CS 21 MP 2 Documentation

Gabriel Aldrich S. Calubayan
CS 21 Lab 2
Submitted to: Mr. Jerome Beltran, Ms. Jozelle Addawe

Documentation Outline

This Machine Problem 2 documentation shall be divided into three subsections:

I. Project Introduction
II. Preliminary Changes
III. `xori` Changes
IV. `lui` Changes
V. `srlv` Changes
VI. `bgtz` Changes
VII. `li` Changes

I. Project Introduction

In this project, we are required to revise the MIPS single-cycle processor (see Figure 1) so that the instruction set that it can execute will be extended to the following instructions. However, my Machine Problem 2 will not be implementing runxor due to time constraints.

- `xori`
- `lui`
- `srlv`
- `bgtz`
- `li`
- `runxor` (not implemented)

The implementation of the revised MIPS single-cycle processor (which I will now refer to as "processor" in the interest of brevity) will be done using SystemVerilog in Vivado 2022.2, as done in previous CS 21 laboratory exercises in the second semester of A.Y. 2022-2023.

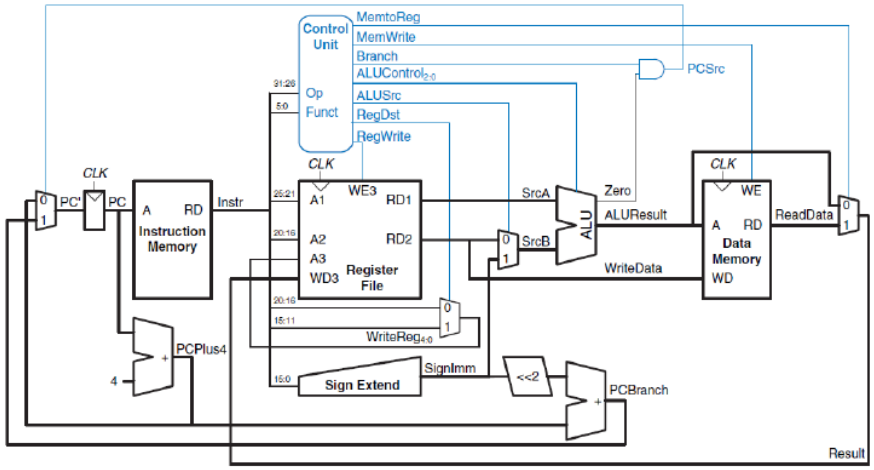


Figure 1. Single-cycle processor (from CS 21 Lecture 9 lecture slides)

Thus, I used the following testbench module (from Laboratory Exercise 12) to test the new instructions in the processor:

```
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     // instantiate device to be tested
10    top dut(clk, reset, writedata, dataadr, memwrite);
11    // initialize test
12    initial
13      begin
14        reset <= 1; # 22; reset <= 0;
15      end
16
17    // generate clock to sequence tests
18    always
19      begin
20        clk <= 1; # 5; clk <= 0; # 5;
21      end
22
23    // check results
24    always @(negedge clk)
25      begin
26        if(memwrite) begin
27          if(dataadr === 'h10 & writedata === 'h0) begin
28            #5;
29            $stop;
30          end
31        end
32      end
33  endmodule
```

Code Block 0. MIPS single-cycle processor testbench module

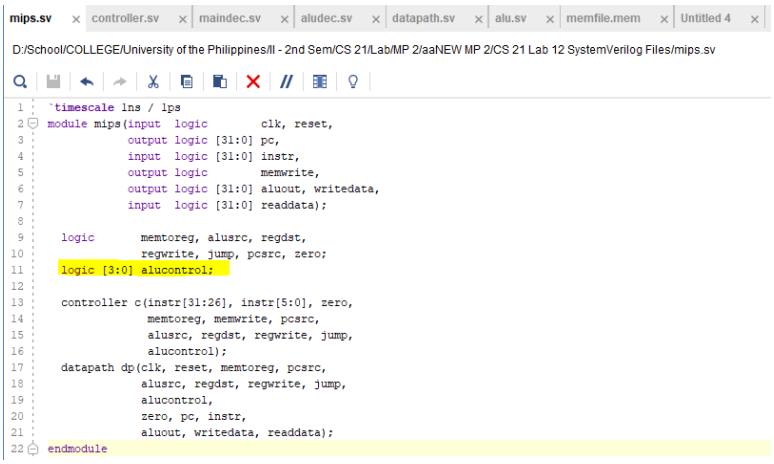
Below is a line-by-line explanation of Code Block 1:

- Line 1: sets the delay unit and delay resolution. This means that a single delay will take 1 ns, and 1 ps will pass per time tick.
- Line 2: module name
- Line 3-7: instantiation of the clock signal, reset signal, 32-bit writedata and dataadr signals, and memwrite signal
- Line 10: instantiation of the `top` module with the `clk` and `reset` signals as input, and `writedata`, `dataadr`, and `memwrite` signals as output
- Line 12-15: sets the `reset` signal to 1, waits 22 delay units, and sets `reset` signal to zero. While `reset` is 1, the program counter will not advance.
- Lines 18-21: oscillates the clock signal between one and zero every 5 delay units

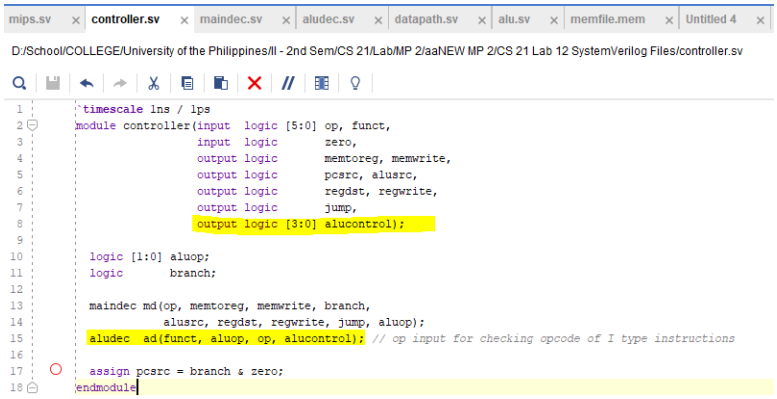
- Line 24: start of an *always begin* block that executes the lines of code every negative edge of the clock (when the `clk` signal is 0)
- Line 25: start of the block of code that gets executed every negative edge of the clock
- Line 26: checks if the `memwrite` signal is 1; if so, it will execute the code block below it
- Line 27-30: checks if the hex value of the dataadr signal is 0x10, and writedata is 0x0. If so, it waits 5 delay units and stops execution
- Lines 31-33: end lines that end the code blocks above

II. Preliminary changes

The first thing I did was to "extend" the bits of the ALUControl signal from three to four in mips.sv (Line 11 in Code Block 1), controller.sv (Line 8 in Code Block 2), datapath.sv (Line 7 in Code Block 3), and in alu.sv (Line 2 in Code Block 6). This is to accommodate for more ALU operations.



