Kenny Zhang

DeLong

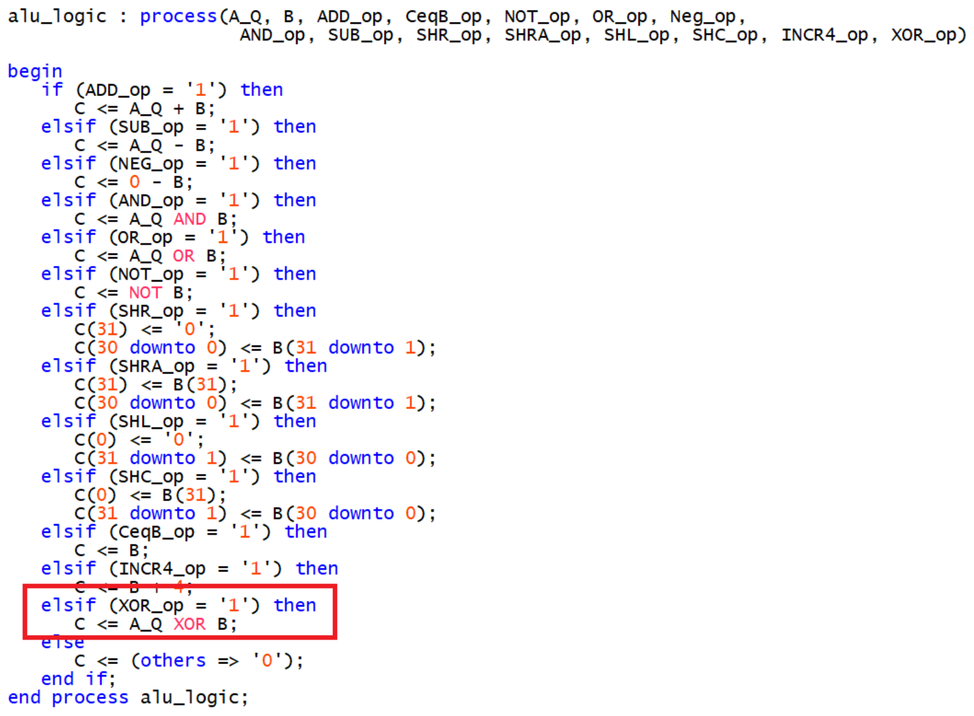
ECE 4435: Computer Architecture and Design

May 1, 2023

Learning Activity 7

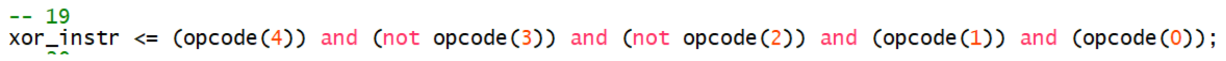
At the end of the last learning activity, students successfully completed the design of the Basic RISC CPU (BRC). Therefore, in this final learning activity, students were required to extend the instruction set of the BRC to include the XOR operation. Additionally, students were tasked with generating comprehensive simulation results that verified the complete functionality of the XOR ISA extension.

Because XOR is a logical operation, the first step is to modify the ALU to support XOR instructions. This can simply be done by appending another condition to the alu_logic process. If the XOR_op control signal input equals 1, set the C = A XOR B.



