MICROPROCESSOR

Introduction

A **microprocessor** is a computer processor on a microchip. The microprocessor contains all, or most of, the components of the Central Processing Unit (CPU).
To understand the microprocessor, we first need to understand the need for a processing unit.

Electronics is simply about controlling the flow of electrons in order to build useful **-automatic-** systems. We have for example: a water level controlled pump, a temperature controlled fan, an ATM machine, any device that can send data through the internet (a phone, a server, a PC). In either of these systems there is always an **input** system, an **output** system, and a **processing or control system**.
In the case of the fan, the input device would be a temperature sensor, the output is the motor that turns the fan, and the control system is the 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 for: when it is supplied with a current or voltage greater than its threshold, it allows current to flow in another branch (or voltage to appear at another node) of the circuit.
-----*Transistor circuit, Ic is determined by Ib(To Do)*-----
We can use a group of such transistors to control the fan.
However, due to the **analog** nature of the transistors, combining many of them can quickly get messy and their behavior can become extremely hard to predict. Also, for many purposes, analog systems might be unnecessarily complicated, and their resistance to noise is often poor. That is why we choose to use digital systems.

In a **digital system**, at any point in time, the current or voltage can only take one of a set of predefined **FINITE** values. We are also no longer interested in the continuous values of our input. We only measure it at **DISCRETE** times. In the case of the ATM machine for example, a user's input is better represented by a digital circuit: The user either presses a button or he doesn't. There is a finite number of possible 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, etc). The logic gates perform logical operations like AND, OR, NOT on binary inputs. They are combined to produce the desired output. That processor is named **random logic** and it is way easier to design than its analog equivalent.

A more advanced kind of digital system, is a **programmable** digital system. In this type of system, no logic circuit is specifically designed to perform a particular task, rather, it contains 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 your smartphone, there is no special circuit for Whatsapp, and another one for Youtube. And there is not a special circuit to play music and another to write documents. All those apps use the same circuitry, but in different ways(combinations and sequences). Their ways of combining your phone's basic available operations is **the software** or **computer program**. The software is a sequence of instructions that are performed by a computer in the order in which they are defined (by the programmer). (An attentive reader might guess 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 all the logical and arithmetic operations that might be needed by the various 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 from/to the memories, Input/Output 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 computers, 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. These kinds of systems allow for the same given circuitry to be able to perform very different tasks simply by programming it.

MicroP Part A

Binary and Bus System

The CPU communicates with the memory and the input and output devices through buses. Buses are simply connections, that can either be set to a High or a Low voltage. The High or Low state of a connection can be transferred(read or written) by any component connected to it.
The High and Low state represent respectively the binary 1 and 0, which is the most common way of encoding data. We can then encode the number **23** as **1011**, and transfer it using a 4-bit bus (four lines), **$L_1L_2L_3L_4$**: L_1 is set to High, L_2 is set to Low, L_3 is set to High, L_4 is set to High.

Fetch-Execute Cycle

The fundamental operation of most CPUs, regardless of the physical form they take, is to execute a sequence of stored instructions(the 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.

