Introduction

A **microprocessor**, sometimes called a logic chip, is a computer processor on a microchip. The microprocessor contains all, or most of, the central processing unit (CPU) functions.
To understand the microprocessor, we first need to understand the need of a processing unit.

Electronics, is basically about controlling the flow of electrons in order to build a useful **automatic** system like for example: a water-level controlled pump, a fan that automatically turns on at a certain high temperature and turns off at a certain low or normal temperature, a Bank ATM, a device that can send messages through the internet (a phone, a server, a PC). In either of these systems there appear to be an **input** system, an **output** one and a **processing or control** system.
In the case of the fan, the input device would be a temperature sensor, the output a motor controlling the fan, and the control system can be a circuitry between the sensor and the motor. That control system is often a combination of transistors.
A **transistor** performs the basic operation we are aiming: when it is supplied by a certain minimum current or voltage, it allows current to flow(or voltage to appear) in another branch (at another node) of the circuit.-----*Transistor circuit, Ic is determined by Ib(To Do)-----.* We can use a group of such transistors to perform the fan behaviour or the tank level pump stuff.
But due to the **analog** character of the transistor, combining many of it can quickly get messy and its behavior extremely hard to predict. Also, for many purposes, analog system might be unnecessary complicated, and their resistance to noise not good. We then choose to use a digital system.

In a **digital system**, the current or voltage can only take some **FINITE** values and at some **DISCRETE** time. We are no longer interested in the continuous states of our input. We only measure some finite values at discrete times. In the case of the ATM, for example, a user input is better represented by a digital circuit : The user either press a button, or he doesn't. There is a finite number of possibles inputs (buttons). Each button sends an input signal to the control/processing circuit. That circuit is a combination of logic gates controlling the output devices (screen, cash dispenser..). The logic gates perform logical operations like AND, OR, NOT on binary inputs. We can then combine those components to produce the desired output. That processor is called **random logic**. That system is easier to design than its analog equivalent.
A more advanced kind of digital system, is a programmable digital system. In this type of system, there is not a logic circuit specially designed to perform one particular task, rather, there is a logic circuit capable of performing a set of basic operations **that can be further combined** to achieve a wide range of tasks. For example, in a phone, there is no special circuit for whatsapp and another special for youtube. And there is not a special circuit to play a music and another to write a document. They all use the same circuitry, but in different ways(combinations and sequence). Their way of combining the phone's basic available operations is **the software** or **program.** The software is a sequence of instructions that are performed by a computer in the order in which they are defined in the software. (An attentive reader might notice here the need

of a control unit in the computer, capable of controlling the order in which the operations will be executed.). This type of system is made of three main components:

- The **ALU** that contains the logical and arithmetic operations that will be needed by the different software.
- Storage elements called **registers** that can store results of operations that will be needed in the future.
- And a **control unit** that controls the order of execution of the operations(ALU operations, data transfer in memories, I/O operations…) as specified by the program.

In conclusion, we need a CPU in order to be able to use the same circuitry for different purposes, simply by changing the software.

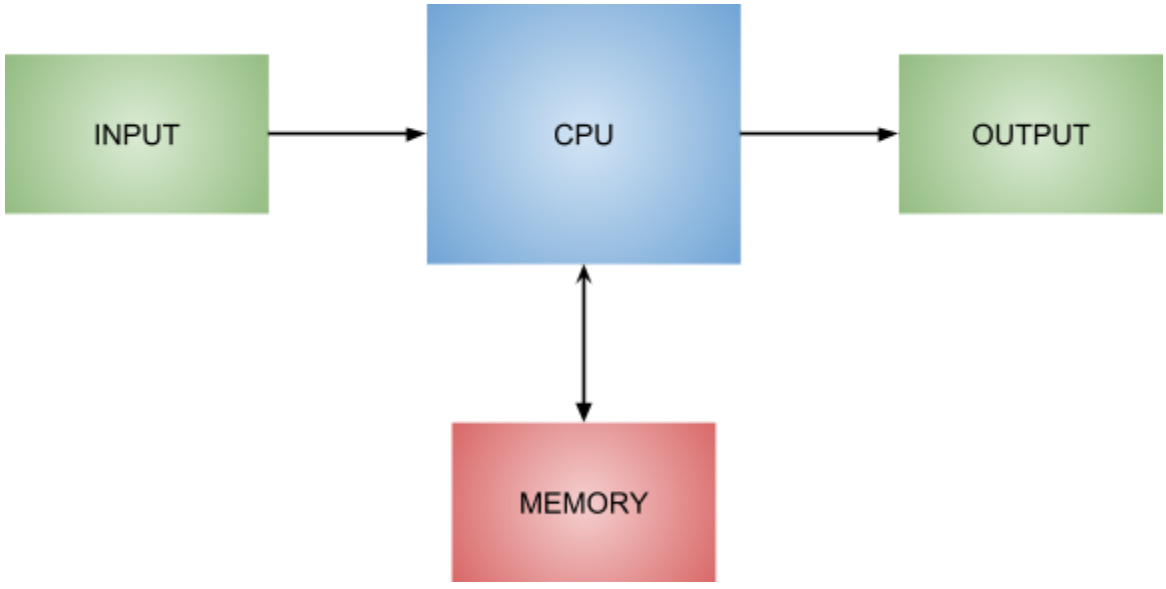
NB: In modern processors, we use very small and very fast memories called registers, and a larger, cheaper and much slower one called "memory" or "RAM". The sequence of instructions to be executed is called the **program** and it is stored in the RAM. This kind of systems allow for a same given circuitry to be able to perform very different tasks simply by programming it.

