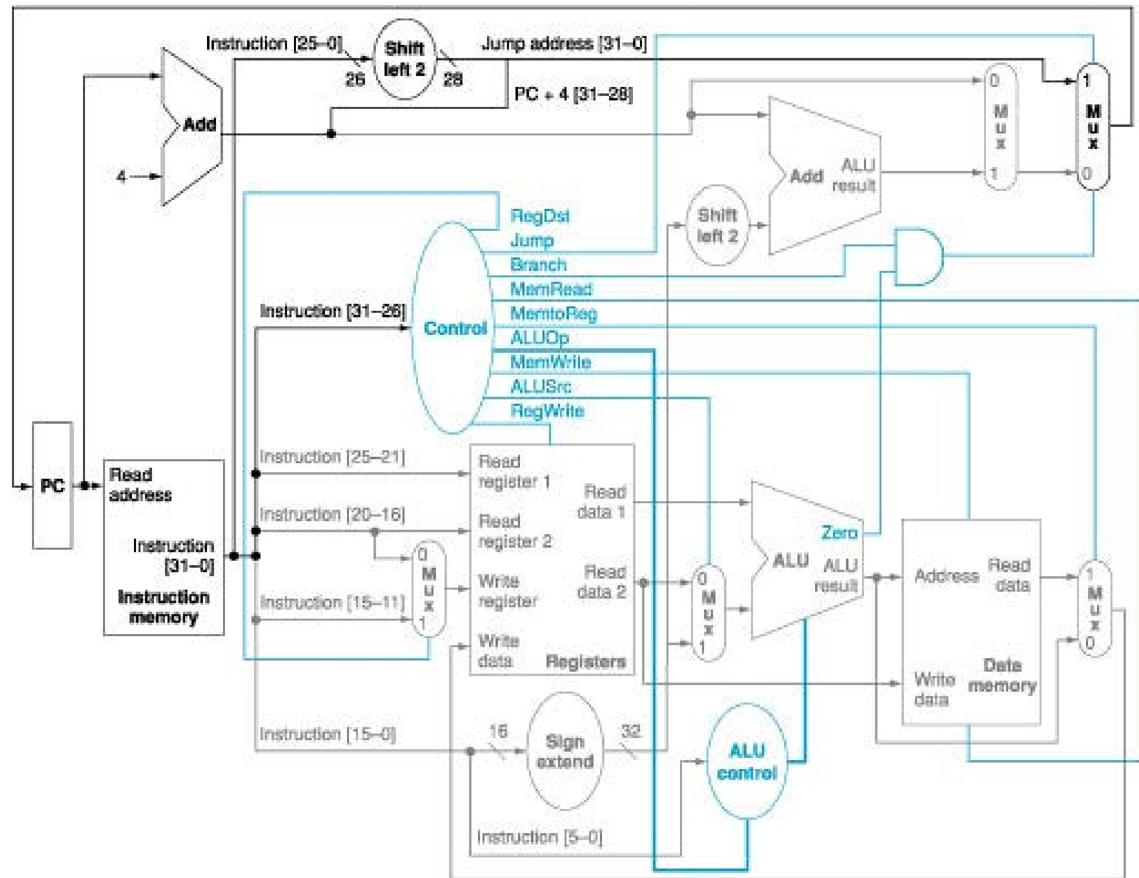
# Lecture 5 - Modeling for Synthesis

## Register Transfer Level (RTL) Design

# Register Transfer Language (RTL) Design

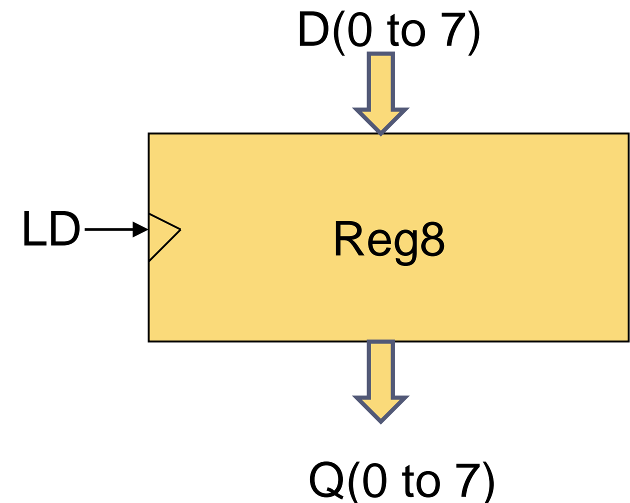
- A **system** is viewed as a structure comprising registers, functions and their control signals
- Show dataflow through the system
- Instructions, Data, Addresses
- Functions store and manipulate data




No gates!!!

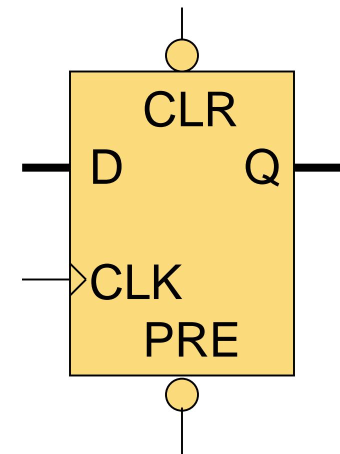
# RTL register model

```
-- Model register to hold one datum of some type
-- Individual bits are not manipulated
library ieee;  use ieee.std_logic_1164.all;

entity Reg8 is
  port (D: in std_logic_vector(7 downto 0);
        Q: out std_logic_vector(7 downto 0);
        LD: in std_logic);
end Reg8;

architecture behave of Reg8 is
begin
   process(LD)
   begin
        if (LD’event and LD=‘1’) then
                Q <= D;   -- load data into the register
        end if;
   end process;
end;
```



# Asynchronous control inputs

```
library ieee;  use ieee.std_logic_1164.all;

entity Reg8 is
  port (D: in std_logic_vector(7 downto 0);
          CLK,PRE,CLR: in bit;                    --Async PRE/CLR
          Q: out std_logic_vector(7 downto 0));
end Reg8;

architecture behave of Reg8 is
begin
   process(clk,PRE,CLR)
   begin
         if (CLR=‘0’) then                  -- async CLR has precedence
            Q <=“00000000”;                 -- force register to all 0s
         elsif (PRE=‘0’) then               -- async PRE has precedence if CLR=‘0’
            Q <= (others => ‘1’);           -- force register to all 1s
         elsif rising_edge(clk) then        -- sync operation only if CLR=PRE=‘1’
            Q <= D;                         -- load D on clock transition
         end if;
   end process;
end;
```



# Synchronous reset/set

```
--Reset function triggered by clock edge
process (clk)
begin
    if (clk'event and clk = '1') then
        if reset = '1' then – reset has precedence over load
            Q <= "00000000" ;
        else
            Q <= D ;
        end if;
    end if;
end process;
```

# Register with clock enable

```
-- “enable” effectively enables/disables clock
process (clk)
begin
     if rising_edge(clk) then  -- detect clock transition
          if enable = ‘1’ then    -- enable load on clock transition
                  Q <= D ;
          end if;
     end if;
end process;
```

# Register with parameterized width

```
-- One model of a given function with variable data size
library ieee;  use ieee.std_logic_1164.all;

entity REGN is
   generic (N: integer := 8);                          -- N specified when REG used
   port (    CLK, RST, PRE, CEN: in std_logic;
             DATAIN: in   std_logic_vector (N-1 downto 0);  -- N-bit data in
             DOUT:    out std_logic_vector (N-1 downto 0)   -- N-bit data out
             );
end entity REGN;

architecture RTL of REGN is
begin
process (CLK) begin
   if (CLK'event and CLK = '1') then
        if      (RST = '1') then DOUT <= (others => '0');  --reset to all 0s
        elsif (PRE = '1') then DOUT <= (others => '1');  --preset to all 1s
        elsif (CEN = '1') then DOUT <= DATAIN;           --load data
        end if;
   end if;
end process;
end architecture RTL;
```

Vectors: “100” = (‘1’,’0’,’0’) = (‘1’, others => ‘0’)

Arbitrarily long: “00…0” = (others => ‘0’)

# Instantiating the parameterized register

```
library ieee;  use ieee.std_logic_1164.all;
entity TOP is
   port (       CLK,X,Y,A,B,C: in std_logic;
                 DIN: in  std_logic_vector(5 downto 0);
                 Q1: out std_logic_vector(5 downto 0);
                 Q2: out std_logic_vector(4 downto 0);
                 Q3: out std_logic_vector(3 downto 0)
                 );
end entity TOP;

architecture HIER of TOP is
component REGN is
  generic (N: integer := 8);
  port (        CLK, RST, PRE, CEN: in std_logic;
                 DATAIN: in   std_logic_vector (N-1 downto 0);
                 DOUT:    out std_logic_vector (N-1 downto 0)
                 );
end component REGN;
begin
R1: REGN generic map (6)  port map    --6-bit register
                (CLK, A, B, C, DIN, Q1);
R2: REGN generic map (5)  port map    --5-bit register (low 5 bits of DIN)
                (CLK, Y, X, C, DIN(4 downto 0),Q2);
R3: REGN generic map (4)  port map    --4-bit register (low 4 bits of DIN)
          (CLK=>CLK, RST=>A, PRE=>B, CEN=>C, DATAIN=>DIN(3 downto 0), DOUT=>Q3);
end architecture HIER;
```

# 2-to-1 mux with parameterized data size

```
entity muxN is
    generic (N: integer := 32);  -- data size parameter
    port (  A,B:  in   std_logic_vector(N-1 downto 0);
            Y:      out std_logic_vector(N-1 downto 0);
            Sel:   in   std_logic);
end muxN;
architecture rtl of muxN is
begin
    Y <= A when Sel = '0' else B;     -- A,B,Y same type
end;
-- specify parameter N at instantiation time
M:  muxN generic map (16)
            port map(A=>In1, B=>In2, Y=>Out1);
```

# Other types of generic parameters

```
entity and02 is
    generic (Tp : time :=  5 ns);     -- gate delay
parameter
    port (A,B: in std_logic;
              Y:     out std_logic);
end and02;
architecture eqn of and02 is
begin
     Y <= A and B after Tp;    -- gate with delay Tp
end;
…..
A_tech1:  and02 generic map (2 ns) port map (M,N,P);
A_tech2:  and02 generic map (1 ns) port map (H,K,L);
```

Gates with different delays.

