CONTENTS

PREFACE 3
XORI INSTRUCTION 3
Maindec changes 3
Aludec changes 4
ALU changes 4
Testing 5
LUI INSTRUCTION 5
Maindec changes 5
Aludec changes 6
ALU changes 7
Controller changes 7
Testing 7
SRLV INSTRUCTION 8
Aludec changes 8
ALU changes 9
Testing 9
BGTZ INSTRUCTION 9
Maindec changes 9
Aludec changes 10
ALU changes 10
Wire changes 11
Testing 12
LI INSTRUCTION 13
Maindec changes 13
Aludec changes 13
ALU changes 13
Testing 13
RUNXOR INSTRUCTION 14
Aludec changes 14
Controller changes 15
ALU changes 16

Testing 16
OTHER CHANGES 17

PREFACE

This documentation, unlike previous documentations (and perhaps other people's documentations), is written **concurrently** with the project. That is, as changes are being done to the project, the documentation is being written.

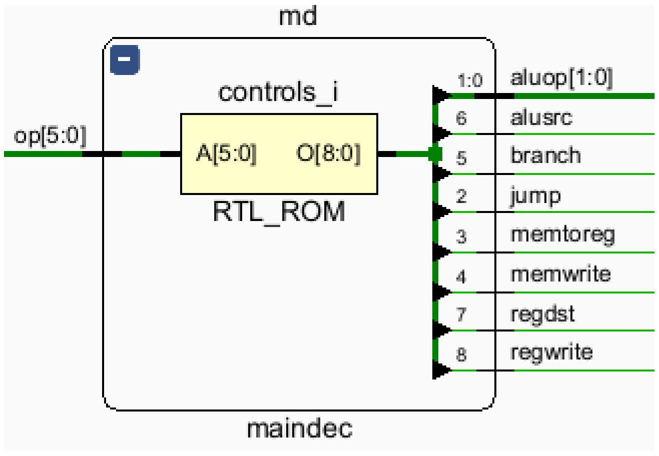
This has two consequences: firstly, that the further down the documentation the later time has passed. That is, the most recently written segments are found at the bottom of the document. Secondly, images in the first segments will likely be outdated when compared to the submission, with the bottom images containing the latest changes.

As a result, this documentation is written **accumulatively**: changes in the first sections will be found in later sections and will not be reverted.

XORI INSTRUCTION

XORI is an i-type instruction. As the current architecture of the processor can already handle ADDI, another i-type instruction which does a similar job, only little needs to be added to the processor to support XORI. The only difference between XORI and ADDI is that XORI performs an xor operation and ADDI performs an add operation. None of the changes to be performed for XORI require a change to the schematics of the processor.

Maindec changes



Seen above is the schematic of the maindec module. The maindec module receives a 6 bit input op and returns 9 bits as a single bitstring:

[regwrite][regdst][alusrc][branch][memwrite][memtoreg][jump][aluop]
where every component above is 1 bit long, barring aluop which is 2 bits

By default, the switch-case block in the maindec module does not support the opcode for the XORI instruction, 001110. We can simply append this into the module:

```
6'b001000: controls <= 9'b101000000; // ADDI
6'b001110: controls <= 9'b101000011; // XORI
```

Note that the control signal is the same as in ADDI except for the last two bits, aluop. This is because the bitstring 00 for aluop indicates addition, 01 indicates subtraction, while 10 is used for r-type instructions, as seen by this line:

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

Thus, we can use the remaining bitstring, 11, to use for XORI. 11 is set to be unused according to lecture 9, however there are no restrictions regarding using unused signals, thus it can be used for XORI in this situation.

Aludec changes

By default, the switch-case block in the aludec module does not support the brand new bitstring 11 as an ALUOp signal. We can simply append this into the module:

```
2'b01: alucontrol <= 3'b110;  // sub (for beq)\
2'b11: alucontrol <= 3'b011;  // xori
```

Note that according to lecture 9, 011 is an unused ALUControl signal. However, no restrictions have been indicated in the project specifications that unused signals in MIPS cannot be used in the project, thus this is okay.

ALU changes

Just because the signal for an XOR instruction will be sent by the ALU decoder does not mean that the ALU will automatically understand what the signal means, thus additions have to be made to the alu module to support the new ALUcontrol signal.

```
assign condinvb = alucontrol[2] ? ~b : b;
assign sum = a + condinvb + alucontrol[2];

always_comb
  case (alucontrol[1:0])
    2'b00: result = a & b;
    2'b01: result = a | b;
    2'b10: result = sum;
    2'b11: result = sum[31];
  endcase
```

Seen above is the default behavior of the alu module. By default, the switch-case block in the alu module does technically support the code 011 however this would lead to erroneous behavior. This is because the alu only uses the first bit of ALUControl to check for subtraction by getting the bit inverse of b, and only requires the last two bits to check for other instructions. In this example, 011 would lead to an wrongly computed SLT. We can add support for an XOR operation by adding another switch case block:

```
always_comb
  case(alucontrol)
    3'b011: result = a ^ b;
    default: case (alucontrol[1:0])
        2'b00: result = a & b;
        2'b01: result = a | b;
        2'b10: result = sum;
        2'b11: result = sum[31];
      endcase
  endcase
```

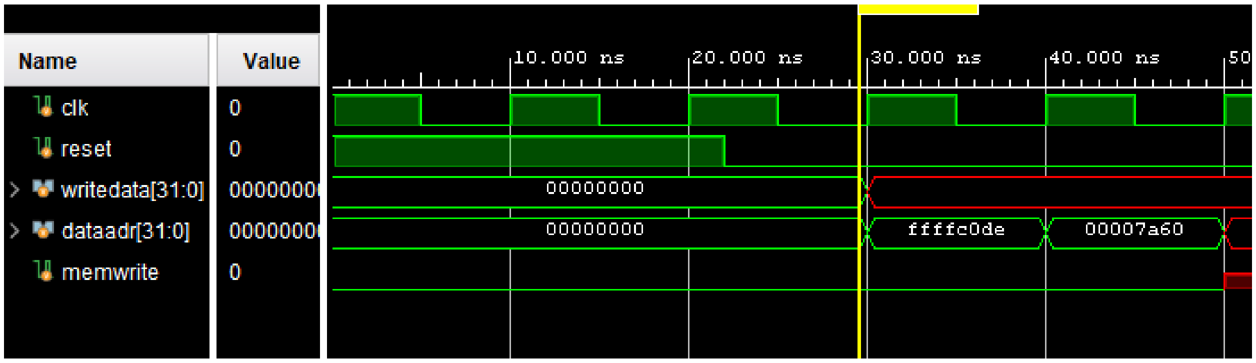
The above modification guarantees that if $\text{ALUControl} = 011$ then an XOR operation is performed. Note that this does not affect other instructions as by default (that is, if $\text{ALUControl} \neq 011$) then the functionality of the alu is unchanged. Also note that in the modification to aludec, 011 is an ALUControl signal unique to XORI.

