ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION PROJECT 2 REPORT

CRN	:	21336 , 21334
LECTURER	:	Mustafa Ersel Kamaşak, Gökhan İnce

GROUP MEMBERS:

820220334	:	İrem Kalay(Group Representative)
150200114	:	Hakan Çetinkaya

SPRING 2024

Contents

1 INTRODUCTION

Using our first project's structures, we designed a hardwired control unit in this second project. There are 34 instructions overall, divided into two categories: instruction with address reference and instruction without address reference . The inputs of the first project's modules are connected to the control unit that we designed. These inputs consist of loading, function selection, selection, ALU inputs, and so forth. We were able to ensure that the first project's components operated correctly and synchronously by developing a timing unit. While implementing the instructions, firstly we used pen and paper then write their codes so that while debugging we were not lost inside the code.

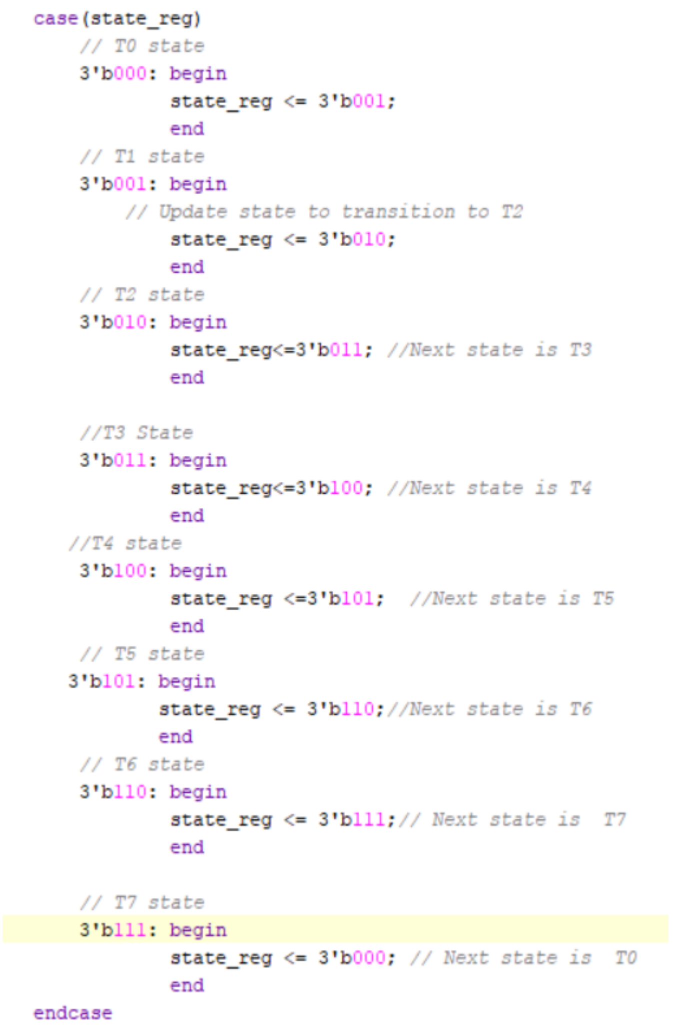
1.1 Task Distribution

For creating timing unit , we togather searched several ways. Most of the code written by İrem. SREG1,SREG2 DSTREG(Instructions without address referance) instructions written by Hakan. Instructions with address referance are written and controls are done by İrem.

2 MATERIALS AND METHODS

2.1 Timing Unit

We require a time control unit in order to ensure that every process runs simultaneously and on schedule. For this reason, we developed a sequence counter that establishes what has to happen when and how. We implemented it as if it was a state machine. Basically, we used a case statement for implementing the states. At the end of each state, the reg that control states is assigned to the next state. However, we needed to decode that reg into distinct lines for each of the times, which is why we employed a decoder attached to it namely we designed a 3 to 8 decoder for timing signals. After each instruction finished, we set the counter reg to zero, so that the state machine starts to count from zero again for next instruction cycle. The simulation of it is at Result part. An example for our Timing machine is below in the Figure 1. We did not use an extra module for it. We just used case statement.



```
case(state_reg)
    // T0 state
    3'b000: begin
            state_reg <= 3'b001;
            end
    // T1 state
    3'b001: begin
        // Update state to transition to T2
            state_reg <= 3'b010;
            end
    // T2 state
    3'b010: begin
            state_reg<=3'b011; //Next state is T3
            end

    //T3 State
    3'b011: begin
            state_reg<=3'b100; //Next state is T4
            end
   //T4 state
    3'b100: begin
            state_reg <=3'b101;  //Next state is T5
            end
    // T5 state
   3'b101: begin
           state_reg <= 3'b110;//Next state is T6
           end
    // T6 state
    3'b110: begin
            state_reg <= 3'b111;// Next state is  T7
            end

    // T7 state
    3'b111: begin
            state_reg <= 3'b000; // Next state is  T0
            end
endcase
```

Figure 1: State machine for Timing Structure

and this is the decoder for decoding the 3 bit state-reg we had. Total we have from T0 to T7 : 8 states.

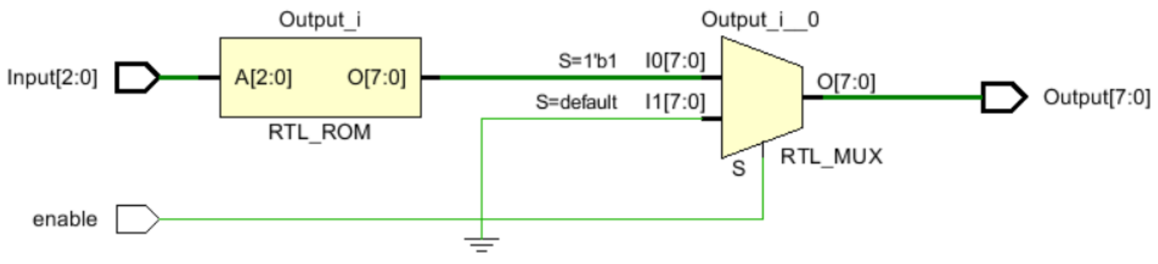
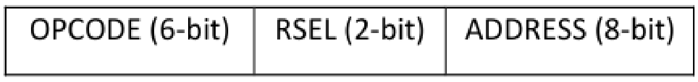


Figure 2: Decoder 3 to 8

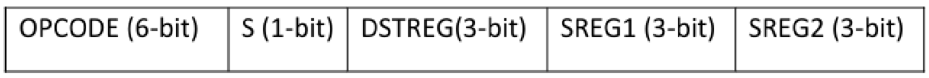
2.2 Fetching and Decoding

In terms of fetching, we had 16 bit long IR which means we would need to fetch 16 bits. However the memory out was 8 bits. The design was suggesting us to take first the low bits for IR, then increment the PC and take the high bits from memory so that we were fetching the wanted 16 bit instruction. Because of the design, an instruction required two cycles to fetch from memory. We retrieve the Low part first, followed by the High part. We begin at position 0 since every register is reset at the beginning. While getting the high part is dependent on T1, obtaining the low part is dependent on T0.



