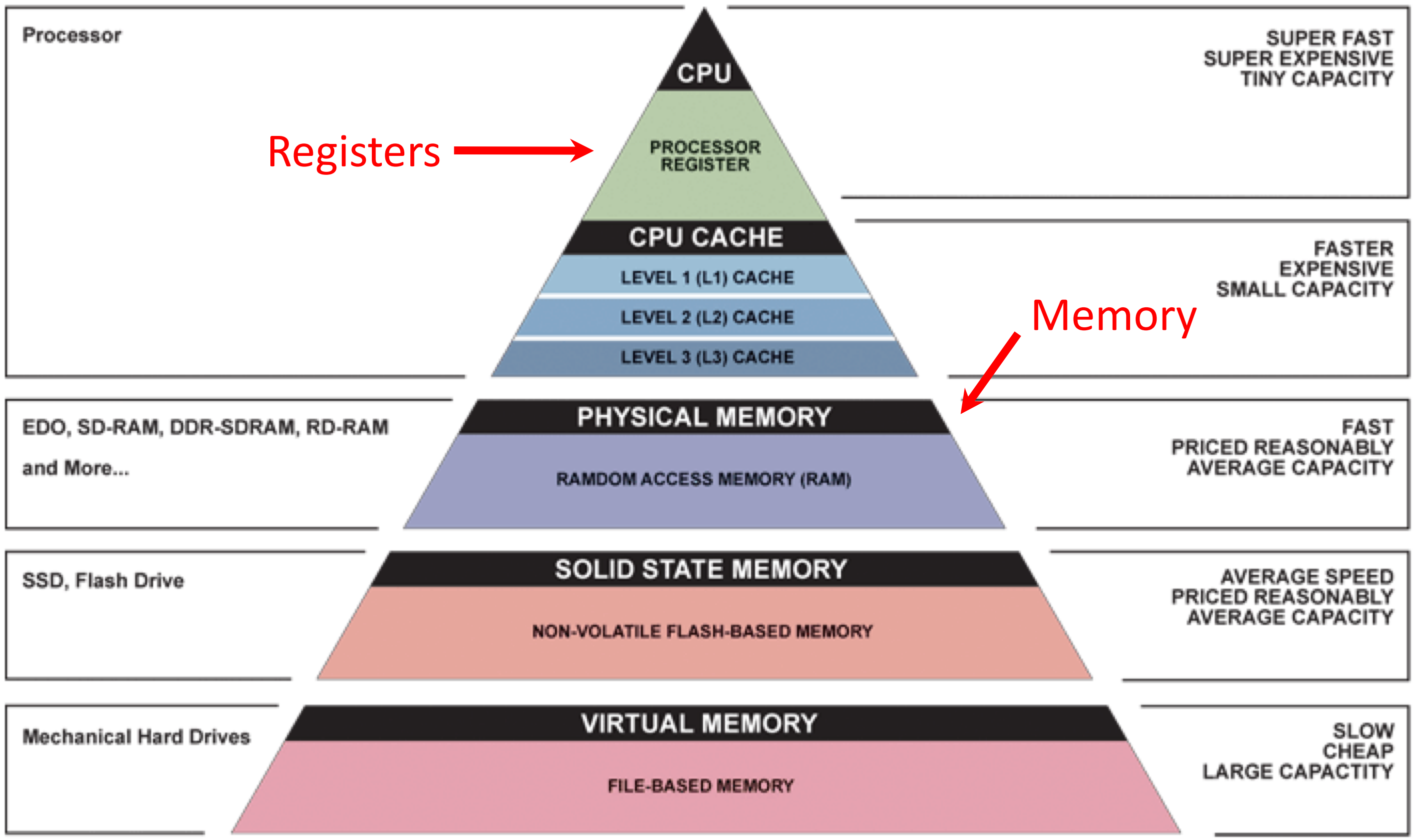
Registers can be accessed by any of these names to work with
8-byte, 4-byte, 2-byte, or 1-byte data

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



Review Question

Which of these is FALSE?

Registers:

[A] Are faster to access than memory

[B] Do not have addresses

[C] Can have special purposes

[D] Are dynamically created as needed

Review Question

Which of these is FALSE?

