# HDL Project

Joshua Allyn Marck Espartero

2019-10009



# 1 Introduction

## 1.1 Objective

The aim of this project is to modify a given MIPS Single Cycle Processor System Verilog code to be able to execute specified additional instructions.

## 1.2 Specifications

The project specifications state that we are to extend the System Verilog code to be able to execute instructions that fall under three (3) categories.

1. **Normal instructions** - these are instructions that are in the instruction set, but

just not currently implemented in the design given in Laboratory Exercise 12.

1. $nor$
2. $lui$
3. $sll$

2. **Pseudo-instructions** - these are instructions that are not in the instruction set,
   but are recognized by the assembler and are converted or assembled as a sequence
   of actual instructions; these can also be found in the MIPS Green Sheet.
   1. $blt$
   2. $li$
3. **Custom instructions** - these are instructions that are not in the MIPS Green
   Sheet.
   1. $mix$

## 1.3 Others

# 2 HDL Code Design Changes

In this section of the documentation, we will divide the changes into two subsections: Adding  More Instructions and Implementing New Instructions. The former will discuss how we changes the modules to allow the des implement more instructions than the already given ones. The latter will discuss how we

## 2.1 Adding More Instructions

Given the current design of the Single Cycle Processor System Verilog Code,  the modules only allow a certain number of instructions to be implemented in the

design. We will be discussing which components and logic variables  did we alter to allow us to implement additional instructions on top of the already working ones.

These are the components that will be discussed in this section:

1. $aludec$
2. $maindec$
3. $controller$
4. $datapath$
5. $mips$
6. $alu$

## 2.1.1 aludec

First we discuss our changes we need to implement with the $aludec$ component.

Here is a snippet of the original $aludec$ System Verilog code:

```
////////////
`timescale 1ns / 1ps
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [2:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b010;  // add (for lw/sw/addi)
      2'b01: alucontrol <= 3'b110;  // sub (for beq)
      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
```

From the code above we see that our $alucontrol$ logic is only 3 bits wide. This means that there only exists $2^3$ unique $alucontrol$ codes that we can use. This is not enough since we see that already most  of them are already used up. So the solution to this is to use a $4$-bit wide $alucontrol$ logic variable.

Along with this, we also need to extend the $2$-bit wide $aluop$ since we need to add 3 more non R-Type instructions to be read in this system. So in our solution we implement a $3$-bit wide $aluop$.

Here is the $aludec$ component implemented in our solution:

```
////////////
`timescale 1ns / 1ps
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'b1011;  // lui
      3'b110: alucontrol <= 4'b0101;  // blt
      3'b111: 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'b100111: alucontrol <= 4'b1100; // nor
          6'b000000: alucontrol <= 4'b1000; // sll
          6'b110011: alucontrol <= 4'b1110; // mix
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase

endmodule
```

For the old instructions, we just appended a 0 as their 4th bit for the $alucontrol$ and as their 3rd bit for the $aluop$.

This already code already contains the assigned $aluop$ codes and $alucontrol$ codes our solution assigned for the new instructions to be implemented.

## 2.1.2 controller

Due to the changes we implemented on our $alucontrol$ and $aluop$. We needed to change the size of the output logic argument $alucontrol$ accordingly (now 4-bits wide). Here the declaration of the logic variable $aluop$ is changed accordingly as well (now 3-bits wide).

Here is the original controller implementation:

```
`timescale 1ns / 1ps
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 [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

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

  assign pcsrc = branch & zero;
endmodule
```

While here is the newly implemented version:

```
`timescale 1ns / 1ps
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;

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

  assign pcsrc = branch & zero;
endmodule
```

## 2.1.3 maindec

Now we discuss the changes we need to do in our $maindec$ component.

The main change implemented here is the $control$ logic variable size change as a consequence of the changed $aluop$. We extend $control$ to be a $10$-bit wide logic variable. We also note the changed $aluop$ size declared in our output variables.

One huge effect this has on our instruction control codes is that we need to alter the control codes of the old instructions. We just insert a 0-bit between the 2nd and 3rd bits. Then we can add new control codes for our new instructions.

