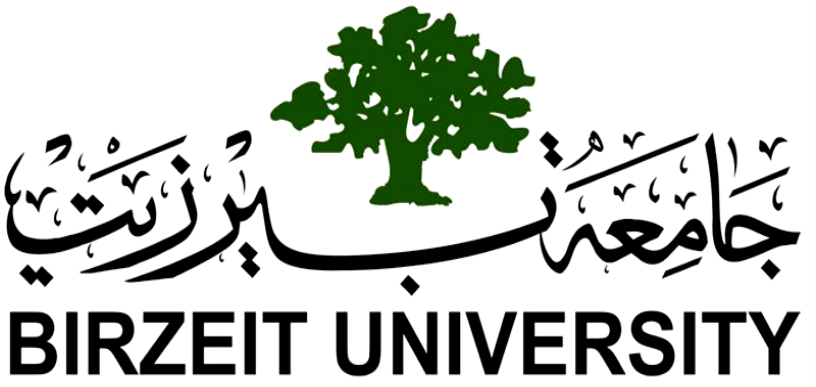

Faculty of Engineering & Technology

Electrical & Computer Engineering Department

First Semester 2022/2023

Computer Architecture ENCS4370

Project #2

Single-Cycle RISC processor using HDL language

Prepared by:

Hasan Hamed 1190496

Shorooq Najjar 1192415

Instructor: Dr. Ayman Hroub

Date: 15/2/2023

1. Abstract

The aim of this project is to design the datapath and control unit for a single cycle processor for a given ISA using Verilog, and to learn the components of the datapath and the importance of each, and to derive the control signals equations for the control unit, then connecting the datapath to the control unit to implement the final datapath and finally, testing all the instructions to verify the design.

Table of Contents

Table of Figures

List of Tables

2. Design Specifications

2.1. Instruction Set Architecture

1. The instruction size is 24 bits (word size 24-bit)
2. All instructions are conditionally executed.
3. Eight 24-bit general-purpose registers: R0 through R7. 4. R0 is hardwired to zero and cannot be written. Any instruction attempts to write R0 is discarded.
5. The program counter (PC) is a 24-bit special-purpose register (24-bit memory address space).
6. 8-bit status register. At this project, consider only the least significant bit in this status register, namely, the zero (Z) flag bit. This bit is set if the last ALU operation result is zero, and it is cleared otherwise.
7. The programmer can determine whether the ALU instruction should update the flag bits or not by appending the suffix SF to the instruction mnemonic, e.g., ADD just adds, but ADDSF adds and updates the flag bits (Z flag). The ALU instruction binary format contains a 1-bit field to encode this.
8. Three instruction types (R-type, I-type, and J-type).
9. Five addressing modes as in MIPS32 ISA

2.2. Instruction Formats

The ISA has three instruction formats, namely, R-type, I-type, and J-type. These three types have the following two common fields:

1. Opcode: a 5-bit field to tell the processor which operation to perform
2. Condition: a 2-bit field that enables every instruction to be conditionally executed. For this project, we will consider two conditions, i.e., equal and not equal as shown in Table 2.1.

Table 2.1: Condition Field and Effects

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

2.2.1. R-type Format

• 2-bit condition (Cond)
• 5-bit opcode (Op)
• 1-bit whether to set the flag bits or not (SF). For this project and for simplicity, this bit is used only with the subtraction instructions only (SUBSF, SUBISF). For other instructions, this bit is always 0. For example, SUBSF, R1, R2, R3 will perform the following
- Reg [R1] = Reg[R2] – Reg[R3]
- Zero-Flag = Reg[R2] == Reg[R3]
• 3-bit destination register (Rd)
• 3-bit source 1 register (Rs)
• 3-bit source 2 register (Rt)
• 7-bit unused (for future use)
Figure 2.1 shows the R-type instruction format.

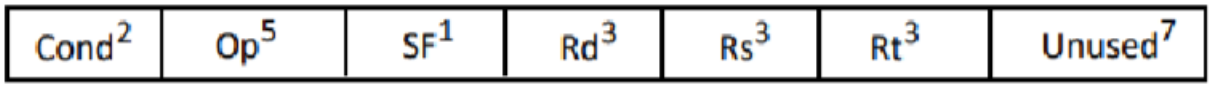

Figure 2.1: R-type Instruction Format

2.2.2. I-type Instruction Format

In addition to the common fields with R-type, it has:
• 3-bit destination register (Rt)
• 3-bit source 1 register (Rs)
• 10-bit signed immediate value in two's complement representation (2nd operand)
Figure 2.2 shows the I-type instruction format.