# IEEE Std. 1076.3 Synthesis Libraries

- Supports arithmetic models
  - ieee.numeric_std (ieee library package)
    - defines UNSIGNED and SIGNED types as arrays of std_logic

      type SIGNED is array(NATURAL range <>) of STD_LOGIC;
      type UNSIGNED is array(NATURAL range <>) of STD_LOGIC;
    - defines arithmetic/relational operators on these types
  - Supports RTL models of functions

- Lesser-used packages:
  - ieee.numeric_bit
    - same as above except SIGNED/UNSIGNED are arrays of type bit
  - ieee.std_logic_arith *(from Synopsis)*
    - Non-standard predecessor of numeric_std/numeric_bit

# NUMERIC_STD package contents

- Arithmetic functions: + - * / rem mod
  - Combinations of operand types for which operators are defined:
    - SIGNED + SIGNED return SIGNED
    - SIGNED + INTEGER return SIGNED
    - INTEGER + SIGNED return SIGNED
    - SIGNED + STD_LOGIC return SIGNED
  - PLUS: above combinations with UNSIGNED and NATURAL

- Other operators for SIGNED/UNSIGNED types:
  - Relational: = /= < > <= >=
  - Shift/rotate: sll, srl, sla, sra, rol, ror
  - Maximum(a,b), Minimum(a,b)

- Convert between types:
  - TO_INTEGER(SIGNED), TO_INTEGER(UNSIGNED)
  - TO_SIGNED(INTEGER,#bits), TO_UNSIGNED(NATURAL,#bits)
  - RESIZE(SIGNED or UNSIGNED,#bits) – changes # bits in the vector

# Arithmetic with NUMERIC_STD package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity Adder4 is
   port ( in1, in2  : in   UNSIGNED(3 downto 0) ;
              mySum : out UNSIGNED(3 downto 0) ) ;
end Adder4;

architecture Behave_B of Adder4 is
begin
    mySum <= in1 + in2; -- overloaded '+‘ operator
end Behave_B;
```

UNSIGNED = UNSIGNED + UNSIGNED

# Conversion of “closely-related” types

- STD_LOGIC_VECTOR, SIGNED, UNSIGNED:
  - All arrays of STD_LOGIC elements
  - Example: How would one interpret “1001” ?
    - STD_LOGIC_VECTOR: simple pattern of four bits
    - SIGNED: 4-bit representation of number -7 (2’s complement #)
    - UNSIGNED: 4-bit representation of number 9 (unsigned #)
- Vectors of same element types can be “converted” (re-typed/re-cast) from one type to another

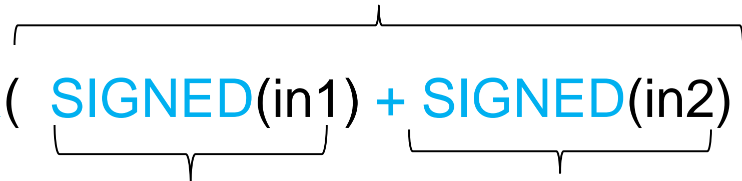
```
signal A:  std_logic_vector(3 downto 0) := "1001";
signal B:  signed(3 downto 0);
signal C:  unsigned(3 downto 0);
B <= signed(A);            -- interpret A value "1001" as number -7
C <= unsigned(A);          -- interpret A value "1001" as number 9
A <= std_logic_vector(B);  -- interpret B as bit pattern "1001"
```

# Conversion of “closely-related” types

For arrays of same dimension, *having elements of same type*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity Adder4 is
   port ( in1, in2  : in   STD_LOGIC_VECTOR(3 downto 0) ;
              mySum : out STD_LOGIC_VECTOR(3 downto 0) ) ;
end Adder4;

architecture Behave_B of Adder4 is
begin
    mySum <=
        STD_LOGIC_VECTOR(  SIGNED(in1) + SIGNED(in2)  );
end Behave_B;
```

SIGNED result



Interpret STD_LOGIC_VECTOR as SIGNED

Function:  SIGNED = SIGNED + SIGNED

Interpret SIGNED result as STD_LOGIC_VECTOR.

# Example – binary counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
ENTITY counter IS
   port( Q:  out std_logic_vector(3 downto 0);
          ….
END counter;

ARCHITECTURE behavior OF counter IS
   signal Qinternal:  unsigned(3 downto 0);
begin

    Qinternal <= Qinternal + 1;      -- UNSIGNED = UNSIGNED + NATURAL
   Q <= std_logic_vector(Qinternal); -- re-type unsigned as std_logic_vector
```

From NUMERIC_STD package

# Using a "variable" to describe sequential behavior within a process

```
-- Assume Din and Dout are std_logic_vector
-- and numeric_std package is included
cnt: process(clk)
        variable count: integer;  -- internal counter state
      begin                        -- valid only within a process
        if clk='1' and clk'event then
          if ld='1' then
                count := to_integer(unsigned(Din));  --update immediately
          elsif cnt='1' then
                count := count + 1;                   --update immediately
          end if;
        end if;
        Dout <= std_logic_vector(to_unsigned(count,32)); --schedule Dout
    end process;
```

# Counting to some max_value (not $2^n$)

-- full-sized comparator circuit generated to check count = max

```
process begin
    wait until clk'event and clk='1' ;
    if (count = max_value) then
        count <= 0 ;             --roll over from max_value to 0
    else
        count <= count + 1 ;     --otherwise increment
    end if ;
end process ;
```

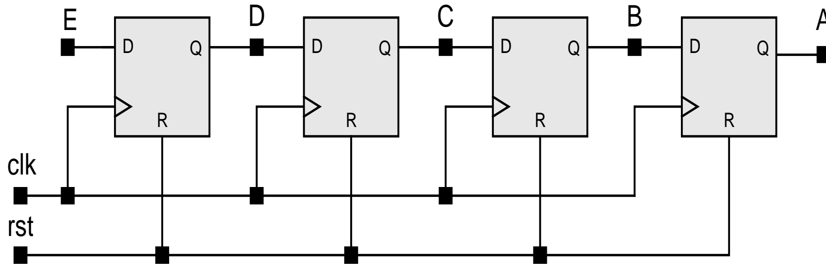
# Decrementer and comparator

```
process begin
    wait until clk'event and clk='1' ;
    if (count = 0) then
        count <= max_value ;  -- roll over from 0 to max_value
    else
        count <= count - 1 ;    -- otherwise decrement
    end if ;
end process ;
```

# Verilog Modeling Trap

- The order of execution of procedural statements in a cyclic behavior may depend on the order in which the statements are listed
- Procedural assignments are called "blocked" assignments (or blocking assignments)
  - Execute sequentially
  - A procedural assignment must complete execution before the next statement can be executed
  - i.e. the statements that follow a procedural statement are **"blocked"** till the current one completes execution
- Expression substitution is recognized by synthesis tools

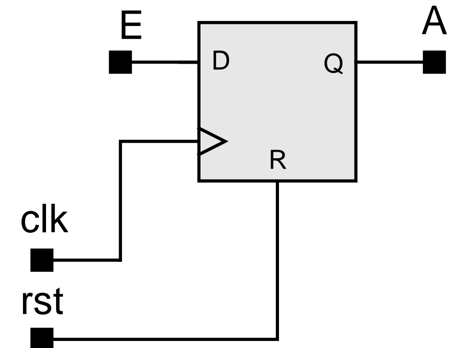
# Example:Modeling Trap of a Shift Register




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

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

```
module shiftreg_PA_rev (E, A, B, C, D,
clk, rst);
  output  A;
  input   E;
  input   clk, rst;
  reg     A, B, C, D;

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

A=E



# Nonblocking Assignment (<=) in Cyclic Behavior

- Effectively execute concurrently rather than sequentially by blocked assignments
  - Independent of the order where they are listed
- Simulator must
  - Sample all variables referenced by RHS with nonblocking assignments
  - Held them in memory
  - Use them to update LHS variables **concurrently**
    - Before the assignments are evaluated
  - Nonblocking makes NO dependency between statements
- Avoid having multiple behaviors assigning values to be the same variable
  - Otherwise, software race condition makes outcome indeterminate
  - For example, multi-driver case

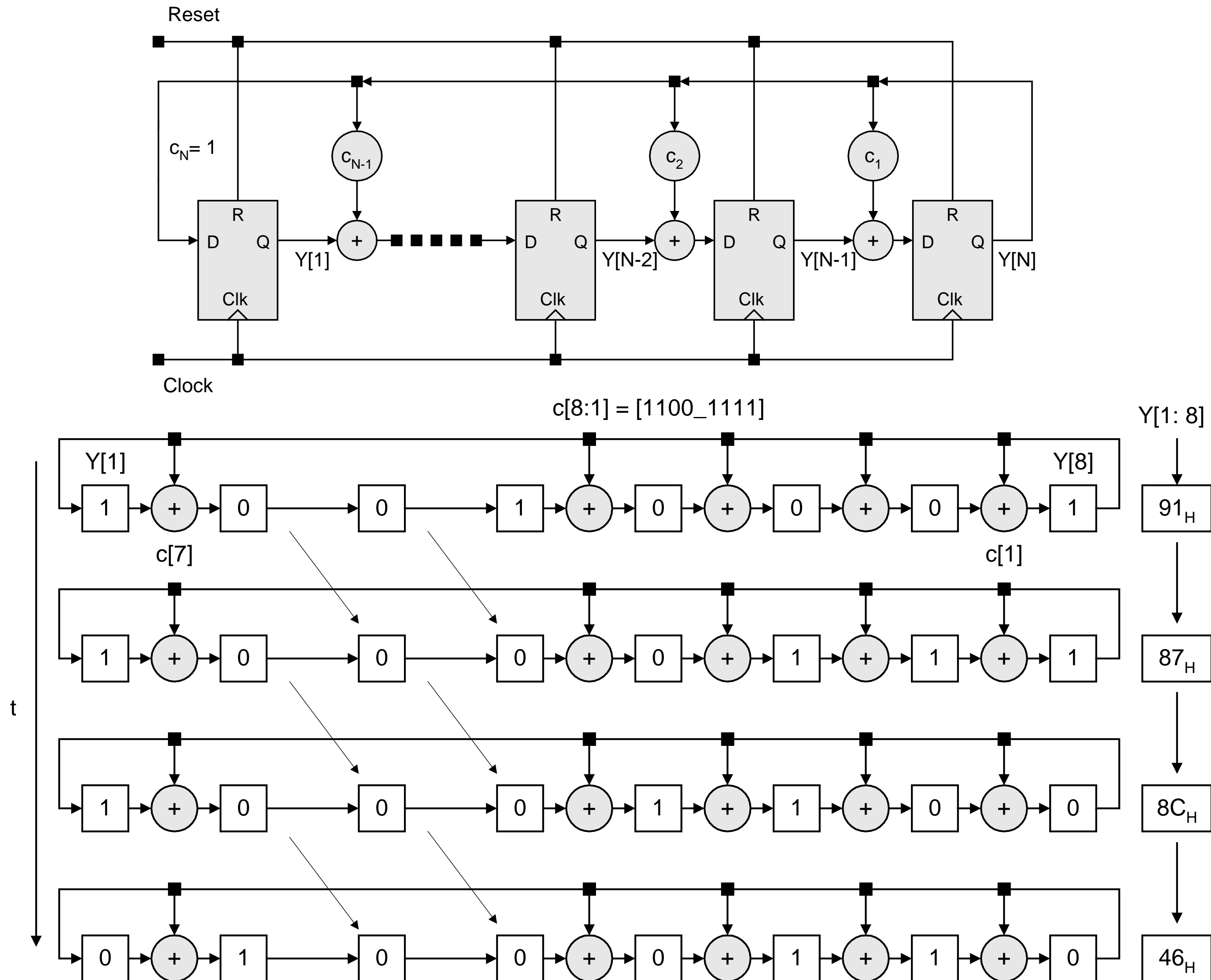
# Blocked (=) v.s. Nonblocking (<=)

- If no data dependency, results of blocked and nonblocking assignments are identical
- Strongly recommend
  - Blocked assignment for combinational logic using level sensitive behavior
  - Nonblocking assignments for edge sensitive behavior

# Shift Register Using Nonblocking Assignments

```
module shiftreg_nb (A, E, clk, rst);
  output A;
  input  E;
  input  clk, rst;
  reg          A, B, C, D;

 always @ (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
```

# Linear-Feedback Shift Register (Type II LFSR) Dataflow



# LFSR --- RTL Dataflow

```
module Auto_LFSR_RTL (Y, Clock, Reset);
 parameter                        Length = 8;
 parameter      [1: Length]       initial_state = 8'b1001_0001;   // 91h
 parameter      [1: Length]       Tap_Coefficient = 8'b1111_0011;

 input                            Clock, Reset;
 output         [1: Length]       Y;
 reg            [1: Length]       Y;

 always @  (posedge Clock)
   if (!Reset) Y <= initial_state;   // Active-low reset to initial state
     else begin
       Y[1] <= Y[8];
       Y[2] <= Tap_Coefficient[7] ? Y[1] ^ Y[8] : Y[1];
       Y[3] <= Tap_Coefficient[6] ? Y[2] ^ Y[8] : Y[2];
       Y[4] <= Tap_Coefficient[5] ? Y[3] ^ Y[8] : Y[3];
       Y[5] <= Tap_Coefficient[4] ? Y[4] ^ Y[8] : Y[4];
       Y[6] <= Tap_Coefficient[3] ? Y[5] ^ Y[8] : Y[5];
       Y[7] <= Tap_Coefficient[2] ? Y[6] ^ Y[8] : Y[6];
       Y[8] <= Tap_Coefficient[1] ? Y[7] ^ Y[8] : Y[7];
   end
endmodule
```

