1. 实验目的和要求

Purpose

- 理解 **RISC-V RV32I**指令集的指令

  Understand RISC-V RV32I instructions
- 设计执行**RV32I**指令集的流水线数据通路

  Master **the design methods of pipelined CPU** executing RV32I instructions
- 设计流水线的**forwarding**探测模块和传递模块

  Master the method of **Pipeline Forwarding Detection** and **bypass unit** design.
- 设计1周期暂停的预测不跳转的分支设计

  Master the methods of **1-cycle stall of Predict-not-taken** branch design
- 设计程序验证流水线

  Master the methods of **program verification of Pipelined CPU** executing RV21I instructions

Task

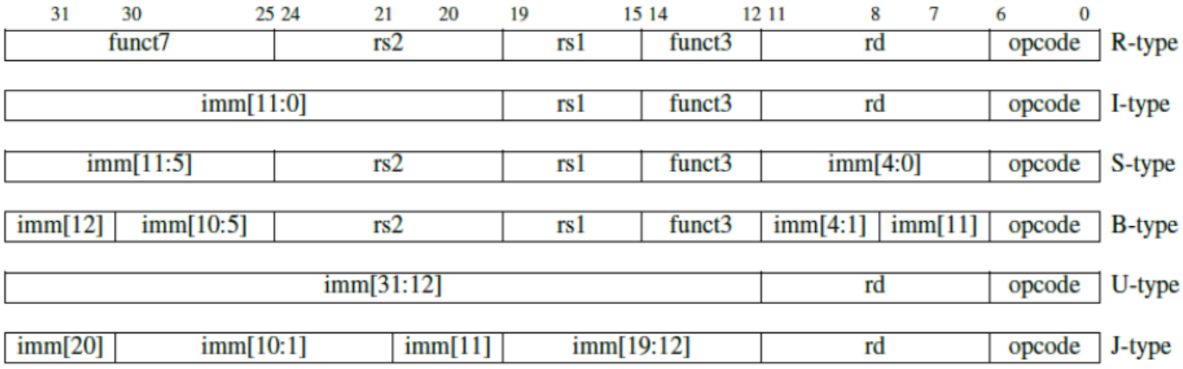
Design of Pipelined CPU executing RV32I instructions:

- Design datapath
- Design Bypass Unit
- Design CPU Controller

2. 实验内容和原理

Instruction

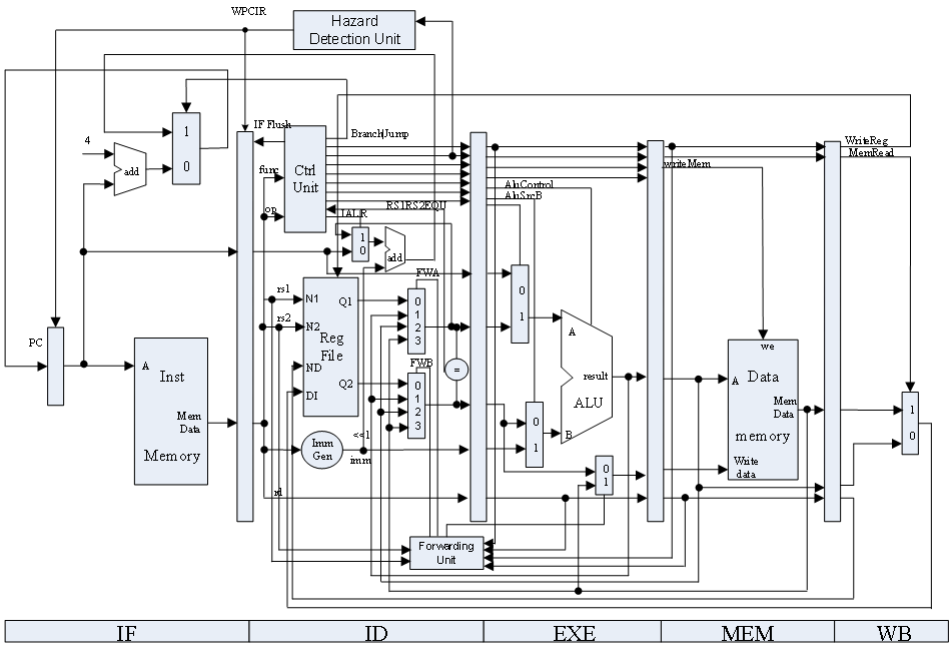
All RV32I instructions follow certain formats, which can be divided into 6 types and are shown in the following figure. The pipeline should decode each instruction according to the formats.



Datapath

The datapath used in this term is shown in the following figure. There are some noteworthy points in the following figure, which makes the design different from the one in course *Computer Organization* and more efficient in my opinion:

1. The operand of ALU is determined in phase *ID*
2. Whether to jump is decided in phase *ID*
3. Hazard Deteciton Unit can be treated as a part of phase *ID*
4. More types of hazard are considered



Hazard

All the types to be considered are as follow:

- the source register of the current instruciton is the same as the destination register of the last instruction
    - similar to

```
add x1, x2, x3
add x2, x1, x3
```

        - no stall is needed
        - the result of ALU in phase *EXE* should be forwarded
    - similar to

```
lw x1, 0(x0)
add x2, x1, x3
```

        - one stall is needed
        - the result of reading memory in phase *MEM* should be forwarded
    - similar to

```
lw x1, 0(x0)
sw x1, 4(x0)
```

        - no stall is needed
        - the result of reading memory should be forwarded
- the source register of the current instruciton is the same as the destination register of the instruction before last
    - similar to

```
add x1, x2, x3
...
add x2, x1, x3
```

        - no stall is needed
        - the result of ALU in phase *MEM* should be forwarded
    - similar to

```
lw x1, 0(x0)
...
sw x1, 4(x0)
```

        - no stall is needed
        - the result of reading memory should be forwarded
- control hazard
    - the pipeline adopts the method of Predict-Not-Taken
    - if the branch result is taken, the incorrect instructions in the pipeline should be flushed

3. 实验过程和数据记录

There are 4 files that need to be finished: cmp_32.v, CtrlUnit.v, RV32core.v and HazardDetectionUnit.v. It is quite easy to fill in the blanks in the first file, so the process of that is left out here.

