FACULTY OF ENGINEERING AND TECHNOLOGY

Department of Electrical and Computer Engineering

First Semester, 2022/2023

Project No. 2 Report

ENCS4370| Computer Architecture

Prepared By:

Laith Swies - 1190513

Supervised by:

Dr. Aziz Qaroush

Dr.Ayman Hroub

Feb 2023

Abstract:

In this project a simple Multi-Cycle RISC processor was designed and tested using HDL language (Verilog).

A RISC processor with Eight 24-bit general-purpose registers: R0 - R7, And a 24-bit program counter register was implemented, supporting 24 bits instruction with three address formats (R-type format, I-type format, and J-type format) and various addressing modes. The five-stages multi-Cycle data path and its control logic were implemented separately before the control and functional units are linked to form the entire data path.

Following that, all details implemented and encountered during the implementation of this data path with its logic, and supported by various test cases.

Contents

Table of Figure:

Table of Tables:

THEORY:

This section describes briefly the datapath, which is a critical component in the construction of any computer, as well as an explanation of what is meant by a Multi-cycle architecture.

Datapath:

A datapath is a collection of functional units responsible for data processing tasks, such as arithmetic logic units or multipliers. t is part of the central processing unit(CPU), along with the control unit. Multiplexers can be used to connect multiple datapaths to form a larger datapath. Because of this, the final datapath for any architecture will be implemented in an incremental approach by considering at each step a set of the instruction and adapting the datapath to support it. Additionally, as we transition from one Architecture to another, such as from a single cycle architecture to a multi-cycle architecture, more components can be added to the datapath. However the main components at any datapath are: the PC, the Instruction memory and the ALU as well as to the register file. These components are enough for the R-type, J-type and the I-type instruction, we further need data memory and sign extenders. Moving to the multi cycle architecture requires more components, mainly the register between stages.

Multi-Cycle Architecture:

A multi-cycle datapath was created by combining single-cycle datapath components. That divides the datapath as well as the instructions into many stages (in our case 5 stages), where each step in the fetch-decode-execute sequence takes one cycle. so different instructions have different execution times. hence another benefit of this architecture is that Each functional unit (like Register File, Data Memory, ALU) can be used more than once during the execution of an instruction, which saves hardware (reduces cost).

As illustrated above, moving to the multi-cycle architecture requires adding more components, like the buffers between stages to store the values of the current instruction from the previous stage and moving them to the next stage,

PROJECT BACKGROUND:

The goal of this project is to implement and test a 24-bit Multi-cycle processor. It was built by first building the basic components, then connecting these various components together, and finally optimizing it.

Our design includes a file register with eight general purpose registers, which each have 24-bits, with the first in the register file (R0) being hardwired. There is also a 16-bit PC register. The instruction set architecture supports three different format types, each of which is 24-bit and obtained as a full word from the instruction memory:

1. **R-type Format:** the instruction here is divided into 7 main parts, 2-bit condition (Cond), 5-bit opcode (Op), 1-bit whether to set the flag bits or not (SF), 3-bit register numbers (Rs, Rt, and Rd), and 7-bit unused (for future use). The Rs and Rt specify the two source register numbers, and Rd specifies the destination register number.
2. **I-type Format:** the instruction is divided into 2-bit condition (Cond), 5-bit opcode (Op), 1-bit set the flag bits (SF), 3-bit register numbers (Rs and Rt), and 10-bit signed immediate value in two's complement representation (2nd operand). Rs is the source register number, and Rt can be either the second source or the destination register number.
3. **J-type Format**: In addition to 2-bit the condition and 5-bit opcode fields, its contains a 17-bit signed immediate constant in two's complement representation.

The following instructions are supported for our datapath.

Table 1: Instruction Encoding

NO.	Instruction	Meaning	Encoding			
R-Type Instructions						
0	**AND**	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 00000	Rs	Rt	Rd
1	**CAS**	Reg(Rd) = Max[Reg(Rs) , Reg(Rt)]	Op = 00001	Rs	Rt	Rd
2	**LWS**	Reg(Rd) = Mem[Reg(Rs) + Reg(Rt)]	Op = 00010	Rs	Rt	Rd
3	**ADD**	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 00011	Rs	Rt	Rd
4	**SUB**	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 00100	Rs	Rt	Rd
5	**CMP**	zero-flag = Reg(Rs) < Reg(Rt)	Op = 00101	Rs	Rt	0000
6	**JR**	PC = Reg(Rs)	Op = 00110	Rs	0000	0000
I-Type Instructions						

7	**ANDI**	Reg(Rt) = Reg(Rs) & Immediate^{10}	Op = 01001	Rs	Rt	Immediate^{10}
8	**ADDI**	Reg(Rt) = Reg(Rs) + Immediate^{10}	Op = 01001	Rs	Rt	Immediate^{10}
9	**LW**	Reg(Rt) = Mem(Reg(Rs) + Imm^{10})	Op = 01001	Rs	Rt	Immediate^{10}
10	**SW**	Mem(Reg(Rs) + Imm^{10}) = Reg(Rt)	Op = 01010	Rs	Rt	Immediate^{10}
11	**BEQ**	Branch if (Reg(Rs) == Reg(Rt))	Op = 01011	Rs	Rt	Immediate^{10}
J-Type Instructions						
12	**J**	PC = PC + Immediate^{17}	Op = 01101	Immediate^{17}		
13	**JAL**	R7 = PC + 1, PC = PC + Immediate^{17}	Op = 01101	Immediate^{17}		
14	**JUI**	R1 = Immediate^{17} << 4	Op = 01110	Immediate^{17}		

All types have the 2-bit condition that enables every instruction to be conditionally executed. we had two conditions, equal and not equal as shown in Table below.

