EECS 579 Homework No. 5 Solutions

Problem 1 (30 points) *Scan design of a modulo-5 counter. Text p.486, Prob. 14.5 and p.487, Prob. 14.6.*

(**a**) A modulo-5 counter is a simple exercise in logic design using K-maps; some of you used the CAD tool *Espresso* (available on CAEN, but overkill for this problem) instead. As shown in the state transition graph of Fig. 1, five states $\{s_0,s_1,s_2,s_3,s_4\}$ are needed. These can be encoded in the natural way as {000, 001, 010, 011, 100} using three flip-flops. The three unused state patterns {101, 110, 111} imply don't care values that can be simplify the combinational logic.The count-up input C = 1 (with CLR = 0) increments the state at every clock tick. The second input CLR = 1 initializes the circuit to the 000 state. The primary output Z remains 0 with the exception of the state 100, which produces output Z = 1.

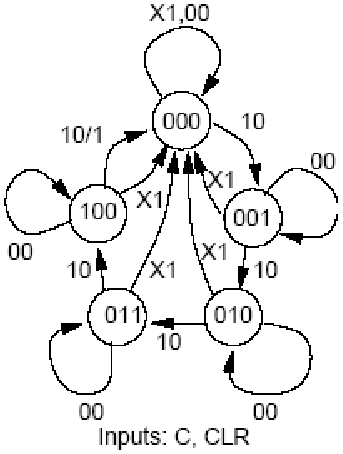

Fig. 1

Non-scan test generation was done by the class in three ways: manually via the low-level PODEM or DALG algorithms, manually using high-level functional tests, or automatically using the *TetraMax* or *FlexScan* ATPG programs available on CAEN. Because of the unused (redundant) states, some stuck-at-faults may be undetectable. The number of tests needed to cover all detectable faults depends on the specific logic design and ATPG method used, and can be as small as six tests, so this was not a huge problem.

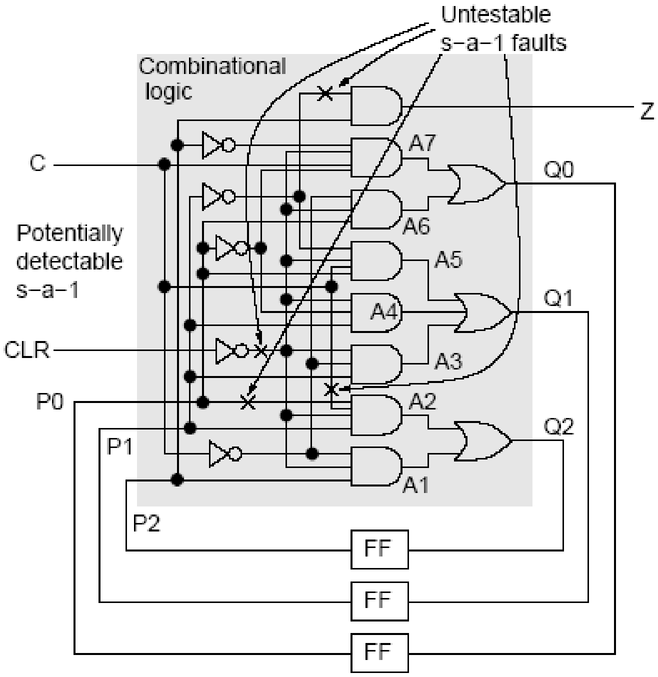

Fig. 2

Figure 2 has the “official” solution generated by Bushnell and Agrawal, which is actually more complicated than most of the solutions found by this class. The combinational part (the grey box) has 62 faults including five (they may mean four) redundant faults, which were found by the authors’ ATPG program (*Gentest*?). That program produced 62 test vectors to obtain a coverage of (57/62) x 100 = 92.98%. Untestable faults were all of the s-a-1 type and are marked in Fig. 2.

