x86 Data Access and Operations

Machine-Level Representations

Prior lectures

- Data representation

This lecture

- Program representation
- Encoding is architecture dependent
  - We will focus on the Intel x86-64 or x64 architecture
  - Prior edition used IA32

Intel x86

Evolutionary design starting in 1978 with 8086

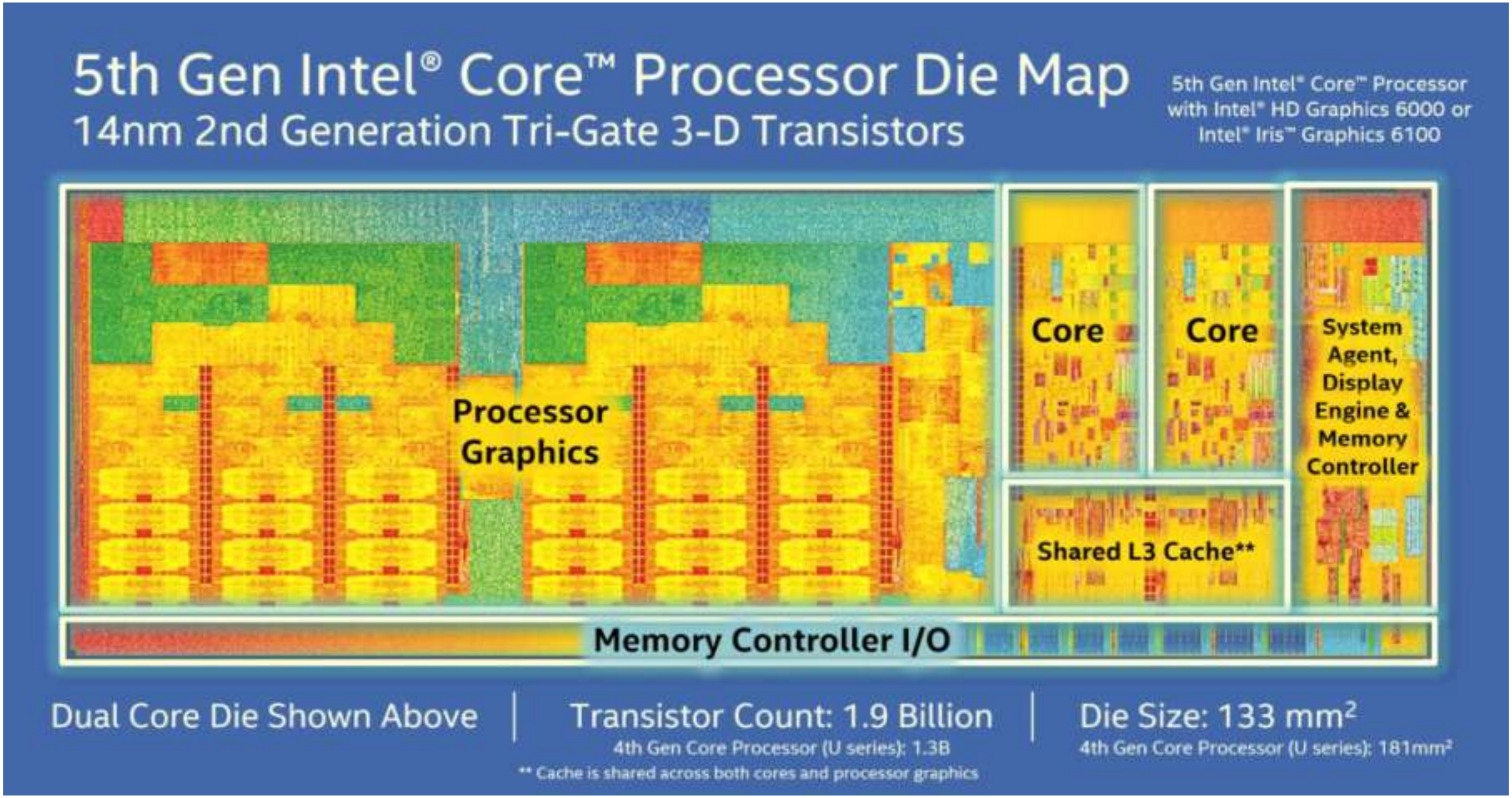
- i386 in 1986: First 32-bit Intel CPU (IA32)
- Pentium4E in 2004: First 64-bit Intel CPU (x86-64)
  - Adopted from AMD Opteron (2003)
- Core 2 in 2006: First multi-core Intel CPU
- Core 7 in 2008: Current generation
- New features and instructions added over time
  - Vector operations for multimedia
  - Memory protection for security
  - Conditional data movement instructions for performance
  - Expanded address space for scaling
- Many obsolete features

Complex Instruction Set Computer (CISC)

- Many different instructions with many different formats
- But we’ll only look at a small subset

2015

Core i7 Broadwell



5th Gen Intel® Core™ Processor Die Map
14nm 2nd Generation Tri-Gate 3-D Transistors

5th Gen Intel® Core™ Processor with Intel® HD Graphics 6000 or Intel® Iris™ Graphics 6100

Dual Core Die Shown Above | Transistor Count: 1.9 Billion | Die Size: 133 mm^2

4th Gen Core Processor (U series): 1.3B

4th Gen Core Processor (U series): 181mm^2

** Cache is shared across both cores and processor graphics

How do you program it?

Initially, no compilers or assemblers

Machine code generated by hand!

- Error-prone
- Time-consuming
- Hard to read and write
- Hard to debug

Assemblers

Assign mnemonics to machine code

- Assembly language for specifying machine instructions
- Names for the machine instructions and registers
  - `movq %rax, %rcx`
- There is no standard for x86 assemblers
  - Intel assembly language
  - AT&T Unix assembler
  - Microsoft assembler
  - GNU uses Unix style with its assembler `gas`

Even with the advent of compilers, assembly still used

- Early compilers made big, slow code
- Operating Systems were written mostly in assembly, into the 1980s
- Accessing new hardware features before compiler has a chance to incorporate them

Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y, long *D)
{
    long t = plus(x, y);
    *D = t;
}
```

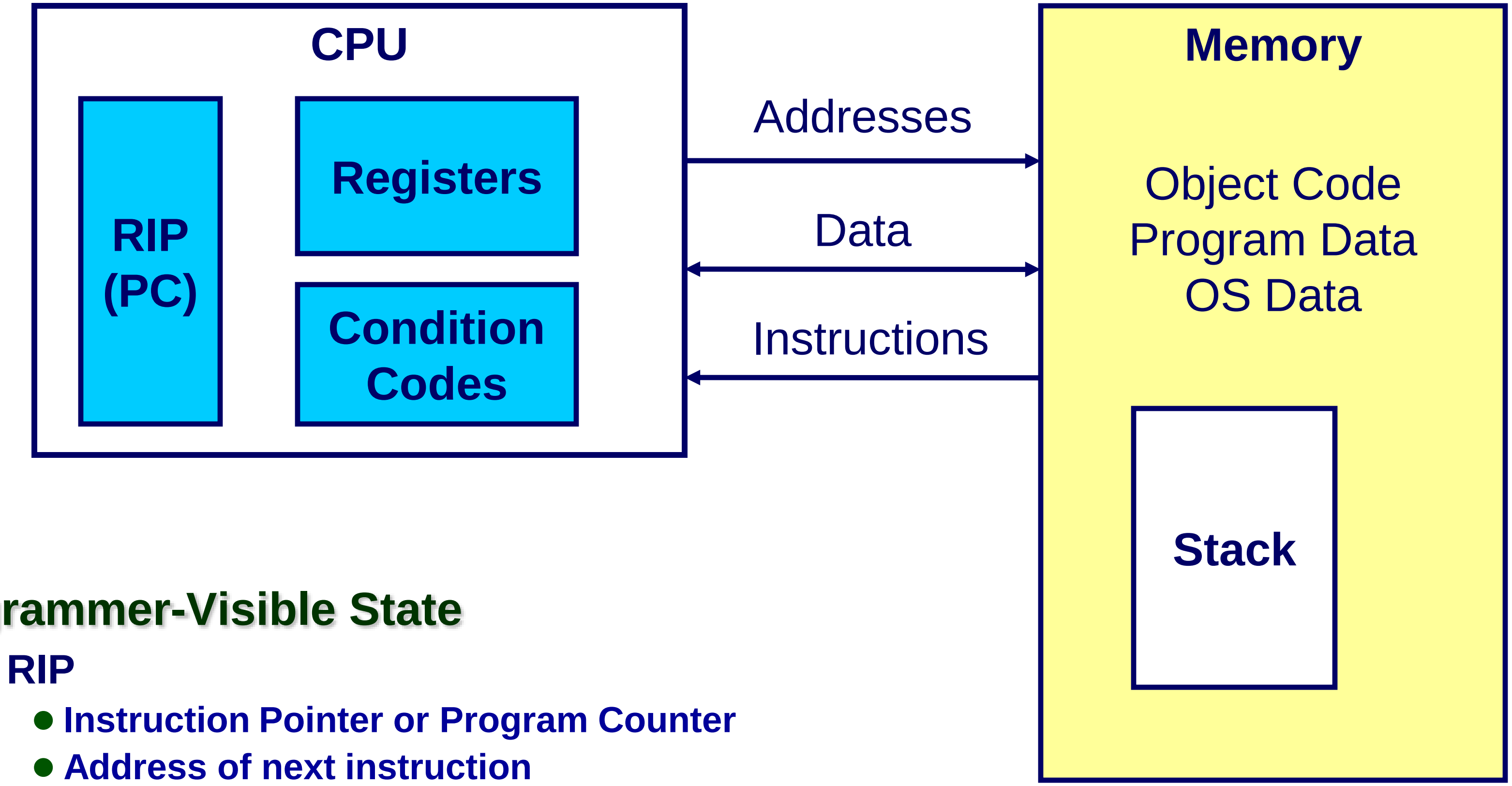
Compiled using basic optimizations (-Og)

```
gcc –Og –S sum.c
```

Generated x86-64 assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Assembly Programmer's View



Programmer-Visible State

- RIP
  - Instruction Pointer or Program Counter
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

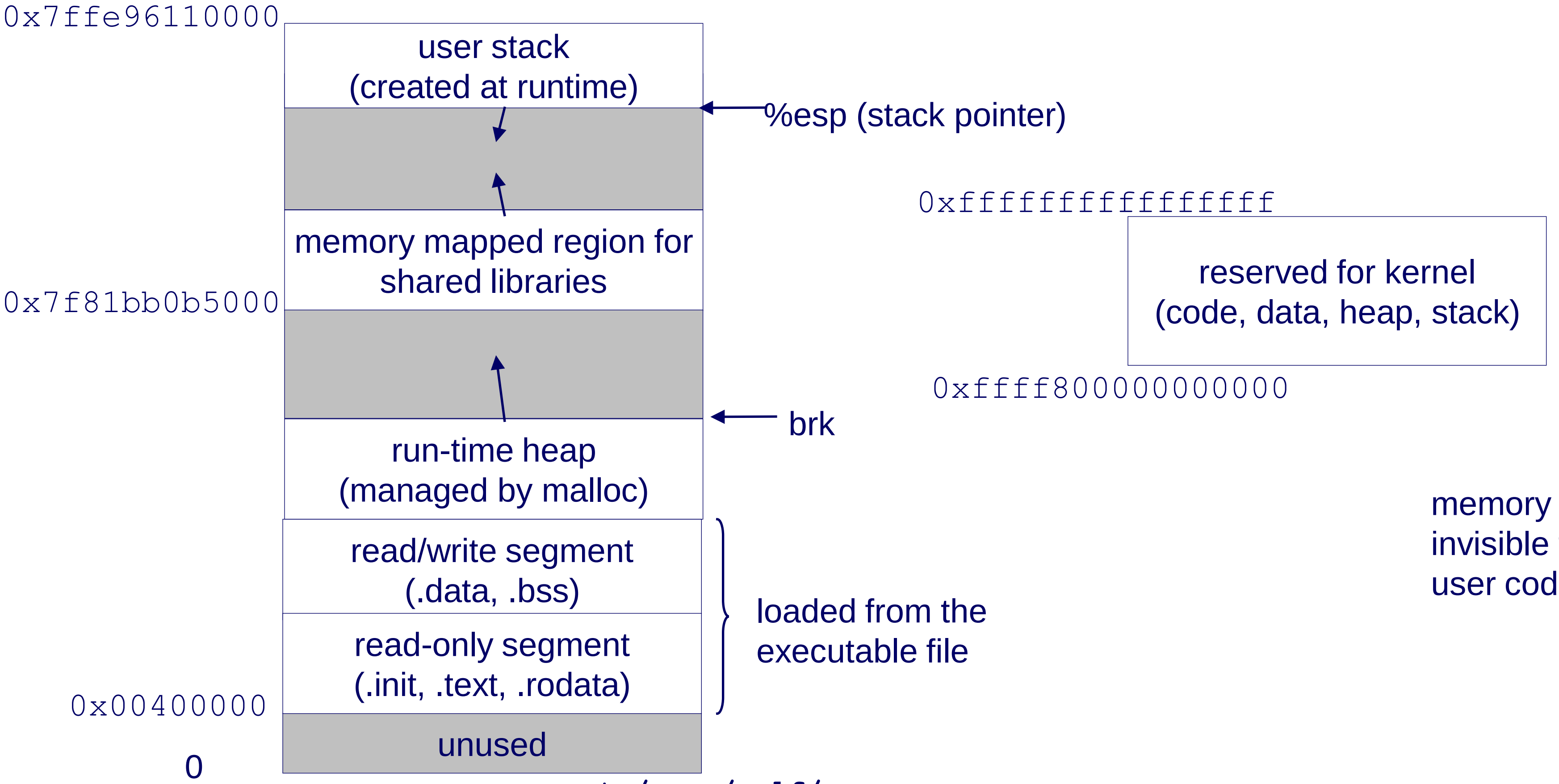
Memory

- Byte addressable array
- Code, user data, OS data
- Includes stack used to support procedures

64-bit memory map

48-bit canonical addresses to make page-tables smaller

Kernel addresses have high-bit set



0x7ffe96110000

user stack
(created at runtime)

%esp (stack pointer)

memory mapped region for
shared libraries

0x7f81bb0b5000

run-time heap
(managed by malloc)

brk

read/write segment
(.data, .bss)

read-only segment
(.init, .text, .rodata)

loaded from the
executable file

0x00400000

unused

0

0xffffffffffffffff

reserved for kernel
(code, data, heap, stack)

0xffff800000000000

memory
invisible to
user code

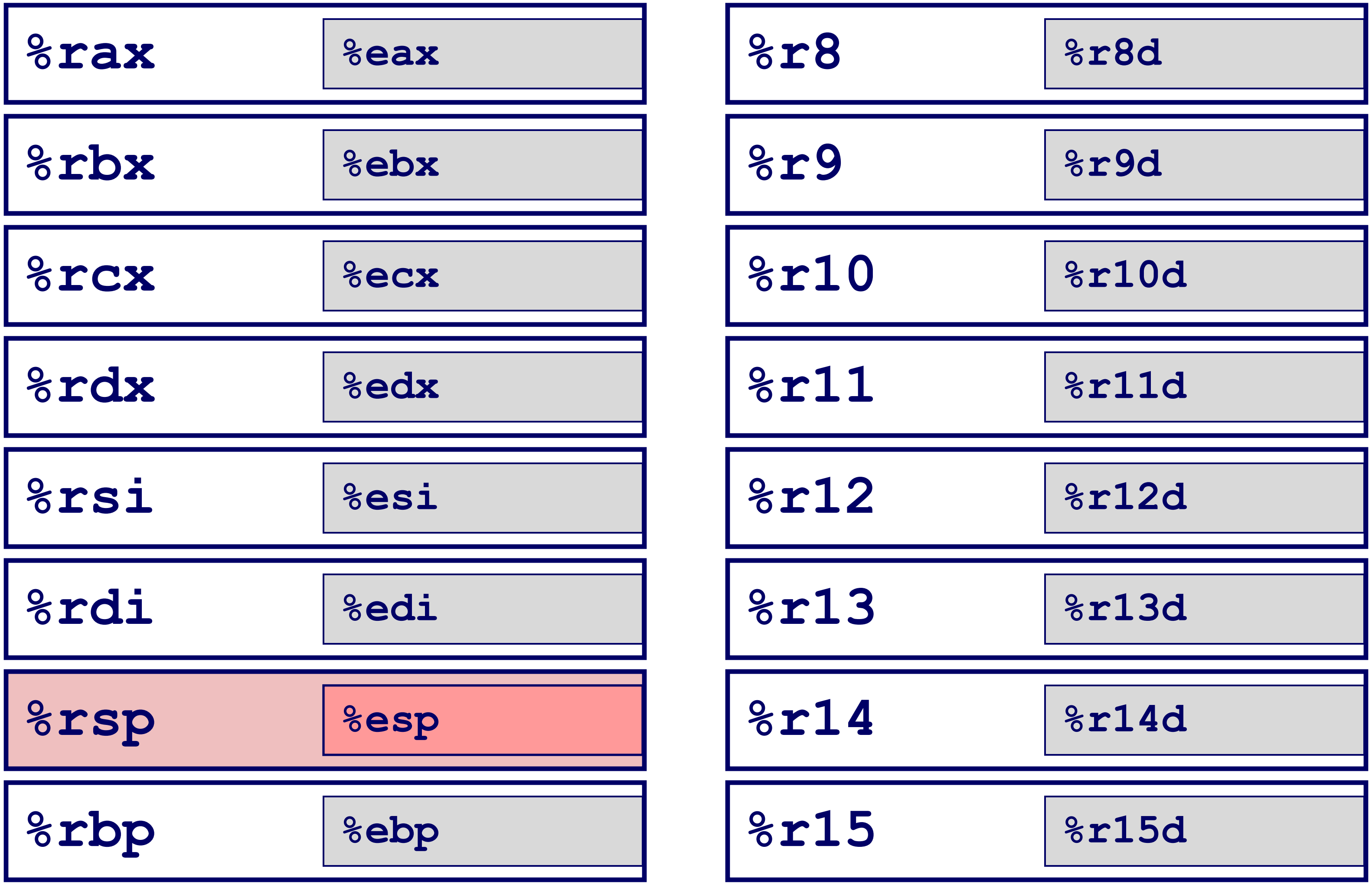
`cat /proc/self/maps`

Registers

Special memory not part of main memory

- Located on CPU
- Used to store temporary values
- Typically, data is loaded into registers, manipulated or used, and then written back to memory

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

- Accessible as 8, 16, 32, 64 bits

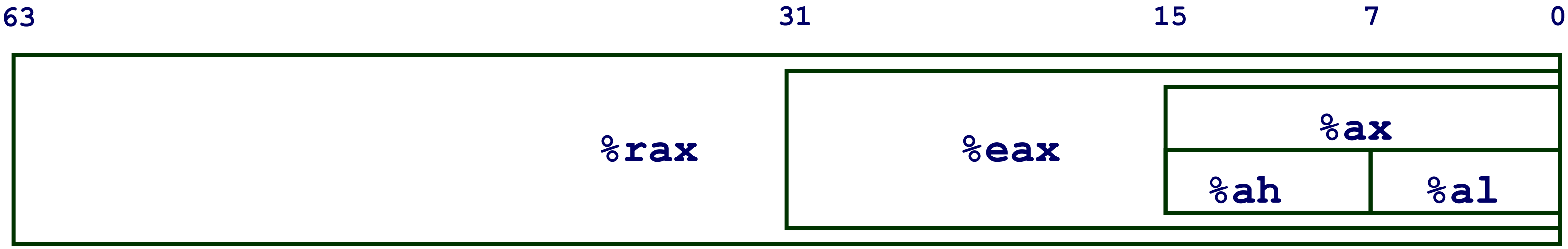
↑

Format different since registers added with x86-64

64-bit registers

Multiple access sizes `%rax, %rbx, %rcx, %rdx`

