Single Cycle RISC-V datapath and Control

University of California Irvine

28763963 Yuki Hayashi
79364141 David Tiao

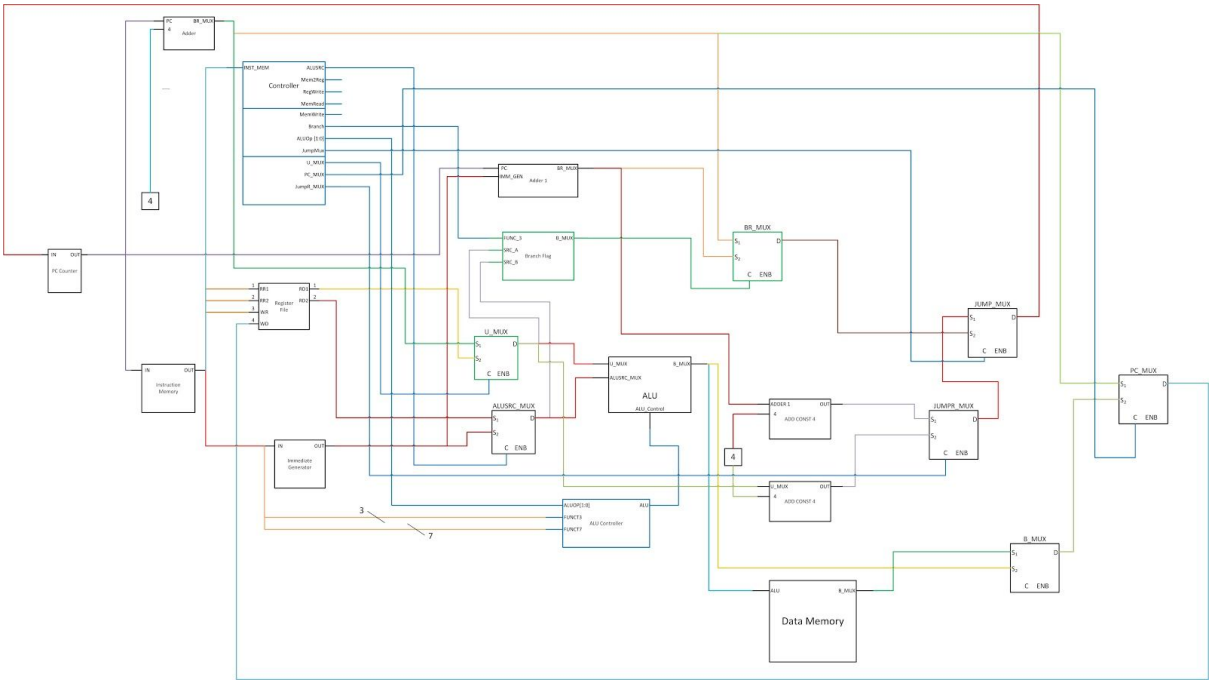
Introduction

The goal of this assignment is to implement the new instruction in SystemVerilog. For this processor implementation, we will be incorporating Integer Computational Instructions, Register Immediate Instructions, Register-Register Instructions, Control Transfer, and Load/Store instructions. In addition, there will be conditional branching support (B-Type instructions) and Unconditional Jumps (J-Type format). The following table is the new instruction set.

Block Diagram & Explanations

The entire design

This is our new RISC-V design. For implementing the new instructions, multiple modules are added to the previous one: branh_frag, u_mux, br_mux, jump_mux, jumpr_mux, pc_mux, and an adder. Also, some files are modified to connect these new modules: immediate generator, ALU, ALU controller, Controller, DataMemoery, and Datapath.



