

CS 201
IA32 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 “IA32” architecture

IA32

- Evolutionary design starting in 1978 with 8086
- Features added over time
- Now there’s lots of baggage, obsolete features

Complex Instruction Set Computer (CISC)

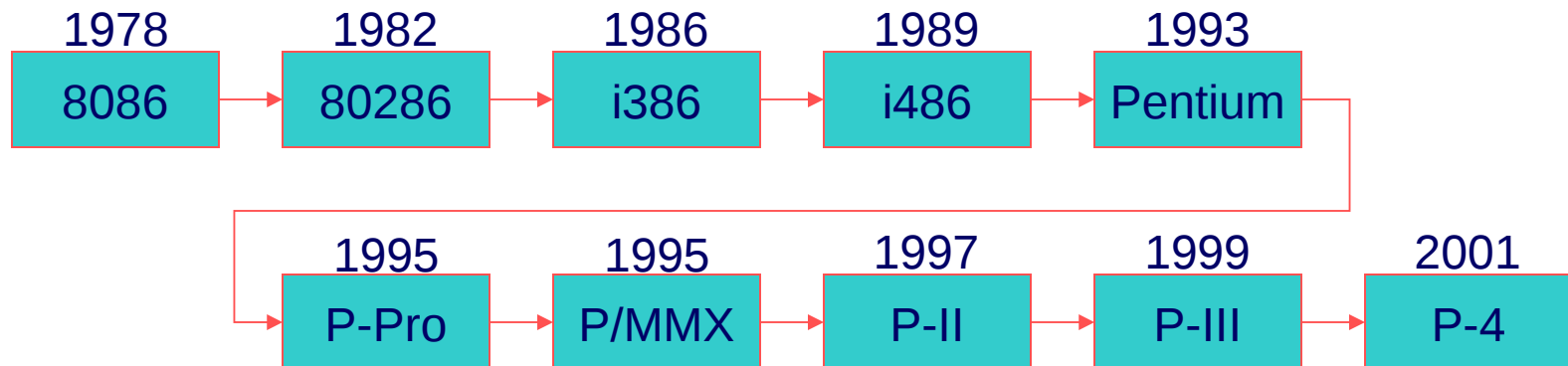
- Many different instructions with many different formats
- But we’ll only look at a small subset

IA32 Historically

1986: i386 was the first Intel 32-bit processor, first Intel processor to run Unix in a meaningful way.

- AT&T Unix System V.3
- GNU gcc still supports approximately the i386 instruction set

1995: Windows 95 was the first native 32-bit O. S. from Microsoft



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

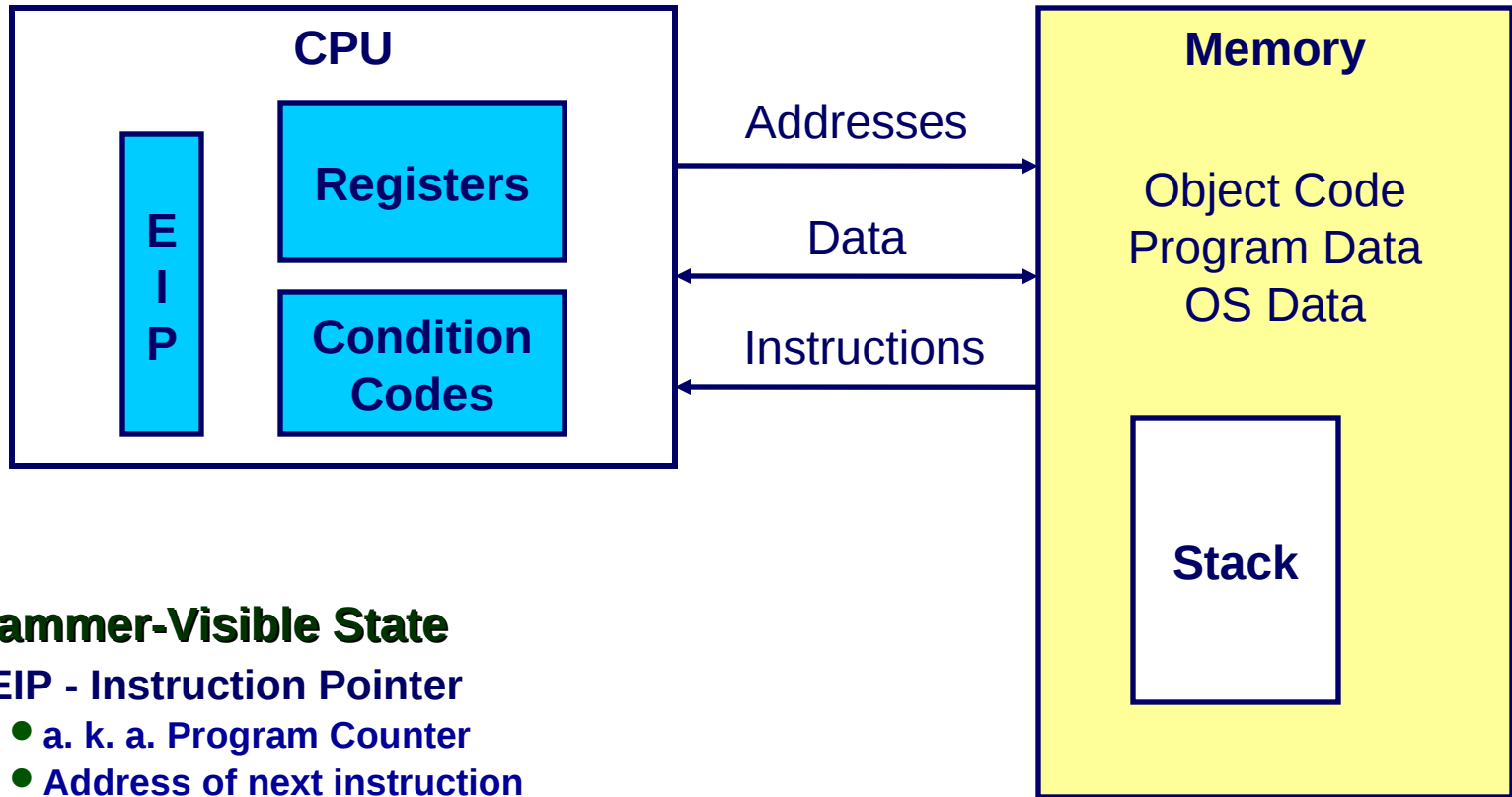
Assign mnemonics to machine code

- Assembly language for specifying machine instructions
- Names for the machine instructions and registers
 - `movl %eax, %ecx`
- There is no standard for IA32 assemblers
 - Intel assembly language
 - AT&T Unix 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