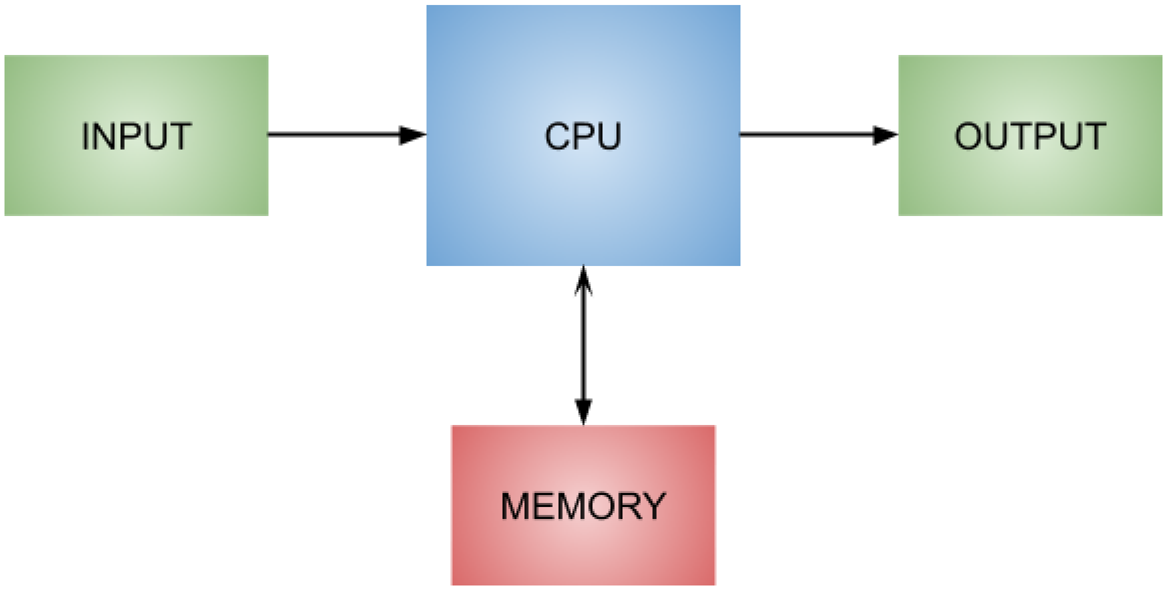


Figure 1

When powered, the CPU 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 location (**address**) in program memory of the instruction to be fetched is stored in the **program counter** (PC). 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 is repeated, 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.

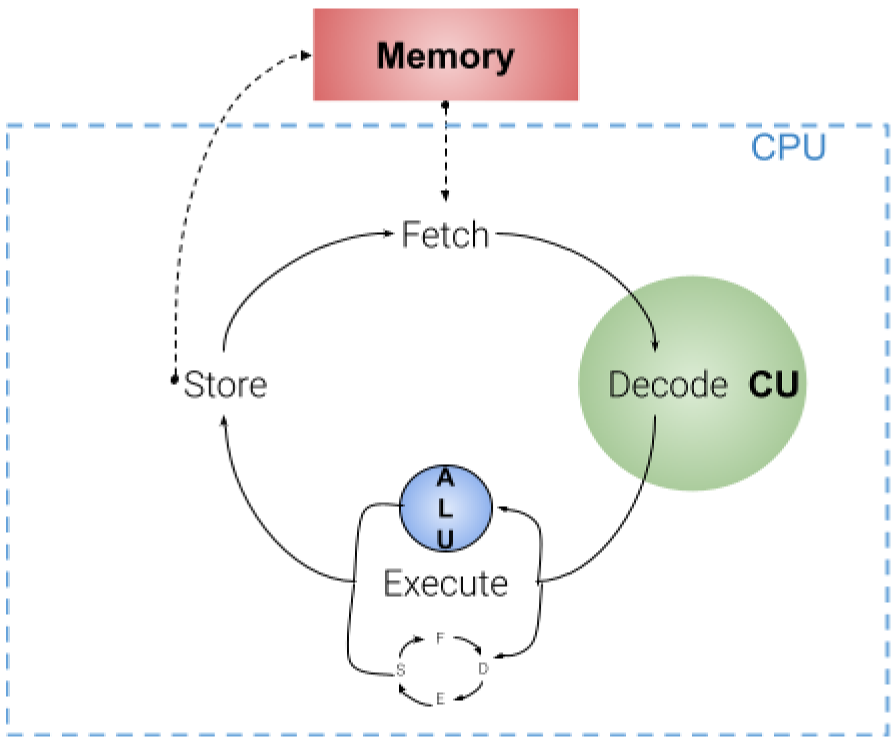


Figure 2

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 4-bit data bus, a 6-bit address bus and an 8-bit 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 our drawings. We will also be using a top-down approach, going from high level blocks diagrams down to registers -and in some cases logic Gates- level (see Logicly files).

The Clock

Unlike analog computation systems, a digital computer cannot respond to inputs continuously. For every change in the state of any input, a digital system (or component) needs to first "integrate" the change. It does that after a clock cycle.
The inner working of 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 triggers 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)**.

