

# Hardware Description Language

# Verilog

Material based on Synopsys University Courseware and Synopsys CES

Developed by Vazgen Melikyan

# Agenda

The role and classification of HDLs

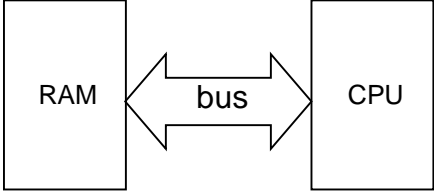
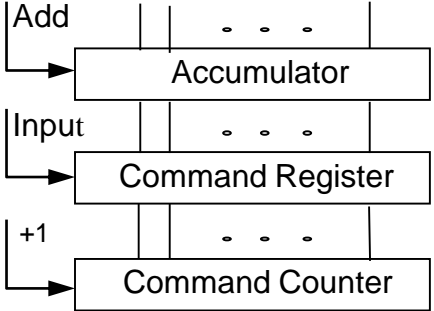
Verilog

Finite State Machine Review

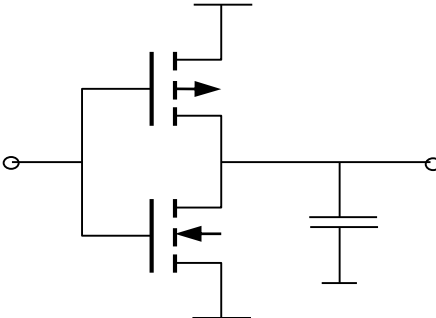
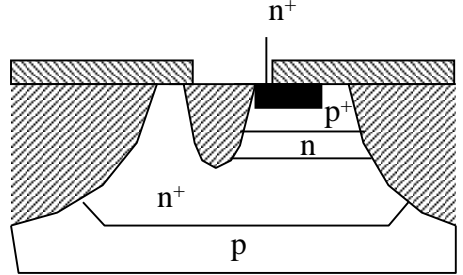
# What is HDL?

- Hard & Difficult Language?
  - No, means **H**ardware **D**escription **L**anguage
- High Level Language
  - To describe the circuits by syntax and sentences
  - As oppose to circuit described by schematics
  - Allows designers to model the **concurrency** of process found in hardware
- Widely used HDLs
  - Verilog – Similar to C
  - SystemVerilog – Similar to C++
  - VHDL – Similar to PASCAL

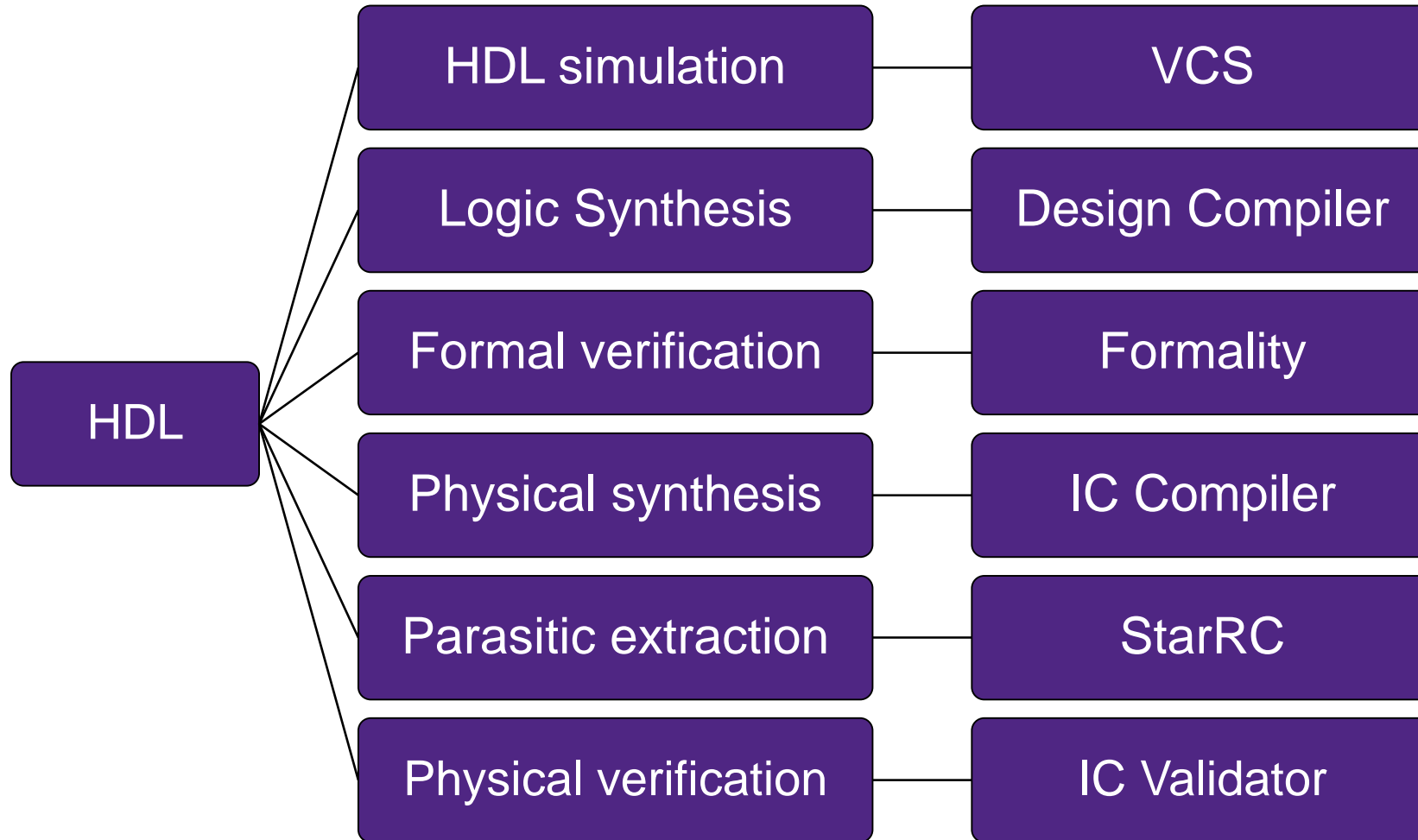
# Design Levels

Level	Modeling Object	Example of Modeling Object	Mathematical Apparatus	HDL
System	Structural Circuit	 <p>A diagram showing a rectangular box labeled 'RAM' on the left and a rectangular box labeled 'CPU' on the right. Between them is a double-headed arrow labeled 'bus'.</p>	Queuing theory	SystemVerilog, SystemC
Register-Transfer	Functional Circuits on the level of multibit devices	 <p>A diagram showing three stacked rectangular boxes. The top box is labeled 'Accumulator' and has an input arrow from the left labeled 'Add'. The middle box is labeled 'Command Register' and has an input arrow from the left labeled 'Input'. The bottom box is labeled 'Command Counter' and has an input arrow from the left labeled '+1'. Each box has three dots in the center, indicating multiple bits.</p>	Boolean Algebra	Verilog,VHDL
Gate	Circuit on the level of gates and flip-flops			

# Design Levels (2)

Level	Modeling Object	Example of Modeling Object	Mathematical Apparatus	HDL
Circuit	Electrical Circuit		System of differential equations	Spice
Device	IC Components		System of differential equations with partial derivative	

# HDL Usage in Digital Design Flow



# Verilog

1984:

- Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation.

1989:

- Gateway was acquired by Cadence

1990:

- Verilog was made an open standard under the control of Open Verilog International.

1995:

- The language became an IEEE standard (IEEE STD 1364) and was updated in 2001 and 2005.

# SystemVerilog

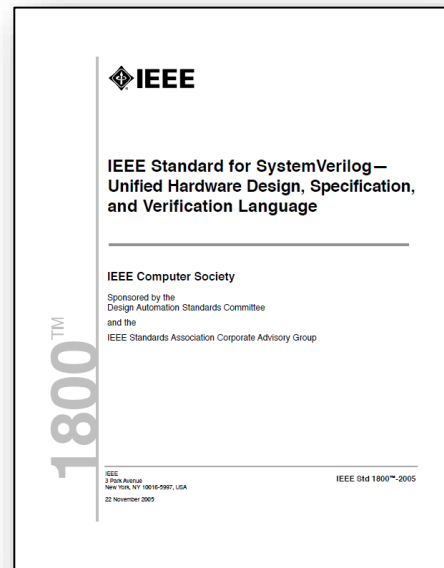
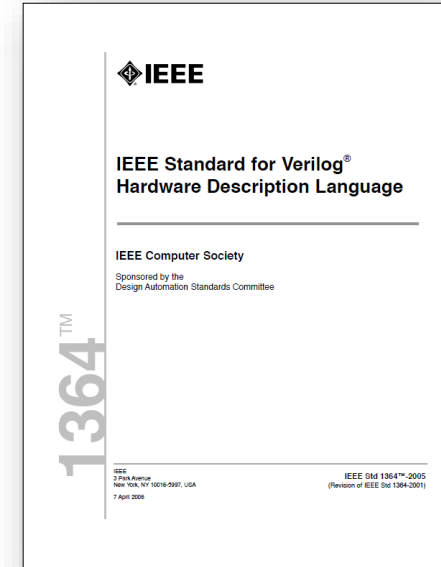
- SystemVerilog is the industry's first unified hardware description and verification language
- Started with Superlog language to Accellera in 2002
- Verification functionality (base on OpenVera language) came from Synopsys
- In 2005 SystemVerilog was adopted as IEEE Standard (1800-2005). The current version is 1800-2009





# IEEE-1364 / IEEE-1800

Verilog 2005 (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features



SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design-verification and design-modeling

# Simulation and Synthesis

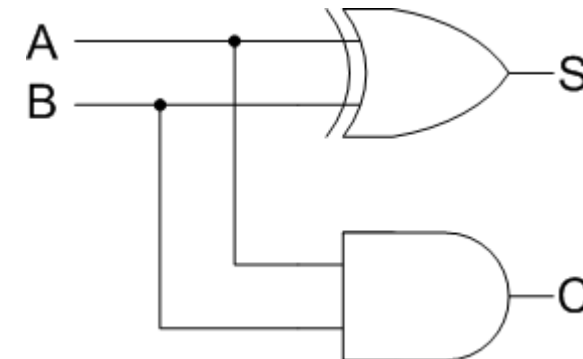
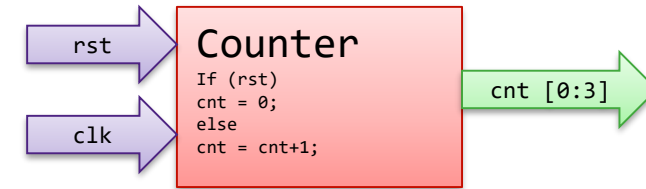
- The two major purposes of HDLs are logic simulation and synthesis
  - During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly
  - During synthesis, the textual description of a module is transformed into logic gates
- Circuit descriptions in HDL resemble code in a programming language. But the code is intended to represent hardware
- Not all of the Verilog commands can be synthesized into hardware

# Simulation and Synthesis

- Our primary interest is to build hardware, we will emphasize a synthesizable subset of the language
- Will divide HDL code into synthesizable modules and a test bench (simulation)
  - The synthesizable modules describe the hardware.
  - The test bench checks whether the output results are correct (only for simulation and cannot be synthesized)

# Types of modeling

- Behavioral
  - Models describe what a module does.
  - Use of assignment statements, loops, if, else kind of statements
- Structural
  - Describes the structure of the hardware components
  - Interconnections of primitive gates (AND, OR, NAND, NOR, etc.) and other modules





