Hardware

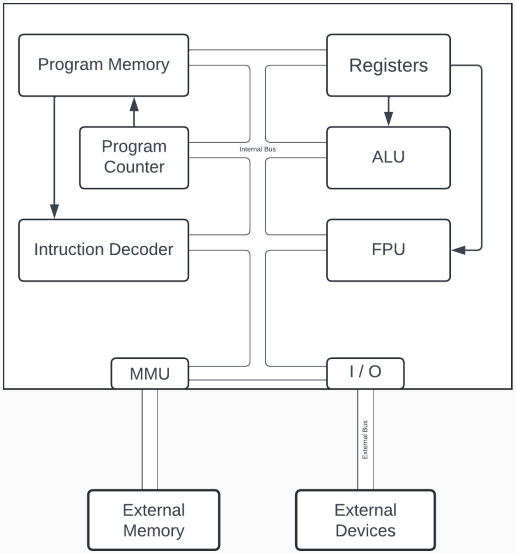
CPU Insight

Jean-Malo Meichel

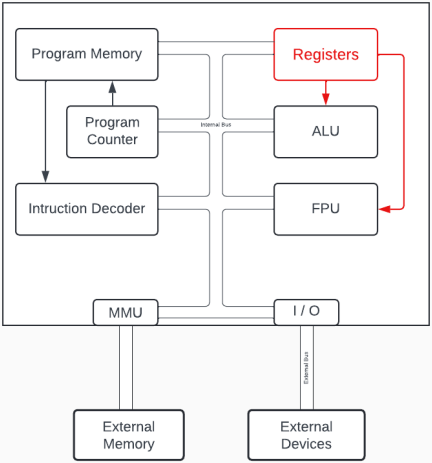
September 2024

Remember that CPU Schematic from the first class ?

CPU Overview



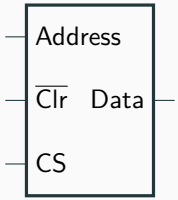
CPU Overview



Let's analyze each of these components

- Registers

Registers

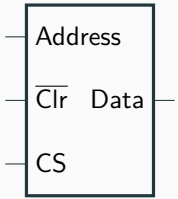


Registers are small data holders.

They are very fast but very small compared to other memory devices.

They are built inside CPU dies.

Registers

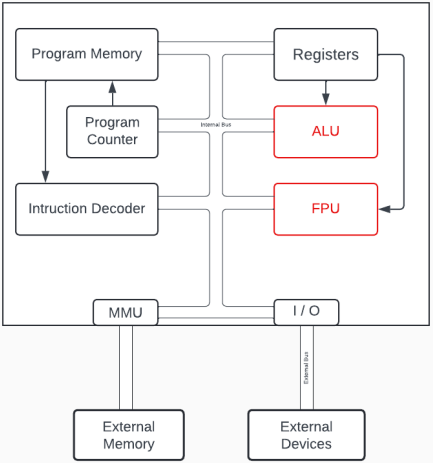


Registers can be general purposed or specialized.

Arithmetic and logic instructions use them directly to manipulate data.

You already saw a register in the last logic tutorial.

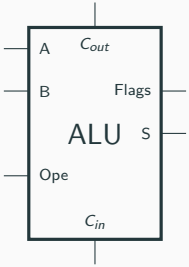
CPU Overview



Let's analyze each of these components

- Registers
- ALU/FPU

ALU/FPU



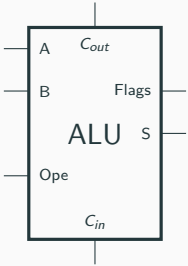
ALU stands for “Arithmetic and Logic Unit”.

FPU stands for “Floating Point Unit”.

They are responsible for performing all arithmetic and logic calculations for integers and floats, respectively.

Generally, the ALU is a pure combinatorial logic circuit that performs calculations depending on the operation asked by the CPU.

ALU/FPU

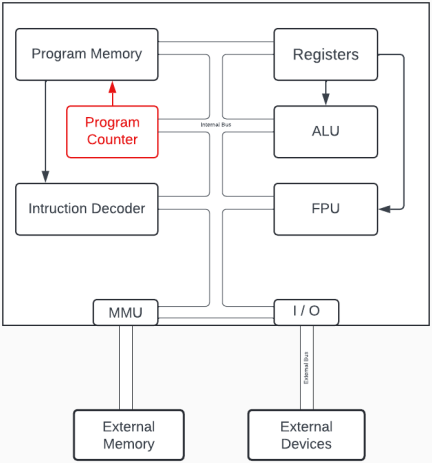


The ALU is also responsible for managing flags.

Flags are a way to monitor operation results.

