

Machine-Level Programming : Basics

Computer Systems

Friday, September 27 2024

Today

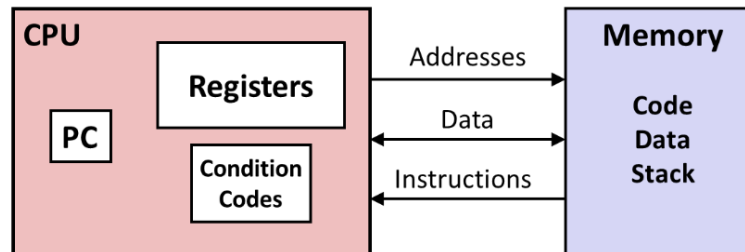
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- C, assembly, machine code

Levels of Abstraction

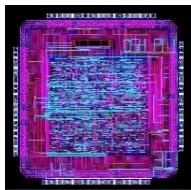
C programmer

```
#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; }
```

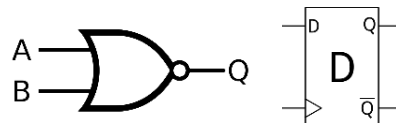
Assembly programmer



Computer Designer



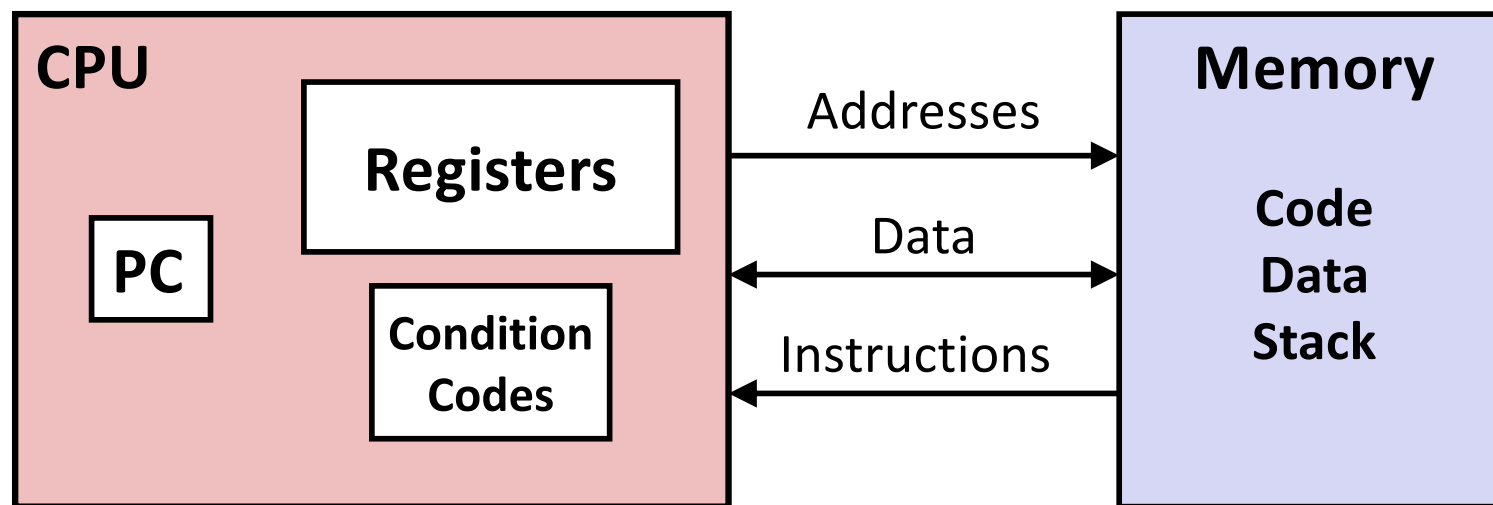
Gates, clocks, circuit layout, ...



