Felix Q. Bueno IV
2020-05609

CS 21 Project 2 HDL Documentation

Google drive link for Video Documentation:

https://drive.google.com/file/d/1FjMWmXV8JJGlIlkZoYq6q8Nq7wl0tuWz/view?usp=sharing

General Changes HDL Code

We must explain the general changes made to some of the modules in the given HDL codes because these changes helped in implementing or adding some of the new instructions. First, we changed the bit width of the *alucontrol* from 3 bits to 4 bits. This is to make way for new bit combinations that we would use as a reference for a new operation to be performed and result given by the *alu* module. Note that the new bit for our *alucontrol* was actually added between the 2nd and 3rd bit of the original *alucontrol*. This was done to make the addition of new instructions easier such that it would not affect the previous instructions working already. Naturally, for all the instructions that were already included in the given code, the new bit added was set to 0. This can be seen in the screenshot of the *aludec.sv* below: (ignore the added instructions for now, they will be explained more later) (Note also that the width of output port for *alucontrol* in *aludec* module was changed to 4 bits).

```
1  ////////////
2  `timescale 1ns / 1ps
3  module aludec(input  logic [5:0] funct,
4                input  logic [1:0] aluop,
5                output logic [3:0] alucontrol);
6
7    always_comb
8      case(aluop)
9        2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/sb/addi)
10       2'b01: alucontrol <= 4'b1010;  // sub (for beq)
11       2'b11: alucontrol <= 4'b1101;  // sub (for ble)
12       default: case(funct)          // R-type instructions
13           6'b100000: alucontrol <= 4'b0010; // add
14           6'b100010: alucontrol <= 4'b1010; // sub
15           6'b100100: alucontrol <= 4'b0000; // and
16           6'b100101: alucontrol <= 4'b0001; // or
17           6'b101010: alucontrol <= 4'b1011; // slt
18           6'b000000: alucontrol <= 4'b0100; // sll
19           6'b110011: alucontrol <= 4'b0110; // zfr
20           default:   alucontrol <= 4'bxxxx; // ???
21         endcase
22     endcase
23 endmodule
24
```

From, the screenshot above, we can see the new bit combinations that would represent the operation to be done by the *alu* module. As mentioned, for previous instructions, the added extra bit (which is 0 for all previous instructions) was in between the 2nd and 3rd bit of the original *alucontrol* since the leftmost bit (3rd bit of original *alucontrol*, 4th bit of modified *alucontrol*) is essential to the code in *alu.sv* because it would determine whether to negate the value of input *b* or not. So, as of now, note the below table for our *alucontrol* (note *a* and *b* are the inputs to the *alu module*):

alucontrol	operation
0000	a AND b
0001	a OR b
0010	a + b
1010	a - b
1011	SLT

Table 1.

This can be seen in the code for *alu.sv* below: (again disregard new instructions for now) (Note again that the width of output port for *alucontrol* in *alu* module was changed to 4 bits)

```
1   module alu(input  logic [31:0] a, b,
2              input  logic [3:0]  alucontrol,
3              input logic [4:0] shamt,
4              output logic [31:0] result,
5              output logic        zero);
6
7     logic [31:0] condinv, sum;
8     logic [31:0] zfr;
9
10    always_comb
11    if (alucontrol == 4'b1101) begin
12       assign condinv = alucontrol[3] ? ~a : a;
13       assign sum = b + condinv + alucontrol[3];
14    end else begin
15      assign condinv = alucontrol[3] ? ~b : b;
16      assign sum = a + condinv + alucontrol[3];
17    end
18
19    assign zfr = a >> (b[4:0] + 1);
20
21    always_comb
22      case (alucontrol[2:0])
23        3'b000: result = a & b;
24        3'b001: result = a | b;
25        3'b010: result = sum;
26        3'b011: result = sum[31];
27        3'b100: result = b << shamt;
28        3'b101: result = sum[31];
29        3'b110: result = zfr << (b[4:0] + 1);
30      endcase
31
32    assign zero = (result == 32'b0);
33  endmodule
```

Note the lines 14-17 handles the addition of *a* and *b* using 2's complement. In line 15, it checks if the leftmost bit of our *alucontrol* is 1, and if it is, then *condinv* is set to NOT *b* otherwise it is set to just *b*. Thus, if the operation is addition, meaning the leftmost bit (*alucontrol[3]*) is zero, then *sum = a + condinv + alucontrol[3]*, which is essentially *a + b + 0*. But when the operation is subtraction (*alucontrol[3] = 1)* then we need to get NOT *b* and get their sum with *a*. Afterwards, we need to add 1 to the result. So for subtraction, *sum = a + condinv + alucontrol[3]* is essentially *a + (inverse of b + 1)*. Lines 22 – 26, shows the results of the operations found in *Table 1* above. Note that line 25, will have a result either for *a + b* or for *a – b*.

Following the change in the bit width of our *alucontrol*, we also have to change the codes for the ports in some modules:

- For *mips.sv*, when the *alucontrol* was first declared in line 11, we changed the bit width from *logic [2:0]* to *logic [3:0]*.

```
1  `timescale 1ns / 1ps
2  module mips(input  logic        clk, reset,
3              output logic [31:0] pc,
4              input  logic [31:0] instr,
5              output logic [1:0]  memwrite,
6              output logic [31:0] aluout, writedata,
7              input  logic [31:0] readdata);
8
9    logic       memtoreg, alusrc, regdst,
10             regwrite, jump, pcsrc, zero;
11   logic [3:0] alucontrol;
12
13   controller c(instr[31:26], instr[5:0], zero,
14                memtoreg, memwrite, pcsrc,
15                alusrc, regdst, regwrite, jump,
16                alucontrol);
17   datapath dp(clk, reset, memtoreg, pcsrc,
18               alusrc, regdst, regwrite, jump,
19               alucontrol,
20               zero, pc, instr,
21               aluout, writedata, readdata);
22 endmodule
```

- For *controller.sv*, we also changed the width of the output port for the *alucontrol* to 4 bits (line 9).

```
1  `timescale 1ns / 1ps
2  module controller(input  logic [5:0] op, funct,
3                    input  logic       zero,
4                    output logic       memtoreg,
5                    output logic [1:0] memwrite,
6                    output logic       pcsrc, alusrc,
7                    output logic       regdst, regwrite,
8                    output logic       jump,
9                    output logic [3:0] alucontrol);
10
11   logic [1:0] aluop;
12   logic       branch;
13
14   maindec md(op, memtoreg, memwrite, branch,
15              alusrc, regdst, regwrite, jump, aluop);
16   aludec  ad(funct, aluop, alucontrol);
17
18   assign pcsrc = branch & zero;
19 endmodule
```

- For *datapath.sv*, we also changed the width of the input port for the *alucontrol* to 4 bits (line 7).

```
1  ////////////
2  `timescale 1ns / 1ps
3  module datapath(input  logic        clk, reset,
4                  input  logic        memtoreg, pcsrc,
5                  input  logic        alusrc, regdst,
6                  input  logic        regwrite, jump,
7                  input  logic [3:0]  alucontrol,
8                  output logic        zero,
9                  output logic [31:0] pc,
10                 input  logic [31:0] instr,
11                 output logic [31:0] aluout, writedata,
12                 input  logic [31:0] readdata);
```

General Testbench used to test all added instructions

To test, the instruction, I used to same *testbench.sv* that was provided in Laboratory Report 12, with slight changes. I removed the $stop in line 33, so that the simulation would not stop whenever the *memwrite != 0* and the *dataadr != 80*.

```
1   `timescale 1ns / 1ps
2   module testbench();
3
4     logic        clk;
5     logic        reset;
6
7     logic [31:0] writedata, dataadr;
8     logic [1:0]   memwrite;
9     // instantiate device to be tested
10    top dut(clk, reset, writedata, dataadr, memwrite);
11
12    // initialize test
13    initial
14      begin
15        reset <= 1; # 22; reset <= 0;
16      end
17
18    // generate clock to sequence tests
19    always
20      begin
21        clk <= 1; # 5; clk <= 0; # 5;
22      end
23
24    // check results
25    always @(negedge clk)
26      begin
27        if(memwrite) begin
28          if(dataadr === 84 & writedata === 7) begin
29            $display("Simulation succeeded");
30            $stop;
31          end else if (dataadr !== 80) begin
32            $display("Simulation failed");
33          end
34        end
35      end
36  endmodule
```

Added Instructions

Normal Instruction – sll (shift-left logical)

Changes in HDL Code

Aside from the changes previously mentioned about the changing the width of the *alucontrol* in different modules, the majority of changes to our processor to add the instruction *sll* was in *aludec.sv* and *alu.sv*.