Testing

Using the same testbench as used in Lab 12, which uses a file memfile.mem which consists of MIPS instructions in machine code, a simple program can be run to test the functionality of XORI.

```
addi 0 0 0          001000 00000 00000 0000000000000000 0x20000000
addi 1 0 0xC0DE     001000 00000 00001 1100000011011110 0x2001C0DE
addi 1 1 0          001000 00001 00001 0000000000000000 0x20210000
xori 2 1 0xBABE     001110 00001 00010 1011101010111110 0x3822BABE
#Final answer should be 0b0111 1010 0110 0000 or 0x7A60
```

Running the above program yields:



Where dataadr $= 0x7A60$, thus the instruction works as intended.

LUI INSTRUCTION

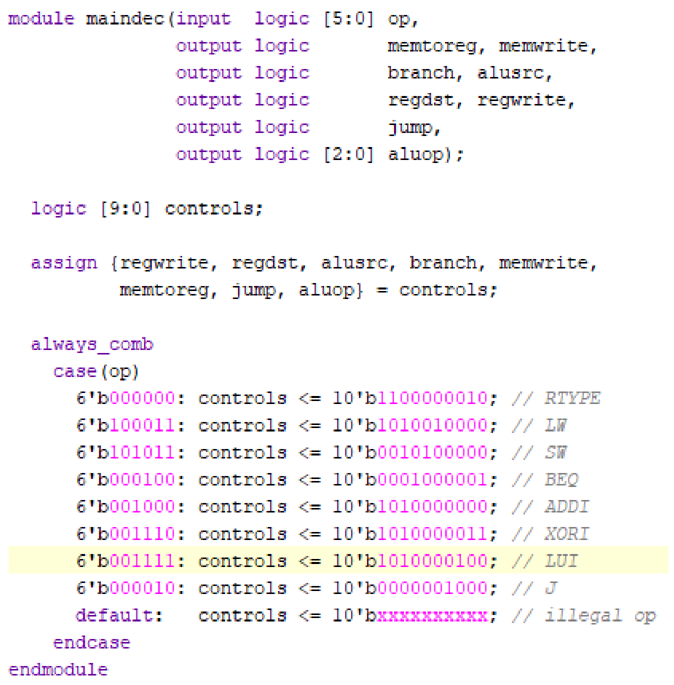
Load upper immediate is an instruction which places a 16 bit immediate into a register's upper 16 bits. As another i-type instruction, there no longer remain any ALUOp bitstrings to uniquely identify LUI (that is, 00, 01, 10, 11 are all under use), thus the schematic must be modified to accommodate for an extra instruction.

Additionally, an extra process must be implemented to shift the immediate 16 bits to the left to guarantee the instruction works correctly.

Maindec changes

As stated, ALUOp does not currently have enough bits to uniquely represent more than 3 i-type instructions. This can be solved by adding more bits to ALUOp. Doing so will render all previous instructions wrong as they all return a 2-bit ALUOp, so those instructions must be modified as well.

After increasing the bit size of ALUOp, the switch-case in the maindec module can then be appended to support the opcode of the LUI instruction, 001111.
All in all, the above changes create:



```
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [2:0] aluop);

  logic [9:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 10'b1100000010; // RTYPE
      6'b100011: controls <= 10'b1010010000; // LW
      6'b101011: controls <= 10'b0010100000; // SW
      6'b000100: controls <= 10'b0001000001; // BEQ
      6'b001000: controls <= 10'b1010000000; // ADDI
      6'b001110: controls <= 10'b1010000011; // XORI
      6'b001111: controls <= 10'b1010000100; // LUI
      6'b000010: controls <= 10'b0000001000; // J
      default:   controls <= 10'bxxxxxxxxxx; // illegal op
    endcase
endmodule
```

Note that controls is now a 10-bit bitstring, and aluop is now 3 bits long. Now 5 of the $2^3 = 8$ options for ALUOp have been covered: 000 for addition, 001 for subtraction, 010 for r-type instructions, 011 for XORI, and 100 for LUI.

Additionally, the remaining control bits of LUI are the same as both XORI and ADDI. This is because they serve very similar functions, just with different operations performed.

Aludec changes

Similarly to maindec, changing ALUOp to be 3 bits long will cause erroneous behavior in previous instructions as well, so aludec must be rewritten to accommodate the extra bits. Additionally, the switch-case in the aludec module can be appended to support the ALUOp of LUI, 100.

```
module aludec(input  logic [5:0] funct,
              input  logic [2:0] aluop,
              output logic [2:0] alucontrol);

  always_comb
    case(aluop)
      3'b000: alucontrol <= 3'b010;  // add (for lw/sw/addi)
      3'b001: alucontrol <= 3'b110;  // sub (for beq)\
      3'b011: alucontrol <= 3'b011;  // xori
      3'b100: alucontrol <= 3'b100;  // lui
      default: case(funct)          // R-type instructions
          6'b100000: alucontrol <= 3'b010; // add
          6'b100010: alucontrol <= 3'b110; // sub
          6'b100100: alucontrol <= 3'b000; // and
          6'b100101: alucontrol <= 3'b001; // or
          6'b101010: alucontrol <= 3'b111; // slt
          default:   alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule
```

Note that aluop is now 3 bits long, and all references to it has been modified accordingly. The associated alucontrol signal for LUI has been arbitrarily chosen to be 100, as this is the lowest numbered unused ALUControl signal (000 for and, 001 for or, 010 for add, 011 for xori, and 100 remains unused).

