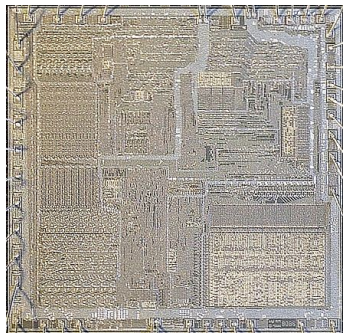
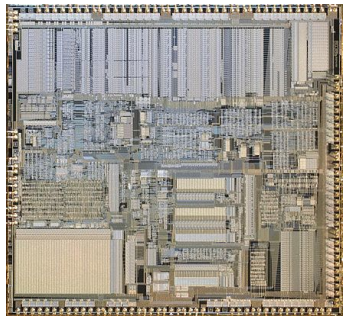
# IA32 Data Access and Operations

# Machine-Level Representations




## Prior lectures

- Data representation

## This lecture

- Program representation
- Encoding is architecture dependent
  - We will focus on the Intel “IA32” architecture
  - ISA = Instruction Set Architecture

## IA32

- Evolutionary design starting in 1978 with 8086
- i386 in 1986: First 32-bit Intel CPU
- Features added over time
  - There are many obsolete features and ugly baggage

## Complex Instruction Set Computer (CISC)

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

# How did we program computers?

**Initially, no compilers or assemblers**

**Machine code generated by hand!**

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

| Address | Instruction |
|---|---|
| 0000 0000 | 0000 1111 |
| 0000 0001 | 1000 0101 |
| 0000 0010 | 0011 1000 |
| 0000 0011 | 0000 1110 |
| 0000 0100 | 1000 0101 |
| 0000 0101 | 0011 0000 |
| 0000 0110 | 1011 0000 |
| 0000 0111 | 0000 1110 |
| 0000 1000 | 1000 0000 |
| 0000 1001 | 0010 0000 |
| 0000 1010 | 0000 1011 |
| 0000 1011 | 0001 0111 |
| 0000 1100 | 1000 1000 |
| 0000 1101 | 0001 0100 |
| 0000 1110 | 1110 1000 |
| 0000 1111 | 0000 0000 |
| 0001 0000 | 0001 0100 |
| 0001 0001 | 0000 1010 |
| 0001 0010 | 1000 0001 |
| 0001 0011 | 0001 0000 |
| 0001 0100 | 1010 1110 |

# Example Assembly Code

```
000e34           MemoryZero:
000e34 8b1f0004             load   [r15+4],r1          ! r1 = arg1 (p)
000e38 8b2f0008             load   [r15+8],r2          ! r2 = arg2 (byteCount)
000e3c           mzLoop1:                              ! LOOP:
000e3c 81020000             cmp    r2,0                !   if byteCount <= 0 exit
000e40 a500002c             ble    mzLoop2Test         !   .
000e44 88310003             and    r1,0x00000003,r3    !   tmp = p % 4
000e48 81030000             cmp    r3,0                !   if tmp == 0 exit
000e4c a2000020             be     mzLoop2Test         !   .
000e50 70010000             storeb r0,[r1]             !   *p = 0x00
000e54 80110001             add    r1,1,r1             !   p = p + 1
000e58 81220001             sub    r2,1,r2             !   byteCount = byteCount - 1
000e5c a1ffffe0             jmp    mzLoop1             ! ENDLOOP
```


*(Not Intel)*

# Assemblers

## Assign mnemonics to machine code

- Assembly language for specifying machine instructions
- Names for the machine instructions and registers

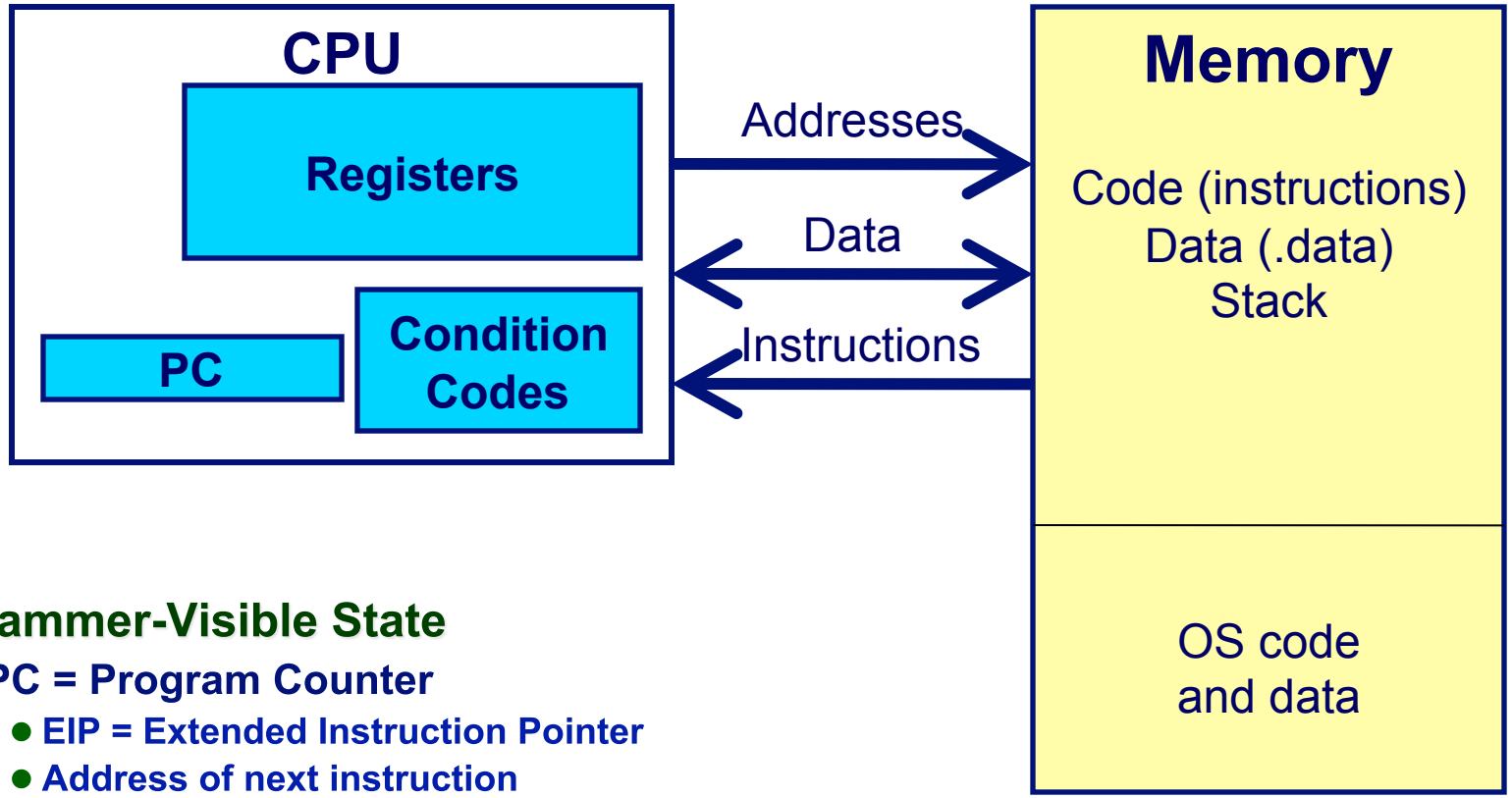
```
movl %eax,%ecx
```

- 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

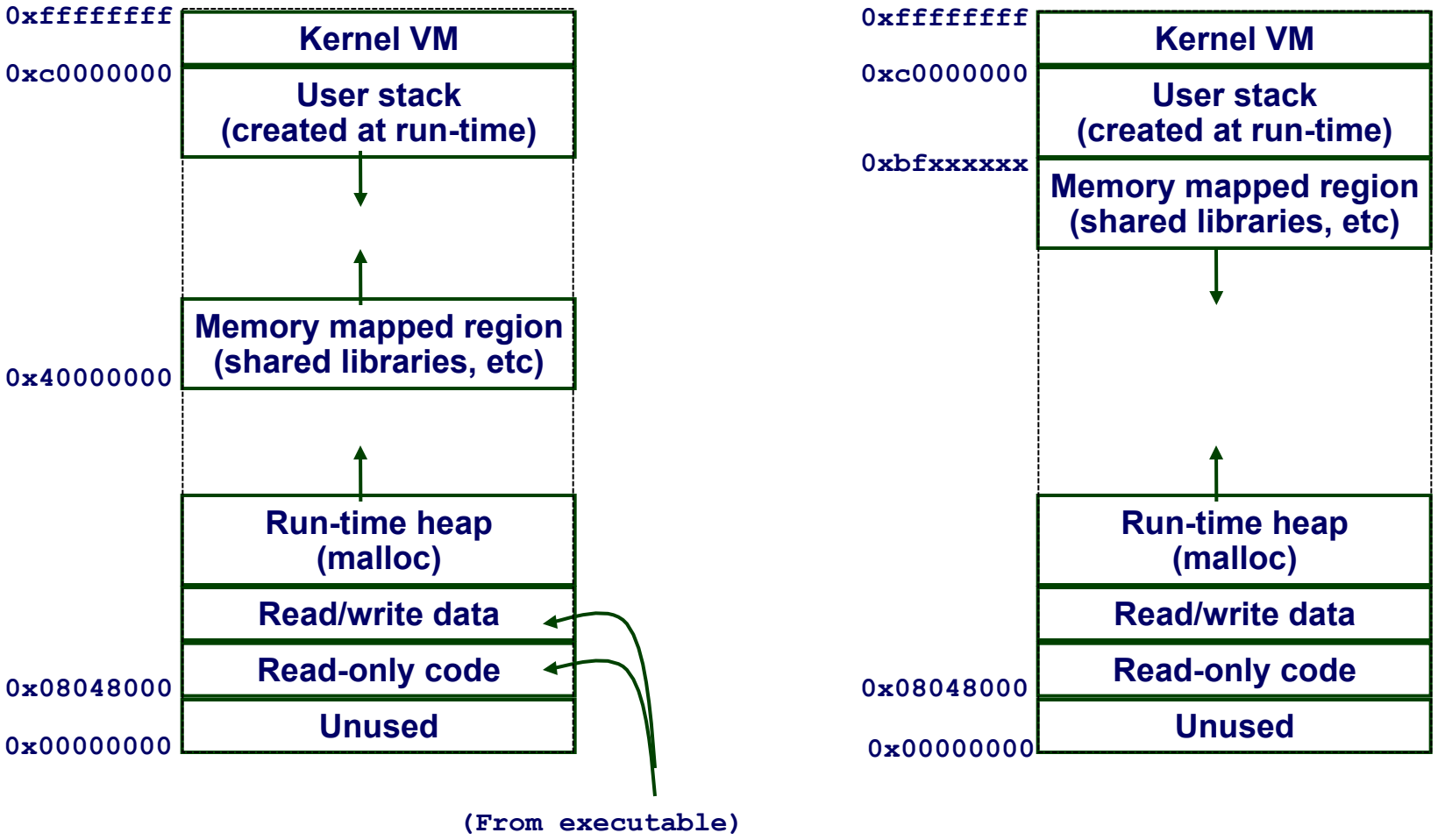
- PC = Program Counter
  - EIP = Extended Instruction Pointer
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

