HIGH-LEVEL PROGRAMMING I

Intro to C Programming (Part 1/3) by Prasanna Ghali

Outline

- What is a Computer Program?
- What is Computer Programming?
- What is a Programming Language?
- How Computers Work?
- How Computers Store Data?
- Data Representation in Machines and C
- Machine Languages
- Assembly Languages
- Disadvantages of Low-Level Languages
- High-Level Programming Languages
- Compilers and Interpreters

What is a Computer Program?

- Program is specific implementation of an algorithm in particular programming language

What is Computer Programming?

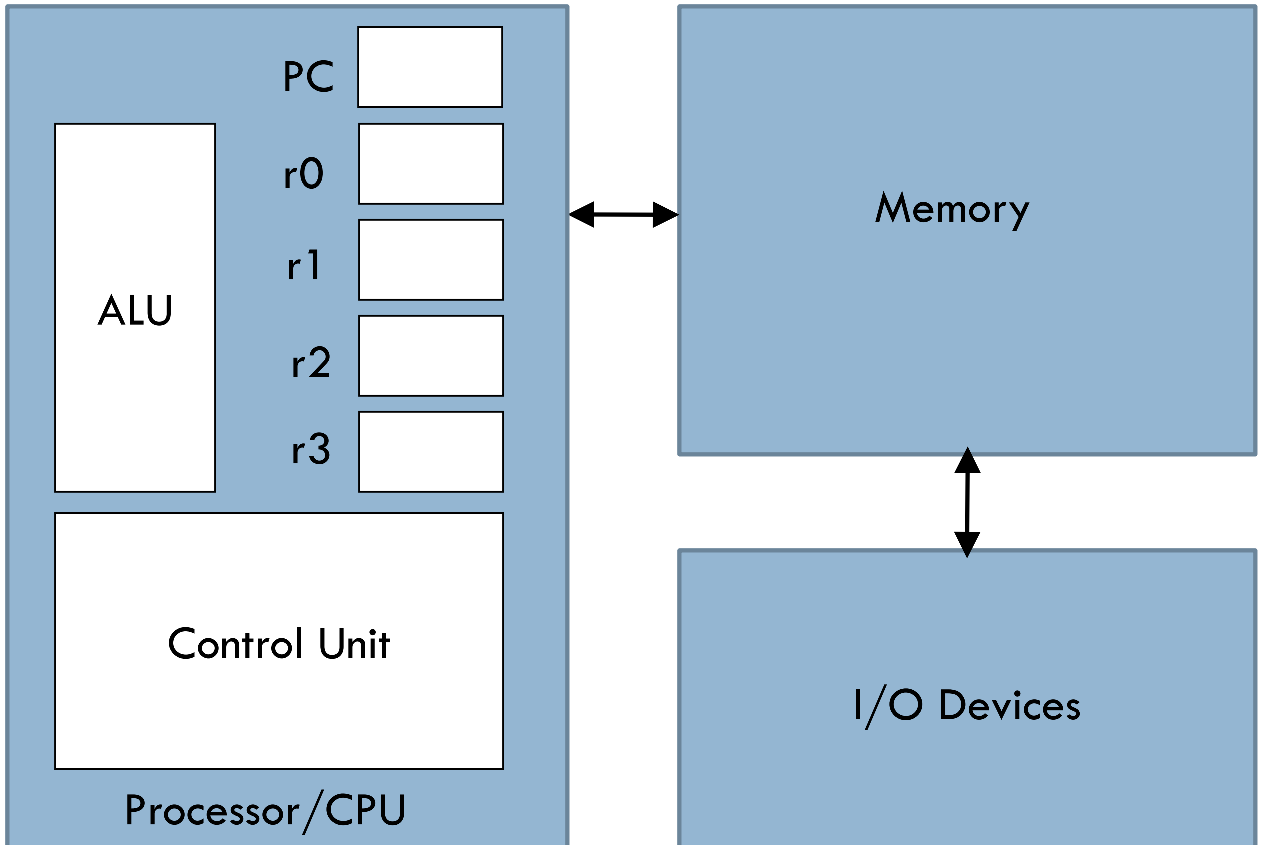
- **Science** and **art** of encoding algorithm into computer program

What is a Programming Language?

- Framework that allows programmers to precisely communicate their algorithms to computers
  - Syntax: What are rules of language?
    - Vocabulary: alphabet, words, sentences
    - Grammar: rules governing language constructs
    - Symbols representing syntax have 7-bit ASCII byte and UTF-8 encoding
  - Semantics: What is the meaning of sentences?

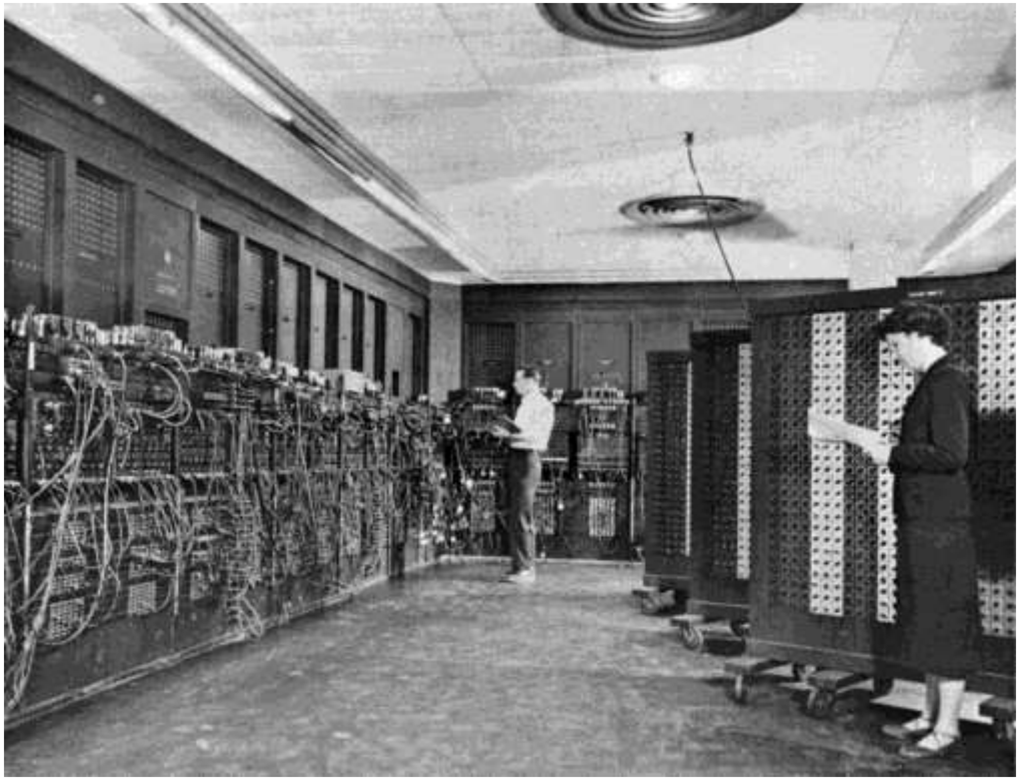
Organization of Computer

- Typical organization of modern computers based on von Neumann architecture described in 1945:



Stored-Program Computer (1/4)

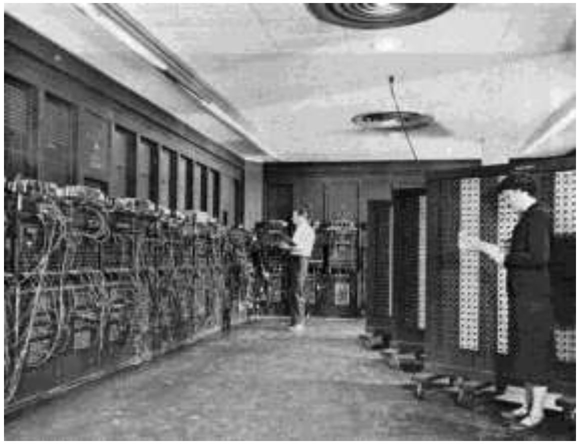
- Earliest machines were hardwired for specific applications



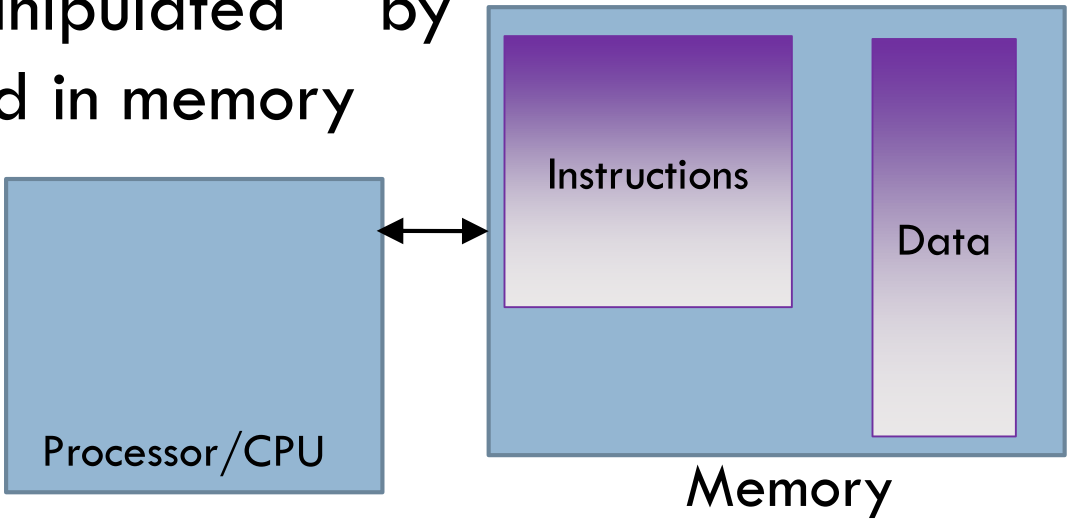
Reference

Stored-Program Computer (2/4)

