CS21 Project: MIPS Single Cycle Processor Extension

Carl David B. Ragunton

2020-04243

Video Explanation Link:

https://drive.google.com/file/d/18EfynYqaeFWfHZhV-uPPuv44wRVp3nYx/view?usp=sharing

I will provide no summary because the info just can't be summarized. They need to be explained by parts.

Changes Made (Line-by-Line Explanation):

This is the part where all the changes in all the parts will be placed and will be explained line-by-line. Note that the line-by-line explanation here is not where how I managed to create the new instructions. These are just purely objective explanations of them. The process of the added instructions will be explained on another part but there are comments regarding those new parts. For now, I will just disregard them for the optimal experience of explaining the changes. I will not include the memfile.mem in this part because I change it for every instructions.

top.sv

```
1  // CS 21 Lab2 -- S2 AY 2021-2022
2  // Carl David B. Ragunton -- 6/5/2022
3  // top.sv -- edited top.sv
4  `timescale 1ns / 1ps
5  module top(input  logic        clk, reset,
6             output logic [31:0] writedata, dataadr,
7             output logic        memwrite,
8             input logic [5:0] opcode);//sb
9
10   logic [31:0] pc, instr, readdata;
11
12   // instantiate processor and memories
13   mips mips(clk, reset, pc, instr, memwrite, dataadr,
14             writedata, readdata, opcode);//sb
15   imem imem(pc[7:2], instr);
16   dmem dmem(clk, memwrite, dataadr, writedata, readdata,
opcode);//sb
17 endmodule
```

Line4: Sets the timescale of the simulation to be 1ns per 1ps

Line5-8: Creates the module top and declares the variables inside it

Line10: Declares other variables that are neither input nor output of this module

Line13: Instantiates the mips module which creates mips

Line15: Instantiates the imem module which creates imem

Line16: Instantiates the dmem module which creates dmem

Line17: End of module

controller.sv

```
1  // CS 21 Lab2 -- S2 AY 2021-2022
2  // Carl David B. Ragunton -- 6/5/2022
3  // controller.sv -- edited controller.sv
4  `timescale 1ns / 1ps
5  module controller(input  logic [5:0] op, funct,
6                    input  logic       zero,less,
7                    output logic       memtoreg, memwrite,
8                    output logic       pcsrc, alusrc,
9                    output logic       regdst, regwrite,
10                   output logic       jump,
11                   output logic [3:0] alucontrol);//ble
12
13   logic [1:0] aluop;
14   logic       branch;
15
16   maindec md(op, memtoreg, memwrite, branch,
17              alusrc, regdst, regwrite, jump, aluop);
18   aludec  ad(funct, aluop, alucontrol,op);  //li
19
20   always_comb//ble
21     case(op)
22       6'b011111: pcsrc = (branch & zero) | (branch & less);
23       default:pcsrc = branch & zero;
24     endcase
25
26 endmodule
```

Line4: Sets the timescale of the simulation to be 1ns per 1ps

Line5-11: Creates the module controller and declares the variables inside it

inputs: 6 bit op and funct, zero and less

outputs: memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, 4 bit alucontrol

Line13-14: Declares other variables that are neither input nor output of this module

2 bit aluop, branch

Line16-17: Instantiates the maindec module which creates md

Line18: Instantiates the aludec module which creates ad

Line20: always_comb for the next lines

Line21-24: Checks the value of op and performs the appropriate action:

6'b011111: pcsrc = (branch & zero) | (branch & less);

default:pcsrc = branch & zero;

Line26: End of module

dmem.sv

```
3  // dmem.sv -- edited dmem.sv
4  `timescale 1ns / 1ps
5  module dmem(input  logic        clk, we,
6              input  logic [31:0] a, wd,
7              output logic [31:0] rd,
8              input logic [5:0] opcode);
9
10   logic [31:0] RAM[63:0];
11
12   assign rd = RAM[a[31:2]]; // word aligned
13
14   always_ff @(posedge clk)
15     case(opcode)//sb
16       6'b101000:
17         if (we)
18           case(a%4)
19             'b0: RAM[a[31:2]][31:24] <= wd[7:0];
20             'b1: RAM[a[31:2]][23:16] <= wd[7:0];
21             'b10: RAM[a[31:2]][15:8] <= wd[7:0];
22             'b11: RAM[a[31:2]][7:0] <= wd[7:0];
23           endcase
24       default: if (we) RAM[a[31:2]] <= wd;
25     endcase
26 endmodule
```

Line4: Sets the timescale of the simulation to be 1ns per 1ps

Line5-8: Creates the module dmem and declares the variables inside it

inputs: clk, we, 32 bit a and wd, 6 bit opcode

output: 32 bit rd

Line10: Declares other variables that are neither input nor output of this module

32 bit of 64 bit RAM

Line12: Set rd=RAM[a[31:2]]

Line14: always_ff @(posedge clk) for the next lines

Line15-25: Checks the value of opcode and performs the appropriate action…

Line18-23: If opcode==101000, check the value of (a%4) and performs the appropriate action

6'b101000:

if (we)

case(a%4)

'b0: RAM[a[31:2]][7:0] <= wd[7:0];

'b1: RAM[a[31:2]][15:8] <= wd[7:0];

'b10: RAM[a[31:2]][23:16] <= wd[7:0];

'b11: RAM[a[31:2]][31:24] <= wd[7:0];

endcase

default: if (we) RAM[a[31:2]] <= wd;

Line26: End of module

maindec.sv

```
1  // CS 21 Lab2 -- S2 AY 2021-2022
2  // Carl David B. Ragunton -- 6/5/2022
3  // maindec.sv -- edited maindec.sv
4  `timescale 1ns / 1ps
5  module maindec(input  logic [5:0] op,
6                 output logic       memtoreg, memwrite,
7                 output logic       branch, alusrc,
8                 output logic       regdst, regwrite,
9                 output logic       jump,
10                output logic [1:0] aluop);
11
12   logic [8:0] controls;
13
14   assign {regwrite, regdst, alusrc, branch, memwrite,
15           memtoreg, jump, aluop} = controls;
16
17   always_comb
18     case(op)
19       6'b000000: controls <= 9'b110000010; // RTYPE
20       6'b100011: controls <= 9'b101001000; // LW
21       6'b101011: controls <= 9'b001010000; // SW
22       6'b101000: controls <= 9'b001010000; // SB
23       6'b000100: controls <= 9'b000100001; // BEQ
24       6'b011111: controls <= 9'b000100001; // BLE
25       6'b001000: controls <= 9'b101000000; // ADDI
26       6'b010001: controls <= 9'b101000000; // LI
27       6'b000010: controls <= 9'b000000100; // J
28       default:   controls <= 9'bxxxxxxxxx; // illegal op
29     endcase
30 endmodule
```

Line4: Sets the timescale of the simulation to be 1ns per 1ps

Line5-10: Creates the module maindec and declares the variables inside it

input: 5 bit op

outputs: memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, 2 bit aluop

Line12: Declares other variables that are neither input nor output of this module

9 bit controls

Line14: Set {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop} = controls (by the number of the bits in order)

Line17: always_comb for the next lines

Line18-29: Checks the value of op and performs the appropriate action:

6'b000000: controls <= 9'b110000010; // RTYPE

6'b100011: controls <= 9'b101001000; // LW

6'b101011: controls <= 9'b001010000; // SW

6'b101000: controls <= 9'b001010000; // SB

6'b000100: controls <= 9'b000100001; // BEQ

6'b011111: controls <= 9'b000100001; // BLE

6'b001000: controls <= 9'b101000000; // ADDI

6'b010001: controls <= 9'b101000000; // LI

6'b000010: controls <= 9'b000000100; // J

default: controls <= 9'bxxxxxxxxx; // illegal op

Line30: End of module

mips.sv

```
1  // CS 21 Lab2 -- S2 AY 2021-2022
2  // Carl David B. Ragunton -- 6/5/2022
3  // mips.sv -- edited mips.sv
4  `timescale 1ns / 1ps
5  module mips(input  logic        clk, reset,
6              output logic [31:0] pc,
7              input  logic [31:0] instr,
8              output logic        memwrite,
9              output logic [31:0] aluout, writedata,
10             input  logic [31:0] readdata,
11             output logic [5:0] opcode);//sb
12
13   assign opcode = instr[31:26];//sb
14
15   logic       memtoreg, alusrc, regdst,
16               regwrite, jump, pcsrc, zero,less;//ble
17   logic [3:0] alucontrol;
18
19   controller c(instr[31:26], instr[5:0], zero,less,
20                memtoreg, memwrite, pcsrc,
21                alusrc, regdst, regwrite, jump,
22                alucontrol);//ble
23   datapath dp(clk, reset, memtoreg, pcsrc,
24               alusrc, regdst, regwrite, jump,
25               alucontrol,
26               zero,less, pc, instr,
27               aluout, writedata, readdata);//ble
28 endmodule
```

Line4: Sets the timescale of the simulation to be 1ns per 1ps

Line5-11: Creates the module mips and declares the variables inside it

inputs: clk and reset, 32 bit instr and readdata

outputs: 32 bit pc, aluout, and writedata, 6 bit opcode, memwrite

Line13: Set opcode=instr[31:26]

Line15-17: Declares other variables that are neither input nor output of this module

memtoreg, alusrc, regdst, regwrite, jump, pcsrc, zero, less, 4 bit alucontrol

Line19-22: Instantiates the controller module which creates c

Line23-27: Instantiates the datapath module which creates dp

Line28: End of module

aludec.sv

```
1  // CS 21 Lab2 -- S2 AY 2021-2022
2  // Carl David B. Ragunton -- 6/5/2022
3  // aludec.sv -- edited aludec.sv
4  ////////////
5  `timescale 1ns / 1ps
6  module aludec(input  logic [5:0] funct,
7                input  logic [1:0] aluop,
8                output logic [3:0] alucontrol,
9                input[5:0] op); //li
10
11   always_comb
12     case(aluop)
13       2'b00:
14         case(op)
15           6'b010001: alucontrol <= 4'b0110; //li
16           default: alucontrol <= 4'b0010;  // add (for
   lw/sw/addi/sb)
17         endcase
18       2'b01: alucontrol <= 4'b1010;  // sub (for beq and ble)
19       default: case(funct)          // R-type instructions
20           6'b000000: alucontrol <= 4'b0100; // sll
21           6'b100000: alucontrol <= 4'b0010; // add
22           6'b100010: alucontrol <= 4'b1010; // sub
23           6'b100100: alucontrol <= 4'b0000; // and
24           6'b100101: alucontrol <= 4'b0001; // or
25           6'b101010: alucontrol <= 4'b1011; // slt
26           6'b110011: alucontrol <= 4'b0101; // zfr
27           default:   alucontrol <= 4'bxxxx; // ???
28         endcase
29     endcase
30 endmodule
```

