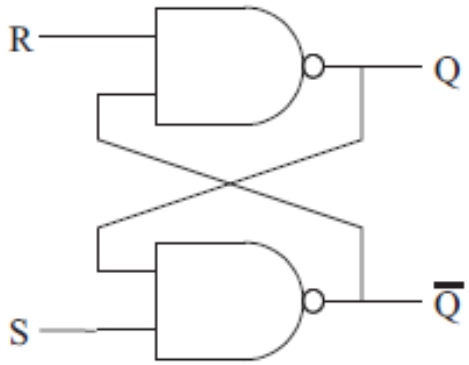
Sequential Logic in SV

SR Latch: Structural Model

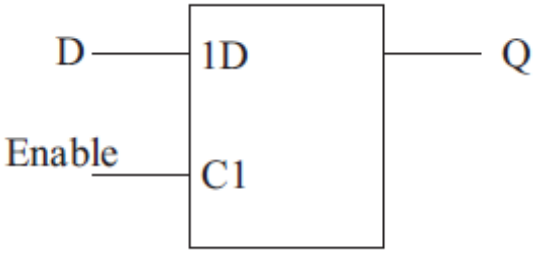
- The latch is modeled using two NAND gates.

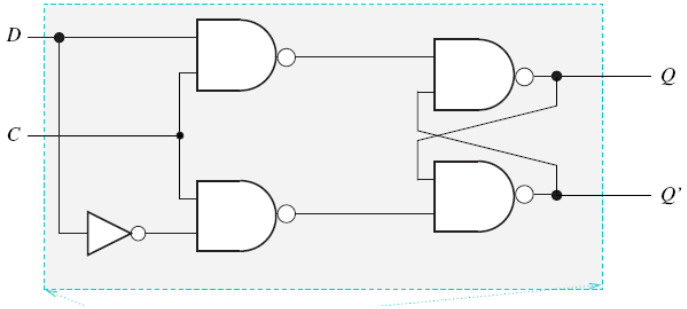



```
module rslatch1 (output wire q, qbar,
                 input logic r, s);
    nand n0(q, qbar, r);
    nand n1(qbar, q, s);
endmodule
```

Behavioral Modeling of Latches

- What is a latch?
  - Nature?
  - Two aspects:
    - Value latched
    - Value Stored





Verilog's **always**

- In Verilog, **t**he general-purpose **always** procedure can be used to model any type of logic, including
  - combinational logic
  - level-sensitive sequential logic (latches)
  - clocked sequential logic (flip-flops)

- Because of this "**massive**" flexibility, it is easy to produce the wrong hardware inadvertently.

SV Solution: New **always**

- SV introduces always_latch, always_ff, and always_comb to reduce the risk of common errors.

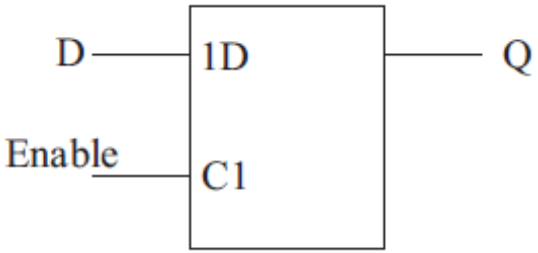
always Statements

- Inside always blocks
  - 1) Sequential statements (intended for Combinational logic)
    - Assignments: blocking statements
    - Decision statements: if, case, ….
    - Loop, …
  - 2) Concurrent statements (intended for Sequential logic)
    - Assignments: non-blocking statements

always_latch

- The always_latch procedural block is very similar to always_comb,
  - except that it documents the designer's intent to represent latch behavior.
- Tools can then verify that this intent is being met.

always_latch



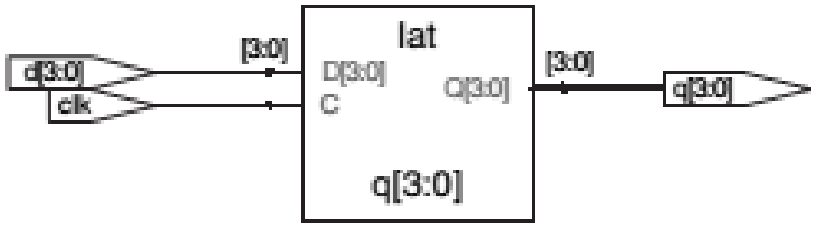

```
module dlatch (output logic q, input logic d, en);
always_latch
    if (en)
        q <= d;
endmodule
```

- The assignment is *nonblocking*.
- Nonblocking assignments (<=) are completed after blocking assignments (=).
- Sequential logic should always be modeled with nonblocking assignments to ensure correct simulation behavior.

always_latch & always @(clk, d)

```
module latch(input logic clk,
             input logic [3:0] d,
             output logic [3:0] q);
always_latch
    if (clk) q <= d;
endmodule
```

- always_latch is equivalent to always @(clk, d) and is the preferred idiom for describing a latch in SV.
- SV can generate a warning if the always_latch block doesn't imply a latch.



- It evaluates any time clk or d changes.
- If clk is 1, d flows through to q, so this code describes a positive level sensitive latch.
- Otherwise, q keeps its old value.

Common Point: always_comb and always_latch

- always_comb and always_latch
  - will execute once at time zero of the simulation,
  - ensuring the variables on the left-hand side of assignments within the block correctly reflect the values on the right-hand side at time 0.

Wrong Usage: always_latch

```
module always_latch_test
    (input a, b, c,
    output logic out);
    always_latch
        if (a) out = b;
        else out = c;
endmodule: always_latch_test
```

- The code uses always_latch, but models combinational logic.
- When this code was read into the synthesis tool,
  - an elaboration warning was issued,
  - noting that no latch was modeled from the always_latch block.

```
Warning:  /test.sv:7: Netlist for always_latch block does not contain a
latch. (ELAB-975)
```

Another Example

```
always_latch
    if (Ctrl)
        Z <= A;
```

```
always @(Ctrl or A)
    if (Ctrl)
        Z <= A;
```

```
always @(Ctrl or A)
    if (Ctrl)
        Z <= A;
    else
        Z <= 1'b0;
```

Behavioral Modeling of Flip-Flops (FFs)

- What is a FF?
  - Nature?
  - Two aspects:
    - Value latched
    - Value Stored

always_ff

```
module flop(input logic clk,
            input logic [3:0] d,
            output logic [3:0] q);
always_ff @(posedge clk)
   q <= d;
endmodule
```

- always_ff behaves like always
  - but is used exclusively to imply flip-flops
  - allows tools to produce a warning if anything else is implied.
- always_ff documents the designer's intent to represent flip-flop behavior.

