Solutions

In this problem set we are going to study the synthesis of sequential circuits as well as sequential equivalence checking and SAT using ABC. We will use a simple sequence detector FSM. To make your life happier, we’ve already synthesized the circuit from RTL code into the gate-level code. This is the code for sequence detector. You can also download it here. For reading this in ABC you need the cadence.genlib standard cell library.

```verilog
module top ( clock, rst, inp, outp );
    input clock, rst, inp;
    output outp;
    wire N12, N13, n10, n11, n12, n13, n14, n15, n16;
    reg [1:0] state;

    always @ (posedge clock) begin
        state[0] <= N12;
        state[1] <= N13;
        outp <= n10;
    end
    initial begin
        state[0] <= 1'b0;
        state[1] <= 1'b0;
        outp <= 1'b0;
    end

    OAI21X1 U14 ( .A0(n11), .A1(n12), .B0(n13), .Y(n10) );
    NAND2X1 U15 ( .A(rst), .B(outp), .Y(n13) );
    NAND2X1 U16 ( .A(state[0]), .B(N12), .Y(n12) );
    INVX1 U17 ( .A(state[1]), .Y(n11) );
    MX2X1 U18 ( .B(n14), .A(n15), .S0(state[0]), .Y(N13) );
    OR2X1 U19 ( .A(inp), .B(rst), .Y(n15) );
    NAND2X1 U20 ( .A(state[1]), .B(N12), .Y(n14) );
    NOR2X1 U21 ( .A(n16), .B(rst), .Y(N12) );
    INVX1 U22 ( .A(inp), .Y(n16) );
endmodule
```
1 Sequential Synthesis [50 points]

Read the netlist `top_syn.v` in ABC and synthesize it. Use the `cadence.genlib` library as the link and target library\(^1\). Each of these problems are separate. For each problem, first reload and resynthesizes the circuit.

The code in the figure above is the synthesized version of the sequencer circuit shown below. We have already synthesized this using a 0.25um OSU standard cell library using Design Compiler.

```verilog
module m101( clk, rst, inp, outp);
  input clk, rst, inp;
  output outp;

  reg [1:0] state;
  reg outp;

  always @( posedge clk )
  begin
    if( rst )
      state <= 2’b00;
    else
      begin
        case( {state,inp} )
          3’b000: begin
            state <= 2’b00;
            outp <= 0;
          end
          3’b001: begin
            state <= 2’b01;
            outp <= 0;
          end
          3’b010: begin
            state <= 2’b10;
            outp <= 0;
          end
          3’b011: begin
            state <= 2’b01;
            outp <= 0;
          end
          3’b100: begin
            state <= 2’b00;
            outp <= 0;
          end
        endcase
      end
  end
endmodule
```

\(^1\)Link library means that the library you use for loading the netlist into ABC. Target library means the library you use for mapping the synthesized netlist into. In practice, most of the times these libraries are different.
```verilog
outp <= 0;
end
3'b110: begin
state <= 2'b10;
outp <= 0;
end
3'b111: begin
state <= 2'b01;
outp <= 1;
end
endcase
derend
endmodule
```

- Show and print the sequential AIG graph. What is the difference between this graph and the combinational AIG graph you saw in the homework 5?

  **Solution:** Figure 1 shows the sequential AIG graph of sequence detector. In comparison to the combinational AIG graph, there are multiple latches at both the inputs and outputs of the AIG graph. Notice the `< outp, outpL, outpL_in >, < state[1]L, state[1]L_in >` and `< state[2]L, state[2]L_in >` nodes in the graph.

```
abc 01> read_library cadence.genlib
Entered genlib library with 28 gates from file "cadence.genlib".
abc 01> read -m SeqDet.v
abc 02> strash
abc 03> show
```

- Synthesize and map the circuit. Show the circuit graph with gate-level netlist (use `show -g`).

  **Solution:** To synthesize the circuit, we use `dc2` script with no other optimization. Because we want the gate level netlist, we also need to map the circuit. Figure 2 shows the circuit graph of the sequencer circuit.

