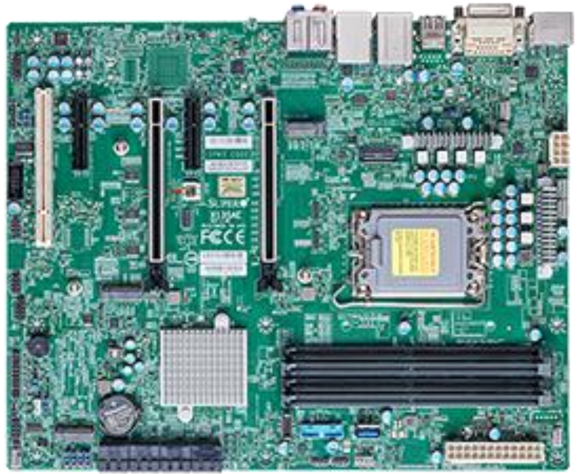
COMP2100 Week 7

Assembly Language in Modern Computer Systems



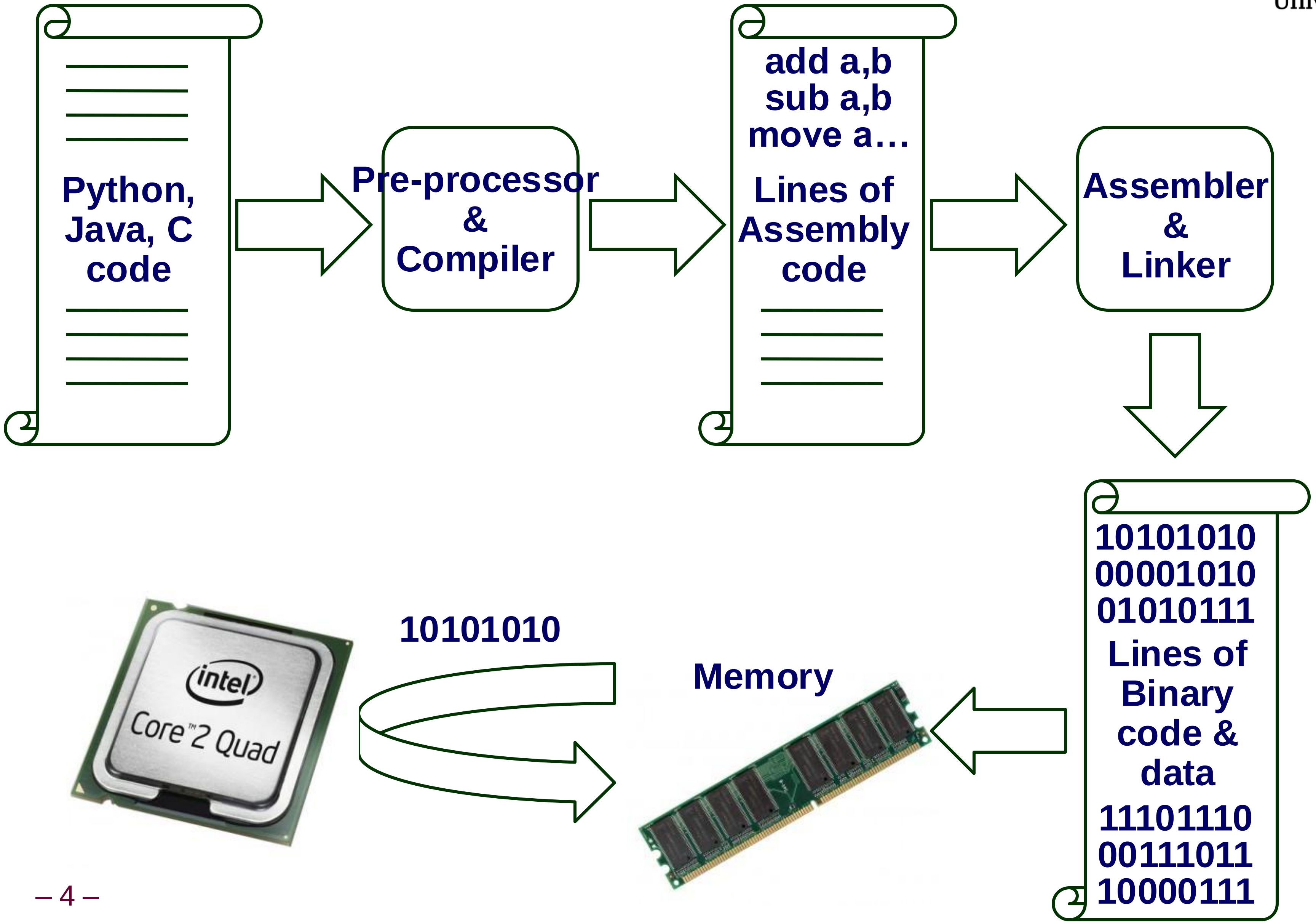
Announcements

- **Midterm exam this week in person at your practicals**
  - **1 ½ hours, safe exam browser mode, closed book, no electronic aids, tutor provides password**
- **Lab #1**
  - **Progress mark 1.4 due Sept 6**
  - **All code due Sept 13, lab closes**
  - **code explanations Weeks 6-8**
- **Bomb Lab #2 to be released on iLearn in week 8**
  - **Tutors will introduce in workshops**
  - **Given a personalized binary executable, solve 5 stages, Learn to read assembly and use the gdb debugger, Avoid "exploding" the bomb by providing right password at each stage**
  - **Important: Set a breakpoint to avoid triggering the bomb message to server and losing marks**

Announcements

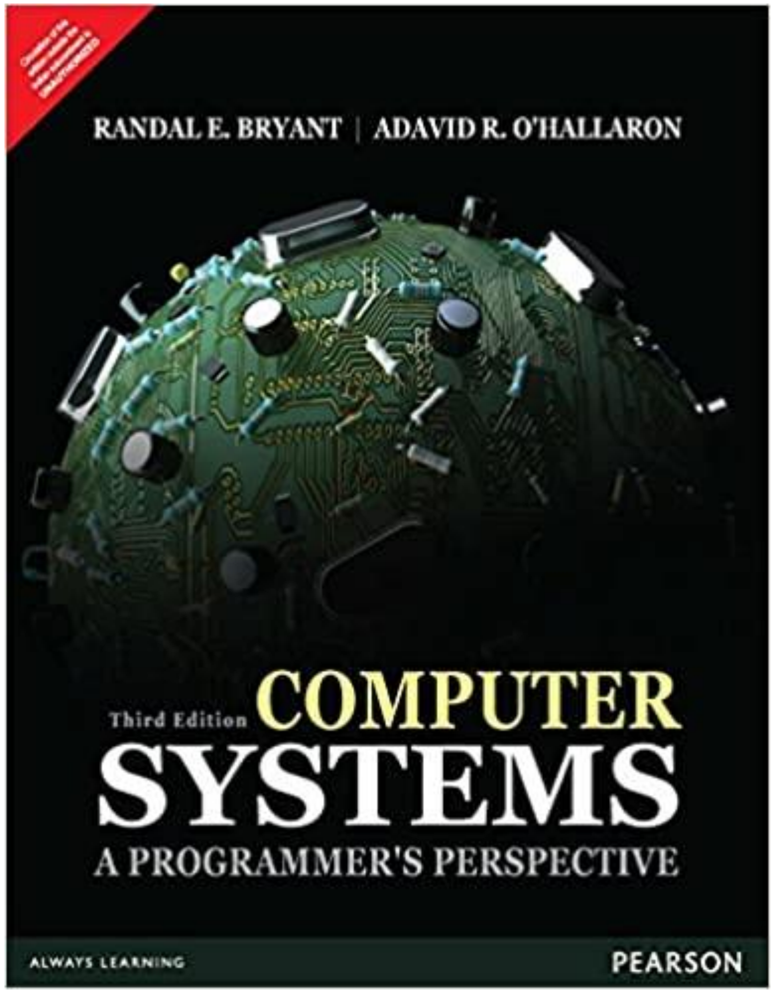
- Read Chapter 3.1-3.12 (except 3.11) and do practice problems
- Week 7 Quiz available, due after the break

2100 In a Nutshell



Hints for 2nd half of class

- Read textbook: Computer Systems, Chapters 2 & 3 in detail (skip 3.11)
  - Do the many practice problems in the textbook – they have solutions
- Start early on Bomb Lab
  - Stay on time with the progress marks – can't afford to fall behind
- Do the quizzes on time
  - Can't afford to fall behind


RANDAL E. BRYANT | DAVID R. O'HALLARON
Third Edition
COMPUTER SYSTEMS
A PROGRAMMER'S PERSPECTIVE
ALWAYS LEARNING
PEARSON

Weeks 7-12 Approximate Timeline

- CPU: Software meets the hardware (Weeks 7-8)
  - Introduction to Assembly
  - Conditional execution (branches)
  - Procedure calls
- Memory: Data meets the hardware (Weeks 9-10)
  - Arrays and memory
  - Buffer overflow exploits
  - Struct and union
  - Memory and cache (Week 11)
- System: Software meets the OS (Week 12)
  - IO
  - Virtual memory
  - Exceptional control flow

