



# **Design using Verilog HDL**

**Kang Yi**  
**School of Computer Science &  
Electronic Engineering**  
**Handong Global University**

# References

- Verilog HDL : A Guide to Digital Design and Synthesis
  - Author : Samir Palnikar
  - Publisher : PTR-PH
- HDL Chip Design
  - Author : Douglas J. Smith
  - Publisher : Doone Publications
- Verilog Center
  - <http://www.angelfire.com/in/rajesh52/verilog.html>
  - Online Books, Technical Papers on Design Tips in Verilog
- Online Quick Reference
  - [http://www.sutherland-hdl.com/on-line\\_ref\\_guide/vlog\\_ref\\_top.html](http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.html)

# Agenda

- Basics
- Modules & Ports
- Verilog Simulation
- Gate level modeling
- dataflow modeling
- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis

# Basics

# Components of Module

- Corresponds to a circuit component

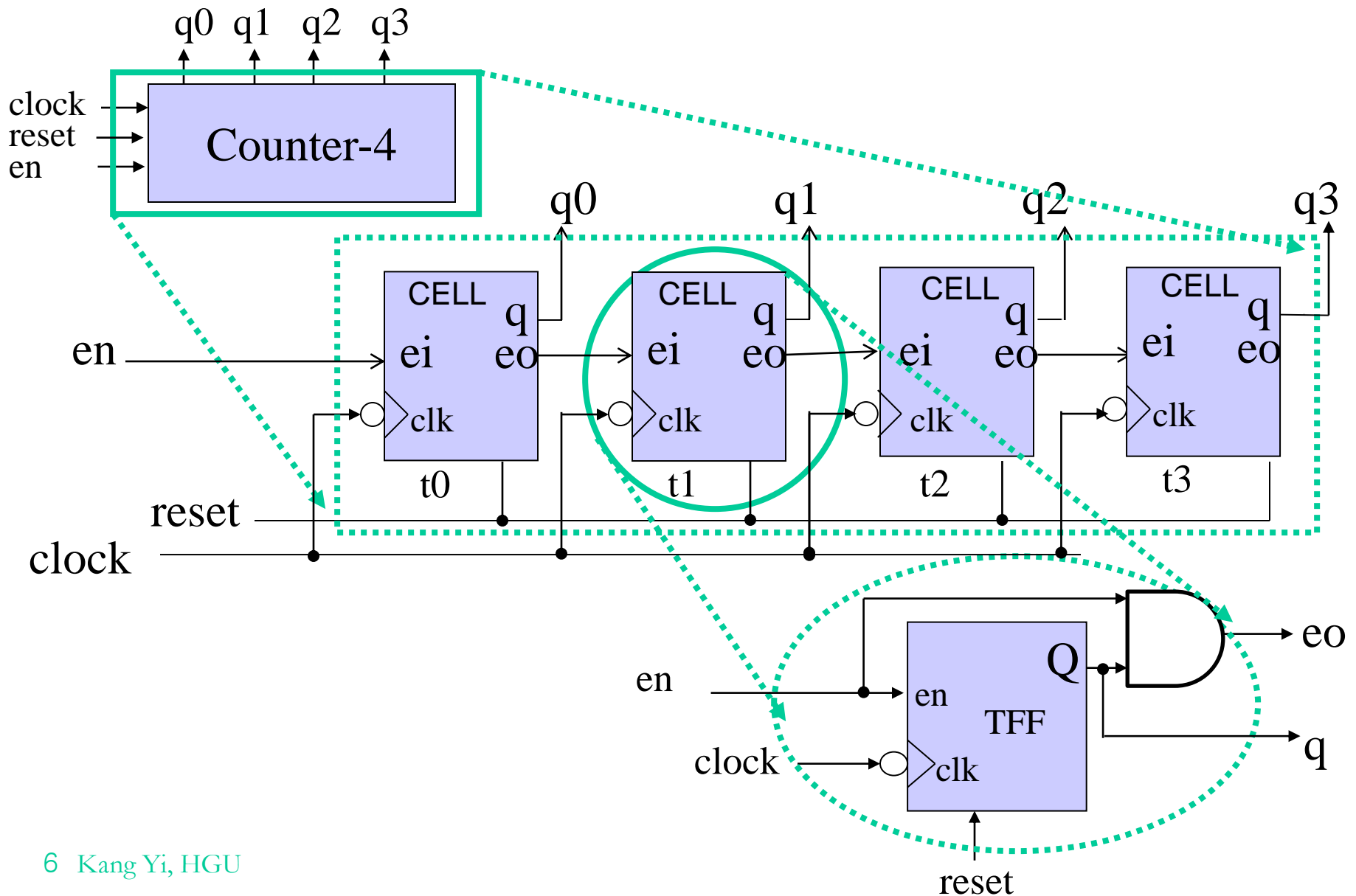
```
module module_name (port_list);  
    module port_declarations and  
    internal_module_descriptions  
endmodule
```

The diagram illustrates the components of a Verilog module declaration. Arrows point from labels to specific parts of the code:

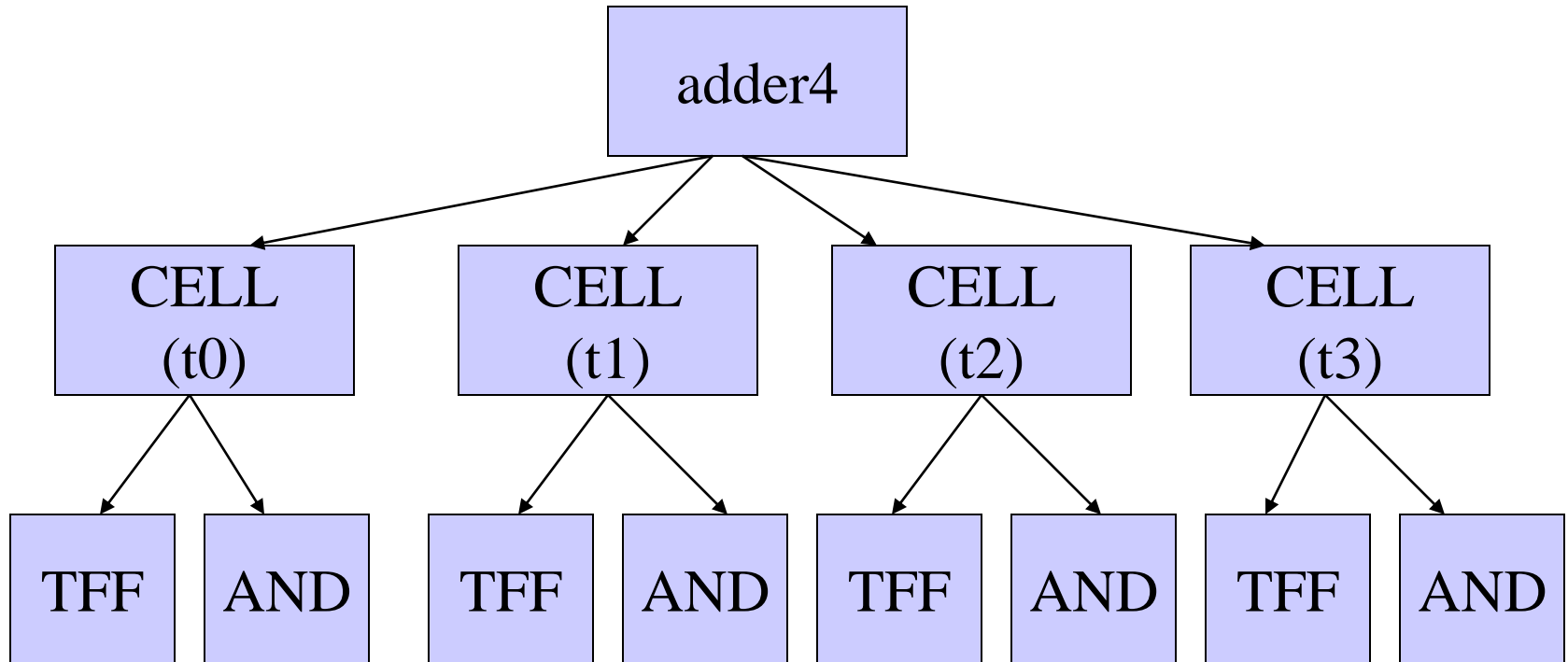
- module name** points to `full_addr`.
- ports** points to the list of ports in parentheses: `(A, B, Cin, S, Cout)`.
- inputs/outputs** points to the `input` and `output` declarations.

```
module full_addr (A, B, Cin, S, Cout);  
    input      A, B, Cin;  
    output    S, Cout;  
  
    assign {Cout, S} = A + B + Cin;  
endmodule
```

# 4-bit counter design example



# Hierarchical view



- Here, each building block is a Verilog **module**  
(in VHDL : *entity declaration + architecture body*)

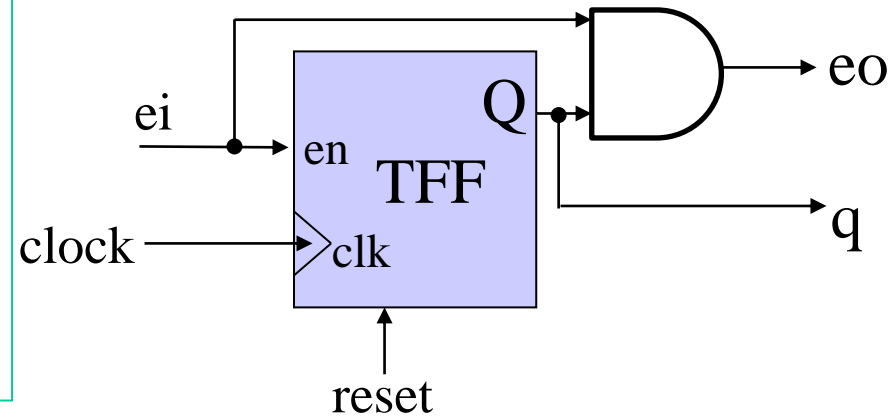
# Example (top-level)

```
module counter_4 (q,clk,reset, en);  
  output [3:0] q;  
  input clk, reset, en;  
  CELL t0 (q[0], clk, reset, en);  
  CELL t1 (q[1], clk, reset, q[0]);  
  CELL t2 (q[2], clk, reset, q[1]);  
  CELL t3 (q[3], clk, reset, q[2]);  
endmodule
```



# Example (Submodule)

```
module CELL (q, clk, reset, ei, eo);  
  output q, eo;  
  input clk, reset, ei;  
  wire d; // internal connection  
  TFF tff0 (q, clk, reset, ei);  
  and a0 (eo, d, ei); // primitive gate  
endmodule
```



```
module TFF (q, clk, reset, en);  
  output q;  
  input d, clk, reset, en;  
  reg q;  
  always @(posedge reset or negedge clk)  
    if(reset)  
      q <= 1'b0;  
    else if (en)  
      q <= ~q;  
  assign eo = ei && q;  
endmodule
```

# Lexical conventions

- Comments : `//` or `/* .... */`
- Identifiers
  - Alphanumeric, underscore(`_`), and \$ sign
  - Should start with an alphanumeric or `_`
  - case sensitive

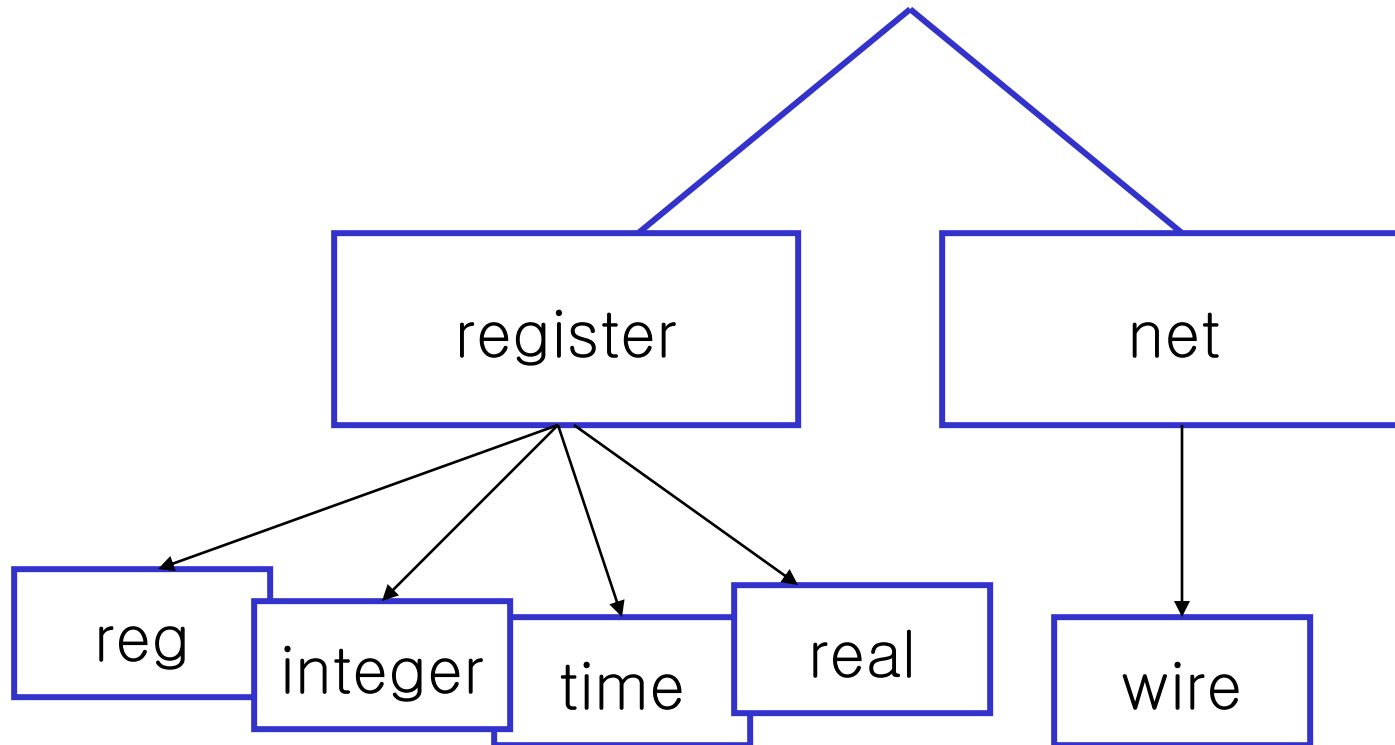
# Verilog Numbers

- Sized number : `<bit#>'<base format> <number>`
  - `4'b1111`    `12'habc`    `16'd255`
  - size : number of bits
- Unsized number : `'<base format> <number>` or `<number>`
  - `23456` // default is decimal (32-bit : machine dependent)
  - `'hc3` // 32-bit hexadecimal
  - `'o21` // 32-bit octal
- Negative numbers : `-3`, `-6'd3` (2's complement representation of -3)
- Underscore character is ignored in a number except for the first character : `12'b1111_0000_1010`
- Quotation marks `"?"` means `"z"` in the context of number : `4'b10??`

# Verilog Data Values

- Value set
  - 0 : logic zero
  - 1 : logic one
  - X : unknown
  - Z : high impedance
- Strength level
  - Supply
  - Strong
  - Pull
  - Large (triereg type only)
  - Weak
  - Medium (triereg type only)
  - Samll (triereg type only)
  - highz

# Verilog Variables



# Nets vs Registers

- Nets
  - Connection between hardware elements
  - Continuously driven by outputs of device type are connected
  - Default value of net is z
  - Declared with keyword **wire, trireg, tri, wand, wor, triand, trior, etc**
  - **ex) wire a;**
- Registers
  - Retain data value until another value is placed on them
  - (a variable that can hold value)
  - Declared with keyword **reg, integer, real, time**
  - Do not confuse reg with hardware registers (FF) in real circuits (reg type is usually corresponds to a wire in the circuit)
  - **ex) reg reset;**  
    initial begin  
        reset = 1'b1;  
        #100 reset = 1'b0;  
    end

# Nets and Registers (example)

```
module design (a,x,b,c);  
input a,x;  
output b,c;  
reg b;  
assign c = x & a;  
always @(a or x)  
    b = a | x;  
endmodule
```

# Integer, Real

- Integer
  - Register type
  - Default bit width is machine-dependent (at least 32)
  - Ex) Integer counter;  
Initial  
counter = -1;
- Real
  - Register type
  - Decimal or scientific notation
  - Ex) real delta;  
initial begin  
delta = 4e10;  
delta = 2.13;  
end



# Time

- Register data type
- Simulation time
- The width is implementation-dependent (at least 64)
- \$time system function returns current simulation time
- Ex) time save\_sim\_time;  
initial  
save\_sim\_time = \$time;
- Simulation time is measured in terms of simulation seconds (denoted by s)
- Time scale (relation between real time and simulation time is defined by user)

# Vectors : multiple-bit data

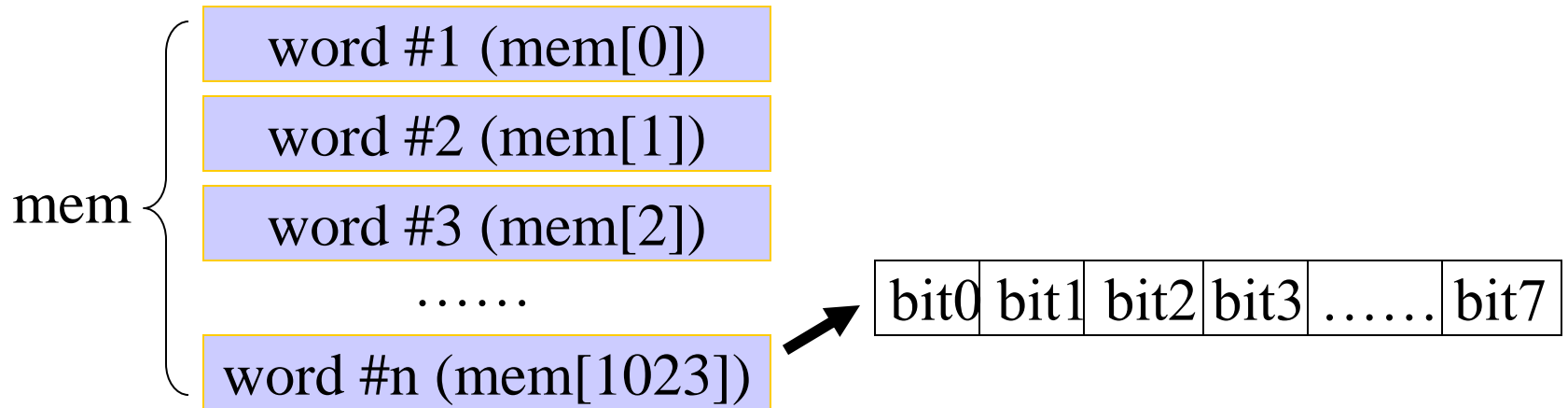
- Nets or reg data type can be declared as multiple bit width using vector
- Data-type [MSB# LSB#] signal-name
  - Left number is MSB and right number is LSB
- Ex) wire [7:0] bus;  
    wire [31:0] busA, busB; // 31 is MSB  
    reg [0:40] addr; // 0 is MSB
- Parts of vector bits can be addressed as :
  - busA[7]
  - busB[2:0]
  - addr[0:1]

# Arrays

- allowed for reg, integer, and time, and vector register data type (not allowed for real variable)
- <array\_name>[<subscript>]
- multi-dimensional array is not permitted
- ex) integer counter [3:0];  
    reg bool [31:0];  
    time chk\_point[1:100];  
    reg [4:0] port\_id[0:7]; // array of 8port\_ids ; each port\_id is 5 bits width
- array elements are addressed as : count[5], chk\_point[0], port\_id[3]
- Notice !
  - **vector** is a **single element** with n-bits wide
  - **arrays** are **multiple elements** that are 1-bit (default) or n-bits wide

# Memories

- RAM, ROM, or register files are modeled as array of registers
- Each element of the array is a word (each word can be multiple-bits)
- ex) `reg [7:0] mem [0:1023] ; // 1K 8bit words`



# Strings

- Array of bytes
- Can be stored in reg

- Example)

Reg [8\*18 : 1] string\_val ; // 18 bytes wide

Initial

string\_val = "Hello Verilog World"; // store string value

- width of reg > the size of string => fills bits to the left with 0s
- width of reg < the size of string => truncates the leftmost bits of string

# Parameters

- Constant definition
- The value can be overridden at compile time by defparam statement

- Example)

```
parameter port_id = 5; // define a constant port_id
parameter cache_width = 256 ; // define a constant cache_width
reg [cache_width - 1 : 0] cache;
dest = port_id ;
```

# Basic Compiler Directives

- Macro definition
  - Syntax : ``define macro_name macro_value`
  - Similar to `#define` in C
  - Ex) ``define WORD_SIZE 32`  
``define WORD_REG reg [32:0]`  
`reg [`WORD_SIZE-1 : 0] line ;`  
``WORD_REG line_var ;`
- File inclusion
  - Syntax : ``include filename`
  - Similar to `#include "filename"` in C
  - Ex) ``include header.v`

# Exercises

1. What are the types of one bit signal value in Verilog ?
2. Which of the following types are net type ?
  1. reg
  2. wire
  3. integer
  4. real
  5. Time
3. Define Constants Phi (=3.14159) and Delay (=10) by means of parameter and define respectively.
4. Find the variable definitions for “a” and “b” .  
reg [7:0] M [0:15];  
a = M[0];  
b = a[0];



# Modules and Ports

# Components of Verilog Module

**module** *modulename* ;

Port list, port declarations (optional)

Parameters (optional)

Declarations of wire,  
reg ,and other variables

Dataflow statements  
(assign statements)

Instantiation of  
lower level modules

Behavioral statements  
(Always and initial blocks)

Tasks and functions

**endmodule**

# Concurrency

- Following Verilog HDL constructs are independent processes that are evaluated concurrently in simulation time :
  - Module Instances
  - Primitive Instances
  - Continuous Assignments
  - Procedural Blocks

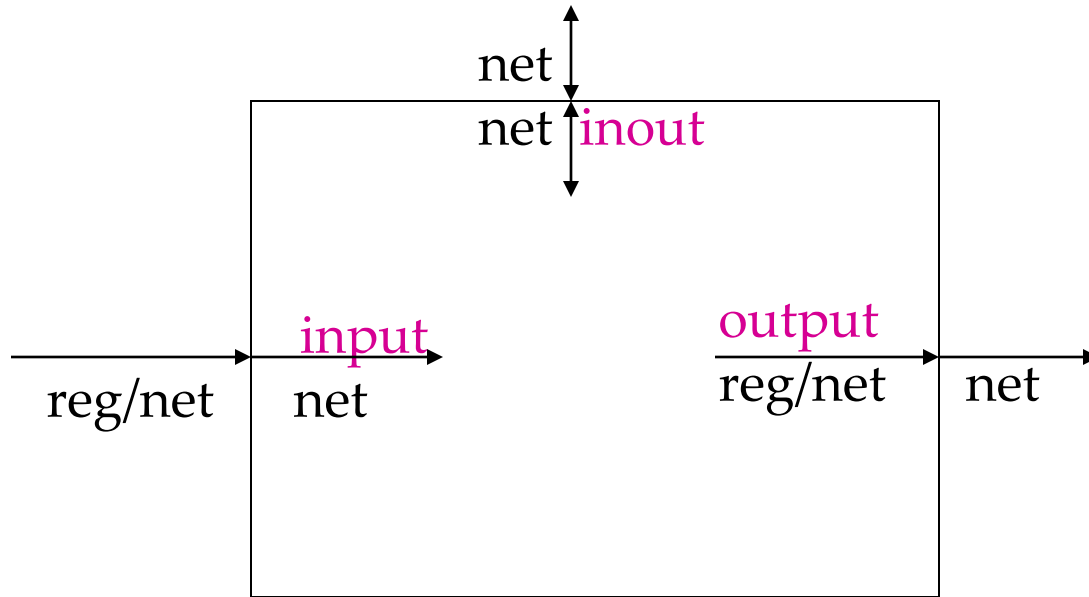
# Ports of Module

- Functions : Provides interface to communicate with its environments
- Port declaration :
  - Each ports has mode (direction) : input, output, inout
  - All ports are implicitly declared as **wire**.
  - So, If a output port hold its value, it must be declared as **reg**

```
module DFF(q,d,clk,reset) ;  
output q;  
input d, clk, reset;  
reg q; // output port q holds value  
always @(posedge reset or negedge clk)  
    if (reset) q = 1'b0;  
    else q = d;  
endmodule
```

- Port of type in or inout cannot be declared as **reg**

# Ports Connection Rules



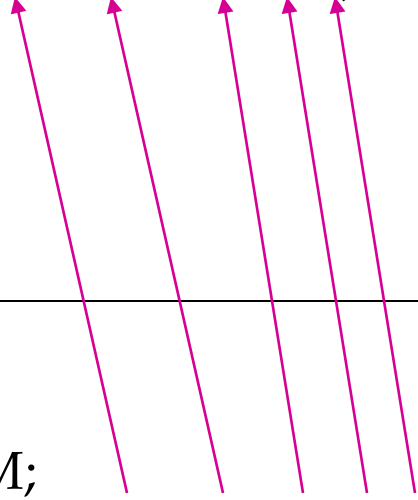
- Inout ports should be net type
- Input ports should be net type
- Output port can be either net or reg type

# Connecting methods between signals and instance ports

```
fulladder4 (sum, cout, a,b,c);  
...  
endmodule
```

- By order

```
module top;  
  reg A,B, C_IN;  
  wire C_O, SUM;  
  fulladder4 U0 (SUM, C_O, A,B,C_IN);  
  ...  
endmodule
```