Here is the original $maindec$ System Verilog code:

```
`timescale 1ns / 1ps
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

  logic [8:0] controls;

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

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

Here is the newly implemented maindec:

```
`timescale 1ns / 1ps
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'b000010: controls <= 10'b0000001000; // J
      6'b001111: controls <= 10'b1010000011; // LUI
      6'b011111: controls <= 10'b0001000110; // blt
      6'b010001: controls <= 10'b1010000111; // li
      default:   controls <= 10'bxxxxxxxxxx; // illegal op
    endcase
endmodule
```

### 2.1.4 datapath

For the *datapath* component, the only relevant thing to this subsection that we implemented is the change in size of the *alucontrol* input logic as a consequence of the changes we did earlier.

Here is the old code for the original  input and output arguments:

```
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [2:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);
```

While our new *datapath* contains this:

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

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

## 2.1.5 mips

For the *mips* component we only needed to change size of the declaration of the *alucontrol* variable (now 4-bits wide). Again, this is a consequence of the changes we did earlier.

Here is the original *mips* System Verilog Code:

```
`timescale 1ns / 1ps
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 [2:0] alucontrol;

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

And here the implemented *mips* System Verilog Code:

```
`timescale 1ns / 1ps
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;

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

### 2.1.6 alu

The same change is done to our $alu$ component. We change the size of the input logic argument $alucontrol$ (now 4-bits wide).

Here is the original code for the argument of the $alu$ module:

```
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  c,
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);
```

Here is the implemented code for the argument of the $alu$ module:

```
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  c,
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);
```

## 2.2 Implementing New Instructions

On this section of our documentation.

We are gonna discuss how the specified instructions that we are ought to implement in this project.

We have six (6) instructions that we are going to discuss in this section.

1. $nor$
2. $lui$
3. $sll$
4. $li$
5. $blt$
6. $mix$

## 2.2.0 Testbench

To test the aforementioned instructions' implementations we use this testbench:

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

  logic        clk;
  logic        reset;

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

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

  // initialize test
  initial
    begin
      reset <= 1; #20; reset <= 0; // reset on for first two clock cycles
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5; // one clock cycle = 10ns
    end

endmodule
```

This will generate the clock to load the instructions located in our instruction memory.

## 2.2.1 nor

In this subsection we will be discussing how the $nor$ instruction or the not $or$ instruction is implemented.

We first needed to encode a unique code to our ALU decoder for this. So we added a specific line of code that will handle the processing of the $funct$ field of the instruction so that when it is equal to the specified function code in the MIPS Green Sheet of the $nor$ ($6'b100111$), it will give us a unique $alucontrol$ value we set to be $4'b1100$.

So we added a line in the default case of our ALU decoder since $nor$ is an R-Type instruction and  it is identified according to its $funct$ field.

Here is the line of coded added for the $nor$ instruction in our $aludec$ component:

```
case(aluop)
      ...
      default: case(funct)
          ...
          6'b100111: alucontrol <= 4'b1100; // nor
          ...
        endcase
    endcase
```

Note that we do not need to change something in our $maindec$ module since the $nor$ instruction will go to the case that handles R-Type instructions as it is an R-Type which means its opcode is $000000$.  Thus getting control values that of an R-Type instruction.

For how it is actually implemented is done in the $alu$ component.

Here we add a case for our $alucontrol$ which we assigned for this instruction $4'b1100$ . We then set the result to the $and$ of the negation of our inputs $a$ and $b$ which corresponds to the values in the specified registers of the instructions' $rs$ and $rt$ field respectively. We note that, $\sim a \& \sim b = \sim (a|b)$ as De Morgan's Law states. Note that we're not using the shamt field here $c$ so it can contain any value and the result of our execution wont change

