

Machine-Level Programming I: Basics and Controls

Computer Systems
Friday, October 6 2023

Today

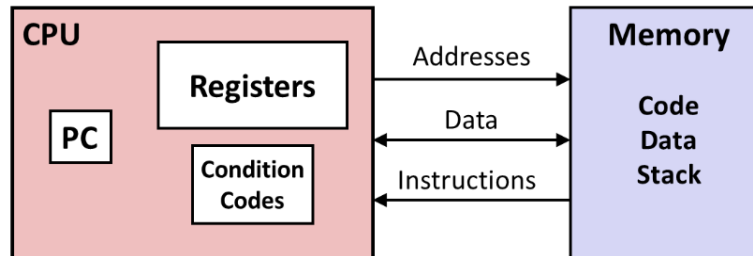
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- C, assembly, machine code
- Basics of control flow
- Condition codes
- Conditional operations
- Loops

Levels of Abstraction

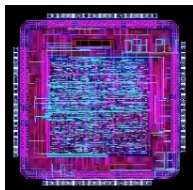
C programmer

```
#include <stdio.h>
int main(){
    int i, n = 10, t1 = 0, t2 = 1, nxt;
    for (i = 1; i <= n; ++i){
        printf("%d, ", t1);
        nxt = t1 + t2;
        t1 = t2;
        t2 = nxt; }
    return 0; }
```

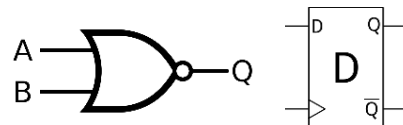
Assembly programmer



Computer Designer



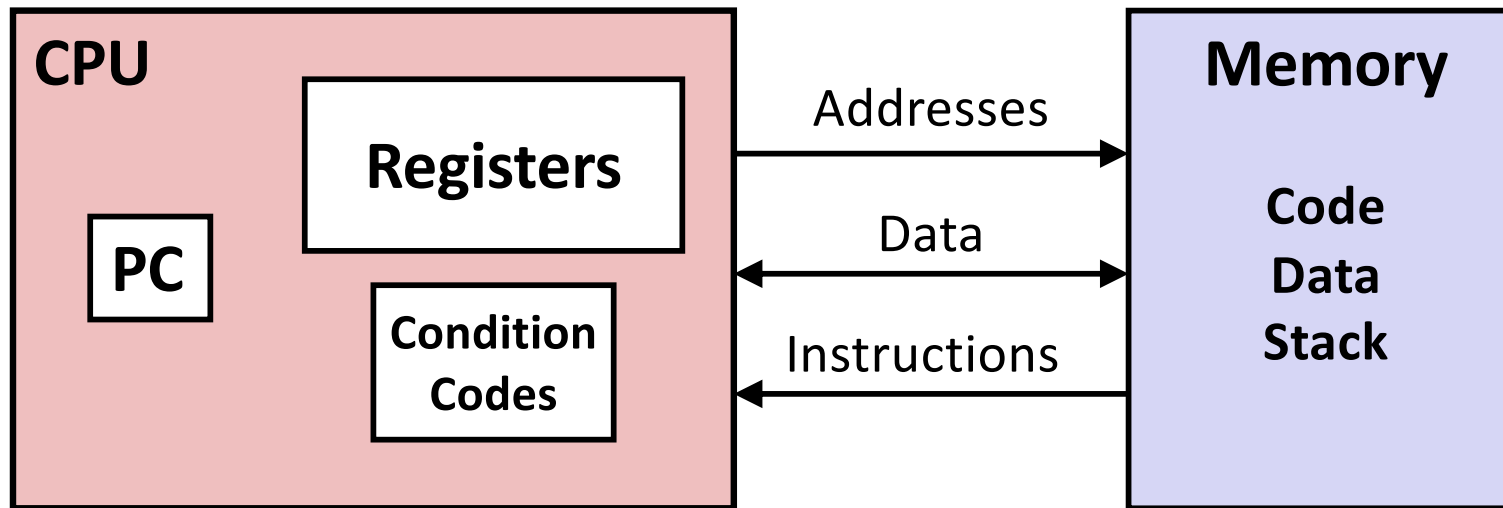
Gates, clocks, circuit layout, ...



Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing 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
 - RISC V: New open-source ISA

Assembly/Machine Code View



Programmer-Visible State

■ PC: Program counter

- Address of next instruction
- Called “RIP” (x86-64)

■ Register file

- 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

Assembly: Data Types


- **“Integer” data of 1, 2, 4, or 8 bytes**
 - Data values
 - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **(SIMD vector data types of 8, 16, 32 or 64 bytes)**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
 - Just contiguously allocated bytes in memory

Assembly: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes

- Data values
- Addresses (untyped pointers)

Register names



```
addq  %rbx, %rax
```

is

rax += rbx

These are 64-bit registers, so we know this is a 64-bit add

x86-64 Integer Registers

%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

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers

				Origin (mostly obsolete)
general purpose	%eax	%ax	%ah %al	<i>accumulate</i>
	%ecx	%cx	%ch %cl	<i>counter</i>
	%edx	%dx	%dh %dl	<i>data</i>
	%ebx	%bx	%bh %bl	<i>base</i>
	%esi	%si		<i>source index</i>
	%edi	%di		<i>destination index</i>
	%esp	%sp		<i>stack pointer</i>
	%ebp	%bp		<i>base pointer</i>
16-bit virtual registers (backwards compatibility)				

Assembly: Operations

- **Transfer data between memory and register**
 - Load data from memory into register
 - Store register data into memory
- **Perform arithmetic function on register or memory data**
- **Transfer control**
 - Unconditional jumps to/from procedures
 - Conditional branches
 - Indirect branches

