Lecture 6:
Data Representation Cont.
and Intro to Assembly

Announcements

- Project 1 due Monday
  - Due July 13th 11:55pm
- Project 2 will be released shortly after
- Additional TA: Abu Shoeb
  - Contact: as2352@scarletmail.rutgers.edu
  - Office Hours: Wednesdays 11am-12pm
- Midterm Exam
  - July 22nd, Two weeks from now
  - Please let me know if you happen to be in a different timezone.
  - Will use put out sample ProctorTrack Onboarding
- Recitation Today:
  - Questions on data representation
  - Questions on assembly
  - Questions on Project 1

Data Representation Cont.

Bit Patterns from N Bits

Number of Bits	Number of Patterns	Number of Patterns as Power of Two
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4

- Number of possible patterns with N bits $= 2^N$
- How many patterns can be formed with
  - 10 bits? $= 2^{10} = 1024$
  - 20 bits? $= 2^{20} = 2^{10} * 2^{10} = 1048576$
  - 30 bits? $= 2^{30} = 2^{10} * 2^{20} = 1073741824$
  - 40 bits? $= 2^{40} = 2^{10} * 2^{30} = 1.0995116e+12$
  - 50 bits? $= 2^{50} = 2^{10} * 2^{40} = 1.1258999e+15$
  - 60 bits? $= 2^{60} = 2^{10} * 2^{50} = 1.1529215e+18$

Unsigned Integers Overview

- All bits represent magnitude

0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1

magnitude

- Can represent range $[0, 2^n - 1]$
- What range of values can be represented for a 8-bit unsigned integer?
  - $[0, 2^8-1]$
  - [0, 255]
- What ranges of values can be represented by an 32-bit unsigned int?
  - $[0, 2^{32}-1]$
  - [0, 4294967296]

Unsigned Integer to Decimal

- Convert unsigned integer to decimal
- Binary number written as $d_{n-1} \dots d_2 d_1 d_0$ (where n = # of bits)
- The decimal value is $\sum_{i=0}^{n-1} d_i \times 2^i$
- Example:
  - 8-bit unsigned integer

Bits:	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
Indexes:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

  - $= 1(2^7) + 0(2^6) + 0(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)$
  - $= 2^7 + 2^4 + 2^2 + 2^0$
  - $= 128 + 16 + 4 + 1$
  - $= 149$

Signed Integer Overview

- Use the leftmost bit for sign

0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1

sign magnitude

- Use twos complement to represents negative numbers
  - Take the ones complement and add one
  - Essentially invert the bits and add one
- Can represent the range $[-2^{n-1}, 2^{n-1}-1]$
- What range of values can an 8-bit signed integer represent?
  - $[-2^{8-1}, 2^{8-1}-1]$
  - [-128, 127]
- What range of values can an 32-bit signed integer represent?
  - $[-2^{32-1}, 2^{32-1}-1]$
  - [-2147483648, 2147483647]

Signed Integer to Decimal

- Convert Signed Integer to Decimal
- Binary number written as $d_{n-1}d_{n-2} \ldots d_1 d_0$ (where n = # of bits)
- Decimal value is interpreted as $-d_{n-1}2^{n-1} + \sum_{i=0}^{n-2} d_i 2^i$
  - Works with both positive and negative numbers
- Example 1:
  - 8-bit signed integer

Bits:	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
Indexes:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

  - $= -(1 \times 2^7) + 0(2^6) + 0(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)$
  - $= -(1 \times 2^7) + 1(2^4) + 1(2^2) + 1(2^0)$
  - $= -128 + 16 + 4 + 1$
  - $= -107$

Signed Integer to Decimal (Ex. Cont.)

- Let's confirm by taking taking the negative value of -107 and reevaluating decimal
- Negate -107 using twos complement
  - $-107_{10} = 10010101_2$
  - 01101010_2 (take complement)
  - 01101011_2 (add 1)
- Convert 01101011_2 to decimal
  - If right, it should be 107

Bits:	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**
Indexes:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

  - $= -(0 \times 2^7) + 1(2^6) + 1(2^5) + 0(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0)$
  - $= 2^6 + 2^5 + 2^3 + 2^1 + 2^0$
  - $= 64 + 32 + 8 + 2 + 1$
  - $= 107$ (correct!)