Code Block 1. mips.sv with extended 4-bit ALUcontrol signal



Code Block 2. controller.sv with extended 4-bit ALUcontrol signal and new opcode input to the ALU decoder

mips.sv | controller.sv | maindec.sv | aludec.sv | datapath.sv | alu.sv | memfile.mem | Untitled 4

D:/School/COLLEGE/University of the Philippines/II - 2nd Sem/CS 21/Lab/MP 2/aaNEW MP 2/CS 21 Lab 12 SystemVerilog Files/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,
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);
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;
```

Code Block 3. Snippet of datapath.sv with extended 4-bit ALUcontrol signal

In Code Block 2, I connected the 6-bit `op` opcode signal (Line 15) to the `aludec` module (Line 15). This is so that the ALU decoder can check the opcode of instructions and accommodate other I-type instructions.

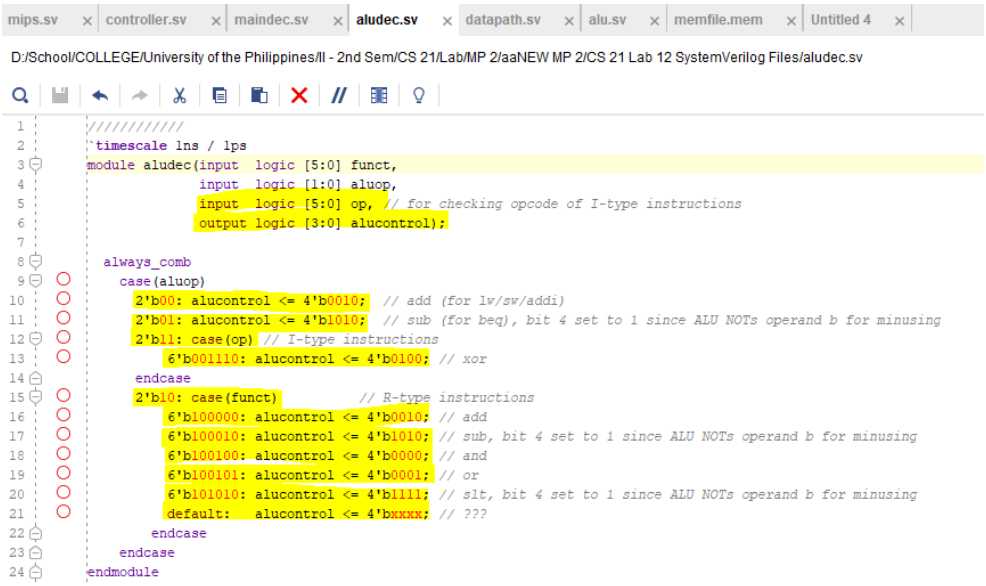
mips.sv | controller.sv | maindec.sv | aludec.sv | datapath.sv | alu.sv | memfile.mem | Untitled 4

D:/School/COLLEGE/University of the Philippines/II - 2nd Sem/CS 21/Lab/MP 2/aaNEW MP 2/CS 21 Lab 12 SystemVerilog Files/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,
7                  output logic [1:0] aluop);
8
9    logic [8:0] controls;
10
11   assign {regwrite, regdst, alusrc, branch, memwrite,
12           memtoreg, jump, aluop} = controls;
13
14   always_comb
15     case(op)
16       6'b000000: controls <= 9'b110000010; // RTYPE
17       6'b100011: controls <= 9'b101001000; // LW
18       6'b101011: controls <= 9'b001010000; // SW
19       6'b000100: controls <= 9'b000100001; // BEQ
20       6'b001000: controls <= 9'b101000000; // ADDI
21       6'b000010: controls <= 9'b000000100; // J
22       6'b001110: controls <= 9'b101000011; // XORI , xor bitwise aluop set to 11 (the unused)
23       default:   controls <= 9'bxxxxxxxxx; // illegal op
24     endcase
25  endmodule
```

Code Block 4. maindec.sv with `xori` opcode condition

In Code Block 5, we can see the newly fed `op` opcode signal in Line 5. I then added a new `aluop` signal case `11` in Lines 12-14 to accommodate more I-type instructions. Note that the `11` ALUOp signal is unused for the *vanilla* processor, so I used that for the I-type instructions in this project such as `xori`. I also changed the default case to have the condition `aluop === 10` in Line 15 of Code Block 5, which stores all the conditions for the funct field of R-type instructions.



Code Block 5. aludec.sv with extended 4-bit ALUcontrol signals and I-type ALUOp case

I then extended all the assigned ALUControl signals by one bit In Lines 10-11, Line 13, and Lines 16-21 with the following values:

- [`aluop == 00`]: `alucontrol <= 0010` (for add, extended with zero)
- [`aluop == 01`]: `alucontrol <= 1010` (I used this signal since extending this to one would cause issues in the processor since the signal `alucontrol[2:0] = 110` is also used by the slt operation. That is why I used 1010 for subtraction so that the rightmost bit is still 1 to invert the b operand, and since subtraction also uses the add operation with the signal `alucontrol[2:0] = 010`)
- [`aluop == 11`]
  - [`op == 001110`]: for the `xori` instruction, with the new `alucontrol <= 0100` signal for the bitwise XOR operation discussed later
- [`aluop == 11`]:
  - [`funct == 100000`]: `alucontrol <= 0010` (for add, extended with zero)
  - [`funct == 100010`]: `alucontrol <= 1010` (for adding with inverted b operand, explained in the case when `aluop == 01`)
  - [`funct == 100100`]: `alucontrol <= 0000` (for AND, extended with zero)
  - [`funct == 100101`]: `alucontrol <= 0001` (for OR, extended with zero)
  - [`funct == 101010`]: `alucontrol <= 1111` (for set less than, extended with one)
  - [default case for `funct`]: `alucontrol <= xxxx` (for set less than, extended with one)

Finally, in Code Block 6, I changed the bit that was checked (in deciding whether to invert the b operand for subtraction) from the third bit (`alucontrol[2]`) to the fourth bit (`alucontrol[3]`) in Lines 8 and 9 to adjust to the extension of the ALUcontrol signal. I also extended the `alucontrol[2:0]` cases to four bits `alucontrol[3:0]` to accommodate

more ALU operations. Also note that I added an additional case for subtraction (for branches) in Line 16 of Code Block 6 which also sets `result` as the computed `sum` (with b operand inverted).

- `alucontrol[3:0] === 0000`: AND case extended by zero
- `alucontrol[3:0] === 0001`: OR case extended by zero
- `alucontrol[3:0] === 0010`: ADD case extended by zero
- `alucontrol[3:0] === 1010`: Subtraction case extended by one
- `alucontrol[3:0] === 1111`: set less than case extended by zero
- `alucontrol[3:0] === 0100`: new bitwise XOR case

```
1  module alu(input  logic [31:0] a, b,
2             input  logic [3:0]  alucontrol,
3             output logic [31:0] result,
4             output logic        zero);
5
6    logic [31:0] condinvb, sum;
7
8    assign condinvb = alucontrol[3] ? ~b : b;
9    assign sum = a + condinvb + alucontrol[3]; // when 4th bit is 1 (sub or slt), NOT b operand for minusing
10
11   always_comb
12     case (alucontrol[3:0])
13       4'b0000: result = a & b; //AND
14       4'b0001: result = a | b; //OR
15       4'b0010: result = sum; //ADD
16       4'b1010: result = sum; //SUBTRACT
17       4'b1111: result = sum[31]; //SLT
18       4'b0100: result = a ^ {16'h0000, b[15:0]}; //XOR
19     endcase
20
21   assign zero = (result == 32'b0);
22 endmodule
```