Table 2: Effect of condition value

Condition Field Value	Effect
00	Always execute the current instruction. In assembly, there is no change on the instruction mnemonic. For example, ADD R1, R2, R3 is always executed.
01	Execute if equal, i.e., execute the current instruction if the previous ALU instruction resulted in a zero result, in other words, if the zero-flag bit is set. Otherwise, the current instruction can be treated as a NOP (no operation). This can be reflected in assembly by appending EQ suffix to the instruction mnemonic, such as, ADDEQ, ANDEQ, SUBEQ etc. For example, ADDEQ R1, R2, R3 is executed if and only if the zero flag bit has the value of 1
10	Execute if not equal, i.e., execute the current instruction if the previous ALU instruction resulted in a nonzero result, in other words, if the zero-flag bit is cleared. Otherwise, the current instruction can be treated as a NOP (no operation). This can be reflected in assembly by appending NE suffix to the instruction mnemonic, such as, ADDNE, ANDNE, SUBNE etc. For example, ADDNE R1, R2, R3 is executed if and only if the zero flag bit has the value of 0
11	Unused

A five-stage Multi-cycle supporting all mentioned above instructions was designed and implemented. The control units were added.

DESIGN AND IMPLEMRNTATION:

In order to implement the data path, each functional unit was implemented individually as well as all the needed control units.

- ### Pc

This unit determines the PC value, which is the address of the next instruction to be executed.

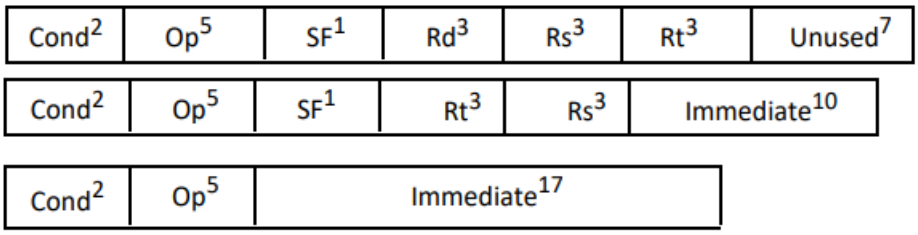
It depends on some of the generated signals like BEQ, J, JR and JAL, the other inputs of this unit are: jump address, branch address, clock and Write bit.

The next, jump and branch addresses are given as follows:

1. Next address = PC +1
2. Jump & Branch address = PC + Immediate^{17}, it is implemented as (JAL+JR+ J + BEQ-ZeroFlag).
3. JR Register address = Reg (Rs), the entered value of the RS register.

- ### Decode Stage

This unit splits the fetched instruction (condition, SF, Rs, Rt, Rd, immediate 10, immediate 17) into different fields based on the given formats, in order to use them in the later stages



- ### Register File

Which consists of set of registers used to stage data between memory and the functional units. In addition to the circuits that are responsible for writing and reading from these registers. It also has eight ports, which are depicted below:

1. BUSA and BUSB: 24-bit output buses for reading two registers.
2. BUSW: 24-bit input bus to write its data into the specified register when the needed conditions are applied as it will be illustrated later.
3. RA: 3-bit input bus that selects the register to be read on BUSA.
4. RB: 3-bit input bus that selects the register to be read on BUSB.

5. RW: 3-bit input bus that selects the register to be loaded into the value in BUSW.
6. RegWrite: 1-bit input that selects if the current action is writing.
7. Clock: The clock input is used only during write operation. During read, register file behaves as a combinational logic block RA or RB valid => BUSA or BUSB valid after access time.

Control Units:

The control unit or CU is a core unit in the implemented datapath, it's responsible for generating the signals that tells the processor components how to respond to a specific instruction.

Main Control Unit:

It takes as input, the opcode and function fields (SF, Cond) of the instruction and then performs its work accordingly

Table 3: Main Control Unit Signal

Signal Name	Effect	
Reg2Sel	0	The Second Source of the Register File come from Rt Field
	1	The Second Source of the Register File come from Rd Field
RegDst	00	The register file destination number for the Write register come from Rt field
	01	The register file destination number for the Write register come from Rd field
	10	The register file destination number for the Write register come from R1 Register
	11	The register file destination number for the Write register come from R7 Register
SeSel	0	The input in I-Type to ALU is the 17 bit Sign-Extended
	1	The input in J-Type to ALU is the 10 bit Sign-Extended
PcSource	00	The output of Adder (Pc +1) is sent to the Pc
	01	The Output of adder (Pc + sign-extended, lower 16 bit) sent to the Pc for Writing
	10	The Output data 1 in Register File sent to the Pc for Writing
MemRead	Content of Memory at the location specified by the address input is put on Memory data Output	
Mem2Reg	00	The value fed to the register file Write data input come from ALUBuffer
	01	The value fed to the register file Write data input come from Memory data Out
	10	The value fed to the register file Write data input come from Pc + 1
	11	The value fed to the register file Write data input come from 17 bit Sign Extend , lower 16 bits shifted left 7 bit
zeroflagWrite	To Enable Write on the Status flag	
AluSrc	0	The Second input of ALU comes from B Register

	1	The Second input of ALU is the Sign-Extended

Write Control Unit:

It contains the Signals Write was includes in a Single control unit (MemWrite, RegWrite, PcWriteCond). It takes the AluOp and zeroflag from the main control unit as input.

