UCLA Samueli School of Engineering

CS-M151B
Computer Systems Architecture

Blaise Tine
UCLA Computer Science

Logistics

- TA introduction
- Bruincast

The Power of Abstraction

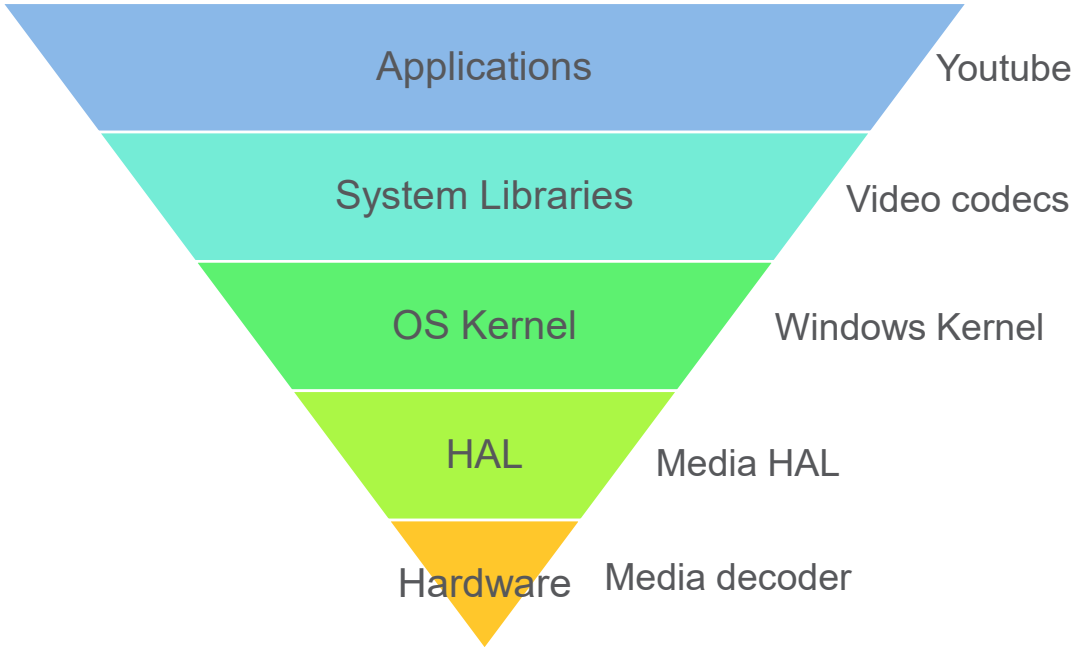
- **What is an abstraction?**
  - Simplifying a complex system using interfaces
  - A fundamental concept in Computer Science
- **Common abstractions we use in computer science?**

The Power of Abstraction

- Reduction of complexity
- Modularity
  - Enhanced focus
  - Reusability
  - Scalability
  - Maintainability
- Portability
- Divergence
  - Separate growth

