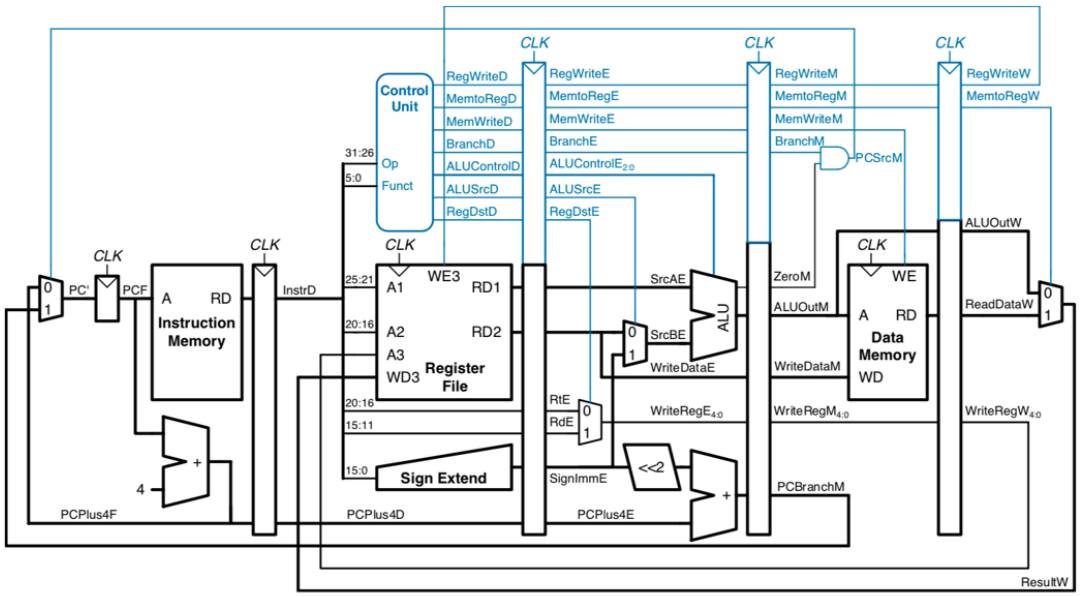
Project Report

李华悦　　118010138

1 understanding of this project:

The purpose of this project is to write a MIPS pipeline processor which can execute MIPS instructions with Verilog. We need to design the pipeline for the CPU, which means we need to design the fetch part, decode part, execute part, memory part and writeback part for the CPU and every instruction will be executed in 5 clock. The testbench will first send the instruction and data into the CPU. Then the CPU will execute the instruction in 5 clock and finally put values in corresponding registers and show the output. And we also need to deal with the hazards problems for the pipeline processor.

2 block diagram:



I have used five always in the CPU module to simulate the pipeline processor.

```
//fetch part
always @(posedge clock)

//decode part
always @(posedge clock)
```

```
//execute part
always @(posedge clock)

//memory part
always @(posedge clock)

//writeback part
always @(posedge clock)
```

To simulate the pipeline processor, I have designed a lot of registers in the CPU module. I will not explain the functions of all the registers one by one because some registers have the same functions and their difference is just that they work in different stages. I use non-blocking assignments to give the values of registers of the last stage to the registers of the next stage at the end of the last stage like this:

```
instrE <= instrD;
regWriteE <= regWriteD;
memToRegE <= memToRegD;
memWriteE <= memWriteD;
branchE <= branchD;
alu_ctrE <= alu_ctrD;
alu_srcE <= alu_srcD;
regDstE <= regDstD;
reg_jumpE <= reg_jumpD;
data_jumpE <= data_jumpD;
overflow_signE <= overflow_signD;
conditionE <= conditionD;
dataE <= dataD;
addressE <= addressD;
```

To deal with the problems of hazards, I put 4 NOP instructions in the testbench after every “real” instruction because NOP instructions change nothing for all the values. And I have used my own solution to design the CPU to solve the problem of hazards of pc. I have put the change of pc from stage1 into stage2.

```
pcBranch = 0;
if ((reg_jumpD==0)&&(conditionD==0)&&(data_jumpD==0))
begin
pc <= pc + 32'h00000004;
end
else if (data_jumpD == 1)
begin
pc <= instrD[25:0];
end
else if (reg_jumpD == 1)
begin
pc <=gr[instrD[25:21]];
end
else if (conditionD == 1)
begin
pcBranch = instrD[15:0];
pcBranch = pcBranch << 2;
pcBranch = {{16{pcBranch[15]}},pcBranch[15:0]};
pc <= pc + pcBranch;
end
```

When you run the program and see the display result from terminal. The pc corresponding to every instruction is always the address of that instruction, which means the pc I have shown is the pcf. Therefore, the problem of hazards of pc has been solved.

