UNIT - I

Digital Design Using Verilog

Adopted from

T. R. Padmanabhan and B. Bala Tripura Sundari,

Design through Verilog HDL –Wiley, 2009.

Prepared BY

D. Khalandar Basha

Assoc Prof., IARE

Unit - I

INTRODUCTION TO VERILOG:

- Verilog as HDL
- Levels of design Description
- Concurrency
- Simulation and Synthesis
- Functional Verification
- System Tasks
- Programming Language Interface (PLI)
- Module
- Simulation and Synthesis Tools
- Test Benches.

LANGUAGE CONSTRUCTS AND CONVENTIONS:

- Introduction, Keywords, Identifiers, White Space Characters, Comments,
- Numbers
- Strings
- Logic Values
- Strengths
- Data Types
- Scalars and Vectors
- Parameters
- Operators.

Objectives and Outcomes

Objective: To make the student learn and understand

- Acquire a basic knowledge of the Verilog HDL
- Language constructs and conventions in Verilog
- Basic Concepts of Verilog HDL like Data Types, System Tasks and Compiler Directives.

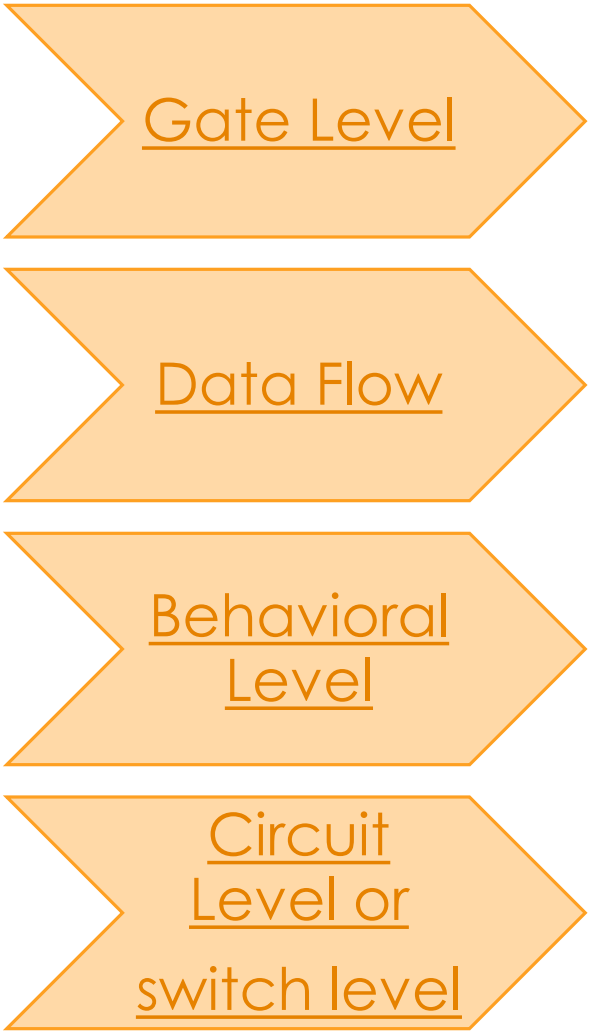
Outcomes: The student will be able to

- Define basic terms in HDL
- Knows Syntax and lexical conventions
- Remembers Data types, operators
- Remember testbenches for simulation and verification

VERILOG AS AN HDL

- Verilog aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC.

LEVELS OF DESIGN DESCRIPTION

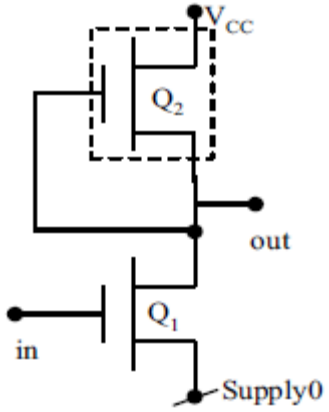


Circuit Level or switch level

- At the circuit level, a switch is the basic element

  with which digital circuits are built.

- Switches can be combined to form inverters and other gates at the next higher level of abstraction.

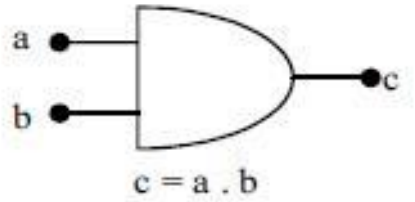


Gate Level

- At the next higher level of abstraction,

  design is carried out in terms of basic gates.

- All the basic gates are available as ready modules called "*Primitives*".



Data Flow

- Data flow is the next higher level of abstraction.
- All possible operations on signals and variables are represented here in terms of assignments

$$y = (ab+cd)$$

Behavioral Level

- Behavioral level constitutes the highest level of design description; it is essentially at the system level itself.
- With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a “C” program.

Verilog Language Concepts

- Concurrency
- Simulation and Synthesis
- Functional Verification
- System Tasks
- Programming Language Interface (PLI)

Concurrency

- In an electronic circuit all the units are to be active and functioning concurrently. The voltages and currents in the different elements in the circuit can change simultaneously. In turn the logic levels too can change.
- Simulation of such a circuit in an HDL calls for concurrency of operation.
- All the activities scheduled at one time step are completed and then the simulator.

Simulation and Synthesis

- The design that is specified and entered as described earlier is simulated for functionality and fully debugged.
- Translation of the debugged design into the corresponding hardware circuit (using an FPGA or an ASIC) is called "synthesis."
- The circuits realized from them are essentially direct translations of functions into circuit elements.

Functional Verification

- Testing is an essential ingredient of the VLSI design process as with any hardware circuit.
- It has two dimensions to it – *functional tests* and *timing tests*.
- Testing or functional verification is carried out by setting up a "test bench" for the design.

System Tasks

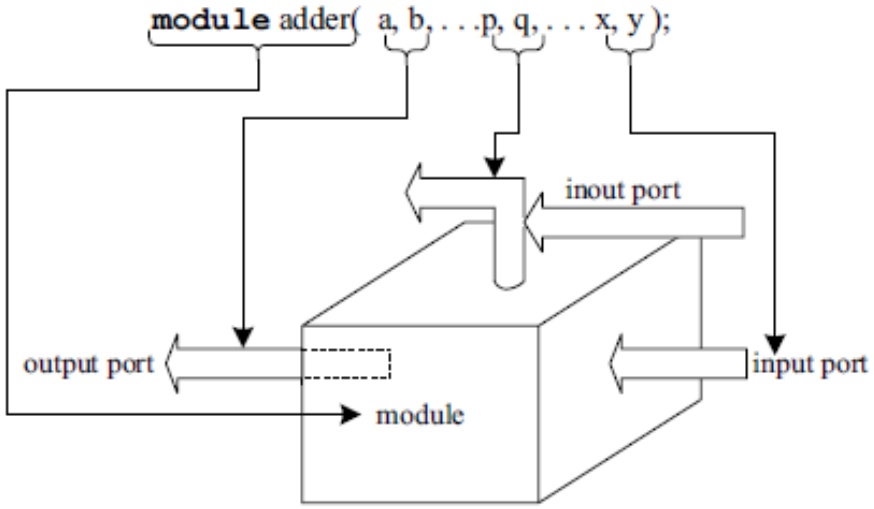
- A number of system tasks are available in Verilog.
- Though used in a design description, they are not part of it.
- Some tasks facilitate control and flow of the testing process.
- A set of system functions add to the flexibility of test benches: They are of three categories:
  - Functions that keep track of the progress of simulation time
  - Functions to convert data or values of variables from one format to another
  - Functions to generate random numbers with specific distributions.
- There are other numerous system tasks and functions

Programming Language Interface (PLI)

- Programming Language Interface (PLI) is a way to provide Application Program Interface (API) to Verilog HDL.
- Essentially it is a mechanism to invoke a C function from a Verilog code.
- PLI is primarily used for doing the things which would not have been possible otherwise using Verilog syntax.

MODULE

- Any Verilog program begins with a keyword – called a "**module**."
- A **module** is the name given to any system considering it as a black box with input and output terminals as shown in Figure
- The terminals of the module are referred to as 'ports'.



Cont…

- The ports attached to a module can be of three types:
  - **input** ports through which one gets entry into the module
  - **output** ports through which one exits the module.
  - **inout** ports: These represent ports through which one gets entry into the module or exits the module
- All the constructs in Verilog are centred on the module.

MODULE SYNTAX

- module module_name (port_list);

  Input, output, inout declaration

  Intermediate variable declarations

  Functional Description

  (gate / switch / data flow / Behv.)

endmodule

SIMULATION AND SYNTHESIS TOOLS

- A variety of Software tools related to VLSI design is available.
- Two of them are
  - Modelsim and
  - Leonardo Spectrum of MentorGraphics.
- Modelsim has been used to simulate the designs.
- Leonardo Spectrum has been used to obtain the synthesized circuits

TEST BENCH SYNTAX

- A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli.

- module tb_module_name ;

    Input, output, inout declaration

    Intermediate variable declarations

    Stimulus (initial / always)

  endmodule

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

- **CASE SENSITIVITY**

  Verilog is a case-sensitive language like C

- **KEYWORDS**
  - The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated
  - All keywords in Verilog are in small letters

IDENTIFIERS

- **IDENTIFIERS**
  - Any program requires blocks of statements, signals, *etc.*, *to be identified with an* attached nametag. Such nametags are identifiers
  - All characters of the alphabet or an underscore can be used as the first character. Subsequent characters can be of alphanumeric type, or the underscore (_), or the dollar ($) sign

WHITE SPACE CHARACTERS , COMMENTS

- **WHITE SPACE CHARACTERS**
  - Blanks (\b), tabs (\t), newlines (\n), and form feed form the white space characters in Verilog
- **COMMENTS**
  - A single line comment begins with “//”
  - multiline comments “/*” signifies the beginning of a comment and “*/” its end.

NUMBERS, STRINGS

- **NUMBERS**

  Integer Numbers : the number is taken as 32 bits wide.

  - 25, 253, –253
  - - 8 'h f 4

  Real Numbers: Real numbers can be specified in decimal or scientific notation

  4.3, 4.3e2

- **STRINGS :** A string is a sequence of characters enclosed within double quotes
  - “This is a string”

LOGIC VALUES

- 1 signifies the 1 or high or true level
- 0 signifies the 0 or low or false level.
- Two additional levels are also possible designated as **x and z.**
  - **x** represents an unknown or an uninitialized value. This corresponds to the don’t care case in logic circuits.
  - **z** represents / signifies a high impedance state

STRENGTHS

Strength Name	Strength Level	Element Modelled	Declaration Abbreviation
Supply Drive	7	Power supply connections.	supply
Strong Drive	6	Default gate & assign output strength.	strong
Pull Drive	5	Gate & assign output strength.	pull
Large Capacitor	4	Size of trireg net capacitor.	large
Weak Capacitor	3	Gate & assign output strength.	weak
Medium Capacitor	2	Size of trireg net capacitor.	medium
Small Capacitor	1	Size of trireg net capacitor.	small
High Impedence	0	Not Applicable.	highz

Data Types

- The data handled in Verilog fall into two categories:

  (i) Net data type

  (ii) Variable data type

- The two types differ in the way they are used as well as with regard to their respective hardware structures.

Net data type

- A net signifies a connection from one circuit unit to another, which carries the value of the signal it is connected to and transmits to the circuit blocks connected to it.
- If the driving end of a net is left floating, the net goes to the high impedance state.
- Various nets supported in Verilog

WIRE / TRI	WAND / TRIAND
WOR / TRIOR	TRI1
TRI0	TRIREG -- Infers a capacitance
SUPPLY1 -- For Vdd	SUPPLY0 -- For Vss

DIFFERENCES BETWEEN *WIRE* AND *TRI*

- **wire**: It represents a simple wire doing an interconnection.

  Only one output is connected to a wire and is driven by that.

- **tr**i: It represents a simple signal line as a wire. Unlike the wire, a tri can be driven by more than one signal outputs.

Contention

WIRE /TRI	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

WAND /TRIAND	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

WOR /TRIOR	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

TRI1(0)	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	1(0)

Variable Data Type

- A variable is an abstraction for a storage device
  - reg
  - time
  - integer
  - real
  - Realtime
- **MEMORY**
  - Reg [15:0] memory[511:0];
    - an array called “memory”; it has 512 locations.
    - Each location is 16 bits wide

Scalars and Vectors

- Entities representing single bits — whether the bit is stored, changed, or transferred — are called “scalars.”
- Multiple lines carry signals in a cluster treated as a “vector.”

```
reg[2:0] b;

reg[4:2] c;

wire[-2:2] d ;
```

  - All the above declarations are vectors.
  - If range is not specifies it is treated as scalars

Parameters, Operators.

PARAMETERS

All constants can be declared as parameters at the outset in a Verilog module

- parameter word_size = 16;
- parameter word_size = 16, mem_size = 256;

OPERATORS

- Unary: – for example, ~a.
- Binary: – for example, a&b.
- Ternary: – for example, a?b:c

Digital Design Through Verilog

Adopted from

T. R. Padmanabhan and B. Bala Tripura Sundari,

Design through Verilog HDL –Wiley, 2009.

UNIT - II

Prepared BY

D. Khalandar Basha

Assoc Prof., IARE

Unit - II

GATE LEVEL MODELING:

- Introduction
- AND Gate Primitive
- Module Structure
- Other Gate Primitives
- Illustrative Examples
- Tri-State Gates
- Array of Instances of Primitives
- Design of Flip – Flops with gate primitives
- Delays
- Strengths and Contention Resolution,
- Net Types
- Design of Basic Circuits.

MODELING AT DATA FLOW LEVEL:

- Introduction
- Continuous Assignment Structures
- Delays and Continuous Assignments
- Assignment to Vectors, Operators.

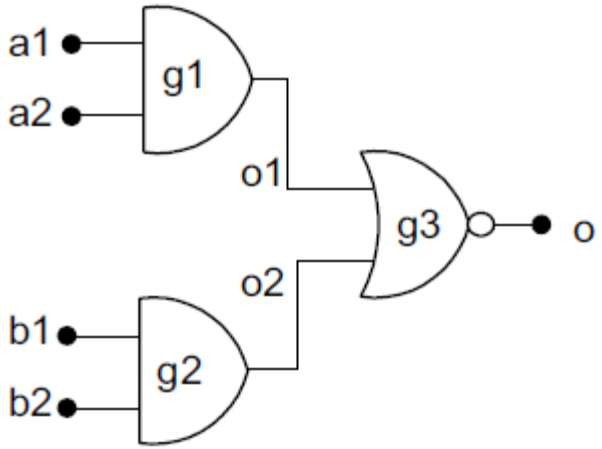
GATE LEVEL MODELING

- All the basic gates are available as “Primitives” in Verilog.

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	`and` ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	`or` gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	`nand` gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	`nor` gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	`xor` gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	`xnor` gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	`buf` gb (o1, o2, ... i);	o1, o2, o3, . .	i
NOT	`not` gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Verilog module for AOI logic

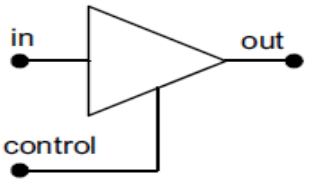
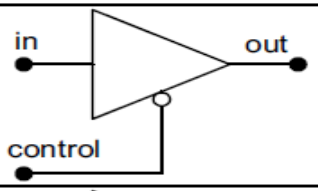
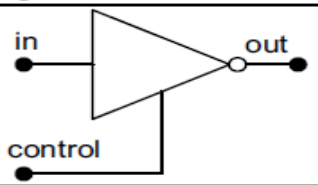
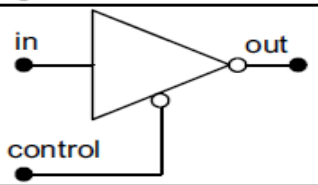
- module aoi_gate(o,a1,a2,b1,b2);
  input a1,a2,b1,b2; output o;
  wire o1,o2;
  and g1(o1,a1,a2);
  and g2(o2,b1,b2);
  nor g3(o,o1,o2);
  endmodule



- module aoi_st;
  reg a1,a2,b1,b2;
  wire o;
  initial
  begin
  a1 = 0; a2 = 0; b1 = 0; b2 = 0;
  #3 a1 = 1; a2 = 1; b1 = 1; b2 = 0;
  end
  initial #100 $stop;
  initial $monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b ",o,a1,a2,b1,b2);
  aoi_gate gg(o,a1,a2,b1,b2);
  endmodule

TRI-STATE GATES

- Four types of tri-state buffers are available in Verilog as primitives

Typical instantiation	Functional representation	Functional description
`bufif1` (out, in, control);		Out = in if control = 1; else out = z
`bufif0` (out, in, control);		Out = in if control = 0; else out = z
`notif1` (out, in, control);		Out = complement of in if control = 1; else out = z
`notif0` (out, in, control);		Out = complement of in if control = 0; else out = z

ARRAY OF INSTANCES OF PRIMITIVES

- and gate [7 : 4] (a, b, c);

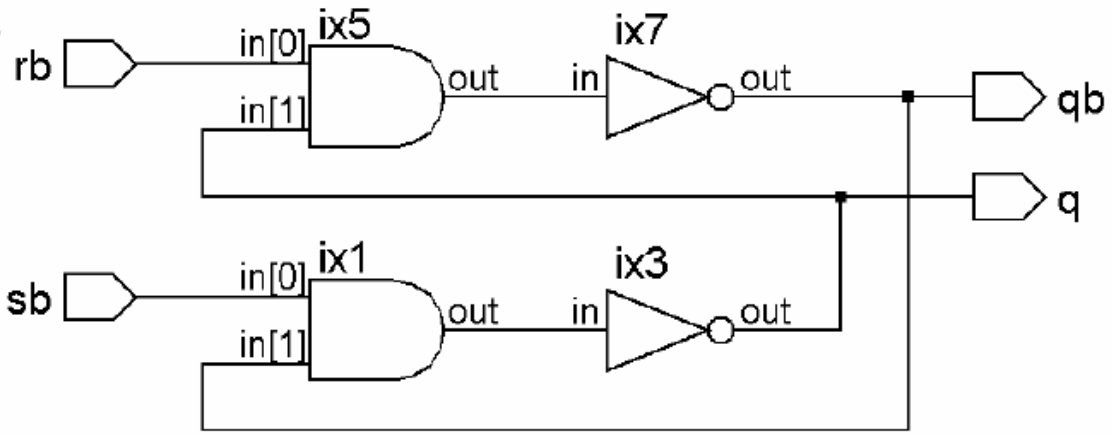
- and gate [7] (a[3], b[3], c[3]),
  gate [6] (a[2], b[2], c[2]),
  gate [5] (a[1], b[1], c[1]),
  gate [4] (a[0], b[0], c[0]);

Syntax: **and gate[mm : nn](a, b, c);**

DESIGN OF FLIP-FLOPS WITH GATE PRIMITIVES

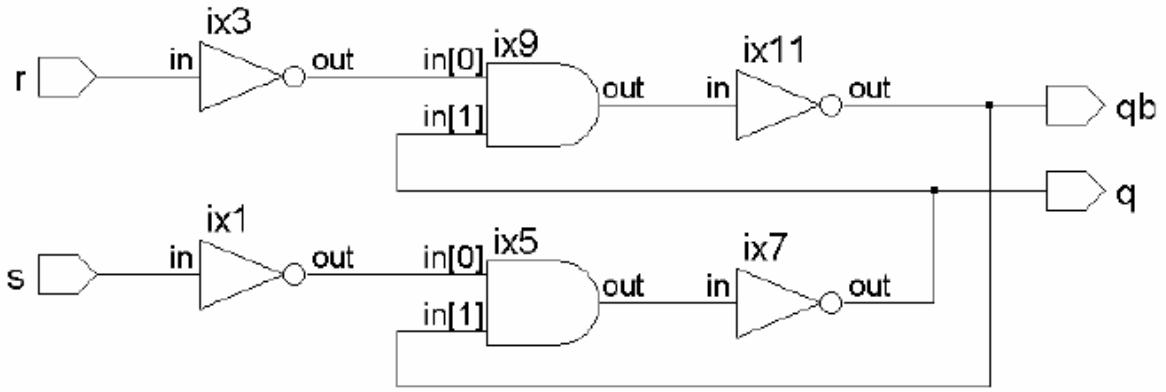
- ***Simple Latch***

```
module sbrbff(sb,rb,q,qb);
    input sb,rb;
    output q,qb;

    nand(q,sb,qb);
    nand(qb,rb,q);

endmodule
```



RS Flip-Flop module

- module srff(s,r,q,qb);

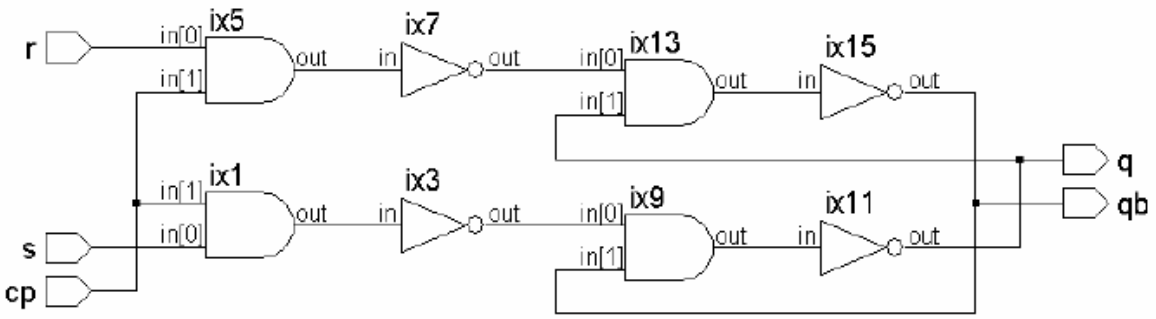



```
input s,r;
output q,qb;
wire ss,rr;

not(ss,s),(rr,r);
nand(q,ss,qb);
nand(qb,rr,q);

endmodule
```

A Clocked RS Flip-Flop module

- module srffcplev(cp,s,r,q,qb);

```
input cp,s,r;
output q,qb;
wire ss,rr;

nand   (ss,s,cp),
       (rr,r,cp),
       (q,ss,qb),
       (qb,rr,q);
endmodule
```



D-Latch module

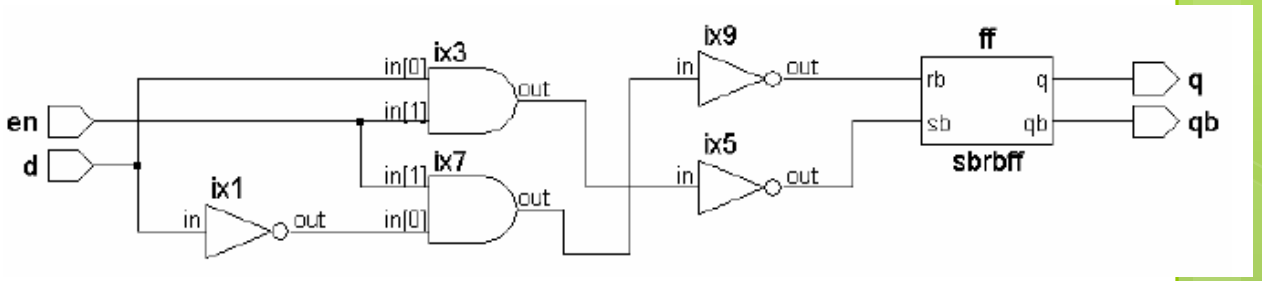
- module dlatch(en,d,q,qb);

  input d,en;

  output q,qb;

  wire dd;

  wire s,r;



  not n1(dd,d);

  nand (sb,d,en);

  nand g2(rb,dd,en);

  sbrbff ff(sb,rb,q,qb);

endmodule

DELAYS

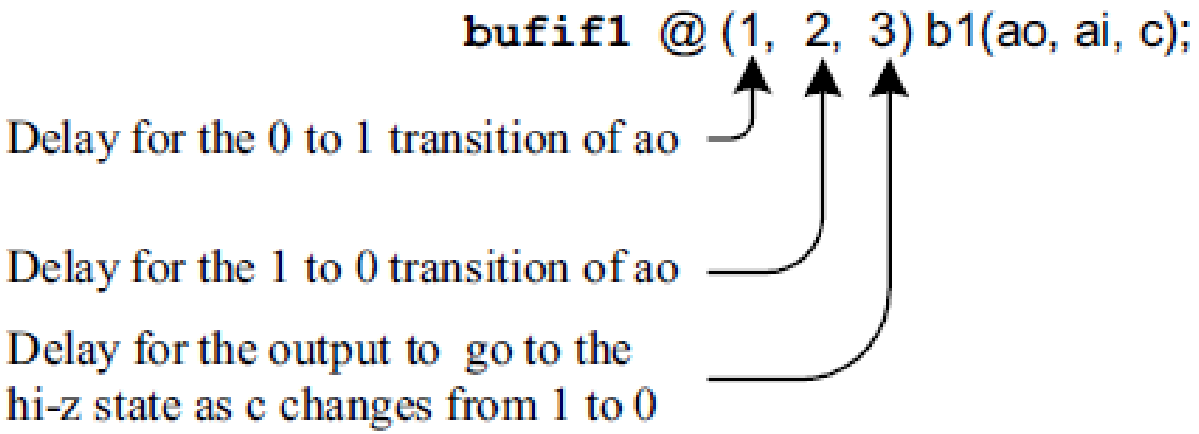
- **Net Delay**

```
wire #2 nn;
        // nn is declared as a net with a propagation delay of 2
           time steps
wire # (2, 1) nm;
        //the positive (0 to 1) transition has a delay of 2 time steps
        //The negative (1 to 0) transition has a delay of 1 time step
```

- **Gate Delay**

```
        and #3   g( a, b, c);
        and #(2, 1)  g(a, b, c);
```

