UNIT-V

Sequential Circuit Description

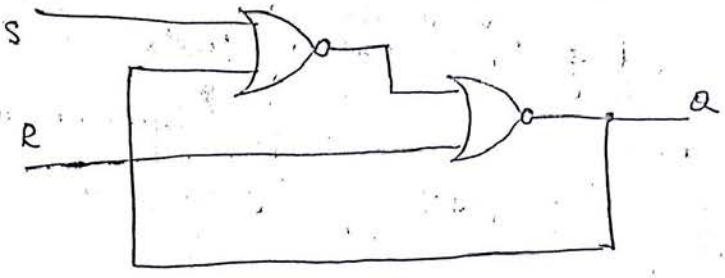
Sequential models:-

In digital circuits, storage of data is done either by feedback, or by gate capacitances that are refreshed frequently.

- Verilog provides language constructs for building memory elements using both these schemes.
- However, more abstract models also exist and are used in most sequential circuit models.

(a) Feedback model:-

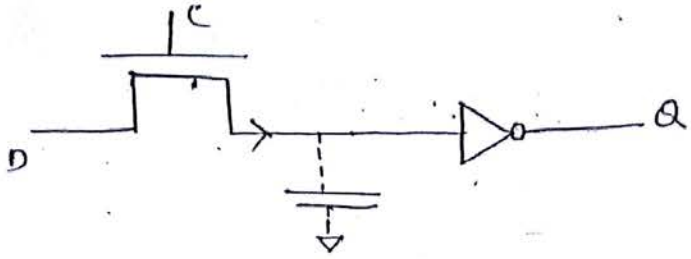
The construct shown in below figure is the most basic feedback circuit that has data storage capability.



This circuit has one feedback line that makes it a two-state (feedback-0 and feedback=1) or a 1-bit, memory element. many verilog constructs can be used for proper modeling of this circuit.

(b) Capacitive model:-

Another hardware structure with storage capability is shown in below figure



When C becomes '1' the value of D is saved in the input gate of the inverter and when C becomes '0', this value will be saved until the next time that C becomes 1 again.

— The output of the inverter is equal to the complement of the stored data.

— Because of powerful switch level capabilities of verilog, the ckt in before figure is very closely modeled in verilog.

(c) Implicit models—

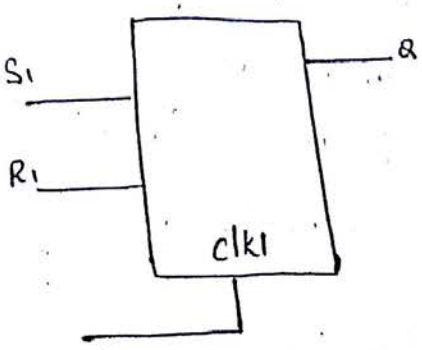
, feedback and capacitive models discussed before are technology dependent, and they have the problem of being too detailed and thus too slow to simulate.

— verilog offers language constructs that model storage elements at more abstract levels than the previous models.

— Such modelings are technology independent and allow much more efficient simulation of circuit with a large no. of storage elements

— The below figure shows SR-latch model without gate level details.

— Because gate and transistor details of models at the block diagram level are not known, verilog provides timing check constructs for ensuring correct operation of this level of modeling.



- The Sections that follow present language constructs for feedback modeling of storage Elements, but concentrate on the more abstract models in which storage is implied by the Verilog code.

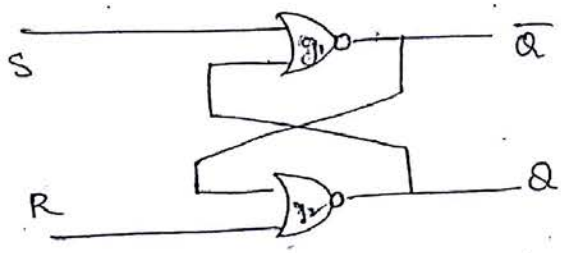
Basic memory components

This Concept discuss about modeling memory components in verilog. first we start with latches and 1-bit & multi dimensional memories.

- These can be discussed by considering the modeling of gates, primitives, assignments and procedural blocks.

1. Gate level primitives :-

The below figure shows a cross-coupled NOR structure that forms a 1-bit storage Element.