# LFSR --- RTL Repetitive Algorithm

```
module Auto_LFSR_ALGO (Y, Clock, Reset);
  parameter                          Length = 8;
  parameter         [1: Length]      initial_state = 8'b1001_0001;
  parameter         [1: Length]      Tap_Coefficient = 8'b1111_0011;
  input                              Clock, Reset;
  output            [1: Length]      Y;
  integer                            Cell_ptr;
  reg                                Y;

  always @  (posedge Clock)
   begin
    if (Reset == 0) Y <= initial_state;      // Arbitrary initial state, 91h
    else
     begin
       for (Cell_ptr = 2; Cell_ptr <= Length; Cell_ptr = Cell_ptr +1)
        if (Tap_Coefficient [Length - Cell_ptr + 1] == 1)
         Y[Cell_ptr] <= Y[Cell_ptr -1]^ Y [Length]; // ^ is xor
        else
         Y[Cell_ptr] <= Y[Cell_ptr -1];
       Y[1] <= Y[Length];
     end
   end
endmodule
```

# Verilog Repetitive Statements

- for, repeat, while, forever
  - All activities of all iterations are done in one time step
  - "disable" to terminate a named block
  - Some logic synthesis tools can only synthesize "for" loop
    - i.e., repeat, while, forever are not synthesizable in these tools

# Verilog Statement

- Statement can be
  - a single statement or
  - a block statement (i.e. **begin** ... **end**)
- A named block statement
  - i.e. **begin:** <block_name> ... **end**

# Ones Counter

- Verilog bitwise right-shift operator (>>),filling with '0'
  - Arithmetic right-shift (>>>)
- Compare the following two designs

// *count_of_1s* declares a named block of statements

```
// Original design
begin: count_of_1s
  reg [7: 0] temp_reg;

  count = 0;
  temp_reg = reg_a; // load a data word
  while (temp_reg)
    begin
      if (temp_reg[0])
            count = count + 1;
      temp_reg = temp_reg >> 1;
    end
end
```

```
// Alternative
begin: count_of_1s
  reg [7: 0] temp_reg;

  count = 0;
  temp_reg = reg_a; // load a data word
  while (temp_reg)
    begin
      count = count + temp_reg[0];
      temp_reg = temp_reg >> 1;
    end
  end
```

# Find_First_One

- Find the location of the first 1 in a 16-bit word
  - The word is assumed to contain at least one 1

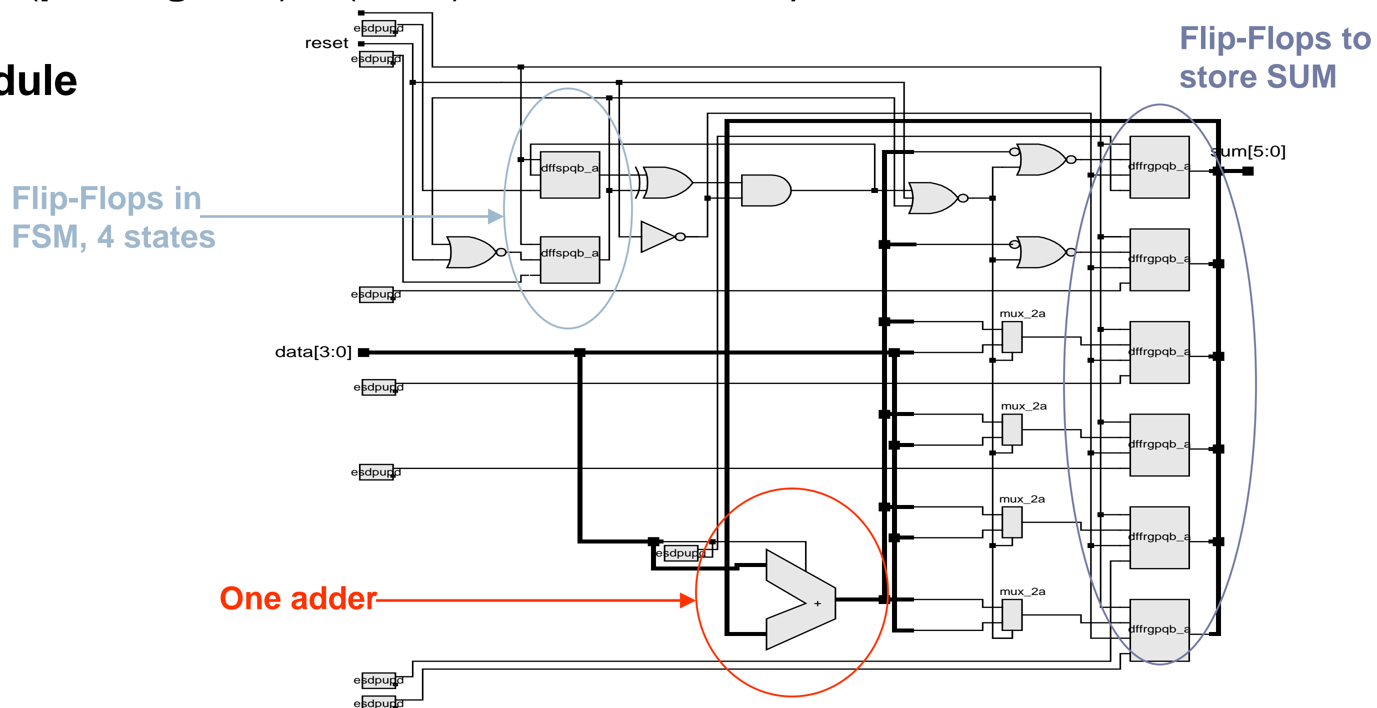
```
module find_first_one (index_value, A_word, trigger);
  output        [3: 0]      index_value;
  input         [15: 0]     A_word;
  input                     trigger;
  reg           [3: 0]      index_value;

  always @  (trigger)
    begin: search_for_1
      index_value = 0;
      for (index_value = 0; index_value <= 15; index_value = index_value + 1)
        if (A_word[index_value] == 1)
                  disable search_for_1;
    end
endmodule
```

# Multicycle Operations -- 4-cycle Adder

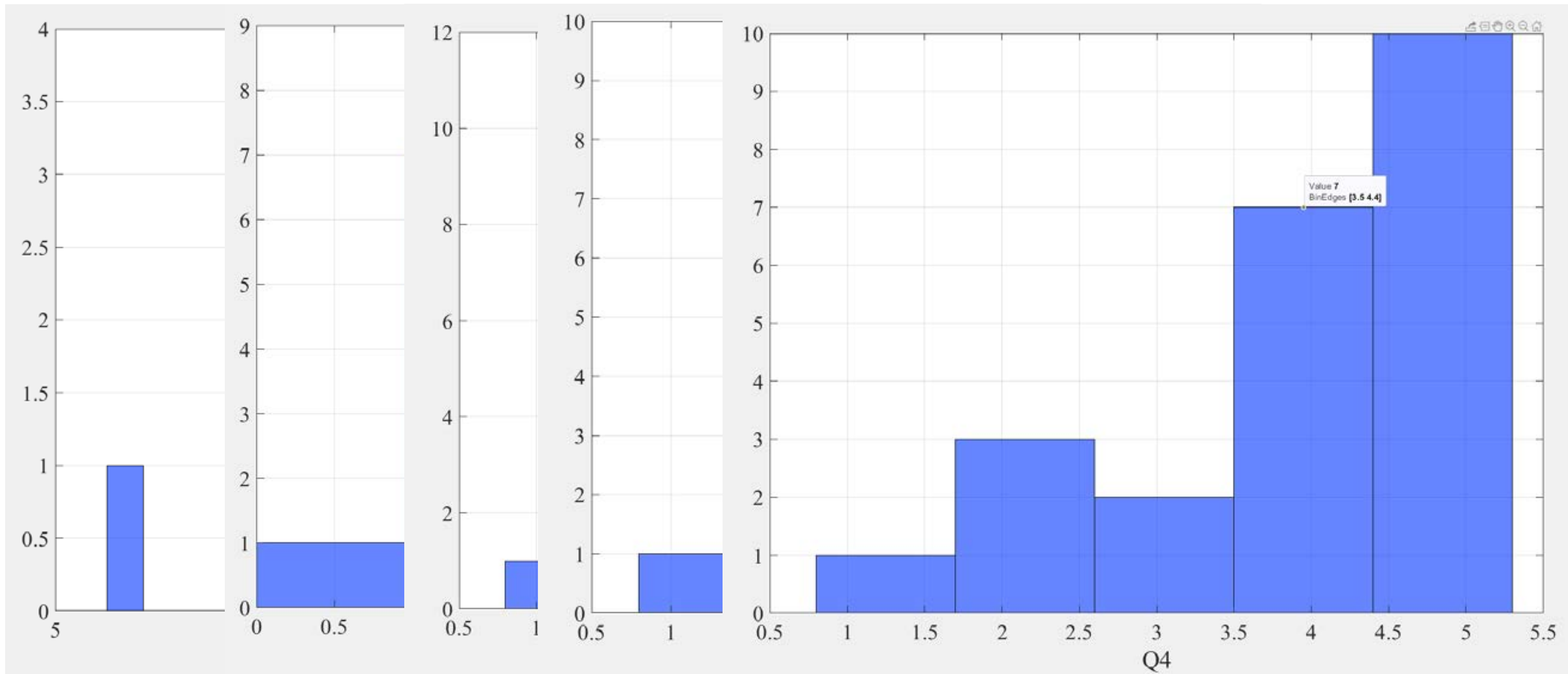
- Some digital machines have repetitive operations distributed over multiple clock cycles
  - Can be modeled in Verilog by a synchronous cyclic behavior that has as many nested edge-sensitive event control expressions as needed to complete the operations
  - May not be synthesizable
- Example: 4-cycle adder
  - To form the sum of four successive samples of a datapath
    - Store the samples in registers then use multiple adders
    - Or, one adder to accumulate the sum sequentially
      - One FSM to control the 4-cycle operation and only one adder
        - The resulting synthesized implementation
  - To ensure proper re-initialization, “disable” is in each clock cycle
    - Regardless when the “reset” is asserted

# 4-cycle Adder

```
module add_4cycle (sum, data, clk, reset);
  output        [5: 0]    sum;
  input         [3: 0]    data;
  input                   clk, reset;
  reg                     sum;
always @ (posedge clk) begin:  add_loop
    if (reset) disable add_loop; else sum <= data;
      @ (posedge clk) if (reset) disable add_loop;   else sum <= sum + data;
        @ (posedge clk) if (reset) disable add_loop;  else sum <= sum + data;
          @ (posedge clk) if (reset) disable add_loop;  else sum <= sum + data;
   end
endmodule
```



# Class Test 1



# Algorithmic State Machine (ASM) Charts

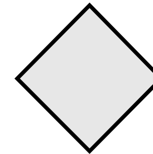
- State Transition Graphs (STGs)
  - Indicate the transitions that result from inputs applied to the state machine in a particular state
  - Do not directly display the evolution of states under the application of input data
- ASM Charts
  - Abstraction of functionality of a sequential machine
    - Reveal the sequential steps of a machine's activity
  - Focus on activities rather than content of storage elements
    - Example: the counter to be introduced shortly
      - Three states: idle, incrementing and decrementing
      - Independent of counter word width
  - ASM chart elements
    - state box
    - decision box
    - conditional box
  - Clock governs transitions between states
  - Linked ASM charts describe complex machines
    - ASM charts represent both Mealy and Moore machines