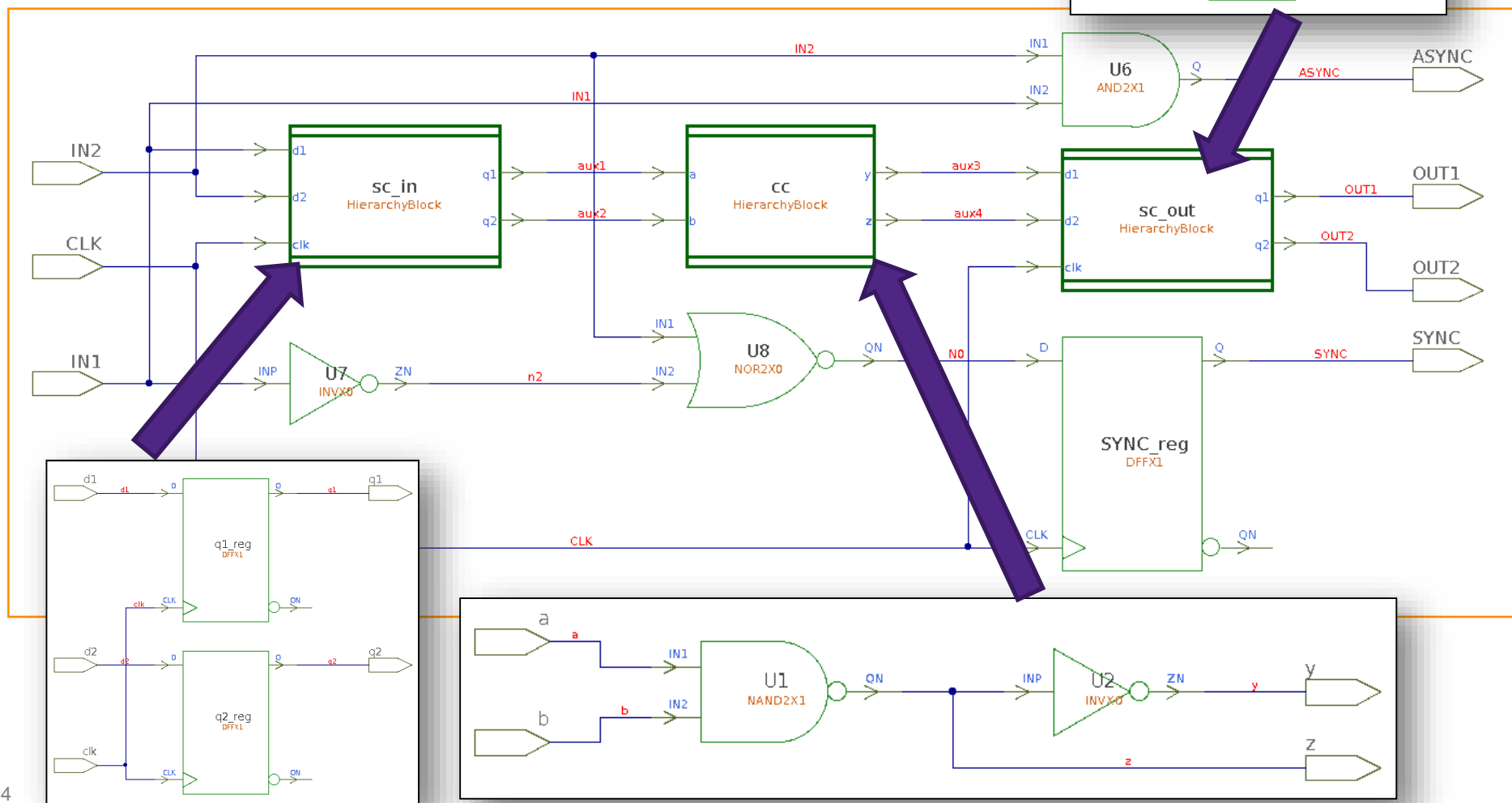
From this...

```
module comb_cell (input a,b, output y,z);  
    assign y = a && b;  
    assign z = ! y ;  
endmodule
```

```
module seq_cell (input d1,d2, clk, output reg q1,q2);  
    always @( posedge clk)  
    begin  
        q1 <= d1;  
        q2 <= d2;  
    end  
endmodule
```

```
module top (input IN1,IN2,CLK, output OUT1, OUT2, ASYNC, output reg SYNC);  
    wire aux1,aux2,aux3,aux4;  
  
    // continous assignment  
    assign ASYNC = IN1 && IN2 ;  
  
    //procedural block  
    always @ (posedge CLK)  
    begin  
        SYNC = IN1 && ( ! IN2 ) ;  
    end  
  
    //instantiation  
    seq_cell sc_in (.clk(CLK), .d1(IN1), .d2(IN2), .q1(aux1), .q2(aux2) );  
    comb_cell cc (.a(aux1), .b(aux2), .y(aux3), .z(aux4));  
    seq_cell sc_out (.clk(CLK), .d1(aux3), .d2(aux4), .q1(OUT1), .q2(OUT2) );  
endmodule
```

To this...



# Verilog

# Verilog

Modeling concept

Basic concept

Behavioral modeling

Data flow modeling

Gate level modeling

Tasks and functions



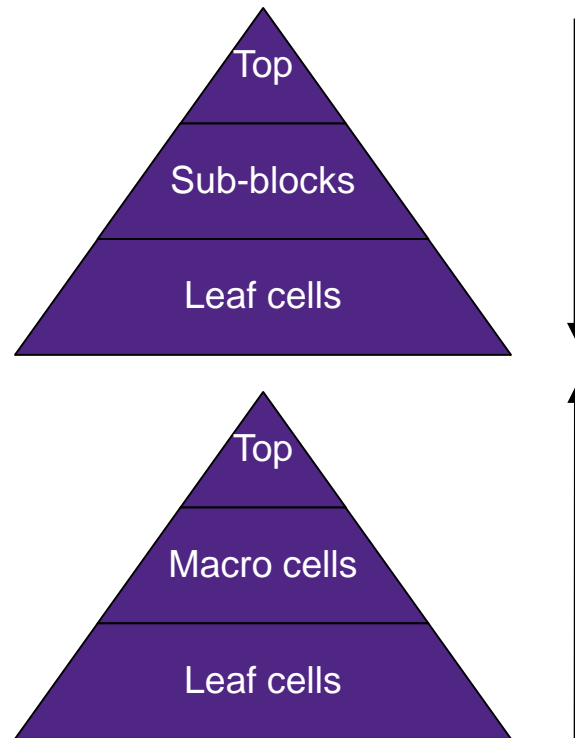
# Verilog

## *Modeling Concept*

# Levels of Abstraction

- Behavioral
  - The highest level of abstraction provided by Verilog HDL. A module is implemented in terms of the desired design algorithm
- Data Flow
  - At this level the module is designed by specifying the data flow
- Gate Level
  - The module is implemented in terms of logic gates and interconnections between these gates
- Switch Level
  - This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes and the interconnections between them

# Design Methodologies



- **Top-Down Design**

- Define the top-level block
- Identify the sub-blocks necessary to build the top-level block
- Sub-blocks are formed by leaf cells which cannot be further divided

- **Bottom-up Design**

- Build macro cells using leaf cells
- Move up the hierarchy until the top-level is reached

# Verilog

## *Basic Concept*

# Module

- Module
  - Basic building block
  - Only user-declared object type
  - Element or a collection of lower level design blocks

# Module Header Format

## Traditional header format

```
module
  count(Cout,Cin,clk);
input Cin, clk;
output Cout;
reg Cout;
.....
endmodule
```

## Newer header format

```
module count(output
Cout, input Cin, clk);
reg CoutR;
assign Cout=CoutR;
.....
endmodule
```

# Lexical Conventions

- Comments
- Numbers
- Strings
- Identifier
- Operators

# Lexical Conventions: Comments

Comments	
Syntax	Example
One line comment- <code>//comment</code> Multiple line comment- <code>/*comment*/</code>	<code>sum=a+b // This is a one line comment</code> <code>d[1]=1'b0; /* d[0]=1'b1;</code> <code>          This is a multiple-line</code> <code>          comment block */</code>



# Lexical Conventions: Identifiers

- Identifiers

- Names given to objects
- Cannot start with a number or dollar sign (\$)
- Can start with alphabetic character or underscore

wire a1;            // wire is a keyword, a1 is an identifier

output sum;        // output is a keyword, sum is an identifier

# Lexical Conventions: Numbers

- Sized numbers `<size>`<base format><number>`
  - `5`b10111` // 5-bit binary number
  - `16`hcdab` // 16-bit hexadecimal number
  - `3`d7` // 3-bit decimal number
- Unsized numbers
  - `16549` // 32-bit decimal number by default
  - ``o21` // 32-bit octal number

# Lexical Conventions: Numbers (2)

- x or z values
  - x – unknown value
  - z – high impedance value
  - 16`h536x // This is a 16-bit hexadecimal value with 4 least significant bits unknown
- Negative numbers
  - Number with a minus sign before the size of a constant number
  - -10`d9

# Lexical Conventions: Numbers (3)

- Underscore, Question mark
  - “\_” character – anywhere in a number except the first character
    - 16`b1010\_0110\_1101
  - “?” question mark – substitutes z in numbers for better readability
    - 8`b111?

# Lexical Conventions: Strings

- Strings
  - Any characters enclosed by double quotes
  - “Verilog HDL Concepts”

# Data Types

- Nets
- Vectors
- Registers
- Arrays

# Data Types: Nets

Name	Keyword	Default value	Default size
net	wire, supply0, supply1	z	1bit

## Example

```
wire net1;  
wire net2, net4;  
supply0 vss;
```

# Data Types: Registers

Name	Keyword	Default value	Default size
register	reg	x	1bit

## Example

```
reg register1;  
reg register2,  
register3;
```



# Data Types: Vectors

- Wire or register which is more than 1 bit in size
  - Definition syntax
    - <data type> [<start>:<end>] <identifier>
  - Reference syntax
    - <variable\_name>{array\_reference}

## Example

```
reg [3:0] output; // output is a 4-bit register
wire [31:0] data; // data is a 32-bit wire
reg [7:0] a;
data[3:0] = output; // partial assignment
output = 4'b0101; // assignment to the whole register
data[24]; // referencing to 24th bit
```

# Data Types: Arrays

- Array

- Definition syntax

- `<data_type_spec> {size} <variable_name> {array_size}`

- Reference syntax

- `<variable_name> {array_reference} {bit_reference}`

## Example

```
reg data [7:0]; // 8 1-bit data elements
integer [3:0] out [31:0]; // 32 4-bit output elements
out[3][27]; // referencing bit number 27 in the 3rd
element
```

# Parameter Types

- Parameters

- Parameters are unsigned integer constants by default
- Must be assigned when declared. Width defaults are enough for the value assigned
- May be declared signed (but many tools reject it)
- May be declared real (but not synthesizable)
- May be typed by vector index range
- Declaration allowed anywhere in module, but local parameter proffered in body

## Example

```
parameter lsb = 7;  
parameter size = 8, word = 32;  
parameter frequency = 100;  
parameter clk_cycle = frequency / 2 ;
```

# Lexical Conventions: Operators

- Operators

- Unary operators

- stand before the operand

- Binary operators

- stand between two operands


- Ternary operators

- have two separate operators that separate three operands

# Operator Types

Operator Type	Operator
Arithmetic operators	*, /, +, -, %
Logical operators	&&,   , !
Relational operators	>, <, >=, <=
Equality	=, !=, ==, !=
Bitwise operators	~, &,  , ^, (~^, ^~)
Reduction Operator	&, ~&,  , ~ , ^, ~^, ^~
Shift Operator	>>, <<
Concatenation	{, }
Replication	c = {4{a}}
Conditional	?:

# Operator Precedence

Operators	Operator Symbols	 <div>Highest Precedence</div> <div>Lowest Precedence</div>
Unary	+ - ! ~	
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& &, ^~ &, ~	
Logical	&& 	
Conditional	?:	

# Sequential Blocks

- Sequential Blocks

- Statements in a sequential block are processed in the order they are specified
- A statement is executed only after its preceding statement completes execution (except for non-blocking assignments)
- If delay or event control is specified it is relative to the simulation time, i.e. execution of the previous statement

# Parallel Blocks

- Statements in a parallel block are executed concurrently
- Ordering of statements is controlled by the delay or event control assigned to each statement
- If delay or event control is specified, it is relative to the time the block was entered



# Comparison of Sequential and Parallel Blocks

- Fundamental difference between sequential and parallel blocks:
  - All statements in a parallel block start at the time when the block was entered
  - Thus, the order in which the statements are written in the block is not important

# Parallel Blocks (2)

## Example

```
reg x, y;  
reg [1:0] z, w;  
initial  
fork  
  x = 1'b0;  
  #5 y = 1'b1;  
  #10 z = {x, y};  
  #20 w = {y, x};  
join
```

x completes at simulation time 0  
y completes at simulation time 5  
z completes at simulation time 10  
w completes at simulation time 20

# Basic Compiler Directives

- Syntax: ``<keyword>` Example:

- ``define MAX 16`b1111111111111111;`

- The given 16-bit number will be substituted wherever ``MAX` appears

- ``include integrator.v`

- Includes the contents of Verilog source file in another Verilog file during compilation

- ``timescale <reference time unit>/<time_precision>`

- `<reference time unit>` specifies the unit of measurement for times and delays

- `<time_precision>` specifies the precision to which the delays are rounded off during simulation

# Verilog

## *Behavioral Modeling*

# Behavioral Modeling Blocks

- Blocks
  - Always
  - Event-based timing control
  - Branch
  - Case, casex, casez

# Behavioral Modeling Blocks: Always Block

Syntax	Example
<code>always operation_name</code>	<pre>module pulse;   reg clock;   initial clock = 1'b0; // start the clock at 0   always #10 clock = ~clock; /* toggle every 10 time units*/ endmodule</pre>

# Behavioral Modeling Blocks: Event-based Timing Control

Syntax	Example
@(event) op_name	@ (clock) a = b; when the clock changes value, execute a = b @ (negedge clock) a = b; when the clock change to a 0, execute a=b a = @(posedge clock) b; evaluate b immediately and assign to a on a positive clock edge.