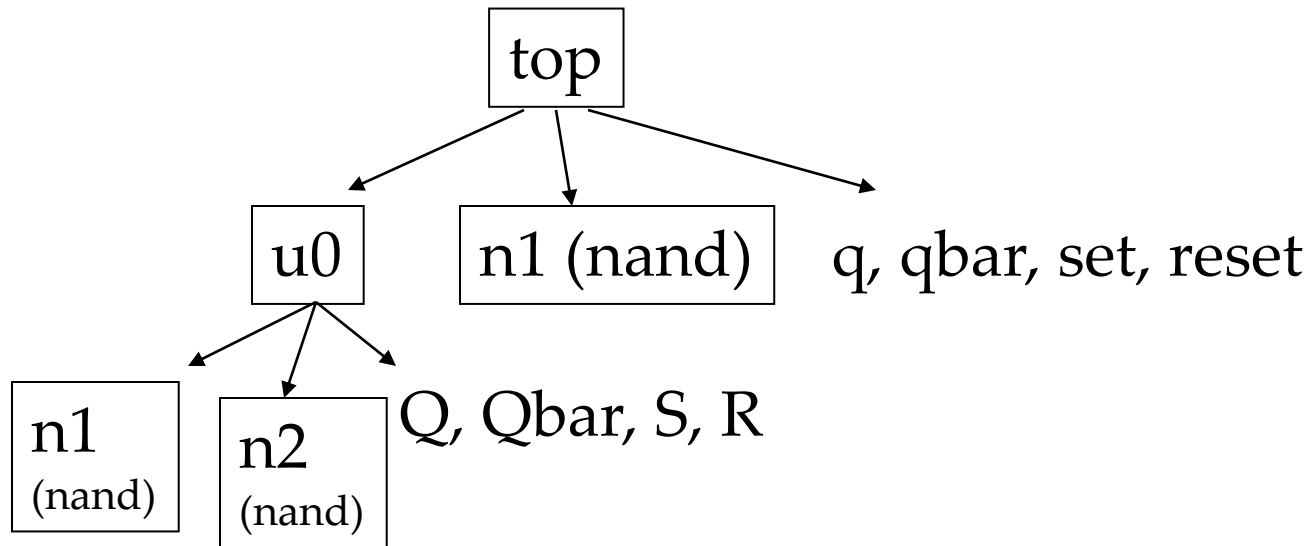
- By name

```
module top ;  
  fulladder4 U0 (.cout(C_O), .sum(SUM), .a(A), .b(B), .c(C_IN));  
  ...  
endmodule
```

# More on Ports

- Width matching
  - **Legal** to connect signals with different width
  - But typically issued **Warning**
- Unconnected ports
  - Ex) `fulladd4 U0 (SUM, , A, B, C_IN);` // `carry_out` is disconnected

# Hierarchical names

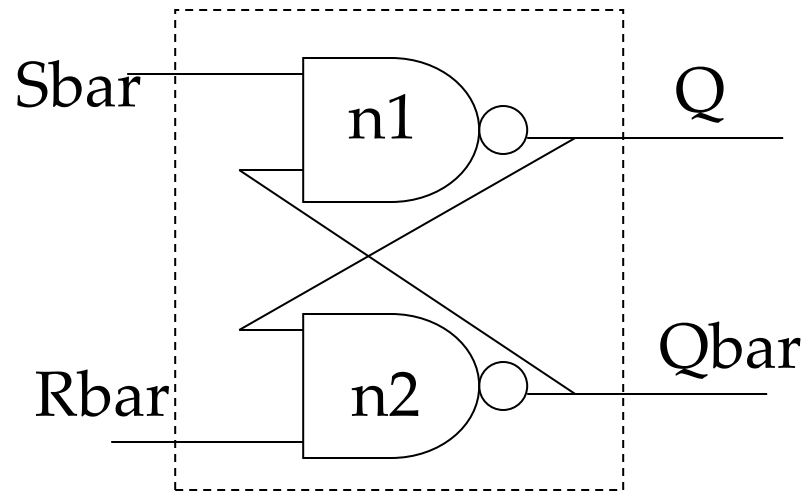


- Use dot(.) for each level of hierarchical name
- Module\_name.instance\_name.port\_name
- Ex) top.q    top.u0.Q    top.n1



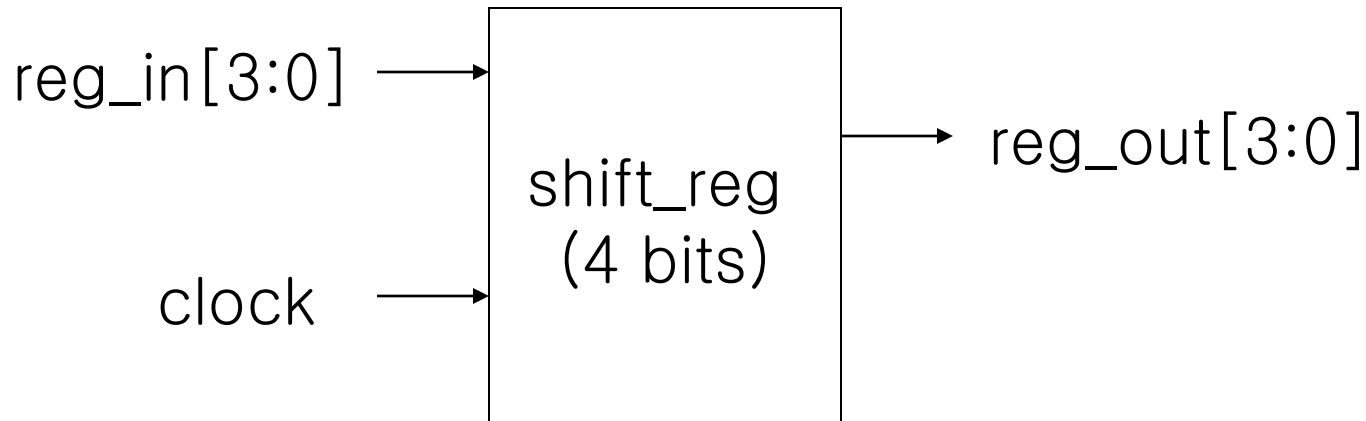
# Circuit Example

```
module SR_latch (Q, Qbar, Sbar,Rbar) ; // SR latch module
output Q, Qbar;
input Sbar, Rbar ;
nand n1 (Q, Sbar, Qbar);
nand n2 (Qbar, Rbar, Q);
endmodule
```



# Exercise

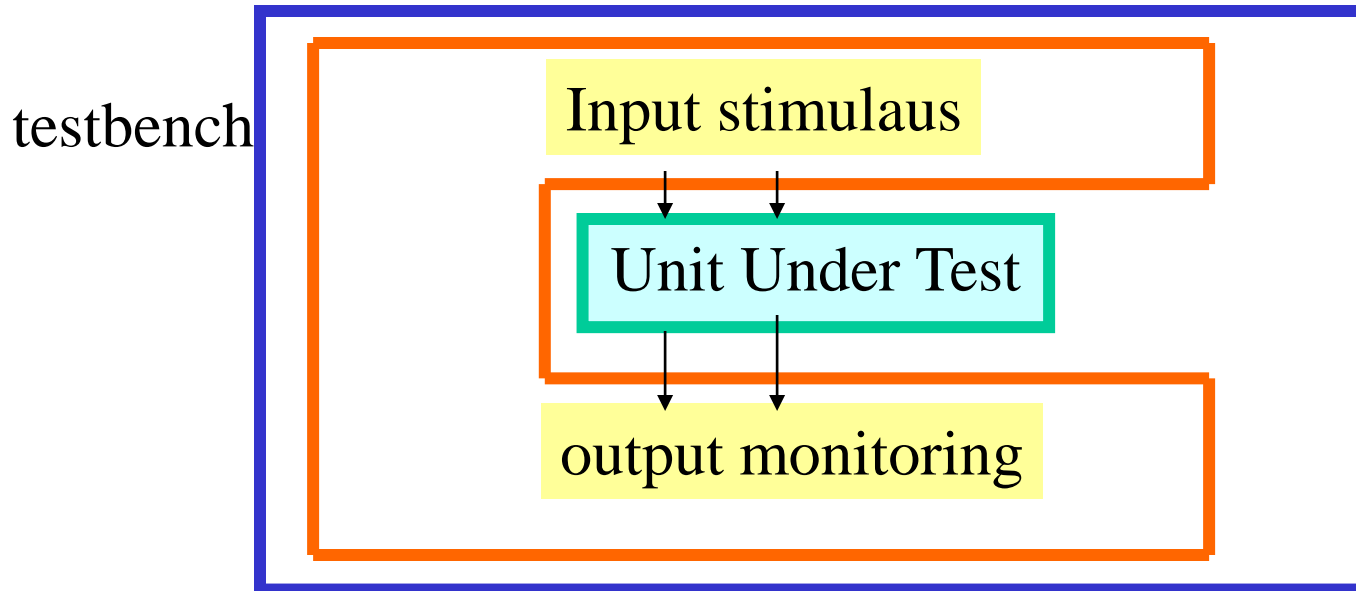
1. What are the basic components of module ? And which are mandatory ?
2. Write a Verilog code of instantiation of a module without ports.
3. Write down the port declaration for the module of the shift\_reg module.



# Verilog Simulation

- Testbench
- Simulation System Tasks
- Timescale

# Testbench and Verilog Simulation



- Testbench
  - is written in verilog
  - is a module for simulation only (not synthesized)
  - Component instantiation +  
input waveform +  
output monitoring statements

# Verilog Testbench Frame

```
module testbench ; // module without IO port
```

```
wire out1, out2;  
reg in1, in2; // Internal signal declaration
```

```
myckt uut ( out1, out2, in1, in2); // module instantiation under test
```

```
// stimulus via input signals
```

```
initial begin
```

```
    in1 = 1'b0; in2 = 1'b0;
```

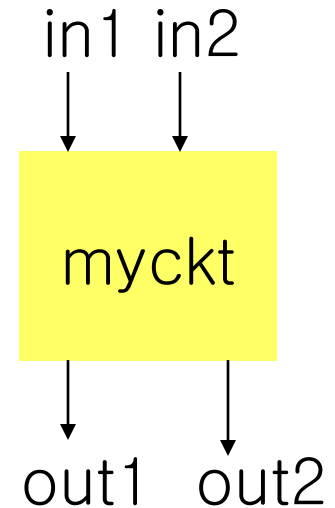
```
    #8 in1 = 1'b1; #8 in2 = 1'b1;
```

```
end
```

```
initial // output signal monitor
```

```
    $monitor("sigA = %d sigB=%d", out1, out2);
```

```
endmodule
```



# Simulation Example

```
module testbench; // simulation module
```

```
wire q, qbar;
```

```
reg set, reset;
```

```
SR_latch UUT (q, qbar, ~set, ~ reset); // lower level module instantiation
```

```
initial begin // stimulus input and monitor outputs
```

```
    set = 0; reset = 0;
```

```
    #5 reset = 1;
```

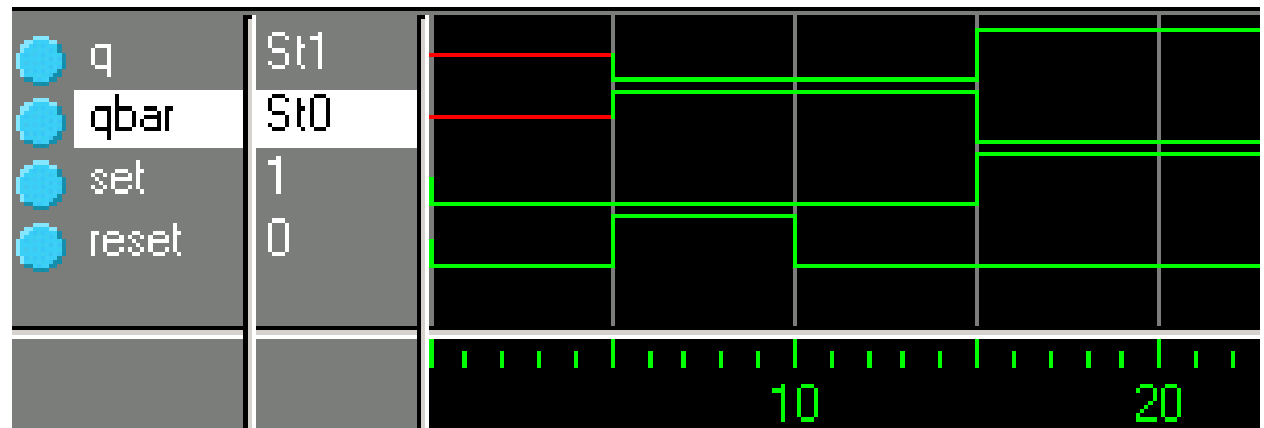
```
    #5 reset = 0;
```

```
    #5 set = 1;
```

```
end
```

```
endmodule
```

The result waveform



# Basic System Tasks for Simulation

- \$display : print text message on the screen once
- \$strobe : similar to \$display except that the printing of text is delay until all events in the current time step have been executed
- \$monitor : print text message on the screen whenever its argument values are changed
- \$stop : suspend simulation
- \$finish : terminate simulation

# \$display

- `$display (p1,p2, ... pn)`
- Similar to `printf` in C Language
- Automatically insert newline
- String format specifier
  - `%d` or `%D` : in decimal
  - `%b` or `%B` : in binary
  - `%h` or `%H` : in Hexadecimal
  - `%o` or `%O` : in Octal
  - `%s` or `%S` : string
  - `%m` or `%M` : hierarchical name of the module (no argument required)
  - `%f` or `%F` : real number in decimal

- Example)

```
$display ("Hello Verilog");
```

```
$display ($time);
```

```
$display("At time %d virtual address is %h", $time, v_addr);
```

```
$display("This is from %m level of hierarchy");
```



# \$monitor

- Usage : `$monitor (p1, p2, ... pn);`
- Unlike `$display` `$monitor` needs to be invoked only once
  - Only one monitoring list can be active at a time.
  - If there is more than one `$monitor`, the last `$monitor` is effective
- Example)  
`initial $monitor($time, “clock =%b reset = %b”,clock, reset);`
- `$monitoron` : a system task enabling monitoring
- `$monitroff` : a system task disabling monitoring
- Monitoring is turned on by default at the beginning of simulation

# \$stop / \$finish

- \$stop
  - Usage : %stop ;
  - Puts the Simulator into Interactive Mode
  - Allows Designer to Debug the Design in the interactive mode
- \$finish
  - Usage : \$finish
  - Terminates the Simulation

- Examples)

```
initial begin
```

```
clock = 0; reset = 1;
```

```
#100 $stop // this will suspend the simulation at time = 100
```

```
#900 $finish // this will terminate the simulation at time = 900
```

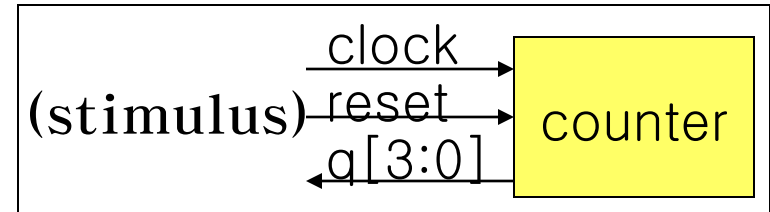
```
end
```

# Testbench Example with System Tasks

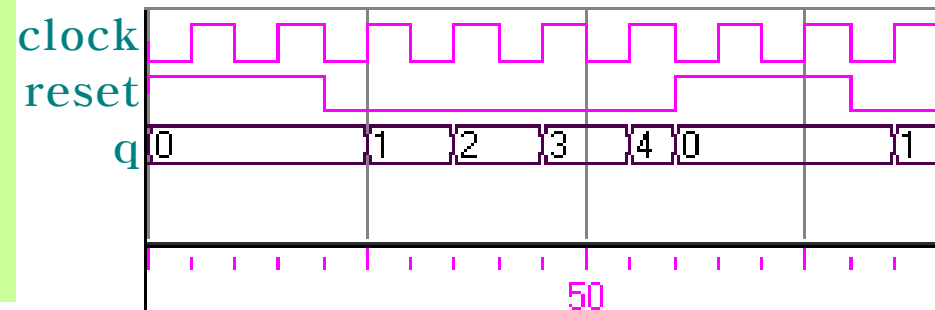
```
module stimulus ;
reg clock, reset;
wire [3:0] q;

bin_counter r1 (q, clock, reset);

// clock period :10
always #5 clock = ~ clock ;
initial begin
    clock = 1'b0;
    reset = 1'b1;
    #20 reset = 1'b0;
    #40 reset = 1'b1;
    #20 reset = 1'b0;
    #10 $finish;
end
initial
    $monitor($time, " output = %d", q);
endmodule
```



0 output = 0  
25 output = 1  
35 output = 2  
45 output = 3  
55 output = 4  
60 output = 0  
85 output = 1



# Time scales

- The Unit of Delay values in a simulation are defined by a compiler directive ``timescale`
- Usage : ``timescale` <reference time unit> / <time\_precision>
- <time\_precision> specifies the precision to which the delays rounded off in simulator
- <reference\_time> and <time\_precision> are one of **1, 10, 100** with time unit (**ps, ns, us, ms**)
- Exmple)
  - ``timescale 100 ns / 1 ns`
  - ``timescale 1 us / 10 ns`

# 3 types of circuit description

- Structural : net-list description
  - Hierarchical design with submodule instantiation
  - list of components and how they are connected
  - just like schematics, but using text
- Dataflow : Boolean expression description
  - Assign statements
- Behavioral : algorithm description
  - describe *what* a component does, not *how* it does it
  - synthesized into a circuit that has this behavior
  - Always block
- Mixed description in a module is also possible

# Gate-level Modeling

- Primitive Gates
- Gate Delay

# Gate level Modeling

- Very intuitive, especially, for small circuit
- Verilog Language Provides *Primitive Logic Gates*
- *Structural description*
- Gate instantiation without instance name : legal
- Instance type (submodule name)
  - and, or, xor, xnor, nand, nor, buf, inv, bufif1, bufif0, invif1, invif0
  - Any number of input, output is possible with 1 type

# Gate Level Modeling (AND/OR type)

- Types : and, or, nand, nor, xor, xnor
- Ports order : Output, Input-1, Input-2, ... Input- $N$  for  $N$  input Gate
- Input/output ports number :
  - only 1 output ports,
  - any number of input ports ( $\geq 2$ )
- Examples)
  - and a1 (OUT, IN1, IN2); -- 2 input AND gate
  - and a2 (OUT, IN1, IN2, IN3); -- 3 input AND gate
  - and (OUT, IN1, IN2, IN3, IN4) ; -- legal gate instantiation

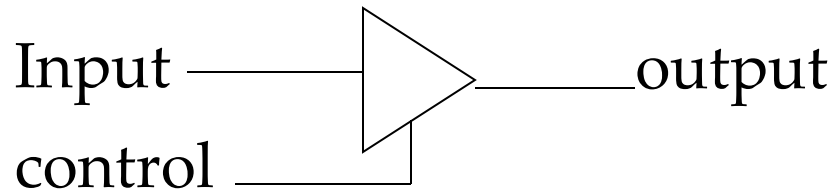


# Gate Level Modeling (buffer)

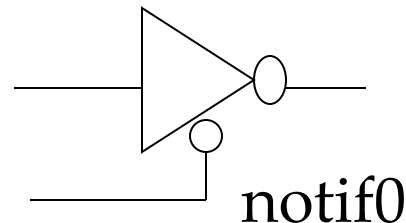
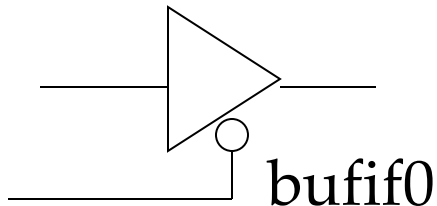
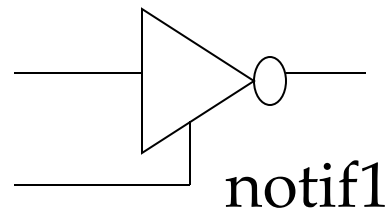
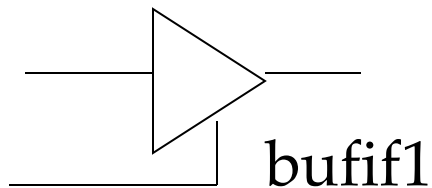
- Types : buf, not
- Ports order : Output1, [Output2, ...,] Input
- Number of IOs
  - any number of output ports ( $\geq 1$ )
  - Only 1 input port
- Examples)
  - buf b1 (OUT, IN); -- simple buffer
  - not n1 (OUT, IN) ; -- inverter
  - buf bf\_2out (OUT1, OUT2, IN) ; -- more than 2 outputs

# Gate level Modeling (tri-state-buf)

- Types : bufif1, notif1, bufif0, notif0

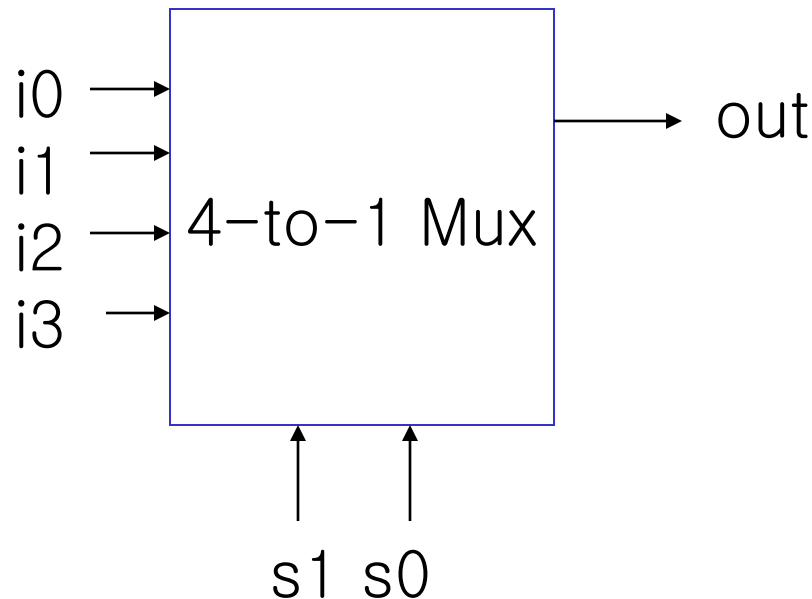


- Port order : output, input, control (Ex)bufif1 u0 (out, in, ctrl);



# example : 4-to-1 MUX

- 4-to-1 MUX example



# example : 4-to-1 MUX

```
module mux4_1 (out, i0,i1,i2,i3, s1,s0);
```

```
output out;
```

```
input i0, i1, i2, i3 ;
```

```
input s1, s0;
```

```
wire s1n, s0n;
```

```
wire y0, y1, y2, y3;
```

```
not (s1n, s1);
```

```
not(s0n, s0);
```

```
and (y0, i0, s1n, s0n);
```

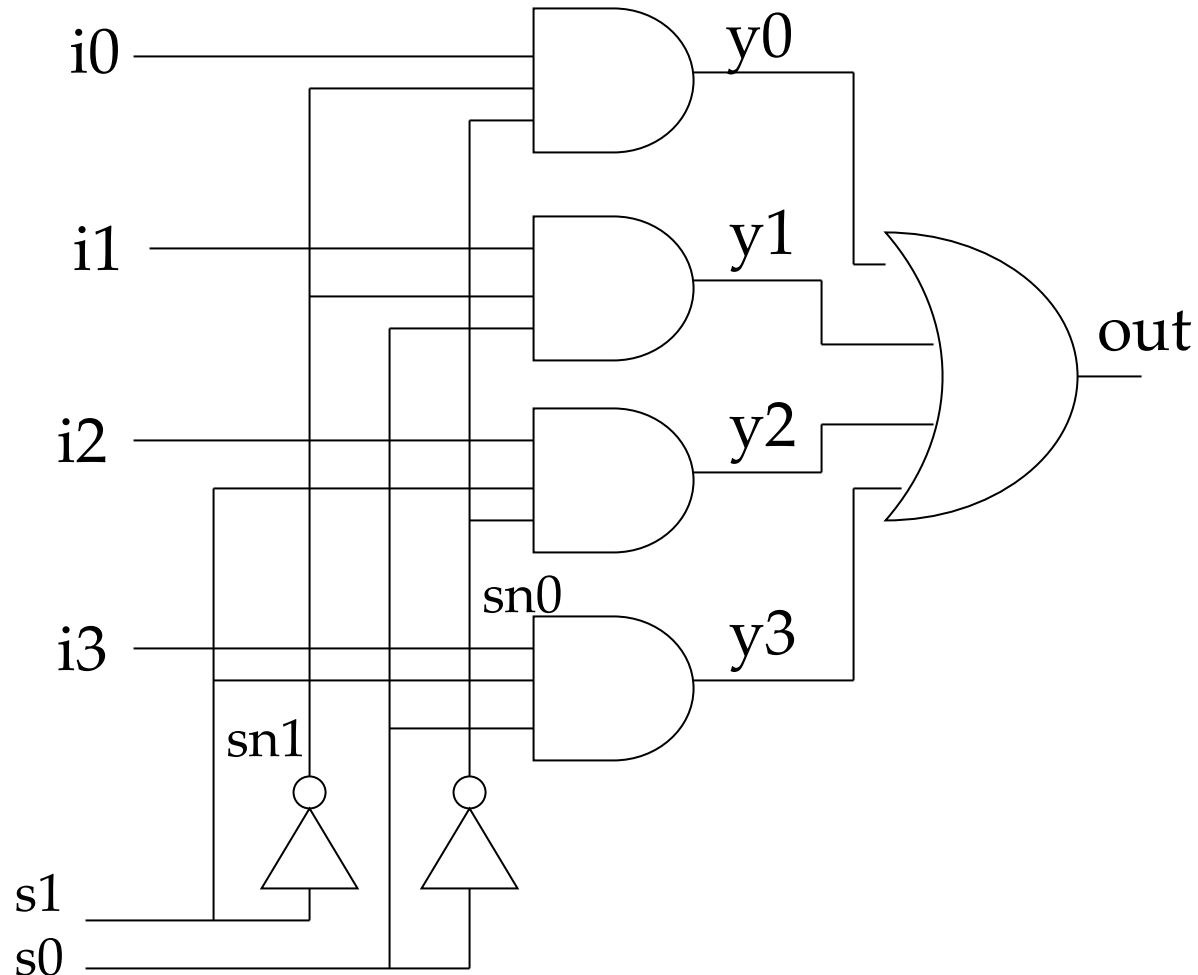
```
and (y1, i1, s1n, s0);
```

```
and (y2, i2, s1, s0n);
```

```
and (y3, i3, s1, s0);
```

```
or (out, y3, y2, y1, y0);
```

```
endmodule
```



