CS 21 PROJECT 2 DOCUMENTATION

MIPS Single Cycle Processor Extension

First Semester, A.Y. 2023–2024

Submitted By:

Ann Gabrielle C. Purisima
2021-10796
CS 21 Lab 1

24 January 2024

Link for documentation video:

Project Overview

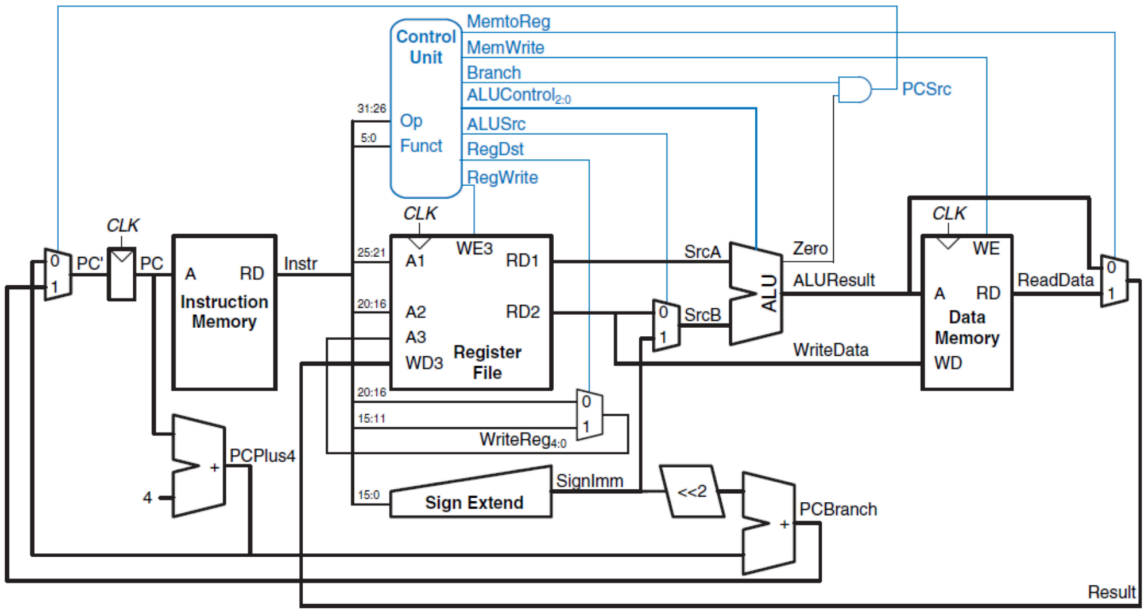

Figure 1. Single-cycle Processor Datapath

The MIPS single-cycle processor is a microprocessor architecture that executes one instruction within a single clock cycle. As seen in Figure 1, the datapath and controller work together to execute the instructions. The datapath is responsible for manipulating the data and its movement during an execution cycle. The components involved in the datapath are the registers, the ALU, and the memory. From the datapath, the controller or the control unit is responsible for managing the control signals required to execute instructions. It receives the instruction opcode and funct and activates the necessary control signals. These signals guide the flow of instructions within the datapath, enabling it to perform the required actions efficiently.

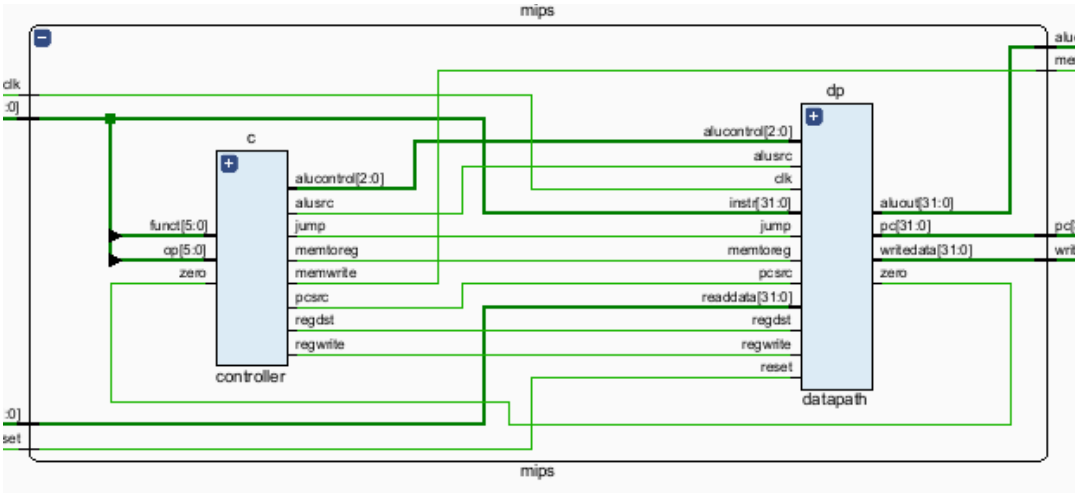

Figure 2. Single-cycle Processor in Vivado

The schematic diagram of the single-cycle processor, shown in Figure 2, displays the datapath and control that will be used in the project. The components of the mentioned datapath and control were given during the previous lab exercises.

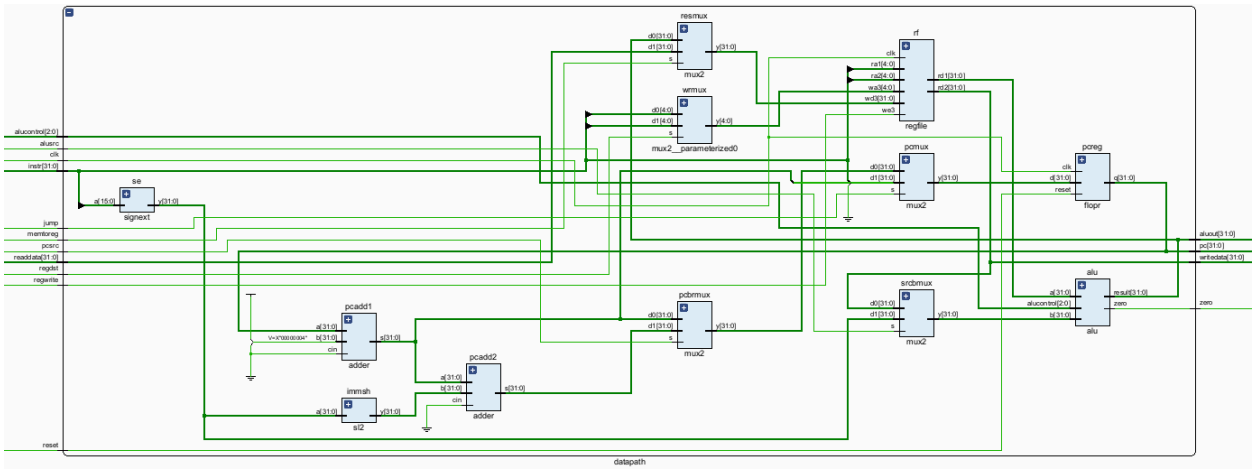

Figure 3. Schematic Diagram of the Datapath

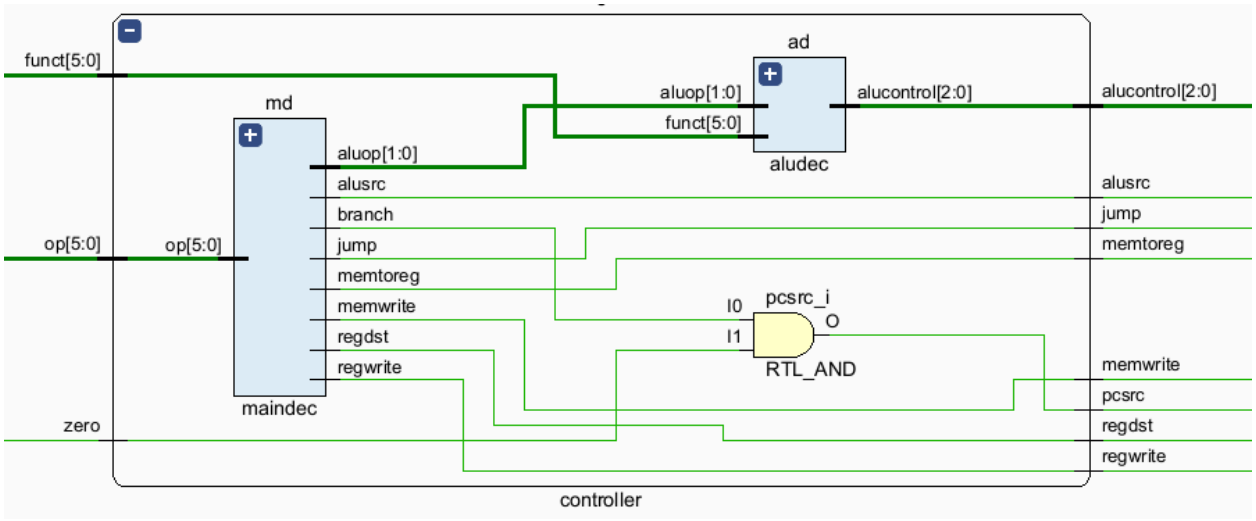

Figure 4. Schematic Diagram of the Controller

To further understand the structure and functionality of the datapath and the controller, their respective schematic diagram are shown in Figures 3 and 4.

