Instruction Set Architecture

9/20/16

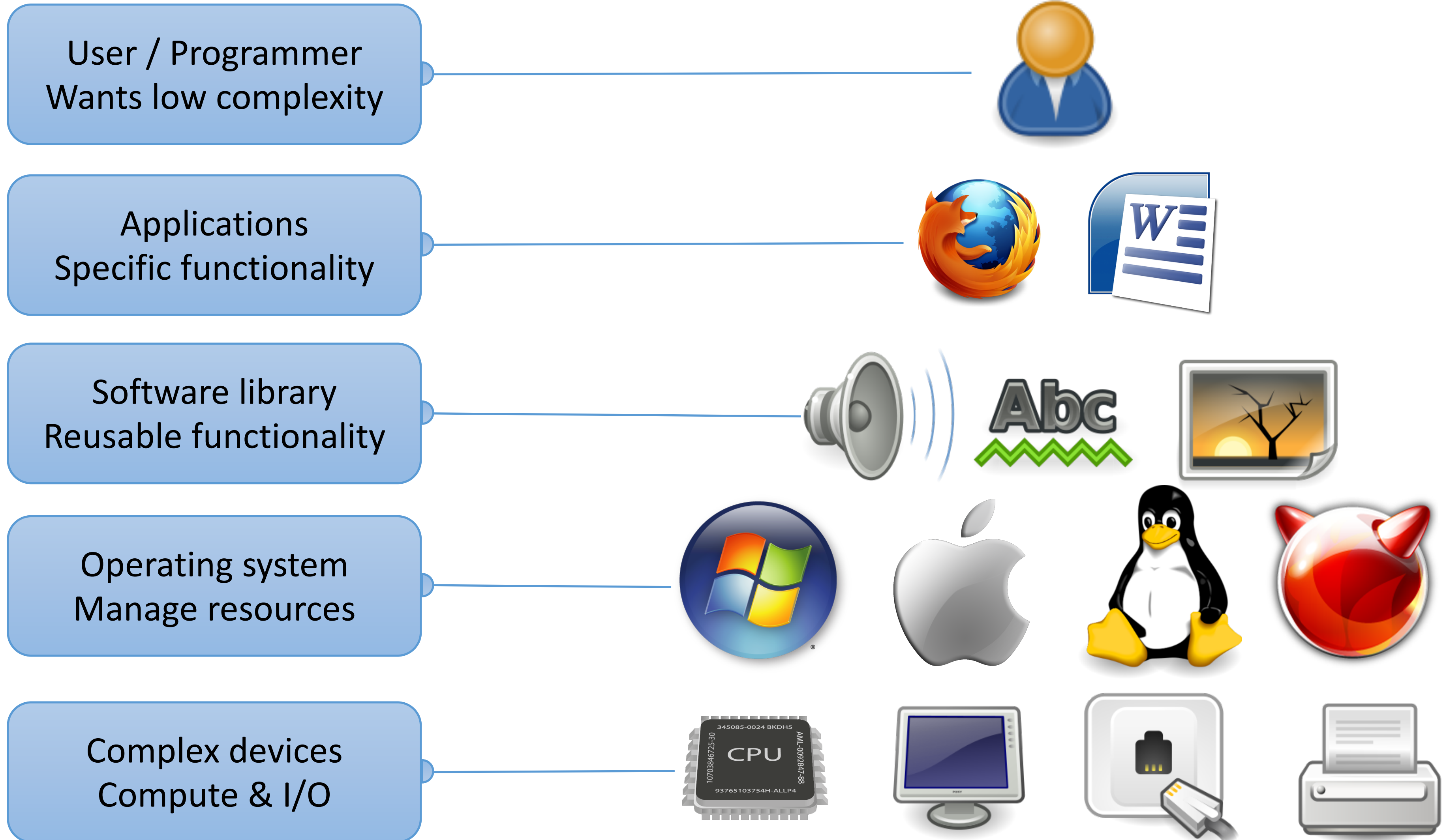
Overview

- How to directly interact with hardware

- Instruction set architecture (ISA)
    - Interface between programmer and CPU
    - Established instruction format (assembly lang)

- Assembly programming (IA-32)

Abstraction

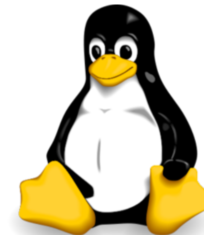


Abstraction

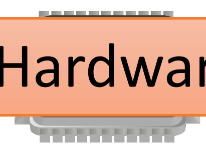
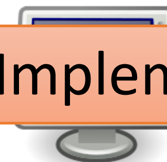
Applications
Specific functionality