Sensitivity Lists: Stimulus Lists

```
module flop(input logic clk,
            input logic [3:0] d,
            output logic [3:0] q);
always_ff @(posedge clk)
   q <= d;
endmodule
```

- always_ff differs from always_comb:
  - the sensitivity list must be specified by the designer.
- A SV always statement is written in the form always @(sensitivity list) statement.
- The statement is executed only when the event specified in the sensitivity list occurs.

Nonblocking Assignment

```
module flop(input logic clk,
            input logic [3:0] d,
            output logic [3:0] q);
always_ff @(posedge clk)
    q <= d;
endmodule
```

- The statement is q <= d ("q gets d").
- FF copies d to q on the positive edge of the clock and otherwise remembers the old state of q.
- <= is called a nonblocking assignment.
- <= is used inside an always statement, instead of assign.

SV: always_ff

- The always_ff procedure
  - requires a sensitivity list that specifies a posedge or negedge of a clock,
  - but always_ff enforces many of the synthesis requirements.

More Synthesis Requirements for FFs

1. The procedure sensitivity list must specify which edge of the clock triggers updating the state of the flip-flop (posedge or negedge).
2. The sensitivity list must specify the leading edge (posedge or negedge) of any asynchronous set or reset signals (synchronous sets or resets are not listed in the sensitivity list).
3. Other than the clock, asynchronous set or asynchronous reset, the sensitivity list cannot contain any other signals, such as the D input or an enable input.
4. The procedure should execute in zero simulation time.
5. A variable assigned a value in a sequential logic procedure cannot be assigned a value by any other procedure or continuous assignment (multiple assignments within the same procedure are permitted).
6. A variable assigned a value in a sequential logic procedure cannot have a mix of blocking and nonblocking assignments.

Verilog always: FFs

- always @(posedge clk)
- q <= d;

- Although this example is functionally accurate,
  - the general **always** does not require nor enforce any of the synthesis requirements.

Verilog always: Simulation

- The following example is syntactically legal:
  - always ©(posedge clk or enable)
  - if (enable) q <= d;
- This example will compile and run in simulation with no warnings or error messages.

Verilog always: Synthesis

- The example is syntactically legal, but will not synthesize:
  - always ©(posedge clk or enable)
  - if (enable) q <= d;
- The synthesis compilers will report an error:
  - It does not meet the requirement that no other signals other than the clock and the leading edge of an asynchronous set or reset can be included in the sensitivity list.

- Careful verification reveals that
  - the state of the flip-flop updates its internal storage each time enable changes value, even when no clock trigger occurred.

Practice Guideline

- Use the SystemVerilog always_ff RTL-specific procedure to model RTL sequential logic.
- Do not use the general purpose always procedure.

- The always_ff RTL-specific procedure enforces the coding style required by synthesis compilers in order to correctly model sequential logic behavior.

always_ff: Warning

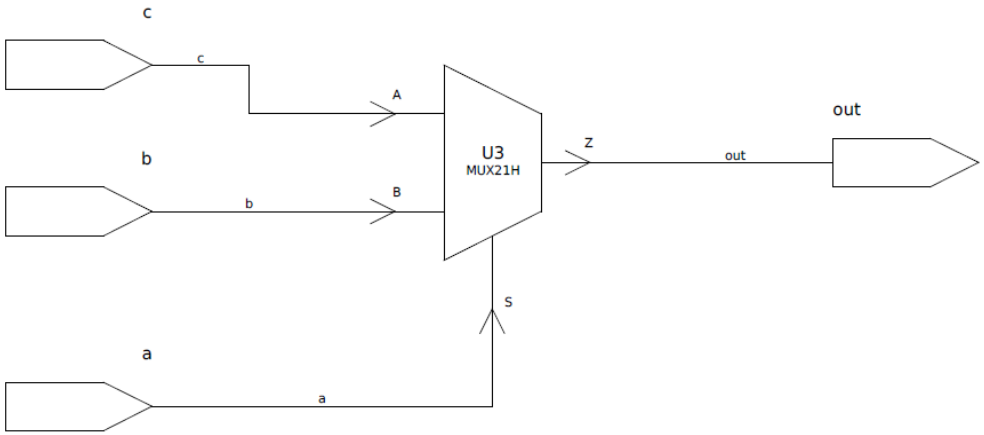
```
module always_ff_test
    (input a, b, c,
    output logic out);
    always_ff @(a, b, c)
        if (a) out = b;
        else out = c;
endmodule: always_ff_test
```

- Tools can verify whether the intent for flip-flop behavior is being met in the body of the procedural block.
- Here always_ff is used for code that does not actually model a flip-flop.
- The warning that is issued by DC follows the example.

```
Warning:  test.sv:5: Netlist for always_ff block does not contain a flip-
flop.  (ELAB-976)
```

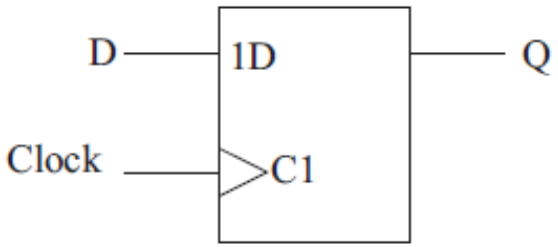
always_ff

```
module always_ff_test
      (input a, b, c,
      output logic out);
      always_ff @(a, b, c)
            if (a) out = b;
            else out = c;
endmodule: always_ff_test
```



More Sequential Systems

Edge-triggered D flip-flop




```
module dff (output logic q, input logic d, clk);
always_ff @(posedge clk)
    q <= d;
endmodule
```

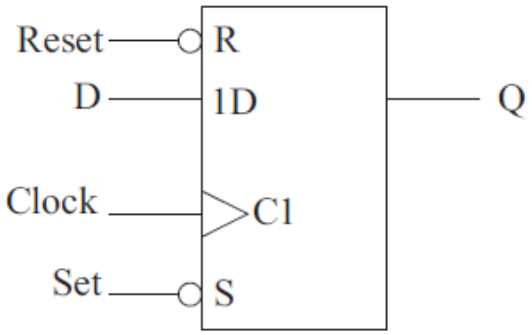
- 1 shows the dependency of D on C.
- Again, a *nonblocking* assignment is used, as this is sequential logic.
- Similarly, a negative edge-triggered flip-flop can be modeled by detecting a transition to logic 0.