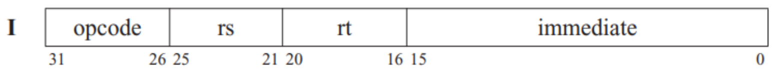
ALU changes

The change in the size of ALUOp no longer affects the alu module, however changes must still be made to accommodate the functionality of LUI. This is achieved by creating a new entry in the switch-case made for XORI:

```
case(alucontrol)
  3'b011: result = a ^ b;
  3'b100: result = b << 16;
  default: case (alucontrol[1:0])
```

The result is now the sign-extended immediate, b, shifted logically left by 16 bits. Note that the ALU input a is not taken into account for LUI: this is because according to the green sheet, the part of the instruction dedicated to rs (which in the case of LUI, is a) does not affect the functionality of LUI whatsoever.

Load Upper Imm. `lui` I R[rt] = {imm, 16’b0}



Thus the value of a can be seen as a don’t-care value for the purposes of the LUI instruction.

Controller changes

While the change in ALUOp’s size may not affect the alu module, it does affect the controller module. This is because controller houses the wire which connects maindec and aludec together, and sends the signal ALUOp from maindec to aludec. The change for controller is simple: aluop must simply be changed to be 3 bits wide.

```
logic [2:0] aluop;
logic       branch;
```

Testing

A short program can be used to quickly check functions of various different instructions (note that all instructions were modified as a result of changing the width of ALUOp):

```
addi 0 0 0          001000 00000 00000 0000000000000000 0x20000000
lui 1 0xC0DE        001111 XXXXX 00001 1100000011011110 0x3C01C0DE
addi 1 1 0xBABE     001000 00001 00001 1011101010111110 0x20210101
xori 2 1 0xFFFF     001110 00001 00010 1111111111111111 0x3822FFFF
#Final answer should be 3F22 4541
```

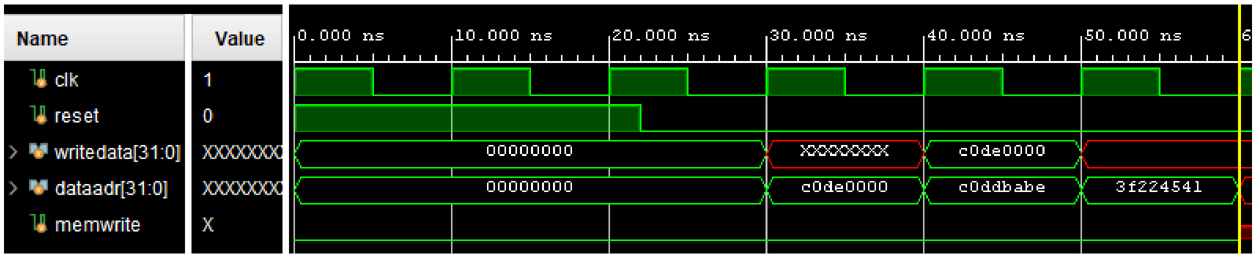
Note that while 5 bits in the lui instruction are don't-cares, this cannot be accurately represented in hexadecimal as 0x3CX1C0DE indicates an rt value of X0001 which is invalid.
However, the above code can still be tested by changing imem.sv to read binary instead, replacing $readmemh with $readmemb:

```
$readmemb("memfile.mem",RAM);
```

Thus memfile.mem becomes:

```
00100000000000000000000000000000
001111XXXXX000011100000011011110
00100000001000011011101010111110
00111000001000101111111111111111
```

The above program returns:



Where the final answer, 0x3F224541, shows the instruction works as intended.

For following testing sections, binary machine code will be used.

SRLV INSTRUCTION

Shift right logical variable is an r-type instruction which takes the value of two registers a and b, and shifts the value of a right logically (or zero-extended) by b bits. As SRLV is an r-type instruction, there is little modification to be done.

Aludec changes

Unlike previous instructions, maindec does not have to be changed to accommodate SRLV as SRLV is an r-type instruction, thus its opcode is 000000, which maindec can already parse by default.

The first change needed is in aludec, where the switch-case must be appended to support the SRLV function. Note that this again necessitates another unique ALUControl signal.

```
6'b101010: alucontrol <= 3'b111; // slt
6'b000110: alucontrol <= 3'b101; // srlv
default:   alucontrol <= 3'bxxx; // ???
```

All 8 possible ALUControl signals now in use: 000 for and, 001 for or, 010 for addition, 011 for XORI, 100 for LUI, 101 for SRLV, and 110 for subtraction, and 111 for SLT.

ALU changes

Similar to previous instructions, the ALU must change to accommodate the new ALUControl signal, 101.

```
case(alucontrol)
  3'b011: result = a ^ b;
  3'b100: result = b << 16;
  3'b101: result = a >> b;
```

a and b in this case represent the two registers involved, rs and rt. ≫ is the shift right logical operator in SystemVerilog.

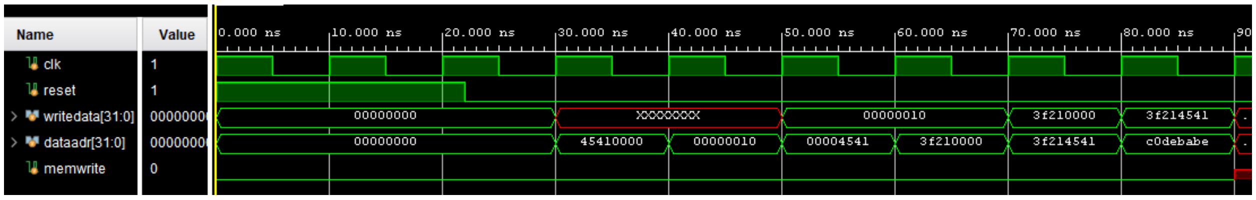
Note: this logic is incorrect and will be changed in a later section.