Assembly Programmer's View



Programmer-Visible State

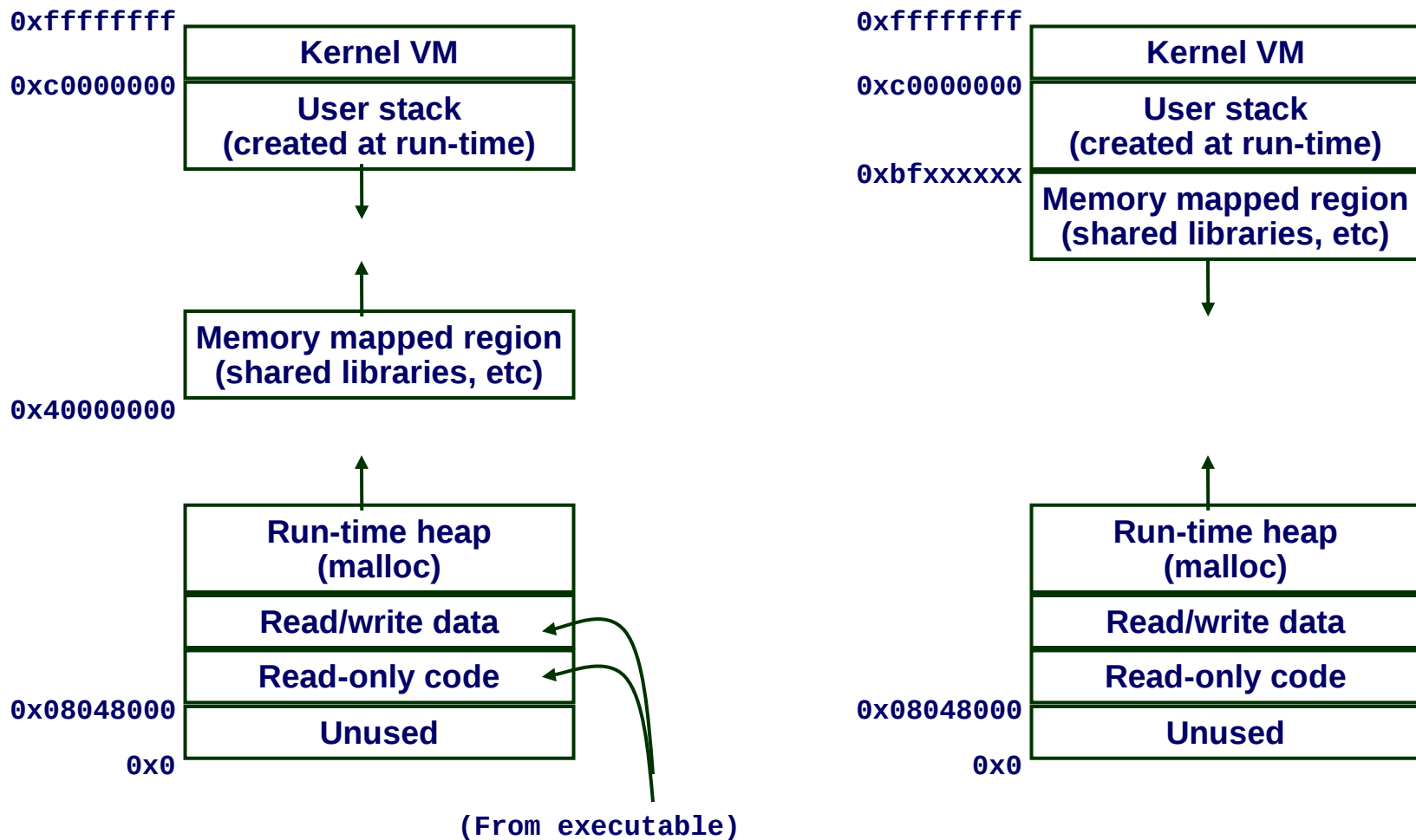
- EIP - Instruction Pointer
 - a. k. a. Program Counter
 - Address of next instruction
- Register File
 - Heavily used program data
- Condition Codes
 - Store status information about most recent arithmetic operation
 - Used for conditional branching

Memory

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

VM abstraction

Linux memory layout



Registers

Special memory not part of main memory

- Located on CPU
- Used to store temporary values

CPU does not directly operate on data in memory

- Data needs to be loaded into registers for CPU
- `movl (%eax), %ecx`
 - Fetch data at address contained in `%eax`
 - Store in register `%ecx`
- Once in a register, processor can operate on it
- Typically, data is loaded into registers, manipulated or used, and then written back to memory

Registers

The IA32 architecture is “register poor”

- Few general purpose registers
 - 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.)
- Compiler complexity issues

IA32 Register Conventions

%eax, %ecx, %edx and %ebx are general purpose registers that can be accessed as 32-bit, 16-bit, or 8-bit storage

