1 CONTENTS

2 TOP-DOWN VIEW

To better understand Multi Cycle processor we should see it with a Top-Down View like the picture below:

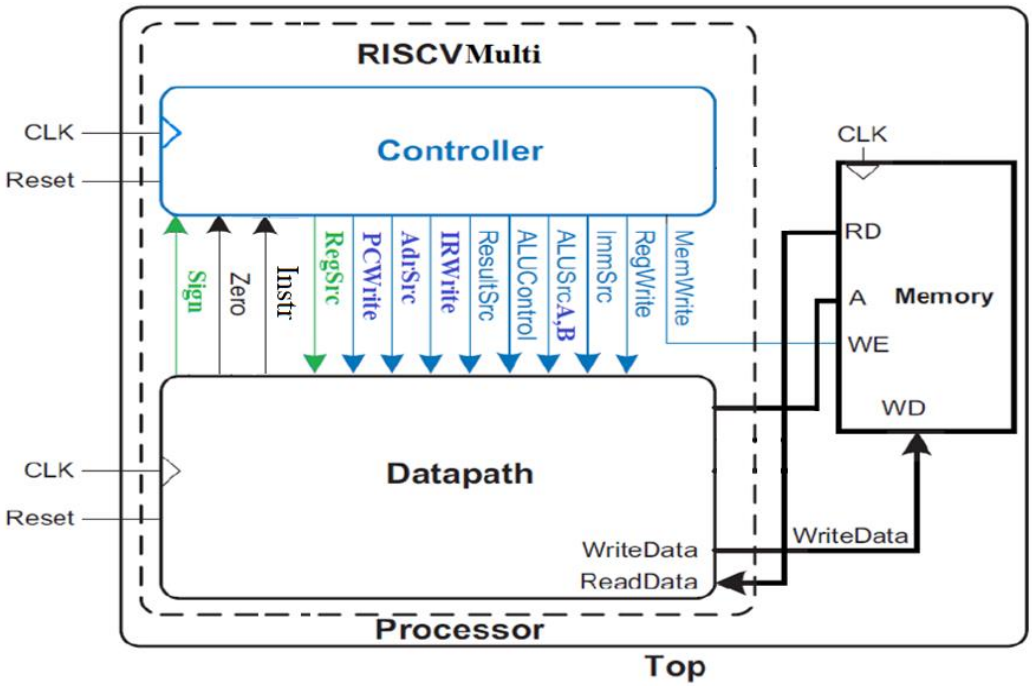


Figure 1- Top-Down View

The green signals will be added in the extended version!

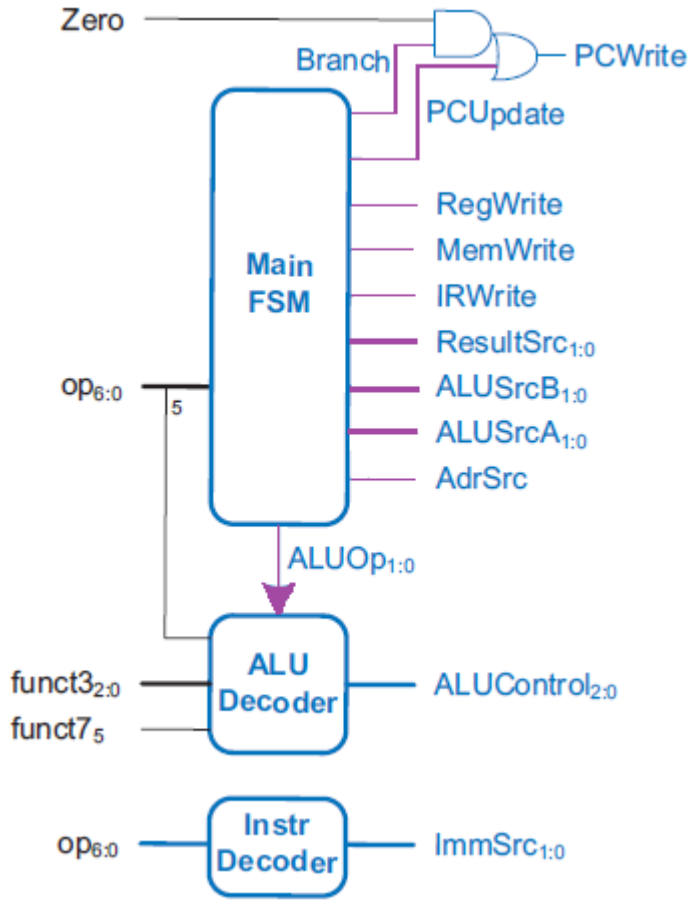
Now we know there are 3 parts to design. Controller, Data Path and Memory. In this project we are going to use a very simple memory with no cache, no virtual memory and virtual addresses, no page table and …, it needs just 4 lines in Verilog:

```
module mem(
        input logic clk, we,
        input logic [31:0] Adr, WD,
        output logic [31:0] RD
);

  logic [31:0] RAM[63:0];

  assign RD = RAM[Adr[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[Adr[31:2]] <= WD;
```

We begin with designing our Control Unit, then we will complete our data path and at last we must connect these two to make the RISC-V section in the above picture, and at last we will connect our RISC-V processor to memory.

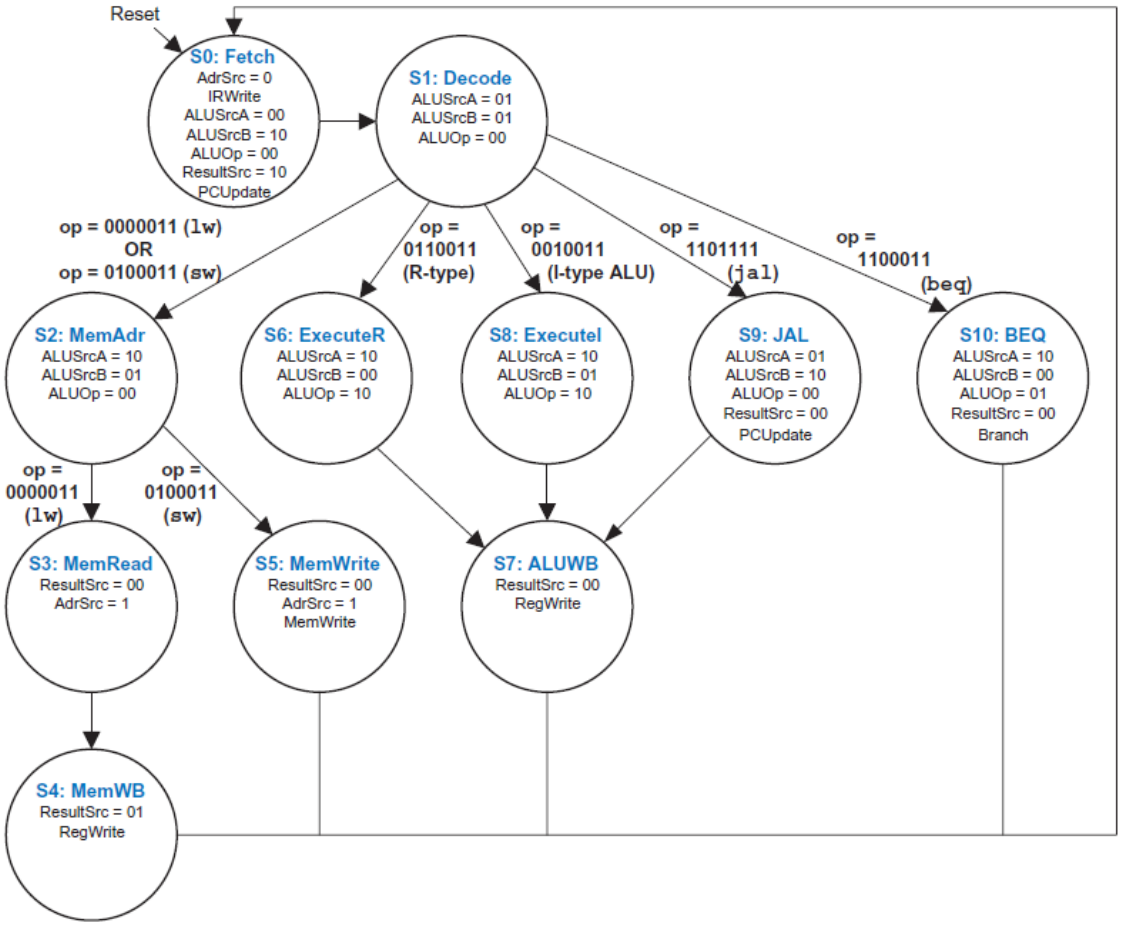
3 CONTROL UNIT

According to Figure 7.28 of the book, the control unit has a Finite State Machine, an ALU decoder and an instruction decoder. We are going to build all these parts!



3.1 FINITE STATE MACHINE

Our FSM has only one input (op) and you can see the outputs in the above picture (Branch, IRWrite, …). This state machine has 11 states which is shown in Figure 7.45:



We must declare an “enum” with 11 states. The first state is S0 (Fetch) and in each rising edge of the clock, our current state will change to our next state.

```
typedef enum logic[3:0]  {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9 ,S10} States ;
States now, next ;

always_ff @(posedge clk, posedge reset) begin
        if(reset) now <= S0;
        else      now <= next ;
end
```

How to find our next state? In Figure 7.45 this question is answered. For instance, if we’re in state S1, with op = 0000011, our next state is S2.

At first, we declare a 14-bit signal called FSMControls which handles every output in Figure 7.28 which is colored in orchid:

```
assign {ALUOp, ALUSrcA, ALUSrcB, ResultSrc,
        AdrSrc, IRWrite, PCUpdate,
        RegWrite, MemWrite, Branch} = FSMControls ;
```

The PCWrite signal is equal to: `assign PCWrite = ( Zero & Branch ) | PCUpdate ;`

According to Figure 7.45, we can determine the next state and our control signals based on our current state and the 7-bit "op" signal which is one of the inputs of our FSM!

For simplicity, each Don't Care signal was converted to 0. (Commented FSMControls are the real ones, the lines beneath them are converted ones!)

```
always_comb begin
        case(now)          // Current State

S0: begin
        next = S1 ;  // Decode
        //Op, SrcA, B, Res, Adr, IR, PCU, Reg, Mem, Br
        FSMControls = 14'b00_00_10_10_0_1_1_0_0_0;
end

S1: begin
        case(op)
                7'b1100011: next = S10 ; //BEQ
                7'b1101111: next = S9 ; //JAL
                7'b0010011: next = S8; //ExecuteI
                7'b0110011: next = S6; //ExecuteR
                7'b0000011: next = S2; //MemAdr
                7'b0100011: next = S2;  //MemAdr
                default:    next = S1; //MemAdr
        endcase
        //FSMControls = 14'b00_01_01_xx_x_0_0_0_0_0;
          FSMControls = 14'b00_01_01_00_0_0_0_0_0_0;
end

S2: begin
        case(op)
                7'b0000011: next = S3; //MemRead
                7'b0100011: next = S5; //MemWrite
                default:    next = S2;
        endcase
        //FSMControls = 14'b00_10_01_xx_x_0_0_0_0_0;
          FSMControls = 14'b00_10_01_00_0_0_0_0_0_0;
end
```

3.2 ALU Decoder

We will build our ALU Decoder using the Table 7.3 of the book:

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

Whenever each of the following signals (ALUOp, funct3, opb5func7b5) changes, we need to check if our ALUControl signal also changes or not.

ALUControl's logic is based on the Table 7.3

```
logic [1:0] opb5func7b5;
assign opb5func7b5 = {op[5], funct7b5} ;

always @(ALUOp, funct3, opb5func7b5) begin
        case(ALUOp)
                2'b00: ALUControl = 3'b000; //ADD
                2'b01: ALUControl = 3'b001; //SUB
                2'b10: case(funct3)          // R-type, I-type
                        3'b000: begin
                                        if(opb5func7b5 == 2'b11)
                                /*SUB*/         ALUControl = 3'b001;
                                        else
                                /*ADD*/         ALUControl = 3'b000;
                                end
                        3'b010: ALUControl = 3'b101; //slt
                        3'b110: ALUControl = 3'b011; //or
                        3'b111: ALUControl = 3'b010; //and
                        endcase
        endcase
end
```

3.3 Instruction Decoder

The Instruction Decoder's logic is also available in the book's Table 7.4:

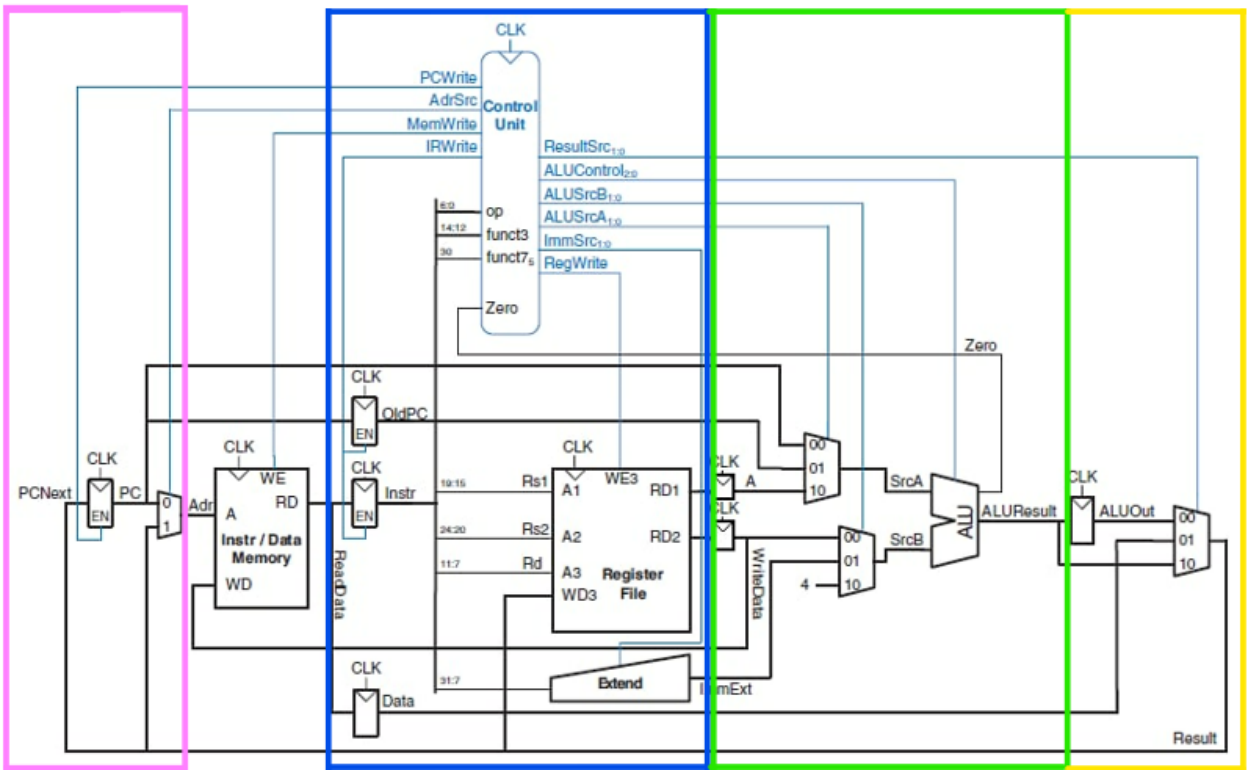
Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
lw	0000011	1	00	1	0	1	0	00
sw	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
beq	1100011	0	10	0	0	x	1	01
addi	0010011	1	00	1	0	0	0	10

And here is it's SystemVerilog code:

```
always @(op) begin
        case(op)
                7'b0000011:   ImmSrc = 2'b00;
                7'b0100011:   ImmSrc = 2'b01;
                7'b0110011: //ImmSrc = 2'bxx;
                              ImmSrc = 2'b00;
                7'b1100011:   ImmSrc = 2'b10;
                7'b0010011:   ImmSrc = 2'b00;
                7'b1101111:   ImmSrc = 2'b11;
                default:      ImmSrc = 2'bxx;
        endcase
end
```

