UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

International semester

Designing of a Basic Monocycle MIPS microprocessor in a FPGA

Autor:

Le Gal Yoann

Tutor:

Caceres Santiago

Electricity

Abstract :

This project is about microprocessor so what is a microprocessor ? A microprocessor is a processor (something who execute instruction) wich all componants are small enough to go in only one small box. It's mostly composed of transistor, invented in 1947, and the first integrated circuits was invented in 1958. At the begining the microprocessor was composed of 4 bits then 8 bits then … and now 64 Bits. To make todays microprocessor we need a lot of part but in this case we need a PC (program counter) unit , an Instruction memory, a Register File, an ALU (aritmectic logic unit), a data memory, some multiplexor, some adder and a sign extension part. We'll see they fonctionnement in this memory. To make all that part working we need to use VHDL code and to see how it work we need to simultate it.

Key words :

Microprocessor, MIPS, FPGA, VHDL, Evolution

Summary

Introduction :

Nowadays we are living in a digital world. So we need a lot of computer and electonic device to go forward and to progress in technology and in innovation in general. We use more and more robot and automatic device (car, factory machine, etc..). So we need to progress in this field and to adapt and prepare (by teaching basic thing in our population) our society to this type of device.

The work is to design a MIPS and to learn a brand new thing so objectives of this final degree project are

Design a Basic Monocylcle MIPS microprocessor in a FPGA :

→ Aquire new knowledge
. Learn about microprocessors (programmable)
. Coding a new type of processor (Hardware) and then learn new type of language.
.
→ Improve and going deeper in some kind knowledge (I learned a lot of thing in electronic during my study and with this final degree project I understand why I have done some thing and how thing

→ Learn by myself a brand knew thing in a knowing field

Learn to write a memory (because I'm only 20 and I have to write my memory in 2 years)

Techicals objectives :

→ Implement a VHDL code in a FPGA and simulate it.
. Code and test each device of the MIPS
. Analyse simulations of the device and understand all the error

In this memory I will begin to introduce the microprocessor. This part will be about the history, the evolution, and the fonctionnement of the microprocessor. It's very important to understand this part to following parts of this work.

Then I will explain what I learn to reach my objective and finish this work. It'll begin by a theoritical part and then it'll we have a practical part (the code of the differents part of the MIPS) with some explaination.

And to end this memory I'll show you my final work and explain my result. So all the Mips, the simulation and the implementation in the FPGA.

I choose this subject because in the future i would work in embedded car system so it's very intresting to know how to build this type of device.

The Microprocessor

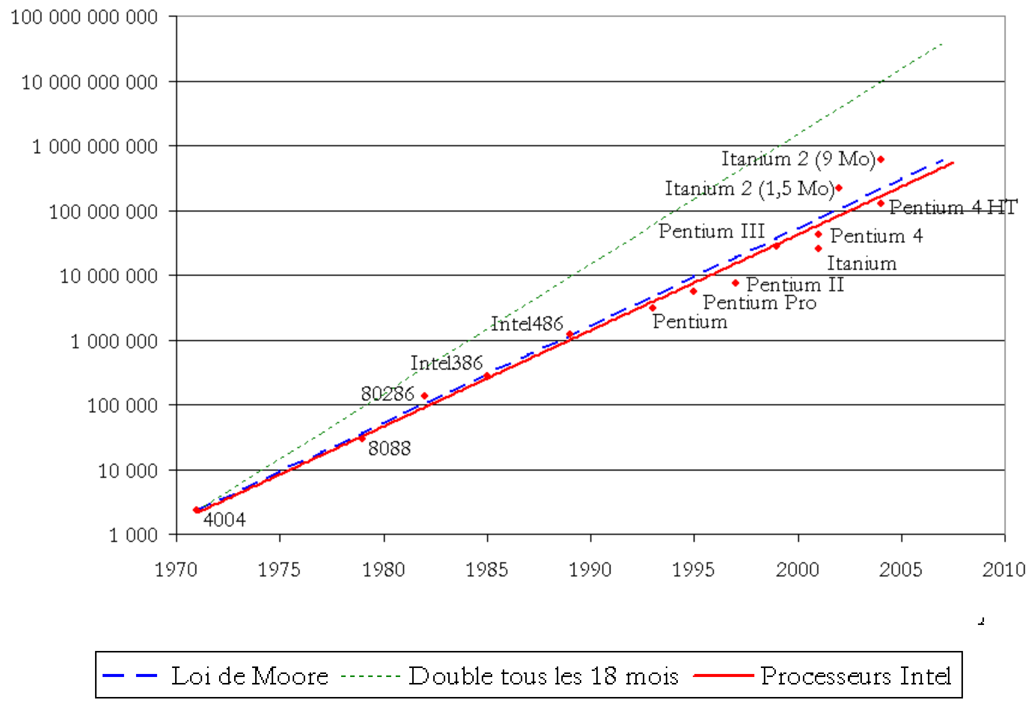
I) History of the micropressor :

So now let's talk about the hisory of the micropressor. To do this part I will mix three very intresting documents, one is a course of M. DALMAU, professor of tecnologic institute of Bayonne (France) the other is a course too of an engineering school of Rennes (France) adn the third is a university course too.

So since thousand of year we use aritmetic calcul for the trade, administration, etc... during the history we founded some tools to calculate like addition or algorithm (euclyde algorithm). We also invented an abacus to help to calculate and the first calculate machine was invented in 1642 by Blaise Pascal (for addition and substracion). So electronic processor appear in the 20th century. At the first time analogic processor have been invented but because of the flexibility not important (because we had to create a new circuits for every operation) and we couldn't resolve all the problem, we invented the coding numeric processor in wich we can change the code without change the circuit (analogic processor still exist but they realize easy task like an "+" or a logical "and"). So now let's go deeper on the coding numeric processor and then on analagogic processor.
The history of microprocessors is closely linked to the history of semiconductor technology. Let's see a couple of important dates

- 1947 Invention of the transistor
- 1958 TEXAS INSTRUMENTS produces the 1st integrated circuit (IC)
- 1961 Development of TTL and ECL Bipolar Technologies
- 1964 Small scale integration (SSI from 1 to 10 transistors)
- 1965 Medium scale integration (MSI from 10 to 500 transistors)
- 1970 Development of MOS technology
- 1971 Large scale integration (LSI from 500 to 20,000 transistors)
- 1985 Integration on a very large scale (VLSI more than 20 000 transistors)

Has we can see the number of transistor rise up very fast so let's take a look of the Moore law."The number of transistor on a silicon chip will double every years" (Gordon Moore,1965) and we know that the surface of a transistor will be divide by two every two year so we obtain theis chart :



<u>Rise of the Number of transistor on chip of silicon threw years</u>

Now let's see the real evolution from first microprocessor to microprocessor of todays. In the first time we'll talk about the 4 and 8bits microprocessor.

1) 4 Bits micropressor :

Manifacturer	INTEL 4004	INTEL 4040	ROCKWELL PPS4	FAIRCHILD PPS25	TEXAS INS TMS 1000
Number of instruction	46	60	50	95	small
Time between registers	8	8	5	3	15
Memory space	4K	8K	4K	6,5K	1K
General using register	16	24	4	1	4

The first microprocessor was the Intel 4004. It was created in 1971 (and destinated to equipe office calculator)He integrated 2250 transistor and had got a 740 Khz clock. His RAM was about 1Kb.
The 4004 has 16 4-bit registers that can also be used as 8 8-bit registers. he manages subroutine calls through a 4-level internal stack.
The 4040, dating from 1972, adds 4 levels to this stack as well as 14 new instructions and the management of interruptions.

2) 8-bits micropresseur

For the 8-bits microprocessor a lot of builder came on "the business" of the micropressor. As 4-bits micropressor we obtain this table :

manifacturer	INTEL 8008	INTEL 8080	INTEL 8085	MOTOROLA 6800
Number of instruction	48	69	71	71
Time between registers	12,5 à 20	1,3 à 2	1,3	2
Memory space	16K	64K	64K	64K
General using register	7	7	7	3
Number of transistor	3300	4000	6200	4000
Clock (MHz)	0,3	2 ou 2,67 ou 3,125	3,5 ou 6	1 ou |1,5 ou 2
Year of creation	1972	1974	1976	1974

Manifacturer	ZILOG Z80	MOS Technologies 6502	ROCKWELL PPS8	NATIONAL SC/MP
Number of instruction	69	71	90	50
Time between registers	1,6	2	4	5 à 25
Memory space	64K	64K	32K	64K en pages de 4K
General using register	17	3	3	6
Year of creation	1976	1975	1976	1976

"The 6800 has, in terms of registers, the strict minimum that is to say an ordinal counter, a stack pointer, a index register, two accumulators and a condition code register. It processes the following 8-bit operands: - Natural number (0 to 255) - Relative integers in representation complement (-127 to 127) - Decimals in DCB on 2 digits (8 bits) The 16-bit operands can only be processed by successive "slices" of 8 bits. However, it can be noted that the index (on 16 bits) can be transferred to and from 2 consecutive bytes in memory as well as increments, decrementations and comparisons of equality with a word consisting of 2 consecutive bytes in memory. The address bus is 16 bits so the memory space can not exceed 64K bytes. The 6800 has the modes of direct addressing (operand designated by its address in memory) and indexed (operand designated by an address obtained by adding the contents of the index register and the value placed in the instruction. The displacement is fixed and consisting of a natural 8-bit integer). The sequencing clock of the 6800 is 1 MHz (later models 68A00 and 68B00 will support 1.5 and 2 MHz clocks). Every instruction lasts at least 2 clock periods and none exceeds 12 periods (most of the instructions are between 2 and 6 periods). Unlike microprocessors of INTEL and ZILOG the operation of the 6800 is biphasic ie during the first half of the period the internal operations (instruction decoding, UAL, register transfers) and during the 2nd half exchanges with the memory. The operation is totally synchronous ie the memory must be able to "answer" during the half-time allocated to him and there is no simple way to wait for the microprocessor if the memory is too slow" (M DALMAU, University of Bayonne, France, 2007).

<u>**II) Evolution of microprocessor :**</u>

With the time 8 Bits micropressor reach their own limits. Their default are:

A Lack of memory space (64Kbits)
Just a few type of manipuled information (1 to 2 Bytes)
Not adapted of microprocessor achitecture
Just a few adressing mode.

For all these reasons, manufacturers are starting to study a new generation of microprocessors. Of so that we will see more powerful 8-bit microprocessors that make it easier to manipulate information on 16 bits and offering new addressing possibilities such as the MOTOROLA 6809 or the8085 from INTEL.

The use of more and more "computer" microprocessors (advanced languages, operating systems performance) quickly attracted manufacturers to products closer to the central computer units.(1)

<u>**III) Evolution of the architecture :**</u>

One of the first improvements has been to increase the size of the data bus and operands and the introduction of new types of addressing (indirect, post-incrementation and pre-decrementation). Then were introduced the long-standing concepts used on virtual memory computers and privilege levels. Then, the degree of integration increasing considerably, appeared the caches making it possible to accelerate access to instructions and operands. Finally, we have seen a real parallelism in the implementation of instructions requiring the establishment of sophisticated prediction methods to search in advance for instructions and operands that they must deal with and to solve the scheduling problems posed by the parallel execution of instructions. Finally, it is by the integration of several processors on the same chip that we will increase the performances from 2002 as in the POWER4 of IBM (2 processors).

1) <u>Data size</u>

As early as 1976, the Texas Instruments TMS 9900 proposed the manipulation of 16-bit operands. The 16-bit microprocessors lasted until the mid-1980s, by which time the 32-bit appeared. The 64 bits appeared in the early 90s and tend to become widespread. Subsequently, with the appearance of this size has become less significant since the accesses are always made from the cache while the exchanges between the memory and the cache can be done on larger buses to make them faster (currently 128 or 256 bits).

2) <u>Virtual memory and paging</u>

The size of the memories grew significantly, which quickly made it go from a few to
a some hundred MB. Meanwhile the size of the microprocessor address bus has been considerably increased (see table). The implementation of a virtual memory management

quickly spread so that the addresses seen by the instructions are not those put on the physical address bus of the microprocessor.

Type	Year	Real adress	Virtual adress
INTEL 4004	1971	12 bits (4 Kb)	No
INTEL 8080	1974	16 bits (64 Kb)	No
INTEL 8086	1978	20 bits (1 Mb)	No
MOTOROLA 68000	1979	24 bits (16 Mb)	No
INTEL 80286	1982	24 bits (16 Mb)	30 bits
INTEL 80386	1985	24 bits (16 Mb)	46 bits
MOTOROLA MPC601	1993	32 bits (4 Gb)	52 bits
INTEL Pentium	1993	32 bits (4 Gb)	46 bits
MIPS R12000	1999	40 bits (1 Tb)	64 bits
MOTOROLA MPC620	1996	40 bits (1 Tb)	80 bits
HP PA-8000	1996	40 bits (1 Tb)	96 bits
INTEL Pentium 2	1999	36 bits (64 Gb)	46 bits
HP PA-8700	2001	44 bits (16 Tb)	96 bits
Itanium2	2002	64 bits	64 bits

3) Privilege level

In order to secure the operating systems written for these microprocessors, it was quickly necessary to protect certain instructions and access to memory. To do this we define levels of privileges which the microprocessor is working on and which instructions can be of his the virtual memory manager is responsible for protecting access. We find this type of working very early on the 68000 (1979) realized in the following way: The processor knows 2 levels of privilege: user and supervisor. He distinguishes them by the value of a bit of the state register. The transition from the supervisor level to the user level is obtained by any instruction having as effect of changing this bit of the status register. The transition from the user level to the supervisor level won't take as place as taking into account an interrupt because the instructions allowing the modifications of this bit are prohibited in user mode.
A little later the system is improved by increasing the number of levels so as to have more management fine security.

Thus one finds on the 80286 (1982) four levels of privilege whose effectiveness is increased by the management of a virtual memory:

level 0: kernel of the operating system

level 1: operating system primitives

level 2: operating system extensions

level 3: applications

4) <u>The cache</u>

Advances in semiconductor technology have made it possible to increase the clock speeds of microprocessors on a regular basis. Starting from a few KHz in the early 1970s we reached a few MHz in the mid-70s and then the dozen MHz in the early 80s, the 50 MHz in the late 80s, the hundred MHz in the early 90s for to reach the GHz at the end of the century and to exceed it from the year 2000. However the speeds of the memories did not follow the same rhythm and it was very quickly necessary to introduce memories fast to avoid having to slow down the microprocessors. As early as 1979, the ZILOG Z8000 had a cache of 256 bytes that could be used for data, the instructions or both. Likewise a 64 bit 32 bit short cache equipped the 68020 with 1984. It was only used to contain instructions which avoided problems of consistency between cache and memory since there was no cache write. However, the following models come out with internal caches. On some the cache is unique and contains both data and instructions. On others it is divided to allow simultaneous access instructions and their operands. Second-level, first external, caches began to be more and more integrated. However he was still difficult to integrate large sizes of fast memory on the same chip as the microprocessor which led, for example, INTEL to propose components on 2 chips encapsulated in a housing (Pentium Pro, Pentium II and III). Currently third level caches are sometimes embedded on the same chip and reach sizes of a few MB.

Year	Unique cache	Separated cache	2nd level cache	3rd level cache
1985		0 + ¼ Ko		
1986		4Ko + 4Ko		
1989	8 Ko 1octet=1byte	4Ko + 4Ko		
1991	16 Ko			
1992		8Ko + 8Ko		
1994		16Ko + 16Ko	256 Ko	2 Mo
1996	abandoned	32Ko + 32Ko	512 Ko	8 Mo
1998		64Ko + 64Ko	1 Mo	
2000			2 Mo	
2002			6 Mo	
2004			8 Mo	32 Mo
2005				64 Mo

→ Cache operation :
-Writing: For the data cache, you can use the "write through" method, that is to say
that when a write takes place, it is done both in the cache and in the memory. The written data is available in the cache for the following instructions even if the actual write in memory does not have still occurred.