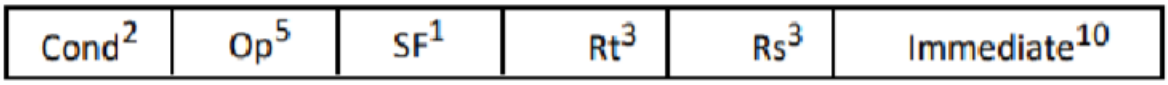

Figure 2.2: I-type Instruction Format

2.2.3. J-type Instruction Format

In addition to the condition and opcode fields, the J-type instruction contains a 17-bit signed immediate constant in two's complement representation, i.e., the jump offset as shown in Figure 2.3.

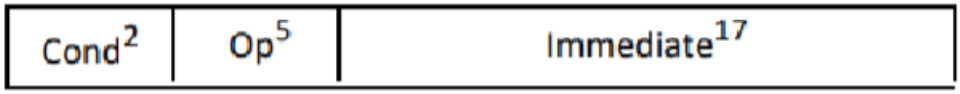

Figure 2.3: J-type Instruction Format

2.3. Instructions Set

Table 2.2 shows the instructions supported by this instruction set, with their meaning and decoding.

Table 2.2: Instruction Set

No.	Instr	Meaning	Encoding			
			RTypeInstructions			
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
			ITypeInstructions			
7	ANDI	Reg(Rt) = Reg(Rs) & Immediate10	Op = 00111	Rs	Rt	Immediate10
8	ADDI	Reg(Rt) = Reg(Rs) + Immediate10	Op = 01000	Rs	Rt	Immediate10
9	LW	Reg(Rt) = Mem(Reg(Rs) + Imm10)	Op = 01001	Rs	Rt	Immediate10
10	SW	Mem(Reg(Rs) + Imm10) = Reg(Rt)	Op = 01010	Rs	Rt	Immediate10
11	BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 01011	Rs	Rt	Immediate10
			JTypeInstructions			
12	J	PC = PC + Immediate17	Op = 01100	Immediate17		
13	JAL	R7 = PC + 1, PC = PC + Immediate17	Op = 01101	Immediate17		
14	LUI		Op = 01110	Immediate17		

3. Components

3.1. Instruction and Data Memory

To address potential conflicts and adhere to the isolation principle, the implementation employs two distinct memory units: one for instructions and another for data. This ensures that, for example, an instruction retrieval process does not interfere with a data loading or storing process, as they are stored in separate memory entities. Thus, the memories are divided into instruction and data memory segments. Figure 3.1 and 3.2 show the block diagrams of each.

Notice that we considered that each memory address contains one instruction and the memory is word addressable.

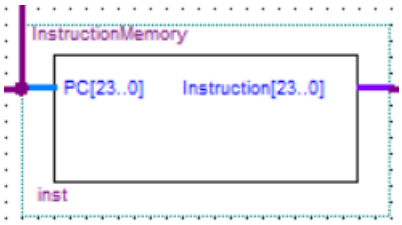

Figure 3.1: Instruction Memory

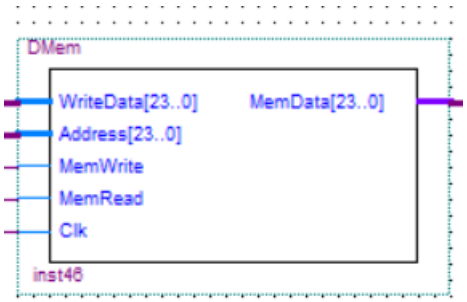

Figure 3.2: Data Memory

3.2. Register File

The register file as shown in Figure 3.3 is used to get the data of the register and decides which register to write.

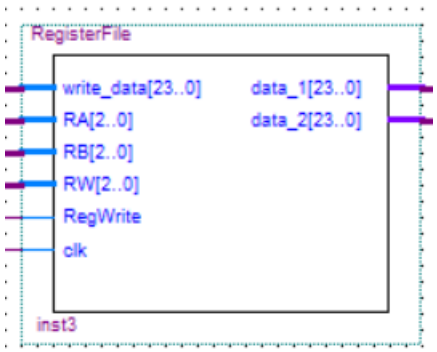

Figure 3.3: Register File

3.3. Extenders

There are two sign extenders as shown in Figure 3.4 for immediate values.

