Logic Design with SystemVerilog (continued)

31/01/2022

In sessions 1 and 2, we studied the digital system design process, and then covered the structural modelling of digital circuits using SystemVerilog built-in primitives and user defined primitives (UDPs). We described hierarchical design using SystemVerilog.

We looked at behavioural modelling with SystemVerilog, and description of level-sensitive and edge-sensitive sequential logic (latches and flip flops). We covered concurrent (non-blocking) assignments and the concept of dataflow (register transfer level) modeling.

In this session we'll cover the design of testbenches, and the synthesis of digital circuits from SystemVerilog descriptions.

7.1 Test benches

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit (the device under test, or DUT) in order to test it and observe its response during simulation.

The examples in the following slides are presented to demonstrate some basic features of HDL stimulus modules.

The following testbench applies a continuous sequence of clock pulses, using the keyword **forever**. Every 10 time units (in this case, every 10 ns), the clock signal, *clk*, changes its value between 0 and 1.

```
//Device under test

module dff_sync_reset(input d, clk, reset,
                                output q);


always @(posedge clk)
begin
 if (reset)
  q <= 1'b0;
 else
  q <= d;
end

endmodule
```

```
module my_testbench();      // Testbench
logic d, clk, reset, q;

d_ff_sync_reset dut(d, clk, reset, q);

initial begin
  clk = 0;
    forever #10 clk = ~clk;
end


initial begin
 reset = 1;
 d = 0;
#25; reset=0; d = 1; #30;  d = 0; #20;  d = 1;
end

endmodule
```

Example - Self-checking Testbench

Device under test:

```
module myfunction(input  logic a, b, c,
                  output logic y);
    assign y = ~b & ~c | a & ~b;
endmodule
```

```
module my_testbench();
  logic  a, b, c;
  logic y;
  myfunction dut(a, b, c, y);  // instantiate dut
  initial begin // apply inputs, check results one at a time
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
end
endmodule
```

7.2 Synthesis

A synthesis tool uses the SystemVerilog model of the system to produce a *netlist* of standard cells for an ASIC implementation, or a database that will configure a target FPGA.

It carries out the following tasks:

1. Detect and eliminate redundant logic
2. Detect combinational feedback loops
3. Exploit don’t-care conditions
4. Detect unused states
5. Detect and collapse equivalent states
6. Make state assignments
7. Synthesise optimal, multilevel realisations of logic, taking into account constraints on area and/or speed in a physical technology
8. Provide timing information

Synthesis of Combinational Logic

Synthesisable combinational logic can be described by (1) a netlist of structural primitives, (2) a set of continuous-assignment statements, and (3) a level-sensitive behaviour.

A design that is expressed as a netlist of primitives should be synthesised to remove any redundant logic before mapping the design into a technology.

