b10902052 lab1 report

Modules Explaination

Adder.v
read an 32-bit input `pc_i` and output a 32-bit value `pc_o`, with relation `pc_o=pc_i+4`

ALU_Control.v
read 3 arguments, `ALUop_i`, `func3`, `func7`, `ALUfunc_o`.
`ALUop_i` is given by `Control` module, `func3` and `func7` is given by the `instruction`.
Output a `ALUfunc_o` with 3-bit, to specify what ALU needs to do.

I done this by observe the following Truth Table.

fn	ALUop_i[1]	func7[5]	func7[0]	func3[2]	func3[0]	ALUfunc
add	1	0	0	0	0	000
sll	1	0	0	0	1	001
xor	1	0	0	1	0	010
and	1	0	0	1	1	011
mul	1	0	1	0	0	100
sub	1	1	0	0	0	101
addi	0	x	x	0	0	110
sari	0	1	0	1	1	111

Obviously,

If `ALUop_i[1] == 1` means it is R-Type and exclude the `sub` option,
The `ALUfunc` is equal to `func7[0]+func3[2]+func3[0]`
And if we done this, the `sub` option is equal to `000` therefor we can just assign it by bitwise or operation `|` to complete the R-Type part.

In `ALUop_i[1] == 0`, the I-Type part, we can assign `ALUfunc[2:1]` to be `1`, and the last bit to be `func3[2]` or `func3[0]`, here I choose `func3[2]`.

In conclusion, we can get the following three formula.

```
ALUfunc_o[0] =
    ( ALUop_i[1] & func3[0] ) | ( ALUop_i[1] & func7[5] ) |
    ( (!ALUop_i[1]) &func3[2] );
ALUfunc_o[1] =
    ( ALUop_i[1] & func3[2] ) | (!ALUop_i[1]);
ALUfunc_o[2] =
    ( ALUop_i[1] & func7[0] ) | ( ALUop_i[1] & func7[5] ) |
    (!ALUop_i[1]);
```

Then we can get a useable `ALUfunc_o` as an usable output.

MUX32.v

With two 32-bit input, one select bit, and a 32-bit output.
Here I use the `?` operator to complete this.
If select bit is false, output first input, else output the second.

ALU.v

With two 32-bit input, one 3-bit ALUfunc, output an 32-bit result.
Almost the same idea as the MUX32.v.
If `ALUfunc==3'b000` output `A add B` ...
Only the `sari` operation with the small modify, since I need to exclude the `func7` part in it, so it turns to `A add B[4:0]` .

Control.v

Take an input from `instruction` .
`RegWrite` is control bit for `Registers.v` .
`ALUSrc` is control bit for `MUX32.v` .
`ALUop` is a two bit control for `ALU_Control.v` .

Observe the Truth Table, x means don't care.

fn	inst_i[5]	RegWrite	ALUSrc	ALUop[1:0]
add	1	1	0	1x
sll	1	1	0	1x
xor	1	1	0	1x
and	1	1	0	1x
mul	1	1	0	1x
sub	1	1	0	1x
addi	0	1	1	0x
sari	0	1	1	0x

As we can see, we only need the `inst_i[5]`, hence we can write down the formula as

```
ALUOp_o = inst_i[5:4];
ALUSrc_o = ~inst_i[5];
RegWrite_o = 1'b1;
```

Sign_Extend.v

It reads an 12-bit immediate `imm` from `instruction`, output a 32-bit same value in 2's complete.

In 2's complete, if we need to extend the bit, we need to repeat the top bit of the value to let the postive and negtive value remain the same.

Therefore, I done this by simply copy the first 12-bit value, and assign `imm[11]` to `output[31:12]`.

CPU.v

In this module, I want to explain it by some verilog code.

First I defined one 32-bit wire, named `PCnow` to get this cycle's PC value.

In the begining of the cycle, we need to get the PC's value, so I use the TA's module to get this. It output an 32-bit PC value names `PCnow`;

```
PC PC(
    .clk_i(clk_i), .rst_i(rst_i), .pc_i(PCnext),
    .pc_o(PCnow)
);
```

Then I figure out that I need to add PC value by my-self, so I go to implement the PC adder, which takes the PCvalue executing now, and output the next one PC value, I named the wire as `PCnext`, It will connect to the PC and write to it when `clk` is triggered.

```
Adder Add_PC(
    .pc_i(PCnow),
    .pc_o(PCnext)
);
```

Then I need to get the instruction, I named it as `instr`, the prefix of instruction. This operation is also use TA's module.

```
Instruction_Memory Instruction_Memory(
    .addr_i(PCnow),
    .instr_o(instr)
);
```

Then the next step I need to split the `instr` to the desired part, and connect them into the corresponding module. The `ALUResult` is the data to write back to `Registers` which will get later from `ALU`.

```
Control Control(
    .inst_i(instr[6:0]) ,
    .ALUOp_o(ALUop), .ALUSrc_o(MUXAlu), .RegWrite_o(RegWrite)
);

Registers Registers(
    .rst_i(rst_i), .clk_i(clk_i),
    .RS1addr_i(instr[19:15]), .RS2addr_i(instr[24:20]),
    .RDaddr_i(instr[11:7]), .RDdata_i(ALUResult),
    .RegWrite_i(RegWrite),

    .RS1data_o(RData1), .RS2data_o(RData2)
);


Sign_Extend Sign_Extend(
    .imm_i(instr[31:20]),
    .ext_o(ImmData)
);
```

After pass this three module, we can get `ALUop`, the opcode for `ALU_Control` to use, `MUXAlu`, the control bit of `MUX32`, `RegWrite`, the control bit of register write (`1` all the time in this assignment), and the data read from register or immediate `RData1`, `RData2`, `ImmData` from the instruction.
Then then we need to select which data is needed to pass into ALU, and what action the ALU needs to do.

```
MUX32 MUX_ALUSrc(
    .a(RData2), .b(ImmData), .s(MUXAlu),
    .c(ALUSrc2)
);

ALU_Control ALU_Control(
    .ALUop_i(ALUop), .func3(instr[14:12]), .func7(instr[31:25]),
    .ALUfunc_o(ALUfunc)
);
```

Then we get the desired source of data, `RData1` and `ALUSrc2` (from `MUX32` , it decided the `RData2` or the `ImmData` to be the second source of `ALU`), and the function we want the `ALU` can have, `ALUfunc` .
When all the things are ready, we can finally pass the value to `ALU` , and get the desired result `ALUResult` . Then store it back to the `Registers` module.

```
ALU ALU(
    .Data1(RData1), .Data2(ALUSrc2), .ALUfunc_i(ALUfunc),
    .ALUresult_o(ALUResult)
);
```

Development Environment

- OS :
  - Distributor ID: Ubuntu
  - Description: Ubuntu 22.04.2 LTS
  - Release: 22.04
  - Codename: jammy
- compiler :
  - iverilog :
    - Icarus Verilog version 11.0 (stable) ()
- IDE :
  - vscode
    - 1.83.1
    - f1b07bd25dfad64b0167beb15359ae573aecd2cc
    - x64