Jack McWeeney
ECE 574 CAD Digital VLSI Design
Final Project Report
2 January 2023

1 Abstract

For my Type-B final project, I created an 8-bit single-cycle processor which includes an ALU, register file, and eleven opcodes. The processor reads from machine code stored in a program ROM and executes the program. In designing this project, I followed several write-ups from people who had done similar projects in Verilog, but modified those projects to create a more fully-featured processor, as well as write original testbenches for each individual component.

2 Introduction

I chose this project because I wanted to push myself to complete a Verilog project that is well beyond the scope of any of the projects I have done before - specifically, the homework assignments in this course. As I completed my undergraduate degree about six years ago, this was also a very helpful in-depth refresher on the concepts behind CPU hardware.

3 Related work

I directly referenced a Medium article written by Sathira Basnayake while writing the modules for my project. [1], except the barrel shifter which was based on an example I found in a blog post. [2] Aside from these resources, I mainly referred to the presentation slides from this course.

4 Data description

The data in my model is machine code stored in a 128-byte ROM. The machine code is 4 bytes long and the final project contains 9 program codes, or 72 bytes, of machine code. As this is programmed in to the testbench, there is no external input from an operating system or user.

5 Method description

To achieve this project, I started by reviewing several existing Verilog CPU projects. [1] [3] [4] I ended up mainly following the Medium article because the write up was simple and easy to follow, and the project was simple enough that I could see early on how I could not only implement their work but improve upon it. I put together the project by writing one module at a time and then writing a testbench for it to confirm it behaved as intended before moving on to the next module or joining it all together.

6 Model description

To accomplish my goal of building a simple CPU, I wrote Verilog files for an ALU, a barrel shifter for the ALU, an 8-bit registry file, and finally a CPU program to tie those all together. In between each, I wrote a testbench file for each component to confirm their operation, and a CPU testbench which confirms the unit as a whole functions as intended. It is worth noting that there were numerous issues when I would attempt to run the Verilog code directly from the Medium article. I believe many of these errors were because the code was written in SystemVerilog, though the author never states this explicitly. Further errors were caused simply by typos and other small mistakes on the author's part.

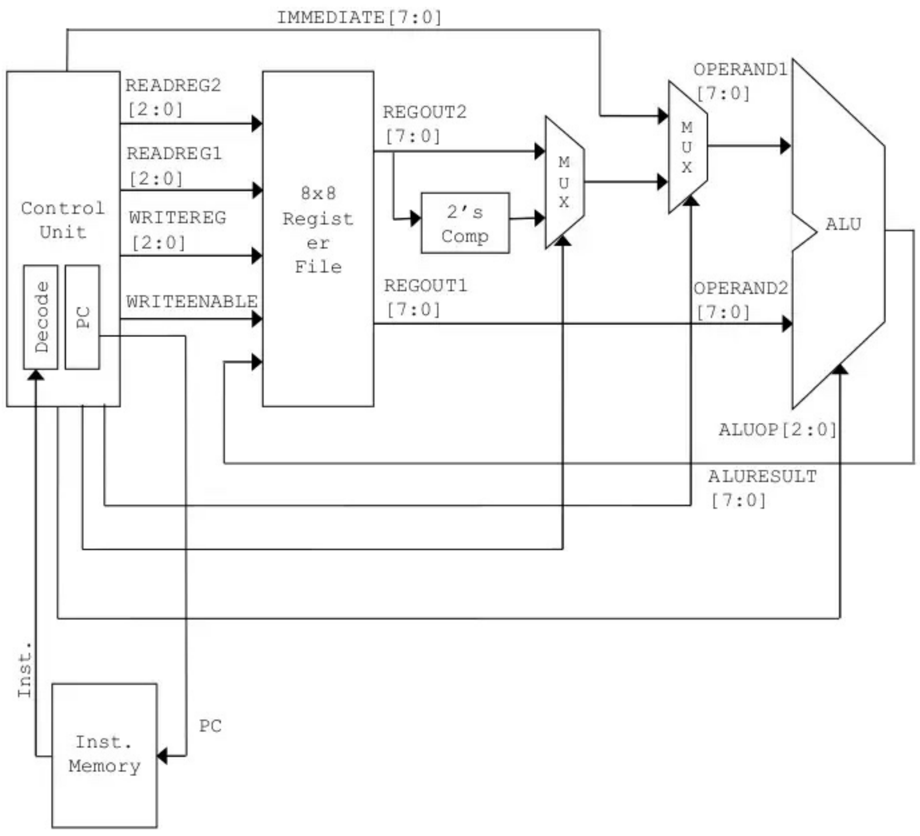

Figure 1: Design of CPU implemented in Medium article.

My 32-bit instruction scheme was based on the scheme presented in the Medium article as well:

Opcode	*Destination*	*Source1*	*Source2*
31:24	23:16	15:8	7:0

The Medium article initially intends to implement opcodes for `loadi, add, sub, and, or, j` and `beq`. However, not all of these would end up being included in their final product. I used 3 bits to address the ALU opcodes for a possible maximum 8 operations. I implemented the following opcodes and corresponding mathematical operations:

ALU Select	*ALU Output*
3'b000	OUT = IN1;
3'b001	OUT = IN1 + IN2;
3'b010	OUT = IN1 & IN2;
3'b011	OUT = IN1 | IN2;
3'b100	OUT = RShiftResult;