Java

C

Assembly

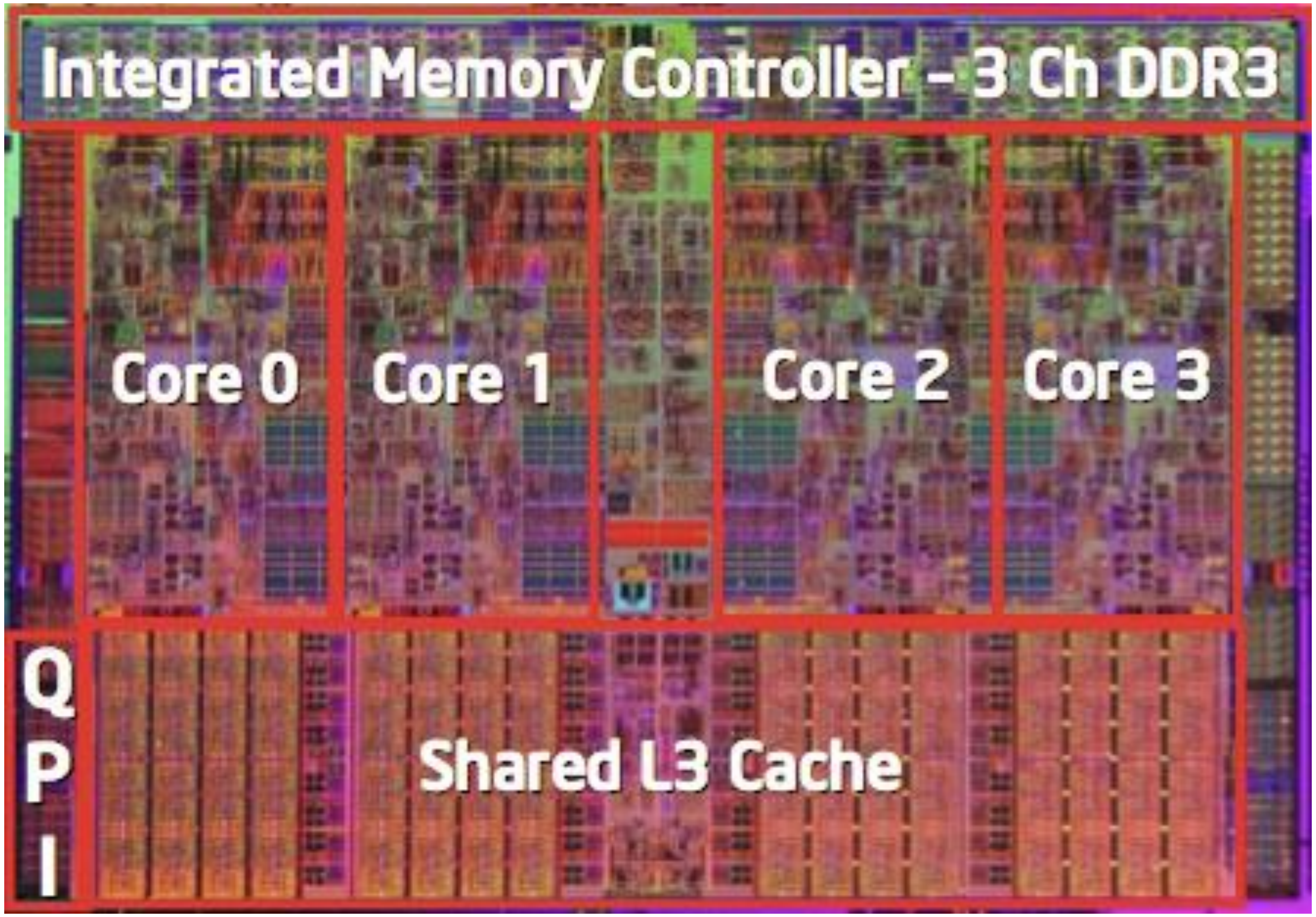
x86 Assembly Language

- The set of assembly language instructions supported by a processor is called an Instruction Set Architecture (ISA)
- x86 ISA is specific to Intel's family of x86 CPUs
    - PC desktops, Cloud servers, Laptops
    - Complex Instruction Set (CISC)
    - AMD chips too
- There are other types of ISAs, e.g. low power ARM processors
    - Mobile smartphones, Laptops
    - Reduced Instruction Set (RISC)

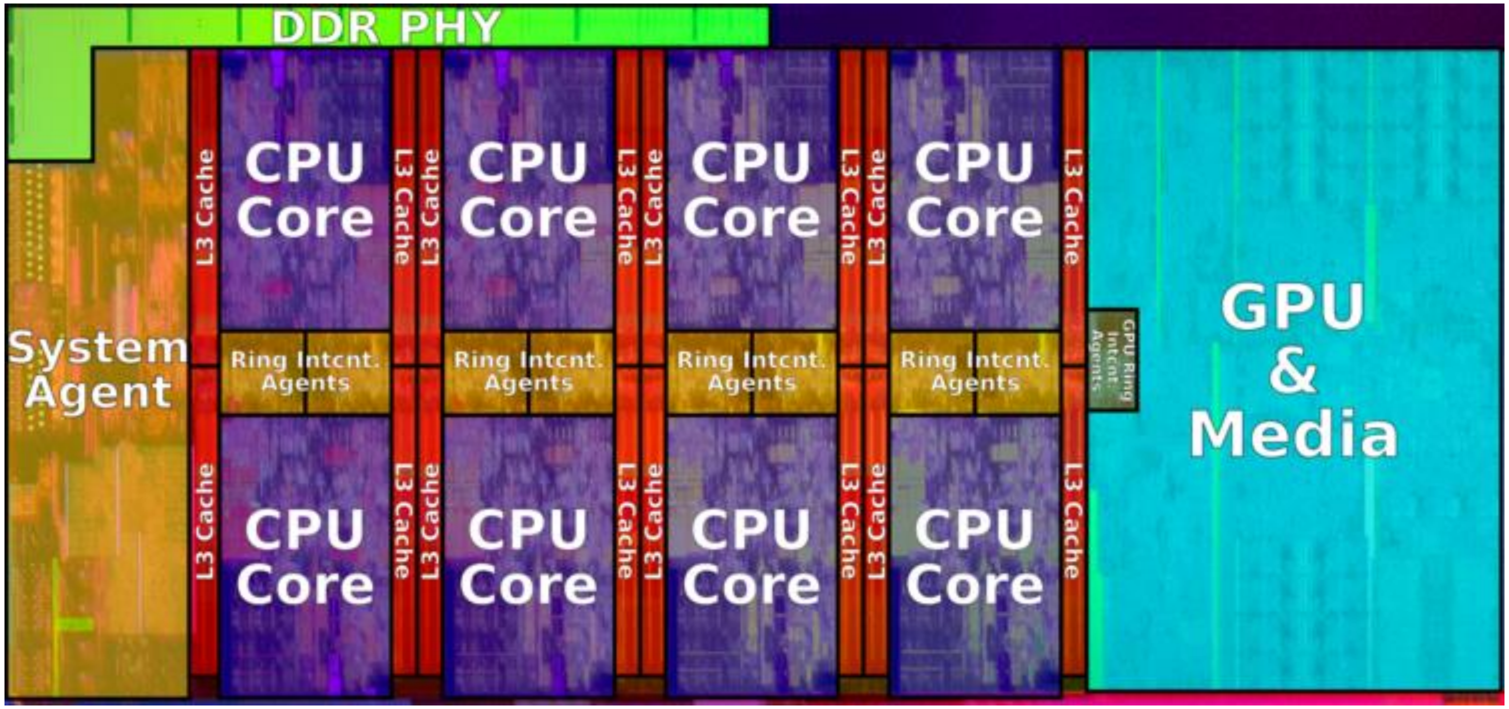
Intel Microprocessor (CPU) Evolution

Name	Date	Transistors	MHz	Feature
8086	1978	29K	5-10	16-bit
80386	1985	275K	16-33	32-bit
Pentium 4F	2004	125M	2800-3800	X86-64
Core i7	2008	731M	2667-3333	
Core i7 3960X	2011	2.3G	3333	6 Cores 140 GFlop
Core i9 9990XE	2019	~10G	4-5 GHz	14 cores 28 HyperThread

