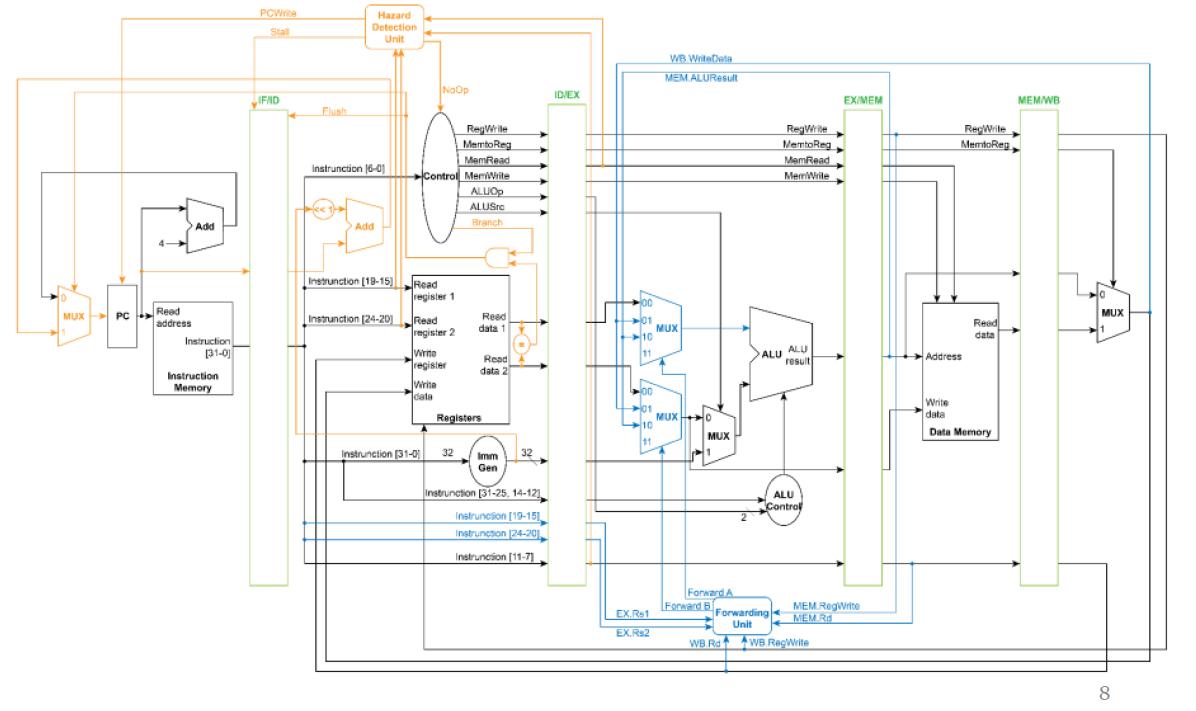
Report for CA Project1 b06611048

1. Modules Explanation



All of the modules are from the picture above, which means that every element in the picture is coded as a module, including AND gate, Equality test operator, and ShiftLeft1-operator, and no additional modules is added in to the CPU.