# example [testbench for 4-to-1 MUX]

```
module testbench;
```

```
reg IN0, IN1, IN2, IN3;
```

```
reg S1, S0;
```

```
wire OUTPUT;
```

```
mux4_to_1 mymux (OUTPUT,IN0, IN1, IN2, IN3, S1, S0);
```

```
initial
```

```
begin
```

```
    IN0 = 1; IN1 = 0;
```

```
    IN2 = 1; IN3 = 0;
```

```
    S1 = 0; S0 = 0;
```

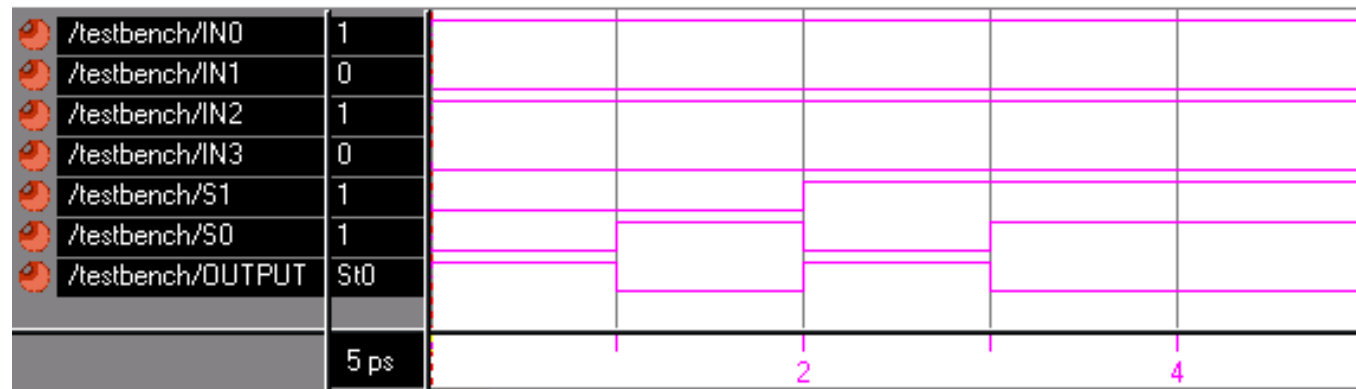
```
#1 S1 = 0; S0 = 1;
```

```
#1 S1 = 1; S0 = 0;
```

```
#1 S1 = 1; S0 = 1;
```

```
end
```

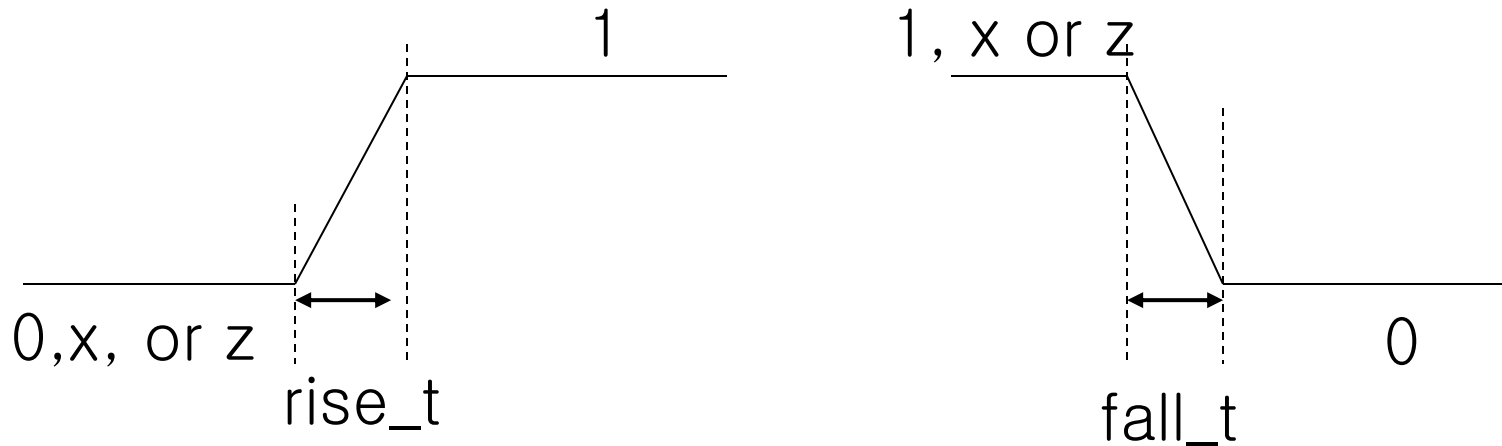
```
endmodule
```



# Exercise

1. Design 1-bit full-adder described as followed. Use only and, or and “not” primitive gates
  - $\text{sum} = A B \text{CIN} + A' B \text{CIN}' + A' B' \text{CIN} + A B' \text{CIN}'$
  - $\text{cout} = A B + B \text{CIN} + A \text{CIN}$
2. Write down a Verilog testbench for the 1-bit full adder above to verify by simulation

# Gate Delays: (rise, fall, turn-off)



- delay for all transition
  - ex) and **#(delay\_time) a1 (out, i1, i2);**
- rise and fall time delay specification
  - ex) and **#(rise\_t, fall\_t) a2 (out, i1, i2);**
- rise, fall, and turn off delay specification
  - **turn-off delay** is a transaction time to z from another value
  - ex) and **#(rise\_t, fall\_t, turn\_off\_t) a3 (out, i1, i2);**

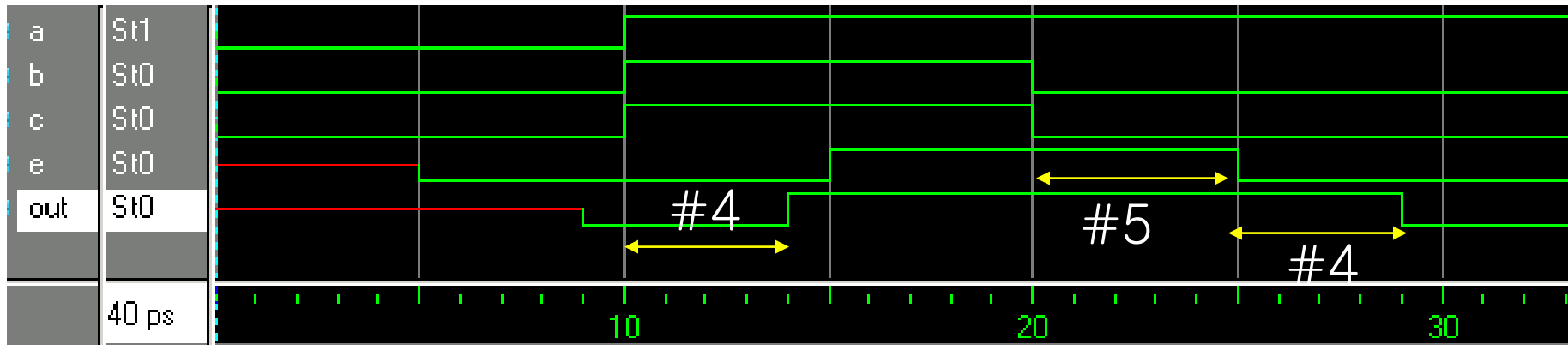
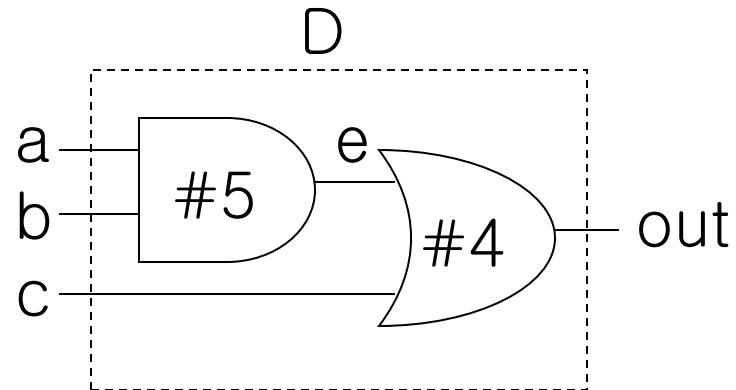
# Gate delays : Min/Typ/Max

- Model a device whose delays vary within a range because IC fabrication process variation
- for each of rise, fall, and turn-off time three values, **min**, **typ**, and **max**, can be specified
- Verilog simulators choose one of these values at run time
- Examples)
  - and # (4:5:6) a1 (out, i1, i2);
  - and #(3:4:5, 5:6:7) a2 (out, i1, i2);
  - and # (2:3:4, 3:4:5, 4:5:6) a3 (out, i1, i2);



# Delay Example

```
module D (out, a, b, c);  
  output out;  
  input a,b,c;  
  wire e;  
  and #5 a1 (e, a, b);  
  or #4 o1 (out, e, c);  
endmodule
```



# Dataflow Modeling

- Continuous Assignment
- Inertia Delay
- Verilog Operators

# Dataflow modeling

- Provides Circuit Description in terms of Data Flow between Registers and Processes on Data rather than Gate Instantiation
- Describes Circuits at a Higher (more abstract) Level of Abstraction than Gate Level Description
- Is Expressed by Continuous Assignments and Operators

# Continuous Assignments

- Are Used to Drive a value onto a net
- Starts with a Keyword “assign”
- Have Syntax :  
`assign <drive_strength> <delay> <list of assignments> ;`
- Have Following Characteristics
  - Left-hand-side of assignment signal *cannot* be a **register** type
  - Continuous assignments are *always active*
  - Right-hand-side operands of assignment *can be* **register** type or *function calls*. And they can be either scalars or vectors
- Examples)  
`assign out = i1 & i2;`  
`assign { c_out, sum[3:0] } = a[3:0] + b[3:0] + c_in; // concatenation of LHS`

# Implicit Continuous Assignments

- It Refers to the Declaration with Assignment Operator (=)
- Not Have the Keyword “assign”
- Examples

```
wire out = in1 & in2 ;
```

- The Above Implicit Continuous Assignment can be represented as follows

```
wire out;  
assign out = in1 & in2 ;
```

# Delay types and examples

- regular delays :

```
assign #10 out = in1 & in2 ; // Delay in a continuous assign
```

- Implicit Continuous Assignment delays :

```
wire #10 out = in1 & in2 ;
```

```
// Declaration + Delay + Implicit Continuous Assignment
```

- **Net Declaration Delay**

- Delay can be specified on a net when declared without a continuous assignment on the net

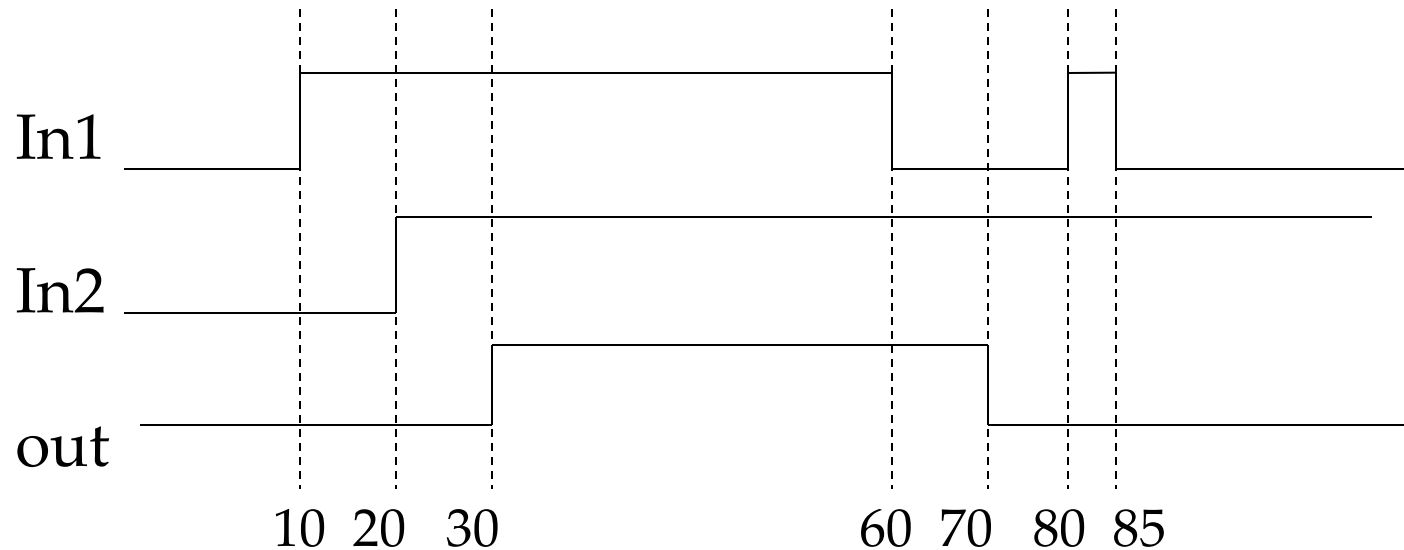
- Ex) **wire # 10 out;**

```
assign out = in1 & in2;
```

# Inertia Delay

- Delay model is basically **Inertia Delay Model**
  - An input pulse shorter than the delay of assignment statement does not propagate to the output

```
assign #10 out = in1 & in2;
```



# Expressions

- Expressions are constructs that combine operators and operands to produce a result
- Operands
  - constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector), part-select (more than 2 bit select), memories, function calls
  - Examples)  
integer count, final\_cnt;  
reg [51:0] reg1, reg2 ;  
reg [3:0] reg\_out ;  
reg ret\_val;  
  
final\_cnt = count + 1;  
reg\_out = reg1[3:0] ^ reg2[3:0];  
reg\_val = calculate\_parity (A,B) ;



# Verilog Operators

Verilog Operator	Name	Functional Group
()	bit-select or part-select	
()	parenthesis	
! ~ &   ~& ~  ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{}	concatenation	Concatenation
{{}}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

# Arithmetic Operators

- Types
  - $*$ ,  $/$ ,  $+$ ,  $-$
  - Modulus:  $\%$
- If any operands has  $x$  value, the the result of the expression is  $x$
- Negative numbers are represented as 2's complement. Do not use negative number except *integer and real type*  
(Ex)  $-d10 / 5$  ; // (2's complement of 10) / 5 =  $(2^{32}-10)/5$
- $\%$  (modulus) operator takes the sign of the first operand
  - $(-7) \% (+2) = -1$
  - $(+7) \% (-2) = +1$

# Logical operators

- Types
  - `&&` (logical AND), `||` (logical OR), `!` (logical Not)
- Logical operators evaluate to *1-bit value 1(True) or 0(False)*
- If an operand is not equal to zero, it is equivalent to a *True(1)*. If an operand is equal to zero, it is *False(0)*. If an operand is `x` or `z`, it is equivalent to `x` (ambiguous) and treated as false condition by simulators
- Examples)

```
A = 3; B = 0;
```

```
A && B // equivalent to 0 (logical-1 AND logical-0)
```

```
(A == 3) && ( B == 0) // evaluates to 1-bit value 1 (1'b1)
```

# Relational Operators

- The relational operators returns a logical value *1 or 0*
- Operators
  - > (greater than)                      < (less than)
  - >= (greater than or equal to)      <= (less than or equal to)
  - == (equality)                          != (inequality)
  - === (case equality: identical)      !== (case inequality: not identical)
- Case/logical equality
  - != and == return *x* if one of operands has *x* or *z* (returns 0, 1, or *x*)
  - !== and === compare bit by bit including *x* and *z* (returns either 1 or 0)

```
– Ex) // X=4'b1010, Y=4'b1101, Z=4'b1xxz, M=4'b1xxz
      X == Y    // 0
      X == Z    // x      M == Z // ? ➔ x
      M === Z   // 1
```

# Bitwise Operators

- $\sim$  (negation),  $\&$  (and),  $|$  (or),  $\wedge$  (xor),  $\wedge\sim$ ,  $\sim\wedge$  (xnor)

- Example)

```
// X = 4'b1010, Y = 4'b1101, Z = 4'b10x1
~ X      // Negation , Result is 4'b0101
X & Y    // Bitwise and , Result is 4'b1000
X | Y    // Bitwise or, Result is 4'b1111
X ^ Y    // Bitwise Xor, result is 4'b0111
X ^~ Y   // Bitwise Xnor, Result is 4'b1000
X & Z    // Bitwise Xor, Result is 4'b10x0
```

- Notice :

Do not confuse bitwise operator  $\sim$ ,  $\&$ ,  $|$  (returns a **vector value**) with logical operator  $!$ ,  $\&\&$ ,  $||$  (returns **1 bit** value)

# Reduction operators

- Are a Unary Operators
- Types :  $\&$ (AND),  $\sim\&$ (NAND),  $|$ (OR),  $\sim|$ (NOR),  $\wedge$ (EXOR),  $\sim\wedge$  or  $\wedge\sim$  (EXNOR)
- Perform a bitwise operation bit-by-bit from right to left on a single vector
- Yield 1-bit result
- (Examples)

```
// X = 4'b1010
```

```
& X // equivalent to 1 & 0 & 1 & 0 = 0
```

```
| X // equivalent to 1 | 0 | 1 | 0 = 1
```

```
^X //equivalent to 1 ^ 0 ^ 1 ^ 0 = 1
```

# Shift Operators

- Types : << (shift left)   >> (shift right)
- Examples)

```
// X = 4'b1101
```

```
Y = X >> 1 // Y is 0110 (0 is filled in MSB position)
```

```
Y = X << 2 // Y is 0100 (0 is filled in LSB position)
```

# Conditional Operator

- Has Three Operands
  - *Conditional\_exp ? True\_exp : false\_exp*
  - If its conditional\_exp returns 'x ' (ambiguous) both true\_exp and false\_exp is evaluated
- Is similar to MUX / if-else statement
- Can be nested
- Example)

```
assign out = control ? In1 : in0 ; // 2-to-1 MUX
```

```
assign out_sig = ( A == 3 ) ? ( control ? x : y ) : ( control ? m : n ); //nested
```



# Concatenation and Replication Operators

- Concatenation Operators

// A = 1'b1   B = 2'b00   C = 2'b10

Y = { B, C } // Results Y is 4'b0010

X = { A, B, 3'b110 } // Result X is 6'b100110

Z = { A, B[0], C[1] } // Result Z is 3'b101

- Replication Operators

reg A = 1'b1;

reg [1:0] B = 2'b01;

reg [1:0] C = 2'b00;

Y = { 4{A} } // Result Y is 4'b1111

X = { 4{A}, 2{B} } // Result X is 11110101

Z = { 4{A}, 2{B}, C } // Result Z is 1111010100

# Operator Precedence

1. Unary, Multiply, Divide, Modulus (\*, /, %)
2. Add, Subtract, Shift (+, -, <<, >>)
3. Relational, Equality (>, <, >=, <=, ==, !=, ===, !==)
4. Reduction, Logical (&, ~&, ^, ~^, |, ~|, &&, ||)
5. Conditional (?:)

↑ highest  
↓ lowest

# Multiplexer Example

```
module mux4_1 (out, i0, i1, i2, i3, s1, s0);  
output out;  
input i0, i1, i2, i3;  
input s1, s0;  
    assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1 ) |  
                (s1 & ~s0 & i2) | (s1 & s0 & i3) ;  
endmodule
```

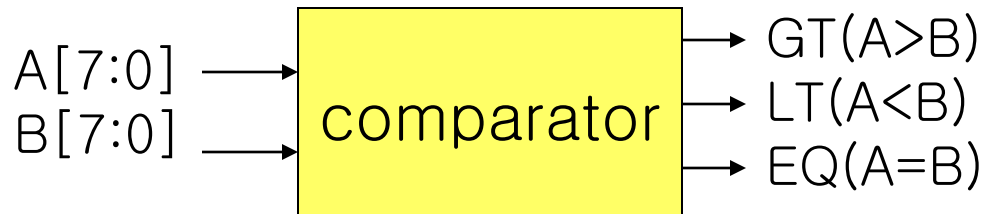
```
module mux4_1 (out, i0, i1, i2, i3, s1, s0);  
output out;  
input i0, i1, i2, i3;  
input s1, s0;  
    assign out = s1 ? (s0 ? i3:i2) : (s0 ? i1:i0);  
endmodule
```

# Exercises

1. Design Parity Generator /Checker for 7-bit data. If control signal PRT is 0 it is even parity system, Others it's odd parity system



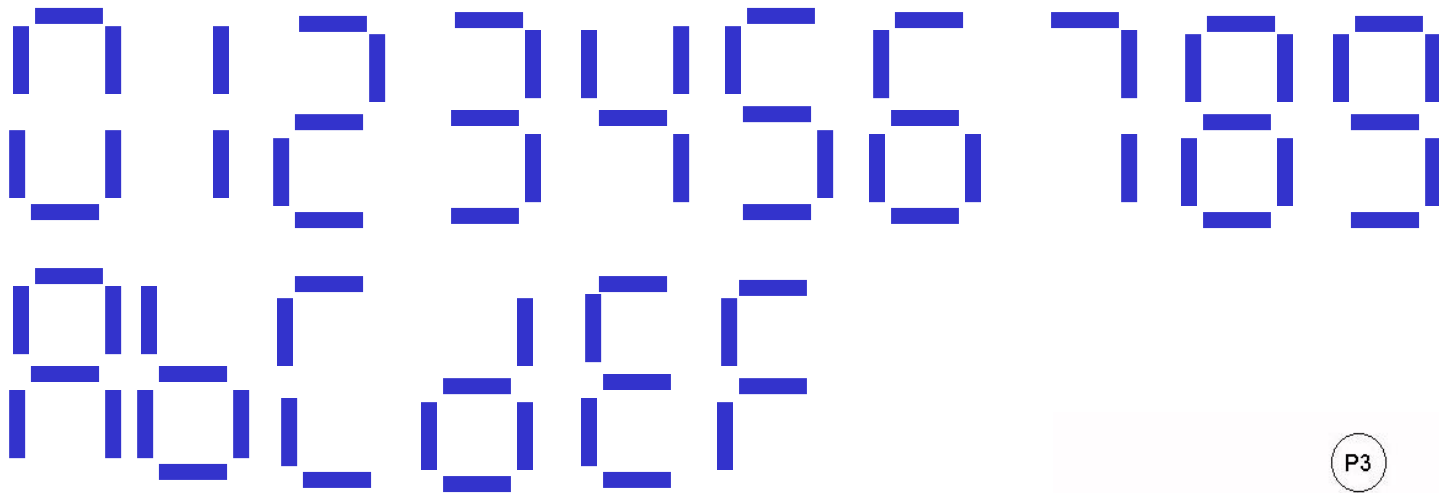
2. Design a magnitude comparator for two 4 bit data.



3. Design a 4-to-1 Multiplexer using (1) only conditional operators *or* (2) logical equation

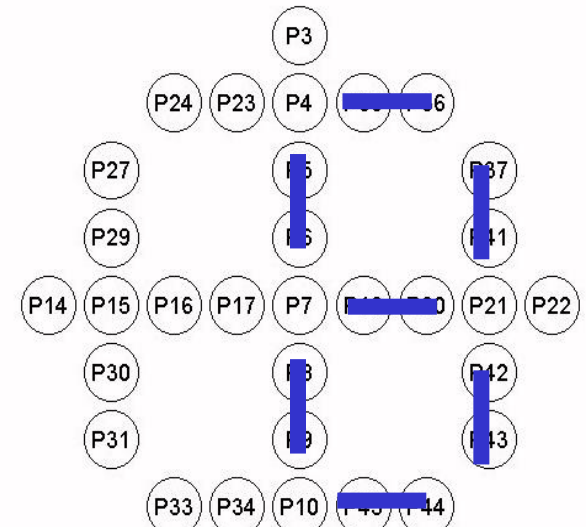
# Exercises [2]

4. Combinational Design : Design 7 segment Display decoder.  
Output form is as follows :



5. Hierarchical Design :

Target output LED shape is shown on the right. Use the module above as a submodule instance.