Intel Core i7 Die



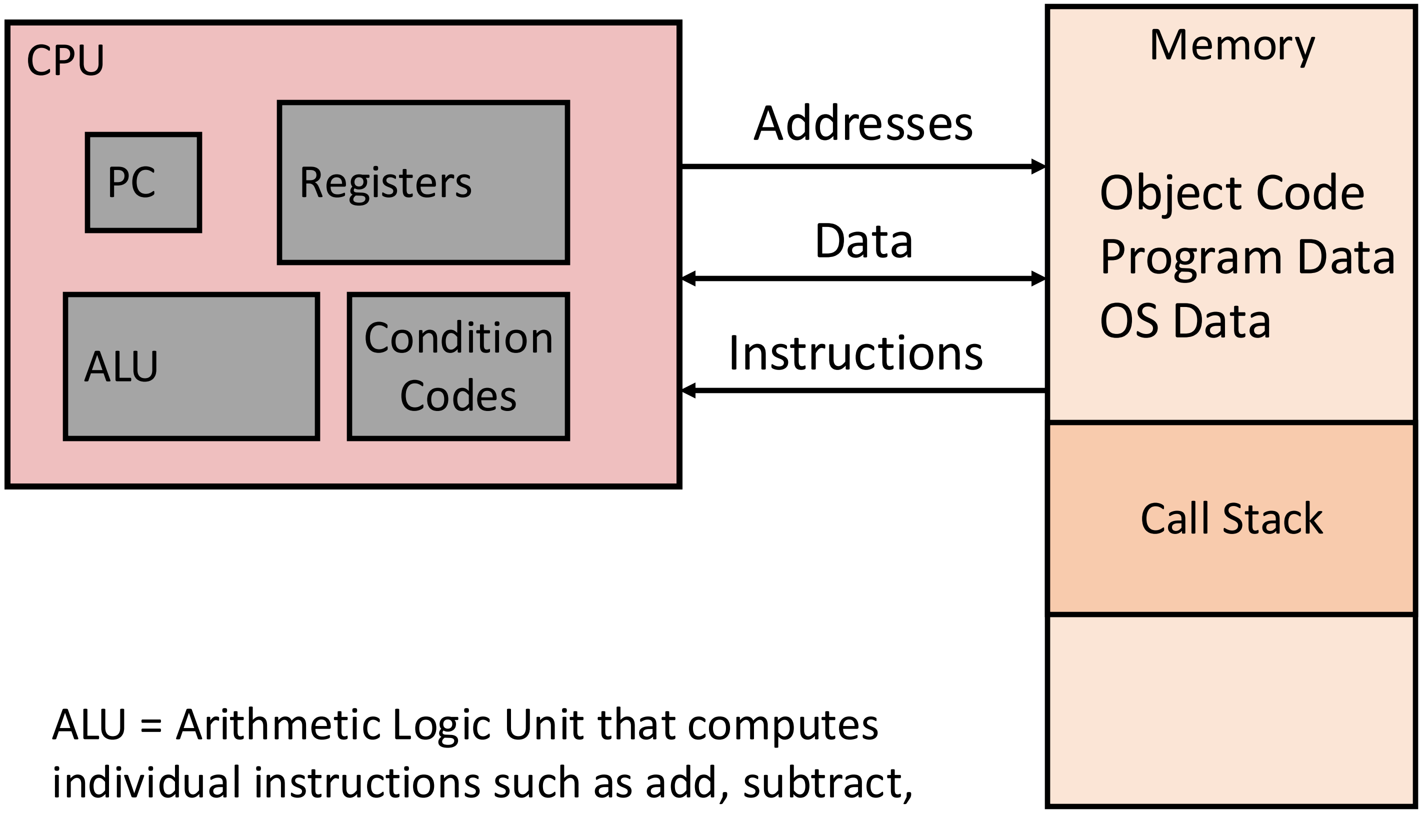
Intel Core i9 "Coffee Lake" Die



What x86 assembly code looks like…

```
diff:
        movq    %rdi, %rax
        subq    %rsi, %rax
        ret
```

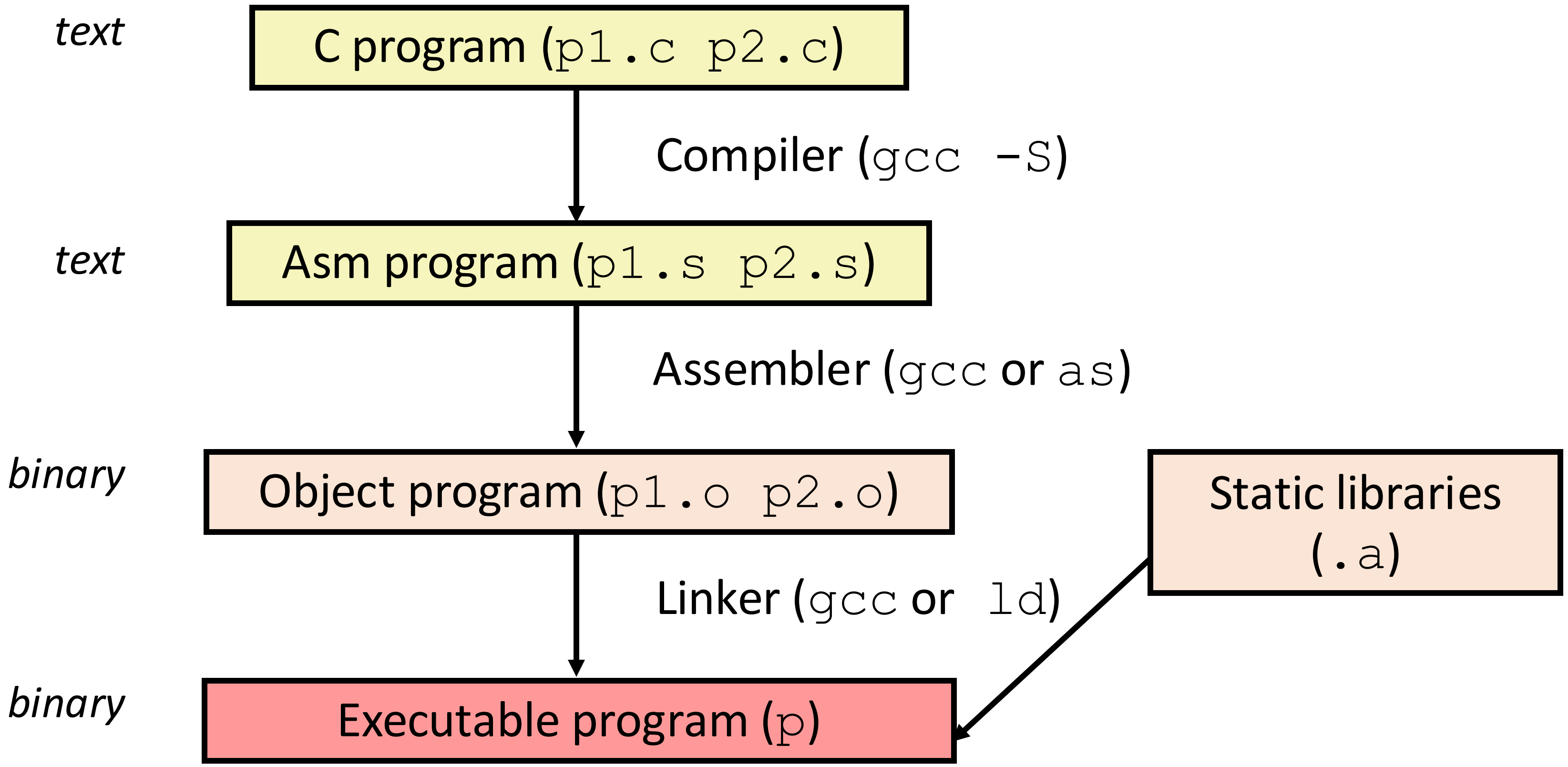
Assembly Programmer's View



ALU = Arithmetic Logic Unit that computes individual instructions such as add, subtract, multiple, and divide
PC = program counter, stores memory address of the next instruction to fetch

Compiling C into Object Code

- `gcc -O1 p1.c p2.c -o p`



Relating High level C To Low level Assembly

C Code

```
long diff(long x, long y)
{
  long t = x-y;
  return t;
}
```

```
gcc -O1 -S diff.c
```

Produces file `diff.s`

Generated x86-64 Assembly

```
diff:
        movq    %rdi, %rax
        subq    %rsi, %rax
        ret
```

AT&T/GNU Syntax:
  mov from, to
  sub   value, away from

Size: b: byte (8) w: word (16)
  l: longword (32) q: quadword (64)

Assembly characteristics

- i.e. Machine characteristics
- Data
  - Integers, float
  - No arrays, structs, classes, etc
- Basic Operations
  - Arithmetic
  - Move (copy)
  - Control flow

Generated x64 Assembly

```
diff:
        movq      %rdi, %rax
        subq      %rsi, %rax
        ret
```

Object Code for diff

```
(gdb) x/7xb diff
0x5fa <diff>: 0x48 0x89 0xf8
              0x48 0x29 0xf0
              0xc3
```

