Design of a Central Processing Unit

2021

TECHNICAL UNIVERSITY OF CLUJ NAPOCA

Grama Mălina

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

DESIGN OF A CENTRAL PROCESSING UNIT

Contents

1. Introduction

1.1 Context [1]

The goal of this project is to design and implement a processor, which is a key component of computer hardware. The processor is able to execute instructions comprising a computer program, like basic arithmetic, logic, controlling, and input/output operations specified by the instructions in the program. In other words, the CPU acts as the “brain” of the machine, controlling the operations carried out by the computer.

1.2 Specifications

The CPU must have the following **features**:

- approximately 20 instructions (arithmetic, logical, data transfer, jump)
- internal registers: accumulator, 8 general registers
- addressing modes: immediate, direct and through a dedicated register
- implemented with flip-flops and logic gates (in VHDL)

The device will be simulated in the IDE provided by Vivado and then programmed into a basys3 board.

1.3 Objectives

The design of a CPU is a complex undertaking. The main goal of CPU design is to produce an architecture that can execute instructions in the fastest, most efficient way possible.

After researching the difference between the Von Neumann and the Harvard models, I decided to implement the processor after the **Harvard Architecture**, because it is simpler than the Harvard architecture. The design and development of the Control Unit is simplified, cheaper and faster. I also decided to implement the CPU following a **RISC** architecture, because RISC processors only use simple instructions that can be executed within one clock cycle. I will implement a **16-bit address and data bus**, because the instruction set is not that large (it does not require a 32-bit address and data bus). The architecture will be a **Single-Cycle**, **Register-Memory** one, because of its ease of implementation.

Project Planning [3]

Meeting 2 (13-10-2021):

- have an outline of the specifications and the objectives of the project (Project Proposal)
- start doing research about the project and compose the Bibliographic Study
- sketch the project planning on meetings

Meeting 3 (27-10-2021):

- define the instructions and the encodings of these instructions (RTL Statements)
- specify the data-path components and their interconnection
- Select a set of data-path components and establish clocking methodology
- Find the worst-time propagation delay in the data-path to determine the data-path clock cycle (CPU clock cycle)

Meeting 4 (10-11-2021):

- Assemble data-path meeting the requirements

- Assemble the control logic
- Start implementing some of the basic components (such as ALU, Registers etc)

Meeting 5 (24-11-2021):

- Have most of the proposed design and features implemented in VHDL

Meeting 6 (8-12-2021):

- Test the fully implemented design on an FPGA board and debug
- Start working on extra features (if time allows it)

Meeting 7 (5-1-2022):

- Writing the documentation and simple programs for testing

2. Bibliographic Study

2.1 CPU

The CPU is the active part of a computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and much more. Going even deeper the processor logically comprises two main components: data path and control. The data path performs the arithmetic operations, and control tells the data path, memory, and I/O devices what to do according to the wishes of the instructions of the program.

2.2 Harvard Architecture (vs Von Neumann Architecture) [5], [6]

When discussing how memory is accessed at the CPU level, there are two designs to consider. The first is a Von Neumann architecture, and the second is a Harvard architecture. The major difference between the two architectures is that the Harvard architecture has separate storage and signal pathways for instructions and data. It contrasts with the von Neumann architecture, where program instructions and data share the same memory and pathways.

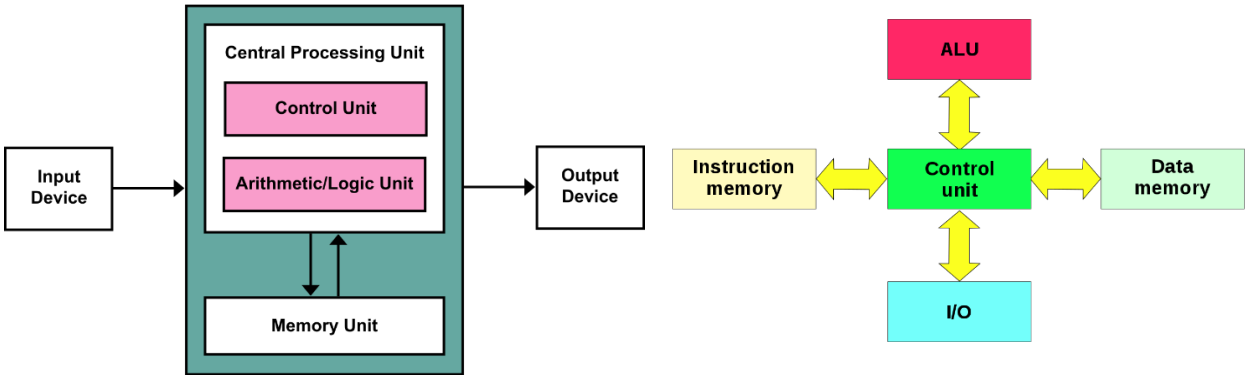


Figure 1: Von Neumann and Harvard Architectures

2.3 RISC vs CISC [7], [8]

A Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions rather than the highly-specialized set of instructions typically found in other architectures. RISC is an alternative to the Complex Instruction Set Computing (CISC) architecture and is often considered the most efficient CPU architecture technology available today.

With RISC, a central processing unit (CPU) implements the processor design principle of simplified instructions that can do less but can execute more rapidly. The result is improved performance. A key RISC feature is that it allows developers to increase the register set and increase internal parallelism by increasing the number of parallel threads executed by the CPU and increasing the speed of the CPU's executing instructions.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

2.4 Register/Memory Architecture [4]

In computer engineering, a register–memory architecture is an instruction set architecture that allows operations to be performed on (or from) memory, as well as registers.

Figure 2: Register-Memory Instruction Format

3. Analysis

3.1 Instruction Set Architecture

The first step in starting to design a processor is to define an instruction set and set a binary encoding specific to each instruction. From the project specification we know that we will need to define arithmetic, logic, transfer and jump instructions. I decided that my instructions will be 32 bits wide, and because the architecture of the CPU will be a RISC type, there will be no direct operations on memory.

The **instruction format** will be the following:

- first byte represents **opcode**
- second byte represents the **register** on which the instruction is working against
- third and fourth byte represent **the immediate(data)/ the address / the register**, depending on the operation

The CPU's Instruction Set Architecture will contain **load** and **store** instructions, **branch** instructions, **jump** instructions, arithmetic and logic instructions (**ADD, SUB, AND, OR, XOR, NOT**), and a **NoOp** instruction.

The Processor will contain 8 general purpose registers (R0 - which will always be zero, and R1 –> R7). They will be encoded like this: the first register (R1) will have the code 0x01, R2 will have the code 0x02, etc.

There will be **4 instruction formats**:

- **R-type instructions**: used for arithmetical/logical operations between two registers (destination and source, the result will be stored in the destination register in the Store phase of the cycle)

31 – 24	23 – 16	15 – 8	7 – 0
opcode	Register destination	0x00	Register source

- **I-type instructions**: used for arithmetical/logical operations between a register and an immediate value (result will be stored in the destination register in the Store phase of the cycle)

31 – 24	23 – 16	15 – 0
opcode	Register destination	Immediate

- **M-type instructions**: used for arithmetical/logical operations between a register and a value stored in memory (result will be stored in the destination register in the Store phase of the cycle)

31 – 24	23 – 16	15 – 0
opcode	Register destination	Memory Address

- **J-type instructions**: used for jump/ branch instructions

31 – 24	23 – 0
opcode	Target

Bellow you will find a table with the instructions' names, descriptions, RTL expressions, and encodings expressed in hexadecimal.

INSTRUCTION	DESCRIPTION	RTL EXPRESSION	OPCODE	REGISTER	IMM\ADDR High	REGISTER\IMM\ADDR Low
NoOp	No Operation	- PC <- PC + 4	0x00	0x00	0x00	0x00
Load Immediate in Register ***lir $rd, imm***	Loads an immediate value into a specified register	RD <- Immediate PC <- PC + 4	0x10	R1-R32 (RD)	Imm high	Imm low
Load Register Register ***lrr $rd, $rs***	Loads the content from a register to another specified register	RD <- RS PC <- PC + 4	0x11	R1-R32 (RD)	0x00 (unused)	R1-R32 (RS)
Load from memory ***lm $rd, addr***	Load the contents from a memory address into a specified register	RD <- M[Address] PC <- PC + 4	0x12	R1-R32 (RD)	Addr high	Addr high
Store register in memory ***sm $rd, addr***	Stores the contents of a register in memory, at a specified memory address	M[Address] <- RD PC <- PC + 4	0x20	R1-R32 (RD)	Addr high	Addr low
Compare register register ***cmp $rd, $rs***	Compares the value of a register with the value of another register, setting the corresponding flags	Flags <- RD - RS PC <- PC + 4	0x30	R1-R32 (RD)	0x00	R1-R32 (RS)

Compare register memory address ***cmpa $rd, addr***	Compares the value of a register with the value found at a memory address, setting the corresponding flags	Flags <- RD – S_ext(M[addr]) PC <- PC + 4	0x31	R1-R32 (RD)	Addr high	Addr low
Jump ***jmp target***	Jumps to the calculated address	PC <- PC[31:28] || target ||00	0x40	Target		
Note: The following branch instructions will only be used after a compare register register instruction						
Branch on equal ***beq target***	Branches if the two registers (compared previously) are equal (if the zero flag is set to 1)	If (Z=1) then PC <- PC + 4 + S_Ext(target) << 2 else PC <- PC + 4	0x41	0x00	Target high	Target low
Branch on greater than ***bgt target***	Branches if between the two registers (compared previously), the first one is greater than the second one (if the zero flag is set to 0, the sign flag is set to 0)	If (((SxorO)orZ)=0) then PC <- PC + 4 + S_Ext(target) << 2 else PC <- PC + 4	0x42	0x00	Target high	Target low
Branch on less than ***blt target***	Branches if between the two registers (compared previously), the first one is less than the second one (if the zero flag is set to 0, the sign flag is set to 1)	If (SxorO=1) then PC <- PC + 4 + S_Ext(target) << 2 else PC <- PC + 4	0x43	0x00	Target high	Target low
Addition register register ***add $rd, $rs***	Adds the values from two registers, and stores the result in the first register	RD <- RD + RS PC <- PC + 4	0x50	R1-R32 (RD)	0x00	R1-R32 (RS)
Addition register immediate ***addi $rd, imm***	Adds the values from a register and an immediate value, and stores the result in the register	RD <- RD + Imm PC <- PC + 4	0x51	R1-R32 (RD)	Imm high	Imm low
Addition register memory ***addm $rd, addr***	Adds the values from a register and a memory address, and stores the result in the register	RD <- RD + M[Addr] PC <- PC + 4	0x52	R1-R32 (RD)	Addr high	Addr low
Subtraction register register	Subtracts the values from two	RD <- RD – RS PC <- PC + 4	0x53	R1-R32 (RD)	0x00	R1-R32 (RS)

sub $rd, $rs	registers, and stores the result in the first register					
Subtraction register immediate ***subi $rd, imm***	Subtracts the values from a register and an immediate value, and stores the result in the register	RD <- RD - Imm PC <- PC + 4	0x54	R1-R32 (RD)	Imm high	Imm low
Subtraction register memory ***subm $rd, addr***	Subtracts the values from a register and a memory address, and stores the result in the register	RD <- RD - M[Addr] PC <- PC + 4	0x55	R1-R32 (RD)	Addr high	Addr low
AND ***and $rd, $rs***	Bitwise ands two registers and stores the result in the first register	RD <- RD & RS PC <- PC + 4	0x60	R1-R32 (RD)	0x00	R1-R32 (RS)
AND Immediate ***andi $rd, imm***	Bitwise ands a register and an immediate value and stores the result in the register	RD <- RD & Imm PC <- PC + 4	0x61	R1-R32 (RD)	Imm high	Imm low
AND Memory ***andm $rd, addr***	Bitwise ands a register and a value stored at a memory address and stores the result in the register	RD <- RD & M[Addr] PC <- PC + 4	0x62	R1-R32 (RD)	Addr high	Addr low
OR ***or $rd, $rs***	Bitwise ors two registers and stores the result in the first register	RD <- RD | RS PC <- PC + 4	0x63	R1-R32 (RD)	0x00	R1-R32 (RS)
OR Immediate ***ori $rd, imm***	Bitwise ors a registers and an immediate unsigned value and stores the result in the register	RD <- RD | Imm PC <- PC + 4	0x64	R1-R32 (RD)	Imm high	Imm low
OR Memory ***orm $rd, addr***	Bitwise ors a register and a value stored at a memory address and stores the result in the register	RD <- RD | M[Addr] PC <- PC + 4	0x65	R1-R32 (RD)	Addr high	Addr low
XOR ***xor $rd, $rs***	Bitwise xors two registers and stores the result in the first register	RD <- RD ^ RS PC <- PC + 4	0x66	R1-R32 (RD)	0x00	R1-R32 (RS)
XOR Immediate ***xori $rd, imm***	Bitwise xors a registers and an immediate unsigned value and	RD <- RD ^ Imm PC <- PC + 4	0x67	R1-R32 (RD)	Imm high	Imm low