Binary and Bus System

The CPU communicates with memory or input/output devices through buses. Buses are simply connections, than can either be set to a High or a low voltage. The High or Low state of a connection can be transferred(read or write) by components of a microprocessor based system. The High and Low state represent respectively the binary 1 and 0, which is the way of encoding data. We can then encode the number 23 as 1011 using a four bit bus (four lines), $L_1L_2L_3L_4$.

Fetch-Execute Cycle

The fundamental operation of most CPUs, regardless of the physical form they take, is to execute a sequence of stored instructions that is called a program. The instructions to be executed are kept in some kind of computer memory. Nearly all CPUs follow the fetch, decode and execute steps in their operation, which are collectively known as the instruction cycle.

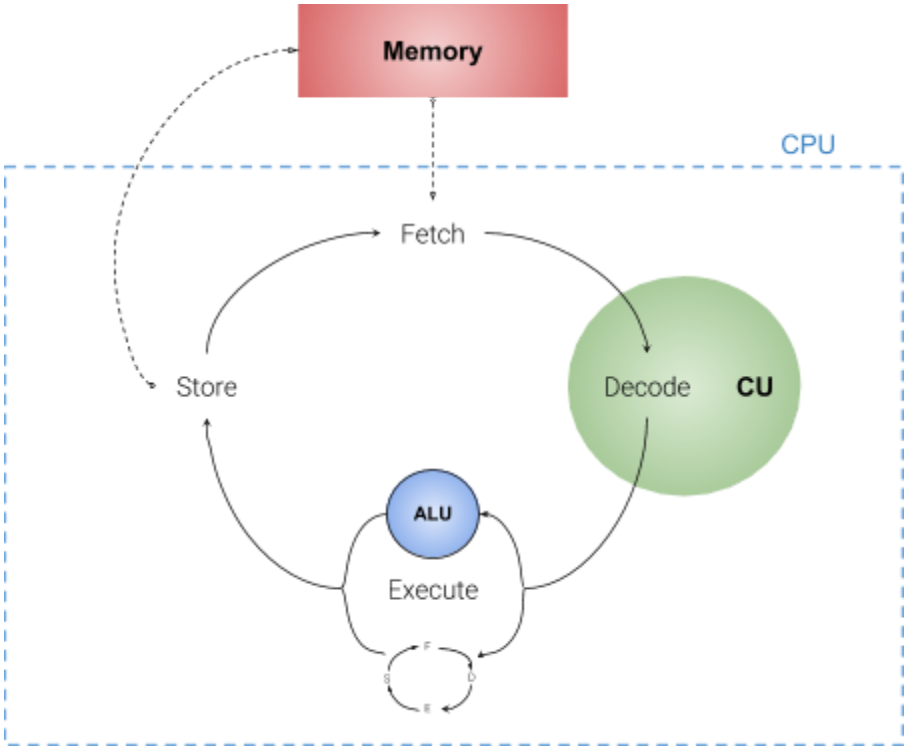


While the CPU is powered, it performs the instruction cycle continuously:

1. ***Fetch*** an instruction from memory: The first step, fetch, involves retrieving an instruction (which can be represented by a number or sequence of numbers) from program memory. The instruction's location (**address**) in program memory is determined by a **program counter** (PC), which stores a number that identifies the address of the next instruction to be fetched. After an instruction is fetched, the PC is incremented by the length of the instruction so that it will contain the address of the next instruction in the sequence.
2. ***Decode*** the instruction: The instruction that the CPU fetches from memory determines what the CPU will do. In the decode step, performed by the circuitry known as the instruction decoder, the instruction is converted into signals that control other parts of the CPU.

   The way in which the instruction is interpreted is defined by the CPU's **instruction set architecture** (ISA). Often, one group of bits within the instruction, called the **opcode**, indicates which operation is to be performed, while the remaining bits usually provide supplemental information required for the operation, such as the **operands**. Those operands may be specified as a constant value, or as the location of a value that may be a processor register or a memory address, as determined by some addressing mode.

   In some CPU designs the instruction decoder is implemented as a hardwired, unchangeable circuit. In others, a microprogram is used to translate instructions into sets of CPU configuration signals that are applied sequentially over multiple clock pulses. In some cases the memory that stores the microprogram is rewritable, making it possible to change the way in which the CPU decodes instructions, thus adding more flexibility to the processor.
3. Perform the necessary steps to carry out the instruction (store/fetch data to/from external memory or internal CPU storage, perform an arithmetic operation on data held in the CPU, jump to a different instruction location based on the outcome of a previous instruction's logical result). Depending on the CPU architecture, this may consist of a single action or a sequence of actions. During each action, various parts of the CPU are electrically connected so they can perform all or part of the desired operation and then the action is completed, typically in response to a clock pulse.
4. Very often the results are written to an internal CPU register for quick access by subsequent instructions. In other cases results may be written to memory(RAM).
5. Repeat all this starting again from step (1) : After the execution of an instruction, the entire process repeats, with the next instruction cycle normally fetching the next-in-sequence instruction because of the incremented value in the program counter. If a jump instruction was executed, the program counter will be modified to contain the address of the instruction that was jumped to and program execution continues normally.


