# CSC3050 Project 2
# Report

Liang Jiarui

May 14, 2021

## 1 Introduction

This project is an implementation of a pipelined MIPS CPU using Verilog. It supports commonly used MIPS instructions, including arithmetic and logic operations, data transfer instructions, conditional branch instructions, and jump instructions. Hazards are avoided by using stalls and forwarding.

## 2 Overview

### 2.1 Pipelined Datapath

The execution of an instruction can be divided into different stages. This processor contains five stages: IF(instruction fetch), ID(instruction decode), EX(execution), MEM(memory access), and WB(write back).

The IF stage fetches the next instruction, and then adds pc by 4. The ID stage obtains the operands of the instruction. The EX stage completes the calculations using ALU (arithmetic and logic unit). The MEM stage reads or writes data from the main memory. The WB stage writes the result of ALU or memory output back to registers.

The processor also contains a control block to generate control signals, such as ALU control, branch, register write, etc.

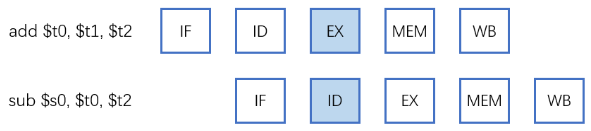
A pipelined datapath is faster than a single-cycle datapath. In a single-cycle datapath, an instruction cannot be executed until the previous one completes all of the processor stages. However, a pipelined datapath allows the processor to execute multiple instructions simultaneously. Each instruction is executed one stage prior to its following stage. For example, when an instruction proceeds to the EX stage, its next instruction is being executed in the ID stage, which improves the efficiency of the processor.

### 2.2 Hazards

Hazards include structural hazards, data hazards, and control hazards.

A structural hazard occurs when the processor tries to use one resource in different ways. In this project, structural hazards do not happen, because the instruction memory and the data memory are separated.

A data hazard occurs when some data are used before they update. For example,



**1.** Data Hazard

The second instruction attempts to access $s0 in the ID stage, but the value of the register is currently calculated in the EX stage.

There are two ways to resolve data hazards. One is using stalls, which means the second instruction wait until the first instruction writes back to the registers in the fifth stage, so three cycles are wasted.

The other way is using forwarding. In this example, the CPU can forward the result of addition from the EX stage to the ID stage. This project uses forwarding to resolve data hazards.

A control hazard occurs to branch instructions and jump instructions. Normally, the branch/jump address is available in the ID stage, and the branch condition is decided in the EX stage, but an instruction has to be fetched in every cycle. This project assumes that the branch is not taken, so its following instructions continued to be fetched. When the branch decision is made, the executing instructions that should be skipped will be discarded. Moreover, the branch decision block is removed to the ID stage, so at most one cycle is wasted for a branch or jump instruction.

## 3 Data Flow Chart

See Figure 2.

## 4 Implementation and Details

The project contains a main file "pipeline.v", several modules, and a test bench "pipeline_test.v". The whole program is divided into different blocks. The implementation is as follows.

### 4.1 Pipeline Module

This is the main module of the program. It combines all of the CPU blocks and controls the input and output of each block. This module also simulates pipeline registers. At the negative clock edge, data are transferred from the previous stage to the next stage.

### 4.2 Instruction_decode Module

MIPS fields are decoded from the instruction, including opcode, rs, rt, imm, and shamt. The write-back value is written to the registers whenever the write-back value or the write-back address changes, usually after the negative edge. Then the values of rs and rt are fetched from the registers at the positive edge. If there is a jump-and-link instruction (jal), the address of the next instruction (pc+4) is saved in $ra.

### 4.3 Control Module

Control signals are generated from the instruction. When control_mux is zero, it means flushing occurs in the control block, and all control signals become zero.