## Memory

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

# Virtual Memory (VM) abstraction

## Linux memory layout




`cat /proc/self/maps`

# Registers

## Located Directly on the CPU

- Instructions operate on data in registers
- Used to store frequently-used, temp data
- Very fast
- Not much storage capacity
  - IA32: 8 registers, 32-bits each
- Different from main memory
  - Main memory: e.g., 2 billion × 8 bits

## CPU does not directly operate on data in memory

- Data needs to be loaded into registers for CPU
- 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”

- There are not many general purpose registers
  - Initially, transistors were expensive
  - Then, “backward compatability”

    Certain 80186 instructions

    pusha (push all), popa (pop all)
- Other reasons
  - Speeds context-switching amongst processes

    (less register-state to store into main memory)
- Compiler complexity issues
- Now:
  - Fast caches (L1, L2, L3, etc.)
  - Have speeds between registers and main memory

# IA32 Registers

Number of registers: 8

Size of registers: 32 bits (=4 bytes)

Each has its own name (begins with %)

%eax

Can also access the low-order 16-bits

Using a different name

%ax

Can also access individual bytes

Using different names (e.g., %ah)

Some registers have special uses

%esp = the “stack pointer”

%ebp = the “frame base pointer”

# 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

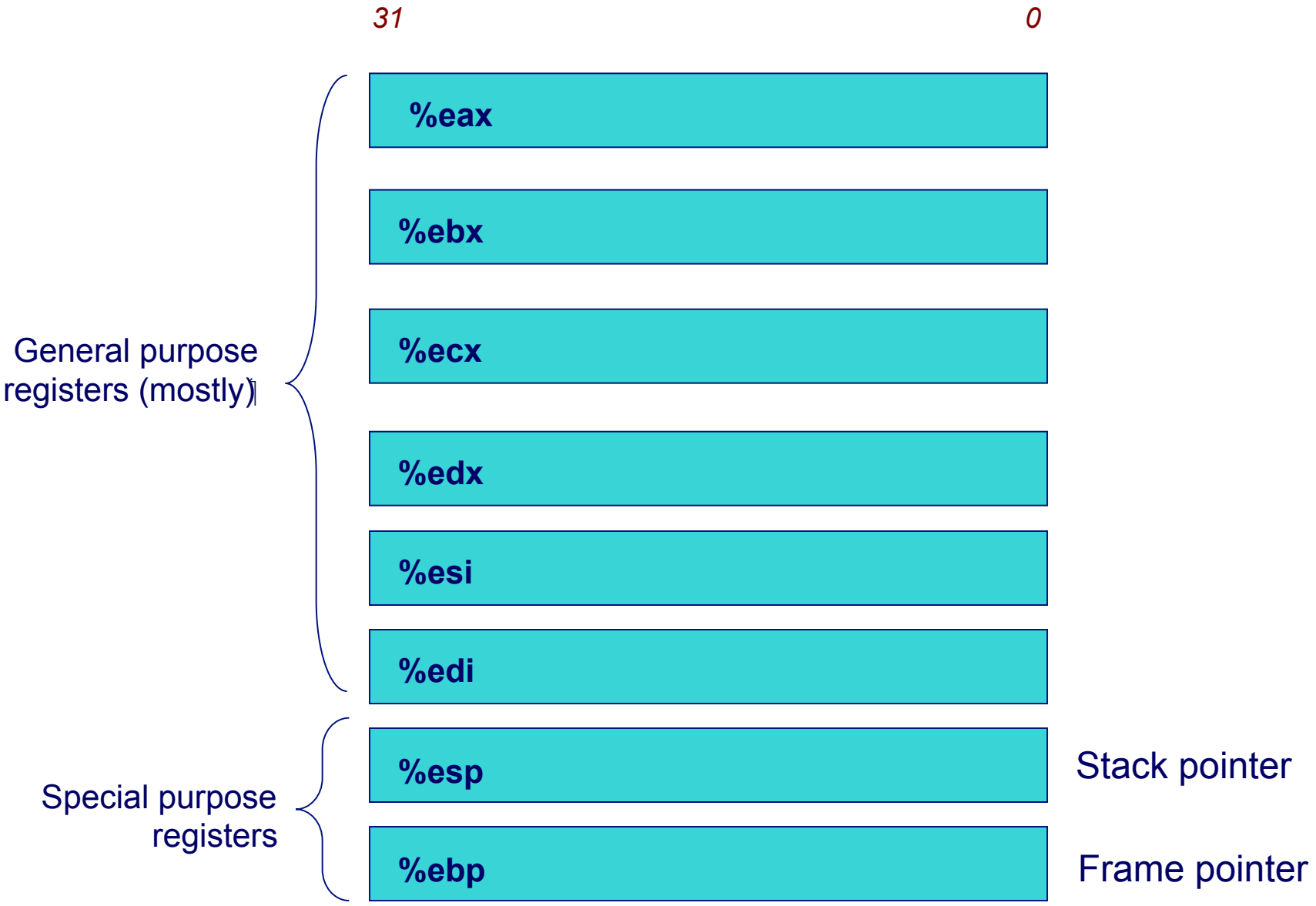
- The ‘e’ in the 32-bit names means “extended” (from 16 bit)
- %eax == 32 bits
- %ax == low 16 bits
- %ah == high byte of %ax
- %al == low byte of %ax

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

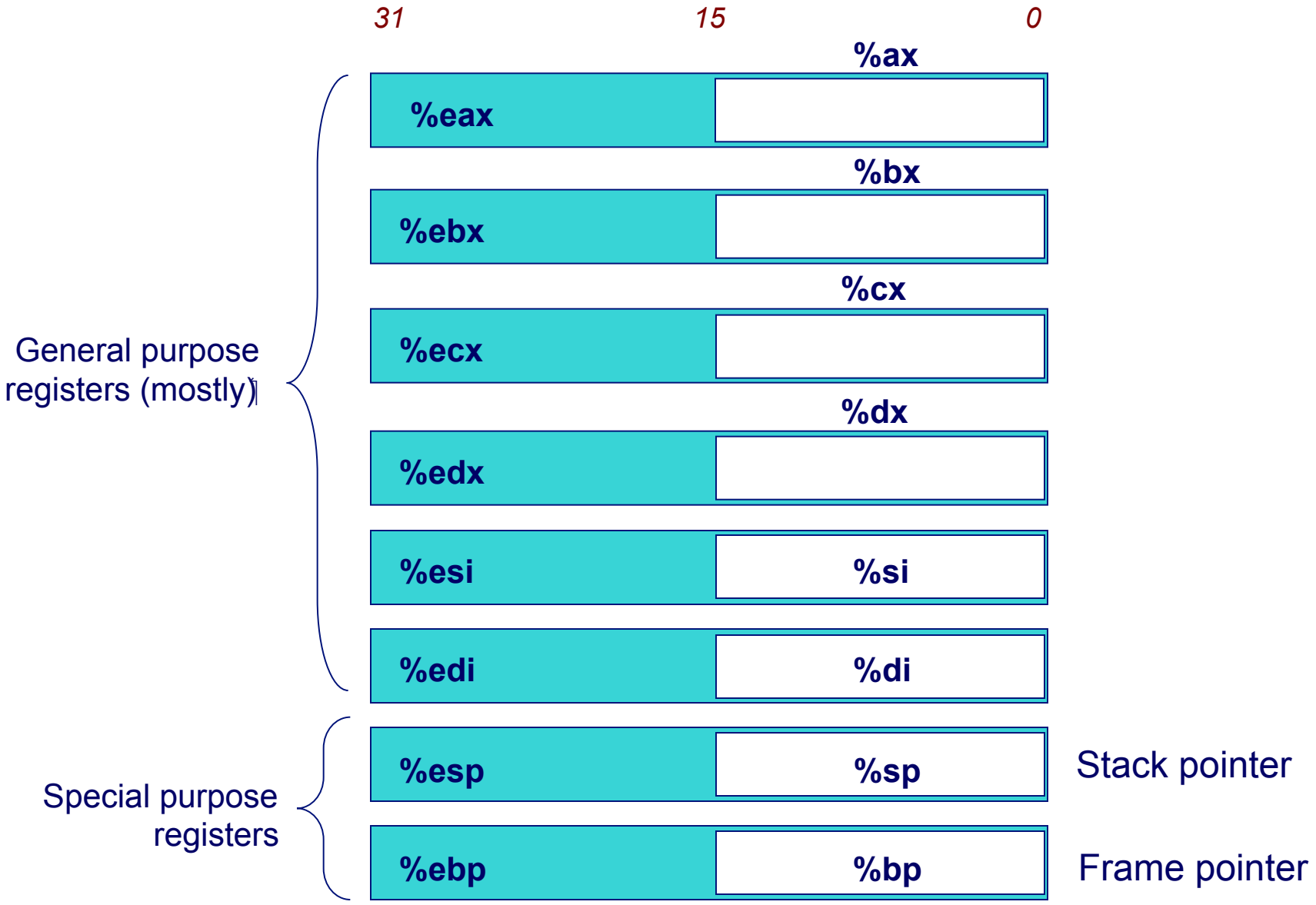
%esp and %ebp are the stack pointer and frame pointer, respectively

