

x86-64 Programming I

CSE 351 Summer 2024

Instructor:

Ellis Haker

Teaching Assistants:

Naama Amiel

Micah Chang

Shananda Dokka

Nikolas McNamee

Jiawei Huang

HTML/CSS



JavaScript



Java



C++



x86
Assembly



Binary



Tapping a
Charged Wire on
the Motherboard



Administrivia

- Due today
 - HW5 (11:59pm)
 - **Lab 1a (11:59pm)**
- Due Friday, 7/5
 - RD8 (1pm)
 - HW6 (11:59pm)
- Quiz 1 released Friday at 11:59pm
- Optional reading posted on Ed
 - Recent article about designing assembly language design
 - Beyond the scope of this course, but very cool!

Review Questions

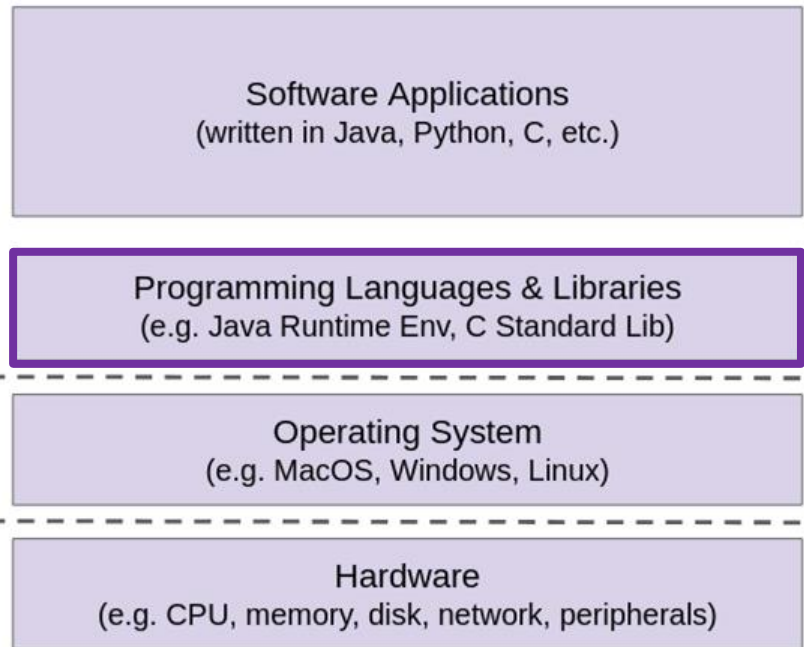
Assume that the register `%rdx` holds the value `0x 01 02 03 04 05 06 07 08`

Answer the following questions about the instruction **`subq $1, %rdx`**

1. Operation type: *arithmetic because subtraction*
2. Operand types: *immediate (\$), register (%rdx)*
3. Operating width: *8B because of "q"*
4. (extra) Result stored in `%rdx`:
 $\%rdx = \%rdx - 1 = 0x\ 00\ 02\ 03\ 04\ 05\ 06\ 07\ 07$

Layers of Computing Revisited

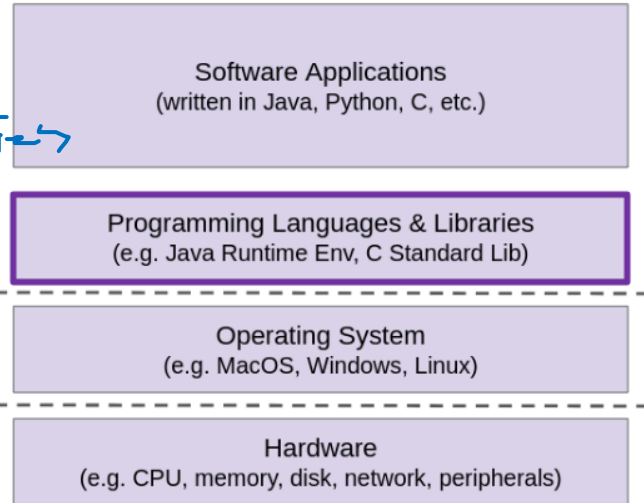
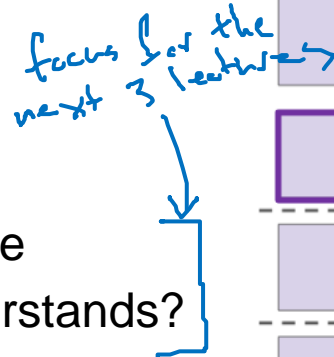
- So far, we've focused on **hardware**
 - How does the CPU store and read data from memory?
- Shifting focus to **languages & libraries**
 - How are programs created and executed on the CPU?
 - Take CSE 401 to learn more
- Still needs hardware support!
 - Take CSE 469 to learn more