Delays with Tri-state Gates



min, typical, max delays

- and #(2:3:4) g1(a0, a1, a2);
  // min, typical, max delays
- and #(1:2:3, 2:4:6) g2(b0, b1, b2);
- bufif1 #(1:2:3, 2:4:6, 3:6:9) g3 (a0, b0, c0);
- wire #(1:2:3) a;

STRENGTHS AND CONTENTION RESOLUTION

Name	supply	strong	pull	weak	High impedance
Abbreviations	su1 su0	st1 st0	pu1 pu0	we1 we0	HiZ1 HiZ0
Strength	Strongest			Weakest	

```
buf  (supply1,  pull0)  (o, i);
```

Strength of 1 state in the output

Strength of 0 state in the output

Logic value of input i1	Logic value of input i2	Logic value of output o	Remarks
0	0	0	No contention
0	1	1	Contention; the stronger signal – i2 – prevails
1	0	1	Contention; the stronger signal – i1 – prevails
1	1	1	No contention

Net Charges

- net can have a capacitor associated with it, which can store the signal level even after the signal source dries up (*i.e.*, *tri-stated).*
- Such nets are declared with the
- keyword **trireg.**

Name	`large`	`medium`	`small`
Strength	Strongest		Weakest

Signal strength names and weights

Signal strength name	Strength level
Supply (drive)	Strongest 7
Strong (drive)	6
Pull (drive)	5
Large (capacitance)	4
Weak (drive)	3
Medium (capacitance)	2
Small (capacitance)	Weakest 1
High impedance	0

MODELING AT DATA FLOW LEVEL

- **CONTINUOUS ASSIGNMENT STRUCTURES**

```
assign c = a && b;
```

- **Combining Assignment and Net Declarations**

```
wire c;
assign c = a & b;
```

can be combined as

```
wire c = a & b;
```

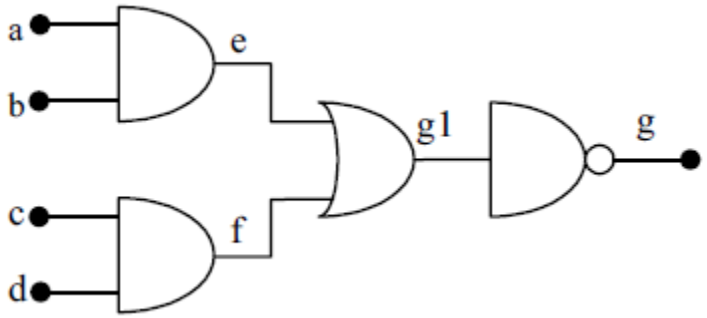
- **Continuous Assignments and Strengths**

```
wire (pull1, strong0)g = ~g1;
```

Data flow module for AOI

- module aoi2(g, a, b, c, d);
  output g;
  input a, b, c, d;
  wire e, f, g1, g;
  assign e = a && b, f = c && d, g1 = e||f, g=~g1;
  endmodule

- module aoi3(g, a, b, c, d);
  output g;
  input a, b, c, d;
  wire g;
  wire e = a && b;
  wire f = c && d;
  wire g1 = e||f;
  assign g = ~g1;
  endmodule



DELAYS AND CONCATENATION

DELAYS AND CONTINUOUS ASSIGNMENTS

- assign #2 c = a & b;

- wire #2 c;
- assign c = a & b;

CONCATENATION OF VECTORS

{a, b, c}

{a(7:4), b(2:0)}

{2{p}} = {p, p}

{2{p}, q} = {p, p, q}

{a, 3 {2{b , c}, d}} = {a, b, c, b, c, d, b, c, b, c, d, b, c, b, c, d }

OPERATORS

Unary Operators

Operator type	Symbol	Remarks
Logical negation	!	Only for scalars
Bit-wise negation	~	For scalars and vectors
Reduction AND	&	For vectors – yields a single bit output
Reduction NAND	~&	
Reduction OR	|	
Reduction NOR	~|	
Reduction XOR	^	
Reduction XNOR	~^ or ^~	

Binary Operators

- Arithmetic operators and their symbols

Operand type	Symbol	Remarks
Multiplication	*	
Division	/	The result is x if the denominator is zero
Modulus	%	
Addition	+	
Subtraction	–	

- Binary logical operators and their symbols

Operator type	Symbol	Possible output value
AND	&&	Single-bit output
OR	||	

- Relational operators and their symbols

Operator type	Symbol	Possible output value
Greater than	>	Single-bit output
Less than	<	
Greater than or equal to	>=	
Less than or equal to	<=	

Cont..

- Equality operators and their symbols

Operand symbol	Description of operand	Possible logical value of result
==	(The symbol comprises two consecutive equal signs.) If the two operands are equal bit by bit, the result is 1 (true); else the result is 0 (false). If either operand has a **x** or **z** bit, the result is **x**.	0, 1, or **x**
!=	(The symbol comprises of an exclamation mark followed by an equal sign.) A bit-by-bit comparison of the two operands is made. The result is a 1 if there is a mismatch for at least one bit position.	0, 1, or **x**
===	(The symbol comprises of three consecutive equal signs.) The operand bits can be 0, 1, **x**, or **z**. If the two operands match on a bit by bit basis, the result is a 1 (true) bit; else it is 0 (false). Note that the result is never **x** here.	0 or 1
!==	(The symbol comprises an exclamation mark followed by 2 consecutive equal signs). The operand bits can be 0, 1, **x**, or **z**. If the two operands do not match on a bit by bit basis, the result is a 1 (true) bit; else it is 0 (false). Note that the result is never **x** here.	0 or 1

cont…

- Bit-wise logical operators and their symbols

Operator type	Symbol	Possible result
AND	&	0, 1, or **x**
OR	|	
XOR	^	
XNOR	~^ or ^~	

- Shift type operators and their symbols

Operand	Typical usage	Operation
>>	A >> b	The set of bits representing A are shifted right repeatedly b times.
<<	A<< b	The set of bits representing A are shifted left repeatedly b times.

Ternary operator

- A ? B : C
- assign y = w ? x : z;
- Assign d = (f == add) ? (a+b) : ((f = sub) ? (a-b) :
  ((f==compl) ? ~a : ~b;

Operator Priority

- The table brings out the order of precedence. The order of precedence decides the priority for sequence of execution and circuit realization in any assignment

Unary operators	! & ~& | ~| ^ ~^ + –	Highest precedence
Binary operator	* ? /	
	+ -	
	<< >>	
	< <= > >=	
	== != === !==	
	& ^ ~^	
	|	
	&&	
	||	
Ternary operators	? :	Lowest precedence

Digital Design Using Verilog

Adopted from
T. R. Padmanabhan and B. Bala Tripura Sundari, Design through Verilog HDL –Wiley, 2009.

Prepared by
D. Khalandar Basha
Assoc Prof., IARE

UNIT - III

- BEHAVIORAL MODELING:
- Introduction
- Operations and Assignments
- Functional Bifurcation
- *Initial* Construct, *Always* Construct
- Assignments with Delays *Wait* Construct
- Multiple Always Blocks
- Designs at Behavioral Level
- Blocking and Non-Blocking Assignments
- The case statement
- Simulation Flow
- *if* and *if-else* constructs
- *assign–deassign* construct, *repeat* construct, *for* loop, the *disable* construct, *while* loop, *forever* loop, parallel blocks, *force-release* construct, Event.

BEHAVIORAL MODELING

- Behavioral level modeling constitutes design description at an abstract level.
- One can visualize the circuit in terms of its key modular functions and their behavior; it can be described at a functional level itself instead of getting bogged down with implementation details.

OPERATIONS AND ASSIGNMENTS

- The design description at the behavioral level is done through a sequence of assignments.
- These are called ‘procedural assignments’ – in contrast to the continuous assignments at the data flow level.
- All the procedural assignments are executed sequentially in the same order as they appear in the design description.

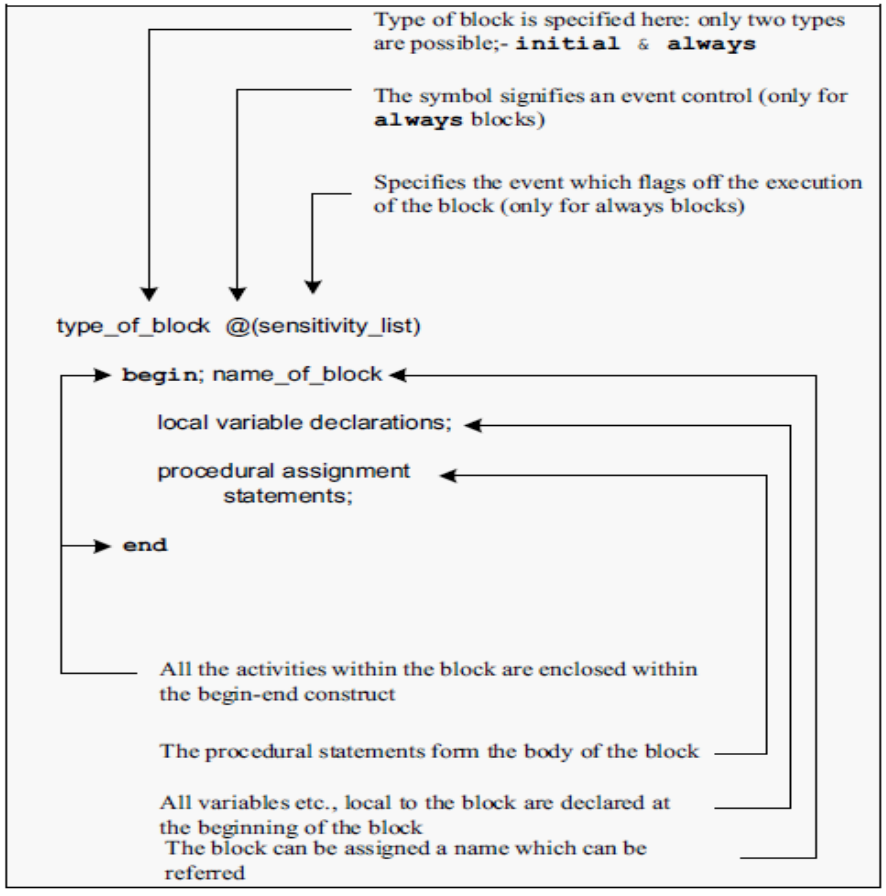
FUNCTIONAL BIFURCATION

- Design description at the behavioral level is done in terms of procedures of two types;
- one involves functional description and interlinks of functional units. It is carried out through a series of blocks under an “always”.
- The second concerns simulation – its starting point, steering the simulation flow, observing the process variables, and stopping of the simulation process; all these can be carried out under the “always” banner, an “initial” banner, or their combinations.

procedure-block structure

- A procedure-block of either type – initial or always

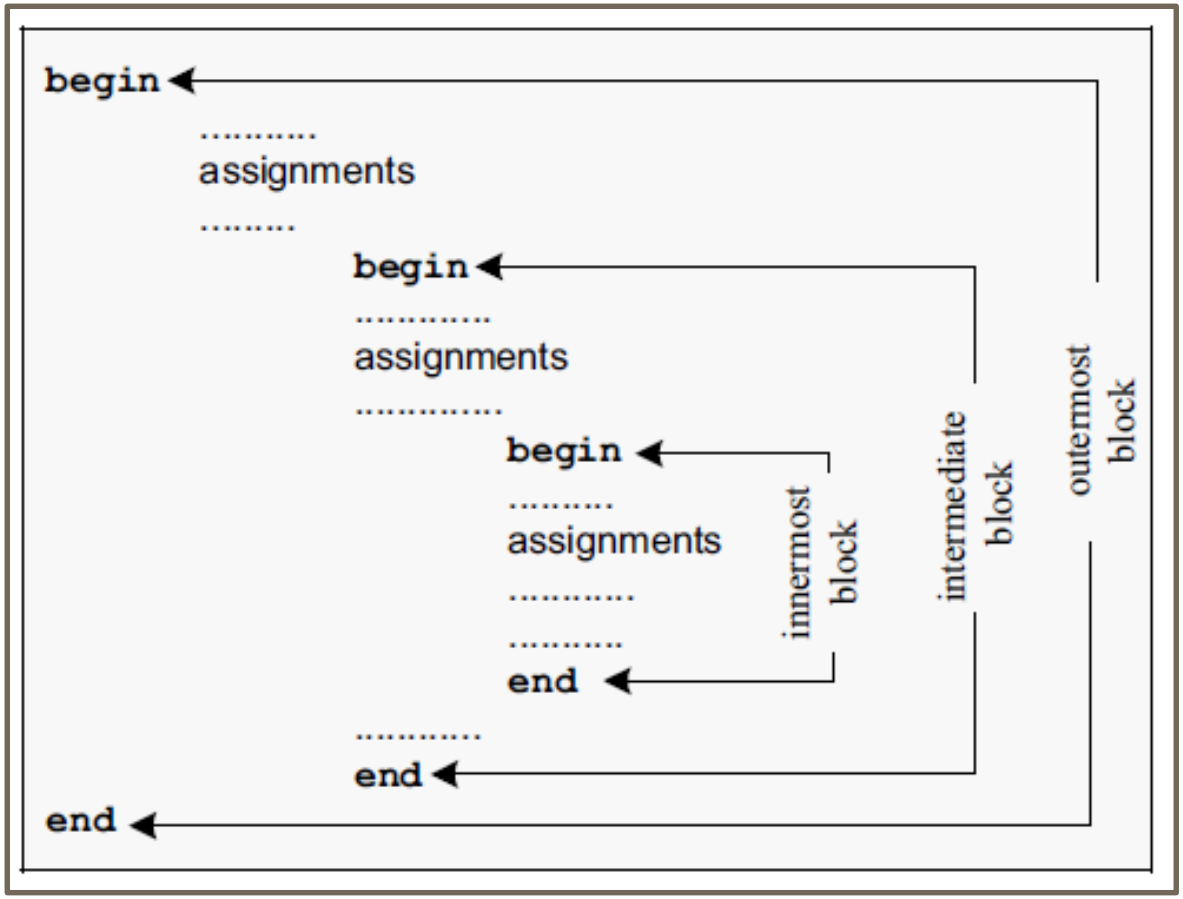
  – can have a structure shown in Figure



BEGIN – END CONSTRUCT

- If a procedural block has only one assignment to be carried out, it can be specified
- as initial #2 a=0;
- More than one procedural assignment is to be carried out in an initial block. All such
- assignments are grouped together between “begin” and “end” declarations.
- Every begin declaration must have its associated end declaration.
- begin – end constructs can be nested as many times as desired.

NESTED **BEGIN** – **END** BLOCKS



INITIAL CONSTRUCT

- A set of procedural assignments within an initial construct are executed only Once
- In any assignment statement the left-hand side has to be a storage type of element (and not a net). It can be a reg, integer, or real type of variable. The right-hand side can be a storage type of variable or a net.

```
initial
   begin
         a = 1'b0;
         b = 1'b0;
         #2 a = 1'b1;
         #3 b = 1'b1;
         #100$stop;
end
```

MULTIPLE INITIAL BLOCKS

```
module nil1;
        initial
          reg a, b;
        begin
                a = 1'b0;       b = 1'b0;
    $display ($time, "display :  a = %b, b = %b", a, b);
                #2 a = 1'b1;
        end
        initial #100$stop;
        initial
        begin           #2 b = 1'b1;
        end
    endmodule
```

ALWAYS CONSTRUCT

- The always process signifies activities to be executed on an “always basis.”
- Its essential characteristics are:
- Any behavioral level design description is done using an always block.
- The process has to be flagged off by an event or a change in a net or a reg. Otherwise it ends in a stalemate.
- The process can have one assignment statement or multiple assignment statements.
- Normally the statements are executed sequentially in the order they appear.

EVENT CONTROL

- The always block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternately such a flagging-off can be done by specifying an event preceded by the symbol “@”.
- @(negedge clk) :executes the following block at the negative edge of clk.
- @(posedge clk) : executes the following block at the positive edge of the clk.
- @clk : executes the following block at both the edges of clk.
- @(prt or clr) :
- @(posedge clk1 or negedge clk2) :
- @ (a or b or c) can also write as @ (a or b or c) @ (a, b, c) @ (a, b or c)

EXAMPLE *COUNTER*

```
module counterup(a,clk,N);
  input clk;
  input[3:0]N;
  output[3:0]a;
  reg[3:0]a;
        initial a=4'b0000;
        always@(negedge clk) a=(a==N)?4'b0000:a+1'b1;
endmodule
```

ASSIGNMENTS WITH DELAYS

- always #3 b = a;
- Values of a at the 3rd, 6th, 9th, *etc*., ns are sampled and assigned to b.
- Initial
- begin
- a = 1’b1;
- b = 1’b0;
- #1 a = 1’b0;
- #3 a = 1’b1;
- #1 a = 1’b0;
- #2 a = 1’b1;
- #3 a = 1’b0;
- end

INTRA-ASSIGNMENT DELAYS

- The “intra-assignment” delay carries out the assignment in two parts.
- An assignment with an intra-assignment has the form
- A = # dl expression;
- Here the expression is scheduled to be evaluated as soon as it is encountered.
- However, the result of the evaluation is assigned to the right-hand side quantity a after a delay specified by dl.
- dl can be an integer or a constant expression
  - always #2 a = a + 1;
  - always #b a = a + 1;
  - always #(b + c) a = a + 1;

ZERO DELAY

- A delay of 0 ns does not really cause any delay.
- However, it ensures that the assignment following is executed last in the concerned time slot.

```
always
        begin a = 1;
                #0 a = 0;
        end
```

WAIT CONSTRUCT

- The wait construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments.
- Its syntax has the form
- wait (alpha) assignment1;
- alpha can be a variable, the value on a net, or an expression involving them.
- @clk a = b; assigns the value of b to a when clk changes;
- wait (clk) #2 a = b; the simulator waits for the clock to be high and then assigns b to a

BLOCKING AND NONBLOCKING ASSIGNMENTS

- All assignment within an initial or an always block done through an equality (“=”) operator. These are executed sequentially. Such assignments block the execution of the following lot of assignments at any time step. Hence they are called “blocking assignments”.
- If the assignments are to be effected concurrently A facility called the “nonblocking assignment” is available for such situations. The symbol “<=” signifies a non-blocking assignment. The main characteristic of a nonblocking assignment is that its execution is concurrent

CONT…

- For all the non-blocking assignments in a block, the right-hand sides are evaluated first. Subsequently the specified assignments are scheduled.

- What will happen if the following statements are executed
- A <= B; // A, B will swapped
- B <= A ;

- And
- A = B;
- B = A ; // A, B will have same value

NONBLOCKING ASSIGNMENTS AND DELAYS

- The principle of Delays of the intra-assignment type operation is similar to that with blocking assignments.
- always @(a or b)
- #3 c1 = a&b;
- which has a delay of 3 ns for the blocking assignment to c1. If a or b changes, the always block is activated. Three ns later, (a&b) is evaluated and assigned to c1. The event “(a or b)” will be checked for change or trigger again. If a or b changes, all the activities are frozen for 3 ns. If a or b changes in the interim period, the block is not activated. Hence the module does not depict the desired output.
- always @(a or b)
- c2 = #3 a&b;
- The always block is activated if a or b changes. (a & b) is evaluated immediately but assigned to c2 only after 3 ns. Only after the delayed assignment to c2, the event (a or b) checked for change. If a or b changes in the interim period, the block is not activated.

- always @(a or b)
- #3 c3 <= a&b;
- The block is entered if the value of a or b changes but the evaluation of a&b and the assignment to c3 take place with a time delay of 3ns. If a or b changes in the interim period, the block is not activated.

- always @(a or b)
- c4 <= #3 a&b;
- represents the best alternative with time delay. The always block is activated if a or b changes. (a&b) is evaluated immediately and scheduled for assignment to c4 with a delay of 3 ns. Without waiting for the assignment to take effect (*i.e.*, at the same time step as the entry to the block), control is returned to the event control operator. Further changes to a or b – if any – are again taken cognizance of.

THE *CASE* STATEMENT

- simple construct for multiple branching in a module. The keywords case, endcase, and default are associated with the case construct.
- Format of the case construct is

```
Case (expression)
        Ref1 : statement1;
        Ref2 : statement2;
        Ref3 : statement3;
        .. .
        . . .
        default: statementd;
endcase
```

EXAMPLE

```
module dec2_4beh(o,i);
        output[3:0]o;
        input[1:0]i;
        reg[3:0]o;
        always@(i)
        begin
                case(i)
                        2'b00:o=4'h0;
                        2'b01:o=4'h1;
                        2'b10:o=4'h2;
                        2'b11:o=4'h4;
                default:    begin        $display ("error");
                                o=4'h0;
```

CASEX AND CASEZ

- The case statement executes a multiway branching where every bit of the case expression contributes to the branching decision. The statement has two variants where some of the bits of the case expression can be selectively treated as don’t cares – that is, ignored.
- Casez allows z to be treated as a don’t care. “?” character also can be used in place of z.
- casex treats x or z as a don’t care.

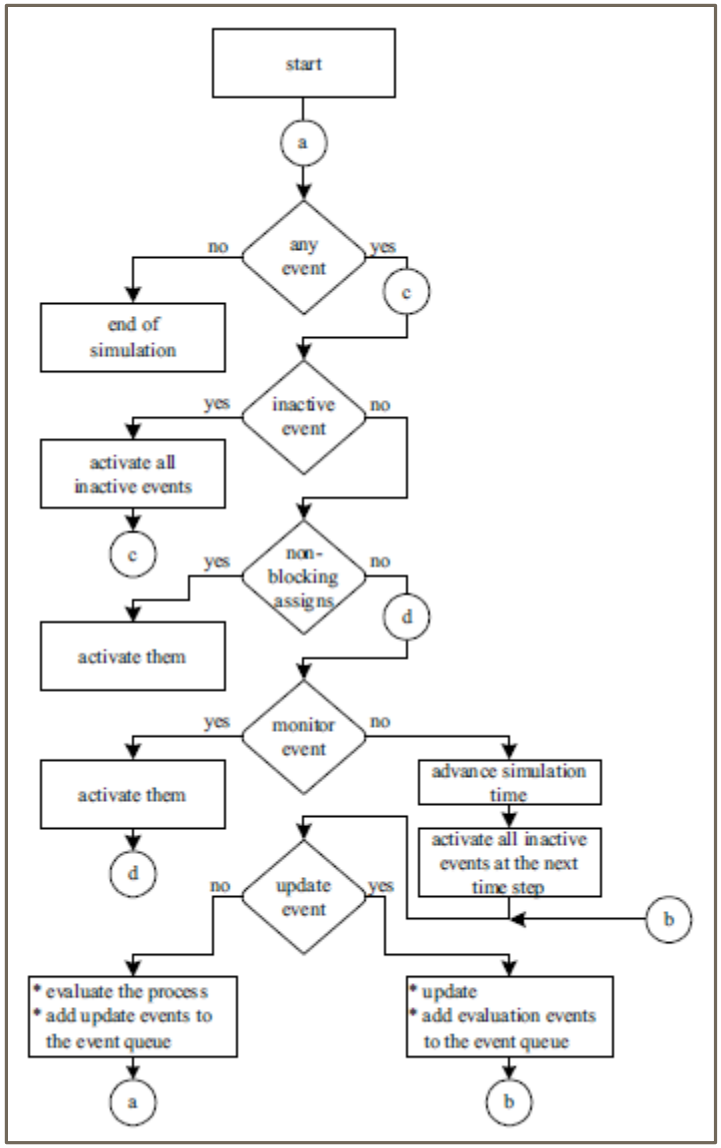
SIMULATION FLOW

- In Verilog the parallel processing is structured through the following [IEEE]:
- Simulation time: Simulation is carried out in simulation time.
- At every simulation step a number of active events are sequentially carried out.
- The simulator maintains an event queue – called the "Stratified Event Queue" – with an active segment at its top. The top most event in the active segment of the queue is taken up for execution next.
- The active event can be of an update type or evaluation type. The evaluation event can be for evaluation of variables, values on nets, expressions, *etc*. Refreshing the queue and rearranging it constitutes the update event.
- Any updating can call for a subsequent evaluation and *vice versa*.
- Only after all the active events in a time step are executed, the simulation advances to the next time step.
- Completion of the sequence of operations above at any time step signifies the parallel nature of the HDL.

STRATIFIED EVENT QUEUE

- The events being carried out at any instant give rise to other events – inherent in
- the execution process. All such events can be grouped into the following 5 types:
- Active events –
- Inactive events – The inactive events are the events lined up for execution immediately after the execution of the active events. Events specified with zero delay are all inactive events.
- Blocking Assignment Events – Operations and processes carried out at previous time steps with results to be updated at the current time step are of this category.
- Monitor Events – The Monitor events at the current time step – $monitor and $strobe – are to be processed after the processing of the active events, inactive events, and nonblocking assignment events.
- Future events – Events scheduled to occur at some future simulation time are the future events.

FLOWCHART FOR THE SIMULATION FLOW.



IF AND IF-ELSE CONSTRUCTS

- The if construct checks a specific condition and decides execution based on the result.

```
assignment1;
if (condition) assignment2;
assignment3;
```

- Use of the if–else construct.

```
assignment1;
if(condition)
        begin // Alternative 1
                assignment2;
        end
else
        begin //alternative 2
                assignment3;
        end
assignment4;
```

EXAMPLE

```
module demux(a,b,s);
        output [3:0]a;
        input b, [1:0]s;
        reg[3:0]a;
        always@(b or s)
                begin   if(s==2'b00)
                        begin   a[2'b0]=b;
        a[3:1]=3'bZZZ;          end
                        else if(s==2'b01)
                        begin   a[2'd1]=b;
        {a[3],a[2],a[0]}=3'bZZZ;        end
                        else if(s==2'b10)
                        begin   a[2'd2]=b;
        {a[3],a[1],a[0]}=3'bZZZ;        end
```

ASSIGN–DEASSIGN CONSTRUCT

- The assign – deassign constructs allow continuous assignments within a behavioral block.
- always@(posedge clk) a = b;
- At the positive edge of clk the value of b is assigned to a, and a remains frozen at that value until the next positive edge of clk. Changes in b in the interval are ignored.