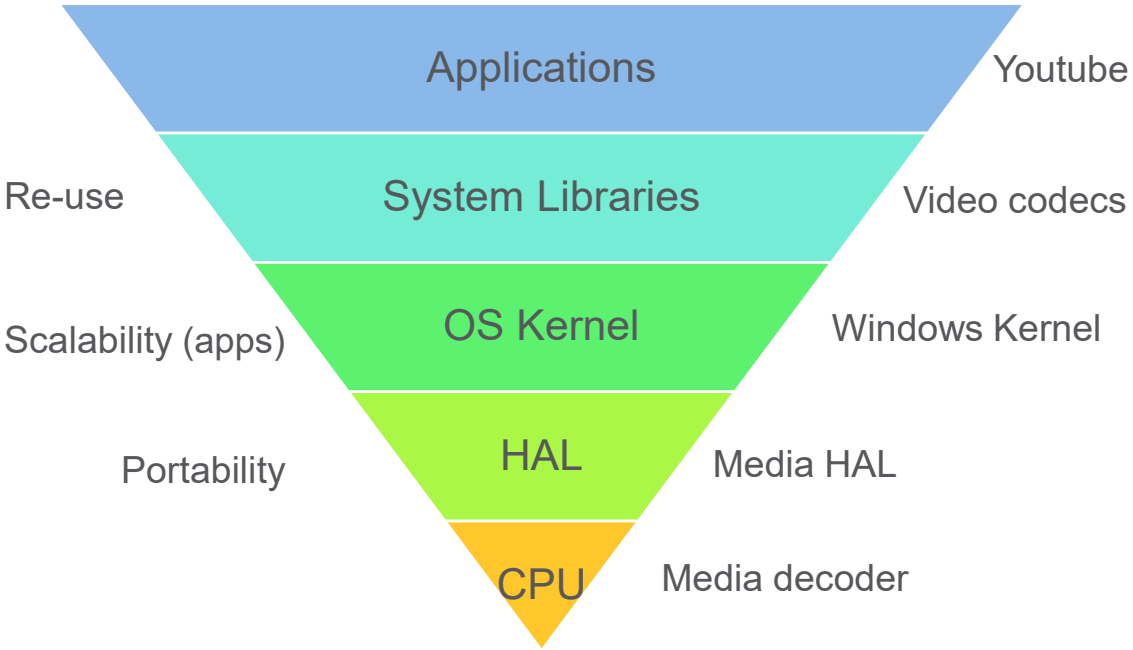
The Power of Abstraction

- **Computer System Abstraction**



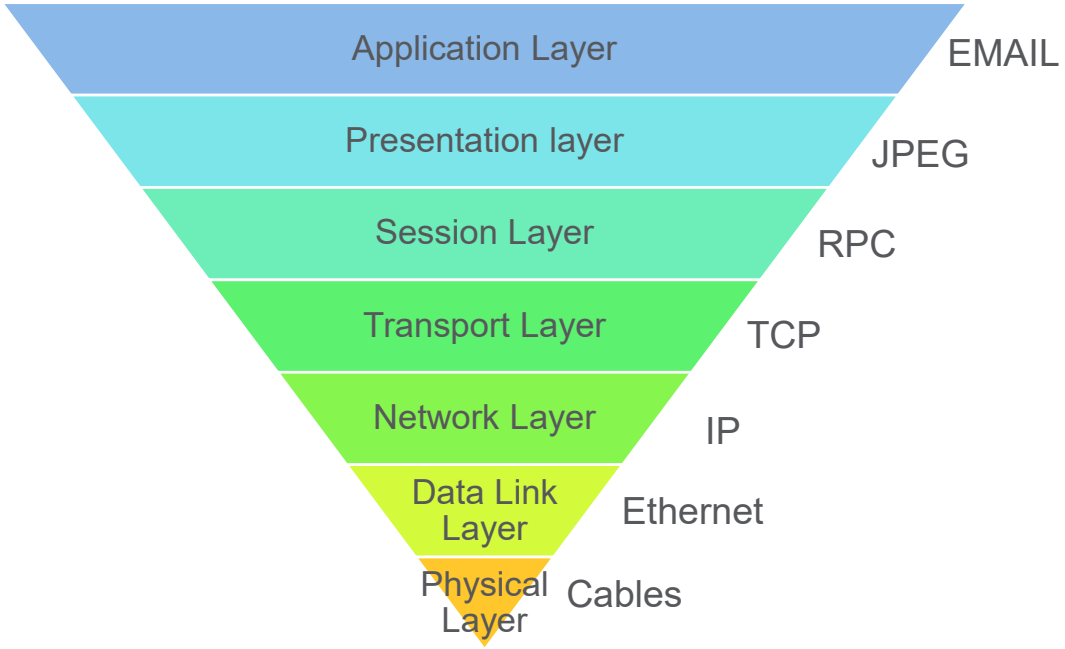
The Power of Abstraction

- **Computer Systems Abstraction**



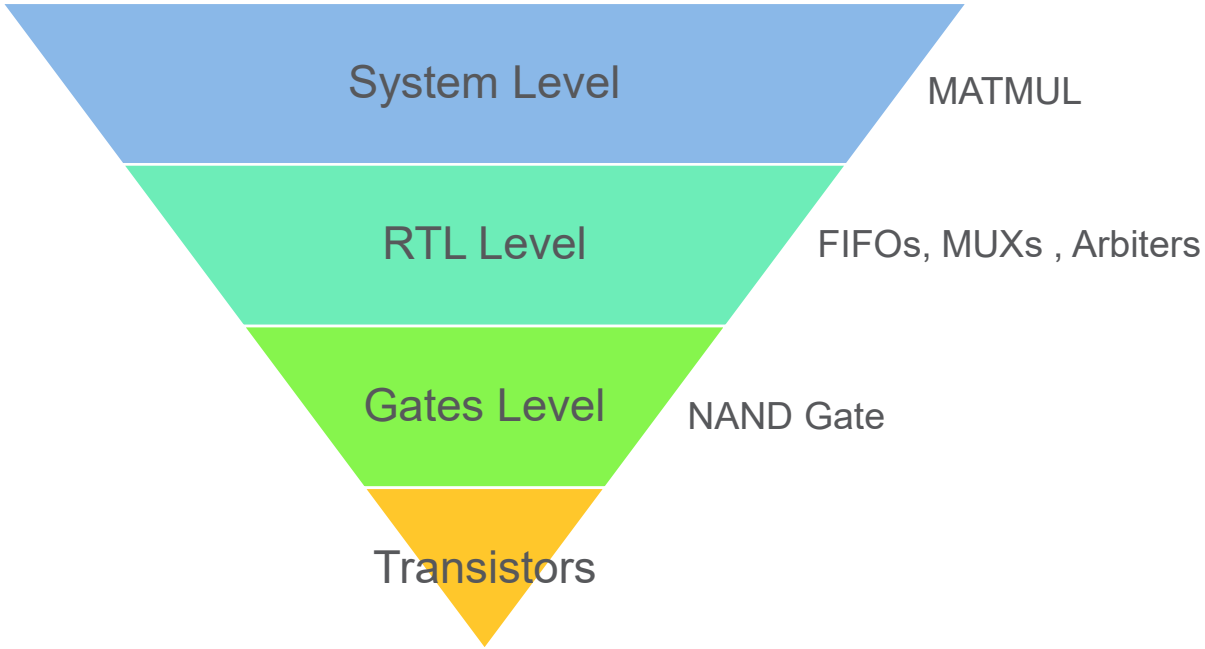
The Power of Abstraction

- **The classic OSI Model of Networking**



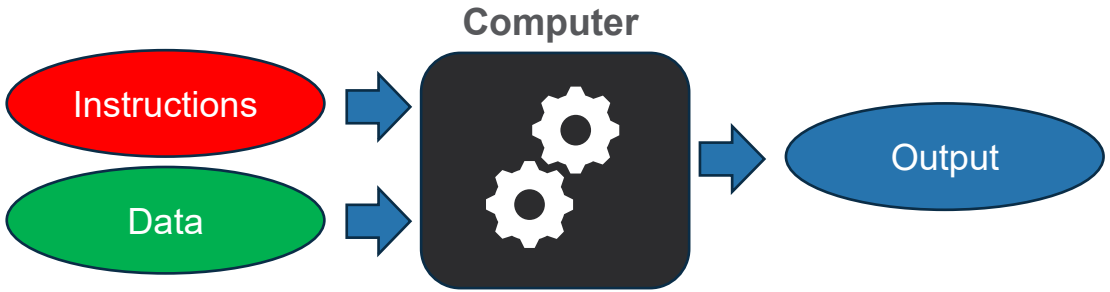
The Power of Abstraction

- **Hardware Abstraction**



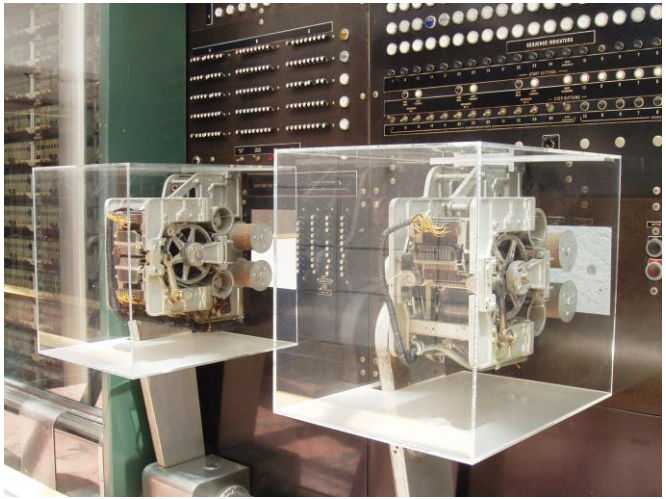
A basic computer model

- Perform tasks
  - Input
    - Instructions
    - Data
  - Execution
  - Output
    - Result



Early computers

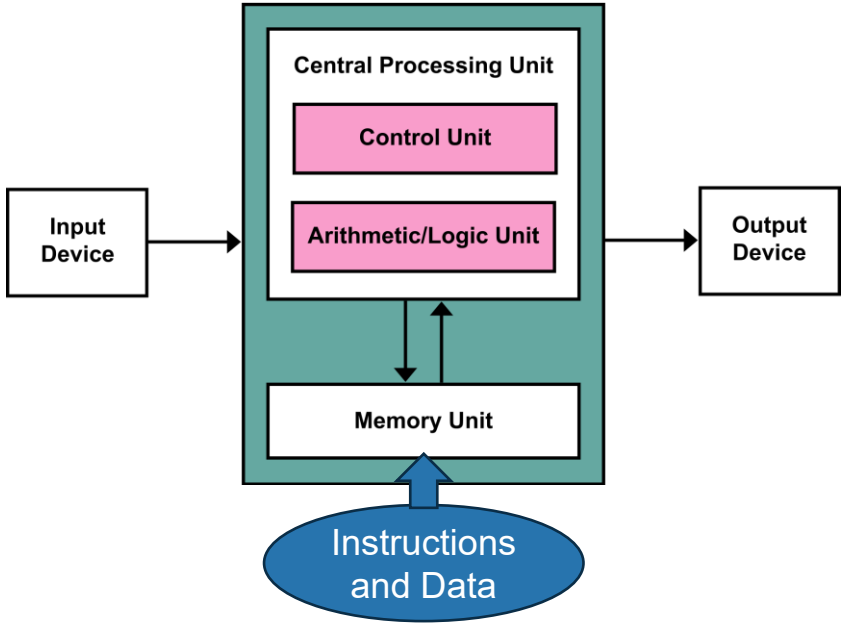
- Operated more like programmable calculators
  - Manual configuration using switches
  - Simple integer arithmetic
  - Instructions physical media
    - Punched cards
  - Electromechanical
  - e.g. Mark I
    - Storage 72 numbers
    - 3 additions / sec



IBM Harvard Mark I (1944)

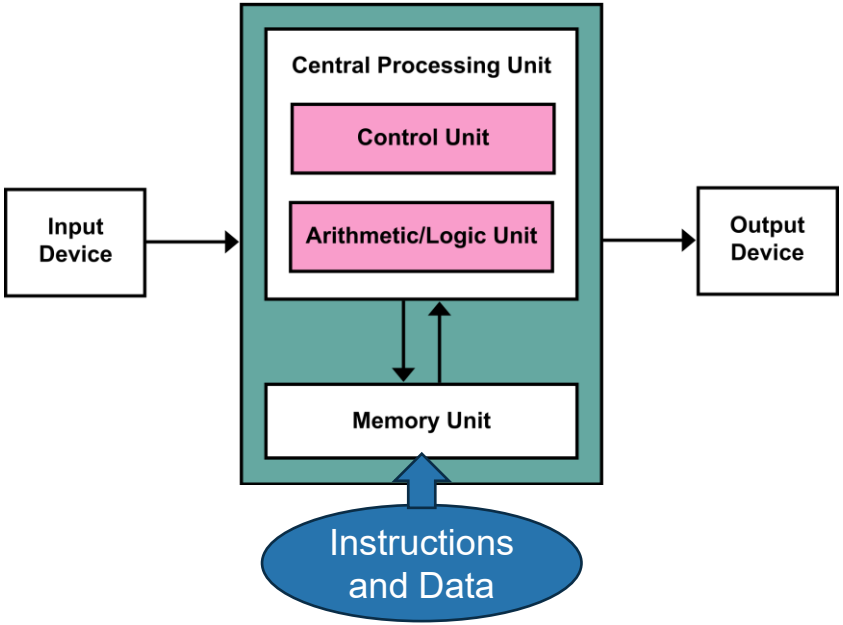
The Von-Newman Architecture

- **First digital computer model (1945)**
  - Stored-program concept
    - Electronic program – no punch card
  - Unified Memory structure
    - Instructions and data together
  - CPU Pipeline
    - Fetch->decode->execute



