# 2 Computer Architecture

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Understand the main components of a computer.
- Understand how the CPU executes instructions.
- Understand how the CPU interacts with the memory and peripheral devices.

## 2.1 Overview

The previous chapter discussed the mechanism of arithmetic circuits, which are core components of CPUs. This chapter further explores the workings of CPUs. Before that, we will first provide an overview of computer architecture. A computer is generally composed of a CPU, memory, input and output devices, storage, and network devices (Figure 2.1).

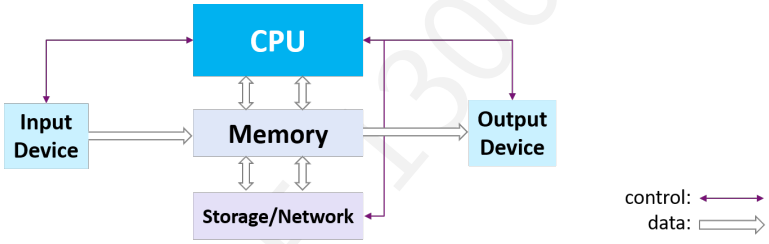



Figure 2.1: Main components of a computer.

- *CPU (Central Processing Unit):* Acts as the brain of the computer by fetching instructions from memory, decoding them, performing computations in the Arithmetic Logic Unit (ALU), and coordinating data transfers between memory, storage, and input/output devices.
- *Memory:* Temporarily holds data and programs for fast access by the CPU.
- *Input devices:* Bring data and instructions into the system from the external world.
- *Output devices:* Present results and processed information to the user.
- *Storage:* Provides persistent storage for data and programs, even when the computer is turned off. It serves as both an input and an output device.
- *Network:* Enables communication and data exchange between computers and other devices over local or wide-area networks (such as the Internet).

These components communicate through buses, such as the control bus, which coordinates signals, and the data bus, which transfers data. Next, we will discuss how a CPU works and interacts with other components.

## 2.2 Central Processing Unit (CPU)

The CPU executes instructions from programs stored in memory, performing arithmetic, logical, control, and input/output operations. It consists of several key components:

- *Arithmetic Logic Unit (ALU):* Performs arithmetic and logical operations on data.
- *Control Unit (CU):* Directs the flow of data and instructions within the CPU by fetching, decoding, and coordinating their execution among the other components.
- *Load/Store Unit (LSU):* Handles memory access operations by loading data from memory into registers and storing data from registers back into memory.
- *Registers:* Small, high-speed storage locations inside the CPU for holding data, instructions, and addresses temporarily.

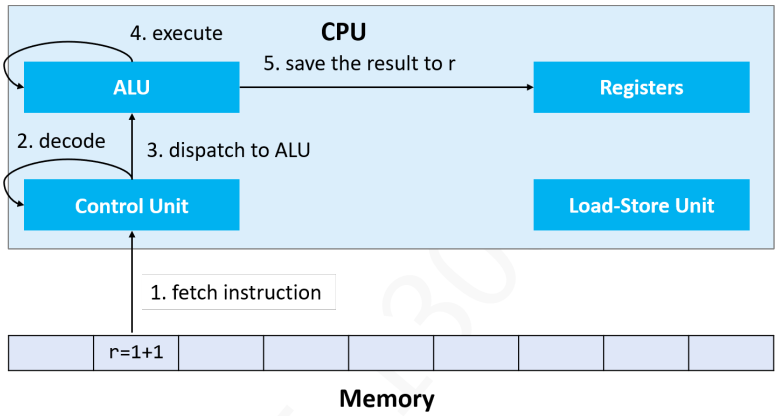



Figure 2.2: Executing the instruction `r=1+1`.

Figure 2.2 demonstrates how these components collaborates in executing an instruction. The execution of instructions follows a sequence known as the fetch–decode–execute cycle:

- *Fetch:* The control unit retrieves the raw binary instruction bits from memory.
- *Decode:* The control unit interprets the instruction, identifying required operands, target registers, and whether the ALU or LSU will be involved.
- *Execute:* The instruction is carried out. For arithmetic or logical operations, they are performed in the ALU; for memory transfers operations, they are handled by the LSU. The results are finally written back to registers or memory as directed by the control unit.

## 2.3 Memory

Memory in a computer generally refers to RAM (Random-Access Memory), a type of volatile memory that allows both reading and writing of data based on specific memory addresses. RAM enables direct access to any data item by specifying its address, ensuring fast retrieval regardless of where the data is located within the physical memory space. Each data unit typically occupies 1 byte, or 8 bits, which is the smallest addressable unit of memory in most systems. One of the defining characteristics of RAM is

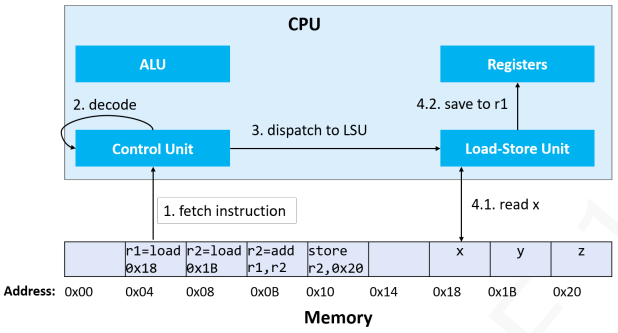
its uniform access time: it provides equal latency for accessing any memory location, meaning that the time to retrieve data does not depend on the specific location of the data within the memory.