Floating Point Overview

- Most computers follow IEEE 754 standard
- Bits split up into three sections:

s	exp	mantissa

  - s: sign field determines if the number is negative (s=1 if negative)
  - exp: biased exponent
  - mantissa: fractional number in binary (base 2)
- Decimal Value = $(-1)^S \times 2^E \times F$
  - E : unbiased exponent in decimal
    - E = exp – bias (where k = number exp bits)
    - bias = $(2^{(k-1)}-1)$
    - The bias allows exp to be represented as an unsigned integer for comparison but represent negative exponents
  - F : binary scientific notation
    - F = 1.<mantissa> (or 0.<mantissa>, we'll see later on)

Converting Floating Point to Decimal

- Recall: Decimal Value = $(-1)^S \times 2^E \times F$
- Basic Steps for converting floating point to decimal
  1. Calculate Unbiased Exponent
     - Get E, where E = exp – bias and bias = $2^{(k-1)}-1$
  2. Get binary scientific notation with mantissa
     - Get F, where F = 1.<mantissa>
  3. Shift binary scientific notation ($2^E \times F$)
  4. Convert binary representation to decimal
  5. Tack on sign (multiply by $(-1)^S$)

Example

- Recall: Decimal Value = $(-1)^S \times 2^E \times F$
- Example: 8-bit floating point
  - 1 bit for sign, 3 bits for exponent, 4 bits for mantissa

sign	exp			mantissa			
1	1	1	0	1	0	1	1

1. Calculate unbiased exponent (E, where E = exp - bias)
   - E = exp – bias
   - $E = 110_2 - \text{bias} = 6_{10} - \text{bias}$ (evaluate exp)
   - $E = 6_{10} - (2^{(k-1)}-1) = 6_{10} - (2^{(3-1)}-1) = 6_{10} - 3_{10}$ (evaluate bias)
   - E = 3
2. Get binary scientific notation
   - F = 1.<mantissa> = 1.1011
3. Shift Binary Representation ($2^E \times F$)
   - $2^3 \times 1.1011_2 = 1101.1_2$
4. Evaluate Binary Result To Decimal
5. Tack on Sign (multiply by $(-1)^S$)

1	1	0	1	.	1
3	2	1	0		-1

$= 1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) + 1(2^{-1})$

$= 2^3 + 2^2 + 2^0 + 2^{-1}$

$= 8 + 4 + 1 + 0.5$

$= 13.5$

$= -13.5$ Final Result

Other Values in Floating Point

- We just went over how normalized values are represented in floating
- However two additional kinds of values are represented by floating point representation
  - How we interpret them is different than normalized values
- Denormal Values
  - When exp is all 0s
  - Represents numbers 0 or very close to zero
  - Difference from normalized values:
    - Different Unbiased Exponent (E) = 1 – bias or $1 - (2^{(k-1)}-1)$
    - Different Binary Scientific Notation (F) = 0.<mantissa>
- Special Values
  - When exp all 1s
  - When mantissa is all 0's
    - Positive or negative Infinity ($\pm\infty$) depending on sign
  - When mantissa is not all 0's
    - NaN = Not a number

Denormal Value Example

- Recall: Decimal Value = $(-1)^S \times 2^E \times F$
- Example: 8-bit floating point
  - 1 bit for sign, 3 bits for exponent, 4 bits for mantissa

0	0	0	0	1	0	1	1
sign	exp			mantissa			

1. Calculate unbiased exponent (E, where E = 1 - bias)
   - E = 1 – bias
   - $E = 1 - (2^{(k-1)}-1) = 1 - (2^{(3-1)}-1) = 1 - 3$ (evaluate bias)
   - E = -2
2. Get binary scientific notation
   - $F = 0.\text{<mantissa>} = 0.1011_2$
3. Shift Binary Representation ($2^E \times F$)
   - $2^{-2} \times 0.1011_2 = 0.001011_2$
4. Evaluate Binary Result To Decimal
5. Tack on Sign (multiply by $(-1)^S$)

0	.	0	0	1	0	1	1
0		-1	-2	-3	-4	-5	-6

$= 0(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 0(2^{-4}) + 1(2^{-5}) + 1(2^{-6})$

$= 2^{-3} + 2^{-5} + 2^{-6}$

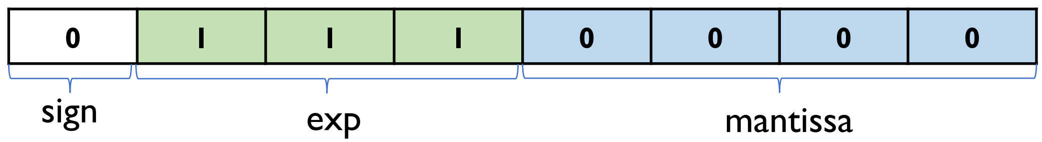
$= 0.125 + 0.03125 + 0.015625$

$= 0.171875$

$= +0.171875$ ← Final Result

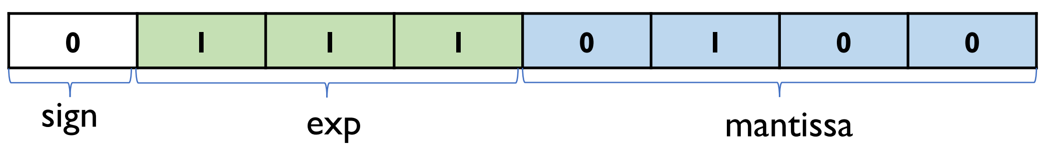
Special Value Examples

- Example 1:



  - exp is all 1s so it must be a special value
  - mantissa is all 0s and the sign is 0 so positive
  - special value + 0 mantissa + positive value = $+\infty$

- Example 2:



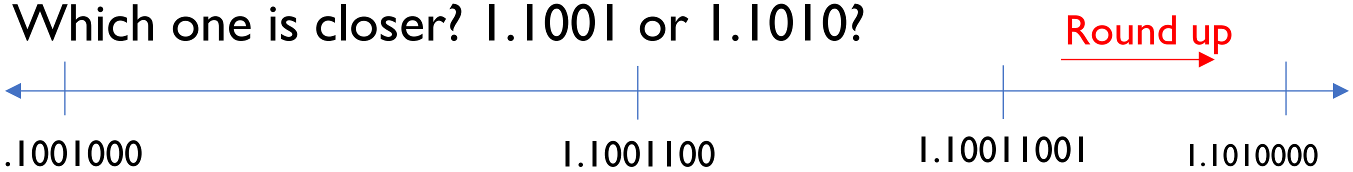
  - exp is all 1s so it must be a special value
  - mantissa is not all zeros
  - special value + non-zero mantissa = NaN

Floating Point Summary

- Three different cases
- Normalized values
  - When exp is not all 0s or not all 1s
  - $E = exp - (2^{(k-1)}-1)$
  - F = 1.<mantissa>
- Denormalized Values
  - When exp is 0
  - $E = 1 - (2^{(k-1)}-1)$ -> (e.g for 32-bit float: $1- 127 = -126$)
  - F = 0.<mantissa>
  - Represents 0 and values very close to 0
- Special Values
  - When exp all 1's
  - When mantissa is all 0's
    - Positive or negative Infinity ($\pm\infty$) depending on sign
  - Else when mantissa is not all 0's
    - NaN = Not a number

Rounding in Floating Point

- Round to the nearest number
- Example:
  - Assume 4 bit mantissa
  - 1.10011001
  - Need to trauncate to 4 mantissa bits
  - Which one is closer? 1.1001 or 1.1010?



  - Round up to 1.1010 because it's closer
- What happens if tie?
  - Round to even binary number (where last digit is 0)
- Example:
  - 1.10011
    - If we round down we get an odd number 1.1001
    - So round up to even number 1.1010
  - 1.10001
    - If we round up we get 1.1001 which is not even
    - Round down to even number 1.1000

ASCII

- American Standard for Computer Information Interchange
  - Defines what character is represents by a sequence of bits
- According to ASCII standard, 1 character is stores with 1 byte (8 bits)
- Based on the English Alphabet
- Originally only encoded 128 character using 7 bits
  - One bit could be used for error detection
- Subsequently extended to use all 256 values

ASCII Table

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Character value stored in 1 byte

Value of character in Hex

- ‘1’ = 0x31
- ‘3’ = 0x33
- ‘9’ = 0x39
- ‘a’ = 0x61
- ‘A’= 0x41

ASCII Character Representing Integer

- Supppose user types a 4 character sequence "123\n"
- Conversion from character representation to the desired two's complement integer representation
  - Integer desired = ASCII representation - 48

ASCII Character	Hex Value	Decimal Value	Binary	Desired Integer	Two's Complement
'1'	0x31	49	00110001	1	00000001
'2'	0x32	50	00110010	2	00000010
'3'	0x33	51	00110011	3	00000011
'\n'	0x01	10	00001010	(NA)	(NA)

Unicode and UTF-8

- What about characters for other languages?
  - ASCII only allows for a small number of characters
- Unicode is a standard that defines more than 107,000 characters across 90 scripts (and more)
- Most Common: UTF-8
  - Variable length encoding of Unicode: 1-4 bytes for each character
  - 1-byte form is reserved for ASCII backward compatibility

Addressing

- All information is represented in binary form but require different sizes
- Pointer sizes are different depending on the architecture:
  - 32-bit machine: 32-bit pointer = 4 bytes
  - 63-bit machine: 64-bit pointer = 8 bytes
- How many different addresses can a pointer have?
  - 32-bits = 2^{32} bytes = 2^{2}x2^{30} bytes = 4 Gigabytes
  - 64-bits = 2^{64} bytes = 2^{4}x2^{60} bytes = 16 Exabytes
- This is what known as the "Address Space" or space of all memory address

Big Endian vs. Little Endian

- How to determine value when you have a binary number spread across multiple bytes?