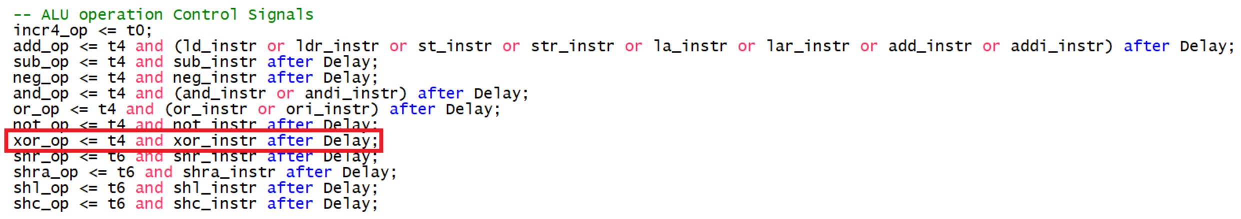
```
alu_logic : process(A_Q, B, ADD_op, CeqB_op, NOT_op, OR_op, Neg_op,
                    AND_op, SUB_op, SHR_op, SHRA_op, SHL_op, SHC_op, INCR4_op, XOR_op)

begin
   if (ADD_op = '1') then
      C <= A_Q + B;
   elsif (SUB_op = '1') then
      C <= A_Q - B;
   elsif (NEG_op = '1') then
      C <= 0 - B;
   elsif (AND_op = '1') then
      C <= A_Q AND B;
   elsif (OR_op = '1') then
      C <= A_Q OR B;
   elsif (NOT_op = '1') then
      C <= NOT B;
   elsif (SHR_op = '1') then
      C(31) <= '0';
      C(30 downto 0) <= B(31 downto 1);
   elsif (SHRA_op = '1') then
      C(31) <= B(31);
      C(30 downto 0) <= B(31 downto 1);
   elsif (SHL_op = '1') then
      C(0) <= '0';
      C(31 downto 1) <= B(30 downto 0);
   elsif (SHC_op = '1') then
      C(0) <= B(31);
      C(31 downto 1) <= B(30 downto 0);
   elsif (CeqB_op = '1') then
      C <= B;
   elsif (INCR4_op = '1') then
      C <= B + 4;
   elsif (XOR_op = '1') then
      C <= A_Q XOR B;
   else
      C <= (others => '0');
   end if;
end process alu_logic;
```

Next, the control signals logic and opcode decoder need to be modified to support the addition of a XOR_op control signal feeding into the ALU. First, some combinational logic is added to the opcode decoder to support decoding the XOR instruction (opcode = 19).



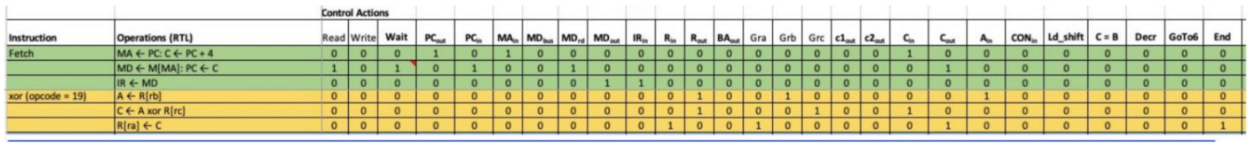
```
-- 19
xor_instr <= (opcode(4)) and (not opcode(3)) and (not opcode(2)) and (opcode(1)) and (opcode(0));
```

This allows the opcode decoder component in the control signals logic to output a signal indicating when a XOR instruction is called. In the control signals logic, the conditions under which XOR_op = 1 are the same as the other ALU control signals.