- Total of 7 bytes
- Each instruction 1 or 3 bytes
- Starts at address `0x5fa`

Machine Instruction Example

C Code

```
long t = x-y;
```

Assembly

```
subq %rsi,%rax
```

Similar to:

```
x -= y
```

More precisely:

```
long rax; //x
long rsi; //y
rax -= rsi //x-y
```

Object

```
5fd:    48 29 f0       subq   %rsi,%rax
```

Compile and dump object

```
$ gcc -O1 -o mdiff mdiff.c
$ objdump -d mdiff | less
$ gdb mdiff
(gdb) disass diff
(gdb) x/7xb diff
(gdb) quit
```

Disassembling Object Code - objdump

```
$ objdump -d mdiff
```

```
00000000000005fa <diff>:
 5fa:    48 89 f8             mov     %rdi,%rax
 5fd:    48 29 f0             sub     %rsi,%rax
 600:    c3                   retq
```

Disassembly in gdb Debugger

Object

```
0x5fa:
0x48
0x89
0xf8
0x48
0x29
0xf0
0xc3
```

Disassembled

```
Dump of assembler code for function diff:
   0x0…05fa <+0>:     mov    %rdi,%rax
   0x0…05fd <+3>:     sub    %rsi,%rax
   0x0…0600 <+6>:     retq
End of assembler dump.
```

```
$ gdb mdiff
(gdb) disass diff
(gdb) x/7xb diff
```

- Examine 7 hex bytes starting at `diff`

What Can be Disassembled?

- Any executable or object file – even from Windows!

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55                push   %ebp
30001001:  8b ec             mov    %esp,%ebp
30001003:  6a ff             push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

Assembly Basics

x86-64 Integer Registers

`%rax`	`%eax`	`%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`

- **`%rsp`** is special purpose – stack pointer.

Moving (Copying) Data

`movq` ***Source*, *Dest*:**

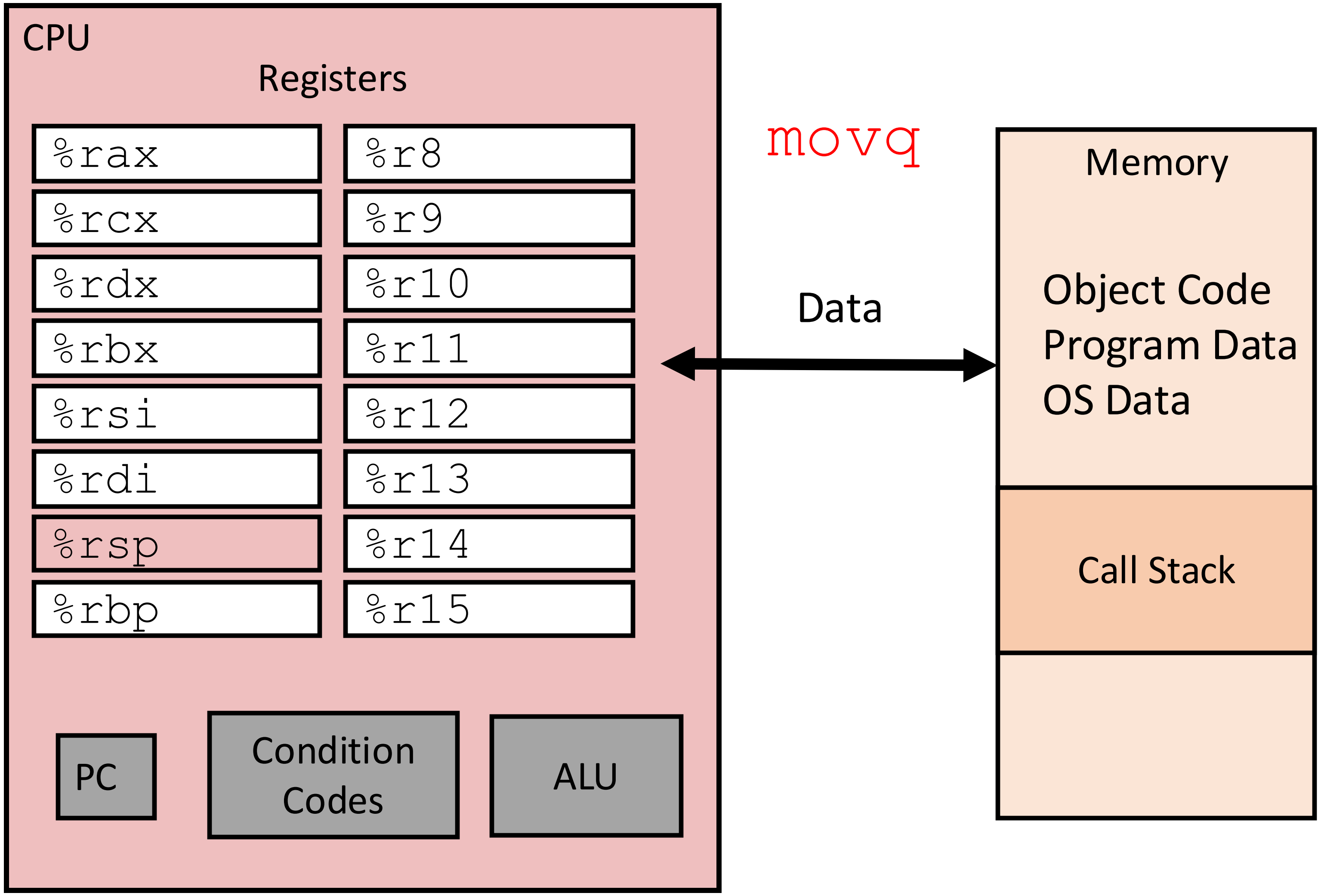
"Move quadword"

```
movq     %rdi, %rax
```

- Operand Types
  - ***Immediate:*** Constant integer
    - **`$0x400,  $-533`**
  - ***Register:*** One of 16 integer registers
    - **`%rax, %rdx`**
    - Reserved: **`%rsp`**
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rdi)`**
    - Various other "address modes"

%rax	%r8
%rcx	%r9
%rdx	%r10
%rbx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

Assembly Programmer's View



`movq` Operand Combinations

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

Cannot do memory-memory transfer with a single instruction

Understanding: %rax vs (%rax)

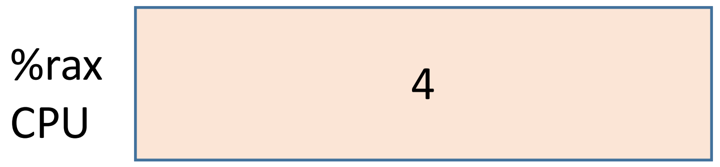
movq $0x4, %rax

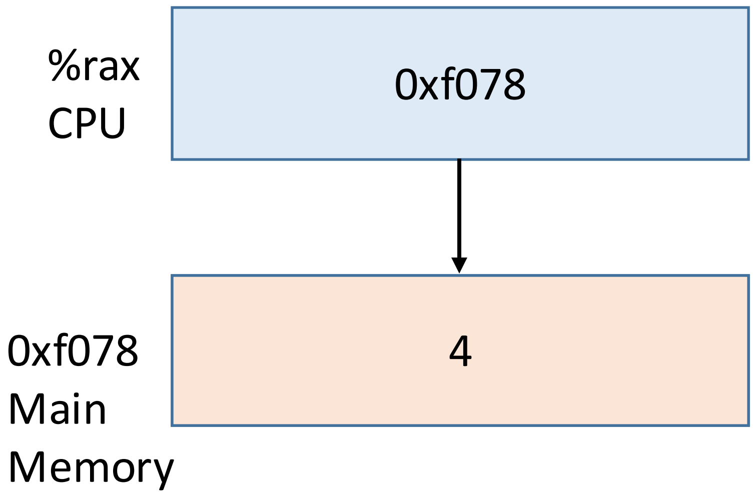
- The destination is register %rax

movq $0x4, (%rax)

- The destination is memory pointed to by %rax

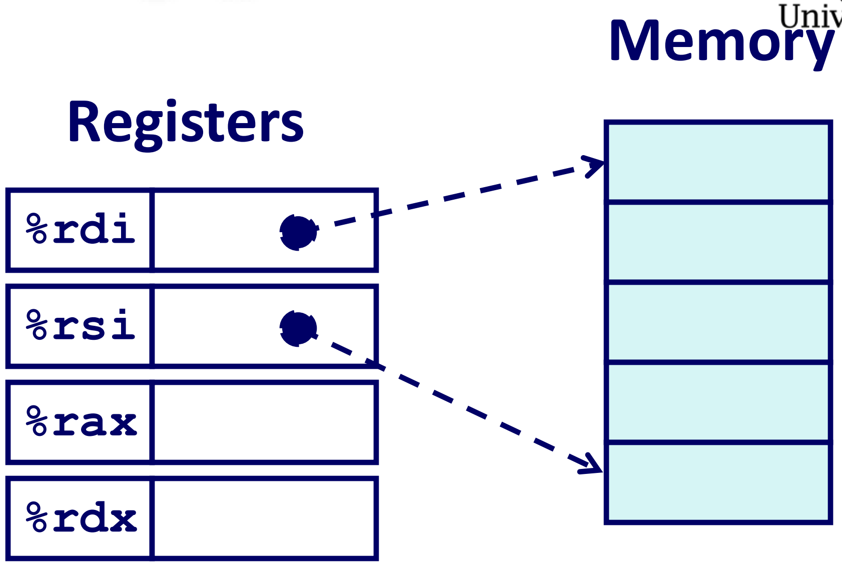
Suppose %rax contains 0xf078





Understanding Swap()