Line5: Sets the timescale of the simulation to be 1ns per 1ps

Line6-9: Creates the module aludec and declares the variables inside it

inputs: 6 bit funct and op, 2 bit aluop

output: 4 bit alucontrol

Line11: always_comb for the next lines

Line12-29: Checks the value of aluop and performs the appropriate action…

Line13-17: If aluop==00, check the value of alucontrol and performs the appropriate action…

Line19-28: If aluop is not equal to 00 or 01, check the value of funct and performs the appropriate action:

```
case(aluop)
        2'b00:
        case(op)
                6'b010001: alucontrol <= 4'b0110; //li
                default: alucontrol <= 4'b0010;  // add (for lw/sw/addi/sb)
        endcase
        2'b01: alucontrol <= 4'b1010;  // sub (for beq and ble)
        default: case(funct)         // R-type instructions
                6'b000000: alucontrol <= 4'b0100; // sll
                 6'b100000: alucontrol <= 4'b0010; // add
                6'b100010: alucontrol <= 4'b1010; // sub
                6'b100100: alucontrol <= 4'b0000; // and
                6'b100101: alucontrol <= 4'b0001; // or
                6'b101010: alucontrol <= 4'b1011; // slt
                6'b110011: alucontrol <= 4'b0101; // zfr
                default:   alucontrol <= 4'bxxxx; // ???
                endcase
endcase
```

Line30: End of module

datapath.sv

```
// CS 21 Lab2 -- S2 AY 2021-2022
// Carl David B. Ragunton -- 6/5/2022
// datapath.sv -- edited datapath.sv
////////////
`timescale 1ns / 1ps
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [3:0]  alucontrol,
                output logic        zero,less,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);//ble

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb,srcB,shamt; //sll
  logic [31:0] result;
  logic alusrcb;//sll
  logic [4:0] instr2521;//sll


  assign alusrcb = (alucontrol[3:0]==4'b0100) ? 1 : 0;//sll
  assign shamt = {27'b0,instr[10:6]};//sll

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder #(32) pcadd1(pc, 32'b100, 'b0, pcplus4); //So we adjust this
to use the more complex adder; wmt-modification
  sl2         immsh(signimm, signimmsh);
  adder #(32) pcadd2(pcplus4, signimmsh, 'b0, pcbranch); //See
comment above
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnext);

  // register file logic
  mux2 #(32)
instr2521sll(instr[25:21],instr[20:16],alusrcb,instr2521);//sll
  regfile     rf(clk, regwrite, instr2521, instr[20:16],
                 writereg, result, srca, writedata);//sll
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  mux2 #(32)  srcBmux(srcb,shamt,alusrcb,srcB);//sll
  alu         alu(srca, srcB, alucontrol, aluout,
zero,less);//sll,ble
endmodule
```

Line5: Sets the timescale of the simulation to be 1ns per 1ps

Line6-15: Creates the module datapath and declares the variables inside it

inputs: clk, reset, memtoreg, pcsrc, alusrc, regdst,regwrite, jump,4 bit alucontrol, 32 bit instr and readdata

outputs: zero and less,32 bit aluout and writedata

Line17-23: Declares other variables that are neither input nor output of this module

alusrcb, 5 bit writereg and instr2521, 32 bit pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, srca, srcb, srcB, shamt, result

Line26: Set alusrcb if (alucontrol[3:0]==4'b0100); otherwise alusrcb=0

Line27: Set shamt={27'b0, instr[10:6]}

Line30: Instantiates the flopr module which creates pcreg

Line31: Instantiates the adder module which creates pcadd1

Line32: Instantiates the sl2 module which creates immsh

Line33: Instantiates the adder module which creates pcadd2

Line34: Instantiates the mux2 module which creates pcbrmux

Line35-36: Instantiates the mux2 module which creates pcmux

Line39: Instantiates the mux2 module which creates instr2521sll

Line40-41: Instantiates the regfile module which creates rf

Line42-43: Instantiates the mux2 module which creates wrmux

Line44: Instantiates the mux2 module which creates resmux

Line45: Instantiates the signext module which creates se

Line48: Instantiates the mux2 module which creates srcbmux

Line49: Instantiates the mux2 module which creates srcBmux

Line50: Instantiates the alu module which creates alu

Line51: End of module

alu.sv

```
// CS 21 Lab2 -- S2 AY 2021-2022
// Carl David B. Ragunton -- 6/5/2022
// alu.sv -- edited alu.sv
module alu(input  logic [31:0] a, b,
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero,less);//ble

  logic [31:0] condinvb, sum;

  assign condinvb = alucontrol[3] ? ~b : b;
  assign sum = a + condinvb + alucontrol[3];

  always_comb
    case (alucontrol[2:0])
      3'b000: result = a & b;
      3'b001: result = a | b;
      3'b010: result = sum;
      3'b011: result = sum[31];
      3'b110: result = 0 | b[15:0]; //li
      3'b100: result = a << b; //sll
      3'b101:  //zfr
        case(b[4:0])
          5'b00000: result = {a[31:1],1'b0};
          5'b00001: result = {a[31:2],2'b0};
          5'b00010: result = {a[31:3],3'b0};
          5'b00011: result = {a[31:4],4'b0};
          5'b00100: result = {a[31:5],5'b0};
          5'b00101: result = {a[31:6],6'b0};
          5'b00110: result = {a[31:7],7'b0};
          5'b00111: result = {a[31:8],8'b0};
          5'b01000: result = {a[31:9],9'b0};
          5'b01001: result = {a[31:10],10'b0};
          5'b01010: result = {a[31:11],11'b0};
          5'b01011: result = {a[31:12],12'b0};
          5'b01100: result = {a[31:13],13'b0};
          5'b01101: result = {a[31:14],14'b0};
          5'b01110: result = {a[31:15],15'b0};
          5'b10000: result = {a[31:17],17'b0};
          5'b10001: result = {a[31:18],18'b0};
          5'b10010: result = {a[31:19],19'b0};
          5'b10011: result = {a[31:20],20'b0};
          5'b10100: result = {a[31:21],21'b0};
          5'b10101: result = {a[31:22],22'b0};
          5'b10110: result = {a[31:23],23'b0};
          5'b10111: result = {a[31:24],24'b0};
          5'b11000: result = {a[31:25],25'b0};
          5'b11001: result = {a[31:26],26'b0};
          5'b11010: result = {a[31:27],27'b0};
          5'b11011: result = {a[31:28],28'b0};
          5'b11100: result = {a[31:29],29'b0};
          5'b11101: result = {a[31:30],30'b0};
          5'b11110: result = {a[31],31'b0};
          5'b11111: result = a;
        endcase
    endcase

  assign zero = ((alucontrol=='b1010) & (a==b)) | (result==0);
  assign less = (alucontrol=='b1010) & (a < b);
endmodule
```

Line4-7: Creates the module alu and declares the variables inside it

inputs: 32 bit a and b, 4 bit alucontrol

output: zero, less, 32 bit result

Line9: Declares other variables that are neither input nor output of this module

32 bit condinvb and sum

Line11: Set condinvb=~b if (alucontrol[3]==1); otherwise, condinvb=b

Line12: Set sum=a+condinvb+alucontrol[3]

Line14: always_comb for the next lines

Line15-56: Checks the value of alucontrol[2:0] and performs the appropriate action…

Line22-55: If, alucontrol[2:0]==101, check the value of b[4:0] and performs the appropriate action:

```
3'b000: result = a & b;
3'b001: result = a | b;
3'b010: result = sum;
3'b011: result = sum[31];
3'b110: result = 0 | b[15:0]; //li
3'b100: result = a << b; //sll
3'b101:  //zfr
  case(b[4:0])
    5'b00000: result = {a[31:1],1'b0};
    5'b00001: result = {a[31:2],2'b0};
    5'b00010: result = {a[31:3],3'b0};
    5'b00011: result = {a[31:4],4'b0};
    5'b00100: result = {a[31:5],5'b0};
    5'b00101: result = {a[31:6],6'b0};
    5'b00110: result = {a[31:7],7'b0};
    5'b00111: result = {a[31:8],8'b0};
    5'b01000: result = {a[31:9],9'b0};
    5'b01001: result = {a[31:10],10'b0};
    5'b01010: result = {a[31:11],11'b0};
```

```
        5'b01011: result = {a[31:12],12'b0};
        5'b01100: result = {a[31:13],13'b0};
        5'b01101: result = {a[31:14],14'b0};
        5'b01110: result = {a[31:15],15'b0};
        5'b10000: result = {a[31:17],17'b0};
        5'b10001: result = {a[31:18],18'b0};
        5'b10010: result = {a[31:19],19'b0};
        5'b10011: result = {a[31:20],20'b0};
        5'b10100: result = {a[31:21],21'b0};
        5'b10101: result = {a[31:22],22'b0};
        5'b10110: result = {a[31:23],23'b0};
        5'b10111: result = {a[31:24],24'b0};
        5'b11000: result = {a[31:25],25'b0};
        5'b11001: result = {a[31:26],26'b0};
        5'b11010: result = {a[31:27],27'b0};
        5'b11011: result = {a[31:28],28'b0};
        5'b11100: result = {a[31:29],29'b0};
        5'b11101: result = {a[31:30],30'b0};
        5'b11110: result = {a[31],31'b0};
        5'b11111: result = a;
      endcase
```

Line58: Set zero=1 if ((alucontrol==1010) AND (a==b)) OR (result==0); otherwise, zero=0

Line59: Set less=1 if (alucontrol==1010) AND (a<b); otherwise, less=0

Line60: End of module

Overall Changes/Preparations:

alucontrol

There is only one change that was made for almost all of the new instructions. That is making the alucontrol 4 bits instead of the original 3 bits. The reason for this is that there are no enough usable combinations to be used in alu.sv if there are only 3 bits. To keep the operations on the old 3bits, I just inserted a 0 after the first left bit. Note that this is because the leftmost bit will decide the sign of b.