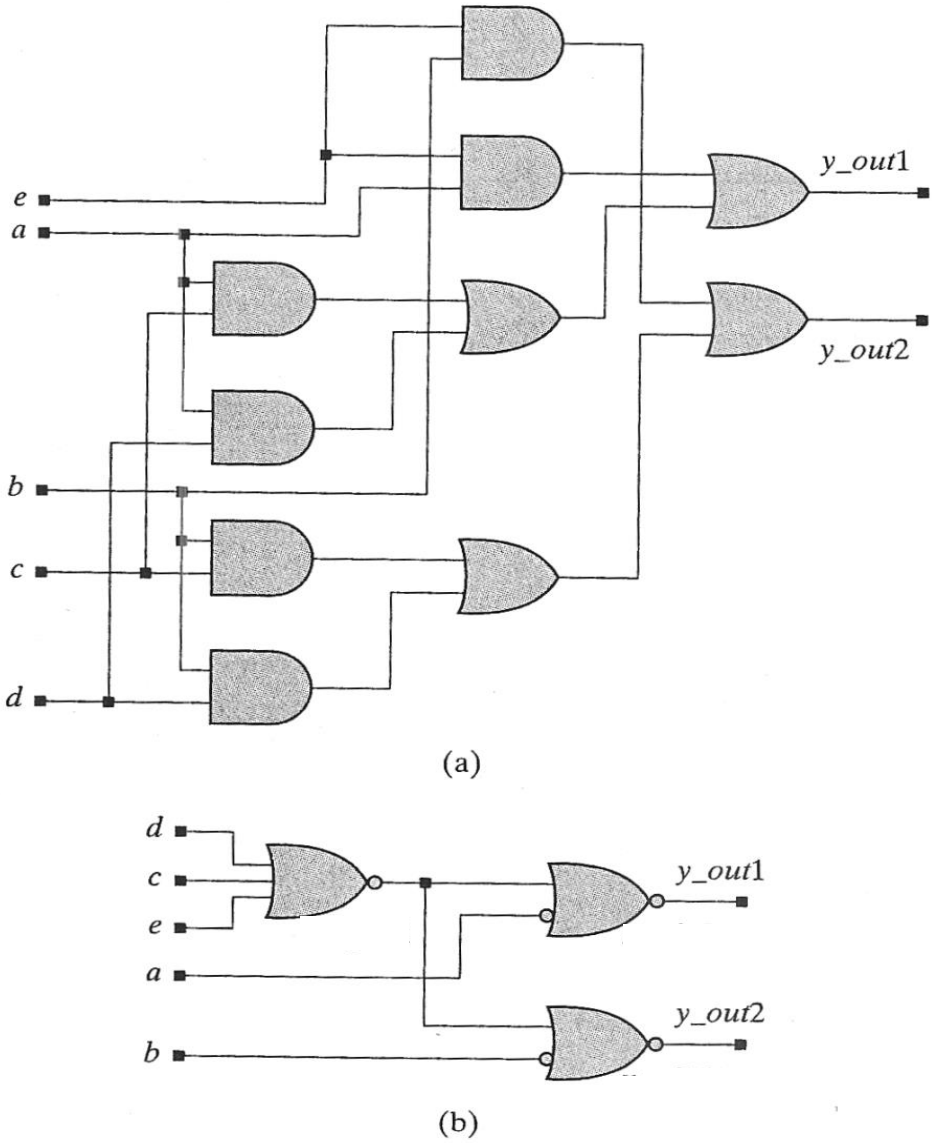
Example

Figure (a) shows the pre-optimised schematic of the circuit described by the netlist of primitives in *boole_opt*. The synthesised circuit shown in Figure (b) has a more efficient implementation.

```
module boole_opt(output logic y_out1, y_out2,
                 input logic a, b, c, d, e)

 and  (y1, a, c);
 and  (y2, a, d);
 and  (y3, a, e);
 or   (y4, y1, y2);
 or   (y_out1, y3, y4);
 and  (y5, b, c);
 and  (y6, b, d);
 and  (y7, b, e);
 or   (y8, y5, y6);
 or   (y_out2, y7, y8);

endmodule
```