  AIG graph of the circuit is a graph where every node is an AND gate and every edge is either normal or inverted. When we use `show` command in AIG mode, ABC produces the AIG graph. On the other hand, the circuit graph is a graph where every node is a logic gate and every edge is a wire. When in logic mode (either by synthesizing the circuit or using `logic` command) ABC generates the circuit graph. If the circuit is unmapped, ABC is going to put a lookup table in place of each gate, but after mapping it is going to select the specific logic gate to perform that functionality.

```
abc 01> read_library cadence.genlib
Entered genlib library with 28 gates from file "cadence.genlib".
abc 01> read -m SeqDet.v
abc 02> strash
abc 03> dc2
```
• What is the area and the delay of the circuit after initial synthesis (dc2)? How many levels does the circuit have? What is the maximum clock frequency of the circuit assuming that delay unit is in us? **solutions:** We can use `print_stats` to determine various statistics about the circuit. The output of `print_stats` varies depending on which mode of the ABC we are in. When in AIG mode, it will tell you how many and gate and levels the circuit has. After logic, it will tell you the number of gates and edges in the circuit graph. After mapping, it will tell you the delay and the area of the circuit.

<table>
<thead>
<tr>
<th>Mode</th>
<th>I/O</th>
<th>Latency</th>
<th>AND</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIG</td>
<td>3/1</td>
<td>3</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>dc2</td>
<td>3/1</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Logic</td>
<td>3/1</td>
<td>3</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td>Mapped</td>
<td>3/1</td>
<td>3</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bdd</td>
<td>19</td>
</tr>
</tbody>
</table>

The circuit has the area of 11 and the delay of 3us, so the clock frequency of the circuit is \( \frac{1}{3\text{us}} = 333\text{kHz} \).

abc 01> read_library cadence.genlib
Entered genlib library with 28 gates from file "cadence.genlib".
abc 01> read -m SeqDet.v
abc 02> strash
abc 03> dc2
abc 04> map
abc 05> print_stats

• Show the BDD graphs for nodes `outp`, and registers `state[0]` and `state[1]`. (Use `print_io` in order to determine the node number for those POs). **solutions:** First, we have to determine what is the node number for nodes `outp`, and registers `state[0]` and `state[1]`. Next, we can draw the BDD graphs by using `show_bdd` command. Figure 3-5 shows the BDD graphs for the output.

abc 10> print_io
Primary inputs (3): clock rst inp
Primary outputs (1): outp
Latches (3): state[0]L(state[0]=n20) state[1]L(state[1]=n23) outpL(outp=n26)

abc 11> show_bdd n20
abc 12> show_bdd n23
abc 13> show_bdd n26

• Collapse the circuit (use `collapse`). Use `show -g` to view the results. What happened here? What is the trade of collapse? Can you map the collapsed circuit? Why? **solution:**
Collapsing the circuit will merge every gate to one gate per output. So the circuit will have only 1 level but with very huge gates. These super gates are bigger, more complex (which means they are more expensive to be implemented) and require more area to be implemented. On the other hand the general delay of the circuit is possibly reduced. We cannot map those gates because we don’t have an equivalent to them in the standard cell library. One application of such gate is in FPGAs where you can implement any gate in the look-up tables. Figure 6 shows the collapsed circuit.

abc 01> read_library cadence.genlib
Entered genlib library with 28 gates from file "cadence.genlib".
abc 01> read -m SeqDet.v
abc 02> strash
abc 03> dc2
abc 04> collapse
abc 05> show -g

• The frame command will allow ABC to unroll the sequential circuit into different frames. How many frame is needed to unroll this circuit and why? Unroll the circuit, and then map it again. Put the circuit graph in your report. How many levels does the unrolled circuit has?

