Introduction to Logic Design with SystemVerilog (continued)

In session 1, we studied the digital system design process, and then covered

- Structural modelling of digital circuits using SystemVerilog built-in primitives and user defined primitives (UDPs).
- We described hierarchical design using SystemVerilog
- Behavioural modelling of combinational logic
- Test benches

In this session, we will look at behavioural modelling of sequential logic with SystemVerilog.

6. Behavioural modelling of sequential logic

Besides modelling sequential logic systems with user defined primitives, as discussed previously, there are a number of ways their behaviour can be described in SystemVerilog:

- continuous assignment statements, and
- cyclic beviour.

6.1 Latches and Level-Sensitive Circuits in SystemVerilog

The level-sensitive storage mechanism of a latch can be modelled using continuous-assignment statements.

When feedback is used in a continuous-assignment statement with a conditional operator, a synthesis tool will infer the functionality of a latch and its hardware implementation.

Example

Write a SystemVerilog model of a latch using a continuous assignment statement with feedback

```
module Latch_CA (input logic data_in, enable,
                 output logic q_out);

  assign  q_out = (enable) ? data_in : q_out;

endmodule
```

6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

Continuous-assignment statements are limited to modelling level-sensitive behaviour

i.e. combinational logic and transparent latches

They cannot model an element that has signal pulse edge-sensitive behaviour, such as a flip-flop.

Verilog uses a **cyclic behaviour** to model edge-sensitive functionality.

6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

Cyclic behaviours are abstract – they do not use hardware to specify signal values. Instead, they execute procedural statements, just like the statements of an ordinary procedural program (e.g. Java).

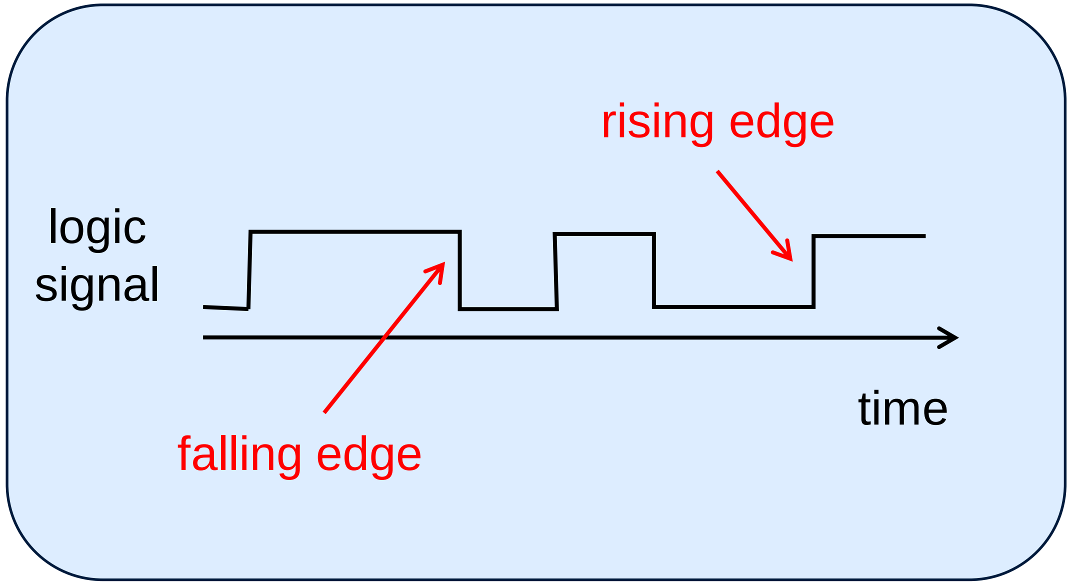
They are called *cyclic*, because they keep re-executing, every time another edge occurs.

Cyclic behaviours can be used to model (and synthesise) both level-sensitive and edge-sensitive (synchronous) behaviour.

6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

Keywords: **posedge**, **negedge**

Edge semantics for rising (**posedge**) and falling (**negedge**) edges are built into Verilog.



6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

The keyword **always_ff** in the Verilog model *df_behav* declares a cyclic behaviour corresponding to an edge-triggered flip-flop.

At every rising edge of *clk*, the behaviour's procedural statements execute, computing the value of *q* and storing it in memory.

