CS21 Project 2
MIPS Single-cycle Processor Extension

Justin Jose R. Ruaya
2019-01120

Submitted to:
Wilson M. Tan
Ivan Carlo Balingit

Department of Computer Science
University of the Philippines - Diliman

Contents

List of Figures

List of Codes

In partial fulfillment
of the requirements
in

CS 21: Computer Organization and Architecture

Abstract

Given the MIPS Single-cycle processor in Laboratory exercise 12, we modify the architecture to support new instructions: **sll**, **sb**, **ble**, **li**, and a custom instruction **zfr** (zero-from-right). The processor and its extension was implemented in SystemVerilog via Xilinx's Vivado software. A testbench was created to (a) check if prior instructions didn't break and (b) to check if the new instructions work.

Part 1

Introduction

1.1 MIPS Single-cycle Processor

We start by introducing the single-cycle processor. This is the same processor in laboratory exercise 12, where it supports these instructions:

1. lw or load word
2. sw or store word
3. beq or branch if equals
4. addi or add immediate
5. j or jump
6. add or add registers
7. sub or subtract registers
8. and
9. or
10. slt or set if less than

We aim to implement new instructions.

1.2 Final Processor

Before I present the Verilog modifications, I will present the full **dut** (top.sv) schematic as seen in the next page (*proudly created in LaTeX's Tikz*). The schematic contains **dut** (top.sv), **mips** (mips.sv), **c** (controller.sv) and **dp** (datapath.sv). The signals in blue signify the control signals. The signals in pink signify altered bits from the HDL code. This includes the inclusion of three control signals: bytemode (writes in bytes rather than in words given A), zeroextend (zero extends immediate value rather than sign extends), and disableRA1 (sets value of RA1 = 0). The **instr** signal was also added in order for the testbench to show the current instruction being executed in the processor.

These modifications (coupled with the modifications inside the modules) allow the new processor to support the following instructions: **sll** (shift left logical), **sb** (store byte), **li** (load immediate), **ble** (branch if less than), and **zfr** (a custom instruction called "zero from right").

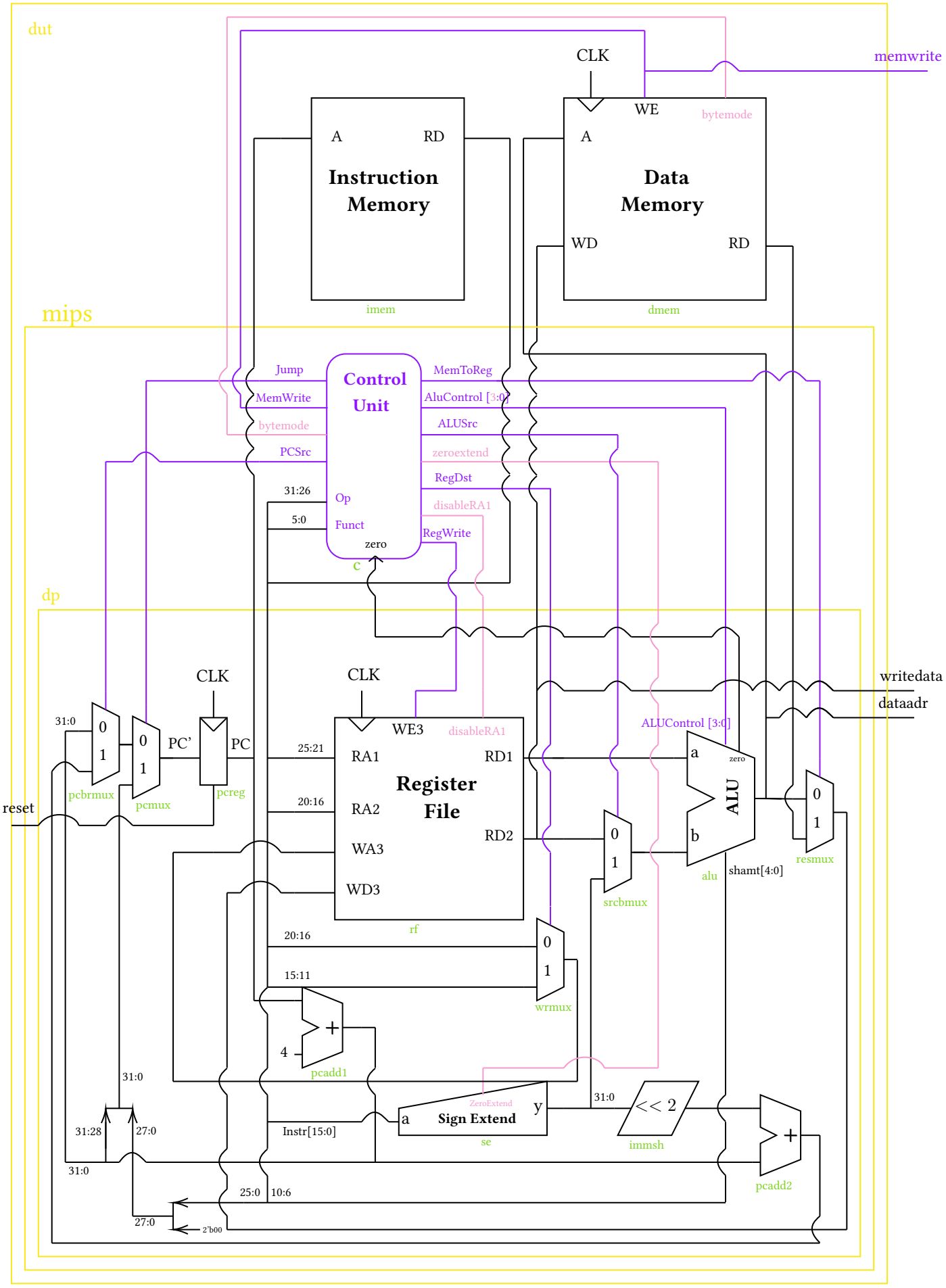

Figure 1.1: FULL Extended SCP Schematic

1.3 Integrity Testbench

Before we explain the additional instructions, we have to **check if our processor works as intended and didn't break anything**. With that said, consider the assembly code (taken from CS 21 Laboratory exercise 12)

```
# mipstest.asm
# David_Harris@hmc.edu, Sarah_Harris@hmc.edu 31 March 2012
#
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#          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
```

Figure 1.2: Integrity Testbench

Figure 1.3: Integrity Testbench Waveform

We can compare it with the results obtained from laboratory exercise 12 (next page):

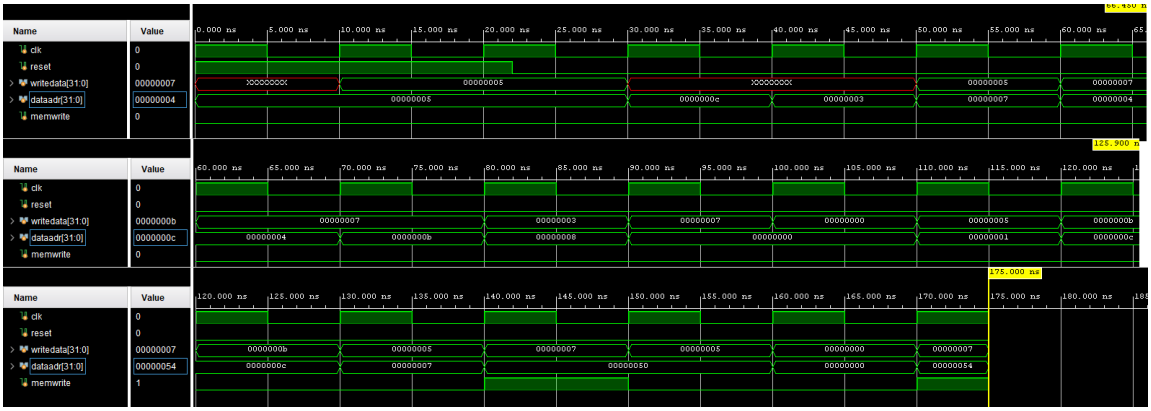

Figure 1.4: Integrity Testbench Waveform from Laboratory exercise 12

With all of that said, the SCP still works and didn't break when inserting Laboratory exercise 12 instructions. Every waveform from both photos match.

1.4 Testbench used

The testbench used in the previous section is as follows:

Code Block 1.1: Testench of the entire project

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

The code is just a modified version of the testbench from laboratory exercise 12. The notable change is we removed the **if(memwrite)** line as we'll be using multiple store bytes and words. **For each instruction**, we will be augmenting line 27 in order to set a stopping point for the program (since $dataadr \neq 84$ for most of the time and so is in writedata). Now, we are ready to introduce the new instructions.

Part 2

Implementing the sll instruction

2.0.1 HDL Code

sll works by taking in the operands b (rd2) and $shamt$ (instr[10:6]). The value in register ra1 (and ra1) is irrelevant, i.e., X. The operand b is shifted to the left by $shamt[4:0]$ or $instr[10:6]$. The following verilog files are modified to support **sll**. The format of the HDL code is first, I will be presenting the final verilog file, and then show the notable changes. Afterwards, a line by line explanation of the entire module is presented. Lastly, the schematic diagram of changes is presented and its corresponding assembly testbench and waveforms.

2.0.2 alu.sv

Code Block 2.1: Alu modification for sll

```
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  shamt,       // instr[10:6]
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;

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

  always_comb
    case (alucontrol[2:0])
     //Added third bit for SLL
      3'b000: result = a & b;
      3'b001: result = a | b;
      3'b010: result = sum;
      3'b011: result = sum[31];
      3'b100: result = b << shamt; // alucontrol 0100 for sll
      3'b110: result = (a >> ({1'b0, b[4:0]} + 6'b00001)) << ({1'b0, b[4:0]} + 6'b00001);  //zfr
    endcase
                //For beq instruction       a<=b
  assign zero = (alucontrol[2:0] == 3'b101) ?  ~($signed(b) < $signed(a)) : (result == 32'b0);
endmodule
```

The notable changes are (next page):

Line 2: ALU now takes in shamt, which is instruction[10:6]

Line 3, 13: alucontrol is now 3 bits to fit more operations.

Line9-10: Changed from alucontrol[2] $\rightarrow$ alucontrol[3] (MSB)

Line 19: The $alucontrol[2:0] = 3'b100$ is the signal for shift left. Explanation on the next page.

Line by Line Explanation of ALU.sv

L1-5 Starts the module definition. It takes in 32-bit operands a and b. Instr[10:6] is inputted as a 4-bit input $shamt$. The alucontrol was changed from $2 \rightarrow 3$ bits to accomodate more ALU operations. It outputs a 32-bit $result$ and a single-bit $zero$.

L7-10 Initializes two new 32-bit wires: $codinvb$ (which takes in the leftmost bit of $alucontrol$ and flips the sign if that bit is a 1) and sum (not changed from the original code).

L12-18 Is the ALU operation. It starts with the **always_comb** and starts a switch statements that takes in the three rightmost bits of the alucontrol. alucontrol[2:0] is just the same cases as the original SCP. 000 is for AND, 001 is for OR, 010 is for SUM, 011 is for set less than. The result is then set to $result$.

L19 Now, we have two new lines. If $alucontrol[2:0] = 100_2$, then it performs shift left. It takes in operand b and shifts left by $shamt$, which is $instr[10:6]$.

L20 The next new line handles the new custom instruction zfr. Basically, what happens is we take in operand a. Then, it shifts right by ($b[4:0]+1$). We concatenated an extra bit to account for the edge case $b[4:0] = 5'b11111$ (so it shifts all the way when the 1 is added rather than resetting to zero). After that, it performs the same style of shifting but this time, to the left. This essentially flushes the bits and we're left with ($b[4:0] + 1$) bits being set to zero from the right.

L23 The $zero$ assignment was also modified. Originally, it just checks if $result$ is zero. This time, it also checks for the $alucontrol[2:0] = 3'b101$, which is the $alucontrol$ code for the ble instruction. If the $alucontrol$ is indeed for ble, then it performs a signed comparison $a \leq b \implies \neg(b < a)$. If the ble condition passes, then $zero = 1$ which takes the branch. Otherwise, it checks if $result = 0$.

The schematic for the ALU is shown after the HDL edits.

2.0.3 datapath.sv

Code Block 2.2: The final datapath.sv

```
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, disableRA1, zeroextend,
7                  input  logic [3:0]  alucontrol, //Extra bit for sll
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);
13
14   logic [4:0]  writereg;
15   logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
16   logic [31:0] signimm, signimmsh;
17   logic [31:0] srca, srcb;
18   logic [31:0] result;
19
20   // next PC logic
21   flopr #(32) pcreg(clk, reset, pcnext, pc);
22   adder #(32) pcadd1(pc, 32'b100, 'b0, pcplus4); //So we adjust this to use the more complex adder; wmt-modificat
23   sl2         immsh(signimm, signimmsh);
24   adder #(32) pcadd2(pcplus4, signimmsh, 'b0, pcbranch); //See comment above
25   mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
26   mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
27                     instr[25:0], 2'b00}, jump, pcnext);
28
29   // register file logic
30   regfile     rf(clk, regwrite, disableRA1, instr[25:21], instr[20:16],
31                  writereg, result, srca, writedata);
32   mux2 #(5)   wrmux(instr[20:16], instr[15:11],
33                     regdst, writereg);
34   mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
35   signext     se(instr[15:0], zeroextend, signimm);
36
37   // ALU logic
38   mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
39   alu         alu(srca, srcb, instr[10:6], alucontrol, aluout, zero); //added shamt
40 endmodule
```

The only notable change here is the addition of the extra bit in line 7. Instead of $alucontrol[2:0]$, we now have $alucontrol[3:0]$ so we can accomodate new instructions such as **sll**.