### 4.4 ALU Module

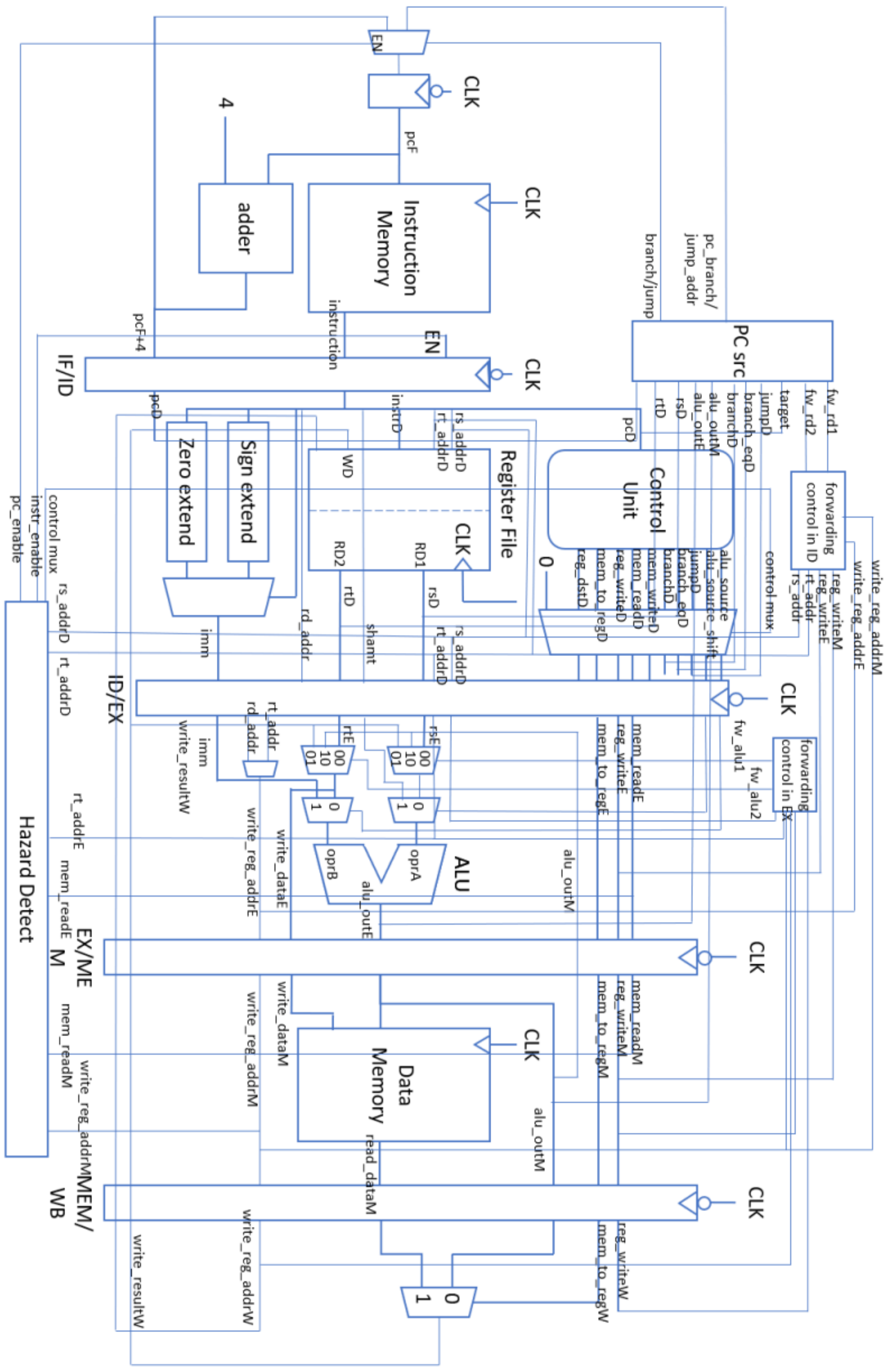
ALU operations are performed based on the ALU control code. Table 1 shows the ALU control codes and their corresponding operations.

### 4.5 Write_back Module

The value to be written back is decided according to mem_to_reg control signal. If mem_to_reg is asserted, the write-back value comes from the data memory; otherwise, the write-back value comes from the ALU result.

### 4.6 ForwardingE Module

This module generates the forwarding signals in the EX stage. This forwarding can resolve a single data hazard. If the address of rd in EX/MEM is the same as the address of rs (or rt) in ID/EX, there is forwarding from EX/MEM pipeline registers; if the address of rd in MEM/WB is the same as the address of rs (or rt) in ID/EX, there is forwarding from MEM/WB registers. The forwarding signal 00 represents no forwarding; 10 represents EX forwarding; 01 represents MEM forwarding. The diagram is shown in Figure 3.