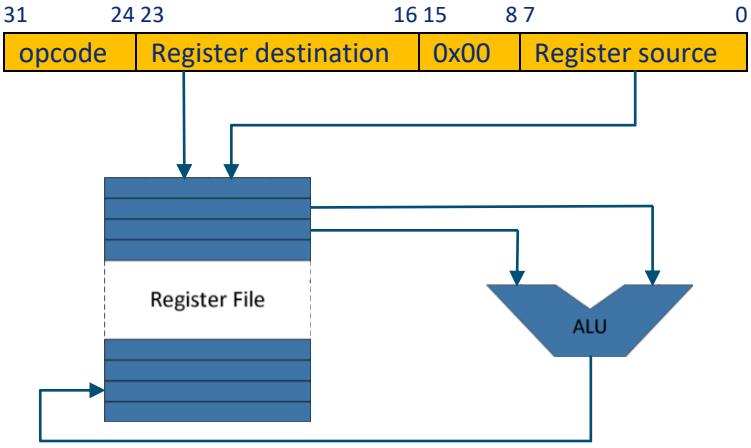
	stores the result in the register					
XOR Memory ***xorm $rd, addr***	Bitwise xors a register and a value stored at a memory address and stores the result in the register	RD <- RD ^ M[Addr] PC <- PC + 4	0x68	R1-R32 (RD)	Addr high	Addr low
NOT ***not $rd***	Negates the bits of the register and stores the result in the same register	RD <- ~RD PC <- PC + 4	0x69	R1-R32 (RD)	0x00	0x00

Table 1: Instruction Set

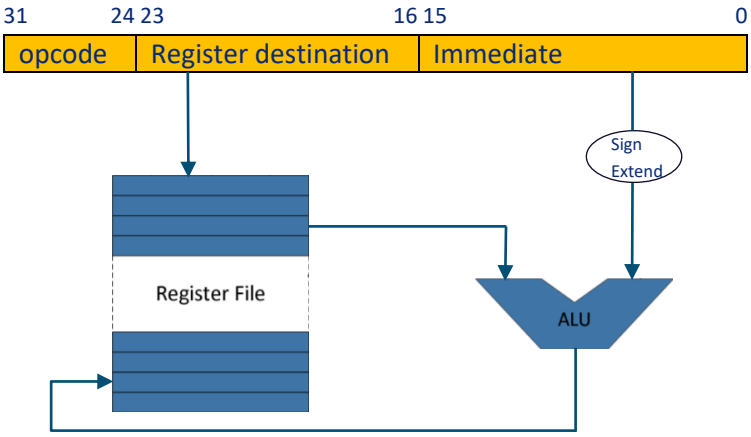
The RTL Expressions describe the flow of data of the architecture for each instruction, which will help us in designing our CPU. These expressions can be found in the table above.

Looking at the RTL expressions, we can design small schematics describing the data flow in the CPU:

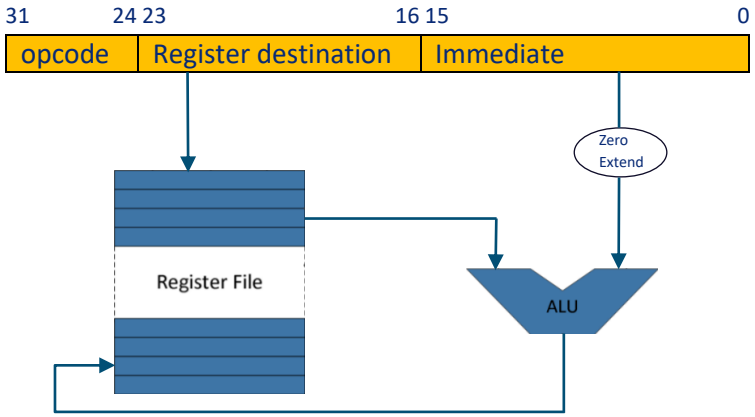
- **Arithmetic/Logical R-type Instructions**: *add $rd, $rs -> RF[rd] <- RF[rd] + RF[rs]*



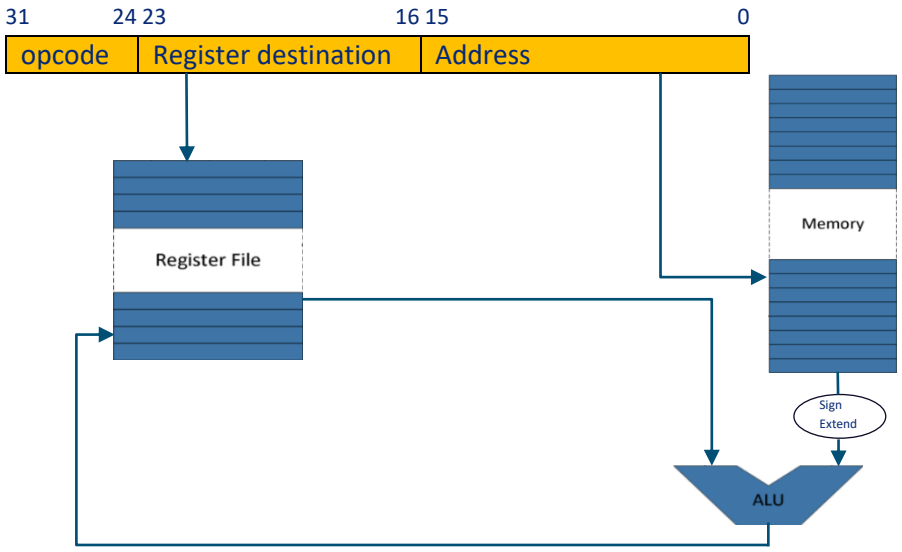
- **Immediate Arithmetic Instructions**: *addi $rd, imm -> RF[rd] <- RF[rd] + S_Ext(imm)*



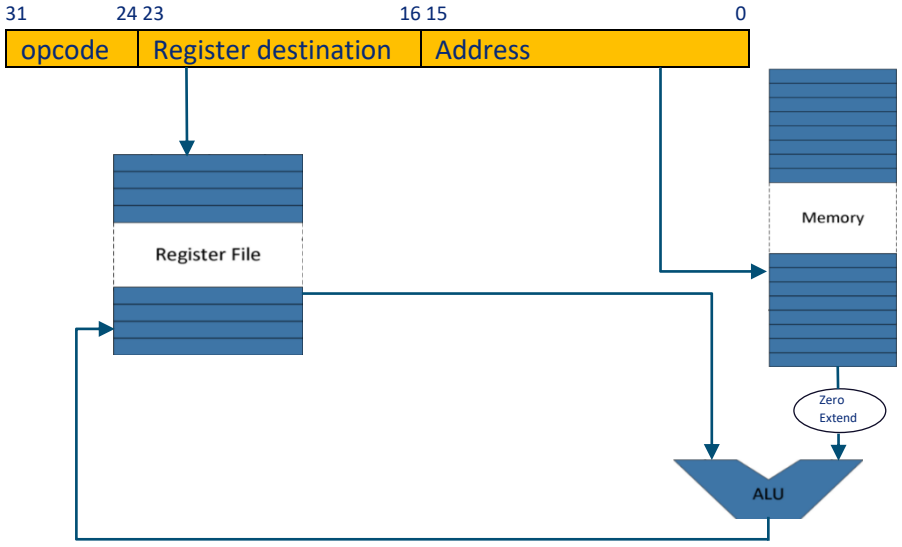
- **Immediate Logical Instructions**: *andi $rd, imm -> RF[rd] <- RF[rd] & Z_Ext(imm)*



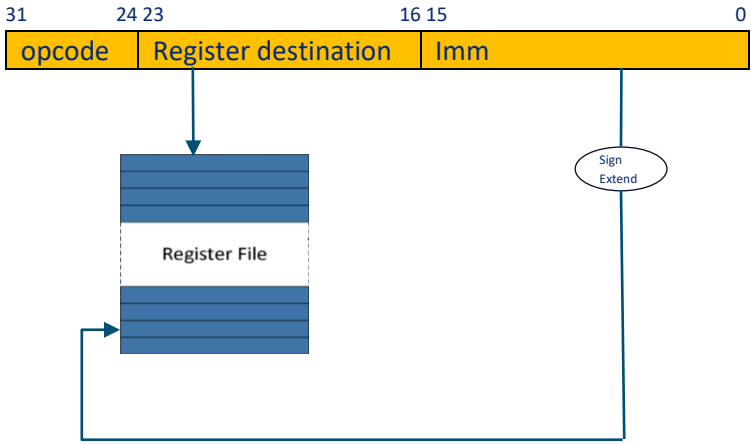
- **Memory Arithmetic Instructions**: *addm $rd, addr -> RF[rd] <- RF[rd] + S_Ext(M[addr])*



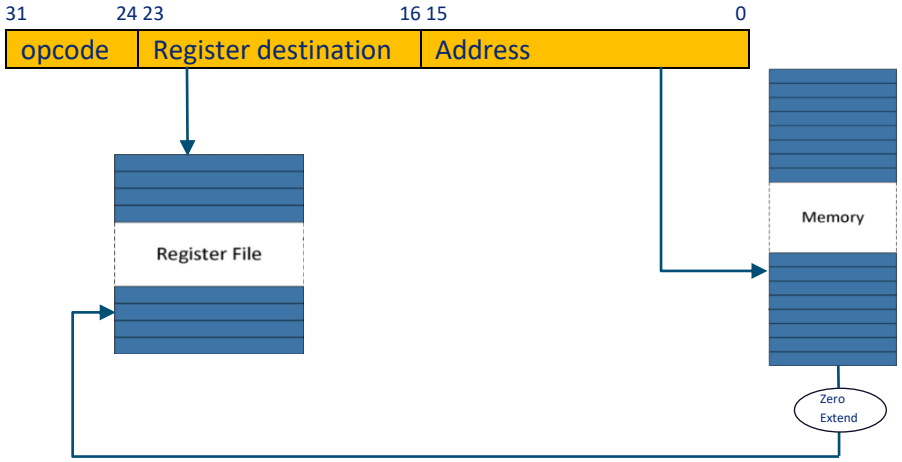
- **Memory Logical Instructions**: *andm $rd, addr -> RF[rd] <- RF[rd] | Z_Ext(M[addr])*



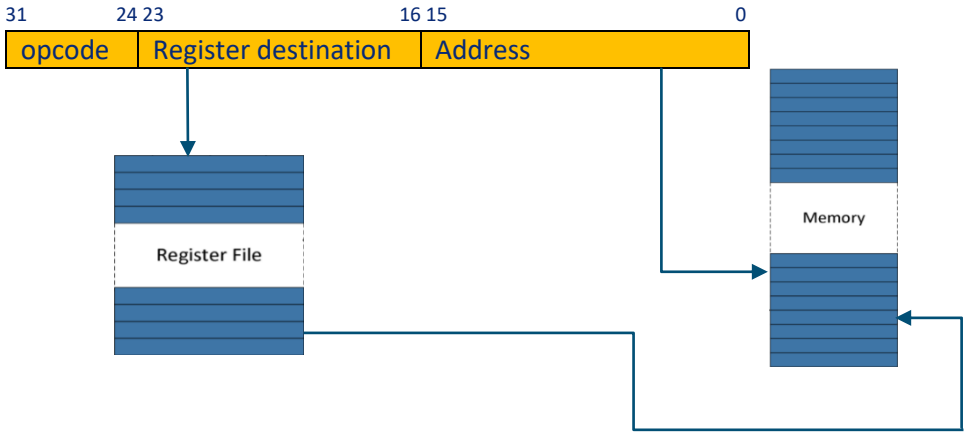
- **Load Immediate in Register**: *lir $rd, imm -> RF[rd] <- S_ext(imm)*



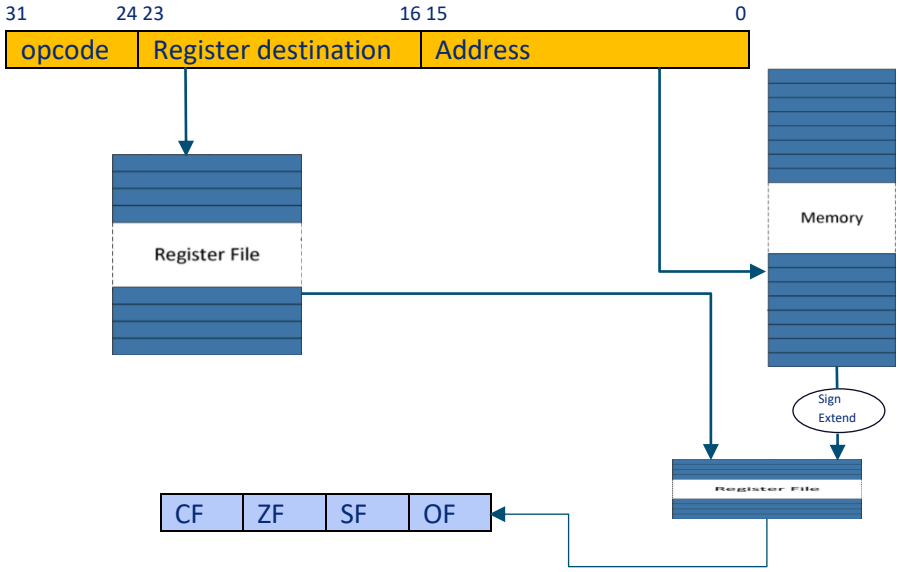
- **Load from Memory**: *lm $rd, addr -> RF[rd] <- Z_Ext(M[addr])*