CtrlUnit.v

This module is the core of phase *ID*. According to **the instruction formats** and with the help of the **figure of datapath**, it's not hard to finish this file. It is somehow complicated but not difficult. **The point is to clearly figure out the usage of each signal or each line.**

Take the line `rs2use` as an example. It tells that whether the instruction being decoded will use rs2 in the following phase. That is, when decoding an instruction of R-type, B-type or S-type `rs2use` should be set to 1, which makes:

```
assign rs2use = R_valid | B_valid | S_valid ;
```

A point worth noting is the position of cmp_32.v. Since the branch result is decided in this module, so I wanted to put module cmp_32 within CtrlUnit. However, I found module cmp_32 already in RV32core. Then I realized that this is also a feasible method which makes better use of the features of hardware. The usage of lines `hazard_optype` is still unclear and so they are left out.

RV32core.v

This module is the top implementation of a pipeline. Thanks to the **regular naming** of lines and the **datapath**, simply guessing the usage of each line and filling in the blanks is all to be done.

Take the module `mux_forward_A` as an example. Simply referring to the datapath tells the answer. The four inputs to be selected comes from regfile, the ALU result in phase *EXE*, the ALU result in phase *MEM* and memory respectively. Finishing this module is even easier than finishing CtrlUnit.v.

HazardDetectionUnit.v

This module is the most challenging one. I felt like making it a combinational logic circuit at first, but found it incovenient. In the sequential logical block, four if-else statements are used and each corresponds to certain outputs. I arrange in this way so that **the assignments of the same lines are put together**, which makes the code easy to read and debug.

`forward_ctrl_A` and `forward_ctrl_B` determine the **correct source of register data**, and their conditions for each value are almost the same. Take `forward_ctrl_A` as an example.

- It selects the source of the ALU result in phase *EXE* (`forward_ctrl_A=1`) when **the last instruction updates a register** (`Regwrite_EXE==1`) and **the destination register of last instruction is the same as the source register of the instruction being decoded**. Also, we can notice that, if the source register is x0, no forwarding is needed. Combine these conditions together and we have the final condition: `rd_EXE == rs1_ID && rs1use_ID && rs1_ID!=0 && Regwrite_EXE`
- It selects the source of the ALU result in phase *MEM* (`forward_ctrl_A=2`) when **the last instruction updates a register by calculation** (`Regwrite_EXE==1`) and **the destination register of last instruction is the same as the source register of the instruction being decoded**. Here, we should distinguish this source from the one coming from memory, and so I make use of signal `DatatoReg_MEM`, which can tell whether the data comes from memory. Also, we can notice that, if the source register is x0, no forwarding is needed. Combine these conditions together and we have the final condition: `rd_MEM == rs1_ID && rs1use_ID && rs1_ID!=0 && DatatoReg_MEM==0`
- It selects the source of the memory in phase *MEM* (`forward_ctrl_A=3`) **under almost the same condition as the one of ALU result in phase *MEM***. The only difference is that `DatatoReg_MEM==1`, which means **this hazard happens when the up-to-date register data is read from memory**. Combine these conditions together and we have the final condition: `rd_MEM == rs1_ID && rs1use_ID && rs1_ID!=0 && DatatoReg_MEM`
- If all the three conditions above are not satisfied, **no hazard happens**. Then `forward_ctrl_A=0`, the data in regfile is up-to-date.

`forward_ctrl_ls` determines **the correct source of data to be written into memory**.

- It selects the data just read from memory (`forward_ctrl_ls=1`) when **a store instruction coming right after a load instruction**, and **the source register of the store instruction is the same as the destination register of the load instruction**. Again, I use `DatatoReg_MEM` to tell whether the instruciton in phase *MEM* is a load instruction. Combine the conditions together and we obtain: `rs2_EXE == rd_MEM && DatatoReg_MEM`
- If the condition above is not satisfied, **no hazard happens** and `forward_ctrl_ls=0`.

The rest outputs are put together since they together control the working of the whole pipeline, by controlling the state of registers between pahses.

