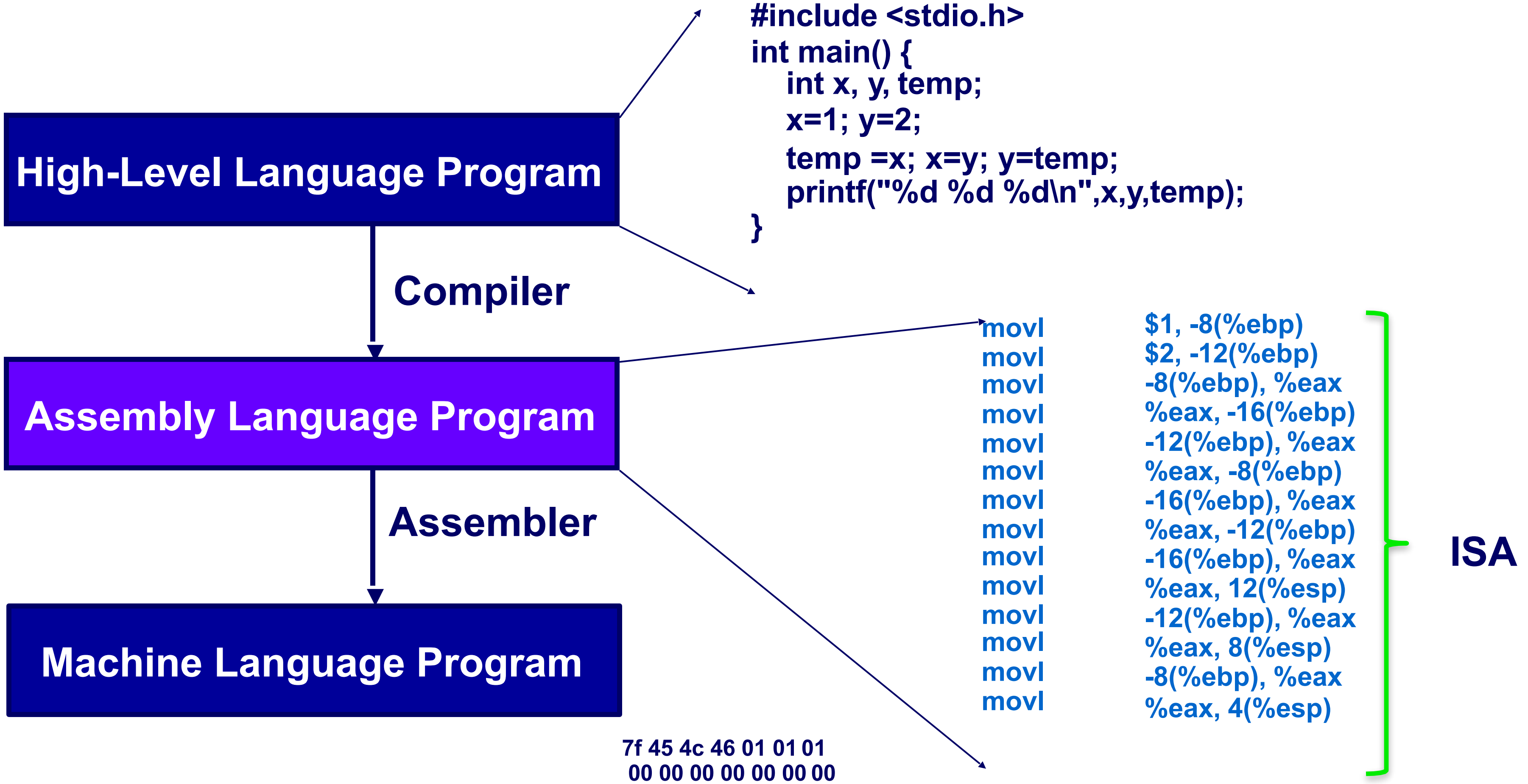
Intro To Assembly:

Topics:

