Design And Test Of Finite State Machines

This contributes 5% of the marks for ELEC2221.

All the processes involved in the design and manufacture of any electronic system are subject to error. Checking that the product has been correctly built (manufacturing test) is an essential part of the production process. The generation of test patterns for manufacturing test can be difficult even for quite simple circuits, but the difficulties can be minimised by applying the principles of design for testability (DFT). This experiment is intended to illustrate some of the aspects of testing and DFT as applied to finite state machines (FSMs).
Schedule

Preparation time : 3 hours
Lab time : 3 hours

Items provided

Tools :
Components :
Equipment : DE1-SoC FPGA Development Board
Software : Altera Quartus, ModelSim

Items to bring

Essentials. A full list is available on the Laboratory website at https://secure.ecs.soton.ac.uk/notes/ellabs/databook/essentials/

Before you come to the lab, it is essential that you read through this document and complete all of the preparation work in section 2. If possible, prepare for the lab with your usual lab partner. Only preparation which is recorded in your laboratory logbook will contribute towards your mark for this exercise. There is no objection to several students working together on preparation, as long as all understand the results of that work. Before starting your preparation, read through all sections of these notes so that you are fully aware of what you will have to do in the lab.

Academic Integrity – If you undertake the preparation jointly with other students, it is important that you acknowledge this fact in your logbook. Similarly, you may want to use sources from the internet or books to help answer some of the questions. Again, record any sources in your logbook.

This exercise does not use the standard mark scheme because there is no additional work section. The mark scheme for this exercise is included on the next page.

Revision History

August 1, 2018  Mark Zwolinski (mz)  Simplified – Combinational logic output removed
August 14, 2017 Mark Zwolinski (mz)  Revised for FPGA
July 6, 2015  Mark Zwolinski (mz)  First version of this lab created in current format

© Electronics and Computer Science, University of Southampton
## Mark Scheme

<table>
<thead>
<tr>
<th>Preparation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Progress &amp; Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Understanding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logbook</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
1 Aims, Learning Outcomes and Outline

This laboratory exercise aims to:

- Introduce you to the idea of transition testing for sequential circuits.
- Demonstrate to you the limitations of sequential testing.
- Introduce you to the concepts of structured design for test, and in particular to scan design.

Having successfully completed the lab, you will be able to:

- Write test sequences for testing state machines.
- Design a simple scan path into a sequential circuit.
- Write test vectors for use in a scan path.

A Finite State Machine (FSM), as its name implies, is a system that can be in one of a finite number of states and remain in that state indefinitely after the inputs have been removed. We will consider exclusively synchronous FSMs, in which the state is represented by the values of a set of state variables, stored in flip-flops driven by a common clock. The state is constant throughout each clock cycle, and changes at the next active edge of the clock.

The operation of the FSM is characterised by the transitions between the state before the clock edge (the present state) and the state after the clock edge (the next state). The transitions are controlled by external inputs in accordance with an algorithm defined by the design requirements: this is the origin of the term “algorithmic state machine” (ASM).

A generalised structure for an FSM is shown in Figure 1. The set of transitions can be described in the form of a state transition diagram or table, or (more clearly) by an ASM chart. The state transitions are determined by the next-state logic, which is a block of combinational logic providing the inputs to the state variable flip-flops. The action of the next-state logic can therefore be described in the form of a set of next state equations, each of which expresses the next value of an individual state variable as a function of the present state (i.e. of all the state variables) and the external inputs.

![General Finite State Machine Diagram](image)

**Figure 1:** General Finite State Machine

The outputs from the ASM could be some or all of the set of state variables. More generally, the outputs are formed as functions of the state variables and the external inputs. These output functions are generated by the output logic, and can be combinational or sequential or both.

This experiment is intended to provide experience in the design and analysis of FSMs, and to introduce some of the concepts of testing and Design For Testability (DFT) as applied to FSMs.
In these notes, an important distinction is made between the internal nodes of a circuit and the primary inputs and outputs (PIs and POs). The difference is simply that the PIs/POs of an integrated circuit are available for control or observation, whereas the internal nodes are not. The difference is important because the basic problem in testing is the access (or lack of it) to the internal nodes of the circuit; the main point of DFT is to provide means for improving this access.

The final aim of this exercise is to show how DFT structures can be built into an integrated circuit (IC) to assist testing. Because it is impractical to insert faults into an IC, we will simulate the behaviour of ICs using an FPGA.

The SystemVerilog and FPGA pin configuration files are available at:

http://www.ecs.soton.ac.uk/notes/ellabs/2/C4

2 Preparation

Read through the course handbook statement on safety and safe working practices, and your copy of the standard operating procedure. Make sure that you understand how to work safely. Read through this document so you are aware of what you will be expected to do in the lab.