The nonblocking, or concurrent, assignment operator **<=** will be explained later

```
module df_behav (input logic data, set, clk, reset,
                 output logic   q, q_bar);
assign  q_bar = ~q;

always_ff @ (posedge clk)
begin
  if (reset == 0) q <= 0;
  else if (set == 0) q <= 1;
  else q <= data;
end
endmodule
```

6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

The keyword **always_latch** can be used to declare a cyclic behaviour of level-sensitive, non-synchronous sequential logic. Unlike **always_ff**, **always_latch** does not use an event control expression, but is triggered by a change in a dependent variable.

For example, a model of a transparent latch is shown below.

```
module tr_latch (output logic q_out,
                 input logic enable, data);

always_latch
if (enable) q_out = data;

endmodule
```

6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

The keyword **always_comb** can be used to declare a combinational logic operation as a cyclic behaviour, as shown in the following example:

always_comb mywire = a & b

As with **always_latch**, **always_comb** is triggered by a change in a dependent variable, and does not use an event control expression.

6.2 Cyclic Behavioural Models of Flip-Flops and Latches; Edge Detection

The keywords **always_ff** can also be used to declare cyclic behaviours with an event-control expression based on more than one signal.

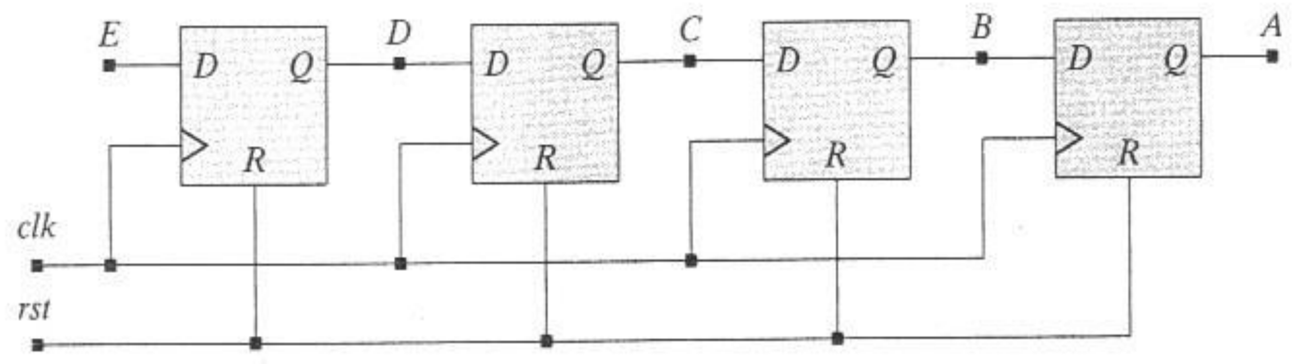
For example, in the functionality of an asynchronous flip-flop modelled in *asynch_df_behav*, the state of the flip-flop is sensitive to the rising edge of the clock, *clk*, but also to the falling edges of *reset* and *set*.

```
module asynch_df_behav (input logic     data, set, clk, reset,
                        output logic   q, q_bar);
assign  q_bar = ~q;

always_ff @ (negedge set or negedge reset or posedge clk)
begin
  if (reset == 0) q<= 0;
  else if (set == 0) q<= 1;
  else q <= data;
end
endmodule
```

6.3 Non-blocking (Concurrent) Assignments in Cyclic Behaviours

Example

Write a SystemVerilog model of the 4-bit shift register shown below



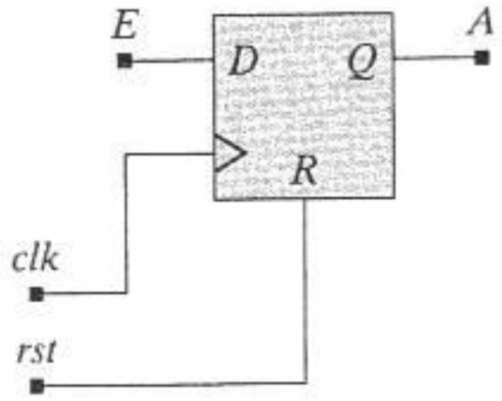
```
module  shiftreg_PA (input logic   E, clk, rst,
                                   output logic A);
logic       B, C, D;

  always_ff @ (posedge clk or posedge rst)
begin
  if (rst) begin A = 0; B = 0; C = 0; D = 0; end
  else begin
    A = B;
    B = C;
    C = D;
    D = E;
  end
end
endmodule
```

6.3 Non-blocking (Concurrent) Assignments in Cyclic Behaviours

Example (continued)

Now consider what happens if the order of the procedural assignments is reversed:

```
module  shiftreg_PA (input logic   E, clk, rst,
                     output logic A);
logic      B, C, D;

 always_ff @ (posedge clk or posedge rst) begin
 if (rst) begin A = 0; B = 0; C = 0; D = 0; end
 else begin
   D = E;
   C = D;
   B = C;
   A = B;
 end
end
endmodule
```



Synthesis tools would produce a single D-flip flop!