4 DATAPATH

Here is the schematic of our Multi cycle's data-path copied and modified from the book:




```
/*Register with Enable Pin*/
/*Register without EN pin*/
```

```
//Pink
EnReg PCHolder(clk, reset, PCWrite, PCNext, PC);
mux2  AdrDeterminer(PC, Result, AdrSrc, Adr);

//Blue
EnReg instrHolder(clk, reset, IRWrite, ReadData, instr);
EnReg OldPCHolder(clk, reset, IRWrite, PC, OldPC);
Register DataHolder(clk, reset, ReadData, Data);
RegFile rf(clk, RegWrite, instr[19:15], instr[24:20], instr[11:7], Result,
           RD1, RD2, x1, x2, x3, x4, x5, x7, x9);
ExtendChip ex(instr, ImmSrc, ImmExt);

//Green
Register AHolder(clk, reset, RD1, A);
Register BHolder(clk, reset, RD2, B);
mux3 SrcADeterminer(PC, OldPC, A, ALUSrcA, SrcA);
mux3 srcBDeterminer(B, ImmExt, 32'd4, ALUSrcB, SrcB);
ALU alu(SrcA, SrcB, ALUControl, ALUResult, Zero);

//Yellow
Register ALUOutHolder(clk, reset, ALUResult, ALUOut);
mux3 res(ALUOut, Data, ALUResult, ResultSrc, Result);
```

4.1 REGISTERS WITH AND WITHOUT ENABLE PIN

```
module EnReg #(parameter N=32)(
        input logic clk, reset, en,
        input logic [N:1] d,
        output logic [N:1] q
);
        always_ff @(posedge clk)
                if(reset)       q<=0;
                else if(en)     q<=d;
endmodule
```

```
module Register #(parameter N=32)(
        input logic clk, reset,
        input logic [N:1] d,
        output logic [N:1] q
);
        always_ff @(posedge clk)
                if(reset)       q<=0;
                else             q<=d;

endmodule
```

4.2 TWO AND THREE CHOICES MULTIPLEXERS

```
module mux2(
        input logic [31:0] c0, c1,
        input logic cntrl,
        output logic [31:0] chosen
);
        assign chosen = cntrl ? c1 : c0 ;
endmodule
```

```
module mux3(
        input logic [31:0] c0, c1, c2,
        input logic [1:0] cntrl,
        output logic [31:0] chosen
);
assign chosen = cntrl[1] ? c2 : cntrl[0] ? c1 : c0 ;
endmodule
```

4.3 REGISTER FILE

Registers 1 to 5, 7 and 9 are being monitored at all times!

Register 0 is hardwired to 0.

Reading register is asynchronous but Writing is synchronous therefore RD1 and RD2 are assigned but WD3 is only written in reg[A3] If the clock is in its rising edge and our write enable signal is high.

```
module RegFile(
        input logic clk, WE3,
        input logic [4:0] A1, A2, A3,
        input logic [31:0] WD3,
        output logic [31:0] RD1, RD2,
        output logic [31:0] x1, x2, x3,
         x4, x5, x7, x9
);
        reg [31:0] regs[31:0];

        // Zero register
        initial regs[0] = 0 ;

        always @(posedge clk)
                if(WE3) regs[A3] <= WD3 ;

        assign RD1 = regs[A1] ;
        assign RD2 = regs[A2] ;

        // Monitored registers :
        assign x1 = regs[1] ;
        assign x2 = regs[2] ;
        assign x3 = regs[3] ;
        assign x4 = regs[4] ;
        assign x5 = regs[5] ;
        assign x7 = regs[7] ;
        assign x9 = regs[9] ;
endmodule
```

4.4 EXTEND CHIP

```
module ExtendChip(
        input logic [31:0] instr,
        input logic [1:0]  ImmSrc,
        output logic [31:0] ImmExt
);
        always @(instr, ImmSrc)
                case(ImmSrc)
                   2'b00 : ImmExt = {{20{instr[31]}}, instr[31:20]} ; // I-type
                   2'b01 : ImmExt = {{20{instr[31]}}, instr[31:25], instr[11:7]} ; // S-type
                   2'b10 : ImmExt = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0} ; //B-type
                   2'b11 : ImmExt = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0} ; // J-type
                endcase
endmodule
```

The extend chip's logic was copied from Table 7.5 of the book:

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

4.5 ARITHMETIC LOGIC UNIT

```
module ALU(
        input logic [31:0] A, B,
        input logic [2:0]  Op,
        output logic [31:0] result,
        output logic Zero
);
        always @(A, B, Op)
                case(Op)
                        3'b000 : result = A + B ;
                        3'b001 : result = A - B ;
                        3'b101 : result = A < B ? 1 : 0 ;
                        3'b011 : result = A | B ;
                        3'b010 : result = A & B ;
                        default : result = 32'b0 ;
                endcase
        assign Zero = &(~result) ;
endmodule
```

Our zero signal is assigned to "&(~result)".

"&(~result)" is 1 if and only if result is 32'b0 (because then "~result" is 32'b1 and "&(~result)" will be 1 otherwise it's zero.). Operations are in synch with Table 7.3 of the book.

5 PREDICTION

Note that because we never left any of our control signals undecided and changed all signals with X value to 0, the Result bus which depends on the ResultSrc control signal is never X and always contains a value. For instance, in the Decode state we don't need the Result, thus the ResultSrc signal can be whatever it wants, thus we set it to be 2'b00 and Result bus will be equal to ALUOut. The same thing happens in the ExecuteI state. So, In the Decode and ExecuteI states where we don't need the value in Result bus it's still equal to ALUOut bus. In the Decode state, the ALUOut is holding the value calculated by the ALU in the previous state which is Fetch. Similarly, in the ExecuteI state, ALUOut bus is holding the value calculated by the ALU in its previous state (Decode) which is the Branch/ Jump target address (OldPc + imm).

In ExecuteR state, ImmSrc is don't care, thus we set it to be 2'b00. This means that the 3 most significant bytes of the coming R-type instruction will be Sign-Extended and then it will be added to the OldPC bus in the Decode state, thus the Result bus in ExecuteR state is equal to 3 most significant bytes of the instruction plus OldPC. Summing it up:

Decode state ➔ Result = PC+4, which was calculated in Fetch state

ExecuteI state ➔ Result = OldPC + imm, which was calculated in Decode state

ExecuteR state ➔ Result = OldPC + (12 MSBs of the instruction) which was calculated in Decode state

Now we begin to fill up the prediction table in the next page.

Step	PC	Instr	State	Result	Result Note
3	00	00500113	Fetch	4	PC+4
4	04	00500113	Decode	4	Still holding PC+4 of the previous state
5	04	00500113	ExecuteI	5	holding OldPC+imm of the previous state = 0 + 5
6	04	00500113	ALUWB	5	ALUout = x0 + 5 = 5
7	04	00500113	Fetch	8	PC+4
8	08	00C00193	Decode	8	Still holding PC+4 from Fetch state
9	08	00C00193	ExecuteI	16	Holding OldPC+imm from Decode state = 4 + 12 = 16
10	08	00C00193	ALUWB	12	ALUout = x0 + 12 = 12
11	08	00C00193	Fetch	12	PC+4
12	12	FF718393	Decode	12	Still holding PC+4 from Fetch state
13	12	FF718393	ExecuteI	-1	Holding BTA from Decode state = OldPC + imm = 8 + (-9) = -1
14	12	FF718393	ALUWB	3	ALUout = x3 + (-9) = 12 – 9 = 3
15	12	FF718393	Fetch	16	PC+4
16	16	0023E233	Decode	16	Still holding PC+4 from Fetch state
17	16	0023E233	ExecuteR	14	OldPC+imm = 12 + 0x002 = 14
18	16	0023E233	ALUWB	7	ALUout = x7 or x2 = 3 or 5 = 7
19	16	0023E233	Fetch	20	PC+4

6 SIMULATION

6.1 CONTROLLER'S TEST

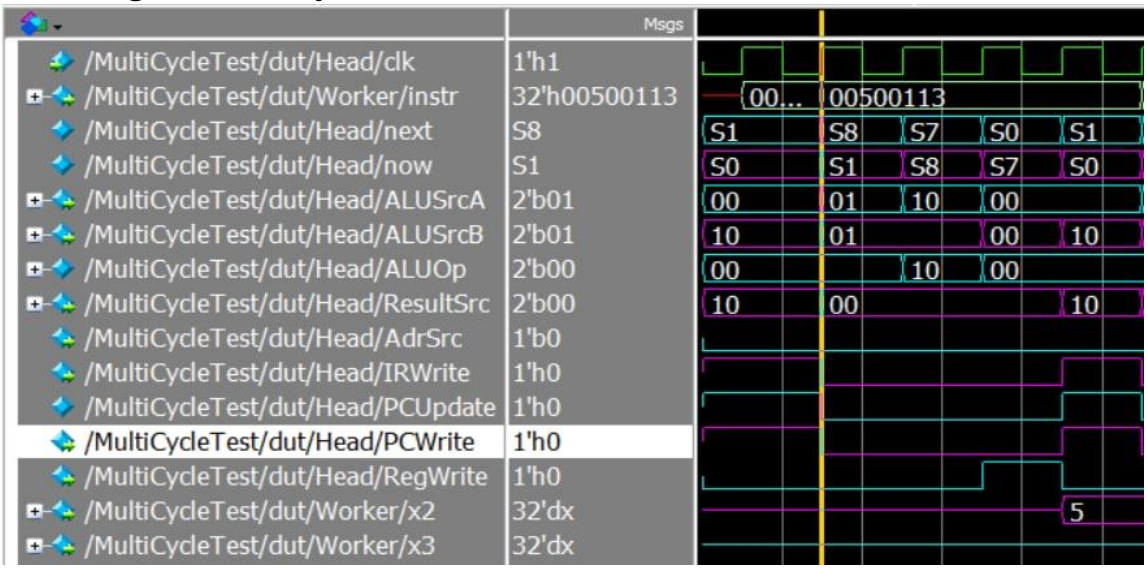
Here is the result of simulating all 47 tests in the controller.tv file on our controller module:

```
ModelSim> vsim -gui -voptargs=+acc work.testbench
# vsim -gui -voptargs="+acc" work.testbench
# Start time: 02:54:59 on Jul 02,2022
# ** Note: (vsim-8009) Loading existing optimized design _opt
# Loading sv_std.std
# Loading work.testbench(fast)
# Loading work.controller(fast)
VSIM 11> run -all
#         47 tests completed with          0 errors
# hash = 39
# ** Note: $stop    : D:/Uni/Term 4/Computer Architecture/Project/ModelSimVersion/testbench.sv(106)
#    Time: 485 ns  Iteration: 1  Instance: /testbench
# Break in Module testbench at D:/Uni/Term 4/Computer Architecture/Project/ModelSimVersion/testbench.sv line 106
```

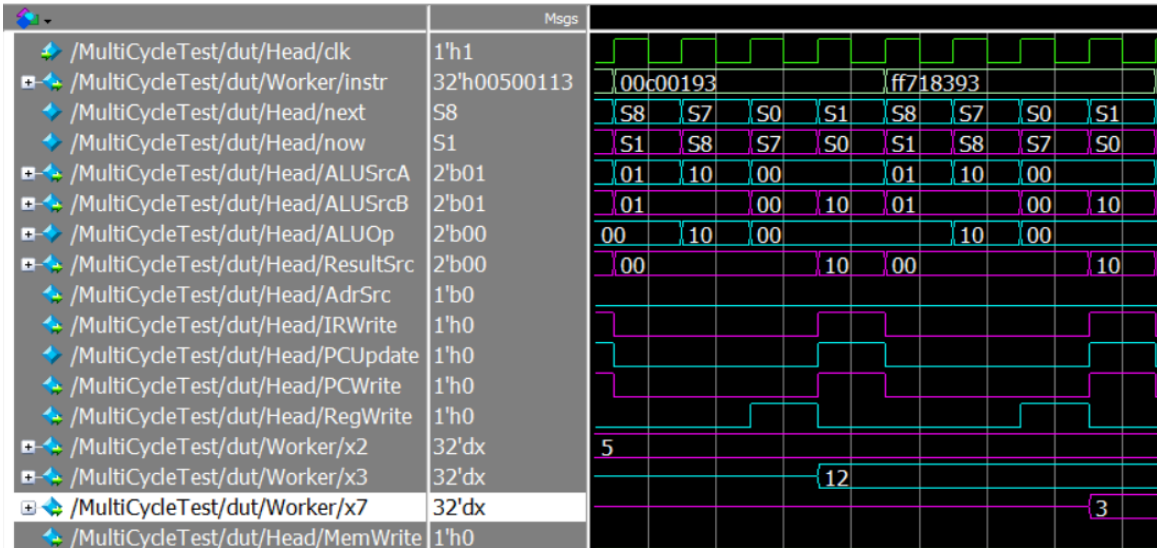
All tests were completed without any error, thus our controller is working perfectly fine.

6.2 PROCESSOR'S TEST

Giving the memory our riscvtest.txt file, here is the simulation's result:



When the first instruction 0x00500113 (addi x2, x0, 5) gets in, we move from S0 to S1(Decode) and ALUSrcA, ALUSrcB and ALUOp signals change to 2'b01, 2'b01 and 2'b00 respectively just like the FSM diagram. Then we move to S8 (ExecuteI) and ALUSrcA and ALUOp chang to 2'b10, again just like the diagram. In this state, ALU will finish calculating and we'll move to S7 (ALUWB), as you can see RegWrite signal becomes 1 and value 5 will be written in register x2.

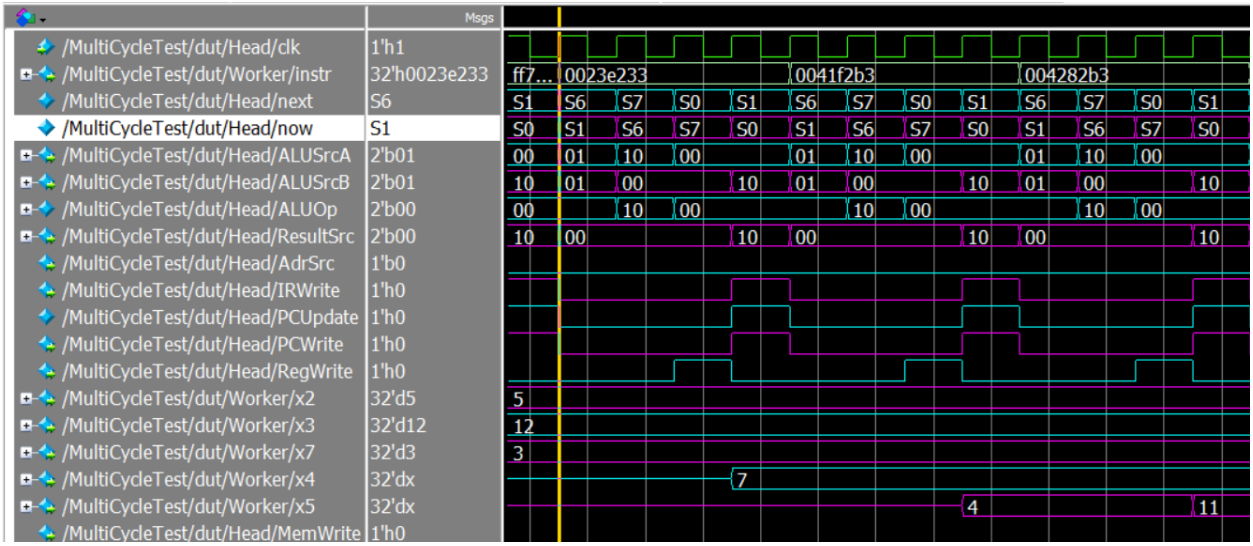


The next two instructions are also I-type like the first, thus they follow the same path and values 12 and 3 will be written into registers x3 and x7 respectively as we wanted to.