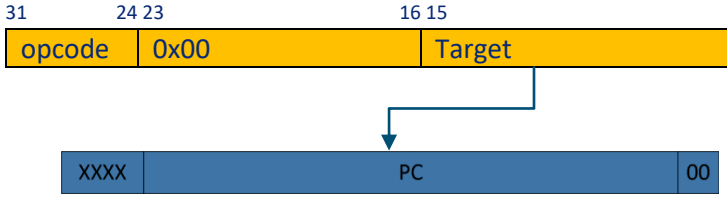
- **Store register in Memory**: *sm $rd, addr -> M[addr] <- RF[rd]*



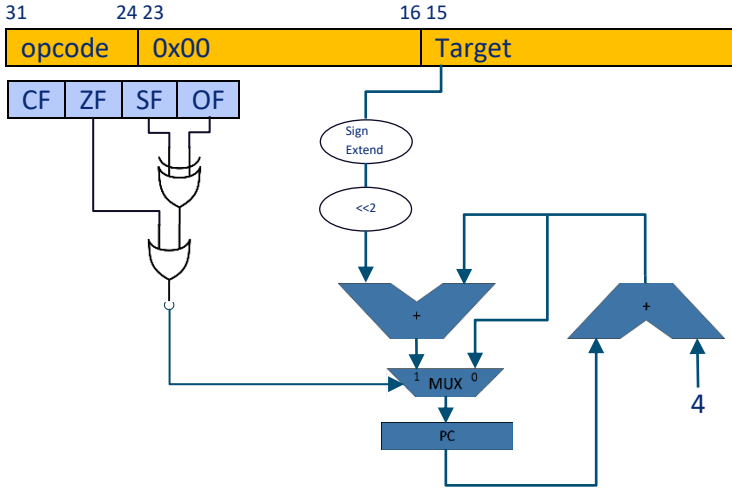
- **Compare register memory address**: *cmpa $rd, addr -> Flags <- RD – S_ext(M[addr])*



- **Jump**: *jmp target -> PC <- PC[31:18] || target ||00*



- **Branch on greater than:** *bgt target* -> If (((S xor O) or Z) = 0) then
  PC <- PC + 4 + S_Ext(imm) << 2
  else PC <- PC + 4



3.2 Components

Next, I will briefly explain the main components that will be used in the CPU implementation.

3.2.1 Registers

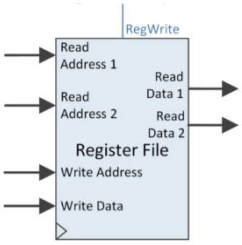


Figure 3: Register File

For the **32 general purpose registers**, a Register File (32x32 bits) will be used, which is an array of processor registers. They will be used to stage data between memory and the functional units in the CPU. Since the register file runs at the full speed of the processor, it must be small and fast. The real register file is usually implemented as a small, fast SRAM memory with multiple accesses. The register file that I will use has two read addresses and one write address. The registers corresponding to the locations indicated by the two read addresses (Read Address 1 & Read Address 2) are delivered at the two output ports (Read Data 1 & Read Data 2). The data provided at the write data input port is written in the register indicated by the write address (Write Address, when the Write control signal (RegWrite) is asserted. The read operations are asynchronous, while the write operation is synchronous. Therefore, the register file supports 2 reads and one write in each clock cycle.

An additional register will be needed to store the **flags** in the processor. This will be a simple, four bit status register that will contain the current state of the CPU, reflecting the result of arithmetic operations that will be made in the ALU. I will implement a total of four flags:

- **Carry Flag (CF),** used to indicate when an arithmetic carry or borrow has been generated out of the most significant arithmetic logic unit (ALU) bit position.
- **Zero Flag (ZF),** that will be used to check the result of an arithmetic operation
- **Sign Flag (SF),** that will be used to indicate whether the result of the last mathematical operation produced a value in which the most significant bit (the left most bit) was set.
- **Overflow Flag (OF),** used to indicate when an arithmetic overflow has occurred in an operation, indicating that the signed two's-complement result would not fit in the number of bits used for the result.

Another register will be used as an **accumulator**. Arithmetical and logical operations done in the ALU will have their results stored in this particular register. The accumulator holds the data being processed and the results of processing.

Some special registers that will be used in the CPU:

- The **Program Counter** register (**PC**) will hold the memory address of the next instruction to be fetched from main memory.

3.2.2 Memory

Because the processor is implemented following the Harvard architecture, two memories will be used: one of them, the **Instruction Memory,** is used to store the instructions that will be used in the CPU. The **Data Memory** will be used to store the data that will be used in memory addressing instructions.

The **Data Memory** component will be a 256x32 SRAM, which is fast and also directly accessible by the processor. It will have two input busses: a 32-bit Address bus, and a Write Data bus. The memory will have one 32-bit output bus, that will represent a Read Data signal. The memory will also have two control signals: MemWrite and MemRead. Both writing and reading from the memory will be synchronous with the clock.

The **Instruction Memory** component will be an 256x32 ROM memory. It will have one address input which will come from the PC, and one output which will represent the current instruction found at the Address given. We can only read from this memory, and the reading will be synchronous.

3.2.3 ALU

An **Arithmetic Logic Unit** will be used to perform the operations required for the CPU. It will need to perform arithmetic and bitwise operations on signed integer binary numbers. The operations performed by the ALU are the following: **addition**, **subtraction**, logical **and**, logical **or**, logical **xor**, logical **not**. All of the operations will modify values in the flags register.

4. Design

4.1 Data Path

The CPU will have five execution steps: ***instruction fetch – decode – data fetch (if working with memory) – execute – store***.

4.1.1 Instruction Fetch

The control unit **fetches** the next instruction from memory. It places the address from the PC on the *address bus* and places a *read* command on the *control bus* to the memory unit. The memory unit then reads the bytes stored at the specified address and sends them to the control unit on the *data bus*. The instruction register (IR) stores the bytes of the instruction received from the memory unit. The control unit also increments the PC's value to store the address of the new next instruction to fetch.

4.1.2 Decode

The control unit **decodes** the instruction stored in the IR. It decodes the instruction bits that encode which operation to perform and the bits that encode where the operands are located. The instruction bits are decoded based on the ISA's definition of the encoding of its instructions. The control unit also fetches the data operand values from their locations (from CPU registers, memory, or encoded in the instruction bits), as input to the processing unit.

4.1.3 Data Fetch

If the processor fetches in the first phase an instruction which works on memory, the data will have to be fetched from the memory address mentioned in the instruction.

4.1.4 Execute

The processing unit **executes** the instruction. The ALU performs the operation defined in the instruction on the data operands.

4.1.5 Store

The control unit **stores** the result to memory. The result of the processing unit's execution of the instruction is stored to memory. additionally, this result will also be stored in the accumulator register. The control unit writes the result to memory by placing the result value on the *data bus*, placing the address of the storage location on the *address bus*, and placing a *write* command on the *control bus*. When received, the memory unit writes the value to memory at the specified address.

4.2 Control Unit

The control unit (**CU**) is a component of the CPU that directs the operation of the processor. It tells the computer's memory and arithmetic and logic unit how to respond to the instructions that have been sent to the processor. The CU is responsible with decoding the instruction and sending several signal bits to the rest of the components of the CPU. The signals of the control unit are the following:

- **Branch –** for branch operations
- **Jump –** for jump operations
- **MemWrite –** for writing into the Data Memory
- **MemRead –** for reading from the Data Memory
- **R –** 1 if the second operand is $RS, 0 otherwise
- **RegWrite –** for writing in the Register File

- **MemToReg –** for choosing if the data written back in the Register File comes from the result of the ALU operation or from the MDR (from memory)
- **ALUSrcA –** selecting the first operand for the ALU
- **ALUSrcB –** selecting the second operand for the ALU
- **ALUControl –** selecting the operation performed by the ALU, based on the opcode
- **FlagEnable –** for enabling the flag register to be set according to the result of the ALU operation
-

The next table shows the control unit signals for each instruction:

	MemWrite	MemRead	R	RegWrite	MemtoReg	ALUSrcA	ALUSrcB	ALUCntrl	FlagEnable	Branch	Jump
lir $rd, imm	0	0	0	1	0	1	1	000	0	0	0
lrr $rd, $rs	0	0	1	1	0	1	0	000	0	0	0
lm $rd, addr	0	1	0	1	1	-	-	-	0	0	0
sm $rd, addr	1	0	0	0	-	0	2	000	0	0	0
cmp $rd, $rs	0	0	1	0	-	0	0	001	1	0	0
cmpa $rd, addr	0	1	0	0	-	0	3	001	1	0	0
jmp target	0	0	0	0	-	-	-	-	0	0	1
beq target	0	0	0	0	-	-	-	-	0	1	0
bgt target	0	0	0	0	-	-	-	-	0	1	0
blt target	0	0	0	0	-	-	-	-	0	1	0
add $rd, $rs	0	0	1	1	0	0	0	000	1	0	0
addi $rd, imm	0	0	0	1	0	0	1	000	1	0	0
addm $rd, addr	0	1	0	1	0	0	3	000	1	0	0
sub $rd, $rs	0	0	1	1	0	0	0	001	1	0	0
subi $rd, imm	0	0	0	1	0	0	1	001	1	0	0
subm $rd, addr	0	1	0	1	0	0	3	001	1	0	0
and $rd, $rs	0	0	1	1	0	0	0	010	1	0	0
andi $rd, imm	0	0	0	1	0	0	1	010	1	0	0
andm $rd, addr	0	1	0	1	0	0	3	010	1	0	0
or $rd, $rs	0	0	1	1	0	0	0	011	1	0	0
ori $rd, imm	0	0	0	1	0	0	1	011	1	0	0
orm $rd, addr	0	1	0	1	0	0	3	011	1	0	0
xor $rd, $rs	0	0	1	1	0	0	0	100	1	0	0
xori $rd, imm	0	0	0	1	0	0	1	100	1	0	0
xorm $rd, addr	0	1	0	1	0	0	3	100	1	0	0
not $rd	0	0	0	1	0	0	-	101	1	0	0

Table 2: Control Unit signals based on the operation

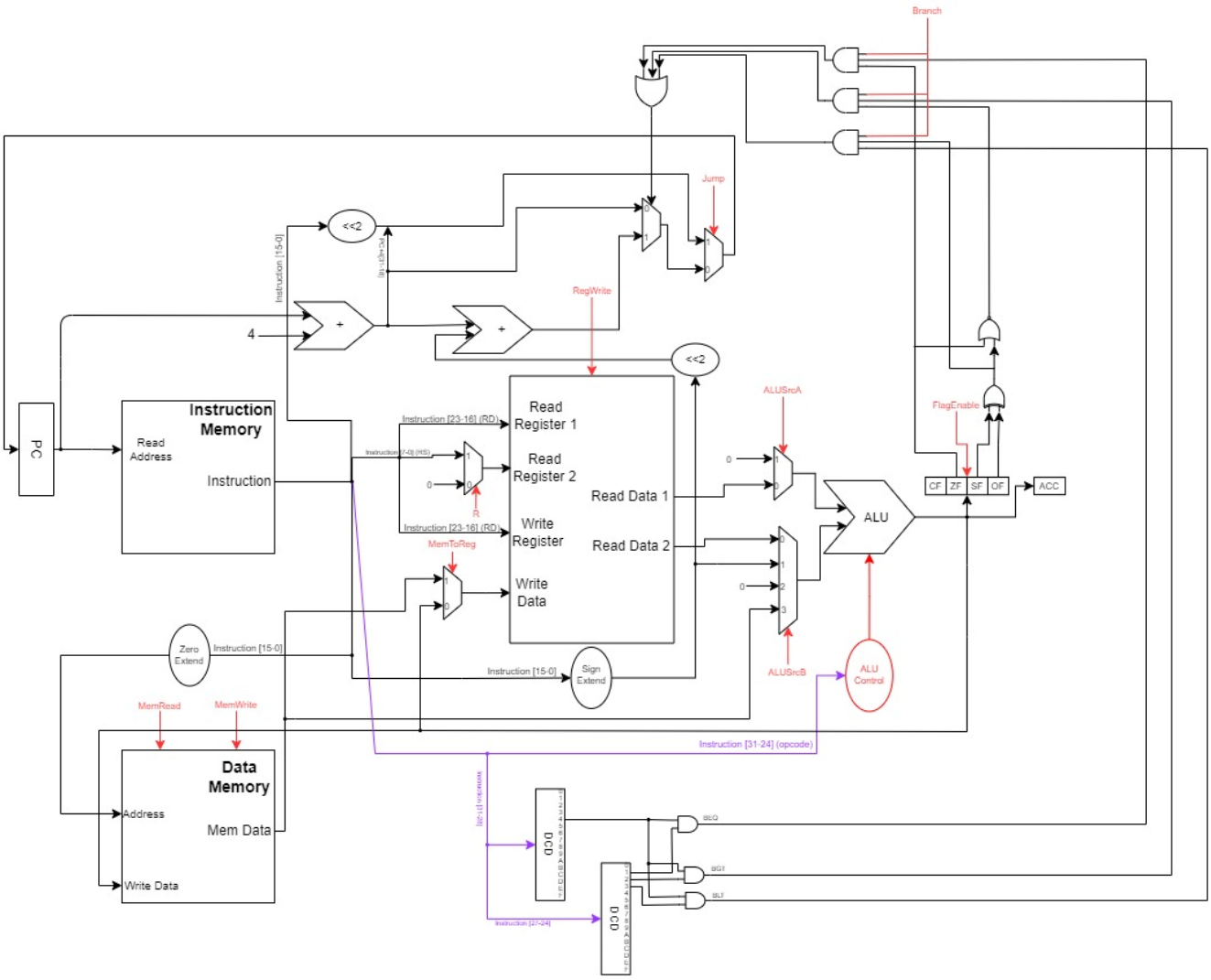


Fig 4 – Schema of the CPU

5. Implementation

The CPU is designed Behaviorally using VHDL, as described in figure 4. The more complex blocks will have dedicated components, such as the ALU, Control Unit, Memory, Register File, Program Counter, Extension Unit, Decoder, Instruction Register and Flag Register. Then, using port maps and processes, the components will be used to describe the behavioral architecture of the whole CPU.