Here is the table that shows the change from 3 to 4 bits and what does alu performs with them:

alucontrol 3bits	alucontrol 4bits	Operation	Purpose of added operation
000	0000	result = a & b	
001	0001	result = a | b	
010	0010	result = sum	
011	0011	result = sum[31]	
100	1000	result = a & b	
101	1001	result = a | b	
110	1010	result = sum	
111	1011		Not used
	0100	result = a << b	sll
	0101	To be explained later	zfr
	0110	result = 0 | b[15:0]	li
	0111		Not used
	1100	result = a << b	Has operation but not used
	1101	To be explained later	Has operation but not used
	1110	result = 0 | b[15:0]	Has operation but not used
	1111		Not used

Here are all the changes regarding this overall change of number of bits of alucontrol:

controller.sv:

```
4  `timescale 1ns / 1ps
5  module controller(input  logic [5:0] op, funct,
6                    input  logic       zero,less,
7                    output logic       memtoreg, memwrite,
8                    output logic       pcsrc, alusrc,
9                    output logic       regdst, regwrite,
10                   output logic       jump,
11                   output logic [3:0] alucontrol);//ble
```

mips.sv:

```
17   logic [3:0] alucontrol;
```

aludec.sv:

```
6  module aludec(input  logic [5:0] funct,
7                input  logic [1:0] aluop,
8                output logic [3:0] alucontrol,
9                input[5:0] op); //li
```

datapath.sv:

```
6  module datapath(input  logic        clk, reset,
7                  input  logic        memtoreg, pcsrc,
8                  input  logic        alusrc, regdst,
9                  input  logic        regwrite, jump,
10                 input  logic [3:0]  alucontrol,
11                 output logic        zero,less,
12                 output logic [31:0] pc,
13                 input  logic [31:0] instr,
14                 output logic [31:0] aluout, writedata,
15                 input  logic [31:0] readdata);//ble
```

alu.sv:

```
4  module alu(input  logic [31:0] a, b,
5             input  logic [3:0]  alucontrol,
6             output logic [31:0] result,
7             output logic        zero,less);//ble
```

controls

This part is more of a preparation in changing the modules. We must first discern what does the added instructions. Here is the table of the controls of the new instructions:

instruction	regwrite	regdst	alusrc	branch	memwrite	memtoreg	jump	aluop
sll	1	1	0	0	0	0	0	10
sb	0	0	1	0	1	0	0	00
ble	0	0	0	1	0	0	0	01
li	1	0	1	0	0	0	0	00
zfr	1	1	0	0	0	0	0	10

testbench.sv/ memfile.mem

This is not really a preparation or change in the actual modules but this is an important part of checking whether each instructions are implemented correctly. For the testbench, I just used the given testbench from Lab12. All I did was to remove the condition at the end so that it can run any instructions without trouble. Here is the testbench:

```
// CS 21 Lab2 -- S2 AY 2021-2022
// Carl David B. Ragunton -- 6/5/2022
// testbench.sv -- a testbench for the single cycle processor in
Project

`timescale 1ns / 1ps
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] writedata, dataadr;
  logic        memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      $dumpfile("dump.vcd"); $dumpvars;
      reset <= 1; # 22; reset <= 0;

    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end


endmodule
```

Line5: Sets the timescale of the simulation to be 1ns per 1ps

Line6: Creates the module testbench

Line8-12: Declares variables that will be used

Line 15: Instantiates top to create dut

Line18: Start of the initial block

Line19: begin

Line20: Something to do with showing the EPWaves

Line21: Changing the value of reset

Line23: end

Line26-29: Changes the clk from 0 to 1 or from 1 to 0 every #5

Line33: End of testbench

For the memfile.mem, I would not place it here because I will change it for every different instruction. The values I will use for this will later be placed in tables under Machine language column.

Added Operations Explanation

From this point onwards, each added instruction will be discussed one by one. All of the changes for them will also be explained. The testcases for them will also be shown here. I will not explain the in a systemized manner. I will explain them based on how I worked on creating each one of them.

li instruction

I will first explain the li instruction. The reason for this is that this will make the testcases for other instructions easier.

This instruction was implemented as an I-type instruction. Also, this is quite similar to addi. Thus I followed where and what addi does in the modules. I based the changes for li from that.

First, we need to put the controls in maindec.sv through its op:

```
18      case(op)
19        6'b000000: controls <= 9'b110000010; // RTYPE
20        6'b100011: controls <= 9'b101001000; // LW
21        6'b101011: controls <= 9'b001010000; // SW
22        6'b101000: controls <= 9'b001010000; // SB
23        6'b000100: controls <= 9'b000100001; // BEQ
24        6'b011111: controls <= 9'b000100001; // BLE
25        6'b001000: controls <= 9'b101000000; // ADDI
26        6'b010001: controls <= 9'b101000000; // LI
27        6'b000010: controls <= 9'b000000100; // J
28        default:   controls <= 9'bxxxxxxxxx; // illegal op
29      endcase
```

We also need to put the alucontrol for it in aludec:

```
12      case(aluop)
13        2'b00:
14          case(op)
15            6'b010001: alucontrol <= 4'b0110; //li
16            default: alucontrol <= 4'b0010;  // add (for
    lw/sw/addi/sb)
17          endcase
```

But note that aluop==00 will make it just like the addi since they also have the similar controls. One way we can avoid this is to check the instruction itself specifically the op. But op is originally not in aludec. So, we force it to go to aludec.sv by doing these changes on other modules:

controller.sv:

```
18    aludec  ad(funct, aluop, alucontrol,op);  //li
```

aludec.sv:

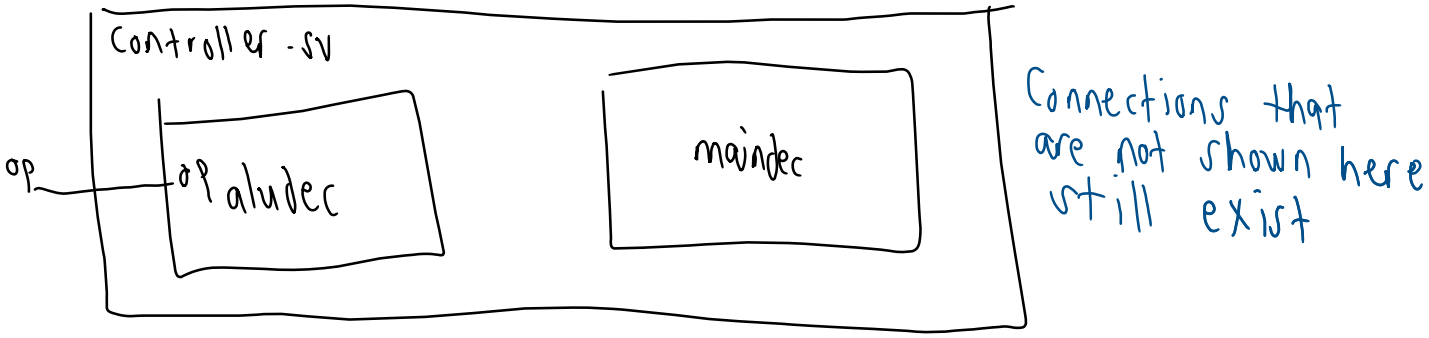
```
6  module aludec(input  logic [5:0] funct,
7                input  logic [1:0] aluop,
8                output logic [3:0] alucontrol,
9                input[5:0] op); //li
```

Now that op has reached aludec.sv, we can now make the alucontrol right.

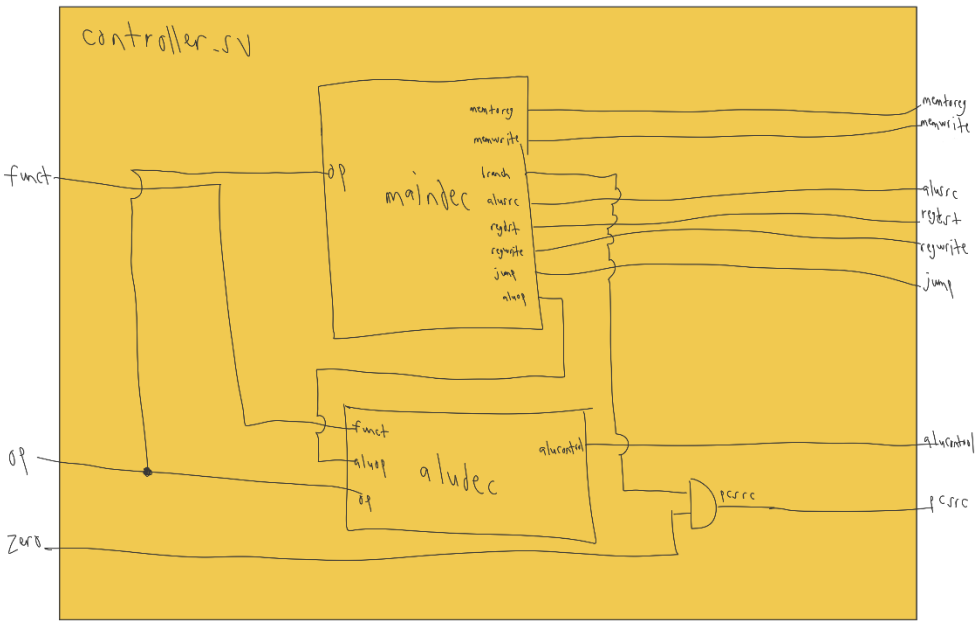
The next place where addi would do its process is the alu.sv. Thus, the same will apply for li. We check its alucontrol and make a special operation for it.

```
15      case (alucontrol[2:0])
16        3'b000: result = a & b;
17        3'b001: result = a | b;
18        3'b010: result = sum;
19        3'b011: result = sum[31];
20        3'b110: result = 0 | b[15:0]; //li
21        3'b100: result = a << b; //sll
22        3'b101:  //zfr
```

The operation I made is 0 | b[15:0]. Note that in li, we load the immediate value into the register no matter what value is currently in the register. That is why we OR it with 0. So that the value will always be the immediate value which is b in here. But why b[15:0] instead of just b? Note that the b here is sign extended. We were asked to make it logically or zero extended. Therefore, we just remove the first 16 bits to disregard the sign extension. This will also automatically zero extend the immediate value into 32 bits. These are all the changes made for the li instruction. This is the only change in the schematic diagram:



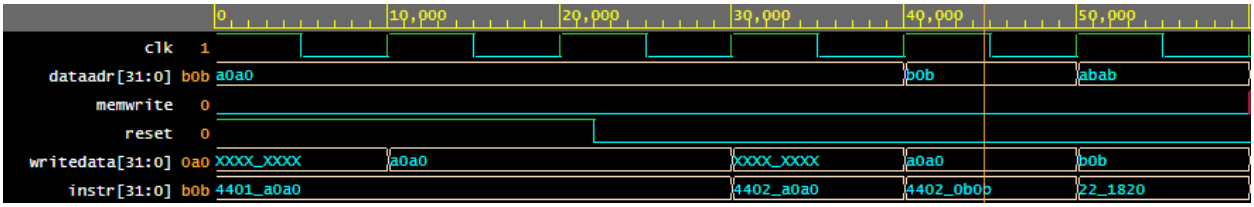
Here is the whole schematic diagram for controller.sv at this point:



Testing li:

instruction	result	Machine language
li $1, 0xA0A0	$1=0xA0A0	4401a0a0
li $2, 0xA0A0	$2=0xA0A0	4402a0a0
li $2, 0xB0B	$2=0xB0B	44020b0b
add $3, $1, $2	$3=ABAB	00221820

Here is the result:

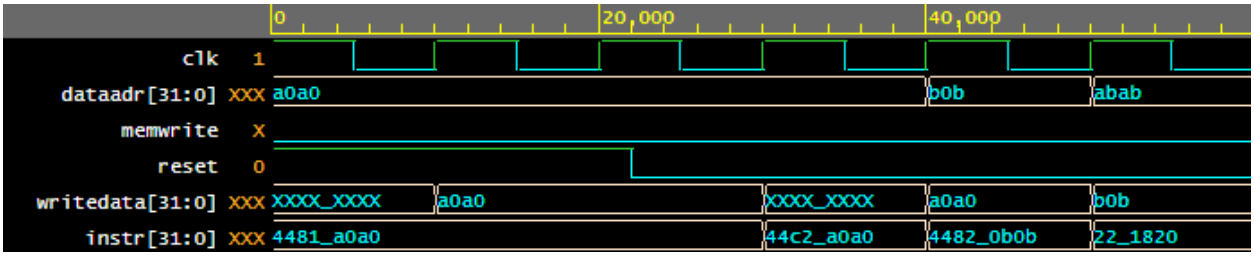


We load 0xa0a0 into register 1. Load 0xa0a0 into register 2. The value of $2 was overwritten when another li instruction was made on it. It is now 0xb0b as shown in the 2nd to the last clock cycle. Note that we are able to add $1 and $2 into $3. Also, This means that their values was changed properly using li. Thus, this instruction is implemented properly.

