Documentation of the calM

Cpu architecture learning machine.

I. Documentation Organization ----------2
II. Introduction: Brief Overview Of The Cpu Architecture. ----------3
III. Calm Architecture Organization ----------4
1. Memory Unit ----------5
Command Unit ----------8
2. Processing Unit ----------10
3. I/O Unit: ----------12
IV. Instructions Set Architecture ----------14
1. General Format ----------14
2. Reduced Format ----------15
3. OPCode for general format instructions + size bit ----------15
4. OPCode for reduced format instructions ----------17
5. Ind bits + Reg/Mod bits ----------17
6. Register's code ----------18
7. Addressing modes ----------19
V. Execution Process In The Calm ----------21
VI. Appendix ----------22

I. Documentation Organization

The documentation contains 5 chapters and one appendix. Chapter 2 is a brief description of the calMachine, Chapter 3 describes the calM architecture, then Chapter 4 will present the ISA: instruction set architecture, Chapter 5 will expose how an instruction is executed in the machine in order to highlight the execution phases and each unit role, finally the Appendix is reserved to explain the concepts that needs clarification for first year computer science students, every fresh starter has to read it carefully.

II.Introduction: brief overview of the CPU architecture.

calM stands for CPU architecture learning machine, and as it's mentioned in the name it's a machine (cpu) that we designed for pedagogical purpose, which provides Computer Science, Computer Science engineering and Electrical engineering first year university students a cpu architecture that groups the most important and fundamental cpu concepts, represented in a simplified way, so to help them understand computers architecture (cpu architecture specifically)

We designed a simple ISA (Instruction Set Architecture) and an assembly language for it. Then we developed an emulator where users can execute programs written in its assembly language, and a simulator that allows the student to visualize how instruction are executed in a cpu while highlighting the phases of this operation.

All those features are integrated in a website which is dedicated to teach computer architecture, based on our "calM" model. It is destined to be used in Archi1 course, the first computers architecture course at ESI, Ecole Nationale Supérieure d'Informatique(ESI-ex-INI), the school where we are having our Computer Science Engineering studies, and this specific project "calM" is part of our curriculum in the second year at the school.

Each team of six students is assigned a project to work on, which is supervised by two professors and considered as a course in our second semester.

III.calM Architecture Organization

As mentioned, **calM** is a pedagogical purpose machine, so it will have a simplified architecture that will englobe the main concepts that can be found in any Von Neumann architecture-based CPU.

calM is a CISC machine with 2 bytes address bus and a *2 bytes* data bus, with a total of 15 registers with a size of 2 bytes each one:

- **4** general purpose registers **R1, R2, R3, R4,** three of them are divided into two parts high and low (Most significant byte is high and least significant byte is low they are used mainly in one byte operations when we have operands in one byte of memory), which are: **R1H, R2H, R3H, R1L, R2L, R3L**.
- **2** in the entrance gateways of the UAL : **UALR1** and **UALR2**.
- **2** used in the based and Indexed addressing modes: **BR, IDR**.
- **1** stack register **: SR**.
- **1** instruction Register: **IR**.
- **1** memory address register: **MAR**.
- **1** memory data register: **MDR**.
- **1** instruction pointer: **IP**.
- **1** accumulator register: **ACC.**
- **1** flags register: **Flags register**.