- We’ll study these later

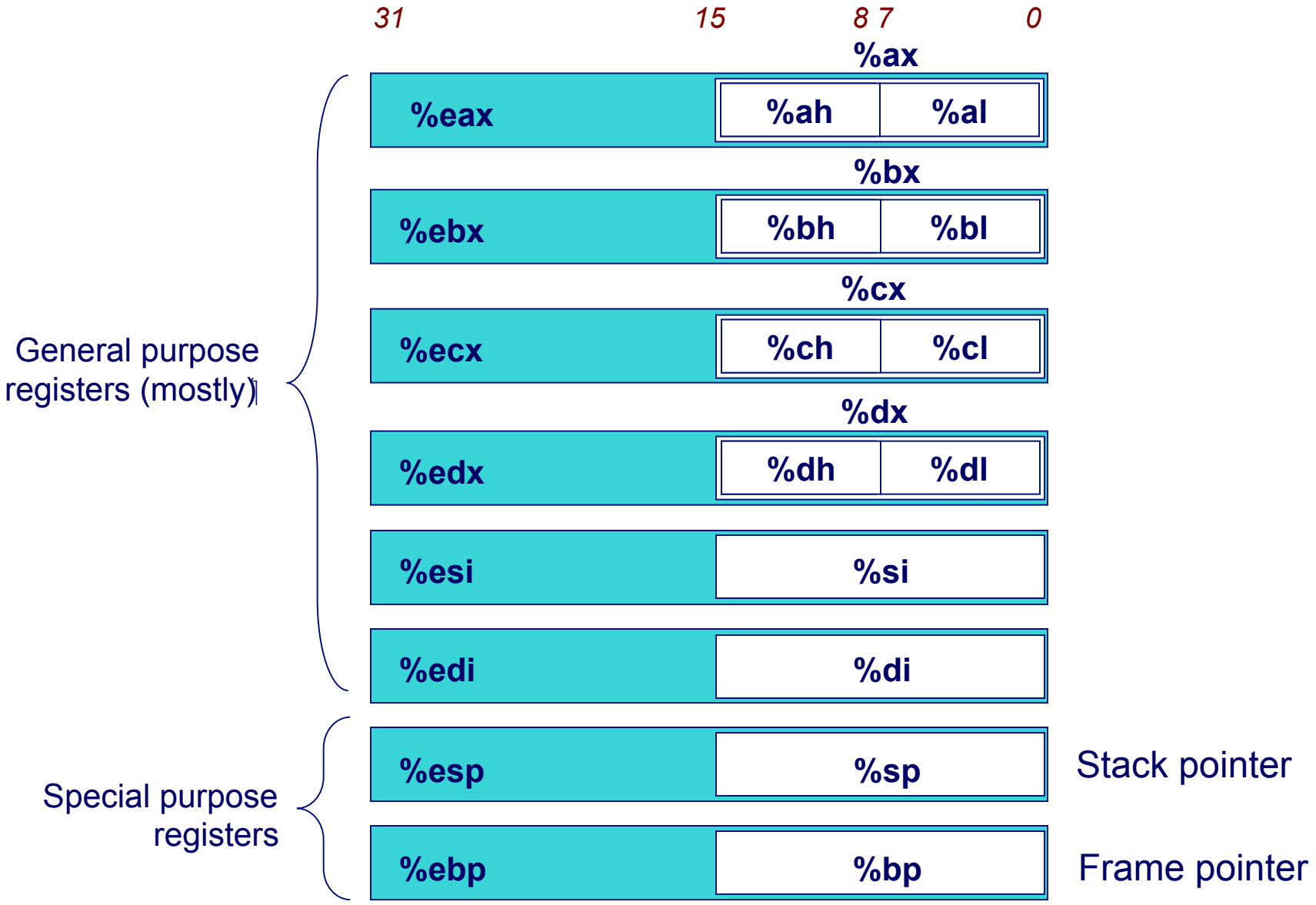
# IA32 General Registers



# IA32 General Registers



# IA32 General Registers



# Instruction types

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

```
addl    %ecx,%edx
```

- Ads the contents of ecx to edx

Size of the operand denoted in instruction

"addl" stands for add "long word" (= 32 bits!)

Why "long word" for 32-bit registers?

Baggage from 16-bit processors

**Now we have these terms**

- addb: 8 bits = byte
- addw: 16 bits = word !!!
- addl: 32 bits = double or long word !!!
- addq: 64 bits = 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 |

# Operand types

## Immediate

The data is included directly in the instrution

```
addl    $123,%eax
```

$0x4b, $-589

Will use 1,2, or 4 bytes in the instruction

## Register

The data is in a register

```
addl    $123,%eax
```

## Memory

The data is in memory

The address is specified in the instruction

Several different “addressing modes”

# Operand examples using `mov`

| | Source | Destination | | 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;` |

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

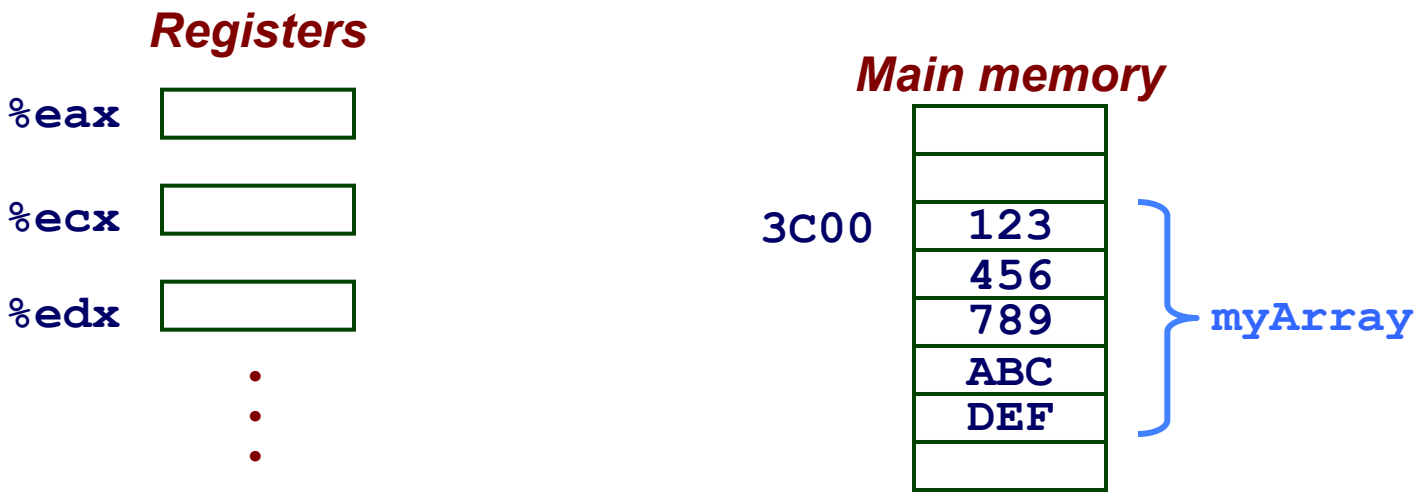
# Immediate mode

## Immediate has only one mode

- Form: *$ImmediateValue*
- Examples:

```
movl    $0x8000,%eax
movl    $myArray,%eax
```

```
int myArray[5];    /* global variable stored at 0x3C00 */
```



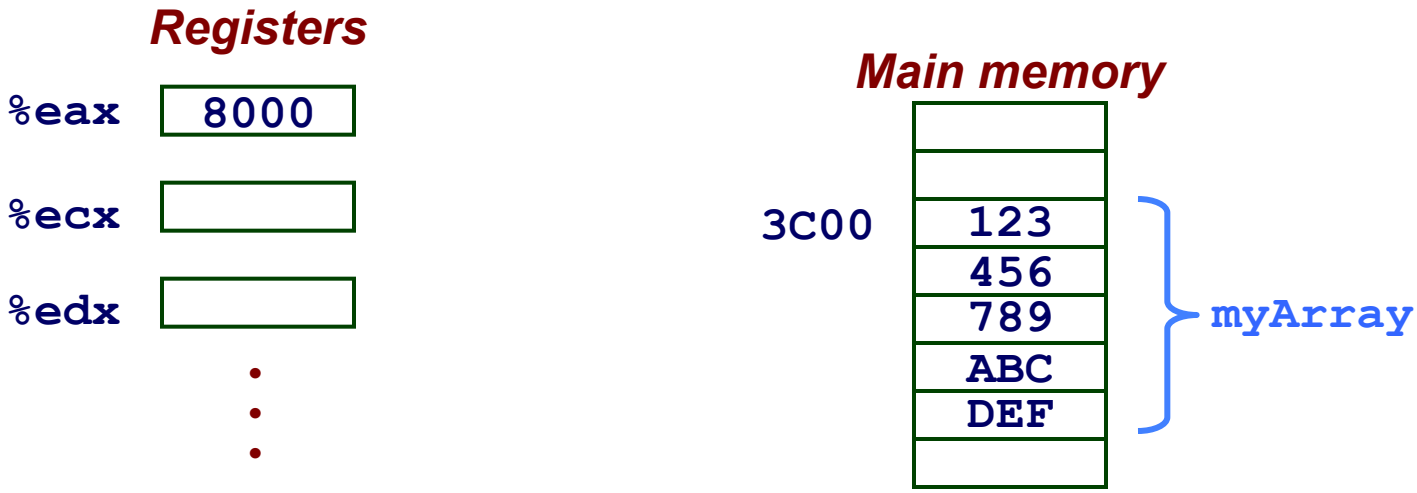
# Immediate mode

## Immediate has only one mode

- Form: *$ImmediateValue*
- Examples:

```
movl    $0x8000,%eax
movl    $myArray,%eax
```

```
int myArray[5];    /* global variable stored at 0x3C00 */
```



