



Verilog-HDL에 의한 디지털시스템 설계

참 고 문 헌

- 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

본 모듈의 특징

- 목적
 - 디지털시스템 설계시 많이 사용되는 Verilog-HDL을 배우고 이를 활용하여 다양한 디지털 시스템을 설계할 수 있도록 한다.
 - 본 모듈을 마치고 나면 PC에서 C 언어를 이용하여 소프트웨어를 개발하듯이, Verilog-HDL을 이용하여 디지털 하드웨어를 설계할 수 있게 된다.
- 소요시간: 2시간 / 주, 3주
- 선수내용: 논리설계

본 모듈에서 다루고자 하는 내용

- 먼저 Verilog-HDL이 무엇인지 알아보기 위한 예제를 다룸
 - Verilog-HDL을 이용하여 4-bit 카운터 설계하기
- 일반적인 디지털시스템 설계 과정에 대해 간략히 소개
- Verilog-HDL를 소개
 - 언어에 대한 기본 이해
 - 조합회로와 순차회로 설계하기
- 위에서 배운 Verilog-HDL을 이용한 디지털시스템 설계
 - 시계, Dice game, 교통신호 제어기 등

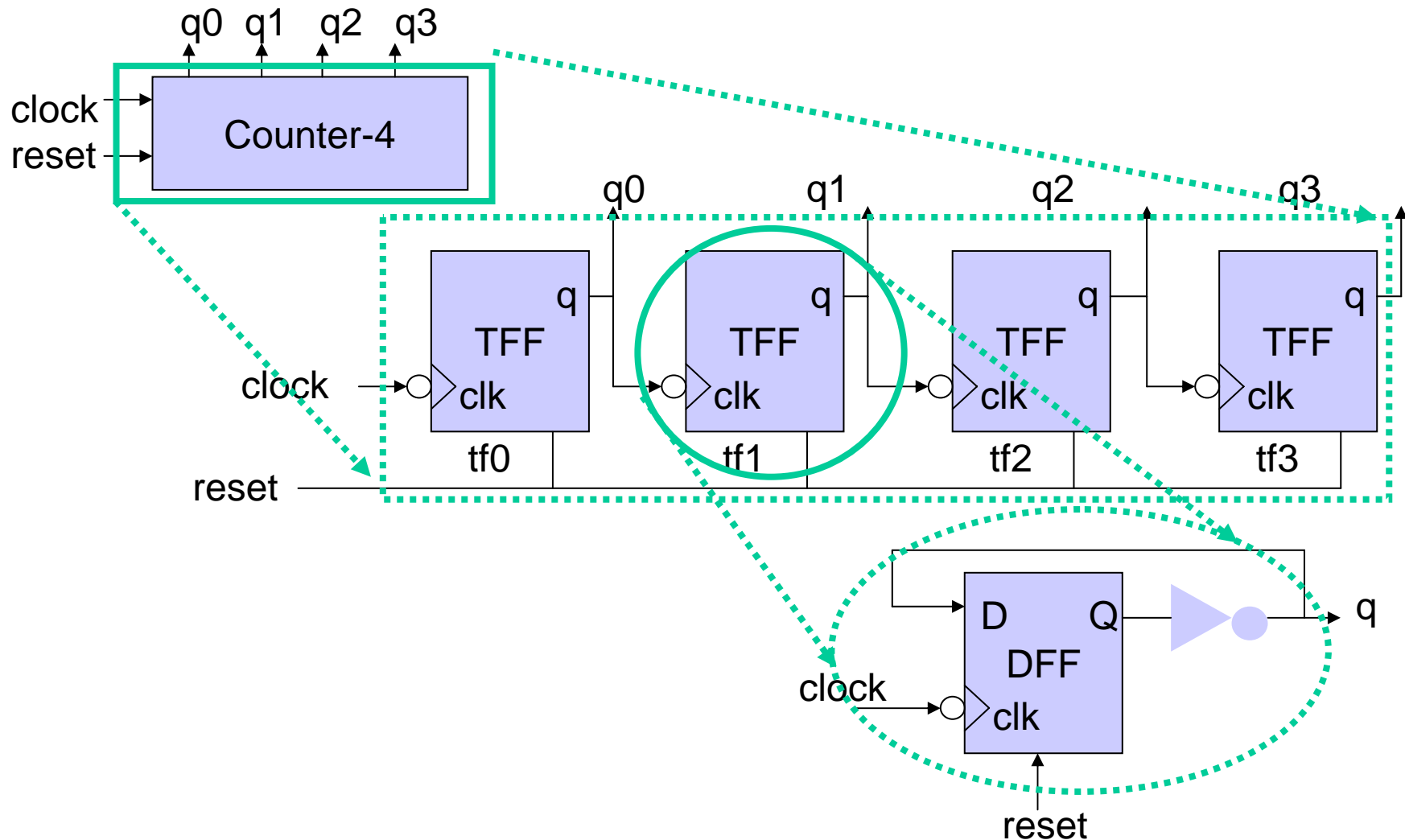
Verilog-HDL을 이용하여 4-bit 카운터 설계하기 (1/5)

사양

- Verilog-HDL이 하드웨어 설계에 어떻게 이용되는지 알아보기 위해 카운터를 설계해 본다.
- 비동기식 (asynchronous) 4-bit 카운터 만들기
 - 비동기식은 카운터를 구성하고 있는 플립플롭(flip-flop, FF)의 출력이 다음단의 플립플롭 클럭단에 연결된다.
- 카운터의 각 bit을 구성하고 있는 부분은 T-FF으로 구성한다.
- T-FF은 D-flipflop과 인버터(inverter)로 구성한다고 가정한다.

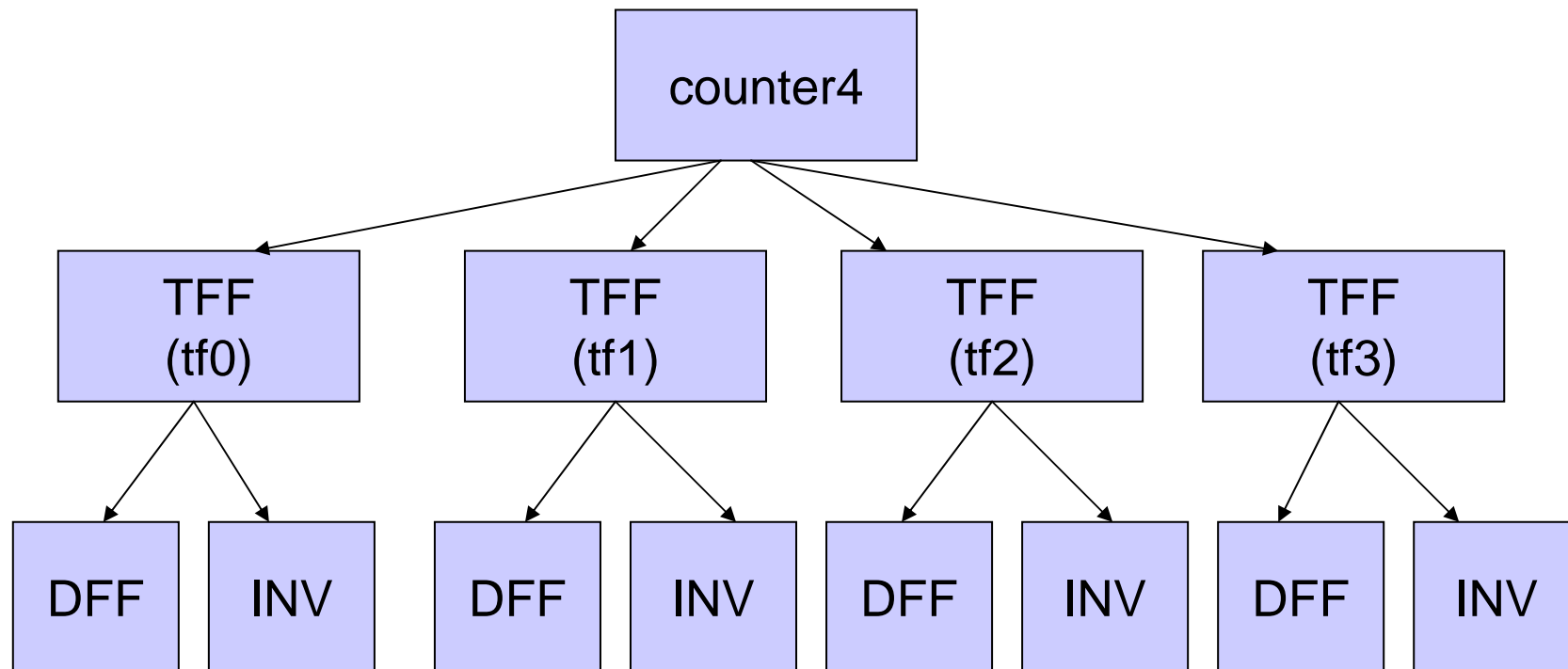
Verilog-HDL을 이용하여 4-bit 카운터 설계하기 (2/5)

4-bit 카운터 구성도



Verilog-HDL을 이용하여 4-bit 카운터 설계하기 (3/5)

계층적 구성도 (Hierarchical view)



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

Verilog-HDL을 이용하여 4-bit 카운터 설계하기 (4/5)

Top-level 코딩

```
module counter4 (q,clk,reset);  
  output [3:0] q;  
  input clk, reset;  
  TFF tf0 (q[0], clk, reset);  
  TFF tf1 (q[1], clk, reset);  
  TFF tf2 (q[2], clk, reset);  
  TFF tf3 (q[3], clk, reset);  
endmodule
```

```
-- in VHDL  
entity counter4 is port  
  (q : out std_logic_vector(3 downto 0);  
   clk, reset : in std_logic)  
end counter4;  
  
architecture RTL of counter_4 is  
  
  component TFF port  
    (q : out std_logic;  
     clk, reset : in std_logic)  
  end component;  
  
begin  
  tf0: TFF port map(q(0), clk, reset);  
  tf1: TFF port map(q(1), clk, reset);  
  tf2: TFF port map(q(2), clk, reset);  
  tf3: TFF port map(q(3), clk, reset);  
end RTL;
```


Verilog-HDL을 이용하여 4-bit 카운터 설계하기 (5/5)

✚ Submodule 예

```
module TFF (q, clk, reset);  
  output q;  
  input clk, reset;  
  wire d; // internal connection  
  D_FF dff0 (q,d,clk, reset);  
  not n1 (d, q); // primitive gate  
endmodule
```

```
module D_FF (q, d, clk, reset);  
  output q;  
  input d, clk, reset;  
  reg q;  
  always @(posedge reset or negedge clk)  
    if(reset)  
      q <= 1'b0;  
    else q <= d;  
endmodule
```

전체 목차

디지털시스템 설계 흐름도

Verilog-HDL Overview

- Basic Idea
- Verilog-HDL의 역사

Verilog-HDL

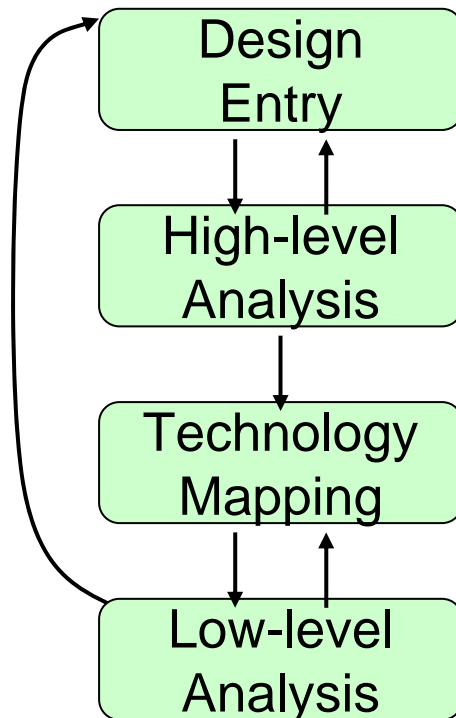
- Basics
- Modules & Ports
- Verilog Simulation
- Gate Level Modeling
- Dataflow Modeling

- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis

참고문헌

디지털시스템 설계 흐름도 (Design Flow)

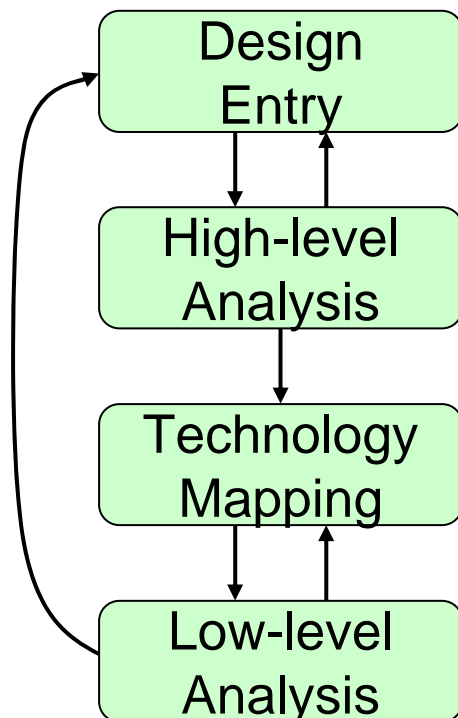
Design Entry



- 회로를 표현하고 기술하는 단계 :
 - 스키매틱(schematic)
 - 하드웨어설계언어 (HDL, 예: verilog, VHDL)
- 회로 설계의 결과물은 다음과 같은 종류들의 netlist들로 표현될 수 있음 :
 - 논리 게이트나 플립플롭과 같은 generic primitive들이거나
 - LUT/CLB, 트랜지스터 등과 같은 technology specific primitive 들,
 - 또는 adder, ALU, 레지스터 파일, 디코더 등과 같은 상위수준의 라이브러리 소자 등

디지털시스템 설계 흐름도

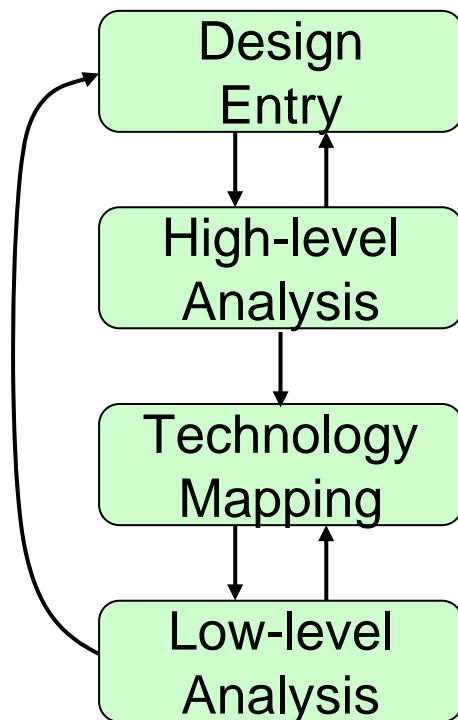
High-Level Analysis



- 상위수준의 분석(High-level analysis)은 원하는 설계 규격대로 정확히 동작하는지를 검증하는 단계이고, 타이밍과 파워, 비용 등에 대해서도 간단한 수준에서 검증
- 주요 사용되는 툴 :
 - simulator (기능 검증)
 - Verilog-XL (Cadence 사)
 - VCX (Synopsys 사)
 - static timing analyzer
 - 라이브러리 소자 또는 기본 primitive 들에 대한 타이밍 모델과 지연 파라미터 등에 기초하여 회로의 지연시간을 분석

디지털시스템 설계 흐름도

Technology Mapping 외



- Technology mapping 단계에서는 상위 설계 단계에서 얻어진 netlist를 사용하여 실제 하드웨어 구현에 사용될 technology에 적합하게 변환시킴 :
 - 라이브러리 소자들을 확장
 - partitioning, placement, routing 등을 수행
- Low-level analysis :
 - simulation과 static 툴들을 사용하여 정확한 타이밍 모델과 선로 지연시간 등을 모두 고려한 하위 수준에서 검증.
 - FPGA를 사용할 경우, 이 단계에서는 실제 FPGA 소자를 정하여 검증을 시행함.

Verilog-HDL : Overview

Basic Idea

- Basic idea :
 - 언어를 사용하여 회로 구성을 기술
 - Structural description은 계층적 netlist와 유사
 - Behavioral description은 보통 프로그래밍처럼 상위 수준의 개체를 사용
- 처음에는 abstraction과 simulation에 도움을 주기 위해 고안
 - 지금은 behavioral description을 자동으로 gate netlist로 변환해 주는 “logic synthesis”라는 툴이 있음
 - 이를 활용하면 설계자의 생산성을 매우 높여줌
 - 그러나, 이것은 하드웨어 설계를 단순히 프로그래밍의 일부처럼 잘못 인식할 수 있음.

• “Structural” example:

```
Decoder (a, b, x0, x1, x2, x3);
input a, b;
output x0, x1, x2, x3;
{
    wire abar, bbar;
    inv(bbar, b);
    inv(abar, a);
    nand(x0, abar, bbar);
    nand(x1, abar, b);
    nand(x2, a, bbar);
    nand(x3, a, b);
}
```

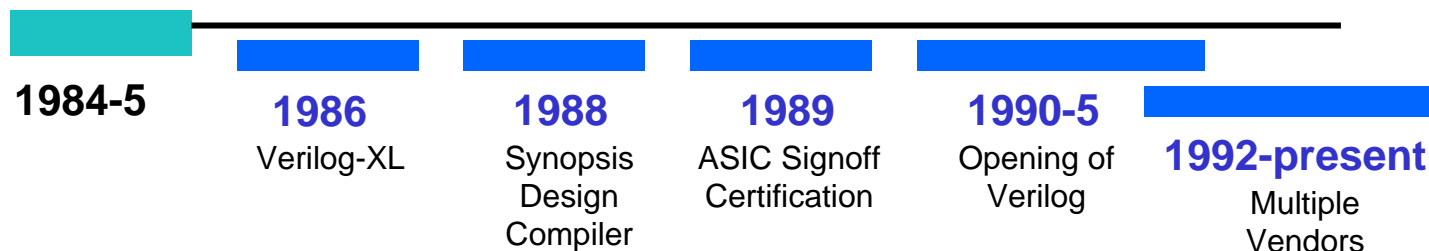
• “Behavioral” example:

```
Decoder (a, b, x0, x1, x2, x3);
input a, b;
output x0, x1, x2, x3;
{
    case (a, b)
        00: [x3 x2 x1 x0] = 0hE;
        01: [x3 x2 x1 x0] = 0hD;
        10: [x3 x2 x1 x0] = 0hB;
        11: [x3 x2 x1 x0] = 0h7;
    endcase;
}
```

Verilog-HDL : Overview

Verilog HDL의 역사

- Originated at Automated Integrated Design Systems (renamed Gateway) in 1985. Acquired by Cadence in 1989.
- Invented as simulation language. Synthesis was an afterthought. Many of the basic techniques for synthesis were developed at Berkeley in the 80's and applied commercially in the 90's.
- Around the same time as the origin of Verilog, the US Department of Defense developed VHDL. Because it was in the public domain it began to grow in popularity.
- Afraid of losing market share, Cadence opened Verilog to the public in 1990.
- An IEEE working group was established in 1993, and ratified IEEE Standard 1394 in 1995.
- Verilog is the language of choice of Silicon Valley companies, initially because of high-quality tool support and its similarity to C-language syntax.
- VHDL is still popular within the government, in Europe and Japan, and some Universities.
- Most major CAD frameworks now support both.
- Latest HDL: C++ based. OSCI (Open System C Initiative).