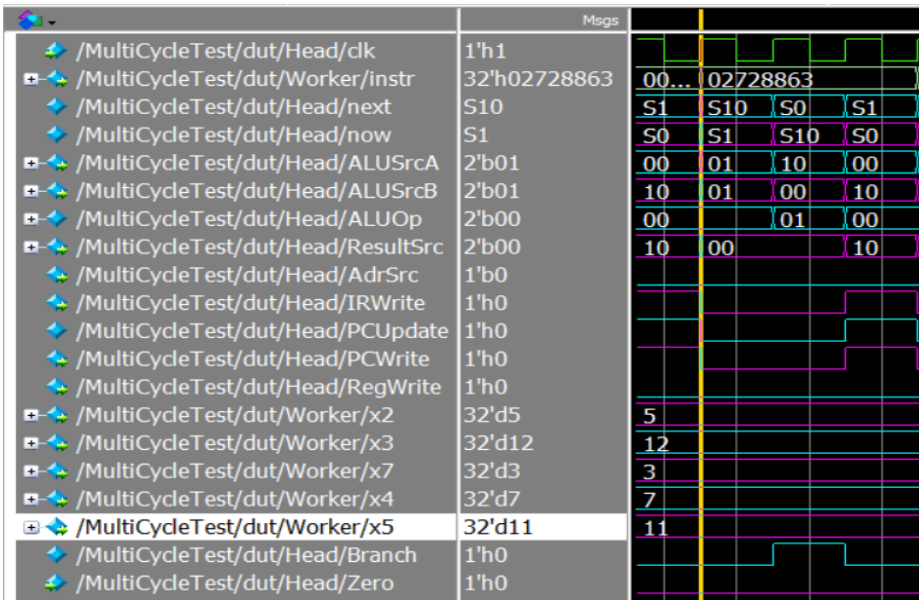
The next 3 instructions are R-type so after the Decode state they must move to S6(ExecuteR). Here are the instructions:

```
or   x4, x7, x2          # x4 = (3 OR 5) = 7
and  x5, x3, x4          # x5 = (12 AND 7) = 4
add  x5, x5, x4          # x5 = 4 + 7 = 11
```

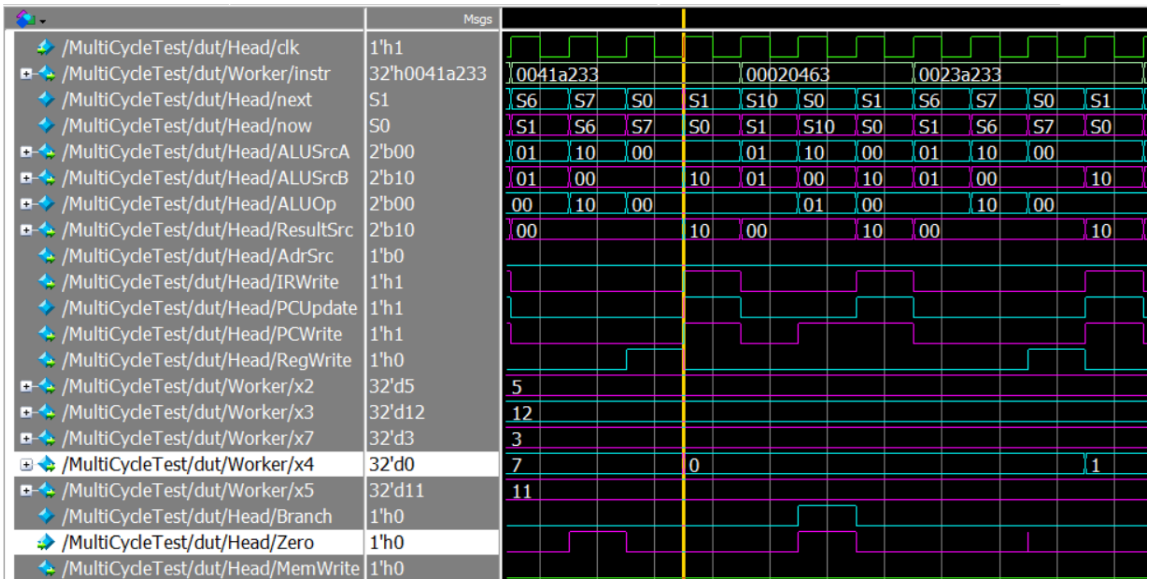
And this is the simulation result's:



Register x4 is holding 7, x5 first becomes 4 and then changes to 11 as we wanted. If you look at the current state signal (now), you realize that after S1 it now changes to S6. Next instruction is 0x02728863 (beq x5, x7, end). We know that x5 and x7 are not equal (11≠3), thus the branch should not be taken:

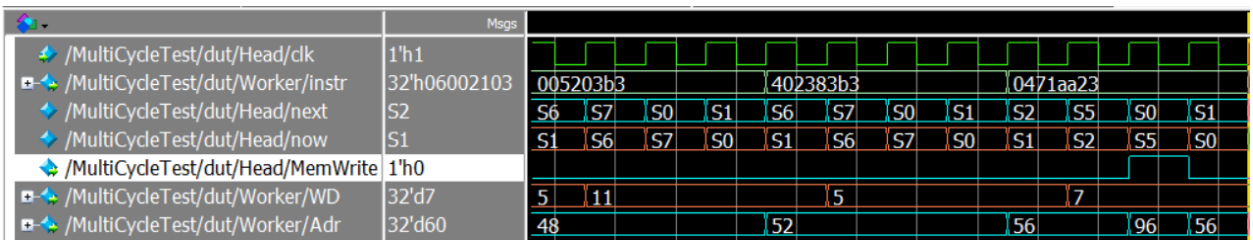


This time after S1, we moved to S10 (BEQ). In S10, branch signal becomes 1 and ALUOp changes to 2'b01. It means that ALU must perform subtraction but the Zero signal remains 0 because x5 and x7 are not equal and because of this, the PCWrite won't change in this state. Then we move to S0. This branch is not taken, thus the next instruction 0x0041A233 will get into data path (slt x4, x3, x4). This instruction will write 0 to x4:



After that, we've another branch instruction (0x0020563, beq x4, x0, around) which must be taken because (x4 = x0 = 0). If you look at the signals below this instruction in the picture, you can see that this time Zero and Branch signal are both 1 at the same time, thus the PCWrite register becomes one and BTA that we calculated in S1 will be written in PC register. Because of this taken branch the next instruction (0x00000293, addi x5, x0, 0) won't get into data path but instead 0x0023a233 (slt x4, x7, x2) gets in and writes 1 into x4.

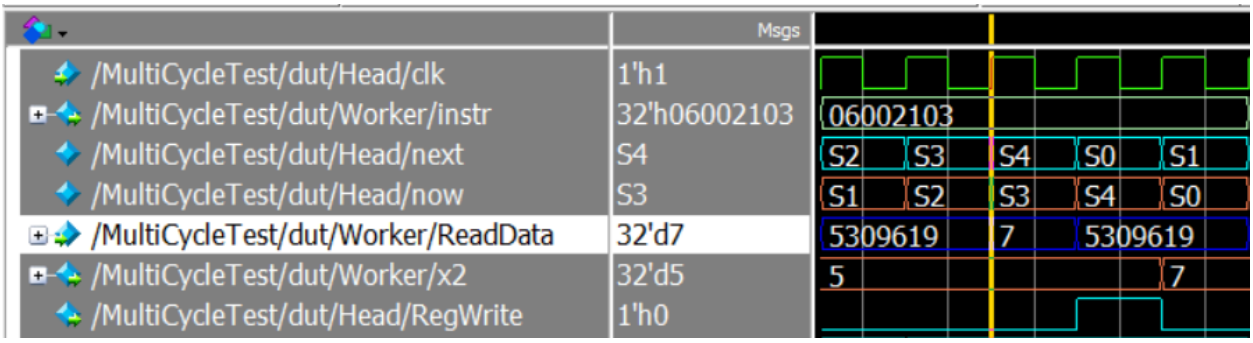
After this we've got some R-type instructions and one SW instruction. Let's take a look at what happens in SW:



0x0471AA23 wants to store value 7 in memory address 96. As it's obvious from the picture, after S1 we move to S5(MemWrite) and MemWrite signal becomes 1.

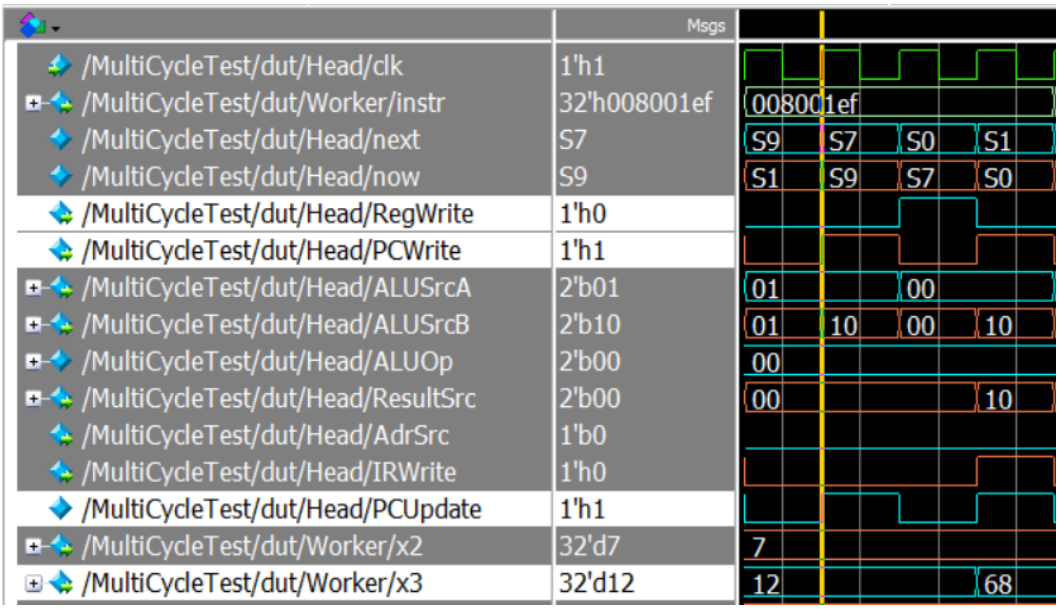
When MemWrite is high, WD is 7 and Adr is 96, it means that 7 will be written in mem[96] as desired.

Now we've got a LW instruction which takes 5 cycles to complete:



In LW, after S1 we move to S2 like SW instruction, then we move to S3 (MemRead) state and as you can see, value 7 is read from the memory. After S3 we move to S4 (MemWB) and in this state RegWrite signal must become 1 for the ReadData to be written in the destination register (x2). When the instruction completed, value 7 is being held by x2.

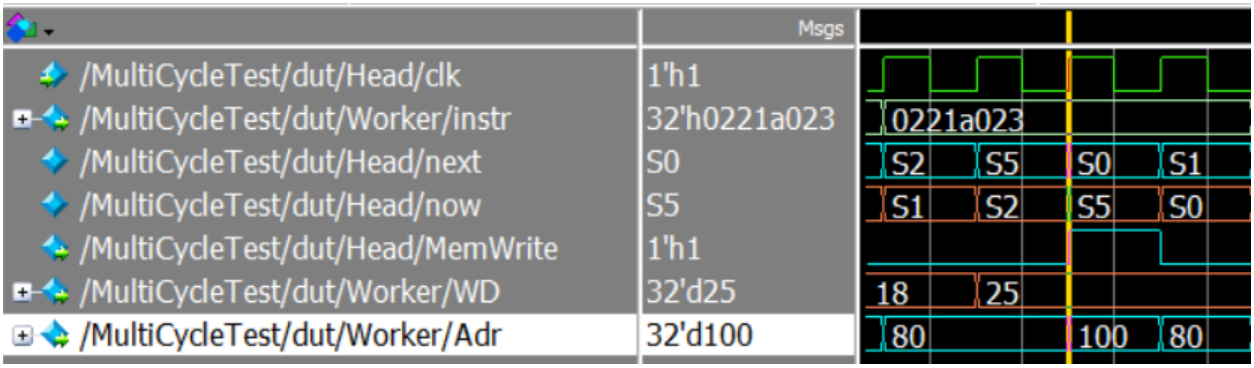
We now have a JAL instruction ahead (0x008001EF, JAL x3, end) which must save PC+4 = 0x44 = $(68)_{10}$ in register x3 and then skips everything between itself and label END:



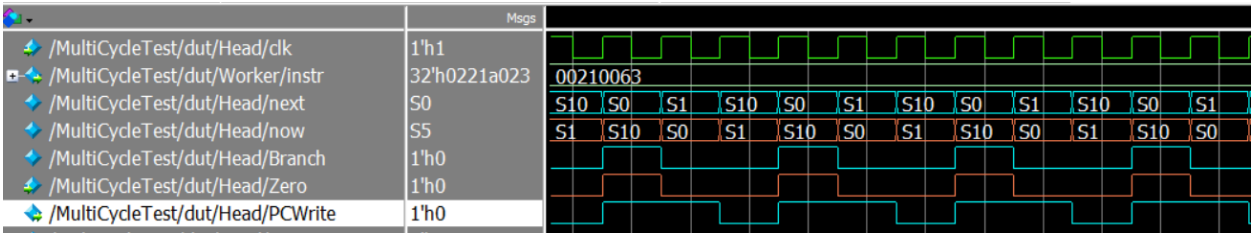
After S1 we move to S9(JAL), in this state JTA calculated in S1 will be written in PC register. As you can see PCUpdate and PCWrite signals become 1 in this state.

Then we move to S7 and write PC+4 in destination register (x3). Again, you can see that in S7 RegWrite signal becomes 1 and when the next clock hits, value 68 (0x44) will be written in x3.

At last, value 100 must be written in mem[100] and we've got an infinite loop:



In S5 Memwrite becomes 1, WD equals 25 and Adr is 100, thus 25 is being written in mem[100] in this state.



Program stays in instruction 0x00210063 forever because in S10 Zero and Branch signals become 1 together, thus PCWrite signal gets high and branch is taken.

7 SYNTHESIS

Now we are going to Synthesize and Simulate our processor in Quartus environment.

First, we create a project and add our Verilog codes to this project and Synthesize it.

