CPU

Big Picture

This program is aimed to implement 5-stage pipeline CPU using verilog language. The five stages are instruction fetching from memory(IF), instruction decoding and reading(ID), executing operation or calculating address(EX), accessing memory operand(MEM), and write back to registers(WB). We use 4 pipelines, which is responsible for data store and data transpose, to connect them and get the whole CPU. When CPU executes instructions, the five stages operate simultaneously which significantly speeds up the operation. Additionally, we also solve 7 hazards to guarantee 100% accuracy as well as minimizing the operation time. The more details are as follows.

Data Flow

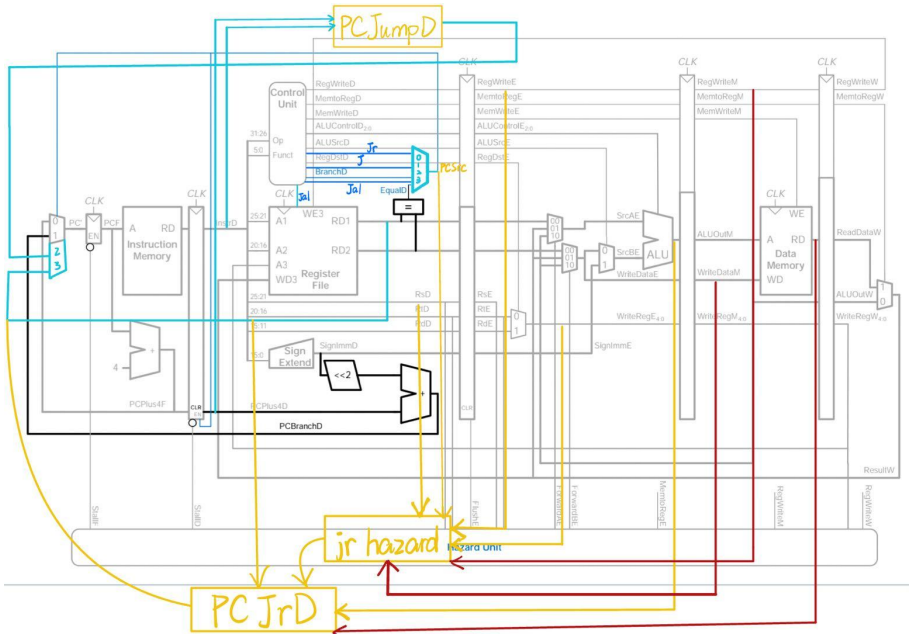

Figure 1: CPU

High Level Implement

1. Whole Program Division

In this project, I break the whole program into 9 modules which are **IF, IFID_Registers, ID, IDEX_Registers, EX, EXMME_Registers, ME, MEWB_Registers,** and **WB**, and solve them one by one. The Registers are 4 pipelines. In the end, I connect them and test. Additionally, after

accomplishing the basic version of CPU, I add some modules and wires to fix seven hazards. They are **MEM/WB_to_EX_hazard, lw_stall_hazard, Register_read_write_hazard, Beq_Bne, J/Jal/Jr, Jr_hazard,** and **Beq_bne_hazard.**

2. Standard CPU

IF I used instruction memory module to fetch instruction and PC address multiplexer to get the right value.

ID I designed control unit module to decode the instruction and register files module to read and write data in 32 regular registers.

EX I implemented a ALU module to execute algorithm and address.

ME I used main memory module to read and write data in the main memory.

WB I used register files module to write data to 32 regular registers.

Pipelines The pipelines are used to store and transpose the data. I utilize same strategy to implement 4 pipelines. Taking IFID_Registers as an example, every positive edge of clock, pipeline will store the data from the IF modules into registers and transpose the data in the registers to ID modules simultaneously.

3. Seven Hazard

MEM/WB_to_EX_hazard This hazard happens when ID module want to read data in some register but register value have not been updated (data in MEM/WB stage). I used forward strategy to solve this problem. That is we add some wires to connect the EXME_Registers/MEWB_Registers and EX module which can transpose the un_updated data to ALU inputs.

lw_stall_hazard This hazard happens when ID module want to read data in some register but the previous instruction is lw and register value have not been updated. I used stall strategy to solve this problem. Firstly I stall instruction in IF, ID modules by re_executing the instruction. Secondly, I change all the control signals to insert a NOP.

Register_read_write_hazard This hazard happens when ID module want to read data in some register but register value have not been updated (data is going to write). I made the program first write the data and then read the data.