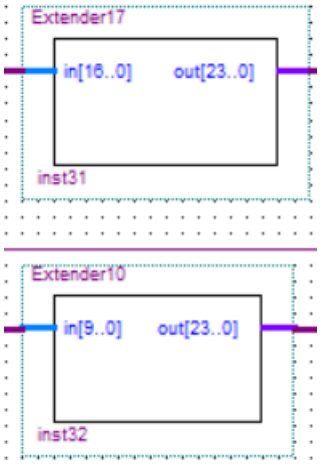

Figure 3.4: Extenders

3.4. Register File Muxes

There are two muxes before the register file as shown in Figure 3.5, the upper one choses the value of rb since it different for load and branch than others according to the specs.
The lower mux choses which register to write at since there are 3 possible registers to write to according to the specs which are R1, R7 and rd, rd and rt are the same when considering the I type.

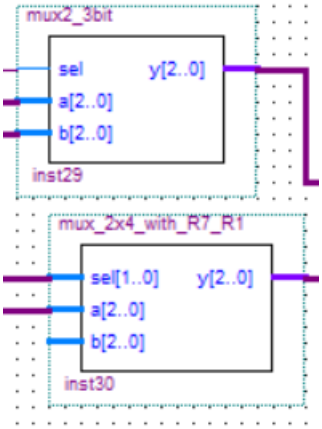

Figure 3.5: Register File Muxes

3.5. Add One

This adder is to increment the value of the PC as shown in Figure 3.6.

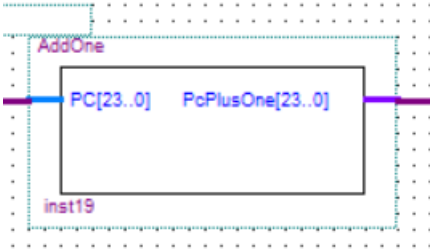

Figure 3.6: Add One Component

3.6. Concate

It is used to find the value of the jump address as shown in Figure 3.7.

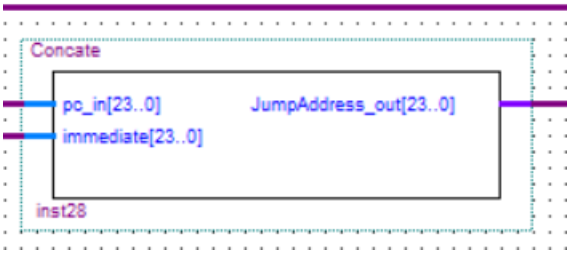

Figure 3.7: Concate Component

3.7. PC Source Mux

It is used to decide the value of the PC according to the instruction since the PC can have different value when branch or jump or JR as shown in Figure 3.8.

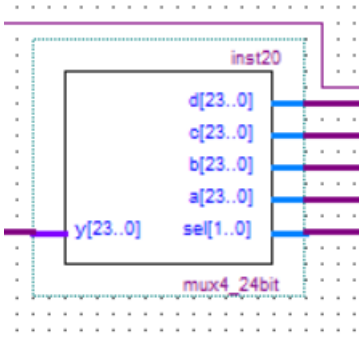

Figure 3.8: PC Source Mux

3.9. ALU and ALU Control

The ALU is used to execute the instructions according to the operation the ALU control gives which is decided based on the opcode of the instruction as shown in Figure 3.10 and 3.11. Notice that the sf was put on the ALU control which signals the ALU whether to update the zero flag or not if sf was 1.

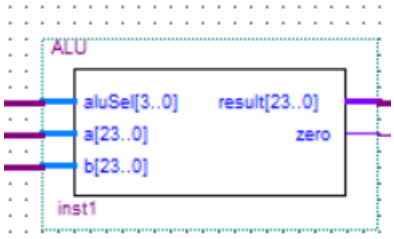

Figure 3.10: ALU

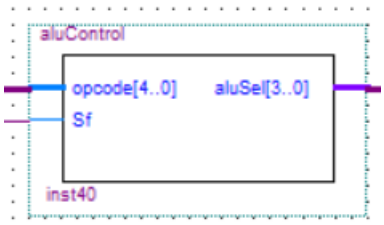

Figure 3.11: ALU Control

3.10. ALU Source Mux

This mux decides the second ALU source according to instruction if it I or R type as shown in Figure 3.12.

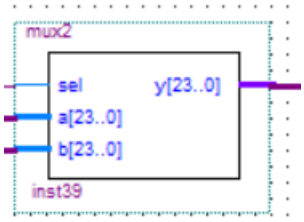

Figure 3.12: ALU Source Register

3.11. Branch Adder and MUX

The branch adder calculates the value of the address to branch to and the mux decides according to the zero flag whether to choose the normal PC or the branch address as shown in Figure 3.13.

