Systems Programming

Machine-Level Programming I: Basics

Textbook coverage:
Ch 3: Machine-Level Representation of Programs

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.

Today: Machine Programming I: Basics

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

Intel x86 Processors

- **Dominate laptop/desktop/server market**

- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
  - 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.

Our Coverage

- **IA32**
  - The traditional x86
- **x86-64**
  - The standard
  - `$ gcc hello.c`
  - `$ gcc –m64 hello.c`
- **Presentation**
  - We will only cover x86-64
  - https://en.wikibooks.org/wiki/X86_Assembly/GNU_assembly_syntax

Today: Machine Programming I: Basics

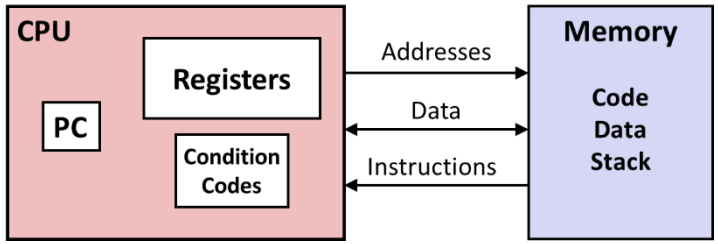
- Intel processors and architectures
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- C, assembly, machine code

Levels of Abstraction

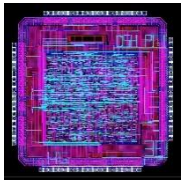
C programmer

```
#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; }
```

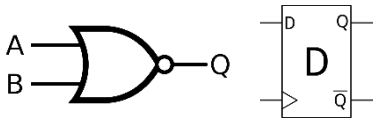
Assembly programmer



Computer Designer



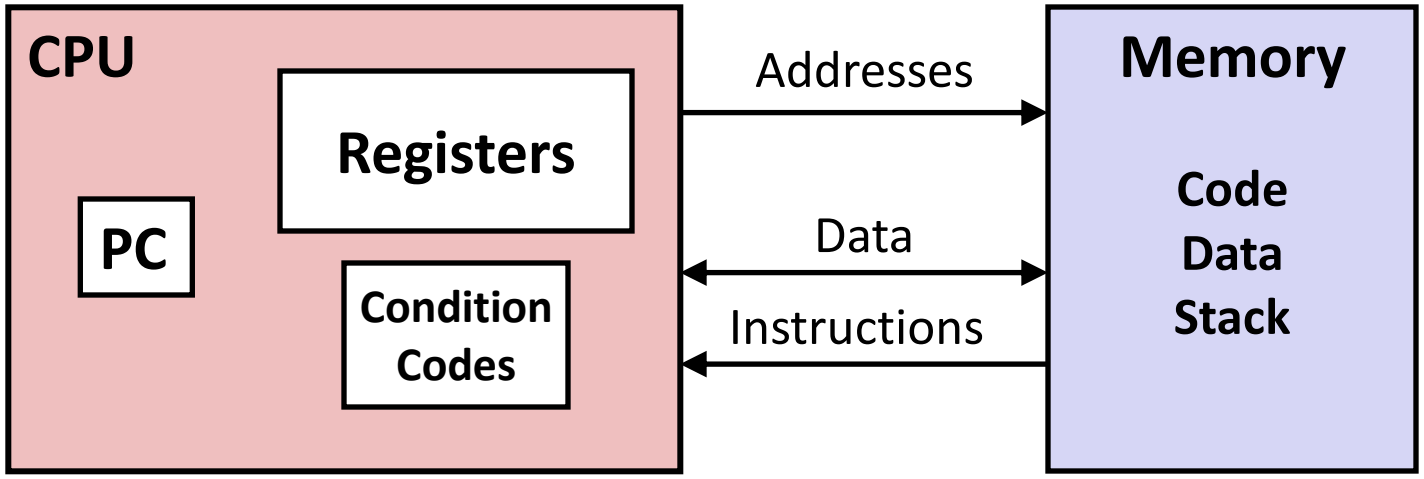
Gates, clocks, circuit layout, …



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, core frequency, pipelines
- **Code Forms:**
  - Assembly Code: A text representation of machine code
  - Machine Code: The byte-level programs that a processor executes
- **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