Registers:

[A] Are faster to access than memory

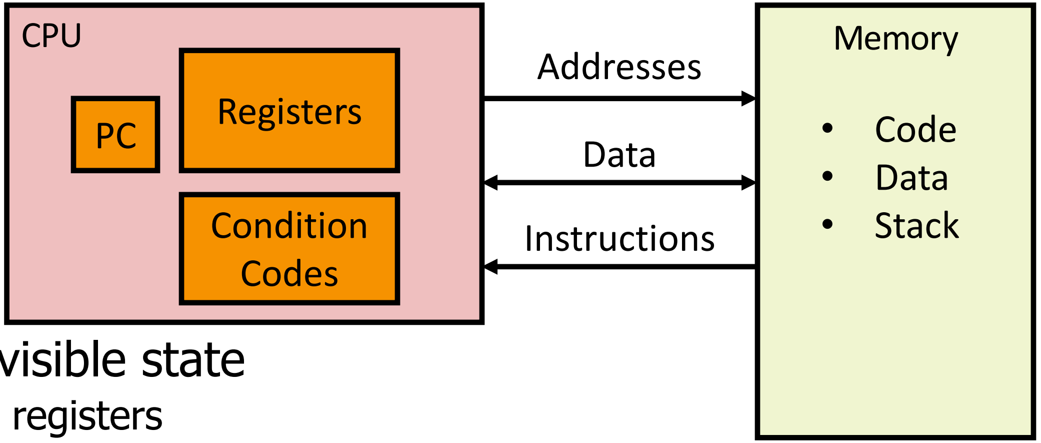
[B] Do not have addresses

[C] Can have special purposes

~~[D] Are dynamically created as needed~~

There are a fixed number of registers in an architecture

Assembly Programmer's View of System State



Programmer-visible state

- Named registers
  - Together in "register file"
  - Heavily used program data
- PC: the Program Counter (`%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)

x86-64 Assembly Overview

Writing Assembly Code? In 2019???

- 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

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

x86-64 Instructions

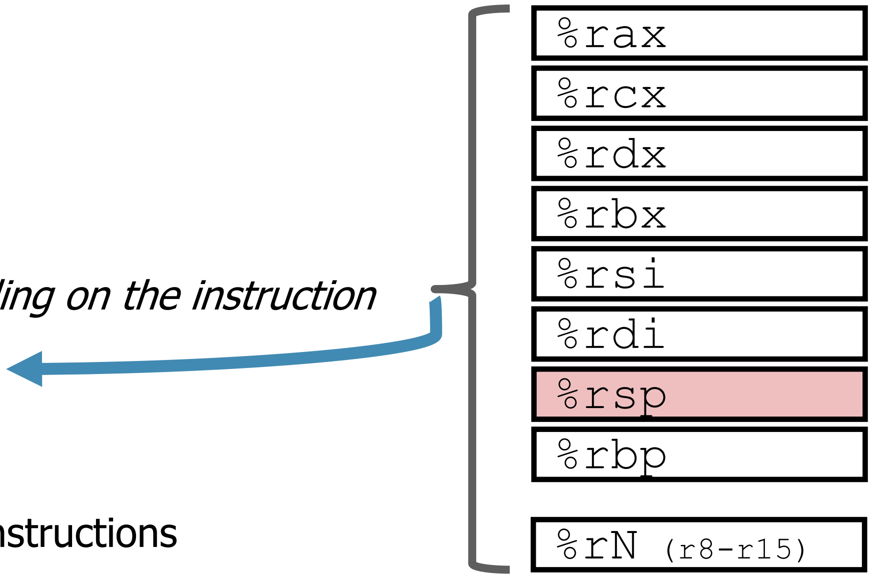
- General Instruction Syntax:

```
op src, dst
```

  - 1 operator, 2 operands
    - `op` = operation name (“operator”)
    - `src1` = source location (“source”)
    - `dst` = destination location (“destination”)
- Keep hardware simple via regularity