There are 3 latches in the circuit. Which can hold at most 8 states. So it is sufficient to unroll for 8 frames. However, this is a sequence detector for 101, so it is necessary to only unroll for 3 frames. Later, we can see that the circuit can become satisfiable in only 3 clocks which also shows that 3 frames is sufficient. The circuit has 5 levels. Figure 7 shows the circuit graph of the unrolled circuit. Unrolling is one of the most powerful tools for analyzing sequential circuit and you should study and understand it very well. It allows you to use combinational circuit analysis tools (such as SAT) to analyze sequential circuits.

abc 05> strash
abc 06> dc2
abc 07> show
abc 07> frames -F 3
abc 08> map
abc 09> print_stats
top_3_frames: i/o =9/3  lat = 3  nd = 32  edge =84  area =32.00  delay = 5.00 lev = 5

• Optimize the circuit using balance command and then map the circuit again. Write the balanced netlist. Show the balanced circuit graph. What is the area and the delay of the optimized circuit? solutions: Balancing the circuit (either using balance command or dc -b) will results in a smaller circuit. Figure 8 shows the balanced circuit graph.

abc 01> read -m SeqDet.v
abc 02> strash
abc 03> dc2
abc 04> balance
abc 05> map
abc 06> print_stats
top: i/o = 3/1  lat =3  nd =8  edge =18  area = 8.00  delay = 3.00 lev = 3
• Optimize the circuit for the minimum area (use `detime` command). Write the retimed netlist and show the retimed circuit graph. You should get the similar results to balance command.

• **Reachability analysis using BDD:** Determine if the output is reachable. (After strash, use `&get; &reachy`). How many clocks does it take to successfully assert the output? ABC uses BDDs to perform the reachability analysis, however the BDDs are not visible. What is the effects of an unreachable output in synthesis?

It takes 3 clocks to assert the output.

```
abc 06> &get
abc 06> &reachy
Output 0 of miter "top" was asserted in frame 2. Time = 0.01 sec
```

If a state or output is unreachable, it should be simply removed from the FSM during the optimization parts.

• **Satisfiability analysis using SAT:** Determine if, ever, the output of the circuit can become 1.
  
  – Unroll and then convert the circuit into a combinational circuit using `frames` and `comb`.
  
  – Use `sat` to check if the output is satisfiable.

```
abc 06> frames -F 3
abc 07> comb
abc 08> sat
SATISFIABLE Time = 0.00 sec
abc 08> sat -v
```

```
<table>
<thead>
<tr>
<th>Conflicts</th>
<th>ORIGINAL</th>
<th>LEARNT</th>
<th>Progress</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clauses</td>
<td>Literals</td>
<td>Limit Clauses</td>
<td>Literals</td>
<td>Lit/Cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
<td>--------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>0</td>
<td>72</td>
<td>187</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

starts : 1
conflicts : 0
decisions : 14
propagations : 29
SATISFIABLE Time = 0.00 sec
```

Which shows that the output is satisfiable, which is expected for a sequence detector circuit.

2 **Sequential equivalence checking [15 points]**

• Your “friend” has used ABC for optimizing the sequence detector. This is the result netlist. Is her circuit equivalent to the original FSM? Use `dsec` command to prove it. (Similar to `cec`, load the first network and then use `dsec target` to compare it with the target.) Download the netlist from [here](#).
module top2 ( clock, rst, inp, outp );
  input  clock, rst, inp;
  output outp;
  reg lo0, lo1, lo2;
  wire n14, n16, n17, n19, n20, li0, li1, li2;
  INVX1 g0(.A(rst), .Y(n14));
  AND2X1 g1(.A(inp), .B(n14), .Y(li0));
  NAND2X1 g2(.A(lo1), .B(rst), .Y(n16));
  NAND4X1 g3(.A(lo1), .B(lo0), .C(inp), .D(n16), .Y(n17));
  NAND2X1 g4(.A(n17), .B(n16), .Y(li1));
  OR2X1  g5(.A(inp), .B(rst), .Y(n19));
  NAND3X1 g6(.A(lo2), .B(inp), .C(n14), .Y(n20));
  MX2X1  g7(.A(n20), .B(n19), .S0(lo0), .Y(li2));
  BUFX1  g8(.A(lo1), .Y(outp));
always @ (posedge clock) begin
  lo0 <= li0;
  lo1 <= li1;
  lo2 <= li2;
end
  initial begin
  lo0 <= 1'b0;
  lo1 <= 1'b0;
  lo2 <= 1'b0;
end
endmodule

- Prove that the balanced circuit and retimed circuit that you have generated in problem 1 are equivalent to the original netlist.