6.3 Non-blocking (Concurrent) Assignments in Dataflow/RTL Models

The assignment operator in the previous Example is the ordinary procedural assignment operator (**=**).

$\rightarrow$ the statements are executed in the listed order, with storage of values occurring immediately after any statement executes., and before the next statement executes.

This means that the order in which the statements are listed can affect the functionality !

6.3 Non-blocking (Concurrent) Assignments in Cyclic Behaviours

An alternative SystemVerilog dataflow model uses **concurrent procedural assignments** (**<=)**

Also referred to as **non-blocking assignments**.

Non-blocking assignments effectively execute concurrently (in parallel), rather than sequentially

So the order in which they are listed has no effect.

Example

Repeat the previous Exercise - write a SystemVerilog model of the 4-bit shift register, but in this case using non-blocking assignments

```
module  shiftreg_PA (input logic   E, clk, rst,
                                   output logic A);
logic B, C, D;

  always_ff @ (posedge clk or posedge rst) begin
  if (rst) begin A = 0; B = 0; C = 0; D = 0; end
  else begin
    A <= B;        //        D <= E;
    B <= C;        //        C <= D;
    C <= D;        //        B <= C;
    D <= E;        //        A <= B;
  end
end
endmodule
```

Here, the order in which the assignments are listed does not affect the functionality. The list of assignments in this module could be replaced with the commented (//) list, without any effect on the synthesised system.

6.3 Non-blocking (Concurrent) Assignments in Cyclic Behaviours

The general rule-of-thumb to follow is:

- in continuous assignments, use procedural assignments (=)
- in cyclic behaviours, use non-blocking assignments (<=)

6.4 Dataflow/RTL Models

Dataflow models describe concurrent operations on signals, usually in a synchronous machine, where

• computations on the signals stored in registers are initiated at the active edges of the clock, and then

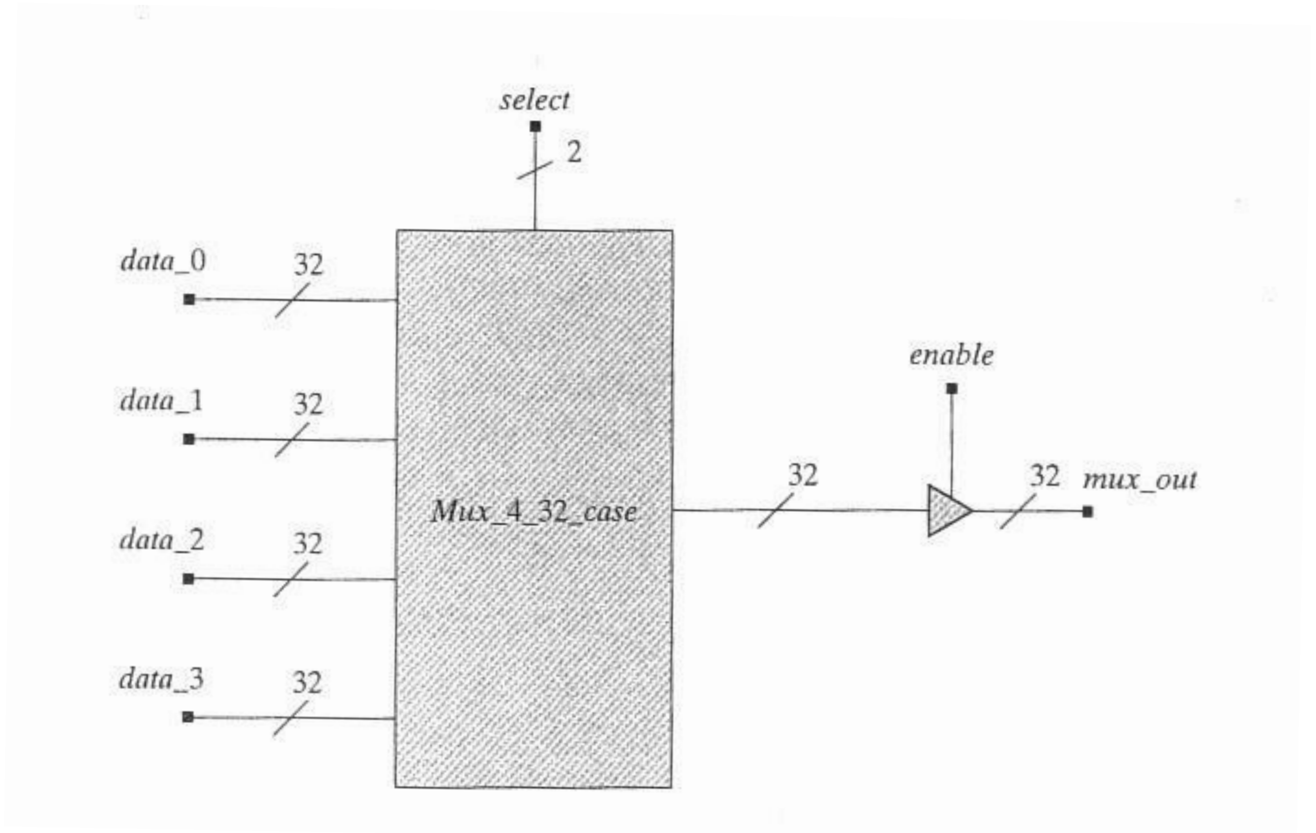
• are completed in time to be stored in a register at the next active edge.