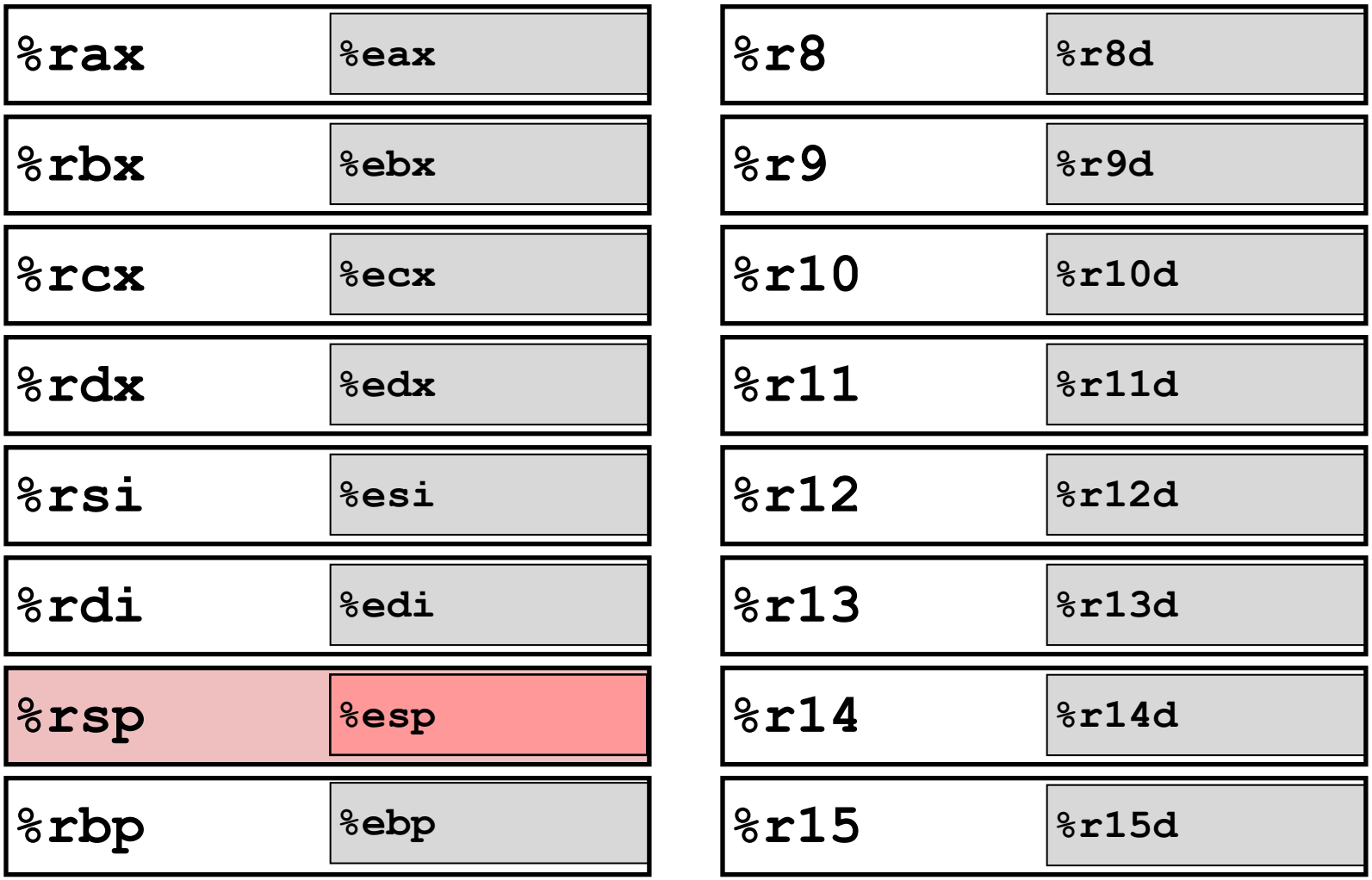


Programmer-Visible State

- **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 data
  - Stack to support procedures