- This is Similar to the 1st figure but its storage is due to the feedback from q back to g1.
- The verilog code for the above diagram is shown as

```
module latch (input s, r, output q, q_b);
  nor #(4)
    g1 (q_b, s, q), g2 (q, r, q_b);
  Endmodule
```

- Here q and q-b outputs are driven by two NOR gates, and ∴ initially 'x' until S≠R=0
- after a delay of 4nsec S≠1 ∴ q=1, and after another 4 nsec delay, q-b=0
- Simultaneous assertion of both inputs results in loss of memory
- This memory element is the base of most static memory components.

② User defined Sequential primitives (UDP):-

for faster Simulation of memory elements and for correspondence with specific component libraries, Verilog provides language constructs for defining Sequential user-defined primitives (UDP's)

- A Sequential UDP has the format of the combinational UDP Except that in addition to the inputs, and outputs of the circuit its present state is also Specified.
- The below program shows a Sequential UDP to define SR clocked latch.

```
primitive latch (q, s, r, c);
    output q;
    reg q;
    input s, r, c;
  initial  q = 1'b0;
    table
        // s  r  c  :  q+ :
        //  _  :  _  :  _ ;
        ?  ?  0  :  ?  :  _ ;
        0  0  1  :  ?  :  _ ;
        0  1  1  :  ?  :  0 ;
        1  0  1  :  ?  :  1 ;
    endtable
```

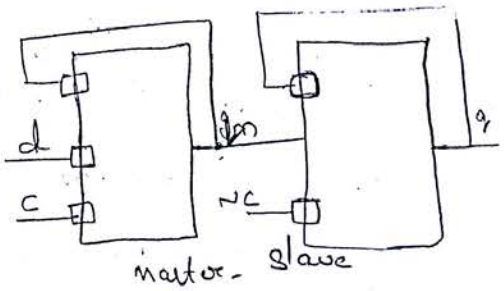
- Here the '?' in the table signifies "any value" and '-' signifies "no change".
- When instantiated, Rise and fall delay values can be specified for a sequential UDP in the same way as specifying delays for other language primitives.

3. Memory Elements using Assignments:-

A continuous assignment is equivalent to a gate structure driving the left-hand side of the assignment.

- We can use these statements for specifying specific gates of a hardware module for a latch or for specifying a feedback ckt.
- The below program shows a master-slave FF that uses assign statement to implement feedback.

```
module ms_si #(parameter delay=3) (input d, c, output q);
  wire qm;
  assign #(delay) qm = c ? d : qm;
  assign #(delay) q = ~c ? qm : q;
endmodule.
```



- Here each assign statement implements a latch, and the module that uses two latches with complementary clocks implements a master-slave flip-flops.

4) Behavioural memory Elements:-

The previous topics showed verilog modules for latches and flipflops by explicit use of feedback or present and next states.

- Such a model corresponds to the actual H/w implementation of memory element.
- The more abstract and easier way of writing verilog code for a latch or flf is by behavioral coding.

ii) Latch modeling:-

for this let us consider verilog code for D latch

```
module dlatch (input d,c output reg q, q_b);
always @ (c or d).
  if (c) begin
  q <= #4 d;
  q_b <= #3 ~d;
  End
Endmodule
```

- from this code when c or d changes, if c=1 q=d. This means that if c or d changes and c is not 1, then q dounot change and it remains its old value.
- This behaviour i.e, while c=1, changes on 'd' directly affect q and q_b. This shows 'c' is level sensitive and ∴ a latch

But here some problems arrises
∴ use nonblocking assignments to describe sequential circuits.

iii) flip-flop modeling:-

consider edge-triggered f/f model as

```
module d_ff (input d, clk, output reg q, q_b);
always @ (posedge clk) begin
  q <= #4 d;
  q_b <= #3 ~d;
End module
```

- This model is sensitive to the positive edge of clock, and uses nonblocking assignments for assignments to q and q_b.
- flow in the procedural block is controlled by the Event control statement that has posedge clk as its Event Expression.
- Assignments to the q and q_b are reached immediately after the flow in the always block begins.
- ∴ with Each clock Edge, the Entire procedural block is Executed once from begin to End.