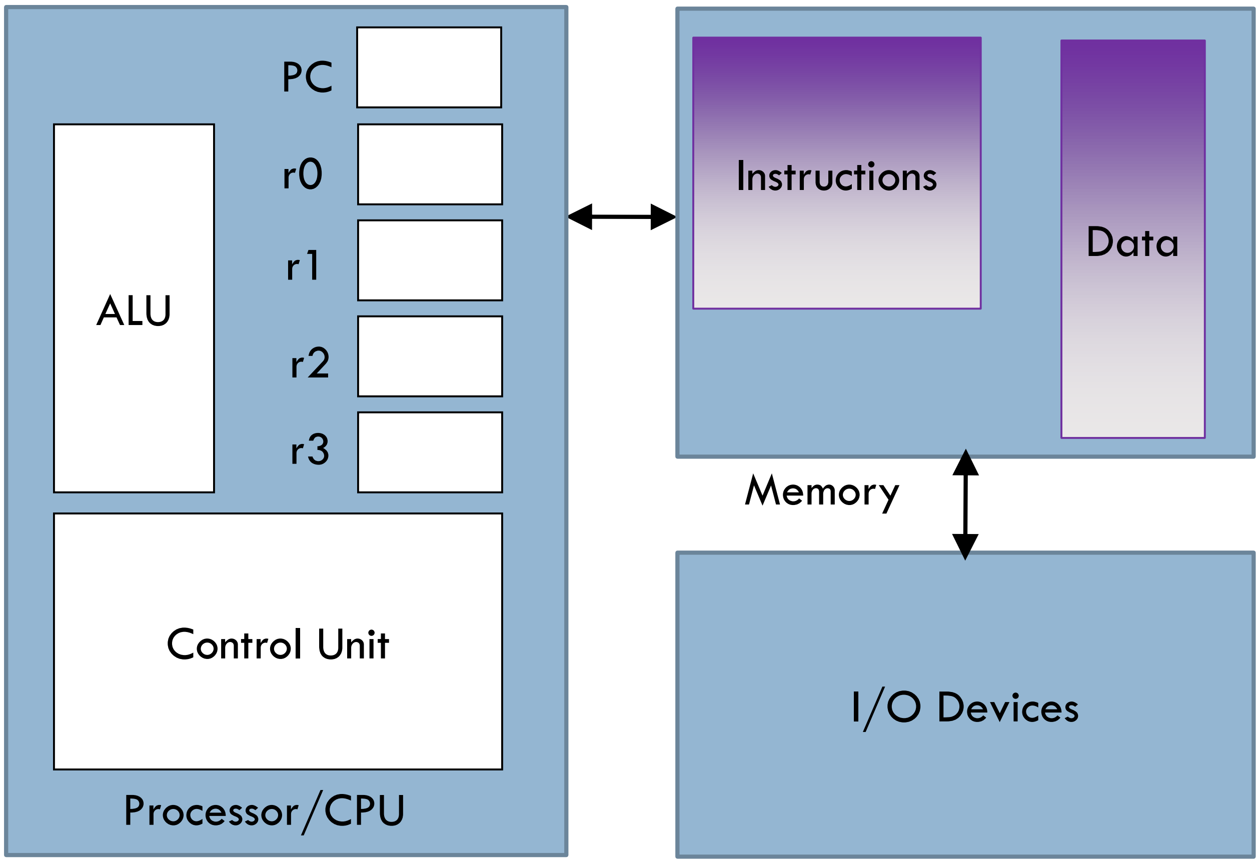
- Computers based on von Neumann architecture use *stored-program* concept:
  - Program that manipulates data is stored in memory
  - Data to be manipulated by program is also stored in memory



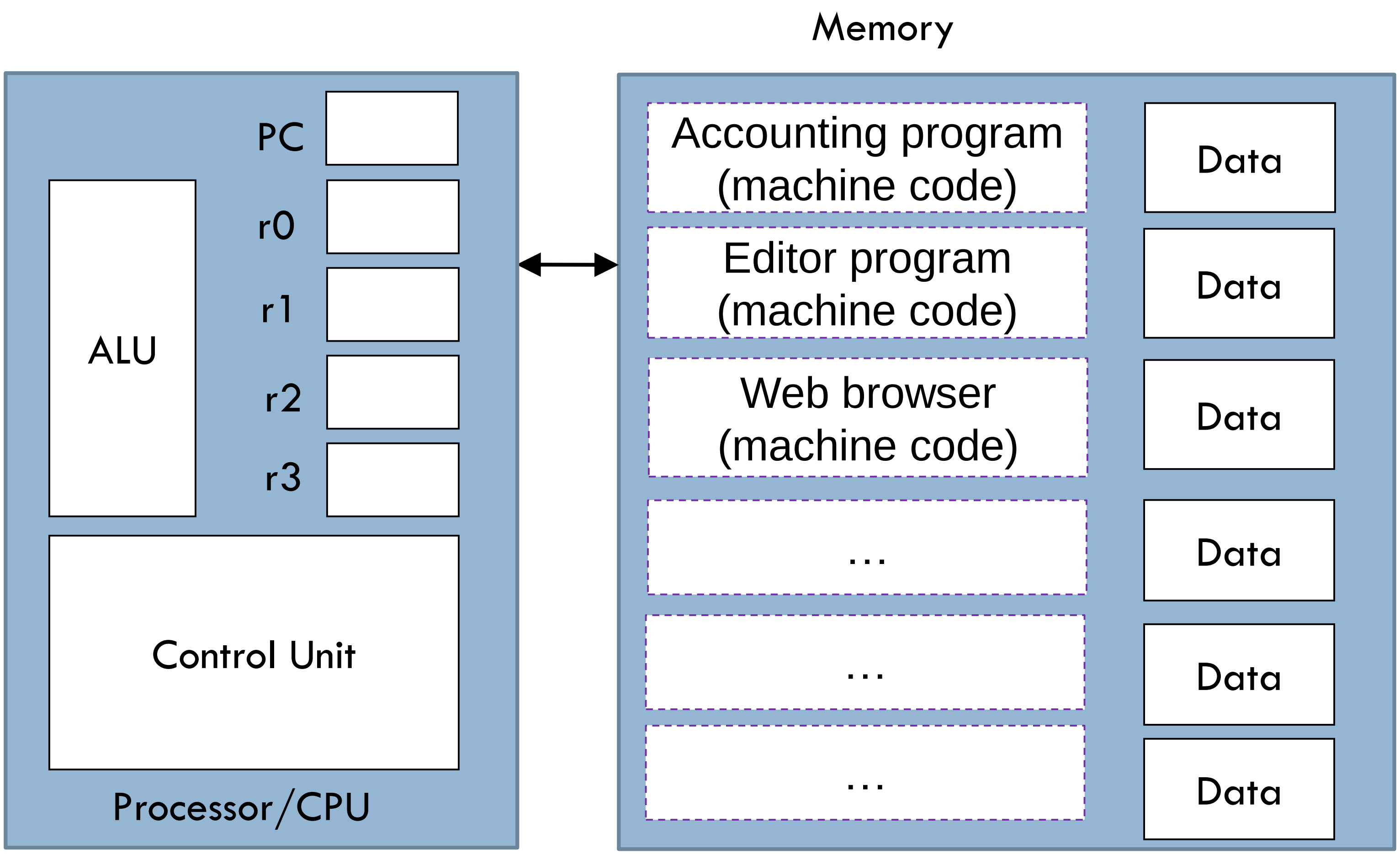
[Reference]



Stored-Program Computer (3/4)



Stored-Program Computer (4/4)

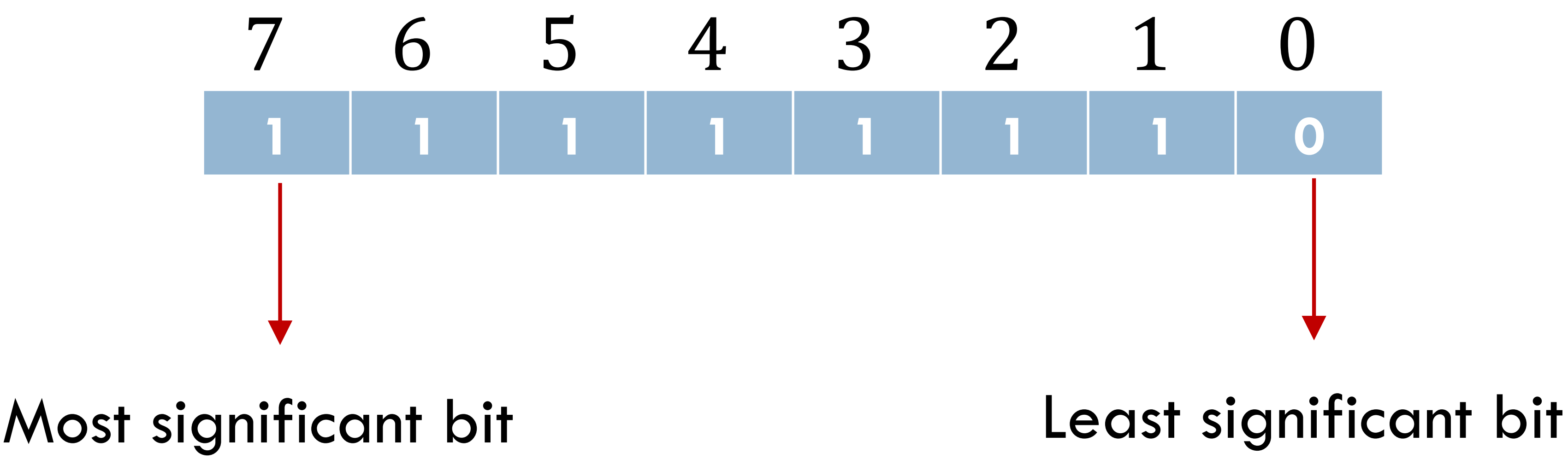


Bits

- Computers are digital electronic devices
- Digital devices represent information with sequences of 0s and 1s
  - Low voltage represents 0 while high voltage represents 1
  - Each of 0 or 1 digit is called ***bi**nary digi**t*** or *bit*
- This is fundamental concept - computers can only store and process information as strings of 0s and 1s