A line by line explanation of datapath is shown on the next page.

Line by Line Explanation of datapath.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L3-12 Initializes the datapath. Most lines except line 6 are the same as in the original file. Line 6 has two new signals: disableRA1 and zeroextend. Line 7 is seen to have the same modifications as the the alu, with $alucontrol$ having a line width of 4 rather than 3 to account for more operations.

L14-18 Are extra wires. These are just the same with the original datapath.

L21-27 , **L32-34** and **L38** Again, nothing changed. These are just connections that connect the input and other components togther. These are instantiations of the modules defined in other .sv files.

L30-31 Is the instantiation of the register file (as "rf"). The only change here is we now insert $disableRA1$ as input.

L35 Is the initialization of the signext module as "se". We now insert the $zeroextend$ input from the module parameters into its corresponding port in the $signext$.

L39 Is the instantiation of the alu called "alu". $instr[10:6]$ is the $shamt[4:0]$ and $alucontrol$ has 4 bits as defined above.

2.0.4 aludec.sv

Code Block 2.3: The final aludec.sv

```
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); //Extra bit for sll
6
7    always_comb
8      case(aluop)
9        2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
10       2'b01: alucontrol <= 4'b1010;  // sub (for beq)
11       2'b11: alucontrol <= 4'b1101; //  BLE
12       default: case(funct)          // R-type instructions (aluop 10)
13                                     //alucontrol[2] is the SLL addition
14           6'b000000: alucontrol <= 4'b0100; // sll (third bit)
15           6'b100000: alucontrol <= 4'b0010; // add
16           6'b100010: alucontrol <= 4'b1010; // sub
17           6'b100100: alucontrol <= 4'b0000; // and
18           6'b100101: alucontrol <= 4'b0001; // or
19           6'b101010: alucontrol <= 4'b1011; // slt
20           6'b110011: alucontrol <= 4'b0110; // zfr
21           default:   alucontrol <= 4'bxxxx; // ???
22         endcase
23     endcase
24 endmodule
```

The only notable change here is the $alucontrol$ having an extra bit and line 14 where a new line in the siwtch statement (funct=000000) was added to accomodate the instruction.

Line by Line Explanation of aludec.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L3-5 Defines the module $aludec$. We have a 6-bit $funct$ code, a 2-bit $aluop$ input and a 4-bit (previously 3 bit) output $alucontrol$.

L7-11 Checks for $aluop$. Before checking for the function code, the decoder first checks the $aluop$ input (given by the maindec). Line 9-10 was not edited apart from the extra 0 on bit 2 of $alucontrol$. Line 11 is the BLE instruction. This inserts 1101 on the $alucontrol[3:0]$ input on the ALU.

L12-23 Are the cases for the function code $funct$.

- Line 14 is the new addition, with $funct = 000000$. The $alucontrol$ is 0100 which was decided in the ALU module.
- Line 15-19 are the original instructions in the Lab 12 SCP ($alucontrol[2:0]$ in the original SCP $\rightarrow$ $alucontrol[3]$ and $alucontrol[1:0]$ in the edited processor). FOr the added bit, $alucontrol[2] = 0$ since that bit is for the new instructions.
- Line 20 is the **zfr** instruction. The $funct$ code is given in the specifications PDF. We have $alucontrol = 0110$ which is fed into the ALU.
- Line 21 is the default case where an unknown $funct$ code gives an X in $alucontrol$.

2.1 Schematic(s)

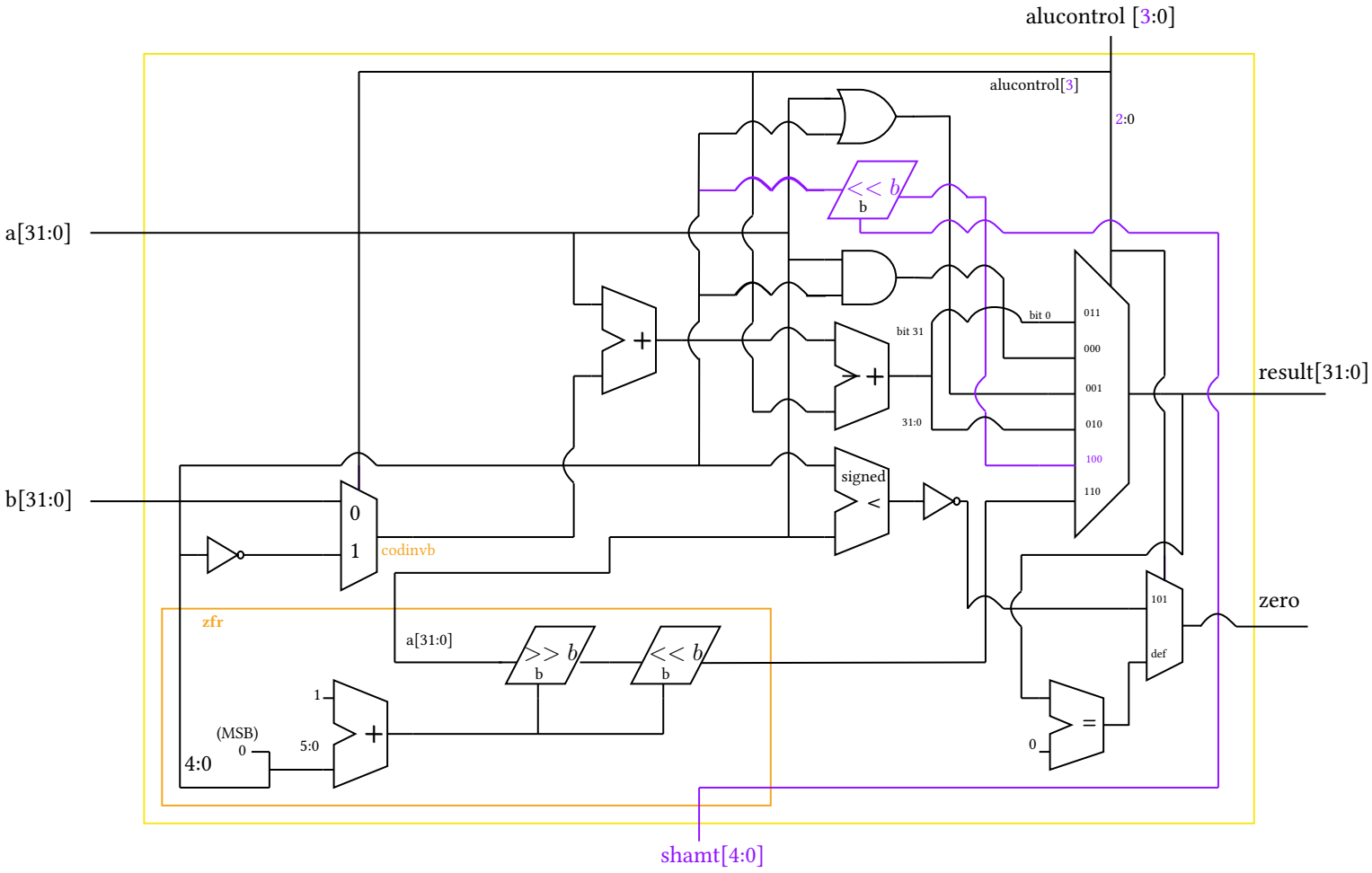

Figure 2.1: ALU Schematic

We take in two operands a and b. The **sll** instruction can be seen as highlighted in blue. Basically, shifter shifts the input (in this case, $b[31:0]$) b (not to be confused with the input $b[31:0]$) units to the left. The $alucontrol[2:0]$ code is 100, so if $alucontrol[2:0] = 100$, then the multiplexer selects that wire and propagates it to $result[31:0]$.

The changes outside the ALU is on the next page

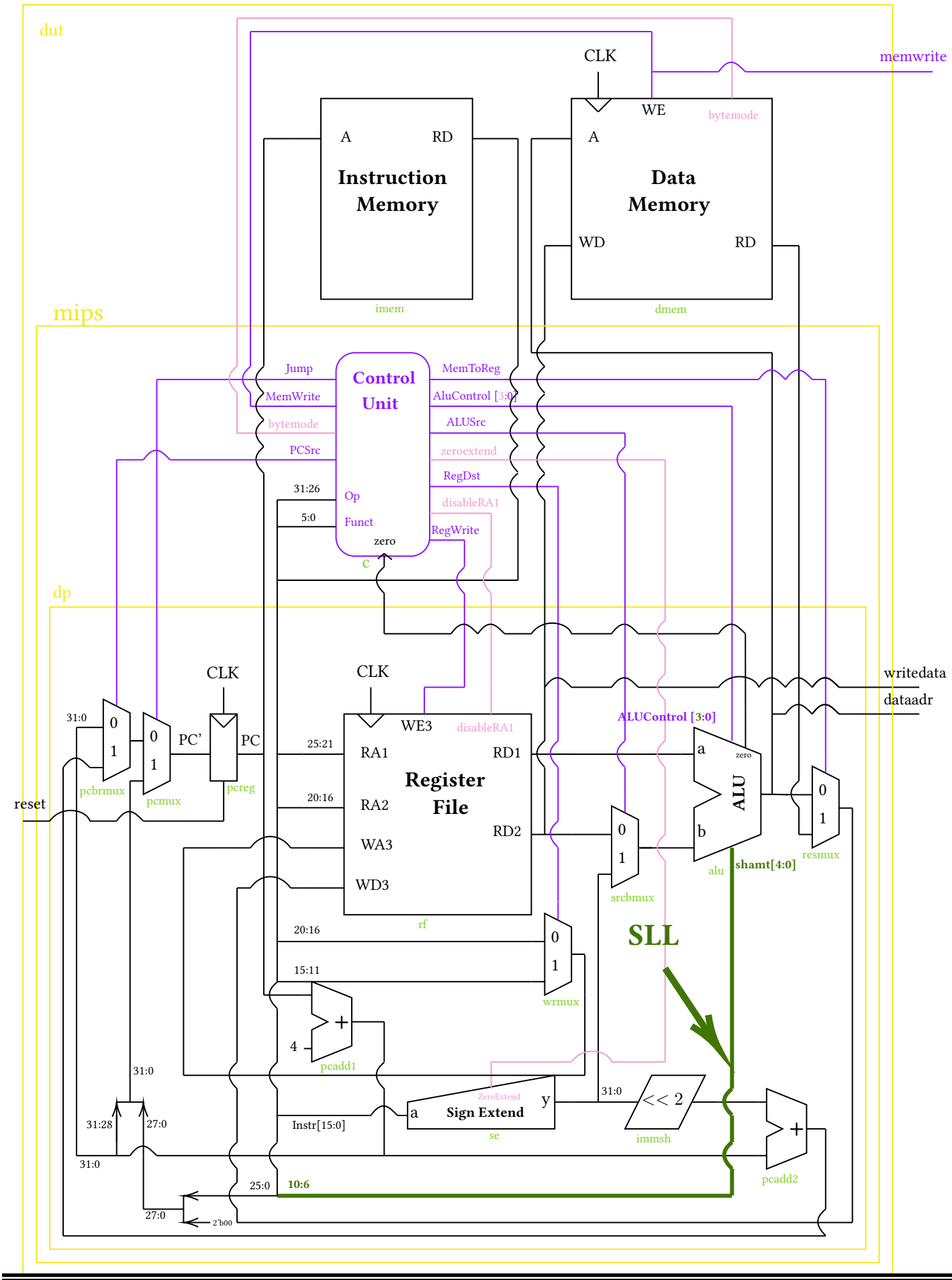

Figure 2.2: SLL Augmentation onto the entire processor

2.2 Assembly Code

A *very famliar* test case...

Code Block 2.4: Assembly code for sll

```
addi    $t4, $zero, 0x14
addi    $t2, $zero, 0x1
sll     $t2, $t2, 0x1
addi    $t2, $t2, 0x0030
sll     $t2, $t2, 0x0008
```

The memfile is as follows:

```
200c0014
200A0001
000a5040
214a0030
000a5200
```

We modify line 28 of the testbench to be

```
if(dataadr === 32'h00003200 & writedata === 32'h00000032) begin
```

2.3 Waveforms

Executing the testbench

Figure 2.3: Waveform for SLL testbench

In every intervals of 10 seconds...