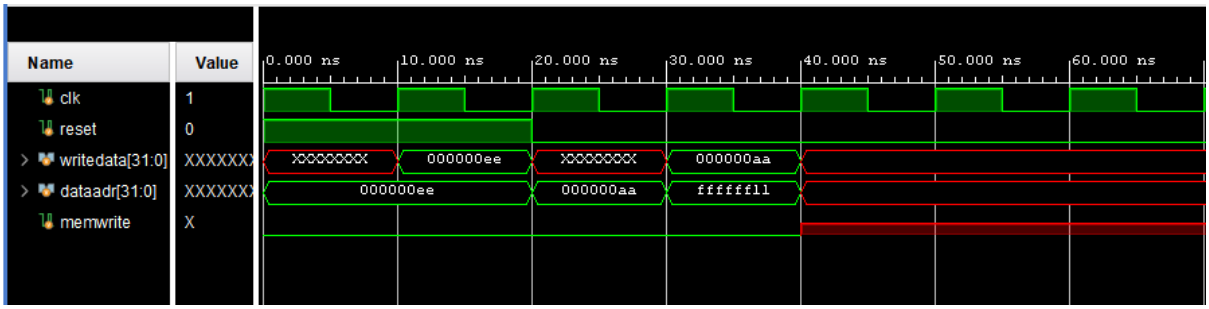
Here is the snippet of the code added for the $nor$ instruction in our $alu$ component:

```
case(alucontrol)
    ...
    4'b1100: result = ~a & ~b; //nor
    ...
    endcase
```

Here is a test code used to test whether the implementation is indeed correct

```
nor:
addi $1, $0, 0xEE
addi $2, $0, 0xAA
nor $3, $1, $2 // should output 0xFFFFFF11

hex equivalent:
200100ee
200200aa
00221867 (0000 0000 0010 0010 0001 1000 0110 0111)
with shamt = 00001 (DONT CARE)
```

Here is a snippet of the waveforms of the simulation:




We see that the last instruction (nor) indeed gives out our desired result $FFFFFF11_{hex}$.

## 2.2.2 lui

In this subsection we will be discussing how the $lui$ instruction or the load upper immediate implemented.

In contrast to the $nor$ instruction, the $lui$ instruction is actually first recognized through its opcode. With this, we need to add a case in our $maindec$ component that will cater to this specific instruction. Since this is an I-Type instruction. Our controls would be similar the already installed $addi$ instruction. We follow the controls of the $addi$ instruction and we just need to change the $aluop$ code to a unique one. We assign it to be $3'b011$. So we add a case wherein when we encounter the *op* (opcode) $6'b001111$ we set the controls to $10'b1010000011$ with the last three bits to be our $aluop$.

Here is a snippet of the code on how it was implemented in our $maindec$ module:

```
always_comb
    case(op)
      ...
      6'b001111: controls <= 10'b1010000011; // LUI
      ...
    endcase
```

We then need to process this $aluop$ in our $aludec$ module. So there we also assign a specific case that will cater to the unique $aluop$ that we assigned to the $lui$ instruction. When it encounters an $aluop$ that is $3'b011$, we design it to give out a unique $alucontrol$ that we assign to be $4'b1011$ for this instruction.

Here is a snippet of the code on how it was implemented in our $aludec$ module:

```
always_comb
    case(aluop)
      ...
      3'b011: alucontrol <= 4'b1011;  // lui
      ...
    endcase
```

Now on we proceed on how do we actually do the operation in our design. We go to the $alu$ module where we also assign a unique case for the $alucontrol$ that we have for the instruction. When our $alucontrol$ value is encountered, we set the $result$ to be the value in the register specified in the $rt$ field of the instruction, which we have as input $b$, multiplied by $2^{16}$ which will shift the bits of our value $16$ times to the left making it now occupy the upper 16 bits of our resulting value. Which the essence of this instruction. Note that we never use $a$ in computing for our result since it

corresponds to our *dont care* field $rs$. So even if we change the value in $rs$ our $result$ will not change.