Table 4: Write Control Unit Signal

Signal Name	Effect
MemWrite	Memory contents at the location specified by the address input is replaced by value on Write data input
RegWrite	The general-purpose register selected by the write register number is written with the value of the write data input
PcWriteCond	Specify when Pc is been Written

Following is the Control unit truth table indicating the generated signals:

Table 5: Control unit signals

	RegDst	*Req2sel*	*ALUSrc*	*ALUOp*	*SeSel*	*MemRd*	*Mem2Reg*	*MemWr*	*RegWrite*	*PCSur*
R-Type Instructions										
AND	00	0	0	00000	X	0	0	0	1	00
CAS	00	0	0	00001	X	0	0	0	1	00
LWS	00	0	0	00010	X	1	1	0	1	00
ADD	00	0	0	00011	X	0	0	0	1	00
SUB	00	0	0	00100	X	0	0	0	1	00
CMP	X	0	0	00101	X	0	0	0	0	00
JR	X	0	X	00110	X	0	X	0	0	10
I-Type Instructions										
ANDI	00	1	1	00111	1	0	0	0	1	00
ADDI	00	1	1	01000	1	0	0	0	1	00
LW	10	1	1	01001	1	1	01	0	1	00
SW	X	1	1	01010	1	0	X	1	0	00
BEQ	X	1	0	01011	1	0	00	0	0	01 || 00

J-Type Instructions										
J	X	X	X	01100	0	0	X	0	0	01
JAL	11	X	X	01101	0	0	10	0	0	11
LUI	10	X	X	01110	0	0	11	0	1	00

Following are the logic equations for the control all generated signals by the control unit.

1. *RegWrite= ~ (CMP + JR +SW +BEQ +J +JAL)*
2. *ALUSrc = (R-Type + BEQ)*
3. *MemRead = LWS + LW*
4. *MemWrite = SW*
5. *Reg2Sel = I-Type*
6. *SeSel = I-Type*

Following is the implemented Control unit after having the truth table and the expression for each signal ready.

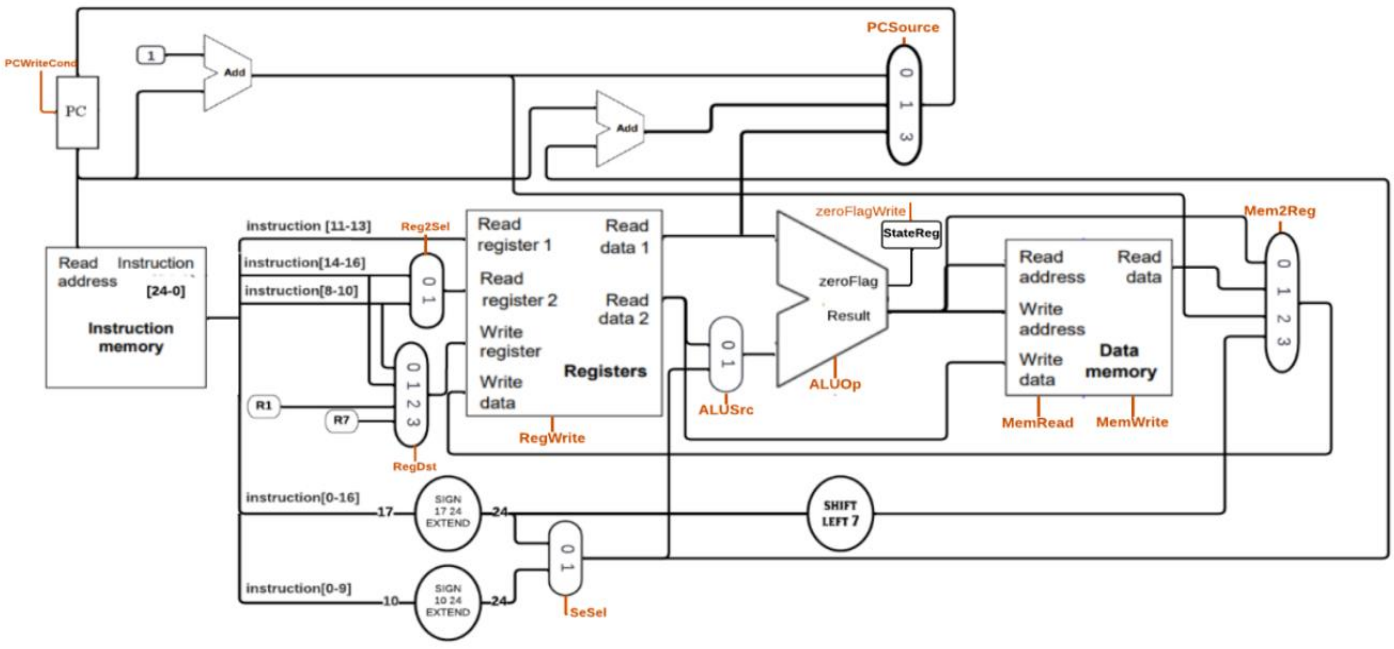


Figure 1: datapath with control signals and extra multiplexer

Finite-state control for multi cycle datapath:

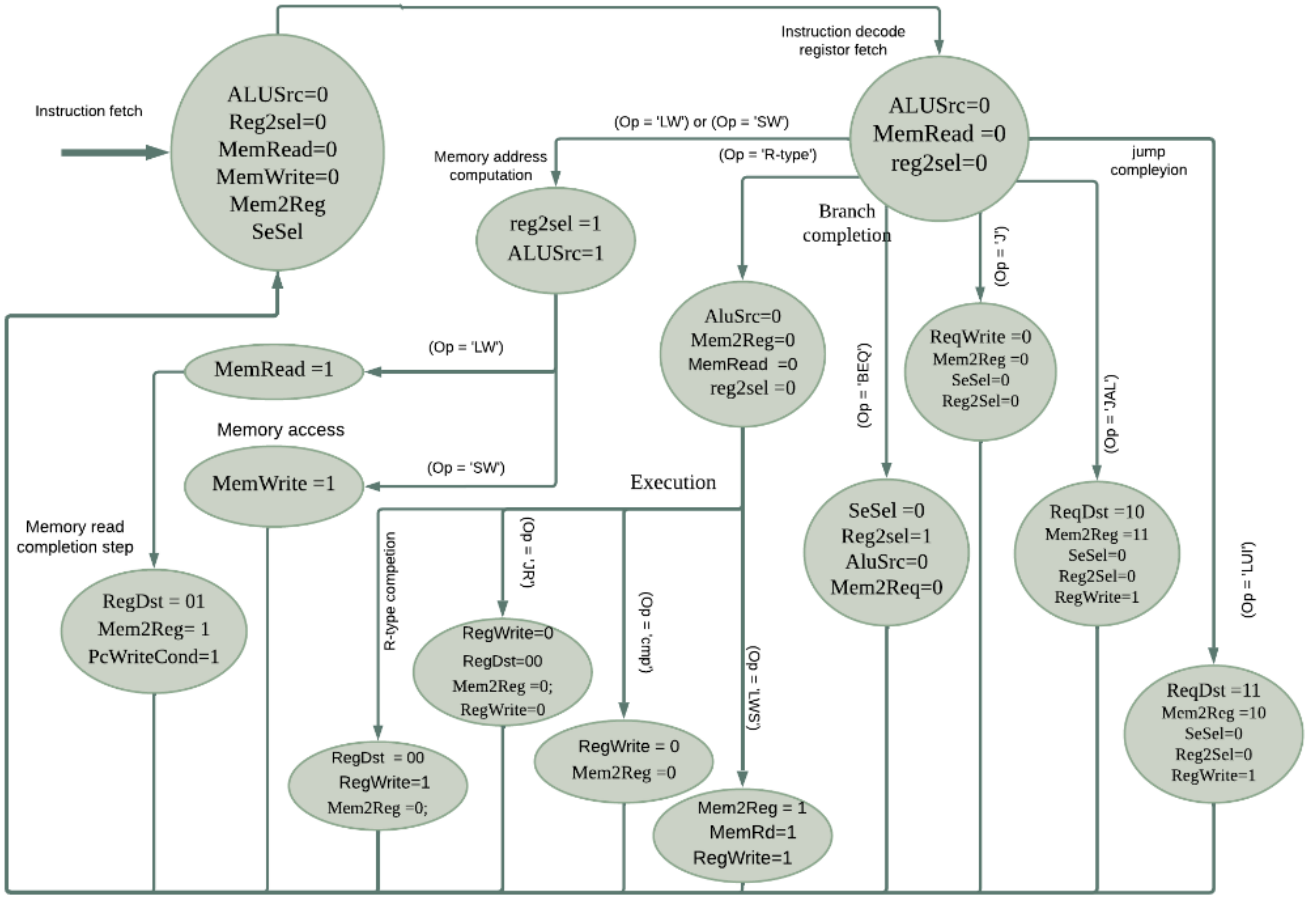


Figure 2 :Finite state control for multi cycle datapath

When computing the performance of the multicycle datapath, we can determine numbers of Cycle for all instruction as following:

Table 6: Multi-Cycle implementation

Instruction	#Cycle
ALU & Store	4
Load	5
Branch	3
Jump	3

Since each state corresponds to a clock cycle we have the following expression for CPI of the multicycle datapath:

CPI = [#Loads.5 + #Stores.4 + #ALU-inst.4 +#Branches.3+ #Jumps.3] / (Total # of Instructions)

- **Buffers Between Stages**

This buffering action stores a value in a temporary register until it is needed or used in a subsequent clock cycle. The buffer was added between each two stages, giving us four buffers: IF/ID, ID/EX, EX/MEM, and MEM/WB. They are all implemented in the same way, with registers for each value to be stored in the buffer, and they are all triggered on the rising edge of the clock.

