# 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)
* fibn.asm (modified to meet requirements)

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 we add outputs and inputs 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). Once the set-up completes, we enter a continuous loop that reads the values from the switches and outputs the Fibonacci number that was looked up in memory based on the switch value as an offset. 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 and padded the file with zeros to be 256 bytes.