A0	**BC**	**00**	**12**

- Is it A0BC0012 or 1200BCA0?
- Big Endian
  - Most significant byte first
  - A0BC0012 in example above
- Little Endian
  - Least significant byte first
  - 1200BCA0 in example above
- Why care?
  - Interpret machine code and values
  - Different computers use different endianness
  - Need to convert into standard form before transmitting

Data in Memory

Integer: 0xA0BC0012

	...
0x100	A0
0x101	BC
0x102	00
0x103	12
	...

Big Endian

	...
0x100	12
0x101	00
0x102	BC
0x103	A0
	...

Little Endian

Intro To Assembly:

Topics:

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

Programming Meets Hardware

High-Level Language Program

```c
#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);
}
```

Compiler

Assembly Language Program

```
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)
```

ISA

Assembler

Machine Language Program

```
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

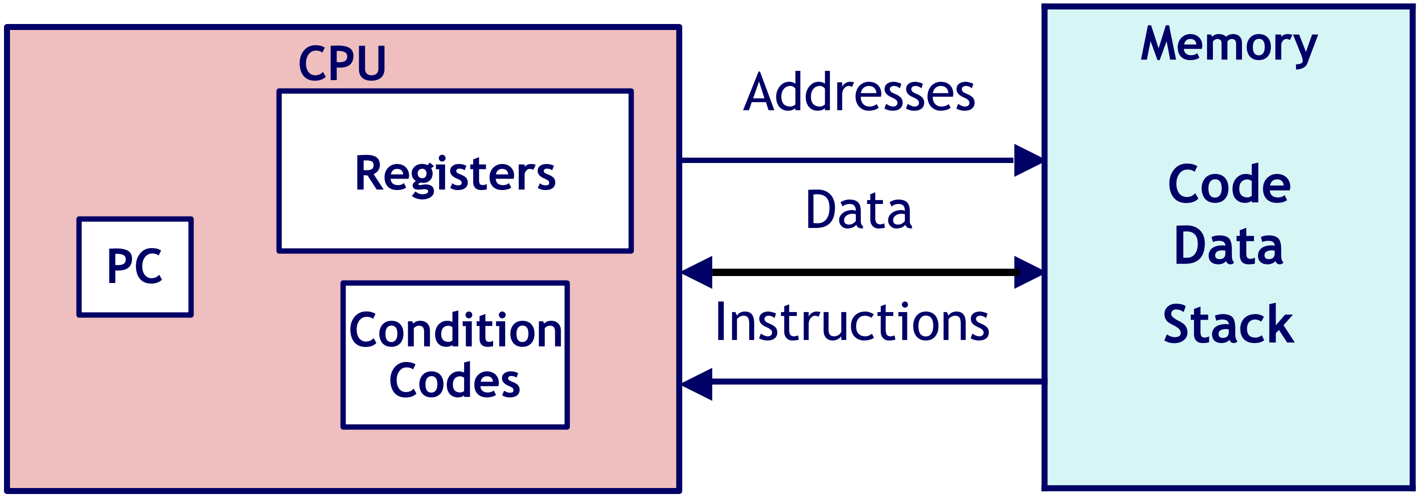
Assembly Programming

- 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



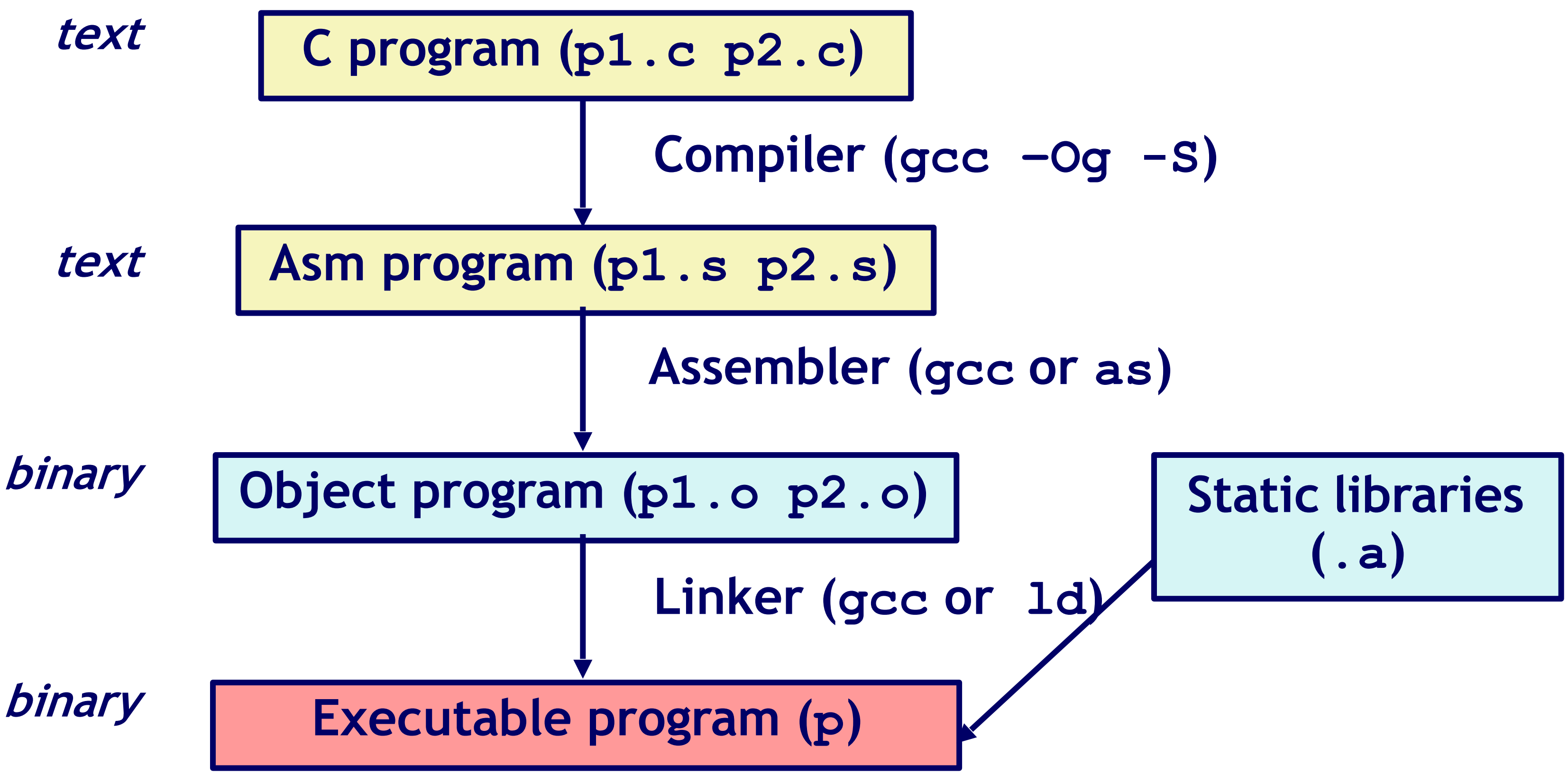
Programmer-Visible State

- **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