3 explanation of all required instructions:

The requirements of this project are just as follows:

The values in corresponding registers can change rightly after an instruction is executed completely.

The value of pc can correspond to the instruction rightly.

The output of the CPU (d_addr and d_dataout) can show the right result for some instructions (lw and sw).

We don't need to implement the instruction memory and data memory, but we need to show the input and output for the two memory. Therefore, I will use my own instructions as an example in the testbench to show the result of all the instructions.

My display in the terminal is as follows:

```
$display("pc    :    instruction          : gr0  : gr1  : gr2  : gr3  : gr31 :dataout : address");
$monitor("%h:%b:%h:%h:%h:%h:%h:%h:%h",
   uut.pc,uut.instr, uut.gr[0], uut.gr[1], uut.gr[2], uut.gr[3],uut.gr[31],d_dataout,d_addr);
```

The pc is always the address that corresponds to the instruction just put in the CPU. The instruction in the display is the instruction that is just put in the CPU. Because the length of the display is limited, I will just use five registers: gr0, gr1,gr2, gr3 and gr31 in my testbench and show the values in them. I will also show the output of the CPU (d_dataout and d_addr). To simulate the whole process of MIPS processor, some instructions in my example may repeat to appear. Because the limited length of the report, every kind of instruction will be explain one time and the repeat of that kind of instruction will just show the input and output and will not be explain again. My examples are as follows:

lw:

Input:

```
//lw
   #period
   d_datain <= 32'h0000_00ab;
   i_datain <= {6'b100011, `gr0, `gr1, 16'h0001};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address

00000000:10001100000000010000000000000001:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
00000004:00000000000000000000000000000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
00000008:00000000000000000000000000000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
0000000c:00000000000000000000000000000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000010:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
```

At the start of the program, address=0, so the pc of the first instruction “lw” is 0 and then for every instruction except branch and jump instructions, the pc will plus 4. Therefore, except branch and jump instructions, I will not explain one by one for the pc. The function of this lw instruction is to store the d_datain in the gr1 register. It is obvious that after 5 clock the value of the gr1 is the d_datain. And the lw also need to show the output. The d_dataout is the same as the d_datain and the d_addr is equal to (0+1=1) 1. It is obvious that after five clock the d_dataout and d_addr show the right output. Therefore, the lw instruction works well in the CPU.

lw:

Input:

```
//lw
   #period d_datain <= 32'h0000_3c00;
   i_datain <= {6'b100011, `gr0, `gr2, 16'h0002};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address

00000014:10001100000000010000000000000010:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000018:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
0000001c:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000020:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:00003c00:00000002
00000024:00000000000000000000000000000000:00000000:000000ab:00003c00:xxxxxxxx:xxxxxxxx:00003c00:00000002
```

add:

Input:

```
//add
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100000};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :       instruction           :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000028:00000000001000100001100000100000:00000000:000000ab:00003c00:xxxxxxxx:xxxxxxxx:00003c00:00000002
0000002c:00000000000000000000000000000000:00000000:000000ab:00003c00:xxxxxxxx:xxxxxxxx:00003c00:00000002
00000030:00000000000000000000000000000000:00000000:000000ab:00003c00:xxxxxxxx:xxxxxxxx:00003c00:00000002
00000034:00000000000000000000000000000000:00000000:000000ab:00003c00:xxxxxxxx:xxxxxxxx:00003c00:00000002
00000038:00000000000000000000000000000000:00000000:000000ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
```

The function of this add instruction is to add the values in gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 3cab (ab + 3c00 = 3cab). It is obvious that the result is right. Therefore, the add instruction works well in the CPU.

addu:

Input:

```
//addu
  #period i_datain <= {6'b000000, `gr2, `gr3, `gr1,  5'b00000, 6'b100001};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :       instruction           :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

0000003c:00000000010000110000100000100001:00000000:000000ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000040:00000000000000000000000000000000:00000000:000000ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000044:00000000000000000000000000000000:00000000:000000ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000048:00000000000000000000000000000000:00000000:000000ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
0000004c:00000000000000000000000000000000:00000000:000078ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
```

The function of this addu instruction is to add the values in gr2 and gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes 78ab (3c00+3cab=78ab). It is obvious that the result is right. Therefore, the addu instruction works well in the CPU.

addi:

Input:

```
//addi
  #period i_datain <= {6'b001000,`gr3, `gr1,  16'b0000000000001000};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :       instruction           :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000050:00100000110000100000000000001000:00000000:000078ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000054:00000000000000000000000000000000:00000000:000078ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000058:00000000000000000000000000000000:00000000:000078ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
0000005c:00000000000000000000000000000000:00000000:000078ab:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000060:00000000000000000000000000000000:00000000:00003cb3:00003c00:00003cab:xxxxxxxx:00003c00:00000002
```

The function of this addi instruction is to add the values in gr3 and imm part of instruction and put the result in gr1. After 5 clock, the value in gr1 becomes 3cb3

(3cab+8=3cb3). It is obvious that the result is right. Therefore, the addi instruction works well in the CPU.

addiu:

Input:

```
//addiu
   #period i_datain <= {6'b001001,`gr2, `gr1,  16'b0000000000001001};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000064:00100100010000010000000000001001:00000000:00003cb3:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000068:00000000000000000000000000000000:00000000:00003cb3:00003c00:00003cab:xxxxxxxx:00003c00:00000002
0000006c:00000000000000000000000000000000:00000000:00003cb3:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000070:00000000000000000000000000000000:00000000:00003cb3:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000074:00000000000000000000000000000000:00000000:00003c09:00003c00:00003cab:xxxxxxxx:00003c00:00000002
```

The function of this addiu instruction is to add the values in gr2 and imm part of instruction and put the result in gr1. After 5 clock, the value in gr1 becomes 3c09 (3c00+9=3c09). It is obvious that the result is right. Therefore, the addiu instruction works well in the CPU.

sub:

Input:

```
//sub
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100010};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000078:00000000001000100001100000100010:00000000:00003c09:00003c00:00003cab:xxxxxxxx:00003c00:00000002
0000007c:00000000000000000000000000000000:00000000:00003c09:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000080:00000000000000000000000000000000:00000000:00003c09:00003c00:00003cab:xxxxxxxx:00003c00:00000002

00000084:00000000000000000000000000000000:00000000:00003c09:00003c00:00003cab:xxxxxxxx:00003c00:00000002
00000088:00000000000000000000000000000000:00000000:00003c09:00003c00:00000009:xxxxxxxx:00003c00:00000002
```

The function of this sub instruction is to do subtraction with the values in gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 9 (3c09-3c00=9). It is obvious that the result is right. Therefore, the sub instruction works well in the CPU.

subu:

Input:

```
//subu
   #period i_datain <= {6'b000000, `gr2, `gr3, `gr1,  5'b00000, 6'b100011};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address
0000008c:00000000010000110000100000100011:00000000:00003c09:00003c00:00000009:xxxxxxxx:00003c00:00000002
00000090:00000000000000000000000000000000:00000000:00003c09:00003c00:00000009:xxxxxxxx:00003c00:00000002
00000094:00000000000000000000000000000000:00000000:00003c09:00003c00:00000009:xxxxxxxx:00003c00:00000002
00000098:00000000000000000000000000000000:00000000:00003c09:00003c00:00000009:xxxxxxxx:00003c00:00000002
0000009c:00000000000000000000000000000000:00000000:00003bf7:00003c00:00000009:xxxxxxxx:00003c00:00000002
```

The function of this subu instruction is to do subtraction with the values in gr2 and gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes 3bf7 (3c00-9=3bf7). It is obvious that the result is right. Therefore, the subu instruction works well in the CPU.

and:

Input:

```
//and
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100100};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address
000000a0:00000000001000100001100000100100:00000000:00003bf7:00003c00:00000009:xxxxxxxx:00003c00:00000002
000000a4:00000000000000000000000000000000:00000000:00003bf7:00003c00:00000009:xxxxxxxx:00003c00:00000002
000000a8:00000000000000000000000000000000:00000000:00003bf7:00003c00:00000009:xxxxxxxx:00003c00:00000002
000000ac:00000000000000000000000000000000:00000000:00003bf7:00003c00:00000009:xxxxxxxx:00003c00:00000002
000000b0:00000000000000000000000000000000:00000000:00003bf7:00003c00:00003800:xxxxxxxx:00003c00:00000002
```

The function of this and instruction is to make “and” logic for every bit of the values in gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 3800 (3bf7 & 3c00=3800). It is obvious that the result is right. Therefore, the and instruction works well in the CPU.

andi:

Input:

```
//andi
   #period i_datain <= {6'b001100,`gr2, `gr1,  16'b1110000000001111};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
000000b4:00110000010000011110000000001111:00000000:00003bf7:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000b8:00000000000000000000000000000000:00000000:00003bf7:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000bc:00000000000000000000000000000000:00000000:00003bf7:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000c0:00000000000000000000000000000000:00000000:00003bf7:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000c4:00000000000000000000000000000000:00000000:00002000:00003c00:00003800:xxxxxxxx:00003c00:00000002
```

The function of this andi instruction is to make “and” logic for every bit of the values in the imm part of the instruction and gr2 and put the result in gr1. After 5 clock, the value in gr1 becomes 2000 (3c00 & f=2000). It is obvious that the result is right. Therefore, the andi instruction works well in the CPU.

or:

Input:

```
//or
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100101};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
000000c8:00000000001000100001100000100101:00000000:00002000:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000cc:00000000000000000000000000000000:00000000:00002000:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000d0:00000000000000000000000000000000:00000000:00002000:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000d4:00000000000000000000000000000000:00000000:00002000:00003c00:00003800:xxxxxxxx:00003c00:00000002
000000d8:00000000000000000000000000000000:00000000:00002000:00003c00:00003c00:xxxxxxxx:00003c00:00000002
```

The function of this or instruction is to make “or” logic for every bit of the values in the gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 3c00 (2000 or 3c00=3c00). It is obvious that the result is right. Therefore, the or instruction works well in the CPU.

ori:

Input:

```
//ori
   #period i_datain <= {6'b001101,`gr2, `gr1,  16'b1110000000011111};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
000000dc:00110100010000011110000000011111:00000000:00002000:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000e0:00000000000000000000000000000000:00000000:00002000:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000e4:00000000000000000000000000000000:00000000:00002000:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000e8:00000000000000000000000000000000:00000000:00002000:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000ec:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00003c00:xxxxxxxx:00003c00:00000002
```

The function of this ori instruction is to make “or” logic for every bit of the values in the gr2 and the imm part of the instruction and put the result in gr1. After 5 clock,

the value in gr1 becomes fc1f (3c00 or e01f=fc1f). It is obvious that the result is right. Therefore, the ori instruction works well in the CPU.

nor:
Input:

```
//nor
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100111};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address
000000f0:00000000001000100001100000100111:00000000:0000fc1f:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000f4:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000f8:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00003c00:xxxxxxxx:00003c00:00000002
000000fc:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00003c00:xxxxxxxx:00003c00:00000002
00000100:00000000000000000000000000000000:00000000:0000fc1f:00003c00:ffff03e0:xxxxxxxx:00003c00:00000002
```

The function of this nor instruction is to make “nor” logic for every bit of the values in the gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes ffff03e0 (fc1f nor 3c00=ffff03e0). It is obvious that the result is right. Therefore, the nor instruction works well in the CPU.

xor:
Input:

```
//xor
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100110};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address
00000104:00000000001000100001100000100110:00000000:0000fc1f:00003c00:ffff03e0:xxxxxxxx:00003c00:00000002
00000108:00000000000000000000000000000000:00000000:0000fc1f:00003c00:ffff03e0:xxxxxxxx:00003c00:00000002
0000010c:00000000000000000000000000000000:00000000:0000fc1f:00003c00:ffff03e0:xxxxxxxx:00003c00:00000002
00000110:00000000000000000000000000000000:00000000:0000fc1f:00003c00:ffff03e0:xxxxxxxx:00003c00:00000002
00000114:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000c01f:xxxxxxxx:00003c00:00000002
```

The function of this xor instruction is to make “xor” logic for every bit of the values in the gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes c01f (fc1f xor 3c00=c01f). It is obvious that the result is right. Therefore, the xor instruction works well in the CPU.