For *aludec.sv*, we added line 18 to the code. We know that the *funct* portion of the *sll* instruction format is 0b000000, and we set a new combination for the *alucontrol* for the *sll* instruction which is 0b0100. Screenshot of *aludec.sv* below:

```
1  ////////////
2  `timescale 1ns / 1ps
3  module aludec(input  logic [5:0] funct,
4                input  logic [1:0] aluop,
5                output logic [3:0] alucontrol);
6
7    always_comb
8      case(aluop)
9        2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/sb/addi)
10       2'b01: alucontrol <= 4'b1010;  // sub (for beq)
11       2'b11: alucontrol <= 4'b1101;  // sub (for ble)
12       default: case(funct)          // R-type instructions
13           6'b100000: alucontrol <= 4'b0010; // add
14           6'b100010: alucontrol <= 4'b1010; // sub
15           6'b100100: alucontrol <= 4'b0000; // and
16           6'b100101: alucontrol <= 4'b0001; // or
17           6'b101010: alucontrol <= 4'b1011; // slt
18           6'b000000: alucontrol <= 4'b0100; // sll
19           6'b110011: alucontrol <= 4'b0110; // zfr
20           default:   alucontrol <= 4'bxxxx; // ???
21         endcase
22     endcase
23 endmodule
24
```

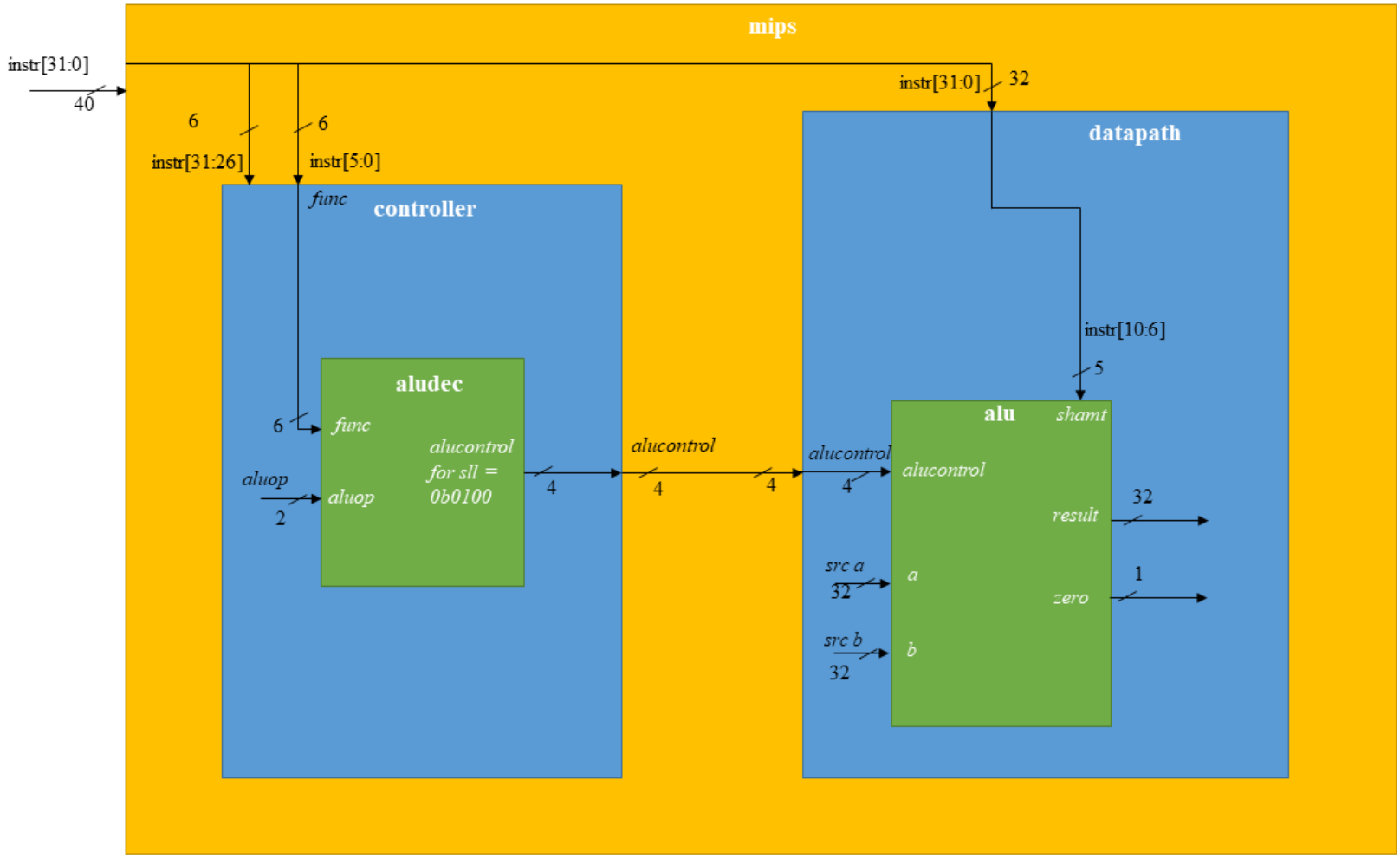
For *alu.sv*, we added an input port for the *shamt* portion of our *sll* instruction format to know how many positions we should shift the *input b* (value in register found in *rt* part of *sll* instruction format). We can ignore the changes we made to lines (10-17) for this instruction since the *result* of the *sll* instruction is not affected by lines 10-17. At line 27, since, we have set the *alucontrol* for *sll* to 0b0100, *alucontrol[2:0]* would be 0b100, and if the *alucontrol[2:0]* happens to have a value of 0b100, the *result* would be *b* << *shamt.* Note that the << operator in SystemVerilog is a standard bitshift operator such that it performs a logical left shift. So, in line 27, input b is shifted to the left by the value of input *shamt.* Screenshot of *alu.sv* below:

```
1   module alu(input  logic [31:0] a, b,
2              input  logic [3:0]  alucontrol,
3              input logic [4:0] shamt,
4              output logic [31:0] result,
5              output logic        zero);
6
7     logic [31:0] condinv, sum;
8     logic [31:0] zfr;
9
10    always_comb
11    if (alucontrol == 4'b1101) begin
12       assign condinv = alucontrol[3] ? ~a : a;
13       assign sum = b + condinv + alucontrol[3];
14    end else begin
15      assign condinv = alucontrol[3] ? ~b : b;
16      assign sum = a + condinv + alucontrol[3];
17    end
18
19    assign zfr = a >> (b[4:0] + 1);
20
21    always_comb
22      case (alucontrol[2:0])
23        3'b000: result = a & b;
24        3'b001: result = a | b;
25        3'b010: result = sum;
26        3'b011: result = sum[31];
27        3'b100: result = b << shamt;
28        3'b101: result = sum[31];
29        3'b110: result = zfr << (b[4:0] + 1);
30      endcase
31
32    assign zero = (result == 32'b0);
33  endmodule
```

Adding an input port to the *alu.sv* for *shamt* will reflect changes to the *datapath.sv*, so the input for *shamt* in *alu.sv* will actually have a value. This can be seen in line 39, wherein the input for *shamt* in *alu.sv* is added when the *alu* was instantiated. We know that the *shamt* corresponds to bits[10:6] of the *sll* instruction format. Screenshot of *datapath.sv* below:

```
////////////
`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,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb;
  logic [31:0] result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder #(32) pcaddl(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
  regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
                 writereg, result, srca, writedata);
  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);
  alu         alu(srca, srcb, alucontrol, instr[10:6], aluout, zero);
endmodule
```

Schematic Diagram (Note that the schematic diagram only shows the parts/wires relevant to the instruction or the parts wherein the code was changed/added for the instruction).