Here is the Flow summary which contains general information about the processor

7.1 IMPLEMENTING

Implementation on the selected board:

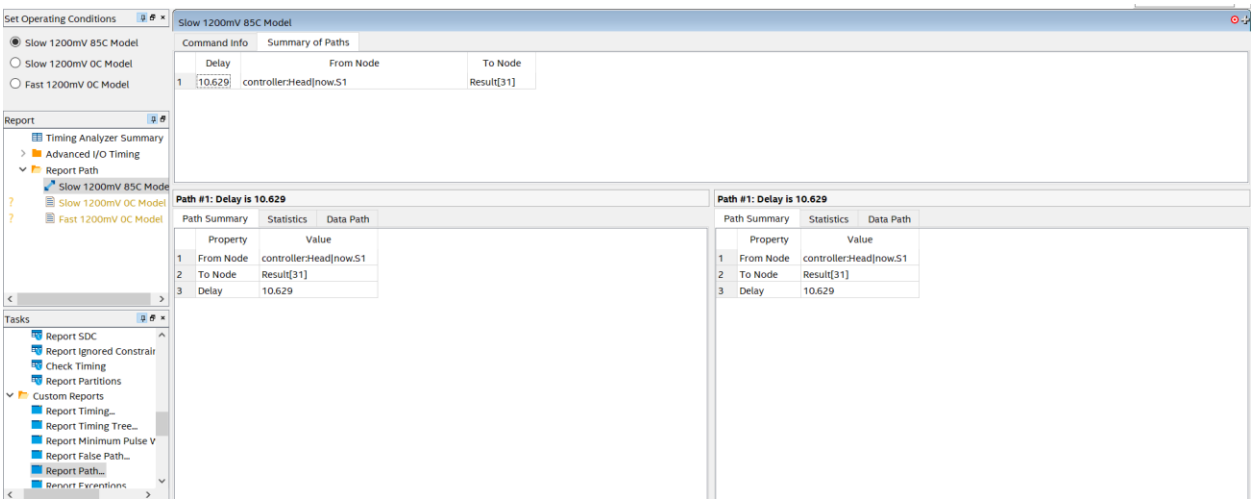
Flow Summary

<<Filter>>

Flow Status	Successful - Sun Jul 03 19:13:01 2022
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	MultiCycle
Top-level Entity Name	MultiCycle
Family	Cyclone IV E
Total logic elements	4,398 / 6,272 (70 %)
Total registers	2371
Total pins	131 / 180 (73 %)
Total virtual pins	0
Total memory bits	2,048 / 276,480 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)
Device	EP4CE6F17C6
Timing Models	Final

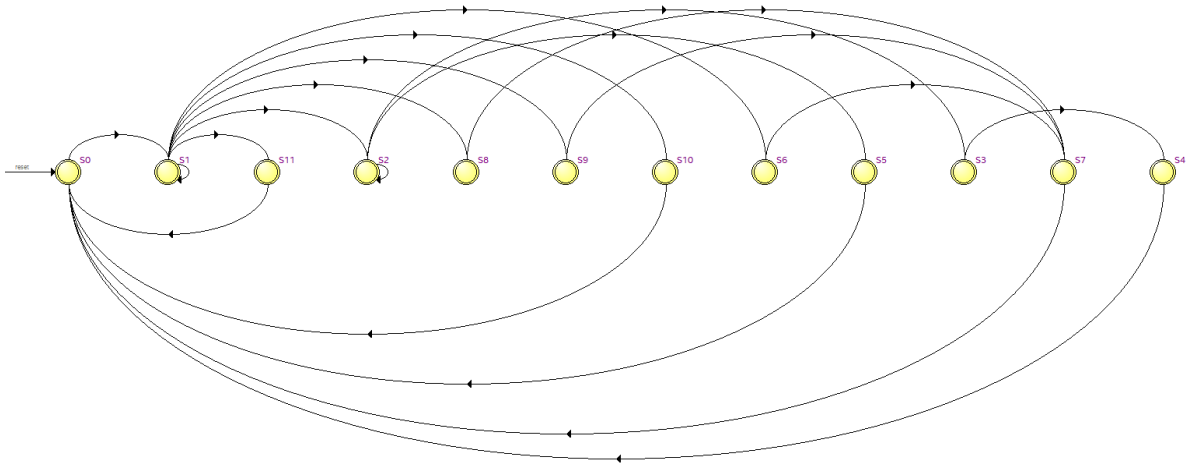
Then we are going to get timing analysis report for our processor which contains delay for critical path in our design. The following is the timing report:

7.2 Time Report

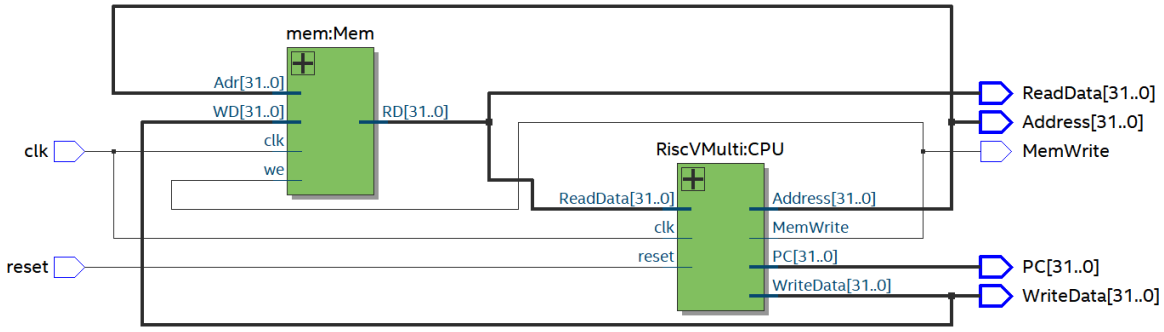


Two other useful reports we can get from quartus are RTL viewer and State Machine viewer that are basically schematic views for our processor (like top-down view) and our FSM.

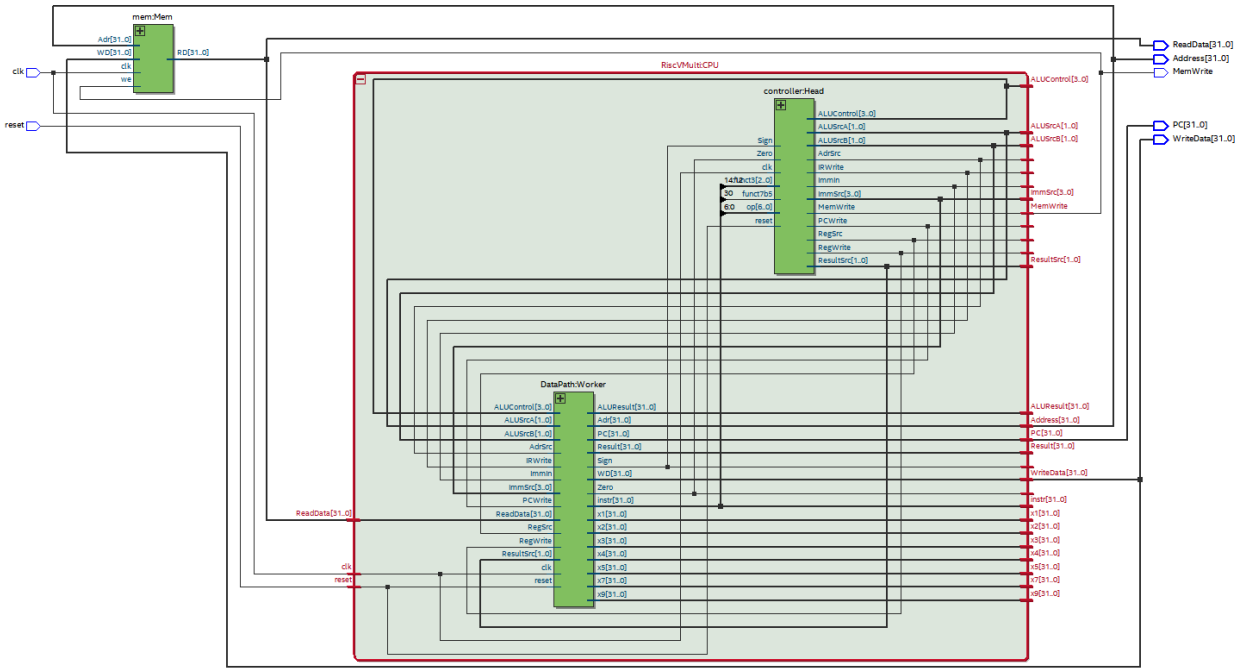
7.3 State Machine Viewer



7.4 RTL VIEWER



Inside the RiscVMulti:



Everything is just like the picture in Figure 1.

8 EXTRA-CREDIT QUESTIONS

In this section we are going to update our Multi Cycle processor to handle more instructions. Then we will implement it on FPGA.

8.1 ADDING NEW INSTRUCTIONS

The easiest instructions we can add to our data path are “XOR” and “XORI”:

8.1.1 XOR and XORI

“XOR” is an R-type instruction and “XORI” is an I-type one. We’ve already handled these types of instructions in our FSM, they all follow the same path in our data path, we just need to add the ability to XOR data in our ALU and choose an ALUControl signal for this operation in our ALU Decoder such that whenever these instructions were read from the memory, the correct signal goes into the ALU and makes it to perform XOR. Both these instructions will be handled in states S6(ExecuteR) and S8(ExecuteI) of the state diagram from the book (Figure 7.45).

Here is the Op, funct3 and funct7 of XOR which was copied from the book:

Op	funct3	funct7		
0110011 (51)	100	0000000	R	xor rd, rs1, rs2

We need to update our ALU to handle “XOR”:

We assigned Op = 3’b100 to “XOR”

```
case(Op)
        3'b000 : result = A + B ;
        3'b001 : result = A - B ;
        3'b101 : result = A < B ? 1 : 0 ;
        3'b011 : result = A | B ;
        3'b010 : result = A & B ;
/*XOR*/ 3'b100 : result = A ^ B ;
        default : result = 32'b0 ;
endcase
```

ALU Decoder must also be updated: (According to its funct3, Op and funct7)

```
always @(ALUOp, funct3, opb5func7b5) begin
        case(ALUOp)
                2'b00: ALUControl = 3'b000; //ADD
                2'b01: ALUControl = 3'b001; //SUB
                2'b10: case(funct3)         // R-type, I-type
                        3'b000: begin
                                        if(opb5func7b5 == 2'b11)
                                /*SUB*/         ALUControl = 3'b001;
                                        else
                                /*ADD*/         ALUControl = 3'b000;
                                end
                        3'b010: ALUControl = 3'b101; //slt
                        3'b110: ALUControl = 3'b011; //or
                        3'b111: ALUControl = 3'b010; //and
                        3'b100: ALUControl = 3'b100; //xor
                        endcase

        endcase
end
```

We must write a test bench first to see if our extended processor works correctly:

addi x1, x0, -1	1111_1111_1111_0000_0000_0000_1001_0011 =	0xFFF00093
addi x2, x0, 2	0000_0000_0010_0000_0000_0001_0001_0011 =	0x00200113
xor x3, x2, x1	0000_0000_0001_0001_0100_0001_1011_0011 =	0x001141B3
xori x4, x2, 7	0000_0000_0111_0001_0100_0010_0001_0011 =	0x00714213

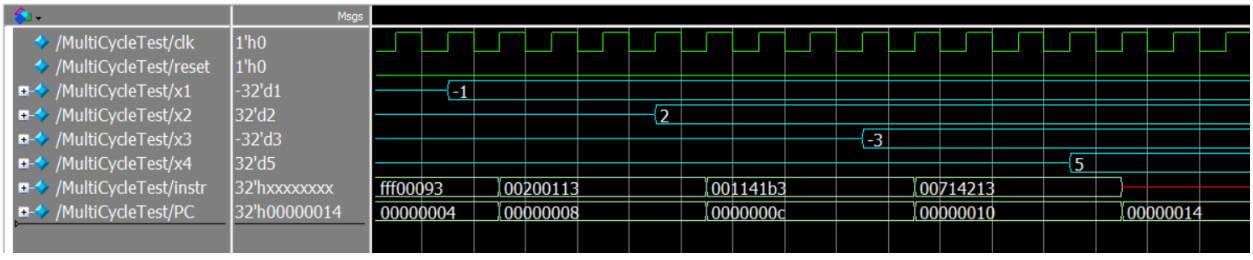
After running the above testbench, final value in x3 must be:

$0xFFFFFFFF \wedge 0x00000002 = 0xFFFFFFFD = (-3)_{10}$

And x4 must be:

$0x00000002 \wedge 0x00000007 = 0010 \wedge 0111 = 0101 = (5)_{10}$

And here is the simulation's result:



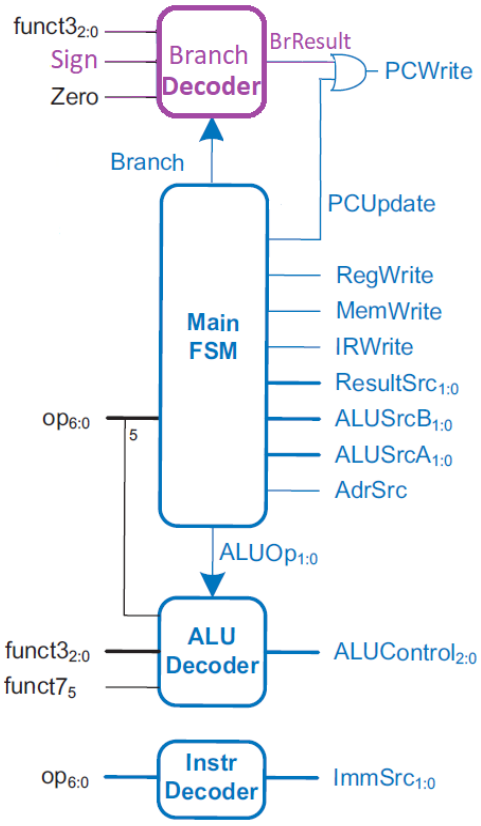
Next easiest instructions to add are "bne", "blt" and "bge":

8.1.2 BNE, BLT, BGE

To handle these instructions, my solution is to add a branch decoder to the control unit. Our FSM must give the decoder a 2-bit branch signal and the branch decoder will tell us if the branch occurred or not. To handle "blt" and "bge" we need to add Sign Flag to our ALU. Here is the Table for this decoder:

Instruction	funct3	Zero	Sign	Branch	BrResult
beq	000	1	X	1	1
		0	X		0
bne	001	0	X		1
		1	X		0
blt	100	X	1		1
		X	0		0
bge	101	X	0		1
		X	1		0
Non-Branch	X	X	X	0	0

Here is the new Control Unit ➔



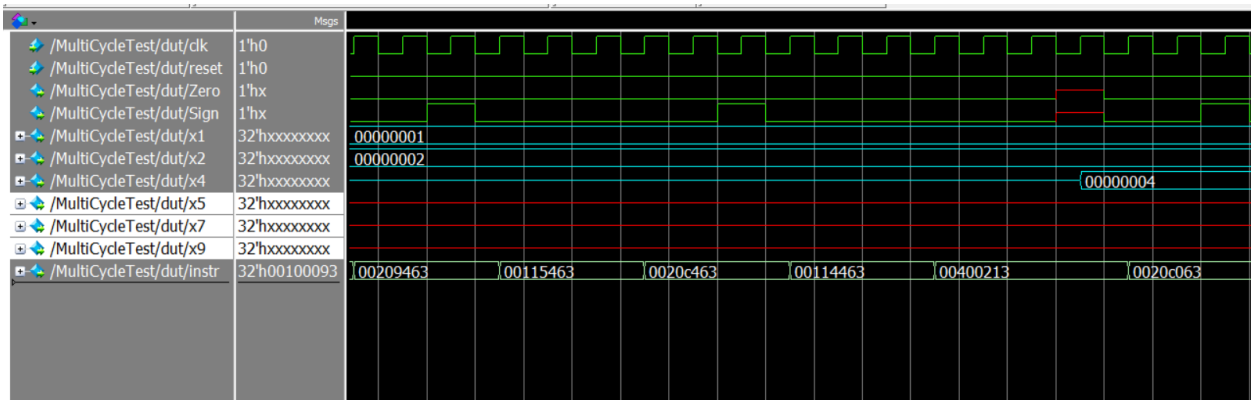
The FSM shown in Figure 7.45 can handle all branches in state "S10", because the result of all these branches depends on the outcome of the ALU which performs subtraction on the two given registers, thus we can still handle these new 3 instructions using the same old FSM with 11 states and there's no need to add another state!

And here is the test bench I wrote:

```
main:
        addi x1, x0, 1  // x1 = 1                00100093
        addi x2, x0, 2  // x2 = 2                00200113
        bne x1, x2, L1  // should BE taken       00209463
        addi x9, x0, 9  // should NOT happen     00900493
L1:
        bge x2, x1, L2  // should BE taken       00115463
        addi x7, x0, 7  // should NOT happen     00700393
L2:
        blt x1, x2, L3  // should BE taken       0020C463
        addi x5, x0, 5  // should Not happen     00500293
L3:
        blt x2, x1, end // should NOT BE taken   00114463
        addi x4, x0, 4  // MUST HAPPEN, x4 = 4   00400213
end:
        blt x1, x2, end // FOREVER LOOP          0020C063
```

When these instructions were performed, the final value of registers x1, x2 and x4 should be 1, 2 and 4 respectively. Registers x9, x7 and x5 must not change because the branches before them were taken! Instructions 0x00900493, 0x00700393 and 0x00500293 must not be feed to the processor.

Simulation's results:



As you can see, Registers x5, x7 and x9 remained intact because the instructions I named in the last paragraph did not get into data path as expected. But the last branch was not taken (again, as expected) and value 0x04 was written into register x4, thus the processor is working fine and it can now handle "bne", "blt" and "bge".