# Behavioral Modeling Blocks: Branch Statements

Syntax	Example
<pre>if (conditional_expression) statement{ else statement}</pre>	<pre>if (x!=1) a=2; else a=5</pre>



# Behavioral Modeling Blocks: Case

Syntax	Example
<pre>case(op)   op1:operation1   op2:operation2   op3:operation3   op4:operation4 default:operation5 endcase</pre>	<pre>case(S3)   00:A=00   01:A=01   10:A=10   11:A=11 default:operation5 endcase</pre>

# Behavioral Modeling Blocks: casex, casez

- Two variations of the case statement

- casez

- Treats all z values in the case alternatives or the case expression as don't cares, all bit positions with z can also be represented by ? in that position

- casex

- Treats all x and z values in the case item or the case expression as don't cares

```
reg [3:0] encoding;  
integer state;  
casex (encoding)  
4'b1xxx : next_state  
         = 3;  
4'bx1xx : next_state  
         = 2;  
4'bxx1x : next_state  
         = 1;  
4'bxxx1 : next_state  
         = 0;  
default : next_state  
         = 0;  
endcase
```

- An input encoding = 4'b10xz would cause next\_state = 3 to be executed

# Behavioral Modeling

- Definition of blocking and non-blocking procedural assignments
- Delay-based timing control mechanism in behavioral modeling
- The significance of structured procedures always and initial in behavioral modeling
- Event-based timing control mechanism in behavioral modeling
- Level-sensitive timing control mechanism in behavioral modeling
- Conditional statements using if and else
- Multiway branching, using case, casex, and casez statements
- Looping statements while, for, repeat, and forever
- Sequential and parallel blocks

# Procedural Assignments

- Update values of reg, integer, real, or time variables
  - Blocking
  - Non-blocking

# Procedural Assignments: Blocking Assignments

- Blocking assignment statements are executed in the order they are specified in a sequential block
- A blocking assignment will not block execution of statements that follow in a parallel block
- The = operator is used to specify blocking assignments

# Procedural Assignments: Blocking Assignments (2)

## Example

```
reg x, y, z;  
reg [15:0] reg_a, reg_b;  
integer count;  
initial begin
```

```
    x = 0; y = 1; z = 1;  
    count = 0;  
    reg_a = 16'b0;  
    reg_b = reg_a;
```

Executed at time zero

```
    #15 reg_a[2] = 1'b1;
```

Executed at time 15

```
    #10 reg_b[15:13] = {x, y, z};
```

Executed at time 25

```
    count = count + 1; //Assignment to an integer (increment)
```

```
end
```

# Procedural Assignments: Non-Blocking Assignments

- Non-blocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block
- A `<=` operator is used to specify non-blocking assignments

# Procedural Assignments: Non-Blocking Assignments (2)

## Example

```
reg x, y, z;  
reg [15:0] reg_a, reg_b;  
integer count;
```

Initial begin

`x = 0; y = 1; z = 1;`  
`count = 0;`  
`reg_a = 16'b0;`  
`reg_b = reg_a;`

Executed at time zero

`reg_a[2] <= #15 1'b1;`

Executed at time 15

`reg_b[15:13] <= #10 {x, y, z};`

Executed at time 10

`count <= count + 1;`

Executed at time zero

end



# Blocking vs. Non-Blocking

- Why use <= to specify flip-flops
- Blocking

```
begin
    A = B;
    C = D;
end
```

– Assignment of C blocked until A = B completed → They executed in sequence

- Non Blocking

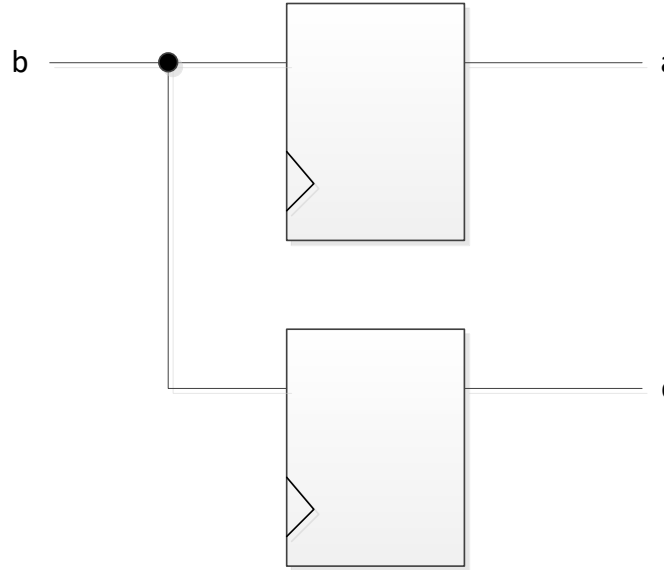
```
begin
    A <= B;
    C <= D;
end
```

– Assignment of C not blocked until A = B completed → They executed in parallel

# Blocking vs. Non-Blocking

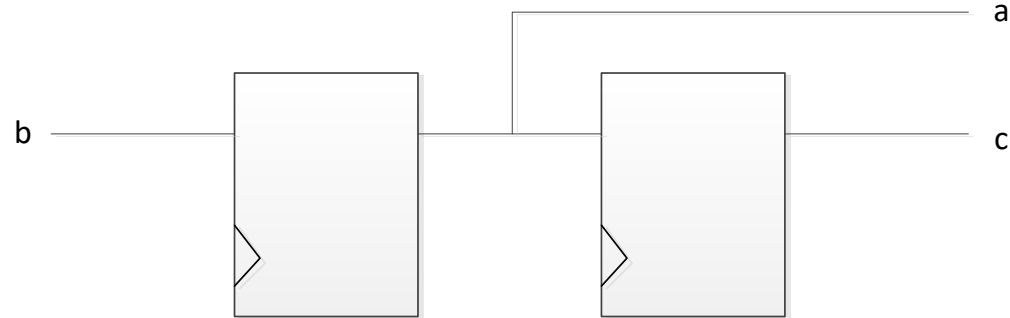
## Blocking

```
begin  
    a = b;  
    c = a;  
end
```



## Non-Blocking

```
begin  
    a <= b;  
    c <= a;  
end
```



# Blocking Statements

- Hand execute the following

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4;
end
//code
always @(posedge clock)
    c = a + b;
    d = c + a;
end
```

- Results?

# Non Blocking Statements

- Contrast it with this code

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4; c = 4'h2;
end
//code
always @(posedge clock)
    c <= a + b;
    d <= c + a;
end
```

- Results?

# Blocking vs. Non Blocking

- Which describes better what you expect to see?
  - Non blocking assignment
- Note
  - Use non blocking for flip flops
  - Use blocking for combinational logic
    - Logic can be evaluated in sequence
    - Not synchronized to clock
  - Don't mix them in the same procedural block

# Implementation of a 4 to 1 Multiplexor Using if Statement

```
module multiplexor4_1 (out, in1, in2, in3 ,in4, cntrl1, cntrl2);  
    output out;  
    input in1, in2, in3, in4, cntrl1, cntrl2;  
    reg out; // Note that this is now a register  
  
    always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)  
        if (cntrl1==1)  
            if (cntrl2==1)  
                out = in4;  
            else out = in3;  
        else  
            if (cntrl2==1)  
                out = in2;  
            else out = in1;  
endmodule
```

# Implementation of a 4 to 1 Multiplexor Using Case Statement

```
module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);  
    output out;  
    input in1, in2, in3, in4, cntrl1, cntrl2;  
    reg out; // note must be a register  
    always @(in1 | in2 | in3 | in4 | cntrl1 | cntrl2)  
        case ({cntrl2, cntrl1}) // concatenation  
            2'b00 : out = in1;  
            2'b01 : out = in2;  
            2'b10 : out = in3;  
            2'b11 : out = in4;  
            default : $display("Please check control bits");  
        endcase  
endmodule
```

# Verilog

## *Data Flow Modeling*



# Data Flow Modeling

- Describe the continuous assignment (assign) statement
- Assignment delay, implicit assignment delay, net declaration delay
- Expressions, operators, operands (already mentioned)
- Examples of digital circuits modeling in Verilog

# Assign Statement

- `assign out = in1 & in2;`
  - Continuous assign
  - Out is a net
  - i1 and i2 are nets
- `assign addit[15:0] = addit1[15:0] ^ addit2[15:0];`
  - Continuous assign for vector nets, addr is a 16 bit vector net
  - Addi1 and addit2 are 16-bit vector registers
- `assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;`
  - Concatenation
  - Left-hand side is a concatenation of a scalar net and a vector net

# Delays

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side.
  - Regular assignment delay
  - Implicit continuous assignment delay
  - Net declaration delay

# Implicit Net Declaration

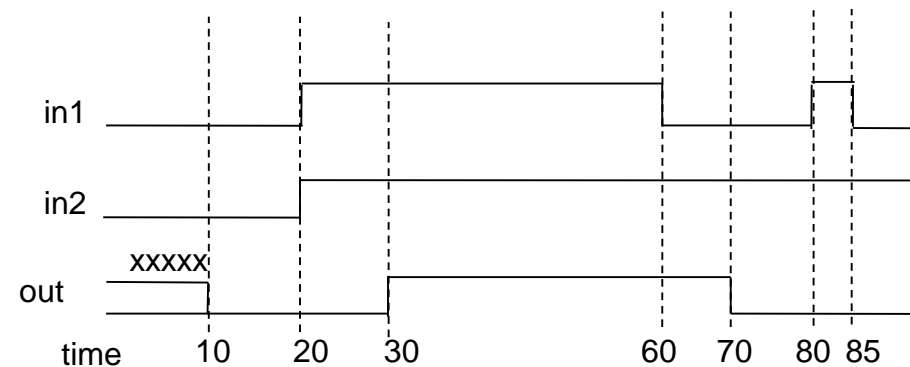
- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name.
- If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
// Continuous assign.  
out is a net.  
wire i1, i2;  
assign out = i1 & i2;
```

Out was not declared as a wire  
but an implicit wire declaration for out  
is done by the simulator

# Regular Assignment Delay

1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
2. When in1 goes low at 60, out changes to low at 70.
3. However, in1 changes to high at 80 but it goes down to low before 10 time units have elapsed.
4. Hence at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0.  
A pulse of width less than the specified assignment delay is not propagated to the output.



# Implicit Continuous Assignment Delay

- An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net

```
//implicit continuous assignment delay  
wire #10 out = in1 & in2;  
  
//same as  
wire out;  
assign #10 out = in1 & in2
```

# Net Declaration Delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.
- If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net Delays  
wire # 10 out;  
assign out = in1 & in2;
```

```
//The above statement has the same effect as the following.  
wire out;  
assign #10 out = in1 & in2;
```

# Logic Statement Implementation

```
module multiplexor4_1 (out, in1, in2, in3 ,in4, cntrl1, cntrl2);  
    output out;  
    input in1, in2, in3, in4, cntrl1, cntrl2;  
    assign out = (in1 & ~cntrl1 & ~cntrl2) |  
                 (in2 & ~cntrl1 & cntrl2) |  
                 (in3 & cntrl1 & ~cntrl2) |  
                 (in4 & cntrl1 & cntrl2);  
endmodule
```



