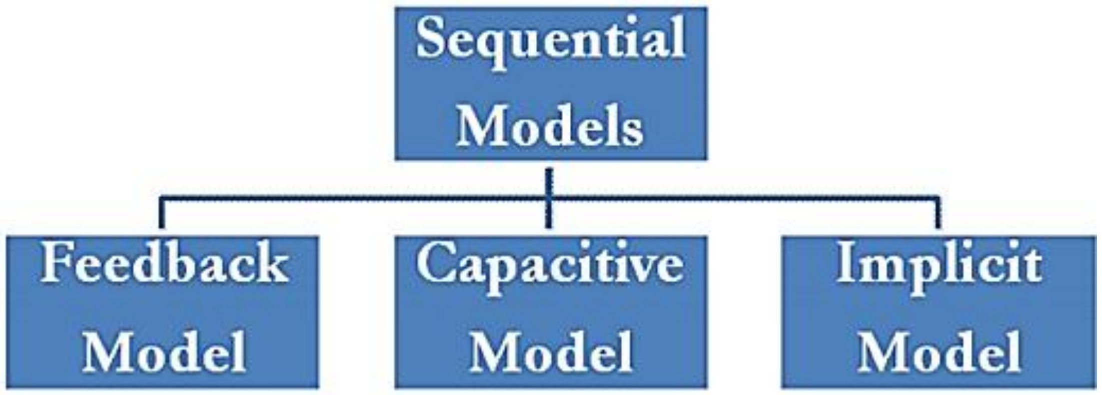
MODULE V

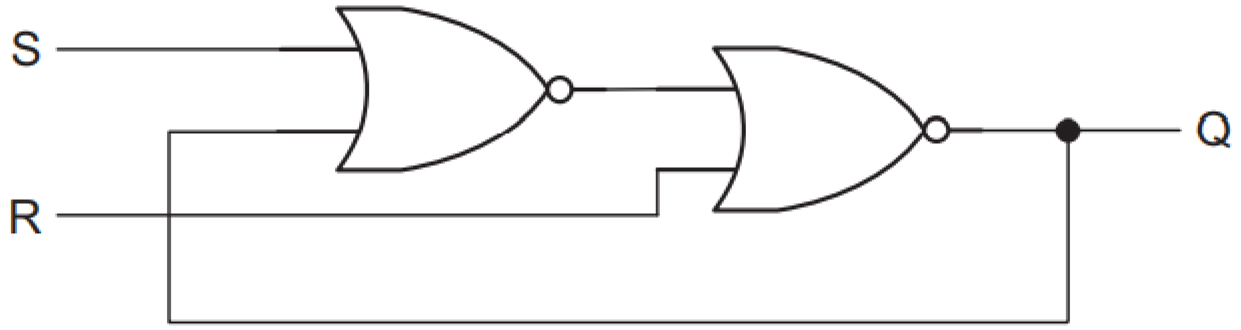
SEQUENTIAL CIRCUIT DESCRIPTION

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.



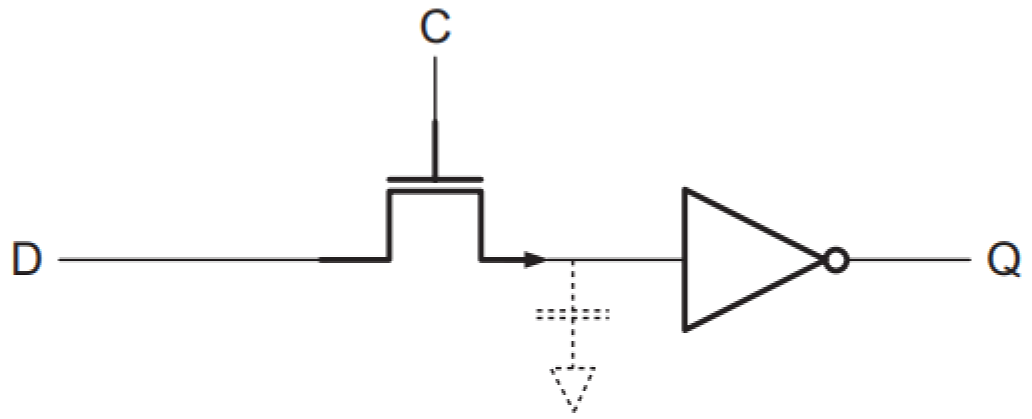
1. Feedback model



Basic Feedback

- The construct shown in 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.

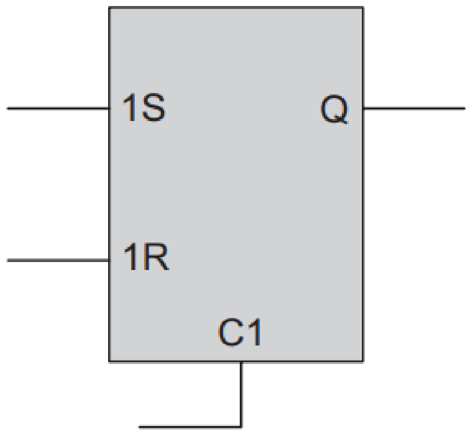
2. Capacitive model



Capacitive Storage

- 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.