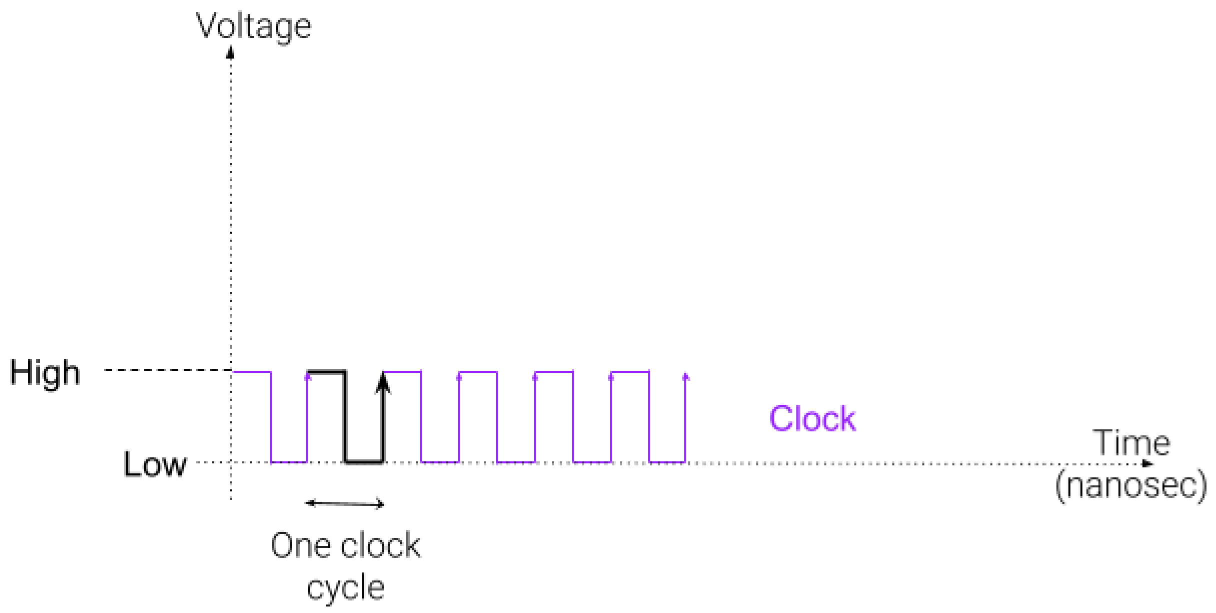


Figure 3

Memory and CPU communications

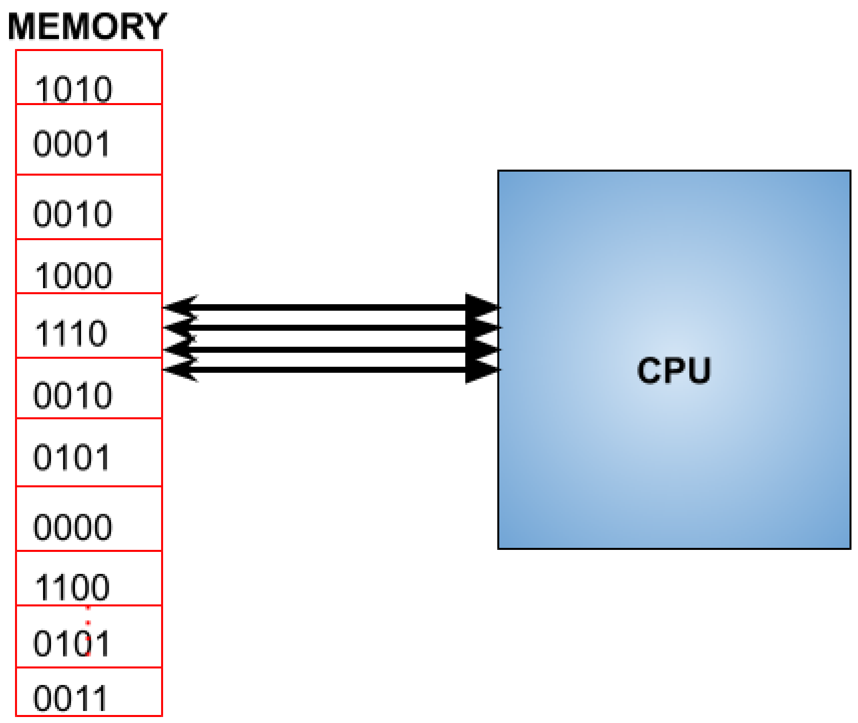
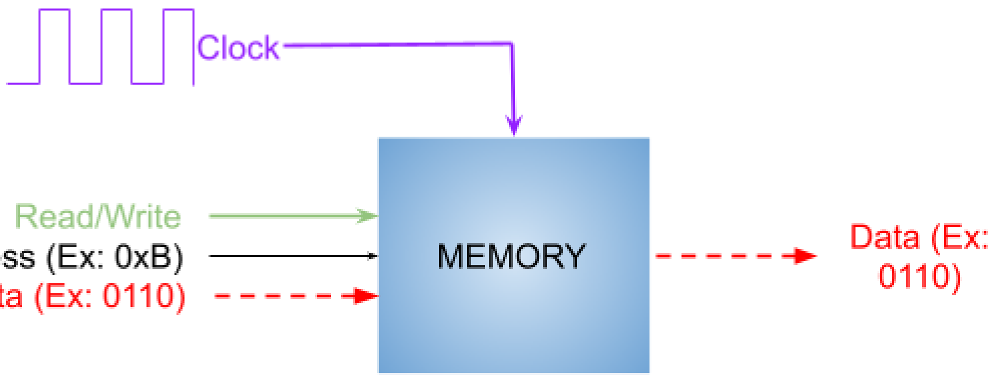


Figure 4

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-bit wide, which is really small.
Also, in order to avoid confusion, we will represent addresses in hexadecimal form 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 sends the control signal "**Read**" or "**Write**" with a memory address onto the address bus (and sometimes data onto 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 stores the data on the data bus (0110) into the appropriate memory location (0xB).