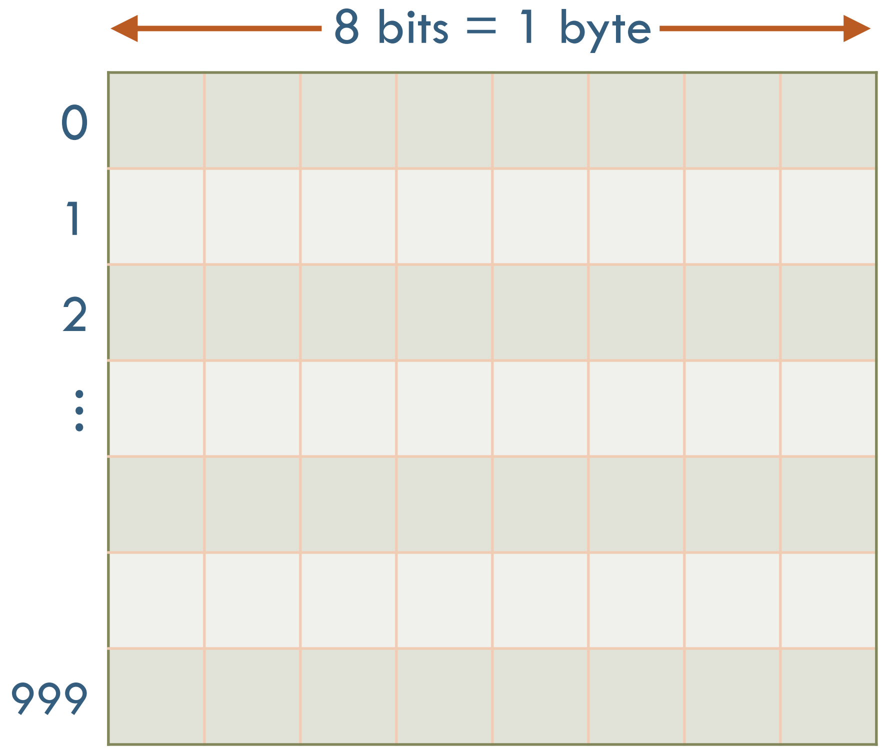
Bits and Bytes

- Language of computers is binary - sequence of 0s and 1s
- Sequence of 8 bits, called *byte*, has become de facto standard for unit of digital information



Computer Memory

- Picture illustrates organization of computer memory as linear array of 1000 bytes



Computer Memory Capacity

Name	Symbol	Size (Bytes)	
		Exponential	Explicit
Kilobyte	KB	2^{10} bytes	1024
Megabyte	MB	2^{20} bytes	1,048,576
Gigabyte	GB	2^{30} bytes	1,073,741,824
Terabyte	TB	2^{40} bytes	1,099,511,627,776

What is Data Type?

- *Data type* is a set of values and set of operations that can be applied on these values
  - Set of values indicates *kind* of data
  - Set of operations indicates *what* can be done with data

CPU Data Types

- Recall digital devices represent and process information using binary numbers which are sequences of 0s and 1s
- CPUs can only represent numbers of two data types:
  - Integer
  - Floating-point

Integer Data Types (1/2)

Type	What values?	Representation
Unsigned	Positive (incl. 0)	Traditional binary encoding
Signed	Negative and positive	2's complement

Integer Data Types (2/2)

- Because of way in which digital computing has evolved, modern CPUs use collection of bits grouped into units of eight:
  - 8 bits (byte)
  - 16 bits
  - 32 bits
  - 64 bits

Bit size	Unsigned	Signed
8-bit	$[0, 2^8 - 1]$	$[-2^7, 2^7 - 1]$
16-bit	$[0, 2^{16} - 1]$	$[-2^{15}, 2^{15} - 1]$
32-bit	$[0, 2^{32} - 1]$	$[-2^{31}, 2^{31} - 1]$
64-bit	$[0, 2^{64} - 1]$	$[-2^{63}, 2^{63} - 1]$

C/C++ Integer Data Types

- Data types for 64-bit C11 compiler used in this course
  - Note: Other compilers might show different behavior for 32- and 64-bit sizes

Bit size	**Unsigned**	**Signed**
8-bit	`unsigned char`	`signed char`
16-bit	`unsigned short int`	`signed short int`
32-bit	`unsigned int`	`signed int`
64-bit	`unsigned long int`	`signed long int`
64-bit	`unsigned long long int`	`signed long long int`

Floating-Point Types (1/2)

- Floating-point types useful for representing very small and very large numbers, but not precisely
- Floating-point values represented in IEEE 754 format
  - Rational numbers $\frac{p}{q}$ where p and q are integers are represented as $(-1)^s \times m \times 2^e$ where s is sign, m is fixed bit length fraction (*mantissa*), and e is exponent
  - Term *floating-point* refers to fact that these numbers can move binary point in rational number to adjust precision
  - *Precision* (how many fractional digits?) is used to distinguish floating-point values

Floating-Point Types (2/2)

- Floating-point representation is $(-1)^s \times m \times 2^e$
- SP means single-precision with 32-bits
- DP means double-precision with 64-bits
- EP means extended-precision with 128-bits

Type	Sign bit	Mantissa bits	Exponent bits
SP	1	23	8
DP	1	52	11
EP	1	112	15

Type	Smallest value	Largest value
SP	$\pm 1.175494351 \times 10^{-38}$	$\pm 3.40282346 \times 10^{38}$
DP	$\pm 2.2250738585072014 \times 10^{-308}$	$\pm 1.797693134862315 8 \times 10^{308}$

C Floating-Point Types

- C has corresponding equivalent types:

Bit size	C type
32-bits	`float`
64-bits	`double`
128-bits	`long double`

- Many languages (C, C++, Python) use `double` as their basic data type for representing rational numbers

Integer vs Floating-Point (1/2)

- General rule of thumb for programmers: prefer integer numbers; use floating-point numbers and arithmetic with caution
  - Integer types encode relatively *small range* of values, which are *exact*
  - Floating-point types encode *large range* of values, but only *approximately*
    - Cannot exactly represent many values such as 0.1, 0.2, … which are then either rounded up or down to nearest representable number
    - May not obey arithmetic rules because of rounding
    - Good precision for numbers around zero; precision decreases for larger numbers
    - Single-precision have precision of about 6 decimal digits
    - Double-precision have precision of about 15 decimal digits

Integer vs Floating-Point (2/2)

- Best advice from me to avoid pain and suffering:
  - For integer values, use 32-bit `signed int` type
    - Because of rules used by programming languages when `signed int` and `unsigned int` values are mixed, results can be unexpected
    - To avoid surprises, stick to `signed int` even if you expect numbers to be only positive
  - For fractional values, use 64-bit double-precision `double` type

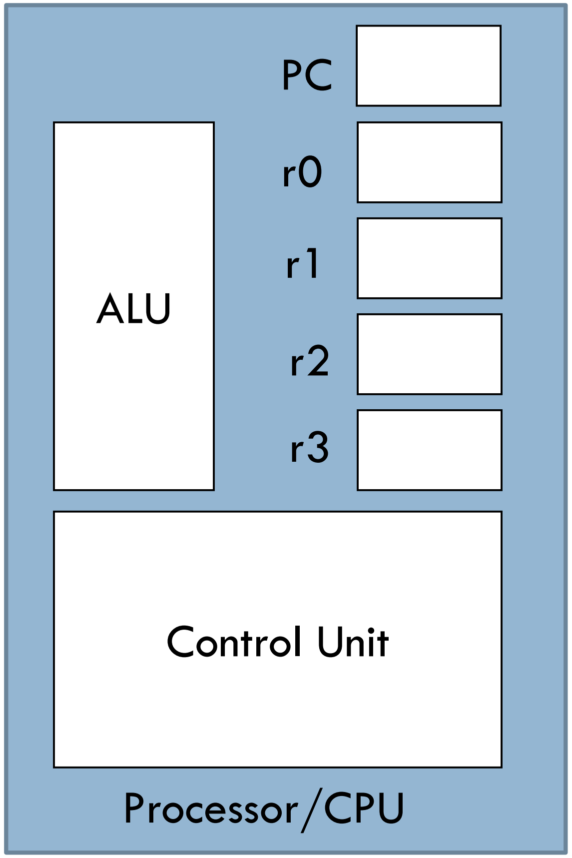
Programming Languages: Classification

- Programming languages can be classified in many different ways
- We'll broadly classify in two ways: low-level languages and high-level languages

Reference

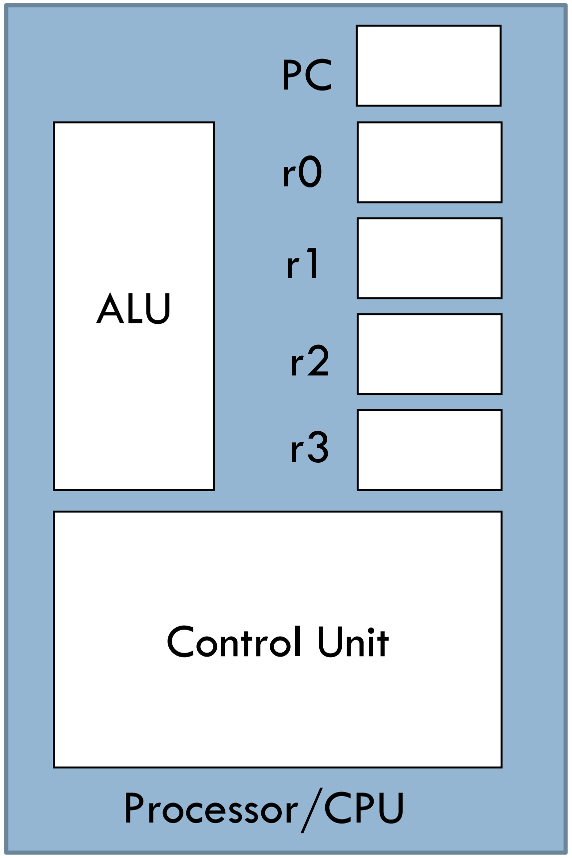
Instruction Set Architecture (1/2)