3. Implicit model

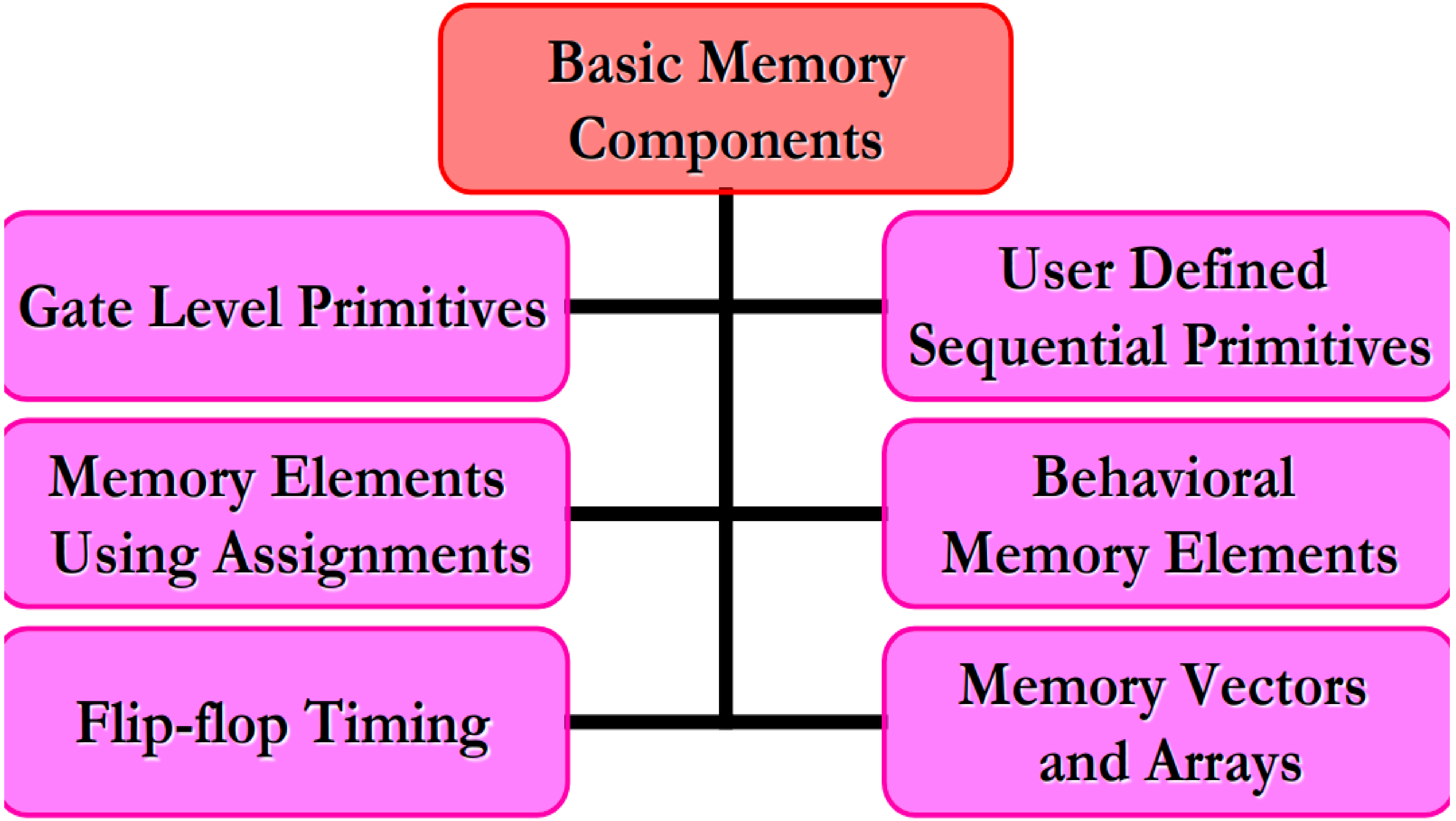


An SR-Latch Notation

- Figure shows an 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.

- Feedback and capacitive models are technology dependent, and they have the problem of being too detailed and thus too slow to simulate.
- Of course, where such details are needed, this level of modeling is possible in Verilog.
- Verilog also 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 number of storage elements.

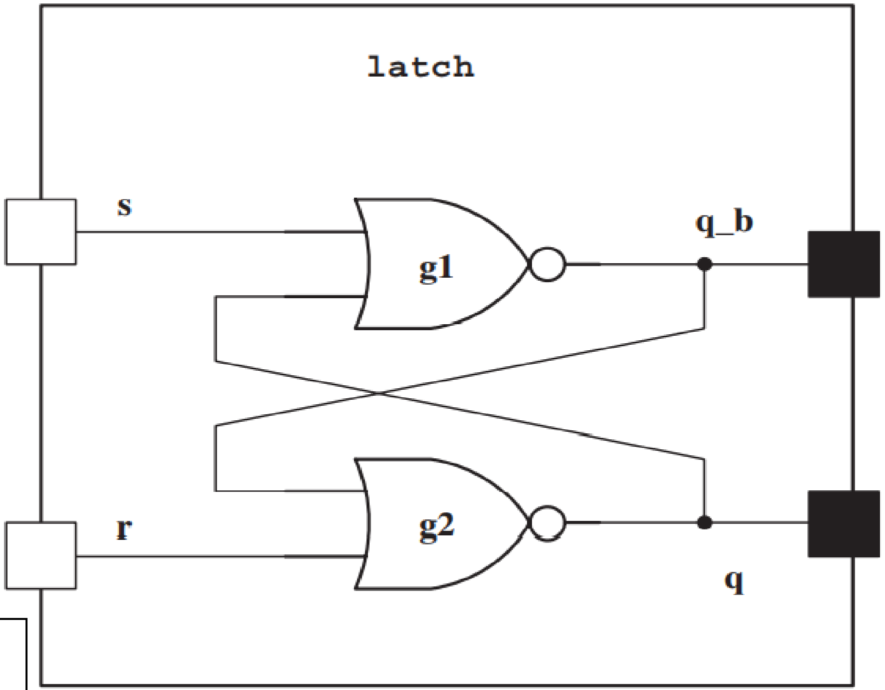
BASIC MEMORY COMPONENTS


Basic Memory Components
Gate Level Primitives
User Defined Sequential Primitives
Memory Elements Using Assignments
Behavioral Memory Elements
Flip-flop Timing
Memory Vectors and Arrays

