Chapter 7 | Computer Architecture

• Processor • Control Unit • Registers • Arithmetic Logic Unit • Status Register • Memory Unit • Datapath • Fetch-Decode-Execute-Writeback • Internal Clock • I/O Devices

CS102 Computer Environment

A computer system consists of three main components:

- A processor or a **C**entral **P**rocessing **U**nit (CPU)
- A memory unit
- Input / output (I/O) devices

An interconnection network, also known as the system bus, allows the communication between those three components. The system bus is formed from wires and through those wires passes electrical signals. We have three main busses that connect the three main components of a computer system, which are:

- An address bus
- A data bus
- A control bus

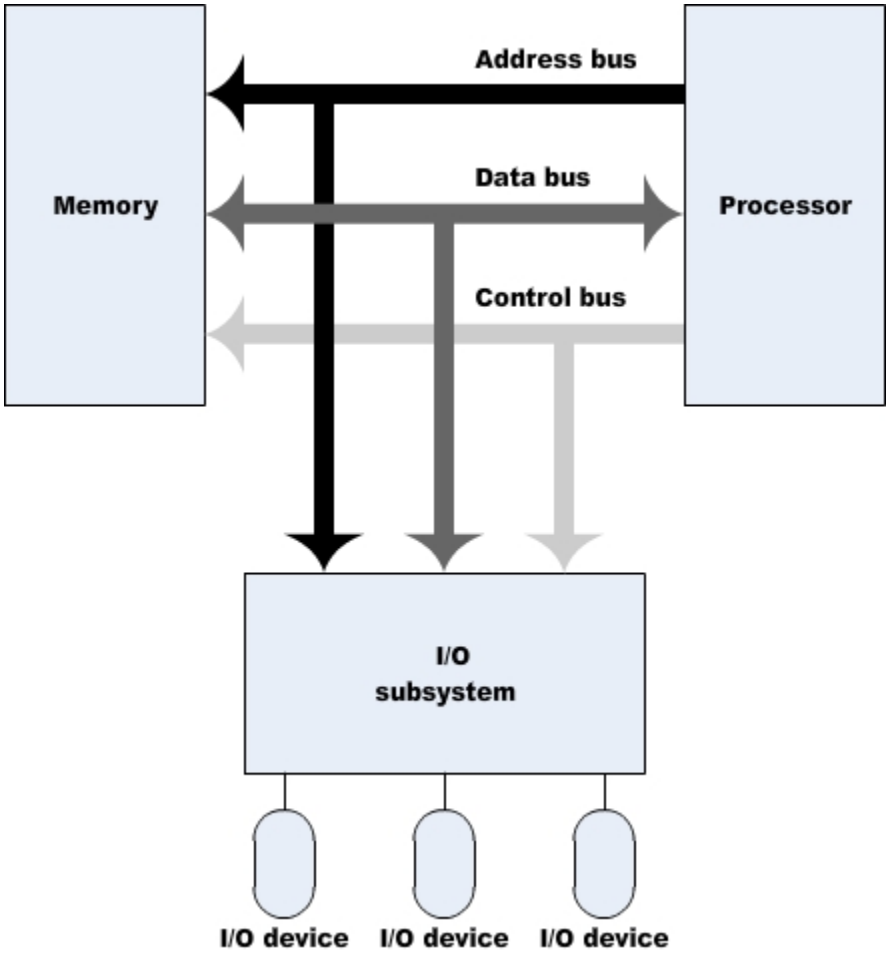

Figure 1: Simple diagram of a computer system

The width of the address bus determines the physical memory size that is addressable by the processor. For example the Pentium processor has 32 address lines, meaning that it can address up to 2^{32} (equal to 4GB) of memory.

The width of the data bus indicates the size of data transferred between processor and memory or I/O device. A Pentium processor has 64 data lines meaning that the data bus can move 64 bits of data.

As for the control bus consists of a set of control signals that indicates the type of action that is taking place on the system bus. Those signals could be memory read or write, an interrupt, an I/O read or write, bus request…

For example, when data needs to be written on the memory, a memory write signal will be generated. Same thing if we are reading from an I/O device, an I/O read signal will be generated.

