In this project, the 5-stage Pipelined CPU is implemented in “CPU.v” file. The “Tools.v” file implements some tools that we need for the 5-stage Pipelined CPU, i.e., sign extender, multiplexer for 5-bits data, multiplexer for 32-bits data and clock. The “Units.v” file implements some units that we need for the 5-stage Pipelined CPU, i.e., control unit, register file and ALU. The “Stages.v” file implements the registers among the 5 stages: IF, ID, EX, MEM, WB, i.e., WB_IF, IF_ID, ID_EX, EX_MEM and MEM_WB. The makefile has been finished, users can directly use “make” in the terminal to compile and run this project.

Big Picture Thoughts and Ideas

According to the 5 stages: IF, ID, EX, MEM, WB, we can divide the whole project into 5 parts. For each part, we implement the operations of the Pipelined CPU.

First of all, we should create a CLK for looping instructions.

For the IF part, we implement the PC multiplexer because that the instructions may be branch or jump instructions. After that, we should use RAM to access the instructions’ machine code. Then, we can get the PCF and use it to get each part of the instruction from the Instruction Memory. We can store the instruction and PC+4 into the IF_ID register.

For the ID part, we get the instruction from the IF_ID register, divided it into Op, Funct, A1, A2, Imm parts. Then, we implement control unit to accept Op and Funct, and output RegWriteD, MemtoRegD, MemWriteD, BranchD, ALUControlD, ALUSrcD, ALUSrc_saD and RegDstD. We implement the register file to accept A1, A2, WriteRegW, ResultW, RegWriteW, and output RD1D and RD2D. We implement the sign extender to sign extend the Imm. We store the RegWriteD, MemtoRegD, MemWriteD, BranchD, ALUControlD, ALUSrcD, ALUSrc_saD, RegDstD, RD1D, RD2D, InstrD[20:16], InstrD[15:11], InstrD[10:6], SignImmD, PCplus4D into the ID_EX register.

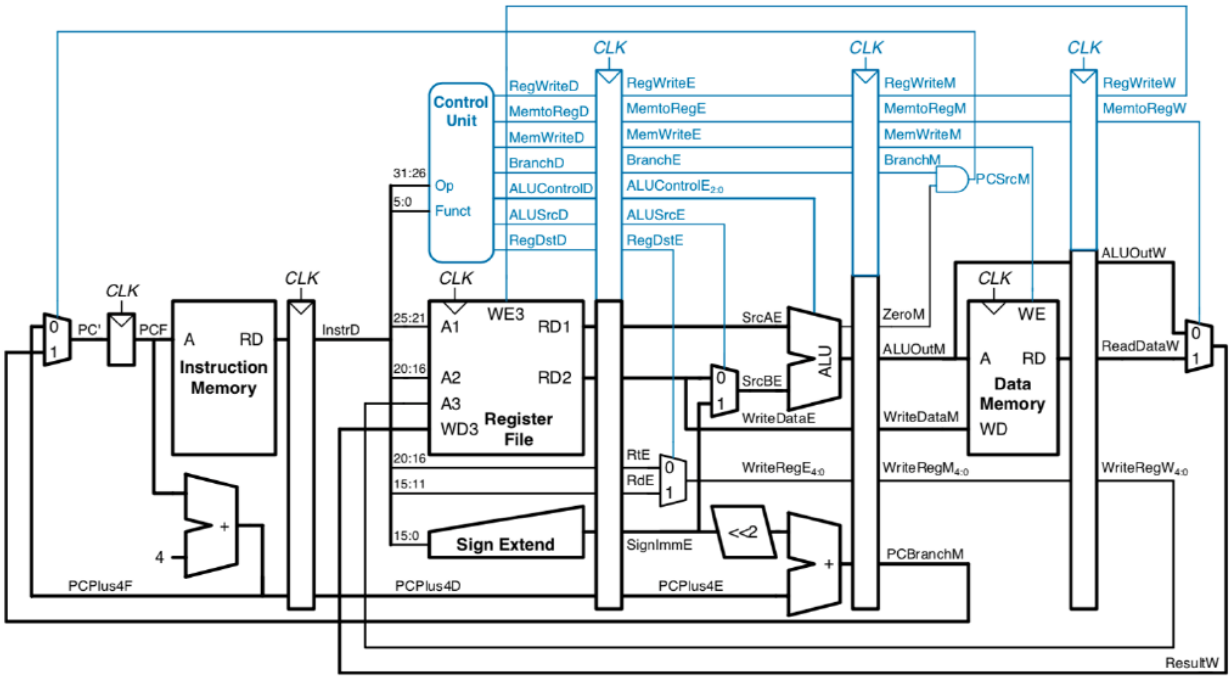
For the EX part, we should implement SrcBE multiplexer, SrcAE multiplexer, WriteRegE multiplexer and the ALU. The SrcBE multiplexer choose SrcBE from RD2E and SignImmE according to ALUSrcE. The SrcAE multiplexer choose SrcAE from

RD1E and shamtE according to ALUSrc_saE. The WriteRegE multiplexer choose WriteRegE from RtE and RdE according to RegDstE. The ALU accept SrcAE and SrcBE to compute the result ALUOutE and zeroE according to the ALUControlE. We store the RegWriteE, MemtoRegE, MemWriteE, BranchE, ALUOutE, zeroE, RD2E, WriteRegE, (SignImmE<<2)+PCplus4E into the EX_MEM register.