5.1 Arithmetic Logic Unit

The ALU will perform operations on signed integers (2's Complement will be used). It has as input signals the two 32-bit operands A and B and a 3-bit opcode which will specify the operation to be performed by the component, enumerated in the following table:

Operation	Effect	Opcode
Addition	A + B	000
Subtraction	A – B	001
AND	A and B	010
OR	A or B	011
XOR	A xor B	100

NOT	(not)A	101

The ALU will have as output the result of the selected operation, as well as the four 1-bit signals that will be written in the Flag Register: Zero, Carry, Sign and Overflow. These will be computed as follows:

- The **Zero** signal will be 1 if and only if the result is equal to 0.
- The **Sign** signal will be 1 if and only if the result is a negative number (having the sign bit 1).
- The **Overflow** signal will be 1 in the following situations:

Addition (000)	A – positive B – positive	A – positive B – negative	A – negative B – positive	A – negative B – negative
result - positive	-	-	-	overflow
result - negative	overflow	-	-	-

Subtraction (001)	A – positive B – positive	A – positive B – negative	A – negative B – positive	A – negative B – negative
result - positive	-	-	overflow	-
result - negative	-	overflow	-	-

- The **Carry** signal will be 1 whenever an arithmetic carry or borrow has been generated out of the most significant arithmetic logic unit (ALU) bit position.

The ALU architecture is based on four processes: the first process sets the operation to be performed based on the opcode, the second process computes the extended result (the first bit of the extended result is used in setting the carry flag), the third process computes the result of the operation, and the final process is responsible for computing the flags. I will insert the second and fourth processes bellow:

```
a_plus_b  <= STD_LOGIC_VECTOR(signed('0'&(a)) + signed('0'&(b)));
a_minus_b <= STD_LOGIC_VECTOR(signed('0'&(a)) - signed('0'&(b)));

-- based on the selected operation, update the reg signal; reg is used in computing the carry signal
 process(A,B,enum_op,result_sig)
 begin
     case enum_op is
         when op_add       => reg <= a_plus_b;
         when op_sub       => reg <= a_minus_b;
         when op_and       => reg <= '0' & (A and B);
         when op_or        => reg <= '0' & (A or B);
         when op_xor       => reg <= '0' & (A xor B);
         when op_neg       => reg <= '0' & (not B);
         when op_nop       => reg <= '0' & x"00000000";
     end case;
 end process;
```

Fig 5 – VHDL Process for computing the extended result

```
-- compute the flag signals
process(A,B,reg,enum_op,result_sig)
begin
    -- set zero flag
    if (result_sig = x"00000000") then
        zero <= '1';
    else
        zero <= '0';
    end if;

    -- set carry flag
    carry <=  reg(32);

    -- set sign flag
    sign <= result_sig(31);

    -- set overflow flag
    if (enum_op = op_add) then
        if (A(31) = '0' and B(31) = '0') then
            if (result_sig(31) = '1') then
                overflow <= '1';
            else
                overflow <= '0';
            end if;
        elsif (A(31) = '1' and B(31) = '1') then
            if (result_sig(31) = '0') then
                overflow <= '1';
            else
                overflow <= '0';
            end if;
        else
            overflow <= '0';
        end if;
    elsif (enum_op = op_sub) then
        if (A(31) = '0' and B(31) = '1') then
            if (result_sig(31) = '1') then
                overflow <= '1';
            else
                overflow <= '0';
            end if;
        elsif (A(31) = '1' and B(31) = '0') then
            if (result_sig(31) = '0') then
                overflow <= '1';
            else
                overflow <= '0';
            end if;
        else
            overflow <= '0';
        end if;
    else
        overflow <= '0';
    end if;
end process;
```

Fig 6 – VHDL Process for computing the flags

5.2 Program Counter

The PC is a simple register which takes as an input the next instruction address from the adder and outputs it synchronously at the next clock cycle, only if the PCen control signal is '1'. If the reset signal is 1, then the PC will start from the value 0 (the reset is asynchronous).

```
entity PC is
    Port ( clk : in STD_LOGIC;
           PC_Enable : in STD_LOGIC;
           PC_Reset : in STD_LOGIC;
           PC_D : in STD_LOGIC_VECTOR (31 downto 0);
           PC_Q : out STD_LOGIC_VECTOR (31 downto 0));
end PC;

architecture Behavioral of PC is
begin

    process(clk, PC_Enable, PC_Reset)
     begin
        if(rising_edge(clk)) then
            if (PC_Enable = '1') then
                PC_Q <= PC_D;
            end if;
        end if;
        if(PC_Reset = '1') then
            PC_Q <= x"00000000";
        end if;
    end process;

end Behavioral;
```

Figure 7 – VHDL implementation of the PC component

5.3 Data Memory

The data memory is implemented as a 255x32 RAM memory, which holds data. It takes as inputs a 32-bit address, which gives the address of the data to be fetched, and a 32-bit Write Data, for writing data into the memory (Store Instructions). As output, the memory component gives us the data which address we specified as input, only when the MemRead control signal is '1'. The read and write operations are synchronous. We also have a MemWrite control signal. When it is set to '1', the data on the Write Data input is written into the memory, at the address specified as an input.

```
type ram_type is array (0 to 255) of std_logic_vector(31 downto 0);
signal RAM:ram_type:=(
        x"00000000",
        x"00000001",
        x"00000002",
        x"00000003",
        x"00000004",
        x"00000005",
        x"00000006",
        x"00000007",
        x"00000008",
        x"00000009",
        x"0000000A",
        x"0000000B",
        x"0000000C",
        x"0000000D",
        x"0000000E",
        x"0000000F",
        x"00000010",
        others => x"00000000");
```

```
process(clk, addr, MemWrite, MemRead)
begin
    if(falling_edge(clk)) then
        if MemWrite = '1' then
            RAM(conv_integer(addr)) <= WriteData;
        end if;
        if MemRead = '1' then
            data <= RAM(conv_integer(addr));
        end if;
    end if;
end process;
```

Fig 9 – VHDL architecture of the Memory component

5.4 Instruction Memory

The instruction memory is implemented as a 255X32 ROM memory, which contains the instructions to be performed by the CPU. It has as input the address of the instruction to be fetched (which comes from the program counter), and as output, the 32-bit binary encoding of the instruction found at the specified address. Only read operations are permitted, and they are done synchronously. No control signals are present, because instructions need to be read from the memory continuously.

```
type rom_array is array (0 to 255) of STD_LOGIC_VECTOR(31 downto 0);
constant rom : rom_array := (
        -- noOp
        x"00010002",
        -- add $r1, $r2 => r1 = 3
        x"50010002",
        -- addi $r2, 000A => r2 = C
        x"5102000A",
        -- addm $r3, 0001 => r3 = 4
        x"52030001",
        -- sub $r4, $r1 => r4 = 1
        x"53040001",
        -- subi r5, 5 => r5 = 0
        x"54050005",
        -- subm r2, A => r2 = 2
        x"5502000A",
        -- and r1, r5 => r1 = 0
        x"60010005",
        -- andi r2, 0 => r2 = 0
        x"61020000",
        -- andm r3, 5 => r3 = 4
        x"62030005",
        -- or r6, r10 => r6 = E
        x"6306000A",
        -- ori r2, 2 => r2 = 2
        x"64020002",
        -- orm r1, F => r1 = F
        x"6501000F",
        -- xor r1, r6 => r1 = 1
        x"66010006",
        -- xori r10, FF => r10 = F5
```

```
process(clk, address)
begin
    if (rising_edge(clk)) then
        instruction <= rom(conv_integer(address));
    end if;
end process;
```

Fig 10 – VHDL implementation of the Instruction Memory

5.5 Register File

The register file contains 32 general purpose registers used for computations in the CPU, and their addresses are on 8 bits. It is implemented as following: as inputs, we have the Read Address 1 and Read Address 2, which will output the RD and RS register on the Read Data 1 and Read Data 2. The Write Data input will write data in the register with the address written in the Write Register input (RD register), only when the RegWrite control signal is set to '1'. The writing is done synchronously, while the reading is asynchronous.

```
38  begin
39
40      process(clk)
41      begin
42          if rising_edge(clk) then
43              if RegWr = '1' then
44                  reg_file(conv_integer(wa)) <= wd;
45              end if;
46          end if;
47      end process;
48
49      rd1 <= reg_file(conv_integer(ra1));
50      rd2 <= reg_file(conv_integer(ra2));
51
52  end Behavioral;
```

Fig 10 – VHDL architecture of the Register File component

5.6 Decoder

The Decoder component takes a 4-bit binary number as input, and outputs a 16-bit result, with the specific bit set to 1, and the rest of the bits set to 0.

```
process(data)
begin
    case data is
        when "0000" => output <= "0000000000000001";
        when "0001" => output <= "0000000000000010";
        when "0010" => output <= "0000000000000100";
        when "0011" => output <= "0000000000001000";
        when "0100" => output <= "0000000000010000";
        when "0101" => output <= "0000000000100000";
        when "0110" => output <= "0000000001000000";
        when "0111" => output <= "0000000010000000";
        when "1000" => output <= "0000000100000000";
        when "1001" => output <= "0000001000000000";
        when "1010" => output <= "0000010000000000";
        when "1011" => output <= "0000100000000000";
        when "1100" => output <= "0001000000000000";
        when "1101" => output <= "0010000000000000";
        when "1110" => output <= "0100000000000000";
        when "1111" => output <= "1000000000000000";
        when others => output <= "0000000000000000";
    end case;
end process;
```

Fig 11 – VHDL process of the DCD component

5.7 Extension Unit

The Extension Unit component extends the lower 16 bits of a 32-bit input either to 0, or to its specific sign, depending on the ExtOp selection.

```
 6  entity ExtUnit is
 7      Port ( ExtOp : in STD_LOGIC;
 8             Instruction : in STD_LOGIC_VECTOR (31 downto 0);
 9             Ext_Imm : out STD_LOGIC_VECTOR (31 downto 0));
10  end ExtUnit;
11
12  architecture Behavioral of ExtUnit is
13
14  begin
15
16  process(ExtOp, Instruction)
17         begin
18             case ExtOp is
19                 when '0' => Ext_Imm <= x"0000"&Instruction(15 downto 0);
20                 when '1' => Ext_Imm <= std_logic_vector(resize(signed(Instruction(15 downto 0)), Ext_Imm'length));
21                 when others => Ext_Imm <= x"00000000";
22             end case;
23          end process;
24
25  end Behavioral;
```

Fig 12 – VHDL implementation of the Extension Unit component

5.8 Control Unit

The Control Unit component has as input the opcode, based on which it sets the control signals for some components found in the CPU. The VHDL implementation contains a simple process with a case based on the opcode. A fragment of this case is exemplified bellow:

```
process(opcode)
begin
    case opcode is
        when x"10" =>   MemWrite <= '0';     -- load immediate register
                        MemRead <= '0';
                        R <= '0';
                        RegWrite <= '1';
                        MemtoReg <= '0';
                        ALUSrcA <= '1';
                        ALUSrcB <= "01";
                        ALUCntrl <= "000";
                        FlagEnable <= '0';
                        Branch <= '0';
                        Jump <= '0';

        when x"11" =>   MemWrite <= '0';     -- load register register
                        MemRead <= '0';
                        R <= '1';
                        RegWrite <= '1';
                        MemtoReg <= '0';
                        ALUSrcA <= '1';
                        ALUSrcB <= "00";
                        ALUCntrl <= "000";
                        FlagEnable <= '0';
                        Branch <= '0';
                        Jump <= '0';

        when x"12" =>   MemWrite <= '0';     -- load memory
```

```
        when x"00"  =>  -- noOp (0+0)
                        MemWrite    <=  '0';
                        MemRead     <=  '0';
                        R           <=  '0';
                        RegWrite    <=  '0';
                        MemtoReg    <=  '0';
                        ALUSrcA     <=  '1';
                        ALUSrcB     <=  "10";
                        ALUCntrl    <=  "000";
                        FlagEnable  <=  '0';
                        Branch      <=  '0';
                        Jump        <=  '0';

        when others =>
                        MemWrite    <=  '0';
                        MemRead     <=  '0';
                        R           <=  '0';
                        RegWrite    <=  '-';
                        MemtoReg    <=  '-';
                        ALUSrcA     <=  '-';
                        ALUSrcB     <=  "--";
                        ALUCntrl    <=  "---";
                        FlagEnable  <=  '-';
                        Branch      <=  '0';
                        Jump        <=  '0';
    end case;
end process;
```

```
entity CU is
    Port ( opcode : in STD_LOGIC_VECTOR (7 downto 0);
           MemWrite : out STD_LOGIC;
           MemRead : out STD_LOGIC;
           R : out STD_LOGIC;
           RegWrite : out STD_LOGIC;
           MemtoReg : out STD_LOGIC;
           ALUSrcA : out STD_LOGIC;
           ALUSrcB : out STD_LOGIC_VECTOR (1 downto 0);
           ALUCntrl : out STD_LOGIC_VECTOR (2 downto 0);
           FlagEnable : out STD_LOGIC;
           Branch : out STD_LOGIC;
           Jump : out STD_LOGIC);
end CU;
```