Dataflow models for synchronous machines are also called register transfer level (RTL) models.

6.5 Use of Conditional Statements (case, if)

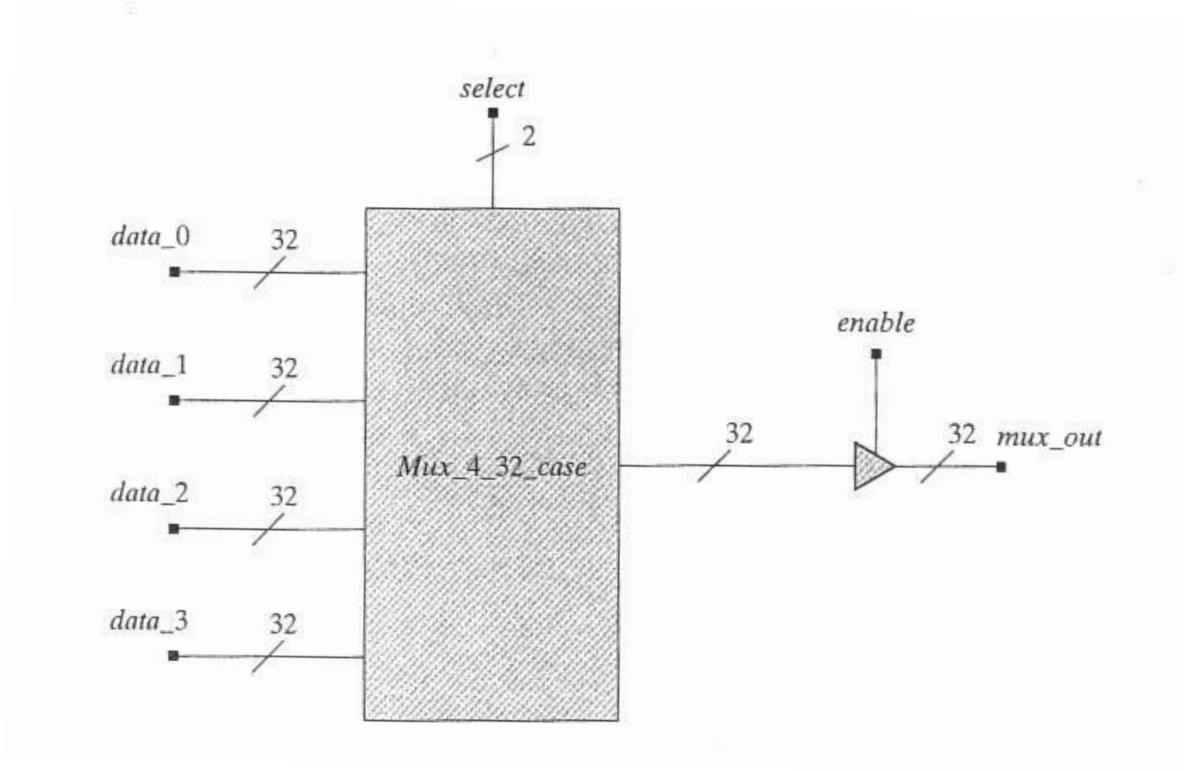
Example Behavioural Model of a Multiplexer with case statements

