ELEC2602

FINAL EXAM REPORT

By Jiawen Wen 500199055

Introduction

The last digit of my student id is 5, so my assigned instruction is **5**.

Instruction A: **5. ONES Rx**. Count the number of 1's that are stored in Register x. Store the result in Register 0. (e.g. if R1=0x14, ONES R1 would set R0=2)

Instruction B: **5. ONESALL**. Set R0 equal to the total number (converted to binary) of 1's in all registers.

According to the above two instructions, all I need to accomplish is to count the number of 1's used for the value in the specified register, and the number of 1's in all registers. Obviously, instructionA needs to be done by modifying the ALU and FSM. And instructionB is to use the content of instructionA to extend the FSM to achieve. Since instructionA is a new calculation method, it needs to imitate the style of Add to modify ALU and FSM. The instructionB does not need to modify the ALU part, and completely relies on both the state change in the FSM and the content of instructionA to complete.

I have successfully completed the implementation of two instructions. For instructionB, since the original project has 16 registers, it is easy to implement by using a counter in FSM without adding too many states.

Implementation

Instruction A

For instructionA, I don't need to modify the structure of the datapath (see **Appendix 2.1** for full size datapath diagram). Instead, I just need to add one more case to the ALU. At the same time, in the output signal of the FSM, the corresponding ALU control signal needs to be added. The following picture is the modification in ALU (Add each bit together to get the number of 1's):

```
//*****FINAL EXAM******
else if (sel == 3'b101)
   //ones
   aluResult = b[0] + b[1] + b[2] + b[3] + b[4] + b[5] + b[6] + b[7] + b[8] + b[9] + b[10] + b[11] + b[12] + b[13] + b[14] + b[15];
```

(ALU.v)

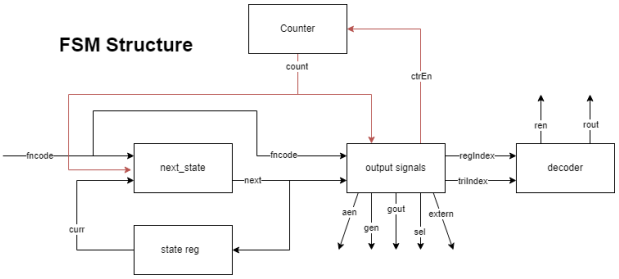
In addition, the corresponding state needs to be added to the FSM to complete the steps of ONES. Then according to the logic of other methods such as Add, for instructionA, I also used 3 steps to implement. (see **Appendix 1.1** for FSM State diagram)

Instruction B

For instructionB, it is essentially to find the number of 1s in each register and add up each value. The specific implementation depends on traversing the register to achieve. Temporarily store the results at the current stage, then calculate the number of 1s in the next register, add it to the temporarily stored result, and repeat the steps until all registers are traversed. For example, the result at the current stage is r0+r1+r2

(total number of 1s), so the number of 1s in r3 will be calculated, and then added to the temporarily stored result, r0+r1+r2+r3 is obtained, and then r4 is the same as r3. The same operation continues until r15. Then store the result in r0, and instructionB is completed.

Since there are 16 registers, for FSM, it is not easy to add states for each register directly and if there are too many registers, the corresponding state is nearly impossible to add. In fact, the operations of traversing registers are the same, so using a counter for traversal reduces the number of states used and simplifies the code structure.



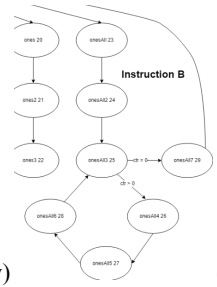
(see **Appendix 1.2** for full size)

According to the above, I added 7 states to complete the operation of instructionB. The first and second steps will calculate the number of 1s in r0 (by using content of instruction A), because before r0, there is no temporary value saved in the system. Steps 3-6 are to traverse operations r1-r15, while step 7 is to output the result (save result to r0) and return to the initial state to wait for the next operation instruction. (see **Appendix 1.1** for FSM State diagram)

Part of the code modifications and fsm states are as follows: (see **Appendix 4, 1.1** for details)

```
//onesAll - r0
5'b10111: begin next <= 5'b11000; end
//onesAll2 - r0
5'b11000: begin next <= 5'b11001; end
//onesAll3
5'b11001: begin
   if (count == 4'b0000)
      //done -> onesAll7
      next <= 5'b11101;
   else
      //continue -> onesAll4
      next <= 5'b11010;
end
//onesAll4
5'b11010: begin next <= 5'b11011; end
```

(NextState.v)



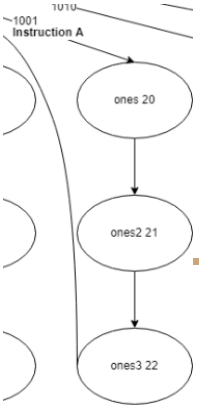
(see **Appendix 1.1** for full FSM State diagram)

Verification Procedure

See **Appendix 1.1** for the complete FSM State diagram.

