EECS/CSE31L Final Project Report
Group name: MITTT

Github repo: Final_Project

Mina Bedwany 69543570, Tianyi Yang 69386993, Takumi Okamoto 86483191

December 10, 2017

0.1 Introduction

This Final Project is a RISC-V processor which combines all the lab modules we have done throughout these several weeks. The structure of this RISC-V processor is basically three parts: **Datapath, Controller, and ALU control modules**. The arrangement of the processor is shown below.

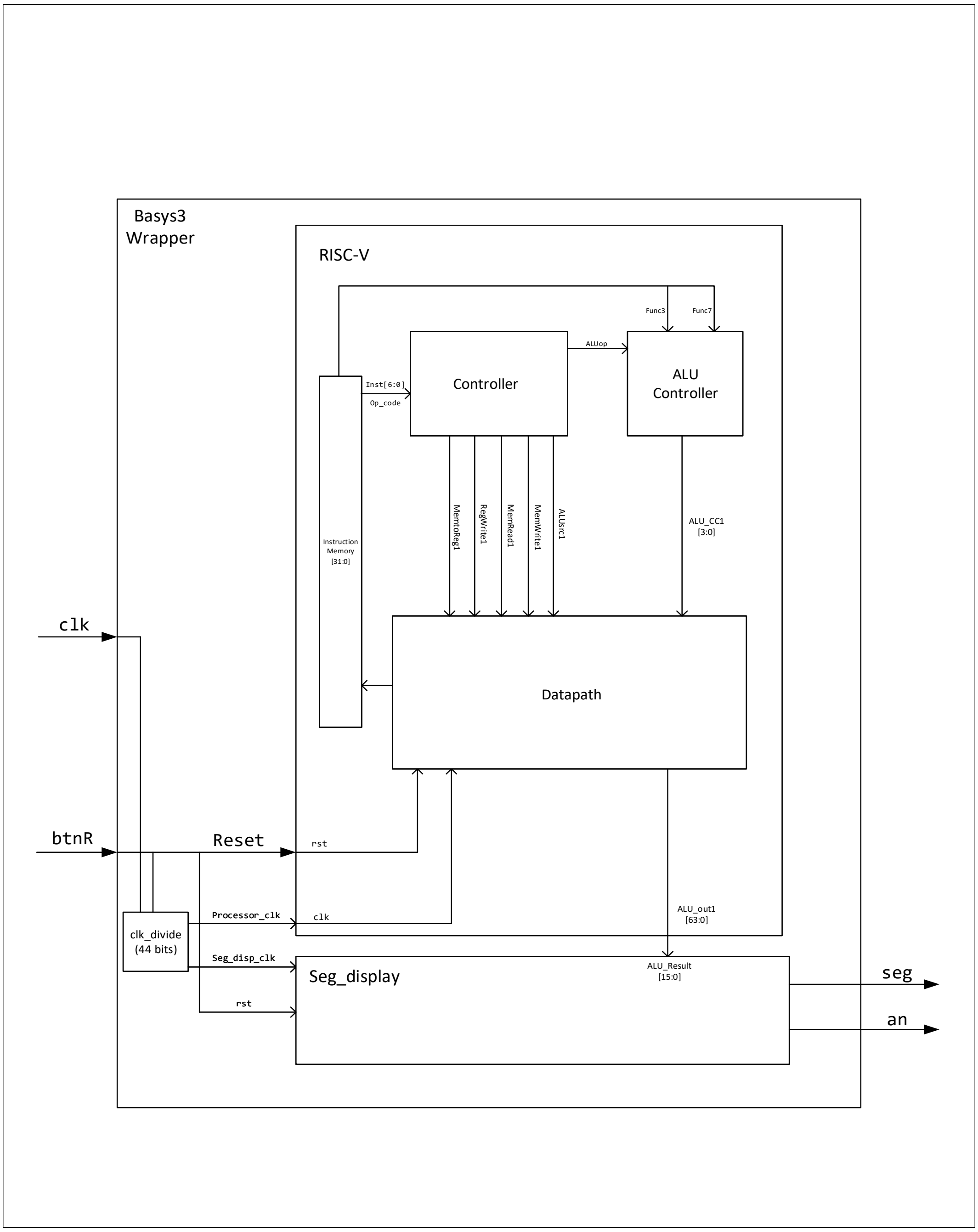

Figure 1: Block Design

0.2 Implementation

The whole project works in the following steps:

1. The utmost module called `"Basys3 Wrapper"` gets the clock signal `clk` and reset signal `btnR`, then divides the clock into two parts: `processor_clk` and `seg_disp_clk`.
2. The following RISC-V processor takes the clock and reset signal passed by the wrapper. Inside, there are four modules: `Instruction memory`, `controller`, `ALU controller`, and `Datapath`.
   (a) The controller takes the operation code `"op_code"` stored from the instruction memory. Then it outputs a responsively ALU operation code `"ALUop"` to the ALU controller with 5 other wires to the Datapath:

   `ALUSrc`

   `0:` **`2nd`** ALU operand comes from the second register file output

   `1:` **`2nd`** ALU operand is the sign - extended , lower 16 bits of the instruction.

