The Chinese University of Hong Kong, Shenzhen

CSC3050

Computer Architecture

# Assignment 3 Report

*Author:* Yang Yuzhe
*Student ID:* 121090684

November 26, 2023

# 1 Overview

In this project, we are required to implement an ALU. Firstly, it will input a 32 bit instruction code, then it will divided the instruction code and fetching the data. After that, it will execute the instruction and output the result and flags.

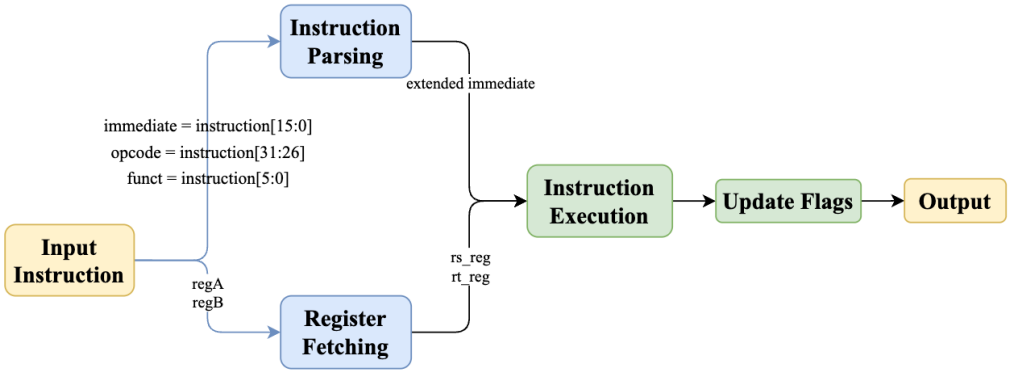


Figure 1: Overview of the project

# 2 Implementation

## 2.1 Input Instruction Code

The instruction code will be divided as following:

```
opcode = instruction[31:26];
funct = instruction[5:0];
rs = instruction[25:21];
rt = instruction[20:16];
rd = instruction[15:11];
shamt = instruction[10:6];
immediate = instruction[15:0];
```

## 2.2 Fetching Data

### Instruction Parsing

The instruction parsing phase involves the extraction of essential fields from the instruction. This includes the following components:

- **opcode:** Instruction code field.
- **funct:** Function code field.
- **rs:** Source register field.
- **rt:** Target register field.
- **rd:** Destination register field.

- **shamt:** Shift amount field.
- **immediate:** Immediate value field.

Furthermore, this phase handles sign extension for specific opcodes such as `addi`, `addiu`, `slti`, `sltiu`, `lw`, and `sw`. Sign extension ensures that immediate values are correctly extended, maintaining the correct sign representation.
If the MSB is 1 (indicating a negative number in two's complement representation), the extended bits will also be 1, effectively keeping the number negative when it's interpreted as a 32-bit number. If the MSB is 0 (indicating a positive number), the extended bits will be 0, keeping the number positive.
Additionally, for opcodes like `andi`, `ori`, and `xori`, the immediate values are adjusted to maintain consistency in the representation. Regardless of whether the original number is positive or negative, the extended number will be positive because the higher order bits are filled with zeros.

```
case(opcode)
    //addi, addiu, slti, sltiu, lw, sw
    6'b001000, 6'b001001, 6'b001010, 6'b001011, 6'b100011, 6'
   b101011: begin
        immediate = {{17{immediate[15]}}, immediate[14:0]};
    end
    //andi, ori, xori
    6'b001100, 6'b001101, 6'b001110: begin
        immediate = {{16{1'b0}}, immediate[15:0]};
    end
endcase
```

### Register Fetching

```
rs_reg = (rs==5'b00000)?regA:regB;
rt_reg = (rt==5'b00000)?regA:regB;
```

This logic determines the source (`rs_reg`) and target (`rt_reg`) registers for the ALU operation based on the values of `rs` and `rt` in the instruction. If either `rs` or `rt` is specified as `$zero` (represented by `5'b00000`), the corresponding register is set to `regA`; otherwise, it is set to `regB`. This approach provides flexibility, allowing the ALU to handle both cases where specific registers are specified and cases where default registers are used.

## 2.3 Instruction Execution

In this part, the instruction execution involves the extraction of essential fields from the instruction and the execution of the ALU operations which is determined based on the opcode and function code.