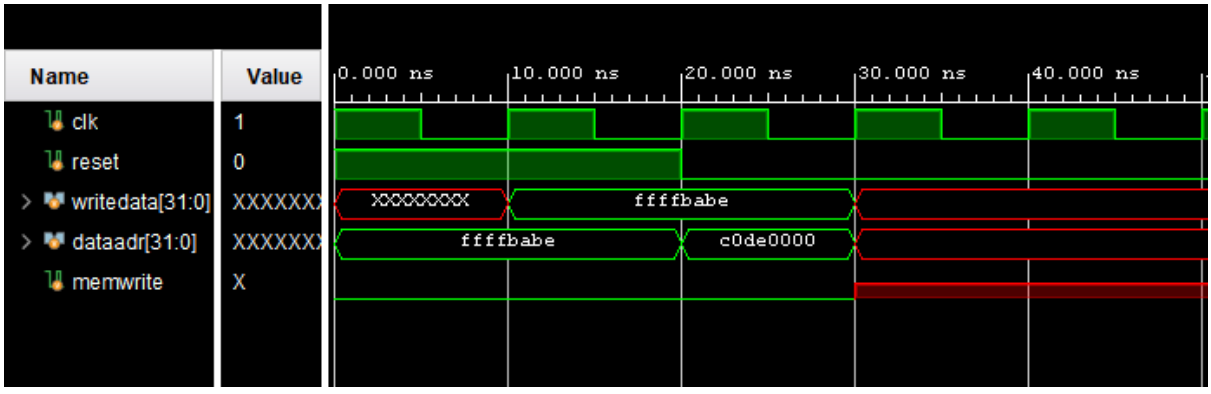
Here is a snippet of the code on how it was implemented in our $alu$ module:

```
always_comb
    case(alucontrol)
    4'b1011: result = b*2**16; //LUI
    ...
    endcase
```

Here is a test code used to test whether the implementation is indeed correct:

```
lui:
addi $1, $0, 0xBABE
lui $1, 0xC0DE

hex equivalent:
2001babe
3d41c0de (00111101010000011100000011011110 with dontCare (rs) = 01010)
```

Here is a snippet of the waveforms of the simulation:




We see that the second instruction (lui) indeed produces the correct alu result $c0de0000_{hex}$. Hence our implementation works.

## 2.2.3 sll

In this subsection we will be discussing how the $sll$ instruction or shift left logical instruction is implemented.

Just like the $nor$ instruction , we first needed to encode a unique code to our ALU decoder for this. So we added a specific line of code that will handle the processing of the $funct$ field of the instruction so that when it is equal to the specified function code in the MIPS Green Sheet of the $nor$ ($6'b000000$), it will give us a unique $alucontrol$ value we set to be $4'b1000$.

Here is a snippet of the code on how it was implemented in our $aludec$ module:

```
always_comb
    case(aluop)
      ...
      default: case(funct)          // R-type instructions
          ...
          6'b000000: alucontrol <= 4'b1000; // sll
          ...
        endcase
    endcase
```

Note that we do not need to change something in our $maindec$ module since the $sll$ instruction will go to the case that handles $op = 6'b000000$ as it is an R-Type which means its opcode is $000000$. Thus getting control values that of an R-Type instruction.

For how it is actually implemented is done in the $alu$ module.

Since for $sll$ we actually need to read the shamt field. We add an argument logic input [4:0] $c$ which will be where the shamt value will go to.

```
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  c, // new added input
           input  logic [3:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);
```

For this we need to edit the instantiation of the $alu$ module in our $datapath$ module. We need to put the shamt input in instantiation which will be the [10:6] bits of our instruction.

Here is a snippet on how this was implemented in our $datapath$ module:

```
`timescale 1ns / 1ps
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [3:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);

  ...
  alu         alu(srca, srcb, instr[10:6], alucontrol, aluout, zero);
endmodule
```

Here we add a case for our $alucontrol$ which we assigned for this instruction $4'b1000$ . We then set the $result$ to the value of the register specified in the $rt$ field of the instruction, which is accepted in our $alu$ module as the input $b$ , multiplied by $2^{shamt}$. $Shamt$ being set to be the newly added input $c$. Note that we never use $a$ in computing for our result since it corresponds to our *dont care* field $rs$. So even if we change the value in $rs$ our $result$ will not change.