Operand Types

- ***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 ⟶ access memory
  - Various other “address modes” we’ll talk about 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
```

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

# 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.

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`)

x86-64 Assembly Move Instructions

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"

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

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

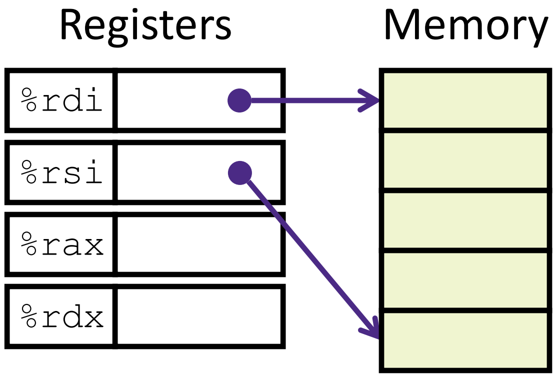
- How would you do it?

Example of Move Instructions: swap()

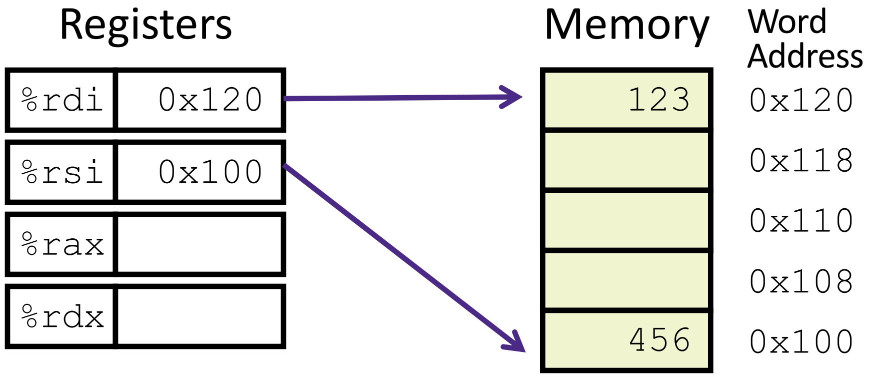
```
void swap(long* xp, long* yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

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

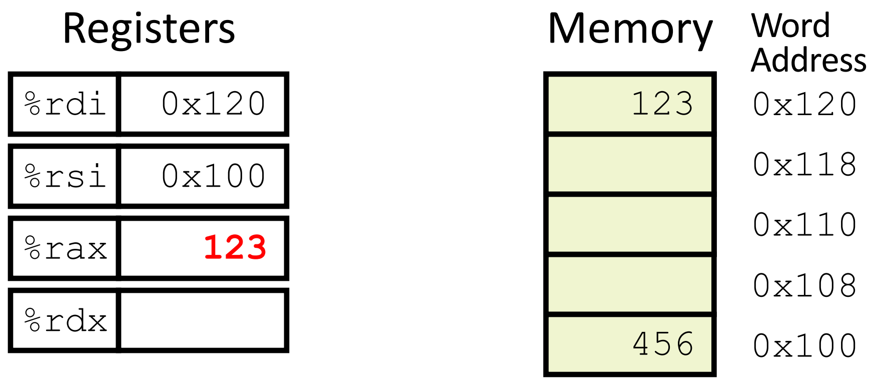


Example of Move Instructions: swap()



