Assertions:

These assertions are inserted at the end of pipeline.sv. For testing the ALU, the assertion only gets checked if the alu_op is the correct one. A constrained random test generator was used to verify the ALU. For the control instructions, a direct test was used to see if the RISCV processor would jump to the correct address. These assertions first checks the PC select, then checks if the next address is the correct number. MUL and DIV have similar assertions to the ALU operations, but they operate on the MD module instead of the ALU. These assertions are checked with random tests.

```
// OR operation assertion
property or_op;
  @(posedge clk)
  (alu_op === `ALU_OP_OR) |-> (alu_out === (alu_src_a | alu_src_b));
endproperty
assert property (or_op) else $error("OR operation error");

// ADD operation assertion
property add_op;
  @(posedge clk)
  (alu_op === `ALU_OP_ADD) |-> (alu_out === (alu_src_a + alu_src_b));
endproperty
assert property (add_op) else $error("ADD operation error");

// AND operation assertion
property and_op;
  @(posedge clk)
  (alu_op === `ALU_OP_AND) |-> (alu_out === (alu_src_a & alu_src_b));
endproperty
assert property (and_op) else $error("AND operation error");

// XOR operation assertion
property xor_op;
  @(posedge clk)
  (alu_op === `ALU_OP_XOR) |-> (alu_out === (alu_src_a ^ alu_src_b));
endproperty
assert property (xor_op) else $error("XOR operation error");

// SLL operation assertion
property sll_op;
```

```
    @(posedge clk)
    (alu_op === `ALU_OP_SLL) |-> (alu_out === (alu_src_a << alu_src_b));
  endproperty
  assert property (sll_op) else $error("SLL operation error");

  // SLT operation assertion
  property slt_op;
    @(posedge clk)
    (alu_op === `ALU_OP_SLT) |-> (alu_out === {31'b0,$signed(alu_src_a) <
$signed(alu_src_b)});
  endproperty
  assert property (slt_op) else $error("SLT operation error");

  // SUB operation assertion
  property sub_op;
    @(posedge clk)
    (alu_op === `ALU_OP_SUB) |-> (alu_out === (alu_src_a - alu_src_b));
  endproperty
  assert property (sub_op) else $error("SUB operation error");

  // Branch operation assertion
  property branch_op;
    @(posedge clk)
    (PC_src_sel === `PC_BRANCH_TARGET) |-> (PC_PIF === (32'd56));
  endproperty
  assert property (branch_op) else $error("Branch operation error");

  // JAL operation assertion
  property jal_op;
    @(posedge clk)
    (PC_src_sel === `PC_JAL_TARGET) |-> (PC_PIF === (32'd12));
  endproperty
  assert property (jal_op) else $error("JAL operation error");

  // DIV operation assertion
  property div_op;
    @(posedge clk)
    (md_req_op === `MD_OP_DIV) |-> (md_resp_result === (rs1_data_bypassed /
rs2_data_bypassed));
```