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

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

Figure 5

Control	**Address**	**Data_in**	**Action**
Read	0x7	XXXX	Returns the value stored in the memory location 7
Read	0xC	XXXX	Returns the value in the memory location 12

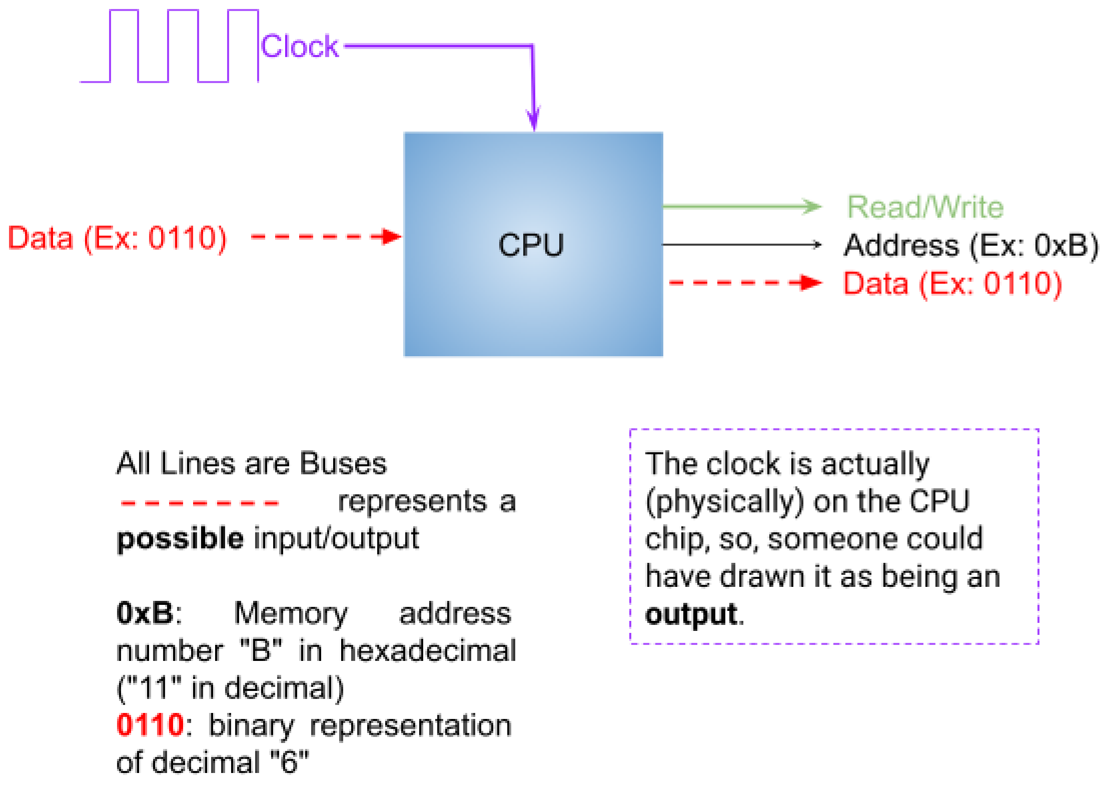
Write	0x7	0010	Stores the value "0010" in the memory location 7
Write	0xB	1001	Stores the value "1001"in the memory location 11

The XXXX means 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 is not to study the memory, so we won't be mentioning the enable wire.
We will also not discuss the **difference in speed between the CPU and the memory**. It has to be taken into account when designing a microprocessor because the memory is too slow to respond to the CPU requests in one clock cycle. In order for the CPU not to wait for the response from the memory, modern computers have one or many **cache memories** which are very fast but small storage elements often put on the CPU itself. They are loaded with the resources in memory that are the most likely to be requested by the CPU.

CPU

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



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

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

The clock is actually (physically) on the CPU chip, so, someone could have drawn it as being an **output**.

Figure 6

When the CPU is powered, it fetches instructions from the memory, **starting from memory location zero**. In other words, it sends a "Read" 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 the execution of that instruction. For example if the instruction fetched is 0010 ("Add"), the CPU will fetch the numbers to be added. (Note that, this can mean making more than a single fetch-execute cycle.) When all operands are available, the CPU performs the operation and puts the result wherever the instruction requires it to be put (the result can be sent back to memory, or kept in a register for further use). The CPU then **moves to the next instruction** in memory and performs the same fetch-execute cycle. It keeps on doing that till it is no longer powered.