As mentioned above, I included a barrel shifter in my ALU, which took an input of 8 bits, right shifted it by a specified amount, and returned the shifted value. This was because the author of the Medium article included it in their ALU, though without example code, which is why I ended up looking elsewhere for an explanation of its function and example implementation. [2] However, the Medium article strangely does not use the barrel shifter for any functions in their final project - I assume it was to be used for one of the intended opcodes they did not implement - so I also did not utilize it. An original feature I implemented in the ALU was a `zero` flag bit, which is set to 1 when the output is zero and 0 all other times. This is accomplished by NOR-ing all of the bits of the output together. The article's CPU does not utilize any flag bits. This will be used to later implement BEQ instructions that are not in the source article.

```
always@(OUT) begin
    ZERO = ~|(OUT);
end
```

I then implemented an 8-bit register file, which I chose to give 8 registers. The register file has 2 read lines, 2 output lines, 1 write line and 1 write enable input signal. It also takes a clock input and reset input. The reset simply sets all registers to 0, and clock is necessary to ensure that we are able to synchronize writing with the CPU as appropriate.

My final version of the CPU Verilog file is based on the CPU presented in the Medium article [1]. However, I made several changes of my own design, such that my final CPU module looks very little like the module presented in the article. The first of these large changes was to reformat the CPU's opcodes to directly encode the control signals, as follows:

OpCode bit	*Control signal (CS)*
7	Write enable
[6:4]	ALU op
3	Immediate value
2	Positive
1	Branch
0	Zero

This allowed me to implement opcodes and control signals implicitly, and to easily parse them with few lines of code:

```
    always@(instruction) begin
        opcode = instruction[31:24];
        ALUop = instruction[30:28];
        b_addr = instruction[23:16];
        regWriteAddr = instruction[18:16];
        RegReadAddr1 = instruction[10:8]; //3 bit addressing for 8 addr
   register
        RegReadAddr2 = instruction[2:0];
        immediateVal = instruction[7:0];
    end
```

Comparing this to the original Verilog code provided where a CASE statement is used to identify the opcode and the control signals must be explicitly defined, we can see my control scheme is much simpler:

```
always @(INSTRUCTION)
 begin
     // taking the opcode from the instruction
     OPCODE = INSTRUCTION[31:24];
     #1
     //decodeing the opcode
  case(OPCODE)
   8'b00000000:
       begin
    writeEnable = 1'b1;
    aluOp = 3'b000;
    isAdd = 1'b1;
    isImediate = 1'b1;
    end
   8'b00000001:
       begin
    writeEnable = 1'b1;
    aluOp = 3'b000;
    isAdd = 1'b1;
    isImediate = 1'b0;
    end
   8'b00000010:
       begin
    writeEnable = 1'b1;
    aluOp = 3'b001;
    isAdd = 1'b1;
    isImediate = 1'b0;
    end
   8'b00000011:
       begin
    writeEnable = 1'b1;
    aluOp = 3'b001;
    isAdd = 1'b0;
    isImediate = 1'b0;
    end
   8'b00000100:
       begin
    writeEnable = 1'b1;
    aluOp = 3'b010;
    isAdd = 1'b1;
    isImediate = 1'b0;
    end
   8'b00000101:
       begin
    writeEnable = 1'b1;
    aluOp = 3'b011;
    isAdd  =1'b1;
    isImediate = 1'b0;
    end
  endcase

    DESTINATION  = INSTRUCTION[18:16];
    SOURCE1   = INSTRUCTION[10:8];
    SOURCE2 = INSTRUCTION[2:0];
    immediateVal =INSTRUCTION[7:0];
 end
```

One factor of the author's original scheme I did not consider when making this change was that the explicit declaration of control sognals as soon at the moment the opcode is decoded makes for very tidy timing of the CPU hardware. In my case, I found during troubleshooting that the additional MUX and logic gate hardware to determine the control signals implicitly introduced various delays, causing control signals to vary a bit in the beginning of the clock cycle. This required additional troubleshooting to diagnose and had to be handled with variations in hardware and control signal design to ensure that no accidental CPU operations were being performed due to fluctuating control signals, memory values, or ALU outputs.

The other large change I made to the CPU design presented was implementing a few additional commands; `jmp`, `jmpi`, and `beq` (while j and beq were listed as intended opcodes during the ALU design earlier in the article, the author did not end up implementing it in their CPU). This required the addition of several control signals (`branch` and `zero`) and several additional MUX's to control the PC. I was able to successfully control the PC with these control signals such that it would be incremented normally unless a control signal determined that a jump or branch address should be followed. The final CPU design that I implemented can be seen in figure 2.