Synthesis of Conditional Statements

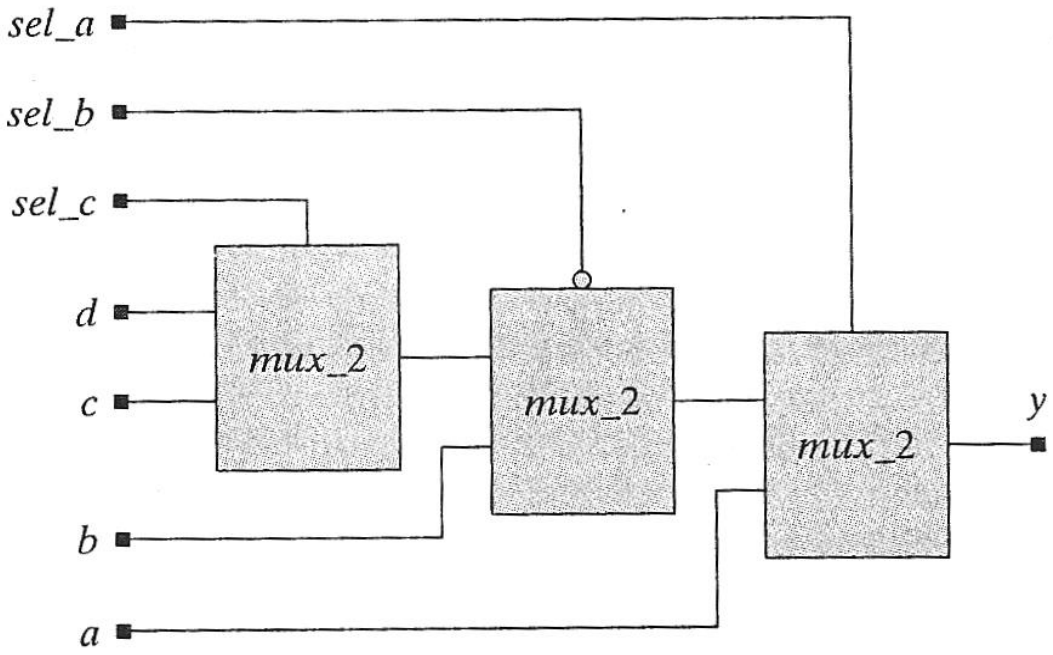
SystemVerilog **case** statements, **if** statements and conditional operator (? :) will synthesise to multiplexer structures.

For example:

```
module mux_4pri(output logic y,
                input logic a, b, c, d, sel_a, sel_b, sel_c);

  always_comb
  begin
    if    (sel_a == 1)      y = a; else
     if   (sel_b == 0)      y = b; else
      if (sel_c == 1)       y = c; else
                            y = d;
  end

endmodule
```



Exploiting Don’t-Care Conditions

When case, conditional branch (**if**), or conditional operator (? :) statements are used, the behavioural model and the synthesised netlist should produce the same simulation results if the code has default assignments that are purely 0 or 1.

Simulation results may differ if the default or branch statements make an assignment of x.

The items that decode to explicit x will be treated as ‘don’t-care’ conditions for the purpose of logic minimisation.

Note that the physical hardware will propagate either a 0 or a 1, while the SystemVerilog simulation will propagate an x in simulation.

This may lead to a mismatch between the results obtained by simulating the SystemVerilog code and the synthesised design.

Resource sharing

A synthesis tool must recognise whether the physical resources required to implement a complex behaviour can be shared.

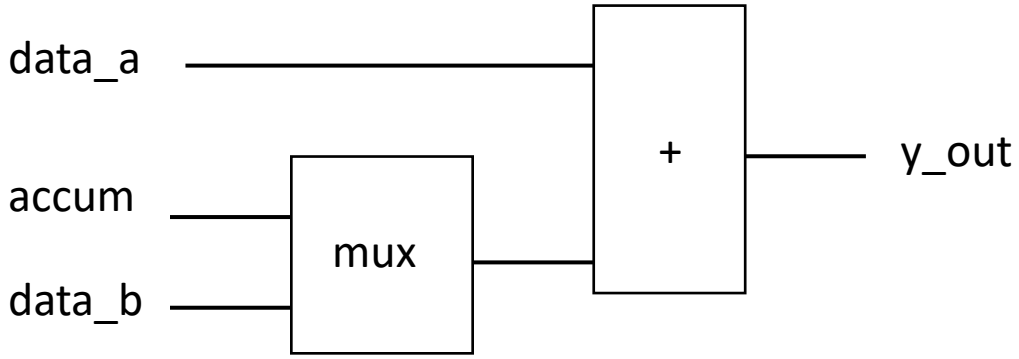
If the data flows within the behaviour do not conflict, the resource can be shared between two or more paths.

Example Resource sharing

The addition operators in the continuous assignment below can be shared in hardware

```
assign y_out = sel ? data_a + accum : data_a + data_b
```

The operators can be implemented by a shared adder, whose input data paths are multiplexed.



Synthesis of Sequential Logic with Latches

Incomplete *if* statement

A synthesis tool infers the need for a latch when a variable in a level-sensitive behaviour is assigned a value in some sequence of executed statements in a set of conditional branch statement (***if*** ... ***else***), but not in others (e.g. an incomplete **if** statement).