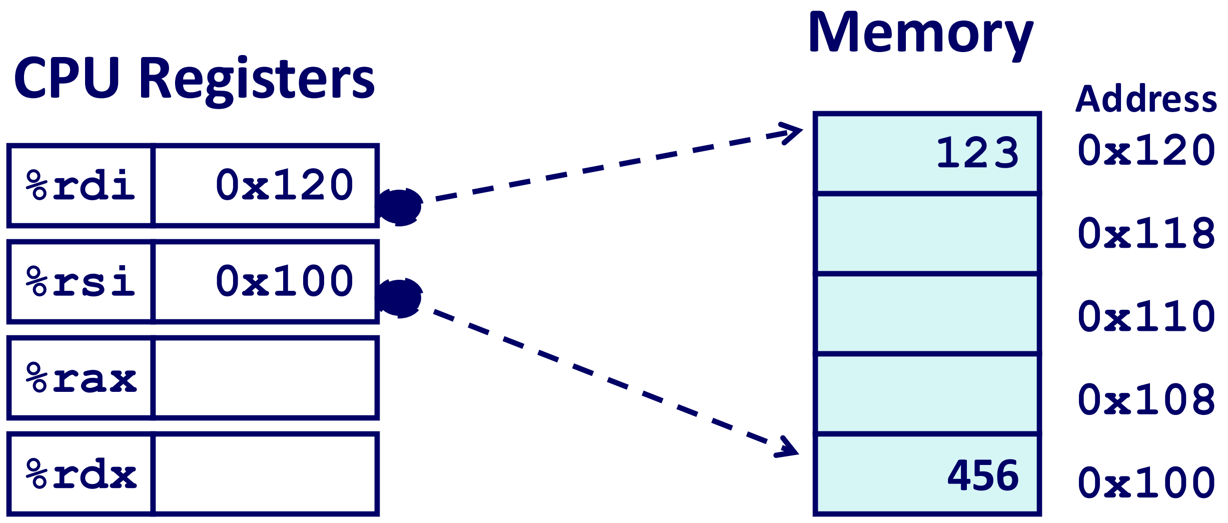
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

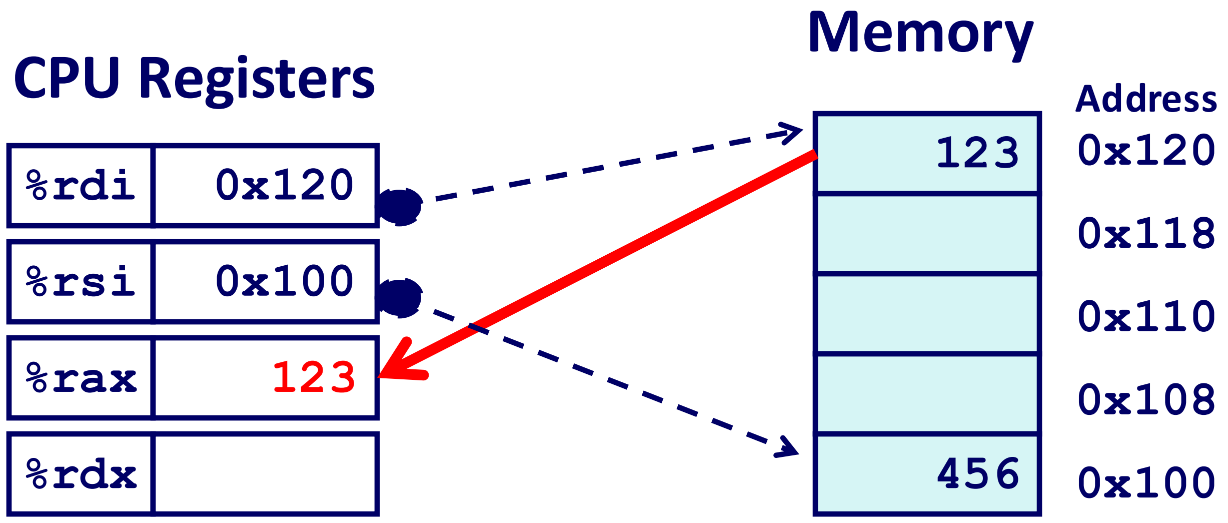
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



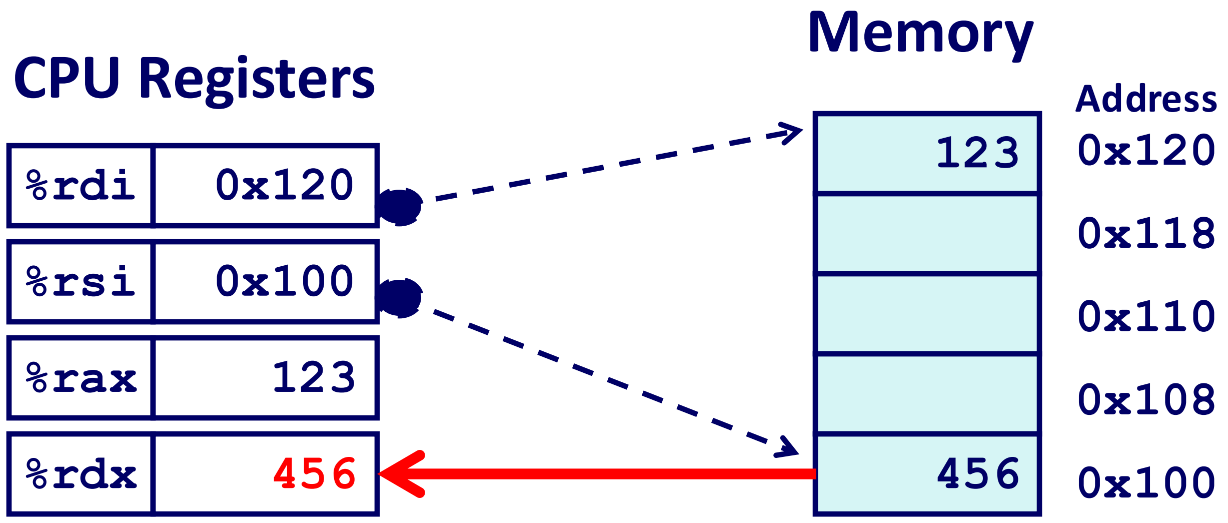
```
swap:
    movq     (%rdi), %rax    # t0 = *xp
    movq     (%rsi), %rdx    # t1 = *yp
    movq     %rdx, (%rdi)    # *xp = t1
    movq     %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()



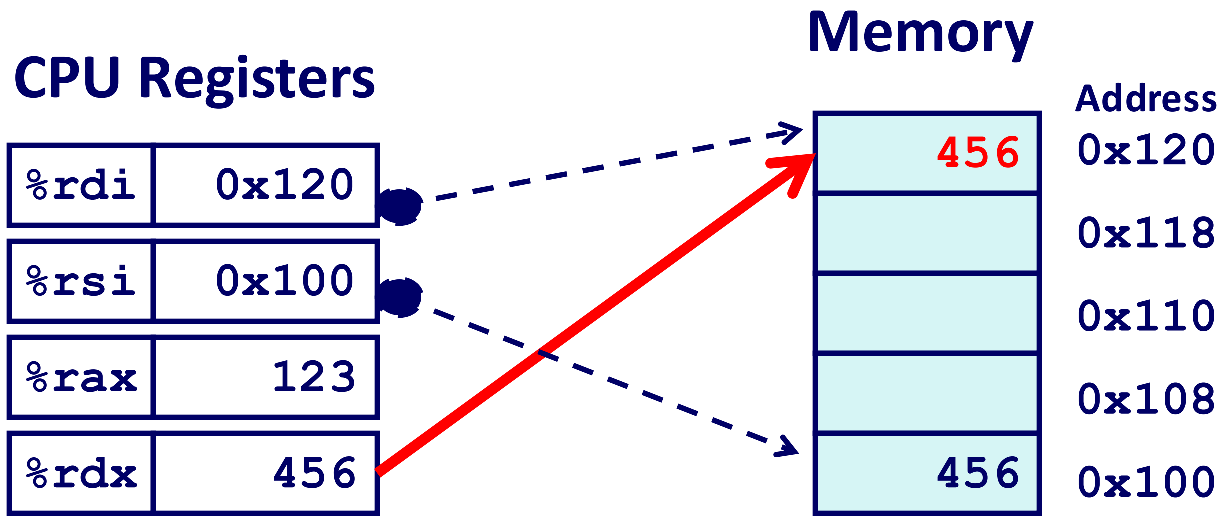
```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



