Agenda

Intro to Assembly: From programs to machine code

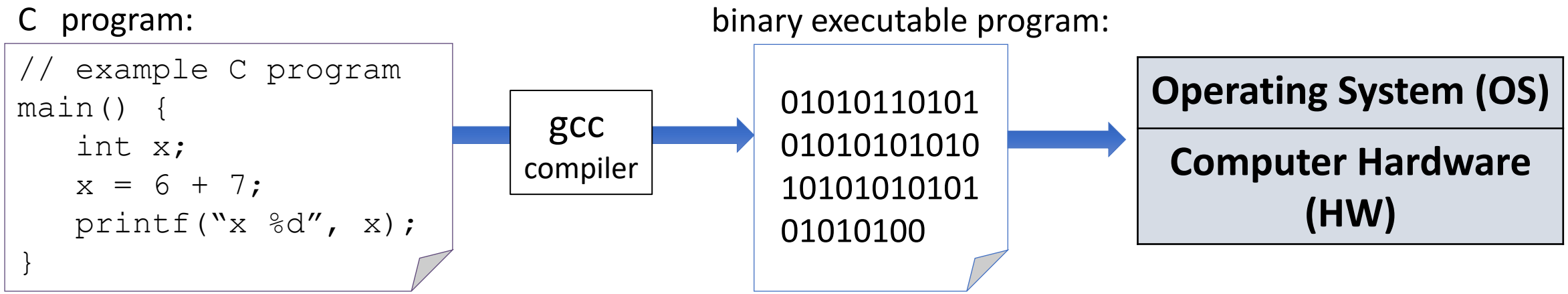
Generating and compiling assembly

Assembly concepts

- Instructions (Operations, Control Flow, Moving data)
- Registers
- von Neuman architecture: stored program model of computation

From programs to machine code

A program written in a high-level language is compiled (e.g. translated) into a binary (0's and 1's) executable:



How is machine code run?

When we say **computer architecture** in this class, we are referring to the design of the computer's processor, not the entire hardware system of the computer

Central processing unit (CPU) executes programs

Random access memory (RAM) stores program instructions + data

Instruction set architecture (ISA) refers to the commands the processor can execute along with how command operands are stored

Assembly language is based on an ISA

Assembly

```
114e:      48 89 e5                 mov    %rsp,%rbp
1151:      48 83 ec 10              sub    $0x10,%rsp
1155:      c7 45 fc 0d 00 00 00     movl   $0xd,-0x4(%rbp)
```

Instruction Address | Machine Language | Assembly Language

Assembly language is a readable form of machine code

-> directly represents the commands that the CPU executes

A **compiler** translates human-readable code into machine code

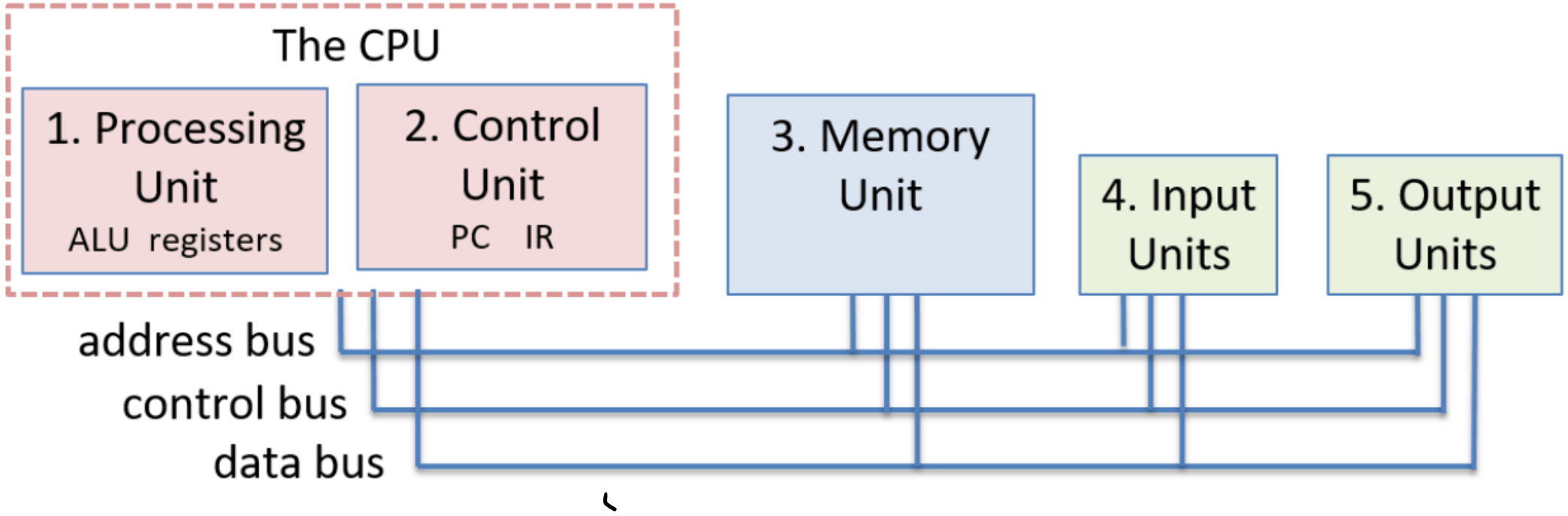
The **Instruction Set Architecture (ISA)** depends on your OS and hardware. We will use **x86_64**