Verilog-HDL

목차 (Verilog-HDL)

- **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

Verilog-HDL : Basics

Lexical conventions - 1

- White Space
 - 단어를 구분하거나 텍스트를 좀 더 알아보기 쉽도록 하는데 사용됨
 - 예: black space, tab, carriage return, new-line, form-feed
- Comment
 - Single line comment
 - 토큰 // 로 시작하고 carriage return으로 끝남
 - Multi line comment
 - 토큰 /* 로 시작하고 토큰 */ 로 끝남

```

/* 1-bit adder example for showing
few verilog */ Multi line comment
module addbit (
a,
b,
ci,
sum,
co);
// Input Ports Single line comment
input      a;
input      b;
input      ci;
// Output ports
output      sum;
output      co;
// Data Types
wire        a;
wire        b;
wire        ci;
wire        sum;

```

Verilog-HDL : Basics

✚ Lexical conventions - 2

- Case sensitivity
 - Verilog HDL은 대소문자 구분함
 - Verilog에서 사용하는 모든 키워드는 소문자임
 - (비고) 키워드는 대소문자 조합 상관없이 변수명 등으로 절대로 사용하지 않기
- Identifiers
 - module이나 signal, variable 등에 부여되는 이름
 - 반드시 영문자나 _ 로 시작 (a-z A-Z _)
 - 사용할 수 있는 문자들
 - a-z A-Z _ 0-9 \$
- Escaped Identifiers
 - Back slash (\)를 사용하며, 임의의 모든 문자를 사용 가능하게 함
 - Escaped identifier는 white space로 끝남

<대소문자 구분하는 이름들의 예>

```
input      // Verilog keyword
wire      // Verilog keyword
WIRE      // unique name (키워드아님)
Wire      // unique name (키워드아님)
```

<Identifier 예>

data_in, my\$clk, i486, A

In VHDL,

- Comments : --
- Identifier cannot start with _
- Case insensitive

Verilog-HDL : Basics – Data Types

Integer Numbers

- Sized or unsized number 지정 가능
- Sized number의 경우
 - Syntax: `<size>'<base><value>`
 - Binary, octal, decimal, hexadecimal 형태
 - 예: `4'b1111` `12'habc` `16'd255`
- Unsized number의 경우
 - Syntax: `'<radix><base>` 또는 `<value>`
 - `23456` // default is decimal (32-bit : machine dependent)
 - `'hc3` // 32-bit hexadecimal
 - `'o21` // 32-bit octal
- Negative numbers: `-3` (32 bit, signed number),
 `-6'd3` (6 bit unsigned nbr, 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??`

In VHDL
: no sized number

Verilog-HDL : Basics – Data Types

Real Numbers

- Verilog에서 실수형 상수 또는 변수를 지원
 - Real number를 integer로 변환시 rounding 함
 - Real number에는 'Z'나 'X'를 포함할 수 없음
 - Decimal이나 공학 표현식으로 지정

Syntax: <value>.<value>
 <mantissa>E<exponent>

예: 1.2 0.6 3.5E6

Verilog-HDL : Basics – Data Types

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)
 - Small (triereg type only)
 - Highz

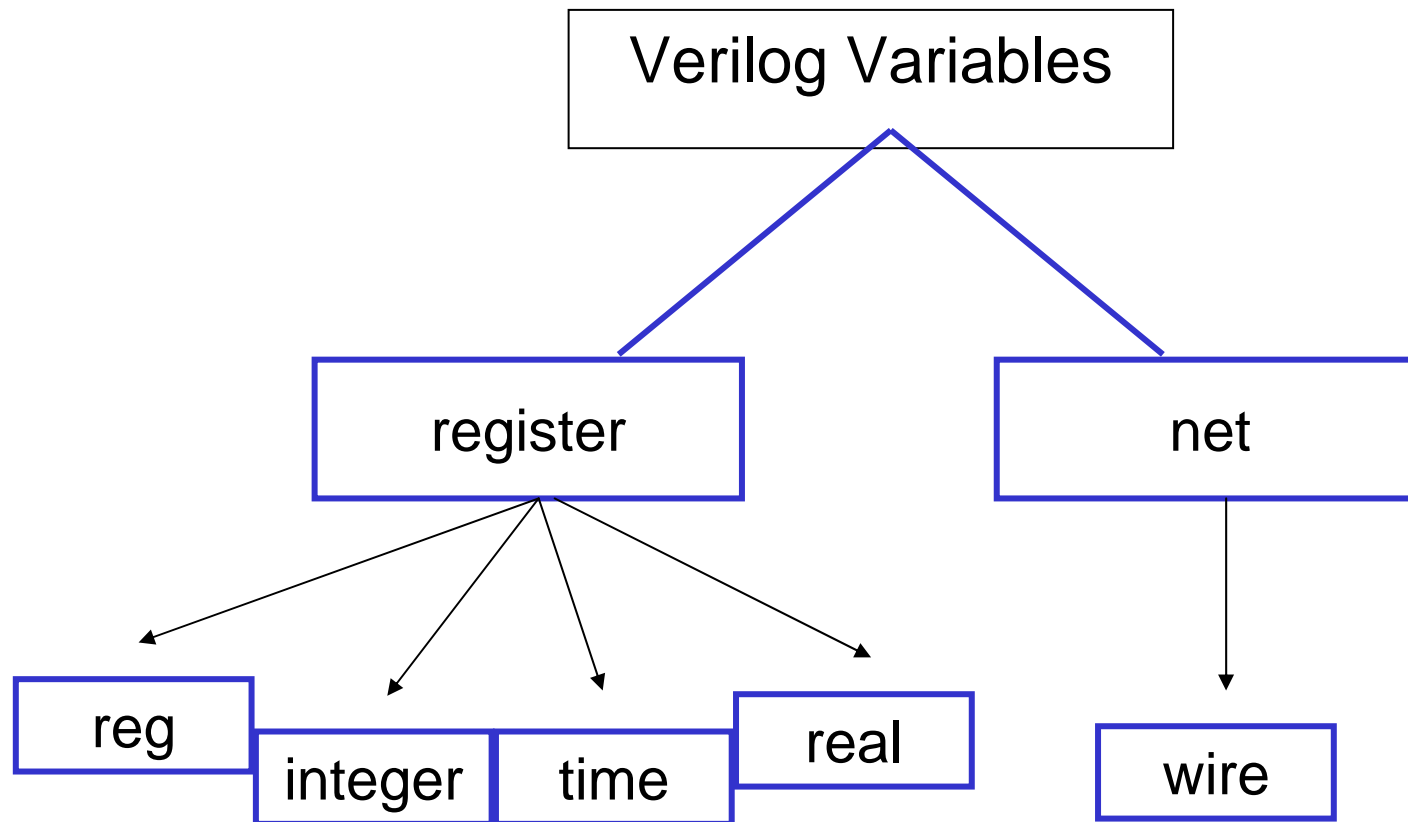
In VHDL, std_logic type

- 9 value types ('0', '1', 'Z', 'X', 'U', '-', 'W', 'H', 'L')
- user defined type

Strength Level	Strength Name	Specification Keyword		Display Mnemonic	
7	Supply Drive	supply0	supply1	Su0	Su1
6	Strong Drive	strong0	strong1	St0	St1
5	Pull Drive	pull0	pull1	Pu0	Pu1
4	Large Capacitive	large		La0	La1
3	Weak Drive	weak0	weak1	We0	We1
2	Med. Capacitive	medium		Me0	Me1
1	Small Capacitive	small		Sm0	Sm1
0	High Impedance	highz0	highz1	HiZ0	HiZ1

Verilog-HDL : Basics – Data Types

+ Data Types - 1



In VHDL, all signals can be both register and net type

Verilog-HDL : Basics – Data Types

✚ Data Types - 2

- 두가지 기본 데이터 형
 - Nets: 컴포넌트 간의 구조적 연결을 나타냄
 - Registers: 데이터를 저장하는데 사용되는 변수를 지정
 - Verilog code에서 내부 기본 선언은 **wire** 형태의 **net**의 속성과 1-bit임
- Net
 - 연결된 디바이스의 출력값으로 항상 구동, default 값은 z 임
 - 키워드 **wire, trireg, tri, wand, wor, triand, trior** 등으로 선언됨
 - **wire**가 가장 많이 사용됨

(예) **wire** a;

Net Data Type		Functionality
wire	tri	Interconnecting wire - no special resolution function
wor	trior	Wired outputs OR together (models ECL)
wand	triand	Wired outputs AND together (models open-collector)
tri0	tri1	Net pulls-down or pulls-up when not driven
supply0	supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg		

Verilog-HDL : Basics – Data Types

+ Data Types - 3

- Register의 데이터 type
 - 데이터를 저장하는데 사용되는 변수를 지정
 - Register 배열을 사용하면 메모리를 선언할 수 있음
 - 키워드 **reg**, **integer**, **real**, **time** 으로 선언
 - 실제 회로에서의 레지스터(플립플롭)과는 상관없음. **reg** type은 실제 회로에서 wire에 해당
 - Register 데이터 형은 procedural block에서 사용
 - Procedural block은 키워드 **initial**과 **always**로 시작함
 - **reg**가 가장 많이 사용됨
 - ex) **reg** reset;


```

          initial begin
              reset = 1'b1;
              #100 reset = 1'b0;
          end
          
```

Data Types	Functionality
reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable

Verilog-HDL : Basics – Data Types

예제 : net와 reg

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

--In VHDL, all output signal is register data type
entity design is
port (a, x : in std_logic;
 b, c : out std_logic);
end design;
architecture archi of design is
begin
 c <= a and x ;
 process (a,x) begin
 b <= a or x;
 end process;
end archi;

Verilog-HDL : Basics – Data Types

✚ reg 형 : 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

Verilog-HDL : Basics – Data Types

reg 형: 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)

Verilog-HDL : Basics – Data Types

Vectors: multiple-bit data

- Nets or reg data type can be declared as multiple bit width using vector
- Syntax: 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]`

Verilog-HDL : Basics – Data Types

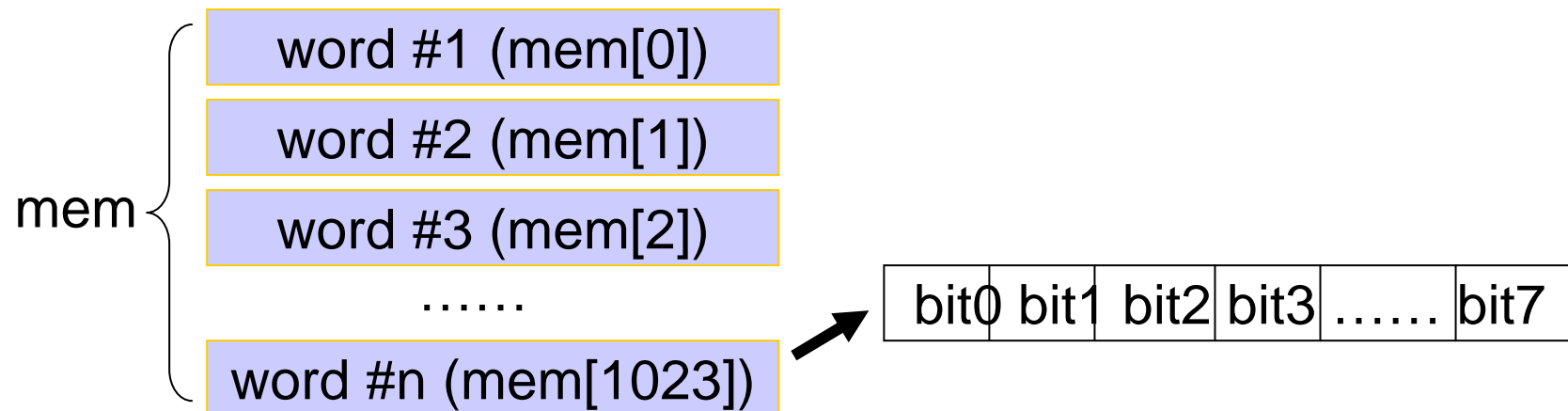
배열 (Arrays)

- Allowed for reg, integer, and time, and vector register data type
- Not allowed for real variable
- Syntax: data_type <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[2], chk_point[3], 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

Verilog-HDL : Basics – Data Types

메모리

- 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`



Verilog-HDL : Basics – Data Types

문자열 (strings)

- 문자열(strings)
 - Double quote를 사용하여 문자열을 만듦
 - 하나의 line에 표현되어야 함
 - 각각의 문자는 1 바이트로 표현되고 레지스터에 저장될 수 있음
 - 예:

```
reg [8*17:0] version;    // declare a register variable that is 18 bytes
reg [8*18:1] string_val; // 18 bytes wide
initial
    version = "model version 1.0";
    string_val = "Hello Verilog World"; // store string value, 19bits
```
- 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

Verilog-HDL : Basics – Data Types

❖ 파라미터 (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 ;
```


Verilog-HDL : Basics

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`

Verilog-HDL : Basics

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];
```

Verilog-HDL : 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

Verilog-HDL : Modules and Ports

동시성 (Concurrency)

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

Verilog-HDL : Module과 Port

✚ 모듈의 포트 (Port)

- 모듈과 외부와 데이터를 주고 받을 수 있도록 해 줌
- 각 포트의 모드 세가지: `input`, `output` 또는 `inout`
- 모든 포트는 기본적으로 `wire`의 속성으로 선언됨
- 따라서 출력 포트 단자의 값을 유지하려면 `reg`로 선언해야 함
- `input`, `inout` 형의 포트는 `reg`로 선언할 수 없음 (뒤에 설명)
- syntax:

```
input  [range_val1:range_val2] list_of_identifiers;  
output [range_val1:range_val2] list_of_identifiers;  
inout  [range_val1:range_val2] list_of_identifiers;
```

- 예:

```
input      clk;           // Clock input  
input  [15:0] data_in;    // 16 bit data input bus  
output [7:0]  count;      // 8 bit counter output  
inout      data_bi;       // Bi-directional data bus  
reg        q;             // Output port q holds value
```

Verilog-HDL : Module과 Port

✚ Signal과 Instance Port간의 연결

- 순서(order)에 의한 연결

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

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

- 이름(name)에 의한 연결

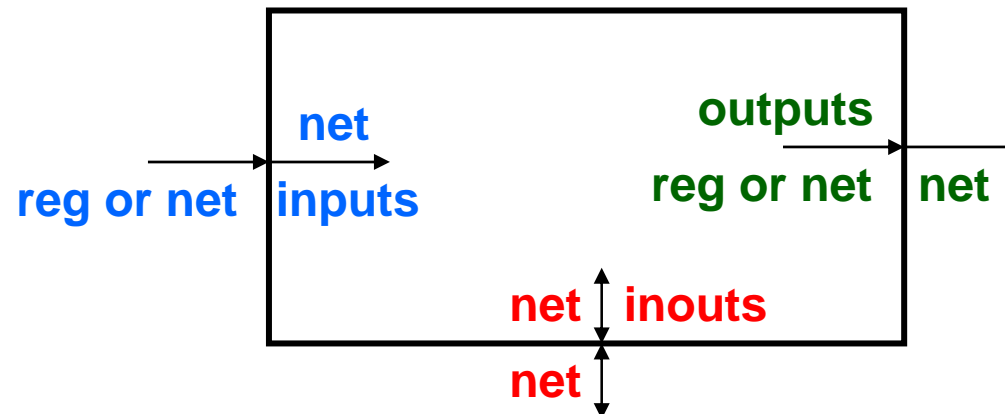
```
module top ;
fulladder4 U0 (.cout(C_O), .sum(SUM), .a(A), .b(B), .c(C_IN));
...
endmodule
```

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

Verilog-HDL : Module과 Port

✚ 포트 연결 규칙 (Port Connection Rules)

- **input** port는 net 형이어야 함
- **inout** port도 net 형이어야 함
- **output** port는 net or reg 타입일 수 있음

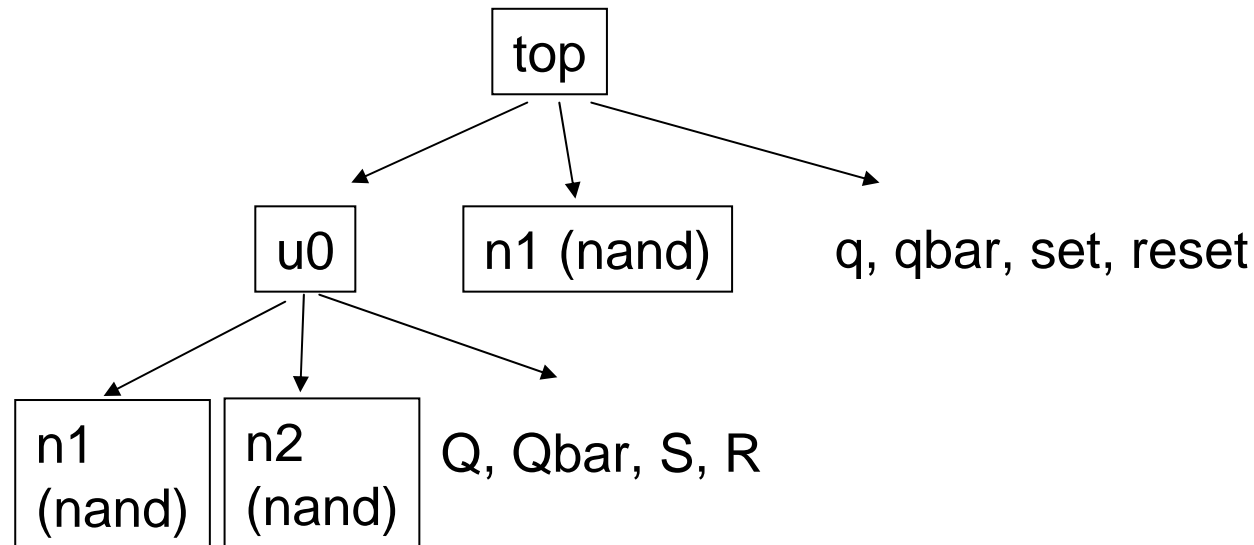


- Width matching: 서로 다른 크기의 내/외부 포트를 연결하는 것이 가능
- Unconnected ports: 연결되지 않는 포트는 “,”를 사용하여 표시
- 예:


```
dff u1 (q, , clk, d, rst, pre); // here second port is not connected
```
- **inout** port도 net 형이어야 함
- **output** port는 net or reg 타입일 수 있음

Verilog-HDL: Module과 Port

계층 이름 (Hierarchical Names)



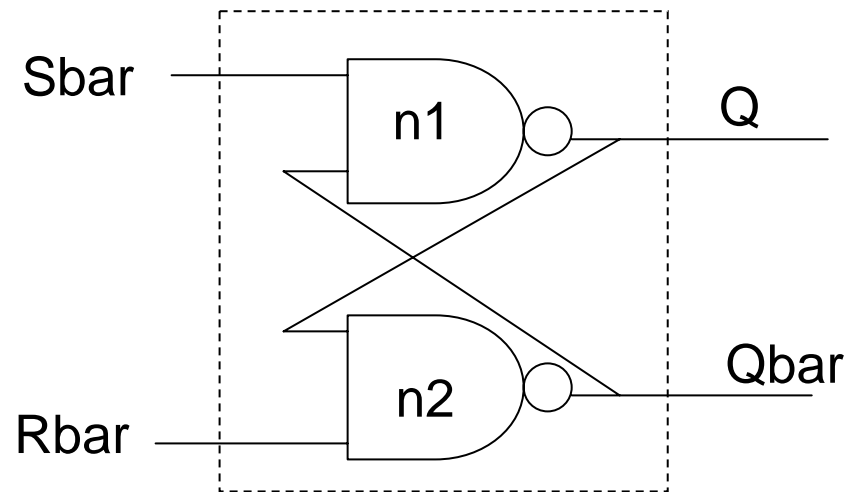
- 계층적 경로 이름은 top module identifier와 module instant identifier로 구성되며 점(.)으로 구분함

Syntax: Module_name.instance_name.port_name

- Ex) top.q top.u0.Q top.n1

Verilog-HDL : Module과 Port

회로 예제

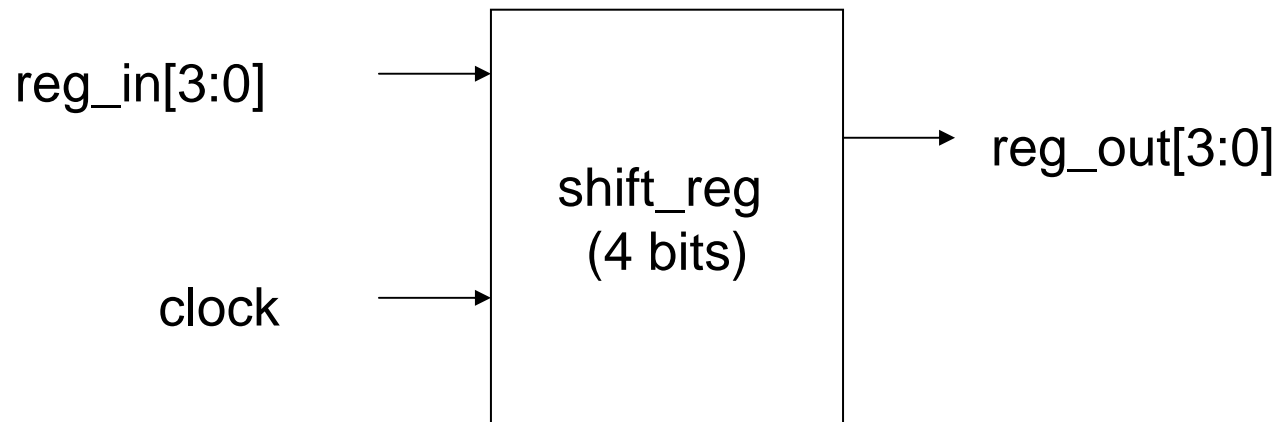


```
module SR_latch (Q, Qbar, Sbar,Rbar) ; // SR latch module
output Q, Qbar;
input Sbar, Rbar ;
nand n1 (Q, Sbar, Qbar);
nand n2 (Qbar, Rbar, Q);
endmodule
```