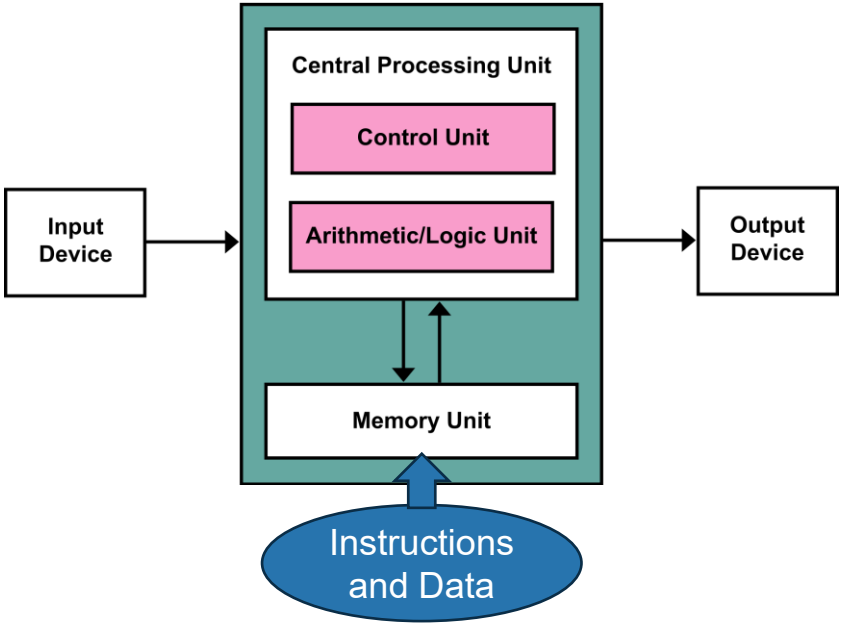
The Von-Newman Architecture

- **Basic architecture**
  - Input device
  - Central processing unit (CPU)
  - Memory Unit
  - Output Device



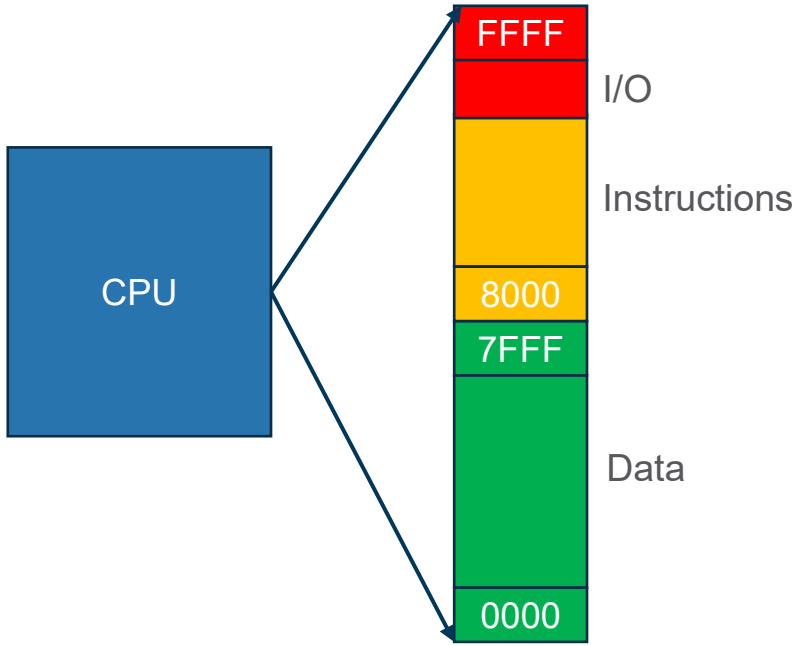
The Von-Newman Architecture

- **Memory Unit**
  - Digital storage
  - Store Instructions and data
  - Programmable
  - Flexibility



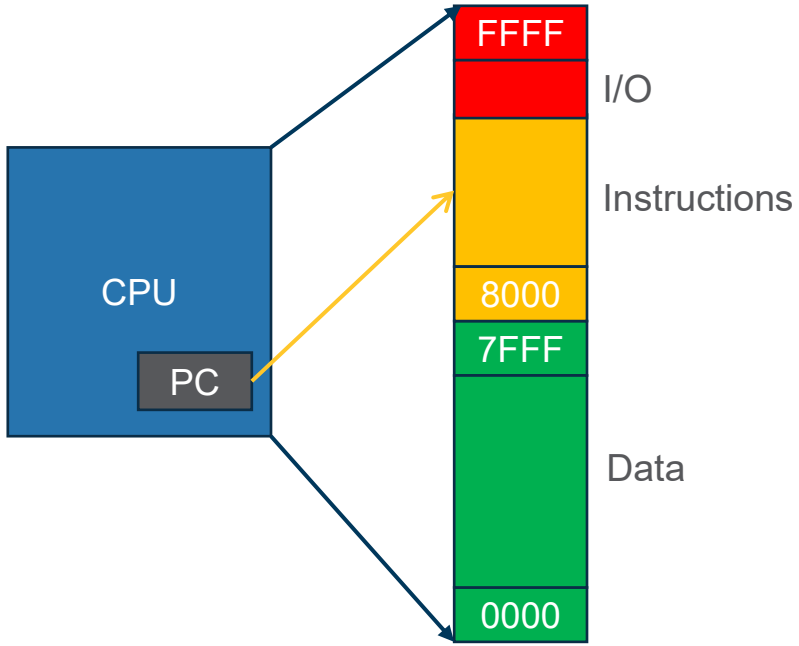
The Von-Newman Architecture

- **Shared address space**
  - Address Partitioning
  - Data region
  - Instruction region
  - Inputs/Outputs region



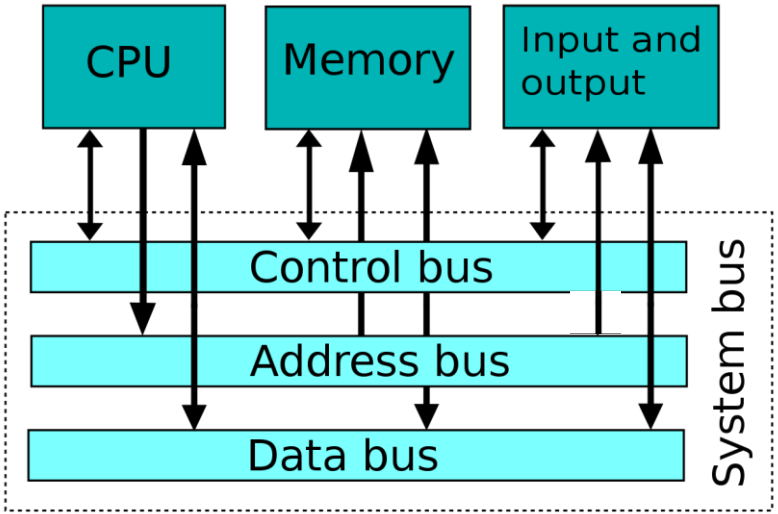
The Von-Newman Architecture

- **Shared address space**
  - Shared system bus
  - Simple control logic
  - Single memory interface
  - Program counter (PC)
    - Next instruction address