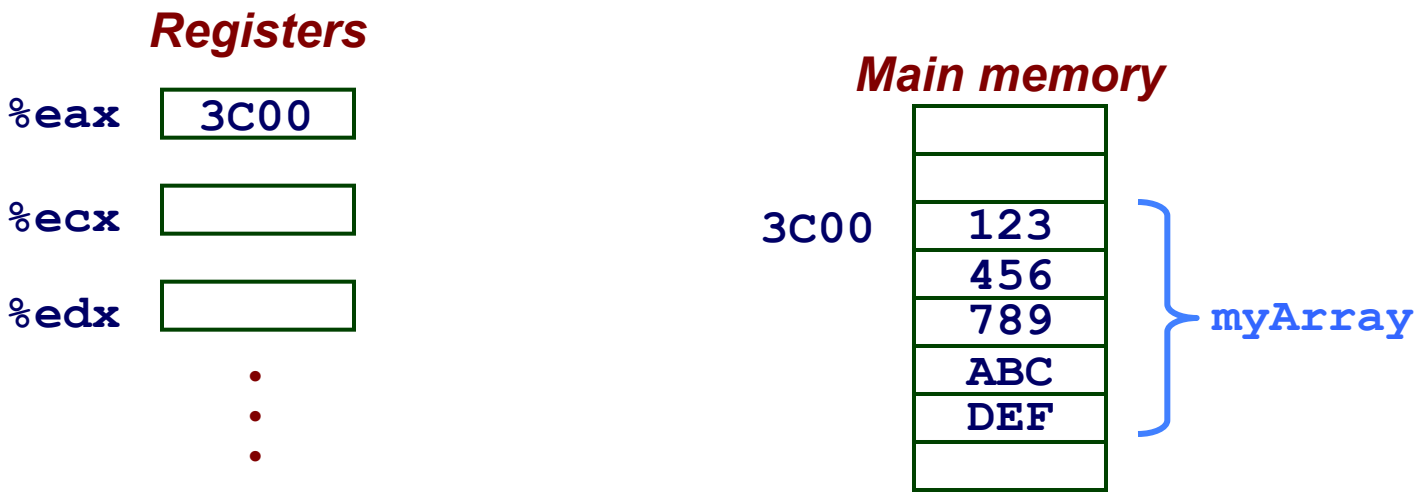
# Immediate mode

## Immediate has only one mode

- Form: *$ImmediateValue*
- Examples:

```
movl    $0x8000,%eax
movl    $myArray,%eax
```

```
int myArray[5];    /* global variable stored at 0x3C00 */
```

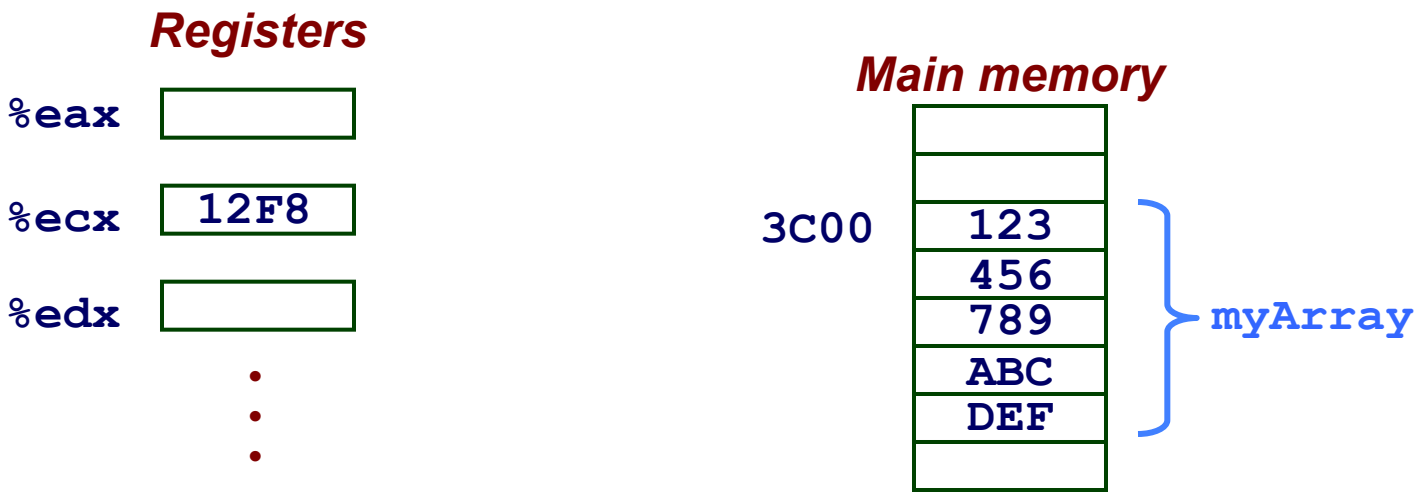


# Register mode

## Register has only one mode

- Form: $E_a$
- Operand value: $R[E_a]$

```
movl    %ecx,%eax
```

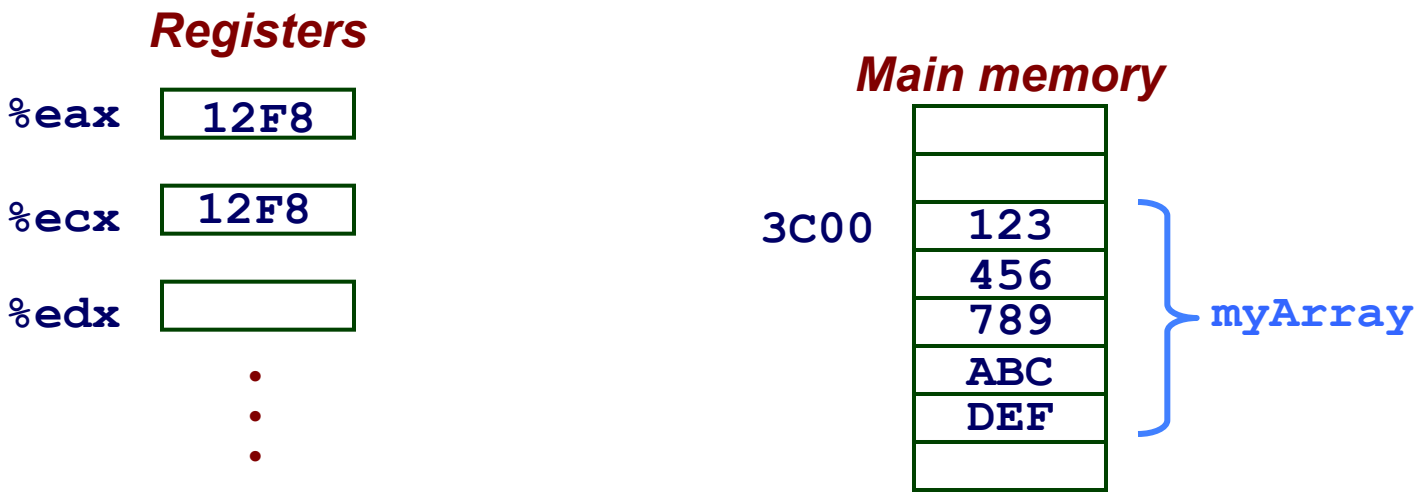


# Register mode

## Register has only one mode

- Form: $E_a$
- Operand value: $R[E_a]$

```
movl    %ecx,%eax
```



# Memory Modes

## Absolute

Specify the address of the data

```
movl    %eax,myArray
```

## Indirect

Use register to calculate address

```
movl    %eax,(%ebx)
```

## Base + displacement

Use register plus absolute address to calculate address

```
movl    %eax,32(%ebx)
```

## Indexed

Indexed: Add contents of an index register

```
movl    %eax,32(%ebx,%ecx)
```

Scaled index: Add contents of an index register scaled by a constant

```
movl    %eax,32(%ebx,%ecx,4)
```

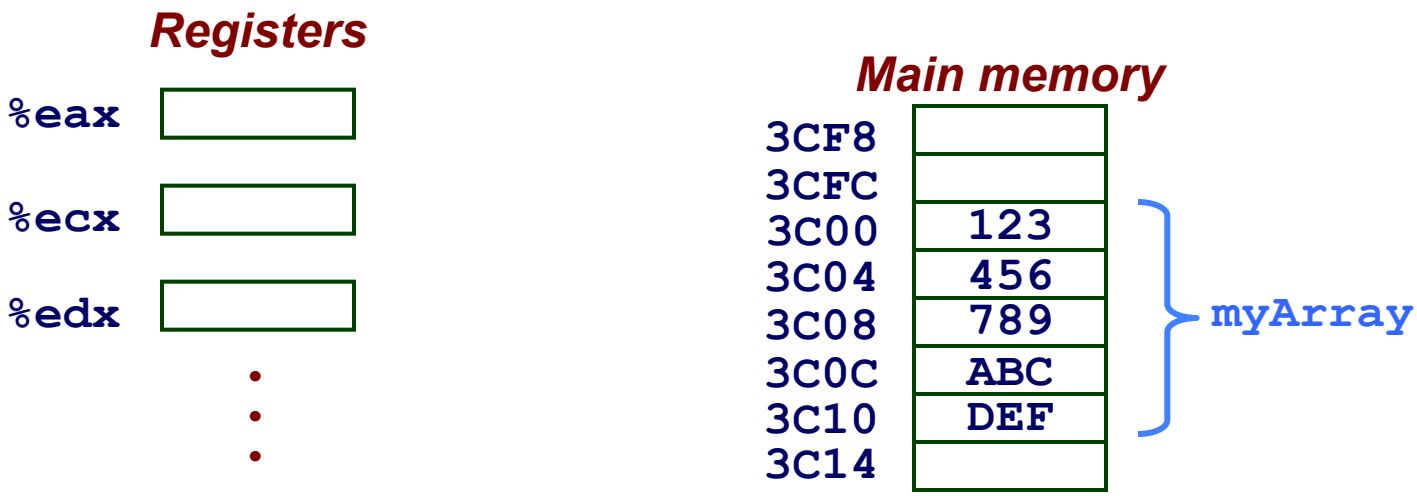
# Absolute Memory Mode

Form: `Imm`

Operand value: `M[Imm]`

```
movl 0x3C04,%eax
movl myArray,%eax
```

```
int myArray[30];   /* global variable stored at 0x3C00 */
```