See **Appendix 2** for the complete Datapath diagram and control signals table.

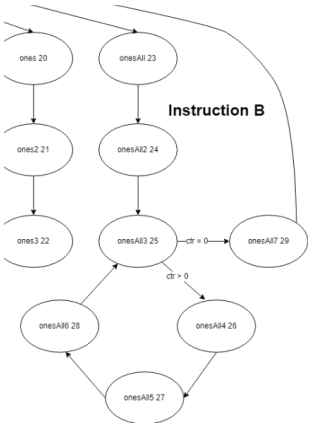
Instruction A



For instructionA, it is the same three steps as Add. In the first step, the tri-buffer of Rx works, transmits the value of Rx to the databus, and temporarily stores it in A. In the second step, the ALU starts to work, calculates the number of 1s in Rx, and temporarily stores the result in G. The third step is to make G's tri-buffer transmit the result to the databus and store the result in R0. After the above three steps are completed, return to the initial state and wait for the next operation.

It should be noted that the first and second steps are actually repeated, because for instructionA, the ALU only needs to obtain the value of one register, but in order to be consistent with other methods, a total of three states are used here. (can actually be reduced to two states)

Instruction B



The first step is to enable the tri-buffer of R0, and use the ALU to calculate the number of 1s and store the result in G. The second step is to enable the tri-buffer of G and store the result in A.

The third step is to determine whether to traverse all the registers. If the traverse is completed, go to the seventh step, and if not, continue to the fourth step. Step 4 is to calculate the number of 1s in Rx and store the result in G. Step 5, enable the tri-buffer of G, pass the value of G into the databus, and use ALU to add A and G, and save the sum in G. Step 6, enable the tri-buffer of G, cache the value in G into A, then go back to step 3 and do the same for the next register. (From R1 to R15)

In step 7, the total number of 1s of all the registers is stored in R0, and the initial stage is returned. After completing the above 7 steps, an operation of instructionB is completed.

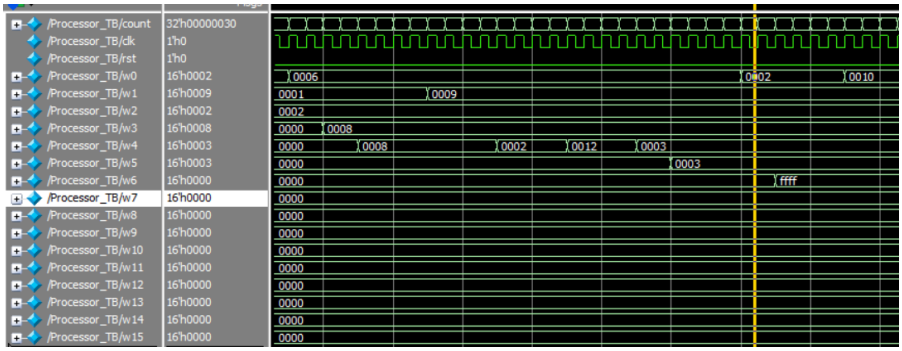
Results

Note that the project uses memory to store instructions and data, so please refer to the **ProMemory.v** file for specific test steps (see **Appendix 3.1**). Among them, steps 0-13 are used for the assignment, and steps 14-24 are used to test two new instructions. See **Appendix 3.2** for full test results.

First, I tested if instructionA is correct. In step 14, check the number of 1s in R5. In step 15-16, assign 0xffff to R6, and check the number of 1s in R6.

```
//ones r5
14: begin fncode = 16'b1001010100000000; data = 16'b0000000000000000; end
//load r6 16 1s
15: begin fncode = 16'b0000011000000000; data = 16'b1111111111111111; end
//ones r6
16: begin fncode = 16'b1001011000000000; data = 16'b0000000000000000; end
//onesAll
17: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
```

