VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRONICS

*

EE4423: COMPUTER ARCHITECTURE
Milestone 1: Design of a Vending Machine

Instructor	:	**Dr. Trần Hoàng Linh**
TA	:	**M.S. Cao Xuân Hải**
Subject	:	**Computer Architecture**
Group	:	**02**
Members	:	**Lương Triển Thắng - 2051194**
		Đinh Hoàng Luân - 2051145
		Nguyễn Nhật Nam - 2051153

Ho Chi Minh City, October 13, 2023

Contents

List of Figures

1. Abstract

2. Introduction

3. Design Strategy

3.1 ALU module

In the RISC-V (Reduced Instruction Set Computer - Five) architecture, an ALU (Arithmetic Logic Unit) is a fundamental component responsible for performing arithmetic and logical operations on binary data. The ALU is responsible for executing various operations such as addition, subtraction, logical XOR, logical AND, logical OR, Set less than, set less than unsigned, Shift left logic, Shift right logic, and Shift right arithmetic

The RISC-V ALU typically operates on fixed-length binary data, which is typically 32 bits or 64 bits depending on the specific RISC-V implementation. It takes input operands from the CPU's registers, performs the specified operation, and stores the result back into a register.

The ALU executes instructions of different types, including R-type and I-type instructions.

- `R-type` instructions in RISC-V are used for arithmetic and logical operations that involve registers. In `R-type` instructions, the ALU performs the specified operation on the values in `rs1` and `rs2` and stores the result in `rd`.
- `I-type` instructions in RISC-V are used for immediate operations where one operand is an immediate value, typically a constant or a small value. In `I-type` instructions, the ALU performs the specified operation between the value in rs1 and the immediate value `imm`, and stores the result in `rd`.

Table 3.1: The operations an RV32I ALU needs to be implemented

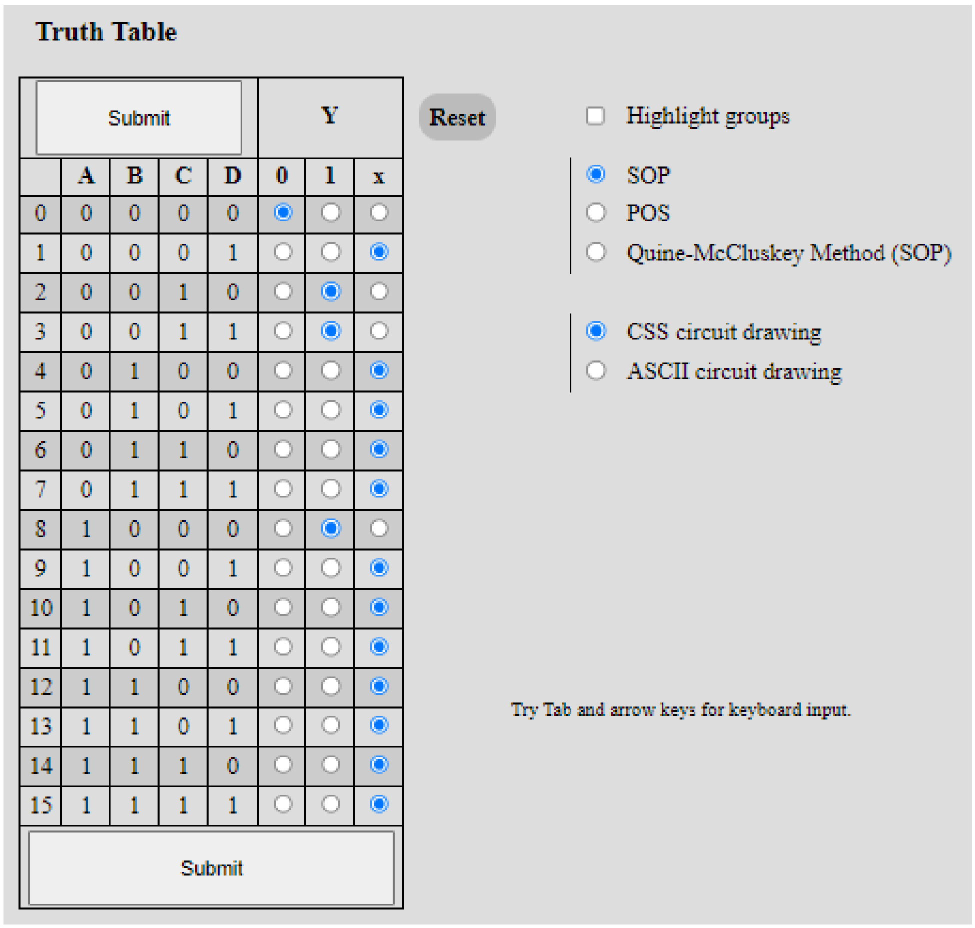
Opcode	**alu_op**	**R-type**	**I-type**
0000	ADD	rd = rs1 + rs2	rd = rs1 + imm
1000	SUB	rd = rs1 - rs2	N/A
0010	SLT	rd = (rs1 < rs2) ? 1 : 0	rd = (rs1 < imm) ? 1 : 0
0011	SLTU	rd = (rs1 < rs2) ? 1 : 0	rd = (rs1 < imm) ? 1 : 0
0100	XOR	rd = rs1 $\oplus$ rs2	rd = rs1 $\oplus$ imm
0110	OR	rd = rs1 $\vee$ rs2	rd = rs1 $\vee$ imm
0111	AND	rd = rs1 $\wedge$ rs2	rd = rs1 $\wedge$ imm
0001	SLL	rd = rs1 $\ll$ rs2 [4:0]	rd = rs1 $\ll$ imm [4:0]
0101	SRL	rd = rs1 $\gg$ rs2 [4:0]	rd = rs1 $\gg$ imm [4:0]
1101	SRA	rd = rs1 $\gg$ rs2 [4:0]	rd = rs1 $\gg$ imm [4:0]
1001	B	rs2	rs2

The purpose of adding opcodes to establish a truth table is to determine the behavior of `unsigned_sel` and `neg_sel` operations. These opcodes are used to perform selection and negation operations on unsigned values in a digital system. By creating a truth table, we can systematically analyze and define the output values of these operations based on different input combinations.

