بسم الله الرحمن الرحيم

Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Architecture (ENCS4370)

Project Report

Multi-Cycle RISC Processor

Prepared by : **ID :**

Momen Salem **1200034**

Abdalkarim Eiss **1200015**

Mahmoud Khatib **1200275**

Instructor : Dr. Aziz Qaroush

Section : 2 **Date : 28-1-24**

Place : Ramallah

Abstract

The aim of this project is to build and test simple multi-cycle RISC processor with 16 32-bit general purpose registers, two separated memory cells, one for instruction and the other for data and stack. Two 32-bit special purpose registers PC (to point at instruction) and SP (to point at top of the stack memory). The processor support 4 different instruction types R, I, J and S -types. The processor was built and tested using Verilog HDL in behavioral manner.

Table of contents

List of Figures

List of Tables

1-Design and Implementation

1. Data Path

The data path design is for a multi-cycle processor which have five stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (EX), Memory Access (MA) and Write Back (WB). For this processor, all components are controlled by a common synchronized clock and triggered by the positive edge. It was assembled using the following components:

Program Counter (PC)

The program counter (PC) was used to store the address of the next instruction that would be fetched. Figure 1 shows a 32-bit PC register, a mux 4x1 that was used to choose the PC input according to the 2-bits PC source signal that generated by PC control, a plus 1 component to make the normal update on the PC and the wires between the components. The PC input have four choices:

- The normal PC (PC+1).
- The branch target address (BTA) which is (PC+1+immediate$_{16}$).
- The jump target address (JTA) which is (PC [31:26] || immediate$_{26}$).
- A PC for the RET instruction which is (PC = top of the stack).

These choices were made to support for different instruction types.

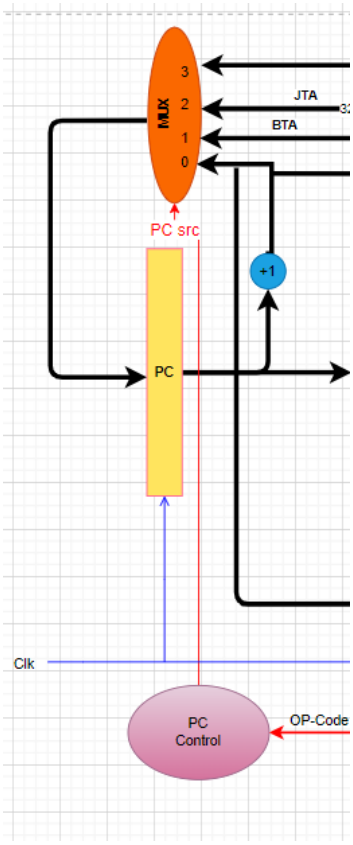

Figure 1: Program Counter Component

Also, the output of PC component enters to the Instructions Memory component.

Instructions Memory

The instructions memory component is one of a physical memories component that was used. This memory was used to store the instructions. The data path supports four instruction types (R-type, I-type, J-type and S-type), each one has a size of 32-bit (word). Figure 2 shows the instruction memory component with one input (32-bit address) and one output (32-bit instruction), also three buffers (two PC buffers and instruction buffer) to store the results of present stage (IF) to be used on the next stage (ID). These buffers were controlled by a common synchronous clock. At the end of this component, the instruction will be fetched.

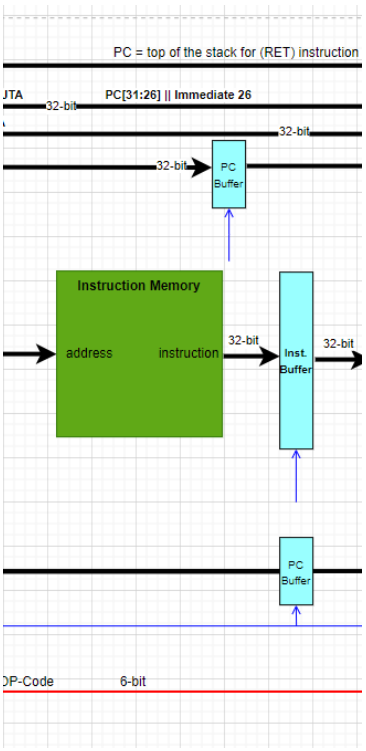

Figure 2: Instruction Memory

```
// the actual Instruction Memory
reg [31:0] Memory[0:1023];
```

Figure 3: Instruction Memory Code-Part One

```
// -------------------- InstructionMemory --------------------------
task InstructionMemory(input [31:0] instAdd, output reg [31:0] instOut);

    instOut = Memory[instAdd];

    $display("instruction = %b\n", instOut);
endtask
```

Figure 4: Instruction Memory Code-Part Two

Registers File

The registers file has 16-general purpose registers (R_0 - R_{15}), each one has a 32-bit size. After the instruction decoded, it was used to read the operands and to write the result on a specific destination register. Figure 5 shows the registers file with eight inputs as following:

- Rs1: The first source which takes the address of the first operand.
- Rs2: The second source which takes the address of the second operand.
- Rd: The destination which takes the address of the destination register.
- BusW: The write data bus which takes the result in the WB stage and write it on the destination register when the RegW is one.
- RegW: A signal to enable the write process on the register file. When it is one, the write process allowed. It was generated by the main control.
- Rs1W: A signal to enable the update on the Rs1 when the instruction is LW.POI. It is determined from the first 2-bits in the instruction.
- BusWA: A write data bus which takes the data from the BusA.
- Clk: To make the registers file controlled by a common synchronous clock.

Also, it has two outputs BusA to. Additionally, a mux 2x1 was used to choose between the Rd and Rs2 to be the second source according to the Reg2Src signal. Also, an extender to extend the 16-bit immediate to be 32-bit according to the ExtOp signal. This signal determines the type of extension process (signed or unsigned). In addition, an adder to compute the branch target address (BTA). As noted on the figure, the buffers were used between the stages to store the results of present stage (ID) to be used on the next stage (ID) and on the pervious stage (IF).

