MIPS ALU Simulator

Project Overview

This project is a simple MIPS ALU simulator which supports following instructions:

```
- add, addi, addu, addiu
- sub, subu
- and, andi, nor, or, ori, xor, xori
- beq, bne, slt, slti, sltiu, sltu
- lw, sw
- sll, sllv, srl, srlv, sra, srav
```

Actually, the true ALU in MIPS architecture only achieves: `and`, `or`, `add`, `subtract`, `set on less that`, `nor` by receiving specific opcodes and contents from two registers (or extended numbers, etc.) and outputing a 32-bit result and a 3-bit flag. And thus, some functions of the above instructions cannot be achived by the only ALU (e.g., `lw`).

Projects Details

This ALU is defined by its behaviors (not by its structure) in Verilog. For the specific file organization and project building instructions, they can be refered in the README file. Basically, to run the project only type

```
> make
```

in the terminal.

How the simulation works

There are three steps for the simulaton to work: parsing the binary instruction, reading the input data, and outputing the results. The details of the three steps are elaborated as follows:

- Parsing:

  ```
  always @(ins, regA, regB) begin
      opcode = ins[31:26];
      funct  = ins[5:0];
      shamt  = ins[10:6];
      ...
      case (opcode)
          ...
      endcase
  end
  ```

- Reading data: the instance of `mips_alu` module reads the instruction, regA, and regB data from three test files: `INS_DATA`, `REGA_DATA`, and `REGB_DATA`, respectively.
- Output: the result of the specific instruction will be assigned to the `result` and `flag`

Test bench

There are 135 instructions are tested in the test bench. There are a few notes in this simulation:

1. Zero detection is simultaneous performed;
2. For `beq` and `bne`, the negative flag will be set to 1 if the result is to branch;
3. For `slt` and related instructions, the negative flag will be set to 1 if rs < rt;
4. For `lw` and `sw`, since there are no memory simulation in this project, there are the two instructions only add the numbers in rs and the offsets.

Below are the part of the test results (some of the result may not be resonable, e.g., the inputs of `lw` or `sw`):

Arithmetic & logic instructions

INS(b)	REGA(h)	REGB(h)	RESULT(h)	FLAG(b)	INS NAME
00000000000000010000000000100000	00000001	00000002	00000003	000	ADD
00000000001000000000000000100000	7fffffff	00000001	80000000	001	ADD
00000000001000000000000000100000	80000000	80000000	00000000	101	ADD
00000000000000010000000000100001	ffffffff	00000001	00000000	100	ADDU
00100000000000011111111111111111	80000000	ffffffff (imm)	7fffffff	001	ADDI
00100100000000011111111111111111	80000000	ffffffff (imm)	7fffffff	000	ADDIU
00000000000000010000000000100100	00000001	ffffffff	00000001	000	AND
00110000000000011111111111111111	ffffffff	0000ffff (imm)	0000ffff	000	ANDI
00000000001000000000000000100110	80000000	ffffffff	7fffffff	000	XOR
00000000000000010000000011000000	xxxxxxxx	7fffffff	ffffffc0	000	SLL
00000000000000010000000011000010	xxxxxxxx	ffffffff	1fffffff	000	SRL
00000000001000000000000000000110	7fffffff	ffffffff	00000000	100	SRLV
00000000000000010000011111000011	xxxxxxxx	80000000	ffffffff	000	SRA
00000000001000000000000000000111	7fffffff	ffffffff	00000000	100	SRAV
00000000001000000000000000100010	80000000	ffffffff	7fffffff	001	SUB
00000000001000000000000000100011	7fffffff	ffffffff	80000000	000	SUBU

Note that it is required for the immediate numbers of `andi`, `ori`, and `xori` to be zero-extended.

Branch & slt instructions

INS(b)	REGA(h)	REGB(h)	RESULT(h)	FLAG(b)	INS NAME
0001000000100000xxxxxxxxxxxxxxxx	8000ffff	80000000	xxxxxxxx	000	BEQ
0001010000000001xxxxxxxxxxxxxxxx	00000001	00000000	xxxxxxxx	100	BNE
00000000000000010000000000101010	00000001	ffffffff	00000000	100	SLT
00101000000000011111111111111111	80000000	ffffffff (imm)	00000001	010	SLTI

Load & store instructions

INS(b)	REGA(h)	REGB(h)	RESULT(h)	FLAG(b)	INS NAME
10001100000000011111111111111000	0000f001	00000009	0000f00a	000	LW
10101100000000010000000000001001	0000f001	00000009	0000f00a	000	SW