OPCODE (6-bit)	RSEL (2-bit)	ADDRESS (8-bit)

Figure 3: Instruction with an Address Reference



OPCODE (6-bit)	S (1-bit)	DSTREG(3-bit)	SREG1 (3-bit)	SREG2 (3-bit)

Figure 4: Instructions without an Address Reference

2.3 The Design of Executions

We have included if conditions in our architecture to control which operations are called from the ALU system. To make things easier, we decided to always utilize SREG1 as the A input of the ALU since it was the most often used register type. Entering data into the registers of register files was made simpler by the fact that ALU's A input was connected to both RF and ARF.

2.3.1 Working of instructions with SREG1, SREG2 and DSTREG

We performed the operations which has both SREG1 and SREG2 with an operation (DSTREG ← SREG1 operation SREG2) in 3 clock cycles : T2, T3, T4 for the simplicity of the design and for not allowing any collusions.

We chose SREG1 in the clock cycle T2 weather it is from RF or from ARF and SREG2 in the clock cycle T3 whether it is from ARF or RF. Then in the T4, the ALUFunsel is set to the required value for the specified operation as well as It is decided whether DSTREG will be in ARF or RF. These steps occurred in the operations containing both SREG1 and SREG2, namely 0x0C, 0x0D, 0x0F, 0x10, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D. Note that we assign 1'b1 to S variable which is responsible for changing flags.The describtions will be explained in more detailed in the discussion part. The instructions are as follows below in the Figure 5:

OPCODE (HEX)	**SYMBOL**	**DESCRIPTION**
0x0C	AND	DSTREG ← SREG1 AND SREG2
0x0D	ORR	DSTREG ← SREG1 OR SREG2
0x0F	XOR	DSTREG ← SREG1 XOR SREG2
0x10	NAND	DSTREG ← SREG1 NAND SREG2
0x15	ADD	DSTREG ← SREG1 + SREG2
0x16	ADC	DSTREG ← SREG1 + SREG2 + CARRY
0x17	SUB	DSTREG ← SREG1 - SREG2
0x18	MOVS	DSTREG ← SREG1 AND SREG2
0x19	ADDS	DSTREG ← SREG1 + SREG2, Flags will change
0x1A	SUBS	DSTREG ← SREG1 - SREG2, Flags will change
0x1B	ANDS	DSTREG ← SREG1 AND SREG2, Flags will change
0x1C	ORRS	DSTREG ← SREG1 OR SREG2, Flags will change
0x1D	XORS	DSTREG ← SREG1 XOR SREG2, Flags will change

Figure 5: Operations with 2 SREG

2.3.2 Working of instructions with only SREG1 and DSTREG

Similar to the SREG1, SREG2, DSTREG approach, this time we used two clock cycles for SREG1 and DSTREG , not 3 cycles the differance is this(because this time we have only SREG1 but no SREG2, and we perform the operation DSTREG ← SREG1. So that while perfoming these operations, we saved 1 clock cycle. In discussion part, the cycles are explained in more detailed. The operations can be seen in the Figure 6 :

OPCODE (HEX)	SYMBOL	DESCRIPTION
0x05	INC	DSTREG ← SREG1 + 1
0x06	DEC	DSTREG← SREG1 - 1
0x07	LSL	DSTREG ← LSL SREG1
0x08	LSR	DSTREG ← LSR SREG1
0x09	ASR	DSTREG ← ASR SREG1
0x0A	CSL	DSTREG ← CSL SREG1
0x0B	CSR	DSTREG ← CSR SREG1
0x0E	NOT	DSTREG ← NOT SREG1

Figure 6: Operations with only SREG1

2.3.3 Working of Address Reference Instructions

For implementing the address reference instructions, we changed the approach we were using. This time , we started to implement them one by one and we grouped some of them if we noticed the same pattern. For instance: BRA , BNE, BEQ all of them performs the instruction PC ← PC + VALUE at the end however their conditions for being executed are different. We used ”else if ” sturctures while writing each of them. Before writing in verilog, we firstly wrote on paper to see how many clock cycles and what joint those codes have. For simplicity, we did not unite much of codes in the address referance instructions.

We had the address referance instructions as : BRA , BNE, BEQ, POP, PSH, LDR, STR, BX, BL, LDRIM, STRIM, MOVH and MOVL.

while writing the codes for POP, PSH, BX , BL and STRIM , we used specifically extra cycle for writing into memory or taking value from memory to load the specific register. Because the specific register was 16 bits but the memory just had 8 bits. So we needed two consecutive memory places for the operations.

The instructions are as follows below in the Table 7:

OPCODE (HEX)	**SYMBOL**	**DESCRIPTION**
0x00	BRA	PC $\leftarrow$ PC + VALUE
0x01	BNE	IF Z=0 THEN PC $\leftarrow$ PC + VALUE
0x02	BEQ	IF Z=1 THEN PC $\leftarrow$PC + VALUE
0x03	POP	SP $\leftarrow$ SP + 1, Rx $\leftarrow$ M[SP]
0x04	PSH	M[SP] $\leftarrow$ Rx, SP $\leftarrow$ SP -1
0x11	MOVH	RSEL[15:8] $\leftarrow$ IMMEDIATE(8 bit)
0x12	LDR	Rx $\leftarrow$ M[AR]
0x13	STR	M[AR] $\leftarrow$ Rx
0x14	MOVL	RSEL[7:0] $\leftarrow$ IMMEDIATE(8 bit)
0x1E	BX	M[SP] $\leftarrow$ PC, PC $\leftarrow$ Rx
0x1F	BL	PC $\leftarrow$ M[SP]
0x20	LDRIM	Rx $\leftarrow$ VALUE
0x21	STRIM	M[AR + OFFSET] $\leftarrow$ Rx

Figure 7: Some Address Referance Instructions

3 RESULTS

Give your results what you get during the project. You can also add the table, image, etc.