CPU components

Let's dive into 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
It is just a set of lines to which are connected every component of the CPU. Data is transferred between components though the internal Bus. In our case, it will be a four-bit bus.

Three-state buffer
Many components are connected to the internal Bus, but not all of them are sending or receiving data every time. There is a need for a control system to prevent the CPU components from writing/reading on/from the bus at all times. For that function, we use a three-state buffer (or tri-state buffer).
A tri-state buffer is similar to a buffer, but it has an additional "enable" input that determines whether the primary input is passed to the output or not. If the "enable" input is true, the tri-state buffer behaves like a normal buffer. If the enable input is false, the tri-state buffer shows a high impedance (hi-Z), 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 enabled. Therefore, all registers and other CPU components can be connected to the bus safely. It is only when **the Control Unit sends the enable signal to a tri-state buffer**, that the component connected to it can exchange data with the bus. Multiple signals can then travel along the same bus.

The Registers
They are very fast and extremely small storage areas (typically from a few bits to several Bytes) directly on the CPU. Thanks to them, the CPU needs not to operate at the memory speed (which is really slow compared to the CPU). They have a wide range of functions, from storing the present instruction to holding the results of an operation.

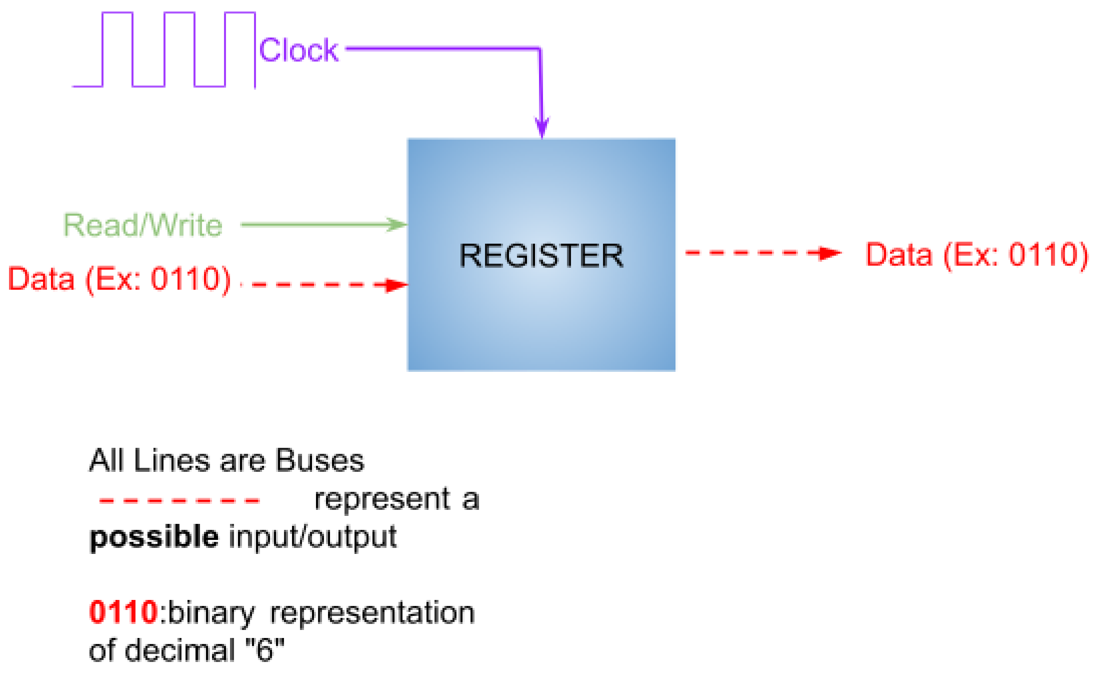

Clock
Read/Write
Data (Ex: 0110)
REGISTER
Data (Ex: 0110)
All Lines are Buses
represent a **possible** input/output
0110:binary representation of decimal "6"

Figure 7

The read and write instructions come from the Control Unit. Data is either copied from or to the internal Bus. For example if a write instruction, with the data "0110" on the internal Bus, is sent to a register, it will, at the beginning of the next clock cycle, copy the value "0110" from the internal bus. Similarly, 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. The execution of every single one of these instructions is a **microcycle**. They occur after only one clock cycle, contrary to an instruction stored in memory which typically takes many clock cycles (many **micro-instructions**) to be executed.
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 holds 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 results from the last ALU operations. We will study in more detail 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 performs 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 ALU operations are executed in one clock cycle, unlike other operations that are combinations of these built-in ones. An example is the multiplication. We can choose to implement multiplication in the ALU 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 has to instruct the ALU on what operation to perform. This is done through control lines. For the control lines to hold information on the operation to be performed, each ALU-operation is assigned a binary equivalent.