1. Gate level primitives

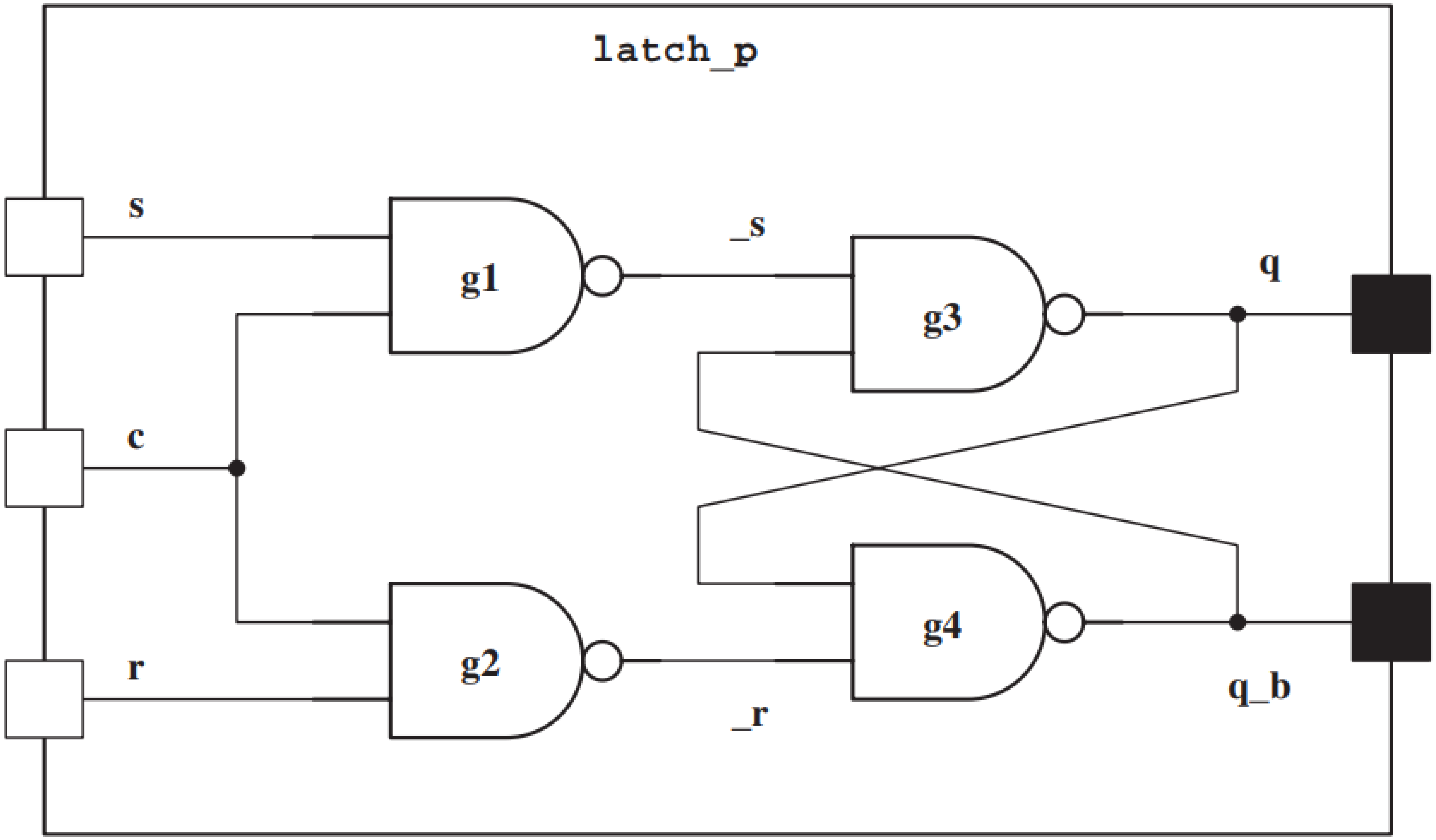
- *Figure shows a cross-coupled NOR structure that forms a 1-bit storage element.*
- *This circuit is no different than that feedback model, and its storage is due to the feedback from q back to g1*

```
`timescale 1ns/100ps

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



Cross-Coupled NOR Latch

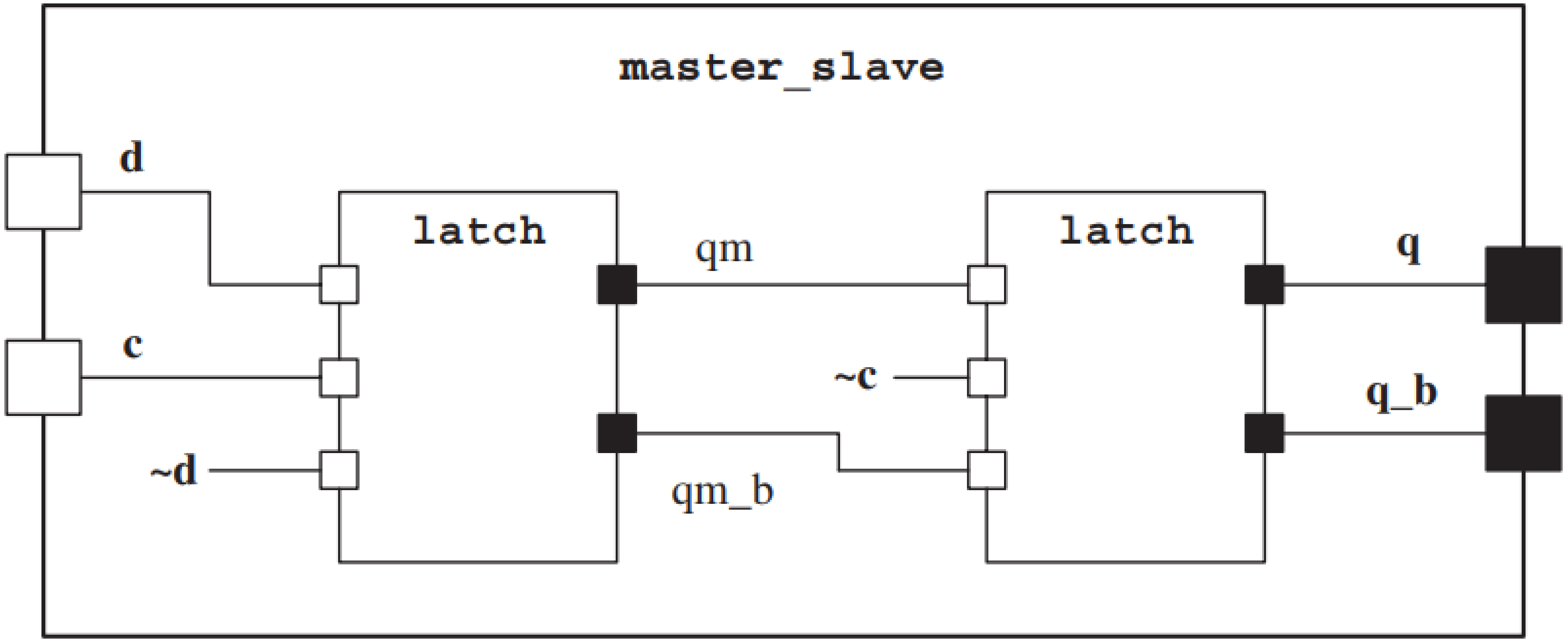
All NAND Clocked SR-Latch



- The Verilog code that corresponds to the diagram is shown below. As shown, the circuit is parameterized so that delay values can be controlled when the latch is instantiated.
- We have declared our parameters in the module header along with module name and ports.
- Wire names _s and _r are used for the set and reset inputs to the crosscoupled core of this memory element.