- As an alternative, consider the block
- always@(posedge clk) assign c = d;
- Here at the positive edge of clk, c is assigned the value of d in a continuous manner; subsequent changes in d are directly reflected as changes in variable c:

- Always
- Begin
- @(posedge clk) assign c = d;
- @(negedge clk) deassign c;
- end
- The above block signifies two activities:
- 1. At the positive edge of clk, c is assigned the value of d in a continuous manner.
- 2. At the following negative edge of clk, the continuous assignment to c is removed; subsequent changes to d are not passed on to c; it is as though c is electrically disconnected from d.

REPEAT CONSTRUCT

- The repeat construct is used to repeat a specified block a specified number of times.
- …
- repeat (a)
- begin
- assignment1;
- assignment2;
- …
- end
- …
- The quantity a can be a number or an expression evaluated to a number.
- The following block is executed “a” times. If “a” evaluates to 0 or x or z, the block is not executed.

FOR LOOP

- The for loop in Verilog is quite similar to the for loop in C
- It has four parts; the sequence of execution is as follows:
  - 1. Execute assignment1.
  - 2. Evaluate *expression*.
  - 3. If the *expression* evaluates to the true state (1), carry out statement. Go to step 5.
  - 4. If *expression* evaluates to the false state (0), exit the loop.
  - 5. Execute assignment2. Go to step 2
- . . . .
- for(assignment1; *expression*; assignment 2)
- statement;
- . . .

THE DISABLE CONSTRUCT

- To break out of a block or loop. The disable statement terminates a named block or task. Control is transferred to the statement immediately following the block
- The disable construct is functionally similar to the *break* in C

```
always@(posedge en)
    begin:OR_gate
        b=1'b0;
        for(i=0;i<=3;i=i+1)
        if(a[i]==1'b1)
        begin    b=1'b1;
                 disable OR_gate;
        end
    end
```

WHILE LOOP

- The Boolean *expression* is evaluated. If it is true, the statement s are executed and expression evaluated and checked. If the *expression* evaluates to false, the loop is terminated and the following statement is taken for execution

```
while(|a)
begin
        b=1'b1;
        @(posedge clk)
        a=a-1'b1;
end
b=1'b0;
```

FOREVER LOOP

- Repeated execution of a block in an endless manner is best done with the forever loop (compare with repeat where the repetition is for a fixed number of times).

- always @(posedge en)
- forever#2 clk=~clk;

PARALLEL BLOCKS

- All the procedural assignments within a begin–end block are executed sequentially. The fork–join block is an alternate one where all the assignments are carried out concurrently (The non-blocking assignments too can be used for the purpose.). One can use a fork-join block within a begin–end block or vice versa.

```
module fk_jn_a;
integer a;
initial
begin
        a=0;
 #1     a=1;
 #2     a=2;
 #3     a=3;
 #4     $stop;
end
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulatiom results
# a=0, t=0
# a=1, t=1
# a=2, t=3
# a=3, t=6
```

```
module fk_jn_b;
integer a;
initial
fork
        a=0;
 #1     a=1;
 #2     a=2;
 #3     a=3;
 #4     $stop;
join
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulation results
# a=0, t=0
# a=1, t=1
# a=2, t=2
# a=3, t=3
```

FORCE–RELEASE CONSTRUCT

- When debugging a design with a number of instantiations, one may be stuck with an unexpected behavior in a localized area. Tracing the paths of individual signals and debugging the design may prove to be too tedious or difficult.
- In such cases suspect blocks may be isolated, tested, and debugged and *status quo ante* established. The force–release construct is for such a localized isolation for a limited period.
- force a = 1'b0;
- forces the variable a to take the value 0.
- force b = c&d;
- forces the variable b to the value obtained by evaluating the expression c&d.

EVENT

- The keyword event allows an abstract event to be declared. The event is not a data type with any specific values; it is not a variable (reg) or a net. It signifies a change that can be used as a trigger to communicate between modules or to synchronize events in different modules.
- The operator "→" signifies the triggering. Subsequently, another activity can be started in the module by the event change.

```
. . .
event change;
. . .
always
. . .
. . . → change;
. . .
.always@change
```

DIGITAL DESIGN USING VERILOG

Prepared BY
D. Khalandar Basha
Assoc Prof., IARE

UNIT - IV

SWITCH LEVEL MODELING

- Basic Transistor Switches, CMOS Switch, Bi – directional Gates, Time Delays with Switch Primitives, Instantiations with Strengths and Delays, Strength Contention with Trireg Nets

SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES:

- Parameters, Path Delays, Module Parameters, System Tasks and Functions, File – Based Tasks and Functions, Compiler Directives, Hierarchical Access, User-defined Primitives (UDP).

INTRODUCTION

The MOS transistor is the basic element around which a VLSI is built. Designers familiar with logic gates and their configurations at the circuit level may choose to do their designs using MOS transistors.

Verilog has the provision to do the design description at the switch level using such MOS transistors, which is the theme of the present chapter.

Switch level modeling forms the basic level of modeling digital circuits. The switches are available as primitives in Verilog

BASIC SWITCH PRIMITIVES

Different switch primitives are available in Verilog

nmos switch primitives

nmos (out, in, control);

pmos switch primitives

pmos (out, in, control);

nmos switch

(Data) input	Control (input) 0	1	X	z
0	Z	0	L	L
1	Z	1	H	H
X	Z	X	X	X
z	z	z	z	z

pmos switch

(Data) input	Control (input) 0	1	X	z
0	Z	0	L	L
1	Z	1	H	H
X	Z	X	X	X
z	z	z	z	z

Resistive Switches

- **nmos** and **pmos** represent switches of low impedance in the on-state. **rnmos** and **rpmos** represent the resistive counterparts of these respectively.

  rnmos (output1, input1, control1);

  rpmos (output2, input2, control2);

- It inserts a definite resistance between the input and the output signals but retains the signal value
- The **rpmos** and **rnmos** switches function as unidirectional switches; the signal flow is from the input to the output side.

strength levels

- **Output-side strength levels for different input-side strength values of rnmos, rpmos, and rcmos switches**

Input strength	Output strength
Supply – drive	Pull – drive
Strong – drive	Pull – drive
Pull – drive	Weak – drive
Weak – drive	Medium – capacitive
Large – capacitive	Medium – capacitive
Medium – capacitive	Small – capacitive
Small – capacitive	Small – capacitive
High impedance	High impedance

pullup and pulldown

- A MOS transistor functions as a resistive element when in the active state. Realization of resistance in this form takes less silicon area in the IC as compared to a resistance realized directly. **pullup** and **pulldown** represent such resistive elements.
- **pullup** (x);

  Here net x is pulled up to the **supply1** through a resistance.
- **pulldown**(y);

  pulls y down to the **supply0** level through a resistance.

The **pullup** and **pulldown** primitives can be used as loads for switches or to connect the unused input ports to *V*CC or GND, respectively.

CMOS SWITCH

- A CMOS switch is formed by connecting a PMOS and an NMOS switch in parallel – the input leads are connected together on the one side and the output leads are connected together on the other side.

- The CMOS switch is instantiated as shown below.

  cmos csw (out, in, N_control, P_control);

BI-DIRECTIONAL GATES

- Verilog has a set of primitives for bi-directional switches as well. They connect the nets on either side when ON and isolate them when OFF. The signal flow can be in either direction

- ***tran and rtran***

The **tran** gate is a bi-directional gate of two ports. When instantiated, it connects the two ports directly.

tran (s1, s2);

connects the signal lines s1 and s2.

Either line can be **input**, **inout** or **output**.

rtran is the resistive counterpart of **tran**.

Cont…

tranif1 and rtranif1

- **tranif1** is a bi-directional switch turned ON/OFF through a control line(c). It is in the ON-state when the control signal is at 1 (high) state

tranif1 (s1, s2, c);

tranif0 and rtranif0

- **tranif0** and **rtranif0** are again bi-directional switches. The switch is OFF if the control line is in the 1 state, and it is ON when the control line is in the 0 state.

tranif0 (s1, s2, c);

TIME DELAYS WITH SWITCH PRIMITIVES

- **nmos** g1 (out, in, ctrl);

  has no delay associated with it. The instantiation

- **nmos** (delay1) g2 (out, in, ctrl);

  has delay1 as the delay for the output to rise, fall, and turn OFF.

- **nmos** (delay_r, delay_f) g3 (out, in, ctrl);

  has delay_r as the rise-time for the output. delay_f is the fall-time for the output. The turn-off time is zero.

- **nmos** (delay_r, delay_f, delay_o) g4 (out, in, ctrl);

  has delay_r as the rise-time for the output. delay_f is the fall-time for the output delay_o is the time to turn OFF when the control signal ctrl goes from 0 to 1.

Cont…

- Delays can be assigned to the other uni-directional gates in a similar manner.
- Bi-directional switches do not delay transmission – their rise- and fall-times are zero. They can have only turn-on and turn-off delays associated with them.
- **tran** has no delay associated with it.
- **tranif1** (delay_r, delay_f) g5 (out, in, ctrl);

When control changes from 0 to 1, the switch turns on with a delay of delay_r.
When control changes from 1 to 0, the switch turns off with a delay of delay_f.

- **transif1** (delay0) g2 (out, in, ctrl);

represents an instantiation with delay0 as the delay for the switch to turn on when control changes from 0 to 1, with the same delay for it to turn off when control changes from 1 to 0

INSTANTIATIONS WITH STRENGTHS AND DELAYS

nmos (**strong1**, **strong0**) (delay_r, delay_f, delay_o) gg (s1, s2, ctrl) ;

rnmos, **pmos**, and **rpmos** switches too can be instantiated in the general form in the same manner. The general instantiation for the bi-directional gates too can be done similarly.

STRENGTH CONTENTION WITH TRIREG NETS

- nets declared as **trireg** can have capacitive storage. Such storage can be assigned one of three strengths – **large**, **medium**, or **small**.
- Driving such a net from different sources can lead to contention

PARAMETERS

- Constants signifying timing values, ranges of variables, wires, *etc*., can be specified in terms of assigned names. Such assigned names are called parameters.
- Two types of parameters are of use in modules
  - Parameters related to timings, time delays, rise and fall times, *etc*., are technology-specific and used during simulation. Parameter values can be assigned or overridden with the keyword "**specparam**" preceding the assignments.
  - Parameters related to design, bus width, and register size are of a different category. They are related to the size or dimension of a specific design; they are technology-independent. Assignment or overriding is with assignments following the keyword "**defparam**".

PATH DELAYS

- Verilog has the provision to specify and check delays associated with total paths – from any input to any output of a module. Such paths and delays are at the chip or system level. They are referred to as “module path delays.”
- **Specify Blocks**

  Module paths are specified and values assigned to their delays through **specify** blocks. They are used to specify rise time, fall time, path delays pulse widths.

```
specify
        specparam rise_time = 5, fall_time = 6;
        (a =>b) = (rise_time, fall_time);
        (c => d) = (6, 7);
endspecify
```

Module Paths

- Module paths can be specified in different ways inside a specify block. The simplest has the form A*>B
- Here “A” is the source and “B” the destination.

```
specify
   (a,b*>s)=1;
   (a,b*>ca)=2;
endspecify
```

Conditional Pin-to-Pin Delays

- The pin to pin path of a signal may change depending on the value of another signal; in turn the number of circuit elements in the alternate path may differ.

```
specify
        if(f==2'b00)(a=>d)=1;
        if(f >2'b00)(a=>d)=2;
        (b,cci*>co)=1;
endspecify
```

MODULE PARAMETERS

- Module parameters are associated with size of bus, register, memory, ALU, and so on. They can be specified within the concerned module but their value can be altered during instantiation. The alterations can be brought about through assignments made with **defparam**. Such **defparam** assignments can appear anywhere in a module.

SYSTEM TASKS AND FUNCTIONS

- A "$" sign preceding a word or a word group signifies a system task or a system function
- **Output Tasks**

  $monitor and **$display**

- **Display Tasks**

The **$display** task, whenever encountered, displays the arguments in the desired format; and the display advances to a new line. **$write** task carries out the desired display but does not advance to the new line

$strobe Task

- When a variable or a set of variables is sampled and its value displayed, the **$strobe** task can be used; it senses the value of the specified variables and displays them.

$monitor Task

- **$monitor** task is activated and displays the arguments specified whenever any of the arguments changes

- **$stop and $finish Tasks**

  The **$stop** task suspends simulation.

  $finish stops simulation, closes the simulation environment, and reverts to the operating system.

$random Function

- One can start with a seed number (optional) and generate a random number repeatedly. Such random number sequences can be fruitfully used for testing.

FILE-BASED TASKS AND FUNCTIONS

- To carry out any file-based task, the file has to be opened, reading, writing, *etc*., completed and the file closed. The keywords for all file-based tasks start with the letter f to distinguish them from the other tasks
- All the system tasks to output information can be used to output to a file. **$display**, **$strobe**, **$monitor**, *etc*., are of this category. The respective keywords to output to the file are **$fdisplay**, **$fstrobe**, **$fmonitor**.
- The first field of the task statement is an argument – the file descriptor. The subsequent fields are identical to the corresponding nonfile tasks.

COMPILER DIRECTIVES

- They allow for macros, inclusion of files, and timescale-related parameters for simulation. All compiler directives are preceded by the ‘`’.
- **`define Directive**

The **`define** directive is used to define and associate the desired text with the macro name

`define add 2'b00

Time-Related Tasks

- The **`timescale** compiler directive allows the time scale to be specified for the design. The **`timescale** directive has two components
- **`timescale** 1 ms/100 µs

HIERARCHICAL ACCESS

- A Verilog design will normally have a module or two at the apex level. A number of modules and UDPs will be instantiated within it.
- $display("fad.a = %0d, fad.b = %0d, fad.fad = %0d", fad.a,fad.b,fad.fad);

USER-DEFINED PRIMITIVES (UDP)

- The primitives available in Verilog are all of the gate or switch types. Verilog has the provision for the user to define primitives – called “user defined primitive (UDP)” and use them.
- A UDP can be defined anywhere in a source text and instantiated in any of the modules. Their definition is in the form of a table in a specific format.
- UDPs are basically of two types – combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function

Combinational UDPs

- A combinational UDP accepts a set of scalar inputs and gives a scalar output. An **inout** declaration is not supported by a UDP.
- The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other mo

```
primitive udp_and (out, in1, in2);
    output out;
    input in1, in2;
    table
            // In1 In2 Out
                0 0: 0;
                 0 1: 0;
                  1 0: 0;
                  1 1: 1;
     endtable
endprimitivedule
```

Sequential UDPs

- Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state A sequential UDP can accommodate all these.

```
primitive dff_pos(q,din,clk,clr);
output q;
input din,clk,clr;
reg q;
    table
        // din clk clr qp qn Whatever be the present
            0 (01) 0: ?: 0; // state of the output, at the
            1 (01) 0: ?: 1; // positive edge of clk input
            ? (10) 0: ?: -; // value is latched and
    endtable
endprimitive
```

Digital Design using Verilog

UNIT - V

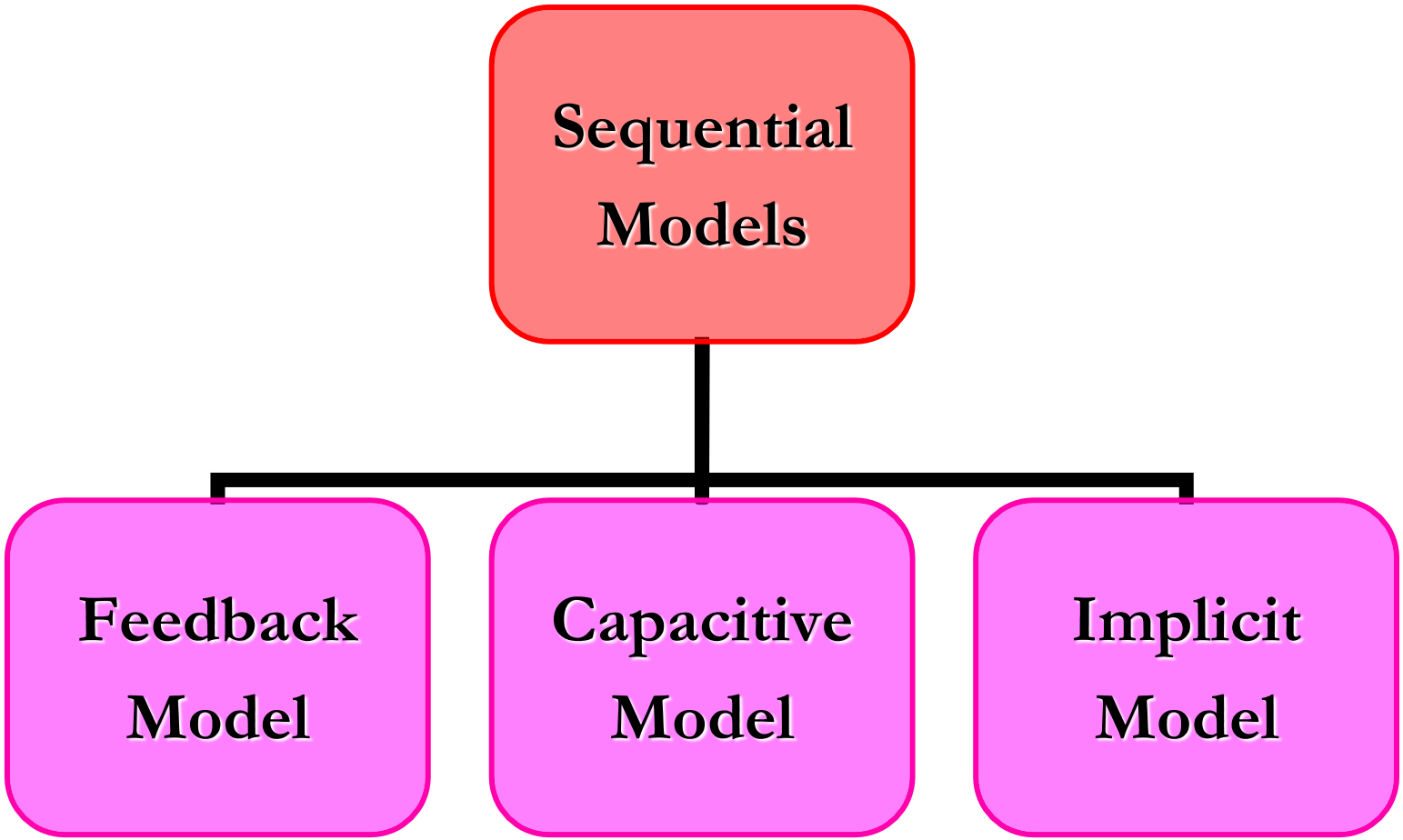
Sequential Circuit Description

- This chapter concentrates on:
  - Using Verilog constructs for description of sequential circuits
- Discussion of using gate level and assignments and procedural statements for describing memory elements.