```
%ah, %al : low order bytes (8 bits)
%ax  : low word (16 bits)
%eax : low “double word” (32 bits)
%rax : quad word (64 bits)
```

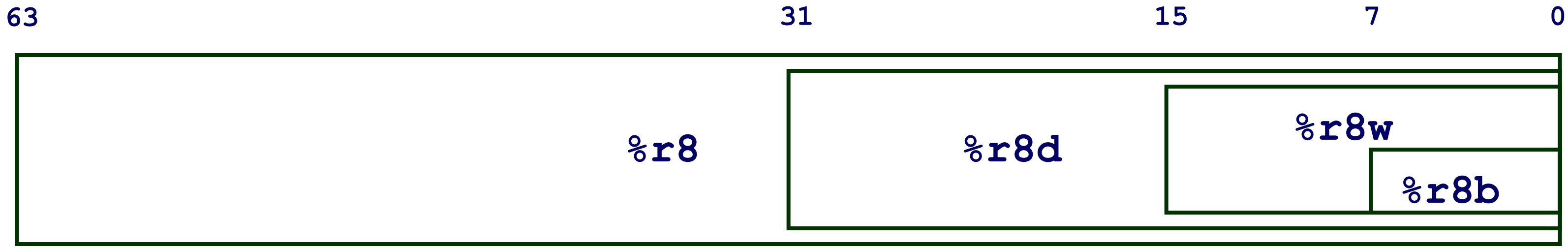


Similar access for `%rdi, %rsi, %rbp, %rsp`

64-bit registers

Multiple access sizes `%r8, %r9, … , %r15`