* 1. PC.v: PC module reads clock signals, reset bit, start bit, PCWrite and next cycle PC as input. This module changes its internal register “pc\_o” at the positive edge of the clock signal. When the reset signal is set, PC is reset to 0. And PC will only be updated by next PC when the start bit is on. Moreover, if the input signal from HDU, PCWrite is disserted, PC’s value cannot be modified.
  2. MUX32.v: There are three MUX32 in the CPU. The first is the one control the PC input. It reads the select bit from the AND gate, then either outputs PC+4 or the branch address from the Adder that calculates the branch address. Another MUX32 controls the second input of the ALU. It reads the signal bit from IDEX reg, and then outputs data from either ImmGen or MUX4 accordingly. The last MUX32 is in the WB stage, controlling to send data from data memory or ALU’s result. It read the signal bit from MEMWB reg to control which data to send back to the registers’ data input.
  3. Adder.v: There are two Adder in the CPU. The first is the one takes the value of PC and then send PC+4 right off the bat to the MUX32 of PC input. The number 4 is hardwired in CPU.v. The other one is used when performing branch instruction, takes the output of ShiftLeft1 and PC from IFID reg as inputs, and outputs the addition of these two back to the MUX32 of PC input.
  4. ShiftLeft1.v: Shift left the input that is the immediate from the ImmGen, and outputs the actual branch offset.
  5. AND.v: Takes Branch bit signal from Control and Equal signal bit from Equal.v, outputs the bitwise and result of these two signals.
  6. Equal.v: Takes both of the data outputs from register file, RS1data and RS2data, as inputs, and then compare them. If they are equal, assert the bit output, and vice versa.
  7. Control.v: Control takes two inputs. The input “Op\_i”, which is the Opcode of the instruction from the IFID reg, and outputs “ALUOp\_o”, “ALUSrc\_o”, and “RegWrite\_o”, “MemtoReg”, ”MemWrite” to IDEX reg, and “Branch” to AND gate, according to the opcode. These are all 1-bit except for “ALUOp\_o”. Moreover, an additional NoOp bit signal is inputed, and if asserted, all of the outputs of Control will send out disserted signal.
  8. Sign\_Extend.v (ImmGen): This module is coded to have five output modes, which are ADDI, SRAI, LW, SW, BEQ (Rtype not included since they won’t use ImmGen). It reads all the instruction code from IFID reg initially. Then it determine which mode is it according to funct3 field + opcode. In each mode the output will differ, but all of them are sign-extended. ADDI mode sign-extends the [31:20] bits, SRAI sign-extends the [24:20], LW works just as ADDI mode, SW sign-extends the {[31:25], [11:7]}, and BEQ sign-extends the {[31], [7], [30:25], [11:8]} bits, all from instruction code and all sign-extends to 32-bit.
  9. HDU.v (Hazard Detection Unit): It reads four signals, which are RS1addr, RS2addr from IFID, plus MemRead, RDaddr from IDEX. Basically it detect ”load-use hazard”. If the preceding instruction is “load” while the next instruction’s r1 or r2 is the rd of load, the HDU will assert Stall to IFID reg, NoOP to Control, and dissert PCWrite to PC. Else, it would be the other way around.
  10. FU.v (Forwarding Unit): It reads 6 signals, which are RS1addr and RS2addr ,from IDEX reg, RegWrite and RDaddr from EXMEM reg, and RDaddr and RegWrite from MEMWB reg. The two 2-bit outputs go to the two MUX4 in the CPU and is regularly 2’b0. But when EX hazard happens, the Fw signals will control the mux to input the right data, and for MEM hazard happens as well. Noted that if both cases happen, only EX hazard will be considered.
  11. ALU\_Control.v: ALU\_Control.v: ALU\_Control has two inputs and one output. The first input, “funct\_i”, which is 10-bit, is the [31:25, 14:12] (funct7, funct3) bits from the IDEX reg. The second input, “ALUOp\_i”, is from the IDEX reg, which is 2-bit. In ALU\_Control, we first split the “funct\_i” signal into “funct7”, “funct3” with the help of wire. The module will first consider the “ALUOp\_i” signal. If “ALUOp\_i” is 2’b00, then we know the instruction is I-type, and if it isn’t, we know it’s R-type. Next, if it is I-type, we check “funct3”. If “funct3” is 3'b101, we set the output, “ALUCtrl\_o”, to perform SRAI. Otherwise, if “funct3” is 3'b000, “ALUCtrl\_o” will be set to perform ADDI for addi, lw, sw instruction. Moreover, if the instruction is R-type, we’ll check “funct7”. If “funct7” is all 0, we know it’s among AND, XOR, SLL, ADD. Otherwise, it’s SUB or MUL. If it’s among AND, XOR, SLL, and ADD, we check “funct3” to know which one is it. If it’s SUB or MUL, we check “funct7” to know which one is it. “ALUCtrl\_o” will send a 3-bit signal to the ALU every time “funct\_i”, “ALUOp\_i”, “funct7”, or “funct3” is updated.
  12. ALU.v: ALU has two 32-bit data input, “data1\_i”and “data2\_i”. “data1\_i” is from the MUX4 of RS1data. “data2\_i” is from the MUX32 module. Additionally, ALU has a 3-bit input, “ALUCtrl\_i”, which is sent by ALU\_Control to control which operation ALU should perform. With these three inputs, ALU will perform arithmetic or logical operations determined by “ALUCtrl\_i” on “data1\_i” and “data2\_i”, and output result whenever “ALUCtrl\_i”, “data1\_i”, or “data2\_i” is updated. The result will be output as “data\_o”. There is also another output “Zero\_o”, but it is not used and implemented in this homework.
  13. MUX4.v: Although this module can fit in four inputs and a control signal, we only use three input ports and a control signal in this CPU. MUX4 will output signal from these three input signals according to the 2-bit control signal.
  14. IFID.v: This pipeline register, with 4 control input, which are start\_i, clk\_i, Flush, Stall. The value of the regs inside can only be change if start\_i is asserted and clk\_i is in posedge. When Flush is asserted, the instruction and PC output will be modified to all zero. When Stall is asserted, the instruction and PC will hold the same value. Otherwise, the outputs, instruction and PC, will be assigned to the value of the next cycle.
  15. IDEX.v: This pipeline register has 2 control signal, start\_i and clk\_i , and can only be write if start\_i is on and clk\_i is on posedge. It has 13 input/output pairs. Each of the input in pairs would assign to the corresponding output when clk\_i is on posedge.
  16. EXMEM.v: This pipeline register has 2 control signal, start\_i and clk\_i , and can only be write if start\_i is on and clk\_i is on posedge.can only be write if start\_i is on and clk\_i is on posedge. It has 7 input/output pairs. Each of the input in pairs would assign to the corresponding output when clk\_i is on posedge.
  17. MEMWB.v: This pipeline register has 2 control signal, start\_i and clk\_i , and can only be write if start\_i is on and clk\_i is on posedge.can only be write if start\_i is on and clk\_i is on posedge. It has 4 input/output pairs. Each of the input in pairs would assign to the corresponding output when clk\_i is on posedge.
  18. CPU.v: Connect all of the modules according to the picture. CPU.v has three inputs, which are start\_i, clk\_i, and rst\_i. Notice that in CPU only Memories, PC, register file, and pipeline registers are sequential elements, so they takes start\_i and clk\_i as inputs.

1. Difficulties Encountered and Solutions in This Project

* Unknown signals [xxxx] appears in the first try even if I initialize every registers in the CPU to 0. Found out that it was due to the value assign in the first clk posedge in pipeline registers. At first, my pipeline regs can change value regardless of start\_i signal whenever clk\_i is posedge. This results in that at the start of the simulation, the pipeline regs would always read in unknown values from the inputs and thus overwrites the 0 in the output regs. After I made pipeline regs takes start\_i as input, the problems was solved.
* SRAI did not output the correct value when operating on a negative value. After a quick google, I changed all the involved regs and wires to signed, and the problem was solved.

1. Development Environment

* OS: Windows 10
* Compiler: iverilog
* Text editing: VS Code
* Debugging using GTKWave