


# Introduction to Verilog HDL

Jorge Ramírez  
Corp Application Engineer

# Outline

- HDL Verilog
- Synthesis Verilog tutorial
- Synthesis coding guidelines
- Verilog - Test bench
- Fine State Machines
- References



- Lexical elements
- Data type representation
- Structures and Hierarchy
- Operators
- Assignments
- Control statements
- Task and functions
- Generate blocks

# HDL VERILOG

# 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
- Widely used HDLs
  - Verilog – Similar to C
  - SystemVerilog – Similar to C++
  - VHDL – Similar to PASCAL

# Verilog

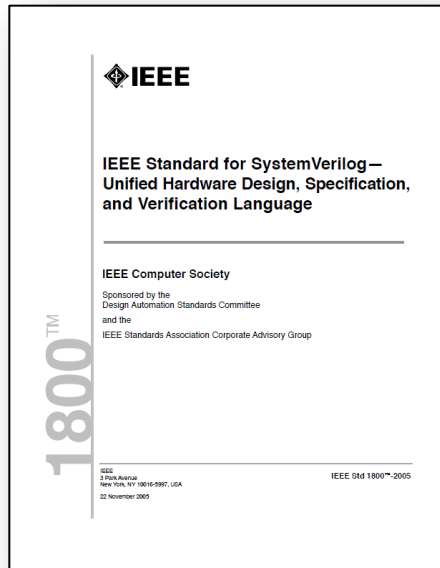
- Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984.
- Gateway was acquired by Cadence in 1989
- Verilog was made an open standard in 1990 under the control of Open Verilog International.
- The language became an IEEE standard in 1995 (IEEE STD 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

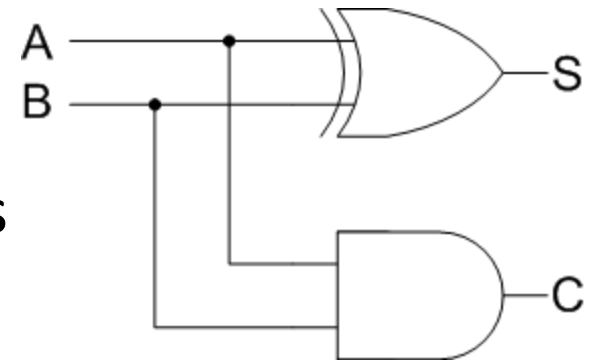
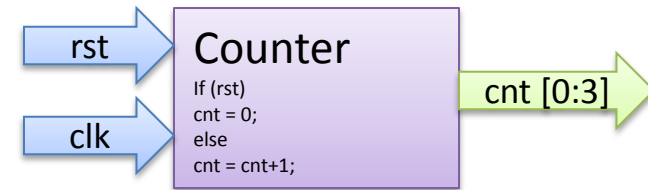
Verilog 2005 (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features



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

# 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





# Behavioral - Structural

## Behavioral

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

## Structural

```
module cter ( rst, clock, count );  
output [1:0] count;  
input  rst, clock;  
wire   N5, n1, n4, n5, n6;  
FFD U0 (.D(N5), .CP(clock),  
        .Q(count[0]), .QN(n6));  
FFD U1 (.D(n1), .CP(clock),  
        .Q(count[1]), .QN(n5));  
MUX21 U2 (.A(N5), .B(n4),  
          .S(n5), .Z(n1) );  
NR U3 (.A(n6), .B(rst), .Z(n4));  
NR U4 (.A(count[0]), .B(rst),  
      .Z(N5));  
endmodule
```

# 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

# Simulation and Synthesis

- Not all of the Verilog commands can be synthesized into hardware
- 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)

# SYNTHESIS VERILOG TUTORIAL

# Outline

- Lexical elements
- Data type representation
- Structures and Hierarchy
- Operators
- Assignments
- Control statements
- Task and functions
- Generate blocks

# Lexical elements

- Case sensitive - keywords are lower case
- Semicolons(;) are line terminators
- Comments:
  - One line comments start with `// ...`
  - Multi-line comments start with `/*and end with*/`
- System tasks and functions start with a dollar sign, ex `$display`, `$signed`

# Lexical elements

- Variable names have to start with an alphabetic character or underscore (\_) followed by alphanumeric or underscore characters
- Escaped identifiers (\)
  - Permit non alphanumeric characters in Verilog name
  - The escaped name includes all the characters following the backslash **until the first white space** character

```
wire \fo+o=a ; // Declare the varaible fo+o=a=a  
wire \fo+o =a ; // Assign a to wire fo+o
```

# Compiler directives

- The directives start with a grave accent ( ` ) followed by some keyword

``define`  
Text-macro substitution

``ifdef, `ifndef, `else, `endif`  
Conditional compilation

``include`  
File inclusion

```
`include "file1.v"
// Used as `WORD_SIZE in code
`define WORD_SIZE 32

module test ();
`ifdef TEST
// A implementation
`else
// B implementation
`endif
assign out = `WORD_SIZE{1'b1};
endmodule
```



# Reserved keywords

and	always	assign	attribute	begin	buf	bufif0	bufif1
case	cmos	deassign	default	defparam	disable	else	endattribute
end	endcase	endfunction	endprimitive	endmodule	endtable	endtask	event
for	force	forever	fork	function	highz0	highz1	if
initial	inout	input	integer	join	large	medium	module
nand	negedge	nor	not	notif0	notif1	nmos	or
output	parameter	pmos	posedge	primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release	repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small	specify	specparam	strong0	strong1
supply0	supply1	table	task	tran	tranif0	tranif1	time
tri	triand	trior	triereg	tri0	tri1	vectored	wait
wand	weak0	weak1	while	wire	wor		

# Outline

- Lexical elements
- **Data type representation**
- Structures and Hierarchy
- Operators
- Assignments
- Control statements
- Task and functions
- Generate blocks

# Logical values

- A bit can have any of these values
  - 0 representing logic low (false)
  - 1 representing logic high (true)
  - X representing either 0, 1, or Z
  - Z representing high impedance for tri-state (unconnected inputs are set to Z)

# Logical values

- Logic with multilevel (0,1,X,Z) logic values
  - Nand anything with 0 is 1
  - Nand two get an X
- True tables define the how outputs are compute

&	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

# Number representation

`<size>'<base format><number>`

- `<size>`:
  - number of bits (optional)
- `<base format>`:
  - It is a single character ' followed by one of the following characters **b**, **d**, **o** and **h**, which stand for binary, decimal, octal and hex, respectively.
- `<number>`
  - Contains digits which are legal for the `<base format>`
  - `'_'` underscore can be use for readability

# Number representation

- Negative numbers are store as 2's complement
- Extended number
  - If MSB is 0, X or Z number is extended to fill MSBs with 0, X, Z respectively

$$3'b01=3'b001 \quad 3'bx1=3'bxx1 \quad 3'bz=3'bzz$$

- If MSB is 1 number is extend to fill MSBs with 0/1, depending on the sign

$$3'b1=3'b001 \quad -3'b1=-3'b01=3'b111$$

# Number representation

Unsigned numbers (at least 32 bit)

```
549      // decimal number
'h8F_F   // hex number
'o765    // octal number
```

Size numbers

```
4'b11    // 4-bit binary number 0011
3'b10x    // 3-bit binary number with LSM bit unknown
8'hz      // 8-bit binary high-impedance number
4'hz1     // 4'bzzz1
5'd3      // 5-bit decimal number
```

Signed numbers

```
-8'd6     // 8-bit two's complement of 6 (-6)
4'shF     // 4-bit number '1111' to be interpreted as
           // 2's complement number
```

# Data types (reg)

- A reg (**reg**) stores its value from one assignment to the next (model data storage elements)
  - Don't confuse **reg** with *register*
  - Default value is X
  - Default range is one bit
  - By default are unsigned, but can be declare signed, using keyword **signed**



# Data types (Nets)

- Nets (**wire**) correspond to physical wires that connect instances
  - Nets do not store values
  - Have to be continuously driven
  - The default range is one bit
  - By default are unsigned
- The **wire** declaration is used most frequently, other net types are **wand**, **wor**, **tri**, **triand**, **trior**, etc.

# Other data types

- Integer (**integer**)
  - Convenient to counting purposes
  - At least 32-bit wide
  - Useful for loop
- Real (**real**) *simulation only*
  - Can be specified in decimal and scientific notation

# Verilog vectors

Know as BUS in hardware

- Declare by a range following the type

```
<data type> [left range : right range] <Variable name>
```

- Single element that is n-bits wide

```
reg [0:7] A, B; //Two 8-bit reg with MSB as the 0th bit
```

```
wire [3:0] Data; //4-bit wide wire MSB as the 4th bit
```

- Vector part select (access)

```
A[5] // bit # 5 of vector A
```

```
Data[2:0] // Three LSB of vector Data
```

# Verilog arrays

- Array: range follows the name

```
<datatype> <array name> [<array indices>]  
reg B [15:0]; // array of 16 reg elements
```

- Array of vectors

```
<data type> [<vector indices>]<array  
name> [<array indices>]  
reg [15:0] C [1023:0]; // array of vectors
```

- Memory access

```
<var name> [<array indices>] [<vector indices>]
```

# 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

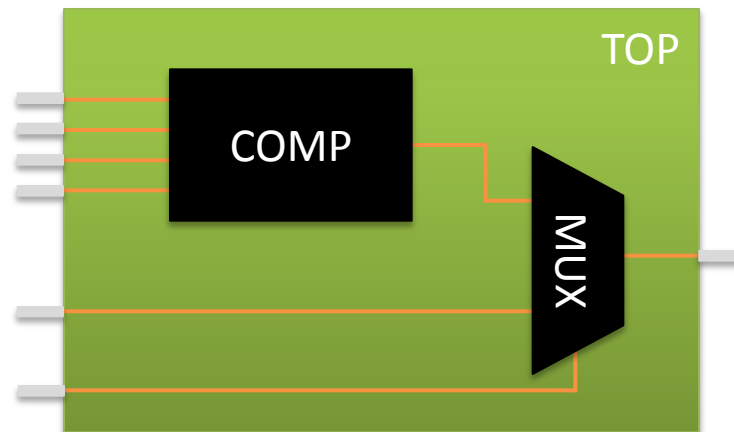
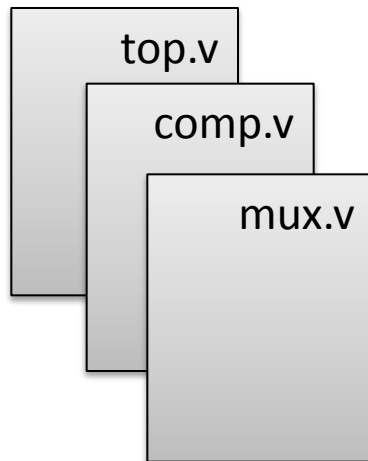
```
module cter (input  rst, clock, jmp,
             input  [7:0] jump,
             output reg [7:0] count
             );
always@(posedge clock)
begin
    if      (rst) count = 8'h00;
    else if (jmp) count = jump + count;
    else      count = count + 8'h1;
end
endmodule
```

# Outline

- Lexical elements
- Data type representation
- **Structures and Hierarchy**
- Operators
- Assignments
- Control statements
- Task and functions
- Generate blocks

# Structures and Hierarchy

- Hierarchical HDL structures are achieved by defining modules and instantiating modules