Verilog-HDL : Module과 Port

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-HDL

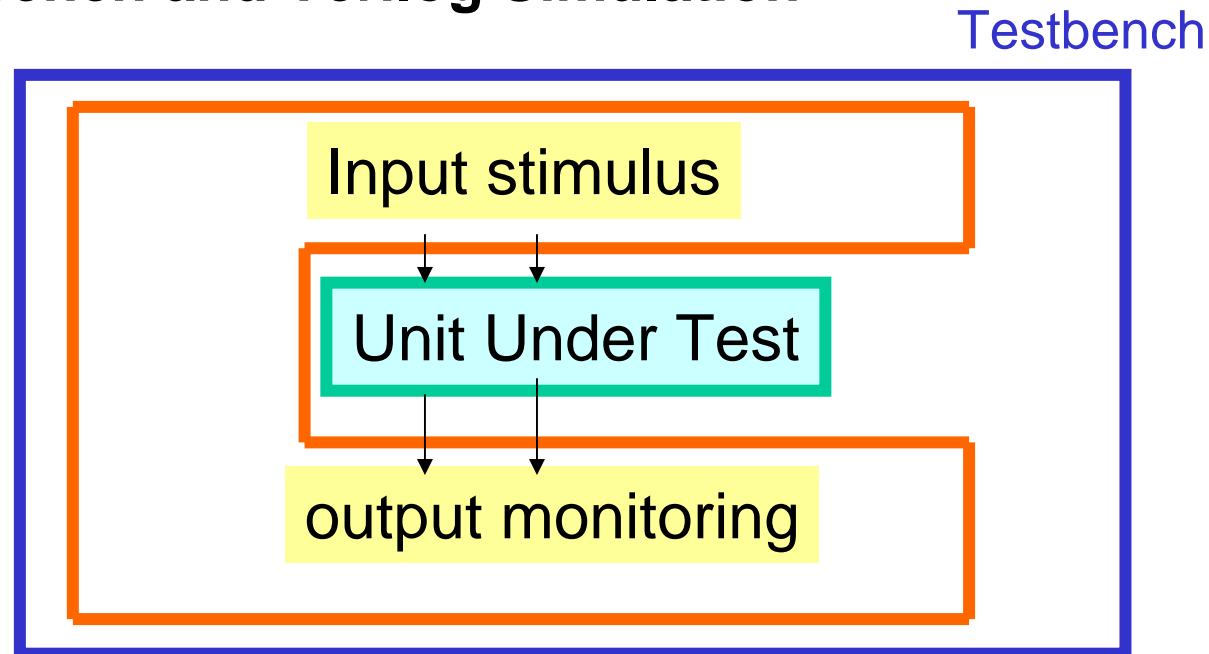


목 차

- Basics
- Modules & Ports
- **Verilog Simulation**
 - Testbench
 - System Tasks for Simulation
 - Timescale
- Gate Level Modeling
- Dataflow Modeling
- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis

Verilog-HDL : Verilog Simulation

+ 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-HDL: Verilog Simulation

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;
```

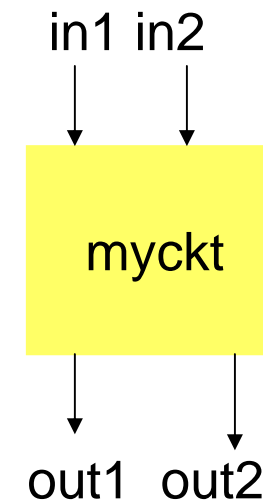
```
    #8 in1 = 1'b1; #8 in2 = 1'b1;
```

```
end
```

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

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

```
endmodule
```



Verilog-HDL : Verilog Simulation

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;
```

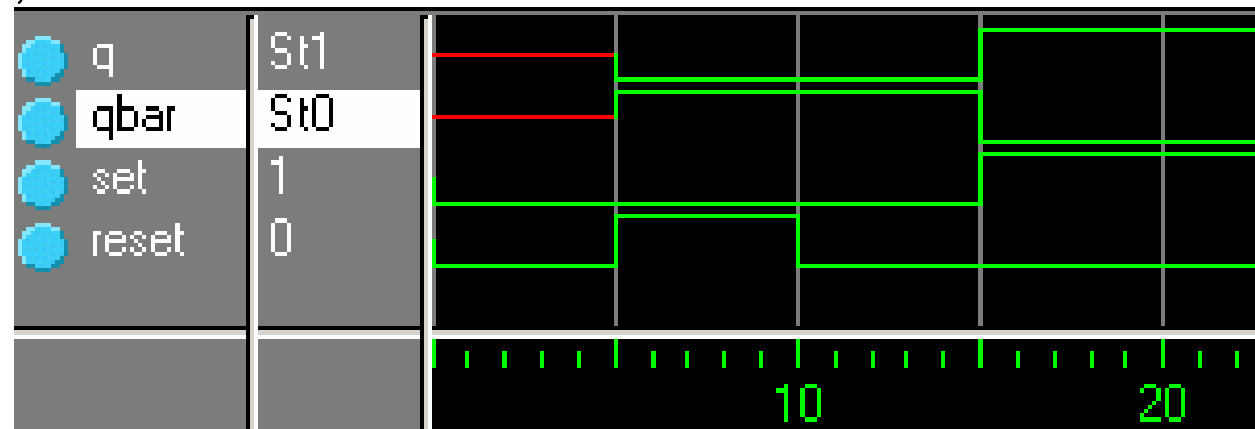
```
    #5 reset = 0;
```

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

```
end
```

```
endmodule
```

The result waveform



Verilog-HDL : Verilog Simulation

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

Verilog-HDL : Verilog Simulation

\$display

- Usage: \$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");

Verilog-HDL : Verilog Simulation

\$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
- \$monitoroff : a system task disabling monitoring
- Monitoring is turned on by default at the beginning of simulation

Verilog-HDL : Verilog 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
```

Verilog-HDL : Verilog Simulation

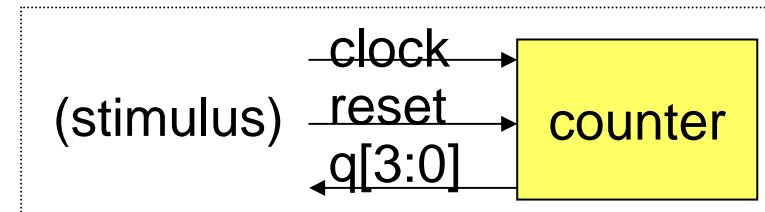
Testbench Example with System Tasks

```

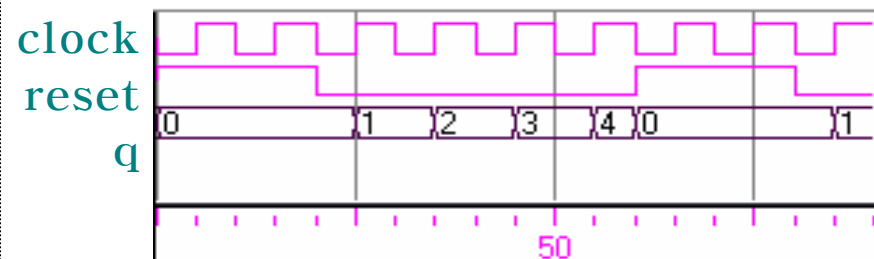
module stimulus ;
reg clock, reset;
wire [3:0] q;

bin_counter r1 (q, clock, reset);

// clock period :10
initial forever #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



Verilog-HDL : Verilog Simulation

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)**
- Example)
 - ``timescale 100 ns / 1 ns`
 - ``timescale 1 us / 10 ns`

Verilog-HDL



목 차

- Basics
- Modules & Ports
- Verilog Simulation
- **Gate Level Modeling**
 - Primitive Gates
 - Gate Delay
- Dataflow Modeling
- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis
- Appendix: FPGA Kit 요약

Verilog-HDL : Gate Level Modeling

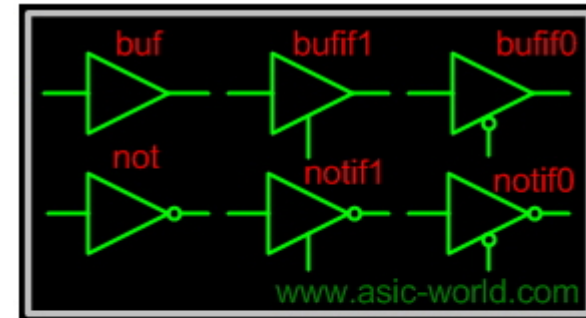
회로 기술의 3가지 유형

- 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

Verilog-HDL : Gate Level Modeling

✚ Introduction

- 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, not, bufif1, bufif0, notif1, notif0
 - One output and any number of inputs



Gate	Description
not	N-output inverter
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tri-state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Verilog-HDL : Gate Level Modeling

AND/OR type (N-input primitives)

- Types : `and`, `or`, `nand`, `nor`, `xor`, `xnor`
- Ports order : for N input gate
Output, Input-1, Input-2, ... Input-N
- Input/output ports number :
 - only 1 output ports,
 - any number of input ports (≥ 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

Verilog-HDL : Gate Level Modeling

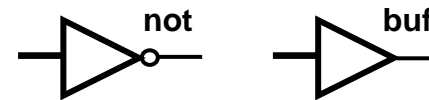
✚ Buffer (N-output primitives)

- Types : `buf`, `not`
- Ports order :
 Output1, [Output2, ...,] Input
- Number of IOs
 - any number of output ports (≥ 1)
 - Only 1 input port
- Examples)

`buf b1 (OUT, IN);` -- simple buffer

`not n1 (OUT, IN);` -- inverter

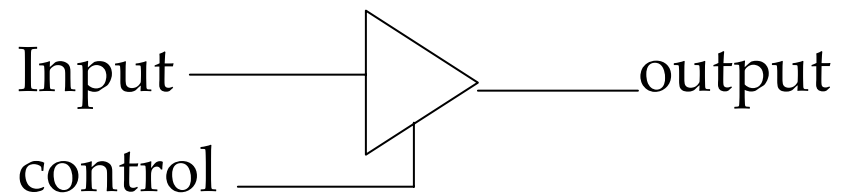
`buf bf_2out (OUT1, OUT2, IN);` -- more than 2 outputs



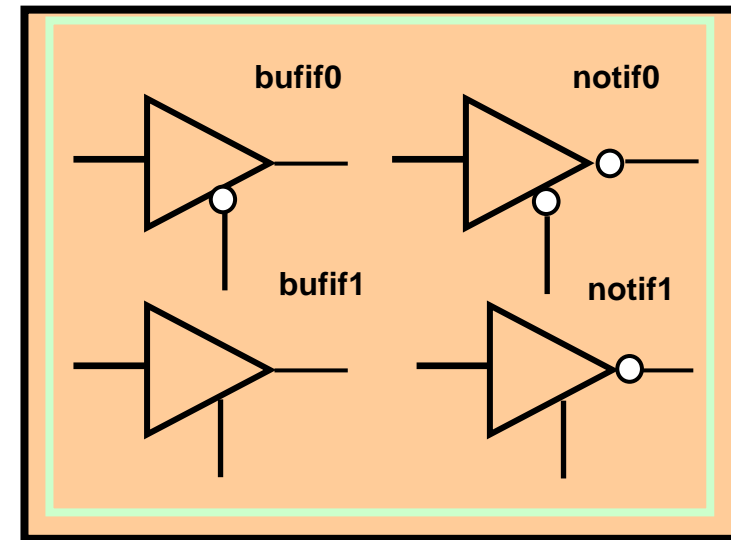
Verilog-HDL : Gate Level Modeling

✚ Tri-state buffer

- Types : bufif1, notif1, bufif0, notif0

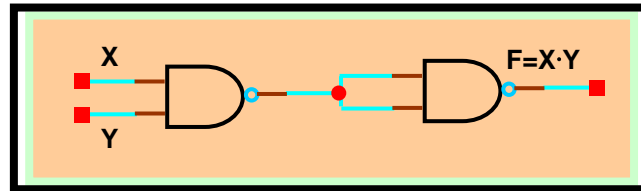


- Port order : output, input, control
– (Ex)bufif1 u0 (out, in, ctrl);



Verilog-HDL : Gate Level Modeling

✚ **NAND** 게이트를 이용하여 **AND** 게이트 만들기



// Structural model of AND gate from two NANDS

```
module and_from_nand(X, Y, F);
```

```
input X, Y;
```

```
output F;
```

```
wire W;
```

// Two instantiations of the module NAND

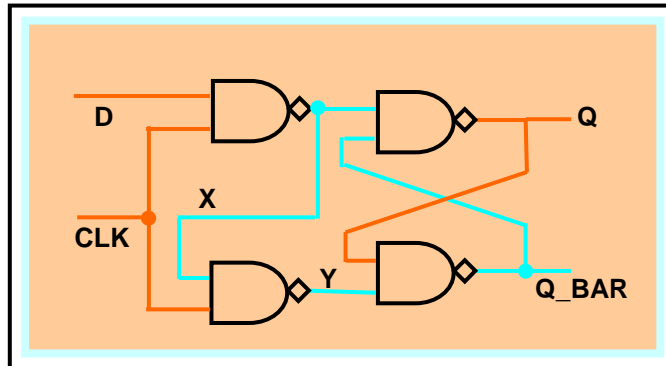
```
nand U1(W, X, Y);
```

```
nand U2(F, W, W);
```

```
endmodule
```

Verilog-HDL : Gate Level Modeling

✚ **NAND** 게이트를 이용하여 **DFF** 만들기



```
module dff(Q, Q_BAR, D, CLK);  
  output Q, Q_BAR;  
  input D, CLK;
```

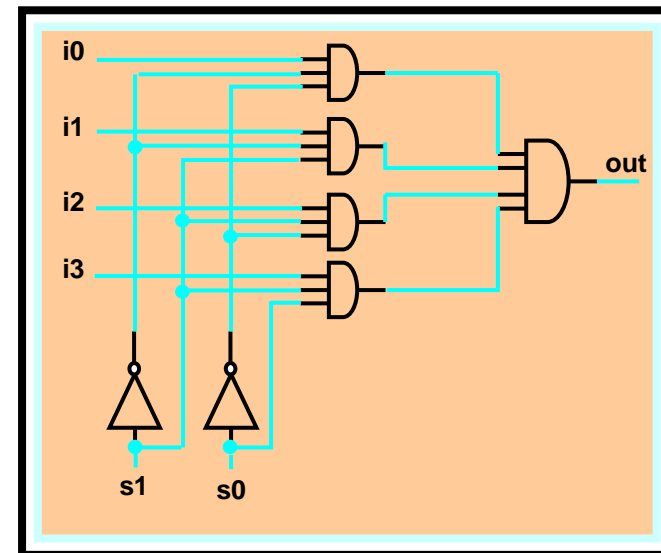
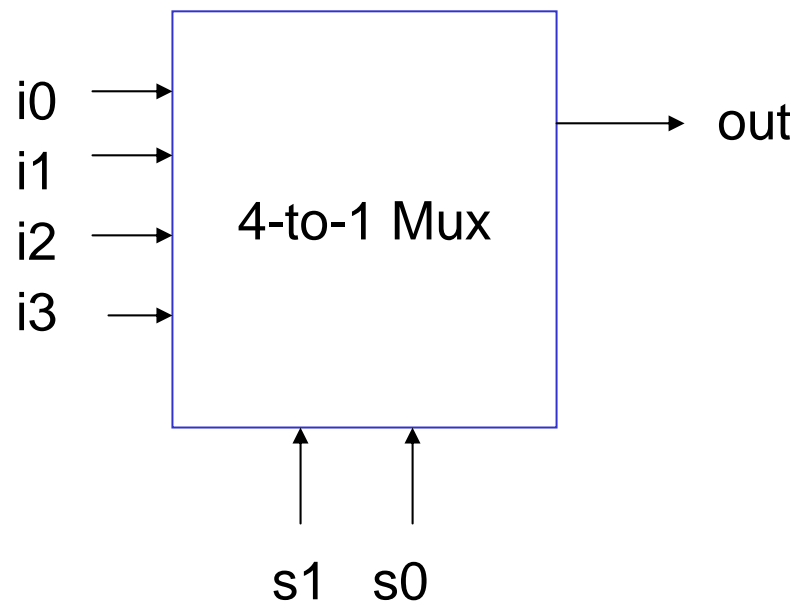
```
  nand U1 (X, D, CLK) ;  
  nand U2 (Y, X, CLK) ;  
  nand U3 (Q, Q_BAR, X);  
  nand U4 (Q_BAR, Q, Y);
```

```
endmodule
```

Verilog-HDL : Gate Level Modeling

Example: 4-to-1 Multiplexer

- Block diagram



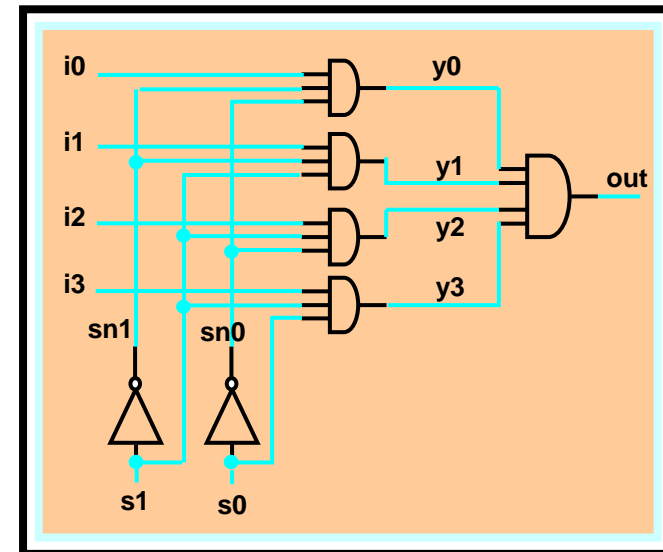
Verilog-HDL : Gate Level Modeling

✚ Example: 4-to-1 Multiplexer – Verilog code

```
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
```



Verilog-HDL : Gate Level Modeling

Example: 4-to-1 Multiplexer – Testbench

```
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;
```

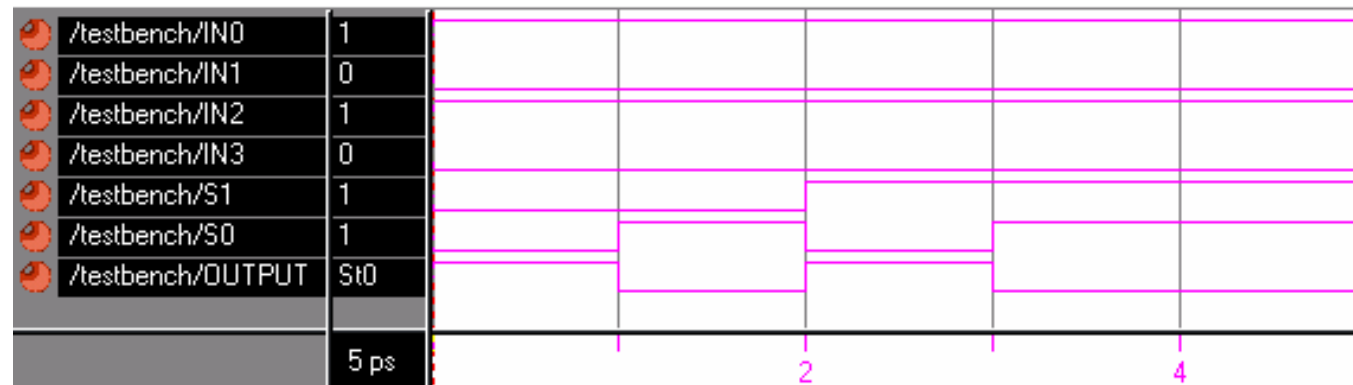
```
#1 S1 = 0; S0 = 1;
```

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

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

```
end
```

```
endmodule
```