- It insert a bubble when **a load instruction is right before an instruciton that needs the data just read from memory for calculation**, which means if the latter instruction is a store instruction no stall is needed. Since the data is read from memory at the end of a clock cycle, the lack of a stall may cause insecurity or lower the clock frequency. The register between *IF* and *ID* is designed well and simple assignment is enough.

```
if( ((rd_EXE == rs1_ID && rs1use_ID && rs1_ID!=0) || (rd_EXE == rs2_ID &&
rs2use_ID && rs2_ID!=0 && WR_ID==0)) && DatatoReg_EX) begin
    // ld, add, (need stall)
    PC_EN_IF <= 0;
    reg_FD_EN <= 1;
    reg_FD_stall <= 1;
    reg_FD_flush <= 0;
    reg_DE_EN <= 1;
    reg_DE_flush <= 1;
    reg_EM_EN <= 1;
    reg_EM_flush <= 0;
    reg_MW_EN <= 1;
end
```

- When a **control hazard** happens, we simply flush the incorrect instructions. Thanks to the good desgin of the registers, the assignments are easy.

```
else if(Branch_ID)begin
    // update PC
    // flush reg_if_id
    PC_EN_IF <= 1;
    reg_FD_EN <= 1;
    reg_FD_stall <= 0;
    reg_FD_flush <= 1; // key
    reg_DE_EN <= 1;
    reg_DE_flush <= 0;
    reg_EM_EN <= 1;
    reg_EM_flush <= 0;
    reg_MW_EN <= 1;
end
```

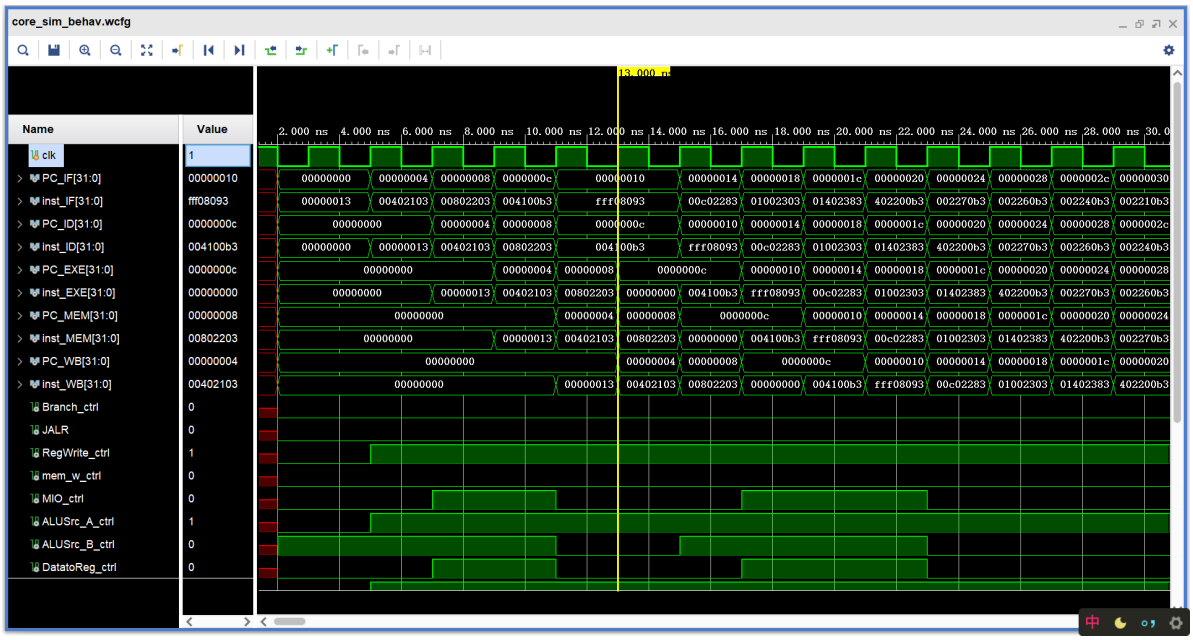
- If the two conditions above are not satisfied, no hazard happens.

```
else begin
// everything normal
PC_EN_IF <= 1;
reg_FD_EN <= 1;
reg_FD_stall <= 0;
reg_FD_flush <= 0;
reg_DE_EN <= 1;
reg_DE_flush <= 0;
reg_EM_EN <= 1;
reg_EM_flush <= 0;
reg_MW_EN <= 1;
end
```

4. 实验结果分析

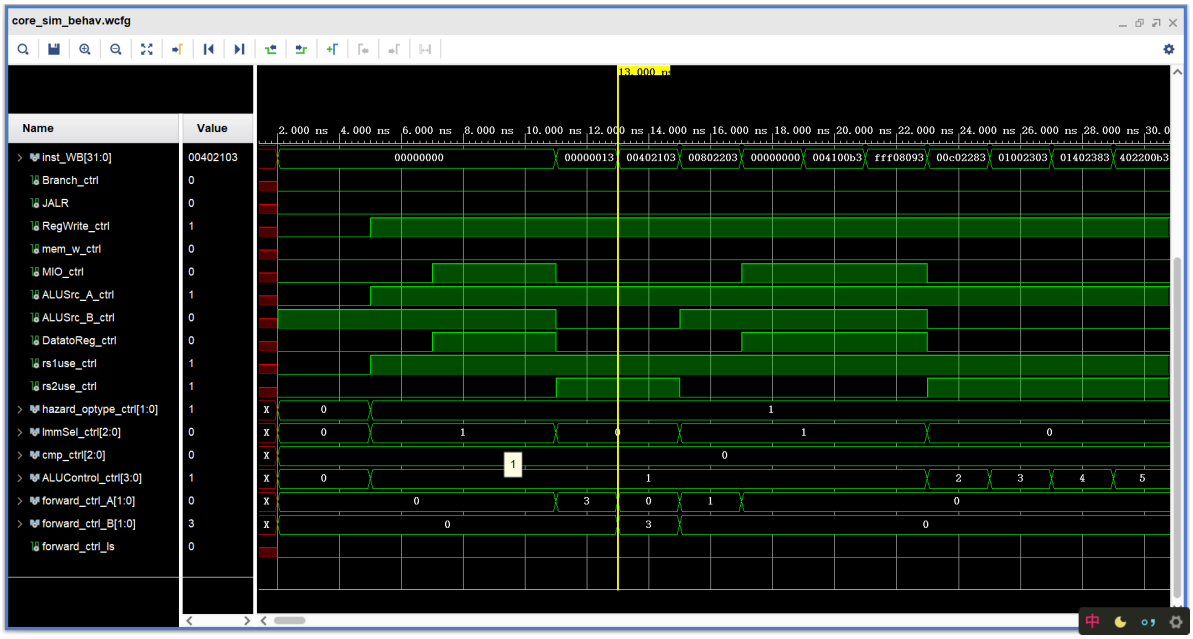
Simulation

Run the simulation and we can see the result below.