# Behavioral Modeling

- initial vs. always
- Blocking vs. non-blocking assignment
- Delay based timing control
- Event-based timing control
- Statements (if-else, case, while, repeat, forever)

# Behavioral Modeling

- Description of Design Functionality in an Higher Level
- All the behavioral statements appear only inside initial and always blocks
- always block / initial block : Tow main Structured procedure in behavioral modeling
  - These are Similar to Process Statements in VHDL
  - Each activity flow starts at time 0
  - They Cannot be Nested
  - They can Have Multiple Statements Between the keyword **begin** and **end**

# Initial Statement

- Usage : **initial** statement
- It starts *at time 0*
- Execute *exactly once* (do not re-executed during simulation)
- If multiple initial block exist, each of them starts *concurrently* and finish *independently* of the other blocks

```
Module stim; reg x,y,a,b,m ;  
initial m = 1'b1;  
initial begin  
    #5  a = 1'b1;  
    #25 b = 1'b0; end  
initial begin  
    #10 x = 1'b0;  
    #25 y = 1'b1;  
end  
initial #50 $finish;  
endmodule
```

Time	statement executed
0	m = 1
5	a = 1
10	x = 0
30	b = 0
35	y = 1
50	\$finish

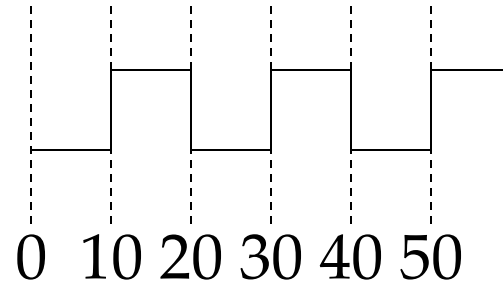


# Always Statements

- Usage : **always** statement
- Starts at time 0
- Executes the statements in the block continuously *in a looping fashion*
- Similar to *infinite loop* in C
- Starts on power-on (simulation begin), stop by power off (\$finish/\$stop)
- Models a block of activity that is repeated continuously (e.g. clock generator)

```
module clock_gen
reg clock;

initial clock = 1'b0;
always #10 clock = ~ clock;
initial #1000 $finish
```



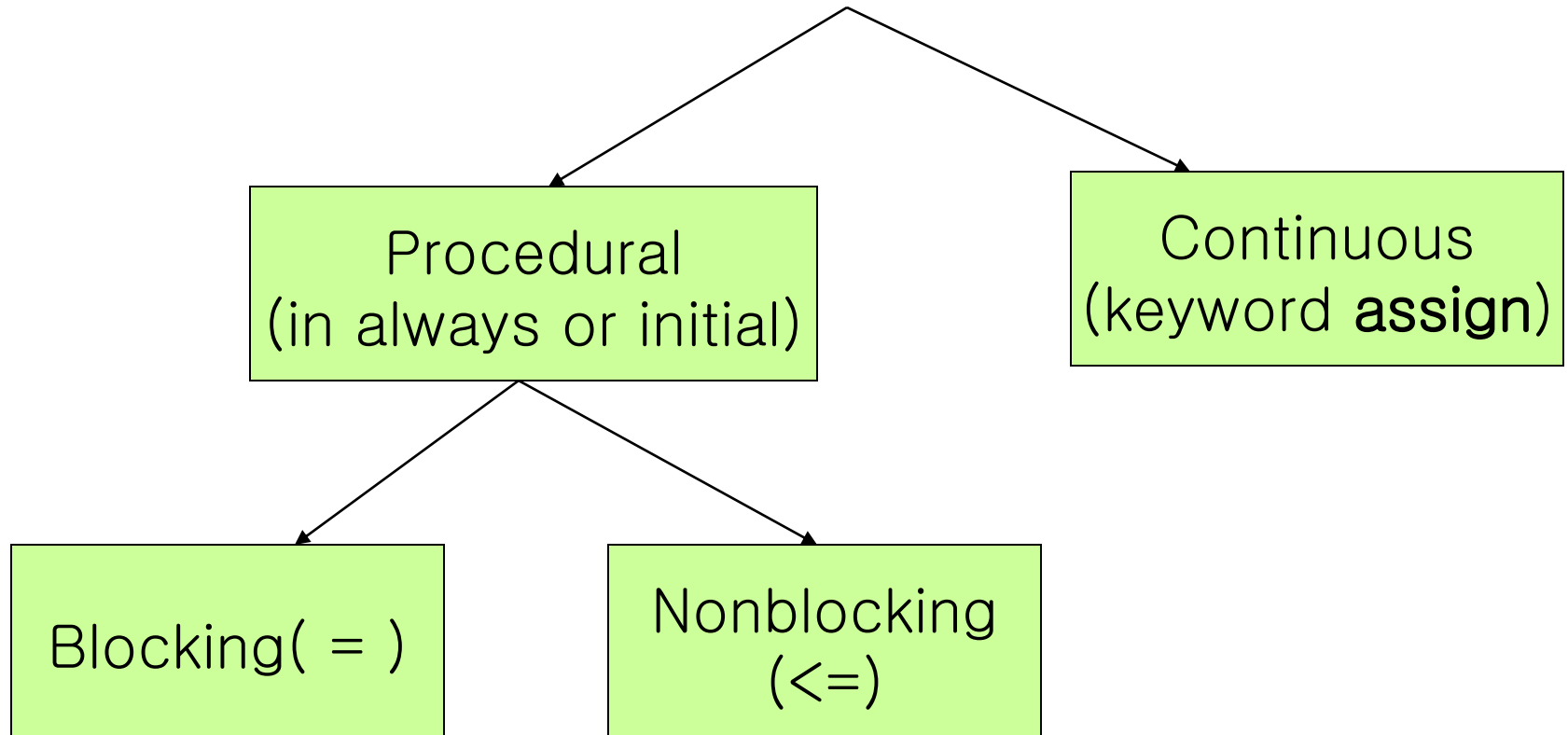
# Exercises

1. What happens if the following code is simulated

```
module test ;  
  reg a, b, c ;  
  always begin  
    a = b ;  
    c = b;  
  end  
  initial begin  
    b = 0;  
    #5 b = 1; end  
endmodule
```

Ans: simulation time cannot advance because of the infinite loop of always block

# Verilog Assignments types



# Procedural Assignments

- Updates values of reg, integer, real, or time variables (register types)
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable
- Syntax :
  - $\langle lvalue \rangle = \langle expression \rangle$  or
  - $\langle lvalue \rangle \leq \langle expression \rangle$
  - The lvalue can be reg, integer, real, or time register
  - The lvalue can be a concatenation of any of the above
  - In the expression of right hand, all the operators in dataflow modeling can be used
  - 2-types of procedural assignments : **Blocking and Non-Blocking**

# Blocking Assignments

- Are Executed in the order they are specified
- Are Sequential Exactly (*similar to variable assignments in VHDL process*)
- Use the '=' assignment operator

```
initial
begin
  x = 0; y = 1; z = 1;
  count = 0;
  reg_a = 16'b0; reg_b = reg_a;
  #15 reg_a[2] = 1'b1;
  #10 reg_b[15:13] = {x, y, z}
  count = count + 1;
end
```

1. Executes All Statements  $x=0$  through  $reg\_b = reg\_a$  (at time = 0)
2.  $reg\_a[2] = 1$  at time = 15
3.  $reg\_b[15:13] = \{x,y,z\}$  at time = 25

# Nonblocking Assignments

- Schedule assignments without blocking execution of the next statements in a sequential block (*similar to signal assignments in VHDL process*)
- Use Operator `<=`
- **Example)** `A <= B;`
- When Verilog Simulator Sees NonBlocking Assignment Statements it *Schedules* the Statements (Read or Evaluate the RHS expression and then Store the RHS value in temporal storage) and *Continue to the Next Statement* without Waiting for the Nonblocking Statement to Complete Execution.

# Comparing Nonblocking and Blocking Assignments

```
initial
begin
  x = 0; y = 1; z = 1;
  count = 0;
  reg_a = 16'b0; reg_b = reg_a;
  reg_a[2] <= #15 1'b1;
  reg_b[15:13] <= #10 {x, y, z}
  count = count + 1;
end
```

1. Execute All Statements  $x=0$  through  $reg_b = reg_a$  (at time = 0)
2.  $reg_a[2] = 1$  is scheduled to execute after 15 time units (i.e. time = 15)
3.  $reg_b[15:13] = \{x,y,z\}$  is scheduled to execute after 10 time units (i.e. time = 10)
4.  $count = count + 1$  is scheduled to be executed without any delay (i.e. time = 0)

# Separating Read and Write in Nonblocking Assignments

```
always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1; // old value of reg1
end
```

At each positive edge of clock,

1. *Read* operation is performed on each RHS(right-hand-side) variable *in1*, *in2*, *in3*, and *reg1*. And right-hand-side expressions are *evaluated*, and the results are *stored internally* in temporary storage of the simulator
2. The *write* operation to the left-hand-side variables are *scheduled* to be executed at the time specified
3. The *write* operations are *executed* at the scheduled time steps. (Note that the order of statements is **not important** because internally stored RHS expression values are used to be assign to the LHS variables)



# Race Condition

```
// illustration 1 : blocking  
always @(posedge clk)  
    a = b ;  
always @(posedge clk)  
    b = a;
```

```
// illustration 2 : nonblocking  
always @(posedge clk)  
    a <= b ;  
always @(posedge clk)  
    b <= a;
```

**There is a race condition.** Two *blocking* assignments in different always are executed sequentially depending on the simulator implementation. The result will be  $b \leftarrow a \leftarrow b$  or  $a \leftarrow b \leftarrow a$

**Race condition is eliminated.**

At the rising edge of clock, RHS value of *nonblocking* assignments are “read” and the expressions evaluated. During write operation, the stored values are used

# Rewriting Nonblocking Assignments

```
// illustration 2 : nonblocking
always @(posedge clk)
  a <= b ;
always @(posedge clk)
  b <= a;
```

Separating *Read* and *Write* Ensures *a* and *b* are swapped  
*Regardless of the Write Operation Order !*

```
// illustration 2 : nonblocking
always @(posedge clk) begin
  temp_a = a; // read operation
  temp_b = b; // read operation
  a = temp_b; // write operation
  b = temp_a; // write operation
end
```

# Why Nonblocking Assignment ?

- NonBlocking Assignments are Used as a Method to Model Several **Concurrent Data Transfers** that take place *After a Common Event*.
- In Nonblocking Assignments, Final Assignment Results Are Not Dependent On the Order They Are Evaluated.  
While In Blocking Assignments, Final results Depends on the Order in which they are Evaluated
- Blocking Assignments can Potentially Cause a Race Condition

# Design Guidance concerned to Signal Assignments

- For combinational logic description Use *Blocking Assignment*
- For data transfer in edge-triggered Memory Elements description Use *Nonblocking Assignment*
- Do not Mix Blocking and Nonblocking Assignments in the same always / initial block
- Do not use the same variable as RHS variable in different always / initial blocks => to prevent race condition

# Verilog Event Queue Priority

Queue Execution  
Order

High  
priority

Blocking assignments,  
\$display

(#0)Zero delay assignments

Update LHS of  
Non-blocking assignments

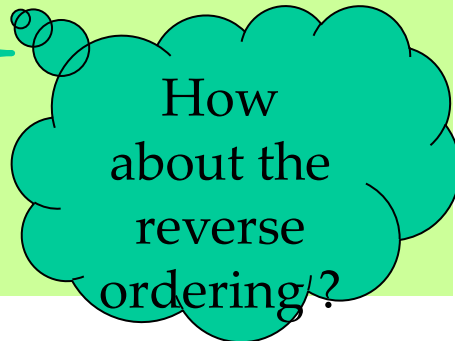
Low  
priority

monitoring statements  
(\$monitor, \$strobe)

# Exercises (1)

1. What is the difference of the two codes ?

```
module pipe1 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d;  
    input clk;  
    reg [7:0] q3, q2, q1;  
  
    always @(posedge clk) begin  
        q1 = d;  
        q2 = q1;  
        q3 = q2;  
    end  
endmodule
```



```
module pipe2 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d;  
    input clk;  
    reg [7:0] q3, q2, q1;  
  
    always @(posedge clk) begin  
        q1 <= d;  
        q2 <= q1;  
        q3 <= q2;  
    end  
endmodule
```

# Exercises (2)

2. What is the difference of the simulation results of the 3 codes ?

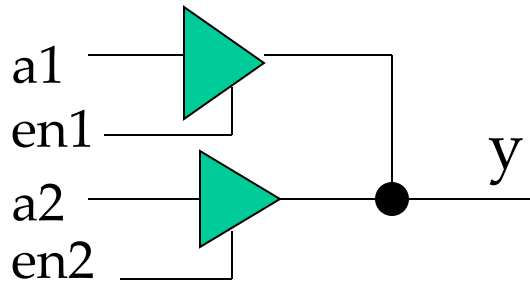
```
// generates clock signal of period 10
module osc1 (clk);
    output clk;
    reg clk;
    initial #5 clk = 0;
    always @(clk) #5 clk = ~clk;
endmodule
```

```
module osc1 (clk);
    output clk;
    reg clk;
    initial #5 clk = 0;
    always @(clk) #5 clk <= ~clk;
endmodule
```

```
// generates clock signal of period 10
module osc1 (clk);
    output clk;
    reg clk;
    initial #5 clk = 0;
    always #5 clk = ~clk;
endmodule
```

# Exercises (3)

3. Find the Difference of the 2 codes. Which corresponds to the picture ?



```
module drivers_cont
    (y,a1,a2,en1,en2);
output y;
input a1, a2, en1, en2;

assign y = en1 ? a1 : 1'bz;
assign y = en2 ? a2 : 1'bz;

endmodule
```

```
module drivers_seq (y,a1,a2,en1,en2);
output y;
input a1, a2, en1, en2;
reg y;

always @(en1 or a1)
    if(en1) y = a1; else y = 1'bz;
always @(en2 or a2)
    if(en2) y = a2; else y = 1'bz;

endmodule
```



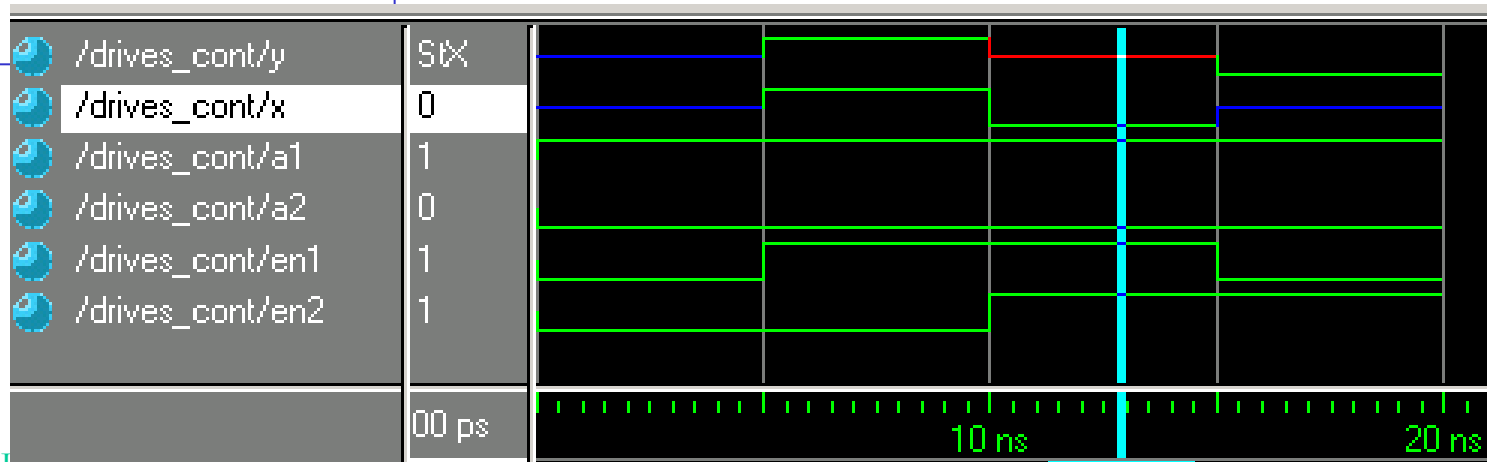
# Ans for Exercises(3)

```
module drives_cont;
wire y;
reg x;
reg a1,a2,en1,en2;
```

```
assign y = en1 ? a1:1'bz;
assign y = en2 ? a2:1'bz;
```

```
always @(en1 or a1)
  if(en1) x = a1; else x = 1'bz;
always @(en2 or a2)
  if(en2) x = a2; else x = 1'bz;
```

```
initial begin
a1 = 1;
a2 = 0;
en1 = 0;
en2 = 0;
#5 en1 = 1;
#5 en2 = 1;
#5 en1 = 0;
end
endmodule
```



# Timing Control

- Timing Control Provides a Way to Specify the Simulation Time at which Procedural Statements will Execute
- If there is No Timing Control Statements, the Verilog Simulator will Not Advance !
- 3 methods of timing control
  - Delay-based control
  - Event-based control
  - Level sensitive control

# Delay-based Timing Control

- Delays are Specified Explicitly in the Procedural Statements

- Syntax :

`<delay> ::= #<number> | #<identifier> |`

`#(<min_typ_max_exp><, <min_typ_max_exp>>*)`

- 3 types of delay-based timing controls

```
initial
#10 z = x + y;
z = #10 x + y
end
```

regular delay  
control

```
initial begin
#0 x = 1
end;
```

intra-assignment  
delay control

```
initial begin
x = 1;
end
```

Zero delay control

# 3 types of delay control

- Regular delay control
  - Is Specified *left* of a procedural statement
  - Defers the execution of entire assignment statements
  - Example) #3 x = y ;
- Intra-assignment delay control
  - Is Specified to the *right* of the assignment operator
  - **Computes** (*Evaluates*) the *RHS* expression **at current time** and **defer the assignment** of the computed value to the *LHS* variable
  - Example) x = #3 y ; —————> In VHDL, x <= y after 3 ns;
- Zero Delay control
  - #0 *procedural\_statements*
  - Example) #0 x = y
  - Objective : To Eliminate the race condition between different always-initial blocks at time 0

# Regular delay control

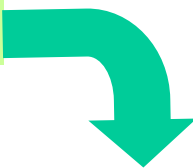
```
parameter latency = 20;
parameter delta = 2 ;

reg x,y,z,p,q;

initial
begin
    x = 0; z = 0 ;
    #10 y = 1;
    #latency z = 0;
    #(latency + delta) p = 1; // delay with expression
    #y x = x + 1; // delay with identifier takes the y value
    #(4:5:6) q = 0; // minimum, typical, maximum values
end
```

# Intra-assignment delay control

```
// intra-assignment delay example
initial
begin
    x = 0; z = 0 ;
    y = #5 x + z; // Take the value of x and z at time 0 and evaluate
                  // x + z and then wait 5 time units to assign it to y
end
```



```
initial // Regular delay control with temporary variable has
        // equivalent effect to the intra-assignment delay above
    x = 0; z = 0;
    temp_xz = x + z; // takes the value of x + z and stores it in a temporary variable.
    #5 y = temp_xz; //Even though x and z might be change between 0 and 5,
                    // the value assigned to y at time 5 is unaffected
end
```

# Zero Delay control

```
initial
begin
  x = 0;
  y = 0;
end
```

```
initial
begin
  #0 x = 1;
  #0 y = 1;
end
```

Results : (deterministic)  
x = 1 and y = 1

- The **Order** of Procedural Statement Executions in *Different Blocks* at the *Same Simulation Time* is **Nondeterministic**
- Zero Delay Control Ensure that the statement is **executed last**, after all other statements in that simulation time.
- But, Between Multiple Zero Delay Statement, The Order Between Them is also Nondeterministic.

# Exercises

## 1. What is the simulation waveform ?

```
module delay_control_test ;  
  
integer a,b,c,d,x ;  
integer A,B,C,D ;  
// blocking assignment  
initial begin  
    #0 a = 0;  
        b = 0;  
        c = 0;  
        d = 0;  
        a = #35 x;  
    #15 b = x;  
    #10 c = # 20 x;  
    d = #10 x;  
end  
  
// nonblocking  
initial  
begin  
    #5 A = 0;  
        B <= 0;  
        C = 0; D = 0;  
        A <= #10 x;  
    #10 B <= x;  
    #10 C <= # 20 x;  
        D <= #10 x;  
end  
  
// stimulus  
initial  
begin  
    x = 5;  
    #10 x = 6;  
    #10 x = 7 ;  
    #10 x = 8;  
    #10 x = 9;  
    #10 x = 10;  
    #10 x = 11;  
    #10 x = 12;  
    #30 $stop;  
end  
endmodule
```



# Hints for the Answer : Blocking

Time for the execution of the assignment statement ( $x = \#5\ y$ )

#10  $X = \#5\ Y ;$

Time for the value of Y to be transferred to the variable X

The control will not pass to the next statement. until the execution (not scheduling ) of the **blocking assignment** is completed. (after 10 + 5 time, the control is passed to the next statement)

Value of Y is read & saved

Saved Y is copied to X

Time = 0

Time = 10

Time = 15

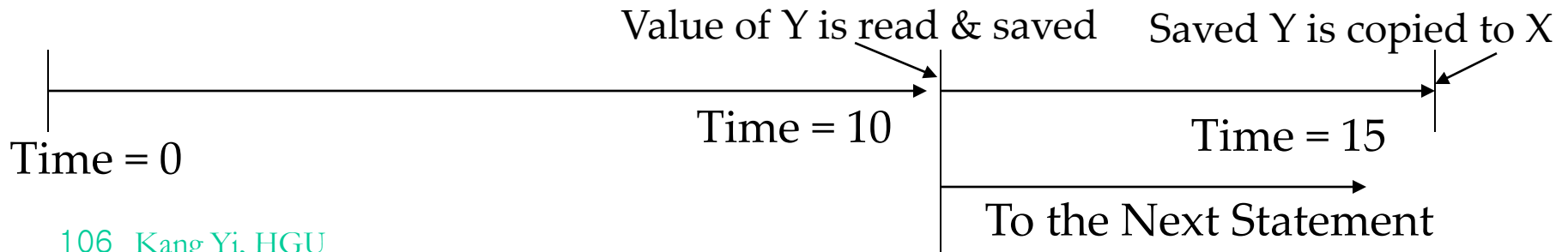
# Hints for the Answer : Nonblocking

Time for the execution of the assignment statement ( $x \leq \#5 y$ )

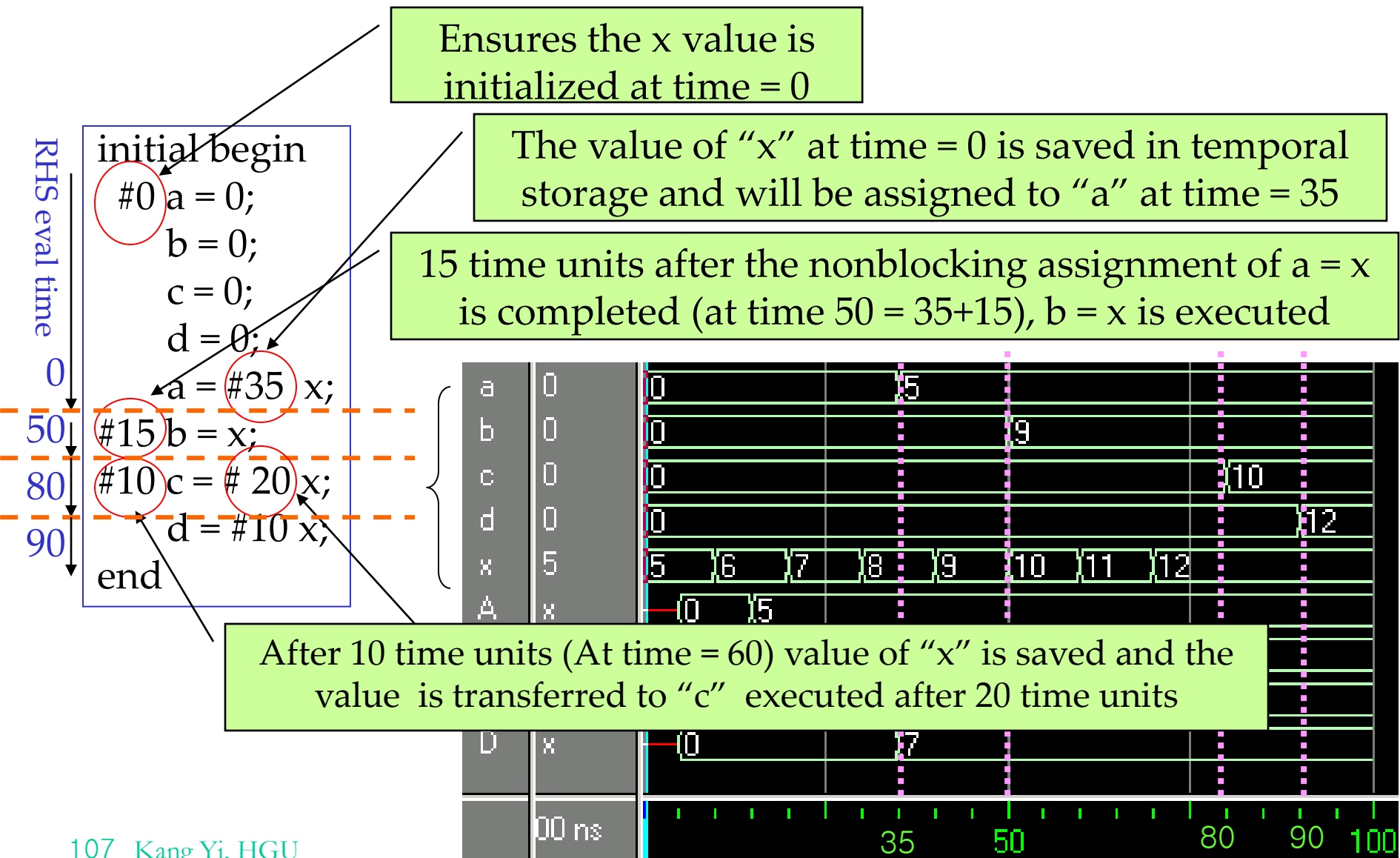
**#10**  $X \leq \#5 Y ;$

Time for the value of Y to be transferred to the variable X

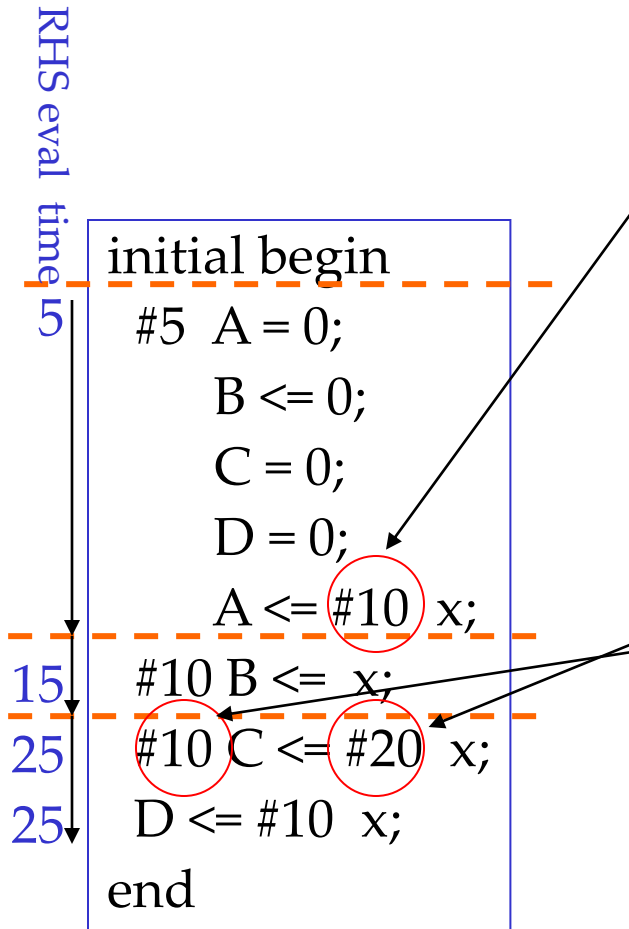
After the evaluation of RHS expression (time = 10) of **nonblocking** assignments, the Control will be passed to the Next Statements