How we have implemented the new instructions (module):

Immediate Generator

added: U-type (LUI, AUIPC), J-type(JAL, JALR), B-type immediate generator

To implement all instructions of different types of the datapath, the first modification is to add U-type, J-type, and B-type immediate generator in the imm_gen.sv. By using the case statement, the program receives the 32-bit instruction codes and looks up the last 7-bit numbers to determines the types of instructions. Then, it returns the immediate-generated numbers as Imm_out.

```
7'b0000011 /*I-type load*/     :
    Imm_out = {inst_code[31]? {20{1'b1}}:{20{1'b0}} , inst_code[31:20]};
7'b0010011 /*I-type addi*/     :
    Imm_out = {inst_code[31]? {20{1'b1}}:{20{1'b0}} , inst_code[31:20]};
7'b0100011 /*S-type*/    :
    Imm_out = {inst_code[31]? {20{1'b1}}:{20{1'b0}} , inst_code[31:25], inst_code[11:7]};
7'b0110111/*U-type LUI*/ :
    Imm_out = {inst_code[31]? {20{1'b1}}:{20{1'b0}}, inst_code[30:20], inst_code[19:12], 12'b0};
7'b0010111/*U-type AUIPC*/ :
    Imm_out = {inst_code[31]? {20{1'b1}}:{20{1'b0}}, inst_code[30:20], inst_code[19:12], 12'b0};
7'b1101111/*JAL*/:
    Imm_out = {inst_code[31]? {10{1'b1}}: {10{1'b0}}, inst_code[19:12], inst_code[20], inst_code[30:25], inst_code[24:21], 1'b0};
7'b1100111/*JALR*/:
    Imm_out = {inst_code[31]? {20{1'b1}}:{20{1'b0}} , inst_code[31:20]};
7'b1100011/*B-type*/:
    Imm_out = {inst_code[31] ? {20{1'b1}}:{20{1'b0}}, inst_code[7], inst_code[30:25], inst_code[11:8], 1'b0};
default                  :
    Imm_out = {32'b0};
```

ALU

added: SLL, SRL, SRA, SLT, SLTU

In the ALU.sv, the new instructions are added: SLL, SRL, SRA, SLT, SLTU. This module does the calculation depending on the 4-bit input. We assigned those new operation bits to determine what ALU has to be operated.

```
case(Operation)

4'b0000:        //add
        ALUResult = SrcA + SrcB;
4'b0001:        //sub
        ALUResult = $signed(SrcA) - $signed(SrcB);
4'b0010:        //AND
        ALUResult = SrcA & SrcB;
4'b0011:        //OR
        ALUResult = SrcA | SrcB;
4'b0100:        //xor
        ALUResult = SrcA ^ SrcB;
4'b0101:        //SLL
        ALUResult = SrcA << SrcB;
4'b0110:        //SRL
        ALUResult = SrcA >> SrcB;
4'b0111:        //SRA
        ALUResult = $signed(SrcA) >>> SrcB;
4'b1000:        //SLT
        ALUResult = (($signed(SrcA) - $signed(SrcB)) < 0) ? 32'd1 : 32'b0;
4'b1001:        //SLTU
        ALUResult = ((SrcA - SrcB) < 0) ? 32'd1 : 32'b0;
default:
        ALUResult = 'b0;
endcase
```

	Operation			
	Op[3]	Op[2]	Op[1]	Op[0]
Add	0	0	0	0
Sub	0	0	0	1
And	0	0	1	0
Or	0	0	1	1
Xor	0	1	0	0
SLL	0	1	0	1
SRL	0	1	1	0
SRA	0	1	1	1
SLT	1	0	0	0
SLTU	1	0	0	1