```
`timescale 1ns/100ps

module latch_p #(parameter tplh=3, tphl=5) (input s, r, c,
                 output q, q_b );
   wire _s, _r;
   nand #(tplh,tphl)
        g1 ( _s, s, c ),
        g2 ( _r, r, c ),
        g3 ( q, _s, q_b ),
        g4 ( q_b, _r, q );
endmodule
```

Master-Slave D Flip-Flop

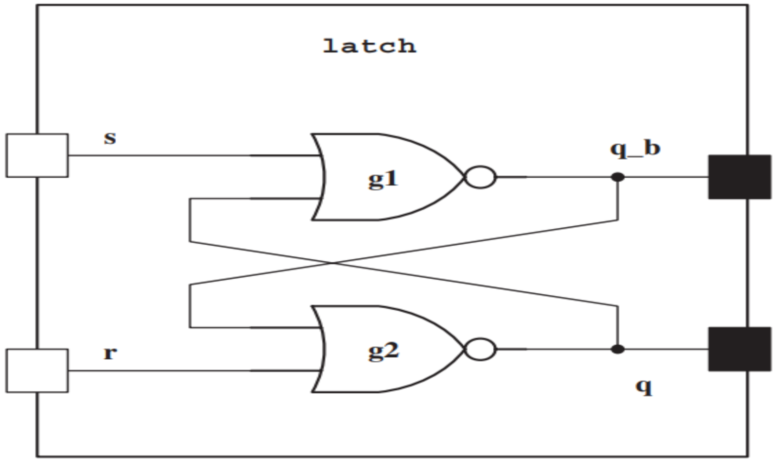


```
`timescale 1ns/100ps

module master_slave (input d, c, output q, q_b );
   wire qm, qm_b;
   defparam master.tplh=4, master.tphl=4, slave.tplh=4,
            slave.tphl=4;
   latch_p
      master ( d, ~d, c, qm, qm_b ),
      slave  ( qm, qm_b, ~c, q, q_b );
endmodule
```

2. User defined sequential primitives

- For faster simulation of memory elements and for correspondence with specific component libraries, Verilog provides language constructs for defining sequential user-defined primitives (UDPs).
- A sequential UDP has the format of the combinational UDP, except that in addition to the inputs, and output of the circuit, its present state is also specified.

Sequential UDP Defining a Latch



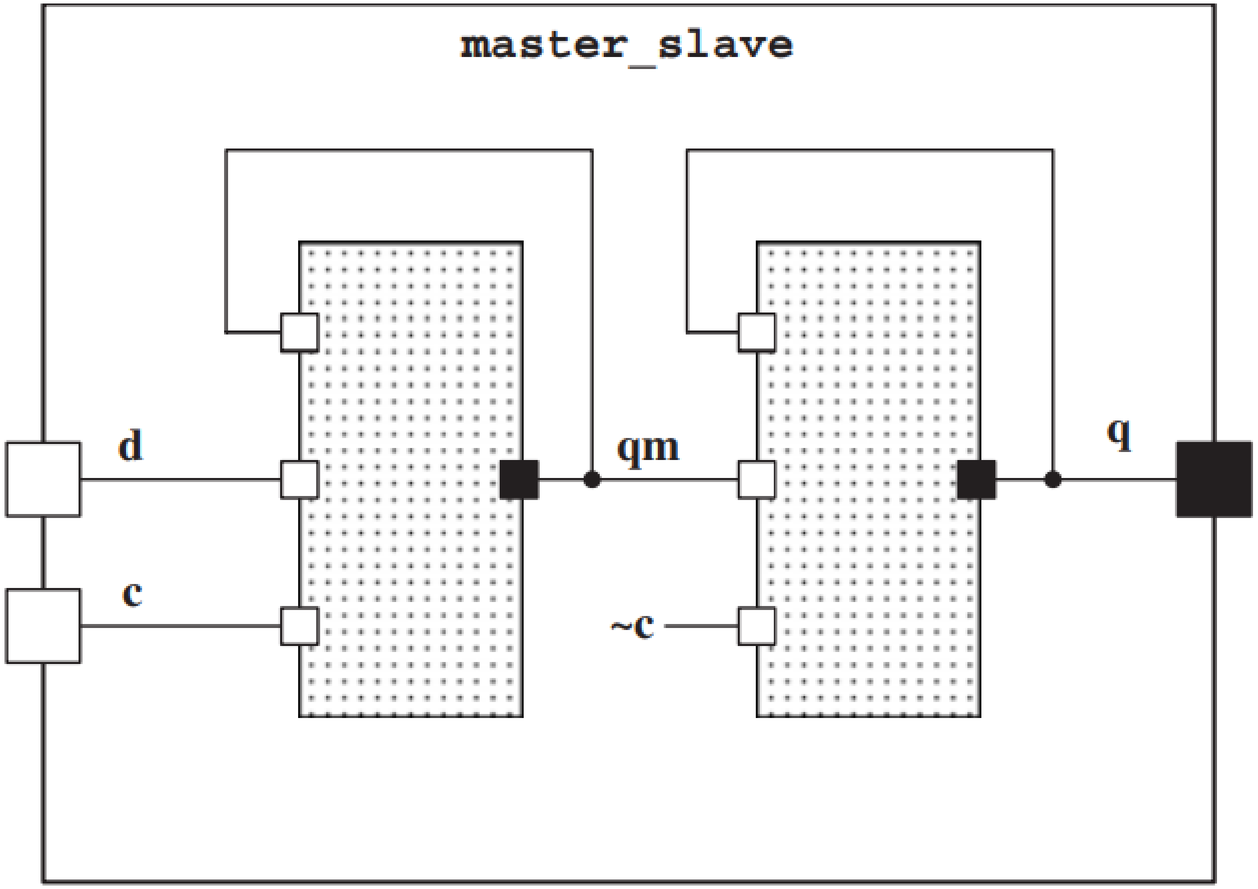
```
primitive latch( q, s, r, c );
    output q;
    reg q;
    input s, r, c;
    initial q=1'b0;
    table
        // s r c    q    q+;
        // ------:----:----;
           ? ? 0  :  ?  :  -  ;
           0 0 1  :  ?  :  -  ;
           0 1 1  :  ?  :  0  ;
           1 0 1  :  ?  :  1  ;
    endtable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