```
%r8b : low order byte (8 bits)
%r8w : low word (16 bits)
%r8d : low “double word” (32 bits)
%r8  : quad word (64 bits)
```



Register evolution

The x86 architecture initially “register poor”

- Few general purpose registers (8 in IA32)
  - Initially, driven by the fact that transistors were expensive
  - Then, driven by the need for backwards compatability for certain instructions pusha (push all) and popa (pop all) from 80186
- Other reasons
  - Makes context-switching amongst processes easy (less register-state to store)
  - Add fast caches instead of more registers (L1, L2, L3 etc.)

Instruction types

A typical instruction acts on 2 or more *operands* of a particular width

- `addq %rcx, %rdx` adds the contents of `rcx` to `rdx`
- "`addq`" stands for add "quad word"
- Size of the operand denoted in instruction
- Why "quad word" for 64-bit registers?
  - Baggage from 16-bit processors

Now we have these crazy terms

- 8 bits = byte = `addb`
- 16 bits = word = `addw`
- 32 bits = double or long word = `addl`
- 64 bits = quad word = `addq`

C types and x86-64 instructions

C Data Type	Intel x86-64 type	GAS suffix	x86-64
`char`	byte	b	1
`short`	word	w	2
`int`	double word	l	4
`long`	quad word	q	8
`float`	single precision	s	4
`double`	double precision	l	8
`long double`	extended precision	t	10/16
`pointer`	quad word	q	8

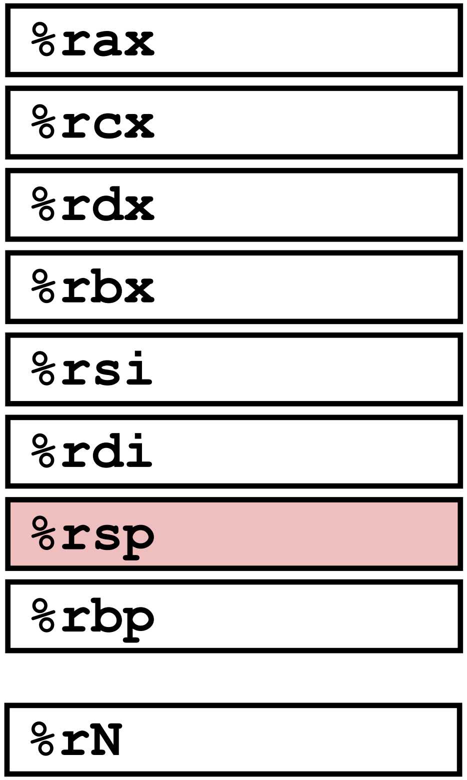
Instruction operands

Example instruction

movq *Source*, *Dest*

Three operand types

- Immediate
  - Constant integer data
  - Like C constant, but preceded by $
  - e.g., $0x400, $-533
  - Encoded directly into instructions
- Register: One of 16 integer registers
  - Example: %rax, %r13
  - Note %rsp reserved for special use
- Memory: a memory address
  - There are many modes for addressing memory
  - Simplest example: (%rax)



%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN

Operand examples using `mov`

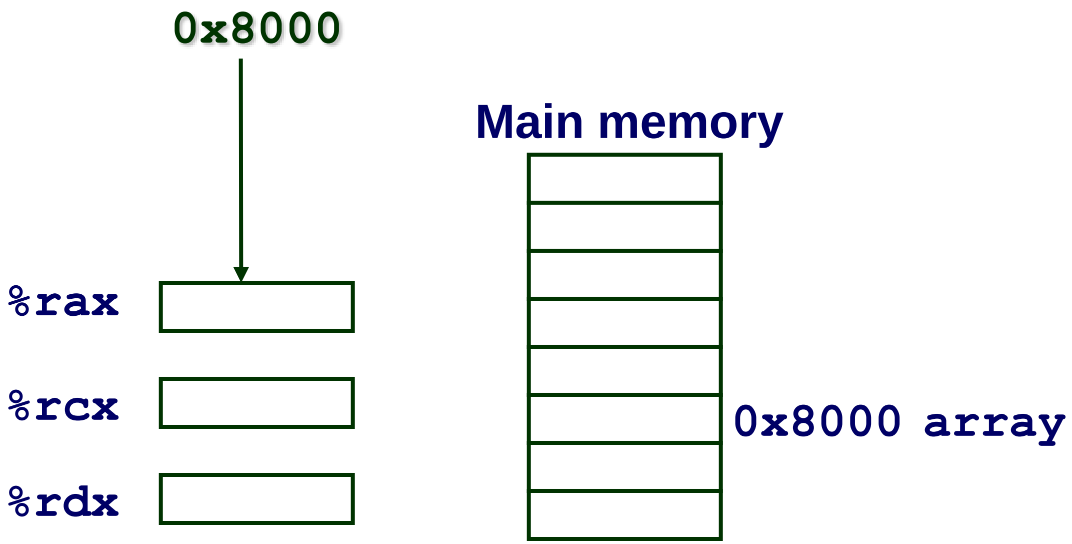
	Source	Destination		C Analog
`movl`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

- **Memory-memory transfers cannot be done with single instruction**

Immediate mode

Immediate has only one mode

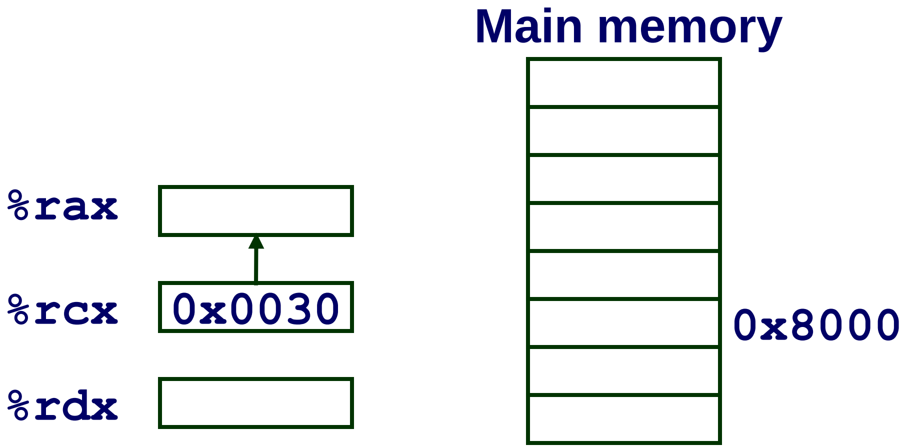
- Form: `$Imm`
- Operand value: `Imm`
  - `movq $0x8000,%rax`
  - `movq $array,%rax`
    - » int array[30]; /* array = global variable stored at 0x8000 */



Register mode

Register has only one mode

- Form: E_a
- Operand value: $R[E_a]$
  - `movq %rcx,%rax`



Memory modes

Memory has multiple modes

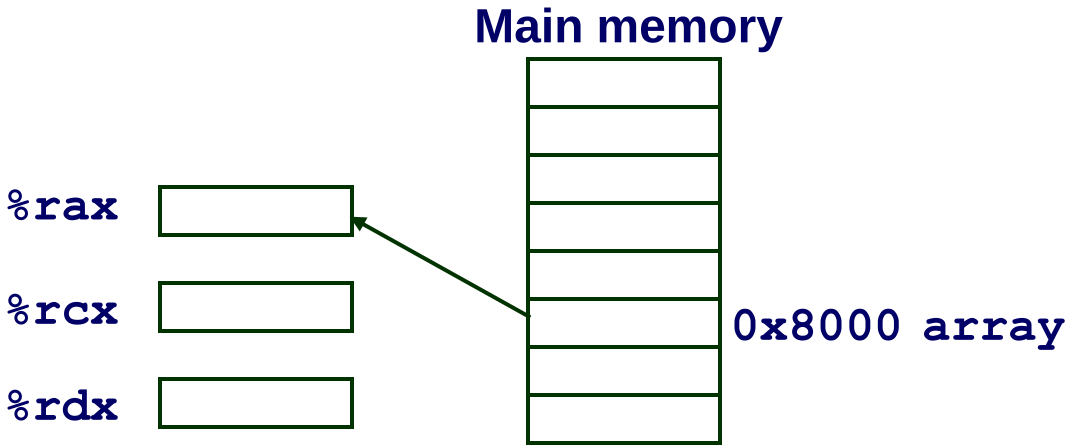
- Absolute
  - specify the address of the data
- Indirect
  - use register to calculate address
- Base + displacement
  - use register plus absolute address to calculate address
- Indexed
  - Indexed
    - » Add contents of an index register
  - Scaled index
    - » Add contents of an index register scaled by a constant

Memory modes

Memory mode: Absolute

- Form: `Imm`
- Operand value: `M[Imm]`
  - `movq 0x8000,%rax`
  - `movq array,%rax`

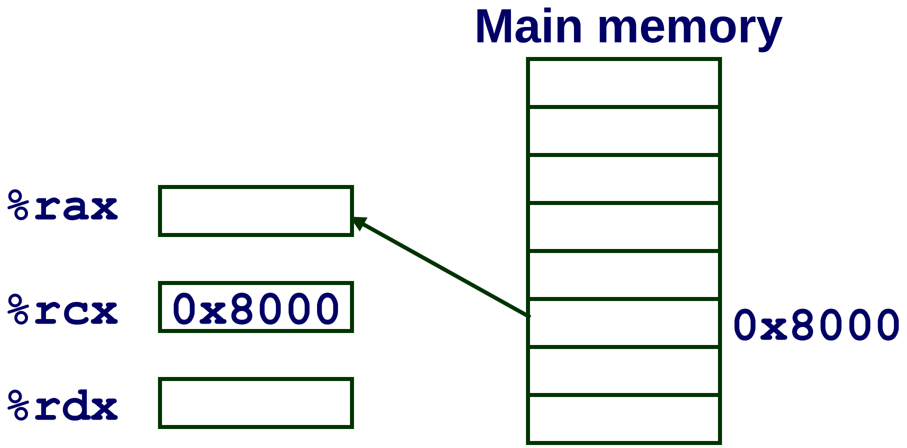
```
long array[30];  /* global variable at 0x8000 */
```



Memory modes

Memory mode: Indirect

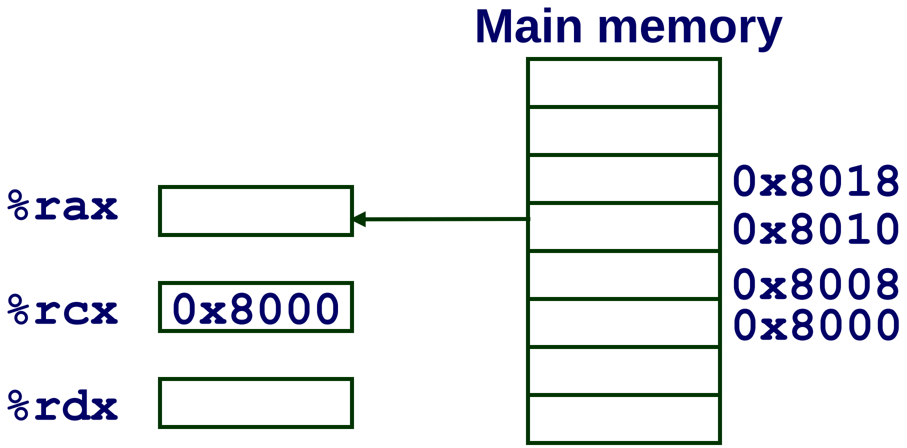
- Form: `(E_a)`
- Operand value: `M[R[E_a]]`
  - Register E_a specifies the memory address
  - `movq (%rcx),%rax`



Memory modes

Memory mode: Base + Displacement

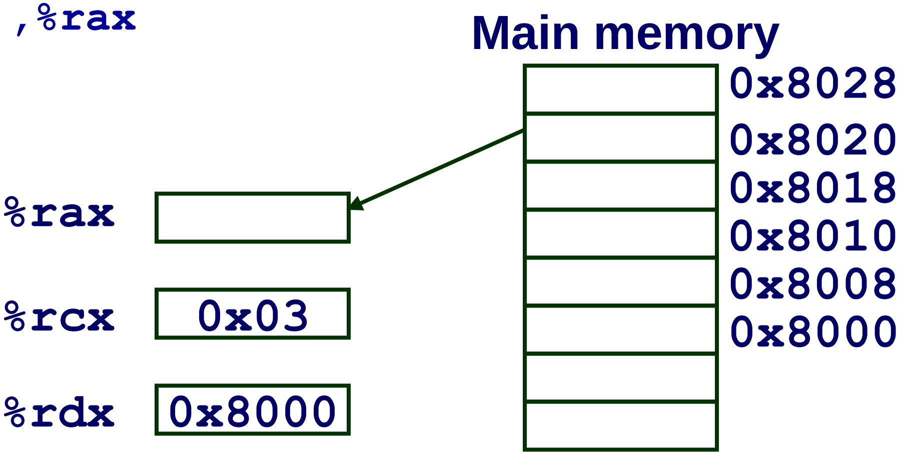
- Form: `Imm(E`$_b$`)`
- Operand value: `M[Imm+R[E`$_b$`]]`
  - Register `E`$_b$ specifies start of memory region
  - `Imm` specifies the offset/displacement
- `movq 16(%rcx),%rax`



Memory modes

Memory mode: Scaled indexed

- Most general format
- Used for accessing structures and arrays in memory
- Form: `Imm(`E_b`,`E_i`,S)`
- Operand value: `M[Imm+R[`E_b`]+S*R[`E_i`]]`