2.1 Background Work

(a) Ensure you understand dependency notation so as to be able to interpret the circuit diagrams. (Also, remember de Morgan’s Law: $\bar{A} \cdot \bar{B} = \bar{A} + \bar{B}$)

(b) Revise ASM charts (First Year notes and supplementary reading if necessary) to ensure that you know the meanings of the symbols and the corresponding circuit implementation, paying particular attention to the timing implications.

(c) Read about the following in the notes for ELEC2221:
   - (i) test pattern generation for combinational logic circuits, based on the single-stuck-at fault model;
   - (ii) scan design.

(d) Plan what you are going to do. You are given SystemVerilog code for this exercise, but you need to understand how that code works and how you might modify it.

(e) If you have forgotten how to use ModelSim or Quartus, go through the walkthroughs on the module web page.

2.2 Functional Testing

Functional testing is based on the concept of checking that the circuit performs its intended function. For an FSM, the function of the circuit is encapsulated in the state transitions and the outputs that should be delivered in each state.

The circuit shown in Figure 2 is an FSM with two flip-flops, and without any output logic. Notice that this circuit has a Reset signal (RST).

- Is the reset to each flip-flop synchronous or asynchronous?
- How would the other sort of reset affect the operation in terms of timing?
Part II EL/EE/EM/EEE Laboratory  Design And Test Of Finite State Machines  C4 - 6

**Figure 2: Basic FSM**

(a) Write down the next-state equations for $S$ and $T$.

(b) Draw an ASM chart describing the operation of the circuit.

(c) Develop a sequence of tests for the circuit that verifies each transition between states in the ASM. (In other words, your tests should “walk” around the ASM chart, visiting each state at least once.) Bear in mind that access is limited to the PIs (including CLK) and the POs. Your sequence must be suitable for automatic testing: i.e. every step from switch-on to decision must be totally unambiguous. The sequence must therefore start with a Reset, since it is not possible to predict what state the circuit will assume at switch on. You can, of course, use Reset again within the sequence wherever convenient.

(d) Download the SystemVerilog code for the basic FSM, together with the outline testbench. Note that the reset is active low and the flip-flops are triggered by the falling edge of the clock. This works better with the push buttons on the FPGA board. Modify the testbench to include the sequence from part (c) and simulate the FSM with the testbench in ModelSim. Satisfy yourself that the sequence fully covers each state transition.

What are the relative merits of the state transition table (or diagram) and the ASM chart for representing the action of an FSM?

### 2.3 Sequential Output Logic

The output logic of an FSM is not necessarily combinational. Data can be stored in flip-flops under the control of the state machine. It is a design decision as to whether the general state of a system is encoded as state variables or as data. Transferring flip-flops to the data path logic reduces the complexity of a state machine (remember that removing one flip-flop halves the number of states).

To what extent can flip-flops be exchanged between the state machine and the data path logic? Can all sequential systems be designed as state machines?
There is a small price to be paid for this simplification of the design process: since data stored in the output logic will, in general, be required to be maintained over more than one clock cycle, the flip-flop used must be able not only to be loaded with 1 or 0, as in a D-type flip-flop, but also to retain (HOLD) its present value. This can be achieved by providing the flip-flop with an enable input as symbolised in Figure 3(a): $F$ and $G$ are functions of the state variables and the external inputs. There are various ways in which an enable facility could be implemented.

(i) An enable input can be accommodated in the standard circuitry of a flip-flop without increasing the number of gates. Most register chips have this facility built-in, but discrete flip-flops very rarely do.

(ii) A D-type flip-flop can be used with the enable used to allow or prevent the CLK from reaching the flip-flop, as shown in Figure 3(b). This is the literal interpretation of Figure 3(a), but **GATING THE CLOCK IS BAD PRACTICE!**

Why is gating the clock bad? (Consider the flip-flop setup time requirement, bearing in mind that the CLK signal is common to the whole FSM.)

(iii) A D-type flip-flop can be used with a multiplexer as shown in Figure 3(c). This is a clean synchronous design, giving $Q^+ = G F + \overline{G} Q$. When it comes to the detailed circuit design of the output logic, this multiplexer function could be incorporated into the circuitry that generates $F$ and $G$.

If the FSM developed in section 2.2 (Figure 2) is extended by adding the output circuitry shown in Figure 4, we have introduced an additional PI, $C$, and PO, $N$ instead of the original POs, $T$ and $S^1$.

---

1 Remember: if this were an integrated circuit, we could only observe the POs. Here, we also restrict ourselves to only being able to observe the POs.
(b) Modify the SystemVerilog code (save each version in a separate file and/or folder, with a new name). Note again, that in this code the flip-flop uses the falling edge of the clock. Modify your testbench from section 2.2 and simulate this code. (Again, save this testbench separately.)