This week: Machine Interface

Operating system
Manage resources

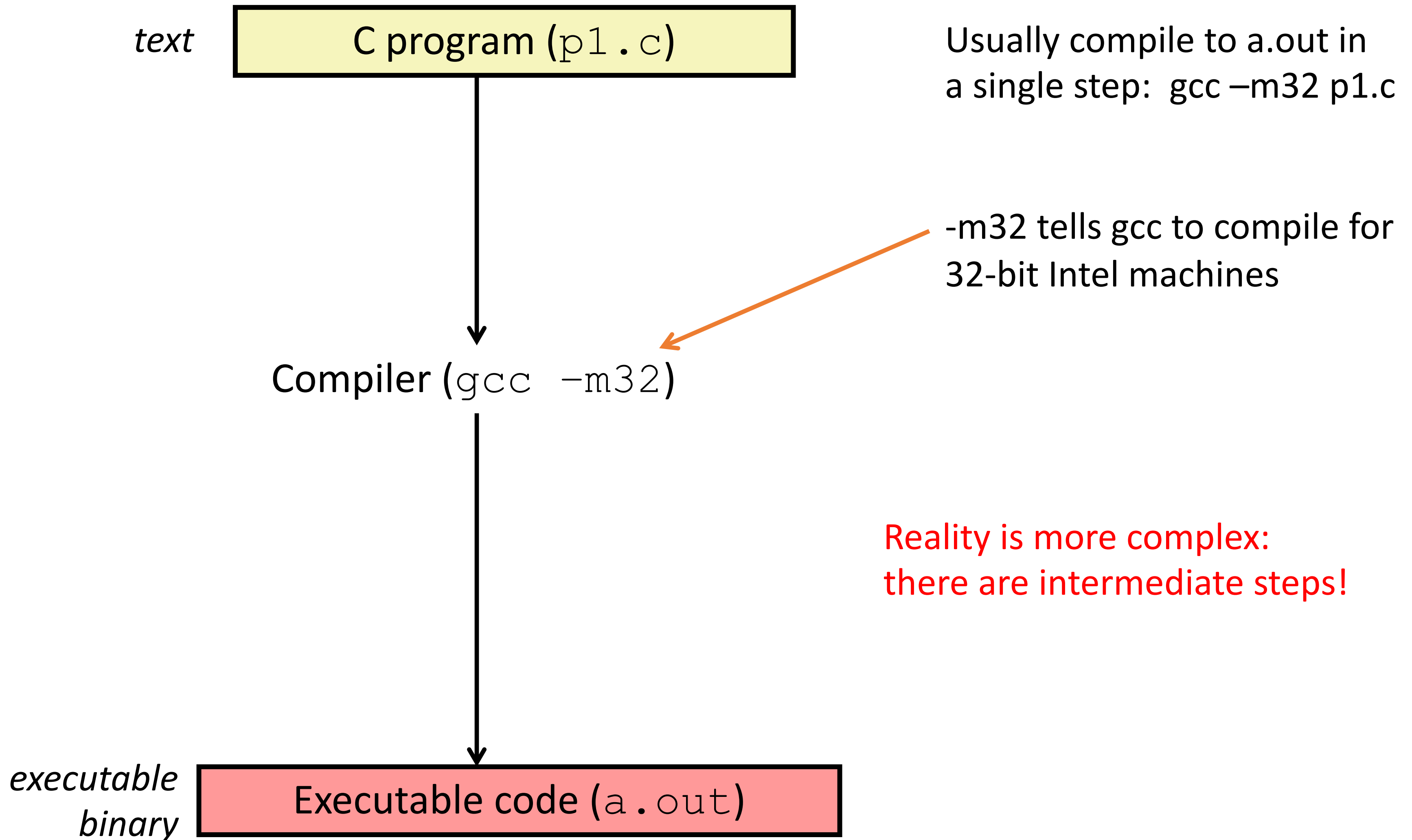


Complex d
Compute

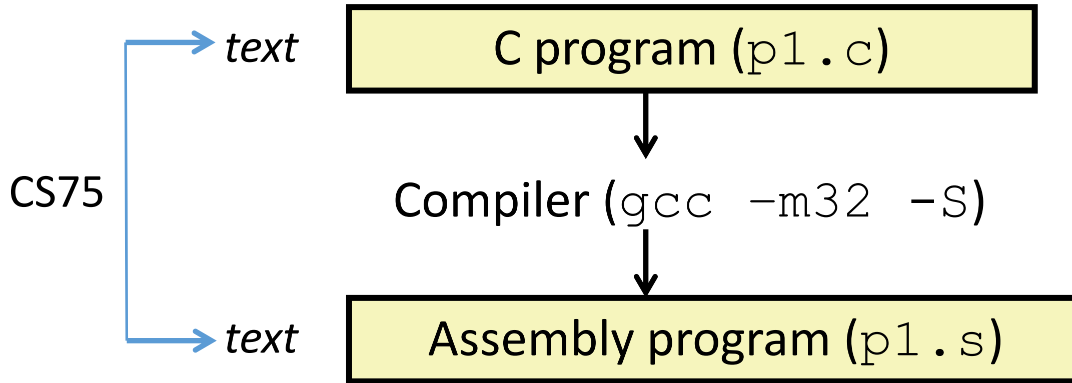



Last week: Circuits, Hardware Implementation

Compilation Steps (.c to a.out)



Compilation Steps (.c to a.out)



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

executable binary

Executable code (a.out)

Assembly Code

Human-readable form of CPU instructions

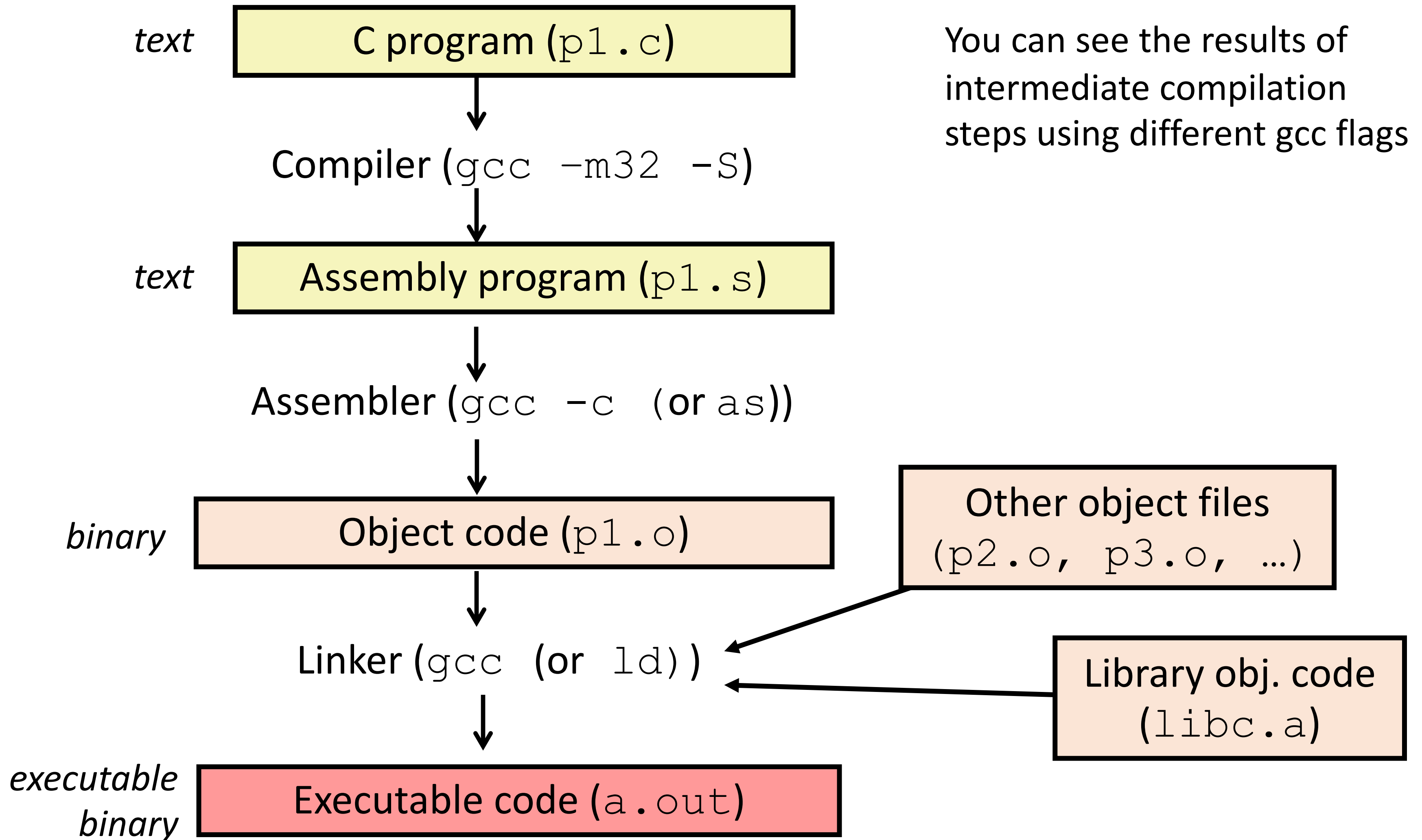
- Almost a 1-to-1 mapping to Machine Code
- Hides some details:
  - Registers have names rather than numbers
  - Instructions have names rather than variable-size codes