Table 3.2: Truth table for ALU signals

Operation	alu_op	neg_sel	unsigned_sel
ADD	0000	0	0
SLL	0001	X	X
SLT	0010	1	0
SLTU	0011	1	1
XOR	0100	X	X
SLR	0101	X	X
OR	0110	X	X
AND	0111	X	X
SUB	1000	1	0
N/A	1001	X	X
N/A	1010	X	X
N/A	1011	X	X
N/A	1100	X	X
SRA	1101	X	X
N/A	1110	X	X
N/A	1111	X	X

Then we use the Truth table to given the output of `unsigned_sel` and `neg_sel`



Truth Table

					Y		
	A	B	C	D	0	1	x
0	0	0	0	0	●	○	○
1	0	0	0	1	○	○	●
2	0	0	1	0	○	●	○
3	0	0	1	1	○	●	○
4	0	1	0	0	○	○	●
5	0	1	0	1	○	○	●
6	0	1	1	0	○	○	●
7	0	1	1	1	○	○	●
8	1	0	0	0	○	●	○
9	1	0	0	1	○	○	●
10	1	0	1	0	○	○	●
11	1	0	1	1	○	○	●
12	1	1	0	0	○	○	●
13	1	1	0	1	○	○	●
14	1	1	1	0	○	○	●
15	1	1	1	1	○	○	●

☐ Highlight groups

[✓ SOP] POS Quine-McCluskey Method (SOP)

[✓ CSS circuit drawing] ASCII circuit drawing

Try Tab and arrow keys for keyboard input.

Figure 3.1: Truth table for `neg_sel`

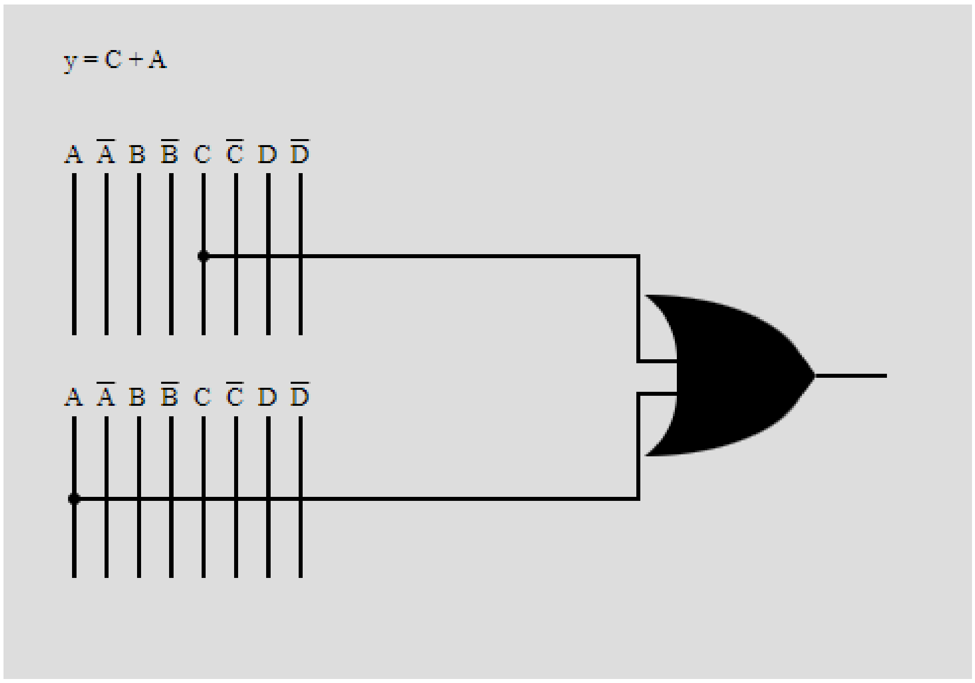
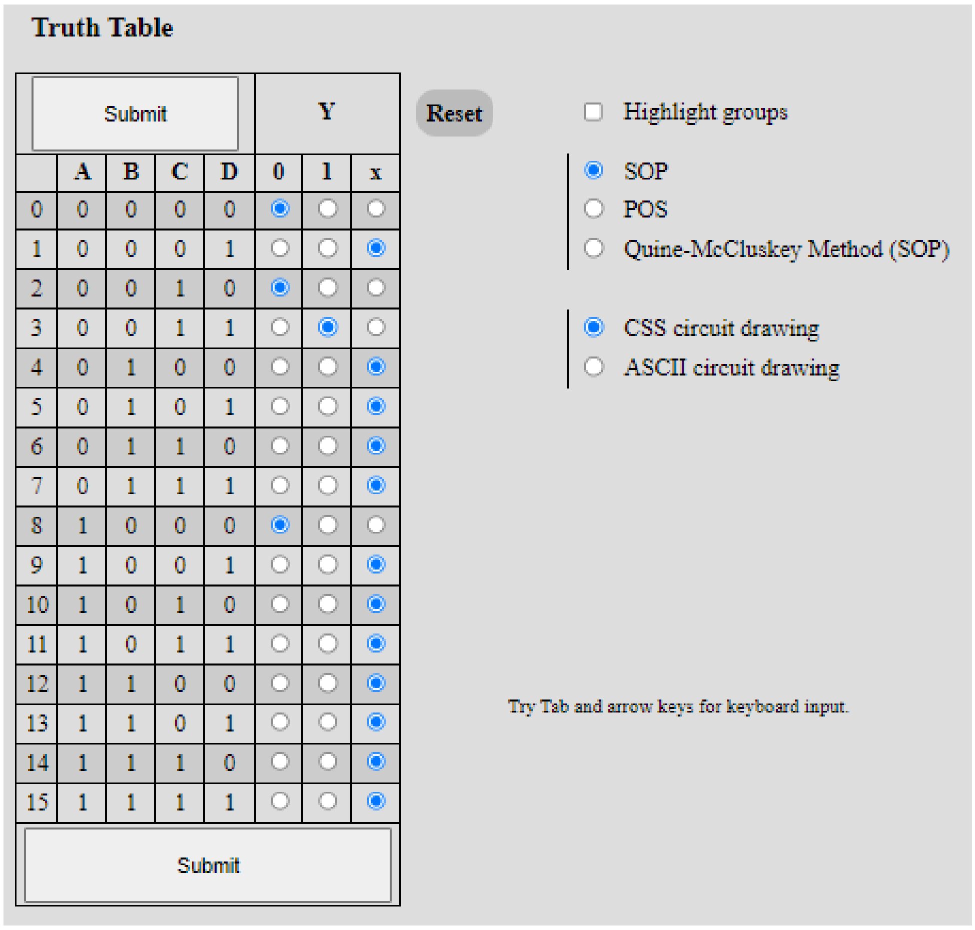


