Machine-Level Programming I: Basics (Cont)

B&O Readings: 3.1-3.5
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Today: Machine Programming I: Basics

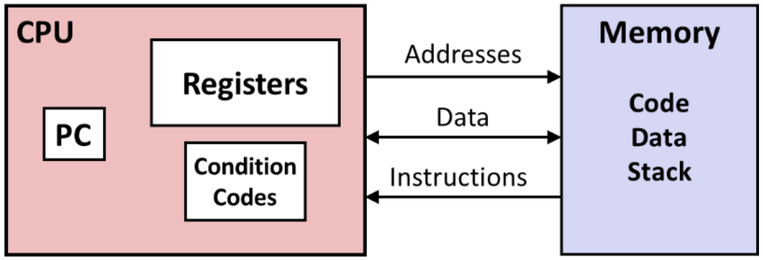
- What is ISA and a little bit of history
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

Levels of Abstraction

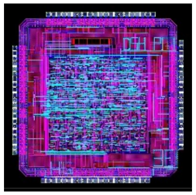
C programmer

```
#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; }
```

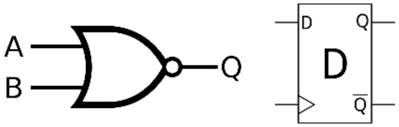
Assembly programmer



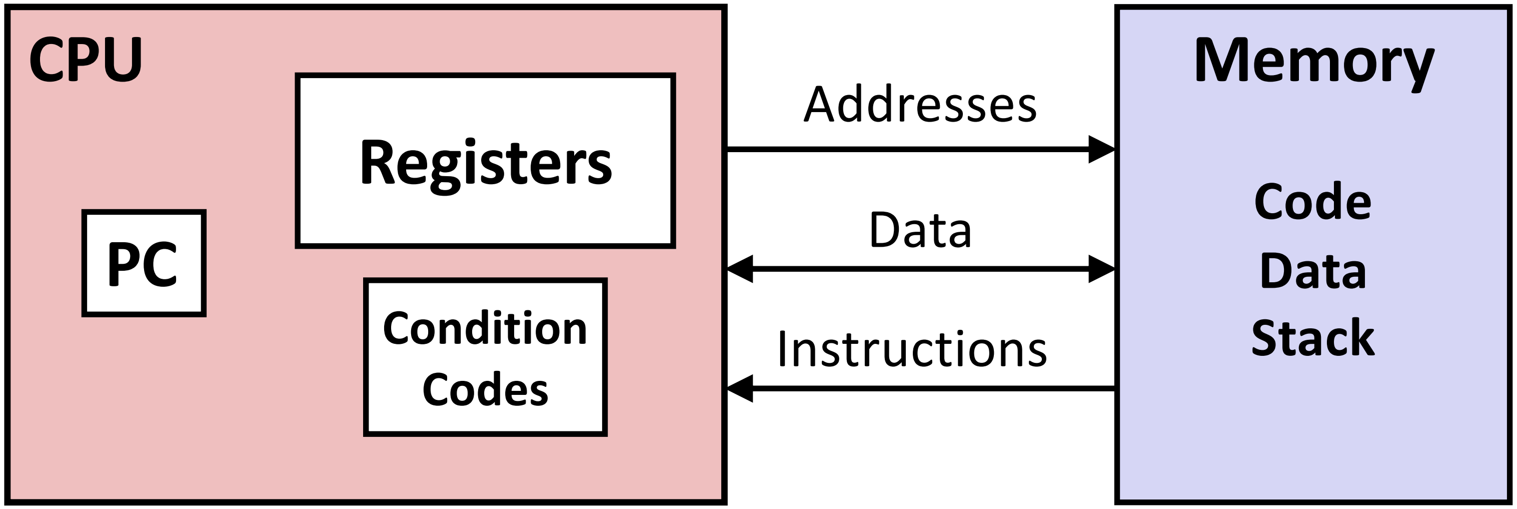
Computer Designer



Gates, clocks, circuit layout, …



Assembly Programmer's 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 Characteristics: Data Types

- **"Integer" data of 1, 2, 4, or 8 bytes**
  - Instruction suffixed with `b` (byte), `w` (word), `l` (long), and `q` (quad)
    - though suffix not actually important
  - Actual size of operand determined by register names (more later)
  - Addresses are always 8 bytes (untyped pointers)

- **Floating point data of 4 or 8 bytes**

- **Code: Byte sequences encoding series of instructions**

- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

Assembly Characteristics: 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**