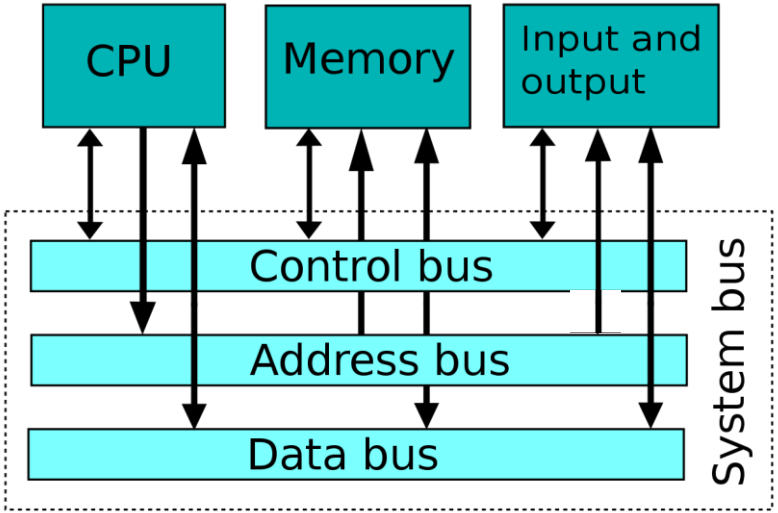
The Von-Newman Architecture

- **System Bus**
  - Address bus
    - Specify memory address
    - Uni-directional
  - Data bus
    - Specify data to transfer
    - Bi-directional
  - Control bus
    - Schedule transfer
    - Bi-directional – why?



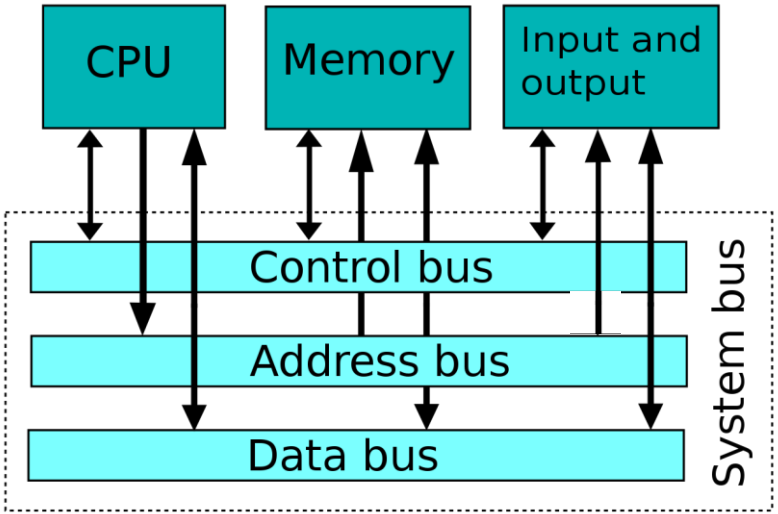
The Von-Newman Architecture

- **Limitations**
  - Shared bus bottleneck
    - Too many connected devices
    - Control switching
  - One instruction at the time



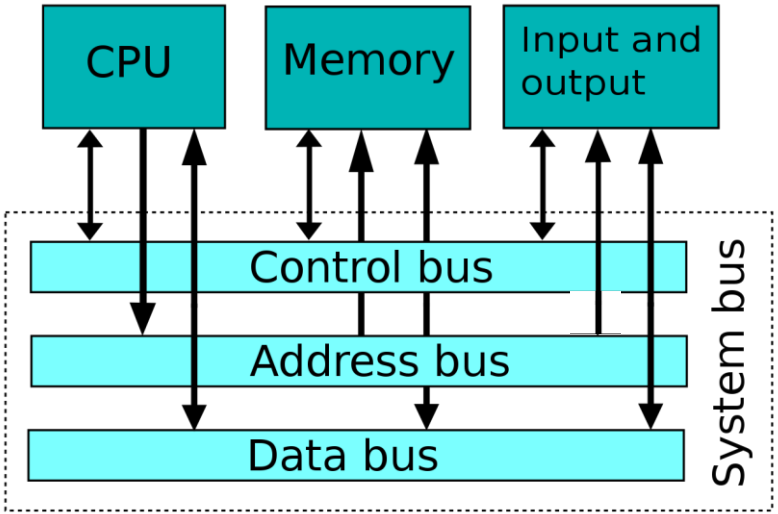
The Von-Newman Architecture

- **Sample Exercise:**
  - Assume following system bus design
    - 32-bit address bus
    - 8-bit data bus
    - 2-bit control bus [W][S]
      - W: read/write enable
        - 0 => read
        - 1 => write
      - S: chip select
        - 0 => Memory
        - 1 => I/O



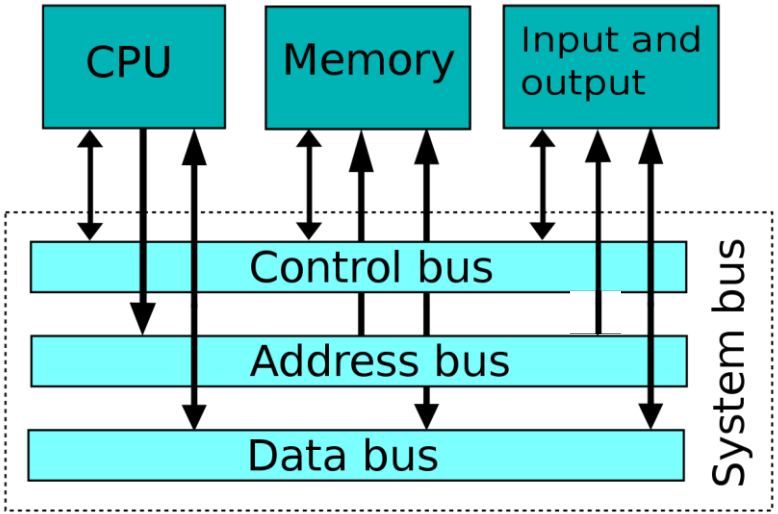
The Von-Newman Architecture

- **Sample Exercise:**
  - **CPU wants to read a byte from memory at address 0x80000000**
  - Step1: CPU places 0x80000000 on address bus.
  - Step2: CPU selects memory for read by placing 00 on control bus
    - W=0, S=0
  - Step3: Memory loads the byte at specified address location in memory and puts it on the data bus
  - Step4: CPU reads the byte from the data bus



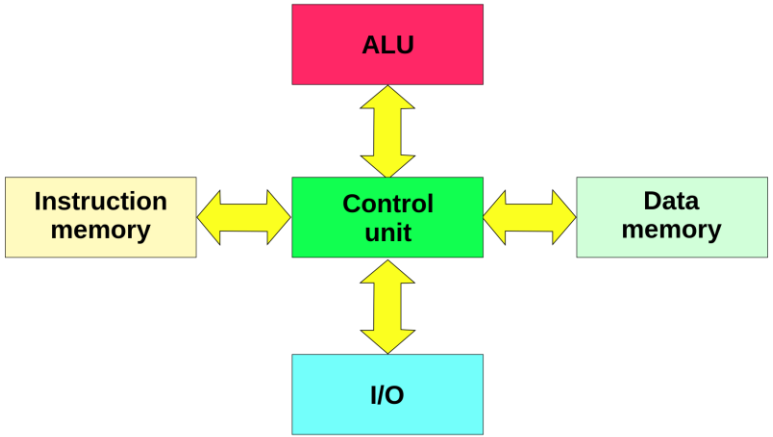
The Von-Newman Architecture

- **Sample Exercise:**
  - **CPU wants to write a byte to I/O at address 0xFF000000**
  - Step1: CPU places 0xFF000000 on address bus.
  - Step2: CPU places the byte to be sent on the data bus
  - Step3: CPU selects I/O for write by placing 11 on control bus
    - W=1, S=1
  - Step4: I/O reads byte from the data bus and stores it to the specified address location in I/O