3.1 Timing Unit

We created Timing unit that counts from T0 to T7 , the simulation of it can be seen in the below simulation in the Figure 8:

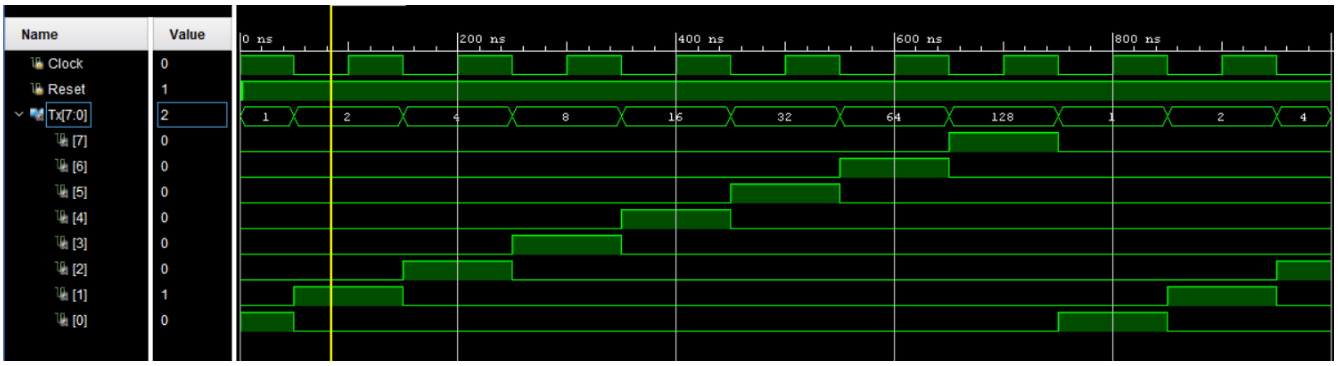


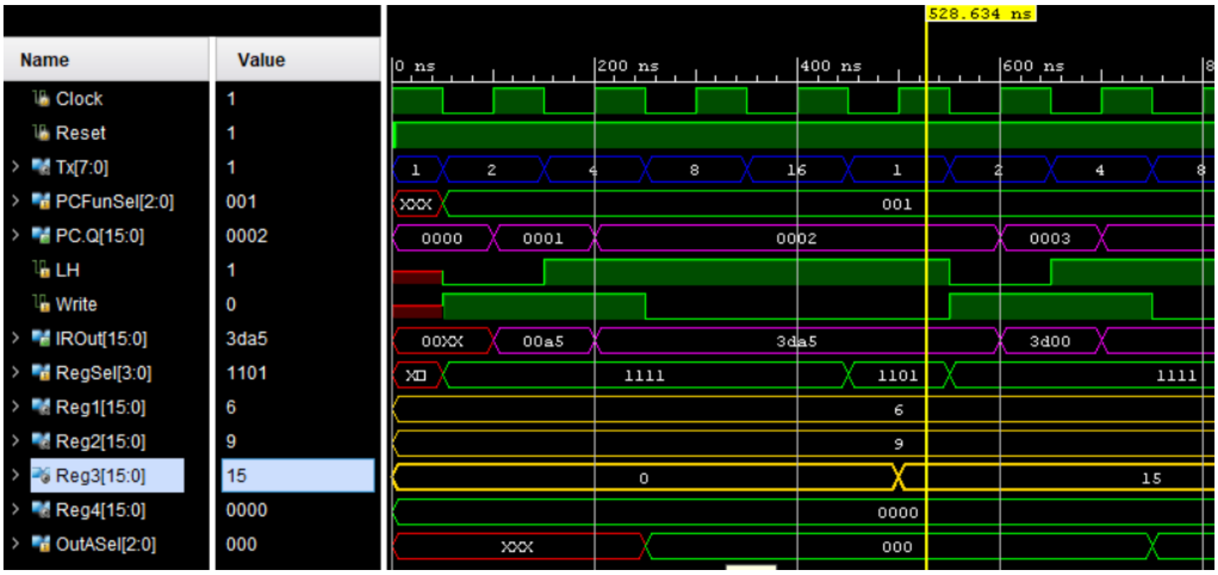
Figure 8: Timing Simulation

3.2 Fetching and Execution

Some instructions that I tested. I tested each operation at PC=0 initially to observe exactly what I see:

3.2.1 DSTREG ← SREG1 operation SREG2

We tried 3D A5 instruction for XOR operation. As can be seen in the Figure 9 the values of SREG1 and SREG2 are 6 and 9 in orderly. The result is equal to 15 as expected.



Name	Value
Clock	1
Reset	1
Tx[7:0]	1
PCFunSel[2:0]	001
PC.Q[15:0]	0002
LH	1
Write	0
IROut[15:0]	3da5
RegSel[3:0]	1101
Reg1[15:0]	6
Reg2[15:0]	9
Reg3[15:0]	15
Reg4[15:0]	0000
OutASel[2:0]	000

Figure 9: Fetching and Executing XOR

3.2.2 BRA:

In the below, fetching for BRA instruction can be seen. In the first two clock cycles fetching is done and afterwards it performs BRA PC ← PC + VALUE. After instruction performs, it resets the counter and again fetches the next instruction where the PC points at memory. In the Figure 10, it took the second instruction as 0000 because the memory block was empty for PC=42 and PC=43 in that illustration.

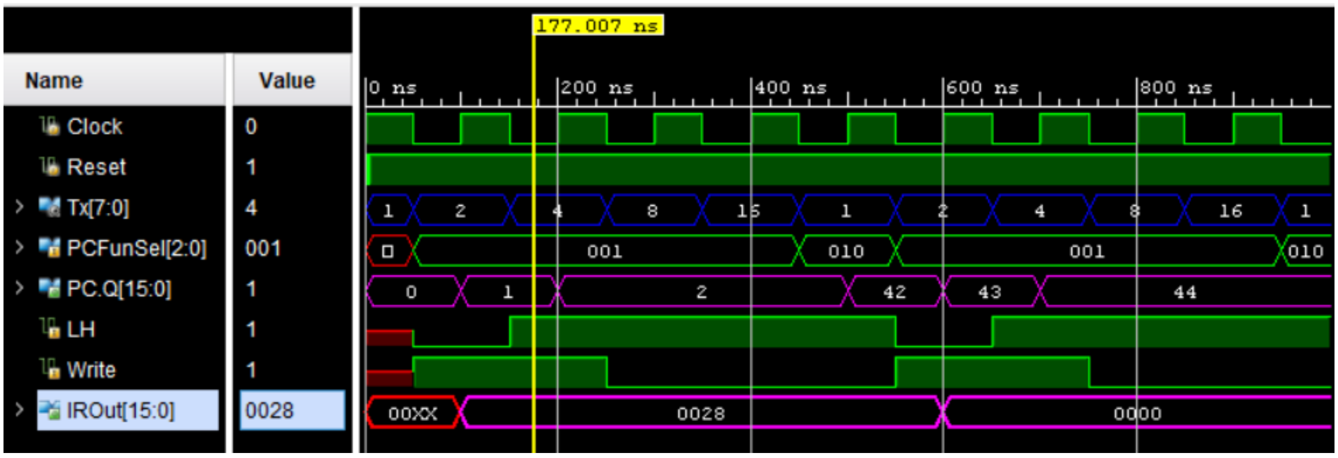


Figure 10: Fetching for BRA instruction

3.2.3 LDR:

The instruction that we tried with was: hex 48 00, RSEL selects R1. AR at initially was pointing 0. That is the reason why , it took the first 2 cells 0 and 1 from memory and wrote them to the selected register R1. In the Figure 11 the R1 value can be seen as 4800 hexadecimal.