  - Register E_b specifies start of memory region
  - E_i holds index
  - `S` is integer scale (1,2,4,8)
  - `movq 8(%rdx,%rcx,8),%rax`



Addressing Mode Examples

`addl 12(%rbp),%ecx`	Add the double word at address rbp + 12 to ecx
`movb (%rax,%rcx),%dl`	Load the byte at address rax + rcx into dl
`subq %rdx,(%rcx,%rax,8)`	Subtract rdx from the quad word at address rcx+(8*rax)
`incw 0xA(,%rcx,8)`	Increment the word at address 0xA+(8*rcx)

Also note: We do not put '$' in front of constants when they are addressing indexes, only when they are literals

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`

Practice Problem 3.1

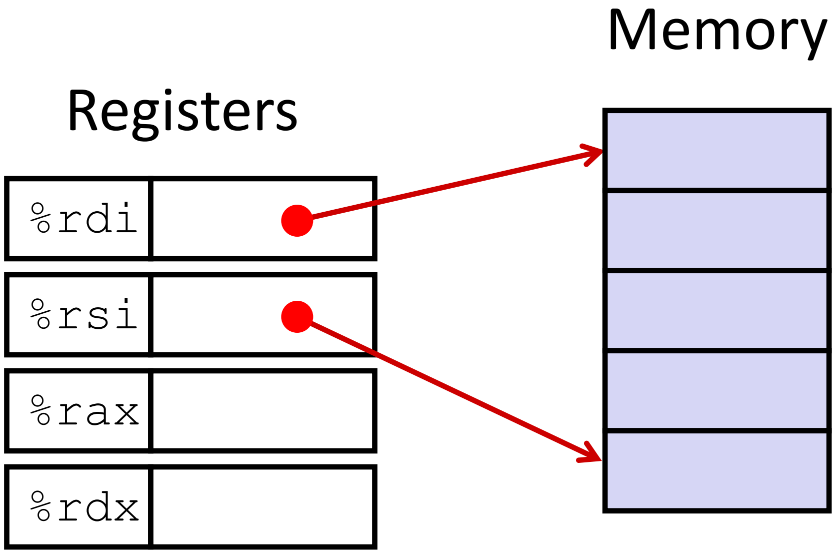
Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

Address	Value
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

Operand	Value
%rax	**0x100**
0x108	**0xAB**
$0x108	**0x108**
(%rax)	**0xFF**
8(%rax)	**0xAB**
13(%rax, %rdx)	**0x13**
260(%rcx, %rdx)	**0xAB**
0xF8(, %rcx, 8)	**0xFF**
(%rax, %rdx, 8)	**0x11**

Example: swap()

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



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

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

Understanding `Swap()`

Registers

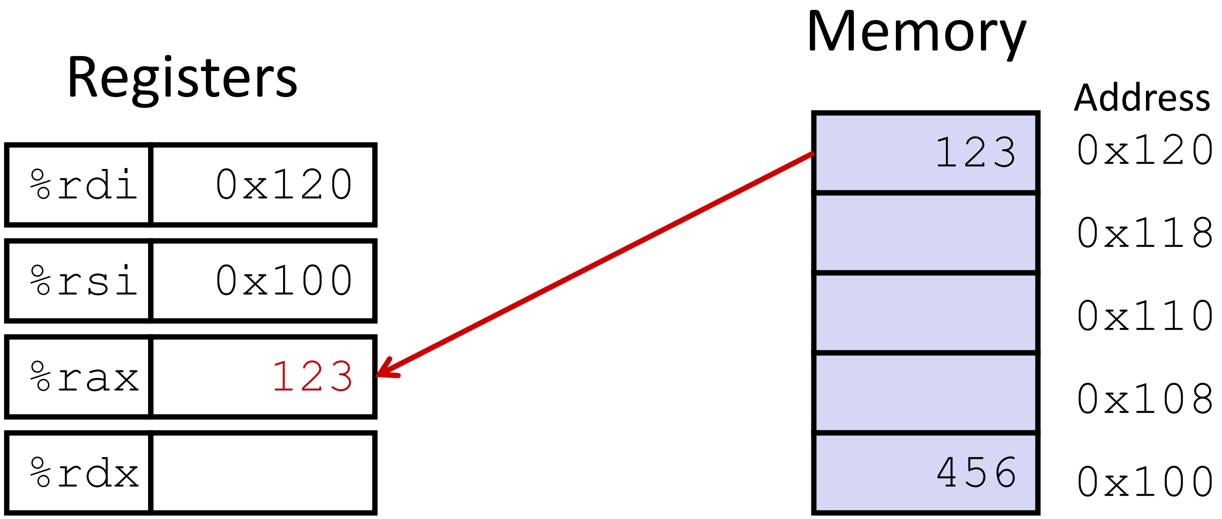
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

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()



Registers

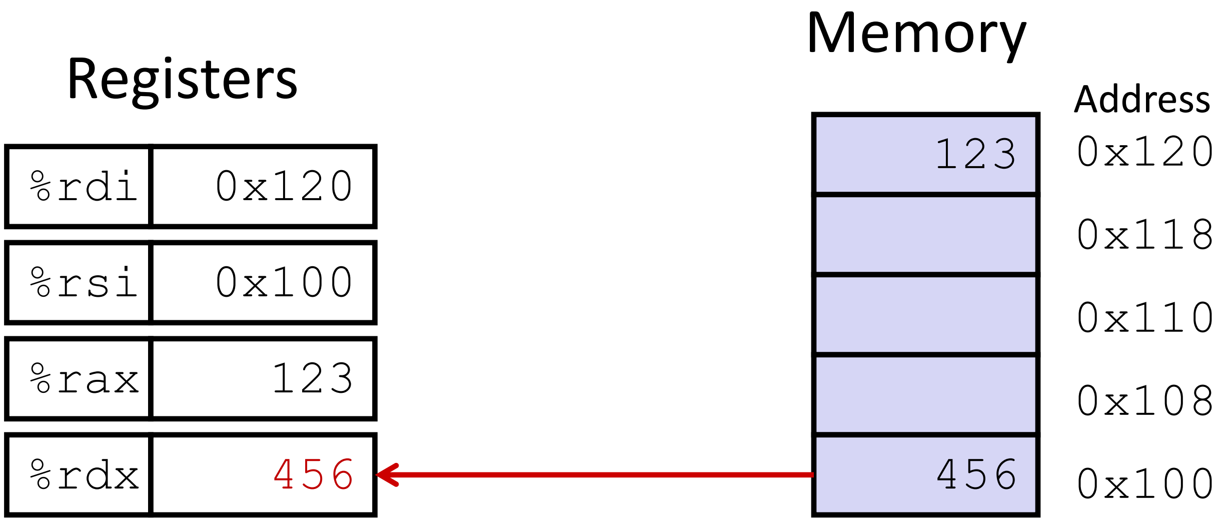
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

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()



Registers

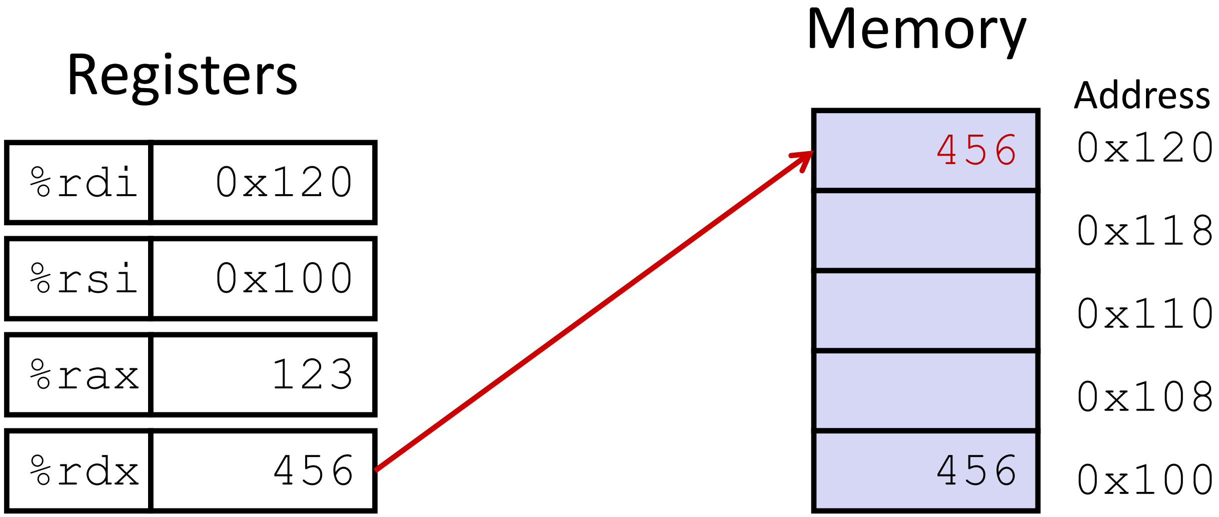
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

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

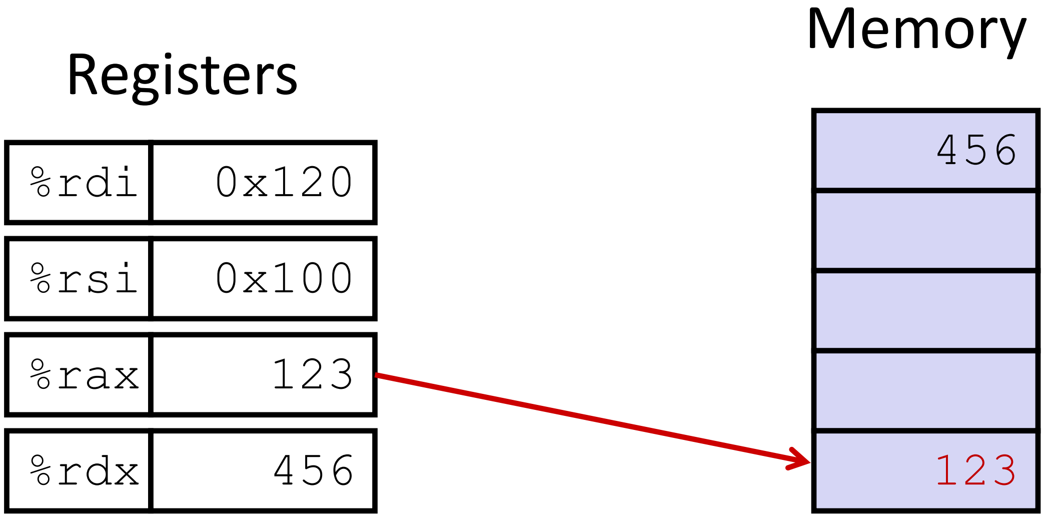
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

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
```