# ASM Chart Elements

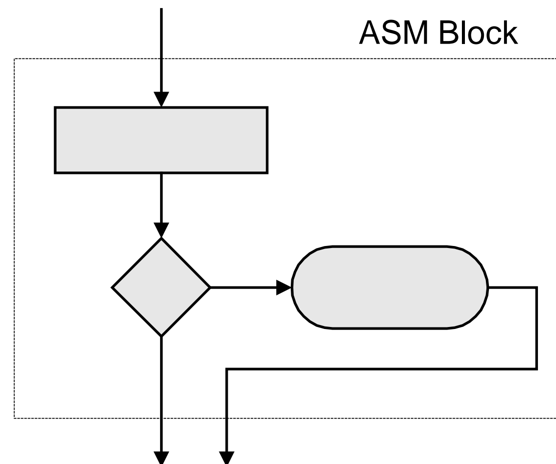
- State box
  - Each state box represents the state of the machine between synchronizing clock events
- Decision box
- Conditional box

State Box

Conditional Output or Register Operation Box

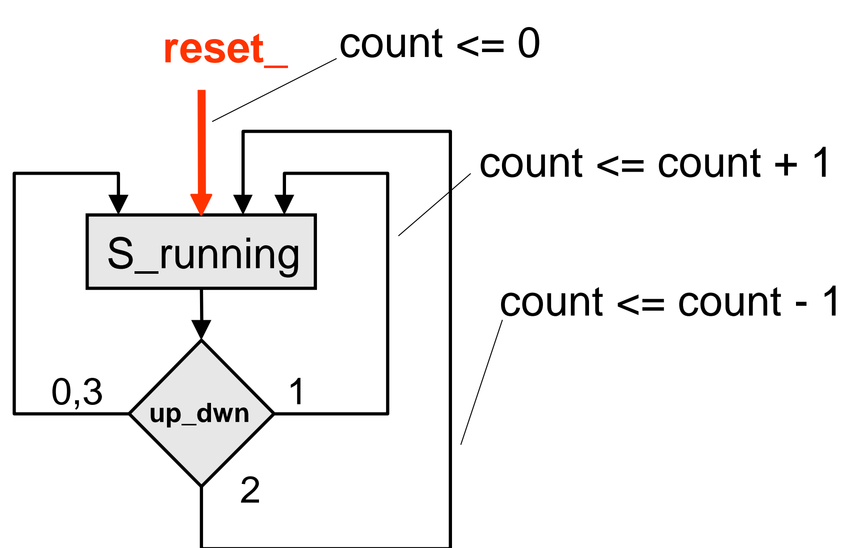


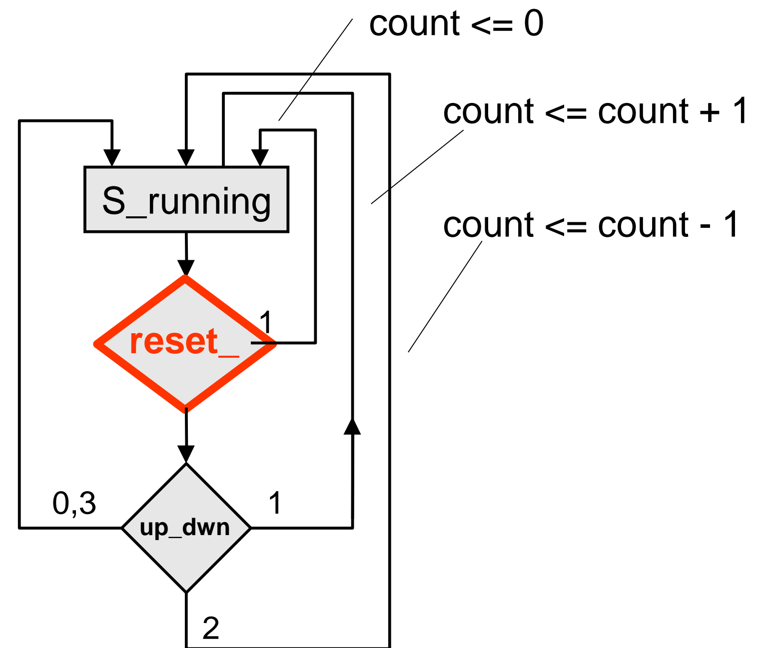
Decision Box



# Asyn/Synchronous Reset in ASM

- Asynchronous reset: a RESET input to the reset state box
- Synchronous reset: one decision box of RESET input



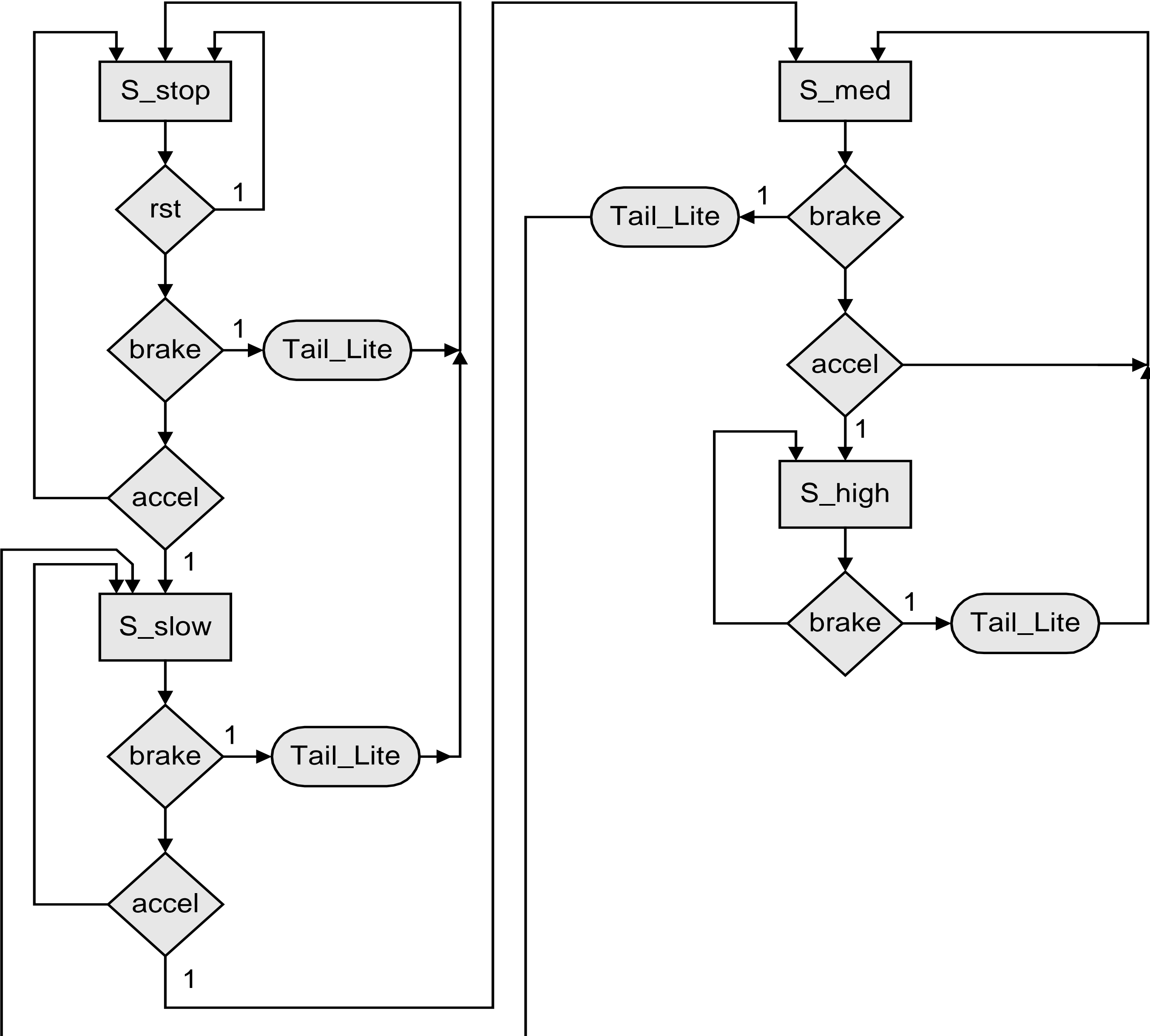



# ASM Chart (cont.)

- Only paths leading to a change in states are shown in ASM
  - If a variable not appear in a decision box on a path leaving a state, then the path is independent of the value of the variable

# ASM Chart Example: Tail Light Controller

## A Mealy Machine with Synchronous Reset



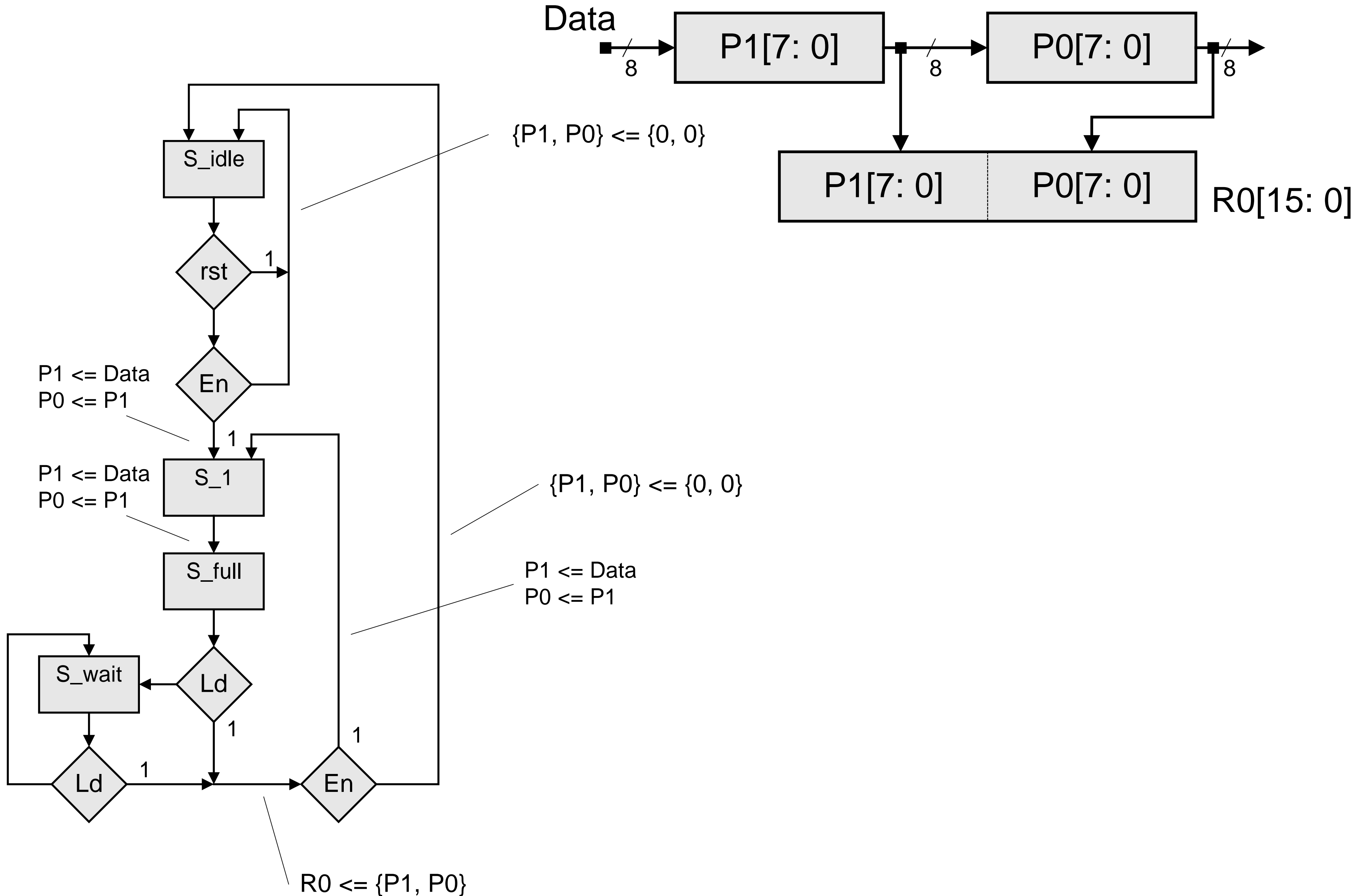
# ASM and Datapath (ASMD) Charts

- To form an ASMD: modify ASM (i.e. controller) by annotating each of its paths to indicate the concurrent register operations (i.e. datapath operations) when the controller makes a transition along the path
  - Not in conditional boxes
  - Not in state boxes
  - Because the datapath registers are not part of the controller
    - Fact: output generated by the controller controls the datapath register
- Clarify a design of a sequential machine by separating the design of its datapath from the design of the controller
- ASMD chart maintains a clear relationship between a datapath and its controller
  - Outputs generated by the controller control the datapath register
  - Outputs generated by datapath report the status of datapath back to the controller