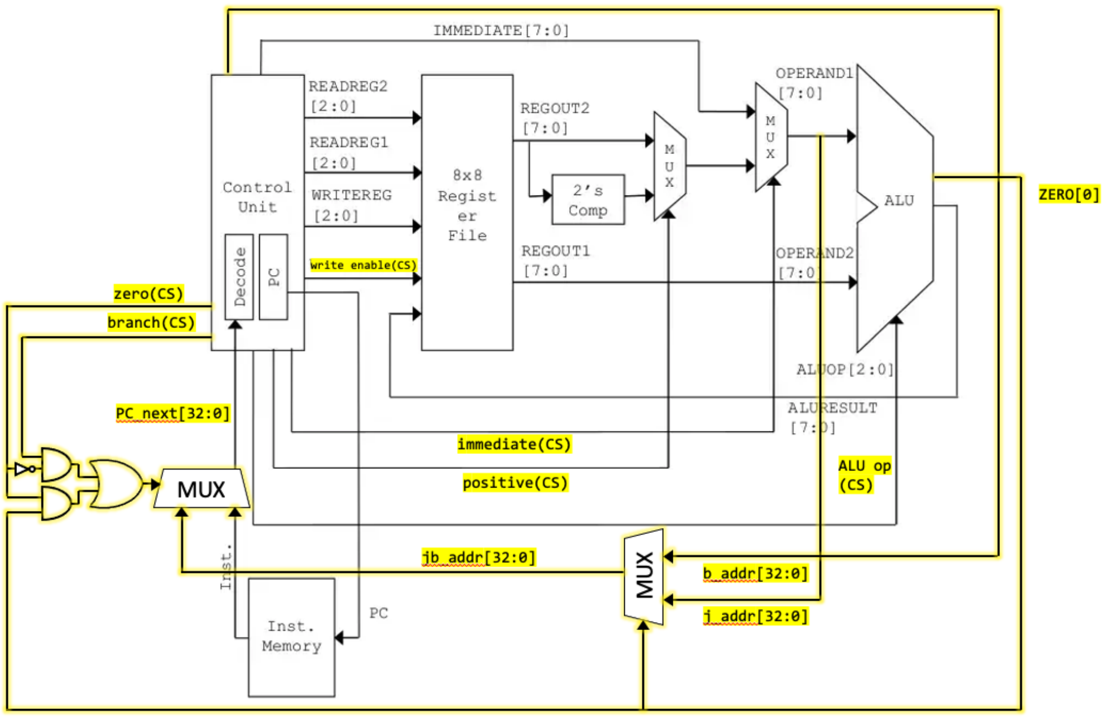

Figure 2: Design of CPU with my added control signals and logic (highlighted for emphasis). Note that CS means "control signal" to avoid confusion with names.

7 Experimental procedure and results

To test my code, I wrote testbench programs for each component before moving on to the next. I did not use any reference when writing these. The testbenches for the barrel shifter, ALU, and register file are simple, similar to those made for this classes' homework problems, and served to confirm expected behavior of the module as simply as possible. The CPU testbench is a little more involved; I created an 8-bit ROM memory, addressed by big endian, and then I came up with a pseudocode program I wanted to implement. I wrote machine code according to my CPU's specifications and saved that code to the program ROM, and then ran the test and ensured that the CPU's behavior, including the PC and the register file's stored values, were performing the program as I intended.

Troubleshooting mainly consisted of fixing slight timing and logic errors in the CPU's Verilog module. When implementing my custom machine code, I would implement one command at a time, and almost none of them worked on my first attempt. So, I would then `$display` various signals and values relevant to the operation I was attempting to perform until I could isolate specific issues and update the control logic and module design to resolve them.

Finally, my intended pseudocode which consists of multiple reads, writes, arithmetic operations, jumps and branches was implemented in machine code and I was able to confirm with CPU and register values that the code was being executed as expected. The final testbench and the successful results can be seen in Appendix 9.2.h.

Memory location	*Pseudocode*	*Machine code*
0	R1 = 10	10001100 00000001 00000000 00001010
4	R1 = R1 + 10	10011100 00000001 00000001 00001010
8	R2 = 15	10001100 00000010 00000000 00001111
12	R3 = R1 + R2	10010100 00000011 00000001 00000010
16	R3 = R3 - 40	10011000 00000011 00000011 00101000
20	R1 = 64	10001100 00000001 00000000 01000000
24	Jump to line (R1)	00000110 00000000 00000000 00000001
64	R2 = R3	10000100 00000010 00000001 00000011
68	BEQ R2 and R3 to line 30	00010011 00011110 00000010 00000011

Pseudocode and machine code of program that is stored in to CPU ROM and run.

8 Conclusion

As I have demonstrated, I was able to successfully develop the Verilog modules for an 8-bit single-cycle processor. While I began by referencing an existing implementation, the existing work required a good amount of troubleshooting in its raw form, and I built on to their implementation by refactoring their code and adding additional CPU opcodes to create a better designed, more complex, and more fully-featured product. The final product can perform the following CPU operations: `ld`, `mov`, `add`, `addi`, `sub`, `subi`, `and`, `or`, `jmp`, `jmpi`, and `beq`. Troubleshooting the final project was very involved and required detailed methodical investigation to resolve various issues across the multiple modules. In the end, I was able to write a program in machine code and store it in to the program ROM, where it would then run on the CPU as intended. Some features I would have liked to implement given more time would have been RAM and pipelining.

Overall, after a semester of learning Verilog I feel that this project was a great exercise in creating a large, complex and multifaceted device using the HDL. Additionally, while it did not end up being utilized in the final product, the barrel shifter was a new concept I had not heard of before, and I feel it was a fascinating example of the sort of efficiency gains you can achieve by using combinatorial logic rather than sequential logic. Though I was not able to use it, I am curious where this barrel shifter could be used in the CPU I created, and how it would improve efficiency. I am pleased by the results I achieved and the experience I gained, and I look forward to continuing to learn and expand my skills in Verilog.