- Hardware-Software Interface
- Assembly Programming
  - Reading: Chapter 3

Programming Meets Hardware



```
#include <stdio.h>
int main() {
    int x, y, temp;
    x=1; y=2;
    temp =x; x=y; y=temp;
    printf("%d %d %d\n",x,y,temp);
}
```

```
movl        $1, -8(%ebp)
movl        $2, -12(%ebp)
movl        -8(%ebp), %eax
movl        %eax, -16(%ebp)
movl        -12(%ebp), %eax
movl        %eax, -8(%ebp)
movl        -16(%ebp), %eax
movl        %eax, -12(%ebp)
movl        -16(%ebp), %eax
movl        %eax, 12(%esp)
movl        -12(%ebp), %eax
movl        %eax, 8(%esp)
movl        -8(%ebp), %eax
movl        %eax, 4(%esp)
```

```
7f 45 4c 46 01 01 01
 00 00 00 00 00 00 00
 00 00 02 00 03 00 01
00 00 00 f0 82 04 08
34 00 00 00 c4 0c 00
00 00 00 00 00 34 00
```

Performance with Programs

1. Program: Data structures + algorithms
2. Compiler translates code
3. Instruction set architecture
4. Hardware Implementation

Instruction Set Architecture

1. Set of instructions that the CPU can execute
    - What instructions are available?
    - How the instructions are encoded? Eventually everything is binary.
2. State of the system (Registers + memory state + program counter)
    - What instruction is going to execute next
    - How many registers? Width of each register?
    - How do we specify memory addresses?
        - Addressing modes
3. Effect of instruction on the state of the system

IA32 (X86 ISA)

- There are many different assembly languages because they are processor-specific
  - IA32 (x86)
    - x86-64 for new 64-bit processors
    - IA-64 radically different for Itanium processors
    - Backward compatibility: instructions added with time
  - PowerPC
  - MIPS
- We will focus on IA32/x86-64 because you can generate and run on iLab machines (as well as your own PC/laptop)
  - IA32 is also dominant in the market although smart phone, eBook readers, etc. are changing this

Aside About Implementation of x86

- About 30 years ago, the instruction set actually reflected the processor hardware
  - E.g., the set of registers in the instruction set is actually what was present in the processor
- As hardware advanced, industry faced with choice
  - Change the instruction set: bad for backward compatibility
  - Keep the instruction set: harder to exploit hardware advances
    - Example: many more registers but only small set introduced circa 1980
- Starting with the P6 (PentiumPro), IA32 actually got implemented by Intel using an "interpreter" that translates IA32 instructions into a simpler "micro" instruction set