To see what your machine supports: **$ uname -p**

Sec 5.1 (history)
Sec 5.2

Von Neuman Architecture

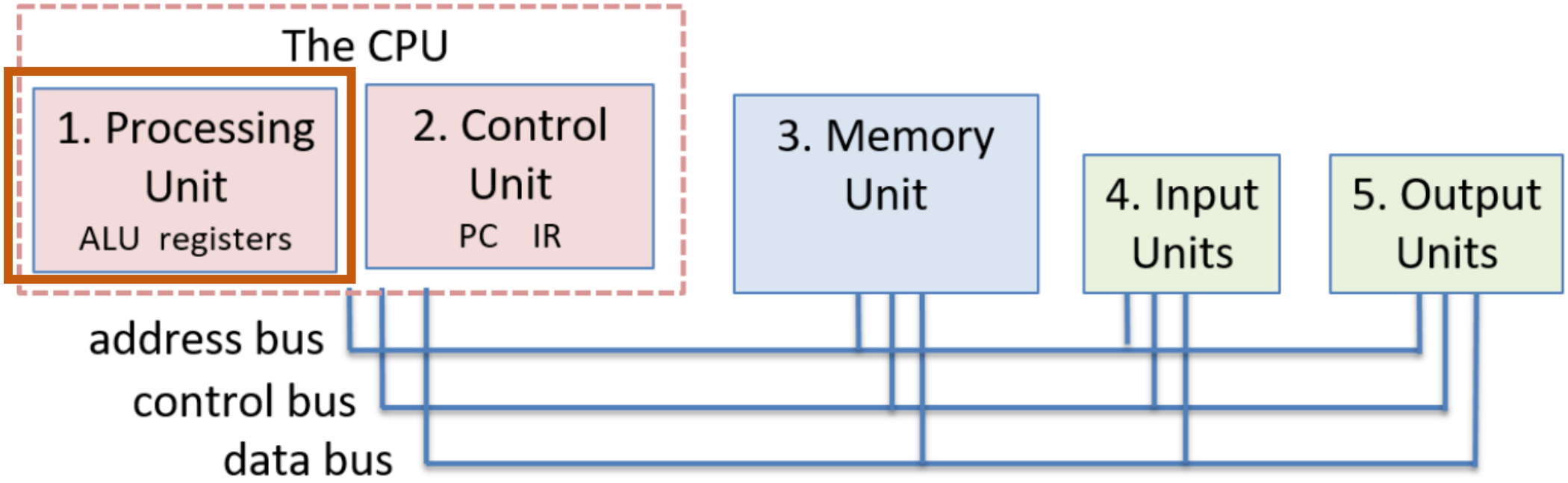
Idea: Stored program model. Hardware is like paper and software is like writing.



Almost all computer architectures in use today are based on the Von Neuman architecture.

When we run machine code, we run it on this type of architecture

Von Neuman Architecture

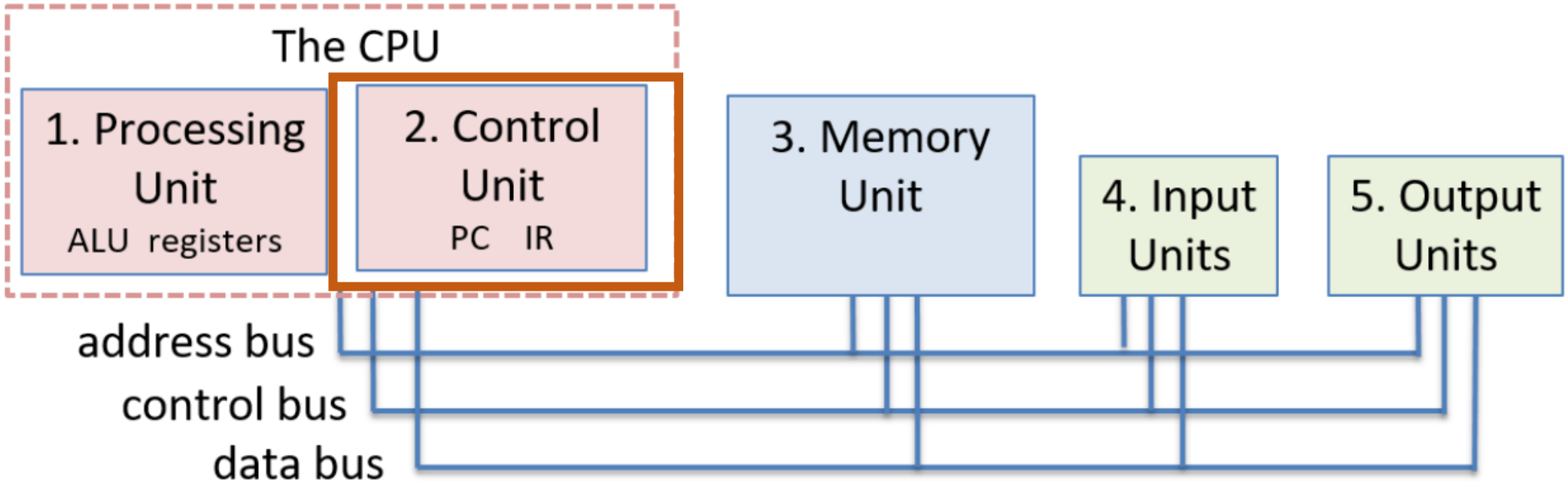


Processing unit contains

- arithmetic/logic unit
- registers

A **register** is a small, fast unit of memory that stores program instructions + data