Mux_4_32_case, on the following slide, is a behavioural model of the four-channel, 32-bit, multiplexer shown below, using the SystemVerilog ***case*** statement, which is similar to its counterpart in other languages (e.g. the ***switch*** statement in C).



Example (continued) Behavioural Model of a Multiplexer with case statements

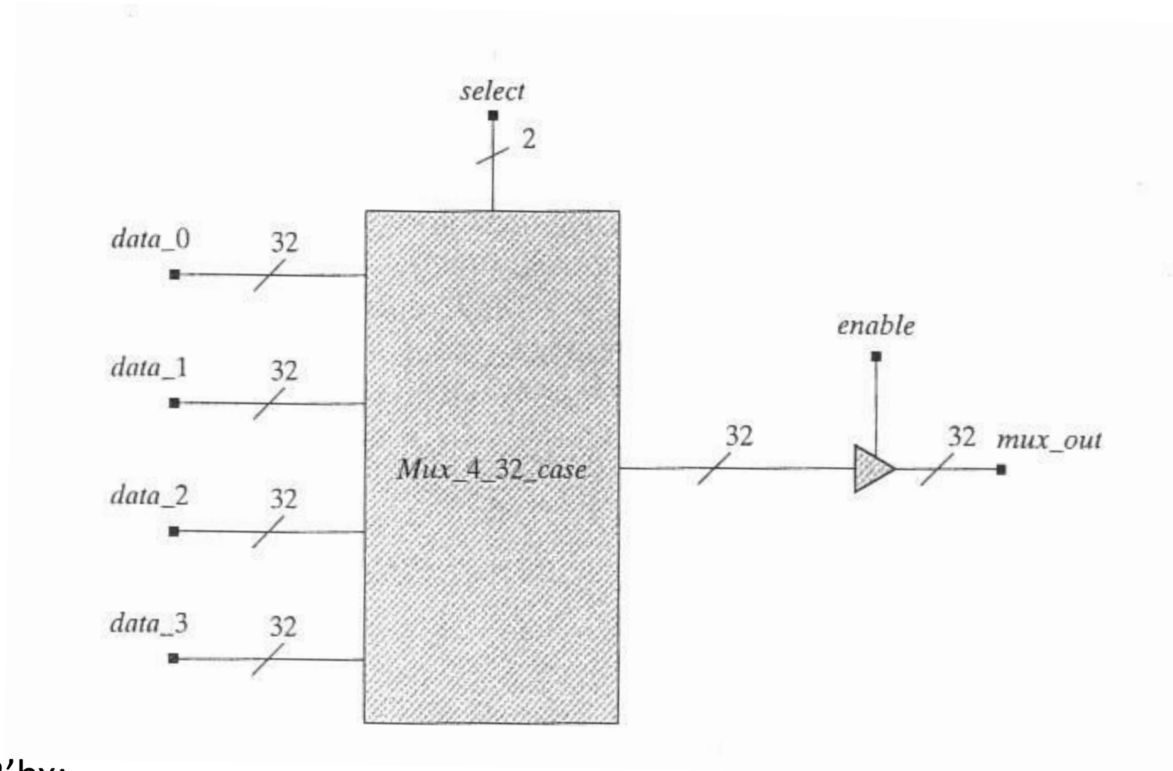
```
module Mux_4_32_case (
input logic     [31:0]   data_3, data_2, data_1, data_0,
input logic     [1:0]    select,
input logic  enable,
output logic  [31:0]   mux_out);

logic      [31:0]   mux_int;

  assign mux_out = enable ? mux_int : 32'bz;

  always_comb
  begin
    case (select)
    0:              mux_int = data_0;
    1:              mux_int = data_1;
    2:              mux_int = data_2;
    3:              mux_int = data_3;
    default:        mux_int = 32'bx;          //Note the use of the default case item, which covers cases
                                              // that might occur in simulation where a case statement
   endcase                                    // is not fully decoded for all possibilities.
  end
endmodule
```