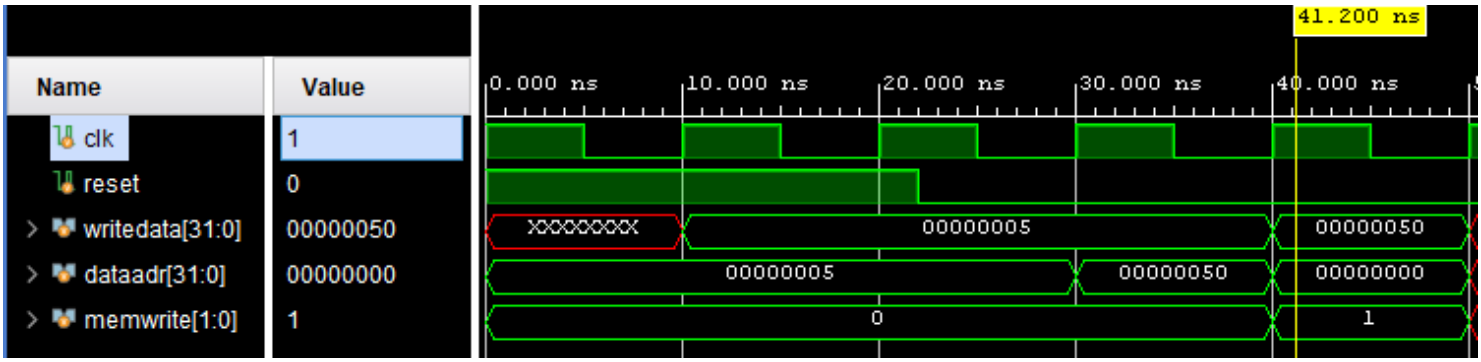
For *sll*, remember that we have changed the bit width of the *alucontrol* from 3 bits to 4 bits, thus, the schematic above shows all the components that was affected by this change. First, the *aludec*, wherein the *alucontrol* gets a value depending on the *func* (0b00000). We have set a value of *alucontrol* to be 0b0100 for *sll*. Next, the wire connecting the output *alucontrol* from *aludec* continues until it reaches *datapath* and *alu*, all have the same width of 4 bits. Next, another change was that we added an input *shamt* (instr[10:6]) of width 5 to *alu* so that the *alu* can compute the result depending on the value of the *alucontrol*. Since, the *alucontrol* for *sll* is 0b0100, then *result* = *b* << *shamt*.

Testing of Instruction

To test the *sll* instruction, we prepared two sample sets of instructions to copy into *memfile.mem*. The first one is:

#	Assembly	Description	Machine
main:	addi $2, $0 , 5	Initialize $2 = 5	20020005
	sll $2, $2, 4	Shift value in $2 to the left by 4	00021100
	sw $2, 0($0)	Store value in $2 to address 0 + $0	ac020000

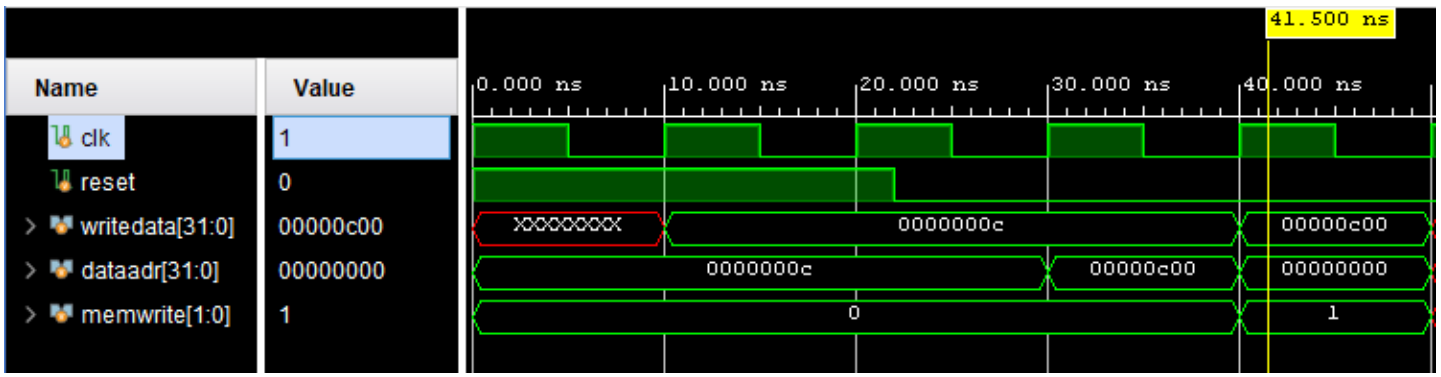
To analyze this example, we know that after the first instruction, $2 will have a value of 0b00000000000000000000000000000101 or 0x5, and we shift this value to the left by 4, we will obtain 0b00000000000000000000000001010000 or 0x50. So, in instruction 3, the value that will be stored in address 0 is 0x50. This can be confirmed in the simulation below: (Note the values at a given time in the cycle can be seen in the left side of the screenshot)



We can see that when *memwrite* has a value of 1 (0b01 = *sw*), the value that was stored (can be seen in *writedata*) is 0x00000050, which is the correct value. Another example would be the following instructions:

#	Assembly	Description	Machine
main:	addi $3, $0, 12	Initialize $3 = 12	2003000c
	sll $3, $3, 8	Shift value in $3 to the left by 8	00031a00
	sw $3, 0($0)	Store value in $3 to address 0 + $0	ac030000

To analyze this example, for the first instruction, we know that $3 has a value of 0b00000000000000000000000000001100 or 0xC, and if we shift this to the left by 8, we will get 0b00000000000000000000110000000000 or 0xC00, so the result to be stored at address 0 (instruction 3) should be 0xC00. We can see in the simulation below that when *memwrite* has a value of 1 (0b01 = *sw*), the value stored was 0xC00.



Normal Instruction: sb (store-byte)

Changes in HDL Code

The changes to implement *sb* happened mostly in *maindec.sv* and *dmem.sv*. First, we decided to change the bit width of *memwrite* from 1 bit to 2 bits. This can be seen in *maindec.sv*, first, when the output port for *memwrite* was changed to 2 bits in line 4. Next, the logic variable *controls* was adjusted to become 10 bits. Note that in line 12, the order of the variables assigned to *controls* remained the same, we just added a new bit to extend *memwrite*. Next, we also changed lines 19 and 20. This is because we know that *memwrite* will affect the *sw and sb* instructions. All other instructions in lines 17-24 shall have a new *memwrite* value of just 00 (added a 0 bit). But for *sw*, we changed *controls[5]* and *controls[4]* to 0 and 1 respectively, which makes *memwrite* for *sw:* 0b01. Next, we also set *controls* for instruction *sb* to be 0b0010100000 which makes *memwrite* for *sb*: 0b10. Screenshot of *maindec.sv* below:

```
1  `timescale 1ns / 1ps
2  module maindec(input  logic [5:0] op,
3                 output logic       memtoreg,
4                 output logic [1:0] memwrite,
5                 output logic       branch, alusrc,
6                 output logic       regdst, regwrite,
7                 output logic       jump,
8                 output logic [1:0] aluop);
9
10   logic [9:0] controls;
11
12   assign {regwrite, regdst, alusrc, branch, memwrite,
13           memtoreg, jump, aluop} = controls;
14
15   always_comb
16     case(op)
17       6'b000000: controls <= 10'b1100000010; // RTYPE and ZFR
18       6'b100011: controls <= 10'b1010001000; // LW
19       6'b101011: controls <= 10'b0010010000; // SW
20       6'b101000: controls <= 10'b0010100000; // SB
21       6'b000100: controls <= 10'b0001000001; // BEQ
22       6'b011111: controls <= 10'b0001000011; // BLE
23       6'b001000: controls <= 10'b1010000000; // ADDI
24       6'b000010: controls <= 10'b0000000100; // J
25       default:   controls <= 10'bxxxxxxxxxx; // illegal op
26     endcase
27 endmodule
```