Figure 3.2: The output of `neg_sel`



Truth Table

Submit | Reset | ☐ Highlight groups

[✓ SOP] POS Quine-McCluskey Method (SOP)

[✓ CSS circuit drawing] ASCII circuit drawing

	A	B	C	D	0	1	x
0	0	0	0	0	●	○	○
1	0	0	0	1	○	○	●
2	0	0	1	0	●	○	○
3	0	0	1	1	○	●	○
4	0	1	0	0	○	○	●
5	0	1	0	1	○	○	●
6	0	1	1	0	○	○	●
7	0	1	1	1	○	○	●
8	1	0	0	0	●	○	○
9	1	0	0	1	○	○	●
10	1	0	1	0	○	○	●
11	1	0	1	1	○	○	●
12	1	1	0	0	○	○	●
13	1	1	0	1	○	○	●
14	1	1	1	0	○	○	●
15	1	1	1	1	○	○	●

(Y)

Submit

Try Tab and arrow keys for keyboard input.

Figure 3.3: Truth table for `unsigned_sel`

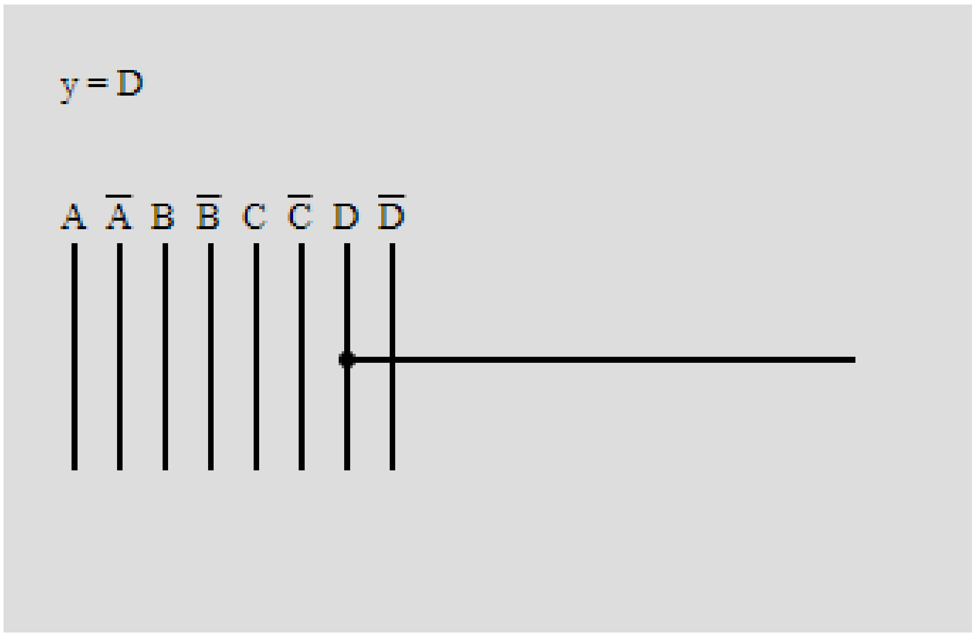


Figure 3.4: The output of `unsigned_sel`

The `AddSub` module is a component in designing an Arithmetic Logic Unit, `AddSub` that performs addition, subtraction operations. This module is designed to select between addition and subtraction based on a control signal (`un_signedl`).

The module takes two input operands, a control signal for operation selection (`neg_sel`), and a control signal for signed or unsigned comparison (`un_signedl`). When the `neg_sel` signal is 1, it indicates an addition operation. In this case, the module performs the addition of the two operands and generates a 32-bit result. This result represents the outcome of the addition operation. At the output, the module separates the result into a 32-bit output and a 1-bit trigger signal.

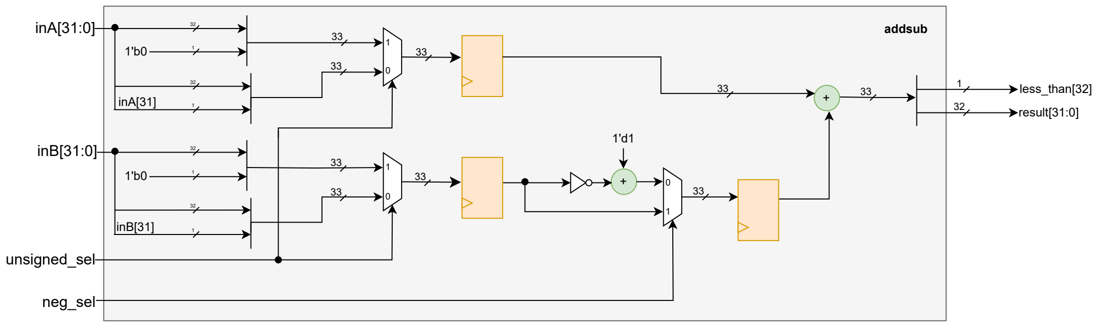


Figure 3.5: `addsub` design

After builing the addsub module,we conduct to integrate the module addsub in to the ALU, the réult of alu will depend on the sel signal