Example Behavioural Model of a Multiplexer with nested conditional statements

Write a model of the multiplexer circuit considered in the previous example, but using nested conditional statements rather than case statements.

```
module Mux_4_32_if(
input logic    [31:0]   data_3, data_2, data_1, data_0,
input logic    [1:0]    select,
input logic  enable,
output logic  [31:0]   mux_out);

logic       [31:0]   mux_int;

  assign mux_out = enable ? mux_int : 32'bz;

  always_comb
  begin
  if (select == 0) mux_int = data_0; else
       if (select == 1) mux_int = data_1; else
         if (select == 2) mux_int = data_2; else
           if (select == 3) mux_int = data_3; else mux_int = 32'bx;
 end
endmodule
```



6.6 Modelling Digital Machines with Repetitive Algorithms

An algorithm for modelling the behaviour of a digital machine may execute some or all of its steps repeatedly in a given machine cycle, depending on whether the steps execute unconditionally or not.

SystemVerilog has four loop constructs for describing repetitive algorithms:

for, repeat, while, forever

*Example A majority circuit using a **for** loop*

A majority circuit outputs ‘1’ if a majority of the bits of an input word are ‘1’. The model *Majority*, uses a ***for*** loop to count the input bits which are ‘1’.

```
module Majority(
input logic [2:0]   Data,
output logic   Y);

parameter      majority = 2;
reg        [2:0]    count;
integer             k;

 always_comb
 begin
   count = 0;
   for (k = 0; k < 3; k = k + 1) begin
     if (Data[k] == 1) count = count + 1;
   end
   Y = (count >= majority);
 end
endmodule
```

This code models a combinational logic circuit with the truth table:

Data[2]	Data[1]	Data[0]	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

*Example Use of a **repeat** loop*

A repeat loop is used in the fragment of code below to initialise a memory array:

```
...
word_address = 0;
repeat (memory_size)
  begin
    memory [word_address] = 0;
    word_address = word_address +1;
  end
...
```

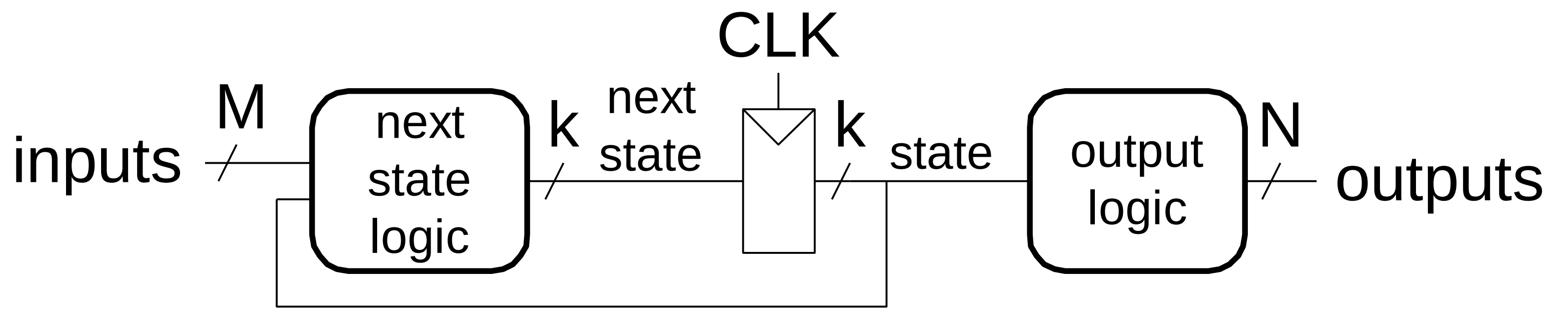
Example

The function *aligned_word* in *word_aligner* shifts (<< is the left-shift operator) an 8-bit word to the left until the most significant bit is a 1.

```
module word_aligner (input logic    [7:0]    word_in,
                     output logic  [7:0]    word_out);

assign word_out = aligned_word(word_in);

function  [7:0]  aligned_word;
  input     [7:0]    word;
  begin
    aligned_word = word;
    if (aligned_word != 0)
      while (aligned_word[7] == 0) aligned_word = aligned_word << 1;
    end
 endfunction

endmodule
```