Code Block 6. alu.sv with extended 4-bit ALUcontrol signal, extended 3-bit ALU control signals, and revised “b operand” inverter and sum

Schematic Changes

- ALU Decoder output for ALUcontrol is now 4 bits wide, and takes in `op` signal input

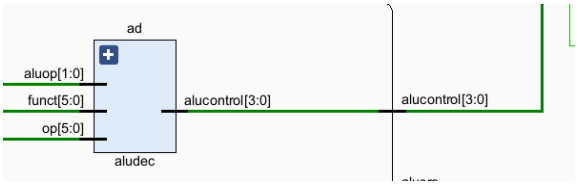


Figure 1. ALU Decoder with widened ALUControl output with `op` signal input

- ALUControl input to the datapath module is now 4 bits wide

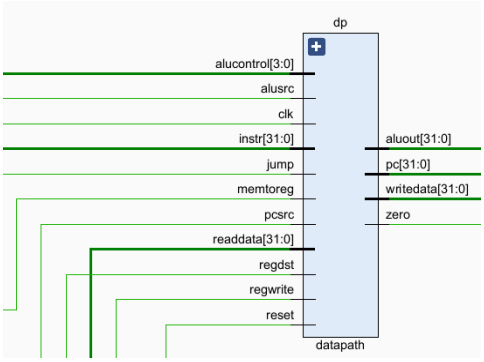


Figure 2. Datapath module with widened ALUControl output

- ALUControl input to ALU is now 4 bits wide

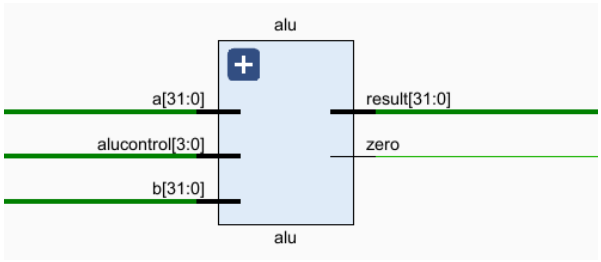


Figure 3. ALU with widened ALUControl signal

- New XOR operation inside ALU, and input signals to mux, and the selector, are now both 4 bits wide

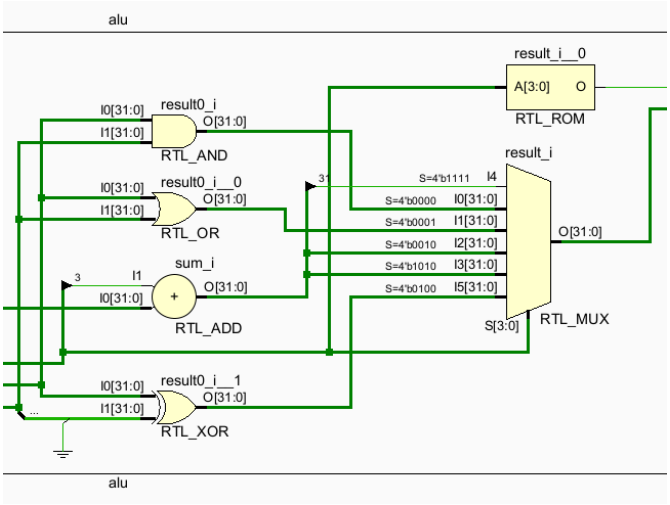


Figure 4. ALU new XOR operation and widened mux inputs

III. `xori` Changes

The `xori` instruction is an I-type instruction with opcode `001110` that takes in a register as its first operand, and an immediate value as its second operand, and applies the bitwise XOR operation in the two operands and stores the zero-extended result in the destination register.

With the preliminary changes, I first added a new opcode case in the Main Decoder (Line 22 of Code Block 4) which sets the control signals as follows:

- `RegWrite = 1` (since `xori` writes to a destination register)
- `RegDST = 0` (since `xori` is an I-type instruction, its destination register is located in bits 20:16 of the 32-bit instruction)
- `ALUSrc = 1` (since `xori` takes in the sign-extended immediate value as the second operand of the ALU)
- `Branch = 0` (`xori` does not branch)
- `MemWrite = 0` (`xori` does not write to memory)
- `MemtoReg = 0` (`xori` does not load from memory to the register file)

- `Jump = 0` (`xori` does not jump)
- `ALUOp = 11` (the unused opcode signal where other I-type instructions are accommodated in this revised processor)

I then added an opcode case in the ALU decoder (Line 13 in Code Block 5) which sets `alucontrol <= 0100`. Note that the fourth bit is zero since the b operand will not be inverted, and that `alucontrol[2:0]` is 100 since it is currently unused.

I then added a new `alucontrol[3:0]` case in Line 17 of Code Block 6 that sets the 32-bit `result` output of the ALU with the bitwise XOR result of operands a and zero-extended 16-bit b.

Schematic changes of `xori`

There is now a new XOR gate inside the ALU that applies the bitwise XOR operation on the two source operandes (see Figure 4).

Testing of `xori`
I used the following MIPS code to test `xori`, with its machine code translation to be loaded in the instruction memory `memfile.mem` with corresponding waveform in Figure 5

```
addi $t0, $zero, 0x0000
xori  $t0, $t0, 0xFFFF
add $t0, $t0, $zero # 0x0000 XOR 0xFFFF = 0x0000FFFF

addi $t0, $zero, 0x0000
xori  $t0, $t0, 0x0000
add $t0, $t0, $zero # 0x0000 XOR 0x0000 = 0x00000000

addi $t0, $zero, 0xFFFF
xori  $t0, $t0, 0xFFFF
add $t0, $t0, $zero # 0xFFFF XOR 0xFFFF = 0xFFFF0000

addi $t0, $zero, 0xF0F0
xori  $t0, $t0, 0xCCCC
add $t0, $t0, $zero # 0xF0F0 XOR 0xCCCC = 0xFFFF3C3C
```

Code Block 7. `xori` test MIPS code

```
20080000
3908FFFF
01004020
20080000
39080000
01004020
```

```
2008FFFF
3908FFFF
01004020
2008F0F0
3908CCCC
01004020
```

Code Block 8. Hexadecimal translation of Code Block 7

Figure 5. `xori` test MIPS code waveform

IV. `lui` Changes

`lui`, or Load Upper Immediate, is an I-type instruction, with opcode `001111`, that takes in an immediate value that is written on the "upper" 16 bits of the destination register. Hence, we make the following adjustments:

I added a new opcode case in the main decoder in Line 23 of Code Block 9, which sets the control signals as follows:

- `RegWrite = 1` (since `lui` writes to a destination register)