Testing li X

instruction	result	Machine language
li $1, 0xA0A0	$1=0xA0A0	4481a0a0
li $2, 0xA0A0	$2=0xA0A0	44c2a0a0
li $2, 0xB0B	$2=0xB0B	44820b0b
add $3, $1, $2	$3=ABAB	00221820

Result:



This is the same even with different values for the don’t cares.

sll instruction

There are many things to consider in sll instruction. Since this is an R-type instruction, there is no need to change its controls. The next thing we need to consider is its alucontrol in aludec:

```
19        default: case(funct)          // R-type instructions
20            6'b000000: alucontrol <= 4'b0100; // sll
21            6'b100000: alucontrol <= 4'b0010; // add
22            6'b100010: alucontrol <= 4'b1010; // sub
23            6'b100100: alucontrol <= 4'b0000; // and
24            6'b100101: alucontrol <= 4'b0001; // or
25            6'b101010: alucontrol <= 4'b1011; // slt
26            6'b110011: alucontrol <= 4'b0101; // zfr
27            default:   alucontrol <= 4'bxxxx; // ???
28          endcase
```

Note that shamt is the number of times we need to shift left logical instr[20:16]. But on the given modules, there is no way that shamt reaches the alu.sv. Also, sll only concerns itself with instr[20:16]. I came up with the idea of making the variable a the register value of instr[20:16] instead of instr[25:21]. This can be done with the following in the datapath.sv:

```
23    logic [4:0] instr2521;//sll
```

First we declare this 5 bit variable. This would be the value of the originally instr[25:21] of the regfile.

```
26    assign alusrcb = (alucontrol[3:0]==4'b0100) ? 1 : 0;//sll
```

This will serve as a control bit for a mux. Note that this will only become 1 if the instruction is sll.

```
39    mux2 #(32)
instr2521sll(instr[25:21],instr[20:16],alusrcb,instr2521);//sll
```

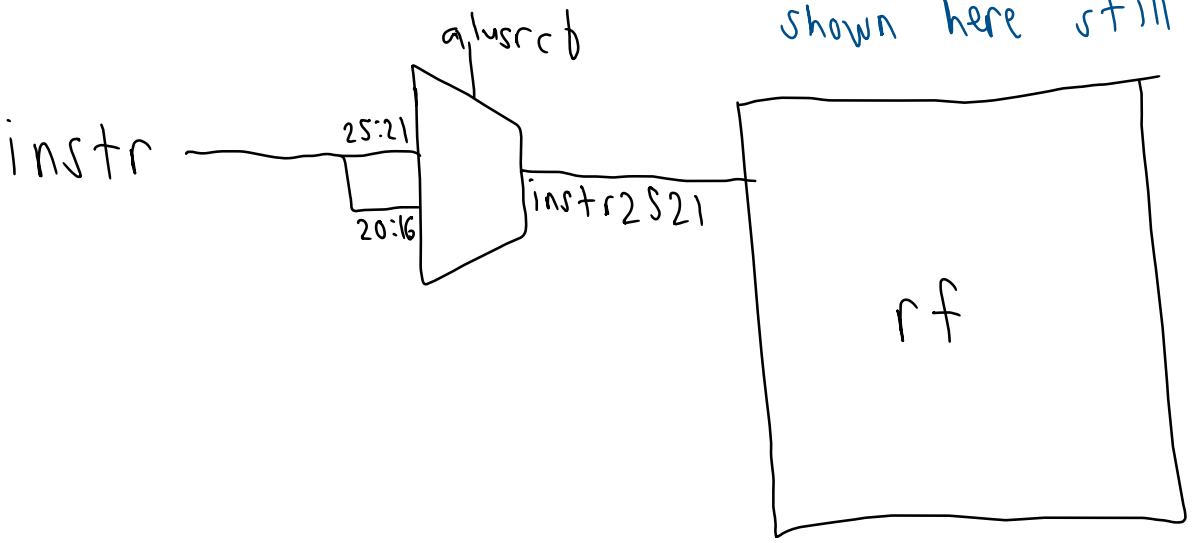
We, then instantiate mux2 to choose a value between instr[25:21] and instr[20:16]. Note that unless the instruction is sll, it will always choose instr[25:21]. Assume that it is sll for now. We assign the chosen value to instr2521.

```
40    regfile     rf(clk, regwrite, instr2521, instr[20:16],
41                   writereg, result, srca, writedata);//sll
```

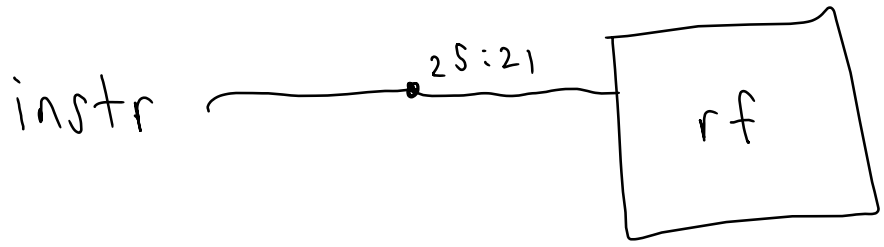
Originally, the instr2521 here is just instr[25:21]. But now, in case the instruction is sll, we can force it to be instr[20:16]. Thus, the a in the alu.sv later will become the register value of instr[20:16] during sll instructions.

The changes till now will lead to this change in the schematic diagram:

connections that are not shown here still exist



instead of



We have successfully changed the value of a in alu.sv. But we still need to deal with b. Note that b comes from srcb. How about we create a srcB after that? A mux that chooses between shamt and srcb.

```
20    logic [31:0] srca, srcb,srcB,shamt; //sll
```

We just declare the needed variables. The new variables here are srcB and shamt.

```
26    assign alusrcb = (alucontrol[3:0]==4'b0100) ? 1 : 0;//sll
27    assign shamt = {27'b0,instr[10:6]};//sll
```

alusrcb will again be used later. We get the value of shamt from instr[10:6] and just zero extend it to 32 bits.

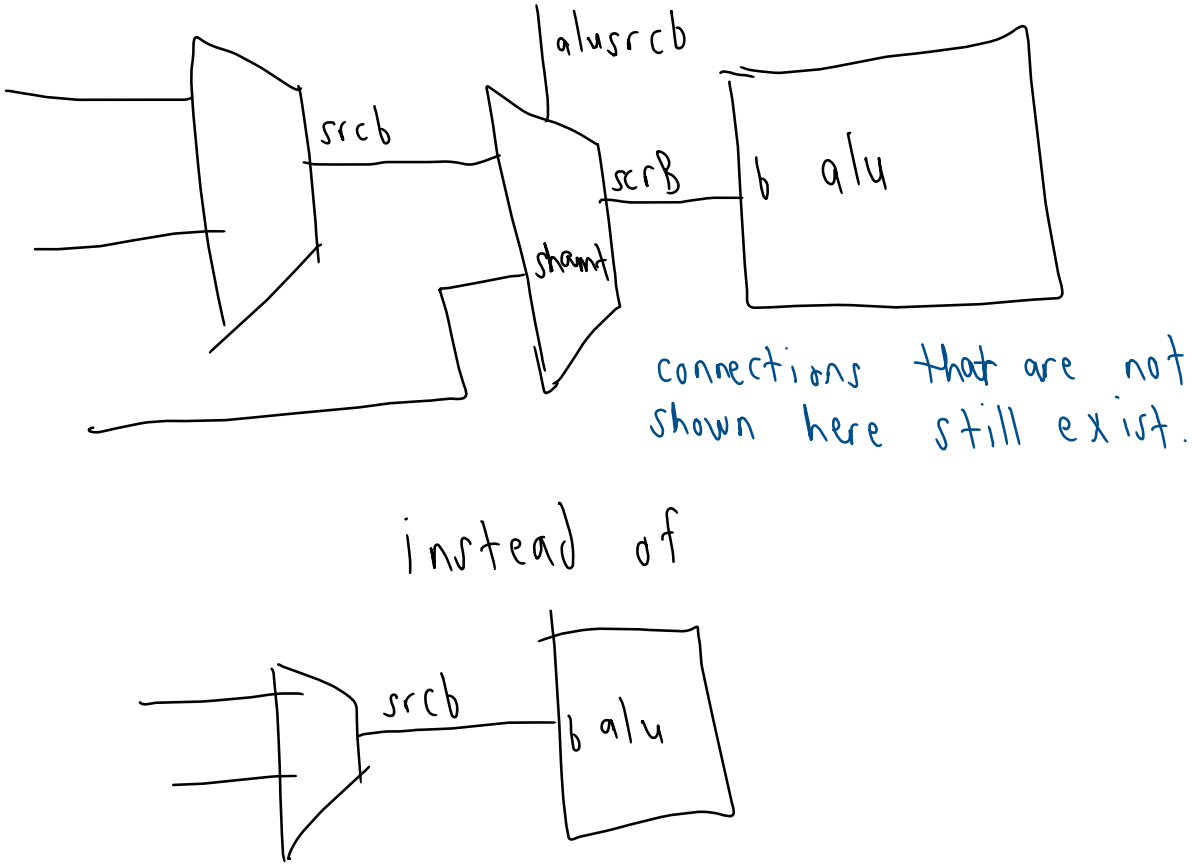
```
49    mux2 #(32)  srcBmux(srcb,shamt,alusrcb,srcB);//sll
```

As mentioned earlier, we create a mux that chooses between srcb and shamt. Note that this will only pick shamt if the instruction is sll. The result is placed in srcB.