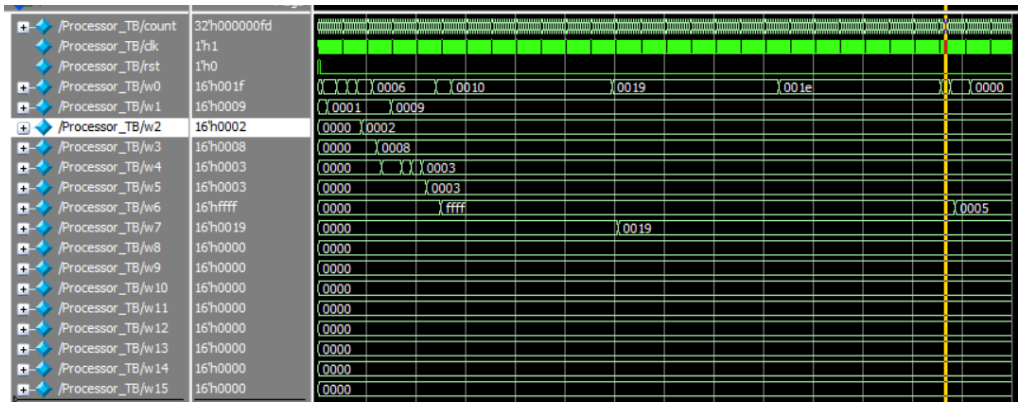
In the figure below, R0 first changed from 6 to 2 (R5 is 0x3). Then it becomes 16 (R6 is 0xffff). Step 16 is used to determine if the ALU can count every 1 in the register. (w0-w15 represent value of each register)



Then three consecutive instructionBs are performed, and after the first instructionB ends, the R0 value is copied into R7. These steps determine if multiple instructionBs can be performed, and if the state returns to the initial state after the end to wait for next.

```
//onesAll
17: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
//move r7 r0
18: begin fncode = 16'b0001011100000000; data = 16'b0000000000000000; end
//onesAll
19: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
//onesAll
20: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
```

As shown in the figure below, R0 changed from 16 to 25 (R0=16, R1=9, R2=2, R3=8, R4=3, R5=3, R6=0xffff, R7-15=0). Then to 30 (R0=25, R7=25), and finally to 31 (R0=30). (The parentheses represent the value of each register at the start of every instructionB.)



The benefits of creating independent instructionB are obvious. No need to add overly complex looping logic to the ALU. The modification of each part has no effect on the entire project, such as no need to change the parameters of Datapath.

Conclusion

The logic of InstructionB is to combine Add and instructionA, and use counter loop to reduce unnecessary operations. Compared to directly reusing instructionA, instructionB only needs four states for each register, that is, four clock cycles. Using instructionA alone to complete the requirements of instructionB requires 6 cycles per register (add 3 states + instructionA 3 states). So instructionB is faster than a sequence of instructions from using the original instruction set supplemented with instruction A.

In addition, by using the counter loop, instructionB avoids unnecessary repeated state usage and code repetition due to too many registers. But the disadvantage is that since the processor has 16 registers, FSM cleverly uses a simple 4-bit counter (from 0-15), which leads to the inevitable need to modify some logic inside the counter if new registers are added.

