

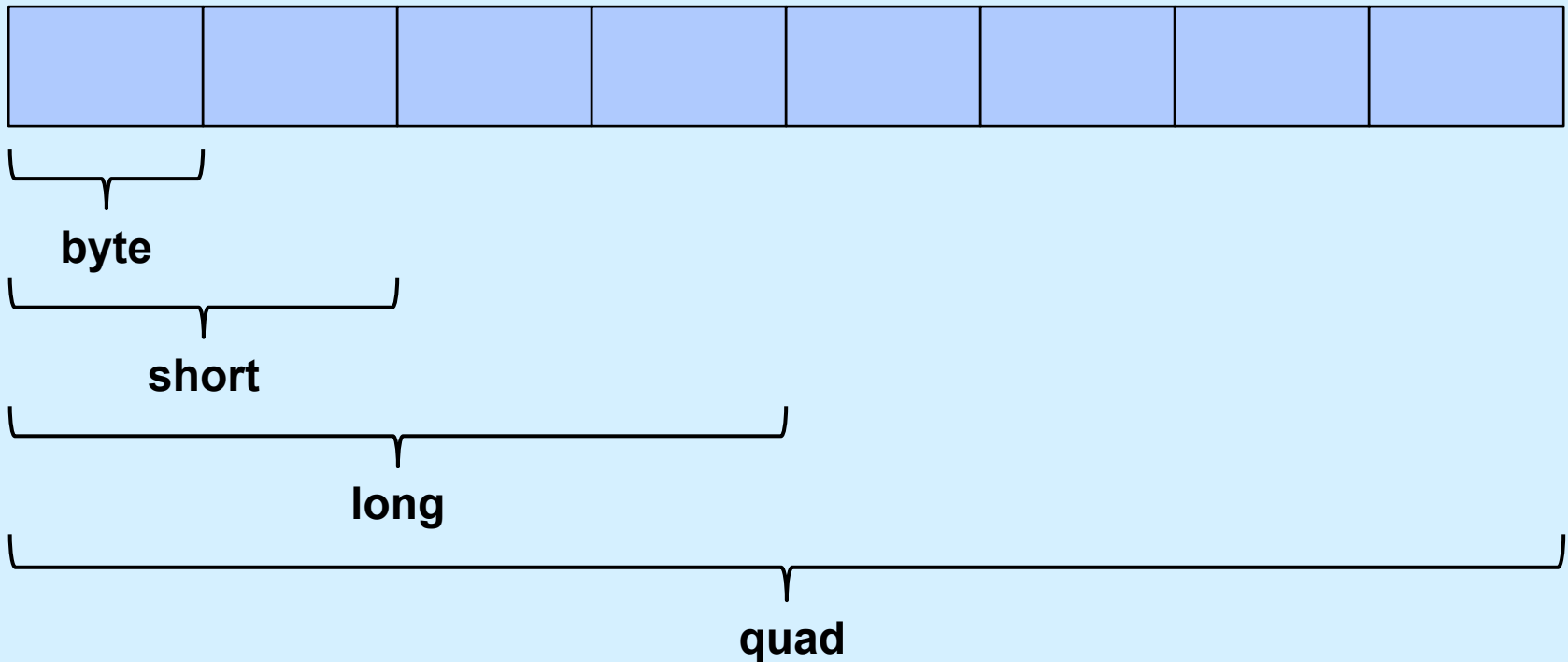
# CS 33

## Machine Programming (1)

# Data Types on Intel x86

- **“Integer” data of 1, 2, or 4 bytes (plus 8 bytes on x86-64)**
  - data values
    - » whether signed or unsigned depends on interpretation
  - addresses (untyped pointers)
- **Floating-point data of 4, 8, or 10 bytes**
- **No aggregate types such as arrays or structures**
  - just contiguously allocated bytes in memory