-Filling: It is done in blocks (burst mode). This leads to the production of memories specially adapted to this type of access.

-Replacement: To choose which blocks can be used and which operations are needed (writing in memory if the block has been modified, synchronization update in multiprocessor operation ...) each block in the caches can be in one of the following 4 states:

- invalid (its content is no longer valid)
- exclusive unmodified (unshared and unmodified)
- modified exclusive (not shared but has been modified)
- shared (there are other copies in other caches)

→ Cache control :
There are some instructions to control how to use caches. They are particularly
useful when multiple processors share the same memory and could have in their caches different (unsynchronized) versions of the information.
In particular, they make it possible to choose whether or not to update the cache at each write to a memory address whose image is in the cache, to release one or more blocks of cache, lock cache contents (no more block loading), and allow or to prohibit the use of the cache.

5) Reals numbers

The first microprocessors manipulated only whole numbers for which they offered only the addition and subtraction operations. The multiplication and division operations only appeared from the end of the 70s. The treatment of reals was then left to the programmer who was to write procedures to that effect. Of course, this deficiency was soon to be filled. The first proposed solution was to use an arithmetic co-processor that is to say a component to which the microprocessor proceed calculations concerning real numbers. The first microprocessor so equipped is probably the AMD 9511 in 1979. These co-processors have subsequently disappeared to be integrated into the microprocessor processing unit as soon as the late 1980s. Current processors process 82-bit reals of which 17 for the exponent according to the standard ANSI / IEEE.

6) <u>Parallelism of execution of the instructions</u>

In order to speed up the execution of the instructions, the manufacturers have gradually introduced modules capable of processing in parallel some of the actions of the microprocessor. We will now describe these improvements.
The first 16-bit microprocessors introduced the predictive of reading instructions and putting them in a queue waiting for execution. Thus 8086 (1978) consists of 2 units:

- The bus management unit (BMU)
- The execution unit (EU)

The BMU is responsible for:

- finding instructions and queuing them
- reading and writing operands
- calculation of addresses in memory
- physical control of memory It works in total parallelism with the EU.

The EU draws the instructions from the 6-byte queue and uses the BMU to get its operands from the memory and to store its results. All addressing problems are therefore managed by the BMU. Parallelism will be improved from model to model by introducing units whose role is to prepare the work others. Thus the 80286 (1982) will be endowed with 2 new units:

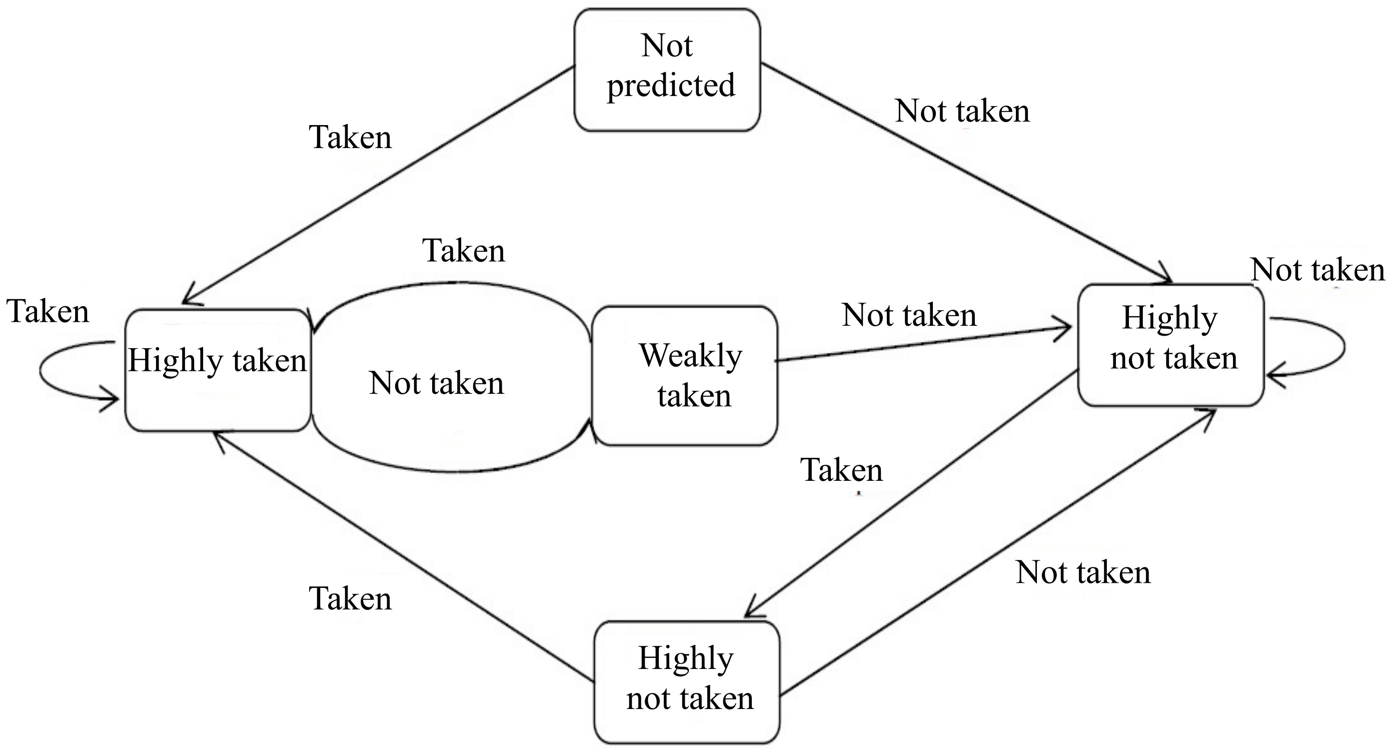
- instruction unit (IU)
- address unit (AU)

The IU instruction unit draws on the 6-byte queue of the BMU to maintain a queue of 3 decoded instructions. The AU address unit calculates the physical addresses both in real mode and in virtual mode. Later, on the 68020 (1984), the execution units will be manufactured according to a pipe-line architecture allowing multiple statements to overlap.
The following table gives, for some processors, the depth of the pipe-line of execution of the instructions as well the maximum number of instructions that can be executed simultaneously with this device.(2)

Name of the processor	**Depth of the pipepline execution**	**Number of parallel instruction**		**Name of the processor**	**Depth of the pipepline execution**	**Number of parallel instruction**
MIPS R10000	7	4		Motorola G4e	7	2+1 connection
DEC alpha 21264	6	6		Motorola G5	10	2+1 connection
SUN UltraSparc II	7	4		INTEL Pentium II	10	3
SUN UltraSparc III	14	4		INTEL Pentium III	10	3
Motorola G3	4	2+1 connection		INTEL Pentium IV	20	3
				INTEL Pentium 4EE	31	

→ connection prediction :

The advance preparation of instructions poses the problem of the accuracy of the prediction. This problem is the one posed during a conditional branch instruction. Indeed, the search device reads in advance the instructions in sequence. But, during a connection, it is impossible to know in advance whether he will have place or not.
The devices used at first were content to act as if the connection should not
take place then, if necessary, stop the execution pipeline and empty the queue of instructions, for restart the search from the new address but such an operation requires several clock cycles.
In order to prevent such a situation from occurring too often, the method generally used is:
The instruction search unit reads the instructions in sequence until it encounters a conditional connection . When this happens it uses a branch prediction unit that tells it if it must continue searching in sequence or from the address specified in the branch instruction.



Exemple of a 5 states automaton

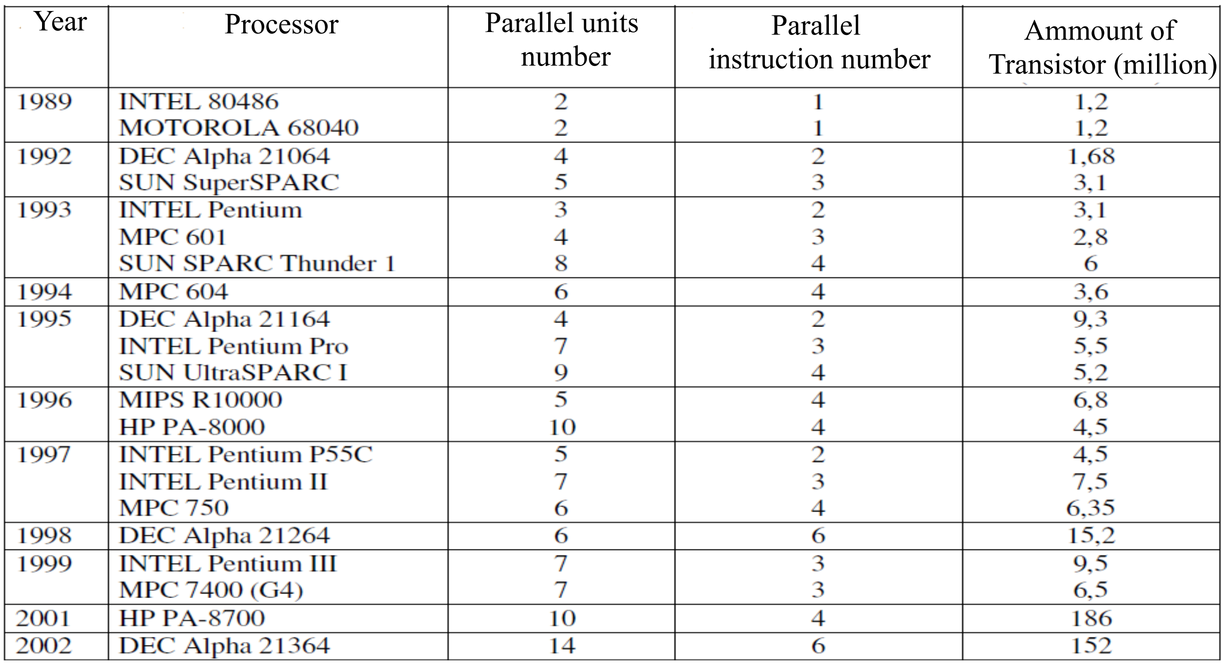
Each time a branch instruction is executed, its state is updated in the history based on whether the connection took place or not. When the connection corresponds to 'unpredictable' state, considers that it will take place if it refers back to an address and that it will not happen if it sends to an address in before. When the instruction address generation unit encounters a branch, it uses a memory associative containing information on the last connections encountered to find the history of this connection. If the connection has no history yet, it is in the 'not predicted' state. With the help of thist, this method produces a prediction that is correct in 90% of cases.

→ Execution parellelism:

The basic principle is to have several identical or partially specialized processing units and attempt to distribute the previously decoded instructions between these units. The first microprocessor to realize this is probably the 32732 of National Semiconductor in 1991. We usually meet:

1 or more units that support read and write operands
1 connection processing unit (see above)
1 or more calculation units on integers
1 or more units of calculation on the real ones

Most current processors can distribute multiple instructions in parallel when, of course, the availability of processing units allows it.



Year	Processor	Parallel units number	Parallel instruction number	Ammount of Transistor (million)
1989	INTEL 80486 MOTOROLA 68040	2 2	1 1	1,2 1,2
1992	DEC Alpha 21064 SUN SuperSPARC	4 5	2 3	1,68 3,1
1993	INTEL Pentium MPC 601 SUN SPARC Thunder 1	3 4 8	2 3 4	3,1 2,8 6
1994	MPC 604	6	4	3,6
1995	DEC Alpha 21164 INTEL Pentium Pro SUN UltraSPARC I	4 7 9	2 3 4	9,3 5,5 5,2
1996	MIPS R10000 HP PA-8000	5 10	4 4	6,8 4,5
1997	INTEL Pentium P55C INTEL Pentium II MPC 750	5 7 6	2 3 4	4,5 7,5 6,35
1998	DEC Alpha 21264	6	6	15,2
1999	INTEL Pentium III MPC 7400 (G4)	7 7	3 3	9,5 6,5
2001	HP PA-8700	10	4	186
2002	DEC Alpha 21364	14	6	152

Table of the evolution of the microprocessor in function of their parallelism

→ Data dependance and rename of registers

Parallel execution of instructions raises the problem of the execution order. Indeed, when two instructions are launched simultaneously it may happen that the one that had the rank i + k in the program ends before the one who had rank i. But this can produce false results when there is a data dependency between these 2
instructions. We distinguish 3 cases:

-Write After Read: The instruction of rank i + k written in an operand that is read by the instruction of rank i. Avoid that of rank i + k does not end before the other.

-Write After Write: The instructions of rank i and i + k write in the same operand. This is the same problem as above.

-Read After Write: The instruction of rank i + k reads an operand written by that of rank i. We must then wait for that of rank i is completed to start that of rank i + k.

To avoid this problem one can, when one detects such a situation, to prohibit the execution in parallel of instructions concerned. This is the solution adopted on the first microprocessors using parallelism execution. However, the registry renaming method offers a more efficient solution: The operands of the instructions are the registers of the machine that the set of instructions allows to designate. In reality the processor has a higher number of registers. When decoding an instruction it assigns one of registers available to each of those designated in the instruction. The first two cases (Write After Read and Write After Write) are then simply resolved by avoiding assigning the same registers to the two linked instructions. Only the last case then requires the sequential execution of the two instructions.

7) <u>Parallelism of execution processes</u>

The establishment of several processing units (superscalar) allows the simultaneous execution of instructions. However, because of the links that generally exist between the successive instructions of a program, it happens quite often that this parallelism can not be implemented. The principle is to do so that the processor is seen by the operating system as several virtual processors capable of operate in parallel. To obtain this result, we duplicate:

- The ordinal meter
- The instruction search unit
- The register renaming device (see parallel execution of the instructions)
- Some registers (in particular related to stack management) .

Each instruction search unit, linked to its ordinal counter, feeds the instruction queue.The instructions are marked so that we can know which process they belong to. Their execution processing units shared by all processes but using the
registers associated with the process in question so that they appear to have separate registers for each virtual processor. Although the instructions from the various processes are mixed in the same instruction queue, we make sure the queue contains instructions for all processes. This is to prevent certain processes
do not occupy the entire queue and prohibit the simultaneous execution of others.
Threads are shared between processes.

So all this part related wit h history and evolution was made with a mix two very intresting documents, one is a course of M. DALMAU, professor of tecnologic institute of Bayonne (France) and the other is a course too of an engineering school of Rennes (France).