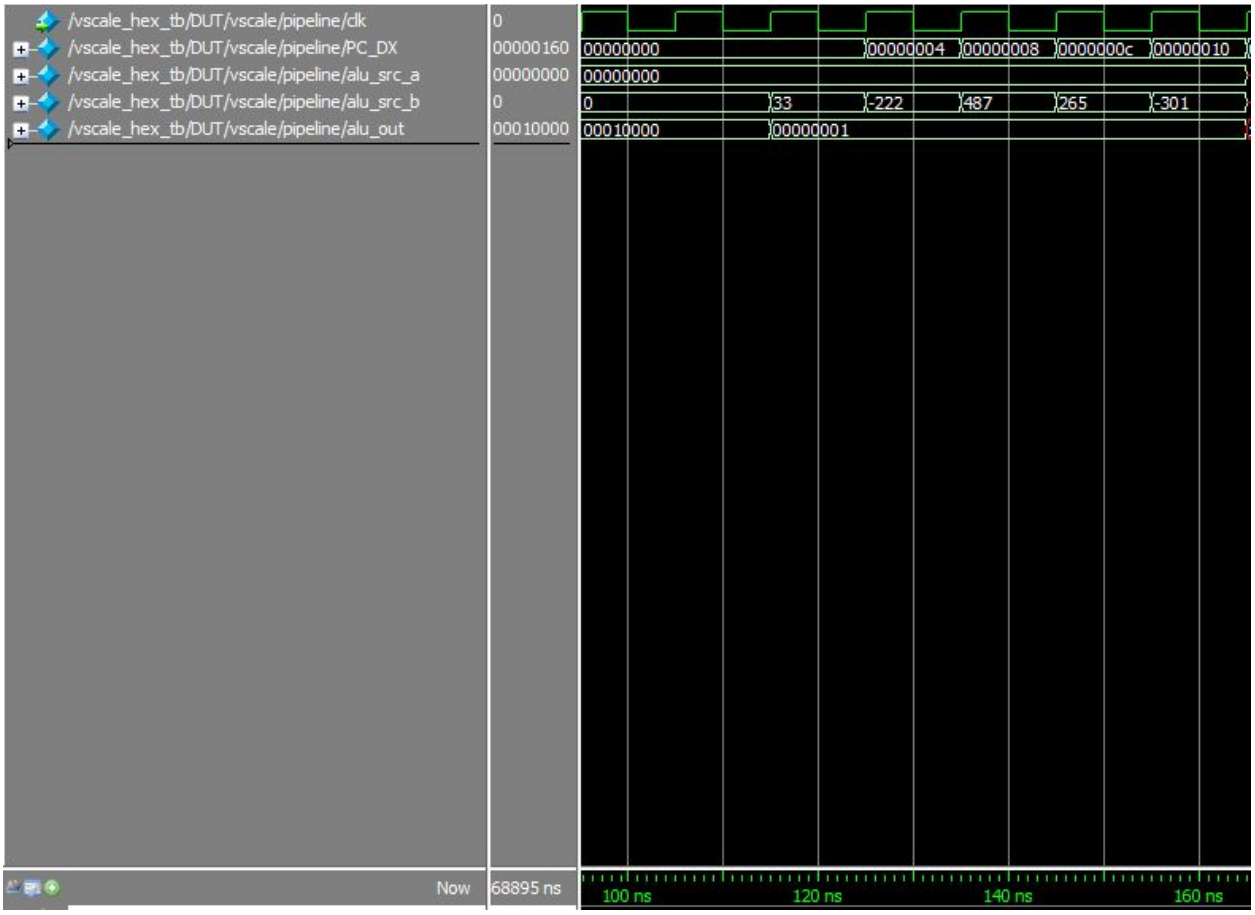
```
  endproperty
  assert property (div_op) else $error("DIV operation error");

  // MUL operation assertion
  property mul_op;
    @(posedge clk)
    (md_req_op === `MD_OP_MUL) |-> (md_resp_result === (rs1_data_bypassed *
rs2_data_bypassed));
  endproperty
  assert property (mul_op) else $error("MUL operation error");
```

Assembly Programs:

The operations in the ALU and the MD module use random tests with both positive and negative sign numbers to verify that that operation works. The control instructions use a direct test to see if the PC jumps to the correct address when the condition is met (Branch).

Bug 1: OR op in ALU

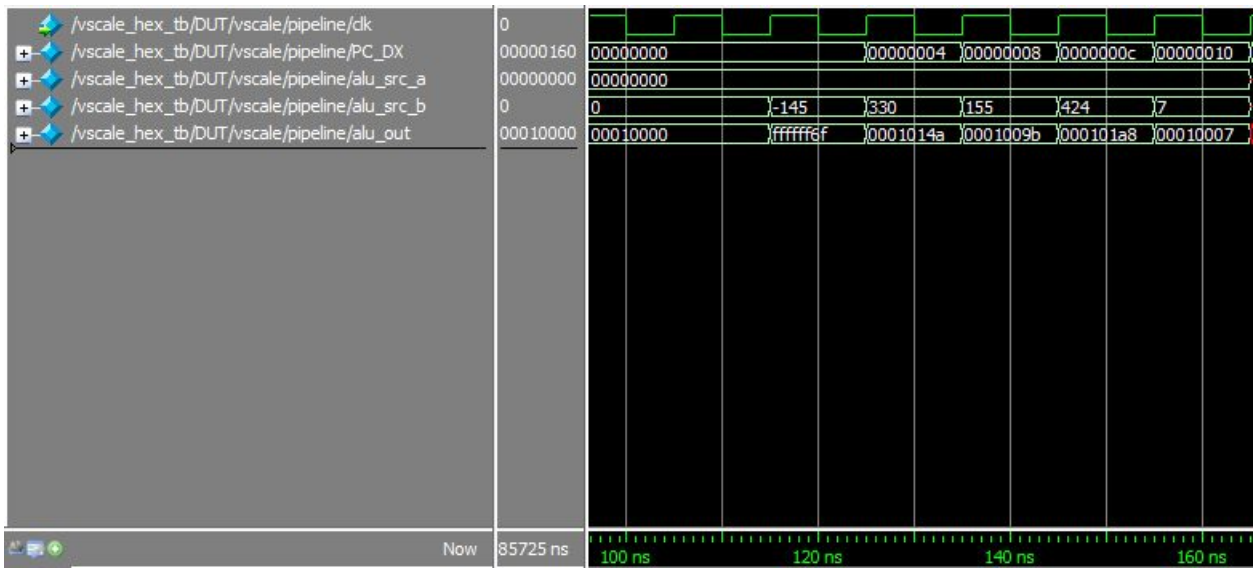
In alu.sv, OR function does not perform a bitwise OR but does a logical OR on the entire value. The ORI operation results in a 0 because both sources were zeroes. All other immediate values are not 0, so it results in a 1.



