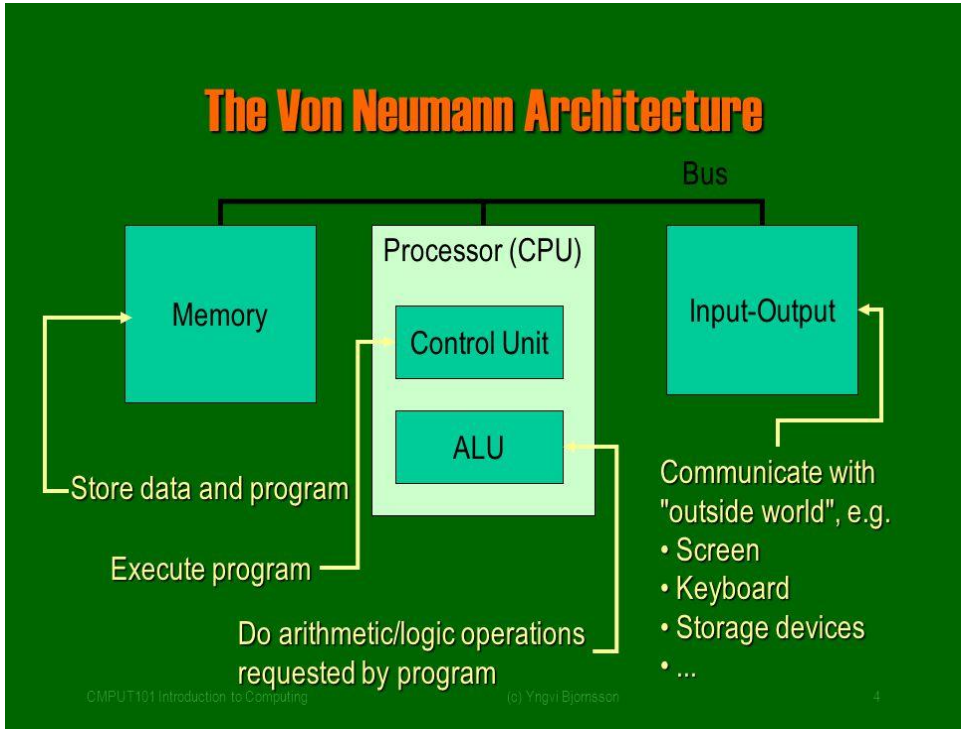
The Von Neumann Machine:
A model for computer's architecture