IV) Todays microprocessor

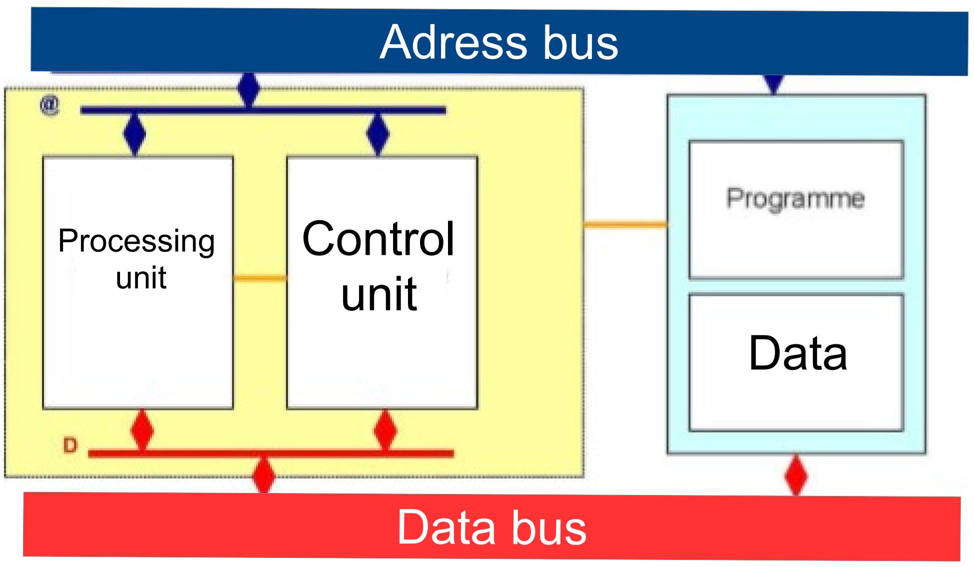
So now after have seen the evolution of a microprocessor let's see how it work.
To begin I'll give you a definition of what is a microprocessor :
"A microprocessor is a complex integrated circuit characterized by a very large integration and endowed with the ability to interpret and execute program instructions. It is responsible for organizing the tasks specified by the program and ensuring their execution. It must also take into account information outside the system and ensure their processing. It's the brain of the system. At present, a microprocessor gathers on a few square millimeters more and more complex functionalities" (French (Bayonne) university course, 2015).

1) Basic architecture of a Microprocessor :

A microprocessor is built around two main elements:

- A control unit: also called a command
- A processing unit

Associated with registers responsible for storing the various information to be processed. These three elements are interconnected by internal buses allowing the exchange of information.



2) Control unit :

It allows to sequence the flow of the instructions. It performs the search in memory of the instruction. Since each instruction is coded in binary form, it decodes it to finally perform its execution and then prepares the next instruction. For that, it is composed by:

→ The program counter: (PC) also called ordinal counter. The PC is constituted by a register whose content represents the address of the next instruction to be executed. It is therefore initialized with the address of the first instruction of the program. Then it will be incremented automatically to point to the next instruction to execute.

→ The instruction register: Contains the instruction being processed.

→ The instruction decoder

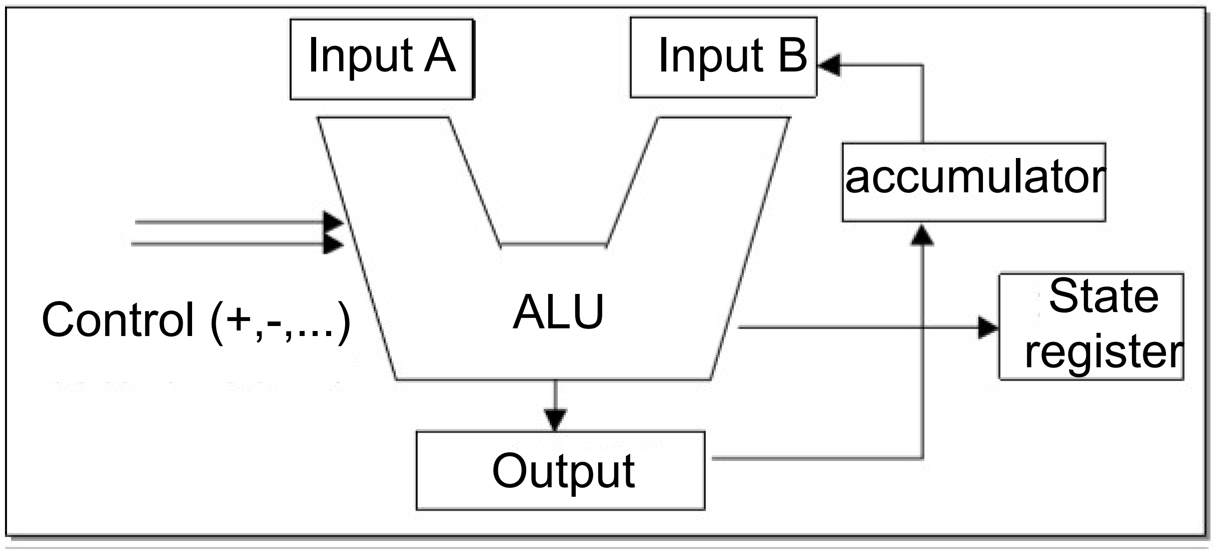
→ The sequencer: It organizes the execution of the instructions at the rhythm of a clock. It develops all the internal or external synchronization signals (control bus) of the microprocessor according to the various control signals from the instruction decoder or the status register, for example. This is a controller made either wired (obsolete) or micro-programmed, it is called micro-microprocessor.

3) Processing Unit :

This is the heart of the microprocessor. It groups the circuits that provide the necessary treatments for the execution of the instructions. The processing unit is composed of three main execution units, the first is the arithmetic logic unit (ALU) and then two more are added which are the floating-point unit and the multimedia unit for reasons for optimizing microprocessor performance.

→ Arithmetic logic unit : (ALU)

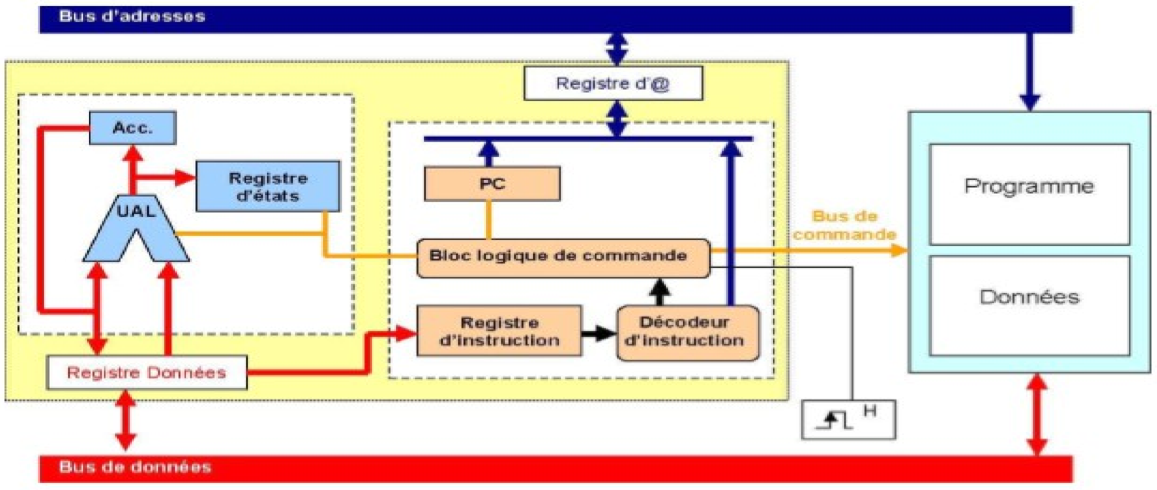
It consists of logic circuits such as adders, subtractors, logic comparators ... etc., in order to perform the calculations and the logical operations of the different instructions to be executed, the data to be processed are presented at the inputs of the UAL, are processed then the result is output and usually stored in a so-called accumulator register. The information concerning the operation is sent to the state register.



→ Floating point calculation unit :
It is a unit that is able to perform computational operations for reals as well as complex mathematical and scientific calculations. Originally the task of this unit was performed by a whole separate processor, in 1989 it was integrated into the microprocessors to optimize calculations.

→ Multimedia unit :
It is a unit that is responsible for accelerating the execution of multimedia programs including videos, sound, 3D graphics, etc. This unit is called MMX for the first pentium (MutiMedia eXtensions) integrating management functions mutimedia, the same 3DNOW technology for AMD and SSE for pentiumIII. These units were created due to the great trend toward multimedia in all types of computer programs (games, Internet software, encyclopedias ...).

4) Other microprocessor unit



Sorry it's in french ...

→ Cache memory unit:

The main function of the cache memory is to optimize access to the different instructions and data that the microprocessor needs when running the programs. The task of this unit is to put in the cache which is much faster than the main memory, the most used information and that the microprocessor needs frequently. Access to information will therefore be faster and program execution is more optimal. The principle of storing information in this memory is done by a controller that uses specific algorithms that allow to choose what are the data and instructions in the central memory to put in this memory and when to replace them since the size of the cache is very limited compared to the size of the main memory. The performance of the microprocessor is very much related to the validity of the predictions made by the cache controller. There are two levels of cache memory. The level 1 cache is still integrated into the microprocessor, its size is a few kilobytes and can reach 256 KB in some microprocessors. The second level cache is usually on the motherboard, its size varies from 256 KB up to 1 MB. (3)

→ Bus interface unit :

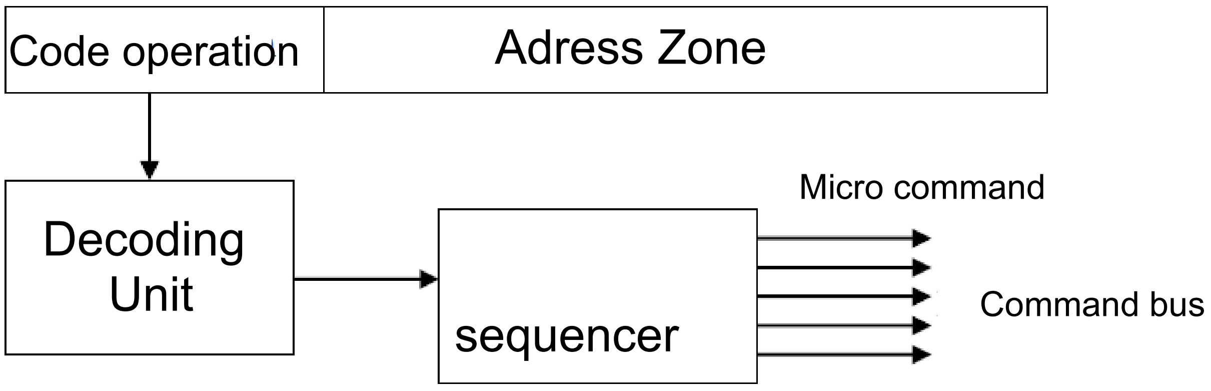
Manages the exchanges through the bus between the microprocessor and the other components, if the microprocessor needs for example a piece of data in RAM, the bus interface unit is responsible for bringing it back by controlling the memory access and putting the result on the data bus that will route it to the data register.

→ Segmentation and paging unit :

These units have become necessary especially for microprocessors of the INTEL X86 family because of their particular ways of managing memory. These units make it possible to translate the logical addresses manipulated in the programs into physical addresses that correspond to real addresses in memory. These units vary from one microprocessor to another depending on the memory management technology used.

→ Decoding unit :

It decomposes and analyzes the instruction found in the instruction register, according to the operation code, it sends the nature of the operations to be performed to the control unit and precisely the sequencer which will then generate the necessary microcommands for the various components involved at the execution of the current instruction.



→ Anticipation unit and queue :

The role of the anticipation unit is to retrieve the instructions to be executed. These instructions are first searched for in the internal cache memory of the processor. If they are not there, the anticipation unit is addressed to the bus interface unit so that they are read into the main memory. During this operation, the contents of the following addresses of the memory are read as well and placed in the cache memory of the processor. In this way, the next instructions sought will be available more quickly (provided that the content is not changed by then). The anticipation unit places the instruction in a queue and takes care of fetching the next one. Thanks to this device, the execution unit hardly ever has to wait until the instructions to be executed are brought to it (this can happen, however, if a series of very fast instructions to execute is presented). Conversely, if the instructions require a significant execution time, the queue is filled. In this case the anticipation unit stops working until space is available for new instructions

5) Microprocessor register

A register is a memory zone inside the small microprocessor, which makes it possible to memorize memory words or addresses temporarily during the execution of the instructions. The number and type of registers the CPU has is a critical part of its architecture and has a significant influence on programming. The structure of the CPU registers varies considerably from one manufacturer to another. However, the basic functions performed by the different registers are essentially the same.

→ General register :

These are fast memories inside the microprocessor that allow the thread to manipulate data at high speed. They are connected to the data bus internal to the microprocessor.
These registers make it possible to save information used in the programs or intermediate results, this avoids access to the memory, thus accelerating the execution of the programs. General registers are available to the programmer who normally has a choice of instructions to manipulate them as :

- Loading a register from the memory or another register.
- Recording in the memory of the contents of a register.
- Transfer the contents of a register into the accumulator and vice versa.
- Incrementation or decrementation of a register.

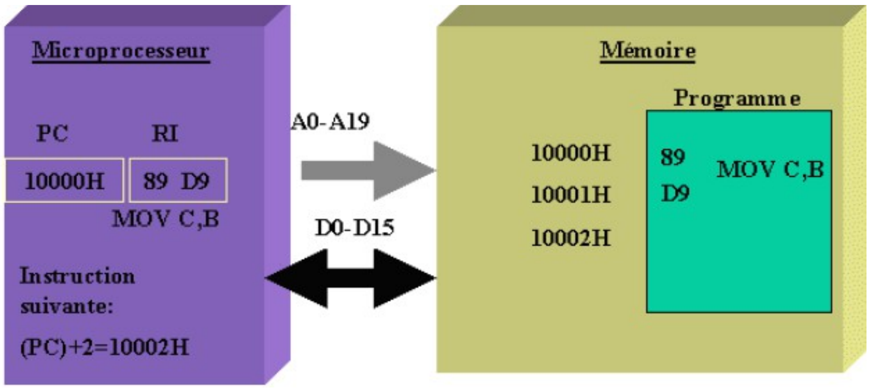
→ Adress register :

These are registers connected on the bus addresses, their content is an address in central memory. There are several types We can cite as registers:

- The ordinal counter
- The stack pointer
- Index registers

> The ordinal counter :

It contains the address of the instruction to search in memory. The control unit increments the ordinal counter (PC) by the number of bytes on which the instruction, being executed, is encoded. The ordinal counter will then contain the address of the next instruction.



> The instruction register :

It contains the instruction that must be processed by the microprocessor, this instruction is looked for in memory and then placed in this register to be decoded by the decoder and prepared for execution. An instruction is an elementary operation of a programming language, that is to say the smallest order that a computer can understand.

Operation zone	Adress zone

Instruction composition

-Operation zone : This zone makes it possible to determine the nature of the operation to be performed by the microprocessor.