In attributing them, we will assign the binaries with the less 1s (the less power consuming) 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

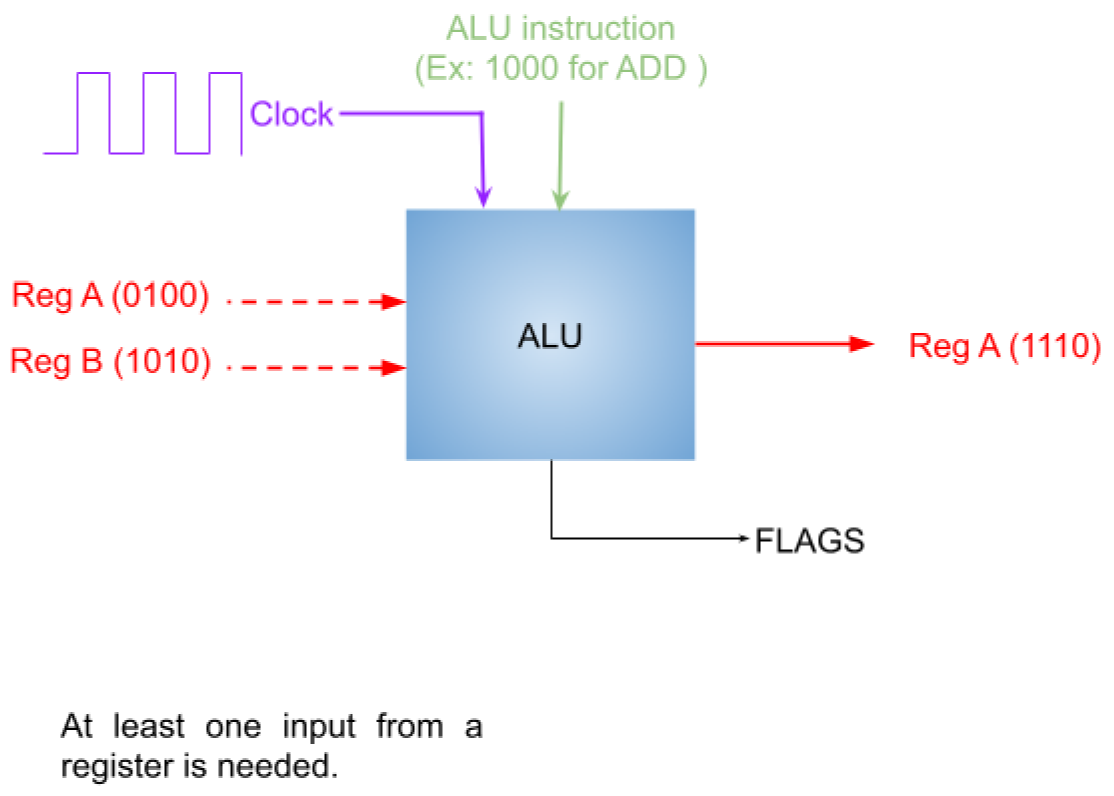


Figure 8

-Let's analyze the content of the FLAGS: (#ToDo)

The Control Unit

The control unit decides the action to be performed by each component of the device, and when to perform it. It is the most complex part of the microprocessor. It is connected to every register, to the ALU, to the memory and to the 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 *Move 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 sends **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. And, for maximum efficiency, it can be performing operations with the internal bus, while, at the same time, interacting with memory.