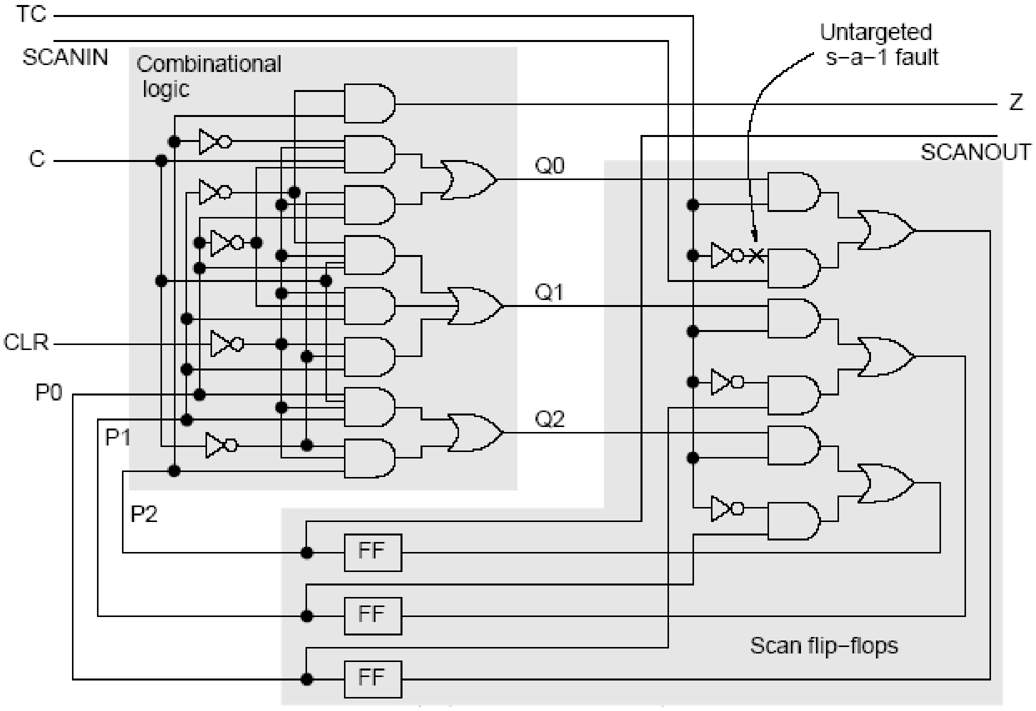

Fig. 3

(**b**) The basic counter is easily made scannable by adding three two-way muxes, one per flip-flop, as shown in Fig. 3 for the design of Fig. 2. (Some students found a program *DFTadvisor* that adds scan automatically!) Scan tests can be derived in various ways, either manually or using CAD tools. Note that any previously undetectable faults typically become detectable when scan testing is used. (Why?)

An interesting manual approach used by a couple of students is to generate functional tests, which can be derived in much the same way as memory tests. The target function “count up modulo 5” is simple, and can be deduced directly from the state transition graph (Fig. 1). There are five states $\{s_0,s_1,s_2,s_3,s_4\}$, and for each we need to test that the following two functions are performed correctly: (1) Hold the current state s_i; (2) Increment s_i to $s_{i+1 \text{ (modulo 5)}}$. The resulting ten or so scan tests then take the following form:

```
Initialize the circuit
for $i$ = 0 to 4 {
    Apply input vector that holds state $s_i$;
    Scan out and check the result:
    Apply input vector that increments $s_i$ to $s_{i+1 \text{ (mod 5)}}$;
    Scan out and check the result: }
```

This testing procedure takes about 25 scan clock cycles. Of course, it does not guarantee 100% coverage of SSL faults; this has to be checked by (manual or automated) simulation. With additional functional tests, one might want to check the reset function $s_i \rightarrow s_0$ for each state s_i, thereby testing every transition appearing in the state transition graph.