# 2:1 Decimator Using 2-stage Pipeline

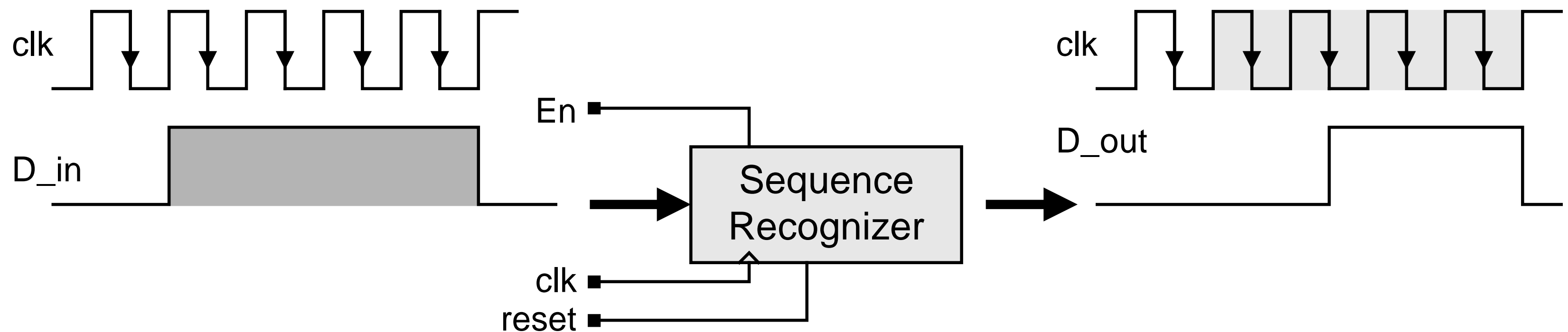
- Used to move data from a high clock rate datapath to a lower data rate datapath
  - Can also used to convert data from a parallel format to a serial format
- ASMD of the 2:1 decimator
  - A Mealy machine with synchronous reset to S_idle
  - An incomplete ASMD
    - Because no conditional outputs
      - i.e the output of the controller to control how datapath works
        - Such as adding an output for load-register
  - E.g. "Ld" state represents load to R0 since R0<={P1,P0} on the path leaving the state when Ld=1
  - Note that datapath register operations made with a <u>nonblocking</u> assignment are concurrent
    - Hence no race between R0<={P1,P0} and {P1,P0}<={0,0}

# 2:1 Decimator Using 2-stage Pipeline (cont.)



# Synthesis of Sequence Recognizer

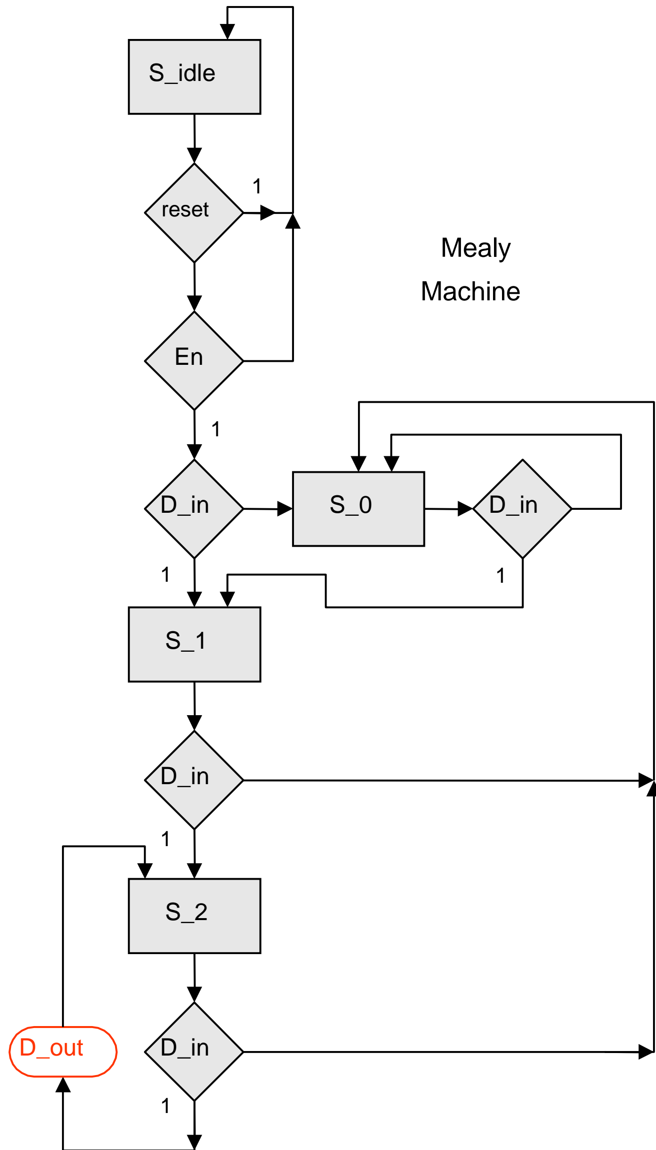
- Example: detect 3 consecutive 1s
  - Assert *D_out* when a given pattern of consecutive bits has been received in its serial input stream, *D_in*
  - Apply data on the rising edge of the clock if the state transitions are to occur on the falling edge of the clock, and visa-versa
    - ➢ Recall the general rule for exercising FSM

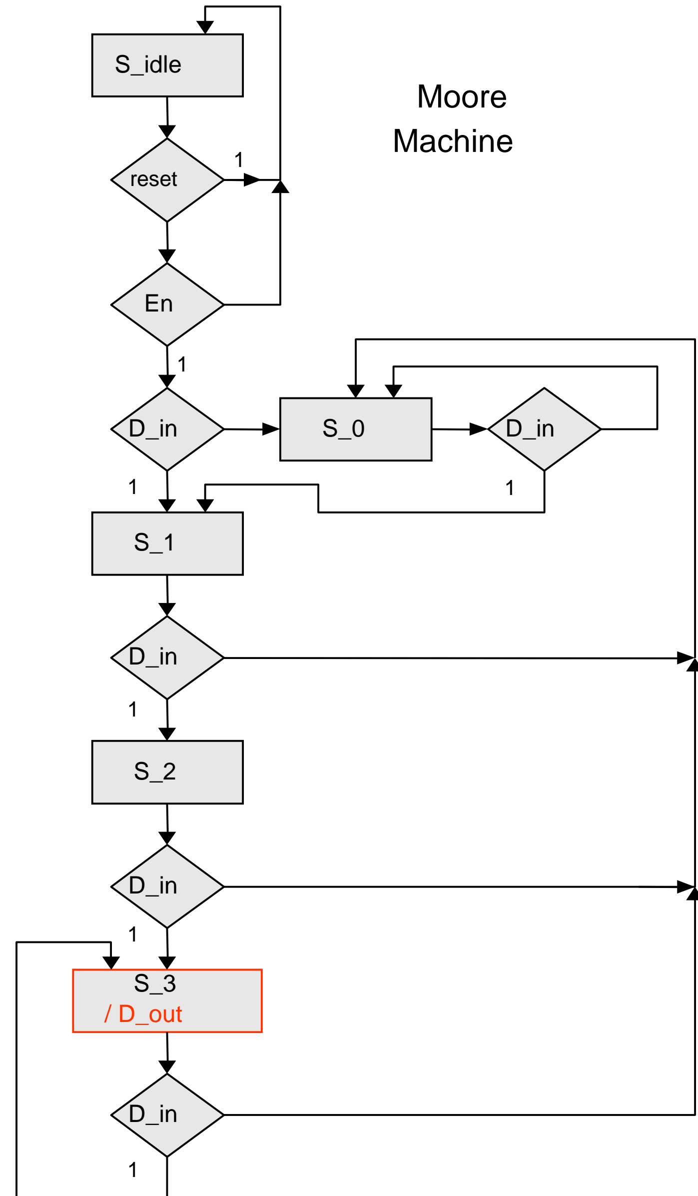


# Conventions to Describe Sequence Recognizers

- The output of a Mealy machine is valid immediately before the active edge of the clock controlling the machine
  - Data must be stable prior to active edge for at least the setup time
- Successive values inputs are received in successive clock cycles.
- A *non-resetting* machine continues to assert its output if the input bit pattern is overlapping
- A *resetting* machine asserts for one cycle after detecting the input sequence, and then de-asserts for one cycle before detecting the next sequence of bits

# Mealy and Moore ASMs (3 Consecutive 1s)



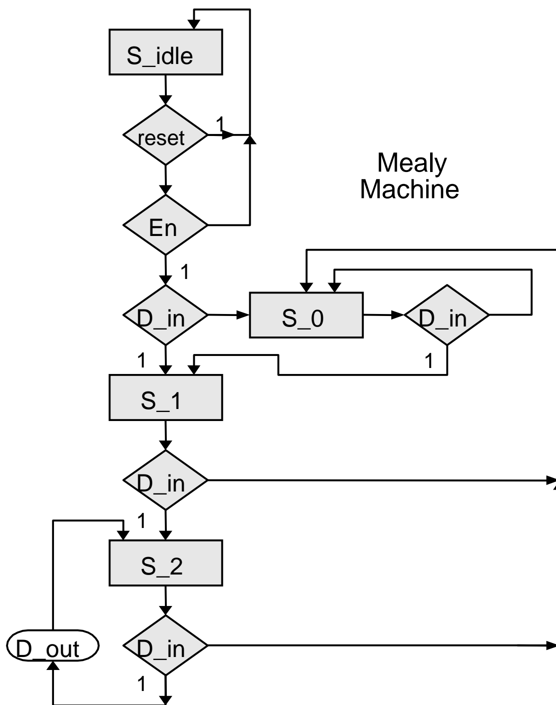



# Mealy and Moore for 3 Consecutive 1s (cont.)

- Both are non-resetting
  - How to modify them into resetting sequence recognizers?
- Moore has one more state than Mealy
- The Mealy machine anticipates *D_in* and asserts *D_out* before the third clock
- The Moore machine does not anticipate *D_in*
  - That is, the Moore machine asserts D_out in the state reached after the third active edge of the clock