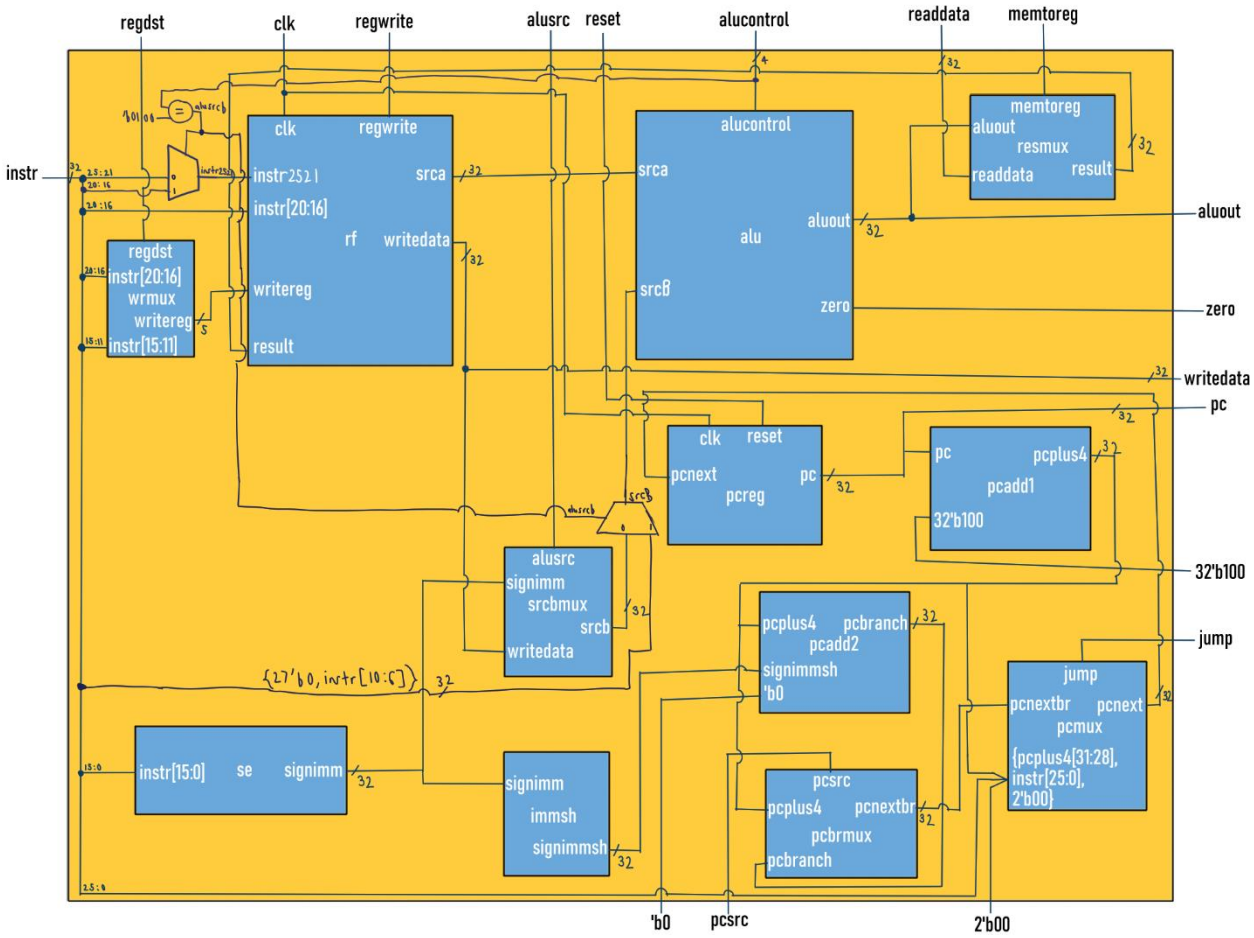
```
50    alu         alu(srca, srcB, alucontrol, aluout,
zero,less);//sll,ble
```

Instead of just srcb, we change it to srcB to make the b in alu.sv the value of shamt.

These changes will make the ff schematic diagram:



Here is the whole datapath.sv diagram:



There is one last thing to change; that is the alu.sv.

```
15      case (alucontrol[2:0])
16        3'b000: result = a & b;
17        3'b001: result = a | b;
18        3'b010: result = sum;
19        3'b011: result = sum[31];
20        3'b110: result = 0 | b[15:0]; //li
21        3'b100: result = a << b; //sll
```

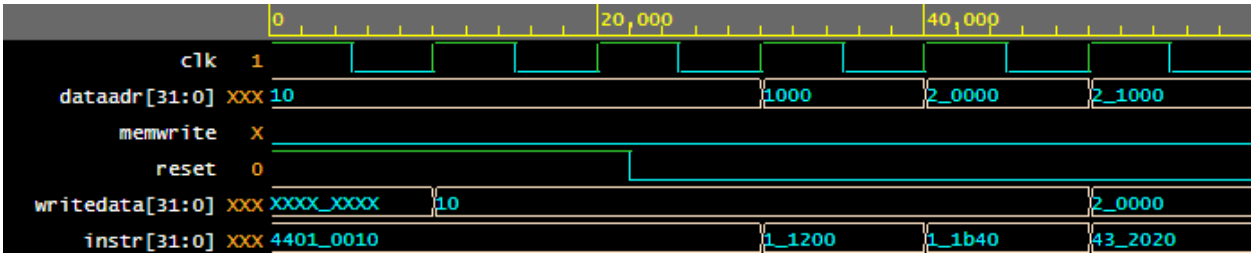
Here, we just simply do a<<b when the instruction is sll.

These are all the changes made for sll.

Testing sll:

instruction	result	Machine language
li $1, 16	$1=0x10	44010010
sll $2, $1, 8	$2=0x1000	00011200
sll $3, $1, 13	$3=0x20000	00011b40
add $4, $2, $3	$4=0x21000	00432020

Here are the results:

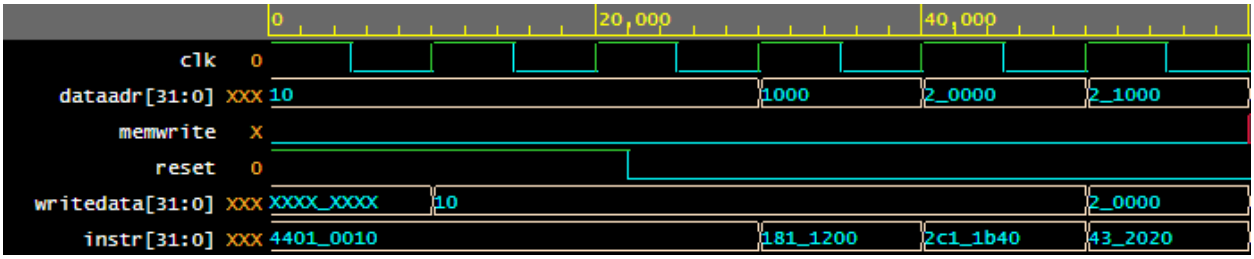


Note that as expected, the values from the table appeared here. We load 0x10 into register1. Then, sll $1 by 8 and put it in $2. This is shown in the 4th clock cycle in the EPWaves. sll $1 by 13 and put it in $3 which means putting 0x2000 in it. This is the 5th clock cycle. In the end, 0x21000 is stored into $4 because the values before it was correct Thus, this sll implementation must be correct.

Testing sll X:

instruction	result	Machine language
li $1, 16	$1=0x10	44010010
sll $2, $1, 8	$2=0x1000	01811200
sll $3, $1, 13	$3=0x20000	02c11b40
add $4, $2, $3	$4=0x21000	00432020

Results:



Same results even with different don't care values.

ble instruction

In making li earlier, we checked for the path of addi. Here, we check for the path of beq and edit it along the way. First thing to do is to arrange the controls for ble in maindec:

```
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b101000: controls <= 9'b001010000; // SB
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b011111: controls <= 9'b000100001; // BLE
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b010001: controls <= 9'b101000000; // LI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
```

The op of 011111 was mentioned in the pdf.

Then we start following beq to check the condition wherein it does its job. After following it, I noticed that it was the variable zero at sets up the pcsrc. Therefore, for each time the zero variable would appear, we would just put a less variable. Here are all the modules where I added less.

mips.sv:

```
  logic       memtoreg, alusrc, regdst,
              regwrite, jump, pcsrc, zero,less;//ble
  controller c(instr[31:26], instr[5:0], zero,less,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);//ble
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero,less, pc, instr,
              aluout, writedata, readdata);//ble
```

controller.sv

```
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,less,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [3:0] alucontrol);//ble
```

datapath.sv

```
6  module datapath(input  logic        clk, reset,
7                  input  logic        memtoreg, pcsrc,
8                  input  logic        alusrc, regdst,
9                  input  logic        regwrite, jump,
10                 input  logic [3:0]  alucontrol,
11                 output logic        zero,less,
12                 output logic [31:0] pc,
13                 input  logic [31:0] instr,
14                 output logic [31:0] aluout, writedata,
15                 input  logic [31:0] readdata);//ble
```

```
50   alu         alu(srca, srcB, alucontrol, aluout,
zero,less);//sll,ble
```

alu.sv:

```
4  module alu(input  logic [31:0] a, b,
5             input  logic [3:0]  alucontrol,
6             output logic [31:0] result,
7             output logic        zero,less);//ble
```

Aside from making the variable less and building its connections, we still need to assign its value along with some changes to zero. This is done in alu.sv.

```
58   assign zero = ((alucontrol=='b1010) & (a==b)) | (result==0);
59   assign less = (alucontrol=='b1010) & (a < b);
```

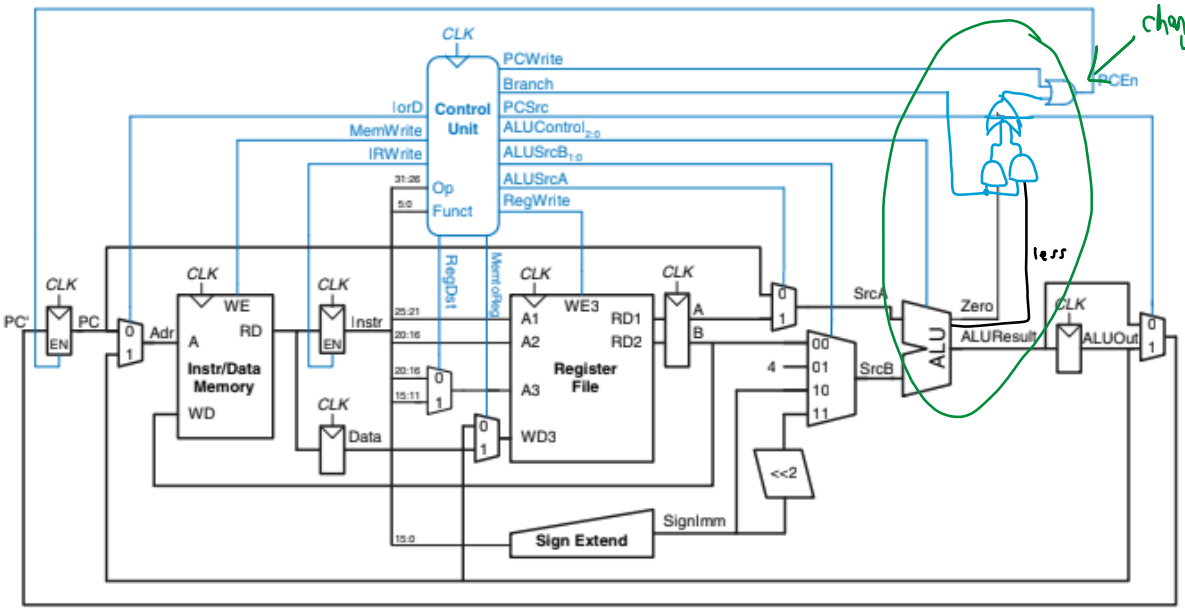
zero becomes 1 if the result==0 OR the (current instruction is ble AND a==b). Meanwhile, less becomes 1 if current instruction is ble AND a<b. Otherwise, their values are 0.

```
21     case(op)
22       6'b011111: pcsrc = (branch & zero) | (branch & less);
23       default:pcsrc = branch & zero;
24     endcase
```