x86-64 Integer Registers



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

NOTE: Memory Hierarchy

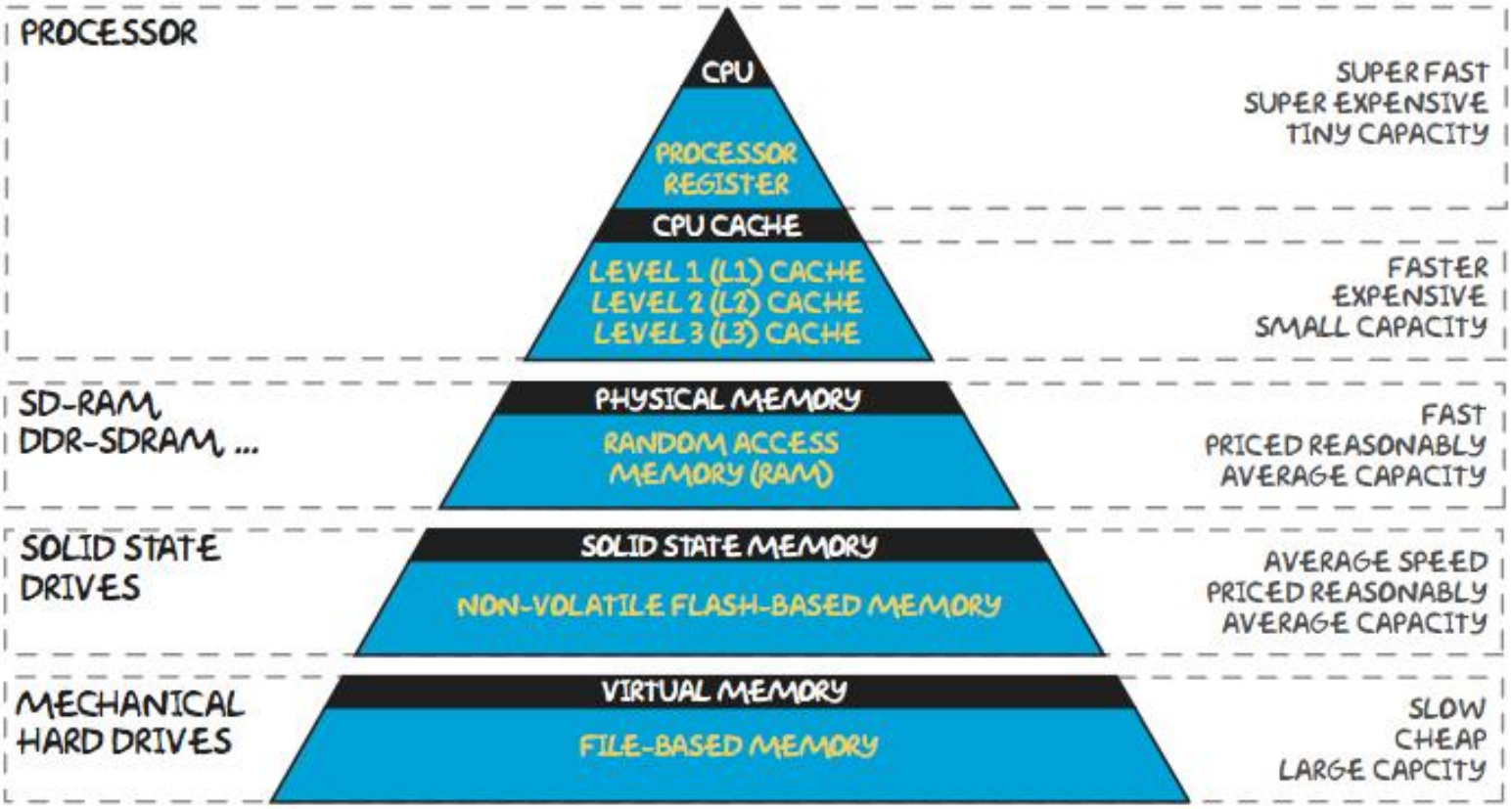


Image via teachbook.com.au

Assembly: Operations

- **#1. Move data between memory and register**
  - Load: Transfer data from memory to register
  - Store: Transfer data from register to memory
- **#2. Perform arithmetic function on register or memory data**
- **#3. Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