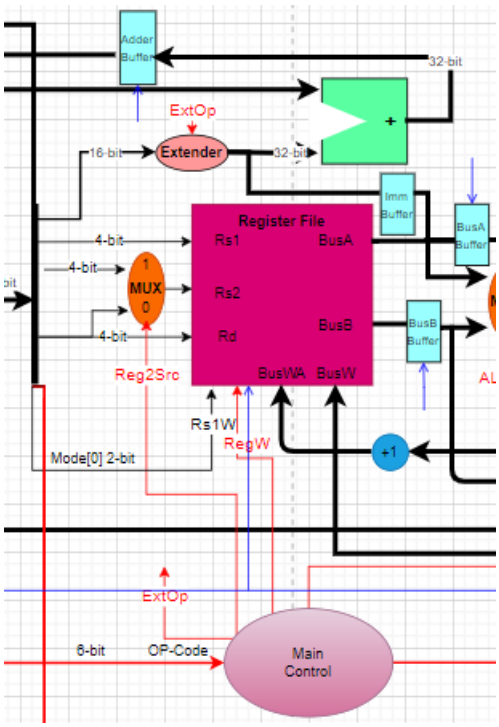

Figure 5: Registers File and the Components of the ID Stage

```
// ------------------- Register Read -------------------------
task RegsiterFileRead(input [3:0] Rs1, Rs2, output signed [31:0] BusA, BusB);
    BusA = Registers[Rs1];
    BusB = Registers[Rs2];
endtask

// ------------------- Register Write -------------------------
task RegsiterFileWrite(input [3:0] Rd,  input RegW, input Rs1W, input [31:0] BusW, input [31:0] BusWA);
    if(Rs1W)
        Registers[Rs1] = BusWA;
    if(RegW)
        Registers[Rd] = BusW;
    //if(Rs1W || RegW)
//      $display("Register[Rs1]: %2d, Written value: %2d\n", Registers[Rs1], Registers[Rd]);
endtask
```

Figure 6: Registers File Code

Sign Extender

The Extender role is to extend a 16-bit immediate number, into a 32-bit number. It has two inputs, one 16-bit input, which is the number that needs to be extended to 32-bit, and a signal input, that specifies if this is signed or unsigned extending. It has only one 32-bit output, which is the extended number.

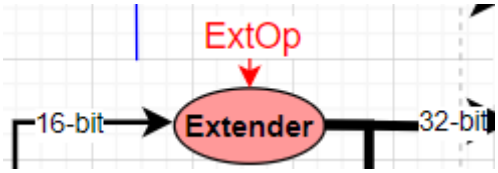

Figure 7: Sign_Extender

```
///----------------------EXTENDER--------------- This task to extend the 1
task Extender(input [15:0]imm, input ext_op,output signed [31:0]outExt);
    if(ext_op)
        begin
            if(imm[15]==1'b1)
                outExt={16'hFFFF,imm};
            else
                outExt={16'b0,imm};
        end
    else
        outExt={16'b0,imm};

endtask
```

Figure 8: Sign_Extender implimentation

Arithmetic & Logical Unit (ALU)

The ALU implanted in this processor can handle three main operations which are (add, bitwise and, subtract) these three operations serve a wide range of instructions.

For the inputs for this ALU, it has two 32-bits inputs, which are the operands, and another 2-bit input, which is the operation that needs to b performed by the ALU (add, and, sub), that was generated by the control unit. The first operand of the always comes from (Bus A) from the register file, while the seconds operand can come from:

- Bus B: if the instruction is R-type, where the second operand is in a register.

- Extended immediate: if the instruction is I-type, where the second operand is the 16-bit immediate from instruction register extended to 32-bit (with the sign taken into consideration).

The choice is made based on a signal input called (ALUsrc), that feeds into the selection line of a 2x1-MUX

Outputs of the ALU are, one 32-bit output (result of the operation), and three 1-bit flags, which are the zero, negative, and the overflow flags. These three flags are required to handle all four branch instructions (BEQ, BNE, BGT, BLT).

BEQ, and BNE uses the zero flag to determine if the condition is true or false, and for BGT, and BLT, negative and overflow.

Note: in the instruction's description of BGT, the branch is taken

if (Reg(Rd) > Reg(Rs1)), but first operand in ALU is Reg[Rs1], so to fix this issue, BGT checks conditions if the first operand is less than the seconds operand, and the same for BLT, the condition checked is if first operand is greater than the second one.

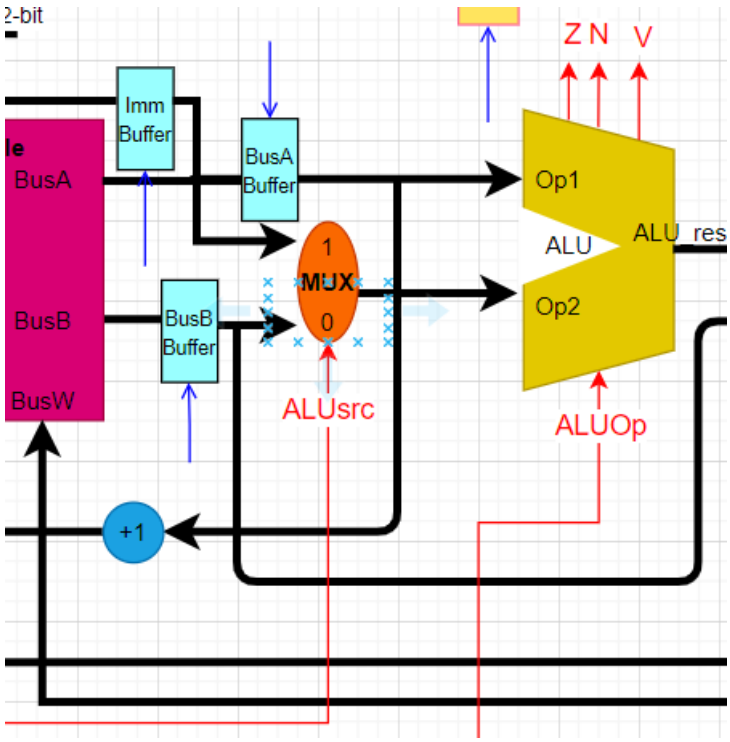

Figure 9: ALU Diagram in Execution Stage

```
// -------------------- ALU -------------------------
task ALU (input [1:0] ALUOp, input [31:0] op1, input [31:0] op2, output reg signed [31:0]result, output reg z, n, v);

    // compute result
    case(ALUOp)
        `AND: begin result = op1 & op2; end
        `ADD: begin result = op1 + op2; end
        `SUB: begin result = op1 - op2; end
    endcase

    // set flags
    z = result == 32'd0;
    v = (op1[31] == op2[31]) && (result[31] != op1[31]);
    n = result < 0;

    $display("op1= %2d, op2=%2d\n", op1, op2);
    $display("ALU result= %2d, z=%b, n=%b, v=%b\n", result, z, n, v);
endtask
```

Figure 10: ALU Implementation

Adder for BTA

Since the implementation of this processor does not support the instruction to use the same component in the data path more than once, an additional adder was added to the data path to calculate the branch target adders, while the ALU is handling the condition checking of the branch instruction.