9 Appendix

9.1 Verilog code

a. barrelShifter.v

```verilog
//from https://esrd2014.blogspot.com/p/barrel-shifter.html

`timescale 1ns/1ps

module mux2x1 (input in1, input in2, output out, input sel);
    reg out;

    always @(*) begin
       if (sel == 1)
            out = in1;
        else
            out = in2;
    end
endmodule

module barrelShifter(IN, SHIFT, OUT);
    input [7:0] IN;
    input [2:0] SHIFT;
    output [7:0] OUT;

    wire [7:0] ST1, ST2;

    mux2x1 m0_0(1'b0, IN[0], ST1[0], SHIFT[0]);
    mux2x1 m0_1(IN[0], IN[1], ST1[1], SHIFT[0]);
    mux2x1 m0_2(IN[1], IN[2], ST1[2], SHIFT[0]);
    mux2x1 m0_3(IN[2], IN[3], ST1[3], SHIFT[0]);
    mux2x1 m0_4(IN[3], IN[4], ST1[4], SHIFT[0]);
    mux2x1 m0_5(IN[4], IN[5], ST1[5], SHIFT[0]);
    mux2x1 m0_6(IN[5], IN[6], ST1[6], SHIFT[0]);
    mux2x1 m0_7(IN[6], IN[7], ST1[7], SHIFT[0]);
    mux2x1 m1_0(1'b0, ST1[0], ST2[0], SHIFT[1]);
    mux2x1 m1_1(1'b0, ST1[1], ST2[1], SHIFT[1]);
    mux2x1 m1_2(ST1[0], ST1[2], ST2[2], SHIFT[1]);
    mux2x1 m1_3(ST1[1], ST1[3], ST2[3], SHIFT[1]);
    mux2x1 m1_4(ST1[2], ST1[4], ST2[4], SHIFT[1]);
    mux2x1 m1_5(ST1[3], ST1[5], ST2[5], SHIFT[1]);
    mux2x1 m1_6(ST1[4], ST1[6], ST2[6], SHIFT[1]);
    mux2x1 m1_7(ST1[5], ST1[7], ST2[7], SHIFT[1]);
    mux2x1 m2_0(1'b0, ST2[0], OUT[0], SHIFT[2]);
    mux2x1 m2_1(1'b0, ST2[1], OUT[1], SHIFT[2]);
    mux2x1 m2_2(1'b0, ST2[2], OUT[2], SHIFT[2]);
    mux2x1 m2_3(1'b0, ST2[3], OUT[3], SHIFT[2]);
    mux2x1 m2_4(ST2[0], ST2[4], OUT[4], SHIFT[2]);
    mux2x1 m2_5(ST2[1], ST2[5], OUT[5], SHIFT[2]);
    mux2x1 m2_6(ST2[2], ST2[6], OUT[6], SHIFT[2]);
    mux2x1 m2_7(ST2[3], ST2[7], OUT[7], SHIFT[2]);

endmodule
```

b. alu.v

```verilog
//from https://studentsxstudents.com/simple-8-bit-processor-design-and-
   verilog-implementation-part-1-8735fac284b
```

```verilog

//add, sub, and, or, mov, loadi, j, and beq
`timescale 1ns/1ps
`include "barrelShifter.v"

//OpCode    Destination In1     In2
//31:24     23:16       15:8    7:0

module twos_comp(input [7:0] in, output reg [7:0] result);
    always@(*) begin
        result <= ~in+1;
    end
endmodule

module mux2x1_var (in1, in2, out, sel);
    parameter WIDTH = 8;
    input [WIDTH-1:0] in1, in2;
    input sel;
    output reg [WIDTH-1:0] out;

    always @(*) begin
       if (sel == 1)
            out = in2;
        else
            out = in1;
    end
endmodule

module alu(IN1, IN2, OUT, ZERO, SEL);
    input [7:0] IN1, IN2;
    input [2:0] SEL;
    input ACT;
    output reg [7:0] OUT;
    output reg ZERO;

    wire [7:0] RshiftResult;

    initial begin
      OUT = {7{1'b0}};
    end

    barrelShifter RightLogicalShifter(IN1, IN2[7:5], RshiftResult); //input
   to be shifted, shift amount, output reg

    always @(IN1, IN2, SEL) begin
        case(SEL)
            3'b000: #2 OUT = IN1; //mov and ld
            3'b001: #2 OUT = IN1 + IN2; //add and sub
            3'b010: #2 OUT = IN1 & IN2; //AND and sub
            3'b011: #2 OUT = IN1 | IN2; //OR and sub
            3'b100: #2 OUT = RshiftResult;
            // 3'b101:
            // 3'b110:
            // 3'b111:
            default: OUT = {8{1'b0}};
        endcase
    end
    //zero bit
    always@(OUT) begin
        ZERO = ~|(OUT);
    end
endmodule
```

c. 8bitReg.v

```
`timescale 1ns/1ps
//`include "clk_gen.v"

module registerFile(in, out1, out2, w_addr, r_addr1, r_addr2, write, clk,
    rst);
    input write, clk, rst;
    input [2:0] w_addr, r_addr1, r_addr2;
    input [7:0] in;
    output [7:0] out1, out2;

    reg [7:0] regFile [7:0];

    integer i;
    always@(rst == 1) begin
        #2 for (i=0; i<8; i=i+1) begin
            regFile[i] = {8{1'b0}};
        end
    end

    always@(negedge clk && in) begin
        if (write == 1 && rst == 0) begin
            $display("writing - R%d - %b", w_addr, in);
            regFile[{{5{1'b0}}, w_addr}] <= in;
        end
    end

    assign #2 out1 = regFile[r_addr1];
    assign #2 out2 = regFile[r_addr2];

endmodule
```