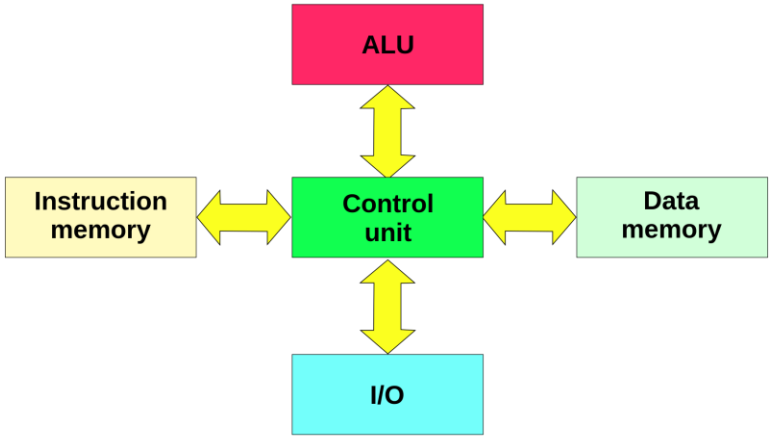
Harvard Mark-IO Architecture

- **Harvard Architecture Model**
  - Harvard mark-I (1930-1952)
  - Separate memory buses
    - Instruction memory
    - Data memory



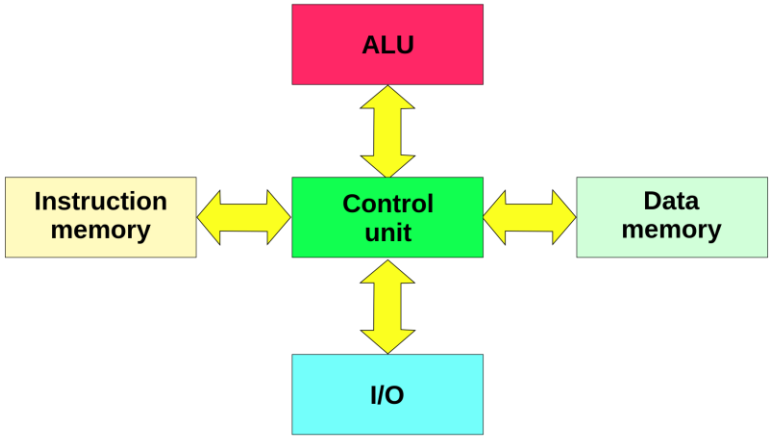
The Von-Newman Architecture

- **Harvard Architecture Model**
  - Advantages
    - Simultaneous access
      - Fetch data and instructions in parallel
      - Increases performance
    - Can specialize instruction memory
      - Can be read-only (security)
      - Can be non-volatile (flexibility)
      - Can be faster technology (speed)



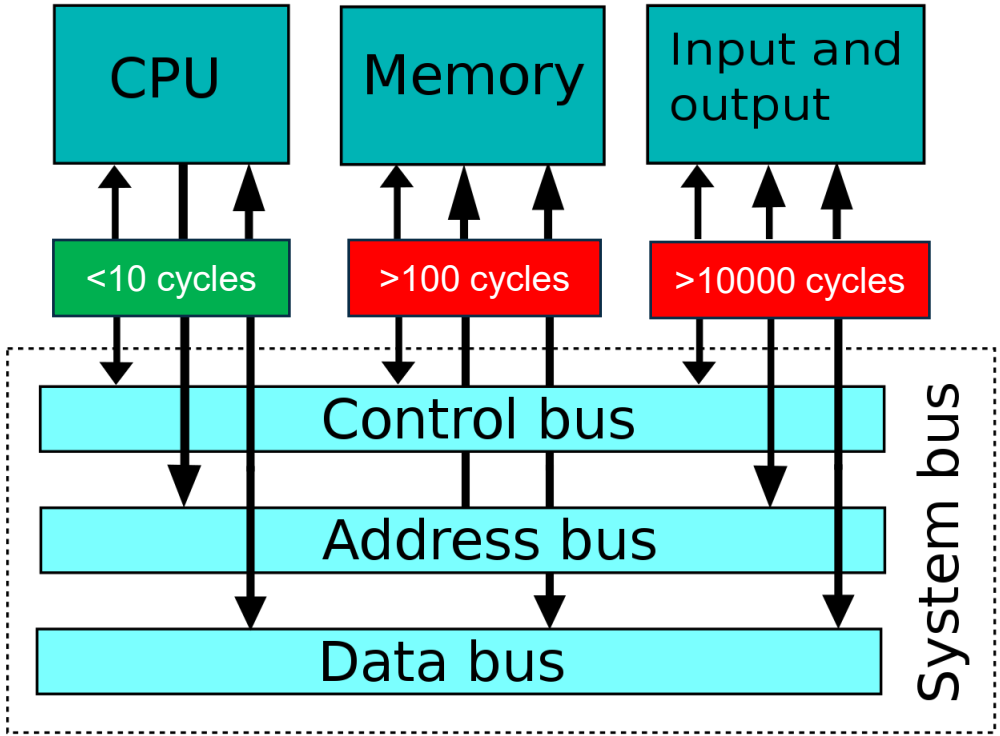
The Von-Newman Architecture

- **Harvard Architecture Model**
  - Limitations
    - Complexity and cost
      - Multiple memory buses
    - Limited software flexibility
      - Self-modifying code



The Von-Newman Architecture

- **Von-Newman "bottleneck"**
  - High memory access latency
  - High I/O latency
  - High CPU idle time
    - Waiting for memory or I/O

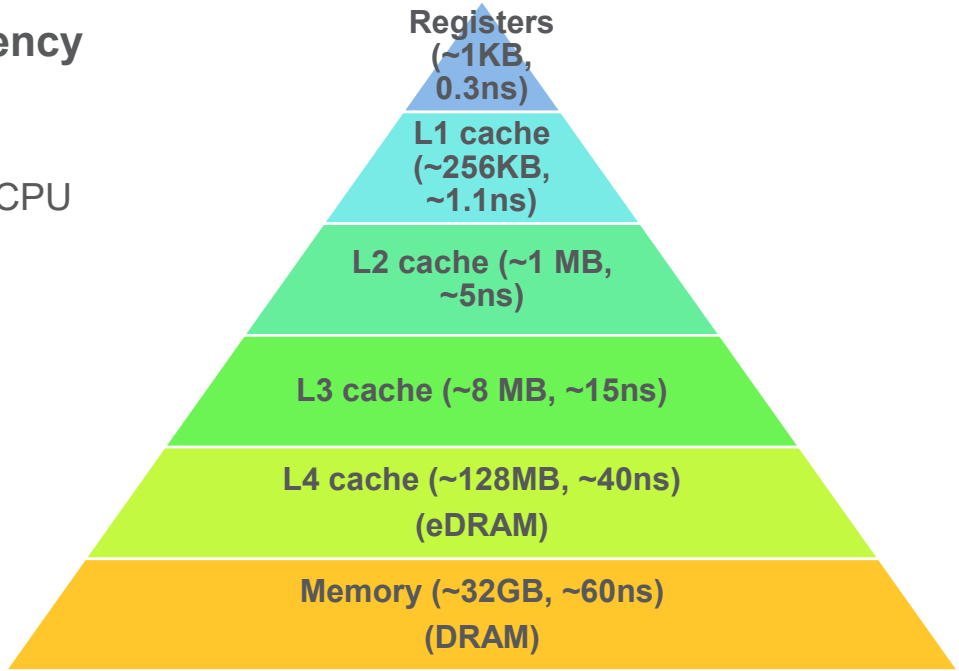


Attendance

- Please Fill the form with today’s keyword to register your attendance.