Processor

The processor consists of three main components:

- A control unit
- Registers
- And an **A**rithmetic **L**ogic **U**nit (ALU) or more

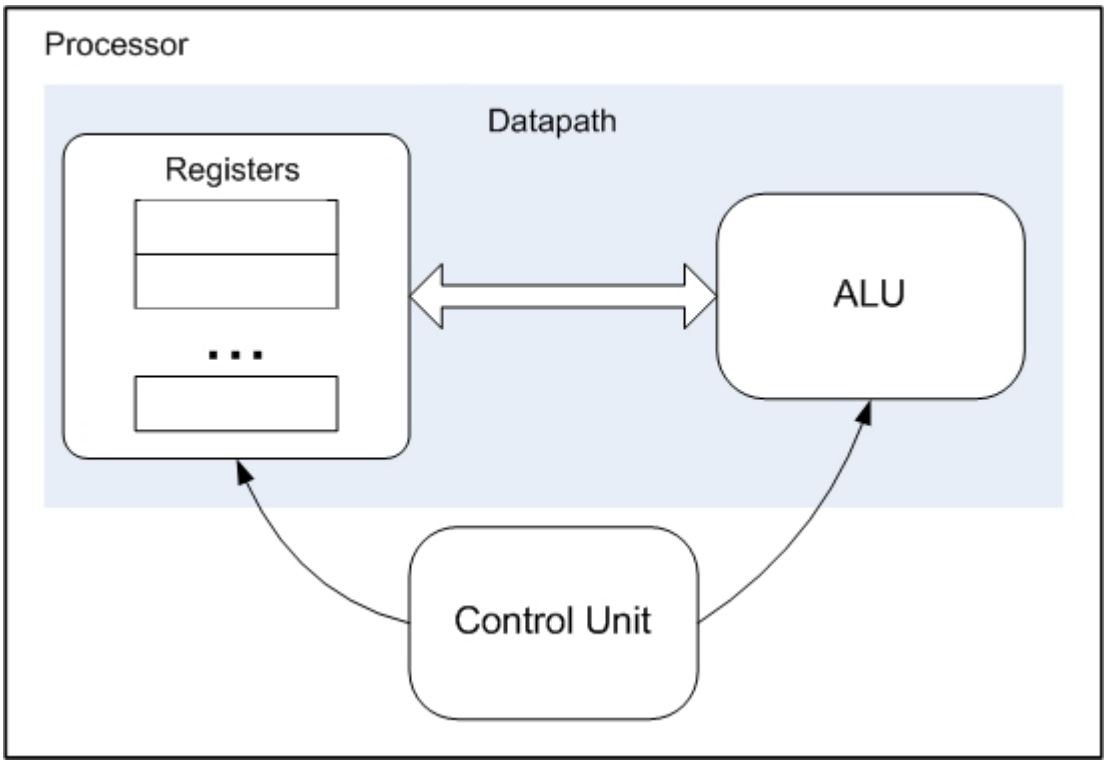

Figure 2: The three components of a processor

Control Unit

The control unit acts like the processor's supervisor or the maestro, meaning it makes sure that everything is running smoothly and in order. It directly controls the operation of the processor by fetching instructions from the main memory and decoding them to find the type of instructions.

Registers

Registers are special high-speed storage areas that hold data and instructions temporarily during processing. They are part of the CPU instead of in main memory; their contents therefore can be handled much faster than can the contents of main memory.

Obviously we would like to have as many registers as possible to make the processing time much faster but this kind of processors would be much more expensive. Note that programmers can have access to most processor registers.

There are several types of registers and some of them contain special values:

- **General Purpose Registers(GPR)**

  GPRs are registers that can be used to hold data for the purpose of computation. For example, if we have a statement in C like " `a = b+c+z;`", where `a`, `b`, `c` and `z` are locations in memory. Usually, the ALU can only perform additions for 2 values at a time. That means that the intermediate results of `b+c` needs to be saved somewhere. The usual practice is to use the GPRs instead of memory locations to store these intermediate results for quick access purposes.