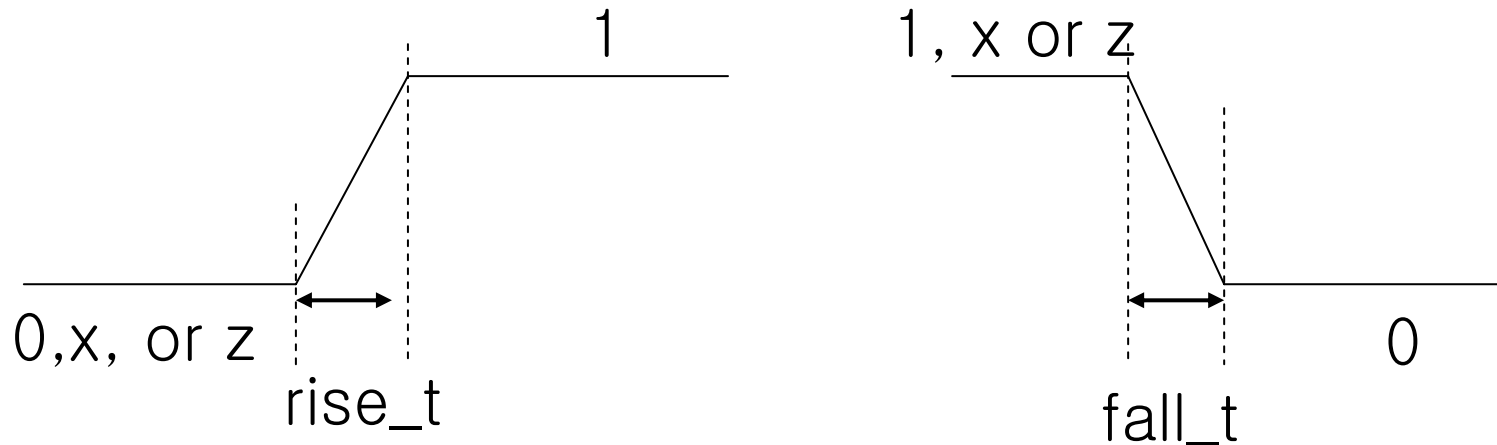
Verilog-HDL : Gate Level Modeling

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

Verilog-HDL : Gate Level Modeling

✚ 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);`

Verilog-HDL : Gate Level Modeling

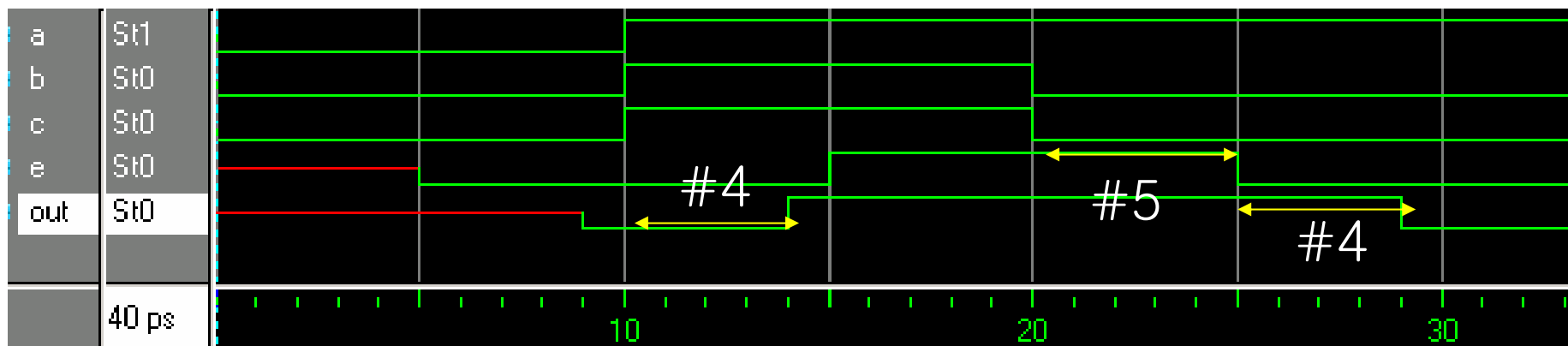
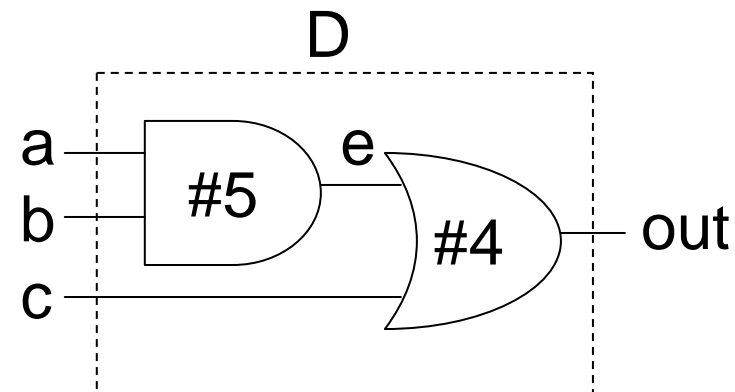
Gate Delays: min, typ, max

- 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);`

Verilog-HDL : Gate Level Modeling

Examples: Gate Delay

```
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
```



Verilog-HDL

목 차

- Basics
- Modules & Ports
- Verilog Simulation
- Gate Level Modeling
- **Dataflow Modeling**
 - Continuous Assignment
 - Inertia Delay
 - Verilog Operators
- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis

Verilog-HDL : Dataflow Modeling

Introduction

- 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

Verilog-HDL : Dataflow Modeling

Continuous Assignments

- Net에 특정 논리 값을 지정하는데 사용
- 키워드 “assign”으로 시작
- Syntax :
`assign <drive_strength> <delay> <list of assignments> ;`
- 특징
 - 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`

Verilog-HDL : Dataflow Modeling

Implicit Continuous Assignments

- 선언시 Assignment Operator (=)로 함께 지정
- 키워드 “assign”으로 시작하지 않음

- Examples

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

- 위의 Implicit Continuous Assignment는 다음과 같이 동일하게 표현할 수 있음

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

Verilog-HDL : Dataflow Modeling

Delay Type과 예제

- 다음의 세 예제는 동일한 효과

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

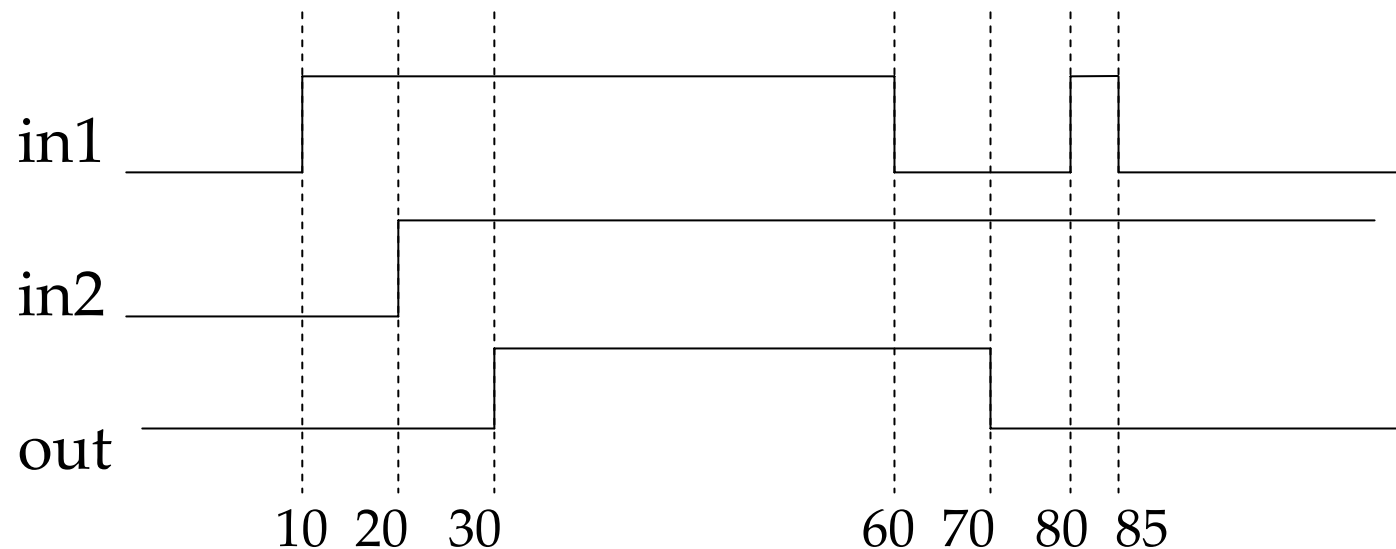
- Delay can be specified on a net when declared without a continuous assignment on the net
- Ex) **wire** # 10 out;
assign out = in1 & in2;

Verilog-HDL : Dataflow Modeling

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;
```



Verilog-HDL : Dataflow Modeling

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 reg_val;
```

```
final_cnt = count + 1;
```

```
reg_out = reg1[3:0] ^ reg2[3:0];
```

```
reg_val = calculate_parity (A,B) ;
```

Verilog-HDL : Dataflow Modeling

Operators

- Arithmetic operators
- Logical operators
- Relational operators
- Equality operators
- Bit-wise operators
- Reduction operators
- Shift operators
- Concatenation operator
- Conditional operator
- Operator precedence

Verilog-HDL : Dataflow Modeling

Arithmetic Operators

- Types

$*$, $/$, $+$, $-$ Arithmetic
 $\%$ Modulus

- If any operands has **x** value, 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) $\text{-d10} / 5$; // (2's complement of 10) / 5 = $(2^{32}-10)/5$

- $\%$ (modulus) operator takes the sign of the first operand

(Ex) $(-7) \% (+2) = -1$

$(+7) \% (-2) = +1$

Verilog-HDL : Dataflow Modeling

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)

Verilog-HDL : Dataflow Modeling

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
```

Verilog-HDL : Dataflow Modeling

Bitwise Operators

- 종류

~	bit-wise negation	&	bit-wise AND
	bit-wise OR	^	bit-wise exclusive OR
^~, ~^ bit-wise equivalence (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
```

- 주의:

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

Verilog-HDL : Dataflow Modeling

Reduction Operators

- Unary Operators
- Types :

&	reduction AND	~&	reduction NAND
	reduction OR	~	reduction NOR
^	reduction XOR	~^, ^~	reduction XNOR
- 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
```


Verilog-HDL : Dataflow Modeling

Shift Operators

- Types :
 - << left shift
 - >> right shift

- 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)
```

Verilog-HDL : Dataflow Modeling

Conditional Operators

- 세 개의 Operand를 가짐
- Syntax

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
- 중첩될 수 있음 (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
```

```
assign muxout = (sel == 1'b1) ? A : B;
```

Verilog-HDL : Dataflow Modeling

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`
`assign #102 {co,sum} = a + b + ci;`
- 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`
`assign byte = {4{2'b01}}; // Generate 8'b01010101`
`assign word = {{8'{byte[7]}}, byte}; // Sign extension`

Verilog-HDL : Dataflow Modeling

Operators Summary - 1

- Arithmetic operators
 - + - * / arithmetic
 - % modulus
- Logical operators
 - ! logical negation
 - && logical and
 - || logical or
- Relational operators
 - > < >= <= relational
- Equality operators
 - == logical equality
 - != logical inequality
 - === case equality
 - !== case inequality
- Bit-wise operators
 - ~ bit-wise negation
 - & bit-wise and
 - | bit-wise inclusive or
 - ^ bit-wise exclusive or
 - ^~ bit-wise equivalence
 - ~^ bit-wise equivalence
- Reduction operators
 - & reduction and
 - ~& reduction nand
 - | reduction or
 - ~| reduction nor
 - ^ reduction xor
 - ~^ reduction xnor
 - ^~ reduction xnor

Verilog-HDL : Dataflow Modeling

Operators Summary - 2

- Shift operators
 - << left shift
 - >> right shift
- Concatenation operator
 - { } concatenation
- Replication
 - { { } } replication
- Conditional operator
 - ? : conditional

Verilog-HDL : Dataflow Modeling

Operator Precedence

Operator	Operator Symbols	Precedence
1. Unary, Multiply, Divide, Modulus	+ - ! ~ * / %	Highest Precedence
2. Add, Subtract, Shift	+ - << >>	↑
3. Relational, Equality	< <= > >= == != === !==	
4. Reduction. Logical	&, ~& ^ ^~ ~ &&	
5. Conditional	?:	↓ Lowest Precedence

Verilog-HDL : Dataflow Modeling

Example: Multiplexer

```
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
```

Verilog-HDL : Dataflow Modeling

Exercises - 1

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

PRT parity
 data[6:0] parity_gen

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

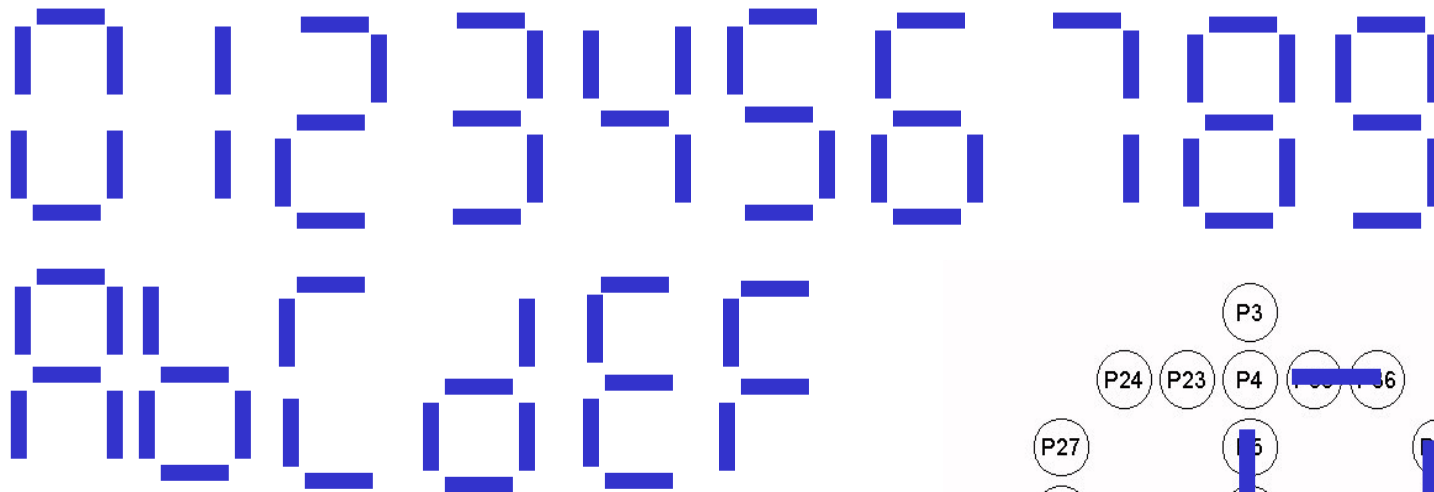
A[7:0] GT(A>B)
 B[7:0] comparator LT(A<B)
 EQ(A=B)

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

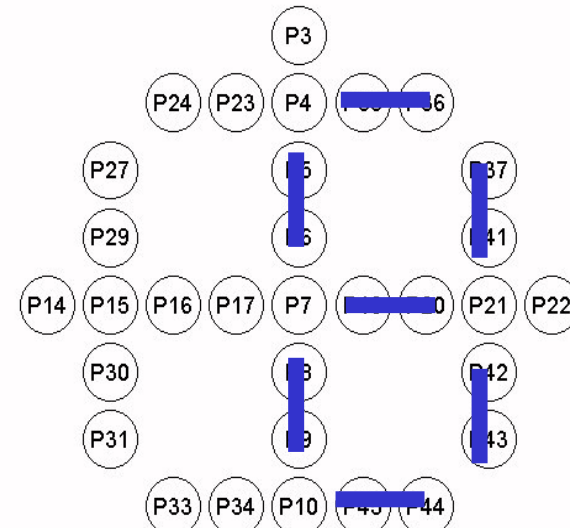
Verilog-HDL : Dataflow Modeling

✚ 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.



Verilog-HDL

목 차

- Basics
- Modules & Ports
- Verilog Simulation
- Gate Level Modeling
- Dataflow Modeling
- **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)
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis

Verilog-HDL : Behavioral Modeling

Introduction

- Description of Design Functionality in an Higher Level
- All the behavioral statements appear only inside initial and always blocks
- always block / initial block :
 - Two 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**

Verilog-HDL : Behavioral Modeling

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

```

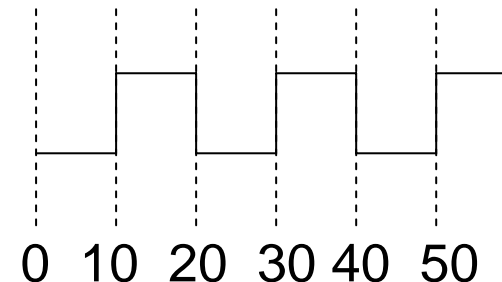
Time	statement executed
0	m = 1
5	a = 1
10	x = 0
30	b = 0
35	y = 1
50	\$finish

Verilog-HDL : Behavioral Modeling

Always Statement

- 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
```



Verilog-HDL : Behavioral Modeling

Exercises

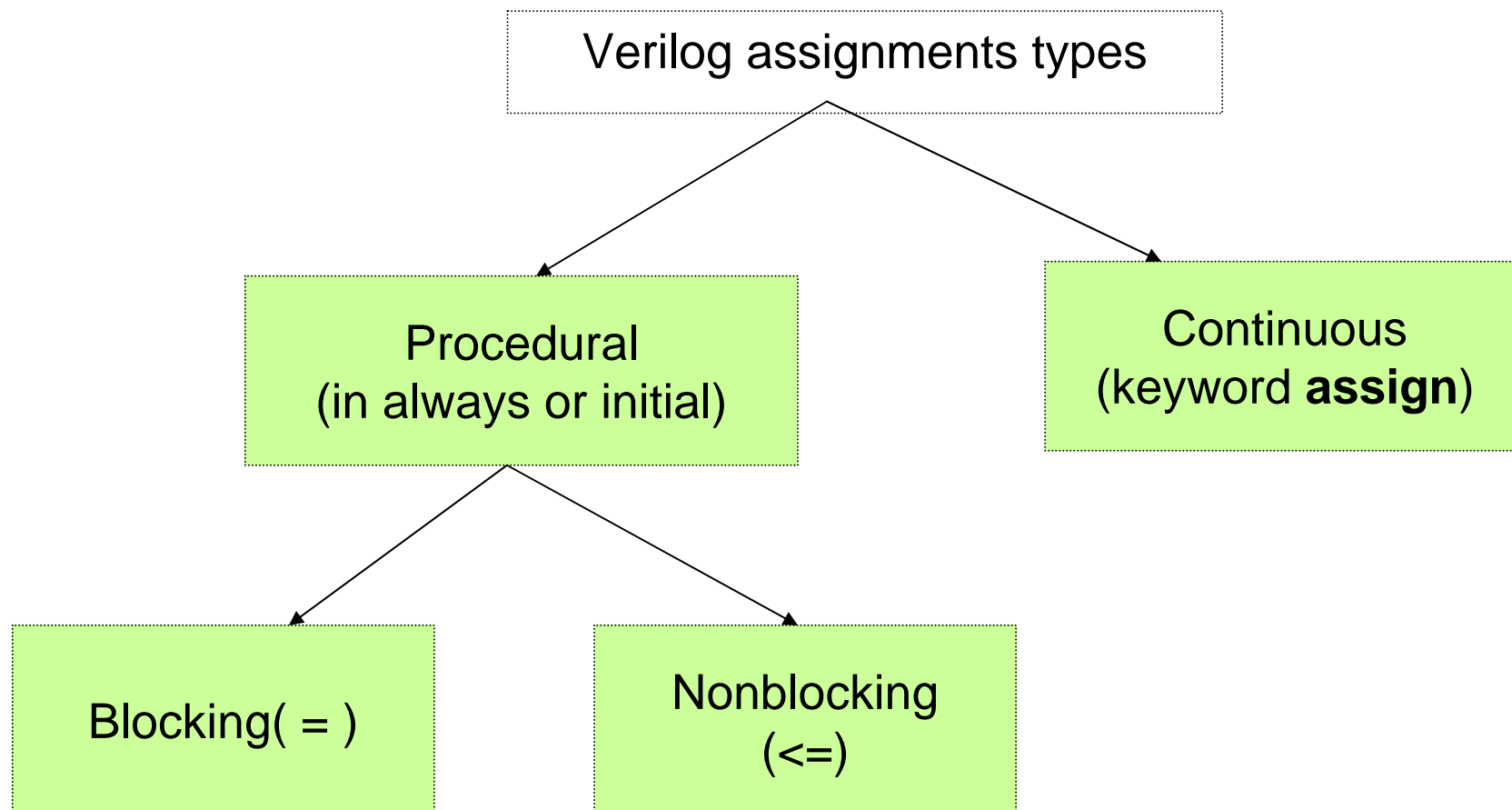
1. What happens if the following code is simulated

```
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-HDL : Behavioral Modeling

✚ Verilog Assignments Types



Verilog-HDL : Behavioral Modeling

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 :
 - <lvalue> = <expression> or***
 - <lvalue> <= <expression>***
 - 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**

Verilog-HDL : Behavioral Modeling

Blocking Statement

- 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

Verilog-HDL : Behavioral Modeling

Nonblocking Statement

- 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.