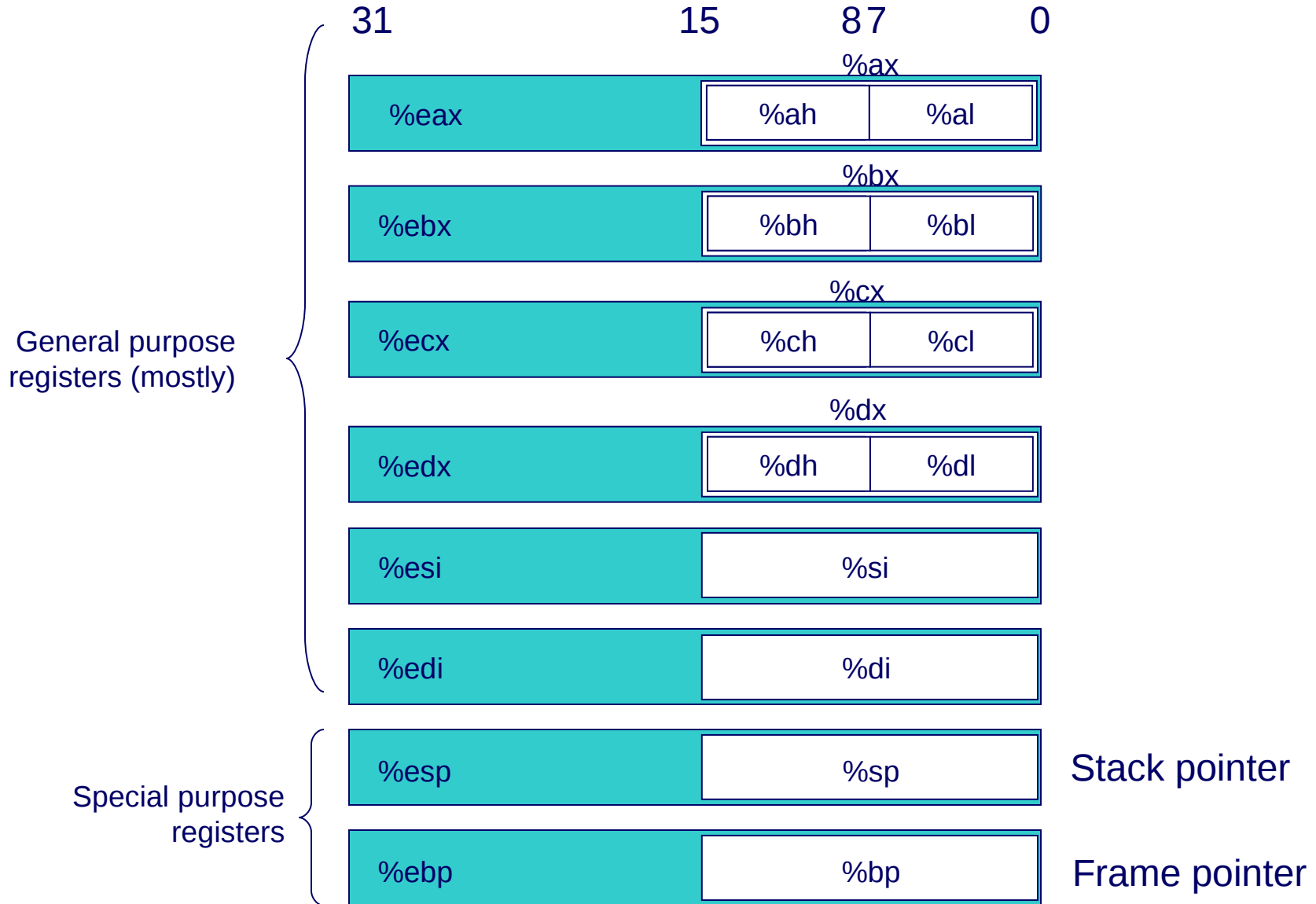
- The 'e' in the 32-bit names means “extended” (from 16 bit)
- %eax == 32 bits
- %ax == low 16 bits
- %ah == high byte of %ax
- %al == low byte of %ax

%esi, %edi general purpose registers that can be accessed as 32-bit and 16-bit

%esp and %ebp are the stack pointer and frame pointer, respectively

- We'll study these later

IA32 General Registers



Compiling for x86

C Code

Add 2 signed integers)

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Obtain with command

```
gcc -O -S code.c
```

Produces file code.s

Code is the same for signed and unsigned

x: Register %eax
y: Memory M[%ebp+8]
t: Register %eax

Code leaves the sum in %eax since integer return values passed back in %eax

Object code for instruction is 3-bytes stored at 0x401046

Generated Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

-12- addl 8(%ebp),%eax

0x401046: 03 45 08

What's the "l" for on the end?

```
addl 8(%ebp),%eax
```

It stands for "long."

It tells the size of the operand.

Baggage from the days of 16-bit processors

Now we have these crazy terms

- 8 bits is a *byte*
- 16 bits is a *word*
- 32 bits is a *double or long word*
- 64 bits is a *quad word*

IA32 Standard Data Types

C Declaration	Intel Data Type	GAS Suffix	Size in bytes
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
unsigned	Double word	l	4
long int	Double word	l	4
unsigned long	Double word	l	4
char *	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

Data Access

Operand types

A typical instruction acts on 2 or more *operands*

- `addl %ecx, %edx` adds the contents of `ecx` to `edx`

Three general types of operands

- **Immediate**
 - Like a C constant, but preceded by `$`
 - e.g., `$0x1F`, `$-533`
 - Encoded with 1, 2, or 4 bytes based on instruction
- **Register:** the value in one of the 8 integer registers
- **Memory:** a memory address
 - There are many modes for addressing memory