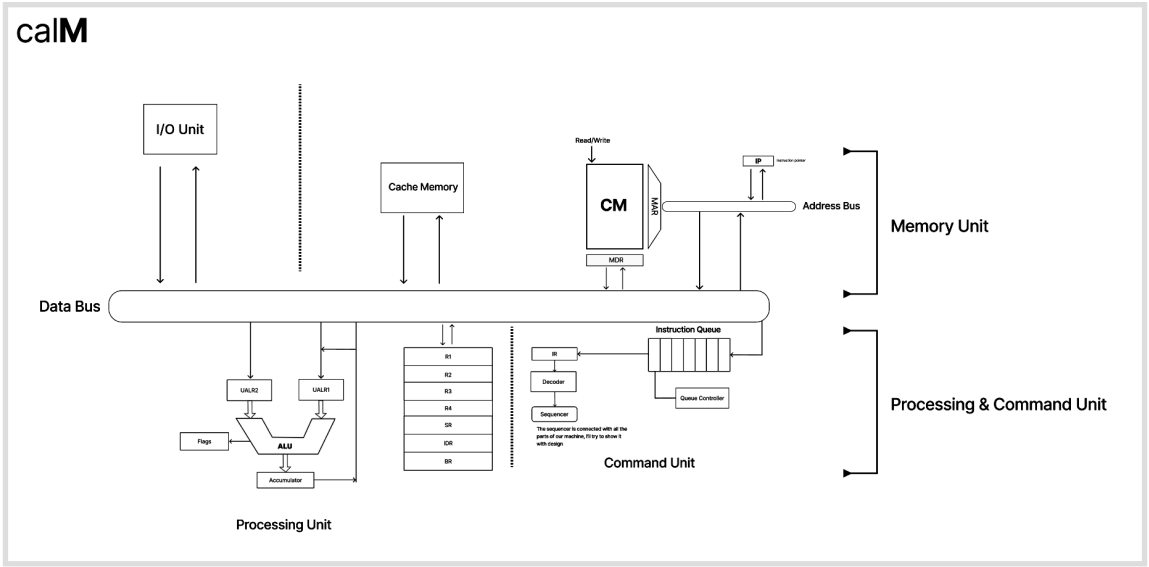


Figure 1: calM Architecture

Our machine is divided into **4 Units** all connected with the **Data Bus:**

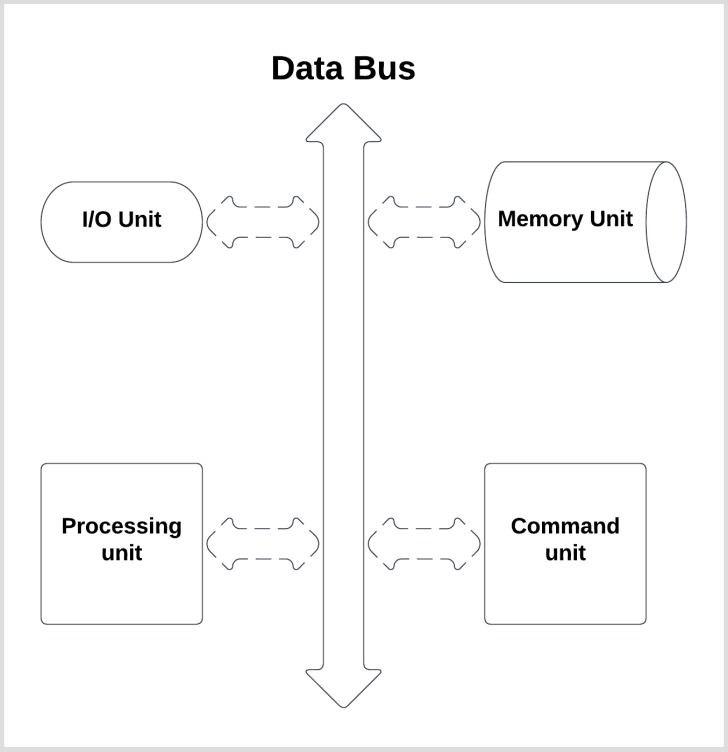


Figure 2: Units diagram in calM.

1.Memory Unit

It is the unit where the cpu stores data it needs in order to perform its basic operations, such as instructions and variables or constants values in addition to stack ..etc

In order to perform such tasks it's constituted from the following parts:

- **IP** register where we store the address of the next instruction the cpu has to execute, we increment its value after executing each instruction, and in the case of a branch instruction to **inst1**, its value gets changed so that it points to the address containing the instruction indicated in the branch instruction.
- **CM:** central memory is our biggest and main component in the memory unit and it has a size of 64 kilobytes, note that the CM is divided into 3 part which are: the DATA segment which contain the data that the program needs to be executed (variables …) , the CODE segment which contain the program that needs to be executed and the STACK segment where we implement the memory stack.

- **MAR:** Memory Address Register which stores the address that we want to perform a write/read operation on it in the CM.
- **MDR:** Memory Data Register where the written/read data gets stored after the R/W operation, for example: if we have 0x0000 in the MAR and we perform a write in the CM we will write the word in MDR in the box of address 0x0000.
- **Data & address buses:** buses are a very important component that ensures the transfer of the data in the cpu, the data bus does only transfer data words, and in the address bus we do only transfer address words.