Inputs and outputs for adder are:

- 32-bit input with the value of the PC.
- 32-bit input immediate value after being sign extended.
- 32-bit output of the branch target address that feeds in the 4x1 MUX of PC source.

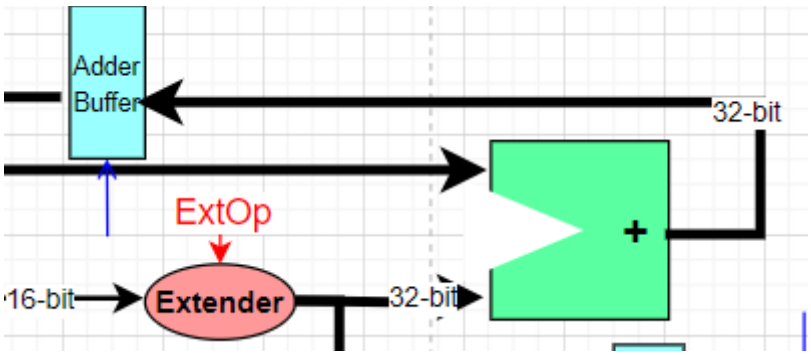

Figure 11: Adder for BTA in Decode Stage

```
if (zeroFlag == 1'b1)
    PC = PC + extendedImm; //branch taken --> PC = BTA
else
    PC = PC + 1;
```

Figure 12: Adder for BTA Implementation

Stack Pointer (SP)

The stack pointer is a 32-bit register that holds the address of the top-most empty element in the stack memory, it has one output that feeds back into the SP, after (increment by '1' in cases

of push, or decrement by '1' in cases of pop, or nothing in other instructions) and goes into a 2x1 MUX.

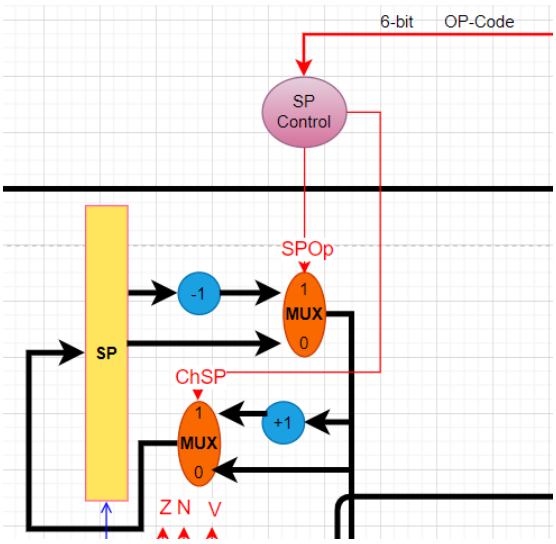

Figure 13: Stack Pointer in Memory Stage

As the figure shows, stack pointer is incremented or decremented depending on the signals that is generated from SP control.

```
reg [31:0] SP;
```

Figure 14: Stack Pointer Implementation

Data & Stack Memory

Both Data, and Stack are implemented inside the data memory of size 1024 words, each word is 32-bit, with the last 256 words reserved for stack uses, and the first 768 words are for the data memory.

This memory has 4 inputs as follows:

- Address (32-bit): the address of the word that needs to be accessed.
- Data_in (32-bit): the data that is to be written on the memory at Memory[address], in cases of write (store, push, pop, call).
- MemR (signal): a signal input that specifies if the instruction is a reading from memory, and it is generated by the Main Control unit.
- MemW (signal): a signal input that specifies if the instruction is writing on the memory, and it is generated by the Main Control unit.

The Memory has one 32-bit output, that holds the read data from memory if the instruction reads from the memory. This output is fed into the 4x1 MUX that selects the proper PC value for next instruction.

The addsrc signal specifies weather to select the address from SP in cases of instructions that uses the stack, or select from ALU output in cases of load and store instructions.

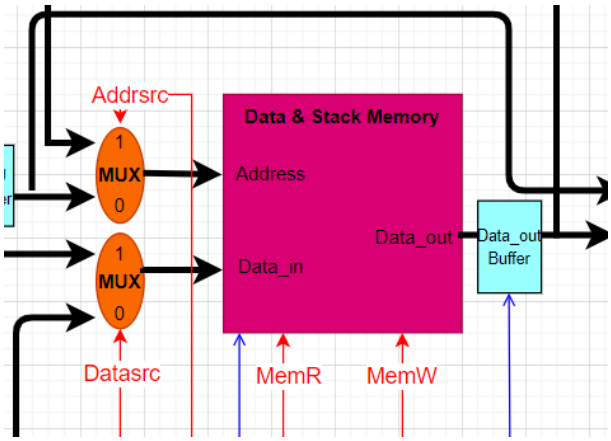

Figure 15: Data_Stack Memory in Memory Stage

```
// define the data and stack memroy
// the actual Data & Stack Memory
reg signed [31:0] Data_SatckMemory[0:1023];
```

Figure 16: Data /Stack Memory Implementation part1

```
    // variable to save address used in memory, data in and data out from memory
    reg signed [31:0] Memaddress;
    reg signed [31:0] dataIn;
    output reg signed [31:0] dataOut;
    output reg signed [31:0] WBData; // Data be written to Rigster file (last MUX)
```

Figure 17: Data /Stack Memory Implementation part2

```
// -------------------- Data Memory -------------------------
task dataMem(input [31:0] Address, input signed [31:0] Data_in, input MemR, MemW, output signed [31:0] Data_out);
    if(MemR==2'b1 && MemW==2'b0)
        Data_out=Data_SatckMemory[Address];
    else if(MemR==2'b0 && MemW==2'b1)
        Data_SatckMemory[Address]=Data_in;

    if(MemR == 1 || MemW == 1)
        begin
            $display("Data_in: %2d, Data_out: %2d, Address: %2d", Data_in, Data_out, Address);
            $display("MemR: %b, MemW: %b, Data_SatckMemory[%0d]: %2d\n", MemR, MemW, Address,Data_SatckMemory[Address]);
        end
endtask
```

Figure 18: Data /Stack Memory Implementation part3

Write Back Stage

In this stage, the output of the ALU or the output of the memory is stored back, and written on the register file, the choice between ALU output and memory output is done by using a 2x1 Mux that selects based on a signal called DataWB, if the data written back is from memory or ALU output. For the data to be written on the register file, the control signal RegW must be set ('1').