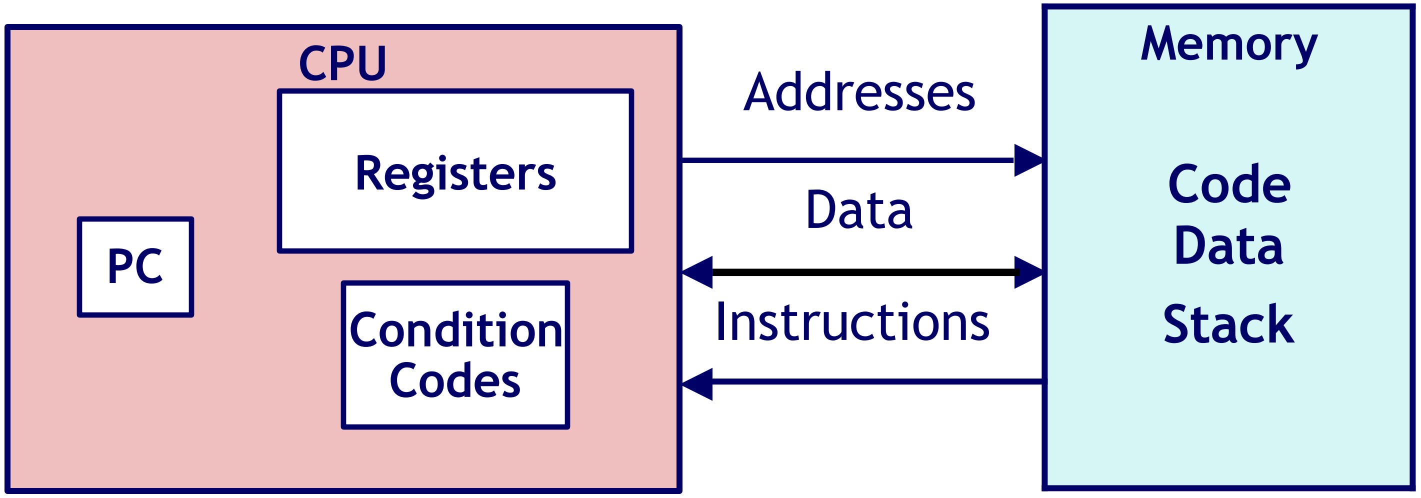
Assembly Programming

- Brief tour through assembly language programming
  - Why?
  - Machine interface: where software meets hardware
  - To understand how the hardware works, we have to understand the interface that it exports
- Why not binary language?
  - Much easier for humans to read and reason about
  - Major differences:
    - Human readable language instead of binary sequences
    - Relative instead of absolute addresses

Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write 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

