ALU and Register File Report

Jonathan Pilling, Steen Sia, Stephan Stankovic, Jeremy Wu

September 18, 2018

Abstract

In labs 1 and 2 the team had to design and create the ALU and Register File, which will be used as a base for our processor. These two pieces are key to allow the team to create a processor and eventually use that processor to create some sort of application.

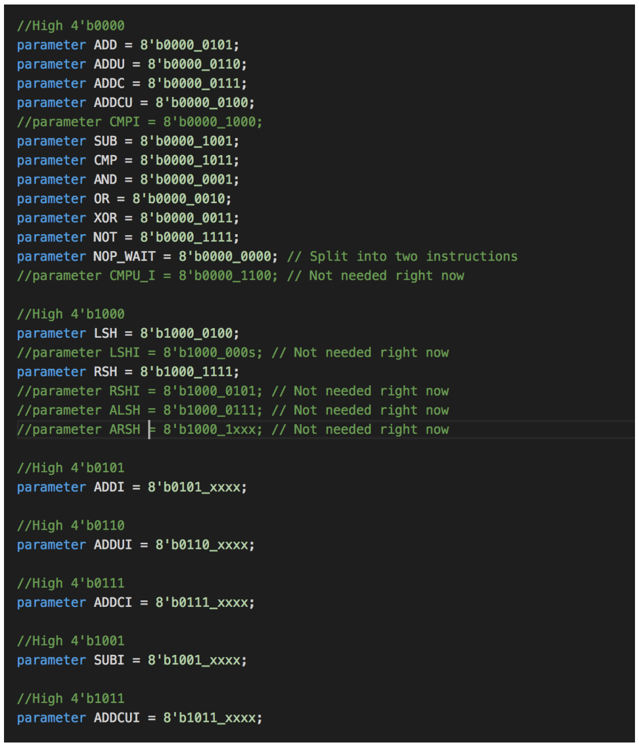
1 Introduction and Objective

Our job was to create an ALU and Register file combination that would allow us to read from two registers and pass that into the ALU created in Lab 1. The ALU is simple combinational logic that does not depend on a clock, while our Register file does require us to work on a clock. The ALU will compute basic arithmetic, logical operations, and shifts on bits. The Register file on the other hand is simply a place for us to store output and to allow our ALU to read from certain locations. The objective of these two labs is to create the basis on which we will continue to build our processor up from. Next steps will be to create memory and integrate that into our current circuit and then create an instruction decoder.

2 Design Organization

2.1 ALU Design

In the process of designing the ALU, we have determined the need to implement the following instructions: ADD, ADDI, ADDU, ADDUI, ADC, ADDCU, ADDCUI, ADDCI, SUB, SUBI, CMP, CMPI, CMPU/I, AND, OR, XOR, NOT, LSH, LSHI, RSH, RSHI, ALSH, ARSH, NOP/WAIT. However, some of the instructions: CMPI, CMPU/I, LSHI, RSHI, and OP/WAIT have been delayed for the later stages of the lab. This list can be seen in Figure 1

