**CMPE-260 Digital System Design**

**Lab Exercise 4**

**Execute Stage**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students. Other than code provided by the instructor for this exercise, all code was developed by me.

**Frank Andes**

Frank Andes

Performed 3/23

Submitted 3/27

Lab Section 4

Instructor: Richard Cliver

TAs: Colin Vo, Robbie, Henry

**Abstract**

Designing the execute stage for the MIPS architecture also required the continued development of the ALU. These developments included the design of the multiplier and ripple add/subtract modules. For context, the execute module manages, as names, all arithmetic executions performed by the architecture apart from the program counter. Hence, the ALU needed to be fleshed out to perform the desired operations to meet the architecture’s requirements. The execute stage consists of the ALU, as mentioned, two multiplexers which are used to determine second operand and write destination, and multiple passthroughs for other control signals. The designs were simulated and the waveforms demonstrated all modules were functional.

**Design Methodology**

The ALU is comprised of nine operations: AND, OR, SLL, SRA, SRL, SUB, XOR, ADD and MULTU. Figure 1 shows a detailed block diagram of the multiplier and adder/subtractor as well as a wrapper for the ALU multiplexer.

Diagram, schematic

Description automatically generated

Figure 1: *Block Diagram of Multiplier and Adder/Subtractor with ALU MUX*

The expanded diagram of the multiplier shows the internal logic. The design shows a 4-bit multiplier, but, as noted in the diagram, the design can be expanded, and made generic, by using “generate.” Generate refers to the VHDL language and allows the designer, in this situation, to make a module generic, meaning that changing only a few parameters allows to design to scale easily. In the case of the MIPS processor, the inputs would be 16-bits and the output would be 32-bits. The expanded diagram of the adder/subtractor also notes that a generate is used to extend the module; however, the logic is simpler and would scale linearly. Both of these modules are attached to the ALU along with the other unexpanded operations. Figure 2 shows the block diagram of the entire execute stage.

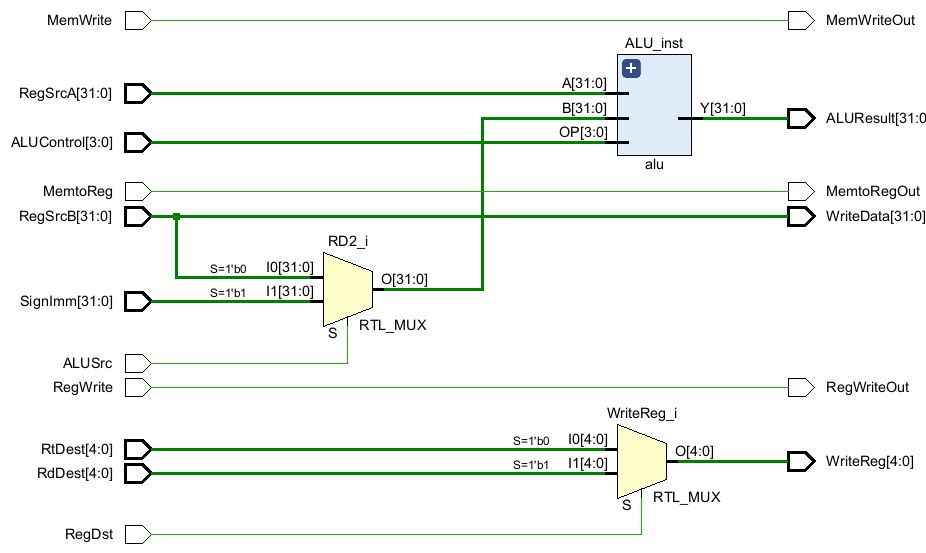


Figure 2: *Block Diagram of Execute Stage*

As seen in the diagram, the majority of inputs are simply passed through the stage. These are control signals from the control unit that are used later on in the processor. The two operands and ALU control signal are shown entering the modules and the result being sent to the next stage. The other signals used are the signed immediate and two possible destination signals as well as the control signals associated. The “ALUSrc” signal shown as the select signal for the leftmost mux chooses either the signed immediate or “RegSrcB” depending on the operation being performed. Similarly, “RegDst” is the select signal for the mux choosing between Rt or Rd. This result is sent to the next stage of the processor.

**Results and Analysis**

First the modules were simulated behaviorally. Figure 3 shows a portion of the output of the behavioral simulation for the ALU.

Graphical user interface

Description automatically generated

Figure 3: *Portion of ALU Behavioral Simulation*

In this snippet of the simulation, two tests for the AND, ADD, and SUB operations are shown respectively. The same snippet of the simulation is shown in Figure 4, but for the Post-Implementation timing simulation.

Graphical user interface

Description automatically generated

Figure 4: *Portion of the ALU Post-Implementation Timing Simulation*

This output shows the delay simulated by Vivado appearing on the “out” signal; however, most important is the consistent correctness between the behavioral and post-implementation simulations. Returning to the behavioral simulation, the waveform of the Execute module is shown in Figure 5.

A picture containing timeline

Description automatically generated

Figure 5: *Execute Stage Behavioral Simulation*

While the waveform is packed, the important signals are the tb\_RegWriteOut, tb\_MemtoRegOut, tb\_MemWriteOut, tb\_ALUResult, tb\_WriteData, and tb\_WriteReg since these are the outputs of the execute stage. These values matched the expected values in the test bench for all cases. The execute stage was also tested using a post-implementation timing simulation. The result for this simulation is shown in Figure 6.

Timeline

Description automatically generated with low confidence

Figure 6: *Execute Stage Post-Implementation Timing Simulation*

This simulation was performed by setting the “-mode out\_of\_context” flag before simulation. This allows the IDE to disregard constraints imposed by the destination chip. Without this flag, implementation is impossible since the number of I/O ports outnumbers that on the board. Most importantly, the simulation shows consistent correctness between the behavioral and implementation.

**Conclusion**

The execute stage is the stage of the MIPS architecture that performs all the arithmetic operations in the processor apart from the program counter. This stage was difficult because of its number of signals as well as the complexity of the ALU extensions: Multiplier and Adder/Subtractor. The power of generate statements is clear with how compact the code was for the multiplier and adder/subtractor; however, it adds a level of debugging difficulty that was unexpected. Moving forward, generate statements are going to be a necessity, but caution will need to be taken if a horrendous debugging experience is to be avoided.

Diagram, schematic

Description automatically generated