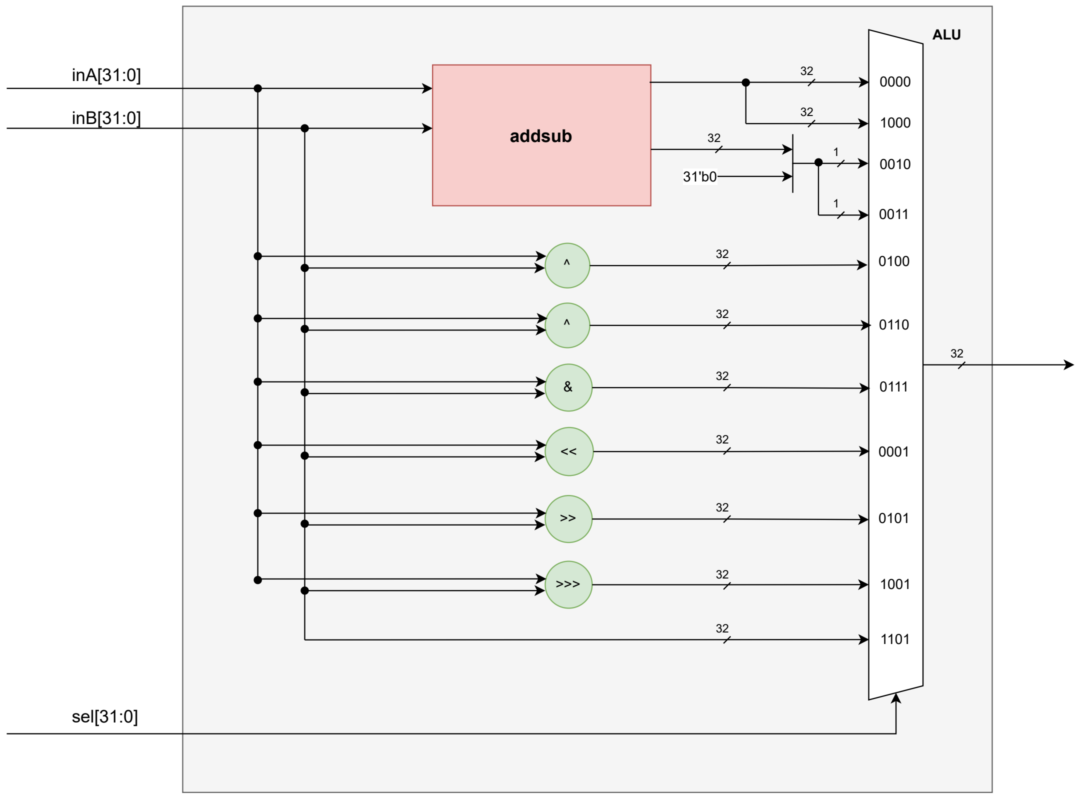


Figure 3.6: `alu` design

3.2 Branch Condition

3.2.1 brcomp

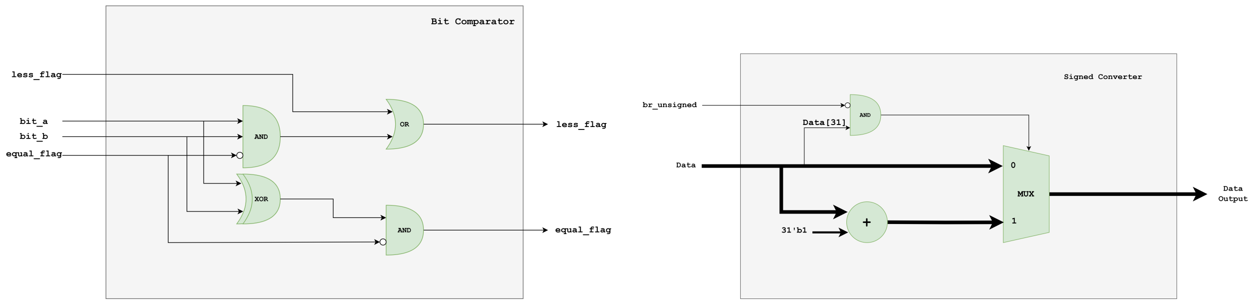


Figure 3.7

The branch comparator, relies on the functionality of two core modules: the bit comparator and the sign converter. The bit comparator handles the intricate task of comparing individual bits in the inputs, `rs1_data` and `rs2_data`, while considering the prior comparison results indicated by `in_less` and `in_equal` for higher bits. Its primary goal is to produce two one-bit outputs, `out_less` and `out_equal`, which signify whether `rs1_data` is less than or equal to texttttrs2_data, respectively. This is achieved through a clever combination of XOR and AND gates, meticulously designed to perform the bit-by-bit comparisons effectively.

On the other hand, the sign converter plays a crucial role in processing 32-bit inputs, either `rs1_data` or `rs2_data`, by considering the context of signed or unsigned comparisons, as indicated by the `br_unsigned` signal. It delivers 32-bit outputs, `data_r1` or `data_r2`, reflecting the converted values based on the input and the selected mode. The sign converter leverages a multiplexer to make the choice between preserving the original value and applying a two's complement transformation, depending on the sign bit and the`br_unsigned` signal. This operation is facilitated through a combination of NOT gates and an adder, ensuring an accurate representation of the converted data

