CS 21 PROJECT 2

MIPS Single Cycle Processor Extension

Submitted by: Joshua L. Felipe
Section: CS21 Lab3

Video Link:

https://drive.google.com/file/d/1XUwxjPFHqqG-MJURATr8YwNNgsz9uws2/view?usp=sharing

Introduction

A single cycle processor is a microarchitecture that takes instructions exactly one clock cycle at a time. It makes use of a datapath and control to execute the instructions properly. The datapath includes the instruction memory, register file, data memory, ALU and other components. This allows the processor to compute the necessary instructions. An illustration of a single cycle processor is represented in Figure 1. Meanwhile, the control sends out signals to the hardware components of the datapath which either activate or deactivate the component. For this project, we will not discuss how the control works in detail and that all control signals would be hardcoded for the new instructions.

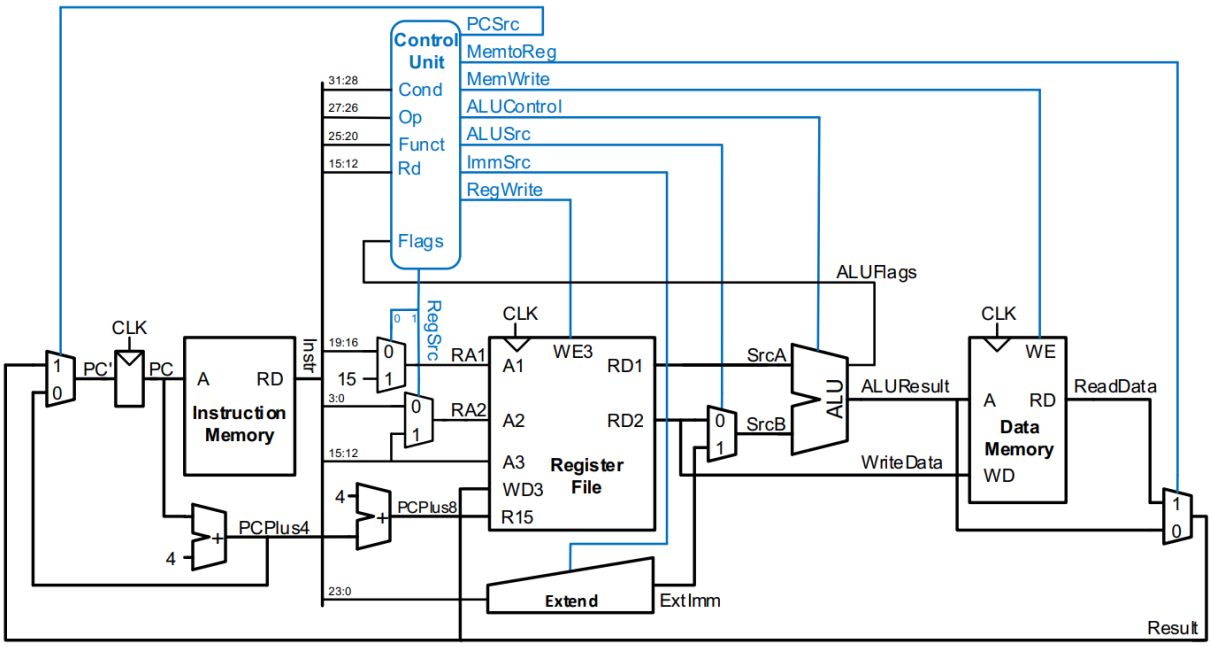

Figure 1. Single Cycle Processor

The datapath and control for this project was built in Vivado and its components were pre-made from previous lab exercises. A schematic of the single cycle processor can be found in Figure 2. Alongside, are the schematic of the datapath and control shown in Figure 3 and 4, respectively.

