

Machine-Level Programming I: Basics

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Intel x86 Processors

→ AMD
→ first released in late
70s

- Dominate laptop/desktop/server market
- Evolutionary design
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
- Complex instruction set computer (CISC)
 - Many different instructions with many different formats
 - But, only small subset encountered with Linux programs
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!
 - In terms of speed. Less so for low power.

Intel x86 Evolution: Milestones

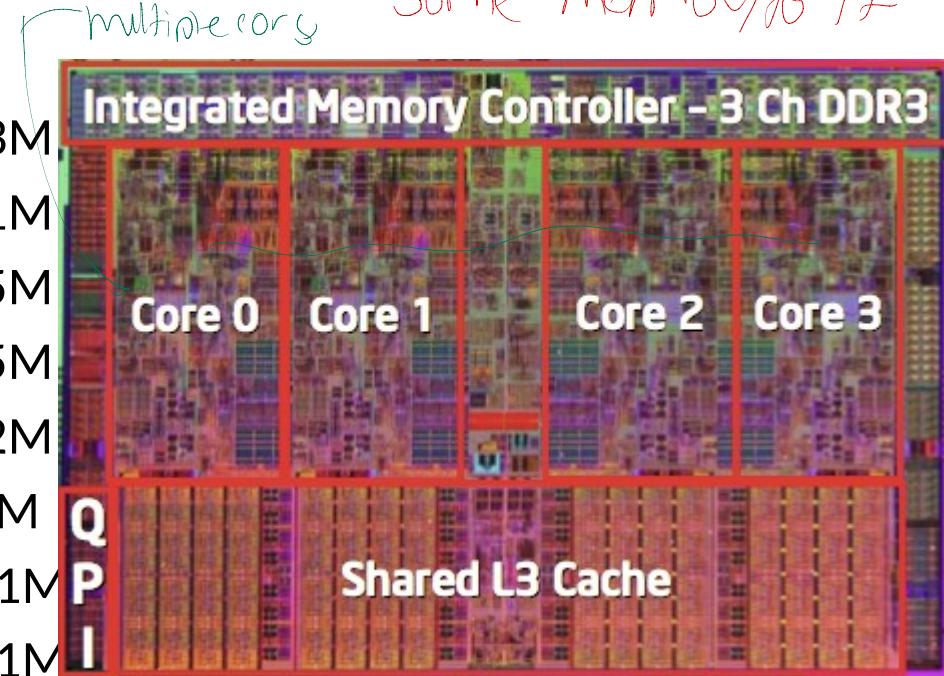
Name	Date	Transistors	MHz
■ 8086	1978	29K	5-10
			→ to 4000 MHz for every 3
■ 386	1985	275K	16-33
■ Pentium 4E	2004	125M	2800-3800
■ Core 2	2006	291M	1060-3500
■ Core i7	2008	731M	1700-3900
		Processor with 11 cores	

Intel x86 Processors, cont.