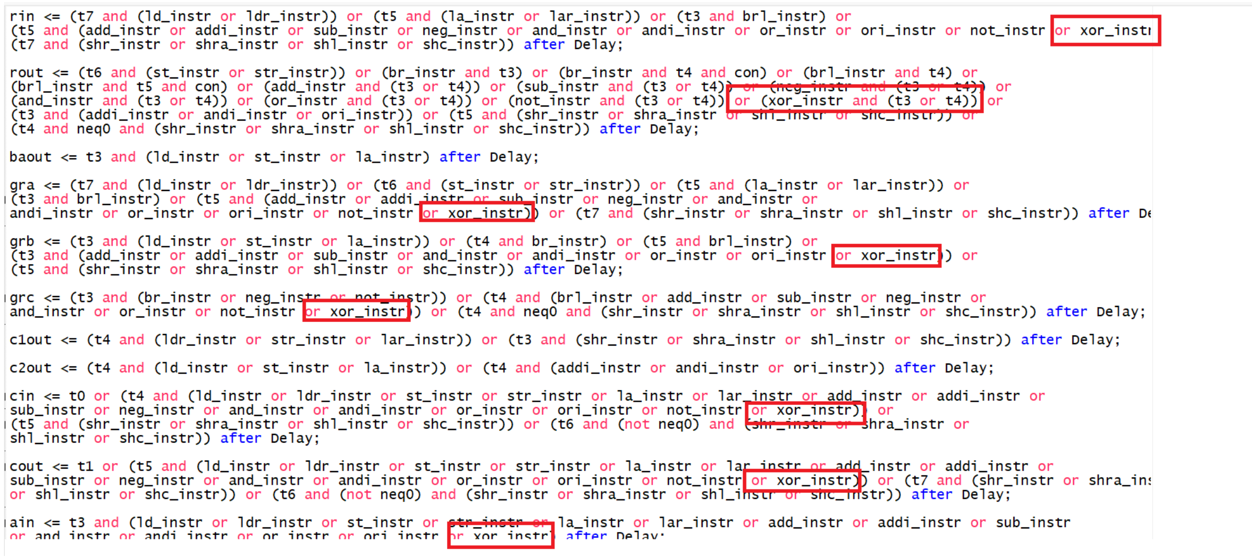
```
-- ALU operation Control Signals
incr4_op <= t0;
add_op <= t4 and (ld_instr or ldr_instr or st_instr or str_instr or la_instr or lar_instr or add_instr or addi_instr) after Delay;
sub_op <= t4 and sub_instr after Delay;
neg_op <= t4 and neg_instr after Delay;
and_op <= t4 and (and_instr or andi_instr) after Delay;
or_op <= t4 and (or_instr or ori_instr) after Delay;
not_op <= t4 and not_instr after Delay;
xor_op <= t4 and xor_instr after Delay;
shr_op <= t6 and shr_instr after Delay;
shra_op <= t6 and shra_instr after Delay;
shl_op <= t6 and shl_instr after Delay;
shc_op <= t6 and shc_instr after Delay;
```

In addition to providing combinational logic for XOR_op, some of the other control signal assignments need to be modified. The following shows the microinstructions for the XOR operation.



		Control Actions																										
Instruction	Operations (RTL)	Read	Write	Wait	PC_{out}	PC_{in}	MA_{in}	MD_{bus}	MD_{rd}	MD_{out}	IR_{in}	R_{in}	R_{out}	BA_{out}	Gra	Grb	Grc	$c1_{out}$	$c2_{out}$	C_{in}	C_{out}	A_{in}	CON_{in}	Ld_shift	C = B	Decr	GoTo6	End
Fetch	MA ← PC: C ← PC + 4	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	MD ← M[MA]: PC ← C	1	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	IR ← MD	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
xor (opcode = 19)	A ← R[rb]	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
	C ← A xor R[rc]	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
	R[ra] ← C	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1

As indicated in the table above, the following control signals should be active when XOR_inst equals 1: rin, rout, gra, grb, grc, cin, cout, ain, and end. To accomplish this, we can simply append a condition for XOR_inst to the existing logic, since these signals follow a pattern for the other logical operations.