- CPU hardwired by computer architect with set of basic instructions called *Instruction Set*
  - Instruction is represented as sequence of 0s and 1s
  - Instruction referred to as ***operation code*** or ***opcode***
  - Things or values that instruction works on are called ***operands***



Instruction Set Architecture (2/2)

- Machine instructions generally fall into 3 categories:
  - Data movement: *Load*, *Store*, *Move*
  - Control flow: *Branch*, *Jump*, *Goto*
  - Arithmetic and Logic: *Add*, *Sub*, *Mul*, *Div*, *And*, *Or*, …
- Computer architects implement digital circuitry in:
  - Control Unit to interpret these instructions
  - ALU to execute these instructions



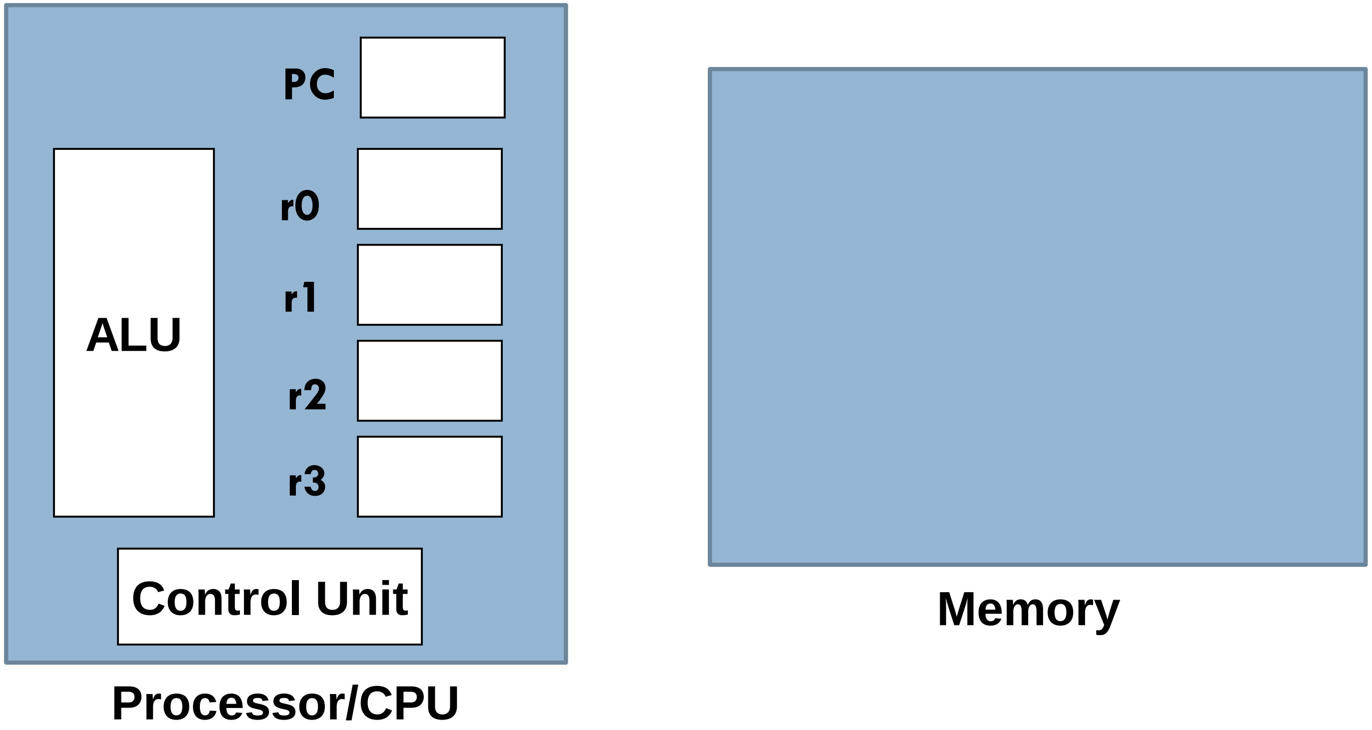
Fetch-Decode-Execute Cycle (1/12)

- Instruction cycle (also known as Fetch-Decode-Execute cycle) is basic operational process of CPU

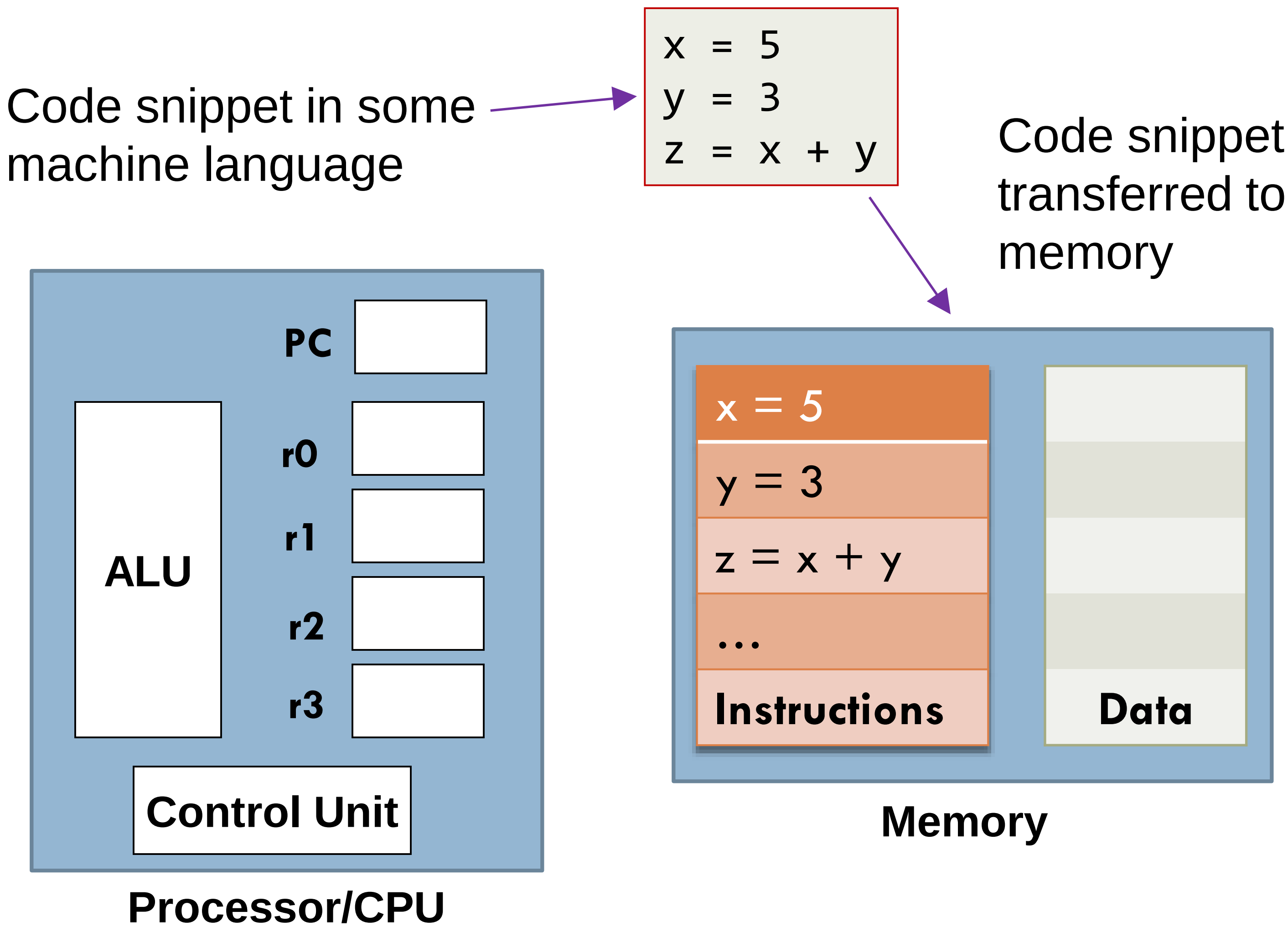
Fetch-Decode-Execute Cycle (2/12)

Code snippet in some machine language

```
x = 5
y = 3
z = x + y
```

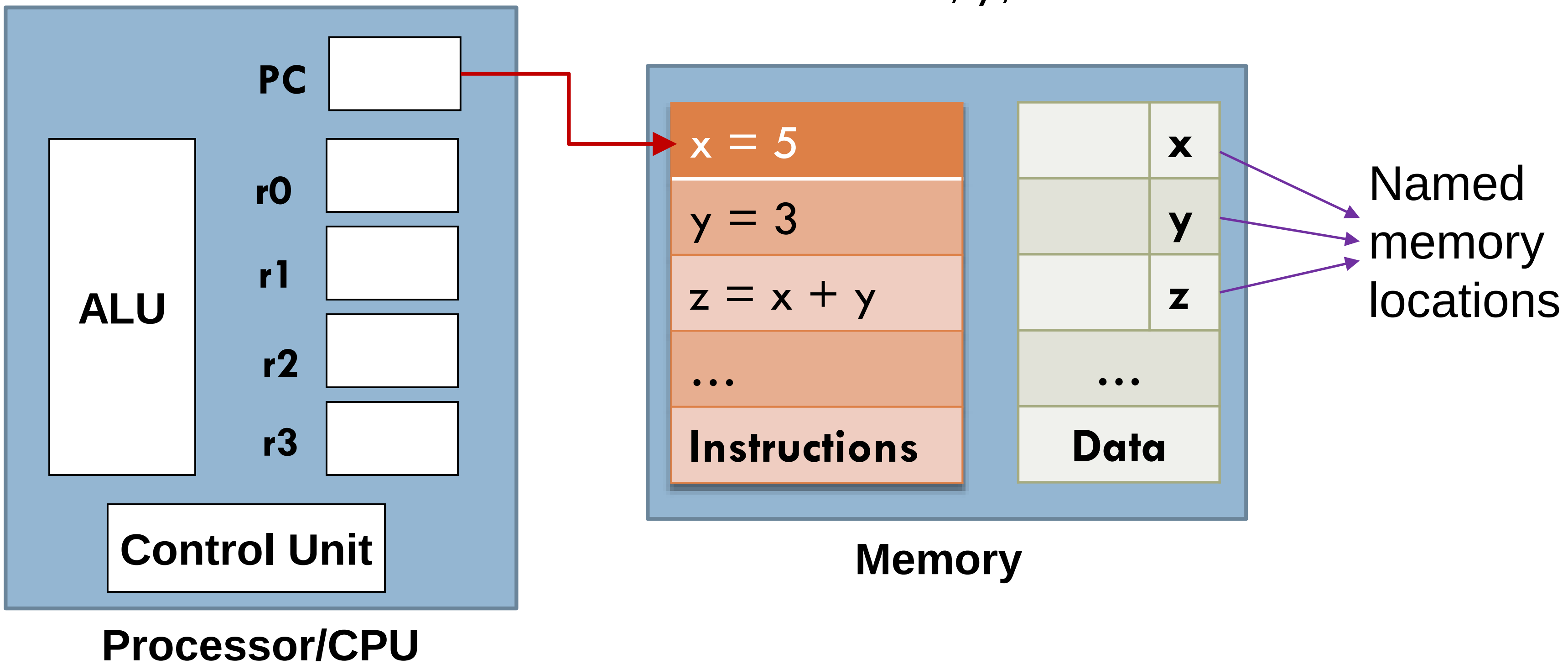


Fetch-Decode-Execute Cycle (3/12)

