# SystemVerilog Simulation and Synthesis

## 1 Aims, Objectives and Outline

This exercise aims to provide you with:

- experience of combinational and sequential logic descriptions in SystemVerilog – an advanced Hardware Description Language.
- an introduction to ModelSim – a SystemVerilog simulator,
- an introduction to SynplifyPro – a hardware synthesis

tool, By the end of the exercise you should be able to:

- use advanced commercial software tools, ModelSim and Synplify Pro, to produce SystemVerilog code, simulate it, debug it and synthesise it into hardware.
- design a combinational logic circuit and a simple state machine, develop their descriptions in SystemVerilog and synthesise their circuits with Synplify Pro.

In laboratory exercises T4 and X5 and in the design exercise, D1, you will have an opportunity to program and test programmable logic device (PLD) implementations of state machines (SMs). This exercise provides a necessary preparation for that work, by introducing the basics of digital system description in a hardware description language (SystemVerilog), allowing you to become familiar with hardware description, simulation and testing.

The software tools installed in the Computing Laboratory machines for this purpose are:

- ModelSim from Mentor Graphics – a digital system simulator,
- Synplify Pro from Synplicity – an advanced digital synthesis tool.

You can access these tools remotely by using one of the ECS compute servers (for example, roo.ecs.soton.ac.uk).

## 2 Preparation

This exercise is not assessed, but you should approach it as a standard laboratory session. You will make more effective use of your time if you prepare before sitting at a PC. Use your laboratory logbook for notes – you won't remember important details if you do not write them down. Read all the notes first, before attempting any of the exercises.

### 2.1 Filestore

For all System Verilog exercises, it is essential that you set up a suitable working directory/folder on your server account, which maps as drive H: on ECS Windows PCs. Do not store files on your Desktop or the C: drive, and do not use these locations for working directory structures. USB drives should be used for backup copies of your work, but should not be used as working filestore.

The software you will be using creates sub-directories and a great many files within the working directory structure. Some of these directories are arranged to provide "version control" – this means you need to take care that you always access and work on the most recent version of relevant files. Do not simply accept default file locations offered to you by the software when creating projects – apart from anything else these are almost invariably on the C: drive, which must not be used for such files.

Make sure Windows is configured so that file extensions are visible – you need to be able to find files generated by ModelSim and Synplify, and without visible extensions this can be very difficult. In the root of your H: drive, click on "Organise" → "Folder and search options". Pick the View tab, **un**check "Hide extensions for known file types" then click "Apply to Folders".

### 2.2 SystemVerilog

Study the examples in your ELEC1202 notes and answer the following questions in your logbook:

- In what order do SystemVerilog **assign** statements execute in a **module** description?
- What is the difference between the following SystemVerilog processes: **always**, **always_ff** and **always_comb** ?
- Why do state machine descriptions typically contain more than one process?
- Write out in SystemVerilog an **initial** process to implement a 20MHz clock.
- Write out the SystemVerilog **module** code for a simple state machine which behaves as follows:
  - If input A is at logic '1', move to state S1 and set output Y to '0'
  - Otherwise move to state S2 and set output Y to '1'
- Modify the SystemVerilog code for the bit pattern recogniser in section 3.2 to implement a Moore rather than a Mealy design of the state machine.

## 3 Practical work – using ModelSim and SynplifyPro in hardware design

Check that you can access your pre-prepared SystemVerilog files and can run the software tools: ModelSim and Synplify Pro.

### 3.1 Combinational logic example - 2-to-1 multiplexer

Prepare two text files with the SystemVerilog source code for the 2-to-1 multiplexer and its testbench as follows:

```
// Simple 2-to-1 mux ------------------------------------------------
// File Name  - mux.sv
// Last revised - 10 Feb 2009
// Author - tjk
//--------------------------------------
module  mux(
  input  logic[1:0]  din  , // Mux data input
  input  logic  sel       , // Select input
  output logic  dout        // Mux data output
  );
//------------- mux behaviour ---------
always_comb // infers combinational logic
  if (sel == 1'b0)
    dout = din[0];
  else
    dout = din[1] ;

endmodule //End Of Module mux----------------------------------------


// 2-1 mux testbench ------------------------------------------------
// File name - testmux.sv
// Last revised - 16 Feb 2009
// Author - tjk
// --------------------------------------
module testmux;
  logic[1:0] din;
  logic sel, dout;

// invoke an instance of the mux
mux m (.*); // .* syntax for port mapping works
// if testbench and mux use the same signal names
```

```
initial
  begin
  din <= 2'b01;
  sel <= 1'b0;
  #25ns sel <= '1;
  #25ns din <= 2'b11;
  #25ns din <= 2'b10;
  #50ns sel <= '0;
  end
endmodule// End of Module testmux ---------------------------------------
```

