Joshua Chen

Computer Architecture

2025 May 13

*Lab 2 Report*

Adder.v:

The adder takes inputs a and b and adds them together. It is used several times. For example, updating the PC, for both branch and non-branch instructions.

ALU\_Control.v:

The ALU Control module takes in three inputs, all through the ID/EX module. These include the funct7 [31:25] and funct3 [14:12] bits, along with the ALUop control signal. It outputs a signal to the ALU that tells it which operation to use. It does this by just going through if-else statements for each possible combination of funct3 and funct7 bits to output the correct operation. However, if the control signal is b00, this means that a memory access instruction is being run, which outputs add, since the CPU needs to calculate the offset value. If the control signal is b01, this means that a branch function was called, which results in the subtraction bits being sent, because for “beq”, we need to check if the registers have equal value. The easiest way is to just see if rs1 – rs2 = 0.

ALU.v:

The ALU does the actual calculating of values. It takes two inputs that the operation needs to be done to, and one control value input from the ALU control. I just needed to have if-else statements that did the operation that was instructed based on the control value. There’s only one output that is the result of the operation.

Control.v:

The control module takes in two inputs: the opcode of the instruction and a nop signal that is needed for stalls. If the nop signal = 1, all control signals outputted are set to 0, which doesn’t allow any operations to happen for that clock cycle. Else, there is a switch statement and sets the control signals to different values based on the type of instruction. It’s split into R, I, load, store, and branch instructions. These signals are sent throughout the CPU to their respective destinations.

CPU.v:

This is the top-level module that connects everything together. It only takes in clock and reset as inputs, while outputting the values that are needed in the testbench to print out the outputs. These include the value of the PC, whether there is a stall, and whether there is a flush. Each clock cycle, these outputs are set at the bottom of the CPU.v file. Much of the rest of the file is just instantiating wires and connecting the different modules together, letting the programming logic within each module dictate when to send what to whatever module. There is one exception, however. For example, for the beq instruction, I need to check if the two source registers are equal, and I need to have a wire that receives a high signal if branch = 1 and the equality of the source registers is 1.

EX\_MEM.v:

One of the modules that implement part of the pipeline. More specifically, the one in between the execute and memory stages. It doesn’t really do much except pass the signals and data to their next destination whenever the clock signal goes from 0 to 1. It also handles the case where the reset signal is equal to 0, which then sets all output signals to 0.

Forwarding\_Unit.v:

Resolves data hazards that happen when an instruction is trying to use a non-up-to-date register value. As inputs, the forwarding unit takes the two source register addresses from the ID/EX pipeline stage and the destination register from the EX/MEM pipeline stage. Additionally, the forwarding unit also takes the RegWrite signals that are stored in EX/MEM and MEM/WB to make sure that it only forwards when it needs to. For example, for something like a branch or store instruction that never needs to write back to a register, forwarding is not needed. The forwarding unit module has two outputs: Forward A and Forward B, which are two-bit signals that are sent to multiplexors that are placed before going into the ALU. There are three possibilities. First is that there is no data hazard, and the instruction can proceed as per usual. The second possibility is that there is a data hazard with the instruction directly before the current instruction in the execute stage. In this case, a signal will be sent to inform the multiplexors to use the values forwarded from the memory stage. Finally, there is the case where data is in the final writeback stage, but hasn’t been written back yet, so the multiplexor then receives the signal that tells it to use data from the writeback stage.

Within the module, to detect whether there has been a data hazard between the EX/MEM and ID/EX pipeline stages, we check to see if one of the current source registers are the same as the return register of the previous instruction, as well as if the RegWrite signal from the EX/MEM module is high. If so, forwarding happens and a value is sent to one of the multiplexors depending on which source register was the same as the previous instruction’s return register. To check for data hazards between the ID/EX and MEM/WB stages, it is basically the same as the previous conditional for the EX/MEM data hazard, except we add one more condition: For each source register, if there was already a data hazard between the EX/MEM and ID/EX stages, then do not forward the value from the MEM/WB stage, because you will have already forwarded the value from the EX/MEM stage, and that is the correct signal to send. This also means that in the previous clock cycle, there was another data hazard that had already been resolved by forwarding, so grabbing data from the previous instruction in the next pipeline stage will work.  
  