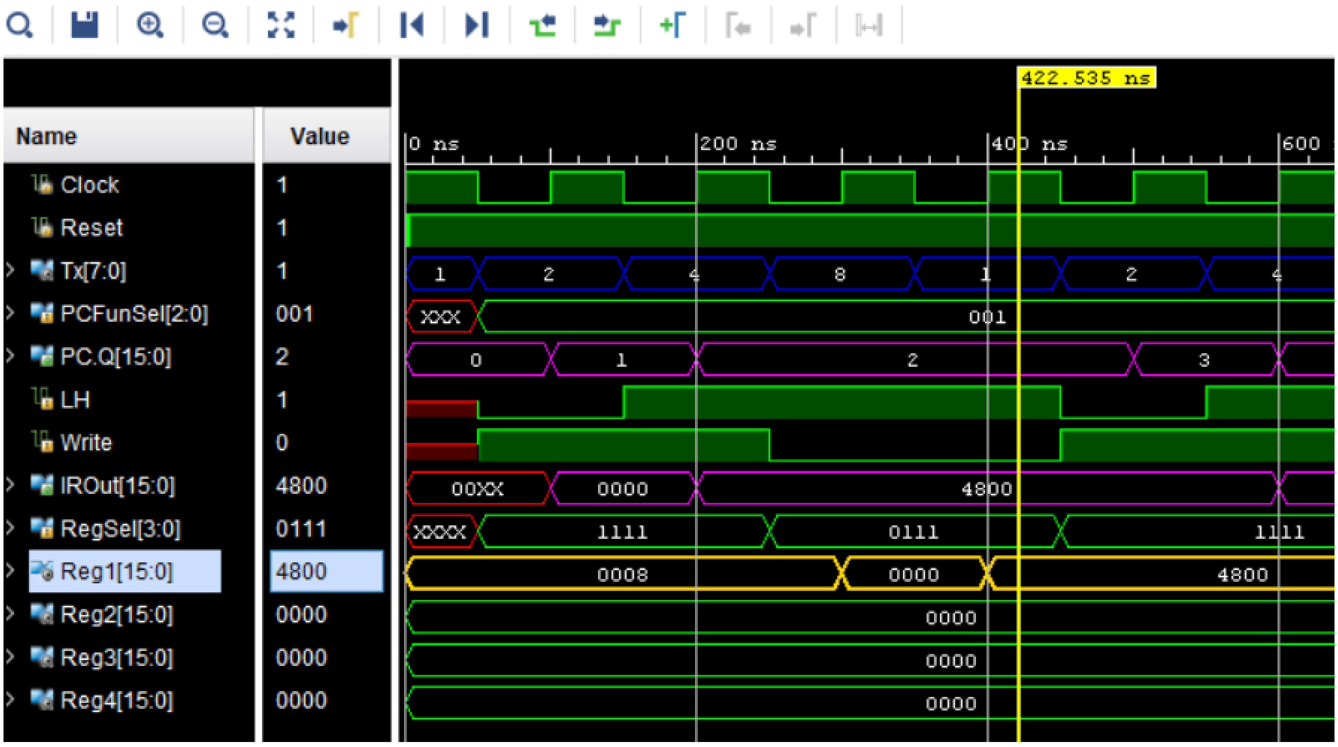
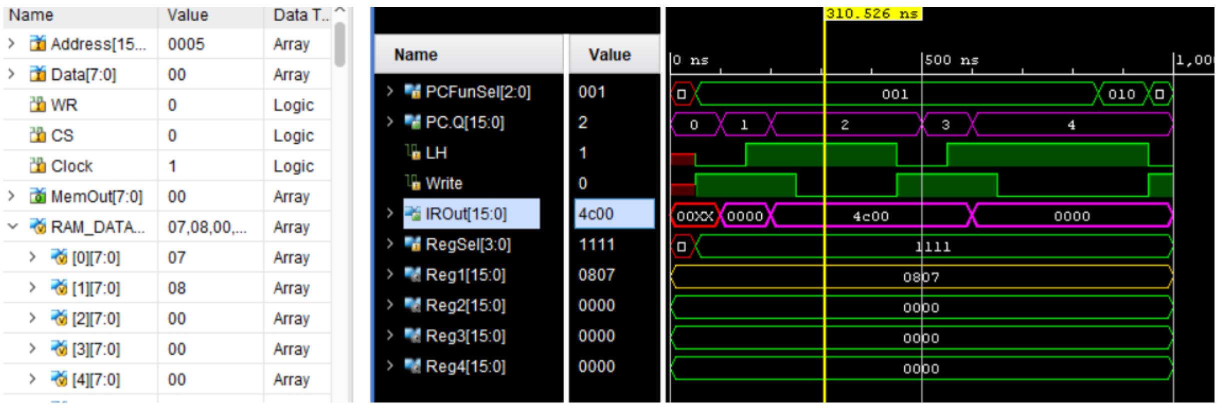


Figure 11: Fetching AND Executing LDR

3.2.4 STR:

The instruction that we tried with was: hex 4C 00, RSEL selects R1. This instruction needed to perform the following: M[AR] ← Rx. We gave to Rx an initial value of hex 0807 and my AR was initially pointing to zero . We observed the 07 08 at RAMDATA array can be seen at left hand side of the Figure 12 .