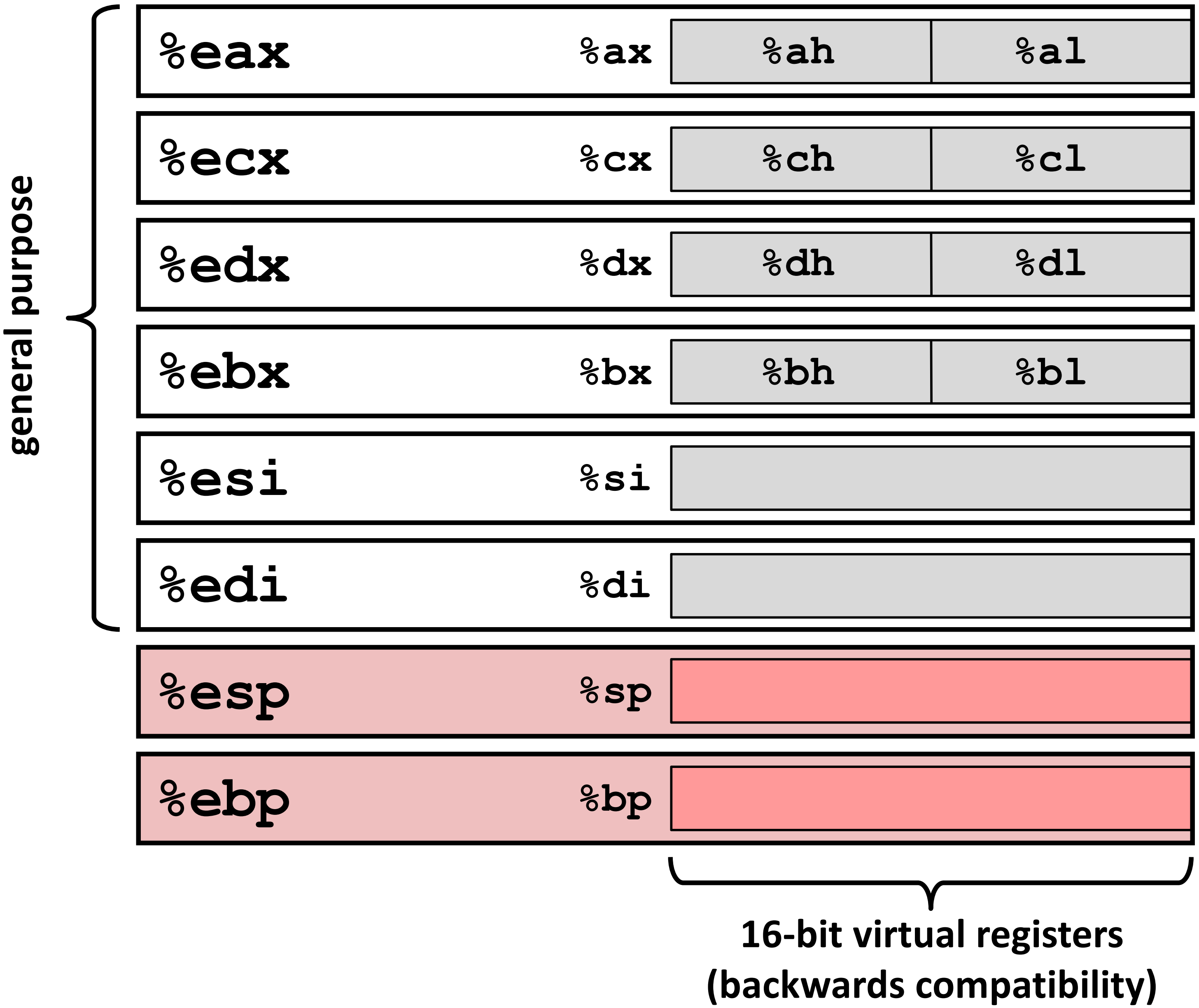
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