Beq_Bne This situation happens when program need to branch, but other instructions are in pipeline. Firstly, I advanced the beq/bne test to ID part. Secondly, when we need to branch, the program will flash the data in the IFID_Register which will create a bubble behind the beq/bne instruction. Thirdly, transpose the branch address from ID modules to IF modules.

J/Jal/Jr This situation happens when program need to jump, but other instructions are in pipeline. This strategy is similar to **Beq_Bne**. Firstly, I used control module to test whether the program needs j/jr/jal. Secondly, when we need to jump, the program will flash the data in the IFID_Register which will create a bubble behind the j/jr/jal instruction. Thirdly, transpose the jump address from ID modules to IF modules.

Jr_hazard This hazard happens when the program need to obtain jump address from some register but register value have not been updated (data in MEM/WB stage). Firstly, we need to test whether it is a Jr_hazard. After that, if there is a hazard, we need to connect EXME_Registers/MEWB_Registers and ID module which can transpose the un_updated data.

beq_bne_hazard This hazard happens when the program need to obtain data from some

register to judge whether we need to branch but register value have not been updated (data in MEM/WB stage). The strategy is similar to **Jr_hazard**. We add a judge unit in the ID module and connect EXME_Registers/MEWB_Registers and ID module.

Implement Details

1. Standard CPU

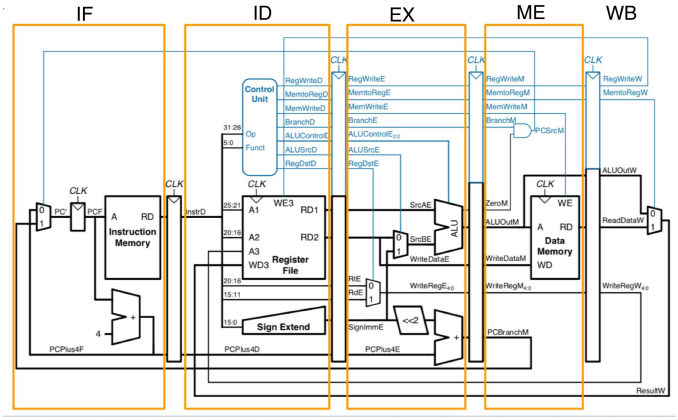

Figure 2: Standard CPU

The above figure shows a standard CPU. Firstly, designed some small modules, such as **Control Unit**, **Register File** and, **ALU** module (Instruction Memory and Main Memory have finished). Secondly, implemented 9 another modules which are **IF, IFID_Registers, ID, IDEX_Registers, EX, EXMME_Registers, ME, MEWB_Registers,** and **WB** module. Finally, assembled them to **Standard CPU**.

Control Unit

For this module, assigned the value of 8 registers according to the opcode and function code. The following two table shows assignments rules. (exclusive of branch and Jump Instructions)

Table 1: Assignment Rule 1

Ins_Name	Reg Dst	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-type	1	0	0	0	0	1	0
I-type	0	1	0	0	0	1	0
lw	0	1	0	1	0	1	1
sw	x	1	0	0	1	0	x

(In Table1: R-type represents add, sub, and, nor, or, xor, sll, sllv, srl, srlv, sra, srav, slt; I-type represents addi, addiu, subu, ori, xori instructions)

Table 2: Assignment Rule 2

	add, addi, addiu, lw, sw	sub, subu	and, andi	nor	or, ori	xor, xori	sll, sllv	srl, srlv	sra, srav	slt
ALUCon	0010	0110	0000	1100	0001	0011	0100	0101	1000	0111

Register File

Initialize 32 registers as 0. When the clock is at positive edge, read the data and write data according to index of registers.

ALU

Calculate data using SrcA and SrcB according to the ALUCon.

IF, ID, EX, ME, WB

Use wires to connect modules.

Pipelines

I utilize same strategy to implement 4 pipelines. Taking IFID_Registers as an example, every positive edge of clock, pipeline will store the data from the IF modules into registers and transpose the data in the registers to ID modules simultaneously.

Standard CPU

Use wires to connect 9 modules.

2. Seven Hazard

MEM/WB_to_EX_hazard

Forwarding Unit

If (EX/ME.RegWrite=1) and (EX/ME.RegRd!=0) and (EX/ME.RegRd=ID/EX.RegRs) ForwardA=10

If (ME/WB.RegWrite=1) and(ME/WB.RegRd!=0)and(ME/WB.RegRd=ID/EX.RegRs)

and not (EX/ME.RegWrite=1) and (EX/ME.RegRd!=0)and(EX/ME.RegRd=ID/EX.RegRs)

ForwardA=01

ForwardB is similar as ForwardA.