ori $t1, $zero, 33	02106313
ori $t2, $zero, -222	f2206393
ori $t1, $zero, 487	1e706313
ori $t1, $zero, 265	10906313
ori $t2, $zero, -301	ed306393

Bug 2: ADD op in ALU

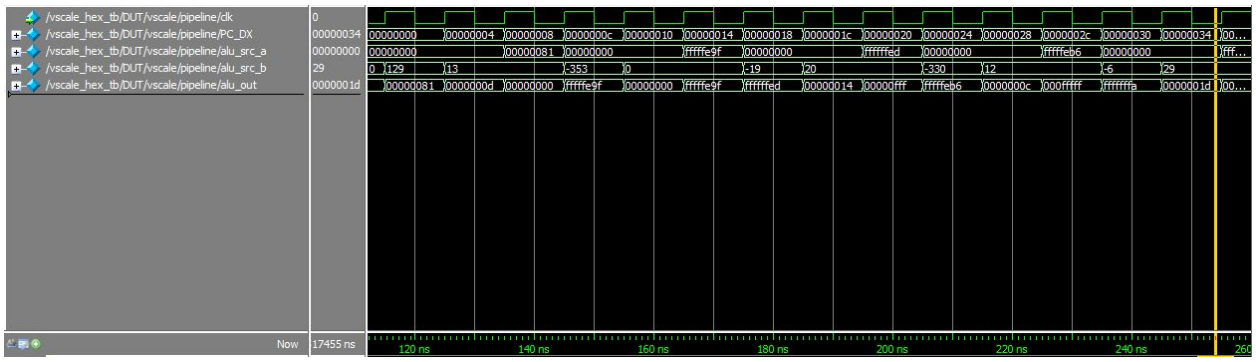
In alu.sv, ADD function first adds, then OR with 0x0010000.



addi $t2, $zero, -145 f6f00393
addi $t1, $zero, 330 14a00313
addi $t0, $zero, 155 09b00293
addi $t1, $zero, 424 1a800313
addi $t1, $zero, 7 00700313

Bug 3: SLL op in ALU

SLL shifts to the right instead of to the left.