# Answer (1)



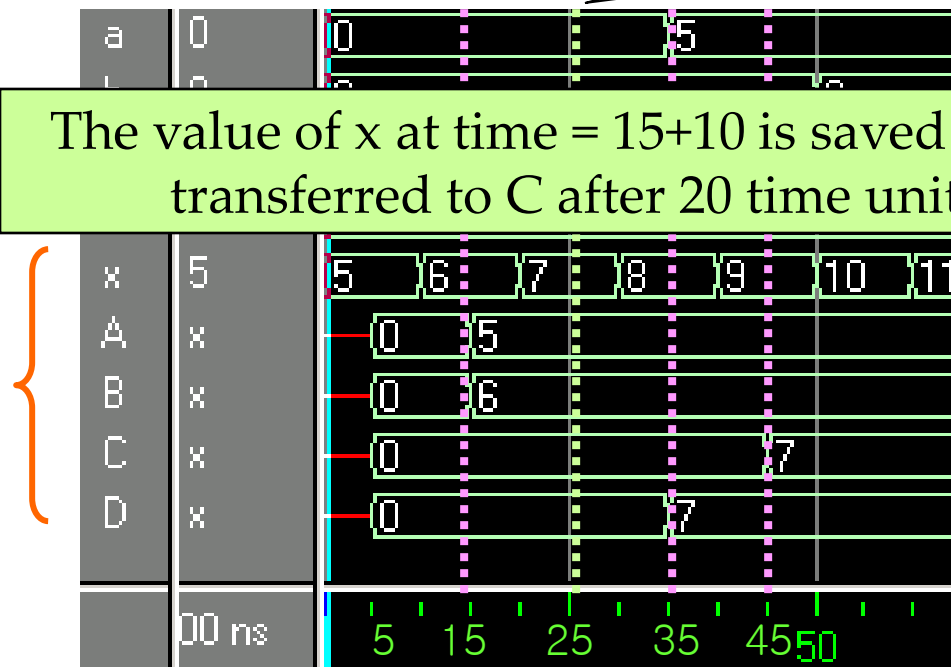
# Answer (2)



The value of x at time = 5 is saved in temporal storage and transferred to A after 10 time units

The value of x (7) is read

The value of x at time = 15+10 is saved and is transferred to C after 20 time units



# Event-based Timing Control

- An Event is a change in the value on register or a net
- Event can be used to trigger execution of a statement or a block of statements
- 4 types of event-based timing control
  - regular event control (@)
  - named event control (-> *and* @)
  - event OR control (*or*)
  - level-sensitive timing control (*wait*)

# Regular event control

- Use “@” symbol to specify event control
- The keyword **posedge** and **negedge** are used to specify transition on the signal value (edge-sensitive)

```
@(clock) q = d; // whenever clock changes value
```

```
@(posedge clock) q = d; // whenever clock does a positive transition  
// (0 to 1, x, or z , x to 1, z to 1)
```

```
@(negedge clock) q = d; // whenever clock does a negative transition  
// (1 to 0, x, or z , x to 0, z to 0)
```

```
q = @(posedge clock) d ; // d is evaluated immediately  
// and assigned to q at the next positive edge of clock signal
```

# Named event control

- In Verilog We can Declare an Event and then Trigger and Recognize the Occurrence of the Event
- An Event is triggered by the symbol “->”
- The triggering of an event is recognized by the symbol “@”

```
event received_data; // declare event variable

always @(posedge clock)
    if(last_data_packet) -> received_data; //trigger event

always @(received_data)
    data_buf = {data_pkt[0], data_pkt[1], data_pkt[2]};
```

# Event OR control

- A Transition on Any One of Multiple Signals or Events can Trigger the Execution of Statements
- The keyword “**OR**” is used to specify multiple triggers

```
always @(reset or clock or d)
begin
    if(reset)
        q = 1'b0;
    else if (clock)
        q = d;
end
```



# Level-Sensitive Timing Control

- Verilog has The ability to wait for a certain condition to be true (Note that @ provide the edge-sensitive control)
- The keyword **wait** is used for level-sensitive constructs

```
always
    wait (count_enable) #20 count = count + 1;

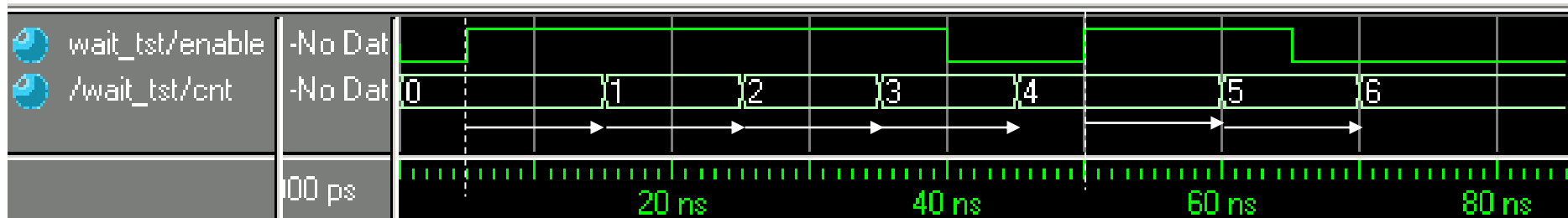
// count_enable is monitored continuously.
// If count_enable = 1, the statement count = count + 1 is
// executed after 20 time units
//     Note that if count_enable stays at 1,
//     count will be incremented every 20 time units
```

# More on the wait (level-sensitive)

```
module wait_tst ;  
  reg enable;  
  integer cnt = 0;
```

```
  initial begin  
    cnt = 0;  
    enable = 0;  
    #5 enable = 1;  
    #35 enable = 0;  
    #10 enable = 1;  
    #15 enable = 0;  
    #20 $stop;  
  end
```

```
  always  
    wait(enable) #10 cnt = cnt + 1;  
  
endmodule
```



# Conditional Statements (if)

- Types :

**if**(<expr>) *true\_statement*; // type 1

**if**(<expr>) *true\_statement* **else** *false\_statement* // Type 2

**if**(<expr1>) *true\_statement1* ; // Type 3  
**else if**(<expr2>) *true\_statement2*;  
**else if**(<expr3>) *true\_statement3*;  
**else** *default\_statement*;

- If <Expression> result is zero, the *true\_statement* is executed
- Each *true\_statement* and *default\_statement* can be a group of statements enclosed by keywords **begin** and **end**

# Multiway Branching (case)

- Case Statement Syntax

**case** (<expr>)

*alternative1* : statement1;

*alternative2* : statement2;

...

**default** : default\_statement ; // optional

**endcase**

- The <expr> is compared to *alternatives* in ***the order they are listed (Priority)***. If none of values are matched, default\_statement is executed
- A block of multiple statements must be a grouped by keywords **begin** and **end**
- Case statements can be nested
- If <expr> and *alternatives* are not equal width, they are *filled with 0s* to match the width of the widest of the them

# Multiway Branch (casex, casez)

- If expr contains x and z value, default is matched
- Casez allows comparison of only non-x and/or non-z position in the case <expr> and the alternatives
- Casez treats all z values in <expr> or alternatives as *don't cares*
- **Casex** treats all x and z values in <expr> or alternatives 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 = 5;  
endcase
```

- encoding = 4'b10xz matches 4'b1xxx resulting in next\_state = 3
- encoding = 4'b00x0 matches 4'bx1xx resulting in next\_state = 1
- encoding = 4'b0110 matches 4'bx1xx resulting in next\_state = 2;
- encoding = 4'bxxxx matches 4'b1xxx resulting in next\_state = 3;

# Example (casex vs. case )

```

module case_test ;

reg [3:0] encode;
integer ns, ns2;

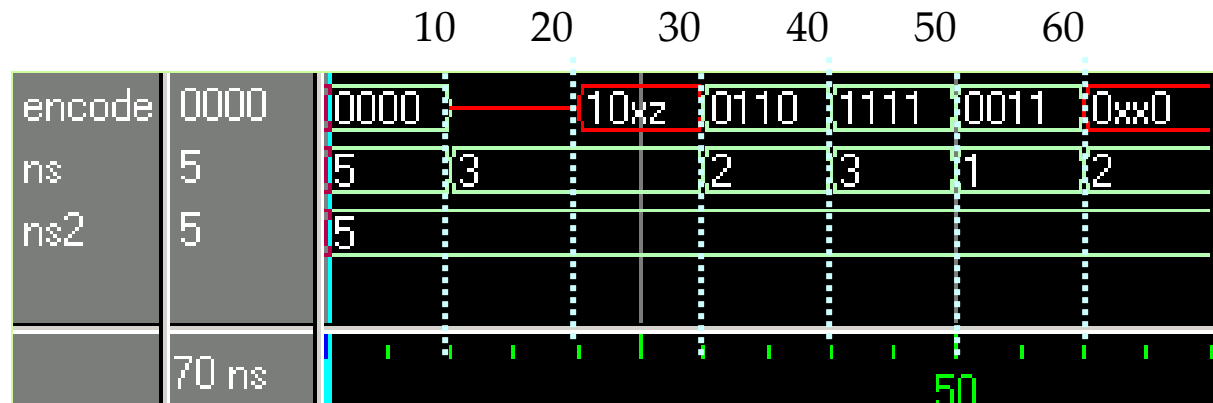
initial
begin
  encode = 0;
  #10 encode = 4'bxxxx;
  #10 encode = 4'b10xz;
  #10 encode = 4'bx11x;
  #10 encode = 4'b1111;
  #10 encode = 4'b0001;
  #10 encode = 4'b0xx0;
  #10 $stop;
end
    
```

```

always @(encode)
begin
  casex (encode)
    4'b1xxx : ns = 3;
    4'bx1xx : ns = 2;
    4'bxx1x : ns = 1;
    4'bxxx1 : ns = 0;
    default : ns = 5;
  endcase
end
    
```

```

case (encode)
  4'b1xxx : ns2 = 3;
  4'bx1xx : ns2 = 2;
  4'bxx1x : ns2 = 1;
  4'bxxx1 : ns2 = 0;
  default : ns2 = 5;
endcase
end
endmodule
    
```



# Loops

- **while** (<expr>) statements
- **for** (initial condition ; check for terminal condition ; procedural assignment to control variable) statements
- **repeat** (a fixed iteration number) statements
- **forever** statements

```
initial
  cnt = 0;
  while (cnt < 128) begin
    $display("Count=%d", cnt);
    cnt = cnt + 1;
  end
end
```

```
initial begin
  clock = 1'b0;
  forever #10 clock = ~clock;
end
```

```
initial
  for (cnt = 0; cnt < 128; cnt = cnt + 1)
    $display("Count=%d", cnt);
```

```
parameter cycle = 8;
reg [15:0] buff [0:7];
always @(posedge clock)
begin
  if(data_start) begin
    repeat (cycle) begin
      @(posedge clock) buff[i] = data;
      i = i + 1;
    end
  end
end
end
```

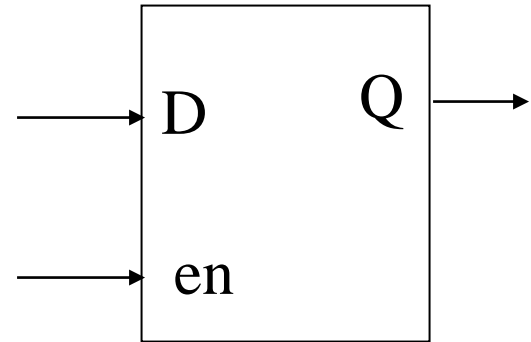
# Application to Synchronous Logic

- Latches
- Flip-Flops & Registers
- Shift Registers
- Counters
- BCD Counters & Cascading Counters



# Level Sensitive Latch

```
module latch (Q, en, D);  
  output Q;  
  reg Q;  
  input en, D;  
  
  always @(en or D)  
    if(en) Q <= D;  
  
endmodule
```



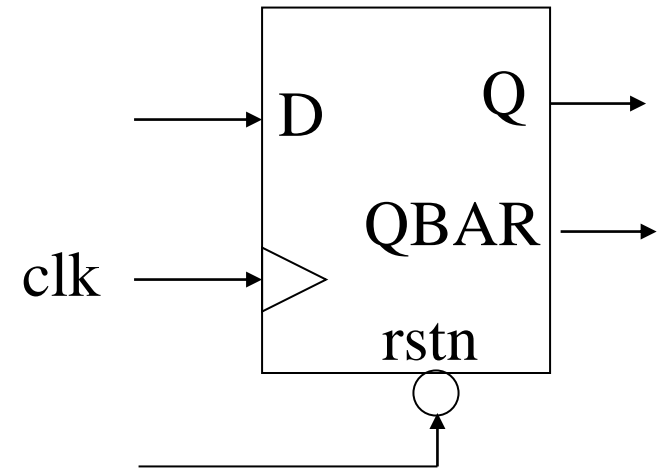
# D flip-Flops

```
// FlipFlop with asynchnous reset
module DFF (Q,QBAR, D, clk, rstn);
output Q, QBAR;
input D,clk, rstn;
reg Q;

always @(posedge clk or negedge rstn)
  if(!rstn) Q <= 1'b0;
  else Q <= D;

assign QBAR = ~ Q ;

endmodule
```



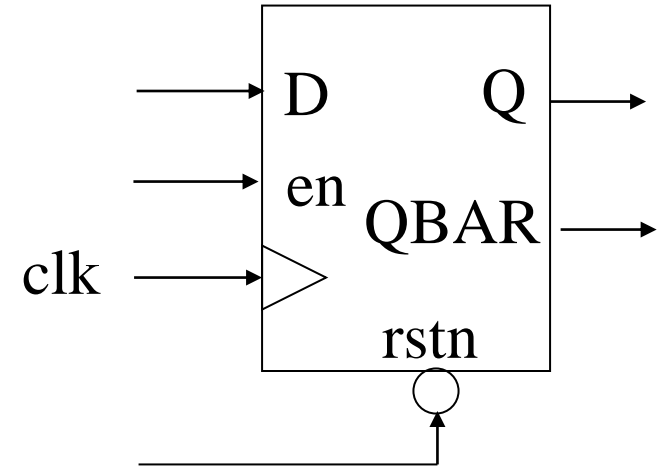
# D Flip-Flops (2)

```
// DFF with synchronous reset and enable
module DFF2 (Q,QBAR, D, en, clk, rstn);
output Q, QBAR;
input D,en, clk, rstn;
reg Q;

always @(posedge clk )
  if(!rstn) Q <= 1'b0;
  else if (en) Q <= D;

assign QBAR = ~ Q ;

endmodule
```

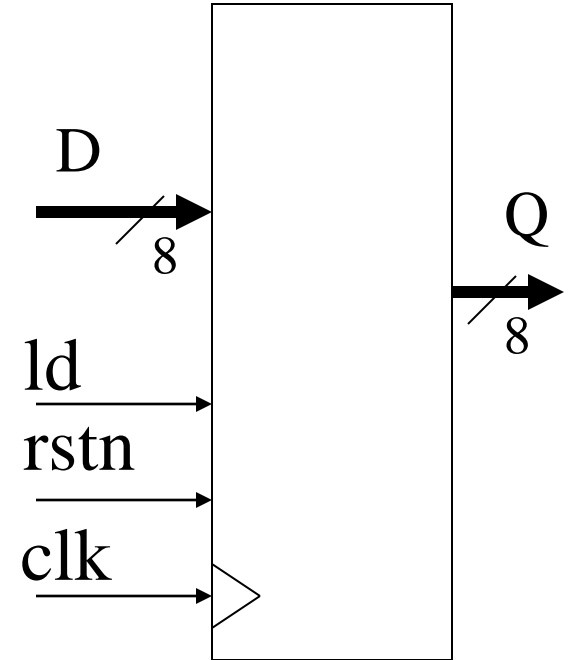


# Data Registers

```
// 8-bit register with synchronous load and reset
module REG (Q, D, ld, rstn, clk);
output [7:0] Q;
input [7:0] D;
input ld, rstn, clk;
reg [7:0] Q;

always @(posedge clk )
    if(!rstn) Q <= 8'd0;
    else if (ld) Q <= D;

endmodule
```

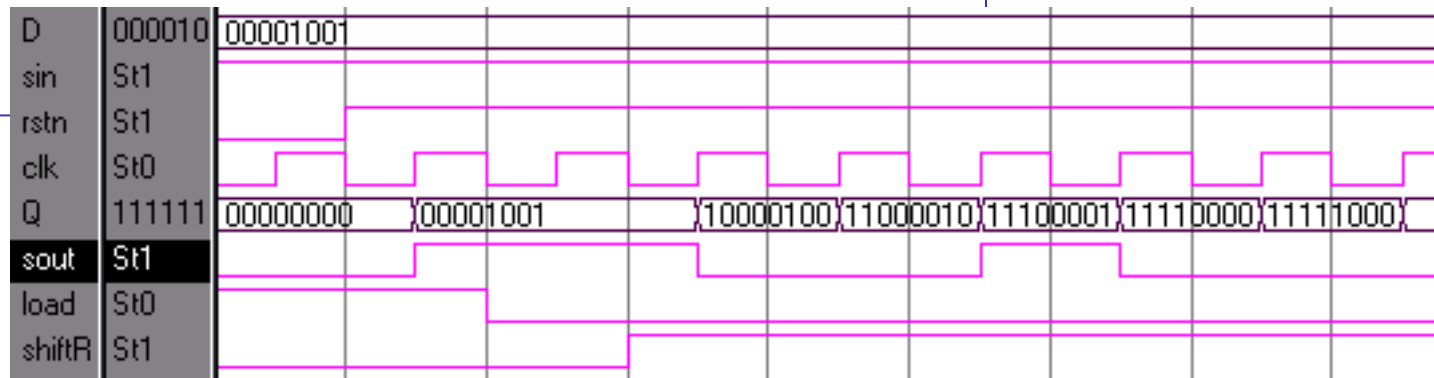


# Shift Registers (PISO type)

```
//shiftregister with asynch reset and synch load, shift,
module shiftReg (Sout,shiftR, D, Sin, load, clk, rstn);
output Sout;
input [7:0] D ;
input shiftR,Sin, load, clk, rstn;
reg [7:0] Q; // temporal register
```

```
always @(posedge clk or negedge rstn)
  if(!rstn) Q <= 8'b0;
  else if (load) Q <= D; // parallel in
  else if (shiftR) Q <= {Sin, Q[6:0]};
assign Sout = Q[0]; // serial out
```

```
endmodule
```



# 4-bit Binary Counter

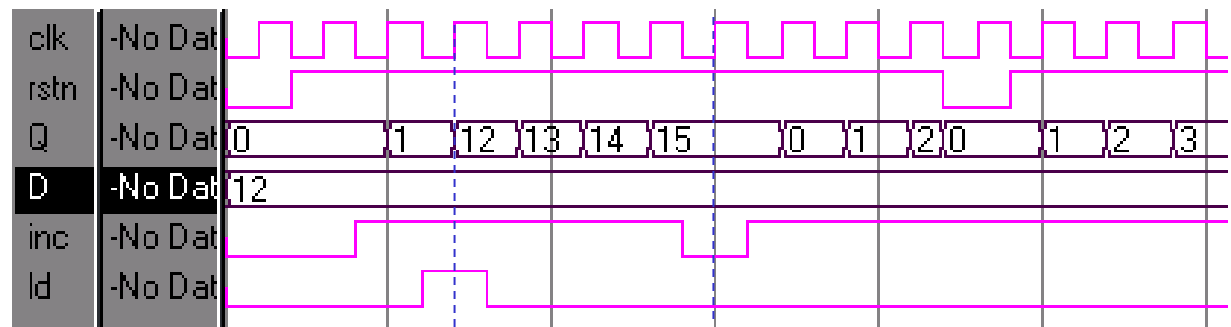
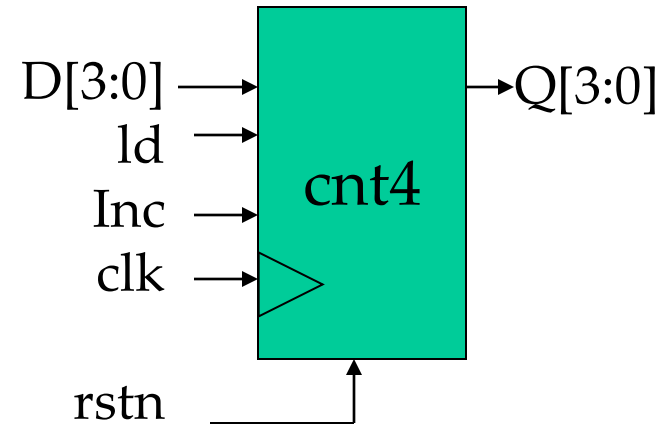
```

module cnt4 (Q, D, ld, inc, rstn, clk);
output [3:0] Q;
input [3:0] D;
input ld, inc, rstn, clk;
reg [3:0] Q;

always @(posedge clk or negedge rstn)
    if(!rstn) Q <= 4'd0;
    else if (ld) Q <= D;
    else if (inc) Q <= Q + 1;

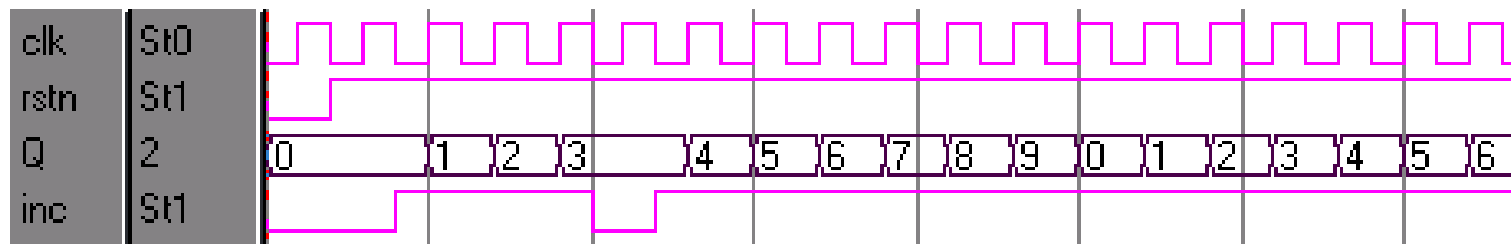
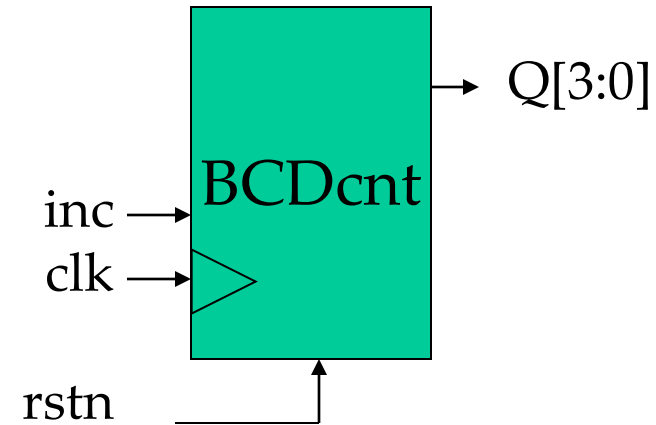
endmodule

```



# BCD Counter

```
module BCDcnt (Q, inc, rstn, clk);  
    output [3:0] Q;  
    input inc, rstn, clk;  
    reg [3:0] Q;  
  
    always @(posedge clk or negedge rstn)  
        if(!rstn) Q <= 4'd0;  
        else if (inc)  
            if(Q == 4'd9) Q <= 4'd0;  
            else Q <= Q + 1;  
  
endmodule
```

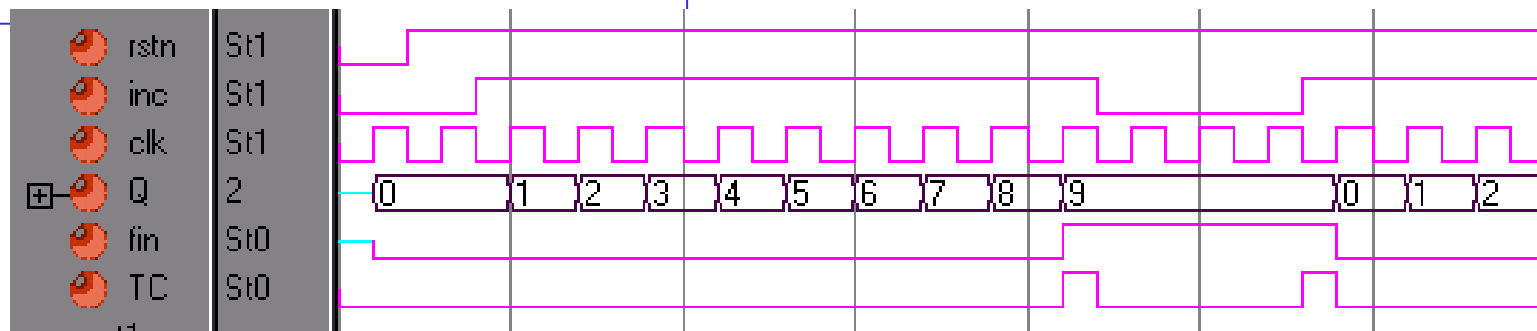
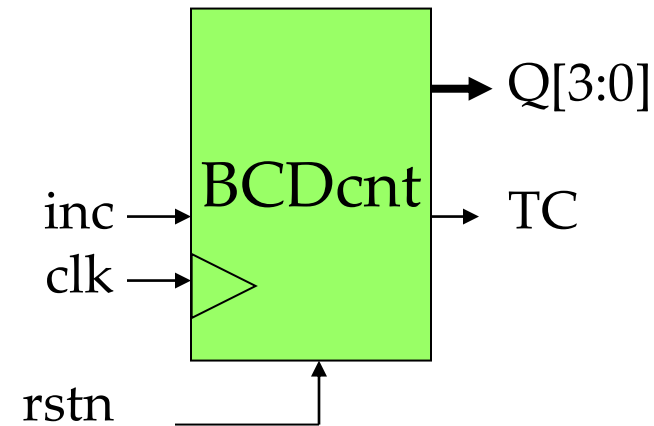


# Cascadable BCD Counter