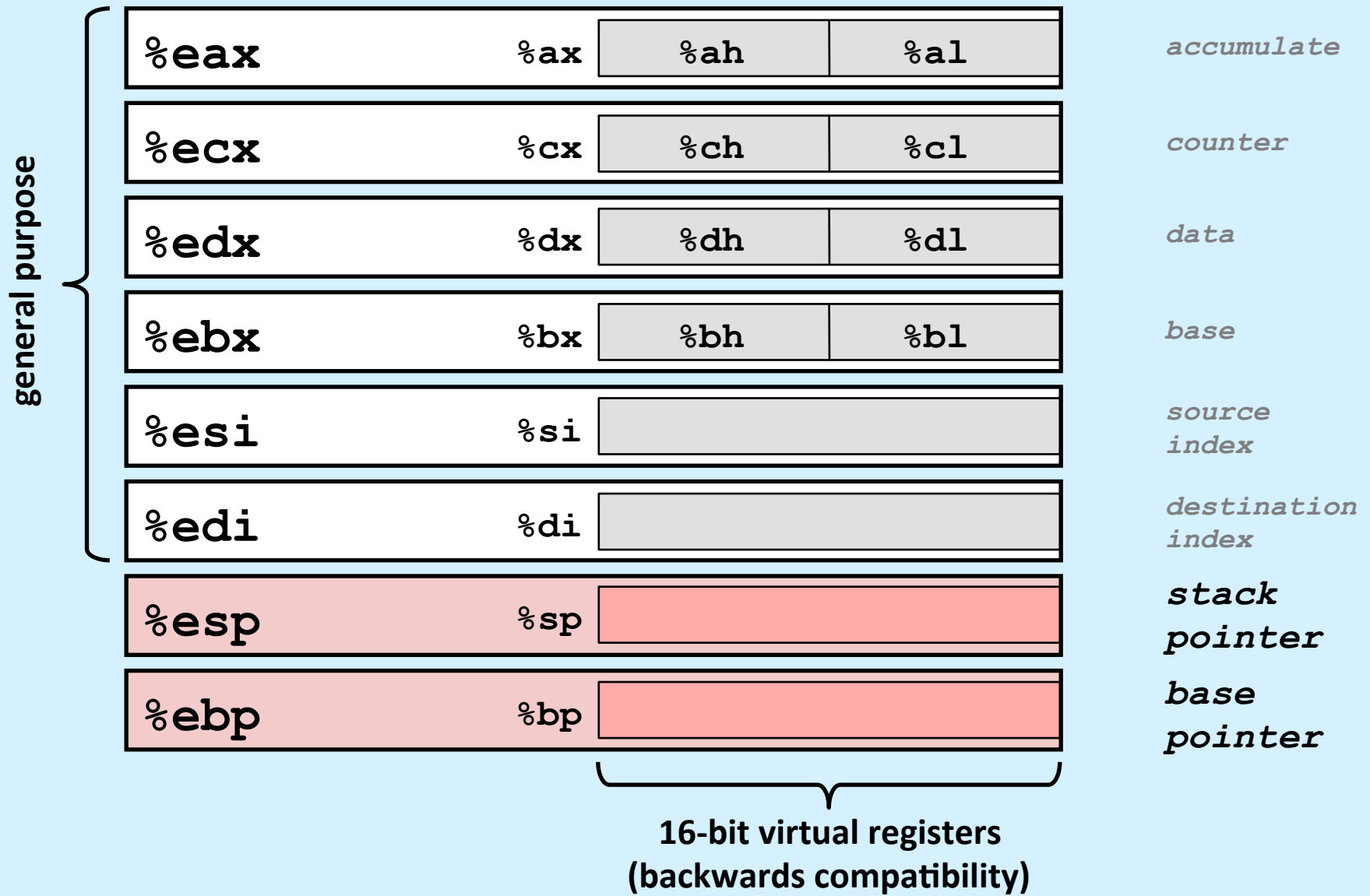
# Operand Size



- **Rather than `mov ...`**
  - `movb`
  - `movs`
  - `movl`
  - `movq` (x86-64 only)

# General-Purpose Registers (IA32)

Origin  
(mostly obsolete)



# Moving Data: IA32

- Moving data

`movl 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 8 integer registers

- » example: `%eax`, `%edx`

- » but `%esp` and `%ebp` reserved for special use

- » others have special uses for particular instructions

- **Memory:** 4 consecutive bytes of memory at address given by register(s)

- » simplest example: `(%eax)`

- » various other “address modes”