(iii) ff with Set_reset Control:-

The D-ff above can be Expanded to ff's with syn and asyn Set & reset control inputs

- The D-ff with syn Set & reset i/p's (s & r) are shown below

```
module d_ff_sr (input d, s, r, clk, output reg q, q_b);
  always @ (posedge clk) begin
    if (s) begin
      q <= #4 1'b1;
      q_b <= #3 1'b0;
    End else if (r) begin
      q <= #4 1'b0;
      q_b <= #3 1'b1;
    End else begin
      q <= #4 d;
      q_b <= #3 ~d;
    End
  End
  Endmodule
```

Note:-

In Syn set & reset, q waits for the edge of the clock to set or reset while in

Asyn set & reset, change in q occur independent of clk when s or r becomes active.

(iv) Other storage element modeling styles:-

Verilog provides other language constructs for latch & flfs for the sake of completeness and presentation.

— The below shows D latch using wait statement instead of event control.

```
module latch (input d,c, output reg q, q_b);
  always begin
   wait (c)
  #4  q<=d;
  #3  q_b <= ~d;
  End
  Endmodule
```

— Here the wait statement shown is a procedural statement that blocks the flow of the procedural block when c=0

— when c=1, the wait statement allows the program flow to pass it and reach assignments to q & q_b.

— The below program shows D ff with fork join

```
module d_ff (input d, clk, output reg q, q_b);
  always @ (posedge clk)
  fork
    #4  q<= d;
    #3  q_b <= ~d;
  join
```

- The fork-join bracketing instead of begin-end causes all Seq. statements or blocks of Seq statements. that are immediately with in this bracketing to be Executed in parallel.

(v) F/F timing:-

Behavioural modelling of FF's allows only a limited timing specifications.

∴ Some constructs include System tasks for checking setup, hold, period and with parameters.

(a) Set-up time:- Set up time is defined as the minimum necessary time that a data input requires to setup before it is clocked into a FF.

- verilog construct for checking the setup time is $ setup, which takes the FF data input, activate clkedge and the setup time as its parameter.

- The $ Setup task is used with in specify block

Eg: specify
$ setup (d, posedge clk, 5);
End specify

- This continiously checks the timing distance between changes on d and the posedge of clk clock.

(b) Hold-time :- Hold time is defined as the minimum necessary time a F/F. data input must stay stable (hold its value) after it is clocked.

- verilog construct for checking hold time is $ hold which is also used in specify block

Eg: specify [illegible]

End specify

(e) width and period:- vsilog $ which width & $ period checks for minimum pulse width and period.

(vi) memory vectors and arrays:-

Coding for ff's and latches can be applied to arrays and vectors as well.

— The only difference is when one dimension vectors or multidimensional arrays are being considered, their input, output ports and memory structures should be declared accordingly.

(a) Vectors:- The below program shows 8-bit transperent D-latch. The data input and latch o/p are declared as 8-bit vectors. The always block is sensitive to c and all 8-bits of d.

```
module vector_dlatch (input [7:0] d, inputc, output reg [7:0] q);
    always @ (c or d)
      if (c)
      #4 q=d;
    Endmodule
```

(b) Array:- Here there will be address and data for read: address location drives data

for write: memory holds the data

(c) memory initialization:- Verilog has the $readmemh and $readmemb tasks for reading External data files and using them for initialization of memory blocks.

(d) Bidirectional memory:- Another features of memory is its bidirectionality. Here data is declared as inout.

— An inout bus is only considered as a net, c[illegible] must be declared as a reg

Functional Registers:-

A Register is defined as group of ff's / latches.

- A functional Register is also a group of ff's with a common clock and with some functionality such as counting and shifting.
- Verilog code for functional register are like to that of ff's but addition of arithematic & logical functionality to the code.

1) Shift Registers:-

The addition of shifting operation to the registers are discussed here

(a) Basic shifter:-

fig shows a basic 4-bit shift register with load, reset and shift capabilities.

- The l-r controls left/right shift
- The vacated bit in filled with S_in
- The verilog code for this is
- ld performs parallel load of d into q

```
module shift_reg (input [3:0]d, input clk,
                  ld, rst, l_r, S_in,
                  output reg [3:0] q);
always @ (posedge clk) begin
  if (rst)
    #5 q<= 4'b0000;
  else if (ld)
    # 5 q<=d;
  else if (l_r)
```

```
#5 q <= {a[2:0], s_in};
  else
  #5 q <= {s_in, q[3:1]};
  End
  endmodule
```

— All operations of this circuit are done in an always block which is sensitive to positive edge of clk.