Programming Languages & Libraries: 351 View

- Topics:
 - **x86-64 assembly**
 - Procedures
 - Stacks
 - Executables
- How does your source code become something that your computer understands?
- How does the CPU organize and manipulate local data?

focus for the next 3 lectures



Lecture Topics

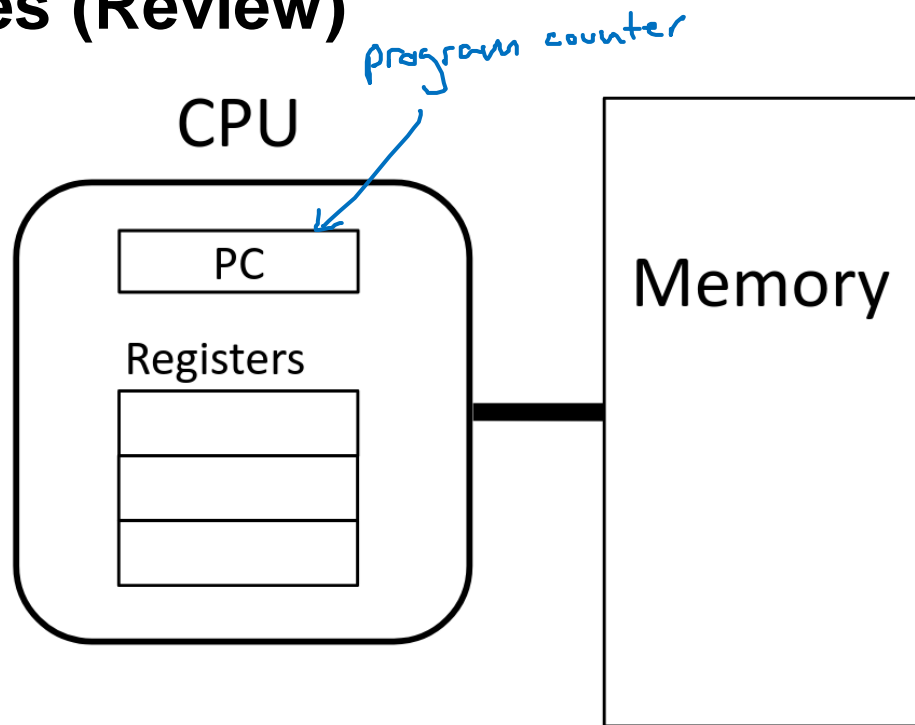
- **Assembly intro**
 - **Instruction set philosophies**
- X86-64 programming
 - Data types
 - Instructions
 - Registers
 - Memory addressing

Definitions

- **Instruction Set Architecture (ISA)**: the parts of a processor design that one needs to understand to write assembly code
 - What is directly visible to software
 - The “contract” between hardware and software
 - 351 focus
- **Microarchitecture**: hardware implementation of the ISA
 - CSE/EE 469

Instruction Set Architectures (Review)

- 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



What is a Register? (Review)

- Special locations on the CPU that store a small amount of data
 - Accessed very quickly (once per clock cycle)
- Have *names*, not addresses
 - In x86, start with % (e.g., %rsi)
- Registers are at the heart of assembly programming
 - Very useful, but scarce, *especially* in x86

Memory vs. Registers (Review)

Memory



- Addresses
 - Ex: 0x7FFFD024C3DC
- Big
 - ~16GB
- Slow
 - ~50-100ns
- Dynamic
 - Can expand as needed

Registers



- Names
 - Ex: %rdi
- Small
 - 16 8-byte registers = 128B
- Fast
 - <1ns
- Static
 - Fixed number in hardware

General ISA Design Decisions

- Instructions
 - What instructions are available? What do they do?
 - How are they encoded?
- Registers
 - How many are there?
 - How wide are they?
- Memory
 - How do you specify a memory location?