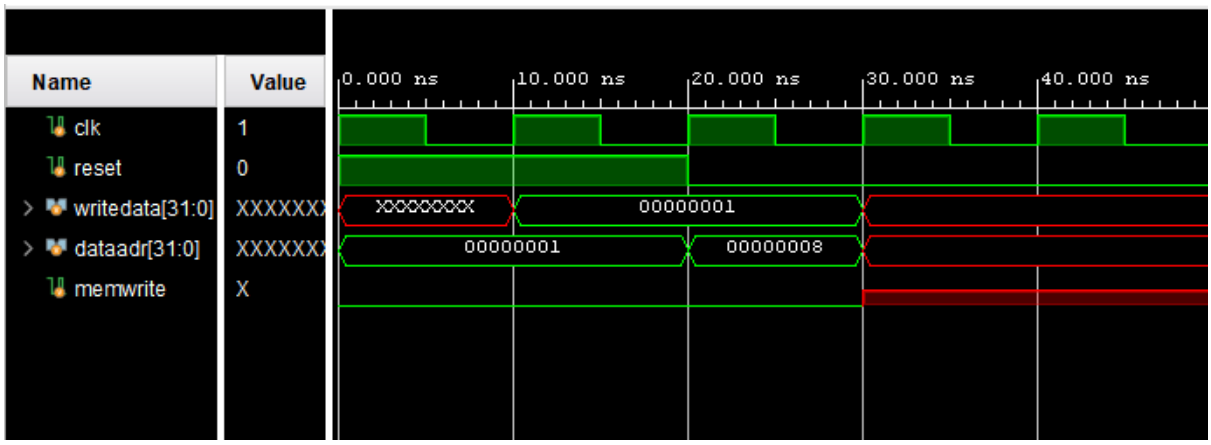
Here is a snippet on how this implemented on our $alu$ module:

```
module alu...
...
always_comb
    case(alucontrol)
    ...
    4'b1000: result = b*(2**c); // sll
    ...
    endcase
```

Here is a test code used to test whether the implementation is indeed correct

```
sll:
addi $1, $0, 1
sll $2, $1, 3

hex equivalent:
20010001
002110c0 (0000 0000 0010 0001 0001 0000 1100 note: rs = 1 dont cares)
```

Here is a snippet of the waveforms of the simulation:




We see that the second instruction (sll) indeed gives out our desired result $8_{hex}$. Hence our implementation works.

## 2.2.4 li

In this subsection we will be discussing how the $li$ instruction or load immediate pseudo-instruction is implemented.

Since this instruction is not specified in the MIPS Green Sheet we base on the specifications given by the project details

1. opcode is $0x11$.
2. target register is in bits 20-16 ($rt$)
3. $rs$ (bits 25-21) are $X$s (Don't cares)
4. immediate field (bits 15-0) contain values to be stored to rt; when used in assembly language, assume value to be stored always no longer than 16 bits; logically- or zero-extend the 16-bit value to 32 when storing to register.

Basing from these specifications, we see that we are making an I-Type instruction. So for the non-ALU changes we follow what we did with the already implemented I-Type instructions like what we did with the $lui$ instruction.

Hence, we start with how I-Type instructions are recognized which is by its opcode. So we go to our $maindec$ module and create a unique case for this new instruction. When it encounters an opcode $0x11$ or $'6b010001$ it will return controls that is similar to the $addi$ instruction but with the $aluop$ to be a unique one, which we set to be $0b'111$. So we will  be assigning$controls = 10'b1010000111$ for our $li$ instruction.

Here is a snippet of how this was  implemented in our $maindec$ module:

```
module maindec...
...
always_comb
    case(op)
      ...
      6'b010001: controls <= 10'b1010000111; // li
      ...
    endcase
```

We then process this $aluop$ in our $aludec$ module. We again create a new case for this specific $aluop$ and give it a unique $alucontrol$. In our design we gave it the unique $alucontrol$, $4'b1001$.

Here is a snippet of how this was implemented in our $aludec$ module:

```
module aludec...
...
  always_comb
    case(aluop)
      ...
      3'b111: alucontrol <= 4'b1001;  // li
      ...
    endcase
endmodule
```

For how it is actually implemented is done in the $alu$ module.

We like the other I-Type instructions we have the input $b$ as the immediate value extracted from the instruction itself. This is because of the control values we had earlier specifically, $alusrc = 1$. So we set the result equal to a zero-extended $b$

which we do by concatenating 16 0 bits to it. Note that we do not use $a$ (rs, dont care) in computing our $result$ so what ever the contents of our $rs$ field is, it wont change $result$.