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`

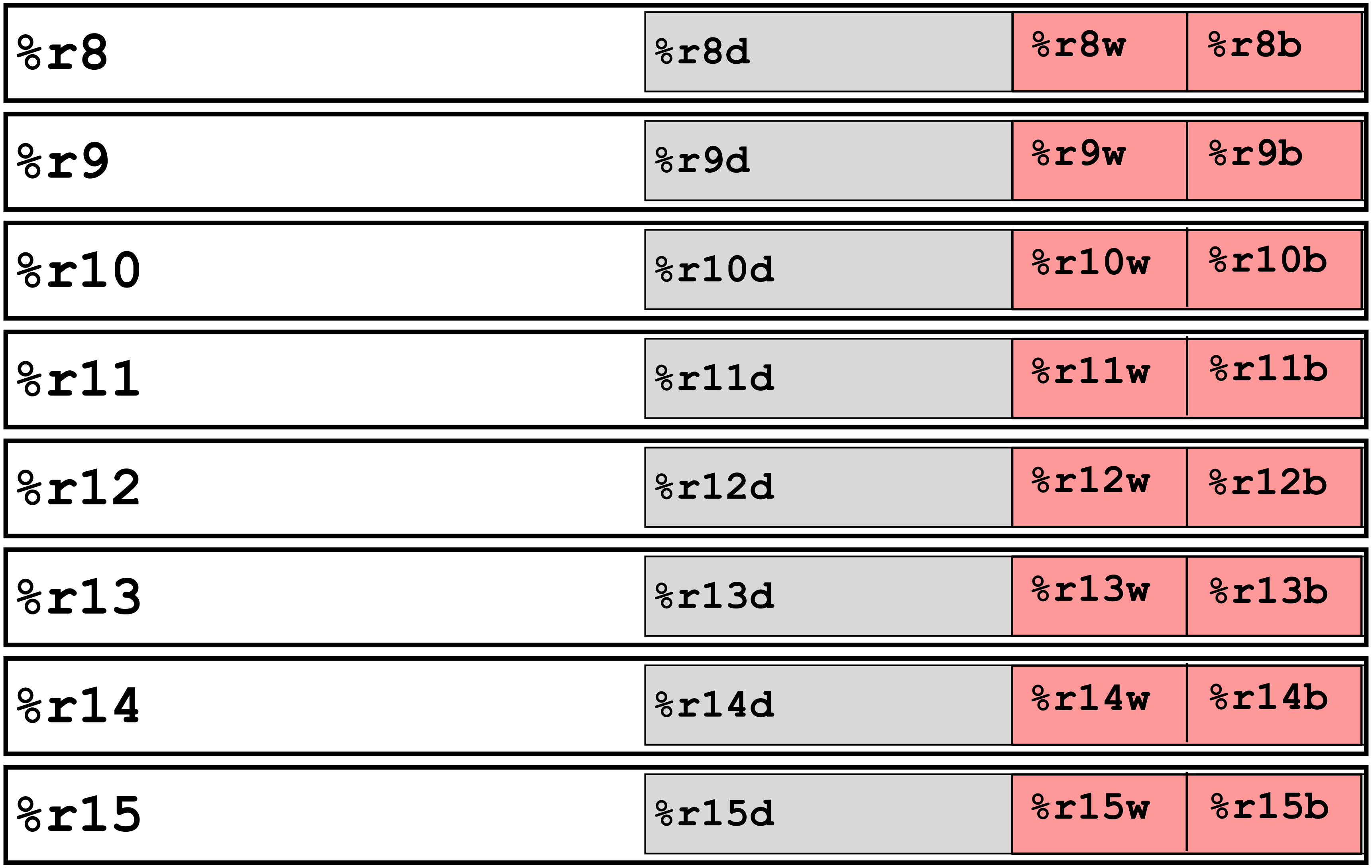
- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers



				Origin (mostly obsolete)
%eax	%ax	%ah	%al	*accumulate*
%ecx	%cx	%ch	%cl	*counter*
%edx	%dx	%dh	%dl	*data*
%ebx	%bx	%bh	%bl	*base*
%esi	%si			*source index*
%edi	%di			*destination index*
%esp	%sp			***stack pointer***
%ebp	%bp			***base pointer***

x86-64 Integer Registers



%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

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

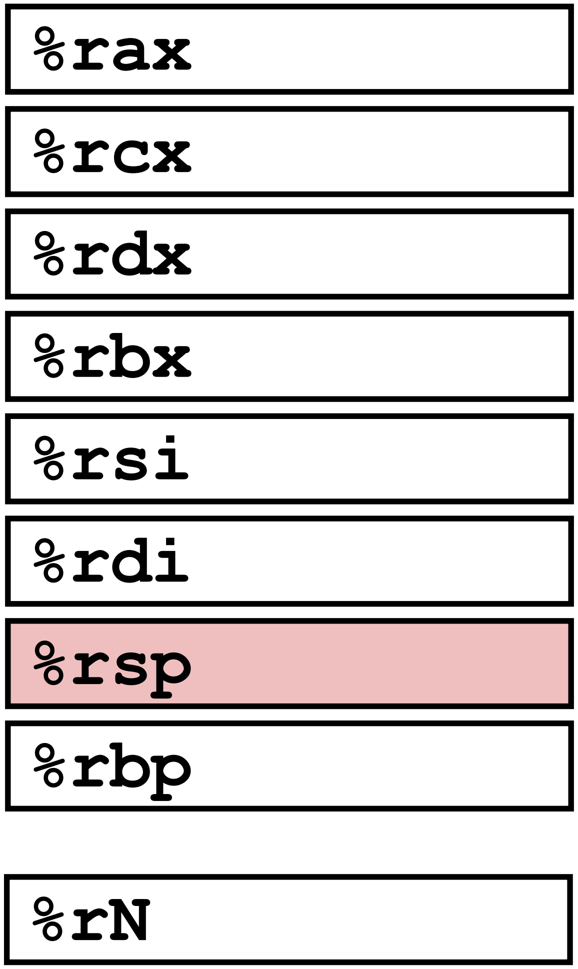
Moving Data

- **Moving Data**

  `movq` ***Source*, *Dest*;**

  (Also `movl`, `movw`, `movb`)

- **x86-64 can still use 32-bit instructions that generate 32-bit results**
  - Higher-order bits of destination register are just set to 0
  - Example: **`movl   %eax, %ebx`**