Fig 13 – VHDL implementation of the Control Unit component (entity and fragments of the process)

5.9 Flag Register

The Flag Register is implemented like a simple register with 4 inputs and 4 outputs. It has an enable and an asynchronous reset signal. The VHDL implementation is a simple process with ifs based on the reset, clock and enable signals. The implementation is exemplified below:

```
entity FlagReg is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           FlagEn : in STD_LOGIC;
           Carry : in STD_LOGIC;
           Zero : in STD_LOGIC;
           Sign : in STD_LOGIC;
           Overflow : in STD_LOGIC;
           CF : out STD_LOGIC;
           ZF : out STD_LOGIC;
           SF : out STD_LOGIC;
           OvF : out STD_LOGIC);
end FlagReg;
```

```
process(clk, rst, FlagEn)
begin
    if (rst = '1') then
        CF <= '0';
        ZF <= '0';
        SF <= '0';
        OvF <= '0';
    elsif (rising_edge(clk)) then
        if(FlagEn = '1') then
            CF <= Carry;
            ZF <= Zero;
            SF <= Sign;
            OvF <= Overflow;
        end if;
    end if;
end process;
```

Fig 14 – VHDL implementation of the Flag Register component

5.10 Top-Level

The Top-Level implementation combines all the components described above with port maps, and also implements some new processes for MUX-es and some registers (ACC Register).

6. Testing and Validation

Each component has been tested with its own testbench in order to ensure proper functionality.

6.1 Component Testing

6.1.1 ALU

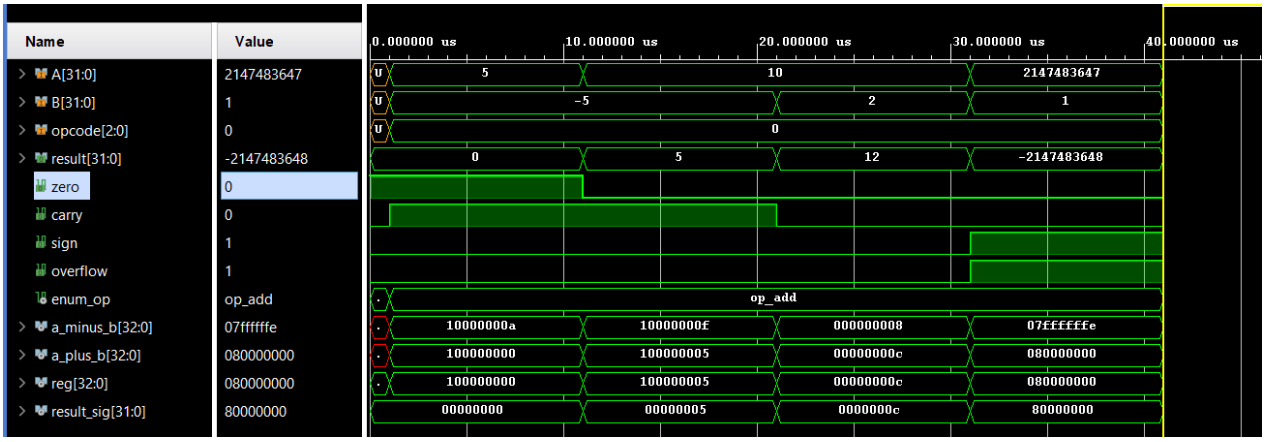


Fig 15 – ALU Addition Waveform

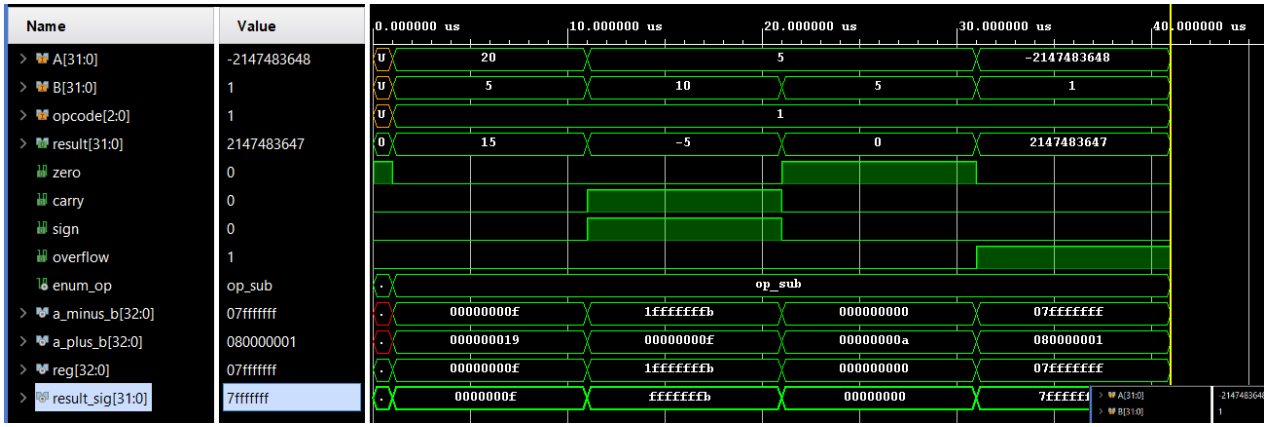
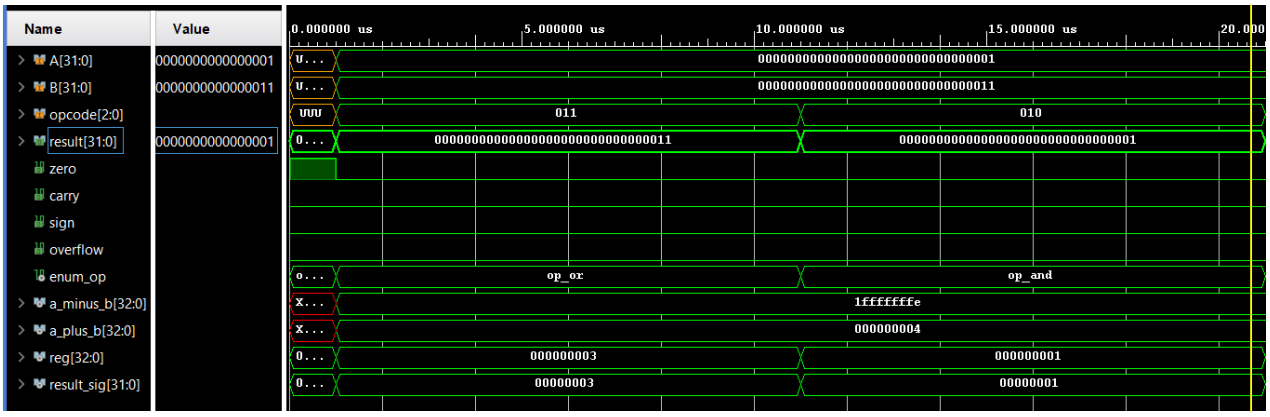


Fig 16 – ALU Subtraction Waveform



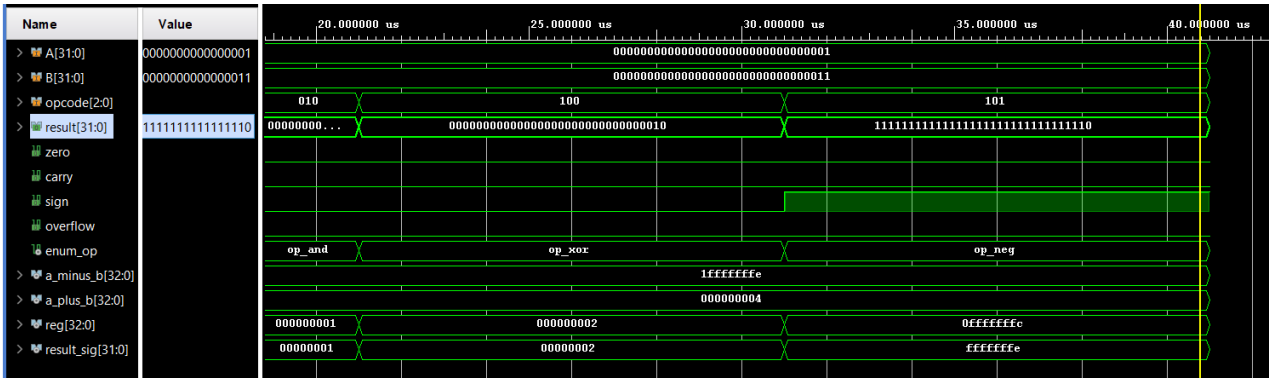


Fig 17 – ALU OR, AND, XOR, NOT Waveform

6.1.2 PC

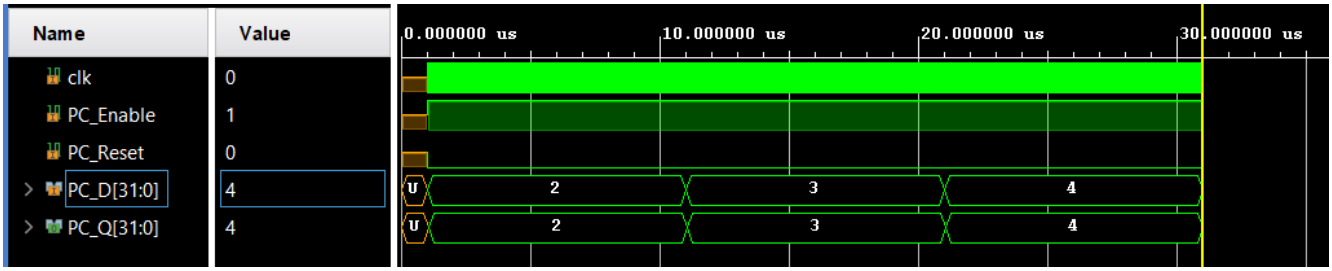


Fig 18 – PC Waveform

6.1.3 Instruction Memory

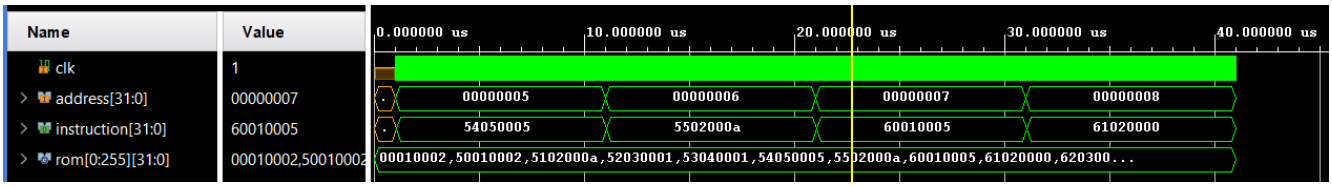


Fig 19 – Instruction Memory Waveform: four reads

6.1.4 Data Memory

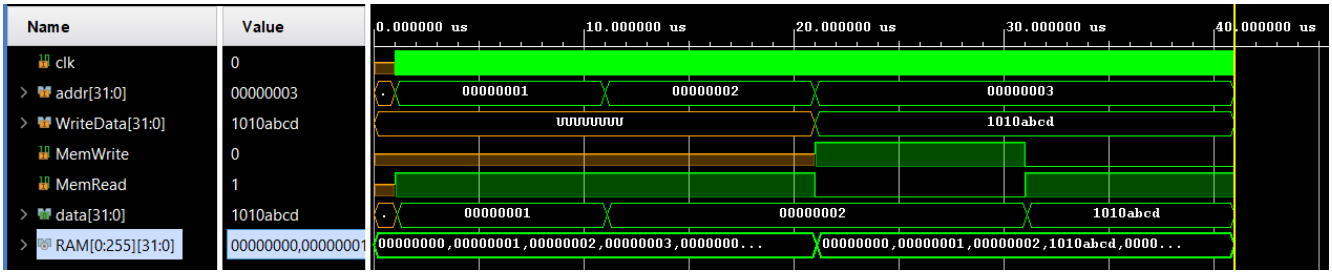
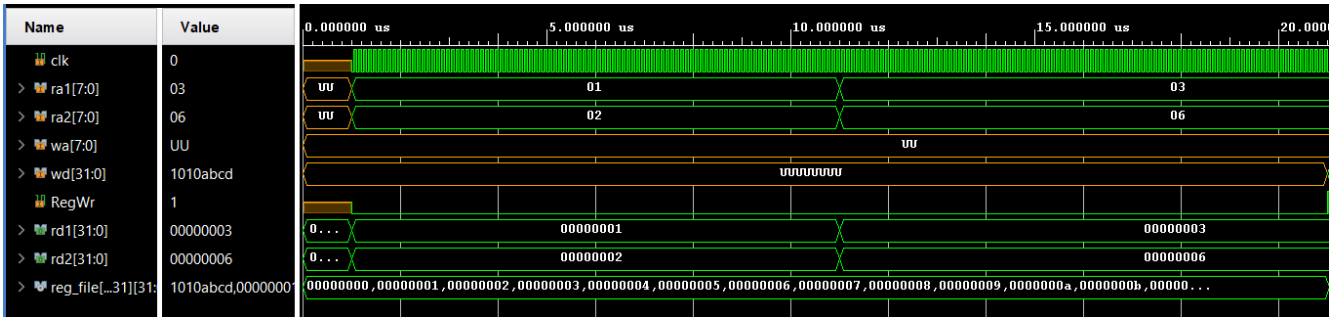