(b) universal shift register:-

— It have bidirectional io

— The circuit has a s_1, s_0 inputs forming a 2-bit number ranging from 3 to 0.

— The shifter performs

- nothing
- shift right
- shift left
- performs parallel load

based on $\{s_1, s_0\}$ values.

— Because this ckt has a bi-directional 'inout' port, we have declared q_int to hold shift reg o/p.

```
module shift_reg (input clk, rst, s_in, l_in, en, s1, s0,
                  inout [7:0] io);

  reg [7:0] q_int;
  assign io = (en)? q_int : 8'bz;
  always @ (posedge clk) begin
    if (rst)
  #5 q_int = 8'b0;
  else
  case ({s1, s0})
  2'b01: // shift right
```

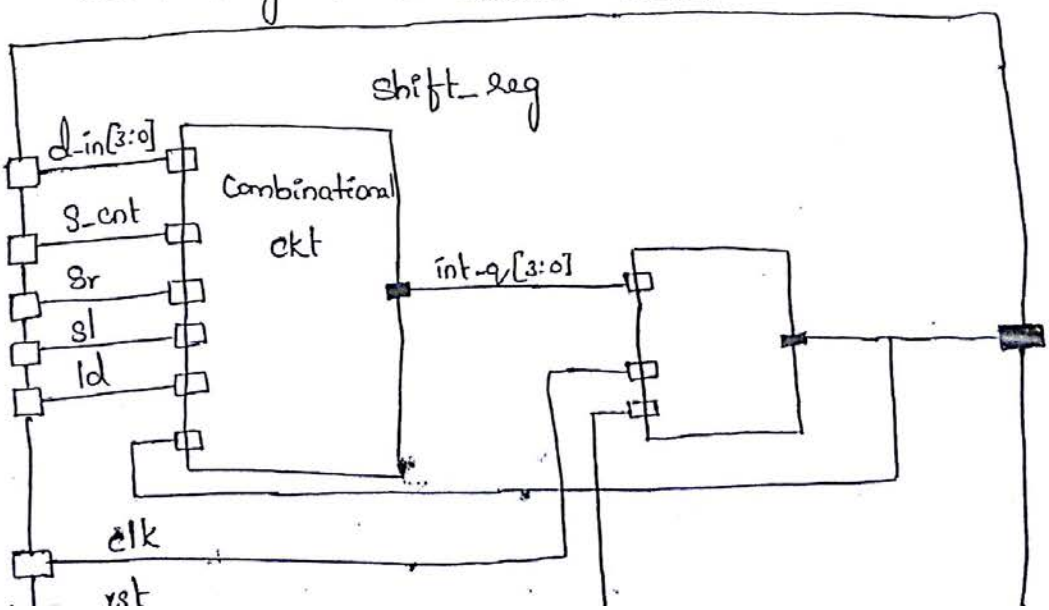
```
q = int <= {r_in, q_int [7:1]};
2'b10 : // shift left
   q_int <= {q_int [6:0], l_in};
2'b11 : // parallel load
      q_int = io;
default : // Do nothing
    q_int <= q_int;
  Endcase
 End
  Endmodule
```

© Seperate register and combinational block:-

For describing sequential ckts with complex functionalities we seperate combinational and sequential blocks.

— we use this style for " shift registers ckts that can shift its contents a specified no. of positions to the right or left"

— The block diagram is shown below



- Left/right shift is ... of ...
- The no. of shift is determined by S_cnt.
- The shifter use 0's to fill vacant positions.
- The ld loads d_in into the shift register
- The verilog code is represented as

```
module shift_reg (input [3:0]d_in, input, clk, sr, sl, ld, rst,
                  input [1:0] s_cnt, output reg [3:0] q);
    reg [3:0] int_q;
always @ (d_in, q, s_cnt, sr, sl, ld)
    begin: combinational
  if (ld)
   int_q = d_in;
  else if (sr)
   int_q = q>>s_cnt;
   else if (sl)
   int_q = q<< s_cnt;
  else int_q = q;
  end
always @ (posedge clk)
    begin: register
    if (rst), q<=0,
    else
      q<= int_q;
    end
     endmodule
```