Assembly: Basics

- **Instruction format**
  - [Operation] [Source Operand], [Destination Operand]
    example: movb $0x05, %al

- **Operation Suffixes**
  - `b` = byte (8 bit).
  - `s` = single (32-bit floating point).
  - `w` = word (16 bit).
  - `l` = long (32 bit integer or 64-bit floating point).
  - `q` = quad (64 bit).
  - `t` = ten bytes (80-bit floating point).

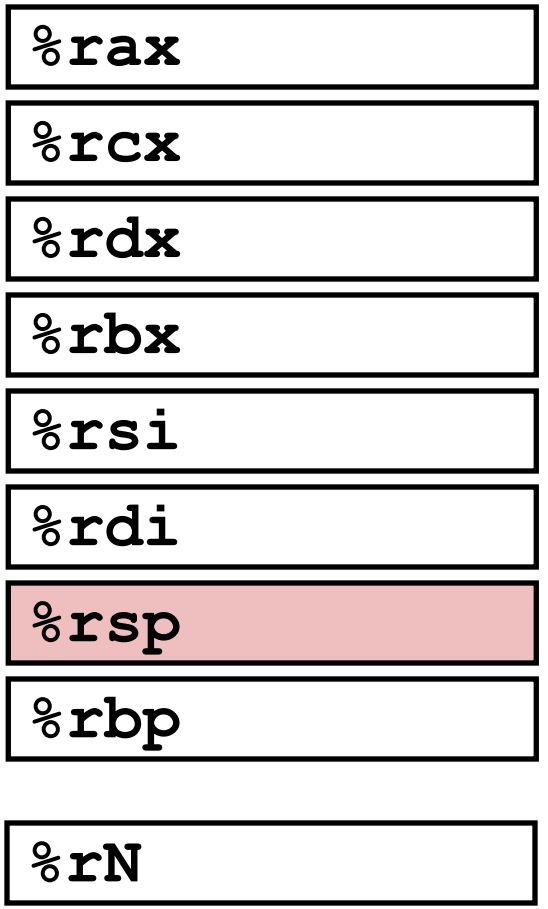
- **Operand Prefixes**
  - $: constant numbers
  - %: register