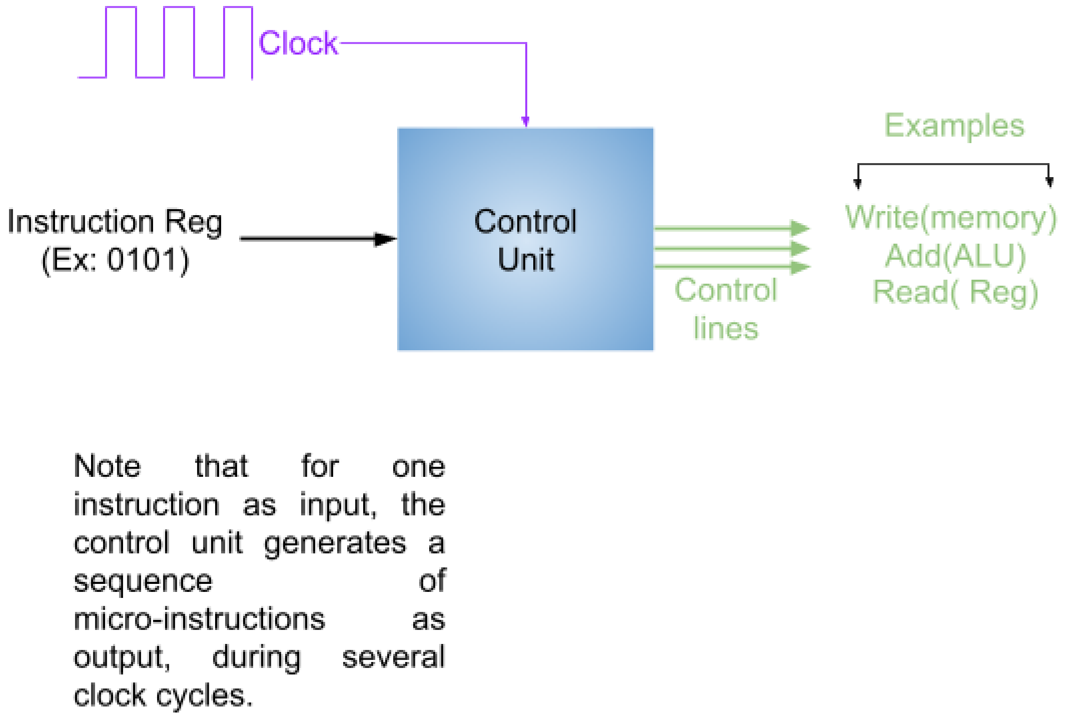

Clock
Examples
Instruction Reg (Ex: 0101)
Control Unit
Write(memory)
Add(ALU)
Read(Reg)
Control lines
Note that for one instruction as input, the control unit generates a sequence of micro-instructions as output, during several clock cycles.

Figure 9

Review of how the whole thing works together

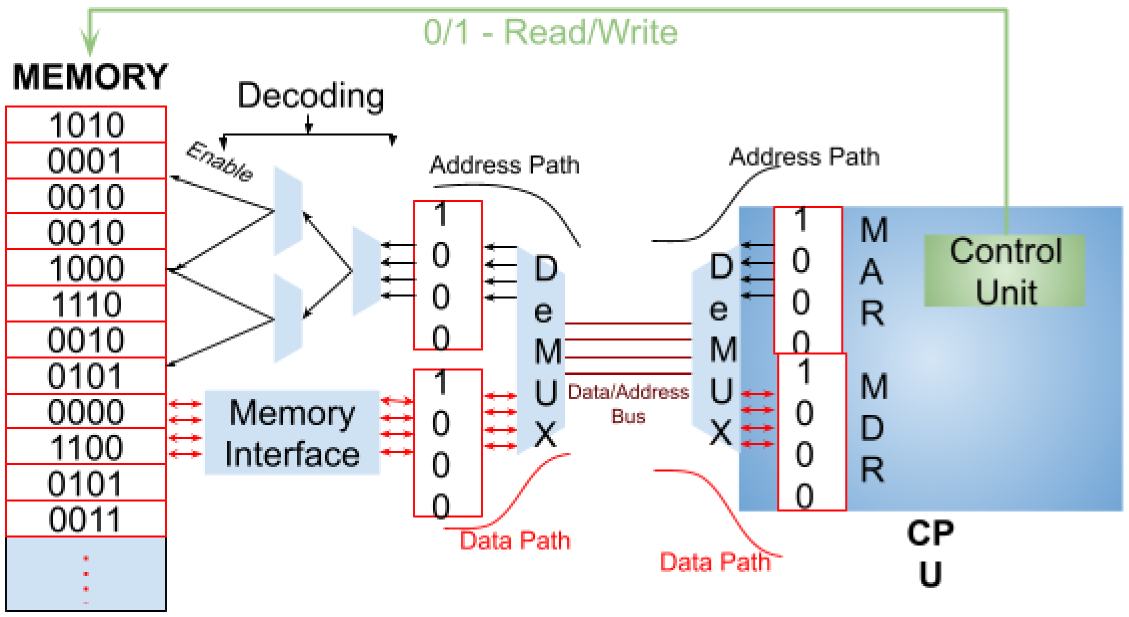


Figure 10

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 via 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 location 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 sends back the data stored in the memory location zero via the data bus. The control Unit sends 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 to the Instruction Register and 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 with memory location 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 0x1.
4. Read MAR: Set the external address bus to the value in MAR
5. Read Memory: Send the control signal "Read" to memory. The value at memory location 0x1 is read. 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 on 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 sends 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 to the control circuit. The default is the action the control unit does by default (incrementing PC, fetching instruction from next memory location and then decoding the instruction).