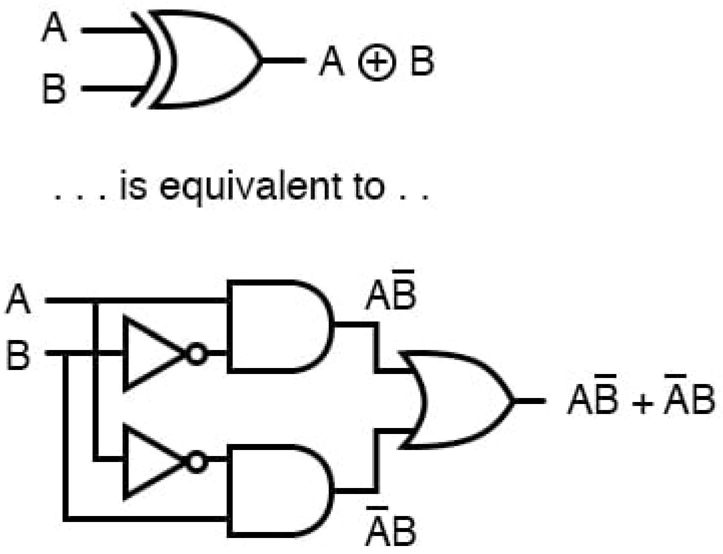
```
rin <= (t7 and (ld_instr or ldr_instr)) or (t5 and (la_instr or lar_instr)) or (t3 and brl_instr) or
(t5 and (add_instr or addi_instr or sub_instr or neg_instr or and_instr or andi_instr or or_instr or ori_instr or not_instr or xor_instr
(t7 and (shr_instr or shra_instr or shl_instr or shc_instr)) after Delay;

rout <= (t6 and (st_instr or str_instr)) or (br_instr and t3) or (br_instr and t4 and con) or (brl_instr and t4) or
(brl_instr and t5 and con) or (add_instr and (t3 or t4)) or (sub_instr and (t3 or t4)) or (neg_instr and (t3 or t4)) or
(and_instr and (t3 or t4)) or (or_instr and (t3 or t4)) or (not_instr and (t3 or t4)) or (xor_instr and (t3 or t4)) or
(t3 and (addi_instr or andi_instr or ori_instr)) or (t5 and (shr_instr or shra_instr or shl_instr or shc_instr)) or
(t4 and neq0 and (shr_instr or shra_instr or shl_instr or shc_instr)) after Delay;

baout <= t3 and (ld_instr or st_instr or la_instr) after Delay;

gra <= (t7 and (ld_instr or ldr_instr)) or (t6 and (st_instr or str_instr)) or (t5 and (la_instr or lar_instr)) or
(t3 and brl_instr) or (t5 and (add_instr or addi_instr or sub_instr or neg_instr or and_instr or
andi_instr or or_instr or ori_instr or not_instr or xor_instr)) or (t7 and (shr_instr or shra_instr or shl_instr or shc_instr)) after De

grb <= (t3 and (ld_instr or st_instr or la_instr)) or (t4 and br_instr) or (t5 and brl_instr) or
(t3 and (add_instr or addi_instr or sub_instr or and_instr or andi_instr or or_instr or ori_instr or xor_instr)) or
(t5 and (shr_instr or shra_instr or shl_instr or shc_instr)) after Delay;

grc <= (t3 and (br_instr or neg_instr or not_instr)) or (t4 and (brl_instr or add_instr or sub_instr or neg_instr or
and_instr or or_instr or not_instr or xor_instr)) or (t4 and neq0 and (shr_instr or shra_instr or shl_instr or shc_instr)) after Delay;

c1out <= (t4 and (ldr_instr or str_instr or lar_instr)) or (t3 and (shr_instr or shra_instr or shl_instr or shc_instr)) after Delay;

c2out <= (t4 and (ld_instr or st_instr or la_instr)) or (t4 and (addi_instr or andi_instr or ori_instr)) after Delay;

cin <= t0 or (t4 and (ld_instr or ldr_instr or st_instr or str_instr or la_instr or lar_instr or add_instr or addi_instr or
sub_instr or neg_instr or and_instr or andi_instr or or_instr or ori_instr or not_instr or xor_instr)) or
(t5 and (shr_instr or shra_instr or shl_instr or shc_instr)) or (t6 and (not neq0) and (shr_instr or shra_instr or
shl_instr or shc_instr)) after Delay;

cout <= t1 or (t5 and (ld_instr or ldr_instr or st_instr or str_instr or la_instr or lar_instr or add_instr or addi_instr or
sub_instr or neg_instr or and_instr or andi_instr or or_instr or ori_instr or not_instr or xor_instr)) or (t7 and (shr_instr or shra_ins
or shl_instr or shc_instr)) or (t6 and (not neq0) and (shr_instr or shra_instr or shl_instr or shc_instr)) after Delay;

ain <= t3 and (ld_instr or ldr_instr or st_instr or str_instr or la_instr or lar_instr or add_instr or addi_instr or sub_instr
or and_instr or andi_instr or or_instr or ori_instr or xor_instr) after Delay;
```

The last step is to simply modify the I/O for the larger Datapath and Control Unit schematics to include the addition of the XOR_op opcode.