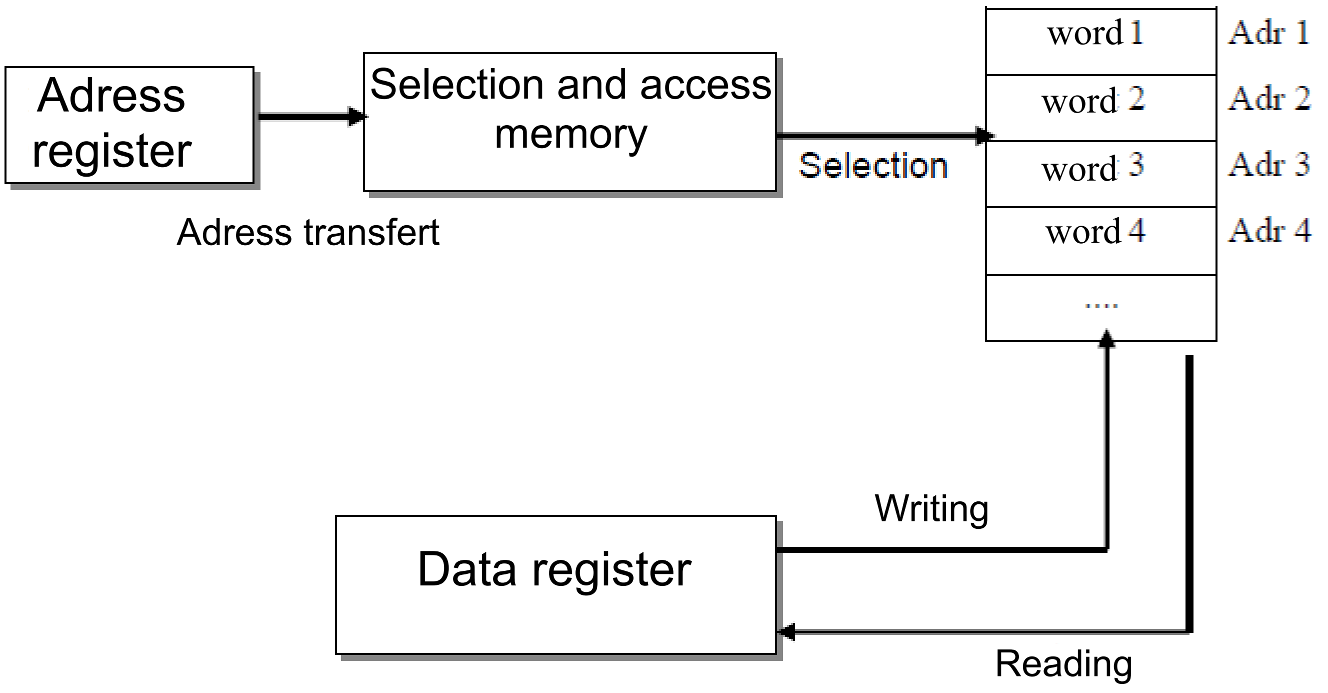
-Adress zone : It contains the addresses in memory of the operands that participate in an operation, in some cases it contains the operand itself. There are several modes of addressing to access the data.

> Adress register :

It is a register that contains the address of the word to access in central memory. At each memory access, the searched address is stored in this register. It has the size of an address which is the same as that of the address bus, which makes it possible to determine the number of addressable memory words and the addressable memory space.

> Data register :

It contains the memory word that is the object of a read or write operation in the main memory. This register has the size of a memory word which is the same as that of the working registers and the accumulator which is equal to the size of the data bus. The diagram below shows the operation of the two address and memory word registers.



-Reading : The address of the word to be read is transferred to the address register, the memory access device is responsible for searching for the word and putting it in the memory word register.

-Writing : The address register is loaded by the memory address or we will write, the word to be written is placed in the memory word register. When the order of writing is given, the contents of the memory boxes will be overwritten and replaced by the new value. On the other hand, in case of reading, the contents of the memory boxes are not destroyed.

> Accumulator register:

It is a very important work register of the ALU, in most of the arithmetic and logical operations, the accumulator contains one of the operands before the execution and the result afterwards. It can also serve as a buffer register in the input / output operations. Generally, the accumulator is the same size as a memory word. (3)

Naturally, the programmer has access to this register, which is always very busy during the data processing. Some processors have multiple accumulators and in this case, the operation codes specify the accumulator used.

> PSW : program status word

It is a register that contains the different bits called Flags indicating the status of a particular condition in the CPU. These bits can be tested by a program and thus decide the action sequences to take. To perform its job properly, the sequencer must also know the status of a number of other components and have information about the operation or operations that have already been performed (for example, must be taken into account in the addition in progress of a possible prior retention generated by a previous addition). The status register provides this information.

U	U	U	U	O	D	I	T	S	Z	U	AC	U	P	U	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Case of an 8088 (16bits registers)
U: undefined bits
Differents bits are:

→ Carry bits: It is the bit C which is the bit n ° 0. This bit is 1 if there is a hold that is generated during an arithmetic operation.
→ Parity bit : bit P n°2, this bit is set to 1 if the number of 1 in the accumulator is even.

→ Auxiliary carry bit : The bit AC , it is the bit n ° 4, it is set to 1 if a restraint is generated between groups of 4 bits.

→ Zero bit : It is bit 6, it is 1 if the result of an arithmetic operation is zero.

→ Signed bit : bit n°7, it is at 1 if the result of an arithmetic operation is negative.

→ Overflow of capicity bit : The bit n ° 11,is at 1 if there is overflow in the arithmetic operations. For example, on 8 bits in addition to 2, the binary numbers are coded on 8 bits. The variant numbers of -128 (100000000) to +127 (01111111) can be encoded. If we do 107 + 28 this gives 135 which can not be represented, or generation of overflow.

> Stack pointer :

→ It contains the address of the stack. This is a part of the memory, it allows to store information (the contents of the registers) relating to the processing of interrupts and subroutines.

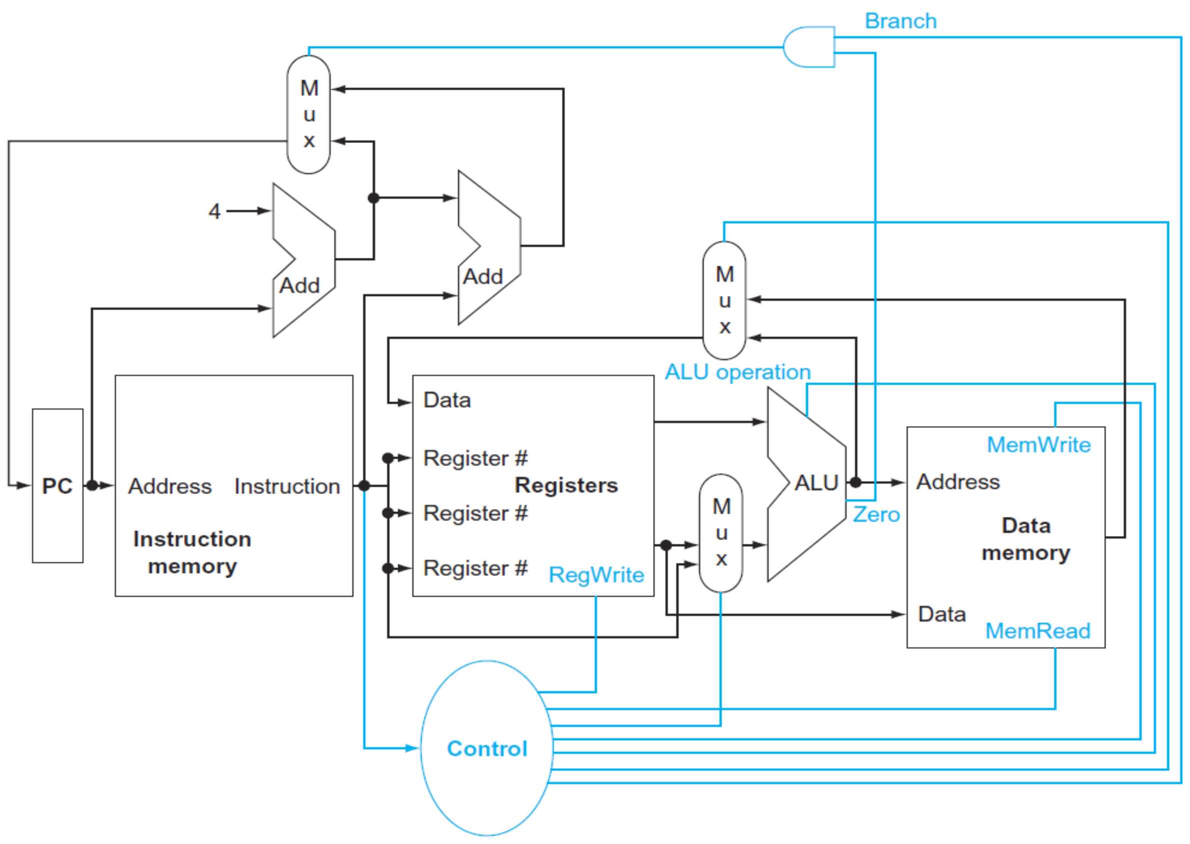
→ The stack is handled in last in first out LIFO: The operation is identical to a pile of plate.

→ The SP stack pointer points to the top of the stack, it is decremented before each stack, and incremented after each unloading.

→ There are two instructions for stacking and popping: PUSH and POP. example: PUSH A will stack the register A and POP A will depilate.

All this part is a mix of 3 university course (1),(2),(3)

Requirement to achieve my work :



This is the microprocessor that I will build at the end of this final degree project. To realize this I had to learn several thing like coding in VHDL language, learn how to build a datapath, how to realize some instruction (add, lw ...), and learn in general about parts of the microprocessor (ALU, Register file...) . All explain come from a book name "computer organization" (4) and a spanish course name "disenyo procesador" (5).

I) Instruction : language of the computer:

1) Operations of the computer hardware:

Every computer must be able to perform arithmetic. The MIPS assembly language notation
add a, b, c

instructs a computer to add the two variables b and c and to put their sum in a. This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables b, c, d, and e into variable a. The following sequence of instructions adds the four variables:

add a, b, c # The sum of b and c is placed in a
add a, a, d # The sum of b, c, and d is now in a
add a, a, e # The sum of b, c, d, and e is now in a
Thus, it takes three instructions to sum the four variables. The words to the right of the sharp symbol (#) on each line above are comments for the human reader, so the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.(4)

Category	Instruction	Exemple	Meaning	Comment
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1=0 or 1`	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

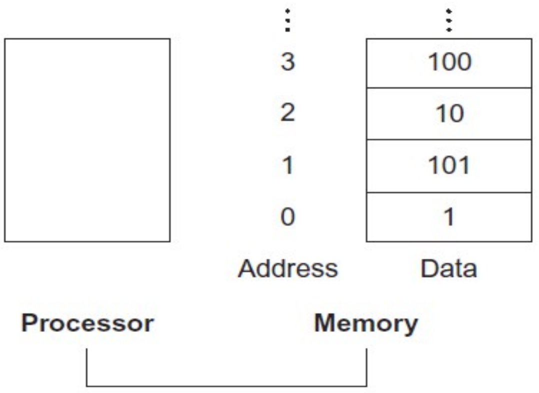
2) Operands of the computer hardware :

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS. Thus, c continuing in our top-down, stepwise evolution of the symbolic representation of the MIPS

language, in this section we have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-bit registers. The reason for the limit of 32 registers may be found in the second of our three underlying design principles of hardware technology:
Design Principle 2 >> Smaller is faster. A very large number of registers may increase the clock cycle time. Guidelines such as "smaller is faster" are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format. Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called registers. Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name word in the MIPS architecture.

→ Memory operand :

Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These complex data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access such large structures ? The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory. As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers.



→ Constant or immediate operand

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the MIPS arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks. Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register $s3, we could use the code

```
lw $t0, AddrConstant4($s1) # $t0 = constant 4
```

```
add $s3,$s3,$t0 # $s3 = $s3 + $t0 ($t0 == 4)
```

assuming that $s1 + AddrConstant4 is the memory address of the constant 4.

An alternative that avoids the load instruction is to off er versions of the arithmetic instructions in which one operand is a constant. Th is quick add instruction with one constant operand is called add immediate or addi. To add 4 to register $s3, we just write

```
addi  $s3,$s3,4     # $s3 = $s3 + 4
```

Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

- Representing instruction in the computer:

We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions. Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.

MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:
>op: Basic operation of the instruction, traditionally called the opcode.
>rs: The first register source operand.
>rt: The second register source operand.
>rd: The register destination operand. It gets the result of the operation.
>shamt: Shift amount.
>funct: Function. This field, oft en called the function code, selects the specific variant of the operation in the op field.
A problem occurs when an instruction needs longer fields than those shown above. For example, the load word instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the constant within the load word instruction would be limited to only 25 or 32. This constant is used to select elements from arrays or data structures, and it oft en needs to be much larger than 32. Th is 5-bit field is too small to be useful. Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This leads us to the final hardware design principle:
Design Principle 3 >> Good design demands good compromises.
The compromise chosen by the MIPS designers is to keep all instructions the same length,

thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called R-type (for register) or R-format. A second type of instruction format is called I-type (for immediate) or I-format and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

The 16-bit address means a load word instruction can load any word within a region of 215 or 32,768 bytes (213 or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than 215. We see that more than 32 registers would be diffi cult in this format, as the rs and rt fields would each need another bit, making it harder to fi t everything in one word.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the fi rst fi eld (op) so that the hardware knows whether to treat the last half of the instruction as three fi elds (R-type) or as a single field (I-type).The table shows the numbers used in each field for the MIPS instructions covered so far.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

- Instruction for making decision:

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the if statement, sometimes combined with go to statements and labels. MIPS assembly language includes two decision-making instructions, similar to an if statement with a go to. The first instruction is

beq register1, register2, L1

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for branch if equal.

The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled L1 if the value in register1 does not equal the value in register2. The mnemonic bne stands for branch if not equal. These two instructions are traditionally called conditional branches. The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the if statement. This example introduces another kind of branch, oft en called an unconditional branch. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is jump, abbreviated as j

```
j Exit       # go to Exit
```

The assignment statement in the else portion of the if statement can again be compiled into a single instruction. We just need to append the label Else to this instruction. We also show the label Exit that is after this instruction, showing the end of the if-then-else compiled code:

```
Else:sub $s0,$s1,$s2 # f = g – h (skipped if i = j)
Exit:
```

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores

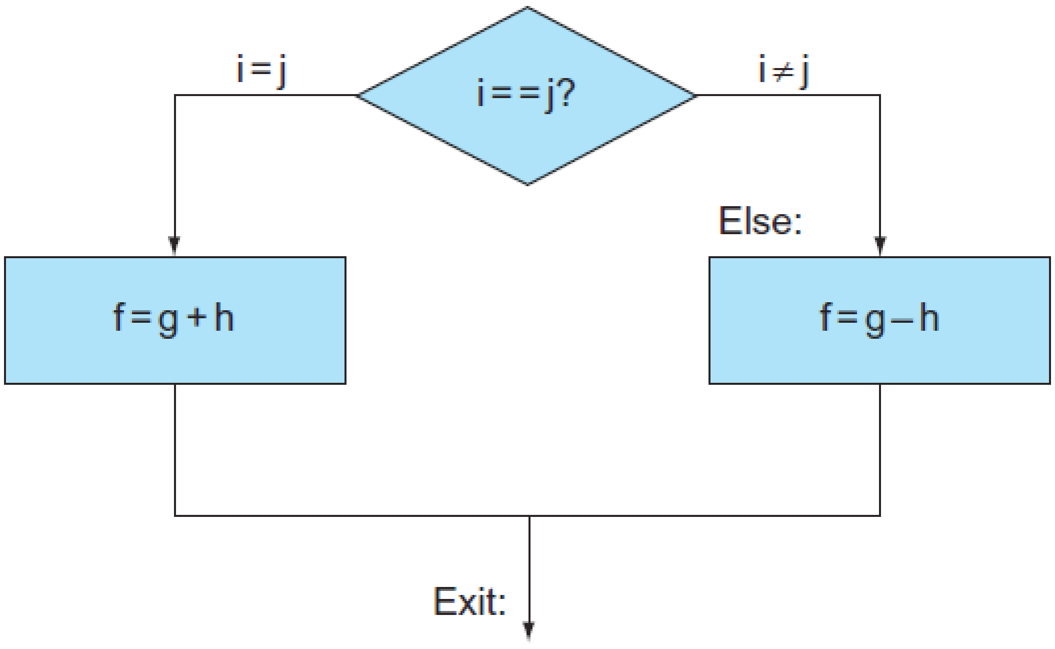


Illustration of the options in the if statement(4)

→ Loops :

Decisions are important both for choosing between two alternatives found in "If" statements, and for iterating a computation, found in loops. The same assembly instructions are the building blocks for both cases.