Next, for *dmem.sv*, we know that it uses *memwrite* as an input so we also changed the width of the input logic *we* to 2 bits. Next, in lines 11-19, we added an *always_ff @ (posedge clk)* block and in it are conditional statements. These conditional statements will handle what instruction to perform whether *sw* or *sb*. If *we* == 0b01, then the instruction is *sw* and we should store the whole 32 bits of input *wd*. However, if *we* = 0b10, then the instruction is *sb* and we should

note since memory in MIPS is word-aligned then, when storing a byte, we should know what specific address the byte should be stored to. Since we need to store it in big endian, and we know that the memory addresses would always have rightmost 2 bits of either 0b00, 0b10, 0b01, or 0b11), we can store the bytes to the address with respect to the rightmost 2 bits of *a*. If a[1:0] is 0b11 then we should store at RAM[a[31:2]][7:0], if a[1:0] is 0b10 then we should store at RAM[a[31:2]][15:8], if a[1:0] is 0b01 then we should store to RAM[a[31:2]][23:16], and if a[1:0] is 0b00 then we should store at RAM[a[31:2]][31:24]]. Screenshot of *dmem.sv* below:

```
1   `timescale 1ns / 1ps
2   module dmem(input  logic        clk,
3               input logic [1:0]   we,
4               input logic [31:0] a, wd,
5               output logic [31:0] rd);
6
7     logic [31:0] RAM[63:0];
8
9     assign rd = RAM[a[31:2]]; // word aligned
10
11    always_ff @(posedge clk)
12          if (we == 2'b01) begin
13              RAM[a[31:2]] <= wd;
14          end else if (we == 2'b10) begin
15              if (a[1:0] == 2'b11) RAM[a[31:2]][7:0] <= wd[7:0];
16              if (a[1:0] == 2'b10) RAM[a[31:2]][15:8] <= wd[7:0];
17              if (a[1:0] == 2'b01) RAM[a[31:2]][23:16] <= wd[7:0];
18              if (a[1:0] == 2'b00) RAM[a[31:2]][31:24] <= wd[7:0];
19          end
20  endmodule
```

Note that changing the width of *memwrite* will also cause changes to other modules' ports. First, *mips.sv*: line(5)

```
1  `timescale 1ns / 1ps
2  module mips(input  logic        clk, reset,
3              output logic [31:0] pc,
4              input  logic [31:0] instr,
5              output logic [1:0]  memwrite,
6              output logic [31:0] aluout, writedata,
7              input  logic [31:0] readdata);
8
9    logic       memtoreg, alusrc, regdst,
10             regwrite, jump, pcsrc, zero;
11   logic [3:0] alucontrol;
12
13   controller c(instr[31:26], instr[5:0], zero,
14                memtoreg, memwrite, pcsrc,
15                alusrc, regdst, regwrite, jump,
16                alucontrol);
17   datapath dp(clk, reset, memtoreg, pcsrc,
18               alusrc, regdst, regwrite, jump,
19               alucontrol,
20               zero, pc, instr,
21               aluout, writedata, readdata);
22 endmodule
```

Next, in line 5 of *controller.sv*, we also changed the width of the output port for *memwrite* to 2 bits:

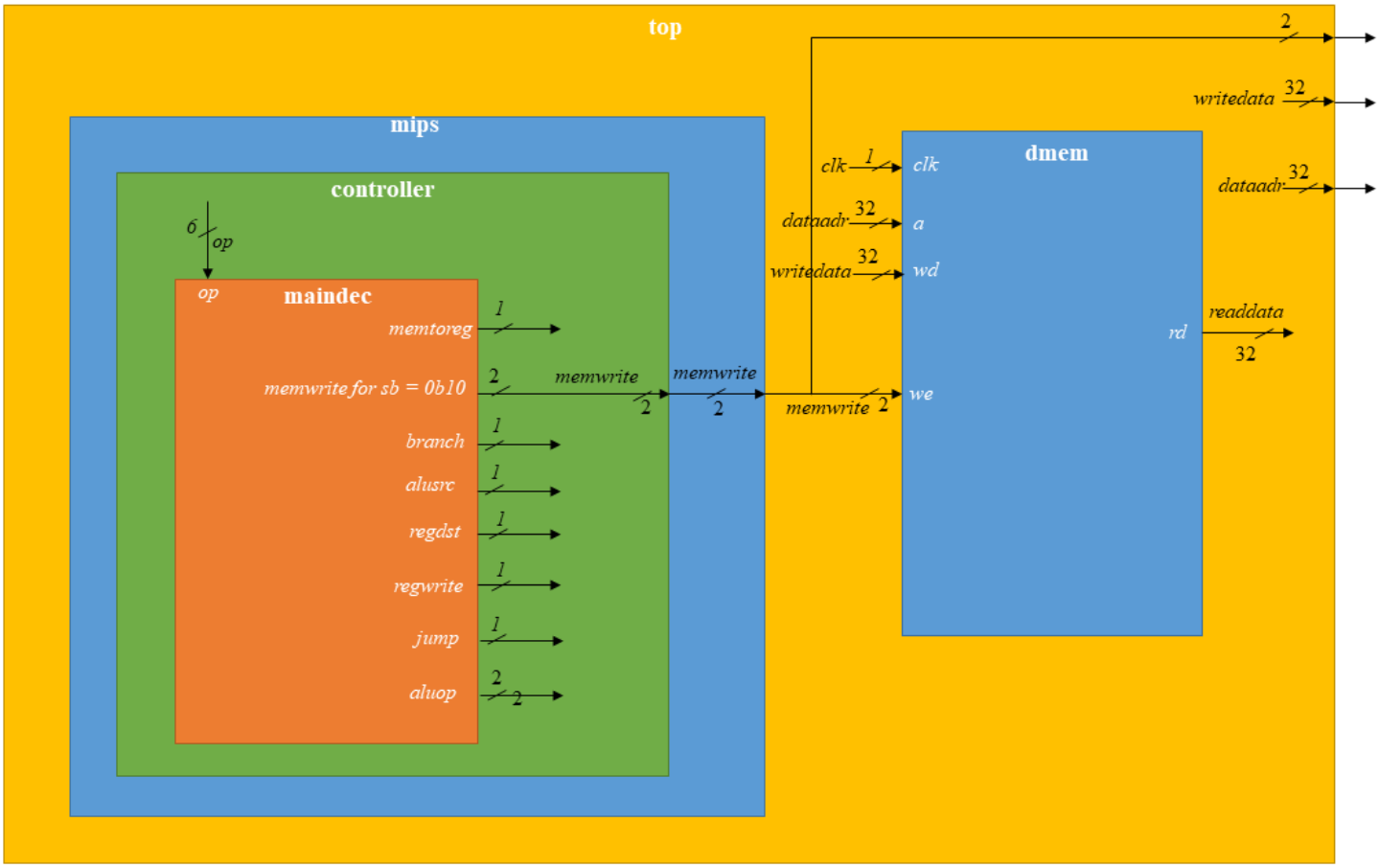
```
1  `timescale 1ns / 1ps
2  module controller(input  logic [5:0] op, funct,
3                    input  logic       zero,
4                    output logic       memtoreg,
5                    output logic [1:0] memwrite,
6                    output logic       pcsrc, alusrc,
7                    output logic       regdst, regwrite,
8                    output logic       jump,
9                    output logic [3:0] alucontrol);
10
11   logic [1:0] aluop;
12   logic       branch;
13
14   maindec md(op, memtoreg, memwrite, branch,
15              alusrc, regdst, regwrite, jump, aluop);
16   aludec  ad(funct, aluop, alucontrol);
17
18   assign pcsrc = branch & zero;
19 endmodule
```

Lastly, in *top.sv*, we also changed the bit width of *memwrite* to 2 bits (line 5).

```
1  `timescale 1ns / 1ps
2  module top(input  logic        clk, reset,
3             output logic [31:0] writedata, dataadr,
4             output logic [1:0]  memwrite);
5
6    logic [31:0] pc, instr, readdata;
7    // instantiate processor and memories
8    mips mips(clk, reset, pc, instr, memwrite, dataadr,
9              writedata, readdata);
10   imem imem(pc[7:2], instr);
11   dmem dmem(clk, memwrite, dataadr, writedata, readdata);
12 endmodule
```

Note that there were no changes in *aludec.sv* since *sw* and *sb* involves the same operation to be done in *alu.sv*.

Schematic Diagram (Note that the schematic diagram only shows the parts/wires relevant to the instruction or the parts wherein the code was changed/added for the instruction).