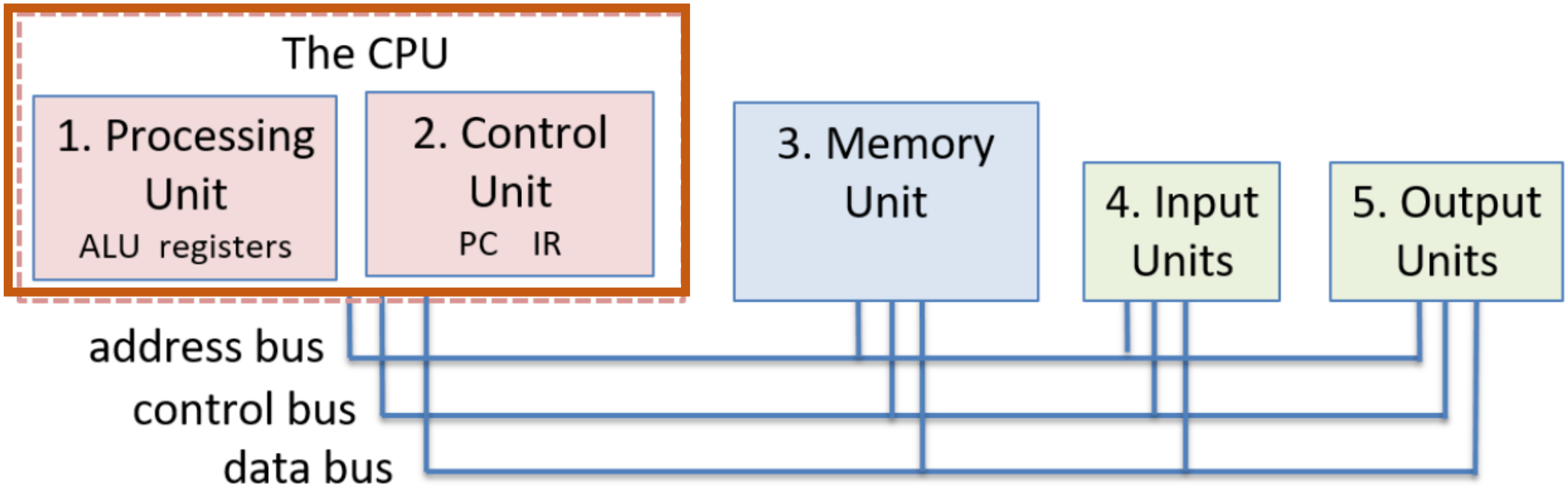
Von Neuman Architecture



The **control unit** drives the execution of the program. It contains the

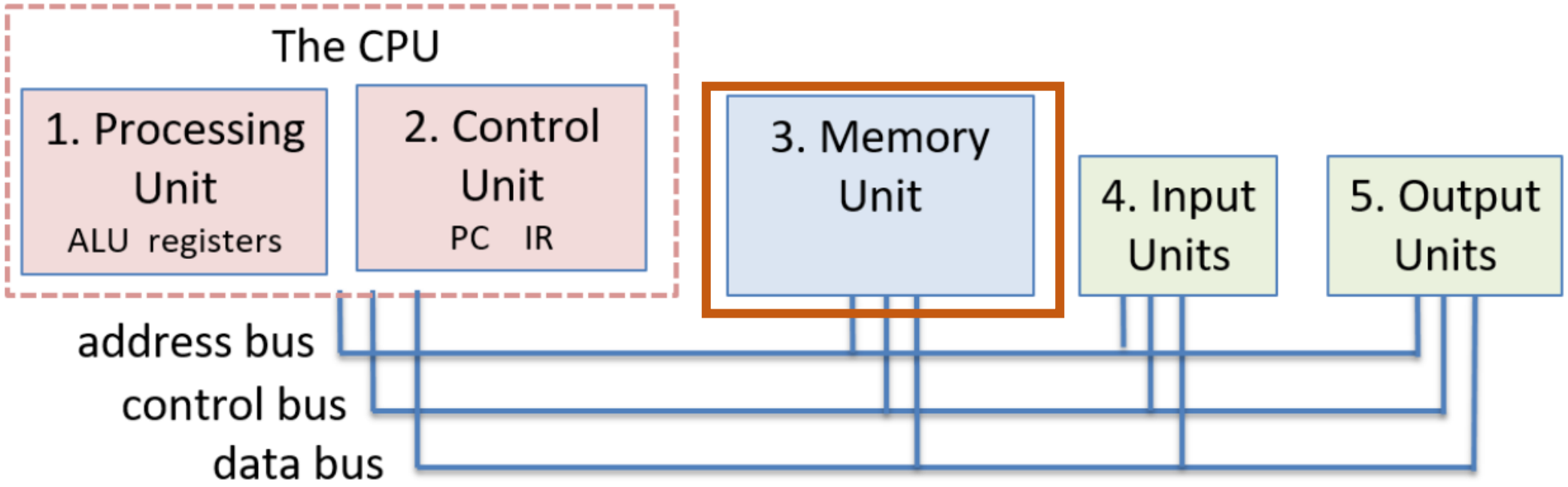
- **program counter (PC)**, which keeps the address of the next instruction to execute
- **instruction register (IR),** which contains the currently executing instruction