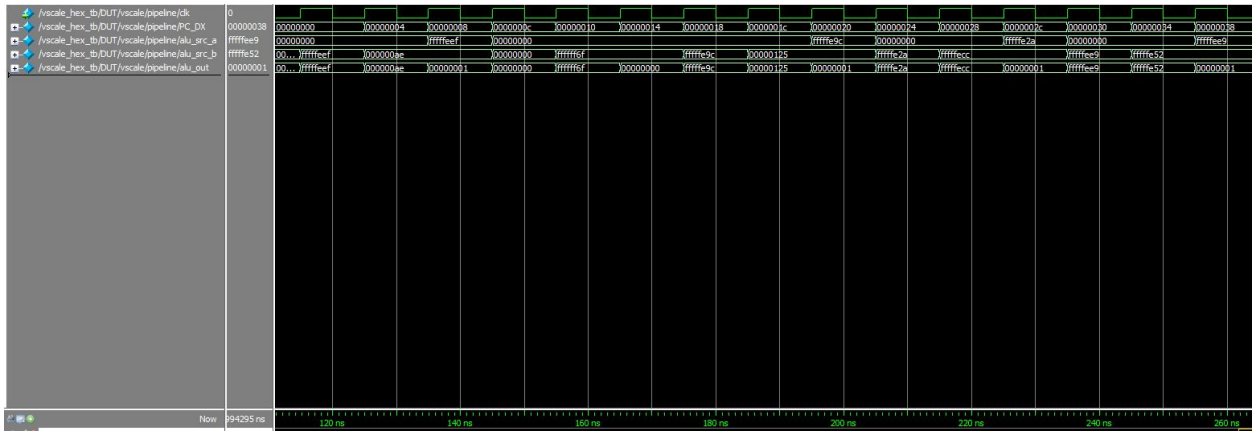


ori $t1, $zero, 129 08106313
ori $t2, $zero, 13 00d06393
sll $t0, $t1, $t2 007312b3
ori $t1, $zero, -353 e9f06313
ori $t2, $zero, 0 00006393
sll $t0, $t1, $t2 007312b3
ori $t1, $zero, -19 fed06313

ori $t2, $zero, 20	01406393
sll $t0, $t1, $t2	007312b3
ori $t1, $zero, -330	eb606313
ori $t2, $zero, 12	00c06393
sll $t0, $t1, $t2	007312b3
ori $t1, $zero, -6	ffa06313
ori $t2, $zero, 29	01d06393
sll $t0, $t1, $t2	007312b3

Bug 4: AND op in ALU

AND performs a logical AND on the entire value instead of a bitwise AND.



ori $t1, $zero, -273	eef06313
ori $t2, $zero, 174	0ae06393
and $t0, $t1, $t2	007372b3
ori $t1, $zero, 0	00006313
ori $t2, $zero, -145	f6f06393
and $t0, $t1, $t2	007372b3
ori $t1, $zero, -356	e9c06313
ori $t2, $zero, 293	12506393
and $t0, $t1, $t2	007372b3
ori $t1, $zero, -470	e2a06313
ori $t2, $zero, -308	ecc06393
and $t0, $t1, $t2	007372b3
ori $t1, $zero, -279	ee906313
ori $t2, $zero, -430	e5206393
and $t0, $t1, $t2	007372b3

