William Armstrong

Michael Than

Dominic Guo

Alisar Barakat

Fall 2020

CS 3650-MIPS Pipeline Processor

This project is a collaboration between 4 students to create a MIPS processor with pipeline architecture along with a forwarding unit. The MIPS pipeline processor is a 32-bit processor that uses a program counter, instruction memory unit, arithmetic logic unit (ALU), data memory unit, and a few multiplexers to go through the data pipelining. The MIPS pipeline processor is a MIPS processor but it uses the pipeline stages of instruction fetching, instruction decoding, execution, memory access, and memory write back. The MIPS pipeline processor will go through those 5 stages for every line of instruction. The issue with the MIPS pipeline is that it could cause stalls from control, data, or structural hazards. The way to resolve this conflict is to add a forwarding unit. This unit will forward memory from after the current execution stage ends, to the registers so that the next line of code could be executed if that line of code needed any value of a register from the previous instruction. The reason we want to add this hazard unit is to limit as many stalls as possible to optimize the MIPS pipeline processor. Without the hazard unit, the MIPS pipeline would have a lot of stalls because of all the instructions that might create data hazards, and from those hazards would create control or structural hazards that we need to take care off. So our MIPS processor would have all the necessary units needed in a pipeline architecture with all five of the stage cycles, included with a hazard unit that forwards memory. This project will be implemented and shown with VHDL code.

Control and Instruction Decoding Logic

The diagram will be referenced to explain the stages of the pipeline that our processor goes through. The first part of our MIPS pipeline will go to the instruction memory and take in the output from the program counter and pull the instruction out to start the instruction fetch cycle. The instruction that is pulled depends on where the program counter points to. Then from the instruction memory unit, it outputs the 32-bit instruction. Then this value will be sent to the instruction decode stage through registers. Then to decode the instruction, it will be divided into the parts that are needed for its type of instruction. It will send the opcode and funct to the control unit. Send bits 25-21, 20-16 to the register file and send the rest of the bits to registers to be sent to the next stage which is execution. The control unit will take in those inputs and determine what control signals need to be sent out. All those signals come from the type of opcode that is from the instruction. The truth table we have shows all the values of the signals of the different types of instructions we have. It will send signal outputs of register destination, jump, branch, memory read, memory to register, ALU operation, ALU source, and register write. All those values will also be sent to registers to be sent to the execution stage. The register file will take in those values from the instruction memory and keep them in its registers to be later used. The register file also holds values that are written. The register file will then output the data that is needed from the registers to be sent to the ALU unit in the execution stage. All the values needed for the next stage, which is the execution stage will be sent through registers on the clock rising edge. That will set up the execution stage of the pipeline.

Execution

In the execution stage, the processor will take any data from the read data 1, read data 2 registers in the register file, any data from the sign-extender unit, and possibly input from a mux, all depending on which operation is being executed. Then, this data is passed to the ALU. The ALU will then perform the operation and calculate the target address, and then continue to the memory access stage. The target address will be stored in the data memory where it will either be stored as an address to read from, or as an address to write to, again depending on what operation was executed. This takes us to the writeback stage of the pipeline.

Forwarding

The forwarding unit provides two types of forwarding. The first type of forwarding checks the register write address in the MEM and WB phases. If it is possible to forward the appropriate value directly to the ALU it will do so. This is done by sending a 2-bit value to a 3-way multiplex indicating which source to use for each operand of the ALU. The second type of forwarding is used for the comparison performed in branch operations. If either of the two register values needed for a comparison are available in the MEM phase, they can be forwarded directly to the comparison, rather than needing to load them from the actual register.