Memory
CPU
Fetch
Store
Decode
CU
ALU
Execute

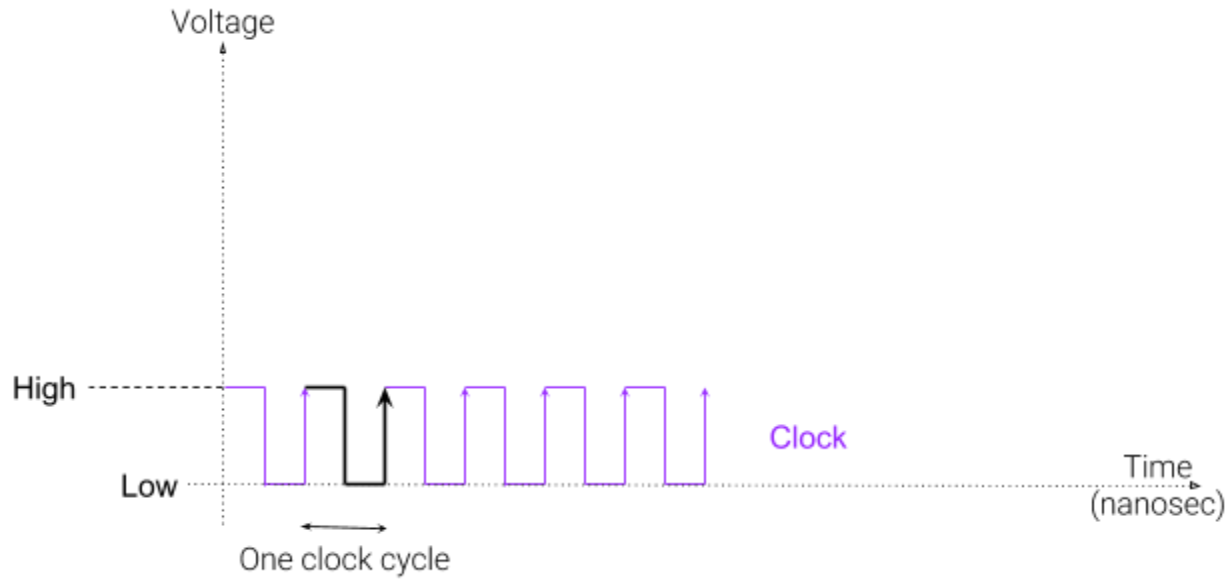
Project goals and implementation

- The present work only covers the design of a microprocessor using block diagrams, Algorithms (flowchart) and Pseudo-codes.
- Our microprocessor will be four bits, which we will connect to a memory. We are using a four bit data bus, a six bits address bus and an 8 bits-wide memory. However, for simplicity, we will assume a four bit wide memory and a four bit address bus (2^4=16 memory locations) in all our following explanations. We will also be using a top-down approach, going from high level blocks down to registers (and in some cases logic Gates) level.

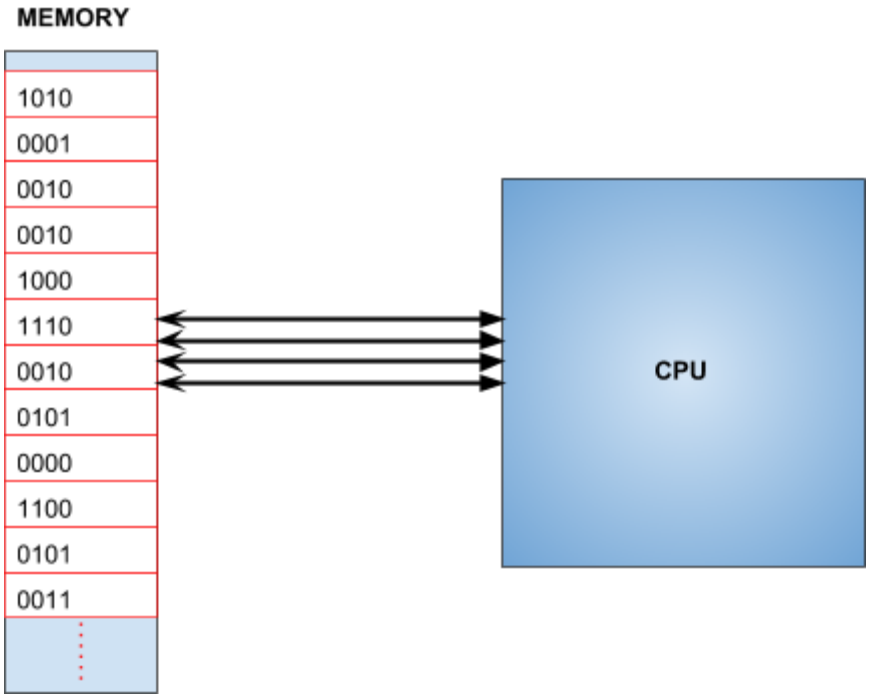
The Clock

Unlike analog computation systems, a digital computer cannot respond to inputs in a continuous-time way. For every change in state of any input, a digital system(component) needs to first "integrate" the change. It does that after a clock signal.