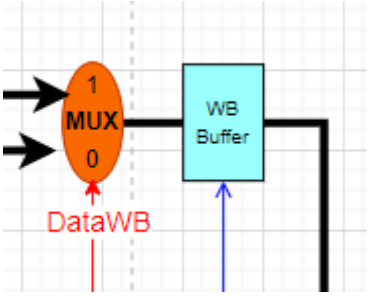

Figure 19: Data Write Bake Stage

2. Control Signals

We have many control signals generated from 4 different units, which are main control unit, PC control unit, SP control unit and ALU control unit. We divide the control signals on these 4 units to simplify understanding of each signal also to build and test it more efficiently. Control signals are used to distinct the needed functional units from instruction to another for example: in R-type instructions the write back data is the one generated from ALU, but for load instruction the write back data is the data loaded from memory. So, to conclude the control signals are basic for true flow in our processor also it is used by processor to help him in distinguishing between instructions.

A- PC Control

This control is used to choose the value of the PC which can be: PC + 1 for normal execution, BTA chosen when the branch is taken, JTA when the instruction is jump or call and top of the stack when the instruction is return. For branches the condition is calculated using ALU subtraction operation and the decision is made on output flags for BEQ, BNE the need flags is the zero flag and BEQ is taken when Z == 1, the same for BNE when Z == 0. For BGT, BLT our main idea is that we revers the operation without any extra hardware needed resources, so if we need to find to decide if Rd > Rs1 the same as is we check if Rs1 < Rd so the logic remain the same and we use this criteria for BLT also, we decide to use this mechanism to keep the first operand of our ALU Rs1 always and Rd for second operand, by this to check BGT to be taken if N != V (flags for normal less than), and for BLT if (Z == 0 && N == V) (flags for normal greater than).

PC Control Truth Table

Table 1. PC Control Truth Table

OpCode	Z_Flag	N_Flag	V_Flag	PCsrc
BGT	X	1	0	1 (T)
BGT	X	0	1	1 (T)
BGT	X	0	0	0 (NT)
BGT	X	1	1	0 (NT)
BLT	0	0	0	1 (T)
BLT	0	1	1	1 (T)
BLT	1	X	X	0 (NT)
BEQ	1	X	X	1 (T)
BEQ	0	X	X	0 (NT)
BNE	0	X	X	1 (T)
BNE	1	X	X	0 (NT)
JMP	X	X	X	2
CALL	X	X	X	2
RET	X	X	X	3
Others	X	X	X	0

PC Control Boolean Equations

If (OpCode == JMP || OpCode == CALL) **PCsrc** = 2; (Jump Target Address)

Else if ((OPCode == BGT && N != V) || (OPCode == BLT && Z == 0 && N == V) || (OPCode == BEQ && Z == 1) || (OPCode == BNE && Z == 0)) **PCsrc** = 1; (Branch Target Address)

Else if (OPCode == RET) **PCsrc** = 3; (PC = Top of the stack)

B- Main Control

This unit is major for most instruction because it produce the most needed signals which are used in decode, execute, memory and write back stages. The Reg2Src signal is used to decide what is the second operand for register file (Rs2 or Rd), ExtOp used to signed or unsigned extend the immediate, Reg1W for enable writing on Rs1 register or not (used in LW.POI), RegW used to enable writing BusW to Rd, ALUsrc to decide if the second operand of ALU is register or extended immediate, Datasrc to choose if the data being writing to memory come from PC + 1 or from BusB, Addsrc to choose if the address used in memory come from ALU result or SP, MemR to enable read from memory, MemW to enable write to memory, and DataWB to decide if the data being written back is from memory output or ALU result.

Main Control Truth Table

Table 2. Main Control Truth Table

OpCode	Reg2Src	Rs1W	RegW	ALUsrc	ExtOp	Datasrc	Addsrc	MemR	MemW	DataWB
R-Type	1	0	1	0	X	X	X	0	0	1
ANDI	X	0	1	1	0^{US}	X	X	0	0	1
ADDI	X	0	1	1	1^{S}	X	X	0	0	1
LW	X	0	1	1	1^{S}	X	0	1	0	0
LW.POI	X	1	1	1	1^{S}	X	0	1	0	0
SW	0	0	0	1	1^{S}	1	0	0	1	X
BGT	0	0	0	0	1^{S}	X	X	0	0	X
BLT	0	0	0	0	1^{S}	X	X	0	0	X
BEQ	0	0	0	0	1^{S}	X	X	0	0	X
BNE	0	0	0	0	1^{S}	X	X	0	0	X
JMP	X	0	0	X	X	X	X	0	0	X
CALL	X	0	0	X	X	0	1	0	1	X
RET	0	0	0	X	X	1	1	0	1	X
PUSH	0	0	0	X	X	1	1	0	1	X
POP	X	0	1	X	X	X	1	1	0	0

Main Control Boolean Equations

Reg2Src = R-Type

Rs1W = LW.POI

RegW = R-Type + ANDI + ADDI + LW + LW.POI

ALUsrc = ANDI + ADDI + LW + LW.POI + SW

ExtOp = ADDI + LW + LW.POI + SW + BGT + BLT + BEQ + BNE

Datasrc = SW + RET + PUSH

Addsrc = CALL + RET + PUSH + POP

MemR = LW + LW.POI + POP

MemW = SW + CALL + RET + PUSH

DataWB = R-Type + ANDI + ADDI

C- SP Control

This control was added to simplify working with SP (as with PC), its responsibility control what enter and out SP and how to update its value. It generates two signals: SPOp which used to choose the address of memory is 0 then the address is SP else the address is SP – 1 indicating that the instruction needs to pop value (we have incremental stack memory), ChSP (change SP) it is used to store the new value on SP it may be the previous value (when ChSP = 0) or it is incremented by 1 (when we push new value the address of stack increased).