It’s possible that this notation will not have been covered in lectures by the time of the lab. Consider the following: In which clock cycle does N change? Does N revert to its previous value at the end of that cycle? Therefore, is the ASM notation that you have previously seen, e.g. N=X.Y, appropriate? If not, we need a new symbol: N←X.Y.

2.4 Scan Design.

Testability problems can be mitigated by using the techniques of design for testability (DFT). The testing of FSMs, in particular, is made much simpler by the use of scan design, the principle of which is illustrated in Figure 5.

The following points should be noted.

(i) All flip-flops (including any in the output logic as well as those in the FSM) are treated alike. Multiplexers (MUX) are inserted in front of each flip-flop. This allows each flip-flop to take one of two signals.

(ii) The mode input (M) is a required additional PI to the circuit. When M=0, the circuit is in its normal operating condition; when M=1, the circuit is in scan mode. Scan mode means that the flip-flops are all connected together to form a shift register or scan path.

(iii) The scan path allows the state of each flip-flop to be set, regardless of the behaviour of the FSM. We simply set M=1 and clock in the values we want. We would then set M=0, to allow the FSM to move to the next state. Similarly, we can read out the state that we’re in by setting M=1 again and clocking the flip-flop values out through SDI.

(a) Redesign your circuit (complete with the output circuit from section 2.3) as a scan design. Your total circuit should have PIs A, C, SDI and M (as well as CLK), and POs N, SDO.

Does it make any difference how the flip-flops are ordered in the scan chain?
(b) Modify your SystemVerilog code from section 2.3 to include scan flip-flops (included in the code examples – again, note the use of the falling edge of the clock). Include the sequential output (N) as well as the additional inputs, C, SDI and M. Also include the scan output SDO. Include the scan connections.

(c) Assuming that the scan path gives you complete control and observability of the combinational part of the circuit, find tests for the stuck at 0 and stuck at 1 faults on the node shown in Figure 6, below.

---

3 Laboratory Work

3.1 Functional Testing

Synthesise the SystemVerilog code for the circuit of Figure 2, using Quartus and program the FPGA on the development board. Use the pin assignment in C4_pins.qsf, following the procedure given in the walkthrough. See Appendix A for the pin assignments. You will get a warning because some of the pins are not used in the first version of the code – you can ignore this. Apply the test sequence using the switches and push buttons on the development board.

Do your test sequences verify that the circuit has been implemented correctly?

3.2 Fault Detection and Diagnosis

In production testing of Printed Circuit Boards (PCBs), it is necessary not only to establish that a particular circuit is not working correctly (fault detection), but also to identify the location of the fault so that it can be repaired (fault location, or fault diagnosis). This is not generally true of Integrated Circuits, which cannot be repaired.

The first step in this process is to decide what faults might appear in the product, so that the test pattern generation can be directed towards these faults. For many years, the standard fault-model has been the single-stuck fault model, in which it is assumed that if there is a fault it will directly affect only a single node of the circuit, and that the fault effect will take the form of the node being stuck at either logic 0 (X/0) or logic 1 (X/1).

The effects of faults on the test sequence with relation to detection and diagnosis will be demonstrated with respect to the particular fault-site indicated in Figure 6.

To apply a fault, include the fault injection module in your SystemVerilog code (see Appendix 2). This requires two additional inputs to the design, which are mapped to switches on the FPGA board – one to emulate a stuck-at-1 fault, and one to emulate a stuck-at-0. (You cannot apply both at the same time – which one takes priority?)

(a) Apply a stuck-at-1 fault using the switch and step through the test sequence.

Is the fault detected? If so, can you deduce where the fault is by knowing at which point in the sequence an error was observed?

(b) Apply a stuck-at-0 fault using the switch and step through the test sequence.

Is the fault detected? If so, can you deduce where the fault is by knowing at which point in the sequence an error was observed?

In each case, how has the ASM chart been modified by the fault?
3.3 Sequential Output Logic

Add the fault injection module to the code for the circuit of section 2.3 and synthesise this to the FPGA. With both faults disabled, verify that your previous test sequence finds no fault.

- Is this test sequence an adequate test of the output logic? If not, what vectors need to be added?

Apply the stuck-at-1 fault as before, and step through your test sequence.

- Is the fault detected? If so, is it possible to diagnose the fault location?

Before going on to the next section, think about the following points:

- How effective is transition verification as a basis for testing an FSM?
- How can you verify transitions from redundant states, and what is the relative cost of doing this?
- How easy is it to detect and diagnose faults in non-scan-path FSMs with and without output logic?

3.4 Scan Design

a) Using the code you developed in your preparation (section 2.4), verify, by simulation in ModelSim, that the system works correctly in ‘normal’ mode and that you can load a bit pattern into the scan chain. You will need to create a new testbench for this. Don’t forget to save all versions of your code.

b) Add the fault injection module to the code for the circuit of section 2.4 and synthesise this to the FPGA. Apply the stuck-at-0 and stuck-at-1 faults of Figure 6 in turn, and step through
the two tests that you wrote as part of the preparation, noting carefully the sequence of operations needed to observe all the test results for the combinational logic.