In terms of verification, a different approach was taken in this LA compared to LA 6. To verify the functionality of the XOR operation, the following assembly program was used over multiple test cases.

```
data1:      .equ  170         ; first data (A)
data2:      .equ  85          ; second data (B)

      .org  0           ; 0x0000 - start of ROM
      la    r1, data1   ; Load first data
      la    r2, data2   ; Load second data
      xor   r3, r1, r2  ; xor two data items together
      st    r3, Result1 ; store result
      la    r4, data1   ; Load first data (r4 = A)
      la    r5, data2   ; Load second data (r5 = B)
      not   r6, r5      ; r6 = NOT(B)
      not   r7, r4      ; r7 = NOT(A)
      and   r4, r4, r6  ; r4 = (A AND NOT(B))
      and   r5, r7, r5  ; r5 = (NOT(A) AND B)
      or    r4, r4, r5  ; r4 = (A AND NOT(B)) OR (NOT(A) AND B) = A XOR B
      st    r4, Result2 ; store result
      stop

      .org  32772       ; 0x8004 - location in RAM
Result1:    .dw   1           ; storage for Result1
Result2:    .dw   1           ; storage for Result2
```

Because the XOR operation can be fundamentally constructed using AND, OR, and NOT, this approach uses two different techniques to perform a XOR on constants (A=data1 and B=data2), then checks whether the results of both are equal. First, the program computes A XOR B directly using the appended XOR instruction. A and B are loaded into r1 and r2 and the program simply performs r3 = r1 XOR r2. In digital logic, a XOR gate can be constructed using the following combination of AND, OR, and NOT gates



$$A \oplus B = A\bar{B} + \bar{A}B$$

Thus, the second part of the program computes XOR through a sequence of AND, OR, and NOT operations. First, A and B along with their complements are loaded into r4 (A), r5 (B), r6 (NOT(B)) and r7 (NOT(A)). Next, the AND operations are performed and stored back into r4 and r5. Finally, the OR operation is performed, with the result stored in r4. After this, the same procedure is used to modify and execute the resulting binary file on the BRC. Note that because the Java simulator does not support XOR, the binary for this assembly is generated using AND as a placeholder in the first step. Then, the binary is directly modified to perform a XOR instruction on r1 and r2.

The table below describes the three test cases used within the assembly program to verify the functionality of the XOR operation.