Low vs. High Level programming languages

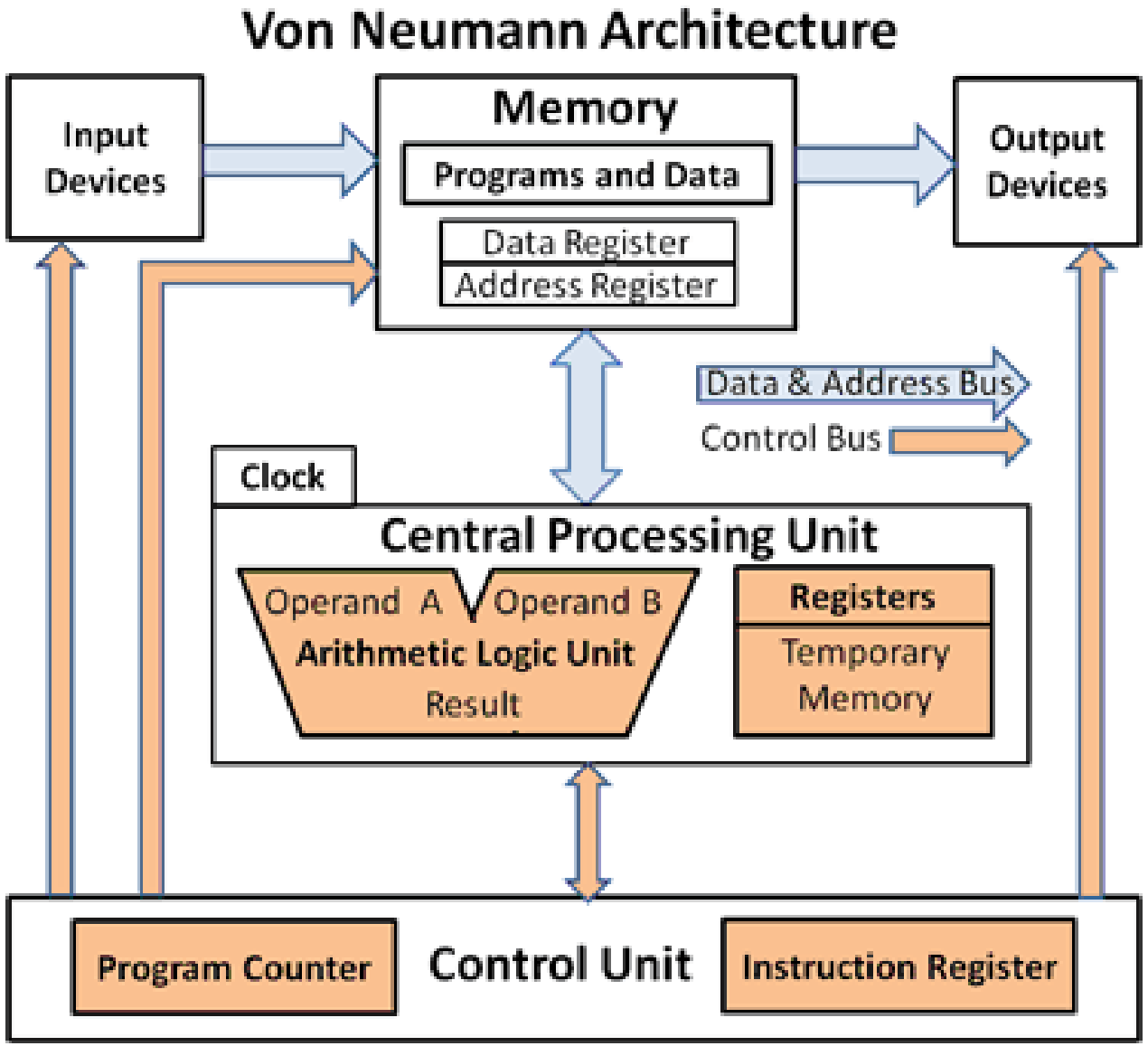
An introduction to C

The basic building blocks of a stored program digital computer, their interconnection, and behavior were described (after the 1945) by mathematician John von Neumann in a model that subsequently became known as "the von Neumann architecture" and is the basis of most general purpose digital computers today.

The von Neumann's architecture has four basic components.



A refined view



The Central Processing Unit (CPU) performs the computer's arithmetic and logical operations. It includes:

The Arithmetic and Logic Unit (ALU)
typically contains functional units which carry out basic mathematical operations such as add, subtract, multiply, divide, increment, compare as well as logical AND, NOT and OR.

Registers
provide temporary (fast access time) storage for operands and intermediate results

The Control Unit
manages the flow and timing of data and instructions through the computer as well as the operations performed by the CPU. It interprets the program's instructions and initiates their execution by the ALU. It contains:

- The Program Counter (PC), which contains the address of the next instruction to be executed.
- The Instruction Register (IR), which reads and stores the current instruction from memory.

The Clock
The execution of instructions is driven by a periodic clock signal ensuring that all parts of the system work in unison. Some instructions execute in one clock cycle. Others may take several cycles. The faster the clock rate, the faster the computer will run.

The Memory (also called **main memory**, or **Random Access Memory** – RAM) contains both instructions and data.

Memory words
memory is organized into fixed-size words, each one with an associated physical address (an integer number). Typically a word may consist of 4, 8, 16, 32, 64 or more bits which are treated as **a single unit** by the computer hardware. The word length is usually long enough to contain both instructions and data as well as addresses.

Modern architectures use 32 or 64 bits memory words.

Memory and storage registers are usually designed to store complete words.

- The Memory Address Register (MAR) contains the address of the memory location currently in use.
- The Memory Data Register (MDR) stores the data currently being transferred to and from memory.