Instruction Set Philosophies (Review)

- **Complex Instruction Set Computing (CISC):** lots of elaborate instructions
 - 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, less power-hungry hardware
 - Let software do the complicated operations by composing simpler ones
 - ARM, RISC-V

Instruction Set Philosophies (Review) (pt 2)

- **Complex Instruction Set Computing (CISC):** lots of elaborate instructions
 - 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

Example: ADDSUBPS

- “Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.”

Mainstream ISAs



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)
Branching	Condition code
Endianness	Little

PCs, older Macs
x86-64 instruction set



ARM

Designer	Arm Holdings
Bits	32-bit, 64-bit
Introduced	1985
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]
Branching	Condition code, compare and branch
Endianness	Bi (little as default)

Mobile devices, M1/M2
Macs
ARM instruction set

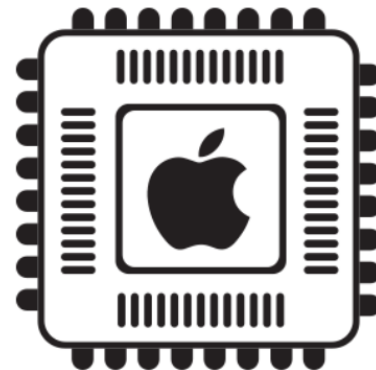


RISC-V

Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Design	RISC
Type	Load-store
Encoding	Variable
Endianness	Little ^{[1][3]}

