Logisim Simulation of a Single Cycle MIPS Architecture CPU

Malachi Henry

# Introduction

This is my final project for Logic and Computer Design. I set out to design and simulate a MIPS style processor at the gate level and recreate the design in Logisim. I created a custom instruction set and implemented a few dozen instructions including more complex operations relating to the program counter, register operations, and RAM reading/writing.

# ALU Functions

Before I dive further into the design of the processor, I believe it is important to list the ALU functions as there are several ALUs present in the design. I chose a 3 bit select line for this ALU meaning there are 23, or 8, different functions. The following is a list of all the functions this ALU is capable of as well as the select code:

Fxn Code

XOR 000

OR 001

ADD 010

SUB 011

AND 100

NAND 101

XNOR 110

NOR 111

# High Level Overview

The processor I designed is a 32-bit processor, meaning the data busses are 32 bits wide. This also means that the maximum register size that can be addressed is 32 x 32 bits, and the maximum amount of addressable RAM is 232 bits.

For the sake of ease of explanation, I will reference the steps that occur within the processor by their names in a pipelined processor, for example instruction fetch, decode, writeback, etc. This is not, however, a pipelined processor so please do not mistake these for actual pipeline stages.

In the instruction fetch stage, the next address is clocked in and the instruction is pulled out of the ROM block. The reason I chose to use a ROM block is because it allows me to write assembly programs and save them easily. Once the instruction is pulled from memory, it is fed to the decoder. The decoder turns the instruction into the essential signals required to execute the instruction. Addresses to locations in the CPU register, along with an “immediate” value comprise all the data required to operate the CPU and execute the program. Additional pieces of information include controlling signals for jump, branch, storing and loading data in RAM, and modifications to the program counter.

Once all the data and controlling signals have propagated throughout the processor, the execution stage can begin. There is a main ALU which computes results of operations and calculates the RAM locations for loading/storing. A shift register completes shift operations, a small logic block determines equality between data busses for branching, and ALU/shift register arrays compute the next address. The end result of the operation is then written to it’s final location on the various output busses, and the processor is ready for a clock cycle to push it to the next instruction.

# Instruction Set

I was not able to implement every MIPS instruction in my CPU, though I have added enough instructions to write simple programs. More complex operations like move and load immediate can be achieved using clever implementations of available instructions. The full list of instruction is as follows:

R type:

XOR OR AND

NAND XNOR NOR

ADD SUB SLL

SRL SRA

I type:

LW SW XORI

XNORI ANDI NANDI

XNORI NORI ADDI

SUBI BEQ BNE

J type:

JMP

As previously mentioned, even though this processor lacks instructions like LWI, other instruction can be used to simulate this function. To load a value into a register, the instruction ADDI can be used. All you must do is add the desired constant to register R0 (which is always 0) and store the result in the desired register. To move a value from one register to another, use ADD to add the source register to R0 (again, always 0) and place the result in the new register.

This CPU has two ways to modify the program counter, branch and jump. Branch is relative, meaning that when the branch hardware is allowed to modify the program counter, it adds a constant to the current value. This is the opposite of jump, which is absolute. When the jump instruction is issued it immediately modifies the program counter with that new value, ignoring whether the new value is larger or smaller than the current value.