10-30 ns Is the first instruction. It sets $t4 = 0 \times 14$ as seen in the dataadr ($0 \times 0 + 0 \times 14$ from the ALU signals). Take note that $writedata = XXXXXXXX$ for the first 10 ns since $rd2$ doesn't have a value yet (registers aren't necessarily 0 at the beginning of the execution). Program counter doesn't increment as $reset = 1$ for the first 22 ns but the first instruction was executed thrice (that's why we get $writedata = 0x14$ on the following waveforms).

30-40 ns We now set $t4 = 0 \times 0 + 0 \times 1 = 0 \times 1$. Thus, dataadr = 0×1 and $writedata = XXXXXXXX$.

40-50 ns We perform the first **sll**. $rd2 = 0 \times 1$ then we shift left by $shamt = 1$ (multiplying by 1) so we get $dataadr = result = 0 \times 2$.

50-60 ns Now we add t2 by 0×30, we have $dataadr = 0 \times 2 + 0 \times 30 = 0 \times 32$ as seen in the ALU result.

60-70 ns We have $writedata = 0 \times 32$ which is the value to be shifted. For the final bit shift, we have $dataadr = 0 \times 00000032 << (shamt = 2) = 0 \times 00003200$.

Therefore, **sll** instruction works as intended.

Part 3

Implementing the sb instruction

Now we implement the store byte instruction. The problem here is that memory access is always word-aligned (as you'll see later on dmem). This means that there is a challenge between on editing

3.1 Edited HDL code

3.1.1 maindec.sv

Code Block 3.1: sb: maindec.sv

```
1  `timescale 1ns / 1ps
2  module maindec(input  logic [5:0] op,
3                 output logic       memtoreg, memwrite,
4                 output logic       branch, alusrc,
5                 output logic       regdst, regwrite,
6                 output logic       jump, disableRA1, zeroextend, bytemode, //Sets ra1 to 0; zero extends rather t
7                 output logic [1:0] aluop);
8
9    logic [11:0] controls; //Extra bits for disableRA1, zeroextend and bytemode
10
11   assign {regwrite, regdst, alusrc, branch, memwrite,
12           memtoreg, jump, disableRA1, zeroextend, bytemode, aluop} = controls; //with Disable RA1
13
14   always_comb
15     case(op)
16       6'b000000: controls <= 12'b110000000010; // RTYPE
17       6'b100011: controls <= 12'b101001000000; // LW
18       6'b101011: controls <= 12'b001010000000; // SW
19       6'b000100: controls <= 12'b000100000001; // BEQ
20       6'b001000: controls <= 12'b101000000000; // ADDI
21       6'b000010: controls <= 12'b000000100000; // J
22       //              CUSTOM INSTRUCTIONS
23       6'b101000: controls <= 12'b001010000100; // SB
24       6'b011111: controls <= 12'b000100000011; // BLE (OP 1F, same controls)
25       6'b010001: controls <= 12'b101000011000; // li pseudoinstruction
26       default:   controls <= 12'bxxxxxxxxxxxx; // illegal op
27     endcase
28 endmodule
```

The changes are as follows: Line 6 and 12 now include the new output signal $bytemode$ that gets connected to the $dmem$ module. Line 23 contains the case for store byte, where $op = 101000$ with $alusrc = 1$, $memwrite = 1$ (similar to store word) and $bytemode = 1$. An explanation of maindec is shown on the next page.

Line by Line Explanation of maindec.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-7 is the initialization phase. Most didn't change. We have the 6-bit op code, some control signals $memtoreg$ to $jump$. We also added new output control signals disableRA1, zeroextend, and bytemode.

L9-12 Since we have 3 extra control signals, $controls$ is now a $12-bit$ signal rather than a $9-bit$ one. With that said, 3 bits before $aluop$ are the new control signals.

L14-27 Is the assignment of the $controls$ (output control signals) given the 6-bit Op code. For lines 16-21, these are the original instructions. However, for $controls[4:2]$ (the new signals), they are all 0. This ensures that the instruction still work the same.

- Line 23 is the store byte instruction ($op = 101000_2$). We have $alusrc = 1$ and $memwrite = 1$ (same as **sw**). This time, $controls[2] = bytemode = 1$. The rest is just zero.
- Line 24 ($Op = 011111_2$) is the **ble** or "branch if less than or equal to" instruction. We have $controls[8] = branch = 1$, which is similar to the **beq** instruction. This time, we set $aluop = 11$ which is a new custom case for $aluop$ that sends the appropriate ALU signal for beq.
- Line 25 handles the li pseudoinstruction (now an actual instruction). In here, $regwrite = 1$, $alusrc = 1$, $zeroextend = 1$ and $disableRA1 = 1$.
- Line 26 handles an unknown Op code, which just sets all $controls$ to X.

3.1.2 controller.sv

Code Block 3.2: sb: controller.sv

```
1  `timescale 1ns / 1ps
2  module controller(input  logic [5:0] op, funct,
3                    input  logic       zero,
4                    output logic       memtoreg, memwrite,
5                    output logic       pcsrc, alusrc,
6                    output logic       regdst, regwrite, disableRA1, zeroextend, bytemode,
7                    output logic       jump,
8                    output logic [3:0] alucontrol); //Added extra bit for sll
9
10   logic [1:0] aluop;
11   logic       branch;
12
13   maindec md(op, memtoreg, memwrite, branch,
14              alusrc, regdst, regwrite, jump, disableRA1, zeroextend, bytemode, aluop);
15   aludec  ad(funct, aluop, alucontrol);
16
17   assign pcsrc = branch & zero;
18 endmodule
```

As said in the previous section, the new signal bytemode was added as defined in maindec. It serves as an output wire of the module that will be connected to the $dmem$ module. The explanation of the controller is shown on the next page.

Line by Line Explanation of controller.sv

L1 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-8 Defines the controller module. For the most part but lines 6 and 8, the verilog code was unchanged. Line 6 now adds the three output signals: $disableRA1$, $zeroextend$ and $bytemode$. Line 6 is $alucontrol$ (4 bits instead of 3)

L10-11 Are just wires for $aluop$ and $branch$.

L13-14 Instantiates a maindec called "md" that takes in op, $memtoreg$, $memwrite$, $branch$, $alusrc$, $regdst$, $regwrite$, $jump$, and $aluop[1:0]$. It also contains the new signals.

L15 Instantiates a new $aludec$ called "ad" that takes in the function code $funct$, $aluop$, and the 4-bit $alucontrol$.

L17 Assigns the signal $pcsrc$ with $branch$ AND $zero$. That is, if $branch = 1$ and $zero = 1$, then $pcsrc = 1$. Else, $pcsrc = 0$.

3.1.3 mips.sv

Code Block 3.3: sb: mips.sv

```
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        memwrite, bytemode,
6              output logic [31:0] aluout, writedata,
7              input  logic [31:0] readdata);
8
9    logic       memtoreg, alusrc, regdst,
10             regwrite, jump, pcsrc, zero, disableRA1, zeroextend; //With disabling in register file
11   logic [3:0] alucontrol; //Edited to account for sll
12   controller c(instr[31:26], instr[5:0], zero,
13                memtoreg, memwrite, pcsrc,
14                alusrc, regdst, regwrite, disableRA1, zeroextend, bytemode, jump, //Custom instructions here
15                alucontrol);
16   datapath dp(clk, reset, memtoreg, pcsrc,
17               alusrc, regdst, regwrite, jump, disableRA1, zeroextend, //Custom instructions here
18               alucontrol,
19               zero, pc, instr,
20               aluout, writedata, readdata);
21 endmodule
```

Since we augmented the definition of the controller, we will also modify mips.sv. Again, the changes for sb is in pink. $bytemode$ is still the output because we're trying to connect it to the $dmem$ module. The explanation of the mips module is shown on the next page.

Line by Line Explanation of mips.sv

L1 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-7 Defines the mips module. Most of these are just from the traditional SCP files. The input and output files. The only change in here is the **bytemode** output. The output of MIPS is connected towards the data memory.

L9-11 Are the signals. The addition here is the $disableRA1$ and $zeroextend$. These lines are connected on lines 13 and 17. The $alucontrol$ is now 4 bits instead of 3 bits.

L12-15 Is the instantiation of the controller "c". The signals are the same with the SCP, but with the addition of $zeroextend$ and $bytemode$.

L16-20 Is the instantiation of the datapath "dp". The only addition is the $disableRA1$ and $zeroextend$.

3.1.4 top.sv

Code Block 3.4: sb: top.sv

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

As said earlier, the connection of top.sv is to connect the control signal to the $dmem$ module. So mips outputs the $bytemode$ signal and then it is the input to the $dmem$ module. The schematic containing top (and everything hierarchically below) in the schematics section.

Line by Line Explanation of top.sv

L1 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-4 Defines the *top* module. It takes in a 1-bit input the input clk and $reset$. It outputs a 32-bit $writedata$ (output of rd2), $dataadr$ (ALU result).

L6 Declares 32-bit wires pc and $readdata$.

L9-12 Instantiates $mips$, $imem$ and $dmem$. The necessary changes from the modules are implemented in these modules, with $mips$ now outputting $bytemode$ which is then connected to $dmem$. The other wires are connected to their corresponding modules (unchanged from the original SCP).

3.1.5 dmem.sv

Code Block 3.5: sb: dmem.sv

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

See lines 10-21 in the explanation below:

Line by Line Explanation of dmem.sv

L1 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-4 Defines the module $dmem$ that takes in 1-bit inputs clk, we (write enable), and $bytemode$. It also takes in a 32-bit input a (address) and wd (writedata). The output is a 32-bit rd (read data).

L6 Creates the RAM module. There are 64 32-bit words ($4 \times 64 = 256$ bytes).

L8 Is the same with the original SCL, where rd is assigned the $a[31:0]$ (we truncate bit $[1:0]$ because RAM accesses from $0, 1, 2, ..$ rather than $0, 4, 8, ..$ to be consistent with the register outputs. In short, to make it word-aligned.

L10-21 Is now the modified $dmem$ module. Line 10 starts the always block (triggering the following lines on the rising edge of clk). Line 11 checks if $we = 1$. If true, then it executes lines 12-21.

- If $bytemode = 0$, then it executes the usual writing from the original SCP (where $RAM[a[31:2]] = wd$).
- If $bytemode = 1$, then it configures the bytemode writing of $dmem$. It now checks for the last 2 bits of a (the offset). Take note that this process is in BIG ENDIAN.
  - If $a[1:0] = 00$, then there is no shifting. It modifies the leftmost byte (ex. if AA BB CC DD, then it modifies AA) with the final byte of wd (7:0).
  - If $a[1:0] = 01$, then it offsets by 1 and modifies BB.
  - If $a[1:0] = 10$, then it offsets by 2 and modifies CC.
  - If $a[1:0] = 11$, then it offsets by 3 and modifies DD.

  This means that the input a is no longer word-aligned and so, we can edit specific bytes rather than the entire word.