Cache dedicated
Some memory to it

■ Machine Evolution

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M



■ Added Features

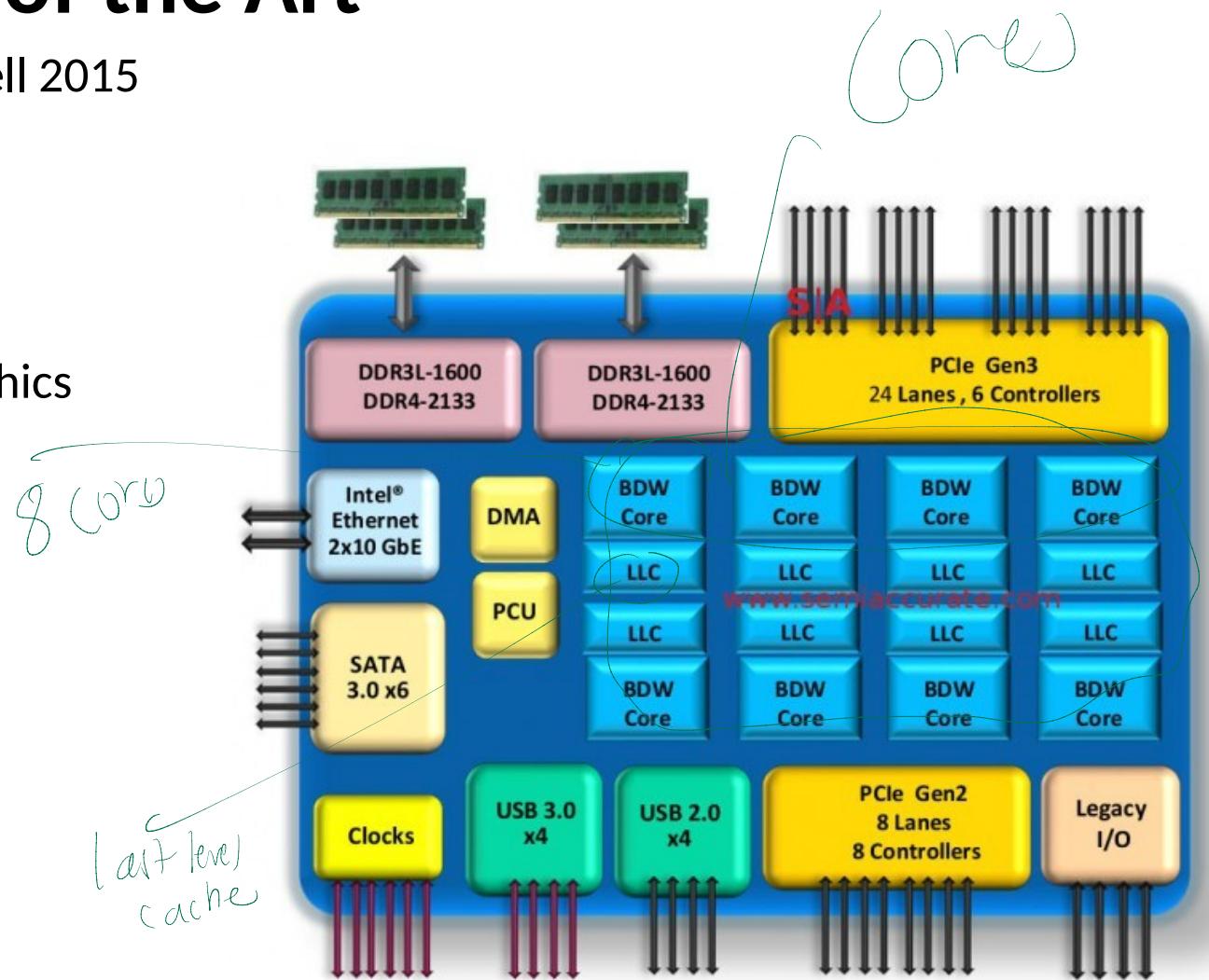
- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

2015 State of the Art

- Core i7 Broadwell 2015

■ Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz
- 65W



■ Server Model

- 8 cores
- Integrated I/O
- 2-2.6 GHz
- 45W

x86 Clones: Advanced Micro Devices (AMD)

■ Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

■ Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

■ Recent Years

- Intel got its act together
 - Leads the world in semiconductor technology
- AMD has fallen behind *Intel now more competitive with current processors*
 - Relies on external semiconductor manufacturer

Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
 - Totally different architecture (Itanium)
 - Executes IA32 code only as legacy
 - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
 - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
 - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
