Processor Technical Report

Clayton Shafer

ECE 350

November 5, 2019

Objective

The purpose of this project was to create a MIPS(ish) processor, to be used in the final project for this course. Design constraints are mostly focused on the speed of the processor, both in overall clock speed and cycles per instruction. Various speed optimizations were made, both to the pipeline and to the arithmetic logic.

ALU

The ALU design is fairly simple. Addition and subtraction is carried out by a two level hierarchical carry-lookahead adder. For subtraction, the subtracted input is flipped and then a one is carried into the adder. Bitwise operations (and/or) are carried out in parallel. Shifting is performed by a barrel shifter. The barrel shifter is made up of 1, 2, 4, 8, and 16 bit shifters. The ALU interface also includes two control bits which are activated if A<B and if A~=B, respectively.

Register File

The register file is made up of 32 copies of a 32 bit register. The read ports of the register file, of which there are two, control the registers using a 5-32 decoder and a series of tri-state buffers. The tri-state buffers are used because of the huge gate delays associated with a 32-1 mux. The register file has one write port. The register file, along with the instruction and data memory, are all clocked on the negative edge of the clock, in order for the values to be steady by the time they need to be latched.

Instructions

The first step taken in creating the processor was implementing the math instructions, or the instructions with opcode 00000. Multiplication and division were excluded initially. This was probably the easiest step in the process of creating the processor. To implement these first instructions, the pipeline had to be laid out. Four special purpose registers were created, each capable of holding the values to be saved after each step in the pipeline. The program counter, pipeline registers, regfile, ALU, and memory were arranged as described in the lecture slides. Once the simple math instructions were implemented, the next step was to add the memory instructions, load word and save word. These were implemented using muxes, with extra control bits added to control when data was read from and written to memory.

Next, the branch and jump instructions were implemented. Jump required a nop instruction to be injected into the pipeline, because it could not be resolved until the decode stage. This is also true for jump register and jump and link. The key difference between these three instructions is the source of the next input to the program counter. For jump, it is simply the immediate value in the instruction. For jump register, the value is found in $r31. For jump

and link, the value is again found in the immediate, but the next instruction must also be saved to $r31. This is all accomplished via mux.

Branching is a little bit more complex. Naive branching incurs a 2-cycle penalty, since it cannot be resolved until the execute stage (at least for this ISA). The next value of program counter is found in the immediate field, which is again piped in using a mux. The control bits for the branch are found through the ports on the ALU, which are conveniently the exact conditions that need to be evaluated: not equal and less than. When both a branch instruction and its corresponding control bit are true, the branch is taken, meaning the immediate is input to the program counter, and the two instructions in the fetch and decode stages are flushed. This is accomplished via mux.

The final set of instructions that was implemented was the exception handling. Using muxes, the system was configured such that if an exception occurred, the appropriate value was written to $r30. The specific exception number was found using an encoder, with the input being a concatenation of all the overflow bits in the system, and the output being the number 1-5 corresponding to the correct error code. Bex was implemented in the same way as the other branch instructions, described above. Setx was also straightforward, involving the use of muxes. Once all this was in place, the multiply and divide units were connected to the circuit. Their operation is described below.

Multiplier

The processor uses a pipelined wallace tree multiplier. The pipeline is made up of two 32 bit multipliers, which are each made of a series of 8 bit wallace tree multipliers. After rigorous testing of the 32 bit multiplier unit, I determined that it would not be possible to pipeline the multiplier in the usual way. The delay of the 32 bit divider is around 1.5 cycles for a 50MHz clock, and it cannot be easily split into parts which can be pipelined. To achieve one cycle multiplication, two 32 bit multipliers were used. A clock divider is used to switch between the two multipliers on alternating clock cycles. The multiplication unit stretches from the execute stage to the writeback stage, since it takes two cycles to complete. The multiplication result is muxed into the writeback data port. The architecture of the multiplier is shown in the Quartus RTL viewer screenshot below. Another important feature is the column of registers seen in the middle of the picture. Because the input operands will switch before a given multiplication operation is complete, the operands must be latched into registers every clock cycle. The half-speed clock from the clock divider is used to control the enable pins of the four registers, so the correct operands are latched every time.

The biggest problem that arose when implementing the multiplier was the construction of the wallace tree. Wallace trees are a way to easily condense the matrix of partial products into two numbers which can be added using a normal two operand adder. The partial products are reduced using a series of full and half adders, arranged different stages. For this particular design, since the wallace trees were 8 bits, 4 stages were needed. Unfortunately, this

arrangement of full and half adders is highly irregular and annoyingly complex. It required a lot of tedious checking and rechecking to find errors. The end result, which was one-cycle multiplication, was well worth it.