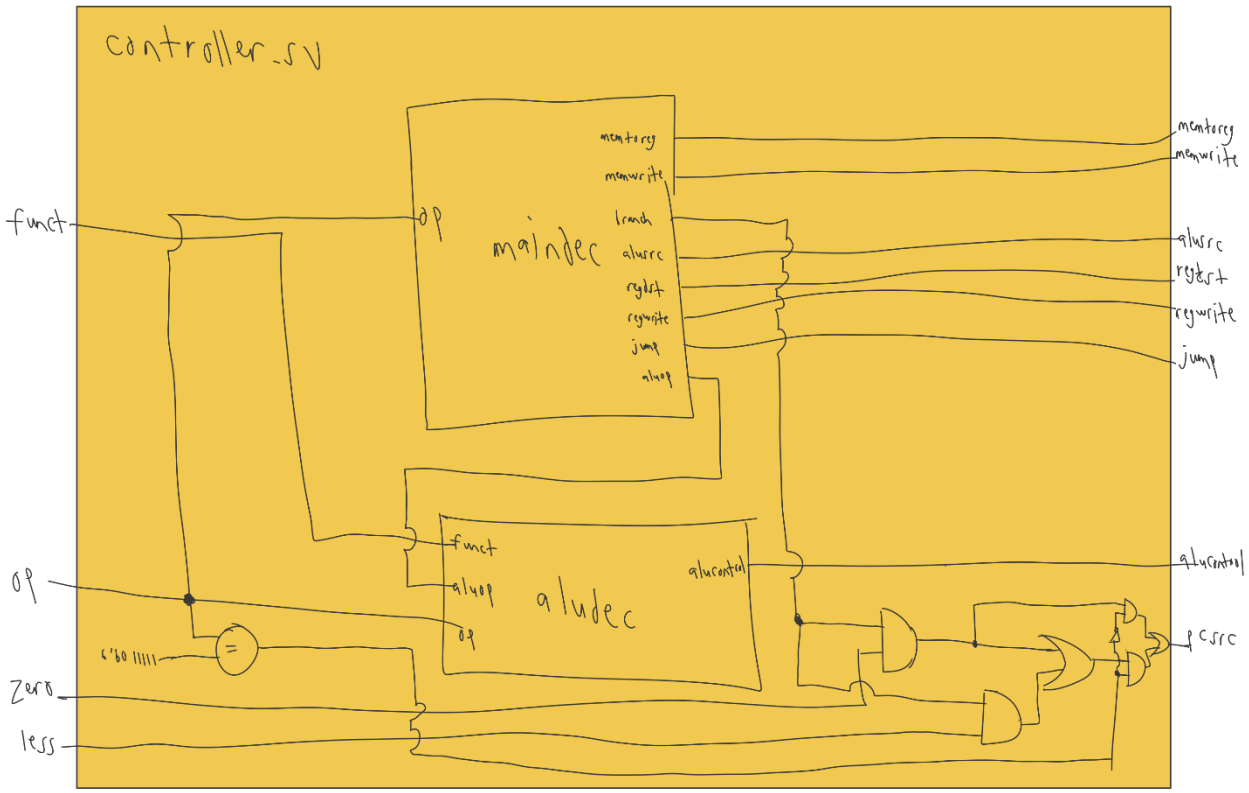
This change was done in controller.sv and this is where the values of zero and less are used. If the current instruction is ble, it would check whether (branch=zero=1) OR (branch=less=0). If that is true, pcsrc=1. If it is not ble, it would go into default or the way it does things before I edited this part.

These are all the changes I made into the modules for ble. There are no significant changes in the schematic diagram of the modules except for the added less each time zero appears.

This is the schematic diagram for a complete scp. It is not necessarily the same as this verilog files. This is just similar to it. This is the part that changed.



Note that the diagram above is just for ble. Here is the correct schematic diagram of controller.sv:

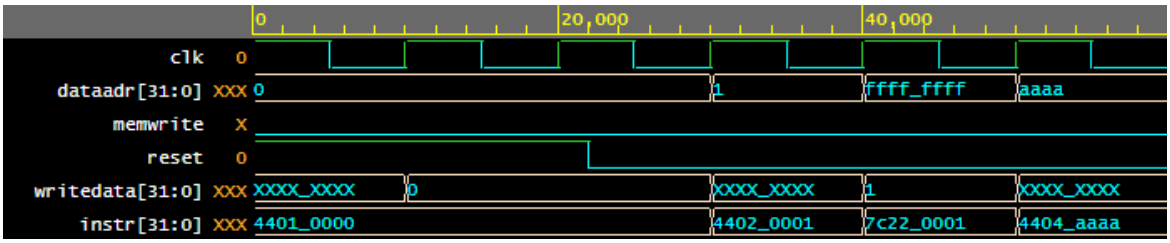


Testing ble:

a<b: branch taken

instruction	result	Machine language
li $1, 0	$1=0	44010000
li $2, 1	$2=1	44020001
ble $1, $2, 1	branch to li $4,0xAAAA	7c220001
li $3, 0xFFFF	not used	4403ffff
li $4, 0xAAAA	$4=0xAAAA	4404aaaa

Result:

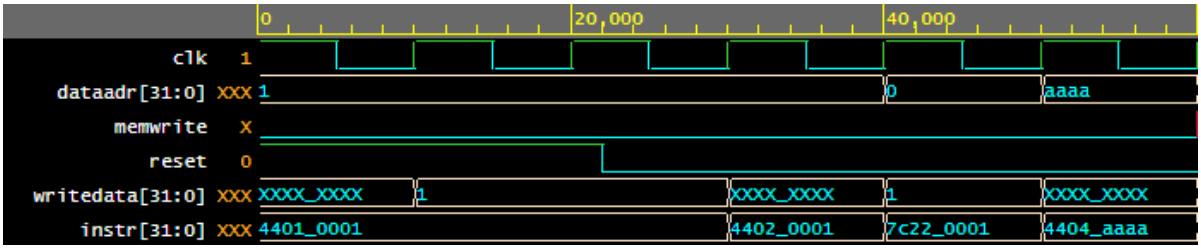


Note that after the instruction 0x7c220001, 0x4404aaaa is the next one. Therefore, it must have branched successfully. If this did not branched, the instruction after it should have been 0x4403ffff.

a==b: branch taken

instruction	result	Machine language
li $1, 1	$1=1	44010001
li $2, 1	$2=1	44020001
ble $1, $2, 1	branch to li $4,0xAAAA	7c220001
li $3, 0xFFFF	not used	4403ffff
li $4, 0xAAAA	$4=0xAAAA	4404aaaa

Result:



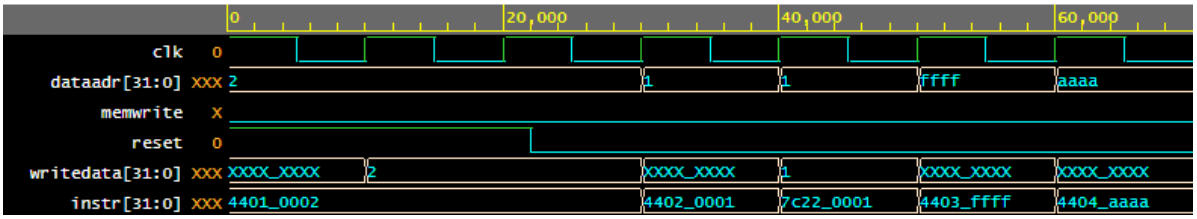
Once again, notice that the instruction 0x4403ffff is not executed. Therefore, it must have branched properly.

a<b: branch not taken

instruction	result	Machine language
li $1, 2	$1=2	44010002
li $2, 1	$2=1	44020001
ble $1, $2, 1	branch not taken	7c220001
li $3, 0xFFFF	$3=0xFFFF	4403ffff

li $4, 0xAAAA	$4=0xAAAA	4404aaaa

Result:



Notice that this time, the instruction 0x4403ffff was executed because the branch was not taken. We have shown that it works when a<=b. Thus, this ble instruction was made correctly.

