CS 31: Introduction to Computer Systems

07-08: ISAs and Assembly

February 11, 13

Today

- How to directly interact with hardware
- Instruction set architecture (ISA)
  - Interface between programmer and CPU
  - Established instruction format (assembly lang)
- Assembly programming (IA-32 or x86)

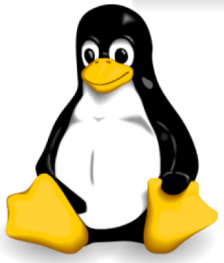
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources



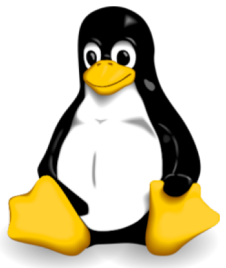
Complex devices
Compute & I/O

Abstraction

Applications
Specific functionality

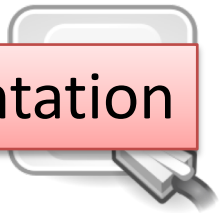
This week: Machine Interface

Operating system
Manage resources

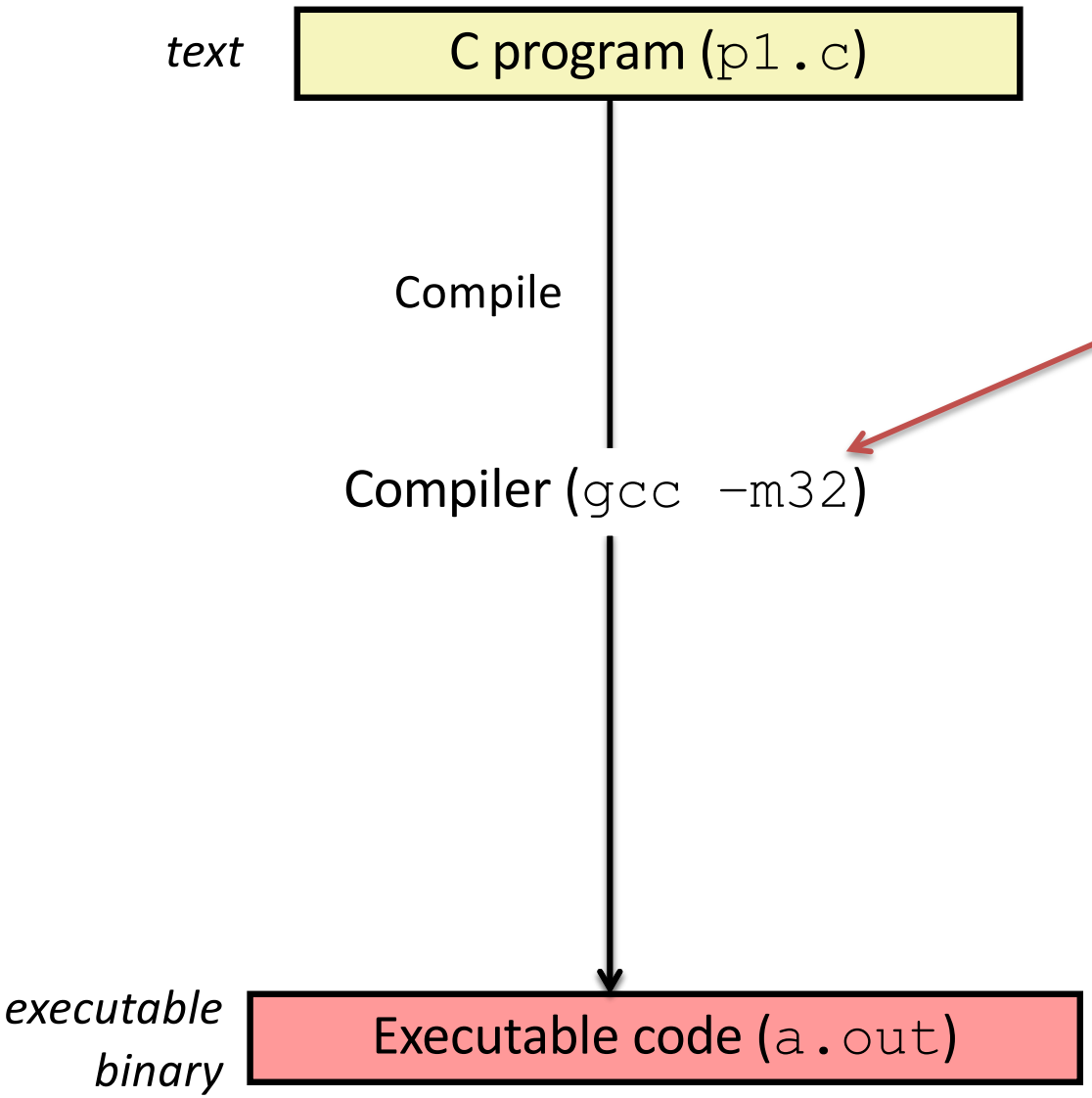


Complex d
Compute & I/O

Last week: Circuits, Hardware Implementation



Compilation Steps (.c to a.out)

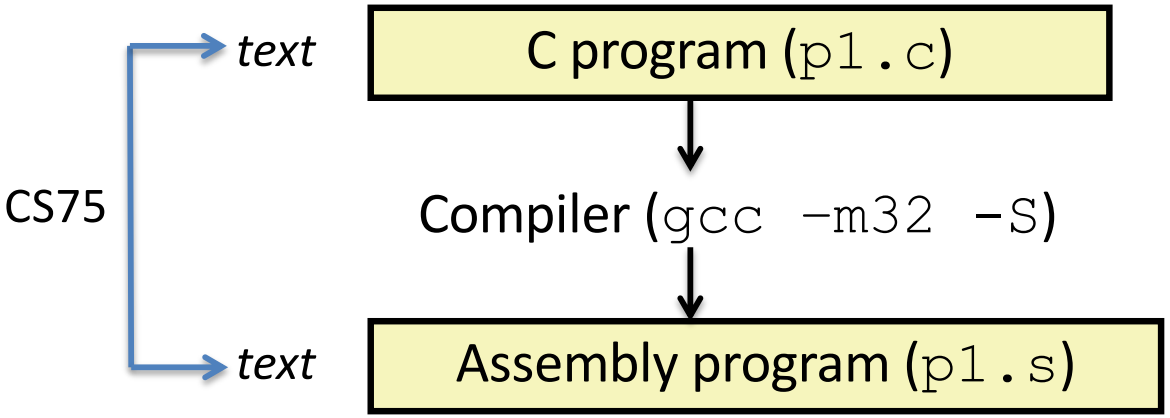


Usually compile to a.out in a single step: gcc –m32 p1.c

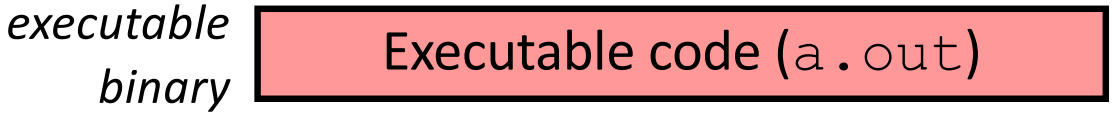
-m32 tells gcc to compile for 32-bit Intel machines

Reality is more complex: there are intermediate steps!

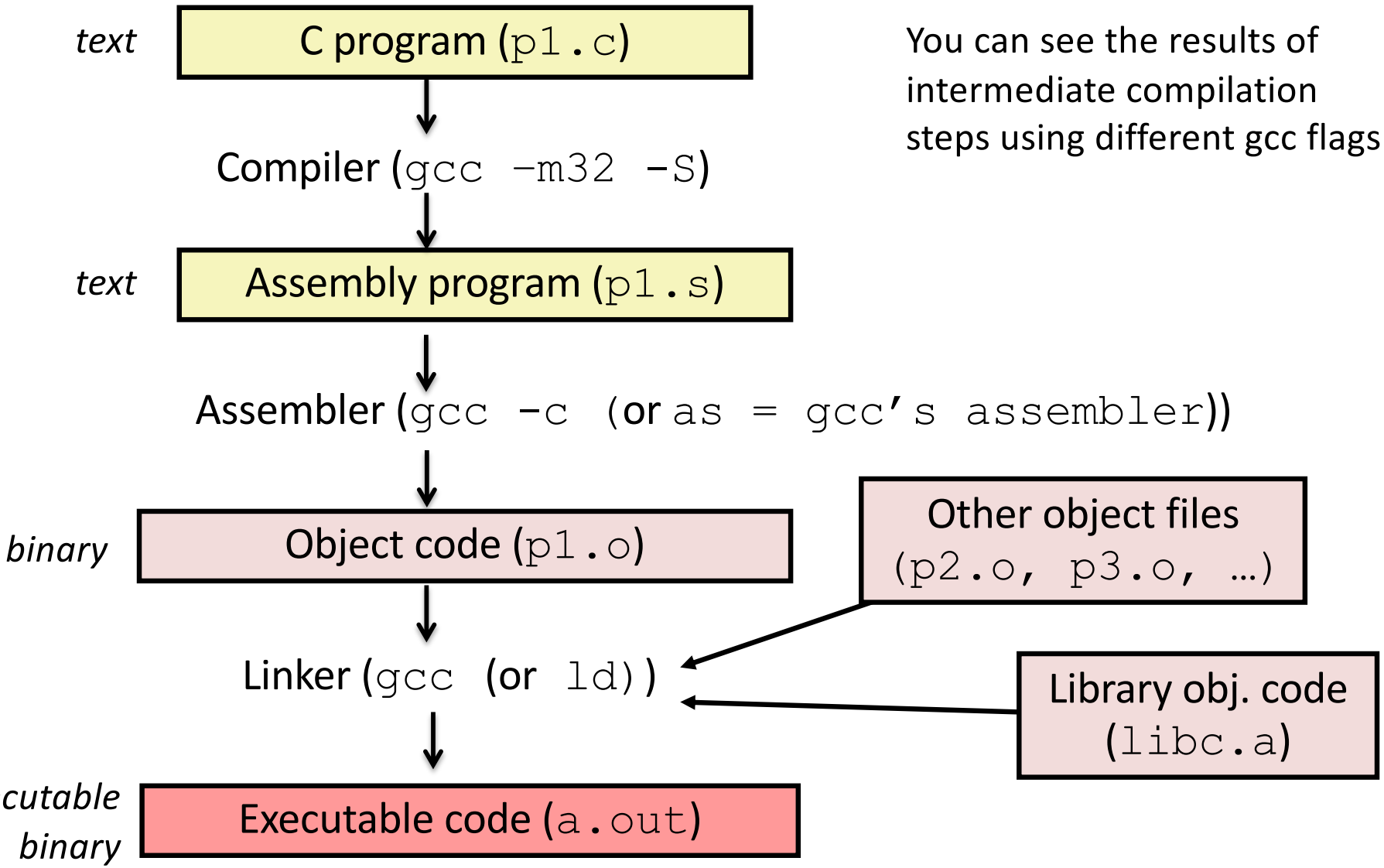
Compilation Steps (.c to a.out)