d. cpu.v

```
`timescale 1ns/1ps
`include "alu.v"
`include "8bitReg.v"

/*
                OpCode  Destination   ReadAddr1 ReadAddr2
                31:24   23:16         15:8      7:0

Immediate inst: OpCode  Destination   ReadAddr1 ImmediateVal
Jump inst:      OpCode  -----         -----     RegValAddr|immediateVal
BEQ inst:       OpCode  BranchAddr    RegVal1   RegVal2

OpCode Key:
    b7 write enable
    b6:4 ALU op
    b3 is immediate
    b2 is out2 positive
    b1 is branch
    b0 ZERO
*/

module PC_adder(input [31:0] PC, output reg [31:0] result);
    always@(PC) result = PC+4;
endmodule

module cpu(PC, instruction, clk, rst);
    input [31:0] instruction;
    input clk, rst;
    output reg [31:0] PC;
    wire [31:0] PC_adder_result, PC_next, j_addr, jb_addr;
    wire [7:0] ALU_result, mux1out, mux2out, out1, out2, minusVal;
    wire reset, zero;

    reg [2:0] ALUop, RegReadAddr1, RegReadAddr2, regWriteAddr;
    reg [7:0] immediateVal, in, opcode, b_addr;
    assign reset = rst;

    registerFile register(in, out1, out2, regWriteAddr, RegReadAddr1,
   RegReadAddr2, opcode[7], clk, reset);
    //in, out1, out2, w_addr, r_addr1, r_addr2, write, clk, rst
    twos_comp negative_val(out2, minusVal);
    alu alu_unit (mux2out, out1, ALU_result, zero, ALUop); //IN1, IN2, OUT,
   ZERO, SEL

    //use PC adder to increment PC
    PC_adder PC_inc(PC, PC_adder_result);

    mux2x1_var mux1(minusVal, out2, mux1out, opcode[2]);
    mux2x1_var mux2(mux1out, immediateVal, mux2out, opcode[3]);

    assign j_addr = {{24{1'b0}}, mux2out};

    //when jump - PC_next is either out2 or imm_val (no negative); when beq,
    it is potentially writeregaddr; select can be zero as if beq is not zero
    the value doesn't matter anyway
    mux2x1_var #(.WIDTH(32)) mux_jump(j_addr, {{24{1'b0}}, b_addr}, jb_addr,
    zero); //jump addr (mux2out) or destination (opcode[23:16]) for next
   address
    //on beq; ALU operation is performed; if ZERO flag is set, PC_next =
   mux2out; else, PC_next = PC_adder_result
    mux2x1_var #(.WIDTH(32)) mux_pc(PC_adder_result, jb_addr, PC_next, ((
   opcode[1] && !zero)|(opcode[0] && zero))); //mux to choose between PC
   incremented val (normally) or jump inst. address

    always@(rst) begin
        if(rst==1) begin
            PC <= -4;
            $display("Reset = %b", rst);
        end
    end

    always@(posedge clk) begin
        PC <= PC_next;
        #5 $display("PC: %b | PC_next: %b", PC, PC_next);
    end

    always@(instruction) begin
    //parse instruction
        $display("Instruction: %b", instruction);
        opcode <= instruction[31:24];
        ALUop <= instruction[30:28];
        b_addr <= instruction[23:16];
        regWriteAddr <= instruction[18:16];
        RegReadAddr1 <= instruction[10:8]; //3 bit addressing for 8 addr
   register
        RegReadAddr2 <= instruction[2:0];
        immediateVal <= instruction[7:0];
        #5
        $display("ALUop: %b, opcode[2](neg) = %b, mux1out %b, mux2out %b,
   out1 %b, ALU out = %b, zero %b", ALUop, opcode[2], mux1out, mux2out, out1
   , ALU_result, zero);
    end

    always@(ALU_result) begin
        in <= ALU_result; //reg file input is alu result
    end

endmodule
```

9.2 Testbench files and results

e. barrelShifter_tb.v

```verilog
`timescale 1ns/1ps
`include "barrelShifter.v"

module barrelShifter_tb();
    reg [7:0] in;
    reg [2:0] shift;
    wire [7:0] out;

    reg [7:0] in_ [2:0];
    reg [2:0] shift_ [2:0];
    reg [1:0] j, k;

    barrelShifter shifter(in, shift, out);

    initial begin
        in_[0] = 8'b00000001;
        in_[1] = 8'b10101010;
        in_[2] = 8'b11111000;
        shift_[0] = 3'b001;
        shift_[1] = 3'b100;
        shift_[2] = 3'b111;

        for (j=0; j<3;j++) begin
            in = in_[j];
            for (k=0; k<3; k++) begin
                shift = shift_[k];
                #10
                $display("In: %8b | Shift: %8b | Out: %8b", in, shift, out);
            end
        end
    end

endmodule
```

Result:

```
[Running] barrelShifter_tb.v
In: 00000001 | Shift:      001 | Out: 00000010
In: 00000001 | Shift:      100 | Out: 00010000
In: 00000001 | Shift:      111 | Out: 10000000
In: 10101010 | Shift:      001 | Out: 01010100
In: 10101010 | Shift:      100 | Out: 10100000
In: 10101010 | Shift:      111 | Out: 00000000
In: 11111000 | Shift:      001 | Out: 11110000
In: 11111000 | Shift:      100 | Out: 10000000
In: 11111000 | Shift:      111 | Out: 00000000
[Done] exit with code=0 in 0.48 seconds
```