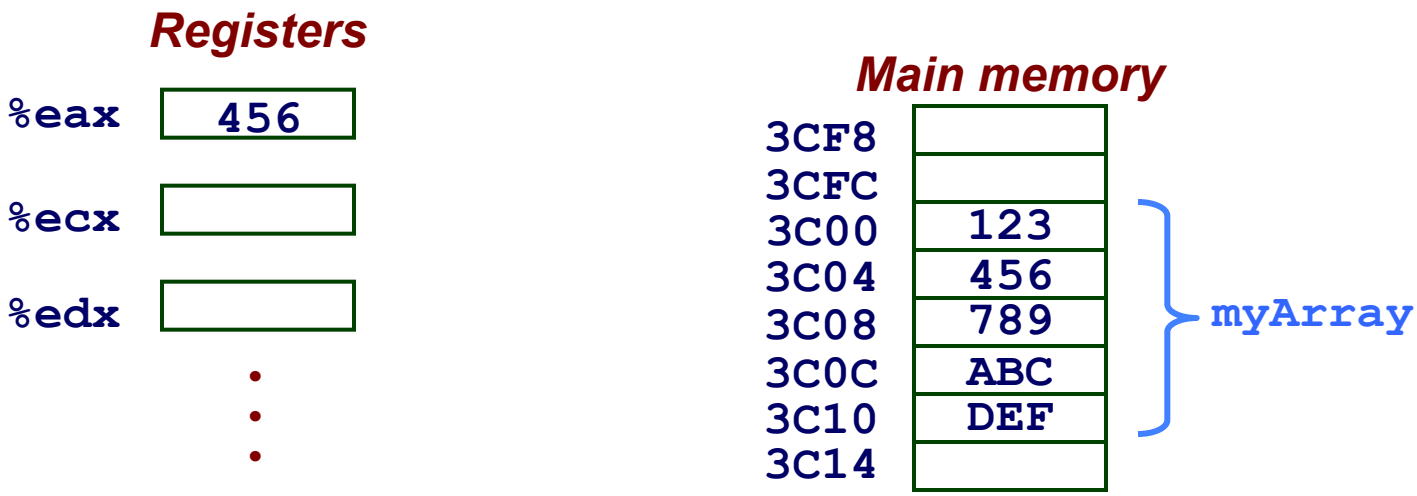
# Absolute Memory Mode

Form: `Imm`

Operand value: `M[Imm]`

```
movl 0x3C04,%eax
movl myArray,%eax
```

```
int myArray[30];   /* global variable stored at 0x3C00 */
```



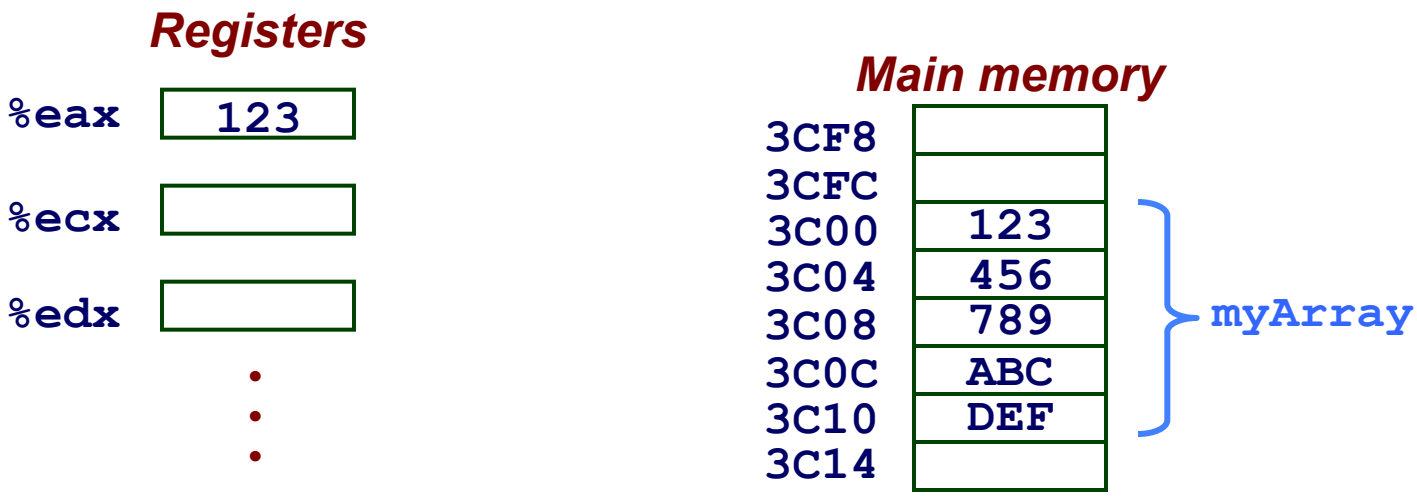
# Absolute Memory Mode

Form: `Imm`

Operand value: `M[Imm]`

```
movl 0x3C04,%eax
movl myArray,%eax
```

```
int myArray[30];   /* global variable stored at 0x3C00 */
```