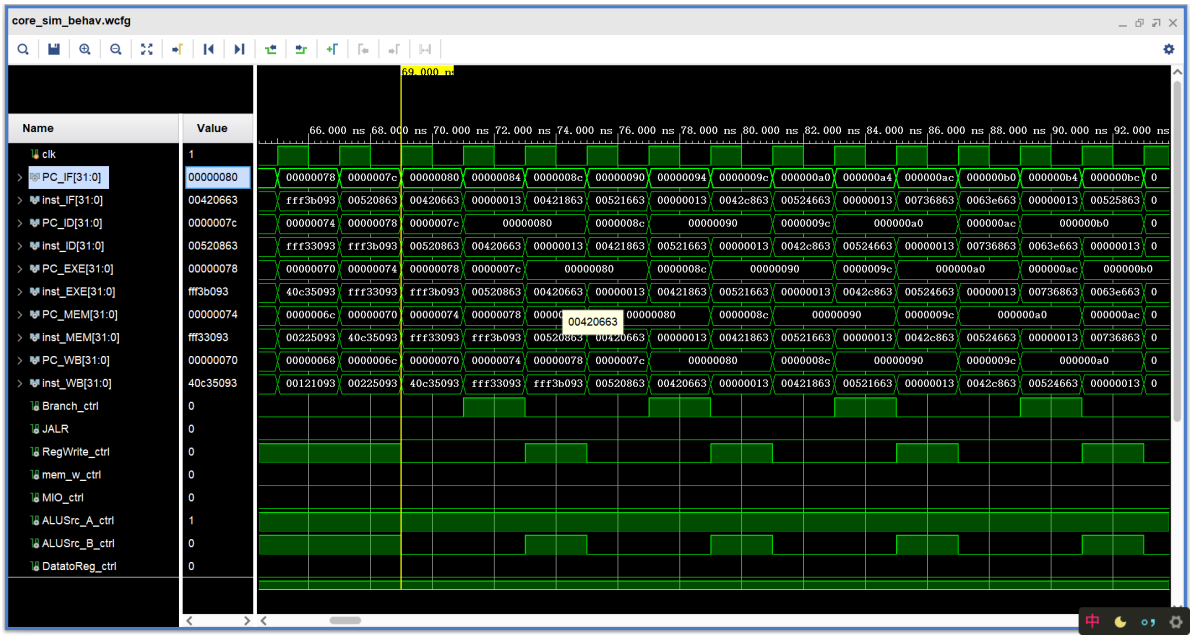


Referring to the instruction we can find that there is a hazard needing one stall between instruction at 0x0C and 0x08. We can see when `PC_ID` reaches 0x0C, `PC_IF`, `inst_IF`, `PC_ID` and `inst_ID` **remain unchanged for 2 clock cycles**, and that is the proof of a stall.

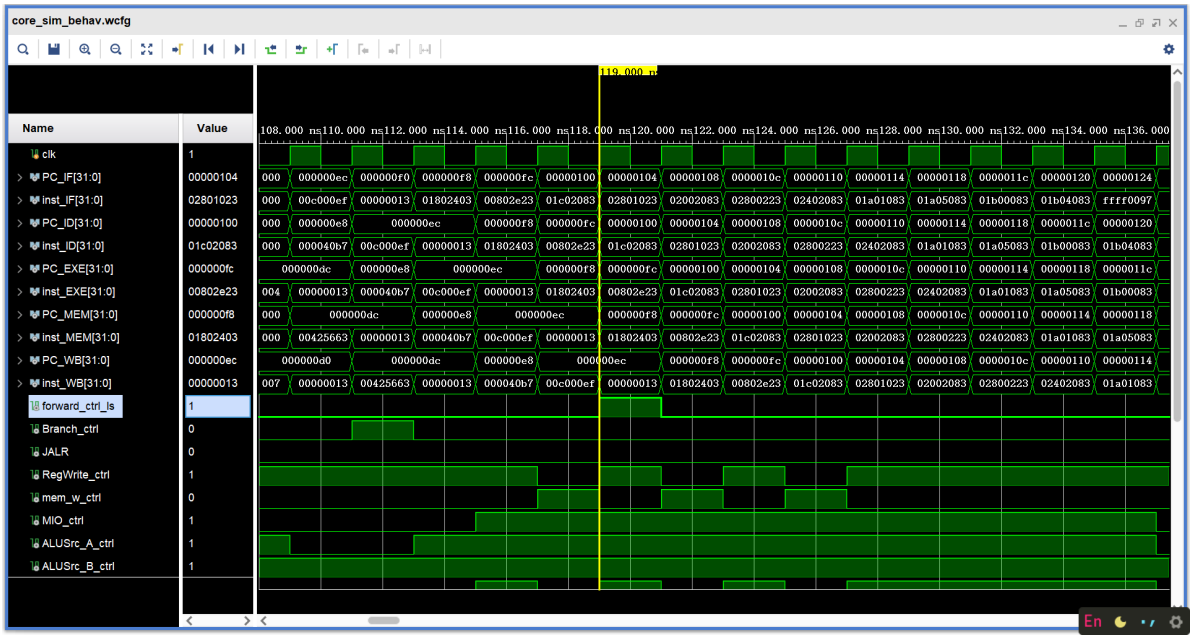
The hazard mentioned above also requires a forwarding. We can see at this time `forward_ctrl_B` is 3. Then, there is a hazard needing forwarding between instruction at 0x0C and 0x10. We can see when `PC_ID` reaches 0x10, `forward_ctrl_A` is 1. That is correct.