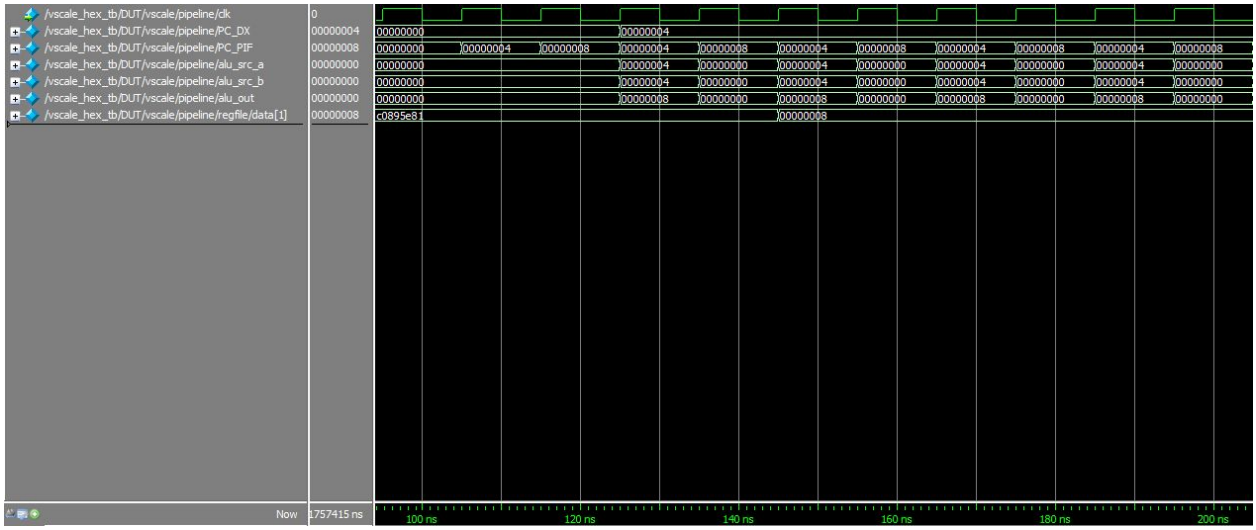
Bug 5: SUB op in ALU

SUB takes the result from the subtraction and ANDs it with 0xfffeffff

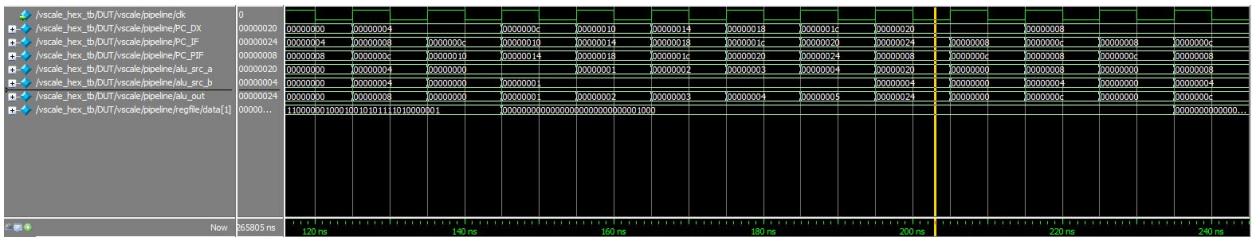
ori $t1, $zero, 212	0d406313
ori $t2, $zero, 253	0fd06393
sub $t0, $t1, $t2	407302b3
ori $t1, $zero, 141	08d06313
ori $t2, $zero, -156	f6406393
sub $t0, $t1, $t2	407302b3
ori $t1, $zero, 291	12306313
ori $t2, $zero, -258	efe06393
sub $t0, $t1, $t2	407302b3
ori $t1, $zero, 79	04f06313
ori $t2, $zero, -395	e7506393
sub $t0, $t1, $t2	407302b3
ori $t1, $zero, 271	10f06313
ori $t2, $zero, 360	16806393
sub $t0, $t1, $t2	407302b3

Bug 6: JAL and JR

JAL does not jump correctly. PC_DX does not get updated and gets stuck at the same value. $ra does hold the correct return address.



After the jal offset was fixed, PC_PIF correctly jumps back to $ra, then halts at address 8.