Verilog-HDL : Behavioral Modeling

Nonblocking과 Blocking Assignment 비교

initial

begin

$x = 0; y = 1; z = 1;$

$count = 0;$

$reg_a = 16'b0; reg_b = reg_a; \leftarrow 0 \text{ unit에 update}$

$reg_a[2] \leq \#15 1'b1; \leftarrow 15 \text{ unit 뒤 에 update}$

$reg_b[15:13] \leq \#10 \{x, y, z\} \leftarrow 10 \text{ unit 뒤 에 update}$

$count = count + 1; \leftarrow 0 \text{ unit에 update}$

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)

Verilog-HDL : Behavioral Modeling

✚ Nonblocking Statement에서 Read, Write 문 분리

```
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)

Verilog-HDL : Behavioral Modeling

Race Condition

```
// illustration 1 : blocking  
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 $a \leftarrow b \leftarrow a$

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

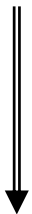
Race condition is eliminated.

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

Verilog-HDL : Behavioral Modeling

✚ 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
```

Verilog-HDL : Behavioral Modeling

Nonblocking Statement 특징

- 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

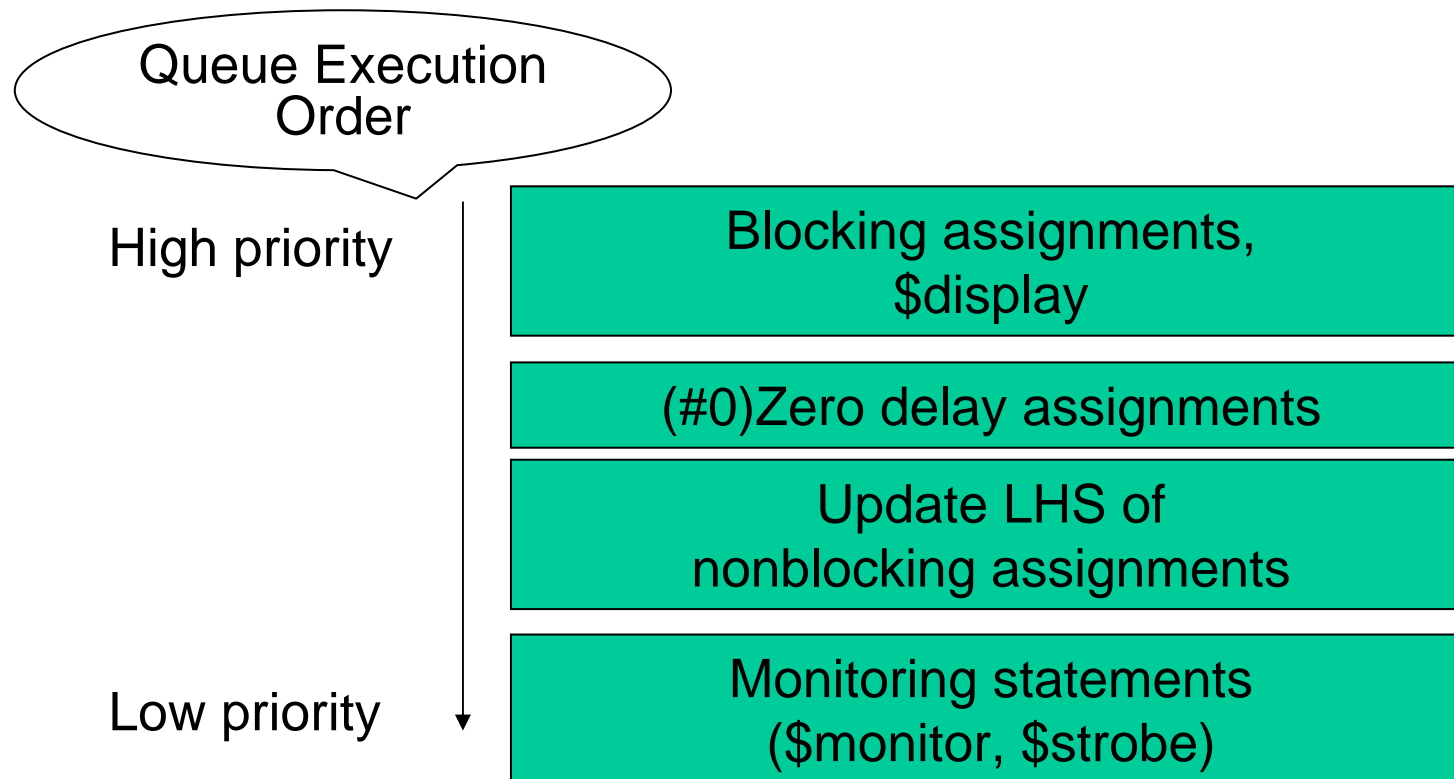
Verilog-HDL : Behavioral Modeling

Signal Statement 사용에 대한 가이드

- 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 LHS variable in different always / initial blocks => to prevent race condition

Verilog-HDL : Behavioral Modeling

Verilog Event Queue Priority

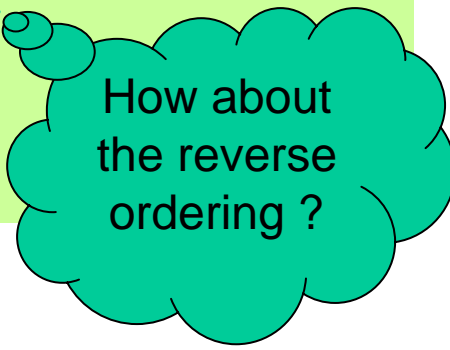


Verilog-HDL : Behavioral Modeling

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
```



How about
the reverse
ordering ?

```
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
```

Verilog-HDL : Behavioral Modeling

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
```

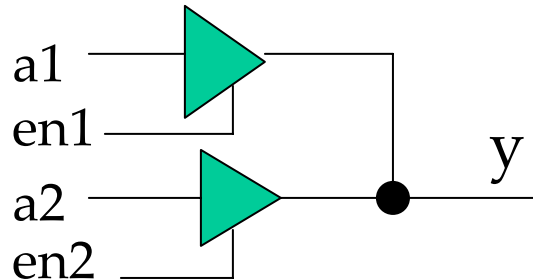
```
module osc1 (clk);
    output clk
    reg clk;
    initial #5 clk = 0;
    always @(clk) #5 clk <= ~clk;
endmodule
```

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

Verilog-HDL : Behavioral Modeling

✚ 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
  
```

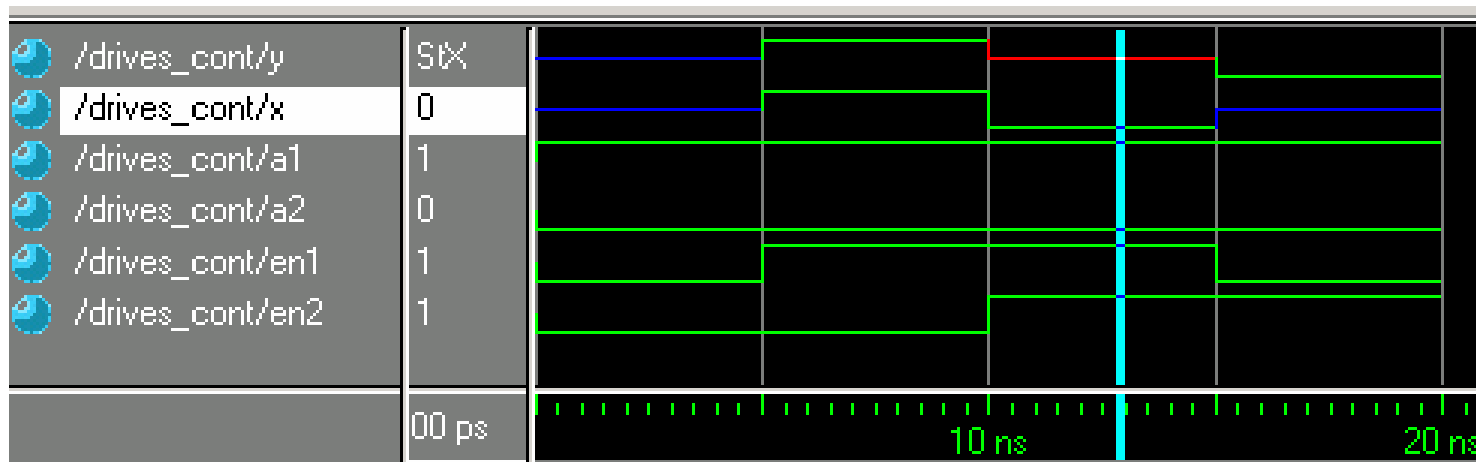
Verilog-HDL : Behavioral Modeling

✚ Answer 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
```



Verilog-HDL : Behavioral Modeling

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

Verilog-HDL : Behavioral Modeling

+ Delay-Based Timing Control

- Delays are Specified Explicitly in the Procedural Statements
- Syntax :
 $\text{<delay> ::= \#<number> | \#<identifier> | \#(<min_typ_max_exp>, << min_typ_max_exp>>*)}$
- 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

Verilog-HDL : Behavioral Modeling

Delay Control의 세가지 형태

- 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 ;
- Zero Delay control
 - #0 *procedural_statements*
 - Example) #0 x = y
 - Objective : To Eliminate the race condition between different always-initial blocks at time 0

Verilog-HDL : Behavioral Modeling

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
```

Verilog-HDL : Behavioral Modeling

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
```

Verilog-HDL : Behavioral Modeling

Zero Delay Control

<code>initial</code>	<code>initial</code>
<code>begin</code>	<code>begin</code>
<code> x = 0;</code>	<code> #0 x = 1;</code>
<code> y = 0;</code>	<code> #0 y = 1;</code>
<code>end</code>	<code>end</code>

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.

Verilog-HDL : Behavioral Modeling

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
```

regular delay : 문장 전체의 접근을 지연시킴 (RHS 자체의 read를 지연)

intra assignment delay : RHS 가 LHS로 전달되는 것을 지연시킴

blocking : LHS가 update된 후에야 다음 문장으로 제어가 넘어감

non-blocking : RHS에 대한 eval(read)만 일어난 후에 다음 문장으로 제어가 넘어감

```
// 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
```

Verilog-HDL : Behavioral Modeling

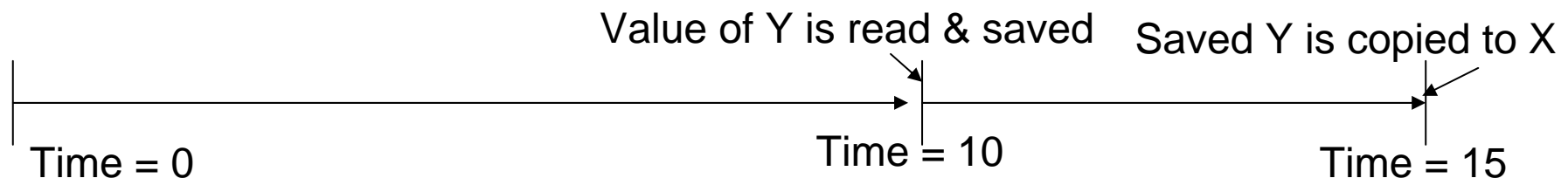
✚ Hints for the Answer : Blocking

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)



Verilog-HDL : Behavioral Modeling

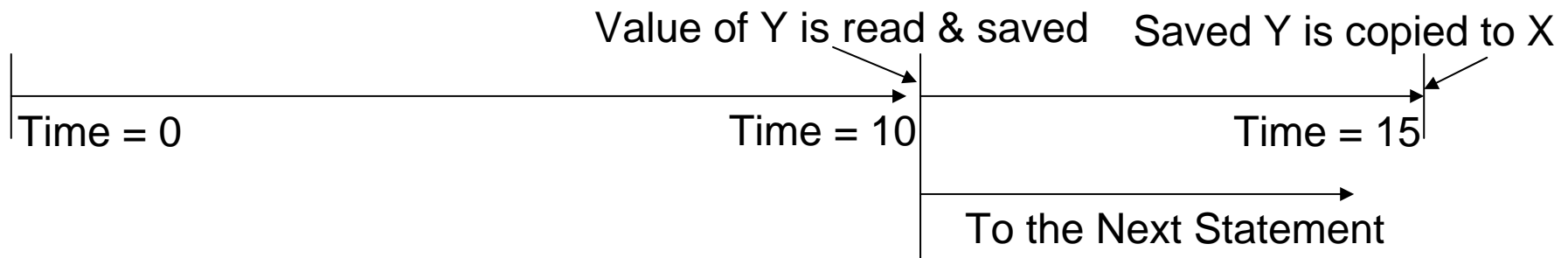
+ Hints for the Answer : Nonblocking

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



Verilog-HDL : Behavioral Modeling

Answer (1)

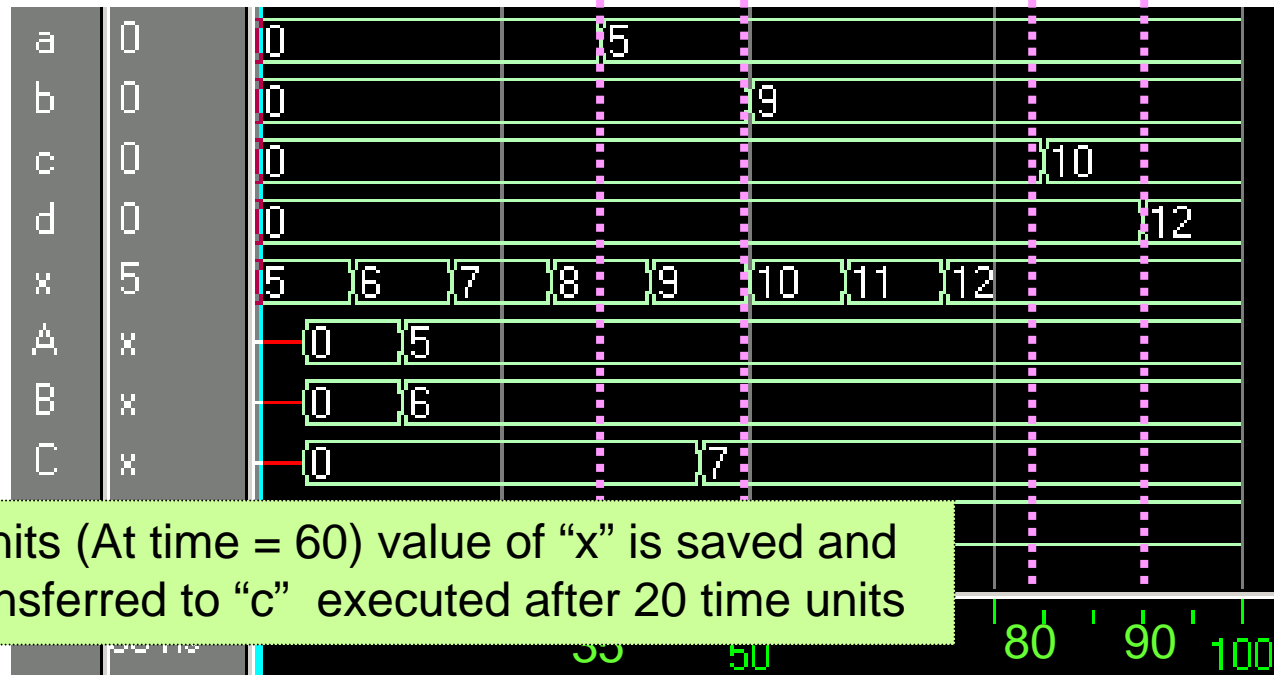
Ensures the x value is initialized at time = 0

The value of "x" at time = 0 is saved in temporal storage and will be assigned to "a" at time = 35

15 time units after the nonblocking assignment of a = x is completed (at time 50 = 35+15), b = x is executed

RHS eval time
0
50
80
90

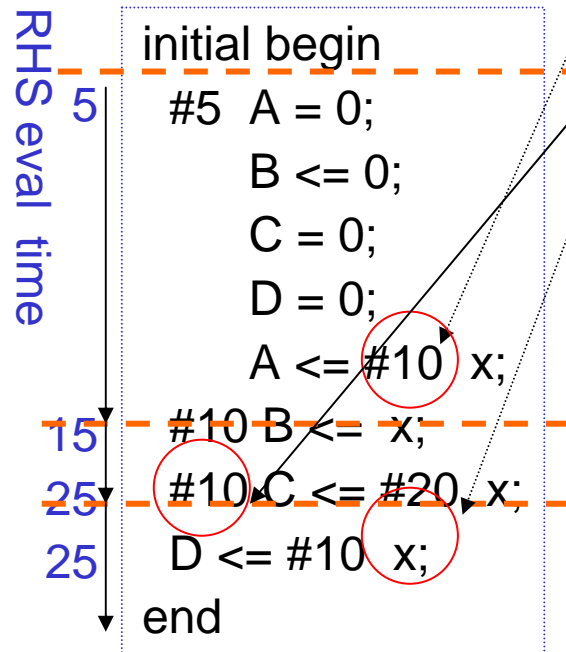
```
initial begin
  #0 a = 0;
    b = 0;
    c = 0;
    d = 0;
  #35 a = x;
  #15 b = x;
  #10 c = x;
  #10 d = x;
end
```



After 10 time units (At time = 60) value of "x" is saved and the value is transferred to "c" executed after 20 time units

Verilog-HDL : Behavioral Modeling

Answer (2)

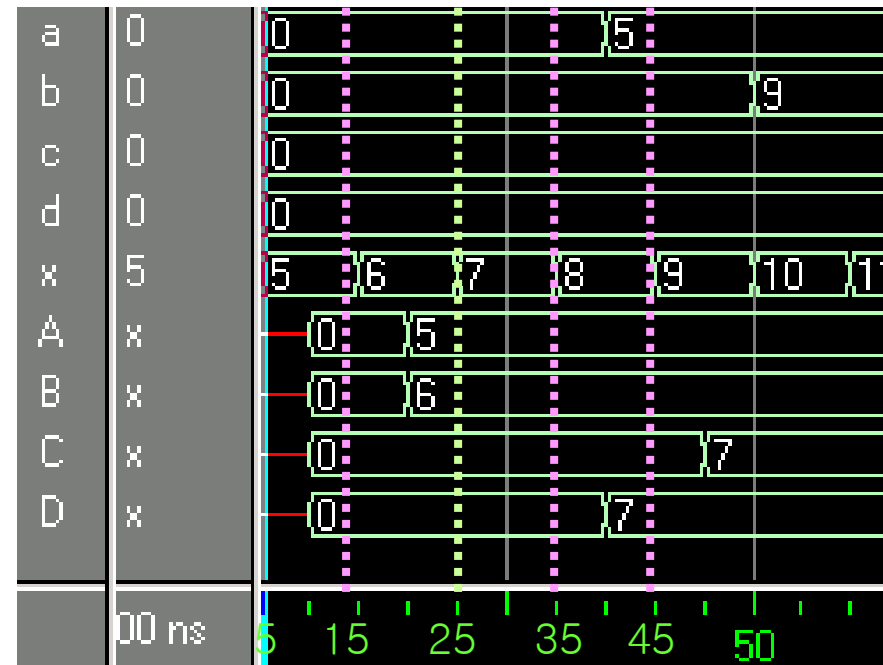


The value of x at time = 5 is saved in temporal storage and transferred to A after 10 time units

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

The value of x (7) is read

non blocking문장이 나오면 더 복잡.
time =5 에서 A가 update되기 전에 다음문장 (#10 B <= x)로 제어가 넘어간다. (A의 update는 time=15에서 이루어진다.)
time = 15 에서 x의 값이 읽히고 스케줄된다.
time =25 에서 x의 값이 읽히고 스케줄된다. 이 값이 전달되는 것은 time=45 이다..