You can see the results of intermediate compilation steps using different gcc flags



Compilation Steps (.c to a.out)



You can see the results of intermediate compilation steps using different gcc flags

What is “assembly”?

```
push %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

Assembly is the “human readable” form of the instructions a machine can understand.

Object / Executable / Machine Code

Assembly	**Machine Code (Hexadecimal)**
`push %ebp`	`55`
`mov %esp, %ebp`	`89 E5`
`sub $16, %esp`	`83 EC 10`
`movl $10, -8(%ebp)`	`C7 45 F8 0A 00 00 00`
`movl $20, -4(%ebp)`	`C7 45 FC 14 00 00 00`
`movl -4(%ebp), $eax`	`8B 45 FC`
`addl $eax, -8(%ebp)`	`01 45 F8`
`movl -8(%ebp), %eax`	`B8 45 F8`
`leave`	`C9`

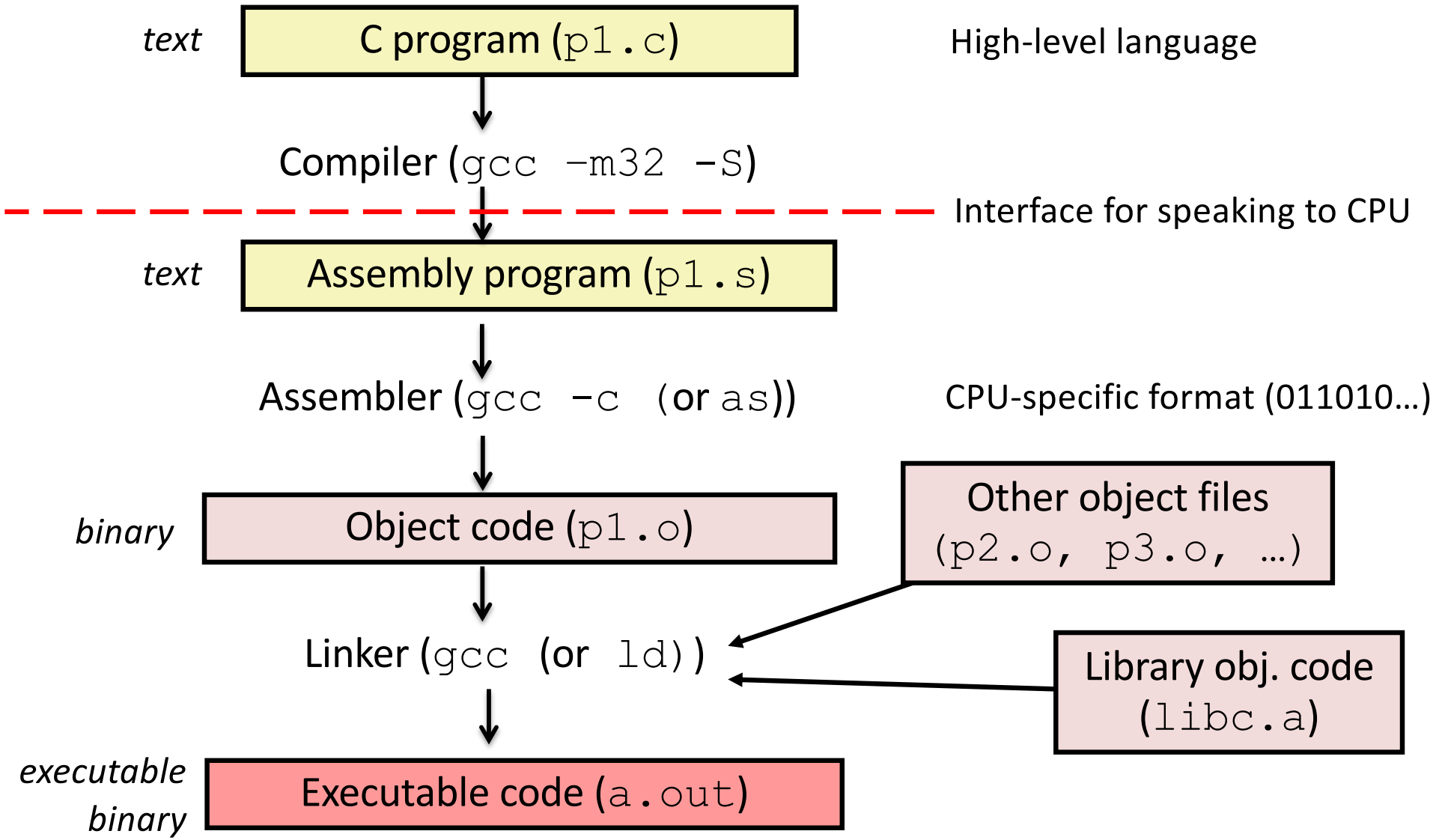
Object / Executable / Machine Code

Assembly

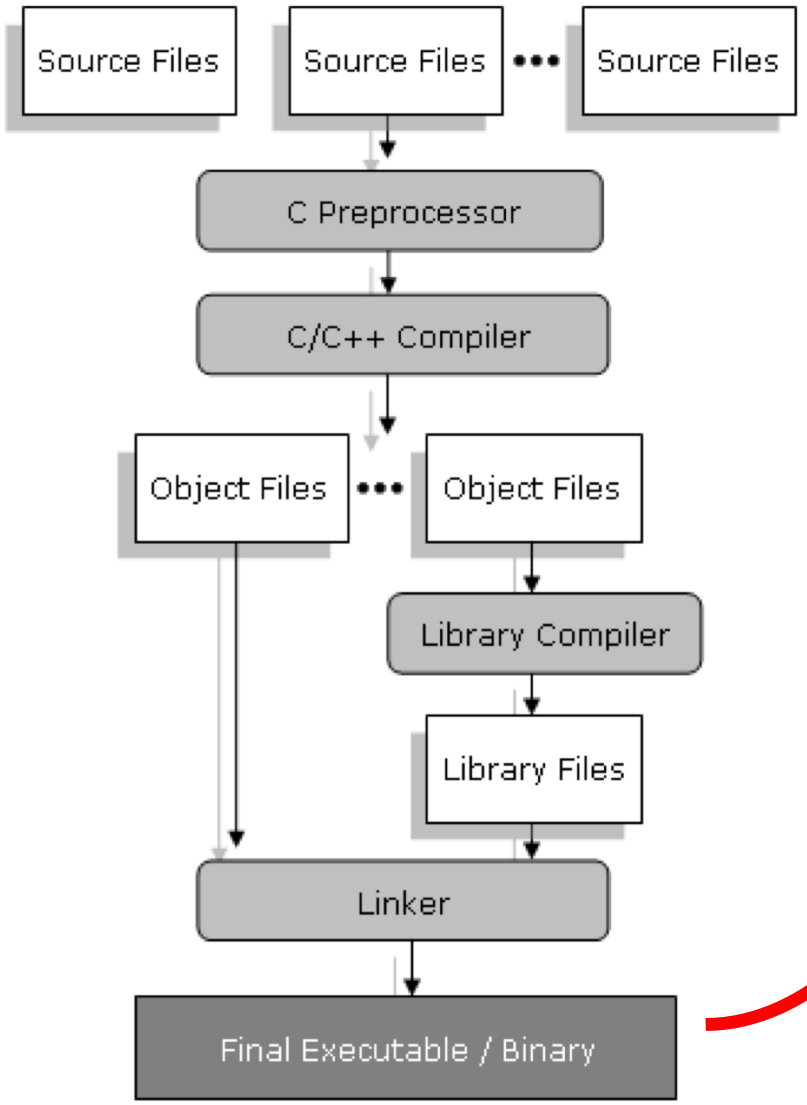
```
push %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

```
int main() {
    int a = 10;
    int b = 20;

    a = a + b;

    return a;
}
```

Compilation Steps (.c to a.out)



Compilation Process



What is going on here?

Core i7 Instructions
(x86 Family)

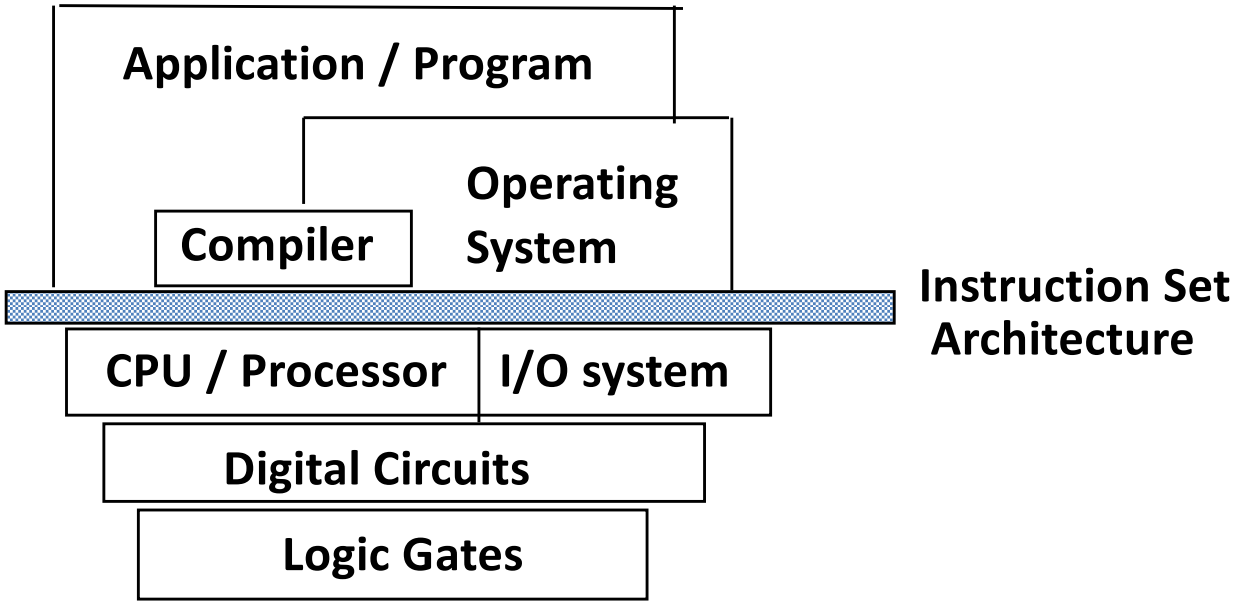
Instruction Set Architecture (ISA)

- ISA (or simply architecture):
  Interface between lowest software level and the hardware.

- Defines specification of the language for controlling CPU state:
  - Provides a set of instructions
  - Makes CPU registers available
  - Allows access to main memory
  - Exports control flow (change what executes next)

Instruction Set Architecture (ISA)

- The agreed-upon interface between all software that runs on the machine and the hardware that executes it.



Instruction Set Architectures: Examples

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360

How many of these ISAs have you used?