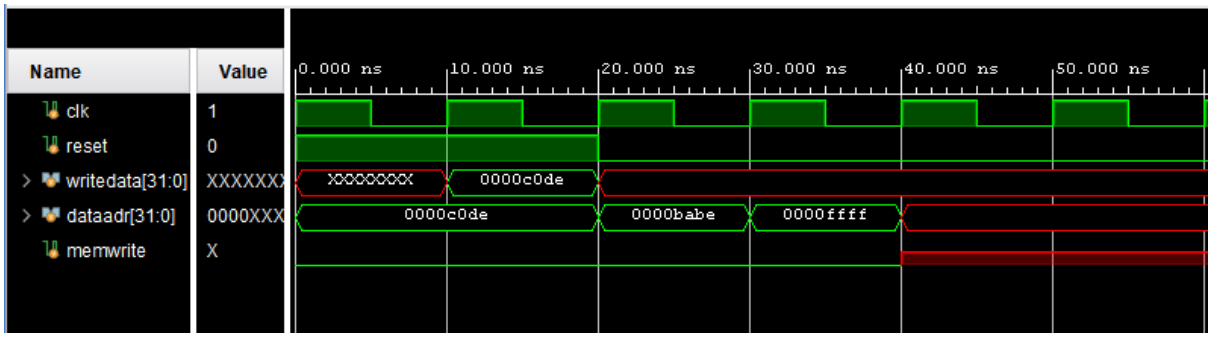
Here is a snippet on how this was implemented in our $alu$ module:

```
module alu...
...
  always_comb
    case(alucontrol)
    ...
    4'b1001: result = {4'h0000, b[15:0]};  //li
    ...
    endcase
  ...
endmodule
```

Here is a test code used to test whether the implementation is indeed correct

```
li:
li $1, 0xC0DE
li $2, 0xBABE
li $3, 0xFFFF

hex equivalent:
4481c0de (01000100100000011100000011011110 with dontCare rs = 00100)
46a2babe (01000110101000101011101010111110 with dontCare rs = 10101)
4703ffff (01000111000000111111111111111111 with dontCare rs = 11000)
```

Here is a snippet of the waveforms of the simulation:




We see that we do produce in our $result$ for all three $li$ instructions the zero-extended immediate values. Hence we know that our implementation works.

## 2.2.5 blt

In this subsection we will be discussing how the $sll$ instruction or shift left logical instruction is implemented.

Since this instruction is not specified in the MIPS Green Sheet we base on the specifications given by the project details

1. blt - same format as beq, but opcode is now $1F_{hex}$.

Basing from these specifications, we see that we are making an I-Type instruction specifically a branch type. So for the non-ALU changes we follow what we did with the already implemented I-Type instructions like what we did with the $lui$ instruction.

Hence, we start with how I-Type instructions are recognized which is by its opcode. So we go to our $maindec$ module and create a unique case for this new instruction. When it encounters an opcode $0x1F$ or $'6b011111$ it will return controls that is similar to the $beq$ instruction (since it is stated to be the same format with beq) but with the $aluop$ to be a unique one, which we set to be $0b'110$. So we will be assigning $controls = 10'b0001000110$ for our $blt$ instruction. We note that for the other controls aside from $aluop$ we only have $branch = 1$ while the others are all 0s.

Here is a snippet of how this was implemented in our $maindec$ module:

```
module maindec...
...
always_comb
    case(op)
      ...
      6'b011111: controls <= 10'b0001000110; // blt
      ...
    endcase
```

We then process this $aluop$ in our $aludec$ module. We again create a new case for this specific $aluop$ and give it a unique $alucontrol$. In our design we gave it the unique $alucontrol$, $4'b0101$.

Here is a snippet of how this was implemented in our $aludec$ module:

```
module aludec...
...
  always_comb
    case(aluop)
      ...
      3'b110: alucontrol <= 4'b0101;  // blt
      ...
    endcase
endmodule
```

We then inspect how the $beq$ instruction works. We see that there are two main key-players for branching to occur. First, is the $pcsrc$ variable located in our $controller$ module:

```
`timescale 1ns / 1ps
module controller...
  ...
  assign pcsrc = branch & zero;
endmodule
```

This turns into 1 if both the $branch$ control variable is equal to 1 and the logic variable $zero$ is also equal to one. This becomes the control signal whether we shall take $pc + 1$ or set it to offset of the branch which is when a branch occurs.
The second key-player we have here is the $zero$ variable that only turns 1 if the $result$ output of the operation $alu$ is equal to 0. So in our implementation we will take advantage of this.