This implies the need for the variable to retain the value it had before the behaviour was activated.

The control signal of a given latch will be the signal whose value controls the branching of the activity.

Incomplete *if* statement

An example of poorly written code, intended to describe combinational logic, but which will synthesise a latch:

```
module unintentional_latch(output logic    [3:0]    data_out,
                           input logic     [3:0]    data_in,
                                  input logic    enable);
    always_comb
        if (enable) data_out = data_in;          // Incompletely specified

endmodule
```

In using the **always_comb** keyword, it seems the coder intended to describe combinational logic.

However, a synthesis tool would infer latches when synthesizing the statements in this construct.

These latches are unintentional and caused by incomplete assignments to variables in the conditional statement.

Incomplete *if* statement

The synthesis tool may remove false latches for some conditional statements.

It may infer a latch initially, but later on, during optimization, will remove the latch as unnecessary.

It is advisable not to use such descriptions. If it's intended to describe latched logic, then the **always_latch** construct should be used.

Otherwise, when describing purely combinational logic, verify that you assign an updated value to every variable in all possible paths through the **always_comb** construct.

Synthesis of Sequential Logic with Latches

Complete *if* statement with feedback

Alternatively, a latch will be synthesised using **if** statements or the conditional operator (? :) with feedback (i.e. if the output variable of the assignment appears in an expression of the operator.

Complete *if* statement with feedback

Example Latch circuit from a complete **if** *statement*

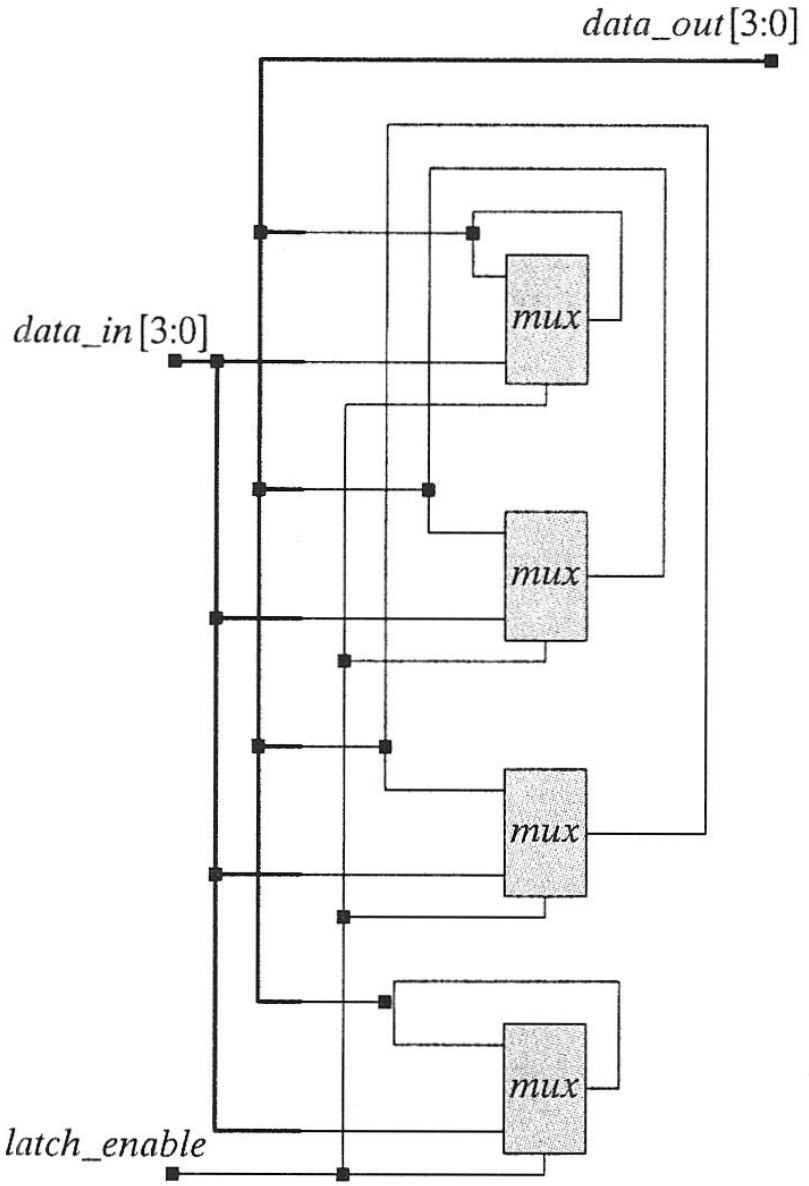
The description given by *latch_if1* below assigns *data_out* with feedback in an **if** statement in a level sensitive cyclic behaviour.

