**Modules explanation**

**CPU.v**

Modules implemented last homework had already been explained, so I will skip those. My CPU is divided into five sections, each section is separated by the IFID, IDEX, EXMEM MEMWB pipeline modules respectively. I will list the units in each section and briefly explain their function in this CPU.v module explanation. More details will be in other modules’ explanation.

The first section is the IF section consisted of a MUX32, PC, PC+4 Adder and Instruction memory. MUX is used to decide whether the PC source should be the branch or the pc+4.. PC, Pc+4 Adder, Instruction Memory is written last time, no need to explain again. The IFID is the separator of this section and the next section, the registers that are sent to the next section will be listed in the IFID.v module explanation.

The second section is consisted of Control, Branch pc adder, Shift left unit, Branch decision unit, RS1 data RS2 data comparison unit, Hazard detection unit, Registers and Sign extend. Control unit has some new added signals for branch and memory, but still used to determine whether to do certain operations or not. Branch address adder is used to calculate the branch pc. Shift left is used to shift the 32bit binary branch pc left 1, since the branch pc doesn’t contain the 0 bit originally. RS1 data RS2 data comparison unit will compare the data from rs1 rs2 for the branch decision unit to determine whether beq condition is met or not. Hazard detection unit will decide whether a stall and noop is needed according to the rd and memwrite in the EX stage and the rs1,rs2 in the current ID stage. Registers unit is still the same, and the sign extend unit pretty much works the same, except for some new kinds of instructions to handle. The IDEX is the separator of this section and the next section, the registers that are sent to the next section will be listed in the IDEX.v module explanation.

The third section is consisted of 2 MUX\_Forward, a MUX32, ALU, ALU control and Forwarding unit. Mux\_Forward is used to choose whether the source of ALU is forwarded or the original data. MUX32 works the same as last time, so does ALU and ALU comtrol. The EXMEM is the separator of this section and the next section, the registers that are sent to the next section will be listed in the EXMEM.v module explanation.

The fourth section is consisted of data memory which is added this time for lw sw. It accepts signals to decide whether to read data out, write data in or do nothing. The MEMWB is the separator of this section and the next section, the registers that are sent to the next section will be listed in the MEMWB.v module explanation.

The last section is consisted of a MUX32 which chooses whether to write back the alu result or the data from memory according to the MemtoReg signal.

The wires of my CPU are mostly the same with the final path figure provided by the TA. I only added five wires additionally, the first one connects Instruction[6:0] to Sign extend unit, to determine what kind of extension should be done according to different instructions. The second one connects Instruction[6:0] to IDEX pipeline, which enables the third wire to connect it with the ALU control, which I also need to determine the ALU operation to be done according to different instructions. The fourth one is a funct3 wire separated from the funct7 wire which were combined in TA’s final data path. The fifth one is another funct3 wire which connects the IDEX pipeline to ALU control.

**Adder.v**

This module is implemented in the last homework. However, in this project, there are a new adder is added, which uses this module directly.

The branch pc adder gets input branch pc shift\_out from the shift left unit and the input current pc IDPC\_current passed from the IFID pipeline. The output is sent to the MUX in IF section.

**Control.v**

Four more signals are added, which are MemtoReg\_o, MemRead\_o, Memwrite\_o for lw and sw, and branch\_inst\_o for beq. Moreover, if the stall condition is met in the Hazard detection unit, the input noop\_i will be 1 and all control signals in the module will be set to zero.

**Sign\_Extend.v**

Three more kinds of immediate input, lw, sw, beq, are extended according to their immediate bits.

**MUX32.v**

This module is implemented in the last homework. However, in this project, there are 2 new MUX32 added, which uses this module directly.

The first one is the MUX in the IF section, used to decide whether to take the branch address branch\_PC or the original pc+4 address PC\_plus4. The select\_i is the branch decision signal decision\_out from the branch\_decision unit. If the signal is 1, the branch address will be chosen, otherwise pc+4 will be chosen. Then it output the address to PC.

The second one is the MUX in WB section, the input data are the ALU result WBalu\_out and the data from memory WBmem\_out passed from the MEMWB pipeline. The output is chosen according to the signal WBMemtoReg also passed from the MEMWB pipeline, if it is 1, the memory data is chosen, otherwise the ALU result is chosen. The output will be sent back to the RDdata\_i in the register unit.

**ALU\_Control.v**

Added more condition to determine if it is lw or sw. Also, the instruction\_3 provided by the TA included “or” (the instruction wasn’t required in the project\_1 slide ) , so I added an “or” condition just in case.

**ALU.v**

Lw and sw also uses add operation, so nothing was added. However, just as mentioned in ALU\_control.v, I added an “or” operation.

**Branch\_decision.v**

The inputs are signals from Control.v as branch\_inst\_in and Comparison\_unit.v

As comparison\_in to determine whether the branch is taken. If both are 1, then the branch is taken, else the branch is not taken. The output is sent to MUX for pc source and IFID.v to determine whether to activate flush.