We’re going to use IA32 (x86) assembly

- CS lab machines are 64 bit version of this ISA, but they can also run the 32-bit version (IA32)
- Can compile C to IA32 assembly on our system:

```
gcc -m32 -S code.c   # open code.s in vim to view
```

Compilation Steps (.c to a.out)



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

Object / Executable / Machine Code

Assembly

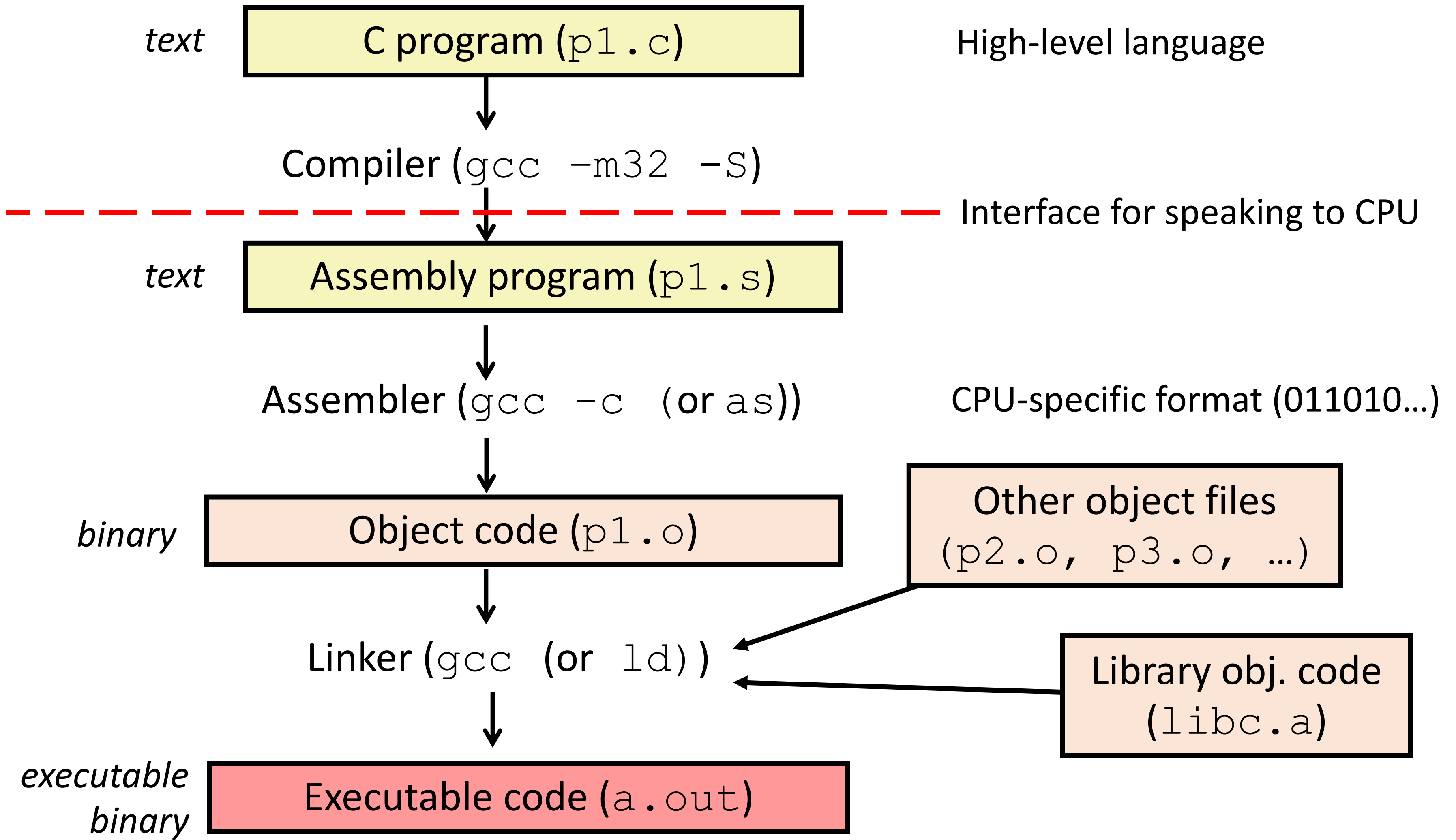
```
push %ebp
mov  %esp, %ebp
sub  $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

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

	a = a + b;

	return a;
}
```

Compilation Steps (.c to a.out)



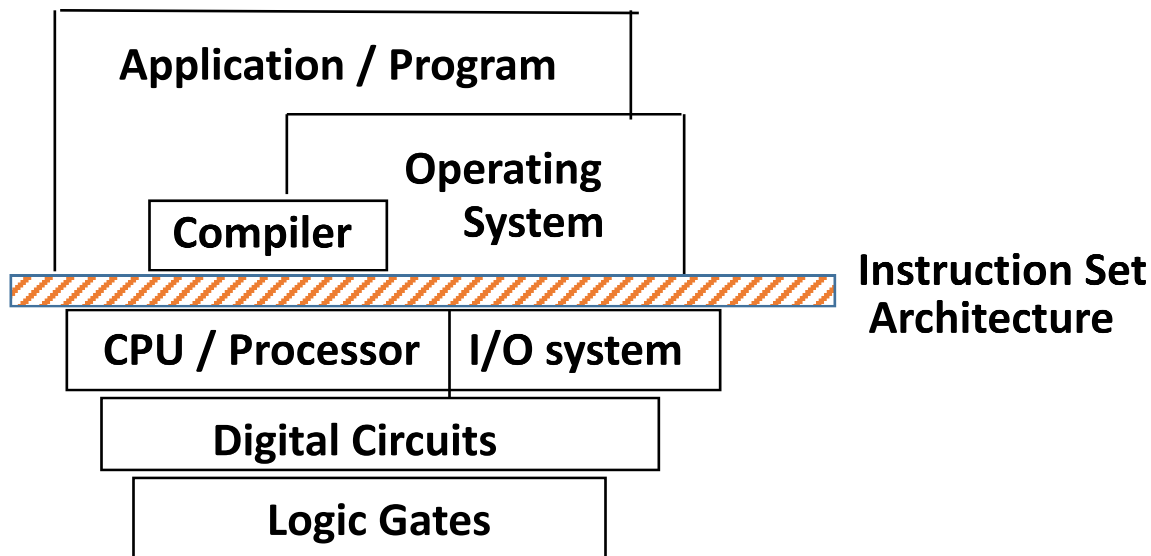
Instruction Set Architecture (ISA)

- 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.