Von Neuman Architecture



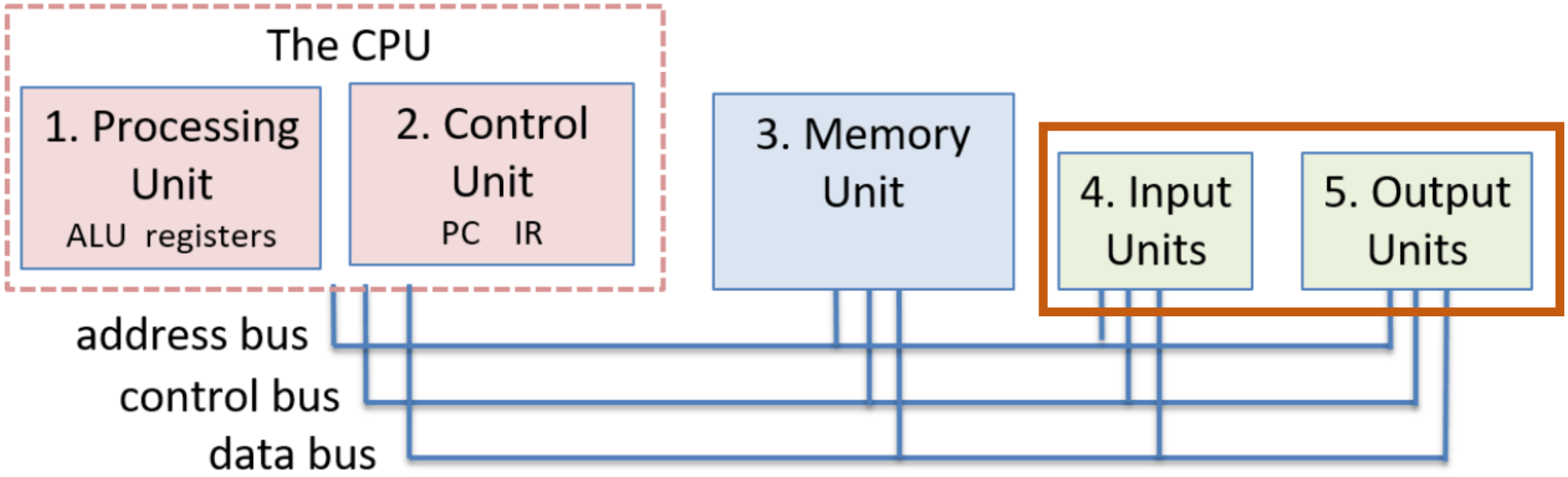
The Processing Unit and Control Unit together make up the Central Processing Unit (CPU)

Von Neuman Architecture



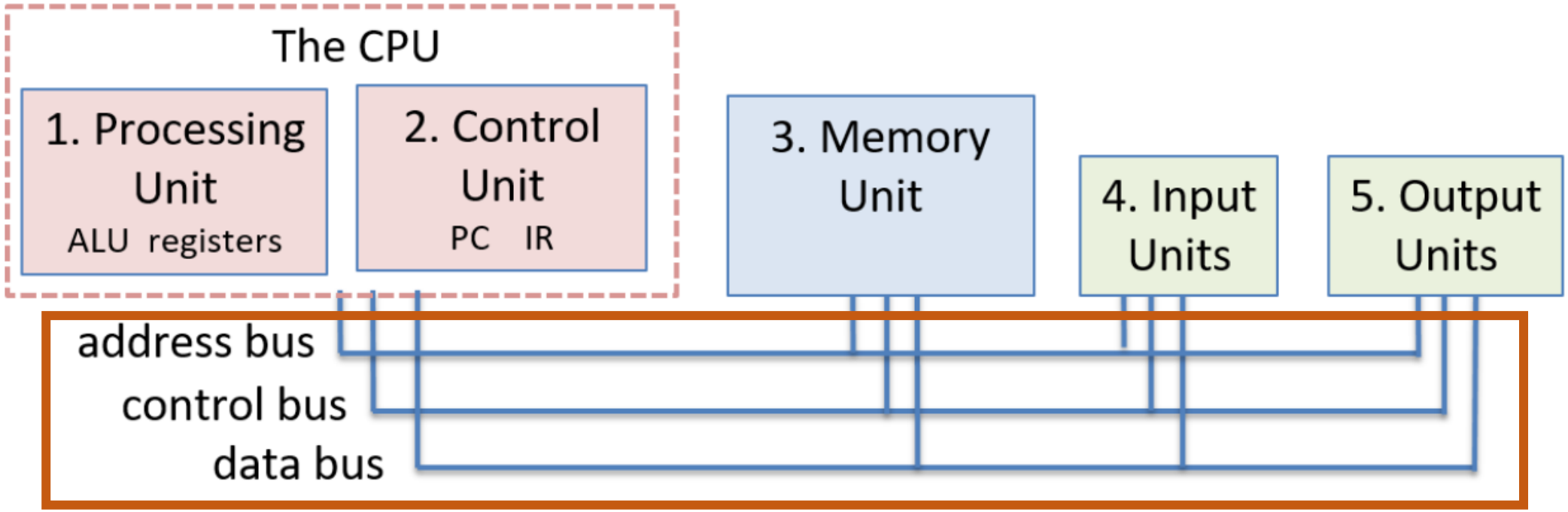
The **Memory Unit** stores both program data and instructions. For now, this can be thought of as the same as **RAM.** The memory unit allows the CPU to access data in memory very quickly.