6.7 Finite State Machines (FSMs)

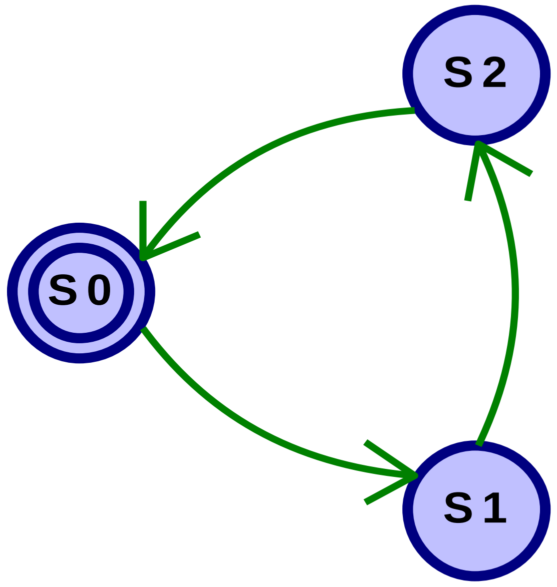
- **Three blocks:**
  - next state logic
  - state register
  - output logic



The state of the system must be held in an internal register. In SystemVerilog, the state can be represented by an enumerated type. The possible values of this type are the state names, and the name of the variable is given after the list of values. For example:

enum {s0, s1, s2, s3} state;

FSM Example1

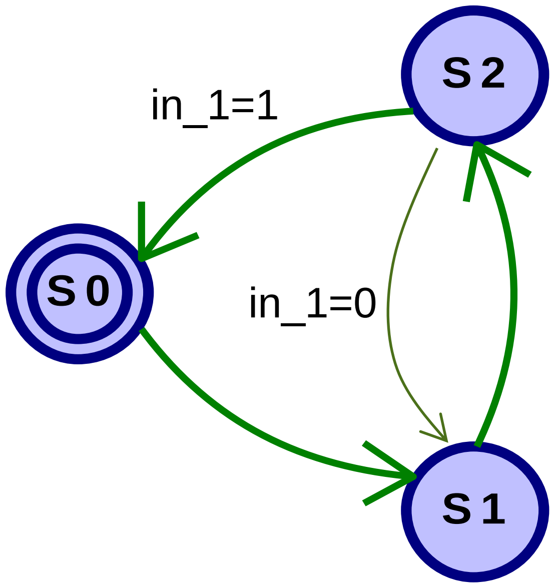


Output q=1 when state is S0, else q=0

The double circle indicates the reset state

```
module FSM_example1 (input  logic clk,
reset,
                                        output logic q);

  enum {s0, s1, s2} present_state,
next_state;

  // state register
  always_ff @ (posedge clk, posedge reset)
    if (reset) present_state <= S0;
    else       present_state <= next_state;

  // next state logic
  always_comb
    case (present_state)
      S0:      next_state = S1;
      S1:      next_state = S2;
      S2:      next_state = S0;
      default: next_state = S0;
    endcase

  // output logic
  assign q = (present_state == S0);
endmodule
```