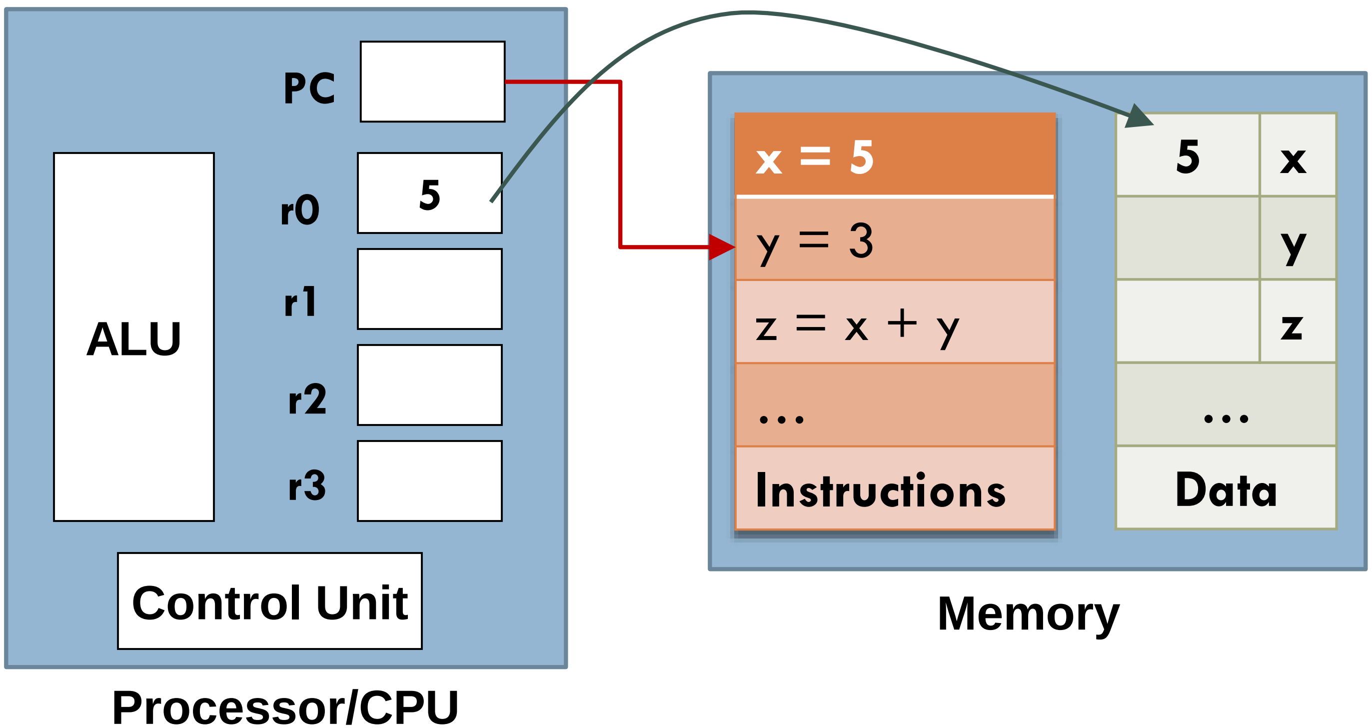


Fetch-Decode-Execute Cycle (4/12)

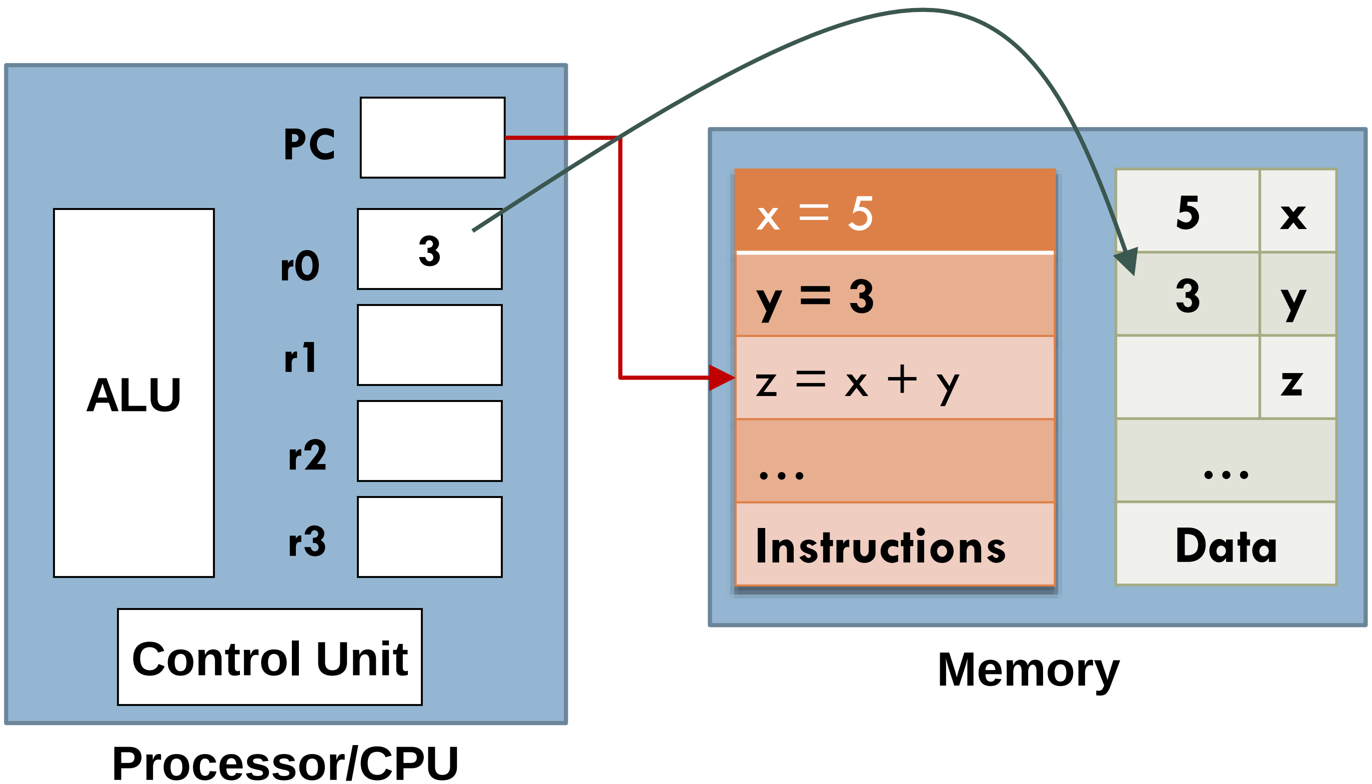
1. Code begins execution
2. Memory storage assigned for variables `x`, `y`, and `z`



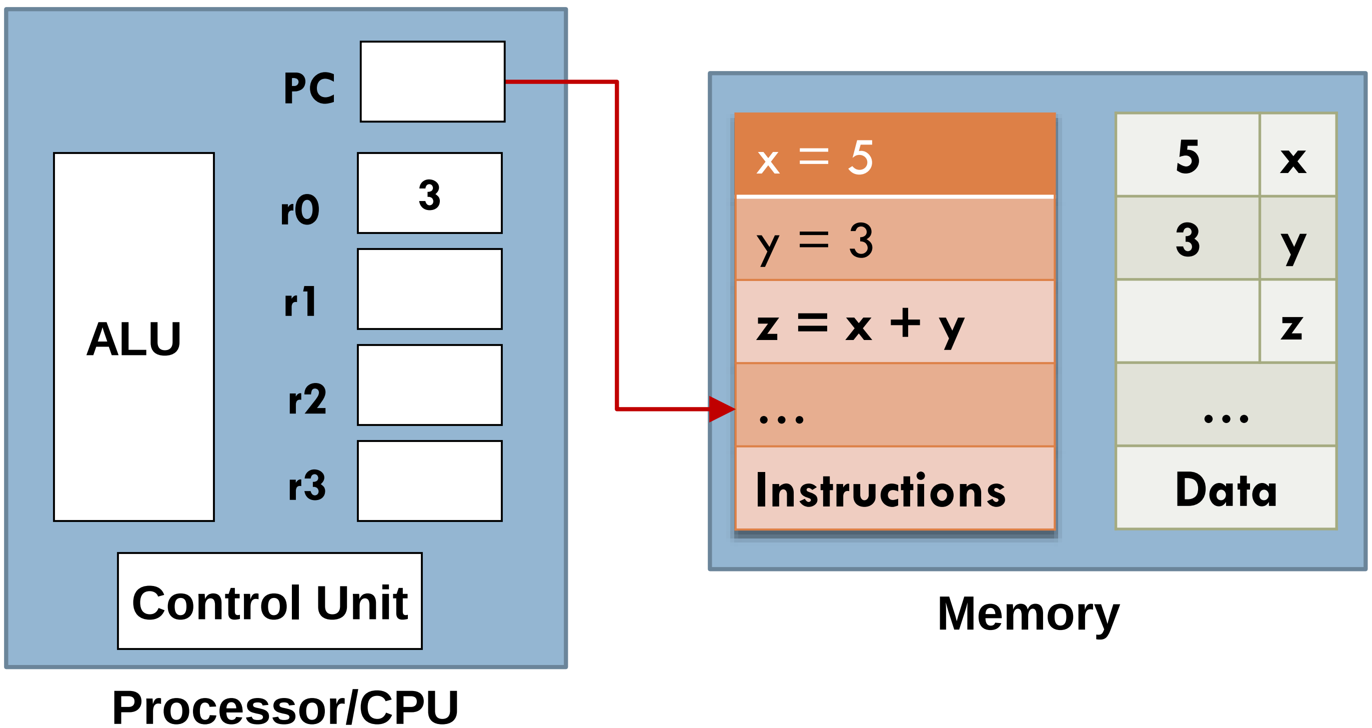
Fetch-Decode-Execute Cycle (5/12)