- `RegDST = 0` (since `lui` is an I-type instruction, its destination register is located in bits 20:16 of the 32-bit instruction)
- `ALUSrc = 1` (since `lui` takes in the sign-extended immediate value as the second operand of the ALU)
- `Branch = 0` (`lui` does not branch)
- `MemWrite = 0` (`lui` does not write to memory)
- `MemtoReg = 0` (`lui` does not load from memory to the register file)
- `Jump = 0` (`lui` does not jump)
- `ALUOp = 11` (the unused opcode signal where other I-type instructions are accommodated in this revised processor)

```
14    always_comb
15      case(op)
16        6'b000000: controls <= 9'b110000010; // RTYPE
17        6'b100011: controls <= 9'b101001000; // LW
18        6'b101011: controls <= 9'b001010000; // SW
19        6'b000100: controls <= 9'b000100001; // BEQ
20        6'b001000: controls <= 9'b101000000; // ADDI
21        6'b000010: controls <= 9'b000000100; // J
22        6'b001110: controls <= 9'b101000011; // XORI, xor bitwise aluop set to 11 (the unused)
23        6'b001111: controls <= 9'b101000011; // LUI, aluop set to 11 (the unused)
24        default:   controls <= 9'bxxxxxxxxx; // illegal op
25      endcase
26  endmodule
```

Code Block 9. Snippet of maindec.sv with `lui` opcode case

I also added a new opcode case in the ALU decoder under the case `op == 11` in Line 14 of Code Block 9 which sets the ALUControl signal to 0101. The rightmost bit is zero since we will not invert the b operand, and the `alucontrol[2:0] = 101` because it is unused.

```
8   always_comb
9     case(aluop)
10      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
11      2'b01: alucontrol <= 4'b1010;  // sub (for beq), bit 4 set to 1 since ALU NOTs operand b for minusing
12      2'b11: case(op) // I-type instructions
13          6'b001110: alucontrol <= 4'b0100; // xor
14          6'b001111: alucontrol <= 4'b0101; // lui
15      endcase
```

Code Block 10. Snippet of aludec.sv with `lui` opcode case

Lastly, I added a new ALUControl case `alucontrol[3:0] === 0101` in the ALU in Line 19 of Code Block 11. This sets the result to the b operand concatenated with four hexadecimal zeroes to the right.

```
11  always_comb
12    case (alucontrol[3:0])
13      4'b0000: result = a & b; //AND
14      4'b0001: result = a | b; //OR
15      4'b0010: result = sum; //ADD
16      4'b1010: result = sum; //SUBTRACT
17      4'b1111: result = sum[31]; //SLT
18      4'b0100: result = a ^ {16'h0000, b[15:0]}; //XOR
19      4'b0101: result = {b, 16'h0000}; //LUI
20    endcase
```

Code Block 11. Snippet of alu.sv with new `lui` operation

Schematic changes of `lui`

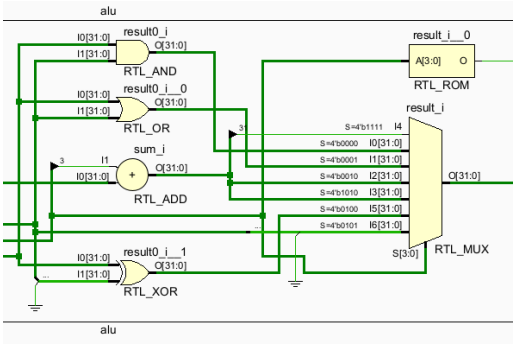


Figure 6. New `0101` mux input line in ALU

Testing of `lui`

I used the following MIPS code to test `lui`, with its machine code translation to be loaded in the instruction memory `memfile.mem` with corresponding waveform in Figure 7.

```
lui $t0, 0xC0DE
add $t0, $zero, $t0 # $t0 stores 0xC0DE0000

lui $t0, 0xDEAD
add $t0, $zero, $t0 # $t0 stores 0xDEAD0000

lui $t0, 0xBABE
add $t0, $zero, $t0 # $t0 stores 0xBABE0000

lui $t0, 0xBEEF
add $t0, $zero, $t0 # $t0 stores 0xBEEF0000
```

Code Block 12. `lui` test MIPS code

```
3C08C0DE
00084020
3C08DEAD
00084020
3C08BABE
00084020
3C08BEEF
00084020
```

Code Block 13. Hexadecimal translation of Code Block 12

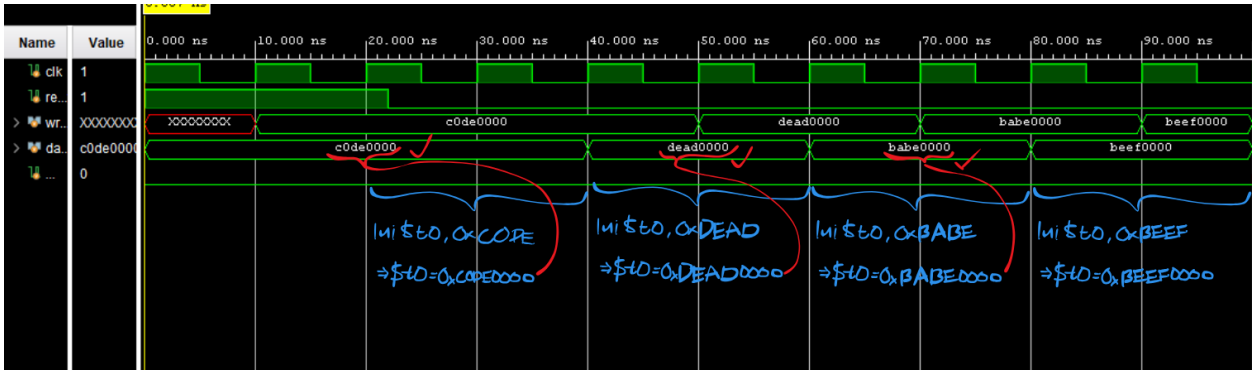


Figure 7. `lui` test MIPS code waveform

V. `srlv` Changes

`srlv`, or Shift Right Logical Variable, is an R-type instruction (with funct `000110`) that shifts to the right the value stored in the first source register, by the times specified in the second source register. Note that since `srlv` is an R-type instruction, no edits are needed in the Main Decoder as it is already handled by Line 16 of Code Block 4.

Also, when this example MIPS Code is run in MARS (Code Block 14), the result is that the shifted value is only shifted two times, and not 0xC0DEBA02 times.

srlv test.asm

```
1 li $t0, 0xC0DEBA02
2 li $t1, 16
3 srlv $t2, $t1, $t0
```

Code Block 14. MIPS code to test `srlv` behavior

$t0	8	0xc0deba02
$t1	9	0x00000010
$t2	10	0x00000004

Figure 8. Part of register file after execution of Code Block 14

After further testing, it is observed that srlv only takes in the 5 lowest bits for the register that specifies the number of shifts. Hence, we make the following adjustments:

```
16      2'b10: case(funct)          // R-type instructions
17          6'b100000: alucontrol <= 4'b0010; // add
18          6'b100010: alucontrol <= 4'b1010; // sub, bit 4 set to 1 since ALU NOTs operand b for minusing
19          6'b100100: alucontrol <= 4'b0000; // and
20          6'b100101: alucontrol <= 4'b0001; // or
21          6'b101010: alucontrol <= 4'b1111; // slt, bit 4 set to 1 since ALU NOTs operand b for minusing
22          6'b000110: alucontrol <= 4'b0110; // srlv
23          default:   alucontrol <= 4'bxxxx; // ???
24      endcase
```

Code Block 15. ALU decoder with new `srlv` funct case

I added a new funct case in the ALU decoder with the funct code of `srlv` in Line 22 of Code Block 15; it sets the `alucontrol` signal to 0110. The rightmost bit is zero since we will not invert the b operand, and `alucontrol[2:0] = 110` because it is unused.