The file name extension should be .sv. You can modify the testbench code to provide more test cases. Create a separate directory on your H: drive for the multiplexer project and save both SystemVerilog source files in it.

Start ModelSim and click on File→New→Project. This brings up the Create Project window as shown in Figure 1.

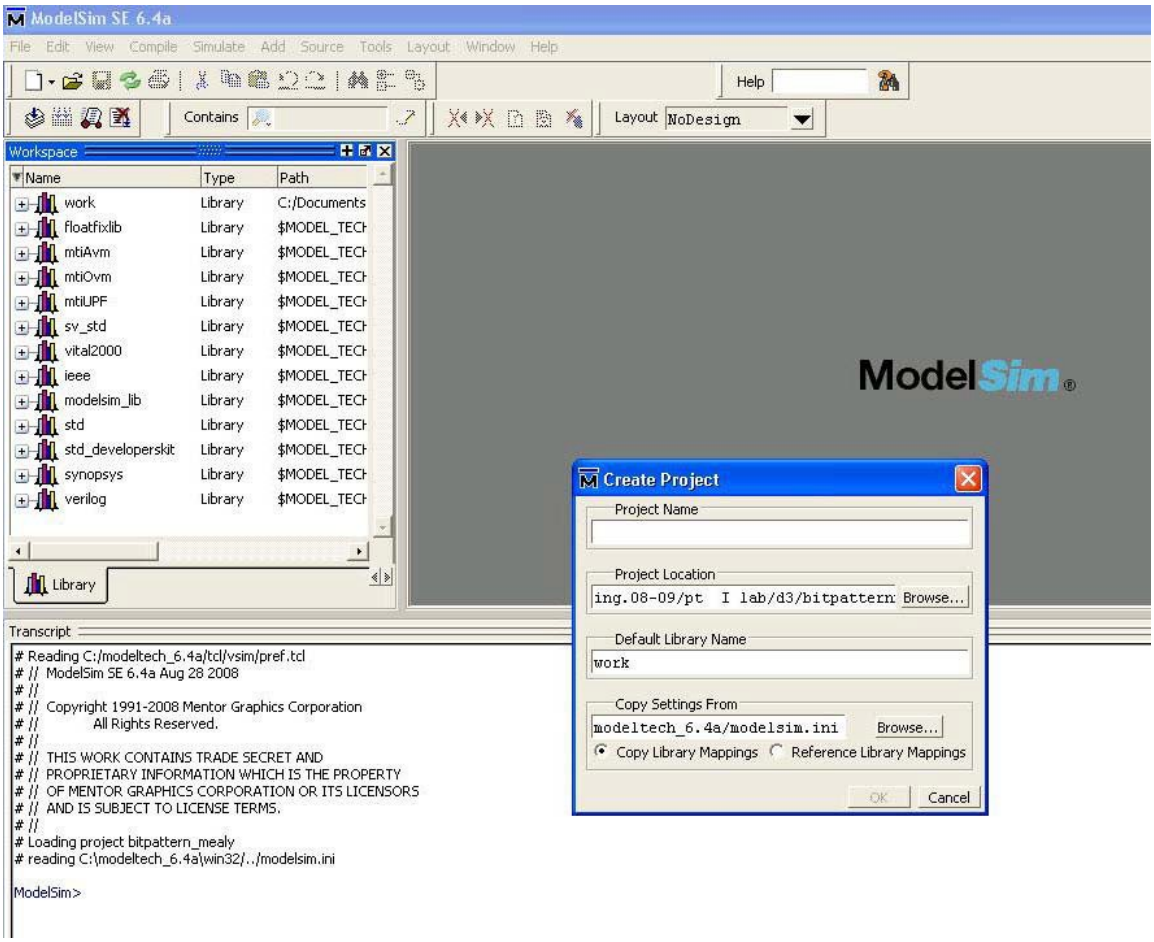


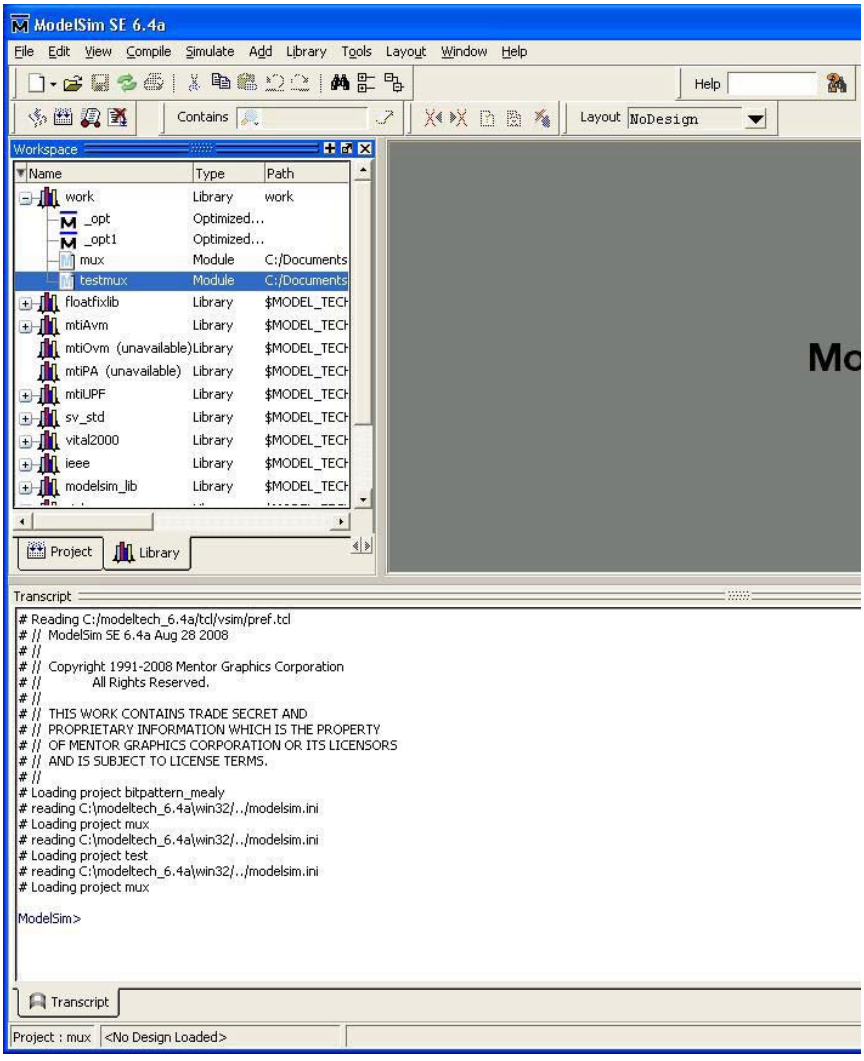

**Figure 1. Create a new project in ModelSim.**

Give the new project a name, such as ‘mux’, and specify the directory where you saved your files as the project location. Click OK. A window appears to add source files to the project. (Figure 2).

Use ‘Add Existing File’ to add both source files to the project and click Close. Now compile the files by selecting Compile→Compile All from the main menu. If there any syntax errors, suitable messages will appear in the Transcript pane of the main ModelSim window. Click on an error message to open a window with the error location and details.


**Figure 2. Add your source files to the project.**

When the compilation is successful, load the design. Select the Library tab, open the work library and double click on the testbench file name.




If the design cannot be loaded, analyse the messages in the Transcript window to find out why. The most common causes of errors at this stage are mismatches between module names or module ports in the project files.

If the design loads successfully, start the simulation environment by selecting Simulate→Start Simulation from the main menu and then double clicking on the testbench file name in the 'Start Simulation' window that opens. Click OK to close the 'Start Simulation' window.

Open the waveform viewer by selecting View→Wave. Add all the signals in the design to the

waveform viewer by right-clicking on a signal name in the Objects pane and selecting Add→To Wave→All items in design from the pop up window.