We can also add shift instructions to our processor:

8.1.3 Shift Instructions (SLL, SRL, SRA, SLLI, SRLI, SRAI)

To do this we need to update our ALU and give it the ability to shift numbers:

```
module ALU(
        input logic signed [31:0] A, B,
        input logic [3:0]  Op,
        output logic [31:0] result,
        output logic Zero, Sign
);
        always @(A, B, Op)
                case(Op)
                        4'b0000 : result = A + B ;      //0
                        4'b0001 : result = A - B ;      //1
                        4'b0101 : result = A < B ? 1 : 0 ; //5
                        4'b0011 : result = A | B ;      //3
                        4'b0010 : result = A & B ;      //2
                /*XOR*/ 4'b0100 : result = A ^ B ;      //4
                /*SLL*/ 4'b0110 : result = A << B;      //6
                /*SLA*/ 4'b0111 : result = A <<< B;     //7
                /*SRL*/ 4'b1000 : result = A >> B;      //8
                /*SRA*/ 4'b1001 : result = A >>> B;     //9
                        default : result = 32'b0 ;
                endcase
        assign Zero = &(~result) ;
        assign Sign = result[31] ;
endmodule
```

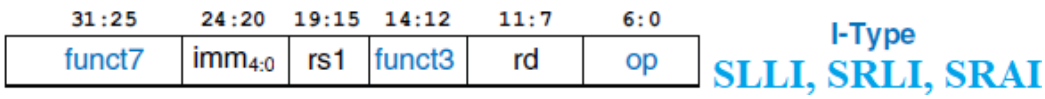
For the shift right arithmetic Operand (>>>) to work correctly, we must change our inputs (A, B) types to "Input Logic Signed"

Opcodes 6 to 9 are given to SLL, SLA, SRL and SRA respectively. Because now our ALU handles more than 8 operations, the Op signal now needs to have at least 4 bits, thus we now need to change our ALUControl signal from 3-bits to 4-bits in every module.

Now, The ALU Decoder in the control unit must also be updated. We update the ALU Decoder according to the RISC-V instruction set at the end of the book:

```
always @(ALUOp, funct3, opb5func7b5, funct7b5) begin
        case(ALUOp)
                2'b00: ALUControl = 4'b0000; //ADD
                2'b01: ALUControl = 4'b0001; //SUB
                2'b10: case(funct3)          // R-type, I-type
                        3'b000: begin
                                        if(opb5func7b5 == 2'b11)
                                /*SUB*/         ALUControl = 4'b0001;
                                        else
                                /*ADD*/         ALUControl = 4'b0000;
                                end
                        3'b010: ALUControl = 4'b0101; //slt
                        3'b110: ALUControl = 4'b0011; //or
                        3'b111: ALUControl = 4'b0010; //and
                        3'b100: ALUControl = 4'b0100; //xor
                        3'b001: ALUControl = 4'b0110; //SLL
                        3'b101: ALUControl = funct7b5 ?
                        /*SRA*/ 4'b1001 : 4'b1000 /*SRL*/ ;
                        endcase
        endcase
end
```

As you know, a 32-bit number can only be shifted 31 times (right or left). When a 32-bit number is shifted 32 times, only zeros remain, thus if a number X was shifted more than 31 times, we know for sure that the result is zero. Because of this fact, shifting more than 31 times is not allowed in RISC-V and we only need 5 bits for the second operand of the shift instructions, thus SLLI, SRLI, SRAI are I-type instruction with a 5-bit immediate:



For these instructions, a new type of extending must be added to our Extend Chip, because now we only need to extend bits 20 to 24 of these instructions. Also, the second operand of shift operation can not be a negative number, thus for shifting we must Zero-Extend instead of Sign-Extend.

```
always @(instr, ImmSrc)
       case(ImmSrc)
          3'b000 : ImmExt = {{20{instr[31]}}, instr[31:20]} ; // I-type
          3'b001 : ImmExt = {{20{instr[31]}}, instr[31:25], instr[11:7]} ; // S-type
          3'b010 : ImmExt = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0} ; //B-type
          3'b011 : ImmExt = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0} ; // J-type
          3'b100 : ImmExt = {27'b0, instr[24:20]} ; // I-type (Shift Instructions)
       endcase
```

The instruction Decoder inside the ALU must also be updated to notify Extend Chip which extending type it must use now. If SLLI has a funct3 of 3'b001 and SRAI and SRLI have a funct3 of 3'b101, thus if our instruction was I-type and its funct3 was equal to 1 or 5 we perform the new extend type on its immediate:

```
always @(op, fucnt3) begin
  case(op)
       7'b0000011:   ImmSrc = 3'b000;
       7'b0100011:   ImmSrc = 3'b001;
       7'b0110011: //ImmSrc = 3'bxxx;
                     ImmSrc = 3'b000;
       7'b1100011:   ImmSrc = 3'b010;
       7'b0010011: /*I-type*/
                     begin
                       if(funct3 == 3'b001 or //SLLI
                              funct3 == 3'b101) //SRAI, SRLI
                                       ImmSrc = 3'b100;
                       else
                                       ImmSrc = 3'b000; //Others
                     end
       7'b1101111:   ImmSrc = 3'b011;
       default    : //ImmSrc = 3'bxxx;
                     ImmSrc = 3'b000;
  endcase
end
```

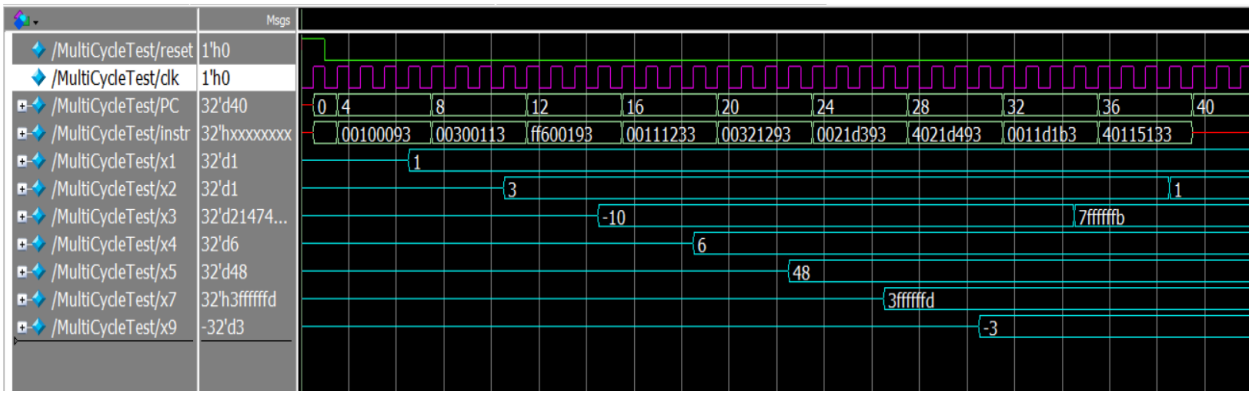
Because the extend chip has now 5 types of extending, our ImmSrc now mus at least have 3 bits, thus we change ImmSrc's bits in all modules from 2 to 3.

To test our Shift-Extended Multi Cycle processor, I wrote this test bench:

```
PC      Machine Code    Original Code
0x0     0x00100093      addi x1, x0, 1    # x1 = 1
0x4     0x00300113      addi x2, x0, 3    # x2 = 3
0x8     0xFF600193      addi x3, x0, -10  # x3 = -10
0xc     0x00111233      sll x4, x2, x1 # x4 = 3 << 1 = 3*(2^1) = 6
0x10    0x00321293      slli x5, x4, 3 # x5 = 3 << 3 = 6*(2^3) = 48
0x14    0x0021D393      srli x7, x3, 2 # x7 = 0xFFFFFFF6 >> 2 = 0x3FFFFFFD = A Big Number :)
0x18    0x4021D493      srai x9, x3, 2 # x9 = Ceil(-10/(2^2)) = -3
0x1c    0x0011D1B3      srl x3, x3, x1 # x3 = 0xFFFFFFF6 >> 1 = 0x7FFFFF7B = Another Big Number:)
0x20    0x40115133      sra x2, x2, x1 # x2 = 3 >>> 1 = floor(3/(2^1)) = 1
```

After these instructions were performed by the processor, final value of registers x4, x5, x7 and x7 must be 6, 48, 0x3FFFFFFD and -3 respectively. Registers x3 and x2 which start with values 3 and -10 must change their values to 1 and 0x7FFFFFFB respectively.

Simulation's Result:



As you can see, when program ends, all registers are holding the values they were meant to hold and now our processor can handle shift instructions correctly.

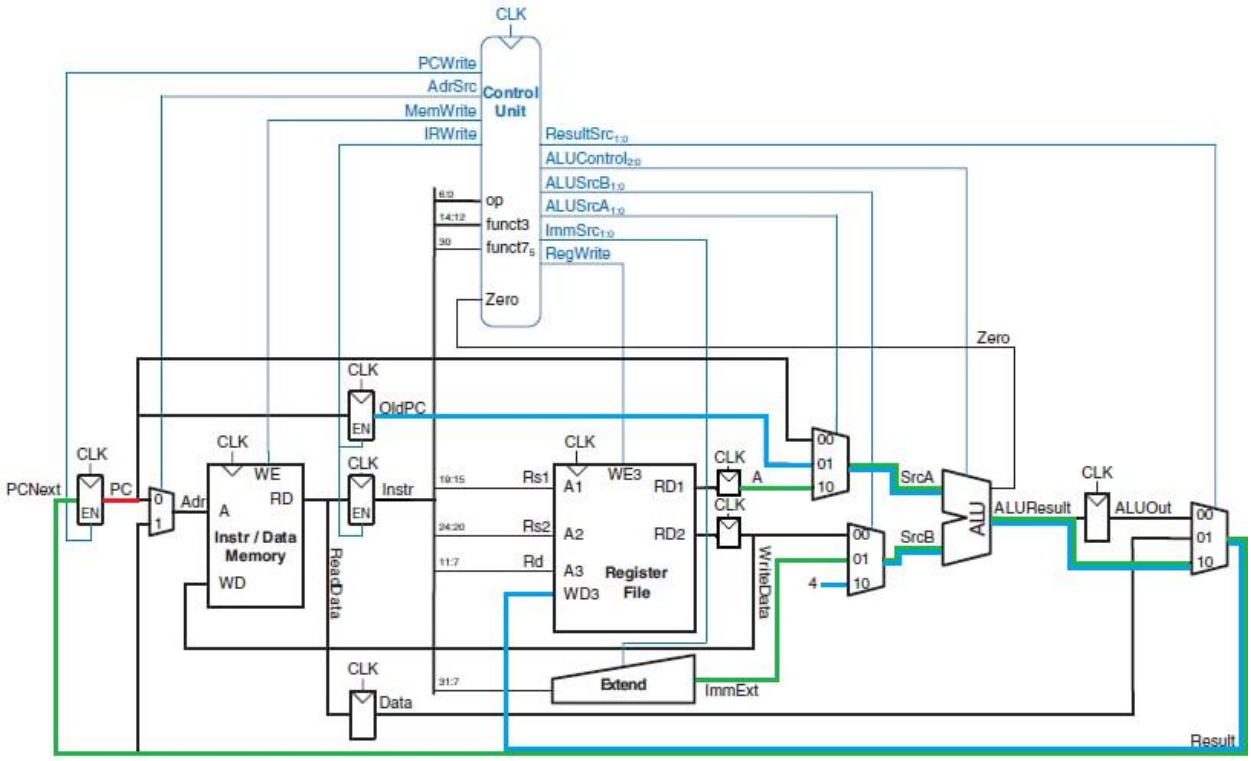
JALR is an I-type instruction, thus it's easy to add this instruction too.

8.1.4 JALR

This is what JALR does:

1100111 (103)	000	-	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4

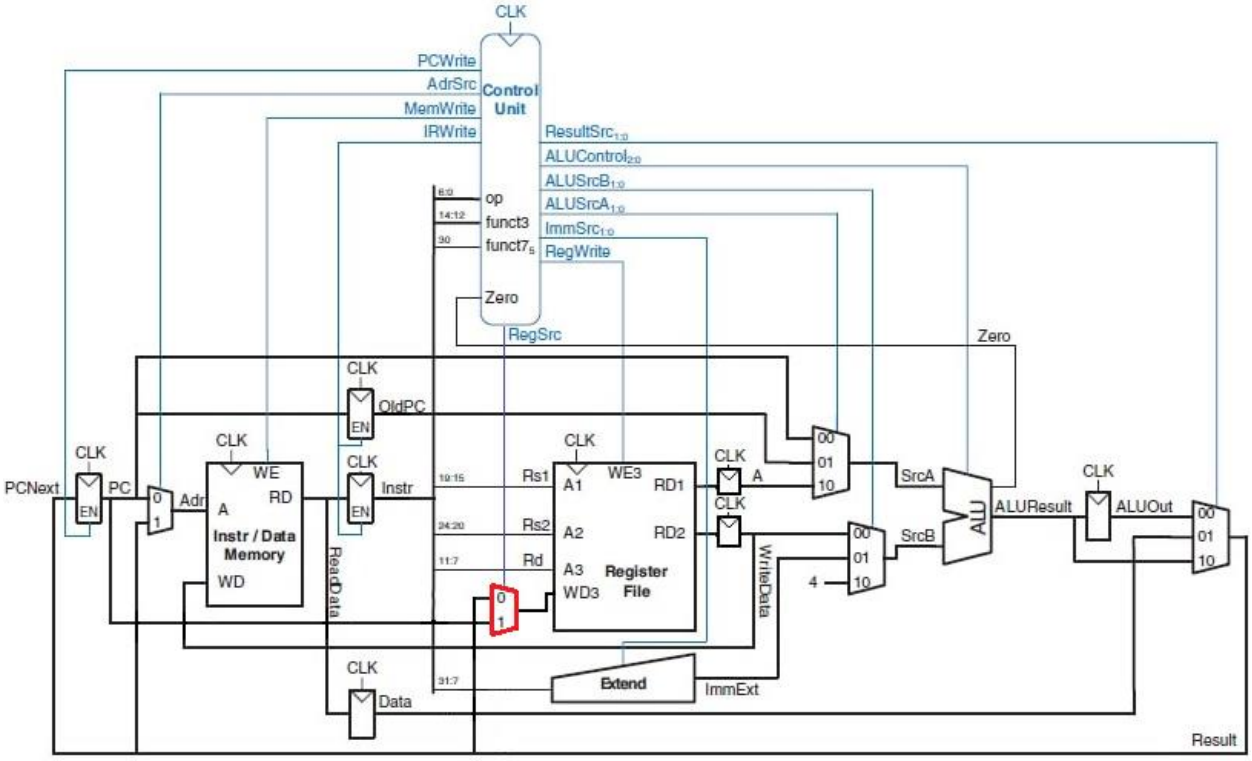
Since JALR is an I-type instruction, Extend Chip already now how to extends its immediate, so no need to add another extend mode. Just like "addi" we must add an immediate to a register, but now we don't want to write it in another register. The ALU result now must be considered as the next PC. The naïve approach is to add 2 cycles for each of the add operations that happens in this instruction (rd = PC+4, PC = rs1 + SignExt(imm)). This means that JALR will require 4 cycles to complete (2 cycles of Fetch and Decode and 1 cycle for each add).



Blue is the path for saving PC+4 in rd.

Green is the path for setting the next PC as rs1 + SignExt(imm).

But we know that PC+4 is saved in a register in the Fetch state. The red line marked in the data path hold PC+4 in Decode state, thus in the third cycle (after Fetch and Decode) we can both write the PC+4 in “rd” and add the first register to the immediate and set it as the next PC. To do this, a new multiplexer is added to our data path (Red mux) to choose the write value to write in “rd”.