Again the number of scan cycles and tests will vary with the circuit and ATPG program. The following results were obtained by the authors of the text with their *Gentest* program targeting SSL faults in Fig. 3. The combinational circuit, whose inputs are C, CLR, P0, P1 and P2, and outputs are Z, Q0,Q1 and Q2, has a collapsed fault set of 57 faults. All these faults were detected by 16 vectors. Their complete scan sequence consists of 74 vectors (see Equation 14.1 in the text), which includes 7 vectors for testing the scan register. The scan circuit contains a collapsed set of 79 faults. Fault simulation of the 74-vector sequence showed that 78 faults were detected. The undetected s-a-1 fault is marked on Fig. 3, and is at the output of the test control (TC) inverter in the first multiplexer. This fault was not detected is that it was never targeted, since the scan register test holds TC at 0 in scan mode, thus preventing the fault from being activated. The fault is, however, activated every time the circuit is set in the normal mode during the application of the scan sequence. Since in the normal mode the state of SCANIN is considered irrelevant, SCANIN was arbitrarily set to 0. That prevented the propagation of the fault effect. A suitable strategy for detecting this fault is to set Q0 outputs of the combinational logic to 0 by applying CLR = 1. At the same time, the circuit is set in the normal mode by applying TC = 1. The fault effect is now propagated to the flip-flop and can be scanned out. Note that similar faults in the other two multiplexers were detected when TC = 1, which would place the fault effect in the flip-flop. TC = 1 was always followed by scan-out that detected the fault. In general, it is recommended that SCANIN be set to 1 whenever the circuit goes to the normal mode (TC = 1), provided the AND-OR type of multiplexer is used

Problem 2 (20 points) *Partial scan and S-graphs*

(**a**) *Text, page 487, Problem 14.8*. The S-graph of the circuit in Figure 14.16 is given below (Fig. 4). By scanning F1, all cycles can be eliminated.

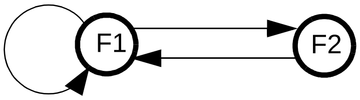

Fig. 4

(**b**) *Text, page 487, Problem 14.11*. From the definition of a strongly connected component (SCC), every vertex in this graph lies on one or more cycles. A good heuristic is to select a vertex that is likely to be on a large number of cycles. Deletion of this vertex then eliminates those cycles. Note that computing the exact or approximate number of cycles on which any vertex lies is *not* simple.

A simple selection rule suggested by many of you is to select a vertex *v* with the largest degree, since large degree implies many paths, and possibly many cycles, passing through *v*. Degrees are very simple to compute. Better (and slightly less simple) selection rules would consider the direction of the edges through *v*, for example, a *v* with large indegree and large outdegree. Several of you suggested selecting the *v* with the maximum sum of indegree and outdegree, which is a good choice. It turns out, however, that the vertex with the highest *product* of indegree and outdegree is even better. [For further discussion of this, see the paper by S. Bhawmik et al.: “PASCANT: A Partial Scan and Test Generation System,” *Proc. IEEE Custom Integrated Circuits Conf.*, May 1991, pp.17.3.1 - 17.3.4.] Once a vertex *v* and all its edges are deleted, the remaining s-graph may have one or more smaller SCCs. The same procedure of finding and deleting the vertex with the largest indegree-by-outdegree product is recursively applied until the remaining s-graph has no SCCs.

As is the case with most heuristics, determining how good a particular heuristic is would require carrying out comparative experiments with lots of representative data.

Problem 3 (10 points) *ATE*

(**a**) The three waveform formats in question, DNRZ, RZ and R1, are clearly defined in the handout for the HP 82000 tester used by Jay Sivagnaname for his lab demo class. Applying them to 11001 gives the answer shown in Fig. 5.