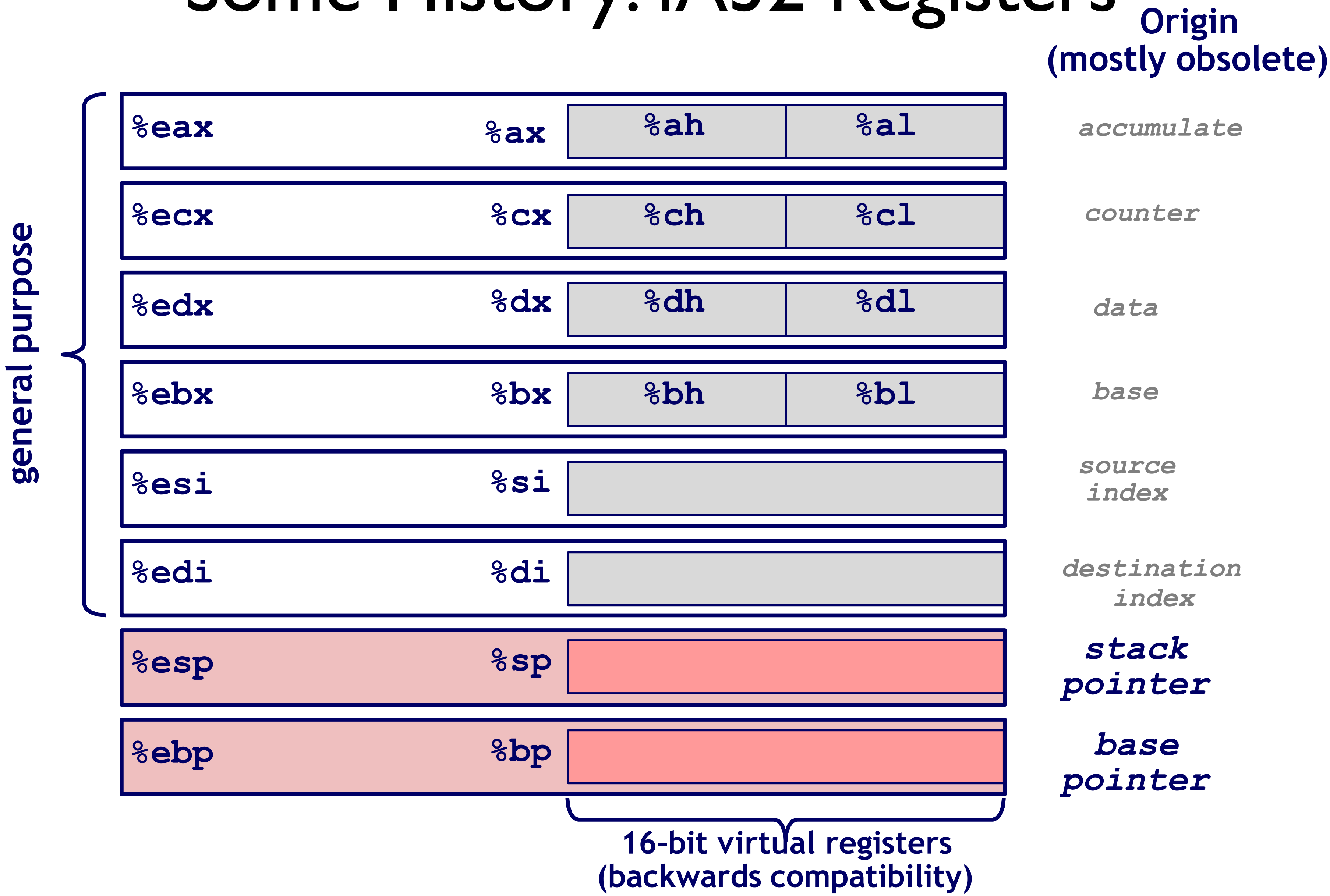
```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push    %rbx
 0x0000000000400596 <+1>: mov     %rdx,%rbx
 0x0000000000400599 <+4>: callq   0x400590 <plus>
 0x000000000040059e <+9>: mov     %rax,(%rbx)
 0x00000000004005a1 <+12>:pop     %rbx
 0x00000000004005a2 <+13>:retq
```

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`

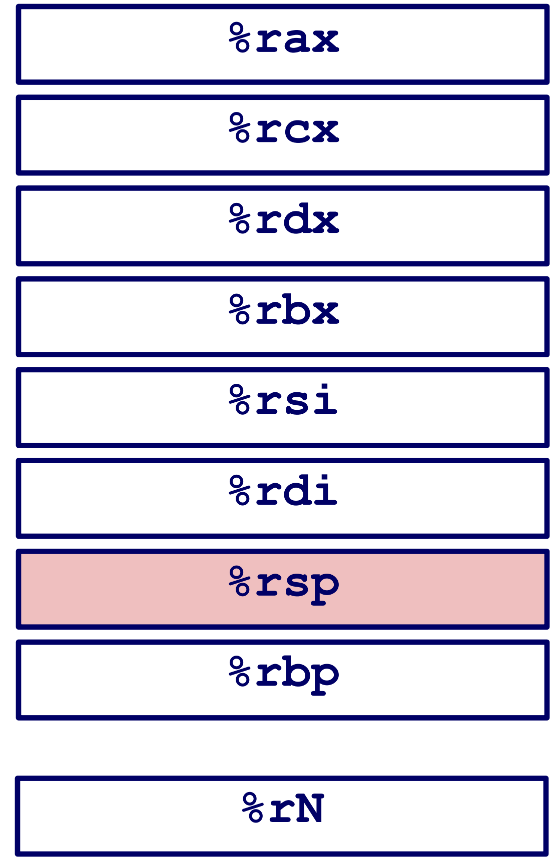
- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

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`

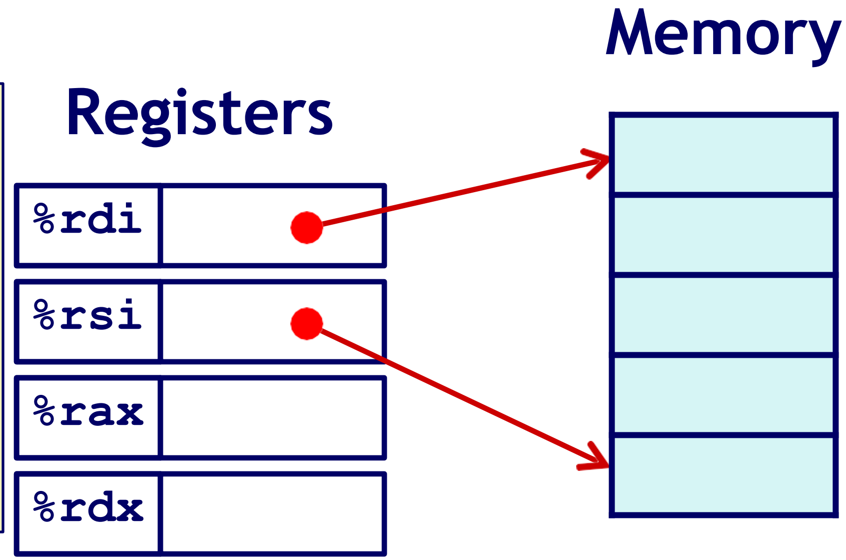
Example of Simple Addressing Modes