ALUController

chooses the control signal and modify the operation depending on the signal

To send the correct control signal to ALU, ALUController.sv needs to be modified. The ALU controller receives the 2-bit ALUOp, 7-bit Funct7, 3-bit Funct3. These inputs are used to assign 4-bit operations for each instruction, which will be sent to the ALU.sv to do the ALU calculation.

```
assign Operation[0]=(
                    ( (ALUOp==2'b10) && (Funct7==7'b0100000) && (Funct3==3'b000) ) ||//sub
                    ( (ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b110) ) ||//or
                    ( (ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b001) ) ||//sll
                    ( (ALUOp==2'b10) && (Funct7==7'b0100000) && (Funct3==3'b101) ) ||//sra
                    ( (ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b011) ) //sltu
                  );
//and, or, srl, sra
assign Operation[1]=(
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b111)) ||//and
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b110)) ||//or
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b101)) ||//srl
                    ((ALUOp==2'b10) && (Funct7==7'b0100000) && (Funct3==3'b101)) //sra

                   );
//xor, sll, srl, sra
assign Operation[2]=(
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b100)) || //xor
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b001)) || //sll
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b101)) || //srl
                    ((ALUOp==2'b10) && (Funct7==7'b0100000) && (Funct3==3'b101)) //sra
                  );

//slt, sltu
assign Operation[3]=(
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b010)) ||//slt
                    ((ALUOp==2'b10) && (Funct7==7'b0000000) && (Funct3==3'b011))//sltu

                  );
```

Controller

assigned each control signal bit depending on the opcode.

To implement this control signal assignment, first of all, we added new signal bits (LUI, JAL, JALR, AUIPC, beq, Branch, JumpMux, Umux, PCMux).

Control Signal									
input or output	signal name	R-format	ld	sd	beq	LUI	JAL	JALR	AUIPC
input	I[6]	0	0	0	1	0	0	1	1
	I[5]	1	0	1	1	1	0	1	1
	I[4]	1	0	0	0	1	1	0	0
	I[3]	0	0	0	0	0	0	1	0
	I[2]	0	0	0	0	1	1	1	1
	I[1]	1	1	1	1	1	1	1	1
	I[0]	1	1	1	1	1	1	1	1
output	ALUSrc	0	1	1	0	0	0	0	0
	MemtoReg	0	1	x	x	0	0	0	0
	RegWrite	1	1	0	0	1	0	0	0
	MemRead	0	1	0	0	0	0	0	0
	MemWrite	0	0	1	0	0	1	1	1
	Branch	0	0	0	1	0	0	0	0
	ALUOp1	1	0	0	0	0	0	0	0
	ALUOp0	0	0	0	1	0	0	0	0
	JumpMux	0	0	0	0	0	1	1	0
	Umux	0	0	0	0	0	0	0	1
	PCMux	0	0	0	0	0	1	1	0

In this module, each control signal 7-bit is assigned as variables. Then, those variables are assigned if the given opcode matches the 7-bit input opcode.

```
  logic [6:0] R_TYPE, LW, SW, RTypeI, BR, LUI, AUIPC, JAL, JALR;
  assign  R_TYPE = 7'b0110011;
  assign  LW     = 7'b0000011;
  assign  SW     = 7'b0100011;
  assign  RTypeI = 7'b0010011; //addi,ori,andi
  assign  BR     = 7'b1100011;

  assign LUI = 7'b0110111;
  assign AUIPC = 7'b0010111;
  assign JAL = 7'b1101111;
  assign JALR = 7'b1100111;



assign ALUSrc   = (Opcode==LW || Opcode==SW || Opcode==AUIPC || Opcode==RTypeI || Opcode==LUI );
assign MemtoReg = (Opcode==LW);
assign RegWrite = (Opcode==R_TYPE || Opcode==LW || Opcode==LUI || Opcode==RTypeI || Opcode== JAL || Opcode==JALR);
assign MemRead  = (Opcode==LW);
assign MemWrite = (Opcode==SW);
assign Branch   = (Opcode==BR);
assign JumpMux  = (Opcode==JAL || Opcode==JALR);
assign Umux     = (Opcode==AUIPC);
assign PCMux    = (Opcode==JAL || Opcode==JALR);
assign JumpRMux = (Opcode==JALR);

assign ALUOp[0] = (Opcode==BR);
assign ALUOp[1] = (Opcode==RTypeI || R_TYPE);
```