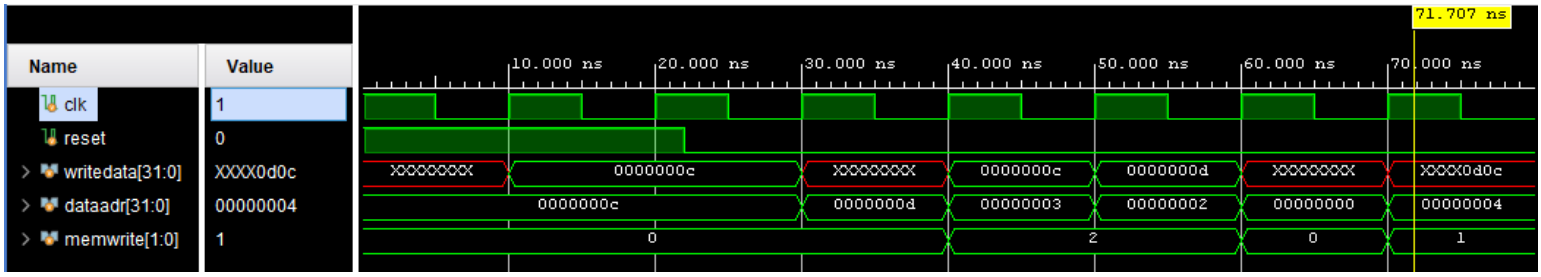
For *sb* remember that we changed the width of *memwrite* to become 2 bits, thus, depending on the value of the *opcode*, we get a value for *memwrite*. Note that the other outputs for *maindec* aside from *memwrite* is the same as the outputs for *sw* instruction. Now, we know that the *opcode* for *sb* is 0b101000 and we set the *memwrite* value for *sb* to 0b10. We also set the *memwrite* value for *sw* to 0b01. Next, the output *memwrite* will also be an output for *controller*, *mips,* and *top.* It is also an input for *dmem* which is named *we*. In *dmem*, we previously stated the operations that were done if *we* is 0b01 or if *we* is 0b10.

Testing of Instruction

To test the *sb* instruction, I prepared a set of instructions to be copied into *memfile.mem*, and be performed by the processor:

#	Assembly	Description	Machine
main:	addi $2, 0, 12	Initialize $2 = 0xC	2002000c
	addi $3, 0, 13	Initialize $2 = 0xD	2003000d
	sb $2, 3($0)	Store byte value of $2 in address 3 + $0	a0020003
	sb $3, 2($0)	Store byte value of $3 in address 2 + $0	a0030002
	lw $4, 0($0)	Load word value in address 0 + 0 + $0	8c040000
	sw $4, 4($0)	Store word value of $4 in address 4 + $0	ac040004

We can see the results of these instructions in the screenshot of the simulation below:



Note that after the first two instructions, $2 has a value of 0x0C and $3 has a value of 0x0D. Next, at 40.000 ns we can see that *memwrite* has a value of 2 (0b10 = *sb*) so it stores a byte value in $2 in address 3 + $0, next at 50.000 ns we can see that *memwrite* is still 2, so it stores a byte value in $3 in address 2 + $0. Next, because of the *lw* instruction, $4 should have a value of = 0xXXXX0d0c (because of big endianness). So, we can see at 70.000 when *memwrite* is 1 (0b01 = *sw*) that the value being stored is the correct value = 0xXXXX0d0c.

Pseudo-Instruction: ble (branch – less than or equal)

Changes in HDL Code

The changes for adding *ble* pseudo-instruction is found in *maindec.sv*, *aludec.sv*, *alu.sv*. For *maindec.sv*, we added an opcode for *ble* which is 0b011111, *controls* for *ble* is almost the same for what we have in *beq*, but they just differ in in their *aluop*, wherein we set the *aluop* of *BLE* to 0b11 (which was previously not used) (Note that *branch* = 1). Screenshot of *maindec.sv* below:

```
1  `timescale 1ns / 1ps
2  module maindec(input  logic [5:0] op,
3                 output logic       memtoreg,
4                 output logic [1:0] memwrite,
5                 output logic       branch, alusrc,
6                 output logic       regdst, regwrite,
7                 output logic       jump,
8                 output logic [1:0] aluop);
9
10   logic [9:0] controls;
11
12   assign {regwrite, regdst, alusrc, branch, memwrite,
13           memtoreg, jump, aluop} = controls;
14
15   always_comb
16     case(op)
17       6'b000000: controls <= 10'b1100000010; // RTYPE and ZFR
18       6'b100011: controls <= 10'b1010001000; // LW
19       6'b101011: controls <= 10'b0010010000; // SW
20       6'b101000: controls <= 10'b0010100000; // SB
21       6'b000100: controls <= 10'b0001000001; // BEQ
22       6'b011111: controls <= 10'b0001000011; // BLE
23       6'b001000: controls <= 10'b1010000000; // ADDI
24       6'b000010: controls <= 10'b0000000100; // J
25       default:   controls <= 10'bxxxxxxxxxx; // illegal op
26     endcase
27 endmodule
```

\

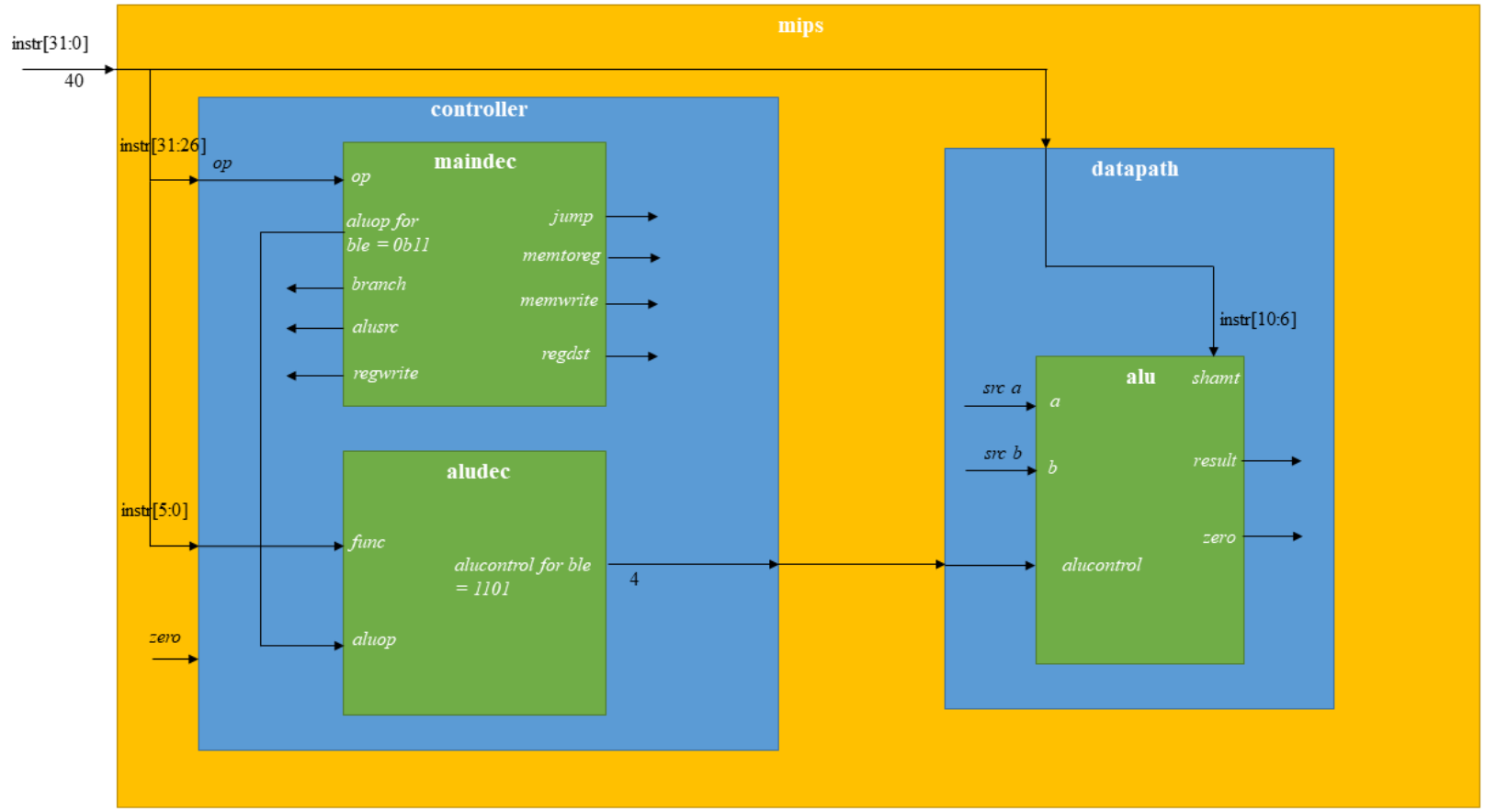
For *aludec.sv*, we also setup an *alucontrol* for *ble* which is 0b1101, note that the leftmost bit is 1 since we need subtraction in *alu.* So, we added line 11, wherein if *aluop* is 0b11, then the *alucontrol*, will be 0b1101 for ble. Screenshot of *aludec.sv* below:

```
1  ////////////
2  `timescale 1ns / 1ps
3  module aludec(input  logic [5:0] funct,
4                input  logic [1:0] aluop,
5                output logic [3:0] alucontrol);
6
7    always_comb
8      case(aluop)
9        2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/sb/addi)
10       2'b01: alucontrol <= 4'b1010;  // sub (for beq)
11       2'b11: alucontrol <= 4'b1101;  // sub (for ble)
12       default: case(funct)          // R-type instructions
13           6'b100000: alucontrol <= 4'b0010; // add
14           6'b100010: alucontrol <= 4'b1010; // sub
15           6'b100100: alucontrol <= 4'b0000; // and
16           6'b100101: alucontrol <= 4'b0001; // or
17           6'b101010: alucontrol <= 4'b1011; // slt
18           6'b000000: alucontrol <= 4'b0100; // sll
19           6'b110011: alucontrol <= 4'b0110; // zfr
20           default:   alucontrol <= 4'bxxxx; // ???
21         endcase
22     endcase
23 endmodule
24
```