Von Neuman Architecture



The **Input Units** represent devices such as the keyboard, camera, and mouse. The **Output Units** represent devices that display the results of computations, such as your monitor or speakers. Some devices are both input and output devices. Can you think of one?

Von Neuman Architecture



A **bus** is a communication channel between components (wires). Typically, there are separate buses for sending data, accessing memory, and commands between units.

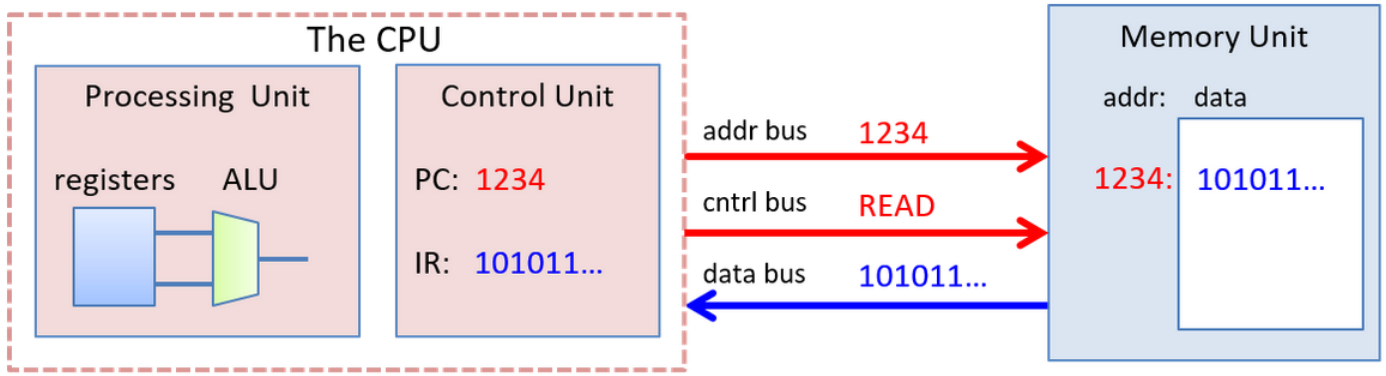
Executing a program

```
mov    %rsp,%rbp
sub    $0x10,%rsp
movl   $0xd,-0x4(%rbp)
mov    -0x4(%rbp),%eax
mov    %eax,%esi
```

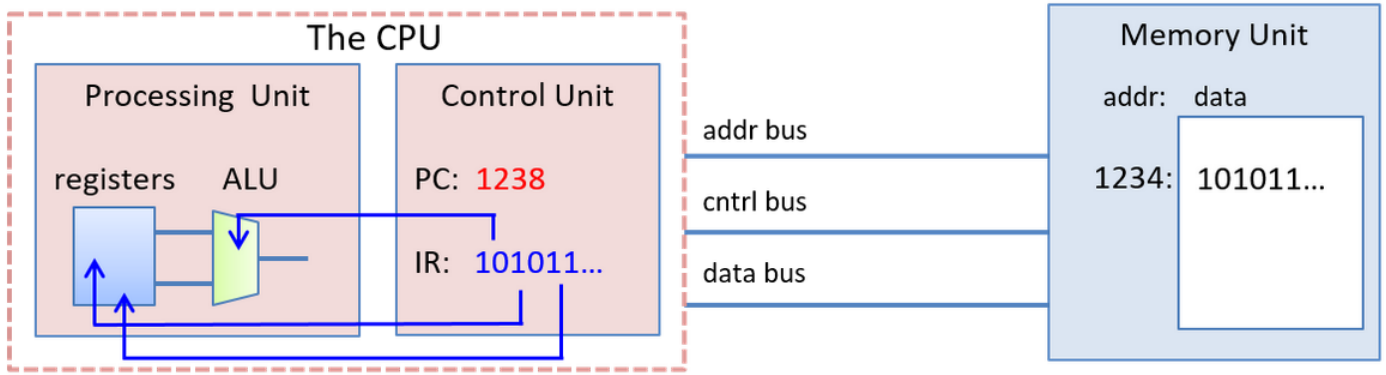
1. The control unit **fetches** the next instruction
2. The control unit **decodes** the instruction in the IR
3. The processing unit **executes** the instruction
4. The control unit **stores** the result to memory

Example: Adding two numbers

add %eax, ,-0x4(%rbp)