Testing

A simple program can be written to check the functionality of SRLV.

```
addi 0 0 0          001000 00000 00000 0000000000000000
lui 1 0x4541        001111 XXXXX 00001 0100010101000001
addi 2 0 0x10       001000 00000 00010 0000000000010000
srlv 1 1 2          000000 00010 00001 00001 XXXXX 000110
lui 2 0x3F21        001111 XXXXX 00010 0011111100100001
add 1 1 2           000000 00001 00010 00001 XXXXX 100000
xori 1 1 0xFFFF     001110 00001 00001 1111111111111111
#Final answer should be C0DE BABE
```

The program generates the following simulation:



Where the final answer, 0xC0DEBABE, shows the instruction works as intended.

BGTZ INSTRUCTION

Branch if greater than zero is an i-type pseudo-instruction, performing similarly to BEQ except instead of equals it checks greater than and instead of a register rt it always checks against zero. As an i-type instruction, BGTZ should get its own ALUControl signal, however as mentioned in the previous section, all 8 options for ALUControl have been exhausted. Thus, to accommodate extra instructions, the width of the ALUControl signal can be extended from 3 bits to 4 bits.

Maindec changes

As an i-type instruction, BGTZ has its own opcode, and thus must be accounted for in the maindec module's switch-case block. According to the specifications, its opcode is 0x1D or 0b011101.

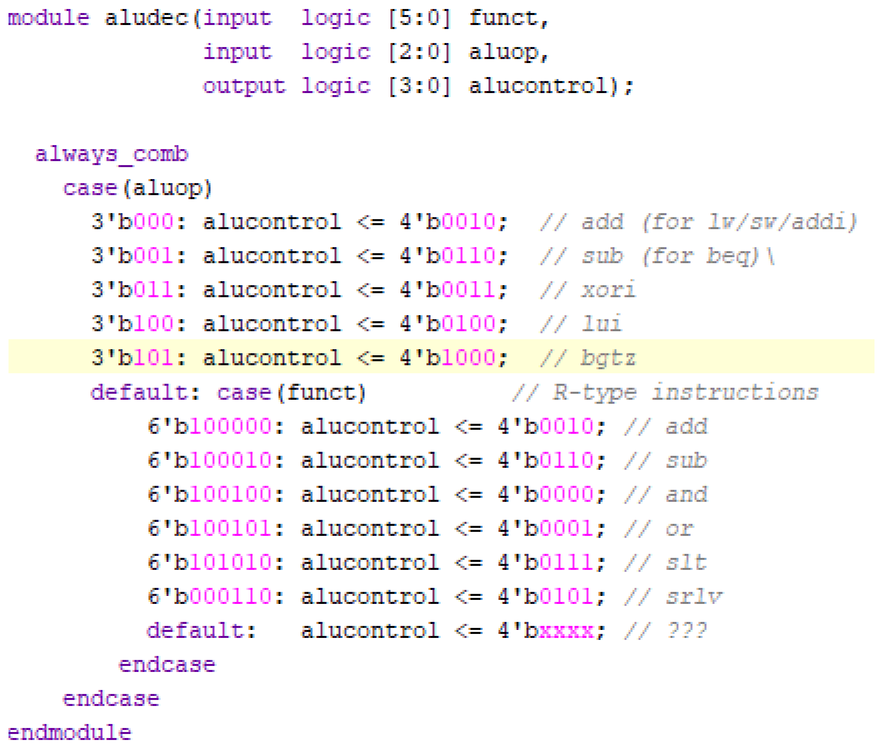
```
6'b000100: controls <= 10'b0001000001; // BEQ
6'b011101: controls <= 10'b0001000101; // BGTZ
```

The ALUOp bitstring for BGTZ is 101. Note that this is unique to BGTZ. Also note that the remaining signals are the same as BEQ. This is because BEQ and BGTZ are very similar, and their only differences are the operation performed and the two values the instruction compares.

Aludec changes

As discussed earlier, $\mathrm{ALUControl}$ is to be extended by 1 bit, of which the process begins at the aludec module. Similar to extending ALUOp, extending the width of $\mathrm{ALUControl}$ will also retroactively break any previous instructions, thus these instructions must also be modified to accommodate the extra bit.

Additionally, an extra check must be in the aludec module's switch-case module for the new ALUOp signal associated with BGTZ. All in all, the above changes lead to:

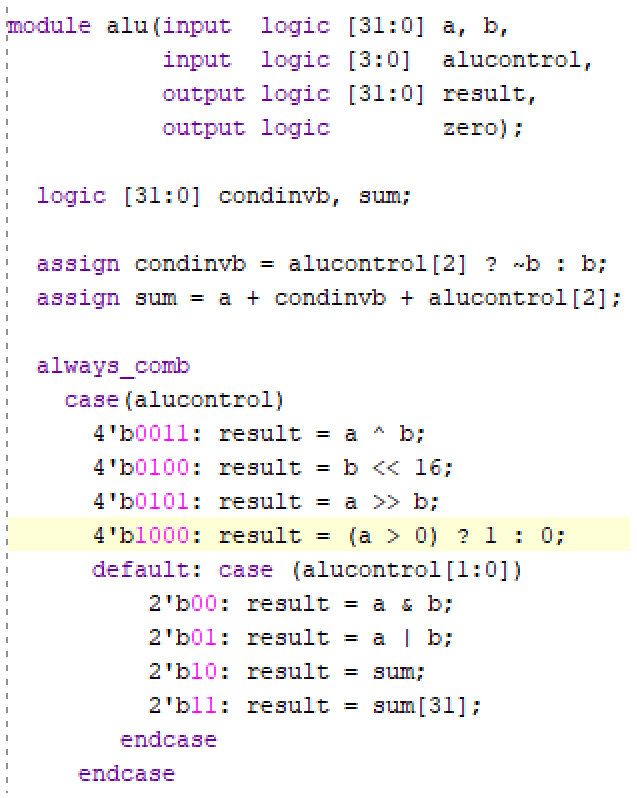


```
module aludec(input  logic [5:0] funct,
              input  logic [2:0] aluop,
              output logic [3:0] alucontrol);

  always_comb
    case(aluop)
      3'b000: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      3'b001: alucontrol <= 4'b0110;  // sub (for beq)\
      3'b011: alucontrol <= 4'b0011;  // xori
      3'b100: alucontrol <= 4'b0100;  // lui
      3'b101: alucontrol <= 4'b1000;  // bgtz
      default: case(funct)          // R-type instructions
          6'b100000: alucontrol <= 4'b0010; // add
          6'b100010: alucontrol <= 4'b0110; // sub
          6'b100100: alucontrol <= 4'b0000; // and
          6'b100101: alucontrol <= 4'b0001; // or
          6'b101010: alucontrol <= 4'b0111; // slt
          6'b000110: alucontrol <= 4'b0101; // srlv
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

Note that $\mathrm{alucontrol}$ and its associated signals are now all 4 bits wide, and an additional condition has been added to the switch-case block to account for $\mathrm{ALUOp} = 101$. The $\mathrm{ALUControl}$ signal associated with BGTZ has been arbitrarily chosen to be 1000, as this is again the lowest unused $\mathrm{ALUControl}$ signal.

ALU changes

As $\mathrm{ALUControl}$'s bit width has changed, so too does the ALU. Similarly, previous instructions must be modified to accommodate the new width as well as a condition to catch BGTZ's signal $\mathrm{ALUControl} = 1000$.



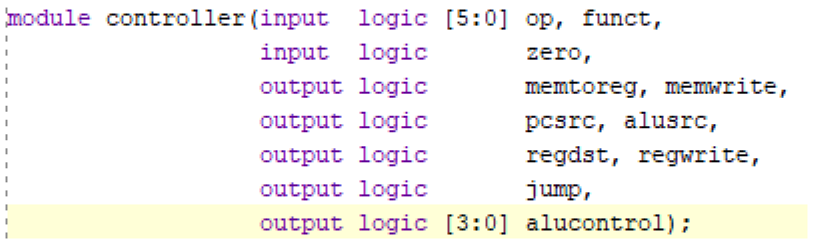
```
module alu(input  logic [31:0] a, b,
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;