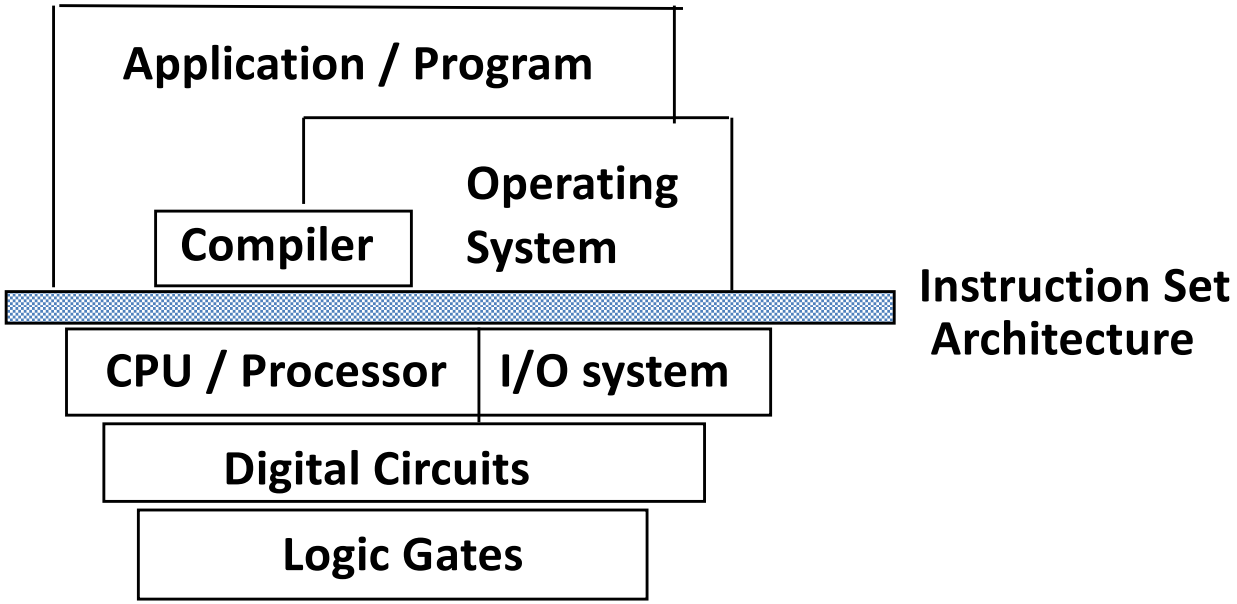
(Don't worry if you're not sure. Try to guess based on the types of CPUs/devices you interact with.)

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- AMD-64 (x86-64)
- VAX
- SPARC
- Alpha
- IBM 360

A. 0
B. 1-2
C. 3-4
D. 5-6
E. 7+

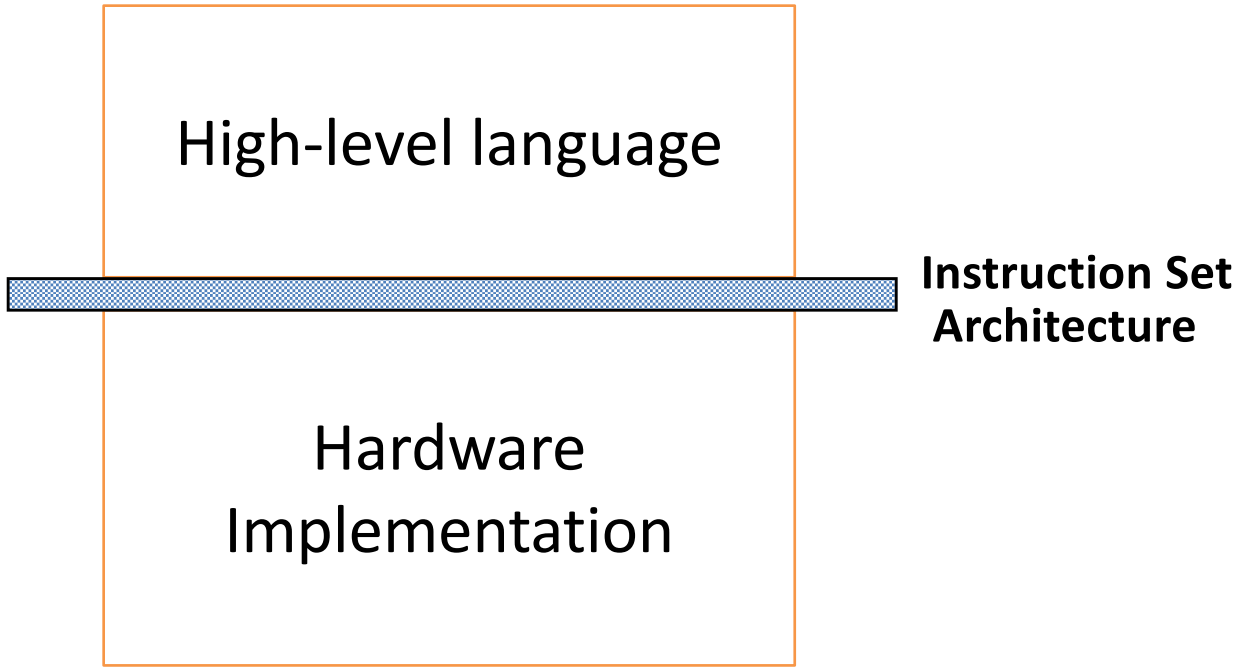
Instruction Set Architecture (ISA)

- The agreed-upon interface between all software that runs on the machine and the hardware that executes it.

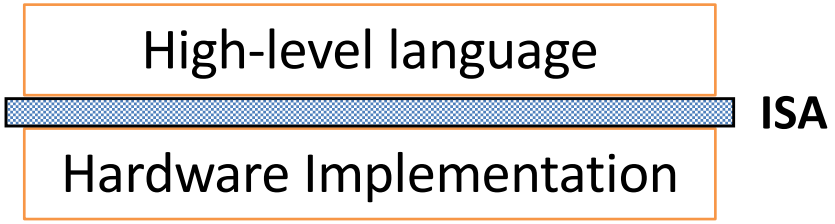


Instruction Set Architecture (ISA)

- The agreed-upon interface between all software that runs on the machine and the hardware that executes it.

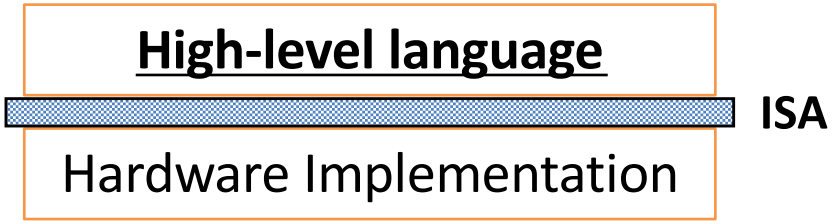


ISA Characteristics



- Above ISA: High-level language (C, Python, …)
  - Hides ISA from users
  - Allows a program to run on any machine (after translation by human and/or compiler)
- Below ISA: Hardware implementing ISA can change (faster, smaller, …)
  - ISA is like a CPU “family”

ISA Characteristics



- Above ISA: High-level language (C, Python, …)
  - Hides ISA from users
  - Allows a program to run on any machine (after translation by human and/or compiler)

- Below ISA: Hardware implementing ISA can change (faster, smaller, …)
  - ISA is like a CPU “family”

Instruction Translation

sum.c (High-level C)

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

sum.s (Assembly)

```
sum:
   pushl %ebp
   movl  %esp,%ebp
   subl  $24, %esp
   movl  12(%ebp),%eax
   addl  8(%ebp),%eax
   movl  %eax, -12(%ebp)
   leave
   ret
```

sum.s from sum.c:

```
gcc -m32 -S sum.c
```

Instructions to set up the stack frame and get argument values

An add instruction to compute sum

Instructions to return from function

ISA Design Questions

sum.c (High-level C)

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

sum.s (Assembly)

```
sum:
   pushl %ebp
   movl  %esp,%ebp
   subl  $24, %esp
   movl  12(%ebp),%eax
   addl  8(%ebp),%eax
   movl  %eax, -12(%ebp)
   leave
   ret
```

sum.s from sum.c:

```
gcc -m32 -S sum.c
```

What should these instructions do?

What is/isn't allowed by hardware?

How complex should they be?

Example: supporting multiplication.

C statement: A = A*B

Simple instructions:

```
LOAD A, eax
LOAD B, ebx
PROD eax, ebx
STORE ebx, A
```

Powerful instructions:

```
MULT B, A
```

Translation:

Load the values ‘A’ and ‘B’ from memory into registers, compute the product, store the result in memory where ‘A’ was.

Which would you use if you were designing an ISA for your CPU? (Why?)

Simple instructions:

```
LOAD A, eax
LOAD B, ebx
PROD eax, ebx
STORE ebx, A
```

Powerful instructions:

```
MULT B, A
```

A. Simple
B. Powerful
C. Something else

RISC versus CISC (Historically)

- Complex Instruction Set Computing (CISC)
  - Large, rich instruction set
  - More complicated instructions built into hardware
  - Multiple clock cycles per instruction
  - Easier for humans to reason about

- Reduced Instruction Set Computing (RISC)
  - Small, highly optimized set of instructions
  - Memory accesses are specific instructions
  - One instruction per clock cycle
  - Compiler: more work, more potential optimization

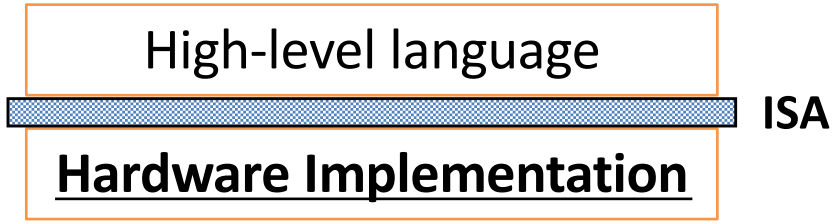
So . . . Which System "Won"?