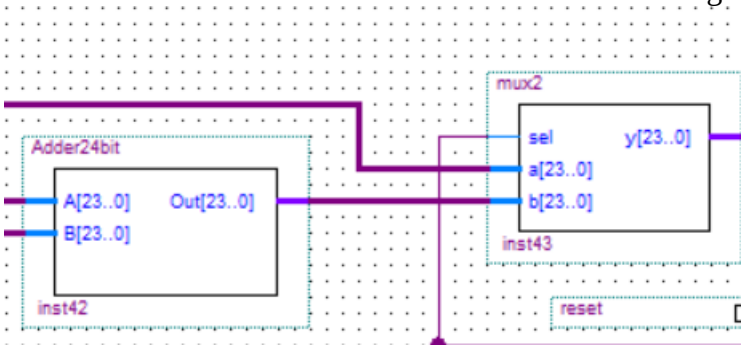

Figure 3.13: Branch Adder and MUX

3.12. Main Control

Which is the most important thing in the datapath used to update the control signals value according to each stage the instruction in as shown in Figure 3.14.

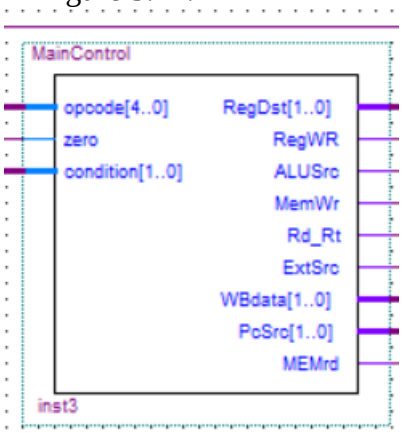

Figure 3.14: Main Control

3.13. Register Data to Write Mux

This MUX decides which data to write at the register since there is an ALU, memory and PC+1 data such as in JAL instruction as shown in Figure 3.15.

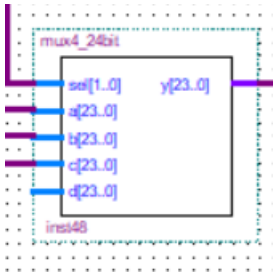

Figure 3.15: Writeback Data MUX

3.14. PC

The PC is used to point to the next instruction to be fetched, the value of the PC is updated according to the instruction which is decided by PCsrc signal as shown in Figure 3.16.

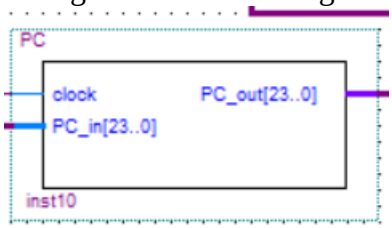

Figure 3.16: PC

3.15. Zero-bit Register

This register is used to save the value of the zero bit after any instruction that updates it as shown in Figure 3.17, the enable that allow the register to be updated will be demonstrated later.

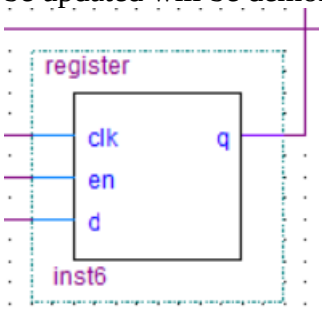

Figure 3.17: Zero-bit Register

4. Control Unit Design

4.1. Control Unit Truth Table

First, the truth table of the control signals was derived as shown in Table 4.1.

Table 4.1: Control Signals Truth Table

Instruction/Signal	PcSrc	ExtSrc	ALUsrc	Regwrite	Regdst	WBdata	MemRD	MemWr	Rd_Rt
AND	00	X	1	1	00	01	0	0	1
CAS	00	X	1	1	00	01	0	0	1
LWS	00	X	1	1	00	00	1	0	1
ADD	00	X	1	1	00	01	0	0	1
SUB	00	X	1	1	00	01	0	0	1
CMP	00	X	1	0	00	XX	0	0	1
JR	11	X	1	0	00	XX	0	0	1
ANDI	00	0	0	1	00	01	0	0	1
ADDI	00	0	0	1	00	01	0	0	1
LW	00	0	0	1	00	00	1	0	1
SW	00	0	0	0	00	XX	0	1	0
BEQ	01	0	1	0	00	XX	0	0	0
J	10	1	X	0	00	XX	0	0	1
JAL	10	1	X	1	11	11	0	0	1

