**DSC204A: Scaleable Data Systems Winter 2025**

4: Basics: Data Representation, Computer Organization, Processor, Memory and Storage

*Instructor: Umesh Bellur* *Scribe: Joel Polizzi, Yi Lien*

# 1 Recap

## 1.1 New Float Standards

Several new floating point standards have become present in recent years. New floating standards allow a developer to use a sliding scale to adjust the ratio of exponent bits and precision bits in a bit string. Our exponent value defines how large our integer can be, while our precision value defines how many decimal places we can store for the given value.

- **FP32**: In Floating Point 32, 1 bit is reserved as a sign bit. The Exponent will contain 8 bits, leaving the remaining 23 bits for the fractional precision.
- **BFloat16**: In BFloat16, the floating-point format consists of 1 sign bit, 8 exponent bits, and 7 fraction bits. Created by Google and is now used by companies such as Meta to sacrifice precision for range in order to obtain a smaller bit string to calculate. Google determined that in many cases, the precision is less important than the leading integer value (range), resulting in the BFloat16, 16-bit float standard.
- **FP16**: In Floating Point 16, its format consists of 1 sign bit, 5 exponent bits, and 10 fraction bits. In situations where we will not be working with large integers, FP16 can be leveraged to slide our focus from the range towards higher precision.

# 2 Floating Point Standards

## 2.1 Impacts of Floating Point Bit Depth

When examining the required bytes involved in storing a floating point, it can be determined that a 16-bit floating point value will require 2 bytes to store, whereas a 32-bit floating point value will require 4 bytes to store. This impacts two key measures in an operation:

- Storage
- Processing time

As noted above, a 32-bit floating point value will require twice the amount of storage as a 16-bit floating point value. If we are using a 32-bit encoding where it is not required, we may be doubling our storage requirement (and cost) to properly store our data. This may significantly affect the ROI of our storage platform, and improper use will lead to waste. The second impact is in the processing time involved for an operation to take place. An operation on a floating point is measured in FLOPS (floating point operation per second).

An example of the performance impacts between FP32 and FP64 bit operations can be seen in looking at the NVIDIA H100 (SXM) benchmarking sheet, which details that an FP64 form factor will calculate 34 teraFLOPS, while an FP32 form factor nearly doubles the amount of FLOPS that can be calculated at 67 teraFLOPS. There are several other and more efficient FP16 form factors, which can be leveraged to increase the throughput of FLOPS on the H100 (SXM) GPU. The key takeaway is that the floating-point standards that a developer may apply can have great impacts on the performance of an operation.

### 2.1.1 Limited precision

Due to representation imprecision issues, it is important to note that the Floating-Point arithmetic (carried out by the ALU) is *not* associative.

## 2.2 Digital Representations and Data Type Casting

Recall that everything in data is in bits and bytes, and digital objects are collections of various basic data types. Data types are then represented using different sequences, which can be used in different programming structures.

### 2.2.1 Type Casting

In some cases, it may be advantageous to type cast a variable from one type to another, essentially converting the value and storing it in a new variable. An example of this can be seen when taking a string of integers, such as the string type "54", and using a type cast to change the string into an integer number, 54. This can be useful for certain mathematical operations; however, type casting can come with some risk. For instance, when type casting the bit string "01101" to an integer type, the type cast would return the integer value 1101. This becomes ambiguous if the user is expecting the bit representation of 01101 instead of a direct integer representation.

In other cases, the type casting may fail altogether, such as the case of attempting to type cast an alphabetical string to an integer or float type. In this case an error would be thrown. Considering that not all types are cast-able, it is important for the developer to build in **Exception Handling** to account for possible casting errors.

### 2.2.2 Serialization and Deserialization

Considering when various data types, such as Lists, Dictionaries, or Arrays, are sent over a network or saved to a file, the structure is a binary array of 0s and 1s. To preserve the structure and content of these data types during transmission or storage, a process called serialization is used. Serialization converts a data structure into a sequence of bytes that can be easily stored or transmitted. Later, during deserialization, this byte sequence is reconstructed back into the original data structure.