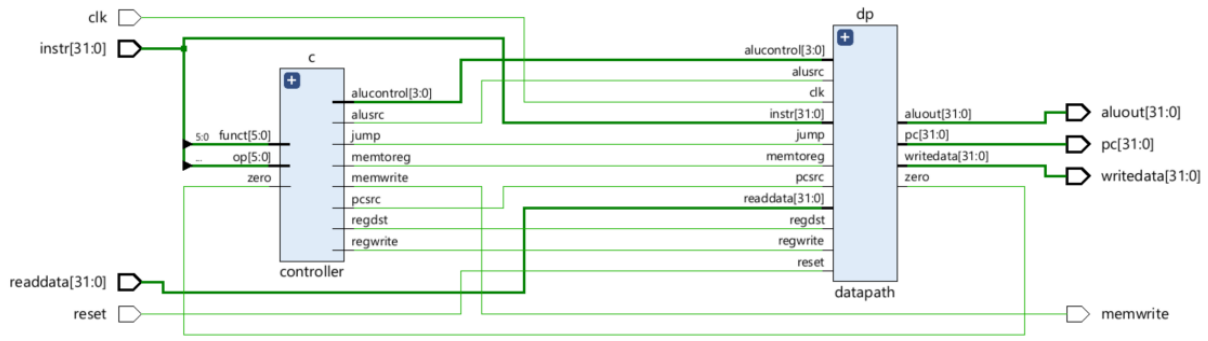

Figure 2. Single Cycle Processor in Vivado

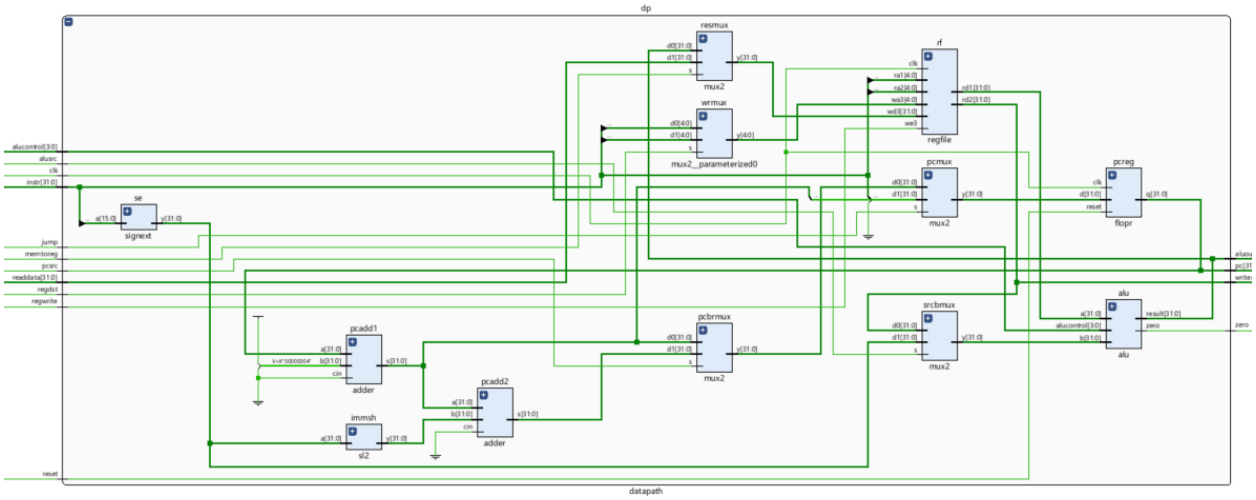

Figure 3. Schematic of Datapath

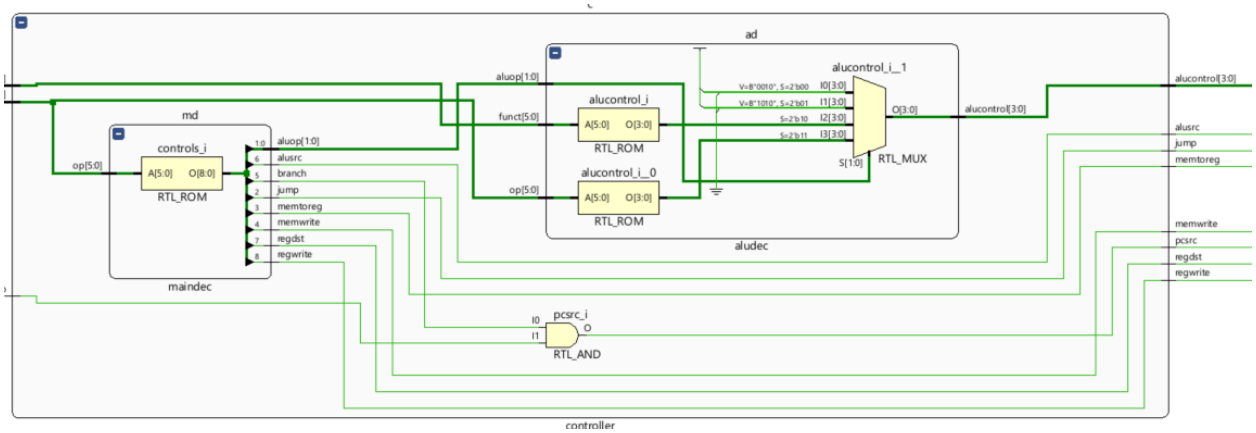

Figure 4. Schematic of the Control

This project required the students to add new instructions to the current implementation of the single-cycle processor. New instructions include Normal instructions which are found in the instruction set like xori, lui, and srlv. Another are the Pseudo-instructions which are instructions not found in the instruction set but are recognized by the assembler. These instructions include

the li, and bgtz. Lastly, custom instructions which are instructions made specifically for this project.

Implementation

As mentioned most of the instructions and files were imported from previous lab exercises. Through this the single cycle microprocessor works as expected given a set of instructions. Currently, the implementation works by reading through a file called memfile.mem which contains the set of instructions written in hex which corresponds to a specific MIPS instruction. These instructions are fed in one at a time to the processor and the processor computes it based on Figure 2. For easy comparison, the original code blocks for the alu, aludec, and maindec will be provided below so that when a new instruction is given it can easily be compared from its previous. Their corresponding schematic will also be provided below.

To reduce repetition on explanation, for all alucontrol input and output parameters the bit length was changed to 5. This allows the programmer to have more space for new instructions. These changes were applied to the maindec, alu, aludec, datapath and control files.

Code Block 1: Maindec original 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
```

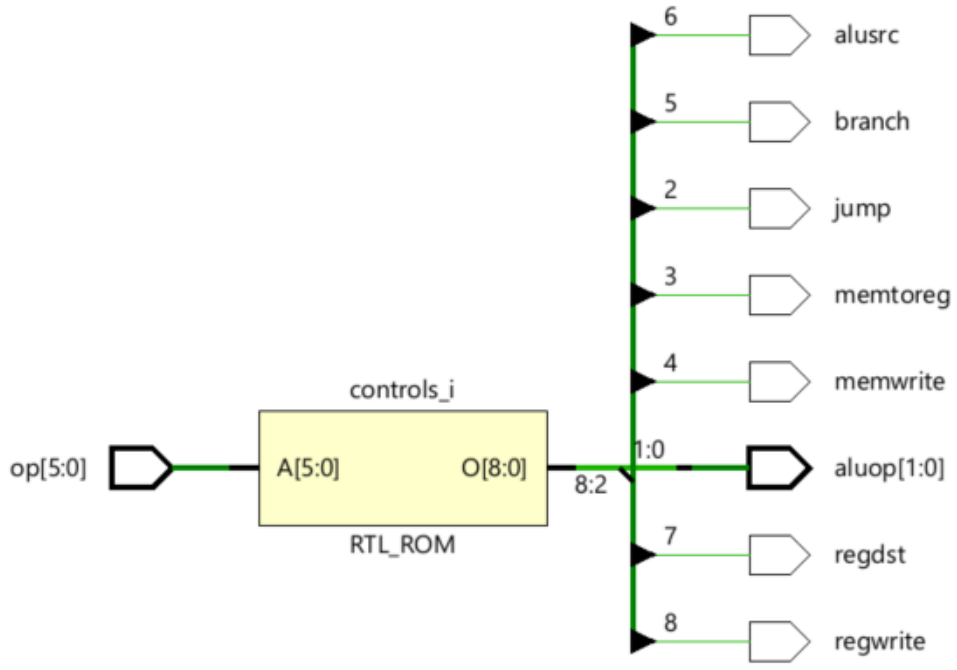

Figure 5. Original maindec schematic

Code Block 2: Aludec original code

```
1  `timescale 1ns / 1ps
2  module aludec(input  logic [5:0] funct,
3                input  logic [1:0] aluop,
4                output logic [2:0] alucontrol);
5
6    always_comb
7      case(aluop)
8        2'b00: alucontrol <= 3'b010;            // add (for lw/sw/addi)
9        2'b01: alucontrol <= 3'b110;            // sub (for beq)
10       default: case(funct)                    // R-type instructions
11           6'b100000: alucontrol <= 3'b010;    // add
12           6'b100010: alucontrol <= 3'b110;    // sub
13           6'b100100: alucontrol <= 3'b000;    // and
14           6'b100101: alucontrol <= 3'b001;    // or
15           6'b101010: alucontrol <= 3'b111;    // slt
16           default:   alucontrol <= 3'bxxx;    // ???
17         endcase
18     endcase
19 endmodule
```

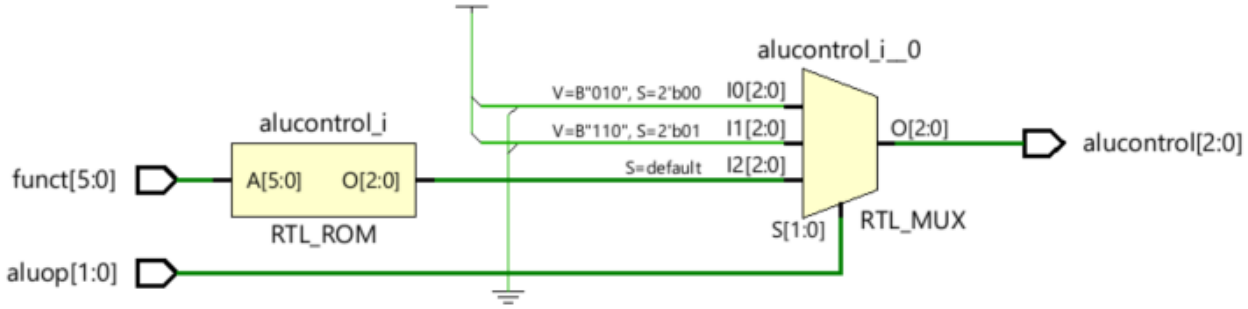

Figure 6. Original aludec schematic

Code block 3: ALU original code