The inner work if a microprocessor based system is controlled by that clock. It is a component capable of generating fast periodic changes of state (High to Low, Low to High). The change in state of the clock provokes the execution of an operation in the microprocessor. **That is needed in order for the system to be discrete-time (therefore needed for it to be digital).**

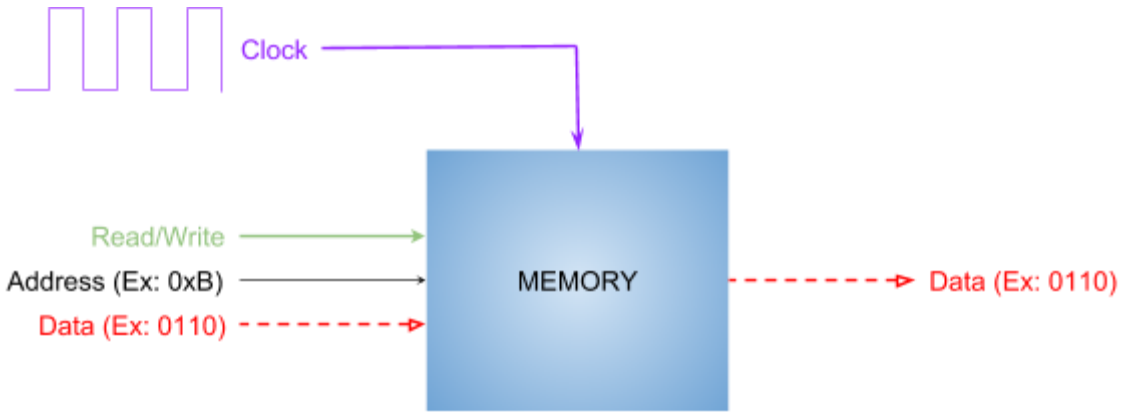


Memory and CPU communications



Let's use block diagrams and a Pseudo Language to describe the function of the memory and of the CPU
For simplicity we have assumed the Memory to be 4 bits wide, which is really small.
Also, in order to avoid confusion, we will represent address in hexadecimal and data in binary. The color red should be associated with data, black with address, green with control and purple with the clock.

Memory: The CPU send the control "Read" or "Write" with a memory address onto to address bus and data(optional) in the data Bus.
The memory then executes the control instruction: It either returns the data at the memory location received (in this case memory location 0xB = 11), or it store the data in the data bus (0110) into the appropriate memory location (0xB).



All Lines are Buses
- - - - - - - - - - represent a **possible** input/output

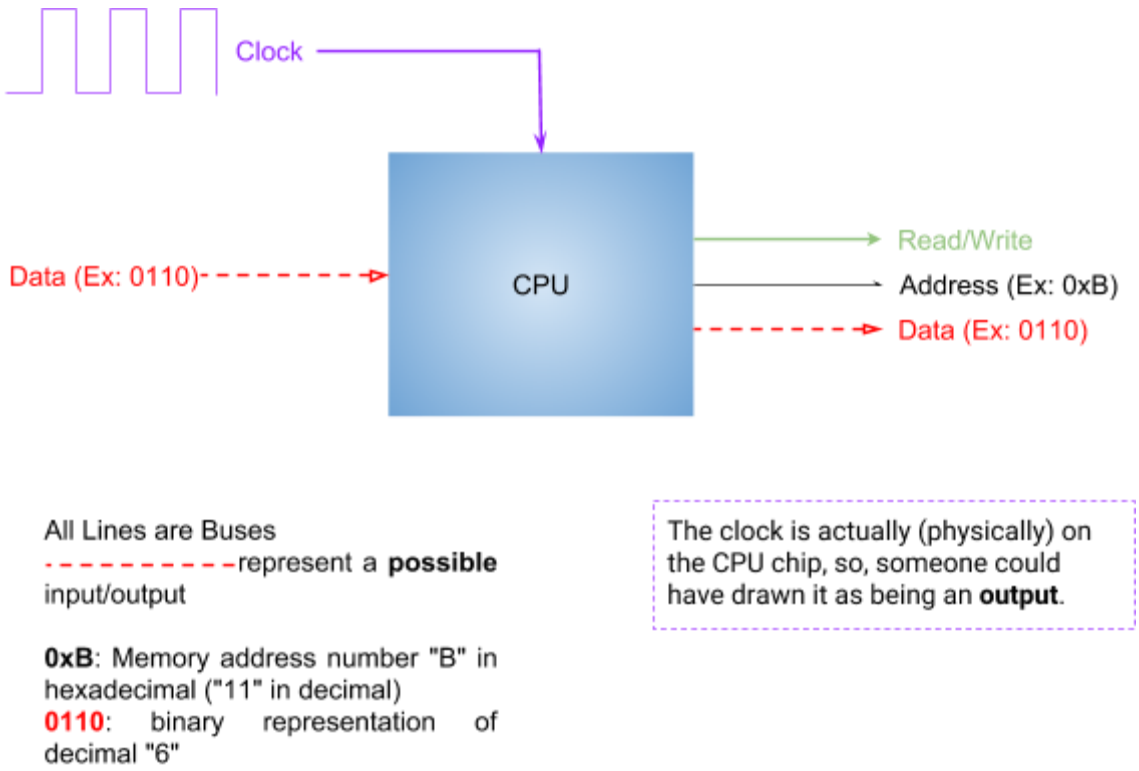
0xB: Memory address number "B" in hexadecimal ("11" in decimal)
0110: binary representation of decimal "6"

Control	Address	Data_in	Action
Read	0x7	XXXX	Return the value stored in the memory location 7
Read	0xC	XXXX	Return the value in the memory location 12
Write	0x7	0010	Store the value "0010" in the memory location 7
Write	0xB	1001	Store the value "1001"in the memory location 11

The XXXX mean the value of that variable is irrelevant to the function performed. In other words, in row 1 and row 2, the data input into the memory doesn't matter when the instruction from the CPU is **Read.**
FlowChart and Testing Data.(To Do)

NB: We are intentionally omitting another input to the memory called "enabled" which can either be True(1) or False(0). That signal determines whether the memory should perform the read/write actions. The purpose of this work not being to study the memory, we won't be mentioning the enable wire, as it would add unnecessary complexity to this document.
We are also not mentioning the asynchronization between the speed of the CPU and the speed of the memory.