`%eax`

`%ecx`

`%edx`

`%ebx`

`%esi`

`%edi`

`%esp`

`%ebp`

# movl Operand Combinations

	Source	Dest	Src, Dest	C Analog
movl	Imm	Reg	movl \$0x4,%eax	temp = 0x4;
		Mem	movl \$-147, (%eax)	*p = -147;
	Reg	Reg	movl %eax,%edx	temp2 = temp1;
		Mem	movl %eax, (%edx)	*p = temp;
	Mem	Reg	movl (%eax), %edx	temp = *p;

**Cannot (normally) do memory-memory transfer with a single instruction**

# Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**  
– register R specifies memory address

```
movl (%ecx) , %eax
```

- **Displacement D(R) Mem[Reg[R]+D]**  
– register R specifies start of memory region  
– constant displacement D specifies offset

```
movl 8(%ebp) , %edx
```

# Using Simple Addressing Modes

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

swap:

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
```

} Set  
Up

```
movl  8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
```

} Body

```
popl %ebx
popl %ebp
ret
```

} Finish

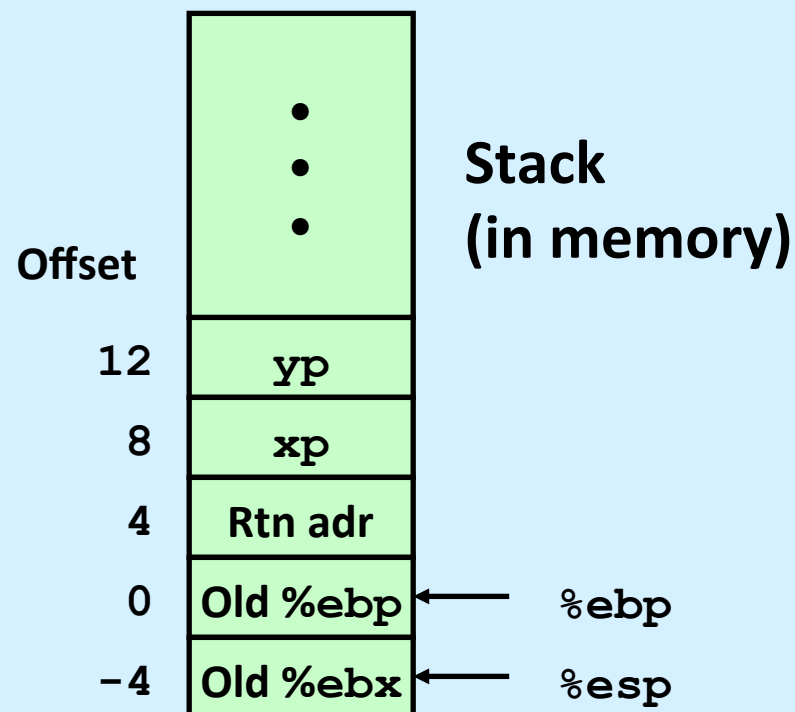


# Understanding Swap

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

Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
```



# Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		123
		456
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	

```

movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
    
```

# Understanding Swap

%eax	
%edx	0x124
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		123
		456
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	
	-4	

```

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
    
```

# Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		0x100

```

movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
    
```

# Understanding Swap

%eax	
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	Rtn adr
%ebp	0	0x108
		0x104
	-4	0x100

```

movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx    # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
    
```

# Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
Offset	12	0x124
	8	0x120
	4	0x11c
	0	0x118
	-4	0x114
		0x110
		0x10c
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	0	0x108
	-4	0x104
		0x100

```

movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax    # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
    
```

# Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		0x124
		0x120
		0x11c
		0x118
		0x114
yp	12	0x110
xp	8	0x10c
	4	Rtn adr
%ebp	0	0x108
		0x104
	-4	0x100

```

movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)    # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
    
```

# Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

		Address
		456
		0x124
		123
		0x120
		0x11c
		0x118
		0x114
yp	12	0x120
0x110		
xp	8	0x124
0x10c		
	4	Rtn adr
0x108		
%ebp	0	
0x104		
	-4	
0x100		