- Most ISAs (after mid/late 1980's) are RISC
- The ubiquitous Intel x86 is CISC
  - Tablets and smartphones (ARM) taking over?
- x86 breaks down CISC assembly into multiple, RISC-like, machine language instructions
- Distinction between RISC and CISC is less clear
  - Some RISC instruction sets have more instructions than some CISC sets

ISA Examples

- Intel IA-32 (CISC)
- ARM (RISC)
- MIPS (RISC)
- PowerPC (RISC)
- IBM Cell (RISC)
- Motorola 68k (CISC)
- Intel IA-64 (Neither)
- VAX (CISC)
- SPARC (RISC)
- Alpha (RISC)
- IBM 360 (CISC)

ISA Characteristics



- Above ISA: High-level language (C, Python, …)
  - Hides ISA from users
  - Allows a program to run on any machine (after translation by human and/or compiler)

- Below ISA: Hardware implementing ISA can change (faster, smaller, …)
  - ISA is like a CPU “family”

Intel x86 Family (IA-32)

Intel i386 (1985)

- 12 MHz - 40 MHz
- ~300,000 transistors
- Component size: 1.5 µm

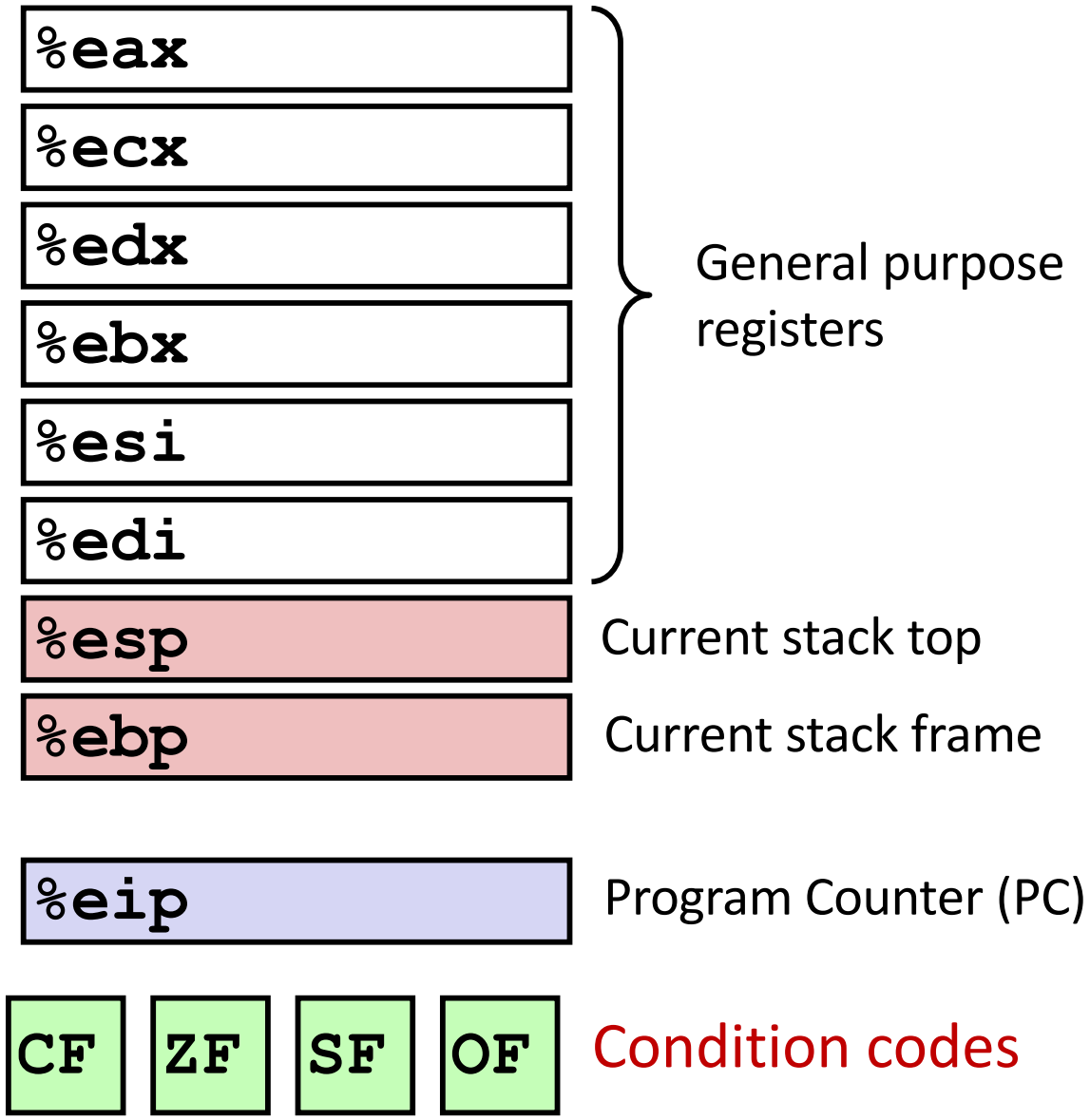
Intel Core i9 9900k (2018)

- ~4,000 MHz
- ~7,000,000,000 transistors
- Component size: 14 nm

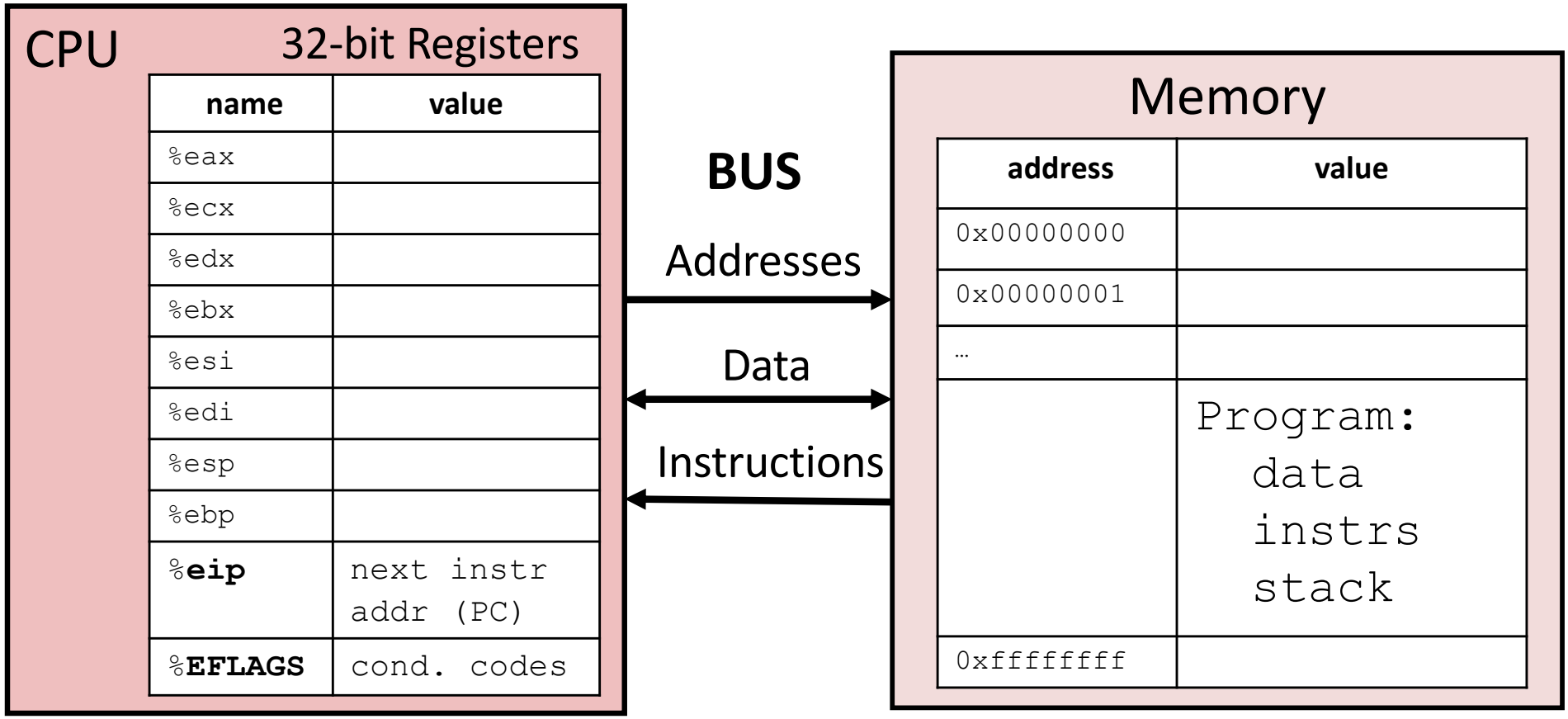
Everything in this family uses the same ISA (Same instructions)!

Processor State in Registers

- Information about currently executing program
  - Temporary data (%eax - %edi)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip, …)
  - Status of recent tests %EFLAGS (CF, ZF, SF, OF)



Assembly Programmer's View of State



Registers:

PC: Program counter (%eip)

Condition codes (%EFLAGS)

General Purpose (%eax - %ebp)

Memory:

- Byte addressable array
- Program code and data
- Execution stack

General purpose Registers

Six are for instruction operands

Can store 4 byte data or address value

The low-order 2 bytes %ax is the low-order 16 bits of %eax

Two low-order 1 bytes %al is the low-order 8 bits of %eax

May see their use in ops involving shorts or chars

Register name
%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp
%eip
%EFLAGS

bits: 31 16		15 8	7 0
%eax	%ax	%ah	%al
%ecx	%cx	%ch	%cl
%edx	%dx	%dh	%dl
%ebx	%bx	%bh	%bl
%esi	%si		
%edi	%di		
%esp	%sp		
%ebp	%bp		

General purpose Registers

Register name	Register value
%eax	3
%ecx	5
%edx	8
%ebx	
%esi	
%edi	
%esp	
%ebp	
%eip	
%EFLAGS	

- Six are for instruction operands
- Can store 4 byte data or address value

Takeaway: the instructions in IA32 assembly will refer to these register names when selecting ALU operands and locations to store results.