You can use dsec to run equivalence. The friends’s netlist is not equal to our circuit (I intentionally modified output of a gate or two in friends netlist). The balanced circuit and retimed circuit should be equivalent to the original circuit (we don’t want the synthesizer to alter the functionality of the circuit)

abc 01> read_library cadence.genlib
Entered genlib library with 28 gates from file "cadence.genlib".
abc 01> read -m seq_balanced.v
abc 02> dsec SeqDet.v
Networks are equivalent. Time = 0.01 sec
abc 02> dsec friend.v
Networks are NOT EQUIVALENT after simulation. Time = 0.01 sec
3 Faulty fault-tolerant design [35 points]

In this problem, we are going to evaluate faulty: a fault-tolerant design. This problem consists of two parts: 1) understanding how to do fault injection using ABC, and 2) Evaluate an example fault tolerant design and identify its vulnerabilities.

3.1 Using ABC for fault injection [10 points]

Soft errors are basically bit-flips in the design. Unlike the hard-errors that occur during manufacturing (such as stuck-at faults), the soft errors are temporary. An easy way to model a single-event-upset at a node \( x \) is to xor the node \( x \) with 1 (i.e. flip the node \( x \)). Then two things can happen: Either this error will propagate to the output, or the error doesn’t change the output. In the second case, we say the error is masked, or the circuit has tolerated the fault.

For example, consider a simple subtracter circuit. We want to analyze whether a bit flip on input \( A_0 \) will propagate to the output. We can use the design shown in Figure 9 to evaluate the soft error rate. We instantiate two copies of the subtracter. One fault free copy and one faulty copy where the fault site \( A_0 \) is XORed with a 1. The output of the faulty and fault-free modules are XORed together (this design is called a miter circuit). If the output of the miter is 1 it means that the fault has propagated to the primary output. Explain why?

- implement the SUB2 and the miter circuit in verilog. Then determine whether the circuit is susceptible to fault injected at each of the primary inputs (\( A_0, A_1, B_0, B_1 \)). Put the entire design, the ABC command that you used and the output of the sat command in your report.

This is the fault-free 2-bit subtracter circuit:

```verilog
module sub ( A, B, C );
    input [1:0] A;
    input [1:0] B;
    output [1:0] C;
    wire n5, n6, n7;

    XOR2X1 U6 ( .A(n5), .B(n6), .Y(C[1]) );
    XOR2X1 U7 ( .A(B[1]), .B(A[1]), .Y(n6) );
    NOR2X1 U8 ( .A(A[0]), .B(n7), .Y(n5) );
    INVX1 U9 ( .A(B[0]), .Y(n7) );
    XOR2X1 U10 ( .A(B[0]), .B(A[0]), .Y(C[0]) );
endmodule
```

The same idea works for sequential circuits too. We just have to unroll the circuit and convert it into a combinational circuit first.

solution: This is the implementation of the miter circuit. We are injecting fault at the input \( A[1] \) or the sub2. This circuit is satisfiable which is shown using ABC. If you remove the injected fault, the result should be unsatisfiable. By repeating this procedure for the other three inputs (\( A_0, B_0, B_1 \)) you should see all of them results in satisfiable results, which means the circuit cannot tolerate
any fault at any of those inputs. This is expected, because a simple subtracter circuit doesn’t have any form of redundancy in it.