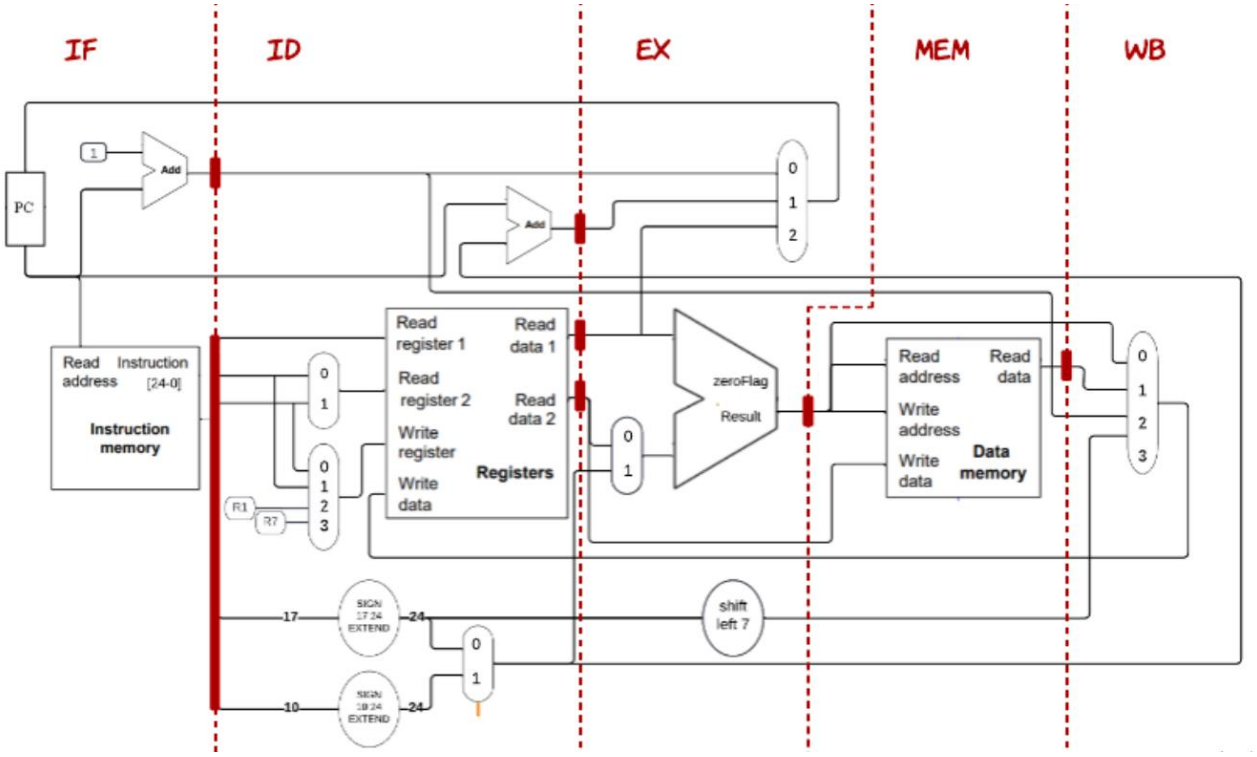


Figure 3: multicycle datapath with buffering registers

As a result of buffering, data produced by memory, register file, or ALU is saved for use in a subsequent cycle. The following temporary registers are important to the multicycle datapath:

- *A & B Registers* (ABuffer, BBuffer) store ALU operand values read from the register file.
- *Memory Data Register* (MemBuffer) save memory output.

- *Instruction Register* (instructionBuffer, PCAddBuffer) saves the data output from the Text Segment of memory for a subsequent instruction read
- *ALU Output Register* (ALUReg) contains the result produced by the ALU.

Instruction Memory:

The Instruction Memory stores all the prefetch instructions. The input for the 24-bit data width memory is the address of the instruction fetched and stored in the instruction register. When the address is ready, the reading process is performed asynchronously.

Data Memory:

The data memory consists of a series of fixed size blocks (24-bit data width).

ALU Unit:

Referring to the given table of instructions, 7 different ALU operations contains all the arithmetic components needed in our instruction set and needed in order to execute all instruction. Which are: an adder, subtractor, comparator, AND gate and OR gate, Also, we need a mux, which is used for the CAS instruction, to output the maximum number between the inputs. Finally, all those components were connected to a mux having the ALU operation control signal on its selection line.

Table 7 : ALU operation decode

Opcode		Condition	SF	Opcode		Condition	SF
00000	**AND**	00	0	00101	**CMP**	X	0
	ANDEQ	01	X	01000	**ADDI**	00	0
	ANDNE	10	X		**ADDIEQ**	01	X
00001	**CAS**	00	0		**ADDINE**	10	X
	CASEQ	01	X	00100	**SUBISF**	X	1
	CASVE	10	X	01001	**LW**	00	0
00010	**LWS**	00	0		**LWEQ**	01	X
	LWSEQ	01	X		**LWNE**	10	X
	LWSNE	10	X	01010	**SW**	00	0
	ADD	00	0		**SWEQ**	01	X
	ADDEQ	01	X		**SWNE**	10	X

00011	**ADDNE**	10	X	01011	**BEQ**	X	0
00100	**SUB**	00	0	01100	**J**	00	0
	SUBEQ	01	X		**JEQ**	01	X
	SUBNE	10	X		**JNEQ**	10	X
00110	**JR**	00	0	01101	**JAL**	00	0
	JREQ	01	X		**JALEQ**	01	X
	JRNE	10	X		**JALNE**	10	X
00111	**ANDI**	00	0	01110	**LUI**	00	0
	ANDIEQ	01	X		**LUIEQ**	01	X
	ANDINE	10	X		**LUINE**	10	X
				01111	**NOP**	00	0

- **10-bit extender:**

Because all I-Type instructions require a sign extension, this block was implemented to convert the 10-bit number into a 24-bit number using the sign extension.

- **17-bit extender:**

This block was implemented to extend the 10-bit number, obtained from the J-Type instruction, into a 24-bit number by using the sign extension since all the J-Type instructions require a sign extension.