Assembly/Machine Code View



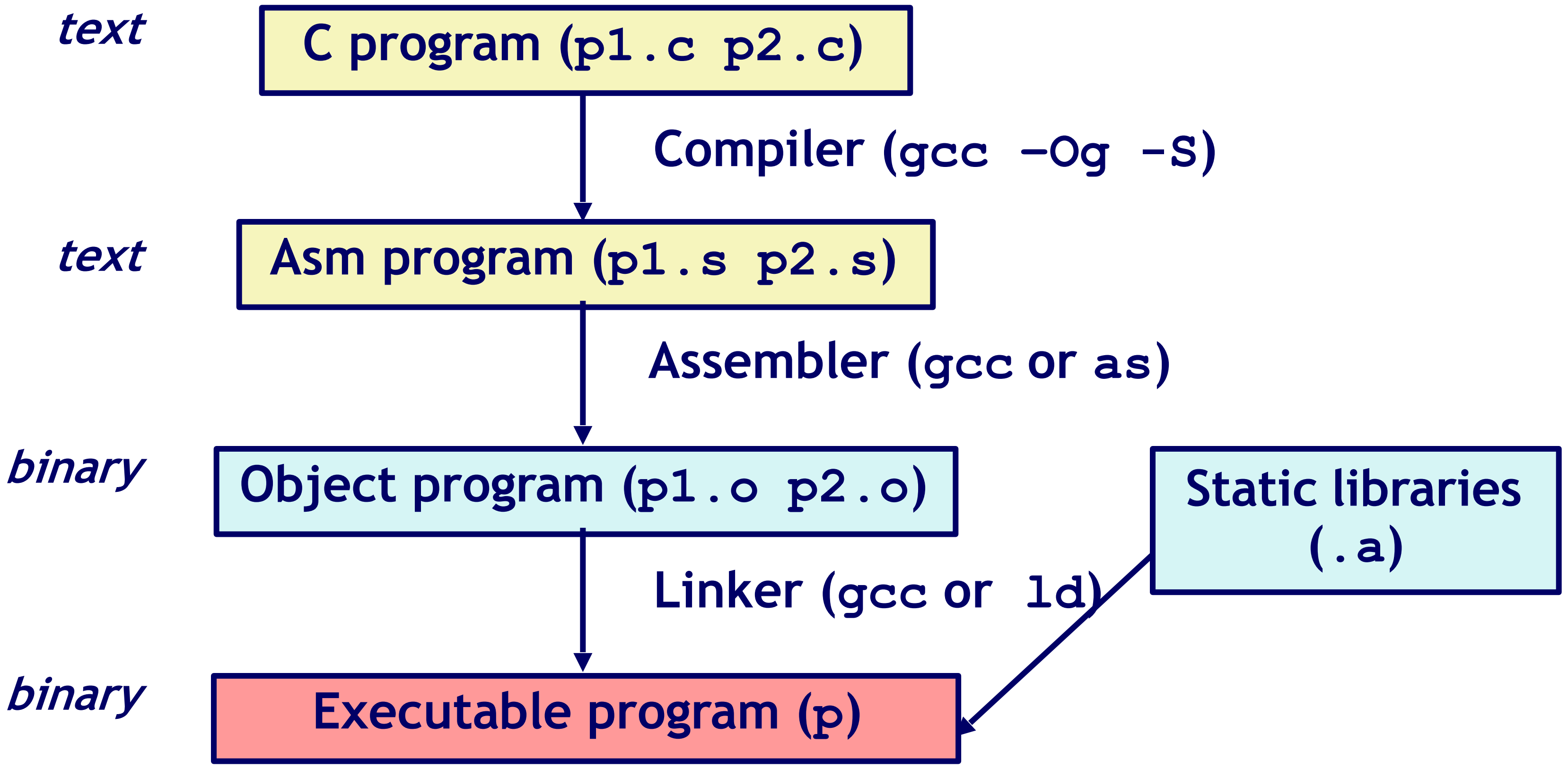
Programmer-Visible State

- **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

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 (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 other machines (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.

Assembly Characteristics: 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
- Code: Byte sequences encoding series of instructions (No aggregate types such as arrays or structures)
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