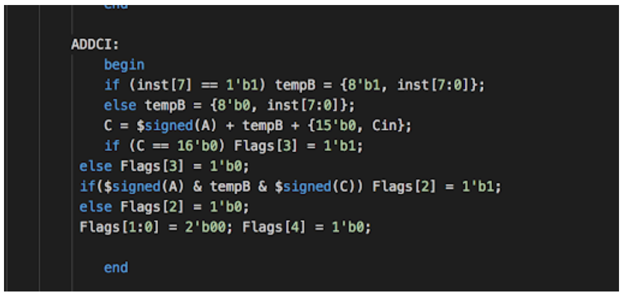


```
//High 4'b0000
parameter ADD = 8'b0000_0101;
parameter ADDU = 8'b0000_0110;
parameter ADDC = 8'b0000_0111;
parameter ADDCU = 8'b0000_0100;
//parameter CMPI = 8'b0000_1000;
parameter SUB = 8'b0000_1001;
parameter CMP = 8'b0000_1011;
parameter AND = 8'b0000_0001;
parameter OR = 8'b0000_0010;
parameter XOR = 8'b0000_0011;
parameter NOT = 8'b0000_1111;
parameter NOP_WAIT = 8'b0000_0000; // Split into two instructions
//parameter CMPU_I = 8'b0000_1100; // Not needed right now

//High 4'b1000
parameter LSH = 8'b1000_0100;
//parameter LSHI = 8'b1000_000s; // Not needed right now
parameter RSH = 8'b1000_1111;
//parameter RSHI = 8'b1000_0101; // Not needed right now
//parameter ALSH = 8'b1000_0111; // Not needed right now
//parameter ARSH = 8'b1000_1xxx; // Not needed right now

//High 4'b0101
parameter ADDI = 8'b0101_xxxx;

//High 4'b0110
parameter ADDUI = 8'b0110_xxxx;

//High 4'b0111
parameter ADDCI = 8'b0111_xxxx;

//High 4'b1001
parameter SUBI = 8'b1001_xxxx;

//High 4'b1011
parameter ADDCUI = 8'b1011_xxxx;
```

Figure 1: Parameter List of all Opcodes/Instructions that we have and will implement

Using the operation codes given in the ISA Manual we were able to assign all these operations 8 value bit vectors. For our ALU.v, we have set up 16-bit vectors for input A, B, and the instructions to be interpreted. With the 16-bit vector for instructions, we are utilizing the higher 4 bits and the 2nd lowest 4 bits of the instruction vector to be sent into the 8-bit vector Opcode. In reference to the ISA for x86, we have determined to use extra 7 unused Opcodes for the operations that didn't have specific bits.

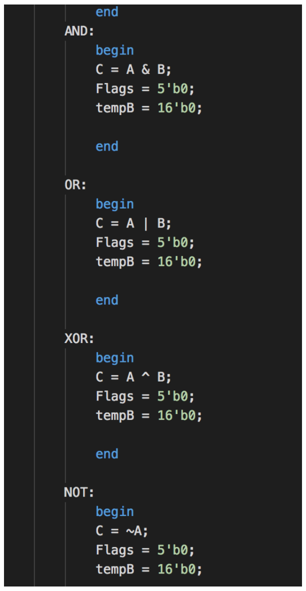
First, we implemented all the add instructions and utilized the $signed function for ADD, ADDI, ADC, and ADDCI since they deal with 2's complement arithmetic. During this process, we encountered a bunch of arithmetic errors in our test bench for the immediate operations. We later found out that manual sign extension was necessary since the compiler automatically sign extends with 0 bits. To solve this issue, we used a temporary register to hold the immediate value that is being passed in and manually sign extended the 8 bits (1 or 0 depending on the MSB of immediate that's passed in). The same process was implemented for the immediate subtraction instructions.



```
    ADDCI:
        begin
        if (inst[7] == 1'b1) tempB = {8'b1, inst[7:0]};
        else tempB = {8'b0, inst[7:0]};
        C = $signed(A) + tempB + {15'b0, Cin};
        if (C == 16'b0) Flags[3] = 1'b1;
      else Flags[3] = 1'b0;
      if($signed(A) & tempB & $signed(C)) Flags[2] = 1'b1;
      else Flags[2] = 1'b0;
      Flags[1:0] = 2'b00; Flags[4] = 1'b0;

        end
```

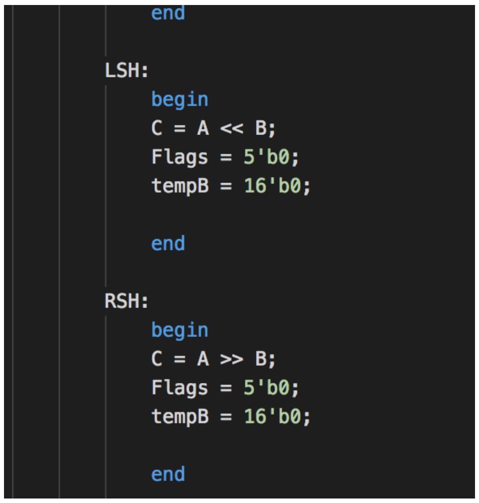
Figure 2: ADDCI Instruction Example of Appropriately Sign Extending Bits