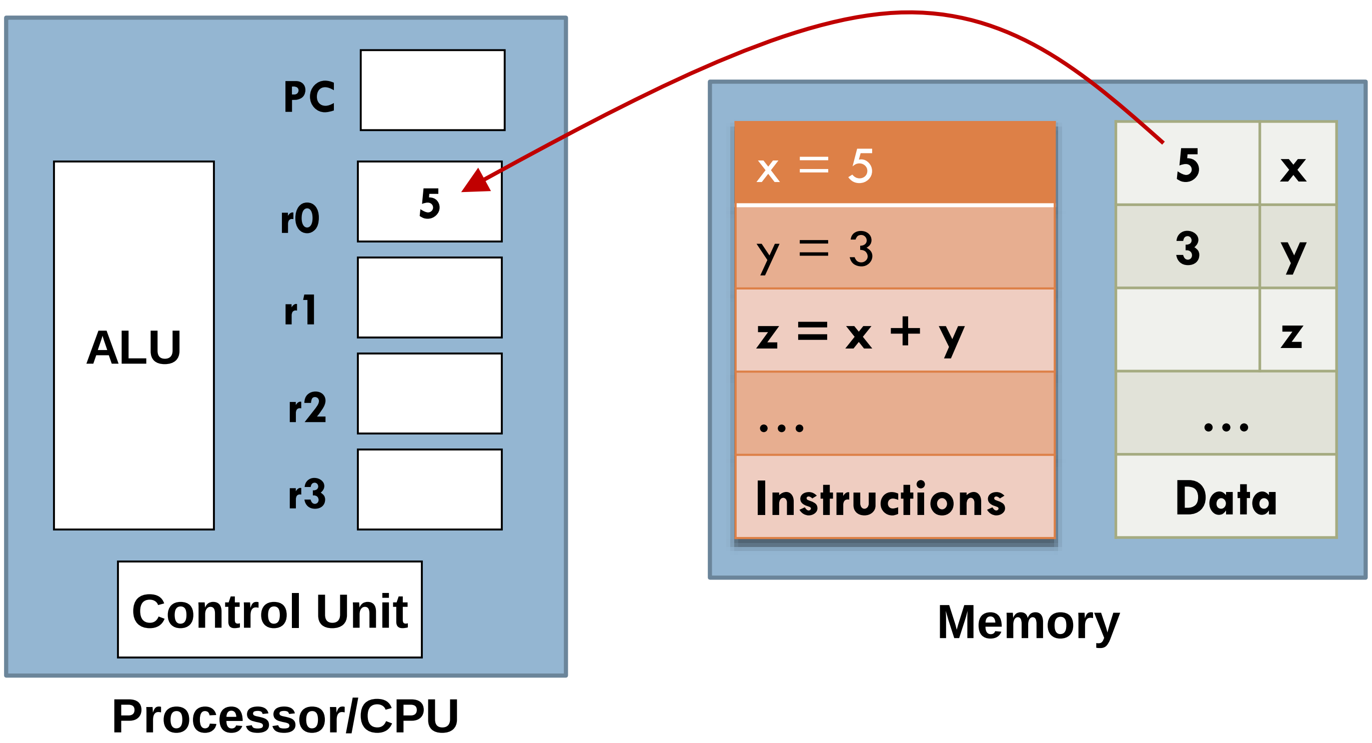
Fetch-Decode-Execute Cycle (6/12)



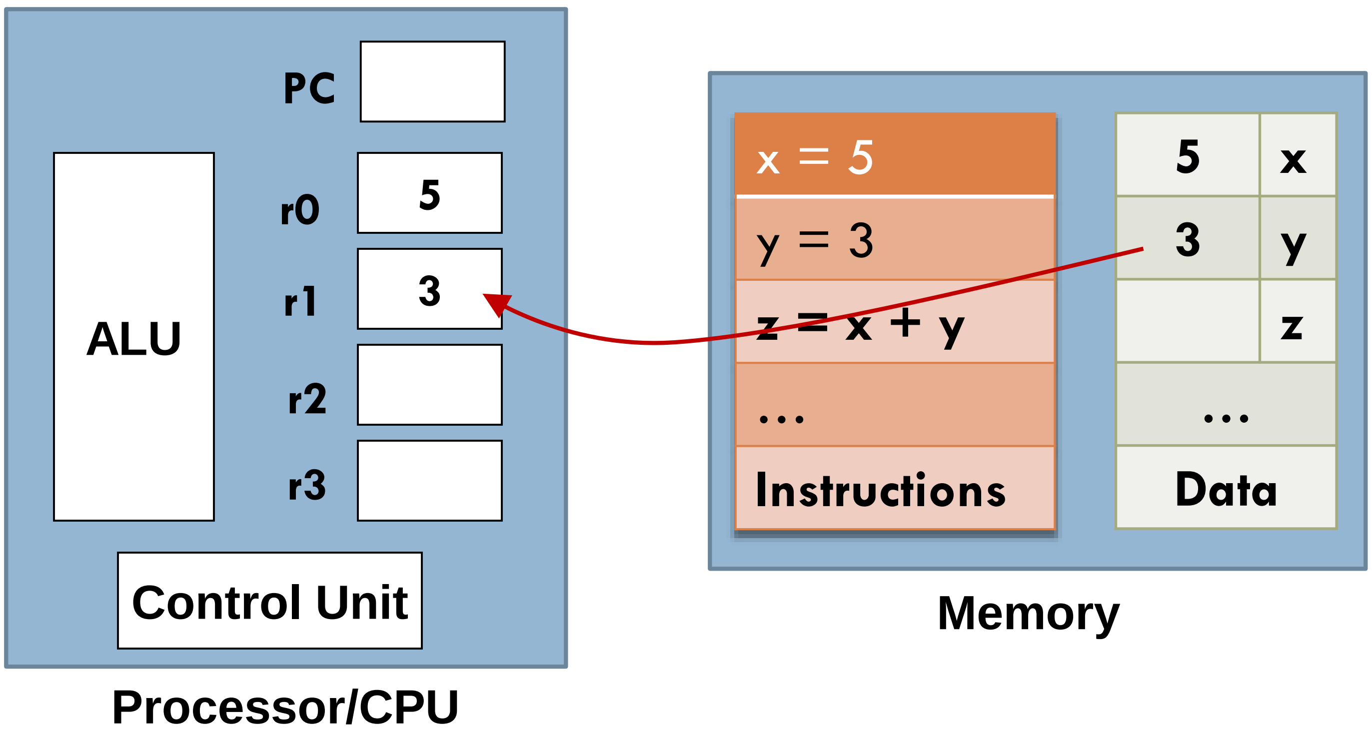
Fetch-Decode-Execute Cycle (7/12)



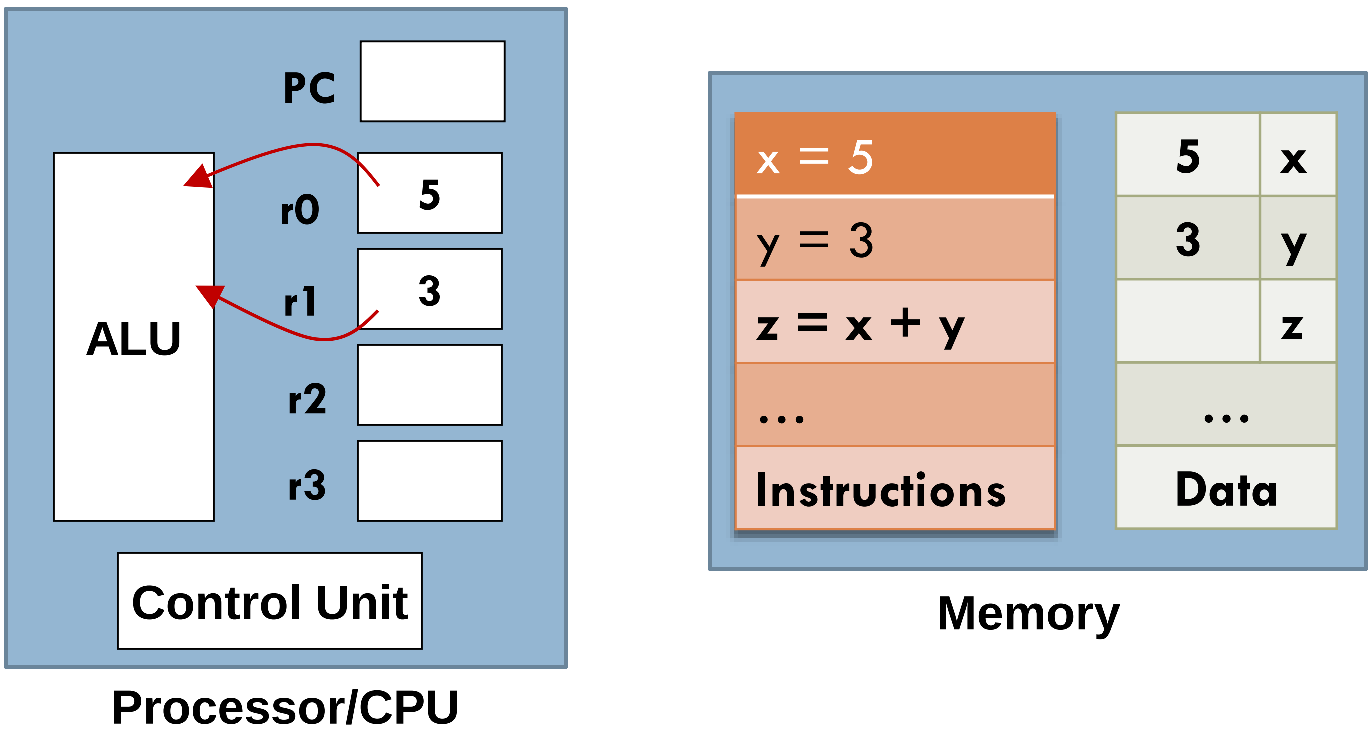
Fetch-Decode-Execute Cycle (8/12)