### 2.2.3 Formats for Encoding Data

Data can also be encoded into various formats which allow for associations between keys and values to stored. The values are then retrievable by calling the keys. Encoding formats are particularly useful in providing **standard** methods of data search and retrieval. Several popular encoding formats include:

- JSON
- Markup Languages (XML, YAML, TOML, etc.)

Although different formats lend themselves to handling specific types of data, such as YAML being commonly used for system configurations, they all provide standardization and APIs to interface with the end-user. The standardization between the various formats allows for interoperability and a high level of efficiency and ease of use.

# 3 Foundation of Data Systems

## 3.1 Instruction Set Architecture (ISA)

The Instruction Set Architecture (ISA) defines the set of instructions that a computer processor can recognize and execute. It acts as the interface between software and hardware, playing a crucial role in converting high-level programming code into executable machine instructions. High-level programming languages, such as C and Java, are ultimately translated into a form that a computer can understand and execute.

Specifically, after we write code in a high-level language, it is compiled into assembly language, which functions similarly to Python in that it bridges the gap between the high-level code and the machine. Assembly language is a semi-human-readable language that enables communication with the processor. This code is then stored in its binary form, known as machine code, which the machine can later execute.

## 3.2 Basics of Processors

The most common way for a processor to execute machine code is load-store architecture. It means that for every instruction that runs on a CPU, there are two operations: load things from memory onto the processor and do operations on the processor, then store the result in the processor or memory.

### 3.2.1 The Von-Neumann Model (a Load-Store Computer)

Figure 1 shows the interactions between the CPU and memory. In the memory section, which is shown on the right side of the diagram, is a general memory block. It represents RAM, which stores both instructions and data. Note that the address `$3FF` shown at the bottom right represents a memory address in *hexadecimal* (base-16) format. It is a 12-bit number since each hexadecimal digit corresponds to 4 bits, and `3FF` contains 3 hex digits:

$$3 \text{ digits} \times 4 \text{ bits/digit} = 12 \text{ bits}$$

This indicates that the system uses **12-bit addressing**, allowing for a total of:

$$2^{12} = 4096$$

distinct memory addresses. Therefore, the addressable memory space contains **4096 bytes**, assuming 1 byte per address.

Regarding the CPU, here is a detailed explanation of its main components:

- **Instruction Handler**: Located at the bottom left of the diagram, it is responsible for fetching, decoding, and coordinating the execution of instructions. It consists of two main components: the **Instruction Register** and **Instruction Decoder**.

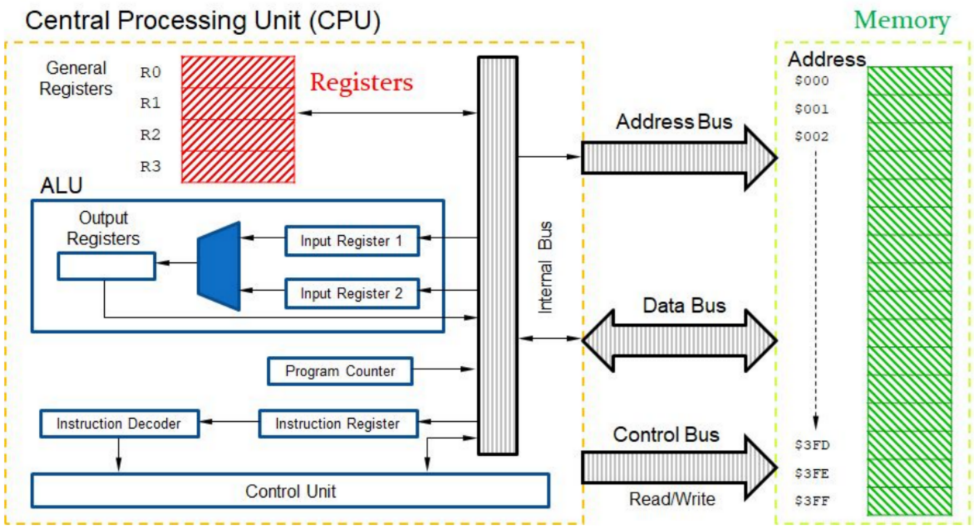


Figure 1: The interaction between the CPU and memory

- **The Arithmetic Logic Unit (ALU)**: The ALU is the core computational component of the CPU that performs arithmetic and logical operations.
- **Registers**: These are small, fast memory locations within the processor. They store temporary data and intermediate results, allowing operations to run efficiently without constant access to the main memory.
- **Internal Bus**: Serves as an interconnection mechanism linking the instruction handler, ALU, and registers to facilitate smooth data and control signal exchanges.
- **Control Unit**: Manages CPU operations by sending signals to various components, instructing them when and how to perform tasks.