```
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()

Registers

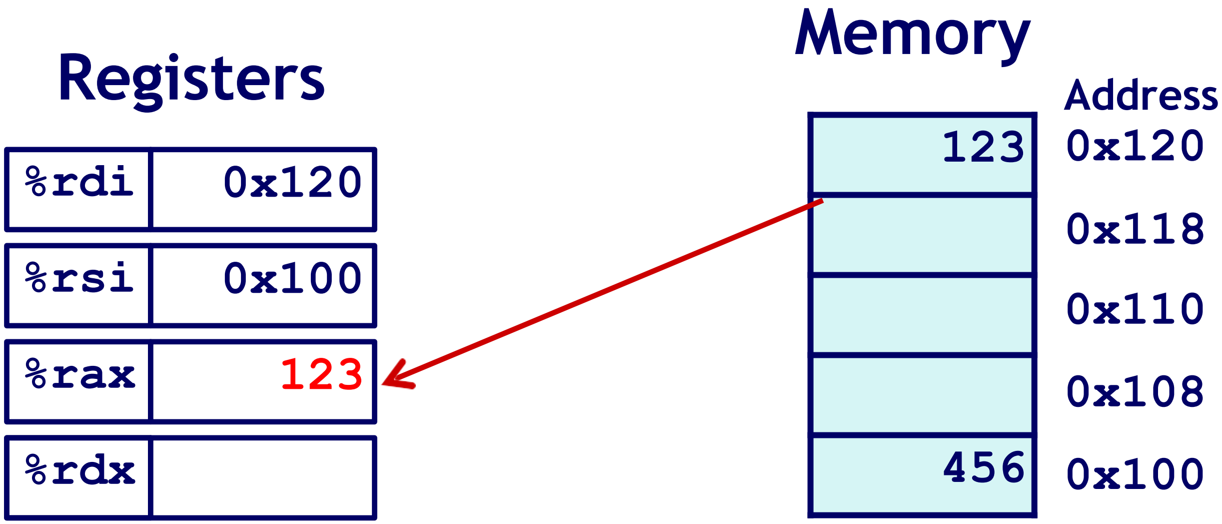
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

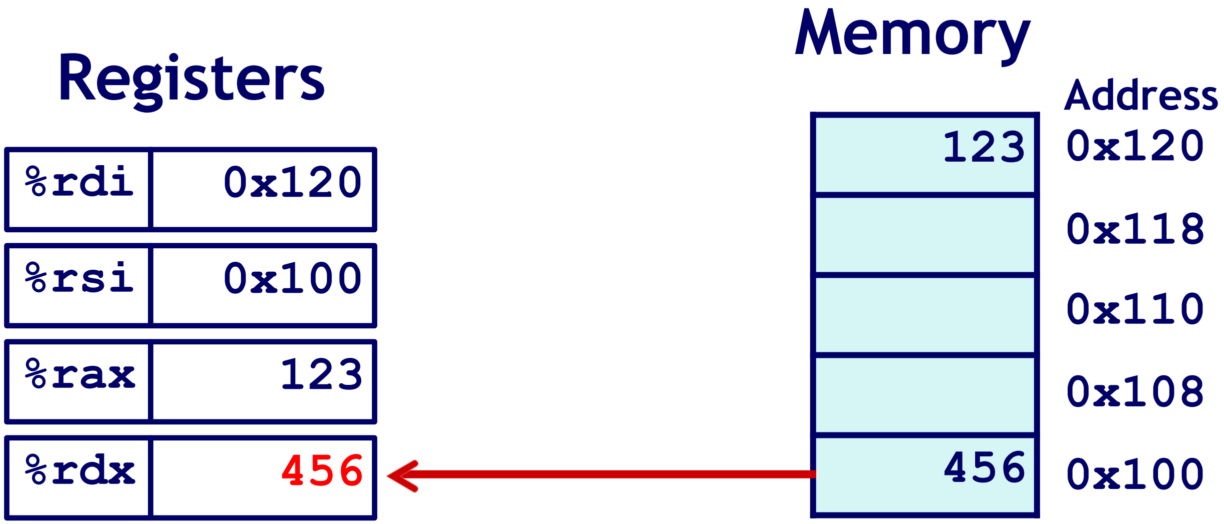
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