Edge-sensitive Flip-flop

```
always_ff @(posedge clk, posedge reset)
    if (reset)
        q <= 1'b0;
    else
        q <= d;
```

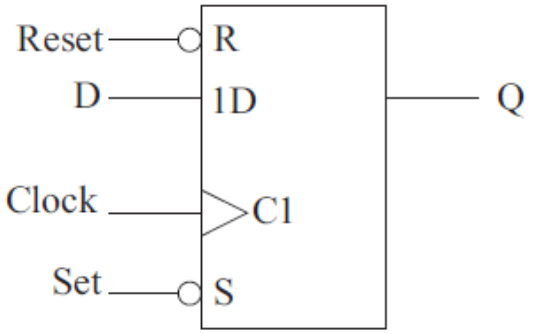
- Asynchronous sets and resets are modeled by including the active edge in the event list.
- The reset is tested before the clock and therefore has an effect irrespective of the clock.
- The clock net to which the flip-flop is edge-sensitive should be tested in the last branch of the if statement.

Asynchronous Set and Reset



- When power is first applied to a flip-flop,
  - its initial state is unpredictable.
- In many applications this is unacceptable,
  - so flip-flops are provided with further inputs to set (or reset) their outputs to 1 or to 0.

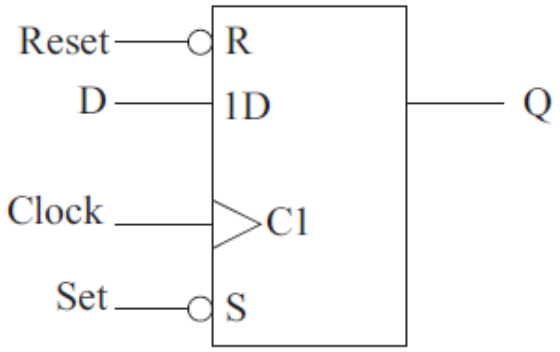
Asynchronous Set and Reset



- In synchronous systems, flip-flops only change state when clocked.

- The absence of any dependency on the clock introduces asynchronous behavior for R and S.
- These inputs should only be used to initialize a flip-flop.
- It is bad practice to use these inputs to set the state of a flip-flop during normal system operation.
- This can lead to all sorts of timing problems.

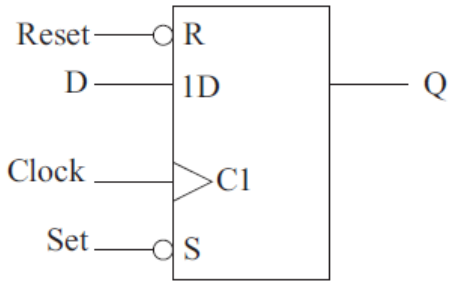
Asynchronous Set and Reset




```
module dffr (output logic q,
                    input logic d, clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
            q <= '0;
        else
            q <= d;
endmodule
```

- A SystemVerilog model of a flip-flop with an asynchronous reset must respond to changes in the clock and in the reset input.

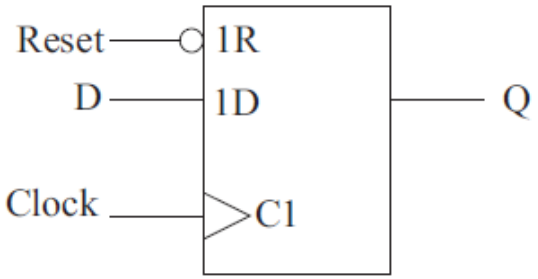
Asynchronous Set and Reset




```
module dffrs (output logic q,
              input logic d, clk, n_reset, n_set);
always_ff @(posedge clk, negedge n_reset,
            negedge n_set)
    if (~n_set)
    q <= '1;
    else if (~n_reset)
    q <= '0;
    else
    q <= d;
endmodule
```

- Have both an asynchronous set and reset.
- asserting both the asynchronous set and reset is usually considered an illegal operation.
- In this model, Q is forced to 1 if Set is 0, regardless of the Reset signal.

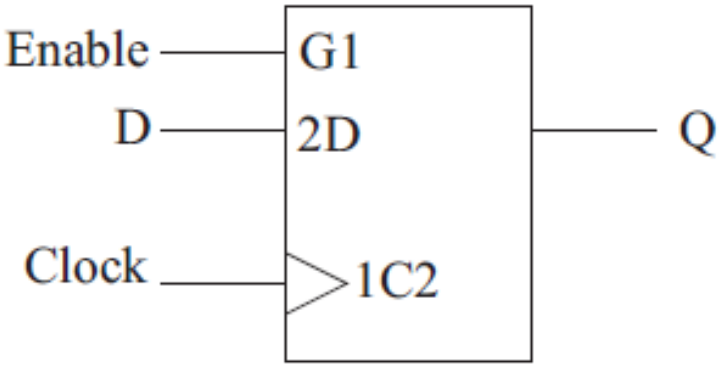
Synchronous Set and Reset




```
module dffsr (output logic q,
input logic d, clk, n_reset);
always_ff @(posedge clk)
    if (~n_reset)
        q <= '0;
    else
        q <= d;
endmodule
```

- R is dependent on C and is therefore synchronous.
- A model must include a check on the set or reset input after the clock edge has been checked.
- The only difference between the synchronous and asynchronous:
  - reset is whether the signal appears in the excitation list of the **always_ff** block.

Synchronous Clock Enable




```
module dffe (output logic q,
             input logic d, clk, enable);
always_ff @(posedge clk)
    if (enable)
    q <= d;
endmodule
```

- Similarly, a flip-flop with a clock enable signal may be modeled with that signal checked after the edge detection.
- C is dependent on G and D is dependent on C.
- Again, the D input is latched if Enable is true and there is a clock edge.

Synchronous Clock Enable

```
module dffe (output logic q,
             input logic d, clk, enable);
always_ff @(posedge clk)
    if (enable)
    q <= d;
endmodule
```

```
module dffce (output logic q,
              input logic d, clk, enable);
logic ce;
always_comb
    ce = enable & clk;
always_ff @(posedge ce)
    q <= d;
endmodule
```

- The D input is also latched if the clock is true and there is a rising edge on the Enable signal !
- This is another example of design that is not truly synchronous and which is therefore liable to timing problems.
- This style of design should generally be avoided,
  - although for low-power applications the ability to turn off the clock inputs to flip-flops can be useful.