Types of IA32 Instructions

- Data movement
  - Move values between registers and memory
  - Example: `movl`

- Load: move data from memory to register

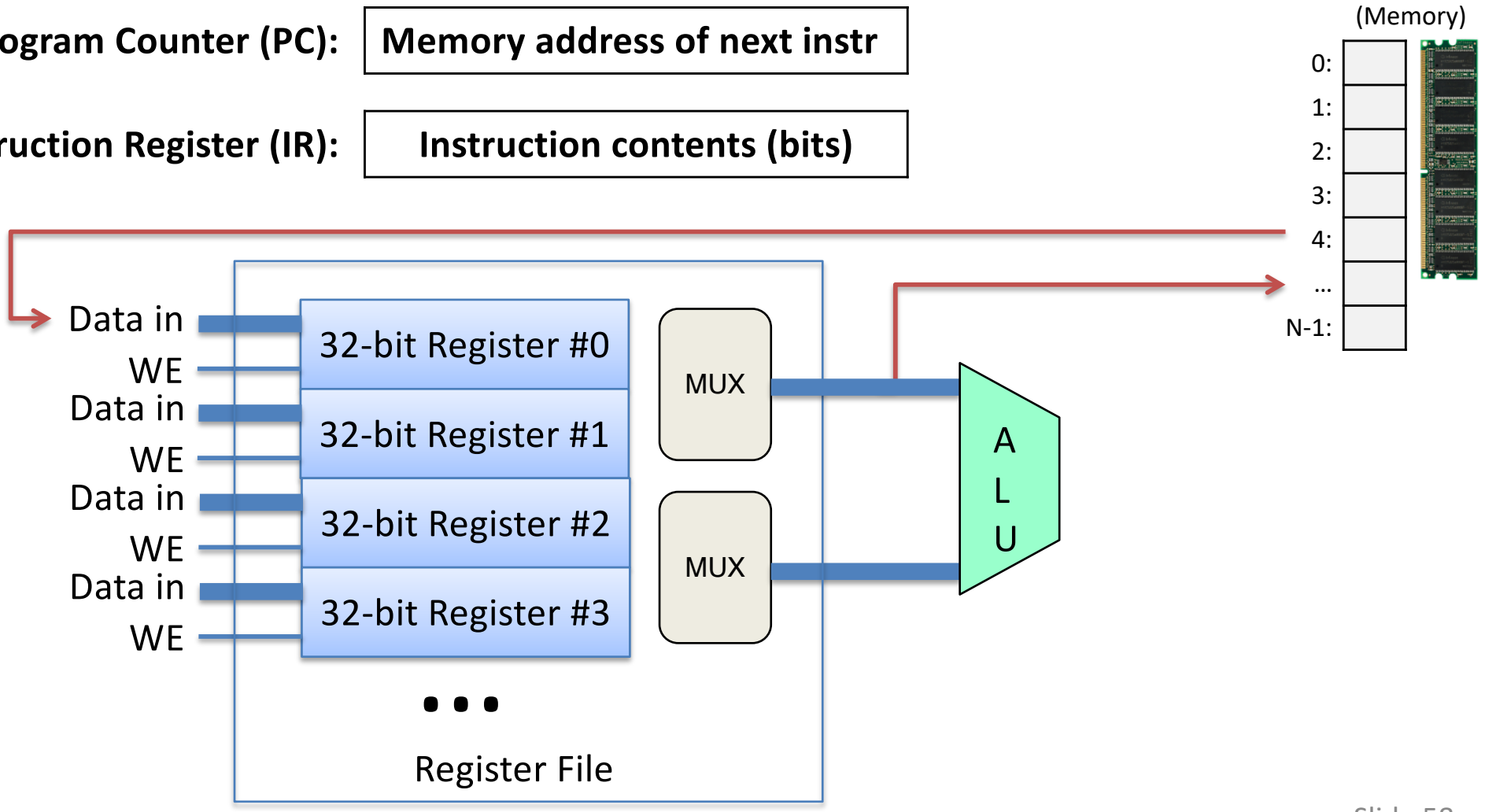
- Store: move data from register to memory

Data Movement

Move values between memory and registers or between two registers.

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**



Types of IA32 Instructions

- Data movement
  - Move values between registers and memory

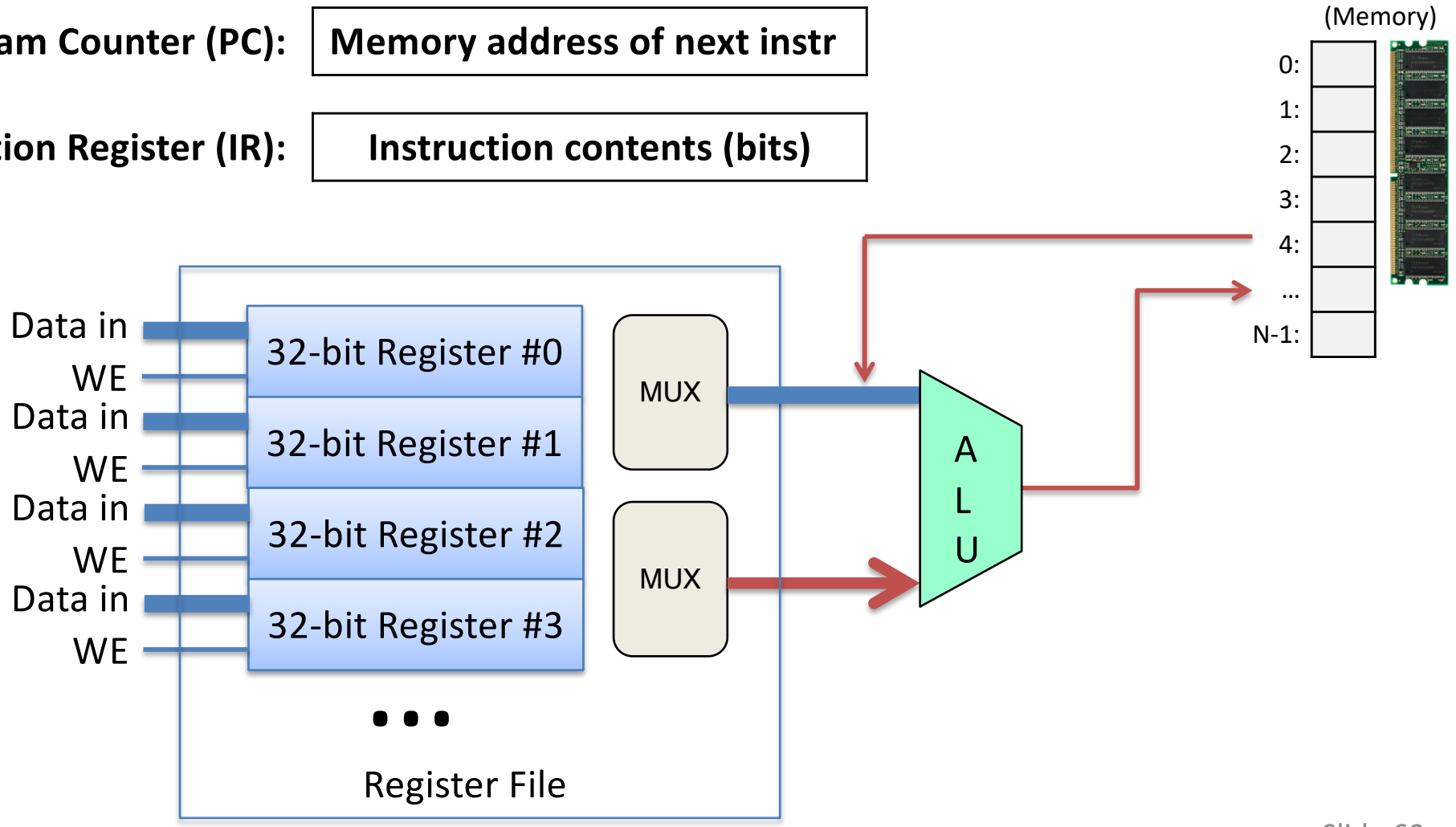
- Arithmetic
  - Uses ALU to compute a value
  - Example: `addl`

Arithmetic

Use ALU to compute a value, store result in register / memory.

Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**

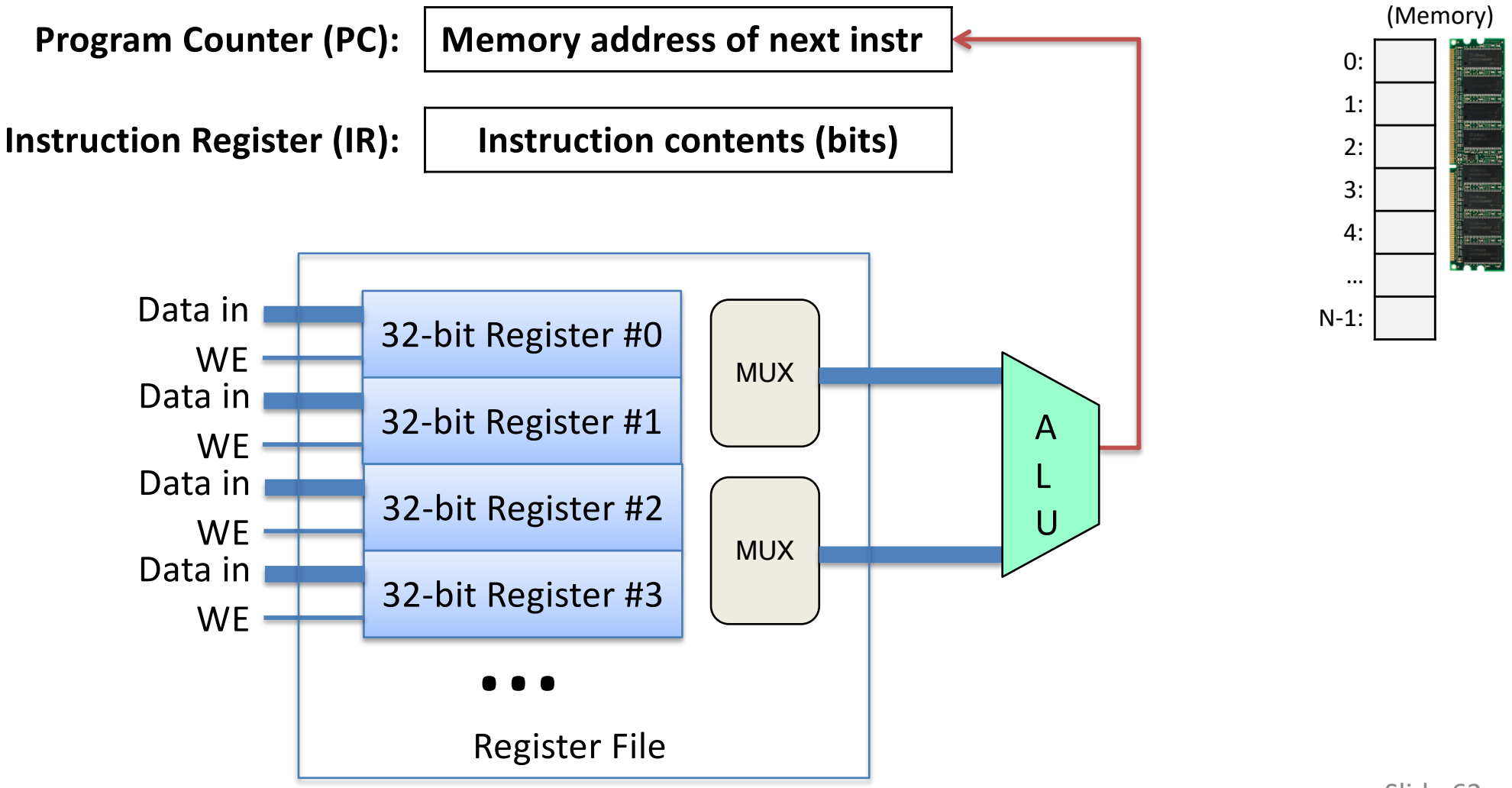


Types of IA32 Instructions

- Data movement
  - Move values between registers and memory
- Arithmetic
  - Uses ALU to compute a value
- Control
  - Change PC based on ALU condition code state
  - Example: `jmp`

Control

Change PC based on ALU condition code state.



Types of IA32 Instructions

- Data movement
  - Move values between registers and memory
- Arithmetic
  - Uses ALU to compute a value
- Control
  - Change PC based on ALU condition code state
- Stack / Function call (We'll cover these in detail later)
  - Shortcut instructions for common operations

Addressing Modes

- Data movement and arithmetic instructions:
  - Must tell CPU where to find operands, store result

- You can refer to a register by using %:
  - %eax

- `addl %ecx, %eax`
  - Add the contents of registers ecx and eax, store result in register eax.