Delving into the project, in the previous lab exercise 12, we were given a MIPS single-cycle processor design that is constrained to execute only a subset of the MIPS instruction set. We were tasked to extend the mentioned design to execute additional types of instruction. Firstly, we incorporate the normal instructions movz, lhu, and sllv, which are part of the instruction set but are currently absent in the current design. Secondly, we integrate pseudo-instructions like blt and li, which are not explicitly included in the instruction set but are

recognized by the assembler, being assembled as a sequence of actual instructions; these instructions can be found in the MIPS green sheet. Lastly, we introduced a custom instruction, mix4, which is not present in the MIPS green sheet.

Expanding the design to incorporate the additional instructions will be made possible by modifying the main decoder, ALU decoder, and ALU modules. We focus on altering these modules because they form the backbone of the datapath and control unit. Before we go into the details of each additional instruction, it's important to note that general modifications have been made in the three modules to accommodate these new instructions.

```
`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
```

Code Block 1. Code of Original Main Decoder Module

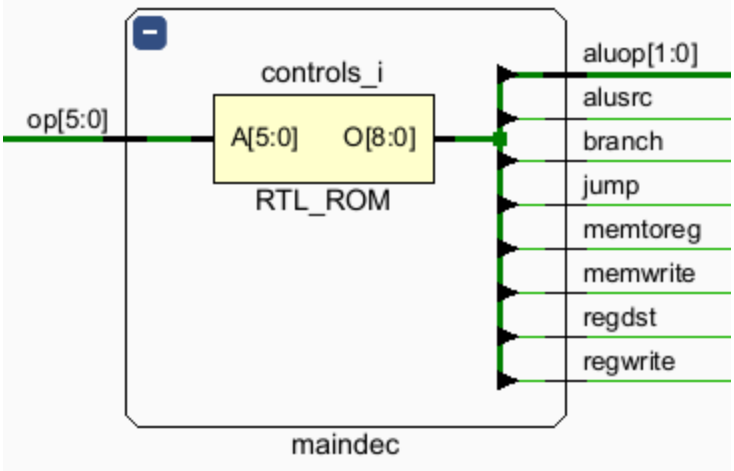

Figure 5. Schematic Diagram of Original Main Decoder Module

The original code for the main decoder module is displayed in Code Block 1, while its associated schematic diagram is illustrated in Figure 5. This module serves as the main decoder for the MIPS single-cycle processor; to be specific, it generates control signals based on the instruction's opcode. The modifications done to this module are additional cases to accommodate the new instructions.

```
`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
```

Code Block 2. Code of Original ALU Decoder Module

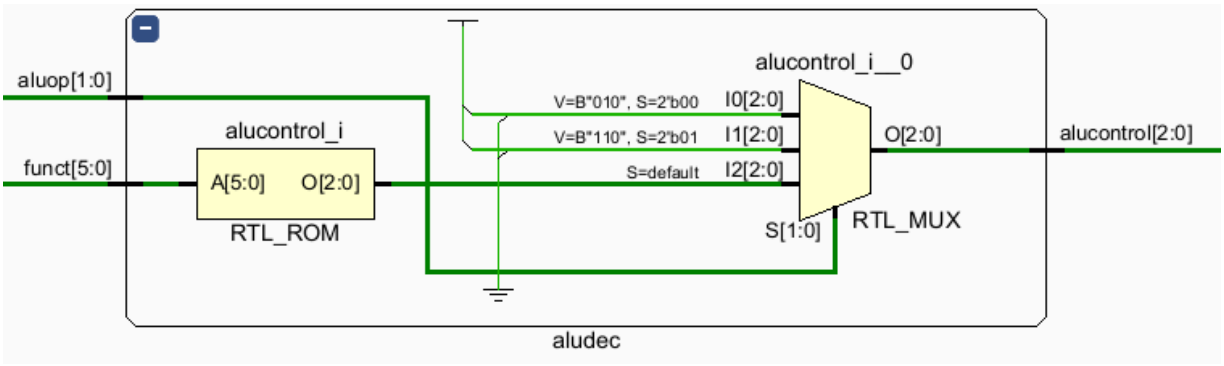

Figure 6. Schematic Diagram of Original ALU Decoder Module

The original code for the ALU decoder module is displayed in Code Block 2, while its associated schematic diagram is illustrated in Figure 6. This module takes in the funct of the instruction and the aluop signal generated from the main decoder module. It will first pass through the aluop cases and if the aluop is not '**2'b00**' nor '**2'b01**' it will pass through the funct cases. This module outputs control signals that will be sent to the ALU module. We present several modifications to integrate new instructions in the design. An additional input parameter 6-bit opcode is introduced for I-type instructions. To have more space for new instructions, the output width of the alucontrol is extended to 5 bits. We set the '**2'b10**' aluop case for the R-type instructions and the '**2'b11**' aluop case for the I-type instructions. Additional cases are also added to '**2'b10**' and '**2'b11**' for the new instructions.

```
`timescale 1ns / 1ps
module alu(input  logic [31:0] a, b,
           input  logic [2:0]  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[1:0])
      2'b00: result = a & b;
      2'b01: result = a | b;
      2'b10: result = sum;
      2'b11: result = sum[31];
    endcase

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

Code Block 3. Code of Original ALU Module

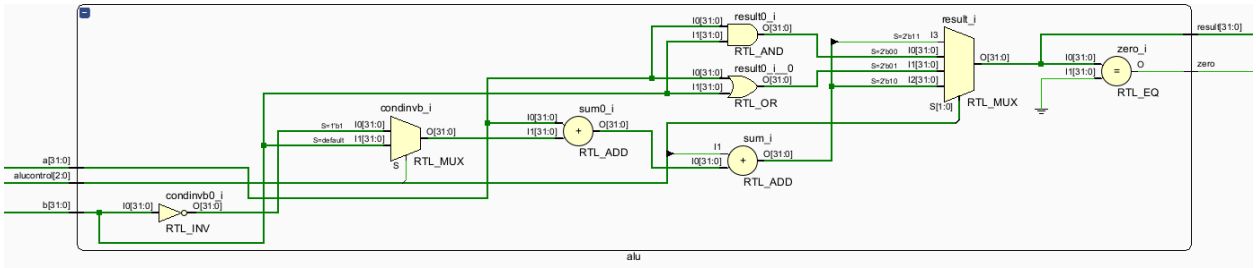

Figure 7. Schematic Diagram of Original ALU Module

The original code for the ALU module is displayed in Code Block 3, while its associated schematic diagram is illustrated in Figure 7. This module is responsible for executing the ALU operation based on the alucontrol signal, taking input operands a and b. We present several modifications to integrate new instructions into this module. Firstly, the input width of alucontrol has been extended to 5 bits, aligning it with the ALU decoder. Additionally, new 32-bit variables, such as uplowbit, assistlhu, lhu, sllv, and slt, have been declared and assigned to accommodate computations for the newly added instructions, as reflected in the assign statements. Furthermore, adjustments to the case statement have been made to incorporate conditions for these new instructions, ensuring the proper assignment of results based on the specified operations.

Changes like the extension of the width of alucontrol to 5 bits are also set on the datapath and controller modules, aligning it with the aludec and alu modules. These changes collectively enable the controller, datapath, maindec, aludec, and alu modules to handle a broader set of instructions, incorporating new functionalities while maintaining the original operations.

Normal Instructions

1. MOVZ

Movz is an R-Type MIPS instruction that stands for "Move Conditional Zero". It takes in 3 operands: 2 source registers '**rs**' and '**rt**' and a destination register '**rd**'. The instruction transfers the value from rs to rd only if the value in rt is zero, otherwise the value in rd would be don't cares or Xs.

```
`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'b001010: controls <= 9'b110000010; // MOVZ
      6'b100101: controls <= 9'b101000011; // LHU
      6'b000100: controls <= 9'b110000010; // SLLV
      6'b011101: controls <= 9'b000100011; // BLT
      6'b010001: controls <= 9'b101000011; // LI
      6'b110011: controls <= 9'b110000010; // MIX4
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code Block 4. Modified Code of Main Decoder Module

The highlighted line of code shown in Code Block 4 reveals the additional case statement made in the maindec module. Though seemingly redundant since the movz instruction is an R-type instruction, it plays a crucial role in ensuring the comprehensive functionality of the maindec module. We use the funct '**6'b001010**' of the movz instruction instead of is opcode in the case statement, and hardcode the necessary control signals '**9'b110000010**' for the movz instruction.