Operand examples using `mov`

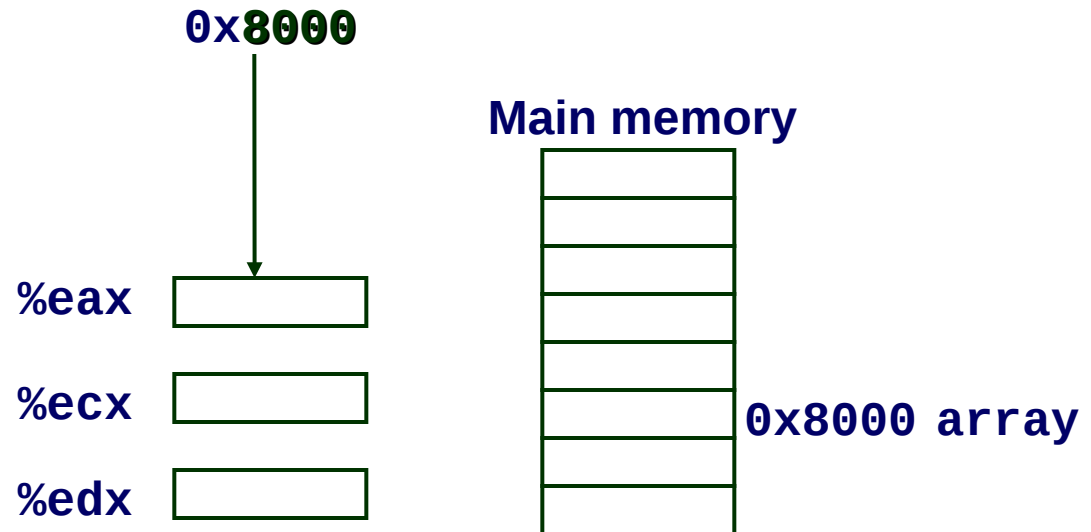
	Source	Destination	C Analog	
movl	Imm	Reg	<code>movl \$0x4,%eax</code>	<code>temp = 0x4;</code>
		Mem	<code>movl \$-147, (%eax)</code>	<code>*p = -147;</code>
	Reg	Reg	<code>movl %eax,%edx</code>	<code>temp2 = temp1;</code>
		Mem	<code>movl %eax, (%edx)</code>	<code>*p = temp;</code>
	Mem	Reg	<code>movl (%eax),%edx</code>	<code>temp = *p;</code>

- Memory-memory transfers cannot be done with single instruction

Immediate mode

Immediate has only one mode

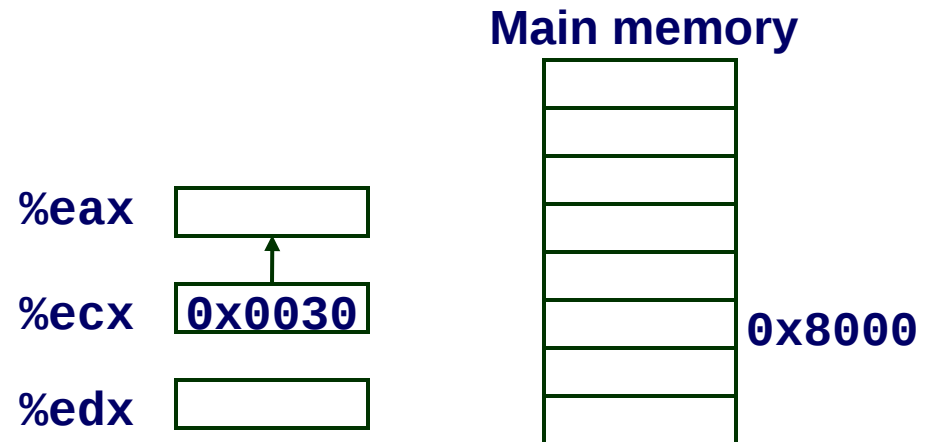
- Form: **\$Imm**
- Operand value: **Imm**
 - `movl $0x8000,%eax`
 - `movl $array,%eax`
 - » `int array[30]; /* array = global variable stored at 0x8000 */`