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a for loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the fi rst is less than the second; otherwise, it is set to 0. Th e MIPS instruction is called s*et on less than,* or slt. For example,

```
slt $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4
```

means that register $t0 is set to 1 if the value in register $s3 is less than the value in register $s4; otherwise, register $t0 is set to 0. Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register $s2 is less than the constant 10, we can just write

```
slti $t0,$s2,10 # $t0 = 1 if $s2 < 10
```

→ Case/ Switch statement :

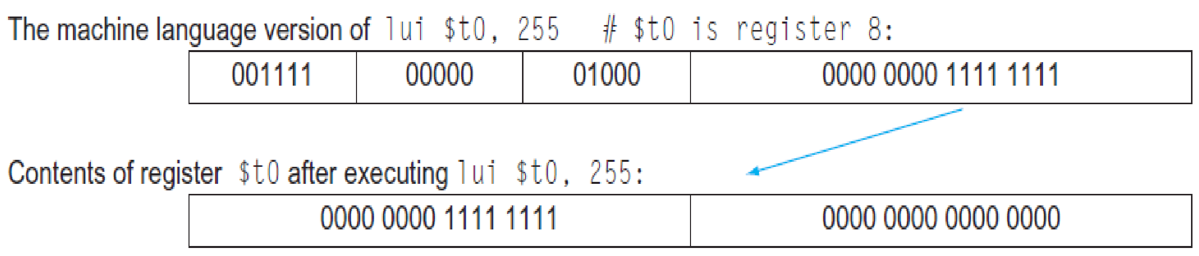
Most programming languages have a case or switch statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement switch is via a sequence of conditional tests, turning the switch statement into a chain of if-then-else statements. Sometimes the alternatives may be more effi ciently encoded as a table of addresses of alternative instruction sequences, called a jump address table or jump table, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a jump register instruction (jr), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction.

- MIPS Addressing for 32-bit Immediates and address

Although keeping all MIPS instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. (4)

→ 32-Bit Immediate Operands :

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction load upper immediate (lui) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant.



The machine language version of `lui $t0, 255   # $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

→ Addressing in Branches and Jumps :

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the J-type, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

j   10000   # go to location 10000

could be assembled into this format (it's actually a bit more complicated, as we will see):

2	10000
6 bits	26 bits

where the value of the jump opcode is 2 and the jump address is 10000. Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

bne $s0,$s1,Exit   # go to Exit if $s0 ≠ $s1

is assembled into this instruction, leaving only 16 bits for the branch address:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

If addresses of the program had to fit in this 16-bit fi eld, it would mean that no program could be bigger than 216, which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following: Program counter = Register Branch address This sum allows the program to be as large as 232 and still be able to useconditional branches, solving the branch address size problem. Then the question is, which register? The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in if statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the program counter (PC) contains the address of the current instruction, we can branch within 2^15 words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and if statements are much smaller than 2^16 words, so the PC is the ideal choice. This form of branch addressing is called PC-relative addressing.

→ MIPS Addressing Mode Summary

Multiple forms of addressing are generically called addressing modes. The MIPS addressing modes are the following :

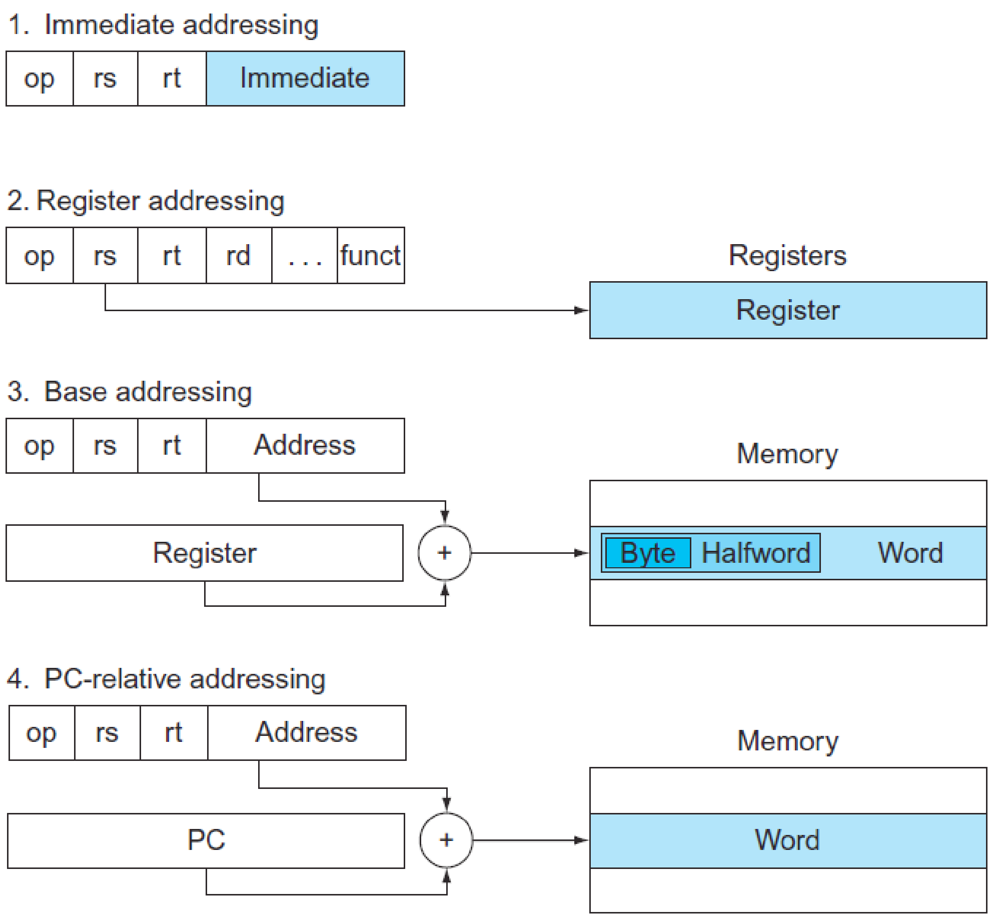
> Immediate addressing, where the operand is a constant within the instruction itself

> Register addressing, where the operand is a register

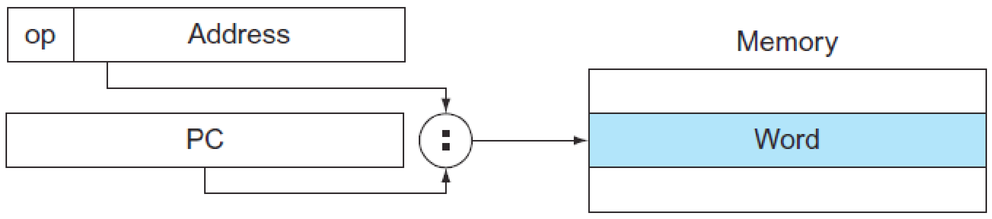
> Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction

> PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction

> Pseudodirect addressing, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



5. Pseudodirect addressing



The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shift ed left 2 bits to the PC and mode 5 concatenating a 26-bit address shift ed left 2 bits with the 4 upper bits of the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (addi) and register (add) addressing.

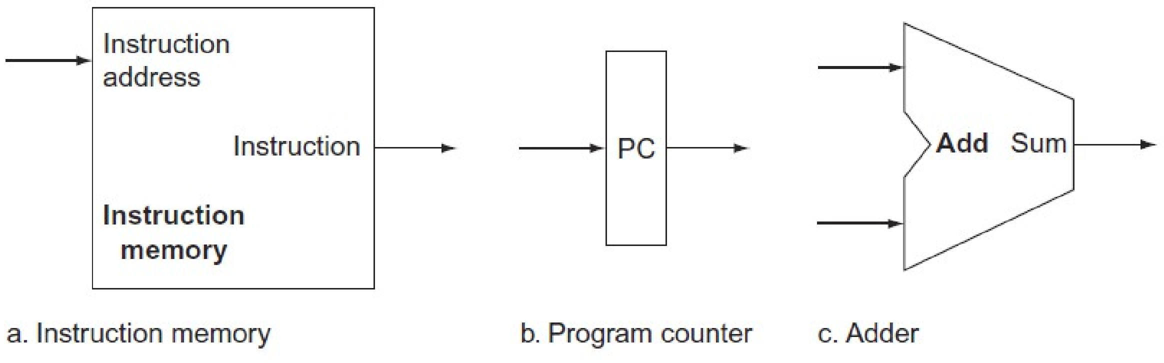
→ Decoding Machine Language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at “core dump.” . This figure helps when translating by hand between assembly language and machine language.

3) <u>How to build a datapath:</u>

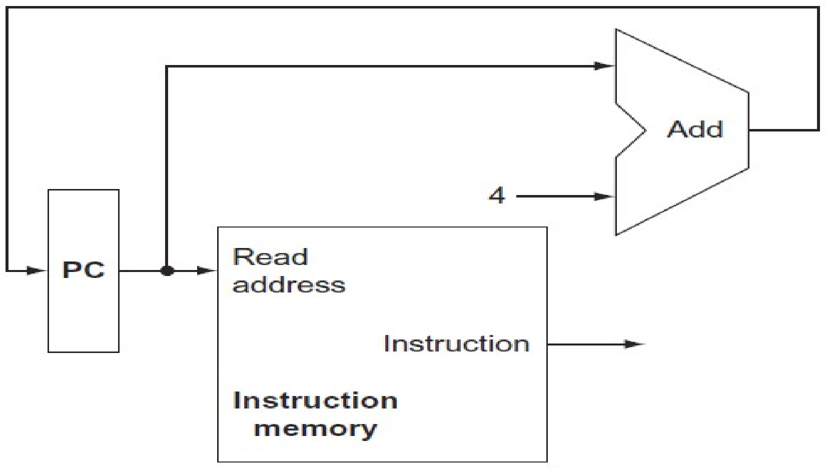
- Way of building a datapath :

Examine the elements required for the execution of the different instructions. Build the different parts of the data path. Determine the necessary control signals. Finally, join the different parts to generate the route of data: The control signals will select the different paths of the data path.



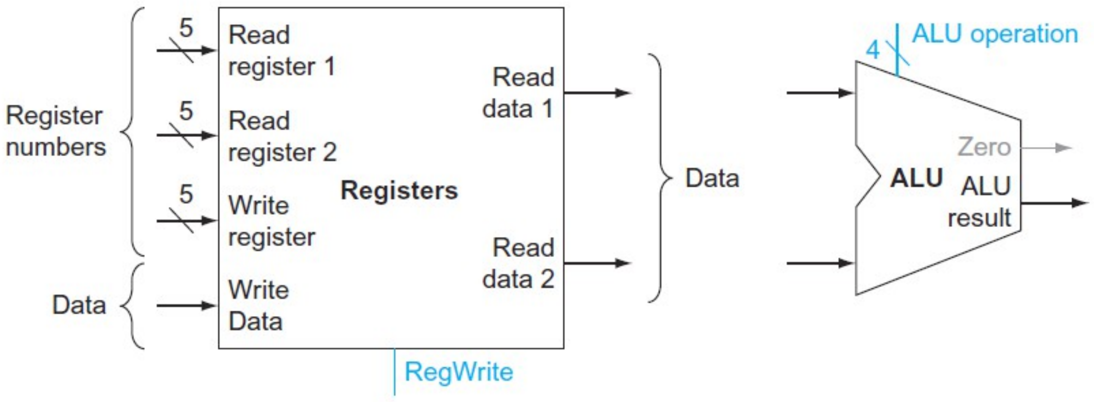
<u>Part needed:</u>

The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.



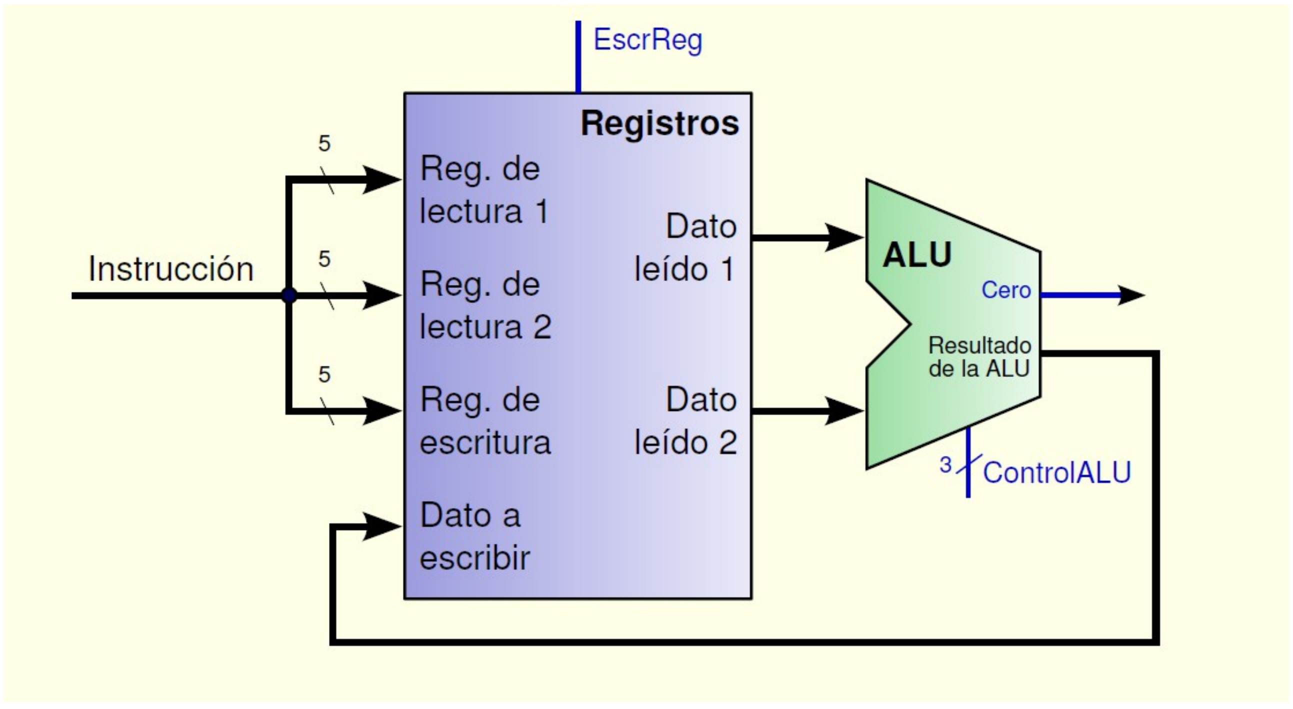
A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

Let's consider the R-format instructions They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either R-type instructions or arithmetic-logical instructions (since they perform arithmetic or logical operations). This instruction class includes add, sub, AND, OR, and slt.

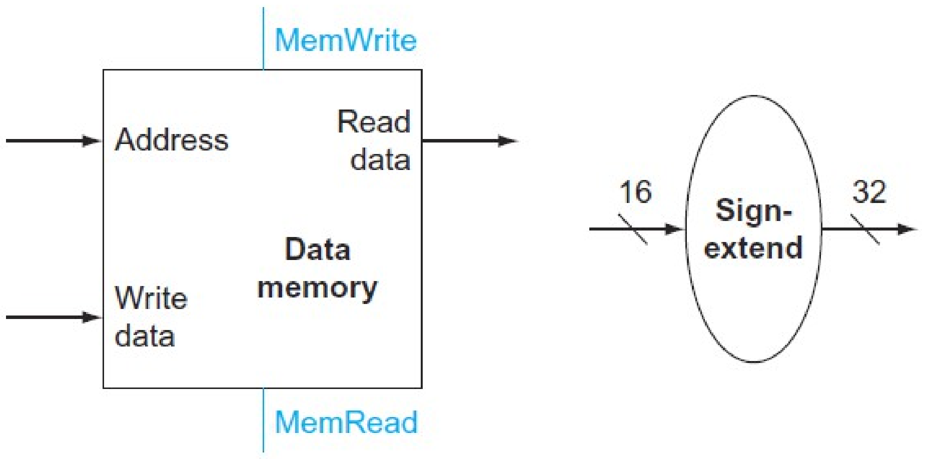


a. Registers b. ALU

The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; noother control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide. We will use the Zero detection output of the ALU shortly to implement branches. Sorry this diagram is in sapnish...