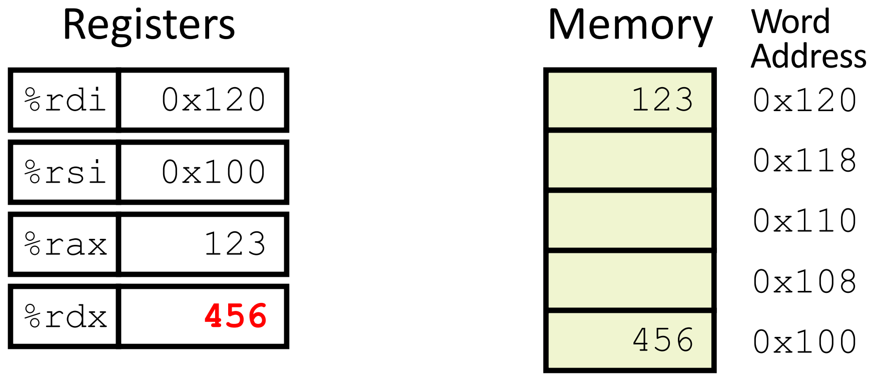
```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Example of Move Instructions: swap()



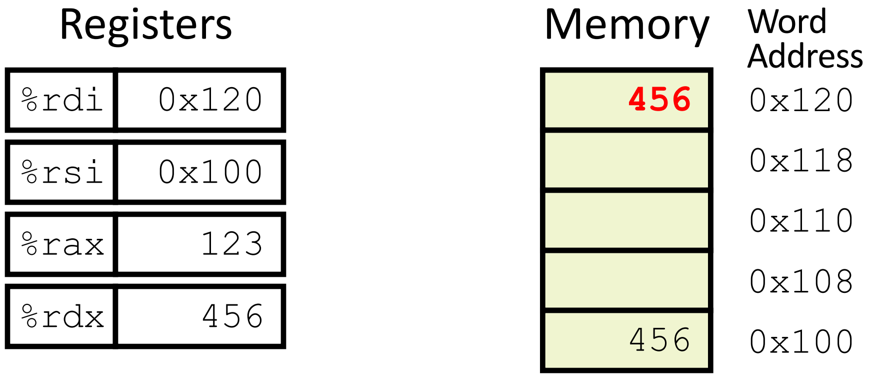
```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Example of Move Instructions: swap()



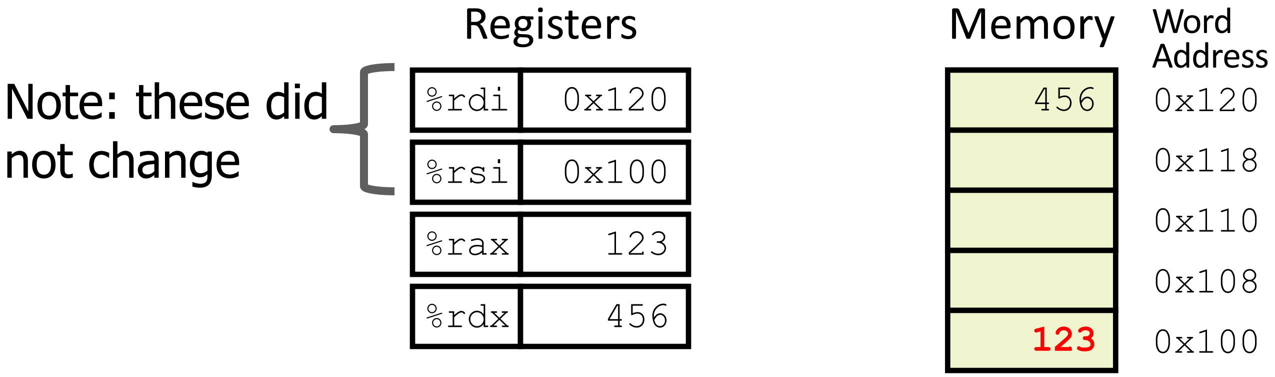
```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

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()



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

x86-64 Assembly Arithmetic Instructions

Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - Maximum of one memory operand!
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts

Format	Computation	
addq *src*, *dst*	*dst = dst + src*	(*dst += src*)
subq *src*, *dst*	*dst = dst – src*	
imulq *src*, *dst*	*dst = dst * src*	signed mult
sarq *src*, *dst*	*dst = dst >> src*	Shift right **A**rightmetic
shrq *src*, *dst*	*dst = dst >> src*	Shift right logical
shlq *src*, *dst*	*dst = dst << src*	(same as `salq`)
xorq *src*, *dst*	*dst = dst ^ src*	
andq *src*, *dst*	*dst = dst & src*	
orq *src*, *dst*	*dst = dst | src*	

↑ operand size specifier

Some Arithmetic Operations

- Unary (one-operand) Instructions:

Format	Computation	
`incq` *`dst`*	*dst = dst + 1*	increment
`decq` *`dst`*	*dst = dst – 1*	decrement
`negq` *`dst`*	*dst = –dst*	negate
`notq` *`dst`*	*dst = ~dst*	bitwise complement

- See Section 3.5.5 for more instructions:
`mulq, cqto, idivq, divq`

Converting C to Assembly

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

Converting C to Assembly

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
movq %rbx, %rax
addq %rcx, %rax
```