For *alu.sv*, we added the conditional statements in line 11 to line 17, wherein lines 14-17 contain the original *assign* methods to *condinv* and *sum*. In lines 11 to 13, we added that if *alucontrol* is 0b1101 (*ble*), then, *condinv* is equal to the inverse of a and that the *sum* is b + (inverse of a) + 1. This is because the MIPS-assembler translates the *ble* pseudo-instruction to two instructions which is *slt* and *beq (branch if result of slt is equal to zero)*. But in *slt*, instead of checking a < b, we check b < a. So that if b < a, then result of *slt* is 1 so the *zero* variable in line 32 will be set to zero and thus, will not branch. But if b < a is false, then the result of *slt* is 0, so the zero variable would be set to 1, and thus, branch. The result of the *slt* can be seen in line 28 wherein it checks the 31st bit of the *sum* because of it is 1 then the sum is negative (b < a), but if it is 0 then the sum is positive or zero (a <= b). Screenshot of *alu.sv* below:

```
1   module alu(input  logic [31:0] a, b,
2              input  logic [3:0]  alucontrol,
3              input logic [4:0] shamt,
4              output logic [31:0] result,
5              output logic        zero);
6
7     logic [31:0] condinv, sum;
8     logic [31:0] zfr;
9
10    always_comb
11    if (alucontrol == 4'b1101) begin
12       assign condinv = alucontrol[3] ? ~a : a;
13       assign sum = b + condinv + alucontrol[3];
14    end else begin
15      assign condinv = alucontrol[3] ? ~b : b;
16      assign sum = a + condinv + alucontrol[3];
17    end
18
19    assign zfr = a >> (b[4:0] + 1);
20
21    always_comb
22      case (alucontrol[2:0])
23        3'b000: result = a & b;
24        3'b001: result = a | b;
25        3'b010: result = sum;
26        3'b011: result = sum[31];
27        3'b100: result = b << shamt;
28        3'b101: result = sum[31];
29        3'b110: result = zfr << (b[4:0] + 1);
30      endcase
31
32    assign zero = (result == 32'b0);
33  endmodule
```

Schematic Diagram (Note that the schematic diagram only shows the parts/wires relevant to the instruction or the parts wherein the code was changed/added for the instruction).



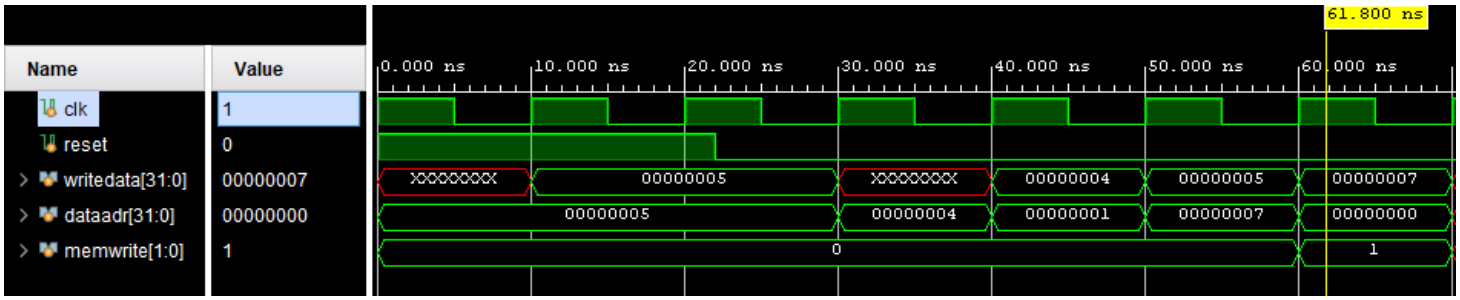
For *ble*, we set its *aluop* in the *maindec* to be 0b11, this *aluop* would go to the *aludec* wherein we use it to set its *alucontrol* in *aludec* to be 0b1101. Then, this *aluop* now goes to *datapath* and *alu*, wherein if the *alucontrol* is 0b1101, the result would be the *slt* result of b < a and the *zero* would be set to 1 if *slt* is 0 and *zero* would be set to 0 if *slt* is 1.

Testing of Instruction

To test the *ble* instruction, we have prepared 2 sets of instructions to be performed. The first one would be:

#	Assembly	Description	Machine
main:	addi $2, $0 ,5	Initialize $2 = 5	20020005
	addi $3, $0, 4	Initialize $3 = 4	20030004
	ble $2, $3, end	branch to end if $2 <= $3	7c430001
	addi $2, $0, 7	Set $2 = 0 + 7 = 7	20020007
end:	sw $2, 0($0)	Store	ac020000

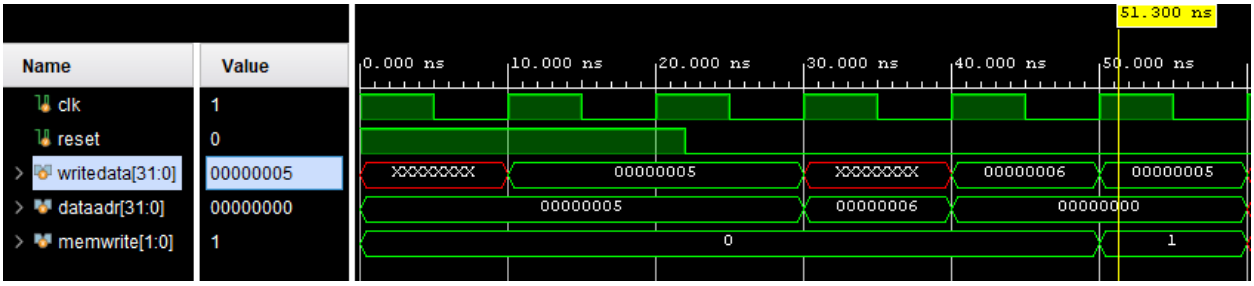
To analyze the set of instructions, we know that after the first two instructions, $2 = 5, and $3 = 4. So in the *ble* instruction, we know that 5 > 4, so we should not branch. Thus, $2 is set to have a value of 7 in instruction 4. And hence, the value to be stored is the value in $2 = 0x7. We can confirm this in the simulation below:



We can see that when *memwrite* is 1 (0b01 = *sw*), the value to be stored is 0x07 which is correct. For the next example, we have:

#	Assembly	Description	Machine
main:	addi $2, $0 ,5	Initialize $2 = 5	20020005
	addi $3, $0, 6	Initialize $3 = 6	20030006
	ble $2, $3, end	branch to end if $2 <= $3	7c430001
	addi $2, $0, 7	Set $2 = 0 + 7 = 7	20020007
end:	sw $2, 0($0)	Store	ac020000

Note that after two instructions, we know that $2 = 5, and $3 = 6, so in the *ble* instruction, we know that 5 < 6 so we should take the branch. Hence, the 4th instruction is skipped. Thus, the value to be stored at label *end* is 0x5. The screenshot of the simulation is found below:



We can see that when *memwrite* is 1 (0b01 = *sw*), the value that would be stored in address 0 is 0x5, which is correct.