LUI	00	1	0	1	10	01	0	0	1

Then, the control signals equations where derived as the follow:

$$MemRD = LW + LWS$$

$$RegWR = (\overline{CMP} + \overline{JR} + \overline{SW} + \overline{BEQ} + \overline{J})$$

$$ExtSrc = LW + J + JAL$$

$$ALUSrc = (\overline{LUI} + \overline{SW} + \overline{LW} + \overline{ADDI} + \overline{ANDI})$$

$$MemWr = SW$$

$$MemRD = LW + LWS$$

$$Rd\_Rt = (\overline{SW} + \overline{BEQ})$$

$$PcSrc = (JR + J + JAL) \,||\, (BEQ + JR)$$

$$RegDst = (LUI + JAL) \,||\, (JAL)$$

$$WBdata = JAL \,||\, (\overline{LW} + \overline{LWS})$$

4.2. Controlling the Zero Flag

As mentioned above, the zero-bit flag is stored in a register which has an input called zerotemp coming from the ALU and an enable signal named zeroen also from the ALU. The value of the zero flag in the register only changes when the enable is 1 which depends on the instruction that update the flag. The enable signal is derived as below:

$$zeroen = (aluSel == 0011) + (aluSel == 0110)$$

The aluSel is the signal coming from the ALU control unit and the values of aluSel are the opcodes of the CMP, and Sub and Subi where Sf=1.

5. Final Datapath

After constructing every component, the main datapath was built as shown in Figure 5.1.

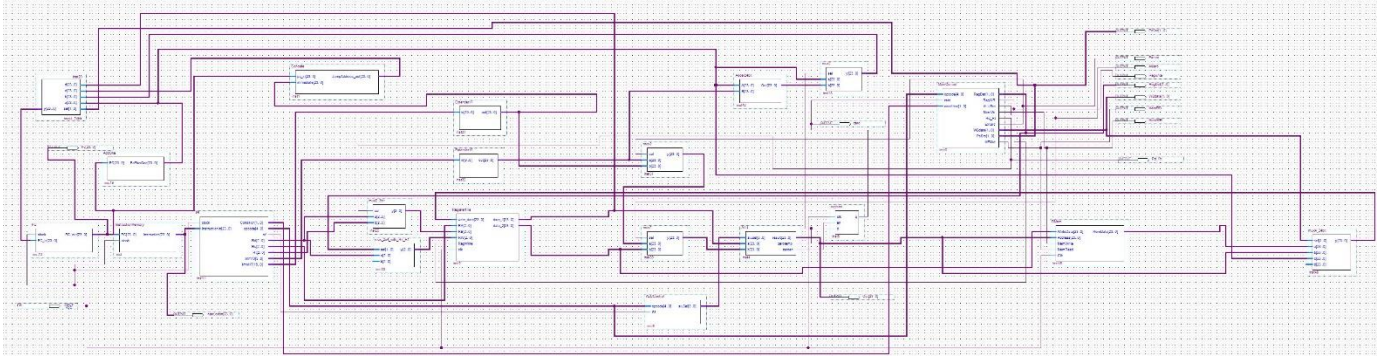

Figure 5.1: Final Datapath

5.1. Datapath Stages

5.1.1. Instruction Fetch

The address from the PC is used to fetch the instruction from memory, and the PC address can be modified to assume four distinct values JR, normal, branch and jump address as demonstrated in the truth table.

5.1.2. Instruction Decode

At this stage, the instruction is decoded into several components, including Cond, Opcode, SF, Rs, Rt, Rd, Immediate10, and Immediate17. Next, the Register File is read based on the control signals. This phase concludes after transferring the values of the registers and Immediate to the Execution stage.

5.1.3. Instruction Execute

In this stage, the ALU unit is utilized to execute the necessary operations as per the instruction. The phase concludes after transmitting the values to the Memory stage.

5.1.4. Memory

Based on the aforementioned control signals, the memory may undergo a read, write, or remain unaffected.

5.1.5. Write back

The write back stage is accountable for writing data to the Register File. The data that is written back can be sourced from the ALU or Memory. Additionally, the next PC value can be written back in the case of the JAL instruction.

6. Simulation Results

There must be notice that each instruction takes one cycle and fetching instruction, writing back to register and writing to memory are happened on the positive edge of the clock and according to the control signals.

6.1. I-type and R-type and Conditions

```
1  module InstructionMemory(
2    input [23:0] PC,
3    input clock,
4    output reg [23:0] Instruction
5  );
6  always@(posedge clock)
7  begin
8    case (PC)
9      0:   Instruction = 24'b00_01000_0_001_001_0000000011;          //ADDI R1, R1,3
10     1:   Instruction = 24'b00_01000_0_011_011_0000000010;          //ADDI R3, R3,2
11     2:   Instruction = 24'b00_00101_0_000_011_001_0000000;         //CMP R1, R3
12     3:   Instruction = 24'b10_00011_0_010_011_001_0000000;         //ADDNEQ R2,R3,R1
13     4:   Instruction = 24'b00_00011_0_100_011_001_0000000;         //ADD R4, R3,R1
14     //2:   Instruction = 24'b00_01000_0_011_011_0000000110;        //ADDI R3, R3,7
15     //2:   Instruction = 24'b00_01100_00000000000000111;
16     //2:   Instruction = 24'b00_01011_0_011_001_0000000001;        //BEQ R3, R1, 1
17
18
19     default: Instruction = 24'b0; // NOOP
20   endcase
21 end
22 endmodule
```