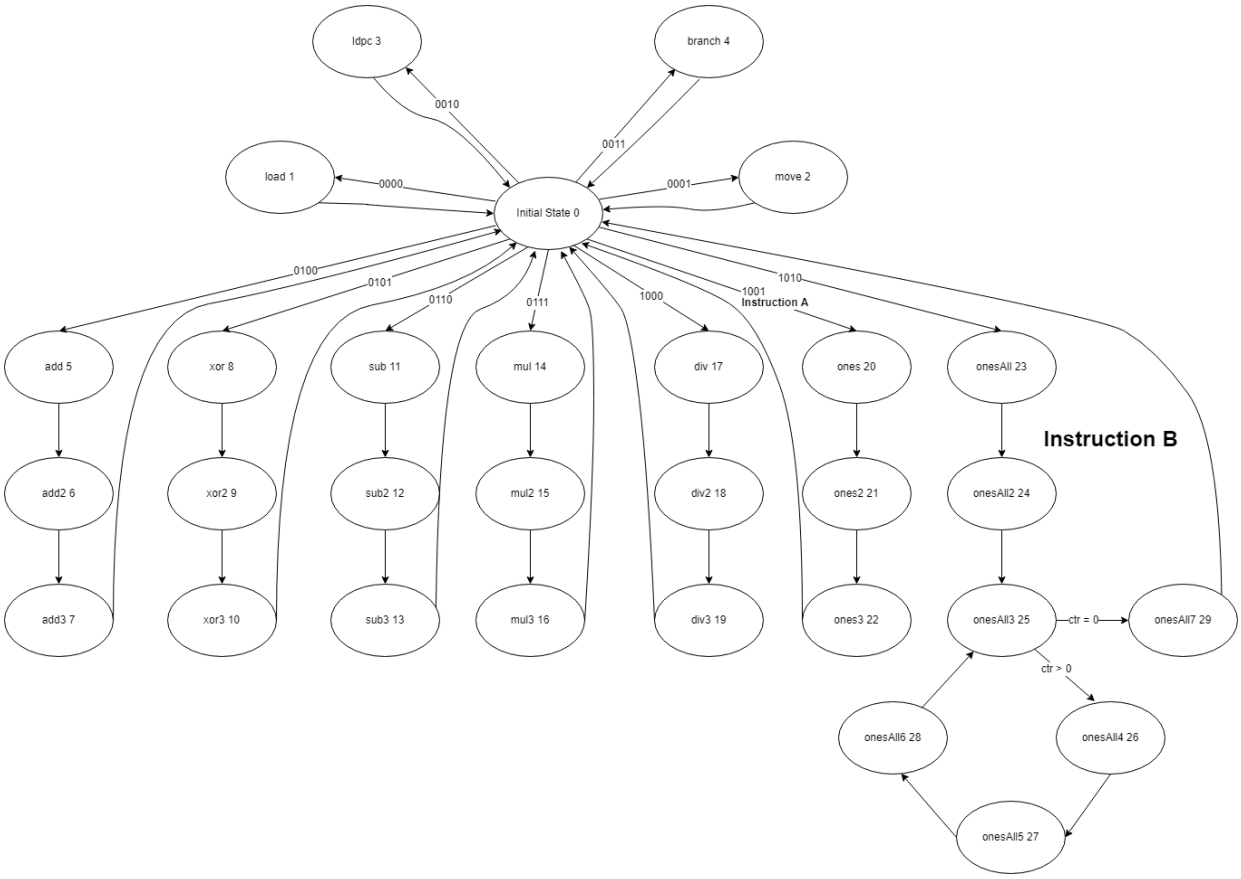
Reference

No reference.