f. alu_tb.v

```verilog
`timescale 1ns/1ps
`include "alu.v"

module tb_alu;
    reg signed [7:0] A, B;
    reg [2:0] sel;
    wire [7:0] out;
    wire zero;

    reg [2:0] i;
    reg [1:0] j, k;

    reg [31:0] A_ [1:0];
    reg [31:0] B_ [2:0];
    reg [2:0] op_codes [5:0];

    alu uut(A, B, out, zero, sel);

        initial begin
            op_codes[0] = 3'b000;
            op_codes[1] = 3'b001;
            op_codes[2] = 3'b010;
            op_codes[3] = 3'b011;
            op_codes[4] = 3'b100;

            A_[0] = 8'b00001111;
            A_[1] = 8'b10101010;
            B_[0] = 8'b01011010;
            B_[1] = 8'b10111011;
            B_[2] = 8'b11110000;

            for (j=0;j<2;j++) begin
                A = A_[j];
                for (k=0;k<3;k++) begin
                    B = B_[k];
                    for (i=0;i<5;i++) begin
                        sel = op_codes[i];
                        #10;
                        $display ("%0d test data: A: %d / %8b, B: %d / %8b,
   ALUop is %b, result is %d / %8b, zero: %1b", (10*j+5*k+i+1), A,A, B,B,
   sel, $signed(out),$signed(out), zero);
                    end
                end
            end
        end
endmodule
```

Result:

```
[Running] alu_tb.v
1 test data: A:   15 / 00001111, B:   90 / 01011010, ALUop is 000, result is
      15 / 00001111, zero: 0
2 test data: A:   15 / 00001111, B:   90 / 01011010, ALUop is 001, result is
     105 / 01101001, zero: 0
3 test data: A:   15 / 00001111, B:   90 / 01011010, ALUop is 010, result is
      10 / 00001010, zero: 0
4 test data: A:   15 / 00001111, B:   90 / 01011010, ALUop is 011, result is
      95 / 01011111, zero: 0
5 test data: A:   15 / 00001111, B:   90 / 01011010, ALUop is 100, result is
      60 / 00111100, zero: 0
6 test data: A:   15 / 00001111, B:  -69 / 10111011, ALUop is 000, result is
      15 / 00001111, zero: 0
7 test data: A:   15 / 00001111, B:  -69 / 10111011, ALUop is 001, result is
```

```
     -54 / 11001010, zero: 0
8 test data: A:   15 / 00001111, B:  -69 / 10111011, ALUop is 010, result is
      11 / 00001011, zero: 0
9 test data: A:   15 / 00001111, B:  -69 / 10111011, ALUop is 011, result is
     -65 / 10111111, zero: 0
10 test data: A:   15 / 00001111, B:  -69 / 10111011, ALUop is 100, result
   is  -32 / 11100000, zero: 0
11 test data: A:   15 / 00001111, B:  -16 / 11110000, ALUop is 000, result
   is   15 / 00001111, zero: 0
12 test data: A:   15 / 00001111, B:  -16 / 11110000, ALUop is 001, result
   is   -1 / 11111111, zero: 0
13 test data: A:   15 / 00001111, B:  -16 / 11110000, ALUop is 010, result
   is    0 / 00000000, zero: 1
14 test data: A:   15 / 00001111, B:  -16 / 11110000, ALUop is 011, result
   is   -1 / 11111111, zero: 0
15 test data: A:   15 / 00001111, B:  -16 / 11110000, ALUop is 100, result
   is -128 / 10000000, zero: 0
11 test data: A:  -86 / 10101010, B:   90 / 01011010, ALUop is 000, result
   is  -86 / 10101010, zero: 0
12 test data: A:  -86 / 10101010, B:   90 / 01011010, ALUop is 001, result
   is    4 / 00000100, zero: 0
13 test data: A:  -86 / 10101010, B:   90 / 01011010, ALUop is 010, result
   is   10 / 00001010, zero: 0
14 test data: A:  -86 / 10101010, B:   90 / 01011010, ALUop is 011, result
   is   -6 / 11111010, zero: 0
15 test data: A:  -86 / 10101010, B:   90 / 01011010, ALUop is 100, result
   is  -88 / 10101000, zero: 0
16 test data: A:  -86 / 10101010, B:  -69 / 10111011, ALUop is 000, result
   is  -86 / 10101010, zero: 0
17 test data: A:  -86 / 10101010, B:  -69 / 10111011, ALUop is 001, result
   is  101 / 01100101, zero: 0
18 test data: A:  -86 / 10101010, B:  -69 / 10111011, ALUop is 010, result
   is  -86 / 10101010, zero: 0
19 test data: A:  -86 / 10101010, B:  -69 / 10111011, ALUop is 011, result
   is  -69 / 10111011, zero: 0
20 test data: A:  -86 / 10101010, B:  -69 / 10111011, ALUop is 100, result
   is   64 / 01000000, zero: 0
21 test data: A:  -86 / 10101010, B:  -16 / 11110000, ALUop is 000, result
   is  -86 / 10101010, zero: 0
22 test data: A:  -86 / 10101010, B:  -16 / 11110000, ALUop is 001, result
   is -102 / 10011010, zero: 0
23 test data: A:  -86 / 10101010, B:  -16 / 11110000, ALUop is 010, result
   is  -96 / 10100000, zero: 0
24 test data: A:  -86 / 10101010, B:  -16 / 11110000, ALUop is 011, result
   is   -6 / 11111010, zero: 0
25 test data: A:  -86 / 10101010, B:  -16 / 11110000, ALUop is 100, result
   is    0 / 00000000, zero: 1
[Done] exit with code=0 in 0.118 seconds
```