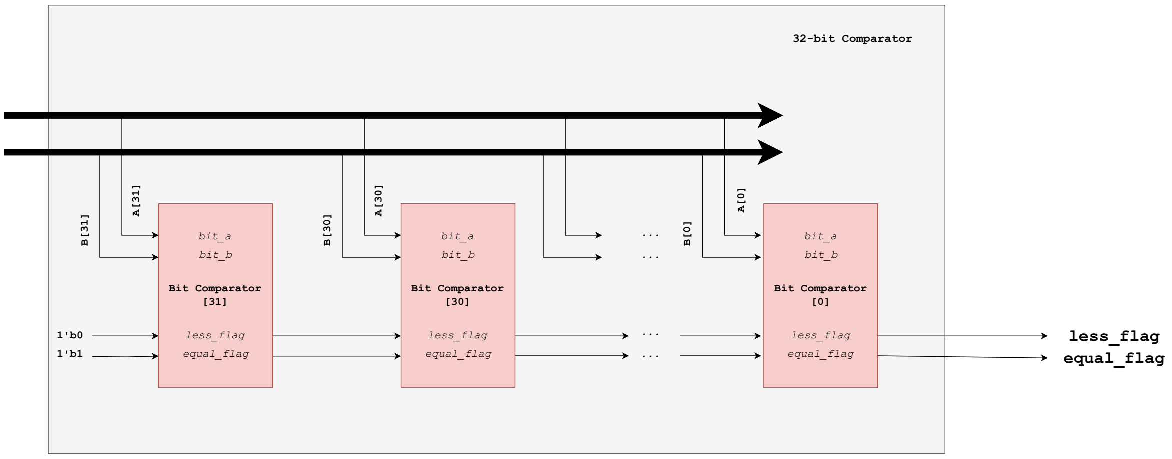


Figure 3.8: `alu` design

The bit comparator's intrinsic capability to compare a single bit necessitates the use of 32 such comparators in the context of the RISC-V CPU, arranged in a sequential series. A noteworthy aspect of this arrangement is that the 31st bit comparator must be linked to the 'equal' and 'less' inputs with values 1 and 0, respectively, a design choice that assumes any 33rd bit not within the defined 32-bit range is considered equal. This precision ensures a meticulous comparison of the two data sets. The resulting output flag signals are interconnected in a series and serve as inputs for the subsequent module, collectively contributing to the creation of a conventional 32-bit comparator. The provided code within the 'regcomp' module effectively captures this concept through a 'generate' loop, wherein bit comparators are instantiated and interconnected to enable precise 32-bit comparisons.

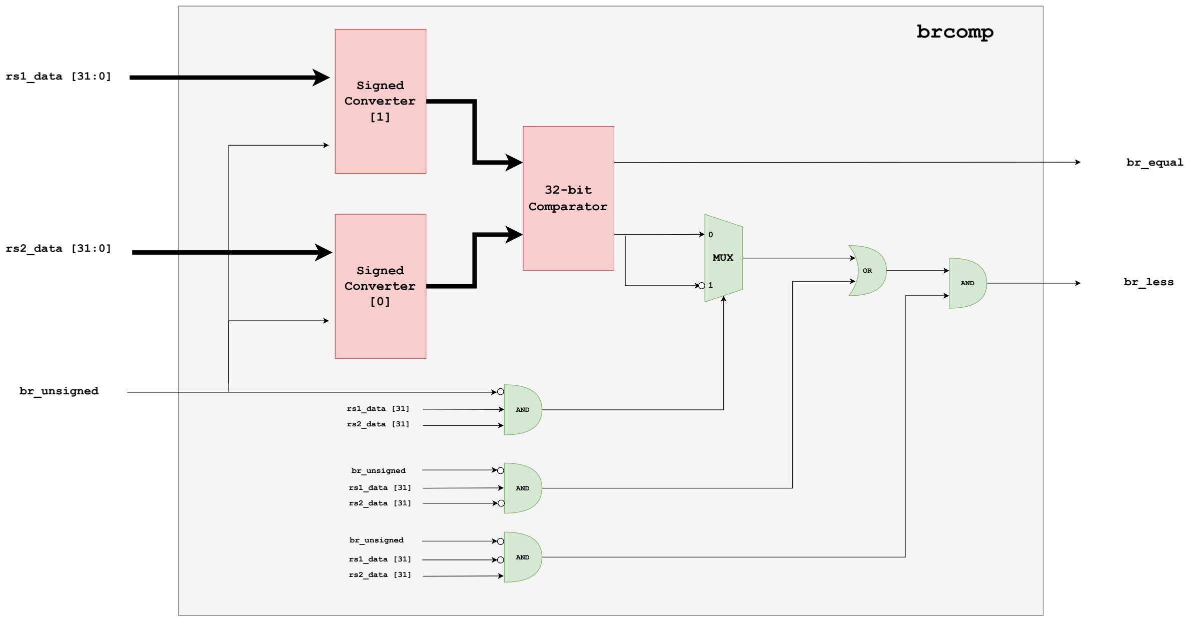


Figure 3.9: `alu` design

The apex of the branch comparator's design is represented by its top-level architecture, an integral component of the Execution stage within a single-cycle RISC-V CPU. This branch comparator encompasses key input ports, namely `rs1_data`, `rs2_data`, and `br_unsigned`, while yielding crucial output ports in the form of `br_less` and `br_equal`. The architectural illustration portrays the interconnection of two sign converters, each receiving data from one of the input ports, and subsequently feeding this processed data into two 32-bit comparators. This configuration lays the foundation for the branch comparator's core structure. However, it introduces a notable challenge, centered around controlling the 'less' signal under distinct conditions for both unsigned and signed comparisons. Unsigned comparisons adhere to a straightforward design, as seen in the former architecture. Conversely, signed comparisons give rise to three distinct scenarios. To present these cases systematically, a table could be constructed, outlining the conditions and their corresponding outcomes. Notably, the 31st bit, which also serves as the signed bit, undergoes meticulous processing through a logic combination circuit, ensuring the precise generation of the 'less' signal across all scenarios.