# Module declaration

```
module <module name> #(<param list>) (<port list>);  
  <Declarations>  
  <Instantiations>  
  <Data flow statements>  
  <Behavioral blocks>  
  <task and functions>  
endmodule
```

# Module header

- Start with `module` keyword, contains the I/O ports
- Port declarations begins with `output`, `input` or `inout` follow by bus indices
- Each directions are followed by one or more I/O names
- Each declaration is separated by comma (,)

```
module ALU (output [31:0] z,  
           input [15:0] A, B,  
           input clock, ena);
```

# Port declaration

- `input` and `inout` are declared as wires
- `outputs` port can be declared as `reg` (holds a value)
- 2 flavors for port declaration:

```
module S1 (a, b, c, d, e);  
input [1:0] a, b;  
input c;  
output reg [1:0] d;  
output e;  
//Verilog 1995 Style  
endmodule
```

```
module S2 (input [1:0] a, b,  
          input c,  
          output reg [1:0] d,  
          output e);  
  
//ANSI C Style  
endmodule
```

# Parameters

- Parameters are means of giving names to constant values
- The values can be overridden when the design is compiled
- Parameters cannot be used as variables
- Syntax:

```
parameter <name> = <constant expression>;
```

# Parameter declaration

- Default value need to be set at declaration time
- 32 bit wide by default, but may be declared of any width

```
parameter [2:0] IDLE = 3'd0;
```

- 2 declaration flavors:

## Inside a module

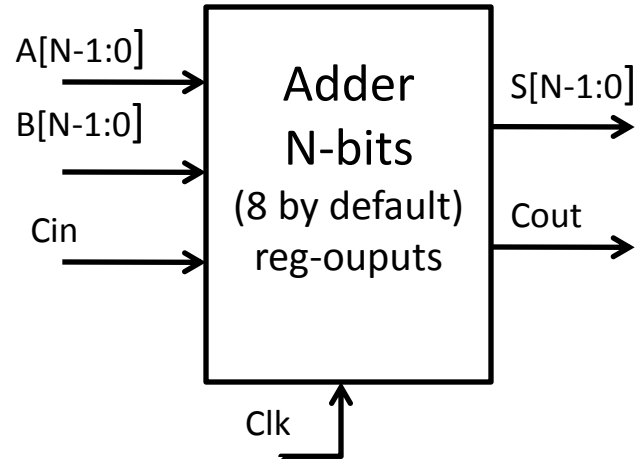
```
module test (... I/O's ...)  
parameter ASIZE = 32, BSIZE =16;  
//...  
reg [ASIZE -1:0] Abus, Zbus;  
wire [BSIZE-1:0] Bwire;  
//...  
endmodule
```

## In module header

```
module test  
    #(parameter ASIZE = 32, BSIZE =16)  
    (... I/O's ...);  
//...  
reg [ASIZE -1:0] Abus, Zbus;  
wire [BSIZE-1:0] Bwire;  
//...  
endmodule
```

# 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
```



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
```

# Structures and Hierarchy

- Instance of a module
  - Instantiation is the process of “calling” a module
  - Create objects from a module template

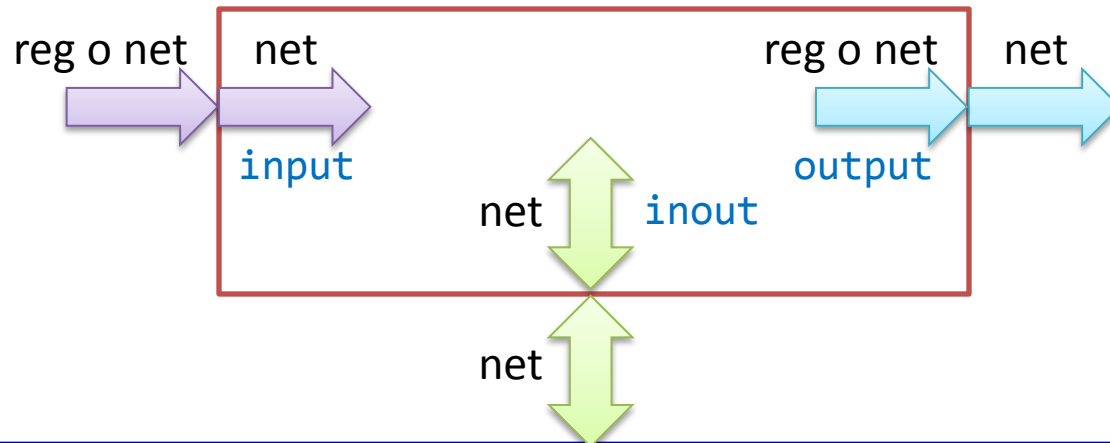
```
<module name> #(<param list>)  
                <instance name> (<port list>);
```

Where:

<module name>	Module to be instantiated
<param list>	Parameters values passed to the instance
<instance name>	Identifies the instance of the module
<port list>	Port list connection

# Port list connections

- Ports
  - Provide the interface by which a module can communicate with the environment
  - Port declarations (`input`, `output`, `inout`)





# Port connections/Parameter overwrite

- Named connection

- Explicitly linking the 2 names for each side of the connection

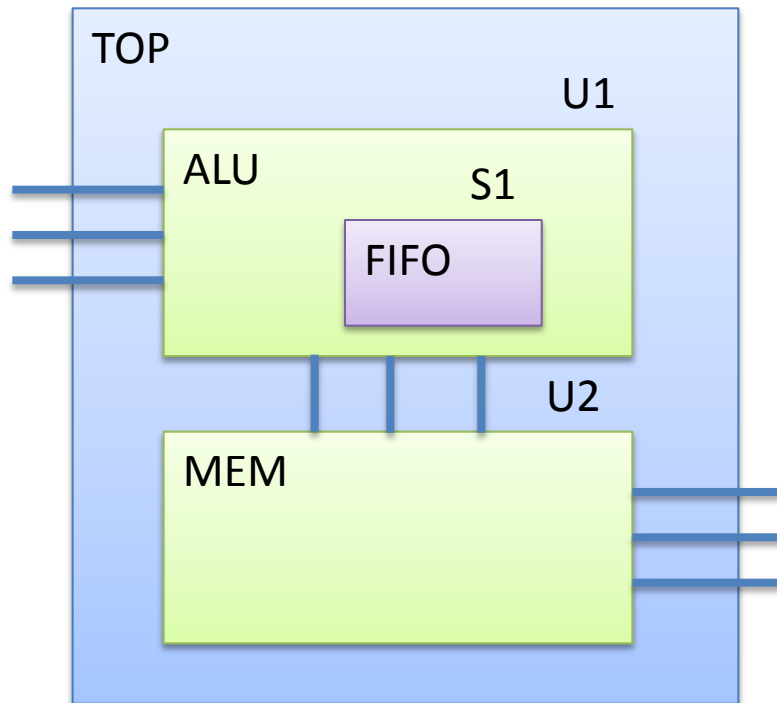
```
my_mod #(.W(1), .N(4)) U1 (.in1(a), .in2(b),  
                           .out(c));
```

- Order connection

- Expression shall be listed in the same order as the port declaration

```
my_mod #(1,4) U2 (a, b, c);
```

# Hierarchy example



```
module TOP (...port_list...);  
  ALU U1 (...port connection...);  
  MEM U2 (...port connection...);  
endmodule  
module ALU (...port_list...);  
  FIFO S1 (...port connection...);  
endmodule  
module FIFO (...port_list...);  
  //...  
endmodule
```

# Outline

- Lexical elements
- Data type representation
- Structures and Hierarchy
- **Operators**
- Assignments
- Control statements
- Task and functions
- Generate blocks

# Relational operators

- Mainly use in expression (e.g. if sentences)
- Returns a logical value (1/true 0/false)
- If there are any X or Z bit returns X (false on a expression)

```
<      a < b    // is a less than b?  
>      a > b    // is a greater than b?  
>=     a >= b   // is a greater than or equal to b  
<=     a <= b   // is a less than or equal to b
```

# Arithmetic operators

- Binary operators
  - Takes 2 operators
- Unary operators (+/-)
  - Specify the sign of the operand
  - Negative numbers are represented as 2's complement

```
*      c = a * b ;      // multiply a with b
/      c = a / b ;      // int divide a by b
+      sum = a + b ;    // add a and b
-      diff = a - b ;   // subtract b from a
%      amodb = a % b ;  // a mod(b)
```

# Logical operators

- Logical operators evaluate to a 1-bit value

```
&&    a && b ;    // is a and b true? returns 1-bit true/false
||    a || b ;    // is a or b true? returns 1-bit true/false
!     if (!a) c = b; // if a is not true assign b to c
```

- If an operand is not zero is treat as logical 1

```
A=3; B=0;
A&&B //Evaluates to 0 Equivalent to (logical-1 && logical-0)
!B   //Evaluates to 1 Equivalent to (!logical-0)
```

- If an operand is Z o X is treat as X (false)

```
A=2'b0x; B=2'b10;
A&&B //Evaluates to x Equivalent to (x && logical-0)
```

# Equality and Identity operators

```
==    c == a ; /* is c equal to a returns 1-bit true/false  
        applies for 1 or 0, logic equality, using X or  
        Z operands returns always false ('hx == 'h5  
        returns 0) */
```

```
!=    c != a ; // is c not equal to a, returns 1-bit true
```

```
===   a === b ; // is a identical to b (includes 0, 1, x, z)
```

```
!==   a !== b ; /* is a not identical to b returns 1-bit  
        true/false*/
```

# Bitwise and Reduction operations

<code>&amp;</code>	<code>b = &amp;a ;</code>	<code>/* AND all bits of a (reduction) */</code>
<code> </code>	<code>b =  a ;</code>	<code>/*OR all bits (reduction)*/</code>
<code>^</code>	<code>b = ^a ;</code>	<code>/*Exclusive or all bits of a (reduction)*/</code>
<code>~&amp;, ~ , ~^</code>	<code>c = ~&amp; b ;</code>	<code>/* NAND, NOR, EX-NOR all bits together */</code>
<code>~, &amp;,  , ^</code>	<code>b = ~a ; e = b   a</code>	<code>/*bit-wise NOT, AND, OR, EX-OR*/</code>
<code>~&amp;, ~ , ~^</code>	<code>e = a ~^ b ;</code>	<code>/*bit-wise NAND, NOR, EX- NOR*/</code>



# Shift and other operator

```
<<          a << 1 ;           // shift left a by 1-bit
>>          a >> 1 ;           // shift right a by 1
<<<         b <<< 1 ;          // arithmetic shift by 1
>>>         b >>> 1 ;          // arithmetic shift by 1


?:          c = sel ? a : b ;   /* if sel is true c = a, else c
                                = b , ?: ternary operator */

{}          {co, sum} = a + b + ci; /* add a, b, ci assign the
                                overflow to co and the result to
                                sum: operator is called
                                concatenation */

{{}}        b = {3{a}}         /* replicate a 3 times,
                                equivalent to {a,a,a} */
```

# Operators precedence

## *Operator precedence*

Unary, Multiply, Divide, Modulus	$+, -, !, \sim$ $*, / \%$	
Add, subtract, shift	$+, -$ $\ll, \gg$	
Relational Equality	$<, <=, >, >=$ $=, ==, !=$ $===, !==$	
Reduction Logical	$\&, \sim\&$ $\wedge, \wedge\sim$ $ , \sim $ $\&\&$ $  $	
Conditional	$?:$	