3.2 Schematic(s)

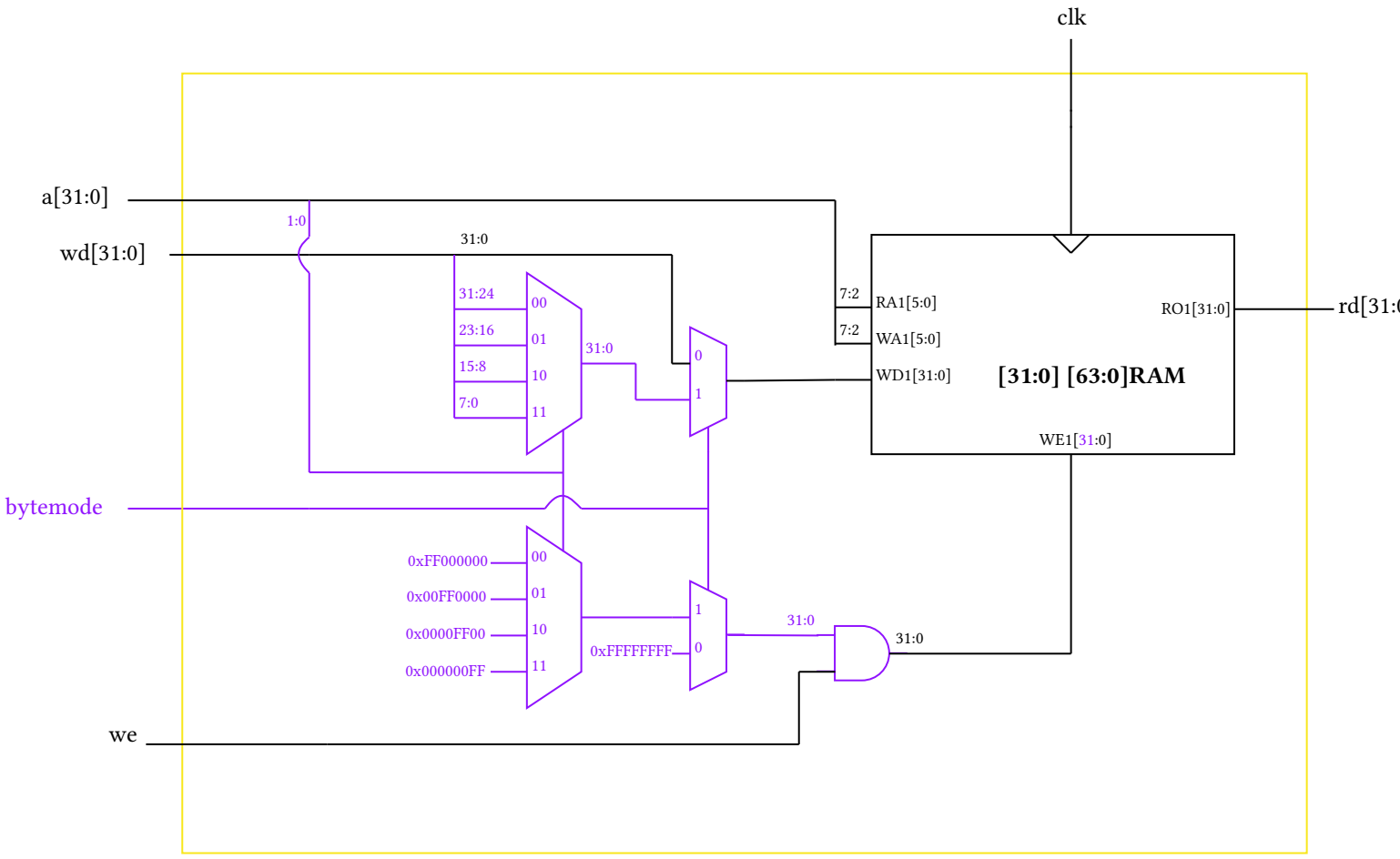

Figure 3.1: The new Dmem module

The changes are in blue. The $dmem$ module now has a **bytemode** input as defined in the HDL code. The address (rightmost two bits) is also connected to two new multiplexer. The top multiplexer handles the selection of bits to propagate to $[7 : 0]$. Then, it gets extended (other values are X's) to 32-bits which is then plugged into the two-way multiplexer. If $bytemode = 0$, then the multiplexer selects the original $wd[31 : 0]$. Otherwise, it selects the previous multiplexer's output. The bottom multiplexer handles the selection of which bits to write onto memory. Previously, the **RAM** module is just $we1$. This time, it is now a 32-bit input that selects **which bits to write** onto memory rather than the entire word. If $bytemode = 0$, then it just selects all bits to be written (Lab 12 writing). Otherwise, it selects the specific byte to be written (as defined by $a[1 : 0]$). The and gate takes in a 32-bit input from multiplexer lower right. If $we = 1$, then the signal gets propagated onto $WE1[31 : 0]$ (*verilog is weird*).

On the next page is the top.sv augmentation (the connection from $mips \rightarrow dmem$).

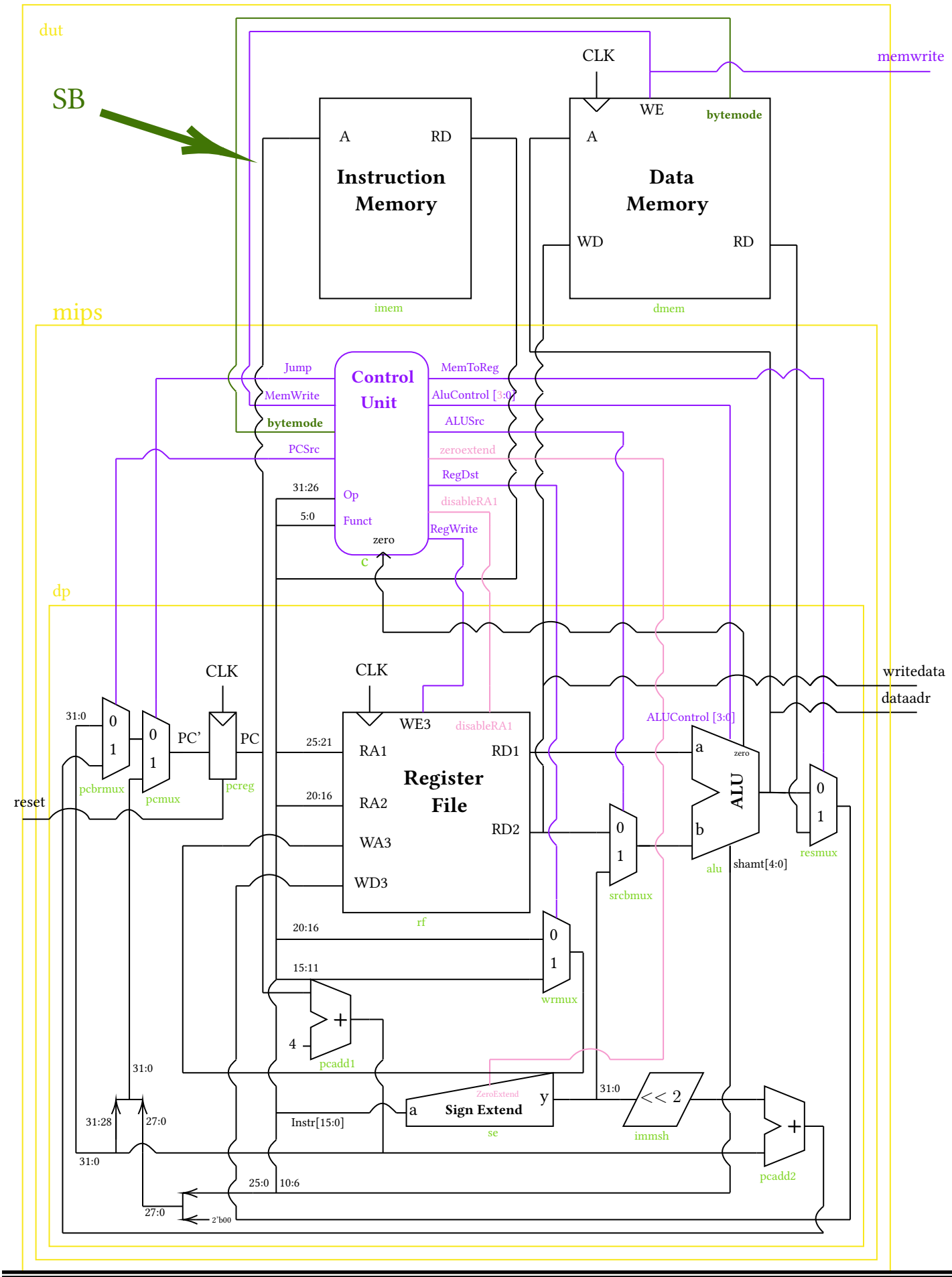

Figure 3.2: SB Augmentation onto the entire processor

3.3 Assembly Code

Code Block 3.6: Tester for sb

```
1  #Test for uninitialized RAM
2  addi    $t0, $0, 0x5     #Memory address
3  addi    $t1, $0, 0xBABE  #Loads 0xFFFFBABE
4  sb      $t1, 1($t0)      #Saves BE at address 6
5  lw      $t2, 0($t0)      #Should get t2=0xXXXXBEXX
6  addi    $t2, $0, 0x0     #See value of t2 in waveform
7  #Testing for store and load word
8  addi    $t1, $t1, 0x1    #t1=FFFFBABF
9  sw      $t1, 0($t0)      #Should write at the same word as be
10 lw      $t2, 0($t0)      #Should get t2=0xFFFFBABF
11 addi    $t2, $t2, 0x0    #See value of t2 in waveform
12 #Memory Augmentation
13 addi    $t1, $0, 0xC0    #t1 = 0xC0
14 addi    $t0, $t0, 0xFFFF #t0 = 5 - 1 = 4
15 sb      $t1, 0($t0)      #Word should be 0xC0FFBABF
16 addi    $t1, $0, 0xDE    #t1 = 0xDE
17 sb      $t1, 1($t0)      #Word should be 0xC0DEBABF
18 lw      $t2, 0($t0)      #t2 = 0xC0DEBABF
19 addi    $t2, $t2, 0x0    #See value of t2 in waveform
```

The memfile is as follows:

```
20080005
2009BABE
A1090001
8D0A0000
214A0000
21290001
AD090000
8D0A0000
214A0000
200900C0
2108FFFF
A1090000
200900DE
A1090001
8D0A0000
214A0000
```

For the testbench, we set line 28 to

```
if(dataadr === 32'hc0debabf & writedata === 32'hc0debabf) begin
```

3.4 Waveforms

Figure 3.3: Waveform of sb test case

This is a little lengthy so we'll summarize it. The first instruction is 10-30 ns. The waveform now also contain RAM[1] which is the word in memory address $0x4$. Every register and memory data is X so $rd2 = XXXXXXXX$ (when it was not yet used) and $RAM[1] = XXXXXXXX$. Also, $shamt$ gets random values. This is fine since the ALU doesn't care about instr[10:6] unless **sll** was called.

1. **20-30 ns** It loads t0=0x5 (read and write address).
2. **The following lines in intervals of 10 ns**. It loads t1=0xFFFFBABE (sign-extended from addi).
3. It stores 0xFFFFBABE onto memory address 6 (in **big endian**). That's why we see **XXXXbeXX** rather than **XXbeXXXX** in RAM[1].
4. Just shows the value in $writedata$. Dataadr is X since we don't know if the addition would carry and effect the value **be** (in this case, it doesn't but vivado outputs an X instead).
5. Adds 1 so we get FFFFBABF
6. We store the ENTIRE word t2 in memory. RAM[1] updates AFTER the instruction.
7. We load the word onto memory. Just like with the **lw** after **sb**, we have $writedata = XXXXXXXX$ since there is a don't care value.
8. Adding a zero, the changes to t2 are now reflected and will no longer be X when loaded.
9. Now loads t1=C0.
10. Subtracts t0 by 1 (adding 0xFFFF) so the result is now address 4 (MSB of the word).

11. Store $C0$ byte on memory address 4 (MSB). We should have $0 \times C0FFBABF$ as seen in RAM[1]
12. It loads t1=DE.
13. It stores DE on memory address $0 \times 4 + 0 \times 1 = 0 \times 5$, where we get $0 \times C0DEBABF$
14. It now loads the load from memory (address 0×4 to 0×7).
15. Adds a zero to reflect the changes in $writedata$ and $dataadr$.

The behavior matches with the assembly code explanation above. Therefore **sb works as intended**.

Part 4

Implementing the li instruction

This instruction is **different** from **addi $reg, $0, imm**. As seen in the previous part, $addi$ is sign-extended. That's why when we load $0 \times BABE$ in the register, we get $0 \times FFFFBABE$ as $BABE[15] = 1$. For the **li** instruction, it **zero-extends** the immediate value.

4.1 Edited HDL code

4.1.1 maindec.sv

To accomodate zero-extend, we define a new control signal **zeroextend**, as seen in the maindec.sv:

Code Block 4.1: li: maindec.sv

```
1  `timescale 1ns / 1ps
2  module maindec(input  logic [5:0] op,
3                 output logic       memtoreg, memwrite,
4                 output logic       branch, alusrc,
5                 output logic       regdst, regwrite,
6                 output logic       jump, disableRA1, zeroextend, bytemode, //Sets ra1 to 0; zero extends rather
7                 output logic [1:0] aluop);
8
9    logic [11:0] controls; //Extra bits for disableRA1, zeroextend and bytemode
10
11   assign {regwrite, regdst, alusrc, branch, memwrite,
12           memtoreg, jump, disableRA1, zeroextend, bytemode, aluop} = controls; //with Disable RA1
13
14   always_comb
15     case(op)
16       6'b000000: controls <= 12'b110000000010; // RTYPE
17       6'b100011: controls <= 12'b101001000000; // LW
18       6'b101011: controls <= 12'b001010000000; // SW
19       6'b000100: controls <= 12'b000100000001; // BEQ
20       6'b001000: controls <= 12'b101000000000; // ADDI
21       6'b000010: controls <= 12'b000000100000; // J
22       //              CUSTOM INSTRUCTIONS
23       6'b101000: controls <= 12'b001010000100; // store byte instruction
24       6'b011111: controls <= 12'b000100000011; // BLE (OP 1F, same controls)
25       6'b010001: controls <= 12'b101000011000; // li pseudoinstruction
26       default:   controls <= 12'bxxxxxxxxxxxx; // illegal op
27     endcase
28 endmodule
```

For **li**, the OP code is $op = 010001_2$. Lines 2 and 12 pertain to the new signal added. Line 25 checks if the op code is **li**'s. If yes, then $regwrite = 1$, $alusrc = 1$ (imm), $disableRA1 = 1$ and $zeroextend = 1$ (with the rest of the signals being zero). $aluop[1:0] = 00$ since we're adding. **Line by line on the next page.**

Line by Line Explanation of maindec.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-7 is the initialization phase. Most didn't change. We have the 6-bit op code, some control signals $memtoreg$ to $jump$. We also added new output control signals disableRA1, zeroextend, and bytemode.

L9-12 Since we have 3 extra control signals, $controls$ is now a $12-bit$ signal rather than a $9-bit$ one. With that said, 3 bits before $aluop$ are the new control signals.

L14-27 Is the assignment of the $controls$ (output control signals) given the 6-bit Op code. For lines 16-21, these are the original instructions. However, for $controls[4:2]$ (the new signals), they are all 0. This ensures that the instruction still work the same.