Verilog-HDL : Behavioral Modeling

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*)

Verilog-HDL : Behavioral Modeling

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
```

Verilog-HDL : Behavioral Modeling

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]};
```

Verilog-HDL : Behavioral Modeling

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
```

Verilog-HDL : Behavioral Modeling

✚ Level-Sensitive Timing Control - 1

- 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
```

Verilog-HDL : Behavioral Modeling

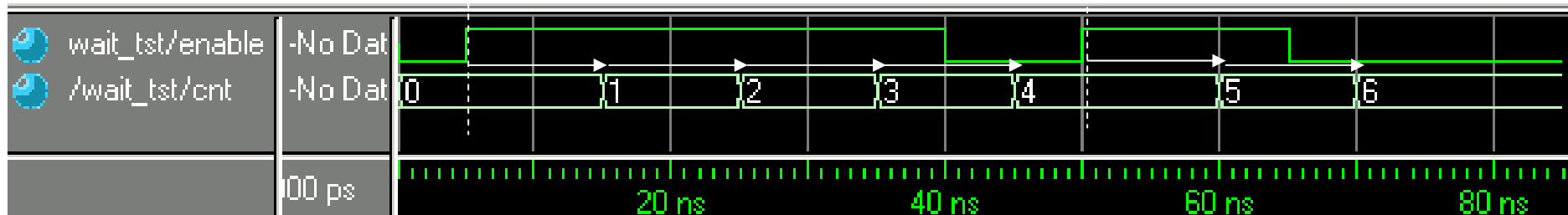
✚ Level-Sensitive Timing Control - 2

```
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
```



Verilog-HDL : Behavioral Modeling

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**

Verilog-HDL : Behavioral Modeling

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

Verilog-HDL : Behavioral Modeling

Multiway Branch : casex, casez

- If expr contains x and z value, default is matched
- Csaex and 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'bxx1x 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;

Verilog-HDL : Behavioral Modeling

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

```

```

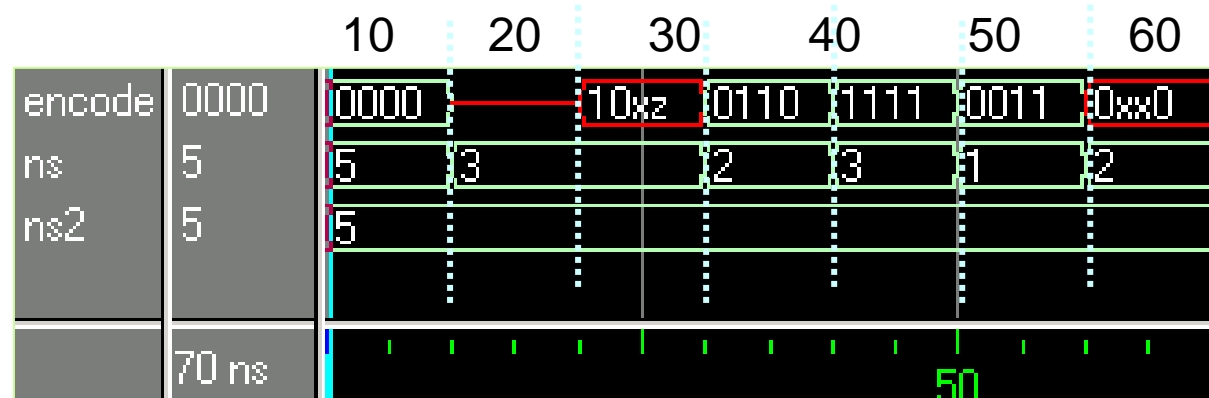
always @(encode)
begin
  casex (encode)
    4'b1xxx : ns = 3;
    4'bx1xx : ns = 2;
    4'bxx1x : ns = 1;
    4'bxxx1 : ns = 0;
    default : ns = 5;
  endcase
endcase

```

```

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

```



Verilog-HDL : Behavioral Modeling

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
```

Verilog-HDL



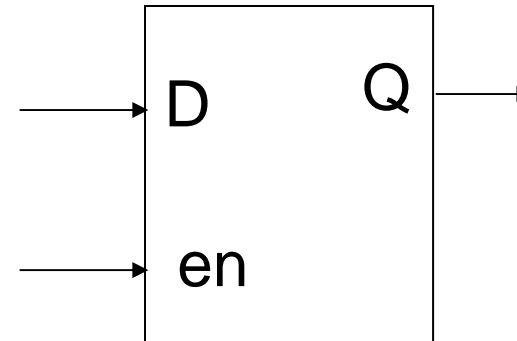
Agenda

- Basics
- Modules & Ports
- Verilog Simulation
- Gate Level Modeling
- Dataflow Modeling
- Behavioral Modeling
- **Application to Synchronous Logic**
 - Latches
 - Flip-Flops & Registers
 - Shift Registers
 - Counters
 - BCD Counters & Cascading Counters
- FSM Design
- Parameterized Design
- More on Blocks
- Tasks & Functions
- Logic Synthesis

Verilog-HDL : Application to Synchronous Logic

Level Sensitive Latch

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



Verilog-HDL : Application to Synchronous Logic

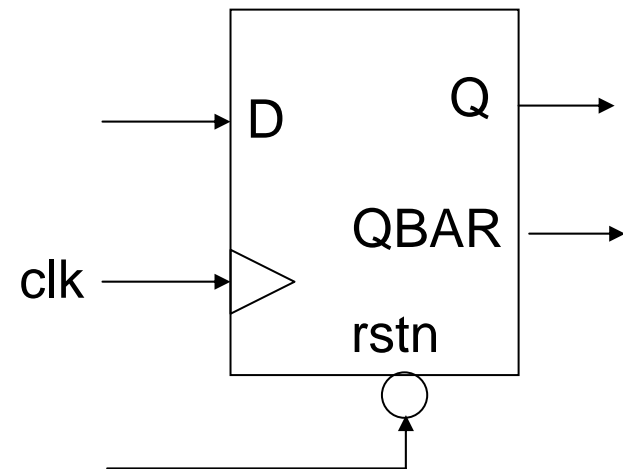
D Flip-Flops - 1

```
// FlipFlop with asynchronous 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
```



Verilog-HDL : Application to Synchronous Logic

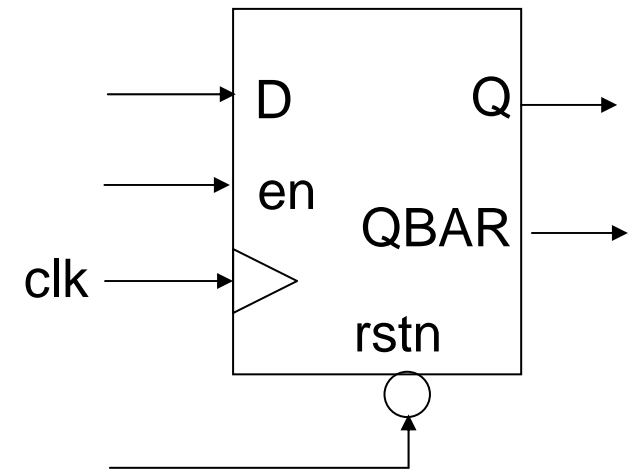
D Flip-Flops - 2

```
// 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
```



Verilog-HDL : Application to Synchronous Logic

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;
```

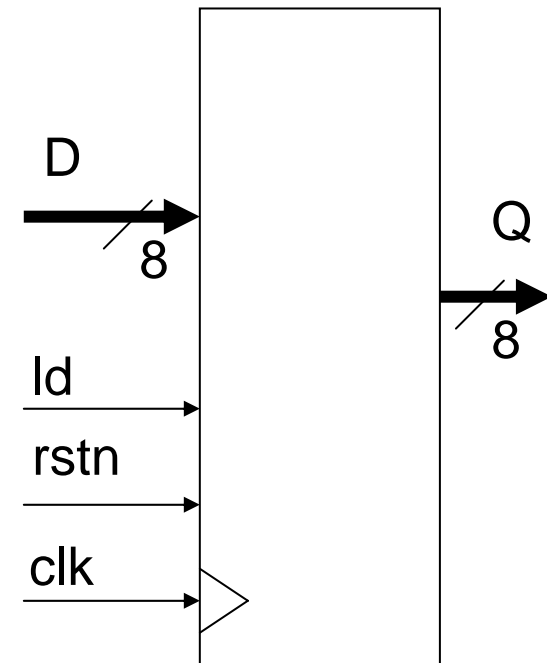
```
reg [7:0] Q;
```

```
always @(posedge clk )
```

```
if (!rstn) Q <= 8'd0;
```

```
else if (ld) Q <= D;
```

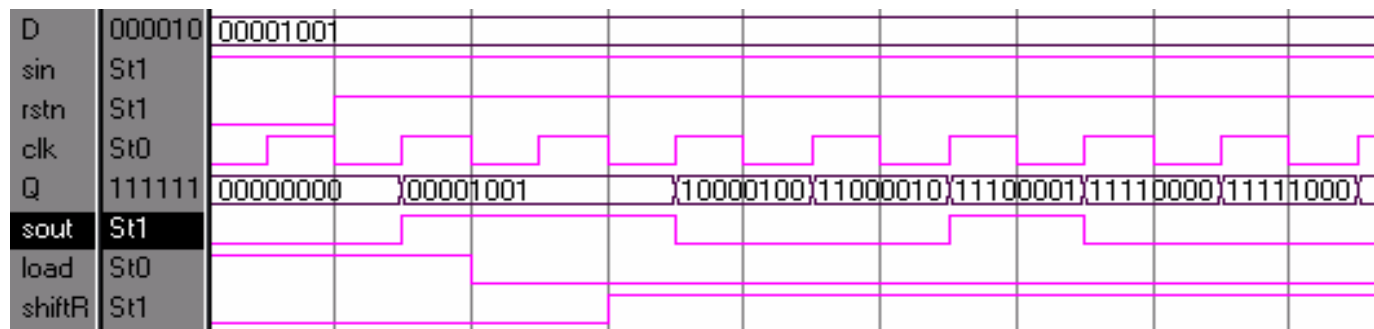
```
endmodule
```



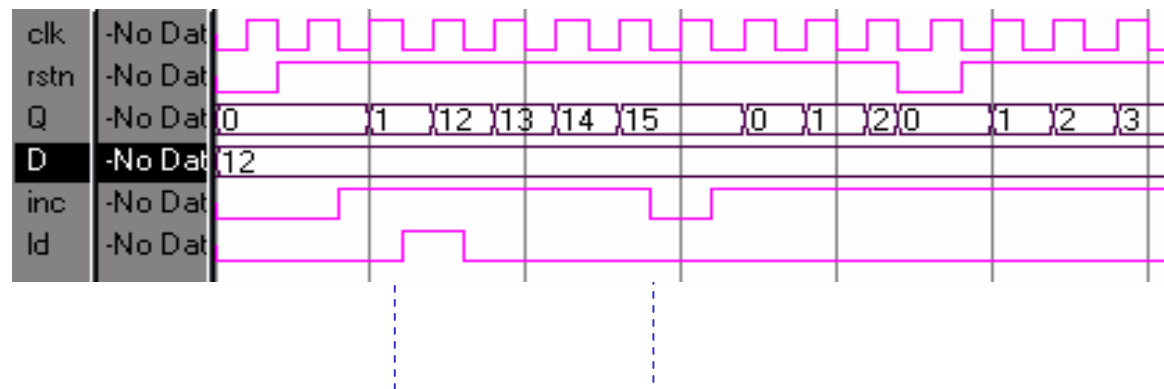
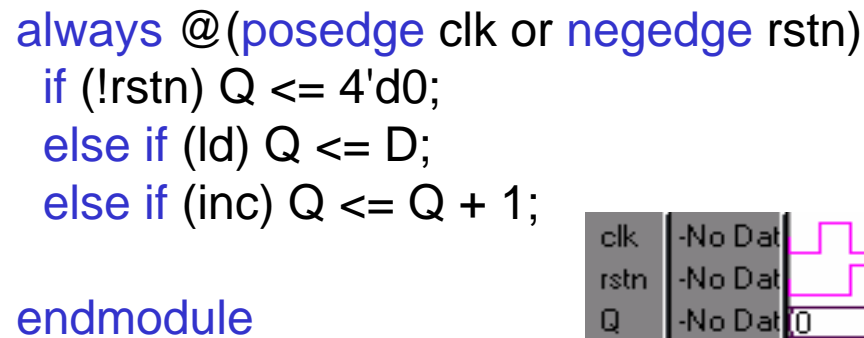
Verilog-HDL : Application to Synchronous Logic

Shift Registers (PISO type)

```
// Shift register with asynchronous reset and synchronous 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
```



```
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;
```



Verilog-HDL : Application to Synchronous Logic

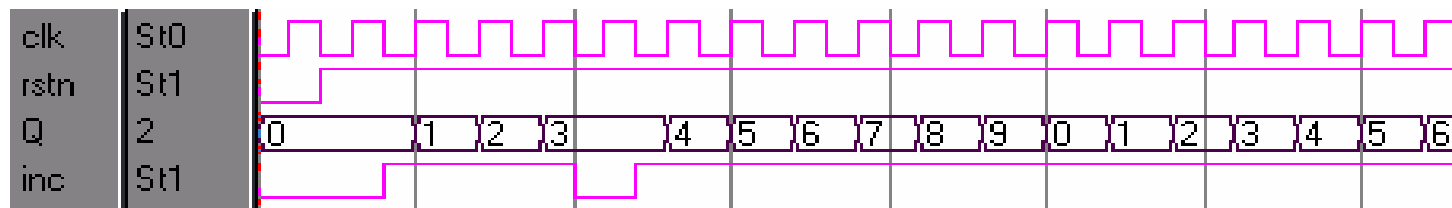
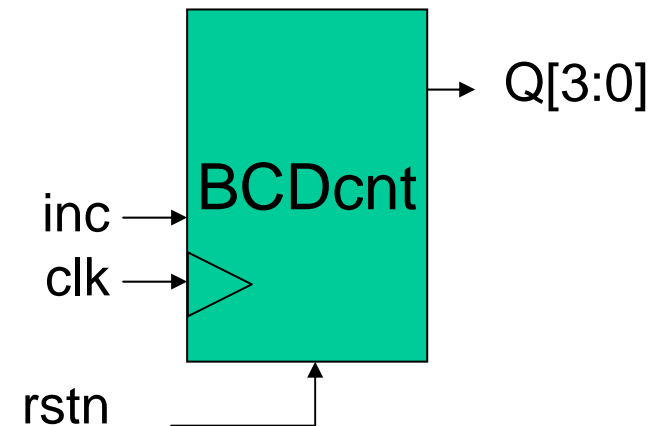
BCD Counter

```

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
    
```



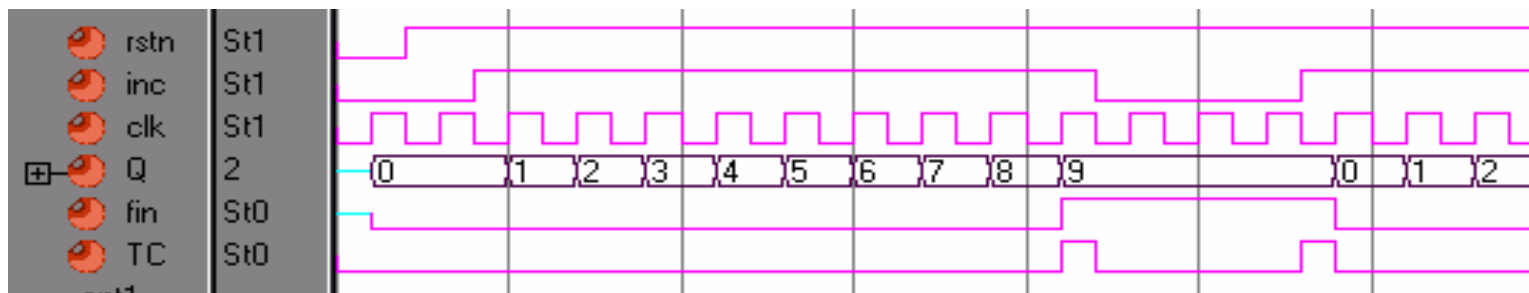
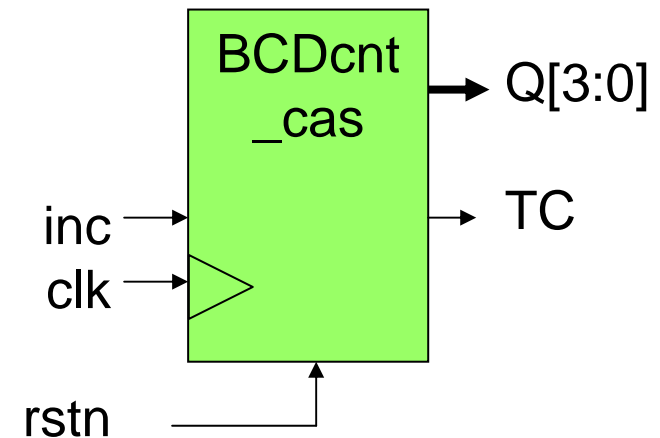
Verilog-HDL : Application to Synchronous Logic

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
    
```



Verilog-HDL : Application to Synchronous Logic

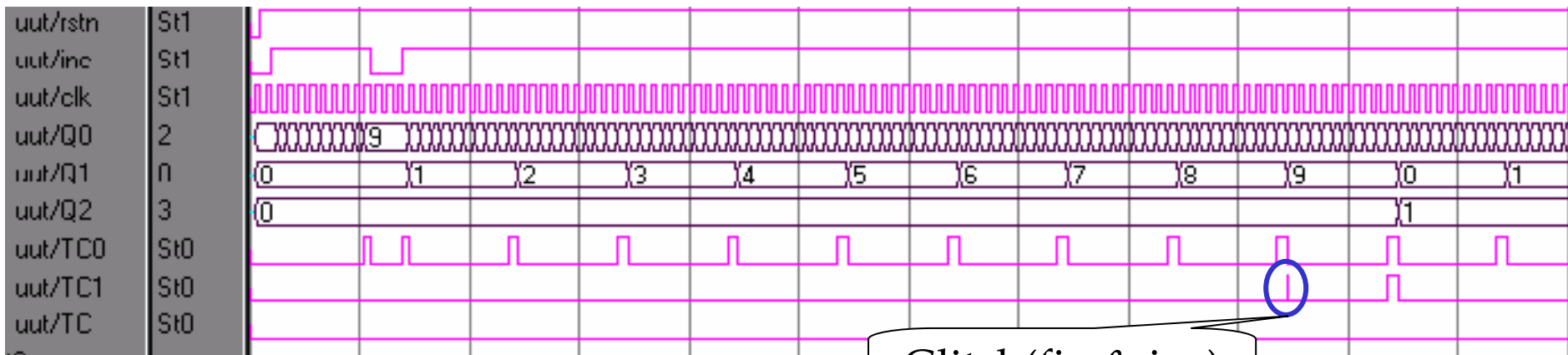
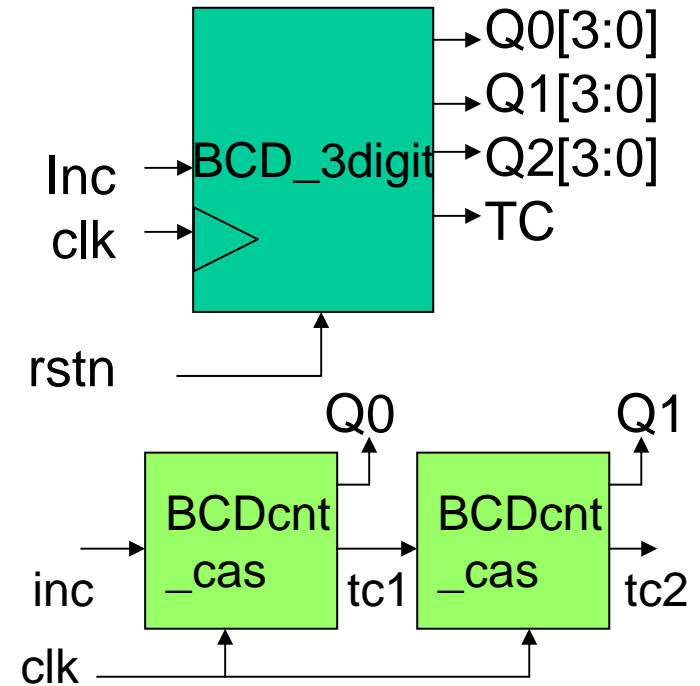
✚ Cascading 3-digit BCD Counters

```

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);

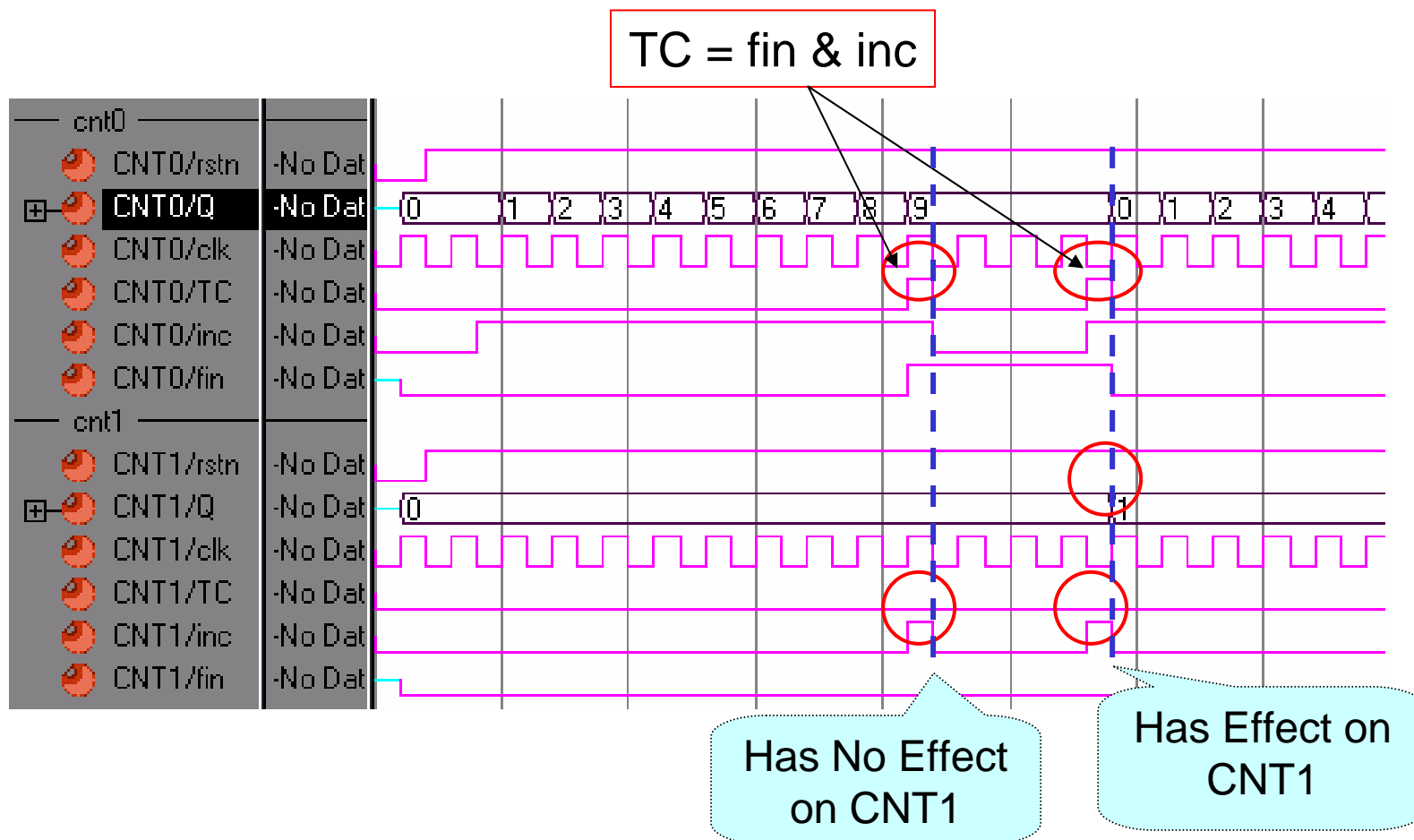
endmodule
    
```