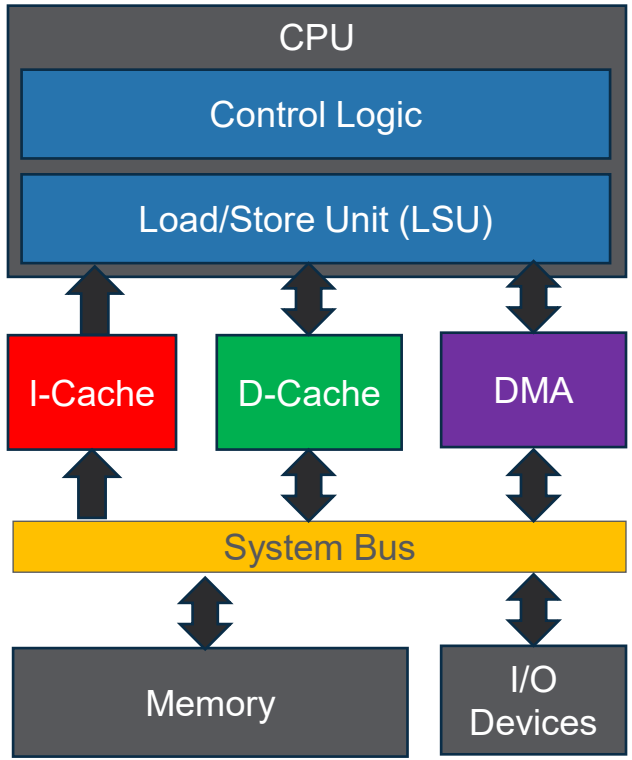
The Von-Newman Architecture

- **Mitigating Memory Access latency**
  - We use deeper caches
  - Worsen by multicore
  - Cache area getting larger than CPU
  - The problem still present today



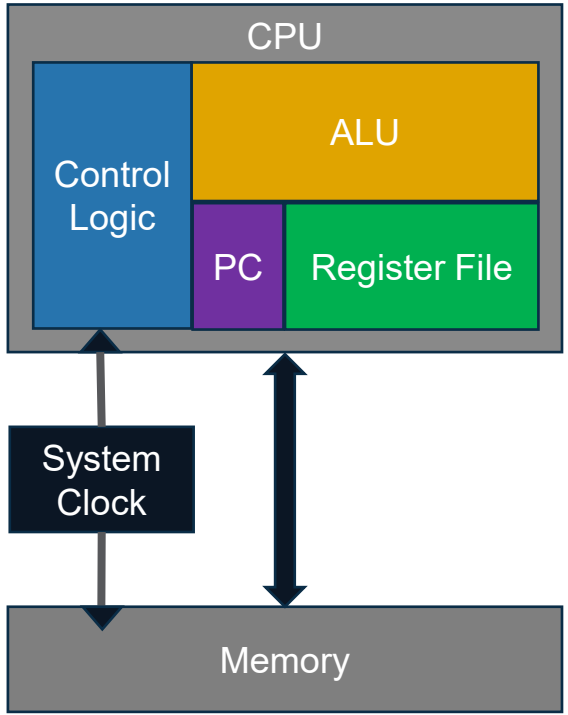
The Von-Newman Architecture

- **Mitigations**
  - Caches
    - Instruction cache
    - Data cache
  - On-chip memory
  - Direct Memory Access (DMA)
    - Deferring I/O transfer



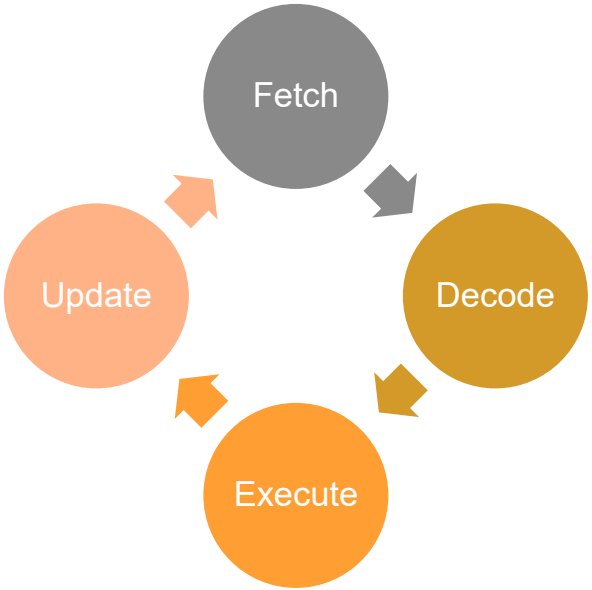
Computer Hardware Model

- System Clock
  - Determine transistors speed
- Control Logic
  - Fetch and schedule instructions
- Arithmetic Logic Unit
  - Execute instructions
- Program Counter (PC)
  - Stores current program location
- Register File
  - Local small and fast storage
- Memory
  - External slow and large storage



Computer Execution Model

- Fetch
  - Fetch next instruction
  - From program counter (PC)
- Decode
  - Extract opcode & operands
- Execute
  - Perform the operation
- Update
  - PC and memory



The Classic CPU Performance Equation

- CPU time = Instruction Count x CPI x Clock Cycle Time
  - CPI = Clock Cycle per Instruction

- CPU time = $\frac{Instruction\ Count\ x\ CPI}{Clock\ Rate}$

- How do you increase performance?
  - Reducing the CPU time

SPECint benchmarks – Patterson & Hennessy book

Reducing CPU Time

- CPU time = $\frac{Instruction\ Count\ x\ CPI}{Clock\ Rate}$
- Reduce Instruction Count
  - Software, compiler, ISA
- Reduce CPI
  - Microarchitecture
- Increase Clock Rate
  - Process technology
  - Microarchitecture efficiency (propagation delay)

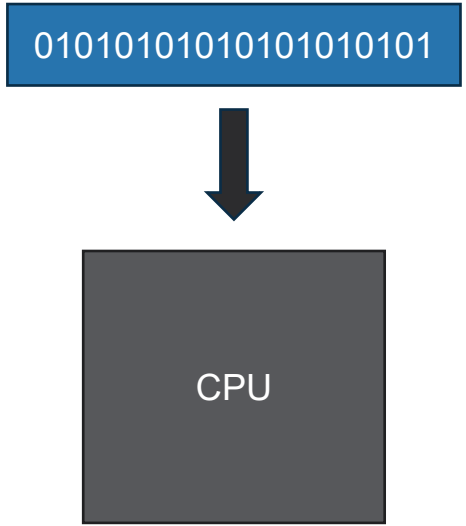
Reducing CPU Time (2)

- Example: CPU executing a MatMul workload has baseline performance:
  - Instruction Count: one billion (10^9) instructions
  - CPI: 2 clock cycles per instructions
  - Clock Rate: 2 GHz
- New process manufacturing increases the clock rate to 2.5 GHz
- What is the speed-up?
  - Baseline CPU Time = $\frac{10^9 \times 2}{2 \times 10^9}$ = 1 sec
  - Improved CPU Time = $\frac{10^9 \times 2}{2.5 \times 10^9}$ = 0.8 sec
  - Speed-up = $\frac{Baseline\ CPU\ Time}{Improved\ CPU\ Time}$ = $\frac{1}{0.25}$ = 1.25x

Writing the program

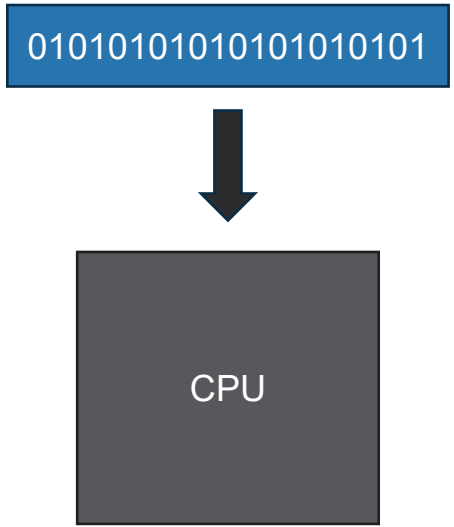
- **Simply write machine code**
  - Early software
- **Disadvantages**
  - Hard to write
  - Difficult to debug

Possible advantages?