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

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;
```

- C Code
  - Store value **t** where designated by **dest**

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

- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

t:	Register	**%rax**
dest:	Register	**%rbx**
***dest**:	Memory	**M[%rbx]**

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

- 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

- Within gdb Debugger
  - **gdb sum**
    - Start program "sum" with gdb
  - **disassemble sumstore**
    - Disassemble procedure
  - **x/14xb sumstore**
    - Examine the 14 bytes starting at sumstore

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

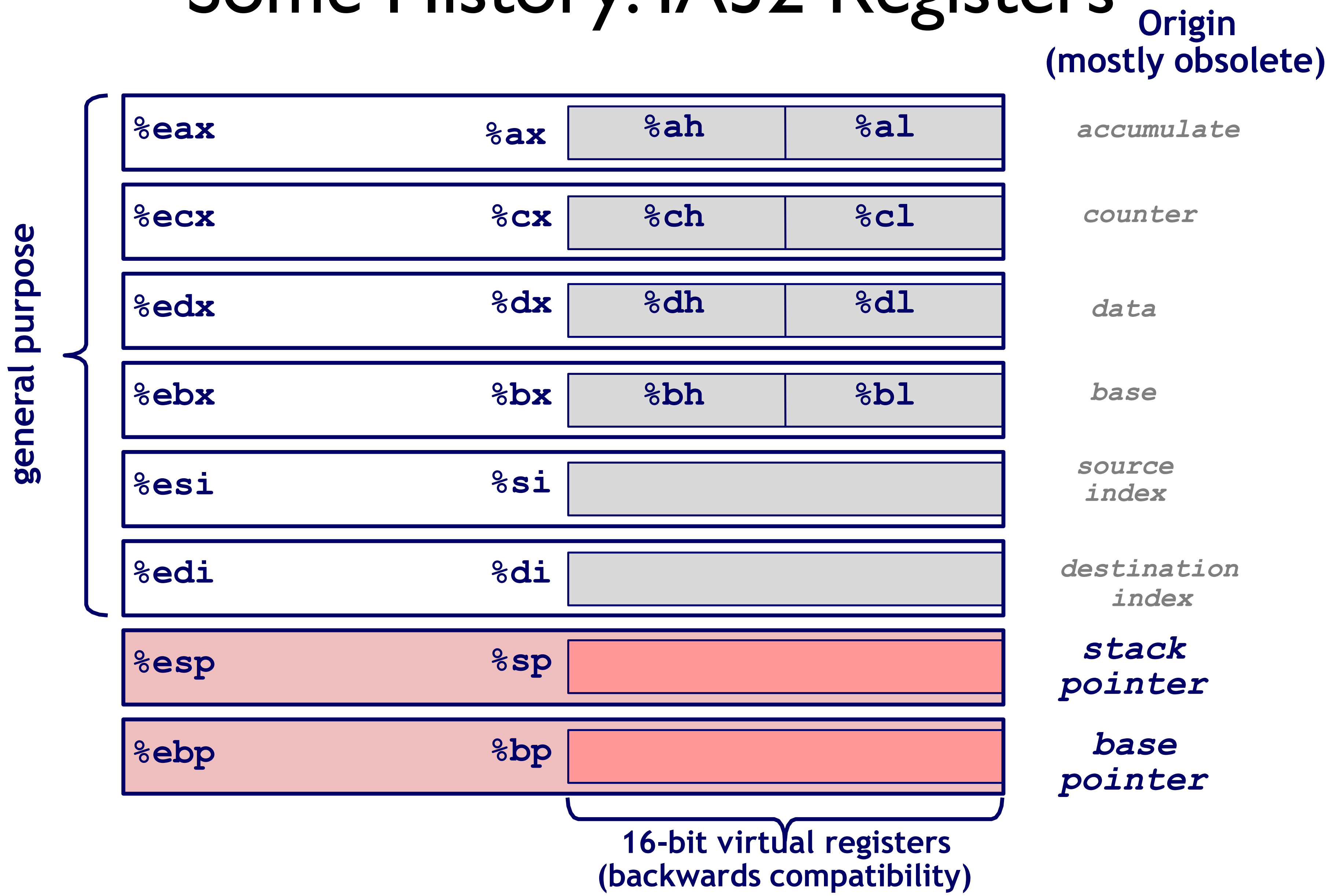
```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push    %rbx
 0x0000000000400596 <+1>: mov     %rdx,%rbx
 0x0000000000400599 <+4>: callq   0x400590 <plus>
 0x000000000040059e <+9>: mov     %rax,(%rbx)
 0x00000000004005a1 <+12>:pop     %rbx
 0x00000000004005a2 <+13>:retq
```

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`

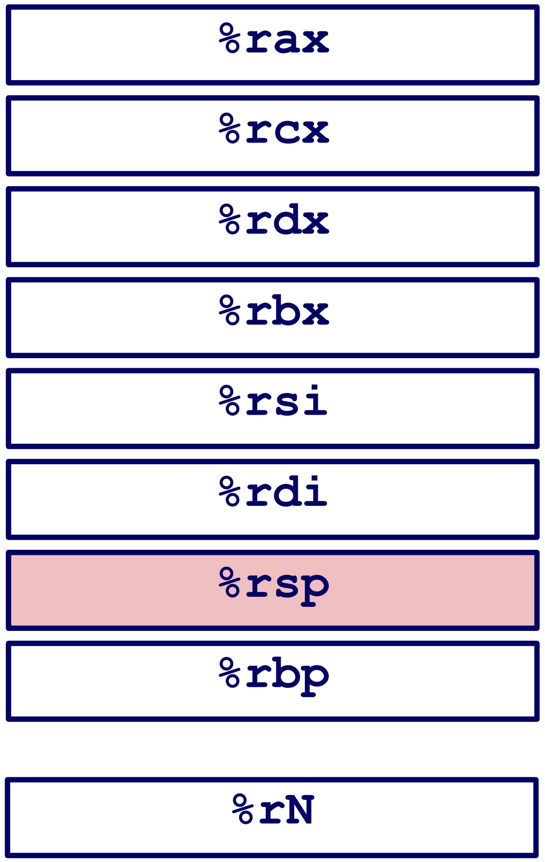
- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers



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 "address modes"



`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
  - Example
    - **`movq (%rcx),%rax`**
- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Example:
    - **`movq 8(%rbp),%rdx`**

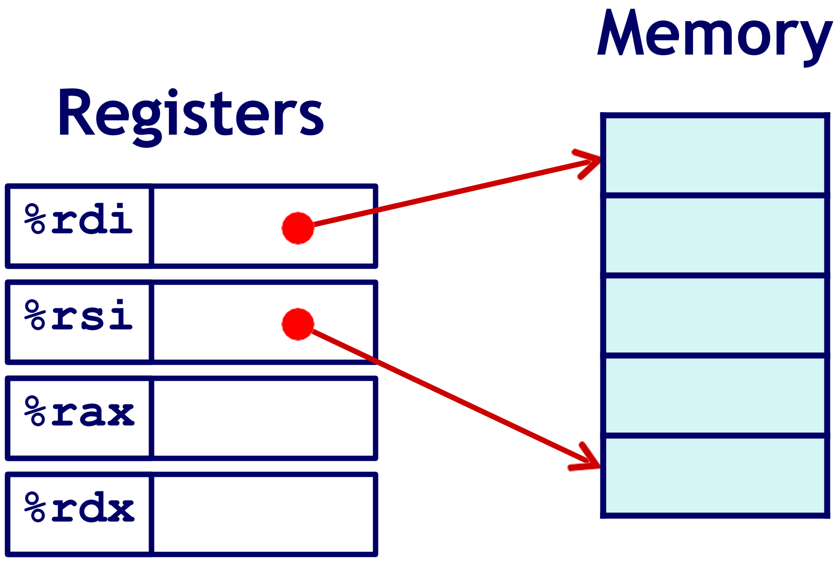
Example of Simple Addressing Modes