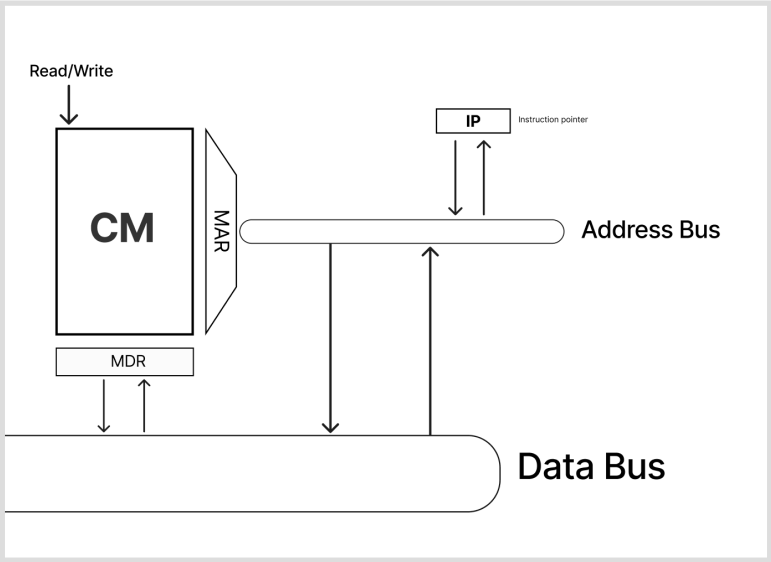


Figure 3: Memory Unit

- **Cache memory:** Cache memory is smaller than **CM** but its way faster. Its purpose is to store frequently used data so it can be accessed quickly (it is an additional component in this unit, so that the student knows that it exists without going through more details).

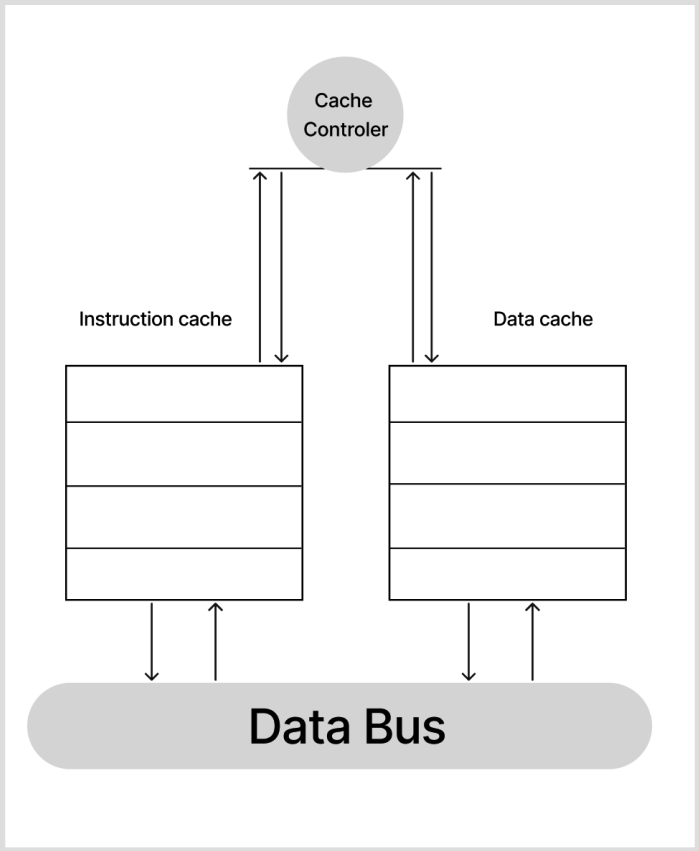


Figure 4: Cache Memory

Command Unit

Its task is commanding the units of the machine, it receives the instructions that comes from the memory unit first, then it will be decoded and passed to the sequencer that executes this instruction by translating it to a micro-commands sequence, that can be executed through all the cpu, the command unit is designed as a set of consecutive components that are:

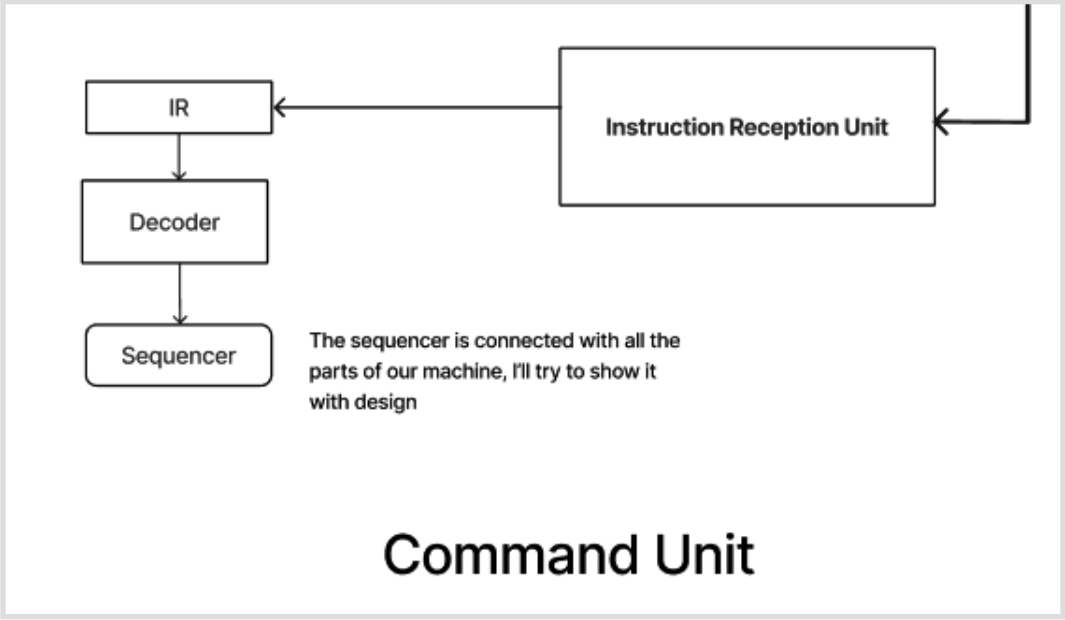


Figure 5: Command Unit

- **Instruction reception unit**: it's the unit that receives the instructions from the memory unit, and it organizes it in a way that makes decoding it simpler and more efficient, it's mainly queue commanded by a controller :
    - **The instruction queue:** it's mainly a fifo queue that stores instructions coming from the memory unit, it has 7 consecutive stages (which is a good number of phases technically), it is related to the number of cycles needed to execute an instruction in this machine (taking in consideration that a general format instruction will take at least 2 bytes). The main goal of this queue is to improve CPU performance by allowing the CPU to fetch and decode multiple instructions at once, reducing the time it takes to fetch and execute each instruction individually.
    - **The queue controller**: the operation of pushing and popping instructions to and from the queue is happening between two independent entities, and this can cause access conflicts so it's supervised and commanded

by this controller, which is communicating with the sequencer, this process can be described as follow: This controller gets a signal from the sequencer to read from the queue an instruction so it can be decoded and then executed, the controller opens the gateway and allows the sequencer to read from the queue, then it requests the next instruction from the memory unit.

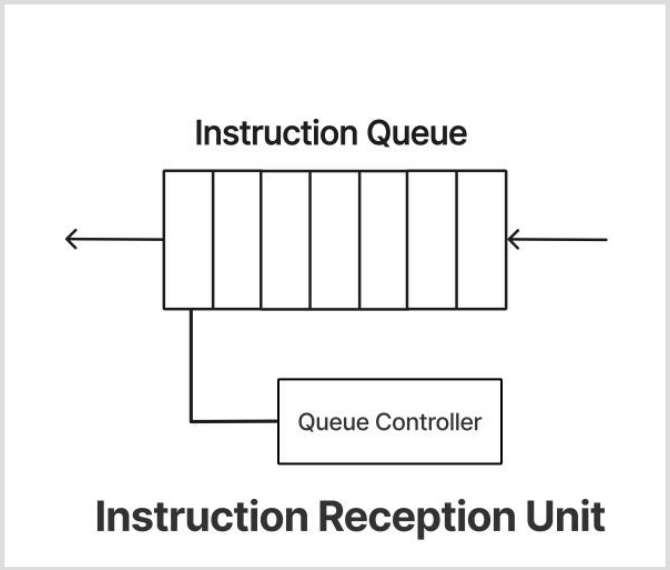


Figure 6: Instruction Reception Unit

- **The sequencer:** is the controller that command and organize all the operations happening in the cpu, it's the bandmaster in the machine, it's connected with all the units and components and manages all the data transfer between all the components, by controlling the buses and the gateway of each component, and communicates with different controllers, like queue and I/O controller. For the type of the sequencer, in our calMachine we have no purpose of specifying its type (cabled or programmed), since it's a concept that won't be teached in the computer's architecture course we are pointing **"Archi1"**, we can make it more specific in the upcoming versions if needed.