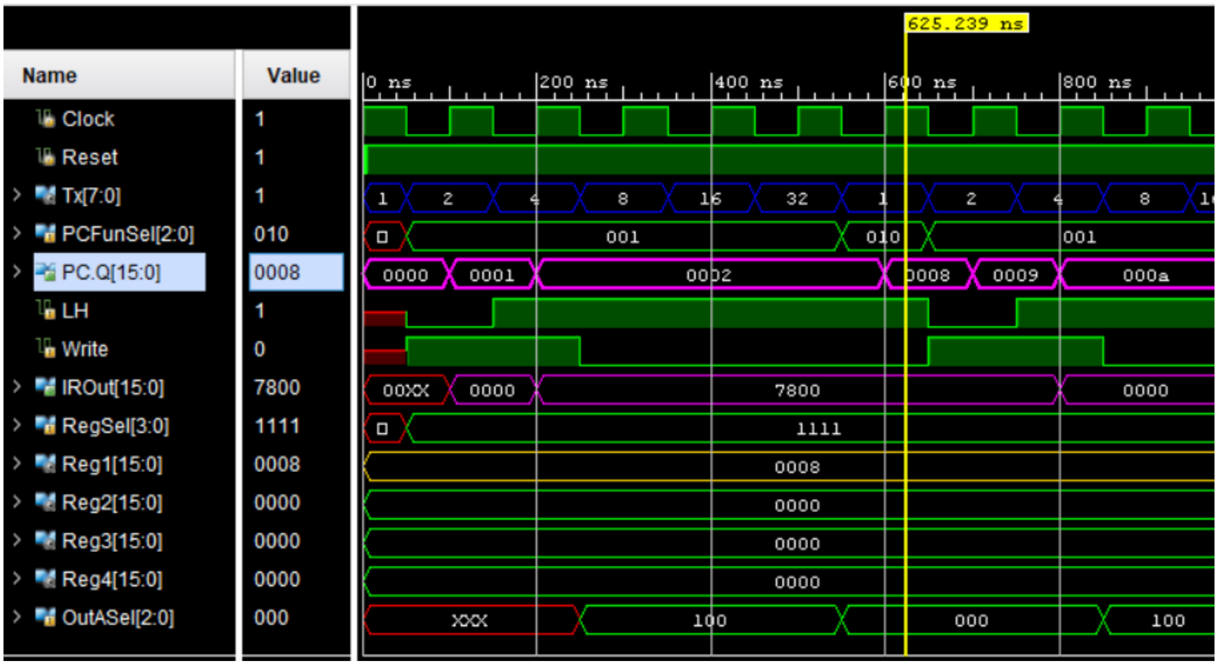
Name	Value	Data T...
Address[15...	0005	Array
Data[7:0]	00	Array
WR	0	Logic
CS	0	Logic
Clock	1	Logic
MemOut[7:0]	00	Array
RAM_DATA...	07,08,00,...	Array
[0][7:0]	07	Array
[1][7:0]	08	Array
[2][7:0]	00	Array
[3][7:0]	00	Array
[4][7:0]	00	Array

Name	Value
PCFunSel[2:0]	001
PC.Q[15:0]	2
LH	1
Write	0
IROut[15:0]	4c00
RegSel[3:0]	1111
Reg1[15:0]	0807
Reg2[15:0]	0000
Reg3[15:0]	0000
Reg4[15:0]	0000

Figure 12: Fetching AND Executing STR

3.2.5 BX:

The instruction that we tried with was: hex 78 00, RSEL selects R1. Figure 13 shows the simulation of BX.

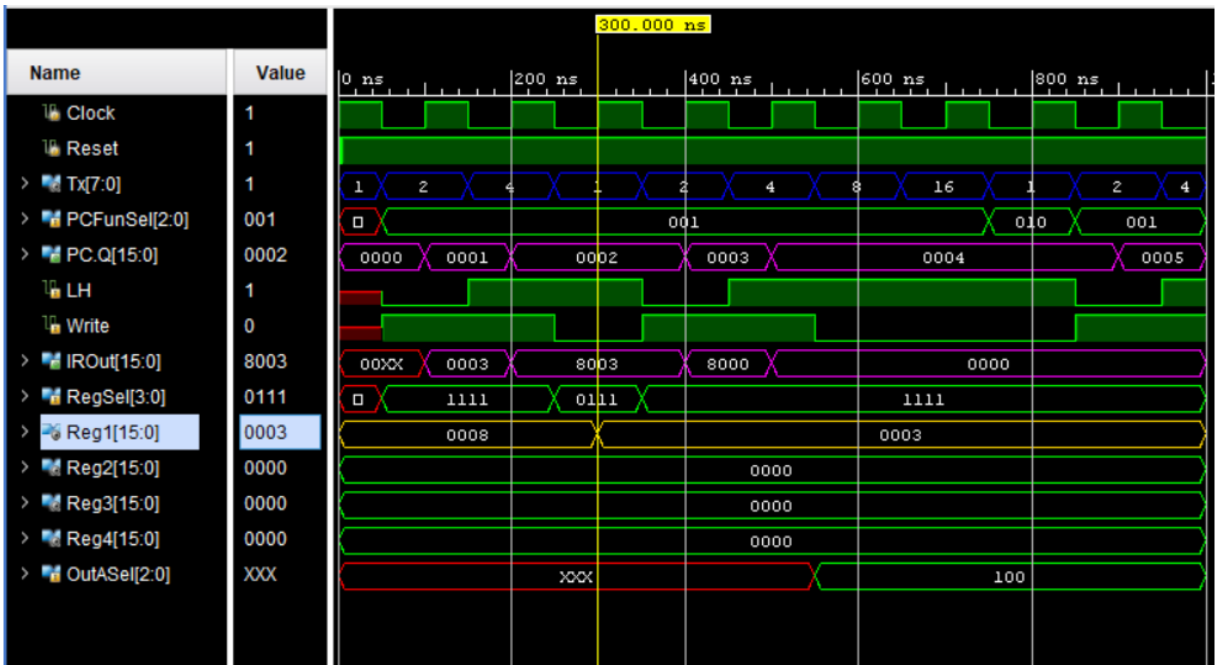


Name	Value
Clock	1
Reset	1
Tx[7:0]	1
PCFunSel[2:0]	010
PC.Q[15:0]	0008
LH	1
Write	0
IROut[15:0]	7800
RegSel[3:0]	1111
Reg1[15:0]	0008
Reg2[15:0]	0000
Reg3[15:0]	0000
Reg4[15:0]	0000
OutASel[2:0]	000

Figure 13: Fetching AND Executing BX

3.2.6 LDRIM:

The instruction that we tried with was: hex 80 03, RSEL selects R1. The simulation results can be seen in the Figure 14.