The result of synthesising the latch circuit, shown on the next slide, has the structure of a multiplexer with feedback.

```
module latch_if1(output logic [3:0]   data_out,
                 input logic   [3:0]  data_in,
                 input log            latch_enable)

 always_latch
    if (latch_enable) data_out = data_in;
   else data_out = data_out;

endmodule
```



Synthesis of Sequential Logic with Flip-Flops

Flip-flops are synthesised from edge-sensitive cyclic behaviours which include conditional behaviours (e.g. ***if ... else*** statements, or ***case*** statements).

Example

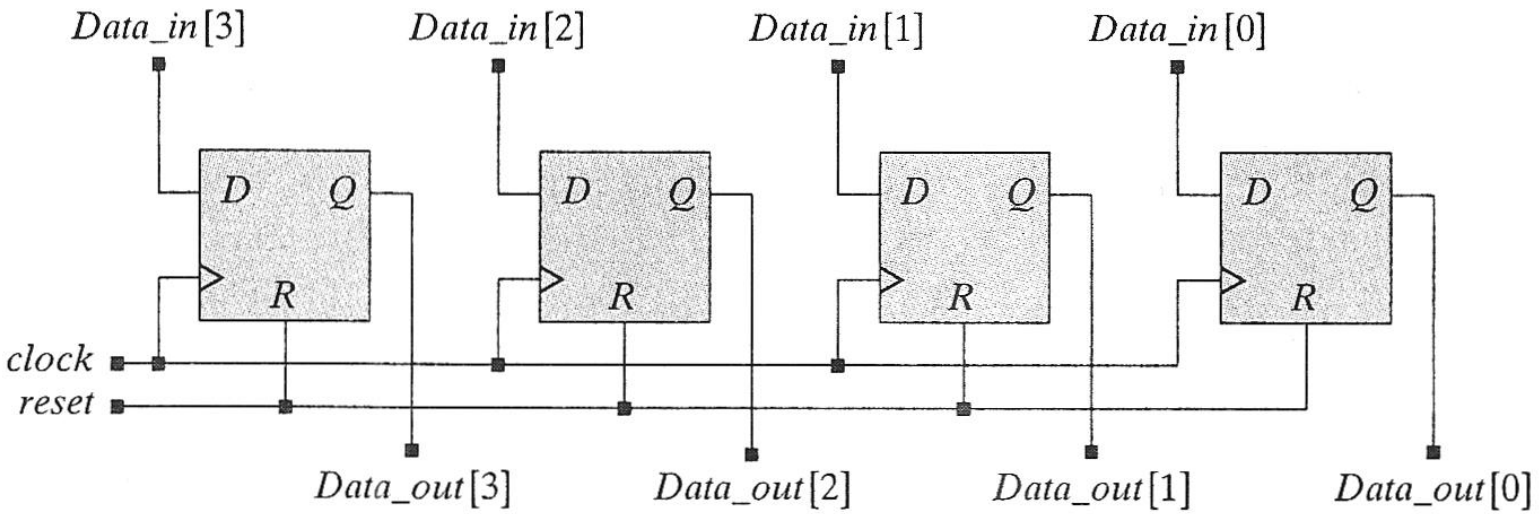
The functionality of a 4-bit parallel-load register, shown below, is described by *D_reg4_a*.

```
module D_reg4_a (output logic   [3:0]   Data_out,
                 input logic    [3:0]   Data_in,
                 input logic            clock, reset);

 always_ff @ (posedge clock or posedge reset)
   begin
    if (reset) Data_out <= 4'b0;
      else Data_out <= Data_in;
   end
endmodule
```

Example (continued)

From the edge-sensitive cyclic behaviours with conditional statements, the synthesis tool infers the need for resettable D-type flip-flops, as shown on the following slide.