module sub ( A, B, C );
    input [1:0] A;
    input [1:0] B;
    output [1:0] C;
    wire n5, n6, n7;
    XOR2X1 U6 ( .A(n5), .B(n6), .Y(C[1]) );
    XOR2X1 U7 ( .A(B[1]), .B(A[1]), .Y(n6) );
    NOR2X1 U8 ( .A(A[0]), .B(n7), .Y(n5) );
    INVX1 U9 ( .A(B[0]), .Y(n7) );
    XOR2X1 U10 ( .A(B[0]), .B(A[0]), .Y(C[0]) );
endmodule

module miter (A, B, C);
    input [1:0] A;
    input [1:0] B;
    output o;
    wire [1:0] C1;
    wire [1:0] C2;
    wire [1:0] faulty;
    BUFX1 (.A(A[0]), .Y(faulty[0]));

    sub sub1 (.A(A), .B(B), .C(C1));
    sub sub2 (.A(faulty), .B(B), .C(C2));

    XOR2X1 (.A(C1[0]), .B(C2[0]), .Y(C[0]));
    XOR2X1 (.A(C1[1]), .B(C2[1]), .Y(C[1]));
    OR2X1 (.A(C[0]), .B(C[1]), .Y(o));
endmodule

-------ABC
abc 01> read_library cadence.genlib
Entered genlib library with 28 gates from file "cadence.genlib".
abc 01> read -m sub_miter.v
Hierarchy statistics can be found in log file "sub_miter.log".
Warning: The network contains hierarchy.
Hierarchy reader flattened 2 instances of logic boxes and left 0 black boxes.
abc 02> strash
abc 03> sat
SATISFIABLE  Time = 0.00 sec
3.2 Evaluate a faulty fault-tolerant design [20 points]

Your “friend” has come up with a fault tolerant design for the sequencer circuit that we studied earlier. Figure 10 shows her idea: There are obvious flaws in this design. We are going to use ABC for computing the soft error rate of this design.

- Implement the fault-tolerant design in verilog
- Convert the circuit into a combinational circuit by unrolling (use frames) and removing the latches (use comb).
- Implement the miter in verilog. Updated: Unroll the circuit into three frames and then pick 10 gates. Inject faults at the output of those ten gates and report how many of them failed the circuit. You need a new verilog netlist file for each fault injection.
- Use ABC to determine whether the output of the miter is satisfiable (use the sat command, you can also use BDDs). What is the error rate of your experiment? (How many of the injected faults caused failure in the output?)

To design the faulty, we use the unrolled seq detector that we generated in the last question. There are multiple inputs that we don’t care about (such as resets and clocks). The only three inputs that are important are the inp0, inp1 and inp2 which indicate the input signal at clock 1, 2 and 3. The other important inputs are the initial state of the DFFs which we set all to zero. The only output that we care about is the output of the system at the 3rd clock so we ignore the other outputs. In the miter design, we instantiate the sequence detector twice with the same inputs, but we wire the outputs to an xor and mux circuit. This is the faulty design in verilog:

```verilog
module seq_unrolled ( clock_00, rst_00, inp_00, clock_01, rst_01, inp_01, clock_02, rst_02, inp_02, state[0], state[1], outp, outp_00, outp_01, outp_02, n20, n23, n26 );

input clock_00, rst_00, inp_00, clock_01, rst_01, inp_01, clock_02, rst_02, inp_02, state[0], state[1], outp;
output outp_00, outp_01, outp_02, n20, n23, n26;
wire n19, n20_1, n21, n22, n24, n25, n26_1, n27, n28, n29, n30, n31, n33, n35, n36, n37, n38, n40, n41;
INVX1 g00(.A(state[1]), .Y(n19));
INVX1 g01(.A(rst_00), .Y(n20_1));
NAND3X1 g02(.A(state[0]), .B(inp_00), .C(n20_1), .Y(n21));
NAND2X1 g03(.A(outp), .B(rst_00), .Y(n22));
OAI21X1 g04(.A0(n21), .A1(n19), .B0(n22), .Y(outp_01));
INVX1 g05(.A(rst_01), .Y(n24));
INVX1 g06(.A(inp_00), .Y(n25));
INVX1 g07(.A(state[0]), .Y(n26_1));
NOR3X1 g08(.A(n26_1), .B(n25), .C(rst_00), .Y(n27));
AOI22X1 g09(.A0(n27), .A1(state[1]), .B0(outp), .B1(rst_00), .Y(n28));
INVX1 g10(.A(inp_01), .Y(n29));
NOR4X1 g11(.A(n29), .B(rst_01), .C(n25), .D(rst_00), .Y(n30));
OAI21X1 g12(.A0(n21), .A1(n19), .B0(n30), .Y(n31));
```
module faulty (inp_00, inp_01, inp_02, po);
    input inp_00, inp_01, inp_02;
    output po;
    wire zero, one, seq1_po, seq2_po, comp, n20_1, n20_2, n23_1, n23_2, n26_1, n26_2,
        ZERO zg(.Y(zero));
    ONE og(.Y(one));
    seq_unrolled seq1(zero, zero, inp_00, zero, zero, inp_01, zero, zero, inp_02, zero, zero, zero, outp_00_1, outp_01_1, outp_02_1, n20_1, n23_1, n26_1);
    seq_unrolled seq2(zero, zero, inp_00, zero, zero, inp_01, zero, zero, inp_02, zero, zero, outp_2, outp_00_2, outp_01_2, outp_02_2, n20_2, n23_2, n26_2);
    XOR2X1 (.A(outp_02_1), .B(outp_02_2), .Y(comp));
    MX2X1 (.A(outp_02_1), .B(outp_02_2), .S0(comp), .Y(po));
endmodule