Moving Data

■ Moving Data

`movq Source, Dest`

■ Operand Types

- **Immediate:** Constant integer data
 - Example: `$0x400`, `$-533`
 - Like C constant, but prefixed with ``$'`
 - Encoded with 1, 2, or 4 bytes
- **Register:** One of 16 integer registers
 - Example: `%rax`, `%r13`
 - But `%rsp` reserved for special use
 - Others have special uses for particular instructions
- **Memory** 8 consecutive bytes of memory at address given by register
 - Simplest example: `(%rax)`
 - Various other “addressing modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

**Warning: Intel docs use
`mov Dest, Source`**

movq Operand Combinations

	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	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

■ Normal (R) Mem[Reg[R]]

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

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

■ Displacement D(R) Mem[Reg[R]+D]

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

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

Complete Memory Addressing Modes

■ 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?*)

■ Special Cases

(Rb,Ri)

Mem[Reg[Rb]+Reg[Ri]]

D(Rb,Ri)

Mem[Reg[Rb]+Reg[Ri]+D]

(Rb,Ri,S)

Mem[Reg[Rb]+S*Reg[Ri]]

Example of Simple Addressing Modes

```
void  
whatAmI (<type> a, <type> b)  
{  
    ????  
}
```

%rdi

%rsi

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

Example of Simple Addressing Modes

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

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

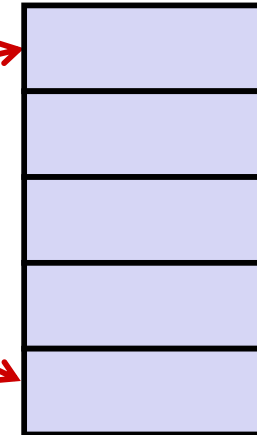

Understanding Swap()

```
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
```

Understanding Swap()

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

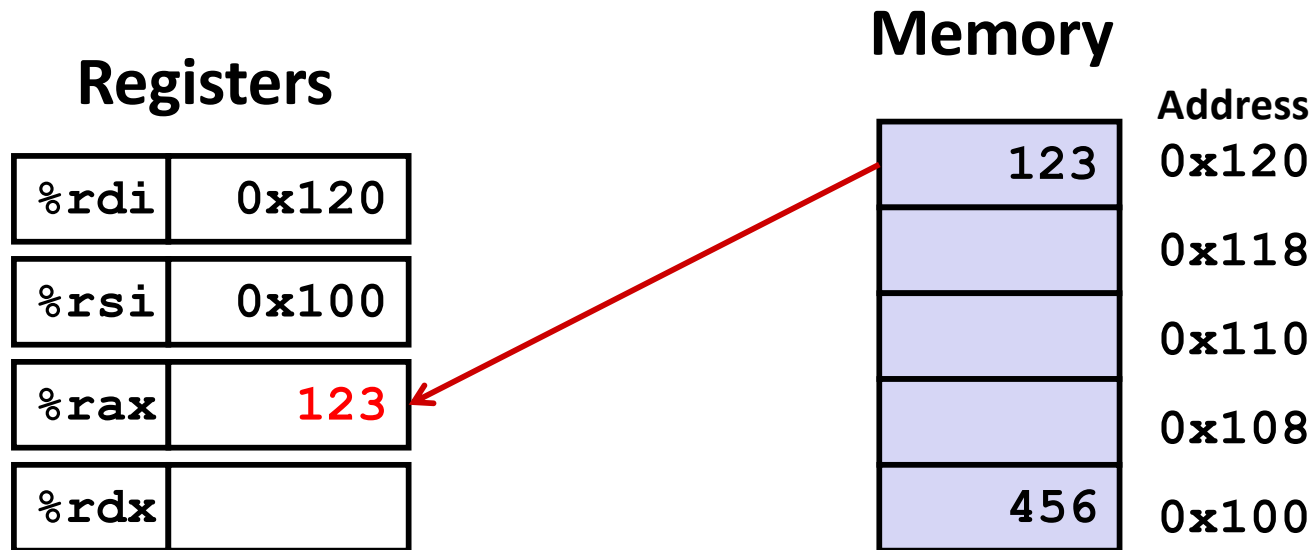
Memory

Address
123
456

swap:

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

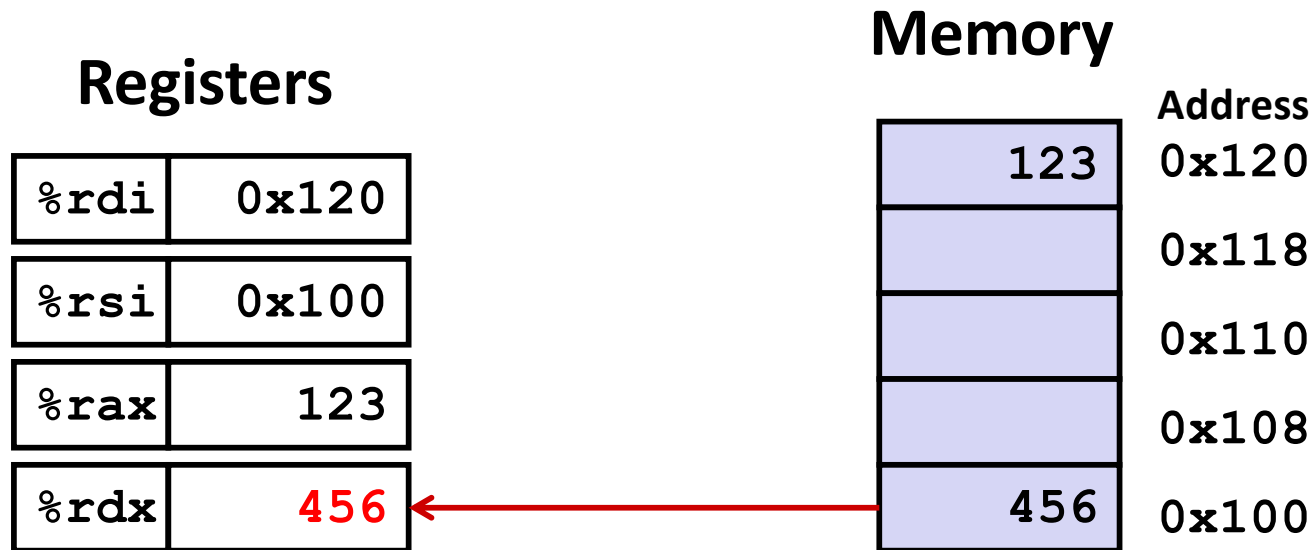
Understanding Swap()



swap:

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

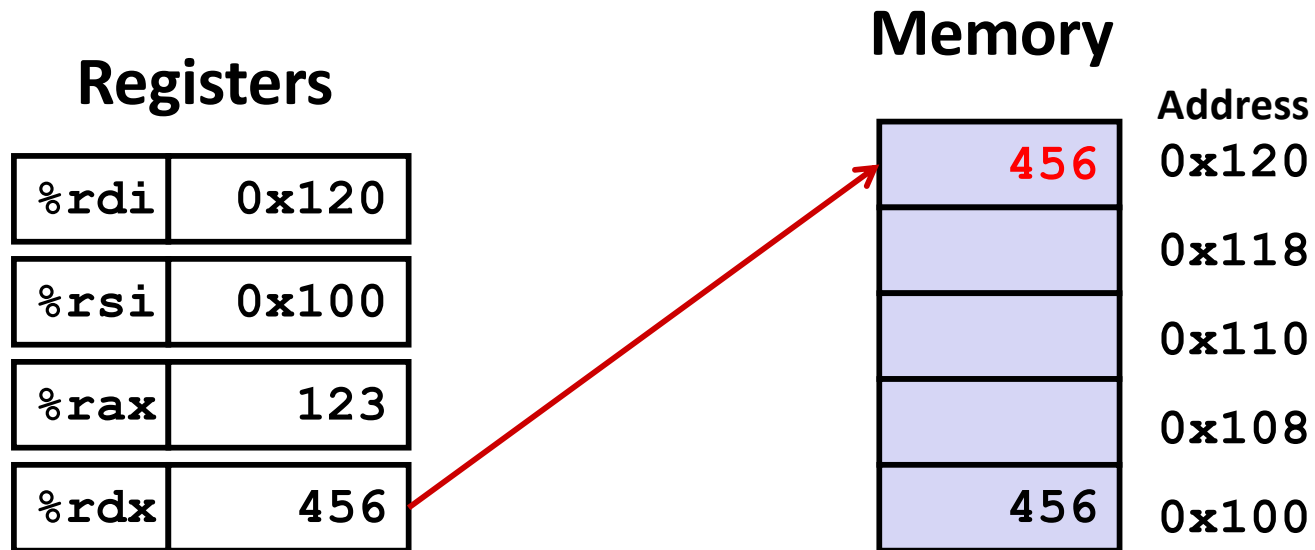
Understanding Swap()



swap:

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

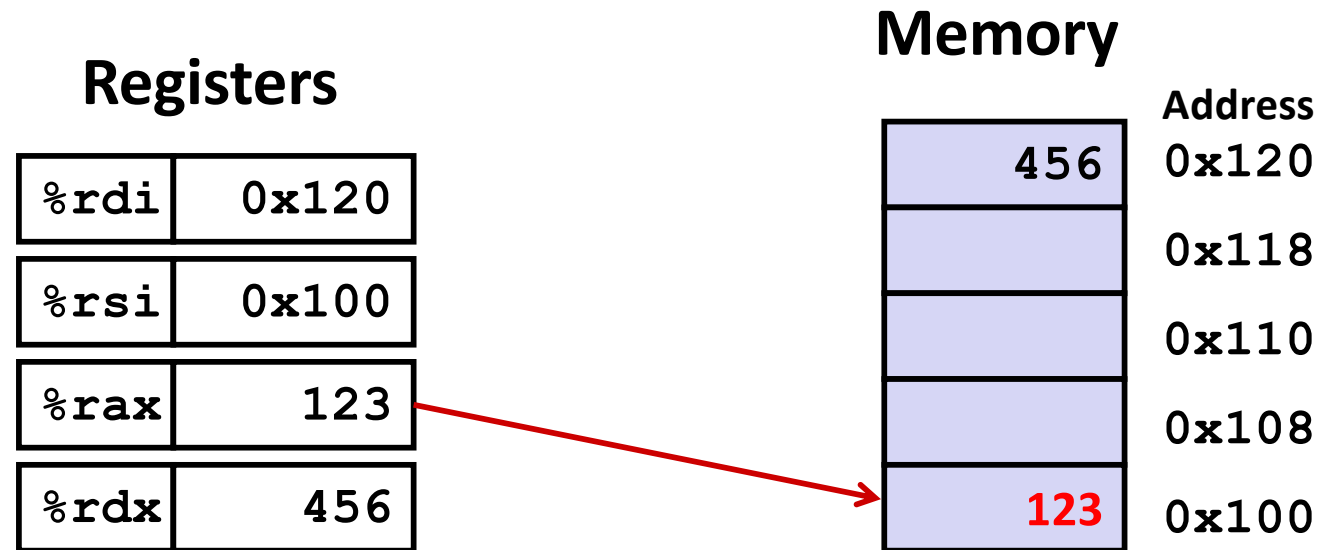
Understanding Swap()



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
```

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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?*)

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

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

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

Today

- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**
- C, assembly, machine code
- Basics of control flow
- Condition codes
- Conditional operations
- Loops

Address Computation Instruction

■ `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+2*x
salq $2, %rax           # return t<<2
```

Some Arithmetic Operations

■ Two Operand Instructions:

Format

Computation

<code>addq</code>	<code>Src, Dest</code>	<code>Dest = Dest + Src</code>
<code>subq</code>	<code>Src, Dest</code>	<code>Dest = Dest - Src</code>
<code>imulq</code>	<code>Src, Dest</code>	<code>Dest = Dest * Src</code>
<code>salq</code>	<code>Src, Dest</code>	<code>Dest = Dest << Src</code>
<code>sarq</code>	<code>Src, Dest</code>	<code>Dest = Dest >> Src</code>
<code>shrq</code>	<code>Src, Dest</code>	<code>Dest = Dest >> Src</code>
<code>xorq</code>	<code>Src, Dest</code>	<code>Dest = Dest ^ Src</code>
<code>andq</code>	<code>Src, Dest</code>	<code>Dest = Dest & Src</code>
<code>orq</code>	<code>Src, Dest</code>	<code>Dest = Dest Src</code>

Also called `shlq`
Arithmetic
Logical

- Watch out for argument order! *Src, Dest*
(Warning: Intel docs use “op *Dest, Src*”)
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

■ One Operand Instructions

<code>incq</code>	<code>Dest</code>	$\text{Dest} = \text{Dest} + 1$
<code>decq</code>	<code>Dest</code>	$\text{Dest} = \text{Dest} - 1$
<code>negq</code>	<code>Dest</code>	$\text{Dest} = -\text{Dest}$
<code>notq</code>	<code>Dest</code>	$\text{Dest} = \sim\text{Dest}$

■ 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;
}
```

arith:

```
leaq    (%rdi,%rsi), %rax    # t1
addq    %rdx, %rax          # t2
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx            # t4
leaq    4(%rdi,%rdx), %rcx   # t5
imulq   %rcx, %rax          # rval
ret
```

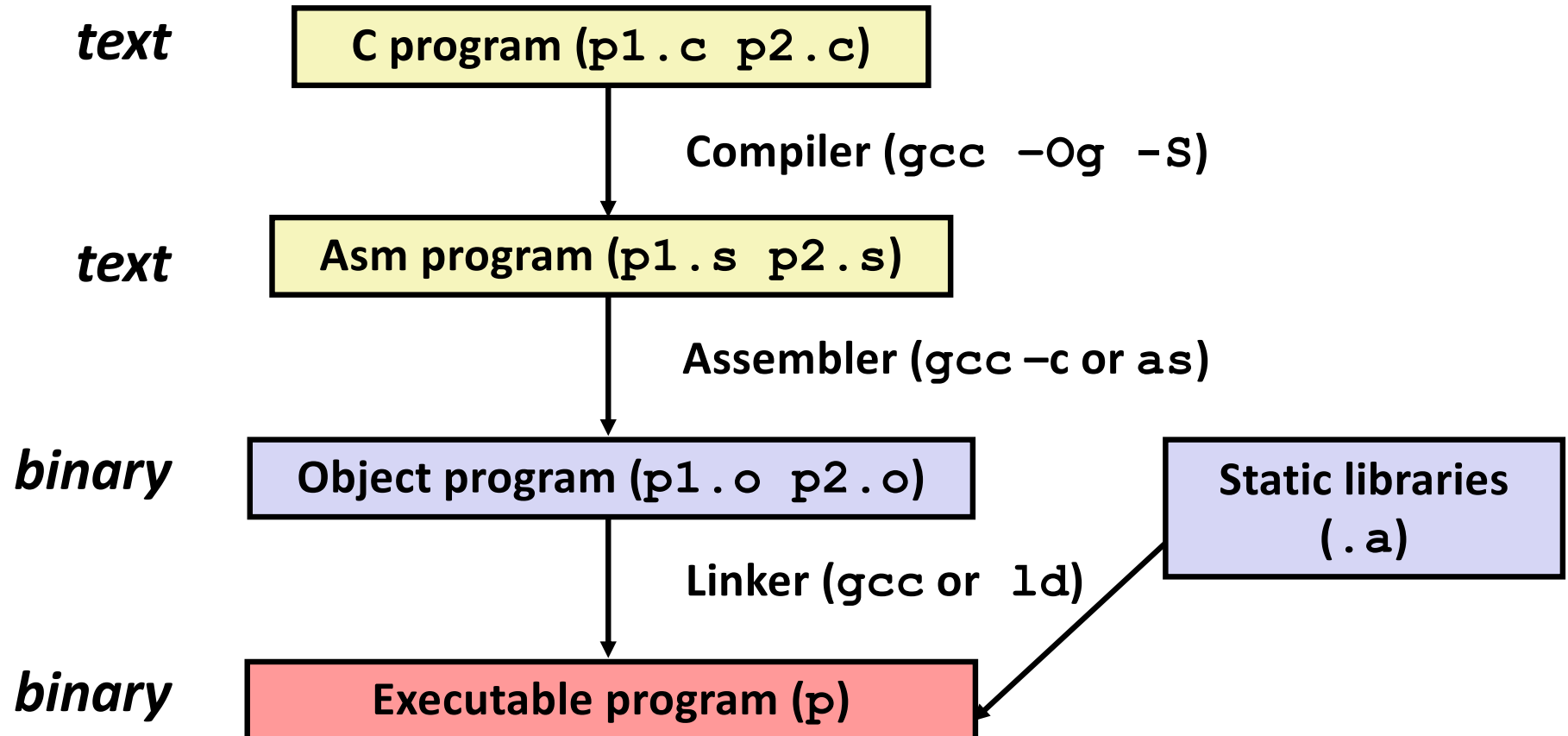
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z , t4
%rax	t1 , t2 , rval
%rcx	t5

Today

- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- **C, assembly, machine code**
- Basics of control flow
- Condition codes
- Conditional operations
- Loops

Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use debugging-friendly optimizations (`-Og`)
 - Put resulting binary in file `p`



Compiling Into Assembly

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`

Warning: Will get very different results on each machines (Ubuntu Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

What it really looks like

```
.globl sumstore
.type sumstore, @function
sumstore:
.LFB35:
.cfi_startproc
pushq %rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
movq %rdx, %rbx
call plus
movq %rax, (%rbx)
popq %rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE35:
.size sumstore, .-sumstore
```

Things that look weird
and are preceded by a '
are generally directives.

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

Object Code

Code for `sumstore`

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

- **Total of 14 bytes**

- **Each instruction
1, 3, or 5 bytes**

- **Starts at address
0x0400595**

■ Assembler

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

■ 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

Machine Instruction Example

```
*dest = t;
```

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

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

■ 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
```

■ 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

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

- Disassemble procedure

```
gdb sum
```

```
disassemble sumstore
```

Alternate Disassembly

Object Code

```
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

- Disassemble procedure

```
gdb sum
```

```
disassemble sumstore
```

- Examine the 14 bytes starting at `sumstore`

```
x/14xb sumstore
```


What Can be Disassembled?