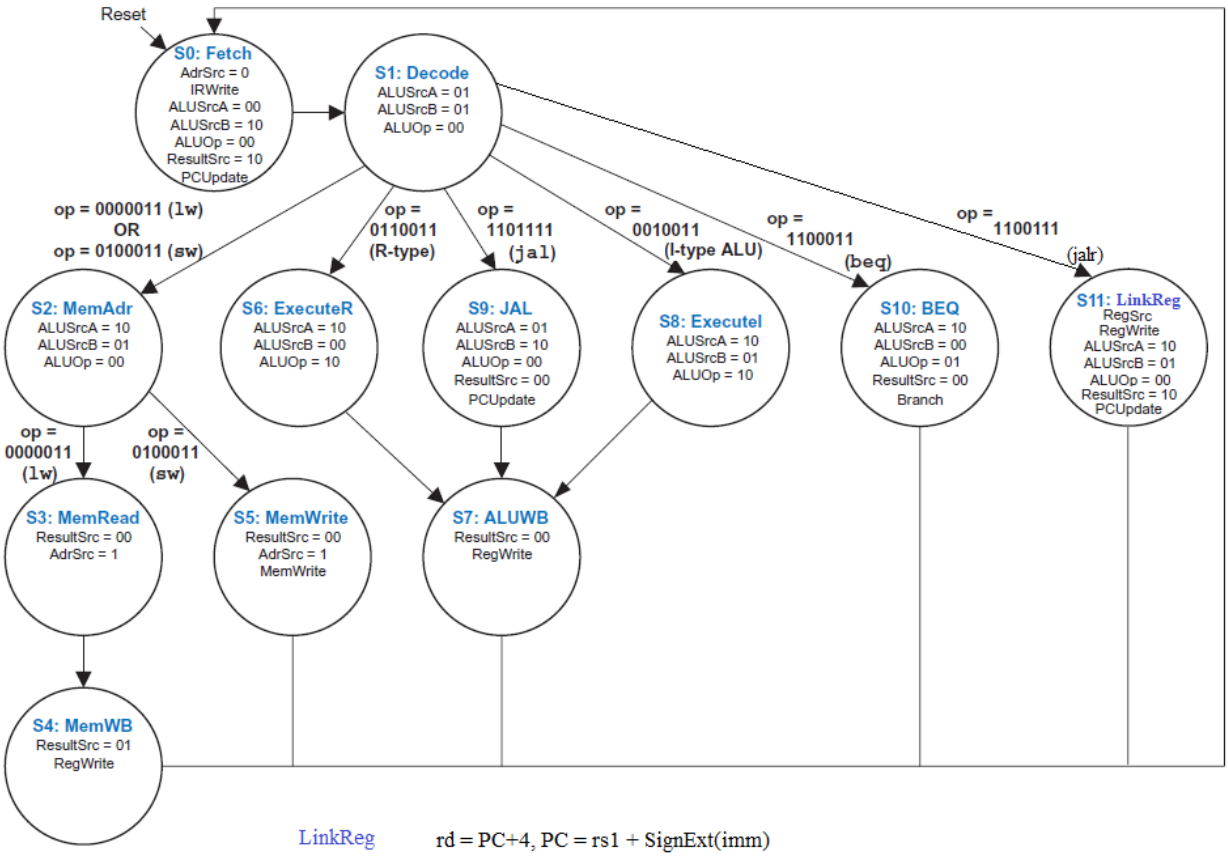


Now we also need a new control signal (RegSrc) for controlling this multiplexer. During all the instructions “RegSrc” remains 0 to pick Result to be written in “rd” except the JALR instruction. RegSrc is 1, if and only if the instruction is JALR (Op = 1100111), thus in our controller we defined RegSrc like this:

```
assign RegSrc = op == 7'b1100111 ? 1 : 0;
```

After decoding, we now add a new state (S11: LinkReg). In this state RegSrc becomes 1 and PC+4 which was calculated in S0 will be written in “rd”. ALU sources will be set to do the addition and ResultSrc will become 2’b10 and PCUpdate signal also becomes 1, thus ALUResult which is “rs1 + imm” will be set as the new PC.

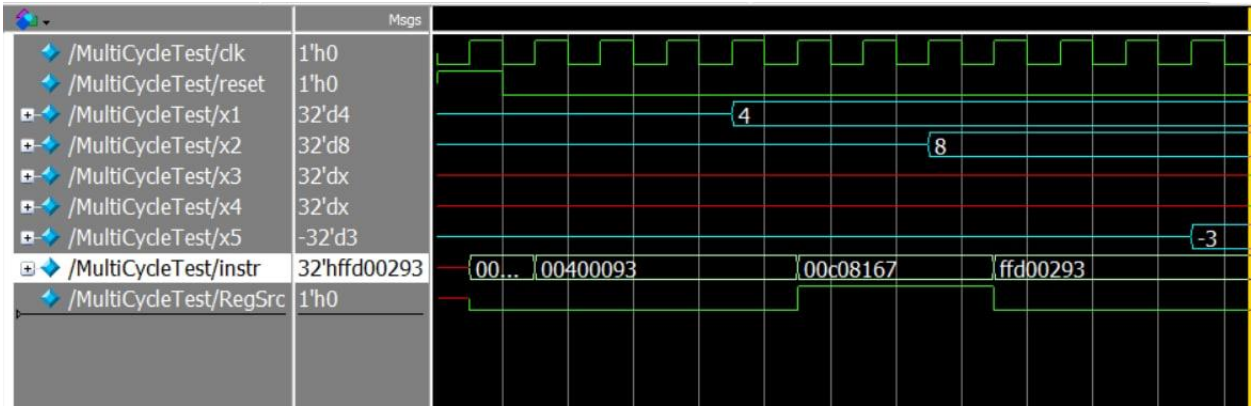
New FSM:



With this test bench,

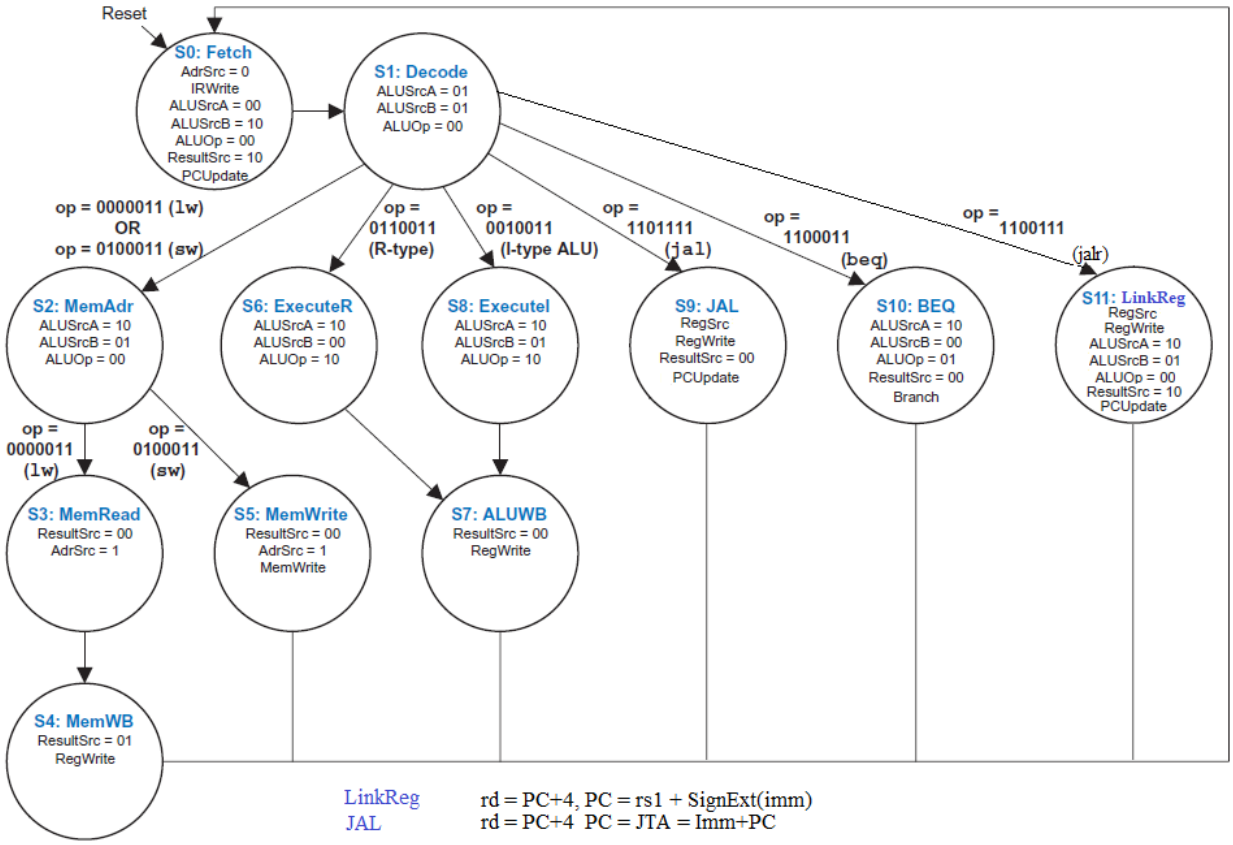
```
PC      Machine Code    Original Code
0x0     0x00400093      addi x1, x0, 4  # x1 = 4
0x4     0x00C08167      jalr x2, x1, 12 # x2 = PC+4 = 8, PC = 12+4 = 16 = 0x10
0x8     0x01300193      addi x3, x0, 19 # Should NOT happen
0xc     0x04900213      addi x4, x0, 73 # Should NOT happen
0x10    0xFFD00293      addi x5, x0, -3 # x5 = -3
```

Here is the result for our new Processor:



8.1.5 Improving JAL

We can now complete JAL instruction in 3 cycles too:



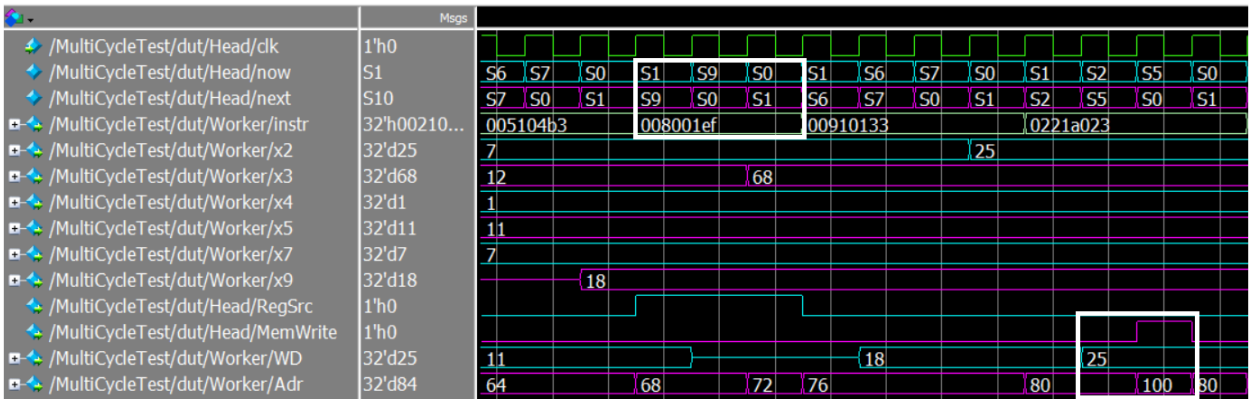
We now assign RegSrc to:

```
                          //JALR                    //JAL
assign RegSrc = op == 7'b1100111 ? 1 : op == 7'b1101111 ? 1 : 0;
```

S9 state's control signals now change to:

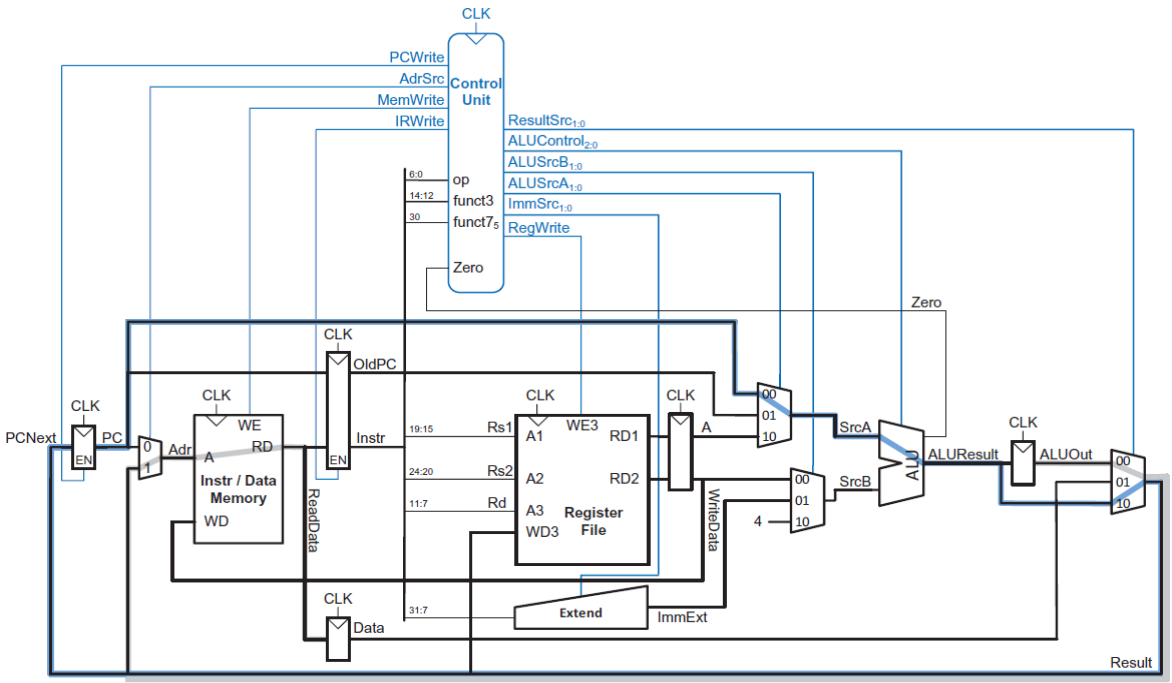
```
S9: begin
        next = S0;
        //FSMControls = 14'b00_01_10_00_x_0_1_0_0_0;
        //FSMControls = 14'b00_01_10_00_0_0_1_0_0_0;
        //New JAL with 3 Cycles:
        //FSMControls = 14'bxx_xx_xx_00_x_x_1_1_x_x;
          FSMControls = 14'b00_00_00_00_0_0_1_1_0_0;
end
```

Now we test our new processor which handles JAL in 3 cycles with the main test bench:



The white rectangle on the top is showing that the instruction 0x008001EF (jal x3, end) completes in 3 cycles (S0 → S1 → S9). The white rectangle in the bottom of the picture shows that the value 25 is still being written in memory address 100, thus the JAL is completed in 3 cycles correctly!

Although we can now handle JAL in 3 instructions instead of 4. A new multiplexer is added to our data path. This means that every time we need to write in a register, we must pass through this new multiplexer (Red one marked in the page 24). But according to Figure 7.46 of the book:



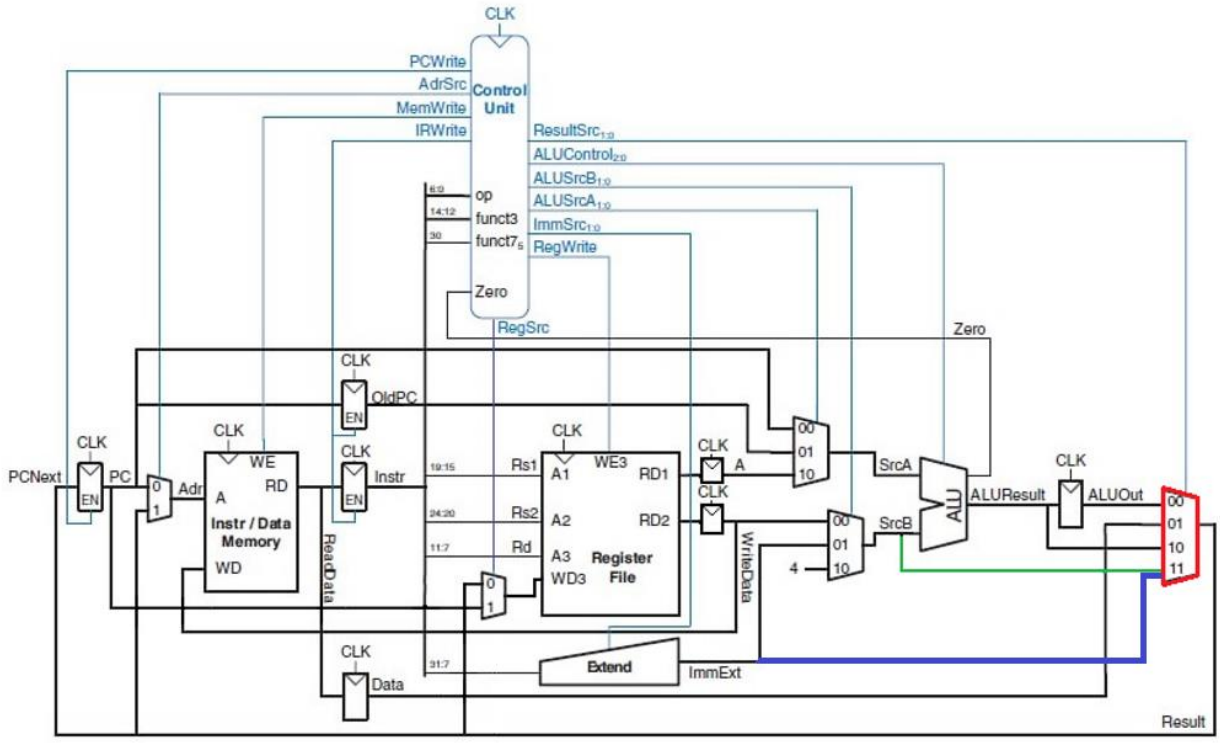
None of these two candidates (Blue and Gray paths) for critical path involves a register write, thus the new multiplexer that we added does not change our critical path and the maximum clock frequency remains intact. This means that the overall performance of our processor has improved!