Custom Instruction: zfr

Changes in HDL Code

The changes in the HDL code for *zfr* is found mainly on *aludec.sv* and *alu.sv*. There were no changes in *maindec.sv* since the opcode for *zfr* is 0x00 is considered an R-type instruction. For the *aludec.sv*, we know that the *func* for *zfr* is 0x33 or 0b110011, so we added line 19 and we set up an *alucontrol* for *zfr* which is 0110. Screenshot of *aludec.sv* below:

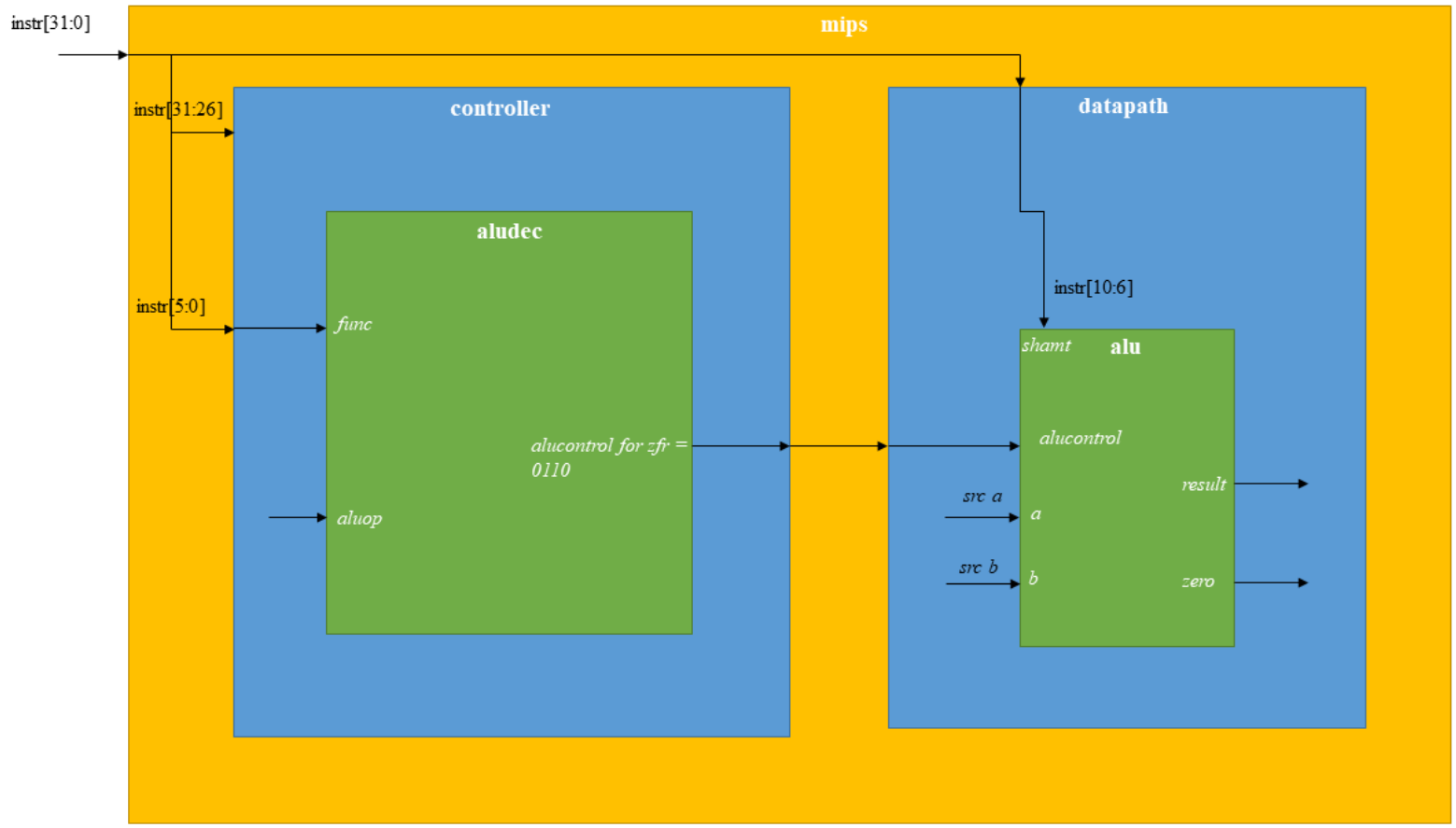
```
1  ////////////
2  `timescale 1ns / 1ps
3  module aludec(input  logic [5:0] funct,
4                input  logic [1:0] aluop,
5                output logic [3:0] alucontrol);
6
7    always_comb
8      case(aluop)
9        2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/sb/addi)
10       2'b01: alucontrol <= 4'b1010;  // sub (for beq)
11       2'b11: alucontrol <= 4'b1101;  // sub (for ble)
12       default: case(funct)          // R-type instructions
13           6'b100000: alucontrol <= 4'b0010; // add
14           6'b100010: alucontrol <= 4'b1010; // sub
15           6'b100100: alucontrol <= 4'b0000; // and
16           6'b100101: alucontrol <= 4'b0001; // or
17           6'b101010: alucontrol <= 4'b1011; // slt
18           6'b000000: alucontrol <= 4'b0100; // sll
19           6'b110011: alucontrol <= 4'b0110; // zfr
20           default:   alucontrol <= 4'bxxxx; // ???
21         endcase
22     endcase
23 endmodule
24
```

For *alu.sv*, we added a new 32-bit logic variable named *zfr* (line 8). Note that the instruction *zfr* shall set the bit 0 to bit[*b*[4:0]] of *a* to zero. Notice that this is equivalent to if we shift a to the