# The Conditional Operator

Syntax	Example
<code>conditional_expression ? true_expression : false_expression</code>	<pre>module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);   output out;   input in1, in2, in3, in4, cntrl1, cntrl2;   assign out = cntrl1 ? (cntrl2 ? in4 : in3) :(cntrl2 ? in2 : in1); endmodule</pre>

# Verilog

## *Gate Level Modeling*

# Gate Level Modeling

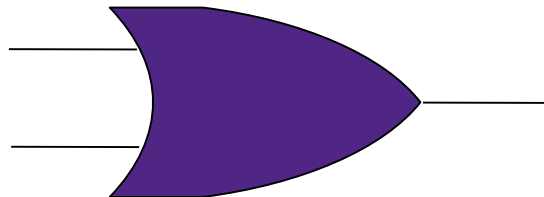
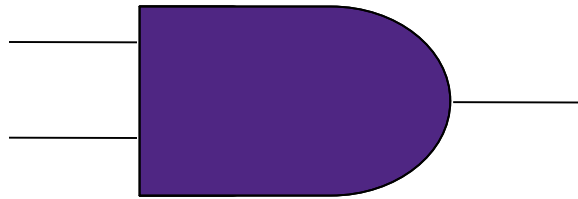
- Identify logic gate primitives provided in Verilog
- Truth tables for and/or, buf/not and bufif/notif type gates
- Examples of gate-level designs
- Rise, fall and turn-off delays in the gate-level design

# Gate Types

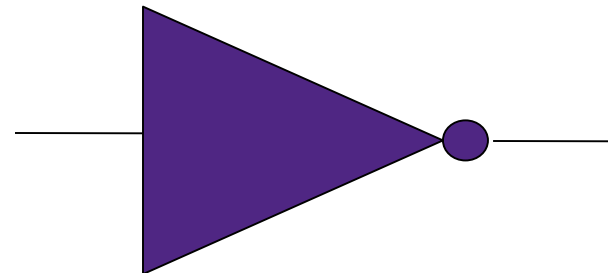
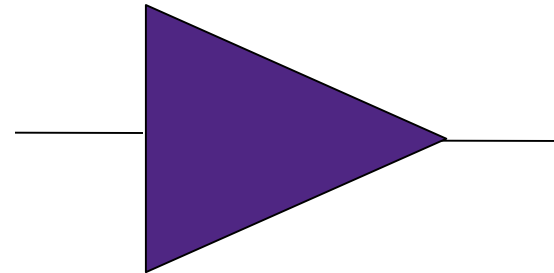
- And/Or
- Buf/not
- Bufif/notif
- UDP

# Gate Types: And/Or, Buf/Not Gates

and/or gates



buf/not gates



# Gate Types: Examples

- Examples of gate instantiations

```
wire OUT, IN1, IN2, IN3;
```

```
// Basic gate instantiations  
and gate1 (OUT, IN1, IN2);  
nand gate2 (OUT, IN1, IN2);  
or gate3 (OUT, IN1, IN2);  
nor gate4 (OUT, IN1, IN2);  
xor gate5 (OUT, IN1, IN2);  
xnor gate6 (OUT, IN1, IN2);
```

```
// More than 2 inputs: 3 input and gate  
and gate7 (OUT, IN1, IN2, IN3);  
// Gate instantiation without instance name  
nand (OUT, IN1, IN2); // This is also a legal instantiation
```

# Truth Tables for AND/OR Gates

AND	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

NOR	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	X	x
z	x	1	X	X

XOR	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

# Truth Tables for AND/OR Gates (2)

NAND	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

NOR	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

XNOR	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

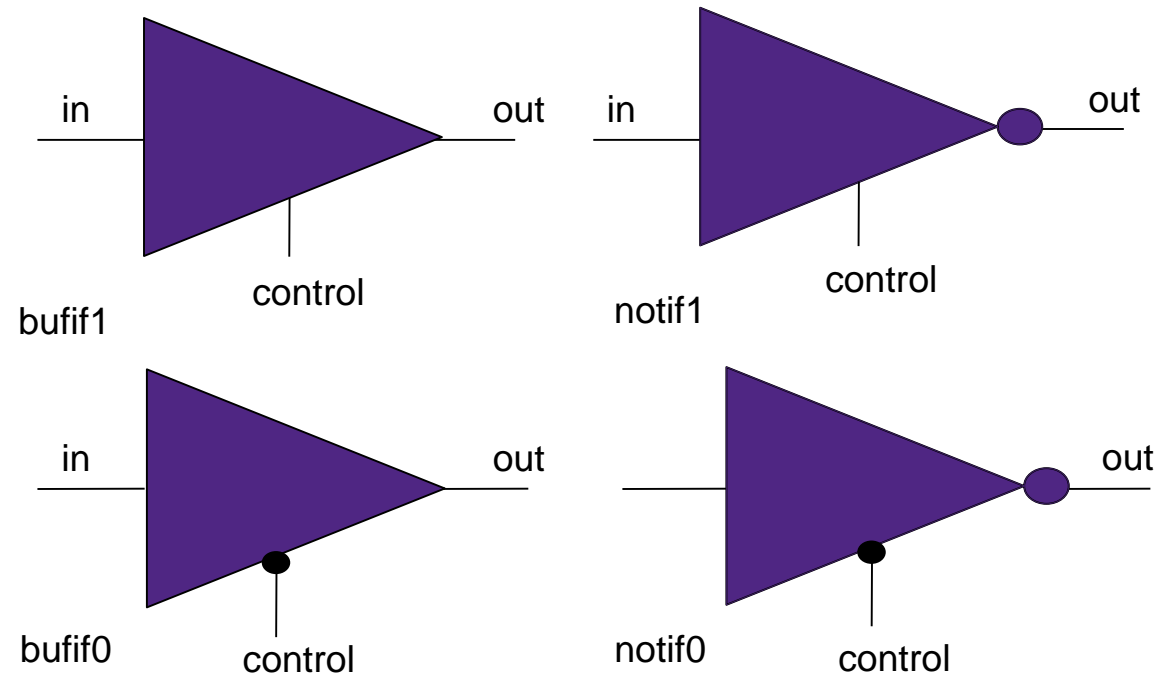


# Truth Tables for Buf/Not Gates

BUF	in	out
	0	0
	1	1
	x	x
	z	x

NOT	in	out
	0	1
	1	0
	x	x
	z	x

# Gate Types: Bufif/Notif Gates



# Gate Types: Examples (2)

- Examples of gate instantiations

```
wire OUT, IN, CONTROL;  
  
    // Basic instantiations of bufif gates  
    bufif1 b_1 (OUT, IN, CONTROL);  
    bufif0 b_0 (OUT, IN, CONTROL);  
  
    // Basic instantiations of notif gates  
    notif1 n_1 (OUT, IN, CONTROL);  
    notif0 n_0 (OUT, IN, CONTROL);
```

# Truth Tables for Bufif/Notif Gates

		control			
in	BUFIF1	0	1	x	z
	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

		control			
in	BUFIF0	0	1	x	z
	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

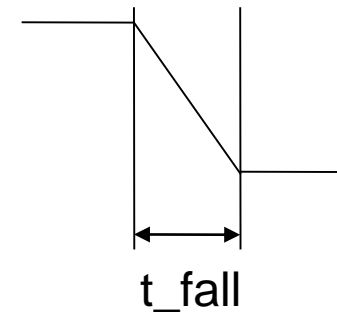
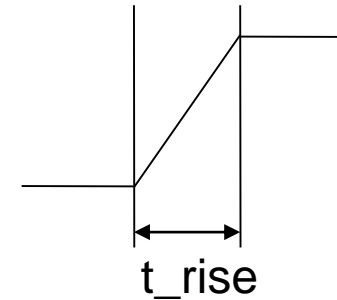
# Truth Tables for Bufif/Notif Gates (2)

		control			
in	NOTIF1	0	1	x	z
	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

		control			
in	NOTIF0	0	1	x	z
	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

# Gate Delays

- Rise delay is associated with a gate output transition to 1 from another value.
- Fall delay is associated with a gate output transition to 0 from another value.
- Turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.



# Gate Delays (2)

- Delay specifications
  - One delay is specified – its value is used for all transitions
  - Two delays are specified – they refer to rise and fall delay values respectively
  - Three delays are specified – they refer to rise, fall and turn-off delay values respectively
  - Default value is zero

# Gate Delays (3): Example

```
// Delay is equal to trans_delay for all transitions
nand #(trans_delay) g1 (out, in1, in2);

// Rise and Fall delays are specified
and #(rise_delay, fall_delay) g2 (out, in1, in2);

// Rise, Fall and Turn-off delays are specified
bufif0 #(5,7,9) b1 (out, in , control);
```



# Specify Block Example

```
module Fullpath(output[2:0] Qbus, output z, input A, B, C, CLOCK);  
  
.....(functionality omitted).....  
  
    specify  
    specparam tALL=10, tR=20, tF=21;  
        (A,B,C *>Qbus) = tALL;  
        (CLOCK *> Qbus) = (tR, tF);  
    endspecify  
  
endmodule
```

# UDP

Logic/state Representation/transition	Abbreviation
Don't care (0, 1, or X)	?
Transitions from logic x to logic y (xy). (01), (10), (0x), (1x), (x1), (x0), (?1) ..	(xy)
Transition from (01)	R or r
Transition from (10)	R or r
(01), (0X), (X1): positive transition	P or p
(10), (1x), (x0): negative transition	N or n
Any transition	* or (??)
Binary don't care (0, 1)	B or b

# UDP (2)

- Latch with asynchronous reset

```
primitive latch (q, clock, reset, data);
  input clock, reset, data;
  output q;
  reg q;

  initial q=1'b1; //initialization
table
  // clock reset data q    q+
  ?   1      ?   :   ?   : 1 ;
  0   0      0   :   ?   : 0 ;
  1   0      ?   :   ?   : - ;
  0   0      1   :   ?   : 1 ;
endtable
endprimitive
```

# Ports

- Descriptions

- ports provide the interface by which a module can communicate with its environment

- Types

- input, output or inout (implicitly all are declared as wire in Verilog)

- Port Data Types

- input or inout ports cannot be of type reg, because reg variables store values
- input ports should only reflect the changes of the external signals they are connected to

```
module DFF (dout,din,clk,resetn);  
    output  dout;  
    input    din,clk,resetn;  
    reg dout; // As the output of D Flip-Flop holds value it is declared as reg  
  
    always @(posedge clk or negedge resetn)  
        if (~resetn) dout<=0;  
        else dout<=din;  
endmodule
```

# Port Connection Rules

- Ports

- Inputs

- Internally, input ports must always be of the type net
    - Externally, the inputs can be connected to a variable which is reg or net

- Outputs

- Internally, output ports can be of the type reg or net
    - Externally, the outputs must always be connected to a net

- Inouts

- Internally, inout ports must always be of the type net
    - Externally, inout ports must always be connected to a net

# Port Connection Rules (3)

- Connections
  - By order
  - By name

# Port Connection Rules: By Order

## Example

```
module TOP;
  reg data_in,clock,resetn;
  wire data_out;

  // In the following block DFF module is instantiated and is called DFF_TEST
  // Signals are connected to ports in order
  DFF DFF_TEST (data_out,data_in,clock,resetn);
  ...
  <Stimulus>

endmodule
```