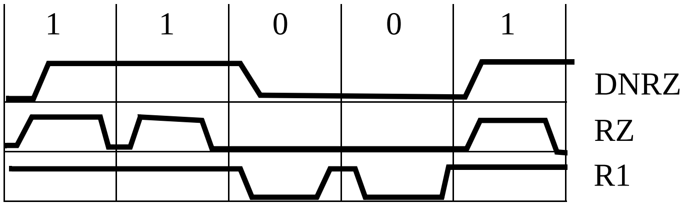

Fig. 5

(**b**) This problem can be tackled by thoughtfully analyzing the somewhat obscure information in the tester handout. Basically, the key issue is that “NR” formats like DNRZ and NRZ (which is not mentioned explicitly here) define level signals, whereas the “R” formats define pulse signals. Each format type therefore has a fundamental advantage when describing level or pulse data. RZ (R1) require a positive (negative) edge or transition at the start of all clock cycles with 1 (0) data values, as is clearly seen in Fig. 5, which may be needed to operate or test synchronous, edge-triggered devices.

Some additional facts: Consider ATE applying a test signal *x* to some pin of a DUT. The NRZ format specifies a stable 1 or 0 signal level for an entire ATE clock cycle, and only allows *x* to change at the start of a cycle. DNRZ or “delayed NRZ” additionally allows a predefined delay to be inserted between the start of the test cycle and a change in *x*. If the ATE handles an “R” format, then several data transitions can occur during a single test cycle. RZ might be used to specify a clock or similar enabling control signal generated by the ATE and applied to the DUT; then each occurrence of 1 in an RZ-formatted test will cause a triggering positive edge to be applied to the DUT. This requires the 1 to return to 0 at the end of each test cycle so a new positive edge can be produced by a 1 in the next test cycle.

RZ and R1 are basically complementary formats that reverse the roles of positive and negative edges. They have nothing to do with testing for stuck-at-0 versus stuck-at-1 faults, as some of you suggested. Power consumption is also not an issue, although it is true that because they cause more signal transitions, i.e., more switching activity, the RZ/R1 formats tend to consume more power than DNRZ.

Problem 4 (30 points) *Pseudorandom test pattern generator*

(**a**) *i*) There is a table of primitive polynomials on text p. 620, where we find $P(x) = x^8 + x^6 + x^5 + x + 1$.
ii) Below is the corresponding Type 1 LFSR version which most people used; the least significant bit is on the right. Not shown are the CLOCK and (asynchronous) RESET inputs to the eight flip-flops.

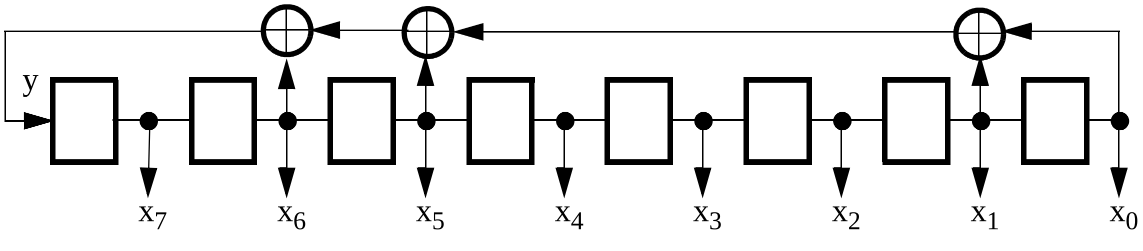

Fig. 6

iii) From clocking the LFSR (manual simulation), the required state sequence is easily found to be:

00000001 → 10000000 → 01000000 → 10100000 → 110100000

Several of you computed the states via the circuit matrix, which is ok but more work.

(**b**) The problem is to insert the state $S_0 = 00000000$ between some two consecutive states S_i and S_{i+1} without altering any other state transitions. It’s desirable to pick the *i* so that only a little extra logic is needed. A systematic design approach, which most people used, is to add a circuit that recognizes S_i and S_0 and makes the corresponding next states S_0 and S_{i+1}, respectively. In other words, replace some state transition $S_i \rightarrow S_{i+1}$ by $S_i \rightarrow S_0 \rightarrow S_{i+1}$.