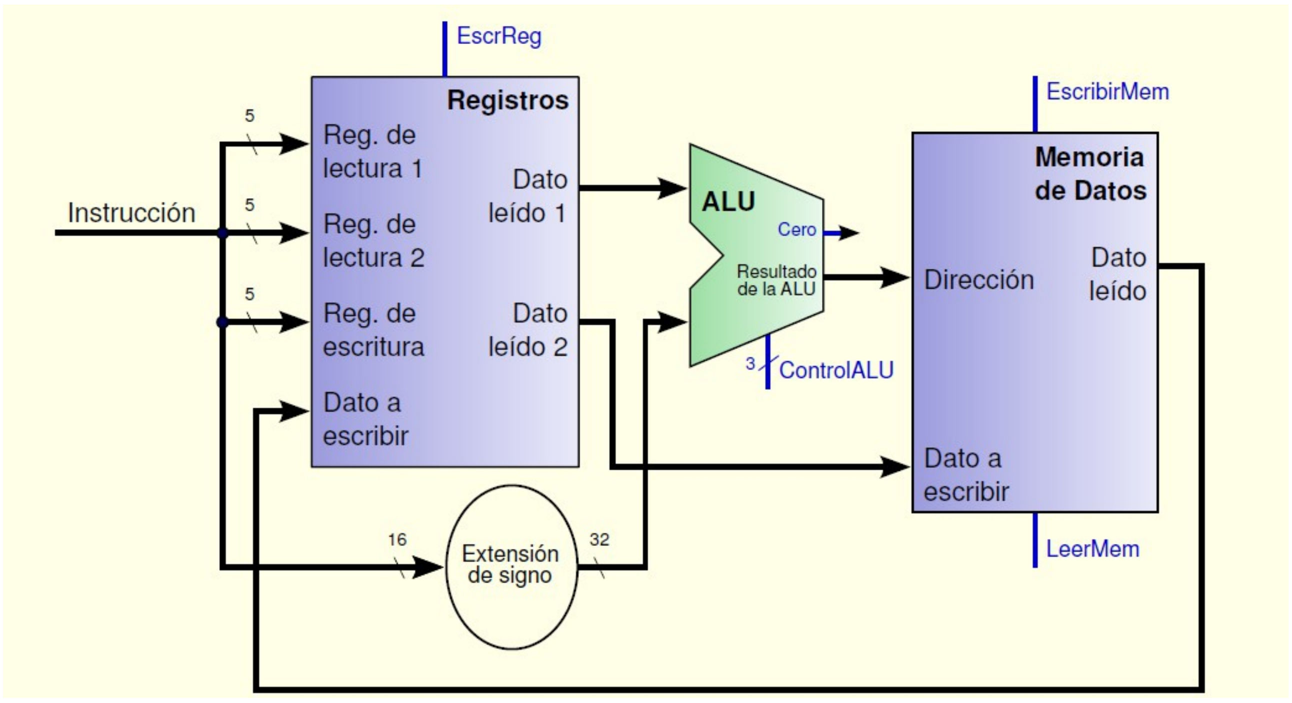
Part of the data path for aritmectic and logical operations(5)



a. Data memory unit b. Sign extension unit

Next, consider the MIPS load word and store word instructions, which have theneral form lw $t1,offset_value($t2) or sw $t1,offset_value ($t2). These instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1. If the instruction is a load, the value read from memory must be written into the register fi le in the specified register, which is $t1. Thus, we will need both the register file and the ALU.The two units needed to implement loads and stores, in addition to the register file and ALU, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output . We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge triggered design could easily be adapted to work with real memory chips.

If we connect all of this part, to realise this operations (lw,sw) we obtain this : (in spanish...)



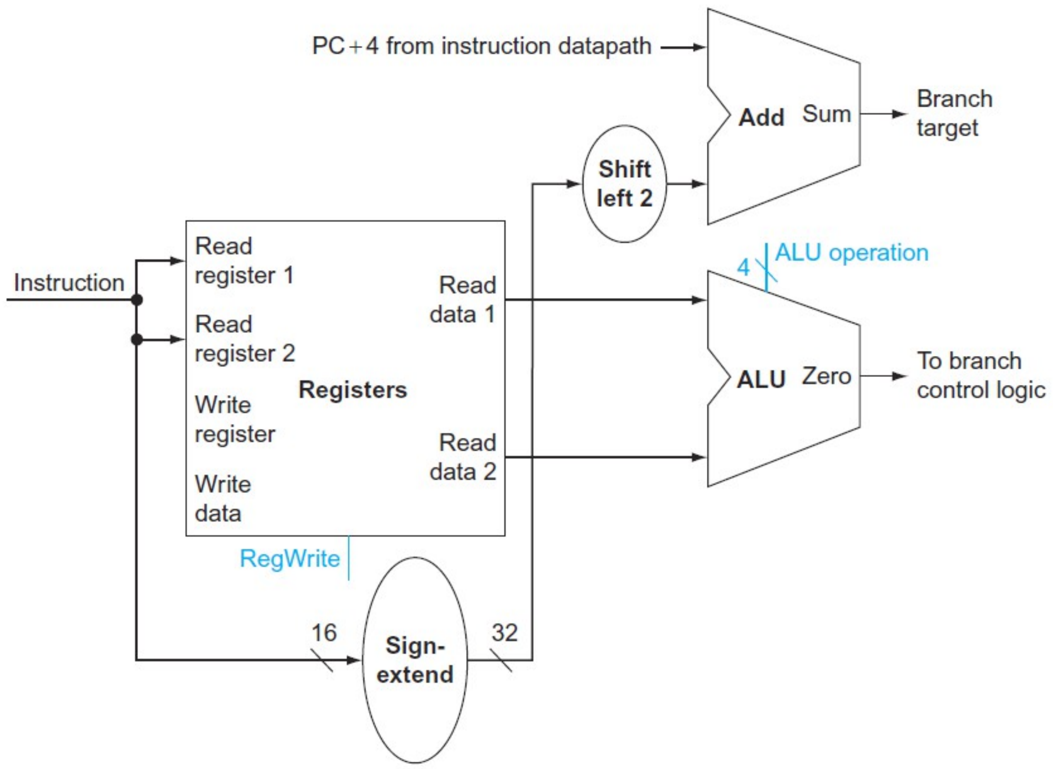
Part of the data path for the instructions of storage charge (5)

The beq instruction has three operands, two registers that are compared for equality, and a 16-bit off set used to compute the branch target address relative to the branch instruction address. Its form is beq $t1,$t2,offset. To implement this instruction, we must compute the branch target address by adding the sign extended off set fi eld of the instruction to the PC. There are two details in the definition of branch instructions to which we must pay attention:

The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.

The architecture also states that the off set fi eld is shift ed left 2 bits so that it is a word off set; this shift increases the effective range of the off set field by a factor of 4.(4)

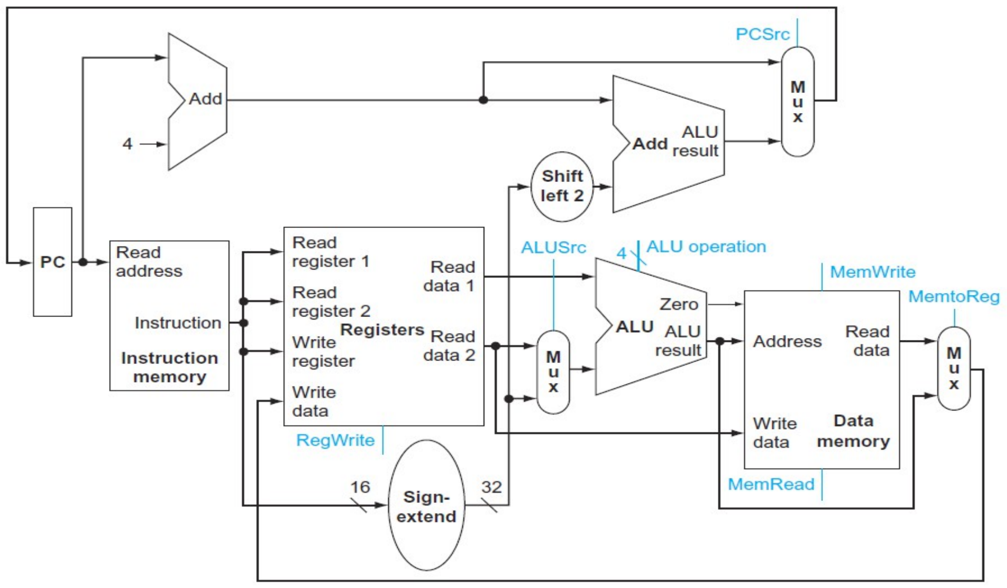
The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and thesign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. Th e unit labeled Shift left 2 is simply a routing of the signals between input and output that adds 00 two to the low-order end of the sign-extended off set fi eld; no actual shift hardware is needed, since the amount of the “shift ” is constant. Since we know that the off set was sign-extended from 16 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.



Part of the data path for the instructions Beq(4)

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation.(4)

A simple datapath for the core MIPS architecture combines the elements required by different instruction classes.This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.



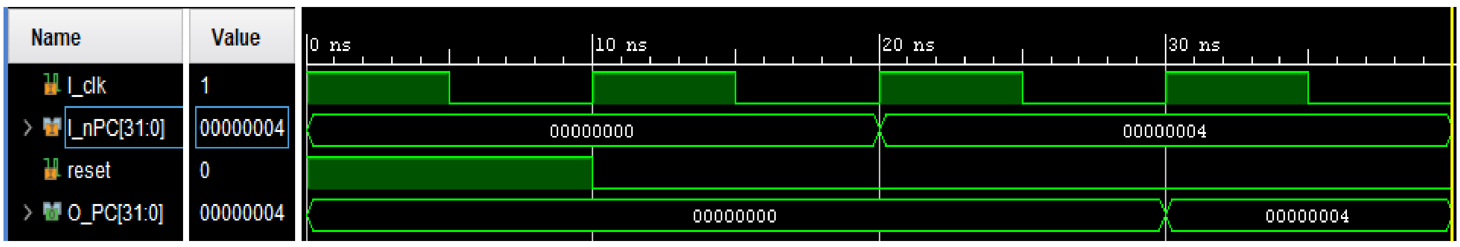
II) Step of my Vhdl learning :

I began to make the code of all the part of the Microprocessor then I connected it and to end I make the code of all the control device.
All in blue is the explainaition of the code an, in the simulation.

. I began by the PC unit (program counter) :

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
entity pc_unit is
    Port ( I_clk : in  STD_LOGIC;
           I_nPC : in  STD_LOGIC_VECTOR (31 downto 0);
          reset : in std_logic;
           O_PC : out  STD_LOGIC_VECTOR (31 downto 0)
           );
end pc_unit;
architecture Behavioral of pc_unit is
  signal current_pc: std_logic_vector( 31 downto 0) := X"00000000";
begin

  process (I_clk,reset)
  begin
  if reset = '1' then
 current_pc <= x"00000000";

  else

    if I_clk'event and I_clk='1' then
      current_pc <= I_nPC ;
    end if;
    end if;
  end process;

  O_PC <= current_pc;

end Behavioral;
```

I sumulate it :

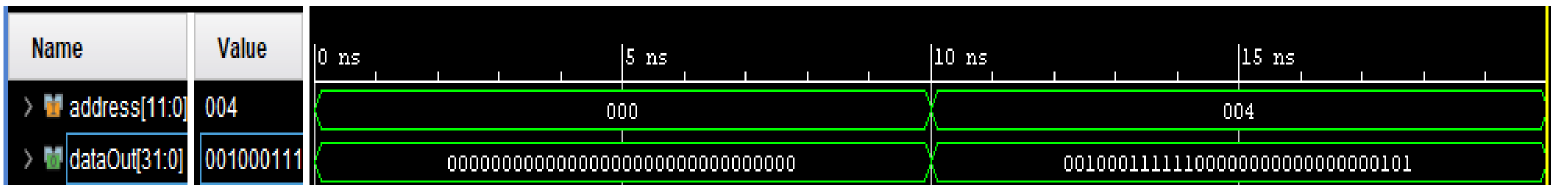


So here we can see that we are waiting for a clock rising edge to put the value in the output. So it work.

. Then I code the instruction memory :

```
 library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.all;
-- VHDL code for the Instruction Memory of the MIPS Processor
entity Instruction_Memory_VHDL is
port (
 address: in std_logic_vector (11 downto 0);
 dataOut: out  std_logic_vector(31 downto 0)
);
end Instruction_Memory_VHDL;

architecture Behavioral of Instruction_Memory_VHDL is
signal rom_addr: std_logic_vector(9 downto 0);

    type rom_type is array (0 to 2**10-1) of std_logic_vector(31 downto 0);
           signal rom: rom_type:= (others => (others => '0'));
 constant rom_data: rom_type:=(
   "00000000000000000000000000000000"
   "00100011111100000000000000000101",-- put 5 in the 16th register
   "00100011111100010000000000000110",-- put 6 in the 17th  register
   "00000010000100011001000000100000",-- add 16th register and 17th register
   "00000010000100011001000000100010",-- sub 16th register and 17th register
   "00000010000100011001000000100100",-- and betwteen 16th register and 17th  register
   "00000010000100011001000000100101",-- or between 16th register and 17th  register
   "00000010000100011001000000101010",-- slt rt<rs
   "00000010001100001001000000101010",-- slt rs<rt
   "10101100001100000000000000000000",-- lw
   "10001100001100100000000000000000",-- sw
   "00010010000100010000000000000000",-- beq 5=6 (16th register=17th register )
   "00010000000000010000000000000000",-- beq 0=0 (1st register = 2nd register )
    x"08000000",-- jump to 00000000000000000000000000000000 instruction
   "00000000000000000000000000000000",
   "00000000000000000000000000000000",
   "00000000000000000000000000000000",
    others => x"00000000"
  );

begin
 rom_addr <= address(11 downto 2);
  dataOut <= rom_data(to_integer(unsigned(rom_addr)));

end Behavioral;
```

I simualte it :

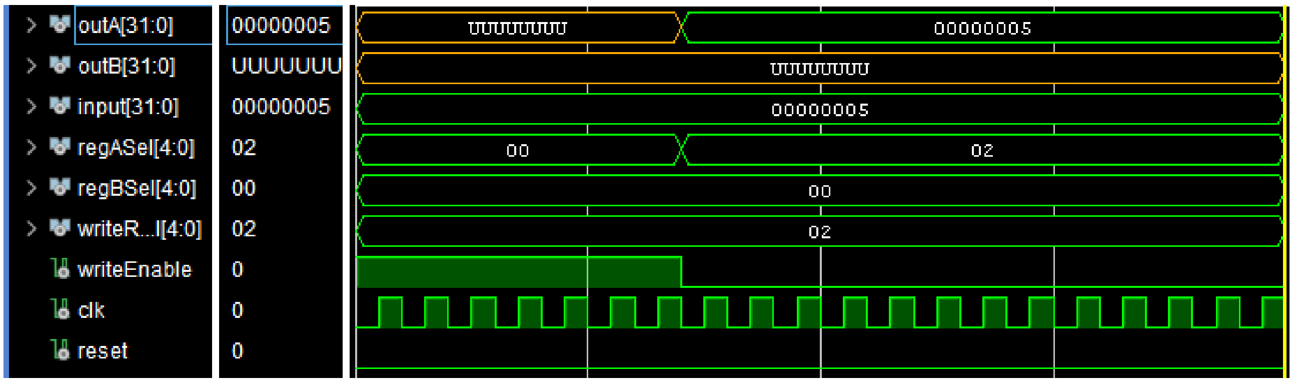


So we it respect the code because we begin by the first instruction then teh second...

Then I code the register file :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity register_file is
  port(
    outA        : out std_logic_vector(31 downto 0);
    outB        : out std_logic_vector(31 downto 0);
    input       : in  std_logic_vector(31 downto 0);
    writeEnable : in  std_logic;
    regASel     : in  std_logic_vector(4 downto 0);
    regBSel     : in  std_logic_vector(4 downto 0);
    writeRegSel : in  std_logic_vector(4 downto 0);
    reset       : in  std_logic;
    clk         : in  std_logic
    );
end register_file;


architecture behavioral of register_file is
  type registerFile is array(0 to 31) of std_logic_vector(31 downto 0);
  signal registers : registerFile;
begin
  regFile : process (clk, reset) is
  begin
    if reset = '1' then registers <= (OTHERS => X"00000000");
    elsif clk'event and clk='1' then
      if writeEnable = '1' then
            registers(to_integer(unsigned(writeRegSel))) <= input;  -- Write
      end if;
    end if;
  end process;

  regFile2 : process (regASel, regBSel) is
   begin
               if regASel = "11111" then outA <= x"00000000";
               else
           -- Read A and B before bypass
              outA <= registers(to_integer(unsigned(regASel)));
               end if;
               if regBSel = "11111" then  outB <= x"00000000";
               else
              outB <= registers(to_integer(unsigned(regBSel)));
           end if;
            -- Write and bypass
    end process;
end behavioral
```