The instruction flow in calM's command unit follows the well-known concept of fetch, decode and execute (which is also used in the 8086 Intel CPU).

Here's how it works:

1. The controller fetches the instruction from the memory unit and puts it in the queue.
2. The decoder decodes the instruction by putting it in the instruction register (IR).
3. The sequencer executes the instruction based on the decoded information.

All of the steps are managed by the sequencer, which plays a crucial role in the process

2.Processing Unit

This is the unit where the calculations are happening in the cpu, the command unit mainly operates on this unit because it contains the **ALU** (calculations unit) and the registers needed to perform the operations:

- **ALU:** Arithmetic and logic unit, it is the union of many combinatorial circuits, that perform logic or arithmetic operations on 1 or 2 bytes words, equipped with a **MUX** that allows the sequencer to choose the operation to perform and its type, it's equipped with **4 registers of 2 bytes:**
  - **2** in the entrance gate of the unit: those registers aren't directly accessed by any of the instructions, but they are specifically used to simplify and speed up the execution process of 2 operands operations performed in the **ALU**, by storing the operands in these two registers, if it's a one operand instruction, the operand will be stored in **RUAL1** by convention.
  - **1** Accumulator register of where the ALU puts the result of the performed operation.
- **Registers:** a total of 7 registers of 2 bytes, those registers are considered to be part of the memory unit because their mission is storing operand while calculations are happening in this unit:
  - **R1, R2, R3, R4:** are general purpose registers, we use them in order to store data while doing the necessary operations, the first three are divided into 2 parts low and high, **R4** a complete 2 bytes register is used to store the carry of an operation that gives a result with a size bigger than 2 bytes (it can't be stored in the ACC, the multiplication of two bytes operands can give 4 bytes result), the least significant 2 bytes are stored in the accumulator and the most significant 2 bytes are stored in the **R4**.
  - **SR:** Stack register is the register where we store the address of the top of the stack, implemented in the CM.

- **IDR:** Index register is used in the indexed addressing mode.
- **BR:** Base register is used in the based addressing mode.

- **Flag Register:** it is a 1 byte size register that describes the state of the accumulator register content and the state of the cpu as well, each single bit representation will be explained, from right to left As illustrated in figure7 (from least significant bit to the most significant bit):
  - **Zero flag**
  - **Overflow flag**
  - **Carry flag**
  - **Sign flag**
  - **Pair impair flag**
  - **Parity flag**
  - **Interruption flag**
  - **I/O operation flag**