Resettable Registers

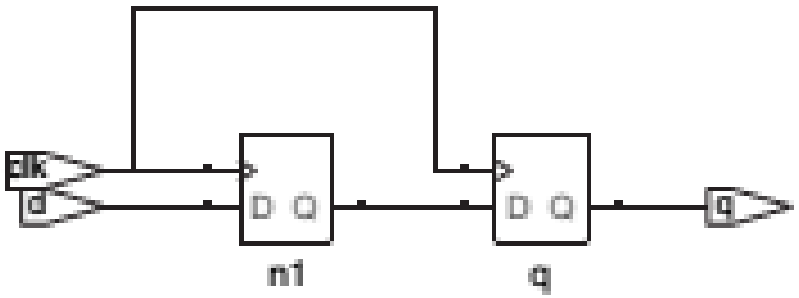
```
module flopr(input logic clk,
             input logic reset,
             input logic [3:0] d,
             output logic [3:0] q);
// asynchronous reset
always_ff @(posedge clk, posedge reset)
if (reset) q <= 4'b0;
else q <= d;
endmodule
module flopr(input logic clk,
             input logic reset,
             input logic [3:0] d,
             output logic [3:0] q);
// synchronous reset
always_ff @(posedge clk)
if (reset) q <= 4'b0;
else q <= d;
endmodule
```

```
module flopr(input logic clk,
             input logic reset,
             input logic [3:0] d,
             output logic [3:0] q);
// asynchronous reset
always_ff @(posedge clk, posedge
reset)
if (reset) q <= 4'b0;
else q <= d;
endmodule
```

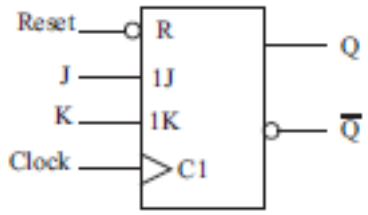
```
module flopr(input logic clk,
             input logic reset,
             input logic [3:0] d,
             output logic [3:0] q);
// synchronous reset
always_ff @(posedge clk)
if (reset) q <= 4'b0;
else q <= d;
endmodule
```

Registers

```
module sync(input logic clk,
            input logic d,
            output logic q);
logic n1;
always_ff @(posedge clk)
begin
   n1 <= d; // nonblocking
   q <= n1; // nonblocking
end
endmodule
```



JK flip-flops

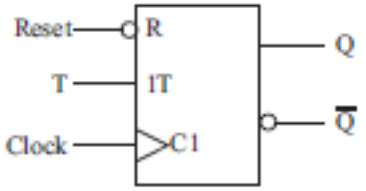


J	K	Q^+	$\bar{Q}^+$
0	0	Q	$\bar{Q}$
0	1	0	1
1	0	1	0
1	1	$\bar{Q}$	Q

```
module jkffr (output logic q, qbar,
              input logic j, k, clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
if (~n_reset)
    {q, qbar} <= {1'b0, 1'b1};
else
    case ({j, k})
        2'b11 : {q, qbar} <= {qbar, q};
        2'b10 : {q, qbar} <= {1'b1, 1'b0};
        2'b01 : {q, qbar} <= {1'b0, 1'b1};
        default:;
endcase
endmodule
```

- A case statement determines the internal state of the JK flip-flop.
- The selector of the case statement is formed by concatenating the J and K inputs.
- The **default** clause covers the '00' case and other undefined values.
- Nothing is done in that clause, so the internal state is retained.

T flip-flops



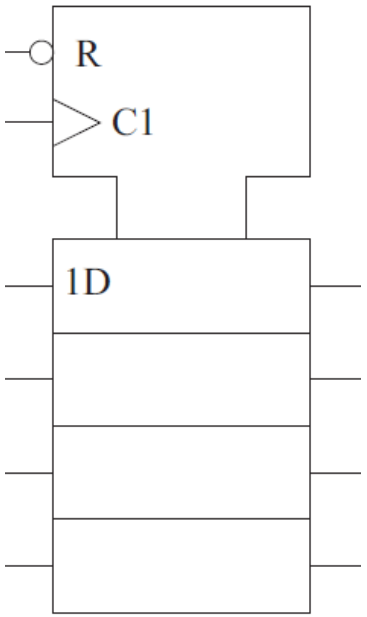
T	Q^+	$\bar{Q}^+$
0	Q	$\bar{Q}$
1	$\bar{Q}$	Q

```
module tffr (output logic q, qbar,
             input logic t, clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
        {q, qbar} <= {1'b0, 1'b1};
        else
        if (t)
        {q, qbar} <= {qbar, q};
endmodule
```

- The internal state of the T flip-flop is retained between activations of the procedural block, if the T input is not set.

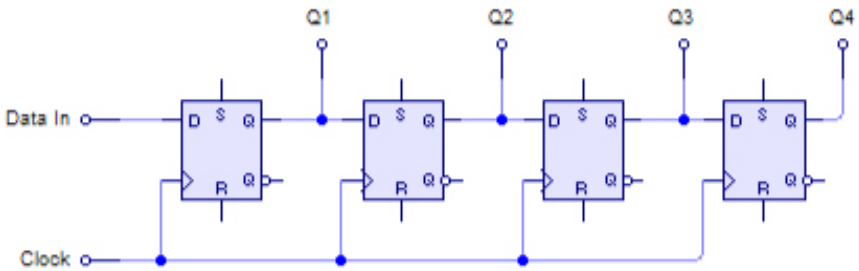
Multiple Bit Register

```
module dffn #(parameter N = 8) (output logic [N-1:0]q,
                                input logic [N-1:0] d,
                                input logic clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
        q <= '0;
    else
        q <= d;
endmodule
```



- The IEEE symbol for a 4-bit register is shown in Figure.
- Note that the common signals are contained in a control block, drawn as a rectangle with the lower corners cut off.