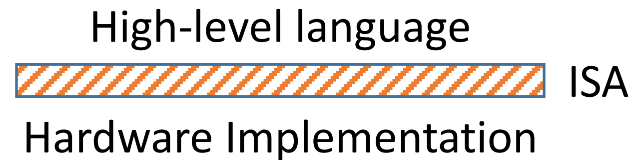
ISA Examples

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k

- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360

How many of these have you used?

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 from sum.c:`

`gcc -m32 -S sum.c`

`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
```

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 from sum.c:

gcc -m32 -S sum.c

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
```

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

Intel x86 Family (IA-32)

Intel i386 (1985)

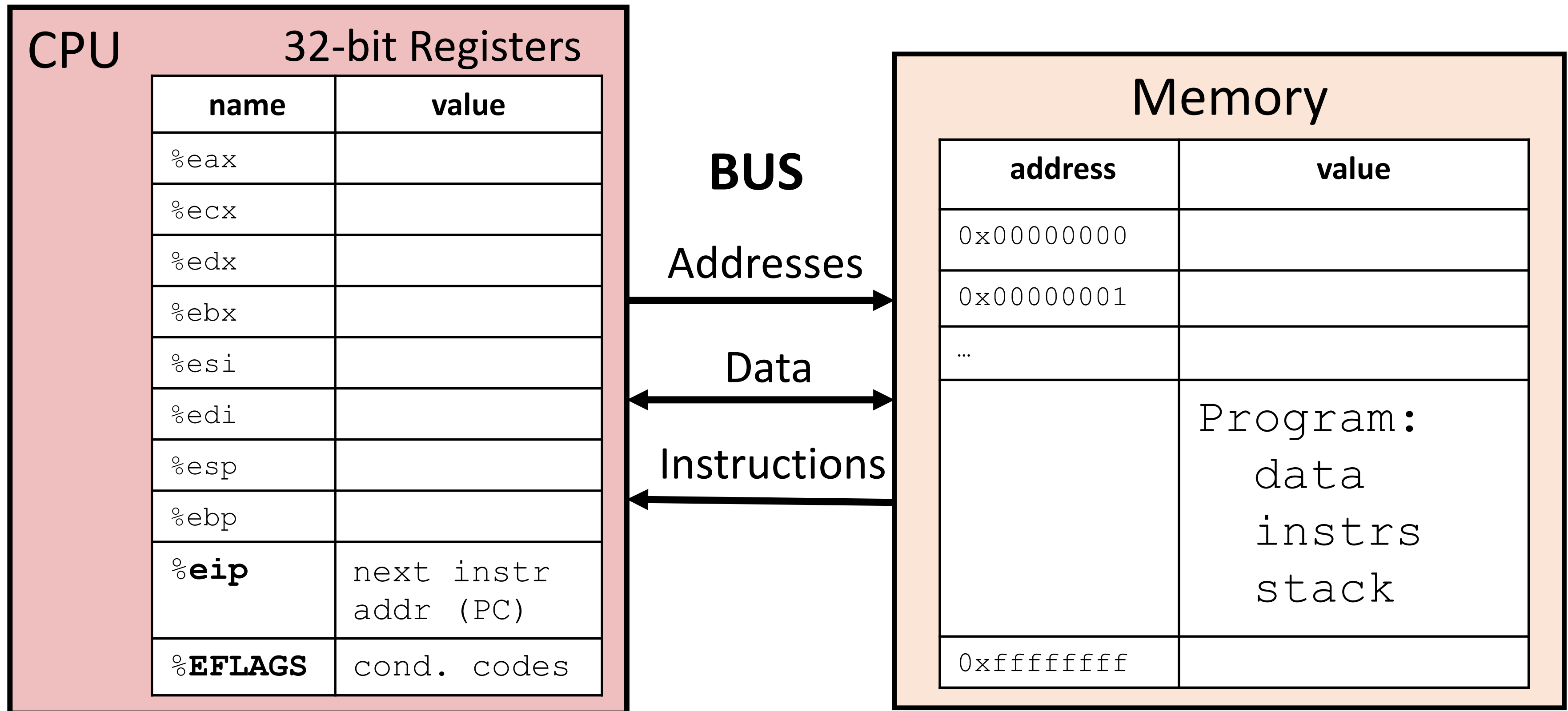
- 12 MHz - 40 MHz
- ~300,000 transistors
- Component size: 1.5 µm

Intel Core i7 4770k (2013)

- 3,500 MHz
- ~1,400,000,000 transistors
- Component size: 22 nm

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

Assembly Programmer's View of State



Registers:

PC: Program counter (`%eip`)

Condition codes (`%EFLAGS`)