Addressing Mode: Immediate

- Refers to a constant value, starts with $

- `movl $10, %eax`
  - Put the constant value 10 in register eax.

Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put address in a register.
  - Access with () around register name.

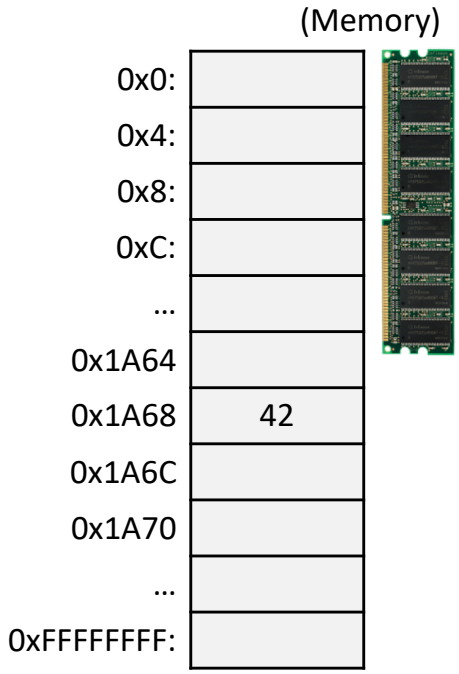
- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax

Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax

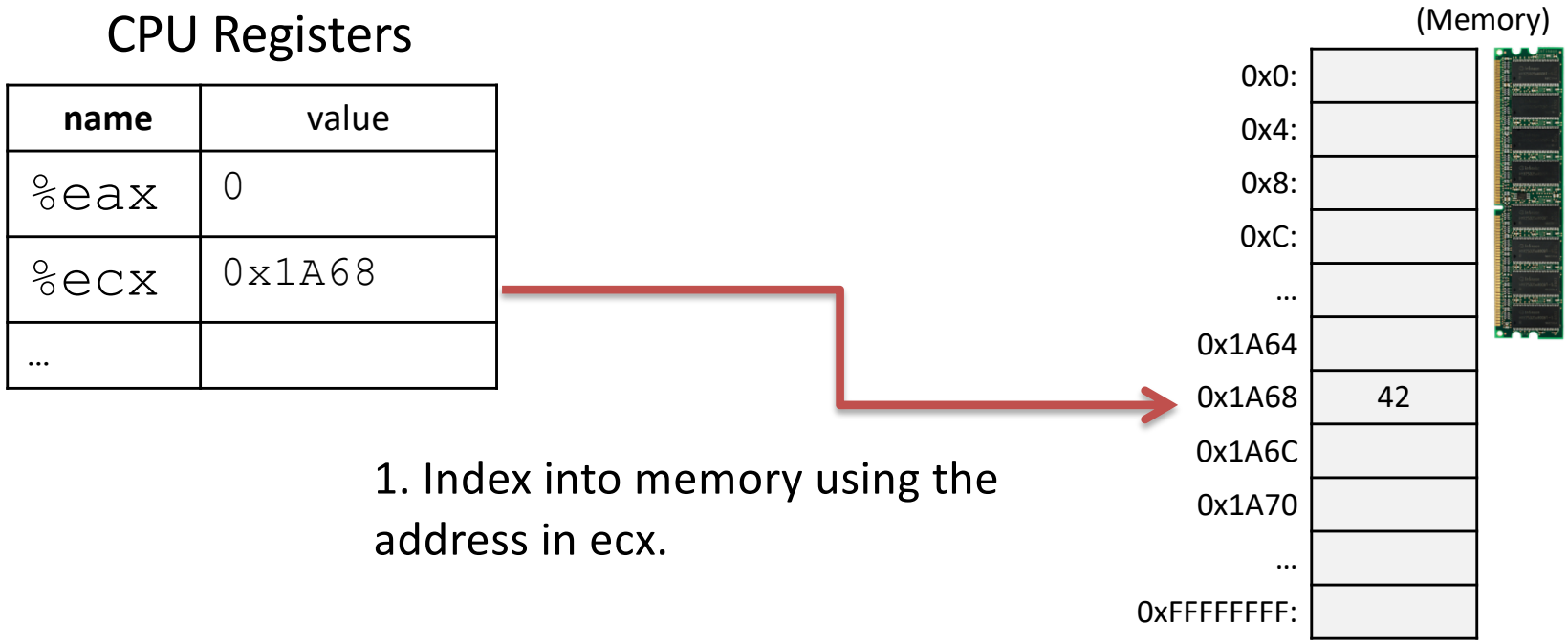
CPU Registers

name	value
%eax	0
%ecx	0x1A68
…	



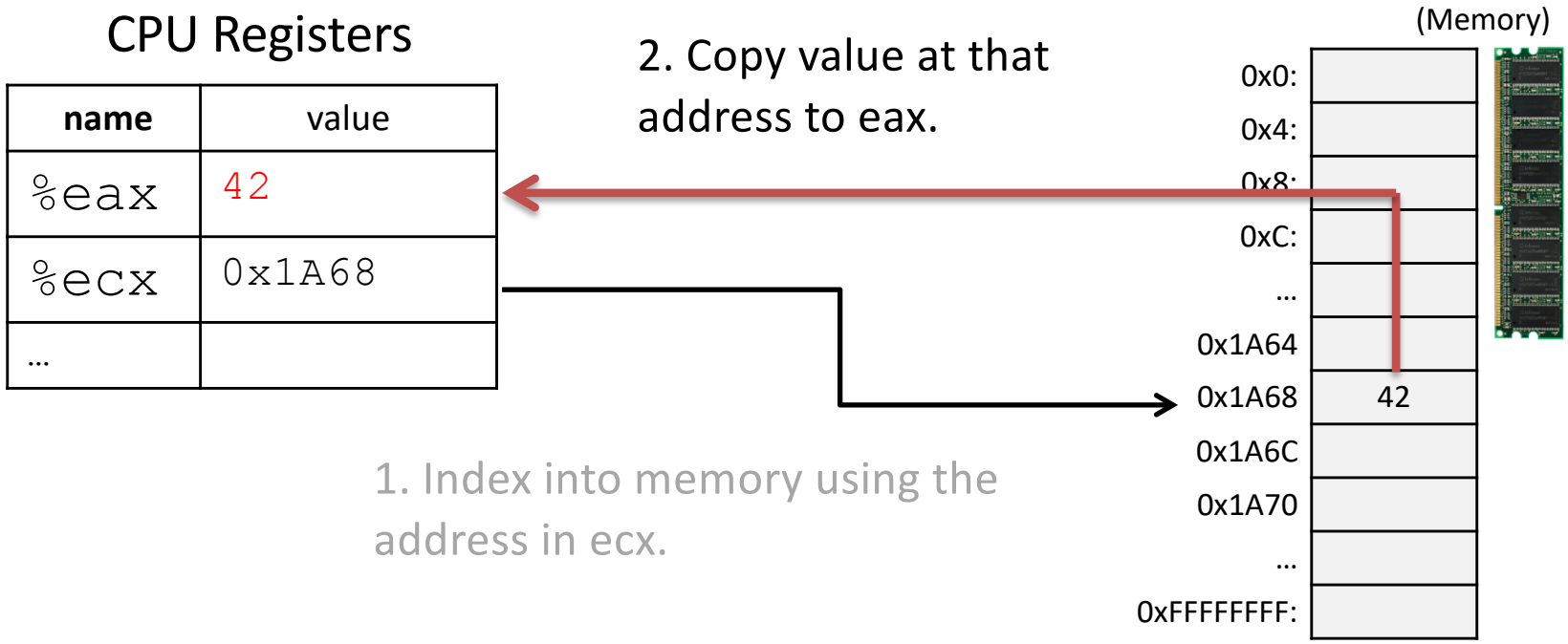
Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax



Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax



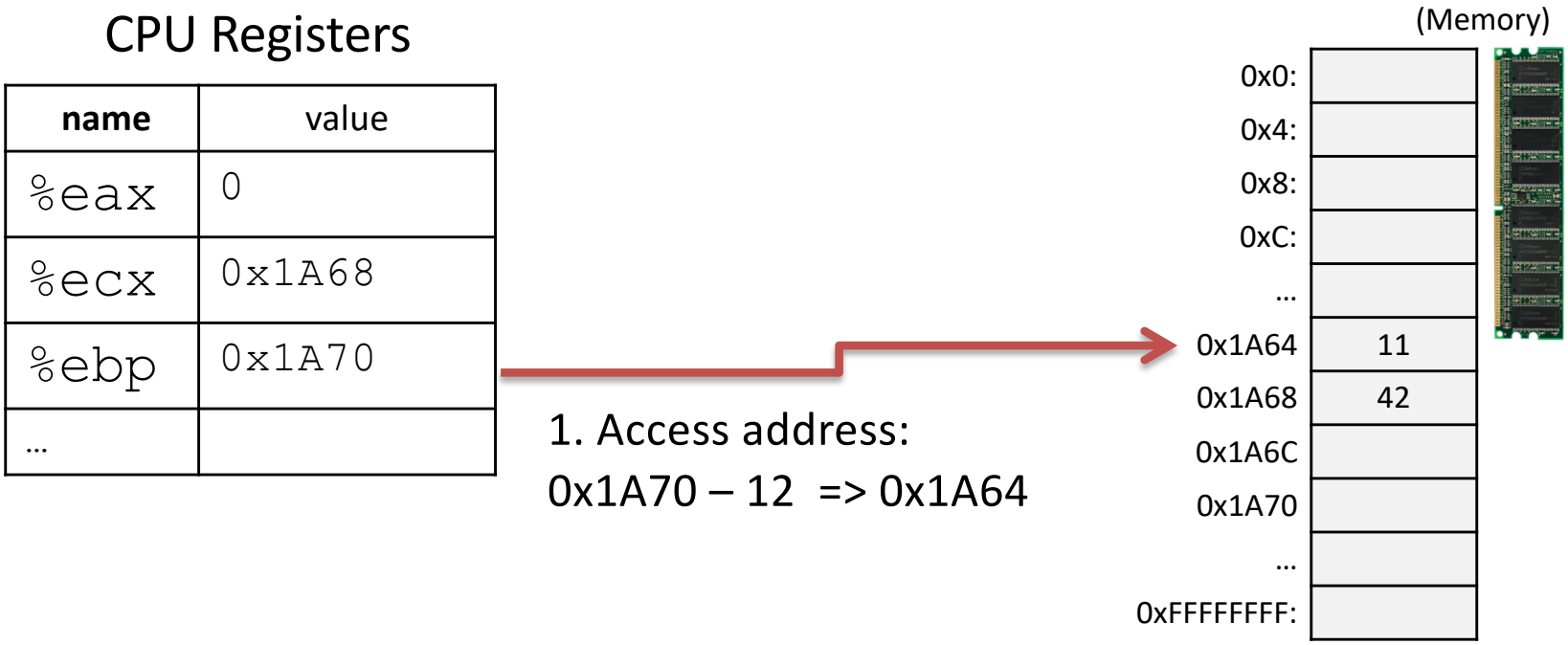
Addressing Mode: Displacement

- Like memory mode, but with constant offset
  - Offset is often negative, relative to %ebp