You can check if this netlist works by checking if the output is satisfiable (i.e. it will detect some sequence). You can furthermore reverse-engineer the functionality of the circuit by looking at the strashed network (this is going to be a gate relating input00, input01, ..., i.e. a gate that detects a sequence).

Adding a miter to this circuit is easy. Keep in mind that there are 4 sequence detector in the miter (each faulty contains two seq. detector circuit).

module miter (inp_00, inp_01, inp_02, mo);
    input inp_00, inp_01, inp_02;
    output mo;
    wire po_1, po_2;
    faulty f1 (inp_00, inp_01, inp_02, po_1);
    faulty f2 (inp_00, inp_01, inp_02, po_2);
    XOR2X1 (.A(po_1), .B(po_2), .Y(mo));
endmodule

Now, we haven’t injected any faults yet, so when we run satisfiability, the result is unsatisfiable.
Now if we inject a fault at the output of a gate such as \( g_{13} \), then the result of the sat algorithm will be satisfiable. This means a fault injected at that gate will be propagated to the output. However, this is not true for all gates. Some gates will be able to actually tolerate fault. Furthermore, if you inject fault at the original sequencer (and not the backup sequencer in faulty) they will also be tolerated. For example, if I inject a fault at the fault at the output of the gate \( g_{04} \) in the backup sequence detector, the result is unsatisfiable. Overall, less than 50% of the faults will become an error and more than 50% should be tolerated.

```
abc 04> strash
abc 05> sat
SATISFIABLE  Time =  0.00  sec
```