Moving Data

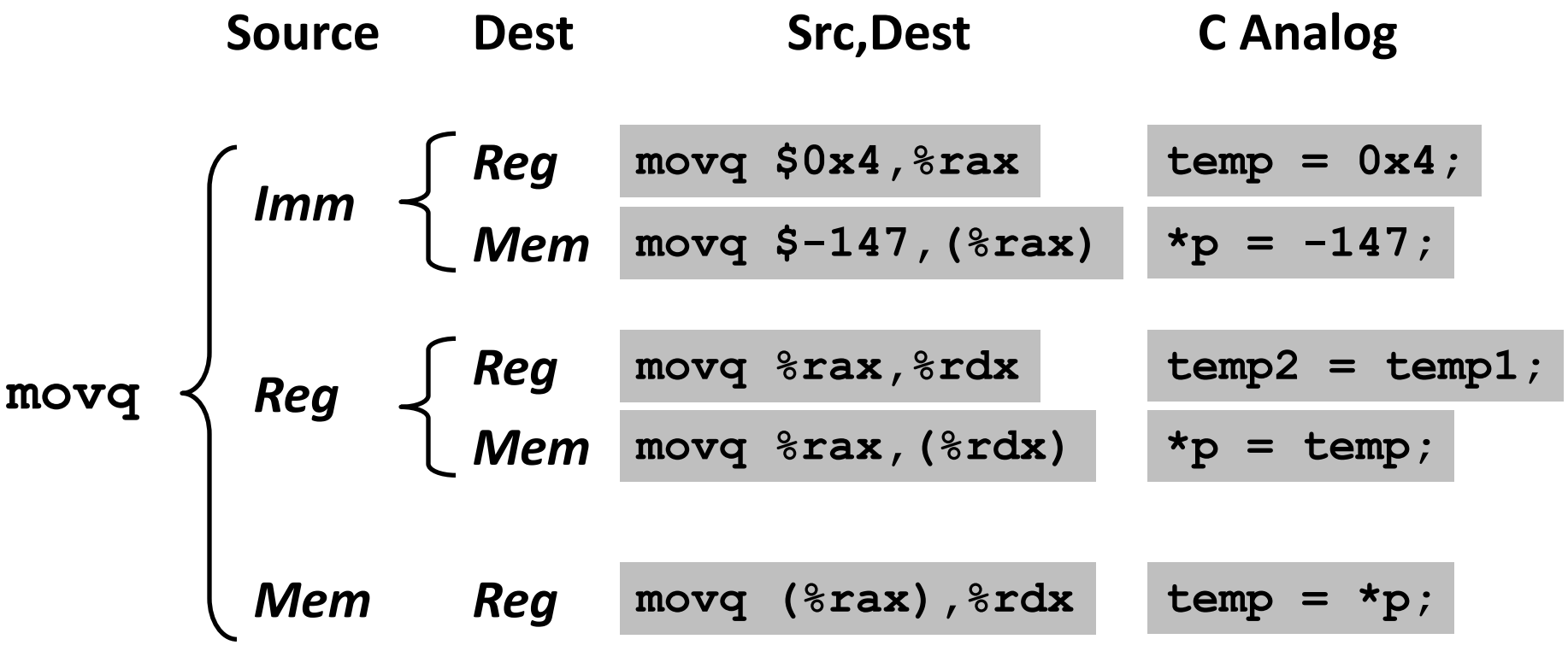
- **Moving Data**

  `movq` ***Src, Dest***

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, 4, or 8 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - Some registers (e.g., **`%rsp`**) are reserved for special use
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "addressing modes"



`movq` Operand Combinations



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        # movq Mem[%rcx], %rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register **R** specifies start of memory region
  - Constant displacement **D** specifies offset

```
movq 8(%rcx),%rax       # movq Mem[%rcx+8], %rax
```

Example: Simple Addressing Modes

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

Q. Can you summarize

what the high-level operation of these instructions perform?

Example: Simple Addressing Modes

```c
#include <stdio.h>

long x = 1234;
long y = 5678;

void swap(long *xp, long *yp){
    long tmp0 = *xp;
    long tmp1 = *yp;

    *xp = tmp1;
    *yp = tmp0;
}

int main(void) {
    printf("[before] %ld %ld\n", x, y);
    swap(&x, &y);
    printf("[after] %ld %ld\n", x, y);

    return 0;
}
```

```
        .globl  swap
        .type   swap, @function
swap:
        movq    (%rdi), %rax    # *xp_2(D), tmp0
        movq    (%rsi), %rdx    # *yp_4(D), tmp1
        movq    %rdx, (%rdi)    # tmp1, *xp_2(D)
        movq    %rax, (%rsi)    # tmp0, *yp_4(D)
        ret
```

```
$ gcc -O2 -S arith.c -o arith.s

$ gcc -c arith.s -o arith.o

$ gcc arith.o -o arith
```

Complete Memory Addressing Modes

- **Most General Form**

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

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

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

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

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

Expression	Meaning	Address Computation	Address
`0x8(%rdx)`	`Mem[%rdx+8]`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`Mem[%rdx+%rcx]`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`Mem[%rdx+4*%rcx]`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`Mem[2*%rdx+0x80]`	`2*0xf000 + 0x80`	`0x1e080`