SP Control Truth Table

Table 3. SP Control Truth Table

OpCode	SPOp	ChSP
AND	0	0
ADD	0	0
SUB	0	0
ANDI	0	0
ADDI	0	0
LW	0	0
LW.POI	0	0
SW	0	0
BGT	0	0
BLT	0	0
BEQ	0	0
BNE	0	0
JMP	0	0
CALL	0	1
RET	1	0
PUSH	0	1
POP	1	0

SP Control Boolean Equations

SPOp = RET + POP

ChSP = CALL + PUSH

D- ALU Control

The responsibility of this unit is to give the ALU the decision about what the proper operation that must done in ALU, which can be: logical and, arithmetic addition or arithmetic subtraction. So, from these 3 different operations we conclude that the ALU needs 2-bits to distinguish operations encoded as (AND = 00, ADD = 01, SUB = 10). After this assignment we check each instruction what its needed operation from ALU and give the signal using this assignment as indicated in the following truth table.

ALU Control Truth Table

Table 4. ALU Control Truth Table

OpCode	ALUOp	Operation
AND	**0**	**AND**
ADD	**1**	**ADD**
SUB	**2**	**SUB**
ANDI	**0**	**AND**
ADDI	**1**	**ADD**
LW	**1**	**ADD**
LW.POI	**1**	**ADD**
SW	**1**	**ADD**
BGT	**2**	**SUB**
BLT	**2**	**SUB**
BEQ	**2**	**SUB**
BNE	**2**	**SUB**
JMP	**X**	**X**
CALL	**X**	**X**
RET	**X**	**X**
PUSH	**X**	**X**
POP	**X**	**X**

ALU Control Boolean Equations

AND = 00 → AND + ANDI

ADD = 01 → ADD + ADDI + LW + LW.POI + SW

SUB = 10 → SUB + BGT + BLT + BEQ + BNE

3. Datapath Block Diagram

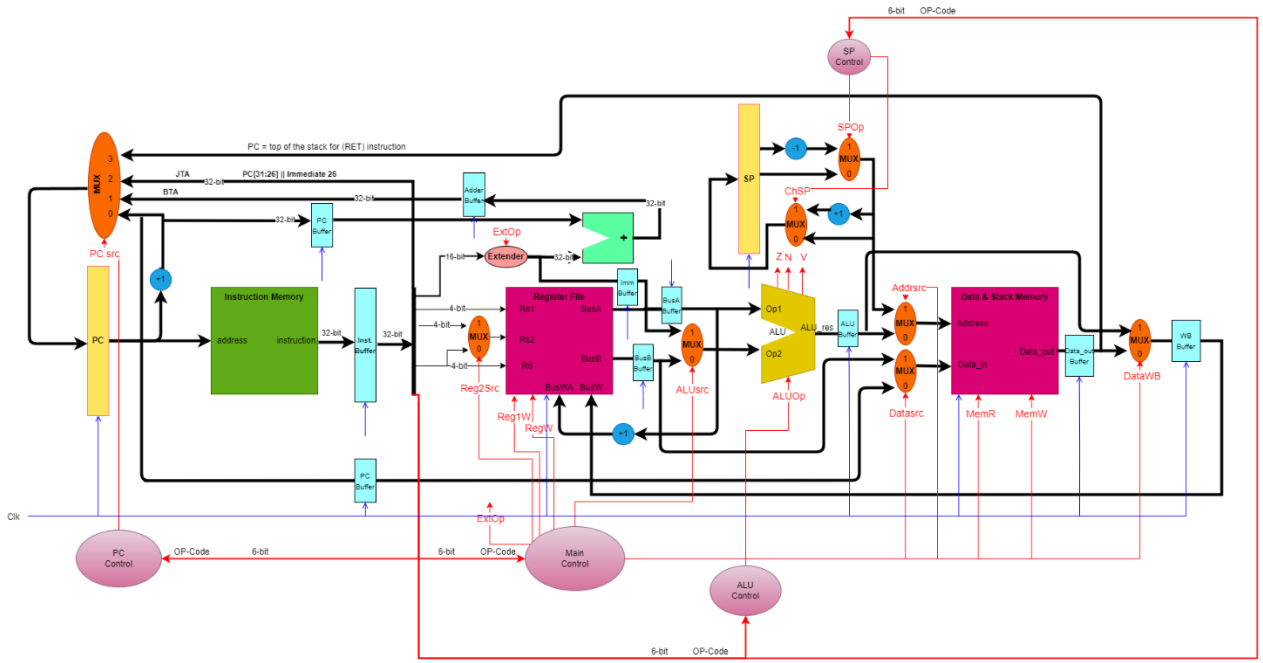

Figure 20. Datapath Block Diagram

We construct this Datapath using iterative approach first by construct the needed functional units for R-type and then for I-type, J-type then S-type and to know what is the needed functional unit for each type we use the register transfer language also to know how many cycles needed for each instruction to complete its operation.

4. Multi-cycle Processor Finite Stat Machine (FSM)

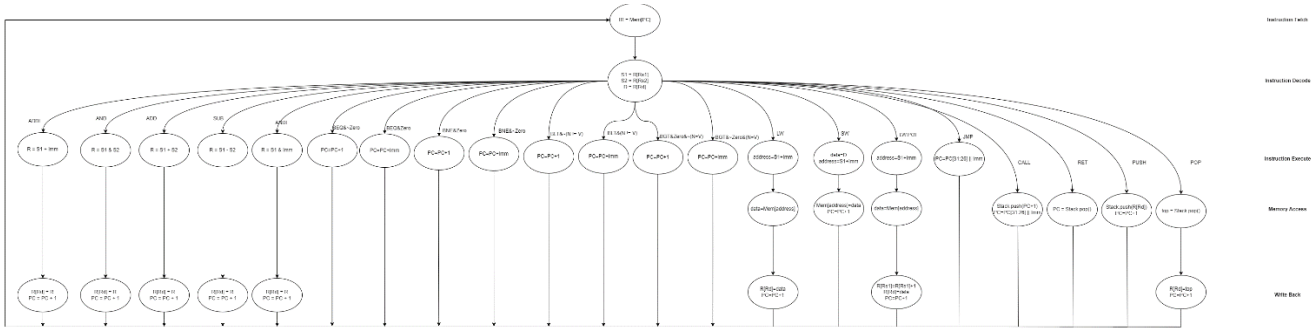

Figure 21. FSM

As we studied in the course to construct the multi-cycle, we need finite state machine because each instruction needs different cycles with different behavior. After this we implement our processor using Verilog code and then test each component and each instruction then check the results as indicated in the following section.

Note: if you need to see figure in full quality the link of each diagrams are in reference part.

2-Simulation and Testing

To test multiple programs, we save the instructions in the instruction memory also any needed data will be saved to data and stack memory, then we run the code and see the output result using both waveforms and display information on console in each stage.