There are no don’t cares for ble.

sb instruction

We first set the controls for sb in maindec.sv:

```
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b101000: controls <= 9'b001010000; // SB
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b011111: controls <= 9'b000100001; // BLE
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b010001: controls <= 9'b101000000; // LI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
```

The controls are as shown in the table before. It is similar to sw. Notice that sb is really just similar to sw. It just becomes different the moment the process of storing into memory happens. Thus, we just follow the sw instruction. After following it, I notice that it stores in the memory at the dmem.sv. All we need to do is to find a way to differentiate sb from sw. I did this by putting opcode into the dmem.sv. Here are all the changes done to bring opcode into dmem.sv:

mips.sv:

```
module mips(input  logic        clk, reset,
            output logic [31:0] pc,
            input  logic [31:0] instr,
            output logic        memwrite,
            output logic [31:0] aluout, writedata,
            input  logic [31:0] readdata,
            output logic [5:0] opcode);//sb
```

```
  assign opcode = instr[31:26];//sb
```

We just assign opcode as instr[31:26].

top.sv:

```
module top(input  logic        clk, reset,
           output logic [31:0] writedata, dataadr,
           output logic        memwrite,
           input logic [5:0] opcode);//sb
```

```
  dmem dmem(clk, memwrite, dataadr, writedata, readdata,
opcode);//sb
```

dmem.sv:

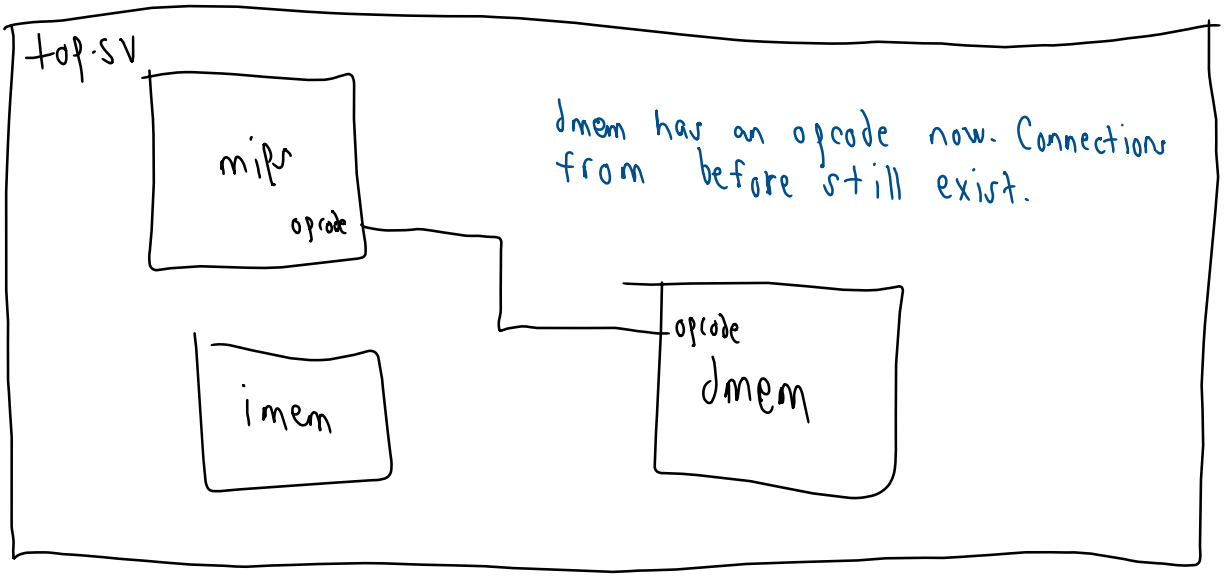
```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd,
            input logic [5:0] opcode);
```

With those changes, opcode is now on dmem.sv.

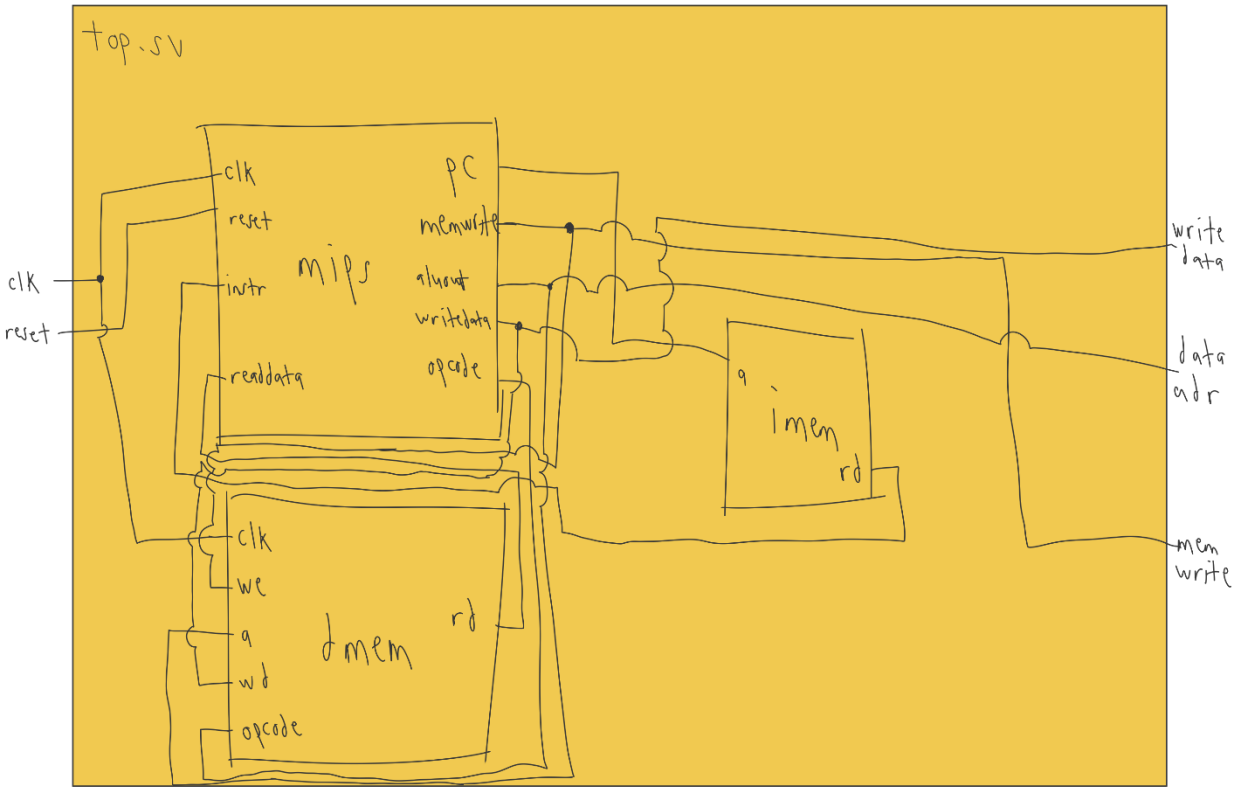
```
18              case(a%4)
19                'b0: RAM[a[31:2]][31:24] <= wd[7:0];
20                'b1: RAM[a[31:2]][23:16] <= wd[7:0];
21                'b10: RAM[a[31:2]][15:8] <= wd[7:0];
22                'b11: RAM[a[31:2]][7:0] <= wd[7:0];
23              endcase
```

Here, if the instruction is not sb, it would reach default which was the original way of dealing with sw before I edited this part. But if opcode==101000 or the current instruction is sb, it would check if we==1. If it is, it would then check the value of a%4. Note that this is in Big Endian. If it is divisible by 4, it would change the leftmost byte of the memory address. If it is equal to 1, it would change the 2nd byte from the left. If it is 2, it would change the 2nd bit from the right. Lastly if it is equal to 3, it would change the rightmost byte of the current memory address.

These are all the changes done to add sb instruction. There are no significant changes in the schematic diagram except for the process of bringing opcode into dmem.sv.



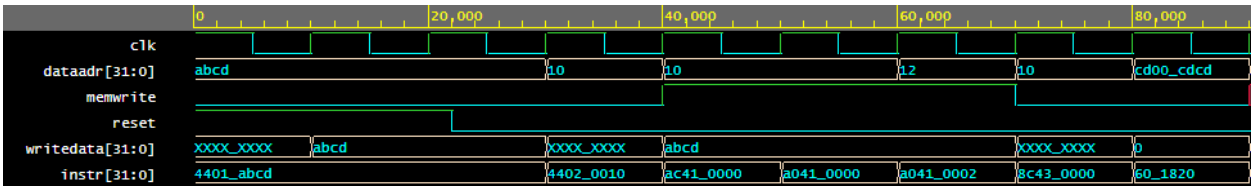
Here is the whole top.sv diagram at this point:



Testing sb:

instruction	result	Machine language
li $1, 0xABCD	$1=0xABCD	4401abcd
li $2, 0x10	$2=0x10	44020010
sw $1, 0($2)	0x16=0xABCD	ac410000
sb $1, 0($2)	0x16=0xCD00ABCD	a0410000
sb $1, 2($2)	0x18=0xCD	a0410002
lw $3, 0($2)	$3=0xCD00CDCD	8c430000
add $3, $3, $0	$3=0xCD00CDCD	00601820

Result:



Note that the results here does not show much of what happens during the store and load instructions. That is why I put an add instruction at the end; to show the final value of 3. Note that $3=0xcd00cdcd just as we expected to happen. Thus, the sb instruction works properly.

There are no don't cares for sb.

zfr instruction

I will no longer explain how this instruction works since that was explained really well in the pdf given. This is an R-type instruction so, there is no need to change the controls in maindec.sv. What we need to add is the alucontrol in aludec.sv.

```
19        default: case(funct)          // R-type instructions
20            6'b000000: alucontrol <= 4'b0100; // sll
21            6'b100000: alucontrol <= 4'b0010; // add
22            6'b100010: alucontrol <= 4'b1010; // sub
23            6'b100100: alucontrol <= 4'b0000; // and
24            6'b100101: alucontrol <= 4'b0001; // or
25            6'b101010: alucontrol <= 4'b1011; // slt
26            6'b110011: alucontrol <= 4'b0101; // zfr
27            default:   alucontrol <= 4'bxxxx; // ???
28          endcase
```

Just like what we did for some instructions before, we just follow similar instructions to this one to see where and what we need to add. We would notice that the next ting we have to edit is the alu.sv. Notice that unlike the sll instruction from before which is also an R-type instruction, the a and b in the alu.sv are already correct for the zfr instruction.