sll:
Input:

```
//sll
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00010, 6'b000000};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             : gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
00000118:00000000001000100001100010000000:00000000:0000fc1f:00003c00:0000c01f:xxxxxxxx:00003c00:00000002
0000011c:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000c01f:xxxxxxxx:00003c00:00000002
00000120:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000c01f:xxxxxxxx:00003c00:00000002
00000124:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000c01f:xxxxxxxx:00003c00:00000002
00000128:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000f000:xxxxxxxx:00003c00:00000002
```

The function of this sll instruction is to logic left shift 2 bit of the value in gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes f000 (3c00<<2=f000). It is obvious that the result is right. Therefore, the sll instruction works well in the CPU.

sra:

Input:

```
//sra
   #period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00010, 6'b000011};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             : gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
0000012c:00000000001000100001100010000011:00000000:0000fc1f:00003c00:0000f000:xxxxxxxx:00003c00:00000002
00000130:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000f000:xxxxxxxx:00003c00:00000002
00000134:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000f000:xxxxxxxx:00003c00:00000002
00000138:00000000000000000000000000000000:00000000:0000fc1f:00003c00:0000f000:xxxxxxxx:00003c00:00000002
0000013c:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00000f00:xxxxxxxx:00003c00:00000002
```

The function of this sra instruction is to arithmetic right shift 3 bit of the value in gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes f00 (3c00>>3=f00). It is obvious that the result is right. Therefore, the sra instruction works well in the CPU.

srl:

Input:

```
//srl
   #period i_datain <= {6'b000000, `gr2, `gr1, `gr3,  5'b00011, 6'b000010};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000140:00000000010000010001100011000010:00000000:0000fc1f:00003c00:00000f00:xxxxxxxx:00003c00:00000002

00000144:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00000f00:xxxxxxxx:00003c00:00000002
00000148:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00000f00:xxxxxxxx:00003c00:00000002
0000014c:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00000f00:xxxxxxxx:00003c00:00000002
00000150:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00001f83:xxxxxxxx:00003c00:00000002
```

The function of this srl instruction is to logic right shift 2 bit of the value in gr1 and put the result in gr3. After 5 clock, the value in gr3 becomes 1f83 (fc1f>>2=1f83). It is obvious that the result is right. Therefore, the srl instruction works well in the CPU.

lw:

Input:

```
//lw
   #period d_datain <= 32'b00000000000000000000000000000011;
   i_datain <= {6'b100011, `gr0, `gr2, 16'h0002};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000154:10001100000000010000000000000010:00000000:0000fc1f:00003c00:00001f83:xxxxxxxx:00003c00:00000002
00000158:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00001f83:xxxxxxxx:00003c00:00000002
0000015c:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00001f83:xxxxxxxx:00003c00:00000002
00000160:00000000000000000000000000000000:00000000:0000fc1f:00003c00:00001f83:xxxxxxxx:00000003:00000002
00000164:00000000000000000000000000000000:00000000:0000fc1f:00000003:00001f83:xxxxxxxx:00000003:00000002
```

sllv:

Input:

```
//sllv
   #period i_datain <= {6'b000000, `gr2, `gr3, `gr1,  5'b00000, 6'b000100};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00000168:00000000010000110000100000000100:00000000:0000fc1f:00000003:00001f83:xxxxxxxx:00000003:00000002
0000016c:00000000000000000000000000000000:00000000:0000fc1f:00000003:00001f83:xxxxxxxx:00000003:00000002
00000170:00000000000000000000000000000000:00000000:0000fc1f:00000003:00001f83:xxxxxxxx:00000003:00000002
00000174:00000000000000000000000000000000:00000000:0000fc1f:00000003:00001f83:xxxxxxxx:00000003:00000002
00000178:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
```

The function of this sllv instruction is to logic left shif value in gr2 of the value in gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes fc18 (1f83<<3==fc18). It is obvious that the result is right. Therefore, the sllv instruction works well in the CPU.

srav:

Input:

```
//srav
   #period i_datain <= {6'b000000, `gr2, `gr1, `gr3,  5'b00000, 6'b000111};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
0000017c:00000000010000100011100000000111:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
00000180:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
00000184:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
00000188:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
0000018c:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
```