- **Program Counter(PC)**

  The program counter register, points at the next instruction to be executed by the CPU. Initially, the program counter (PC) is set to the address of the first instructions in the program. The first instruction fetched from the program memory is placed in the instruction register (IR). After that, the instruction decoder interprets the instruction from the IR and executes it.

- **Instruction Register (IR)**

  The instruction register contains the next instruction to be executed by the arithmetic logic unit (ALU) through the instruction decoder.

- **Memory Address Register (MAR)**

  The memory address register (MAR) holds the address of the memory where data should be stored.

- **Memory Data Register (MDR)**

  The memory data register (MDR) stores the results of an instruction.

Arithmetic Logic Unit

The Arithmetic logical unit (ALU) performs two types of operations: arithmetic, and logical operations. The arithmetic part consists of an addition of two binary digits, as for the logical part handles logical operations like *and*-ing two input operands.

The ALU has input buses, through them data from register set are fed. Now the ALU's control input determines the operation that needs to be done on the input operands. After doing the right operations an output is generated. This output could be placed back into one of the registers via the internal data bus or can be written in the main memory. If the result is to be written in the main memory, then the ALU output should be stored in the MDR where the value of the MDR is written at the memory address in MAR.

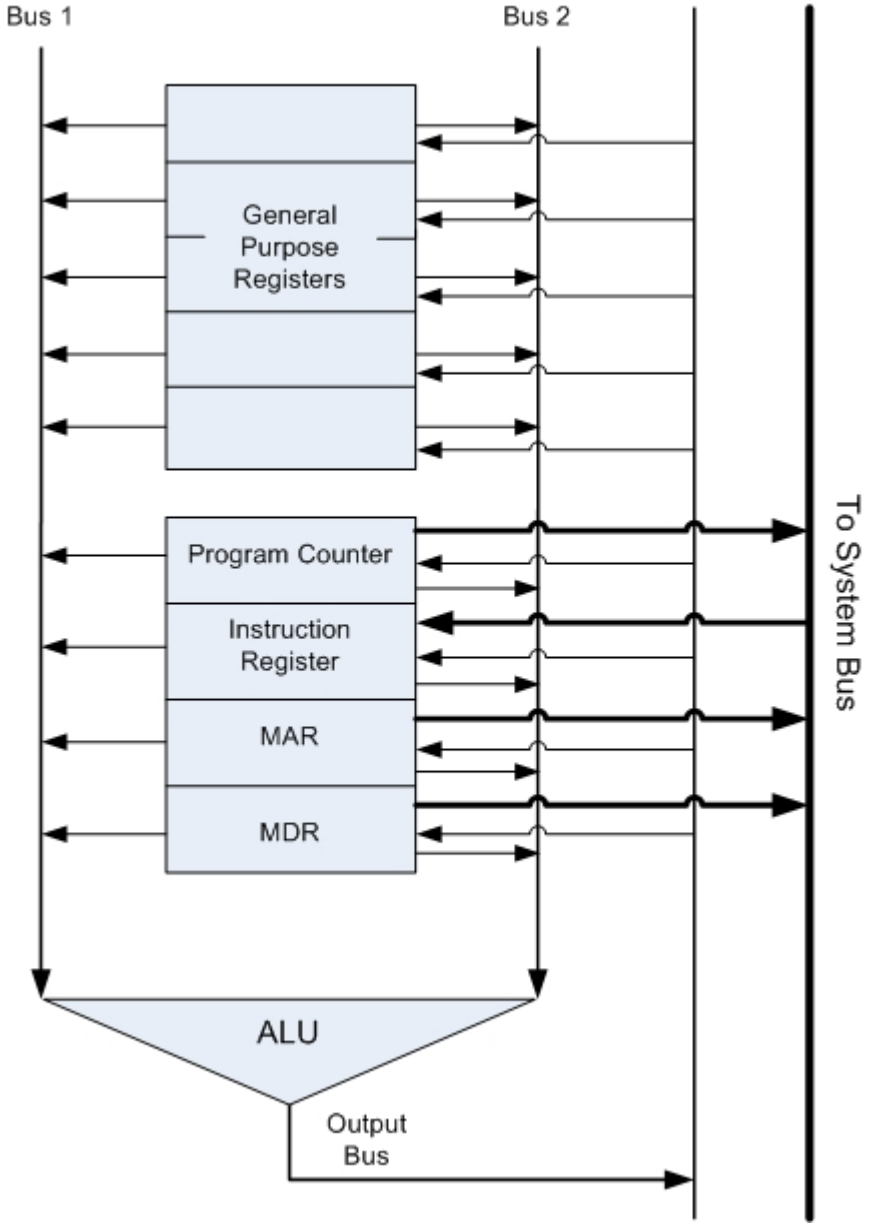

Figure 3: The processor components interconnected using buses (the datapath)