# Outline

- Lexical elements
- Data type representation
- Structures and Hierarchy
- Operators
- **Assignments**
- Control statements
- Task and functions
- Generate blocks

# Concurrent blocks

- Blocks of code with no well-defined order relative to one another
  - Module instance is the most important concurrent block
  - Continuous assignments, and procedural blocks are concurrent within a module

```
module AND (input A, B, output C);  
wire w;  
NAND U1 (A, B, w);  
NAND U2 (w, w, C);  
endmodule
```

# Continuous assignments

- Continuous assignments imply that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS
- Continuous assignments always implement combinational logic
- Continuous assignments drive wire variables

```
wire A;  
assign A = (B|C)&D;
```

# Continuous assignments

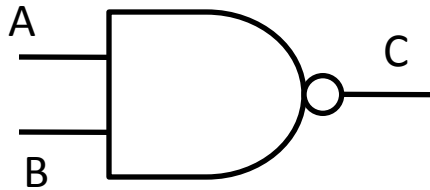
- Implicit continuous assignment
  - Continuous assignment can be placed when the net is declared

```
wire A = i1 & i2;
```

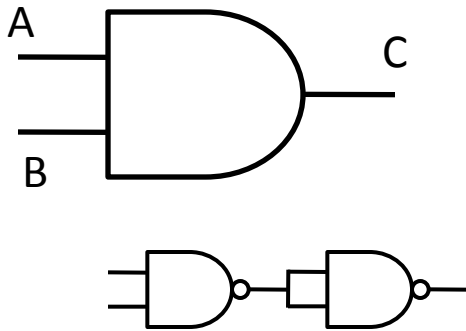
- Implicit net declaration (not recommended)
  - If a signal name is used to the left of a continuous assignment, an implicit net declaration will be inferred

```
wire i1, i2;  
assign A = i1 & i2;
```

# Example



```
module NAND (A, B, C);  
input A, B;  
output C;  
// Continuous assignments  
assign C = ~(A&B);  
endmodule
```



```
module AND (A, B, C);  
input A, B;  
output C;  
wire w;  
// 2 NAND instantiations  
NAND U1 (A, B, w);  
NAND U2 (w, w, C);  
endmodule
```

# Procedural blocks

- Each procedural block represent a separate activity flow in Verilog
- Procedural blocks
  - `always` blocks
    - To model a block of activity that is repeated continuously
  - `initial` blocks *simulation only*
    - To model a block of activity that is executed at the beginning
- Multiple behavioral statements can be grouped using keywords `begin` and `end`



# Procedural assignments

- Procedural assignment changes the state of a reg
- Used for both **combinational and sequential logic inference**
- All procedural statements must be within `always` (or `initial`) block

```
reg A;  
always @ (B or C)  
begin  
    A = ~(B & C);  
end
```

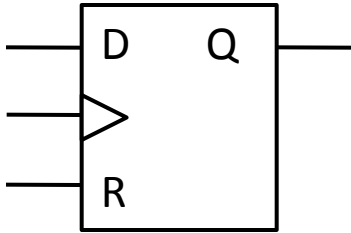
# Always block – Event control (@)

- Always blocks model an activity that is repeated continuously
- @ can control the execution
  - `posedge` or `negedge` make sensitive to edge
  - `@*` / `@(*)`, are sensitive to any signal that may be read in the statement group
  - Use “,” / `or` for multiple signals

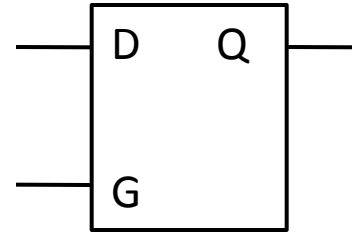
# Always block – Event control (@)

```
module M1 (input B, C, clk, rst, output reg X, Y,Z);  
// controlled by any value change in B or C  
always @ (B or C)  
    X = B & C;  
  
// Controlled by positive edge of clk  
always @(posedge clk)  
    Y = B & C;  
  
// Controlled by negative edge of clk or rst  
always @(negedge clk, negedge rst)  
    if (!rst) Z = B & C;  
    else      Z = B & C;  
endmodule
```

# Example



```
module FFD (input Clk, R, D,
            output reg Q);
always @ (posedge Clk)
begin
    if (R)
        Q = 1'b0;
    else
        Q = D;
end
endmodule
```



```
module LD (G, D, Q);
input G, D;
output Q;
reg Q;
always @(G or D)
    if (G)
        Q = D;
endmodule
```

# Blocking / Non-Blocking assignment

## Blocking assignment

- (= operator) acts much like in traditional programming languages
- The whole statement is done before control passes on to the next statement.

## Non-blocking assignment

- (<= operator) Evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit.

```
always @(posedge Clk)
begin
    //blocking procedural assignment
    C = C + 1;
    A = C + 1;
end

always @(posedge Clk)
begin
    //non-blocking procedural assignment
    D <= D + 1;
    B <= D + 1;
end
```

### Example: During every clock cycle

A is ahead of C by 1

B is same as D

# Procedural blocks (summary)

- Blocks of code within a concurrent block which are read (simulated, executed) in order
- Procedural blocks may contain:
  - Blocking assignments
  - Nonblocking assignments
  - Procedural control statements (`if`, `for`, `case`)
  - `function`, or task calls
  - Event control ( `'@'` )
  - Nested procedural blocks enclosed in `begin ... end`

# Outline

- Lexical elements
- Data type representation
- Structures and Hierarchy
- Operators
- Assignments
- **Control statements**
- Task and functions
- Generate blocks

# Conditional statements (`if ... else`)

- The statement occurs if the expressions controlling the if statement evaluates to true
  - True: 1 or non-zero value
  - False: 0 or ambiguous (X)
- Explicit priority

```
if (<expression>)  
// statement1  
else if (<expression>)  
// statement2  
else  
// statement3
```

```
always @ (WRITE or STATUS)  
begin  
  if (!WRITE)  
  begin  
    out = oldvalue;  
  end  
  else if (!STATUS)  
  begin  
    q = newstatus;  
  end  
end
```



# Conditional statements (**case**)

- **case**, **casex**, **casez**: case statements are used for switching between multiple selections
  - If there are multiple matches only the first is evaluated
  - Breaks automatically
- **casez** treats Z as don't care
- **casex** treats Z and X as don't care

```
case (<expression>)  
    <alternative 1> : <statement 1>;  
    <alternative 2> : <statement 2>;  
    default         : <default statement>;  
endcase
```

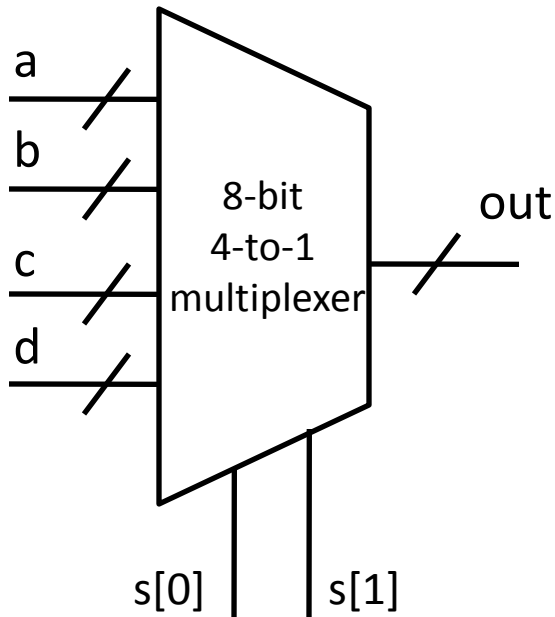
# Conditional statements (**case**)

```
always @(s, a, b, c, d)
case (s)
  2'b00: out = a;
  2'b01: out = b;
  2'b10: out = c;
  2'b11: out = d;
endcase
```

```
always @*
casez (state)
// 3'b11z, 3'b1zz, ... match
3'b1??
3'b1??: fsm = 0;
3'b01?: fsm = 1;
endcase
```

```
always @*
casex (state)
/*
during comparison : 3'b01z,
3'b01x, 3'b'011 ... match case
3'b01x
*/
3'b01x: fsm = 0 ;
3'b0xx: fsm = 1 ;
default: fsm = 1 ;
endcase
```

# 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
```

# Latches / Muxes (Comb logic)

- Assuming only level sensitivity on a always block:
  - A variable or signal when is fully specified (it is assigned under all possible conditions) a mux or combinational logic.
  - If a variable or signal is not fully specified a latch will be inferred

<pre>always @ (a,b,sel)   if (sel==1'b1)     z=a;   else     z=b;</pre>	<pre>always @ (DATA, GATE)   if (GATE)     Q = DATA;</pre>	<pre>always @ (DATA, GATE) begin   Q = 0;   if (GATE) Q = DATA; end</pre>
---	--	---

MUX

LATCH

COMB LOGIC

# Loop statements (**for**)

- Works the same ways as C
- Unary increment/decrement is not allowed

```
for (<loop var init>; <loop var reentry expr>; <loop var update>)  
<statement>;
```

```
// General purpose loop  
integer i;  
always @*  
for (i = 0 ; i < 7 ; i=i+1)  
    memory[i] = 0;
```

# Loop statements (**while**)

- Loop execute until the expression is not true

```
always @*  
while(delay)  
// multiple statement groups with begin-end  
begin  
    ldlang = oldldlang;  
    delay  = delay - 1;  
end
```

# Loop statements (**repeat**)

- Repeat statement a specified number of times
- The number is evaluated only at the beginning

```
always @*  
repeat(`BIT-WIDTH)  
begin  
    if (a[0]) out = b + out;  
    a = a << 1;  
end
```

# Outline

- Lexical elements
- Data type representation
- Structures and Hierarchy
- Operators
- Assignments
- Control statements
- **Task and functions**
- Generate blocks



# Tasks and Functions

Task and function serve the same purpose on Verilog as subroutines do in C

## Task:

- Declare with `task` and `endtask`
- May have zero arguments or more arguments of type `input`, `output`, `inout`
- Do not return with a value, can pass values through `output` and `inout` arguments

## Functions:

- Declare with `function` and `endfunction`
- Must have at least one input
- Always return a single value (cannot have `output` or `inout` arguments)

# Tasks and Functions - example