Glitch(fin & inc)

Verilog-HDL : Application to Synchronous Logic

+ Cascading 3-digit BCD Counters - 동작



Verilog-HDL : Application to Synchronous Logic

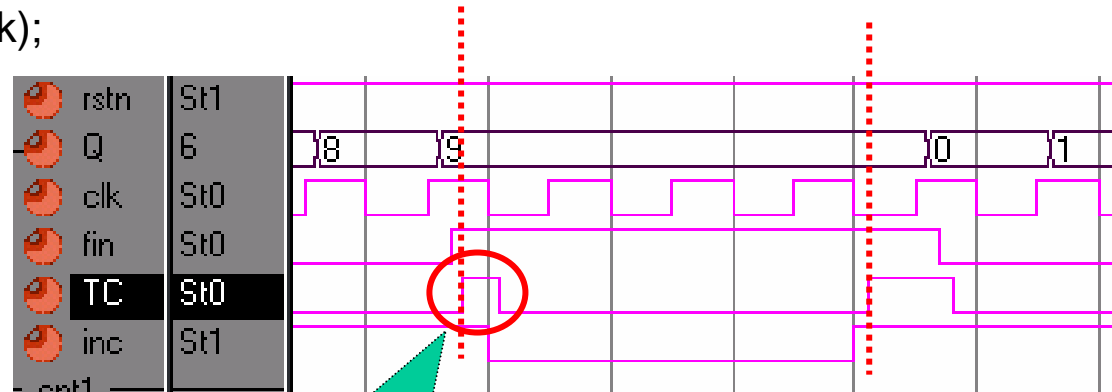
❖ Cascading 3-digit BCD Counters – Gate Delay를 고려한 경우

```

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

```



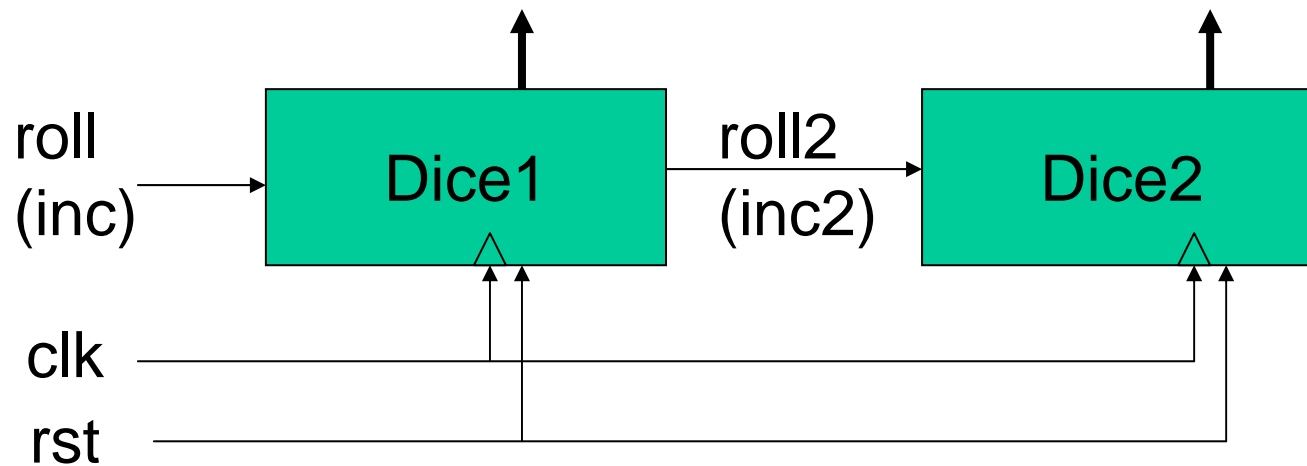
- Explicit Delay Specification Clarifies the Precedence between signals (Delay helps to figure out which one precedes the other)
 - It Makes Clear if a glitch is harmless or not (effective)

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

Verilog-HDL : Application to Synchronous Logic

✚ Exercises (1)

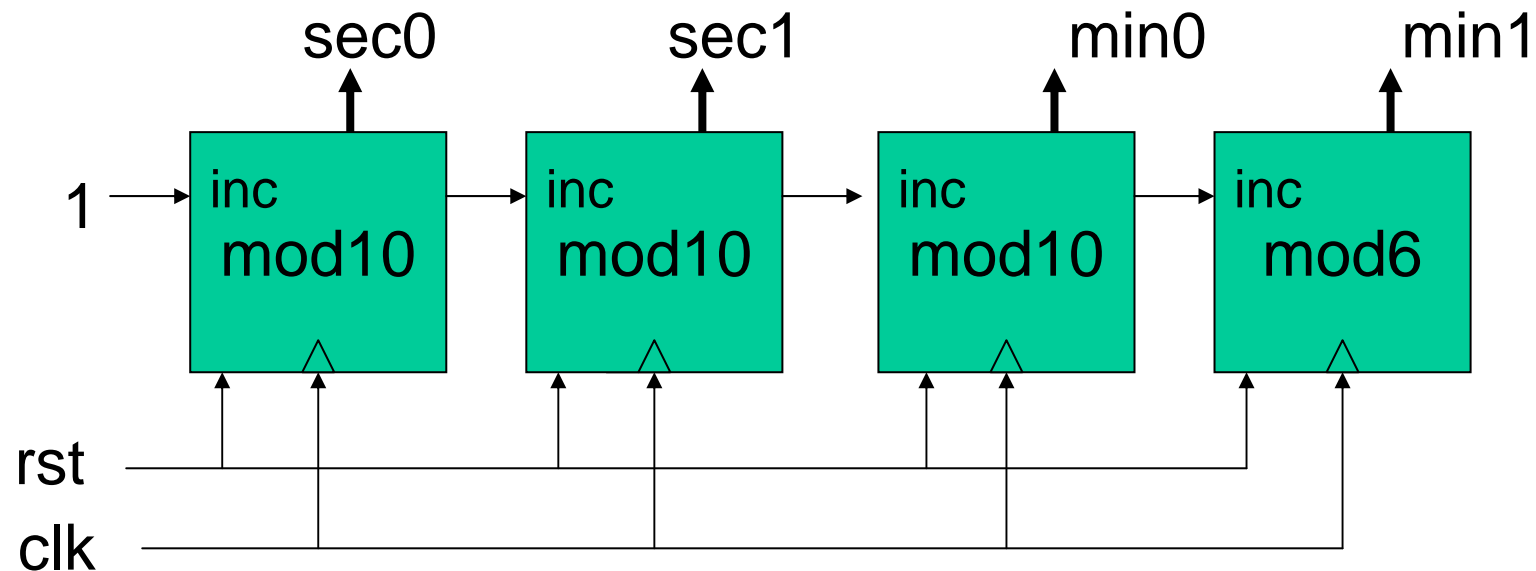
1. Design a Dice (mod-6 counter) that counts from 1 to 6.
2. Cascade the two Dice designed above Question . If one dice rolls 6 times the next dice increments by one.



Verilog-HDL : Application to Synchronous Logic

✚ Exercises (2)

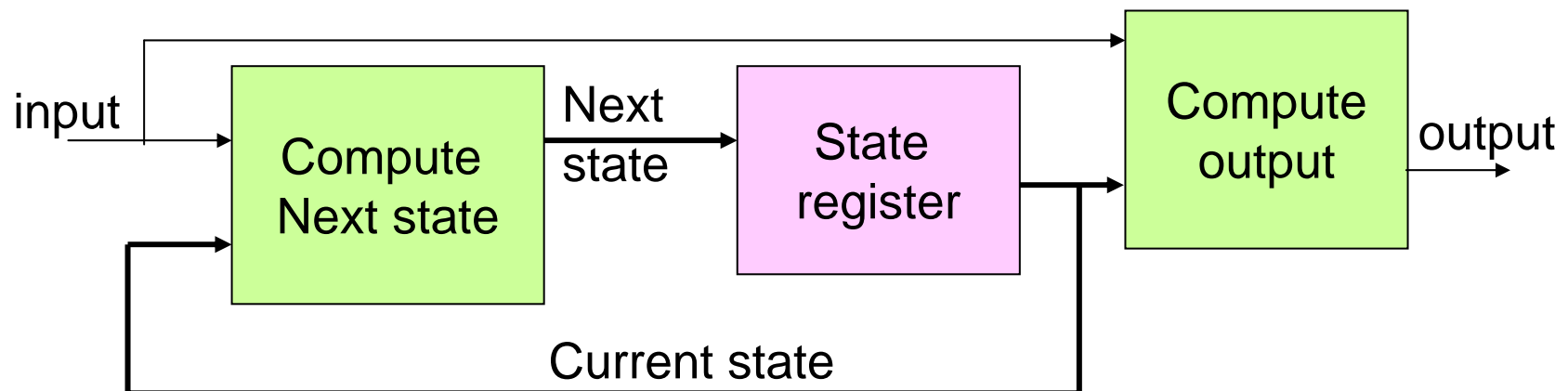
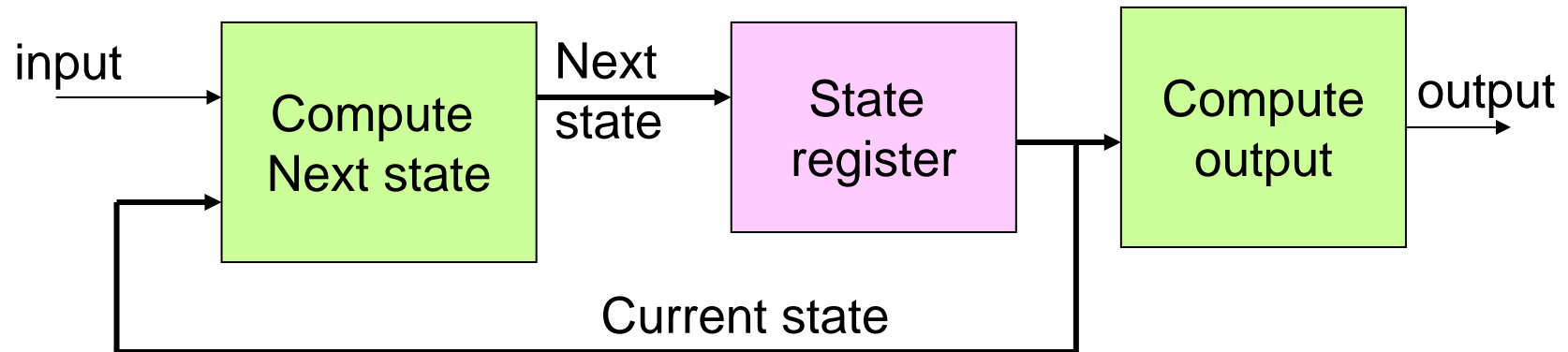
3. Design a clock circuit which has 2 digit for second and 2 digits for minute. Design the logic in a synchronous fashion (NOT Asynchronous manner). Assume that the input clock is 25MHz.



Verilog-HDL : FSM Design

FSM (Finite State Machine) 설계

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



Verilog-HDL : FSM Design

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_STATE : if (inputs == 1 )
                    ns = `WAIT_STATE ;
                else ns = `INIT_STATE;
`WAIT_STATE : ...
`RUN_STATE:    ...
default : ns <= `INIT_STATE
endcase
```

Verilog-HDL : FSM Design

Synchronous FSM State Register의 두가지 형태

- Synchronous Reset logic in next_state computation circuit

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

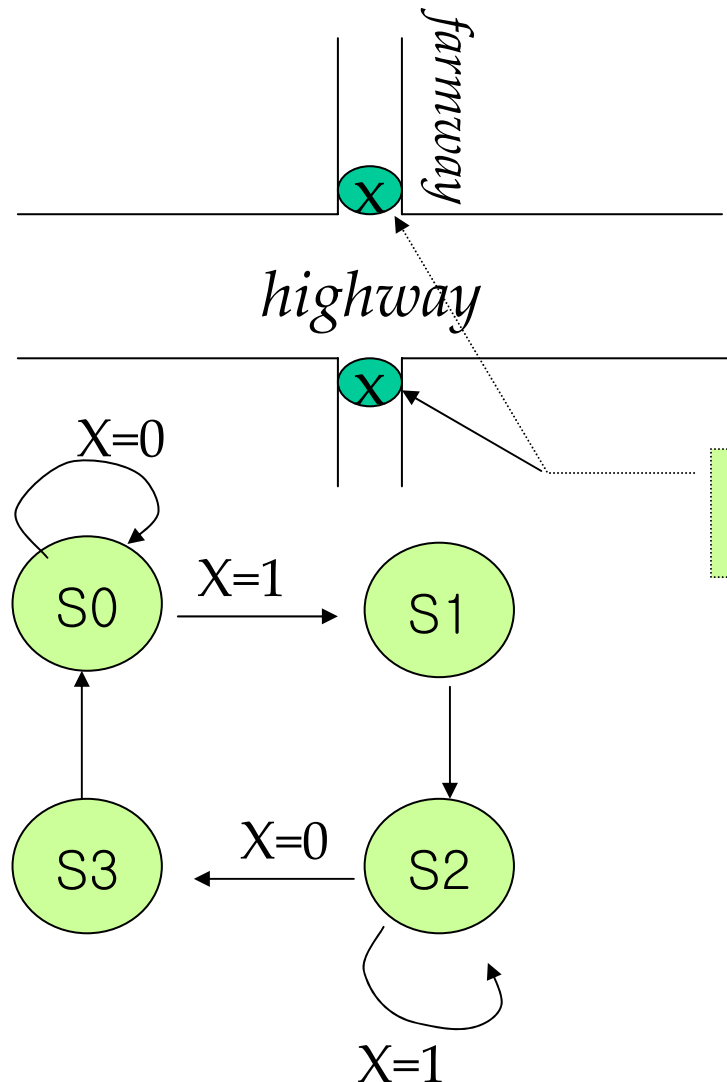
always @(current_state or inputs or reset) begin
    if(reset == 1) next_state <= `INIT_STATE;
    else case (current_state)
        `INIT_STATE :
            if(inputs == 1)
                next_state = `WAIT_STATE;
            else next_state = `INIT_STATE;
        `WAIT_STATE : ...
        `RUN_STATE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

- Synchronous Reset in the State Register

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

Verilog-HDL : FSM Design

예제: Traffic Light Controller



X : sensor that indicates if there is a traffic in the farmway

Output Table (Moore type)

	highway	farmway
S0	Green	Red
S1	Yellow	Red
S2	Red	Green
S3	Red	Yellow

Verilog-HDL : FSM Design

예제: Traffic Light Controller – Verilog Code

```
// 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
```

State Reg Next state FSM Output

```
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
```

```
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
```

```
endmodule
```

Verilog-HDL : FSM Design

예제: TLC – Verilog Code (Synchronous Version)

FSM Output	<pre> 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 </pre>	State Reg	<pre> 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 </pre>

endmodule

Verilog-HDL : FSM Design

예제: Traffic Light Controller - Testbench

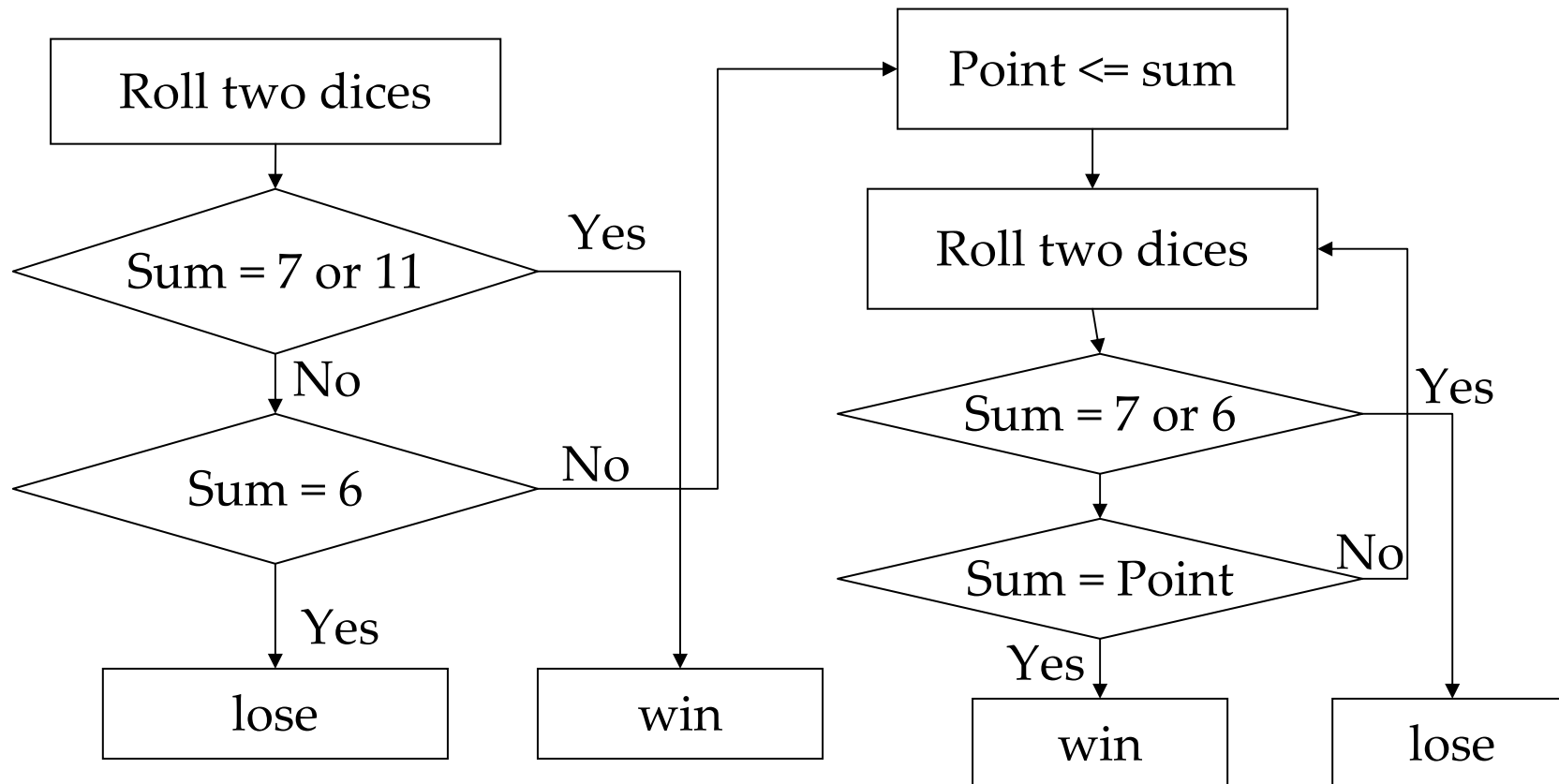
```
module testbench;
reg clk = 0;
reg rst ,X ;
wire [1:0] hwy, fwy;

