# README

The following files have been modified and will be included:

* mipssystem.v (top-level module to connect cpu and exmem)
* mipscpu.v
* exmem.v (added memory mapped I/O space)
* tb\_mipscpu.v (for testing the addi instruction)
* controller.v (added addi instruction)
* fib-14.asm (modified to meet new specifications)

Summary:

The first task was to add the ADDI instruction. This took place in the controller module. After implementing these changes, a testbench was set up to ensure that the ADDI instruction was working as expected. In this testbench the controller and alu modules are instantiated. During the initial begin block the op code is set to 6'b001000 which corresponds to the ADDI instruction op code. The controller will output the appropriate aluop signal to be passed to the alucontroller. The alucont signal expected to be output by the alucontroller is 3'b000 and this will be passed to the alu module to execute the operation.

The following waveform shows when we input the ADDI aluop code to the controller it outputs the appropriate aluop signal which is 00. We then test the alu and see when we add two operands together, a and b, being 1 and 2 respectively, the alu result is 3.

A screenshot of a computer

Description automatically generated

Next is to set up the memory mapped I/O in the exmem module. This required adding inputs and outputs for the switches and LEDs on the board. We added extra logic so that when we are reading in an address that has the top two bits equal to 11 we will output the values from the switches or we will send the data out to the LEDs.

After getting the memory mapped I/O we need to connect the cpu and the exmem modules. This was done in the mipssystem module where we instantiate the two and the appropriate inputs/ouputs feed into each other.

The fibn.asm code was modified to compute and store each of the Fibonacci numbers up to 14 (zero-based). A problem we ran into here was with the logic in the modified Fibonacci MIPS code. We failed to see that the algorithm was decrementing our iterator (n) for the while loop. We were using n as the offset to store the numbers calculated in memory. This of course stored the numbers in reverse. We noticed this when we uploaded the code to the board and saw that each index offset was displaying the opposite fib number we were expecting. We decided to introduce a new variable for an index to use as the offset for storing the number. After correcting this we got the expected results.

Mars was used to assemble and simulate the modified MIPS code. The format that Mars outputs is all 32 bits included on one line. Our FSM reads one byte of the instruction at a time, therefore, we had to modify the assembled MIPS code so that it broke this down to be one byte per line and padded the file with zeros to be 256 bytes which is what our FPGA is expecting.