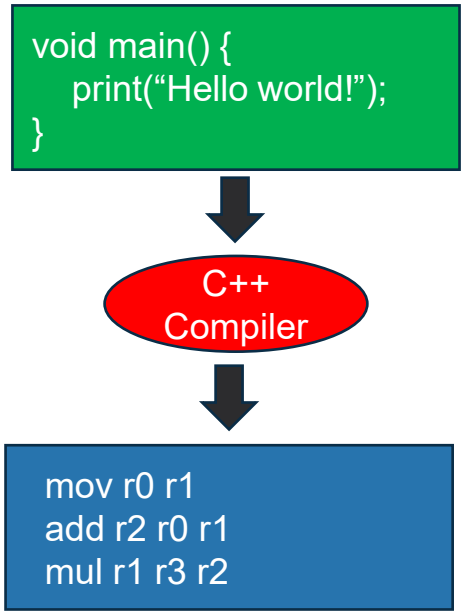
Writing the program

- **Hand-written machine code advantages**
  - Full h/w control
  - Can write efficient code



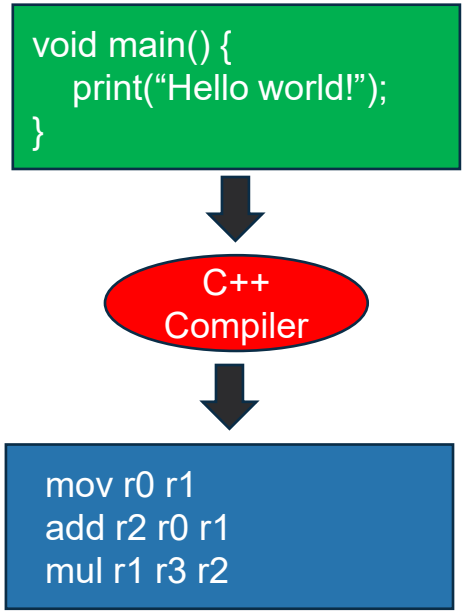
Writing the program

- **Use a high-level language**
  - C++, Fortran, Python
  - Higher abstraction
    - Closer to the problem domain
    - Objects and algorithms



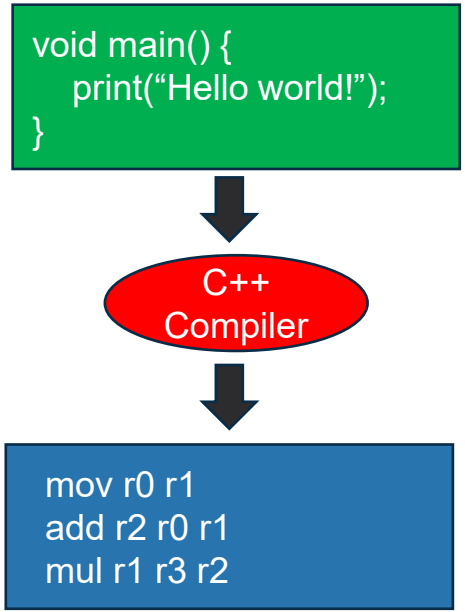
Writing the program

- **Compiler**
  - Convert to assembly code
  - Enable advanced optimizations
    - Requiring program analysis
    - Difficult to do manually



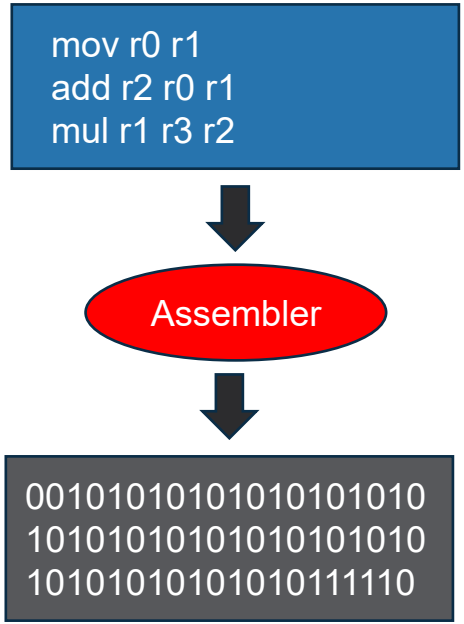
Writing the program

- **Assembly code**
  - Textual representation of instructions
  - Depend on target machine architecture



Writing the program

- **Assembler**
  - Convert assembly code to machine code
  - Machine code contains encoded binary
    - Instructions and data
  - Why is this step separate from the compiler?

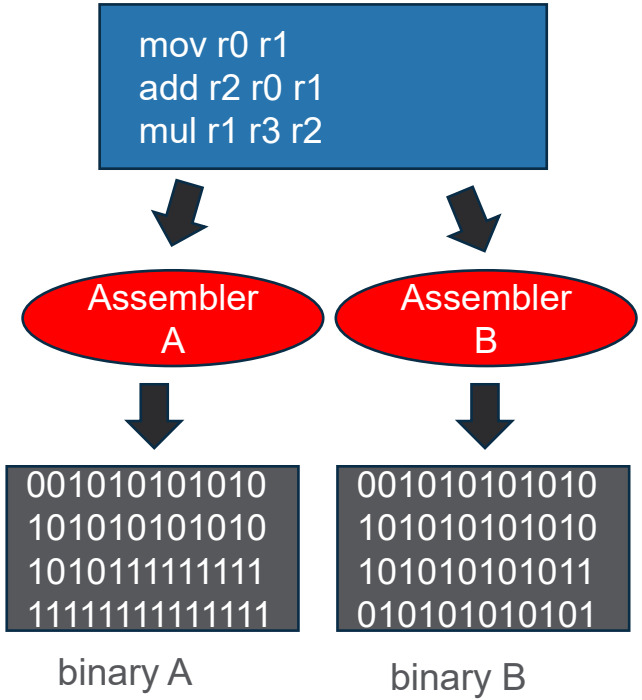


Writing the program

- **Handling Machine code variations**
  - Simplified pseudo instructions
  - Context-specific pseudo instructions
  - Context-specific macro expansion
  - Can depend on the platform or machine

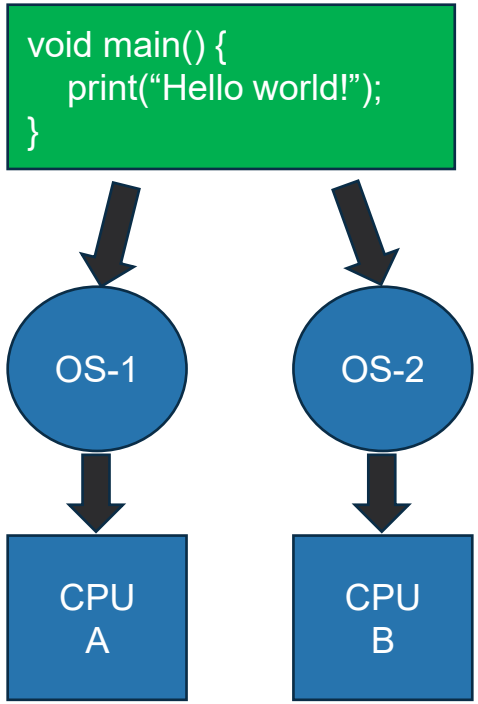
Example: RISC-V Nop pseudo instruction

- nop ⇔ addi x0, x0, 0



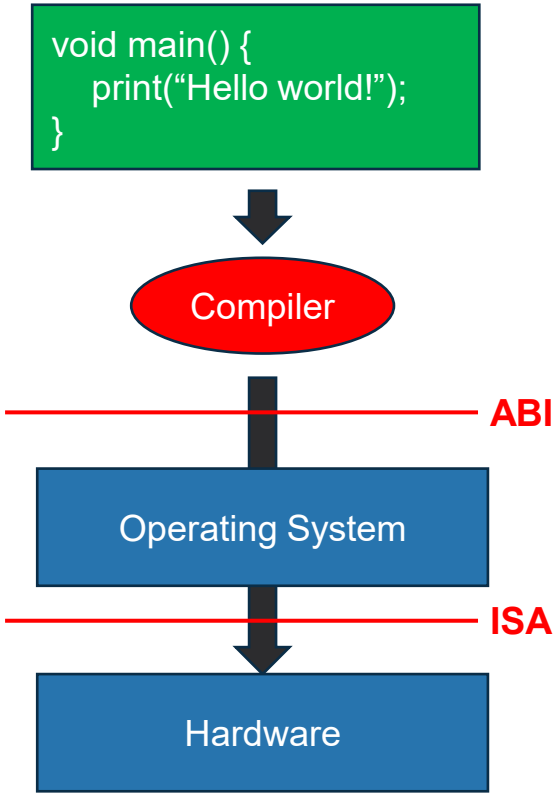
Writing the program

- **Software portability challenges**
  - Want to use a different CPU?
  - Want to use a different OS?



