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 | 1[6] | 0 | 0 | 0 | 1 |
| | I[5] | 1 | 0 | 1 | 1 |
| | I[4] | 1 | 0 | 0 | 0 |
| | 1[3] | 0 | 0 | 0 | 0 |
| | I[2] | 0 | 0 | 0 | 0 |
| | I[1] | 1 | 1 | 1 | 1 |
| | I[O] | 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 |
| | ALUOpO | 0 | 0 | 0 | 1 |

MemtoReg = WriteBack0, ALUSrc = SrcB

o nop: instr_i == 7'b00000000
set all the output as 0

o Rtype: instr_i == 7'b0110011

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

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

o addi: instr_i == 7'b0010011

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

```
o lw : instr_i == 7'b0000011
   WriteBack1 = 1'b0 : we want to write the read data back
   ALUSTCA = don't care : always PC = PC+4
o 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
o 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.
  ALUSTCA = don't care : always PC = PC+4
• |a|: 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
```

o jalr: instr_i == 7'b1100111

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 \rightarrow I type; sw \rightarrow S type; beq \rightarrow B type; jal \rightarrow J type
 - add → R type, which do not need ImmGen, so we didn't write anything, means don't care.

| 31 | 30 2 | 5 24 | 21 | 20 | 19 | 15 | 14 1 | 2 11 | 8 | 7 | 6 | 0 | |
|---------|----------------|-------|--------|--------|-----|------|--------|-------|----------|---------|------|-----|--------|
| | funct7 | | rs2 | | rs1 | | funct3 | | re | l | opco | ode | R-type |
| | | | | | | | | | | | | | |
| | imm[] | [1:0] | | | rsl | | funct3 | | ro | l: | opco | ode | I-type |
| _ | [11.5] | | - 0 | | | _ | 6 | | | 4.0 | | 1 | 0.7 |
| 111 | nm[11:5] | | rs2 | | rsl | | funct3 | | imm | 4:0] | opco | ode | S-type |
| [imm[12 | imm[10:5] | | rs2 | | rs1 | _ | funct3 | limr | n[4:1] [| imm[11] | one | odo | B-type |
| mmi[12 |] IIIIII[10.5] | | 152 | | 151 | | Tuncto | 11111 | 11[4.1] | mm[11] | ope | oue | D-type |
| | | imr | n[31:1 | 2 | | | | Т | ro | | opco | ode | U-type |
| | | | | , | | | | | | | | | |
| imm[20 |] imm[1 | [0:1] | in | nm[11] | imn | n[19 | 9:12] | | ro | l | opco | ode | J-type |
| | | | | | | | | | | | | | |

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

```
CONGRATULATION!!
MMMMMMMMWk:;1k0nnnnnnnnnnnnnnnkko;:dnmmmmmmmm
MMMMMMNc'ONNNNNNNKXNNNNNNNNNXXNNNNNNNO;;XMMMMMM
MMMMMMW;, KNNNNNNNNO.xNNNNNNNNN1; NNNNNNNNXc'NMMMMMM
MMMMWK1.;dXNN01NNNd.KNNNNNNNNNX.KNNdkNNNxc.:KWMMMM
MMX1,c1...0NNd.:::;,:::::::;,:::.cNNX'..cc;cKMM
WO'OXNNK., NNNd'WWWWWWWWWWWWWWWC:NNNc.dNNNx'cN
O;;.ONX,.:NNNo,Wk:NWWWWWWWWWWWWOOWl;NNNo.'0NK.,;k
MM0.0x,c.cNNNo,W1.XWWWWWWWWWWWW,,W1,KNNx.:'OK'xMM
MMX.,.:0.1NNNo,Wo'XWWWWWWWWWWWW::Wl'ONN0.do.,.0MM
MMMM0.kl.xKNNo'WWWWWWNcoocKWWWWWWl,kONN:;K.dMMMM
MMMMd.K:,xoKNd'WWWWWWWXOkKWWWWWWWc,kxxKo'N;:MMMM
MMMM;:N'.','0d.;codxxkkkOOOOkkxdoc;.;d'.,..Kd.WMMM
MMMN.xNkOXk,.'..kOkxd..looo'.oxkO0'...;xN0xX0.0MMM
MMMx'XNNNNNO....,:ox00:.dx';k0xo:,..'.knnnnnn:lmmm
MMW; lnxok0x..:..;,,''......',,;:'.;,.o0k0xnx.nmm
MMO.cc'.ox'.:;,,:::::::::::::;,,:'.dx..:l.dMM
MMMMMW';;.Oko,..................''ox0;'..KMMMMM
MMMMMWxoc.,lx,..::::::::::ccc,.xo;.;odXMMMMM
MMMMMMMk.0k..,::::::::::ccccc:..dK,lMMMMMMM
MMMMMMMX'....,:::::::::::::ccccccc.....OMMMMMMMM
```

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'b00000000, we fix this mistake.

Lab04 : Single-Cycle CPU

```
782] Instrction = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
: Line
     783] Register
: Line
     784] R 2 =
               1024
     785] Data Memory
: Line
: Line
     786
: Line
     : Line
     : Line
     789] Register
     790] R 2 =
: Line
               1024
: Line
     791 Data Memory
: Line
     792]
: Line
     : Line
: Line
     795] Register
               1024
: Line[
     796] R 2 =
 Line
     797 Data Memory
: Line
     798
: Line
     : Line
     Line
     801]
        Register
: Line[
     802 R 2 =
               1024
: Line[
     803] Data Memory
     804]
: Line[
 Line
     : Line[
     : Line[
     807] Register
     808] R 2 =
809] Data Memory
: Line[
              1024
: Line
: Line
     810]
: Line[
     : Line
     813]
 Line
        Register
: Line
     814] R 2 =
               1024
: Line[
     815] Data Memory
```

- 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, <code>jalr</code> is I type instruction but with opcode <code>ll00111</code> (which is not equal to <code>0010011</code>), <code>lw</code> is R type instruction but with opcode <code>0000011</code> (which is not equal to <code>0110011</code>).

Lab04 : Single-Cycle CPU