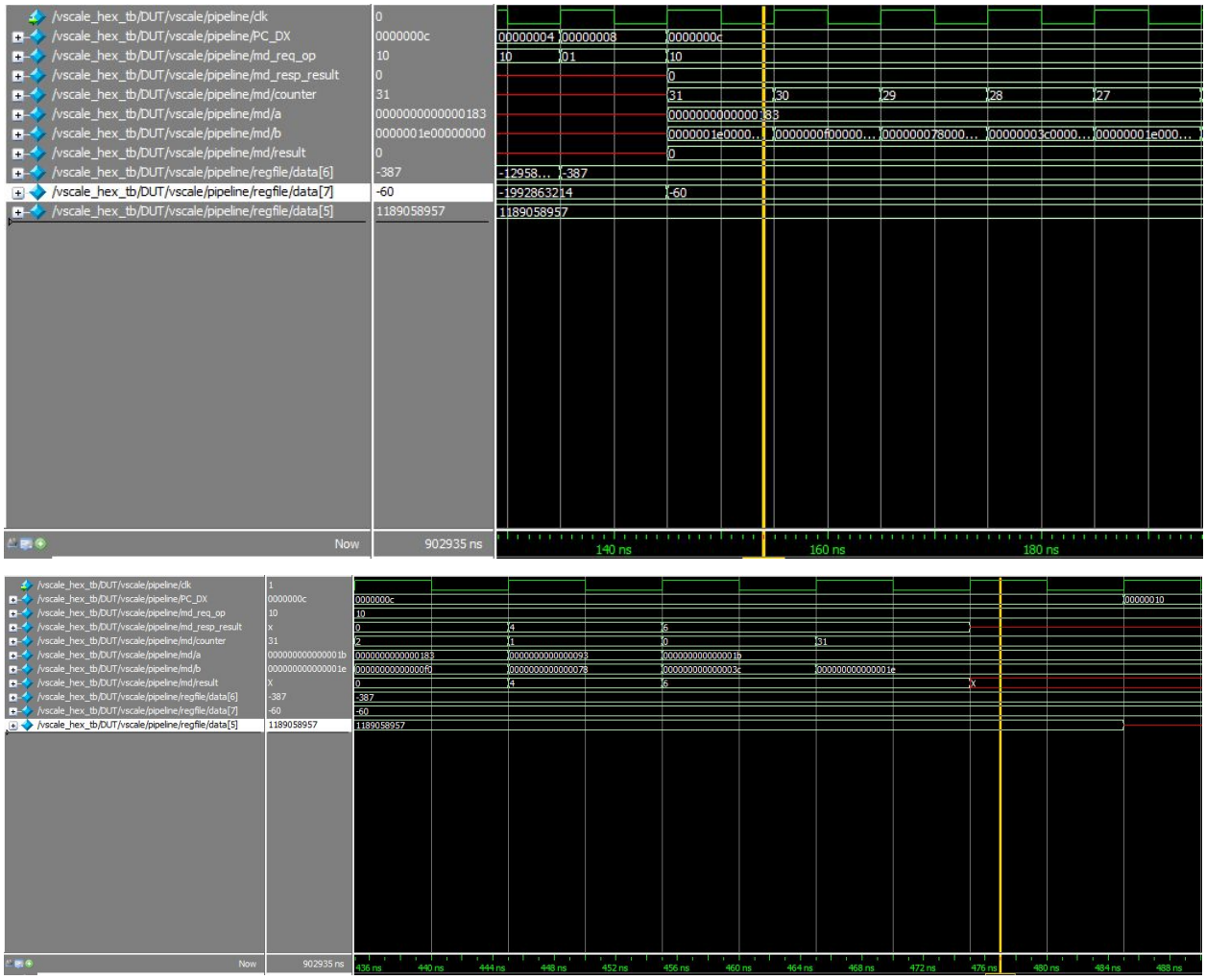


ori $t0, $zero, 0	00006293
jal $ra, there	008000ef
halt:	
jal $ra, halt	000000ef
there:	
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
jr $ra	00008067

Bug #7: DIV

At the start of the DIV operation, the value for B is divided by two, or shifted 1 place to the right. md_resp_result shows the correct answer before it becomes undefined. reg[6] and reg[7] are the registers (represented in decimal) DIV is operating on and it is trying to store floor(reg[6]/reg[7]) into reg[5]. The counter is also resetted back to 31 before the result was able

to stored. This mess up which state the module is in. The module does not stop and output the result when the counter is 0.