- Line 23 is the store byte instruction ($op = 101000_2$). We have $alusrc = 1$ and $memwrite = 1$ (same as **sw**). This time, $controls[2] = bytemode = 1$. The rest is just zero.
- Line 24 ($Op = 011111_2$) is the **ble** or "branch if less than or equal to" instruction. We have $controls[8] = branch = 1$, which is similar to the **beq** instruction. This time, we set $aluop = 11$ which is a new custom case for $aluop$ that sends the appropriate ALU signal for beq.
- Line 25 handles the li pseudoinstruction (now an actual instruction). In here, $regwrite = 1$, $alusrc = 1$, $zeroextend = 1$ and $disableRA1 = 1$.
- Line 26 handles an unknown Op code, which just sets all $controls$ to X.

4.1.2 controller.sv

Code Block 4.2: li: controller.sv

```
1  `timescale 1ns / 1ps
2  module controller(input  logic [5:0] op, funct,
3                    input  logic       zero,
4                    output logic       memtoreg, memwrite,
5                    output logic       pcsrc, alusrc,
6                    output logic       regdst, regwrite, disableRA1, zeroextend, bytemode,
7                    output logic       jump,
8                    output logic [3:0] alucontrol); //Added extra bit for sll
9
10   logic [1:0] aluop;
11   logic       branch;
12
13   maindec md(op, memtoreg, memwrite, branch,
14              alusrc, regdst, regwrite, jump, disableRA1, zeroextend, bytemode, aluop);
15   aludec  ad(funct, aluop, alucontrol);
16
17   assign pcsrc = branch & zero;
18 endmodule
```

This part only talks about the new control signals that will go through the datapath. Since maindec is inside controller, the signal must pass outside the controller before it goes through the datapath.

Line by Line Explanation of controller.sv

L1 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-8 Defines the controller module. For the most part but lines 6 and 8, the verilog code was unchanged. Line 6 now adds the three output signals: $disableRA1$, $zeroextend$ and $bytemode$. Line 6 is $alucontrol$ (4 bits instead of 3)

L10-11 Are just wires for $aluop$ and $branch$.

L13-14 Instantiates a maindec called "md" that takes in op, $memtoreg$, $memwrite$, $branch$, $alusrc$, $regdst$, $regwrite$, $jump$, and $aluop[1:0]$. It also contains the new signals.

L15 Instantiates a new $aludec$ called "ad" that takes in the function code $funct$, $aluop$, and the 4-bit $alucontrol$.

L17 Assigns the signal $pcsrc$ with $branch$ AND $zero$. That is, if $branch = 1$ and $zero = 1$, then $pcsrc = 1$. Else, $pcsrc = 0$.

4.1.3 signext.sv

Code Block 4.3: li: signext.sv

```
1 module signext(input  logic [15:0] a,
2                input  logic        zeroextend, //Signs or zero extends for li instruction
3                output logic [31:0] y);
4   assign y = zeroextend ? 16'b0, a : {{16{a[15]}}, a};
5 endmodule
```

This should be enough:

Line by Line Explanation of signext.sv

L1-3 is the initialization phase. Lines 1 and 3 are the same, taking in a 16-bit input a and outputs a 32-bit output y. However, we added a new input called zeroextend that changes $signext \rightarrow zeroext$.

L4 is the logic. If $zeroextend = 1$, then it performs a zero extend by concatenating a 16-bit zero and a. Otherwise, it a gets concatenated by the 16-bit extension of the sign bit of a.

4.1.4 regfile.sv

Code Block 4.4: li: regfile.sv

```
1  // regfile.v
2  // Register file for the single-cycle and multicycle processors
3
4  module regfile(input  logic        clk,
5                 input  logic        we3, disableRA1, //Disable RA1 for Li
6                 input  logic [4:0]  ra1, ra2, wa3,
7                 input  logic [31:0] wd3,
8                 output logic [31:0] rd1, rd2);
9
10   logic [31:0] rf[31:0];
11
12   // three ported register file
13   // read two ports combinationally
14   // write third port on rising edge of clk
15   // register 0 hardwired to 0
16   // note: for pipelined processor, write on
17   // falling edge of clk
18
19   always_ff (posedge clk) //Should be @(posedge clk); latex issues
20     if (we3) rf[wa3] <= wd3;
21     ^^I //more latex issues; ignore
22
23     assign rd1 = (ra1 == 0 | disableRA1) ? 32'b0 : rf[ra1]; //Setter for disable RA1
24     assign rd2 = (ra2 != 0) ? rf[ra2] : 32'b0;
25 endmodule
```

Basically, $disableRA1$ sets $rd1 = 0$ when $disableRA1 = 1$. This is the same as accessing the zero register. This means that for **li**, $ra1 = XXXXX$ and $rd1 = 0x00000000$.

Line by Line Explanation of regfile.sv

L4-8 Initializes the register file, taking in 1-bit inputs clk, $we3$ and disableRA1 (new input that sets RD1 = 0; this is the only change). It also takes in three 5-bit inputs $ra1$, $ra2$, $wa3$, and a 32-bit input $wd3$. The 32-bit outputs are $rd1$ and $rd2$.

L10 Is just the same initialization, 32 32-bit registers as the original file.

L19-21 Was done without modifications. Basically if $we3 = 1$, then it writes $wd3$ to register $wa3$.

L23 Was changed. Instead of just checking if $ra1 = 0$, it also checks for $disableRA1$ (for use with the **li** pseudoinstruction). With that said, if either of them are true, then we set $rd1 = 0$. Otherwise, we take in the value of $rf[ra1]$.

L24 Is still the same. $rd2$ is set to its appropriate register value if $ra2 \neq 0$. Otherwise, it sets $ra2 = 0$ if we're accessing register 0.

4.1.5 datapath.sv

Code Block 4.5: li: datapath.sv

```
////////////
`timescale 1ns / 1ps
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump, disableRA1, zeroextend,
                input  logic [3:0]  alucontrol, //Extra bit for sll
                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) pcadd1(pc, 32'b100, 'b0, pcplus4); //So we adjust this to use the more complex adder; wmt-modificat
  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, disableRA1, 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], zeroextend, signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc, srcb);
  alu         alu(srca, srcb, instr[10:6], alucontrol, aluout, zero); //added shamt
endmodule
```

The main change here for **li** is that it takes in the new signals $disableRA1$ and $zeroextend$. This connects to rf and se, respectively.

A line by line explanation of datapath is on the next page.

Line by Line Explanation of datapath.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L3-12 Initializes the datapath. Most lines except line 6 are the same as in the original file. Line 6 has two new signals: disableRA1 and zeroextend. Line 7 is seen to have the same modifications as the the alu, with $alucontrol$ having a line width of 4 rather than 3 to account for more operations.

L14-18 Are extra wires. These are just the same with the original datapath.

L21-27 , **L32-34** and **L38** Again, nothing changed. These are just connections that connect the input and other components togther. These are instantiations of the modules defined in other .sv files.

L30-31 Is the instantiation of the register file (as "rf"). The only change here is we now insert $disableRA1$ as input.

L35 Is the initialization of the signext module as "se". We now insert the $zeroextend$ input from the module parameters into its corresponding port in the $signext$.

L39 Is the instantiation of the alu called "alu". $instr[10:6]$ is the $shamt[4:0]$ and $alucontrol$ has 4 bits as defined above.

4.1.6 mips.sv

Code Block 4.6: li: mips.sv

```
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        memwrite, bytemode,
6              output logic [31:0] aluout, writedata,
7              input  logic [31:0] readdata);
8
9    logic       memtoreg, alusrc, regdst,
10             regwrite, jump, pcsrc, zero, disableRA1, zeroextend; //With disabling in register file
11   logic [3:0] alucontrol; //Edited to account for sll
12   controller c(instr[31:26], instr[5:0], zero,
13                memtoreg, memwrite, pcsrc,
14                alusrc, regdst, regwrite, disableRA1, zeroextend, bytemode, jump, //Custom instructions here
15                alucontrol);
16   datapath dp(clk, reset, memtoreg, pcsrc,
17               alusrc, regdst, regwrite, jump, disableRA1, zeroextend, //Custom instructions here
18               alucontrol,
19               zero, pc, instr,
20               aluout, writedata, readdata);
21 endmodule
```

Again, connections! The new signals $disableRA1$ and $zeroextend$ (defining the wire on line 10) are connected from controller (output) to datapath (input).

A line by line explanation of mips.sv is on the next page.

Line by Line Explanation of mips.sv

L1 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-7 Defines the mips module. Most of these are just from the traditional SCP files. The input and output files. The only change in here is the **bytemode** output. The output of MIPS is connected towards the data memory.

L9-11 Are the signals. The addition here is the $disableRA1$ and $zeroextend$. These lines are connected on lines 13 and 17. The $alucontrol$ is now 4 bits instead of 3 bits.

L12-15 Is the instantiation of the controller "c". The signals are the same with the SCP, but with the addition of $zeroextend$ and $bytemode$.

L16-20 Is the instantiation of the datapath "dp". The only addition is the $disableRA1$ and $zeroextend$.

4.2 Schematic(s)

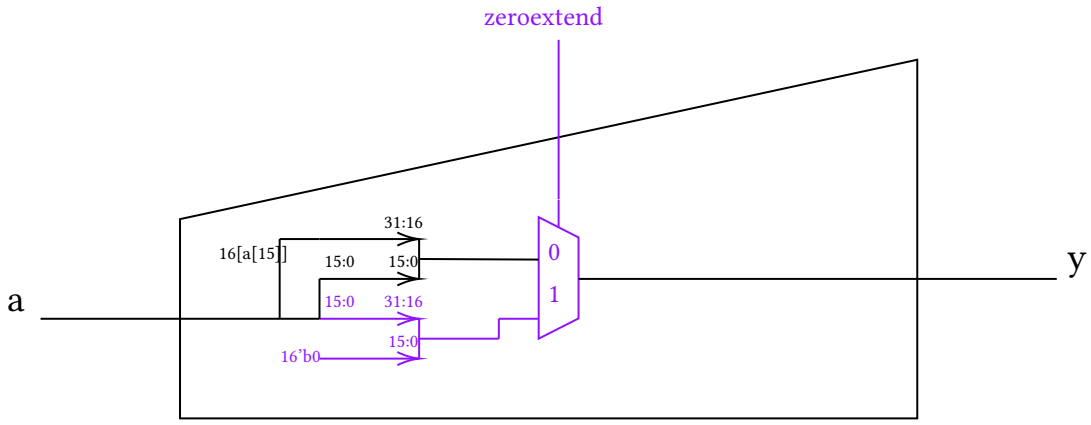

Figure 4.1: The new signextend module

We now have a multiplexer that switches between the sign extended and the zero extended input a.

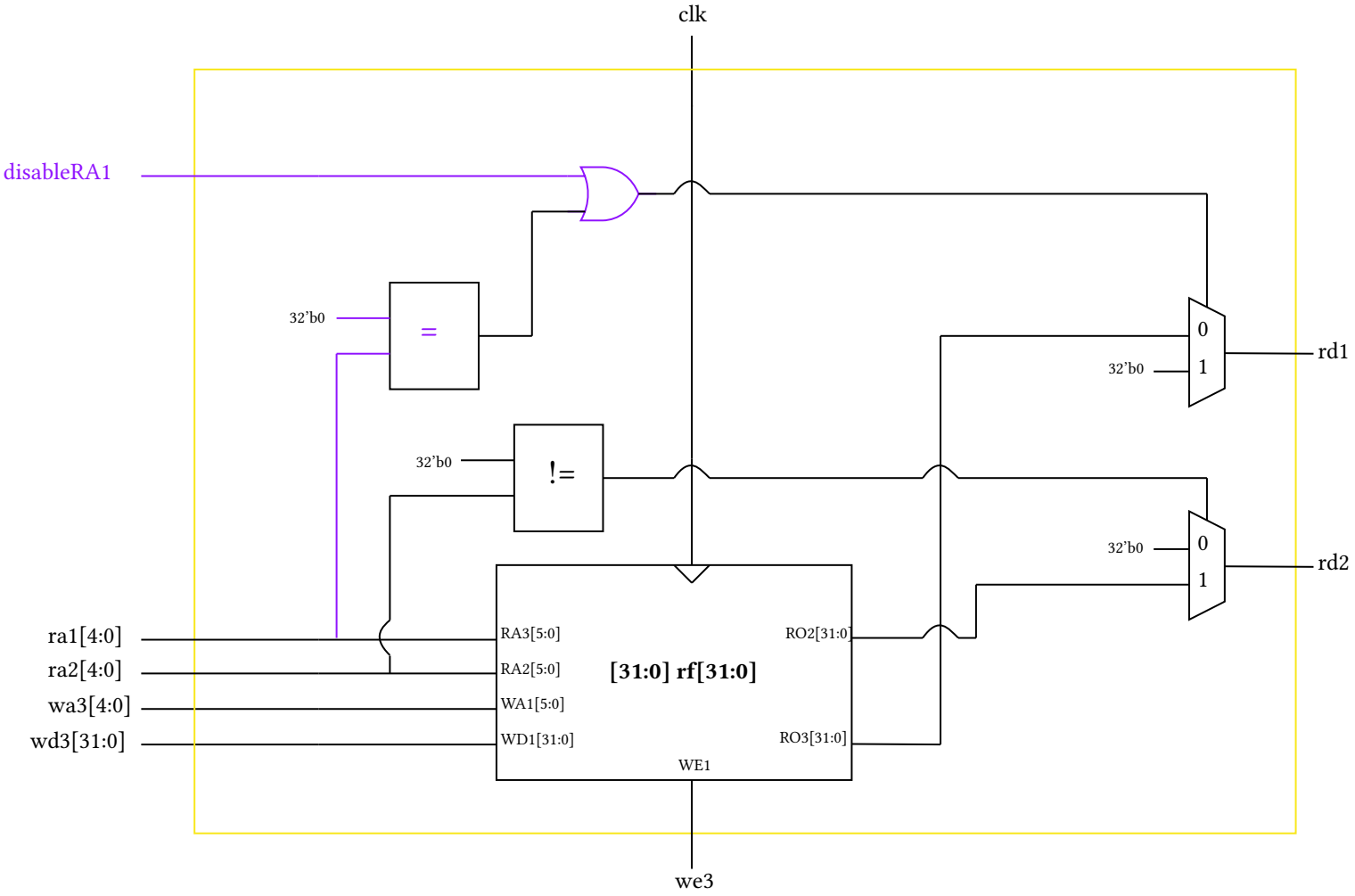

Figure 4.2: The new signextend module

The difference here is now we have a new signal disableRA1.
On the next page is the top.sv augmentation.