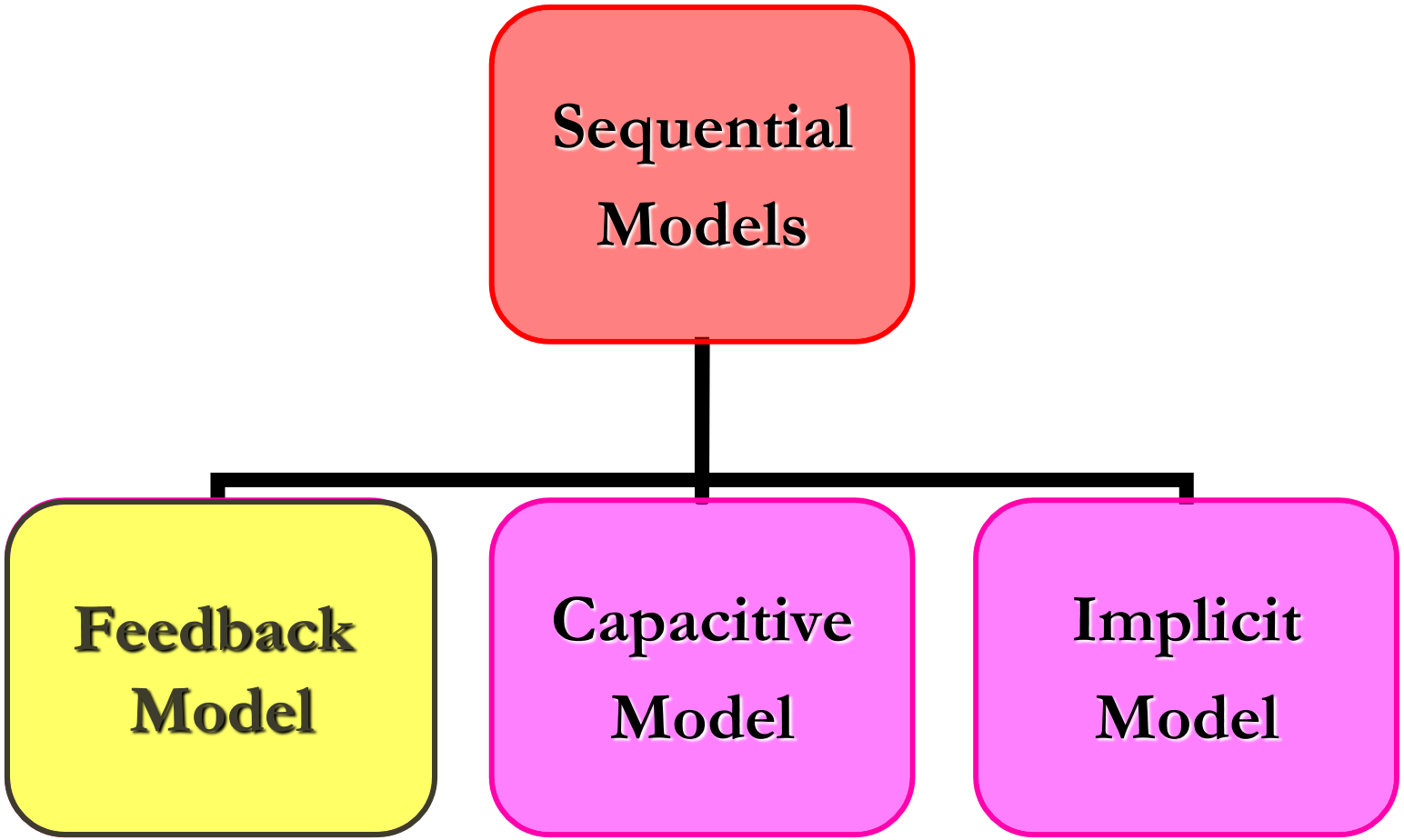
Sequential Models

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

Sequential Models

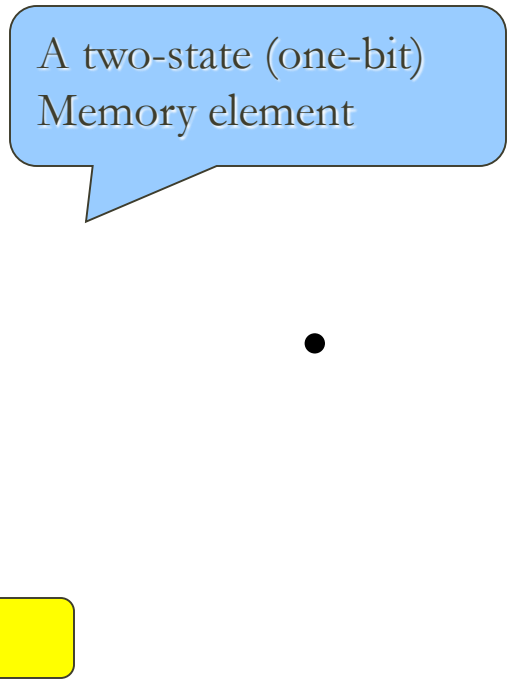


Feedback Model

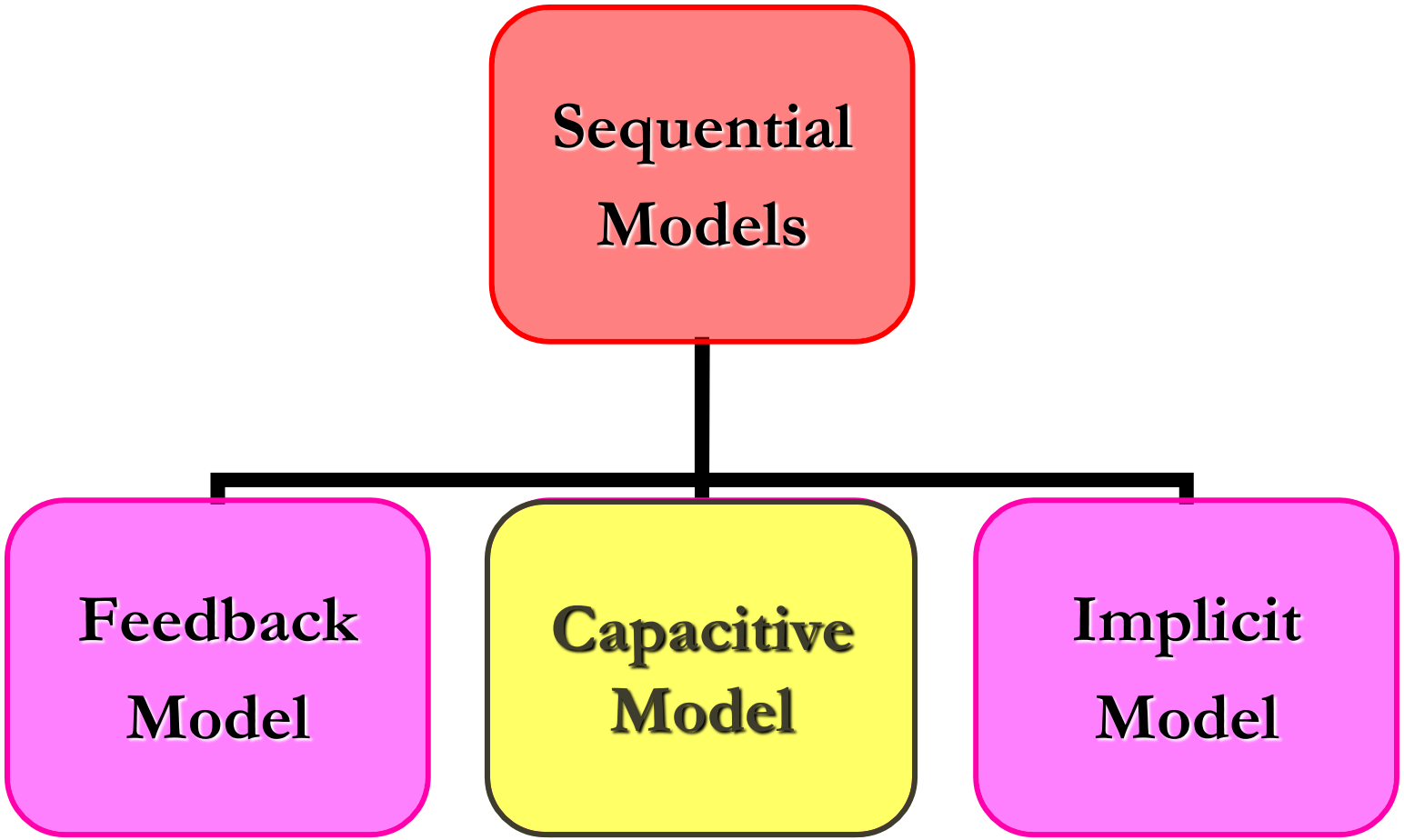


Feedback Model

- Basic Feedback



Capacitive Model



Capacitive Model

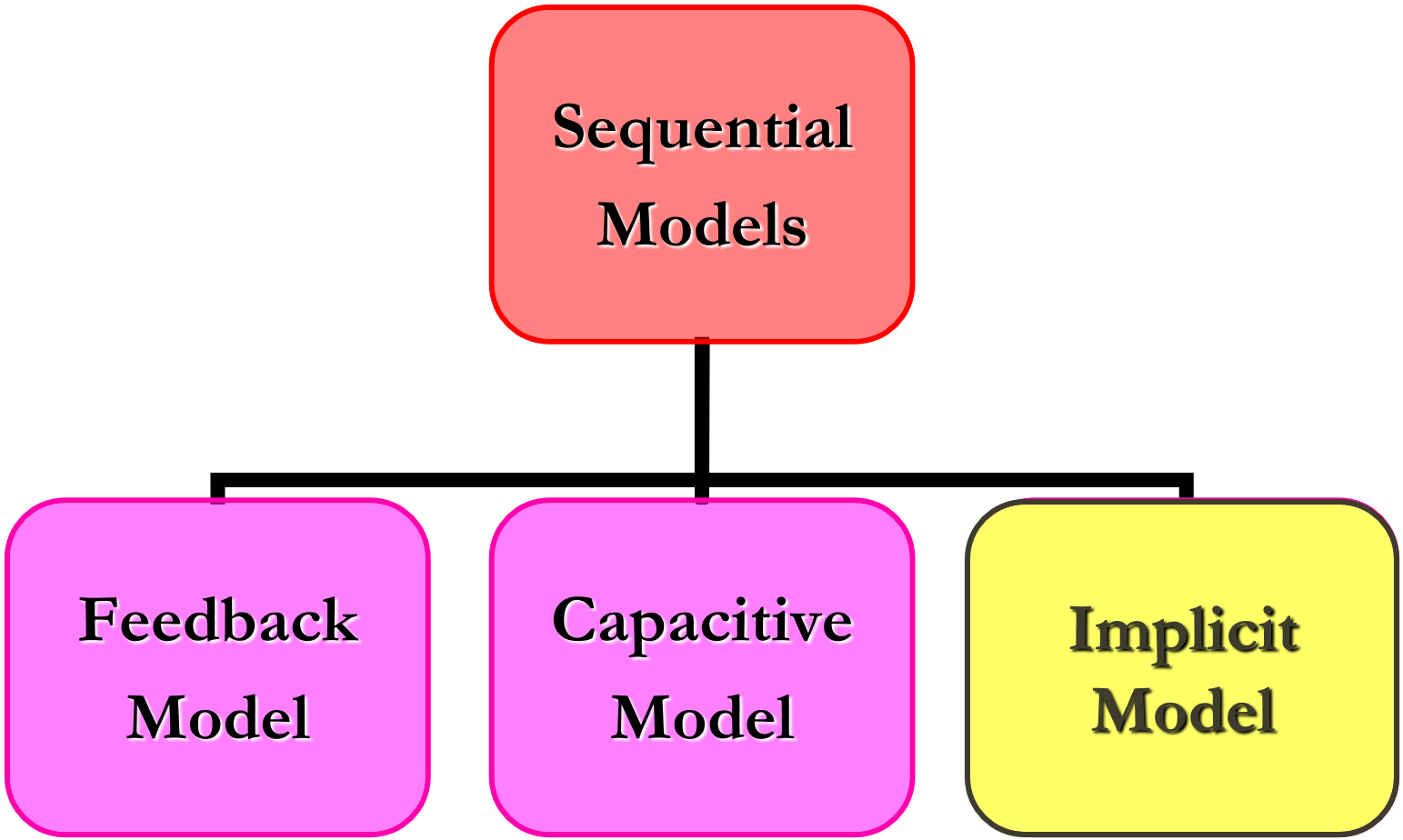
When c becomes 1 the value of D is saved in the input gate of the inverter and when c becomes 0 this value will be saved until the next time that c becomes 1 again.

The complement of the stored data

$\overline{\quad}$

- Capacitive Storage

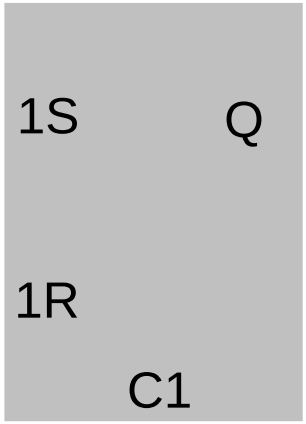
Implicit Model



Implicit Model

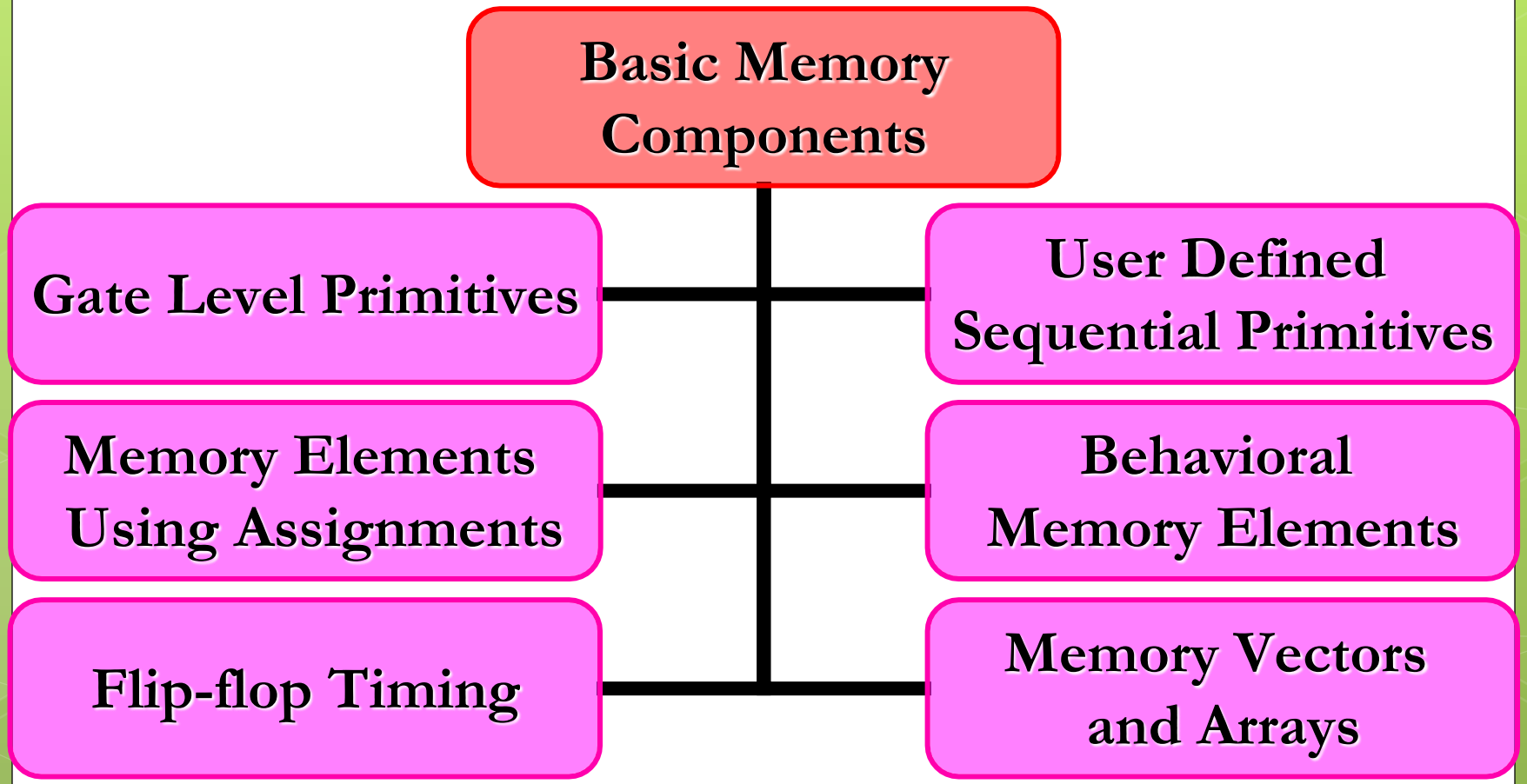
Feedback and capacitive models are technology dependent and have the problem of being too detailed and too slow to simulate.

Verilog offers language constructs that are technology independent and allow much more efficient simulation of circuits with a large number of storage elements.

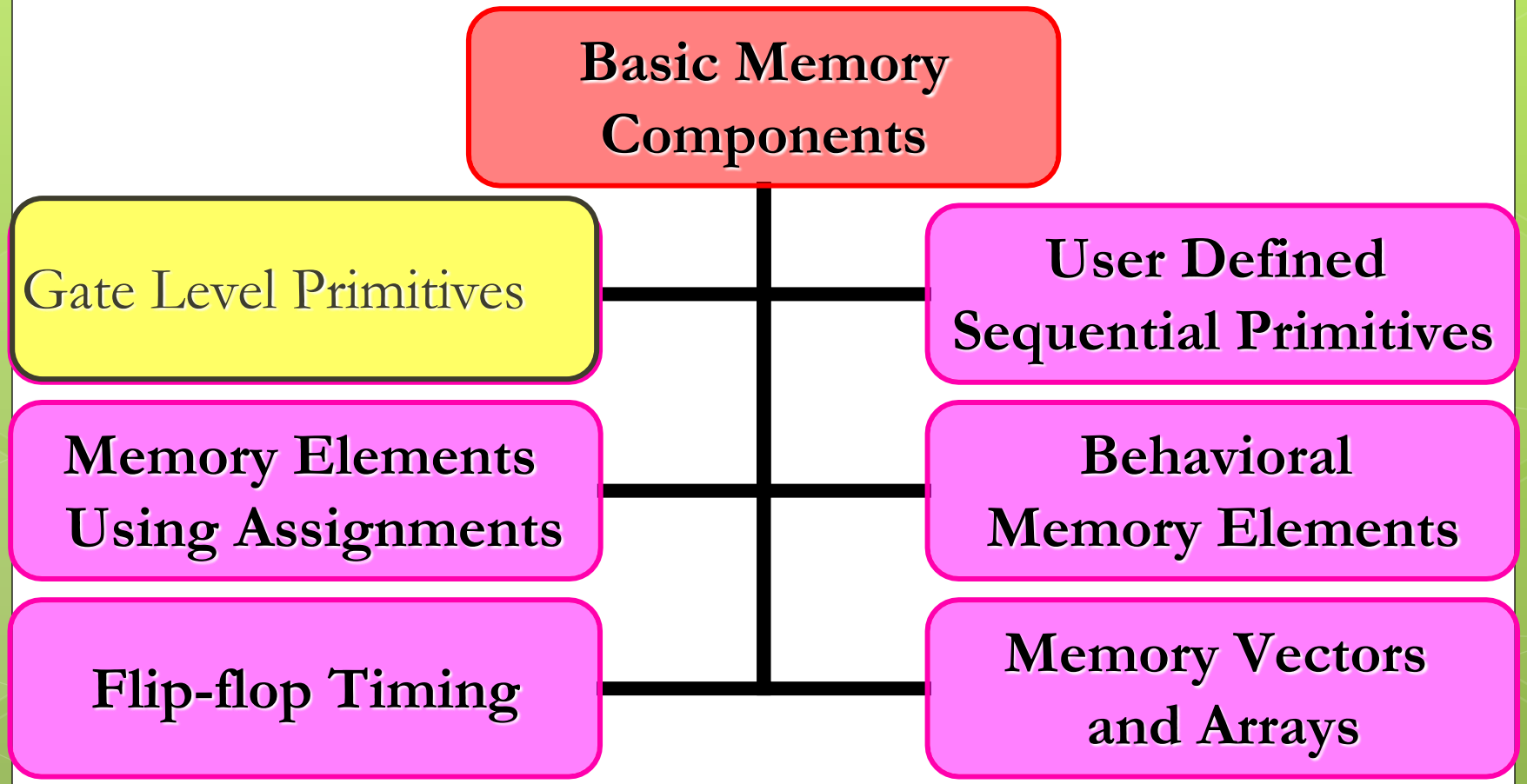


- An SR-Latch Notation

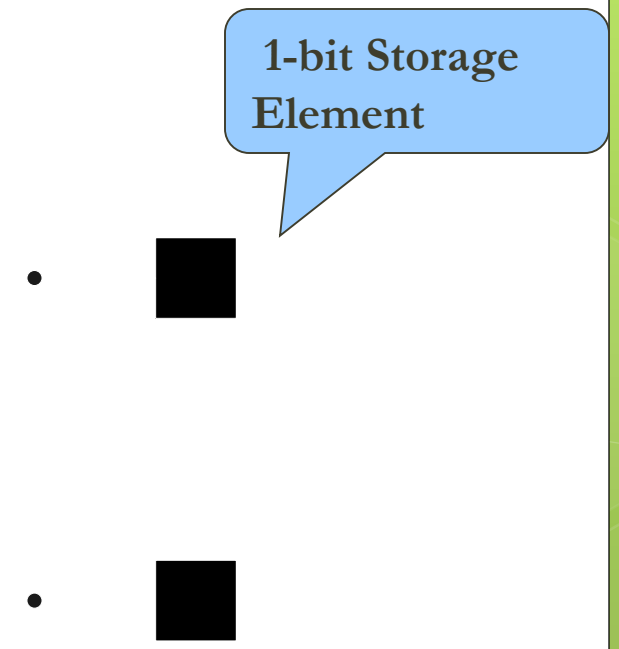
Basic Memory Components



Gate Level Primitives



Gate Level Primitives



- Cross-Coupled NOR Latch

Gate Level Primitives

Base of most static memory components

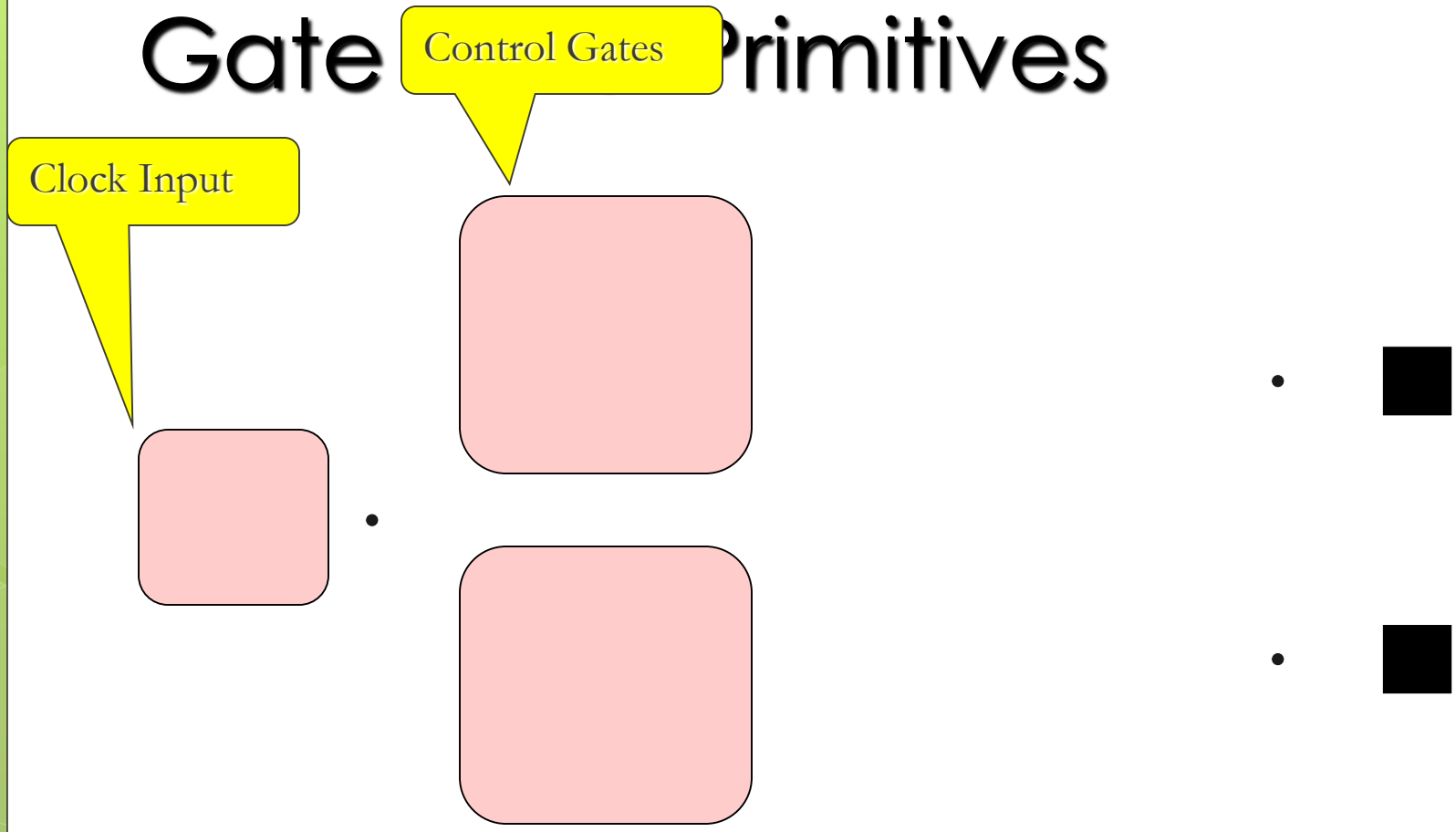
```
`timescale 1ns/100ps

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

q and *q_b* outputs are initially *X* and remain at this ambiguous state for as long as *s* and *r* remain 0.

- SR-Latch Verilo

Simultaneous assertion of both inputs results in loss of memory.

Gate Primitives



- All NAND Clocked SR-Latch

Gate Level Primit

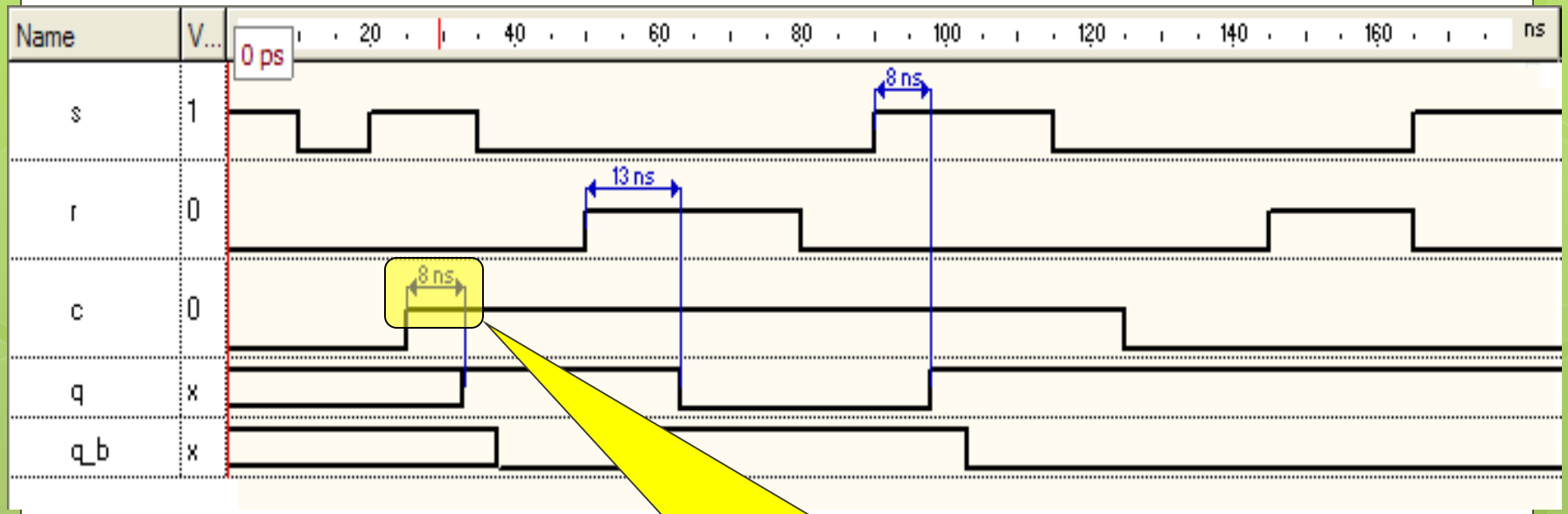
Delay values can be controlled when the latch is instantiated.