Register mode

Register has only one mode

- Form: E_a
- Operand value: $R[E_a]$
 - `movl %ecx,%eax`



Memory modes

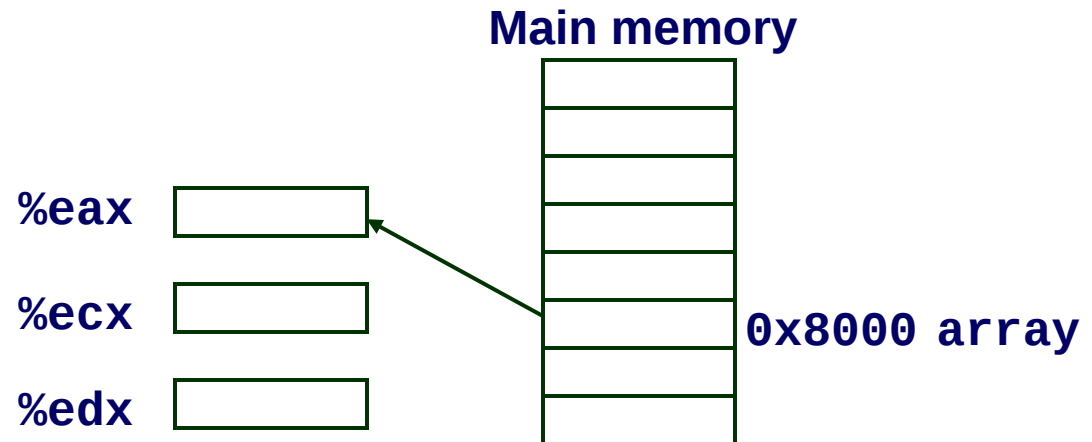
Memory has multiple modes

- **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

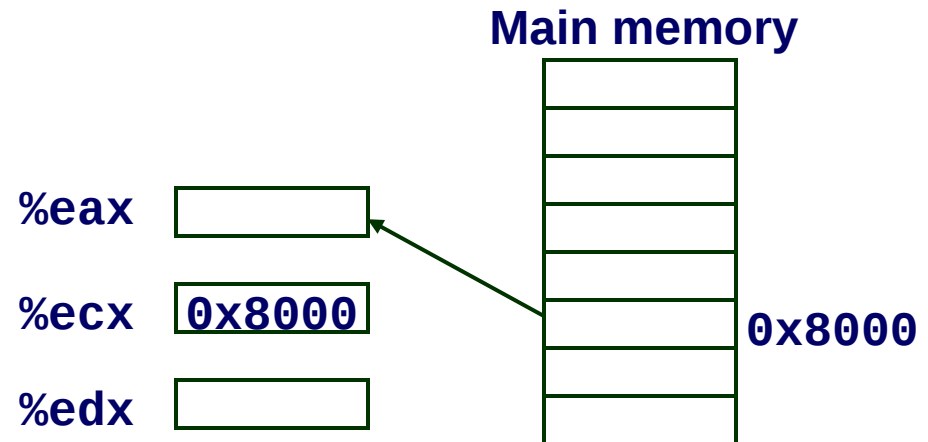
- Form: **Imm**
 - Operand value: **M[Imm]**
 - `movl 0x8000,%eax`
 - `movl array,%eax`
- » `int array[30]; /* array = global variable stored at 0x8000 */`



Memory modes

Memory mode: Indirect

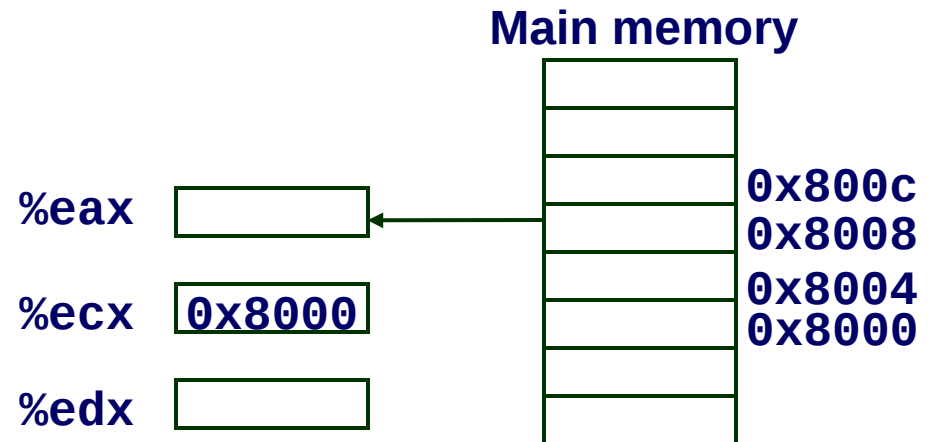
- Form: (E_a)
- Operand value: $M[R[E_a]]$
 - Register E_a specifies the memory address
 - `movl (%ecx), %eax`



Memory modes

Memory mode: Base + Displacement

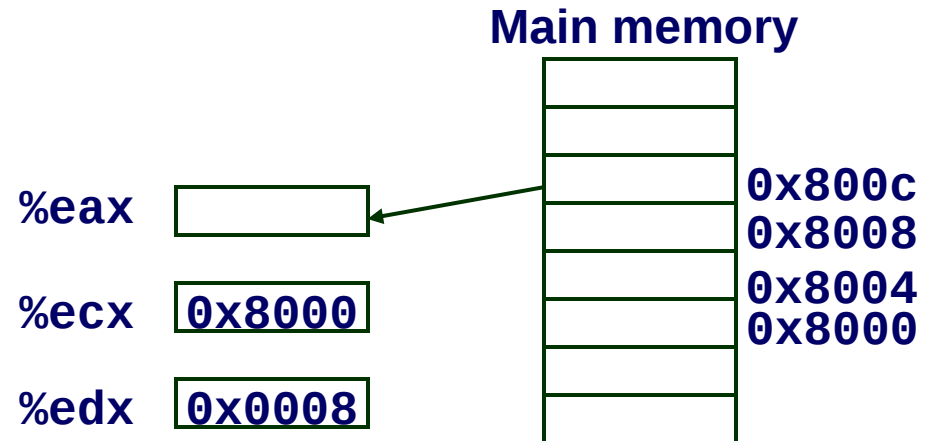
- Form: $\text{Imm}(E_b)$
- Operand value: $M[\text{Imm} + R[E_b]]$
 - Register E_b specifies start of memory region
 - Imm specifies the offset/displacement
- `movl 8(%ecx), %eax`



Memory modes

Memory mode: Indexed

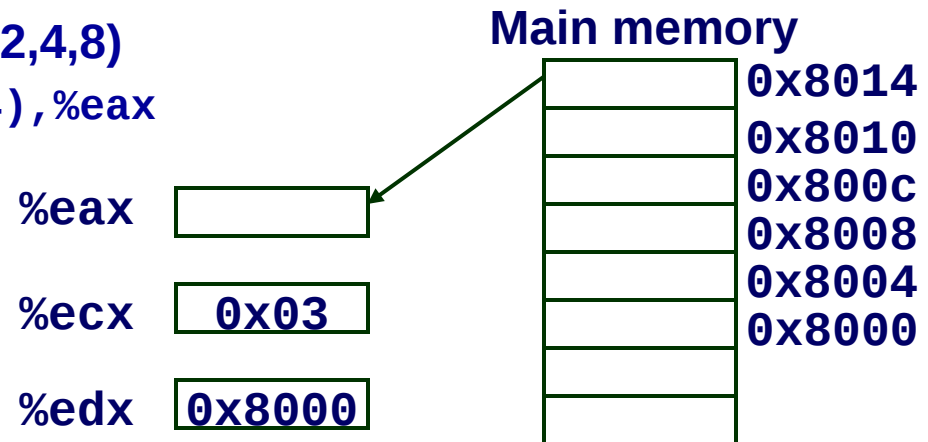
- Form: $\text{Imm}(E_b, E_i)$
- Operand value: $M[\text{Imm} + R[E_b] + \text{Reg}[E_i]]$
 - Register E_b specifies start of memory region
 - Register E_i holds index
 - `movl 4(%ecx,%edx),%eax`



