CPU Simulation Strategy
CS141 Lab 4 - CPU

Alex Kahng, George Zhang

Introduction

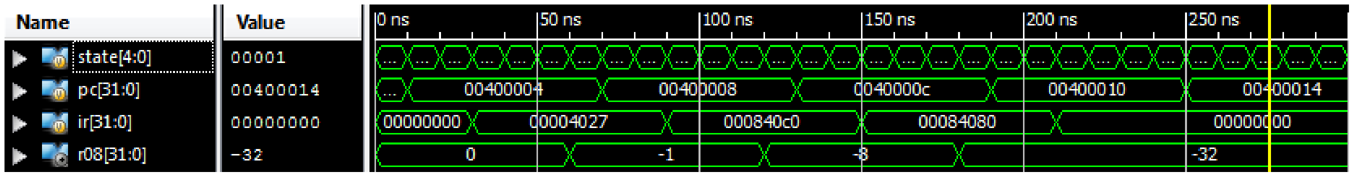
In lab 4, we were tasked with building a CPU that could handle a restricted subset of MIPS assembly instructions. We built the CPU incrementally over three weeks, starting with support for R-type instructions, then I-type instructions, and finally jump / branch instructions. As a result, our testing was broken up naturally into these three categories as well. For each category, we started with testing each instruction individually, and then we wrote a program that combined all instructions that fell into that category. After testing all the instructions in this way, we wrote one more program that included instructions from all the categories, allowing our CPU to run a complete program.

R-type Instructions

We had to implement the following R-type instructions: and, or, xor, nor, sll, srl, sra, slt, add, sub. Much of the final datapath was implemented through these instructions alone, so we wrote tests for all the instructions and confirmed the CPU's functionality by examining the waveforms. Below is a sample test we ran:

```
nor   $t0   $zero   $zero      # Set $t0 = -1
sll   $t0   $t0     3          # Set $t0 = -8
sll   $t0   $t0     2          # Set $t0 = -32
```

This test was specifically for the sll instruction. The waveforms it produced are below:



Here we list the state, the program counter (pc), the instruction register (ir), and the $t0 register (r08). We can see as the program steps through, we do indeed see our registers changing. After the first instruction, r08 is set to -1, followed by -8 and -32. Each of these changes corresponds to one of our instructions, as can be seen in the ir register.

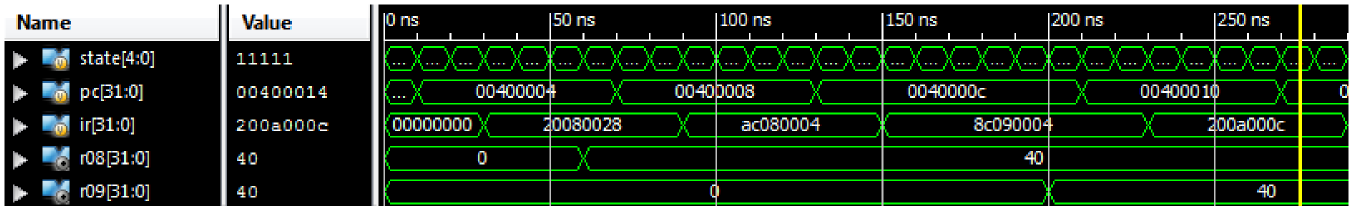
We tested the rest of the R-type instructions separately using similar approaches, first writing the assembly, compiling it to machine code using our assembler, and then simulating it to view the waveforms. Our tests passed in each case, and so we were satisfied that our implementation correctly handled the R-type instructions. As a final check, we ran a program with all the R-type instructions together, which can be found in rType.asm. Since there are many instructions in it, modifying several registers, we shall not include it in this paper, but it indeed produced the waveforms we expected.