A "load" instruction reads a value from the memory address held in the MAR and stores it in the MDR which acts as a buffer and makes it accessible to the ALU.

A "store" instruction writes the value contained in the MDR to the memory location contained in the MAR to store it in memory.

Input and Output (I/O) Devices

I/O devices such as keyboards, pointers, card readers, displays, and printers are allocated specific memory locations for moving data in and out of the computer memory.

The Communications Bus

is a set of parallel electrical tracks interconnecting the components within the computer. The von Neumann architecture combines signals from three separate buses, the control bus, the address bus, and the data bus (which carries both data and instructions) into a single systems bus.

The von Neumann architecture has only one bus which is used for both data transfers and instruction fetches, that cannot be performed simultaneously.

Modern architectures employ a different, much more efficient, bus system.

The Fetch-Execution Cycle

This is the basic computer operating cycle.

The control unit fetches an instruction from the memory and decodes it producing an operation code (op-code).

The op-code is passed to the ALU which receives the data from the memory, executes the instruction, and stores the result in memory.

The following describes the sequence in more detail.

Fetch Sequence

1. The program counter (PC), a CPU register, holds the address of the next instruction to be implemented.
2. The address held in the PC is first transferred to the memory address register (MAR) which acts as a buffer holding it until it is ready to access the main memory via the address bus.
3. The content of the PC is incremented by 1 to indicate where the next instruction is located in memory (**sequential computational model**).
4. The instruction (op-code + operands) contained in the main memory location specified by the MAR are transferred along the data bus to the memory data register (MDR) which also acts as a buffer.
5. The contents of MDR are then transferred to the Instruction Register (IR), a CPU register.
6. The IR separates the instruction into its op-code (add, load, store etc) and its operands (the data on which it operates).

Execute Sequence

1. The IR sends its op-code through an instruction decoder to generate its digital instruction code.
2. The digital instruction, together with its operands, are transferred to ALU which executes the instruction.
3. The result is stored in a temporary accumulator.

Next Instruction

Storing the accumulator contents into the main memory is a separate task performed at the next CPU clock.

Low-Level programming

Instruction Set

- A list of all the possible operation codes available on a particular machine type, together with their associated memory addressing schemes.
- Each machine (CPU) is (only) able to directly execute instructions of its own set

Instruction

- A description of an operation to be performed including the operands, or where to find them, and where to put the result.
- Fixed-size string of bits (typically of the same length as memory words)

Example of machine code (MIPS architecture, 32 bits)

Load a value into register 8, taken from the memory cell located 68 cells after the location listed in register 3:

```
[  op |  rs |  rt |   (relative) address]
   35     3     8           68          decimal
 100011 00011 01000 00000 00001 000100   binary
```

Assembly language(s)

- Consists of machine Op-codes written in alphabetic form with mnemonic significance.
- Symbols instead of bits denote registers and memory addresses
- One-to-one correspondence between Assembly instructions and machine operations
- Enables compact and efficient code.
- The programming effort is however complex and difficult, even for the simplest machine operations.
- The assembly code is machine-dependent and not easy to port to different architectures.

Example: Euclide's algorithm

Used for calculating the greatest common divisor (gcd) of two positive integers **a**, **b** such that **a ≠ 0 OR b ≠ 0**

Based on this property:

- If **b ≠ 0** then **gcd (a, b) = gcd (b, a mod b)**
- **gcd (a, 0) = a**

Euclide's algorithm pseudo-code

Given **a, b**

1. If the value of **b** equals 0 go to step 5
2. Let **r** be the remainder of the division of **a** by **b**.
3. Store in **a** the value of **b**
4. Store in **b** the value of **r**.
5. Go back to step 1.
6. Print the value of **a** (gcd = **a**)