Memory modes

Memory mode: Scaled indexed

- Most general format
- Form: $\text{Imm}(E_b, E_i, S)$
- Operand value: $M[\text{Imm} + R[E_b] + S * R[E_i]]$
 - Register E_b specifies start of memory region
 - E_i holds index
 - S is integer scale (1,2,4,8)
 - `movl 8(%edx,%ecx,4),%eax`



Memory modes

Memory mode: Scaled indexed

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

Summary of IA32 Operand Forms

Type	Form	Operand Value	Name
Immediate	\$Imm	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] * s]$	Scaled Indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] * s]$	Scaled Indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] * s]$	Scaled Indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] * s]$	Scaled Indexed

Addressing Mode Examples

`addl 12(%ebp),%ecx`

Add the long at address
`ebp + 12` to `ecx`

`movb (%eax,%ecx),%dl`

Load the byte at address
`eax + ecx` into `dl`

`subl %edx, (%ecx,%eax, 4)`

Subtract `edx` from the long at
address `ecx+(4*eax)`

`incl 0xA(,%ecx, 8)`

Increment the long at address
`0xA+(8*ecx)`

Note: “long” refers to 4-bytes in gas assembly

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

Practice Problem 3.1

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Operand	Value
%eax	
0x104	
\$0x108	
(%eax)	
4(%eax)	
9(%eax, %edx)	
260(%ecx, %edx)	
0xFC(, %ecx, 4)	
(%eax, %edx, 4)	

Practice Problem 3.1

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Operand	Value
%eax	0x100
0x104	0xAB
\$0x108	0x108
(%eax)	0xFF
4(%eax)	0xAB
9(%eax, %edx)	0x11
260(%ecx, %edx)	0x13
0xFC(, %ecx, 4)	0xFF
(%eax, %edx, 4)	0x11

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

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
} Finish
```

What are these things “int *xp” and “int *yp” ?

Why do we have to use them instead of just int x, y; ?

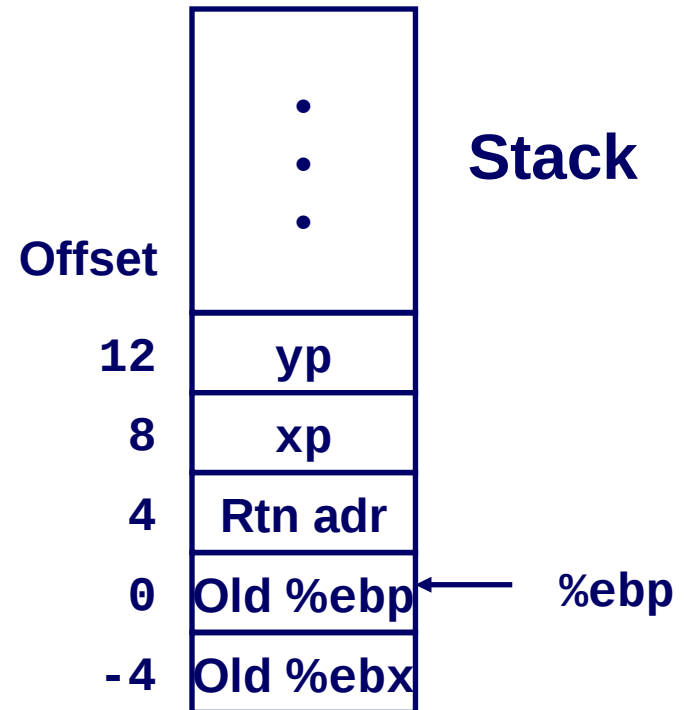
•Pass by reference

Understanding Swap

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

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

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



Practice Problem 3.5

A function has this prototype:

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

Here is the body of the code in assembly language:

```
movl 16(%ebp),%esi
movl 12(%ebp),%ebx
movl 8(%ebp),%edi
movl (%edi),%eax
movl (%ebx),%edx
movl (%esi),%ecx
movl %eax, (%ebx)
movl %edx, (%esi)
movl %ecx, (%edi)
```

Write C code for this function

Practice Problem 3.5

A function has this prototype:

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

Here is the body of the code in assembly language:

```
movl 16(%ebp),%esi
movl 12(%ebp),%ebx
movl 8(%ebp),%edi
movl (%edi),%eax
movl (%ebx),%edx
movl (%esi),%ecx
movl %eax,(%ebx)
movl %edx,(%esi)
movl %ecx,(%edi)
```

Write C code for this function

```
void decode(int *xp, int *yp, int *zp) {
    int t0=*xp;
    int t1=*yp;
    int t2=*zp;
    *yp=t0;
    *zp=t1;
    *xp=t2;
    return t0;
}
```

Practice Problem

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

```
int array[34];
```

Write some assembly code that:

- sets `esi` to the address of array
- sets `ebx` to the constant 9
- loads `array[9]` into register `eax`.

Use scaled index memory mode

Practice Problem

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

```
int array[34];
```

Write some assembly code that:

- sets `esi` to the address of `array`
- sets `ebx` to the constant 9
- loads `array[9]` into register `eax`.

Use scaled index memory mode