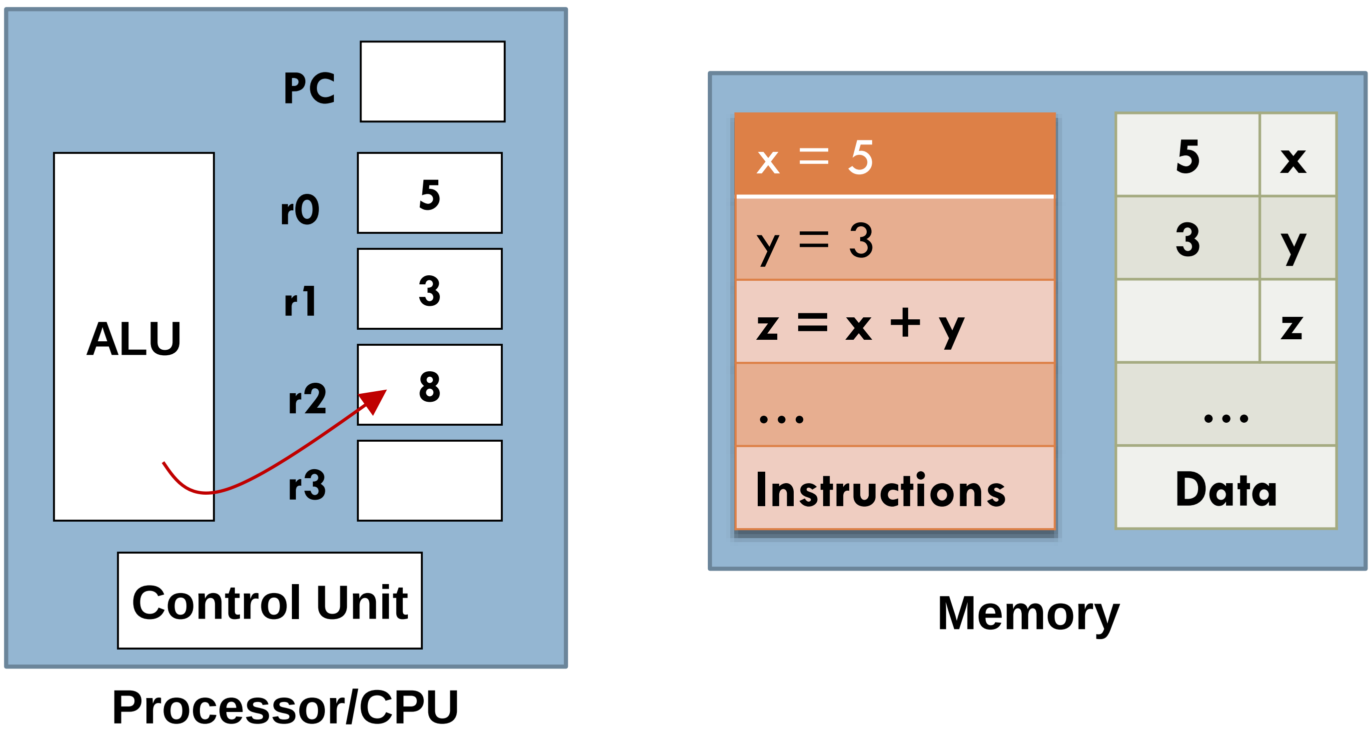
Fetch-Decode-Execute Cycle (9/12)



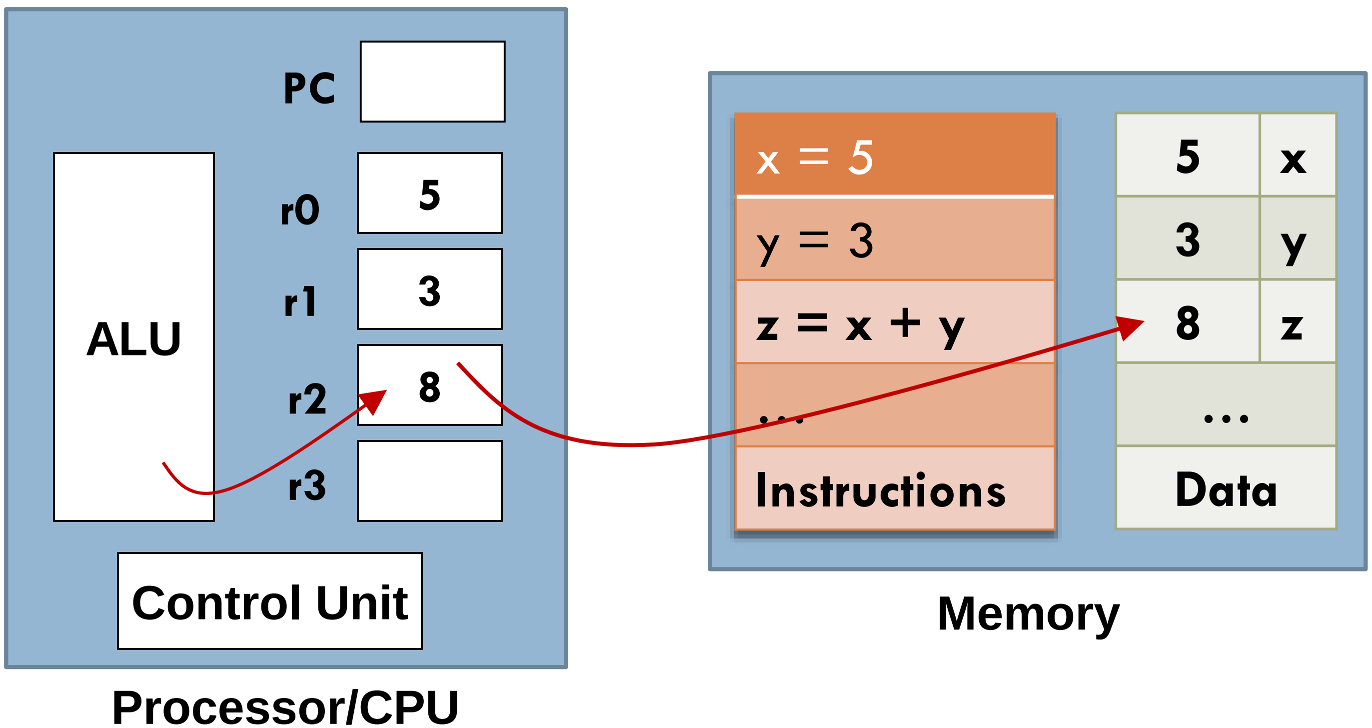
Fetch-Decode-Execute Cycle (10/12)



Fetch-Decode-Execute Cycle (11/12)



Fetch-Decode-Execute Cycle (12/12)



Machine Languages

- Recall that computer architects provide CPUs with set of basic machine instructions represented in numbers called *Instruction Set*
- Instruction set of CPU with additional tools called *Machine Language*
  - Unique to each CPU family (x86, PowerPC, ARM, …)
  - Fixed-width patterns of 1's and 0's correspond to *opcodes* and *operands*

Machine Language: Example

- Euclid's GCD algorithm:
  - To compute greatest common divisor of integers *a* and *b*, check to see if *a* and *b* are equal. If so, print one of them and stop. Otherwise, replace the larger one by their difference and repeat.
- Machine code (Intel x86) for Euclid's GCD algorithm looks like this:

```
55 89 e5 53 83 ec 04 83 e4 f0 e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00 00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00 8b 5d fc c9 c3 29 d8 eb eb 90
```

Machine Languages: Disadvantage

- Earliest computers could only be programmed in their machine languages
  - Programming was tedious, cumbersome, and error-prone process

Assembly Language (1/2)

- BIG IDEA: Abstraction
  - When something is hard, use abstraction to hide complexity
  - Build hierarchical layers with each lower layer hiding details from the layer above
- Use simpler *intermediate* language to provide *abstraction* between low-level languages and programmer

Assembly Language (2/2)

- *Assembly language* is first intermediate language to be invented
  - Provides English-like mnemonics to replace binary numbers in machine language programs
  - Example: machine instruction 01110011 00001001 00000011 specified as ADD r9, r3
- Translator called *assembler* now required to convert assembly code into machine code

Assembly Language: Example

- Assembly code (Intel x86) for Euclid's GCD:

```
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    andl $-16, %esp
    call getint
    movl %eax, %ebx
    call getint
    cmpl %eax, %ebx
    je C
```

```
A:  cmpl %eax, %ebx
    jle D
    subl %eax, %ebx
B:  cmpl %eax, %ebx
    jne A
C:  movl %ebx, (%esp)
    call putint
    movl -4(%ebp), %ebx
    leave
    ret
D:  subl %ebx, %eax
    jmp B
```

Low-Level Languages: Disadvantages (1/2)