%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

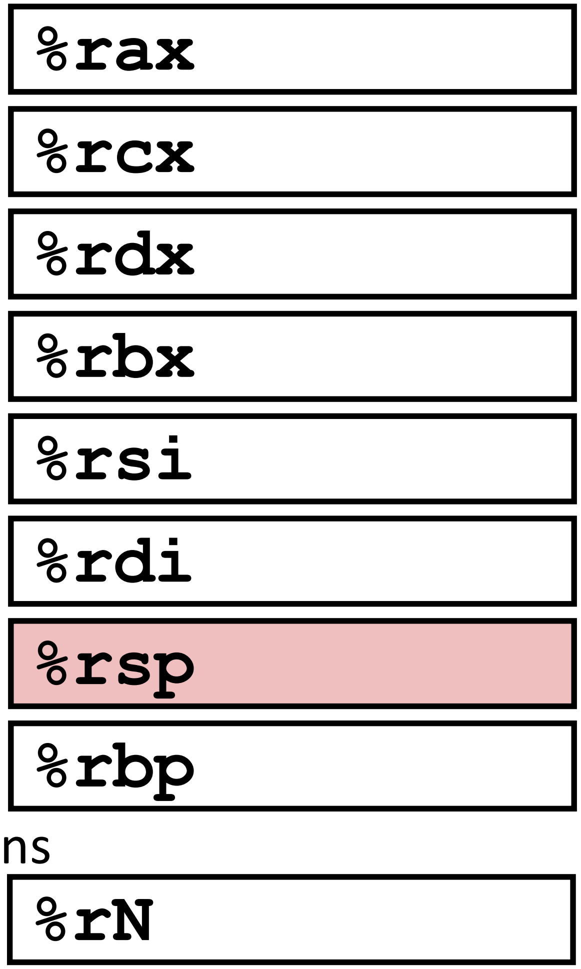
%rbp

%rN

Warning: Intel docs use mov *Dest, Source*

Moving Data: 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:***
  - Immediate (constant) but *NOT* prefixed with '$'
  - consecutive bytes of memory at address given by register
    - Have to use the 8-byte form for the register! Ex: **`(%rax)`**
  - Various other "address modes"



%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

%rbp

%rN

`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
```

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]]**

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x0100`

Expression	Address Computation	Address
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Example via Swap()

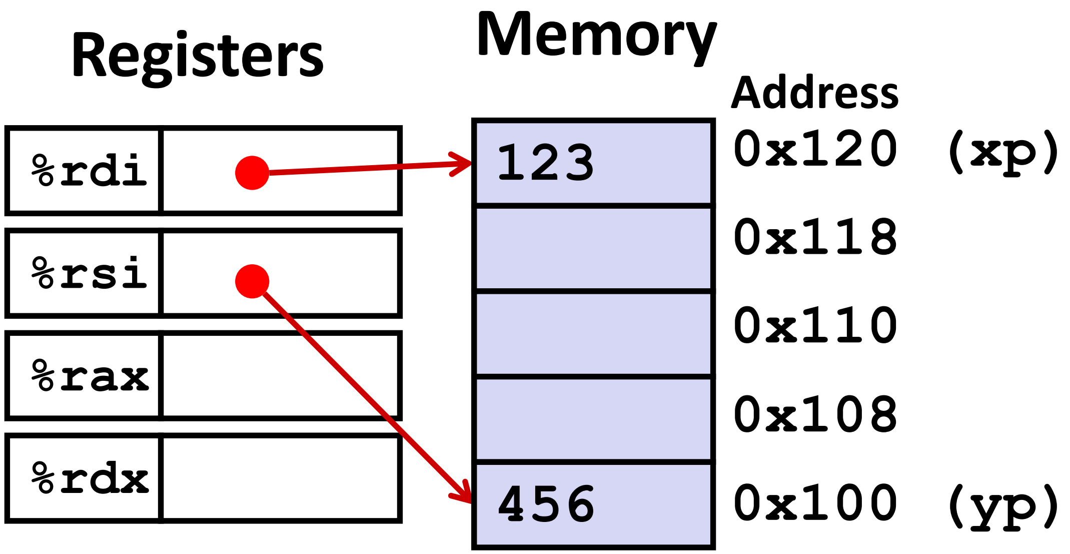
```
void swap(long *xp,
          long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Register	Value
%rdi	xp
%rsi	yp

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

	Address
123	0x120 (xp)
	0x118
	0x110
	0x108
456	0x100 (yp)

Register	Value
%rdi	xp
%rsi	yp

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

Understanding Swap()

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

Registers

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

Memory

	Address
123	0x120 (xp)
	0x118
	0x110
	0x108
456	0x100 (yp)

Register	Value
%rdi	xp
%rsi	yp

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

Understanding Swap()