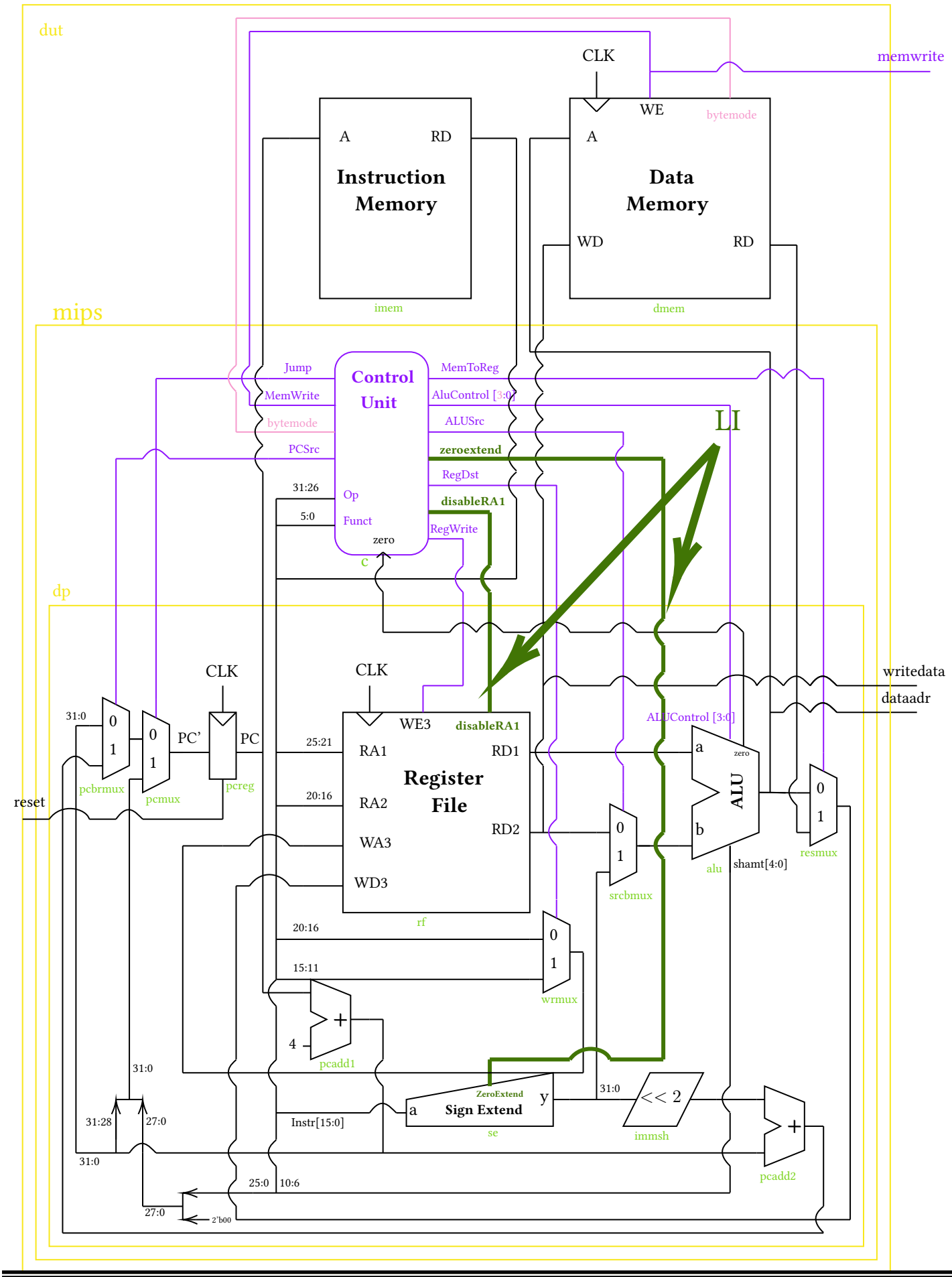

Figure 4.3: LI Augmentation onto the entire processor

4.3 Assembly Code

Code Block 4.7: Test case for li

```
addi    $t0, $zero, 0xBBAA   # Loads -17494
li      $t1, 0xBBAA             # Loads 48042
sll     $t1, $t1, 16         # t1 = 0xBBAA0000
sub     $t3, $t1, $t0    # t3= 0xBBAA0000 - 0xFFFFBBAA
addi    $t0, $t0, 0x0        # Show register values of three registers in
addi    $t1, $t1, 0x0        # dataadr and writedata
addi    $t3, $t3, 0x0
```

The machine code is as follows:

```
2008BBAA
4409BBAA
00094C00
01285822
21080000
21290000
216B0000
```

Modifying line 28 as

```
if(dataadr === 32'hbbaa4456 & writedata === 32'hbbaa4456) begin
```

4.4 Waveforms

Executing in vivado, we have

Figure 4.4: Waveform for li test case

The main difference between **li** and **addi with register 0** is that addi sign extends, i.e. we get $0\times FFFFBBAA$ (0-30 ns) while **li** zero-extends, i.e. we get $0 \times 0000BBAA$ (30-40 ns). The code goes as follows, where $0 \times 0000BBAA$ is shifted 16 times which we get $0 \times BBAA0000$. Waveform 50-60 ns performs subtraction where $0 \times BBAA0000 - 0 \times FFFFBBAA = 0 \times BBAA0000 + 0 \times 00004456 = 0 \times BBAA4456$.

Therefore, li works as intended.

Part 5

Implementing the ble instruction

This is the **branch if less than or equal** intruction. It works the same as **beq**, but the condition is now different.

5.1 Edited HDL code

5.1.1 maindec.sv

Code Block 5.1: ble: maindec.sv

```
1  `timescale 1ns / 1ps
2  module maindec(input  logic [5:0] op,
3                 output logic       memtoreg, memwrite,
4                 output logic       branch, alusrc,
5                 output logic       regdst, regwrite,
6                 output logic       jump, disableRA1, zeroextend, bytemode, //Sets ra1 to 0; zero extends rather th
7                 output logic [1:0] aluop);
8
9    logic [11:0] controls; //Extra bits for disableRA1, zeroextend and bytemode
10
11   assign {regwrite, regdst, alusrc, branch, memwrite,
12           memtoreg, jump, disableRA1, zeroextend, bytemode, aluop} = controls; //with Disable RA1
13
14   always_comb
15     case(op)
16       6'b000000: controls <= 12'b110000000010; // RTYPE
17       6'b100011: controls <= 12'b101001000000; // LW
18       6'b101011: controls <= 12'b001010000000; // SW
19       6'b000100: controls <= 12'b000100000001; // BEQ
20       6'b001000: controls <= 12'b101000000000; // ADDI
21       6'b000010: controls <= 12'b000000100000; // J
22       //              CUSTOM INSTRUCTIONS
23       6'b101000: controls <= 12'b001010000100; // store byte instruction
24       6'b011111: controls <= 12'b000100000011; // BLE (OP 1F, same controls)
25       6'b010001: controls <= 12'b101000011000; // li pseudoinstruction
26       default:   controls <= 12'bxxxxxxxxxxxx; // illegal op
27     endcase
28 endmodule
```

The highlight here is just a single line. Line 24 handles **ble** with $op = 011111$. It has almost the same signals as beq, with $branch = 1$, $aluop = 11$ (custom ALU code so this instruction can communicate with the ALU Decoder). A line by line explanation is on the next page.

Line by Line Explanation of maindec.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L2-7 is the initialization phase. Most didn't change. We have the 6-bit op code, some control signals $memtoreg$ to $jump$. We also added new output control signals disableRA1, zeroextend, and bytemode.

L9-12 Since we have 3 extra control signals, $controls$ is now a $12-bit$ signal rather than a $9-bit$ one. With that said, 3 bits before $aluop$ are the new control signals.

L14-27 Is the assignment of the $controls$ (output control signals) given the 6-bit Op code. For lines 16-21, these are the original instructions. However, for $controls[4:2]$ (the new signals), they are all 0. This ensures that the instruction still work the same.

- Line 23 is the store byte instruction ($op = 101000_2$). We have $alusrc = 1$ and $memwrite = 1$ (same as **sw**). This time, $controls[2] = bytemode = 1$. The rest is just zero.
- Line 24 ($Op = 011111_2$) is the **ble** or "branch if less than or equal to" instruction. We have $controls[8] = branch = 1$, which is similar to the **beq** instruction. This time, we set $aluop = 11$ which is a new custom case for $aluop$ that sends the appropriate ALU signal for beq.
- Line 25 handles the li pseudoinstruction (now an actual instruction). In here, $regwrite = 1$, $alusrc = 1$, $zeroextend = 1$ and $disableRA1 = 1$.
- Line 26 handles an unknown Op code, which just sets all $controls$ to X.

5.1.2 aludec.sv

Code Block 5.2: ble: aludec.sv

```
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); //Extra bit for sll
6
7    always_comb
8      case(aluop)
9        2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
10       2'b01: alucontrol <= 4'b1010;  // sub (for beq)
11       2'b11: alucontrol <= 4'b1101;  //  BLE
12       default: case(funct)          // R-type instructions (aluop 10)
13                                     //alucontrol[2] is the SLL addition
14           6'b000000: alucontrol <= 4'b0100; // sll (third bit)
15           6'b100000: alucontrol <= 4'b0010; // add
16           6'b100010: alucontrol <= 4'b1010; // sub
17           6'b100100: alucontrol <= 4'b0000; // and
18           6'b100101: alucontrol <= 4'b0001; // or
19           6'b101010: alucontrol <= 4'b1011; // slt
20           6'b110011: alucontrol <= 4'b0110; // zfr
21           default:   alucontrol <= 4'bxxxx; // ???
22         endcase
23     endcase
24 endmodule
```

In BLE, we have a new control unit. Later in the ALU, we will see the branching. We didn't add any new control signals or modify the datapath. **Again**, the line by line explanation is on the next page.

Line by Line Explanation of aludec.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L3-5 Defines the module $aludec$. We have a 6-bit $funct$ code, a 2-bit $aluop$ input and a 4-bit (previously 3 bit) output $alucontrol$.

L7-11 Checks for $aluop$. Before checking for the function code, the decoder first checks the $aluop$ input (given by the maindec). Line 9-10 was not edited apart from the extra 0 on bit 2 of $alucontrol$. Line 11 is the BLE instruction. This inserts 1101 on the $alucontrol[3:0]$ input on the ALU.

L12-23 Are the cases for the function code $funct$.

- Line 14 is the new addition, with $funct = 000000$. The $alucontrol$ is 0100 which was decided in the ALU module.
- Line 15-19 are the original instructions in the Lab 12 SCP ($alucontrol[2:0]$ in the original SCP $\rightarrow$ $alucontrol[3]$ and $alucontrol[1:0]$ in the edited processor). FOr the added bit, $alucontrol[2] = 0$ since that bit is for the new instructions.
- Line 20 is the **zfr** instruction. The $funct$ code is given in the specifications PDF. We have $alucontrol = 0110$ which is fed into the ALU.
- Line 21 is the default case where an unknown $funct$ code gives an X in $alucontrol$.