 - Extended Memory 64-bit Technology
 - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
 - But, lots of code still runs in 32-bit mode

Our Coverage

- IA32
 - The traditional x86
- x86-64
 - The standard
 - \$ gcc hello.c
 - \$ gcc -m64 hello.c
- Presentation
 - Book covers x86-64
 - Web aside on IA32
 - We will only cover x86-64

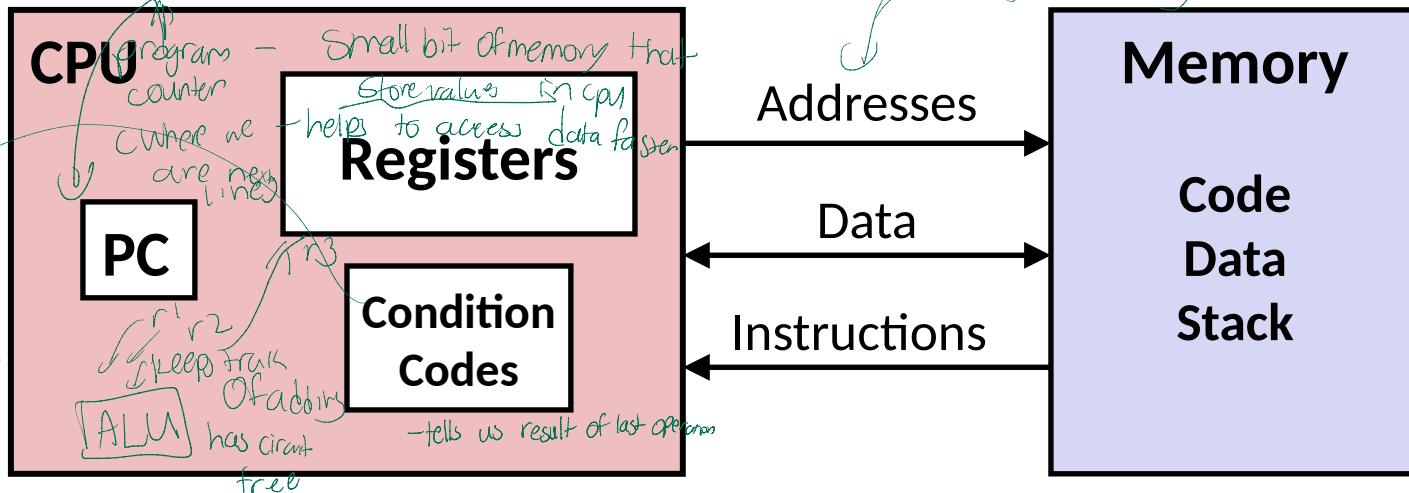
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
 - Examples: instruction set specification, registers.
 - **Microarchitecture:** Implementation of the architecture.
 - Examples: cache sizes and core frequency.
 - **Code Forms:**
 - **Machine Code:** The byte-level programs that a processor executes
 - **Assembly Code:** A text representation of machine code
 - **Example ISAs:**
 - Intel: x86, IA32, Itanium, x86-64
 - ARM: Used in almost all mobile phones
- What instructions are available
on a processor
What registers are there
Concern about
What actually gets executed
in assembly we can read
r1 → r2, move, make sure
we copy data from one register to another ISA*