```
`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
```

Set and Reset inputs to the cross_coupled core of this memory element

- All NAND Clocked Latch

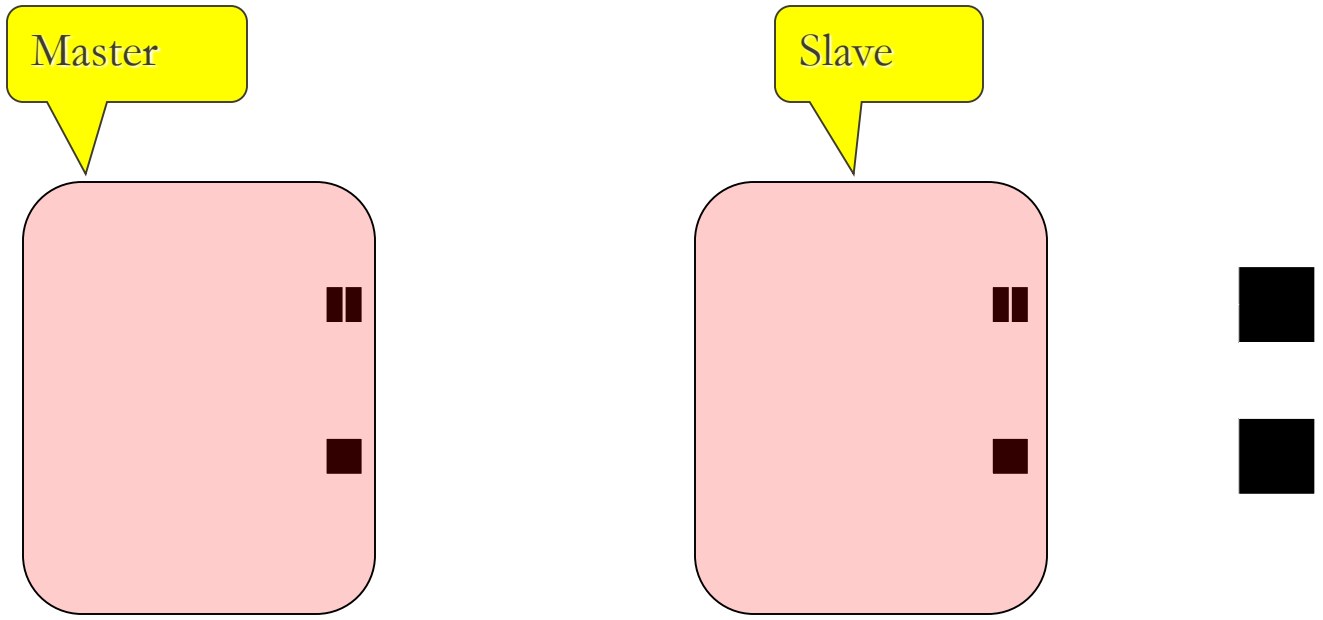
Gate Level Primitives



- SR Latch Simulation

This delay is due to a fall of 3ns and a rise of 5 ns in the NAND gates of the circuit.

Gate Level Primitives



- Master-Slave D Flip-Flop

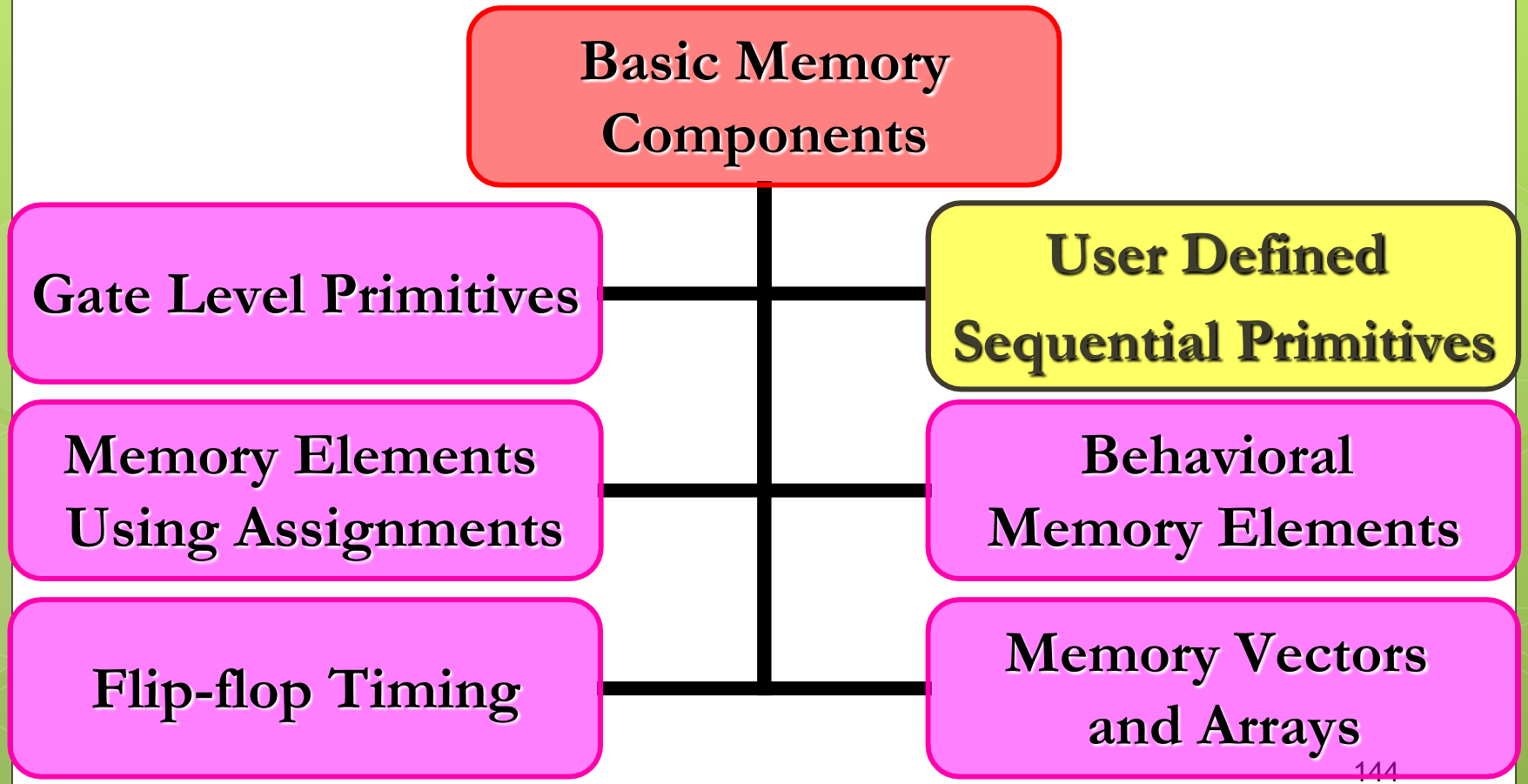
Gate Level Primitives

```
`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
```

Hierarchical Naming

- Master-Slave D Flip-Flop Verilog Code

User Defined Sequential Primitives



User Defined Sequential Primitives

- Verilog provides language constructs for defining sequential UDPs:
  - Faster Simulation of memory elements
  - Correspondence to specific component libraries

User Defined Sequential Primitives

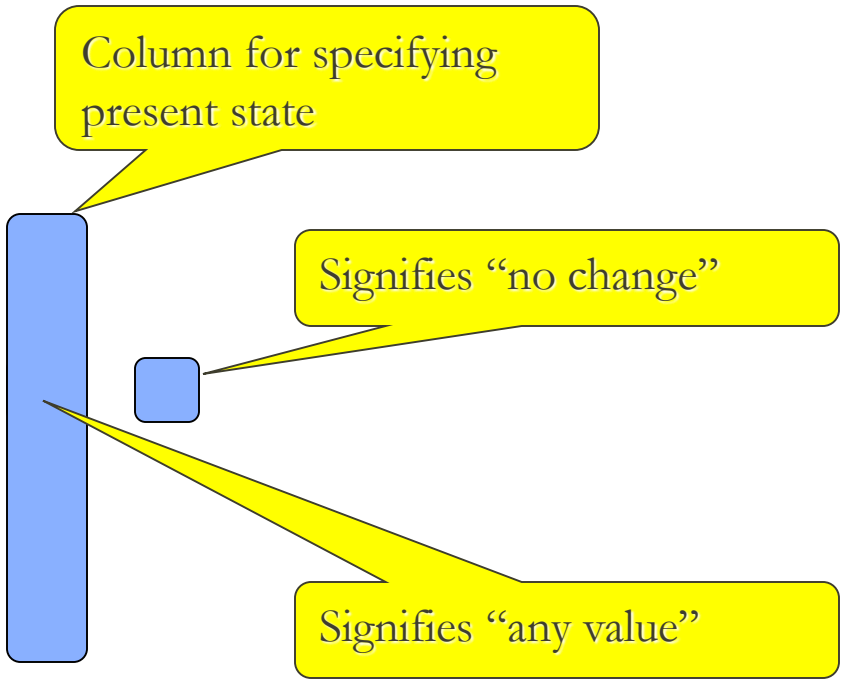
```
primitive latch( q, s, r, c );
  output q;
  reg q;
  input s, r, c;
  initial q=1'b0;
  table
//   s r c   q   q+ ;
//   ------:---:----;
     ? ? 0 : ? : -  ;
     0 0 1 : ? : -  ;
     0 1 1 : ? : 0  ;
     1 0 1 : ? : 1  ;
  endtable
endprimitive
```

Table defining the latch output

- Sequential UDP Defining a Latch

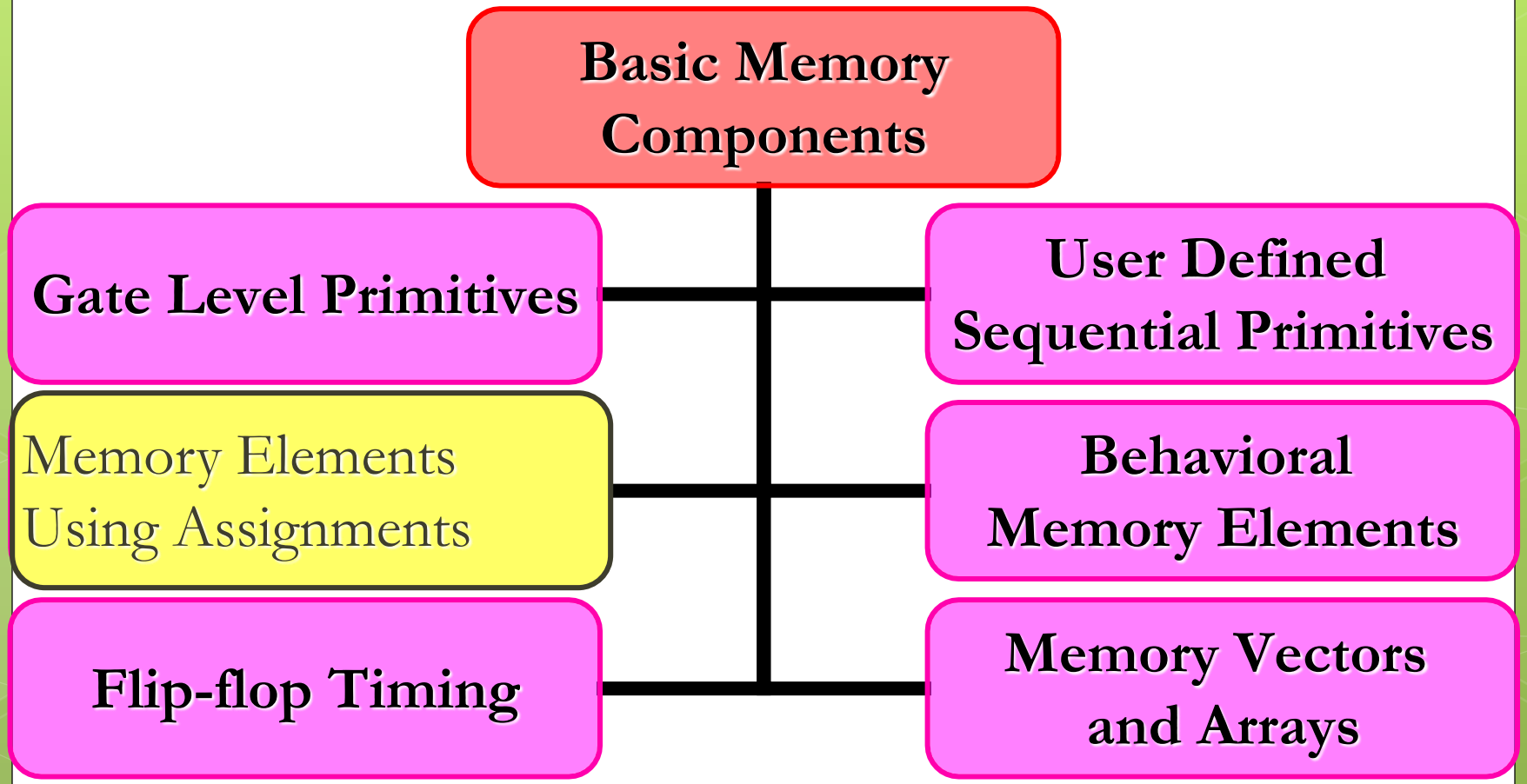
User Defined Sequential Primitives

```
primitive latch( q, s, r, c );
     .............
     .............
table
//    s r c   q   q+ ;
//    ------:---:----;
      ? ? 0 : ? : - ;
      0 0 1 : ? : - ;
      0 1 1 : ? : 0 ;
      1 0 1 : ? : 1 ;
   endtable
endprimitive
```



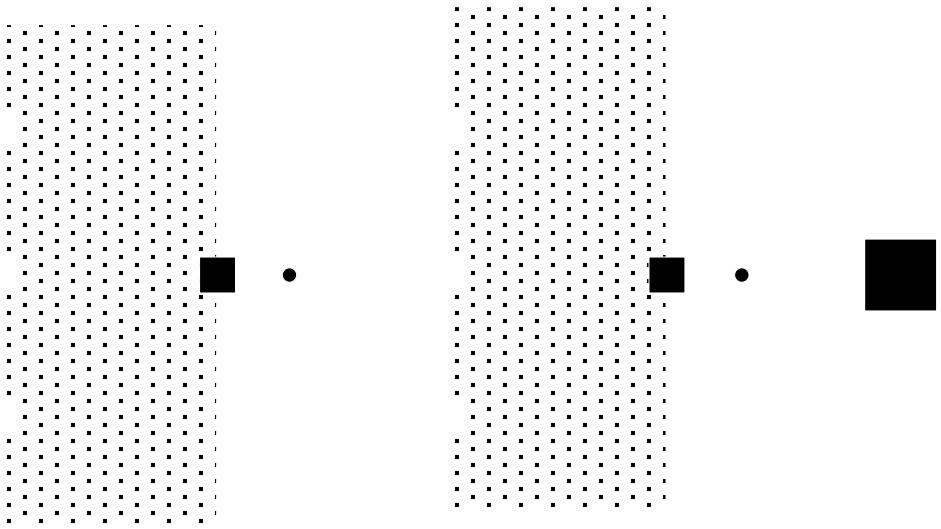
- Sequential UDP Defining a Latch

Memory Elements Using Assignments



ry Elements Using ments

When a block's clock input is 0, it puts its output back to itself (feedback), and when its clock is 1 it puts its data input into its output.



- Master-Slave Using Two Feedback Blocks

Memory Elements Using Assignments

```
`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
```

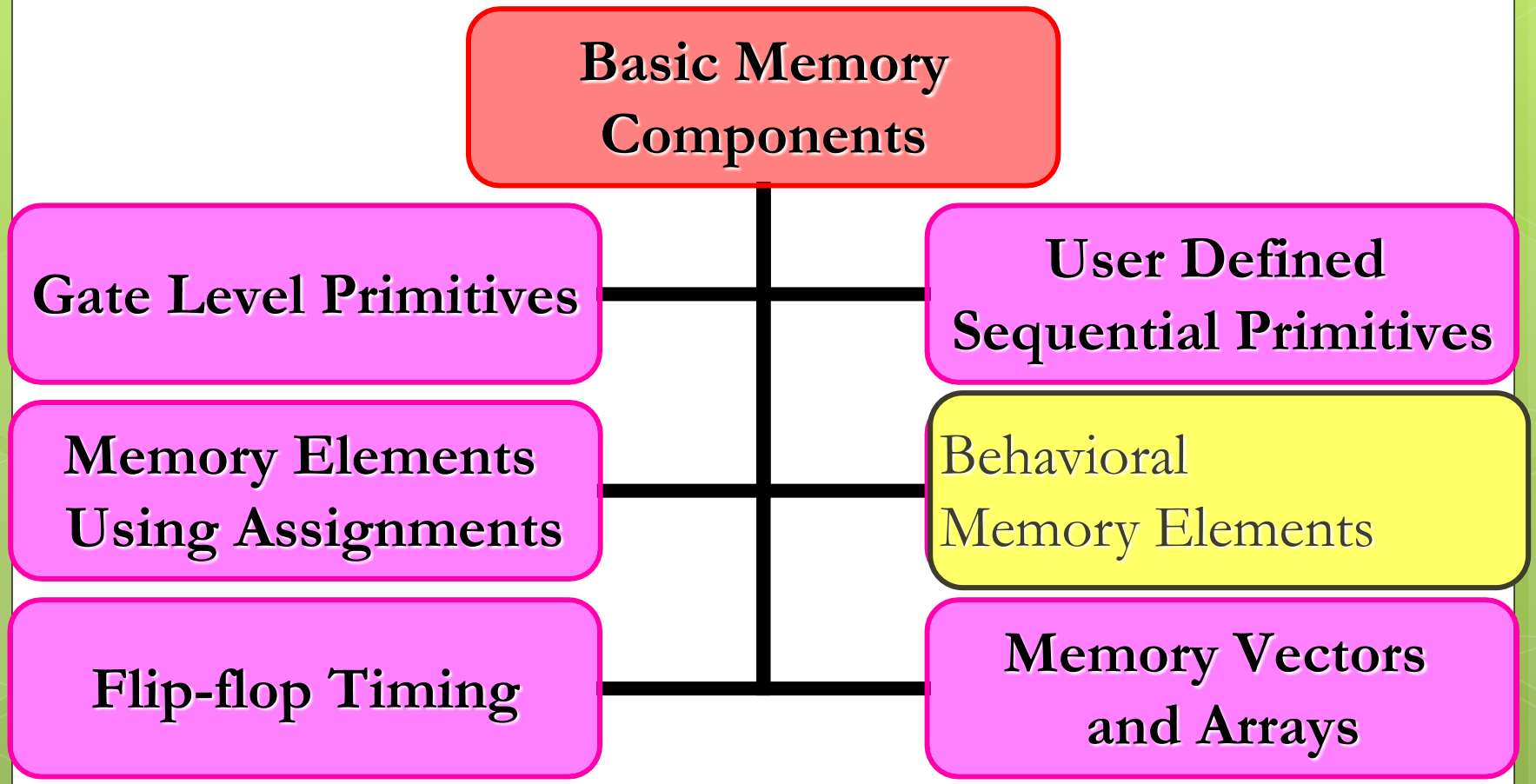
The feedback of *qm* output back to its input

Each *assign* statement implements a latch

Complementary Clocks: Implements master-slave flip-flop

- Assign Statements Implementi

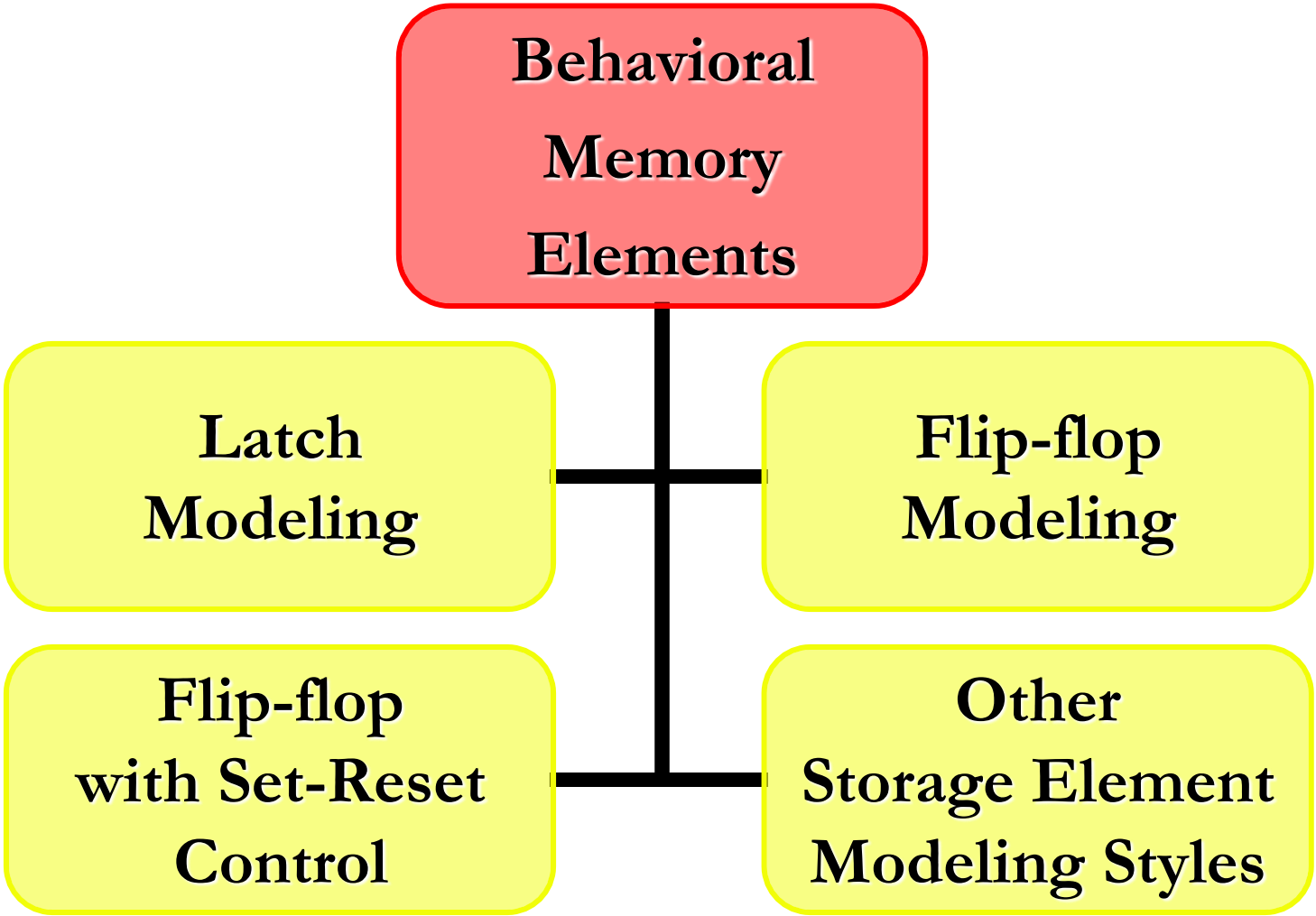
Behavioral Memory Elements



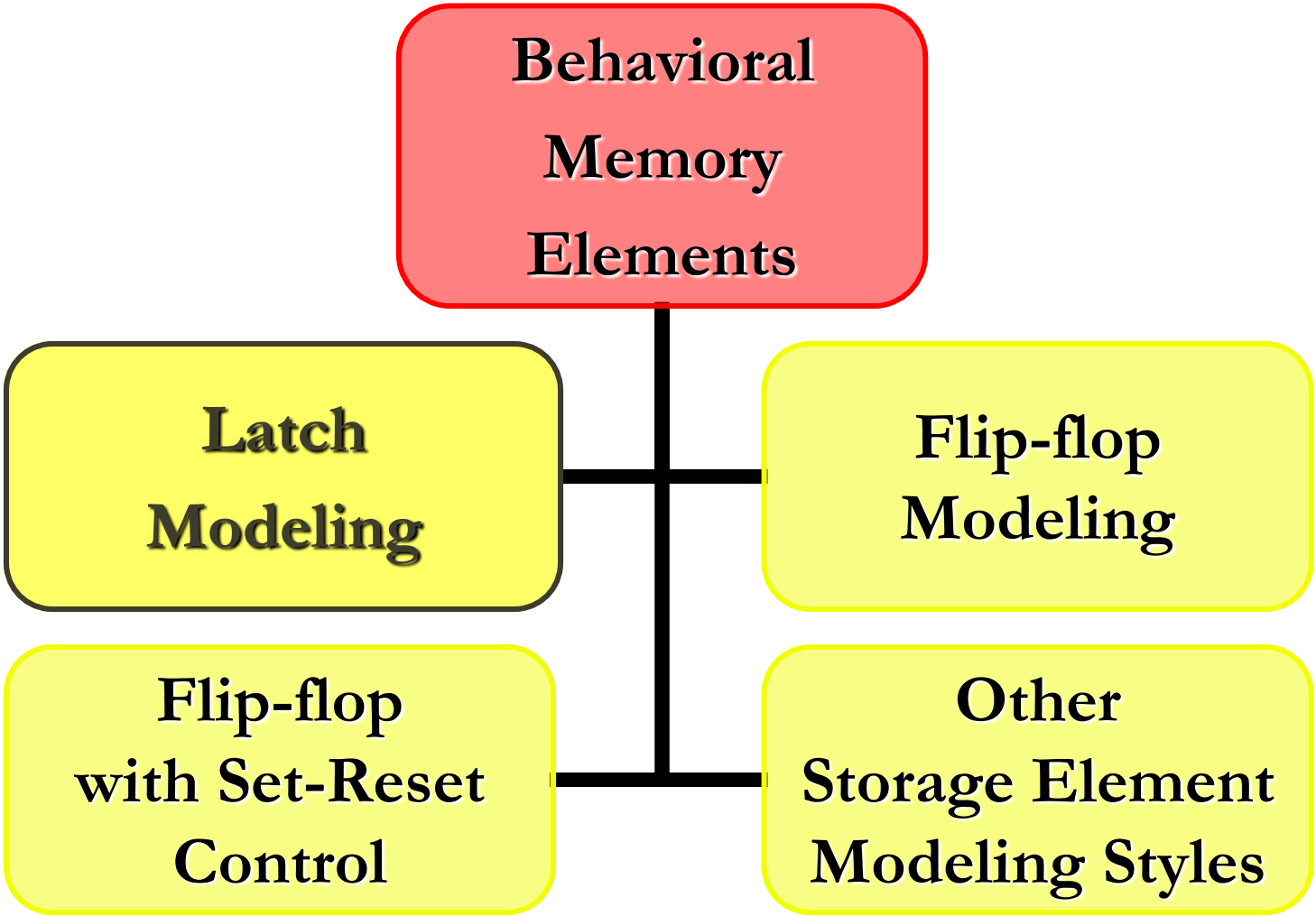
Behavioral Memory Elements

- Behavioral Coding:
  - A more abstract and easier way of writing Verilog code for a latch or flip-flop.
  - 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



Latch Modeling

```
`timescale 1ns/100ps

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
```

While *c* is 1
changes on *d* directly affect *q* and *q_b* outputs.

A Storage unit
Level Sensitive to *c* :
A Latch

After 4ns *d* input is read and assigned to *q* output.

If *d* changes between the time it is read for *q* and *q_b* *erroneous results happen.*

After another wait of 3ns, *d* is read again and ~*d* is assigned to *q_b* output.

- **A D-Type Latch Verilog Code**

Latch Modeling

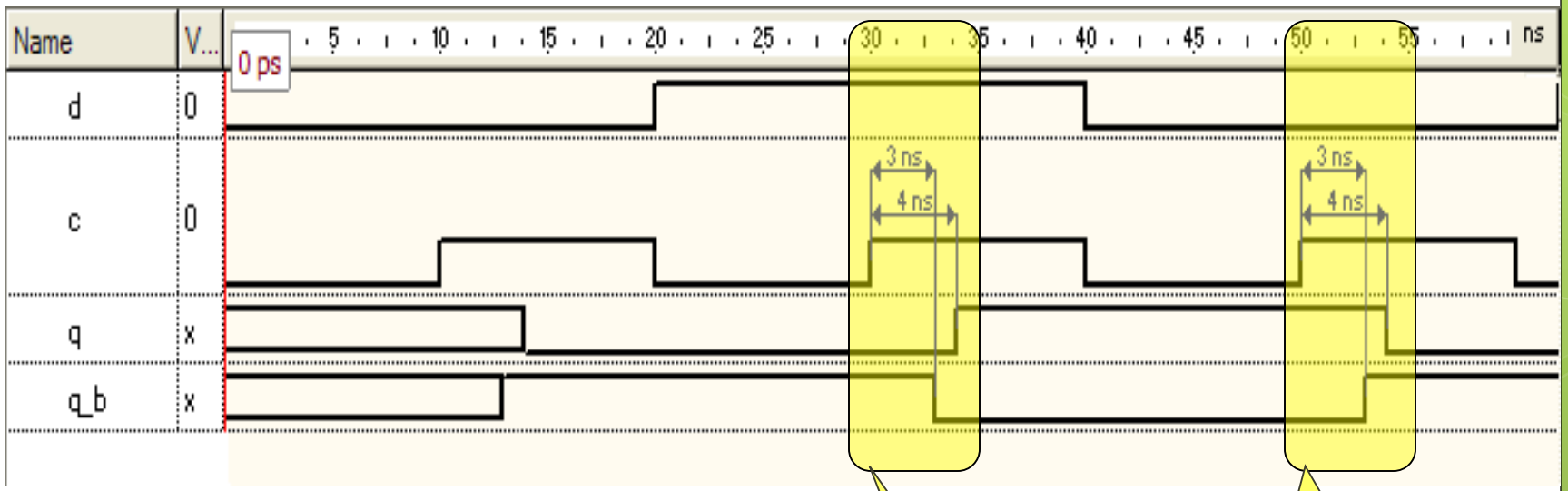
Corrects the timing problem of blocking assignments.