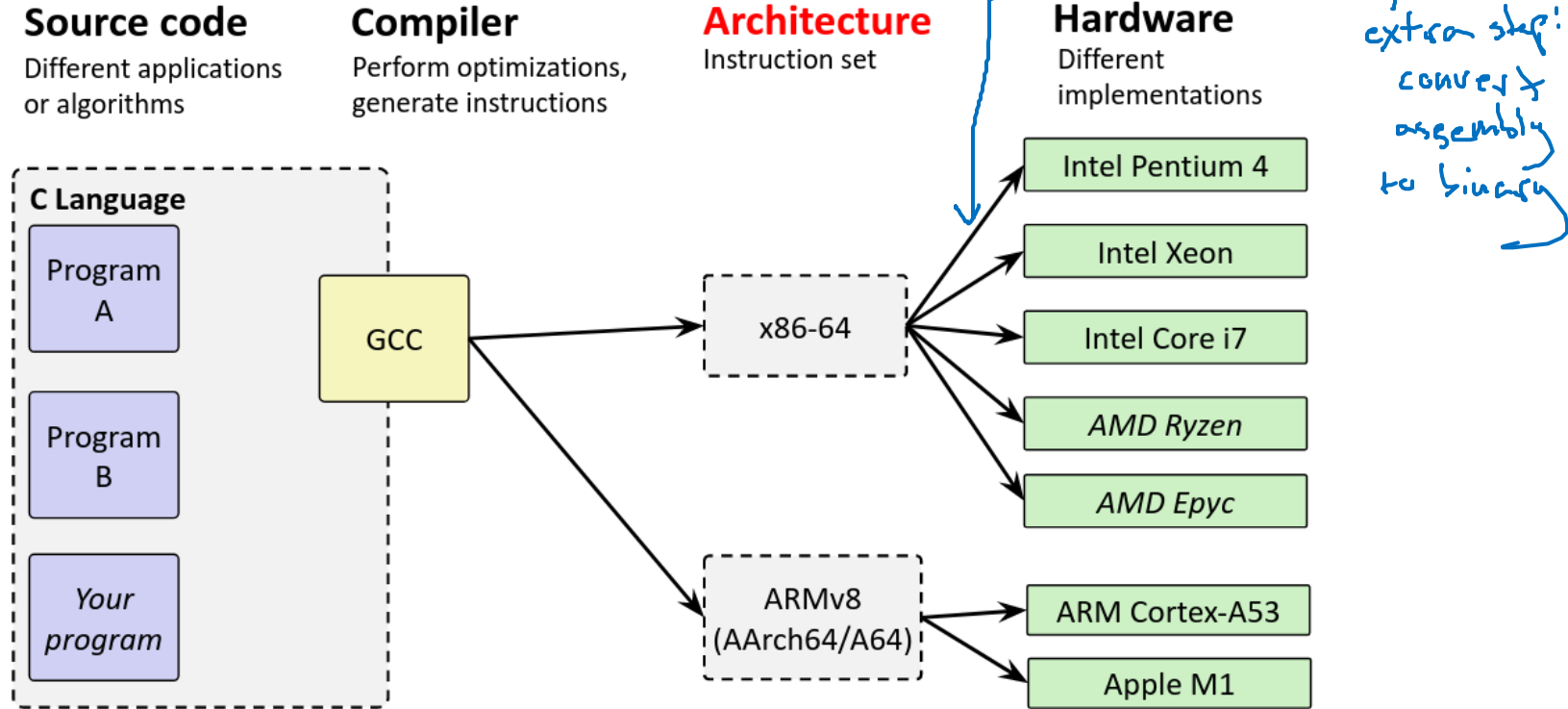
Mostly research
RISC-V instruction set

Current Industry Trends - A RISC-y Shift



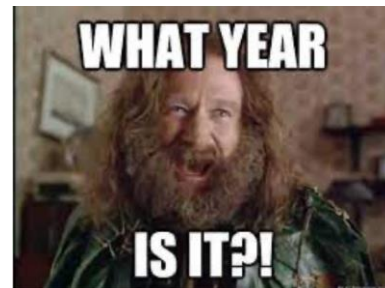
- Historically, there was a lot of debate about RISC vs CISC
 - Intel went the CISC route in the 1980s
 - Would make programming in assembly easier
 - Implementing more things in hardware
- Traditional wisdom says the RISC is better for simple systems, not PCs
- *But* things are shifting!
 - Apple switched to ARM in 2020
- Why?
 - **Efficiency:** RISC uses less power
 - **Performance:** each instruction is faster, easier to parallelize
 - **Scalability:** suitable for devices of all sizes (desktops, laptops, and phones)

Architecture Sits at the Hardware Interface



Writing Assembly Code? In \$CURRENT_YEAR???

- Chances are, you'll never write a program in assembly, but understanding it 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
 - Understand optimizations done/not done by the compiler
 - 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



Lecture Topics

- Assembly intro
 - Instruction set philosophies
- **X86-64 programming**
 - **Data types**
 - **Instructions**
 - **Registers**
 - **Memory addressing**

x86-64 Integer Registers – 64 bits wide

= 8 bytes

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

special
register,
we'll talk
about it
later!

8B
4B

Some History: IA32 Registers – 32 bits wide

we still have access to these old register names!
now refer to the lower bytes of the 8B register

general purpose

%eax	%ax	%ah	%al	accumulate
%ecx	%cx	%ch	%cl	counter
%edx	%dx	%dh	%dl	data
%ebx	%bx	%bh	%bl	base
%esi	%si			source index
%edi	%di			destination index
%esp	%sp			stack pointer
%ebp	%bp			base pointer

16-bit virtual registers
(backwards compatibility)

Name Origin
(mostly obsolete)

All of these are on the Mithras Reference Sheet

x86-64 Assembly “Data Types”

- Integral data of 1, 2, 4, or 8 bytes (b, w, l, q) — signed and unsigned. HW doesn't distinguish!
- Floating point data, not covered in 351
 - Different registers for those (e.g., %xmm1, %ymm2)
 - Come from extensions to x86 (SSE, AVX, ...)
- No aggregate types such as arrays or structs
 - Just contiguously allocate bytes in memory — only get single elements at a time
- Two common syntaxes—Must know which you're reading!
 - **AT&T**: used in our course, gnu tools (including gcc), ...
 - **Intel**: used in Intel documentation, Intel tools, ...