```
module top (input a1, a2, output reg [1:0] b1, b2);
always @ (a1, a2)
begin
    b1 = out (a1, a2);      // function calling
    out_task (a1, a2, b2); // task calling
end
function [1:0] out (input in1, in2); // Function Declaration
begin
    if (in1) out = {in2,in1}; else out = {in1,in2};
end
endfunction
task out_task (input in1, in2, output [1:0] out); // Task Declaration
begin
    if (in1) out = {in2,in1}; else out = {in1,in2};
end
endtask
endmodule
```

# Task and Functions

- Functions are simpler

Function	Task
Can call another function	Can call another function or task
Can modify only one value	Can modify multiple values

- Data Sharing

- Functions and task could be declare as **automatic**
- A static function retains the value of all it's internal variables between calls. An automatic function re-initializes them each call

# Outline

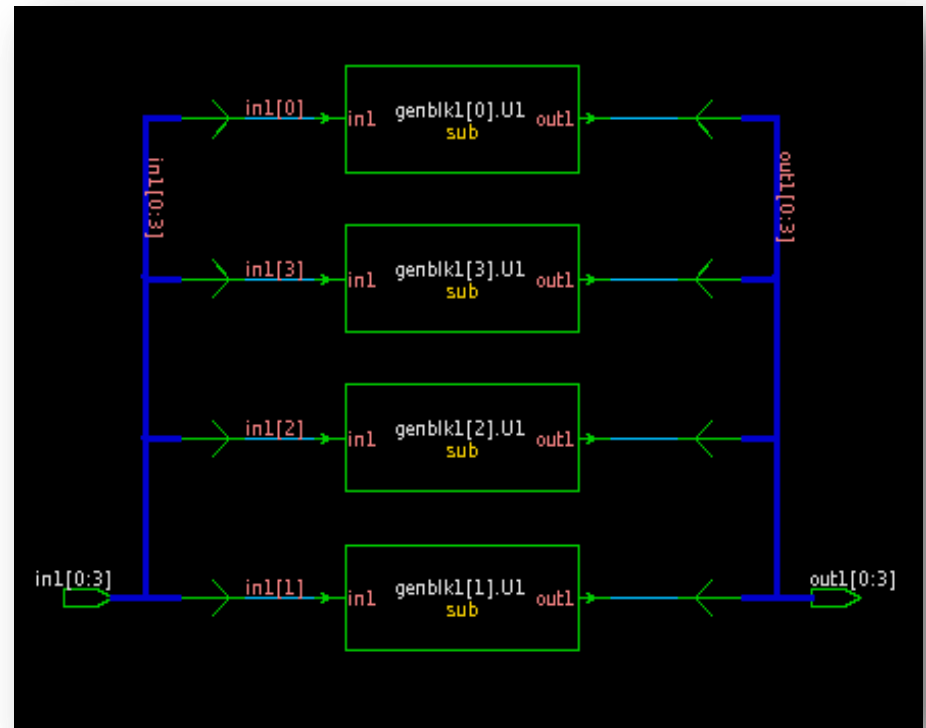
- Lexical elements
- Data type representation
- Structures and Hierarchy
- Operators
- Assignments
- Control statements
- Task and functions
- **Generate blocks**

# Generate blocks

- Allow to generate Verilog code dynamically at elaboration time
  - Facilitated the parameterized model generation
  - Required the keywords `generate` – `endgenerate`
  - Generate instantiations can be
    - Module instantiations
    - Continuous assignments
    - `initial` / `always` blocks
  - Typically used are generate loop and conditional generate

# Generate loop

```
module top( input [0:3] in1,  
           output [0:3] out1);  
  // genvar control the loop  
  genvar I;  
  generate  
  for( I = 0; I <= 3; I = I+1 )  
  begin  
    sub U1(in1[I], out1[I]);  
  end  
endgenerate  
endmodule
```



# Conditional generate

```
module top #(parameter POS=0)
    (input in, clk, output reg out);

generate
if(POS==1)
always @ (posedge clk)
    out = in;
else
always @ (negedge clk)
    out = in;
endgenerate
endmodule
```

# SYNTHESIS CODING GUIDELINES



# Synthesis coding guidelines

- Inferring Three-State Drivers
  - Never use high-impedance values in a conditional expression (Evaluates expressions compared to high-impedance values as false)
- Sensitivity Lists
  - You should completely specify the sensitivity list for each always block. Incomplete sensitivity lists can result in simulation mismatches

```
always @ (A)  
C <= A | B;
```

# Synthesis coding guidelines

- Value Assignments
  - The hardware generated by blocking assignments (=) is dependent on the ordering of the assignments
  - The hardware generated by nonblocking assignments (<=) is independent of the ordering of the assignments
  - For correct simulation results, Use nonblocking assignments within sequential Verilog always blocks

# Synthesis coding guidelines

- Value Assignments
  - Do not mix blocking and nonblocking assignments in the same always block
  - Do not make assignments to the same variable from more than one always block. It is a Verilog race condition, even when using nonblocking assignments
- Structures and Hierarchy
  - Place one module per file

# Synthesis coding guidelines

- If Statements
  - When an if statement used in a Verilog always block as part of a continuous assignment does not include an else clause, synthesis tool may create a latch.
- Case Statements
  - If your if statement contains more than three conditions, consider using the case statement to improve the parallelism of your design and the clarity of your code
  - An incomplete case statement results in the creation of a latch

# Synthesis coding guidelines

- Constant Definitions
  - Use the Verilog ``define` statement to define global constants
  - Keep global constant definitions in a separate file.
- Using Verilog Macro Definitions
  - In Verilog, macros are implemented using the ``define` statement
  - Keep ``define` statements in a separate file.
  - Do not use nested ``define` statements (difficult to read)

# Synthesis coding guidelines

- Guidelines for Identifiers
  - Ensure that the signal name conveys the meaning of the signal or the value of a variable without being verbose
  - Use a consistent naming style for capitalization and to distinguish separate words in the name.
  - Examples:
    - C style uses lowercase names and separates words with an underscore, for example, `packet_addr`, `data_in`, and `first_grant_enable`
    - Pascal style capitalizes the first letter of the name and first letter of each word, for example, `PacketAddr`, `DataIn`, and `FirstGrantEnable`

# JUST FOR FUN

# Question 1

```
input  [7:0] data;  
output [7:0] data_reversed;
```

Can the bit reversal be done by applying the next assignment?

```
data_reversed = data[0:7];
```

```
data_reversed = {data_8[0], data_8[1], data_8[2], data_8[3],  
                data_8[4], data_8[5], data_8[6], data_8[7]};
```

```
input  [127:0] data;  
output [127:0] data_reversed;  
for (i = 127; i >= 0; i = i - 1)  
    data_reversed[i] = data[127 - i];
```



# Question 2

```
parameter WIDTH = 4;
```

```
output [WIDTH - 1 : 0] data;
```

How we can assign to each bit a 1'b1?

```
assign data= {WIDTH{1'b1}};
```

```
assign data= (1 << WIDTH) - 1;
```

# Question 3

Which is the correct macro usage?

```
`define TEST
`define SIZE 32

module test ();
`ifdef TEST
assign out = SIZE{1'b1};
`else
assign out = SIZE{1'b0};
`endif
endmodule
```



```
`define TEST
`define SIZE 32

module test ();
`ifdef TEST
assign out = `SIZE{1'b1};
`else
assign out = `SIZE{1'b0};
`endif
endmodule
```



```
`define TEST
`define SIZE 32

module test ();
`ifdef `TEST
assign out = `SIZE{1'b1};
`else
assign out = `SIZE{1'b0};
`endif
endmodule
```



# Question 4

- Which code infer a asynchronous reset FF?

```
module FFD (input Clk, R, D,
            output reg Q);
always @ (posedge Clk, negedge R)
begin
    if (!R)
        Q = 1'b0;
    else
        Q = D;
    end
endmodule
```



```
module FFD (input Clk, R, D,
            output reg Q);
always @ (posedge Clk, negedge R)
begin
    if (R)
        Q = 1'b0;
    else
        Q = D;
    end
endmodule
```



# Question 5

Is the next module header correct?

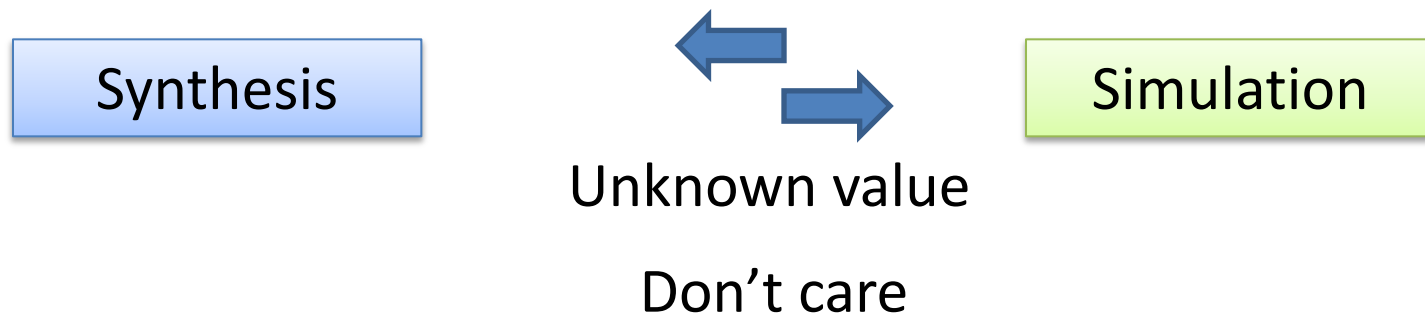
```
module Adder(input [N-1:0]A, B,  
            input Cin,  
            input Clk,  
            output reg [N-1:0] S,  
            output reg Cout);  
  
parameter N=8;  
//module internals  
endmodule
```

NO!  
Parameter is used  
before it is declared !

```
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
```

# Question 6

What does X means in synthesis and simulation?



Assigning “X” to a wire or reg is highly encouraged for synthesis: it specifies a don't care condition, letting the synthesis tool do further optimization

Be aware, when assigning X's they may propagate throughout your design under simulation

# VERILOG TEST BENCH

# Writing Test Bench

- A test bench specifies a sequence of inputs to be applied by the simulator to an Verilog-based design.
- The test bench uses an initial block and delay statements and procedural statement.
- Verilog has advanced “behavioral” commands to facilitate this:
  - Delay for n units of time
  - Full high-level constructs: if, while, sequential assignment.
  - Input/output: file I/O, output to display, etc.

# Test Bench

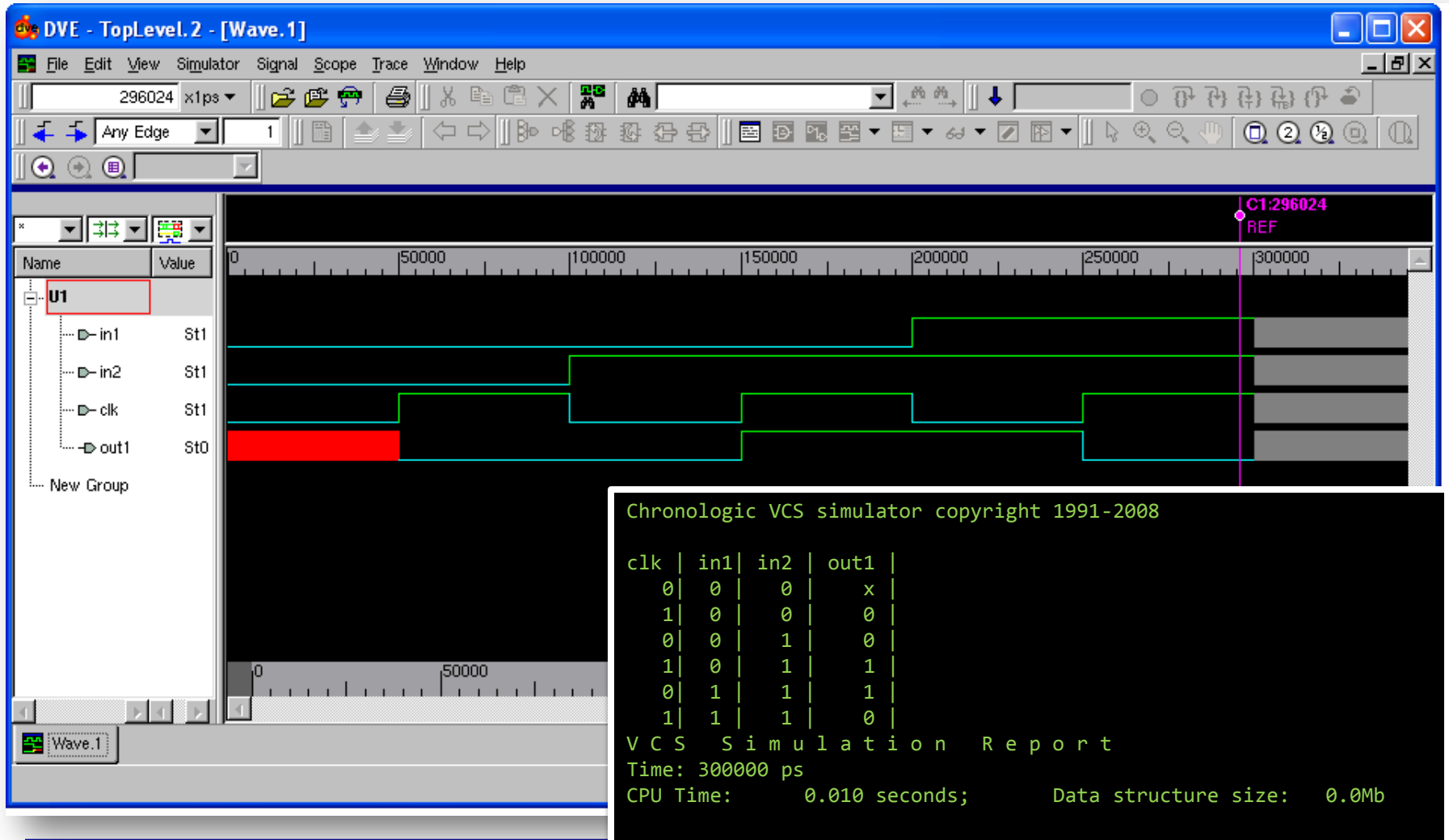
```
`timescale 10ns/1ps
module test_bench;
// Interface to communicate with the DUT
reg a, b, clk;
wire c;
// Device under test instantiation
DUT U1 (.in1(a), .in2(b), .clk(clk), .out1(c));
initial
begin // Test program
    test1 ();
    $finish;
end
initial
begin
    clk = 0;
    forever #5 clk = ~clk;
end
initial
begin // Monitor the simulation
    $dumpvars;
    $display ("clk | in1 | in2 | out1 |");
    $monitor (" %b | %b | %b | %b |", clk, a, b, c);
end
endmodule
```