## 2.4 Update flags

The flags are updated based on the result of the ALU operation. So I write a function to update the flags after all the ALU operations. The function will input the opcode, function code, and the result of the ALU operation and reset the flags

first. The it will detect the overflow, negative or zero condition and update the flags accordingly.
In this project, we only focused on there instruction:

- **Overflow**: `add`, `addu`, `sub`
- **Negative**: `slt`, `slti`, `sltu`, `sltiu`
- **Zero**: `beq`, `bne`

The overflow flag (`overflow_flag`) is addressed for instructions involving addition or subtraction. Overflow conditions are checked by examining the signs of the inputs and the result. If the signs of the inputs are the same, and the sign of the result is different, an overflow has occurred, triggering the setting of the overflow flag.

```
\\addi sub
overflow_flag = ((regA[31] != regB[31]) && (result[31] != regA[31])
    );
\\adddi
overflow_flag = ((regA[31] == immediate[31]) && (result[31] != regA
    [31]))?1:0;
```

Next, the zero flag (`zero_flag`) is determined by conditions specific to branch instructions, such as `beq` and `bne`. It is set when the subtraction of relevant register values, like `rs_reg` and `rt_reg`, results in zero, indicating equality or inequality depending on the opcode.

```
zero_flag = ((rs_reg - rt_reg)==0)?1:0;
```

Then,the negative flag (`negative_flag`) is updated, catering to various instructions such as `slt`, `slti`, `sltiu`, and `sltu`. The signed or unsigned comparison of involved registers or immediate values influences the setting of this flag. The task includes conditions to appropriately set the negative flag based on the comparison results.

```
//slt
if (opcode == 6'b000000 && funct == 6'b101010) begin
    negative_flag = (result==1)?1:0;
    $display("negative flag: %d\n", negative_flag);
end
//slti
else if (opcode == 6'b001010) begin
    if(rs_reg[31]==1&&immediate[31]==0)
        negative_flag = 1;
    else if(rs_reg[31]==0&&immediate[31]==1)
        negative_flag = 0;
    else
        negative_flag = (rs_reg<immediate)?1:0;
    $display("negative flag: %d\n", negative_flag);
end
//sltiu
else if (opcode == 6'b001011) begin
    negative_flag = (result==1)?1:0;
    $display("negative flag: %d\n", negative_flag);
end
//sltu
else if (opcode == 6'b000000 && funct == 6'b101011) begin
    negative_flag = (result==1)?1:0;
    $display("negative flag: %d\n", negative_flag);
end
```

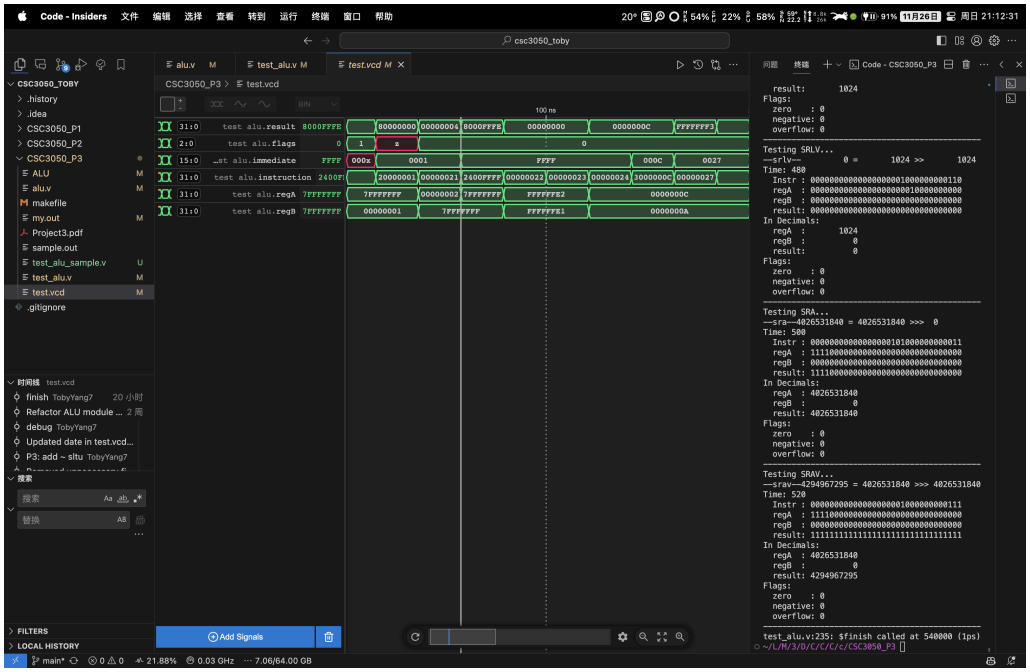


Figure 2: Test result

# 3 Testing & Debugging

In the figure 2, it shows how I test and debug alu program. In the right part of my figure, it shows the output of the `test_alu.v`. Furthermore, in the middel part of the screenshot, the program will generate a vcd file, which can be viewed in Vivado. It counld help me to check all the signals along the time and help me to debug the program.