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op, // for i-type instructions
              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'b001010: alucontrol <= 5'b00100; // MOVZ
          6'b000100: alucontrol <= 5'b00110; // SLLV
          6'b110011: alucontrol <= 5'b01001; // MIX4
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
      2'b11: case(op)                  // I-type instructions
          6'b100101: alucontrol <= 5'b00101; // LHU
          6'b011101: alucontrol <= 5'b00111; // BLT
          6'b010001: alucontrol <= 5'b01000; // LI
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

Code Block 5. Modified Code of ALU Decoder Module

The highlighted line of code shown in Code Block 5 reveals the additional case statement made in the aludec module. We already stated why the alucontrol for R-type instructions are under the aluop '**2'b10**', to accommodate the I-type instructions in the future. Since the aluop of movz instruction, from the output of the maindec module, is '**2'b10**', it goes through the case statements using the funct of the R-type instructions. Once the funct matches '**6'b001010**', the alucontrol output is the unique hardcoded value '**5'b00100**'. The control signals in the resulting alucontrol will be used in the ALU module.

```
`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, uplowbit, assistlhu, lhu, sllv, slt;

  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  assign uplowbit = (b == 32'b0) ? a[15:0] : a[31:16];
  assign assistlhu = (b == 32'b0 || b == 'd2) ? uplowbit : 16'b0;
  assign lhu = {16'b0, assistlhu}; // LHU
  assign sllv = {16'b0, a[4:0]}; // SLLV
  assign slt = (a < b) ? 1 : 0; // BLT; slt: 1 if rs < rt, else 0

  always_comb
    case (alucontrol)
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = (b == 32'b0) ? a : 'bx; // MOVZ
      4'b0101: result = lhu; // LHU
      4'b0110: result = b << sllv; // SLLV
      4'b0111: result = (slt != 'b0) ? 0 : 1; // BLT; bne: branch if slt!=0
      4'b1000: result = {16'b0, b[15:0]}; // LI; zero extend immediate
      4'b1001: result = {a[31:24], b[23:16], a[15:8], b[7:0]}; // MIX4
    endcase

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

Code Block 6. Modified Code of ALU Module

The highlighted line of codes shown in Code Block 6 reveals the additional statement made in the ALU module. We used a conditional ternary operator '**(b == 32'b0) ? a : 'bx;'** to implement the movz instruction. The condition '**(b == 32'b0)**' checks if b or rt is equal to zero. If the condition is true, the value of rs will be assigned to the output of ALU module 'result'; otherwise, don't care values or '**'bx**' will be assigned to the output of ALU module 'result'.

2. LHU

LHU is an I-Type MIPS instruction that stands for “Load Halfword Unsigned”. The instruction takes in 3 operands: 1 source register ‘rs’, 1 destination register ‘rt’, and an immediate value ‘imm’. The lhu instruction either loads the lowest or highest 16 bits of the value in the given memory in rs and zero extends this 16-bit value.

```
`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'b001010: controls <= 9'b110000010; // MOVZ
      6'b100101: controls <= 9'b101000011; // LHU
      6'b000100: controls <= 9'b110000010; // SLLV
      6'b011101: controls <= 9'b000100011; // BLT
      6'b010001: controls <= 9'b101000011; // LI
      6'b110011: controls <= 9'b110000010; // MIX4
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code Block 7. Modified Code of Main Decoder Module

The highlighted line of code shown in Code Block 7 reveals the additional case statement made in the maindec module. We use the opcode ‘**6’b100101**’ of the lhu instruction and hardcode the necessary control signals ‘**9'b101000011**’ for the lhu instruction.

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op, // for i-type instructions
              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'b001010: alucontrol <= 5'b00100; // MOVZ
          6'b000100: alucontrol <= 5'b00110; // SLLV
          6'b110011: alucontrol <= 5'b01001; // MIX4
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
      2'b11: case(op)                  // I-type instructions
          6'b100101: alucontrol <= 5'b00101; // LHU
          6'b011101: alucontrol <= 5'b00111; // BLT
          6'b010001: alucontrol <= 5'b01000; // LI
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

Code Block 8. Modified Code of ALU Decoder Module

The highlighted line of code shown in Code Block 8 reveals the additional statements made in the aludec module. Since the aluop of the lhu instruction, from the output of the maindec module, is '**2'b11**', it goes through the case statements using the opcode of the I-type instructions. Once the opcode matches '**6'b100101**', the alucontrol output is the unique hardcoded value '**5'b00101**'. The control signals in the resulting alucontrol will be used in the ALU module.

```
`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, uplowbit, assistlhu, lhu, sllv, slt;

  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  assign uplowbit = (b == 32'b0) ? a[15:0] : a[31:16];
  assign assistlhu = (b == 32'b0 || b == 'd2) ? uplowbit : 16'b0;
  assign lhu = {16'b0, assistlhu}; // LHU
  assign sllv = {16'b0, a[4:0]}; // SLLV
  assign slt = (a < b) ? 1 : 0; // BLT; slt: 1 if rs < rt, else 0

  always_comb
    case (alucontrol)
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = (b == 32'b0) ? a : 'bx; // MOVZ
      4'b0101: result = lhu; // LHU
      4'b0110: result = b << sllv; // SLLV
      4'b0111: result = (slt != 'b0) ? 0 : 1; // BLT; bne: branch if slt!=0
      4'b1000: result = {16'b0, b[15:0]}; // LI; zero extend immediate
      4'b1001: result = {a[31:24], b[23:16], a[15:8], b[7:0]}; // MIX4
    endcase

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

Code Block 9. Modified Code of ALU Module

The highlighted line of code shown in Code Block 9 reveals the additional statements made in the ALU module. We declared 3 new 32-bit variables '**uplowbit**', '**assistlhu**', and '**lhu**'. To operate the lhu instruction, we first used a conditional ternary operator '**(b == 32'b0) ? a[15:0] : a[31:16];**' which checks if the imm is equal to 0. If the condition is true, the lower 16 bits of rs are stored in the variable uplowbit; otherwise, the upper 16 bits of rs are stored in the variable uplowbit. After that, we again used a conditional ternary operator '**(b == 32'b0 || b == 'd2) ? uplowbit : 0;**' which checks if the offset is equal to 0 or 2. If the condition is true, the value in the variable uplowbit is stored in the variable assistlhu; otherwise, the value '**16'b0**' is stored in the variable assistlhu. We then zero extend the value stored in the variable assistlhu and store the resulting 32-bit value to the variable lhu. In the alucontrol case statements, once it matches '**4'b0101**', the result from the variable lhu will be assigned to the output of ALU module 'result'.

3. SLLV

SLLV is an R-Type MIPS instruction that stands for “Shift Left Logical Variable”. It takes in 3 operands: 2 source registers ‘rs’ and ‘rt’ and a destination register ‘rd’. The instruction shifts the value of rt to the left by the lower 5 bits of rs and stores the resulting value in regist rd.

```
`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'b001010: controls <= 9'b110000010; // MOVZ
      6'b100101: controls <= 9'b101000011; // LHU
      6'b000100: controls <= 9'b110000010; // SLLV
      6'b011101: controls <= 9'b000100011; // BLT
      6'b010001: controls <= 9'b101000011; // LI
      6'b110011: controls <= 9'b110000010; // MIX4
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code Block 10. Modified Code of Main Decoder Module

The highlighted line of code shown in Code Block 10 reveals the additional case statement made in the maindec module. Though seemingly redundant since the sllv instruction is an R-type instruction, it plays a crucial role in ensuring the comprehensive functionality of the maindec module. We use the funct ‘**6'b000100**’ of the sllv instruction instead of is opcode in the case statement, and hardcode the necessary control signals ‘**9'b110000010**’ for the sllv instruction.

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op, // for i-type instructions
              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'b001010: alucontrol <= 5'b00100; // MOVZ
          6'b000100: alucontrol <= 5'b00110; // SLLV
          6'b110011: alucontrol <= 5'b01001; // MIX4
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
      2'b11: case(op)               // I-type instructions
          6'b100101: alucontrol <= 5'b00101; // LHU
          6'b011101: alucontrol <= 5'b00111; // BLT
          6'b010001: alucontrol <= 5'b01000; // LI
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