(2) counters:-

Counter coding is similar to shift register except that arithematic add (+) and substract (-) operations must be used for count_up & count_down

(a) up_counter:-

The below code shows 4-bit upcounter ckt

```
module counter (input [3:0] d_in , input, clk, rst, ld, ud,
                output reg [3:0] q);
always @ (posedge clk)
  begin
    if (rst)
      q = 4'b0000;
  elsif (ld)
      q = d_in;
    else if (!u_d)
      q=q+1;
    else  q= q-1;
    End
  endmodule
```

- If u_d =1 the counter counts up-words and if u_d=0 the counter counts downwards.
- when counting if q reaches 1111, then adding a1 and capturing the MSB 4 bits causes the count sequence to roll back to 0000 and continue the count from there

(b) Gray_code counter :-

A Gray_code counter cannot use arithematic operators

∴ Gray code counter is developed by using look_up table using a external memory file.

- The next count is lookup from a memory of sixteen 4-bit entries
- Each memory location contains the address of the location treated as a gray code no +1.

Eg:- location 7(0111) contains 0101 (∵ 0111 i... ...)

and its next count up is 0101 that is gra...

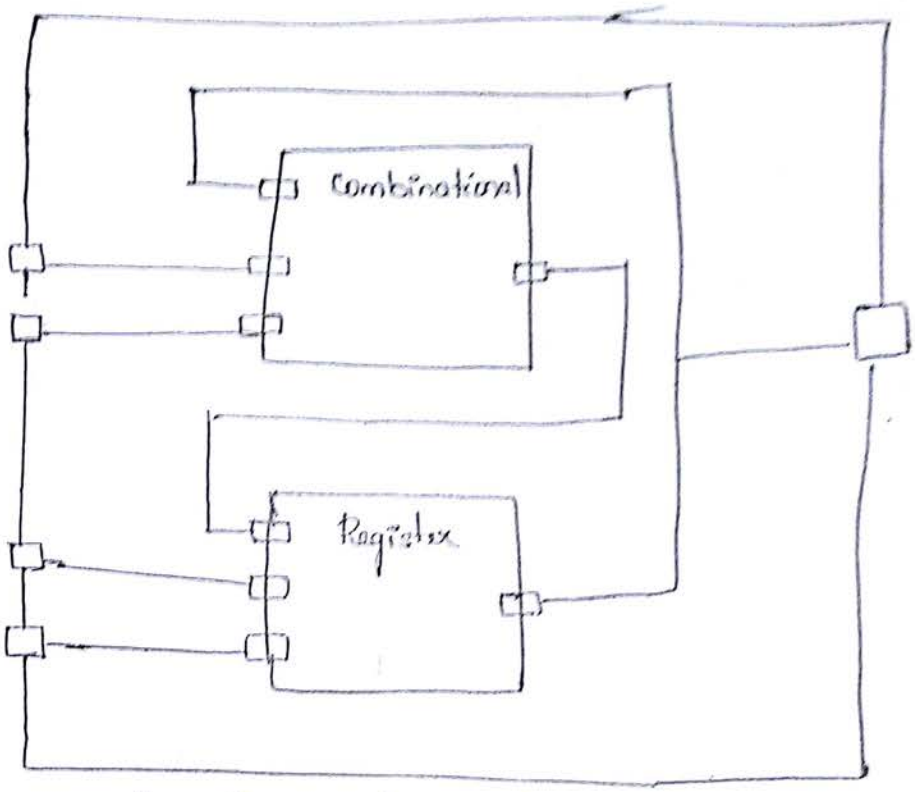


fig: Gray - code counter diagram

— The Verilog code for gray-code counter is

```
module gray_counter (input [3:0] d_in, input clk, rst, ld,
                     output reg [3:0] q);
  reg [3:0] mem [0:15];
  reg [3:0] im_q;
initial
  $ readmemb ("mem.dat", mem);
always @(d_in or ld or q)
    begin: combinational
  if (ld)
    im_q = d_in;
    else
   im_q = mem[q];
  end
always @ (posedge clk)
    begin: register
  if (rst)
```

```
    q <= 4'b0000;
  Else
    q <= in-q;
  End
Endmodule
```

③ LFSR and MISR:-

A linear feedback shift Register (LFSR) is used for pseudo random number generation.