The function of this srav instruction is to arithmetic right shift value in gr2 of the value in gr1 and put the result in gr3. After 5 clock, the value in gr3 becomes 1f83 (fc18>>3==1f83). It is obvious that the result is right. Therefore, the srav instruction works well in the CPU.

srlv:

Input:

```
//srlv
   #period i_datain <= {6'b000000, `gr2, `gr3, `gr1,  5'b00000, 6'b000110};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            : gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
00000190:00000000010000110000100000000110:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
00000194:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
00000198:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
0000019c:00000000000000000000000000000000:00000000:0000fc18:00000003:00001f83:xxxxxxxx:00000003:00000002
000001a0:00000000000000000000000000000000:00000000:000003f0:00000003:00001f83:xxxxxxxx:00000003:00000002
```

The function of this srlv instruction is to logic right shif value in gr2 of the value in gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes 3f0 (1f83>>3==3f0). It is obvious that the result is right. Therefore, the srlv instruction works well in the CPU.

slt:

Input:

```
//slt
   #period i_datain <= {6'b000000, `gr3, `gr2, `gr1,  5'b00000, 6'b101010};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
000001a4:00000000011000100000100000101010:00000000:000003f0:00000003:00001f83:xxxxxxxx:00000003:00000002
000001a8:00000000000000000000000000000000:00000000:000003f0:00000003:00001f83:xxxxxxxx:00000003:00000002
000001ac:00000000000000000000000000000000:00000000:000003f0:00000003:00001f83:xxxxxxxx:00000003:00000002
000001b0:00000000000000000000000000000000:00000000:000003f0:00000003:00001f83:xxxxxxxx:00000003:00000002
000001b4:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
```

The function of this slt instruction is to judge whether the value in gr3 is less than the value in gr2 or not. If it is yes, the value in gr1 will change to 1. If it is no, the value in gr1 will change to 0. In this case, the value in gr3 is not less than the value in gr2 (1f83>3). After 5 clock, the value in gr1 changes to 0. It is obvious that the result is right. Therefore, the slt instruction works well in the CPU.

jr:

Input:

```
//jr
  #period i_datain <= {6'b000000, `gr2,21'b000000000000000001000};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
000001b8:00000000010000000000000000001000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
00000003:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
00000007:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
0000000b:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
0000000f:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
```

The function of this jr instruction is to change the pc to the address stored in gr2. The pc of the first NOP instruction becomes 3 which is exactly the value in gr2. It is obvious that the result is right. Therefore, the jr instruction works well in the CPU.

j:

Input:

```
//j
  #period i_datain <= {6'b000010, 26'b00000000000011111111111111};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
00000013:00001000000000000011111111111111:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
00003fff:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
00004003:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
00004007:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
0000400b:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
```

The function of this j instruction is to change the pc to the target part of the instruction. The pc of the first NOP instruction becomes 3fff which is exactly the value

of the target. It is obvious that the result is right. Therefore, the j instruction works well in the CPU.

jal:

Input:

```
//jal
   #period i_datain <= {6'b000011, 26'b00000000000011111111110000};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address
0000400f:00001100000000000011111111110000:00000000:00000000:00000003:00001f83:xxxxxxxx:00000003:00000002
00003ff0:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00003ff4:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00003ff8:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00003ffc:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
```

The function of this jal instruction is to change the pc to the target part of the instruction and store the original next address in the gr31 (ra) register. The pc of the first NOP instruction becomes 3ff0 which is exactly the value of the target and after 5 clock, the value in gr31 becomes 4013 which is exactly the original next address (400f+4=4013). It is obvious that the result is right. Therefore, the jal instruction works well in the CPU.

beq:

Input:

```
//beq
   #period i_datain <= {6'b000100,`gr0,`gr0,16'b0000000000001111};
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
   #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             : gr0   : gr1   : gr2   : gr3   :   gr31 :dataout : address
00004000:00010000000000000000000000001111:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004040:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004044:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004048:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
0000404c:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
```

The function of this beq instruction is to judge whether the values in gr0 and gr0 are equal or not. If they are equal, the pc will change to $pc + offset \times 4 + 4$. In this case, the values in gr0 and gr0 are equal. The pc of the first NOP instruction becomes 4040 $(4000 + f \times 4 + 4 = 4040)$. It is obvious that the result is right. Therefore, the beq instruction works well in the CPU.