Hazard Unit

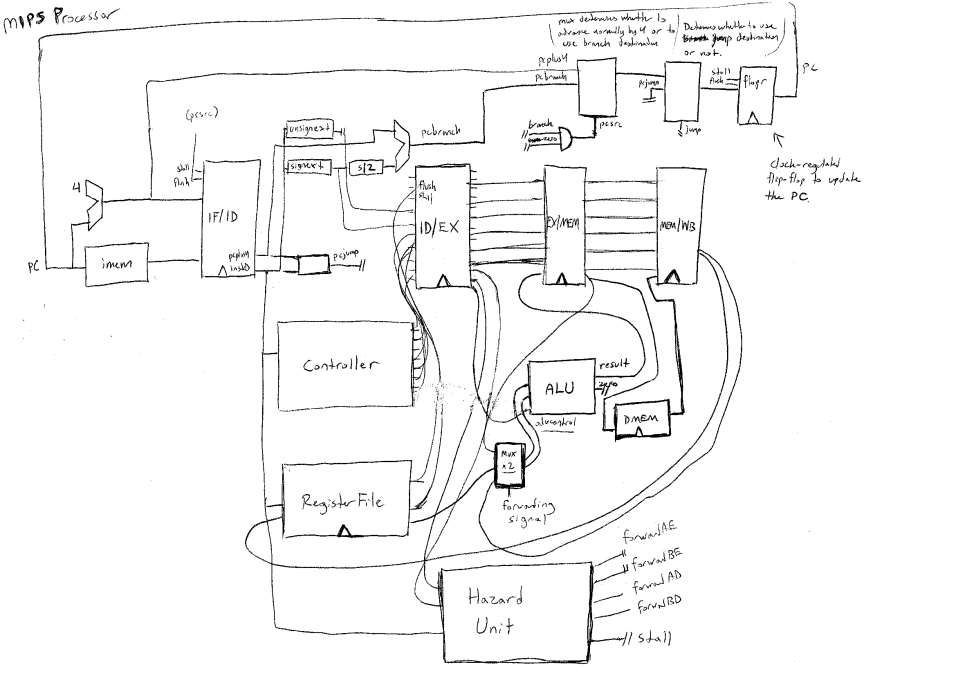
The hazard unit checks for two types of stall: a hardware stall (conflict between the ID/EX phases) and a branch stall (conflict between the ID, EX and MEM phases). If one or both of these stalls are detected, a signal is sent which delays the IF and ID phases by one clock cycle.

Memory Writeback

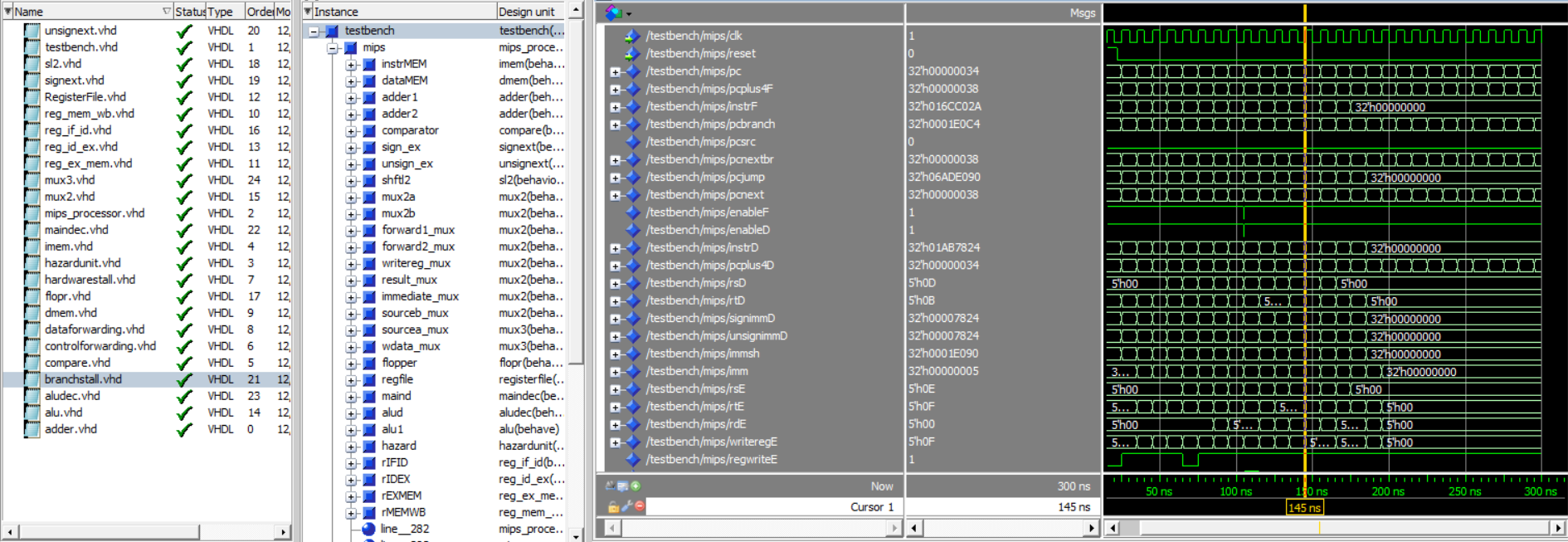
The final state of the MIPS pipeline is the writeback stage. In this stage, the results of any operation that involves the ALU or any operation where data is read from memory is written into the register. The address of the destination register is stored, and once the calculation or memory read operation is complete, the control unit will write the result into the appropriate register. While this is a simple operation, there are a few hazards associated with it. For example, a data hazard could occur, particularly in a multi-stage pipeline such as ours. In this example, the next instruction could attempt to access the information before the current instruction has finished writing the output back, reading back the old data that happened to be in the register already. Another potential hazard would be a structural hazard. In a MIPS processor, the processor cannot both read and write to the same memory location or register at the same time. If this were to happen, either the register or memory being read will have the incorrect value, or will have the incorrect value written in, or both.