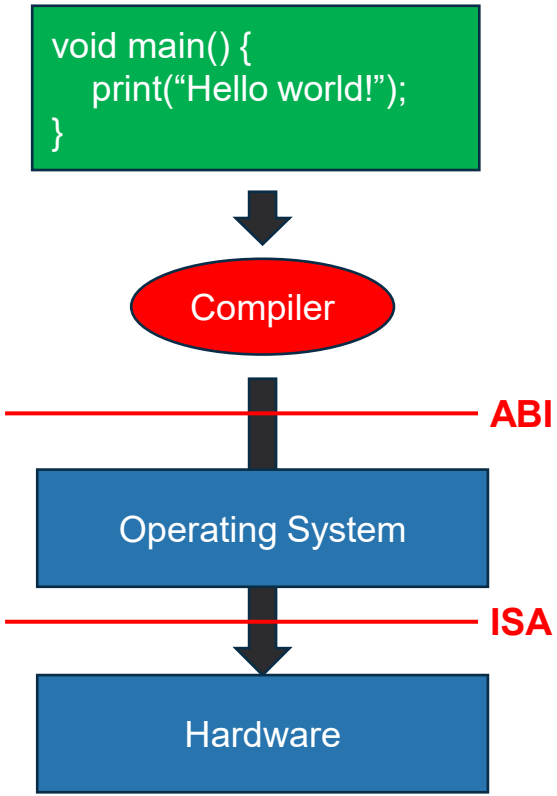
Writing the program

- **Software Abstraction layers**
  - Application Binary Interface
  - Instruction Set Architecture



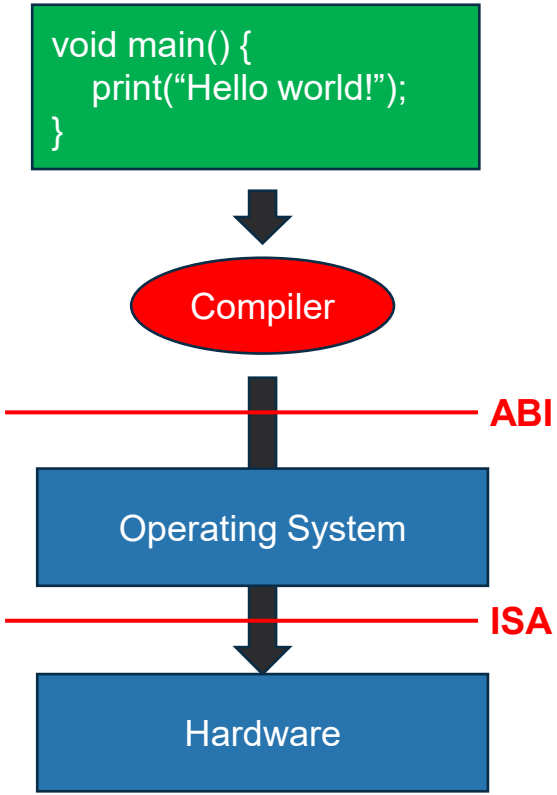
Writing the program

- **Application Binary Interface**
  - A contract between S/W and OS
  - Binary compatibility
    - Can OS launch application A
    - Can application A interact with application B
  - How?
    - Function calls convention
    - Stack frame organization
    - Binary object format
    - Etc..



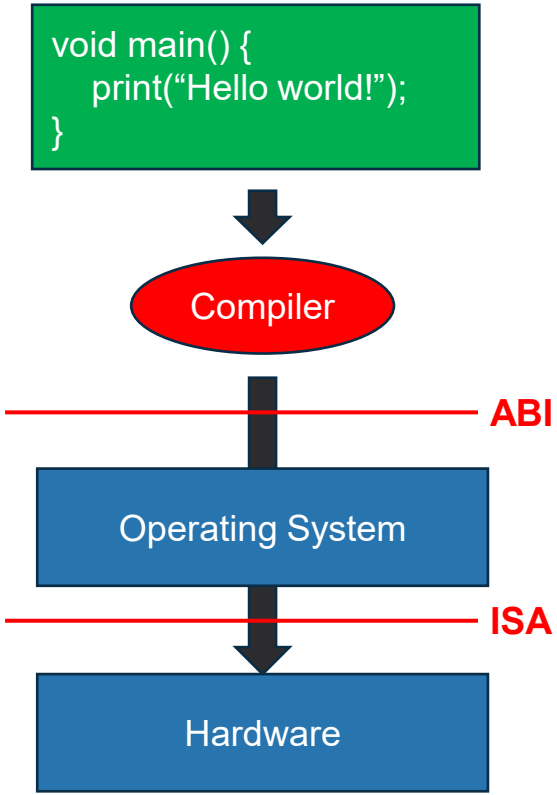
Writing the program

- **Instruction Set Architecture**
  - A contract between S/W and H/W
  - Software/Hardware divergence
    - Can evolve independently
  - Hardware abstraction
    - H/W Simulation



Writing the program

- **Instruction Set Architecture**
  - S/W view
    - Operations
    - Storage locations
  - Hardware view
    - Instructions
    - Data



Instruction Set Architecture

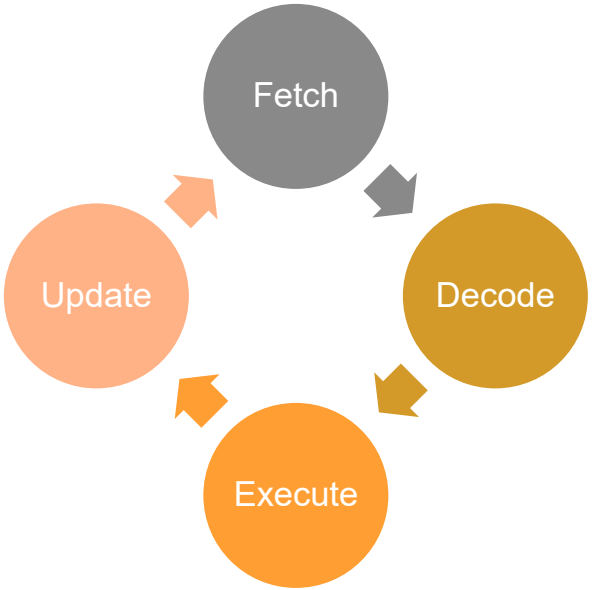
- **Background**
  - IBM System/360 architecture (1964)
  - First introduction of separation between H/W and S/H
  - Features
    - Complex Instruction Set Computer (CISC)
    - x16 32-bit general-purpose registers
    - x4 64-bit floating-point registers

Instruction Set Architecture

- **Mainstream ISA's**
  - x86/x86-64 (AMD64) ISAs: Intel, AMD
  - ARM ISA: Apple, Samsung, Qualcomm
  - MIPS ISA: Embedded systems, network routers, etc..
  - SPARC ISA: Sun Microsystems
  - RISC-V ISA: SiFive, academia, enthusiasts

How do you design an ISA?

- How fast can we fetch?
- How fast can we decode?
- How fast can we execute?
- How efficient will the hardware be?



Q&A