bne:

Input:

```
//bne
  #period i_datain <= {6'b000101,`gr0,`gr0,16'b0000000000001111};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00004050:00010100000000000000000000001111:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004054:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004058:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
0000405c:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004060:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
```

The function of this bne instruction is to judge whether the values in gr0 and gr0 are equal or not. If they are not equal, the pc will change to $pc + offset \times 4 + 4$. In this case, the values in gr0 and gr0 are equal. Nothing will happen. The pc of the first NOP instruction becomes 4054 (4050+4=4054). It is obvious that the result is right. Therefore, the bne instruction works well in the CPU.

add:

Input:

```
//add
  #period i_datain <= {6'b000000, `gr3, `gr2, `gr1,  5'b00000, 6'b100000};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00004064:00000000011000100000100000100000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004068:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
0000406c:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004070:00000000000000000000000000000000:00000000:00000000:00000003:00001f83:00004013:00000003:00000002
00004074:00000000000000000000000000000000:00000000:00001f86:00000003:00001f83:00004013:00000003:00000002
```

sw:

Input:

```
//sw
  #period i_datain <= {6'b101011,`gr0,`gr1,16'b0000000000001111};
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
  #period i_datain <= 32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

Output:

```
pc      :       instruction             :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address

00004078:10101100000000010000000000001111:00000000:00001f86:00000003:00001f83:00004013:00000003:00000002
0000407c:00000000000000000000000000000000:00000000:00001f86:00000003:00001f83:00004013:00000003:00000002
00004080:00000000000000000000000000000000:00000000:00001f86:00000003:00001f83:00004013:00000003:00000002
00004084:00000000000000000000000000000000:00000000:00001f86:00000003:00001f83:00004013:00001f86:0000000f
00004088:00000000000000000000000000000000:00000000:00001f86:00000003:00001f83:00004013:00001f86:0000000f
```

The function of this sw instruction is to get the value stored in gr1 and get the address from gr0 + offset. After 5 clock, the d_dataout is 1f86 which is equal to the

value in gr1 and the d_addr is f which is equal to gr0 + offset (0+f=f). It is obvious that the result is right. Therefore, the sw instruction works well in the CPU.

4 explanation of the change of pc:

In my program, my pc is the pcf, which means the pc is exact the address of the corresponding instruction but not the next instruction. For most of instructions required in this project, when the next instruction comes, we just need to make the pc plus 4. For jump instruction, we need to get the jump address and make the pc of the next instruction of the jump instruction be the jump address. For the branch instruction beq and bne, if they meet their branch condition, we just calculate the branch address (pc + offset*4+4) and then make the pc of the next instruction of the branch instruction be the branch address.

5 explanation of the output (d_addr and d_dataout):

The output is only useful for lw and sw. For lw, the d_dataout is the same as d_datain and the d_addr is equal to the value in rs plus offset. For sw, the d_dataout is the same as the value in rt and the d_addr is equal to the value in rs plus offset. For both lw and sw, the right d_addr and d_dataout will be shown after 5 clock (when the instruction executed completely). For other instructions, the output is not very important because it will not be used.

6 how to run my code (in windows):

```
E:\term5\CSC3050\project4\12>iverilog -o dsn CPU.v test_CPU.v

E:\term5\CSC3050\project4\12>vvp dsn
pc       :        instruction            :  gr0   :  gr1   :  gr2   :  gr3   :   gr31 :dataout : address
fffffff8:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
fffffffc:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
00000000:10001100000000010000000000000001:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
00000004:00000000000000000000000000000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
00000008:00000000000000000000000000000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx
0000000c:00000000000000000000000000000000:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000010:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000014:10001100000000010000000000000010:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000018:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
0000001c:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000001
00000020:00000000000000000000000000000000:00000000:000000ab:xxxxxxxx:xxxxxxxx:xxxxxxxx:00003c00:00000002
00000024:00000000000000000000000000000000:00000000:000000ab:00003c00:xxxxxxxx:xxxxxxxx:00003c00:00000002
```

7 summary:

I spent almost a whole week to write this project. I really try my best to finish this project. I have learnt a lot about Verilog and CPU from this project. With the help of TA and many students, I have overcome a lot of problems. I am grateful to them.