Jonathan Collins, Arnav Srivastava

All of our testing was done by waveform analysis.

R-type testing (except jr):

```
nor $t0, $zero, $zero   # $t0 = 111111...111
slt $t1, $t0, $zero     # $t1 = 1, since -1 < 0
add $t2, $t1, $t1       # $t2 = 2
add $t3, $t2, $t2       # $t3 = 2+2 = 4
sub $t4, $t3, $t1       # $t3 = 4-1 = 3
and $t5, $t3, $t3       # $t5 = $t3 = 4
or  $t6, $t3, $t4       # $t6 = (100)|(011) = (111) = 7
sra $t7, $t4, 1       # $t7 = 1
srl $s0, $t0, 4       # $s0 = 00001111111111...111
sll $s3, $t0, 4       # $s3 = 111111...11110000
xor $s4, $t3, $t4     # $s4 = 111 = 7 = 7
```

For our testing methodology, we tested each R-type signal for the ALU and ensured that it was working properly. We used a NOR and SLT to get a “1” value in register $t1. From there, we could use the ALU operations to ensure the datapath gave us the outputs we wanted. In order to ensure the entire datapath was working as expected, we checked the control signals worked as expected at every stage, and paying special attention to the ALU Output (ALUResult).

By looking at the waveforms in the cycle before our enable signals were high, we could catch the values that were written into our registers and ensure they were the values we were expecting. For example, we would check the ALUout the signal before the PC enable was high and check to see if the result was PC+4. We also made interdependencies which carried from one register to the next. For example, the register between the immediate output of the ALU and the input to write to the register file helped us to see if there were errors in our other control signals (in this case it would be the Mem-to-Reg mux select signal). With interdependencies, we also made sure that each register was assigned the expected value: if one instruction sets $t3 and consequent instructions use $t3 as an input, then the expected ALU output for the later instruction indicates that our write back worked as expected.

addi, andi, xori, ori, slti testing:

```
addi $t0, $zero, 5      # $t0 = 5
andi $t1, $t0, 3        # $t1 =  ...00001 = 1
xori $t2, $t0, 11       # $t2 = ...00101^...01011 = ...001110 = 14
ori  $t3, $t1, 14       # $t3 = ...001 | ...001110 = ...001110 = 15
slti $t4, $t2, 15       # $t4 = 32'd1 since 14 < 15
slti $t5, $t3, -1       # $t5 = 32'd0 since 15 > -1
addi $t6, $t4, -2       # $t6 = 1 - 2 = -1
```

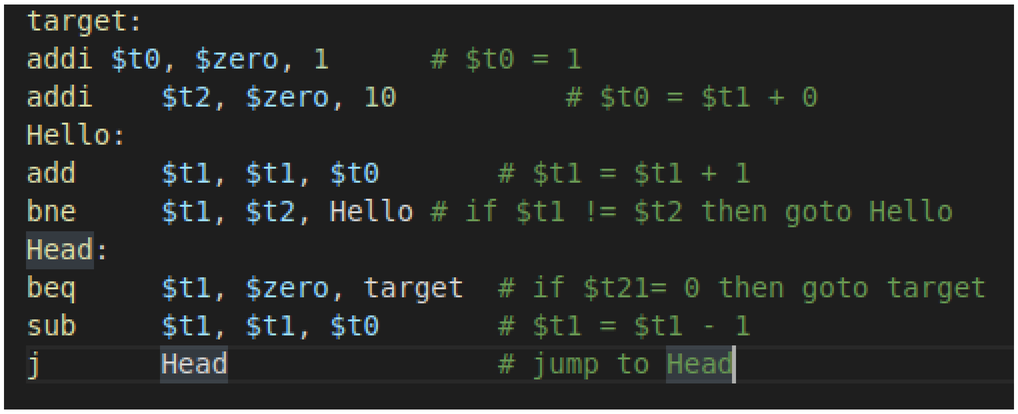
Similar to our R-type testing procedure, we made sure we had interdependencies in our instructions, using registers that were assigned values previously as inputs to new instructions. We once again made sure all control signals were satisfied at each step. We further made sure that the correct inputs were going into the ALU, that the PC was progressing as expected, and the ALU was outputting the expected result of the instruction. Instructions which sign extended the immediate (slti and addi) were tested with positive and negative immediates to confirm that the sign interpretation worked as expected.