Today: Machine Programming I: Basics

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

Address Computation Instruction

- **`leaq` *Src*, *Dst***
  - LEA: Load Effective Address
  - *Src* is address mode expression
  - *Dst* is a register, which will be set with address denoted by expression

- **Uses**
  - Computing addresses
    - E.g., **`p = &x[i];`**
  - Computing arithmetic expressions of the form “x + k*y”
    - k = 1, 2, 4, or 8

- **Example**

```
long m3(long x){
  return x*3;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # rax = rdi+2*rdi
```

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 shift***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical shift***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- **Watch out for argument order! *Src,Dest***
  (Warning: Intel docs use "op *Dest,Src*")

Some Arithmetic Operations

- **One Operand Instructions**

`incq`	*Dest*	*Dest = Dest + 1*	
`decq`	*Dest*	*Dest = Dest – 1*	
`negq`	*Dest*	*Dest = – Dest*	***Two's complement negation***
`notq`	*Dest*	*Dest = ~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 res = t2 * t5;

  return res;
}
```

```
arith:
        leaq    (%rsi,%rsi,2), %rax     #, t4
        salq    $4, %rax                #, tmp96
        leaq    4(%rdi,%rax), %rax      #, t5
        addq    %rsi, %rdi              # y, t1
        addq    %rdx, %rdi              # tmp102, t2
        imulq   %rdi, %rax              # t2, res
        ret
```

Interesting Instructions

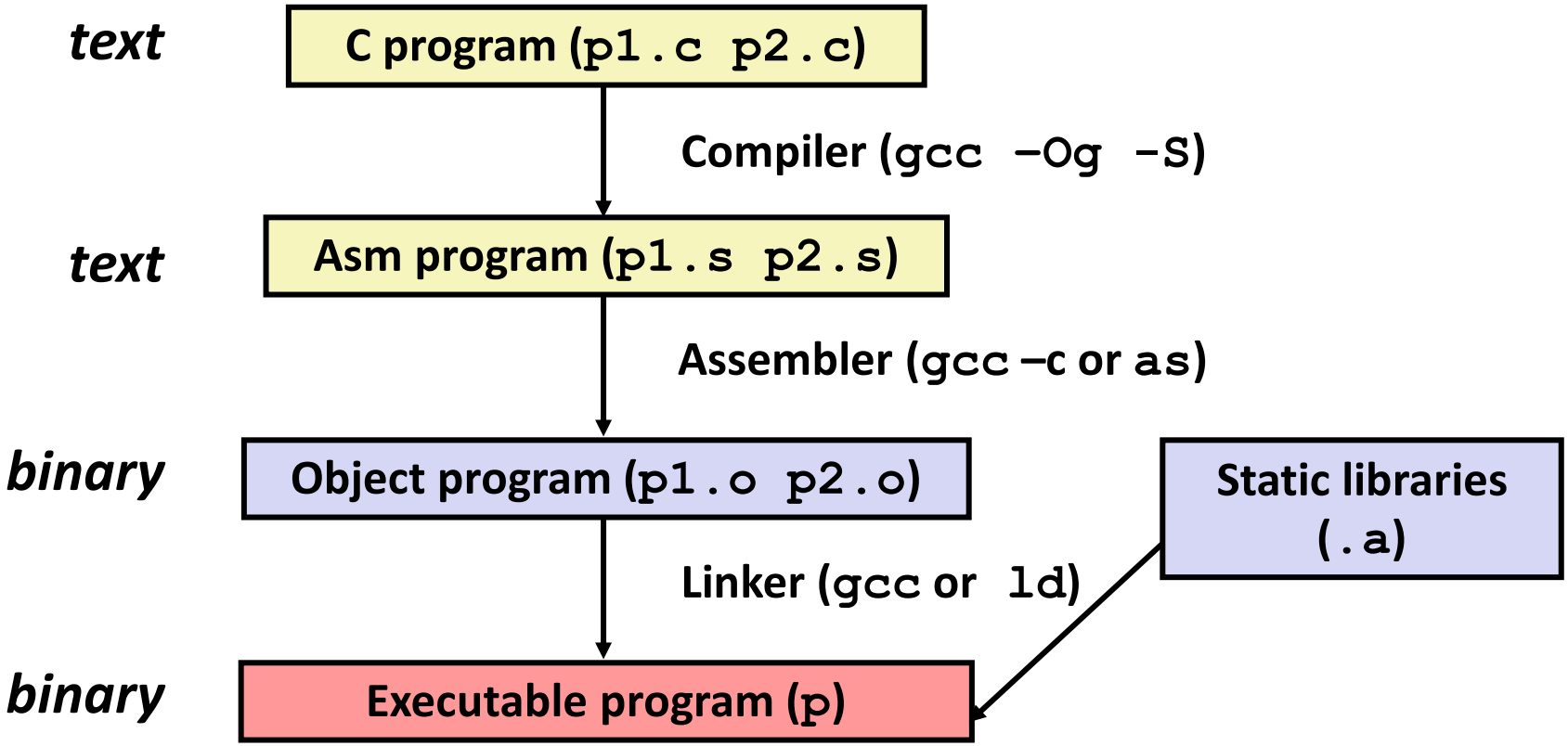
- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Today: Machine Programming I: Basics

- Intel processors and architectures
- 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 basic optimizations (`–Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`



Assembler and Linker

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code

- **Linker**
  - Resolves references between `.o` files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution
  - We will cover details in the later session

Machine Instruction Example