I sumalate it :



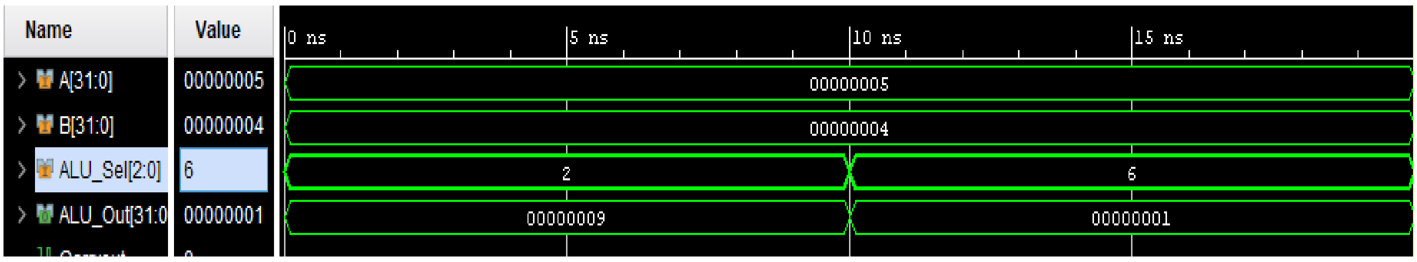
```
outA[31:0]      00000005   UUUUUUUU  00000005
outB[31:0]      UUUUUUUU   UUUUUUUU
input[31:0]     00000005   00000005
regASel[4:0]    02         00  02
regBSel[4:0]    00         00
writeR...l[4:0] 02         02
writeEnable     0
clk             0
reset           0
```

. Then I code the ALU (aritmectic logic unit) :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.NUMERIC_STD.all;
-----------------------------------------------
---------- ALU 32-bit VHDL ---------------------
-----------------------------------------------
entity ALU is

   Port (
   A, B     : in  STD_LOGIC_VECTOR(31 downto 0);  -- 2 inputs 32-bit
   ALU_Sel  : in  STD_LOGIC_VECTOR(2 downto 0);  -- 1 input 5-bit for selecting function
   ALU_Out   : out  STD_LOGIC_VECTOR(31 downto 0); -- 1 output 32-bit
   Carryout : out std_logic        -- Carryout flag

   );
end ALU;
architecture Behavioral of ALU is

signal ALU_Result : std_logic_vector (31 downto 0);
```

```
begin

   process(A,B,ALU_Sel)
 begin
  case(ALU_Sel) is
  when "010" => -- Addition
   ALU_Result <= std_logic_vector (unsigned (A) + unsigned (B)) ;
  when "110" => -- Subtraction
   ALU_Result <= std_logic_vector  (unsigned (A) - unsigned (B)) ;
  when "000" => -- Logical and
   ALU_Result <= A and B;
  when "001" => -- Logical or
   ALU_Result <= A or B;
  when "111" => -- Greater comparison
   if(A<B) then
    ALU_Result <= x"00000001" ;
   else
    ALU_Result <= x"00000000" ;
   end if;
  when others => ALU_Result <= std_logic_vector  (unsigned (A) + unsigned (B));
  end case;
  if ALU_Result = x"00000000" then
   Carryout <= '1';
   else
   Carryout <= '0'; -- Carryout flag
   end if;
 end process;
 ALU_Out <= ALU_Result; -- ALU out

end Behavioral;
```

Then I simulate it :



So here I sumalted the addition and the substraction and it work .

Then I code the data memory :

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
```

```
--use ieee.std_logic_arith.all;
entity ram6116 is
port(
        address: in unsigned(9 downto 0);
        dataIn: in std_logic_vector(31 downto 0);
        dataOut: out std_logic_vector (31 downto 0);
        WE_b, CS_b, OE_b: in std_logic);
end entity ram6116;
architecture
        simple_ram of ram6116 is type ram_type is array (0 to (2**10)-1) of
std_logic_vector(31 downto 0);
        signal ram1: ram_type:= (others => (others => '0'));
begin
process
        begin
                dataOut <= (others => 'Z'); -- chip is not selected
                if (CS_b = '0') then
                        if WE_b'event and WE_b='1' then -- write
                                ram1(to_integer(address)) <= dataIn;
                                wait for 0 ns;
                        end if;
                        if WE_b = '0' and OE_b = '1' then -- read
                                dataOut <= ram1(to_integer(address));
                        else
                                dataOut <= (others => 'Z');
                        end if;
                end if;
        wait on WE_b, CS_b, OE_b, address;
end process;
end simple_ram;
```

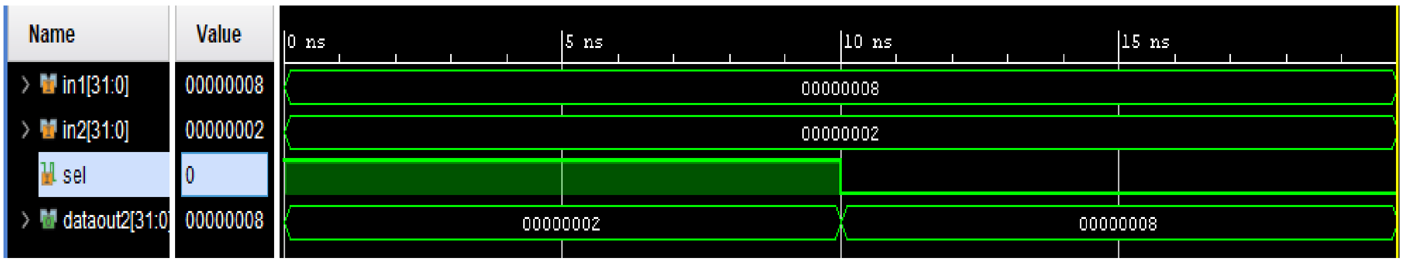
I didn't simulate it because it is more intresting to see this component connected to others.

. Then I code Multiplexor one 5 bits and othet 32 bits. One 5 bit who is connected to the writing register who is also a 5bits input. The only change is to change the number of bits of each input and output.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity depun_mux_out is
Port (
in1 : in std_logic_vector (31 downto 0); -- mux input1
in2 : in std_logic_vector (31 downto 0); -- mux input2
sel : in std_logic; -- selection line
dataout2 : out std_logic_vector (31 downto 0) -- output data
);
end depun_mux_out;
```

```
architecture Behavioral of depun_mux_out is

begin
-- This process for mux logic
process (sel, in1 , in2 )
begin
case sel is
when '0' => dataout2 <= in1;
when '1' => dataout2 <= in2;
when others => dataout2 <= x"00000000";
end case;
end process;

end Behavioral;
```

Then I simulate it :



When the selection is 1 the output take the value of the second input and when it 0 it take the value of the first input. It work !

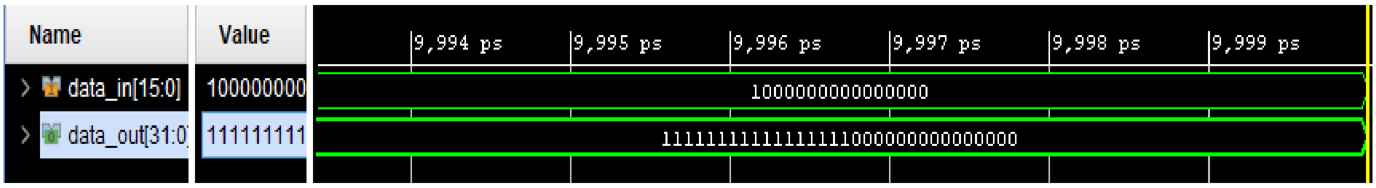
.Then I had to code several small thing like sign-extended unit, adder, and some think to make the jump instruction :

. Sign-extended :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SEXT is
  port ( data_in : in std_logic_vector(15 downto 0);
         data_out : out std_logic_vector(31 downto 0));
end SEXT;


architecture behavioral of SEXT is
begin
  process (data_in)
begin


  data_out(15 downto 0) <= data_in;
  data_out(31 downto 16) <= (31 downto 16 => data_in(15));
```

```
end process;
end behavioral;
```

Simulation :



```
data_in[15:0]    100000000   1000000000000000
data_out[31:0]   111111111   11111111111111111000000000000000
```

. Adder :

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
entity adder is
    Port ( InA : in std_logic_vector (31 downto 0);
                    InB : in std_logic_vector (31 downto 0);
                    out2 : out std_logic_vector (31 downto 0)
           );
end adder;

architecture Behavioral of adder is
    signal adder_output : std_logic_vector (31 downto 0);
begin

  process(InA, InB)
  begin
      adder_output <= std_logic_vector (unsigned(InA) + unsigned(InB));
  end process;
 out2 <= adder_output;

end Behavioral;
```

. Jump instruction :

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
entity jump_operation is
    Port (
           PC_four : in  std_logic_vector (3 downto 0);
                    address : in std_logic_vector ( 25 downto 0);
                    new_address : out std_logic_vector ( 31 downto 0)
           );
end jump_operation;
architecture Behavioral of jump_operation is
```

```
begin

  process (PC_four, address)
  begin
   new_address (31 downto 28) <= Pc_four;
   new_address (27 downto 2) <= address;
   new_address (1 downto 0) <= "00";
  end process;

end Behavioral;
```

To make all work we need to control all of it so here is the control unit and the alu control unit.

. Control unit :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-- VHDL code for Control Unit of the MIPS Processor
entity control_unit_VHDL is
port (
  Instruction: in std_logic_vector(5 downto 0);
 FuenteAlu,Escregister,leerMem,EscrMem,SaltoCond, saltoIncond : out std_logic ;
 MemaReg,RegDesr : out std_logic ;
 Aluop : out std_logic_vector (1 downto 0)
 );
end control_unit_VHDL;

architecture Behavioral of control_unit_VHDL is

begin
process(Instruction)
begin

 case Instruction is
  when "000000" => -- R instruction (add, sub, ...)
    RegDesr <= '1';
          FuenteAlu <= '0';
          MemaReg <= '0';
          Escregister <= '1';
          leerMem <= '0';
          EscrMem <= '0';
          SaltoCond <= '0';
          saltoIncond <= '0';
          Aluop <= "10";

 when "100011" =>-- lw
   RegDesr <= '0';
          FuenteAlu <= '1';
```

```
        MemaReg <= '1';
        Escregister <= '1';
        leerMem <= '1';
        EscrMem <= '0';
        SaltoCond <= '0';
        saltoIncond <= '0';
        Aluop <= "00";

when "101011" => -- sw
 RegDesr <= '0';
        FuenteAlu <= '1';
        MemaReg <= '1';
        Escregister <= '0';
        leerMem <= '0';
        EscrMem <= '1';
        SaltoCond <= '0';
        saltoIncond <= '0';
        Aluop <= "00";

when "000100" => -- beq
  RegDesr <= '0';
        FuenteAlu <= '0';
        MemaReg <= '0';
        Escregister <= '0';
        leerMem <= '0';
        EscrMem <= '0';
        SaltoCond <= '1';
        saltoIncond <= '0';
        Aluop <= "01";

when "001000" =>-- addi
  RegDesr <= '0';
        FuenteAlu <= '1';
        MemaReg <= '0';
        Escregister <= '1';
        leerMem <= '0';
        EscrMem <= '0';
        SaltoCond <= '0';
        saltoIncond <= '0';
        Aluop <= "00";


when "000010" => -- jump
RegDesr <= '0';
        FuenteAlu <= '0';
        MemaReg <= '0';
        Escregister <= '0';
        leerMem <= '0';
        EscrMem <= '0';
        SaltoCond <= '1';
```

```
          saltoIncond <= '1';
          Aluop <= "00";


          when others =>
          RegDesr <= '0';
          FuenteAlu <= '0';
          MemaReg <= '0';
          Escregister <= '0' ;
          leerMem <= '0';
          EscrMem <= '0';
          SaltoCond <= '0';
          saltoIncond <= '0';
          Aluop <= "00";

 end case;
end process;

end Behavioral;
```

ALU Control unit :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- VHDL code for ALU Control Unit of the MIPS Processor
entity ALU_Control_VHDL is
port(
  ALU_Sel: out std_logic_vector(2 downto 0);
  ALUOp : in std_logic_vector(1 downto 0);
  ALU_Funct : in std_logic_vector(5 downto 0)
);
end ALU_Control_VHDL;

architecture Behavioral of ALU_Control_VHDL is
begin
process(ALUOp,ALU_Funct)
begin
case ALUOp is
when "10" => --R Type
 case ALU_Funct is
 when "100000" =>
 ALU_Sel <= "010"; --add
when "100010" =>
 ALU_Sel <= "110"; --sub
when "100100" =>
ALU_Sel <= "000"; --and
when "100101" =>
ALU_Sel <= "001"; -- or
```

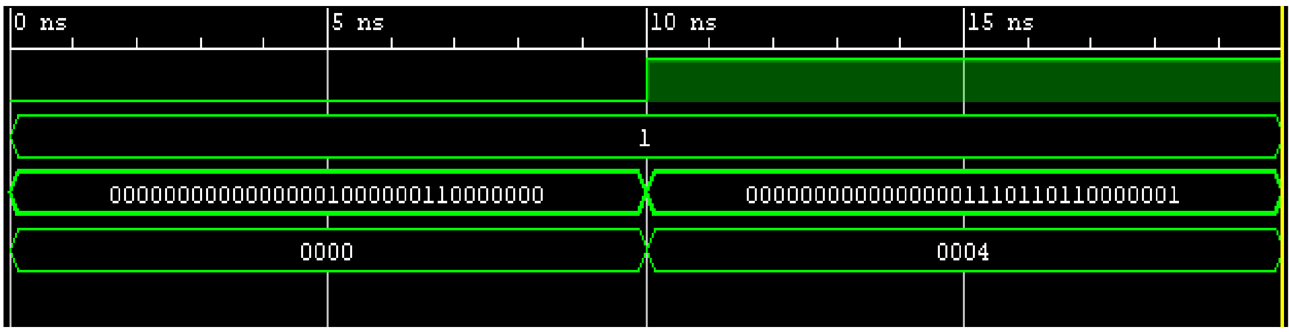
```
when "101010" =>
ALU_Sel <= "111";--put if better
when others =>
ALU_Sel <= "010";
end case ;
when others =>
ALU_Sel <= "010"; --add
end case;
end process;
end Behavioral;
```