Lastly, I added a new ALUControl case `alucontrol[3:0] === 0110` in the ALU in Line 20 of Code Block 16. This sets the result to the first operand shifted to the right logically, with the number of times stated by the value of the second operand (only first five bits to replicate the behavior in Code Block 14).

```
11    always_comb
12      case (alucontrol[3:0])
13        4'b0000: result = a & b; //AND
14        4'b0001: result = a | b; //OR
15        4'b0010: result = sum; //ADD
16        4'b1010: result = sum; //SUBTRACT
17        4'b1111: result = sum[31]; //SLT
18        4'b0100: result = a ^ {16'h0000, b[15:0]}; //XOR
19        4'b0101: result = {b, 16'h0000}; //LUI
20        4'b0110: result = b >> a[4:0]; //SRLV
21      endcase
```

Code Block 16. Snippet of alu.sv with new `srlv` operation

Schematic changes of `srlv`

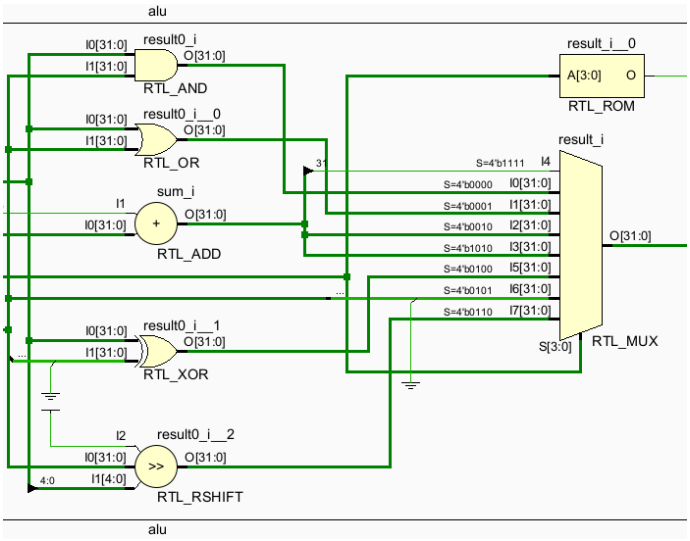


Figure 9. New RSHIFT operator and `srlv 0110` input line in the mux in the ALU

Testing of `srlv`

I used the following MIPS code to test `srlv`, with its machine code translation to be loaded in the instruction memory `memfile.mem` with corresponding waveform in Figure 10.

```
addi $t0, $zero, 0x80
addi $t1, $zero, 0x3
srlv $t2, $t0, $t1
add $t2, $t2, $zero # $t2 should store 0x00000010
```

```
addi $t0, $zero, 0x539
addi $t1, $zero, 0x0A
srlv $t2, $t0, $t1
add $t2, $t2, $zero # $t2 should store 0x00000001

addi $t0, $zero, 0x8
srlv $t1, $t0, $zero
add $t1, $t1, $zero # $t1 should store 0x00000008

addi $t0, $zero, 0xB
addi $t1, $zero, 0x2
srlv $t2, $t0, $t1
add $t2, $t2, $zero # $t2 should store 0x00000002

addi $t0, $zero, 0xB182
srlv $t1, $zero, $t0
add $t1, $t1, $zero # $t1 should store 0x00000000
```

Code Block 17. `srlv` test MIPS code

```
20080080
20090003
01285006
01405020
20080539
2009000A
01285006
01405020
20080008
00084806
01204820
2008000B
20090002
01285006
01405020
2008B182
01004806
01204820
```

Code Block 18. Hexadecimal translation of Code Block 17

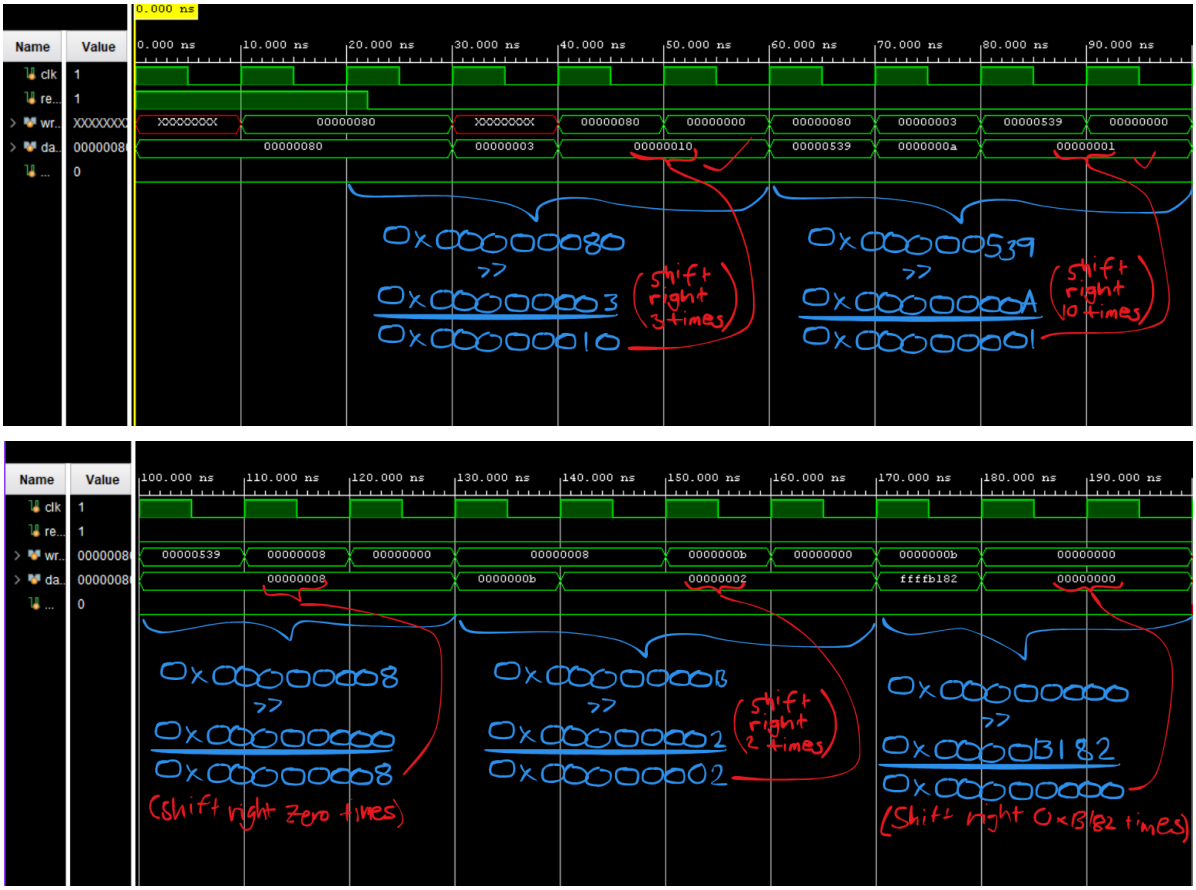


Figure 10. `srlv` test MIPS code waveform

VI. `bgtz` Changes

`bgtz`, or Branch on Greater Than Zero (with opcode `000111`), is a pseudo-instruction that takes one source register and compares it with zero; if it is greater than zero, it branches to the Branch Target Address specified by the offset in the instruction's immediate field.