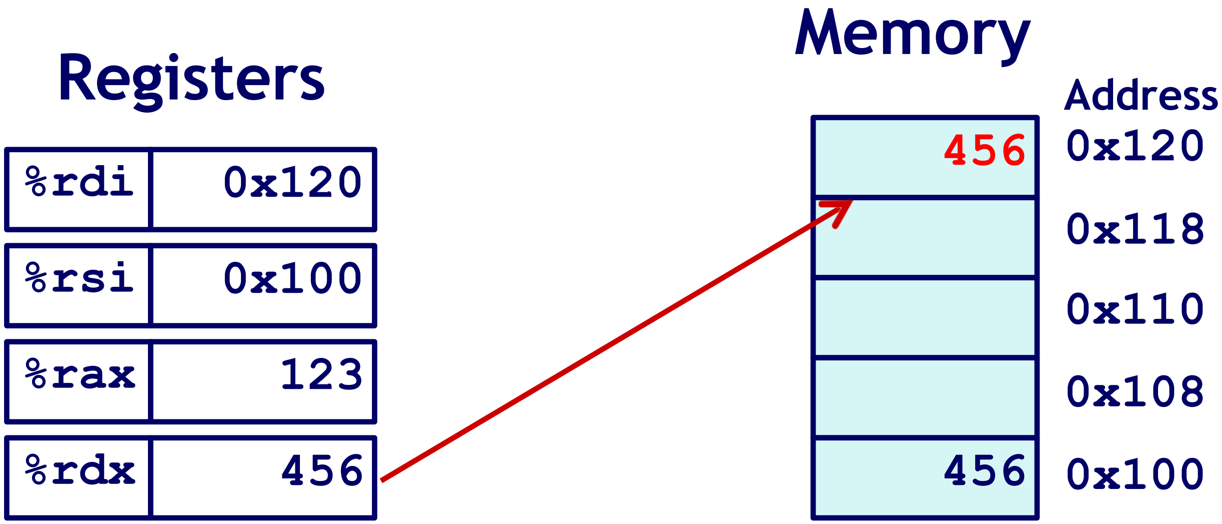

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



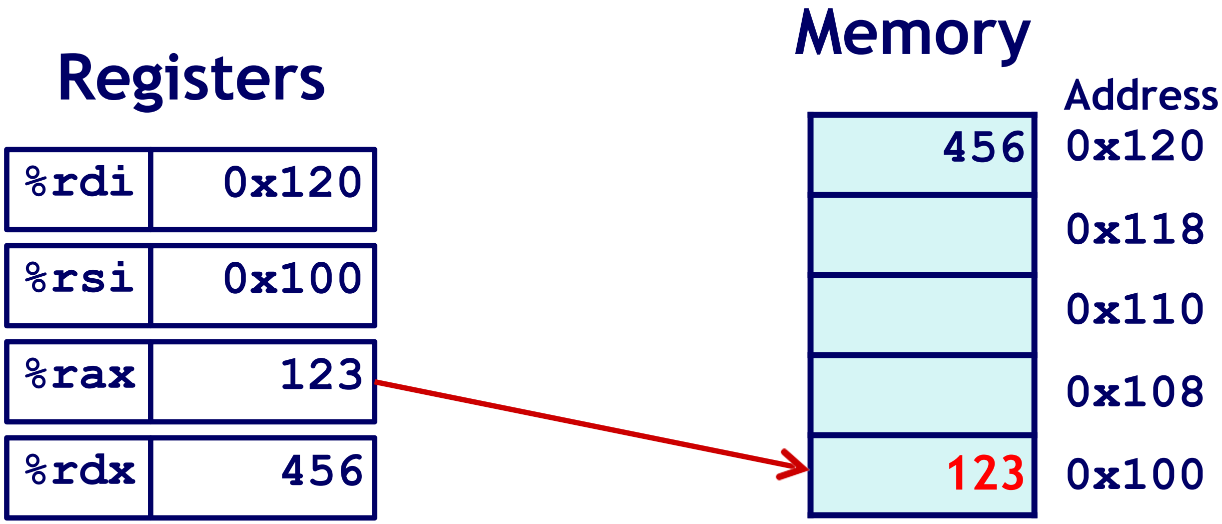

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

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

```
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
```

```
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;
}
```

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