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

Registers

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

Memory

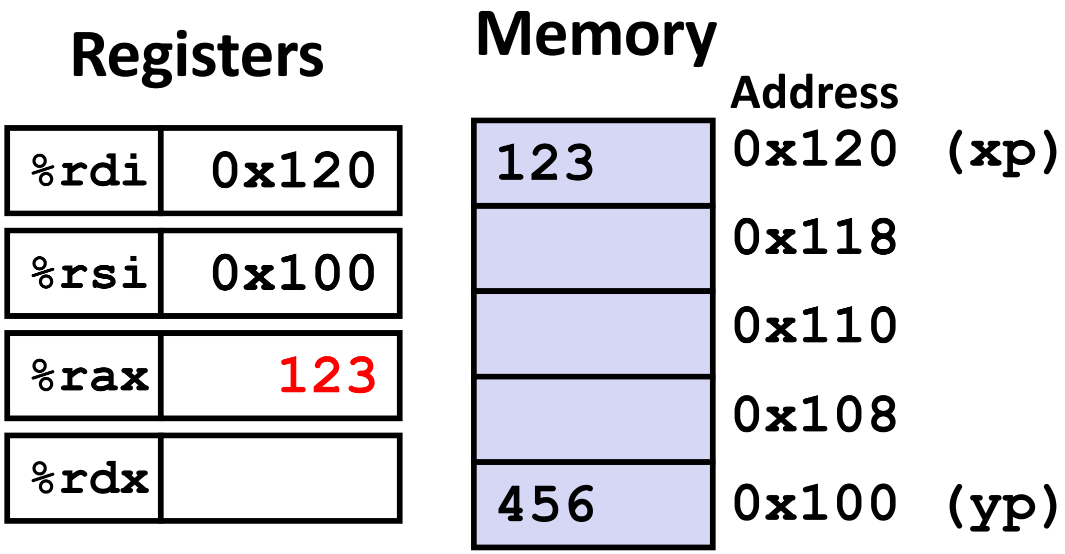
	Address	
123	0x120	(xp)
	0x118	
	0x110	
	0x108	
456	0x100	(yp)

Register	Value
%rdi	xp
%rsi	yp

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

Understanding Swap()

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



Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Memory

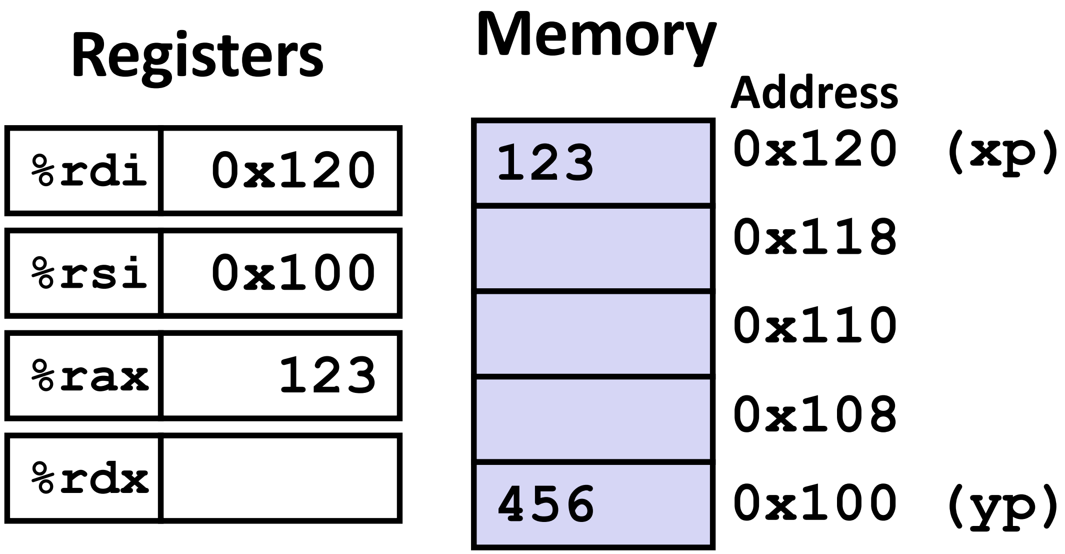
	Address	
123	0x120	(xp)
	0x118	
	0x110	
	0x108	
456	0x100	(yp)

Register	Value
%rdi	xp
%rsi	yp
%rax	t0

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

Understanding Swap()