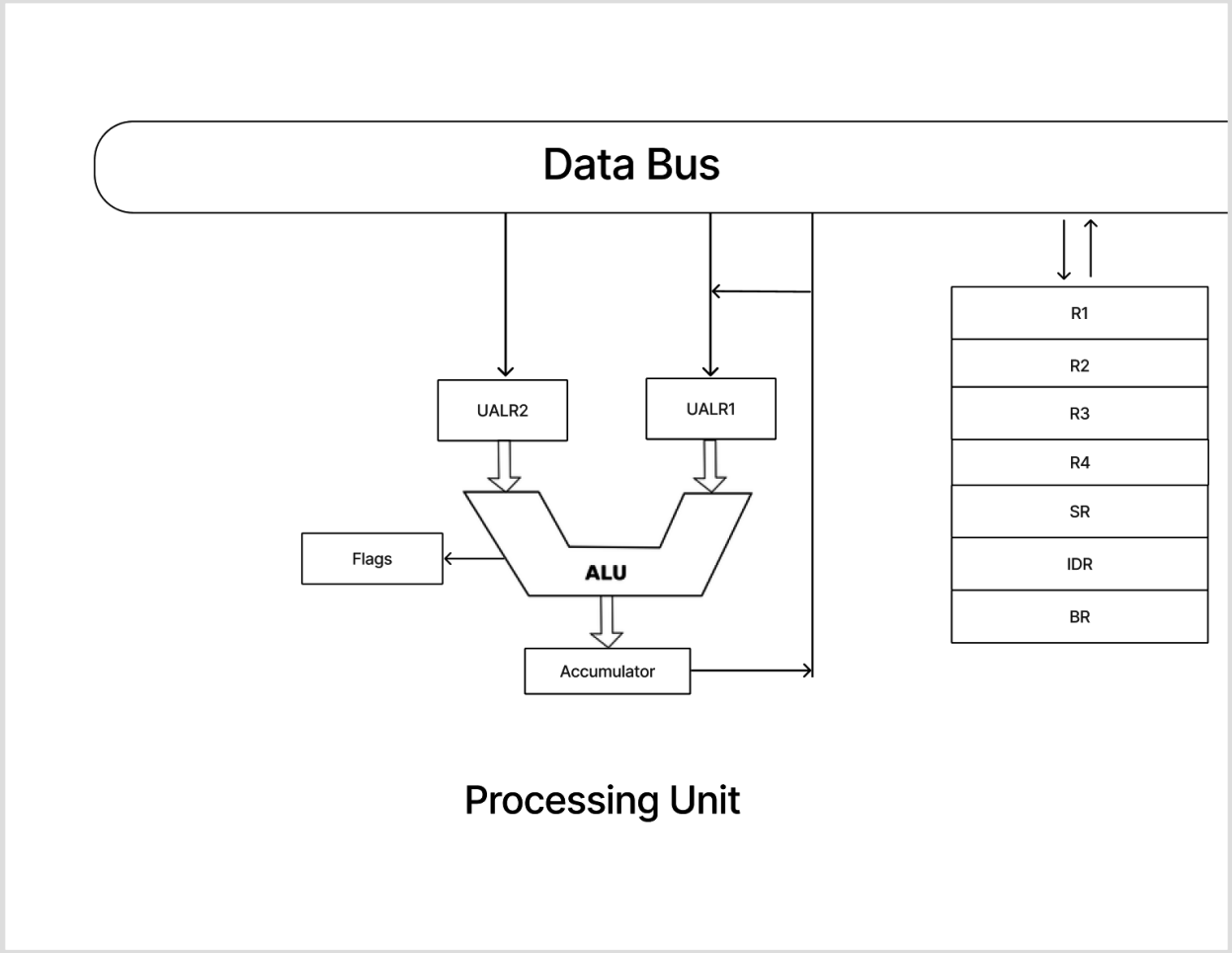


Figure 7: Processing Unit

Zero flag	Sign flag	Carry flag	Parity flag	Pair Impair flag	Overflow flag	Interrupt flag	I/O flag

Figure 8: Flags Register

3.I/O Unit:

It's the unit responsible of getting data to the CPU from the outside, and vice versa, it's connected directly to the data bus, and it contains:

- **Buffer:** a set of 2 bytes 4 registers where the transferred data is stored.

- **I/O controller:** a control unit that supervises the transfer process, it communicates with the sequencer and the I/O interface(the set of controllers for each port or peripheral), and it's connected with the buffer, so it can write or read data from it.

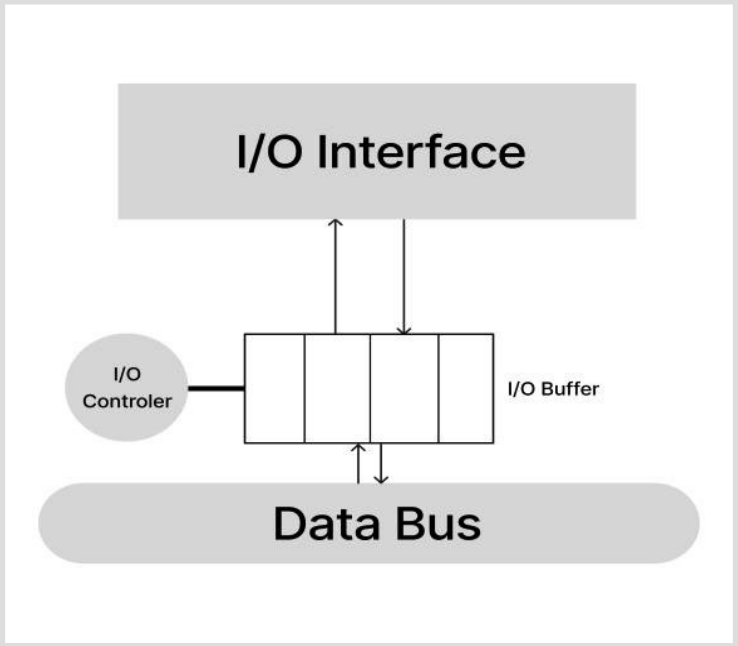
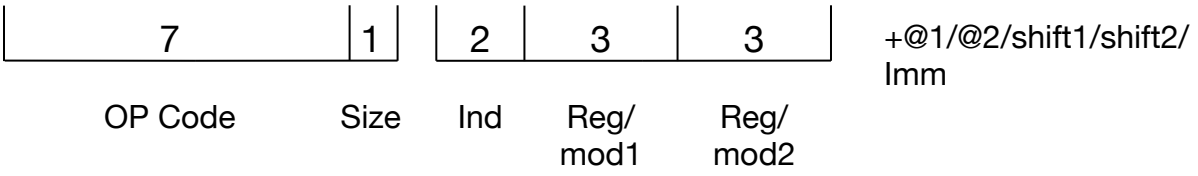