Test Number	Inputs (binary)	Inputs (32-bit Hex)	Expected Outputs

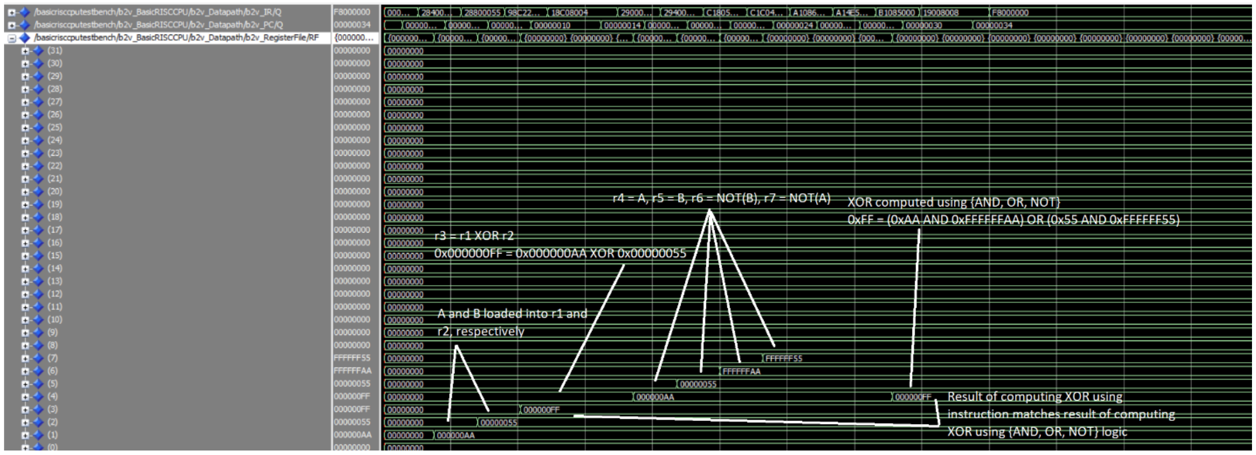
1	A = 10101010
	B = 01010101
		A = 0x000000AA
		B = 0x00000055
			Binary: 11111111
			32-bit Hex: 0x000000FF

2	A = 11111111
	B = 11111111
		A = 0x000000FF
		B = 0x000000FF
			Binary: 00000000
			32-bit Hex:
			0x00000000

3	A = 10100110
	B = 00111011
		A = 0x000000A6
		B = 0x0000003B
			Binary: 10011101
			32-bit Hex:
			0x0000009D

The following waveforms depict the Modelsim output from running these three test cases. In each waveform, both methods of computing XOR (XOR instruction and {AND, OR, NOT} combination) produce the expected results. Thus, we can conclude that the XOR instruction is properly appended to the BRC Instruction Set.

Test Case 1 (A = 0x000000AA, B = 0x00000055)



r4 = A, r5 = B, r6 = NOT(B), r7 = NOT(A)

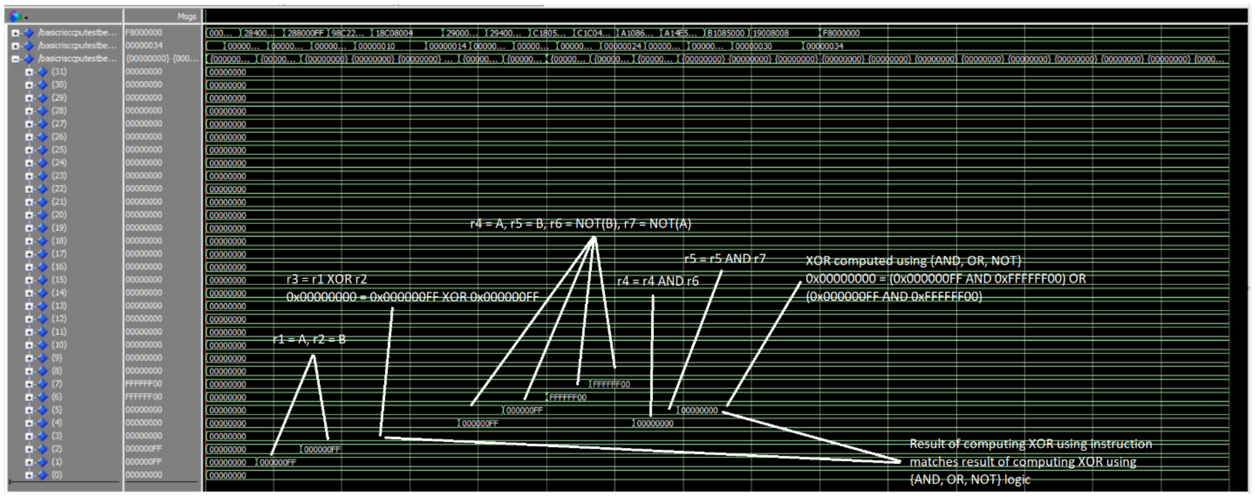
XOR computed using {AND, OR, NOT}
0xFF = (0xAA AND 0xFFFFFFAA) OR (0x55 AND 0xFFFFFF55)

r3 = r1 XOR r2
0x000000FF = 0x000000AA XOR 0x00000055

A and B loaded into r1 and r2, respectively

Result of computing XOR using instruction matches result of computing XOR using {AND, OR, NOT} logic

