1 Usage

```
$ sudo apt-get install gtkwave
$ cd pipelineCPU
$ make
```

Gtkwave is used to display waveform. If you installed it, you can ignore the first code. The input file of instruction memory is “instructions.bin”. You should put the instruction machine codes which you’d like to test into this file in advance.

The output file of data memory is “data.bin”. You can observe the number of clock cycles in the console.

2 Overview

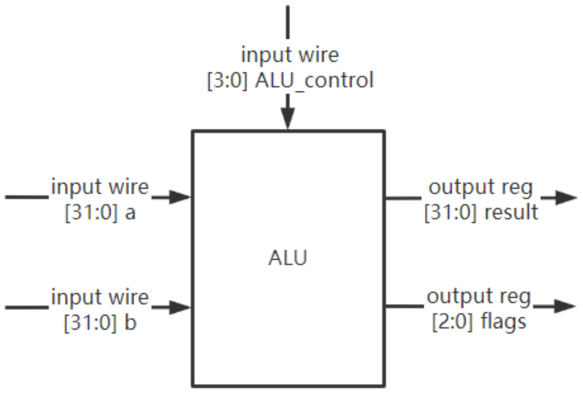
In project 4, we are required to implement a 5-stage pipelined CPU using Verilog language which can execute the MIPS instructions. We are provided the instruction memory module and data memory module. We are expected to implement the register file, ALU module, and the control unit. For testing, we have totally 8 test files, which have different types of hazard existing in each test file.

In my project, I design the CPU with 6 files: alu.v, CONTROL_UNIT.v, CPU.v, RegisterFile.v, Instruction.v, and MainMemory.v. In alu.v, I implement all arithmetic methods including add, addi, sub, slt, etc. In CONTROL_UNIT.v, I implement the controller in the pipelined CPU, which can generate different control signals for different instructions at different stages. In RegisterFile.v, 32-bit register groups are provided. The Instruction.v and the MainMemory.v are provided by the teacher, I use them to get the series of instructions and operate the main memory to modify their value according to the instructions. Finally, I use the CPU.v file as the top-level file of the whole pipelined CPU, which organizes other .v files to achieve all the functions of the CPU. In addition, I solve all the data hazards and control hazards using the methods of forwarding and stalling to make my pipelined CPU run as fast as it can.

3 Main Elements

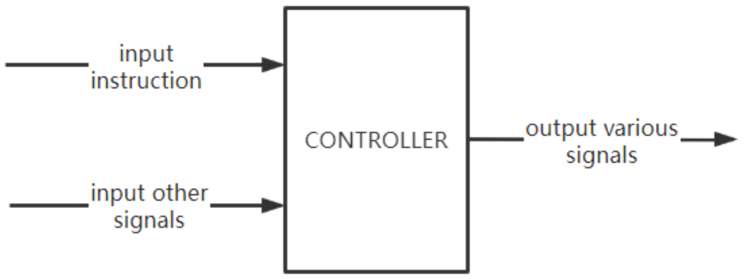
3.1 alu.v

In this file, I have implemented a variety of computing functions of ALU, including add, addu, sub, subu, and, nor, or, xor, sll, sllv, srl, srlv, sra, srav, slt. The input of the component is two 32-bit binary numbers. It judges which function to perform through the ALU_control signal, and finally outputs the result and flags.



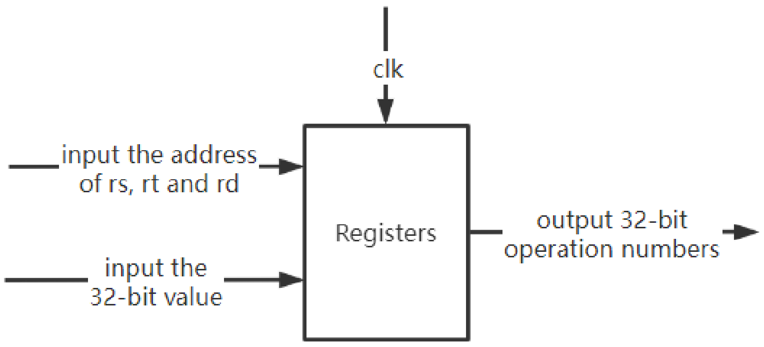
3.2 CONTROLL_UNIT.v

In this element, we need to decide what operations different instructions need to perform at different stages. In addition to inputting 32-bit machine code instructions, we also need to input the signal sent back by the previous instruction in the EXE or MEM cycle to decide whether to adopt the forwarding strategy.



3.3 RegisterFile.v

This file simulates the 32-bit register groups used in the real computer. Simply enter the register address of the operation numbers and we can get the value of the 32-bit operation numbers from it. We can also input the 32-bit value to be stored in the register and store it correctly according to the input address.


clk
input the address of rs, rt and rd
input the 32-bit value
Registers
output 32-bit operation numbers

3.4 InstructionRAM.v

This element is provided by the teacher. Input the address of the instruction, and the instruction machine code of the address can be output.