I added a new opcode case in the main decoder in Line 24 of Code Block 19, which sets the control signals as follows:

- `RegWrite = 0` (`bgtz` does not write to the register file)
- `RegDST = 0` (`bgtz` does not have a destination register)
- `ALUSrc = 0` (`bgtz` only takes one source register for the ALU, so this will also be unused)
- `Branch = 1` (`bgtz` is a branch instruction, and thus, has the capacity to branch)
- `MemWrite = 0` (`bgtz` does not write to memory)
- `MemtoReg = 0` (`bgtz` does not load from memory to the register file)
- `Jump = 0` (`bgtz` does not jump)
- `ALUOp = 11` (`bgtz` is bunched with the I-type instructions since its opcode is not 0x00)

```
14  always_comb
15    case(op)
16      6'b000000: controls <= 9'b110000010; // RTYPE
17      6'b100011: controls <= 9'b101001000; // LW
18      6'b101011: controls <= 9'b001010000; // SW
19      6'b000100: controls <= 9'b000100001; // BEQ
20      6'b001000: controls <= 9'b101000000; // ADDI
21      6'b000010: controls <= 9'b000000100; // J
22      6'b001110: controls <= 9'b101000011; // XORI, xor bitwise aluop set to 11 (the unused)
23      6'b001111: controls <= 9'b101000011; // LUI, aluop set to 11 (the unused)
24      6'b000111: controls <= 9'b000100011; // BGTZ, aluop set to 11 (the unused)
25      default:   controls <= 9'bxxxxxxxxx; // illegal op
26    endcase
27  endmodule
```

Code Block 19. Snippet of maindec.sv with `bgtz` opcode case

I then added a new opcode case under [`aluop === 11`] for `bgtz`, which sets `alucontrol` to 0011 in Line 15 of Code Block 20.

```
8   always_comb
9     case(aluop)
10      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
11      2'b01: alucontrol <= 4'b1010;  // sub (for beq), bit 4 set to 1 since ALU NOTs operand b for minusing
12      2'b11: case(op) // I-type instructions
13          6'b001110: alucontrol <= 4'b0100; // xor
14          6'b001111: alucontrol <= 4'b0101; // lui
15          6'b000111: alucontrol <= 4'b0011; // bgtz
16      endcase
17      2'b10: case(funct)          // R-type instructions
18          6'b100000: alucontrol <= 4'b0010; // add
19          6'b100010: alucontrol <= 4'b1010; // sub, bit 4 set to 1 since ALU NOTs operand b for minusing
20          6'b100100: alucontrol <= 4'b0000; // and
21          6'b100101: alucontrol <= 4'b0001; // or
22          6'b101010: alucontrol <= 4'b1111; // slt, bit 4 set to 1 since ALU NOTs operand b for minusing
23          6'b000110: alucontrol <= 4'b0110; // srlv
24          default:   alucontrol <= 4'bxxxx; // ???
25        endcase
26    endcase
27  endmodule
```

Code Block 20. Snippet of aludec.sv with `bgtz` opcode case

Lastly, I added a new ALUControl case `alucontrol[3:0] === 0011` in the ALU in Lines 21-26 of Code Block 21. Line 22 checks of the value of the a operand is nonzero, and its rightmost bit is not 1 (this means that it is not a negative number); if so, Line 23 sets the result to 0. If not, Line 25 sets the result to 1.

```
11  always_comb
12    case (alucontrol[3:0])
13      4'b0000: result = a & b; //AND
14      4'b0001: result = a | b; //OR
15      4'b0010: result = sum; //ADD
16      4'b1010: result = sum; //SUBTRACT
17      4'b1111: result = sum[31]; //SLT
18      4'b0100: result = a ^ b; //XOR
19      4'b0101: result = {b, 16'h0000}; //LUI
20      4'b0110: result = b >> a; //SRLV
21      4'b0011: begin // BGTZ
22        if (a > 0 & a[31] ==! 1) // if a is gr8r than 0 and non negative
23            result = 0;
24        else
25            result = 1;
26        end
27    endcase
28
29  assign zero = (result == 32'b0);
```

Code Block 21. Snippet of alu.sv with new `bgtz` operation

When the first if statement is fulfilled and the result is zero, Line 29 sets the zero bit to 1. When `zero === 1`, it will let the high branch signal from the control unit go through (via the AND gate in Figure 11) to the mux that chooses between PC + 4 and the Branch Target Address; the BTA is then let through to be executed by the processor.