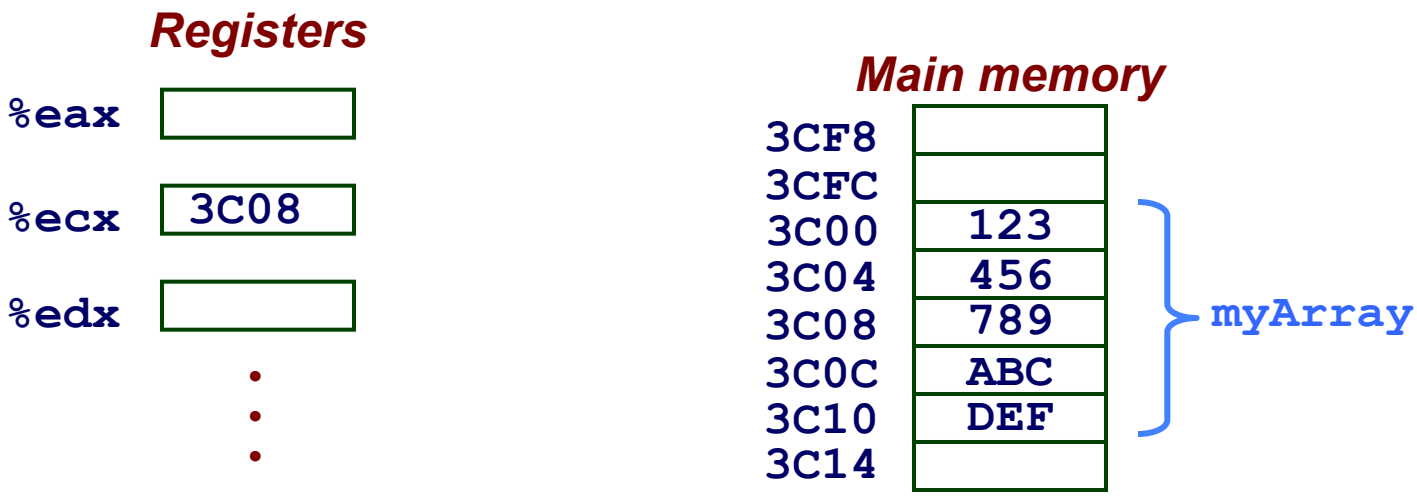
# Indirect Memory Mode

**Form:** $(E_a)$

**Operand value:** $M[R[E_a]]$

Register $E_a$ specifies the memory address

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



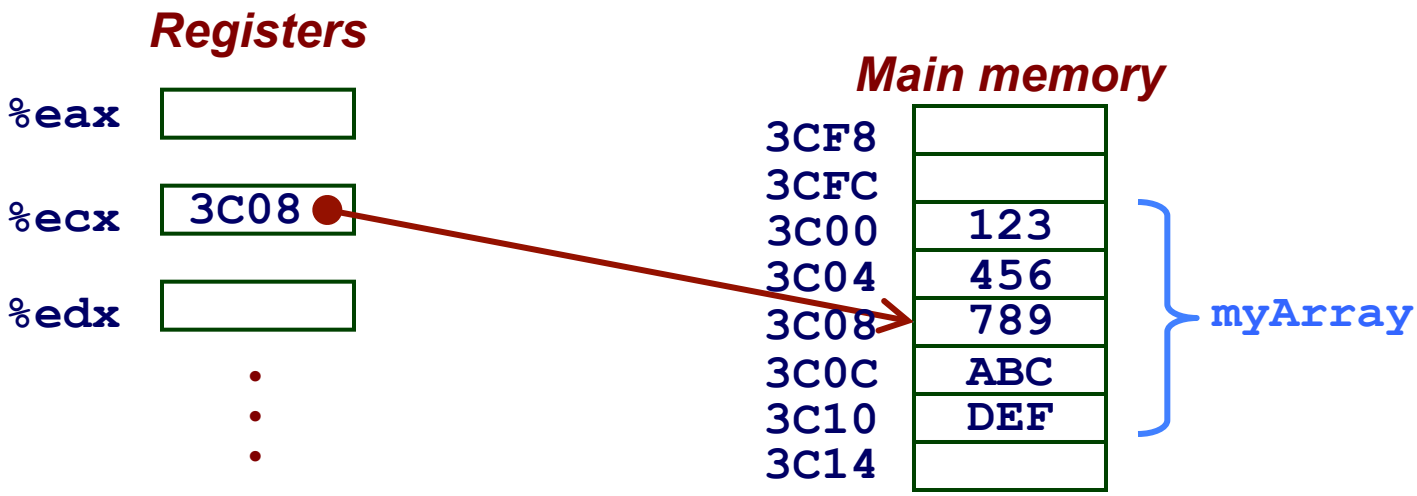
# Indirect Memory Mode

Form: $(E_a)$

Operand value: $M[R[E_a]]$

Register $E_a$ specifies the memory address

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



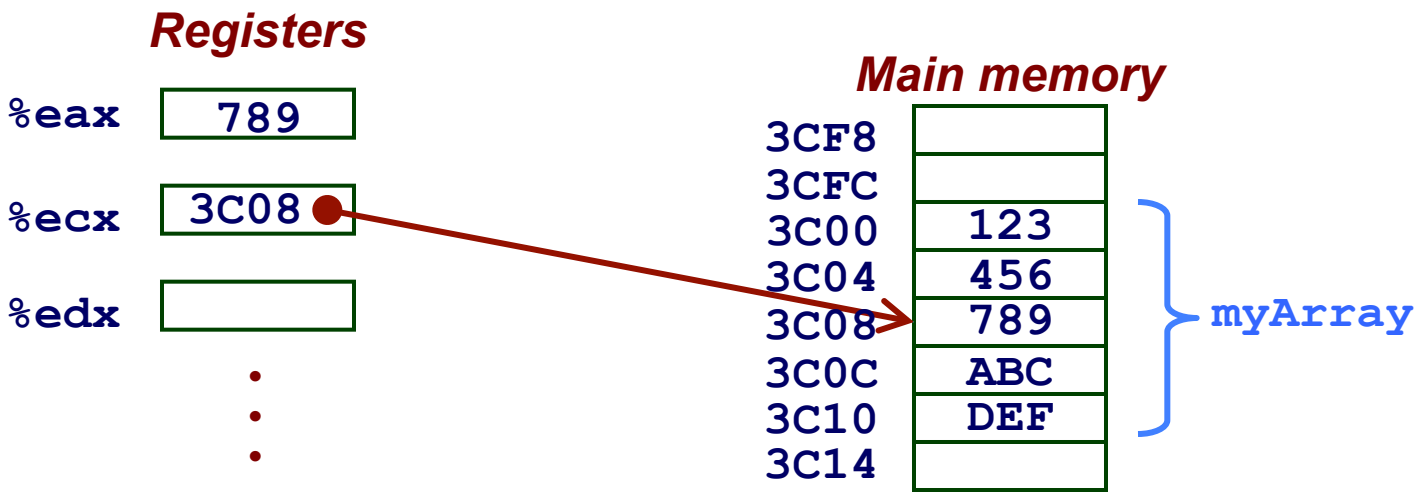
# Indirect Memory Mode

Form: $(E_a)$

Operand value: $M[R[E_a]]$

Register $E_a$ specifies the memory address

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



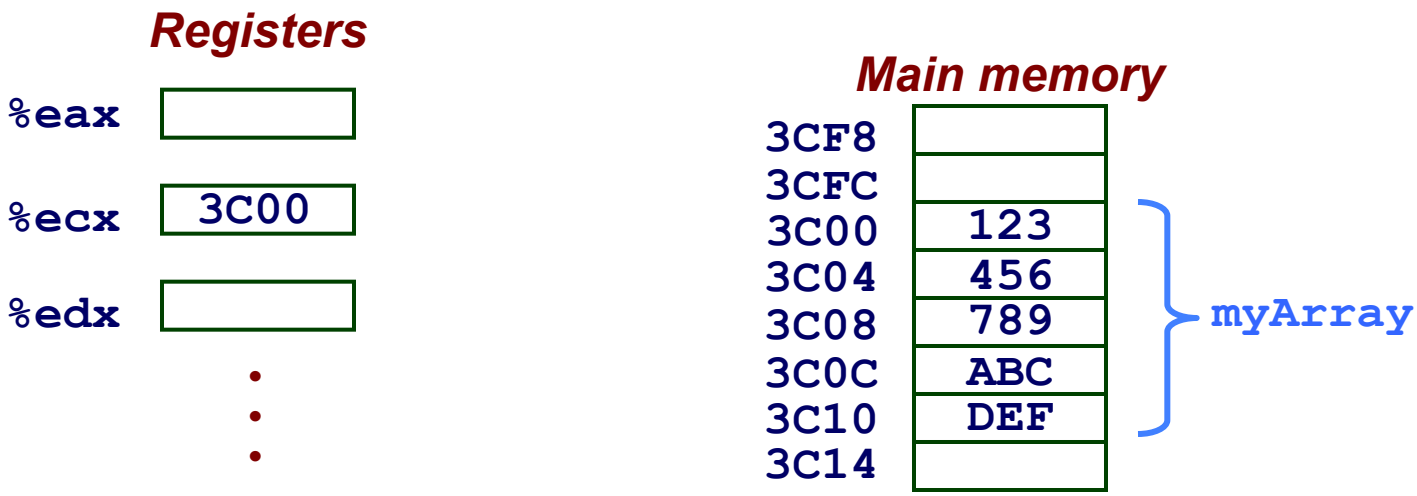
# Memory Mode: Base + Displacement

Form: `Imm(E_b)`

Operand value: `M[Imm+R[E_b]]`

- Register $E_b$ specifies start of memory region
- `Imm` specifies the offset/displacement

```
movl 8(%ecx),%eax
```



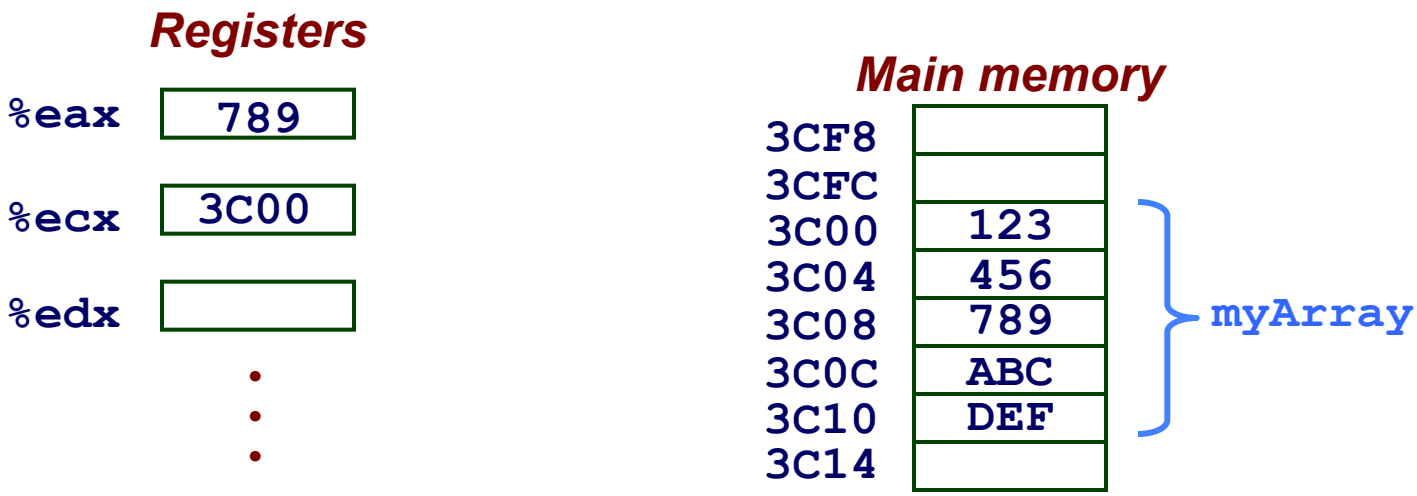
# Memory Mode: Base + Displacement

Form: `Imm(E_b)`

Operand value: `M[Imm+R[E_b]]`

- Register $E_b$ specifies start of memory region
- `Imm` specifies the offset/displacement

```
movl 8(%ecx),%eax
```