### 3.2.2 A simple Processor Design

Figure 2 provides a more detailed and granular view of the processor design.

When the program starts, the Program Counter (PC) on the left side of the diagram holds the address of the first instruction in memory, which is sent out by the CPU onto the address bus. The PC shows where the program currently is and which instruction is being executed. It points to a memory location where the instruction is stored.

The CPU sends this address out, requests a read, and the instruction is fetched from memory. The instruction is then placed into a small register called IF/ID (shown in the figure).

Next, the instruction is decoded. The decoder sees the instruction and looks at its bit pattern to figure out what kind of instruction it is. The decoder helps the CPU understand what needs to be done.

Once the instruction is understood, the required values are placed into the correct registers. For example, if the instruction is to add two numbers, the first register will hold the first number, the second register will hold the second number, and a control signal will be set to perform the addition.

In short, after the instruction is fetched, it is decoded and then executed by the ALU for computation. The result is then stored temporarily in an internal register.

Note that there is a 4 on the left-hand side of the diagram, which indicates that an instruction occupies one word, and a word typically consists of 4 bytes on a 32-bit machine. On a 64-bit machine, a word would typically be 8 bytes instead.[1]

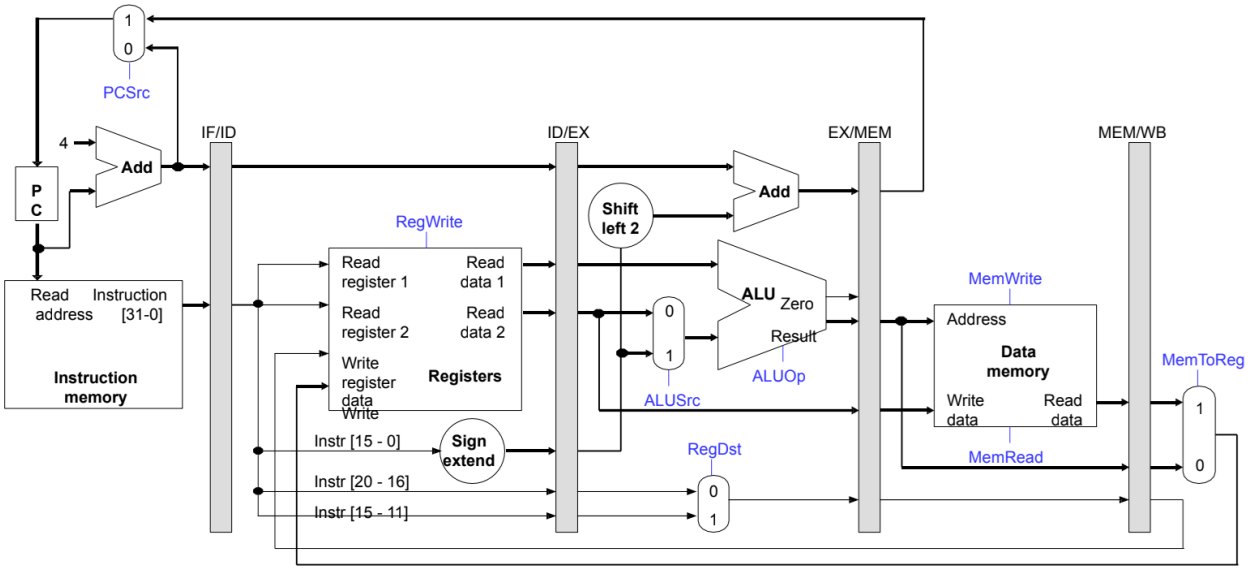


Figure 2: Simple processor design

### 3.2.3 Executing an Instruction

There are five operations in an instruction:

- Instruction Fetch (IF): Read an instruction from memory
- Instruction Decode (ID): Read source registers and generate control signal
- Execute (EX): Compute an R-type result or a branch outcome
- Memory (MEM): Read or write the data memory
- Writeback (WB): Store a result in the destination register

## 3.3 CPU interacting with Memory

### 3.3.1 Memory Read Instructions

To see how CPU and Memory interact with each other, we can see from the example of a read operation:

[1] A 64-bit machine means there are $2^{64}$ addressable locations, allowing for much larger memory addressing—typically supporting up to 16 exabytes of addressable space, though practical limits (like 8 GB of RAM) depend on the system.