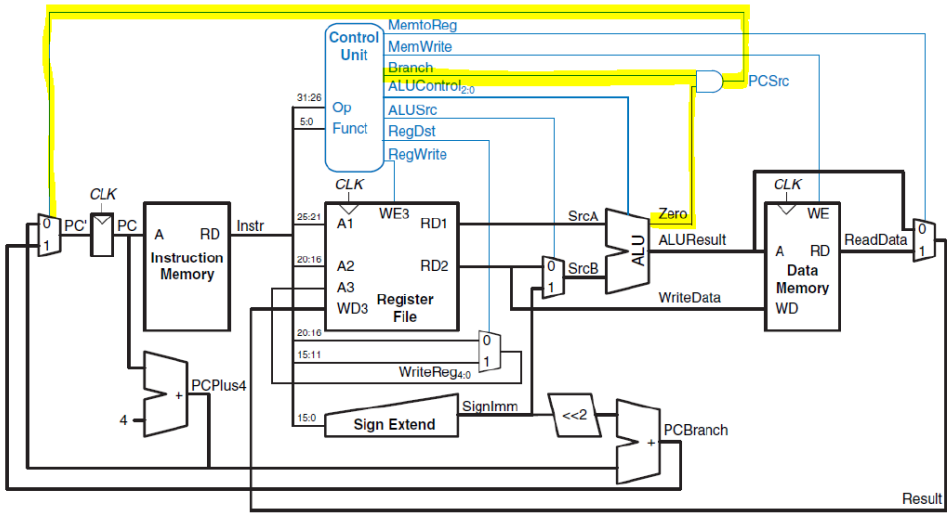


Figure 11. MIPS single-cycle processor with the branch path highlighted

Schematic changes of `bgtz`

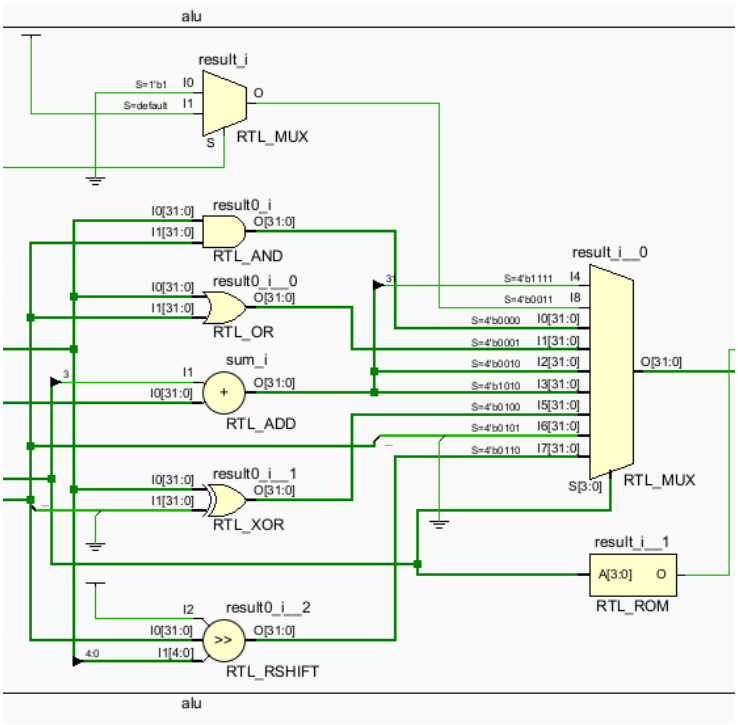


Figure 12. New “zero” mux and `bgtz 0011` input line in big mux in ALU

Testing of `bgtz`

I used the following MIPS code to test `bgtz`, with its machine code translation to be loaded in the instruction memory `memfile.mem` with corresponding waveform in Figure 13.

```
add $t0, $zero, $zero
bgtz $t0, 0x1       # not taken
addi $t0, $t0, 0x7
add $t0, $t0, $zero # $t0 should store 0x7

addi $t0, $zero, 0x1
bgtz $t0, 0x1       # taken
addi $t0, $t0, 0x7
add $t0, $t0, $zero # $t0 should store 0x1

lui $t0, 0xFFFF
addi $t1, $zero, 0xFFF7
or $t0, $t0, $t1      # $t0 stores -9 in 2C
bgtz $t0, 0x1         # not taken
addi $t0, $t0, 0x9
add $t0, $t0, $zero   # $t0 should store 0x0
```

Code Block 22. `bgtz` test MIPS code

```
00004020
1D000001
21080007
01004020
20080001
1D000001
21080007
01004020
3C08FFFF
2009FFF7
01094025
1D000001
21080009
01004020
```

Code Block 23. Hexadecimal translation of Code Block 22

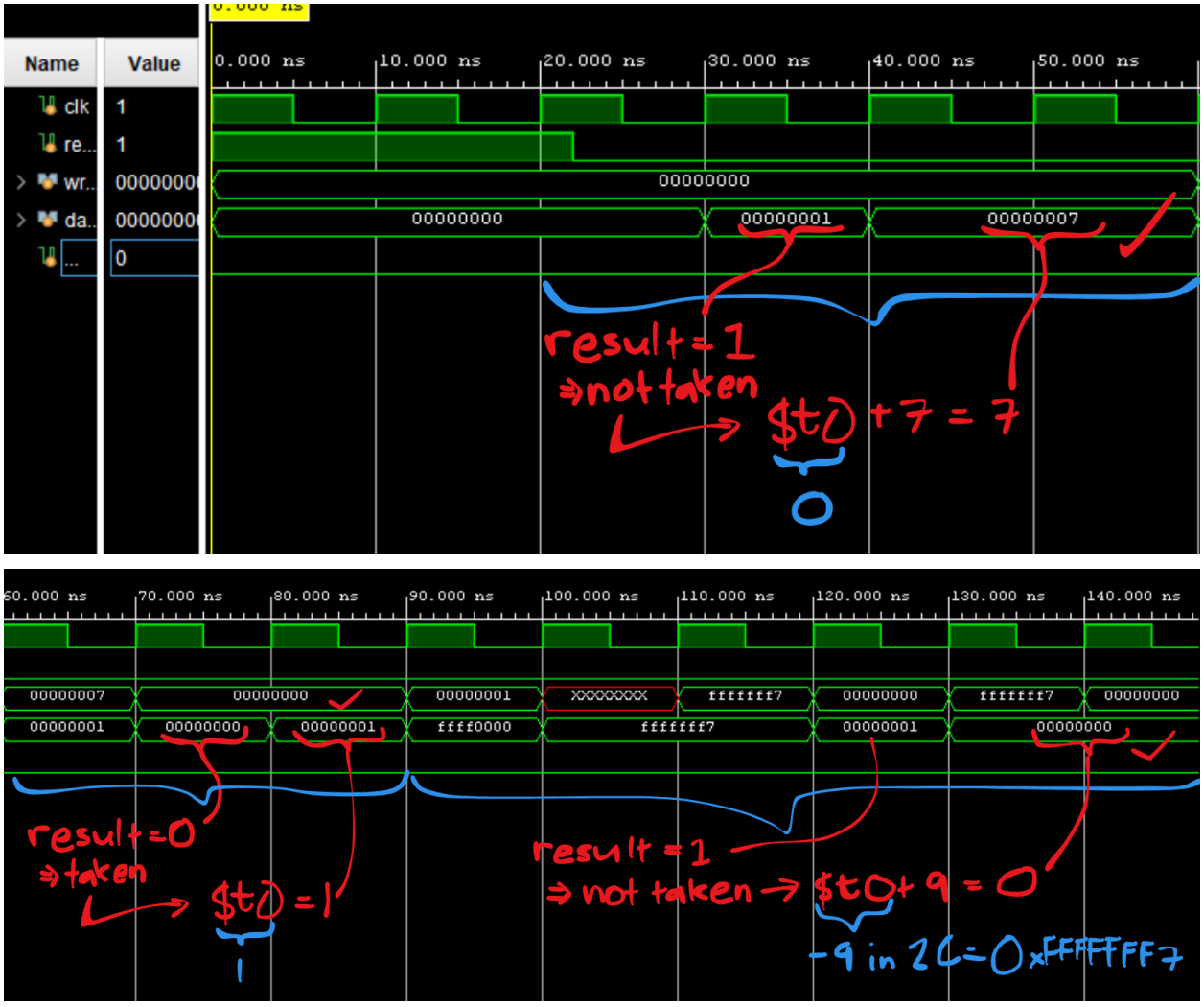


Figure 13. `bgtz` test MIPS code waveform

VII. `li` Changes

`li`, or Load Immediate (with opcode `010001` as specified in the project specs), is a pseudo-instruction that takes in a destination register (for this project, the `rt` register), and takes in a 16-bit immediate value that is loaded to said register, zero-extended. Originally, `li` can load 32-bit values to the destination register, but the MP 2 specs only require at most 16-bit immediate values to be loaded to the `rt` register. Also note that the `rs` bits are *don't cares*.