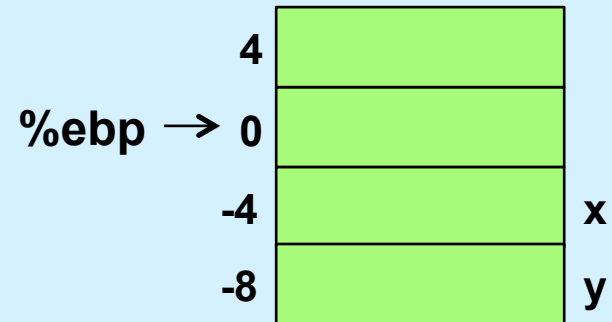
```

movl 8(%ebp), %edx    # edx = xp
movl 12(%ebp), %ecx   # ecx = yp
movl (%edx), %ebx     # ebx = *xp (t0)
movl (%ecx), %eax     # eax = *yp (t1)
movl %eax, (%edx)     # *xp = t1
movl %ebx, (%ecx)     # *yp = t0
  
```



# Quiz 1

```
movl -4(%ebp), %eax
movl (%eax), %eax
movl (%eax), %eax
movl %eax, -8(%ebp)
```



**Which C statements best describe the assembler code?**

```
// a
int x;
int y;
y = x;
```

```
// b
int *x;
int y;
y = *x;
```

```
// c
int **x;
int y;
y = **x;
```

```
// d
int ***x;
int y;
y = ***x;
```

# Complete Memory-Addressing Modes

- Most general form

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

- D:    constant “displacement”
- Rb:   base register: any of 8 integer registers
- Ri:   index register: any, except for `%esp`
  - » unlikely you’d use `%ebp` either
- S:    scale: 1, 2, 4, or 8

- Special cases

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

# Address-Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x0100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x0100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

# Address-Computation Instruction

- **leal src, dest**
  - src is address mode expression
  - set dest 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  $x + k*y$ 
    - »  $k = 1, 2, 4, \text{ or } 8$
- **Example**

```
int mul12(int x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
movl 8(%ebp), %eax    # get arg
leal (%eax,%eax,2), %eax # t <- x+x*2
sall $2, %eax         # return t<<2
```

# Quiz 2

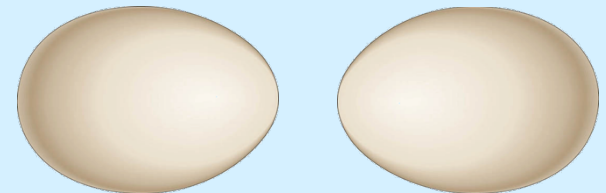
**What value ends up in %ecx?**

```
movl $1000,%eax
movl $1,%ebx
movl 2(%eax,%ebx,4),%ecx
```

- a) 0x02030405
- b) 0x05040302
- c) 0x06070809
- d) 0x09080706

1009:	0x09
1008:	0x08
1007:	0x07
1006:	0x06
1005:	0x05
1004:	0x04
1003:	0x03
1002:	0x02
1001:	0x01
%eax → 1000:	0x00

**Hint:**



# x86-64 General-Purpose Registers

<code>%rax</code>	<code>%eax</code>
<code>%rbx</code>	<code>%ebx</code>
<code>%rcx</code>	<code>%ecx</code>
<code>%rdx</code>	<code>%edx</code>
<code>%rsi</code>	<code>%esi</code>
<code>%rdi</code>	<code>%edi</code>
<code>%rsp</code>	<code>%esp</code>
<code>%rbp</code>	<code>%ebp</code>

<code>%r8</code>	<code>%r8d</code>
<code>%r9</code>	<code>%r9d</code>
<code>%r10</code>	<code>%r10d</code>
<code>%r11</code>	<code>%r11d</code>
<code>%r12</code>	<code>%r12d</code>
<code>%r13</code>	<code>%r13d</code>
<code>%r14</code>	<code>%r14d</code>
<code>%r15</code>	<code>%r15d</code>

- Extend existing registers to 64 bits. Add 8 new ones.
- No special purpose for `%ebp/%rbp`

# 32-bit Instructions on x86-64

- **addl 4(%rdx), %eax**
  - memory address must be 64 bits
  - operands (in this case) are 32-bit
    - » result goes into %eax
      - lower half of %rax
      - upper half is filled with zeroes

# 32-bit code for swap

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

swap:

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
```

} Set  
Up

```
movl  8(%ebp), %edx
movl  12(%ebp), %ecx
movl  (%edx), %ebx
movl  (%ecx), %eax
movl  %eax, (%edx)
movl  %ebx, (%ecx)
```

} Body