You are now ready to simulate the design. Enter an appropriate simulation time in the simulation time pane, e.g. 600ns, and click on the Run icon which is placed next to the simulation time pane. You can also start a simulation by selecting Simulate→Run→Run All from the main menu. Analyse the waveforms in the waveform viewer (see Figure 3) to verify whether the design works correctly.

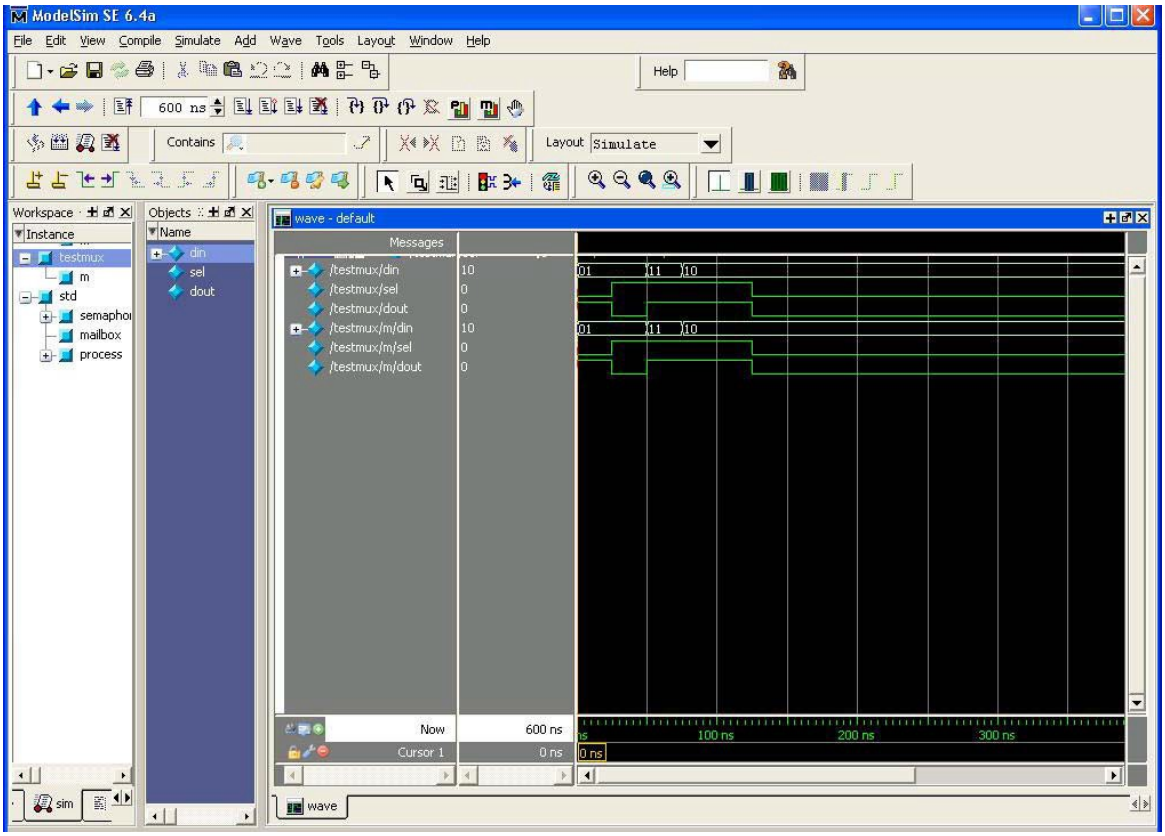



**Figure 3. Multiplexer waveforms .**

When you are satisfied that the multiplexer works as expected, close ModelSim and start Synplify Pro to synthesise the multiplexer description into hardware. Figure 4 shows the SynplifyPro start window.

Click on ‘Add File’ , uncheck the ‘Use relative paths’ option, navigate to your multiplexer module SystemVerilog file and add it to the SynplifyPro project. **Do not include your testbench** in the project. The purpose of a testbench is to provide signal excitations for testing. There is little point in trying to turn a testbench into hardware.