8.1.6 LUI

This is what LUI instruction does:

0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}

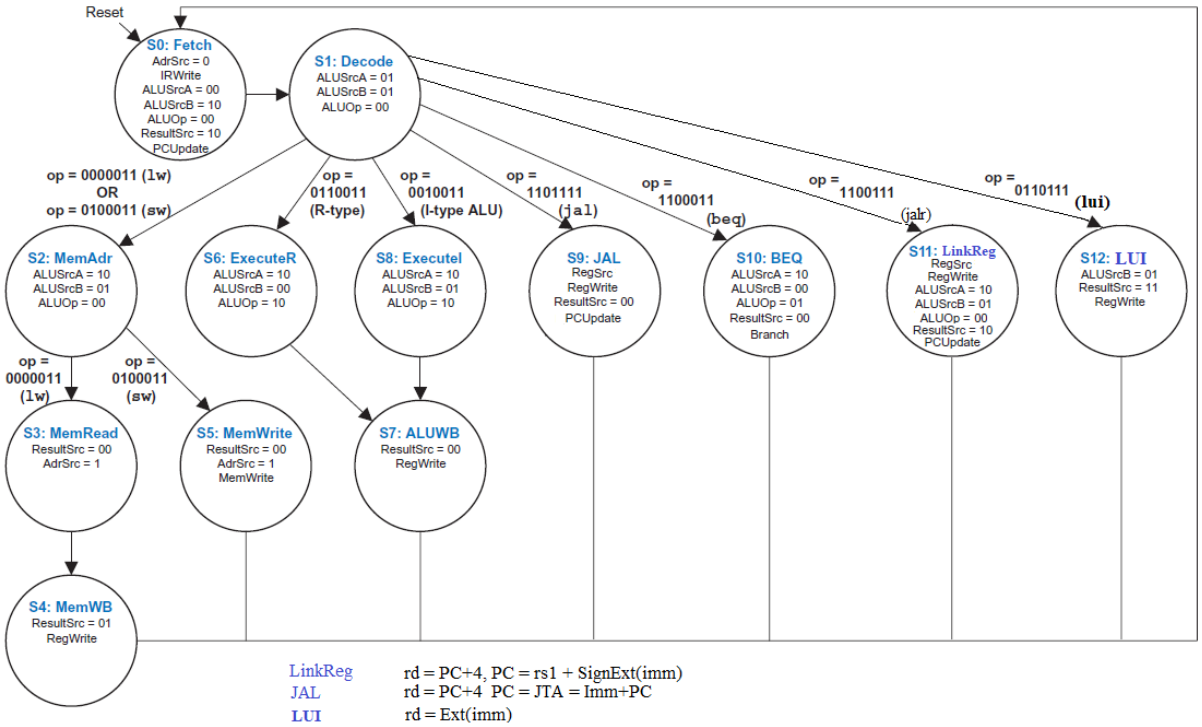
We need to improve our data path for handling this instruction. We could add another choice for the Result Mux and give it the ImmExt bus (Blue wire), this would solve our problem, but if we attach SrcB bus to the last choice of Result Mux we could also handle another instruction like "MOV rd, rs1", Thus we attach the green wire to the Result Mux.



LUI is a U-type instruction, therefore we need to add another extend mode to our extend chip. Based on Figure B.1 of the book, here is how we must extend a U-type instruction's immediate:

```
always @(instr, ImmSrc)
        case(ImmSrc)
           3'b000 : ImmExt = {{20{instr[31]}}, instr[31:20]} ; // I-type
           3'b001 : ImmExt = {{20{instr[31]}}, instr[31:25], instr[11:7]} ; // S-type
           3'b010 : ImmExt = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0} ; //B-type
           3'b011 : ImmExt = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0} ; // J-type
           3'b100 : ImmExt = {27'b0, instr[24:20]} ; // I-type (Shift Instructions)
           3'b101 : ImmExt = (instr[31:12], 12'b0) ; // U-type
        endcase
```

FSM must be updated:



Instruction decoder is also modified:

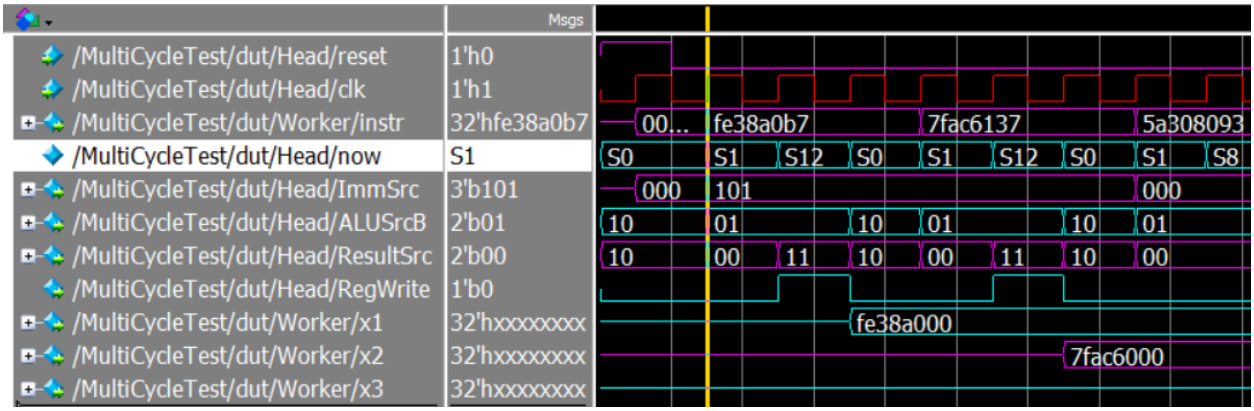
```
//Instruction Decoder
always @(op, funct3) begin
  case(op)
        7'b0000011:    ImmSrc = 3'b000;
        7'b0100011:    ImmSrc = 3'b001;
        7'b0110011: //ImmSrc = 3'bxxx;
                       ImmSrc = 3'b000;
        7'b1100011:    ImmSrc = 3'b010;
        7'b0010011: /*I-type*/
                       begin
                         if(funct3 == 3'b001 || //SLLI
                                 funct3 == 3'b101) //SRAI, SRLI
                                       ImmSrc = 3'b100;
                         else
                                       ImmSrc = 3'b000; //Others
                       end
        7'b1101111:    ImmSrc = 3'b011;
        7'b0110111:    ImmSrc = 3'b101; //Upper Immediate
        default    : //ImmSrc = 3'bxxx;
                       ImmSrc = 3'b000;
  endcase
end
```

Test bench:

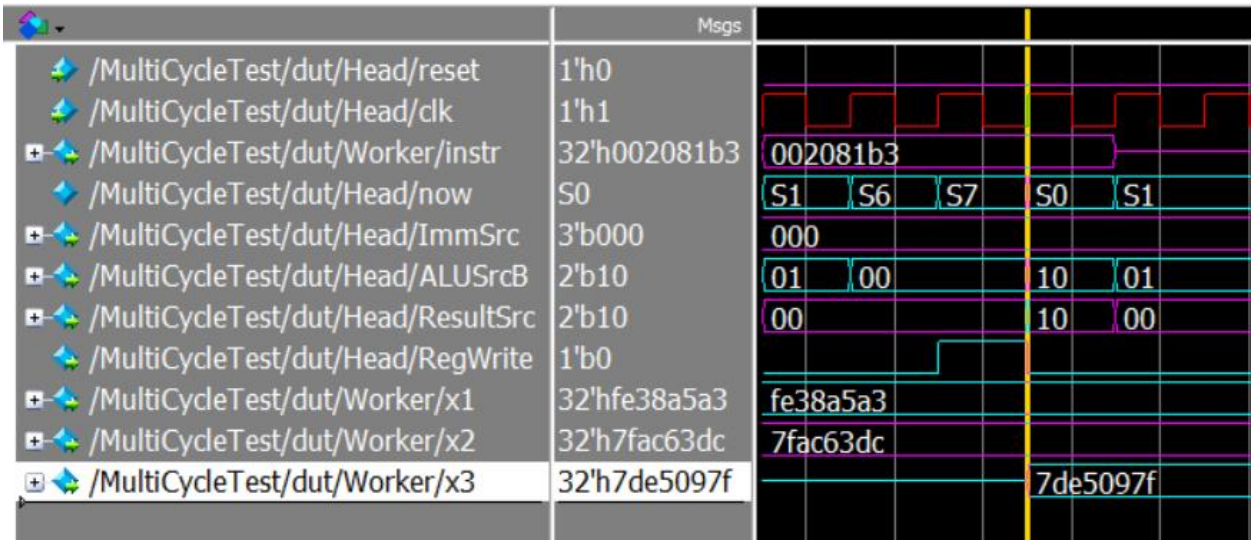
```
PC      Machine Code    Original Code
0x0     0xFE38A0B7      lui x1, 0xFE38A     # x1 = 0xFE38A000
0x4     0x7FAC6137      lui x2, 0x7FAC6     # x2 = 0x7FAC6000
0x8     0x5A308093      addi x1, x1, 0x5A3  # x1 = 0xFE38A5A3
0xc     0x3DC10113      addi x2, x2, 0x3DC  # x2 = 0x7FAC63DC
0x10    0x002081B3      add x3, x1, x2      # x3 = 0x7DE5097F
```

If these instructions were completed successfully, the value written in register x3 must be 0x7DE5097F!

Simulation's Results:



In S12, signals are set as we wanted them to. (ImmSrc = 3'b101, ALUSrcB = 2'b01, ResultSrc = 2'b11, RegWrite = 1). As you can see values 0xFE38A000 and 0x7FAC6000 are written in registers x1 and x2 successfully.



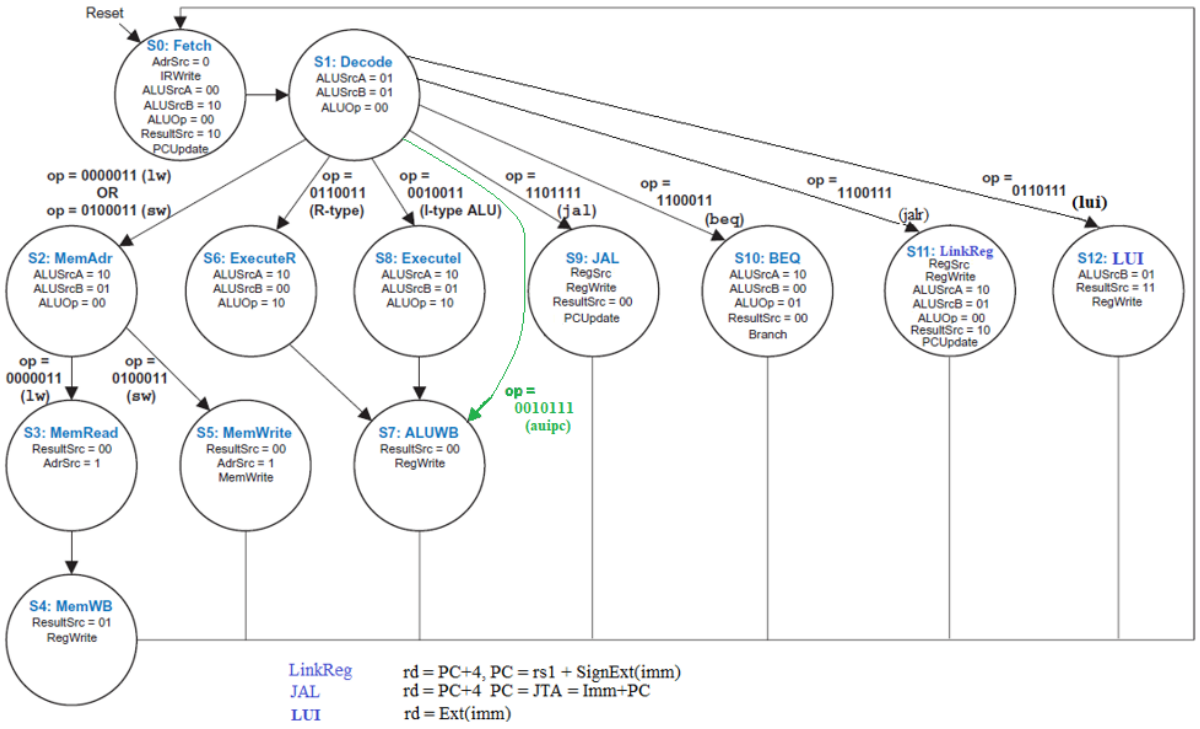
And at last, 0x7DE5097F is written in x3 as expected!

8.1.7 AUIPC

The “auipc” instruction adds a U-type immediate to PC and saves it in a register:

0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC

Adding an immediate to the PC is done in the Decode state, thus in “auipc” instruction we can go from Decode state straight to the ALUWB state (S7) for the imm+PC to be written in the desired register:

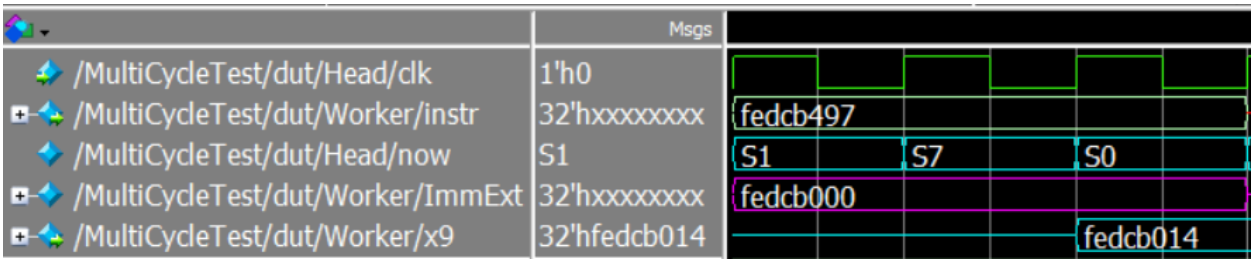


I just added one “auipc” instruction to the end of our last test bench:

```
PC      Machine Code    Original Code
0x0     0xFE38A0B7      lui x1, 0xFE38A     # x1 = 0xFE38A000
0x4     0x7FAC6137      lui x2, 0x7FAC6     # x2 = 0x7FAC6000
0x8     0x5A308093      addi x1, x1, 0x5A3  # x1 = 0xFE38A5A3
0xc     0x3DC10113      addi x2, x2, 0x3DC  # x2 = 0x7FAC63DC
0x10    0x002081B3      add x3, x1, x2      # x3 = 0x7DE5097F
0x14    0xFEDCB497      auipc x9, 0xFEDCB   # x9 = 0xFEDCB014
```

X9 = 0x14 + 0xFEDCB000 = 0xDEDCB014

Simulation’s Result:



“0xFEDCB497” is completed in 3 cycles (S0 → S1 → S7) and in the end the correct value 0xFEDCB014 is written in x9.