3.5 MainMemory.v

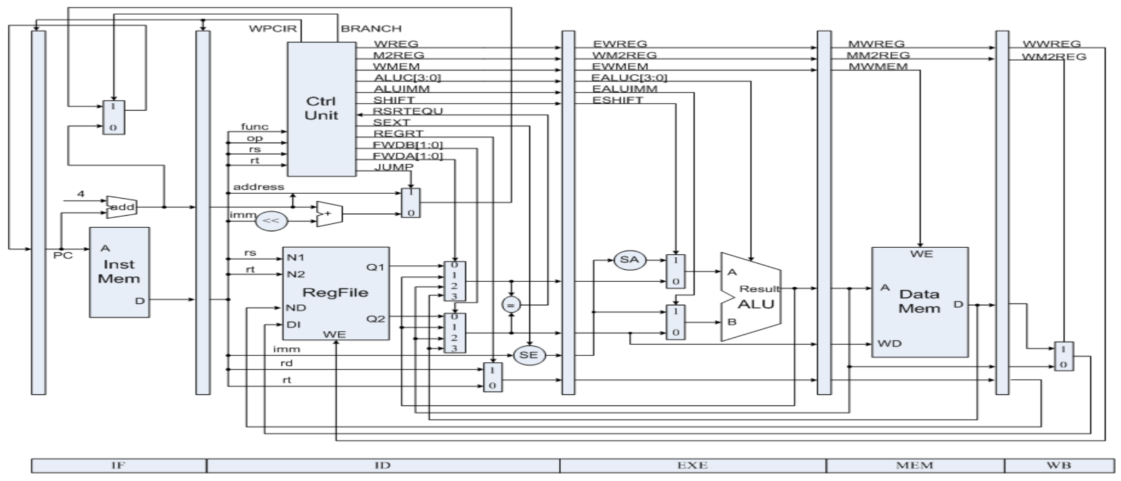
This element is provided by the teacher, too. Input the address of the memory, and output the value stored in the address.

3.6 CPU.v

This is the top-level file that assembles the pipelined CPU. This element combines each of the elements described above and allows them to perform their respective functions. Its input only needs clock signal and reset signal. It realizes the functions of each stage of the 5-stage CPU and the register groups used to save and transfer parameters between stages. The 5-stage is the following:

1) Instruction Fetch (IF)
2) Instruction Decode (ID)
3) Execute (EXE)
4) Memory Access (MEM)
5) Write Back (WB)

Each stage of the 5-stage CPU will occupy one clock cycle. In order to make multiple instructions run in the CPU at the same time, we need to set register groups between each stage to temporarily save the running results of the previous stage. At the same time, the interstage register is also conducive to the implementation of forwarding strategy and stalling strategy to deal with data conflict and control conflict. The top-level structure of my pipelined CPU is roughly shown in the figure below:



4 Hazards

Our CPU implements the forwarding strategy to achieve almost all hazards, but there are also two cases where we must add a stall of one clock cycle to ensure the correct operation of the program. There are two situations when we must use stall:

1) Consider the following two instructions:

lw $t1, 0($t0)	#0
add $t2, $t1, $t0	#1

Similar to this situation: #0 instruction takes data from memory and stores it in $t1 register, but this register will participate in operation in subsequent #1 instructions. Analyzing the pipelined of the two instructions, lw instruction takes out the value and stores it in the register only at the rising edge of the clock in the MEM stage. At this moment, the add instruction runs to the EXE stage and cannot obtain the latest value of the required register. Therefore, we need to stall #1 instruction for a clock cycle, block it in the ID stage and wait for the register value update in the MEM stage.

2) Consider the jump instruction

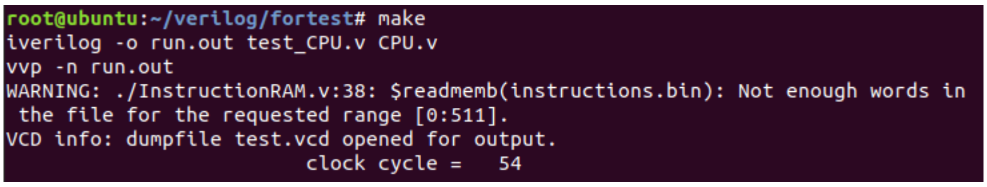
We implement the prediction of no jump in the5-stage pipelined CPU, that is, when the jump conditions of beq and bne are not met, the subsequent instructions do not need to stall and can be executed sequentially. When the jump conditions meet the need to jump, the instruction to jump needs to stall a clock cycle. The specific reasons are as follows: In our 5-stage pipelined CPU, we put the judgment of jump condition in the ID stage of jump instruction, and the calculation of jump address in the EXE stage of jump instruction. Because of the prediction of no jump, the PC value of the next instruction is always the one immediately after beq or bne, and the PC value of the instruction after jump is calculated in the EXE stage. We use the forwarding strategy to send the address to the IF stage, and select whether the PC value is PC+4 or jump address by judging the jump signal in the ID stage of the jump instruction. Therefore, for the instruction to jump, it actually stalls a clock cycle. If the instruction jumps, it can be considered the clock cycle of stall as the IF stage of the next instruction. If the instruction doesn't jump, it just continues to execute from the IF stage of the next instruction. As for the unconditional jump instructions, such as j, jal and jr, the principle of stalling is the same as which of beq or bne when the jump conditions are met, so it will not be repeated here.

5 Testing Results

I test the eight test samples given by the teacher, and get the correct running results of data memory. The following is the number of clock cycles used by the eight test samples:

Test No.	The number of clock cycles	Test No.	The number of clock cycles
1	56	5	179
2	15	6	54
3	18	7	45
4	17	8	28

On the console, after executing the "make" command, we can see the running result as shown in the figure below (This is the running result of test6. The number of clock cycle is displayed and the data memory result is in the data.bin):



```
root@ubuntu:~/verilog/fortest# make
iverilog -o run.out test_CPU.v CPU.v
vvp -n run.out
WARNING: ./InstructionRAM.v:38: $readmemb(instructions.bin): Not enough words in
 the file for the requested range [0:511].
VCD info: dumpfile test.vcd opened for output.
                             clock cycle =    54
```