```
% objdump -d WINWORD.EXE
```

```
WINWORD.EXE:      file format pei-i386
```

```
No symbols in "WINWORD.EXE".
```

```
Disassembly of section .text:
```

```
30001000 <.text>:
```

```
30001000:
```

```
30001001:
```

```
30001003:
```

```
30001005:
```

```
3000100a:
```

**Reverse engineering forbidden by
Microsoft End User License Agreement**

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Today

Assembly Basics: Registers, operands, move

Arithmetic & logical operations

C, assembly, machine code

Basics of control flow

Condition codes

Conditional operations

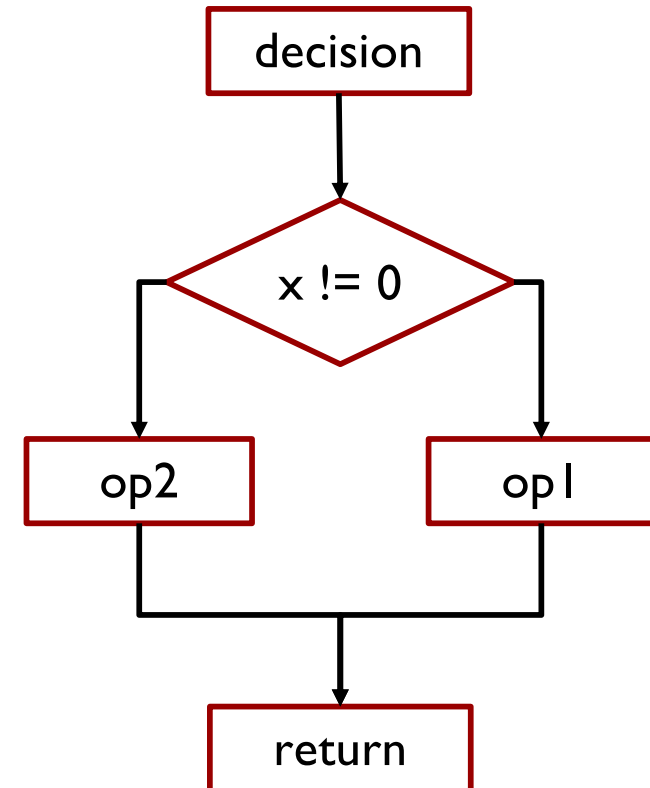
Loops

If we have time: switch statements

Control flow

```
extern void op1(void) ;  
extern void op2(void) ;
```

```
void decision(int x) {  
    if (x) {  
        op1() ;  
    } else {  
        op2() ;  
    }  
}
```



Control flow in assembly language

```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```

```
decision:
    subq    $8, %rsp
    testl   %edi, %edi
    je      .L2
    call    op1
    jmp     .L1
.L2:
    call    op2
.L1:
    addq    $8, %rsp
    ret
```



It's all done with
GOTO!

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`)

Current stack top

Registers

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

`%rip` Instruction pointer

<code>CF</code>	<code>ZF</code>	<code>SF</code>	<code>OF</code>
-----------------	-----------------	-----------------	-----------------

Condition codes

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)
- **GDB prints these as one “eflags” register**
 eflags **0x246** [**PF** **ZF** **IF**] *Z set, CSO clear*

Implicitly set (as side effect) of arithmetic operations

Example: `addq Src, Dest` \leftrightarrow `t = a+b`

CF set if carry out from most significant bit (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

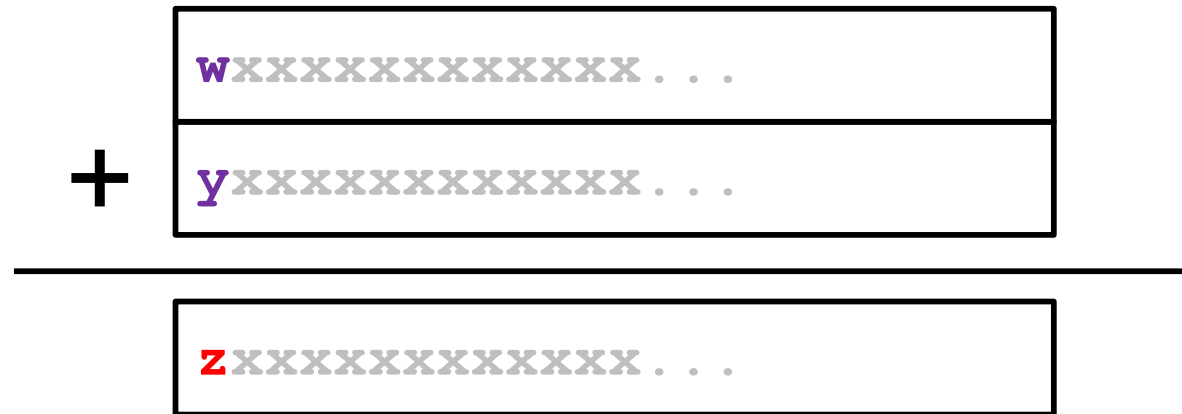
ZF set when

00000000000000...000000000000

CF set when

$$\begin{array}{r} \boxed{\text{yxxxxxxxxxxxxx} \dots} \\ + \boxed{\text{yxxxxxxxxxxxxx} \dots} \\ \hline \text{1} \boxed{\text{zxxxxxxxxxxxxx} \dots} \end{array}$$