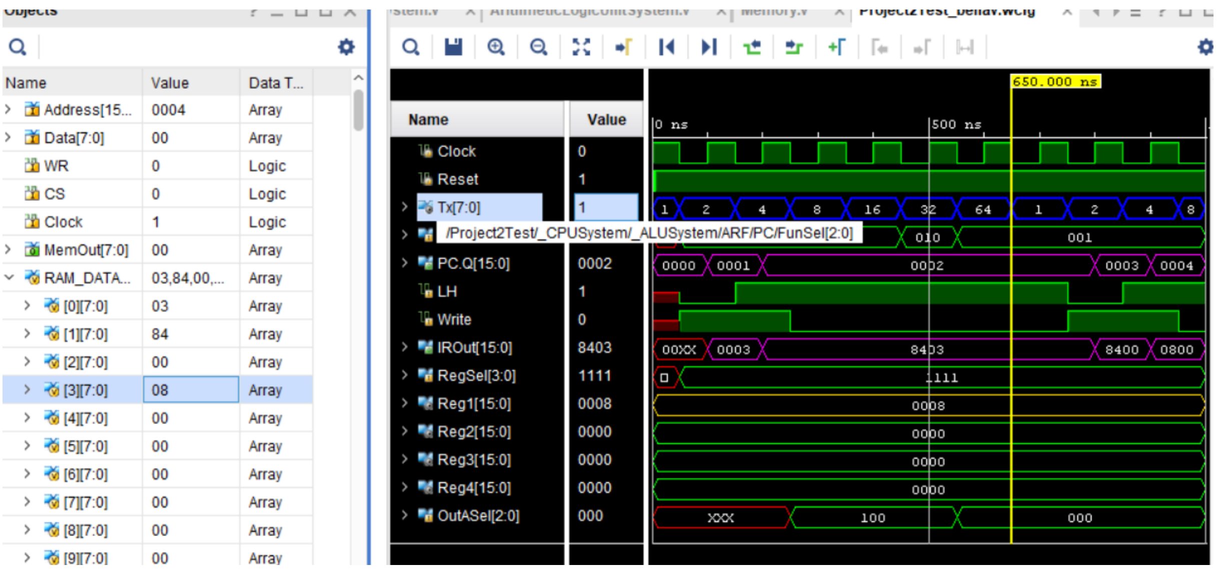


Name	Value
Clock	1
Reset	1
Tx[7:0]	1
PCFunSel[2:0]	001
PC.Q[15:0]	0002
LH	1
Write	0
IROut[15:0]	8003
RegSel[3:0]	0111
Reg1[15:0]	0003
Reg2[15:0]	0000
Reg3[15:0]	0000
Reg4[15:0]	0000
OutASel[2:0]	XXX

Figure 14: Fetching AND Executing LDRIM

3.2.7 STRIM:

The instruction that we tried with was: hex 84 03, RSEL selects R1. The simulation results can be seen in the Figure 15.



Name	Value	Data T...
Address[15...	0004	Array
Data[7:0]	00	Array
WR	0	Logic
CS	0	Logic
Clock	1	Logic
MemOut[7:0]	00	Array
RAM_DATA...	03,84,00,...	Array
[0][7:0]	03	Array
[1][7:0]	84	Array
[2][7:0]	00	Array
[3][7:0]	08	Array
[4][7:0]	00	Array
[5][7:0]	00	Array
[6][7:0]	00	Array
[7][7:0]	00	Array
[8][7:0]	00	Array
[9][7:0]	00	Array

Name	Value
Clock	0
Reset	1
Tx[7:0]	1
/Project2Test/_CPUSystem/_ALUSystem/ARF/PC/FunSel[2:0]	
PC.Q[15:0]	0002
LH	1
Write	0
IROut[15:0]	8403
RegSel[3:0]	1111
Reg1[15:0]	0008
Reg2[15:0]	0000
Reg3[15:0]	0000
Reg4[15:0]	0000
OutASel[2:0]	000

Figure 15: Fetching and Executing STRIM

4 DISCUSSION

4.1 Timing Unit

We created the timing unit as a state machine as we mentioned above. It changes states with the clock's negative edge so that we are having true timing signals.

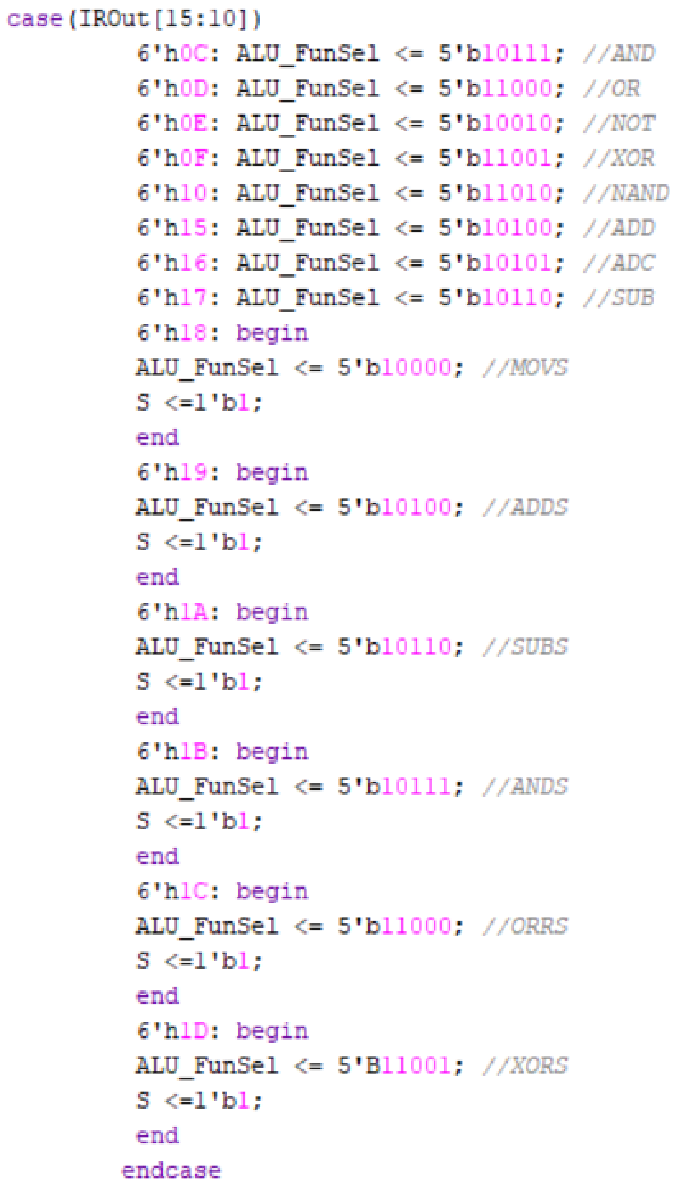
4.2 Fetching and Execution

4.2.1 DSTREG ← SREG1 operation SREG2

We selected SREG1 from RF or ARF in the clock cycle T2. We set the RF_OutASel with the two bits of SREG1 in the instructions without an address reference and 1'b0 as MSB. In this way, the fifth and fourth bits of the SREG1 part in the instructions can select the registers in the RF. Then we can also select the SREG1 from the ARF. We set the OutCSel of the ARF and then we can load the content by passing it through MuxA and setting ScrSel of the RF to S1 register. Finally, S1 was selected with RF_OutASel.

In the clock cycle T3, we selected SREG2 in the same way with SREG1 and the only difference was that we set RF_OutBSel in this time because we loaded SREG2 to the OutB input of the ALU.