Shift Registers: SIPO

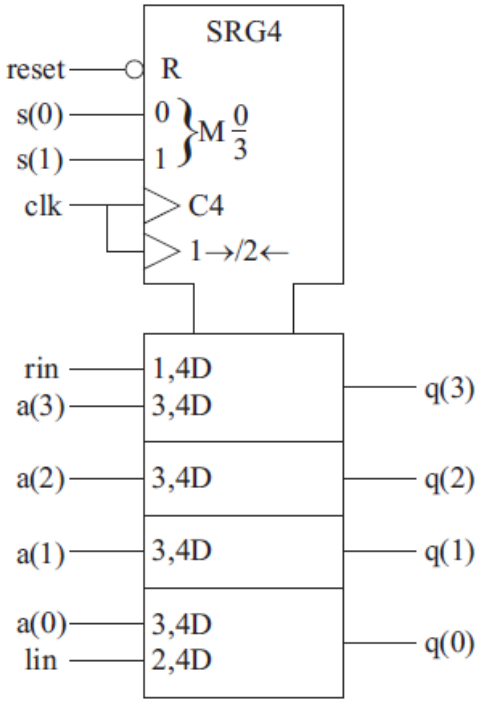



```
module sipo #(parameter N = 8)
                    (output logic [N-1:0] q,
                    input logic a, clk);
always_ff @(posedge clk)
    q <= {q[N-2:0], a};
endmodule
```

- At each clock edge, the bits of the register are moved along by one, and the input, a, is shifted into the 0th bit.
- The assignment does this by assigning bits n–2 to 0 to bits n–1 to 1, respectively, and concatenating a to the end of the assignment.
- The old value for bit n–1 is lost.

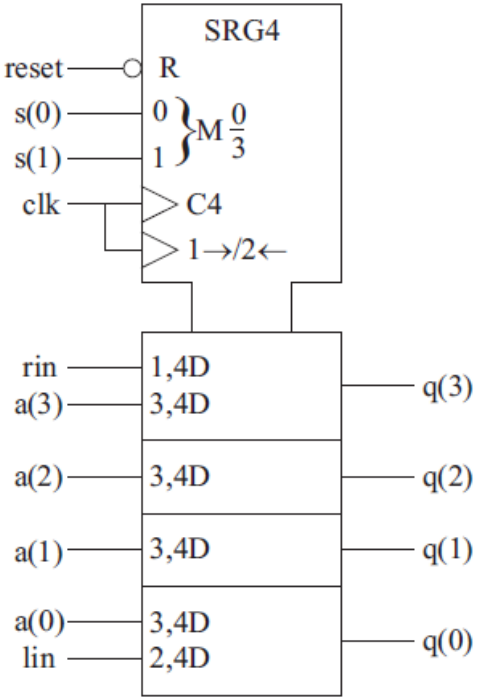
Universal Shift Register

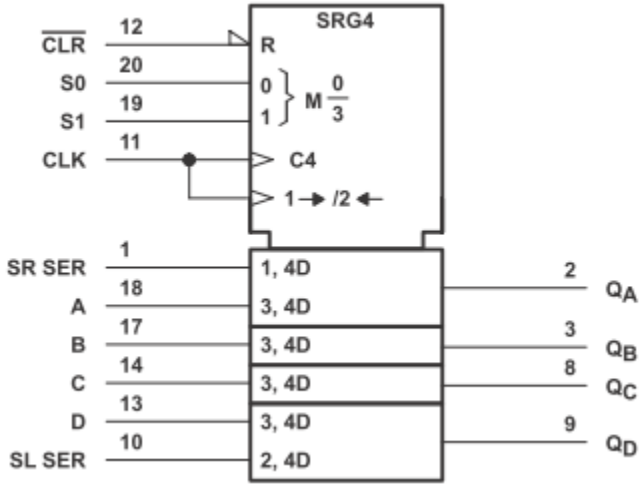
- It can shift bits to the left or to the right, and can load an entire new word in parallel.
- To do this, two control bits are needed.
- There are four control modes shown by M0/3.
- The clock signal is split into two for convenience.
- Control signal 4 is generated and in modes 1 and 2 a shift left or shift right operation, respectively, is performed.
- 1,4D means that a D-type operation occurs in mode 1 when control signal 4 is asserted.



$S_1 S_0$	Action
00	Hold
01	Shift right
10	Shift left
11	Parallel load

Universal Shift Register (TI: 74AC11194)