1. Test program 1

The first program check R-type and load instructions as following code sequence:

LW.POI R1, 0(R0)

LW R2, 0(R0)

ADD R3, R1, R2

SUB R4, R1, R2

AND R5, R1, R2

Data memory have two values as follows → Memory [0] = 10, Memory [1] = 2

And the register R0 pointing to the first address of memory → R0 = 0

So, after executing the code the result must be → R1 = 10, R2 = 2, R3 = R1 + R2 = 10 + 2 = 12

R4 = R1 – R2 = 10 – 2 = 8, R5 = R1 & R2 = 10 & 2 = 2. To check the result, we use wave form and the console output.

```
// program instructions
Memory[0] = 32'b00010100010000000000000000000001; // LW.POI R1, 0(R0) --> R0 points to the address of data being loadded (R1 = 10) --> R0 = R0 + 1
Memory[1] = 32'b00010100100000000000000000000000; // LW R2, 0(R0) --> R0 points to the address of data being loadded (R2 = 2)
Memory[2] = 32'b00000100110001001000000000000000; // ADD R0, R1, R2
Memory[3] = 32'b00001001000001001000000000000000; // SUB R4, R1, R2
Memory[4] = 32'b00000001010001001000000000000000; // SUB R5, R1, R2

// data instructions
Data_SatckMemory[0] = 32'd10; // the data for R1
Data_SatckMemory[1] = 32'd2; // the data for R2
```

Figure 22. Program 1 code and data

The first instruction **LW.POI R1, 0(R0)** is executed as follows:

```
***************Welcome to our processor***************

-----------------Fetch Cycle-----------------

PC=         0
Instruction = 00010100010000000000000000000001

-----------------Decode Cycle-----------------

Signals:
Reg2Src = x, RegW = 1, Rs1W, = 1, ALUsrc = 1, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 1, MemW = 0, DataWB = 0

-----------------Execute Cycle-----------------

Extended immediate= 0, BusA= 0, BusB= 0

op1=  0, op2= 0

ALU result=  0, z=1, n=0, v=0

-----------------Memory Cycle-----------------

Data_in:  x, Data_out: 10, Address:  0
MemR: 1, MemW: 0, Data_SatckMemory[0]: 10

-----------------Write Back Cycle-----------------

The register R0 updated with new value =  1

The register R1 updated new value = 10
```

Figure 23. P1 I1

The instruction LW.POI needs 5 stages → IF, ID, Ex, Mem, WB

And the result of each stage is indicated in the figure the final result is that value 1 is stored in R0 and value 10 in R1 and this is what expected from instruction to do and it needs 5 positive edge cycles to perform the operation.

The second instruction **LW R2, 0(R0)** we expect to need 5 cycles as first instruction but the main difference is that the value of R0 must not change (no increment).

```
-----------------Fetch Cycle-----------------

PC=         1
Instruction = 00010100100000000000000000000000

-----------------Decode Cycle-----------------

Signals:
Reg2Src = x, RegW = 1, Rs1W, = 0, ALUsrc = 1, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 1, MemW = 0, DataWB = 0

-----------------Execute Cycle-----------------

Extended immediate= 0, BusA= 1, BusB= 0

op1=  1, op2= 0

ALU result=  1, z=0, n=0, v=0

-----------------Memory Cycle-----------------

Data_in:  x, Data_out:  2, Address:  1
MemR: 1, MemW: 0, Data_SatckMemory[1]:  2

-----------------Write Back Cycle-----------------

The register R2 updated new value =  2
```

Figure 24. P1 I2

We see that the instruction is done properly as what expected and R2 is loaded with 2 without any change to R0.

The third instruction **ADD R3, R1, R2**

If from R-type and needs only 4 cycles → IF, ID, Ex, WB (no need to memory is this instruction). After this the change is only on R3

```
------------------Fetch Cycle------------------

PC=         2
Instruction = 00000100110001001000000000000000


------------------Decode Cycle------------------

Signals:
Reg2Src = 1, RegW = 1, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 1


------------------Execute Cycle------------------

Extended immediate=8192, BusA=10, BusB= 2

op1= 10, op2= 2

ALU result= 12, z=0, n=0, v=0


------------------Write Back Cycle------------------

The register R3 updated new value = 12
```

Figure 25. P1 I3

Also, the result is same as we expected.

The fourth instruction **SUB R4, R1, R2**

Needs 4 cycles as above one (R-type)

```
------------------Fetch Cycle------------------

PC=         3
Instruction = 00001001000001001000000000000000


------------------Decode Cycle------------------

Signals:
Reg2Src = 1, RegW = 1, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 1


------------------Execute Cycle------------------

Extended immediate=8192, BusA=10, BusB= 2

op1= 10, op2= 2

ALU result=  8, z=0, n=0, v=0


------------------Write Back Cycle------------------

The register R4 updated new value =  8
```

Figure 26. P1 I4

The result is stored in R4 = 8.

The fourth instruction **AND R5, R1, R2**

Needs 4 cycles.

```
------------------Fetch Cycle------------------

PC=         4
Instruction = 00000001010001001000000000000000


------------------Decode Cycle------------------

Signals:
Reg2Src = 1, RegW = 1, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 1


------------------Execute Cycle------------------

Extended immediate=8192, BusA=10, BusB= 2

op1= 10, op2= 2

ALU result=  2, z=0, n=0, v=0


------------------Write Back Cycle------------------

The register R5 updated new value =  2
```

Figure 27. P1 I5

The result is stored in R5 = 2.

The waveform:

Figure 28. P1 WF

The waveform is accepted and good because we see at each clock cycle positive edge new stage is operated. We put the result of ALU, Memory out and writeback for our program.

If we trace first instruction **LW.POI R1, 0(R0)** → the first cycle is fetch, then we decode it, then execute the address used for memory, then load the value from memory and the last stage is to write the result obtained form memory to the register destination.

To conclude this simple program code, check the R-type, LW and LW.POI instructions and it perform as we expect with good results.