```

module BCDcnt_cas (Q, TC, inc, rstn, clk);
output [3:0] Q;
output TC; // indicate terminal count
input inc, rstn, clk;
reg [3:0] Q;
wire fin; // Q reached terminal value
always @(posedge clk)
    if(!rstn) Q <= 4'd0; // synchronous reset
    else if (inc)
        if(fin) Q <= 4'd0;
        else Q <= Q + 1;
assign fin = (Q == 4'd9)?1:0;
assign TC = fin & inc;
endmodule

```





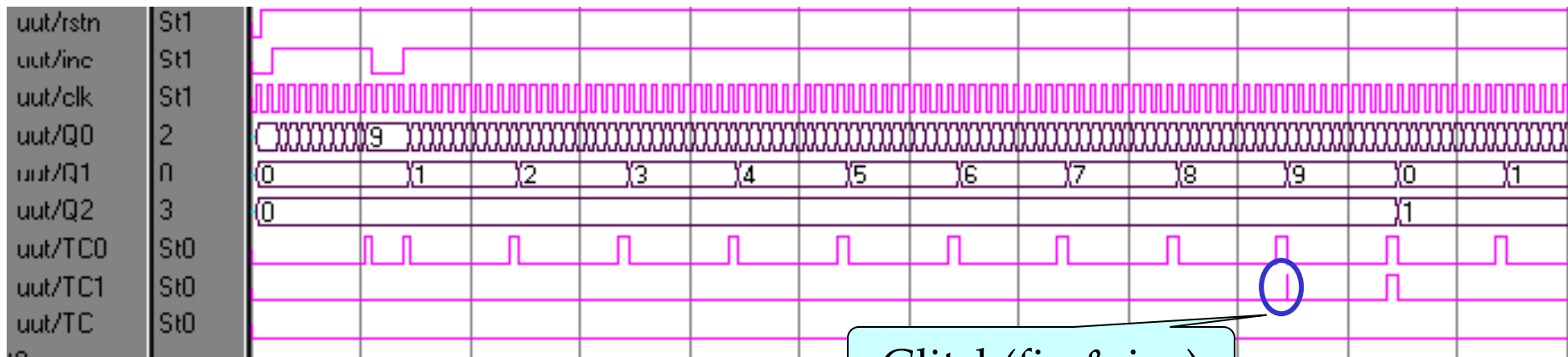
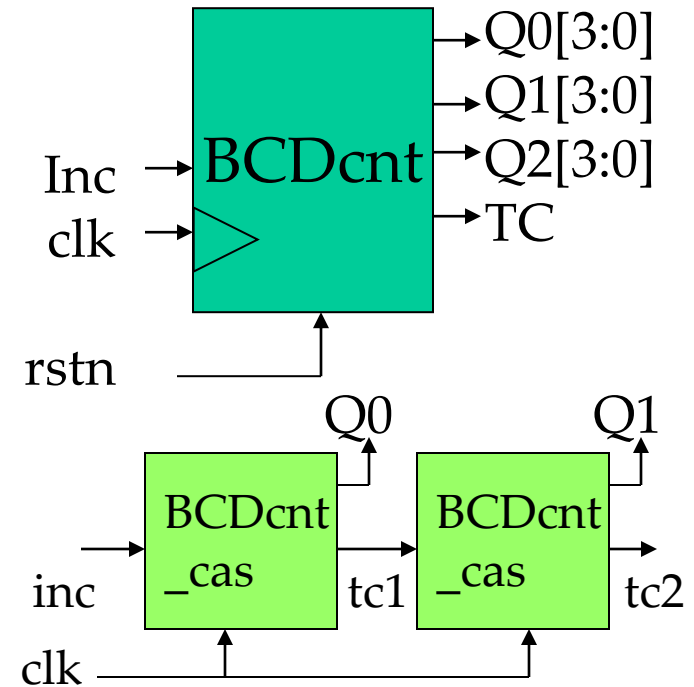
# Cascading 3-digit BCD Counters

```

module BCD_3digit (Q0,Q1,Q2, TC, inc, rstn, clk);
output [3:0] Q0,Q1,Q2 ;
output TC;
input inc, rstn, clk;
wire tc1,tc2;

BCDcnt_cas CNT0(Q0, tc1, inc, rstn, clk);
BCDcnt_cas CNT1(Q1, tc2, tc1, rstn, clk);
BCDcnt_cas CNT2(Q2, TC, tc2, rstn, clk);

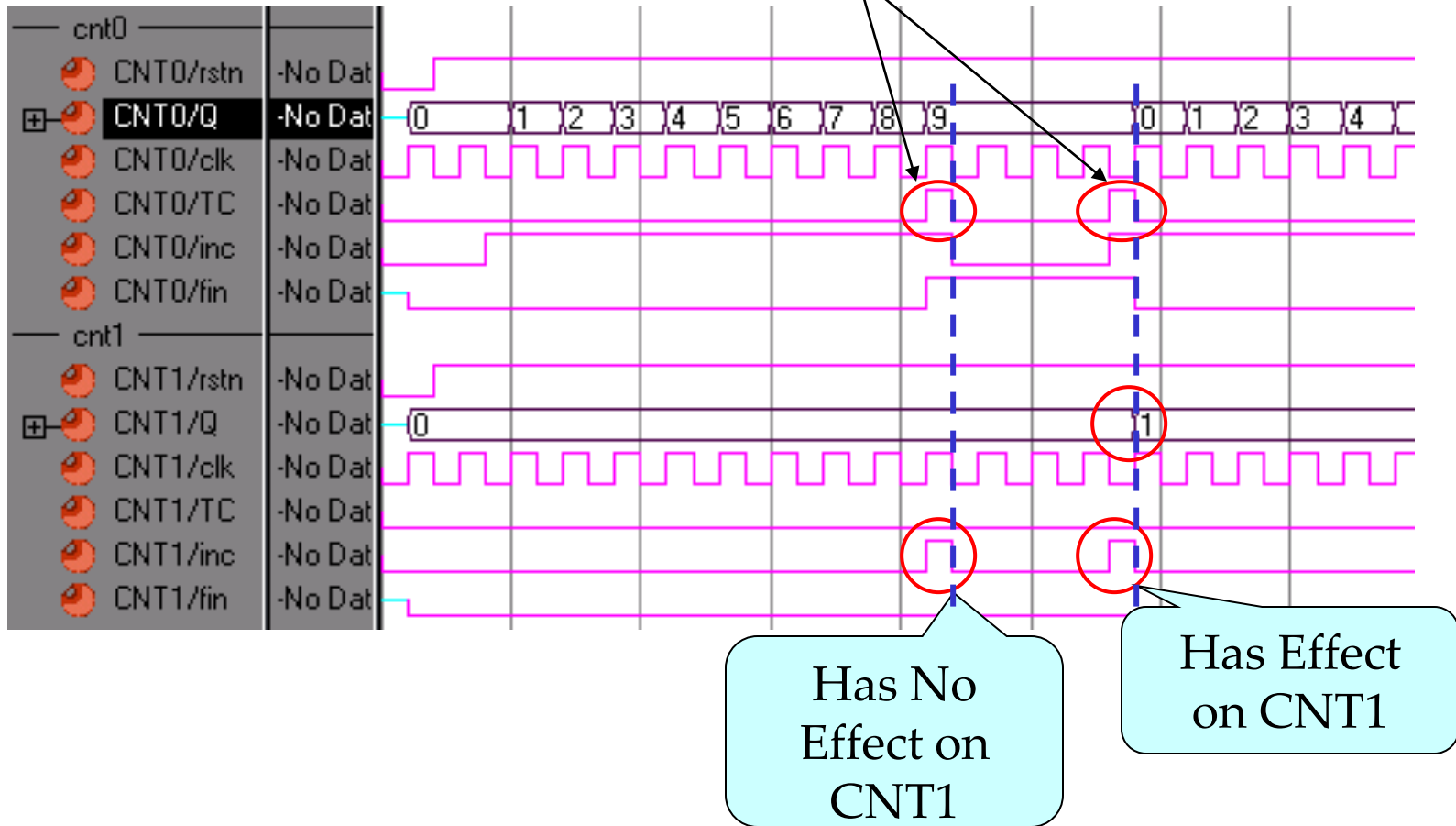
endmodule
    
```



Glitch(fin & inc)

# Closer Look at the Waveform

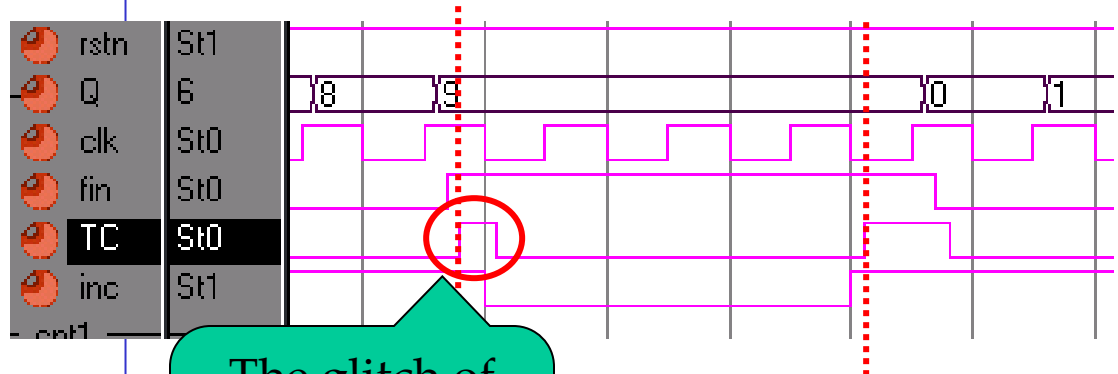
TC = fin & inc



# If Circuit Delay is Considered ...

```
module BCDcnt (Q,TC, inc, rstn, clk);  
parameter Delay = 1;  
output [3:0] Q ;  
output TC;  
input inc, rstn, clk;  
reg [3:0] Q;  
wire fin;
```

```
always @(posedge clk)  
  if(!rstn) Q <= #Delay 4'd0;  
  else if (inc)  
    begin  
      if(fin) Q <= #Delay 4'd0;  
      else Q <= #Delay Q + 1;  
    end  
  assign #Delay fin = (Q == 4'd9)? 1 : 0 ;  
  assign #Delay TC = fin & inc ;  
endmodule
```



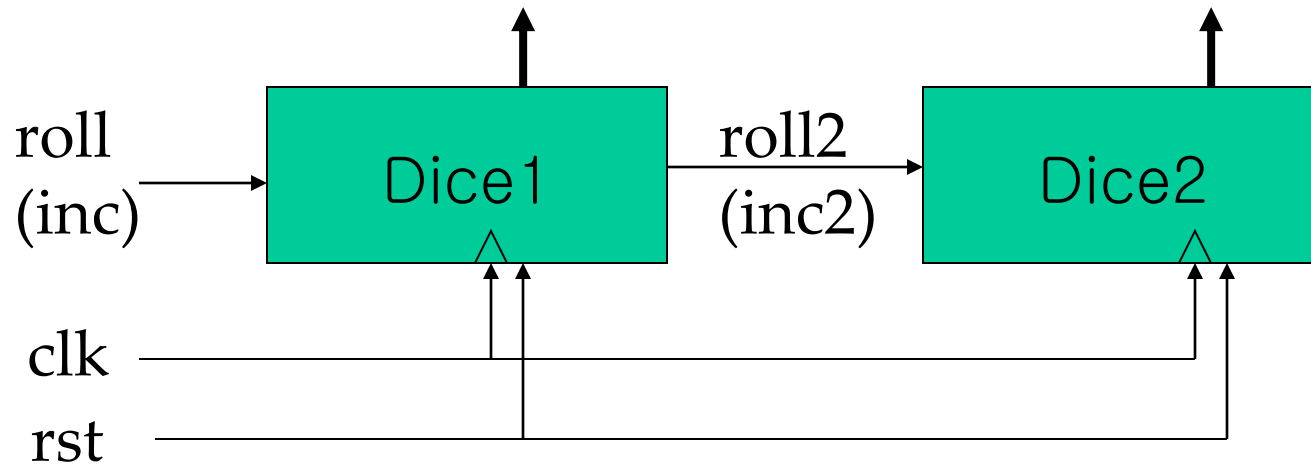
The glitch of  
TC signal has  
no Effect  
Clearly

- Explicit Delay Specification Clarifies the Precedence between signals (Delay helps to figure out which one precedes the other)
  - It Makes Clear if a glitch is harmless or not (effective)

Note : Signal Dependency is clk -> Q -> fin -> TC

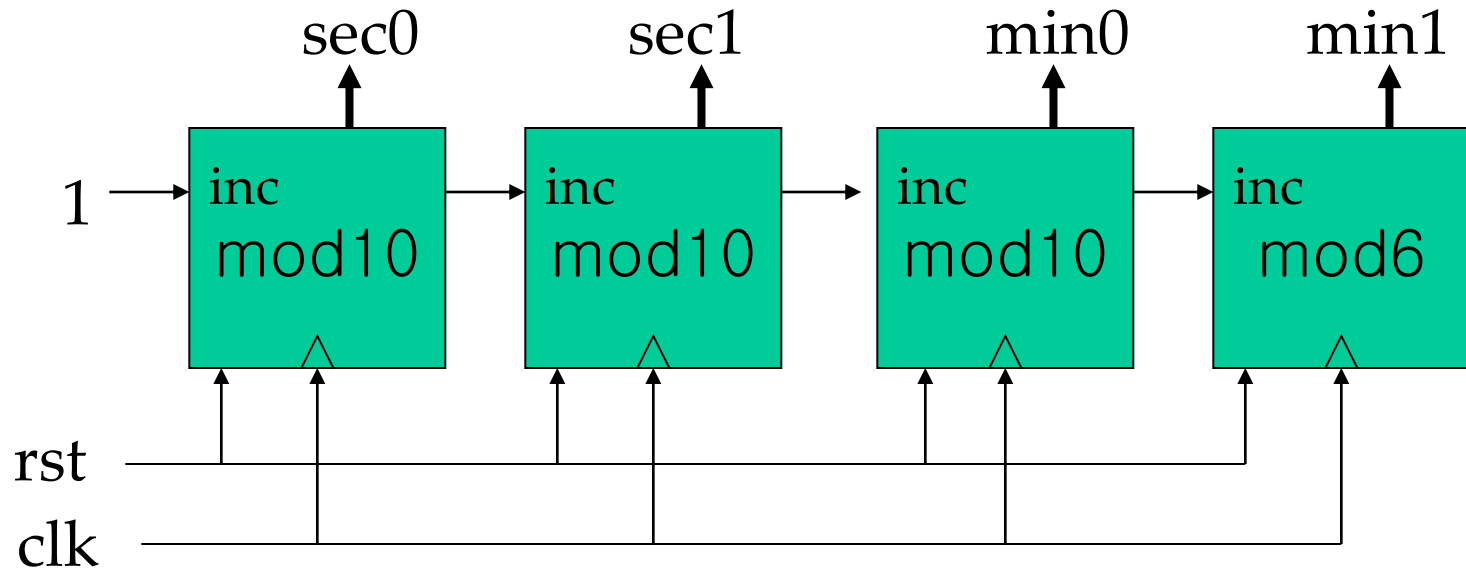
# Exercises (1)

1. Design a Dice (mod-6 counter) that counts from 1 to 6
2. Cascade the two Dice designed above Question . If one dice rolls 6 times the next dice increments by one.



# Exercises(2)

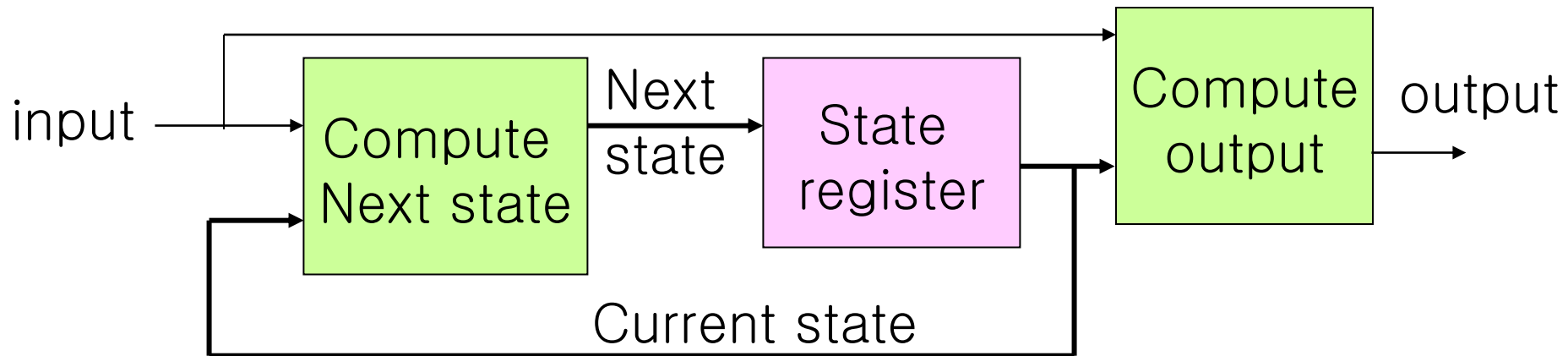
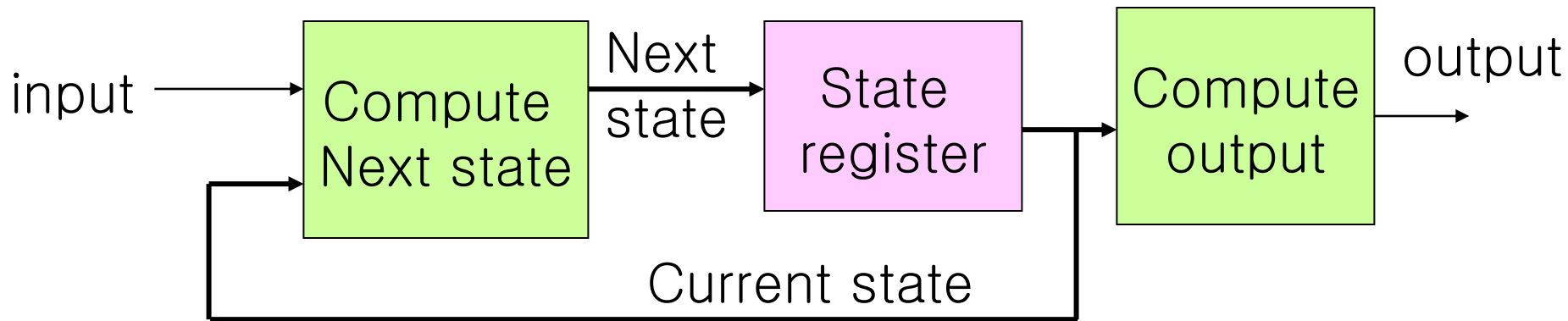
3. Design a clock circuit which has 2 digit for second and 2 digits for minute. Design the logic in a synchronous fashion (NOT Asynchronous manner). Assume that the input clock is 25MHz.



# FSM Design

# FSM design

- Moore output =  $f(\text{current\_state})$
- Mealy output =  $f(\text{current\_state}, \text{input})$



# FSM coding with Verilog

- State Register with Asynchronous Reset

```
always @(posedge clk or posedge rst)
begin
    if(rst == 1)
        current_state <= `INIT_STATE;
    else
        current_state <= next_state;
end
```

- Combinational Logic Computing Next State

```
always @(current_state or inputs)
case (current_state)
`INIT_SATE : if (inputs == 1 )
                ns = `WAIT_STATE ;
            else ns = `INIT_STATE;
`WAIT_STATE : ...
`RUN_STATE:    ...
default : ns <= `INIT_STATE
endcase
```



# 2 Types Synchronous FSM State Register

- Synchronous Reset logic in next\_state computation circuit

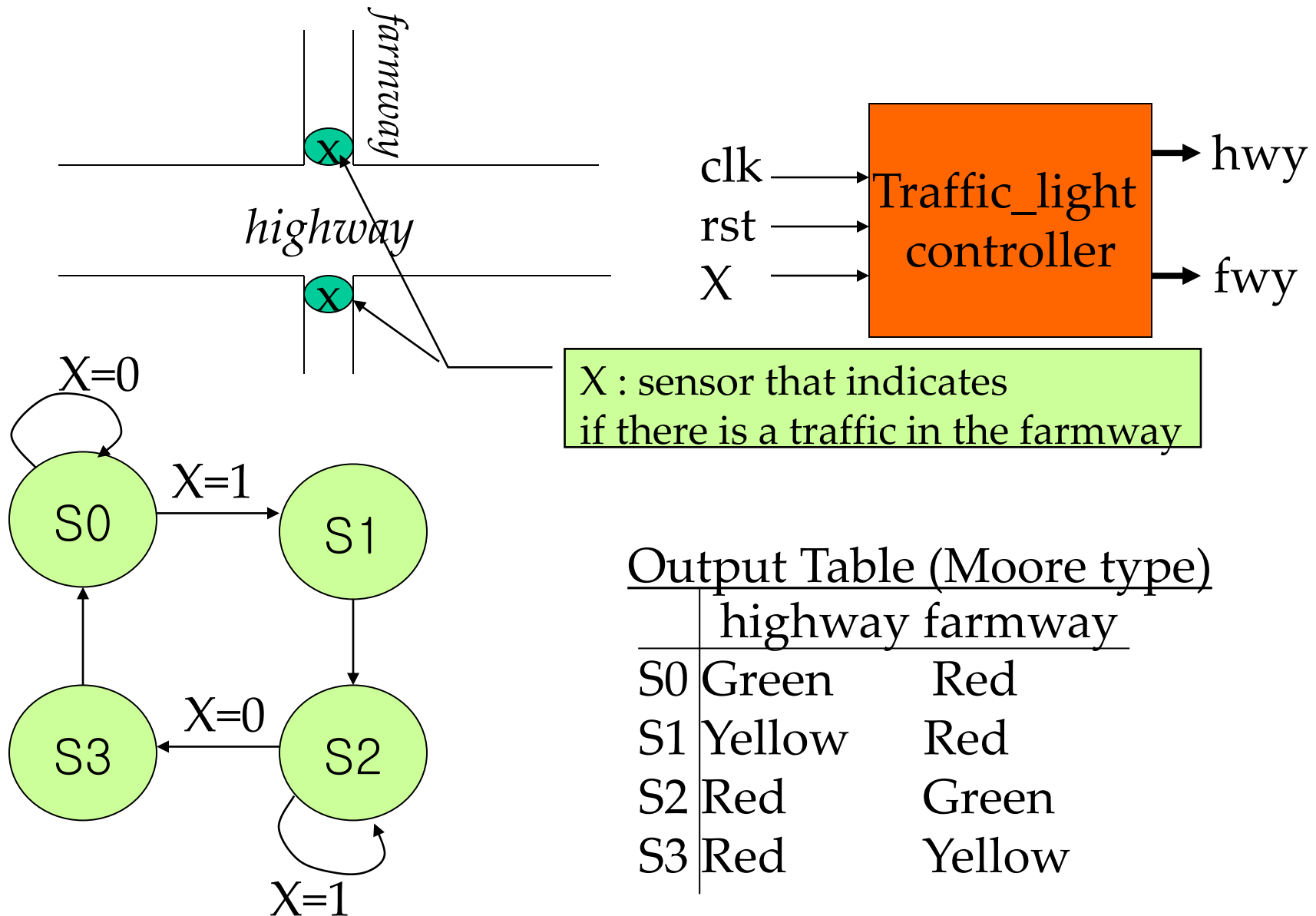
```
always @(posedge clk )
    current_state <= next_state;

always @(current_state or inputs or reset) begin
    if(reset == 1) next_state <= `INIT_STATE;
    else case (current_state)
        `INIT_STATE :
            if(inputs == 1)
                next_state = `WAIT_SATE;
            else next_state = `INIT_STATE;
        `WAIT_STATE : ...
        `RUN_STATE : ...
    endcase
end
```

- Synchronous Reset in the State Register

```
always @(posedge clk)
    if(reset == 1)
        current_state <= `INIT_STATE;
    else
        current_state <= next_state;
```

# Traffic Light Controller Example



# Traffic Light Controller Verilog Source