```
1  `timescale 1ns / 1ps
2  module alu(input  logic [31:0] a, b,
3             input  logic [2:0]  alucontrol,
4             output logic [31:0] result,
5             output logic        zero);
6
7    logic [31:0] condinvb, sum;
8
9    assign condinvb = alucontrol[2] ? ~b : b;
10   assign sum = a + condinvb + alucontrol[2];
11
12   always_comb
13     case (alucontrol[1:0])
14       2'b00: result = a & b;
15       2'b01: result = a | b;
16       2'b10: result = sum;
17       2'b11: result = sum[31];
18     endcase
19
20   assign zero = (result == 32'b0);
21 endmodule
```

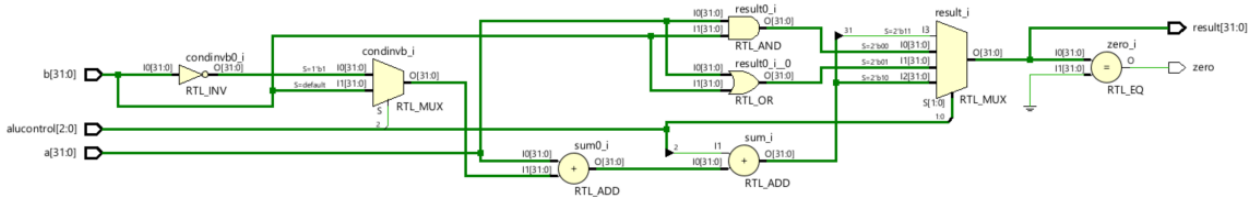

Figure 7. Original ALU schematic

XORI

Xori is a MIPS bitwise instruction wherein it takes in 2 inputs (a register value and immediate value) and it outputs the bitwise operation of the 2 inputs. To recall 0 xor 0 and 1 xor 1 are equivalent to 1. Otherwise, it equals to 0. For this instruction a few lines of code were edited on the ALU, Aludec and maindec files. For the maindec file, the line of code highlighted shown in code block 4 was added. This code provides the hardcoded values for the control signals of the datapath. ***6`b001110*** simply checks if the opcode from the given instructions in the memfile is equivalent to a xori instruction. If it is it must send out the following control signals ***101000011*** which activates the regwrite and alusrc since xori must write to a register and where to get the 2nd input (if from rt register or immediate) in this case it must get from the immediate. The remaining controls are 0 since they are irrelevant for xori. Lastly the aluop is set to 11 so that in the aludec file it would send out the appropriate signal to the ALU.

Code block 4: New 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 [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
      6'b001110: controls <= 9'b101000011; // XORI
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

For the aludec file, the line of code highlighted shown in code block 5 was added. The reason why 11 from the aludec was used was to properly have a portion for I-instructions. Also a wire for the op was added as an input since I-instructions require a unique opcode to determine what type of instructions they are. Moreover, once the opcode has been identified it would send out a unique alucontrol signal to the alu which would do the command.

Code block 5: New aludec

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op,
              input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [4:0] alucontrol);

   always_comb
    case(aluop)
      2'b00: alucontrol <= 5'b00010;              // add (for lw/sw/addi)
      2'b01: alucontrol <= 5'b10010;              // sub (for beq)
      2'b10: case(funct)                         // R-type instructions
          6'b100000: alucontrol <= 5'b00010;     // add
          6'b100010: alucontrol <= 5'b10010;     // sub
          6'b100100: alucontrol <= 5'b00000;     // and
          6'b100101: alucontrol <= 5'b00001;     // or
          6'b101010: alucontrol <= 5'b10011;     // slt
          default:   alucontrol <= 5'bxxxxx;    // ???
        endcase
      2'b11: case(op)                          // I-type instructions
          6'b001110: alucontrol <= 5'b00100; // xori
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

For the alu file the highlighted portion of the code was added from the original. Since xori instruction is zero extended, I hardcoded the immediate input to have 0 bits from 16-31 then added the original lower bits. Since vivado has a bitwise operator for xor, this was used to compute the result (a^xori).

Code block 6: alu modified code

```
`timescale 1ns / 1ps
```

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

  logic [31:0] condinvb, sum, xori;

  assign xori = {16'b0, b[15:0]};
  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  always_comb
    case (alucontrol[3:0])
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = a^xori;      // XORI
    endcase

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

LUI

Lui or load upper immediate is a MIPS instruction wherein it takes an immediate value and a register destination and it loads the immediate to the upper half bits of the register. In short it stores the immediate to bits 16-31. For this implementation lui takes in 2 inputs the immediate value and register destination (rt). The rs would be a don't care so whatever the value used should not affect the output of the lui. For this instruction a few lines of code were edited on the ALU, Aludec and maindec files. For the maindec file, the line of code highlighted shown in code block 7 was added. This code provides the hardcoded values for the control signals of the datapath. ***6`b001111*** simply checks if the opcode from the given instructions in the memfile is equivalent to a lui instruction. If it is it must send out the following control signals ***101000011*** which activates the regwrite and alusrc since lui must write to a register and where to get the 2nd input (if from rt register or immediate) in this case it must get from the immediate. The remaining controls are 0 since they are irrelevant for lui. Lastly the aluop is set to 11 so that in the aludec file it would send out the appropriate signal to the ALU. For the aludec file, the line of code highlighted shown in code block 8 was added. Once the opcode has been identified it would send out a unique alucontrol signal to the alu which would do the command.

Code block 7: New 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 [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
      6'b001110: controls <= 9'b101000011; // XORI
      6'b001111: controls <= 9'b101000011; // LUI
      6'b000110: controls <= 9'b110000010; // SRLV
      6'b011101: controls <= 9'b000100011; // bgtz
      6'b010001: controls <= 9'b101000011; // LI
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code block 8: New aludec

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op,
              input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [4:0] alucontrol);

   always_comb
    case(aluop)
```

```
      2'b00: alucontrol <= 5'b00010;             // add (for lw/sw/addi)
      2'b01: alucontrol <= 5'b10010;             // sub (for beq)
      2'b10: case(funct)                        // R-type instructions
          6'b100000: alucontrol <= 5'b00010;     // add
          6'b100010: alucontrol <= 5'b10010;     // sub
          6'b100100: alucontrol <= 5'b00000;     // and
          6'b100101: alucontrol <= 5'b00001;     // or
          6'b101010: alucontrol <= 5'b10011;     // slt
          6'b000110: alucontrol <= 5'b00110;     // SRLV
          default:   alucontrol <= 5'bxxxxx;    // ???
        endcase
      2'b11: case(op)                         // I-type instructions
          6'b001110: alucontrol <= 5'b00100; // xori
          6'b001111: alucontrol <= 5'b00101; // lui
          6'b011101: alucontrol <= 5'b01000; // bgtz
          6'b010001: alucontrol <= 5'b00111; // li
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

For the alu file the highlighted portion of the code was added from the modified code. Since lui must be added to the upper half bits, what was done was I first placed the result of the immediate or b input then added a hardcoded string of 16 bits of 0s. The result is equivalent to having a lui instruction.

Code block 9: alu modified code