```
popl  %ebx
popl  %ebp
ret
```

} Finish



# 64-bit code for swap

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

swap:

```
movl    (%rdi), %edx
movl    (%rsi), %eax
movl    %eax, (%rdi)
movl    %edx, (%rsi)
```

ret

} Set Up

} Body

} Finish

- **Arguments passed in registers (why useful?)**
  - first (**xp**) in **%rdi**, second (**yp**) in **%rsi**
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - data held in registers **%eax** and **%edx**
  - **movl operation**

# 64-bit code for long int swap

swap\_l:

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

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

ret

} Set Up

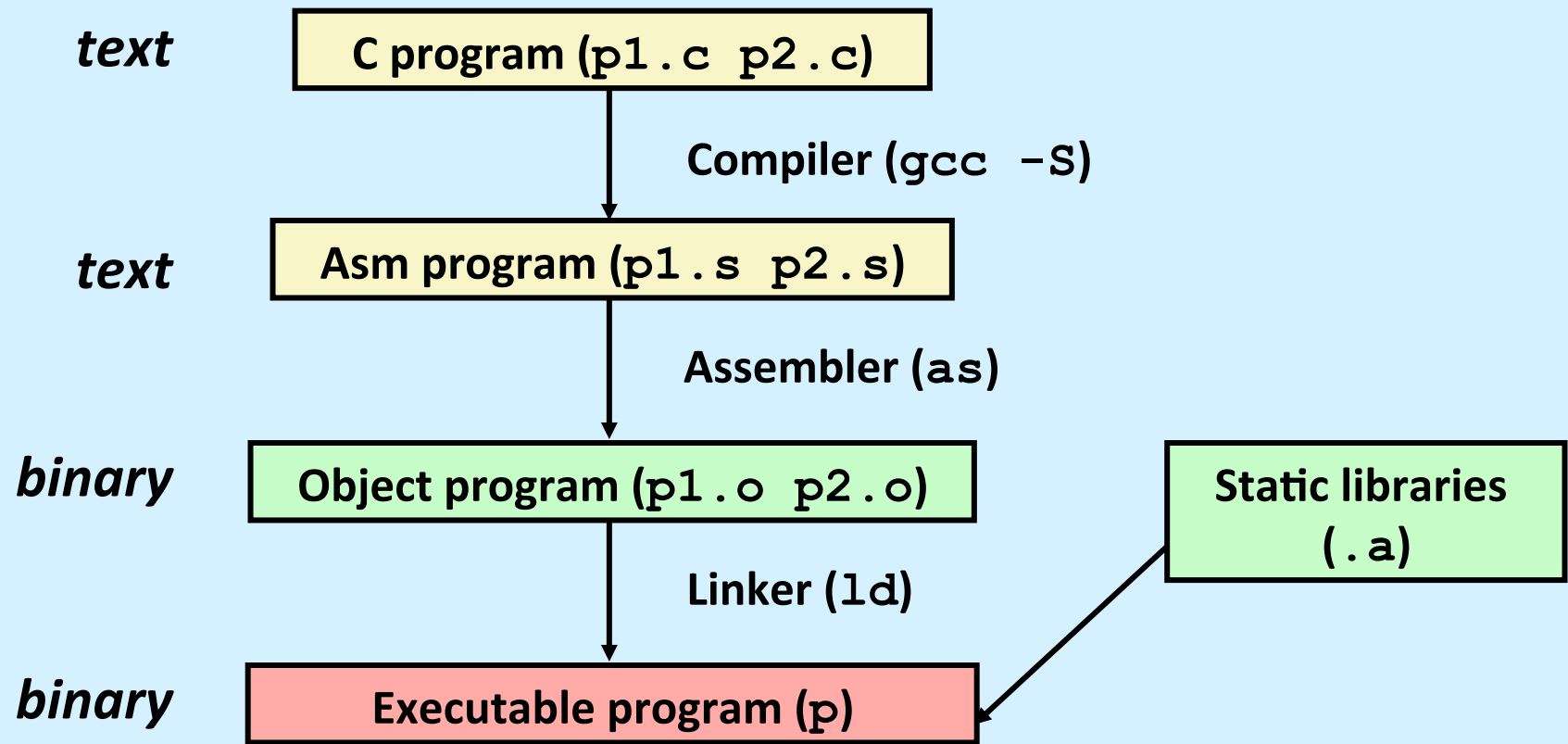
} Body

} Finish

- **64-bit data**
  - data held in registers `%rax` and `%rdx`
  - `movq` operation
    - » “q” stands for quad-word

# Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - » use basic optimizations (`-O1`)
  - » put resulting binary in file `p`



# Object Code

## Code for sum

0x401040 <sum>:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x5d

0xc3

- **Total of 11 bytes**
- **Each instruction:  
1, 2, or 3 bytes**
- **Starts at address  
0x401040**

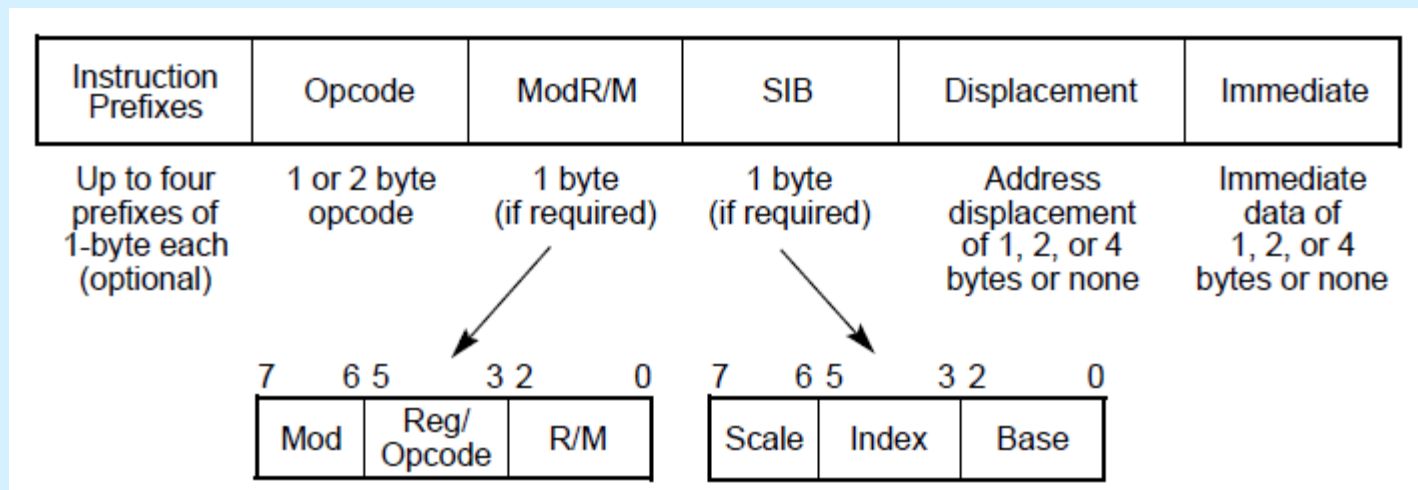
- **Assembler**

- 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 printf
- some libraries are *dynamically linked*
  - » linking occurs when program begins execution

# Instruction Format



# Disassembling Object Code

## Disassembled