- The table defining the latch output has a column for specifying its present state.
- This column comes before the output specification column and is separated from inputs and output by colons.
- Question marks (?) in the table signify "any value" and dashes (-) are for "no change."
- For example the first row of the table of primitive latch reads as: any value on s, r and the present sate (q), for as long as c is 0, keeps the next state of the machine (q+) unchanged.
- 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

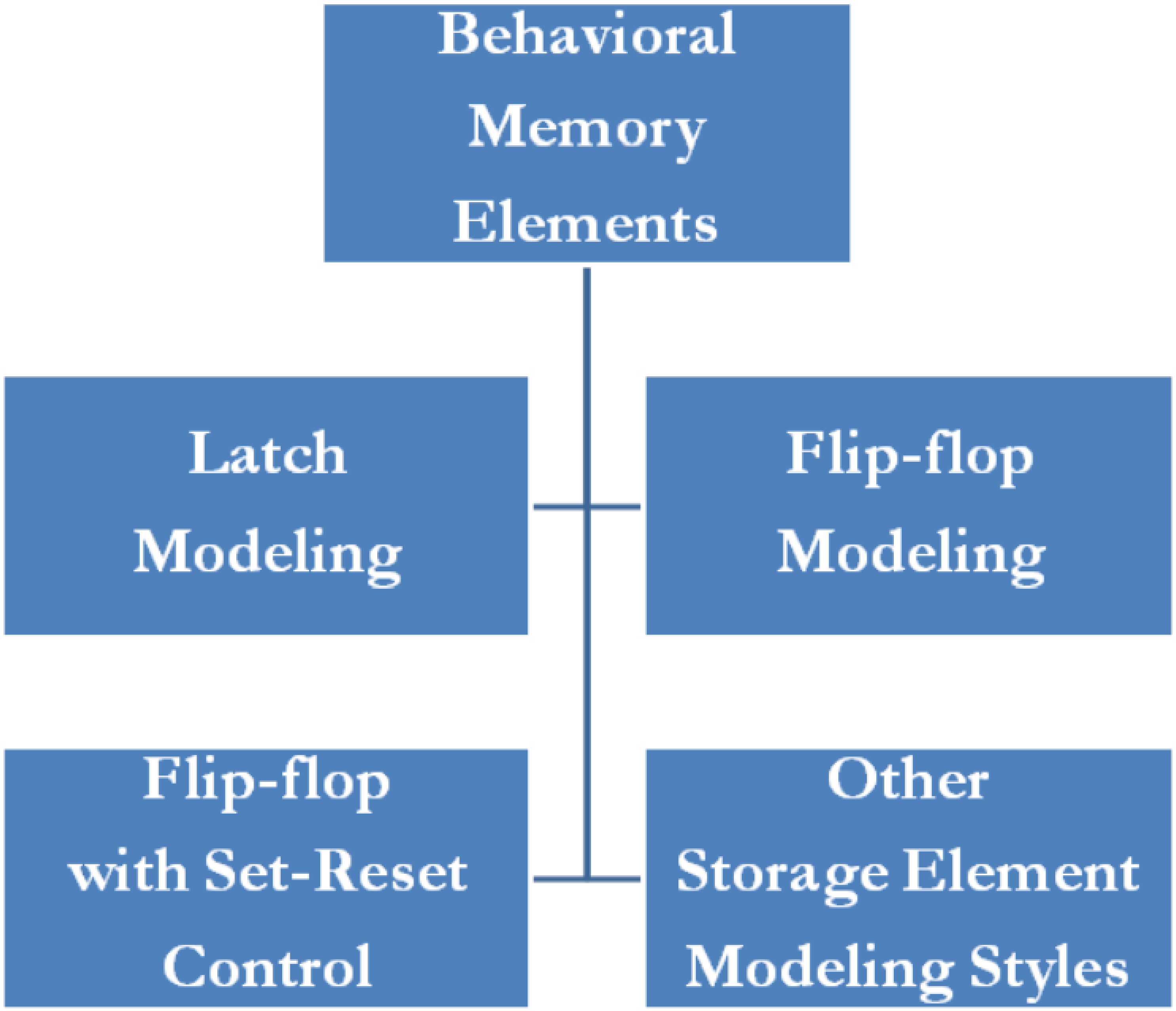
- As a continuous assignment is equivalent to a gate structure driving the left-hand side of the assignment. We can use these statements for specifying specific gates of a hardware module for a latch, or for specifying a feedback circuit.
- Figure below shows 2 feedback blocks forming a master-slave flipflop. Each block has 3 inputs and 1 output. When a block's clock input is 0, it put its output back to itself (feedback), and when its clock is 1 it puts its data input into its output.
- Verilog code uses assign statements to implement feedbacks. In the 1st assignment the use of qm on the right and left of the assign statement corresponds to the feedback of this output back to its input. Each assign statement implements a latch, and the module that uses two latches with complementary clocks implements a master-slave flip-flop

```
`timescale 1ns/100ps

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



4. Behavioral memory elements

- The previous sections showed Verilog models for latches and flip-flops by explicit use of feedback or present and next states.
- Such a model corresponds to the actual hardware implementing a memory element, and has the potential of having all gate level delays specified.
- A more abstract and easier way of writing Verilog code for a latch or flip-flop is by behavioral coding.
- This way, the storage of data and its sensitivity to its clock and other control inputs will be implied in the way model is written.


Behavioral Memory Elements
Latch Modeling
Flip-flop Modeling
Flip-flop with Set-Reset Control
Other Storage Element Modeling Styles

a) Latch modeling

- As our first behavioral model of a memory element consider the Verilog code of the D latch .
- This code reads as: when c or d changes, if c is 1, q gets d. This means that if c or d changes and c is not 1, then q does not change and it retains its old value