5.1.3 alu.sv

Code Block 5.3: ble: ALU

```
1  module alu(input  logic [31:0] a, b,
2             input  logic [4:0]  shamt,       // instr[10:6]
3             input  logic [3:0]  alucontrol,
4             output logic [31:0] result,
5             output logic        zero);
6
7    logic [31:0] condinvb, sum;
8
9    assign condinvb = alucontrol[3] ? ~b : b;
10   assign sum = a + condinvb + alucontrol[3];
11
12   always_comb
13     case (alucontrol[2:0])
14      //Added third bit for SLL
15       3'b000: result = a & b;
16       3'b001: result = a | b;
17       3'b010: result = sum;
18       3'b011: result = sum[31];
19       3'b100: result =  b << shamt; // alucontrol 0100
20       3'b110: result = (a >> ({1'b0, b[4:0]} + 6'b00001)) << ({1'b0, b[4:0]} + 6'b00001);  //zfr
21     endcase
22                   //For beq instruction        a<=b
23   assign zero = (alucontrol[2:0] == 3'b101) ?  ~($signed(b) < $signed(a)) : (result == 32'b0);
24 endmodule
```

In the ALU, we have modified line 23. The ALU signal for **ble** is 101. Then, it performs a **signed** comparison between a and b. If $\neg(b < a) \implies b \geq a$, then $zero = 1$. The signed operation is important otherwise the comparator would compare unsigned bits. That is, ex., $1001_2 > 0101_2$ which is wrong under 2C. If $alucontrol[2:0]$ is not 101, then it just compares if $result == 0$. **A line by line explanation of alu.sv is on the next page.**

Line by Line Explanation of ALU.sv

L1-5 Starts the module definition. It takes in 32-bit operands a and b. Instr[10:6] is inputted as a 4-bit input $shamt$. The alucontrol was changed from $2 \rightarrow 3$ bits to accomodate more ALU operations. It outputs a 32-bit $result$ and a single-bit $zero$.

L7-10 Initializes two new 32-bit wires: $codinvb$ (which takes in the leftmost bit of $alucontrol$ and flips the sign if that bit is a 1) and sum (not changed from the original code).

L12-18 Is the ALU operation. It starts with the **always_comb** and starts a switch statements that takes in the three rightmost bits of the alucontrol. alucontrol[2:0] is just the same cases as the original SCP. 000 is for AND, 001 is for OR, 010 is for SUM, 011 is for set less than. The result is then set to $result$.

L19 Now, we have two new lines. If $alucontrol[2:0] = 100_2$, then it performs shift left. It takes in operand b and shifts left by $shamt$, which is $instr[10:6]$.

L20 The next new line handles the new custom instruction zfr. Basically, what happens is we take in operand a. Then, it shifts right by $(b[4:0]+1)$. We concatenated an extra bit to account for the edge case $b[4:0] = 5'b11111$ (so it shifts all the way when the 1 is added rather than resetting to zero). After that, it performs the same style of shifting but this time, to the left. This essentially flushes the bits and we're left with $(b[4:0]+1)$ bits being set to zero from the right.

L23 The $zero$ assignment was also modified. Originally, it just checks if $result$ is zero. This time, it also checks for the $alucontrol[2:0] = 3'b101$, which is the $alucontrol$ code for the ble instruction. If the $alucontrol$ is indeed for ble, then it performs a signed comparison $a \leq b \implies \neg(b < a)$. If the ble condition passes, then $zero = 1$ which takes the branch. Otherwise, it checks if $result = 0$.

The other lines are the same since it just functions like beq.

5.2 Schematic(s)

The only major change would be the ALU. The maindec and aludec modules only require an addition case, which doesn't affect the circuit diagram of the module (internally done, even if you look at it on vivado's elaborated design).

With that said, here is the ALU.

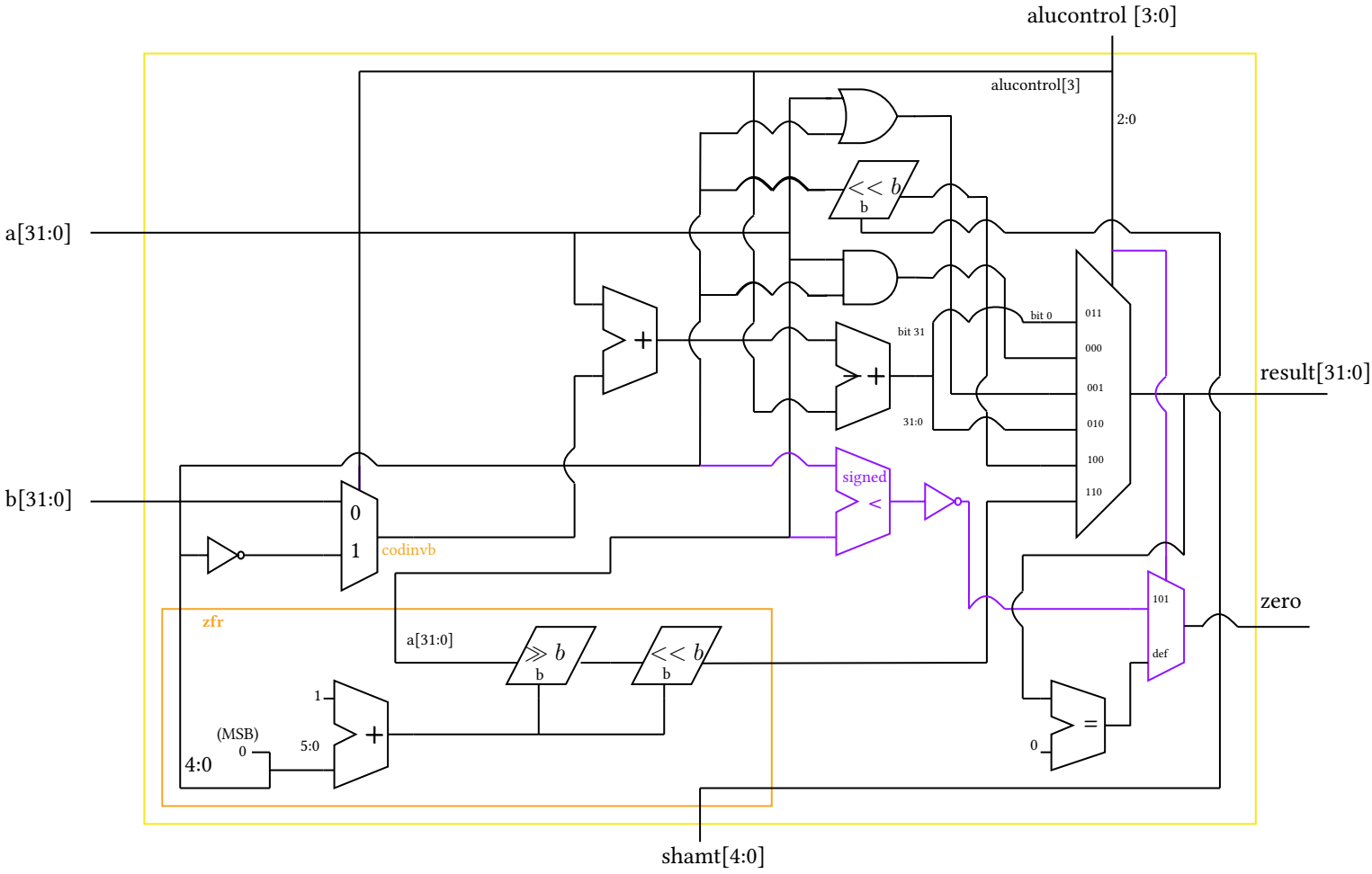

Figure 5.1: BLE Augmentation in the ALU

The edit here is we have a new comparator (the path is in blue). The NOT gate negates the signal then it is fed into the mux. The mux decides if $alucontrol[2:0]$ is 101. Otherwise, it just propagates the original comparator for $result == 0$.

Take note that my design does NOT use the ALU subtraction. It feeds directly from the input and onto the multiplexer. So expect different waveforms than a typical beq.

5.3 Assembly Code

Code Block 5.4: ble: Test Case

```
addi $t0, $zero, 0x69 # 105
addi $t1, $zero, 0xBBAA # -17494
addi $t2, $zero, 0xB0B0 # -20304
addi $t3, $zero, 0x420 # 1056
ble $t0, $t3, 3 #105 <= 1056; take
addi $v0, $zero 0x0
addi $v0, $zero 0x0
addi $v0, $zero 0x0
ble $t3, $t1, 3 #1056 <= -17494; don't take
addi $v0, $zero 0x0
addi $v0, $zero 0x0
addi $v0, $zero 0x0
ble $t2, $t3, 3 #-20304 <= -17494; take
addi $v0, $zero 0x0
addi $v0, $zero 0x0
addi $v0, $zero 0x0
ble $t2, $t0, 3 #-20304 <= 105; take
addi $v0, $zero 0x0
addi $v0, $zero 0x0
addi $v0, $zero 0x0
ble $t3, $t3, 3 #1056 <= 1056; take
addi $v0, $zero 0x0
addi $v0, $zero 0x0
addi $v0, $zero 0x0
addi $v0, $zero, 0xBABE #Last execution when branch is taken
```

```
20080069
2009BBAA
200AB0B0
200B0420
7d0b0003
20020000
20020000
20020000
7d690003
20020000
20020000
20020000
7d4b0003
20020000
20020000
20020000
7d480003
20020000
20020000
20020000
7d6b0003
20020000
20020000
20020000
2002BABE
```

Do not forget to change condition

```
if(dataadr === 32'hffffbabe & writedata === 32'h00000000) begin
```

5.4 Waveforms

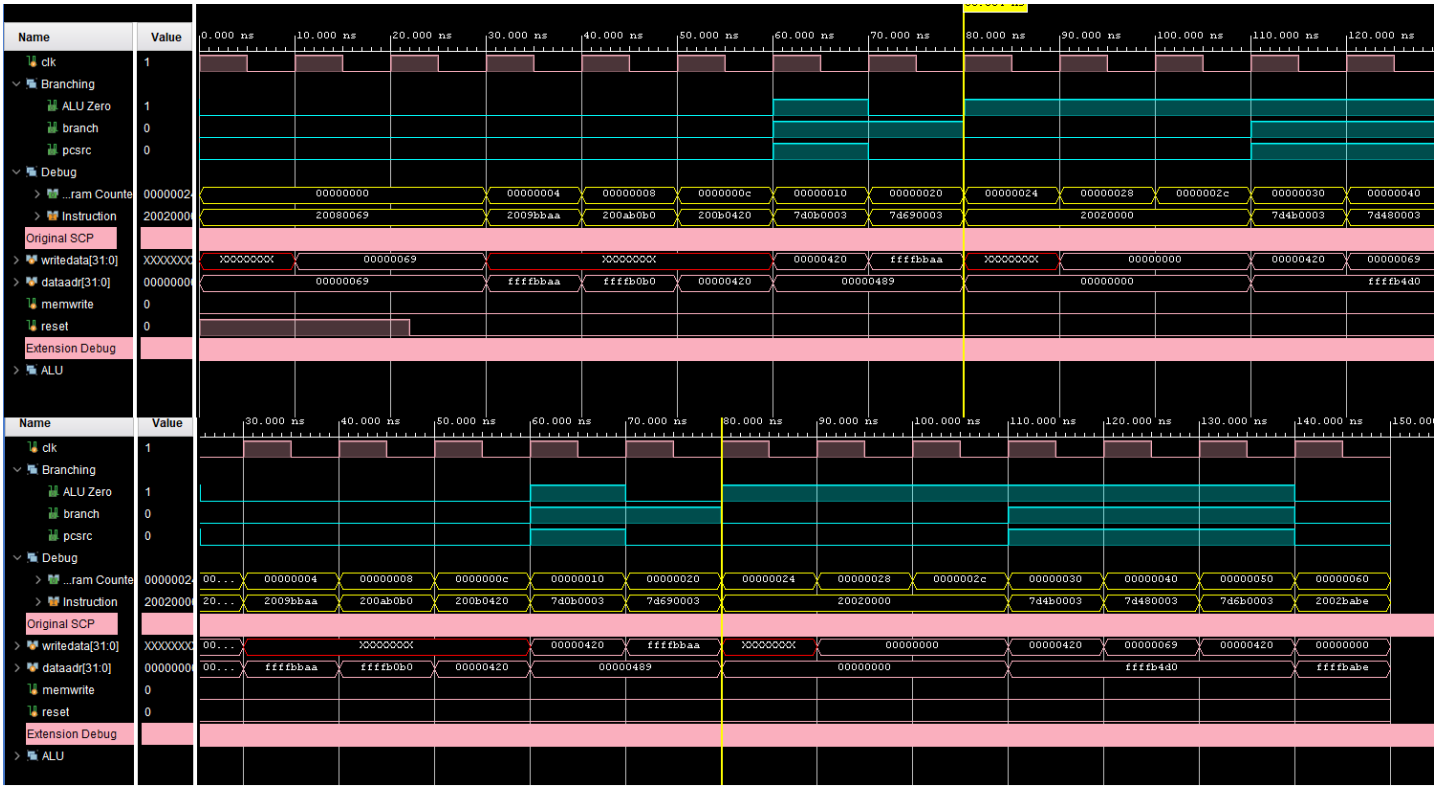

Figure 5.2: BLE Test case Waveform

0-60 ns loads the values onto the specific registers. To see that a branch has been taken, we need to look at the **program counter** waveform and **pcsrc**. If a branch has been taken, program counter increments by 0×10. The first branch occurs at 60-70 ns, which was taken ($pc = 0 \times 10 \rightarrow 0 \times 20$). The second branch is at 70-80 ns, which was **NOT** taken as expected in the assembly code. So, the filler instructions were called thrice until the next branch at 110-120 ns. These three branches are taken so it jumps to the final instruction at 140 - 150 ns. Take note that these test cases are meant to hit the edge cases of ble. The beq instruction still works as intended as seen on the integrity testbench.