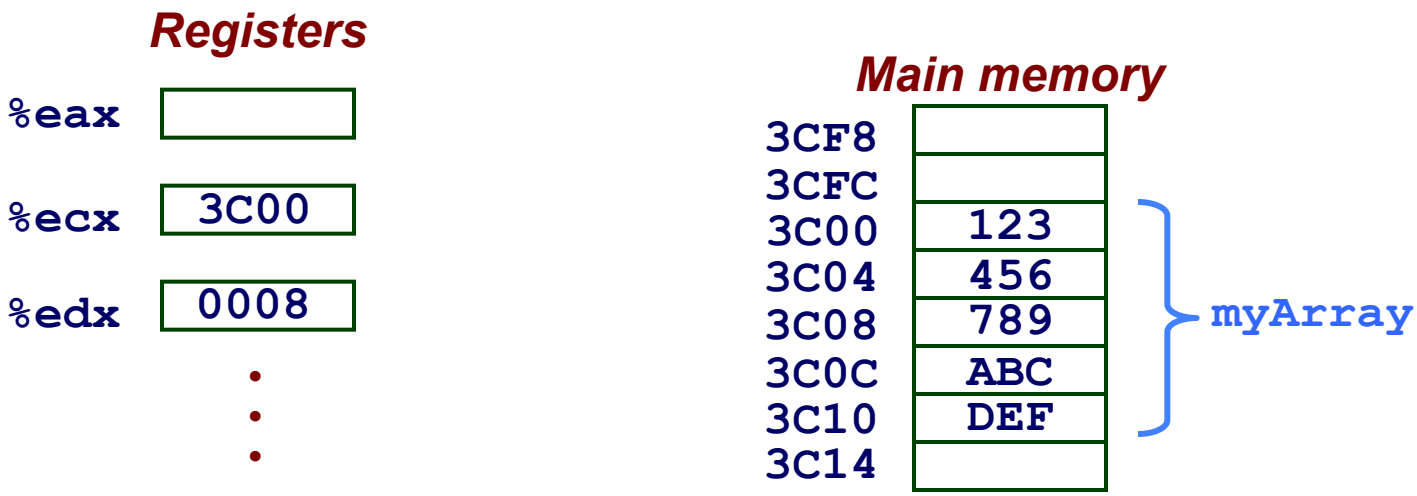
# Indexed Memory Mode

Form: `Imm(E_b,E_i)`

Operand value: `M[Imm+R[E_b]+Reg[E_i]]`

- Register $E_b$ specifies start of memory region
- Register $E_i$ holds index

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



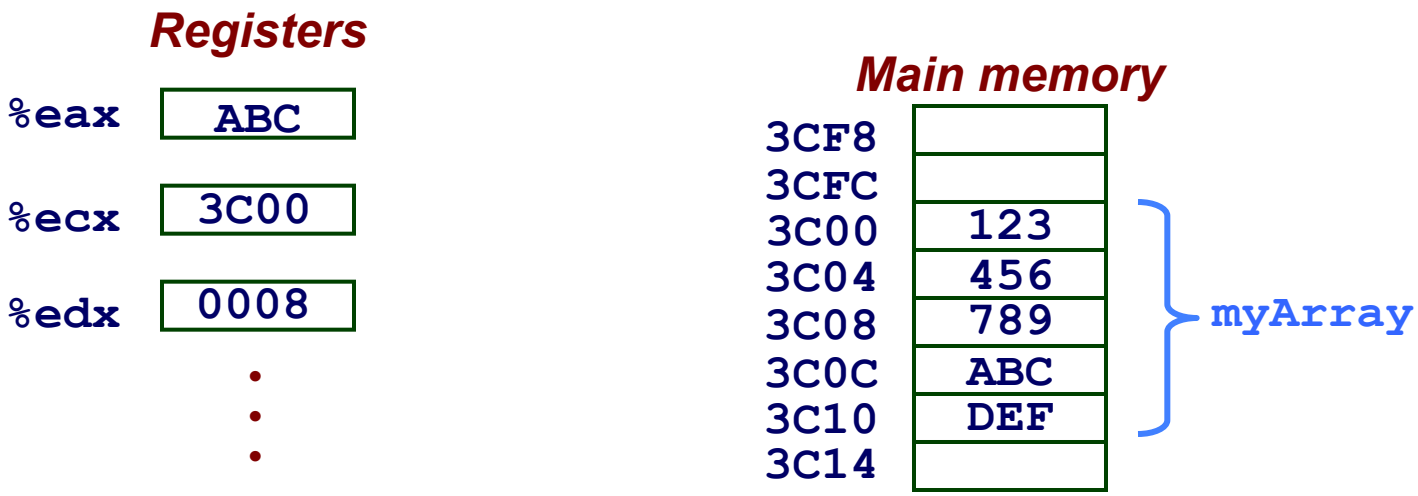
# Indexed Memory Mode

Form: `Imm(E_b,E_i)`

Operand value: `M[Imm+R[E_b]+Reg[E_i]]`

- Register $E_b$ specifies start of memory region
- Register $E_i$ holds index

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



# Memory Mode: Scaled Indexed

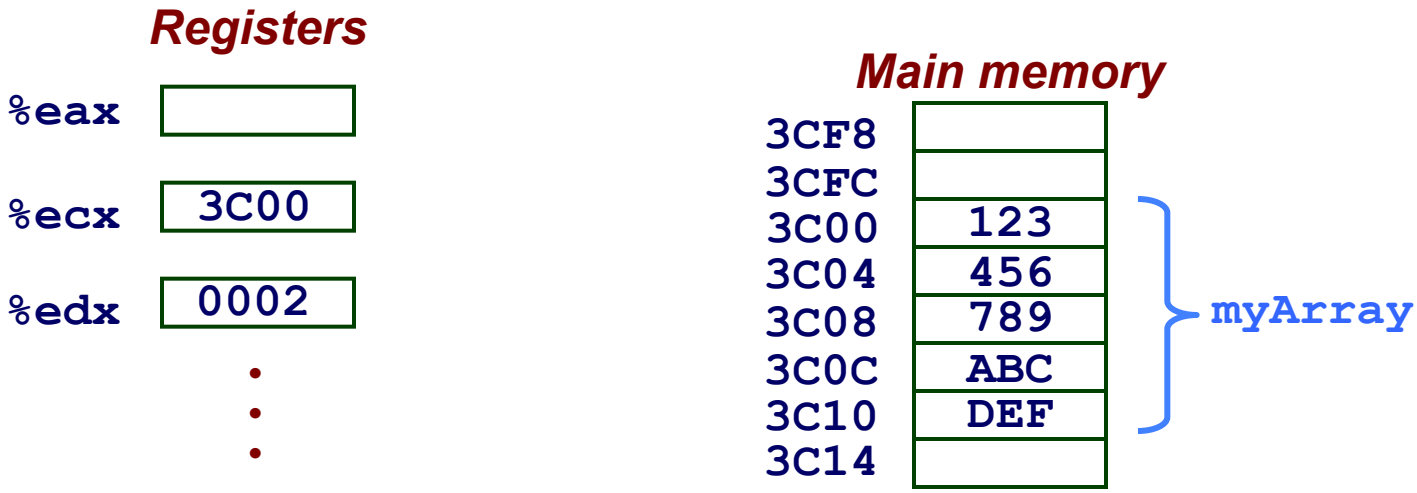
The most general format

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)

```
movl 8(%ecx,%edx,4),%eax
```



# Memory Mode: Scaled Indexed

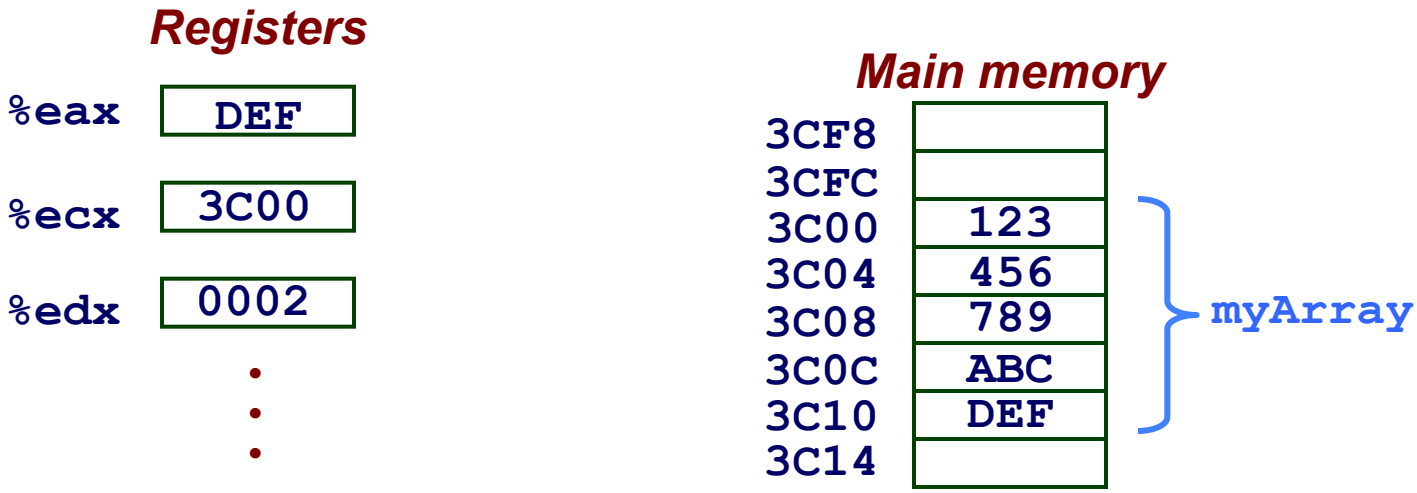
The most general format

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)

```
movl 8(%ecx,%edx,4),%eax
```



# Most General Form: Scaled indexed

**Absolute, indirect, base+displacement, and indexed are simply special cases of scaled indexed.**

## General Form

| | |
|---|---|
| $Imm(E_b,E_i,S)$ | $M[Imm + R[E_b] + R[E_i]*S]$ |

## Examples

| | |
|---|---|
| $Imm$ | $M[Imm]$ |
| $Imm(E_b)$ | $M[Imm + R[E_b]]$ |
| $(E_b,E_i,S)$ | $M[R[E_b] + R[E_i]*S]$ |
| $(E_b,E_i)$ | $M[R[E_b] + R[E_i]]$ |
| $(,E_i,S)$ | $M[R[E_i]*S]$ |
| $Imm(,E_i,S)$ | $M[Imm + R[E_i]*S]$ |

# 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`

## Generated Assembly

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

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

```
addl 8(%ebp),%eax  →  0x401046:  03 45 08
```