# Sequence Recognizer for 3 Consecutive 1s (cont.)

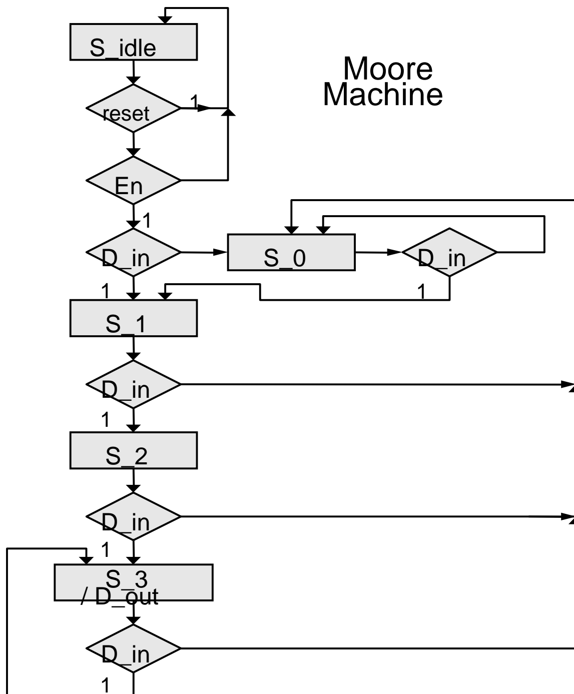
```
module Seq_Rec_3_1s_Mealy
        (D_out, D_in, En, clk, reset);
 output          D_out;
 input           D_in, En;
 input           clk, reset;
 // Binary coding for states
 parameter S_idle =       0;
 parameter S_0 =          1;
 parameter S_1 =          2;
 parameter S_2 =          3;
 reg[1: 0] state, next_state;
```




```
always @ (negedge clk)
  if (reset == 1) state <= S_idle; else state <= next_state;

always @ (state or D_in) begin
  case (state)     // Partially decoded
    S_idle:  if ((En == 1) && (D_in == 1)) next_state = S_1;
             else  if ((En  == 1) && (D_in == 0)) next_state = S_0;
             else                             next_state = S_idle;
    S_0:     if (D_in == 0)                   next_state = S_0;
             else  if (D_in == 1)             next_state = S_1;
             else                             next_state = S_idle;
    S_1:     if (D_in == 0)                   next_state = S_0;
             else if (D_in == 1)              next_state = S_2;
             else                             next_state = S_idle;
    S_2:     if (D_in == 0)                   next_state = S_0;
             else if (D_in == 1)              next_state = S_2;
             else                             next_state = S_idle;
    default:                                  next_state = S_idle;
  endcase
end
assign D_out = ((state == S_2) && (D_in == 1 )); // Mealy output
endmodule
```

# Sequence Recognizer for 3 Consecutive 1s (cont.)

```
module Seq_Rec_3_1s_Moore
        (D_out, D_in, En, clk, reset);
  output     D_out;
  input      D_in, En;
  input      clk, reset;
  // Binary coding for states
  parameter S_idle =   0;
  parameter S_0 =      1;
  parameter S_1 =      2;
  parameter S_2 =      3;
  parameter S_3 =      4;
  reg[2: 0]  state, next_state;
```




```
always @ (negedge clk)
  if (reset == 1) state <= S_idle; else state <= next_state;

always @ (state or D_in) begin
  case (state)
    S_idle: if ((En == 1) && (D_in == 1))       next_state = S_1; else
             if ((En  == 1) && (D_in == 0))      next_state = S_0;
            else                                 next_state = S_idle;
    S_0:    if (D_in == 0)                       next_state = S_0; else
            if (D_in == 1)                       next_state = S_1;
            else                                 next_state = S_idle;
    S_1:    if (D_in == 0)                       next_state = S_0; else
            if (D_in == 1)                       next_state = S_2;
            else                                 next_state = S_idle;
    S_2, S_3:if (D_in == 0)                      next_state = S_0; else
              if (D_in == 1)                     next_state = S_3;
              else                               next_state = S_idle;
    default:                                     next_state = S_idle;
  endcase
end
assign D_out = (state == S_3);    // Moore output
endmodule
```

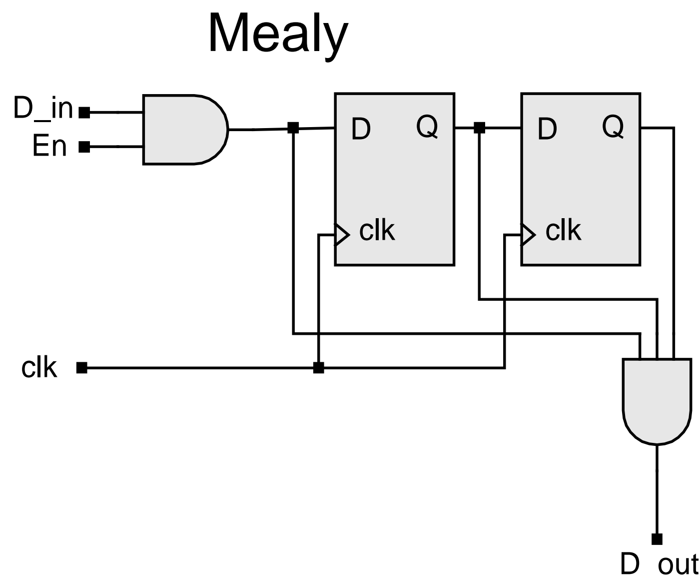
# Alternative Design for Sequence Recognizer

- Alternative approach: Shift input bits through a register and detect contents
  - Consider sequence recognizer as a datapath unit
  - Such as a shift register
  - Then compare the content of shift register with the expected pattern
- Note: an explicit state machine implementation of the alternative design for a sequence recognizer is not necessarily the most efficient implementation

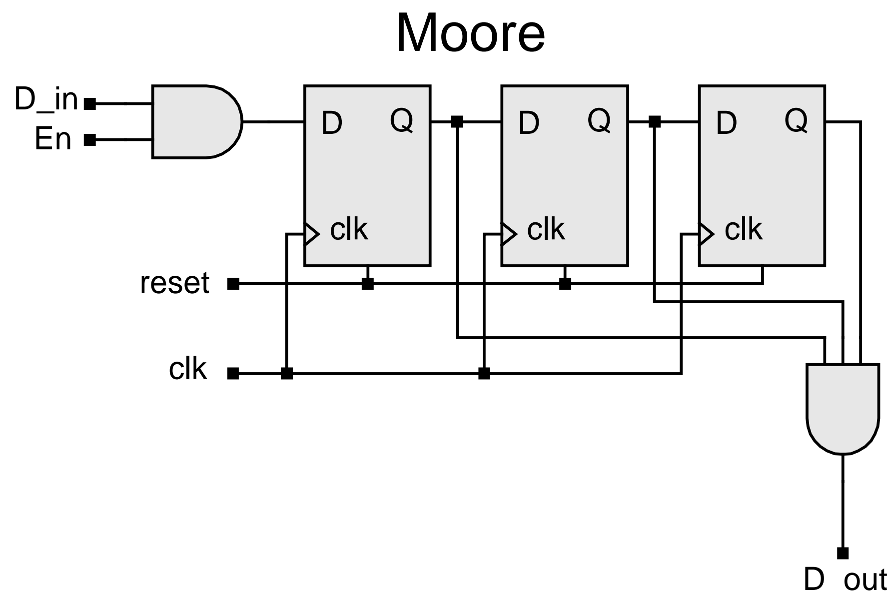
# Alternative Design for Sequence Recognizer (cont.)

- The Mealy/Moore machines below are gated the datapath with *En*
  - *What happens if En=0?*
    - Register content will be lost
- Mealy has one less FF than Moore

Mealy




Moore



# Alternative Design for Sequence Recognizer (cont.)

```
module Seq_Rec_3_1s_Mealy_Shft_Reg (D_out, D_in, En, clk, reset);
  output              D_out;
  input               D_in, En;
  input               clk, reset;
  parameter           Empty = 2'b00;
  reg      [1: 0]     Data;
  always @ (negedge clk)
    if (reset == 1) Data <= Empty; else if (En  == 1) Data <= {D_in, Data[1]};
  assign D_out = ((Data == 2'b11) && (D_in == 1 )); // Mealy output depends on primary input
endmodule

module Seq_Rec_3_1s_Moore_Shft_Reg (D_out, D_in, En, clk, reset);
  output              D_out;
  input               D_in, En;
  input               clk, reset;
  parameter           Empty = 3'b000;
  reg      [2: 0]     Data;
  always @ (negedge clk)
    if (reset == 1) Data <= Empty; else if (En == 1) Data <= {D_in, Data[2:1]};
  assign D_out = (Data == 3'b111);     // Moore output depends on state only
endmodule
```

# Design of a Datapath Controller

1. **Understand the problem**
   - Especially the register operations that must execute on a given datapath architecture
2. **Define ASM**
   - A state machine controlled by primary inputs and status of datapath register (i.e. the feedback linkage from datapath to controller)
3. **Create ASMD**
   - Annotating ASM with datapath operations associated with state transitions (i.e. path) of the controller
   - Register operation of ASMD written in register transfer notations with NONBLOCKING assignments
     - since they are executed concurrently in the datapath
4. **Controller outputs to datapath**
   - For Moore machines: Annotate state of the controller with unconditional output signals (i.e. outputs of a state)
   - For Mealy machines: Include conditional boxes for controller output signals to control datapath
5. **Feedback linkage from datapath to controller**
   - If there are signals reports status of datapath back to the controller, then use decision box
6. **Integration**
   - Integrate the verified datapath module and the verified controller module with one parent module to verify the overall functionality

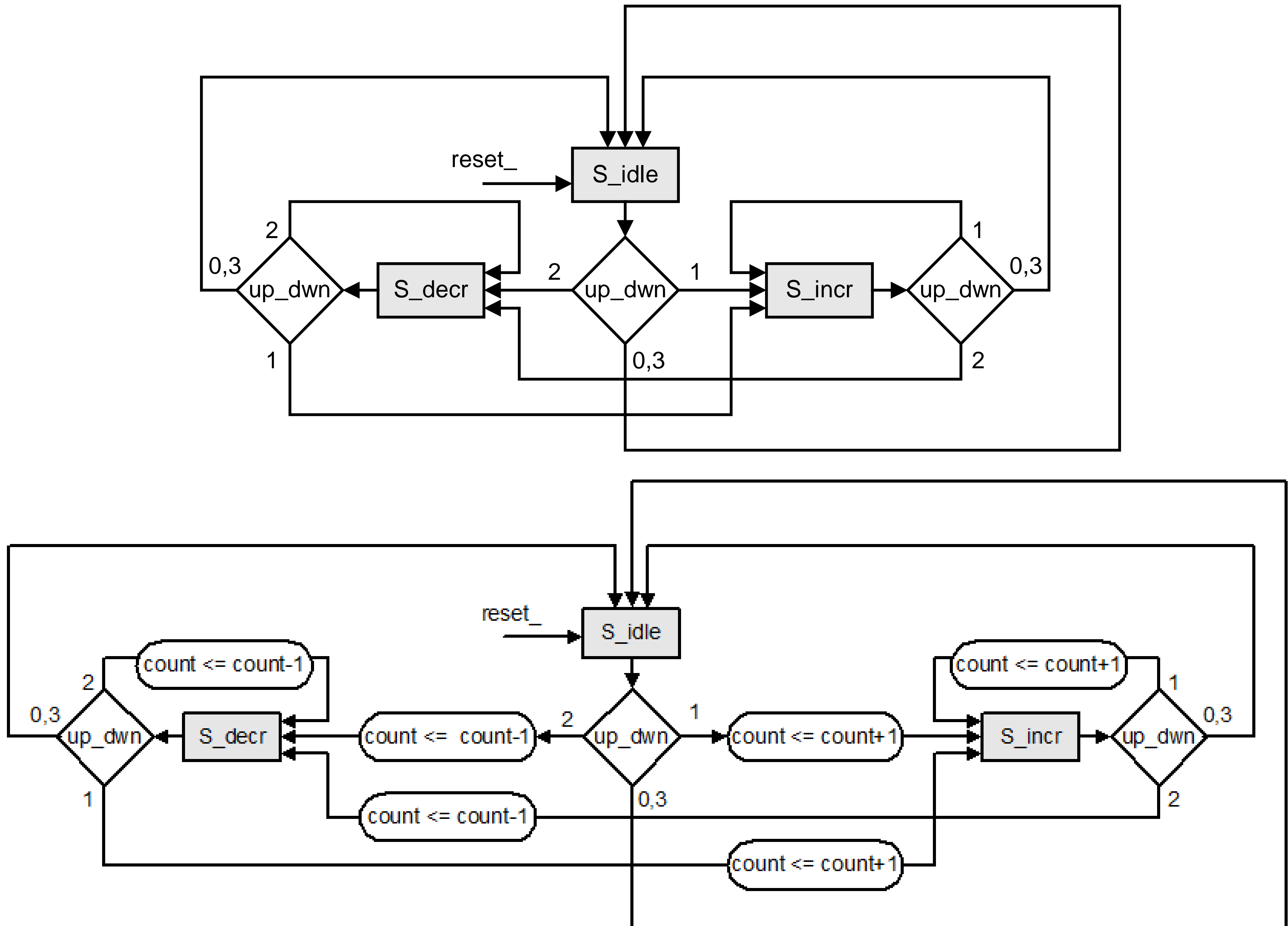
# Counters and Registers

- Storage elements of counters and registers usually have the same synchronizing and control signals
  - One exception: ripple counter
    - Connects the output of a stage to the clock input of an adjacent stage
- Counters with asynchronous reset
- Ring counter
- Up/down/load counter
- Shift register
- Parallel load register
- Universal shift register
- Register file