```
`timescale 1ns/100ps

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

Non-blocking assignments With intra-statement delay controls

- **Latch Model Using Nonblocking** Assignments

Latch Modeling

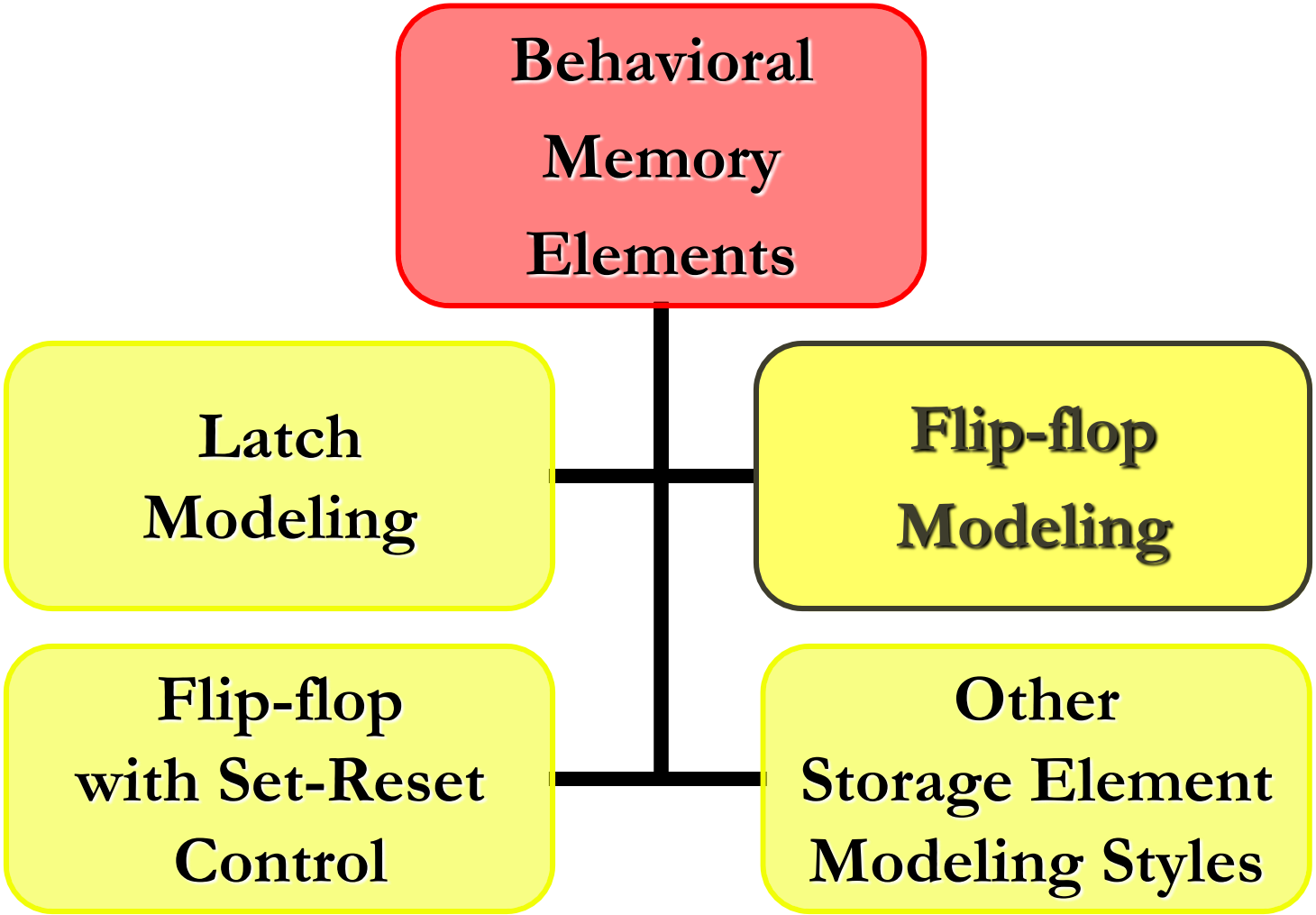


- **Testing Latch with Nonblocking Assignments**

Storing a 1 at time 30

Storing a 0 at time 50

Flip-flop Modeling



Modeling

With each clock edge, the entire procedural block is executed once from begin to end.

A basic edge trigger flip-flop model at the behavioral level

```
`tim    le 1ns/100ps

mod    e d_ff (input d, clk, output reg q, q_b );
   alw    ys @( posedge clk )
   beg    in
      q   <= #4  d;
      q_b <= #3 ~d;
   end
endmodule
```

Sensitive to the positive edge of the clock

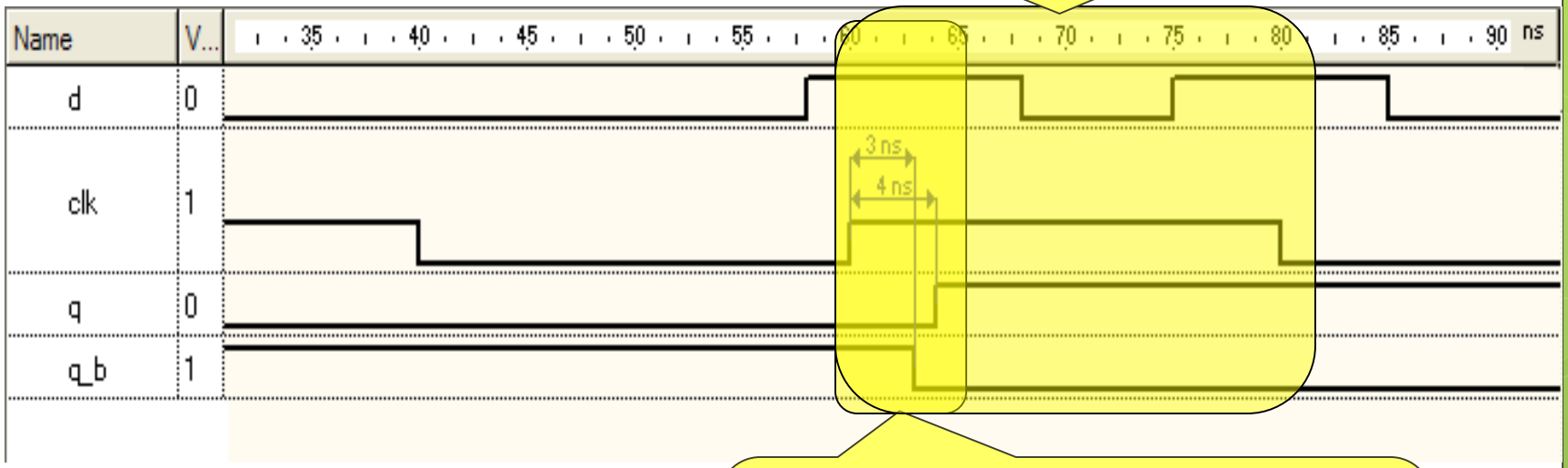
Assignments to *q* and *q_b* are reached immediately after the flow in *always* block begins.

The actual assignments of values are delayed.

- Positive Edge Trigger Flip-Flop

Flip-flop Modeli

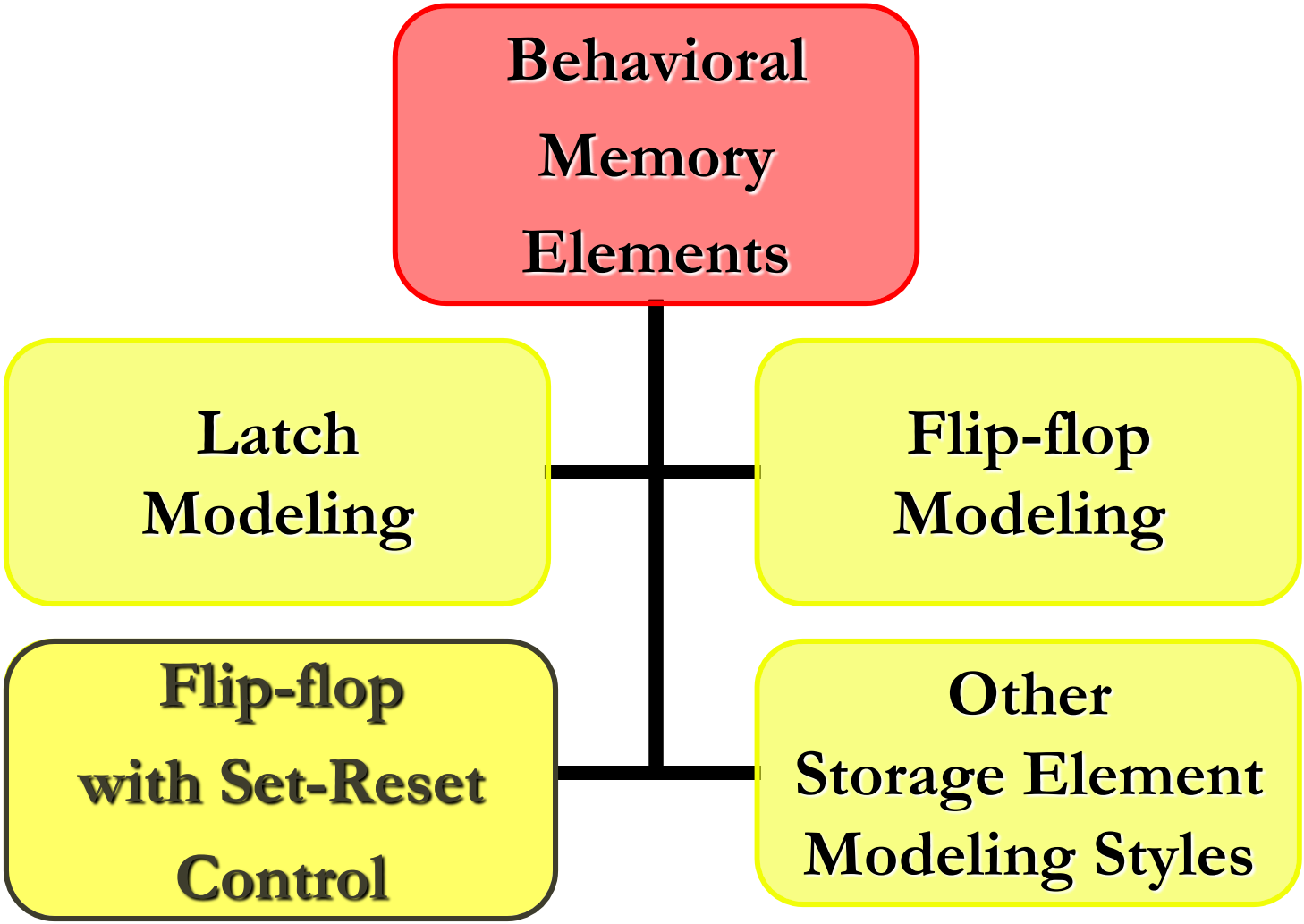
During the time *clk* is 1 (from 60ns to 80ns exclusive of 60 and inclusive of 80), changes on *d* do not affect the state of flip-flop



- **Simulation of a Positive Edge Flip-F**

At 60ns, on the positive edge of clock, the value of *d* is read and scheduled into *q* and *q_b* for times 64ns and 63ns respectively.

Flip-flop with Set-Reset Control



Flip-flop With Set-Reset Control

```
`timescale 1ns/100ps

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
```

- D Type Flip-Flop with Synchronous Control

Flip-flop With Set-Reset Control

```
module d_ff_sr_Synch (input d, s, r, clk,
                                    output reg q, q_b );
  always @(posedge clk) begin
    if( s ) begin
            .................
          end else if( r ) begin
      .................
    end else begin
      .................
          end
     end
endmodule
```

The flow into *always* block is only initiated by the *posedge* of *clk*

These if-statements with *s* and *r* conditions are only examined after the positive edge of the clock

Synchronous *s* and *r* control inputs

- **D Type Flip-Flop with Synchronous Control (Continued)**

Flip-flop With Set-Reset Control

```
        ..................
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
..................
```

These if-statements with *s* and *r* conditions are only examined after the positive edge of the clock

- D Type Flip-Flop with Synchronous Control (Continued)

Flip-flop With Set-Reset Control

```
`timescale 1ns/100ps
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
```

- D-type Flip-Flop with Asynchronous Control

Flip-flop With Set-Reset 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
            .....................
     end else if( r ) begin
            .....................
     end else begin
            .....................
     end
   end
endmodule
```

The sensitivity list of the *always* block

Asynchronous set and reset inputs

- D-type Flip-Flop with Asynchronous Control (Continued)

Flip-flop With Set-Reset Control

```
.....................
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
        .....................
```

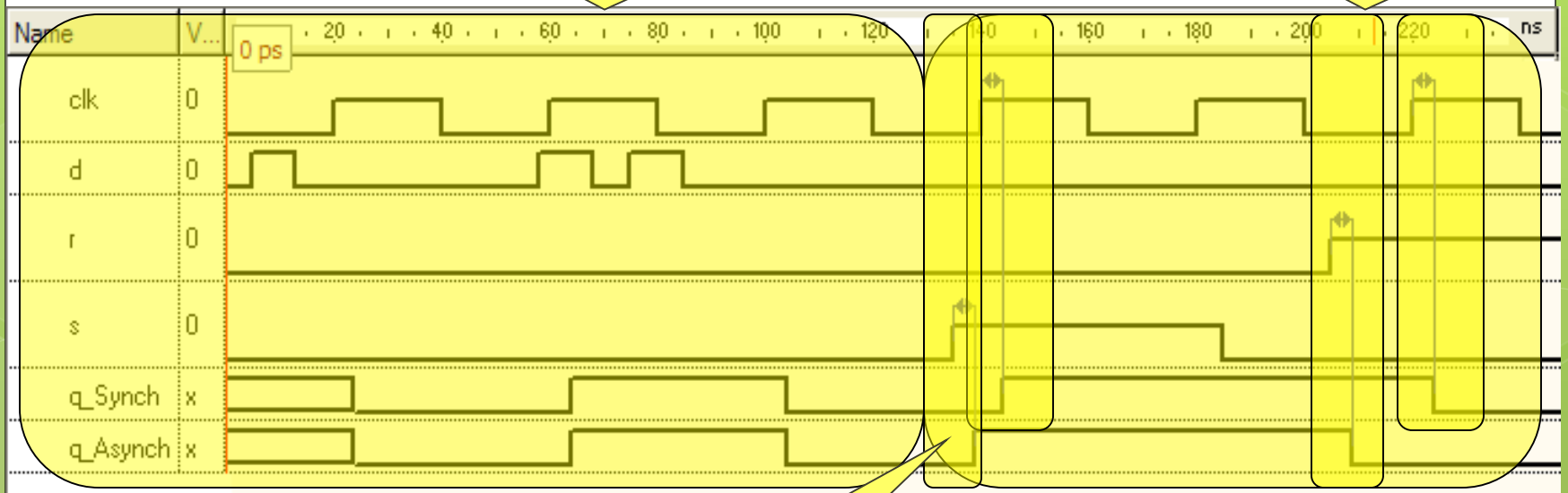
This flip-flop is sensitive to the edge of clock, but to the levels of *s* and *r* .

- D-type Flip-Flop with Asynchronous Control (Continued)

R et-
C

Before 120 ns, changes to q is triggered by the clock and *q_Synch* and *q_Asynch* are the same.

s and *r* become active and cause changes to the flip-flop output.

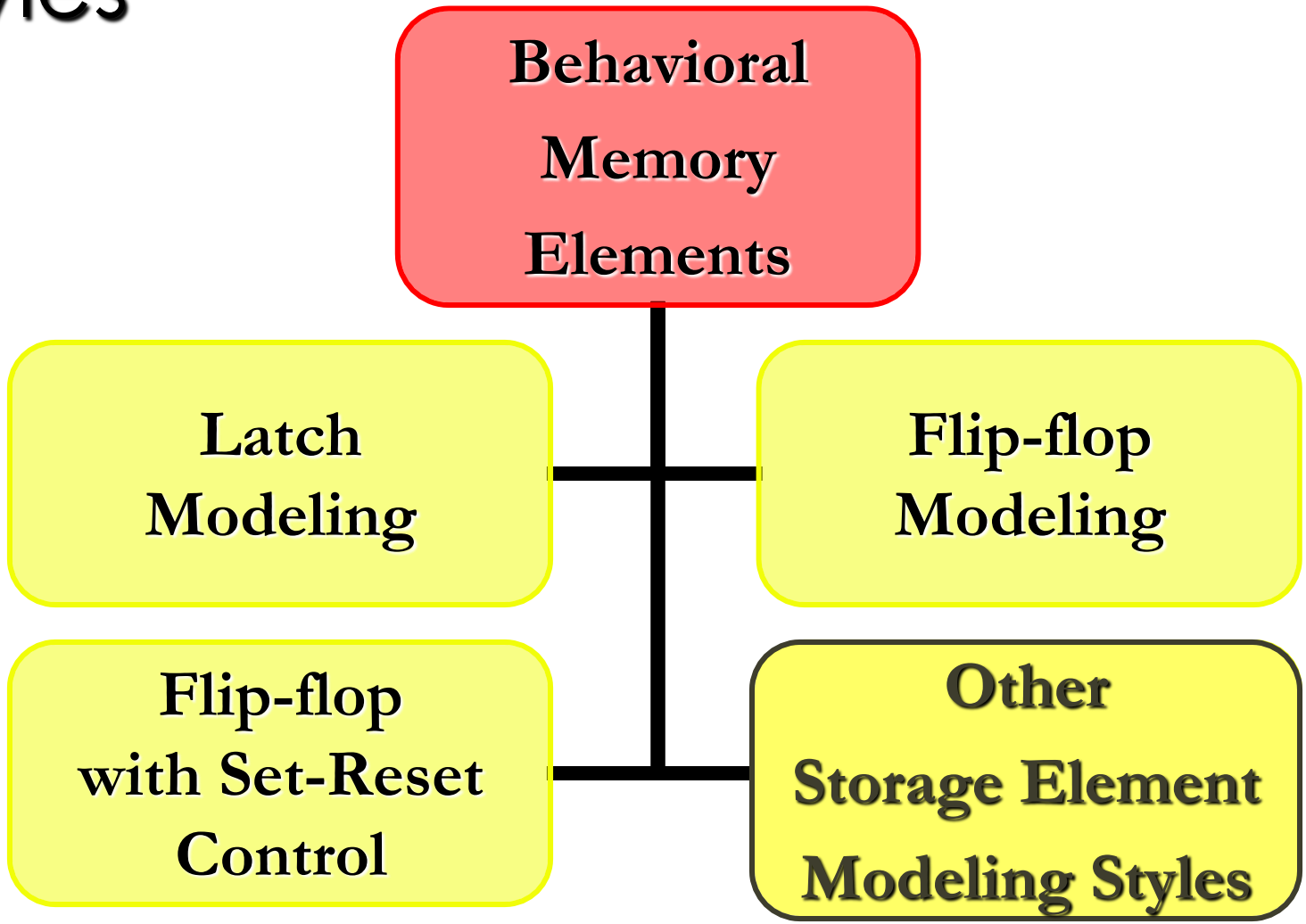


- **Comparing Synchronous and Asy ous Flip**

q_Synch will waits for the edge of the clock to set or reset

q_Asynch changes occur independent of the clock when *s* or *r* becomes active

Other Storage Element Modeling Styles



Other Storage Element Modeling Styles

```
`timescale 1ns/100ps

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

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

A latch using a *wait* statement instead of an event control statement

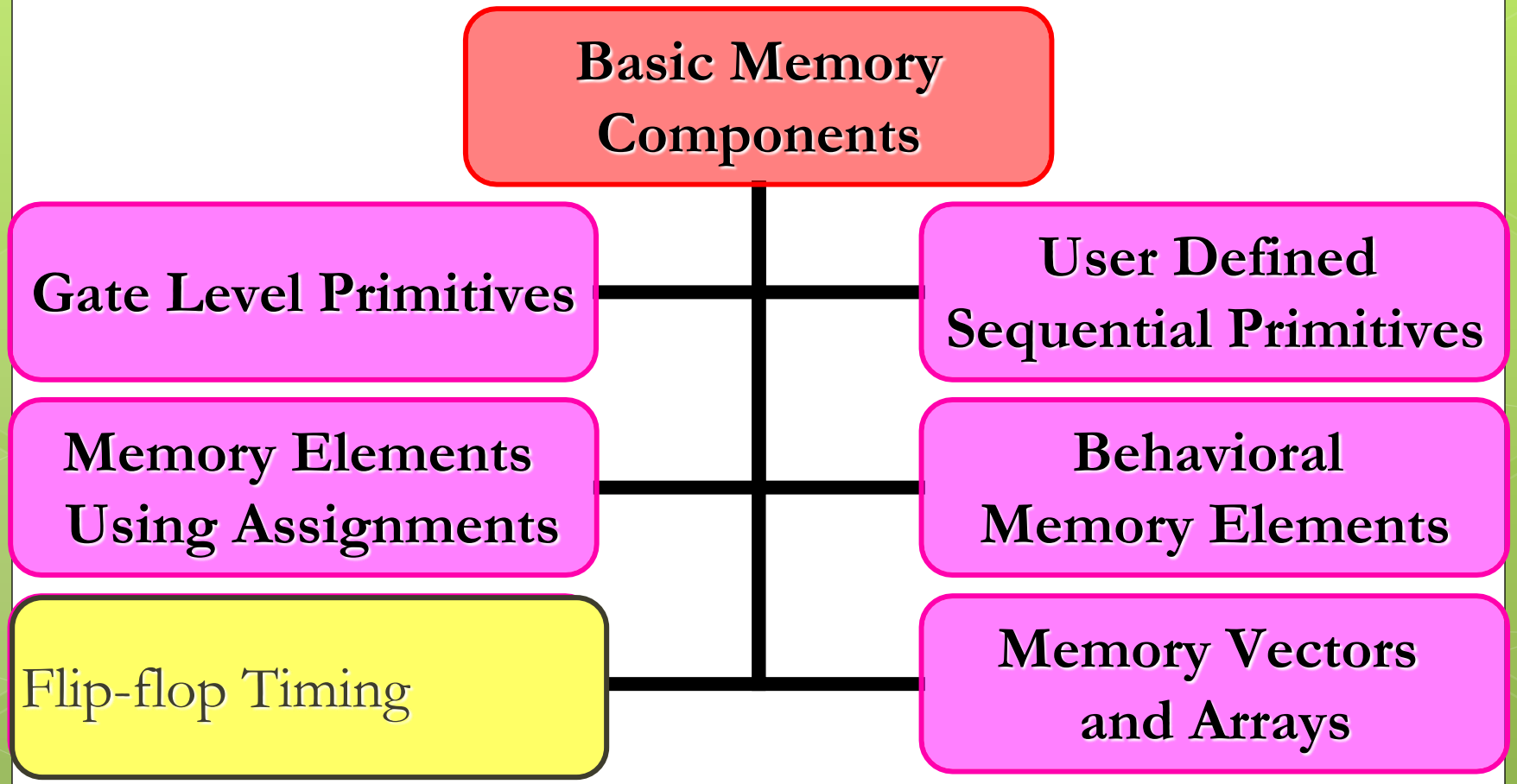
Blocks the flow of procedural block when *c* is 0.

If *c* becomes 1 and remains at this value, the body of the *always* statement repeats itself every 7 ns.

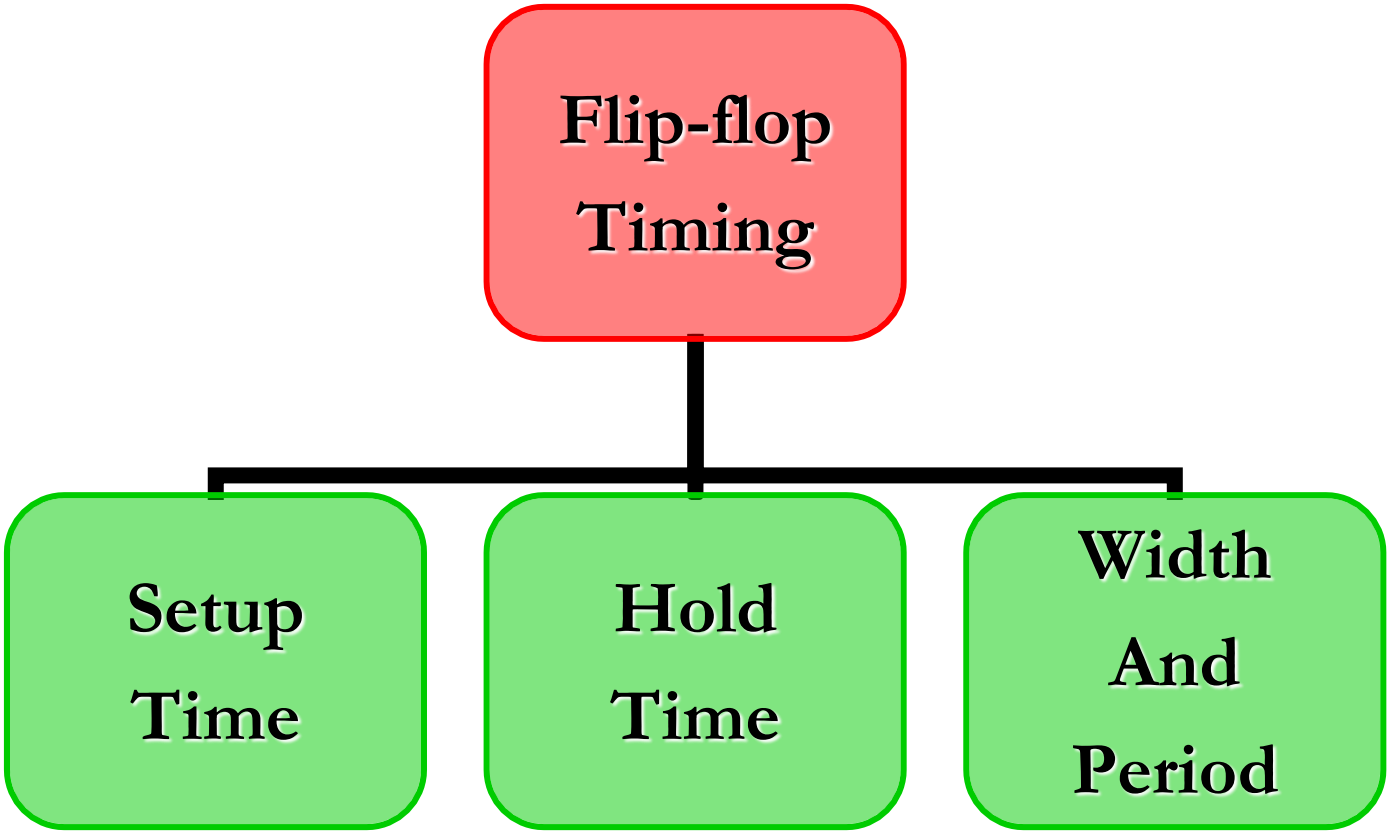
If the delay control statements are omitted, then the looping of the *always* block happens in zero time, causing an infinite loop in simulation.