FSM Example 2

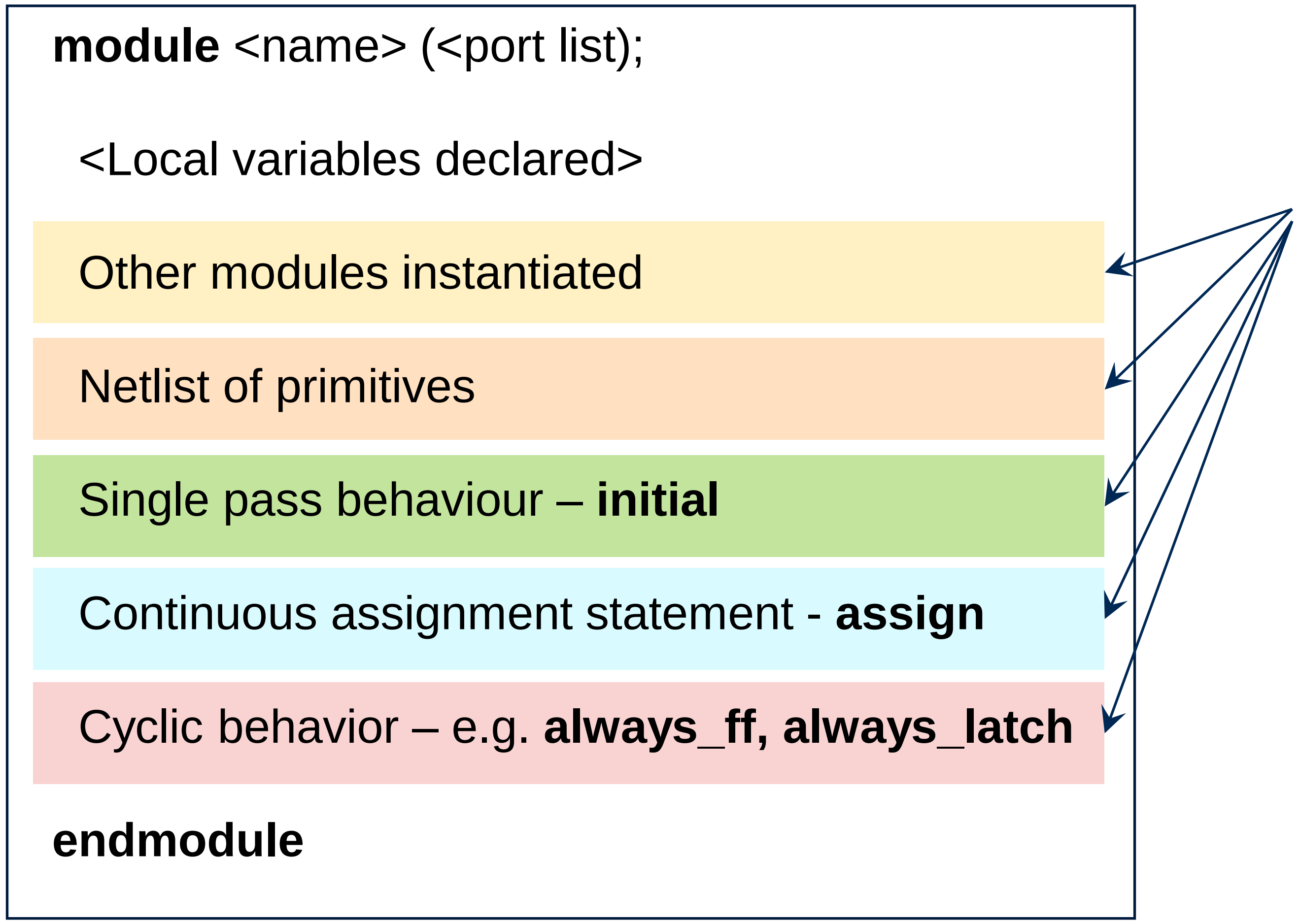


Output q=1 when state is S0, else q=0

The double circle indicates the reset state

```
module FSM_example2 (input  logic clk, reset,
in_1,
                                 output logic q);

   enum {s0, s1, s2} present_state, next_state;

   // state register
   always_ff @ (posedge clk, posedge reset)
      if (reset) present_state <= S0;
      else       present_state <= next_state;

   // next state logic
   always_comb
      case (present_state)
         S0:     next_state = S1;
         S1:     next_state = S2;
         S2:     if (in_1)
                     next_state = S0;
                 else
                     next_state = S1;
         default: next_state = S0;
      endcase

   // output logic
   assign q = (present_state == S0);
endmodule
```

Common mistakes to avoid in SystemVerilog coding



module <name> (<port list);

<Local variables declared>

Other modules instantiated

Netlist of primitives

Single pass behaviour – **initial**

Continuous assignment statement - **assign**

Cyclic behavior – e.g. **always_ff, always_latch**

endmodule

All the blocks are running simultaneously

The order of the blocks is not important

Do not mix up the styles of the blocks! Each block of code has to be one of these types.

For example:

Don’t use continuous assignments (**assign**) inside cyclic behaviours (e.g. **always_ff**)

Don’t instantiate modules or primitives inside continuous assignment statements, single pass behaviours or cyclic behaviours

Common mistakes to avoid in SystemVerilog coding

Examples of incorrect code:

```
always_ff @ (posedge clock)
begin
  assign y = a & b;
end
```

❌ Can’t use a continuous assignment statement inside a cyclic behaviour

```
always_latch
begin
  and (y, a, b);
  module my_circuit (w, c, d);
end
```

❌ Can’t instantiate primitives or other modules inside a cyclic behaviour

```
assign xor (t, e, f);
```

❌ Can’t instantiate primitives or other modules inside a continuous assignment statement