This is an example of a full netlist used for injecting fault:

```vhdl
module seq_unrolled_golden ( clock_00, rst_00, inp_00, clock_01, rst_01, 
inp_01, clock_02, rst_02, inp_02, \state[0] , \state[1] , outp, outp_00, 
outp_01, outp_02, n20, n23, n26 );
  input  clock_00, rst_00, inp_00, clock_01, rst_01, inp_01, clock_02, 
rst_02, inp_02, \state[0] , \state[1] , outp;
  output outp_00, outp_01, outp_02, n20, n23, n26;
wire n19, n20_1, n21, n22, n24, n25, n26_1, n27, n28, n29, n30, n31, 
n33, n35, n36, n37, n38, n40, n41;
INVX1  g00(.A(\state[1] ), .Y(n19));
INVX1  g01(.A(rst_00), .Y(n20_1));
NAND3X1  g02(.A(\state[0] ), .B(inp_00), .C(n20_1), .Y(n21));
NAND2X1  g03(.A(outp), .B(rst_00), .Y(n22));
OAI21X1  g04(.A0(n21), .A1(n19), .B0(n22), .Y(outp_01));
INVX1  g05(.A(rst_01), .Y(n24));
INVX1  g06(.A(inp_00), .Y(n25));
INVX1  g07(.A(\state[0] ), .Y(n26_1));
NOR3X1  g08(.A(n26_1), .B(n25), .C(rst_00), .Y(n27));
AOI22X1  g09(.A0(n27), .A1(\state[1] ), .B0(outp), .B1(rst_00), .Y(n28));
INVX1  g10(.A(inp_01), .Y(n29));
NOR4X1  g11(.A(n29), .B(rst_01), .C(n25), .D(rst_00), .Y(n30));
OAI21X1  g12(.A0(n21), .A1(n19), .B0(n30), .Y(n31));
OAI21X1  g13(.A0(n28), .A1(n24), .B0(n31), .Y(outp_02));
INVX1  g14(.A(rst_02), .Y(n33));
AND2X1  g15(.A(inp_02), .B(n33), .Y(n20));
AND2X1  g16(.A(inp_01), .B(n24), .Y(n35));
NAND3X1  g17(.A(n20), .B(n31), .C(n35), .Y(n36));
AOI21X1  g18(.A0(inp_01), .A1(n24), .B0(inp_02), .Y(n37));
NAND2X1  g19(.A(n37), .B(n33), .Y(n38));
AND2X1  g20(.A(n38), .B(n36), .Y(n23));
OR2X1  g21(.A(n21), .B(n19), .Y(n40));
AOI22X1  g22(.A0(n30), .A1(n40), .B0(outp_01), .B1(rst_01), .Y(n41));
OAI21X1  g23(.A0(n41), .A1(n33), .B0(n36), .Y(n26));
BUFX1  g24(.A(outp), .Y(outp_00));
```
module seq_unrolled_fi (clock_00, rst_00, inp_00, clock_01, rst_01, inp_01, clock_02, rst_02, inp_02, \state[0], \state[1], outp, outp_00, outp_01, outp_02, n20, n23, n26);
    input clock_00, rst_00, inp_00, clock_01, rst_01, inp_01, clock_02, rst_02, inp_02, \state[0], \state[1], outp;
    output outp_00, outp_01, outp_02, n20, n23, n26;
    wire n19, n20_1, n21, n22, n24, n25, n26_1, n27, n28, n29, n30, n31, n33, n35, n36, n37, n38, n40, n41;
    wire temp;
    INVX1 g00(.A(\state[1]), .Y(n19));
    INVX1 g01(.A(rst_00), .Y(n20_1));
    NAND3X1 g02(.A(\state[0]), .B(inp_00), .C(n20_1), .Y(n21));
    NAND2X1 g03(.A(outp), .B(rst_00), .Y(n22));
    OAI21X1 g04(.A0(n21), .A1(n19), .B0(n22), .Y(outp_01));
    INVX1 g05(.A(rst_01), .Y(n24));
    INVX1 g06(.A(inp_00), .Y(n25));
    INVX1 g07(.A(\state[0]), .Y(n26_1));
    NOR3X1 g08(.A(n26_1), .B(n25), .C(rst_00), .Y(n27));
    AOI22X1 g09(.A0(n27), .A1(\state[1]), .B0(outp), .B1(rst_00), .Y(n28));
    INVX1 g10(.A(inp_01), .Y(n29));
    NOR4X1 g11(.A(n29), .B(rst_01), .C(n25), .D(rst_00), .Y(n30));
    OAI21X1 g12(.A0(n21), .A1(n19), .B0(n30), .Y(n31));
    //---------------------------------------
    OAI21X1 g13(.A0(n28), .A1(n24), .B0(n31), .Y(temp));
    INVX1 fi(.A(temp), .Y(outp_02));
    //---------------------------------------
    INVX1 g14(.A(rst_02), .Y(n33));
    AND2X1 g15(.A(inp_02), .B(n33), .Y(n20));
    AND2X1 g16(.A(inp_01), .B(n24), .Y(n35));
    NAND3X1 g17(.A(n20), .B(n31), .C(n35), .Y(n36));
    AOI21X1 g18(.A0(inp_01), .A1(n24), .B0(inp_02), .Y(n37));
    NAND2X1 g19(.A(n37), .B(n33), .Y(n38));
    AND2X1 g20(.A(n38), .B(n36), .Y(n23));
    OR2X1 g21(.A(n21), .B(n19), .Y(n40));
    AOI21X1 g22(.A0(n30), .A1(n40), .B0(outp_01), .B1(rst_01), .Y(n41));
    OAI21X1 g23(.A0(n41), .A1(n33), .B0(n36), .Y(n26));
    BUFX1 g24(.A(outp), .Y(outp_00));
endmodule