Instructions performed in the ALU can be divided into two categories depending on the location of operands:

- **Register-Register** or **Register-Memory** instructions allow memory words to be fetched into registers, where they can be used as ALU inputs, perform some operations on them, and then store the result back in a register.
- A **Memory-Memory** instruction fetches its operands from the memory into the ALU inputs registers, performs its operation, and then writes the result back into memory.

The ALU contains also a status register, but what does it do?

Status Register

Status register also referred to as the condition code or flag, is used to make decisions depending on the value of flags. Those flags are set depending on the result of the last operation.

The most important status register flags are:

- Zero flag: ZF
- Overflow flag: OF
- Sign flag: SF
- Carry flag: CF

Memory Unit

The computer's memory unit, also known as the main or primary memory or random-access memory (RAM), stores all the program instructions and data. Its main advantage is that it could be accessed directly from the processor.

The main memory is usually too small to store all programs and data permanently, that's why it is considered as a volatile storage device meaning that it loses all its content when power is turned off.

Since we have a primary memory, a secondary memory should exist and that is an extension to main memory. Its job is to store large quantities of data permanently (non-volatile) but with this power comes a disadvantage, which is a slow access to data unlike the main memory that is really fast to access.

Datapath

We are now in a position to describe the datapath using Figure 4 below as an example. As mentioned in chapter 1, every instruction that is run in the computer has to go through the stages of Fetch, Execute, Decode and Writeback. All of these stages are done within the datapath. In fact, we have already described most of the components in the datapath so far in this chapter already. We just need to describe the decoder, the MUX and the register files before describing how the datapath works altogether.

- ## Multiplexer(MUX)

  The Multiplexer is a particular piece of logic that selects one of the many input signals and forwards it as the output. The way the selection occur is by way of having additional select lines that informs the multiplexer which input signal to use as output.

- ## Register files

  Basically, the register files is the collection of registers (usually both General Purpose and Floating point) that can be accessed by programmers.

- ## Decoder

  As we have already indicated so far, the only thing understood in the computer are 0s and 1s. So, even the instructions that form our programs are encoded in 1s and 0s. These instructions need to be decoded before execution can be carried out. Usually, the encoding for the instructions are already determined during design time. In particular, the decoder needs to do 2 things:

  1. What ***operation*** need to be performed

     We shall be covering this in more detail in chapter 9 when we discuss assembly code. For now, it is sufficient to know that there are basically 2 kinds of instructions. Arithmetic instructions include operations such as addition, subtraction, multiplication etc while memory operations are only either load or store. Load is reading data from a particular address location in the memory to a register while store is the reverse