Condition	Action
rs1[31]=1 and rs2[31]=1 (both neg)	Convert data and complement 'less' output
rs1[31]=1 (neg) and rs2[31]=0 (pos)	'Less' is immediately set to 'on'
rs1[31]=0 (neg) and rs2[31]=1 (pos)	'Less' is immediately set to 'off'

3.2.2 immgen

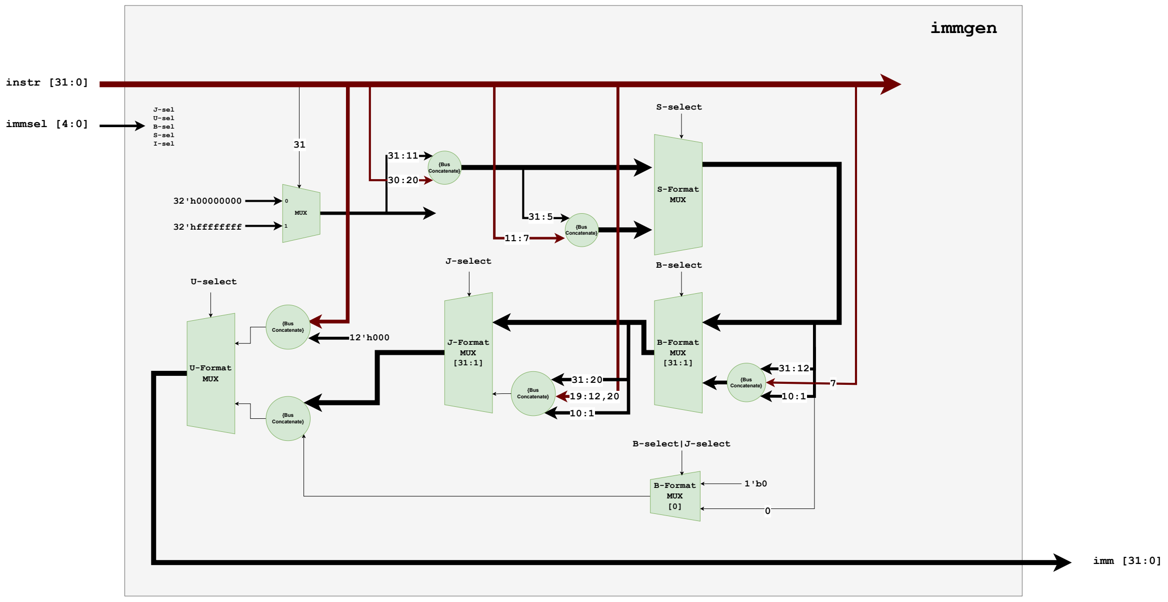


Figure 3.10: `alu` design

The implementation of the immediate generator is a critical component in the design of a RISC-V CPU, responsible for arranging the bits or bytes of instructions to form the appropriate immediate values. In the RISC-V format, there are various types of immediates, including I, B, S, J, and U formats, which require the use of multiplexers to direct the wiring order of the bus. The figure illustrates a design that guides the bus of immediates for these different formats. The I-format serves as the starting point, requiring minimal sign extension and acting as a reference for subsequent stages. The S-stage, identified by a MUX, transforms the bus into an S-type immediate, distinct from the I-type. Similar approaches are applied to B-type masking over S-type and J-type masking over J-type, with special note taken for the commonality of the last bits in J and B types, which are handled separately to reduce the need for additional MUXes. The U-type is entirely different from the others, and the chosen mechanism is applied to determine whether it is a U-type or part of the multi-stage ISBJ formats. The selection of the immediate bus path is governed by the 'imm_sel' port, which is protocol decoded in a one-hot format with 5 bits, ensuring precise and effective control of immediate generation. The provided code in the 'immgen' module aligns with this description, providing a detailed and accurate portrayal of the immediate generation process for the RISC-V CPU.

The instruction 32 bit in Single Cycle RISCV CPU is composed of different fields that encode the operation, the operands and the destination of the result. One of these fields is the bit that, which indicates whether the instruction is an immediate or a register instruction. The table below shows some summary of how the bit that affects the generation of immediate values from the instruction 32 bit in Single Cycle RISCV CPU.

Immediate format	**Size**	**Descriptio**
I-type	signxt[31:11],instr[30:20]	Used for load instr
S-type	Itype[31:5],instr[11:7]	Used for store instr
B-type	Stype[31:12],instr[7],Stype[10:1],1'b0	Used for branch inst
U-type	instr[31:12],12'd0	Used for lui and auipc
J-type	Btype[31:20],instr[19:12],instr[20],Btype[10:1],1'b0	Used for j and jal ins

3.3 Logic Control Unit

	opcode	inst[6:2]	inst[14:12]	inst[30]	BrEq	BrLT	PCSel	ImmSel	BrUn	Asel	Bsel	ALUSel	MemRW	RegWEn	WBSel	LdStSel
R-type	add	01100	000	0			0			0	0	0000	0	1	01	
	sub	01100	000	1			0			0	0	1000	0	1	01	
	xor	01100	100	0			0			0	0	0100	0	1	01	
	or	01100	110	0			0			0	0	0110	0	1	01	
	and	01100	111	0			0			0	0	0111	0	1	01	
	sll	01100	001	0			0			0	0	0001	0	1	01	
	srl	01100	101	0			0			0	0	0101	0	1	01	
	sra	01100	101	1			0			0	0	1101	0	1	01	
	slt	01100	010	0			0			0	0	0010	0	1	01	
	sltu	01100	011	0			0			0	0	0011	0	1	01	
I-type	addi	00100	000				0	000 (I)		0	1	0000	0	1	01	
	xori	00100	100				0	000 (I)		0	1	0100	0	1	01	
	ori	00100	110				0	000 (I)		0	1	0110	0	1	01	
	andi	00100	111				0	000 (I)		0	1	0111	0	1	01	
	slli	00100	001	0			0	000 (I)		0	1	0001	0	1	01	
	srli	00100	101	0			0	000 (I)		0	1	0101	0	1	01	
	srai	00100	101	1			0	000 (I)		0	1	1101	0	1	01	
	slti	00100	010				0	000 (I)		0	1	0010	0	1	01	
	sltiu	00100	011				0	000 (I)		0	1	0011	0	1	01	
	lb	00000	000				0	000 (I)		0	1	0000	0	1	00	000
	lh	00000	001				0	000 (I)		0	1	0000	0	1	00	001
	lw	00000	010				0	000 (I)		0	1	0000	0	1	00	010
	lbu	00000	100				0	000 (I)		0	1	0000	0	1	00	100
	lhu	00000	101				0	000 (I)		0	1	0000	0	1	00	101
S-type	sb	01000	000				0	001 (S)		0	1	0000	1	0		000
	sh	01000	001				0	001 (S)		0	1	0000	1	0		001
	sw	01000	010				0	001 (S)		0	1	0000	1	0		010