**Comparison\_unit.v**

Comparison\_unit takes input rs1\_data\_in, rs2\_data\_in from the rs1 data output, rs2 data output from register.v, respectively. If two data are equal, then the output is 1, else is 0. The output is sent to Branch\_decision unit as a signal.

**Shift\_left.v**

Shift\_left input shift\_in is the output of sign extend, which is a 32bit immediate and will be shifted left 1 bit for further use of branch pc. The output is sent to the adder used to calculate the branch PC.

**Fowarding\_control.v**

Forwarding takes MEM stage’s MEM\_RD, WB stage’s MEM\_WB and rs1 EX\_rs1, rs2 EX\_rs2 in current stage as inputs and also MEM\_RegWrite, WB\_RegWrite as input signals. Six wires are created in this module, the functions of these wires are to compare MEM\_RD with EX\_rs1 and EX\_rs2, compare WB\_RD with EX\_rs1 and EX\_rs2, and to check the value of MEM\_RegWrite, WB\_RegWrite. Then these six wires are use as conditions in the if-else statement to determine the output FowardA and FowardB. The details of the conditions and the outputs are already stated clearly in the TA’s slide or the textbook, so I think explanation are not necessary. Last, the output will be sent to the MUX\_Forward as a signal to decide which source should be passed on.

**MUX\_Foward.v**

MUX\_Foward is another kind of MUX which takes three sources WB data in01, MEM ALU result in10 and rs1 or rs2 data as input. Then decides the output according to input signal control fro forwarding unit. Then the chosen data will be output to ALU and a MUX32.

**Hazard\_detection.v**

Hazard\_detection module takes EX stage’s RD EX\_RD\_in and MemRead MemRead\_in, and rs1 rs1\_in, rs2 rs2\_in in the current stage as inputs. Two wires are created in this module, the functions of these wires are to compare RD with rs1 and with rs2. The two wires are used as conditions along with signal MemRead in the if-else statement to determine the output. If MemRead\_in is 1 and either one of the wires are 1, then it means the instruction’s rs1 or rs2 is the same with previous lw instruction’s rd, therefore a stall is needed, so the output stall\_out = 1, also the PC shouldn’t be written, so the PCWrite\_out = 0 and the stages after this should do no operation, so noop\_out = 1. Else, the pipeline should go normally, which is stall\_out = 0, PCWrite\_out = 1, noop\_out = 0. PCWrite\_out will be sent to PC module, noop\_out will be sent to Control module and stall\_out will be sent to IFID pipeline.

**IFID.v**

The IFID pipeline passes two register to the next stage, which are instruction IFID\_instr\_i and current PC PC\_current\_i. However, there is two input signals stall\_i and flush\_i which will effect the pipeline. The pipeline will only pass the registers to next stage if stall\_i = 0, which means the stall condition isn’t met. If flush\_i = 1, which means the branch is taken, then a flush operation should be done, so all of the registers in the pipeline should be set to zero.

**IDEX.v**

The IDEX pipeline passes all six signals from control module to EX stage, except the branch signal, which is handled in the ID stage. The data from rs1,rs2, the extended immediate and the rd[11:7] are also passed to the EX stage. The instruction divided into Op[6:0], funct3[14:12], funct7[31:25], rs1[19:15], rs2[24:20] are all passed to the EX stage for further use.

**EXMEM.v**

The EXMEM pipeline passes four signals, RegWrite, MemtoReg, MemRead, MemWrite to the MEM stage. RD is also passed to the next stage, because it needs to be used in write back at the last stage. The ALU output is passed and the output from MUX\_Forward MUX\_rs2\_data is passed as well, because it may be the data that needs to be written into the memory in sw instructions.

**MEMWB.v**

In MEMWB pipeline, the signals regarding write back is passed to the WB stage, which are RegWrite and MemtoReg. Of course, the RD which is the write back address is also passed.

**Difficulties Encountered and Solutions in This Project**

1. Difficulties: The overall path is to complicated.

Solutions: I break down the implementation into four steps according to the four data path figures provided by the TA. It became easier and my thoughts became clearer when the path is simplified. Also, I would spend some time to think how the data should be passed and why before implementing in Verilog.

1. Difficulties: After compiling and testing, there were a lot of mistakes between my output file and the output file provided by the TA. However, there are too many modules, wires and registers to check, so it was really hard to debug.

Solutions: I decided to use the gtkwave to debug. It turns out that this thing is amazing, because I can trace the whole dataflow and visualize it through the timeline. I use the PC value as timeline and comparing all the signals that I doubt would have gone wrong, this method helped me find out which step went wrong or which signal wasn’t set properly. Then I can go back to the code and check the specific part, which often turned out to be some minor mistake that is really hard to discover when I look through the whole program code. Also, the data format function in gtkwave also helped a lot, because I don’t need to calculate the values between binary and decimals.

**Development environment**

My OS is win10, and compiler is iverilog. My IDE is vscode and I compiled under the

powershell terminal

**\*I changed the reg in data\_memory.v to signed to get the right result for the new test data.**