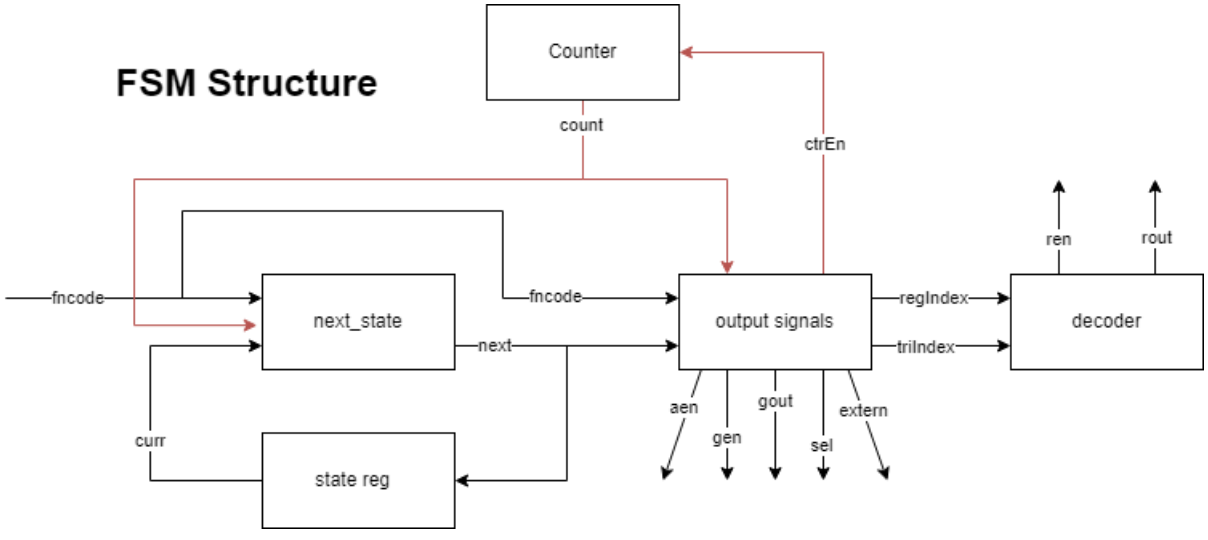
Appendix

1. FSM

1.1 FSM States Diagram (30 states in total)

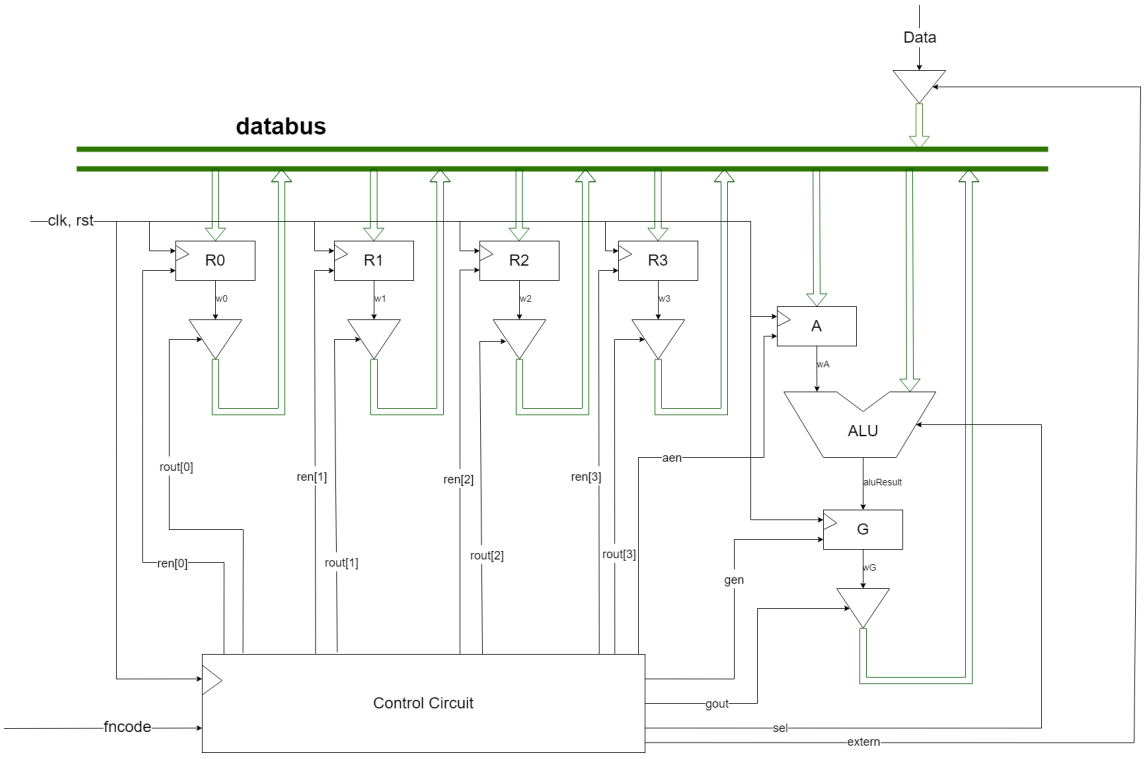


1.2 FSM Structure



2. Datapath

2.1 Datapath Diagram (There are actually 16 registers, the following picture uses R0-R3 for better view)



2.2 Control Signals Table

Signals	Effects
w0 - w15	value of R0-R15
ren[0] - ren[15]	signal that enables R0-R15 to save value from databus
rout[0] - rout[15]	signal that enables Tri-Buffer to output from corresponding register
aen	signal that enables data enter A
wA	value of A
aluResult	result calculated by ALU
gen	signal that enables G to save result from aluResult
gout	signal that enables Tri-Buf to output to databus
wG	value of G
clk	clock signal

rst	reset signal
sel	3-bit signal for ALU operations
extern	signal that enables data enter the databus