Final Datapath:

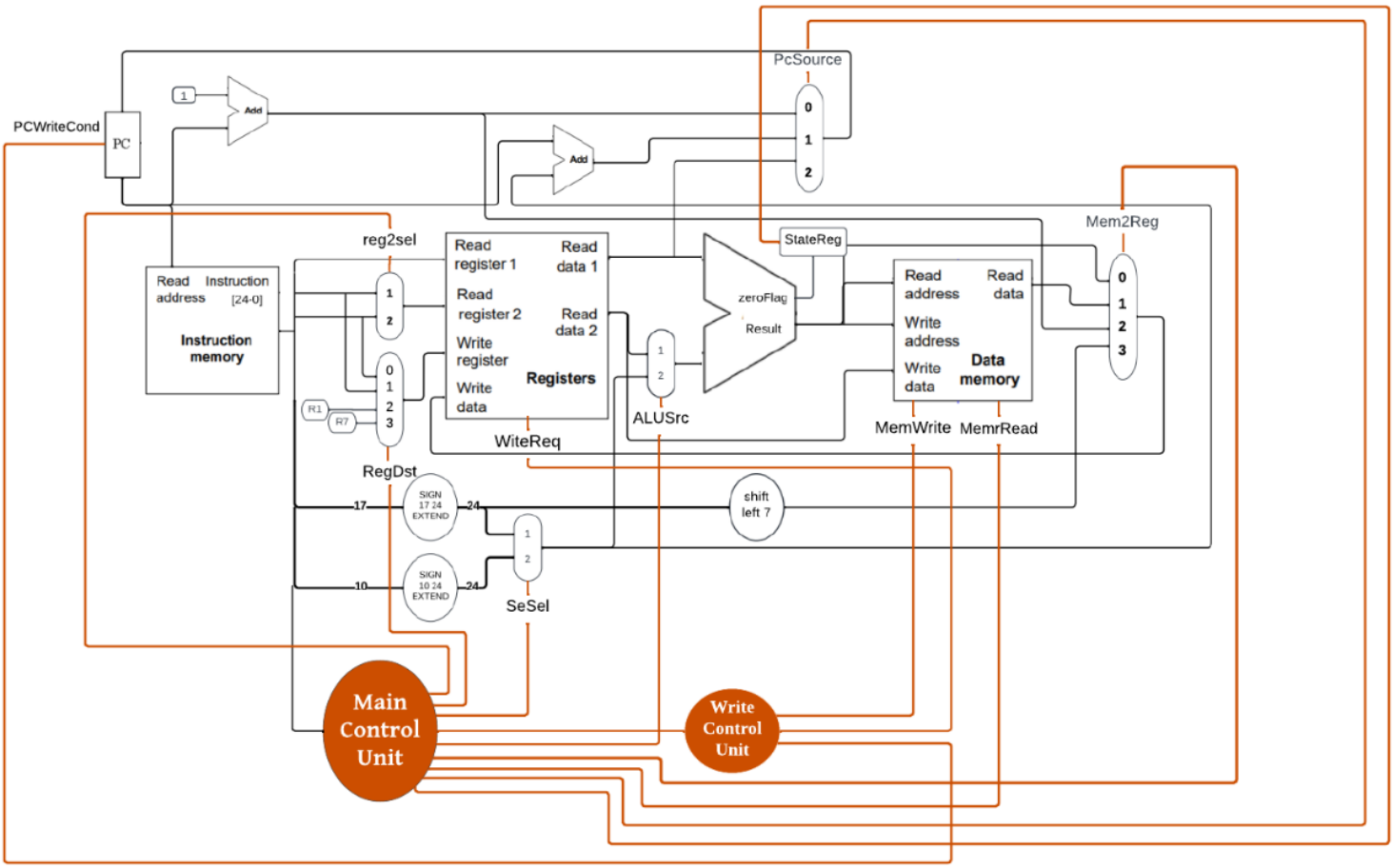


Figure 4: Multi Cycle datapath With Control Units

SIMULATION AND TESTING:

TEST1:

```
LUI R1 = decimal (600) << 7
ADDI R4 = R0 + 6                    ; store the array length in R4
ADDI R2 = R0 + 0                    ; i=0 in R2
LOOP1:
ADD R3 = R1 + R2                    ; initialize the element address in R3
Sw Mem(R3+0) = Reg(R2)              ;store Mem(address) = i
ADDI R2 = R2 + 1                    ; i++
BEQ if (R2 == R4)                   ; Branch to the next loop if finished
ADDI R7 = R0 + 4                ; save the starting address of the loop in R7 if not finished
JR PC = Reg(R7)                 ; jump to the first instruction if not  finished
LOOP2:
ADDI R2 = R0 + 0
LWS R5 = Mem(Reg(R2)+Reg(R1))
ADDI R2 = R2 + 1
BEQ if (R2 == R4)
ADDI R7 = R0 + 11
JR PC = Reg(R7)
```

Description:

This program initializes an array Pointer in the data memory, which its first element address is 600 << 7 , and it stores the I^{th} element (0-5) into its corresponding address .And the second loop loads back the stored array element to R5. The full Execution of the Program is shown in the Wave form below:

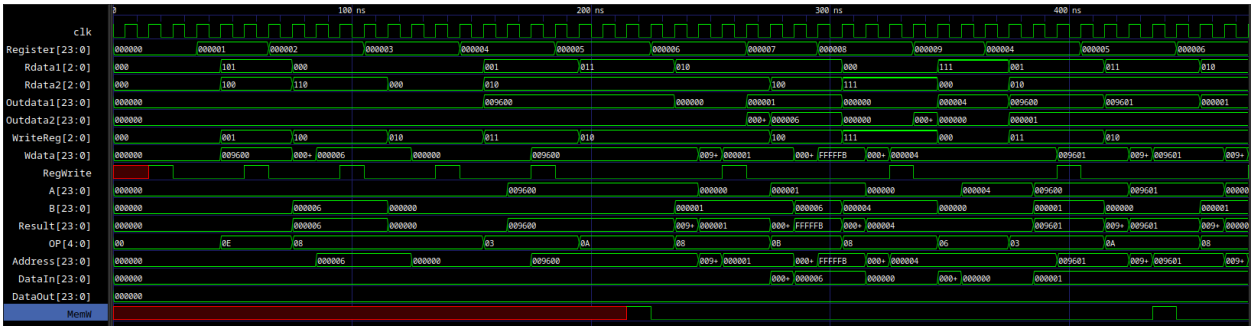


Figure 5: First loop Exection 1

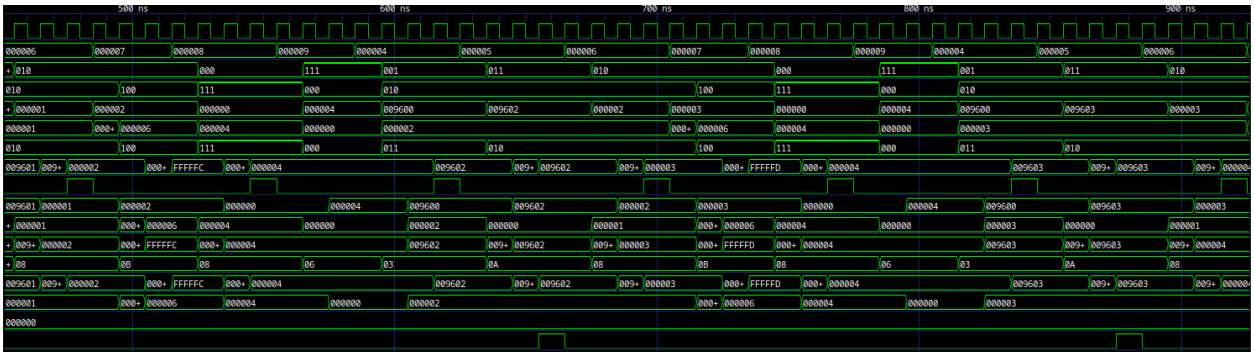


Figure 6: First loop Exection 2

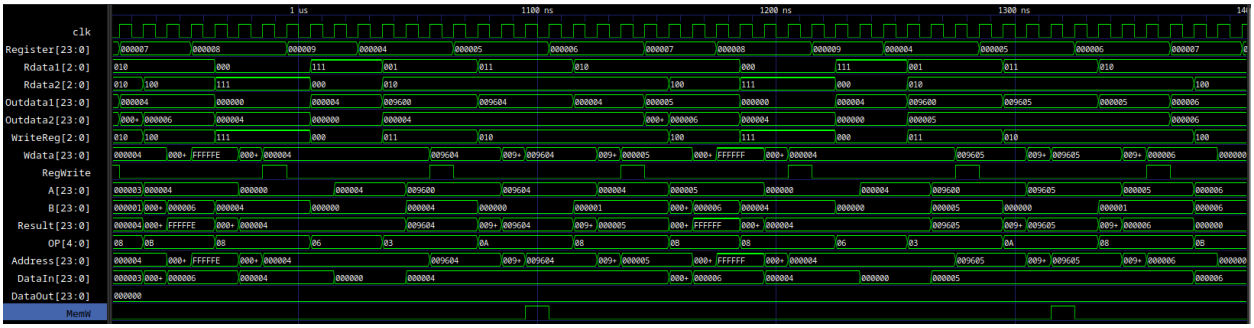


Figure 7:L First loop Execution 3

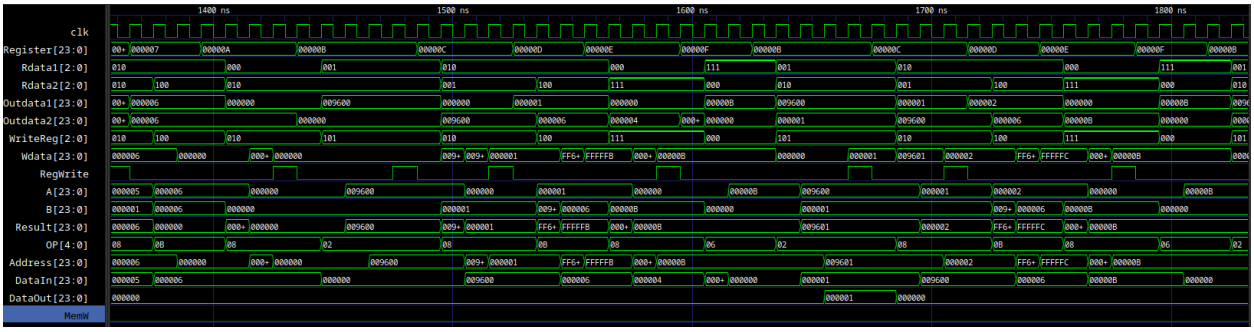


Figure 8: First loop Exaction 4

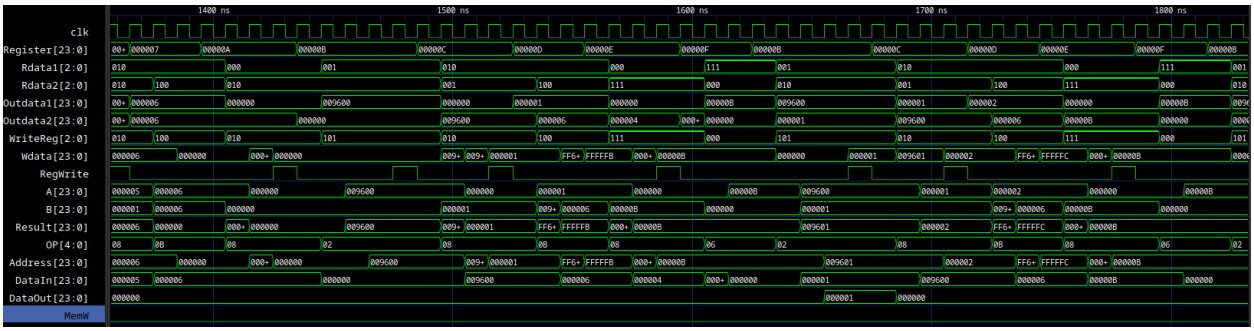


Figure 9: First loop Exaction 5

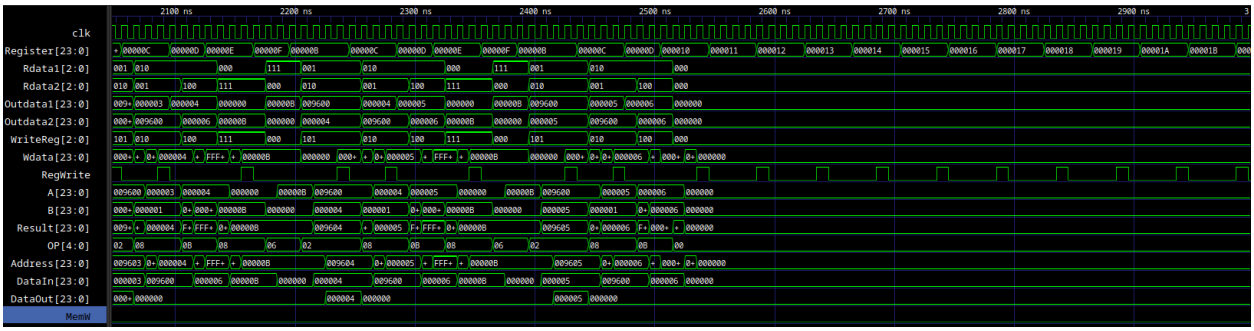


Figure 10: First loop Exaction 5

Test2:

```
SUBIF    R2,R1,R0
ADDNE  R2,R1,R0
```

Since all of the Register in the Register file initialized Zero at the beginning of the program, the following figure, in which the first instruction (SUBIF R2, R1, R0) in the third stage (Execution) and we notice that the ALU works properly as the result of the SUBSF instruction will lead to a value