```
module DUT (in1, in2, clk, out1);
input in1, in2;
input clk;
output reg out1;
always @(posedge clk)
    out1 = in1^in2;
endmodule
```

```
task test1 ();
begin
    a = 0;    b = 0;
    #10 a = 0; b = 1;
    #10 a = 1; b = 1;
    #10 a = 1; b = 0;
end
endtask
```



# Simulation results



# Precision macro

- ``timescale`
  - Defines the time units and simulation precision (smallest increment)
    - ``timescale Time_Unit/Precision_Unit`
  - Time : 1 10 100
  - Units: ms us ns ps fs
  - The precision unit must be less than or equal to the time unit
  - Example:

```
`timescale 10ns/1ps
```

# Initial block

- `initial` block
  - Contains a statement or block of statement which is executed only once, starting at the beginning of the simulation
  - Each block is executed concurrently before simulation time 0 (ignore by synthesis)
  - No sensitivity list

```
initial
begin
    X = 1'b0;
end
```

# Forever block

- `forever` block
  - Cause one or more statements to be executed in an infinite loop.
  - Example: clock signal generation

```
initial
begin
  clk = 0;
  forever #5 clk = ~clk;
end
```

# Delay

- Delay (#)
  - Specifies the delay time units before a statement is executed during simulation
  - Regular delay control
    - `#<num> y = 1;`
      - Regular delay control is used when a non-zero value is specified
  - Zero delay control
    - `#0;`
      - Ensure that statements are executed at the end of time 0
  - Intra-assignment delay control
    - `y = #<num> x+z;`

# Delay example

```
parameter sim_cycle = 6;
initial
begin
    x=0;
    #10 y=1;           //assignment is delayed 10 time units
    #(sim_cycle/3) x=2; //delay number can come from a parameter
end
initial
begin
    p = 0; q = 0;
    r = #5 p+q;       //Take the value of p and q at time 0, evaluate
                    // p+q and wait 5 time units to assign value to r
end
initial
begin
    #0 x =1;         //x=0,p=0;q=0,x=1 are executed a time 0 but x=1 is
                    //executed at the end
end
```

# Task

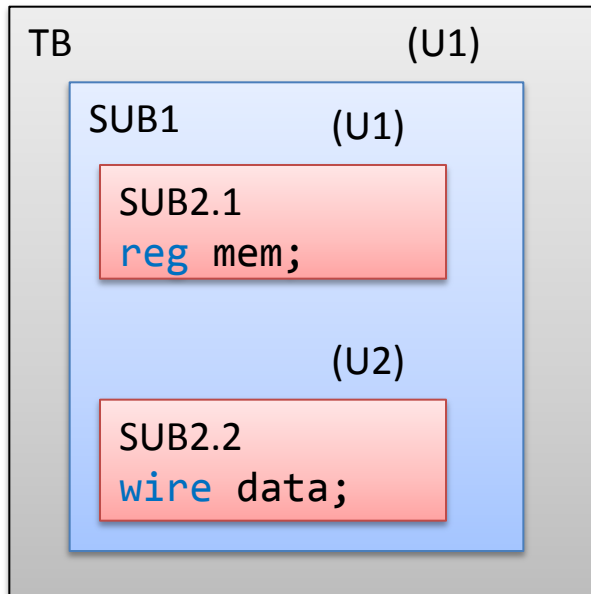
- Task helps to simplify the test bench
- Can include timing control
  - Inputs
  - Delays (#) and regular event control (@)

```
task load (input [7:0] data,  
          input enable);  
begin  
    #2  rst_n = 1'b0;  
    #10 data_in=data;  
    #2  read_ena=enab;  
end  
endtask
```

```
task reset;  
begin  
    rst_n = 1'b0;  
    repeat(3)  
        @(negedge clk);  
    rst_n = 1'b1;  
end  
endtask
```

# Hierarchical names

- Hierarchical name references allows us to denote every identifier in the design with a unique name
- Hierarchical name is a list of identifier separated by dots (“.”)



```
module TB ();  
//...  
$monitor ("%b", TB.U1.U1.mem);  
$monitor ("%b", TB.U1.U2.data);  
  
endmodule
```



# System tasks and functions

- System tasks are tool specific tasks and functions.

```
$display, $write // utility to display information
$monitor        // monitor, display
$time, $realtime // current simulation time
$finish         // exit the simulator
$stop          // stop the simulator
$timeformat     // format for printing simulation
$random        // random number generation
$dumpvars      // dump signals to file
```

# \$display - \$strobe - \$monitor

- Print message to a simulator (similar to C printf)
- **\$display** (*format, args*)
  - Display information before RHS evaluation (before nonblocking assignments)
- **\$strobe** (*format, args*)
  - Display information after RHS evaluation (after nonblocking assignments)
- **\$monitor** (*format, args*)
  - Print-on-change after RHS and nonblocking assignments whenever one argument change

# Verilog string and messages

```
initial
begin
$display ("Results\n");
$monitor ("\n Time=%t  X=%b", $time , X);
end
```

## Useful format specifiers:

**%h** hex integer

**%d** decimal integer

**%b** binary

**%c** single ASCII character

**%s** character string

**%t** simulation time

**%u** 2-value data

**%z** 4-value data

**%m** module instance name

# \$random

- Random number generation
- Returns as 32-bit signed integer

```
reg [31:0] rand1, rand2, rand3;  
rand1 = $random;           // generates random numbers  
rand2 = $random % 60;      // random numbers between -59 and 59  
rand3 = {$random} % 60;    // random positive values  
                           // between 0 and 59
```

# \$dumpvars

- The Verilog `$dumpvars` command is used to generate a value change dump (VCD)
- VCD is an ASCII file that contains information about simulation, time, scope and signal definition, and signal value changes
- VCD files can be read on graphical wave from displays

```
initial
begin // Monitor the simulation
    $dumpfile ("myfile.dump");
    $dumpvars;
end
```

# Test Bench

```
`timescale 10ns/1ps
module test_bench;
// Interface to communicate with the DUT
reg a, b, clk;
wire c;
// Device under test instantiation
DUT U1 (.in1(a), .in2(b), .clk(clk), .out1(c));
initial
begin // Test program
    test1 ();
    $finish;
end
initial
begin
    clk = 0;
    forever #5 clk = ~clk;
end
initial
begin // Monitor the simulation
    $dumpvars;
    $display ("clk | in1 | in2 | out1 |");
    $monitor (" %b| %b | %b | %b |",clk, a, b, c);
end
endmodule
```

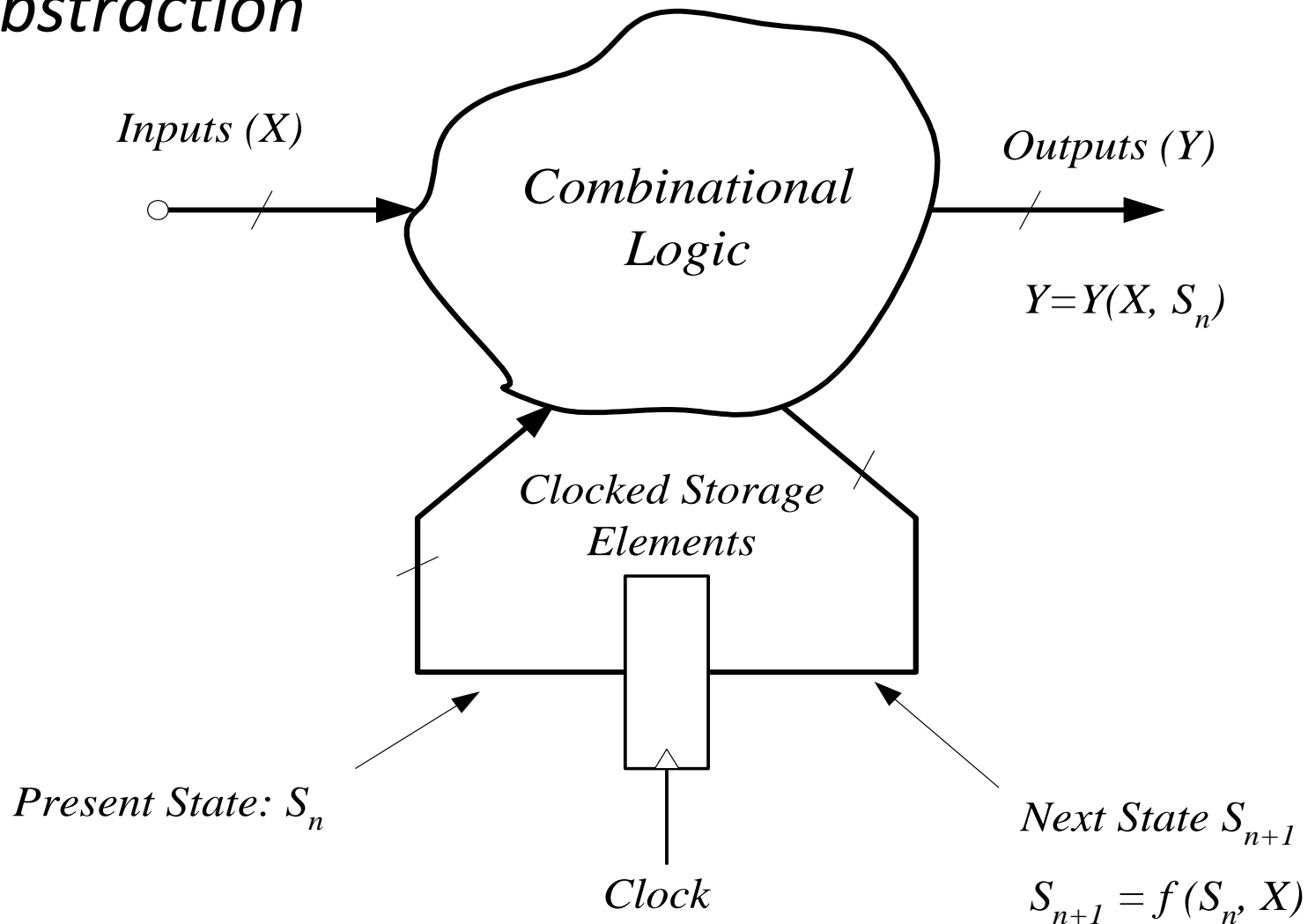
```
module DUT (in1, in2, clk, out1);
input in1, in2;
input clk;
output reg out1;
always @(posedge clk)
    out1 = in1^in2;
endmodule
```