Test Case 2 (A = 0x000000FF, B = 0x000000FF)



r4 = A, r5 = B, r6 = NOT(B), r7 = NOT(A)

r5 = r5 AND r7

XOR computed using {AND, OR, NOT}
0x00000000 = (0x000000FF AND 0xFFFFFF00) OR
(0x000000FF AND 0xFFFFFF00)

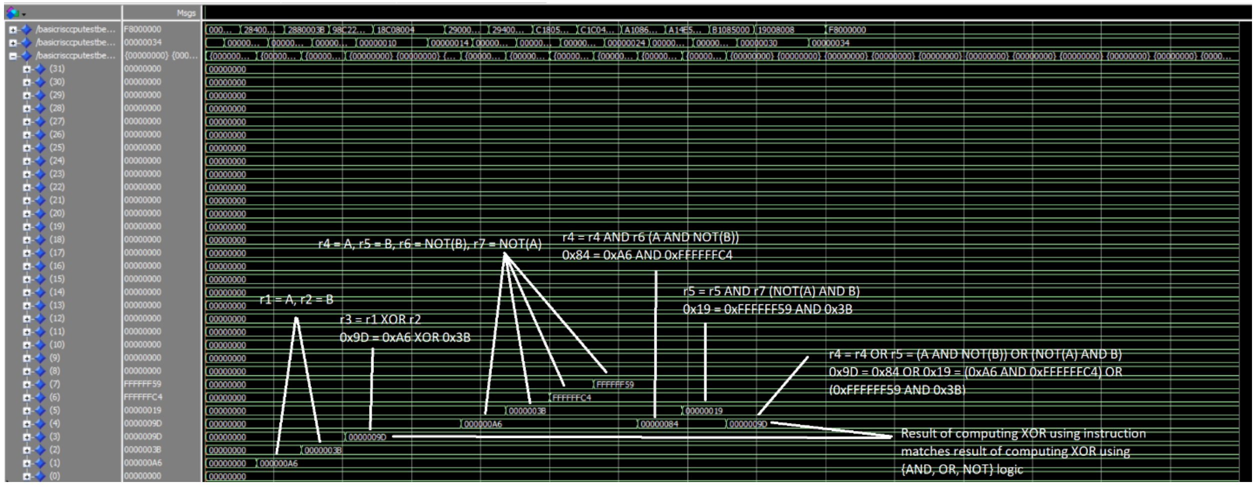
r3 = r1 XOR r2
0x00000000 = 0x000000FF XOR 0x000000FF

r4 = r4 AND r6

r1 = A, r2 = B

Result of computing XOR using instruction
matches result of computing XOR using
{AND, OR, NOT} logic

Test Case 3 (A = 0x000000A6, B = 0x00000036)



r4 = A, r5 = B, r6 = NOT(B), r7 = NOT(A)

r4 = r4 AND r6 (A AND NOT(B))
0x84 = 0xA6 AND 0xFFFFFFC4

r5 = r5 AND r7 (NOT(A) AND B)
0x19 = 0xFFFFFF59 AND 0x3B

r1 = A, r2 = B

r3 = r1 XOR r2
0x9D = 0xA6 XOR 0x3B

r4 = r4 OR r5 = (A AND NOT(B)) OR (NOT(A) AND B)
0x9D = 0x84 OR 0x19 = (0xA6 AND 0xFFFFFFC4) OR
(0xFFFFFF59 AND 0x3B)

Result of computing XOR using instruction
matches result of computing XOR using
{AND, OR, NOT} logic