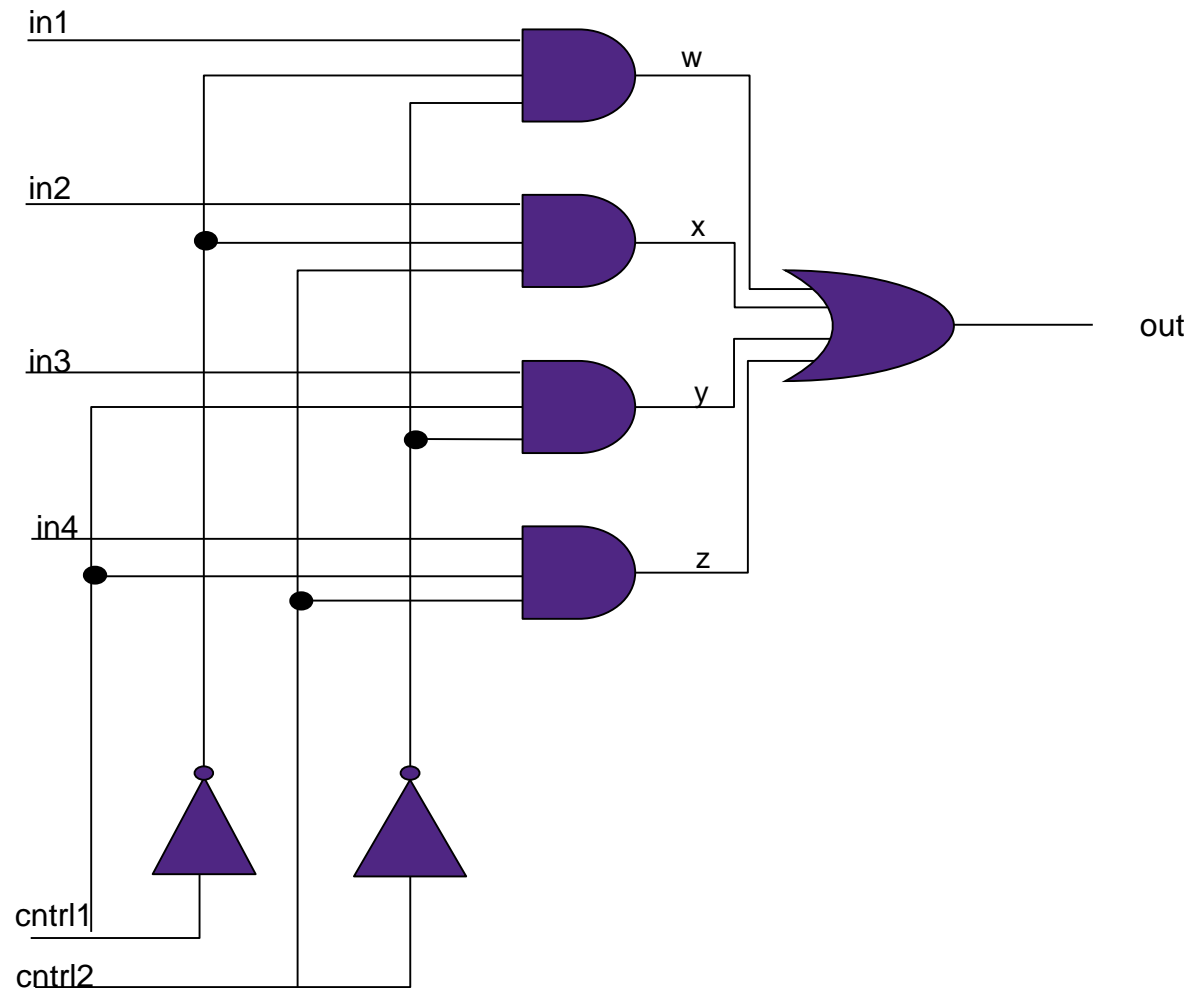
# Port Connection Rules: By Name

- Connecting ports by name
  - In large designs where the number of ports is great it may seem hard to use the ordered method of port connection
  - Verilog provides the capability to connect external signals to ports by the port names
  - As long as the port name is not changed the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations

```
DFF DFF_TEST (.din(data_in), .dout(data_out), .resetn(resetn), .clk(clock));
```



# Gate Level Implementation of a 4 to 1 Multiplexer



# Gate Level Implementation of a 4 to 1 Multiplexer: Verilog Description

```
module multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);
    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;
    wire notcntrlr1, notcntrl2, w, x, y, z;

    not (notcntrl1, cntrl1);
    not (notcntrl2, cntrl2);
    and (w, in1, notcntrl1, notcntrl2);
    and (x, in2, notcntrl1, cntrl2);
    and (y, in3, cntrl1, notcntrl2);
    and (z, in4, cntrl1, cntrl2);
    or (out, w, x, y, z);

endmodule
```

# Verilog

## *Tasks and Functions*

# Tasks and Functions

- **Tasks**

- Can be addressed by means of hierarchical names
- Included in design hierarchy
- Arguments
  - Output
  - Inout
  - Input

- **Functions**

- Can be addressed by means of hierarchical names
- Included in design hierarchy
- Arguments
  - Input

# Differences Between Tasks and Functions

- Functions

- A function can enable another function but not another task
- Functions always execute in 0 simulation time
- Functions must not contain any delay, event or timing control statements
- Functions must have at least one input argument. They can have more than one input
- Functions always return a single value. They cannot have output or inout arguments

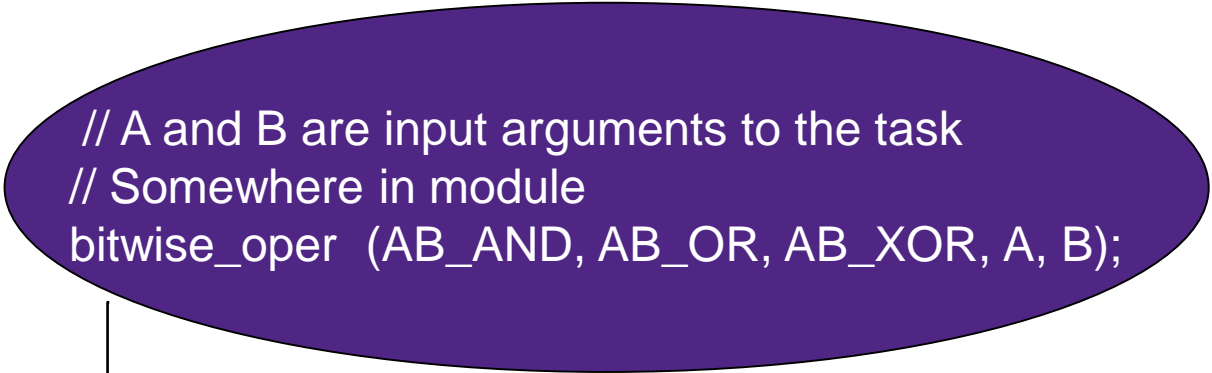
- Tasks

- A task can enable other tasks and functions
- Tasks may execute in non-zero simulation time
- Tasks may contain delay, event, or timing control statements
- Tasks may have zero or more arguments of type input, output or inout
- Tasks do not return with a value but can pass multiple values through output and inout arguments

# Tasks

- Keywords – task and endtask
- Use tasks if any of these conditions is true
  - There are delay, timing or event control constructs in the procedure
  - The procedure has zero or more than one output arguments
  - The procedure has no input arguments

```
task bitwise_oper;  
    output  [15:0] ab_and, ab_or, ab_xor;  
    input   [15:0] a,b;  
    begin  
        #10 ab_and = a & b;  
        ab_or = a | b;  
        ab_xor = a ^ b;  
    end  
  
endtask
```



```
// A and B are input arguments to the task  
// Somewhere in module  
bitwise_oper (AB_AND, AB_OR, AB_XOR, A, B);
```

task invocation

# Functions

- Keywords – function and endfunction
- Use functions if all of these conditions are true
  - There are no delay, timing or event control constructs in the procedure
  - The procedure returns a single value
  - There is at least one input argument

```
function calc_parity;  
    input  [31:0] address;  
    begin  
        calc_parity = ^ address  
    end  
endfunction
```

// addr is an input to the function

// Somewhere in module

parity = calc\_parity(addr);

function invocation

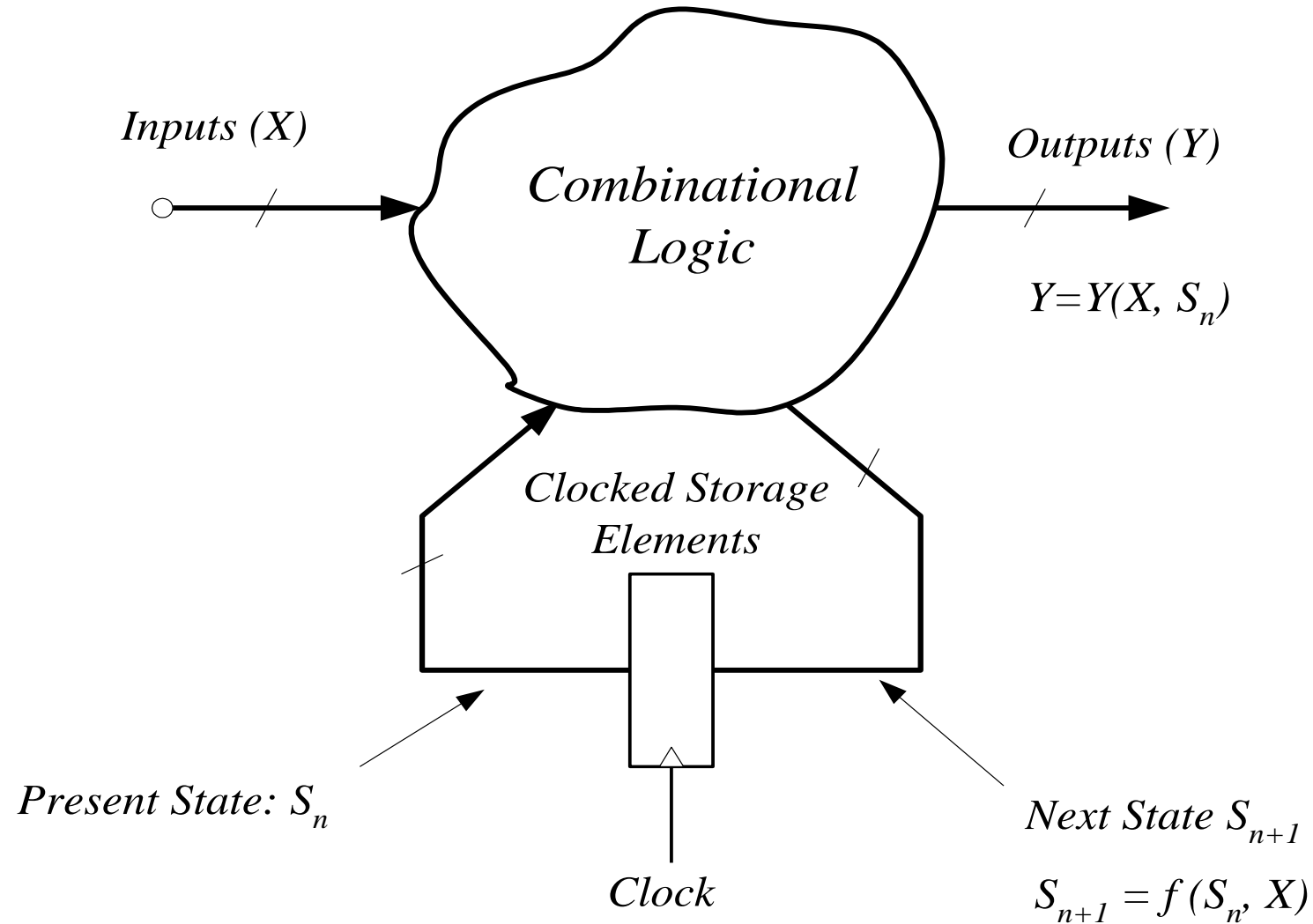
# Summary

- Tasks and functions are used to define common Verilog functionality that is used in many places in the design.
- Tasks can take any number of input, inout or output arguments. Delay, event or timing control constructs are permitted in tasks. Tasks can enable other tasks and functions.
- Functions are used when exactly one return value is required and at least one input argument is specified. Delay, event or timing control constructs are not permitted in functions. Functions can invoke other functions but cannot invoke other tasks.
- Tasks and functions are included in a design hierarchy and can be addressed by hierarchical name referencing.
- A register with name as the function name is declared implicitly when a function is declared. The return value of the function is passed back in this register.