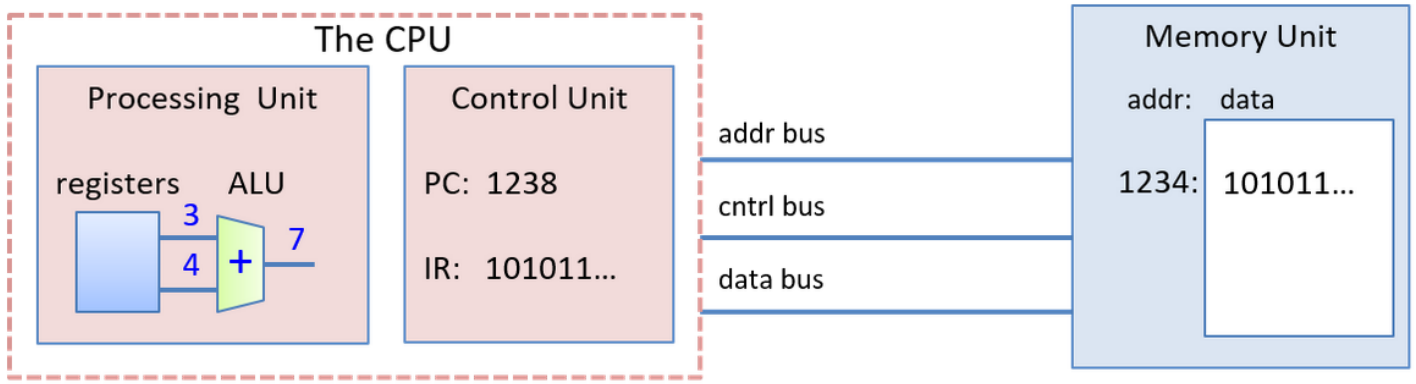


1. Fetch: Read instruction bits from memory at address in PC (1234), and store in IR

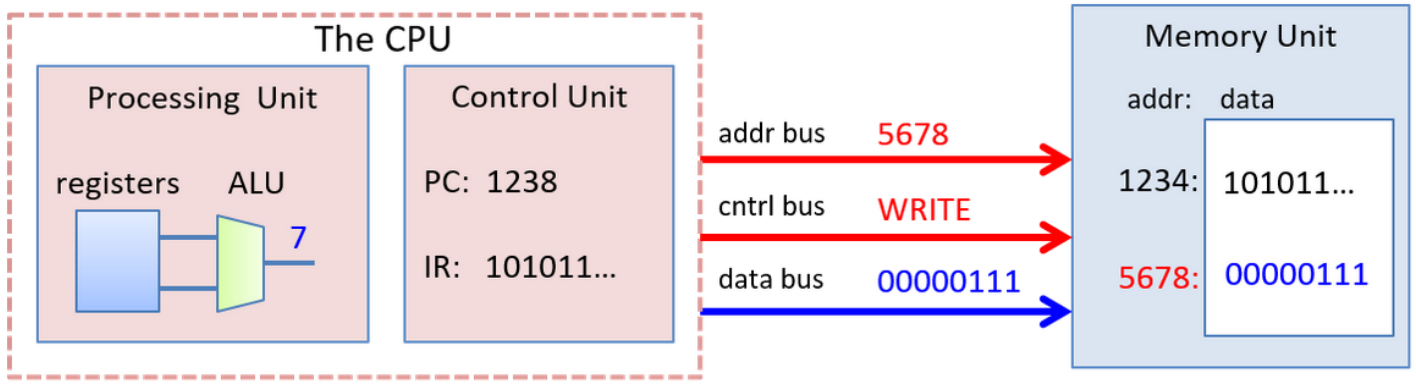


2. Decode: instruction bits in IR encode which registers store operands & the ALU operation

Example: Adding two numbers



3. Execute: ALU performs instruction operation (+) on operands (3,4) to compute result (7)



4. Store: the control unit stores the ALU result (7, binary 00000111) to memory

Example: From programs to machine code

```
#include <stdio.h>

int main() {
  int x;
  x = 6 + 7;
  printf("x %d\n", x);
  return 0;
}
```

The compiler translates a high-level language to machine code through a series of steps: pre-compiling, compiling, and linking

```
00000000   7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  03 00 3E 00  .ELF..............>.
00000014   01 00 00 00  60 10 00 00  00 00 00 00  40 00 00 00  00 00 00 00  ....`.......@.......
00000028   78 39 00 00  00 00 00 00  00 00 00 00  40 00 38 00  0D 00 40 00  x9..........@.8...@.
```

```
00000000   7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  03 00 3E 00  .ELF..............>.
00000014   01 00 00 00  60 10 00 00  00 00 00 00  40 00 00 00  00 00 00 00  ....`.......@.......
00000028   78 39 00 00  00 00 00 00  00 00 00 00  40 00 38 00  0D 00 40 00  x9..........@.8...@.
```

```
0000000000001149 <main>:
   1149: f3 0f 1e fa                    endbr64
   114d: 55                             push   %rbp
   114e: 48 89 e5                       mov    %rsp,%rbp
   1151: 48 83 ec 10                    sub    $0x10,%rsp
   1155: c7 45 fc 0d 00 00 00           movl   $0xd,-0x4(%rbp)
   115c: 8b 45 fc                       mov    -0x4(%rbp),%eax
   115f: 89 c6                          mov    %eax,%esi
   1161: 48 8d 3d 9c 0e 00 00           lea    0xe9c(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
   1168: b8 00 00 00 00                 mov    $0x0,%eax
   116d: e8 de fe ff ff                 callq  1050 <printf@plt>
   1172: b8 00 00 00 00                 mov    $0x0,%eax
   1177: c9                             leaveq
   1178: c3                             retq
   1179: 0f 1f 80 00 00 00 00           nopl   0x0(%rax)
```

The compiler can show you the results of each step. For example, we can look at the assembly code.