Converting C to Assembly

- Suppose `a → %rax, b → %rbx, c → %rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
movq    $0, %rax
addq %rbx, %rax
addq %rcx, %rax
```

Is this okay?

Converting C to Assembly

- Suppose `a → %rax, b → %rbx, c → %rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
movq    $0, %rax
addq %rbx, %rax
addq %rcx, %rax
```

Is this okay?

Yes: just a little slower

Converting C to Assembly

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
addq %rbx, %rcx
movq %rcx, %rax
```

Is this okay?

Converting C to Assembly

- Suppose `a` → `%rax`, `b` → `%rbx`, `c` → `%rcx`
  Convert the following C statement to x86-64:

```
a = b + c;
```

```
addq %rbx, %rcx
movq %rcx, %rax
```

Is this okay?

No: overwrites C

Review Question

Reminder
`addq, src, dst → dst = dst + src`

- Suppose `a → %rax, b → %rbx, c→ %rcx`
  Convert the following C statement to x86-64:

```
c = (a-b)+5;
```

```
[A]
movq %rax, %rcx
subq %rbx, %rcx
addq   $5, %rcx
```

```
[B]
movq %rax, %rcx
subq %rbx, %rcx
movq   $5, %rcx
```

```
[C]
subq %rcx, %rax, %rbc
addq %rcx, %rcx, $5
```

```
[D]
subq %rbx, %rax
addq $5, %rax
movq %rax, %rcx
```

Review Question

Reminder

`addq, src, dst → dst = dst + src`

- Suppose `a → %rax, b → %rbx, c → %rcx`
  Convert the following C statement to x86-64:

```
c = (a-b)+5;
```

```
[A]
movq %rax, %rcx
subq %rbx, %rcx
addq   $5, %rcx
```

```
[B]
movq %rax, %rcx
subq %rbx, %rcx
movq   $5, %rcx
```

c = 5

```
[C]
subq %rcx, %rax, %rbc
addq %rcx, %rcx, $5
```

Not x86

```
[D]
subq %rbx, %rax
addq $5, %rax
movq %rax, %rcx
```

Overwrites a

x86-64 Assembly
Memory Addressing Modes

Memory Addressing Modes: Basic

- **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)

- **Special cases** (see CSPP Figure 3.3 on p.181)
  - `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)`

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) $\rightarrow$
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`

Wrap Up

Levels of Representation

Higher-Level Language Program (e.g. C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g. x86-64)

```
pushq %rbx
movq %rdx, %rbx
movq %rax, (%rbx)
popq %rbx
```

We are here

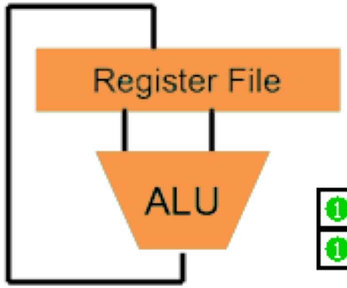
Assembler

Machine Language Program (x86-64)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

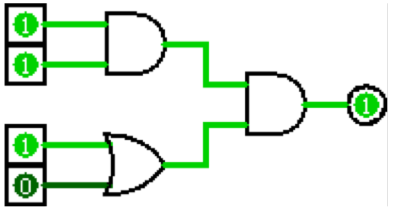
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to x86-64 Assembly

- An Instruction Set Architecture is the software model of a processor
  - Operations, Registers, and Memory interactions
- Introduction to x86-64 ISA
  - 16 registers each 64-bits in size
  - Operations with immediates, registers, or memory
- Remaining details of x86-64 assembly
  - Condition codes and control flow (if, while, for)
  - Function calls and calling conventions

Backup Slides

Address Computation Instruction

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
  - `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

%rax	
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

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

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

Example: `lea` vs. `mov`

Registers

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

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

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