# Finite State Machine Review

# Finite-State machine *Abstraction*



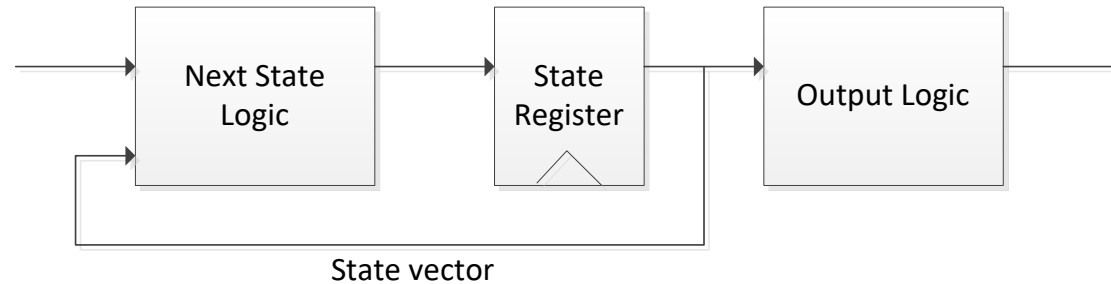
# Finite-State machine

## *Abstraction*

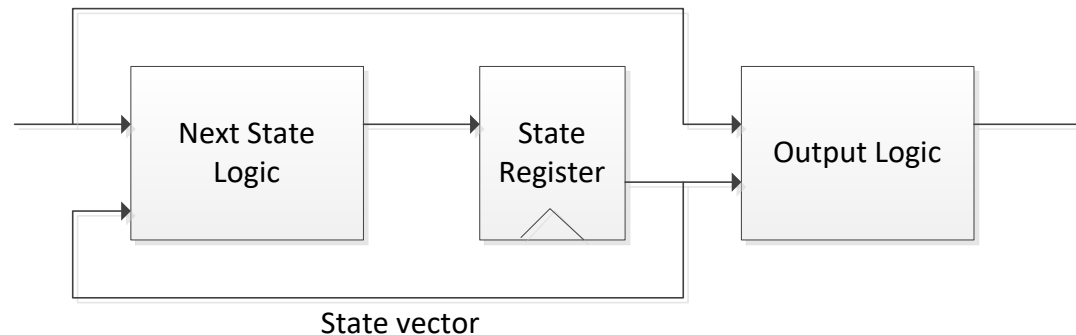
- Clocked Storage Elements: Flip-Flops and Latches should be viewed as **synchronization elements**, not merely as **storage elements** !
- Their main purpose is to **synchronize** fast and slow paths:
  - Prevent the fast path from corrupting the state
- Function of clock signals is to provide a reference point in time when the FSM changes states

# FSM - Types

- Moore machine
  - Outputs depend solely on state vector (simplest to design)

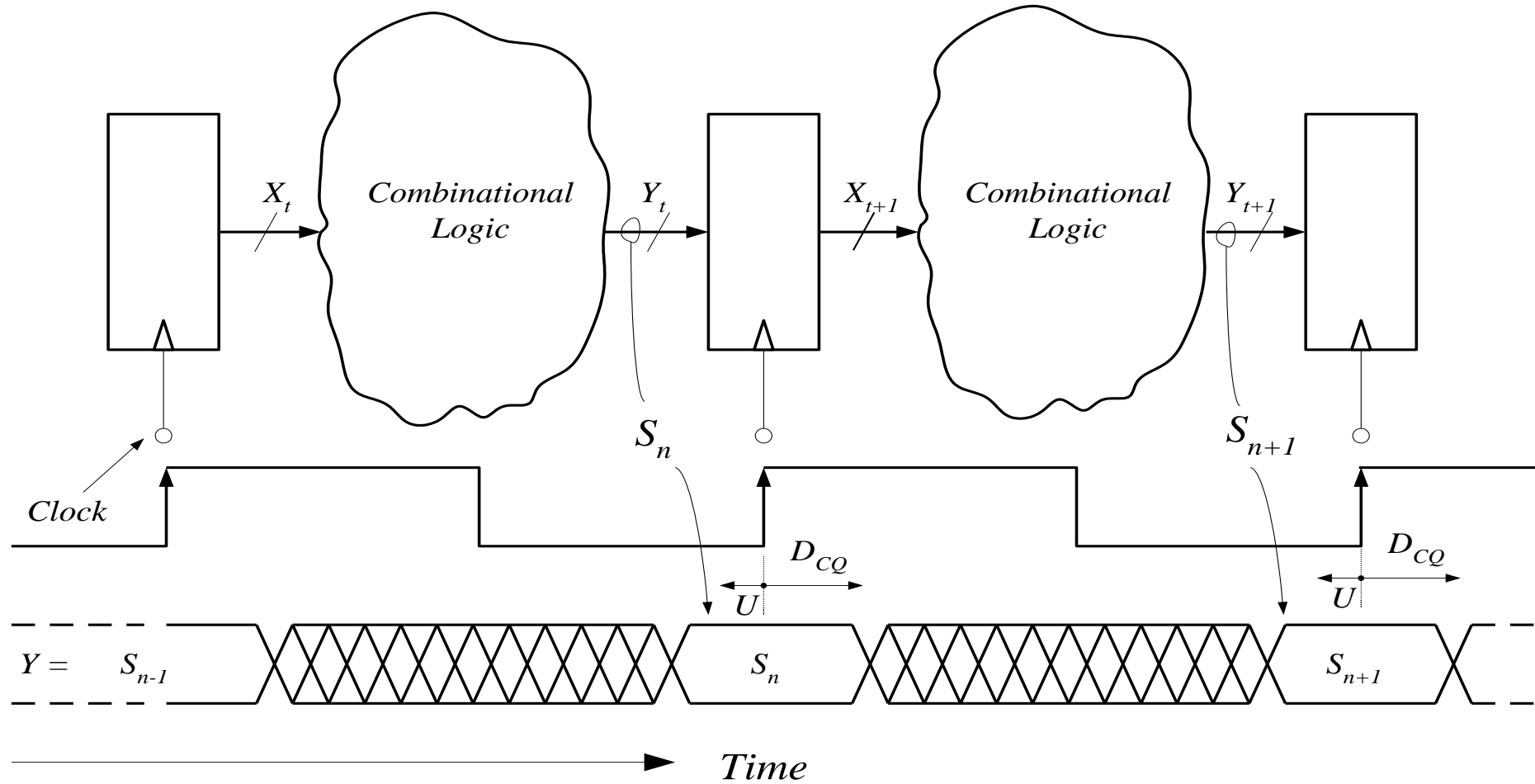


- Mealy machine
  - Outputs depend on inputs and state vector (only use it if smaller or faster machines are needed)



# Finite-State machine

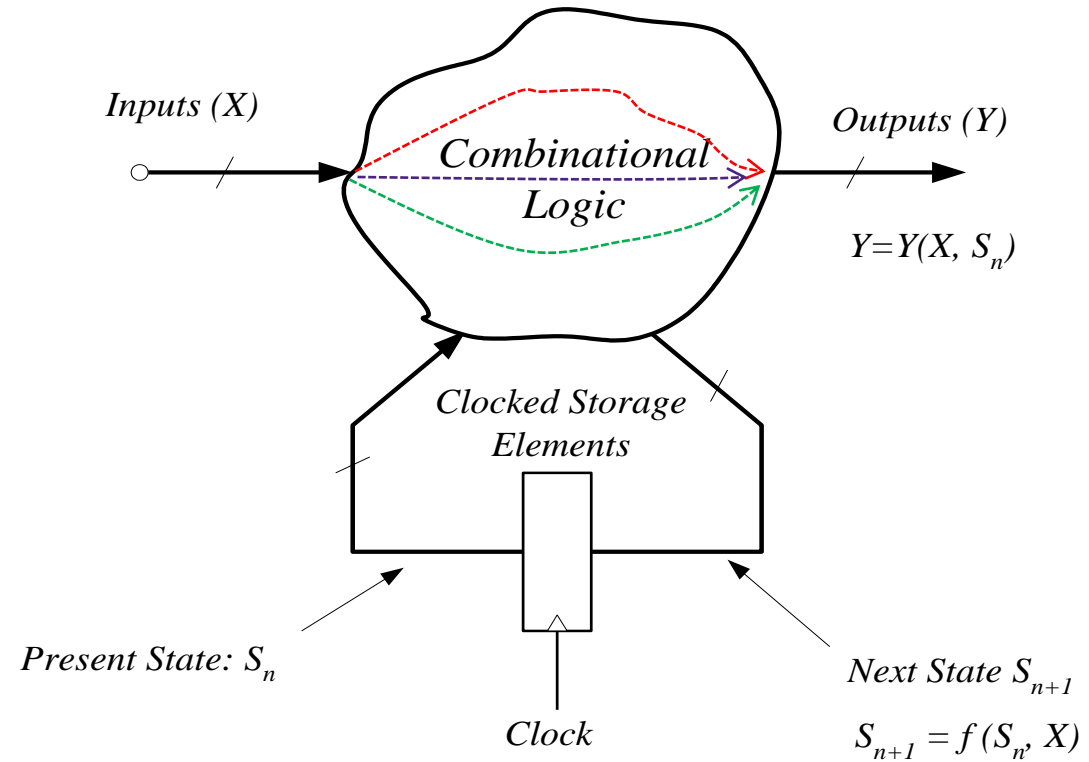
## *State Changes*



# Finite-State machine

## *Critical Path*

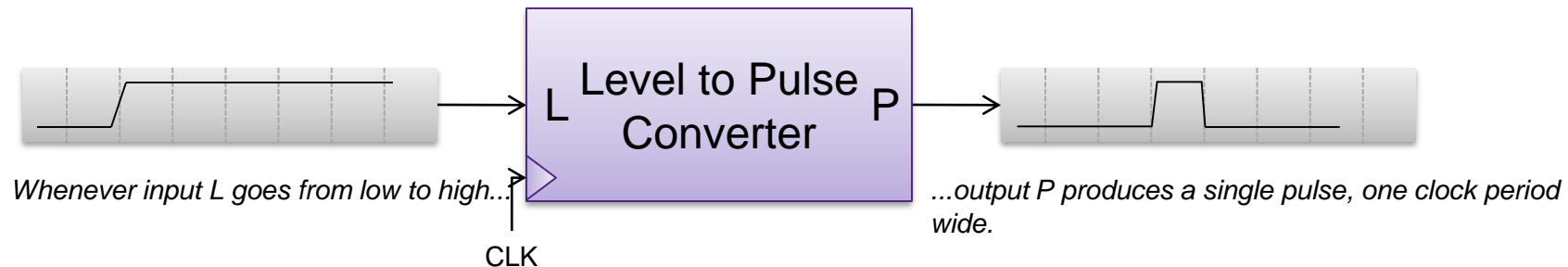
Critical path is defined as the chain of gates in the longest (slowed) path through the logic



# FSM Implementation

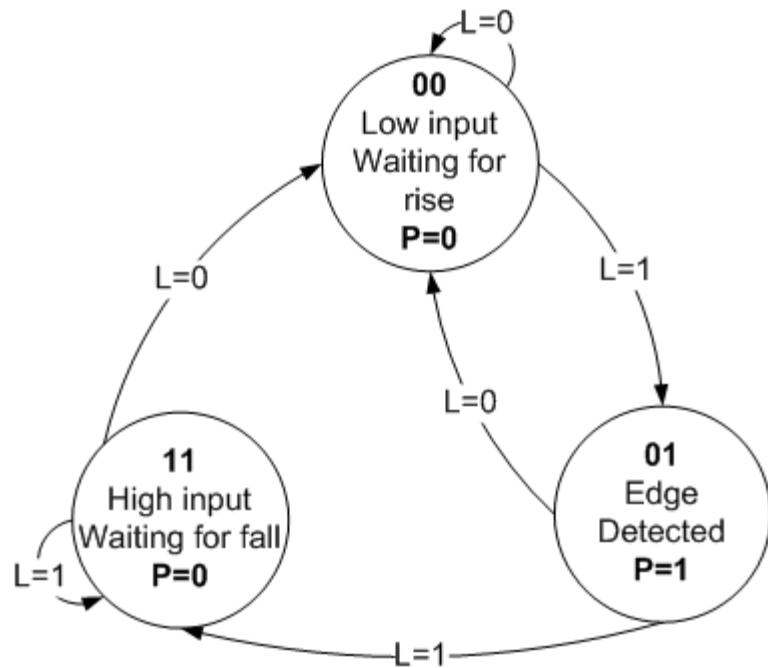
## *Design example*

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- In other words, it's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters



# FSM Implementation

## *State Diagram (Moore implementation)*

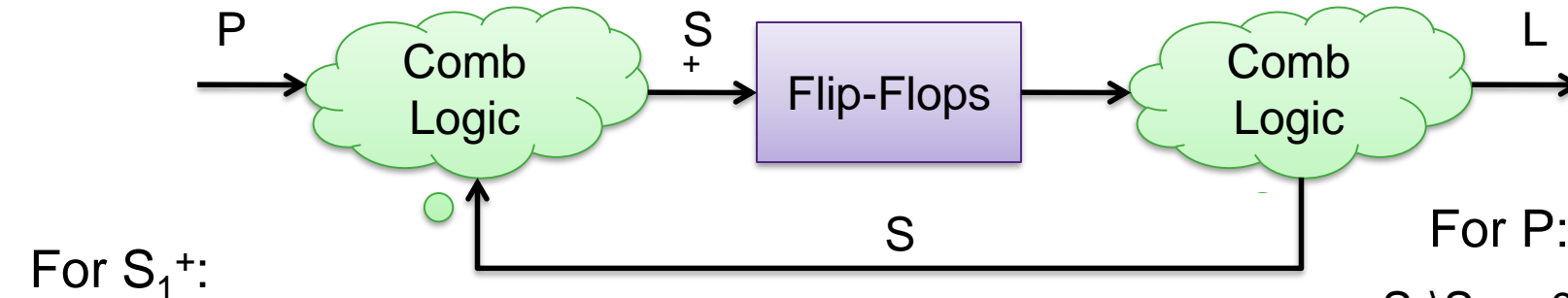


Current State		In	Next State		Out
$S_1$	$S_0$	$L$	$S_1^+$	$S_0^+$	$P$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0



# FSM Implementation

## *Logic implementation*



For  $S_1^+$ :

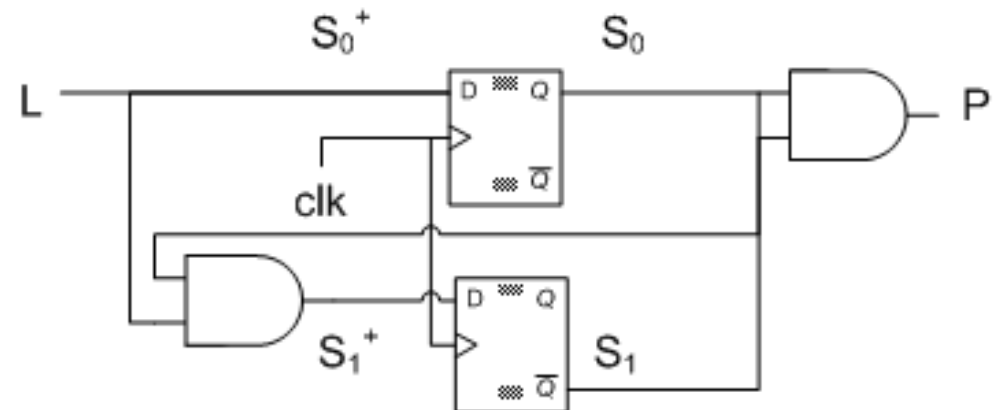
$L \backslash S_1$	00	01	11	10
$S_0$				
0	0	0	0	X
1	0	1	1	X

For  $S_0^+$ :

$L \backslash S_1$	00	01	11	10
$S_0$				
0	0	0	0	X
1	1	1	1	X

For P:

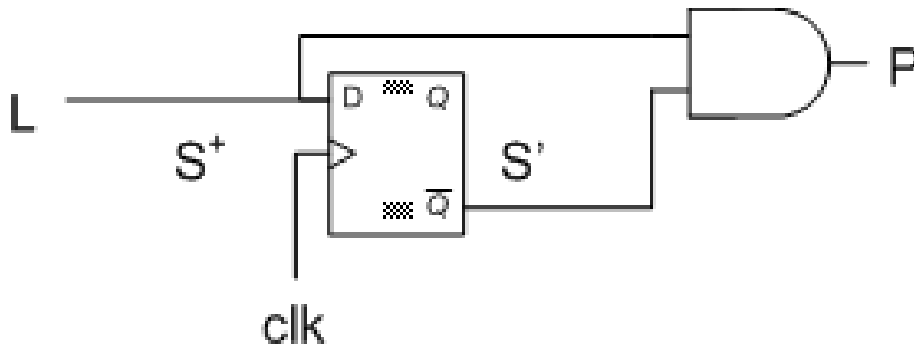
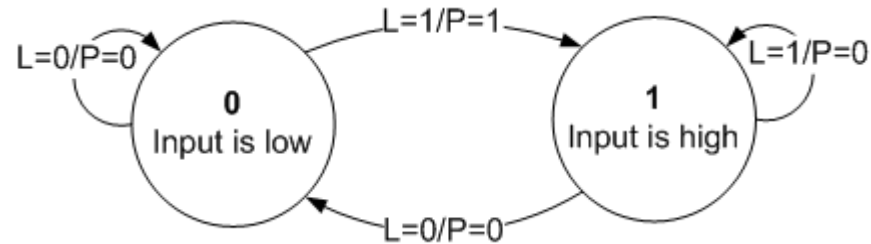
$S_0 \backslash S_1$	0	1
0	0	X
1	1	0



# FSM Implementation

## *Mealy implementation*

- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations

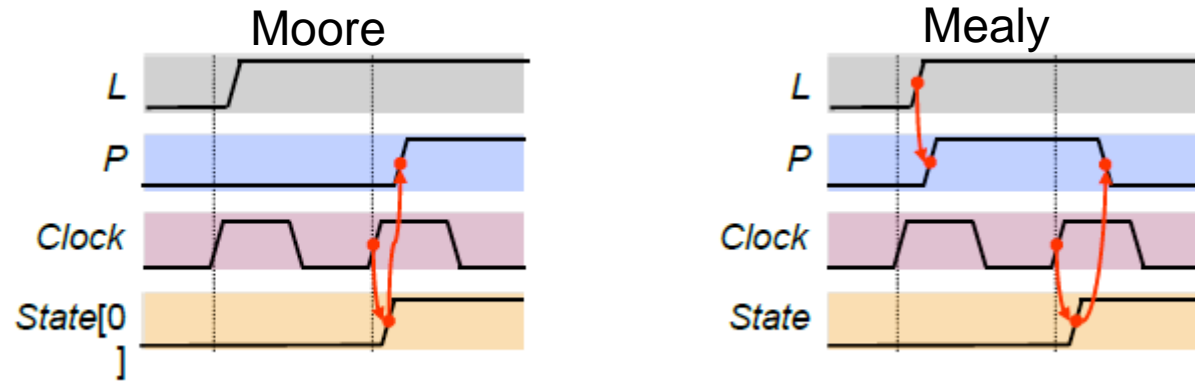


Pre State	In	Next State	Out
S	L	S <sup>+</sup>	P
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

# FSM Implementation

## *Moore/Mealy trade-off*

- Remember that the difference is in the output:
  - Moore outputs are based on state only
  - Mealy outputs are based on state and input
  - Therefore, Mealy outputs generally occur one cycle earlier than a Moore:



- Compared to a Moore FSM, a Mealy FSM might...
  - Be more difficult to conceptualize and design
  - Have fewer states

# HDL FSM Implementation

## *Binary Encoding*

- Straight encoding of states

$S0 = "00"$   $S1 = "01"$   $S2 = "10"$   $S3 = "11"$

- For  $n$  states, there are  $\lceil \log_2(n) \rceil$  flip-flops needed
- This gives the least numbers of flip-flops
- Good for “Area” constrained designs
- Number of possible illegal states =  $2^{\lceil \log_2(n) \rceil} - n$
- Drawbacks:
  - Multiple bits switch at the same time = Increased noise and power
  - Next state logic is multi-level = Increased power and reduced speed

# HDL FSM Implementation

## *Gray-Code Encoding*

- Encoding using a gray code where only one bit switches at a time

$S0 = "00"$   $S1 = "01"$   $S2 = "11"$   $S3 = "10"$

- For  $n$  states, there are  $\lceil \log_2(n) \rceil$  flip-flops needed
- This gives low power and noise due to only one bit switching
- Good for “power/noise” constrained designs
- Number of possible illegal states =  $2^{\lceil \log_2(n) \rceil} - n$
- Drawbacks:
  - The next state logic is multi-level = Increased power and reduced speed

# HDL FSM Implementation

## *One-Hot Encoding*

- Encoding one flip-flop for each state

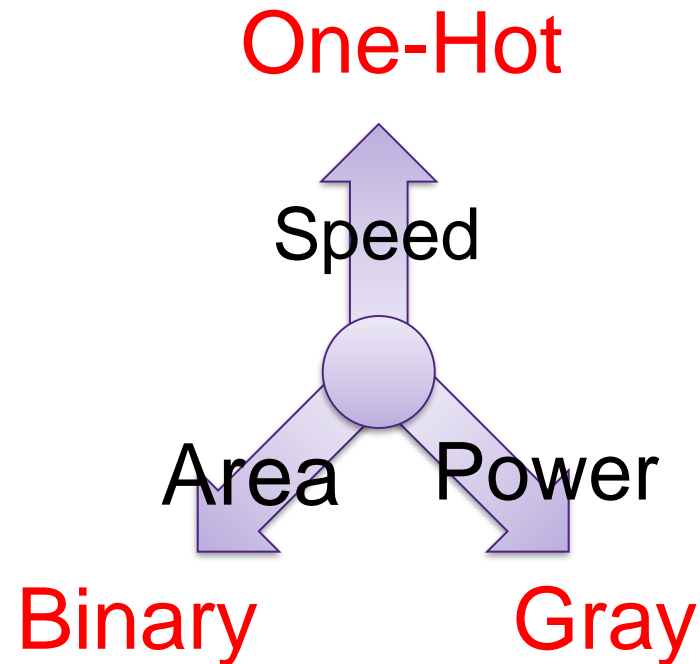
$S0 = \text{"0001"} \quad S1 = \text{"0010"} \quad S2 = \text{"0100"} \quad S3 = \text{"1000"}$

- For n states, there are n flip-flops needed
- The combination logic is one level (i.e., a decoder)
- Good for speed
- Especially good for FPGA due to “Programmable Logic Block”
- Number of possible illegal states =  $2^n - n$
- Drawbacks:
  - Takes more area

# HDL FSM Implementation

## *State Encoding Trade-Offs*

- Typically trade off Speed, Area, and Power



# HDL FSM Implementation

## *FSM Coding goals*

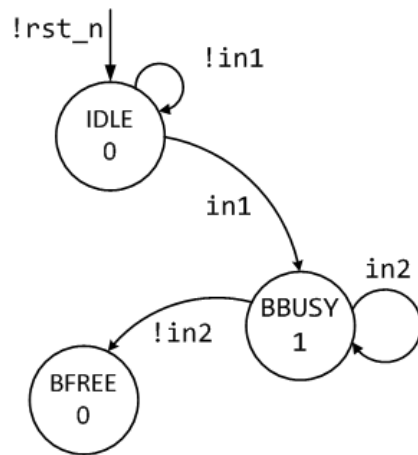
- The FSM coding style should be easily modified to change state encodings and FSM styles
- The coding style should be compact
- The coding style should be easy to code and understand
- The coding style should facilitate debugging
- The coding style should yield efficient synthesis results



# HDL FSM Implementation

## *Two Always Block FSM Style (Good Style)*

- One of the best Verilog coding styles
- Code the FSM design using two always blocks,
  - One for the sequential state register
  - One for the combinational next-state and combinational output logic.



```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01,BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx;  out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
         else      next = IDLE;
  BBUSY: begin
         out1 = 1'b1;
         if (in2) next = BBUSY;
         else      next = BFREE;
       end
  //...
endcase
end
```

# HDL FSM Implementation

## *Two Always Block FSM Style (Good Style)*

1. Use parameters to define state names because the state name is a constant that applies only to the FSM module
2. The sequential always block is coded using nonblocking assignments, in order to module sequential logic (accurately simulate hardware)

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01,BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx;  out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
         else      next = IDLE;
  BBUSY: begin
         out1 = 1'b1;
         if (in2) next = BBUSY;
         else      next = BFREE;
       end
  //...
endcase
end
```



# HDL FSM Implementation

## *Two Always Block FSM Style (Good Style)*

3. The combinational always block sensitivity list is sensitive to changes on the state variable and all of the inputs referenced in the combinational always block.
4. Assignments within the combinational always block are made using Verilog blocking assignments, in order to module sequential logic (accurately simulate hardware)

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01,BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx; out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
        else      next = IDLE;
  BBUSY: begin
        out1 = 1'b1;
        if (in2) next = BBUSY;
        else      next = BFREE;
        end
//...
endcase
end
```



