# Lab04 : Single-Cycle CPU

team 29, teammate ：109550053 楊立嘉，109550137 徐敏芝

# Detailed description of the implementation

- *Decoder.v*

  According to `instr_i` , there are 8 types of condition that we considered.

  And since `wire` can'y be assigned in `always` , I use `reg` data type to help, and assign them back in the end.

| Input or output | Signal name | R-format | ld | sd | beq |
|---|---|---|---|---|---|
| Inputs | I[6] | 0 | 0 | 0 | 1 |
| | I[5] | 1 | 0 | 1 | 1 |
| | I[4] | 1 | 0 | 0 | 0 |
| | I[3] | 0 | 0 | 0 | 0 |
| | I[2] | 0 | 0 | 0 | 0 |
| | I[1] | 1 | 1 | 1 | 1 |
| | I[0] | 1 | 1 | 1 | 1 |
| Outputs | ALUSrc | 0 | 1 | 1 | 0 |
| | MemtoReg | 0 | 1 | X | X |
| | RegWrite | 1 | 1 | 0 | 0 |
| | MemRead | 0 | 1 | 0 | 0 |
| | MemWrite | 0 | 0 | 1 | 0 |
| | Branch | 0 | 0 | 0 | 1 |
| | ALUOp1 | 1 | 0 | 0 | 0 |
| | ALUOp0 | 0 | 0 | 0 | 1 |

  MemtoReg = WriteBack0, ALUSrc = SrcB

  - nop : `instr_i == 7'b0000000`

    set all the output as 0

  - Rtype : `instr_i == 7'b0110011`

    `WriteBack1 = 1'b0` : we want to write our ALUresult back

    `ALUSrcA = don't care` : always PC = PC+4

  - addi : `instr_i == 7'b0010011`

the only difference between *Rtype* and *addi* is `ALUSrcB` , because we need **imm** to enter ALU rather than **src2**.

- lw : `instr_i == 7'b0000011`

  `WriteBack1 = 1'b0` : we want to write the read data back

  `ALUSrcA = don't care` : always PC = PC+4

- rw : `instr_i == 7'b0100011`

  `WriteBack1 = don't care` : just need to write data in *Data Memory*

  `ALUSrcA = don't care` : always PC = PC+4

- beq : `instr_i == 7'b1100011`

  `WriteBack1 = don't care` : just need the **ALUreslut** to compare, so we don't need to do anything after *ALU*.

  `ALUSrcA = don't care` : always PC = PC+4

- jal : `instr_i == 7'b1101111`

```
else if(instr_i == 7'b1101111)begin //jal
    temp_Reg = 1'b1; //write PC+imm into rd
    temp_B = 1'b0;   //not branch
    temp_J = 1'b1;   //jump
    temp_WB1 = 1'b1; //choose PC+imm
    //temp_WB0 = x;  //since WriteBack1 will choose 1, don't need to care
    temp_MR = 1'b0;  //no need to read memory
    temp_MW = 1'b0;  //no need to write memory
    temp_SrcA = 1'b0;//choose PC, PC = PC +imm
    //temp_SrcB = x; //imm is for PC, don't need to care here
    //temp_ALUOp[1] = x; //don't need to do ALU
    //temp_ALUOp[0] = x;
end
```

- jalr : `instr_i == 7'b1100111`

```
else if(instr_i == 7'b1100111)begin //jalr
    temp_Reg = 1'b1; //write src1+imm into rd
    temp_B = 1'b0;   //no branch
    temp_J = 1'b1;   //jump
    temp_WB1 = 1'b1; //choose src1+imm
    //temp_WB0 = x;  //since WriteBack1 will choose 1, don't need to care
    temp_MR = 1'b0;  //no need to read memory
    temp_MW = 1'b0;  //no need to write memory
    temp_SrcA = 1'b1;//choose src1, PC = src1 +imm
    //temp_SrcB = x; //imm is for PC, don't need to care here
    //temp_ALUOp[1] = x; //don't need to do ALU
    //temp_ALUOp[0] = x;
end
```