```
module latch (input d, c, output reg q, q_b );
    always @( c or d )
        if ( c ) begin
            #4 q = d;
            #3 q_b = ~d;
        end
endmodule
```

b. Flip-flop modeling

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

- A basic edge trigger flip-flop model at the behavioral level is shown above
- This model is sensitive to the positive edge of the clock, and uses nonblocking assignments for assignments to q and q_b.
- Flow into the procedural block is controlled by the event control statement that has posedge clk as its event expression..

- Assignments to q and q_b are reached immediately after the flow in the always block begins. As shown, the actual assignment of values to q and q_b are delayed by 4 and 3 ns, respectively.
- With each clock edge, the entire procedural block is executed once from begin to end.

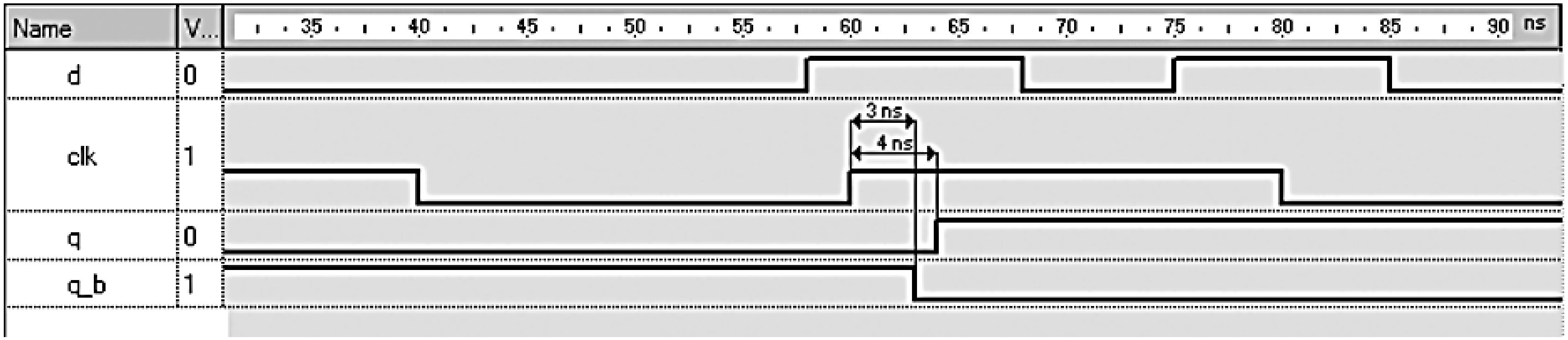


Fig: A **partial waveform of a simulation run of our D flipflop.**

- At 60 ns when we have the positive edge of the clock, the value of d is read and scheduled into q and q_b for times 64 and 63 ns, respectively.
- The sensitivity to the positive edge of the clock is illustrated by the fact that during the time that clk is 1 (from 60 to 80 ns, exclusive of 60 ns, and inclusive of 80 ns), changes on d do not affect the state of flip-flop

c. Flip-flop with set-reset control

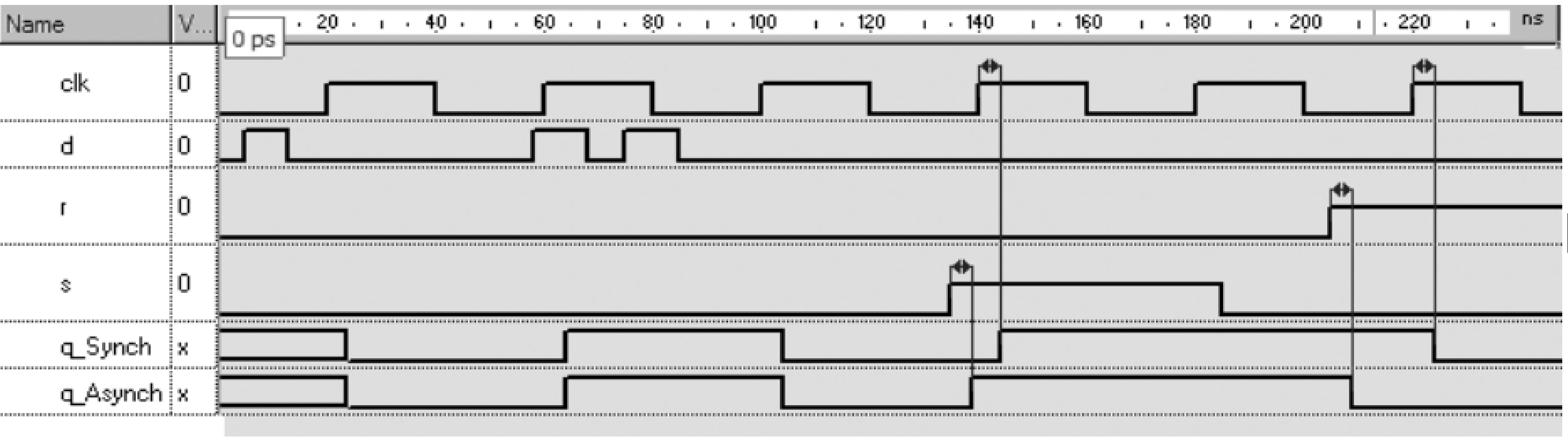
Verilog code of D-type flip-flop with synchronous set and reset (s and r) :