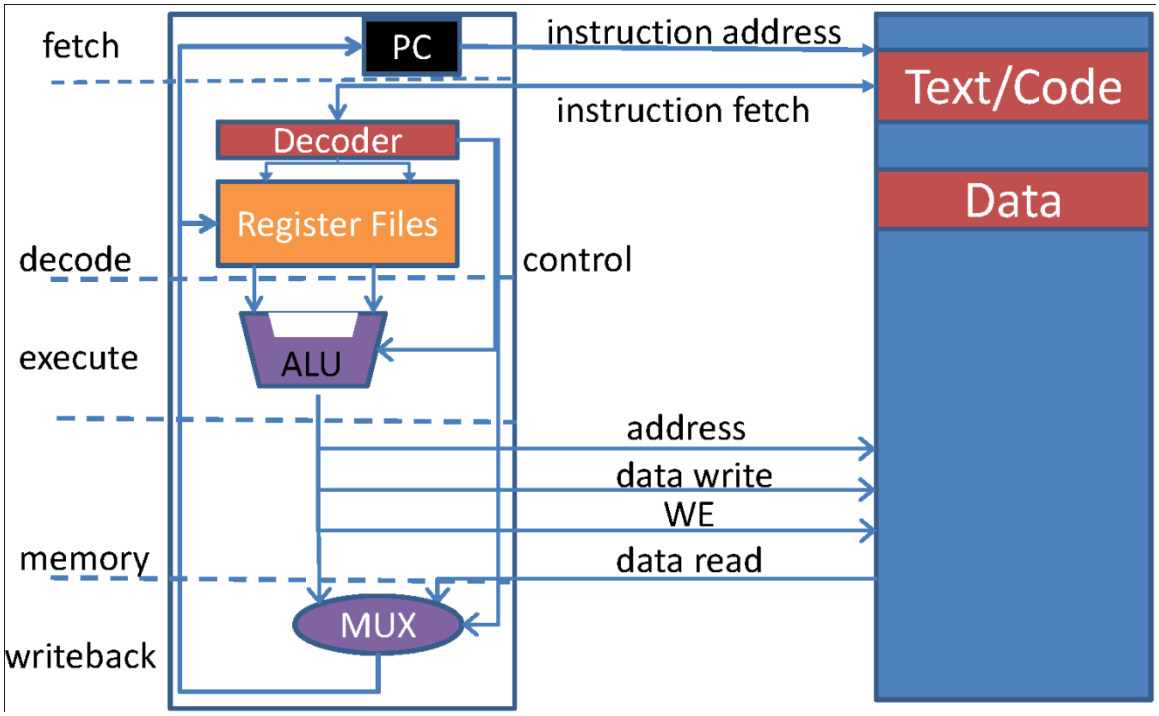

Figure 4: An example datapath

2. What ***operands*** for the operations.

Operands are parameters needed for the operations. We may want to add two registers, or multiply a register with a value stored in the memory. The instruction encoding will inform the decoder where to get the values needed for the operations.

Fetch-Decode-Execute-Writeback

We now describe how each stage works using figure 4 as an illustration.

- **Fetch Stage**

  The value stored in the PC is the address of the instruction we want to fetch. That instruction is fetched into the instruction register which is an input into the decoder.

- **Decode Stage**

  The decoder extracts the following from the instruction stored in the instruction register – Operands involved for the current operation, What the ALU needs to do, and the desired effect of the operation.

- **Execute Stage**

  The ALU performs the operation based on the inputs given from the register file and the decoder. The control line of the decoder indicates the desired operation (i.e., whether addition. Subtraction etc) while the input from the register files are the operands of the operation. As we can see from figure 4, we only support arithmetic operations between registers in this datapath. A datapath that supports direct arithmetic operations on locations in main memory and register will be more complex.

- **Writeback Stage**

  Again, the input from the decoder determines what happens during this stage. Depending on the instruction that is decoded, the result of the ALU may be written back into memory, a value may be loaded onto the register from the memory or the result of the ALU may be written back into the register file immediately.

Internal Clock

Each CPU has an internal clock where all the processor actions are synchronized according to the clock input. The clock signal consists of clock cycles (every clock cycle is called the clock period) that are generally represented by a square wave.

Now in order to specify the clock speed or cycle, we use the clock frequency that is calculated using the inverse of the clock period. We measure the clock frequency in Hz (Hertz) and it represents one cycle/second.

For example a 2.00GHz Pentium processor has $2x10^9$ (2,000,000,000) clock cycles per second.