```
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()

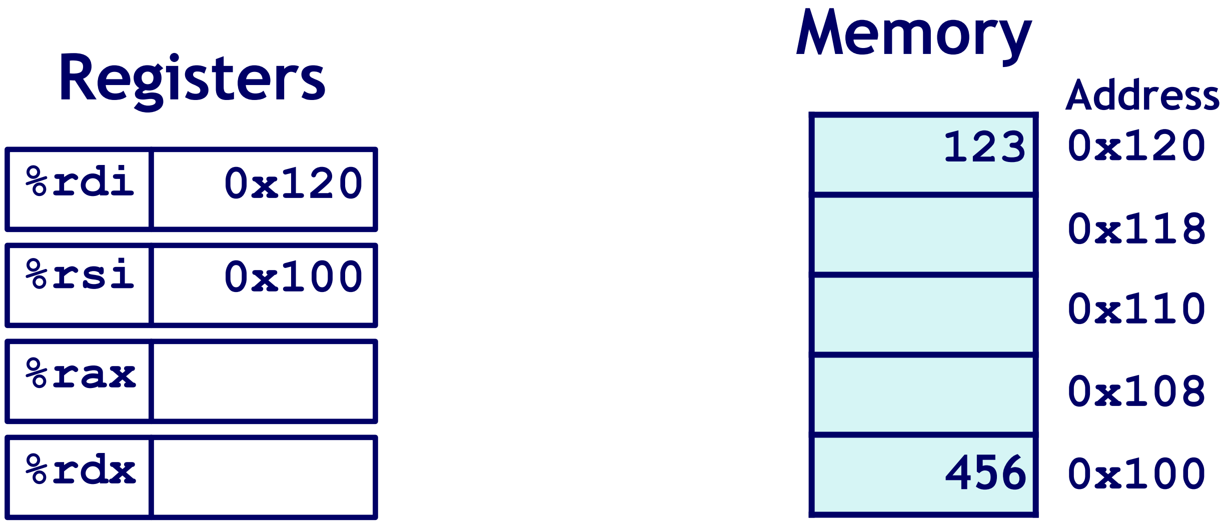
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

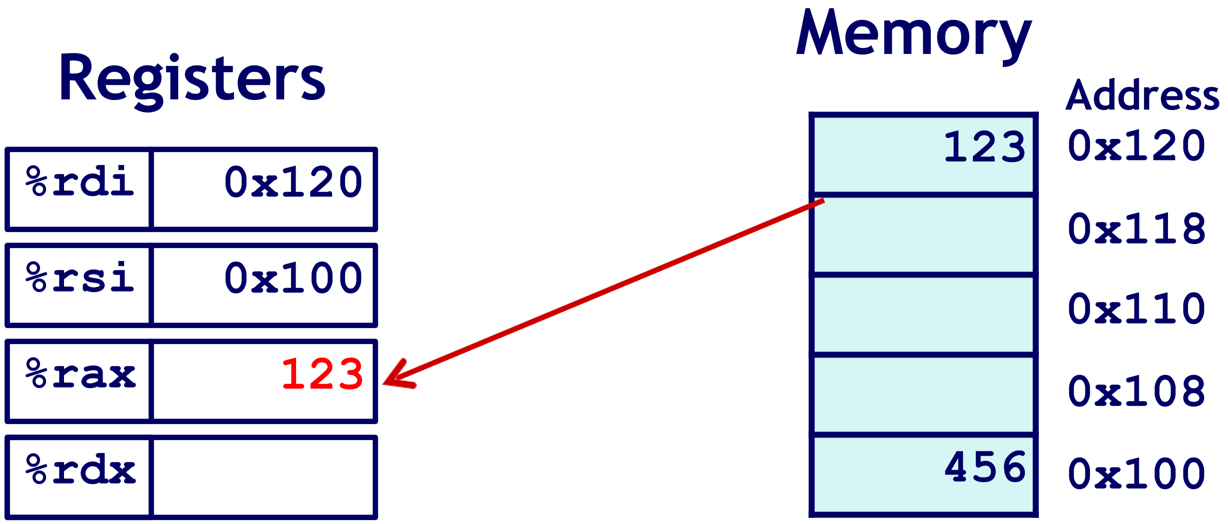
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