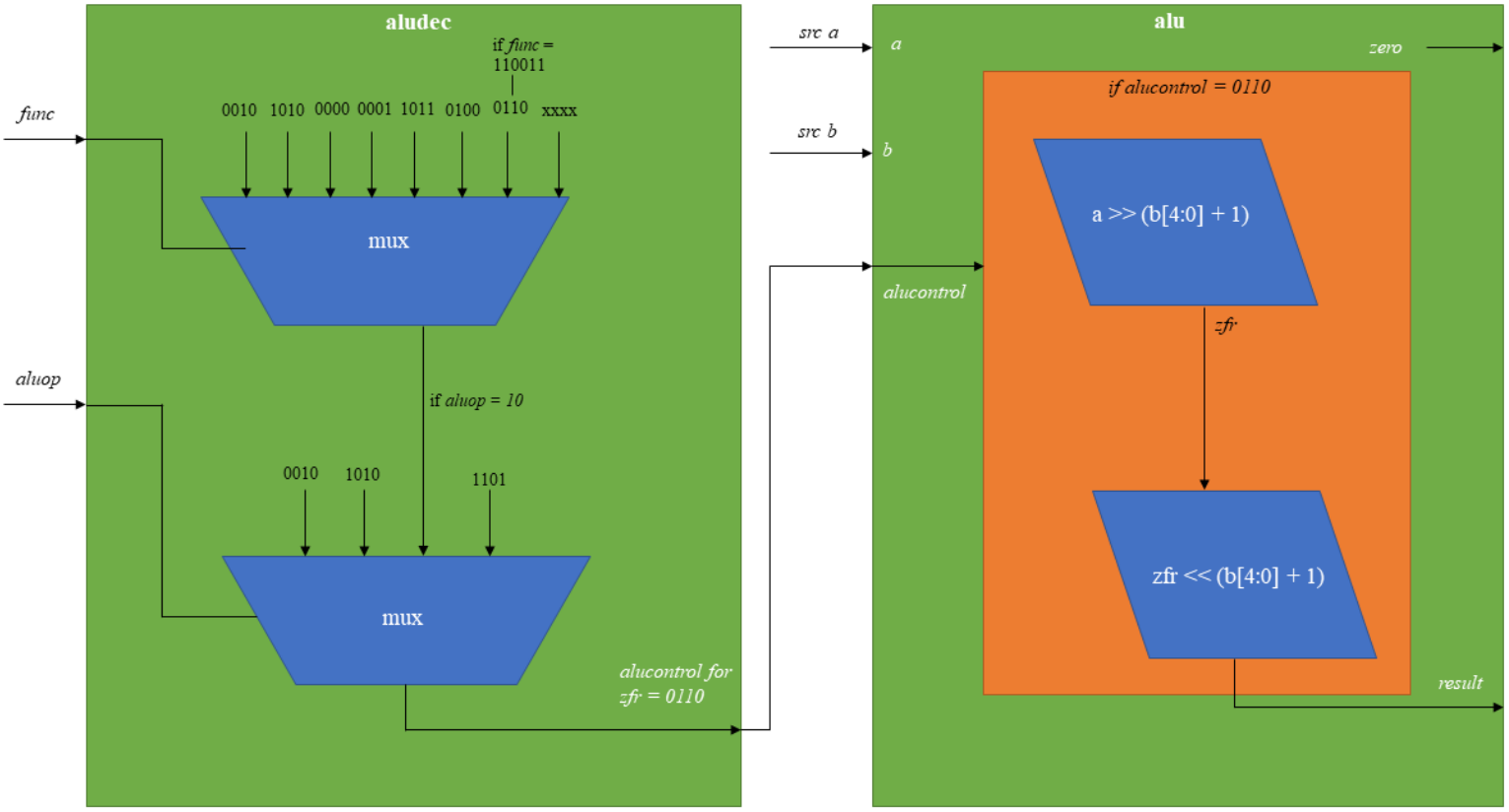
right by the value in *b[4:0] + 1* which gets rid of bit 0 to bit[*b*[4:0]] of *a* (we set the result of this operation to *zfr* (line 19)) and then shifting the result again but this time to the left by the value *b[4:0] + 1*, which then automatically uses zeros to fill the spaces up (line 29). Note that we will use >> operator for the right shift and << operator for the left shift. Screenshot of *alu.sv* below:

```
1   module alu(input  logic [31:0] a, b,
2              input  logic [3:0]  alucontrol,
3              input logic [4:0] shamt,
4              output logic [31:0] result,
5              output logic        zero);
6
7     logic [31:0] condinv, sum;
8     logic [31:0] zfr;
9
10    always_comb
11    if (alucontrol == 4'b1101) begin
12       assign condinv = alucontrol[3] ? ~a : a;
13       assign sum = b + condinv + alucontrol[3];
14    end else begin
15      assign condinv = alucontrol[3] ? ~b : b;
16      assign sum = a + condinv + alucontrol[3];
17    end
18
19    assign zfr = a >> (b[4:0] + 1);
20
21    always_comb
22      case (alucontrol[2:0])
23        3'b000: result = a & b;
24        3'b001: result = a | b;
25        3'b010: result = sum;
26        3'b011: result = sum[31];
27        3'b100: result = b << shamt;
28        3'b101: result = sum[31];
29        3'b110: result = zfr << (b[4:0] + 1);
30      endcase
31
32    assign zero = (result == 32'b0);
33  endmodule
```

Schematic Diagram (Note that the schematic diagram only shows the parts/wires relevant to the instruction or the parts wherein the code was changed/added for the instruction).



For *zfr*, we know that the *func* is 0x33 = 0b110011 and if that is the value of input *func* in *aludec*, then the *alucontrol* shall be equal to 0b0110. This *alucontrol* will then go to *alu* to determine what operation to perform. For additional information on these, consider another diagram:



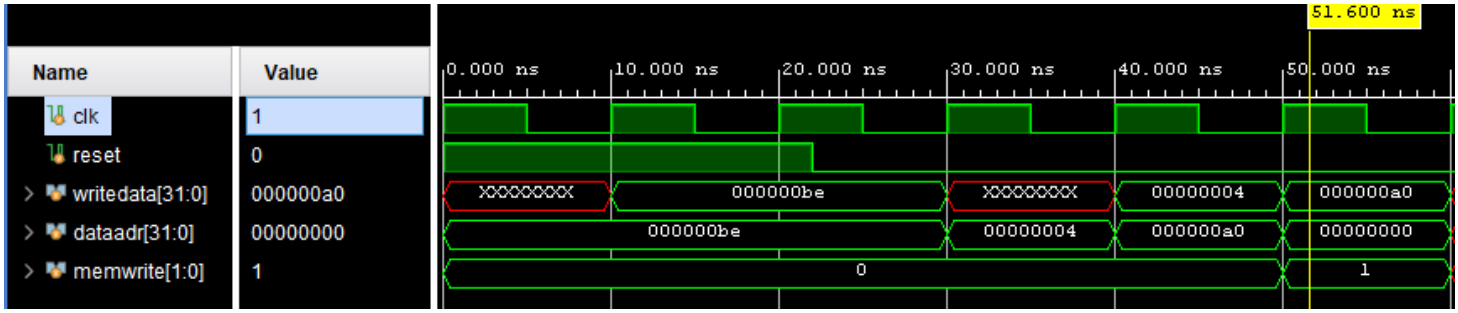
Note that multiplexers were used to represent the *case* implementation in the code. So, given the input *func* in *aludec*, when the *func* is 0b110011, then a value of 0b0110 will pass through the first multiplexer. For the next multiplexer, if input *aluop* is 0b10, then the same value will pass through it. And this in turn is the *alucontrol* dedicated for instruction *zfr*. This *alucontrol* will then continue as an input to the *alu*. Note that we have *a* and *b* also as inputs for *alu*. If the *alucontrol* is 0b0110, then the following operations inside the orange rectangle above will be performed. First, the value in *a* shall be shifted to the right by (b[4:0] + 1) positions, we call this value *zfr*. Next, *zfr* will now get shifted to the left b y (b[4:0] + 1) positions, thus completing the *zfr* instruction. This value is the *result*.

Testing of Instruction

To test the *zfr* instruction, we have prepared two sets of instructions to be performed. The first one is:

#	Assembly	Description	Machine
main:	addi $2, 0, 0xBE	Initialize $2 = 0xBE	200200be
	addi $3, $0, 4	Initialize $3 = 4	20030004
	zfr $2, $2, $3	Apply *zfr* to value in $2 by value in $3	00431033
	ac020000	Set $2 = 0 + 7 = 7	ac020000

To analyze the instructions, we know that after two instructions, the value of $2 = 0xBE and the value of $3 = 4. So, applying the *zfr* to $2 by the value in $3, we know that 0xBE = 0b10111110, so if we set bits [4:0] of the value in $2, we have 0b10100000 = 0xA0. So, the value to be stored in address 0 is 0x50. We can see this in the simulation below:



We can see that when *memwrite = 1* (0b01 = *sw*), the value to be stored at address 0 is indeed 0xa0. For the next set of instructions, we have:

#	Assembly	Description	Machine
main:	addi $2, 0, 0xFC	Initialize $2 = 0xFC	200200FC
	addi $3, $0, 5	Initialize $3 = 5	20030005
	zfr $2, $2, $3	Apply *zfr* to value in $2 by value in $3	00431033
	ac020000	Set $2 = 0 + 7 = 7	ac020000

The given example above performs the *zfr* instruction, wherein if $2 = 0xFF and we apply *zfr* with a value of $3 = 5 to it, the value we should get is 0xFFC0. This is because 0xFF = 0b11111111 and if we set its bits [5:0] to zero, we would obtain 0b11000000 = 0xC0. We can confirm this in the simulation below wherein if *memwrite* is = 1 (0b01 = *sw*). The value to be stored at address 0 is indeed 0xC0.