Why AT&T? It used to be Bell Labs!

Instruction Sizes and Operands (Review)

- Size specifiers

- **b** = 1-byte (“byte”)
- **w** = 2-byte (“word”)
- **l** = 4-byte (“long word”)
- **q** = 8-byte (“quad word”)
- If using registers, must match width

Why is “word” 2 bytes? Because that was the word size when x86 was new, and it has to be maintained for backwards compatibility.

register size must match instruction
ex: if using q, use 8B register
if using l, use 4B
etc...

- Operand types

- **Immediate**: constant value (\$)
- **Register**: 1 of 16 general-purpose registers (%)
- **Memory**: consecutive bytes of memory at a computed address (())

↑
see memory addressing slides

Instruction Types (Review)

1. Transfer data between memory and a register

- Load from memory -> register
 - `%reg = Memory[address]`
- Store from register -> memory
 - `Memory[address] = %reg`

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

- Note: cannot transfer between two memory locations in one instruction!

2. Perform arithmetic operation on register or memory data

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

3. Control flow: what instruction to execute next

- Unconditional jumps to/from procedures
- Conditional branches

future lecture

Moving Data

b
w
l
q

imm
reg
mem

reg
mem

can't have an immediate
as the destination

- General form: `movq <source>, <destination>`
 - More of a “copy” than a “move”
 - Missing letter (`_`) is for the width specifier

Ex: `movq %rax, %rbx`

- Copies the 8-byte value from register `%rax` into register `%rbx`
- Operand Combinations:
 - **Immediate** -> **Register** or **Memory** (copies Immediate value to location)
 - **Register** -> **Register** or **Memory** (copies data in register to location)
 - **Memory** -> **Register** (copies data in memory to register)

- Can't go from memory -> memory in a single instruction!

requires 2 instructions
mem → reg
+ reg → mem

Some Arithmetic Operations

- Binary (two-argument) operations
 - Beware argument order!
 - src can be immediate, register, or memory
 - dst only register or memory
 - Results always stored in dst
 - Maximum of one memory operand! *same as mov*
 - No distinction between signed and unsigned
 - Only arithmetic vs logical shifts

Format	Computation	Notes
addq src, dst	$\text{dst} = \text{dst} + \text{src}$	
subq src, dst	$\text{dst} = \text{dst} - \text{src}$	
imulq src, dst	$\text{dst} = \text{dst} * \text{src}$	
sarq src, dst	$\text{dst} = \text{dst} \gg \text{src}$	Arithmetic
shrq src, dst	$\text{dst} = \text{dst} \gg \text{src}$	Logical
shlq src, dst	$\text{dst} = \text{dst} \ll \text{src}$	Same as shlq
xorq src, dst	$\text{dst} = \text{dst} \wedge \text{src}$	
andq src, dst	$\text{dst} = \text{dst} \& \text{src}$	
orq src, dst	$\text{dst} = \text{dst} \text{src}$	

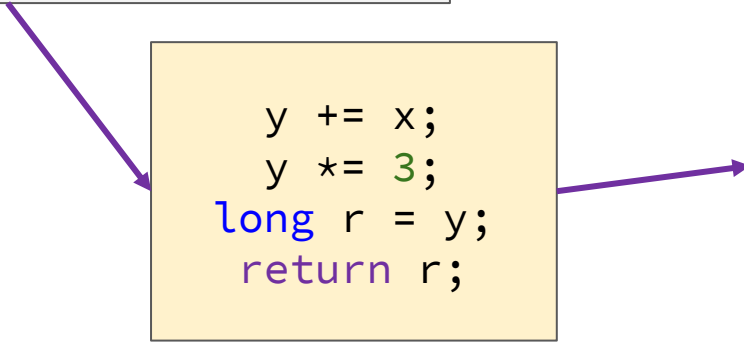
Practice Question

Which of the following are valid implementations of $rcx = rax + rbx$?