Select Options→Configure Verilog Compiler. The ‘Implementation Options’ window opens. Check SystemVerilog on the Verilog Language pane. Note that you will have to repeat this step before each simulation. Then click on the ‘Device Tab’ to select a target device. Select ‘Lattice ispGAL’ as from the Technology list and ‘ispGAL22V10B’ from the Part list. Click OK to close the ‘Implementation Options’ window.
You are ready to synthesise. Click on the large Run button and expect a slight delay because hardware synthesis is a very CPU intensive process. Analyse the messages in the Messages pane to make sure that the synthesis is successful. Note that Synplify might complain about the SystemVerilog syntax in your files even if the ModelSim compilations were successful. This might be because both tools implement different subsets of the SystemVerilog standard, or because the

design is not synthesisable even though it may have simulated correctly.

If the synthesis was successful you can view RTL circuit level diagrams of the hardware implementation. Select HDL-Analyst→RTL from the main menu to view the diagrams as illustrated in Figure 5.

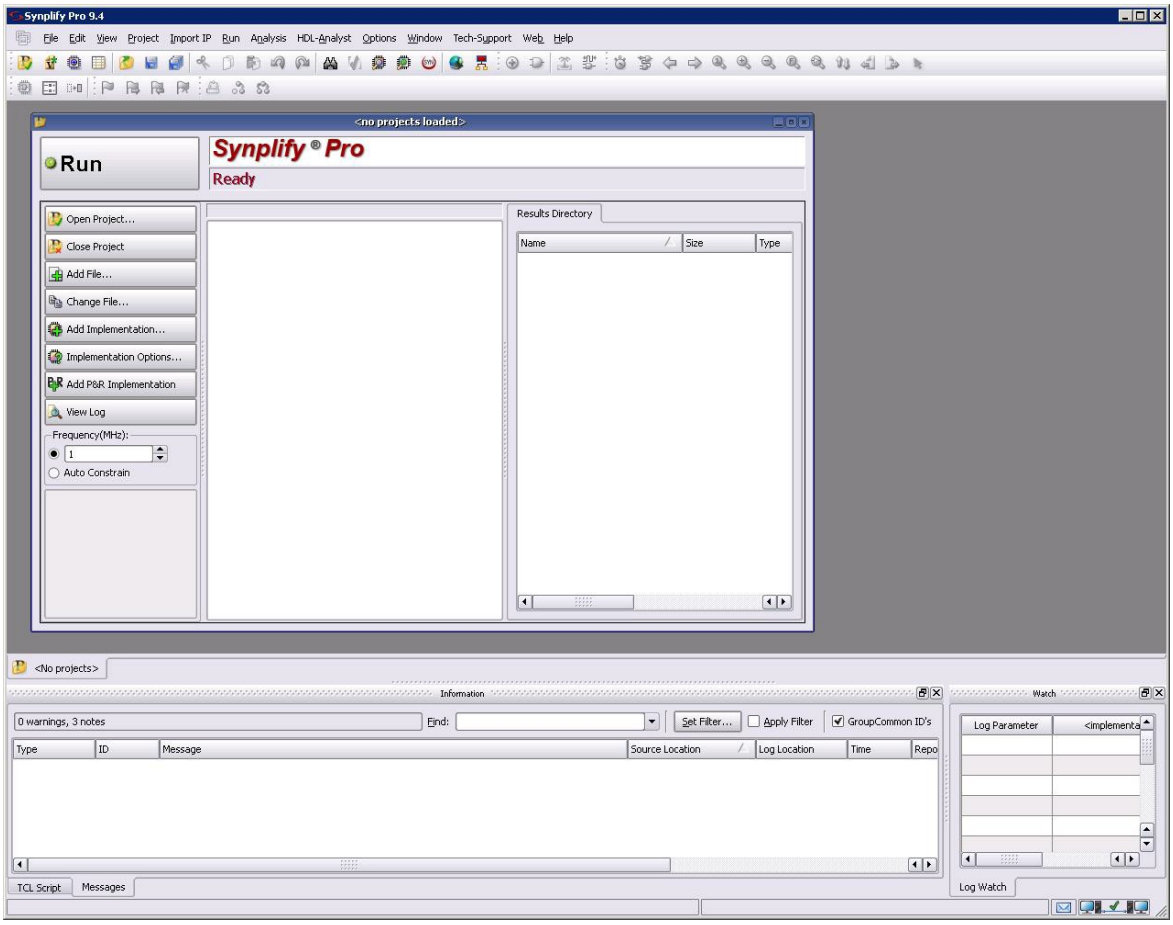



**Figure 4. Synplify start window.**

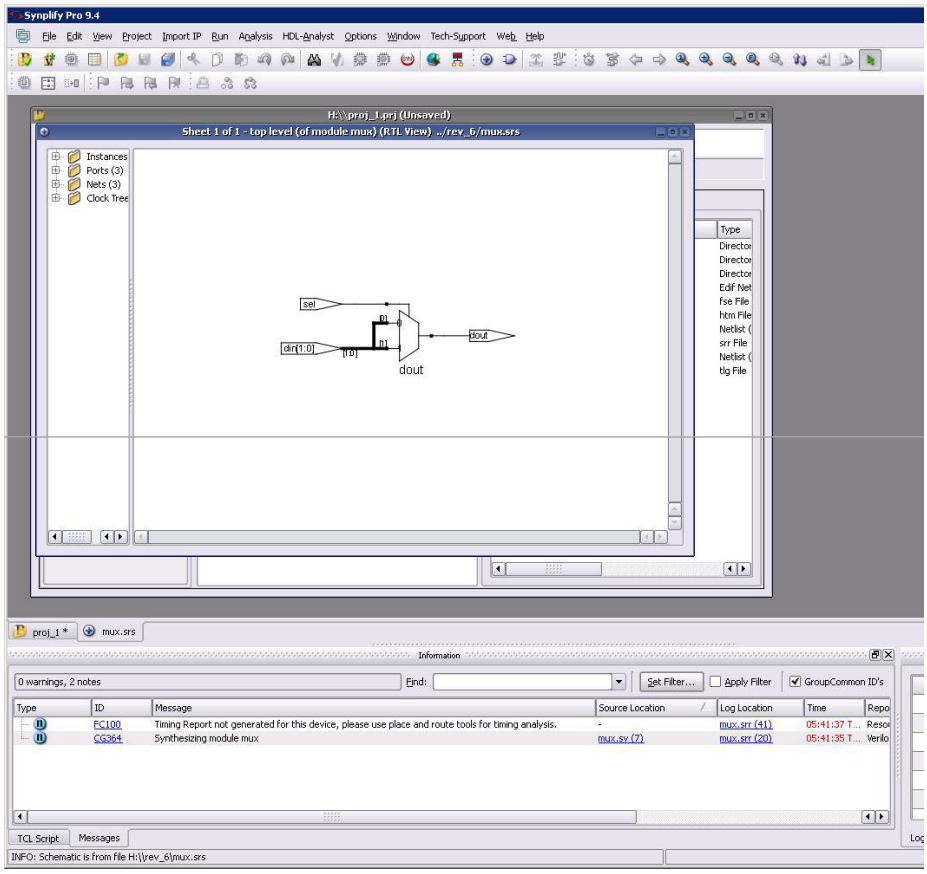



**Figure 5. RTL diagram of synthesised hardware.**

## 3.2 Simple Mealy state machine - bit pattern recogniser

Repeat the simulation and synthesis steps from section 3.1 to implement in hardware a bit pattern recogniser using the following SystemVerilog files:

```
// state machine to recognise bit pattern 101 - Mealy version----------------
// File name: bitpatternsm.sv
// Last modified: 16/02/09
// Author: tjk
// -----------------------------------------
module bitpatternsm(
  input logic inp, clock, nreset,
  output logic outp
  );
// ----------- local declarations -------
typedef enum logic[1:0] {start, cycle1, cycle2, unused} states;

states state,next; // local signals for present and next state

// fsm behaviour consists of two processes,