Then, let's check the function of branch instructions and omit those insignificant instructions. The instruction at 0x7C is a branch instruction that is not taken. The instruction at 0x80 is a branch instruction that should be taken. We can see the pipeline keeps on working sequentially after the first branch instruction. However, the second branch instruction occupies 2 clock cycles in phase *ID*, and **the instruction in this phase becomes `nop` in the second clock cycle**. Focus on the PC of phase *IF*, we can see that **0x88 is skipped because the branch is taken**. We can know the branch instructions function well.

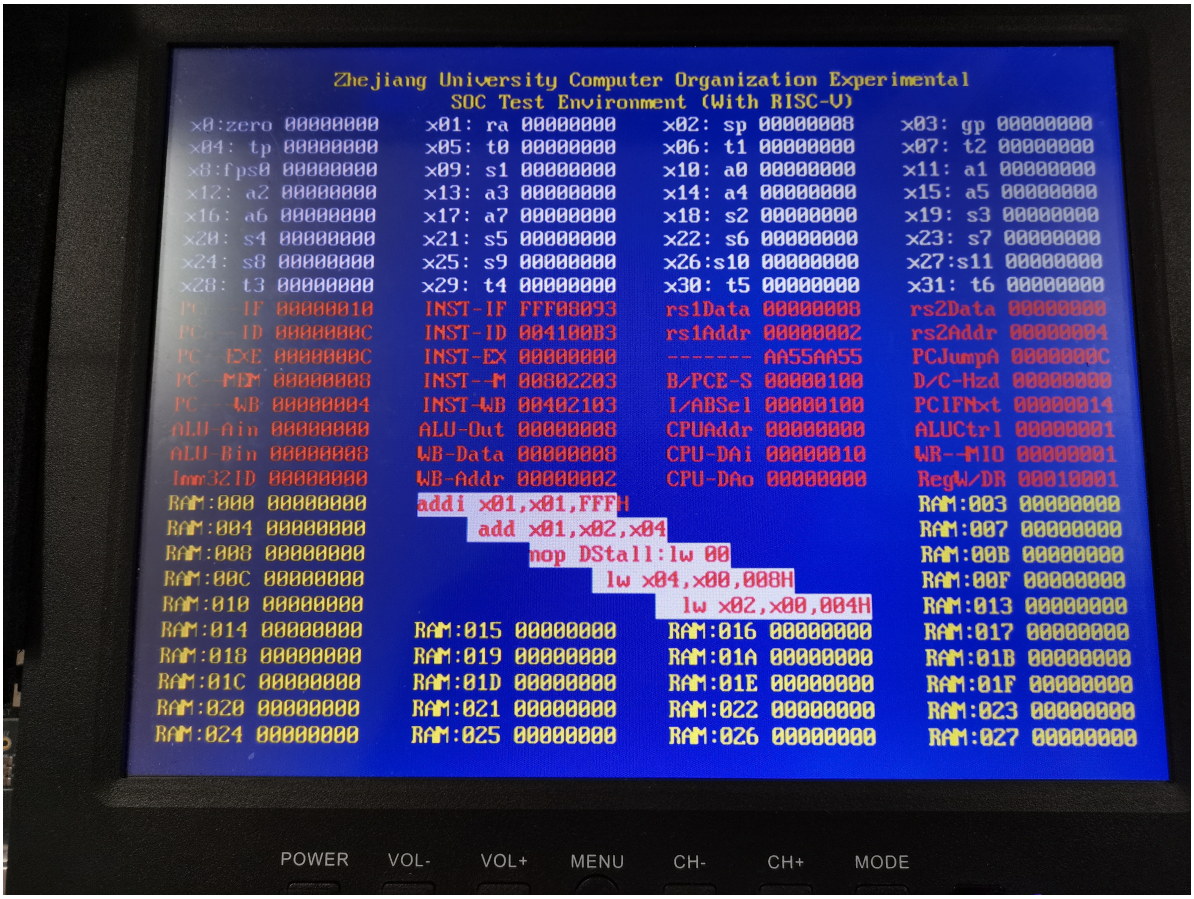


The last one should be noticed is that `sw x8, 28(x0)` coming right after `lw x8, 24(x0)`. The figure below shows that the forwarding signal `forward_ctrl_ls` functions as expected.