Figure 9: I/O Unit

IV.Instructions Set Architecture

1.General Format



- **OP Code:** Operation code for the instruction.
- **Size:** is the size of the operands:
  - 0: Operands are on 8 bits
  - 1: Operands are on 16 bits
- **Ind:** Indicates the type of the operands, “0” for the registers and “1” for memory:
  - 00: Register - Register
  - 01: Register - Memory
  - 10: Memory - Register
  - 11: Memory - Memory

- **Reg/Mod1:** If it corresponds to a register (i.e. Ind=0X) than we will put the name of the register in it, otherwise we will put the corresponding addressing mode.
- **Reg/Mod2:** Same thing as Reg/Mod1.

Note: The jump instructions will only use the first byte (OPCode + Size = 1) and the jump address will be held on the remaining 16 bits.

2.Reduced Format

Used for one operand instructions which are: NOT, NEG, SHL, SHR, READ, WRITE, PUSH, POP, ROL, ROR.

+ address (if it exists) on 16 bits

OP Code Reg/Mod Size

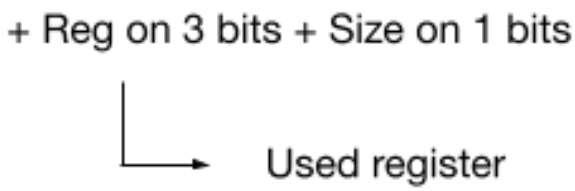
3.OPCode for general format instructions + size bit

Instructions	OPCode	Size	Operands
ADD	0000000	1 or 0	2 operands (reg ou mem) / Check Ind bit
SUB	0000001	1 or 0	2 operands (reg ou mem) / Check Ind bit
MUL	0000010	1 or 0	2 operands (reg ou mem) / Check Ind bit
DIV	0000011	1 or 0	2 operands (reg ou mem) / Check Ind bit
AND	0000100	1 or 0	2 operands (reg ou mem) / Check Ind bit
OR	0000101	1 or 0	2 operands (reg ou mem) / Check Ind bit
XOR	0000110	1 or 0	2 operands (reg ou mem) / Check Ind bit
NOR	0000111	1 or 0	2 operands (reg ou mem) / Check Ind bit

Instructions	OPCode	Size	Operands
NAND	0001000	1 or 0	2 operands (reg ou mem) / Check Ind bit
CMP	0001001	1 or 0	2 operands (reg ou mem) / Check Ind bit
MOV	0001100	1 or 0	2 operands (reg ou mem) / Check Ind bit
CALL	0011001	Only 1	1 adresse operand (label)
RET	0011010	Only 1	No operands
PUSHA	0010000	Only 1	No operands
POPA	0010001	Only 1	No operands
BE	0010010	Only 1	1 address operand (label)
BNE	0010011	Only 1	1 address operand (label)
BS	0010100	Only 1	1 address operand (label)
BI	0010101	Only 1	1 address operand (label)
BIE	0010110	Only 1	1 address operand (label)
BSE	0010111	Only 1	1 address operand (label)
BR	0011000	Only 1	1 address operand (label)

4.OPCode for reduced format instructions

Instruction	COP sur 4 bits
NEG	0100
NOT	0101
SHL	0110
SHR	0111
READ	1000
WRITE	1001
PUSH	1010
POP	1011
ROR	1100
ROL	1101



5.Ind bits + Reg/Mod bits

Ind Combinations	The Operands	Reg/mod1	Reg/mod2
00	Registre - Registre	Number of the first register	Number of the second register
01	Registre – Memory	Register number	Addressing mode of the used address

Ind Combinations	The Operands	Reg/mod1	Reg/mod2
10	Memory – Registre	Addressing mode of the used address	Register number
11	Memory - Memory	Addressing mode of the first address	Addressing mode of the second address