// one for the memory and one for combinational logic

// always_ff process implements the flip-flops
// the process is sensitive to pve clock edge and async reset
  always_ff@(posedge clock or negedge nreset)
    if (~nreset) // active low asynchronous reset
      state <= start;
    else
      state <= next;

// combinational logic process, implements both next state and output
```

```
  always_comb
    case(state)
    start: //behaviour in state start
      begin
      outp <= '0;
      if(inp)
        next <= cycle1; // first bit is 1
      else
        next <= start;
      end

    cycle1:     // behaviour in state cycle1
      begin
      outp <= '0;
      if(inp)
        next <= cycle1; // first bit, stay in cycle1
      else
        next <= cycle2; // second bit = 0
      end

    cycle2:     // behaviour in state cycle2
      begin
      outp <= inp; // assert output if inp==1 (third bit)
      // nb. this is Mealy behaviour; why?
      next <= start; // start over again
      end

    unused: // unused state, proceed to start
      begin
      outp <= '0;
      next <= start;
      end
    endcase

endmodule // End Of Module bitpatternsm ------------------------------------


// testbench for bit pattern recogniser -------------------------------------
// File name: bitpatternsm_test.sv
// Last modified: 16/02/09
// Author: tjk
// -------------------------
module bitpatternsm_test();

// declare signals for fsm ports
  logic inp, clock, nreset, outp;

  bitpatternsm sm (.*); // create fsm instance

  always // clock process
    begin
    clock = 0;
    #50ns clock = 1;
    #50ns clock = 0;
    end

  initial // test the machine
    begin
    inp= 0;
    nreset = 0; // assert reset

    #100ns nreset = 1; // remove reset
    #100ns inp = 1; // first bit
    #100ns inp = 0; // 2nd bit
    #100ns inp = 1; // 3rd bit
    #100ns inp = 0; //
    end
```

```
endmodule // End of Module bitpatternsm_test ------------------------------
```

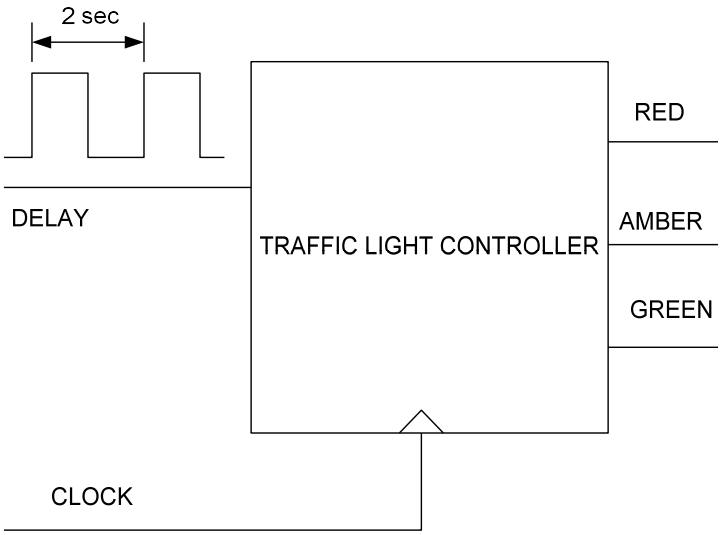