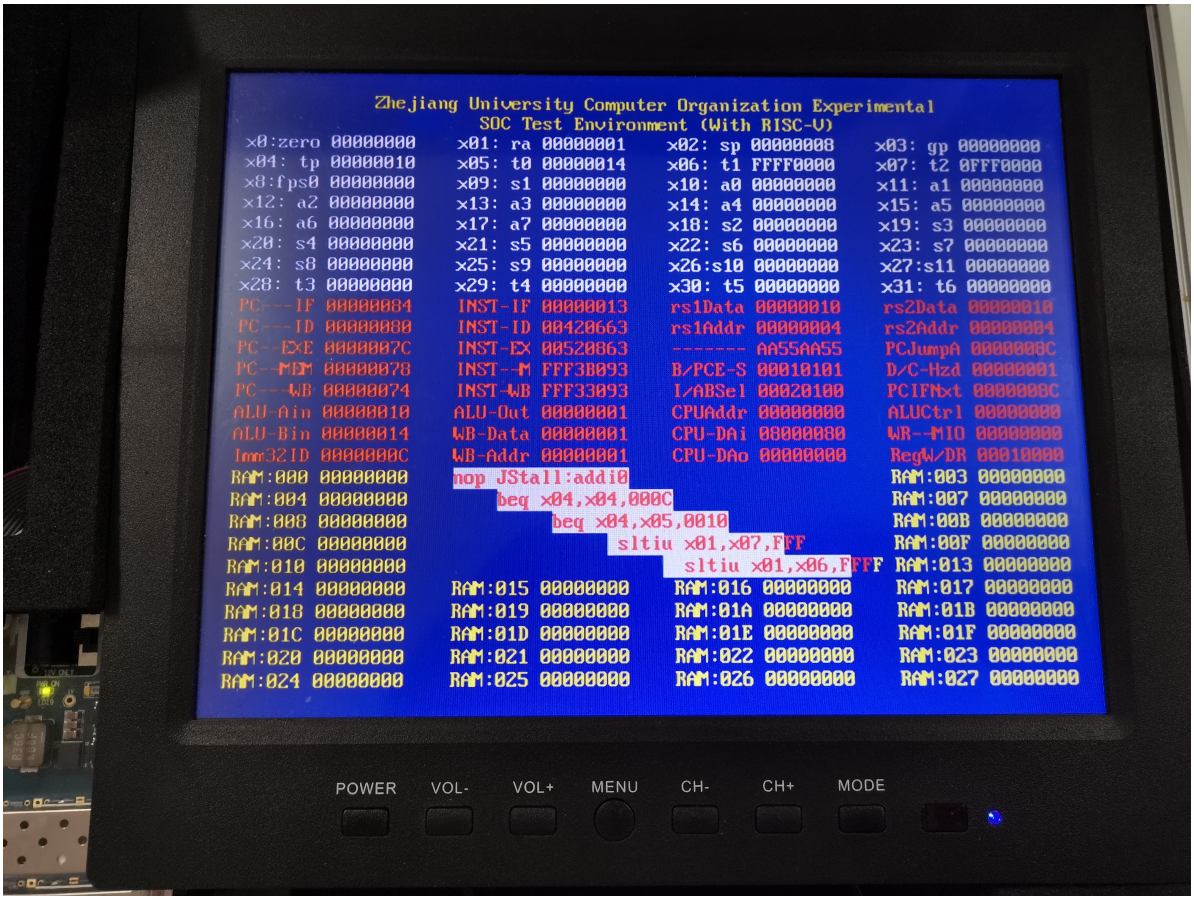


Program Device

Run the program on a FPGA board and check some points. The first one to check is that `add x1, x2, x4` coming right after `lw x4, 8(x0)`. We know that a bubble should be inserted. The figure below shows that a bubble is successfully inserted.