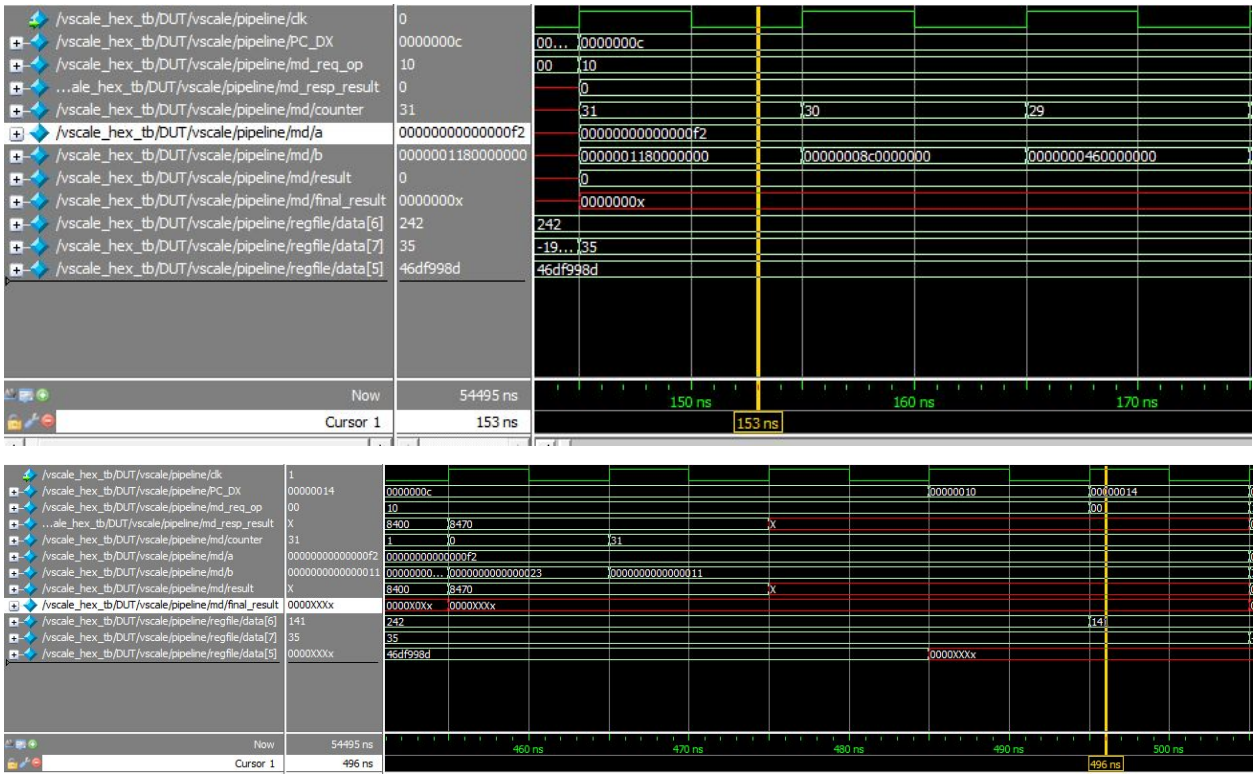


ori $t1, $zero, -387	e7d06313
ori $t2, $zero, -60	fc406393
div $t0, $t1, $t2	027342b3
ori $t1, $zero, 74	04a06313
ori $t2, $zero, -101	f9b06393
div $t0, $t1, $t2	027342b3
ori $t1, $zero, 378	17a06313
ori $t2, $zero, 183	0b706393
div $t0, $t1, $t2	027342b3
ori $t1, $zero, -255	f0106313
ori $t2, $zero, 188	0bc06393
div $t0, $t1, $t2	027342b3

ori $t1, $zero, -75 fb506313
ori $t2, $zero, -280 ee806393
div $t0, $t1, $t2 027342b3

Bug #8: MUL

Just like DIV, the starting value for B is shifted to the right one bit position. mp_resp_result holds the correct value when the counter reaches 0. This value is supposed to get stored into regfile[5], but after the counter resets to 5'b11111, wherever the result value is supposed to be a 1 becomes undefined. Final_result is always undefined, and is used when the module is in the output state. The MUL assertion is always triggered even when MUL is not executed because the default operation is multiply.



ori $t1, $zero, 242 0f206313
ori $t2, $zero, 35 02306393
mul $t0, $t1, $t2 027302b3
ori $t1, $zero, 141 08d06313
ori $t2, $zero, -142 f7206393
mul $t0, $t1, $t2 027302b3
ori $t1, $zero, 287 11f06313
ori $t2, $zero, 112 07006393
mul $t0, $t1, $t2 027302b3

ori $t1, $zero, 315	13b06313
ori $t2, $zero, -217	f2706393
mul $t0, $t1, $t2	027302b3
ori $t1, $zero, -124	f8406313
ori $t2, $zero, 369	17106393
mul $t0, $t1, $t2	027302b3

Unsuccessful Tests

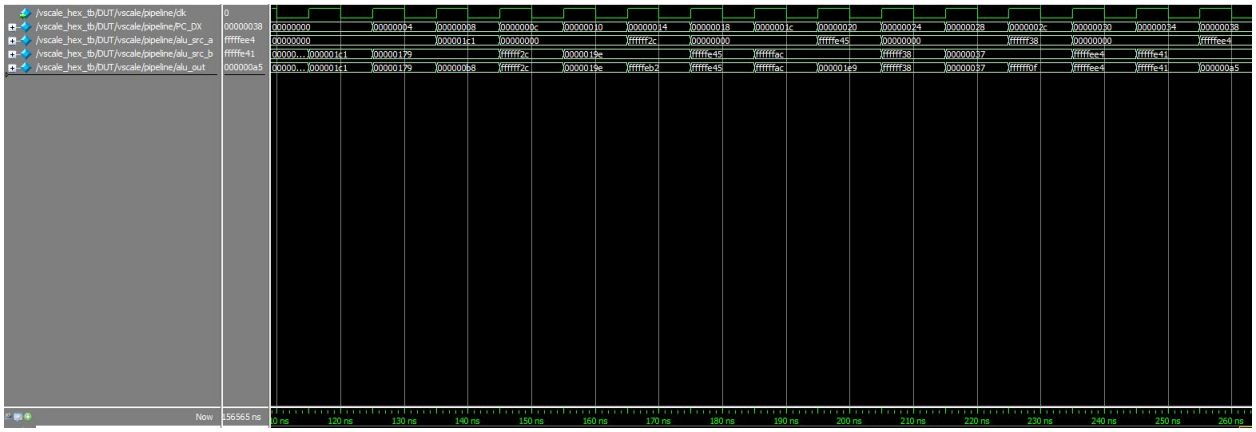
1: SLT op in ALU

The ALU correctly outputs if a < b for both positive and negative numbers.