DataMemory

split the address to implement for LB, LBU, LH, LHU, LW, SB, SH, SW

To implement LB, LH, SB, SH, the address of mem is extended 4 times to handle different size of the address. In the save and load part, the funct3 input determines to handle binary, half or word. For load half and binary, sign extended bits are concatenated as needed.

```
logic [8-1:0] mem [4*(2**DM_ADDRESS)-1:0];
```

```
always_comb
begin
    if(MemRead)begin
        if(Funct3[1])begin          //LW
            rd = {mem[a+3], mem[a+2], mem[a+1], mem[a]};
        end else if(Funct3[0])begin //LB, LBU
            rd = {(mem[a+1][7]&&!Funct3[2])? {{16{1'b1}}, mem[a+1], mem[a]}:{{16{1'b0}}, mem[a+1], mem[a]} };
        end else begin              //LH, LHU
            rd = (mem[a][7]&&!Funct3[2])? {{24{1'b1}}, mem[a]}:{{24{1'b0}}, mem[a]};
        end
    end
end
```

```
always @(posedge clk) begin
    if(MemWrite)begin
        if(Funct3[1])begin          //SW
            mem[a+3] = wd[31:24];
            mem[a+2] = wd[23:16];
            mem[a+1] = wd[15:8];
            mem[a] = wd[7:0];
        end else if(Funct3[0])begin //SH
            mem[a+1] = wd[15:8];
            mem[a] = wd[7:0];
        end else begin              //SB
            mem[a] = wd[7:0];
        end
    end
end
```

DataPath

connect everything

How we have implemented special instructions:

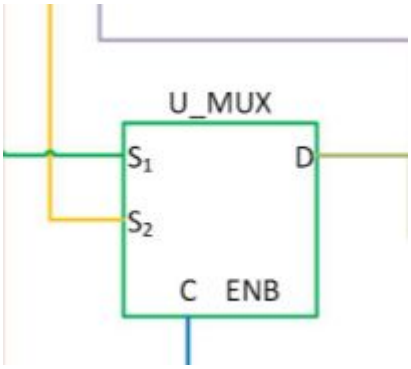
U-type

LUI (load upper immediate)

places the immediate value to the top 20 bits of rd and fill in the lowest 12 bits with zeros.

AUIPC (add upper immediate to pc)

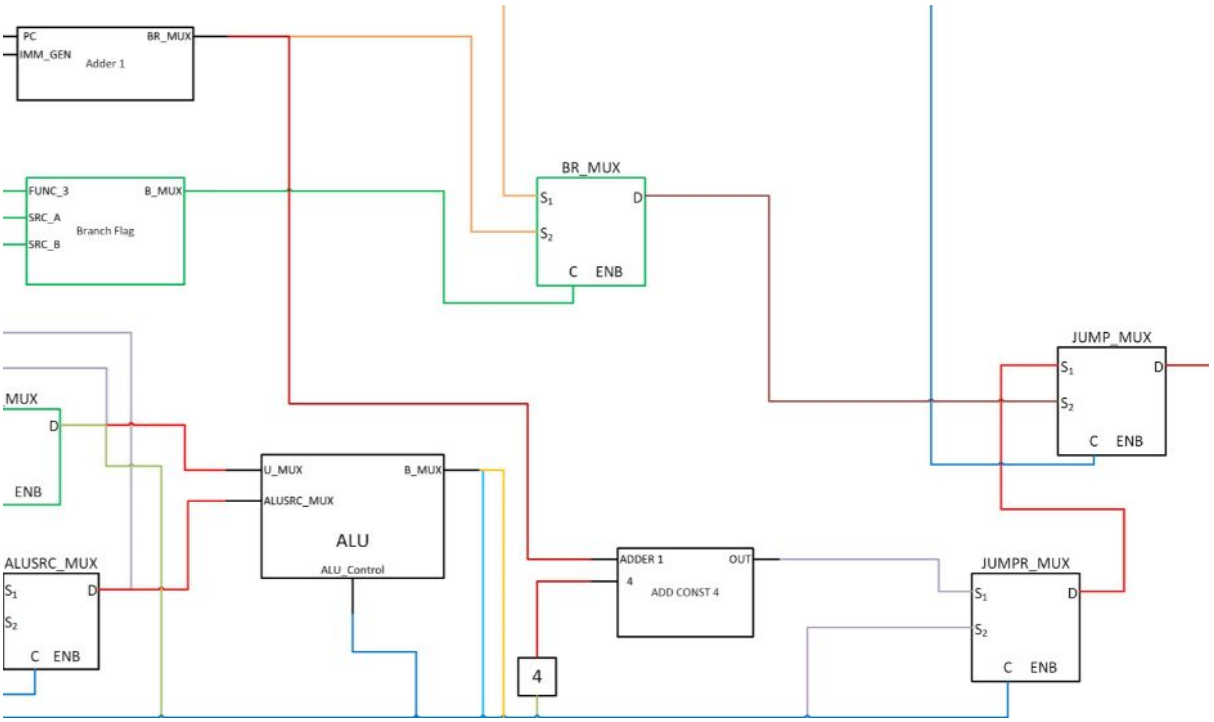
forms 32 bits offset from 20-bit immediate, fills in the lowest 12 bits with zeros, add this offset to the pc and places the result in rd.