Figure 6.1: Test Case 1

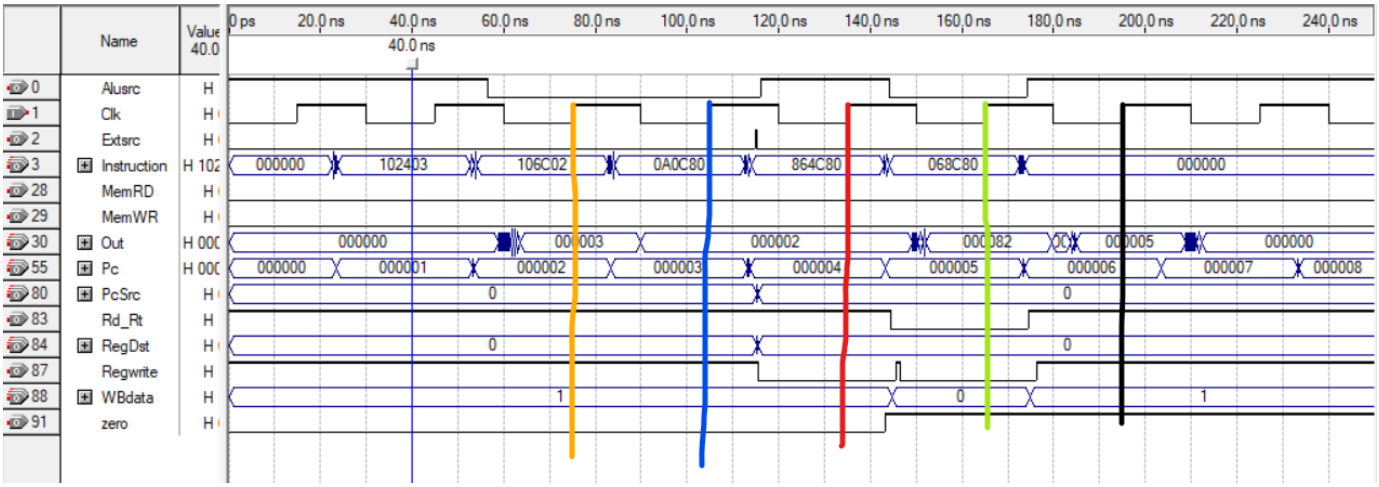

Figure 6.2: First Test Case Result

As shown in Figure 6.2, each instruction executed correctly and they all output correct results and the compare and conditions are correct.

6.2. BEQ

```
1   module InstructionMemory(
2     input [23:0] PC,
3     input clock,
4     output reg [23:0] Instruction
5   );
6   always@(posedge clock)
7   begin
8     case (PC)
9       0:   Instruction = 24'b00_01000_0_001_001_0000000011;          //ADDI R1, R1,3
10      1:   Instruction = 24'b00_01000_0_011_011_0000000011;          //ADDI R3, R3,3
11      2:   Instruction = 24'b00_01011_0_011_001_0000000001;          //BEQ R3, R1, 1
12      //2:   Instruction = 24'b00_00101_0_000_011_001_0000000;           //CMP R1, R3
13      //3:   Instruction = 24'b10_00011_0_010_011_001_0000000;           //ADDNEQ R2,R3,R1
14      //4:   Instruction = 24'b00_00011_0_100_011_001_0000000;           //ADD R4, R3,R1
15      //2:   Instruction = 24'b00_01000_0_011_011_0000000110;         //ADDI R3, R3,7
16      //2:   Instruction = 24'b00_01100_00000000000000111;
17
18
19
20      default: Instruction = 24'b0; // NOOP
21    endcase
22  end
23  endmodule
```