To select S_i, we can look at the possible state transitions and take advantage of the fact that C is a shift register. For example, we see from part (a) that if S_i = 00000001, then S_{i+1} = 10000000, and S_0 can be interpolated between them just by controlling the signal *y* shifted into the leftmost flip-flip of C. The desired result $S_i \rightarrow S_0 \rightarrow S_{i+1}$ then becomes

00000001 → 00000000 → 10000000 → ...

There are several ways to achieve this, and the necessary circuits can always be implemented in NANDs since NAND is a universal logic element. Here's one design. We need to perform the following functions:

if $x_7x_6x_5x_4x_3x_2x_1x_0$ = 00000001 **then** $y = 0$ **else** *y* is unchanged

if $x_7x_6x_5x_4x_3x_2x_1x_0$ = 00000000 **then** $y = 1$ **else** *y* is unchanged

The two **if** functions reduce to a simpler one

if $x_7x_6x_5x_4x_3x_2x_1$ = 0000000 **then** $y = \overline{x}_0$ **else** *y* is unchanged

and can be implemented directly as shown in Fig. 7.

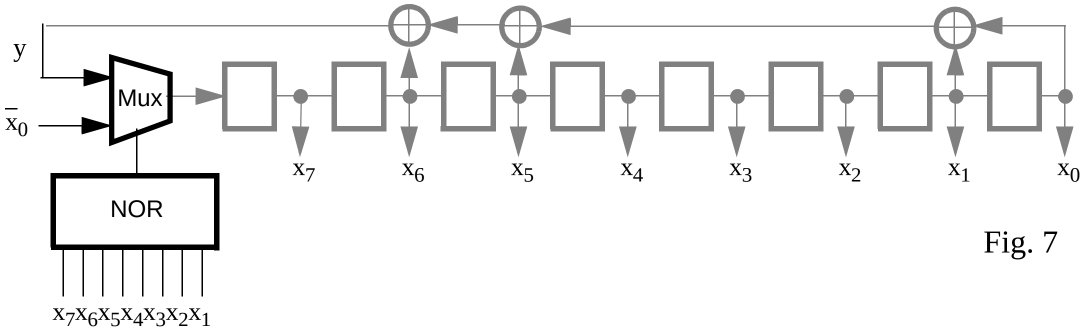

Fig. 7

You can also replace the last **if** by

if $x_7x_6x_5x_4x_3x_2x_1$ = 0000000 **then** *y* is inverted **else** *y* is unchanged

in which case Mux is replaced by an XOR gate. This was the design that most people came up with.

All the class designs had about the same cost in terms of (2-input) NAND gates. There are much cheaper solutions, which require more effort to find, however. Note that all designs involve non-linear FSRs and there's no known minimization theory for NLFSRs.

Note: A couple of you attempted to introduce the all-0 state S_0 into the state-transition cycle by applying signals to the flip-flops' RESET input, or by temporarily turning off the clock. RESET is an asynchronous control line, so asserting it in the middle of synchronous (clocked) operation will seriously mess up timing, and almost certainly cause the circuit to fail. Putting gates in the clock distribution network ("gated clock" design) is also dangerous and should normally be avoided.

a	b	c	f_{good}	f_{bad}	S_{good}	S_{bad}
0	0	0	0	0	000	000
0	0	1	1	0	000	000
0	1	0	0	0	001	000
0	1	1	0	0	010	000
1	0	0	0	0	100	000
1	0	1	1	0	101	000
1	1	0	1	0	110	000
1	1	1	1	0	000	000
					001	000

Fig. 8

Problem 5 (10 points) *Compression testing. Text, page 545, Problem 15.12.*

For multiple fault {*b* s-a-0, *c* s-a-0}, we have the table of Fig. 8 (see the text, p.520, Table 5.4 for some of the same information). After 7 clock periods the LFSR signatures are S_{good} = 001 and S_{bad} = 000. The corresponding transition count (TC) signatures are S_{good} = 3 and S_{bad} = 0. Hence both TC and LFSR compression detect this particular multiple fault.