	opcode	inst[6:2]	inst[14:12]	inst[30]	BrEq	BrLT	PCSel	ImmSel	BrUn	Asel	Bsel	ALUSel	MemRW	RegWEn	WBSel	LdStSel
B-type	beq	11000	000		0		0	010 (B)		1	1	0000	0	0		
	beq	11000	000		1		1	010 (B)		1	1	0000	0	0		
	bne	11000	001		0		1	010 (B)		1	1	0000	0	0		
	bne	11000	001		1		0	010 (B)		1	1	0000	0	0		
	blt	11000	100			1	1	010 (B)	0	1	1	0000	0	0		
	blt	11000	100			0	0	010 (B)	0	1	1	0000	0	0		
	bltu	11000	110			1	1	010 (B)	1	1	1	0000	0	0		
	bltu	11000	110			0	0	010 (B)	1	1	1	0000	0	0		
	bge	11000	101		1	0	1	010 (B)	0	1	1	0000	0	0		
	bge	11000	101		0	0	1	010 (B)	0	1	1	0000	0	0		
	bge	11000	101		0	1	0	010 (B)	0	1	1	0000	0	0		
	bgeu	11000	111		1	0	1	010 (B)	1	1	1	0000	0	0		
	bgeu	11000	111		0	0	1	010 (B)	1	1	1	0000	0	0		
	bgeu	11000	111		0	1	0	010 (B)	1	1	1	0000	0	0		
J-type	jal	11011					1	100 (J)		1	1	0000	0	1	10	
JI-type	jalr	11001					1	000 (I)		0	1	0000	0	1	10	
U-type	lui	01101					0	011 (U)			1	1001	0	1	01	
	auipc	00101					0	011 (U)		1	1	0000	0	1	01	

From the table, we can observe the following encoding patterns for different instruction types:

- For R-type instructions, `inst[6:2] == 01100`.
- For I-type instructions, `inst[6:2] == 00100` or `00000`.
- For S-type instructions, the condition is `inst[6:2] == 01000`.
- For B-type instructions, the condition is `inst[6:2] == 11000`.
- For J-type instructions, the condition is `inst[6:2] == 11011`.
- For U-type instructions, the condition is `inst[6:2] == 01101` or `00101`.
- For JI-type instructions, the condition is `inst[6:2] == 11001`.

`PCSel` controls the program counter (PC) selection. It is set to 0 for sequential execution (PC + 4), and 1 for non-sequential jumps (PC + immediate). Specifically, `PCSel` is set to 1 for J-type, JI-type, or B-type instructions with a true branch condition, defined as `PCSel = Jtype | JItype | (Btype & branchTrue)`.

`ImmSel` determines the immediate extension mode, which specifies how to correctly extend the immediate value based on the type of the instruction. Thus, `ImmSel = {Jtype, Utype, Btype, Stype, Itype | JItype}`.

`BrUn` is a signal used in the Branch Comparison module. It is set to 0 when the branch instruction is for signed comparison and is set to 1 when the branch instruction is for unsigned comparison. From the table, we observed that `inst[13]` is the bit for selecting unsigned mode in B-type instruction. Thus, `BrUn = inst[13]`.

`ASel` selects the A-value input for the ALU. It's set to 0 for most instructions, using `rs1` from the Register File. However, for B-type, J-type, and U-type instructions, it's set to 1, indicating the selection of the program counter (PC). So, `ASel = Btype | Jtype | Utype`.

`BSel` selects the B-value input for the ALU. When it's 0, it uses `rs2` from the Register File. In R-type instructions, `BSel` is 0. For all other instructions, `BSel` is 1, indicating the use of the immediate generator output. So, `BSel = ~Rtype`.

`MemRW` enables writing to the data memory, specifically in the Load-Store Unit (LSU). It is only enabled for S-type instructions. So, `MemRW = Stype`.

`RegWEn` is the enable signal for writing to the Register File. It allows write operations to the register file. It is active for most instruction types, except for S-type and B-type instructions, where it is not used for register file writing. So, `RegWEn = ~Stype & ~Btype`.

`WBSel` determines the data source for writing to the register file.

- 00: Selects the output data from the Load-Store Unit (LSU).
- 01: Selects the output of the Arithmetic Logic Unit (ALU).
- 10: Selects the value of PC + 4, applicable for `jal` and `jalr` instructions.

⇒ `WBSel = Ltype ? 00 : (Jtype | JItype) ? 10 : 01`

`ALUSel` is a dynamic selection signal based on instruction type, and its behavior is determined as follows:

- For R-type instructions, it is determined by `{inst[30], inst[14:12]}`.

- For I-type instructions (excluding loads (Ltype)), if the instruction is a shift operator (`sll`, `srl`, `sra`), it is {`inst[30], inst[14:12]`}; otherwise, {`0, inst[14:12]`}.
- For the `lui` instruction, it is set to `1001`, forwarding the B-value.
- For all other instructions, it defaults to `0000`, indicating an addition operation.

Lastly, the `branchTrue` signal,........

4. Verification Strategy

4.1 ALU verification

5. Alternative Design

6. Conclusion

In conclusion, the project has effectively met all the specified requirements stated in the question and conducted thorough testing using a variety of random test cases. An important aspect of the project was the addition of assertions to the testbench file, which facilitated easy error detection in the design. Thorough simulation, the testbench file has been executed without any errors, indicating the robustness and correctness of the design.