Practice Problem 3.5

A function has this prototype:

```
void decode(long *xp, long *yp, long *zp);
```

Here is the body of the code in assembly language:

```
/* xp in %rdi, yp in %rsi, zp in %rdx */
1 movq (%rdi), %r8
2 movq (%rsi), %rcx
3 movq (%rdx), %rax
4 movq %r8,(%rsi)
5 movq %rcx,(%rdx)
6 movq %rax,(%rdi)
```

Write C code for this function

```
void decode(long *xp, long *yp, long *zp) {
    long x = *xp; /* Line 1 */
    long y = *yp; /* Line 2 */
    long z = *zp; /* Line 3 */
    *yp = x;     /* Line 6 */
    *zp = y;     /* Line 8 */
    *xp = z;     /* Line 7 */
    return z;
}
```

Practice Problem

Suppose an array in C is declared as a global variable:

```
long array[34];
```

Write some assembly code that:

- sets `rsi` to the address of array
- sets `rbx` to the constant 9
- loads `array[9]` into register `rax`.

Use scaled index memory mode

Practice Problem

Suppose an array in C is declared as a global variable:

```
long array[34];
```

Write some assembly code that:

- sets `rsi` to the address of array
- sets `rbx` to the constant 9
- loads `array[9]` into register `rax`.

Use scaled index memory mode

```
movl $array,%rsi
movl $0x9,%rbx
movl (%rsi,%rbx,8),%rax
```

Arithmetic and Logical Operations

Load address

Load Effective Address (Quad)

`leaq S, D` ⇒ D ← &S

- Loads the *address* of S in D, not the *contents*
  - `leaq (%rax),%rdx`
  - Equivalent to `movq %rax,%rdx`
- Destination must be a register
- Used to compute addresses without a memory reference
  - e.g., translation of `p = &x[i];`

Load address

`leaq S, D` ⇒ `D` ← `&S`

- Commonly used by compiler to do simple arithmetic
  - If `%rdx = x`,
    - » `leaq 7(%rdx, %rdx, 4), %rdx` ⇒ 5x + 7
    - » Multiply and add all in one instruction
- 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
```

Practice Problem 3.6

%rax = x, %rcx = y

Expression	Result in %rdx
leaq 6(%rax), %rdx	**x+6**
leaq (%rax, %rcx), %rdx	**x+y**
leaq (%rax, %rcx, 4), %rdx	**x+4y**
leaq 7(%rax, %rax, 8), %rdx	**9x+7**
leaq 0xA(, %rcx, 4), %rdx	**4y+10**
leaq 9(%rax, %rcx, 2), %rdx	**x+2y+9**

Two Operand Arithmetic Operations

A little bit tricky

- Second operand is both a source and destination
- A bit like C operators ‘+=‘, ‘-=‘, etc.
- Max shift is 64 bits, so k is either an immediate byte, or register (e.g. %cl where %cl is byte 0 of register %rcx)
- No distinction between signed and unsigned int (why?)

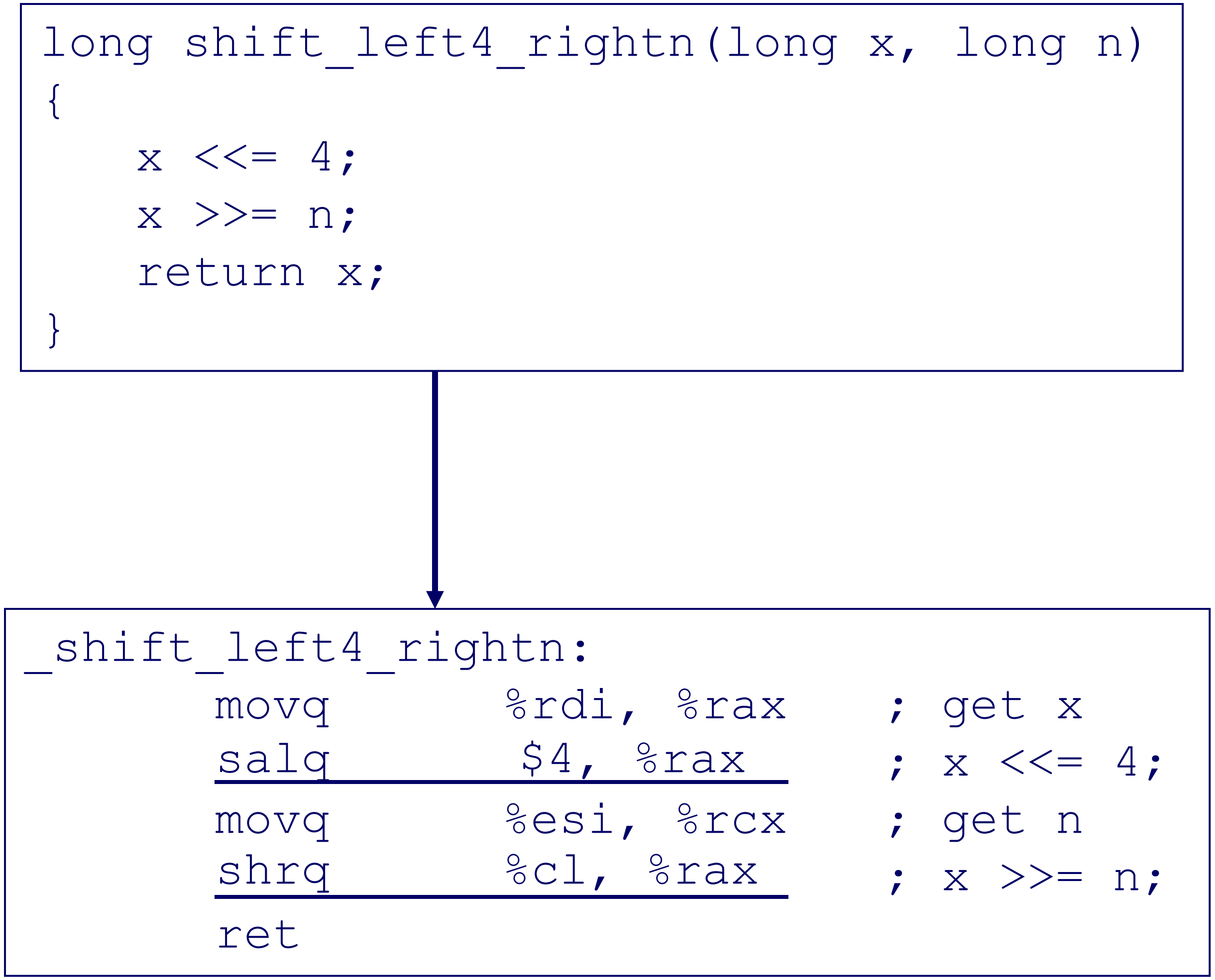
Format		*Computation*	
`addq`	`S, D`	`D = D + S`	
`subq`	`S, D`	`D = D – S`	
`imulq`	`S, D`	`D = D * S`	
`salq`	`S, D`	`D = D << S`	*Also called shlq*
`sarq`	`S, D`	`D = D >> S`	*Arithmetic shift right (sign extend)*
`shrq`	`S, D`	`D = D >> S`	*Logical shift right (zero fill)*
`xorq`	`S, D`	`D = D ^ S`	
`andq`	`S, D`	`D = D & S`	
`orq`	`S, D`	`D = D | S`	

One Operand Arithmetic Operations

Format		*Computation*
`incq`	`D`	`D = D + 1`
`decq`	`D`	`D = D – 1`
`negq`	`D`	`D = – D`
`notq`	`D`	`D = ~D`

See book for more instructions

Practice Problem 3.9