```
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, xori, srlv;


assign xori = {16'b0, b[15:0]};
assign srlv = {16'b0, a[4:0]};
assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];
```

```
  always_comb
    case (alucontrol[3:0])
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = a^xori;     // XORI
      4'b0101: result = {b, 16'b0};      // LUI
      4'b0110: result = b >> srlv;          // SRLV
      4'b0111: result = xori;         // LI
      4'b1000: result = (a[31] === 0 & a > 0) ? 0 : 1; // BGTZ
    endcase

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

SRLV

SRLV or shift right logical variable is a MIPS instruction that takes in 2 inputs from rs and rt. As specified from a source, it shifts the values from rt to the right n number of times as specified in the rs. Which is why it has the following illustration ***rd = rt >> rs***. For this implementation srlv takes in 2 inputs the register sources (rs and rt) and an output destination or rd. For this instruction a few lines of code were edited on the ALU, Aludec and maindec files. For the maindec file, the line of code highlighted shown in code block 10 was added. This code provides the hardcoded values for the control signals of the datapath. ***6`b000110*** simply checks if the opcode from the given instructions in the memfile is equivalent to a srlvi instruction. If it is it must send out the following control signals ***110000010*** which activates the regwrite and regdst signals since srlv must write to a register and where to write it depends on the input signal to be sent to the mux (if from rt register or rd register) in this case it must store to the rd since this is an r-type instruction. Meanwhile the alusrc must be set to 0 since we want our 2nd input to be from rt. The remaining controls are 0 since they are irrelevant for srlv. Lastly the aluop is set to 10 so that in the aludec file it would send out the appropriate signal to the ALU. For the aludec file, the line of code highlighted shown in code block 11 was added. Since the input is 10 it would select from the r-type instructions. It will select the case where the funct code is equivalent to srlv. Once the funct has been identified it would send out a unique alucontrol signal to the alu which would do the command.

Code block 10: New 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 [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
      6'b001110: controls <= 9'b101000011; // XORI
      6'b001111: controls <= 9'b101000011; // LUI
      6'b000110: controls <= 9'b110000010; // SRLV
      6'b011101: controls <= 9'b000100011; // bgtz
      6'b010001: controls <= 9'b101000011; // LI
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code block 11: New aludec

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op,
              input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [4:0] alucontrol);

   always_comb
    case(aluop)
      2'b00: alucontrol <= 5'b00010;             // add (for lw/sw/addi)
      2'b01: alucontrol <= 5'b10010;             // sub (for beq)
      2'b10: case(funct)                        // R-type instructions
```

```
            6'b100000: alucontrol <= 5'b00010;     // add
            6'b100010: alucontrol <= 5'b10010;     // sub
            6'b100100: alucontrol <= 5'b00000;     // and
            6'b100101: alucontrol <= 5'b00001;     // or
            6'b101010: alucontrol <= 5'b10011;     // slt
            6'b000110: alucontrol <= 5'b00110;     // SRLV
            default:   alucontrol <= 5'bxxxxx;    // ???
          endcase
        2'b11: case(op)                        // I-type instructions
            6'b001110: alucontrol <= 5'b00100; // xori
            6'b001111: alucontrol <= 5'b00101; // lui
            6'b011101: alucontrol <= 5'b01000; // bgtz
            6'b010001: alucontrol <= 5'b00111; // li
            default:   alucontrol <= 5'bxxxxx; // ???
          endcase
      endcase
endmodule
```

For the alu file the highlighted portion of the code was added from the modified code. Since srlv takes in the register rs and rt as inputs they are wired as inputs a and b respectively. Srlv works as follows rt is shifted right an amount of times as specified in rs (rd = rt >> rs) hence the implementation in the alu. However, srlv can only shift a maximum of 32 bits hence, it is important to only get the 5 lowest bits from the rs before shifting right hence the addition of the assign srlv in the code. This instruction is fairly straightforward on the implementation.

Code block 12: alu modified code

```
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, xori, srlv;

assign xori = {16'b0, b[15:0]};
assign srlv = {16'b0, a[4:0]};
  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  always_comb
```

```
    case (alucontrol[3:0])
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = a^xori;     // XORI
      4'b0101: result = {b, 16'b0};      // LUI
      4'b0110: result = b >> srlv;          // SRLV
      4'b0111: result = xori;         // LI
      4'b1000: result = (a[31] === 0 & a > 0) ? 0 : 1; // BGTZ
    endcase

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

LI

Li or load immediate is a MIPS pseudo-instruction which means it is a combination of at least 2 instructions to implement. Li based on the mips green sheet is a lui + ori instruction which means it uses lui to load the upper half bits and ori to load the lower half bits. For this project, however, we were given the assumption that the immediate for the li is a maximum of 16 bits and we can assume that there will not be any upper bits. In short this is equivalent to only implementing the ori instruction. Hence, the implementation takes in 2 inputs from rs and immediate and it stores the output to the rt register. For this instruction a few lines of code were edited on the ALU, Aludec and maindec files. For the maindec file, the line of code highlighted shown in code block 13 was added. This code provides the hardcoded values for the control signals of the datapath. ***6`b010001*** simply checks if the opcode from the given instructions in the memfile is equivalent to a li instruction. If it is, it must send out the following control signals ***101000011*** which activates the regwrite and alusrc signals since li must write to a register and select whether to take the rt or immediate as 2nd input, in this case take the immediate. Meanwhile the regdst must be set to 0 since we want to store to rt. The remaining controls are 0 since they are irrelevant for li. Lastly the aluop is set to 11 so that in the aludec file it would send out the appropriate signal to the ALU. For the aludec file, the line of code highlighted shown in code block 14 was added. Since the input is 11 it would select from the i-type instructions. It will select the case where the opcode is equivalent to li. Once the opcode has been identified it would send out a unique alucontrol signal to the alu which would do the command.

Code block 13: New 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 [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
      6'b001110: controls <= 9'b101000011; // XORI
      6'b001111: controls <= 9'b101000011; // LUI
      6'b000110: controls <= 9'b110000010; // SRLV
      6'b011101: controls <= 9'b000100011; // bgtz
      6'b010001: controls <= 9'b101000011; // LI
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code block 14: New aludec

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op,
              input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [4:0] alucontrol);

   always_comb
    case(aluop)
      2'b00: alucontrol <= 5'b00010;             // add (for lw/sw/addi)
      2'b01: alucontrol <= 5'b10010;             // sub (for beq)
      2'b10: case(funct)                        // R-type instructions
          6'b100000: alucontrol <= 5'b00010;     // add
```

```
          6'b100010: alucontrol <= 5'b10010;     // sub
          6'b100100: alucontrol <= 5'b00000;     // and
          6'b100101: alucontrol <= 5'b00001;     // or
          6'b101010: alucontrol <= 5'b10011;     // slt
          6'b000110: alucontrol <= 5'b00110;     // SRLV
          default:   alucontrol <= 5'bxxxxx;    // ???
        endcase
      2'b11: case(op)                        // I-type instructions
          6'b001110: alucontrol <= 5'b00100; // xori
          6'b001111: alucontrol <= 5'b00101; // lui
          6'b011101: alucontrol <= 5'b01000; // bgtz
          6'b010001: alucontrol <= 5'b00111; // li
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

For the alu file the highlighted portion of the code was added from the modified code. Since li must store the lower half bits to the register, what was done was set the result equal to the immediate value (b) input. However, since li is equivalent to implementing ori for this project, what I did was get first the first 16 bits and zero extend it in Vivado. This method was already done in xori hence, I just set my result to xori. Do not get confused since xori is just a variable name.