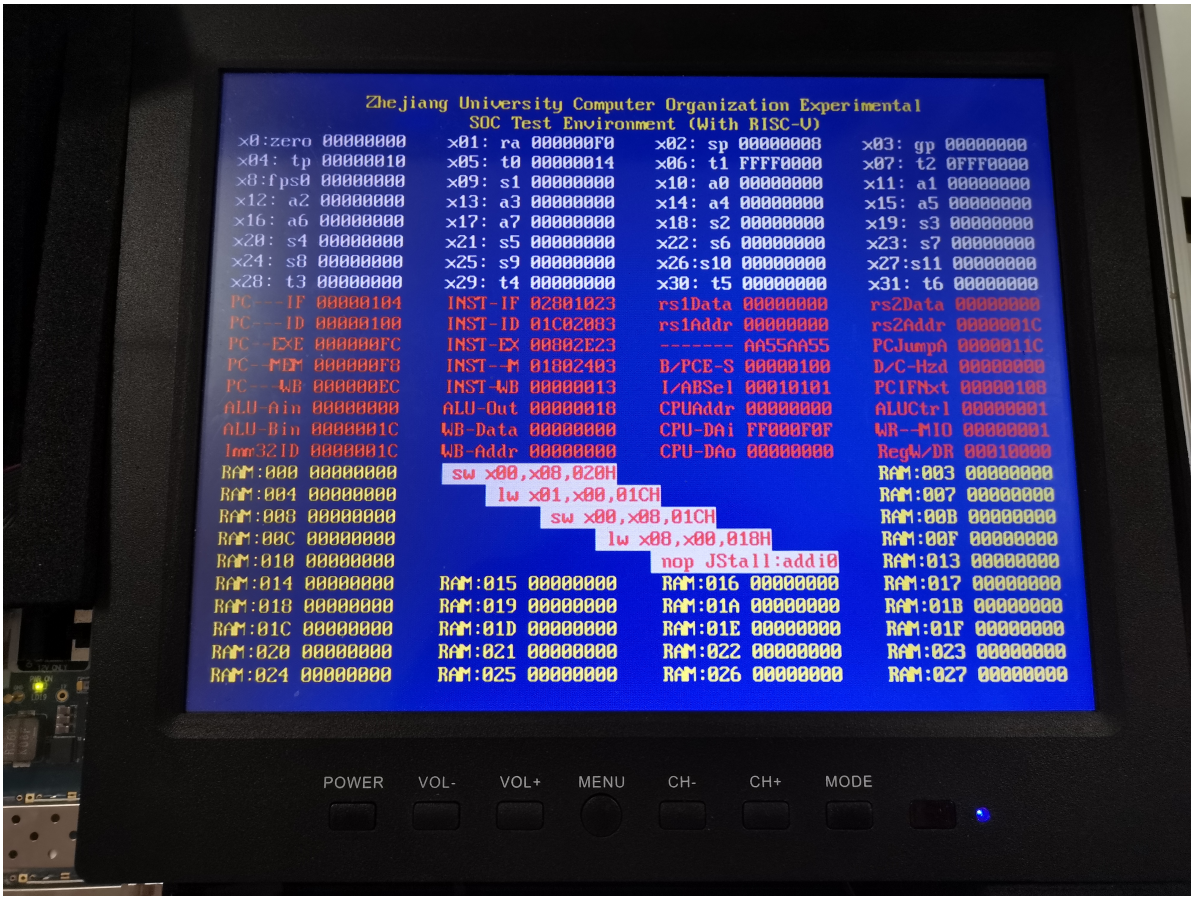


The second point to check is that flushing incorrect instructions when the branch should be taken. There are 2 branch instructions in the figure below. The first one does not branch and hence the instructions after it are not affected by it. The second one `beq x4,x4,label0` should branch, and we can see the incorrect instruction in phase *IF* is flushed.Then the pipeline goes on as the simulation result.

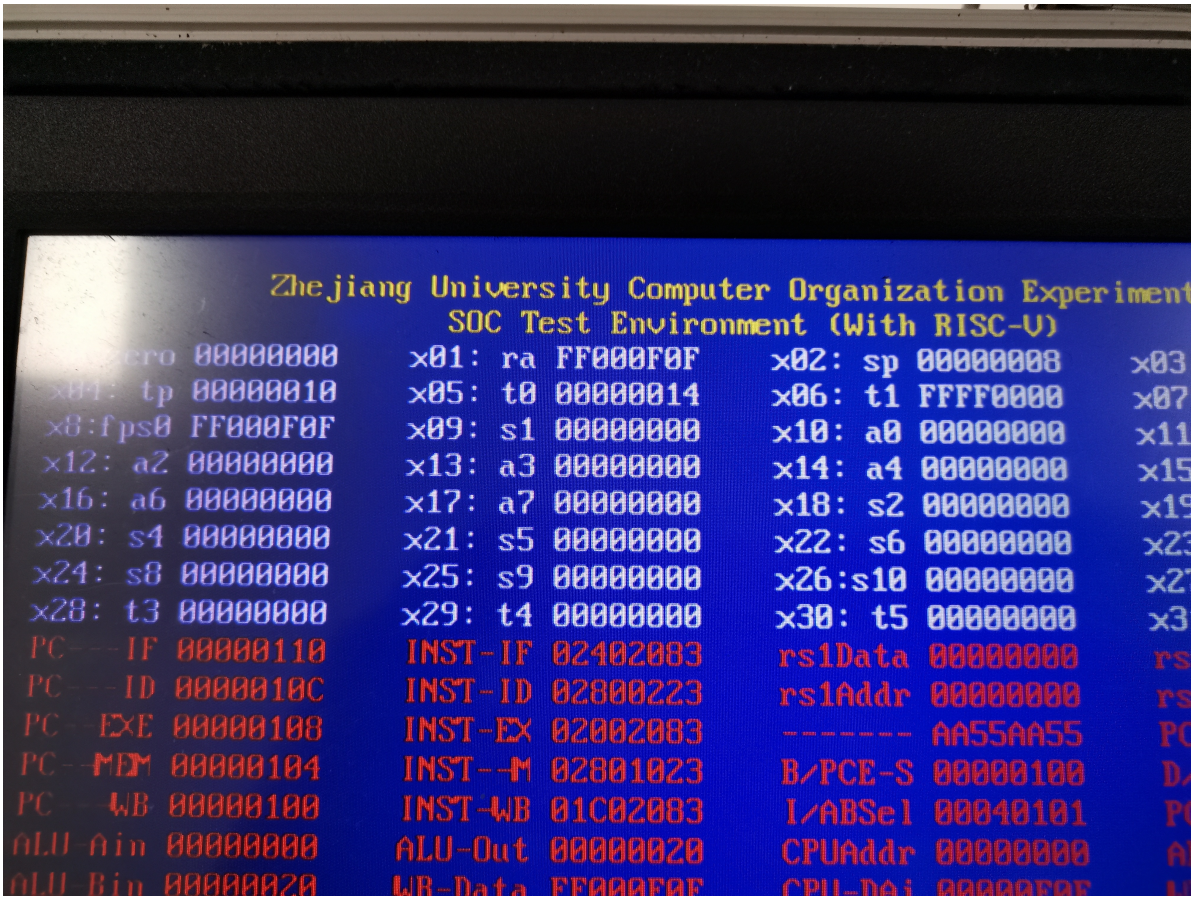


The third point to check is that flushing incorrect instructions when there is a jump instruction. Just the same as the second point, the incorrect instruction in phase *IF* is flushed. Then the pipeline goes on as the simulation result.