g. 8bitReg_tb.v

```
`timescale 1ns/1ps
`include "8bitReg.v"

// module clock(input enable, output reg clk);
//     reg not_clk;

//     initial begin
//          $display("here");
//          forever #5 clk = not_clk;
//          not_clk = ~clk;
//          $display("%1b", clk);
//      end
// endmodule

module registerFile_tb();
    reg write, reset;
    reg [2:0] w_addr, r_addr1, r_addr2;
    reg [7:0] in;
    wire [7:0] out1, out2;
    reg clk;

    registerFile reg_test(in, out1, out2, w_addr, r_addr1, r_addr2, write,
   clk, reset);

    initial begin
        $monitor(clk);
        w_addr = 3'b010;
        r_addr1 = 3'b010;
        reset = 1;
        reset = 0;
        clk = 0;
        #5 clk = 1;
        write = 1;
        in = 8'b00001111;
        #5 clk = 0;
        $display("writing %d to R%d", in, w_addr);
        #5 write = 0;
        clk = 1;
        #5 $display ("Address %d contains %d", r_addr1, out1);
        $display(reg_test.regFile[2]);
    end

endmodule
```

Result:

```
[Running] 8bitReg_tb.v
0
1
writing  15 to R2
writing - R2 - 00001111
0
1
Address 2 contains  15
 15
[Done] exit with code=0 in 0.118 seconds
```

h. cpu_tb.v

```
`timescale 1ns/1ps
`include "cpu.v"

/*
                OpCode  Destination    ReadAddr1 ReadAddr2
                31:24   23:16          15:8      7:0
```

```

Immediate inst: OpCode  Destination ReadAddr1   ImmediateVal
Jump inst:      OpCode  -----         -----     RegValAddr|immediateVal

OpCode Key:
    b7 write enable
    b6:4 ALU op
    b3 is immediate
    b2 is out2 positive
    b1 is branch
    b0 ZERO
*/

module cpu_tb();
    reg clk, rst;
    wire [31:0] PC;
    reg [31:0] instruction;
    reg [7:0] opcodes [15:0];

    //instruction memory: 8bit wide, up to 2^32 locations
    reg [7:0] instruction_mem [127:0];

    integer i;

    initial begin
        $display("cpu_tb here");
        clk = 0;
        for (i=0; i<128; i=i+1) instruction_mem[i] = {8{1'b0}};
        forever begin
            #5 clk = ~clk;
            $display("clk = %b", clk);
            //clk_counter = clk_counter + 1
        end
    end

    always@(PC) begin
        instruction = {
            instruction_mem[PC+3],
            instruction_mem[PC+2],
            instruction_mem[PC+1],
            instruction_mem[PC]
        };
        //$display("Instruction: %b", instruction);
    end

    cpu cpu_test(PC, instruction, clk, rst);

    initial begin

        $display("R1 = %8b", cpu_test.register.regFile[1]);

        opcodes[0]  = 8'b1000_1100; //ld
        opcodes[1]  = 8'b1000_0100; //mov
        opcodes[2]  = 8'b1001_0100; //add
        opcodes[3]  = 8'b1001_1100; //addi
        opcodes[4]  = 8'b1001_0000; //sub
        opcodes[5]  = 8'b1001_1000; //subi
        opcodes[6]  = 8'b1010_0100; //and
        opcodes[7]  = 8'b1011_0100; //or
        opcodes[8]  = 8'b0000_0110; //jmp
        opcodes[9]  = 8'b0000_1110; //jmpi
        opcodes[10] = 8'b0001_0011; //beq

        for (i=11; i<16; i=i+1) opcodes[i] = {32{1'b0}};

        /*
            Desired pseudocode:
        0:  store 10 in R1
            r1 = r1 + #10
            store 15 in R2
            R3 = R1 + R2
            R3 = R3 - #40
            R1 = 64
            jump to addr in R1
        64: store val in R3 in R2
            beq R2 and R3 to 30
        */

        // instruction_mem[3:0]   = {opcodes[0], 8'd1, 8'dx, 8'd10};
        //10001100 00000001 00000000 00001010
        instruction_mem[3] = opcodes[0];
        instruction_mem[2] = 8'd1;
        instruction_mem[1] = 8'd0;
        instruction_mem[0] = 8'd10;

        // instruction_mem[7:4]   = {opcodes[3], 8'd1, 8'd1, 8'd10};
        //10011100 00000001 00000001 00001010
        instruction_mem[7] = opcodes[3];
        instruction_mem[6] = 8'd1;
        instruction_mem[5] = 8'd1;
        instruction_mem[4] = 8'd10;

        // instruction_mem[11:8]  = {opcodes[0], 8'd2, 8'dx, 8'd15};
        // 10001100000000100000000000001111
        instruction_mem[11] = opcodes[0];
        instruction_mem[10]  = 8'd2;
        instruction_mem[9]  = 8'd0;
        instruction_mem[8]  = 8'd15;

        // instruction_mem[15:12] = {opcodes[2], 8'd3, 8'd1, 8'd2};
        //10010100 00000011 00000001 00000010
        instruction_mem[15] = opcodes[2];
        instruction_mem[14] = 8'd3;
        instruction_mem[13] = 8'd1;
        instruction_mem[12] = 8'd2;

        // instruction_mem[19:16] = {opcodes[5], 8'd3, 8'd3, 8'd40};
        //10011000 00000011 00000011 00101000
        instruction_mem[19] = opcodes[5];
        instruction_mem[18] = 8'd3;
        instruction_mem[17] = 8'd3;
        instruction_mem[16] = -8'd40;

        // instruction_mem[23:20] = {opcodes[0], 8'd1, 8'dx, 8'd64};
        //10001100 00000001 00000000 01000000
        instruction_mem[23] = opcodes[0];
```

```verilog
        instruction_mem[22] = 8'd1;
        instruction_mem[21] = 8'd0;
        instruction_mem[20] = 8'd64;

        // instruction_mem[27:24] = {opcodes[8], 8'dx, 8'dx, 8'd1};
        //00000110 00000000 00000000 00000001
        instruction_mem[27] = opcodes[8];
        instruction_mem[26] = 8'd0;
        instruction_mem[25] = 8'd0;
        instruction_mem[24] = 8'd1;

        // instruction_mem[67:64] = {opcodes[1], 8'd2, 8'dx, 8'd3};
        //1000010 00000001 00000000 100000011
        instruction_mem[67] = opcodes[1];
        instruction_mem[66] = 8'd2;
        instruction_mem[65] = 8'd1;
        instruction_mem[64] = 8'd3;

        // instruction_mem[71:68] = {opcodes[10], 8'dx, 8'd2, 8'd3};
        //00010011 00011110 00000010 00000011
        instruction_mem[71] = opcodes[10];
        instruction_mem[70] = 8'd30;
        instruction_mem[69] = 8'd2;
        instruction_mem[68] = 8'd3;

        rst = 1;
        #1 rst = 0;

        #100
        $finish;
    end

endmodule
```