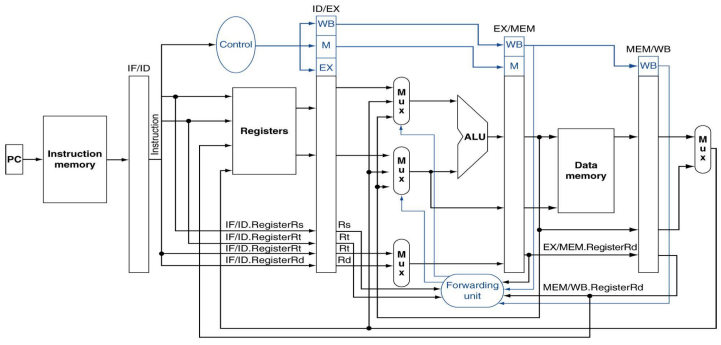

Figure 3: Forward Strategy

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 4: Forward Explanation

lw_stall_hazard

Hazard detection unit

if (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs)or(ID/EX.RegisterRt = IF/ID.registerRt))) lw_hazard = 1

special tricks: When lw_hazard = 1, the program will re_execute instructions in IF and do not change the value of IFID_Registers. Additionally, set output of control unit to 0.

Register_read_write_hazard

We design register file to firstly write and then read data in the registers.

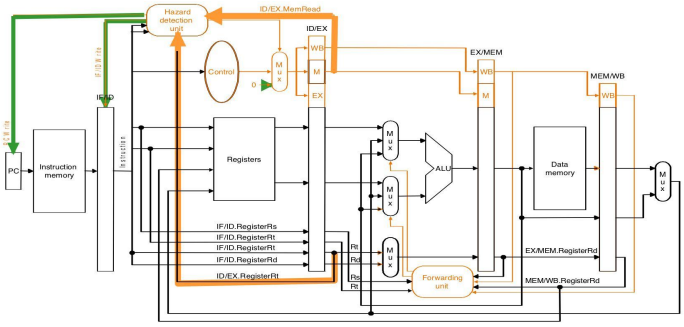

Figure 5: Stall strategy

beq_Bne

If beq/bne, branch=01/10. If equal = 1, PCSrc=1. And then the program will change the PC address to branch address and then flash the data in the IFID_Registers.

special tricks: Flash means we set the value of IFD_Registers to 0 when clock is positive edge.

beq_bne_hazard

if ((RegWriteE==1) & (WriteRegE==rsD) & (BranchD!=0)) beq_hazard = 3'b001, data1=ALUResult;

If ((RegWriteE==1) & (WriteRegE==rtD) & (BranchD!=0)) beq_hazard = 3'b010, data2=ALUResult;

if ((RegWriteE==1) & (WriteRegW==rsD) & (BranchD!=0)) beq_hazard = 3'b001, data1=ReadData;

If ((RegWriteE==1) & (WriteRegW==rtD) & (BranchD!=0)) beq_hazard = 3'b010, data2=ReadData;

(data1 and data2 is used to get value of equal, default they are equal to read1 and read2)

(PCBranch=signed_extended（offset）+GPR[base])

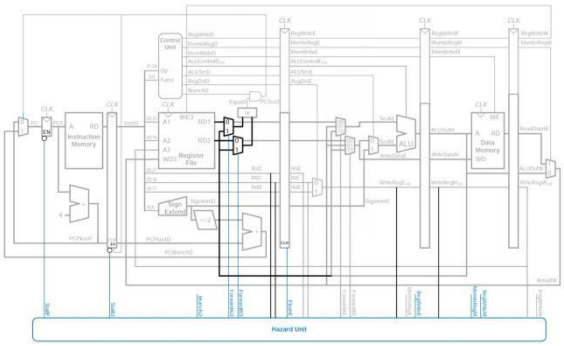

Figure 6: beq_bne & beq_bne_hazard

J/Jal/Jr

Firstly, I use control unit to test, if we need j/jal/jr, then register j/jal/jr = 1. Secondly, I extended the PCSrc in the following way. (jal need to write PCPlus4 to register $ra)

Table 3: PCSrc & PCAddress

Situation	PCSrcD	PC Address
Normal	000	PCPlus4
beq/bne	001	PCBranch
j	010	PCJump
jr	011	PCJr
jal	100	PCJump

(PCJump={(PCPlus4)[31,28]||target||00}, PCJr=read1 by default)

Jr_hazard

if ((JrD==1) & (RegWriteE==1) & (rsD==WriteRegE)) , jr_hazard = 2'b01; (PCJr=ALUReuslt)

if ((JrD==1) & (RegWriteM==1) & (rsD==WriteRegM)), jr_hazard = 2'b10; (PCJr=ReadData)