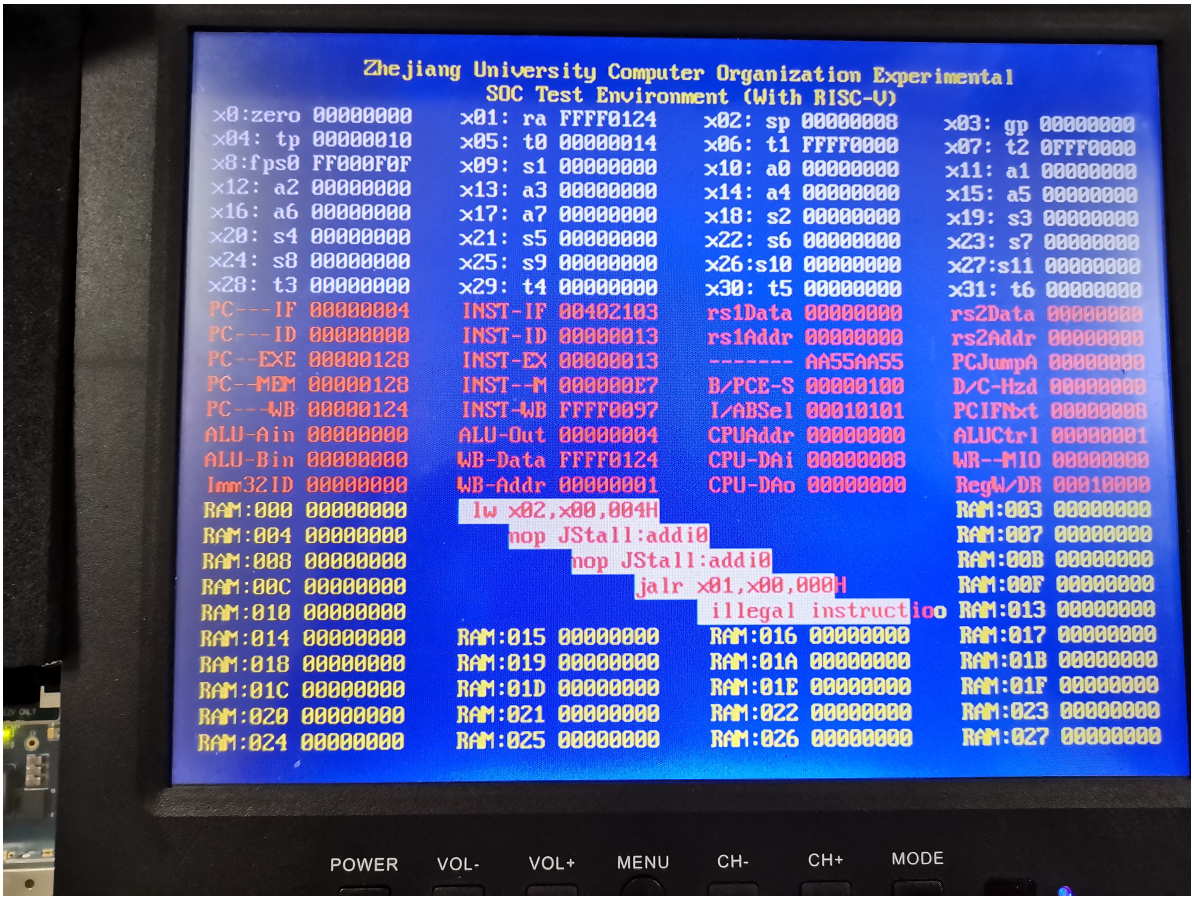
The fourth point to check is that `sw  x8, 28(x0)` coming right after `lw  x8, 24(x0)`. The data just read from memory should be forwarded immediately and no stall is needed. The figure below shows that after the load instruction finishes phase *MEM*, no stall happens. The forwarding should work well, as previous simulation shows.



The figure shows the content of register x8 after the instruction is finished, which is 0xFF000F0F, the content at address 0x18 in memory.



The last point to check is the functionality of `jalr`. The figure below shows that the incorrect instructions are flushed and PC returns to the beginning. The content of x1 is not 0x12C because the execution of `jalr` hasn't been completely done, and so **the content has not been written back into regfile yet** (figure this out after the discussion in the groupchat).



5. 讨论与心得

I encounter some problem while coding. The most important 2 ways to solve problem is to clearly clarify the functionality of the lines and to refer to the datapath. Keeping effective communication with the partner is the secret to efficient development.

Some impressive points are as follow.

1. `rs1use` and `rs2use` tells whether the instruction uses source registers. They are important since we are implementing a hardware architecture. Even if the instruction does not tend to use source registers, the hardware will always decode source registers from the instruciton.
2. The condition for `forward_ctrl_ls` does not take the type of instruction into consideration. We do that because of the line `WE_MEM`, which controls the writing of memory. If the instruction is not of S-type, the data will never be written into memory.
3. The final point is when `PC+4` is written back into regfile during the execution of `jal(r)`. To be honest, the code for this function is somehow too covert for me. After the discussion then I realize the cleverness of modularity. I did not tend to complete a certain functionality and I thought the work was not done at first, but in fact, I finished each module then I completed everything.

The implementation of a pipeline this time is different from the one I've learned before, and it even seems unnatural and unfriendly to me. But I have to admit the it is very efficient and ingenious.