- The Control Unit instructs the Memory to perform a read.
- The Address Bus carries the address of the required memory location.
- The Memory places the data from that address onto the Data Bus.
- The CPU then loads this data into one of its registers.

### 3.3.2 Memory Write Instructions

In a write operation:

- The CPU places the target memory address on the Address Bus.
- It sends the data (from a register) onto the Data Bus.
- The Control Unit signals memory to write that data to the specified address.

Note that if the processor is performing a computation instead of writing to memory, the ALU is used. It typically takes two input values from registers, performs an operation like addition or subtraction, and stores the result back into a register via its output register.

## 3.4 A Register type instruction on the MIPS processor

The following section explains how Register-type (R-type) instructions are formatted and interpreted on the MIPS processor.

R-type (Register-type) instructions involve operations between registers, such as add, sub, and, or, etc. R-type instructions are 32 bits long and broken into fields:

| Field | Bits | Meaning |
|---|---|---|
| opcode | 6 | Operation code (for R-type, always 000000) |
| rs | 5 | First source register |
| rt | 5 | Second source register |
| rd | 5 | Destination register |
| shamt | 5 | Shift amount (used for shift operations only) |
| funct | 6 | Function code: specifies the exact operation (e.g., add, sub) |

Table 1: Breakdown of R-type Format

shamt (Shift Amount) is only used in shift operations like sll (shift left logical) or srl (shift right logical). It tells how many bits to shift. In the following example, as there is no shift instruction, we set this to 0.

In addition, funct is used to specify which exact R-type operation to perform. For 'add', the funct code is 100000.

In the example of 'add $t0, $t1, $t2', the instruction is translated into '$t0 = $t1 + $t2'. Based on the binary representation, as illustrated in Table 2, Register Numbers become $t0 = 8, $t1 = 9, $t2 = 10.

So, the fields become:

| Field | Binary | Meaning |
|---|---|---|
| opcode | 000000 | R-type instruction |
| rs | 01001 | Source register ($t1 = 9) |
| rt | 01010 | Source register ($t2 = 10) |
| rd | 01000 | Destination ($t0 =8) |
| shamt | 00000 | No shift (unused here) |
| funct | 100000 | Code for 'add' operation |

Table 2: Breakdown of R-type Format

## 3.5 Processor Performance

Processor performance refers to how fast a processor can execute a program, which is largely determined by clock cycles. Each instruction takes a certain number of clock cycles to execute, as defined by the Instruction Set Architecture (ISA). The processor's clock rate, typically measured in gigahertz (GHz), tells us how many cycles occur per second. For example, a 1 GHz processor performs 1 billion cycles per second. If an instruction takes two or three cycles to execute—depending on its complexity—then the processor can ideally complete hundreds of millions of instructions per second. In the best-case scenario, if all instructions were single-cycle, the processor could execute 1 billion instructions per second.

However, in practice, the processor is much faster than the rest of the system around it, especially memory. Most programs do not keep the CPU fully busy because memory access commands can stall the processor. When the CPU waits for data to be transferred between memory and registers, the Arithmetic & Logic Unit (ALU) and Control Unit are often idle. Accessing data from memory can take 10 to 15 cycles, or even longer if the data is not in DRAM and must be fetched from disk. During this time, the CPU is doing nothing, and performance suffers significantly.

As a result, the actual execution time of a program can be orders of magnitude worse than what the raw clock rate suggests. To minimize this gap, it is essential to write programs that maximize the use of the processor's internal resources—particularly the cache, which provides much faster access than the main memory.

**Key Principle**: Optimizing access to main memory and effective use of the processor's cache are critical for maximizing processor performance.