Moving on, the simpler part of the ALU design was implementing the logical operations. The logical operations: AND, OR, XOR, utilize the bitwise operators &, —, and ˆrespectively. With these three instructions, the ordering of A and B does not matter when applying the appropriate bitwise operator. However, the following instructions: NOT, LSH, and RSH work slightly different. For the NOT instruction, we are ignoring B and inverting the bits of A only. For LSH and RSH, the first register A is being shifted left(¡¡), right(¿¿) respectively by register B. Not to mention of the instructions have their appropriate flags set. Our Flags[4:0] are set such that Carry Flag(C), represents the least significant bit and the Negative Flag(N), represents the most significant bit.



```
        end
    AND:
        begin
        C = A & B;
        Flags = 5'b0;
        tempB = 16'b0;

        end

    OR:
        begin
        C = A | B;
        Flags = 5'b0;
        tempB = 16'b0;

        end

    XOR:
        begin
        C = A ^ B;
        Flags = 5'b0;
        tempB = 16'b0;

        end

    NOT:
        begin
        C = ~A;
        Flags = 5'b0;
        tempB = 16'b0;
```

Figure 3: Implementation of Logical Operations/Instructions



```
        end

    LSH:
        begin
        C = A << B;
        Flags = 5'b0;
        tempB = 16'b0;

        end

    RSH:
        begin
        C = A >> B;
        Flags = 5'b0;
        tempB = 16'b0;

        end
```

Figure 4: Implementation of Shift Operations/Instructions

2.2 Register File Design

The register bank was the first module and first step in creating the Datapath wrapper. This module was able to instantiate registers inside of itself and map these registers as outputs to the ALU. This register bank module takes in a clock signal, register enable signal for writing, and ALU bus output. This ALU bus output is the value that's potentially written to a register. Wiring up this register bank to the ALU, we needed to have a buffer in between the 16 registers and the two inputs for the ALU. This way we were able to decide which two inputs we wanted for different computations.

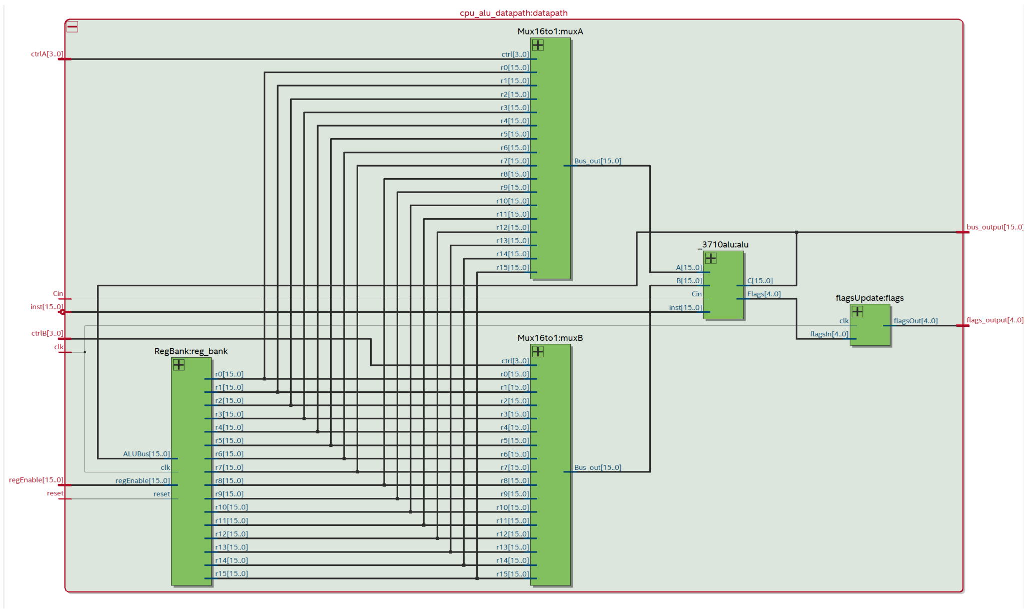
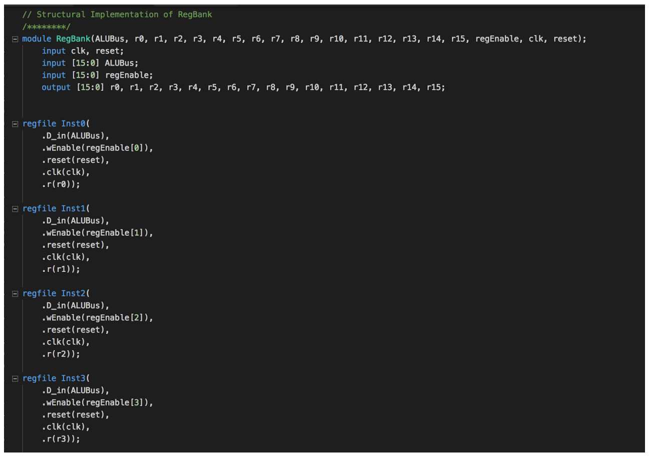