2. Test Program 2

BGT R1, R2, sub // displacement by 2
ADD R1, R1, R2
BLT R2, R1, exit // displacement by 1
sub: SUB R4, R1, R2
exit:

Binary sequence:

Memory [0] = 32'b00100000010010000000000000001100;
Memory [1] = 32'b00000100010001001000000000000000;
Memory [2] = 32'b00100000010010000000000000011100;
Memory [3] = 32'b00001001000001001000000000000000;

Registers [1] = 32'd10;
Registers [2] = 32'd5;

Expected output: R4 = 5

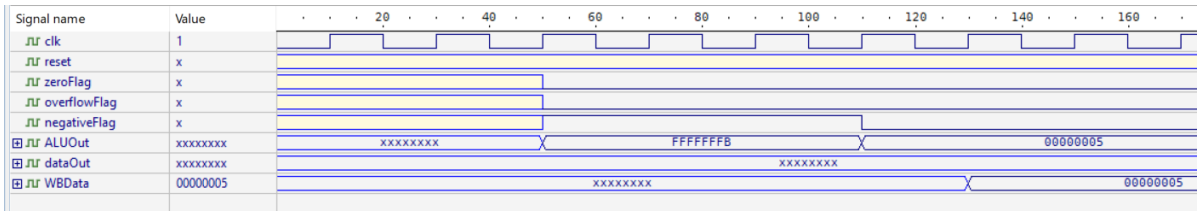

Figure 29: Test program 2 Waveform

as we can see from the figure, the final data on the write back bus is 5, which is correct.

First instruction:

BGT R1, R2, sub // displacement by 2

This branch condition should be true, since r1 > r2, and it should branch to memory address 3.

```
------------------Fetch Cycle------------------

PC=         0
Instruction = 00100000010010000000000000001100


------------------Decode Cycle------------------

Signals:
Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = x, MemR = 0, MemW = 0, DataWB = x


------------------Execute Cycle------------------

Extended immediate= 3, BusA= 5, BusB=10

op1=  5, op2=10

ALU result= -5, z=0, n=1, v=0


------------------Fetch Cycle------------------

PC=         3
Instruction = 00001001000001001000000000000000
```

Figure 30: P2 Console 1

As we saw from the above figure, after the first instruction is executed, the alu result is -5, which is (5 - 10), and the flags are set correctly, after that, in the next cycle the PC = 3, which is where sub label is pointing to.

Second instruction:

sub: SUB R4, R1, R2

The figure below shows that after instruction at address 3 is loaded and decoded, the ALU inputs are op1 = 10, op2 = 5, which is the values of R1, and R2 respectively. In the write back stage, the register R4 is updated with value = 5, this is the final answer of the program and it is running as expected.

```
PC=         3
Instruction = 00001001000001001000000000000000

-----------------Decode Cycle-----------------

Signals:
Reg2Src = 1, RegW = 1, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = x, MemR = 0, MemW = 0, DataWB = 1

-----------------Execute Cycle-----------------

Extended immediate=8192, BusA=10, BusB= 5

op1= 10, op2= 5

ALU result=  5, z=0, n=0, v=0

-----------------Write Back Cycle-----------------

The register R4 updated new value =  5
```

Figure31 : P2 Console 2

Repeating the program after updating the values of R1, and R2 as follows:

Registers [1] = 32'd5;
Registers [2] = 32'd10;

Final output: R1 = 15

The figure below shows that the data at ALU output, and the data on WB bus is equal to 15 (F) in hex.

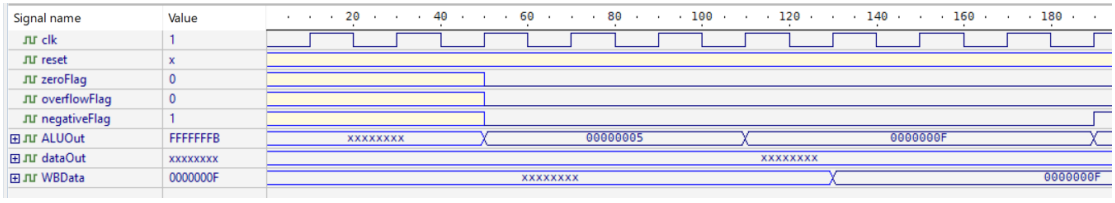

Figure 32: Program 2 Waveform 2

First instruction:

BGT R1, R2, sub // displacement by 2 if true

This instruction condition will be false, hence the PC will be 1 (no displacement). The following figure, shows the console output, after the first instruction is executed, and the branch did not happen.

```
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         0
# KERNEL: Instruction = 00100000010010000000000000001100
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = x, MemR = 0, MemW = 0, DataWB = x
# KERNEL:
# KERNEL:
# KERNEL: ------------------Execute Cycle------------------
# KERNEL:
# KERNEL: Extended immediate= 3, BusA=10, BusB= 5
# KERNEL:
# KERNEL: op1= 10, op2= 5
# KERNEL:
# KERNEL: ALU result=  5, z=0, n=0, v=0
# KERNEL:
# KERNEL:
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         1
# KERNEL: Instruction = 00000100010001001000000000000000
```

Figure 33: P2 Console 3

Second instruction:

ADD R1, R1, R2 // R1 = 15

The figure below shows that after executing this instruction, value of 15 is stored at R1, which is correct.

```
------------------Fetch Cycle------------------

PC=         1
Instruction = 00000100010001001000000000000000


------------------Decode Cycle------------------

Signals:
Reg2Src = 1, RegW = 1, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = x, MemR = 0, MemW = 0, DataWB = 1


------------------Execute Cycle------------------

Extended immediate=8192, BusA= 5, BusB=10

op1=  5, op2=10

ALU result= 15, z=0, n=0, v=0


------------------Write Back Cycle------------------

The register R1 updated new value = 15
```

Figure 34: P2 Console 4

Third instruction:

BLT R2, R1, Exit // Exit is memory address 9 (displacement by 7)

The console log below shows the BLT execution, since R2 < R1, then the condition is true, and the branch should take place, so that PC will be equal to 9 (displaced by 7), and it is correct as the PC value after this instruction is 9.

```
------------------Fetch Cycle------------------

PC=         2
Instruction = 00100000010010000000000000011100


------------------Decode Cycle------------------

Signals:
Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = x, MemR = 0, MemW = 0, DataWB = 1


------------------Execute Cycle------------------

Extended immediate= 7, BusA=10, BusB=15

op1= 10, op2=15

ALU result= -5, z=0, n=1, v=0


------------------Fetch Cycle------------------

PC=         9
Instruction = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 35: P2 Console 5

3. Test Program 3

The program in assembly:

AND R0, R1, R2

ADD R0, R1, R2

LW R4, 1(R5)

LW.POI R6, 2(R5)

loop: BEQ R4, R6, toPush

SUB R6, R6, R1

JMP loop

PUSH R6

ANDI R0, R7, 1

BNE R7, R2, store

JMP ADDThenSW

RET

ADD R0, R1, R2

store: SW R0, 2(R3)

CALL the popFunc

ADDI R0, R7, 1

SW R0, 2(R3)

The data memory has the following values:

Memory [1] = 20 and Memory [2] = 23

Also, the register file has the following values that it needed in the program:

R1 = 32'd1;

R2 = 32'd2;

R3 = 32'd10;

R5 = 32'd0;

R7 = 32'b01111111111111111111111111111111;

After this program is executed, all results were as needed, also the branches are executed properly as shown on the figures below.

