CMPE 260 Laboratory Exercise 6
Pipelined MIPS Processor

Glenn Vodra
Performed: April 19, 2023
Submitted: April 20, 2023

Lab Section: 3
Instructor Sutradhar
TA: Ian Chasse
Cole Johnson
Robbie Riviere
Jonathan Russo
Colin Vo

Lecture Section: 1
Professor Cliver

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further, acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Abstract

The purpose of this exercise was to design a full five-stage MIPS processor. The processor supports the execution of simple R and I-type instructions, which are initialized in instruction memory when the design is run. To design this processor, previously designed stages, Instruction Fetch, Instruction Decode, Execute, Memory, and Writeback were connected to each other with registers. Each register holds data and control signals for a single instruction, which moves through the five stages until execution is complete. This design allows multiple stages to work on separate instructions, increasing throughput. The MIPS processor implementation was tested with two programs and simulated their execution correctly. This exercise was successful because a processor was successfully designed and programmed.

Design Methodology

To understand the full processor design, each of the five stages must first be briefly described. The first stage is the Instruction Fetch stage. This stage retrieves instructions from memory four bytes at a time to fetch full 32-bit instructions. This stage also updates a program counter which points to the next instruction in memory. This value is updated by adding four to address the next word, rather than the next byte. This stage outputs a single instruction which is then stored in a register on the rising edge. This is all the Instruction fetch does at this time. Explanation of how and what instructions were stored in this stage will follow with the discussion of testing, and writing programs for this processor.

The next stage is the Decode stage. This stage takes instruction and starts to understand what data and control signals are needed to execute the instructions. Some of the control signals include what bits to use for operands, where to store data, and what arithmetic operation to perform. There are six control signals that are created at this stage. In addition to the control signals, this stage also reads and writes to a register file. The register file has an asynchronous read and can read from two ports at a time. The writing functionality is falling edge triggered and is controlled by the data and signals that make it to the Writeback stage of the processor. As with all stages, this is then fed into a set of registers for pipelining.

The stage that follows is the Execution stage. This stage performs arithmetic operations on the data passed into this stage. It also passes through three control signals and some information about write data and write back location. The arithmetic logic unit designed in this stage supports nine operations. These are all 32-bit operations unless noted. The operations are, addition, logical AND, 16-bit multiplication, logical OR, logical XOR, shift left logical, shift right logical, shift right arithmetic and subtraction. The type of operation chosen is determined by the instruction and what control signals are generated in the control stage. As stated previously, this processor supports I and R-type instructions. For memory operations, 32-bit immediate values are passed from the decode stage. These operations, use the Addition functionality to calculate memory addresses. Output signals are passed into a register.

Following, the next stage is the Memory stage. This stage holds data memory. Data memory is used to hold variables, as well as store more data than the general purpose register file. This stage takes all of the output signals from the Execution stage. Two of the control signals are passed through this stage. The other is used to set the write enable on the memory module. The memory module reads and writes on the

rising edge of the clock, however, with the instructions this processor supports, only one of these operations is needed at a time. The output of the Memory stage is then clocked. The final stage, the Writeback stage contains a single multiplexer. This checks if data memory or ALU data is stored back in registers. This stage also connects the necessary signals to the decode stage's register file. Depending on if writeback is needed, a write enable signal will be set. There is no output or this stage that needs to be clocked.

Two programs were written in MIPS assembly to be executed on this processor design. The first of these is a simple program that loads two values into registers with an "addi" instruction, then performs every operation and load/store the data, to test all the functions of the design. The second program generates a "Fibonacci" sequence. This sequence is generated by adding the previous two numbers, starting with a single zero and one. The code that was written archives this by adding two registers and storing the result in the register holding the older number. This is simply done by alternating. The code also uses the "addi" mentioned before, for the same purpose of loading initial values to the registers. To avoid data hazards, a situation where old data is read before a correct value can be written back, no operation or nop instructions were used to pad the pipeline. This gave the instructions time to execute and write correct data so it could be used. This is very important for this implementation of the Fibonacci generation, as only two registers are constantly being read and overwritten. In addition, all of the values were stored to separate memory locations. The first ten values of the Fibonacci sequence were generated.

These two programs were added to the instruction memory by encoding the instructions in hex, then breaking the 32-bit instructions over four bytes. The writing of memory was done when the memory was instantiated, as the processor provides no way to write data to instruction memory while running.