```
module d_ff_sr_Synch (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
```

- *As shown in this figure, a single always statement is used for describing the d_ff_sr_Synch module.*
- *The flow into the always block is only initiated by the posedge of clk.*
- *Therefore, the if-statements with s and r conditions are only examined after the positive edge of the clock.*
- *This behavior is in accordance with synchronicity of s and r control inputs.*

D-type Flip-Flop with Asynchronous Control

```
module d_ff_sr_Asynch (input d, s, r, clk, output reg q, q_b );
          always @( posedge clk, posedge s, posedge r )
          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
```



- Figure shows output waveforms of d_ff_sr_Synch and d_ff_sr_Asynch for data applied to d, s, and r.
- In the first half of this waveform (before 120 ns), changes to q are triggered by the clock and q_Synch and q_Asynch are exactly the same.
- In the second half of this waveform, s and r become active and cause changes to the flip-flop output.
- Note that q_Synch still waits for the edge of the clock to set or reset, while q_Asynch changes occur independent of clk when s or r becomes active.

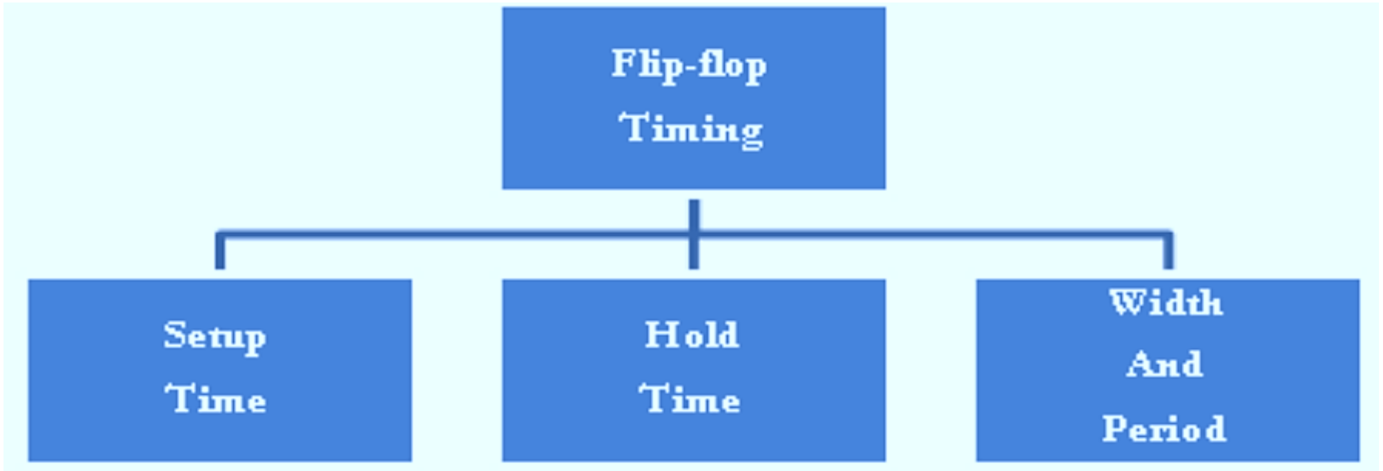
d. Other storage element modeling styles

- Models presented above are the most commonly used models for latches and flip-flops. Verilog provides other language constructs that can be used for this purpose,
- For example, a latch using a wait statement instead of an event control statement. The code shown models a positive level sensitive D-type latch

```
module latch (input d, c, output reg q, q_b );

    always begin
        wait ( c );
            #4 q <= d;
            #3 q_b <= ~d;
    end
endmodule
```

5. Flip-Flop timing

- In behavioral modeling of flip-flops only allows a limited timing specification. Furthermore, the body of a procedural statement, where a flip-flop behavior is described, is not an appropriate place for checking for inappropriate timings of flip-flop inputs and clock.
- Here, Verilog language constructs for detecting and reporting timing violations. Such constructs include system tasks for checking setup, hold, period, and width parameters



a) Set up time

- Setup time is defined as the minimum necessary time that a data input requires to setup before it is clocked into a flipflop.
- Verilog construct for checking the setup time is **$setup**, which takes the flip-flop data input, active clock edge and the setup time as its parameters.
- The $setup task is used within a specify block.

Ex:

```
specify
$setup (d, posedge clk, 5);
endspecify
```

The $setup task that is used within the specify block continuously checks the timing distance between changes on d and the positive edge of the clk clock. If this distance is less than 5 ns, a violation message will be issued.

b) Hold time

- Hold time is defined as the minimum necessary time a flip-flop data input must stay stable (hold its value) after it is clocked.
- The Verilog construct for checking the hold time is **$hold**, which takes the flip-flop active edge of the clock, its data input, and the required hold time as its parameters.
- The $hold task must be used inside a specify block

Ex:
```
specify
$hold ( posedge clk, d, 3 );
endspecify
```

- The Verilog $setuphold task combines setup and hold timing checks.

Ex: `$setuphold  (posedge clk,  d,  5,  3);`

c) Width and period

- Verilog **$width and $period** check for minimum pulse width and period.
- **Pulse width** checks the time from a specified edge of a reference signal to its opposite edge.
- **Period** checks the time from a specified edge of a reference signal to the same edge.

EX:

```
specify
        $setuphold ( posedge clk, d, 5, 3 );
        $width  (posedge r, 4);
        $width  (posedge s, 4);
        $period (negedge clk, 43);
endspecify
```

6. Memory vectors and arrays

- Coding styles and timings discussed in the previous sections apply to arrays and vectors as well.
- The only difference is that when one-dimensional vectors or multi-dimensional arrays are being considered, their input output ports and their memory structures should be declared accordingly.

1. Vectors

- Figure shows an 8-bit transparent D-latch.
- Data input and latch output are declared as 8-bit vercors.
- The always block shown is sensitive to c and all eight bits of d

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

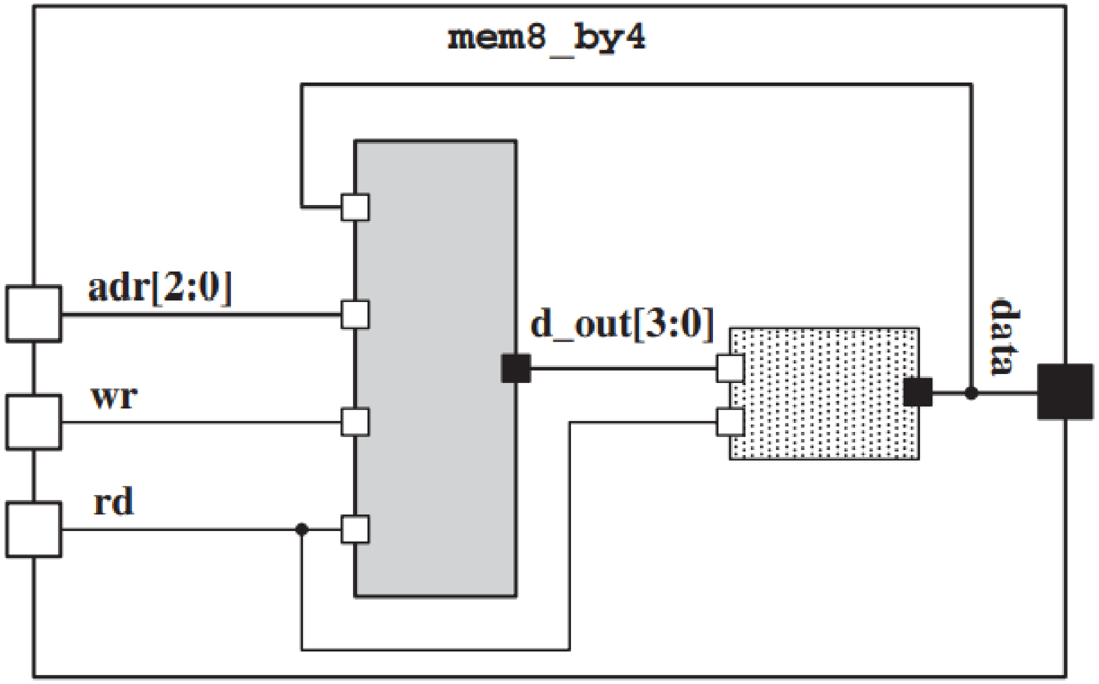
An 8-bit register with a synchronous rst input, and a tri-state output controlled by oe

```
module vector_ff (input [7:0] d, input clk, rst, oe,
                  output [7:0] q);
   reg [7:0] internal_q;

   always @( posedge clk )
      if( rst )
         #4 internal_q <= 8'b0000_0000;
      else
         #4 internal_q <= d;

   assign q = oe ? internal_q : 8'bZ;
endmodule
```

A sizable register whose size can be specified when instantiated. This register acts as an asynchronous active low reset input.

```
module sizable_reg #(size) ( input [size-1:0] d, input clk, rst,
                            output reg [size-1:0] q );
  always @( posedge clk, negedge rst )
     begin
        if( ~rst )
           #4 q <= 0;
        else
           #4 q <= d;
     end

endmodule
```

2. Arrays



- Figure shows the block diagram of an unclocked memory with an address space of 8 and word length of 4. This memory has a rd read and a wr write inputs.

```
module Memory_2Power_M_by_N #(parameter M=3, N=4)
      (input [M-1:0] adr, input rd, wr, inout [N-1:0] data);

   reg [N-1:0] mem [0:2**M-1];
   reg [N-1:0] temp;
   assign data = rd ? temp : 'bZ;
   always @( data, adr, rd, wr )
      begin
         if( wr )
            #4 mem[adr] = data;
         else if( rd )
            #4 temp = mem[adr];
         else
            #4 temp = 'bZ;
      end
   initial $readmemh("mem.dat", mem);

endmodule
```

- *A sizable Verilog code that corresponds to this block diagram.*
- *Parameters used in this code are M and N for the length of the address and data, respectively.*
- *When written into, mem holds the data, and when being read, the addressed location drives data.*
- *When rd is not active, data is float*

THANK YOU