Figure 6.3: Test Case 2

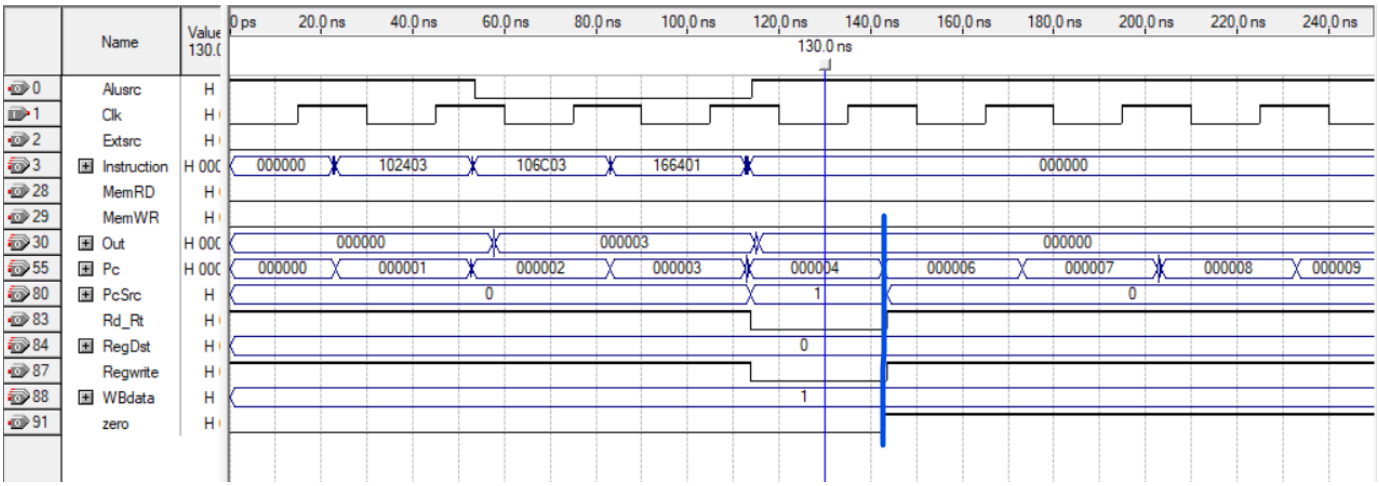

Figure 6.4: BEQ Output

As shown in Figure 6.4 the BEQ instruction executed correctly and the value of the PC was changed according to the offset.

6.3. Jump

```
module InstructionMemory(
  input [23:0] PC,
  input clock,
  output reg [23:0] Instruction
);
always@(posedge clock)
begin
  case (PC)
    0:   Instruction = 24'b00_01000_0_001_001_0000000011;          //ADDI R1, R1,3
    1:   Instruction = 24'b00_01000_0_011_011_0000000011;          //ADDI R3, R3,3
    2:   Instruction = 24'b00_01100_00000000000000111;             // J 7
    //2:   Instruction = 24'b00_00101_0_000_011_001_0000000;           //CMP R1, R3
    //3:   Instruction = 24'b10_00011_0_010_011_001_0000000;           //ADDNEQ R2,R3,R1
    //4:   Instruction = 24'b00_00011_0_100_011_001_0000000;           //ADD R4, R3,R1
    //2:   Instruction = 24'b00_01000_0_011_011_0000000110;         //ADDI R3, R3,7



    default: Instruction = 24'b0; // NOOP
  endcase
end
endmodule
```

Figure 6.5: Test Case 3

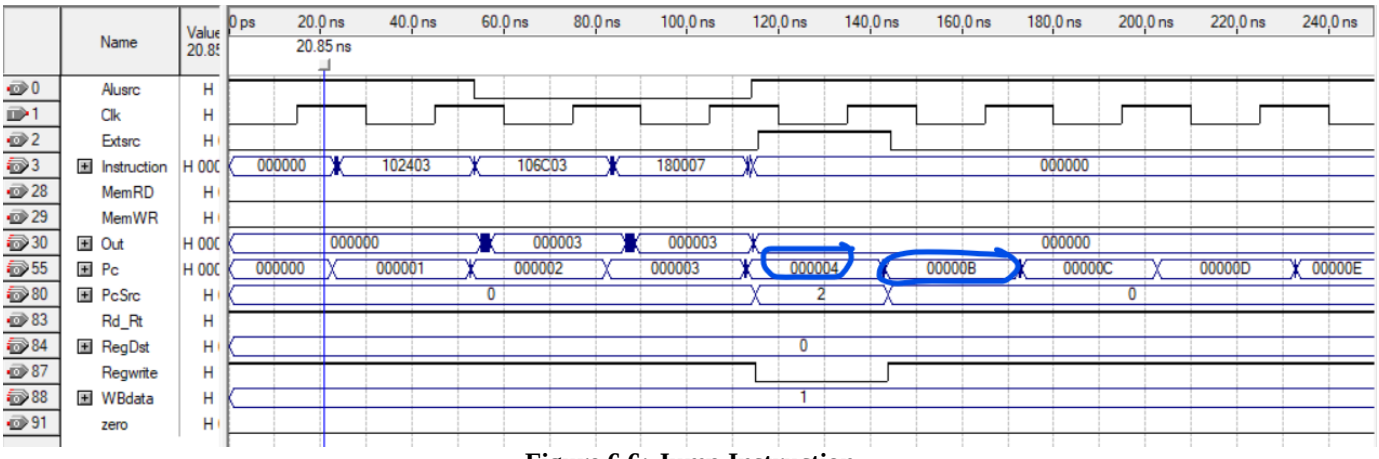

Figure 6.6: Jump Instruction

As shown in Figure 6.6, the unconditional jump executed correctly and the value of the PC was changed according to the immediate