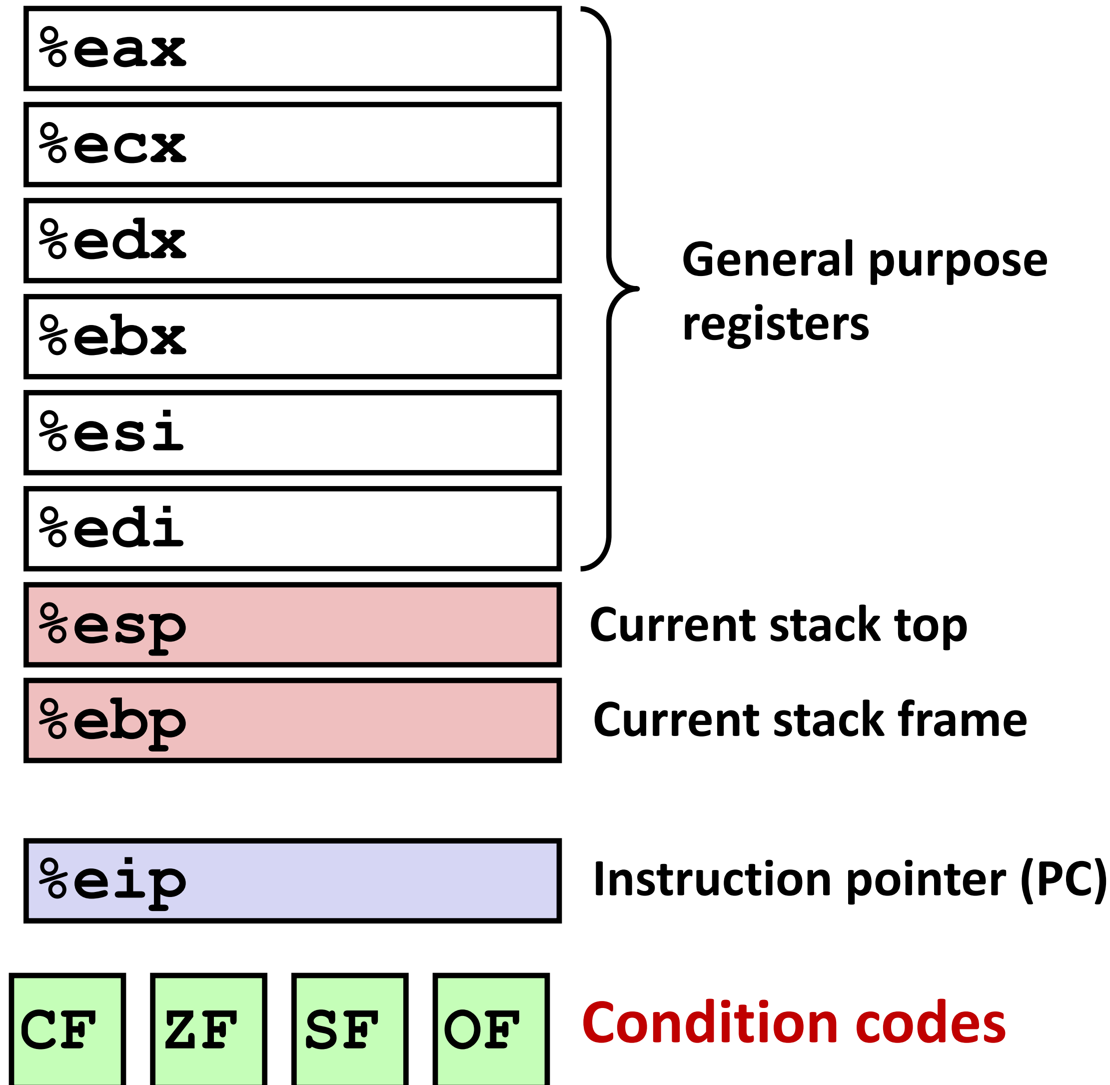
General Purpose (`%eax` - `%ebp`)

Memory:

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

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)



General purpose Registers

- Remaining Six are for instruction operands
  - Can store 4 byte data or address value (ex. 3 + 5)

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

The low-order 2 bytes and two low-order 1 bytes of some of these can be named.

%ax is the low-order 16 bits of %eax

%al is the low-order 8 bits of %eax

May see their use in ops involving shorts or chars

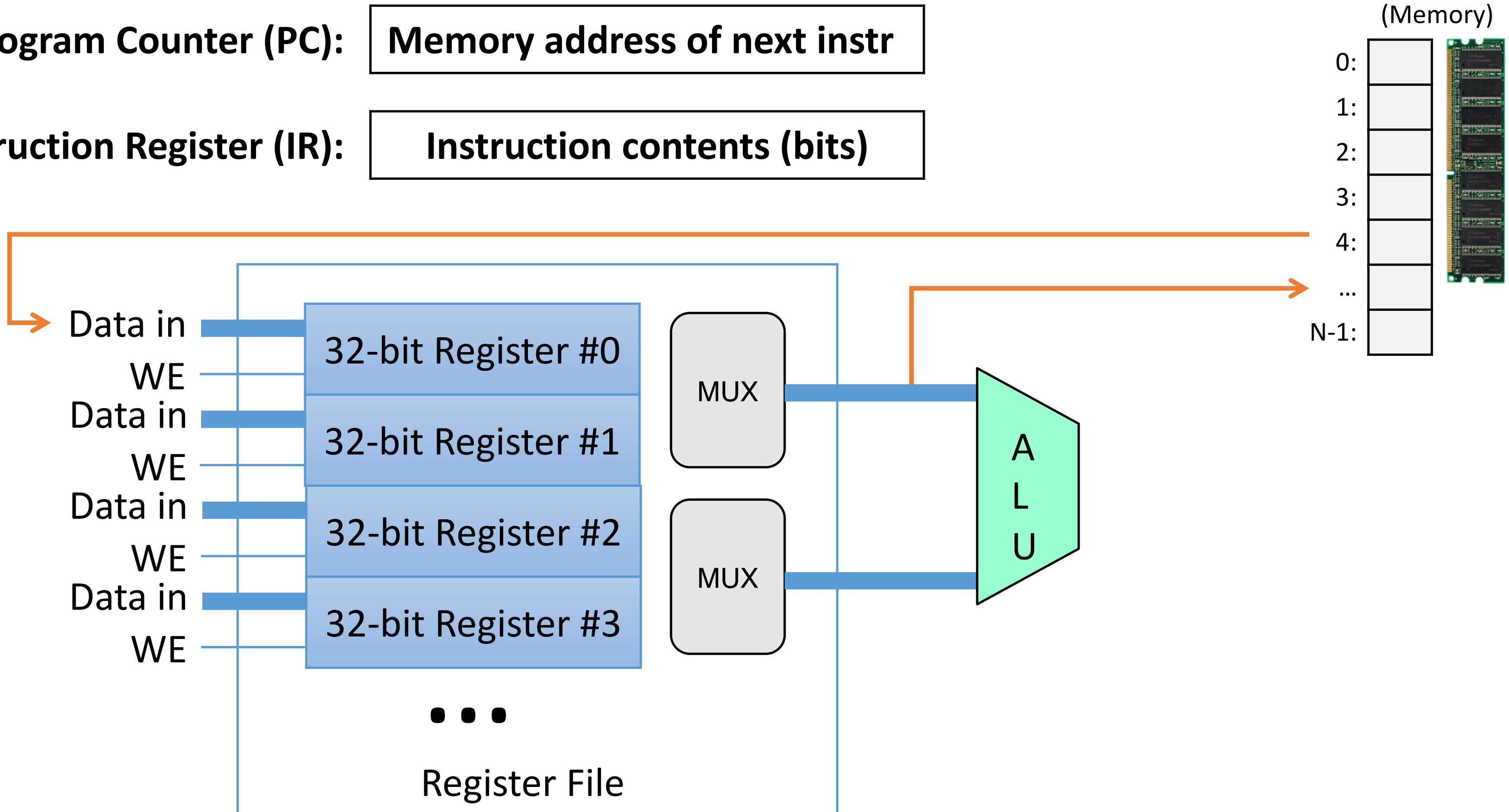
31 16		15 8	7 0
%eax	%ax	%ah	%al
%ecx	%cx	%ch	%cl
%edx	%dx	%dh	%dl
%ebx	%bx	%bh	%bl
%esi	%si		
%edi	%di		
%esp	%sp		
%ebp	%bp		

Types of IA32 Instructions

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

Data Movement

Move values between memory and registers or between two registers.