Code Block 11. Modified Code of ALU Decoder Module

The highlighted line of code shown in Code Block 11 reveals the additional case statement made in the aludec module. We already stated why the alucontrol for R-type instructions are under the aluop '**2'b10**'. Since the aluop of the sllv instruction, from the output of the maindec module, is '**2'b10**', it goes through the case statements using the funct of the R-type instructions. Once the funct matches '**6'b000100**', the alucontrol output is the unique hardcoded value '**5'b00110**'. The control signals in the resulting alucontrol will be used in the ALU module.

```
`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, uplowbit, assistlhu, lhu, sllv, slt;

  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  assign uplowbit = (b == 32'b0) ? a[15:0] : a[31:16];
  assign assistlhu = (b == 32'b0 || b == 'd2) ? uplowbit : 16'b0;
  assign lhu = {16'b0, assistlhu}; // LHU
  assign sllv = {16'b0, a[4:0]}; // SLLV
  assign slt = (a < b) ? 1 : 0; // BLT; slt: 1 if rs < rt, else 0

  always_comb
    case (alucontrol)
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = (b == 32'b0) ? a : 'bx; // MOVZ
      4'b0101: result = lhu; // LHU
      4'b0110: result = b << sllv; // SLLV
      4'b0111: result = (slt != 'b0) ? 0 : 1; // BLT; bne: branch if slt!=0
      4'b1000: result = {16'b0, b[15:0]}; // LI; zero extend immediate
      4'b1001: result = {a[31:24], b[23:16], a[15:8], b[7:0]}; // MIX4
    endcase

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

Code Block 12. Modified Code of ALU Module

The highlighted line of code shown in Code Block 12 reveals the additional statements made in the ALU module. We declared a new 32-bit variable '**sllv**' to assist in operating the sllv instruction. Since the instruction can only shift a maximum of 32 bits, we are only allowed to use the lowest 5 bits from rs. We zero extend these 5 bits and store them to the variable sllv. In the alucontrol case statements, once it matches '**4'b0110**', the operation for the sllv instruction will be performed which is '**b << sllv**'. This shifts the value of rt to the left by the value in variable sllv. The resulting value of this operation will be assigned to the output of the ALU module 'result'.

Pseudo–Instructions

1. BLT

BLT is a MIPS pseudo-instruction that stands for “Branch Less Than”. We were instructed that the opcode of the instruction is ‘**’h1D**’ which is ‘**6’b011101**’ in binary. The instruction takes in 3 operands: 2 source registers ‘rs’ and ‘rt’ and an immediate value ‘imm’. Since blt is a pseudo-instruction, the 2 instructions that make up the blt instruction are the slt and the bne instructions. The instruction branches to the immediate value given if the value in rs is less than the value in rt, otherwise branch will not take place.

```
`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'b001010: controls <= 9'b110000010; // MOVZ
      6'b100101: controls <= 9'b101000011; // LHU
      6'b000100: controls <= 9'b110000010; // SLLV
      6'b011101: controls <= 9'b000100011; // BLT
      6'b010001: controls <= 9'b101000011; // LI
      6'b110011: controls <= 9'b110000010; // MIX4
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code Block 13. Modified Code of Main Decoder Module

The highlighted line of code shown in Code Block 13 reveals the additional case statement made in the maindec module. We use the opcode ‘**6’b011101**’ of the blt instruction and hardcode the necessary control signals ‘**9'b000100011**’ for the blt instruction.

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op, // for i-type instructions
              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'b001010: alucontrol <= 5'b00100; // MOVZ
          6'b000100: alucontrol <= 5'b00110; // SLLV
          6'b110011: alucontrol <= 5'b01001; // MIX4
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
      2'b11: case(op)                  // I-type instructions
          6'b100101: alucontrol <= 5'b00101; // LHU
          6'b011101: alucontrol <= 5'b00111; // BLT
          6'b010001: alucontrol <= 5'b01000; // LI
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

Code Block 14. Modified Code of ALU Decoder Module

The highlighted line of code shown in Code Block 14 reveals the additional case statement made in the aludec module. Since the aluop of the blt instruction, from the output of the maindec module, is '**2'b11**', it goes through the case statements using the opcode of the I-type instructions. Once the opcode matches '**6'b011101**', the alucontrol output is the unique hardcoded value '**5'b00111**'. The control signals in the resulting alucontrol will be used in the ALU module.

```
`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, uplowbit, assistlhu, lhu, sllv, slt;

  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  assign uplowbit = (b == 32'b0) ? a[15:0] : a[31:16];
  assign assistlhu = (b == 32'b0 || b == 'd2) ? uplowbit : 16'b0;
  assign lhu = {16'b0, assistlhu}; // LHU
  assign sllv = {16'b0, a[4:0]}; // SLLV
  assign slt = (a < b) ? 1 : 0; // BLT; slt: 1 if rs < rt, else 0

  always_comb
    case (alucontrol)
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = (b == 32'b0) ? a : 'bx; // MOVZ
      4'b0101: result = lhu; // LHU
      4'b0110: result = b << sllv; // SLLV
      4'b0111: result = (slt != 'b0) ? 0 : 1; // BLT; bne: branch if slt!=0
      4'b1000: result = {16'b0, b[15:0]}; // LI; zero extend immediate
      4'b1001: result = {a[31:24], b[23:16], a[15:8], b[7:0]}; // MIX4
    endcase

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

Code Block 15. Modified Code of ALU Module

The highlighted line of code shown in Code Block 15 reveals the additional statements made in the alu module. To assist the blt operation, we declared a new 32-bit variable '**slt**'. We used a conditional ternary operator '**(a < b) ? 1 : 0;'** that checks if the value in a or the register 'rs' is less than the value in b or the register 'rt'. If the condition is true, the value 1 is stored in the variable slt, otherwise, the value 0 is stored in the variable slt. In the alucontrol case statements, once it matches '**4'b0111**', it goes through another conditional ternary operator '**(slt != 'b0) ? 0 : 1;'** that checks if the value in the variable slt is not equal to 0. If the condition is true, the value 0 will be stored to the output of ALU module 'result'; otherwise, 1 will be stored to the output of the ALU module 'result'. If the value of the ALU module 'result' is 0 then it will branch; otherwise, if the value of the ALU module 'result' is 1 then the branch will not happen.

2. LI

LI is an MIPS pseudo-instruction that stands for "Load Immediate". We were instructed that the opcode of the li instruction is '**'h11**' which is '**6'b010001**' in binary, the source register 'rs' which is the bits 21 to 25 have don't care values or Xs, the target register 'rt' is in bits 16 to 20, and the immediate field 'imm' which is the bits 0 to 15 are zero-extended. Though li is a pseudo-instruction that is a combination of the lui and the ori instructions, we can directly hardcode zero extending the value of the immediate field.

```
`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'b001010: controls <= 9'b110000010; // MOVZ
      6'b100101: controls <= 9'b101000011; // LHU
      6'b000100: controls <= 9'b110000010; // SLLV
      6'b011101: controls <= 9'b000100011; // BLT
      6'b010001: controls <= 9'b101000011; // LI
      6'b110011: controls <= 9'b110000010; // MIX4
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code Block 16. Modified Code of Main Decoder Module

The highlighted line of code shown in Code Block 16 reveals the additional case statement made in the maindec module. We use the opcode '**6'b010001**' of the li instruction and hardcode the necessary control signals '**9'b000100011**' for the li instruction.

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op, // for i-type instructions
              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'b001010: alucontrol <= 5'b00100; // MOVZ
          6'b000100: alucontrol <= 5'b00110; // SLLV
          6'b110011: alucontrol <= 5'b01001; // MIX4
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
      2'b11: case(op)                  // I-type instructions
          6'b100101: alucontrol <= 5'b00101; // LHU
          6'b011101: alucontrol <= 5'b00111; // BLT
          6'b010001: alucontrol <= 5'b01000; // LI
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

Code Block 17. Modified Code of ALU Decoder Module

The highlighted line of code shown in Code Block 17 reveals the additional case statement made in the aludec module. Since the aluop of the li instruction, from the output of the maindec module, is '**2'b11**', it goes through the case statements using the opcode of the I-type instructions. Once the opcode matches '**6'b010001**', the alucontrol output is the unique hardcoded value '**5'b01000**'. The control signals in the resulting alucontrol will be used in the ALU module.

```
`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, uplowbit, assistlhu, lhu, sllv, slt;

  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  assign uplowbit = (b == 32'b0) ? a[15:0] : a[31:16];
  assign assistlhu = (b == 32'b0 || b == 'd2) ? uplowbit : 16'b0;
  assign lhu = {16'b0, assistlhu}; // LHU
  assign sllv = {16'b0, a[4:0]}; // SLLV
  assign slt = (a < b) ? 1 : 0; // BLT; slt: 1 if rs < rt, else 0

  always_comb
    case (alucontrol)
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = (b == 32'b0) ? a : 'bx; // MOVZ
      4'b0101: result = lhu; // LHU
      4'b0110: result = b << sllv; // SLLV
      4'b0111: result = (slt != 'b0) ? 0 : 1; // BLT; bne: branch if slt!=0
      4'b1000: result = {16'b0, b[15:0]}; // LI; zero extend immediate
      4'b1001: result = {a[31:24], b[23:16], a[15:8], b[7:0]}; // MIX4
    endcase

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

Code Block 18. Modified Code of ALU Module

The highlighted line of code shown in Code Block 18 reveals the additional case statement made in the alu module. To perform the operation of the li instruction, we hardcoded zero extending the lower 16 bits of rt. In the alucontrol case statements, once it matches '**4'b1000**', the resulting value of this operation will be assigned to the output of ALU module 'result'.

Custom Instructions

1. MIX4

MIX4 is a custom instruction that takes in 3 operands: 2 source registers ‘rs’ and ‘rt’ and a destination register ‘rd’. We were instructed that the format of the mix4 instruction is as follows, bits 31-26 is the opcode ‘**’h00**’, bits 25-21 is rs, bits 20-16 is rt, bits 15-11 is rd, bits 10-6 are don’t cares ‘**’bx**’, and bits 5-0 is the funct ‘**’h33**’. A detailed description of the instruction’s operation will be provided in the modification of alu module.