Example: simple.c

```
// simple.c
#include <stdio.h>

int main() {
  int x;
  x = 6 + 7;
  printf("x %d\n", x);
  return 0;
}
```

To produce human readable machine code using **objdump**

$ objdump –d <exe> > output.txt

To produce programmable assembly code use **gcc**

$ gcc simple.c –S –o simple.S

To compile assembly code

$ gcc simple.S

Example: simple.c (objdump output)

```
0000000000001149 <main>:
    1149:       f3 0f 1e fa                 endbr64
    114d:       55                          push   %rbp
    114e:       48 89 e5                    mov    %rsp,%rbp
    1151:       48 83 ec 10                 sub    $0x10,%rsp
    1155:       c7 45 fc 0f 00 00 00        movl   $0xf,-0x4(%rbp)
    115c:       8b 45 fc                    mov    -0x4(%rbp),%eax
    115f:       89 c6                       mov    %eax,%esi
    1161:       48 8d 3d 9c 0e 00 00        lea    0xe9c(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    1168:       b8 00 00 00 00              mov    $0x0,%eax
    116d:       e8 de fe ff ff              callq  1050 <printf@plt>
    1172:       b8 00 00 00 00              mov    $0x0,%eax
    1177:       c9                          leaveq
    1178:       c3                          retq
    1179:       0f 1f 80 00 00 00 00        nopl   0x0(%rax)
```

Example: simple.S

C instructions are converted to one or more assembly instructions

Find the instruction(s) that corresponds to the line

x = 6 + 7

```
main:
.LFB0:
  .cfi_startproc
  endbr64
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  subq  $16, %rsp
  movl  $13, -4(%rbp)
  movl  -4(%rbp), %eax
  movl  %eax, %esi
  leaq  .LC0(%rip), %rdi
  movl  $0, %eax
  call  printf@PLT
  movl  $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

Why learn assembly?

To understand what is happening under the hood

Some computing systems are too hardware constrained for compilers

Vulnerability analysis

Critical code in system-level software

Understanding program behavior

What is the output of this program?

```
int adder() {
  int a;
  return a + 2;
}
int assign() {
  int y = 40;
  return y;
}
int main() {
  int x;
  assign();
  x = adder();
  printf("x is: %d\n", x);
  return 0;
}
```

What types of commands does assembly support?

Arithmetic and logical operations

Control flow

Moving data between CPU and memory

Pushing/popping the function stack

Assembly Basics

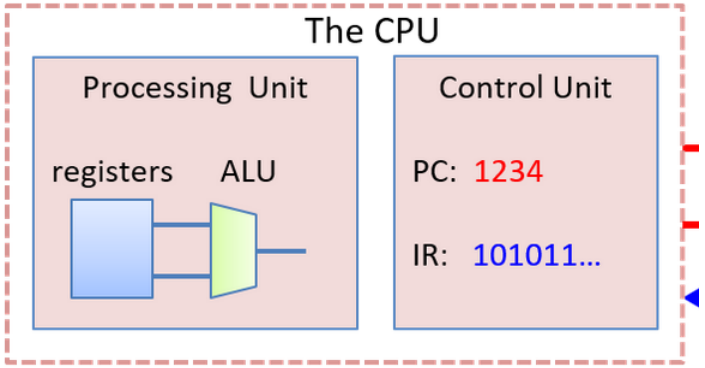
```
#include <stdio.h>

int adder2(int a) {
    return a + 2;
}

int main(){
    int x = 40;
    x = adder2(x);
    printf("x is: %d\n", x);
    return 0;
}
```