Arithmetic

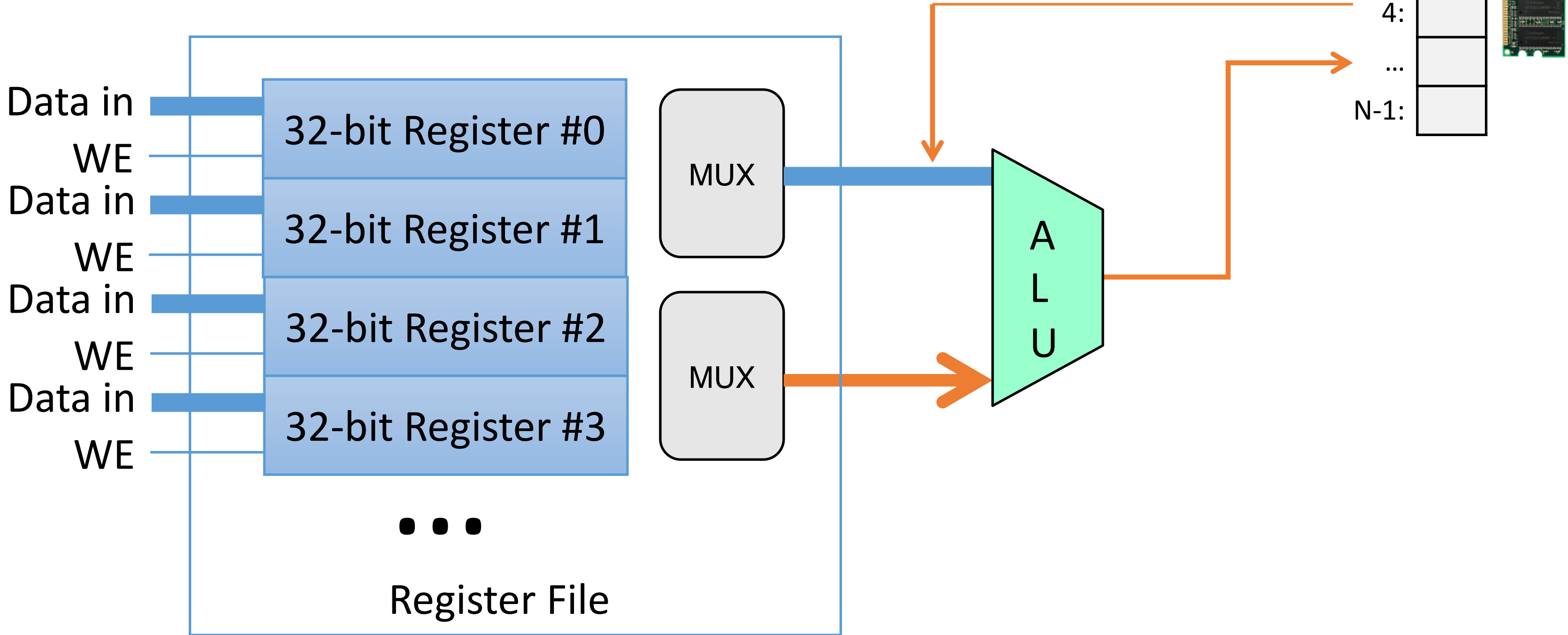
Use ALU to compute a value, store result in register / memory.

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

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

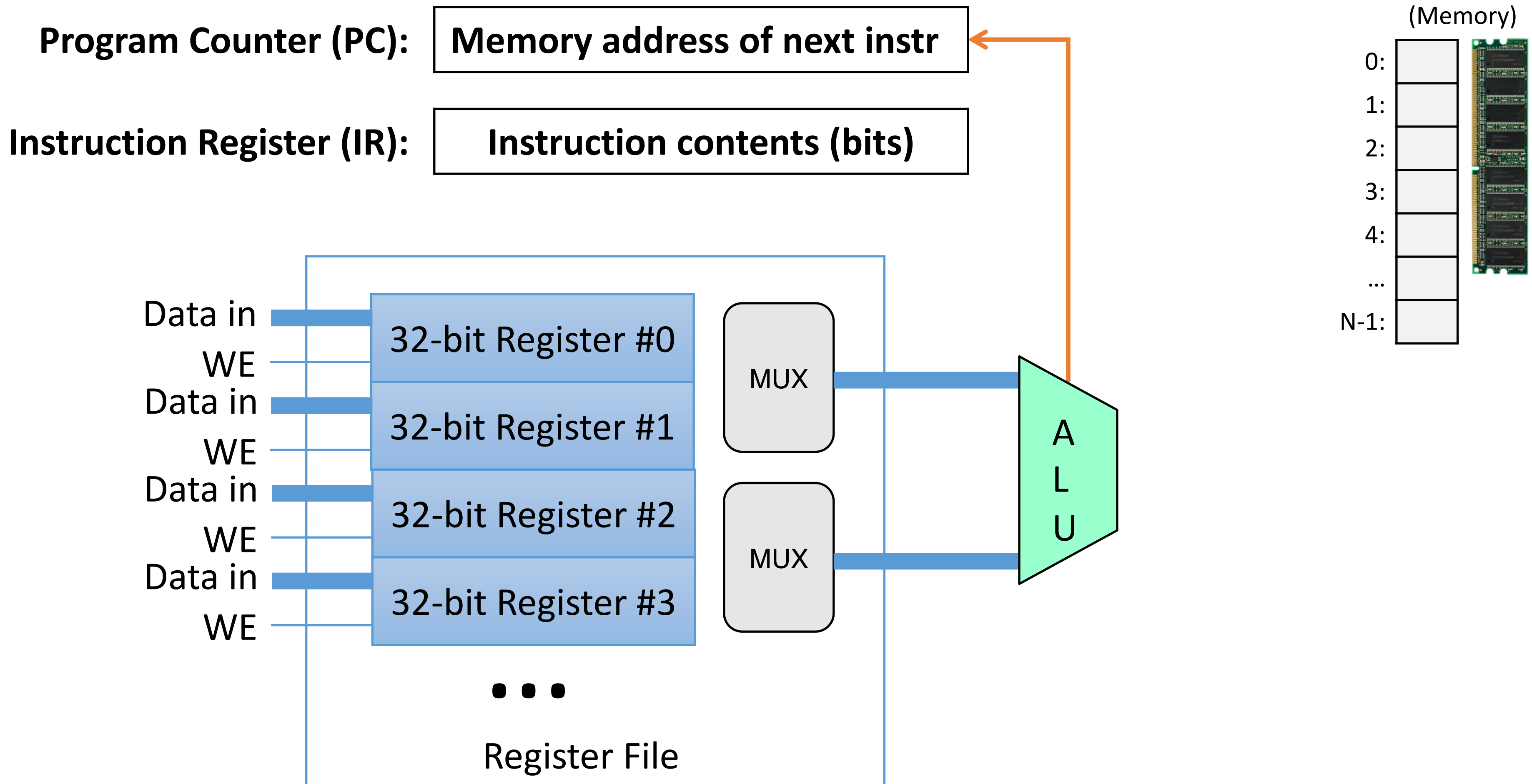
(Memory)

0:
1:
2:
3:



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.