Results and Analysis

The two programs were run on the processor. The first program, the one designed to test the processor's functionality was tested first. It starts by loading a 10 and a 12 into $s0 and $s1 respectively. Each of the arithmetic functions was tested, with each result being written to a $t register, starting at $t0 and finishing at $t8. $t9 contains the result of a load word operation, "lw", which validates the functionality of that instruction as well as the store word, "sw", instruction that put the value in memory. The arithmetic instructions in the order they were executed are, "add", "and", "multu", "or", "xor", "sub", "andi", "ori", and "xori". The result of "xori" was selected as the value written and then read from data memory. The following figure, figure one, shows registers eight through fifteen, with results, and sixteen and seventeen with values 10 and 12.

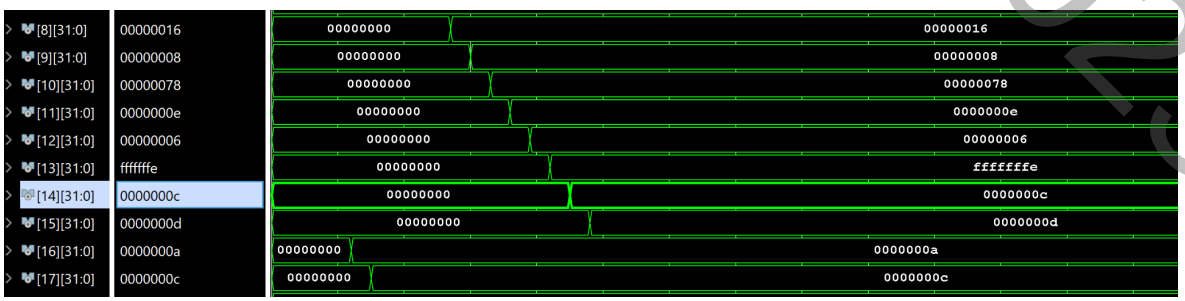

Figure 1: *Register Eight Through Seventeen*

Figure one shows the result of several arithmetic operations that were executed from instructions. The first value is “0x16” or 22, the result of 10 + 12. There is also “0x8” the result of the AND “1010” with “1100”, “0x78” the result of 12*10, 120, and so on. Registers sixteen and seventeen hold the values ten and twelve. Load and store instructions were also tested. This is shown in figure two.

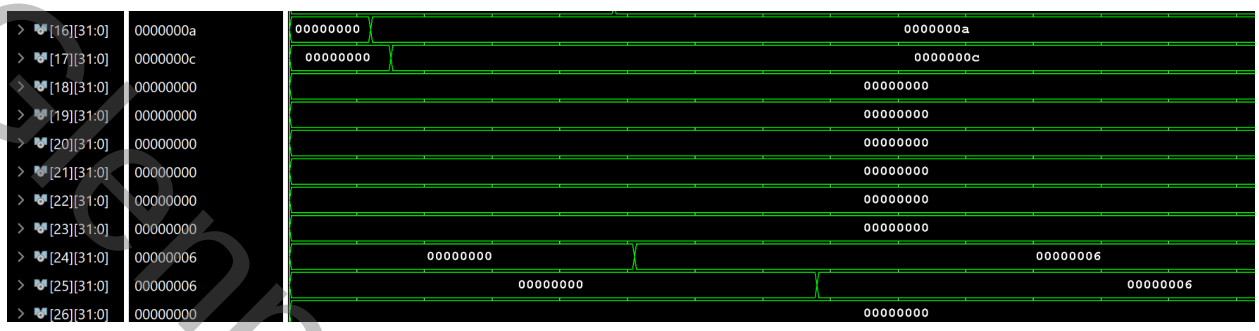

Figure 2: *Load and Store*

In figure two, register twenty-four shows the result of the “xori” instruction, which is then stored to memory address “0x0000000C”. This value is then loaded into register twenty-five several clock cycles later. In the previous two figures, the clock was not visible. Figure three shows the clock and the test bench signals for this series of instructions.

Figure 3: *Full Processor First Program Simulation*

In this figure, figure three, the clock signal is visible, as well as the memory stage value to write, and the ALU result. The signals are shown in signed decimal so they are readable.