   `MemtoReg`

   `0:` The value fed to the register Write data input comes from the ALU

   `1:` The value fed to the register Write data input comes from the data memory

   `RegWrite` `Regfile [ addr ]` is written with the value on the Write data input

   `MemRead` `Mem [ addr ]` is put on the Read data output

   `MemWrite` Write data input is written into `Mem[ addr ]`

   `ALUOp` Determine functionality of the ALU unit

   (b) The ALU controller takes the operation code passed by the controller. Depends on the `ALUop`, it takes two segments from the instruction memory called `"Func3"` and `"Func7"`. Finally, it will generate a **ALU**

Control Code and pass it to Datapath.

(c) The Datapath takes in total 8 inputs. Two of the input are `clk` from the `processor_clk` and `rst` from the `btnR`. For the rest of six inputs, five of them are provided by controller and the last on is from the ALU controller. Finally, the Datapath will update the Instruction Memory and pass another 64-bits output `ALU_result`.

3. The last module `seg_display` uses the clock signal from `seg_disp_clk`, the reset signal from `btnR`, and the first 16-bits from the `ALU_result` to process and generate our two last output `seg` and `an`.

Overall, the table below shows all the supported operations for our RISC-V processor.

Table 1: Supported RISC-V Instruction Set

imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

0.3 Test Results

0.3.1 Our Instruction Set for Testing

```
module instructionmemory#(
    parameter INS_ADDRESS = 9,
    parameter INS_W = 32
     )(
    input logic [ INS_ADDRESS -1:0] ra , // Read address of the instruction memory , comes from PC
    output logic [ INS_W -1:0] rd // Read Data
    );


logic [INS_W-1 :0] Inst_mem [(2**(INS_ADDRESS-2))-1:0];
//                                                                               Operation  HEX OUTPUT      DECIMAL OUTPUT
assign Inst_mem[0]  = 32'h00007033;//  0000000 00000 00000 111 00000 011 0011;     and      0               0
assign Inst_mem[1]  = 32'h00110093;//  0000 0000 0001 00010 000 00001 001 0011;    addi     1               1
assign Inst_mem[2]  = 32'h01208113;//  0000 0001 0010 00001 000 00010 001 0011;    addi     13              19
assign Inst_mem[3]  = 32'h30300193;//  0011 0000 0011 00000 000 00011 001 0011;    addi     303             771
assign Inst_mem[4]  = 32'h06438213;//  0000 0110 0100 00111 000 00100 001 0011;    addi     64              100
assign Inst_mem[5]  = 32'h80100293;//  1000 0000 0001 00000 000 00101 001 0011;    addi     fffffffffffff801   -2047
assign Inst_mem[6]  = 32'h0e618313;//  0000 1110 0110 00011 000 00110 001 0011;    addi     3e9             1001
assign Inst_mem[7]  = 32'hd0718393;//  1101 0000 0111 00011 000 00111 001 0011;    addi     a               10
assign Inst_mem[8]  = 32'h00318433;//  0000000 00011 00011 000 01000 011 0011;     add      606             1542
assign Inst_mem[9]  = 32'h405404b3;//  0100000 00101 01000 000 01001 011 0011;     sub      e05             3589
assign Inst_mem[10]  = 32'h0079f533;//  0000000 00111 10011 111 01010 011 0011;    and      0               0
assign Inst_mem[11]  = 32'h002165b3;//  0000000 00010 00010 110 01011 011 0011;    or       13              19
assign Inst_mem[12]  = 32'h12a1a8a3;//  0001001 01010 00011 010 10001 010 0011;    sw       434             1076
assign Inst_mem[13]  = 32'h108420a3;//  0001000 01000 01000 010 00001 010 0011;    sw       707             1799
assign Inst_mem[14]  = 32'h0104a603;//  0000 0001 0000 01001 010 01100 000 0011;   lw       e15             3605

assign rd =  Inst_mem [ra[INS_ADDRESS-1:2]];


endmodule
```