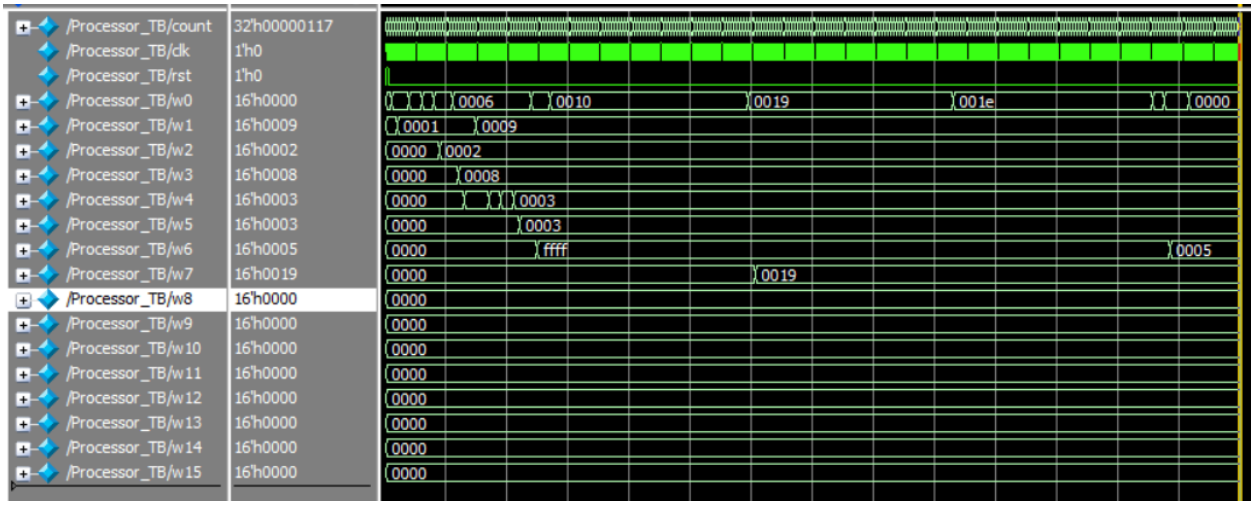
3. Testbench

3.1 Testbench Instructions (ProMemory.v)

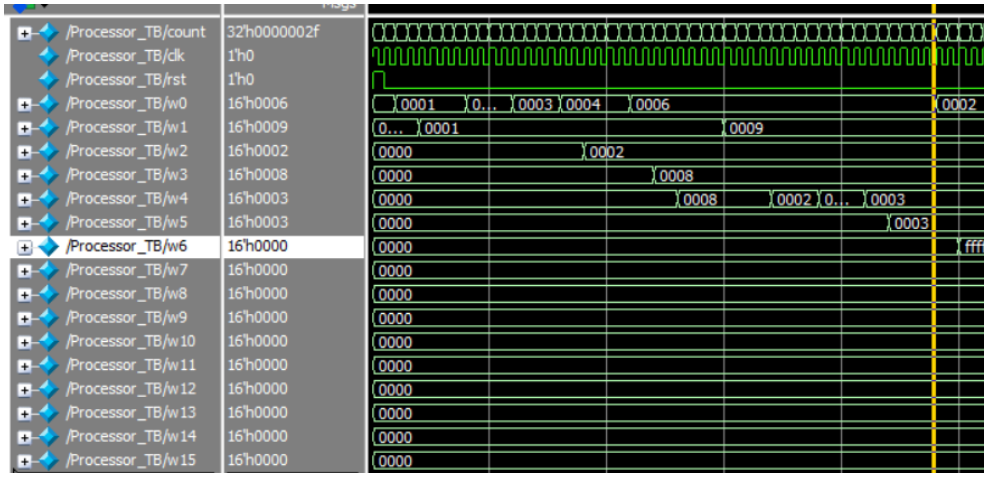
```
41          //************************FINAL EXAM MODIFICATION************************
42          //ones r5
43          14: begin fncode = 16'b1001010100000000; data = 16'b0000000000000000; end
44          //load r6 16 1s
45          15: begin fncode = 16'b0000011000000000; data = 16'b1111111111111111; end
46          //ones r6
47          16: begin fncode = 16'b1001011000000000; data = 16'b0000000000000000; end
48          //onesAll
49          17: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
50          //move r7 r0
51          18: begin fncode = 16'b0001011100000000; data = 16'b0000000000000000; end
52          //onesAll
53          19: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
54          //onesAll
55          20: begin fncode = 16'b1010000000000000; data = 16'b0000000000000000; end
56          //ones r0
57          21: begin fncode = 16'b1001000000000000; data = 16'b0000000000000000; end
58          //move r6 r0
59          22: begin fncode = 16'b0001011000000000; data = 16'b0000000000000000; end
60          //load r8 0
61          23: begin fncode = 16'b0000100000000000; data = 16'b0000000000000000; end
62          //ones r8
63          24: begin fncode = 16'b1001100000000000; data = 16'b0000000000000000; end
64
```