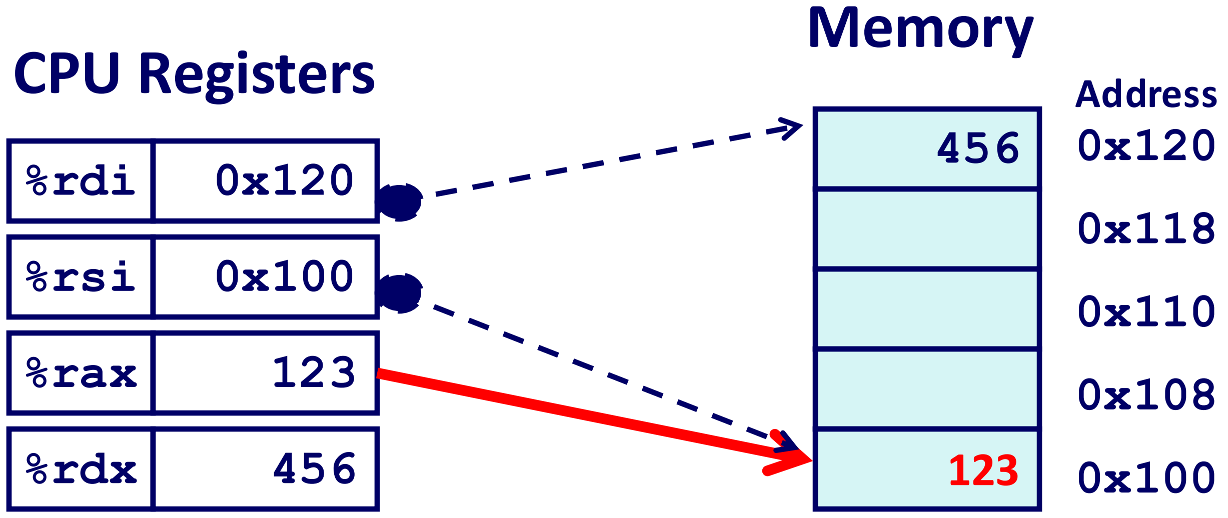
```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



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

Understanding Swap()



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

Moving different word sizes

- `movq %rax, %rdx`
  - Move a “quad” word (4*16 = 64 bits = 8 bytes) from register %rax to register %rdx
- `movl %eax, %edx`
  - Move a “long” word (2*16 = 32 bits = 4 bytes) from register %eax to register %edx
- `movw %ax, %dx`
  - Move a word (16 bits = 2 bytes) from register %ax to register %dx
- `movb %al, %dl`
  - Move a byte from register %al to register %dl

Complex/Indexed Addressing Modes

```
movq 24(%rdi,%rsi,4), %rax
```

This means:

- Move a quad word (8 bytes) from the memory location %rdi + 4*%rsi + 24 to register %rax

Most General Form

D(Rb,Ri,S) Mem[Reg[Rb]+S*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 (*why these numbers?*)

Indexed Addressing Modes (2)

MACQUARIE University

❿Special Cases

- **(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]]**
  - `movq (%rax,%rbx), %rdx`
- **D(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]+D]**
  - `movq %rdx, 12(%rax,%rbx)`
- **(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]]**
  - `movq (%rax,%rbx,8), %rdx`
- **(Rb)** **Mem[Reg[Rb]+S*Reg[Ri]]**
  - `movq %rdx, (%rax)`

`lea` Instruction for Address Computation

MACQUARIE University

`lea` = "Load effective address"

`leaq` *Src*,*Dest*

- *Src* is indexed address mode expression
- Set *Dest* (must be register) to value denoted by expressio

- Example:

```
leaq 10(%rdx, %rdx, 4), %rax
```

%rdx + 4*%rdx + 10
= 5*%rdx + 10

Therefore"%rax = 5 * %rdx + 10"

- Compare to:

```
movq 10(%rdx, %rdx, 4), %rax
```

means "%rax = Mem[5*%rdx + 10]

`lea` Instruction for Address Computation

MACQUARIE University

Uses

- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8.
- Computing address without doing memory reference
  - E.g., translation of `p = &x[i];`

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
```

Arithmetic

Some Arithmetic Operations

- Two Operand Instructions:

Format		***Computation***	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic (>> Signed)***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical (>> Unsigned)***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- Watch out for argument order!

MACQUARIE
University

Some Arithmetic Operations

- One Operand Instructions

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

- Lea instruction

`leaq`	*Address,Dest*	*Dest = Address*
`leaq`	$Imm(r_b,R_i,s),Dest$	$Dest = Imm + R[r_b] + R[r_i] * s$

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

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;
}
```

Parameters passed in via CPU registers:

```
%rdi = x
%rsi = y
%rdx = z
```

```
arith:
    leaq  (%rsi,%rsi,2), %rax # rax = rsi * 3
    salq  $4, %rax            # rax = rax*16
    leaq  4(%rdi,%rax), %rax  # rax = 4+rdi+rax
    addq  %rdi, %rsi          # rsi += rdi
    addq  %rdx, %rsi          # rsi += rdx
    imulq %rsi, %rax          # rax *= rsi
    ret                       # return rax
```

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;
}
```

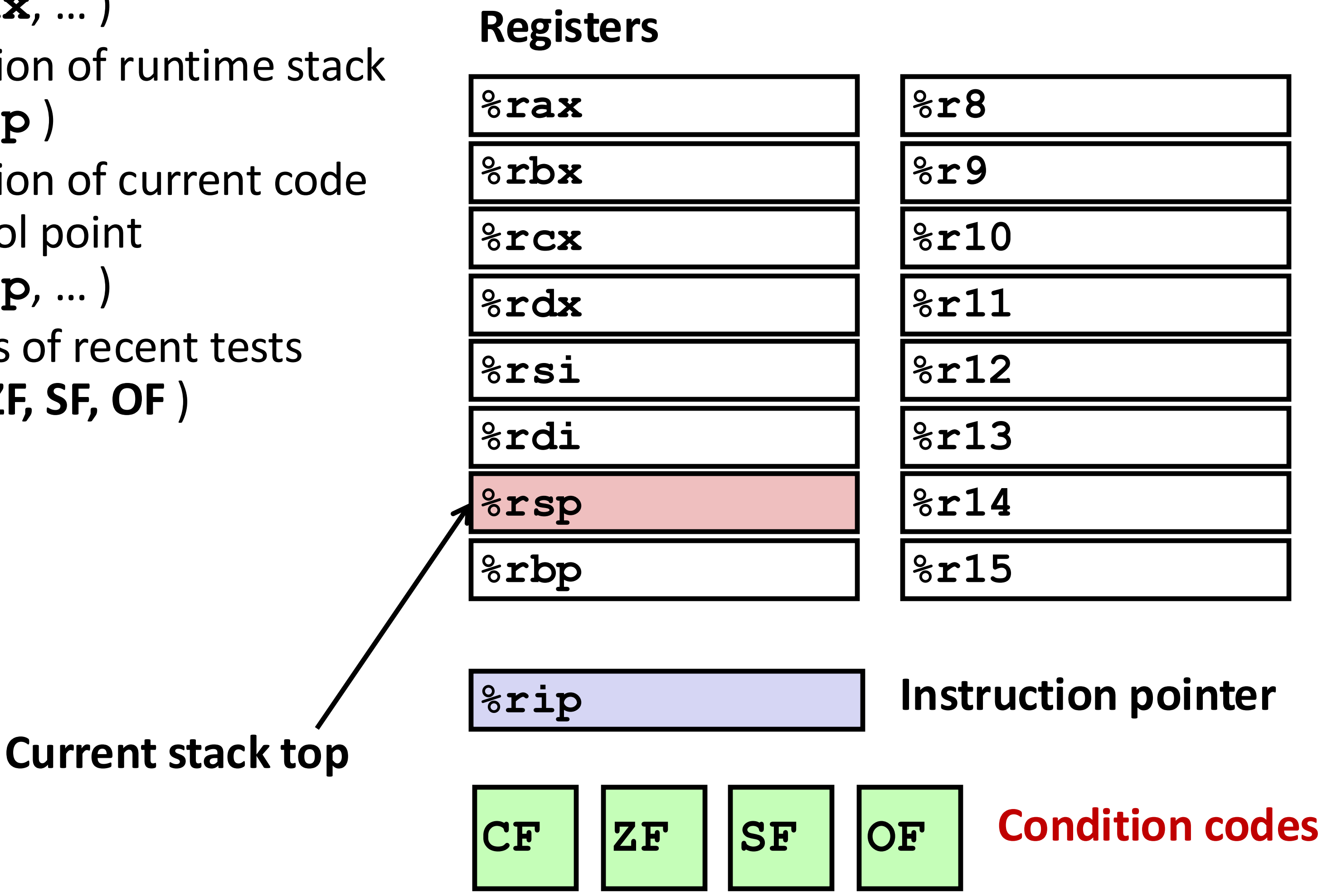
Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y** / t1 / t2: x+y
`%rdx`	Argument **z**
`%rax`	t5 / 4+x+y*48 / rval / t2*t5

```
arith:
    leaq   (%rsi,%rsi,2), %rax # rax = rsi * 3
    salq   $4, %rax            # rax = rax*16
    leaq   4(%rdi,%rax), %rax  # rax = 4+rdi+rax
    addq   %rdi, %rsi          # rsi += rdi
    addq   %rdx, %rsi          # rsi += rdx
    imulq  %rsi, %rax          # rax *= rsi
    ret                        # return rax
```