Result:

```
[Running] cpu_tb.v
cpu_tb here
R1 = xxxxxxxx
Reset = 1
clk = 1
Instruction: 10001100000000010000000000001010
clk = 0
PC: 00000000000000000000000000000000 | PC_next:
   00000000000000000000000000000100
ALUop: 000, opcode[2](neg) = 1, mux1out 00000000, mux2out 00001010, out1
   00000000, ALU out = 00001010, zero 0
writing - R1 - 00001010
clk = 1
Instruction: 10011100000000010000000100001010
clk = 0
PC: 00000000000000000000000000000100 | PC_next:
   00000000000000000000000000001000
ALUop: 001, opcode[2](neg) = 1, mux1out 00000000, mux2out 00001010, out1
   00001010, ALU out = 00010100, zero 0
writing - R1 - 00010100
clk = 1
Instruction: 10001100000000010000000000001111
```

```
clk = 0
PC: 00000000000000000000000000001000 | PC_next:
   00000000000000000000000000001100
ALUop: 000, opcode[2](neg) = 1, mux1out 00000000, mux2out 00001111, out1
   00000000, ALU out = 00001111, zero 0
writing - R2 - 00001111
clk = 1
Instruction: 10010100000000110000000100000010
clk = 0
PC: 00000000000000000000000000001100 | PC_next:
   00000000000000000000000000010000
ALUop: 001, opcode[2](neg) = 1, mux1out 00001111, mux2out 00001111, out1
   00010100, ALU out = 00100011, zero 0
writing - R3 - 00100011
clk = 1
Instruction: 10011000000000110000001111011000
clk = 0
PC: 00000000000000000000000000010000 | PC_next:
   00000000000000000000000000010100
ALUop: 001, opcode[2](neg) = 0, mux1out 00000000, mux2out 11011000, out1
   00100011, ALU out = 11111011, zero 0
writing - R3 - 11111011
clk = 1
Instruction: 10001100000000010000000001000000
clk = 0
PC: 00000000000000000000000000010100 | PC_next:
   00000000000000000000000000011000
ALUop: 000, opcode[2](neg) = 1, mux1out 00000000, mux2out 01000000, out1
   00000000, ALU out = 01000000, zero 0
writing - R1 - 01000000
clk = 1
Instruction: 00000110000000000000000000000001
clk = 0
PC: 00000000000000000000000000011000 | PC_next:
   00000000000000000000000001000000
ALUop: 000, opcode[2](neg) = 1, mux1out 01000000, mux2out 01000000, out1
   00000000, ALU out = 01000000, zero 0
clk = 1
Instruction: 10000100000000100000000100000011
clk = 0
PC: 00000000000000000000000001000000 | PC_next:
   00000000000000000000000001000100
ALUop: 000, opcode[2](neg) = 1, mux1out 11111011, mux2out 11111011, out1
   01000000, ALU out = 11111011, zero 0
writing - R2 - 11111011
clk = 1
Instruction: 00010011000111100000001000000011
clk = 0
PC: 00000000000000000000000001000100 | PC_next:
   00000000000000000000000000011110
ALUop: 001, opcode[2](neg) = 0, mux1out 00000101, mux2out 00000101, out1
   11111011, ALU out = 00000000, zero 1
clk = 1
Instruction: 00000000000000000000000000000000
clk = 0
PC: 00000000000000000000000000011110 | PC_next:
   00000000000000000000000000100010
ALUop: 000, opcode[2](neg) = 0, mux1out 00000000, mux2out 00000000, out1
```

```
    00000000, ALU out = 00000000, zero 1
[Done] exit with code=0 in 0.069 seconds
```

References

[1] S. Basnayake, “Simple 8-bit processor design and verilog implementation (part 1).” https://studentsxstudents.com/simple-8-bit-processor-design-and-verilog-implementation-part-1-8735fac284b, Aug 2021.

[2] “Barrel shifter.” https://esrd2014.blogspot.com/p/barrel-shifter.html.

[3] “Teach yourself verilog with this tiny cpu design.” https://hackaday.com/2015/08/01/teach-yourself-verilog-with-this-tiny-cpu-design/.

[4] “Verilog code for 16-bit risc processor.” https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html.