CMPE 260 Laboratory Exercise 6

Project 1 - Pipelined MIPS

Andrei Tumbar
Performed: May 1st
Submitted: May 3rd

Lab Section: 1
Instructor: Moskal
TA: Jacob Meyerson
Dennis Lam

Lecture Section: 1
Professor: Cliver

Abstract

In this laboratory exercise, the pipelined MIPS was implemented by placing a register in-between the stages of the MIPS. The result of pipelining is that the clock period can be shorter but there is a delay when the instruction is loaded from instruction memory and when the result is ready for use. The programmer must be mindful of this delay and take it into account when writing the tests. To test the pipelined MIPS, two distinct programs were written: the first separately tests each valid instruction available on the processor. The second test will implement a fibonacci sequence and load the results into memory.

Design Methodology

There are five main stages to the MIPS pipeline: Fetch, Decode/Register File, Execute, Memory, and Writeback. Signals between each stage are held between rising edge triggered flip-flops. This way their values may be propagated down the pipeline after each clock cycle.

To easily test the process a set of scripts were created in Python. The first script, `mips_asm.py` compiles a file written in standard MIPS assembly by encoding lines of assembly in hex. The second script will reprocess the compiled program into a format that VHDL can load into instruction memory. The instruction memory was adapted to read lines of `.mem` files.

Using the these two scripts to compile and initialize instruction memory, `part_a.s` was implemented to test all of the separate instruction and `fib.s` was written to implement the fibonacci sequence. Separate registers were used to store each of the items in the fibanacci sequence up to a_{10}. Store instructions were performed after three clock cycles to write the data back to memory. The reason we have to wait three clock cycles is because there are four sets of registers and it takes a single cycle to load the instruction. To fill one of the dead cycles with a useful operation, the memory load from the previous item can be placed here. This means that there are two dead cycles per item in the sequence.

To properly test that the results are correct and loaded into memory, after all of the sequence items are calculated and stored, we execute an instruction where to result is a special value that the testbench can verify. In this case the value `0xFFFFAB12` was chosen because its a large distinct value. Following the execution of this special instruction the output of each fibonacci item is loaded from memory and the test bench is able to verify the results by looking at the results of the instruction. A list was written in the testbench with the true first ten elements of the fibonacci sequence and verified against the results loaded from memory.

Results & Analysis

Test benches for part a and part b are shipped with expected outputs. There are self verifying by design with assertions placed throughout the simulation. A behavioural waveform was generated for part a to show the results. The part A testbench uses the lowest three registers for inputs to test all operations.

Table 1: Input registers for Part A

Register	Value
r1	0xFFFFFEFE
r2	0xFFFFCECE
r3	0x8

Using these input values, all of the operations are tested. The following table will show the expected values for each of the ALU operations.

Table 2: Input registers for Part A

Operation	Operator 1	Operator 2	Output
and	0x8	0xFFFFCECE	0x8
or	0x7	0xFFFFCECE	0xFFFFCECF
xor	0x7	0xFFFFCECE	0xFFFFCECF
multu	0xFEFE	0xCECE	0xCDFD9464
sll	0xFFFFFEFE	0x8	0xFFFEFE00
sra	0xFFFFCECE	0x8	0xFFFFFFCE
srl	0xFFFFCECE	0x8	0x00FFFFCE
sub	0xFFFFFEFE	0xFFFFCECE	0x3030
add	0xFFFFFEFE	0xFFFFCECE	0xFFFFCDCC

Both the immediate and R-Type instructions for the relevant operations were tested. A waveform was created to show the output.

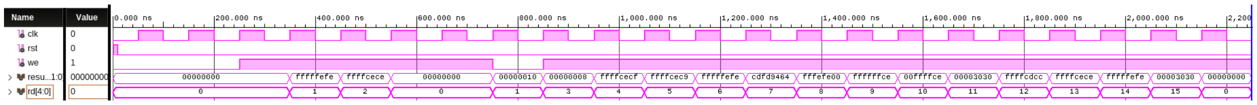

Figure 1: Behavioural simulation of Part A

The assertions placed in this test bench verify that the outputs are valid.

Testing the fibonacci program is far simpler than the testbench in Part A. To test it simply verify that the output of the first then fibonacci numbers match the expected sequence: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55`.
A behavioural waveform was generated to verify the results. Outputs are formatted to decimal format to more easily see the results.

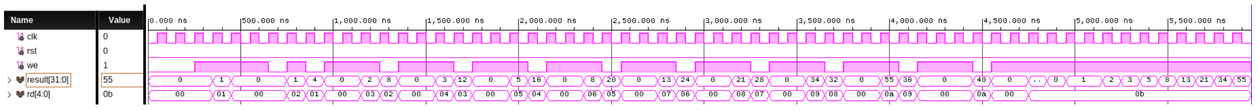

Figure 2: Behavioural simulation of Part B

Figure 2 and the fact that the simulation finished without tripping an shows the validity of the program and processor implementation.

Conclusion

This laboratory exercise finished the implementation of the pipelined MIPS processor. Testbenches written along with the assembly files helped verify the processor. While writing the pipelined MIPS, the it was deemed more efficient to get rid of the passthrough signals on the execute stage. The same functionality could be kept by passing signals through the pipeline instead. Looking at the results of the test programs and the waveforms generated by the testbenches, the implementation of the MIPS processor is a success.

Demo results

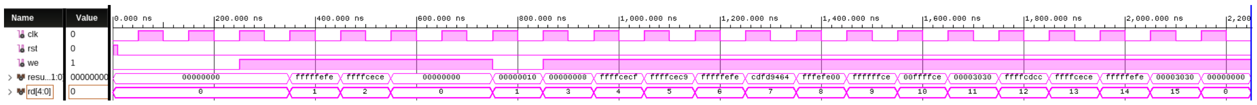
Figure 3: Behavioural simulation of all instructions

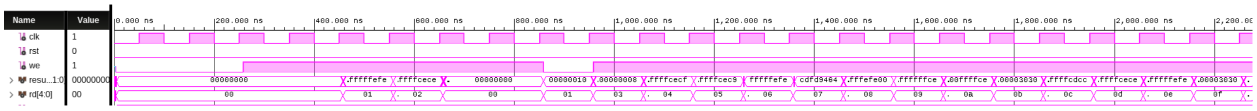
Figure 4: Post-implementation timing simulation of all instructions

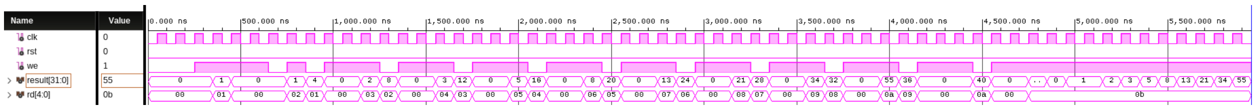
Figure 5: Behavioural simulation of fibonacci sequence

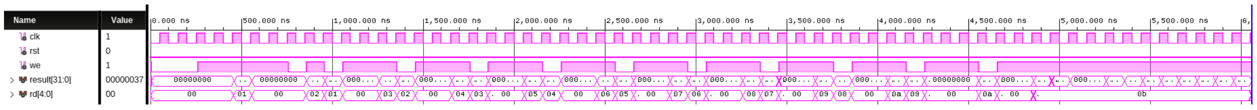
Figure 6: Post-implementation timing simulation of fibonacci sequence