```
// traffic light definition
`define YELLOW 2'd0
`define RED     2'd1
`define GREEN   2'd2
// state assignment definition
`define S0 2'b00
`define S1 2'b01
`define S2 2'b10
`define S3 2'b11

module sig_controller
    (hwy,fwy, X, clk,rst) ;

output [1:0] hwy, fwy;
reg [1:0] hwy, fwy ;
input X, clk, rst ;

reg [1:0] cs, ns ; // state variable
```

```
Next state {
    always @(posedge clk or posedge rst)
        if(rst) cs <= `S0;
        else cs <= ns;

    always @(cs or x) case (cs)
        `S0 : if(X) ns <= `S1; else ns <= `S0;
        `S1 : ns <= `S2;
        `S2 : if(X) ns <= `S2; else ns <= `S3;
        `S3 : ns <= `S0;
    endcase
}

FSM Output {
    always @(cs)
        case (cs)
        `S0 : begin hwy = `GREEN; fwy = `RED; end
        `S1 : begin hwy = `YELLOW; fwy = `RED; end
        `S2 : begin hwy = `RED; fwy = `GREEN; end
        `S3 : begin hwy = `RED; fwy = `YELLOW; end
    endcase
}

endmodule
```

# Traffic Light Controller Verilog Source (Synchronous version)

```
module sig_controller
    (hwy,fwy, X, clk,rst) ;

    output [1:0] hwy, fwy;
    reg [1:0] hwy, fwy ;
    input X, clk, rst ;

    reg [1:0] cs, ns ; // state variable

    always @(cs)
        case cs
            `S0 : hwy = `GREEN; fwy = `RED;
            `S1 : hwy = `YELLOW; fwy = `RED;
            `S2 : hwy = `RED; fwy = `GREEN;
            `S3 : hwy = `RED; fwy = `YELLOW;
        endcase
```

```
    always @(posedge clk or posedge rst)
        cs <= ns;

    always @(cs or x or rst)
        if(rst)
            ns <= `S0;
        else
            case cs
                `S0 : if(X) ns <= `S1; else ns <= `S0;
                `S1 : ns <= `S2;
                `S2 : if(X) ns <= `S2; else ns <= `S3;
                `S3 : ns <= `S0;
            endcase
    endmodule
```

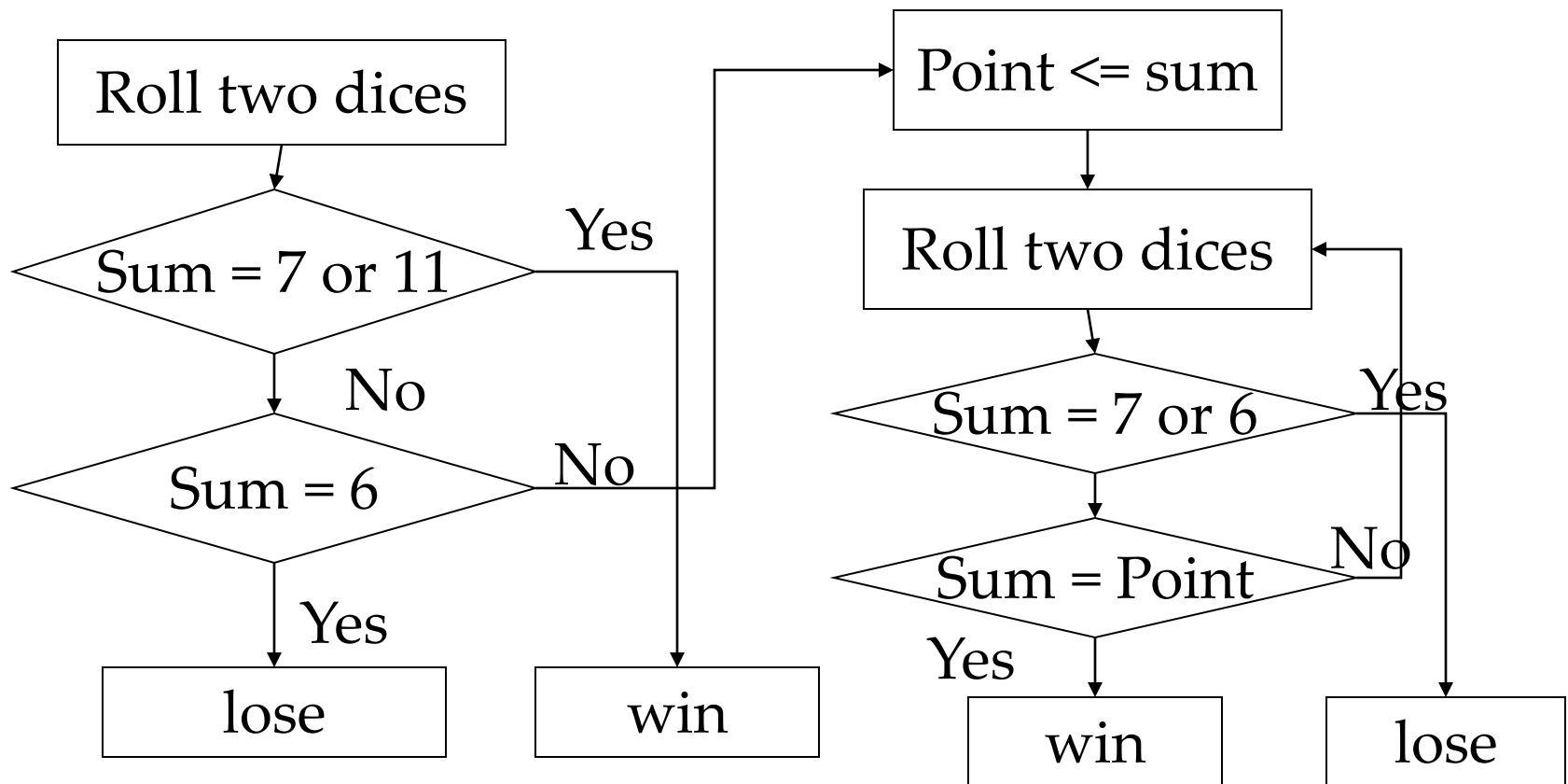
# Testbench

```
module testbench;
reg clk = 0;
reg rst ,X ;
wire [1:0] hwy, fwy;

module sig_controller (hwy,fwy, X, clk,rst) ;
initial forever #5 clk = ~ clk; // or always # clk = ~ clk;
initial begin
    rst = 1;
    #10 rst = 0; X = 1;
    #40 X = 0;
    #20 X = 1;
    #10 rst = 1;
    #10 rst = 0;
    #20 $stop;
end
endmodule
```

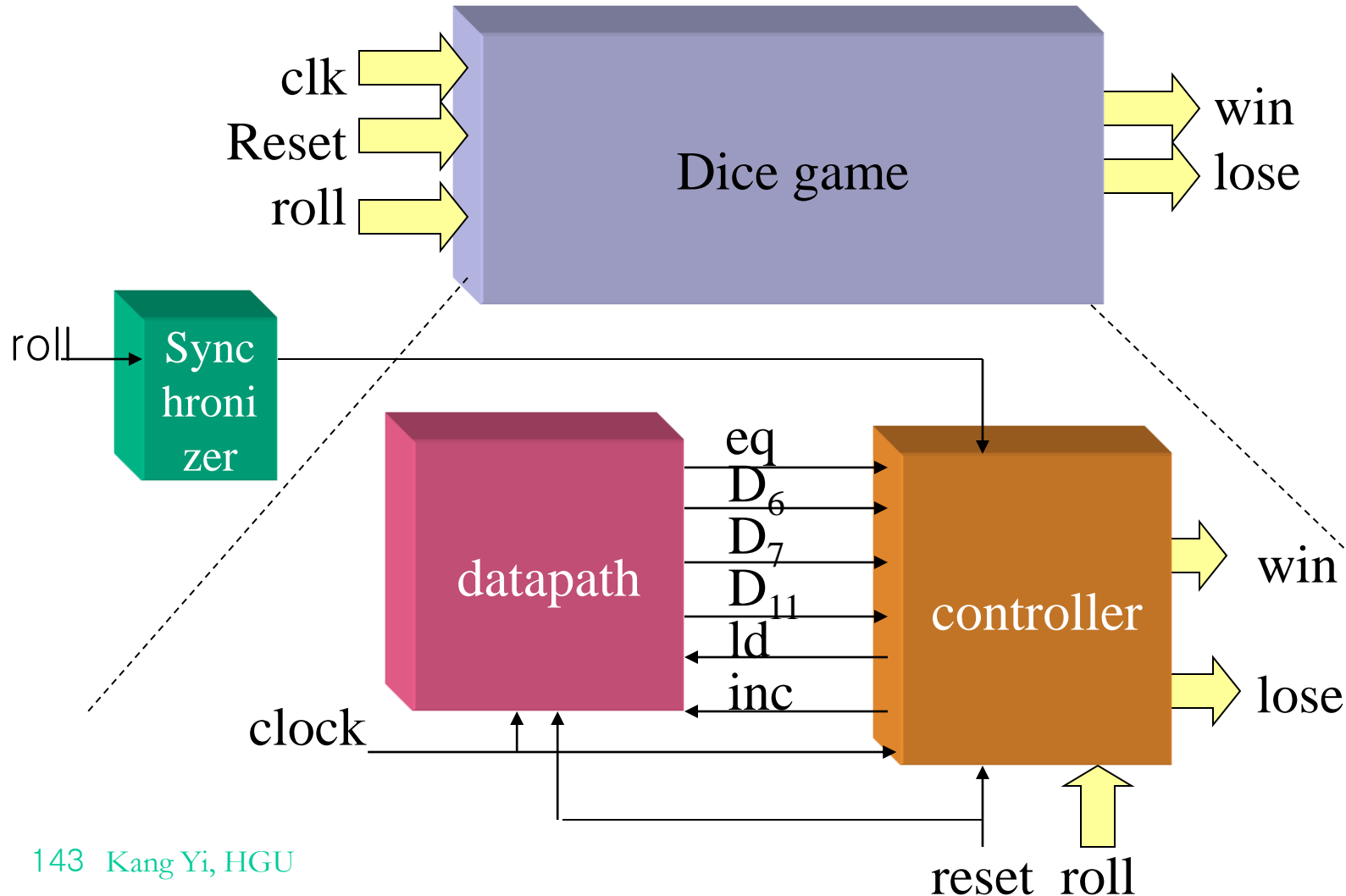
# Exercises

1. Design a Dice Game Machine. The game is played with 2 dices. The controller inputs are X, clock, and reset. The outputs are Win, Lose, Dice0[2:0], and Dice1[1:0]. The Flowchart for the game is as follows.



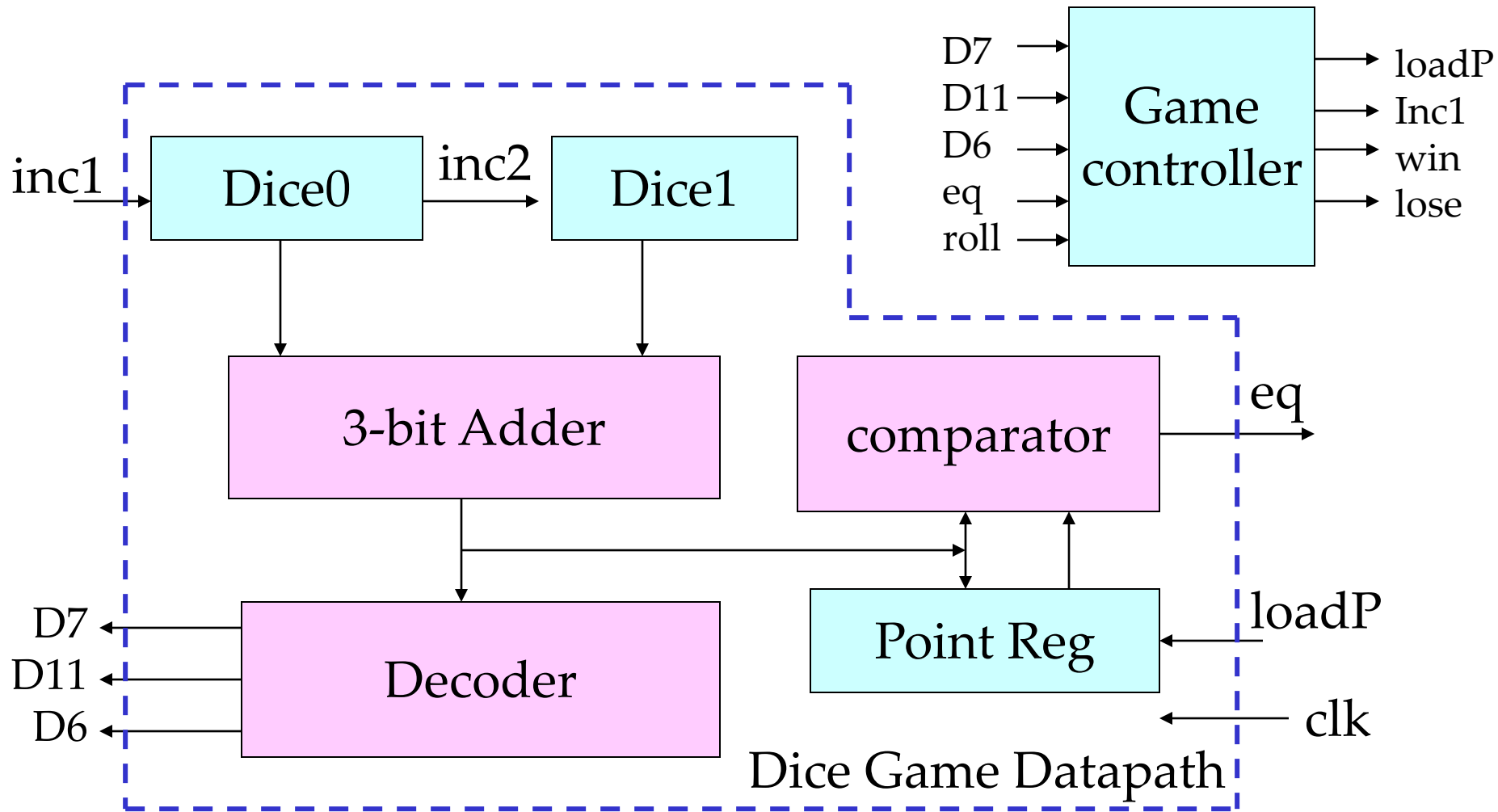
# Hints for the Ans (1)

- The Top-Level Submodules



# Hints for the Ans (2)

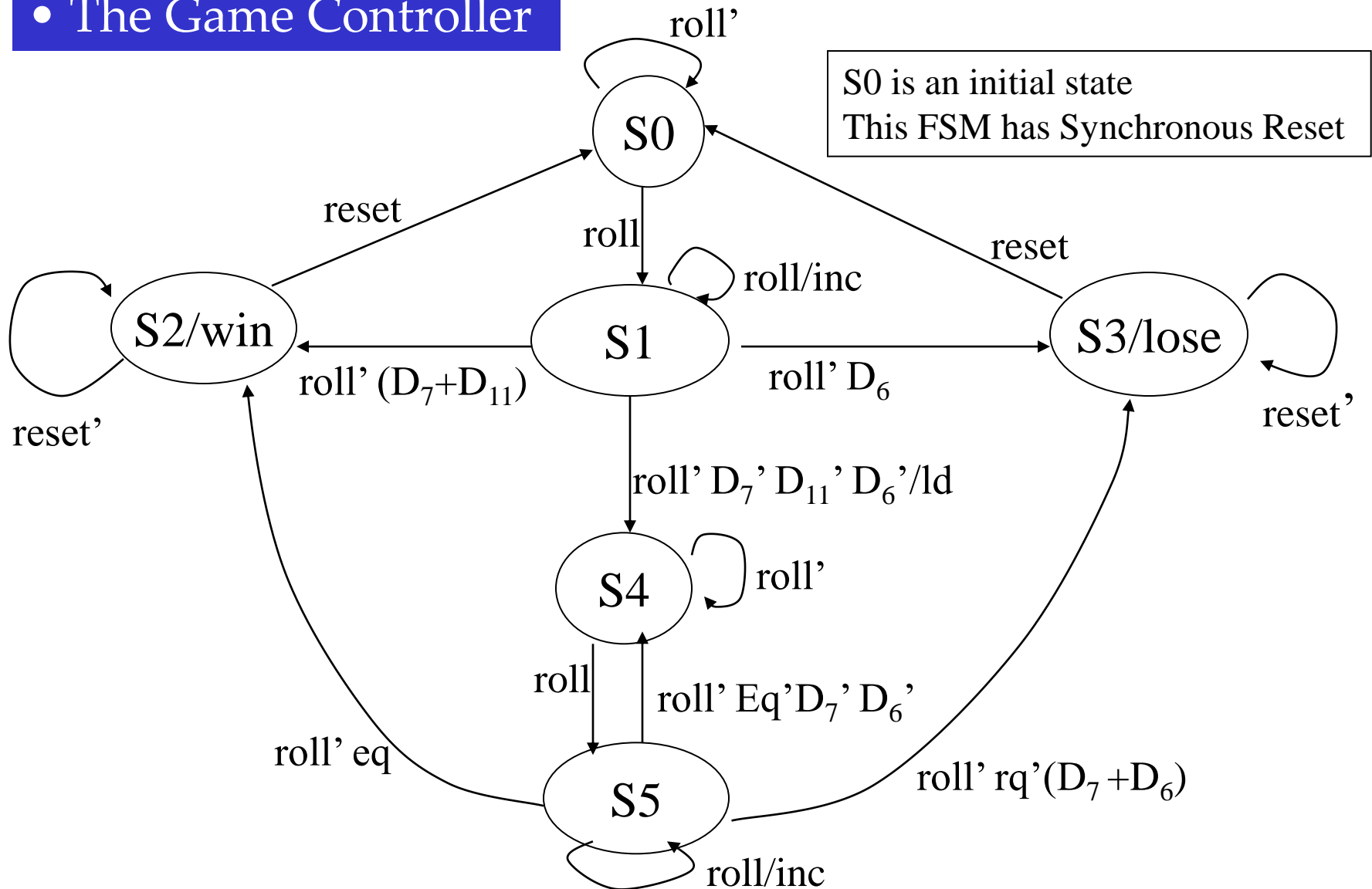
- The Datapath and Control Signals





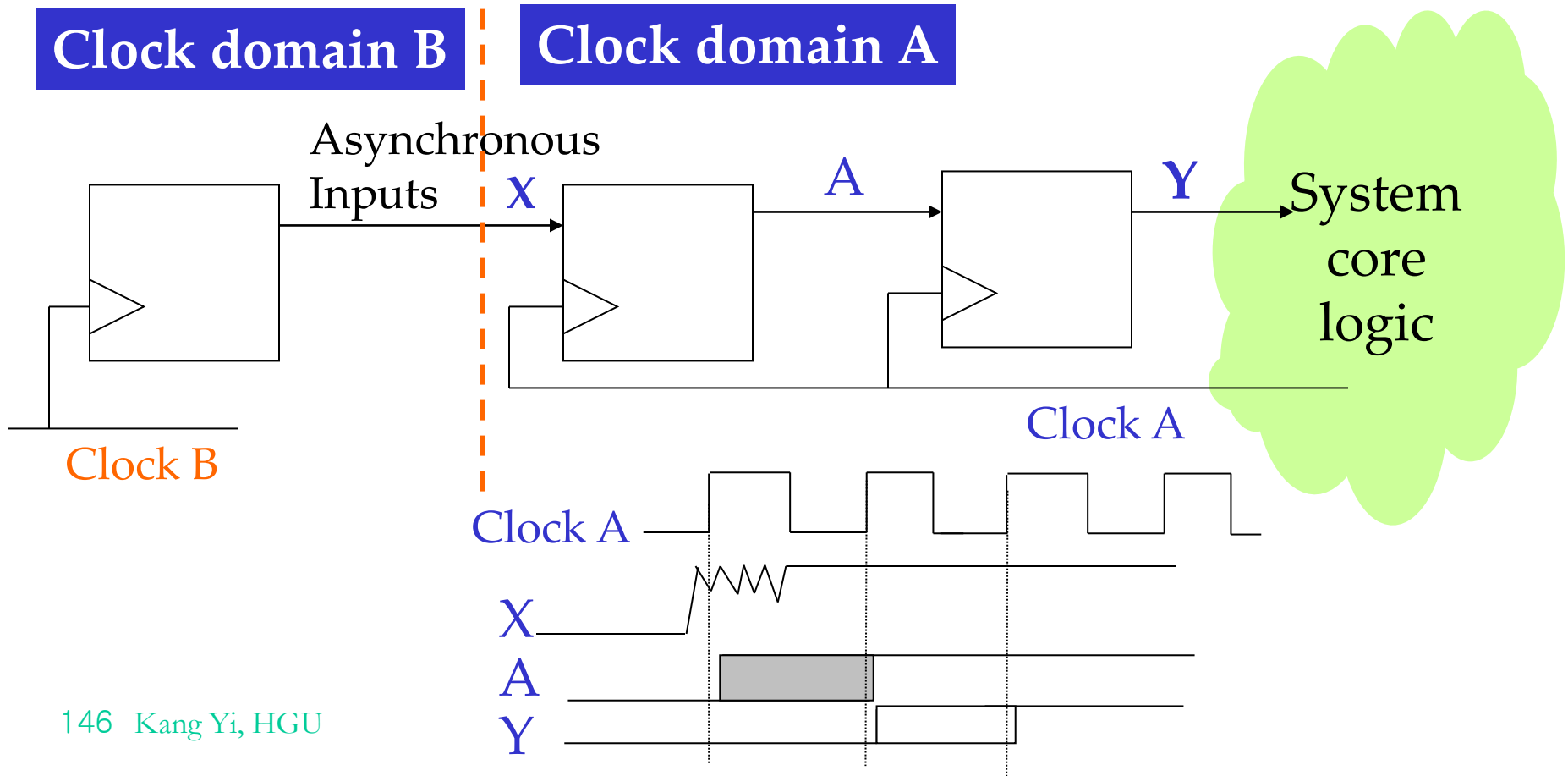
# Hints for the Ans (3)

- The Game Controller



# Hints (Synchronizer Design)

- Sometimes Asynchronous Inputs are unavoidable (e.g. User Inputs: roll)
- Asynchronous Inputs : Signals from different Clock Domain
- Synchronizer Objectives : Synchronization & DeBouncing



# Parameterized Design

- Parameter Value Change by Defparam
- Parameter value change at Module instance
- Conditional Compilation

# Parameter Value Change

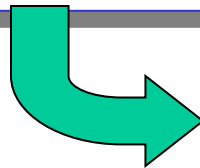
- Parameters are defined in a module with *predefined value*
- Parameter values can be *overridden* in any module instance during compilation time by 2 way
  - *defparam* statement
  - module instance parameter value assignment

# defparam

- In defparam statements use hierarchical name of module instance to override the parameter values

```
module hello_world ;  
    parameter id_num= 0;  
    initial $display ("Hello_world id num = %d", id_num) ;  
endmodule
```

```
module top ;  
    defparam w1.id_num = 1, w2.id_num = 2;  
    Hello_world w1 ();  
    Hello_world w2 ();  
endmodule
```



```
Hello_world id num = 1  
Hello_world id num = 2
```

# Module Instance Parameter Values

- New parameter values are passed during module instantiation

```
module hello_world ;  
parameter id_num= 0;  
initial $display ("Hello_world id num = %d", id_num) ;  
endmodule
```

```
module top ;  
Hello_world #(1) w1 ();  
Hello_world #(2) w2 ();  
endmodule
```

- Multiple parameters can be override by specifying new values in the same order as the parameter declarations the in module

```
module bus_master ;  
parameter delay1 = 2;  
Parameter delay2 = 3 ;  
endmodule
```

```
Module top;  
bus_master #(4, 5) b1 ();  
bus_master #(9) b2 ; // delay2=3
```

# Compiler Directives : Conditional Compilation

- ``ifdef` ``else` ``endif`
- A portion of Verilog might be suitable for one environments and not for other.