- `addq %rax, %rcx` $rcx = rcx + rax$
`addq %rbx, %rcx` $rcx = rcx + rax + rbx$
- `movq %rax, %rcx` $rcx = rax$
`addq %rbx, %rcx` $rcx = rax + rbx$
anything XOR itself = 0!
- `movq $0, %rcx` $rcx = 0$
`addq %rbx, %rcx` $rcx = 0 + rbx = rbx$
`addq %rax, %rcx` $rcx = rax + rbx$
- `xorq %rax, %rax` $rax = 0$
`addq %rax, %rcx` $rcx = rcx + 0 = rcx$
`addq %rbx, %rcx` $rcx = rbx + rcx$

Arithmetic Example

```
long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```



```
y += x;
y *= 3;
long r = y;
return r;
```

```
simple_arith:
    addq    %rdi, %rsi
    imulq   $3, %rsi
    movq    %rsi, %rax
    ret
```

Register	Uses
%rdi	1st arg (x)
%rsi	2nd arg (y)
%rax	return value

Example of Basic Addressing Modes

```
long add_ptr(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    return t0 + t1;
}
```

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

- Parentheses = memory addressing
 - Treat the value in the register as an address

Compiler Explorer:

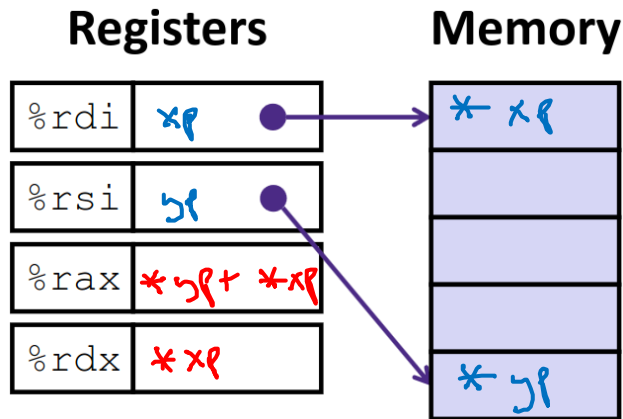
<https://godbolt.org/z/zc4Pcq>

Understanding add_ptr()

```
long add_ptr(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    return t0 + t1;
}
```

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

Register	Variable
%rdi	xp
%rsi	yp
%rdx	t0
%rax	return



Memory Addressing Modes

- General format: $D(Rb, Ri, S) = \text{Mem}[\text{Reg}(Rb) + \text{Reg}(Ri) * S + D]$
 - Rb = base register (any register)
 - Ri = index register (any register except `%rsp`)
 - S = scale factor (1, 2, 4, 8) - **Why these numbers?** *same as data widths used for pointer arithmetic / array indexing!*
 - D = displacement value (immediate)
- Can leave any of these out: *S defaults to 1, all others default to 0*
 - $D(Rb, Ri) = Rb + Ri + D$ - $S=1$
 - $(Rb, Ri, S) = Rb + Ri * S$ - $D=0$
 - $(Rb, Ri) = Rb + Ri$ - $D=0, S=1$
 - $(\textcircled{Ri}, S) = Ri * S$ - $D=0, Rb=0$
 - etc...

in the example `(./rdi)` on previous slide, `./rdi` was Rb

Address Computation Examples

*Stopped here, we'll continue
on Friday :)*

- 8-bit addresses
- Reminder: $D(Rb, Ri, S) \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$
 - (ignoring memory addressing portion for this exercise)

%rdx	0xF000
%rcx	0x0100

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

Summary

- **x86-64** is a complex (**CISC**) architecture
 - There are 3 types instructions
 - Data transfer
 - Arithmetic
 - Control flow
 - There are 3 types of operands
 - **Registers** (%)
 - **Immediates** (\$)
 - **Memory** (())
- **Registers** are small, fast places to store memory
 - Limited number, each with their own name