Code block 15: alu modified code

```
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, xori, srlv;
  assign xori = {16'b0, b[15:0]};
assign srlv = {16'b0, a[4:0]};
  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  always_comb
    case (alucontrol[3:0])
      4'b0000: result = a & b;
```

```
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = a^xori;     // XORI
      4'b0101: result = {b, 16'b0};      // LUI
      4'b0110: result = b >> srlv;          // SRLV
      4'b0111: result = xori;          // LI
      4'b1000: result = (a[31] === 0 & a > 0) ? 0 : 1; // BGTZ
    endcase

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

BGTZ

Bgtz or branch greater than zero is a MIPS pseudo-instruction which means it is a combination of at least 2 instructions to implement. On a personal perspective, bgtz is a combination of bne where bne has $zero as a register input and bgt where an input is $zero also. Bgtz as the name suggests it branches if the register value is greater than 0. For this project, we were given the following assumptions, the opcode is 0x1D and the rt field will contain 0s. For this instruction a few lines of code were edited on the ALU, Aludec and maindec files. For the maindec file, the line of code highlighted shown in code block 16 was added. This code provides the hardcoded values for the control signals of the datapath. ***6`b011101*** simply checks if the opcode from the given instructions in the memfile is equivalent to a bgtz instruction. If it is, it must send out the following control signals ***000100011*** which activates the branch signal since bgtz is a branch instruction and is similar to the implementation of beq. The remaining controls are 0 since they are irrelevant for bgtz. Lastly the aluop is set to 11 so that in the aludec file it would send out the appropriate signal to the ALU. For the aludec file, the line of code highlighted shown in code block 17 was added. Since the input is 11 it would select from the i-type instructions. It will select the case where the opcode is equivalent to bgtz. Once the opcode has been identified it would send out a unique alucontrol signal to the alu which would do the command.

Code block 16: New 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 [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
      6'b001110: controls <= 9'b101000011; // XORI
      6'b001111: controls <= 9'b101000011; // LUI
      6'b000110: controls <= 9'b110000010; // SRLV
      6'b011101: controls <= 9'b000100011; // bgtz
      6'b010001: controls <= 9'b101000011; // LI
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code block 17: New aludec

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op,
              input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [4:0] alucontrol);

   always_comb
    case(aluop)
      2'b00: alucontrol <= 5'b00010;             // add (for lw/sw/addi)
      2'b01: alucontrol <= 5'b10010;             // sub (for beq)
      2'b10: case(funct)                        // R-type instructions
          6'b100000: alucontrol <= 5'b00010;     // add
          6'b100010: alucontrol <= 5'b10010;     // sub
          6'b100100: alucontrol <= 5'b00000;     // and
          6'b100101: alucontrol <= 5'b00001;     // or
          6'b101010: alucontrol <= 5'b10011;     // slt
```

```
          6'b000110: alucontrol <= 5'b00110;     // SRLV
          default:   alucontrol <= 5'bxxxxx;    // ???
        endcase
      2'b11: case(op)                        // I-type instructions
          6'b001110: alucontrol <= 5'b00100; // xori
          6'b001111: alucontrol <= 5'b00101; // lui
          6'b011101: alucontrol <= 5'b01000; // bgtz
          6'b010001: alucontrol <= 5'b00111; // li
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

For the alu file the highlighted portion of the code was added from the modified code. Since bgtz is a branch instruction the zero wire is important hence the value being sent out by bgtz (0 or 1) is relevant for branching. Hence, to calculate if it must branch or not a ternary operation was made it checks if the MSB is 0 to ensure that the register value is positive and 2nd to check if it is greater than 0. If this is satisfied it must return 0, else 1. This is setup like this for the assign zero wire where it is bitwise and to 1 to see if it should branch. If result = 0 then it should branch because zero wire would be 1 else, it would be 0 which means do not branch.

Code block 18: alu modified code

```
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b,
           input  logic [4:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum, xori, srlv;
  assign xori = {16'b0, b[15:0]};
assign srlv = {16'b0, a[4:0]};
  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  always_comb
    case (alucontrol[3:0])
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
```

```
      4'b0100: result = a^xori;     // XORI
      4'b0101: result = {b, 16'b0};      // LUI
      4'b0110: result = b >> srlv;          // SRLV
      4'b0111: result = xori;         // LI
      4'b1000: result = (a[31] === 0 & a > 0) ? 0 : 1; // BGTZ
    endcase

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

Testing

For this portion of the documentation, I will be providing test cases for each new instruction. Code blocks for the MIPS equivalent, hexadecimal machine code and output in Vivado would be provided. For easier representation on the output of the code in Vivado, I will add a store word (sw) instruction after each test case to be shown in the write address line in the waveforms.

Code block 19: XORI MIPS code test cases

```
# Testcase 1:
addi $s0, $zero, 0x0005
xori  $s0, $s0, 0x5
sw   $s0, 0($zero) # Expected output: 0x0

# Testcase 2:
addi $s0, $zero, 0x0003
xori  $s0, $s0, 0x4
sw   $s0, 0($zero) # Expected output: 0x0007

# Testcase 3:
addi $s0, $zero, 0x8000
xori  $s0, $s0, 0x20
sw   $s0, 0($zero) # Expected output: 0xFFFF8020

# Testcase 4:
addi $s0, $zero, 0xFFFF
xori  $s0, $s0, 0xFFFF
sw   $s0, 0($zero) # Expected output: 0xFFFF0000

# Testcase 5:
addi $s0, $zero, 0x0000
xori  $s0, $s0, 0xFFFF
```

```
sw   $s0, 0($zero) # Expected output: 0xFFFF
```

Code block 20: XORI assembly code test cases

```
20100005
3A100005
AC100000
20100003
3A100004
AC100000
20108000
3A100020
AC100000
2010FFFF
3A10FFFF
AC100000
20100000
3A10FFFF
AC100000
```