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



Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Memory

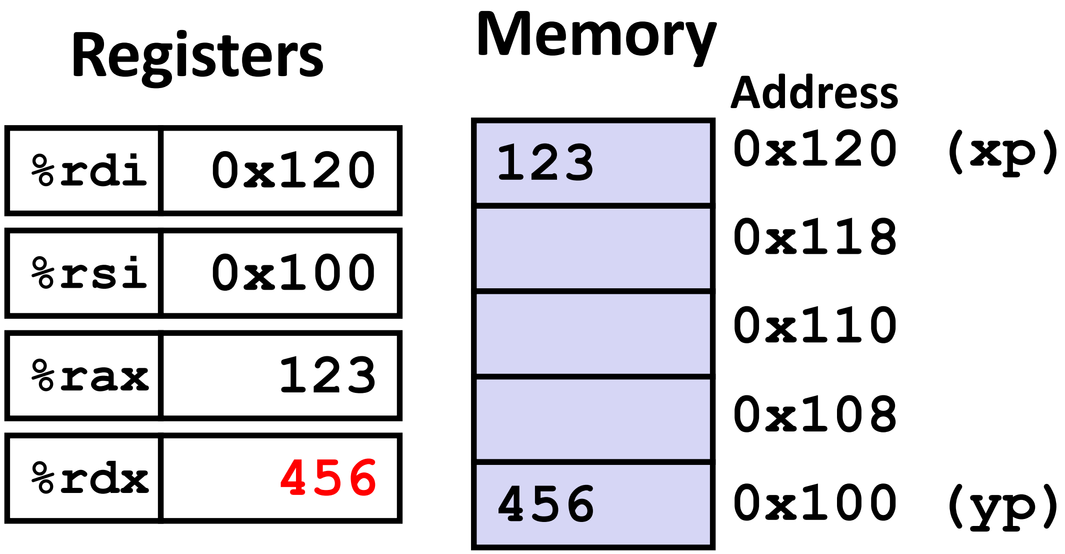
	Address
123	0x120 (xp)
	0x118
	0x110
	0x108
456	0x100 (yp)

Register	Value
%rdi	xp
%rsi	yp
%rax	t0

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

Understanding Swap()

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



Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

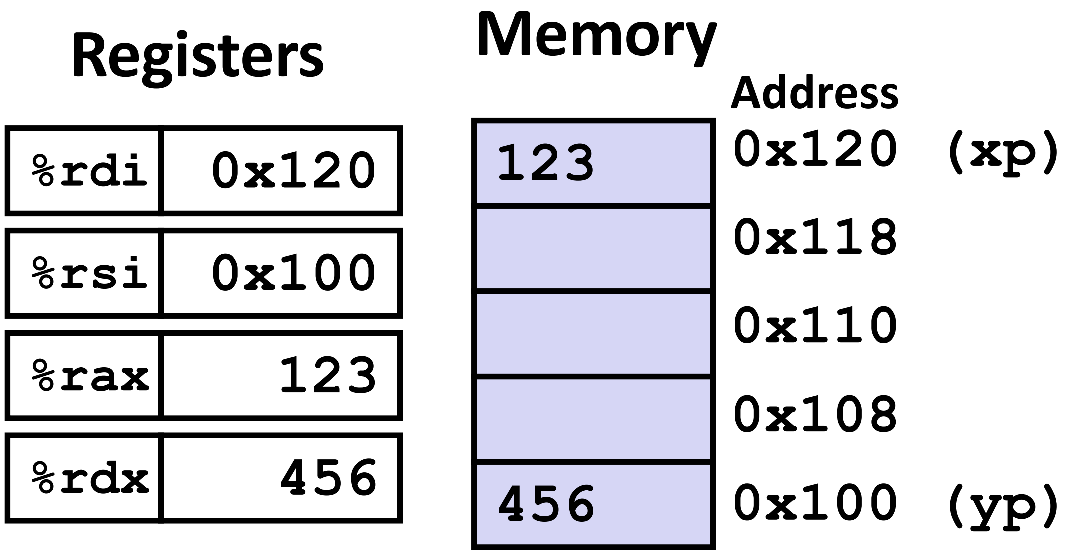
	Address
123	0x120 (xp)
	0x118
	0x110
	0x108
456	0x100 (yp)

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()

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



Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

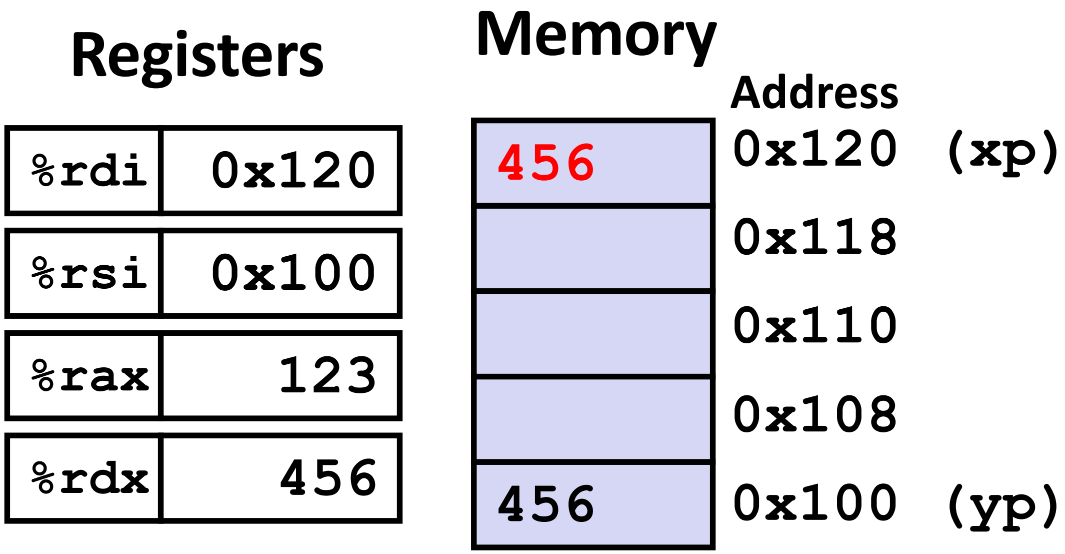
	Address
123	0x120 (xp)
	0x118
	0x110
	0x108
456	0x100 (yp)