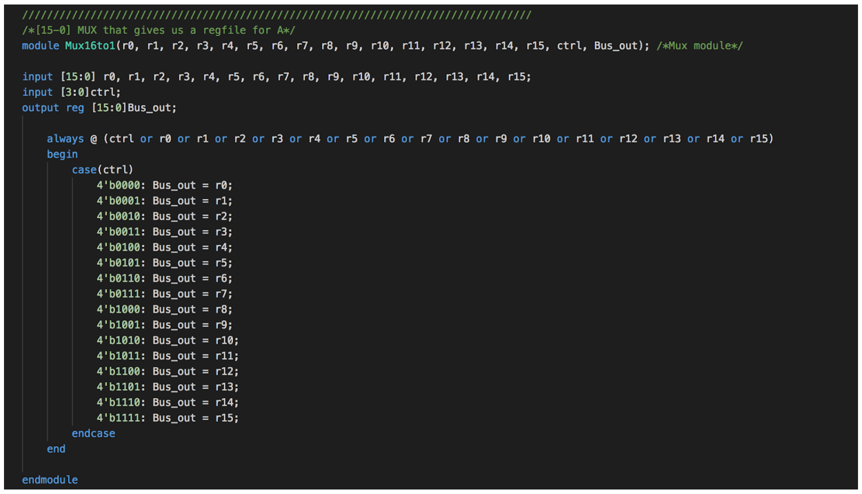
Figure 5: All of our wiring that is done



```
// Structural Implementation of RegBank
/********/
module RegBank(ALUBus, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, regEnable, clk, reset);
    input clk, reset;
    input [15:0] ALUBus;
    input [15:0] regEnable;
    output [15:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15;


regfile Inst0(
    .D_in(ALUBus),
    .wEnable(regEnable[0]),
    .reset(reset),
    .clk(clk),
    .r(r0));

regfile Inst1(
    .D_in(ALUBus),
    .wEnable(regEnable[1]),
    .reset(reset),
    .clk(clk),
    .r(r1));

regfile Inst2(
    .D_in(ALUBus),
    .wEnable(regEnable[2]),
    .reset(reset),
    .clk(clk),
    .r(r2));

regfile Inst3(
    .D_in(ALUBus),
    .wEnable(regEnable[3]),
    .reset(reset),
    .clk(clk),
    .r(r3));
```

Figure 6: Register Bank Module in regfile.v

To solve this issue we designed a 16 to 1 multiplexor as seen in the next page, that took in all of our registers as inputs. The mux control signal would allow one register value to pass through in the ALU. We designed a mux_a and a mux_b in our total Wrapper.v file to connect all the inputs from the register bank into ALU inputs A and B.