```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}
```

```
_shift_left4_rightn:
      movq      %rdi, %rax    ; get x
      salq       $4, %rax     ; x <<= 4;
      movq      %esi, %rcx    ; get n
      shrq      %cl, %rax     ; x >>= n;
      ret
```

Practice Problem 3.8

Address	Value
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

Instruction	Destination address	Result
`addq %rcx, (%rax)`	**0x100**	**0x100**
`subq %rdx, 8(%rax)`	**0x108**	**0xA8**
`imulq $16, (%rax, %rdx, 8)`	**0x118**	**0x110**
`incq 16(%rax)`	**0x110**	**0x14**
`decq %rcx`	**%rcx**	**0x0**
`subq %rdx, %rax`	**%rax**	**0xFD**

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
```

Compiler trick to generate efficient code

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

Practice Problem 3.10

What does this instruction do?

```
xorq     %rdx, %rdx
```

Zeros out register

How might it be different than this instruction?

```
movq     $0,  %rdx
```

3-byte instruction versus 7-byte

Null bytes encoded in instruction

Exam practice

Chapter 3 Problems (Part 1)

3.1	x86 operands
3.2,3.3	instruction operand sizes
3.4	instruction construction
3.5	disassemble to C
3.6	leaq
3.7	leaq disassembly
3.8	operations in x86
3.9	fill in x86 from C
3.10	fill in C from x86
3.11	xorq

Extra slides

Definitions

Architecture or instruction set architecture (ISA)

- Instruction specification, registers
- Examples: x86 IA32, x86-64, ARM

Microarchitecture

- Implementation of the architecture
- Examples: cache sizes and core frequency

Machine code (or object code)

- Byte-level programs that a processor executes

Assembly code

- A text representation of machine code

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 sumstore`

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

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
```

Within gdb Debugger

```
gdb sum
disassemble sumstore
```

Disassemble procedure

```
x/14xb sumstore
```

Examine the 14 bytes starting at `sumstore`

http://thefengs.com/wuchang/courses/cs201/class/05/math_examples.c

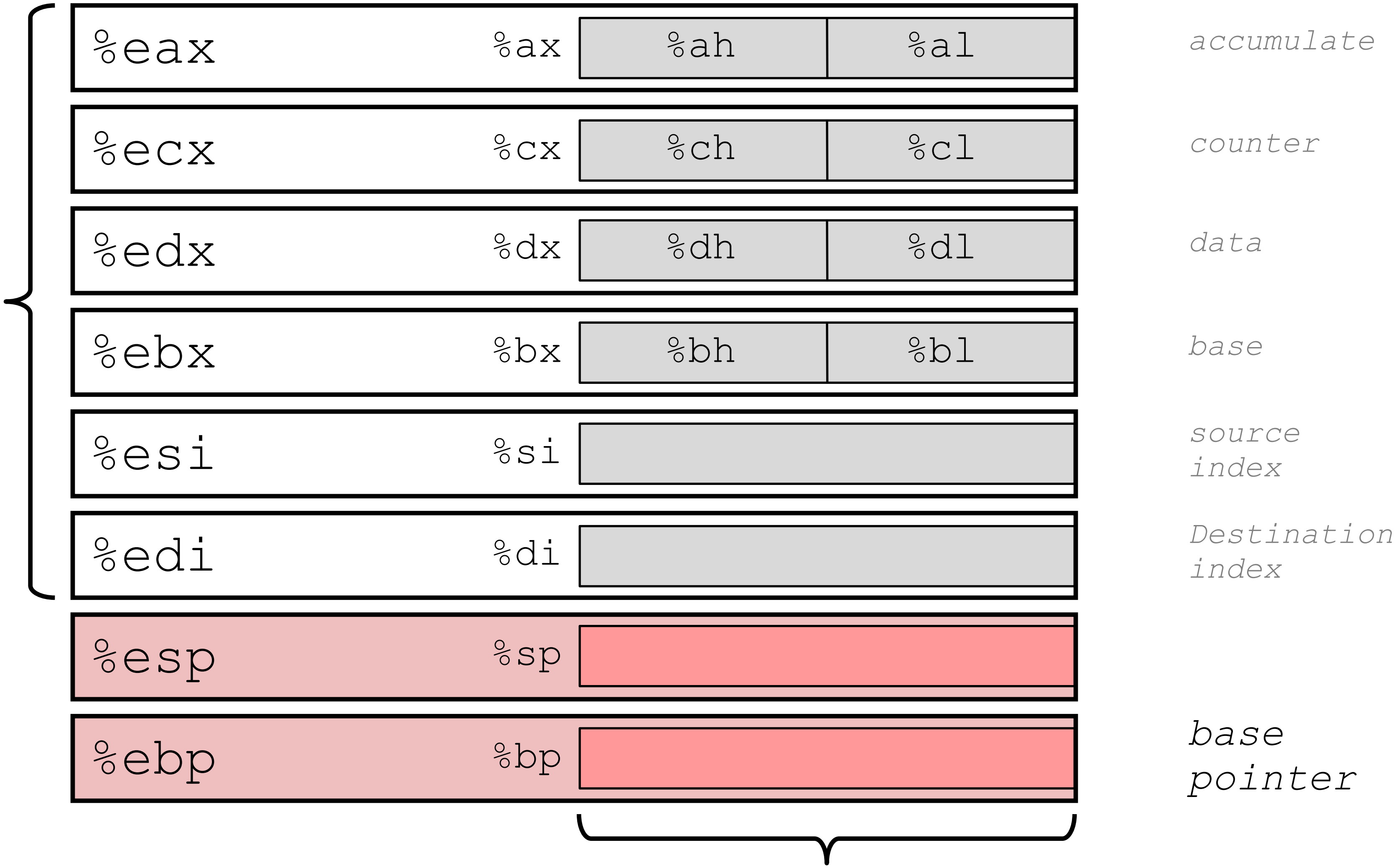
Object Code

Code for `sumstore`

- Total of 14 bytes
- Each instruction 1,3, or 5 bytes
- Starts at address `0x0400595`

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

Some History: IA32 Registers



%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			
%ebp	%bp			*base pointer*

Memory modes

Memory mode: Scaled indexed

- Absolute, indirect, base+displacement, indexed are simply special cases of Scaled indexed
- More special cases
  - (E_b,E_i,S) $M[R[E_b] + R[E_i]*S]$
  - (E_b,E_i) $M[R[E_b] + R[E_i]]$
  - $(,E_i,S)$ $M[R[E_i]*S]$
  - $Imm(,E_i,S)$ $M[Imm + R[E_i]*S]$

Alternate mov instructions

Not all move instructions are equivalent

- **There are three byte move instructions and each produces a different result**

    movb only changes specific byte

    movsbl does sign extension

    movzbl sets other bytes to zero

```
Assumptions: %dh = 0x8D, %rax = 0x98765432

movb     %dh, %al                         %rax = 0x9876548D
movsbl   %dh, %rax                       %rax = 0xFFFFFF8D
movzbl   %dh, %rax                        %rax = 0x0000008D
```

Data Movement Instructions

Instruction		Effect	Description
`movl` `movw`	`S,D` `S,D`	D ← S D ← S	Move double word Move word
`movb`	`S,D`	D ← S	Move byte
`movsbl`	`S,D`	D ← SignExtend(S)	Move sign-extended byte
`movzbl`	`S,D`	D ← ZeroExtend(S)	Move zero-extended byte