Fig 20 – Data Memory – 3 reads and 1 write

6.1.5 Register File



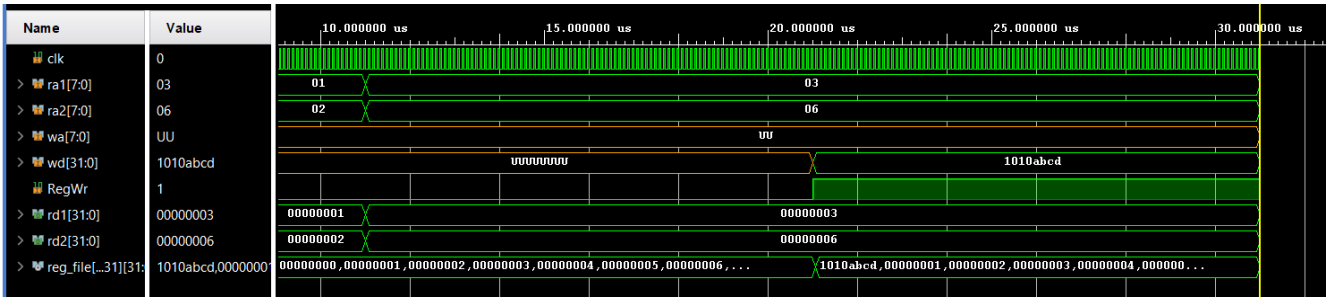


Fig 21 – Register File Waveform: two reads and one write

6.1.6 Decoder

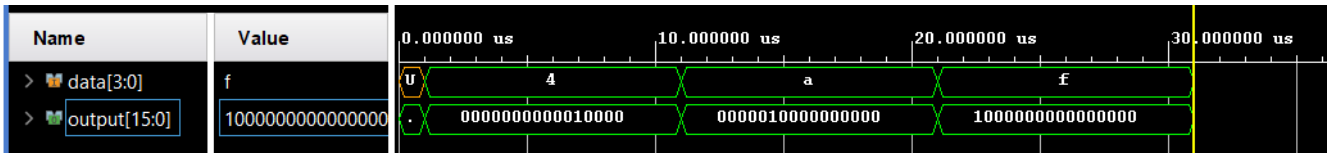


Fig 22 – 4-bit decoder

6.1.7 Extension Unit

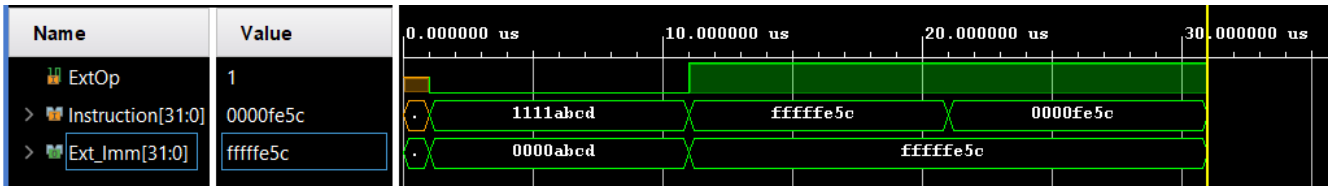


Fig 23 – Extension Unit: Zero Extend and Sign Extend

6.1.8 Control Unit

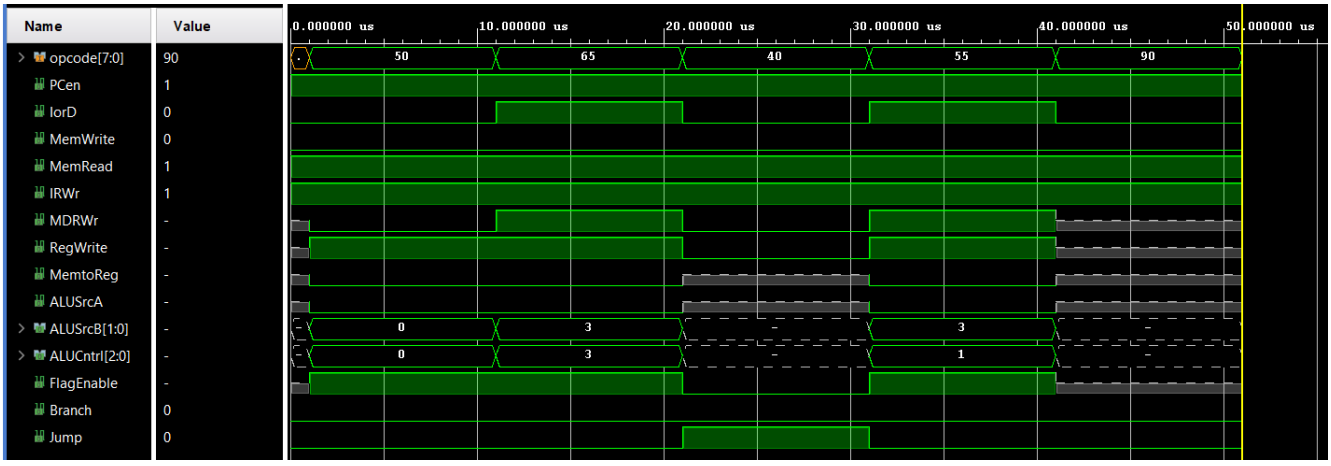


Fig 24 – Control Unit: five opcodes provided (see tables 1 and 2 for control signals)

6.1.9 Flag Register

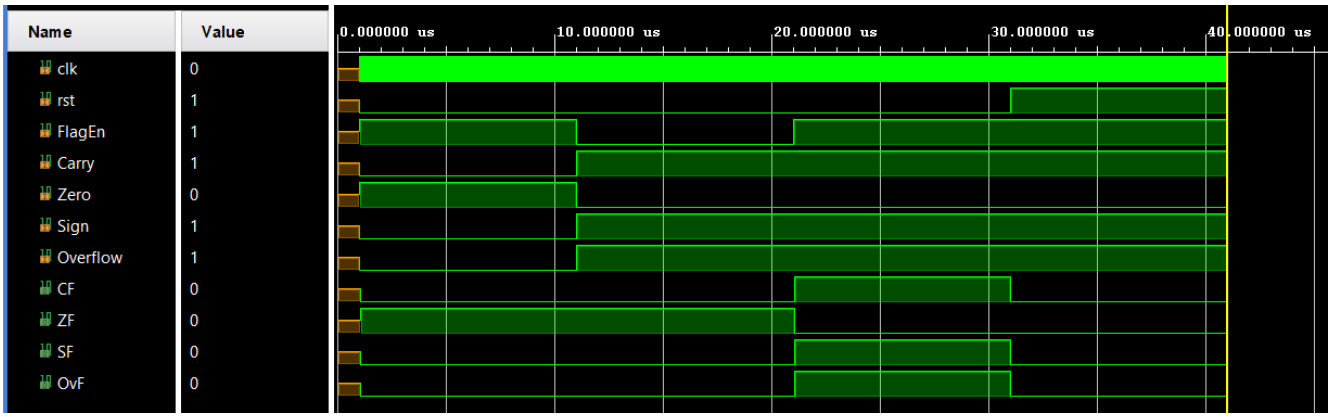


Fig 25 – Flag Register

6.2 Top-Level Testing

The Top-Level testing is done with the help of a testbench. Its implementation is the following:

```
architecture Behavioral of CPU_tb is

component CPU is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC);
end component;

signal clk, rst: std_logic := '0';

begin
l1: CPU port map(
    clk => clk,
    rst => rst);

process
    begin
        rst <= '1';
        wait for 1 us;
        rst <= '0';
        wait;
    end process;

process
    begin
        clk <= '1';
        wait for 10 ns;
        clk <= '0';
        wait for 10 ns;
    end process;
end Behavioral;
```

Fig 26 – CPU testbench

6.2.1 Fibonacci Program

The only thing left for us is to test the instruction set to check if every instruction behaves correctly. To do this, firstly, I composed a simple assembly program that computes the n-th Fibonacci number. The program written in the C language would look like this:

```c
int fib(int n) {
    int a = 0, b = 1, c = 0, i = 2;
    while (i < n) {
        c = a + b;
        a = b;
        b = c;
        i++;
    }

    return c;
}
```

Fig 27 – Fibonacci Program written in C

Now we will translate this program using our own instructions. It will look like this:

```
        lir $R1, A              # n = 10
        lir $R2, 0              # a = 0
        lir $R3, 1              # b = 1
        lir $R4, 0              # c = 0
        lir $R5, 2              # i = 2
        loop:
                cmp $R5, $R1    # compare i with n
                beq endloop     # if equal, the program is over
                add $R2, $R3    # a = a + b
                lrr $R4, $R2    # c = a
                lrr $R2, $R3    # a = b
                lrr $R3, $R4    # b = c
                addi $R5, 1     # i++
                jump loop
        endloop:
        lrr $R7, $R4    # write the result in R7
```

The next table shows the instructions with their pc address and their respective encoding:

PC	Instruction	Encoding
0001	lir $R1, A	x"1001000A",
0002	lir $R2, 0	x"10020000",
0003	lir $R3, 1	x"10030001",
0004	lir $R4, 0	x"10040000",
0005	lir $R5, 2	x"10050002",
0006	cmp $R5, $R1	x"30050001",
0007	beq 0007	x"41000007",
0008	add $R2, $R3	x"50020003",
0009	lrr $R4, $R2	x"11040002",
000A	lrr $R2, $R3	x"11020003",
000B	lrr $R3, $R4	x"11030004",
000C	addi $R5, 1	x"51050001",
000D	jump 0005	x"40000005",
000E	lrr $R7, $R4	x"11070004",

Now, we will store the instructions in the Instruction Memory:

```
type rom_array is array (0 to 255) of STD_LOGIC_VECTOR(31 downto 0);
constant rom : rom_array := (
        -- noOp
--        x"00010002",
        x"1001000A",
        x"10020000",
        x"10030001",
        x"10040000",
        x"10050002",
        x"30050001",
        x"41000007",
        x"50020003",
        x"11040002",
        x"11020003",
        x"11030004",
        x"51050001",
        x"40000005",
        x"11070004",
    others => x"00000000"
    );
```

And, testing this program:

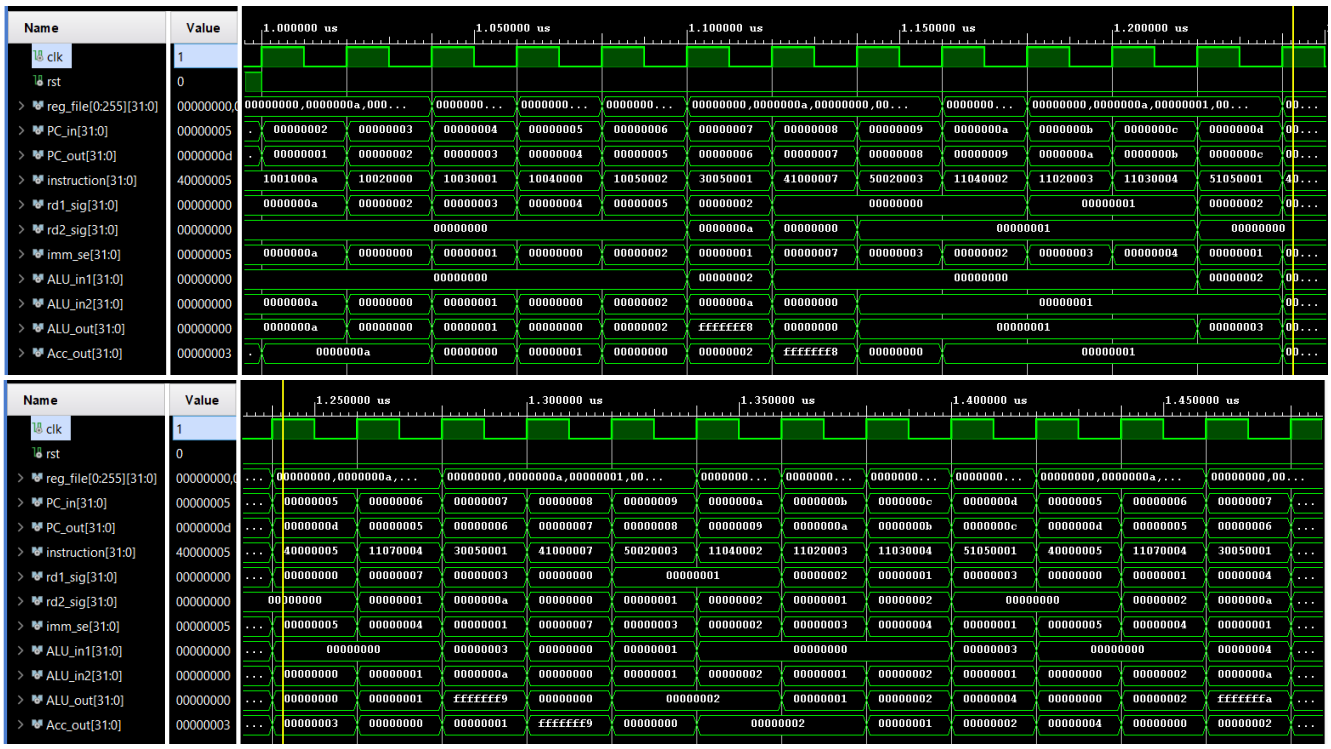


Fig 28 – Waveform of the Fibonacci Program

Verifying each instruction is a tedious task, but we notice that at the end of the execution, register R7 holds the correct value, namely the 10th Fibonacci number in hexadecimal format (22 in hex or 34 in decimal).