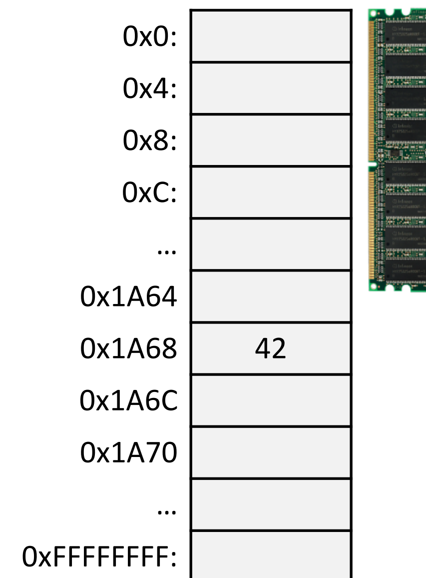
- `movl (%ecx), %eax`
  - Use the ***address in*** register `ecx` to access memory, store result in register `eax`

Addressing Mode: Memory

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

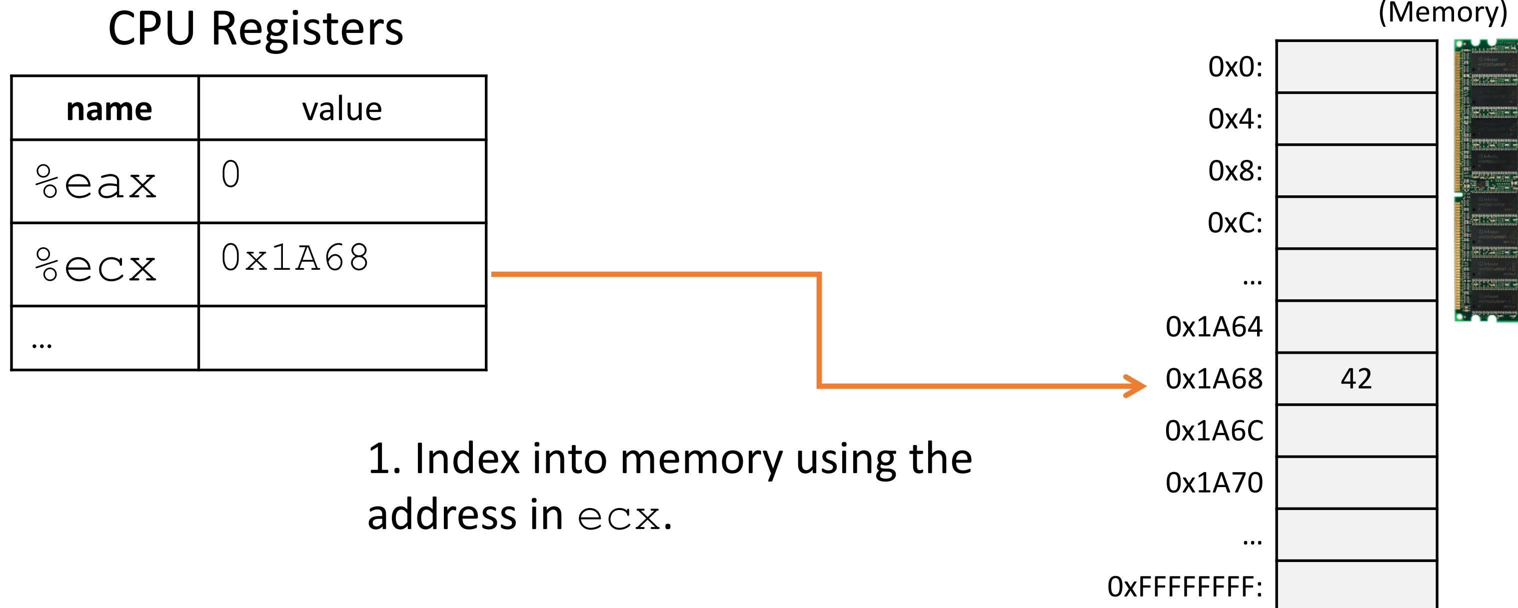
CPU Registers

name	value
`%eax`	`0`
`%ecx`	`0x1A68`
…	



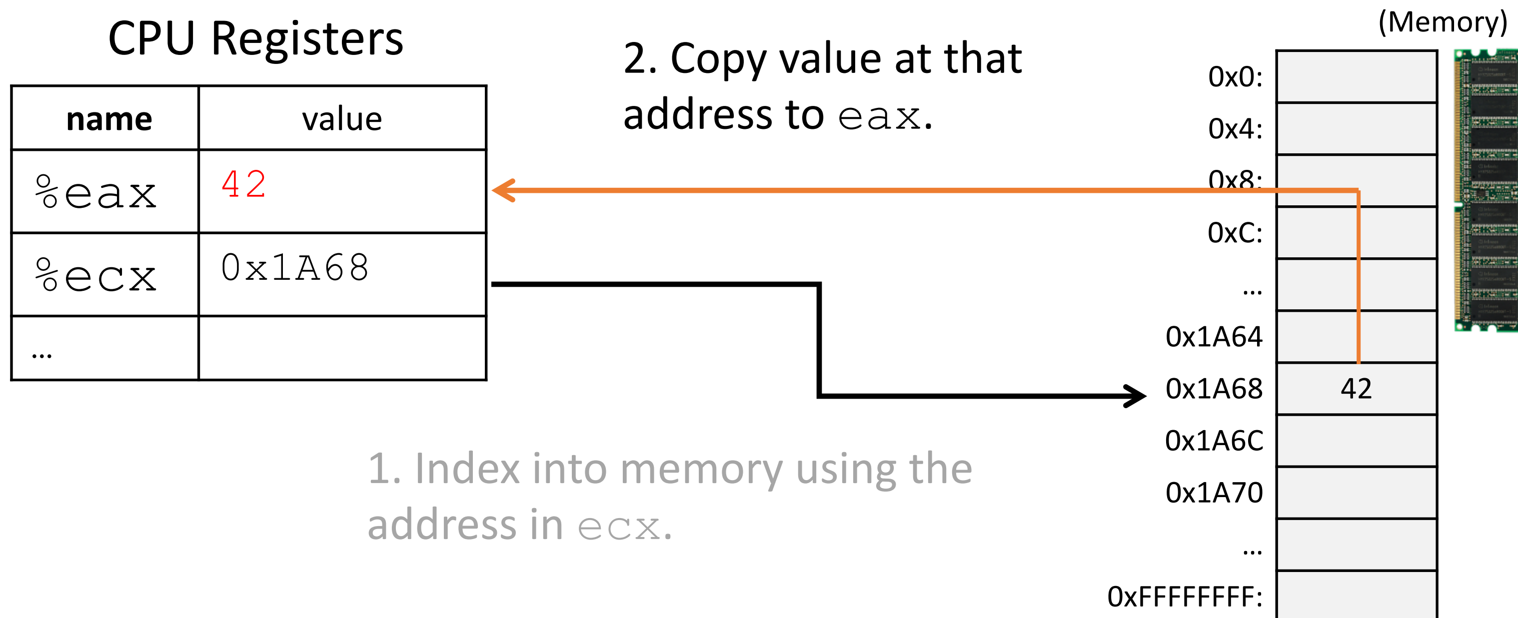
Addressing Mode: Memory

- `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 `12` from it, index into memory and store the result in `eax`

Addressing Mode: Displacement

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

CPU Registers

name	value
`%eax`	`0`
`%ecx`	`0x1A68`
`%ebp`	`0x1A70`
...	

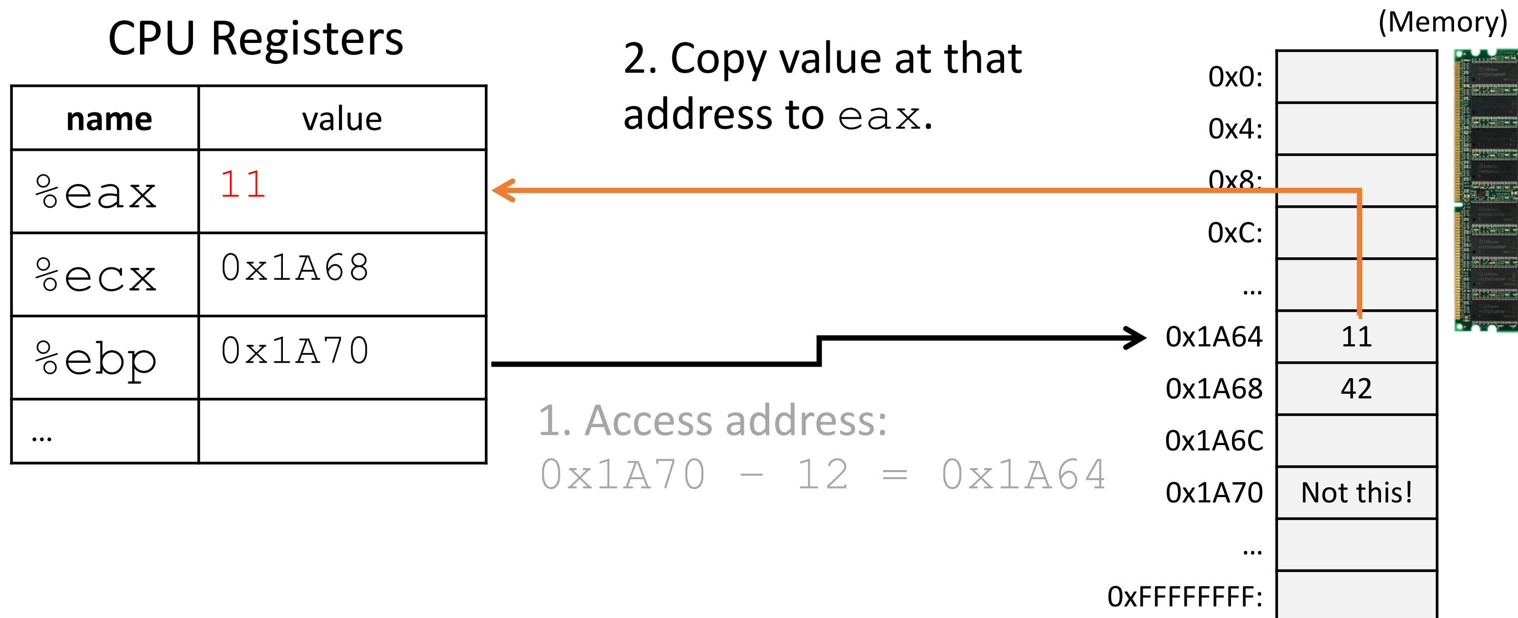
1. Access address:
`0x1A70 - 12 = 0x1A64`

(Memory)

0x0:	
0x4:	
0x8:	
0xC:	
...	
0x1A64	11
0x1A68	42
0x1A6C	
0x1A70	
...	
0xFFFFFFFF:	

Addressing Mode: Displacement

- `movl -12(%ebp), %eax`