CPU:
-write some assembly program and show its execution in steps (interactive simulation). (To Do)
-ALU Logic, CU Logic, Flip flops, transistors. (To Do)



When the CPU is powered, it fetches instruction from the memory, starting at memory location zero. In other words, it sends a read control instruction to the memory with the address 0x0. It then waits for the memory to respond, by sending the instruction to be performed as long with the operands needed for executing that instruction. For example if the instruction fetched is 0010 ("Add"), the CPU will then fetch the numbers to be added. Notice, this can mean making more than one fetch execute cycle. When all operands are available, the CPU performs the operation and puts the result wherever the instruction asked it to put it(it can be sent back to

memory). It then moves to the next instruction in memory and performs the same fetch-execute cycle till it is no more powered.
NB: With the CPU, it is hard to define what is an optional input/output. The execution of some instructions may not need to fetch data from memory, while the instruction couldn't have been executed without first fetching it from memory.

CPU components: Let's dive in the inner working of the CPU. The CPU is made of four main components: the control unit, the Arithmetic and Logic Unit (ALU), the registers and the internal bus.

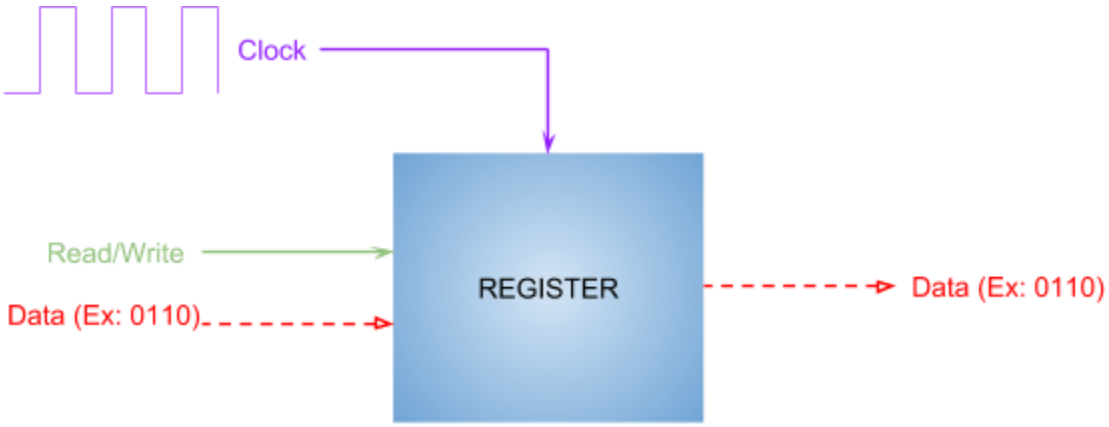
The internal bus: This is connection lines to which are connected every components of the CPU. Data is transferred between components though the internal Bus. In our case, it will be a four bits bus.

Three state buffer
Many components are connected to the internal Bus, but not all of them need to be enabled at the same time. In other words, there is a need of a control system that prevents the CPU components to affect the value in the bus. For that we use a three state buffer.
A tri-state buffer is similar to a buffer, but it adds an additional "enable" input that controls whether the primary input is passed to its output or not. If the enable inputs is trus, the tri-state buffer behaves like a normal buffer. If the enable input is false, the tri-state buffer passes a high impedance (hi-Z) signal, which effectively disconnects its output from the circuit.

Enable Input	Input A	Output
False	False	hi-Z
False	True	hi-Z
True	False	False
True	True	True

It allows current to flow, only when it is enables. Therefore all registers and other CPU components can be connected to the bus safely. It is only when the enable signal comes from the CU that the component will allow itself to exchange data with the bus. Multiple signals can then travel along the same bus.

The Registers: They are **very fast** small storage areas (typically from some few bits to several Bytes) directly on the CPU. They avoid needing to operate at the memory speed(which is really slow compare to the CPU). They have a wide range of function, from storing the present instruction to holding the results of an operation.



All Lines are Buses
- - - - - - - - - - represent a **possible** input/output

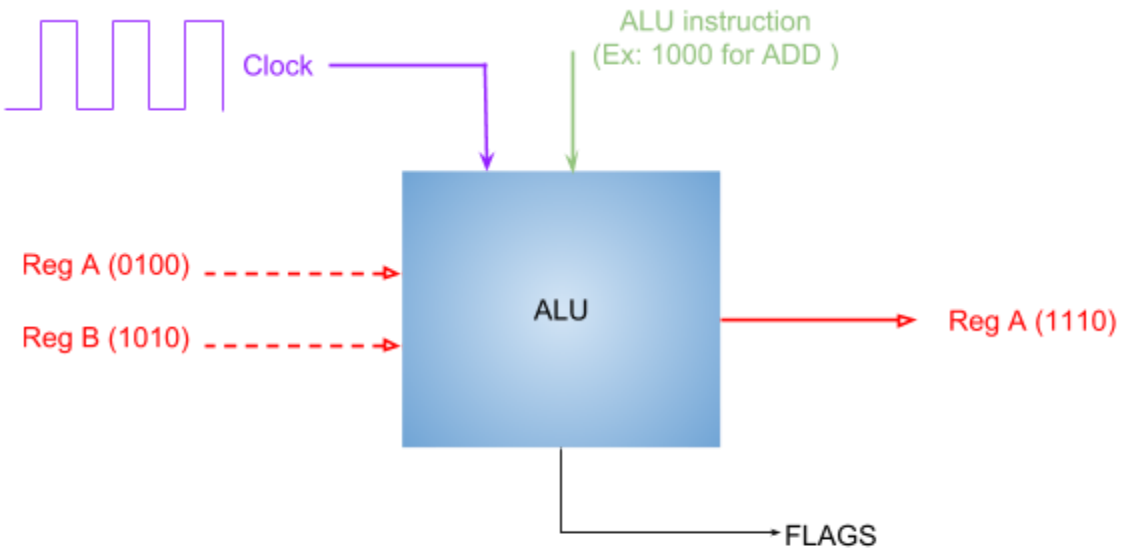
0110: binary representation of decimal "6"