OF set when



`w == y && w != z`

Compare Instruction

cmp a, b

- Computes $b - a$ (just like **sub**)
- Sets condition codes based on result, but...
- **Does not change b**
- Used for **if (a < b) { ... }**
whenever $b - a$ isn't needed for anything else

Test Instruction

test a, b

- Computes $b \& a$ (just like **and**)
- Sets condition codes (only SF and ZF) based on result, but...
- **Does not change b**
- Most common use: `test %rX, %rX`
to compare `%rX` to zero
- Second most common use: `test %rX, %rY`
tests if any of the 1-bits in `%rY` are also 1 in `%rX` (or vice versa)

Today

Assembly Basics: Registers, operands, move

Arithmetic & logical operations

C, assembly, machine code

Basics of control flow

Condition codes

Conditional operations

Loops

Jumping

jX Instructions

- Jump to different part of code depending on condition codes

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

Reading Condition Codes

SetX Instructions

- Set low-order byte of destination to 0 or 1 based on *combinations* of condition codes
- Does not alter remaining 7 bytes

SetX	Condition	Description
<code>sete</code>	ZF	Equal / Zero
<code>setne</code>	$\sim ZF$	Not Equal / Not Zero
<code>sets</code>	SF	Negative
<code>setns</code>	$\sim SF$	Nonnegative
<code>setg</code>	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
<code>setge</code>	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
<code>setl</code>	$(SF \wedge OF)$	Less (Signed)
<code>setle</code>	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
<code>seta</code>	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
<code>setb</code>	CF	Below (unsigned)

x86-64 Integer Registers

%rax	%al
%rbx	%bl
%rcx	%cl
%rdx	%dl
%rsi	%sil
%rdi	%dil
%rsp	%spl
%rbp	%bpl

%r8	%r8b
%r9	%r9b
%r10	%r10b
%r11	%r11b
%r12	%r12b
%r13	%r13b
%r14	%r14b
%r15	%r15b

- SetX argument is always a low byte (%al, %r8b, etc.)

Reading Condition Codes (Cont.)

SetX Instructions:

- Set single byte based on combination of condition codes

One of addressable byte registers

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
 - 32-bit instructions also set upper 32 bits to 0

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

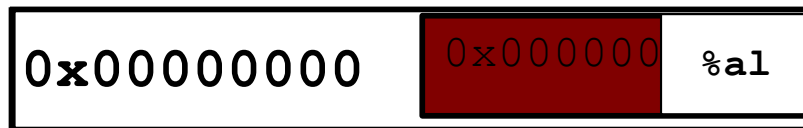
```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al          # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

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

Reading Condition Codes (Cont.)

Beware weirdness `movzbl` (and others)

`movzbl %al, %eax`



Zapped to all
0's

Use(s)

Argument **x**

Argument **y**

Return value

```

cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
  
```

Conditional Branch Example (Old Style)

Generation

```
linux> gcc -Og -S -fno-if-conversion cont
```

I'll get to this shortly.

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

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    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

Expressing with Goto Code

C allows goto statement

Jump to position designated by label

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

```
long absdiff_j
(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x > y ? x - y : y - x;
```

Goto Version

```
    ntest = !Test;  
    if (ntest) goto  
Else;  
    val = Then_Expr;  
    goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

Conditional Move Instructions

- Instruction supports:
if (Test) Dest \leftarrow Src
- Supported in post-1995 x86 processors
- GCC tries to use them
 - But, only when known to be safe

Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

C Code

```
val = Test  
    ? Then_Expr  
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

Conditional Move Example

```

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

```

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

absdiff:

```

movq    %rdi, %rax    # x
subq    %rsi, %rax    # result = x-y
movq    %rsi, %rdx
subq    %rdi, %rdx    # eval = y-x
cmpq    %rsi, %rdi    # x:y
cmovle  %rdx, %rax    # if <=, result = eval
ret

```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

Both values get computed

Only makes sense when computations
are very simple

Bad Performance

Risky Computations

```
val = p ? *p : 0;
```

Both values get computed

May have undesirable effects

Unsafe

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

Both values get computed

Must be side-effect free

Illegal

Today

Assembly Basics: Registers, operands, move

Arithmetic & logical operations

C, assembly, machine code

Basics of control flow

Condition codes

Conditional operations

Loops

“Do-While” Loop Example

C Code

```
long pcount_do
(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    return result;
}
```

Count number of 1's in argument *x* (“popcount”)

Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```

movl    $0, %eax           # result = 0
.L2:
                                # loop:
    movq    %rdi, %rdx
    andl    $1, %edx        # t = x & 0x1
    addq    %rdx, %rax      # result += t
    shrq    %rdi            # x >>= 1
    jne     .L2             # if (x) goto
loop
    rep; ret
```

General “Do-While” Translation

C Code

```
do  
    Body  
while (Test) ;
```

```
Body: {  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```

Goto Version

```
loop:  
    Body  
    if (Test)  
        goto loop
```

General “While” Translation #1

“Jump-to-middle” translation

Used with -Og

While version

```
while (Test)  
    Body
```



Goto Version

```
    goto test;  
loop:  
    Body  
test:  
    if (Test)  
        goto loop;  
done:
```

While Loop Example #1

C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
(unsigned long x) {
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if(x) goto loop;
    return result;
}
```

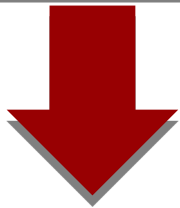
Compare to do-while version of function

Initial goto starts loop at test

General “While” Translation #2

While version

```
while (Test)  
    Body
```



Do-While Version

```
if (!Test)  
    goto done;  
do  
    Body  
    while (Test);  
done:
```

“Do-while” conversion
Used with -O1

Goto Version

```
if (!Test)  
    goto done;  
loop:  
    Body  
    if (Test)  
        goto loop;  
done:
```



While Loop Example #2

C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Do-While Version

```
long pcount_goto_dw
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

Compare to do-while version of function

Initial conditional guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update )
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
    unsigned bit =
        (x >> i) & 0x1;
    result += bit;
}
```