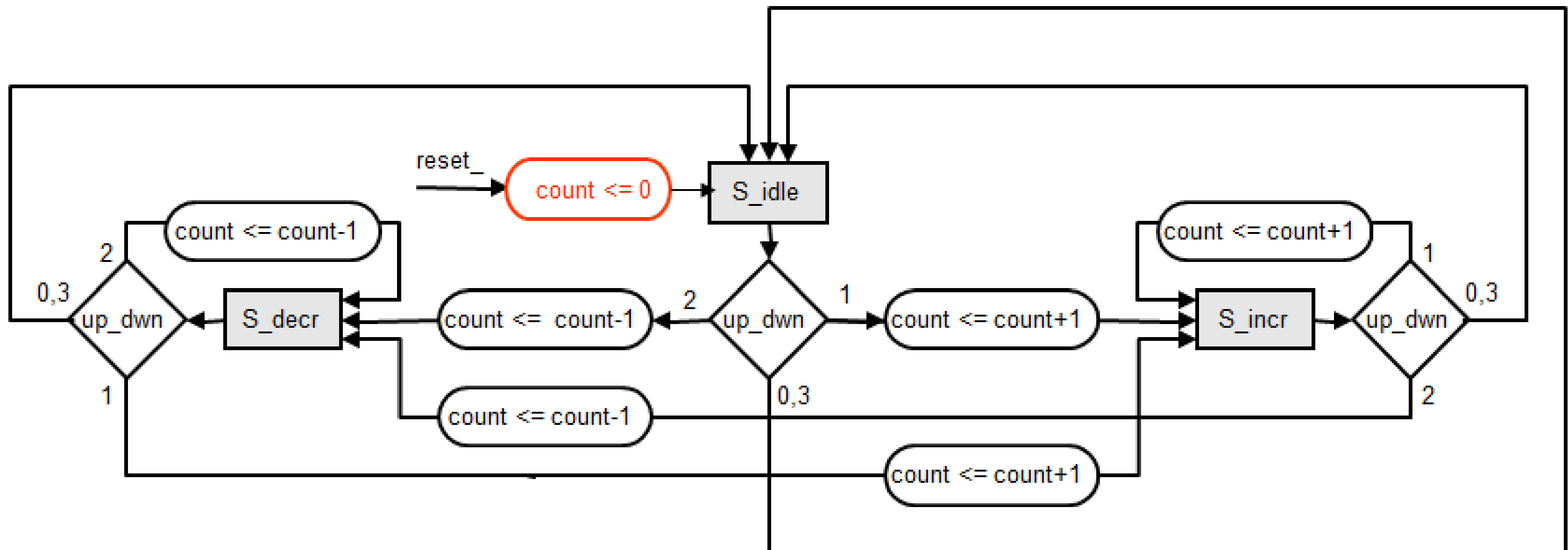
# Counters

- The ASM/ASMD have no indication of the bit-width of the counter
- Three states: S_idle, S_incr and S_decr
  - May be further simplified to a single state, S_running
- 2-bit input up_down to count up(1), count down(2) and hold the count (0 and 3)
- Active low asynchronous reset

# Counters(3 states) with Async Reset_



# Counters (cont.)

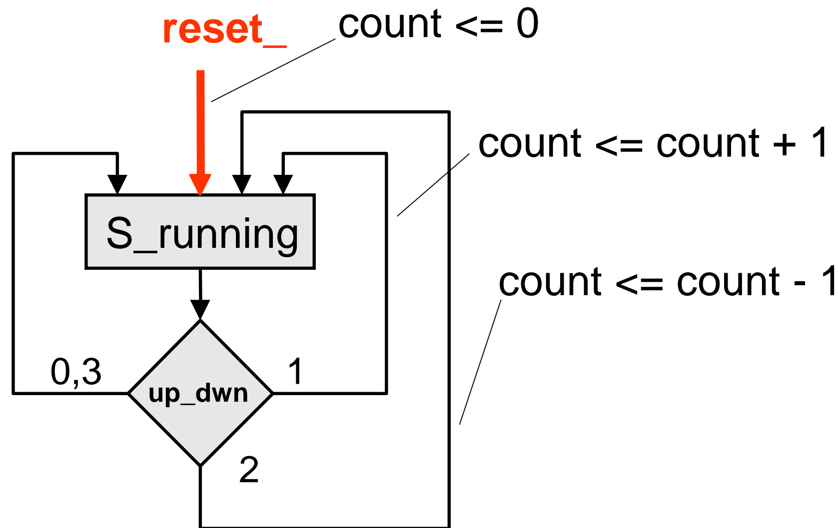


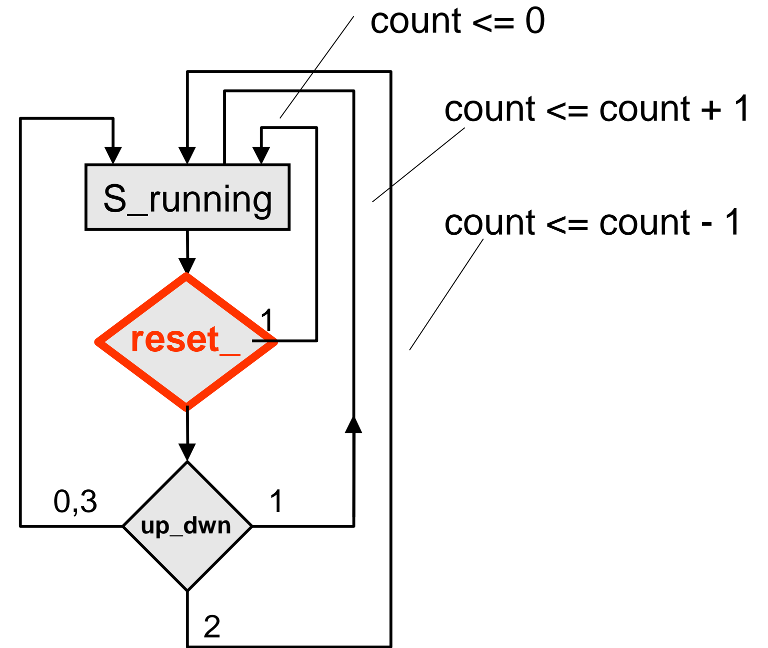
# Counters (3 states) with Async Reset_ (cont.)

- It is an implicit state machine
  - No explicit states (S_idle, S_incr, S_decr) used in the design
  - Implemented using if-then-else within edge-sensitive behavior

```
module Up_Down_Implicit1 (count, up_dwn, clock, reset_);
 output  [2: 0]      count;
 input   [1: 0]      up_dwn;
 input               clock, reset_;

 reg [2: 0] count;

 always @  (negedge clock or negedge reset_)
  if (reset_ == 0)                                  count <= 3'b0; else
   if (up_dwn == 2'b00 || up_dwn == 2'b11) count <= count; else
    if (up_dwn == 2'b01)                            count <= count + 1; else
     if (up_dwn == 2'b10)                           count <= count –1;

endmodule
```

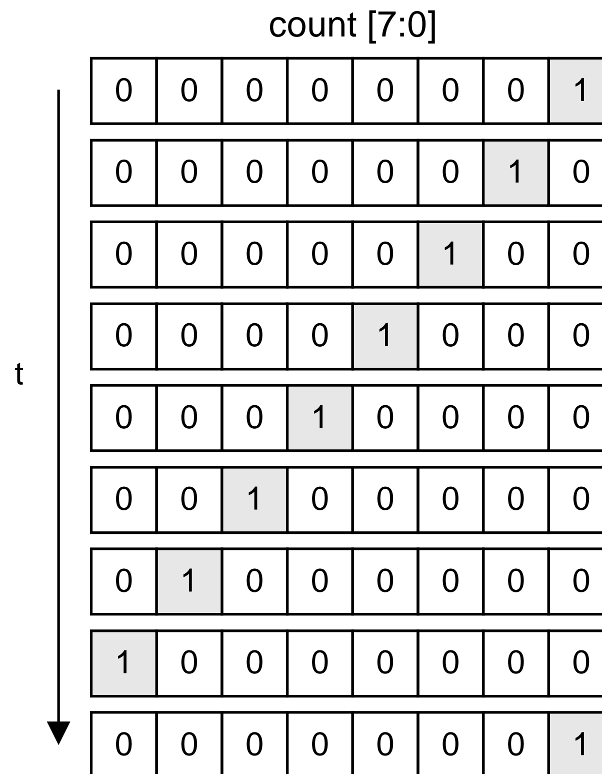
# Simplified Counter ASMDs with Async/Sync Reset_



# Ring Counter

- Ring counter asserts a single bit that circulates through the counter in a synchronous manner



# Ring Counter (cont.)

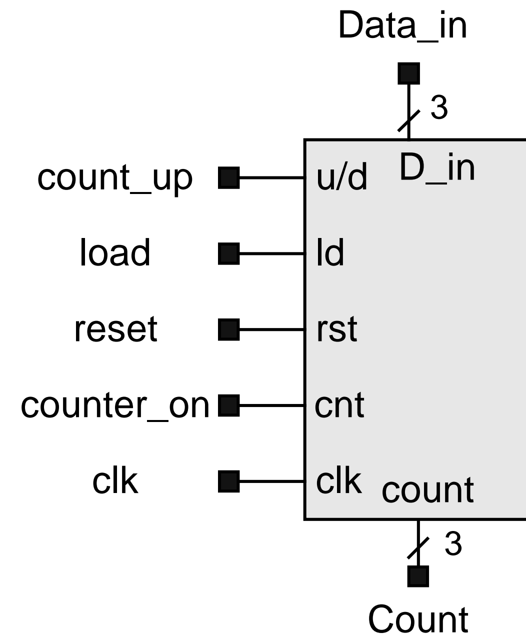
- Activity of the machine is the same in every clock cycle
- This implementation is an implicit state machine

```
module ring_counter (count, enable, clock, reset);
  output          [7: 0]     count;
  input                      enable, reset, clock;
  reg             [7: 0]     count;

  always @  (posedge reset or posedge clock)
    if (reset == 1'b1)     count <= 8'b0000_0001; else
      if (enable == 1'b1) count <= {count[6: 0], count[7]};        // Concatenation operator
endmodule
```

VHDL:: count <= count[6: 0] & count[7];