Term	Decimal (base 10)
K (kilo)	10^3
M (mega)	10^6
G (giga)	10^9
T (tera)	10^{12}
P (peta)	10^{15}

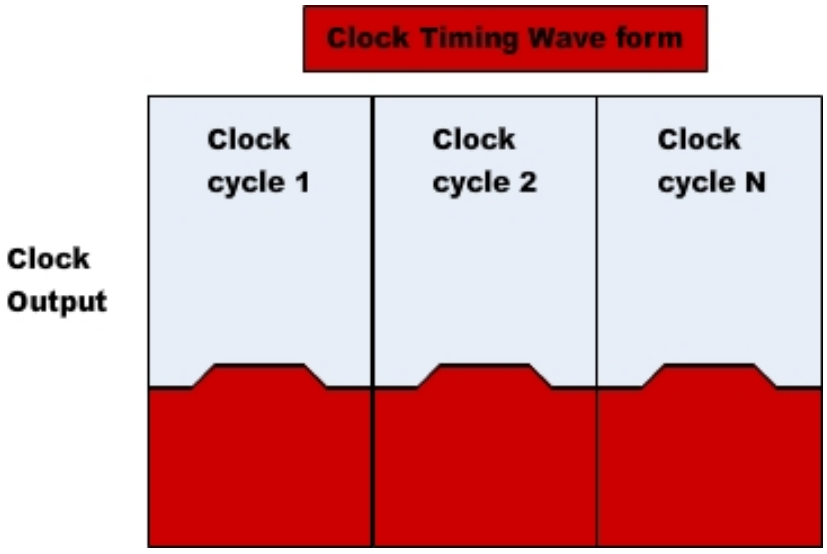

Figure 5: Clock Timing Wave Form

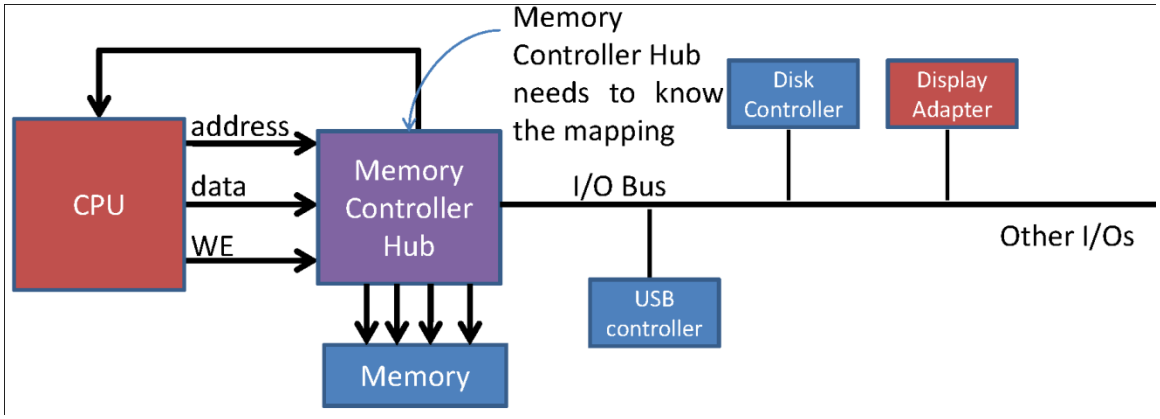

Figure 6: I/O Bus

I/O Devices

I/O devices are connected to the CPU through a bus. Basically, we need the following:

- The CPU to initiate communication with the I/O devices
  - Special Instructions

    Besides arithmetic and memory instructions, we include special instructions for interacting with I/O. This is the approach taken later for the virtual machine detailed in chapter 9.
  - Memory Mapped Instructions

    In this setup, not all addresses are mapped to the RAM directly. Some parts may correspond to specific I/Os. This mapping is dependent on the motherboard and the computer architecture. This is achieved through using a memory controller hub which is an arbiter on the I/O bus. The memory controller hub is analogous to a postman who delivers letters to the correct household. Thus when the addresses are those mapped to the RAM, the RAM is accessed. Otherwise, the information is passed onto the I/O bus for the specific I/O device. This way, from the CPU point of view, communication with I/O is just like reading or writing to memory.
- I/O devices to inform CPU of events happening
  - Polling

    The CPU executes instructions to keep on checking whether a desired I/O operation is done or not. Undesirable because it wastes CPU cycles that can be used to do other useful tasks.
  - Interrupt

    Enable the I/O devices to signal the CPU electronically when a task is completed. The advantage of this approach is that we have increased CPU utilization. The downside of this, however, is that complex procedures need to be put in place in order to handle interrupts. The details of interrupt handling are covered in CS180.