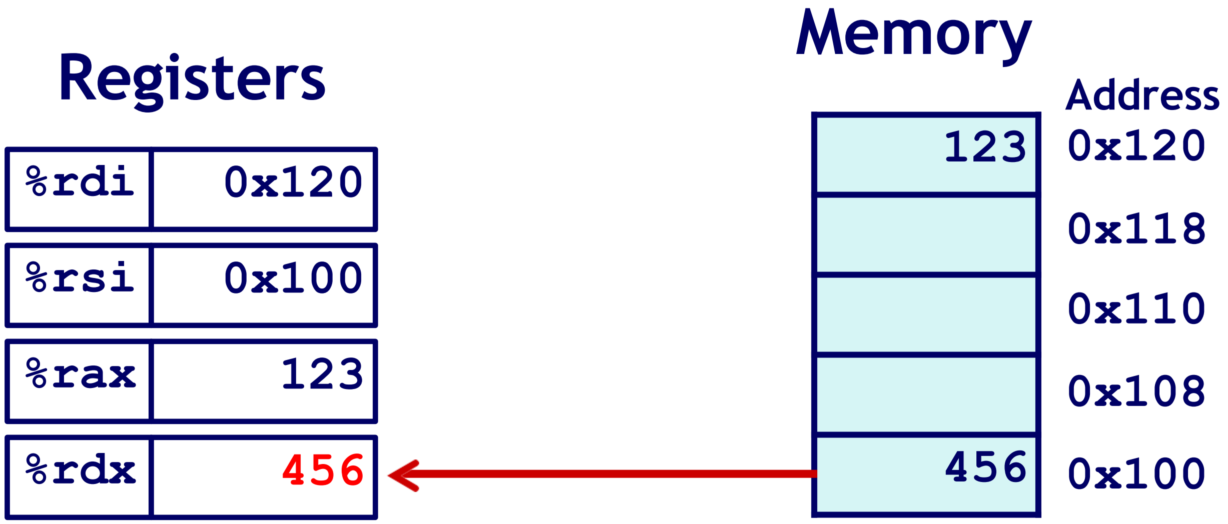
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