Euclide's algorithm encoding (generic Assembly, no I/O)

```
begin: LOAD R1 a        ; contents of memory cell a copied into reg. 1
       LOAD R2 b        ; contents of memory cell b copied into reg. 2
       JZERO R2 end     ; jump to "end" if R2 holds 0
       MOD R1 R2        ; remainder of a div b stored in R1
       STORE R1 b       ; R1 contents stored in memory cell b
       STORE R2 a       ; (old)  b contents stored in a
       JUMP begin       ; jump to "begin" (looping …)
end : ….                ; gcd is stored in a
```

Euclide's algorithm (Intel 8086 Assembly)

```
DATA SEGMENT
  NUM1 DW 000AH
  NUM2 DW 0004H
  GCD DW ?
DATA ENDS
CODE SEGMENT
  ASSUME CS:CODE,DS:DATA
  START: MOV AX,DATA    ;Load the Data to AX.
         MOV DS,AX      ;Move the Data AX to DS.
         MOV AX,NUM1    ;Move the first number to AX.
         MOV BX,NUM2    ;Move the second number to BX.
  UP: CMP AX,BX         ;Compare the two numbers.
      JE EXIT           ;If equal, go to EXIT label.
      JB EXCG           ;If first number is below than second,
                        ;go to EXCG label.
  UP1: MOV DX,0H        ;Initialize the DX.
       DIV BX           ;Divide the first number by second number.
       CMP DX,0         ;Compare remainder is zero or not.
       JE EXIT          ;If zero, jump to EXIT label.
       MOV AX,DX        ;If non-zero, move remainder to AX.
       JMP UP           ;Jump to UP label.
  EXCG:XCHG AX,BX      ;Exchange the remainder and quotient.
       JMP UP1         ;Jump to UP1.
  EXIT:MOV GCD,BX      ;Store the result in GCD.
       MOV AH,4CH
       INT 21H
  CODE ENDS
  END START
```

High-Level programming Languages

- **High-Level Language** Easier to read, write, and maintain than assembly language and machine code.
- Abstractions on data (data types are introduced) and execution's control-flow (jumping is avoided)
- It is processor independent and can be run on different machines types, but must first be translated into a machine language by a compiler or interpreter which are both processor specific..
- High level code is much easier to read and maintain.

Execution of a program written in a HL language: compilation

- **Compiler** A software program which converts source code written in high level language to machine code which runs on a specific machine.

- (Usually) produces an efficient code (embedded into an executable file), even if less efficient than using low-level languages.

- Execution and memory requirements may be too large for resource-limited applications.

Execution of a program written in a HL language: interpretation

- **Interpreter** An interpreter also converts high-level language instructions to machine code but enables the source code to run "directly" on the machine
- High-level instructions are translated to machine code every time a new instruction is encountered in the program .
- The execution may be slow and inefficient.
- But it is less resource-consuming and enables a faster development process (more flexibility)

Examples of interpreted/compiled languages

- Compiled:

  C, Pascal, C++, Java, Go, C#, …

- Interpreted:

  Python, Javascript, Lisp, Ruby, Linux Shell, PHP, Perl

- Some (Python, Java, ...) actually use an hybrid approach

  e.g., the Java compiler results in a "bytecode" which is interpreted by the Java Virtual Machine (which is machine dependent)

HL programming languages paradigms

- Imperative/Procedural (C, Go, Pascal, Fortran, Shell, ...)
- Object-Oriented (Java, C++, C#, SmallTalk, Objective-C, Swift …)
- Functional (Lisp, Scheme, Haskell, …)
- Logic/declarative (Prolog, …)

Some modern languages (e.g., Swift, Java, Python, Scala) are "multi-paradigm" (e.g., combine OO and functional features)

The C language

- First appearance: 1972
- Developed (originally) by Denis Ritchie at Bell Labs, as the language of Unix operating system (Ritchie + Ken Thompson)
- Made popular by the textbook "The C Programming Language", 1978, Kernighan, Ritchie
- standardized by the American National Standards Institute (ANSI) since 1989 (C89, C99, C11, ..)
- Pure imperative, general-purpose, high level
- Very efficient (equipped with low-level features)

Structure of a simple C program

```
#include <stdio.h>

int main() {
   /* this is a C program */
   printf("Welcome to Programming-C! \n");

   return 0;
}
```