```
080483c4 <sum>:  
80483c4: 55          push    %ebp  
80483c5: 89 e5       mov     %esp, %ebp  
80483c7: 8b 45 0c    mov     0xc(%ebp), %eax  
80483ca: 03 45 08    add     0x8(%ebp), %eax  
80483cd: 5d          pop     %ebp  
80483ce: c3          ret
```

- **Disassembler**

`objdump -d <file>`

- useful tool for examining object code
- analyzes bit pattern of series of instructions
- produces approximate rendition of assembly code
- can be run on either executable or object (.o) file

# Alternate Disassembly

## Object

0x401040:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x5d

0xc3

## Disassembled

Dump of assembler code for function sum:

0x080483c4 <sum+0>: push %ebp

0x080483c5 <sum+1>: mov %esp,%ebp

0x080483c7 <sum+3>: mov 0xc(%ebp),%eax

0x080483ca <sum+6>: add 0x8(%ebp),%eax

0x080483cd <sum+9>: pop %ebp

0x080483ce <sum+10>: ret

- **Within gdb debugger**

`gdb <file>`

`disassemble sum`

– `disassemble procedure`

`x/11xb sum`

– `examine the 11 bytes starting at sum`

# How Many Instructions are There?

- We cover ~29
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn't count:
  - floating-point instructions
  - SIMD instructions
  - AMD-added instructions
  - undocumented instructions



# Some Arithmetic Operations

- Two-operand instructions:

Format	Computation		
addl	Src, Dest	Dest = Dest + Src	
subl	Src, Dest	Dest = Dest - Src	
imull	Src, Dest	Dest = Dest * Src	
sall	Src, Dest	Dest = Dest << Src	Also called shll Arithmetic Logical
sarl	Src, Dest	Dest = Dest >> Src	
shrl	Src, Dest	Dest = Dest >> Src	
xorl	Src, Dest	Dest = Dest ^ Src	
andl	Src, Dest	Dest = Dest & Src	
orl	Src, Dest	Dest = Dest   Src	

- watch out for argument order!
- no distinction between signed and unsigned int (why?)

# Some Arithmetic Operations

- **One-operand Instructions**

`incl`      `Dest`           $= \text{Dest} + 1$

`decl`      `Dest`           $= \text{Dest} - 1$

`negl`      `Dest`           $= - \text{Dest}$

`notl`      `Dest`           $= \sim \text{Dest}$

- **See book for more instructions**

# Arithmetic Expression Example

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
arith:
    leal    (%rdi,%rsi), %eax
    addl    %edx, %eax
    leal    (%rsi,%rsi,2), %edx
    sall    $4, %edx
    leal    4(%rdi,%rdx), %ecx
    imull    %ecx, %eax
    ret
```