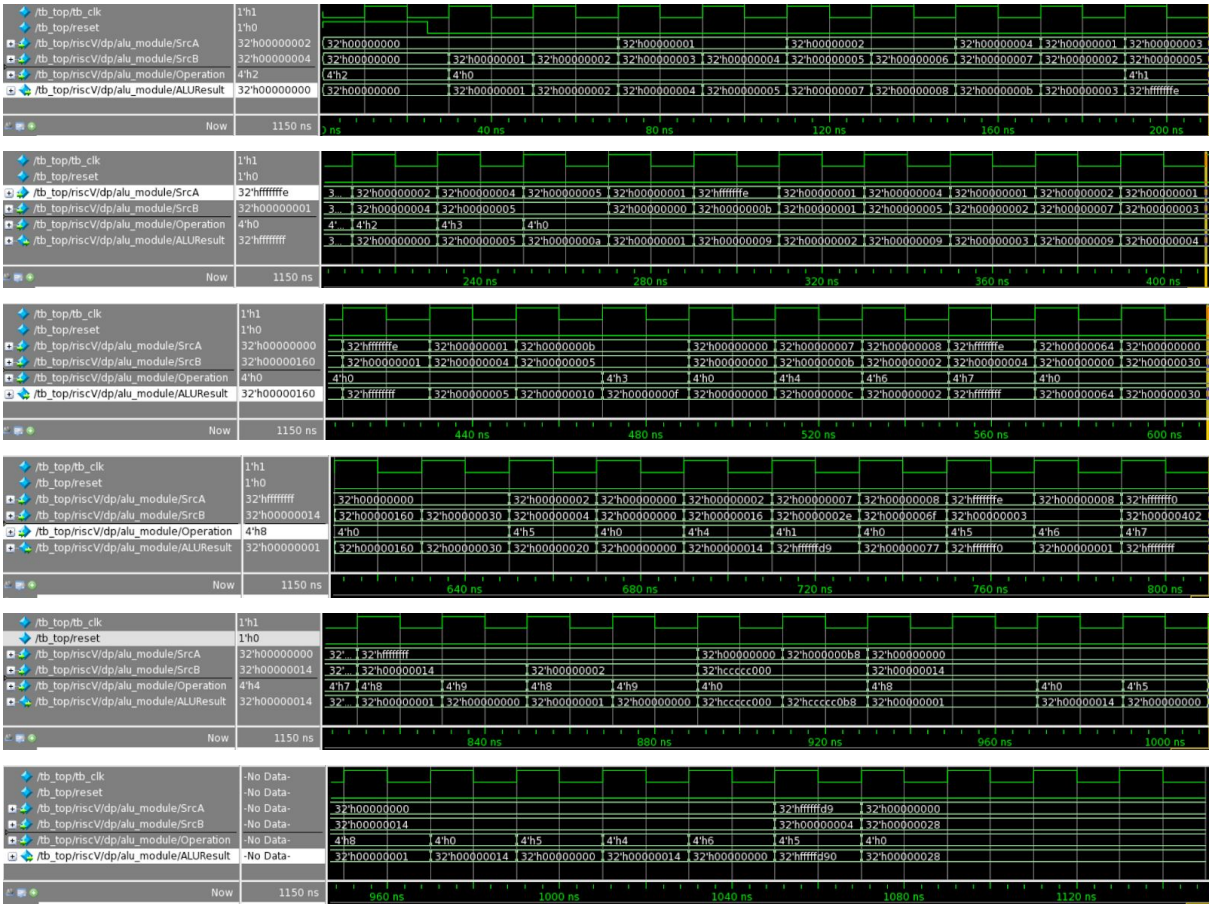
These two u type operations are implemented by using u-mux place between register and ALU. The mux gets input1 from pc, input2 from the register, and a control signal(U_mux) from the controller.