- **Latch Using wait, a Potentially Danger**

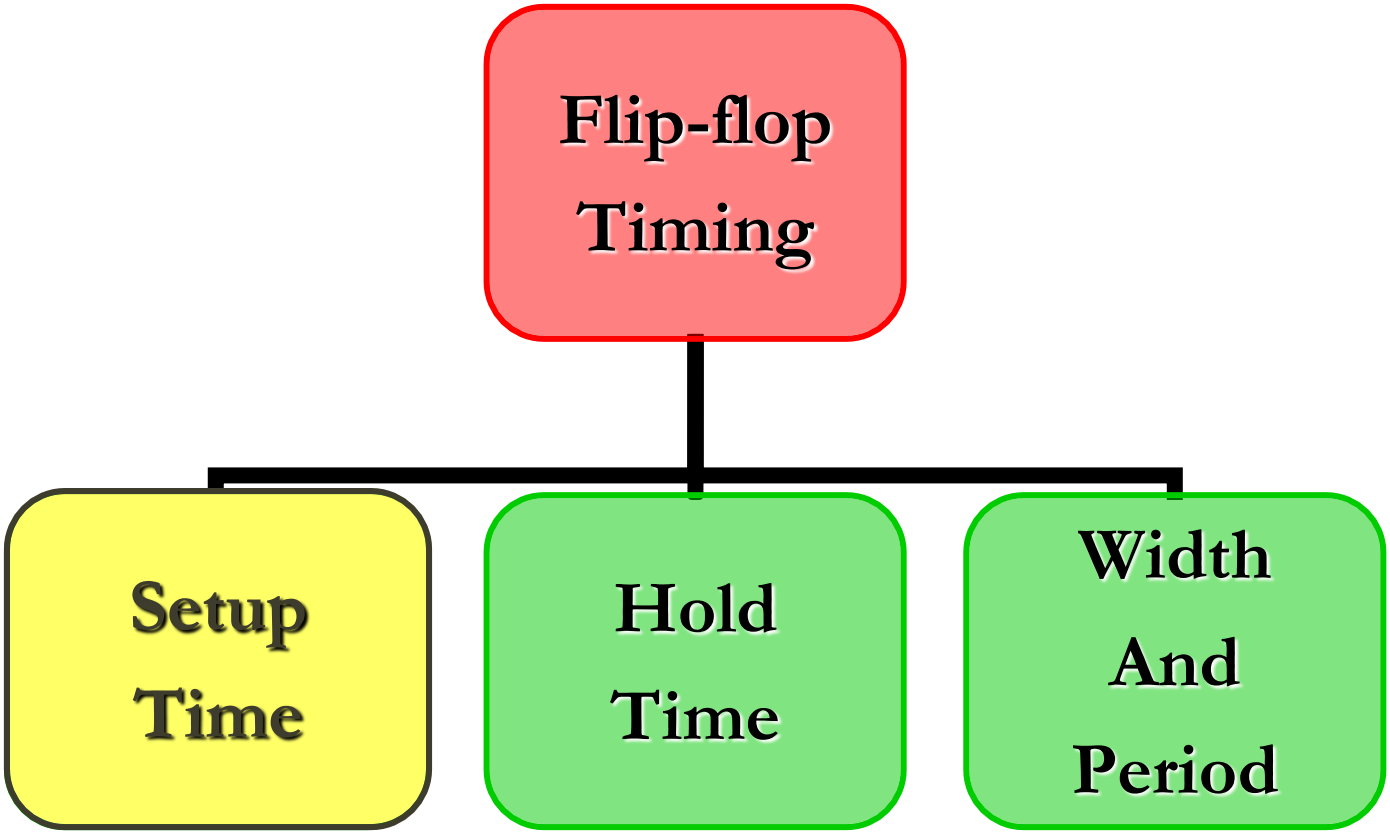
Flip-flop Timing



Flip-flop Timing



Setup Time



Setup Time

- Setup Time
  - The Minimum necessary time that a data input requires to setup before it is clocked into a flip-flop.
- Verilog construct for checking the setup time: $setup task
- The $setup task:
  - Takes flip-flop data input, active clock edge and the setup time as its parameters.
  - Is used within a specify block.

$setup task within a *specify* block

Continuously checks timing distance between changes on *d* and the positive edge of *clk*. If this distance is less than 5ns, a violation message will be issued.

```
`timescale 1ns/
module d_ff ( in    d, clk, s, r, output reg q, q
  specify
    $setup ( d, p   sedge clk, 5 );
  endspecify
  always @( posedge clk or posedge s or posedge r )
   begin
        ..............
   end
endmodule
```

Positive edge trigger flip-flop and Asynchronous set and reset controls

- Flip-Flop **with Setup Time**

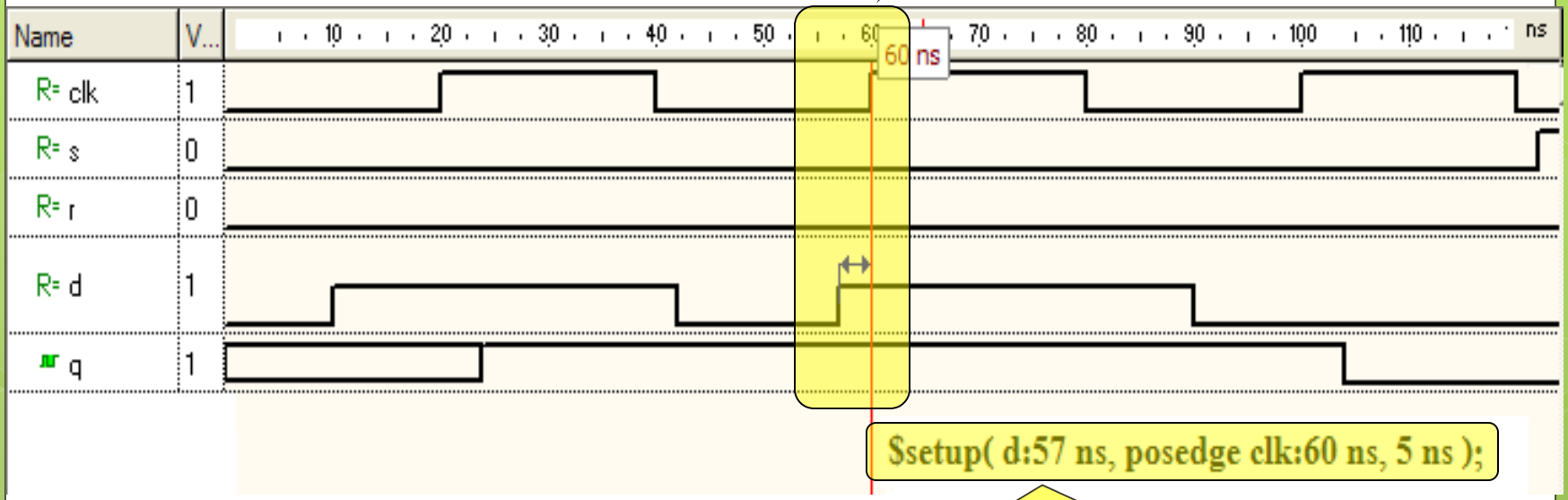
Setup Time

```
.....
always @( posedge clk or posedge s or 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
```

- Flip-Flop with Setup Time (Continued)

Setup Time

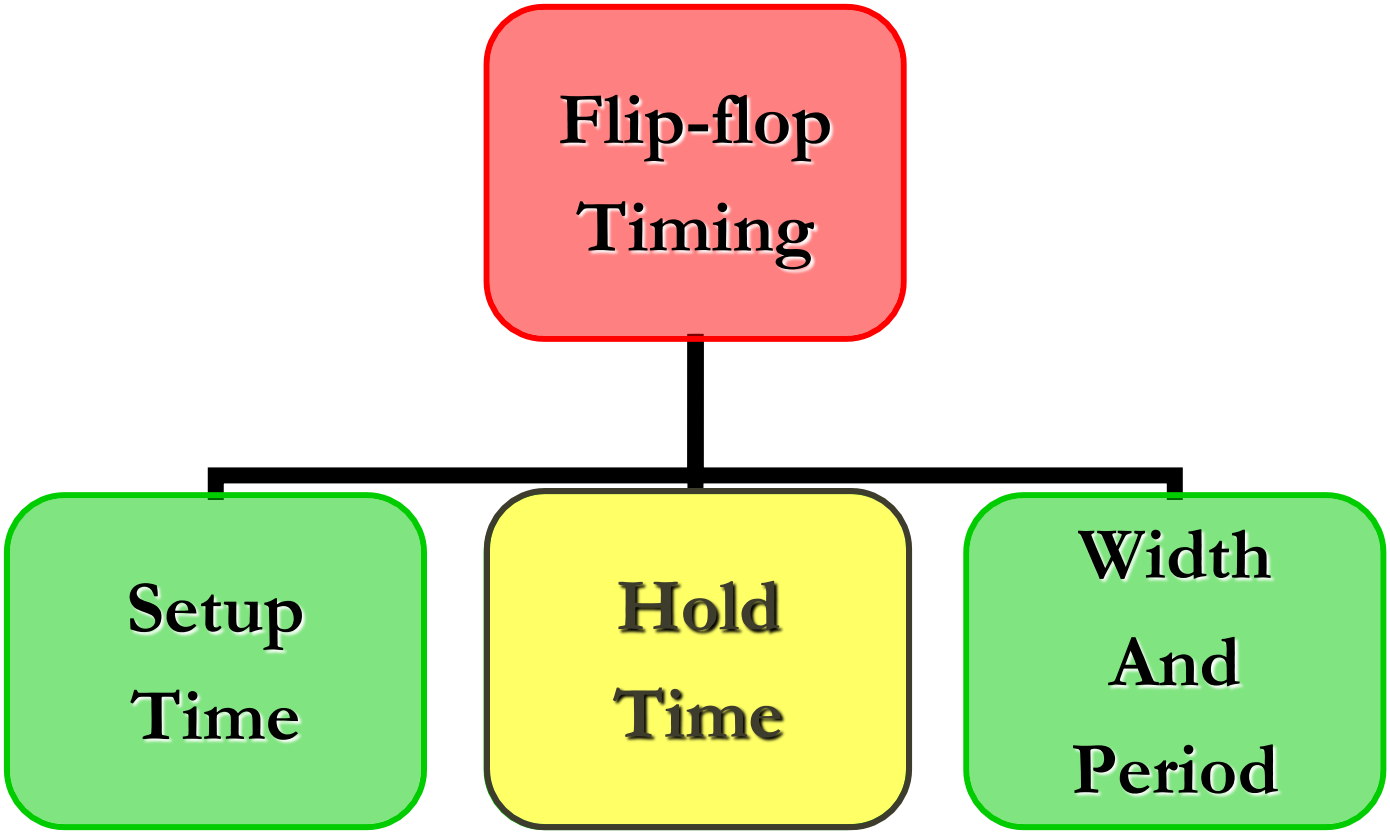
The *d* input changes at 57ns and the data is clocked into the flip-flop at 60ns,
only 3ns after *d*.



- **Setup Time Violation**

The simulation run reports the violation.

Hold Time



Hold Time

- Hold Time
  - The Minimum necessary time a flip-flop data input must stay stable (holds its value) after it is clocked.
- Verilog construct for checking the setup time: $hold task
- The $setup task:
  - Takes flip-flop data input, active clock edge and the required hold time as its parameters.
  - Is used within a specify block.

Hold Time

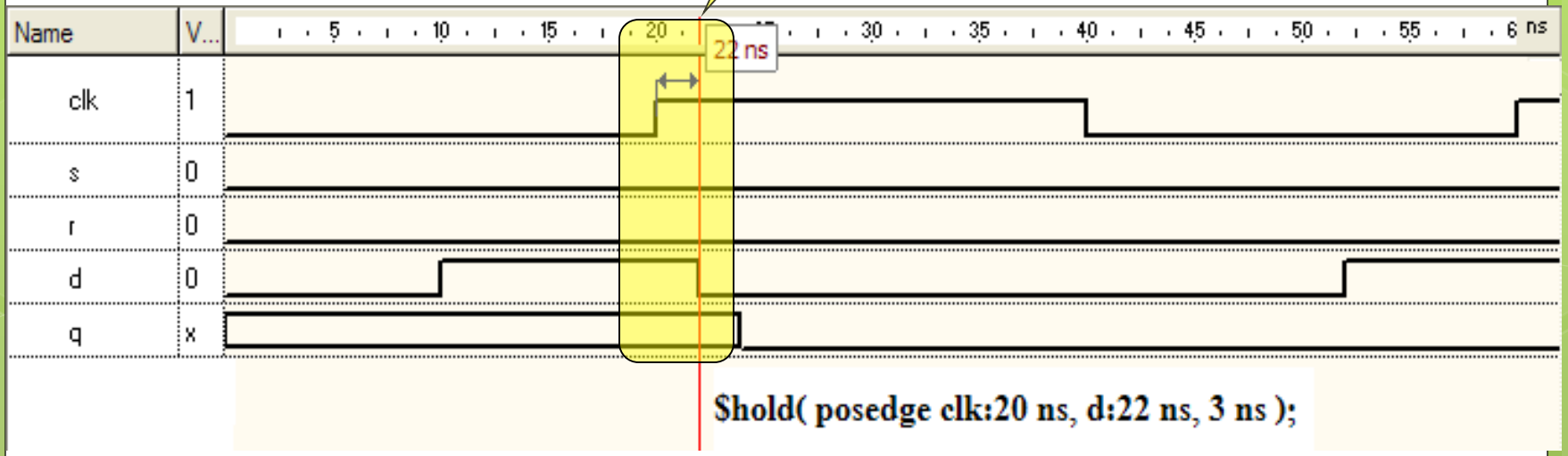
```
`timescale 1ns/100ps

module d_ff ( input d, clk, s, r, output re
  specify
    $hold ( posedge clk, d, 3 );
  endspecify
  always @( posedge clk or posedge s or posedge r )
  begin
    ..........................................
  end
endmodule
```

Flip-flop with hold time of 3ns.

- Flip-Flop **with** Hold **Time**

Hold Time

The clock samples the *d* value of 1 at 20ns. At 22ns, *d* changes. This violates the minimum required hold time of 3ns.

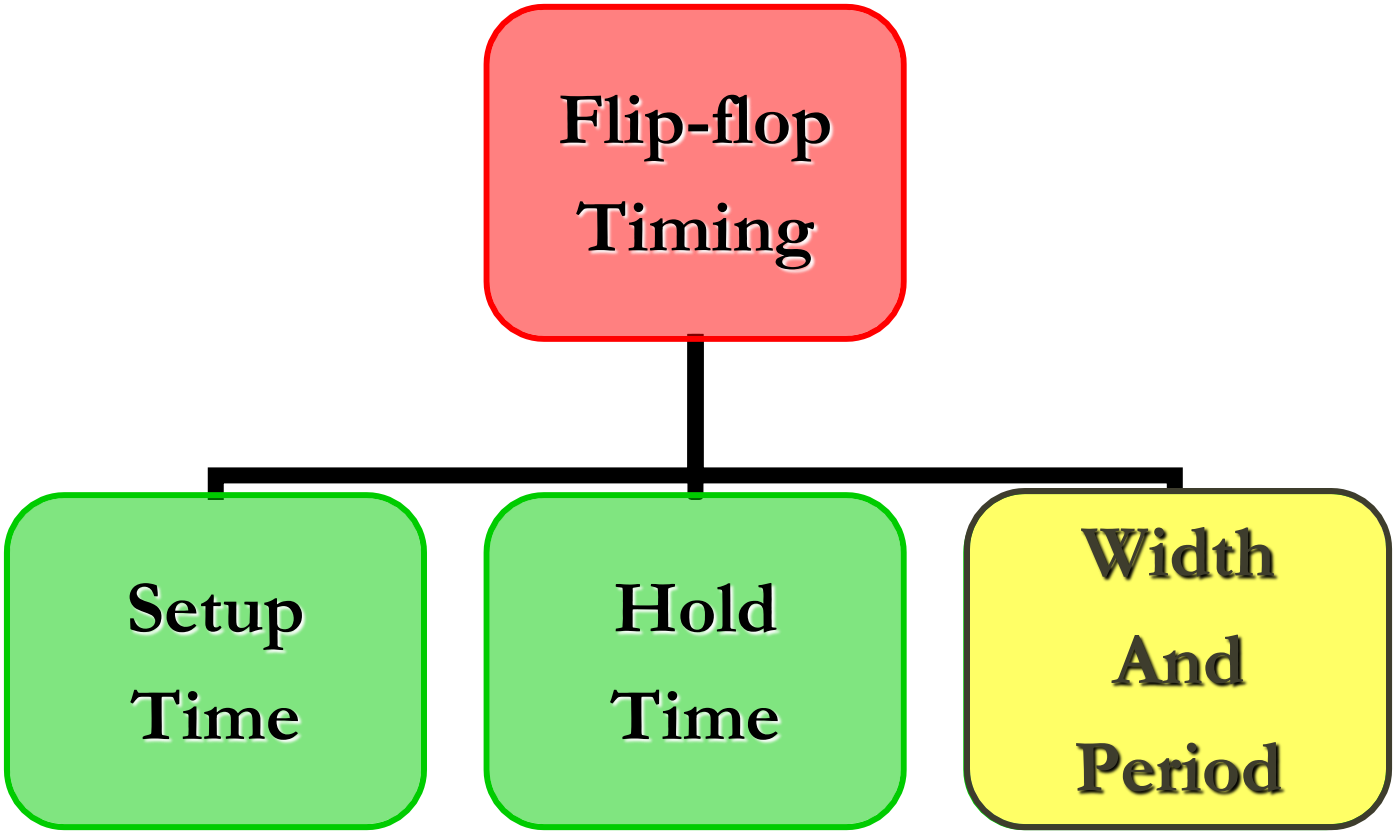


- Hold Time Violation

Hold Time

- The Verilog $setuphold task combines setup and hold timing checks.
- Example:
  - $setuphold (posedge clk, d, 5, 3)

Width And Period



Width And Period

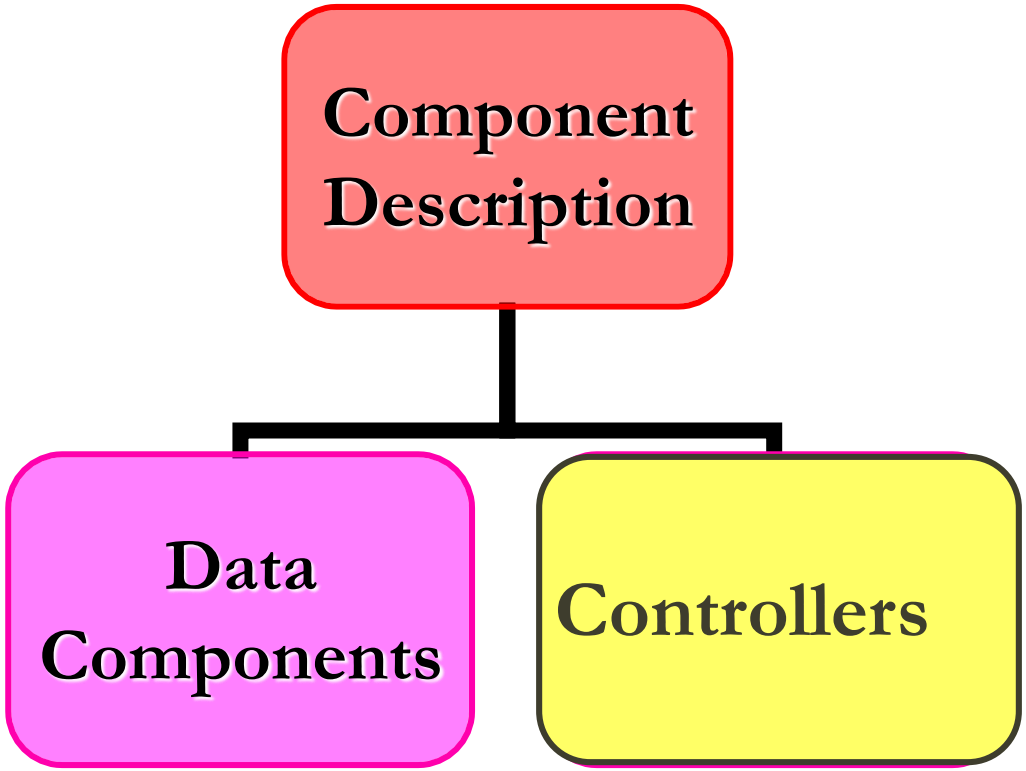
- Verilog $width and $period check for minimum pulse width and period.
- Pulse Width: Checks the time from a specified edge of a reference signal to its opposite edge.
- Period: Checks the time from a specified edge of a reference signal to the same edge.