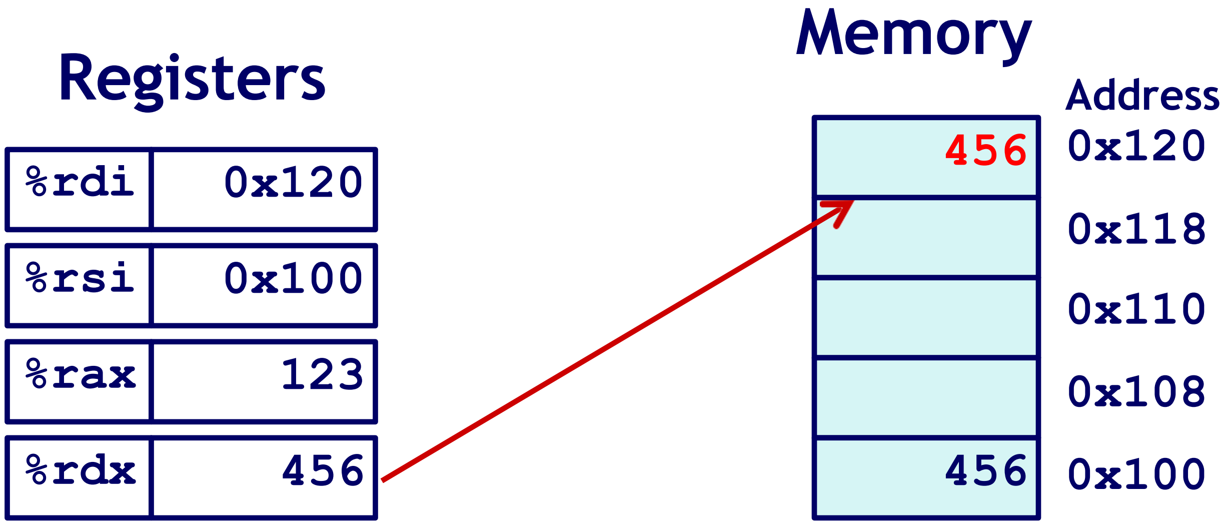
```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



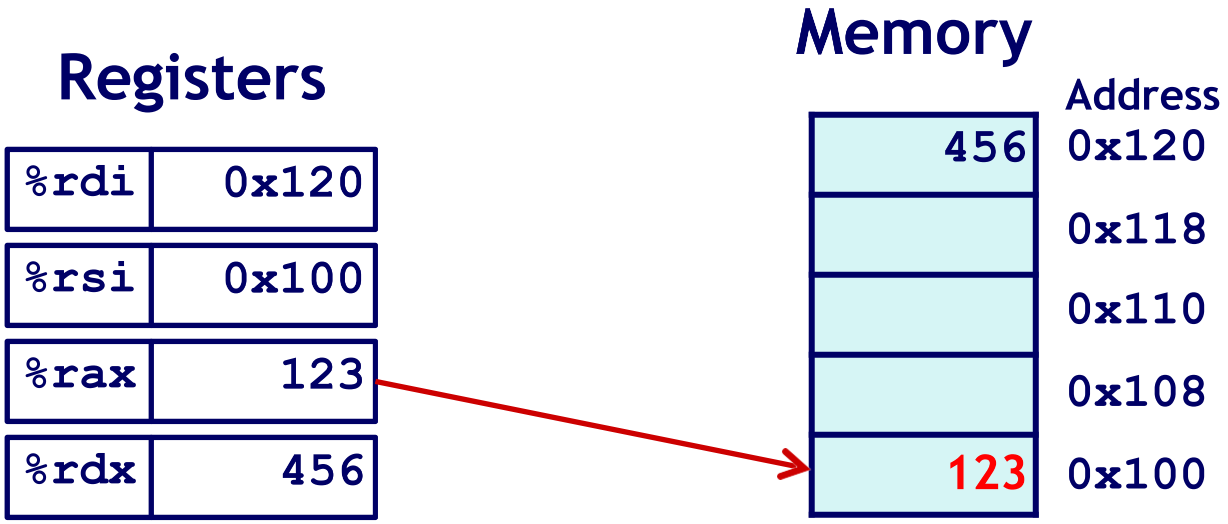
```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()



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

Understanding Swap()



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

Recap: Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Example
    - `movq (%rcx),%rax`
- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
  - Example:
    - `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

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

What is the Address?

`%rdx`	`0xf000`
`%rcx`	`0x0100`

- What is the Address of 0x80 (%rdx, %rcx, 8)?
    - A: 0x0f88
    - B: 0xf880
    - C: 0xf188
    - D: 0xf088
    - E: 0xf480

What is the Address?

`%rdx`	`0xf000`
`%rcx`	`0x0100`

- What is the Address of 0x80 (%rdx, %rcx, 8)?
    - A: 0x0f88
    - B: 0xf880
    - C: 0xf188
    - D: 0xf088
    - E: 0xf480

Address Computation Instruction (LEAQ)

- `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, or 8
    - Example:

Register	Value
`%eax`	`x`
`%ecx`	`y`

Instruction	Result
`leaq 6(%eax),%edx`	`6 + x`
`leaq (%eax,%ecx), %edx`	`x + y`
`leaq (%eax,%ecx,4), %edx`	`x + 4y`
`leaq 7(%eax,%eax,8),`	`7 + 9x`
`leaq 0xA (,%ecx,4), %edx`	`10 + 4y`
`leaq 9(%eax,%ecx,2), %edx`	`9 + x + 2y`

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	

- 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

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

    Note: 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`**
`%rax`	`t1`, `t2`, `rval`
`%rdx`	`t4`
`%rcx`	`t5`