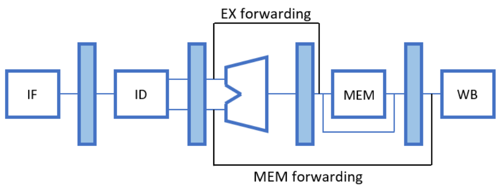


CLK
EN
pcF
4
adder
Instruction Memory
instruction
pcF+4
IF/ID
pcD
instrD
rs_addrD
rt_addrD
Register File
WD
RD1
RD2
Sign extend
Zero extend
PC src
pc_branch/ jump_addr
branch/jump
fw_rd1
fw_rd2
target
jumpD
branch_eqD
branchD
alu_outM
alu_outE
rsD
rtD
forwarding control in ID
Control Unit
alu_source
alu_source_shift
jumpD
branch_eqD
branchD
mem_writeD
mem_readD
reg_writeD
mem_to_regD
reg_dstD
control mux
rs_addr
rt_addr
reg_writeE
reg_writeM
write_reg_addrE
write_reg_addrM
0
rd_addr
shamt
imm
ID/EX
rsE
rtE
00
10
01
0
1
rt_addr
rd_addr
write_resultW
fw_alu1
fw_alu2
forwarding control in EX
mem_readE
reg_writeE
mem_to_regE
ALU
oprA
oprB
alu_outE
alu_outM
write_dataE
write_reg_addrE
EX/MEM
mem_readM
reg_writeM
mem_to_regM
Data Memory
write_dataM
write_reg_addrM
read_dataM
MEM/WB
reg_writeW
mem_to_regW
write_reg_addrW
write_resultW
Hazard Detect
control mux
instr_enable
pc_enable
rs_addrD
rt_addrD
rt_addrE
mem_readE
mem_readM
write_reg_addrM

**2.** Data Flow

| ALU operation | ALU control code |
|---|---|
| add | 0001 |
| sub | 0010 |
| and | 0011 |
| or | 0100 |
| xor | 0101 |
| nor | 0110 |
| slt | 0111 |
| shift left | 1000 |
| shift right (logic) | 1001 |
| shift right (arithmetic) | 1010 |

Table 1: ALU control code

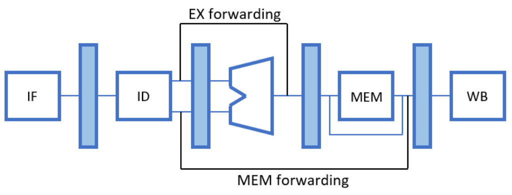


**3.** ALU forwarding

## 4.7 ForwardingD Module

This module deals with control hazards for branch and jump instructions. The branch/jump decision is made in the ID stage. For branch instructions, the program may try to read the values of the operands before they are written back. For jump instructions, if the jump address is stored in register rs, the address may not be written back either.

The ID forwarding signals are similar to ALU forwarding. The implementation is a little different from ALU forwarding, which is shown in Figure 4.



**4.** ID forwarding

## 4.8 Hazard Module

Some hazards cannot be resolved by forwarding. When these hazards occur, stalls and flushing will be inserted into the pipeline.

- lw: If lw is followed by an instruction that tries to fetch the loaded data, the second instruction proceeds to the ID stage, but the lw is still in EX stage, in which the load operation does not occur yet. The pipeline will insert a stall. PC and ID instruction will remain unchanged, and control_mux becomes zero.

- Branch/jump: When a branch/jump decision is made in the ID stage, the instruction in IF stage will be skipped by flushing the IF/ID to zero.

- Both lw and branch/jump hazards: This situation happens when a branch/jump instruction tries to approach the loaded data. Since the branch/jump decision is made in the ID stage, and the loaded register is available in the MEM stage, two stalls are required. In this program, the first stall is detected the same as lw hazard; the second stall is detected additionally.

### 4.9 PCSrc Module

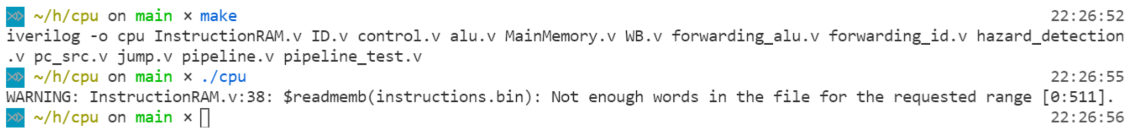
The branch decision is made in this module.

### 4.10 Jump Module

The jump address is computed in this module.

## 5 Run and Test

The source files can be compiled using iverilog. A makefile is included in the source files, so one can use “make” to compile the program. Then an executable file “cpu” is generated. To run the file, type “./cpu” in Terminal. An external file “instructions.bin” with instruction codes should be contained in the path. After execution, the content of the main memory will be saved in an external file “out.txt”.



```
~/h/cpu on main × make                                                                              22:26:52
iverilog -o cpu InstructionRAM.v ID.v control.v alu.v MainMemory.v WB.v forwarding_alu.v forwarding_id.v hazard_detection
.v pc_src.v jump.v pipeline.v pipeline_test.v
~/h/cpu on main × ./cpu                                                                             22:26:55
WARNING: InstructionRAM.v:38: $readmemb(instructions.bin): Not enough words in the file for the requested range [0:511].
~/h/cpu on main ×                                                                                   22:26:56
```

**5.** Run and Test