```
task test1 ();
begin
    a = 0;    b = 0;
    #10 a = 0;    b = 1;
    #10 a = 1;    b = 1;
    #10 a = 1;    b = 0;
end
endtask
```

# FINITE STATE MACHINE

# Finite-State machine

## Abstraction





# Finite-State machine

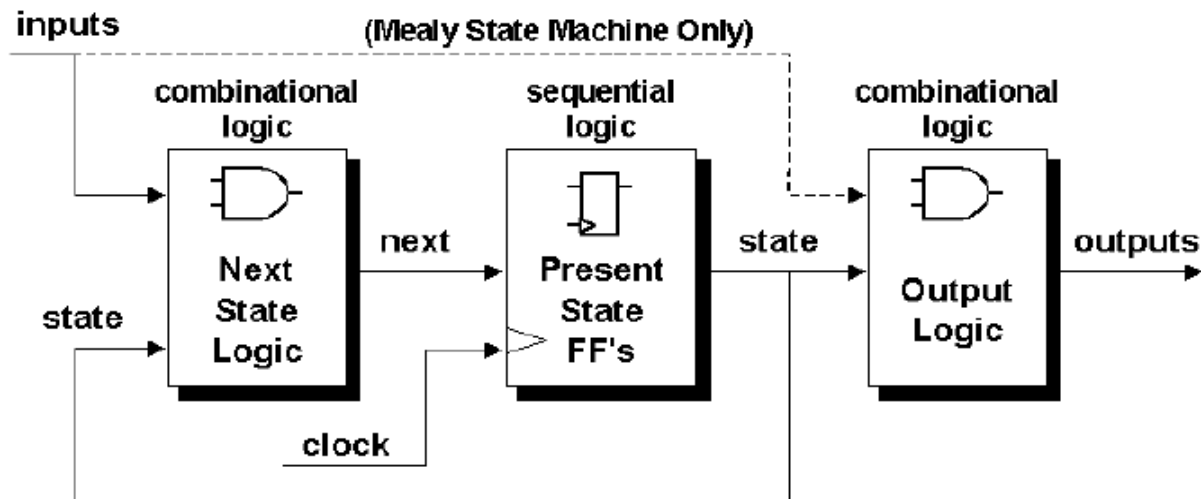
## *Abstraction*

- 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

# Finite State machine

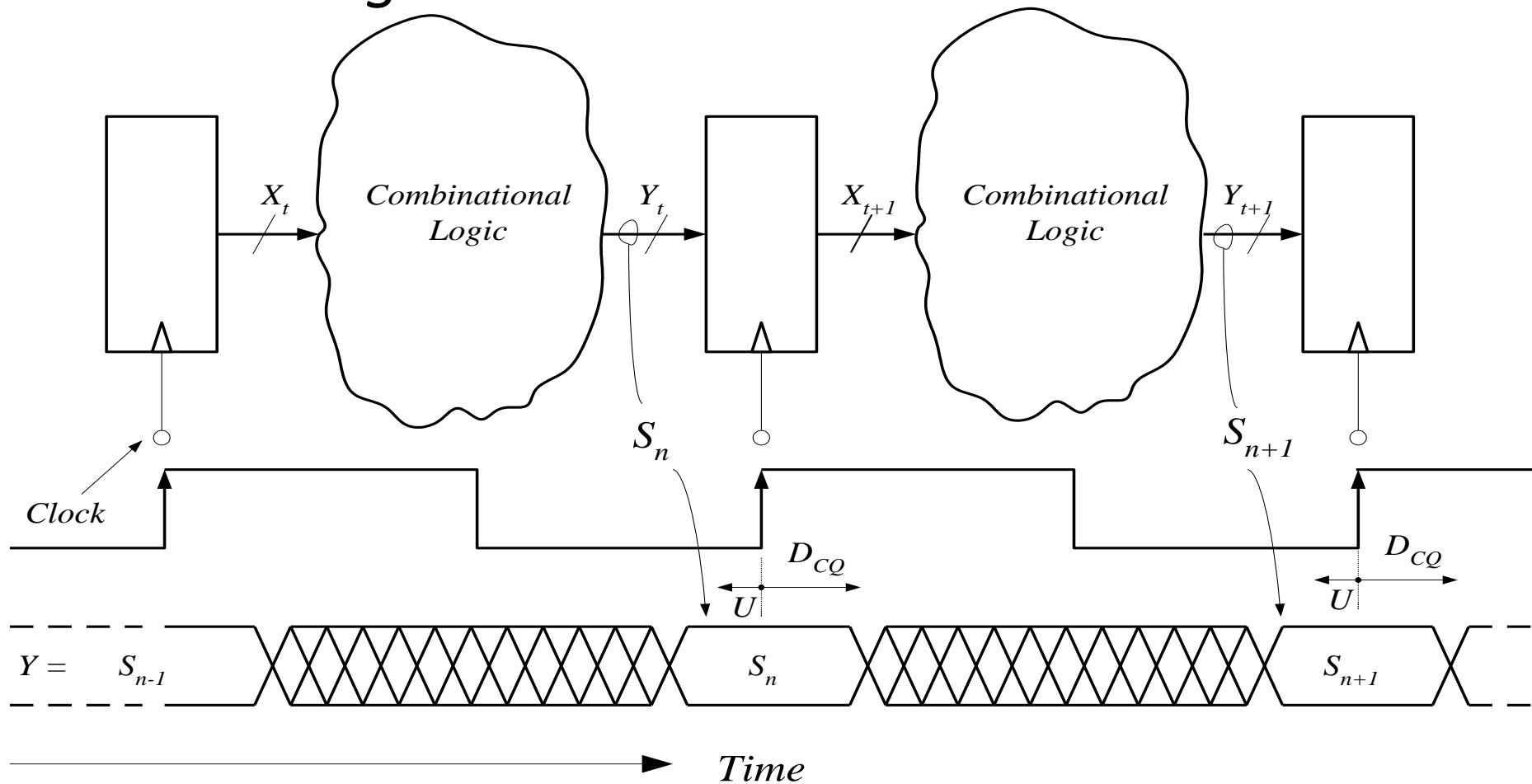
## *Mealy and More*

- Mealy and More FSM
  - A Moore FSM is a state machine where the outputs are only a function of the present state.
  - A Mealy FSM is a state machine where one or more of the outputs is a function of the present state and one or more of the inputs.



# Finite-State machine

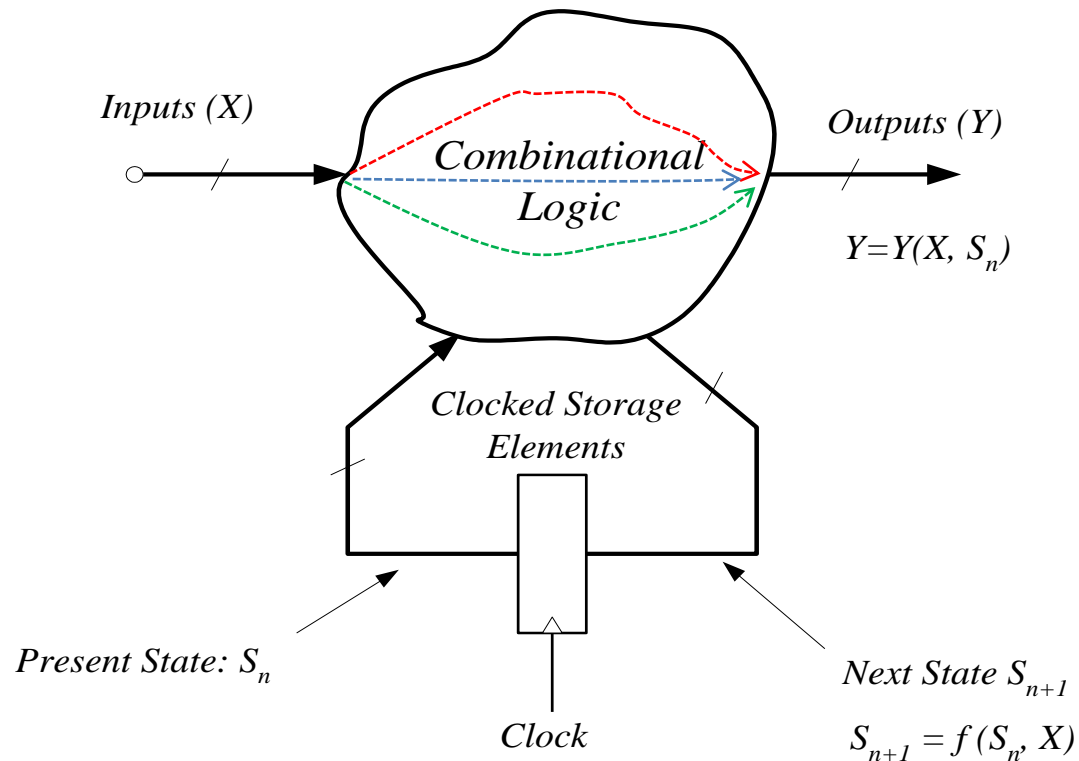
## State Changes



# Finite-State machine

## Critical Path

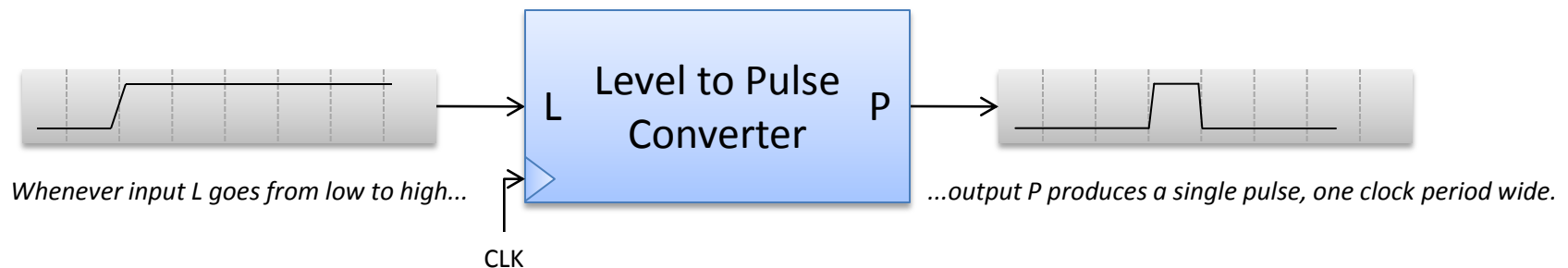
Critical path is defined as the chain of gates in the longest (slowed) path through the logic



# FSM Implementation

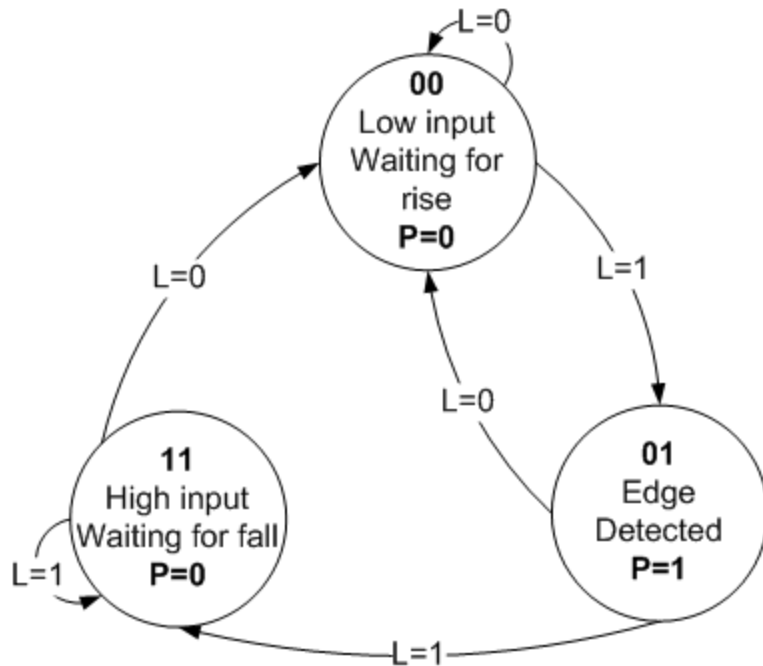
## *Design example*

- 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

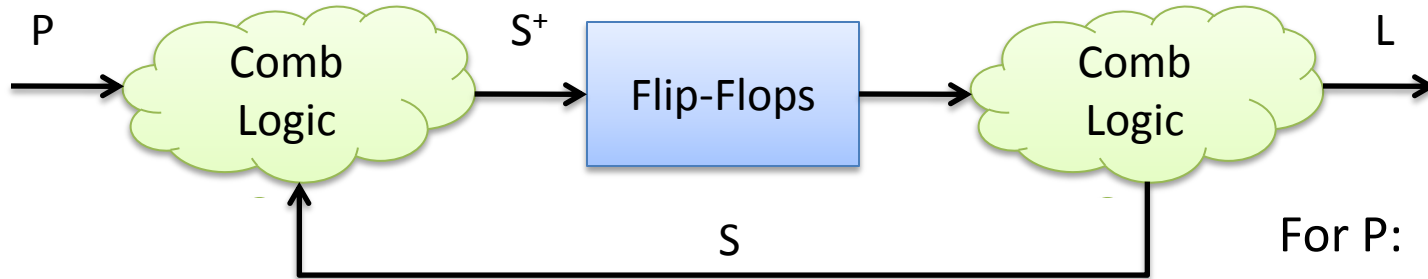
## *State Diagram (Moore implementation)*



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  $S_1^+$ :

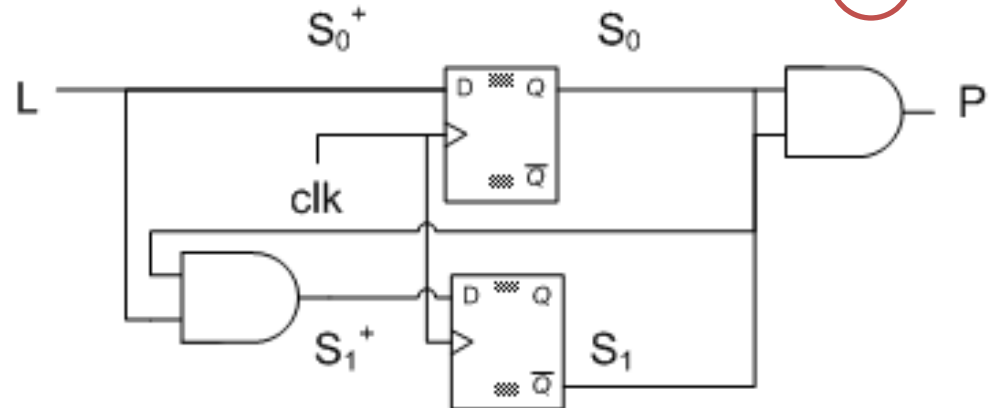
$L \backslash S_1 S_0$	00	01	11	10
0	0	0	0	X
1	0	1	1	X

For P:

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

For  $S_0^+$ :

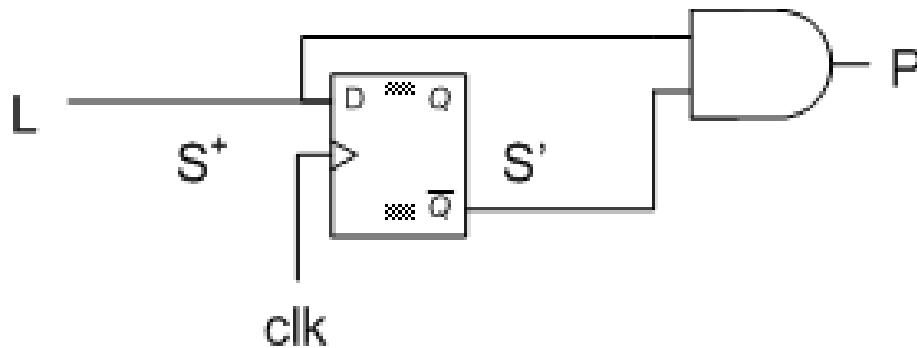
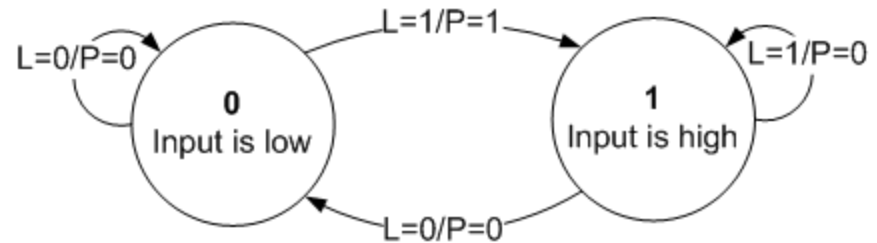
$L \backslash S_1 S_0$	00	01	11	10
0	0	0	0	X
1	1	1	1	X