The read and write instruction comes from the control unit. Data are either copied from or to the internal Bus. For example if a write instruction with data "0110" on the internal Bus is applied to a register, it will, at the beginning of the next clock cycle, copy the value "0110" from the internal bus. The same way, if a read instruction is sent to the register, it will, at the starting of the next clock cycle copy that value "0110" onto the internal bus. Every single one of these instructions are called microcycles. They happen after only one clock cycle, contrary to an instruction stored in memory which typically takes many clock cycles (many micro-instructions) to be performed.
The registers we will be using are (six):

- **Instruction register** (IR): It stores the current instruction to be or being performed. It is directly needed by the control unit
- **Memory address register** (MAR): this register hold a memory address. It is used when fetching data from a certain memory location. **It is directly connected to the memory**.
- **Buffer Register**: The buffer register is a **general purpose** register that **is directly connected to the memory**. Data fetched from memory comes directly in it and it also stores data to be written into memory. That role makes it a **Memory Data Register**. But at the same time, it can be used to store whatever data we want, during the execution of an instruction. It can store an operand to an operation for example. It is probably the most flexible register in our microprocessor. We will call it **register B**.
- **Accumulator**: which we will call **register A**. It automatically holds the result of any ALU operation, though it can be used as a **general purpose register** too.
- **FLAGS**: This special register also called status register stores different data in every bit it holds. The kind of data it stores are relevant to the CPU operations. For example it stores the carry Bit which result from the last ALU operations. We will dive into more detail concerning the FLAGS when we study the ALU.
- **Program Counter** (PC): This register holds the memory address of the next instruction of the program. It automatically increments itself after every instruction execution.

The ALU: It is the part of the processor that perform arithmetic and logical operations. The result of these operations are automatically stored in the accumulator. The common operations are addition, subtraction, logical AND, OR and comparisons. In choosing the operations to implement into the ALU, one should consider the most common tasks the CPU will have to perform, because the built-in ALU operations are executed in one clock cycle, unlike other operations that are a combinations of these built-in ones. An example is the multiplication. We can choose to implement multiplication if we think it will be a very common operation performed by the CPU. that way it will only take one clock cycle to perform a multiplication. Otherwise, we can still perform multiplication by doing a succession of additions, what will take way more clock cycles to be performed. In our case these are the built-in operations we are implementing in our ALU: addition, subtraction, AND, OR, NOT, SLT(set less than), Increment A, Decrement A, Complement A. We will explain our choice later. It is also important to note that reg A and reg B are not connected to the ALU through the internal bus. They are connected by a special connection. This is to allow other CPU operations to be performed while the ALU is performing an operation. *The control unit have to instruct the ALU on what operation to perform. This is done through control lines. Each operation is assigned a binary equivalent. In attributing them, we will assign the binaries with the less 1s (the less power) to the most frequently performed operations (in our opinion).*

0000	Decrement
0001	Increment
0010	SLT
0011	SUB
0100	AND
0101	OR
0110	Complement A
0111	XXXX
1000	ADD
1001	NOT
1010	XXXX
1011	XXXX
1100	XXXX
1101	XXXX
1110	XXXX
1111	XXXX



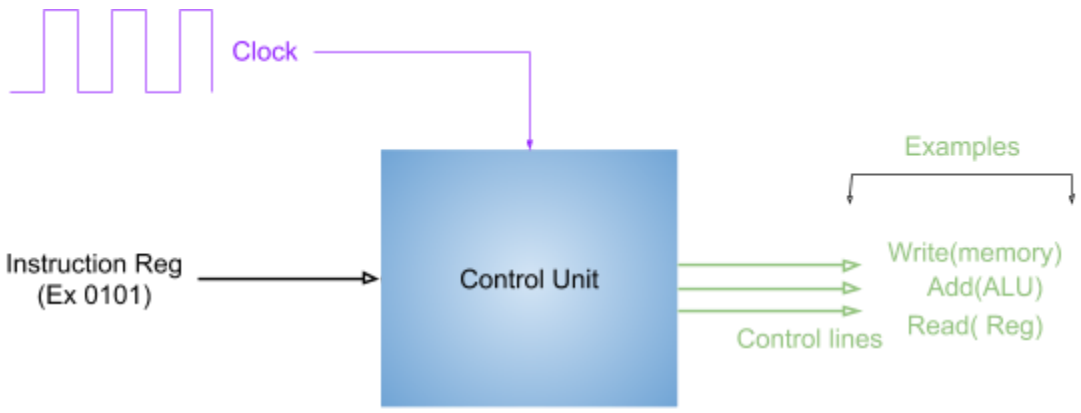
At least one input from a register is needed.