We now proceed to the $alu$ module:

Note that since we have $alusrc = 0$. Inputs $a$ and $b$ are the values in the registers specified in the $rs$ and $rt$ field respectively. For $blt$ we only branch if $a < b$ and for us to check this, we can use the currently implemented mechanism for subtraction in our $alu$ module. We have two lines of code for that:

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

It checks if $alucontrol[2]$ is equal to one, then we set $condinvb$ equal to the flipped version of $b$ else we just use the real value of $b$ then we use that to compute the $sum$ of $a$ and $b$. We note the $+alucontrol[2]$ which is equal to $1$ when we want to do a subtraction operation. This is exactly how we find the negative of an number in $2$'s complement. Flip the bits and add $1$. So essentially, when $alucontrol[2] = 1$ our $sum = a - b$.

To take advantage of this, we strategically set our $alucontrol$ to be $4'b0101$ with $alucontrol[2] = 1$. This will make our $sum = a - b$. To check if $a < b$ we only need to check the $MSB$ of our $sum$ if it is equal to $1$ which is true for numbers in their $2$'s complement. Note that if $a < b$ we need to set $result = 0$. So we just set it as $!sum[31]$ since if $sum[31]$ is $1$ it means $a < b$ and we need $result = 0 = !(1)$. Otherwise, if it is not negative ($sum[31] = 0$) we need to set it to a number not equal to $0$, conveniently equal to $1$.

Here is a snippet on how this was implemented in our $alu$ module:

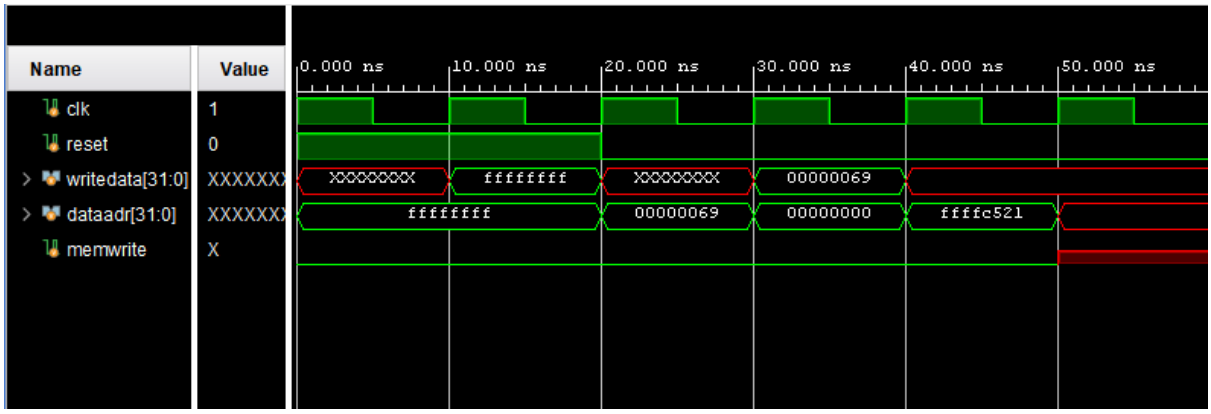
```
module alu...
...
  assign condinvb = alucontrol[2] ? ~b : b;
  assign sum = a + condinvb + alucontrol[2];

  always_comb
    case(alucontrol)
    ...
    4'b0101: result = sum[31] ? 0 : 1;  //blt
    ...
    endcase
  ...
endmodule
```

Here is a test code used to test whether the implementation is indeed correct

```
blt:
addi $1, $0, 0xFFFF   // $1 = -1
addi $2, $0, 0x69     // $2 = 0x69
blt $1, $2, branch    // 1<2 TRUE
addi $1, $0, 0x6969   //should skip
addi $2, $0, 0xb0b0   //should skip
```

```
branch:
addi $3, $0, 0xc521 // should show 0xFFFFC521

hex equivalent:
2001ffff
20020069
7c220002  (blt 01111100001000100000000000000010)
20016969
2002b0b0
2003c521
```

Here is a snippet of the waveforms of the simulation:




We see that the 3rd instruction shows $0$ in our alu result which is how we implemented our $blt$ instruction in our $alu$ module. Two instructions after that was skipped and it just showed the last instruction. Which means our implementation indeed works.