. Then I began to learn about connection so I connected the program counter with the Instruction memory.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.all;

ENTITY connection IS
        PORT (I_clk : in  STD_LOGIC;
                I_nPCop : in  STD_LOGIC_VECTOR (1 downto 0);
                  dataOut: out  std_logic_vector(31 downto 0)
);

END connection;

ARCHITECTURE structure OF connection IS
        COMPONENT pc_unit
                PORT (
                I_clk : in  STD_LOGIC;
        I_nPC : in  STD_LOGIC_VECTOR (15 downto 0);
        I_nPCop : in  STD_LOGIC_VECTOR (1 downto 0);
        O_PC : out  STD_LOGIC_VECTOR (15 downto 0)
                  );
        END COMPONENT;
        COMPONENT Instruction_Memory_VHDL
                PORT (address: in std_logic_vector(15 downto 0);
                dataOut: out  std_logic_vector(31 downto 0)
);
        END COMPONENT;
        SIGNAL s1: STD_LOGIC_VECTOR (15 downto 0);
        FOR instruction1 : pc_unit USE ENTITY WORK.pc_unit;
        FOR instruction2 : Instruction_Memory_VHDL USE ENTITY
WORK.Instruction_Memory_VHDL;
        BEGIN
        instruction1: pc_unit PORT MAP (I_clk, s1, I_nPCop, s1);

        instruction2: Instruction_Memory_VHDL PORT MAP (s1, dataOut ) ;

END structure;
```

Clk
I_nPCop
dataOut

s1



As we can see there is a problem because we choose the first instruction then the fourth … So we had to fix this.

My Results

So here is the final MIPS with all the connection. So he realized all the function coded in the instruction memory (add, sub,or, and, slt ,lw,sw,beq,jump).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.all;

ENTITY connection IS
        PORT (I_clk : in  STD_LOGIC;
                reset : in  STD_LOGIC

);

END connection;

ARCHITECTURE structure OF connection IS
        COMPONENT pc_unit
                PORT (
                I_clk : in  STD_LOGIC;
        I_nPC : in  STD_LOGIC_VECTOR (31 downto 0);
       reset : in  STD_LOGIC;
        O_PC : out  STD_LOGIC_VECTOR (31 downto 0)
                  );
        END COMPONENT;
        COMPONENT Instruction_Memory_VHDL
                PORT (address: in std_logic_vector(11 downto 0);
                dataOut: out  std_logic_vector(31 downto 0)
);
        END COMPONENT;
        COMPONENT register_file
                PORT (
        outA       : out std_logic_vector(31 downto 0);
   outB       : out std_logic_vector(31 downto 0);
   input      : in  std_logic_vector(31 downto 0);
   writeEnable : in  std_logic;
   regASel     : in  std_logic_vector(4 downto 0);
   regBSel     : in  std_logic_vector(4 downto 0);
   writeRegSel : in  std_logic_vector(4 downto 0);
   reset      : in  std_logic;
   clk        : in  std_logic
);
        END COMPONENT;
        COMPONENT depun_mux_out
                Port (
in1 : in std_logic_vector (31 downto 0); -- mux input1
in2 : in std_logic_vector (31 downto 0); -- mux input2
sel : in std_logic; -- selection line
dataout2 : out std_logic_vector (31 downto 0) -- output data
);
```

```
   End component;
        COMPONENT control_unit_VHDL
                Port (
Instruction: in std_logic_vector(5 downto 0);
 FuenteAlu,Escregister,leerMem,EscrMem,SaltoCond,MemaReg,RegDesr,saltoIncond  :
out std_logic ;
 Aluop : out std_logic_vector (1 downto 0)
);
   End component;

        COMPONENT ALU
                 Port (
   A, B     : in  STD_LOGIC_VECTOR(31 downto 0);  -- 2 inputs 32-bit
   ALU_Sel  : in  STD_LOGIC_VECTOR(2 downto 0);  -- 1 input 5-bit for selecting function
   ALU_Out   : out  STD_LOGIC_VECTOR(31 downto 0); -- 1 output 32-bit
   Carryout : out std_logic        -- Carryout flag
   );
        end component;

        COMPONENT SEXT
                 Port (
   data_in : in std_logic_vector(15 downto 0);
       data_out : out std_logic_vector(31 downto 0)
   );
        end component;

        COMPONENT ALU_Control_VHDL
                 port(
  ALU_Sel: out std_logic_vector(2 downto 0);
  ALUOp : in std_logic_vector(1 downto 0);
  ALU_Funct : in std_logic_vector(5 downto 0)
);
        end component;
        component  ram6116 --pb
port(
        address: in unsigned (9 downto 0);
        dataIn: in std_logic_vector(31 downto 0);
        dataOut: out std_logic_vector (31 downto 0);
        WE_b, CS_b, OE_b: in std_logic);
end component ;

component  jump_operation --pb
        Port (  PC_four : in  std_logic_vector (3 downto 0);
                        address : in std_logic_vector ( 25 downto 0);
                        new_address : out std_logic_vector ( 31 downto 0)
        );
end component ;
component  desp_2_izq

        Port ( In0 : in std_logic_vector (31 downto 0);
                        out0 : out std_logic_vector (31 downto 0)
        );
end component ;
component  adder

         Port ( InA : in std_logic_vector (31 downto 0);
                        InB : in std_logic_vector (31 downto 0);
                        out2 : out std_logic_vector (31 downto 0)
        );
end component ;
component  Mux_5_bits

        Port (
in1 : in std_logic_vector (4 downto 0); -- mux input1
in2 : in std_logic_vector (4 downto 0); -- mux input2
sel : in std_logic; -- selection line
dataout2 : out std_logic_vector (4 downto 0) -- output data
);
end component ;

        SIGNAL s1: STD_LOGIC_VECTOR (31 downto 0);
        Signal s2 : std_logic_vector (31 downto 0);
        signal s3 : std_logic_vector (4 downto 0);
        signal s4 : std_logic;
        signal s5 : std_logic;
        signal s6 : std_logic_vector (31 downto 0);
        signal s7 : std_logic_vector (31 downto 0);
        signal s8 : std_logic_vector (31 downto 0);
        signal s9 : std_logic_vector (31 downto 0);
        signal s10 : std_logic_vector (1 downto 0);
        signal s11,s15,s16,s19,s25,s24,s28,s29 : std_logic;
        signal s12 : std_logic_vector (2 downto 0);
        signal s13 : std_logic_vector (31 downto 0);
        signal s14 : std_logic_vector (31 downto 0);
        signal s17 : std_logic_vector (31 downto 0);
        signal s18 : std_logic_vector (31 downto 0);
        signal s20 : std_logic_vector (31 downto 0);
        signal s21 : std_logic_vector (31 downto 0);
        signal s22 : std_logic_vector (31 downto 0);
        signal s23 : std_logic_vector (31 downto 0);
        signal s26 : std_logic_vector (31 downto 0);
        signal s27 : std_logic_vector (31 downto 0);
        signal s30 : std_logic;
        signal s31 : unsigned (9 downto 0);



        FOR instruction1 : pc_unit USE ENTITY WORK.pc_unit;
        FOR instruction2 : Instruction_Memory_VHDL USE ENTITY
```

```
WORK.Instruction_Memory_VHDL;
        For instruction3 : control_unit_VHDL Use entity Work.control_unit_VHDL;
        For instruction4 : Mux_5_bits Use entity Work.Mux_5_bits;
        For instruction5 : register_file use entity Work.register_file;
        For instruction6 : SEXT use entity Work.SEXT;
        For instruction7 : depun_mux_out use entity Work.depun_mux_out;
        for instruction8 : ALU use entity Work.ALU;
        for instruction9 : ALU_Control_VHDL use entity Work.ALU_Control_VHDL;
        for instruction10 : ram6116 use entity Work.ram6116;
        for instruction11 : depun_mux_out use entity Work.depun_mux_out;
        for instruction12 : jump_operation use entity Work.jump_operation;
        for instruction13 : desp_2_izq use entity Work.desp_2_izq;
        for instruction14 : adder use entity Work.adder;
        for instruction15 : depun_mux_out use entity Work.depun_mux_out;
        for instruction16 : depun_mux_out use entity Work.depun_mux_out;
        for instruction17 : adder use entity Work.adder;

        BEGIN
        s26 <= x"00000004";
        s29 <= s25 AND s28;
        s30 <= '0';
        s31 <= unsigned (s13(9 downto 0));
        instruction1: pc_unit PORT MAP (I_clk, s20, reset, s1);

        instruction2: Instruction_Memory_VHDL PORT MAP (s1 (11 downto 0), s2 ) ;

        instruction3: control_unit_VHDL port map (s2 (31 downto
26),s11,s4,s16,s15,s25,s19,s5,s24,s10);

        instruction4: Mux_5_bits Port map (s2 (20 downto 16), s2 (15 downto 11),s5, s3);

        instruction5: register_file Port map (s7, s8, s21, s4, s2(25 downto 21),s2(20 downto
16), s3, reset, I_clk);

        instruction6: SEXT port map (s2(15 downto 0), s6);

        instruction7: depun_mux_out port map (s8,s6,s11,s9);

        instruction8: ALU port map (s7,s9,s12,s13,s28);

        instruction9: ALU_Control_VHDL port map (s12,s10,s2(5 downto 0));

        instruction10 : ram6116 port map (s31,s8,s14,s15,s30,s16);

        instruction11: depun_mux_out port map (s13,s14,s19,s21);

        instruction12 : jump_operation port map (s27 (31 downto 28),s2(25 downto 0),s17);

        instruction13 : desp_2_izq port map (s6,s18);
```

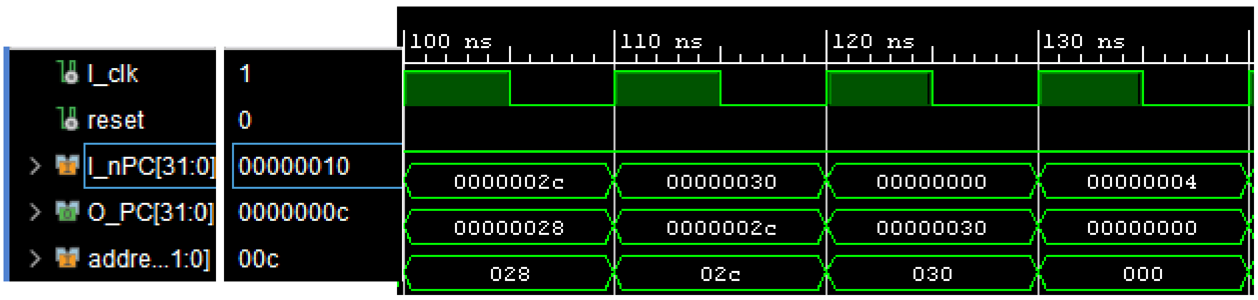
instruction14 : adder port map (s27,s18,s22);

instruction15 : depun_mux_out port map (s1,s18,s29,s23);

instruction16 : depun_mux_out port map (s27,s17,s24,s20);

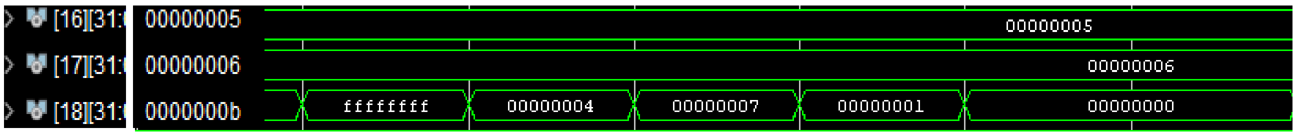
instruction17 : adder port map (s1,s26,s27);

END structure;



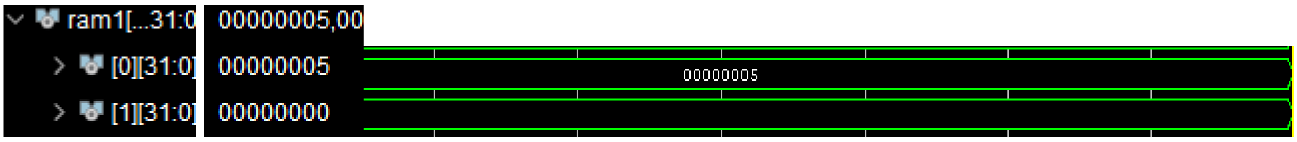
```
l_clk          1
reset          0
l_nPC[31:0]    00000010      0000002c   00000030   00000000   00000004
O_PC[31:0]     0000000c      00000028   0000002c   00000030   00000000
addre...1:0]   00c           028        02c        030        000
```

Here we can see the jump instruction because in the instruction memory we said that we hd to jump unit 00000000000000000000000000000000 instruction.



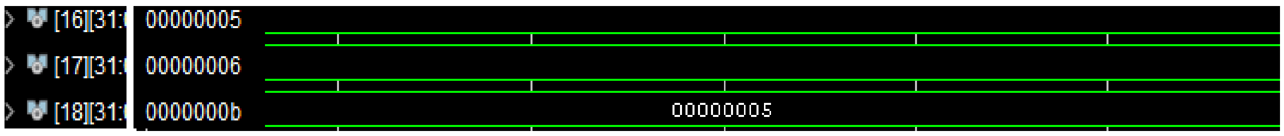
```
[16][31:   00000005                                                          00000005
[17][31:   00000006                                                          00000006
[18][31:   0000000b   ffffffff   00000004   00000007   00000001   00000000
```

So here we can see the addition, substraction, and, or and slt instruction betwenn the 16th and the 17th register. (5+6=b 5-6= fffffff 5and6=4 5or6=7 in 6>5 so 1 and 6<5 so 0) Then we put all the value in the 18th register. (all the value are in hexadecimal)



```
ram1[...31:0   00000005,00
  [0][31:0]    00000005      00000005
  [1][31:0]    00000000
```

Now we store (sw function) the 5 value in the data memory (adress 00000000000000000000000000000000)



```
[16][31:   00000005
[17][31:   00000006
[18][31:   0000000b      00000005
```

And now we load this value in the 18th register (lw function) .

Conclusion

To cut a long story short, it was very intresting to learn about microprocessor, about VHDL language, about FPGA. It related with my study and my professional project and I acquired a lot of knowledges thanks to this memory. I learnt to learn by myself without daily classes. And I complete all my objective except th implemetation in a FPGA. But I simulate several function of the microprocessor. We had a lot of problem and a small amount of time so we were reactive and we solved all of them. I'm happy to know a new coding language and a new software. I'm also happy to understand the all functionement of a simple microprocessor. I'm also happy to see that I'm able to write a memory in a small amout of time.

I'd like to thank Santiago Caceres for all the time and ressources that he gaves to me. Without this I wouldn't be able to achieve my work.

Bibliography

(1) → https://www.irisa.fr/alf/downloads/michaud/ESIR2_2013.pdf (2016)

(2) → University Course (Bayonne institue) mde by M Dalmau (2007)

(3) → University course based on one reference : « Architectures des ordinateurs » M.C.BELAID » (2015) https://www.academiepro.com/uploads/cours/2015_08_21_chapitre_3_le_microprocesseur.pdf

(4) → "Computer organization and design" David A. Petterson , John L. Hennessy

(5) → "Diseño del procesador MIPS" Sergio Barrachina Mir (2006)