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract twelve from it, index into memory and store the result in eax

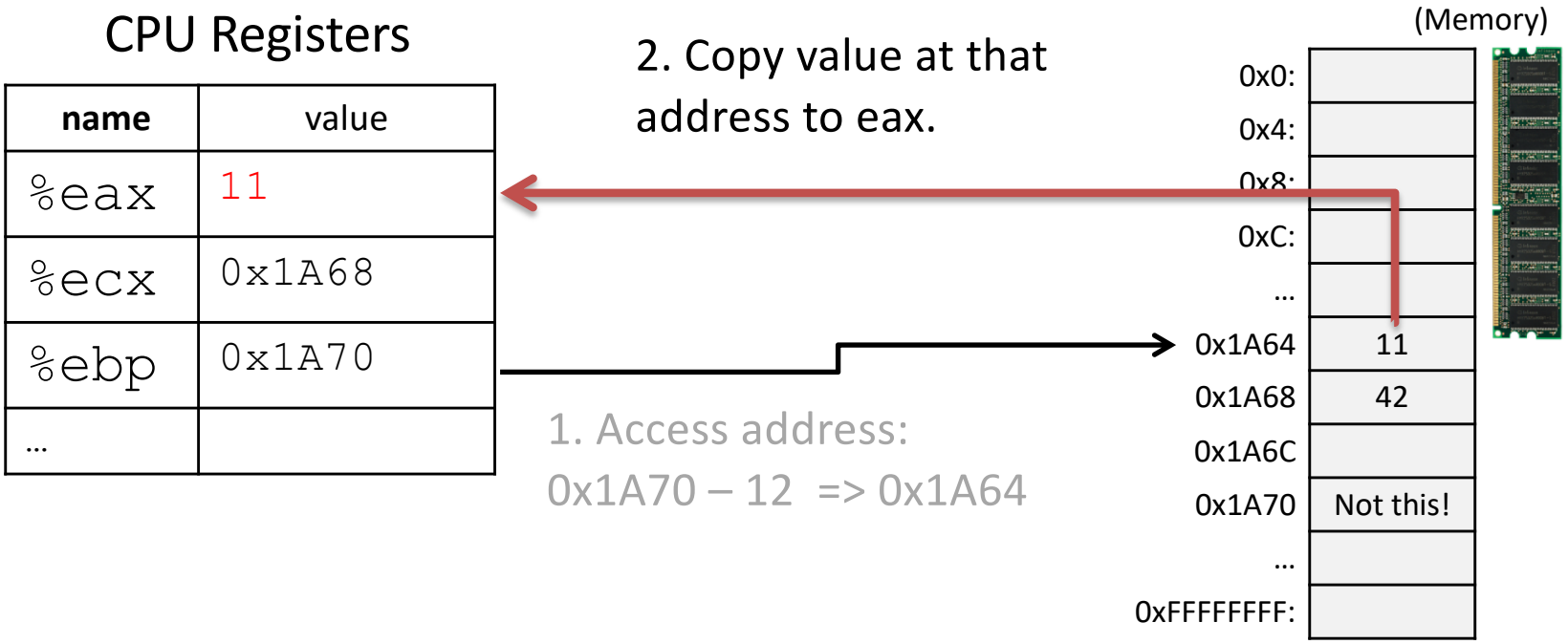
Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract twelve from it, index into memory and store the result in eax



Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract three from it, index into memory and store the result in eax



Let’s try a few examples...

What will memory look like after these instructions?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl -16(%ebp),%eax
sall  $3, %eax
imull $3, %eax
movl -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

Registers

name	value
%eax	?
%edx	?
%ebp	**0x1270**

Memory

address	value
0x1260	2
0x1264	3
0x1268	2
0x126c	
0x1270	
…	

What will memory look like after these instructions?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl -16(%ebp),%eax
sall  $3, %eax
imull $3, %eax
movl -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

A:

address	value
0x1260	53
0x1264	3
0x1268	24
0x126c	
0x1270	
…	

B:

address	value
0x1260	53
0x1264	3
0x1268	2
0x126c	
0x1270	
…	

C:

address	value
0x1260	2
0x1264	16
0x1268	24
0x126c	
0x1270	
…	

D:

address	value
0x1260	2
0x1264	3
0x1268	53
0x126c	
0x1270	
…	

What will memory look like after these instructions?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl -16(%ebp),%eax
sall  $3, %eax
imull $3, %eax
movl -12(%ebp), %edx
addl  -8(%ebp), %edx
addl  %edx, %eax
movl  %eax, -8(%ebp)
```

D:

address	value
0x1260	2
0x1264	3
0x1268	53
0x126c	
0x1270	
…	

A:

address	value
0x1260	53
0x1264	3
0x1268	24
0x126c	
0x1270	
…	

B:

address	value
0x1260	53
0x1264	3
0x1268	2
0x126c	
0x1270	
…	

C:

address	value
0x1260	2
0x1264	16
0x1268	24
0x126c	
0x1270	
…	

Solution

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl   -16(%ebp), %eax
sall    $3, %eax
imull   $3, %eax
movl   -12(%ebp), %edx
addl   -8(%ebp), %edx
addl   %edx, %eax
movl   %eax, -8(%ebp)
```

Equivalent C code:

```
x = z*24 + y + x;
```

name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260	2
0x1264	3
0x1268	2
0x126c	
0x1270	

Solution

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
movl   -16(%ebp), %eax  # R[%eax] ← z        (2)
sall    $3, %eax        # R[%eax] ← z<<3     (16)
imull   $3, %eax        # R[%eax] ← 16*3     (48)
movl   -12(%ebp), %edx  # R[%edx] ← y        (3)
addl   -8(%ebp), %edx   # R[%edx] ← y + x    (5)
addl   %edx, %eax       # R[%eax] ← 48+5     (53)
movl   %eax, -8(%ebp)   # M[R[%ebp]+8]←5     (x=53)
```

Z*24

Equivalent C code:

```
x = z*24 + y + x;
```

name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260	2	z
0x1264	3	y
0x1268	2	x
0x126c		
0x1270		

What will the machine state be after executing these instructions?

```
movl %ebp, %ecx
subl $16, %ecx
movl (%ecx), %eax
orl  %eax, -8(%ebp)
negl %eax
movl %eax, 4(%ecx)
```

name	value
%eax	?
%ecx	?
%ebp	**0x456C**

address	value
0x455C	7
0x4560	11
0x4564	5
0x4568	3
0x456C	
…	

How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

name	value
%eax	
%edx	
%ebp	**0x1270**

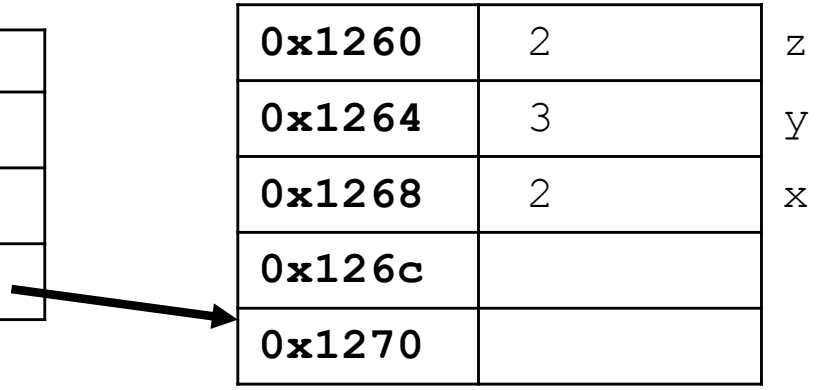
0x1260	2	z
0x1264	3	y
0x1268	2	x
0x126c		
0x1270		

C code: `z = x ^ y`

How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

name	value
%eax	
%edx	
%ebp	**0x1270**



C code: `z = x ^ y`

A:
```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %eax, %edx
movl %eax, -16(%ebp)
```

B:
```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %edx, %eax
movl %eax, -16(%ebp)
```

C:
```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %eax, %edx
movl %eax, -8(%ebp)
```

D:
```
movl -16(%ebp), %eax
movl -12(%ebp), %edx
xorl %edx, %eax
movl %eax, -8(%ebp)
```

How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260	2	z
0x1264	3	y
0x1268	2	x
0x126c		
0x1270		

C code: `z = x ^ y`

A:
```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %eax, %edx
movl %eax, -16(%ebp)
```

B:
```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %edx, %eax
movl %eax, -16(%ebp)
```

C:
```
movl -8(%ebp), %eax
movl -12(%ebp), %edx
xorl %eax, %edx
movl %eax, -8(%ebp)
```

D:
```
movl -16(%ebp), %eax
movl -12(%ebp), %edx
xorl %edx, %eax
movl %eax, -8(%ebp)
```

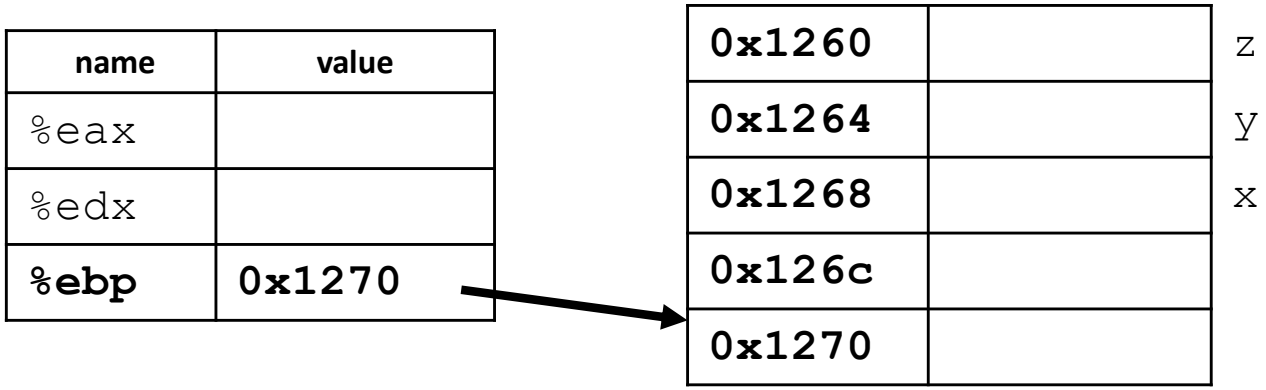
How would you do this in IA32?

x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

name	value
%eax	
%edx	
%ebp	**0x1270**

0x1260	2	z
0x1264	3	y
0x1268	2	x
0x126c		
0x1270		

```
x = y >> 3 | x * 8
```




```
(1)z = x ^ y
   movl -8(%ebp), %eax    # R[%eax] ← x
   movl -12(%ebp), %edx   # R[%edx] ← y
   xorl %edx, %eax        # R[%eax] ← x ^ y
   movl %eax, -16(%ebp)   # M[R[%ebp-16]] ← x^y