```
*dest = t;
```

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

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

```
0100 1 0 0 0   10001011   00 000 011
REX  W R X B     Move     Mod   R   M
```

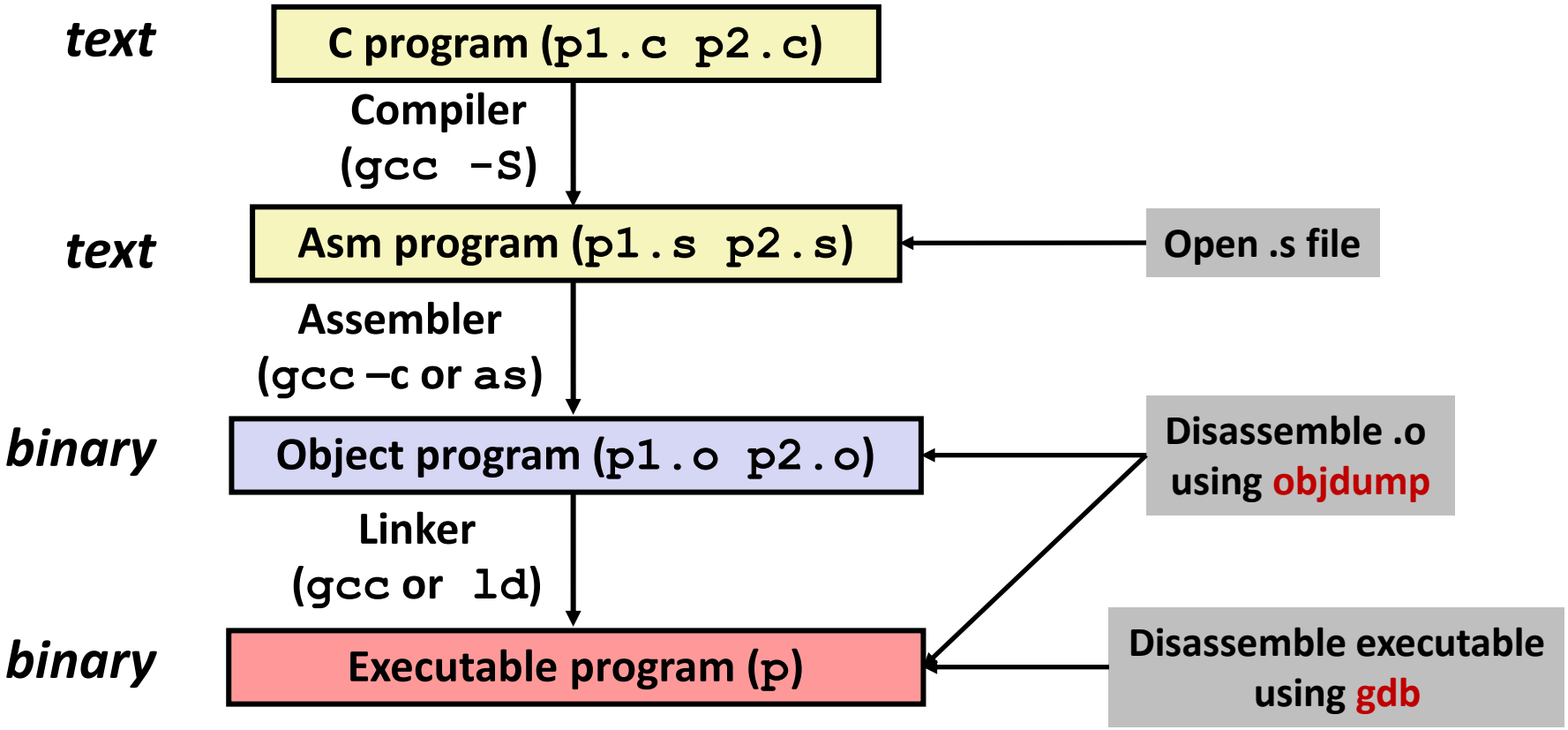
- **C**
  - Store value **t** where designated by **dest**
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

    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

Accessing Assemblies



Open assembly file