Hazard\_Detection.v:

The hazard detection module takes in 4 inputs: the memory read signal from the ID/EX stage, the two source registers from the IF/ID stage and the destination register of the previous instruction, located in the ID/EX stage. I then check for a load-use hazard by checking to see if either of the source registers is the same as the previous instruction’s destination register, and I also check to see if the memory read signal is high, because that means that a lw function is called. If these conditions are met, the stall and nop signals are turned on and pcwrite is turned off. This results in a stall nop being run for that clock cycle for the lw to have time to write to get to a point where it can be forwarded back to the execution stage. If the conditions are not met, then PC write’s signal is set to one, while the stall and nop signals are set to 0.

ID\_EX.v:

The ID/EX stage has a huge number of inputs and outputs. Most of them are control signals from the Control Module, as well as bit sections of instructions that are split into different wires, that end up going to different places, such as the forwarding unit, ALU control, etc. Implementation was very simple. Just setting all values to 0 if reset = 0, and else, just pass them onto their respective output wires.

IF\_ID.v:

The implementation of the IF/ID is very similar to how the ID/EX was implemented. However, with much fewer input values, it’s much shorter. I made a conditional that sets instruction\_output to 0 and pc\_out to 0 if reset = 0 or flush = 1. If there is no stall, I just pass on the inputs to their respective output wires. Else, this means that stall = 1, meaning that I just do nothing this clock cycle and wait until the next one to pass values on.

Imm\_Gen.v:

This imm\_gen module was a bit more complex than the one from the previous lab, mostly because of the addition of required instructions to implement. These include lw, sw, and beq. Previously, I just had to check whether an instruction was R or I type and then differentiate addi and srai due to their extension bits. This time, there are three more distinct opcodes that had to be included. One that was very strange to do was beq, which has its immediate bits all over the place. So after sign extending, you almost have to scavenge around the instruction to get it in the correct order.

MEM\_WB.v:

The MEM/WB module just like the other pipeline modules. If ~rst, set all the outputs to 0. Else, just pass the five inputs (not including clock or reset) through to their destination wires.

MUX2to1.v:

Takes in two inputs and outputs one of those based on a 0 or 1 signal that is sent from the control unit. There is one that decides the input for changing PC (for branch or non-branch), one that decides between R-type input and I-type, and one that decides between ALU result or data memory.

MUX4to1.v:

Takes in four inputs and a signal and depending on which two-bit signal is inputted, chooses the one of the four inputs as the output. This module is used twice in the CPU implementation: both are used to decide the inputs for the ALU. Though this is a 4 to 1 multiplexor, we only have three inputs. The register values from the ID/EX stage, the forwarded values from EX/MEM, and the forwarded values from MEM/WB. The signal is from the forwarding unit.

Development Environment: VS Code, GTKwave, Verilog

Difficulties/Solutions:

I had a lot of difficulties with this. Firstly, the high-level CPU.v file was quite annoying and time consuming to get right. The large number of wires that connected modules with other far-away modules made it very easy to make an error that is difficult to find. The best solution I could come up with this was to use the provided pipeline CPU diagram and trace the lines that I had finished connecting. Also, I previously would try and define and instantiate wires close to the modules that would be connected to it, but with a pipeline system where forwarding and branching operations need to connect across the CPU, the best option for me was to just define everything at the top of the document.

Another big difficulty for me was misinterpreting what the reset value meant. I thought that reset = 1 meant that the CPU was resetting all signals to 0, and that reset = 0 meant that it was running regularly. On the previous lab, none of my modules used the reset control signal at all except for my CPU.v, which I’m not even sure used the signal. This time, however, on all the pipeline modules in between the five stages, they all made use of the reset signals. This meant that I had if(reset) instead of if(~reset), which was resulting in nothing passing through the pipeline at all. It was an easy fix after the TAs helping me out, but was hard to find the problem initially. Another related problem that I had was that I was using “posedge reset” in my “always” statements, which differed from the given PC.v file that used “negedge reset.” Fixing this made everything work out thankfully.