Width And Period

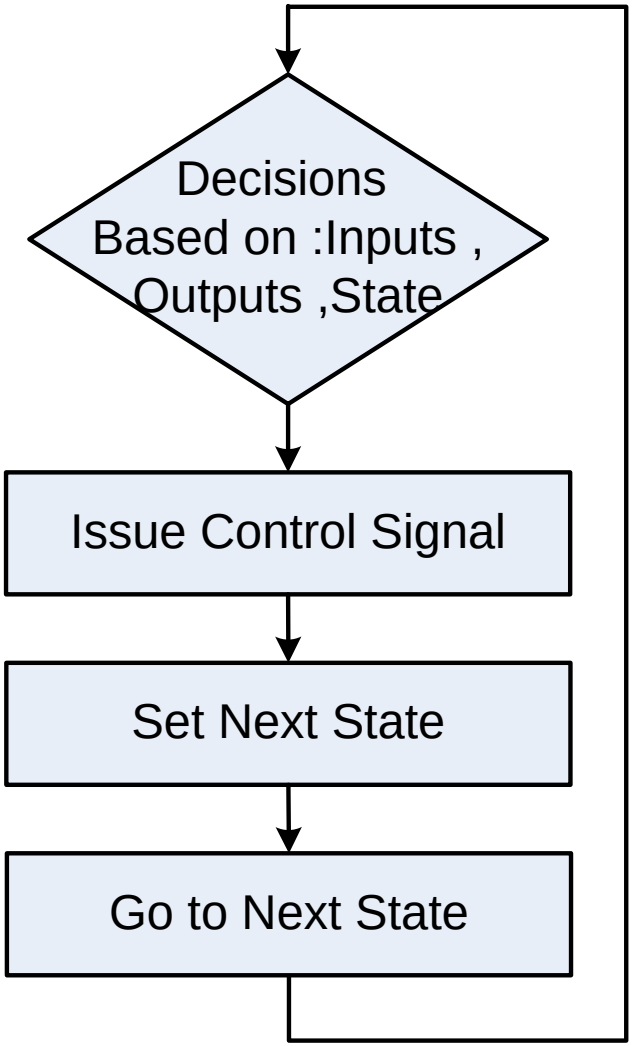
```
specify
   $setuphold ( posedge clk, d, 5, 3 );
   $width  (posedge r, 4);
   $width  (posedge s, 4);
   $period (negedge clk, 43);
endspecify

always @( posedge clk or posedge s or posedge r )
   if( s ) q <= #4 1'b1;
   else if( r ) q <= #4 1'b0;
        else q <= #4 d;
.....
```


- Setup, Hold, Width, and Period Checks (Continued)

Controllers



Controllers

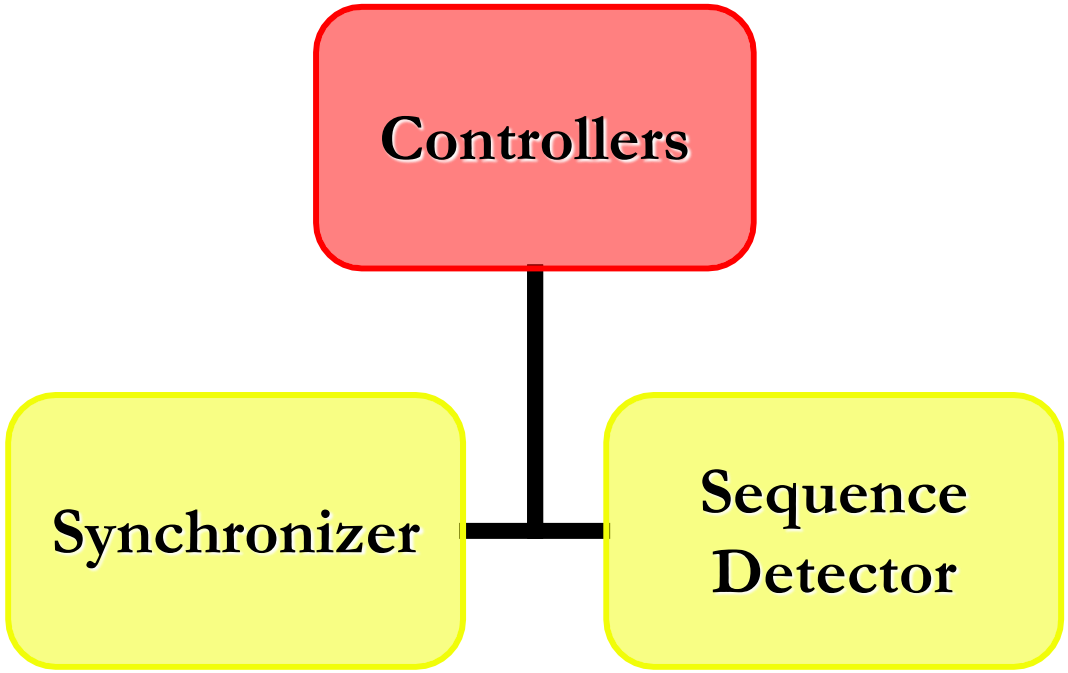


- Controller Outline

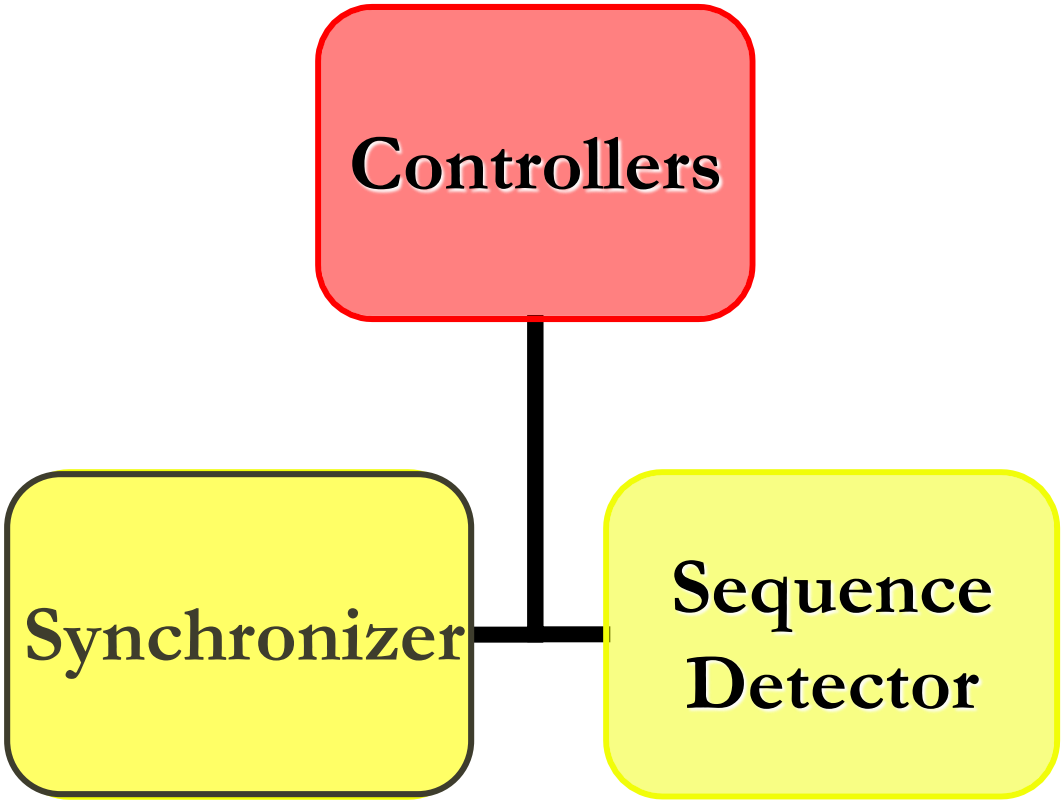
Controllers

- Controller:
  - Is wired into data part to control its flow of data.
  - The inputs to controller determine its next states and outputs.
  - Monitors its inputs and makes decisions as to when and what output signals to assert.
  - Keeps the history of circuit data by switching to appropriate states.
- Two examples to illustrate the features of Verilog for describing state machines:
  - Synchronizer
  - Sequence Detector

Controllers



Synchronizer



Synchronizer

Clk

adata

synched

- Synchronizing *adata*

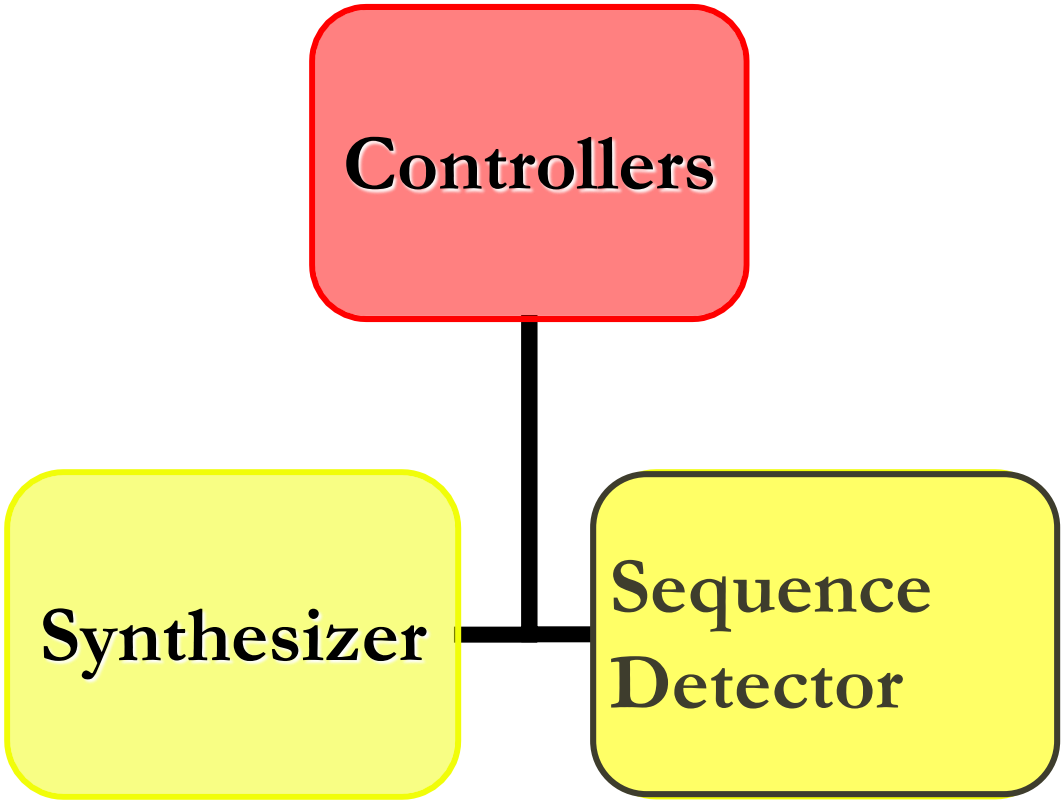
Synchronizer

```
`timescale 1ns/100ps

module Synchronizer (input clk, adata,
                                output reg synched);
    always @(posedge clk)
        if (adata == 0) synched <= 0;
        else synched <= 1;
endmodule
```

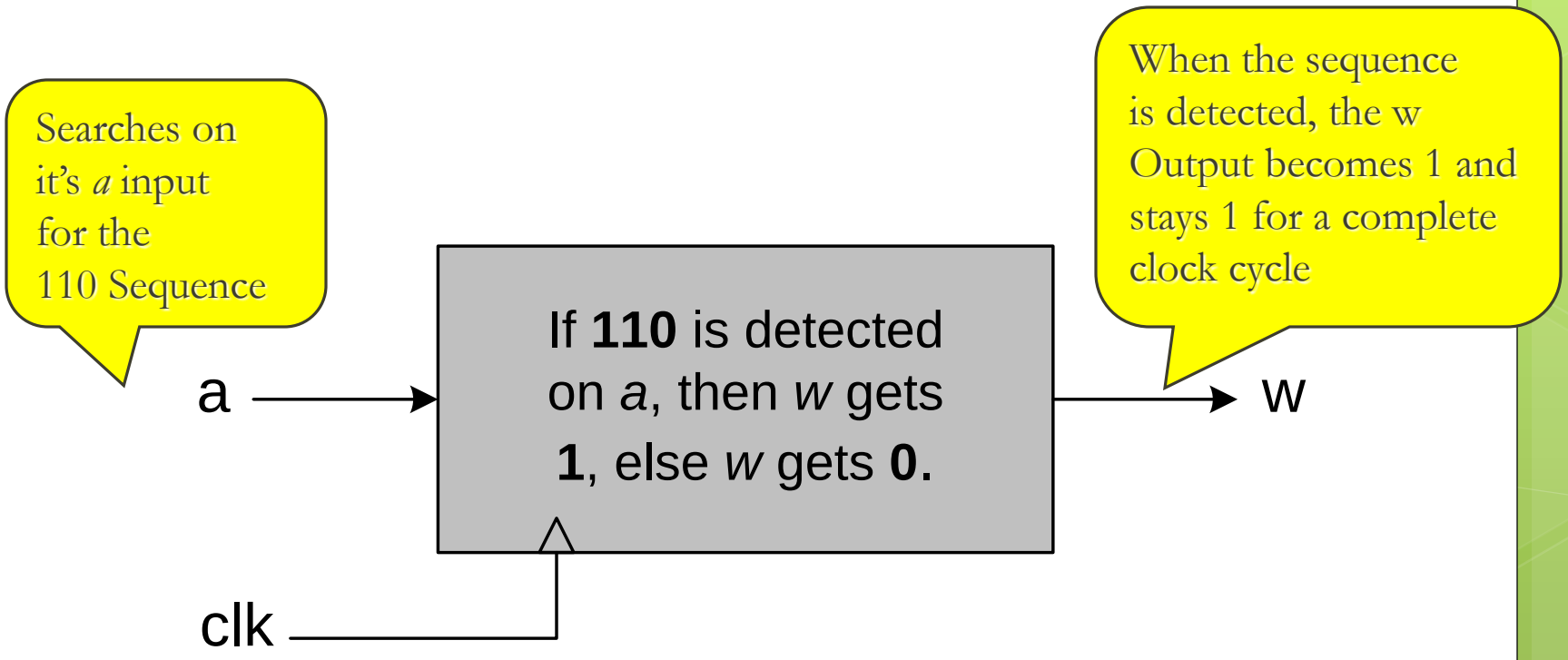
If a *1* is Detected on *adata* on the rising edge of clock, *synched* becomes *1* and remains *1* for at least one clock period

- A Simple Synchronization Circuit

Sequence Detector



Sequence Detector



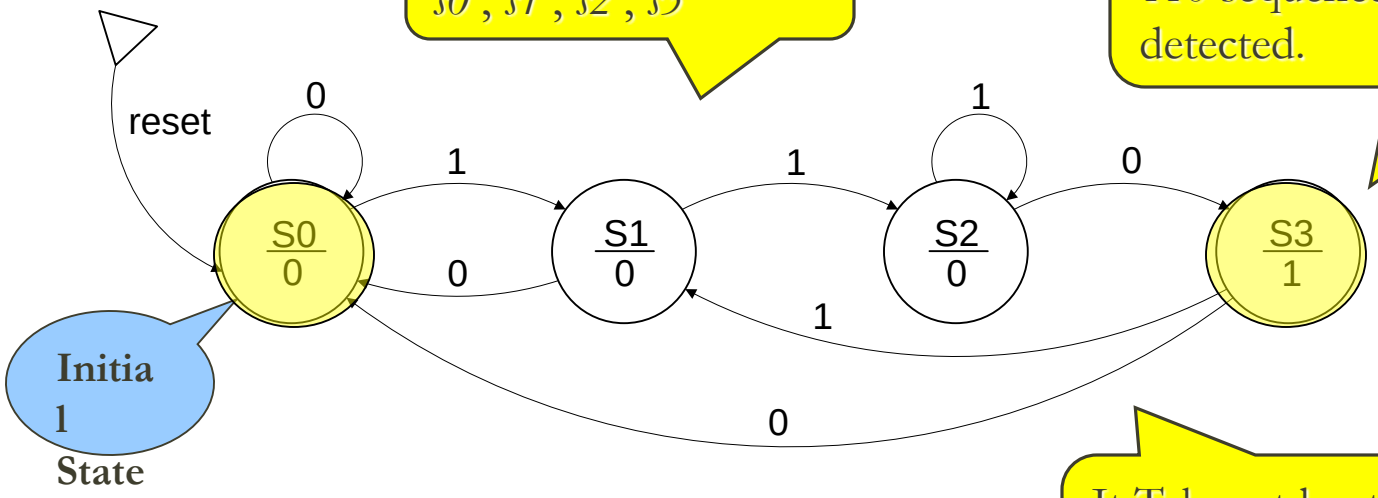
- State Machine Description

Sequence Detector

A Moore Machine Sequence Detector

States are named: *s0* , *s1* , *s2* , *s3*

The State in which the 110 sequence is detected.



It Takes at least 3 clock periods to get to the *s3* state

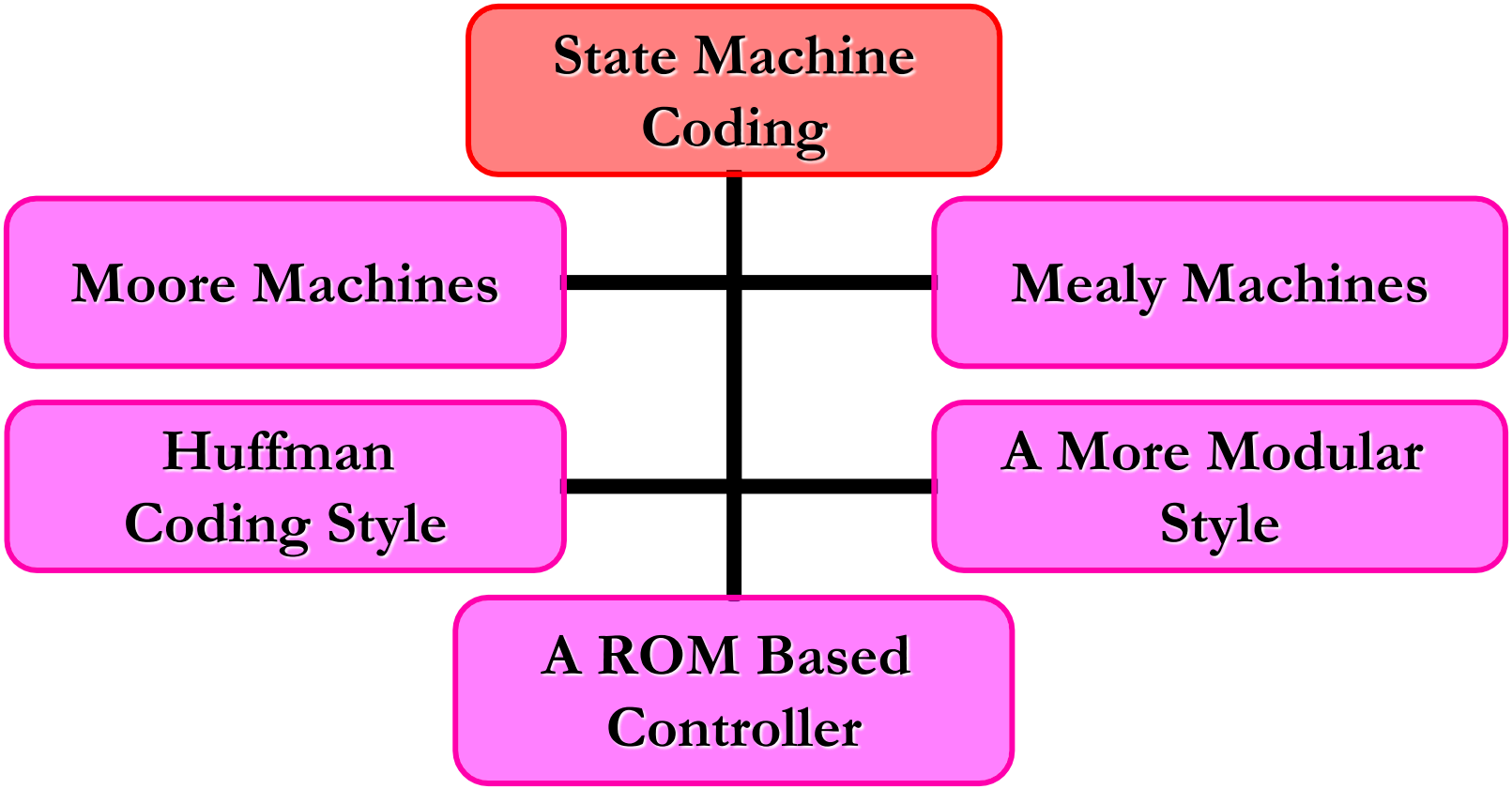
- Sequence Detector State Machine

Sequence Detector

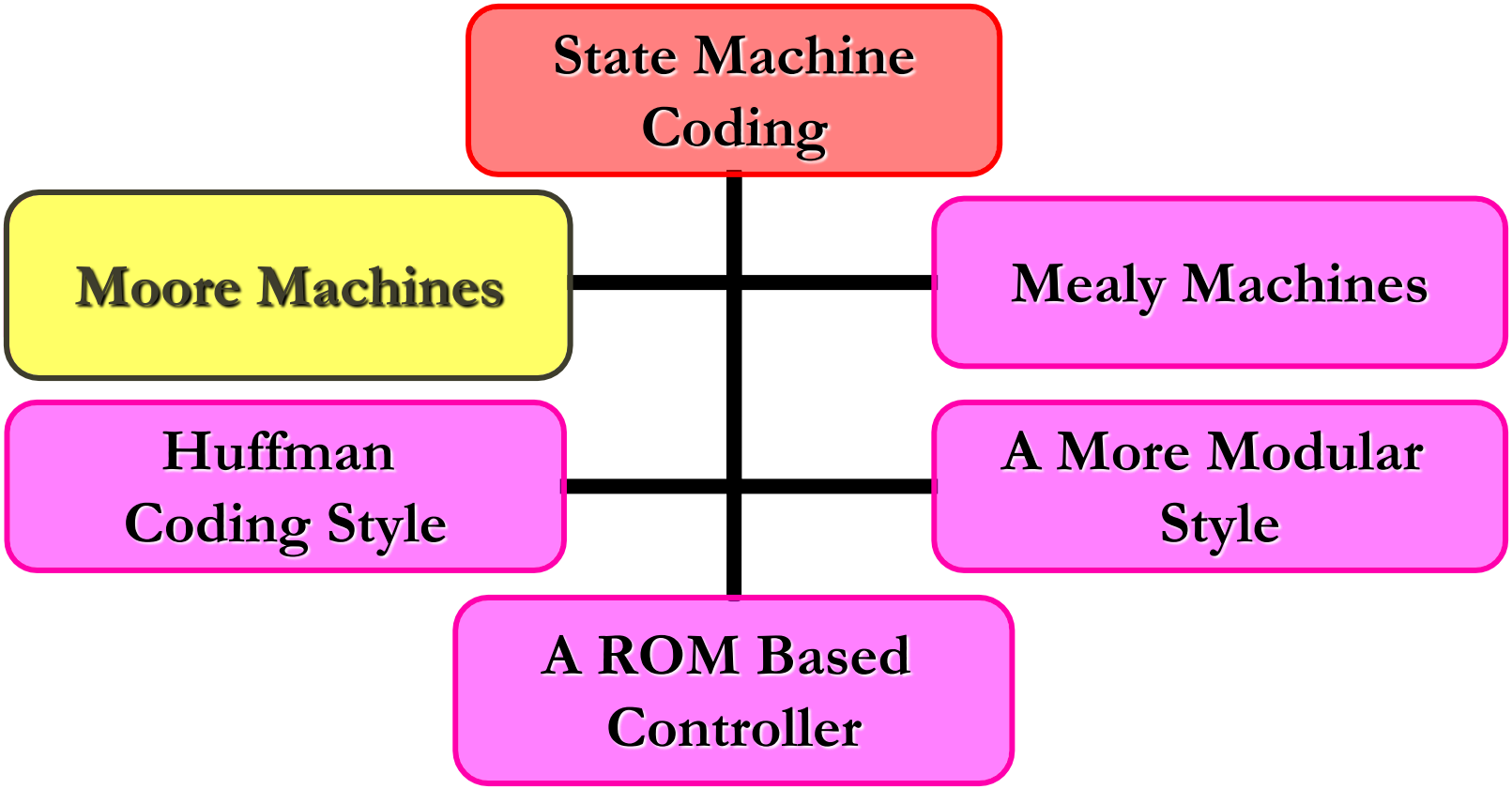
```
module Detector110 (input a, clk, reset, output w);
    parameter [1:0] s0=2'b00, s1=2'b01, s2=2'b10, s3=2'b11;
    reg [1:0] current;

    always @(posedge clk) begin
    if (reset) current = s0;
    else
            case (current)
            s0: if (a) current <= s1; else current <= s0;
            s1: if (a) current <= s2; else current <= s0;
            s2: if (a) current <= s2; else current <= s3;
            s3: if (a) current <= s1; else current <= s0;
            endcase
    end

    assign w = (current == s3) ? 1 : 0;

endmodule
```

- Verilog Code for 110 Detector

State Machine Coding



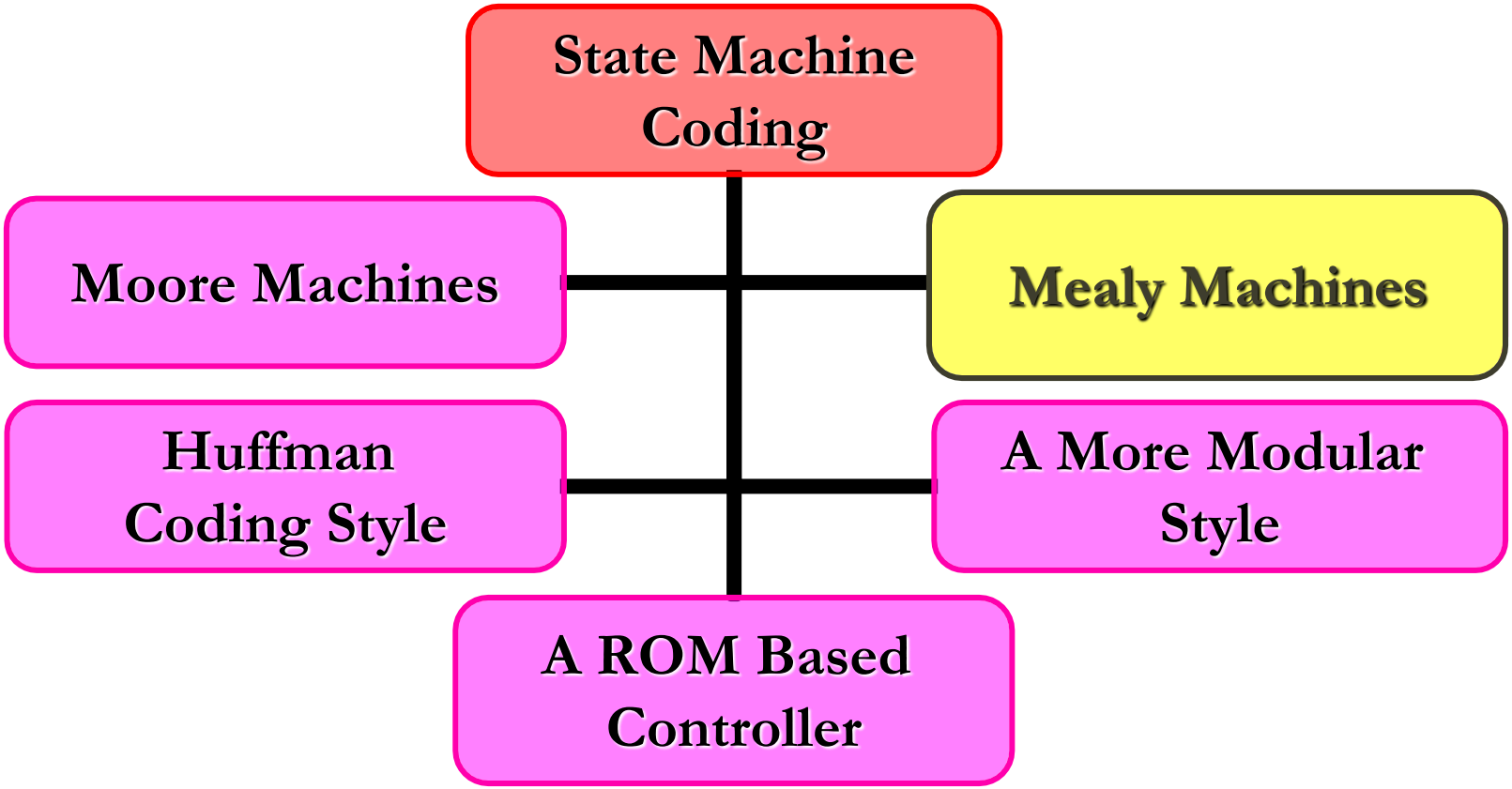
Moore Machines



Moore Machines

- Moore Machine :
  - A state machine in which all outputs are carefully synchronized with the circuit clock.
  - In the state diagram form, each state of the machine specifies its outputs independent of circuit inputs.
  - In Verilog code of a state machine, only circuit state variables participate in the output expression of the circuit.

Mealy Machines



Mealy Machines

- Mealy Machine :
  - Is different from a Moore machine in that its output depends on its current state and inputs while in that state.
  - State transitions and clocking and resetting the machine are no different from those of a Moore machine. The same coding techniques are used.