```
`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'b001010: controls <= 9'b110000010; // MOVZ
      6'b100101: controls <= 9'b101000011; // LHU
      6'b000100: controls <= 9'b110000010; // SLLV
      6'b011101: controls <= 9'b000100011; // BLT
      6'b010001: controls <= 9'b101000011; // LI
      6'b110011: controls <= 9'b110000010; // MIX4
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

Code Block 19. Modified Code of Main Decoder Module

The highlighted line of code shown in Code Block 19 reveals the additional case statement made in the maindec module. Though seemingly redundant since the mix4 instruction is an R-type instruction, it plays a crucial role in ensuring the comprehensive functionality of the maindec module. We use the funct ‘**6'b110011**’ of the mix4 instruction instead of is opcode in the case statement, and hardcode the necessary control signals ‘**9'b110000010**’ for the mix4 instruction.

```
`timescale 1ns / 1ps
module aludec(input  logic [5:0] op, // for i-type instructions
              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'b001010: alucontrol <= 5'b00100; // MOVZ
          6'b000100: alucontrol <= 5'b00110; // SLLV
          6'b110011: alucontrol <= 5'b01001; // MIX4
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
      2'b11: case(op)                // I-type instructions
          6'b100101: alucontrol <= 5'b00101; // LHU
          6'b011101: alucontrol <= 5'b00111; // BLT
          6'b010001: alucontrol <= 5'b01000; // LI
          default:   alucontrol <= 5'bxxxxx; // ???
        endcase
    endcase
endmodule
```

Code Block 20. Modified Code of ALU Decoder Module

The highlighted line of code shown in Code Block 20 reveals the additional case statement made in the aludec module. Since aluop of mix4 instruction, from the output of maindec module, is '**2'b10**', it goes through the case statements using the funct of the R-type instructions. Once the funct matches '**6'b110011**', the alucontrol output is the unique hardcoded value '**5'b01001**'. The control signals in the resulting alucontrol will be used in the ALU module.

```
`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, uplowbit, assistlhu, lhu, sllv, slt;

  assign condinvb = alucontrol[4] ? ~b : b;
  assign sum = a + condinvb + alucontrol[4];

  assign uplowbit = (b == 32'b0) ? a[15:0] : a[31:16];
  assign assistlhu = (b == 32'b0 || b == 'd2) ? uplowbit : 16'b0;
  assign lhu = {16'b0, assistlhu}; // LHU
  assign sllv = {16'b0, a[4:0]}; // SLLV
  assign slt = (a < b) ? 1 : 0; // BLT; slt: 1 if rs < rt, else 0

  always_comb
    case (alucontrol)
      4'b0000: result = a & b;
      4'b0001: result = a | b;
      4'b0010: result = sum;
      4'b0011: result = sum[31];
      4'b0100: result = (b == 32'b0) ? a : 'bx; // MOVZ
      4'b0101: result = lhu; // LHU
      4'b0110: result = b << sllv; // SLLV
      4'b0111: result = (slt != 'b0) ? 0 : 1; // BLT; bne: branch if slt!=0
      4'b1000: result = {16'b0, b[15:0]}; // LI; zero extend immediate
      4'b1001: result = {a[31:24], b[23:16], a[15:8], b[7:0]}; // MIX4
    endcase

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

Code Block 21. Modified Code of ALU Module

The highlighted line of code shown in Code Block 21 reveals the additional statement made in the alu module. We were instructed that the operation of the mix4 instruction is as follows, bits 31-24 of rd is the bits 31-24 of rs, bits 23-16 of rd is the bits 23-16 of rt, bits 15-8 of rd is the bits 15-8 of rs, and bits 7-0 of rd is the bits 7-0 of rt. To operate the mix4 instruction, we hardcoded the values as directed. In the alucontrol case statements, once it matches '**4'b1001**', the resulting value of this operation will be assigned to the output of ALU module 'result'.

Testcases

This section of the documentation aims to provide comprehensive test cases for the newly added instructions to the MIPS single-cycle processor design. Each testcase shows the assembly code and its corresponding machine code. By exploring these test cases, we aim to demonstrate that the processor can now successfully execute the new instructions.

1. MOVZ Testcases

```
# FOR TEST CASES 1 TO 5
li $t0, 0 (0x0 in hex)
li $t1, 3 (0x3 in hex)

# Testcase 1
li $t2, 11 (0xB in hex)
movz $s0, $t2, $t0
sw $s0, 0($zero) # Expected Output: 0xB

# Testcase 2
li $t3, 126 (0x7E in hex)
movz $s1, $t3, $t1
sw $s1, 0($zero) # Expected Output: XXXXXXXX

# Testcase 3
li $t4, 1191 (0x4A7 in hex)
movz $s2, $t4, $t0
sw $s2, 0($zero) # Expected Output: 0x4A7

# Testcase 4
li $t5, 3375 (0xD2F in hex)
movz $s3, $t5, $t1
sw $s3, 0($zero) # Expected Output: XXXXXXXX

# Testcase 5
li $t6, 37318 (0x91C6 in hex)
movz $s4, $t6, $t0
sw $s4, 0($zero) # Expected Output: 0x91C6
```

Code Block 22. MOVZ MIPS Assembly Code Test Cases

```
44080000
44090003

440A000B
148800A
AC100000

440B007E
169880A
AC110000

440C04A7
188900A
AC120000

440D0D2F
1A9980A
AC130000

440E91C6
1C8A00A
AC140000
```

Code Block 23. MOVZ MIPS Machine Code Test Cases

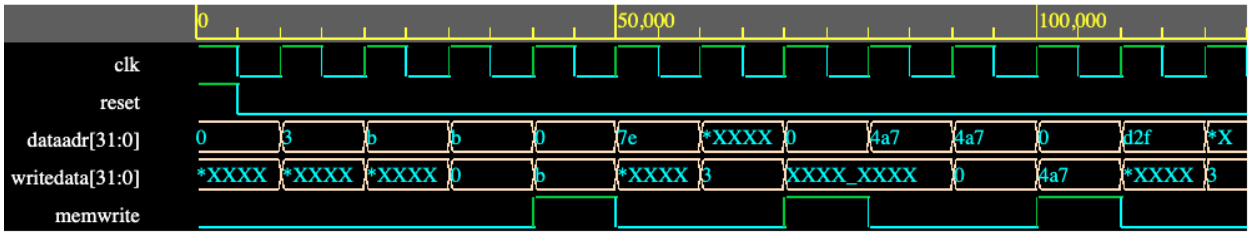

Figure 8. MOVZ Test Cases Waveform 1

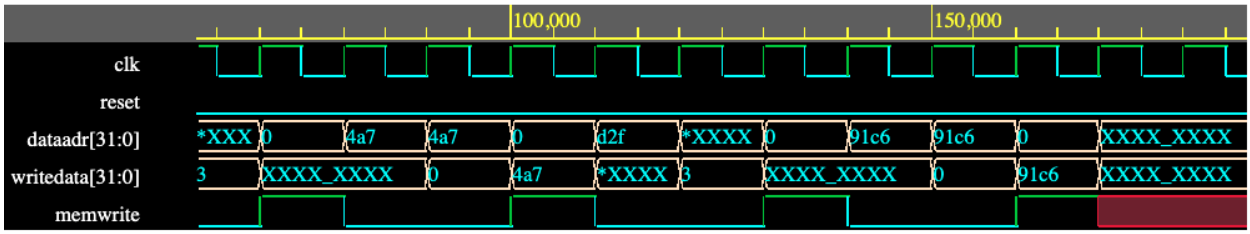

Figure 9. MOVZ Test Cases Waveform 2

2. LHU Testcases

```
# FOR TESTCASES 1 TO 3
li $t0, 49374 (0xC0DE in hex)
li $t1, 16 (0x10 in hex)
sllv $s0, $t0, $t1 (0xC0DE0000)
li $t2, 47806 (0xBABE in hex)
add $s1, $s0, $t2 (0xC0DEBABE)

# Testcase 1
lhu $s2, 44($zero)
sw $s2, 0($zero) # Expected Output: 0xBABE

# Testcase 2
lhu $s3, 2($s1)
sw $s3, 0($zero) # Expected Output: 0xC0DE

# Testcase 3
lhu $s3, 4($s1)
sw $s3, 0($zero) # Expected Output: 0x0

# FOR TESTCASES 4 AND 5
li $t0, 57005 (0xDEAD in hex)
li $t1, 16 (0x10 in hex)
sllv $s0, $t0, $t1 (0xDEAD0000)
li $t2, 48879 (0xBABE in hex)
add $s1, $s0, $t2 (0xDEADBEEF)

# Testcase 4
lhu $s2, 0($s1)
sw $s2, 0($zero) # Expected Output: 0xBEEF

# Testcase 5
lhu $s3, 2($s1)
sw $s3, 0($zero) # Expected Output: 0xDEAD
```

Code Block 24. LHU MIPS Assembly Code Test Cases