8.1.8 SLTU, SLTIU, BLTU, BGEU

For SLTU and SLTIU instructions we just add another mode to our ALU to compare the unsigned version of its two inputs A and B. But BLTU and BGEU instructions depend on the Sign flag output of the ALU, thus somehow, we need to set the sign flag to be 1 if the input A was little than B. Given that the result bus itself does not matter in branch instructions and we only care about the sign flag, I added a new mode to the ALU with the Op of 0xB. In this mode if the unsigned version of A was little than B, the result will be equal to 0xFFFFFFFF, thus the sign flag will rise to 1 and the branch decoder in control unit will handle BLTU and BGEU just like their unsigned version:

```
always @(A, B, Op)
        case(Op)
                4'b0000 : result = A + B ;      //0
                4'b0001 : result = A - B ;      //1
                4'b0101 : result = A < B ? 1 : 0 ; //5
                4'b0011 : result = A | B ;      //3
                4'b0010 : result = A & B ;      //2
        /*XOR*/ 4'b0100 : result = A ^ B ;      //4
        /*SLL*/ 4'b0110 : result = A << B;      //6
        /*SLA*/ 4'b0111 : result = A <<< B;     //7
        /*SRL*/ 4'b1000 : result = A >> B;      //8
        /*SRA*/ 4'b1001 : result = A >>> B;     //9
        /*SLTU*/4'b1010 : result = $unsigned(A) < $unsigned(B) ? 1 : 0;
        /*bltu*/4'b1011 : result = $unsigned(A) < $unsigned(B) ? -1 : 0;
                default : result = 32'b0 ;
        endcase
assign Zero = &(~result) ;
assign Sign = result[31] ;
```

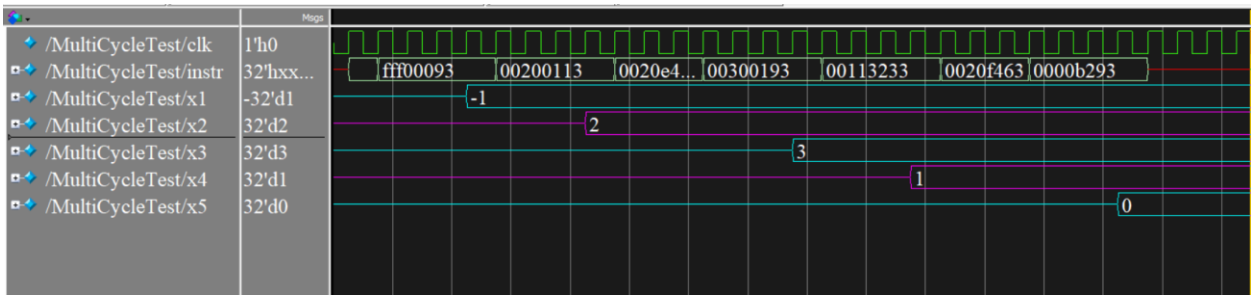
Two other modes must be added to the Branch Decoder:

```
//Branch Decoder:
always @(Branch, funct3, Zero, Sign)
        if(Branch)
          case(funct3)
        /*BEQ*/3'b000: BrResult = Zero;
        /*BNE*/3'b001: BrResult = ~Zero;
        /*BLT*/3'b100: BrResult = Sign;
        /*BGE*/3'b101: BrResult = ~Sign;
        /*BLTU*/3'b110: BrResult = Sign;
        /*BGEU*/3'b111: BrResult = ~Sign;
          endcase
        else BrResult = 0;
```

Here is the test bench:

```
PC      Machine Code    Original Code
0x0     0xFFF00093      addi x1, x0, -1     # x1 = -1 = (4294967295) unsigned
0x4     0x00200113      addi x2, x0, 2      # x2 = 2
0x8     0x0020E463      bltu x1, x2, L1     # 4294967295 > 2 ==> Branch NOT Taken
0xc     0x00300193      addi x3, x0, 3      # x3 = 3
                    L1:
0x10    0x00113233      sltu x4, x2, x1     # 2 < 4294967295 ==> x4 = 1
                    loop:
0x14    0x0020F463      bgeu x1, x2, end    # 4294967295 > 2 ==> Branch Taken
0x18    0xFE209EE3      bne x1, x2, loop
                    end:
0x1c    0x0000B293      sltiu x5, x1, 0     # 4294967295 > 0 ==> x5 = 0
```

Simulation's Results:

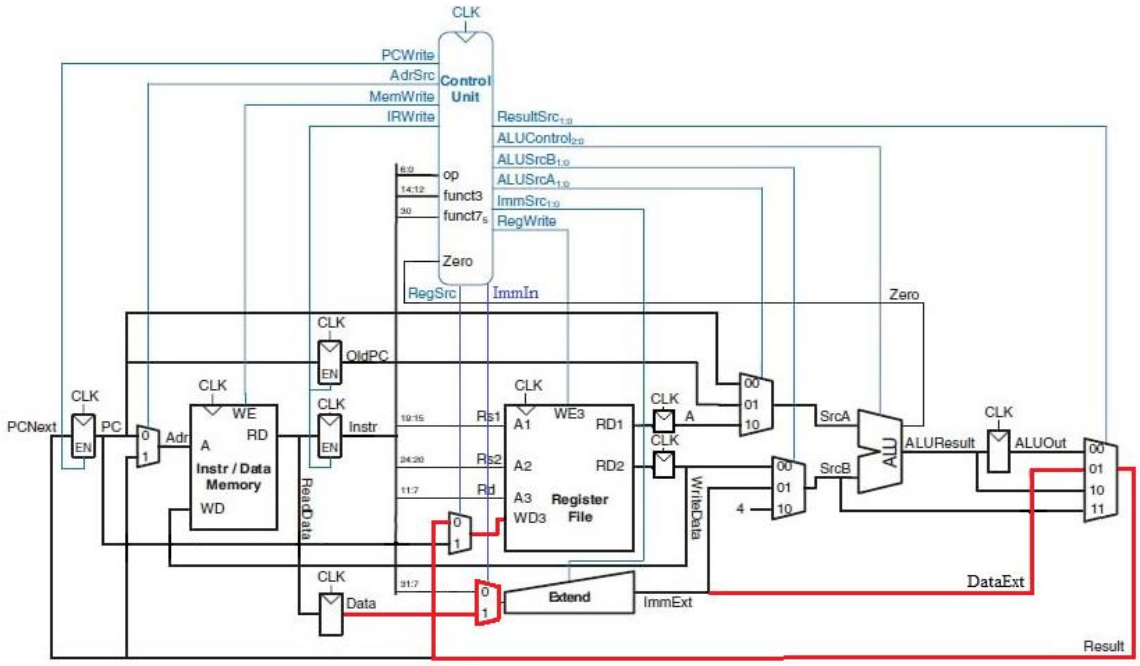


Everything is just as it was supposed to be. Instruction 0xFE209EE3 was skipped and 0x00300193 was not, thus "bgeu" and "bltu" are working fine. In the end, x5 is equal to 0 and x4 is equal to 1, thus "sltiu" and "sltu" are also working fine!

8.1.9 LB, LH, LBU, LHU

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`

According to the above table, we only want the first half or quarter of the ReadData bus from MemRead state. ReadData will be saved in a register and now we need to Extend the first half or quarter of it in MemWB state before writing it in the "rd" register. We could use our Extend Chip again to do this for us. For using the extend chip, we need to put a multiplexer before it to decide what value must be extended:

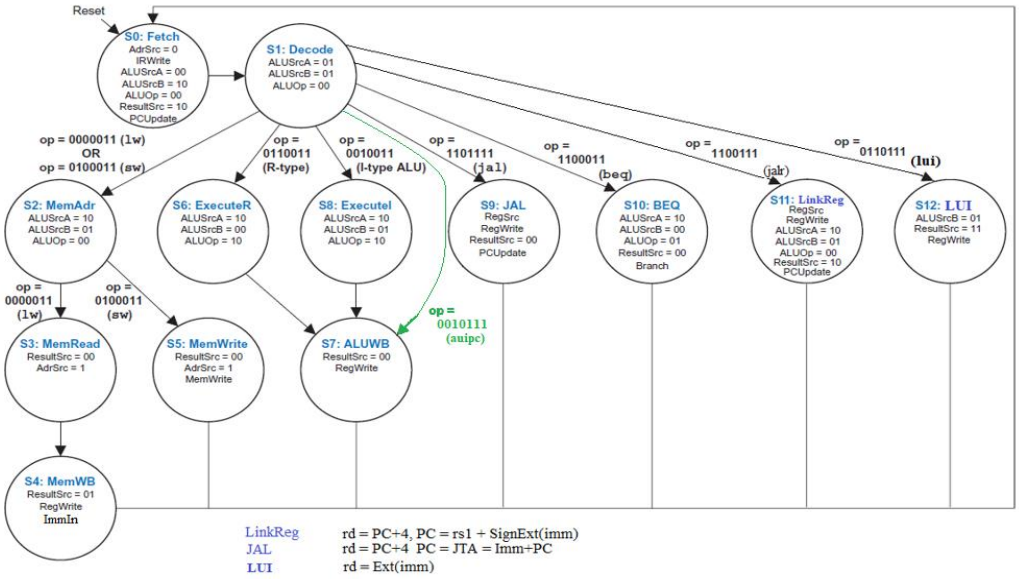


This is path that ReadData bus take before being written in register file. ImmIn signal only rises in S4. Now, we must add new modes to our Extend Chip:

```
always @(in, ImmSrc)
        case(ImmSrc)
           4'b0000 : ImmExt = {{20{in[31]}}, in[31:20]} ; // I-type
           4'b0001 : ImmExt = {{20{in[31]}}, in[31:25], in[11:7]} ; // S-type
           4'b0010 : ImmExt = {{20{in[31]}}, in[7], in[30:25], in[11:8], 1'b0} ; //B-type
           4'b0011 : ImmExt = {{12{in[31]}}, in[19:12], in[20], in[30:21], 1'b0} ; // J-type
           4'b0100 : ImmExt = {27'b0, in[24:20]} ; // I-type (Shift Instructions)
           4'b0101 : ImmExt = {in[31:12], 12'b0} ; // U-type
           4'b0110 : ImmExt = {16'b0, in[15:0]};  //lhu
           4'b0111 : ImmExt = {24'b0, in[7:0]};   //lbu
           4'b1000 : ImmExt = {{16{in[15]}}, in[15:0]}; //lh
           4'b1001 : ImmExt = {{24{in[7]}}, in[7:0]}; //lb
           default : ImmExt = in; //lw
        endcase
```

In "lw" instruction, the exact input of the Extend Chip must come out. Other load instruction's extend modes are set according to the book.

ImmIn signal must become 1 in S4 of the FSM:



This is the test bench:

```
PC      Machine Code    Original Code
0x0     0xABCDE0B7      lui x1, 0xABCDE     # x1 = 0xABCDE000
0x4     0x1A308093      addi x1, x1, 0x1A3  # x1 = 0xABCDE123
0x8     0x06102023      sw x1, 96(x0)       # mem[96] = 0xABCDE123
0xc     0x06001103      lh x2, 96(x0)       # x2 = 0xFFFFE123 SignExtended
0x10    0x06005183      lhu x3 96(x0)       # x3 = 0x0000E123 ZeroExtended
0x14    0x06000203      lb x4, 96(x0)       # x4 = 0xFFFFFFA3 SignExtended
0x18    0x06004283      lbu x5, 96(x0)      # x5 = 0x000000A3 ZeroExtended
```

And here is the Simulation's result:

All registers all holding the values which they were supposed to hold!

8.1.10 SB, SH

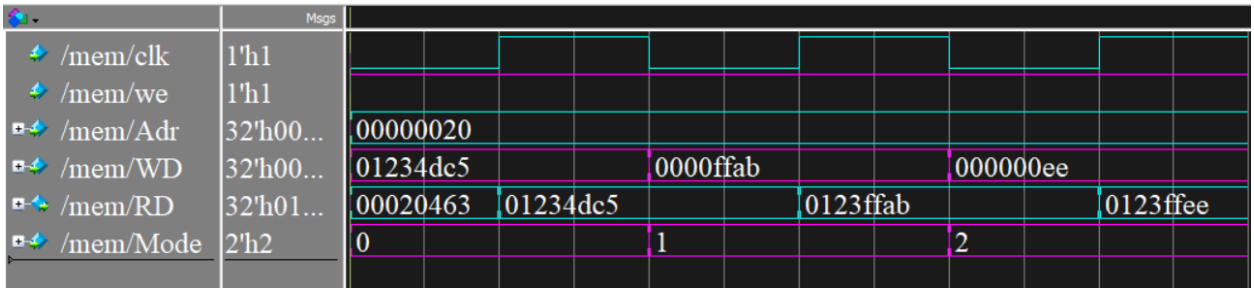
We need to another new modes for writing in our memory:

```
assign RD = RAM[Adr[31:2]]; // word aligned

always_ff @(posedge clk)
  //if(we) RAM[Adr[31:2]] <= WD;
  if (we) case(Mode)
              2'b01: if(Adr[1]==0) RAM[Adr[31:2]][15:0] <= WD[15:0]; //half
                     else          RAM[Adr[31:2]][31:16]<= WD[15:0];
              2'b10: case(Adr[1:0])
                      2'b00: RAM[Adr[31:2]][7:0] <= WD[7:0]; //byte
                      2'b01: RAM[Adr[31:2]][15:8] <= WD[7:0];
                      2'b10: RAM[Adr[31:2]][23:16] <= WD[7:0];
                      2'b11: RAM[Adr[31:2]][31:24] <= WD[7:0];
                      endcase
              default: RAM[Adr[31:2]] <= WD;
              endcase
```

A new Signal MemMode will control which one of the writing types should happen:

```
always @(op, funct3)
        if(op==7'b0100011)  //Store
                if(funct3 == 3'b000) //sb
                        MemMode = 2'b10;
                else if(funct3 == 3'b001) //sh
                        MemMode = 2'b01;
                else MemMode = 2'b00; //sw
        else MemMode = 2'b00; //Non-Store
```

Testing the memory alone:

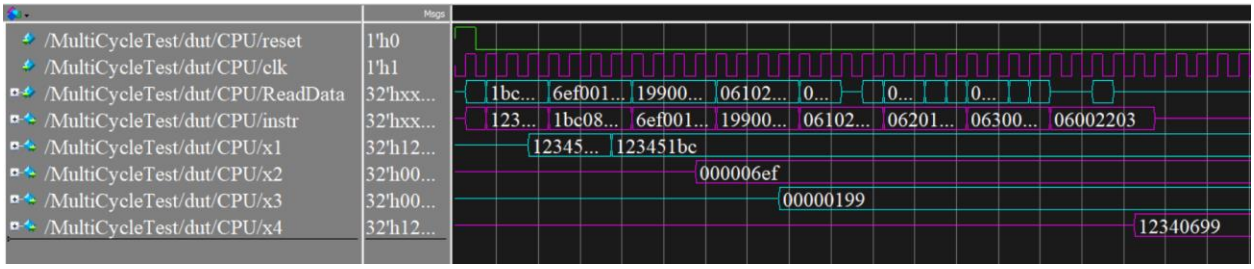


With Mode = 0, 0x1234dc5 is written in mem[0x20]. With Mode = 1, 0xffab is written in lower half of mem[0x20] and in Mode = 2, 0xee is written in the least significant byte of mem[0x20].

Test bench written:

```
PC      Machine Code    Original Code
0x0     0x123450B7      lui x1, 0x12345     # x1 = 0x12345000
0x4     0x1BC08093      addi x1, x1, 0x1bc  # x1 = 0x123451bc
0x8     0x6EF00113      addi x2, x0, 0x6ef  # x2 = 0x6ef
0xc     0x19900193      addi x3, x0, 0x199  # x3 = 0x199
0x10    0x06102023      sw x1, 96(x0)       # mem[96] = 0x123451bc
0x14    0x06201023      sh x2, 96(x0)       # mem[96] = 0x123406ef
0x18    0x06300023      sb x3, 96(x0)       # mem[96] = 0x12340699
0x1c    0x06002203      lw x4, 96(x0)       # x4 = 0x12340699
```

Simulation's Results:



Final value in x4 is 0x12340699 as expected!