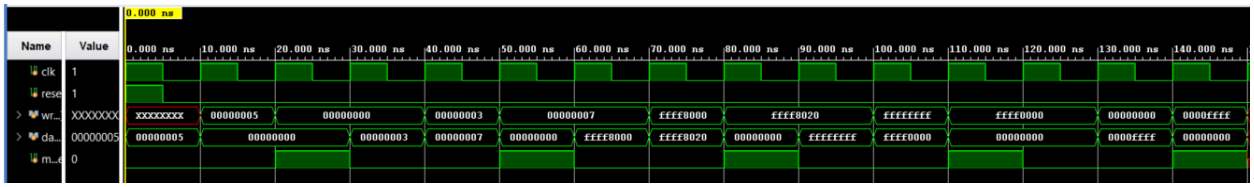

Figure 8: Waveform of XORI implementation

Code block 21: LUI MIPS code test cases

```
# Testcase 1:
lui    $s0, 0x5
sw   $s0, 0($zero) # Expected output: 0x00050000

# Testcase 2:
lui    $s0, 0x0080
sw   $s0, 0($zero) # Expected output: 0x00800000

# Testcase 3:
lui    $s0, 0x8000
```

```
sw   $s0, 0($zero) # Expected output: 0x80000000

# Testcase 4:
lui    $s0, 0xFFFF
sw   $s0, 0($zero) # Expected output: 0xFFFF0000

# Testcase 5:
lui    $s0, 0xC0DE
sw   $s0, 0($zero) # Expected output: 0xC0DE0000
```

Code block 22: LUI assembly code test cases

```
3C100005
AC100000
3C100080
AC100000
3C108000
AC100000
3C10FFFF
AC100000
3C10C0DE
AC100000
```

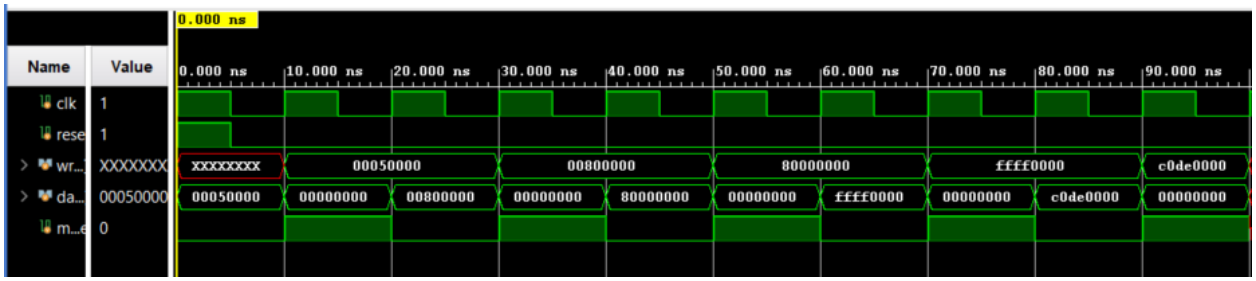

Figure 9: Waveform for lui implementation

Code block 23: SRLV MIPS code test cases

```
# Testcase 1:
addi $s0, $zero, 0x0005
addi $s1, $zero, 0x0002
srlv  $s0, $s0, $s1
sw   $s0, 0($zero) # Expected output: 0x0001
```

```
# Testcase 2:
addi $s0, $zero, 0x0005
addi $s1, $zero, 0x0002
srlv   $s0, $s1, $s0
sw   $s0, 0($zero) # Expected output: 0x0000

# Testcase 3:
addi $s0, $zero, 0x8010
addi $s1, $zero, 0x4
srlv  $s0, $s0, $s1
sw   $s0, 0($zero) # Expected output: 0xffff0801

# Testcase 4:
addi $s0, $zero, 0x0003
addi $s1, $zero, 0xC0DE
srlv   $s0, $s1, $s0
sw   $s0, 0($zero) # Expected output: 0x1FFFF81B

# Testcase 5:
addi $s0, $zero, 0x0000
addi $s1, $zero, 0xC0DE
srlv   $s0, $s1, $s0
sw   $s0, 0($zero) # Expected output: 0xFFFFC0DE
```

Code block 24: SRLV assembly code test cases

```
20100005
20110002
02308006
AC100000
20100005
20110002
02118006
AC100000
20108010
20110004
02308006
AC100000
20100003
2011C0DE
```

```
02118006
AC100000
20100000
2011C0DE
02118006
AC100000
```

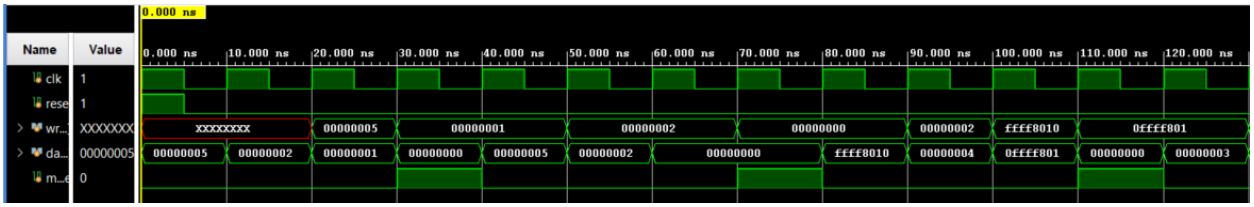

Figure 10: Part 1 of the Waveforms of srlv implementation

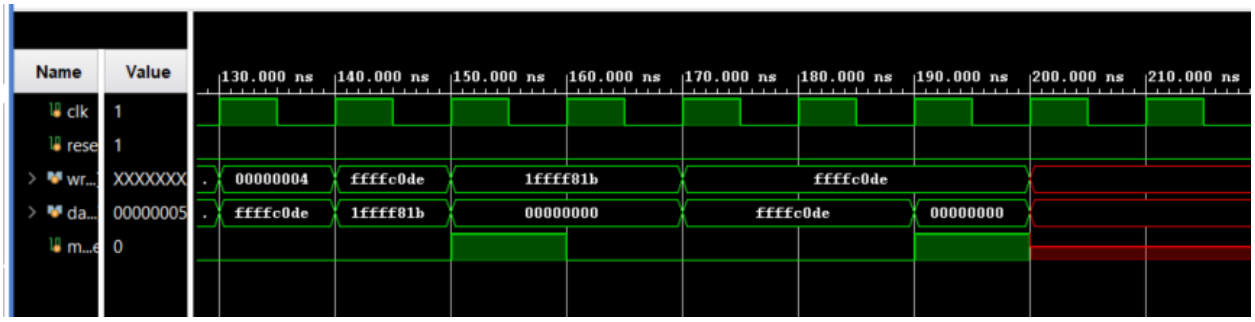

Figure 11: Part 2 of the Waveforms of srlv implementation

Code block 25: LI MIPS code test cases

```
# Testcase 1:
li    $s0, 0x5
sw   $s0, 0($zero) # Expected output: 0x0005

# Testcase 2:
li    $s0, 0x0080
sw   $s0, 0($zero) # Expected output: 0x0080

# Testcase 3:
li    $s0, 0x8000
sw   $s0, 0($zero) # Expected output: 0x8000

# Testcase 4:
li    $s0, 0xFFFF
```

```
sw   $s0, 0($zero) # Expected output: 0xFFFF

# Testcase 5:
li    $s0, 0xC0DE
sw   $s0, 0($zero) # Expected output: 0xC0DE
```