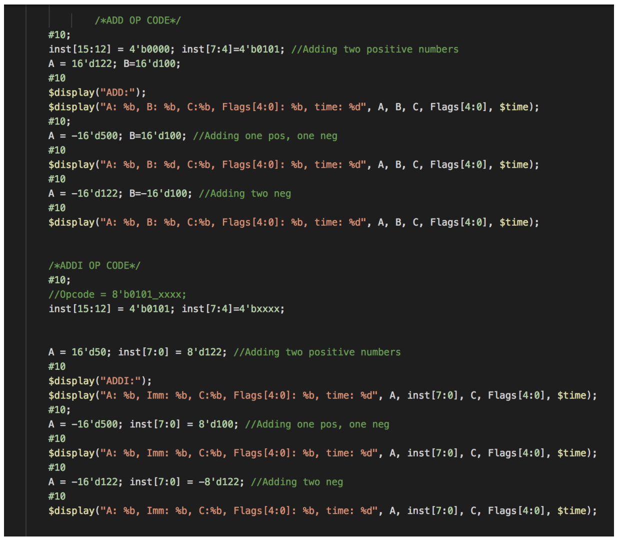
```
//////////////////////////////////////////////////////////////////////////////////
/*[15-0] MUX that gives us a regfile for A*/
module Mux16to1(r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, ctrl, Bus_out); /*Mux module*/

input [15:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15;
input [3:0]ctrl;
output reg [15:0]Bus_out;

    always @ (ctrl or r0 or r1 or r2 or r3 or r4 or r5 or r6 or r7 or r8 or r9 or r10 or r11 or r12 or r13 or r14 or r15)
    begin
        case(ctrl)
            4'b0000: Bus_out = r0;
            4'b0001: Bus_out = r1;
            4'b0010: Bus_out = r2;
            4'b0011: Bus_out = r3;
            4'b0100: Bus_out = r4;
            4'b0101: Bus_out = r5;
            4'b0110: Bus_out = r6;
            4'b0111: Bus_out = r7;
            4'b1000: Bus_out = r8;
            4'b1001: Bus_out = r9;
            4'b1010: Bus_out = r10;
            4'b1011: Bus_out = r11;
            4'b1100: Bus_out = r12;
            4'b1101: Bus_out = r13;
            4'b1110: Bus_out = r14;
            4'b1111: Bus_out = r15;
        endcase
    end

endmodule
```