Condition Codes

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data
    (**`%rax`**, …)
  - Location of runtime stack
    (**`%rsp`**)
  - Location of current code control point
    (**`%rip`**, …)
  - Status of recent tests
    (**CF, ZF, SF, OF**)



Condition Codes (Implicit Setting)

- Single bit registers

•**CF**	Carry Flag (for unsigned)
•**SF**	Sign Flag (for signed)
•**ZF**	Zero Flag
•**OF**	Overflow Flag (for signed)

`addq` *Src,Dest* ↔ **`t = a+b`**

CF set if carry out (unsigned overflow)

ZF set if `t == 0`

SF set if `t < 0` (as signed)

OF set if two's-complement (signed) overflow

```
(a>0 && b>0 && t<0) ||
(a<0 && b<0 && t>=0)
```

- Not set by **`leaq`** instruction
- Full documentation , link on course website

Condition Codes (Explicit Setting: Compare)

MACQUARIE University

- Explicit Setting by Compare Instruction
  - **`cmpl/cmpq`** *Src2, Src1*
  - **`cmpl b,a`**
    - like computing **`a-b`** without setting destination

  - **CF set** if carry out (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

Condition Codes (Explicit Setting: Test)

MACQUARIE University

- Explicit Setting by Test instruction
  - **`testl`/`testq`** *Src2*, *Src1*

  - **`testl b,a`**
    - like computing **`a&b`** without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

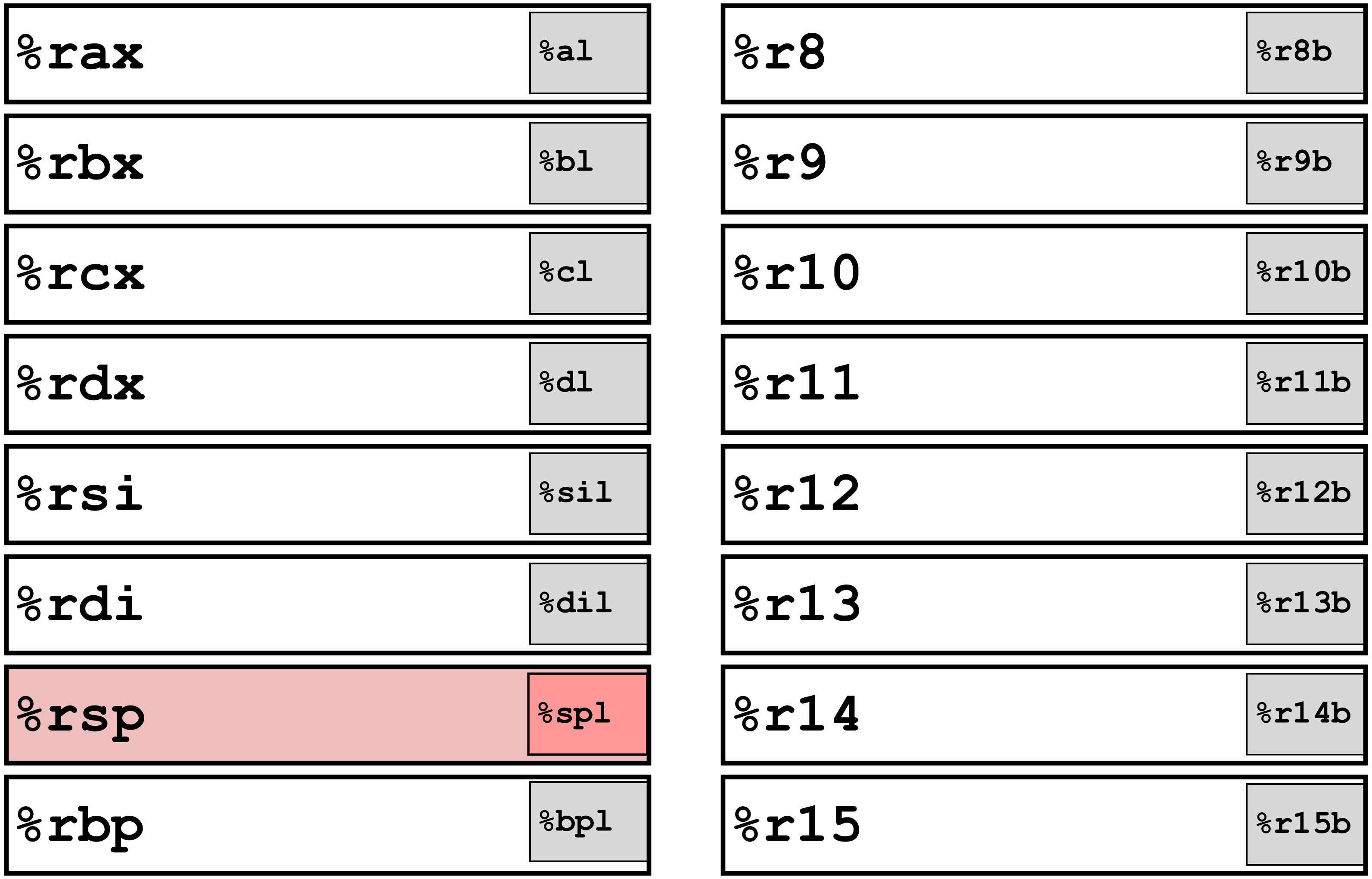
  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Reading Condition Codes

- setX dest // e.g. `setl %al`
  - Set lower order byte of destination register to 0 or 1 based on condition codes
    - Remaining 7 bytes **unchanged**

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

x86-64 Integer Registers



- Can reference low-order byte

Reading Condition Codes (Cont.)

- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **movzbl** to finish job
  - 32-bit instruction sets upper 32 bits to 0.

```
long gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
    cmpq   %rsi, %rdi    # flags = compare(x, y)
    setg   %al           # al = flags.g // al = x > y
    movzbl %al, %eax     # Zero rest of %rax
                         #  // rax = (unsigned) al
    ret
```

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branch Example (Old Style)

- If-then-else converted to assembly for

```
long absdiff
   (long x, long y)
{
   long result;
   if (x > y)
      result = x-y;
   else
      result = y-x;
   return result;
}
```

```
absdiff:
   cmpq    %rsi, %rdi   # y:x
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:               # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rax`	Return value

Backup

Today:
Machine-level programming I

- Core Ideas
- Intel Architecture
- C, Assembly and Machine Code
- Assembly: Registers, operands, move

Core Ideas

- Instruction
- Compiler
- Bug – compiler, hardware, programmer
- Programmer bugs:
  - Abstraction, limits e.g. int wrap around, array bounds
  - Semantics e.g. x=1, y=2; if (x & y)…
  - Inefficient code

- Approach
  - Assembly language
  - CPU model
  - Code from C compiler
  - Use x86-64

Key terms

- Instruction Set Architecture (ISA)

- Microarchitecture

Simple Memory Addressing Modes

- Normal `(r)` M[R[r]]
  - Register r contains memory address (a pointer)

```
movq (%rcx),%rax
```

- Displacement `Imm(r)` M[Imm+R[r]]
  - Register r specifies start of memory region
  - Constant displacement Imm specifies offset

```
movq 8(%rdi),%rdx
```

Using Simple Addressing Modes

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