State encoding

The method for designing a sequential machine requires that a set of flip-flops be chosen to represent the state of the machine, and that a unique binary code be assigned to each state.

The task of assigning a code to the states of a machine is called *state assignment* or *state encoding*.

For complex systems, the task of manually carrying out optimised state encoding is not possible. Hence, computer-based heuristic methods are used instead.

Algorithms are used by the synthesis tool to search for a good state assignment, leading to minimal logic requirements.

Synthesis of Loops

Static loops

A loop in a cyclic behaviour is said to be static, or data-independent, if the number of iterations can be determined by the compiler *before* simulation. (i.e. the number of iterations is fixed, and independent of the data).

In this case, the iterative computation sequence has a non-iterative counterpart, which can be obtained by the synthesis tool by unrolling the loop.

The operations in the unrolled loop can occur at a single time step of the simulator.

Example Synthesis of a static loop

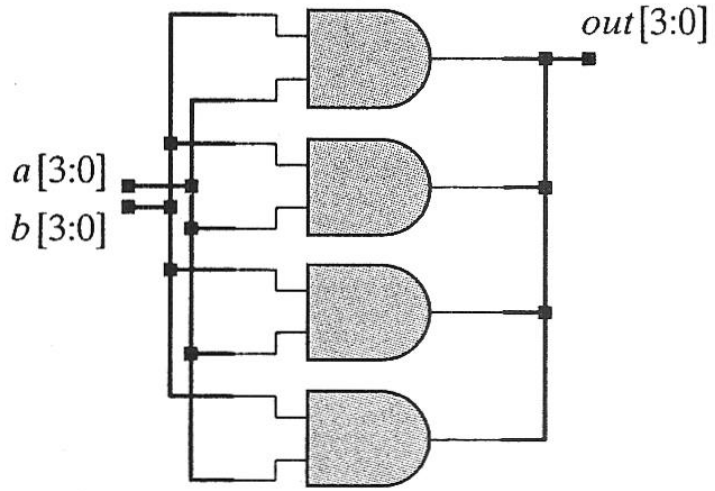
The loop in *for_and_loop_comb* does not depend on the data, and iterates for a fixed, predetermined number of steps and terminates.

```
module for_and_loop_comb (output logic   [3:0]   out,
                          input logic    [3:0]   a, b);
 reg        [2:0]   i;

 always_comb
  begin
   for (i = 0; i <= 3; i = i + 1)
     out[i] = a[i] & b[i];
  end
endmodule
```

```
out[0] = a[0] & b[0];
out[1] = a[1] & b[1];
out[2] = a[2] & b[2];
out[3] = a[3] & b[3];
```

The unrolled loop is equivalent to the following assignments, and the module is synthesised to the circuit:



Nonstatic loops

A nonstatic or data-dependent loop has a number of iterations which is dependent on the data.

It may be implemented in a multicycle operation – the activity of the loop is distributed over multiple cycles of the clock.

In contrast to static loops, the iterations of a non-static loop must be separated by a synchronising edge-sensitive event control expression in order to be synthesised.

Conclusion to Digital Design using SystemVerilog

This completes the material on digital design using the SystemVerilog hardware description language.

We have studied both structural and behavioural models written in SystemVerilog, the use of testbenches for simulations, and the tasks carried out by synthesis tools to create the physical circuit design based on the SystemVerilog description.

Part 1 of the ELEC0010 module will be completed in Session 4, in which we will study technologies for complementary metal oxide semiconductor (CMOS) circuits, and timing and power consumption in CMOS digital electronics.