3.2 Simulation Result (Testbench file is Processor_TB.v)

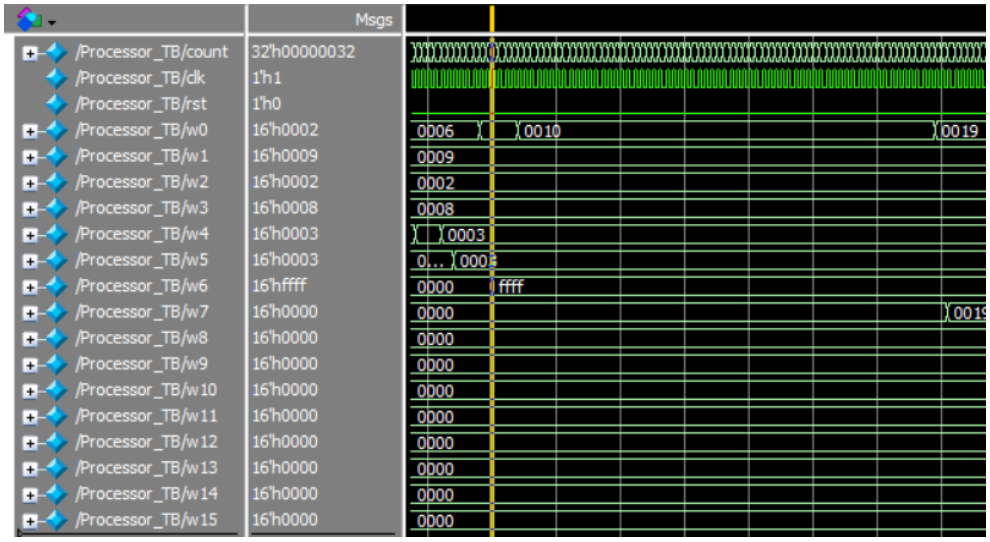
Overall waveform



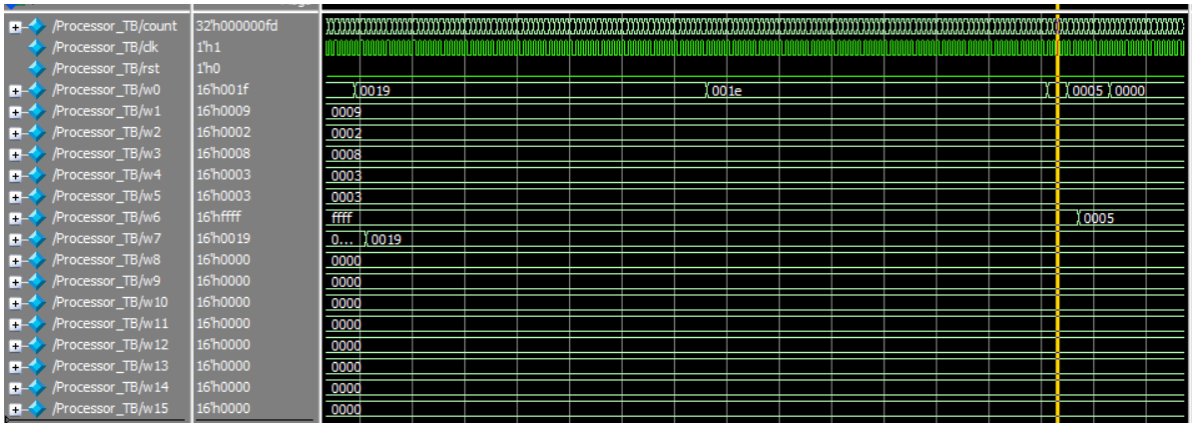
Step 0 - 13



Step 14 - 17



Step 18 - 24



4. Code Modification Snippets

FSM.v (counter added for Instruction B)

FSM.v

```
1  //state count for ONESALL
2  //***********FINAL EXAM MODIFICATION**********
3  module Counter(clk, rst, ctrEn, count);
4     //clk, rst
5     input clk, rst;
6
7     //enable signal for count to increment
8     input ctrEn;
9
10    //count
11    reg [3:0] ctrTemp;
12
13    //ouput current state
14    output [3:0] count;
15
16    always @(posedge clk or posedge rst) begin
17       if (rst == 1'b1)
18          //rst
19          ctrTemp <= 0;
20       else if (ctrEn == 1'b1)
21          //increment
22          ctrTemp <= ctrTemp + 1;
23    end
24
25    //output
26    assign count = ctrTemp;
27
28 endmodule
```

NextState.v (new states for Instruction A & B)