```
swap:
     movl    (%rdi), %eax
     movl    (%rsi), %edx
     movl    %edx, (%rdi)
     movl    %eax, (%rsi)
     ret
```

Understanding Swap

MACQUARIE
University

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

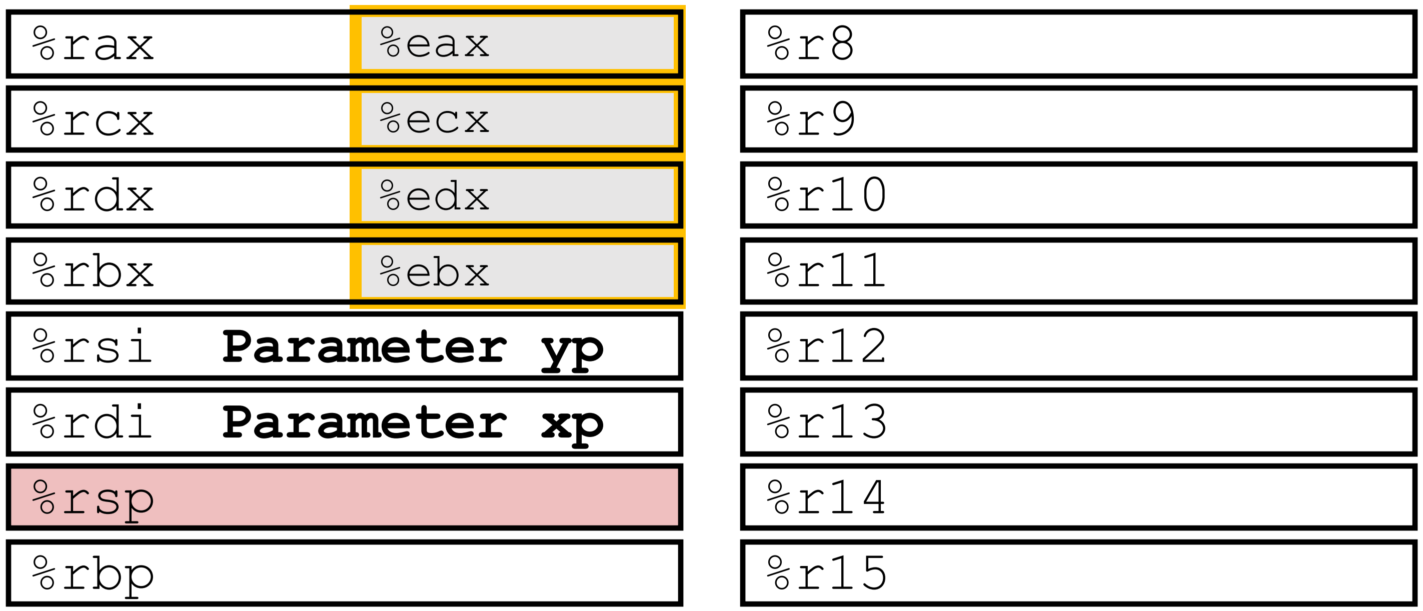
Integer/pointers parameters (x64-SysV):

(rdi, rsi, rdx, rcx, r8, r9)

Register	Value
%rdi	xp
%rsi	yp
%edx	t0
%eax	t1

```
swap:
    movl (%rdi), %eax # eax = *rdi
    movl (%rsi), %edx # edx = *rsi
    movl %edx, (%rdi) # *rdi = edx
    movl %eax, (%rsi) # *rsi = eax
    ret
```

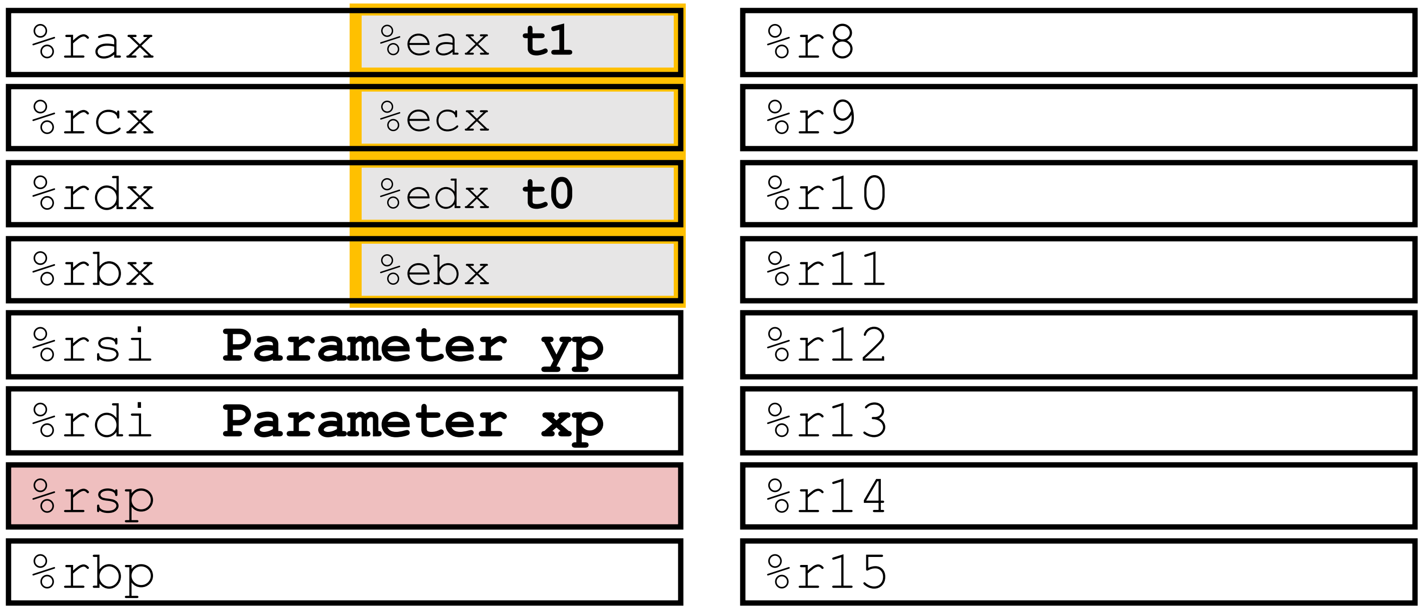
Understanding Swap



Register	Value
%rdi	xp
%rsi	yp
%edx	t0
%eax	t1

```
swap:
   movl (%rdi), %eax # eax = *rdi
   movl (%rsi), %edx # edx = *rsi
   movl %edx, (%rdi) # *rdi = edx
   movl %eax, (%rsi) # *rsi = eax
   ret
```

Understanding Swap



Register	Value
%rdi	xp
%rsi	yp
%edx	t0
%eax	t1

```
swap:
    movl (%rdi), %eax # eax = *rdi
    movl (%rsi), %edx # edx = *rsi
    movl %edx, (%rdi) # *rdi = edx
    movl %eax, (%rsi) # *rsi = eax
    ret
```

64-bit code for long int swap

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

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

Complete Memory Addressing Modes

Most General Form

`Imm(`r_b`,`r_i`,s)` $M[Imm+R[r_b]+R[r_i]*s]$

Imm: Immediate value 1, 2, 4 or 8 bytes

r_b: Base register ID: Any 64-bit register

r_i: Index register ID: Any 64-bit register

s: Scale: 1, 2, 4, or 8 (why these numbers?)

Special Cases

`(`r_b`,`r_i`)` $M[R[r_b]+R[r_i]]$

`Imm(`r_b`,`r_i`)` $M[Imm+R[r_b]+R[r_i]]$

`(`r_b`,`r_i`,s)` $M[R[r_b]+R[r_i]*s]$

Address Computation Instruction

- `leaq Src,Dest`
  - "Load Effective Address, Quadword"
  - Src is address mode expression
  - Set Dest to address denoted by expression

- Uses
  - Pointer computation p = &x[i]
  - Arithmetic of form x + k*y k = 1, 2, 4, or 8

Address Computation Instruction

- `leaq Src,Dest`
  - Arithmetic of form x + k*y k = 1, 2, 4, or 8

- Example

```
long mul12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax    # rax = rdi + rdi*2
salq $2, %rax               # rax = rax << 2
ret                         # return rax
```

Data Representations: x64

- Sizes of C Objects (in Bytes)

C Data Type	*Generic 32-bit*	*Intel IA32*	*x64*
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8

  - *Or any other pointer*

Instructions

- Long word **`l`** (4 Bytes) ↔ Quad word **`q`** (8 Bytes)

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: **`addl`**

is_ok example

```
long is_ok(char *line)
{
  if (line[0] == 'o' && line[1] == 'k')
    return 1;
  return 0;
}
```

```
…<is_ok>:
 0:…  mov    $0x0,%eax         # eax = 0 {rax = 0}
 5:…  cmpb   $0x6f,(%rdi)      # *rdi ? 0x6f 'o'
 8:…  jne    14 <is_ok+0x14>   # jump !=
 a:…  cmpb   $0x6b,0x1(%rdi)   # rdi[1] ? 0x6b 'k'
 e:…  sete   %al               # al = rdi[1]==0x6b
11:…  movzbl %al,%eax          # eax=(unsigned)al
14:…  repz retq                # return
```

Instructions…

• cmpb	compare byte
• jne	jump not equal
• sete	set equal
• movzbl	move zero-extend byte to longword
• ret	return
• repz	(used here as a compiler trick for the hardware)