module faulty_golden (inp_00, inp_01, inp_02, po);
    input inp_00, inp_01, inp_02;
    output po;
    wire zero, one, seq1_po, seq2_po, comp, n20_1, n20_2, n23_1, n23_2, n26_1, n26_2, outp_1, outp_00_1, outp_01_1, outp_02_1, outp_2, outp_00_2,
module faulty_fi (inp_00, inp_01, inp_02, po);
    input inp_00, inp_01, inp_02;
    output po;
    wire zero, one, seq1_po, seq2_po, comp, n20_1, n20_2, n23_1, n23_2,
    n26_1, n26_2, outp_00, outp_01, outp_02, outp_00_2, outp_01_2, outp_02_2;
    ZERO zg(.Y(zero));
    ONE og(.Y(one));
    seq_unrolled_golden seq1(zero, zero, inp_00, zero, zero, inp_01, zero,
    zero, inp_02, zero, zero, zero, outp_00, outp_01, outp_02, n20_1, n23_1, n26_1);
    seq_unrolled_golden seq2(zero, zero, inp_00, zero, zero, inp_01, zero,
    zero, inp_02, zero, zero, zero, outp_2, outp_00_2, outp_01_2, outp_02_2, n20_2,
    n23_2, n26_2);
    XOR2X1 (.A(outp_02_1), .B(outp_02_2), .Y(comp));
    MX2X1 (.A(outp_02_1), .B(outp_02_2), .S0(comp), .Y(po));
endmodule

module miter (inp_00, inp_01, inp_02, mo);
    input inp_00, inp_01, inp_02;
    output mo;
    wire po_1, po_2;
    faulty_fi f1 (inp_00, inp_01, inp_02, po_1);
    faulty_golden f2 (inp_00, inp_01, inp_02, po_2);
    XOR2X1 (.A(po_1), .B(po_2), .Y(mo));
endmodule

injecting fault at the gate g14:
    wire temp;
    OAI21X1 g04(.A0(n21), .A1(n19), .B0(n22), .Y(temp));
    INVX1 fi(.A(temp), .Y(outp_01));
Network structure visualized by ABC
Benchmark "top". Time was Thu Oct 30 11:33:31 2014.

The network contains 11 logic nodes and 3 latches.

Figure 1: Sequential AIG of the sequence detector circuit
Network structure visualized by ABC
Benchmark "top". Time was Thu Oct 30 11:42:20 2014.

The network contains 11 logic nodes and 3 latches.

Figure 2: The circuit graph of the sequencer circuit
Figure 3: BDD graph for node outp
Figure 4: BDD graph for state0
Figure 5: BDD graph for state1
Network structure visualized by ABC
Benchmark "top". Time was Thu Oct 30 12:04:24 2014.

The network contains 3 logic nodes and 3 latches.

Figure 6: Circuit graph for the collapsed circuit
Figure 7: Circuit graph for the unrolled circuit
Network structure visualized by ABC
Benchmark "top". Time was Thu Oct 30 18:41:10 2014.

The network contains 8 logic nodes and 3 latches.

Figure 8: Circuit graph for the balanced circuit
Figure 9: Fault injection miter circuit

Figure 10: Faulty: The fault tolerant design