# Understanding arith

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
leal    (%rdi,%rsi), %eax
addl    %edx, %eax
leal    (%rsi,%rsi,2), %edx
sall    $4, %edx
leal    4(%rdi,%rdx), %ecx
imull    %ecx, %eax
ret
```

%rdx	z
%rsi	y
%rdi	x

# Understanding arith

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

%rdx	z
%rsi	y
%rdi	x

```
leal    (%rdi,%rsi), %eax    # eax = x+y      (t1)
addl    %edx, %eax          # eax = t1+z      (t2)
leal    (%rsi,%rsi,2), %edx  # edx = 3*y      (t4)
sall    $4, %edx            # edx = t4*16      (t4)
leal    4(%rdi,%rdx), %ecx   # ecx = x+4+t4   (t5)
imull   %ecx, %eax          # eax *= t5        (rval)
ret
```

# Observations about `arith`

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

<code>leal</code>	<code>(%rdi,%rsi), %eax</code>	<code>#</code>	<code>eax = x+y</code>	<code>(t1)</code>
<code>addl</code>	<code>%edx, %eax</code>	<code>#</code>	<code>eax = t1+z</code>	<code>(t2)</code>
<code>leal</code>	<code>(%rsi,%rsi,2), %edx</code>	<code>#</code>	<code>edx = 3*y</code>	<code>(t4)</code>
<code>sall</code>	<code>\$4, %edx</code>	<code>#</code>	<code>edx = t4*16</code>	<code>(t4)</code>
<code>leal</code>	<code>4(%rdi,%rdx), %ecx</code>	<code>#</code>	<code>ecx = x+4+t4</code>	<code>(t5)</code>
<code>imull</code>	<code>%ecx, %eax</code>	<code>#</code>	<code>eax *= t5</code>	<code>(rval)</code>
<code>ret</code>				

# Another Example

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$2^{13} = 8192$ ,  $2^{13} - 7 = 8185$

<code>xorl %esi, %edi</code>	<code># edi = x^y</code>	<code>(t1)</code>
<code>sarl \$17, %edi</code>	<code># edi = t1&gt;&gt;17</code>	<code>(t2)</code>
<code>movl %edi, %eax</code>	<code># eax = edi</code>	
<code>andl \$8185, %eax</code>	<code># eax = t2 &amp; mask</code>	<code>(rval)</code>

# Quiz 3

- What is the final value in %esi?

```
xorl %esi, %esi  
incl %esi  
sall %esi, %esi  
addl %esi, %esi
```

- a) 2
- b) 4
- c) 8
- d) indeterminate