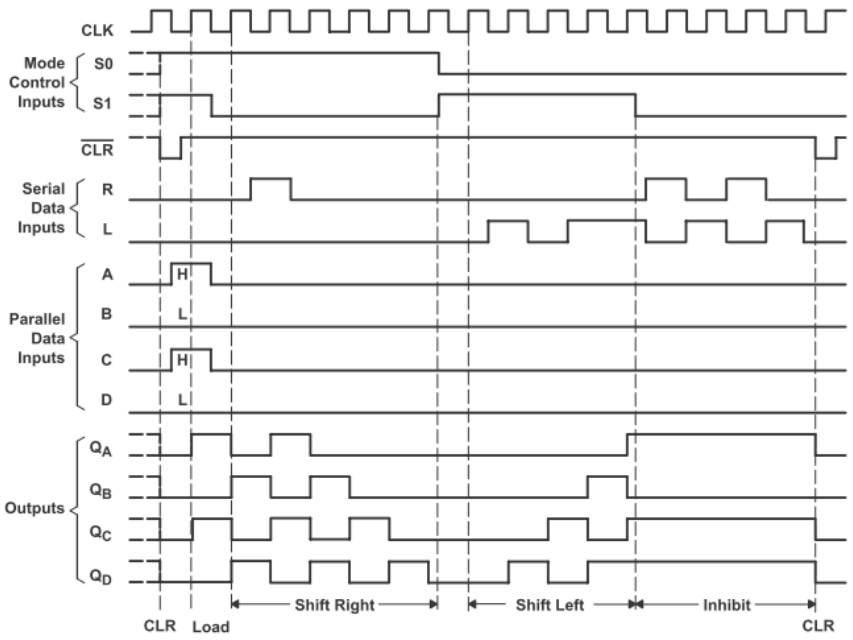


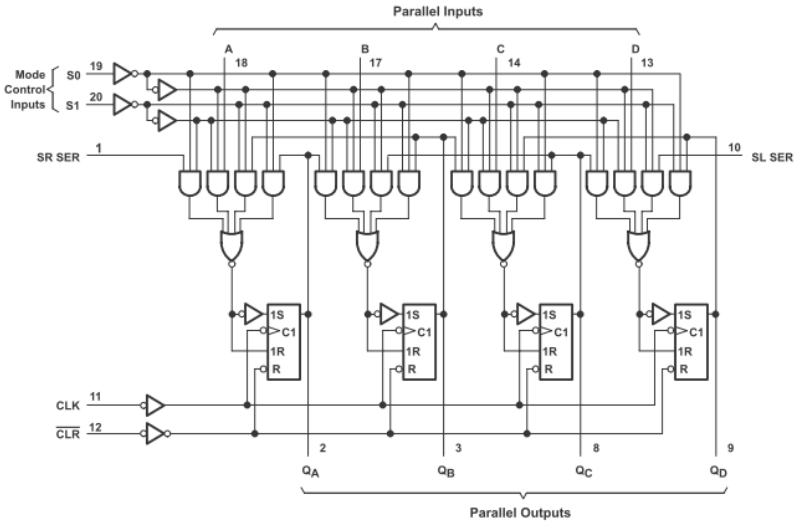
S_1 S_0	Action
00	Hold
01	Shift right
10	Shift left
11	Parallel load

INPUTS										OUTPUTS			
$\overline{\text{CLEAR}}$	MODE		CLOCK	SERIAL		PARALLEL				Q_A	Q_B	Q_C	Q_D
	S1	S0		LEFT	RIGHT	A	B	C	D				
L	X	X	X	X	X	X	X	X	X	L	L	L	L
H	X	X	L	X	X	X	X	X	X	Q_{A0}	Q_{B0}	Q_{C0}	Q_{D0}
H	H	H	↑	X	X	a	b	c	d	a	b	c	d
H	L	H	↑	X	H	X	X	X	X	H	Q_{An}	Q_{Bn}	Q_{Cn}
H	L	H	↑	X	L	X	X	X	X	L	Q_{An}	Q_{Bn}	Q_{Cn}
H	H	L	↑	H	X	X	X	X	X	Q_{Bn}	Q_{Cn}	Q_{Dn}	H
H	H	L	↑	L	X	X	X	X	X	Q_{Bn}	Q_{Cn}	Q_{Dn}	L
H	L	L	X	X	X	X	X	X	X	Q_{AO}	Q_{BO}	Q_{CO}	Q_{DO}

Universal Shift Register (TI: 74AC11194)

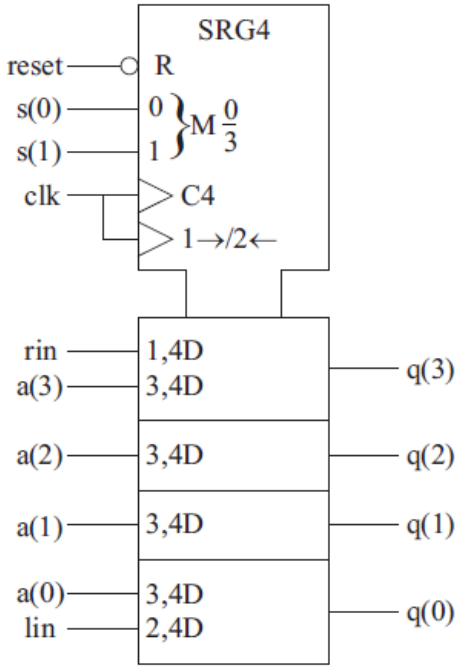
INPUTS										OUTPUTS			
$\overline{CLEAR}$	MODE		CLOCK	SERIAL		PARALLEL				Q_A	Q_B	Q_C	Q_D
	S1	S0		LEFT	RIGHT	A	B	C	D				
L	X	X	X	X	X	X	X	X	X	L	L	L	L
H	X	X	L	X	X	X	X	X	X	Q_{A0}	Q_{B0}	Q_{C0}	Q_{D0}
H	H	H	↑	X	X	a	b	c	d	a	b	c	d
H	L	H	↑	X	H	X	X	X	X	H	Q_{An}	Q_{Bn}	Q_{Cn}
H	L	H	↑	X	L	X	X	X	X	L	Q_{An}	Q_{Bn}	Q_{Cn}
H	H	L	↑	H	X	X	X	X	X	Q_{Bn}	Q_{Cn}	Q_{Dn}	H
H	H	L	↑	L	X	X	X	X	X	Q_{Bn}	Q_{Cn}	Q_{Dn}	L
H	L	L	X	X	X	X	X	X	X	Q_{AO}	Q_{BO}	Q_{CO}	Q_{DO}





Universal Shift Register

```
module usr #(parameter N = 8) (output logic [N-1:0]q,
          input logic [N-1:0] a, input logic [1:0] s,
          input logic lin, rin, clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
     if (~n_reset)
          q <= '0;
          else
               case (s)
                    2'b11: q <= a;
                    2'b10: q <= {q[n-2:0], lin};
                    2'b01: q <= {rin, q[n-1:1]};
                    default:;
               endcase
endmodule
```



$S_1 S_0$	Action
00	Hold
01	Shift right
10	Shift left
11	Parallel load

Counters

- Counters are used for a number of functions in digital design, e.g.
  - counting the number of occurrences of an event;
    - storing the address of the current instruction in a program; or
    - generating test data.
- Although a counter typically starts at zero and increments monotonically to some larger value,
  - it is also possible to use different sequences of values, which can result in simpler combinational logic.

Binary Counter

```
module bincounter #(parameter N = 8)
                   (output logic [N-1:0] count,
                   input logic n_reset, clk);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
        count <= 0;
    else
        count <= count + 1;
endmodule
```

- A binary counter consists of a register of a number of D flip-flops, the content of which is the binary representation of a decimal number.
- At each clock edge the contents of the counter is increased by one.
- We can easily model this in SystemVerilog, using the '+' operator.

Binary Counter

```
module bincounter #(parameter N = 8)
                   (output logic [N-1:0] count,
                   input logic n_reset, clk);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
        count <= 0;
    else
        count <= count + 1;
endmodule
```

- Note that the + operator does not generate a carry out.
- Thus when the counter has reached its maximum integer value (all 1s) the next clock edge will cause the counter to 'wrap round' and its next value will be zero (all 0s).

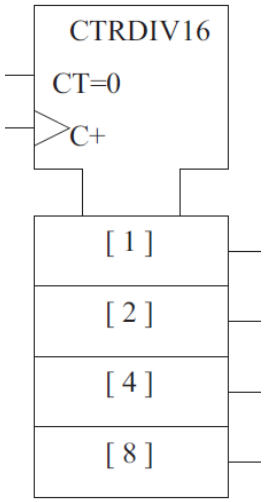
- We could modify the counter to generate a carry out, but in general counters are usually designed to detect the all-1s state and to output a signal when that state is reached.
- A carry out signal would be generated one clock cycle later.