Because of this, we can ensure that the **lir, cmp, beq, add, lrr, addi** and **jump** instructions are working accordingly.

6.2.2 Fibonacci Program – Memory Addressing

To test more instruction types, we can modify the main program to use different instructions. For example, instead of using registers to store our data, we could store it in the memory, this way testing the memory addressing of our CPU. The program written in assembly would look like this:

```
lir $R1, A              # n = 10
lir $R2, 0              # a = 0
lir $R3, 1              # b = 1
lir $R4, 0              # c = 0
lir $R5, 2              # i = 2
sm $R1, 0001            # n
sm $R2, 0002            # a
sm $R3, 0003            # b
sm $R4, 0004            # c

loop:
        cmpa $R5, 0001          # compare i with n
        beq endloop             # if equal, the program is over
        lir $R1, 0              # initialize R1
        lm $R1, 0002            # R1 = a
        addm $R1, 0003          # R1 = a + b
        sm $R1, 0004            # c = a + b
        lm $R1, 0003            # R1 = b
        sm $R1, 0002            # a = b
        lm $R1, 0004            # R1 = c
        sm $R1, 0003            # b = c
        addi $R5, 1             # i++
        jump loop
endloop:
lm $R7, 0004            # write the result in R7
```

The table with the PC address and the encodings:

PC	Instruction	Encoding
0001	lir $R1, A	x"1001000A",
0002	lir $R2, 0	x"10020000",
0003	lir $R3, 1	x"10030001",
0004	lir $R4, 0	x"10040000",
0005	lir $R5, 2	x"10050002",
0006	sm $R1, 0001	x"20010001",
0007	sm $R2, 0002	x"20020002",
0008	sm $R3, 0003	x"20030003",
0009	sm $R4, 0004	x"20040004",
000A	cmpa $R5, 0001	x"31050001",
000B	beq 000A	x"4100000A",
000C	lir $R1, 0	x"10010000",
000D	lm $R1, 0002	x"12010002",
000E	addm $R1, 0003	x"52010003",
000F	sm $R1, 0004	x"20010004",
0010	lm $R1, 0003	x"12010003",
0011	sm $R1, 0002	x"20010002",
0012	lm $R1, 0004	x"12010004",
0013	sm $R1, 0003	x"20010003",
0014	addi $R5, 1	x"51050001",
0015	jump 0009	x"40000009",
0016	lm $R7, 0004	x"12070004",

Testing the program, we get the following waveform:

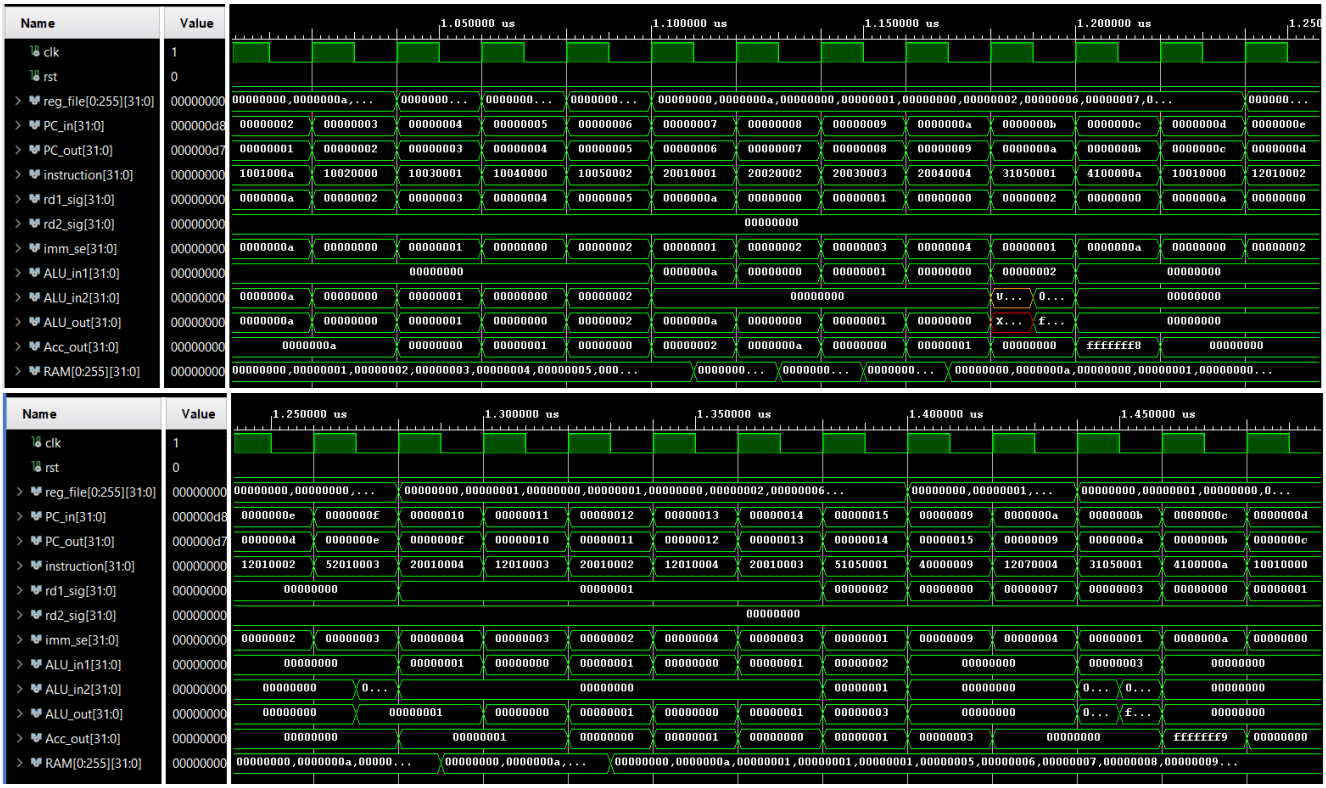


Fig 29 – Fibonacci Program with Memory Addressing – waveform

After several executional loops, we reach the correct result in register $R7, namely 0x22:

This means that all of the instructions used in this program are working correctly, namely **sm, cmpa, lm** and **addm**.

6.2.3 Euclid Program

Another program that we can use to test our instructions would be Euclid's algorithm for finding the greatest common divisor between two positive integers. The implementation in C looks like this:

```
int euclid(int n, int m) {

        while (n != m)
            if (n > m)
                n = n - m;
            else
                m = m - n;
        return n;
}
```

Fig 30 – Euclid's Algorithm implemented in C

We can easily write this program using our own set of instructions. The program would look like this:

```
lir $R1, 000F  # n = 15
lir $R2, 000A  # m = 10
loop:
        cmp $R1, $R2          # compare n and m and branch to the appropriate step in the
                              #execution
        bgt greater_than
        blt less_than
        beq endloop
greater_than:                  # if (n > m)
        sub $R1, $R2           # n = n - m
        jump loop
less_than:                     # if (n < m)
        sub $R2, $R1           # m = m - n
        jump loop
endloop:
        lrr $R3, $R1           # return n (result in $R3)
```

The table with the PC Addresses and the encodings:

PC	Instruction	Encoding
0001	lir $R1, 00C6	x”100100C6”,
0002	lir $R2, 002A	x”1002002A”,
	loop:	
0003	cmp $R1, $R2	x”30010002”,
0004	bgt greater_than	x”42000001”,
0005	blt less_than	x”43000003”,
0006	beq endloop	x”41000005”,
	greater_than:	
0007	sub $R1, $R2	x”53010002”,
0008	jump loop	x”40000003”,
0009	noOp	x”00000000”,
	less_than:	
000A	sub $R2, $R1	x”53020001”,
000B	jump loop	x”40000002”,
000C	noOp	x”00000000”,
	endloop:	
000D	lrr $R3, $R1	x”11030001”,

And the waveform:

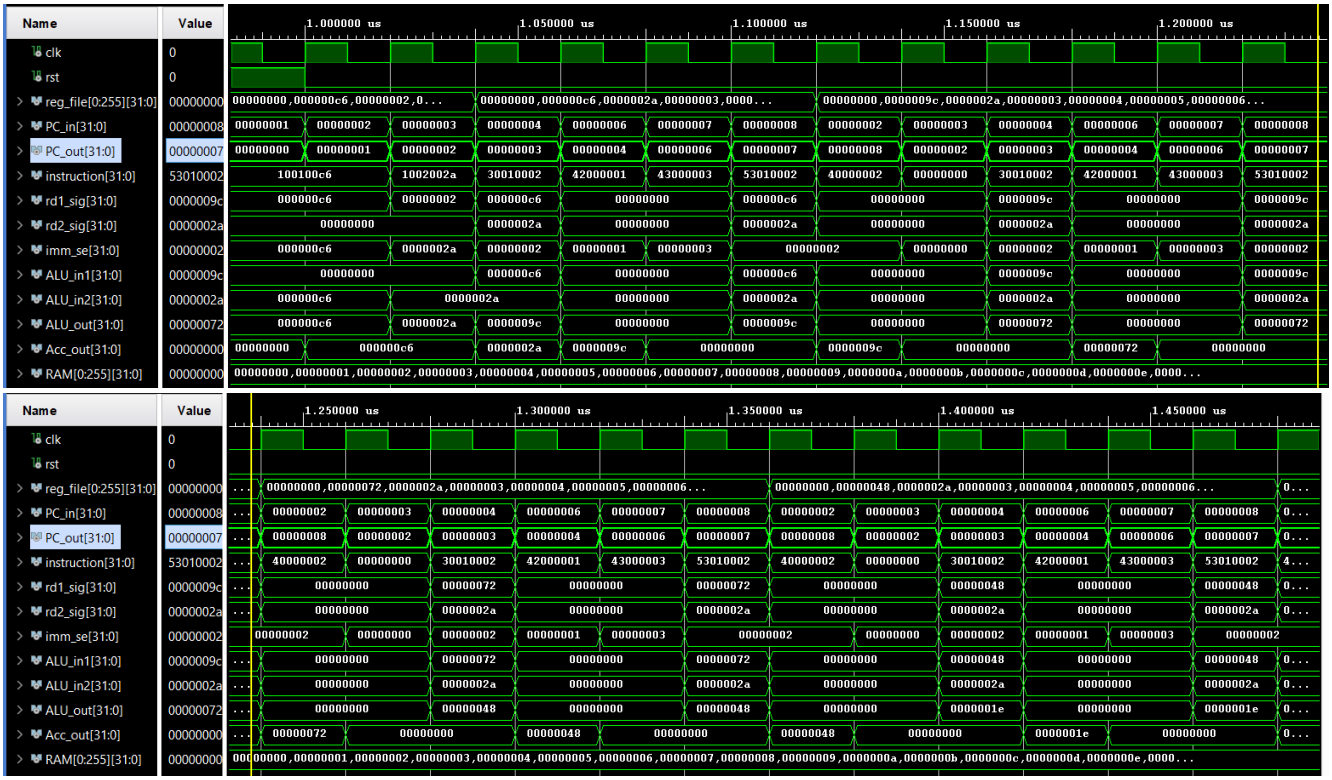


Fig 31 – Euclid Program – waveform

The two numbers, n and m, are stored in R1 and R2 in the first two instructions. In our case, we're trying to find the greatest common divisor between 198 (0xC6) and 42 (0x2A), which is 6. At the end of the program, we notice that R3 contains 6, which is our result.

Therefore, we know that the instructions that we used in the program are working properly, namely **bgt, blt** and **sub**.

The only operations left for testing are some arithmetic and logical operations.