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()

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



Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

	Address
456	0x120 (xp)
	0x118
	0x110
	0x108
456	0x100 (yp)

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()

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

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

	Address	
456	0x120	(xp)
	0x118	
	0x110	
	0x108	
456	0x100	(yp)

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()

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

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

	Address	
456	0x120	(xp)
	0x118	
	0x110	
	0x108	
123	0x100	(yp)

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

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

Code Puzzle: Implement `foo`

```
void foo(int a, int *p) {


}
```

```
movl   %edi, 4(%rsi)
ret
```

Register	Use(s)
`%edi`	Argument **`a`**
`%rsi`	Argument **`p`**

Recap: What We Learned Thus Far

- **How assembly instructions move data around**
  - The set of registers available to you
  - How to reference data in memory (addressing mode)
  - The format of move instructions

Today: Machine Programming I: Basics

- What is ISA and a little bit of history
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

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

- The Src can be reg, imm, or mem; the Dest can be reg or mem.
- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions

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

- See book for more instructions:
  `movzbw`, **`movzbl`**, **`movzwl`**, **`movzbq`**, **`movzwq`**
  `movsbw`, **`movsbl`**, **`movswl`**, **`movsbq`**, **`movswq`**, **`movslq`**

- Why is there not a **`movzlq`**?

Address Computation Instruction

- **`leaq`** Src, Dst
  - Src is address mode expression (i.e., in the form of D(Rb,Ri,S))
  - Set Dst to address denoted by expression Src
  - (lea stands for *load effective address)*
- 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, or 8
- Example

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

Converted to ASM by compiler:

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

Code Puzzle: Fill in the Blank

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

  return ___________;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	return value