Most common flags are: N, Z, V and C.

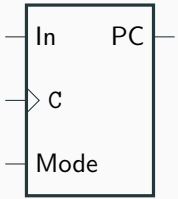
CPU Overview



Let's analyze each of these components

- Registers
- ALU/FPU
- Program Counter

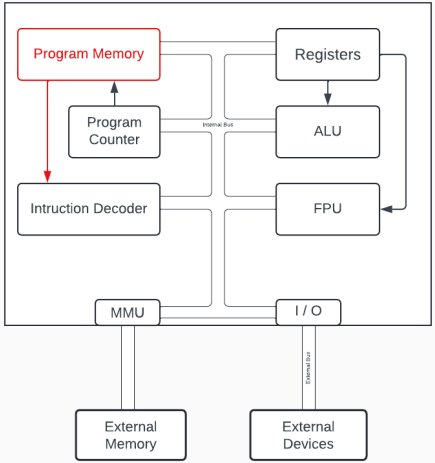
Program Counter



The program counter is a mix of a counter and a register.
Its role is to provide the current instruction address.
Its value can change in different ways:

- Automatic Increment
- Absolute value
- Relative Increment/Decrement

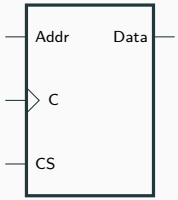
CPU Overview



Let's analyze each of these components

- Registers
- ALU/FPU
- Program Counter
- Program Memory

Program Memory



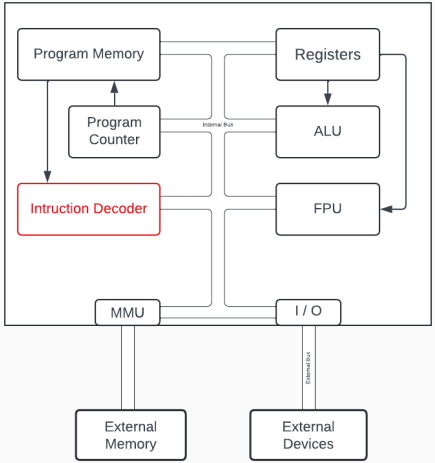
The program memory contains the bytecode the CPU must execute.

It can be internal or external to the CPU.

It is addressed directly by the program counter.

Fetched instructions are sent directly to the instruction decoder.

CPU Overview



Let's analyze each of these components

- Registers
- ALU/FPU
- Program Counter
- Program Memory
- Instruction Decoder

Instruction Decoder

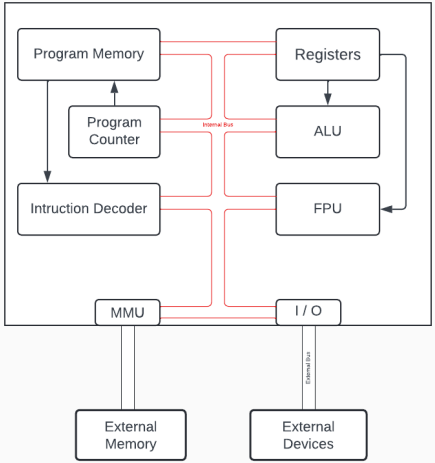
It receives the instructions from the program memory.

It converts each instruction to control signals for all the CPU components.

It is the “Maestro” of the CPU.

Instructions are binary data of fixed or variable size depending on the architecture.

CPU Overview



Let's analyze each of these components

- Registers
- ALU/FPU
- Program Counter
- Program Memory
- Instruction Decoder
- Bus

Bus

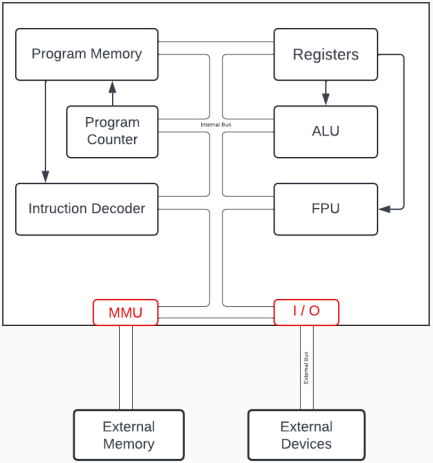
A ***bus*** is a ***communication mean*** between CPU components.

It is a collection of physical ***wires*** that transmit parallel data.

Multiple components can be connected to a single bus as long as only ***one*** of them writes data on it at ***any*** time.

Different busses coexist inside and outside of a CPU (Data bus, address bus, control bus etc...).