In the Clock cycle T4, we set the ALU_FunSel according to required operation and we set the S as 1'b1 which is responsible for changing flags in the flag change required operations. Then we set the MuxASel and MuxBSel for RF and ARF in orderly to select DSTREG from these files. Finally, the content of the result was loaded to DSTREG by adjusting FunSel control inputs of RF and ARF. Here is the ALU_FunSel cases and occurred operations in our code in the Figure 16.



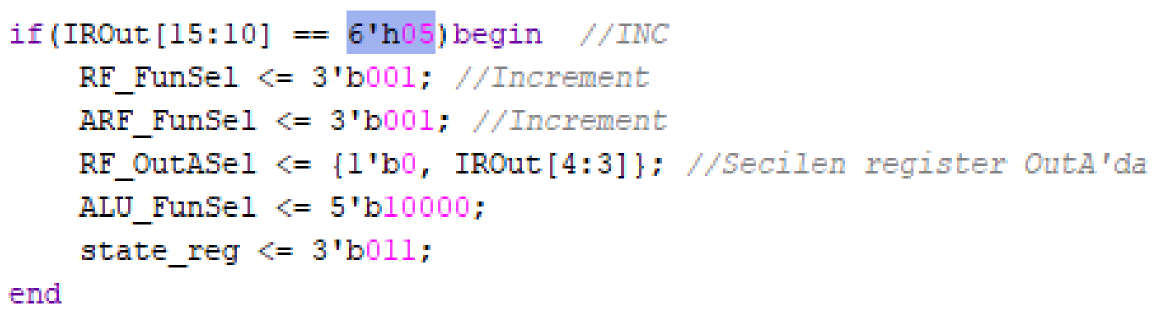
```
case(IROut[15:10])
         6'h0C: ALU_FunSel <= 5'b10111; //AND
         6'h0D: ALU_FunSel <= 5'b11000; //OR
         6'h0E: ALU_FunSel <= 5'b10010; //NOT
         6'h0F: ALU_FunSel <= 5'b11001; //XOR
         6'h10: ALU_FunSel <= 5'b11010; //NAND
         6'h15: ALU_FunSel <= 5'b10100; //ADD
         6'h16: ALU_FunSel <= 5'b10101; //ADC
         6'h17: ALU_FunSel <= 5'b10110; //SUB
         6'h18: begin
         ALU_FunSel <= 5'b10000; //MOVS
         S <=1'b1;
         end
         6'h19: begin
         ALU_FunSel <= 5'b10100; //ADDS
         S <=1'b1;
         end
         6'h1A: begin
         ALU_FunSel <= 5'b10110; //SUBS
         S <=1'b1;
         end
         6'h1B: begin
         ALU_FunSel <= 5'b10111; //ANDS
         S <=1'b1;
         end
         6'h1C: begin
         ALU_FunSel <= 5'b11000; //ORRS
         S <=1'b1;
         end
         6'h1D: begin
         ALU_FunSel <= 5'B11001; //XORS
         S <=1'b1;
         end
        endcase
```

Figure 16: Occurred operations and ALU_FunSel

4.2.2 DSTREG ← operation SREG1

Increment and Decrement Operations

We selected the SREG1 in RF or ARF according to value of the 3 bit SREG1 value in the instructions without an address reference. We set the FunSel control inputs of RF and ARF appropriate to increment and decrement operations. Note that increment and decrement operations were done in the RF or ARF, ALU was not included in that operation. Then we set the Alu_FunSel as 5'b10000 which is equal to Load A operation. We did these in the clock cycle T2. Here is the code snippet for increment operation in the Figure 17.



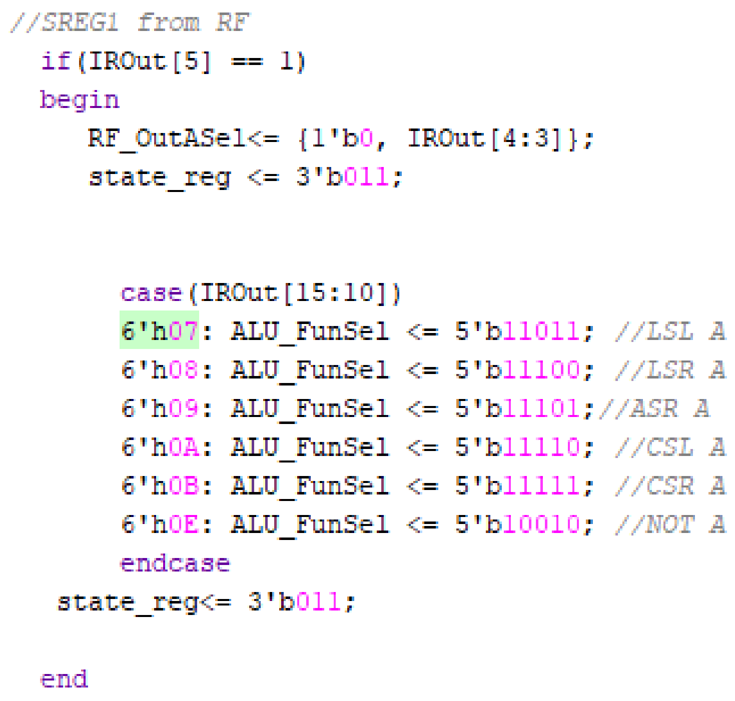
```
if(IROut[15:10] == 6'h05)begin  //INC
    RF_FunSel <= 3'b001; //Increment
    ARF_FunSel <= 3'b001; //Increment
    RF_OutASel <= {1'b0, IROut[4:3]}; //Secilen register OutA'da
    ALU_FunSel <= 5'b10000;
    state_reg <= 3'b011;
end
```

Figure 17: Increment in the clock cycle T2

In the clock cycle T3, we wrote our code by interpreting four possibilities. RF-RF, ARF-RF, RF-ARF, ARF-ARF for DSTREG and SREG1 in orderly. To explain this, SREG1 which is selected in RF or ARF is loaded into RF or ARF.

Other Operations with only SREG1

In the clock cycle T2, we selected SREG1 from RF or ARF and set the ALU_FunSel for required operations. Here is an example in the Figure 18.



```
//SREG1 from RF
  if(IROut[5] == 1)
  begin
     RF_OutASel<= {1'b0, IROut[4:3]};
     state_reg <= 3'b011;


       case(IROut[15:10])
       6'h07: ALU_FunSel <= 5'b11011; //LSL A
       6'h08: ALU_FunSel <= 5'b11100; //LSR A
       6'h09: ALU_FunSel <= 5'b11101;//ASR A
       6'h0A: ALU_FunSel <= 5'b11110; //CSL A
       6'h0B: ALU_FunSel <= 5'b11111; //CSR A
       6'h0E: ALU_FunSel <= 5'b10010; //NOT A
       endcase
   state_reg<= 3'b011;

  end
```

Figure 18: SREG1 from RF

We chose the DSTREG as it would be in the RF or ARF according to ninth bith of the instruction. Then we set the FunSel control inputs of RF and ARF as 3'b010 to load.

