Project 3 Report

0. Overview

In this project, I will implement a simple CPU that supports:

1. Simple instruction parsing,
2. Register value fetching,
3. ALU functions.

In this report, I will mainly cover parts including:

1. Overview,
2. Implementation & details,
3. Test (ALU test bench).

1. Implementation

My ALU format will be as below:

```
// ALU format
module alu(instruction, gr0, gr1, result, flags);
// I/O
input  signed[31:0] instruction, gr0, gr1; // address of gr0 is 00000, address of gr1 is 00001
output signed[31:0] result;
output [2:0] flags; // [ZF,SF,OF]
// ALU local registers declaration
reg signed [31:0] Rset[0:1] // Rset = [gr0, gr1]; regesters in ALU
reg ZF, SF, OF;
reg signed [31:0] reg_A, reg_B, reg_C;     // reg_A, reg_B will be input of ALU, reg_C will be output of ALU

// step 1.  Parsing the Instruction
// obtain opcode, function, immediate value, shamt, &
add_rs = instruction[20:16];
add_rt = instruction[25:21];
// step 2.  Fetch values in mem
reg_rs = Rset[add_rs];
// step 3.  Implementation functionalities
// add, and, beq, lw, sll
// step 4.  output values and flags
assign result = reg_C[31:0]
assign flags = { ZF, SF, OF}
endmodule
```

name	opcode	function	ALU input: reg_A	ALU input: reg_B	ALU_function	flags	remark
sll	000000	000000	R[rt]	sa	reg_C = reg_A << reg_B		
srl		000010		sa	...>>...		
sra		000011		sa	...>>>...		1
sllv		000100	R[rt]	R[rs] [4:0]	... << ...		
srlv		000110			... >> ...		
srav		000111			... >>> ...		1
add	000000	100000	R[rs]	R[rt]	reg_C = reg_A + reg_B	OF	2
addu		100001			... + ...		
addi	001000			{ { (16){ imm[15] } },imm };	... + ...	OF	2
addiu	001001				... + ...		
lw	100011				... + ...		
sw	101011				... + ...		

sub		100010	R[rs]	R[rt]	... - ...	OF	2
subu		100011			... - ...		
beq	000100				ZF = (reg_rs == reg_rt)	ZF	
bne	000101				...		
and	000000	100100	R[rs]	R[rt]	reg_C = reg_A & reg_B		
or		100101			... | ...		
xor		100110			... ^ ...		
nor		100111			~(... | ...)		
andi	001100			{ { (16){1'b0} } ,imm}	...&...		
ori	001101				...|...		
xori	001110				....^....		
slt	000000	101010	signed R[rs]	signed R[rt]	SF = (reg_A - reg_B)[31]	SF	3
sltu		101011	unsigned R[rs]	unsigned R[rt]	SF = (reg_A < reg_B)	SF	3
slti	001010			{ { (16){ imm[15] } },imm };	SF = ...		
sltiu	001011				SF = (reg_A < reg_B)		

Remark:

(1). >>> : arithmetic extention

>>: logic extention

Eg. a = 4’b 1001;

a >> 2 : 4’b 0010;

a >>> 2 : 4’b 1110;

(2). overflow checking

Note that overflow occurs if and only if when

(a) + plus + = -

(b) - plus - = +

or

(c) - minus + = +

(d) + minus - = -

Now let **a, b** be the sign of register A and B respectively, **c** be the sign of register C, where register C = sum of register A and register B, then

For add/addi: OF = (~(a xor b)) & (a & c)

For sub: OF = (a xor b) & (a xor c)

(3). slt: signed minus;

sltu: unsigned minus;

For unsigned minus: let (unsigned) x, y = (signed) rs, rt respectively, then applying minus.

2. Test — ALU test bench

Sample Testing examples:

1. instruction <=32'b0000_0000_0000_0001_0001_0001_0000_0111; srav rd rt sa

gr1 <=32'b1100_0000_0100_0000_0100_0000_0100_0000;
gr0 <=32'b0000_0000_0000_0000_0000_0000_0000_0100;
exp_result <=32'b1111_1100_0000_0100_0000_0100_0000_0100;

Testing Result:

fc040404

2. instruction <=32'b0000_0000_0000_0001_0001_0001_0010_0010;
   gr0 <=32'b0111_1111_1111_1111_1111_1111_1111_1111;
   gr1 <=32'b1000_0000_0000_0000_0000_0000_0000_0001;
   exp_result <=32'b1111_1111_1111_1111_1111_1111_1111_1110;

   negative **overflow**

sub rd rs rt

Testing Result & flags

fffffffe; 00**1**

3. instruction <=32'b0001_0000_0000_0001_1111_1111_1111_1111;
   gr0 <=32'b0111_1111_1111_1111_1111_1111_1111_1011;
   gr1 <=32'b0111_1111_1111_1111_1111_1111_1111_1011;
   exp_result <=0; **zero flag = 1**;

beq rs rt label

Testing Results & flags

00000; **1**00

4. instruction <=32'b0010_1000_0000_0001_0000_0000_0000_0001;
   gr0 <=32'b1111_1111_1111_1111_1111_1111_1111_1111;
   exp_result <=0; **negative flag = 1;**

slti rt rs (imm)

testing results and flags :

000000;0**1**0