I added a new opcode case in the main decoder in Line 25 of Code Block 24, which sets the control signals as follows:

- `RegWrite = 1` (`li` writes to the register file)
- `RegDST = 0` (`li` has `rt` as its destination register)
- `ALUSrc = 0` (`li` takes in the zero-extended immediate value as the second operand of the ALU)
- `Branch = 0` (`li` does not branch)

- `MemWrite = 0` (`li` does not write to memory)
- `MemtoReg = 0` (`li` does not load from memory to the register file)
- `Jump = 0` (`li` does not jump)
- `ALUOp = 11` (the unused opcode signal where other I-type instructions are accommodated in this revised processor)

```
14      always_comb
15        case(op)
16          6'b000000: controls <= 9'b110000010; // RTYPE
17          6'b100011: controls <= 9'b101001000; // LW
18          6'b101011: controls <= 9'b001010000; // SW
19          6'b000100: controls <= 9'b000100001; // BEQ
20          6'b001000: controls <= 9'b101000000; // ADDI
21          6'b000010: controls <= 9'b000000100; // J
22          6'b001110: controls <= 9'b101000011; // XORI, xor bitwise aluop set to 11 (the unused)
23          6'b001111: controls <= 9'b101000011; // LUI, aluop set to 11 (the unused)
24          6'b000111: controls <= 9'b000100011; // BGTZ, aluop set to 11 (the unused)
25          6'b010001: controls <= 9'b101000011; // LI (16-bit from the vault version), aluop set to 11 (the unused)
26          default:   controls <= 9'bxxxxxxxxx; // illegal op
27        endcase
28    endmodule
```

Code Block 24. Snippet of maindec.sv with `li` opcode case

I also added a new opcode case in the ALU decoder under the case `op == 11` in Line 16 of Code Block 25 which sets the ALUControl signal to 0111. The rightmost bit is zero since we will not invert the b operand, and the `alucontrol[2:0] = 111` because it is unused.

```
10      2'b00: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
11      2'b01: alucontrol <= 4'b1010;  // sub (for beq), bit 4 set to 1 since ALU NOTs operand b for minusing
12      2'b11: case(op) // I-type instructions
13          6'b001110: alucontrol <= 4'b0100; // xor
14          6'b001111: alucontrol <= 4'b0101; // lui
15          6'b000111: alucontrol <= 4'b0011; // bgtz
16          6'b010001: alucontrol <= 4'b0111; // li (16-bit from the vault version)
17      endcase
```

Code Block 25. Snippet of aludec.sv with `li` opcode case

Lastly, I added a new ALUControl case `alucontrol === 0111` in the ALU in Line 27 of Code Block 26. This sets the result to the b operand concatenated with four hexadecimal zeroes to the left since it is zero-extended.

```
11    always_comb
12      case (alucontrol[3:0])
13        4'b0000: result = a & b; //AND
14        4'b0001: result = a | b; //OR
15        4'b0010: result = sum; //ADD
16        4'b1010: result = sum; //SUBTRACT
17        4'b1111: result = sum[31]; //SLT
18        4'b0100: result = a ^ b; //XOR
19        4'b0101: result = {b, 16'h0000}; //LUI
20        4'b0110: result = b >> a; //SRLV
21        4'b0011: begin // BGTZ
22          if (a > 0 & a[31] ==! 1) // if a is gr8r than 0 and non negative
23              result = 0;
24          else
25              result = 1;
26          end
27        4'b0111: result = {16'h0000, b}; // LI (16-bit from the vault version)
28      endcase
```

Code Block 26. Snippet of alu.sv with new `li` operation

Schematic changes of `li`

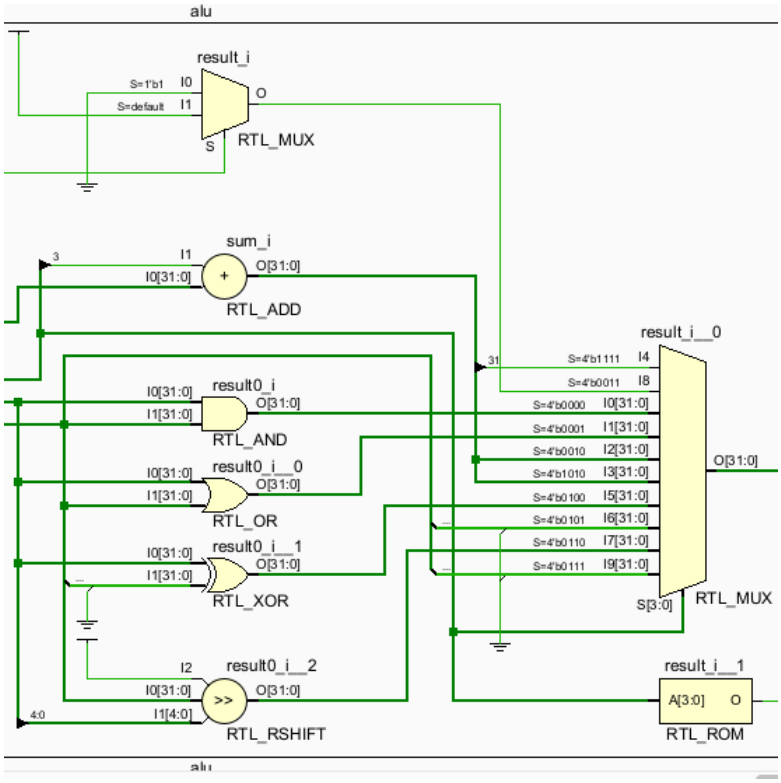


Figure 14. New `li 0111` mux input line in the ALU

Testing of `li`

I used the following MIPS code to test `li`, with its machine code translation to be loaded in the instruction memory `memfile.mem` with corresponding waveform in Figure 15.

```
li $t0, 0xC0DE
add $t0, $zero, $t0 # $t0 stores 0x0000C0DE

li $t0, 0xDEAD
add $t0, $zero, $t0 # $t0 stores 0x0000DEAD

li $t0, 0xC0DE # different register in rs (to show it doesn't matter)
add $t0, $zero, $t0 # $t0 stores 0x0000C0DE

li $t0, 0xDEAD # different register in rs (to show it doesn't matter)
add $t0, $zero, $t0 # $t0 stores 0x0000DEAD
```

Code Block 27. `li` test MIPS code

```
4608C0DE
00084020
4608DEAD
00084020
4628C0DE
00084020
4628DEAD
00084020
```

Code Block 28. Hexadecimal translation of Code Block 27

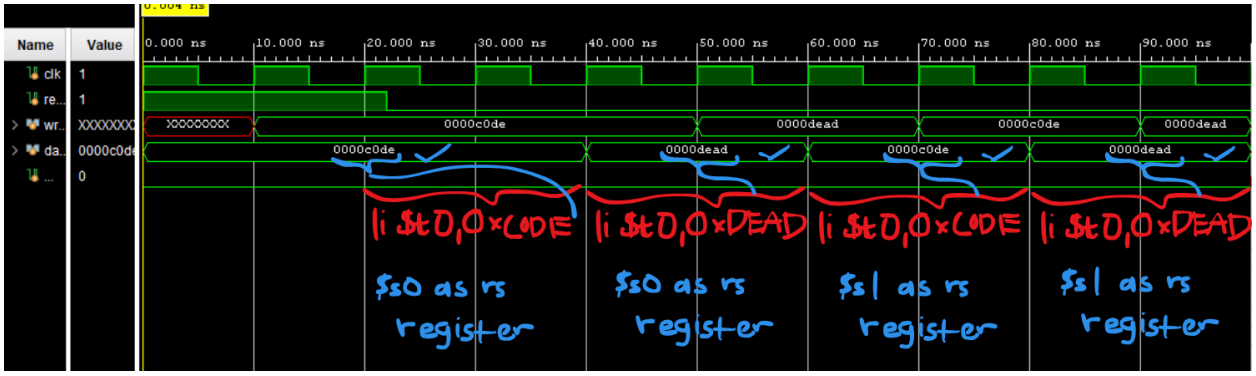


Figure 15. `li` test MIPS code waveform