“For” Loop → While Loop

For Version

```
for (Init; Test; Update )  
    Body
```



While Version

```
Init ;  
while (Test ) {  
    Body  
    Update ;  
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

“For” Loop Do-While Conversion

C Code Goto Version

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Initial test can be optimized
away

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!(i < WSIZE)) Ini
    goto done; !Test
loop:
{
    unsigned bit =
        (x >> i) & 0x1; Body
    result += bit;
}
i++; Update
if (i < WSIZE) Test
    goto loop;
done:
    return result;
}
```

Machine Programming I: Summary

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

Summary: Machine Instructions

```
*dest = t;
```

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

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

C

- 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]`

Machine

- 3 bytes at address `0x40059e`
- Compact representation of the assembly instruction
- (Relatively) easy for hardware to interpret

Summary: Machine Instructions

```
*dest = t;
```

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

0x40059e: 48 89 03

0100 1 0 0 0 10001011 00 000 011
 REX W R X B MOV r->x Mod R M

C

- 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]`

Machine

- 3 bytes at address `0x40059e`
- Compact representation of the assembly instruction
- (Relatively) easy for hardware to interpret

Summary: Address Modes

Most General Form

$D(Rb, Ri, S) \quad Mem[Reg[Rb] + S * Reg[Ri] + D]$

- 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?*)

Special Cases

$(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri]]$

$D(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri] + D]$

$(Rb, Ri, S) \quad Mem[Reg[Rb] + S * Reg[Ri]]$

Memory operands and LEA

In most instructions, a memory operand accesses memory

Assembly	C equivalent
<code>mov 6(%rbx,%rdi,8), %ax</code>	<code>ax = *(rbx + rdi*8 + 6)</code>
<code>add 6(%rbx,%rdi,8), %ax</code>	<code>ax += *(rbx + rdi*8 + 6)</code>
<code>xor %ax, 6(%rbx,%rdi,8)</code>	<code>*(rbx + rdi*8 + 6) ^= ax</code>

LEA is special: it *doesn't* access memory

Assembly	C equivalent
<code>lea 6(%rbx,%rdi,8), %rax</code>	<code>rax = rbx + rdi*8 + 6</code>

Why use LEA?

CPU designers' intended use: calculate a pointer to an object

- An array element, perhaps
- For instance, to pass just one array element to another function

Assembly	C equivalent
<code>lea (%rbx,%rdi,8), %rax</code>	<code>rax = &rbx[rdi]</code>

Compiler authors like to use it for ordinary arithmetic

- It can do complex calculations in one instruction
- It's one of the only three-operand instructions the x86 has
- It doesn't touch the condition codes (we'll come back to this)

Assembly	C equivalent
<code>lea (%rbx,%rbx,2), %rax</code>	<code>rax = rbx * 3</code>

Which numbers are pointers?

They aren't labeled
You have to figure it
out from context

```
(gdb) info registers
rax      0x40057d      4195709
rbx      0x0           0
rcx      0x4005e0      4195808
rdx      0x7fffffffdc28 140737488346152
rsi      0x7fffffffdc18 140737488346136
rdi      0x1           1
rbp      0x0           0x0
rsp      0x7fffffffdb38 0x7fffffffdb38
r8       0x7ffff7dd5e80 140737351868032
r9       0x0           0
r10      0x7fffffff7c0 140737488345024
r11      0x7ffff7a2f460 140737348039776
r12      0x400490      4195472
r13      0x7fffffffdc10 140737488346128
r14      0x0           0
r15      0x0           0
rip      0x40057d      0x40057d
```

Which numbers are pointers?

They aren't labeled

You have to figure it out from context

%rsp and **%rip** always hold pointers

```
(gdb) info registers
rax      0x40057d      4195709
rbx      0x0          0
rcx      0x4005e0      4195808
rdx      0x7fffffffdc28 140737488346152
rsi      0x7fffffffdc18 140737488346136
rdi      0x1          1
rbp      0x0          0x0
rsp     0x7fffffffdb38 0x7fffffffdb38
r8       0x7ffff7dd5e80 140737351868032
r9       0x0          0
r10      0x7fffffffcd7c0 140737488345024
r11      0x7ffff7a2f460 140737348039776
r12      0x400490      4195472
r13      0x7fffffffcd10 140737488346128
r14      0x0          0
r15      0x0          0
rip     0x40057d      0x40057d
```

Which numbers are pointers?

They aren't labeled

You have to figure it out from context

%rsp and **%rip** always hold pointers

- Register values that are “close” to %rsp or %rip are *probably* also pointers

(gdb) info registers

rax	0x40057d	4195709
rbx	0x0	0
rcx	0x4005e0	4195808
rdx	0x7fffffffdc28	140737488346152
rsi	0x7fffffffdc18	140737488346136
rdi	0x1	1
rbp	0x0	0x0
rsp	0x7fffffffdb38	0x7fffffffdb38
r8	0x7ffff7dd5e80	140737351868032
r9	0x0	0
r10	0x7fffffff7c0	140737488345024
r11	0x7ffff7a2f460	140737348039776
r12	0x400490	4195472
r13	0x7fffffffdc10	140737488346128
r14	0x0	0
r15	0x0	0
rip	0x40057d	0x40057d

Which numbers are pointers?

If a register is being
used as a pointer...

Dump of assembler code for function main:

```
=> 0x40057d <+0>:  sub    $0x8,%rsp
      0x400581 <+4>:  mov     (%rsi),%rsi
      0x400584 <+7>:  mov     $0x400670,%edi
      0x400589 <+12>: mov     $0x0,%eax
      0x40058e <+17>: call    0x400460
```

Which numbers are pointers?

If a register is being *used*
as a pointer...

- `mov (%rsi), %rsi`
- ...Then its value is *expected*
to be a pointer.
 - There might be a bug that makes its value incorrect.