Branch

compares rs1 and rs2 are equal, unequal, greater than or equal, less than and takes to the destination(pc+4 + immediate) if they are true.



Screenshot of the waveform



Synthesis

Critical path length
Critical path slack
Area
Power report

```
Timing Path Group 'clk'
-----------------------------------
Levels of Logic:              20.00
Critical Path Length:          9.58
Critical Path Slack:           0.41
Critical Path Clk Period:     10.00
Total Negative Slack:          0.00
No. of Violating Paths:        0.00
Worst Hold Violation:          0.00
Total Hold Violation:          0.00
No. of Hold Violations:        0.00
-----------------------------------
```

```
Area
-----------------------------------
Combinational Area:    10349.506126
Noncombinational Area:  6994.043106
Buf/Inv Area:            724.310403
Total Buffer Area:           410.19
Total Inverter Area:         314.12
Macro/Black Box Area:  50667.683594
Net Area:               6807.302180
-----------------------------------
Cell Area:             68011.232826
Design Area:           74818.535006


Design Rules
-----------------------------------
Total Number of Nets:          5363
Nets With Violations:             7
Max Trans Violations:             0
Max Cap Violations:               7
-----------------------------------
```

```
------------------------------------------------------------------------------
                                  Switch   Int      Leak     Total
Hierarchy                         Power    Power    Power    Power    %
------------------------------------------------------------------------------
riscv                              30.322  737.353 1.28e+10 1.35e+04 100.0
  dp (Datapath)                    30.406  737.087 1.28e+10 1.35e+04  99.8
    data_mem (datamemory)           0.109   27.613 1.48e+04   27.737   0.2
    brmuxPlus4 (adder_WIDTH32_1)      N/A    0.849 1.04e+08  103.525   0.8
    brmux (mux2_WIDTH32_1)          0.396    0.753 5.75e+07   58.653   0.4
    bradder (adder_WIDTH32_2)         N/A    1.234 9.38e+07   94.644   0.7
    alu_module (alu)                7.040   21.606 1.21e+09 1.24e+03   9.2
    srcbmux (mux2_WIDTH32_2)        1.675    0.537 1.99e+07   22.097   0.2
    Ext_Imm (imm_Gen)               1.010    0.285 4.09e+07   42.222   0.3
    umux (mux2_WIDTH32_0)           2.062    0.502 3.67e+07   39.286   0.3
    rf (RegFile)                   14.265  660.229 1.07e+10 1.14e+04  84.0
    pcreg (flopr_WIDTH32)           1.236   18.643 2.68e+08  287.467   2.1
    pcadd (adder_WIDTH32_0)           N/A    1.001 9.09e+07   91.568   0.7
```