```
`ifdef TEST
module test ;
.....
endmodule;

`else
module stimulus ;
.....
endmodule
`endif
```

# Compiler Directives : Conditional Execution

- All statements are compiled but executed conditionally
- Use System task **\$test\$pulsargs** to check if a flag is set during the run-time (in Verilog-XL simulator)

```
module test ;  
.....  
initial begin  
    if($test$pulsargs("DISPLAY_VAR"))  
        $display ("Display = %b", {a,b,c});  
    else  
        $display("No Display");  
end  
endmodule
```



# More on Blocks

- Parallel blocks
- Named blocks
- Nested blocks

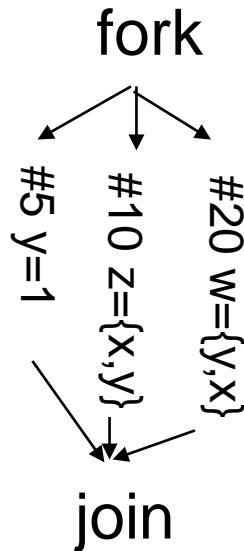
# Block Types

- Block is used to group multiple statements
  - Begin ... end
  - Fork ... join
- Sequential blocks
  - Keywords **begin** and **end** are used
  - Statements in a block execute in the order listed *one after another*.
  - delay or event control is *relative to* the time when the *previous statement completed* its execution
- Parallel blocks
  - Keywords **fork** and **join** are used
  - Ordering of statements is controlled by delay or event control
  - Delay or Event control is *relative to* the time the block was entered

# Parallel Block : Fork and Join

- Parallel Block is Surrounded by **fork** and **join**
- All Statements in a Parallel Block *Starts at the Same Time* when the Block was Entered. => The *order* of Statements in the Block is Not Important.

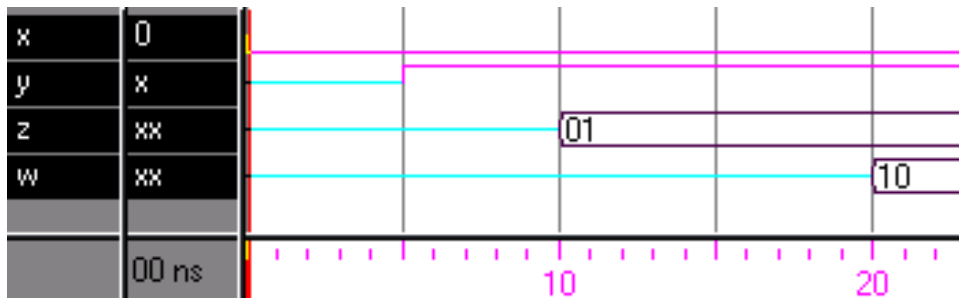
```
reg x, y;
reg [1:0] z, w;
initial
fork  x = 0;
      #5 y = 1;
      #10 z = {x, y};
      #20 w = {y, x};
join
```



Race Condition

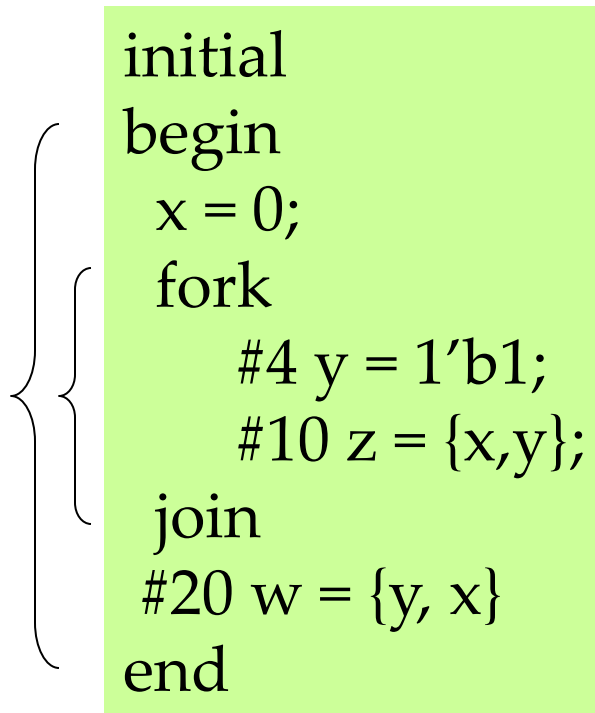
```
reg x, y;
reg [1:0] z, w;
initial
fork  x = 0;
      y = 1;
      z = {x, y};
      w = {y, x};
join
```

$z = 01$  or  $xx$   
 $w = 10$  or  $xx$



# Nested Blocks

- Blocks can be nested. Sequential and Parallel Blocks can be mixed



```
initial
begin
  x = 0;
  fork
    #4 y = 1'b1;
    #10 z = {x,y};
  join
  #20 w = {y, x}
end
```

The image shows a VHDL code snippet on a light green background. The code is: `initial`, `begin`, `x = 0;`, `fork`, `#4 y = 1'b1;`, `#10 z = {x,y};`, `join`, `#20 w = {y, x}`, `end`. To the left of the code, there are curly braces. A large brace groups the entire block from `begin` to `end`. A smaller brace groups the `fork` block, which includes the two delay-based assignment statements. Another brace is positioned to the left of the `join` statement, indicating its relationship to the `fork` block.

# Named Blocks

- Blocks can be Named
  - Local Variables can be Declared for the Named Block
  - Named Blocks are a Part of Design Hierarchy
  - Named Blocks can be disabled (The Execution can be stopped)

```
module top ;  
initial  
begin : block1 // sequential block named as block1  
    integer i ; // integer i is static & local to block1 (top.block1.i)  
    ...  
end  
  
initial  
fork : block2 // parallel block named as block2  
    reg I ; // reg I is static & local to block2 (top.block2.i)  
    ...  
join
```

# Disabling Named Blocks

- **disable**

- Terminates the Execution of a *any Named Block*
- Is Similar to *break* statement in C
- Disabling a Named Block cause the Control to be Passed to the Statement Immediately Succeeding the Block
- Used to get out of Loops, handle Error Condition, etc.

```
reg [3:0] flag ; integer i;  
initial begin  
    flag = 4'b1000;  
    i = 0;  
    begin : block1  
        while (i < 4) begin  
            if (flag[i]) disable block1;  
            i = i + 1;  
        end  
    end  
end
```

# Tasks & Functions

- Tasks
- Functions
- Useful System Tasks

# Common features

- Similar to SUBROUTINE and FUNCTIONS
- Both Tasks and Functions are defined **in a module** and are **local to the module**
- Both cannot declare *wires* local to the function or tasks
- Both contain behavioral statements only
- Both do not contain *always* or *initial* procedure blocks
- Both do not contain other functions or tasks



# Differences between Functions and Tasks

Functions	Tasks
Enables another functions, but not another tasks	Enable other tasks and functions
Execute in 0 simulation time	Execute in non-zero simulation time
Cannot contain any delay, event, or timing control statements	Can contain delay, events, or timing control statements
At least one input argument (only input)	May have 0 argument of type input, output, or inout
Always return a single value	Do not return a value

# When Tasks ?

- Tasks
  - Can Have Delay, timing, or event control constructs
  - Zero or more than one output arguments
  - No input arguments
- *Output and inout argument values are passed back to the variables in the task invocation statement when the task is completed.*

# Tasks Declaration Syntax

```
task <name_of_task> ;  
    <task_declaration>  
    <statements>  
endtask
```

- <task\_declaration> may contain
  - Parameter Declaration
  - I/O arguments declarations (input, output, or inout declaration)
  - variable declarations (reg, time, integer, or real declaration)
  - Event declarations
- Example)

```
task init_sequence ;  
begin  
    clock = 1'b0;  
end  
endtask
```

# Task Invocation Syntax

<name\_of\_task> ;

*or*

<name\_of\_task> ( <parameter\_list> ) ;

- (Example)
  1. initiate\_sequences ;
  2. Compute\_values (output\_val, input\_val1, input\_val2) ;

# Task example (1)

module operation ;

.....

parameter delay = 10;

reg [15:0] A,B, AB\_AND, AB\_OR, AB\_XOR;

always @(A or B)

*bit\_operator*(AB\_AND, AB\_OR, AB\_XOR,A,B) ; // task invoking } invocation

.....

**task** *bit\_operator* ; // task definition

output [15:0] ab\_and, ab\_or, ab\_xor;

input [15:0] a, b ;

begin

#delay ab\_and = a & b;

ab\_or = a | b ;

ab\_xor = a ^ b ;

end

**endtask**

.....

endmodule

definition

# Task Example (2)

```
module sequence_gen ;
```

```
.....
```

```
reg clock ;
```

```
.....
```

```
initial
```

```
    init_sequence ; // task invocation
```

```
always
```

```
    asymmetric_sequence ; // task invocation
```

```
.....
```

```
task init_sequence ; // task definition 1
```

```
    clock = 1'b0 ;
```

```
endtask
```

```
task asymmetric_sequence ; // task definition 2
```

```
begin
```

```
    #12 clock = 1'b0; #5 clock = 1'b1 ; #3 clock = 1'b0 ; #10 clock = 1'b1 ;
```

```
end
```

```
endtask
```

```
endmodule
```

Note : the scope of variables in module Reaches into tasks in the module

# When Function ?

- Functions
  - Have No delay, No Timing Control, No Event Control inside function
  - Returns a single value
  - Have At least one argument
- Typically Functions are Used for Combinational Function Computation

# Function Declaration Syntax

```
function [<return_range_or_type>] <function_name> ;  
<function_declaration>  
<statement>  
endfunction
```

- <return\_range\_or\_type> is either
  - <INTEGR> or <REAL> or <vector\_range>
- <function\_declaration> may contain
  - parameter\_declaration
  - input\_declaration
  - variable\_declarations (reg, time, integer, real type declaration)



# Function Invocation Syntax

- `<function_name> (<arguments_list>) ;`
- (Example)
  - `result = compute_a_value (a, b, c) ;`

# Function Example (1)

```
module parity ;
```

```
reg [31:0] addr;
```

```
reg parity ;
```

```
...
```

```
always @(addr)
```

invocation { parity = calc\_parity (addr) ; // function invocation

```
.....
```

```
function calc_parity ; // function declaration
```

```
input [31:0] address;
```

definition { calc\_parity = ^ address;

```
endfunction
```

```
.....
```

```
endmodule
```

# Function Example (2)

```
module shifter ;  
reg [31:0] addr, l_addr, r_addr ;  
reg control ;
```

```
.....
```

invocation {

```
    always @(addr) begin  
        l_addr = shift(addr, 0); // function invocation  
        r_addr = shift(addr, 1 ); // function invocation  
    end
```

```
.....
```

definition {

```
    function [31:0] shift ;  
    input [31:0] address ;  
    input direction ;  
    shift = (control == 0) ? (address << 1) : (address >> 1);  
    endfunction
```

```
.....
```

```
endmodule
```

# Exercises

1. Define a *function* that computes factorial for a 4-bit number. Output is 32-bit number.
2. Define a *task* that computes factorial for a 4-bit number. The output is 32-bit value that is assigned to the output after delay of 10 time units.
3. Define a *function* to multiply two 4-bit numbers a and b. The output is 8-bit value.

# Answers

```
// ans to the question 1.  
function [31:0] factorial ;  
input [3:0] a;  
integer i;  
reg [31:0] val ;  
begin  
    val = 1;  
    for(i = 1 ; i <= a; i = i + 1)  
        val = val * i;  
    factorial = val;  
end  
endfunction
```

```
// ans to the question 2.  
task factorial ;  
output [31:0] res ;  
input [3:0] a;  
integer i;  
begin  
    res = 1;  
    for(i = 1 ; i <= a; i = i + 1)  
        res = res * i ;  
end  
endtask
```

# Useful System tasks

- Simulation Stop/Suspend
- File I/O
- Displaying Hierarchy
- Strobing
- Random number generation
- Initializing Memory from File
- Value Change Dump File

# Opening File for Output

- **\$fopen**
  - Usage : **\$fopen**("<filename>");
  - Usage : <file\_handler> = **\$fopen**("<filename>");
  - <file\_handler> : 32-bit integer value called *multichannel descriptor*
- Multichannel descirptor
  - each successive call to **\$fopen** return *multichannel descriptor* with only one bit set (with bit 1 set, bit 2 set, ..., upto bit 31 set)
  - So, It is possible to write to Multiple Files at the Same Time selectively
  - *Multichannel descriptor* for Standard Output : 0000...01 (bit 0 set)

```
integer handle1, handle2, handle3 ;
initial begin
    handle1 = $fopen("file1.out");
    handle2 = $fopen("file2.out");
    handle3 = $fopen("file3.out");
end
```

# Writing to Files

- Tasks for writing
  - `$fdisplay`, `$fmonitor`, `$fstrobe`, and `$fwrite`
  - Usage : **`$fdisplay`** (<file\_descriptor>, p1, p2, ..., pn)
  - Usage : **`$fmonitor`** (<file\_descriptor>, p1, p2, ... , pn)

```
integer desc1, desc2, desc3 ;
initial begin
    desc1 = handle1 | 1 ; // file1.out + stdout
    $fdisplay(desc1, "Display 1");

    desc2 = handle1 | hndle2 ; // file1.out + file2.out
    $fdisplay(desc2, "Display 2");

    desc3 = handle3 ;
    $fdisplay (desc3, "Display 3");
end
```



# Closing Files

- Files can be closed by `fclose` system task call with `file_handler`
- Usage : `$fclose (handle1);`
- A file cannot be written to once closed
- The corresponding bit in multichannel descriptor is set to 0. The next call to `$fopen` will reuse the bit

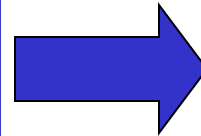
```
// closing file  
$fclose(handle1); // closing file1.out  
$fclose(handle2); // closing file2.out
```

# Displaying Hierarchy

- Using `%m` option in `$display`, `$write`, `$monitor`, or `$strobe` task arguments
- When multiple instances are executing the same time, `%m` is very useful to distinguish where the output comes from.

```
// just display hierarchy information
Module M;
initial $display ("Displaying in %m");
endmodule;

module top;
M m1 ();
M m2 ();
M m3 ();
endmodule
```




```
Displaying in top.m1
Displaying in top.m2
Displaying in top.m3
```

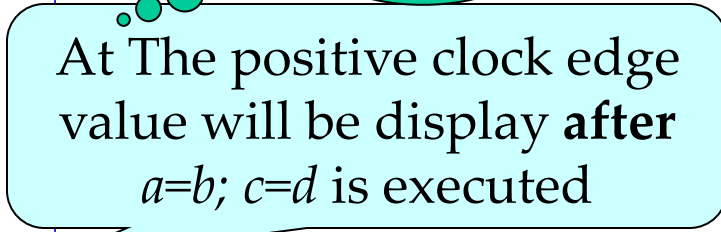
# \$Strobe

- **\$strobe** system task
  - Very similar to **\$display** task
  - **\$display** task has Nondeterministic order If many other statements are executed in the Same Time unit
  - **\$strobe** is *always* executed *after all other assignments in the same time unit are executed*. Thus, \$strobe provides means to **ensure** that the data is displayed with correct value after assignments are done

```
always @(posedge clk)
begin
    a = b;
    c = d;
end
always @(posedge clk)
    $strobe ("Displaying a=%b c=%b",a,c);
```



\$display might  
execute **before** any  
assignments



At The positive clock edge  
value will be display **after**  
*a=b; c=d* is executed

# Random Number Generation

- RNG is a Very Important task in Design verification & performance analysis
- Usage : **\$random** ; *or* **\$random** (<seed>);
  - <seed> is an optional number
  - Returns 32-bit number

```
module RNGtest;
integer seed;
reg [31:0] addr; // input to ROM
wire [31:0] data ; // output from ROM

ROM rom1 (data,addr);
initial seed = 2;
always @(posedge clk)
    addr = $random(seed);
// check if ROM output is correct against expected value
endmodule
```

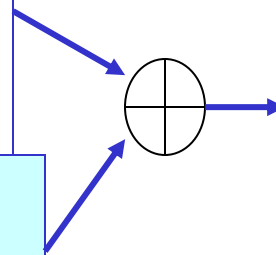
# Initializing Memory from File

- Initializing the contents of Memory in Verilog
- 2 System Tasks : **\$readmemb** (binary) , **\$readmemh** (hexadecimal)
- **\$readmemb** / **\$readmemh** Usage :
  - **\$readmemb** ("**<file\_name>**", **<memory\_name>**);
  - **\$readmemb** ("**<file\_name>**", **<memory\_name>**, **<start\_addr>**);
  - **\$readmemb** ("**<file\_name>**", **<memory\_name>**, **<start\_addr>**, **<finish\_addr>**);

```
module ram_test;  
  reg [7:0] mem [0:14] ; integer i;  
  initial begin  
    $readmemb ("init.dat", mem);  
    for(i=0; i<15 ; i = i + 1)  
      $display("%d = %b",i,mem[i]);  
  end  
endmodule
```

"init.dat" File →

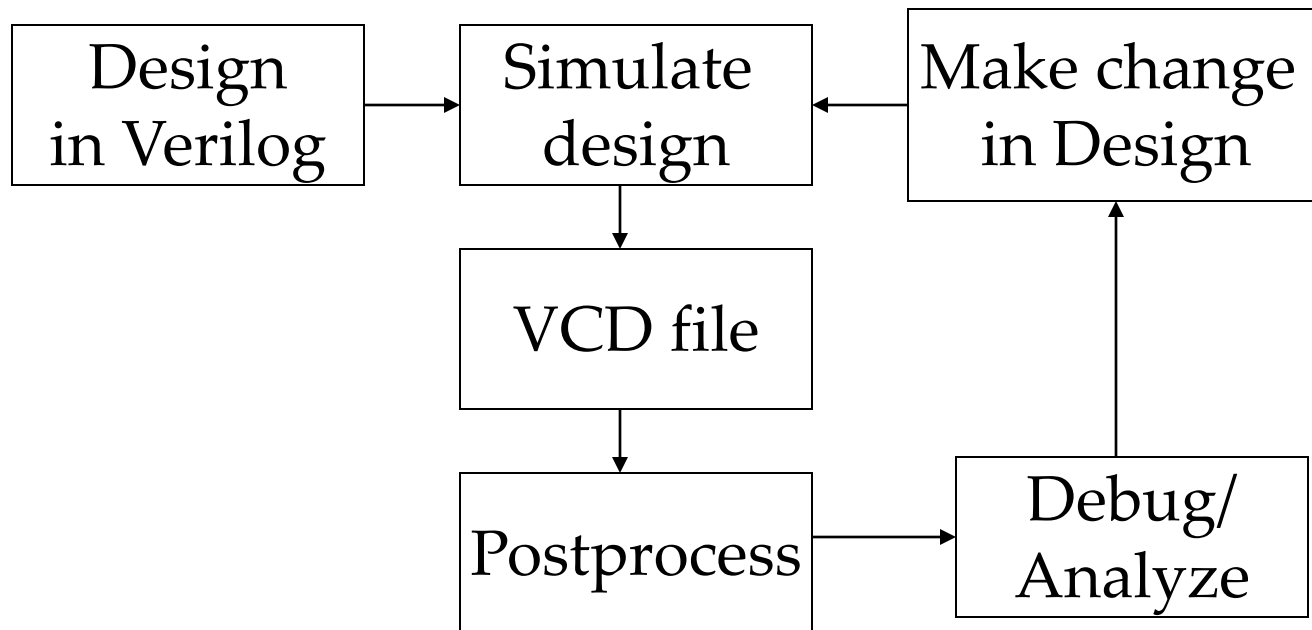
```
@002  
11111111 01010101  
00000000  
@006  
1111zzzz 00001111
```



```
M[0] =xxxxxxx  
M[1] = xxxxxxx  
M[2] = 11111111  
M[3] = 01010101  
M[4] = 00000000  
M[5] = xxxxxxx  
M[6] = 1111zzzz  
M[7] = 00001111  
M[8] = xxxxxxx  
M[9] = xxxxxxx  
M[10] = xxxxxxx  
M[11] = xxxxxxx  
M[12] = xxxxxxx  
M[13] = xxxxxxx  
M[14] = xxxxxxx
```

# Value Change Dump (VCD) File

- VCD (Value Change Dump) is an ASCII file that contains information about simulation time, scope, signal definition, signal value changes while simulator runs
- VCD + Post Process Tools is Very Useful for Simulation Result Analyzing
- Graphical Tools are Provided for VCD Post Process
- Very Large file size (100s of Mbytes) : selectively dump



# VCD Related System Tasks

- Related System Tasks : \$dumpfile, \$dumpvars, \$dumpall, \$dumpon, \$dumpoff
- **\$dumpfile** : Assign the Name of VCD file
- **\$dumpvars** : selects module instances or signals to dump
- **\$dumpon / \$dumpoff** : start and stop dump process
- **\$dumpall** : generate check points

# VCD Example

```
// specify the name of dump file. Otherwise default name is assigned
initial $dumpfile ("dump.out"); // dump simulation info into dump.out

// Dump signals in a module
initial $dumpvars ; // dump all arguments
initial $dumpvars(1,top) ; // dump variables in module instance top
    // number 1 indicates the level of hierarchy (1 level below top)
    // signals in top module, but signals in the modules instantiated by top)
initial $dumpvars (2, top.m1) ; // dump upto 2 levels below top.m1
initial $dumpvars (0, top.m1); // 0 means entire hierarchy below top.m1

initial begin
    $dumpon ; // begin dump process
    #10000 $dumpoff; // stop dump process after 10000 time units
end

// create checkpoints. Dump all current value of all VCD variable
initial $dumpall;
```



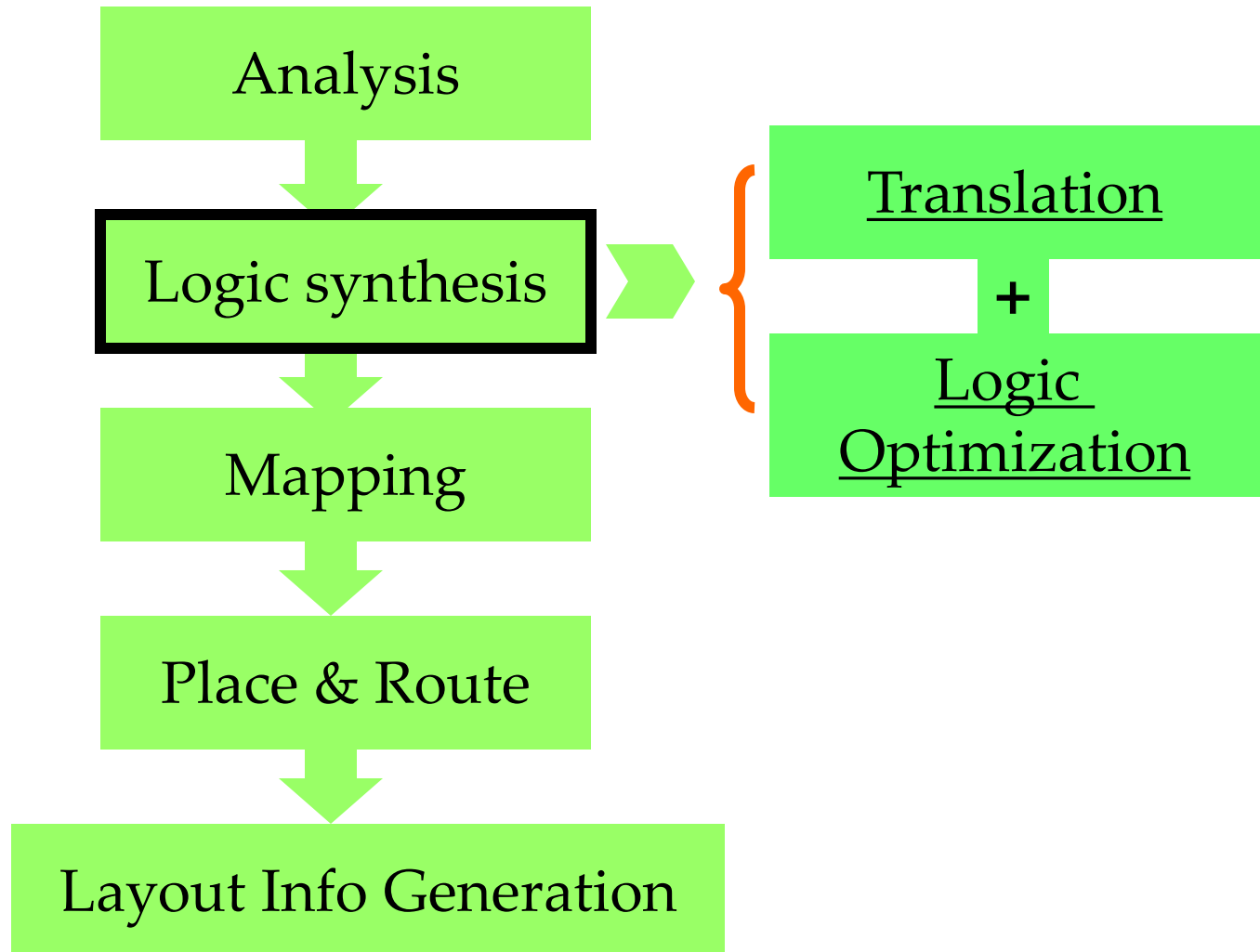
# Logic Synthesis

- Intro. To Logic Synthesis
- Coding Style Guidance for Synthesis

# What and Why Logic Synthesis ?

- Logic Synthesis :
  - Short Definition : A Process of Converting *High-Level* Description of Design into Optimized *Gate-Level* Representation
  - Input : Design Constraints + Design Source Code (in Verilog)
  - Output : Gate-Level Circuit (Netlist)
  - Means : Design Automation Software Tools like Synopsys
- Logic Synthesis = Translation + Optimization
- Impacts of Logic Synthesis
  - Technology independent description
  - Design Reuse
  - Time-to-Market
  - Higher Reliability
  - Fast Technology Migration

# Typical Logic Synthesis Flow



# Think in Hardware !

- Keep in mind that the code should imply hardware structure
  - Avoid purely algorithmic descriptions of hardware
- Avoid programming as in C or Java where we try exploit the sequentiality of the code. This will lead to long signal paths.
  - Attempt to minimize dependencies between statements and try to promote *concurrency*

# General Guide for Synthesis

- Do not Mix positive and negative edge-triggered Logic in One System
- Do Design in Synchronous Manner : Use Only One Clock Source
- Match the Simulation Results and Synthesis Results
  - Complete Sensitivity List in always block
  - Avoid Unwanted Latch Inference (if and case statement)
  - Note that initial blocks and Delay information are ignored during Synthesis

# Mismatch between Synthesis and Simulation

```
always @(a)
  o = a & b;
```



```
always @(a or b)
  o = a & b;
```

```
always @(a or b or c or d)
begin
  o = a & b | temp;
  temp = c & d;
end
```



```
always @(a or b or c or d)
begin
  temp = c & d;
  o = a & b | temp;
end
```

# Unwanted Latch Inference (1)

- If-else statements

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



```
always @(a or b)
if (a )
    c = b;
else
    c = 1'b0;
```

```
always @(a or b)
if (a )
    c = a;
else
    d = b;
```



```
always @(a or b)
begin
    c = 0; d = 0;
    if (a ) c = a;
    else d = b;
end
```

# Unwanted Latch Inference (2)

- Case statements

```
always @(addr) //infers latches for rce_n, mce1, mce0
    casez (addr) // synopsys full_case
        2'b10: {mce1, mce0} = 2'b10;
        2'b11: {mce1, mce0} = 2'b01;
        2'b0?: rce= 1'b0;
    endcase
```



```
always @(addr) begin
    {mce1_n, mce0_n, rce_n} = 3'b111; // initialize
    casez (addr)
        2'b10: {mce1, mce0} = 2'b10;
        2'b11: {mce1, mce0} = 2'b01;
        2'b0?: rce= 1'b0;
    endcase
end
```