  - Take the address in `ebp`, subtract `12` 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)
```

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

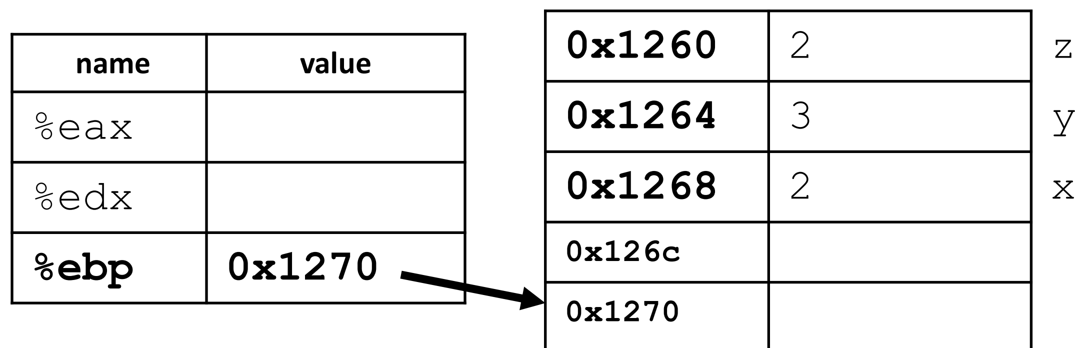
x is 2 at `%ebp-8`, y is 3 at `%ebp-12`, z is 2 at `%ebp-16`

```
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;
```



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)
```

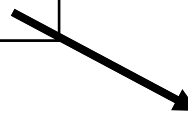
name	value
%eax	?
%ecx	?
%ebp	**0x456C**

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

Solution

```
movl %ebp, %ecx          # %ecx = 0x456c
subl $16, %ecx           # %ecx = 0x455c
movl (%ecx), %eax        # %eax = 7
orl  %eax, -8(%ebp)      # (4564) = 111 | 101
negl %eax                # %eax = -7
movl %eax, 4(%ecx)       # (4560) = -7
```

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



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