Truth Table

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| instructions | RegDst | jump | branch | memread | memtoreg | Aluctrl | memwrite | alusrc | regwrite |
| addi | 1 | 0 | 0 | 0 | 0 | 010 | 0 | 0 | 1 |
| andi | 0 | 0 | 0 | 0 | 0 | 000 | 0 | 0 | 1 |
| ori | 0 | 0 | 0 | 0 | 0 | 001 | 0 | 0 | 1 |
| slti | 0 | 0 | 0 | 0 | 0 | 111 | 0 | 0 | 1 |
| add | 1 | 0 | 0 | 0 | 0 | 010 | 0 | 0 | 1 |
| sub | 1 | 0 | 0 | 0 | 0 | 110 | 0 | 0 | 1 |
| and | 1 | 0 | 0 | 0 | 0 | 000 | 0 | 0 | 1 |
| or | 1 | 0 | 0 | 0 | 0 | 001 | 0 | 0 | 1 |
| slt | 1 | 0 | 0 | 0 | 0 | 111 | 0 | 0 | 1 |
| xor | 1 | 0 | 0 | 0 | 0 | 101 | 0 | 0 | 1 |
| sll | 1 | 0 | 0 | 0 | 0 | 011 | 0 | 0 | 1 |
| srl | 1 | 0 | 0 | 0 | 0 | 100 | 0 | 0 | 1 |
| lw | 0 | 0 | 0 | 1 | 1 | 010 | 0 | 00 | 1 |
| sw | X | 0 | 0 | 1 | 1 | 010 | 0 | 1 | 1 |
| j | X | 1 | 0 | 0 | X | X | 0 | 0 | 0 |
| beq | X | 0 | 1 | 0 | X | 110 | 0 | 0 | 0 |

Diagram

Testbench



Conclusion

The MIPS pipeline processor is a 32-bit processor that uses a five stage procedure for each instruction it receives: instruction fetching, instruction decoding, execution, memory access, and memory write back. The processor has several modules built into it to execute instructions, which consist of the program counter, instruction memory unit, control unit, arithmetic logic unit, data memory unit, and several multiplexers to control the data flow for each stage of the pipeline. First, the instruction is pulled from instruction memory, and sent to be decoded and divided into parts. From there, the appropriate parts are sent to their respective destinations, register bits to the register file, opcodes and functions to the control unit. The control then decides what to do with the instruction based on the opcodes and function inputs, and will appropriately send out the information to the ALU or any other module required to execute the instruction. After the instruction is executed, the result is stored into a register, and then the processor moves onto the next required instruction.

We used existing VHDL examples, such as those provided in the project assignment, as our main reference point for this project, but we still ran into difficulties implementing it and getting it running for all of the required instructions. Most of the examples we saw did not implement the full instruction list. To get the ALU to work, we modified the ALUOp signal to be 3 bits instead of 2 and created custom codes for SLT/SLL/SRL/XOR so that these could be executed through the ALU. For the shift commands, we created a new “shift” signal from the instruction signal and fed this value directly into the ALU to be used in case of shift commands. It might have been better to use a multiplexer to render the shift value as one of the two operands of the ALU, rather than a brand new signal, but this was a quick and easy workaround.

We ran into trouble each time we tried implementing a new operation, and learned a lot about how to effectively trace the results of the testbench to pinpoint the exact location of errors and fix them. For example, to begin with, we were receiving a lot of errors where signals were receiving uninitialized or garbage values, and we had to work backwards phase-by-phase, tracing the wires in our diagram, until we determined in which component the value originated, at which point we would tinker with the code until it gave us expected results. Working through this project really helped us to understand the capacity of VHDL as a design and testing tool.

We were successful in implementing all of the required instructions for the project, as well as J, BEQ and BNE. Forwarding appears to be working as well, although we have not exhaustively tested it. The one thing we tried to get working but ultimately were not able to is stalling. We have stalling units to detect both hardware and branch stalling, but in the instruction set we created, while a stall was detected, it does not appear to have been held for a full clock cycle, and thus a “stale” value was loaded rather than the processor waiting for the writeback.

References

<https://github.com/rmohashi/Mips/blob/master/mipspipeline.vhd>

<https://www.youtube.com/user/twalsh123>

William’s GitHub: <https://github.com/wxarmstrong/mips-final>