- *ALU_Ctrl.v*

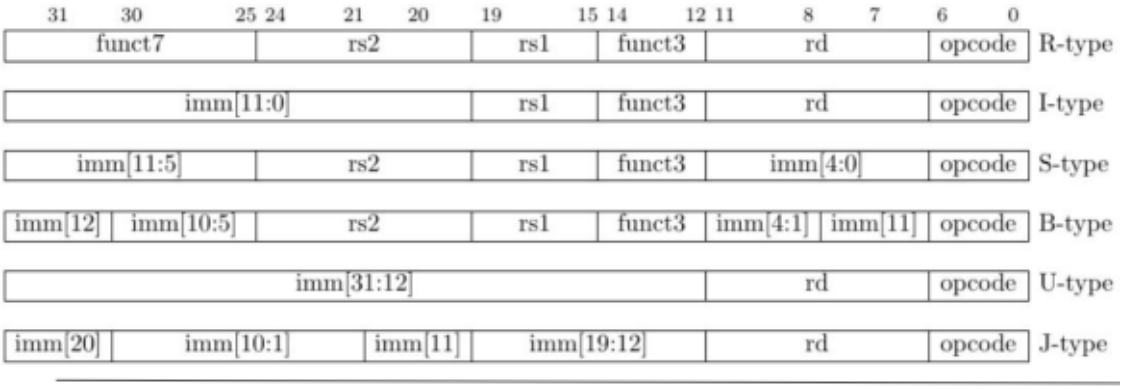
use `ALUOp` to decide whether it is **Rtype** , **lw, sw** or **beq** (jal and jalr won't need ALU)

In **Rtype** , we need `func3` and `instr` help to seperate *add, addi* and *slt*

```
if(ALUOp == 2'b10)begin // Rtype
    if(func3 == 3'b000 ) ALU_Ctrl_o = 4'b0010; //add,addi
    else if(instr == 4'b0010) ALU_Ctrl_o = 4'b0111; //slt

end
else if(ALUOp == 2'b00)begin //lw, sw
    ALU_Ctrl_o = 4'b0010;
end
else if(ALUOp == 2'b01)begin //beq
    ALU_Ctrl_o = 4'b0110;
end
```

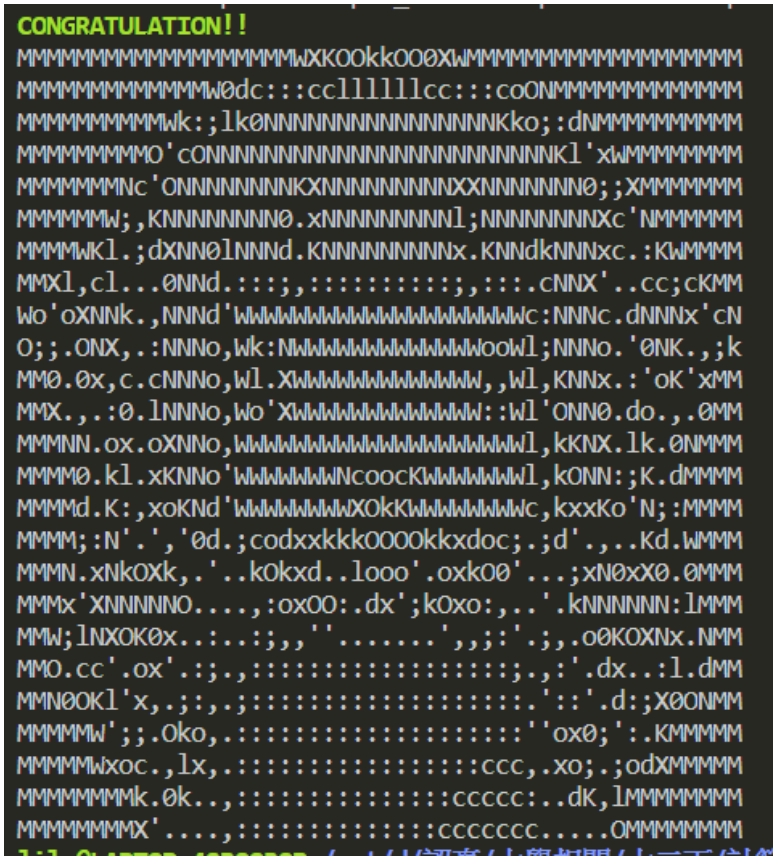
- *Imm_Gen.v*
  - For sign extension, because for every type of instructions, the MSB are at `instr_i[31]`, so I just copy `instr_i[31]` to fill up till 32 bits.
  - And the only thing need to check is what are types for those instructions, and once you know which type the instruction belongs to, just follow the picture below to extract `imm` from instruction.
    - `addi`, `lw`, `jalr` → I type; `sw` → S type; `beq` → B type; `jal` → J type
    - `add` → R type, which do not need `ImmGen`, so we didn't write anything, means don't care.