Code block 26: LI assembly code test cases

```
47F00005  # Assume rs is all 1s to show even 1s it won't affect output
AC100000
44100080
AC100000
44108000
AC100000
4410FFFF
AC100000
4410C0DE
AC100000
```

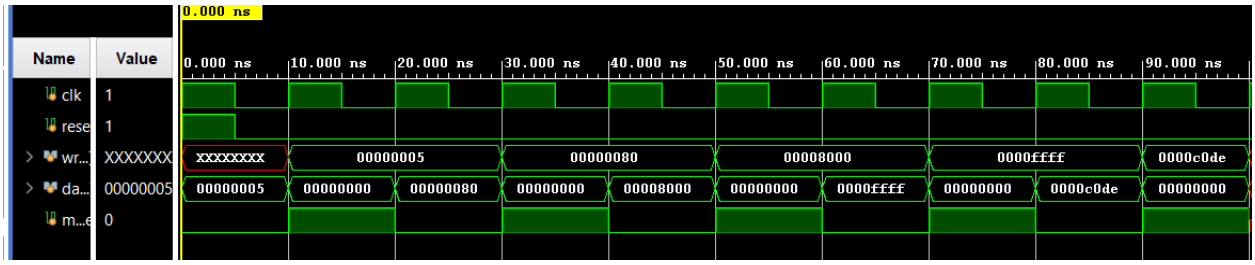

Figure 12: Waveforms for li implementation

Code block 27: BGTZ MIPS code test cases

```
# Testcase 1:
addi $s0, $zero, 0x0005
bgtz $s0, 0x1
addi $s0, $s0, 0x0002
addi $s0, $s0, 0x0001
sw   $s0, 0($zero) # Expected output: 0x0006

# Testcase 2:
addi $s0, $zero, 0x0000
```

```
bgtz $s0, 0x1
addi $s0, $s0, 0x0002
addi $s0, $s0, 0x0001
sw   $s0, 0($zero) # Expected output: 0x0003

# Testcase 3:
addi $s0, $zero, 0x8000
bgtz $s0, 0x1
addi $s0, $s0, 0x0002
addi $s0, $s0, 0x0001
sw   $s0, 0($zero) # Expected output: 0xFFFF8003

# Testcase 4:
addi $s0, $zero, 0xC0DE
bgtz $s0, 0x1
addi $s0, $s0, 0x0002
addi $s0, $s0, 0x0001
sw   $s0, 0($zero) # Expected output: 0xFFFFC0E1

# Testcase 5:
addi $s0, $zero, 0xFFFF
bgtz $s0, 0x1
addi $s0, $s0, 0x0002
addi $s0, $s0, 0x0001
sw   $s0, 0($zero) # Expected output: 0x0002
```

Code block 28: BGTZ assembly code test cases

```
20100005
76000001
22100002
22100001
AC100000
20100000
76000001
22100002
22100001
AC100000
20108000
76000001
22100002
```

```
22100001
AC100000
2010C0DE
76000001
22100002
22100001
AC100000
2010FFFF
76000001
22100002
22100001
AC100000
```

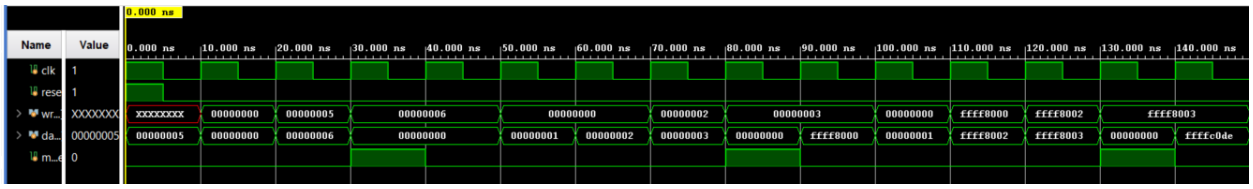

Figure 13: Part 1 of waveforms of bgtz implementation

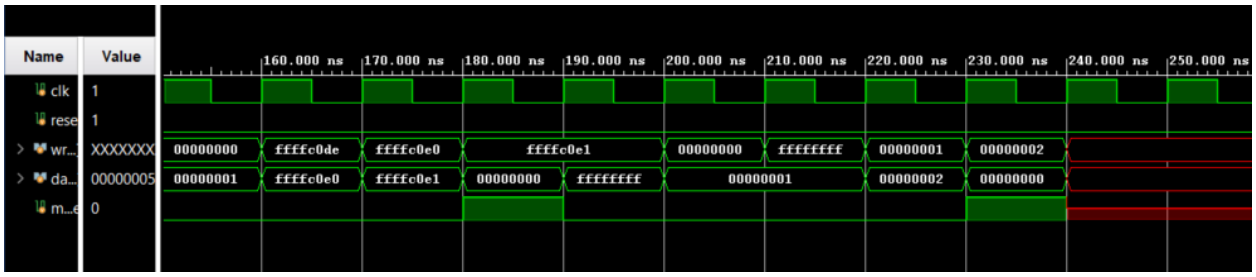

Figure 14: Part 2 of waverforms of bgtz implementation

Code block 29: MIPS code test case for new instructions

```
addi $s0, $zero, 0x5    # s0 = 5
addi $s1, $zero, 0x5    # s1 = 5
bgtz $s0, 0x1           # true
sub  $s2, $s1, $s0      # skip
xori $s3, $s1, 0x1      # s3 = 4
srlv $s5, $s0, $s3      # s5 = 0
lui  $s4, 0x2           # s4 = 0x00020000
add  $s2, $s4, $s5      # s2 = 0x00020000
li   $s6, 0x4           # s6 = 4
add  $s2, $s2, $s6      # s2 = 0x00020004
```

Code block 30: Assembly code test case for new instructions

```
20100005
20110005
76000001
02309022
3A330001
0270A806
3C140002
02959020
44160004
02569020
```

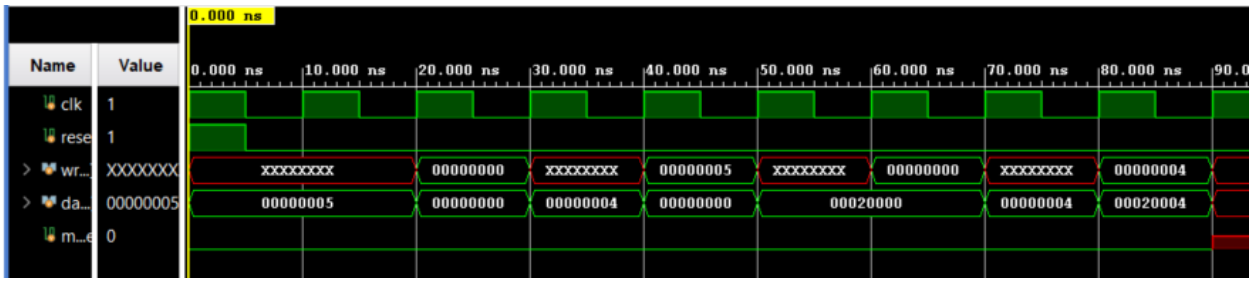

Figure 15: Waveform for Codeblock 30

Code block 31: MIPS code test case for new instructions

```
addi $s0, $zero, 0x0    # s0 = 0
addi $s1, $zero, 0x5    # s1 = 5
bgtz $s0, 0x1           # false
sub  $s2, $s1, $s0      # s2 = 5
xori $s3, $s2, 0x1      # s3 = 4
srlv $s5, $s0, $s3      # s5 = 0
lui  $s4, 0x2           # s4 = 0x00020000
add  $s2, $s4, $s5      # s2 = 0x00020000
li   $s6, 0x4           # s6 = 4
add  $s2, $s2, $s6      # s2 = 0x00020004
```