# FSM Implementation

## *Mealy implementation*

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



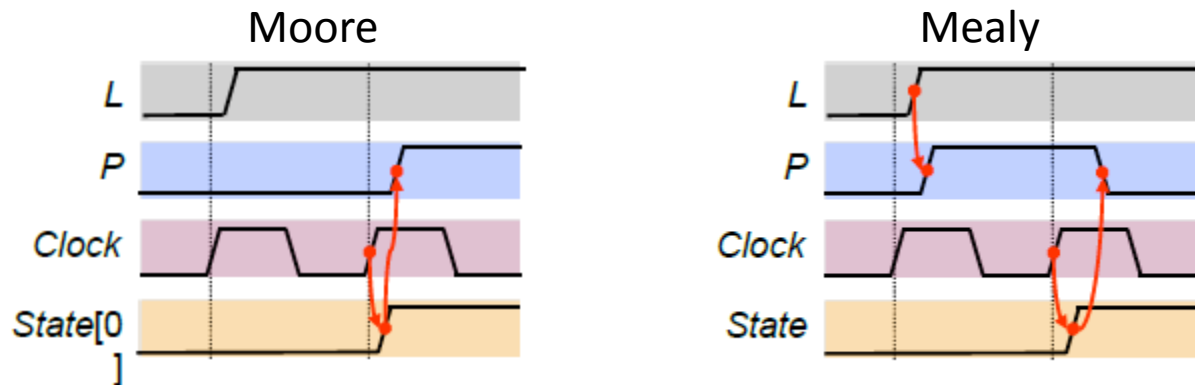
Pres State	In	Next State	Out
S	L	S <sup>+</sup>	P
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0



# 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

## *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

## *Binary Encoding*

- Straight encoding of states  
 $S_0 = "00"$   $S_1 = "01"$   $S_2 = "10"$   $S_3 = "11"$
- 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 bits switches at a time

$$S_0 = "00" \quad S_1 = "01" \quad S_2 = "11" \quad S_3 = "10"$$

- 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

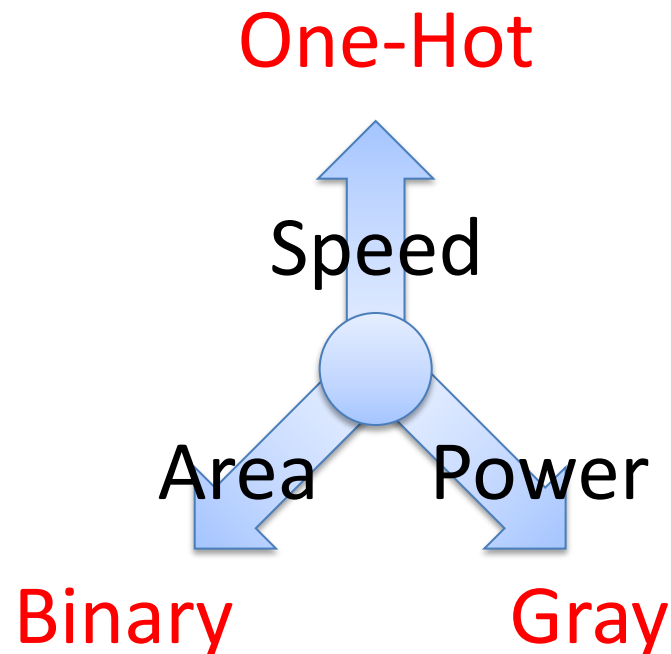
## *One-Hot Encoding*

- Encoding one flip-flop for each state  
     $S_0 = "0001"$     $S_1 = "0010"$     $S_2 = "0100"$     $S_3 = "1000"$
- 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

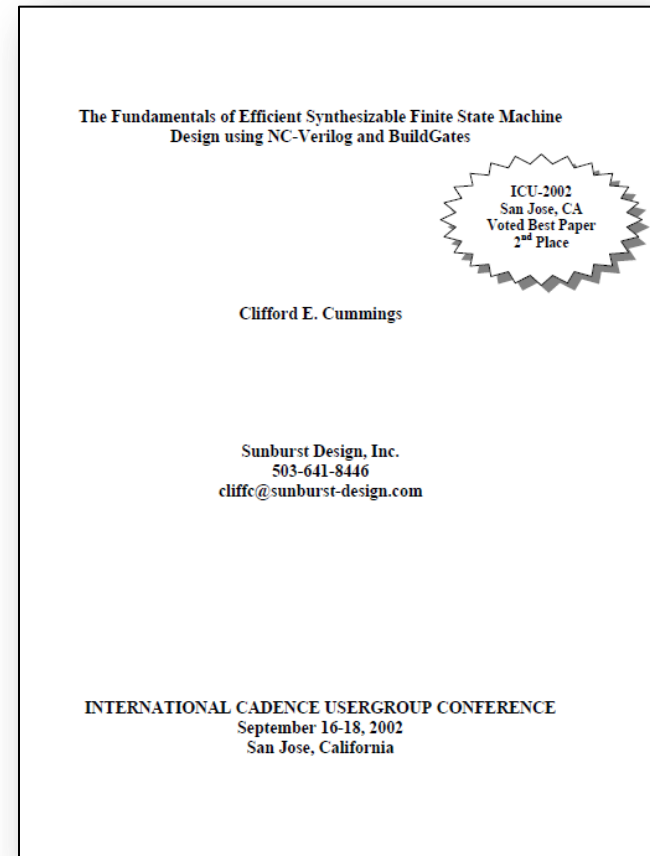


# HDL FSM Implementation

The Fundamentals of Efficient Synthesizable  
Finite State Machine.

Clifford E. Cummings

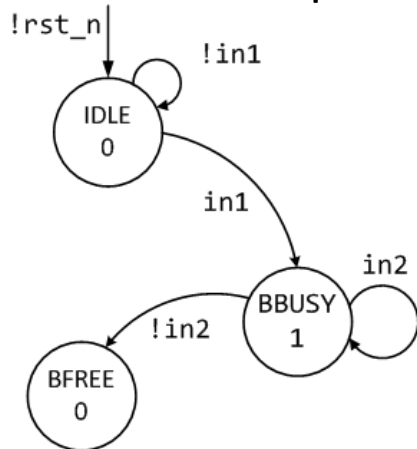
[http://www.sunburst-  
design.com/papers/CummingsICU2  
002\\_FSMFundamentals.pdf](http://www.sunburst-design.com/papers/CummingsICU2002_FSMFundamentals.pdf)



# 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. The sequential always block is coded using nonblocking assignments.
2. 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.
3. Assignments within the combinational always block are made using Verilog blocking assignments.