— A LFSR is a shift register with feedback and XOR gates in its feedback or shift path.

— The initial content of the register is referred to as "Seed", and the position of XOR gates is determined by the polynomial (poly) of the LFSR.

— A multiple input Signature Register (MISR) is like an LSR, but with parallel input and output.

— A MISR is used for Signature generation of multi-bit input vectors.

ⓐ LFSR:-

— An LFSR with 10101 polynomial is shown as

—The figure shows LFSR made of DFF's and XOR gates in its shift path.

— The position of XOR gates determine the poly of the circuit which is 10101.

—The Seed, which is the initial value of register, attach set and reset inputs of individual ff's of the shift reg.

—The LFSR seed and poly determine bit values that are generated on the Serial output of the ckt (Sout) as Serial input bits (Sin) are being shifted in the program is written as follows

```
module dff (input clk, Set, rst, d, output reg q);
always@(posedge clk or posedge Set or posedge rst)
    if (set)
      q<= 1'b1;
    else if (rst)
        q<= 1'b0;
    Else
       q<= d;
    Endmodule

module structural_lfsr # (parameter [3:0] seed = 4'b0)
                (input clk, init, Sin, output Sout);

wire im1, im2, Im3, im4, im5;
dff #[3:0] (clk, {4{int}} & seed, {4{int}} & seed;
          {im1, im2, im3, im5} ; {im2, im3, im5, Sout});
xor (im1, Sin, Sout);
xor (im4, im5, Sout);
Endmodule
```

(b) MISR:-

— MISR is used for Signature generation and data Compression

— over a period of several clks, parallel data into a MISR are compressed with the existing MISR data.

— The final data depends on the MISR initial data (seed) and its XOR and feedback structure (poly)

— The below shows MISR program

```
module #(parameter [3:0] poly=0) misr (input clk, rst,
        input [3:0] d_in, output reg [3:0] d_out);

always @ (posedge clk)
    if (rst)
      d_out = 4'b0000;
    else
      d_out = d_in

End module
```

(4) stacks and Queues:-

Register modeling and memory read write operations can be combined for describing stacks and Queues

— A Queue has a memory block and read & write pointers.

— Read and write pointers are described as registers and counters that provide address pointers for the Queue memory.

— Combinational logic blocks are used for providing full and Empty indicators for the Queue memory.

② FIFO:-

A first in first out (FIFO) Queue is a Queue of data such that the data that is written into it first, is read from it first.

- The below figure shows the block diagram of FIFO.

- The 'pointer' block provides read & write pointers according to the operations, into the Queue.

- The 'count' block keeps track of the no. of data in the Queue and diccuss issues Empty, full flage. These flags are generated by combinational blocks using the present count of data as input.

- The 'Read' block uses the read pointer to read from the Queue memory.
- The 'write' block uses the write pointer to write pointer to write into the Queue memory.
- The Read write blocks share the 'fifo_ram' memory.
- The verilog code for fifo is written as follows

```
module fifo (input [7:0] data_in,
             input clk, rst, rd, wr,
             output Empty, full,
             output reg [3:0] fifo_cnt,
             output reg [7:0] data_out);

reg [7:0] fifo_ram [0:7];
reg [2:0] rd_ptr, wr, ptr;
assign Empty = (fifo_cnt == 0);
assign full = (fifo_cnt == 8);
always @ (posedge clk)
  begin: write
  if (wr && !full)
  fifo_ram [wr_ptr] <= data_in;
  else if (wr && rd)
  fifo_ram [wr_ptr] <= data_in;
  end
always @ (posedge clk)
  begin: read
if (rd && ! Empty)
  data_out <= fifo_ram [rd_ptr];
```

```
else if (rd && wr && empty)
  data_out <= fifo_ram [rd_ptr];
  else if (rd && wr && empty)
    data_out <= fifo_ram [rd_ptr];
    end
    always @ (posedge clk)
      begin : pointer
      if (rst) begin
            wr_ptr <= 0;
            rd_ptr <= 0;
    end
    else  begin
        wr_ptr <= ((wr && !full) || (wr && rd)) ? wr_ptr+1 :
                                                  wr_ptr;

  rd_ptr <= ((rd && !empty) || (wr && rd)) ? rd_ptr+1 :
                                             rd_ptr;

    end
    end
always @ (posedge clk)
      begin : count
      if (rst) fifo_cnt <= 0;
      else  begin
        case ({wr, rd})
      2'b00 : fifo_cnt <= fifo_cnt;
      2'b01 : fifo_cnt <= (fifo_cnt == 0) ? 0 : fifo_cnt-1;
      2'b00 : fifo_cnt <= (fifo_cnt == 8) ? 8 : fifo_cnt+1;
      2'b11 : fifo_cnt <= fifo_cnt;
        default : fifo_cnt <= fifo_cnt;
            endcase
```