Binary Counter

```
module bincounter #(parameter N = 8)
                   (output logic [N-1:0] count,
                   input logic n_reset, clk);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
    count <= 0;
    else
    count <= count + 1;
endmodule
```

- It would be incorrect to use the increment operator, "++"
  - for example by writing count++; instead of the assignment.
- Although more concise, the increment is a blocking assignment (equivalent to count = count + 1;).
- Using blocking assignments in sequential logic can cause erroneous simulated behavior.

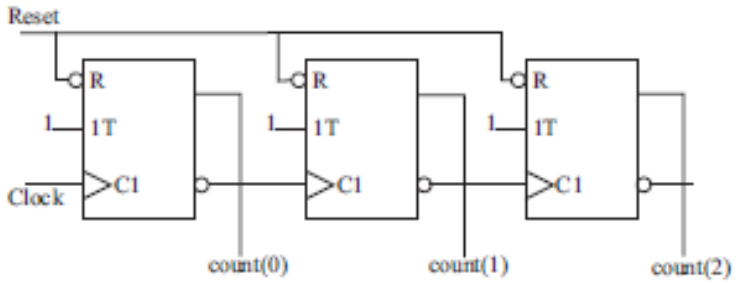
Binary Counter



- The advantage of describing a counter behaviorally is that the underlying combinational next state logic is hidden.
- For a counter with eight or more bits, the combinational logic can be very complex, but a synthesis system will generate that logic automatically.

```
module bincounter #(parameter N = 8)
                   (output logic [N-1:0] count,
                   input logic n_reset, clk);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
    count <= 0;
    else
    count <= count + 1;
endmodule
```

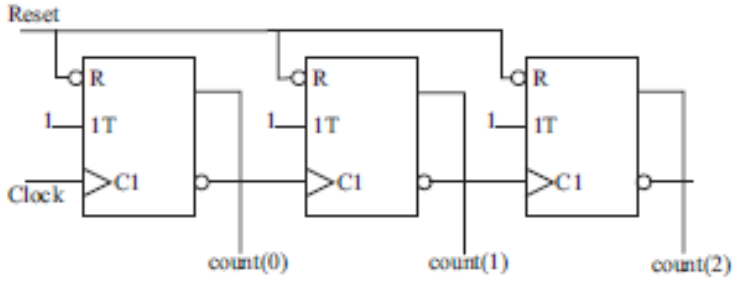
The Ripple Counter




```
module ripple_counter #(parameter N = 8)
            (output logic [N-1:0] count,
            input logic n_reset, clk);
logic [N:1] Ca;
genvar i;
tffr t0 (count[0], Ca[1], '1, clk, n_reset);
generate for (i = 1; i < N; i++)
      begin : t_loop
      tffr ti (count[i], Ca[i+1], '1, Ca[i], n_reset);
      end
endgenerate
endmodule
```

- A simpler form of binary counter is the ripple counter.
- An example of a ripple counter using T flip-flops is described in SystemVerilog.
- Note that the T input is held at a constant value in the description.
- When simulated using the T flip-flop model, above, this circuit behaves identically to the RTL model.

The Ripple Counter

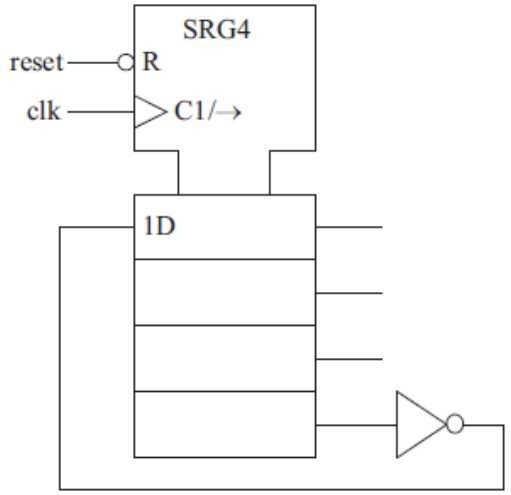


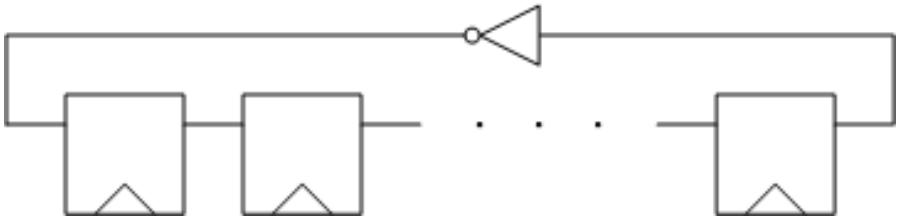

```
module ripple_counter #(parameter N = 8)
            (output logic [N-1:0] count,
            input logic n_reset, clk);
logic [N:1] Ca;
genvar i;
tffr t0 (count[0], Ca[1], '1, clk, n_reset);
generate for (i = 1; i < N; i++)
      begin : t_loop
      tffr ti (count[i], Ca[i+1], '1, Ca[i], n_reset);
      end
endgenerate
endmodule
```

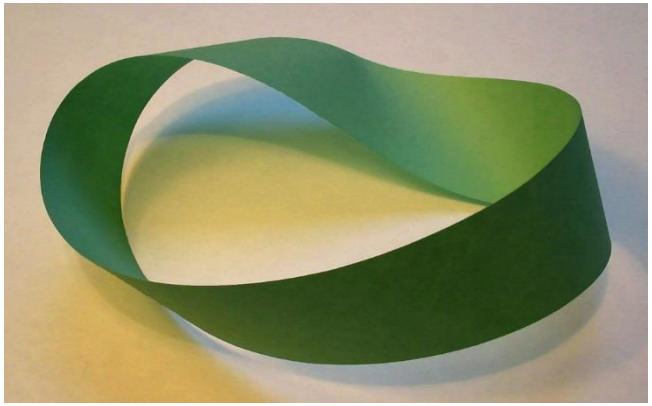
- The ripple counter is asynchronous.
- The second FF is clocked from the Q output of the first FF.
- A change in this output is delayed relative to the clock.
- Hence, the second FF is clocked by a signal behind the true clock.
- With further counter stages, the delay is increased.
- Provided the clock speed is sufficiently slow, a ripple counter can be used instead of a synchronous counter, but in many applications a synchronous counter is preferred.