Dump of assembler code for function main:

```
=> 0x40057d <+0>:  sub    $0x8,%rsp
      0x400581 <+4>:  mov     (%rsi),%rsi
      0x400584 <+7>:  mov     $0x400670,%edi
      0x400589 <+12>: mov     $0x0,%eax
      0x40058e <+17>: call    0x400460
```

Which numbers are pointers?

If a register is being *used* as a pointer...

- `mov (%rsi), %rsi`
- ...Then its value is *expected* to be a pointer.

- There might be a bug that makes its value incorrect.

Dump of assembler code for function main:

```
=> 0x40057d <+0>:  sub    $0x8,%rsp
      0x400581 <+4>:  mov     (%rsi),%rsi
      0x400584 <+7>:  mov     $0x400670,%edi
      0x400589 <+12>: mov     $0x0,%eax
      0x40058e <+17>: call    0x400460
```

Not as obvious with complicated address “modes”

- `(%rsi, %rbx)` – *One* of these is a pointer, we don’t know which.
- `(%rsi, %rbx, 2)` – `%rsi` is a pointer, `%rbx` isn’t (why?)
- `0x400570(, %rbx, 2)` – `0x400570` is a pointer, `%rbx` isn’t (why?)
- `lea (anything), %rax` – (anything) *may or may not* be a pointer

Summary: Condition Codes

Single bit registers

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

jX and SetX instructions

jX	Condition	Description
jmp	1	Unconditional
jbe	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	$\sim ZF$	Not Equal / Not Zero
sets	SF	Negative
setns	$\sim SF$	Nonnegative
setg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
setge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
setl	$(SF \wedge OF)$	Less (Signed)
setle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
seta	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
setb	CF	Below (unsigned)

Machine Level Programming – Control

C Control

- if-then-else
- do-while
- while, for
- switch

Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

Standard Techniques

- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (if-elseif-elseif-else)

Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

- Procedure arguments
- Return value

Memory management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

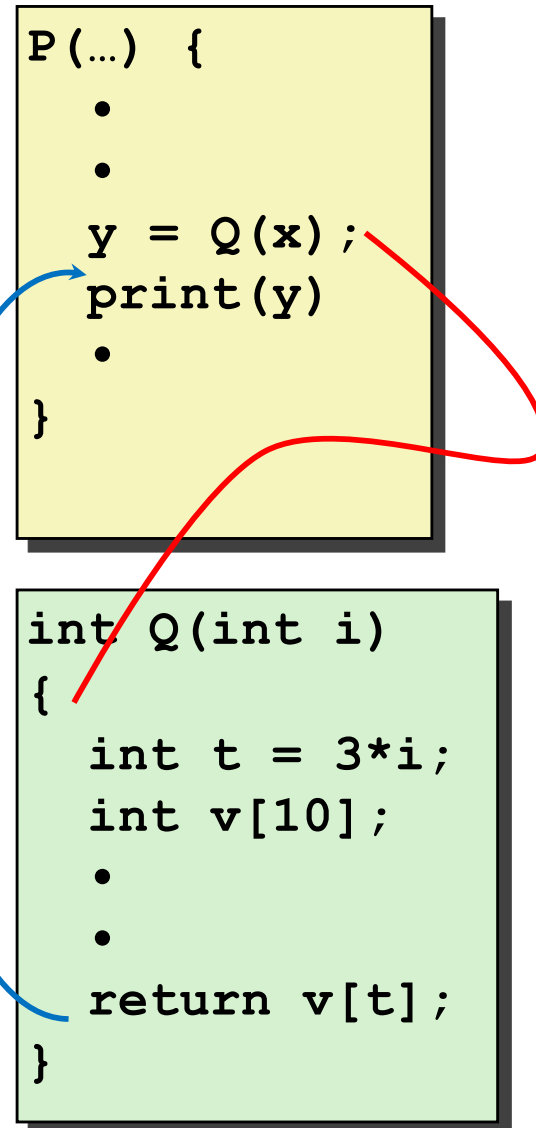
- Procedure arguments
- Return value

Memory management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required



Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

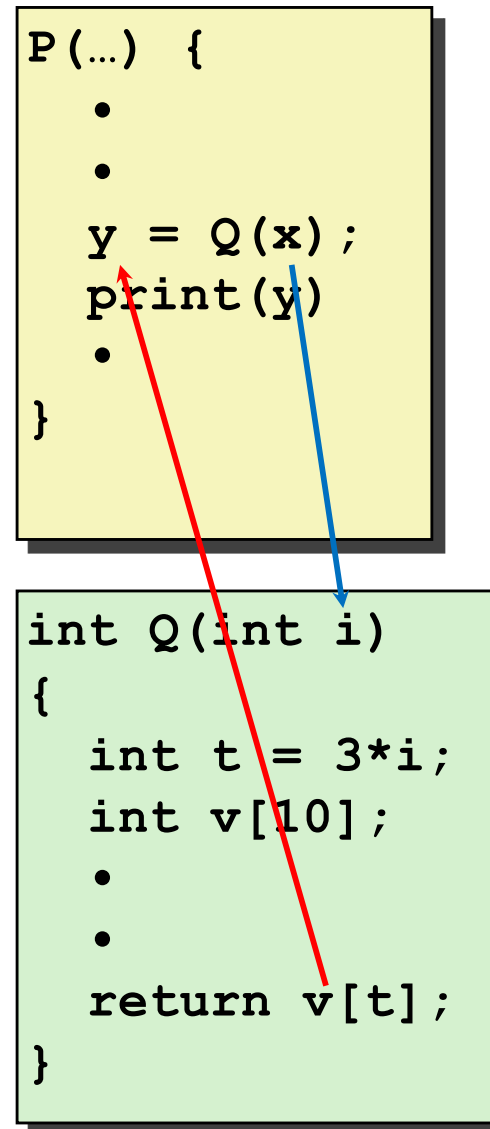
- Procedure arguments
- Return value

Memory management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required



Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

- Procedure arguments
- Return value

Memory management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```