Therefore, **ble works as intended** (although, observe that the waveforms **may be different** as we didn't use the ALU on the instruction as the comparator feeds directly from the input operands of the alu).

Part 6

Implementing the zfr instruction

6.1 Edited HDL code

6.1.1 aludec.sv

Code Block 6.1: zfr: aludec.sv

```
////////////
`timescale 1ns / 1ps
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [3:0] alucontrol); //Extra bit for sll

  always_comb
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      2'b01: alucontrol <= 4'b1010;  // sub (for beq)
      2'b11: alucontrol <= 4'b1101; //  BLE
      default: case(funct)          // R-type instructions (aluop 10)
                                   //alucontrol[2] is the SLL addition
          6'b000000: alucontrol <= 4'b0100; // sll (third bit)
          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'b0110;  // zfr
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

The **only** change outside the ALU is the ALU decoder. For zfr, we have $funct = 110011_2$. If this is true, then $alucontrol[3 : 0] = 0110_2$. This is a custom ALU case which will be discussed after the line by line explanation on the next page.

Line by Line Explanation of aludec.sv

L2 Initializes the timescale to 1 ns. So, 1 (1 time unit) is 1 ns.

L3-5 Defines the module $aludec$. We have a 6-bit $funct$ code, a 2-bit $aluop$ input and a 4-bit (previously 3 bit) output $alucontrol$.

L7-11 Checks for $aluop$. Before checking for the function code, the decoder first checks the $aluop$ input (given by the maindec). Line 9-10 was not edited apart from the extra 0 on bit 2 of $alucontrol$. Line 11 is the BLE instruction. This inserts 1101 on the $alucontrol[3:0]$ input on the ALU.

L12-23 Are the cases for the function code $funct$.

- Line 14 is the new addition, with $funct = 000000$. The $alucontrol$ is 0100 which was decided in the ALU module.
- Line 15-19 are the original instructions in the Lab 12 SCP ($alucontrol[2:0]$ in the original SCP $\rightarrow$ $alucontrol[3]$ and $alucontrol[1:0]$ in the edited processor). FOr the added bit, $alucontrol[2] = 0$ since that bit is for the new instructions.
- Line 20 is the **zfr** instruction. The $funct$ code is given in the specifications PDF. We have $alucontrol = 0110$ which is fed into the ALU.
- Line 21 is the default case where an unknown $funct$ code gives an X in $alucontrol$.

6.1.2 alu.sv

Code Block 6.2: zfr: alu.sv

```
1  module alu(input  logic [31:0] a, b,
2             input  logic [4:0]  shamt,       // instr[10:6]
3             input  logic [3:0]  alucontrol,
4             output logic [31:0] result,
5             output logic        zero);
6
7    logic [31:0] condinvb, sum;
8
9    assign condinvb = alucontrol[3] ? ~b : b;
10   assign sum = a + condinvb + alucontrol[3];
11
12   always_comb
13     case (alucontrol[2:0])
14      //Added third bit for SLL
15       3'b000: result = a & b;
16       3'b001: result = a | b;
17       3'b010: result = sum;
18       3'b011: result = sum[31];
19       3'b100: result =  b << shamt; // alucontrol 0100
20       3'b110: result = (a >> ({1'b0, b[4:0]} + 6'b00001)) << ({1'b0, b[4:0]} + 6'b00001);  //zfr
21     endcase
22                  //For beq instruction        a<=b
23   assign zero = (alucontrol[2:0] == 3'b101) ?  ~($signed(b) < $signed(a)) : (result == 32'b0);
24 endmodule
```

One line! Basically, it shifts right by $b[4:0]+1$ and then shifts left by the same amount. A 0 was concatenated on the addition to account for overflow ($b[4:0] = 11111$). The line-by-line explanation is on the next page

Line by Line Explanation of ALU.sv

L1-5 Starts the module definition. It takes in 32-bit operands a and b. Instr[10:6] is inputted as a 4-bit input $shamt$. The alucontrol was changed from $2 \rightarrow 3$ bits to accomodate more ALU operations. It outputs a 32-bit $result$ and a single-bit $zero$.

L7-10 Initializes two new 32-bit wires: $codinvb$ (which takes in the leftmost bit of $alucontrol$ and flips the sign if that bit is a 1) and sum (not changed from the original code).

L12-18 Is the ALU operation. It starts with the **always_comb** and starts a switch statements that takes in the three rightmost bits of the alucontrol. alucontrol[2:0] is just the same cases as the original SCP. 000 is for AND, 001 is for OR, 010 is for SUM, 011 is for set less than. The result is then set to $result$.

L19 Now, we have two new lines. If $alucontrol[2:0] = 100_2$, then it performs shift left. It takes in operand b and shifts left by $shamt$, which is $instr[10:6]$.

L20 The next new line handles the new custom instruction zfr. Basically, what happens is we take in operand a. Then, it shifts right by $(b[4:0]+1)$. We concatenated an extra bit to account for the edge case $b[4:0] = 5'b11111$ (so it shifts all the way when the 1 is added rather than resetting to zero). After that, it performs the same style of shifting but this time, to the left. This essentially flushes the bits and we're left with $(b[4:0]+1)$ bits being set to zero from the right.

L23 The $zero$ assignment was also modified. Originally, it just checks if $result$ is zero. This time, it also checks for the $alucontrol[2:0] = 3'b101$, which is the $alucontrol$ code for the ble instruction. If the $alucontrol$ is indeed for ble, then it performs a signed comparison $a \leq b \implies \neg(b < a)$. If the ble condition passes, then $zero = 1$ which takes the branch. Otherwise, it checks if $result = 0$.

6.2 Schematic(s)

ALU decoder is no longer edited to add extra lines and circuits (just a new switch case but not seen on Vivado). For the ALU, it goes as follows, highlighted in blue:

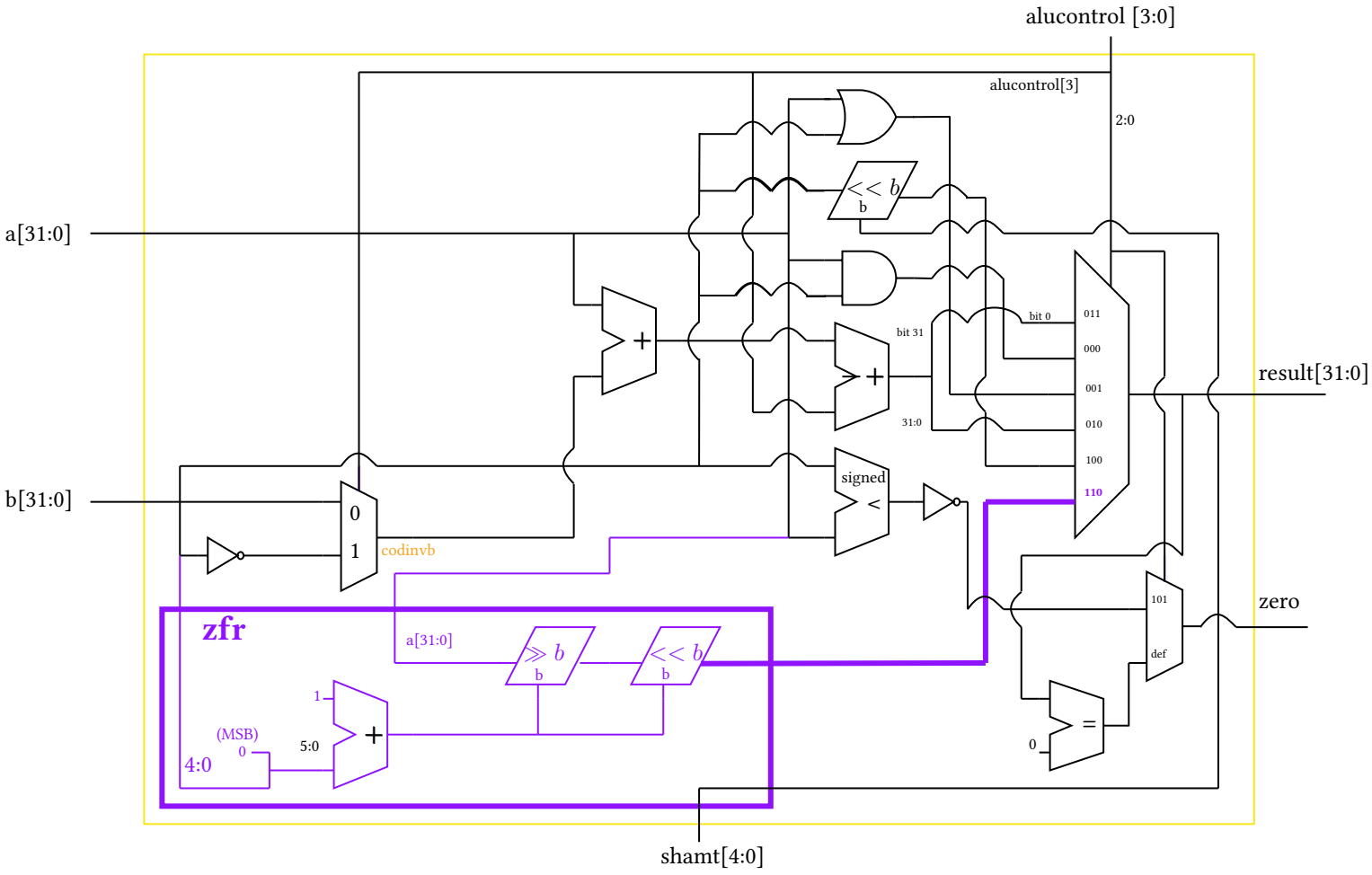

Figure 6.1: ZFR Augmentation in the ALU

It starts with the input operand $a[31:0]$ which is the value to be shifted. It also takes in $b[4:0]$ (the number of times to shift + 1). The concatenation of a rightmost 0 was also done after it. We have an adder that adds 1 on $\{1'b0, b[4:0]\}$. This value gets plugged on both bit shifters.

Goiung back to signal $a[31:0]$, it enters the first bit shifter that shifts right by $b = \{1'b0, b[4:0]\} + 1$ and then shifts left by the same amount. The output is plugged onto the multiplexer with $alucontrol[2:0] = 110$. That is, if $alucontrol[2:0] = 110$, then the result of the ZFR instruction gets propagated as $result[31:0]$.

6.3 Assembly Code

Code Block 6.3: zfr test case

```
addi    $t0, $zero, 0xFFFF
li  $t1, 0x6969 #t1 =
sll     $t1, $t1, 16 #t1 = 0x69690000
li  $at, 0xFFFF #at = 0xFFFFFFFF
add     $t1, $t1, $at #t1 = 6969FFFF #zfr operand a
addi    $t2, $zero, 0x5 #zfr operand B
addi    $t3, $zero, 0xF005 #Sign extends; zfr operand b
#ZFR
zfr     $t4, $t0, $t2 # 0xFFFFFFFF zfr by 0x5
addi    $t4, $zero, 0x0
zfr     $t4, $t0, $t0 #0xFFFFFFFF zfr 0xFFFFFFFF
addi    $t4, $zero, 0x0
zfr     $t4, $t0, $t3 #zfr by 0xFFFFF005; should be the same as the first zfr
addi    $t5, $zero, 0xB0B0
```

The memfile is as follows:

```
2008ffff
44096969
00094c00
4401ffff
01214820
200a0005
200bf005
010a6033
200c0000
01086033
200c0000
010b6033
200db0b0
```

We modify the testbench condition to

```
if(dataadr === 32'h00000000 & writedata === 32'hfffffffc0) begin
```

6.4 Waveforms

Figure 6.2: ZFR Waveform

1. 0-90 ns loads the appropriate values as defined in the assembly code (lines 1-7 of code block 6.3).
2. The first ZFR is $0 \times FFFFFFFF$ by 0×5, which (from the project PDF), we get $0 \times FFFFFFC0$ as seen in $dataadr$. Then, it flushes the result by adding \$t4 and \$zero.
3. The second ZFR is $0 \times FFFFFFFF$ by itself, Since $b[4:0] = 11111$, all 32 bits are set to 0 so we get $dataadr = 0 \times 00000000$ (110-120 ns). Then, it flushes the result on 120-130 ns.
4. The third ZFR now test for *don't care values*. We have $0 \times FFFFF005$ but $b[4:0]$ is still 0×5. So, we get the same result as the first one with $dataadr = 0 \times FFFFFFC0$. Then it stores the value $0 \times FFFFB0B0$ on register \$t5 (where we get an X because it was initially X's; untouched).

Therefore, **zfr** instruction works as intended!