```
4408C0DE
44090010
01288004
440ABABE
01508820

96320000
AC120000

96330002
AC130000

96330004
AC130000

4408DEAD
44090010
01288004
440ABEEF
01508820

96320000
AC120000

96330002
AC130000
```

Code Block 25. LHU MIPS Machine Code Test Cases

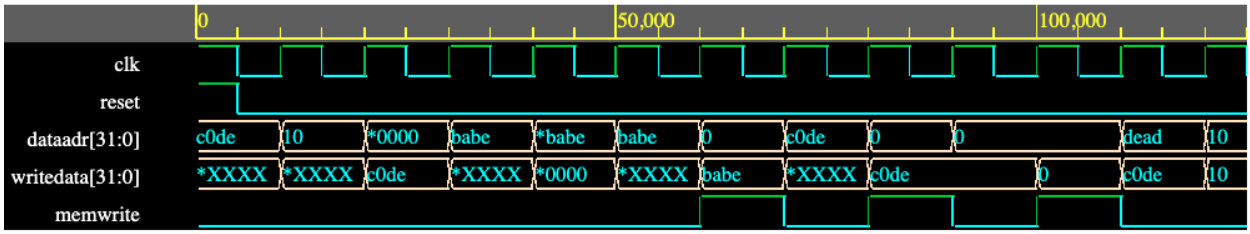

Figure 10. LHU Test Cases Waveform 1

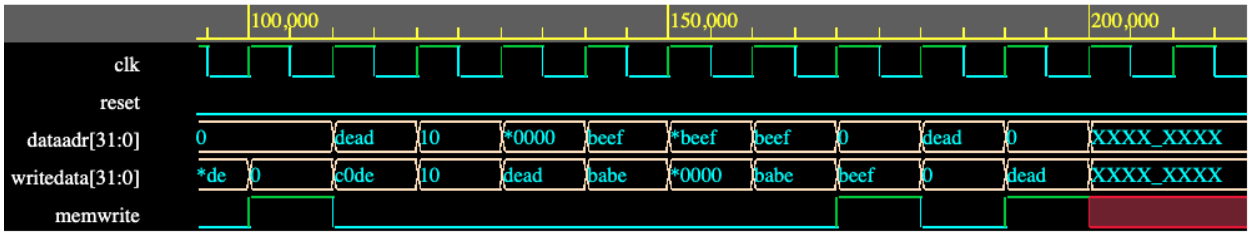

Figure 11. LHU Test Cases Waveform 2

3. SLLV Testcases

```
# Testcase 1
addi $t0, $zero, 20 (0x14 in hex)              20080014
addi $t1, $zero, 10 (0xA in hex)               2009000A
sllv $s0, $t1, $t0                             01098004
sw $s0, 0($zero) # Expected Output: 0xA00000   AC100000

# Testcase 2
addi $t0, $zero, 8 (0x8 in hex)                20080008
addi $t1, $zero, 63 (0x3F in hex)              2009003F
sllv $s0, $t1, $t0                             01098004
sw $s0, 0($zero) # Expected Output: 0x3F00     AC100000

# Testcase 3
addi $t0, $zero, 12 (0xC in hex)               2008000C
addi $t1, $zero, 1443 (0x5A3 in hex)           200905A3
sllv $s0, $t1, $t0                             01098004
sw $s0, 0($zero) # Expected Output: 0x5A3000   AC100000

# Testcase 4
addi $t0, $zero, 4 (0x4 in hex)                20080004
addi $t1, $zero, 3707 (0xE7B in hex)           20090E7B
sllv $s0, $t1, $t0                             01098004
sw $s0, 0($zero) # Expected Output: 0xE7B0     AC100000

# Testcase 5
addi $t0, $zero, 16 (0x10 in hex)              20080010
addi $t1, $zero, 36175 (0x8D4F in hex)         20098D4F
sllv $s0, $t1, $t0                             01098004
sw $s0, 0($zero) # Expected Output: 0x8D4F0000 AC100000
```

Code Block 26. SLLV MIPS Assembly Code Test Cases

Code Block 27. SLLV MIPS Machine Code Test Cases

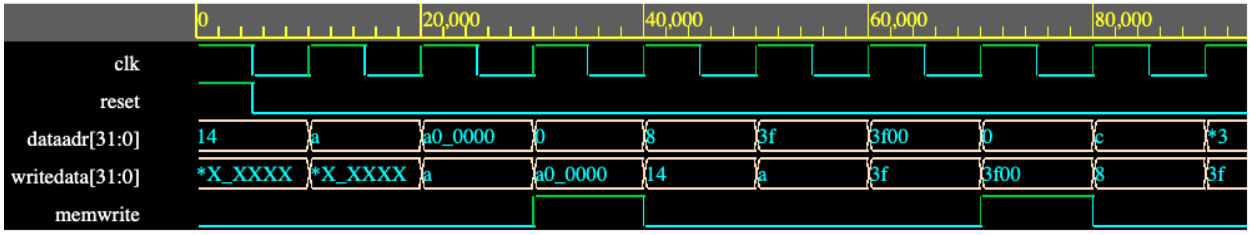

Figure 12. SLLV Test Cases Waveform 1

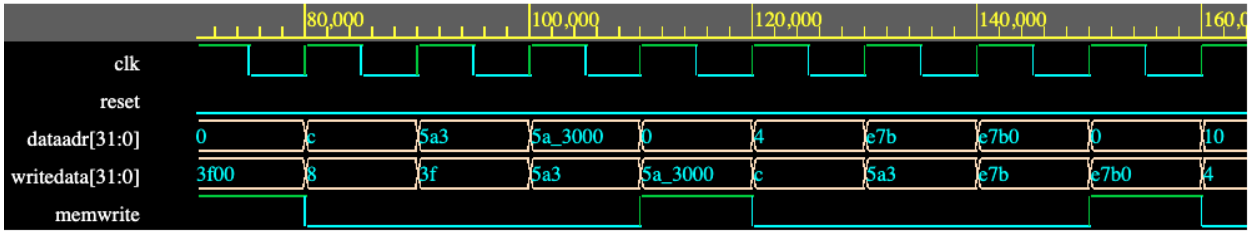

Figure 13. SLLV Test Cases Waveform 2

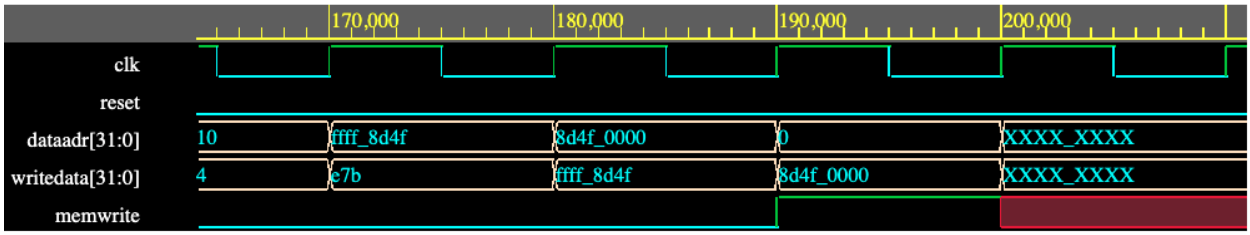

Figure 14. SLLV Test Cases Waveform 3

4. BLT Testcases

```
# Testcase 1
addi $t0, $zero, 0 (0x0 in hex)
addi $t1, $zero, 12 (0xC in hex)
blt $t0, $t1, 1 (0x1 in hex)
addi $t0, $t0, 5 (0x5 in hex)
addi $t0, $t0, 2 (0x2 in hex)
addi $t0, $t0, 3 (0x3 in hex)
sw $t0, 0($zero) # Expected Output: 0x5

# Testcase 2
addi $t0, $zero, 0 (0x0 in hex)
addi $t1, $zero, 12 (0xC in hex)
blt $t1, $t0, 1 (0x1 in hex)
addi $t0, $t0, 5 (0x5 in hex)
addi $t0, $t0, 2 (0x2 in hex)
addi $t0, $t0, 3 (0x3 in hex)
sw $t0, 0($zero) # Expected Output: 0xA

# Testcase 3
addi $t0, $zero, 126 (0x7E in hex)
addi $t1, $zero, 45 (0x2D in hex)
blt $t0, $t1, 1 (0x1 in hex)
addi $t0, $t0, 4 (0x4 in hex)
addi $t0, $t0, 7 (0x7 in hex)
addi $t0, $t0, 3 (0x3 in hex)
sw $t0, 0($zero) # Expected Output: 0x8C

# Testcase 4
addi $t0, $zero, 126 (0x7E in hex)
addi $t1, $zero, 45 (0x2D in hex)
blt $t1, $t0, 1 (0x1 in hex)
addi $t0, $t0, 4 (0x4 in hex)
addi $t0, $t0, 7 (0x7 in hex)
addi $t0, $t0, 3 (0x3 in hex)
sw $t0, 0($zero) # Expected Output: 0x88

