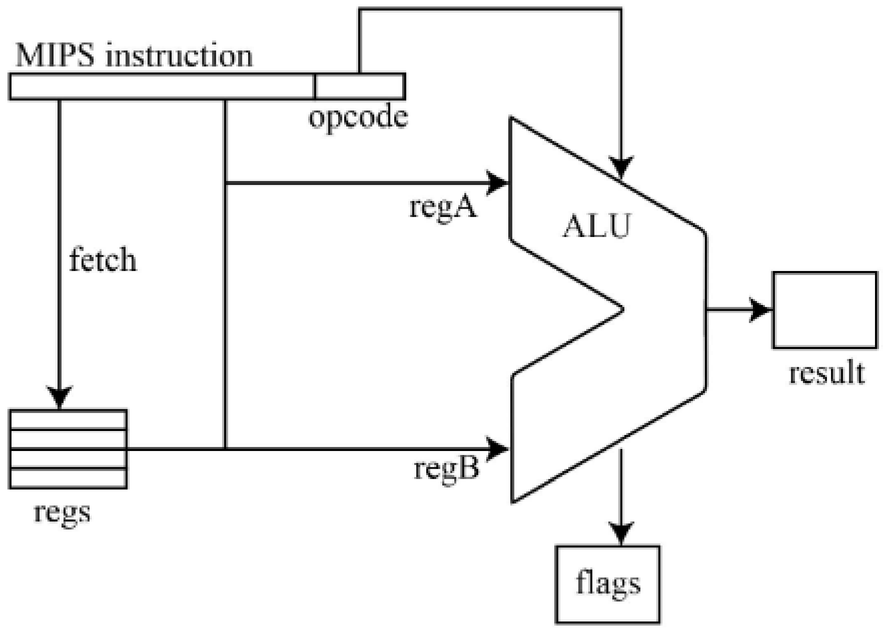
0. Brief Introduction

Since there are two parts in this project, I would also divide my report to two parts: the alu part and the cpu part.

1. ALU

1. Design Idea

The project asked us to implement an ALU using verilog. This ALU required us to run some MIPS codes and output corresponding results. The following graph shows the structure of my ALU,which is based on the knowledge from the tutorial:



2. Implements

Since certain parts of the MIPS code performed certain roles in the instruction， I use the following code to divide instruction to several parts:

```
    opcode=instruction[31:26];
    funct=instruction[5:0];
    flags=3'b000;
    if(instruction[25:21]=={5'b0})
    begin
        rs=regA;
        rt=regB;
    end
    else
    begin
        rs=regB;
        rt=regA;
    end
    shamt=instruction[10:6];
```

Here I assumed that the address of regA and regB can only be 00000 and 00001, so it is easy to find out which register corresponds to its number.

By using `case()` function, I managed to run different computing process:

```
case(opcode)
0'b000000://R type
    case(funct)
    0'b100000://add
    ......//specific implementation
    endcase
......//specific implementation
endcase
```

3. About Slt and Sltu

Function ***slt*** use signed numbers to do the calculation, and ***sltu*** use unsigned number to do the calculation. It happened that "-" and "<" treat numbers in different ways: "-" treat numbers signed and "<" treats number unsigned. So I use:

```
rd=rs-rt;
flags[1]=rd[31];
```

in ***slt***, and:

```
rd=rs-rt;
flags[1]=(rs<rt);
```

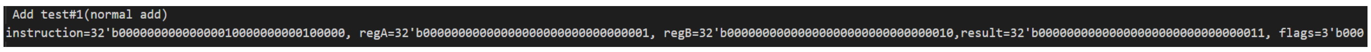
in ***sltu***.

4. Test

I write a lot of testbench ot test my alu, and make sure that I have tested all functions. A testbench looks like this:

```
#100
$display("\n Add test#1(normal add)");
instruction=32'b000000_00000_00001_00000_00000_100000;
regA=32'b0000_0000_0000_0000_0000_0000_0000_0001;
regB=32'b0000_0000_0000_0000_0000_0000_0000_0010;
```

and its corresponding output is:

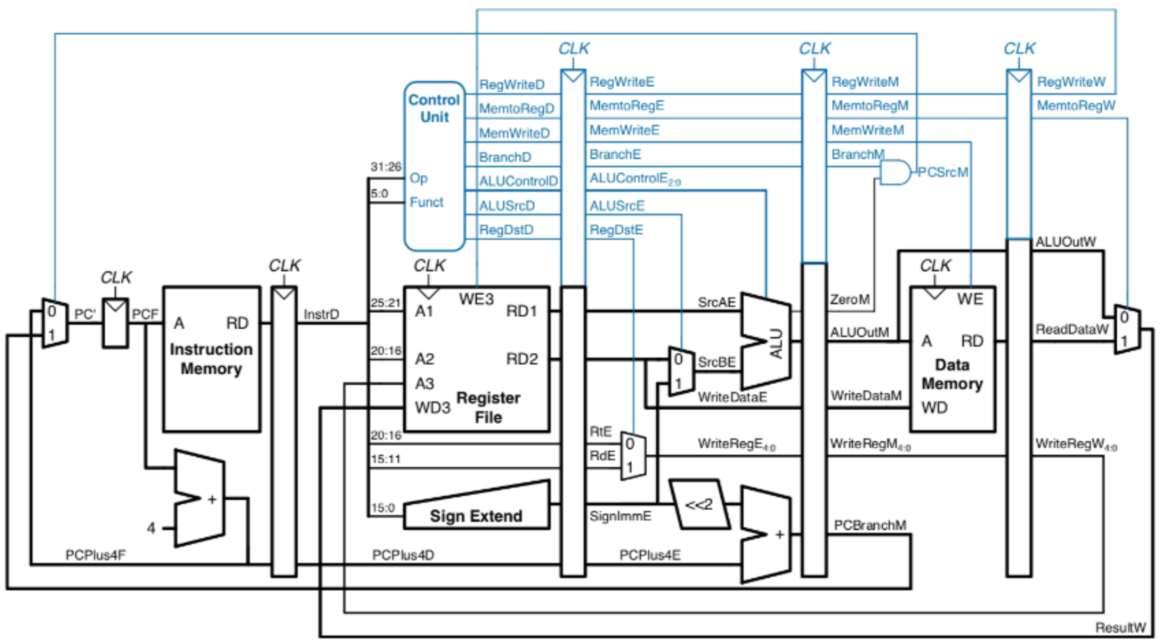


```
 Add test#1(normal add)
instruction=32'b00000000000000010000000000100000, regA=32'b00000000000000000000000000000001, regB=32'b00000000000000000000000000000010,result=32'b00000000000000000000000000000011, flags=3'b000
```

2. CPU

1. Design Idea

A five stage pipeline cpu has the following structure:



So I divided my cpu into these parts: ***ControlUnit.v***, ***RegisterFile.v***, ***alu.v***, ***InstructionRAM.v***, ***MainMemory.v*** and ***cpu.v***.

2. Implements

(1) ControlUnit.v

This part is used to recieve various input signals and output other signals which can influence the process of cpu. For example, it outputs the `alu_control` signal to tell alu which function is in need. Also, this part will also consider when to stall the cpu(details are in the part of hazards).

(2) RegisterFile.v

This part is used to maintain the registers of the cpu. We input the address of the registers (which would be used later) and the compute result from the previous *Write Back* stage, the output are the values of the corresponding register.

(3) Alu.v

This part is to input `alu_control` signal from the *ControlUnit.v* part and the values from *RegisterFile.v* part, the output are the result of caculation and flags.

(4) InstructionRAM.v & MainMemory.v

These two parts were given. I used them to read the instructions from *instructions.bin*, and maintain and output simulated memories.

(5) Cpu.v

This is the core of this cpu. It simulates the five stage, passing variables through each stage. Each stage occupies one clock cycle, and there will be a one-clock-cycle's gap between same variables from adjoint stage(like `reg_write_e` and `reg_write_m`).

3. Hazards

(1) Two Adjoint Instructions Use a Same Register

To discribe the hazard precisly, I would give two instructions as an example:

```
addu $t3,$t2,$t1
add  $t4,$t3,$t2
```

The result of the first instruction *t3* is used in the next instruction, but at this time the value in *t3* is not updated. At this time we would forward the calculation result to the *add* instrution, and update *t3* after this process.

Also, if the load instruction is dealt with during EXE stage, and the register to be used is same with the register which is going to be accessed in ID stage, a bubble will be inserted into ID-EXE pipeline to clear all the control signal, and keep the value of the IF-ID pipeline. The above process is called "stall". The codes are as follows:

```
//in ControlUnit.v
if(rs_address==e_write_reg&&e_reg_write)//e_reg_write is a reg that shows
    begin                               //whether the reg of e_write_reg
        if(e_is_load)                   // is used in the previous stage
            stall=1;
        else
        begin
            e_alu_out_A=1;              //Used to forward values
            d_dr_A=0;                   //rs!=d_dr_A
        end
    end
```

(2) Jump Instructions

I predict if the instruction need to jump everytime. If we do not need to jump, the `pc_reg` will go to `pc_reg+4`, if any jump instruction is met we will do corresponding operation to it, and stall it for one clock cycle:

```
//in ControlUnit.v
case(opcode)
......
6'b000010://j
 begin
    pc_jump=1;
    is_link=1;
end
......
if(is_link)
    d_rst=1; //the reset sign of ID stage
```

In our cpu, we judge whether jump or not in ID stade, while we can get get jump address in EXE stage. So we forward the address in EXE stage to IF stage, to judge the `pc_reg` value. It is similar to "stall" mentioned above.