Code block 32: Assembly code test case for new instructions

```
20100000
20110005
76000001
02309022
3A330001
0270A806
3C140002
02959020
44160004
02569020
```

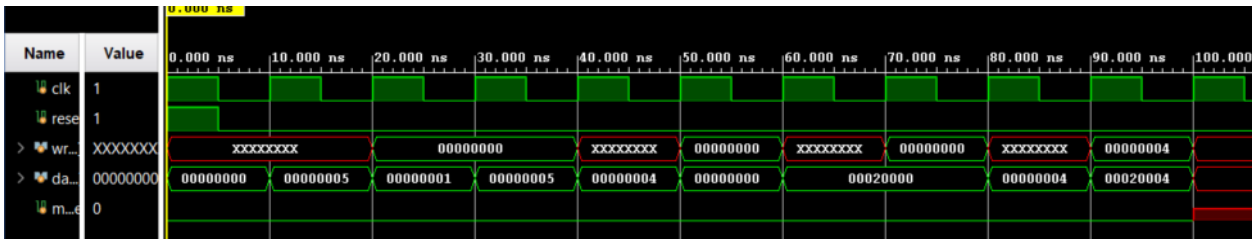

Figure 16: Waveform for Codeblock 32

Supplementary Figures

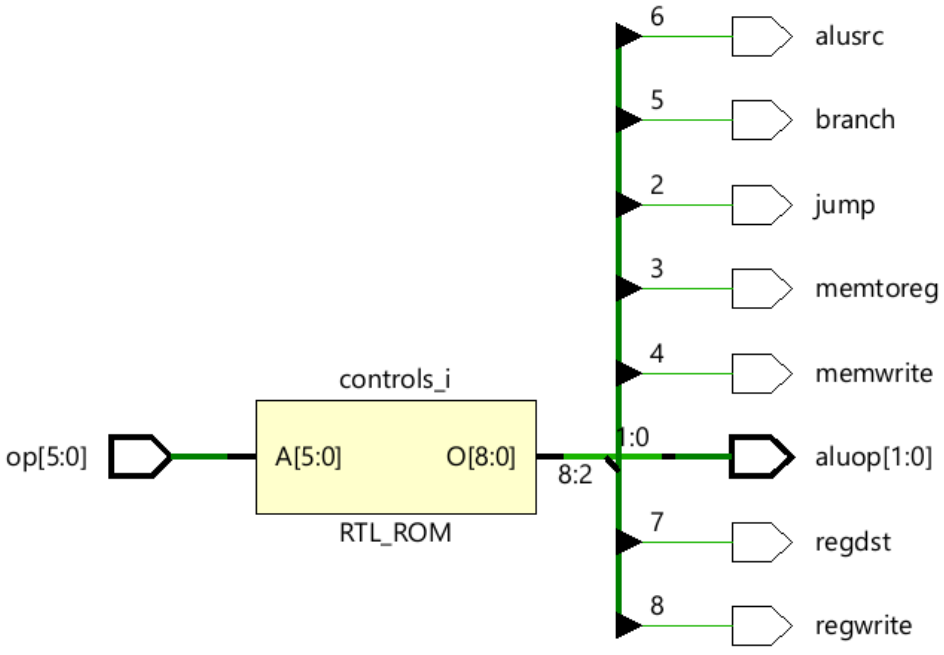

Figure 17: Updated schematic for maindec file

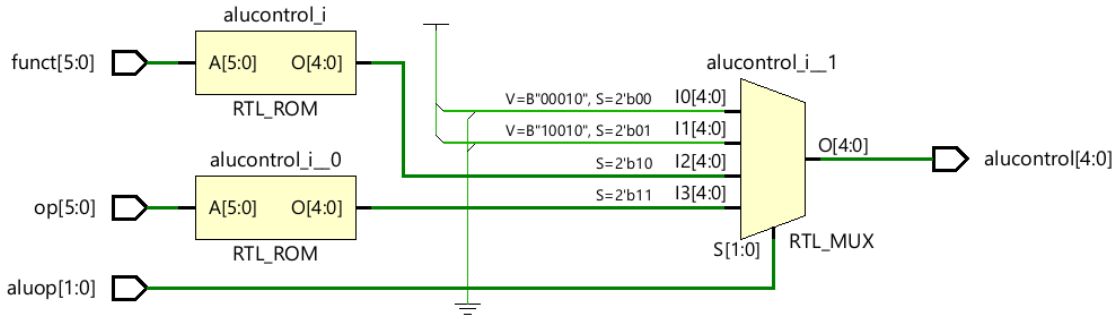

Figure 18: Updated schematic for aludec file

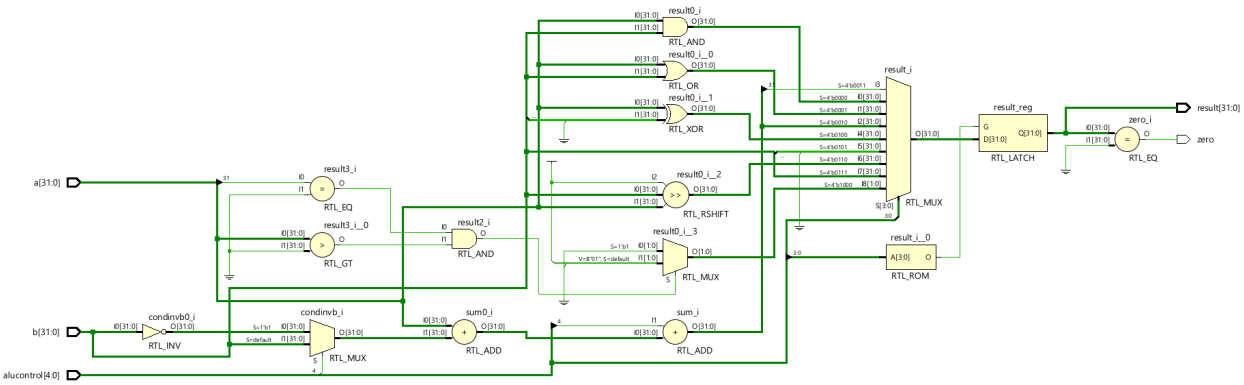

Figure 19: Updated schematic for alu file

References

CS 21 Reference materials

MIPS Instruction Reference. (n.d.). Phoenix.goucher.edu. Retrieved June 25, 2023, from https://phoenix.goucher.edu/~kelliher/f2009/cs220/mipsir.html#:~:text=SRLV%20%2D%2D%20Shift%20right%20logical