```
0000000000001149 <adder2>:
    1149: f3 0f 1e fa          endbr64
    114d: 55                   push   %rbp
    114e: 48 89 e5             mov    %rsp,%rbp
    1151: 89 7d fc             mov    %edi,-0x4(%rbp)
    1154: 8b 45 fc             mov    -0x4(%rbp),%eax
    1157: 83 c0 02             add    $0x2,%eax
    115a: 5d                   pop    %rbp
    115b: c3                   retq
```

Registers



Recall: A **register** holds data on the CPU

All registers have a name (start with **%**)

%rax, %eax – holds return value
%rdi, %rsi, %rdx, %rcx, %r8, %r9 – typically hold function parameters
%rsp – stack pointer
%rbp – base pointer, aka frame pointer
%rip – instruction pointer, aka program counter (read-only)

Registers

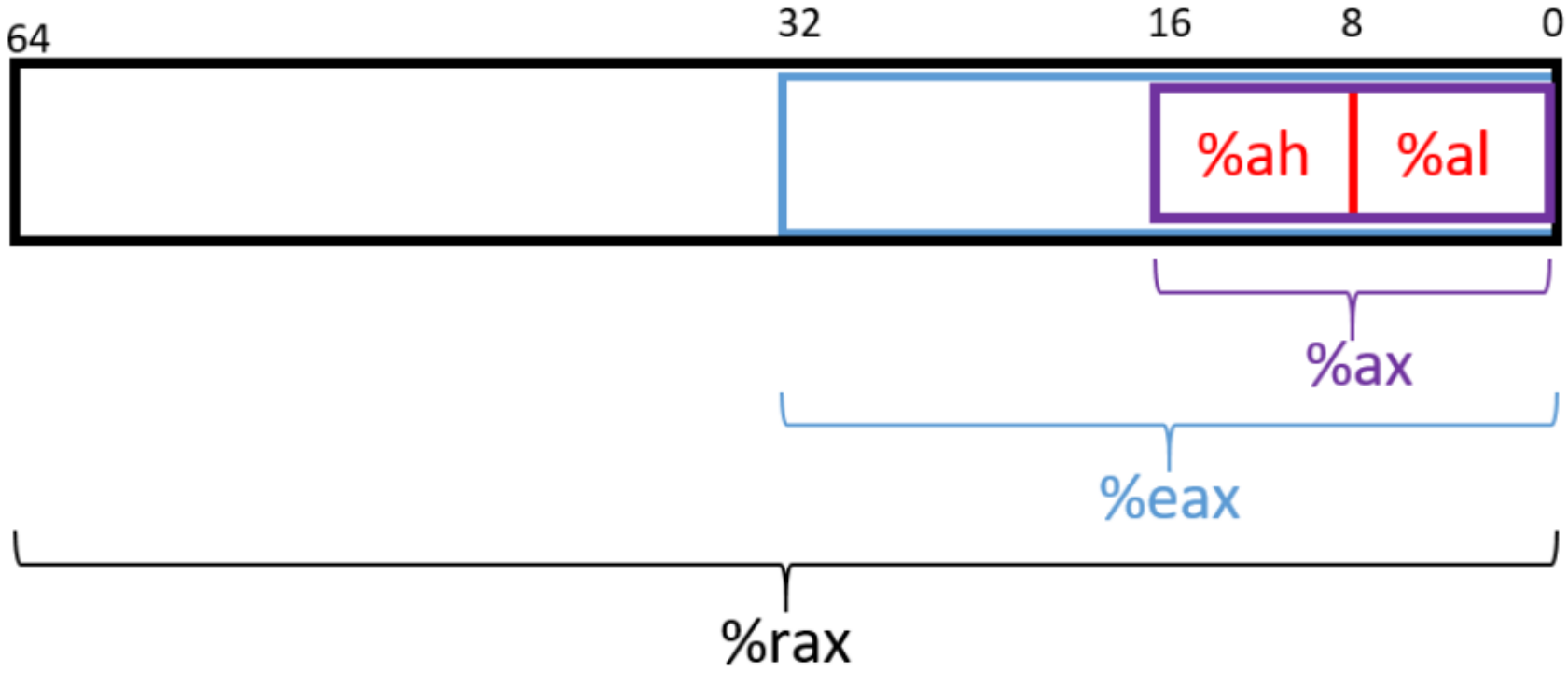


Figure 1. The names that refer to subsets of register %rax.

x86_64 registers and mechanisms for accessing lower bytes (from *Dive into Systems*)

64-bit Register	32-bit Register	Lower 16 Bits	Lower 8 Bits
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rdi	%edi	%di	%dil
%rsi	%esi	%si	%sil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Exercise: What are the capacities of these registers?

%edi

%rdi

%al

%r8w

Instruction suffixes determine sizes

Suffix	C Type	Size (bytes)
b	char	1
w	short	2
l	int or unsigned	4
s	float	4
q	long, unsigned long, all pointers	8
d	double	8

Ex: movl $13, -4(%rbp)

Registers

No data types in assembly!

But compiler will often choose registers based on data types

- ints will be stored in %eax or %esi because they are 32 bits
- longs will be stored in %rax or %rsi because they are 64 bits

Registers: Example

```
0000000000001149 <adder2>:
   1149: f3 0f 1e fa          endbr64
   114d: 55                   push   %rbp
   114e: 48 89 e5             mov    %rsp,%rbp
   1151: 89 7d fc             mov    %edi,-0x4(%rbp)
   1154: 8b 45 fc             mov    -0x4(%rbp),%eax
   1157: 83 c0 02             add    $0x2,%eax
   115a: 5d                   pop    %rbp
   115b: c3                   retq
```

Instruction structure

Instructions contain an **op code** and **operands**.

Ex**.** mov -0x4(%rbp), %eax

Ex. add $0x2,%eax

Types of operands

Constant (literal) values, for example, **$0x2**

Register forms reference to an individual register, for example, **%eax**

Memory forms refer to address lookups in RAM, for example,

-0x4(%rbp)

%rax contains 0x4
%rbp contains 0x800

Memory Forms

Syntax: <Offset>(Register1, Register2, <Scale>)
Scale must be 1,2,4,8

Meaning: M[Offset + Register1 + Register2 * Scale]

Examples:

- -0x4(%rbp)
- -0x4(,%rax,4)
- -0x4(%rbp, %rax, 8)
- 0x8(%rbp, %rax)
- (%rbp)
- 0x80c

Example: operands

Operand	Form	Translation	Value
%rcx			
(%rax)			
$0x808			
0x808			
0x8(%rax)			
(%rax, %rcx)			
0x4(%rax, %rcx)			
0x800(,%rdx,4)			

Memory

Address	Value
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

Registers

Register	Value
%rax	0x804
%rbx	0x10
%rcx	0x4
%rdx	0x1

Operand Restrictions

- Constant forms cannot serve as destination operands.
- Memory forms cannot serve as *both* the source and destination operand in a single instruction.
- In cases of scaling operations “-0x4(%rbp, %rax, 8)”, the scaling factor is a third parameter in the parentheses. Scaling factors can be one of 1, 2, 4, or 8.

Exercise: Which of these instructions are valid?

add %rax, (%rbx)

add (%rax), %rbx

add %rax, $8

add (%rax, %rbx, 10), %rcx

add (%rax), (%rbx)