```
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;           ← 1
else state <= next;

always @(state or in1 or in2) begin ← 2
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                                  ← 3
```

# HDL FSM Implementation

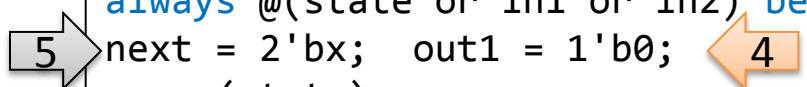
## *Two Always Block FSM Style (Good Style)*

4. 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)
5. 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
  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
```

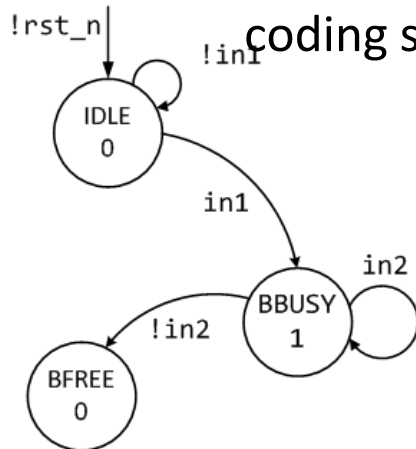


# HDL FSM Implementation

## *One Always Block FSM Style (Avoid This Style!)*

- One of the most common FSM coding styles in use today
  - It is more verbose
  - It is more confusing
  - It is more error prone

(comparable two always block coding style)



```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01,BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
state <= IDLE;
out1 <= 1'b0;
end
else begin
state <= 2'bx; out1 <= 1'b0;
case (state)
IDLE : if (in1) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= IDLE;
BBUSY: if (in2) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= BFREE;
endcase
end
```

# HDL FSM Implementation

## *One Always Block FSM Style (Avoid This Style!)*

1. A declaration is made for state. Not for next.
2. The state assignments do not correspond to the current state of the case statement, but the state that case statement is transitioning to. This is **error prone** (but it does work if coded correctly).

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
state <= IDLE;
out1 <= 1'b0;
end
else begin
state <= 2'bx; out1 <= 1'b0;
case (state)
IDLE : if (in1) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= IDLE;
BBUSY: if (in2) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= BFREE;
endcase
end
```

# HDL FSM Implementation

## *One Always Block FSM Style (Avoid This Style!)*

3. There is just one sequential always block, coded using nonblocking assignments.
4. All outputs will be registered (unless the outputs are placed into a separate combinational always block or assigned using continuous assignments).  
No asynchronous Mealy outputs can be generated from a single synchronous always block.

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
state <= IDLE;
out1 <= 1'b0;
end
else begin
state <= 2'bx; out1 <= 1'b0;
case (state)
IDLE : if (in1) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= IDLE;
BBUSY: if (in2) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= BFREE;
endcase
end
```

34

# HDL FSM Implementation

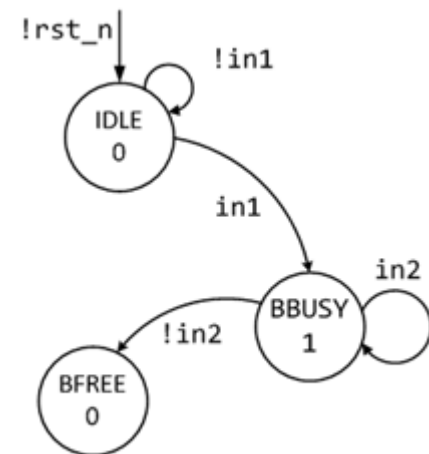
## *Onehot FSM Coding Style (Good Style)*

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

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= 3'b001;
else      state <= next;

always @(state or in1 or in2) begin
next = 3'b000;  out1 = 1'b0;
case (1'b1) //ambit synthesis case full, parallel
state[IDLE] : if (in1) next[BBUSY] = 1'b1;
              else      next[IDLE]  = 1'b1;
state[BBUSY]: begin
out1 = 1'b1;
if (in2) next[BBUSY] = 1'b1;
else      next[BFREE] = 1'b1;
end
//...
endcase
end
```

Efficient (small and fast) Onehot state machines can be coded using an inverse case statement



# HDL FSM Implementation

## *Onehot FSM Coding Style (Good Style)*

1. Index into the state register, not state encodings
2. Onehot requires larger declarations
3. State reset, set to 1 the IDLE bit
4. Next state assignment must make all-0's

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

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= 3'b001;
else state <= next;

always @(state or in1 or in2) begin
next = 3'b000; out1 = 1'b0;
case (1'b1) //ambit synthesis case full, parallel
state[IDLE] : if (in1) next[BBUSY] = 1'b1;
else next[IDLE] = 1'b1;
state[BBUSY]: begin
out1 = 1'b1;
if (in2) next[BBUSY] = 1'b1;
else next[BFREE] = 1'b1;
end
//...
endcase
end
```

# HDL FSM Implementation

## *Onehot FSM Coding Style (Good Style)*

1. Inverse case statement usage
2. Case branch check state values
3. Added “full” and parallel case pragmas
4. Only the next state bit

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

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= 3'b001;
else      state <= next;

always @(state or in1 or in2) begin
next = 3'b000; out1 = 1'b0;
case (1'b1) //ambit synthesis case full, parallel
state[IDLE] : if (in1) next[BBUSY] = 1'b1;
               else      next[IDLE]  = 1'b1;
state[BBUSY]: begin
               out1 = 1'b1;
               if (in2) next[BBUSY] = 1'b1;
               else      next[BFREE] = 1'b1;
               end
//...
endcase
end
```





# Full-Parallel Case

From:

Verilog: Frequently Asked Questions: Language, Applications and Extensions

Shivakumar S. Chonnad,  
Needamangalam B.  
Balachander

ISBN: 1441919864

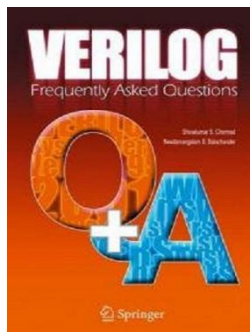


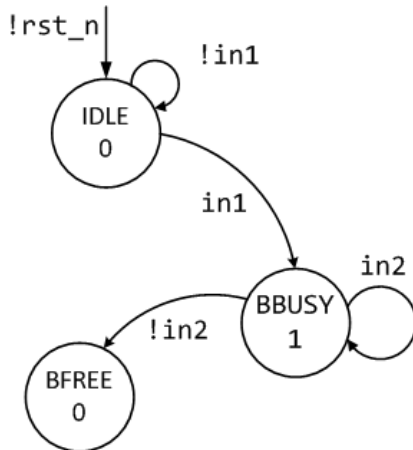
Table 2-5. Difference between full case and parallel case

full_case	parallel_case
Indicates that the case statement has been fully specified, and all unspecified case expressions can be optimized away	Indicates that all case items need to be evaluated in parallel and not infer any priority encoding logic
All control paths are specified explicitly or by using a default	There is no overlap among the case items
Helps avoid latches as all cases are fully specified	Results in multiplexor logic as a parallel logic
Although not recommended, the default clause can be avoided, and still not infer a latch	A priority encoder is NOT synthesized, as each path is unique
An example of a case statement that is full (and parallel) is shown below:  <pre>reg var1 [1:0]; always @(a or b or c) begin   case (var1)     2'b00 : out1 = a;     2'b01 : out1 = b;     2'b10 : out1 = c;     2'b11 : out1 = a&amp;b;   endcase end</pre>	An example of a case statement that is parallel (not full) is shown as follows:  <pre>reg var1 [2:0]; always @(a or b or c) begin   case (var1)     3'b000 : out1 = a;     3'b001 : out1 = b;     3'b010 : out1 = c;     // rest of the cases are //     not defined   endcase end</pre>
Note that the <b>default</b> clause was not required here as it is fully specified (although having it is a good coding practice).	Note that the above case <b>doesn't</b> have a <b>default</b> clause; but each branch is definitely <b>unique</b> , but all cases are not specified, that is, branches missing for 2,3,4,5,6,7. The out1 register <b>will</b> get synthesized into a latch

# HDL FSM Implementation

## *Registered FSM Outputs (Good Style)*

- Registering the outputs of an FSM design
  - Insures that the outputs are glitch-free and frequently
  - Improves synthesis results by standardizing the output and input delay constraints of synthesized modules



```
output reg out1;
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;
case (state)
IDLE : if (in1) next = BBUSY;
else next = IDLE;
BBUSY: if (in2) next = BBUSY;
else next = BFREE;
endcase
end
always @(posedge clk or negedge rst_n)
if (!rst_n) out1 <= 1'b0;
else begin
out1 <= 1'b0;
case (next)
IDLE, BFREE: ; // default outputs
BBUSY, BWAIT: out1 <= 1'b1;
endcase
end
```

# REFERENCES

# References

- Verilog HDL – Samir Palnitkar
- The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and Build Gates – Clifford E. Cummings
- Verilog: Frequently Asked Questions, Springer Science

# References

- Verilog: Frequently Asked Questions: Language, Applications and Extensions - Shivakumar S. Chonnad, Needamangalam B. Balachander
- ECE 369 - Fundamentals of Computer Architecture.  
[http://www2.engr.arizona.edu/~ece372\\_spr04/ece369\\_spr05](http://www2.engr.arizona.edu/~ece372_spr04/ece369_spr05)
- Madhavan, R., Quick Reference for Verilog HDL  
<http://www.stanford.edu/class/ee183/handouts.shtml>