  assign condinvb = alucontrol[2] ? ~b : b;
  assign sum = a + condinvb + alucontrol[2];

  always_comb
    case(alucontrol)
      4'b0011: result = a ^ b;
      4'b0100: result = b << 16;
      4'b0101: result = a >> b;
      4'b1000: result = (a > 0) ? 1 : 0;
      default: case (alucontrol[1:0])
          2'b00: result = a & b;
          2'b01: result = a | b;
          2'b10: result = sum;
          2'b11: result = sum[31];
        endcase
     endcase
```

While alucontrol and its associated signals have been modified to support 4 bits, the original set of instructions have not been modified. This is because of the nature of the switch-case: the original instructions are not affected by the addition of a new bit because they only need to check the first 3 bits. In fact, the addition of the 4th bit is only ever relevant to the newly added instructions, so it only has to be checked if the current instruction is new (which would be on the outer switch-case). For a new instruction, the inner switch-case is never touched, and for an old instruction, the outer switch-case doesn't matter.

Note that a branch is executed if $\text{zero} = 1$, or if $\text{result} = 0$. As such, result is assigned to an inline conditional: with a as the value of the register provided in the instruction, if $a > 0$, then result is set to 0, otherwise, result is set to 1. This behavior is the required behavior needed from BGTZ.

Wire changes

Because of the way the processor is created, the signal ALUControl does not only pass through maindec, aludec, and alu, but it is also passed around in other components, namely the modules controller, mips, and datapath. These must all be modified to accommodate the new 4-bit ALUControl.



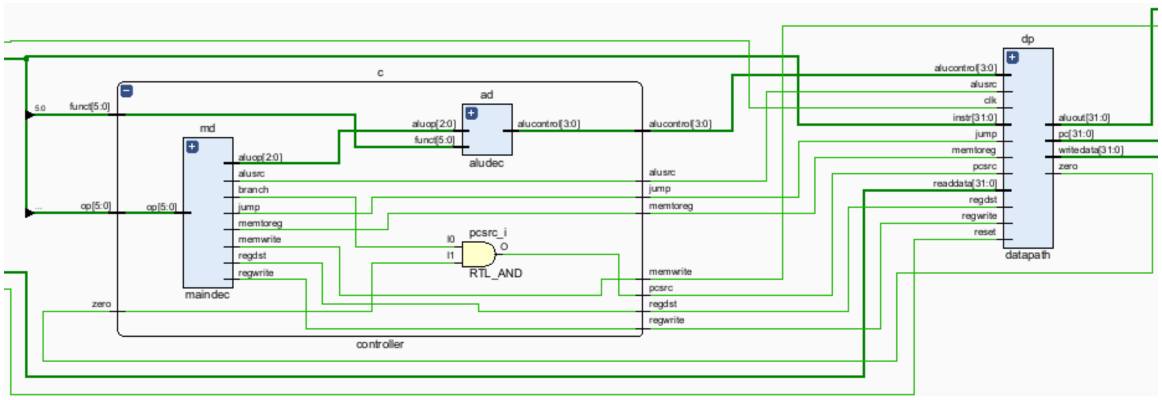
```
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,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

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

  logic       memtoreg, alusrc, regdst,
              regwrite, jump, pcsrc, zero;
  logic [3:0] alucontrol;
```

```
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [3:0]  alucontrol,
```

Testing

As a quick sanity check, Vivado's schematic function can be used to ascertain that the correct number of bits are being passed around between components.

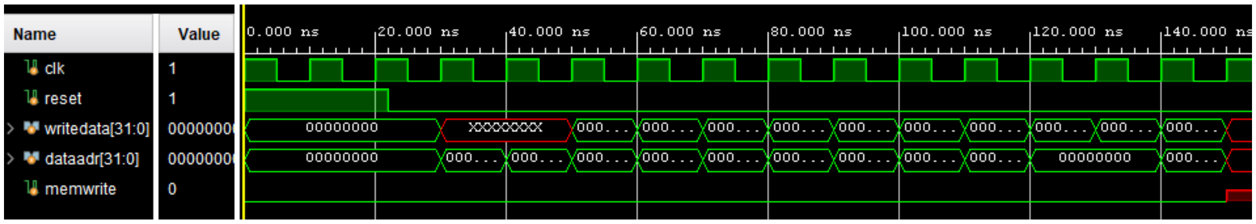


ALUControl appears to be 4 bits wide as intended.

A simple program can be created to check for the functionality of BGTZ:

```
addi 0 0 0          001000 00000 00000 0000000000000000
addi 1 0 0x1        001000 00000 00001 0000000000000001
addi 2 0 0x10       001000 00000 00010 0000000000010000
srlv 2 2 1          000000 00001 00010 00010 XXXXX 000110
bgtz 2 -2           011101 00010 00000 1111111111111110
#If the instruction works then the loop should end in 3 + 5(2) = 13 cycles.
```

Which creates the following simulation:



Name	Value
clk	1
reset	1
writedata[31:0]	0000000
dataadr[31:0]	0000000
memwrite	0

Note that 13 clock cycles pass between 20ns and 150ns, thus the instruction functions as intended.

LI INSTRUCTION

Load immediate is an i-type instruction that functions similarly to LUI, except it loads to the bottom 16 bits of a register instead of to the top. This is also equivalent to a zero-extended ADDI instruction (note that the immediate in ADDI is sign-extended).

Maindec changes

According to the specifications, the opcode of LI is 0x11 or 0b010001. This must be added to maindec as a condition in its switch-case block:

```
6'b001111: controls <= 10'b1010000100; // LUI
6'b010001: controls <= 10'b1010000110; // LI
```

As LUI is very similar to LI, they share the same signals barring ALUOp, of which the chosen arbitrary signal is 110.

Aludec changes

Aludec must also be able to accommodate the new signal for LI's ALUOp, 110.

```
3'b101: alucontrol <= 4'b1000;  // bgtz
3'b110: alucontrol <= 4'b1001;  // li
```

The associated ALUControl signal for LI is 1001.

ALU changes

The ALU must also be modified to process the new ALUControl signal.

```
4'b1000: result = (a > 0) ? 0 : 1;
4'b1001: result = {16'b0, b[15:0]};
```

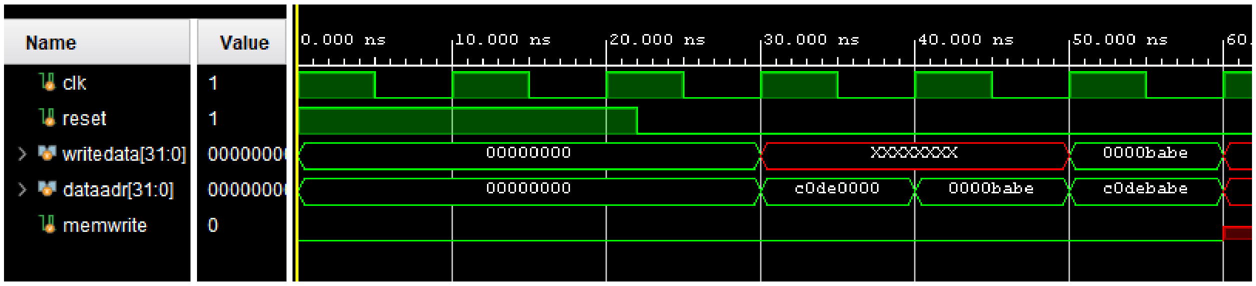
result is set to be a concatenation of 16 0 bits and the bottom 16 bits of b, which in this case is set to be the immediate. Note that a is not in the calculation for the result; this is because the register equivalent of a, rs, is set to be don't-cares, and thus it is irrelevant to the function of LI.

Testing

A simple program can be run to test the functionality of LI:

```
addi 0 0 0          001000 00000 00000 0000000000000000
lui 1 0xC0DE        001111 XXXXX 00001 1100000011011110
li 2 0xBABE         010001 XXXXX 00010 1011101010111110
add 2 1 2           000000 00001 00010 00010 00000 100000
#Final answer should be C0DE BABE
```

This creates the simulation:



The final ALU result is 0xC0DEBABE, thus the instruction works as intended.

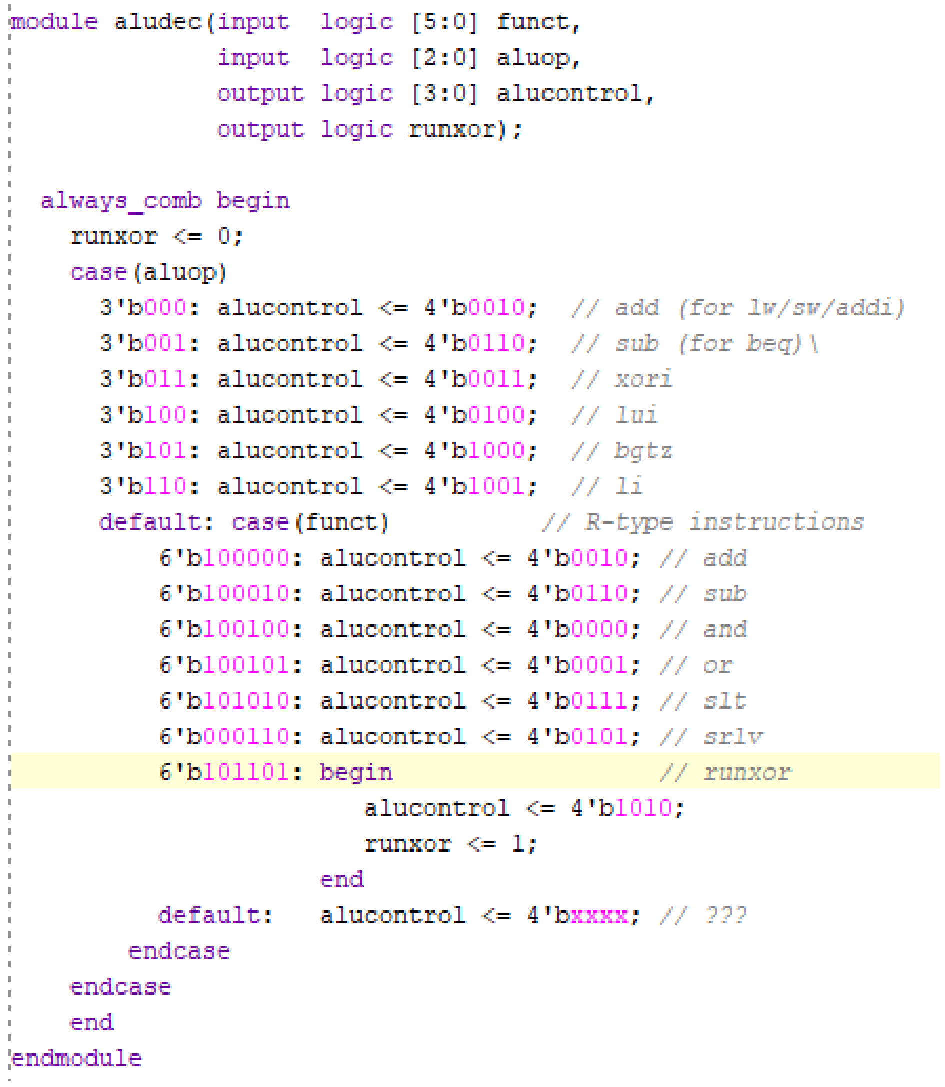
RUNXOR INSTRUCTION

Running xor is a custom r-type instruction which takes the value of a register a and performs bitwise XOR operations on every pair of bits in the value a, where the first bit of a remains the same, to create a new value b which is placed in some other register.
That is, the second highest bit of b is the highest and second highest bits of a xor'd together, the third highest of b is the second and third highest bit of a xor'd together, etc.

Aludec changes

While RUNXOR is an r-type instruction, it differs from normal r-type instructions because RUNXOR's destination register is set to be rt while normal r-type instructions have their destination registers set to rd. This cannot be solved in the maindec module without heavy modification as the maindec module does not have access to funct, which is the only way to differentiate RUNXOR from any other r-type instruction.

However, funct is accessible in a different component: the aludec module. Thus, in order to bridge the two together, a new output signal can be created in the aludec module called runxor, which defaults to being 0 but is set to 1 if the current instruction being processed is RUNXOR:



```
module aludec(input  logic [5:0] funct,
              input  logic [2:0] aluop,
              output logic [3:0] alucontrol,
              output logic runxor);

  always_comb begin
    runxor <= 0;
    case(aluop)
      3'b000: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
      3'b001: alucontrol <= 4'b0110;  // sub (for beq)\
      3'b011: alucontrol <= 4'b0011;  // xori
      3'b100: alucontrol <= 4'b0100;  // lui
      3'b101: alucontrol <= 4'b1000;  // bgtz
      3'b110: alucontrol <= 4'b1001;  // li
      default: case(funct)          // R-type instructions
          6'b100000: alucontrol <= 4'b0010; // add
          6'b100010: alucontrol <= 4'b0110; // sub
          6'b100100: alucontrol <= 4'b0000; // and
          6'b100101: alucontrol <= 4'b0001; // or
          6'b101010: alucontrol <= 4'b0111; // slt
          6'b000110: alucontrol <= 4'b0101; // srlv
          6'b101101: begin                  // runxor
                        alucontrol <= 4'b1010;
                        runxor <= 1;
                     end
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
    end
endmodule
```

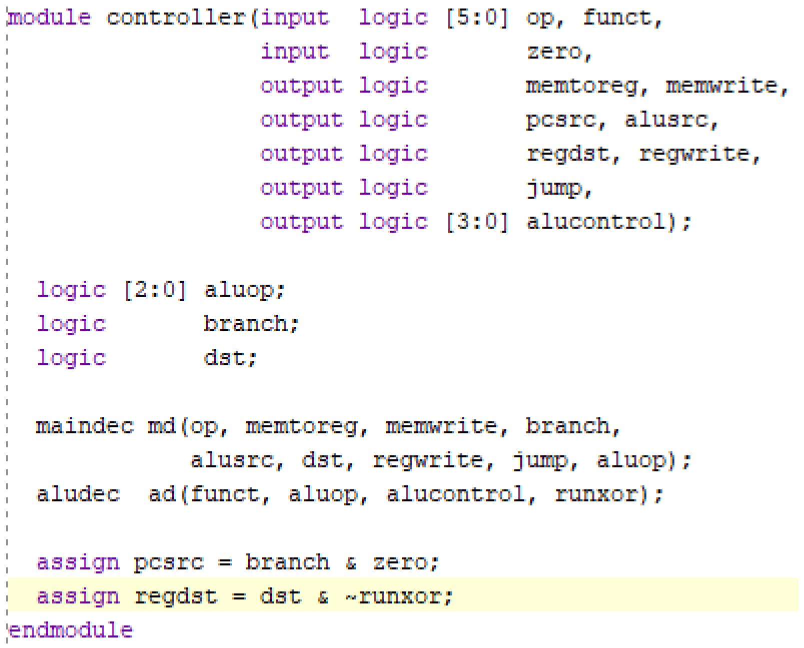
Note that all necessary changes have been committed:
1) the switch-case block now has a condition for the funct of RUNXOR, 101101
2) there now exists a new output node runxor which is set to 0 by default but changes to 1 if the current instruction is RUNXOR
3) when the current instruction is RUNXOR, ALUControl is set to a previously unused signal 1010.

Controller changes
Because of the creation of a new output signal in the aludec module, the controller module must also be modified to have its output signal regdst change depending on the value of runxor. This can be done through a simple and gate:

Implication	regdst	runxor	$\overline{\text{runxor}}$	regdst ∩ $\overline{\text{runxor}}$
Instruction is not r-type	0	0	1	0
Instruction is not r-type	0	1	0	0
Instruction is r-type but not RUNXOR	1	0	1	1
Instruction is RUNXOR	1	1	0	0

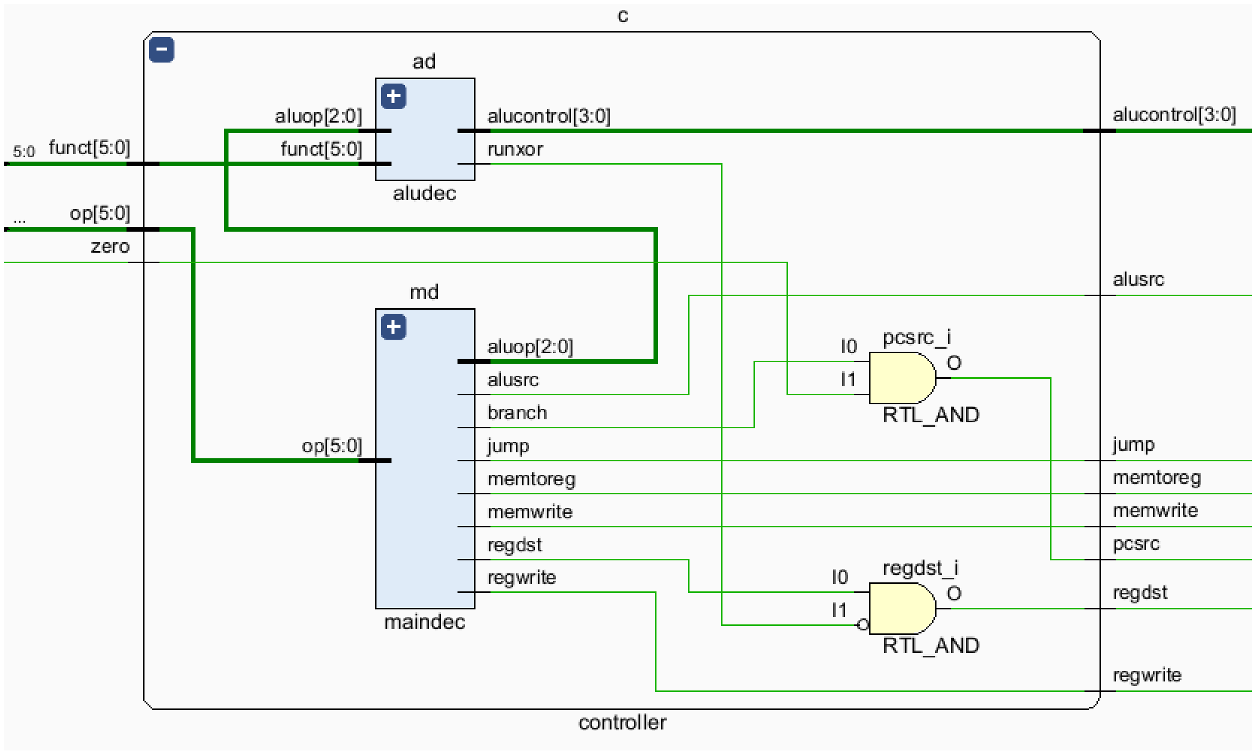
As seen in the table above, normal r-type instructions function normally, however the RUNXOR instruction sets the runxor signal to be 1, returning the final regdst signal back to 0 (that is, the destination register becomes rt instead of rd).

The above truth table can be implemented by modifying the controller module:



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

  logic [2:0] aluop;
  logic       branch;
  logic       dst;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, dst, regwrite, jump, aluop);
  aludec  ad(funct, aluop, alucontrol, runxor);

  assign pcsrc = branch & zero;
  assign regdst = dst & ~runxor;
endmodule
```