Johnson Counter

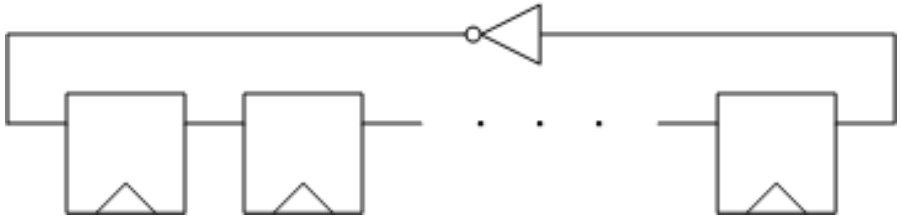
- A Johnson counter
  - known as a Mobius counter
    - after a Mobius strip: a strip of paper formed into a circle with a single twist, resulting in a single surface
  - is built from a shift register with the least significant bit inverted and fed back to the most significant bit.







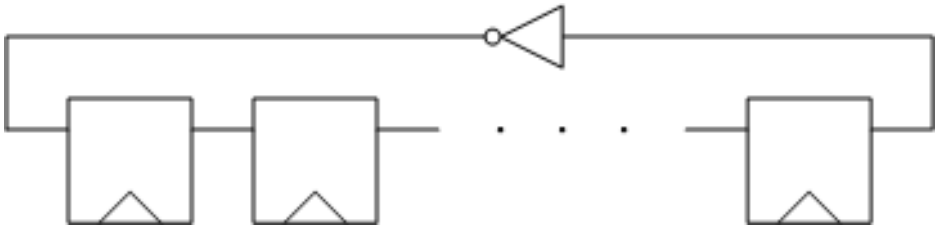
Johnson Counter



- Since each time only a single bit is changing – Johnson counter states form a sort of a Gray code. The next picture shows the 12 states of a 6 bit Johnson counter as an example.
- An n-bit binary counter has 2^n states.
- An n-bit Johnson counter has 2n states.
- The advantage of a Johnson counter is that it is simple to build (like a ripple counter), but is synchronous.

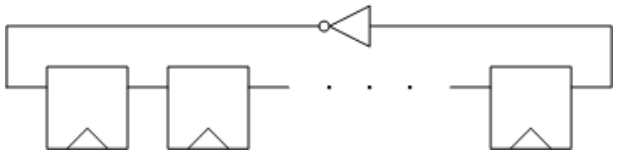
000000
100000
110000
111000
111100
111110
111111
011111
001111
000111
000011
000001

Johnson Counter



```
module johnson #(parameter N = 8)
                (output logic [N-1:0] q,
                input logic clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
        q <= '0;
    else
        q <= {~q[0], q[N-1:1]};
endmodule
```

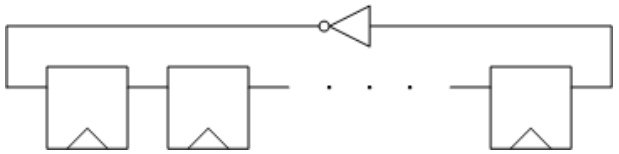
Johnson Counter: Hang States



Normal counting sequence	Parasitic counting sequence
0000	0010
1000	1001
1100	0100
1110	1010
1111	1101
0111	0110
0011	1011
0001	0101

- The disadvantage is the large number of unused states that form an autonomous counter in their own right.
  - We have the intended counter and a *parasitic* state machine coexisting in the same hardware.
- The unused states form a single parasitic counter with 2^n-2n states.
- Both sequences repeat but do not intersect at any point.

Johnson Counter: Hang States



Normal counting sequence	Parasitic counting sequence
0000	0010
1000	1001
1100	0100
1110	1010
1111	1101
0111	0110
0011	1011
0001	0101

- The parasitic set of states of a Johnson counter should never occur,
  - but if one of the states did occur somehow, perhaps because of a power supply glitch or because of some asynchronous input,
  - the system can never return to its normal sequence.
- The subsequent behavior might be unexpected.

Partial Resolution

Normal counting sequence	Parasitic counting sequence
0000	0010
1000	1001
1100	0100
1110	1010
1111	1101
0111	0110
0011	1011
0001	0101

```
module scjohnson #(parameter N = 8)
        (output logic [N-1:0] q,
        input logic clk, n_reset);
always_ff @(posedge clk, negedge n_reset)
    if (~n_reset)
        q <= '0;
    else
    if (~q[N-1] & ~q[0])
        q <= {1'b1, {(N-1){1'b0}}};
    else
        q <= {~q[0], q[N-1:1]};
endmodule
```

- To make the counter self-correcting, detect every one of the parasitic states and to force a synchronous reset
- But for an n-bit counter, it is difficult.
- Easy solution:
  - The only legal state with a 0 in both the most significant and least significant bits is the all zeros state.
  - Three of the parasitic states have zeros in those positions.

FF Assignments

- Synthesis compiler will infer FFs for each variable that is assigned with a nonblocking statement.
- Blocking statements might also infer FFs, depending on the order and context of the assignment statement relative to other assignments and operations in the procedure.

References

- RTL Modeling with SystemVerilog for Simulation and Synthesis by Stuart Sutherland
- Digital System Design with SystemVerilog by Mark Zwolinski. Prentice Hall, 2009.
- Language Reference Manual (LRM) – IEEE 1800-2005
- "Synthesizing SystemVerilog" by S. Sutherland, D. Mills. 2013.
- Sutherland, "SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes", Synopsys Users Group (SNUG) conference, San Jose, California, March 2005.
- Mills and Sutherland, "SystemVerilog Assertions Are For Design Engineers Too!", Synopsys Users Group (SNUG) conference, San Jose, California, March 2006.
- Mills, "Being assertive with your X (SystemVerilog assertions for dummies)", Synopsys Users Group Conference (SNUG) San Jose, California, 2004.
- Mills, "Yet Another Latch and Gotchas Paper", Synopsys Users Group (SNUG) conference, San Jose, California, March 2012.
- Greene, Salz and Booth, "X-Optimism Elimination during RTL Verification", Synopsys Users Group Conference (SNUG) San Jose, 2012.