```
movl $array,%esi  
movl $0x9,%ebx  
movl (%esi,%ebx,4),%eax
```

Comparing Instructions

Not all 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`, `%eax = 0x98765432`

<code>movb</code>	<code>%dh, %al</code>	<code>%eax = 0x987654</code> 8D
<code>movsbl</code>	<code>%dh, %eax</code>	<code>%eax = 0x</code> FFFFFF <code>8D</code>
<code>movzbl</code>	<code>%dh, %eax</code>	<code>%eax = 0x</code> 000000 <code>8D</code>

Data Movement Instructions

Instruction		Effect	Description
movl	S, D	$D \leftarrow S$	Move double word
movw	S, D	$D \leftarrow S$	Move word
movb	S, D	$D \leftarrow S$	Move byte
movsbl	S, D	$D \leftarrow \text{SignExtend}(S)$	Move sign-extended byte
movzbl	S, D	$D \leftarrow \text{ZeroExtend}(S)$	Move zero-extended byte

Arithmetic and Logical Operations

Load address

Load Effective Address (Long)

- $\text{leal } S, D \Rightarrow D \leftarrow \&S$
- Loads the *address* of *S* in *D*, not the *contents*
 - `leal (%eax),%edx`
 - Equivalent to `movl %eax,%edx`
- Destination must be a register
 - Commonly used by compiler to do simple arithmetic
 - If $\%edx = x$,
 - » `leal 7(%edx, %edx, 4), %edx $\Rightarrow 5x + 7$`
 - » Multiply and add all in one instruction

Practice Problem 3.6

`%eax = x, %ecx = y`

Expression	Result in %edx
<code>leal 6(%eax), %edx</code>	
<code>leal (%eax, %ecx), %edx</code>	
<code>leal (%eax, %ecx, 4), %edx</code>	
<code>leal 7(%eax, %eax, 8), %edx</code>	
<code>leal 0xA(, %ecx, 4), %edx</code>	
<code>leal 9(%eax, %ecx, 2), %edx</code>	
<code>leal (%ecx, %eax, 4), %edx</code>	
<code>leal 1(, %eax, 2), %edx</code>	

Practice Problem 3.6

`%eax = x, %ecx = y`

Expression	Result in %edx
<code>leal 6(%eax), %edx</code>	$x+6$
<code>leal (%eax, %ecx), %edx</code>	$x+y$
<code>leal (%eax, %ecx, 4), %edx</code>	$x+4y$
<code>leal 7(%eax, %eax, 8), %edx</code>	$9x+7$
<code>leal 0xA(, %ecx, 4), %edx</code>	$4y+10$
<code>leal 9(%eax, %ecx, 2), %edx</code>	$x+2y+9$
<code>leal (%ecx, %eax, 4), %edx</code>	$4x+y$
<code>leal 1(, %eax, 2), %edx</code>	$2x+1$

Unary Operations

Unary \Rightarrow one operand

`inc` - increment $\Rightarrow D \leftarrow D + 1$

`dec` - decrement $\Rightarrow D \leftarrow D - 1$

`neg` - negate $\Rightarrow D \leftarrow -D$

`not` - complement $\Rightarrow D \leftarrow \sim D$

Examples

`incl (%esp)`

- Increment 32-bit quantity at top of stack

`notl %eax`

- Complement 32-bit quantity in register `%eax`

Binary Operations

A little bit tricky

- The second operand is used as both a source and destination
- A bit like C operators '+=', '-=', etc.

Format

- `<op> S, D` $\Rightarrow D = D <op> S$

Can be confusing

- `subl S, D` $\Rightarrow D = D - S$
- Not $S - D$!! Be careful

Examples

`add S, D` $\Rightarrow D = D + S$

`sub S, D` $\Rightarrow D = D - S$

`xor S, D` $\Rightarrow D = D \wedge S$

`or S, D` $\Rightarrow D = D \mid S$

`and S, D` $\Rightarrow D = D \& S$

Integer Multiply

`imull <operand>`

`imul` = signed multiply

`mul` = unsigned multiply

- Many forms based on 8-bit, 16-bit, or 32-bit
 - Binary and unary forms supported
 - Unary form
 - » One operand assumed to be in `eax`
 - » Or in `al` or `ax` for the shorter forms
 - 64-bit result in `edx:eax`
 - See the programmer's reference manual

Integer Divide

`idivl <operand>`

- `idiv` = signed division
- `div` = unsigned division
- Many forms based on 8-bit, 16-bit, or 32-bit
 - Binary and unary forms supported
 - Unary form
 - » Dividend assumed to be `edx:eax`
 - » Use `cld` to sign-extend 32-bit value in `eax` into `edx` for `idivl`
 - Divisor specified by operand
 - Quotient stored in `eax`
 - Remainder stored in `edx`
 - See the programmer's reference manual

Instruction	Effect
<code>imull S</code>	$R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax]$; signed
<code>mull S</code>	$R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax]$; unsigned
<code>cld</code>	$R[\%edx]:R[\%eax] \leftarrow \text{SignExtend}(R[\%eax])$
<code>idivl S</code>	$R[\%edx] \leftarrow R[\%edx]:R[\%eax] \bmod S$; signed $R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S$
<code>divl S</code>	$R[\%edx] \leftarrow R[\%edx]:R[\%eax] \bmod S$; unsigned $R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S$

$R[\%edx]:R[\%eax]$ is viewed as a 64-bit quad word

Practice Problem 3.7

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination address	Result
addl %ecx, (%eax)		
subl %edx, 4(%eax)		
imull \$16, (%eax, %edx, 4)		
incl 8(%eax)		
decl %ecx		
subl %edx, %eax		
xorl %eax, 4(%eax)		

Practice Problem 3.7

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination address	Result
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull \$16, (%eax, %edx, 4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx, %eax	%eax	0xFD
xorl %eax, 4(%eax)	0x104	0x1AB

Logical Operators in C

Bitwise operators == assembly instructions

`&` => `andl`

`|` => `orl`

`^` => `xorl`

`~` => `notl`

Expression logical operators give “true” or “false” result

- `&&`, `||`, `!`

- False is integer 0, true is integer non-zero

`cmpl`, `testl` plus condition code registers (more later)

Shift Operations

Arithmetic and logical shifts are possible

<op> amount value

`sal k, D` $\Rightarrow D = D \ll k$

`shl k, D` $\Rightarrow D = D \ll k$

`sar k, D` $\Rightarrow D = D \gg k$, sign extend

`shr k, D` $\Rightarrow D = D \gg k$, zero fill

Max shift is 32 bits, so k is either an immediate byte, or register (e.g. %c1)

`%c1` is byte 0 of register `%ecx`

Practice Problem 3.8