# 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 + displacment |
| 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-ani

```
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

| 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) | |

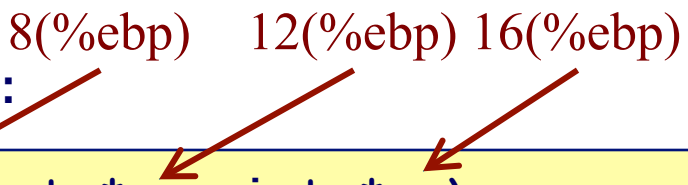
# Practice Problem

| 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** |

# Practice Problem




**A function has this prototype:**

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

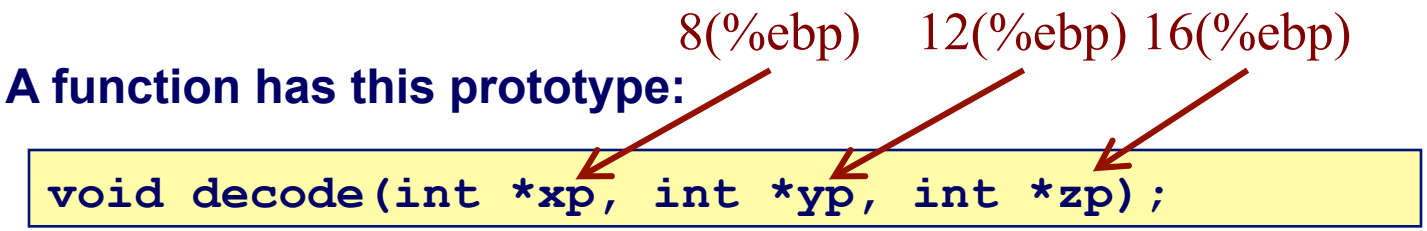
**Here is the body of the code in assembly language:**

```
1       movl 16(%ebp),%esi
2       movl 12(%ebp),%ebx
3       movl 8(%ebp),%edi
4       movl (%edi),%eax
5       movl (%ebx),%edx
6       movl (%esi),%ecx
7       movl %eax,(%ebx)
8       movl %edx,(%esi)
9       movl %ecx,(%edi)
```

***Write C code for this function***

# Practice Problem

A function has this prototype:




Here is the body of the code in assembly language:

```
1       movl 16(%ebp),%esi
2       movl 12(%ebp),%ebx
3       movl 8(%ebp),%edi
4       movl (%edi),%eax
5       movl (%ebx),%edx
6       movl (%esi),%ecx
7       movl %eax,(%ebx)
8       movl %edx,(%esi)
9       movl %ecx,(%edi)
```

*Write C code for this function*

```
void decode(int *xp, int *yp, int *zp)
{
    int t0=*xp; /* Lines 3, 4 */
    int t1=*yp; /* Lines 2, 5 */
    int t2=*zp; /* Lines 1, 6 */
    *yp=t0;     /* Line 6 */
    *zp=t1;     /* Line 8 */
    *xp=t2;     /* Line 7 */
    return t0;
}
```

# Practice Problem

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

```
int myArray[34];
```

**Write some assembly code that:**

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

**Use scaled index memory mode**

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

# 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, %eax = 0x98765432

| | |
|---|---|
| `movb %dh, %al` | %eax = 0x9876548D |
| `movsbl %dh, %eax` | %eax = 0xFFFFFF8D |
| `movzbl %dh, %eax` | %eax = 0x0000008D |

# Data Movement Instructions

| Instruction | | Effect | Description |
|---|---|---|---|
| `movl` | `S,D` | D ← S | Move double word |
| `movw` | `S,D` | D ← S | 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 |

# Arithmetic and Logical Operations

# Load address

## Load Effective Address (Long)

```
leal  S,D
```

D ← &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 = i,

```
leal 7(%edx,%edx,4),%edx
```

%edx ← 5i + 7

Multiply and add all in one instruction

# Practice Problem

%eax = x, %ecx = y

| Expression | Result in %edx |
|---|---|
| leal 6(%eax), %edx | **x+6** |
| leal (%eax, %ecx), %edx | **x+y** |
| leal (%eax, %ecx, 4), %edx | **x+4y** |
| leal 7(%eax, %eax, 8), %edx | **9x+7** |
| leal 0xA(, %ecx, 4), %edx | **4y+10** |
| leal 9(%eax, %ecx, 2), %edx | **x+2y+9** |
| leal (%ecx, %eax, 4), %edx | **4x+y** |
| leal 1(, %eax, 2), %edx | **2x+1** |

# Unary Operations

## Unary ⇒ one operand

```
inc = increment      ⇒ D ← D + 1
dec = decrement      ⇒ D ← D - 1
neg = negate         ⇒ D ← -D
not = complement     ⇒ D ← ~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` ⇒ D = D <op> S

## Can be confusing

`subl S,D` ⇒ D = D – S

Not S – D!! Be careful

## Examples

```
add S,D   ⇒ D = D + S
sub S,D   ⇒ D = D - S
xor S,D   ⇒ D = D ^ S
or  S,D   ⇒ D = D | S
and S,D   ⇒ 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 cltd 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

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

| Instruction | Effect | |
|---|---|---|
| **imull S** | R[%edx]:R[%eax] ← S x R[%eax] | signed |
| **mull S** | R[%edx]:R[%eax] ← S x R[%eax] | unsigned |
| **cltd** | R[%edx]:R[%eax] ← SignExtend(R[%eax]) | |
| **idivl S** | R[%edx] ← R[%edx]:R[%eax] mod S<br>R[%eax] ← R[%edx]:R[%eax] ÷ S | signed |
| **divl S** | R[%edx] ← R[%edx]:R[%eax] mod S<br>R[%eax] ← R[%edx]:R[%eax] ÷ S | unsigned |

# Practice Problem

| 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

| 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**

```
     &&     ||      !
```

`cmpl, testl` set the condition code registers

**Condition Codes (each is one bit):**

- CF = Carry Flag
- ZF = Zero Flag
- SF = Sign Flag
- OF = Overflow Flag

**Can be tested by other subsequent instructions**

# Shift Operations

**Arithmetic and logical shifts are possible**

**<op> amount value**

```
sal k,D      ⇒ D = D << k
shl k,D      ⇒ D = D << k
sar k,D      ⇒ D = D >> k, sign extend
shr k,D      ⇒ D = D >> k, zero fill
```

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

- `%cl` **is byte 0 of register** `%ecx`

# Practice Problem

```
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

```
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

**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
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>` `elfdump (on Sun)`

- 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 llinuxassembly.org and follow link to docs
- Start with existing assembly code and modify it

# From Intel Manual

## AND—Logical AND

| Opcode | Instruction | Description |
|---|---|---|
| 24 *ib* | AND AL,*imm8* | AL AND *imm8* |
| 25 *iw* | AND AX,*imm16* | AX AND i*mm16* |
| 25 *id* | AND EAX,*imm32* | EAX AND *imm32* |
| 80 /4 *ib* | AND *r/m8,imm8* | *r/m8* AND *imm8* |
| 81 /4 *iw* | AND *r/m16,imm16* | *r/m16* AND *imm16* |
| 81 /4 *id* | AND *r/m32,imm32* | *r/m32* AND *imm32* |
| 83 /4 *ib* | AND *r/m16,imm8* | *r/m16* AND *imm8 (sign-extended)* |
| 83 /4 *ib* | AND *r/m32,imm8* | *r/m32* AND *imm8 (sign-extended)* |
| 20 /*r* | AND *r/m8,r8* | *r/m8* AND *r8* |
| 21 /*r* | AND *r/m16,r16* | *r/m16* AND *r16* |
| 21 /*r* | AND *r/m32,r32* | *r/m32* AND *r32* |
| 22 /*r* | AND *r8,r/m8* | *r8* AND *r/m8* |
| 23 /*r* | AND *r16,r/m16* | *r16* AND *r/m16* |
| 23 /*r* | AND *r32,r/m32* | *r32* AND *r/m32* |

SOURCE: http://download.intel.com/design/intarch/manuals/24319101.pdf

# From Intel Manual

### Description

This instruction performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. Two memory operands cannot, however, be used in one instruction. Each bit of the instruction result is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

### Operation

DEST ← DEST AND SRC;

### Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

SOURCE: http://download.intel.com/design/intarch/manuals/24319101.pdf

# From Intel Manual

**Protected Mode Exceptions**

| | |
|---|---|
| #GP(0) | If the destination operand points to a nonwritable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a null segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

**Real-Address Mode Exceptions**

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

**Virtual-8086 Mode Exceptions**

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

SOURCE: http://download.intel.com/design/intarch/manuals/24319101.pdf

# Bonus material

# 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 | | |

# 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

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

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

Set Up

Body

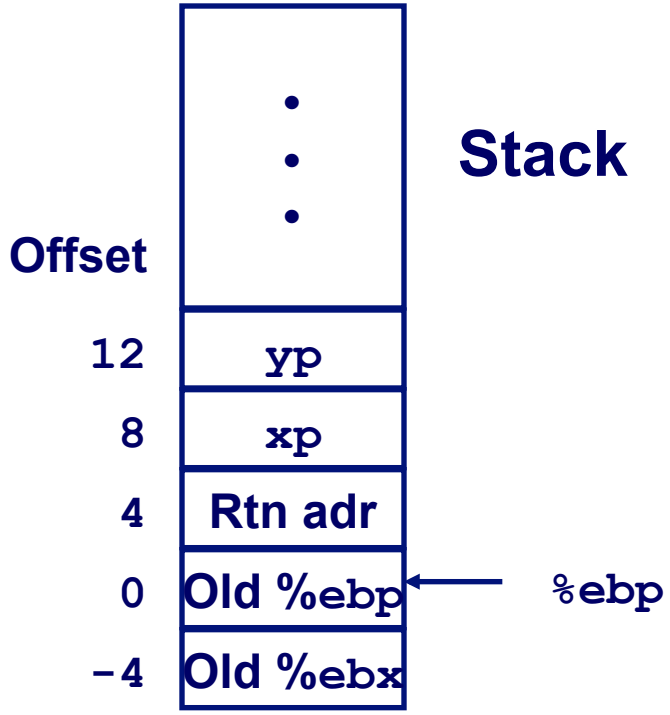
Finish

**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
```