In addition to testing all of the MIPS instructions, the Fibonacci sequence generator was also tested. Details about what instructions were executed are provided in the design methodology section. The first 10 values of the Fibonacci sequence are generated in the simulation below.

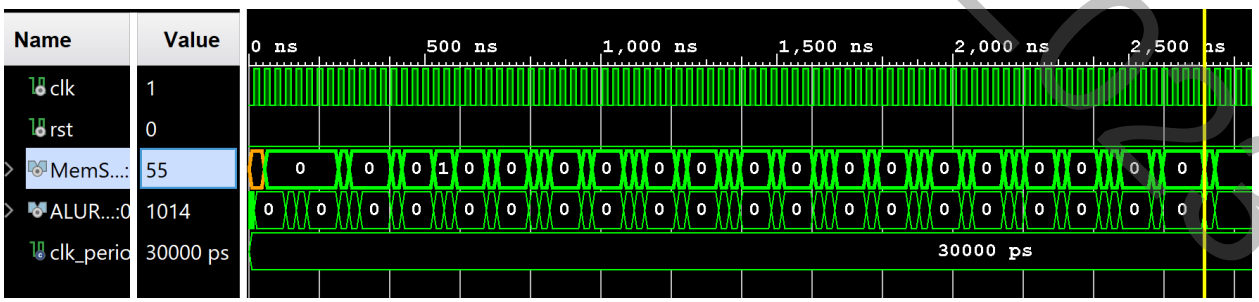

Figure 4: *Full Processor Fibonacci Program Simulation*

To get a clearer view of the values being generated in the simulation shown in figure four, figure five is provided. Each value of the sequence was generated and stored in memory. This is shown in figure five.

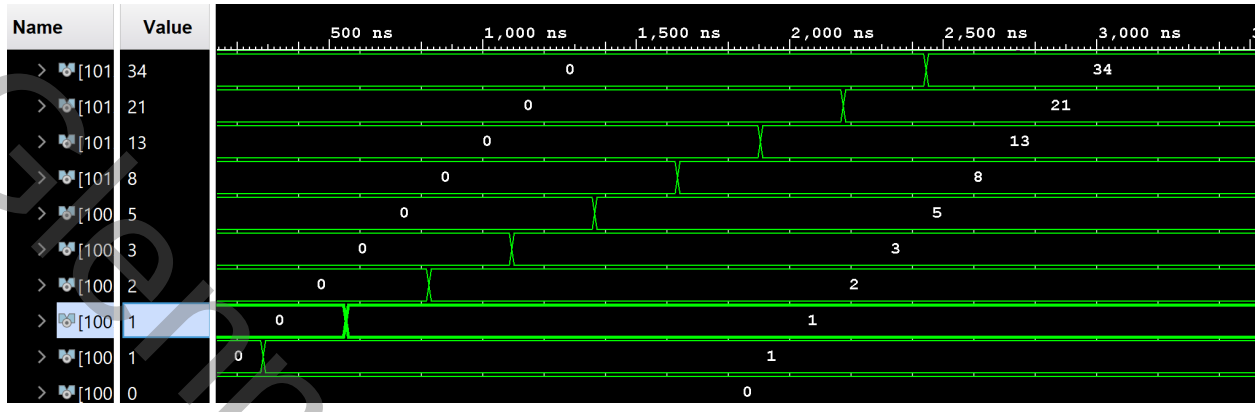

Figure 5: *Fibonacci In Decimal*

This figure shows the result of the Fibonacci sequence generation. Here are the steps:

1 + 0 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34

Fibonacci Equations

Since the calculations match the waveform this sequence was generated correctly.

Conclusion

This exercise was successful as a pipelined MIPS processor was designed and tested. All five stages were connected together, and the processor was programmed with two different tests. The two tests performed as expected when consideration was given to data hazards in the design. While this process was simple to put together, it involved a few layers of abstraction that helped everything work. When connecting the five stages of the processor, there is no need to be concerned with how instructions are automatically fetched, or how control signals are generated. All that matters is that the signal connects to the locations they need in order to pass data through the stages. Abstraction is a very important concept in hardware design as it helps direct focus, and simplifies the design process. Implementing a system that supports a simple set of versatile instructions is also worth acknowledging. Processors should be designed for the task they are needed for, even if the task is to be a general-purpose machine. Understanding what role hardware plays in supporting the functionality of the design is very important. Overall this exercise was successful.