Figure 7: 16-to-1 Multiplexor to control inputs into the ALU

3 Testing Strategies

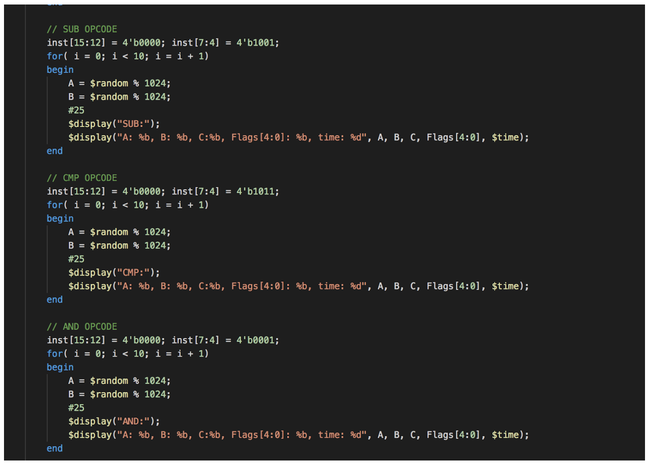
3.1 Testing the ALU

Testing the ALU we used two test benches that would simulate op codes, A inputs, B inputs, and Cin signals. These inputs would take assigned values and output a C value after the computation, and the generated flag signals. We wrote test benches that tested for a lot of corner case scenarios, and we wrote test benches that generated random number for different operation codes. We included lengthy tests that would generate a couple different inputs for each instruction. During testing we were able to troubleshoot some signed arithmetic error for signed operations and fix these issues accordingly.



```
        /*ADD OP CODE*/
#10;
inst[15:12] = 4'b0000; inst[7:4]=4'b0101; //Adding two positive numbers
A = 16'd122; B=16'd100;
#10
$display("ADD:");
$display("A: %b, B: %b, C:%b, Flags[4:0]: %b, time: %d", A, B, C, Flags[4:0], $time);
#10;
A = -16'd500; B=16'd100; //Adding one pos, one neg
#10
$display("A: %b, B: %d, C:%b, Flags[4:0]: %b, time: %d", A, B, C, Flags[4:0], $time);
#10
A = -16'd122; B=-16'd100; //Adding two neg
#10
$display("A: %b, B: %b, C:%b, Flags[4:0]: %b, time: %d", A, B, C, Flags[4:0], $time);


/*ADDI OP CODE*/
#10;
//Opcode = 8'b0101_xxxx;
inst[15:12] = 4'b0101; inst[7:4]=4'bxxxx;


A = 16'd50; inst[7:0] = 8'd122; //Adding two positive numbers
#10
$display("ADDI:");
$display("A: %b, Imm: %b, C:%b, Flags[4:0]: %b, time: %d", A, inst[7:0], C, Flags[4:0], $time);
#10;
A = -16'd500; inst[7:0] = 8'd100; //Adding one pos, one neg
#10
$display("A: %b, Imm: %b, C:%b, Flags[4:0]: %b, time: %d", A, inst[7:0], C, Flags[4:0], $time);
#10
A = -16'd122; inst[7:0] = -8'd122; //Adding two neg
#10
$display("A: %b, Imm: %b, C:%b, Flags[4:0]: %b, time: %d", A, inst[7:0], C, Flags[4:0], $time);
```

Figure 8: Examples of Corner Case Testing in ALU Test Bench



```
// SUB OPCODE
inst[15:12] = 4'b0000; inst[7:4] = 4'b1001;
for( i = 0; i < 10; i = i + 1)
begin
    A = $random % 1024;
    B = $random % 1024;
    #25
    $display("SUB:");
    $display("A: %b, B: %b, C:%b, Flags[4:0]: %b, time: %d", A, B, C, Flags[4:0], $time);
end

// CMP OPCODE
inst[15:12] = 4'b0000; inst[7:4] = 4'b1011;
for( i = 0; i < 10; i = i + 1)
begin
    A = $random % 1024;
    B = $random % 1024;
    #25
    $display("CMP:");
    $display("A: %b, B: %b, C:%b, Flags[4:0]: %b, time: %d", A, B, C, Flags[4:0], $time);
end

// AND OPCODE
inst[15:12] = 4'b0000; inst[7:4] = 4'b0001;
for( i = 0; i < 10; i = i + 1)
begin
    A = $random % 1024;
    B = $random % 1024;
    #25
    $display("AND:");
    $display("A: %b, B: %b, C:%b, Flags[4:0]: %b, time: %d", A, B, C, Flags[4:0], $time);
end
```

Figure 9: Examples of Random Testing in ALU test bench

3.2 Testing the Register File

To test the Wrapper we implemented a Finite State Machine that instantiated the wrapper and applied some register reads and writes. One of the Finite State Machines we implemented was the Fibonacci Sequence. This would set the value initially, (hard-coded) into r0 and r1, to zero and one. After these values were set we would go through different states to add r0 and r1, and write the value into r2. Then we would perform r1+ r2 and write the value into r3.

This would continue until we hit the end of our register bank. The final value of r15 being equal to the 15th number in the Fibonacci sequence, 610. Putting this value onto the board we got utilized the Hex to 7_Seg display module. This would display the end result of our Fibonacci state machine in hexadecimal.