4.2.3 BRA:

While conditions are T2 and IROut[15:10]==h'00 : S1← VALUE. We performed this with giving the appropriate signals as MuxASel← 2'b11(IR Out[7:0] was selected to go to S1), RFScrSel←4'b0111 (Only S1 enabled), RFFunSel←3'b010 (Load was selected as RF), RFOutASel ← 3'b100(S1 is at OutA), ARFOutCSel←2'b00;(PC is at OutC at next clock cycle) statereg←3'b011(Go to next state T3). T3 and h'00: PC ← S1 + S2, counter ← 0. so that the value and PC's value added and loaded back to PC. I will not write whole code here. I am explaining the logic lies in the cycles

4.2.4 BNE:

BNE is working same with BRA but it has extra condition for Z=0 being zero. so while writing the if condition, I wrote as: else if(IROut[15:10] =6'h00 —— (IROut[15:10] =6'h01 and Flags[2]= 0) —— (IROut[15:10] =6'h02 and Flags[2] = 1). Here the opcode BRA is 01 and opcode BEQ is 02.

4.2.5 BEQ:

BEQ is working same with BRA but it has extra condition for Z=1 being one. The if condition is already written above.

4.2.6 POP:

T2 and 6'h03: SP $\leftarrow$ SP + 1. was performed having proper signals.

T3 and 6'h03: Rx$\leftarrow$ M[SP] performed for low bits of Rx.in the next cycle it will perform for high bits.

T4 and 6'h03: Rx$\leftarrow$ M[SP] for high bits. and counter zeroed.

4.2.7 PSH:

T2 and 6'h04: M[SP] $\leftarrow$ Rx, for low bits. in the next cycle it will for high bits

T3 and 6'h04: M[SP] $\leftarrow$ Rx, for high bits, SP $\leftarrow$ SP-1 and counter is zeroed because end of instruction.

4.2.8 LDR:

T2 and 6'h12:Rx$\leftarrow$ M[AR] performed for low bits of Rx

T3 and 6'h12: Rx$\leftarrow$ M[AR] performed for high bits of Rx. and also the counter is zeroed because end of instruction.

4.2.9 STR:

T2 and 6'h13: M[AR] $\leftarrow$ Rx, for low bits. In the next cycle it will for high bits.

T3 and 6'h13: M[AR] $\leftarrow$ Rx, for high bits and counter zeroed because end of instruction.

4.2.10 MOVL and MOVH:

T2 and (6'h11 or 6'14): I implemented this in directly one clock cycle. Rx $\leftarrow$ IMMEDIATE(address 8 bit). I selected the RFFunSel according to instruction. If it is 6'h11, high bits are loaded. If it is 6'h14 low bits of Rx is loaded.

4.2.11 BX:

T2 and 6'h1E: M[SP] $\leftarrow$ PC. For this part PC selected and was loaded to S1 then S1 is selected as OutA, simultaneously PC is at ALUOut. Then go to next state "T3".

T3 and 6'h1E: M[SP] $\leftarrow$ PC, now low bits are ready to be performed. then SP is incremented by one because in the next cycle I will put the high bits of PC into that memory block

T4 and 6'h1E: M[SP] $\leftarrow$ PC, high bits of PC is witten to M[SP], now in the next cycle PC $\leftarrow$ Rx will be performed.

T5 and 6'h1E: PC $\leftarrow$ Rx , is performed in one clock cycle. Because of this is the end of instruction, counter is again zeroed.

4.2.12 BL: PC $\leftarrow$ M[SP]

T2 and 6'h1F: The 8 bits from SP is written to PC 7:0 bits.(PC' s low bits). Now go to next cycle.

T3 and 6'h1F: For having the required 8 bit for the high part of PC, I incremented SP by 1 to have the other consecutive memory block. Now go to T4 for having the high bits.

T4 and 6'h1F: Now it can read from the memory and load to PC in this cycle. The instruction end, so counter is set to zero again.

4.2.13 LDRIM : Rx $\leftarrow$ VALUE (VALUE defined in address bits)

We implemented this instruction in only one clock cycle because structure let us to do, so is done int T2. The value from address bits went to Register Files throu MuxA and loaded to the desired register. since it is only 8 bits , I loaded them to LSB bits and I made zero the MSB bits by selecting the RFFunSel : 100. Since it is the end of instruction , I made the counter zero.

4.2.14 STRIM:

T2 and 6'h21: n this cycle , I loaded the AR's values into S1 and put S1 to OutA (for preparing ALU), then went to T3

T3 and 6'h21: In this cycle , I put address bits to S2 and I put S2 to OutB (for having the ALU ADD operation), then went to T4. Since both address and AR's value uses MuxA to reach ALU, I devided them as T2 and T3 cycles as I just explained.

T4 and 6'h21: In this cycle, I wrote the appropriate signals for ALUFunSel MuxBSel ARFRegSel and ARFFunSel to perform the addition and loading it into again AR. Then go to T5

T5 and 6'h21: In this cycle now my memory place is ready to be loaded. First I loaded the low 8 bits of Rx (desired Register) into the memory block. Then I incremented AR by 1 to have the consecutive ememory block in order to load the MSB 8 bits of Rx in next clock cycle.

T6 and 6'h21:In this cycle , I loaded the MSB 8 bits of Rx and then set the counter as zero because the end of instruction it is. So that the "M[AR + OFFSET] $\leftarrow$ Rx" instruction performed successfully.

5 CONCLUSION

Comment on any difficulties you have faced, what you have learned etc.

During the project, we faced with many difficulties. Although we already learned how to write in verilog (because we were experianced from the project 1), at first making the connection from project 1 to project 2 was difficult. We needed to create internal reg types for the wires that ALU System had. After that we needed to implement a sequence counter. At the first try, we attempted to create a module which works with a clock, however it did not work as we expected , it was counting starting from T1, not T0 as we required. Then we changed our approach as making a state machine for our timing unit. Actually , after this stage project became easier, however we still had some challenges. The first challenge for us was implementing the instructions where SREG1 and SREG2 were togather. After several tries, we implemented them with an extra clock cycle for simplicity and working correctly. And while writing each operation, we wrote them on paper then put them on verilog. This blueprint ensure us not to be lost in the code and made the debugging proccess much more easier. All in all, we learned how to implement a CPU Sytem and how it's internal system works with a common clock simultaneously.

REFERENCES