ori $t1, $zero, 221	0dd06313
ori $t2, $zero, 75	04b06393
slt $t0, $t1, $t2	007322b3
ori $t1, $zero, -158	f6206313
ori $t2, $zero, 494	1ee06393
slt $t0, $t1, $t2	007322b3
ori $t1, $zero, 440	1b806313
ori $t2, $zero, -299	ed506393
slt $t0, $t1, $t2	007322b3
ori $t1, $zero, 432	1b006313
ori $t2, $zero, 257	10106393
slt $t0, $t1, $t2	007322b3
ori $t1, $zero, 395	18b06313
ori $t2, $zero, 274	11206393
slt $t0, $t1, $t2	007322b3

2: XOR op in ALU

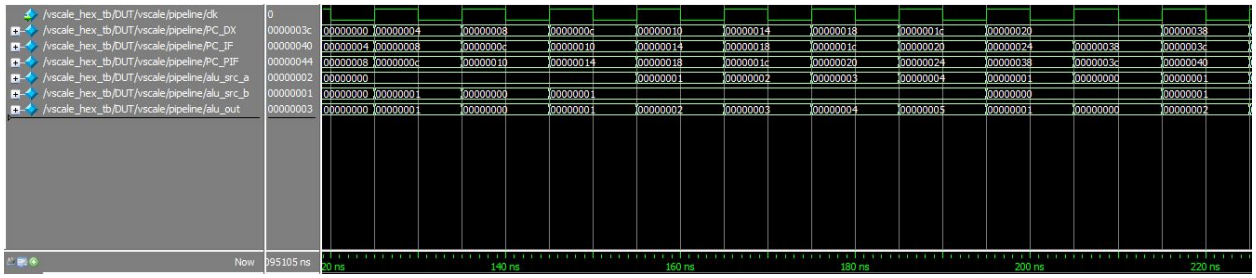
The ALU successfully performs a bitwise XOR on the two ALU sources.

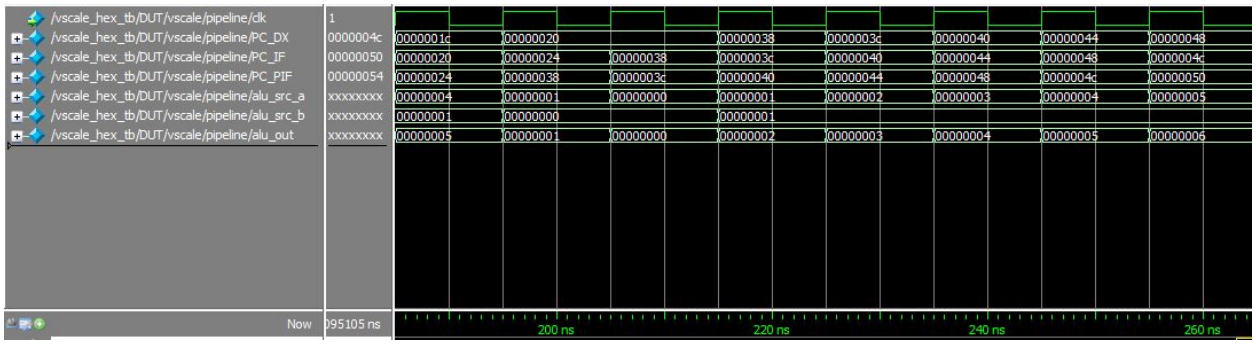


ori $t1, $zero, 449	1c106313
ori $t2, $zero, 377	17906393
xor $t0, $t1, $t2	007342b3
ori $t1, $zero, -212	f2c06313
ori $t2, $zero, 414	19e06393
xor $t0, $t1, $t2	007342b3
ori $t1, $zero, -443	e4506313
ori $t2, $zero, -84	fac06393
xor $t0, $t1, $t2	007342b3
ori $t1, $zero, -200	f3806313
ori $t2, $zero, 55	03706393
xor $t0, $t1, $t2	007342b3
ori $t1, $zero, -284	ee406313
ori $t2, $zero, -447	e4106393
xor $t0, $t1, $t2	007342b3

3. BNE

The program does not branch to L1 because $t0 = $0. Then, it successfully branches to the correct address when the second branch statement is called.





ori $t0, $zero, 0	00006293
ori $t1, $zero, 1	00106313
bne $t0, $zero, L1	00029c63
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
addi $t0, $t0, 1	00128293
L1:	
bne $t1, $zero, L2	00031c63
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
L2:	
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313
addi $t1, $t1, 1	00130313