- Programming in low-level languages is machine-centered enterprise
  - Instructions can only be specified at machine level
  - For example, mathematicians cannot express solutions to numerical problems using mathematical functions such as $a \times \sin(2 \times \pi + b)/c$
  - Difficult to support data types not native to machine

Low-Level Languages: Disadvantages (2/2)

- Low-level programs are not portable
  - Each CPU family has to be programmed in its own machine or assembly language
  - Expensive and error-prone as CPUs evolve and competing designs are developed

Evolution of Computing (1/2)

- Complexity of CPUs continues to grow at quantum leap
  - Difficult for humans to keep track of wealth of details
- Computers progressively being used to solve problems of increasing complexity
  - Advanced algorithms, more complex data structures become difficult to implement in low-level languages

Evolution of Computing (2/2)

- Programmers began to wish for machine independent languages
- Idea of *abstraction* again came to the rescue
  - Why not create high-level languages to abstract away low-level machine details?
  - Programmers need not work directly with nor worry about registers, memory addresses, …

Trend Towards High-Level Languages

- Thousands of high-level languages invented since 1950s such as COBOL, FORTRAN, ALGOL, Forth, Ada, C, C++, Java, …

High-Level Languages (1/2)

- Term *high* in high-level language means
  - *closer* to way humans think
  - *closer* to problems being solved
- Uses English-like mnemonics for groups of actions and data
  - `a = sqrt(b);`
  - `if (ammo > 0)`
    `fire_weapon();`

High-Level Languages (2/2)

- Easier to tackle complex problems
  - Programmers can spend more time on high-level concepts such as algorithm design
  - Data can be expressed in hierarchy of data types derived from built-in machine data types
- Programs are portable
  - Program written in high-level language can be translated for different machines
- Programming becomes more accessible

C/C++ and Python: Examples

- Euclid's algorithm for GCD: To compute the greatest common divisor of integers *a* and *b*, check to see if *a* and *b* are equal. If so, print one of them and stop. Otherwise, replace the larger one by their difference and repeat.

```c
// C/C++ code for GCD
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }
  return a;
}
```

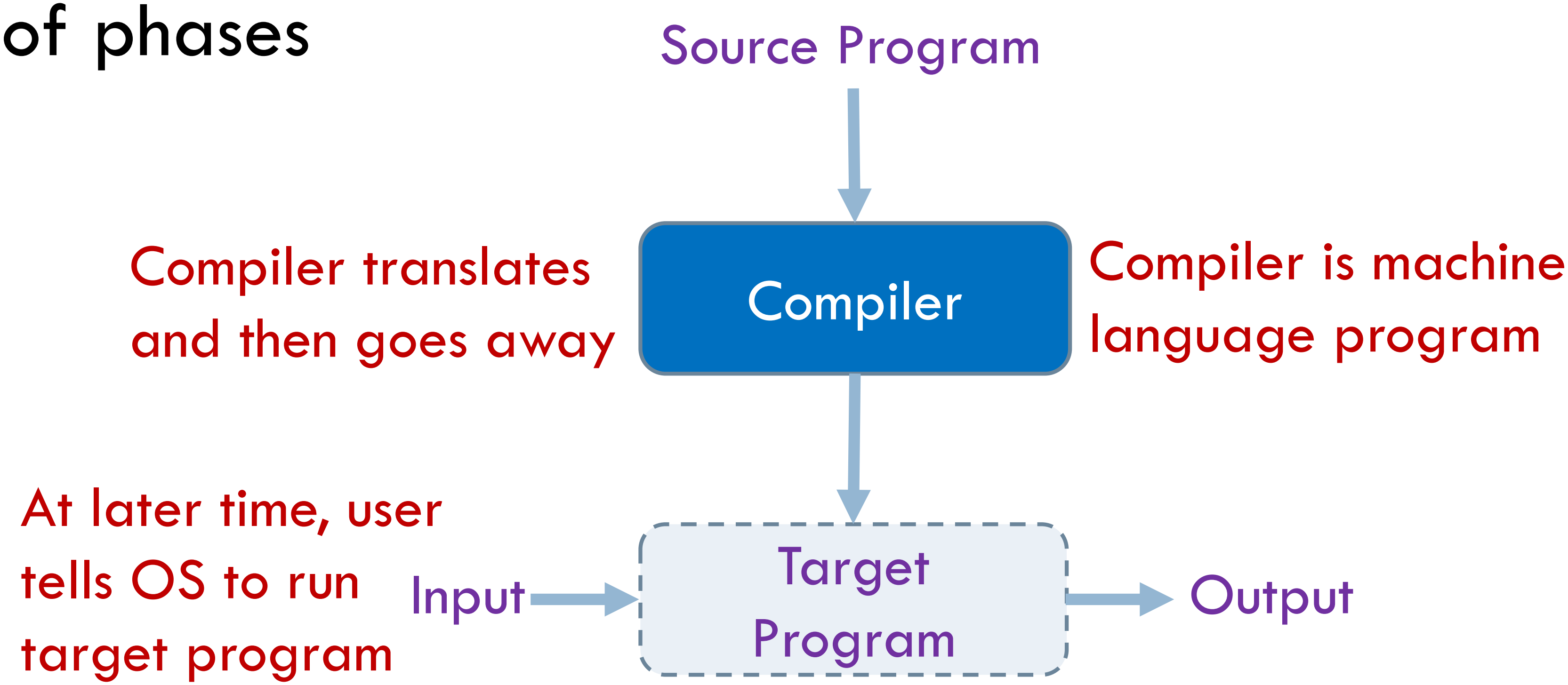
```python
# Python code for GCD
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Translators

- *Compilers* and *interpreters* are translators to convert programs written in high-level language into machine language

Compilers: C, C++, …

- *Compiler* translates *source program* (written in high-level language) into *target program* (usually in machine language) using a number of phases



Interpreters: Python, JavaScript

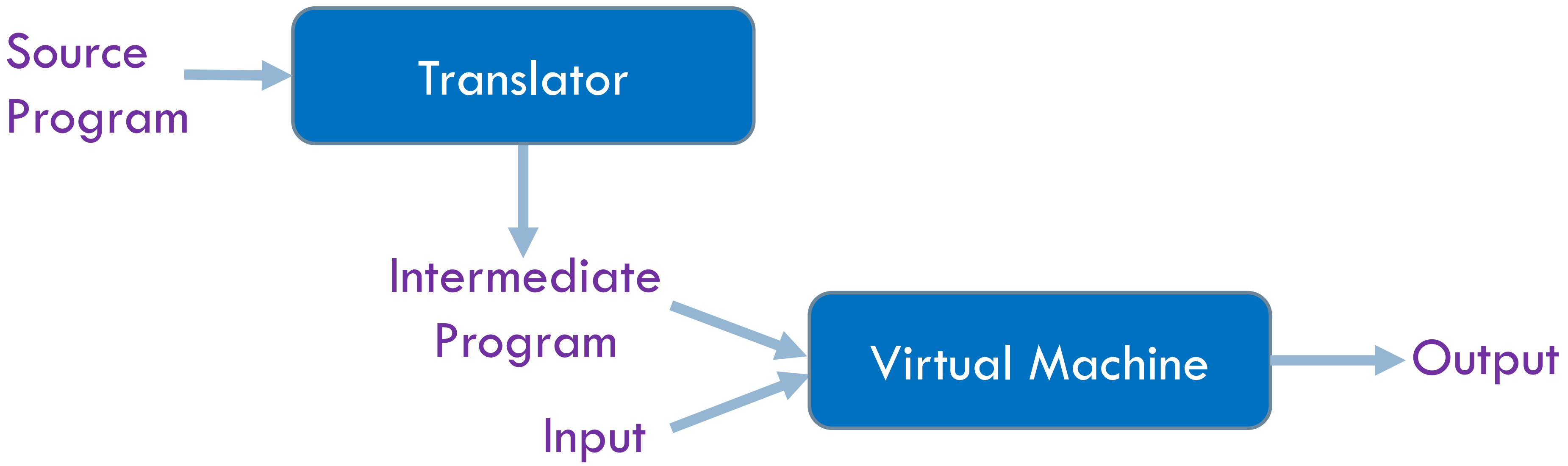
- Interpreter provides *virtual machine* that:
  - Reads one high-level statement at a time
  - Converts statement into machine language instructions
  - Has CPU execute these instructions

Compilers vs. Interpreters (1/2)

- In general, interpretation leads to
  - Greater flexibility – program can generate new pieces of itself and execute them on the fly
  - Better diagnostics – because interpreter is executing source code directly, it can provide debug information
- Compilation, by contrast, leads to better performance
  - In general, a decision made at compile time is a decision that doesn’t need to be made at run time

Compilers vs. Interpreters (2/2)

- While conceptual differences are clear, most language implementations include mixture of both
- Java, C#



Summary (1/2)

- Typical organization of modern computers based on von Neumann architecture of stored-program concept
- Language of computers is binary - sequence of 0s and 1s
- Modern CPUs use collection of bits grouped into units of eight
  - Sequence of 8 bits, called *byte*, has become de facto standard for unit of digital information
  - 16-bits, 32-bits, 64-bits
- *Data type* is a set of values and set of operations that can be applied on these values
- CPUs can only represent numbers which are either integer or floating-point data types
  - Integer: signed and unsigned 8-, 16-, 32-, and 64-bits
  - Floating-point: single-, double-, and extended-precision
- In this course, use `signed int` and `double` as standard data types for integer and floating-point data types

Summary (2/2)

- Programming languages broadly classified as low-level and high-level languages
  - Machine languages use Instruction Set Architecture that consists of instructions hardwired into CPU
  - Instruction consists of opcode and operands
  - Instructions executed using Fetch-Decode-Cycle
  - Assembly language use mnemonics; assembler is required to convert assembly code to machine lnaguage
  - Using low-level languages is tedious, cumbersome, error-prone, and beyond capability of humans
  - High-level languages provide portability and higher levels of abstraction from underlying machine
    - Syntax and semantics
    - Compilers (C, C++) and interpreters (Python, JavaScript) are translators of high-level language code into machine language
    - Some languages (Java, C#) use combination of compilation and interpretation