6.Register's code

For Size bit = 1

Register	Register's Code
R1	000
R2	001
R3	010
R4	011
ACC	100
BR	101
IR	110
SR	111

For Size bit = 0

Register	Register's Code
R1R	000
R2R	001
R3R	010
ACCR	011
R1L	100
R2L	101
R3L	110
ACCL	111

7.Addressing modes

Mode	Code
Immediate	000
Direct	001
Indirect	010
Based	011
Indexed	100

Mode	Code
Based indexed	101
Shift on 8 bits	110
Shift on 16 bits	111

Note: For the memory memory operations, there's some combinations that aren't possible:

- Immediate for the first operand.
- Based for the first mode and based or based indexed for the second mode (because we only have one base register).
- Indexed for first mode and based and indexed for the second mode (because we only have one indexed register).
- Based indexed for the first mode and based or indexed for the second mode.

V.Execution Process in the calM

The binary code is stored in the external memory, we will just simulate the execution of one instruction, the first instruction is on 16 bits it is present as an input to the cpu, so the request come to the I/O controller to open the gateway towards the I/O unit, when the instruction is present it is written in the buffer, then the sequencer requests the data to be written in the CM, the I/O controller opens the gate to the data bus, then the data words will go to MDR (memory data register) so that it can be written in the CM, the sequencer writes the address 0x0000 in the MAR (memory address register), and then perform the write operation in the CM, this phase will continue until the binary code is all written in the memory, and at each time the sequencer will increment +1 the MAR value compared to its value in the previous instruction so the instruction will be written one after the other, so it goes address 0x0000, 0x0001, 0x0002..etc.

That was the phase of writing code in the CM, now we go to execute one instruction written in the address 0x0000.

First the sequencer writes 0x0000 in the MAR through the address bus, and performs a read operation so it collects the value from DMAR and through the data bus it goes to the command unit, and that was the first phase of execution: **searching the instruction**.

Second the sequencer sends a signal to the queue controller in order to fetch this instruction, the controller accepts since there is enough space: **Instruction fetch phase**.

The sequencer then asks the controller to prepare another instruction for fetch and liberate the first instruction to the IR, where it is stored to pass to the decoder that decodes the 2 bytes binary code: that was the **decoding phase** and gives the result to the sequencer that is able to execute the instruction encoded in that 2 words byte, so it does the necessary operations in order to perform it: **execution phase**, which is different from machine to machine.

VI.Appendix

Since this documentation is dedicated to every computer science student, electrical engineering student or anyone that can benefit from it but also and mainly for the first year students that are new to this field, it has to be explained with more details and many concepts has to be defined since they may seem very difficult for the first time.

- **Buffer**: A buffer is a temporary storage area in a computer's memory used to hold data while it is being moved from one place to another.
- **CISC**(Complex Instruction Set Computer): is a type of computer architecture that emphasizes a large number of complex instructions that can perform multiple operations, CISC processors have a wide variety of instructions, which can perform operations like memory access, arithmetic operations, and logical operations in a single instruction. This makes programming easier and faster but can make the processor more complicated and slower. Examples of CISC architectures include the Intel x86 and Motorola 68K series of processors.
- **Von Neumann architecture**: Von Neumann architecture is a type of computer architecture design where both program instructions and data are stored in the same memory and share the same communication bus. The architecture was proposed by John Von Neumann in the 1940s and became the basis for most modern computers.
- **Register**: a register is a small, fast storage area used to store data that the CPU needs to access quickly during processing.
- **General purpose register**: is a register in a computer processor that can store any type of data and be used for any purpose.
- **Cache memory**: Cache memory is a small and fast memory that stores frequently used data to improve the performance of the computer system. It acts as a buffer between the CPU and main memory to reduce the time it takes to access data.
- **Stack implementation in Central Memory**:A stack is a type of data structure in computer science that stores and retrieves data items in a specific order. It works on the principle of last in, first out (LIFO).In a CPU architecture, a stack can be implemented in the central memory, cpu uses a special register called the stack register to keep track of the memory location of the top of stack, this stack grows from down of CM to up, we have its top in an address then the stack will be after this address, so when the cpu needs to push data onto the stack, it first decrements the stack register to reserve memory space for the data. The data is then written to the memory location pointed to by the stack pointer. When the program needs to pop data from the stack, it first reads the data from the memory location pointed to by the stack pointer and then increments the stack pointer to release the memory space.