## 2.2.6 mix

In this subsection we will be discussing how the $mix$ instruction is implemented.

Since this instruction is not specified in the MIPS Green Sheet we base on the specifications given by the project details

1. Bits 31-26 - opcode = $00_{hex}$
2. Bits 25-21 - rs
3. Bits 20-16 - rt
4. Bits 15-11 - rd
5. Bits 10-6 - Xs (Don't cares - not necessarily zeroes)
6. Bits 5-0 - funct = $33_{hex}$

Operation

1. R[rd[31:16]] = R[rs[31:16]]
2. R[rd[15:0]] = R[rt[15:0]]

From these details we can see that we are dealing with an R-Type instruction since it has that opcode. So to in our $maindec$ module, it will be given a set of controls (including aluop) of that of an R-Type instruction. Hence to recognize that we're dealing with a $mix$ instruction we make a new case in our $aludec$ when it encounters the specified value for the $funct$ field. We assign the $alucontrol = 4'b1110$ if we encounter $funct = 6'b110011$ ($33_{hex}$).

Here is the snippet of code on how this was implemented in our $aludec$ module:

```
case(aluop)
      ...
      default: case(funct)
          ...
          6'b110011: alucontrol <= 4'b1110; // mix
          ...
        endcase
    endcase
```

For how it is actually implemented is done in the $alu$ component.

Here we add a case for our $alucontrol$ which we assigned for this instruction $4'b1110$ . Note that we have $alusrc = 0$, hence, $a$ and $b$ are the values of the registers specified in the $rs$ and $rt$ fields respectively. So we set our result to be the concatenation of the upper 16 bits of $a$ and the lower 16 bits of $b$. Which is the essence of this instruction. Note that we never used $c$ (shamt) in computing for our $result$. Hence, what ever the contents of our shamt field is, it wont affect our alu result.

Here is the snippet of the code added for the $mix$ instruction in our $alu$ component:

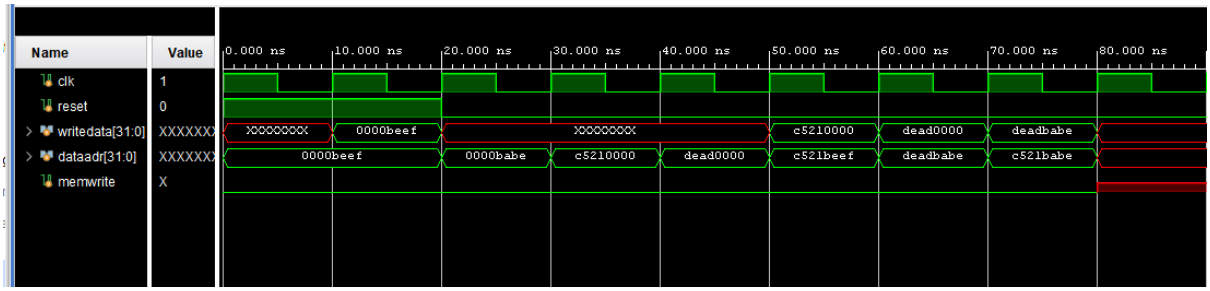
```
module alu...
  ...
  always_comb
    case(alucontrol)
    ...
    4'b1110: result = {a[31:16], b[15:0]}; // mix
    default: case (alucontrol[1:0])
                ...
              endcase
```

```
    endcase

  assign zero = (result == 32'b0);
endmodule
```

Here is a test code used to test whether the implementation is indeed correct

```
mix:
li $1, 0xBEEF
li $2, 0xBABE
lui $3, 0xC521
lui $4, 0xDEAD
or $5, $1, $3   //0xC521BEEF
or $6, $2, $4   //0xDEADBABE
mix $7, $5, $6  //0xC521BABE

hex equivalent:
4481beef
47e2babe
3c83c521
3e04dead
00232825
00443025
00a63b33 (mix: 00000000101001100011101100110011 with dontCare shamt = 01100)
```

Here is a snippet of the waveforms of the simulation:




We see that the last instruction $(mix)$ produced the result that we need according to the specs. It concatenated the upper 16 bits of our register specified in the $rs$ field and the lower 16 bits of the register specified in the $rt$ field of our instruction. Hence, we confirm that our implement does indeed work.