# Up/Down/Load Counter

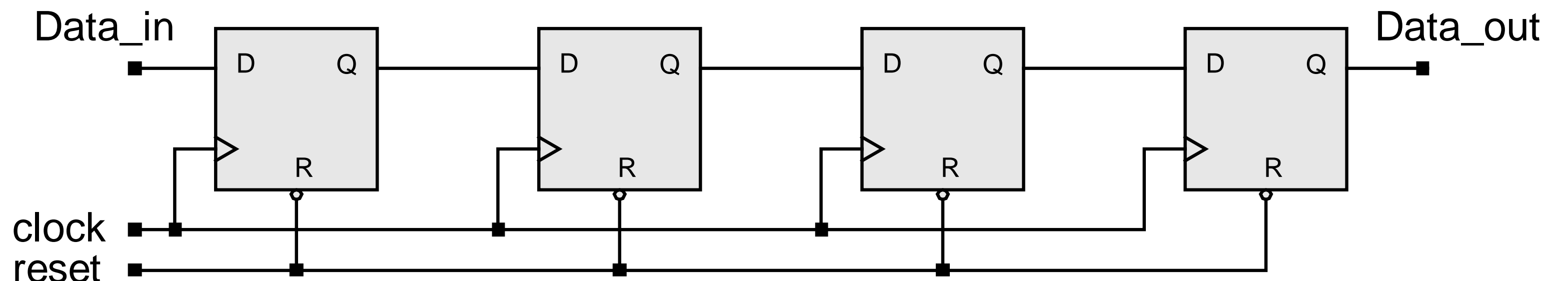
```
module up_down_counter (Count, Data_in, load, count_up, counter_on, clk, reset);
  output        [2: 0]     Count;
  input                    load, count_up, counter_on, clk, reset,;
  input         [2: 0]     Data_in;
  reg           [2: 0]     Count;

  always @  (posedge reset or posedge clk)
    if (reset == 1'b1) Count <= 3'b0; else
      if (load == 1'b1) Count <= Data_in; else
        if (counter_on == 1'b1) begin
          if (count_up   == 1'b1) Count <= Count +1;
            else Count <= Count –1;
      end
endmodule
```



# Shift Register

- Remember the "model trap"
  - Must use nonblocking assignments in this design

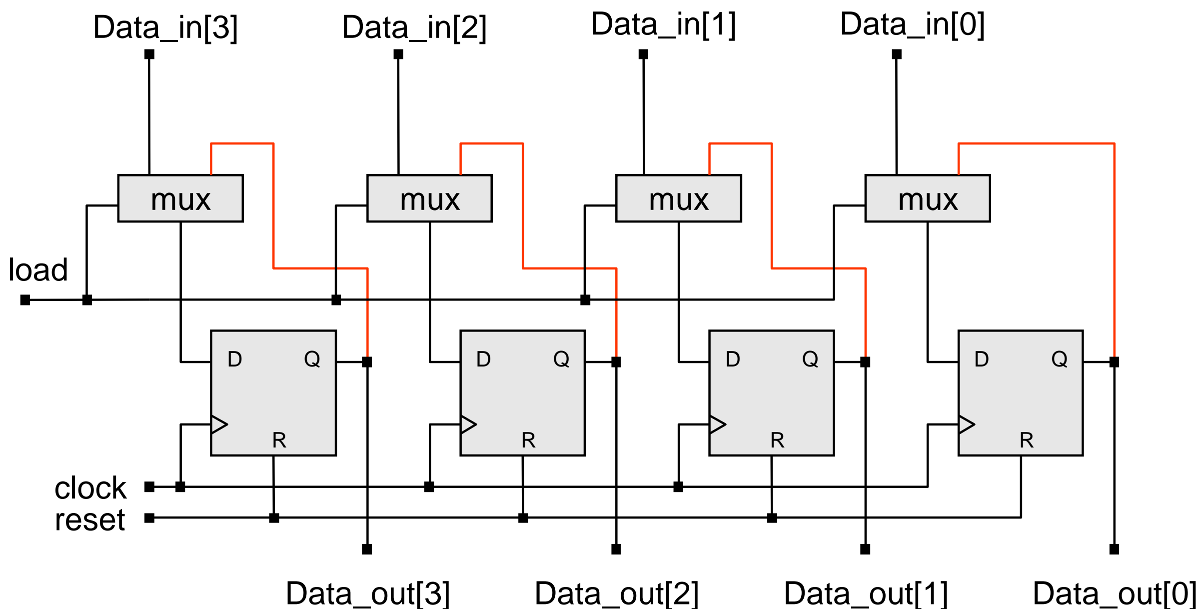
```
module Shift_reg4 (Data_out, Data_in, clock, reset);
  output              Data_out;
  input               Data_in, clock, reset;
  reg      [3: 0]     Data_reg;

  assign  Data_out = Data_reg[0];

  always @  (negedge reset or posedge clock)
    begin
      if (reset == 1'b0)   Data_reg <= 4'b0;
      else                 Data_reg <= {Data_in, Data_reg[3:1]};  //shift right
    end
endmodule
```



# Parallel Load Register

- MUX is synthesized from “else if (load==1’b1)”
  - How about “else” i.e. (load==1’b0)?
    - If not specified, retain the previous value

```
module Par_load_reg4 (Data_out, Data_in, load, clock, reset);
  input     [3: 0]      Data_in;
  input                 load, clock, reset;
  output    [3: 0]      Data_out;           // Port size
  reg                    Data_out;          // Data type
  always @  (posedge reset or posedge clock)
   begin
    if (reset == 1'b1)
       Data_out <= 4'b0;
    else if (load == 1'b1)
       Data_out <= Data_in;
   end
endmodule
```

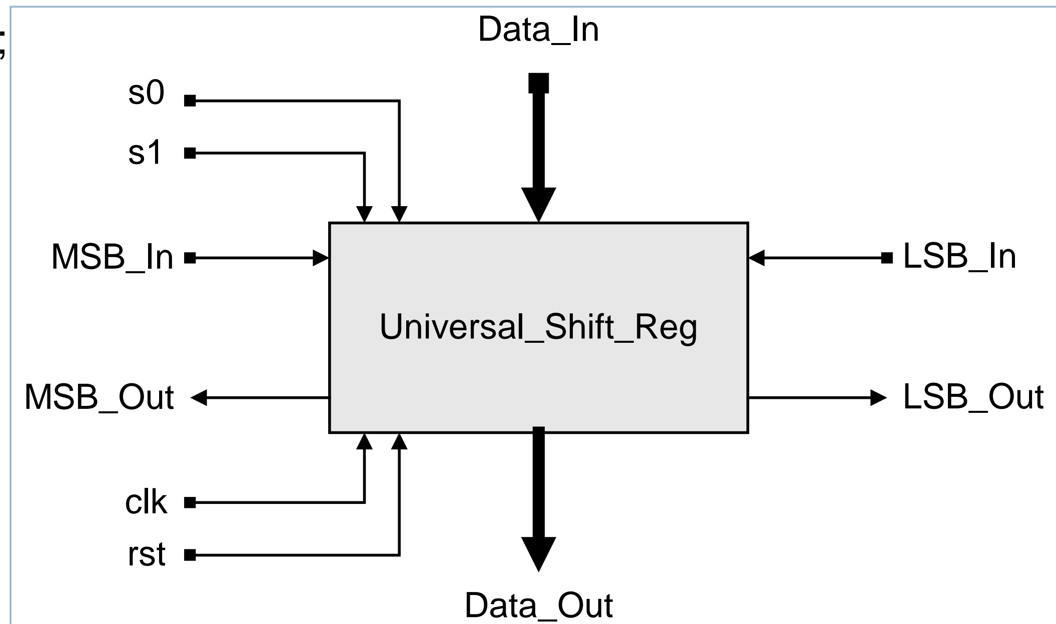


# Shift Registers

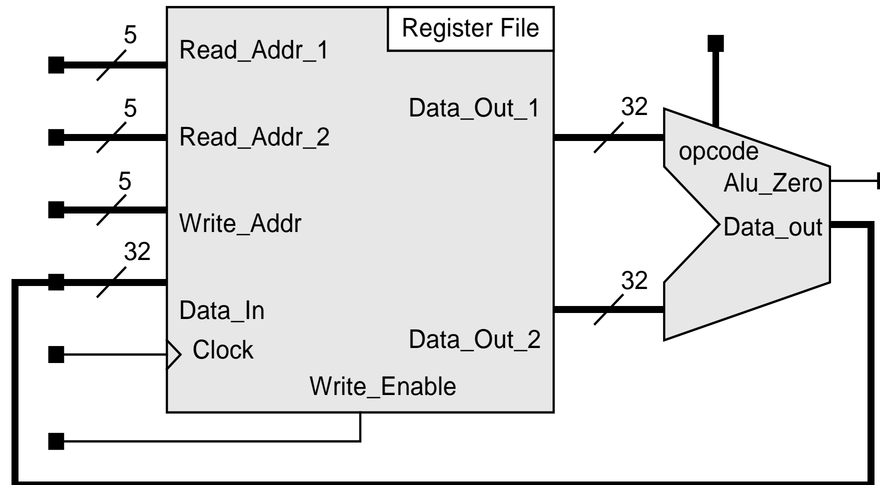
- Shift register with parallel load
  - later
- Arithmetic shift register
  - For signed number operation
    - MSB is preserved
  - Shift-left: multiply by 2
  - Shift-right: divide by 2

# Universal Shift Register

```
module Universal_Shift_Reg
  (Data_Out, MSB_Out, LSB_Out, Data_In, MSB_In, LSB_In, s1, s0, clk, rst);
 output  [3: 0]   Data_Out;
 output           MSB_Out, LSB_Out;
 input   [3: 0]   Data_In;
 input            MSB_In, LSB_In;
 input            s1, s0, clk, rst;
 reg              Data_Out;

 assign MSB_Out = Data_Out[3];
 assign LSB_Out = Data_Out[0];

 always @ (posedge clk) begin
  if (rst) Data_Out <= 0;
  else  case ({s1, s0})
   0:    Data_Out <= Data_Out;                    // Hold
   1:    Data_Out <= {MSB_In, Data_Out[3:1]};     // Serial shift from MSB
   2:    Data_Out <= {Data_Out[2: 0], LSB_In};    // Serial shift from LSB
   3:    Data_Out <= Data_In;                     // Parallel Load
  endcase
 end
endmodule
```



# Register File




```
module Register_File (Data_Out_1,Data_Out_2,Data_in,
                        Read_Addr_1,Read_Addr_2,Write_Addr,Write_Enable,Clock);
  output  [31: 0]   Data_Out_1, Data_Out_2;
  input   [31: 0]   Data_in;
  input   [4: 0]    Read_Addr_1, Read_Addr_2, Write_Addr;
  input             Write_Enable, Clock;
  reg     [31: 0]   Reg_File [31: 0];    // 32bit  x32 word memory declaration

  assign Data_Out_1 = Reg_File[Read_Addr_1];
  assign Data_Out_2 = Reg_File[Read_Addr_2];
  always @ (posedge Clock) begin
    if (Write_Enable) Reg_File [Write_Addr] <= Data_in;
  end
endmodule
```

# “Concept of Memory” in Verilog

- Memory
  - Declaration an array of words
  - E.g. reg [31:0] data_out; // one 32-bit word
    reg [31:0] Reg_file [31:0]; // 32x32 bit word memory
- Verilog does not support 2-dimensional array
  - However, a word in a Verilog memory can be addressed directly
    - E.g., Reg_file [12]
  - A cell bit in a word can also be addressed indirectly by first loading the word into a buffer register then addressing the bit of the word
    - E.g. Data_out = Reg_file [12];
      Data_out [1:0]
- Decoder are synthesized automatically by synthesis tool in Reg_file[] to decode the address which locates a specific register