Moving forward, the project team plans to focus on further improvements and optimizations. This may include refining the design, enhancing performance, or addressing any identified issues. By leveraging the added assertions and conducting rigorous testing, the team aims to ensure the reliability and correctness of the project implementation.

7. Source Code

7.1 src/coin_converter.sv

```
module coin_converter (
  input              nickel_i, dime_i, quarter_i,
  output reg [2:0] amount_i
);

  always_comb begin
    case ({quarter_i, dime_i, nickel_i})
      3'b001  : amount_i = 3'b001; // A nickel_i
      3'b010  : amount_i = 3'b010; // A dime_i
      3'b100  : amount_i = 3'b101; // A quarter_i
      default : amount_i = 3'b000; // No recognized combination
    endcase
  end

endmodule
```

7.2 src/top.sv

```
module top (
  input         clk_i    , // Clock signal
  input         reset_n_i, // Reset signal (active low)
  input         nickel_i , // Nickel coin input
  input         dime_i   , // Dime coin input
  input         quarter_i, // Quarter coin input
  output        soda_o   , // Output signal for soda dispensing
  output [2:0] change_o    // Output signal for returning change
);

  // Wire declarations
  wire [2:0] amount_w          ; // Amount calculated from inserted coins
  wire       there_is_a_coin_w; // Indicates if there is any coin inserted
  wire [3:0] new_amount_w      ; // Wire for the new amount after
  ↪  processing inserted coins, based on whether there is enough money
  wire       enough_w          ; // Wire to indicate if there is enough
  ↪  money to buy a soda

  // Instantiate the coin_converter module to calculate the amount
  coin_converter cc (
    .nickel_i (nickel_i ),
    .dime_i   (dime_i   ),
    .quarter_i(quarter_i),
    .amount_i (amount_w )
  );

  // Register for storing the current amount of coins
  reg [3:0] coin_storer_r = 4'd0;

  assign there_is_a_coin_w = (nickel_i | dime_i | quarter_i);
  assign enough_w          = (coin_storer_r[3] | coin_storer_r[2]) ? 1'b1
  ↪  : 1'b0;
  assign soda_o            = enough_w;
  assign change_o          = enough_w ? coin_storer_r[2:0] - 3'b100 :
  ↪  3'b000;
  assign new_amount_w      = (enough_w ? 4'd0 : coin_storer_r) + amount_w;

  // Always block for updating coin_storer
  always_ff @(posedge clk_i or negedge reset_n_i) begin
    if (~reset_n_i) begin
      coin_storer_r <= 4'd0;      // Reset the coin_storer on reset
    end else begin
      if (there_is_a_coin_w) begin
        coin_storer_r <= new_amount_w; // Update coin_storer with new amount
      end else if (enough_w) begin
        coin_storer_r <= 4'd0;
      end else begin
        coin_storer_r <= coin_storer_r; // Maintain current coin_storer value
      end
    end
  end
endmodule
```

7.3 tb/tb.cpp

```
#include <verilated.h>
#include <verilated_fst_c.h>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "Vtop.h"

vluint64_t main_time = 0;             // Current simulation time
const vluint64_t sim_duration = 500; // Simulation duration in time units
VerilatedFstC *tfp = nullptr;

int main(int argc, char **argv) {
    Verilated::commandArgs(argc, argv);
    Verilated::traceEverOn(true); // Enable waveform tracing

    // Instantiate the DUT (Design Under Test)
    Vtop *top = new Vtop;

    // Initialize the VCD trace file
    tfp = new VerilatedFstC;
    top->trace(tfp, 99); // Trace all levels
    tfp->open("wave.fst");

    // Initialize simulation inputs
    top->clk_i = 0;
    top->reset_n_i = 1;

    top->nickel_i = 0;
    top->dime_i = 0;
    top->quarter_i = 0;

    // Seed the random number generator
    std::srand(std::time(nullptr));

    int total_money = 0;
    int change_o = 0;

    while (!Verilated::gotFinish() && main_time < sim_duration) {
        top->clk_i = !top->clk_i;                        // Your clock generation logic
        top->reset_n_i = main_time < 20 ? 0 : 1; // Your reset generation logic

        if (main_time % 10 && top->reset_n_i == 1) {
            int rand_num = rand() % 3;
            switch (rand_num) {
            case 0:
                top->nickel_i = 1;
                top->dime_i = 0;
                top->quarter_i = 0;
                total_money += 1;
                break;
            case 1:
                top->nickel_i = 0;
                top->dime_i = 1;
                top->quarter_i = 0;
                total_money += 2;
                break;
            case 2:
                top->nickel_i = 0;
                top->dime_i = 0;
                top->quarter_i = 1;
                total_money += 5;
                break;
            default:
                top->nickel_i = 0;
                top->dime_i = 0;
                top->quarter_i = 0;
                break;
            }
        }

        // Evaluate the DUT
        top->eval();

        // Print simulation time and DUT outputs
        if ((main_time + 5) % 10) {
            printf("Time %4u: Calculated Total Money = %2d - n=%d / d=%d /
             ↪  q=%d, soda_o=%d, change_o=%2d\n", main_time, 5 *
             ↪  total_money, top->nickel_i, top->dime_i, top->quarter_i,
             ↪  top->soda_o, 5 * top->change_o);
            if (total_money >= 4) {
                assert(top->soda_o == 1 && top->change_o == (total_money -
                 ↪  4));
                total_money = 0;
            }
        }
        // Dump waveform data
        tfp->dump(main_time);

        // Advance simulation time
        main_time += 5; // Assuming 1ps/1ps timescale, 10 time units per
         ↪  cycle
    }

    printf("TEST PASSED\n");

    // Close the VCD trace file and clean up
    tfp->close();
    delete top;
    delete tfp;
    return 0;
}
```