# HDL FSM Implementation

## *Two Always Block FSM Style (Good Style)*

5. Default output assignments are made before coding the case statement (this eliminates latches and reduces the amount of code required to code the rest of the)
6. Placing a default next state assignment on the line immediately following the always block sensitivity list is a very efficient coding style

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
6  next = 2'bx; out1 = 1'b0; 5
  case (state)
    IDLE : if (in1) next = BBUSY;
           else   next = IDLE;
    BBUSY: begin
              out1 = 1'b1;
              if (in2) next = BBUSY;
              else   next = BFREE;
            end
    //...
  endcase
end
```

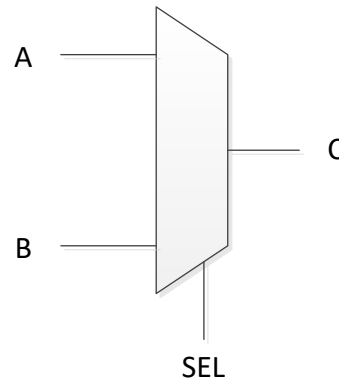
# What we Know

# Examples

- What is the behavior and matching logic for this code fragment?

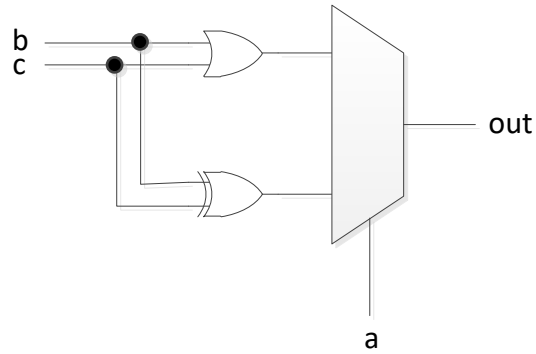
```
always @(clk or D)
  if (clock) Q >= D;
```

- What is the behavior and code for this schematic?



# Combinational Logic

- Combinational logic example
- How would you describe the behavior of this function in words?

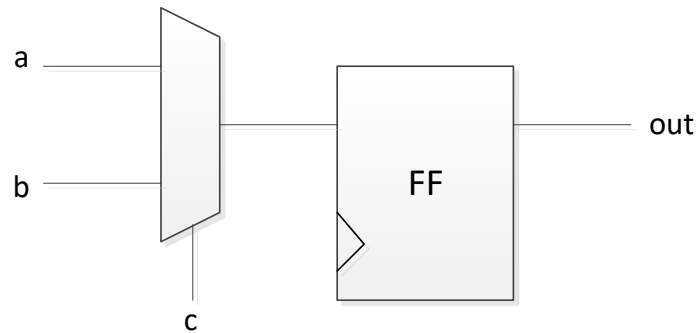


- And in code?

# Input Logic to Flip-Flops

- What is the behavior of the following code?
- What is the corresponding schematic?

```
always @(posedge clk)
  if (c)
    out <= b;
  else
    out <= a;
```





# Common Problems and Fixes

- Unintentional latches
  - Detected after reading the design
  - How to fix: make sure every variable is assigned for every way code is executed (except for flip flops)
  - If unfixed: you can have glitches on “irregular clock” to latch cause set up and hold problems in actual hardware (transient failures)

## Problem code

```
always @(a or b) begin
    if (a) c = ~b;
    else d = |b;
end
```

# Common Problems and Fixes

- Incomplete sensitivity list
  - Detected after reading the design
  - How to fix: all logic inputs have to appear in sensitivity list or use always @(\*) (Verilog 2001)
  - If unfixed: since simulation results won't match what actual hardware will do, bugs can remain undetected

## Problem code

```
always @(a or b) begin
    if (a) c = b ^ a;
    else c = d & e;
    f = c | a;
end
```

# Common Problems and Fixes

- Unintentional wired-or logic
  - Detected after reading the design, variables cannot be assigned in more than one block
  - How to fix: redesign hardware so that every signal is driven by only one piece of logic (or redesign s a tri-state buf if that is the intention)
  - If unfixed: unsynthesizable, this is a symptom of not designing before coding

## Problem code

```
always @(a or b)  
    c = |b;
```

```
always @(d or e)  
    c = ^e;
```

# Common Problems and Fixes

- Improper startup
  - Cannot be detected
  - How to fix: make sure don't cares are propagated
  - If unfixed: possible undetected bug in reset logic

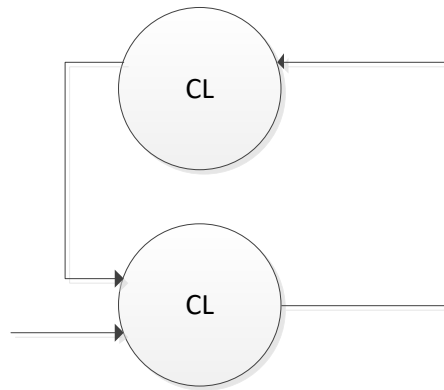
## Problem code

```
always @(posedge clock)
    if (a) q <= d;
```

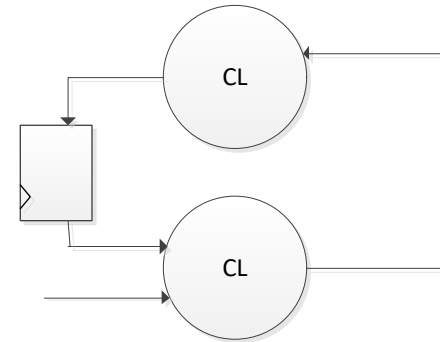
```
always @(q or e)
    case (q)
        0: f = e;
        default: f = 1;
    endcase
```

# Common Problems and Fixes

- Feedback in combinational logic
  - Either results in
    - Latches, when the feedback path is short
    - Timing Arcs when feedback path is convoluted
  - Fix by redesigning logic to remove feedback
    - Feedback can only be through FFs



**WRONG**



**OK**

# Common Problems and Fixes

- Incorrect use of FOR loops
  - Only correct use is to iterate through an array of bits
  - If in doubt, do not use it
- Unconstrained timing
  - To calculate permitted delay, synthesis must know where the FFs are
  - If you have a path from input port to output port that does not path through a FF, synthesis cannot calculate timing
  - To fix: revisit modules partitioning (see later) to include FFs in all paths

# Data storage and Verilog arrays

- Simple RAM Model

```
module RAM (output [7:0] Obus,  
            input  [7:0] Ibus,  
            input  [3:0] Adr,  
            input          Clk, Read  
            );  
reg [7:0] Storage [15:0];  
reg [7:0] ObusReg;  
  
assign Obus = ObusReg;  
  
always @(posedge Clk)  
    if (Read==1'b0) Storage[Adr] = Ibus;  
    else           ObusReg  = Storage[Adr];  
  
endmodule
```

# Data storage and Verilog arrays

- Counter

```
module cter (input  rst, clock, jmp,
             input  [7:0] jump,
             output reg [7:0] count
             );
always@(posedge clock)
begin
    if      (rst) count = 0;
    else if (jmp) count = jump + count;
    else      count = count + 1;
end
endmodule
```



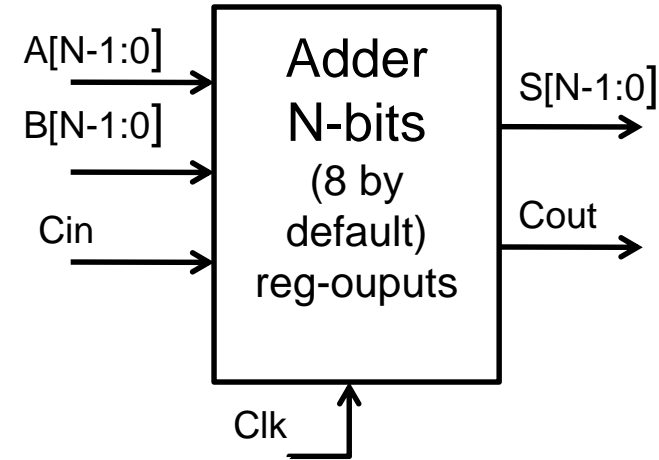
# Parameter example

```
module Adder (A, B, Cin, S, Cout, Clk);  
parameter N=8;  
input  [N-1:0]A, B;  
input  Cin;  
input  Clk;  
output [N-1:0] S;  
output Cout;  
reg [N-1:0] S;  
reg Cout;  
//module internals  
endmodule
```

Verilog 95  
Style

ANSI C style

```
module Adder #(parameter N=8)  
  (input [N-1:0]A, B,  
   input Cin,  
   input Clk,  
   output reg [N-1:0] S,  
   output reg Cout  
  );  
//module internals  
endmodule
```

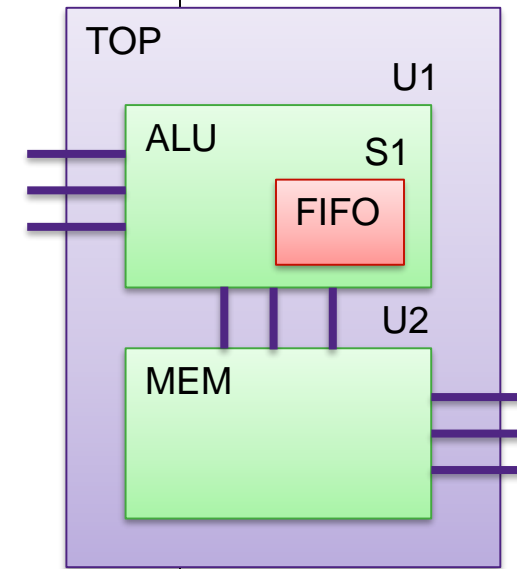


# Hierarchy Example

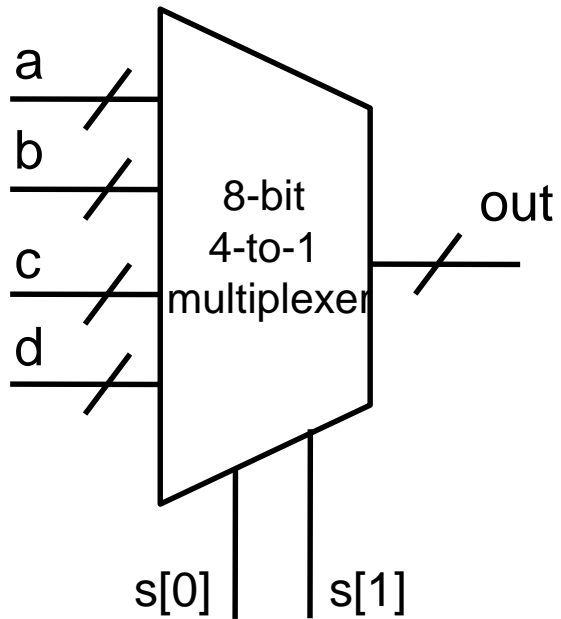
```
module TOP (input t1, t2, t3, output t4, t5, t6);  
  wire w1, w2, w3;  
  ALU #(S(2)) U1 (.a1(t1), .a2(t2), .a3(t3), .a4(w1), .a5(w2), .a6(w3));  
  MEM U2 (w1, w2, w3, t4, t5, t6);  
endmodule
```

```
module ALU (input a1, a2, a3, output a4, a5, a6);  
  parameter S=4;  
  reg rsl;  
  wire sig;  
  FIFO S1 (.f1(rsl), .f3(sig));  
  //...  
endmodule
```

```
module FIFO #(parameter F=2)  
  (input f1, output reg f2);  
  //...  
endmodule
```



# Example



```
module mux(a, b, c, d, s, out);  
  input  [7:0] a, b, c, d;  
  input  [1:0] s;  
  output [7:0] out;  
  reg    [7:0] out;  
  // used in procedural statement  
  always @ (s or a or b or c or d)  
    case (s)  
      2'b00: out = a;  
      2'b01: out = b;  
      2'b10: out = c;  
      2'b11: out = d;  
    endcase  
endmodule
```

# Thank You