Do your tests find the faults? If you did not know where the faults were, could you diagnose the location from your tests?

c) Note that this structure, although inserted as an aid for manufacturing test, is also a valuable aid to design verification. Confirm this by verifying that, in the presence of the stuck-at-1 fault (simulating a missing connection in the design), it is easy to check the state transitions and identify the incorrect logic equation.

Are there any benefits in using scan design in the design verification phase (i.e., while the design is still only a computer model in a simulator)?

Appendix 1 Switch and LED Mapping

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Switch/LED</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>SW0</td>
<td>AB12</td>
</tr>
<tr>
<td>c</td>
<td>SW1</td>
<td>AC12</td>
</tr>
<tr>
<td>mode</td>
<td>SW2</td>
<td>AF9</td>
</tr>
<tr>
<td>sa0</td>
<td>SW3</td>
<td>AF10</td>
</tr>
<tr>
<td>sa1</td>
<td>SW4</td>
<td>AD11</td>
</tr>
<tr>
<td>rst</td>
<td>KEY0</td>
<td>AA14</td>
</tr>
<tr>
<td>clk</td>
<td>KEY1</td>
<td>AA15</td>
</tr>
<tr>
<td>s</td>
<td>LEDR0</td>
<td>V16</td>
</tr>
<tr>
<td>t</td>
<td>LEDR1</td>
<td>W16</td>
</tr>
<tr>
<td>n</td>
<td>LEDR2</td>
<td>V17</td>
</tr>
</tbody>
</table>

Appendix 2 How to use fault emulation

The following SystemVerilog code is used to emulate a stuck fault.

```verilog
module inject_fault (output logic o, input logic i, sa0, sal);

always_comb
  if (sa0)
    o = 1'b0;
  else if (sal)
    o = 1'b1;
  else
    o = i;
endmodule
```

If one of the control signals, sa0, sal is asserted, the stuck behaviour is modelled. To apply this to, say, node f in the next state module, you need to add a logic signal and make the following changes:

```verilog
logic f, f_prime;
and g0 (s_plus, e, f_prime);
inject_fault f0 (.o(f_prime), .i(f), .sa0(sa0), .sal(sal));
```
In other words, f_prime now becomes the affected node and drives the nand gate. The two signals, sa0 and sa1 must also be included in the module header for next_state and for the overall C4 module. If you wish to simulate fault insertion in ModelSim, you must also change the testbench, test_C4.

Appendix 3 SystemVerilog Code

This is the basic code for section 2.2, together with that for other modules in sections 2.3 and 2.4.

```verilog
module d_ff (output logic q, qbar, input logic clk, rst, d);

always_ff @(negedge clk, negedge rst)
  if (~rst)
    begin
      q <= 1'b0;
      qbar <= 1'b1;
    end
  else
    begin
      q <= d;
      qbar <= ~d;
    end
endmodule

module scan_dff (output logic q, qbar, input logic clk, rst, d, mode, scan_in);

always_ff @(negedge clk, negedge rst)
  if (~rst)
    begin
      q <= 1'b0;
      qbar <= 1'b1;
    end
  else if (mode)
    begin
      q <= scan_in;
      qbar <= ~scan_in;
    end
  else
    begin
      q <= d;
      qbar <= ~d;
    end
endmodule

module next_state (output logic s_plus, t_plus, input logic s, s_bar, t, t_bar, a);

logic a_bar, e, f, g, h;

nand g0 (s_plus, e, f);

nand g1 (e, s_bar, t);

nand g2 (f, s, a, t_bar);
```
nand g3 (t_plus, g, h);
nand g4 (g, a, s_bar, t_bar);
nand g5 (h, a_bar, s_bar, t);
not g6 (a_bar, a);
endmodule

module output_reg (output logic n, input logic clk, s, t, c);
always_ff @(negedge clk)
    if (s)
        n <= t & c;
endmodule

module C4 (output logic s, t, input logic clk, rst, a);
logic s_plus, t_plus, s_bar, t_bar;
next_state n0 (.*)
d_ff d0 (.q(s), .qbar(s_bar), .clk(clk), .rst(rst), .d(s_plus));
d_ff d1 (.q(t), .qbar(t_bar), .clk(clk), .rst(rst), .d(t_plus));
endmodule

module test_C4;
logic s, t;
logic clk, rst, a;
C4 c4 (.*);
initial
    begin
        clk = 0;
        #20ns;
        forever #10ns clk = ~clk;
    end
initial
    begin
        rst = 0;
        a = 0;
        #10ns rst = 1;
        #10ns rst = 0;
        // include your test sequence here.
    end
endmodule