After that first instruction, the PC contains the value 2 and the microprocessor fetches the instruction "0000" ADD (Add to accumulator, 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 microprocessor 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 performed

The choice of a set of instructions 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. In our case, we chose a maximum of four-bit wide opcodes.

We don't want to wait for the memory to fetch instructions taking multiple read cycles. That is why most of our instructions will have 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 used? 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 number of clock cycles? Those choices can make a big difference in the performance of a microprocessor.

We also think of the operations that will be the most common, so that we can 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, and not worry much about 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 from 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 the zero-flag is True
JC (address)	1010	Jump to address when the 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 to perform the product of 4 and 3.
This program decrements one of the factors (the number 4) every time it performs an addition of the other factor and reg A (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 (12) is in memory location 0x2
```

MicroP Part B (Circuits): See Logicly files

ALU

A decoder is used to separate the circuitry of all different control instructions received by the CPU.

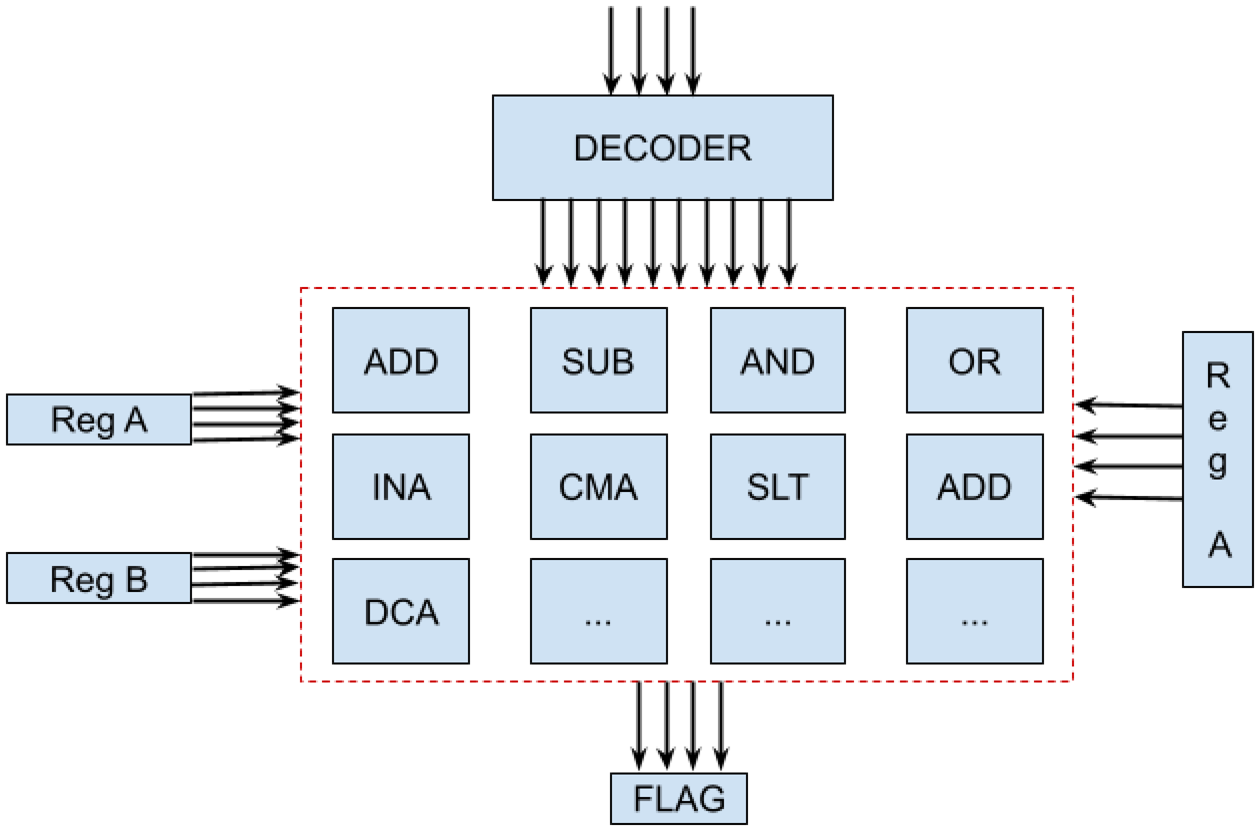


Figure 11

Control Unit (#ToDo)

Registers (#ToDo)

Memory (#ToDo)

MicroP (#ToDo)

CONCLUSION

#ToDo