Let's analyze the content of the FLAGS: (To Do)

The Control Unit: The control unit determines the action to be performed by every component of the device, and when to perform it. It is the most complex part of the device. It is connected to every registers, to the ALU, the memory and input and output devices. When an instruction is fetched from memory into the Instruction Register, the control unit breaks the instruction down to steps of micro-instructions. For example when it receives the instruction *Mov the data in reg A to memory location 0xB*, it breaks it into these steps:

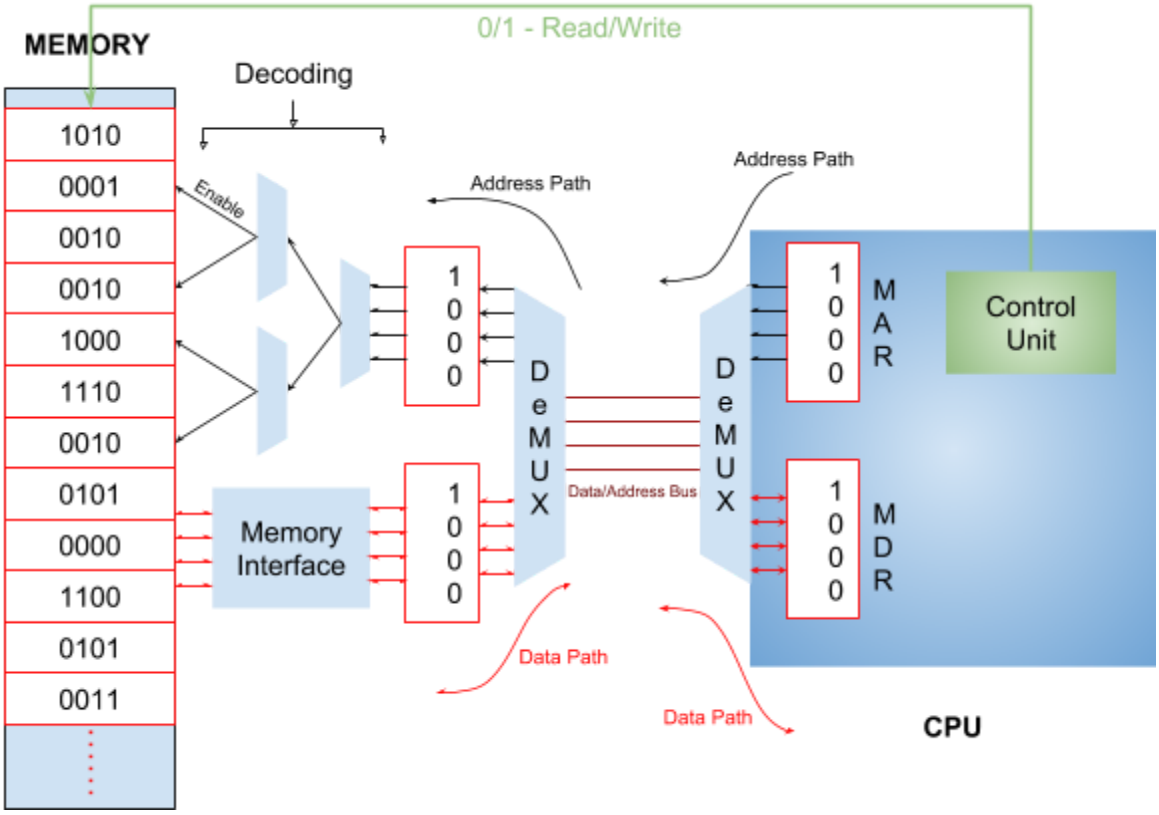
1. Read the address (0xB) from the memory data register (in our case, reg B) into the internal bus
2. Write data (from the internal bus) into the Memory **Address** Register
3. Read data from reg A (to the internal bus)
4. Write data (from the internal bus) into the Memory Data Register (in our case, reg B)
5. Write data into the memory

For each of these steps, the Control unit send the right instruction to the right component at the right time. It is also capable of handling the latency between the processor and the memory speed. And, for maximum efficiency, it can be performing operations with the internal bus, while, at the same time, interacting with memory.



Note that for one instruction as input, the control unit generates a sequence of micro-instructions as output, during several clock cycles.

Review of how the whole thing works together



The Microprocessor's job is to fetch and execute instructions. First, it sets the program counter to zero and sets the address register to the value of the program counter; it then sends the control instruction read to the memory through the control bus with the address bus set to the value of the MAR (zero).
Let's assume the memory content is:
0001-1100-0000-0101-0011-1101-1000-X-X-X-X-X-0110-X
A pseudo assembly code for this might be:

```
MOV , 0xC          ;Move the data at memory 0xC to reg A
ADD 5              ;Add 5 to reg A
MOV 0xD, reg A     ;Move the content of reg A to memory 0xD
HLT                ;end of program
```

The memory send back the data stored in the memory location zero through the data bus. The control Unit send the instruction "read from data bus" to the memory data register (buffer register or register B). The state (High or Low) of the connections in the bus is then transferred to (stored in) the reg B. It is then copied into the instruction Register and then decoded into a number of steps. For example, let's assume the content of memory location zero is "0001" which is the binary representation for the instruction "move the data **which memory location is stored at the next memory space** into the register A". That means the content of the next memory location is the address of the data we want to copy into the reg A. That instruction will be decoded to:

1. Send "end of instruction signal to the Program Counter": which basically means, increment the PC (PC←PC+1 = PC←0+1)
2. Read PC: Set the internal Bus to the value in the Program counter.
3. Write MAR: Set the memory address register to the value in the internal bus. MAR now contains the value one.
4. Read MAR: Set the **external address bus** to the value in MAR
5. Read Memory: Send control signal read to memory. The memory read the value at memory location one. That value is the address of the data we want to move to reg A. Let's say that value is twelve (0xC).
6. Write reg B: Set reg B to the value in the external data bus
7. Read regB
8. Write MAR: The memory address register now contains the address we just copied from memory 0xC)
9. Read MAR
10. Read Memory: Memory send back the data at memory location 0xC. Let's say that is binary "0110"
11. Write reg B: Reg B is set to "0110"
12. Read Reg B
13. Write reg A: Reg A finally contains "0110"
14. Start **default**: (steps 1,2,3,4,5,6,7) and then Send the data in reg B into the control circuit. The **default** is the action the control unit does by default (incrementing PC and fetching instruction from next memory location and decoding that instruction).