CPU Overview



Let's analyze each of these components

- Registers
- ALU/FPU
- Program Counter
- Program Memory
- Instruction Decoder
- Bus
- Inputs/Outputs

Inputs/Outputs

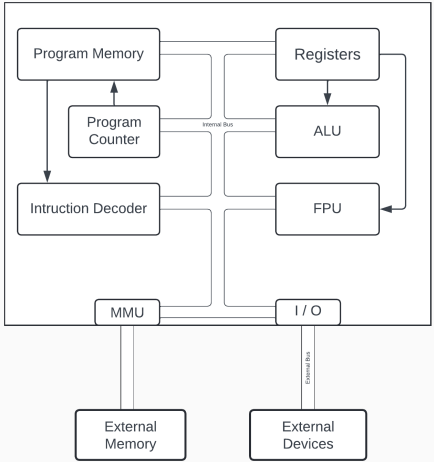
A CPU alone isn't enough to perform complex tasks.

It needs to communicate with external devices.

Busses are used to communicate with theses devices.

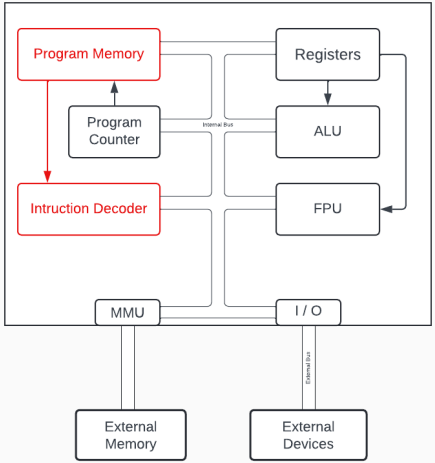
On modern systems, memory is mainly external to the CPU. A component is used to manage this memory: the MMU.

Instruction Execution Example



Example: ADD D0,D1

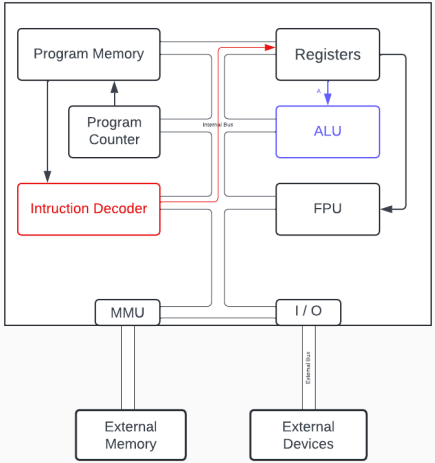
Instruction Execution Example



Example: ADD D0,D1

- The instruction is fetched from the Program Memory

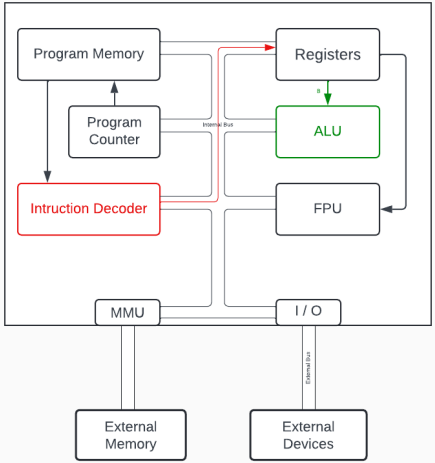
Instruction Execution Example



Example: ADD D0,D1

- The instruction is fetched from the Program Memory
- The instruction decoder tells D0 to write on bus A

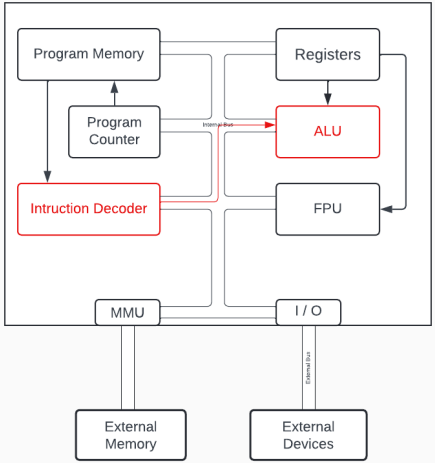
Instruction Execution Example



Example: ADD D0,D1

- The instruction is fetched from the Program Memory
- The instruction decoder tells D0 to write on bus A
- It tells D1 to write on bus B