(2)x = y >> 3 | x * 8
      movl -8(%ebp), %eax    # R[%eax] ← x
      imull $8, %eax         # R[%eax] ← x*8
      movl -12(%ebp), %edx   # R[%edx] ← y
   rshl $3, %edx          # R[%edx] ← y >> 3
   orl  %eax, %edx        # R[%edx] ← y>>3 | x*8
   movl %edx, -8(%ebp)    # M[R[%ebp-8]] ← result
```

Recall Memory Operands

- `displacement(%reg)`
  - e.g., `addl %eax, -8(%ebp)`

- IA32 allows a memory operand as the source or destination, but NOT BOTH
  - One of the operands must be a register

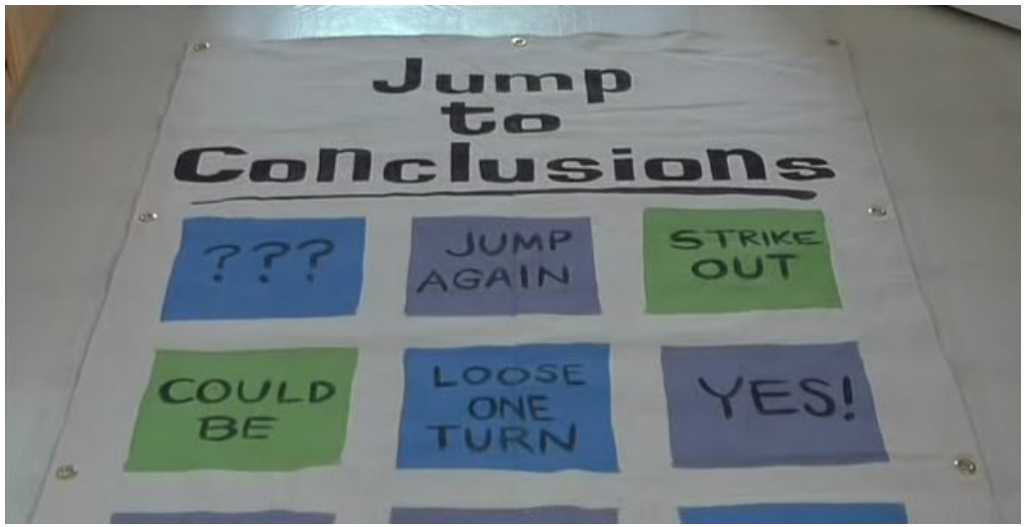
- This would not be allowed:
  - `addl -4(%ebp), -8(%ebp)`
  - If you wanted this, `movl` one value into a register first

Control Flow

- Previous examples focused on:
  - data movement (movl)
  - arithmetic (addl, subl, orl, negl, sall, etc.)

- Up next: Jumping!

  (Changing which instruction we execute next.)



Relevant XKCD

[xkcd #292]

Unconditional Jumping / Goto

```
int main() {
  int a = 10;
  int b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

A label is a place you might jump to.

Labels ignored except for goto/jumps.

(Skipped over if encountered)

```
 int x = 20;
L1:
  int y = x + 30;
L2:
  printf("%d, %d\n", x, y);
```

Unconditional Jumping / Goto

```
int main() {
  int a = 10;
  int b = 20;

  goto label1;
  a = a + b;

label1:
  return;
```

```
push  %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
jmp label1
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
label1:
leave
```

Unconditional Jumping

Usage besides GOTO?

```
push  %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
jmp label1
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
label1:
leave
```

Unconditional Jumping

- Usage besides GOTO?
  - infinite loop
  - break;
  - continue;
  - functions (handled differently)
- Often, we only want to jump when *something* is true / false.
- Need some way to compare values, jump based on comparison results.

```
push%ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
jmp label1
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
label1:
leave
```

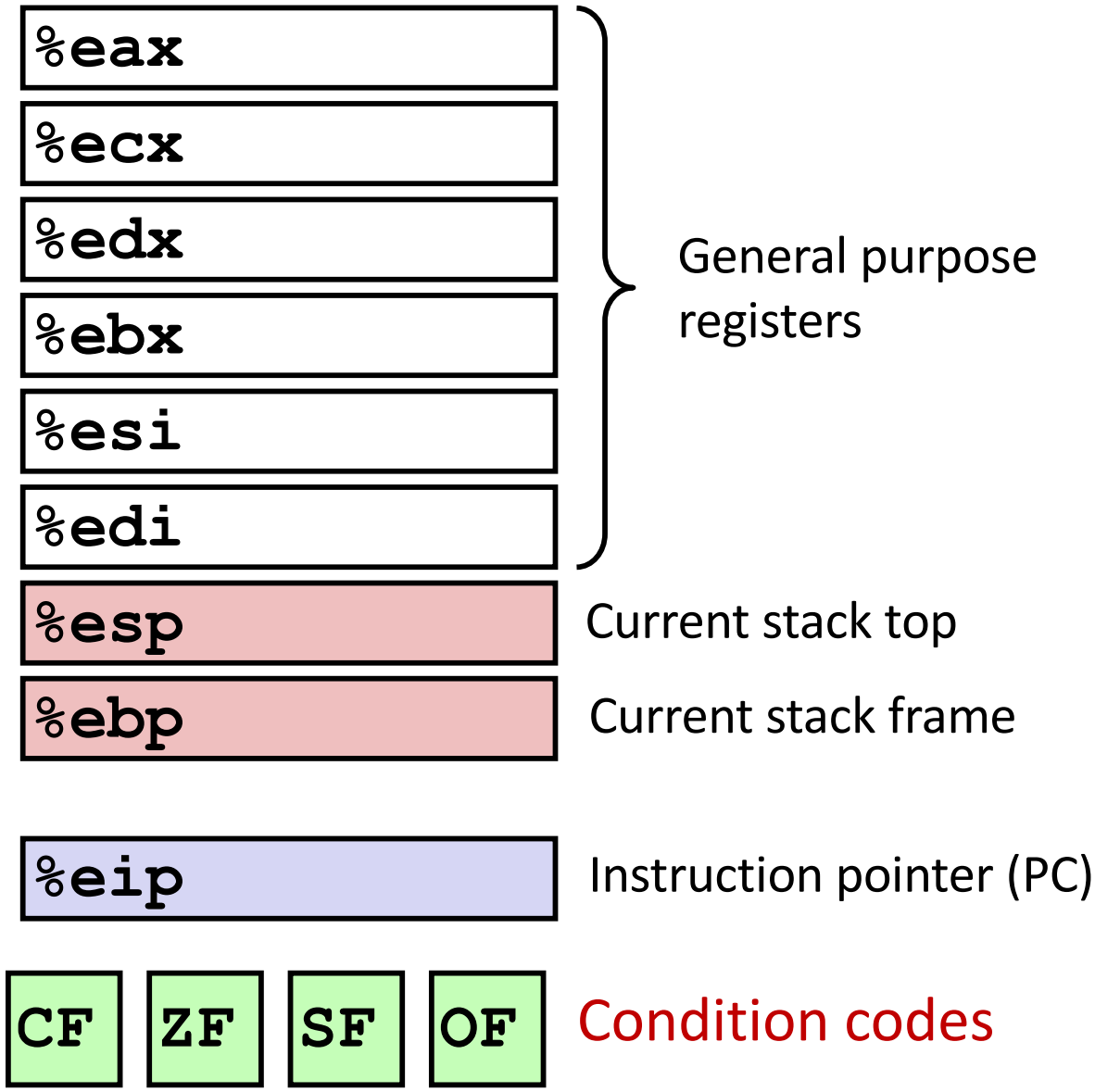
Condition Codes (or Flags)

- Set in two ways:
  1. As “side effects” produced by ALU
  2. In response to explicit comparison instructions

- IA-32, condition codes tell you:
  - If the result is zero (ZF)
  - If the result’s first bit is set (negative if signed) (SF)
  - If the result overflowed (assuming unsigned) (CF)
  - If the result overflowed (assuming signed) (OF)

Processor State in Registers

- Information about currently executing program
  - Temporary data (%eax - %edi)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip, …)
  - Status of recent tests %EFLAGS (CF, ZF, SF, OF)



Instructions that set condition codes

1. Arithmetic/logic side effects (addl, subl, orl, etc.)

2. CMP and TEST:

   `cmpl b,a` like computing **`a-b`** without storing result

   - Sets OF if overflow, Sets CF if carry-out,
     Sets ZF if result zero, Sets SF if results is negative

   `testl b,a` like computing **`a&b`** without storing result

   - Sets ZF if result zero, sets SF if a&b < 0
     OF and CF flags are zero (there is no overflow with &)

Which flags would this `subl` set?

- Suppose %eax holds 5, %ecx holds 7

```
subl $5, %eax
```

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF

Which flags would this `cmpl` set?

- Suppose %eax holds 5, %ecx holds 7

```
cmpl %ecx, %eax
```

If the result is zero (ZF)
If the result's first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF

Conditional Jumping

- Jump based on which condition codes are set

Jump Instructions: (fig. 3.12)

You do not need to memorize these.

	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned jg)
`jb`	`CF`	Below (unsigned)

Example Scenario

```
int userval;
scanf("%d", &userval);

if (userval == 42) {
  userval += 5;
} else {
  userval -= 10;
}
…
```

- Suppose user gives us a value via scanf
- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10

How would we use jumps/CCs for this?

```
int userval;
scanf("%d", &userval);


if (userval == 42) {
  userval += 5;
} else {
  userval -= 10;
}
…
```

Assume userval is stored in %eax at this point.

How would we use jumps/CCs for this?

```
int userval;
scanf("%d", &userval);


if (userval == 42) {
  userval += 5;
} else {
  userval -= 10;
}
…
```

Assume userval is stored in %eax at this point.

(A)
```
  cmpl $42, %eax
  je L2
L1:
  subl $10, %eax
  jmp DONE
L2:
  addl $5, %eax
DONE:
  …
```

(B)
```
  cmpl $42, %eax
  jne L2
L1:
  subl $10, %eax
  jmp DONE
L2:
  addl $5, %eax
DONE:
  …
```

(C)
```
  cmpl $42, %eax
  jne L2
L1:
  addl $5, %eax
  jmp DONE
L2:
  subl $10, %eax
DONE:
  …
```

Loops

- We'll look at these in the lab!

Summary

- ISA defines what programmer can do on hardware
  - Which instructions are available
  - How to access state (registers, memory, etc.)
  - This is the architecture's *assembly language*

- In this course, we'll be using IA-32
  - Instructions for:
    - moving data (movl)
    - arithmetic (addl, subl, imull, orl, sall, etc.)
    - control (jmp, je, jne, etc.)
  - Condition codes for making control decisions
    - If the result is zero (ZF)
    - If the result's first bit is set (negative if signed) (SF)
    - If the result overflowed (assuming unsigned) (CF)
    - If the result overflowed (assuming signed) (OF)