After that first instruction, the PC is 2 and the microprocessor fetches the instruction "0000" ADD (Add accumulator to the value in the next memory location). That instruction is decoded into the following:

1. Increment PC: PC = 3
2. Read PC
3. Write MAR
4. Read MAR (by the memory)
5. Read Memory
6. Write reg B: Reg B = 5 (5 is the value stored at memory location 3)
7. Send reg A(6) and reg B(5) to ALU, send control instruction ADD "1000" to ALU. The ALU performs the operation and the result is automatically in reg A. Reg A is now 11.
8. Start default.

The third instruction is:
MOV 0xD, reg A *;Move the content of reg A to memory 0xD*

The steps this instruction is decoded into are:

1. Increment PC: PC = 5
2. Read PC
3. Write MAR
4. Read MAR
5. Read Memory:
6. Write Reg B: reg B = 1100 = 0xD
7. Read reg B
8. Write MAR: MAR = 0xD

9. Read reg A
10. Write reg B: reg B = 11
11. Read MAR into address bus and reg B into data bus
12. Write memory: The memory stores the value 11 at the memory location 0xD
13. Start default

The fourth instruction is HALT. In our microprocessor this means the microP stops its default work, while in an advanced microP the meaning of the HALT might be different (go back to OS program).

The Instruction Set

To choose a set of instruction, we are concerned with:

1. The maximum number of instructions that we allow ourselves
2. The instructions needed
3. The common operations to be programmed

The choice of a set of instruction is mainly governed by efficiency. We must think of the kind of operations the microP based system is going to be programmed to perform, with the constraint of a maximum number of instructions. That constraint is often due to the bus width and memory width. We don't want to wait for the memory to fetch instructions taking multiple read cycles. In our case, we chose a maximum of four-bit wide opcodes, with most of them having implicit operands.

Implementing those operations in the hardware will be more efficient than implementing another set of instructions that could be combined to perform that operation. For example we might think about implementing the instruction MOV which needs the source and the destination to be specified, or simply implementing a MOV with implicit destination being the reg A. Another example is the NAND and NOR operations. How often will they be use? Knowing that AND and OR instructions are already implemented, should we implement one NOT instruction or two NAND and NOR instructions. Should we implement multiplication as an instruction or we leave it to the programmer? What operations do we want to be performed in the minimum clock cycles? Those choices can make a big difference in the performance of a microProcessor.

We also think what operations will be the most common in order to choose the operations **binary equivalent** that will contain the least numbers of High states (binary 1). That can save energy.

As the purpose of our work is purely educational, we chose a simple set of instruction, without much worry about the efficiency. Our choice is the following:

FORMAT	OPCODE	DESCRIPTION
LDA (address)	0000	Load Register A
STA (address)	0001	Set Register A
ADD (address)	0010	Add reg A
SUB (address)	0011	Subtract reg A
AND (address)	0100	AND reg A
OR (address)	0101	OR reg A
NOT	0110	NOT reg A
SLT (address)	0111	Set less than (reg A = 0001 if regA < [address])
MOV	1000	set reg A with data in next memory location

JZ (address)	1001	Jump to address when zero flag is True
JC (address)	1010	Jump to address when carry flag is True
INA	1011	Increment A
DEA	1100	Decrement A
CMA	1101	Complement A
JMP (address)	1110	Jump to memory location
HLT	1111	Halt Processor

Here is an example of Assembly code that could be written. It performs multiplication of 4 and 3. This program decrements one of the factors (the number 4) when it performs an addition of the other factor (reg A + 3) . The result is stored at memory location
Starting at memory location 0x0:

```
0x0    MOV 4      ; Move 4 to reg A
0x2    DEA        ; Decrement reg A. reg A = 3
0x3    LDA 0      ; Store the content of reg A in memory location 0x0
0x5    MOV 3      ; Move 3 to reg A
0x7    LDA 1      ; Store the content of reg A in memory location 0x1
                  ; The content of reg A is still 3
0x9    LDA 2      ; Store the content of reg A in memory location 0x2
                  ; [0x0]=3 [0x1]=3 [0x2]=3
0xB    ADD 2      ; Add reg A and the content of memory location 0x2 (the number 3)
0xD    LDA 2      ; Store the content of reg A in memory location 0x2
                  ; [0x0]=3 [0x1]=3 [0x2]=6
0xF    STA 0      ; Set A with the content of memory location 0x0 (the number 3)
0x11   DEA        ; Decrement reg A
                  ; reg A = 2
0x12   JZ 26      ; If Zero flag is True, then jump to memory location 0x1A
0x14   LDA 0      ; Store the content of reg A in memory location 0x0
                  ; [0x0]=2 [0x1]=3 [0x2]=6
0X16   STA 1      ; Set A with the content of memory location 0x1 (the number 3)
0x18   JMP 11     ; Jump to memory location 0xB
0x1A   HLT        ; HALT
                  ; the result is in memory location 0x2
```