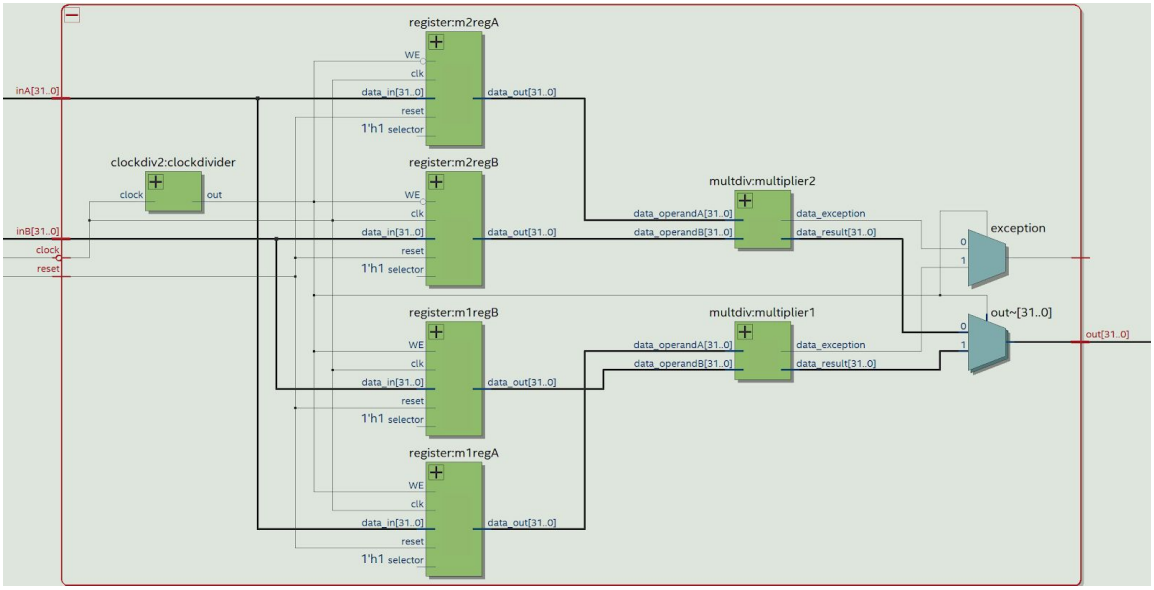

Figure 1 - Pipelined Multiplier

Divider

Instead of a sequential divider, the processor uses a combinational array divider. Array dividers have a huge delay time, but they are still faster than the sequential divider recommended for this course. The divider is a series of 32 bit adders. The particular algorithm used was nonrestoring division, meaning each row of the array either adds or subtracts the divisor. The high delay time is due to the fact that the add/subtract signal must propagate down each row of the divider, in roughly n^2 time. The finish pulse of the divider was produced by hooking together a chain of D flip flops. The start pulse is the input at one end, and a certain number of cycles later, the same pulse comes out the other side. Through rigorous testing, it was determined that the minimum number of cycles needed for the divider to complete its task is 24 cycles. This is a fairly significant improvement over the 32 cycle sequential divider, although it comes at significant hardware cost.

Unlike the combinational multiplier described earlier, this was straightforward to implement. 32 adders were cascaded together, each offset by one, to form a parallelogram. The biggest challenge here was to create a circuit that would integrate the circuit properly into the processor. A one bit control signal initiates the division process, which is triggered when a divide instruction is in the execute stage of the pipeline. Once this signal goes high, the pipeline stalls after a delay of one cycle. The pipeline holds in this position for 24 cycles, with the divide instruction in the memory stage. When the control signal has made its way through the chain of

flip flops, the result ready signal is asserted, which restarts the pipeline by latching the division result into the memory/writeback register. From this point, the pipeline proceeds as normal.

Pipeline Hazards

In the processor, data hazards were avoided using a series of bypassing systems, which consisted of muxes between the different stages of the pipeline. However, there were also several scenarios which could not be resolved using bypassing. The first of these is loading a value from memory directly into the ALU input. Since both these actions need to happen at the same time in order to keep the pipeline moving, a bubble must be inserted into the pipeline to prevent the wrong value from being used in the execute stage of the second instruction. Once the bubble is inserted via mux, the correct value can be bypassed from the writeback stage to the execute stage. In addition, bubbles must be inserted into the pipeline in certain mul/div situations. Because of the nature of the multiplier, its result cannot be read until it is in the writeback stage. This means that if there is a read after write hazard involving the output of the multiplier, a bubble must be inserted in order to prevent the second instruction from using the wrong value. Another similar situation arises when two divide instructions are next to each other. Because of the way the divide control signals work, the instruction in the execute stage must not be a divide while there is a divide in the memory stage, which is effectively when a divide is executing. This situation would cause an infinite stall in the pipeline. To prevent this, a bubble is placed between any two adjacent instructions. All three of these types of bubbles are placed in the decode stage.

The second type of hazard under consideration is control hazards. Unfortunately, due to time constraints, the control hazard logic in this processor is not super efficient. If more time were available, a branch predictor would be a good way to increase the average performance of the processor. However, in this case, the two instructions after an incorrect branch are simply flushed, without any optimizations.

Testing

The processor was tested using waveforms and testbenches. Waveforms were used in the early stages, in order to see the behavior of the system over the entirety of the simulation. However, waveforms are also pretty annoying, so once the behavior of the latches was verified, most of the debugging was done using testbenches. The values of all the registers was printed to the modelsim terminal, in order to track the changes as the instructions flow through the system. Due to time constraints, most of the testing done on the processor was functional testing. Some timing simulations were carried out, but mostly functional simulations. The only errors which are known about the system are found in the timing simulations. At the required clock speed, sometimes instructions are not completed by the time the pipeline latches. The source of this error is unclear, but it is probably caused by a section of the circuit that is too long. After testing, I think it is in the writeback stage, but it is really hard to tell. I have tried a variety of different configurations of the way the writeback control and data are chosen, but they all seem to take too long.