Now we extend the integer addition instruction in Figure 2.2 to the addition of two variables (*i.e.*, `z = x + y`) whose values are stored in memory. This addition operation is compiled into four instructions, as shown in Code 2.1.

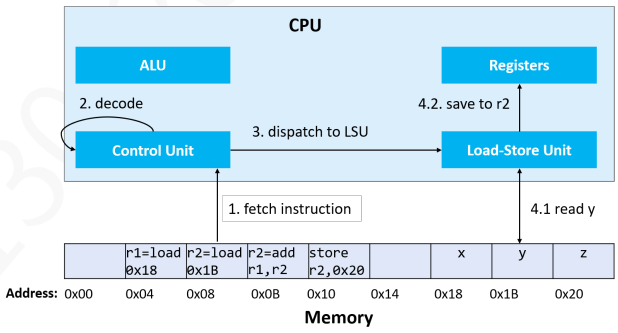
```
r1 = load x ; load the value of x from memory to register r1.
r2 = load y ; load the value of y from memory to register r2.
r2 = add r1, r2 ; add the two registers and save the result in r2.
store r2, z ; store the value of register r2 to the memory unit of z.
```

Code 2.1: Instructions of `z=x+y`.

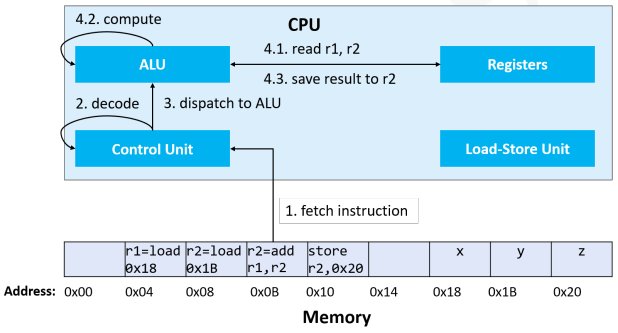
Note that the identifiers `x`, `y`, and `z` do not exist in the actual machine instructions. Instead, they are replaced with the corresponding memory addresses. Figure 2.3 illustrates a sample layout of these instructions and variables, as well as the execution steps. In this example, the three variables are stored at memory addresses `0x18`, `0x1B`, and `0x20`, respectively. The first instruction, `r1 = load x`, is located at memory address `0x04`. After executing this instruction, the program counter in the CPU points to the next instruction at address `0x08`. The CPU will then continue execution based on the program counter.



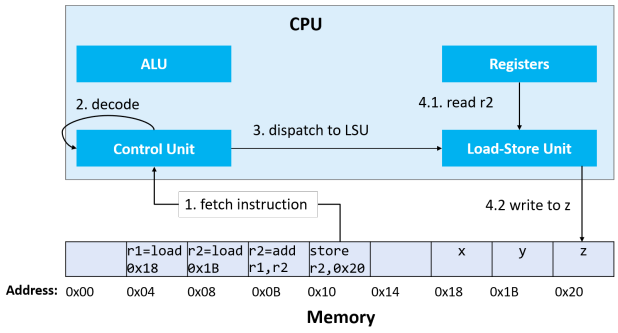

(a) Step 1: executing `r1 = load x`.




(b) Step 2: executing `r2 = load y`.




(c) Step 3: executing `r3 = add r1, r2`.




(d) Step 4: executing `store r2, z`.

Figure 2.3: An example layout of four instructions to compute `z = x + y` and their execution.

## 2.4 Instruction Set Architecture (ISA)

The ISA is the abstract interface between the software and hardware. It defines the instructions that a processor can execute, including the specific instructions and registers available in the CPU. Since instructions are encoded in memory, how many bits are used for each instruction? Are they of fixed length

or variable length? All these details are specified in the ISA documentation.

In general, an ISA defines several types of instructions, including *data movement*, *arithmetic and logic operations*, *control flow*, *etc.* Data movement instructions, such as `load` and `store`, transfer data between registers, memory, and I/O devices. Arithmetic and logic instructions, like `add`, `sub`, `and`, and `or`, perform mathematical and bitwise operations on data. Control flow instructions, such as `jump` and `branch`, modify the sequence of program execution, enabling conditional operations or redirection to different parts of the program. These instruction types are fundamental to how a processor interacts with both the program and memory, facilitating all essential computational tasks.

