Final Project Report

Judah Ben-Eliezer | Tennyson Cheng

Introduction:

The multimedia unit design we made was designed with simplicity and speed in mind. The processor is designed in such a way that the maximum number of operations are synchronized, and it takes advantage of pipelining to ensure that no component is idle at any time.

ALU:

For the ALU, we used a behavioral model with a case? statement to encapsulate the opcodes. The additional file ALU_functions.vhd contains operations that are used repeatedly in the ALU, so as to avoid repetition. The ALU is timeless, with output sensitive to any change in the input vectors.

Register File:

For the register file, the biggest challenge was getting the simulator to accept writing from different source signals without driving the file to the std_logic value 'X'. In the end, we used a behavioral model with an infinite while loop rather than a process sensitive to changing inputs. Instead, the while loop merely halts while waiting for change. This way we were able to use a variable rather than a signal to model the register file. Like the ALU, it is timeless.

Instruction Buffer:

This was nothing complicated, just a simple buffer that writes the input program upon reset and writes the instruction pointed to by the program counter. Again, this module operates without a notion of time.

Forwarding Unit:

For the forwarding unit, we wanted to minimize the times when it would be used as it is likely to be time consuming. Thus for operations that don't use the register inputs to the ALU, the comparison is avoided completely. For those that do, the forwarding unit compares the addresses of the input vectors to that of the ALU output destination. If there is a match, the data is forwarded so the ALU can use the most recent value of the register.

MMU:

This module contains the full pipeline. We implemented it with a structural design, using the previously mentioned components as well as three registers, REG0, REG1, and REG2, to save the intermediate values in the pipeline. On each clock cycle, an instruction is read into REG0, REG0's value is put into the register file and the appropriate registers are read into REG1, REG1's values are put though the forwarding unit into the ALU and the result is written into REG2, and finally REG2's value is written back into the register file.

Testbench:

The testbench is a simple module that initializes a MMU module, and passes in an input from a text file containing opcodes to test on the MMU. It also reads the status of the pipeline on each clock cycle, and writes the output to a file results.txt.

Assembler:

We chose to write the assembler in python because it was simple. The format for writing programs is as follows:
Li instructions: li immediate, offset(register)
R4 instructions: op rd, r1, r2, r3
R3 instructions: op rd, r1, r2
It is fairly strict with syntax, each register name must be preceded by the letter r and the opcode must be valid or the compiler will simply fail with a format error.

Results File:

The results are written by the testbench into a text file. At each clock cycle, the testbench writes the following signals: PC, instr0, instr1, instr2, write_enable, li, write_addr, and write_data, as well as the registers in the pipeline rf_out, fu_in, and alu_in.

Conclusions:

The pipeline is definitely a great way to speed up a processor to handle operations in a shorter time period. While it requires a lot of extra circuitry to avoid hazards and save registers, it can process 4x the number of instructions in the same time. Some changes to improve it would be to make opcodes for easier loading, which would be especially helpful for testing purposes. Either that, or make the instruction buffer bigger. The test program shown in the submission contains mostly li instructions to make the register file more interesting. In any case, we were able to effectively leverage pipelining to make a processor capable of handling lots of data at high speed.