```
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         4
# KERNEL: Instruction = 00101001000110000000000000001100
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 0
# KERNEL:
# KERNEL:
# KERNEL: ------------------Execute Cycle------------------
# KERNEL:
# KERNEL: Extended immediate= 3, BusA=23, BusB=20
# KERNEL:
# KERNEL: op1= 23, op2=20
# KERNEL:
# KERNEL: ALU result=  3, z=0, n=0, v=0
# KERNEL:
# KERNEL:
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         5
# KERNEL: Instruction = 00001001100110000100000000000000
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 1, RegW = 1, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 1
# KERNEL:
# KERNEL:
# KERNEL: ------------------Execute Cycle------------------
# KERNEL:
# KERNEL: Extended immediate=4096, BusA=23, BusB= 1
# KERNEL:
# KERNEL: op1= 23, op2= 1
# KERNEL:
# KERNEL: ALU result= 22, z=0, n=0, v=0
# KERNEL:
# KERNEL:
# KERNEL: ------------------Write Back Cycle------------------
```

Figure 36: P3 Console 1

As noted, when the values are not equal, the BEQ is not taken and enter in subtraction loop. After the values are equal, the branch will occur. So, the branches work properly. As shown on the figure below.

```
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         6
# KERNEL: Instruction = 00110000000000000000000000000100
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 1, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 1
# KERNEL:
# KERNEL:
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         4
# KERNEL: Instruction = 00101001000110000000000000001100
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = x, AddSrc = 0, MemR = 0, MemW = 0, DataWB = 1
# KERNEL:
# KERNEL:
# KERNEL: ------------------Execute Cycle------------------
# KERNEL:
# KERNEL: Extended immediate= 3, BusA=20, BusB=20
# KERNEL:
# KERNEL: op1= 20, op2=20
# KERNEL:
# KERNEL: ALU result=  0, z=1, n=0, v=0
# KERNEL:
# KERNEL:
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=         7
# KERNEL: Instruction = 00111101100000000000000000000000
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 0, ExtOp = 1, DataSrc = 1, AddSrc = 1, MemR = 0, MemW = 1, DataWB = 1
```

Figure 37: P3 Console 2

The last result was as the expected. Since the result of this operation (in binary) is 01111111111111111111111111111111 + 1. So, the result is correct.

```
# KERNEL: ------------------Fetch Cycle------------------
# KERNEL:
# KERNEL: PC=        16
# KERNEL: Instruction = 00011100000011000000000000001000
# KERNEL:
# KERNEL:
# KERNEL: ------------------Decode Cycle------------------
# KERNEL:
# KERNEL: Signals:
# KERNEL: Reg2Src = 0, RegW = 0, Rs1W, = 0, ALUsrc = 1, ExtOp = 1, DataSrc = 1, AddSrc = 0, MemR = 0, MemW = 1, DataWB = 1
# KERNEL:
# KERNEL:
# KERNEL: ------------------Execute Cycle------------------
# KERNEL:
# KERNEL: Extended immediate= 2, BusA=10, BusB=-2147483648
# KERNEL:
# KERNEL: op1= 10, op2= 2
# KERNEL:
# KERNEL: ALU result= 12, z=0, n=0, v=0
# KERNEL:
# KERNEL:
# KERNEL: ------------------Memory Cycle------------------
# KERNEL:
# KERNEL: Data_in: -2147483648, Data_out: 15, Address: 12
# KERNEL: MemR: 0, MemW: 1, Data_SatckMemory[12]: -2147483648
# KERNEL:
```

Figure 38: P3 Console 3

The waveform of the compilation:

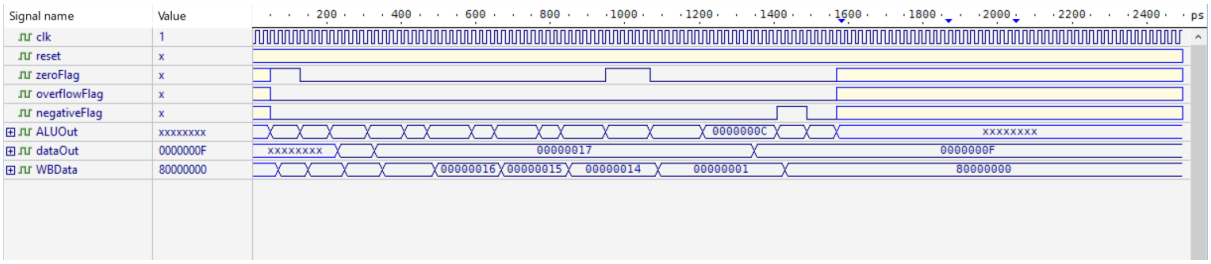

Figure 39: P3 WaveForm

As noted, the waveform shows each stage on the clk and the steps of executing the program. The result is correct since as shown the last value of dataOut (memoryOut) = F (15) which is the returned address.

3-Teamwork

We work together at each project stage from thinking to designing and finally testing. We change our wrong ideas. Also, we cooperate in writing the report and tracing and testing our programs code.

4-Conclusion

In this project we understand how to build processor and how the processor works. How to operate different instructions and see its different behavior and needs. To add, how to build processor Datapath and how to generate the controls needed for proper flow of data. Finally, we test our processor using different programs to see if there is any failure, detect it and solve it.

5-References

[1]: Course Slides

[2]: https://drive.google.com/file/d/1_tOGO18yeU33j8KcmIpEaPgwZ--HxOiu/view?usp=sharing [for Datapath block diagram]

[3]:https://drive.google.com/file/d/1jzvdmbbpq6is2vy4VCNUQKihJeLVPZ0B/view?usp=sharing [for FSM block diagram]

[4]: draw.io for drawing diagrams

[5]: advanced digital Verilog slides (to review Verilog code)