End
End
Endmodule

In the above program 4 always blocks that are sensitive to the positive edge of the clock handle.

writing (write), reading (read), updating read & write pointers (pointer), and keeping the FIFO count (count) exists.

— Concurent with these block two assign statements issue Empty & full.

— The write block writes into fifo_ram if it is not full. If both rd and wr inputs are active, full is not checked and memory is written into.

If none of these conditions hold, the memory is intact.

— The read block reads from fifo_ram if it is not Empty. If it is Empty and boths rd & wr inputs are active, data from the present pointer location is read into data-out.

— The pointer block implements two counters for read and write pointers. Seperate from these pointers, the count always block performs incrementing and decrementing fifo_cnt depending on read & write operations being done.

— The Countblock uses a case statement with default, which handles ambiguous values on wr and rd. In this case, fifo_cnt is left intact.

State machine Coding:-

coding styles presented so far can further be generalized to cover finite state machines of any type

- This Session shows coding for moore and mealy s.m's.
- Here we use sequence detector as an Example.

(a) Moore machines:-

A moore machine is a state machine in which all outputs are fully synchronized with the circuit clock. In the state diagram form, Each state of the machine specifies its outputs independent of the circuit inputs.

- In verilog code of moore state machine, only circuit state variables participate in the output Expression of the ckt.
- fig shows 101 moore sequence detector with its corresponding block diagram & related verilog code.

— Verilog code

```
module moore_det (input x, rst, clk, output z);
  localparam [1:0] // To assign values to states of machine
   reset=0, got1=1, got10=2, got101=3;
   reg [1:0] current; // Holds current state of machine.
   always @ (posedge clk) begin
      if (rst)
         current <= reset;
      else
      case (current)
         reset : begin
         if (x == 1'b1)
            current <= got1;
         else
            current <= reset;
         End
      got1 : begin
            if (x == 1'b0)
               current <= got10;
            else
               current <= got1;
            End
      got10: begin
            if (x == 1'b1)
               current <= got101;
            else
               current <= reset;
            End
      got101 : begin
            if (x == 1'b1)
               current <= got1;
```

```
        else
          current <= got10;
        End
    default: begin
               current <= Reset;
             End
    Endcase
    End
    assign z = (current == got101) ? 1 : 0;
    Endmodule
```

- In the before program, we used localparam declaration to assign values to the states of machine.
- Because our state diagram has 4 states, 2-bit parameters are used for the state names.
- we also used current as a 2-bit reg which is used to hold the current state of machine.

(b) mealy machines:-

A mealy machine is differ from a moore machine in that its outputs depends on its current state as well as inputs while in that state.

- fig shows 101 mealy machine sequence detector and its corresponding block diagram and verilog code.

Verilog code:-

```
module mealy_dector2 (input x, rst, clk, output z);
  localparam [1:0]
   reset=0, got1=1, got10=2;
  reg [1:0] current;
always @ (posedge clk) begin
    if (rst)
     current <= reset;
  Else case (current)
     reset :
        if (x==1'b1)
           current <= got1;
       Else
          current <= reset;

got1;  if (x=1'b0)
           current <= got10;
       Else current <= got1;

got10:
   if (x=1'b1)
          current <= got1;
   else
      current <= reset;
   default :
```

```
current <= Reset;
Endcase
End
assign z = (current == got10 && x==1'b1) ? 1'b1: 1'b0;
Endmodule
```

- Here a 2-bit localparam construct is used for defining the states of this machine.
- Because the machine has three states, and two state variables are used to represent them, one combination (i.e, 11) of the state variables becomes unused. The default in the case statement handles this.

(c) Huffman coding style:-

Huffman coding style of digital system characteristics fit as a combinational block with feedbacks through an array of registers.