Different ISAs have different design details, each tailored to specific needs and trade-offs. The most commonly known ISAs include x86, x86_64, ARM, and RISC-V. x86 and x86_64 are complex instruction set architectures (CISC), known for their rich set of instructions that can perform multiple tasks (especially arithmetic operations and data movement) in a single operation. x86 is a 32-bit ISA, while x86_64 is its 64-bit counterpart, offering larger address spaces and enhanced performance for modern computing. In contrast, ARM and RISC-V are based on reduced instruction set computing (RISC) principles, with simpler and more streamlined instructions that allow for faster execution.

```
ldr   x1, [x0] ; load x to register x1
ldr   x2, [x1] ; load y to register x2
add   x2, x1, x2 ; x2 = x1 + x2
str   x2, [x0]  ; store the result in z
```

Code 2.2: Assembly code of ARM for `z = x + y`.

```
mov   eax, [x] ; load x to register eax

add   eax, [y] ; add y to register eax
mov   [z], eax ; store the result in z
```

Code 2.3: Assembly code of x86 for `z = x + y`.

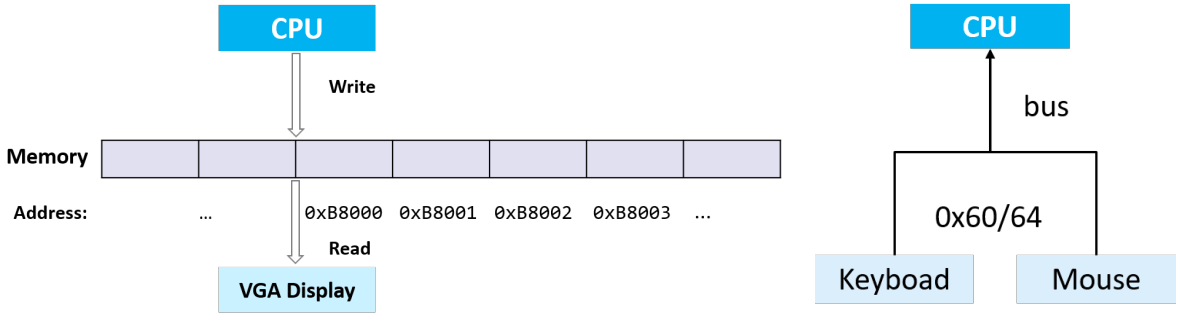
Code 2.2 and 2.3 demonstrate the assembly code of ARM and x86 for Code 2.1 respectively. By comparing these two versions, we can observe that in x86, the second instruction (loading the memory of `y`) and the third instruction (adding the value of `y`) can be combined into a single instruction, a capability that is not typically allowed in ARM or other RISC-based CPUs.

x86 remains the dominant instruction set architecture in desktop and server computers, known for its long history and extensive software ecosystem. ARM is now widely used in mobile devices and embedded systems due to its power efficiency, while RISC-V is an open-source ISA gaining popularity for its flexibility in academic research, embedded systems, and custom hardware development.

## 2.5 Input and Output Devices

I/O devices allow computers to interact with the external world, enabling data entry, display, communication, and storage. In general, there are two ways for the CPU to communicate with I/O devices:

- *Memory-mapped I/O*: I/O device registers or frame buffers are assigned specific addresses within the system' s memory space, allowing the CPU to use the same load and store instructions as it does for normal memory access to read from or write to these devices. Typical examples include VGA controllers (Figure 2.4a), disk controllers, and network interface cards.
- *Port-based I/O*: Also called isolated I/O, where I/O devices are accessed through a separate address space using special instructions (such as `in` and `out` on x86), which distinguish device operations from regular memory operations. Examples include legacy PC peripherals like the keyboard controller (Figure 2.4b) or some serial/parallel ports.




(a) Memory-mapped I/O.

(b) Step 2: Port-based I/O.

Figure 2.4: Two I/O modes.

Regardless of whether I/O devices are accessed through memory-mapped I/O or port-based I/O, a key question arises: when the CPU attempts to read data from a device, how does it know that the data is ready? Two common mechanisms are used to coordinate between the CPU and I/O devices:

- *Polling*: The CPU repeatedly checks a status register of the device until it indicates that data is ready. For example, a keyboard controller sets a "data ready" bit that the CPU must poll before reading the keystroke.
- *Interrupts*: The device sends an interrupt signal to the CPU when it is ready to be serviced, eliminating the need for constant polling. For instance, a network card generates an interrupt when a packet arrives, allowing the CPU to process it immediately.

For large data transfers, the CPU can rely on *Direct Memory Access (DMA)*, which allows the device to read from or write to main memory directly, bypassing the CPU. DMA is often combined with interrupts to notify the CPU when the transfer is complete. For example, a disk controller uses DMA to transfer disk blocks directly into RAM while the CPU continues executing other tasks.

## Excersize

1) The following code computes the largest Fibonacci number less than 100 and saves the result to the register `x0`, *i.e.,* 1+1=2, 1+2=3, 2+3=5, 3+5=8,..., 34+55=89. Modify the code to compute the largest accumulated value less than 100, starting from 1, *i.e.,* result = max(1+2+3+4+5+$\cdots$), s.t. result $<$ 100

```
_main:
    mov x0, #1
    mov x2, #1

_loop:
    mov x1, x0
    mov x0, x2
    add x2, x2, x1
    cmp x2, #100
    ble _loop
    ret
```

Code 2.4: ARM assembly code.