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
```

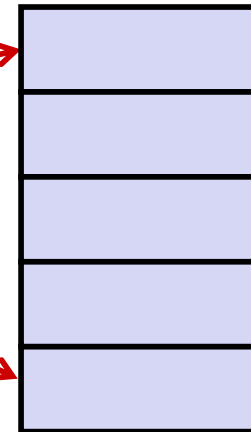

Understanding Swap()

```
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

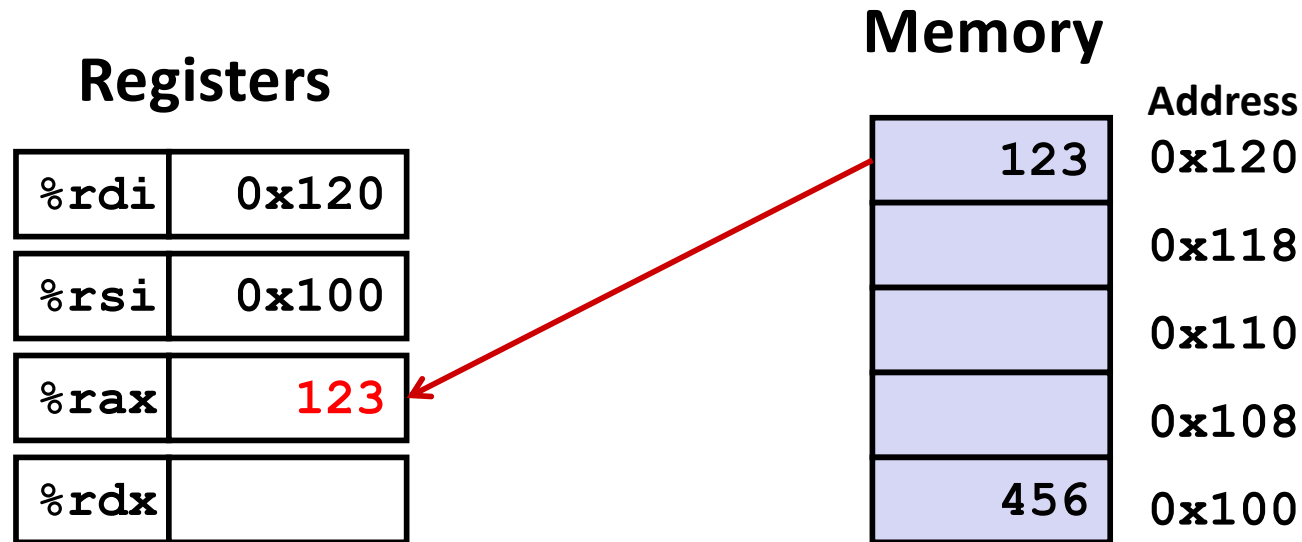
Address	
<code>0x120</code>	123
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	456

swap:

```

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

Understanding Swap()



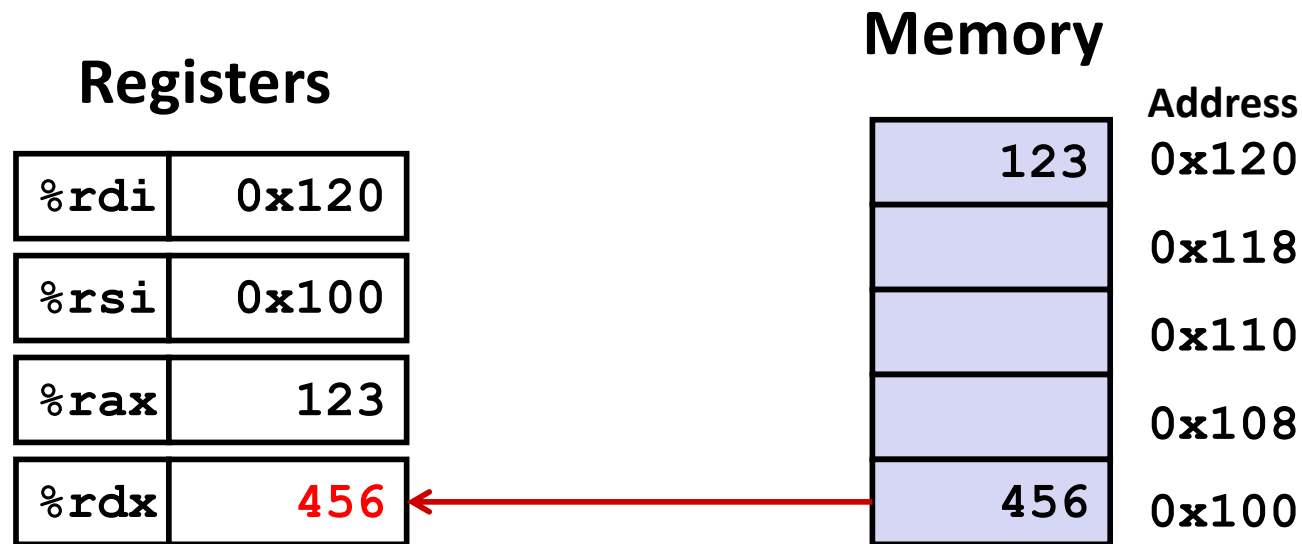
swap:

```

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

```

Understanding Swap()



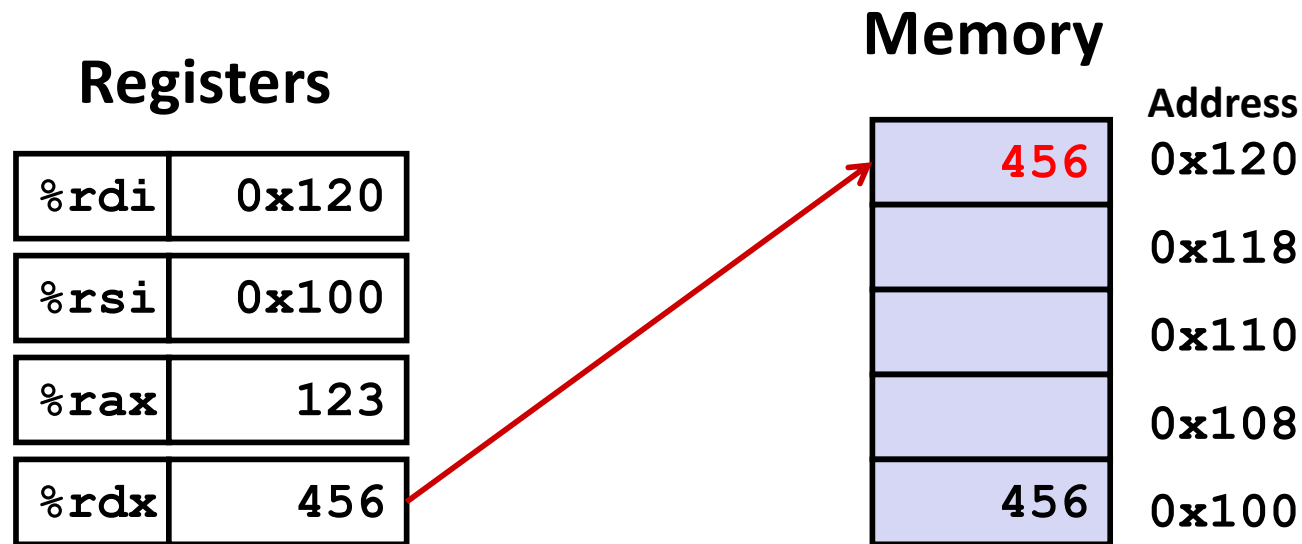
swap:

```

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

```

Understanding Swap()



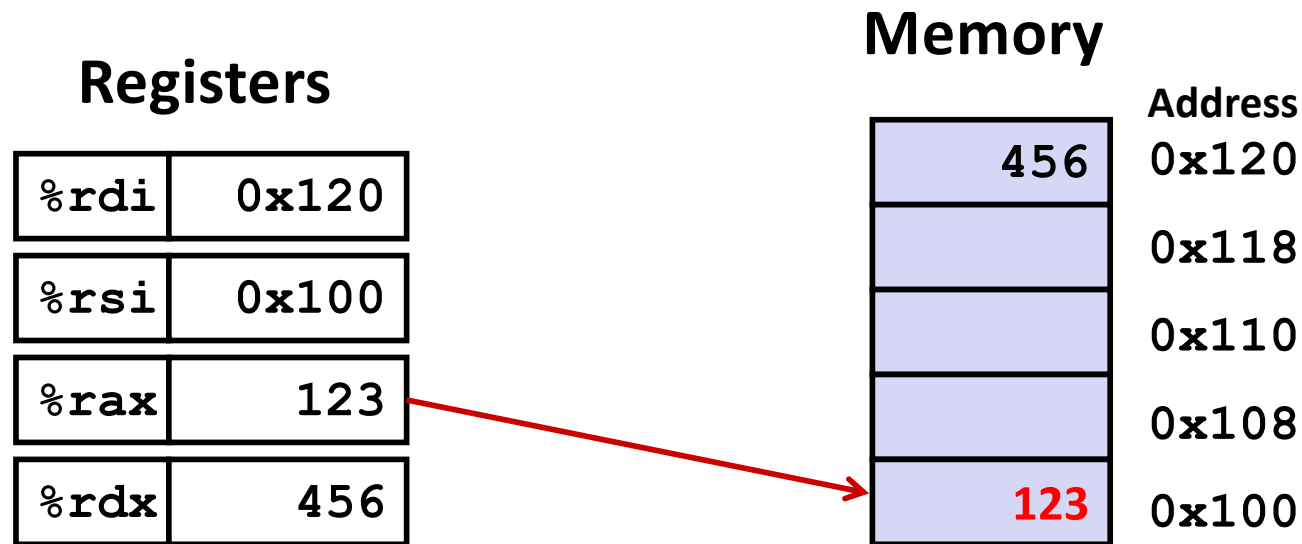
swap:

```

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

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>	$\text{Dest} = \text{Dest} + \text{Src}$	
<code>subq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} - \text{Src}$	
<code>imulq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} * \text{Src}$	
<code>salq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \ll \text{Src}$	Also called <code>shlq</code>
<code>sarq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \gg \text{Src}$	Arithmetic
<code>shrq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \gg \text{Src}$	Logical
<code>xorq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \wedge \text{Src}$	
<code>andq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \& \text{Src}$	
<code>orq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \text{Src}$	