The new output of the aludec module is set to be runxor and a new wire is created named dst. The regdst output of maindec is now set to dst and the regdst output of the controller module is set to be dst ∩ $\overline{\text{runxor}}$. This is better visualized through Vivado's schematic function:



The figure above shows the new schematic of the controller module. Notice the presence of a new output signal from aludec, runxor, and the new signal attached to regdst, which is an AND gate as discussed earlier.

ALU changes

The majority of the work done for RUNXOR is performed in the ALU. This is first done by creating a new switch-case condition for the ALUControl signal of RUNXOR, 1010, and creating a new block in this condition which handles all of the necessary logic.

```
4'b1001: result = {16'b0, b[15:0]};
4'b1010: begin
          result[31] = a[31];
          for (int i = 30; i >=0; i--) begin
            result[i] = a[i] ^ a[i+1];
          end
         end
```

After the condition for RUNXOR is achieved (ALUControl = 1010), the MSB of result is set to the MSB of a, which is the value of rs. Then a for loop is created which cycles from the second most significant bit of result all the way to the LSB, and assigns to them the respective xor'd bits from a.

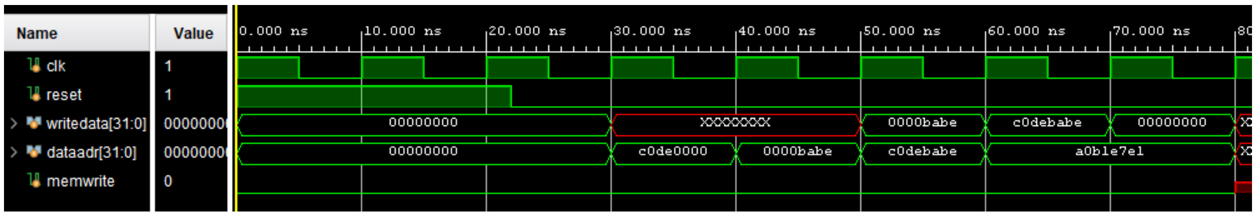
Testing

The specifications indicate that $runxor(\text{0xC0DEBABE}) = \text{0xA0B1E7E1}$, so a testing program can be created through with that in mind:

```
addi 0 0 0          001000 00000 00000 0000000000000000
lui 1 0xC0DE        001111 XXXXX 00001 1100000011011110
li 2 0xBABE         010001 XXXXX 00010 1011101010111110
add 2 1 2           000000 00001 00010 00010 00000 100000
runxor 2 2          000000 00010 00010 XXXXX XXXXX 101101
add 2 2 0           000000 00010 00000 00010 00000 100000
#Final answer should be A0B1 E7E1
```

Notice that the final instruction adds a register and 0: this is so the value of the register is visible in dataadr, which helps verify if the RUNXOR instruction saved its result to the correct register.

The above program creates the following simulation:



Note that the final two clock cycles have the correct value, 0xA0B1E7E1, thus the result of the instruction is correct and the result is saved to the correct register.

OTHER CHANGES

The correct function of SRLV is rt ≫ rs, or in the alu module, b ≫ a. This is in contrast to what was previously written in the SRLV instruction section.

SRLV can be corrected by modifying the alu module as so:

```
case(alucontrol)
  4'b0011: result = a ^ b;
  4'b0100: result = b << 16;
  4'b0101: result = b >> a;
```