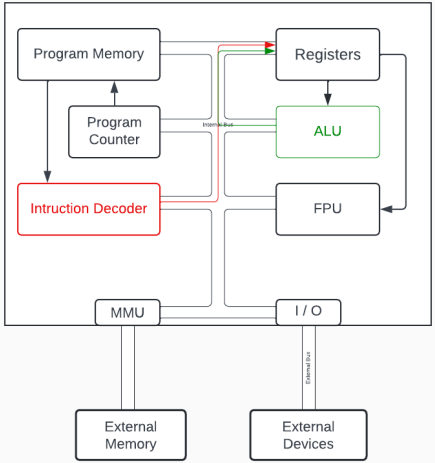
Instruction Execution Example



Example: ADD D0,D1

- The instruction is fetched from the Program Memory
- The instruction decoder tells D0 to write on bus A
- It tells D1 to write on bus B
- It tells the ALU to compute an addition

Instruction Execution Example



Example: ADD D0,D1

- The instruction is fetched from the Program Memory
- The instruction decoder tells D0 to write on bus A
- It tells D1 to write on bus B
- It tells the ALU to compute an addition
- It tells D1 to read the data bus

Program Execution

Program execution: Memory

Programs are stored in memory.

Either ROM or RAM depending on the system.

The memory can be included in a microcontroller chip or be external.

Program execution: Instructions

A program is nothing more than a sequence of instructions.

An instruction is a numeric value that has a special meaning for a processor.

Each processor family has its own instruction set.

An instruction can be followed by some arguments that further specify what the processor needs to do.

Instructions

Instructions: Standard Instruction Types

All instruction sets are different in some ways, but most of them implements common types of instructions:

- Arithmetic & Logic instructions
- Conditional jump instructions
- Function instructions
- CPU control instructions

Instructions: Arithmetic & Logic Instructions

Arithmetic & Logic instructions are all the instructions that can compute the result of a given mathematical or logical operation.

Examples:

- Add numbers
- Substract numbers
- Perform a logical OR on two numbers
- Shift a binary value
- Compare two values
- etc...

Instructions: Arithmetic & Logic Instructions

Most of the arithmetic & logic instructions are using the registers of the CPU as data source.

The registers are small memory slots that are directly included in the CPU. They are the fastest way of storing data but they are also very limited in space.

Most registers can only save one value at a time.

Instructions: Conditionnal jump instructions

Conditional jump instructions can change the execution path of a program depending on a previous operation.

The “if/then/else” blocks of higher level programming languages are built on them.

Instructions: Conditionnal jump instructions

Conditional jumps are built on the concept of flags.

Flags are boolean values that are updated by other instructions (mostly arithmetic & logic ones). They give information about the result of previous operations.

Most common flags are:

- **Z**ero flag
- **N**egative flag
- **C**arry flag
- O**V**erflow flag

Jump instructions checks the values of the flags and jumps to another part of the program if their conditions are met.

Instructions: Function instructions

Function instructions are similar to jump instructions.

They can change the execution path of a program by jumping to other parts of the code.

The difference is they are able to jump back to the previous location after running the code they jumped to.

Instructions: Function instructions

The most common function instructions are “call” and “return”.

A call is a jump to another part of the program. The difference with a simple jump is that the location of the call is saved to be able to jump back to it later.

A return is also a jump. It is able to read the last saved location and jump back to it.

The combination of theses two instructions is used to create functions: Small pieces of code that can be called whenever they are needed.

The function instructions are built on the stack concept, which we will explore later.

Instructions: CPU control instructions

CPU control instructions are meaned to control different functionnalities of the CPU.

They are very specific to the CPU model and instruction set.

Examples:

- Put the CPU in standby mode
- Reset the CPU
- Change the access level of the CPU
- etc...

The Stack

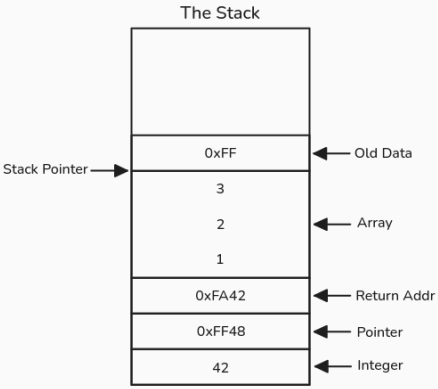
The Stack

The ***stack*** is a fundamental concept in almost all programming languages.

Languages with functions or routines = stack involved.

A stack is a place in ***memory***. It is organized as a ***LIFO***.

The Stack



To manage the stack, the address of its ***last element*** needs to be stored.

It is called the ***stack pointer***.

Usually the stack begins at the very end of the memory and grows up.

The Stack

Function instructions use the stack to store their ***return location***.

The return instruction retreive this address from the stack to ***jump back***.

It is also used to ***save*** the values of the registers before calling a function.

Higher level programming languages usually store ***local variables*** on the stack (Ex: C, C++).

Dynamically allocated variable are ***NOT*** stored on the stack.