# Testcase 5
addi $t0, $zero, 126 (0x7E in hex)
addi $t1, $zero, 45 (0x2D in hex)
blt $t1, $t0, 2 (0x2 in hex)
addi $t0, $t0, 4 (0x4 in hex)
addi $t0, $t0, 7 (0x7 in hex)
addi $t0, $t0, 24 (0x18 in hex)
sw $t0, 0($zero) # Expected Output: 0x96
```

Code Block 28. BLT MIPS Assembly Code Test Cases

```
20080000
2009000C
75090001
21080005
21080002
21080003
AC080000

20080000
2009000C
75280001
21080005
21080002
21080003
AC080000

2008007E
2009002D
75090001
21080004
21080007
21080003
AC080000

2008007E
2009002D
75280001
21080004
21080007
21080003
AC080000

2008007E
2009002D
75280002
21080004
21080007
21080018
AC080000
```

Code Block 29. BLT MIPS Machine Code Test Cases

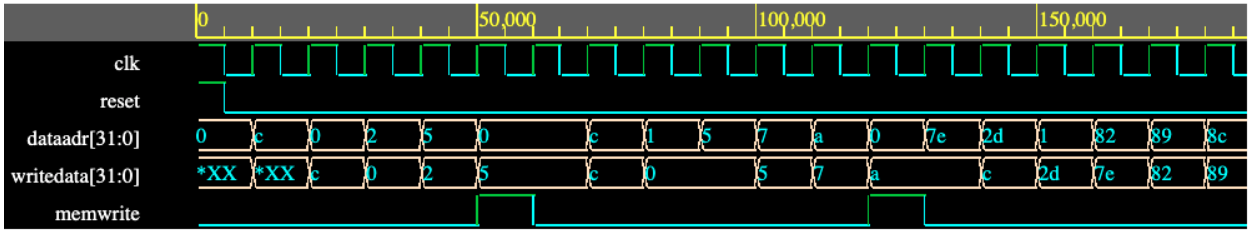

Figure 15. BLT Test Cases Waveform 1

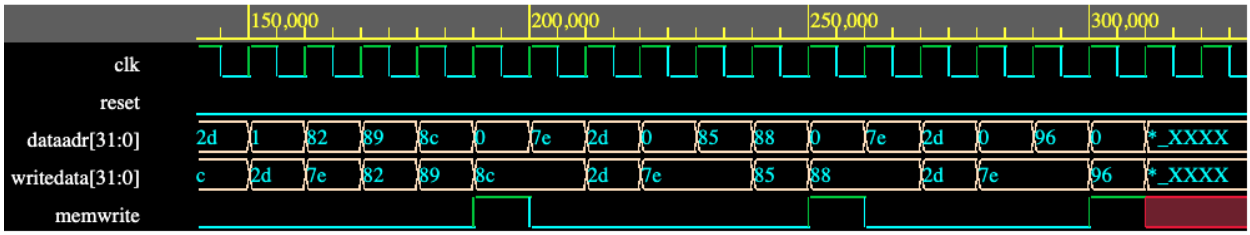

Figure 16. BLT Test Cases Waveform 2

5. LI Testcases

```
# Testcase 1
li $s0, 15 (0xF in hex)
sw $s0, 0($zero) # Expected Output: 0xF

# Testcase 2
li $s0, 165 (0xA5 in hex)
sw $s0, 0($zero) # Expected Output: 0xA5

# Testcase 3
li $s0, 60 (0x3C in hex)
sw $s0, 0($zero) # Expected Output: 0x3C

# Testcase 4
li $s0, 3631 (0xE2F in hex)
sw $s0, 0($zero) # Expected Output: 0xE2F

# Testcase 5
li $s0, 32075 (0x7D4B in hex)
sw $s0, 0($zero) # Expected Output: 0x7D4B
```

Code Block 30. LI MIPS Assembly Code Test Cases

```
4410000F
AC100000

441000A5
AC100000

4410003C
AC100000

44100E2F
AC100000

44107D4B
AC100000
```

Code Block 31. LI MIPS Machine Code Test Cases

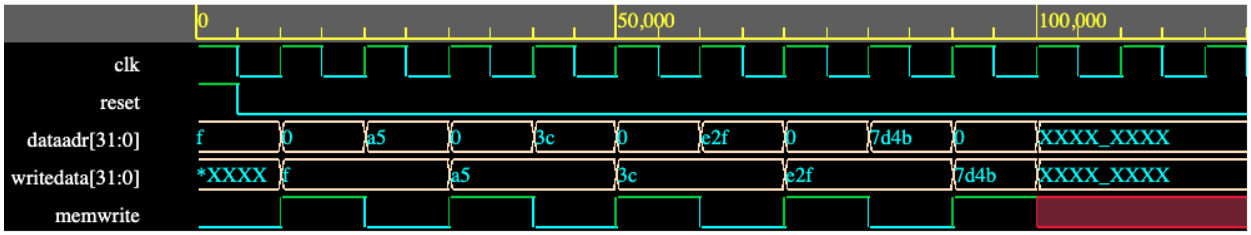

Figure 17. LI Test Cases Waveform

6. MIX4 Testcases

```
# FOR TESTCASES 1 AND 2
li $s0, 49374 (0xC0DE in hex)
li $s1, 16 (0x10 in hex)
sllv $t0, $s1, $s0 (0xC0DE0000)
li $s2, 47806 (0xBABE in hex)
add $t1, $t0, $s2 (0xC0DEBABE)
li $s3, 57005 (0xDEAD in hex)
li $s4, 16 (0x10 in hex)
sllv $t2, $s4, $s3 (0xDEAD0000)
li $s5, 48879 (0xBEEF in hex)
add $t3, $t2, $s5 (0xDEADBEEF)

# Testcase 1
mix4 $t4, $t1, $t3
sw $t4, 0($zero) # Expected Output:0xC0ADBAEF

# Testcase 2
mix4 $t5, $t3, $1
sw $t5, 0($zero) # Expected Output:0xDEDEBEBE

# FOR TESTCASES 3 AND 4
li $s0, 45242 (0xB0BA in hex)
li $s1, 16 (0x10 in hex)
sllv $t0, $s1, $s0 (0xB0BA0000)
li $s2, 7397 (0x1CE5 in hex)
add $t1, $t0, $s2 (0xB0BA1CE5)
li $s3, 44061 (0xAC1D in hex)
li $s4, 16 (0x10 in hex)
sllv $t2, $s4, $s3 (0xAC1D0000)
li $s5, 51966 (0xCAFE in hex)
add $t3, $t2, $s5 (0xAC1DCAFE)

# Testcase 3
mix4 $t4, $t1, $t3
sw $t4, 0($zero) # Expected Output:0xB01D1CFE

# Testcase 4
mix4 $t5, $t3, $1 0xAC
sw $t5, 0($zero) # Expected Output:0xACBACAE5
```

Code Block 32. MIX4 MIPS Assembly Code Test Cases