- 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

```

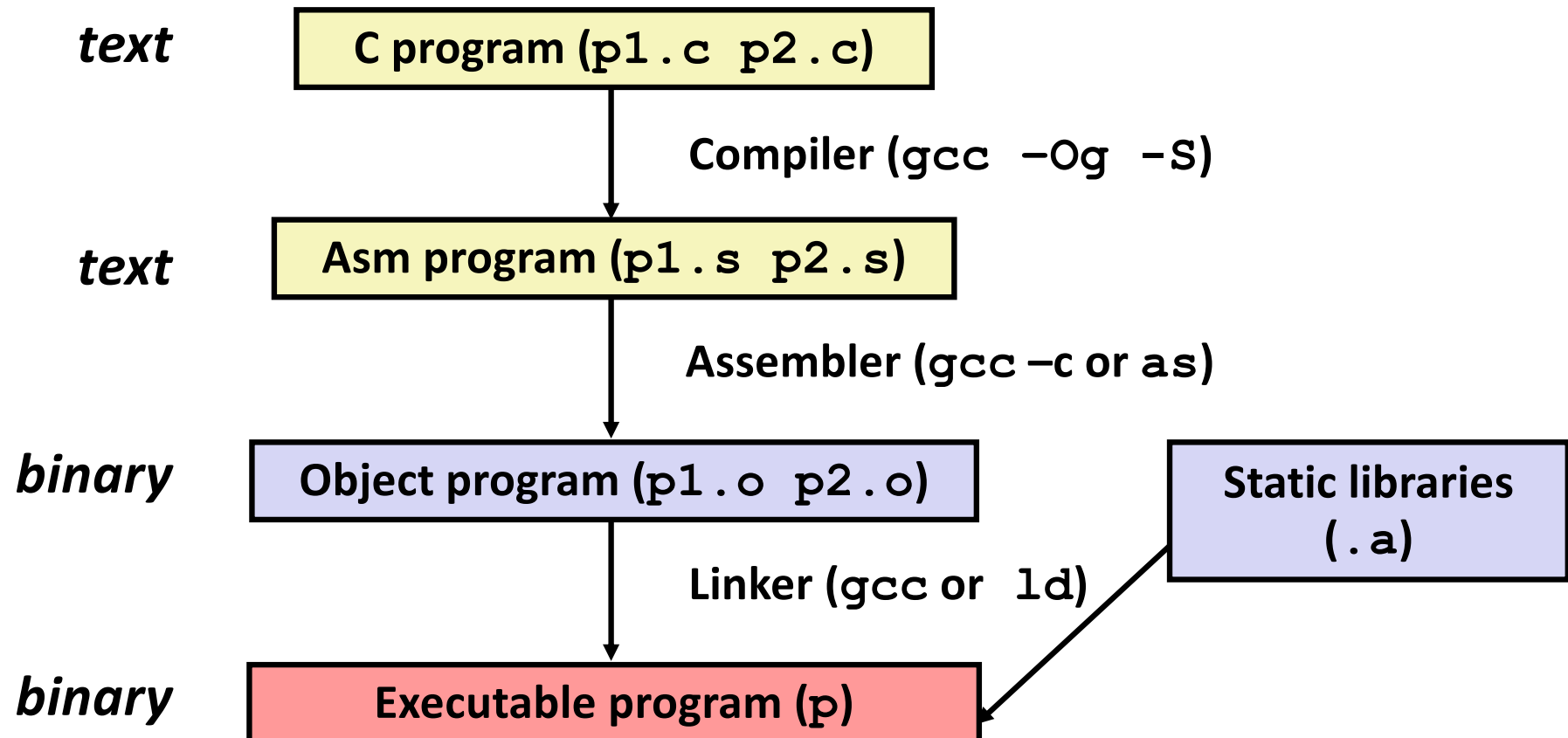
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z , t4
%rax	t1 , t2 , rval
%rcx	t5

Today

- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- **C, assembly, machine code**

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

C++ source #1

Save/Load

+ Add new...

Vim

Cpplnsights

Quick-bench

C++

--INSERT--

```

1 // Type your code here, or load an example.
2 #include <stdio.h>
3 int square(int num) {
4     printf("num=%d\n", num);
5     return num * num;
6 }
7
8 int caller(int num) {
9     int result = square(3);
10    printf("result = %d\n", result);
11    return 0;
12 }
13

```

x86-64 gcc 14.2 (Editor #1)

x86-64 gcc 14.2

Compiler options...

Output...

Filter...

Libraries

Overrides

+ Add new...

Add tool...

```

11     mov     eax, 0
12     call    printf
13     mov     eax, DWORD PTR [rbp-4]
14     imul    eax, eax
15     leave
16     ret
17
18 .LC1:
19     .string "result = %d\n"
20 caller(int):
21     push    rbp
22     mov     rbp, rsp
23     sub     rsp, 32
24     mov     DWORD PTR [rbp-20], edi
25     mov     edi, 3
26     call    square(int)
27     mov     DWORD PTR [rbp-4], eax
28     mov     eax, DWORD PTR [rbp-4]
29     mov     esi, eax
30     mov     edi, OFFSET FLAT:.LC1
31     mov     eax, 0
32     call    printf
33     mov     eax, 0
34     leave
35     ret

```