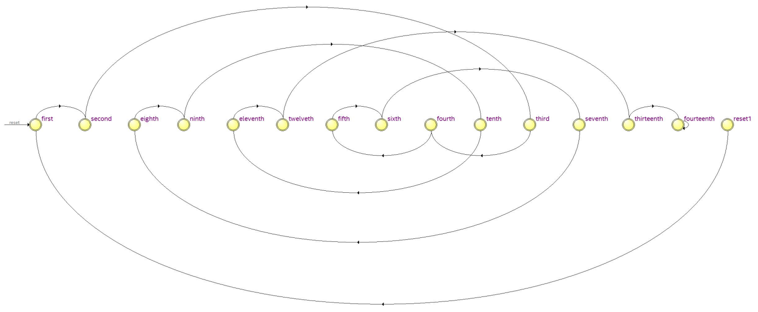


Figure 10: States of our FSM

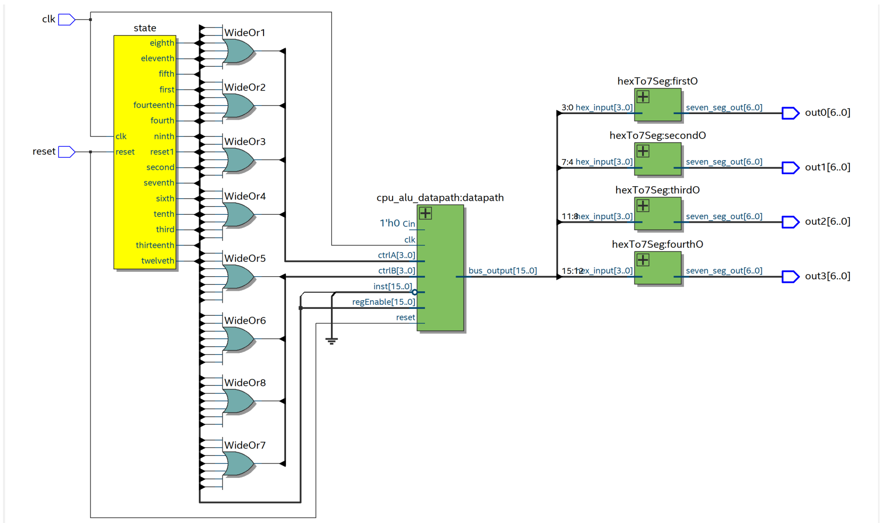


Figure 11: Finite State Machine connecting to the CPU Datapath

4 Synthesis Results and Observations

For the synthesis results we have decided to only report the synthesis results from the CPU Datapath, Register File, and ALU being synthesized together.

The synthesis completes with few warnings that are not related to the actual Verilog code. On the summary page we observe that the total number of registers used is 255 and the total number of pins used is 30. Clicking through the synthesis report there is a page that takes each instantiation and how many combinational ALUTs and Logic Registers that are assigned to each. An interesting report that is output is the Optimization Results where we can see which registers were removed and why they were removed. Our project had 20 registers merged with other registers saving resources. The Connectivity Checks section told us that 4 ports are stuck at GND. The ports that are stuck are things that have no use since they have not yet been implemented in our processor.

5 Division of Labour

Labor was evenly divided between the entire group. In the ALU Lab, Stephan and Jeremy worked on the ALU design and coded up the main module driving the Flags, Cin, and output, while Steen and Jonathan worked on testing the ALU with a testbench. Stephan and Jeremy also worked on programming the FPGA with a simple wrapper module. On the Register File lab we switched roles. Stephan worked on an FSM that would use the wrapper created by Steen and Jonathan and map it to the board, while Jonathan and Steen worked on the logic, wrapper, test bench, and one FSM. Jeremy worked on helping Stephan with FSM logic and mapping and also created an FSM of his own to test the entire circuit. All in all the work was decently divided.