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

Interesting Instructions

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

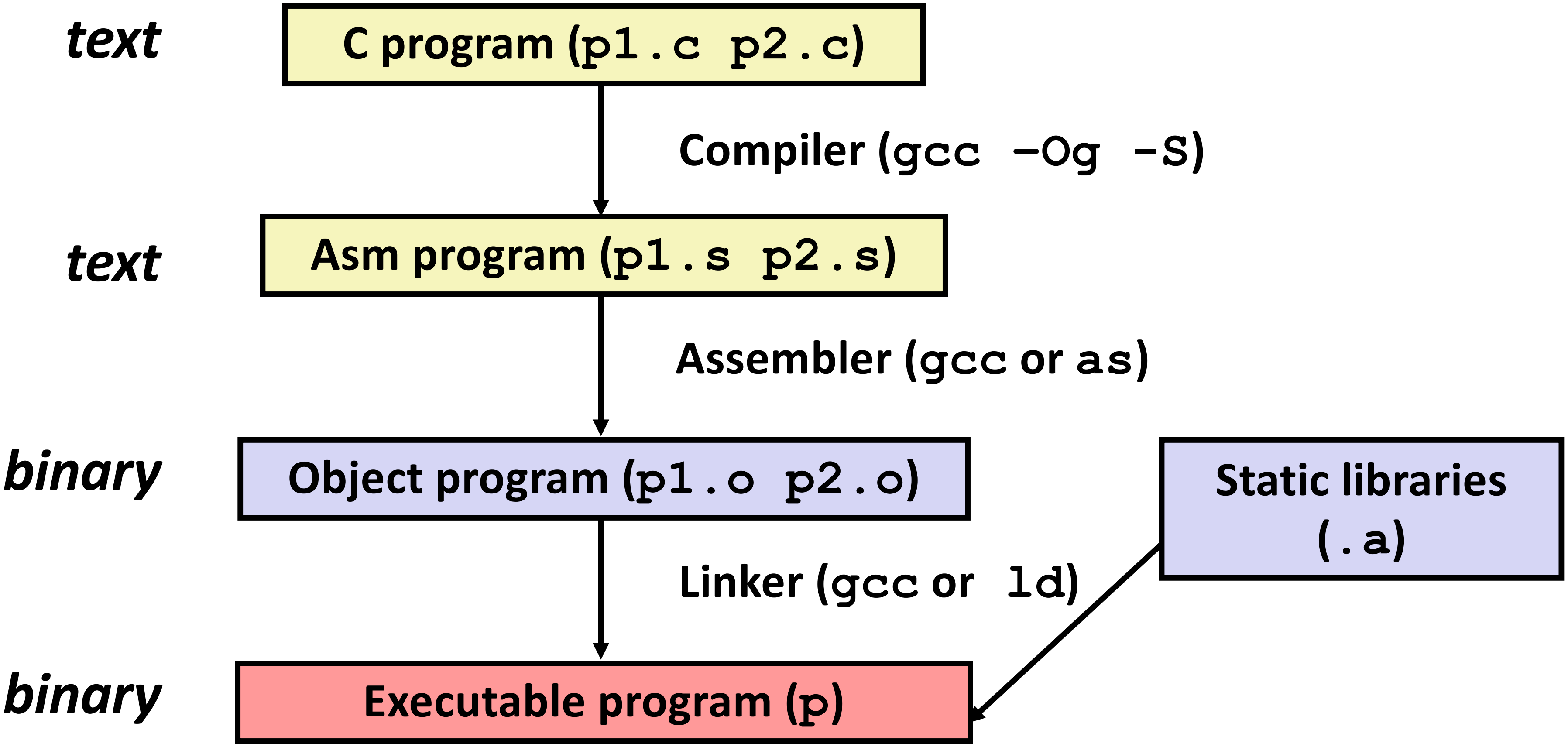
Answer: **`t1*t4`**

Today: Machine Programming I: Basics

- What is ISA and a little bit of history
- 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`



Compiling Into Assembly

C Code (absdiff.c)

```
void absdiff(
long x, long y, long *dest) {
    long t = 0;
    if(x > y) { t = x - y; }
    else      { t = y - x; }
    *dest = t;
}
```

Generated x86-64 Assembly

```
absdiff:
   cmpq    %rsi, %rdi
   jle     .L2
   subq    %rsi, %rdi
   movq    %rdi, %rsi
   jmp     .L3
.L2:
   subq    %rdi, %rsi
.L3:
   movq    %rsi, (%rdx)
   ret
```

Obtain with command (on linuxlab machines)

```
gcc -Og -S absdiff.c
```

Produces file `absdiff.s`

Warning: your result may vary due to different compiler versions or platform

Object Code

Object code for `absdiff`

```
0000000000000000 <absdiff>:
   0:    48 39 f7           cmp    %rsi,%rdi
   3:    7e 08              jle    d <absdiff+0xd>
   5:    48 29 f7           sub    %rsi,%rdi
   8:    48 89 fe           mov    %rdi,%rsi
   b:    eb 03              jmp    10 <absdiff+0x10>
   d:    48 29 fe           sub    %rdi,%rsi
  10:    48 89 32           mov    %rsi,(%rdx)
  13:    c3                 retq
```

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
  - Obtained via `gcc –Og –g –c –o absdiff.o absdiff.c`
    `objdump -d diff.o > diff.d`

Disassembling the Final Executable

```
000000000040057d <absdiff>:
  40057d:   48 39 f7cmp   %rsi,%rdi
  400580:   7e 08         jle   40058a <absdiff+0xd>
  400582:   48 29 f7sub   %rsi,%rdi
  400585:   48 89 femov    %rdi,%rsi
  400588:   eb 03         jmp    40058d <absdiff+0x10>
  40058a:   48 29 fesub    %rdi,%rsi
  40058d:   48 89 32mov    %rsi,(%rdx)
  400590:   c3            retq
```

- **Ways to disassemble your binary:**
  - obtained via **`objdump -d absdiff`**
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either the final executable or a single **`*.o`** file

Alternate Disassembly via gdb

```
> gdb absdiff
```

```
(gdb) disassemble absdiff
Dump of assembler code for function absdiff:
   0x000000000040057d <+0>:   cmp    %rsi,%rdi
   0x0000000000400580 <+3>:   jle    0x40058a <absdiff+13>
   0x0000000000400582 <+5>:   sub    %rsi,%rdi
   0x0000000000400585 <+8>:   mov    %rdi,%rsi
   0x0000000000400588 <+11>:  jmp    0x40058d <absdiff+16>
   0x000000000040058a <+13>:  sub    %rdi,%rsi
   0x000000000040058d <+16>:  mov    %rsi,(%rdx)
   0x0000000000400590 <+19>:  retq
End of assembler dump.
```

```
(gdb) x/14xb absdiff
0x40057d <absdiff>:   0x48  0x39  0xf7  0x7e  0x08  0x48  0x29  0xf7
0x400585 <absdiff+8>: 0x48  0x89  0xfe  0xeb  0x03  0x48
```

Which of the Following Statement Is False?

A) The compiler performs type checks before translating the code into assembly.

B) A single C statement can potentially be compiled into multiple assembly instructions.

C) I cannot disassemble an executable on Windows because it's closed-source.

D) All of the above

Answer: C)

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**

Recap: What We Learned Thus Far

- How your compiler compile your code
- How to disassemble your code

- Why should you care?
  - Learning how to read assembly code allows you to better understand the behavior and performance of your program.
  - Software has bugs (including your compiler!)
    Learning how to read assembly code allows you to detect bugs introduced by system software.
  - Knowing assembly is sometimes required for developing low-level system software.

Machine Programming I: Summary

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