I-type Instructions

Implementing I-type instructions required adding several more pieces to the datapath. We were tasked with implementing the following I-type instructions: andi, ori, xori, slti, addi, lw, sw. To test these additions, we used a similar algorithm. Below is a sample test assembly file for load and store word instructions:

```
addi   $t0   $zero      40        # Set $t0 = 40
sw     $t0   4($zero)             # Store $t0 at address 4
lw     $t1   4($zero)             # Set $t1 = 40
```

Here, we simply set the value of $t0 to be 40, saved it in memory (at address 4), and then loaded it back into register $t1. The waveform for this test is below:



As we can see, on the first instruction, we set r08 to be 40. The subsequent instruction does nothing to the registers, since it saves the value to memory. The instruction after that then loads the value into r09 ($t1), which is what we see occur at 200ns. The result of this tests suggests that we are correctly storing and loading words into and from memory.

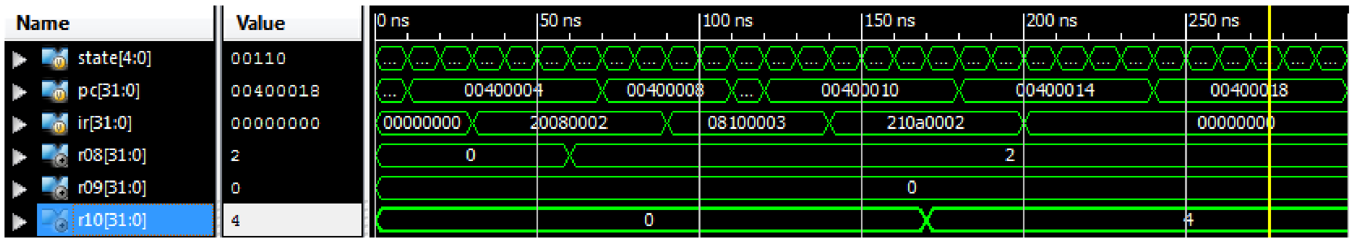
As with the R-type instructions, we wrote individual tests for all I-type instructions and confirmed that the waveforms produced the expected results. We then wrote a program that tested all the I-type instructions together, which can be found in iType.asm. This file also produced the waveforms we desired.

Jump/Branch Instructions

Jump / branch instructions were the final piece of our CPU. We had to implement: j, jal, jr, beq, bne. Here we cared more about what our pc would result in, but we tried to write tests in which it was obvious that jumps / branches were occurring. The following is a sample test of the j instruction:

```
    addi   $t1   $zero   2      # Set $t0 = 2
    j      A                    # Jumps to label A
    addi   $t1   $t0     4      # Should NOT set $t1 to 6!
A   addi   $t2   $t0     2      # Sets $t2 = 4
```

Here we perform a jump that skips over one of the lines. Thus we should be able to see a change in the pc, but we should also see the effect of the jump, namely that $t1 should not get set to 6. The code above produces the following waveform:



As we can see, we properly set $t0 to 2, perform the jump, and set $t2 to 4 while not setting $t1 to 6. In addition, at 110ns, we see a change in the pc, signifying that our jump was successful. Thus both in terms of code functionality and through examination of the pc, we see that our jump was successful.

The rest of the jump and branch instructions were tested using a similar methodology. In each case, we not only looked at the change in the pc, but also the effect such a jump or branch would have. Through our testing, we confirmed the correctness of our jumps and branches. However, we did not write a combined test for this part. Having 5 different jump / branch scenarios seemed clunky and ineffective, so we settled for individual testing. However, we did write one final full program test that combined instructions from each of the three parts, and in particular, utilized several jump / branch operations within the same program.

Full Program Test

Our full program test can be found in fullTest.asm. We shall reproduce neither the code nor the waveform analysis here for the sake of space, but the machine code for the full test is currently loaded in as the current_test and so can be run easily using the simulator. Here we shall simply give an overview of what the program does.

Our goal was to utilize as many of the instructions as possible while writing a logical (i.e. not dummy) program. We decided to model our program after the factorial program from lecture, except instead of recursively multiplying, which we did not support, we recursively added. Thus our program essentially found the nth triangular number. We managed to make use of both jumps and branches (for checking base case and for recursive function calls), as well as addition (for computation) and load / store word (for stack management). The waveforms produced the expected values, and so overall, we were satisfied with the result.

Conclusion

Here we have detailed our testing methodology for this lab. We took an incremental approach to testing, both across categories and within a single category. This gave us confidence in our previously tested code as we delved deeper into other parts of the lab.