```
└─$ cat ./swap.s
```

```
swap:
# swap.c:7:      long tmp0 = *xp;
        movq    (%rdi), %rax    # *xp_2(D), tmp0
# swap.c:8:      long tmp1 = *yp;
        movq    (%rsi), %rdx    # *yp_4(D), tmp1
# swap.c:10:      *xp = tmp1;
        movq    %rdx, (%rdi)    # tmp1, *xp_2(D)
# swap.c:11:      *yp = tmp0;
        movq    %rax, (%rsi)    # tmp0, *yp_4(D)
# swap.c:12: }
        ret
```

Disassembly using objdump

```
└─$ objdump -d ./swap.o

./swap.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <swap>:
   0:   48 8b 07                mov    (%rdi),%rax
   3:   48 8b 16                mov    (%rsi),%rdx
   6:   48 89 17                mov    %rdx,(%rdi)
   9:   48 89 06                mov    %rax,(%rsi)
   c:   c3                      ret
```

Disassembly using objdump

```
└─$ objdump -d ./swap
```

```
00000000000011a0 <swap>:
    11a0:       48 8b 07                mov    (%rdi),%rax
    11a3:       48 8b 16                mov    (%rsi),%rdx
    11a6:       48 89 17                mov    %rdx,(%rdi)
    11a9:       48 89 06                mov    %rax,(%rsi)
    11ac:       c3                      ret
```

Disassembly using gdb

```
└─$ gdb -q ./swap
GEF for linux ready, type `gef' to start, `gef config' to
93 commands loaded and 5 functions added for GDB 15.0.50.
2
Reading symbols from ./swap...
(No debugging symbols found in ./swap)
gef➤
gef➤  set disassembly-flavor att
gef➤
gef➤  disas swap
Dump of assembler code for function swap:
   0x00000000000011a0 <+0>:     mov    (%rdi),%rax
   0x00000000000011a3 <+3>:     mov    (%rsi),%rdx
   0x00000000000011a6 <+6>:     mov    %rdx,(%rdi)
   0x00000000000011a9 <+9>:     mov    %rax,(%rsi)
   0x00000000000011ac <+12>:    ret
End of assembler dump.
gef➤
```