### 3.3 Moore state machine

Using the file you created as part of your preparation, repeat the simulation and synthesis procedures of section 3.2 for the bit pattern recogniser implemented as a Moore state machine. This may well give you experience of debugging to clear errors, including syntax and other errors due to mistyping. All this debugging work should be recorded carefully in your logbook, so that you have a reference showing how to locate and correct errors in System Verilog source.

Can you use the same testbench for this Moore machine implementation as for the Mealy implementation? If not, what changes are required? If so, can the testbench be extended so as to allow the simulation to determine whether the implementation uses a Moore or a Mealy machine? Make sure these questions are answered clearly in your logbook.

### 3.4 Further work - a traffic light controller

If you have time, develop a hardware implementation of a simple traffic light controller that cycles through the following four states: RED → RED_AND_AMBER → GREEN → AMBER. The system should have three outputs; RED, AMBER and GREEN as shown in Figure 6. The challenge here is to use a fast clock (say 1kHz) and a timing reference waveform to generate delays of 2 seconds or more between the traffic light phases. Produce a suitable ASM chart, develop a SystemVerilog description and then simulate your design in ModelSim and synthesise into hardware in SynplifyPro.




.

**Figure 6. Traffic light controller with a fast clock and delay reference.**

## 4 Recording your work

Keep a complete record of your SystemVerilog source code, ModelSim simulation results, synthesised RTL and Technology diagrams as well as other details of the design and tests (successful or not) in your laboratory logbook. This includes sticking a printout of at least the final version of your source code into the record of this exercise.

TJK, February 2009.
Minor update GVM/TMF/KM3 April 2011
Update MZ, October 2012
Revised MZ, July 2013