jr testing:

```
addi $t0, $zero, 1       # $t0 = 1
sll  $s0, $t0, 22        # shift left to get $t1 = 0x00400000 (start addr)
addi $s0, $s0, 24         # $s0 = PC + 24
addi $t1, $t0, 1         # $t1 = 2
jr   $s0                 # jump to PC+24 (addr of last instruc)
addi $t1, $t1, 10        # $t1 = 12, BUT THIS IS SKIPPED
addi $t1, $t1, 1         # $t1 = 3,  IF PREV INSTR SKIPPED
```

In this testing code, we use addi and sll, which we are certain function as desired. As we can see, $s0 was assigned to the address of the final (6th) instruction (so we have PC + 4*6 = PC+24) as shown above. When jr is reached, it skips to the address in $s0, thereby skipping over the second to last instruction. The value of $t1 in the final instruction will be 3 if the second to last instruction is skipped (as it should be), whereas it will be 13 if the instruction is not skipped. By confirming that the ALU result for the final instruction was in fact 3, we are certain that jr skipped the second to last instruction.
(not shown) We also tested that jr works in the "backwards" direction, by setting jr's register to starting PC, and calling jr as the last instruction. This created an infinite loop (as expected) since every time the last register is reached we go back to the starting PC; the waveforms carried on until timeout of 1000ns.

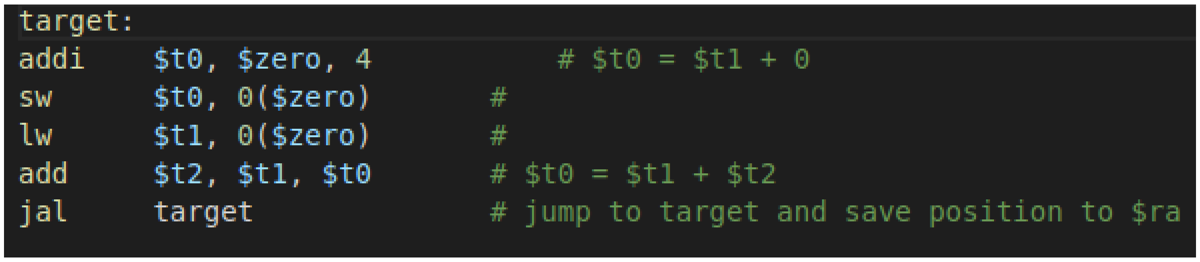
beq,bne, j testing:



```
target:
addi $t0, $zero, 1      # $t0 = 1
addi    $t2, $zero, 10          # $t0 = $t1 + 0
Hello:
add     $t1, $t1, $t0       # $t1 = $t1 + 1
bne     $t1, $t2, Hello # if $t1 != $t2 then goto Hello
Head:
beq     $t1, $zero, target  # if $t21= 0 then goto target
sub     $t1, $t1, $t0       # $t1 = $t1 - 1
j       Head                # jump to Head
```

In testing the branch functions, we created an infinite loop that iterated through values of 1-10. The first branch checks that the $t1 register is not equal to 10 If that condition is true it goes to “Hello” and adds 1 to $t1 until $t1 is equal to 10. The next branch instruction tests if $t1 is equal to 0. If that condition is not true then it subtracts 1 from $t1. The jump at the bottom continues to bring the program back to this condition. Once the condition is true, the program starts over again with $t1 = 0. Through this test, we can ensure that the conditions are branching when they are supposed to and not branching when they are not supposed to.

lw,sw, jal testing:



```
target:
addi    $t0, $zero, 4           # $t0 = $t1 + 0
sw      $t0, 0($zero)       #
lw      $t1, 0($zero)       #
add     $t2, $t1, $t0       # $t0 = $t1 + $t2
jal     target              # jump to target and save position to $ra
```

This testing uses the sw function to force store the four generated in the above line at an address in the data memory. Once this value is stored, the lw retrieves that value and puts it in a new register. Finally, the add function adds the two registers, both the retrieved and the original to get twice the original value. The jal function makes this a loop. Using the register file I can go in and check that the jal is linking the register correctly as well.