6.4. SUBSF

```
module InstructionMemory(
  input [23:0] PC,
  input clock,
  output reg [23:0] Instruction
);
always@(posedge clock)
begin
  case (PC)
    0:   Instruction = 24'b00_01000_0_001_001_0000000011;          //ADDI R1, R1,3
    1:   Instruction = 24'b00_01000_0_011_011_0000000011;          //ADDI R3, R3,3
    2:   Instruction = 24'b00_00011_1_100_011_001_0000000;         //SUBSF R4, R3,R1
    //2:   Instruction = 24'b00_01100_00000000000000111;              // J 7
    //2:   Instruction = 24'b00_00101_0_000_011_001_0000000;          //CMP R1, R3
    //3:   Instruction = 24'b10_00011_0_010_011_001_0000000;          //ADDNEQ R2,R3,R1
    default: Instruction = 24'b0; // NOOP
  endcase
end
endmodule
```

Figure 6.7: Test case 4

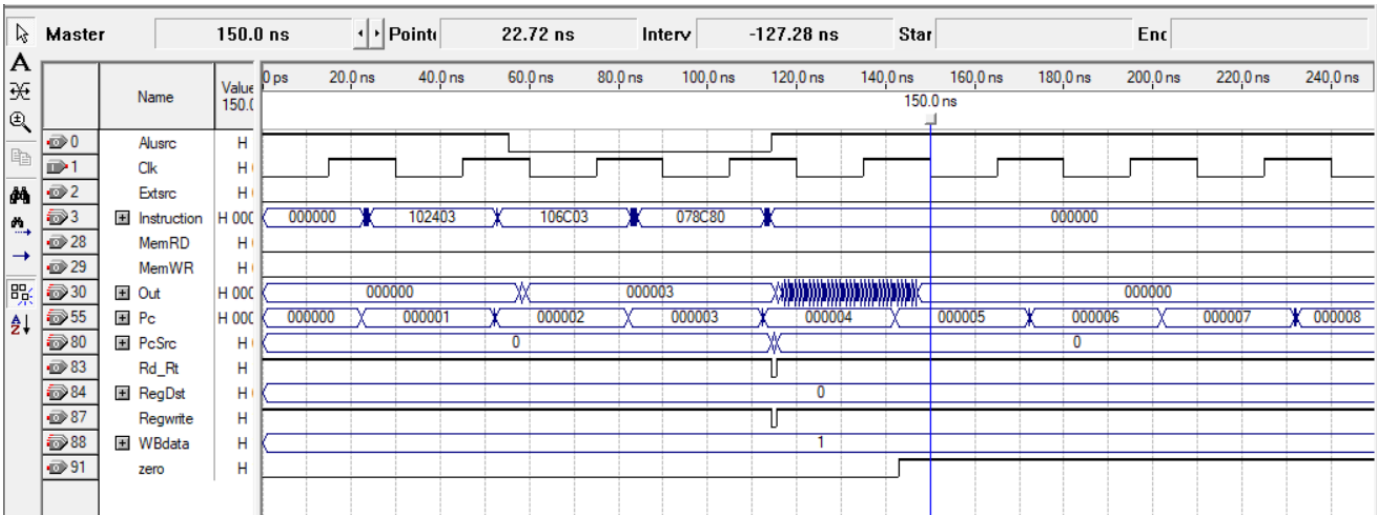

Figure 6.8: SUBSF Simulation

As shown in Figure 6.8 that adding the suffix has effected the zero flag and changed it to 1 since both registers subtraction is zero.

7. Conclusion

In conclusion, building the datapath and it components were learnt, adding the new instructions and conditions were also learnt, and deriving the control unit signals using the truth table was learnt, and finally after connecting all the components in the datapath the results were correct for all instructions.

8. Appendix

Most of the codes needed for the datapath were already implemented in this link and were fully understood and used. The codes taken were edited and some codes were added.

[1] https://github.com/gremerritt/multicycle-processor

The datapath concept was taken from previous semester and was updated.