0.3.2 Simulated Waveforms

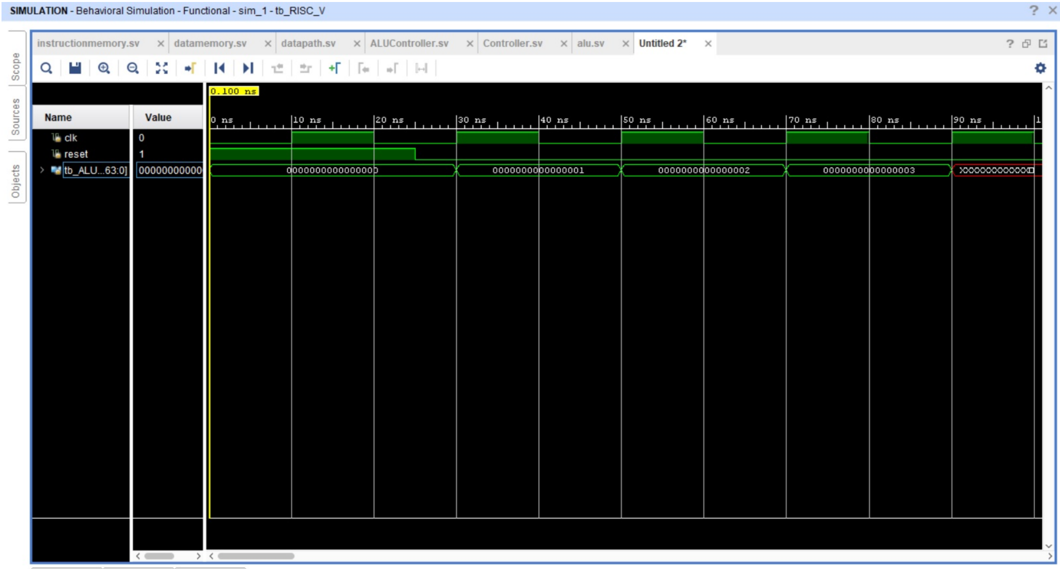

Figure 2: Simulated Waveform for Instruction Set 1

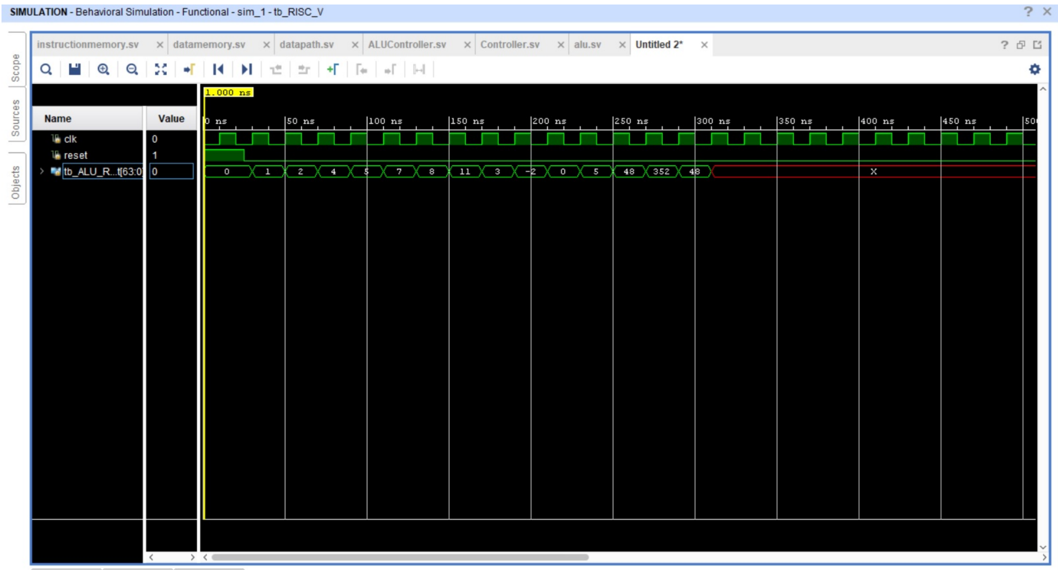

Figure 3: Simulated Waveform for Instruction Set 2

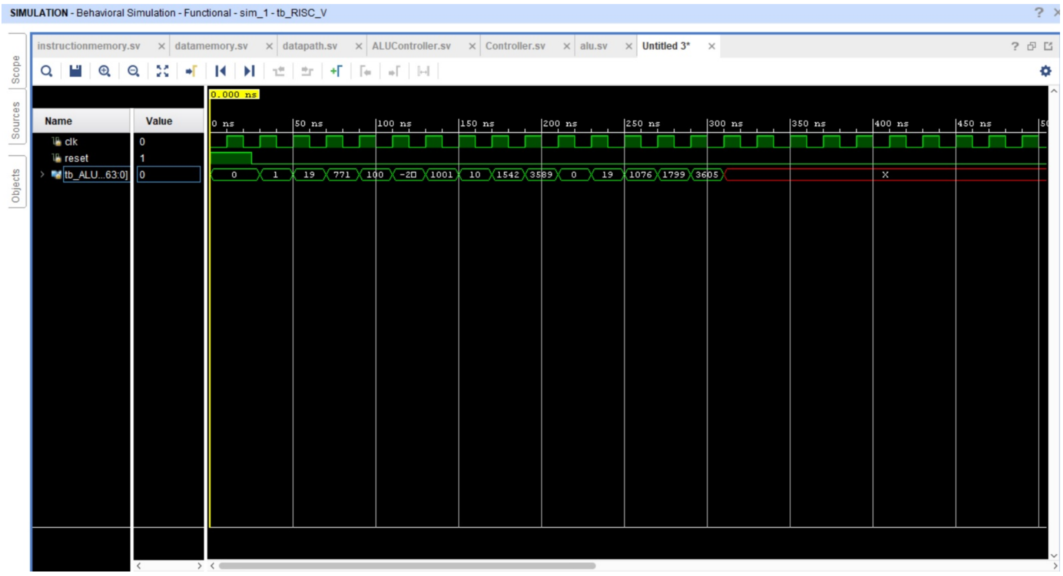

Figure 4: Simulated Waveform for Instruction Set 3