- *alu.v*
  - Because we don't need to extend the `alu.v` in our lab3, so we decided to use verilog operator directly.
  - We add `signed` to our `src1` and `src2`.

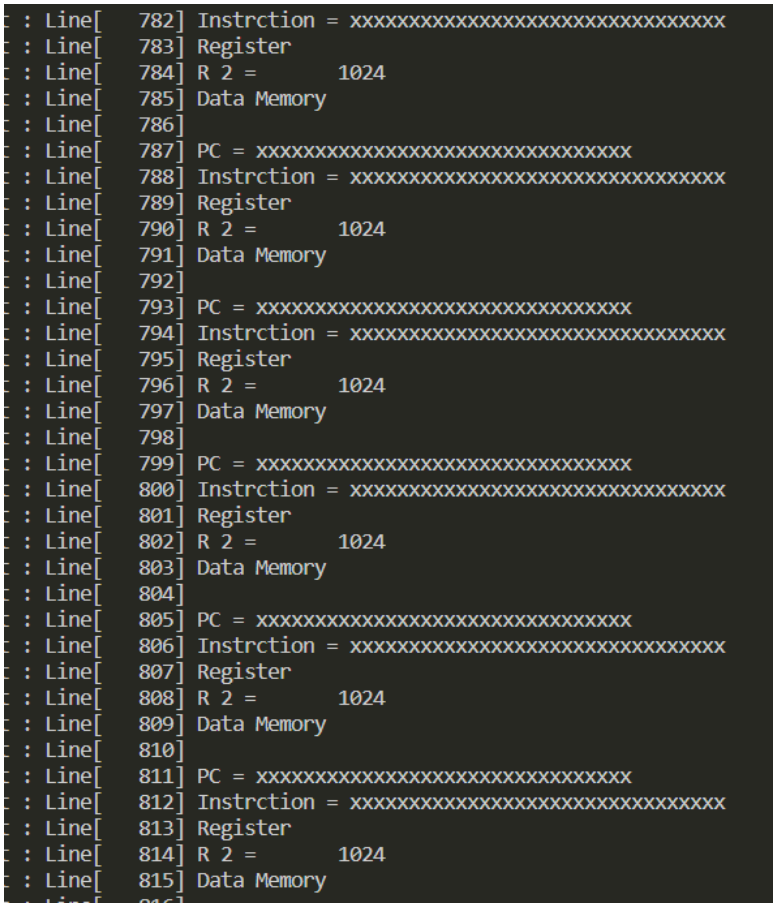
- *single_cycle_CPU.v*
    - We add lots of wire by ourself to transmit data, just follow the circuit diagram that TAs gives in slide, connect every wire to where it should go.

# Implementation results




# Problems encountered and solutions

- At the first time we test our code, we found that our results of every test cases are `XXXX...XXX` (like the screenshot below), we found that instruction is set to be `32'b0` means a `nop` instruction, and when we encounter a `nop`, we should set every control line equal to `0` to make sure circuit will not do anything. So after we assign `0` to every control line whenever `instr_i == 7'b0000000`, we fix this mistake.




- It is easy to set control output in decoder wrong uncarefully, so we spend lots of time to find out which signal is set wrong.
- We remember that teacher said in class: “The instructions belong to same type will share the same opcode. ”, but we found that, `jalr` is I type instruction but with opcode `1100111` (which is not equal to `0010011` ), `lw` is R type instruction but with opcode `0000011` (which is not equal to `0110011` ).