6.2.4 Testing Arithmetic and Logical operations

First, let's test the **subi** and **subm** instructions. We load some values into the registers and the memory, and subtract:

```
        -- Arithmetic and Logical instructions - subi and subm
--        lir $R1, 000A
        x"1001000A",
--        subi $R1, 0005
        x"54010005",
--        lir $R1, 00A5
        x"1001000A",
--        sm $R1, 0001
        x"20010001",
--        lir $R2, 00C2
        x"10020005",
--        subm $R2, 0001
        x"55020001",
```

Fig 32 – Subi and subm operations written in the Instruction Memory

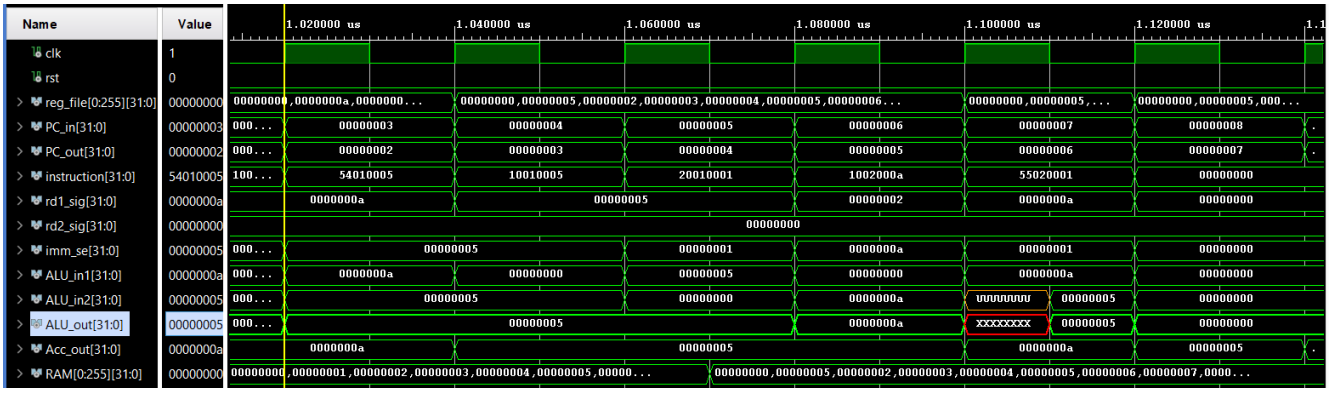


Fig 33 – Subi and subm waveform

We can notice that for both instructions (54010005 and 55020001), the ALU_out result is 5, which is the desired result.

Now we're going to test the logical instructions. To read the results more easily, I'm going to test and, or and xor operations on the numbers B and 5. Here is a table of the results:

Operation	Decimal	Hexadecimal	Octal	Senary	Binary
x	11	B	13	15	1011
y	5	5	5	5	101
x **and** y	1	1	1	1	1
x **or** y	15	F	17	23	1111
x **xor** y	14	E	16	22	1110

The not instruction will be tested on the number 0000000B, and the result will be FFFFFFF4.

Below is a screenshot of the Instruction Memory contents:

```
-- Logical instructions
-- AND
x"1001000B",        -- lir $R1, 000B
x"10020005",        -- lir $R2, 0005
x"20020001",        -- sm $R2, 0001

x"60010002",        -- and $R1, $R2 => R1 = 1
x"1001000B",        -- lir $R1, 000B
x"61010005",        -- andi $R1, 0005 => R1 = 1
x"1001000B",        -- lir $R1, 000B
x"62010001",        -- andm $R1, 0001 => R1 = 1
-- OR
x"1001000B",        -- lir $R1, 000B
x"10020005",        -- lir $R2, 0005
x"20020001",        -- sm $R2, 0001

x"63010002",        -- or $R1, $R2 => R1 = F
x"1001000B",        -- lir $R1, 000B
x"64010005",        -- ori $R1, 0005 => R1 = F
x"1001000B",        -- lir $R1, 000B
x"65010001",        -- orm $R1, 0001 => R1 = F
-- XOR
x"1001000B",        -- lir $R1, 000B
x"10020005",        -- lir $R2, 0005
x"20020001",        -- sm $R2, 0001

x"66010002",        -- xor $R1, $R2 => R1 = E
x"1001000B",        -- lir $R1, 000B
x"67010005",        -- xori $R1, 0005 => R1 = E
x"1001000B",        -- lir $R1, 000B
x"68010001",        -- xorm $R1, 0001 => R1 = E

-- NOT
x"1001000B",        -- lir $R1, 000B
x"69010000",        -- not $R1 => R1 = 4

others => x"00000000"
```

*Fig 34 – **And, andi, andm, or, ori, orm, xor, xori, xorm** and **not** operations written in the Instruction Memory*

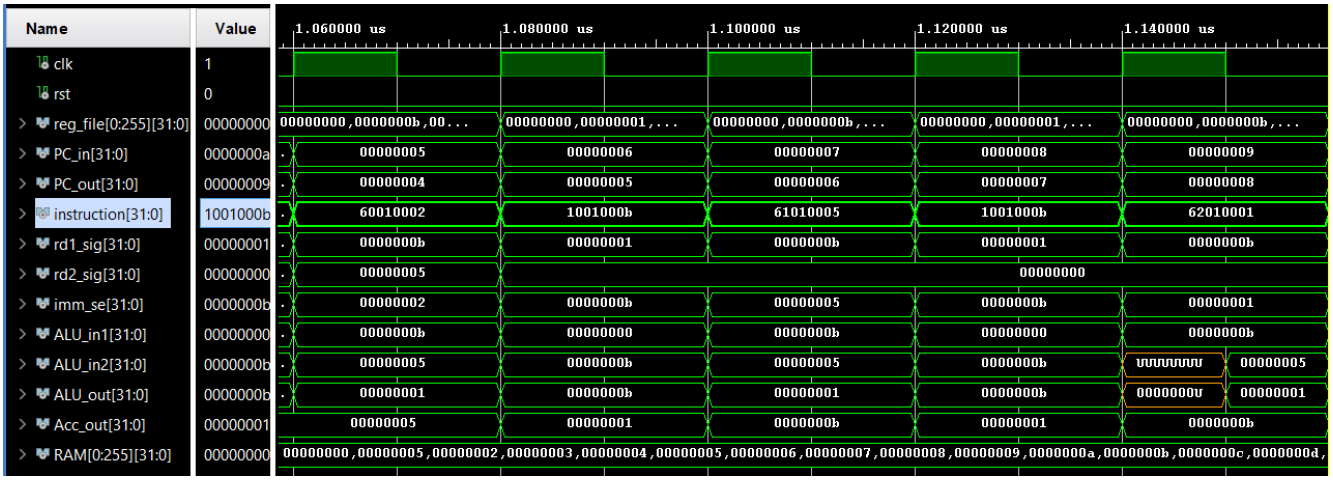


Fig 35 – And instructions (op codes 60, 61 and 62)

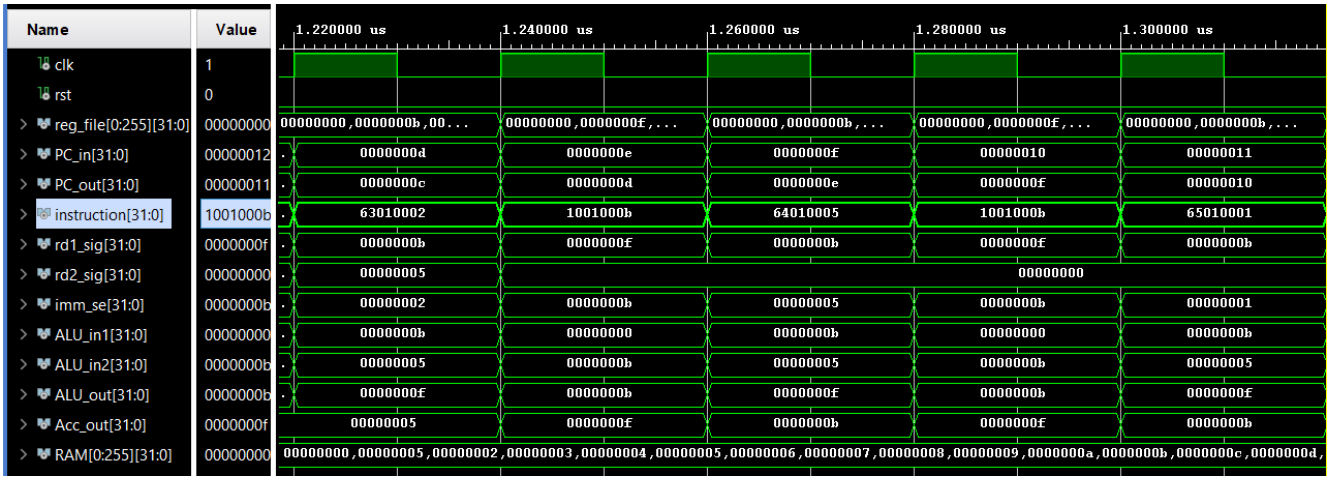


Fig 36 – Or instructions (op codes 63, 64 and 65)

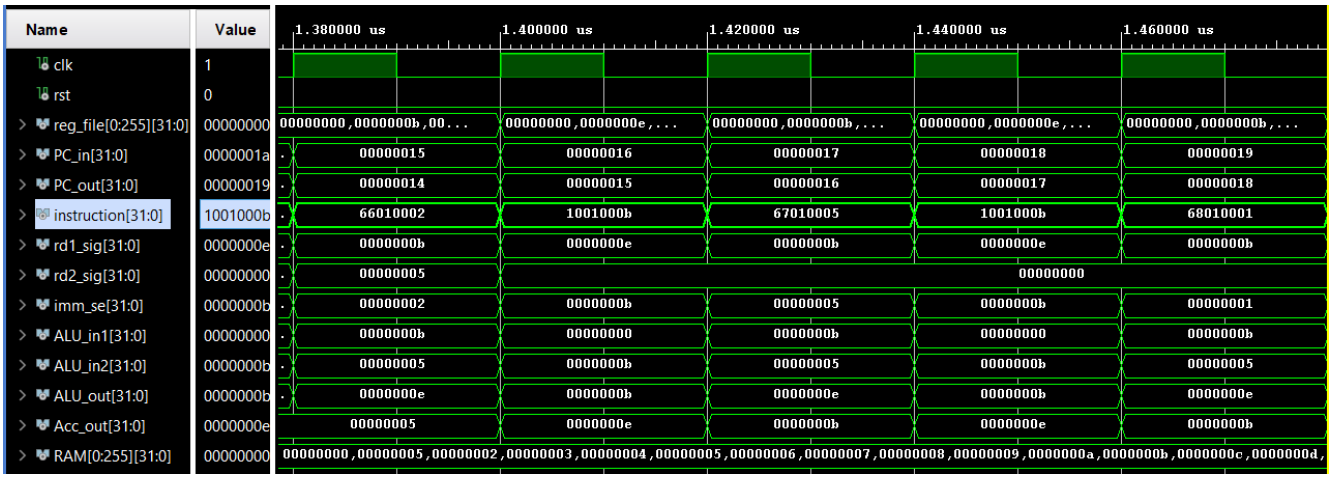


Fig 37 – Xor instructions (op codes 66, 67 and 68)

Fig 38 – Not instruction (op code 69)

After analyzing the waveforms, we notice that all the logical instructions are working properly.

This concludes our testing phase. We can admit now that all the instructions in the instruction set are working correctly.

7 Conclusions

Designing a processor from scratch was not an easy task. I stumbled upon various problems, from the design phase all the way to the testing phase, but I was able to overcome and solve them the best that I could. The biggest challenge for me was to design the processor: I learned the hard way that you can't possibly have a processor which is both single cycle and only has one memory unit, so I had to reconfigure the whole project and add an additional memory (after wasting numerous hours on the initial design and thinking about workarounds with one single memory). I was also worried about the way I encoded my instructions, because it was not the traditional way of encoding, but I managed to work around this in the design phase.

Now that the project is finished, I can finally say that I thoroughly enjoyed working on this project. I developed a passion for hardware design and computer architecture along the way, and I will never take the processor inside my computer for granted ever again 😊.

8 Bibliography

1. "**Processor Design**" [Online], available at https://en.wikipedia.org/wiki/Processor_design

2. Karansingh P. Thakor, Ankushkumar Pal, Prof. Madhura Shirodkar, "**Design of a 16-bit RISC Processor Using VHDL**" [Online], available at https://www.ijert.org/research/design-of-a-16-bit-risc-processor-using-vhdl-IJERTV6IS040284.pdf

3. "**Lecture 4 – Single-cycle CPU Design**", M. Negru [Year 2 Course]

4. MIPS32™ Architecture for Programmers, Volume I: "**Introduction to the MIPS32™ Architecture**" [Online], available at https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol1.pdf

5. "**Von Neumann and Harvard Architectures**" [Online], available at https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Electronics/Implementing_a_One_Address_CPU_in_Logisim_(Kann)/01%3A_Introduction/1.03%3A_Von_Neumann_and_Harvard_Architectures

6. "**von Neumann Architecture"** [Online], available at https://en.wikipedia.org/wiki/Von_Neumann_architecture

7. "**RISC vs CISC**" [Online], available at https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

8. "**What is RISC?**" [Online], available at https://www.arm.com/glossary/risc