```
4410C0DE
44110010
02304004
4412BABE
01124820
4413DEAD
44140010
02935004
4415BEEF
01555820

012B6033
AC0C0000

01696833
AC0D0000

4410B0BA
44110010
02304004
44121CE5
01124820
4413AC1D
44140010
02935004
4415CAFE
01555820

012B6033
AC0C0000

01696833
AC0D0000
```

Code Block 33. MIX4 MIPS Machine Code Test Cases

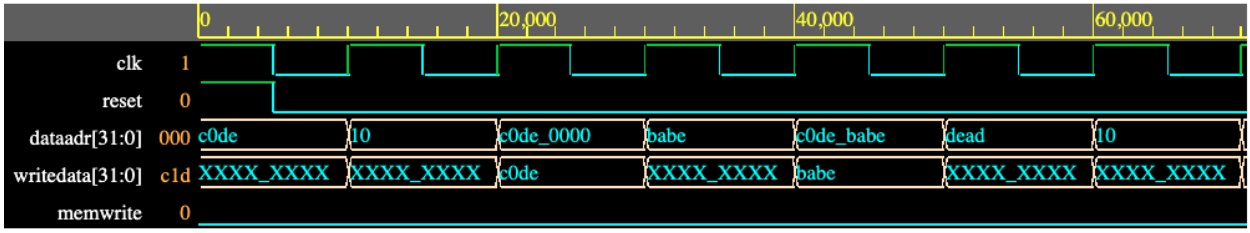

Figure 18. MIX4 Test Cases Waveform 1

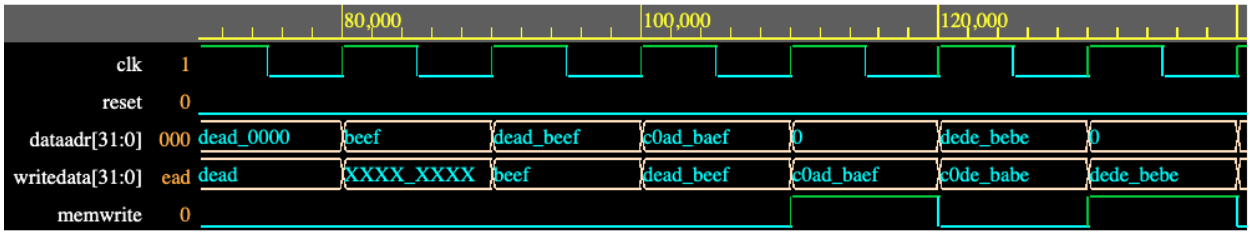

Figure 19. MIX4 Test Cases Waveform 2

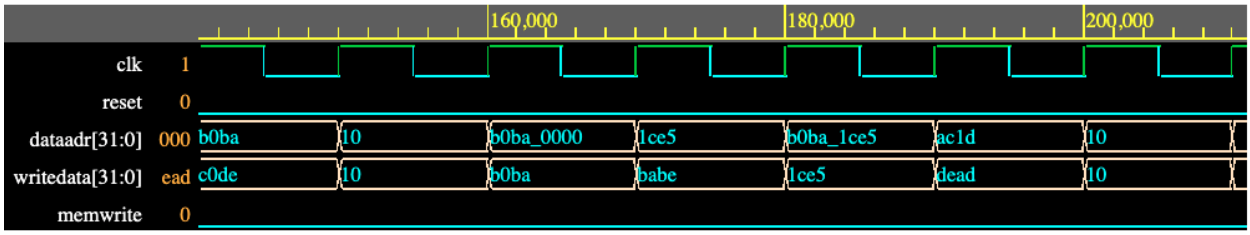

Figure 20. MIX4 Test Cases Waveform 3

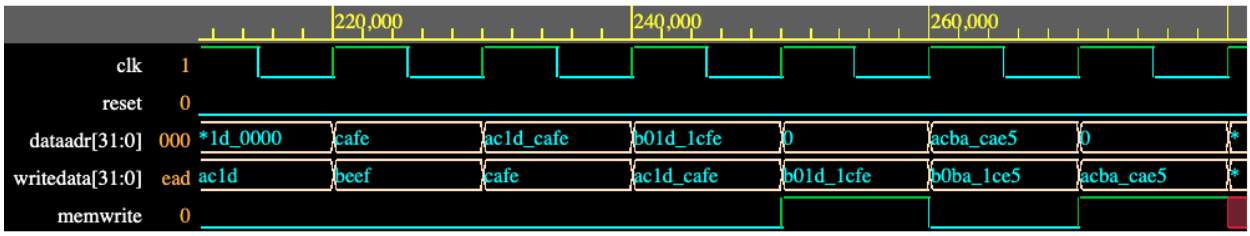

Figure 21. MIX4 Test Cases Waveform 4

Updated Schematic Diagrams

After a thorough series of modifications to the Main Decoder, ALU Decoder, and ALU modules, the processor has undergone a significant enhancement in its instruction execution capabilities, compared to its pre-modification state. With these upgrades, the processor can now seamlessly execute a broader range of instructions. The updated schematic diagrams for the MainDec, ALUDec, and ALU modules are illustrated in Figures 22, 23, and 24, respectively. The new schematic diagrams of these modules showcase the detailed changes implemented to achieve the given task of extending the given MIPS single-cycle processor.

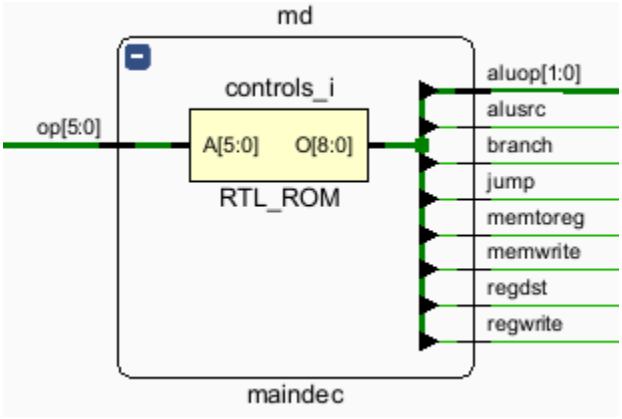

Figure 22. New Schematic Diagram of Main Decoder Module

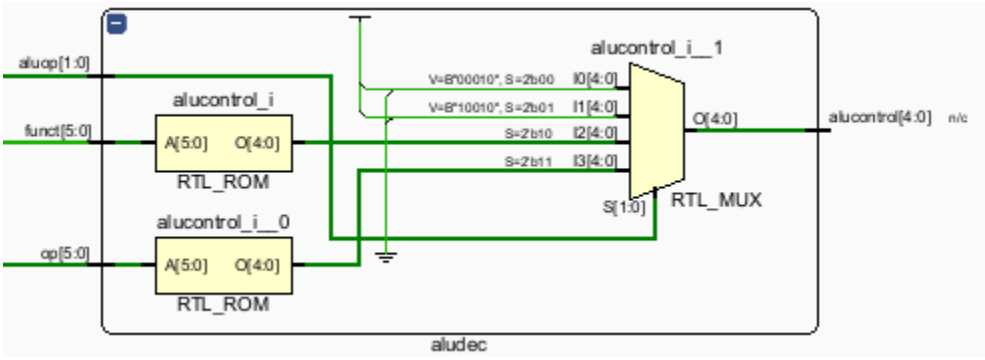

Figure 23. New Schematic Diagram of ALU Decoder Module

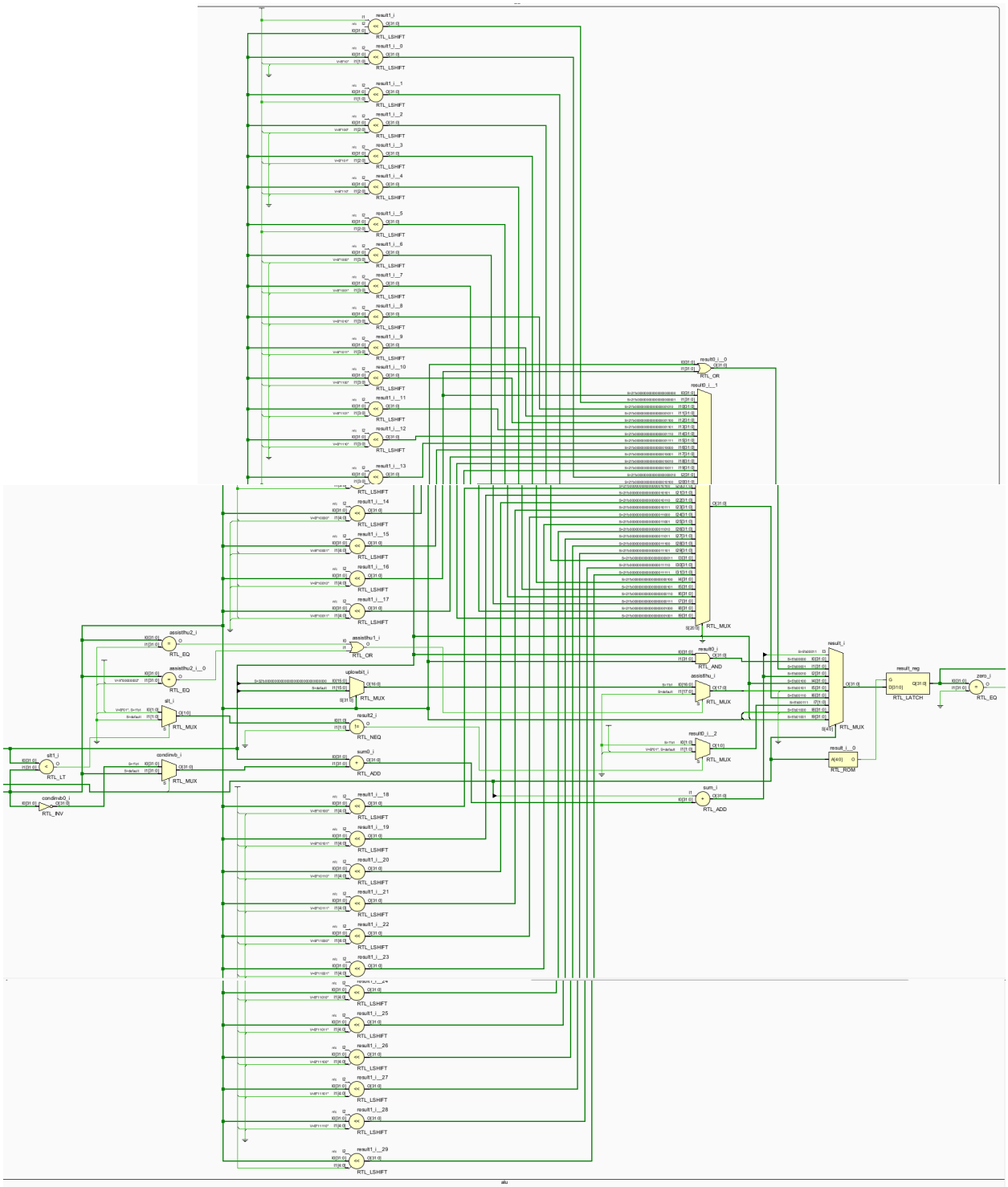

Figure 24. New Schematic Diagram of ALU Module

References

Liston, K. (n.d.). *MIPS Reference Sheet*. MIPS reference sheet. Retrieved January 22, 2024. https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_help.html