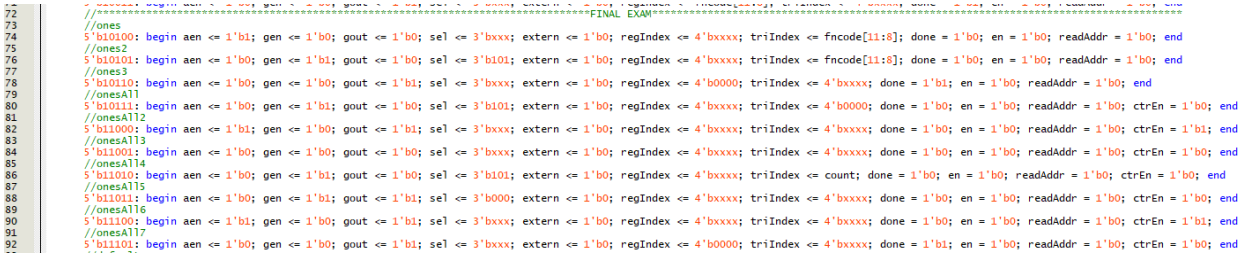
NextState.v

```
88        5'b10010: begin next <= 5'b10011; end
89        //div3
90        5'b10011: begin next <= 5'b00000; end
91        //************FINAL EXAM*************
92        //ones
93        5'b10100: begin next <= 5'b10101; end
94        //ones2
95        5'b10101: begin next <= 5'b10110; end
96        //ones3
97        5'b10110: begin next <= 5'b00000; end
98        //onesAll - r0
99        5'b10111: begin next <= 5'b11000; end
100       //onesAll2 - r0
101       5'b11000: begin next <= 5'b11001; end
102       //onesAll3
103       5'b11001: begin
104          if (count == 4'b0000)
105             //done -> onesAll7
106             next <= 5'b11101;
107          else
108             //continue -> onesAll4
109             next <= 5'b11010;
110       end
111       //onesAll4
112       5'b11010: begin next <= 5'b11011; end
113       //onesAll5
114       5'b11011: begin next <= 5'b11100; end
115       //onesAll6 -> onesAll3 cycle r1-r15
116       5'b11100: begin next <= 5'b11001; end
117       //onesAll7 -> initial state
118       5'b11101: begin next <= 5'b00000; end
```

NextState.v

```
43          else if (fncode[15:12] == 4'b1000)
44             next <= 5'b10001;
45          //***********FINAL EXAM***********
46          //ones
47          else if (fncode[15:12] == 4'b1001)
48             next <= 5'b10100;
49          //onesAll
50          else if (fncode[15:12] == 4'b1010)
51             next <= 5'b10111;
```

OutputSig.v (new control signal states for Instruction A & B)



```
//*************************************************************************FINAL EXAM*************************************************************************
//ones
5'b10100: begin aen <= 1'b1; gen <= 1'b0; gout <= 1'b0; sel <= 3'bxxx; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= fncode[11:8]; done = 1'b0; en = 1'b0; readAddr = 1'b0; end
//ones2
5'b10101: begin aen <= 1'b0; gen <= 1'b1; gout <= 1'b0; sel <= 3'b101; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= fncode[11:8]; done = 1'b0; en = 1'b0; readAddr = 1'b0; end
//ones3
5'b10110: begin aen <= 1'b0; gen <= 1'b0; gout <= 1'b1; sel <= 3'bxxx; extern <= 1'b0; regIndex <= 4'b0000; triIndex <= 4'bxxxx; done = 1'b1; en = 1'b0; readAddr = 1'b0; end
//onesAll
5'b10111: begin aen <= 1'b0; gen <= 1'b1; gout <= 1'b0; sel <= 3'b101; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= 4'b0000; done = 1'b0; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b0; end
//onesAll2
5'b11000: begin aen <= 1'b1; gen <= 1'b0; gout <= 1'b1; sel <= 3'bxxx; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= 4'bxxxx; done = 1'b0; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b1; end
//onesAll3
5'b11001: begin aen <= 1'b0; gen <= 1'b0; gout <= 1'b0; sel <= 3'bxxx; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= 4'bxxxx; done = 1'b0; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b0; end
//onesAll4
5'b11010: begin aen <= 1'b0; gen <= 1'b1; gout <= 1'b0; sel <= 3'b101; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= count; done = 1'b0; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b0; end
//onesAll5
5'b11011: begin aen <= 1'b0; gen <= 1'b1; gout <= 1'b1; sel <= 3'b000; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= 4'bxxxx; done = 1'b0; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b0; end
//onesAll6
5'b11100: begin aen <= 1'b1; gen <= 1'b0; gout <= 1'b1; sel <= 3'bxxx; extern <= 1'b0; regIndex <= 4'bxxxx; triIndex <= 4'bxxxx; done = 1'b0; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b1; end
//onesAll7
5'b11101: begin aen <= 1'b0; gen <= 1'b0; gout <= 1'b1; sel <= 3'bxxx; extern <= 1'b0; regIndex <= 4'b0000; triIndex <= 4'bxxxx; done = 1'b1; en = 1'b0; readAddr = 1'b0; ctrEn = 1'b0; end
```

ALU.v (new case for Instruction A)

```
    //*****FINAL EXAM*****
    else if (sel == 3'b101)
        //ones
        aluResult = b[0] + b[1] + b[2] + b[3] + b[4] + b[5] + b[6] + b[7] + b[8] + b[9] + b[10] + b[11] + b[12] + b[13] + b[14] + b[15];
end
```