Assembly/Machine Code View

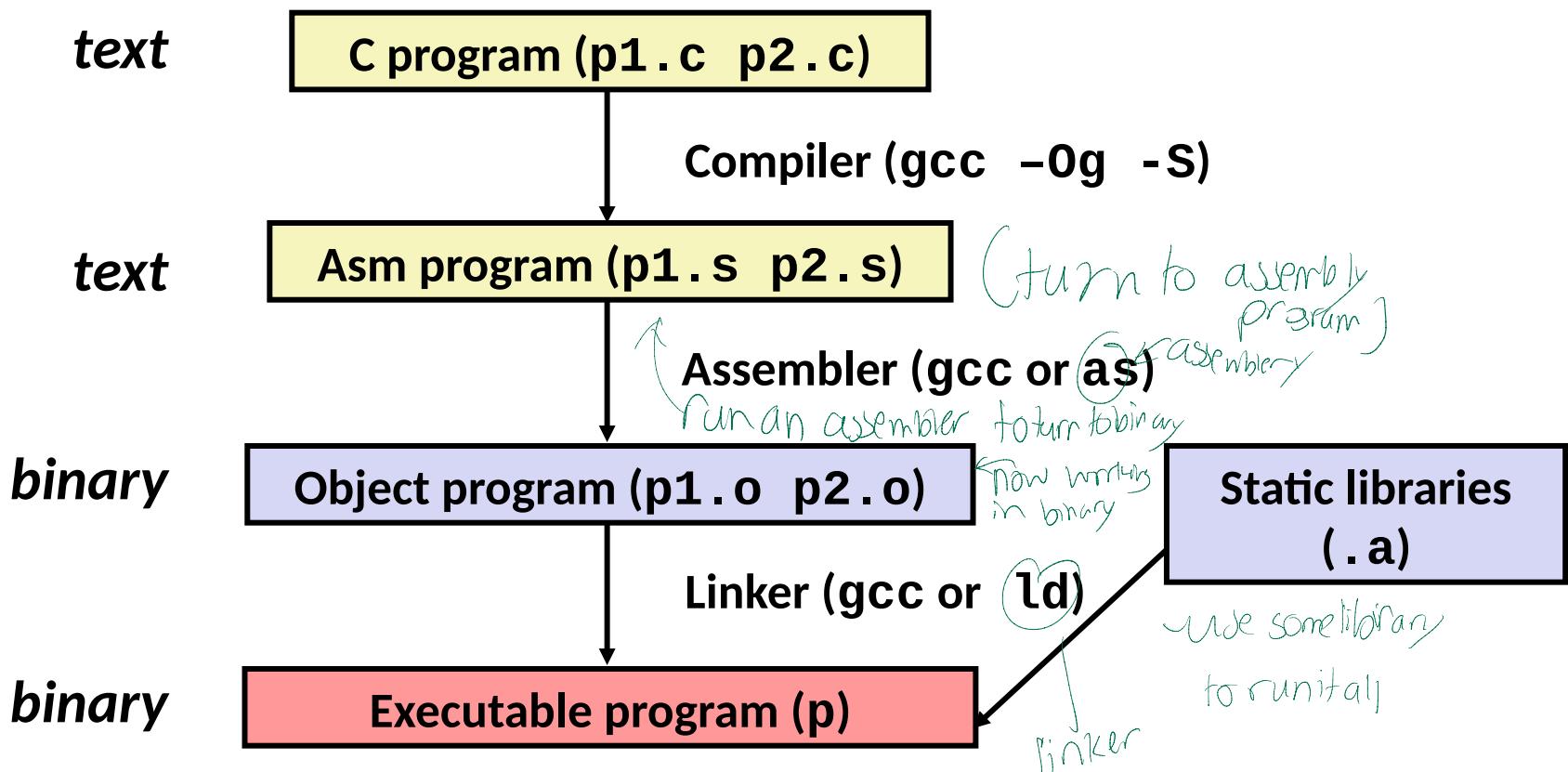


Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called "RIP" (x86-64)
- **Register file** (name of a register in 64bit)
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use basic optimizations (`-Og`) [New to recent versions of GCC]
 - Put resulting binary in file `p`



Compiling Into Assembly

go back
to thi)

C Code (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Generated x86-64 Assembly

```
sumstore:  
    pushq   %rbx  
    movq   %rdx, %rbx  
    call    plus  
    movq   %rax, (%rbx)  
    popq   %rbx  
    ret
```

Obtain with command

gcc -Og -S sum.c

Produces file sum.s

produce
of th's bdc

Different ways to represent
Assembly code
AT&T syntax
Intel
2nd min
way

Warning: Will get different results on different machines due to different versions of gcc and different compiler settings.

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
 - Data values
 - Addresses (untyped pointers)

width

NOT much

*can be
can have different
width*

- Floating point data of 4, 8, or 10 bytes

*Separate from integer
values*

- Code: Byte sequences encoding series of instructions



- No aggregate types such as arrays or structures

- Just contiguously allocated bytes in memory

no data structures in assembly, just data in memory

Assembly Characteristics: Operations

■ Perform arithmetic function on register or memory data

— add, multiply, subtract (arithmetic operations)

— Can do bitshifting and masking

■ Transfer data between memory and register

- Load data from memory into register
- Store register data into memory

— transfer between memory and register

■ Transfer control

- Unconditional jumps to/from procedures

— Conditional branches

Some are

unconditional/conditional

“goto” like goto

Not

if, while, for keyword

if ($x > y$)

else

conditional jump

Object Code

Code for sumstore

address bar
0x0400595:

0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

object
code

instruction
or data
we operate
on

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

watch
coordinates

Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

assembly file
object file
binary object file
-f (name)
.o .o look for lib m.a
linker error exception

Linker

- Resolves references between files
- Combines with static run-time libraries
 - E.g., code for **malloc**, **printf**
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution

not linked at compile time but at runtime
a.out lib m.a links your code with dynamic library

Machine Instruction Example

```
*dest = t;
```

mov instruction
src
name of 2 registers
in 64bit assembly
dest
put from register

```
movq %rax, (%rbx)
```

↑
specifies
register
it is

()
specifies
register
name
mean
diferencing a point

```
0x40059e: 48 89 03
```

bytes that get fed
to processor to decode
and execute

C Code

- Store value **t** where designated by **dest**

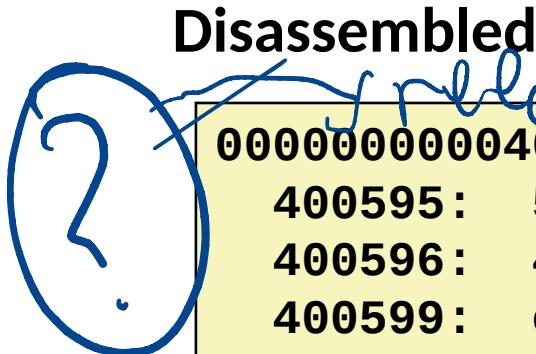
Assembly

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:
 - t:** Register **%rax**
 - dest:** Register **%rbx**
 - *dest:** Memory **M[%rbx]**

Object Code

- 3-byte instruction
- Stored at address **0x40059e**

Disassembling Object Code



Disassembled

0000000000400595 <sumstore>

400595:	53	push	%rbx
400596:	48 89 d3	mov	%rdx,%rbx
400599:	e8 f2 ff ff ff	callq	400590 <plus>
40059e:	48 89 03	mov	%rax,(%rbx)
4005a1:	5b	pop	%rbx
4005a2:	c3	retq	

Assembly
of
Object code

Assembly
of
Object code

■ Disassembler

objdump -d sum

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a .out (complete executable) or .o file

Alternate Disassembly

Object

0x0400595:

0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff

0xff
0xff
0x48
0x89
0x03
0x5b
0xc3

Disassembled

```
Dump of assembler code for function sumstore:  
0x0000000000400595 <+0>: push    %rbx  
0x0000000000400596 <+1>: mov     %rdx,%rbx  
0x0000000000400599 <+4>: callq   0x400590 <plus>  
0x000000000040059e <+9>: mov     %rax,(%rbx)  
0x00000000004005a1 <+12>:pop    %rbx  
0x00000000004005a2 <+13>:retq
```

■ Within gdb Debugger

gdb sum *Use disassemble to see what the executable code looks like*
disassemble sumstore

- Disassemble procedure

x/14xb *Look at your code in memory starting where this function memory* sumstore
(lower level detail)

- Examine the 14 bytes starting at sumstore

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations

4 bytes (want to look for overbytes)

32 bits

x86-64 Integer Registers

Number of registers to store data

%rax	%eax
	64 bit register

%r8	%r8d

%rbx	%ebx

%r9	%r9d

%rcx	%ecx

%r10	%r10d

%rdx	%edx

%r11	%r11d

%rsi	%esi

%r12	%r12d

%rdi	%edi

%r13	%r13d

%rsp	%esp

%r14	%r14d

%rbp	%ebp

%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)



Some History: IA32 Registers

general purpose

%eax	32 bit	%ax	%ah	%al
%ecx		%cx	%ch	%cl
%edx		%dx	%dh	%dl
%ebx		%bx	%bh	%bl
%esi		%si		
%edi		%di		
%esp		%sp		
%ebp		%bp		

16-bit virtual registers
(backwards compatibility)

Origin
(mostly obsolete)

accumulate

counter

data

base

source index

destination index

stack pointer

base pointer

Moving Data

Moving Data

movq Source, Dest:

move data shows how much data we move

Operand Types

- **Immediate:** Constant integer data
 - Example: \$0x400, \$-533 $movq \$0, %rax$
 - Like C constant, but prefixed with '\$'
 - Encoded with 1, 2, or 4 bytes
- **Register:** One of 16 integer registers

move %rax, %rbx: Example: **%rax**, **%r13**

we take whatever is in %rax: But **%rsp** reserved for special use

we put into %rcx: Others have special uses for particular instructions

- **Memory:** 8 consecutive bytes of memory at address given by register
 - Simplest example: (**%rax**)
 - Various other “address modes”

%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

%rbp

%rN

movq Operand Combinations

immediat argument

	Source	Dest	Src,Dest	C Analog
movq	Imm	Reg	movq \$0x4,%rax	temp = 0x4;
		Mem	movq \$-147,(%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
	Mem	Mem	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

WE CAN'T MOVE FROM MEMORY TO MEMORY

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

■ Normal (R) $\text{Mem}[\text{Reg}[R]]$

- Register R specifies memory address
- Aha! Pointer dereferencing in C

movq (%rcx), %rax

*pointer we
dereference*

■ Displacement D(R) $\text{Mem}[\text{Reg}[R]+D]$

- Register R specifies start of memory region
- Constant displacement D specifies offset

movq 8(%rbp), %rdx

*Constant * (rbp+8)
we don't know the kind of pointer dealing with it*

Example of Simple Addressing Modes

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

name of functio

Swap:

movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)

— ret (return statement)

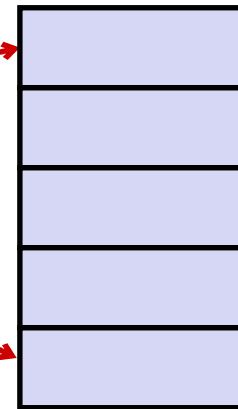
Understanding Swap()

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

Registers

%rdi	
%rsi	
%rax	
%rdx	

Memory



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

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

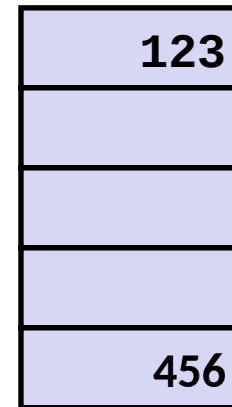
*Code refers new
and putting it to
them*

Understanding Swap()

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory



swap:

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

✓ look at
address 120

Address

0x120

0x118

0x110

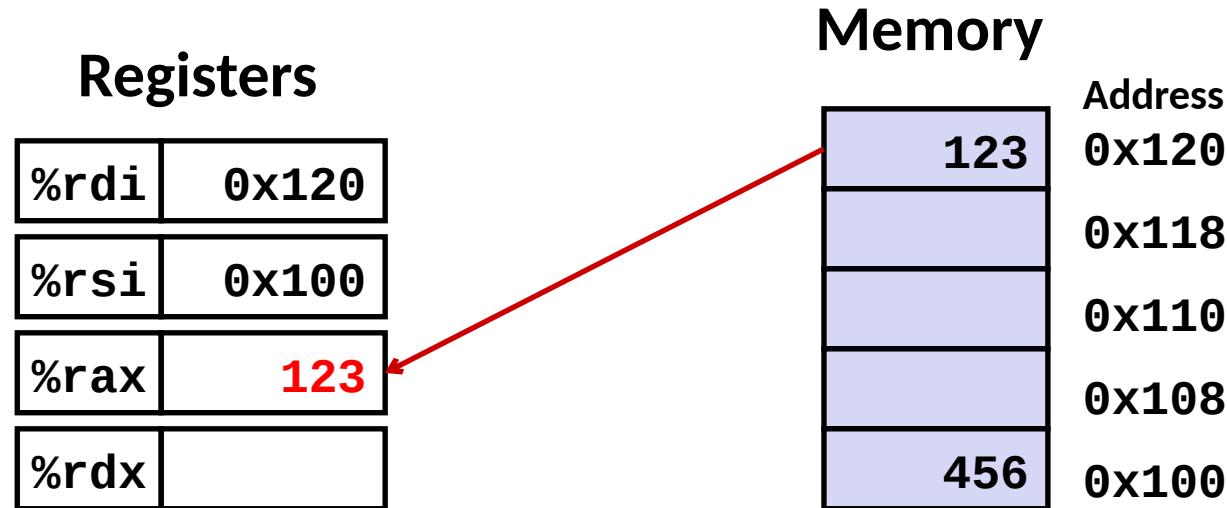
0x108

0x100

location 120 ← xp

location 100 ← yp

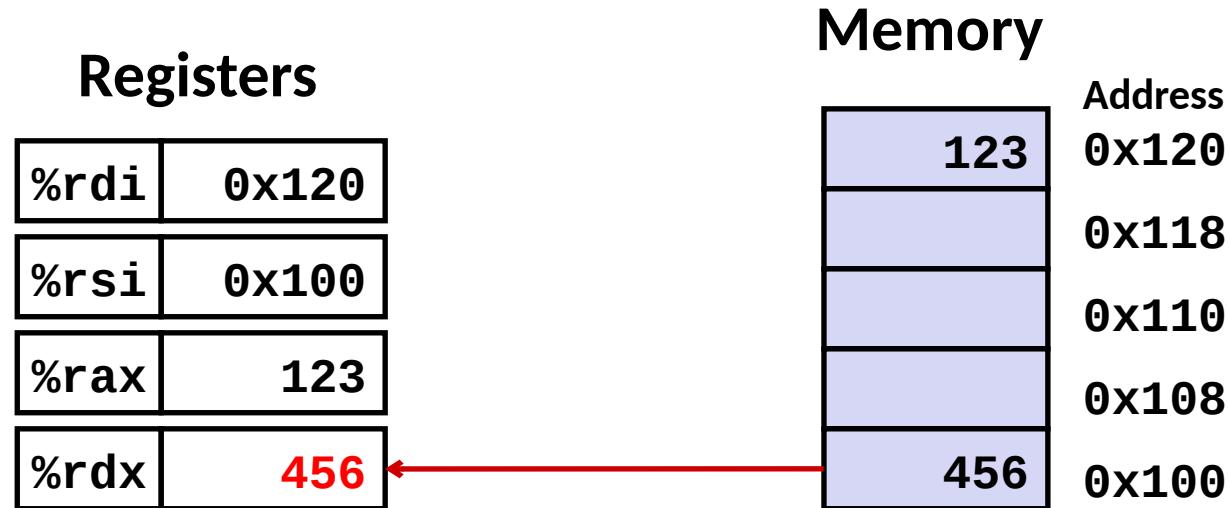
Understanding Swap()



swap:

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

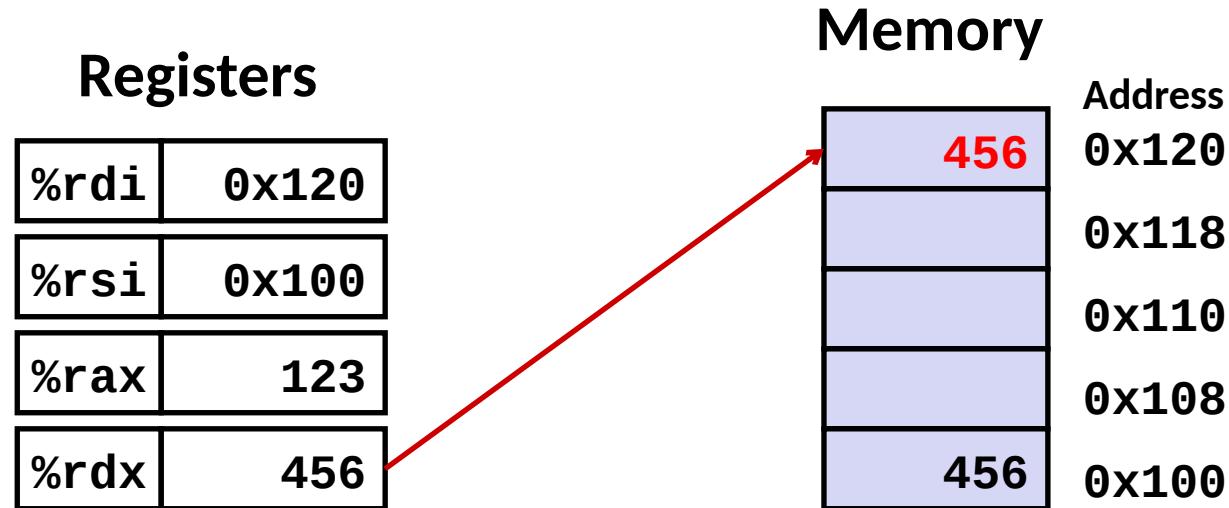
Understanding Swap()



swap:

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

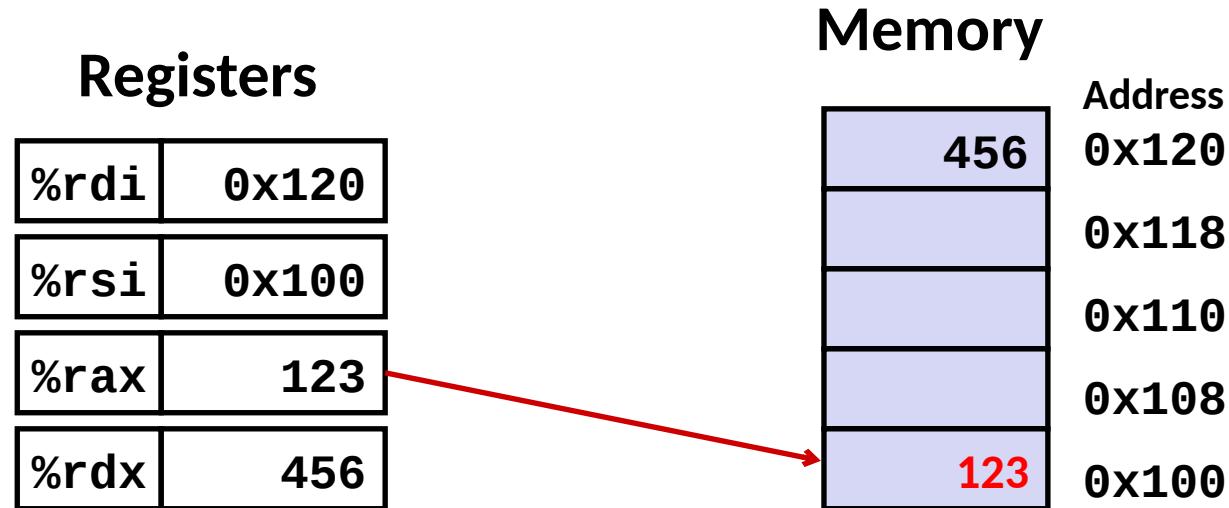
Understanding Swap()



swap:

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

Understanding Swap()



swap:

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

Simple Memory Addressing Modes

■ Normal (R) $\text{Mem}[\text{Reg}[R]]$

- Register R specifies memory address
- Aha! Pointer dereferencing in C

`movq (%rcx), %rax`

■ Displacement D(R) $\text{Mem}[\text{Reg}[R]+D]$

- Register R specifies start of memory region
- Constant displacement D specifies offset

`movq 8(%rbp), %rdx` ~~(%rbp+8)~~

Complete Memory Addressing Modes

■ Most General Form

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

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8

$\begin{array}{c} y_2=2^0 \\ y_1=1^0 \\ y_0=0^1 \\ \hline y=2^0+1^0+0^1=1 \end{array}$

$\begin{array}{c} y_2=2^0 \\ y_1=1^0 \\ y_0=0^1 \\ \hline y=2^0+1^0+0^1=1 \end{array}$

■ Special Cases

(Rb,Ri)

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

~~D(Rb,Ri)~~

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

(Rb,Ri,S)

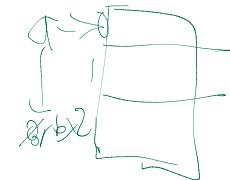
$\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri]]$