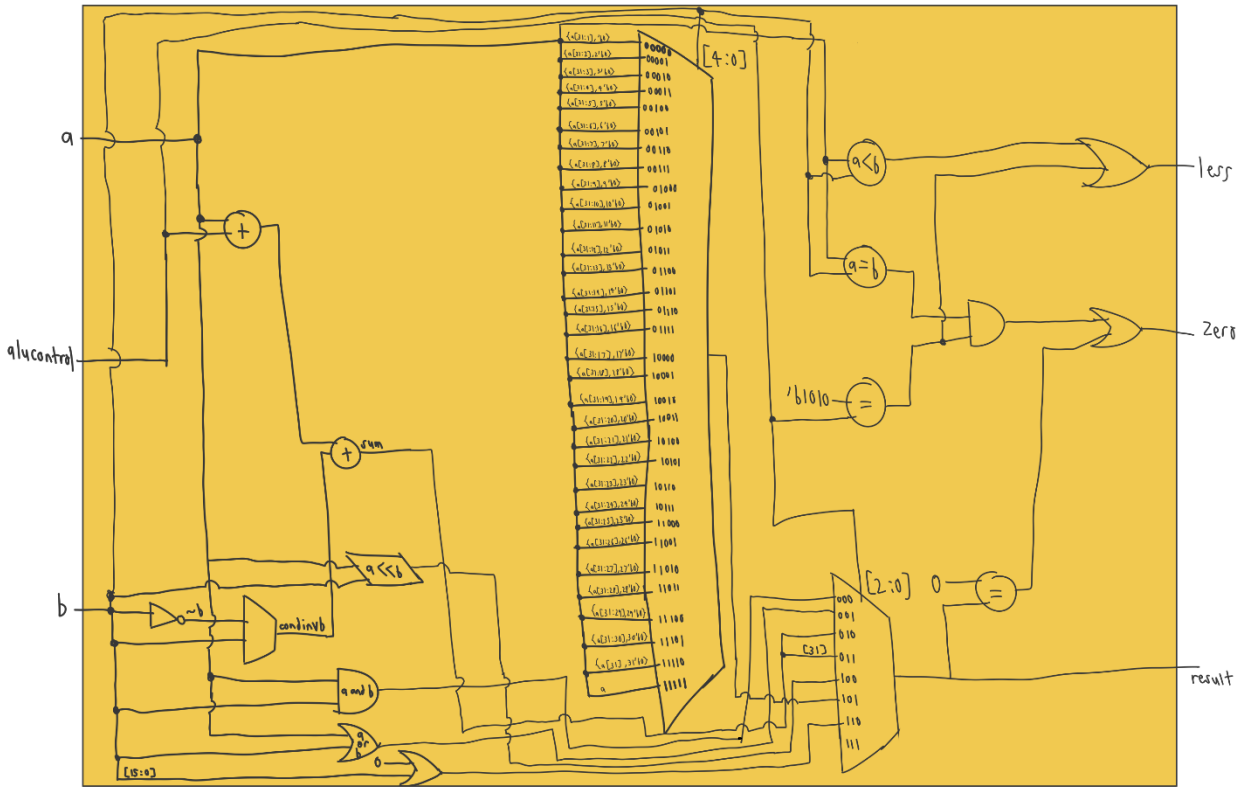
Since the change in a depends on the last 5 bits of the b value, I decided to do this manually. 5 bits because 2^5=32 which is the number of bits of a.

```
3'b101:  //zfr
  case(b[4:0])
    5'b00000: result = {a[31:1],1'b0};
    5'b00001: result = {a[31:2],2'b0};
    5'b00010: result = {a[31:3],3'b0};
    5'b00011: result = {a[31:4],4'b0};
    5'b00100: result = {a[31:5],5'b0};
    5'b00101: result = {a[31:6],6'b0};
    5'b00110: result = {a[31:7],7'b0};
    5'b00111: result = {a[31:8],8'b0};
    5'b01000: result = {a[31:9],9'b0};
    5'b01001: result = {a[31:10],10'b0};
    5'b01010: result = {a[31:11],11'b0};
    5'b01011: result = {a[31:12],12'b0};
    5'b01100: result = {a[31:13],13'b0};
    5'b01101: result = {a[31:14],14'b0};
    5'b01110: result = {a[31:15],15'b0};
    5'b10000: result = {a[31:17],17'b0};
    5'b10001: result = {a[31:18],18'b0};
    5'b10010: result = {a[31:19],19'b0};
    5'b10011: result = {a[31:20],20'b0};
    5'b10100: result = {a[31:21],21'b0};
    5'b10101: result = {a[31:22],22'b0};
    5'b10110: result = {a[31:23],23'b0};
    5'b10111: result = {a[31:24],24'b0};
    5'b11000: result = {a[31:25],25'b0};
    5'b11001: result = {a[31:26],26'b0};
    5'b11010: result = {a[31:27],27'b0};
    5'b11011: result = {a[31:28],28'b0};
    5'b11100: result = {a[31:29],29'b0};
    5'b11101: result = {a[31:30],30'b0};
    5'b11110: result = {a[31],31'b0};
    5'b11111: result = a;
  endcase
```

If the current instruction is zfr or alucontrol[2:0]==101, it would check the last 5 bits of b. Depending on its value, a would be zeroed starting from its right. If b[4:0]==00000, a bit of a would become 0. If it is equal to 00001, 2 rightmost bits of a would be zeroed. If it is 11111, nothing would change. I would not explain them one by one since they are pretty much self-explanatory and it would become redundant.

These are all the changes done in order to add zfr instruction. When the whole schematic diagram is to be considered, there are almost no changes into it. The biggest change happens within the alu.sv.

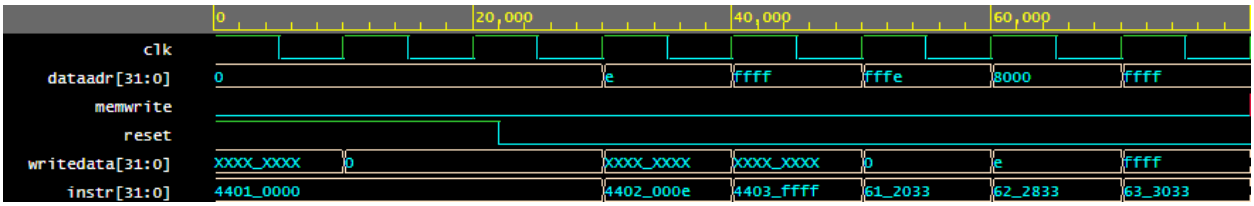
Here is the schematic diagram at gate level showing the changes in alu.sv. (sorry if the wires are hard to understand)



Testing zfr:

instruction	result	Machine language
li $1, 0	$1=0	44010000
li $2, 0xE	$2=0xE	4402000e
li $3, 0xFFFF	$2=0xFFFF	4403ffff
zfr $4, $3, $1	$4=0xFFFE	00612033
zfr $5, $3, $2	$5=0x8000	00622833
zfr $6, $3, $3	$6=0xFFFF	00633033

Result:



We just need to look at the last 3 instructions because they are the zfr instruction and see if they match our answers. Note that at the first zfr instruction, $4=0xfffe. It makes sense since 0xffff is zfr'd by 0x0; it's right most bit would be zeroed. In the second zfr, $5=0x8000. This is 0xffff zfr'd by 0xE; doing this would just leave the leftmost 1 of a intact, so this is correct. For the last zfr instruction, $6=0xffff. This is

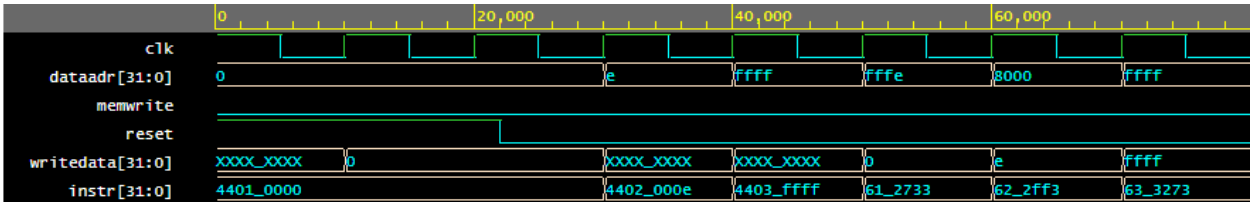
0xffff zfr'd by itself. This means that b[4:0]=11111 and as we mentioned before, this would cause no changes in a.

All the results were correct. Therefore, this zfr instruction was implemented correctly.

Testing zfr X:

instruction	result	Machine language
li $1, 0	$1=0	44010000
li $2, 0xE	$2=0xE	4402000e
li $3, 0xFFFF	$2=0xFFFF	4403ffff
zfr $4, $3, $1	$4=0xFFFE	00612733
zfr $5, $3, $2	$5=0x8000	00622ff3
zfr $6, $3, $3	$6=0xFFFF	00633273

Result:

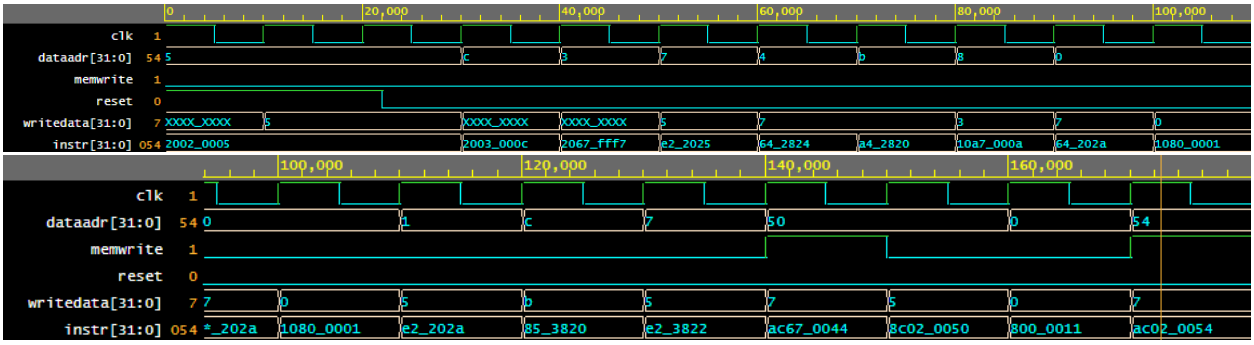


Same results even with different don't care values.

Checking the Old Instructions

I have added all the new instructions, but we still need to check whether the old instructions still work properly. For this, I will use the instructions given at lab12. Note that this contains all the old instructions.

```
#          Assembly              Description              Address    Machine
main:      addi $2, $0, 5        # initialize $2 = 5       0          20020005
           addi $3, $0, 12       # initialize $3 = 12      4          2003000c
           addi $7, $3, -9       # initialize $7 = 3       8          2067fff7
           or   $4, $7, $2       # $4 = (3 OR 5) = 7       c          00e22025
           and  $5, $3, $4       # $5 = (12 AND 7) = 4     10         00642824
           add  $5, $5, $4       # $5 = 4 + 7 = 11         14         00a42820
           beq  $5, $7, end      # shouldn't be taken      18         10a7000a
           slt  $4, $3, $4       # $4 = 12 < 7 = 0         1c         0064202a
           beq  $4, $0, around   # should be taken         20         10800001
           addi $5, $0, 0        # shouldn't happen        24         20050000
around:    slt  $4, $7, $2       # $4 = 3 < 5 = 1          28         00e2202a
           add  $7, $4, $5       # $7 = 1 + 11 = 12        2c         00853820
           sub  $7, $7, $2       # $7 = 12 - 5 = 7         30         00e23822
           sw   $7, 68($3)       # [80] = 7                34         ac670044
           lw   $2, 80($0)       # $2 = [80] = 7           38         8c020050
           j    end              # should be taken         3c         08000011
           addi $2, $0, 1        # shouldn't happen        40         20020001
end:       sw   $2, 84($0)       # write mem[84] = 7       44         ac020054
```



Instead of looking at the registers per instructions, we can check this by just looking at the instr and dataadr. Note that dataadr contains the value that would be written in the register if that is the case. For other instructions, it should contain the address where a data would be written. Let's see if it matches the results above. For beq and jump instruction, we would base if it works on the next instruction that comes after it.

20020005-5

2003000c-c

2067fff7-3

00e22025-7

00642824-4

00a42820-b

10a7000a- not taken because next instruction stays the same

0064202a-0

10800001- taken because the next instruction changed

20050000-did not happen

00e2202a-1

00853820-c

00e23822-7

ac670044-80

8c020050-7

08000011-taken because next instruction changed

20020001-did not happen

ac020054-84

Note that the dataadr and instr in the EPWaves completely matches with this. Thus, the old instructions must be working just fine.