```
int shift_left_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```



```
_shift_left2_rightn:
    pushl   %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %eax    ; get x
    movl   12(%ebp), %ecx   ; get n
    _____ ; x <<= 2;
    _____ ; x >>= n;
    popl   %ebp
    ret
```

Practice Problem 3.8

```
int shift_left_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```



```
_shift_left2_rightn:
    pushl   %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %eax    ; get x
    movl   12(%ebp), %ecx   ; get n
    sall $2, %eax          ; x <<= 2;
    sarl %cl, %eax        ; x >>= n;
    popl   %ebp
    ret
```

Compiler “Tricks”

The compiler will try to generate efficient code

- Resultant assembly code may not readily map to C code, but is functionally the same

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    return t4;
}
```

```
movl 12(%ebp), %eax
movl 16(%ebp), %edx
addl 8(%ebp), %eax
leal (%edx, %edx, 2), %edx
sall $4, %edx
andl $65535, %eax
imull %eax, %edx
movl %edx, %eax
```

Practice Problem 3.10

What does this xorl do?

```
int junk(int n)
{
    int i, v=0;

    for (i=0; i < n; i++)
        v += i;

    return v;
}
```

```
_junk:
    pushl %ebp
    xorl %eax, %eax
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    xorl %edx, %edx
    cmpl %ecx, %eax
    jge L8
    .p2align 4,,15
L6:
    addl %edx, %eax
    incl %edx
    cmpl %ecx, %edx
    jl L6
L8:
    popl %ebp
    ret
```

Disassembling Object Code

Disassembled

```
00401040 <_sum>:  
  0:  55                push   %ebp  
  1:  89 e5             mov    %esp,%ebp  
  3:  8b 45 0c          mov    0xc(%ebp),%eax  
  6:  03 45 08          add   0x8(%ebp),%eax  
  9:  89 ec             mov    %ebp,%esp  
 b:  5d                pop    %ebp  
 c:  c3                ret  
 d:  8d 76 00          lea   0x0(%esi),%esi
```

Disassembler

`objdump -d <object_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 relocatable (.o) file

Using gdb to disassemble

Object

0x401040:
0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x89
0xec
0x5d
0xc3

Disassembled

```
0x401040 <sum>:      push    %ebp
0x401041 <sum+1>:      mov     %esp,%ebp
0x401043 <sum+3>:      mov     0xc(%ebp),%eax
0x401046 <sum+6>:      add     0x8(%ebp),%eax
0x401049 <sum+9>:      mov     %ebp,%esp
0x40104b <sum+11>:     pop     %ebp
0x40104c <sum+12>:     ret
0x40104d <sum+13>:     lea    0x0(%esi),%esi
```

Within gdb Debugger

```
gdb p
disassemble sum
■ Disassemble procedure
x/13b sum
■ Examine the 13 bytes starting at sum
```

Where can you look this stuff up?

Is there a manual for assembly language?

Intel Architecture Software Developer's Manual,

- Vol 2: Instruction Set Reference
- A complete reference to the machine instruction set
- Some indication of the assembly language also
- Intel publishes an assembly language manual

For Linux, gas assembler

- Go to <http://linuxassembly.org> and follow link to docs
- Start with existing assembly code and modify it

Extra slides

Extended Example: simple.c

```
gcc -O2 -c simple.c
```

```
int simple(int *xp, int y)
{
    int t = *xp + y;
    *xp = t;
    return t;
}
```

`_simple:`

```
    pushl   %ebp                Setup stack frame pointer
    movl    %esp, %ebp
    movl    8(%ebp), %edx        get xp
    movl    12(%ebp), %ecx      get y
    movl    (%edx), %eax        move *xp to t
    addl    %ecx, %eax          add y to t
    movl    %eax, (%edx)        store t at *xp
    popl    %ebp                restore frame pointer
    ret                                return to caller
```

Why do we use Linux instead of Windows?

It's free.

The source code is available to us

The compilers and other tools are available to us

An excellent software development environment

- **Thousands of open-source projects**

Practice

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

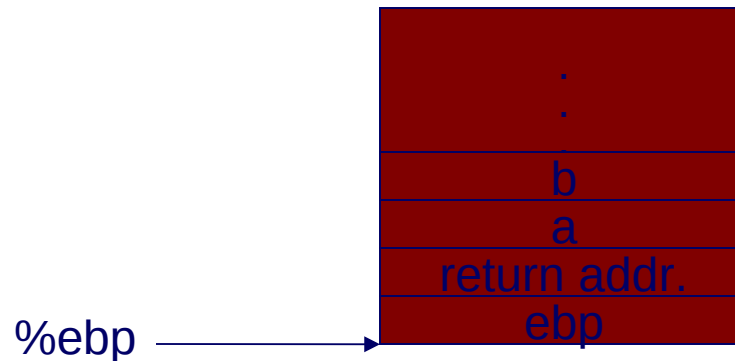
Instruction	Destination	Value
addw %cx, (%eax)		
subb %dl, 4(%eax)		
sarb %cl, (%eax)		
sarl \$2, (%eax)		
shrb %cl, (%eax)		
shll \$8, %edx		

Example

Write an assembly routine that multiplies two 32-bit integers and returns the 64 bit result

- C prototype:
void product(int *a, int *b)
- Return high 32 bits in a and low 32 bits in b

Assume this stack setup



Example

Assume x is at %ebp+8, y at %ebp+12

```
movl 8(%ebp), %eax  
imull 12(%ebp)  
pushl %edx  
pushl %eax
```

```
movl 8(%ebp), %eax  
cld  
idivl 12(%ebp)  
pushl %eax  
pushl %edx
```