For the MEM part, we should implement the Data Memory to deal with data according to MemWriteM, ALUOutM, WriteDataM. And then, store RegWriteM, MemtoRegM, ALUOutM, ReadDataM and WriteRegM into MEM_WB register.

For the WB part, we should implement the ResultW multiplexer. It chooses ResultW from ALUOutW and ReadDataW according to MemtoRegW. The ResultW will enter the register file WD3 port and the WriteRegW will enter the register file A3 port.

Data Flow Chart



The High-Level Implementation Ideas

In this part, I will introduction function of all modules that I implemented.

In “Tools.v”, SignExtend is used for extending a value from 16 bits to 32 bits by repeating the sign bit. MUX5 accept two 5 bits values and a control value, according to the control value, the output will be one of the two 5 bits values. MUX32 accept two 32

bits values and a control value, according to the control value, the output will be one of the two 32 bits values. The module named clock is used for implementing the CLK looping in the CPU.

In “Units.v”, ControlUnit is used for simulating the control unit of the CPU, according to the Op and Funct, it can give the relative RegWriteD, MemtoRegD, MemWriteD, BranchD, ALUControlD, ALUSrcD, ALUSrc_saD, RegDstD. The ALUSrc_saD is added for simplifying the control unit. RegisterFile is used for simulating the register file of the CPU, it can output RD1 and RD2 according A1 and A2, it can also store WD according to A3 and WE. The ALU is used for simulating the ALU of the CPU, according to the ALUControl, it do some operations by using A and B, then it store the results into C and zero, last, it output C and zero.

In “Stages.v”, WB_IF is used for beginning the CPU and connecting WB stage and IF stage. IF_ID is used for simulating the register between IF and ID stages. ID_EX is used for simulating the register between ID and EX stages. EX_MEM is used for simulating the register between EX and MEM stages. MEM_WB is used for simulating the register between MEM and WB stages.

In “CPU.v”, CPU is used for simulating the whole 5-stage Pipelined CPU and display the whole Main Memory in the screen. In “test_CPU.v”, test_CPU is used for creating a CPU instance and run this project.

The Implementation Details

For SignExtend module, we can directly set the output as reg signed [31:0] to implement sign extend. For MUX5 and MUX32 module, if the Control equal to 0, C will equal to A0, else, C equal to A1. For clock module, the first half cycle is #(CYCLE/2), and the second half cycle can get by CLK=~CLK.

For ControlUnit module, we can distinguish the instructions according to Op and Funct. Once we know the specific instruction, we can set the outputs of RegWrite, MemtoReg, MemWrite, Branch, ALUControl, ALUSrc, ALUSrc_sa, RegDst. For RegisterFile module, we create a reg array to store data: reg signed [31:0] data[31:0]; according to A1, A2, we can read data from the array, according to WE, A3, we can write

WD into the array. For ALU module, we match the specific instructions with the different ALUControls: 4'b0000- lw, sw, add, addu, addi, addiu; 4'b0001- sub, subu; 4'b0010-and, andi; 4'b0011- nor; 4'b0100- or, ori; 4'b0101-xor, xori; 4'b0110-sll, sllv; 4'b0111-srl, srlv; 4'b1000-sra, srav; 4'b1001-beq; 4'b1010-bne; 4'b1011-slt; for j, jr, jal instructions, they don't need to enter the ALU, we will implement them in the "CPU.v" later.

For WB_IF, IF_ID, ID_EX, EX_MEM, MEM_WB modules, they just store their own inputs temporarily, so at the beginning, all the outputs will be set as 0, when it comes to posedge CLK and the RESET is not active, all the outputs will be set according to the inputs.

For CPU module, we create all the components' instances in it and manage the branch instructions by zeroM and BranchM. Meanwhile, we manage the jump instructions directly by using the RESET of each stages' registers. Finally, we need to display the whole main memory. When no instruction gets into IF stage, we still need 4 clocks to finish the last instruction completely, so we use an integer final_cycle to record the last 4 clocks. When final_cycle>=5, it means that all the instructions has been finished, so we can display the main memory.