of 1 in the zero flag. The second instruction (ADDNE R2,R1, R0) and we can notice that it's not executed from the Op field in the following figure which is the ALUOp and its equal to 0F and that's the NOP Opcode in our design.

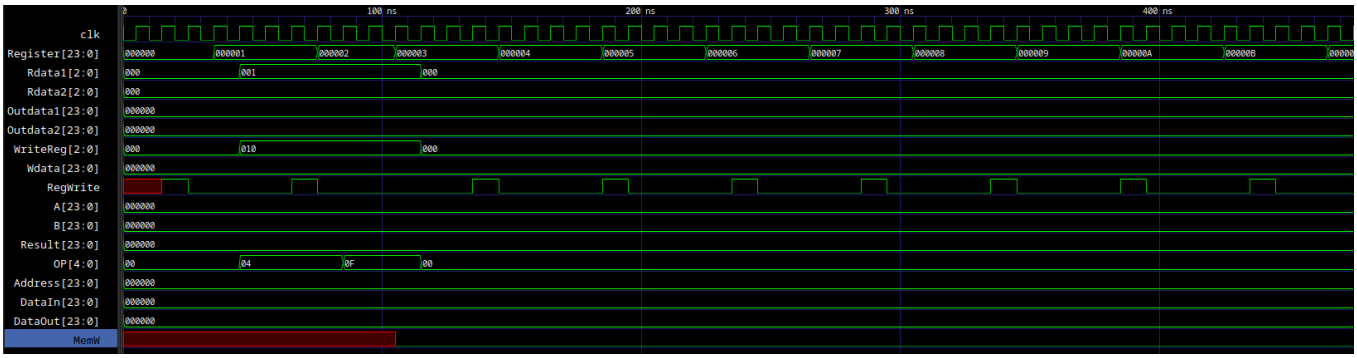


Figure 11: Testing the Conditional Executions

CONCULUSION AND FUTURE WORK:

In this project we have learned a lot about the Multi-cycle structure and the implementation of it. We have succeed in implementing a Multi cycle 24-bit processor using Verilog language with all the aims that were required in this project.

As a future plan, this project encourages us to learn more about computer architecture techniques and how to optimize the possible designs that exist nowadays and Maybe Create an assembler for this ISA using C language.