$0x08$

$0x4008 + 8$

$4008 + 8 = 4016$

Arm: $2(0brbx)$



$8(%rbp)$ *should be %rbp rax in 86-84*

$8(%rbp,%eax)$

Rb

$(4 * %eax) + %rbp + 8$

$(2 * 0x04) + (0x4000)$

$0x08 + 0x4000$

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

her values

Common mistake

$$2 \cdot 0x8 = 0x16$$

$y = \text{left}_{\text{shf}}$

$$2 \cdot 0x8 \xrightarrow{\downarrow} \text{On/O} \xrightarrow{f=15} 1111 \\ (6)_{10} = 010 \quad 10000 \approx \\ 100000 \approx \begin{matrix} y \\ | \\ 1 \end{matrix} \quad 10$$

Expression	Address Computation	Address
0x8(%rdx)	$0xf000 + 0x8$	0xf008
(%rdx,%rcx)	$0xf000 + 0x100$	0xf100
(%rdx,%rcx,4)	$0xf000 + 4 * 0x100$	0xf400
0x80(%rdx,2)	$2 * 0xf000 + 0x80$	0x1e080

הנתקן

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Address Computation Instruction

(load effective address)

■ **leaq Src, Dst**

- Src is address mode expression
- Set Dst to address denoted by expression

■ **Uses**

- Computing addresses without a memory reference
 - E.g., translation of $p = \&x[i]$;
- Computing arithmetic expressions of the form $x + k^*y$
 - $k = 1, 2, 4, \text{ or } 8$

■ **Example**

```
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

** rdi + rdi*

shift by 2 | if DOES NOT dereference here

write the result of into Rax

Another ex.
 $0111\ 1111 = 127$
 $1111\ 0110 = 284$ since it's
 $284 > 127$

$1011\ 1111 = -127 + 63 = -65 \neq 2$ since it -127 > -128 then
 $0111\ 1110 = 126$
 $1111\ 1110 = -2$ by 2 multiply

Some Arithmetic Operations

■ Two Operand Instructions:

Format

Computation

$$-2^7 = -128$$

addq Src,Dest Dest = Dest + Src

subq Src,Dest Dest = Dest $\underline{-}$ Src (subtraction)

imulq Src,Dest Dest = Dest * Src

salq Src,Dest Dest = Dest \ll Src (left shift)

sard Src,Dest Dest = Dest $\underline{\gg}$ Src (copy right)

shrq Src,Dest Dest = Dest \gg Src

xorq Src,Dest Dest = Dest \wedge Src

andq Src,Dest Dest = Dest & Src

orq Src,Dest Dest = Dest | Src

bitwise operators

Also called Arithmetic

Logical

Two types of left

Shifts

- arithmetic

- logical

$0000\ 1110 \Rightarrow 000000$
 copy leftmost 4 / 2
 \ to right

$\downarrow 0001\ 010 \gg 1 = 00001$

$$-111\ 12 \neq 69$$

$$\gg [1] 100010 \\ = -127 + 63 \\ = -64$$

■ Watch out for argument order!

■ No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

■ One Operand Instructions

{ incq Dest Dest = Dest + 1
decq Dest Dest = Dest - 1
negq Dest Dest = - Dest
notq Dest Dest = ~Dest
*bitwise
nor*

■ See book for more instructions

Arithmetic Expression Example

```
long arith  
(long x, long y, long z)  
{  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

arith:

```
    leaq    (%rdi,%rsi), %rax  
    addq    %rdx, %rax  
    leaq    (%rsi,%rsi,2), %rdx  
    salq    $4, %rdx  
    leaq    4(%rdi,%rdx), %rcx  
    imulq   %rcx, %rax  
    ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
 - But, only used once

Understanding Arithmetic Expression

Example

```
long arith  
(long x, long y, long z)  
{  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

If it is ok
to have it
One reference

arith:
leaq
addq
leaq
salq
leaq
imulq
ret

all on the stack

NO DE REFERNCING

push into %rax

$(%rdi, \%rsi), \%rax \# t1$

$\$ \%rdx, \%rax D=D+r \# t2$

$(%rsi, \%rsi, 2), \%rdx / / 3^y$

$\$4, \%rdx 2^y+4^y \rightarrow \text{shift left} \# t4 2^4=16$

$4(%rdi, \%rdx), \%rcx \# t5$

$\%rcx, \%rax +4 \# rval$

D=D+s

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Machine Programming I: Summary

- **History of Intel processors and architectures**
 - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
 - New forms of visible state: program counter, registers, ...
 - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **Assembly Basics: Registers, operands, move**
 - The x86-64 move instructions cover wide range of data movement forms
- **Arithmetic**
 - C compiler will figure out different instruction combinations to carry out computation