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
```


Verilog-HDL : FSM Design

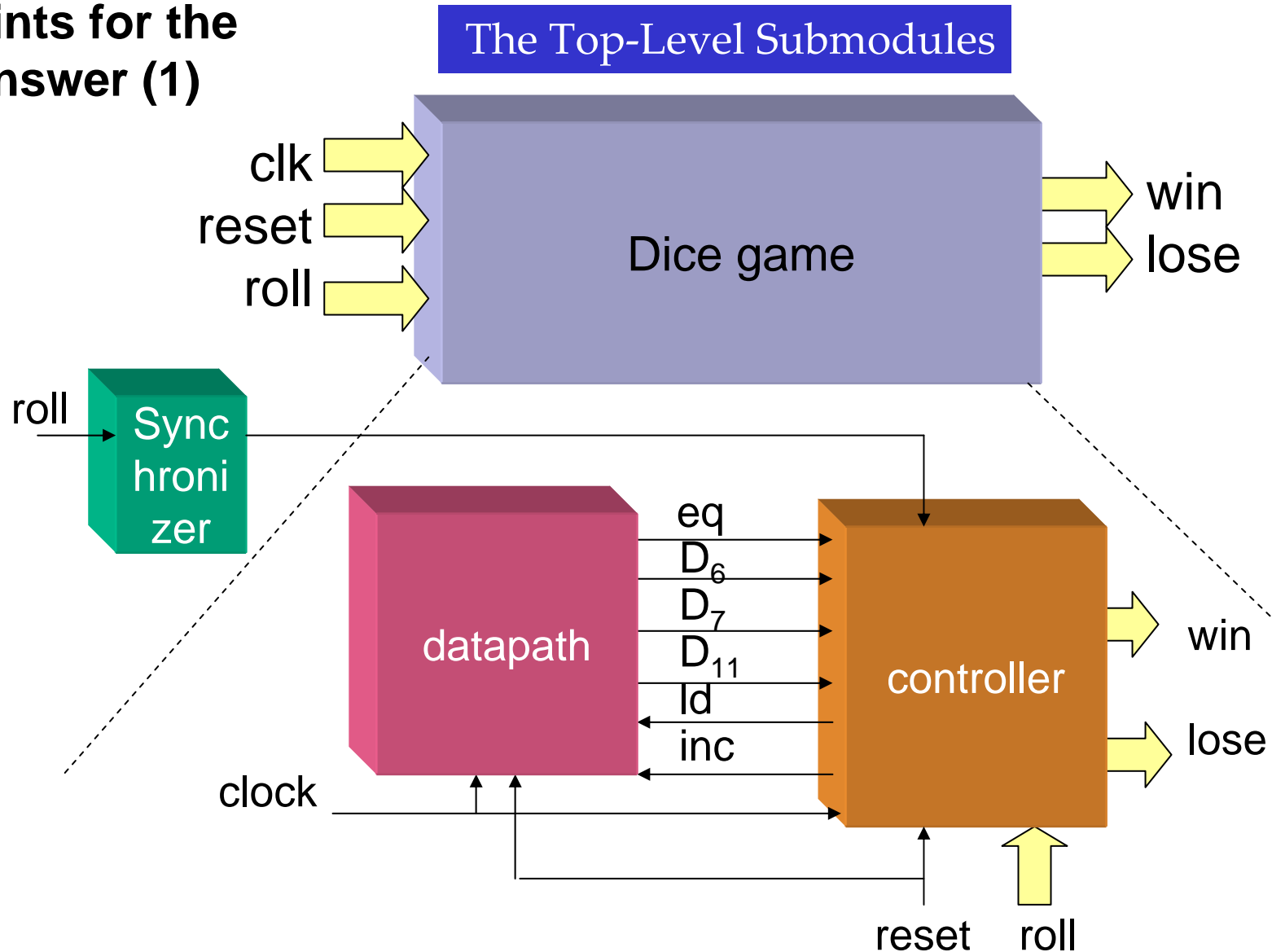
✚ Exercises (1)

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.



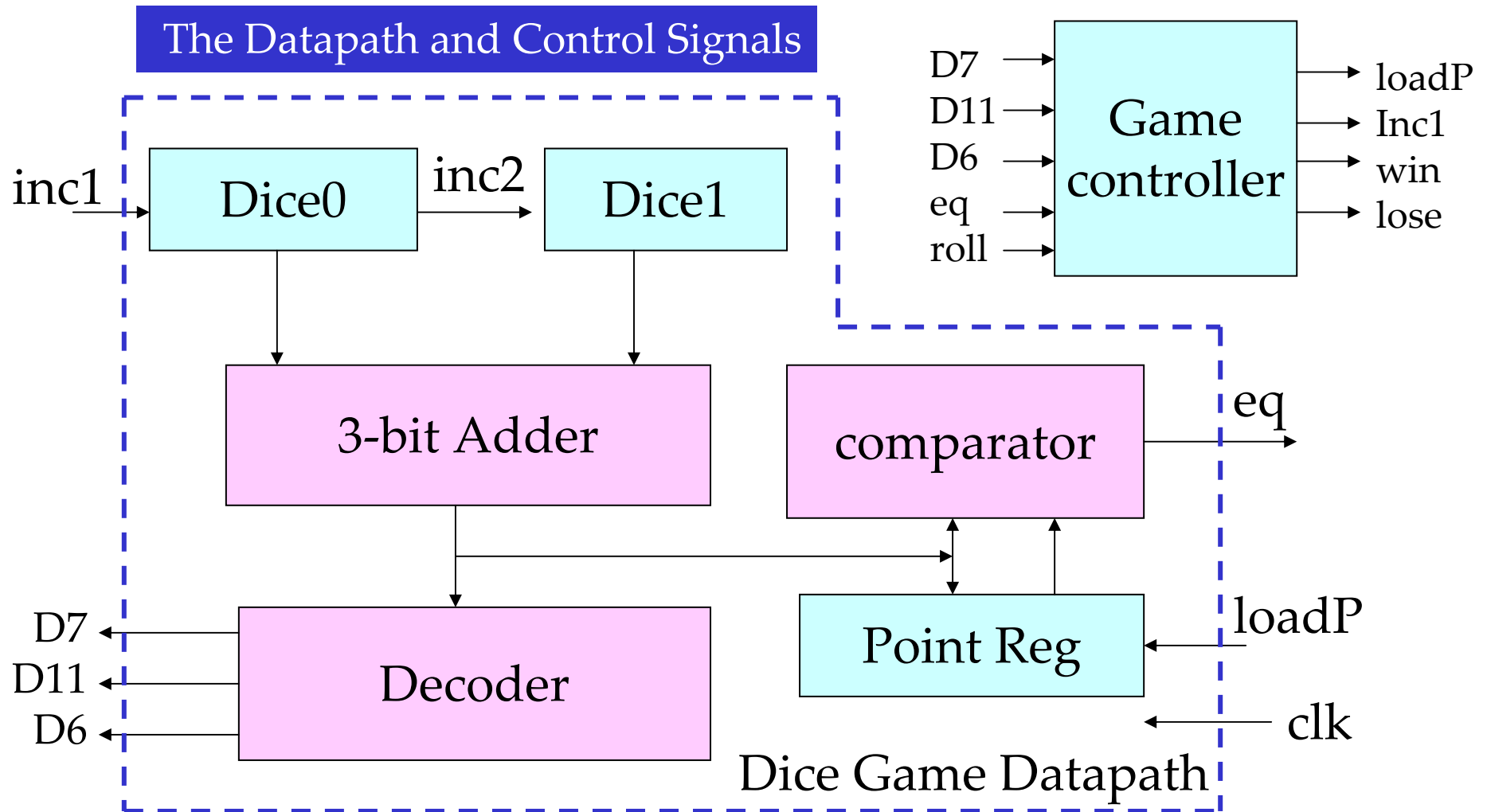
Verilog-HDL : FSM Design

✚ Hints for the Answer (1)



Verilog-HDL : FSM Design

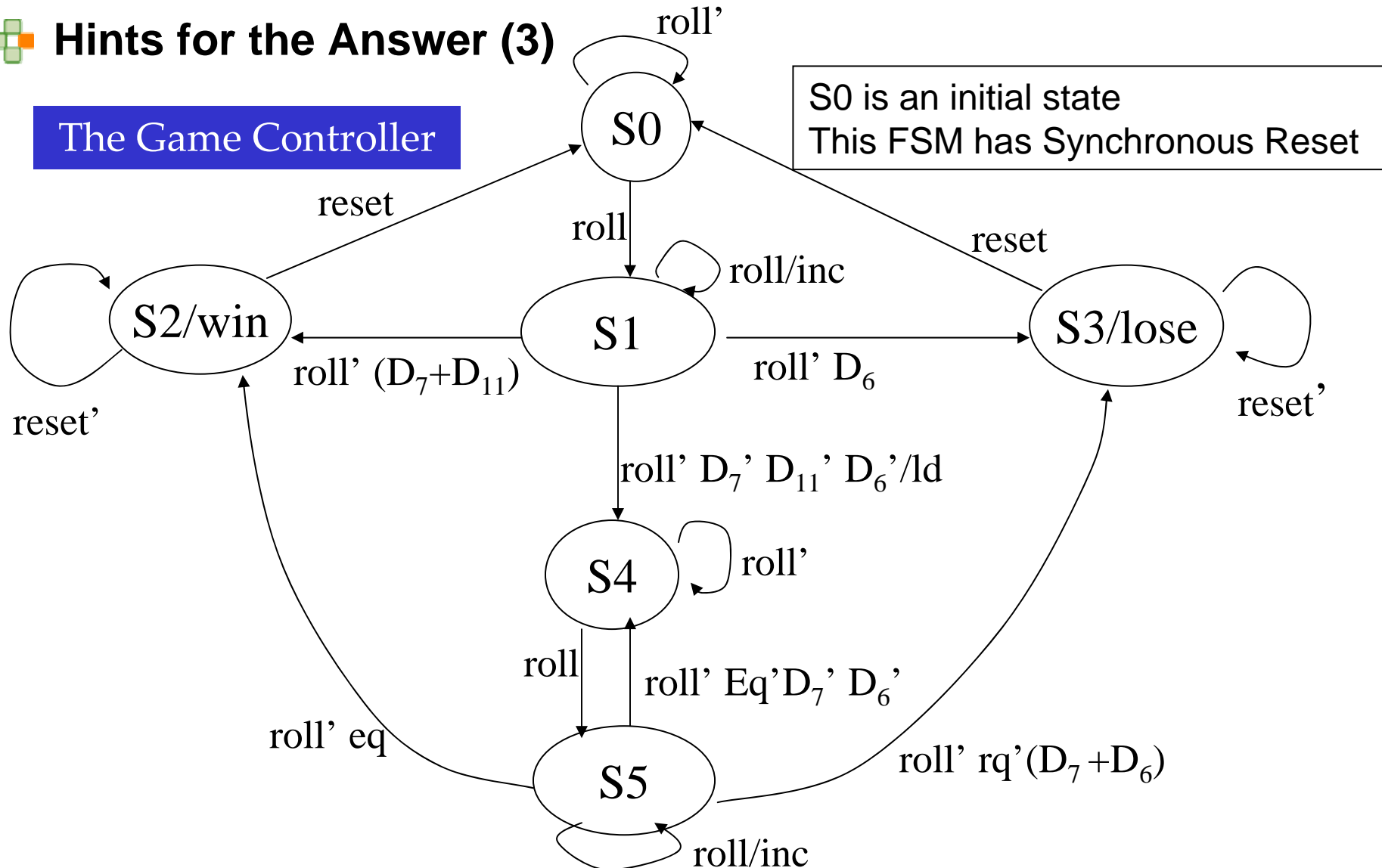
✚ Hints for the Answer (2)



Verilog-HDL : FSM Design

✚ Hints for the Answer (3)

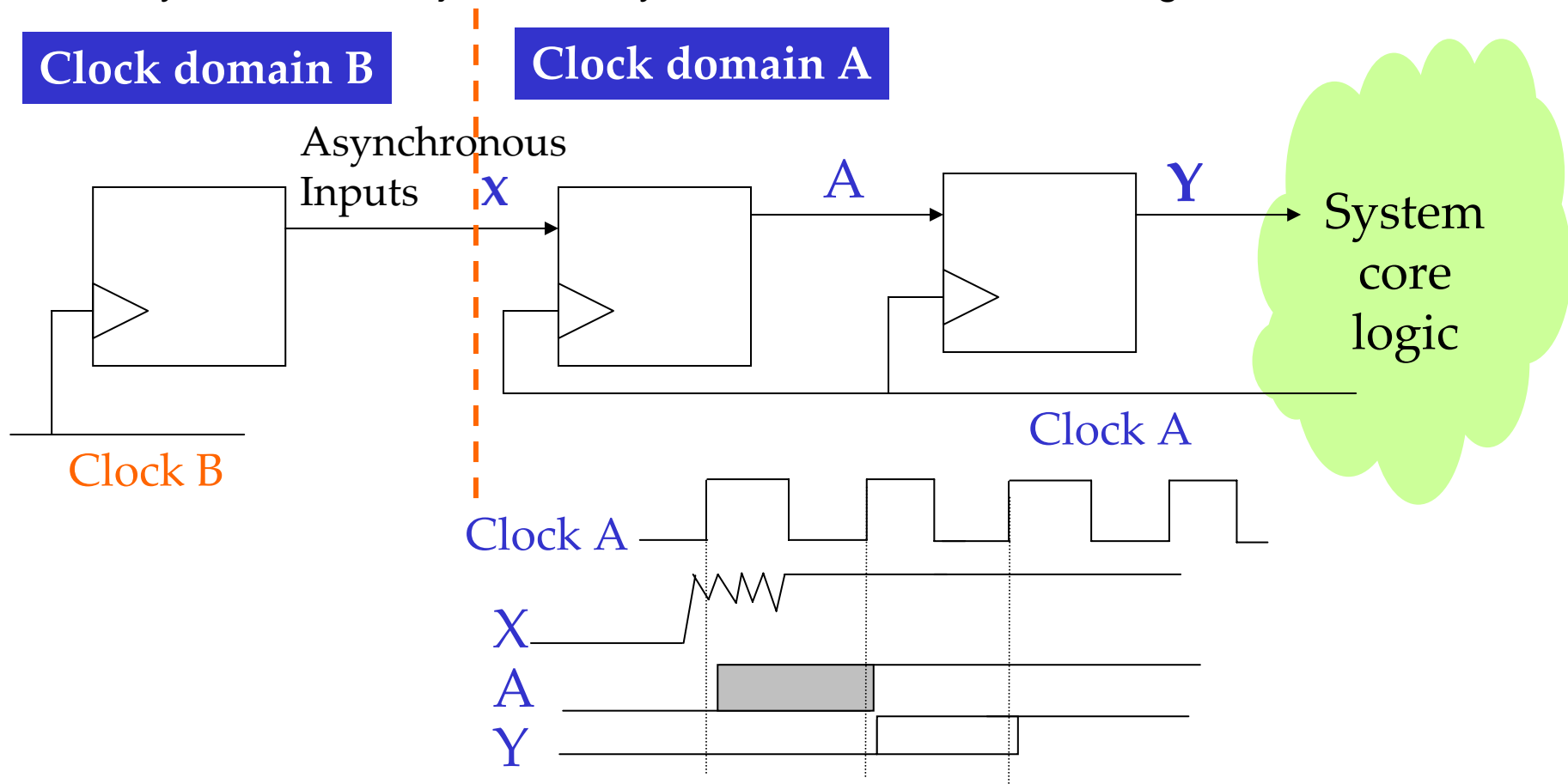
The Game Controller



Verilog-HDL : FSM Design

✚ Hints for the Answer (4) – Synchronizer Design

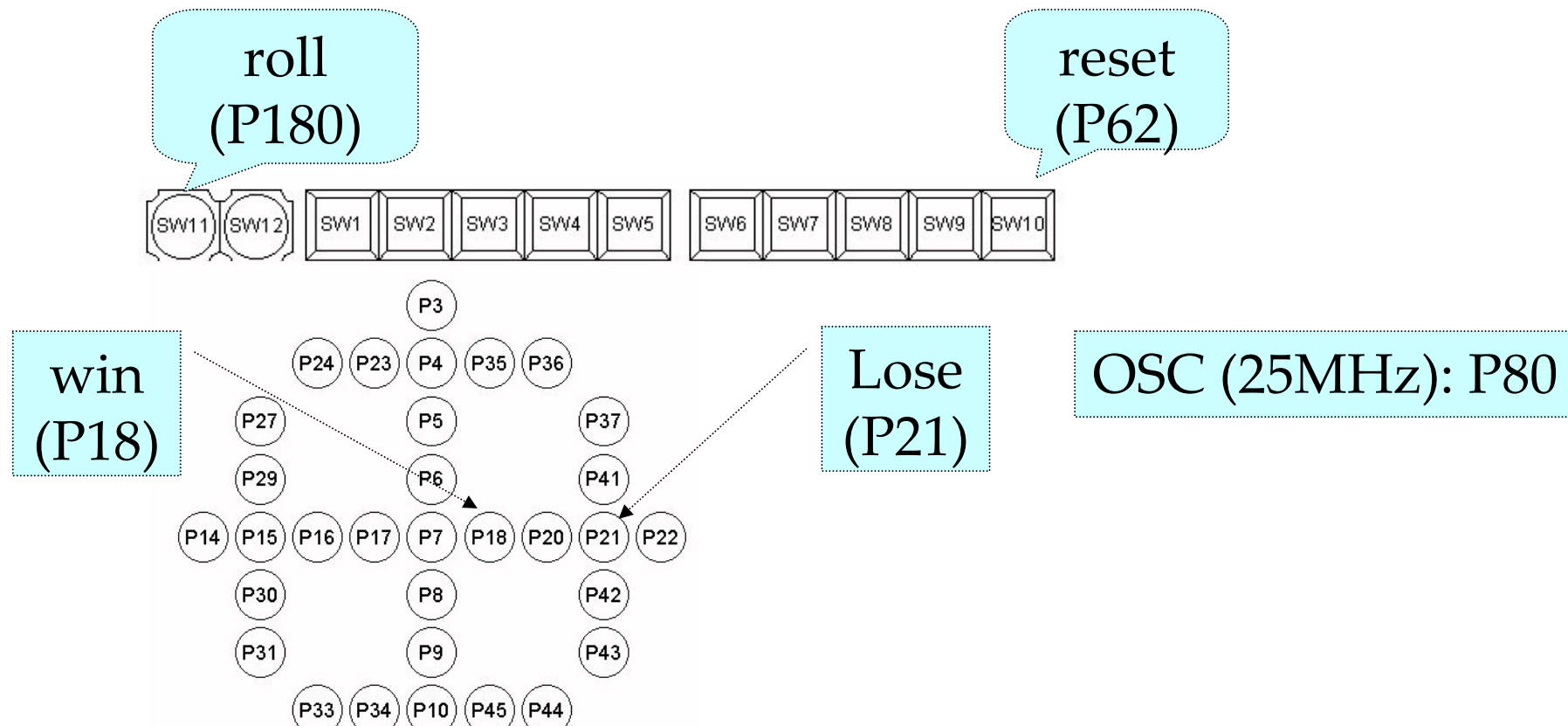
- Sometimes Asynchronous Inputs are unavoidable (e.g. User Inputs: roll)
- Asynchronous Inputs : Signals from different Clock Domain
- Synchronizer Objectives : Synchronization & DeBouncing



Verilog-HDL : FSM Design

Exercises (2)

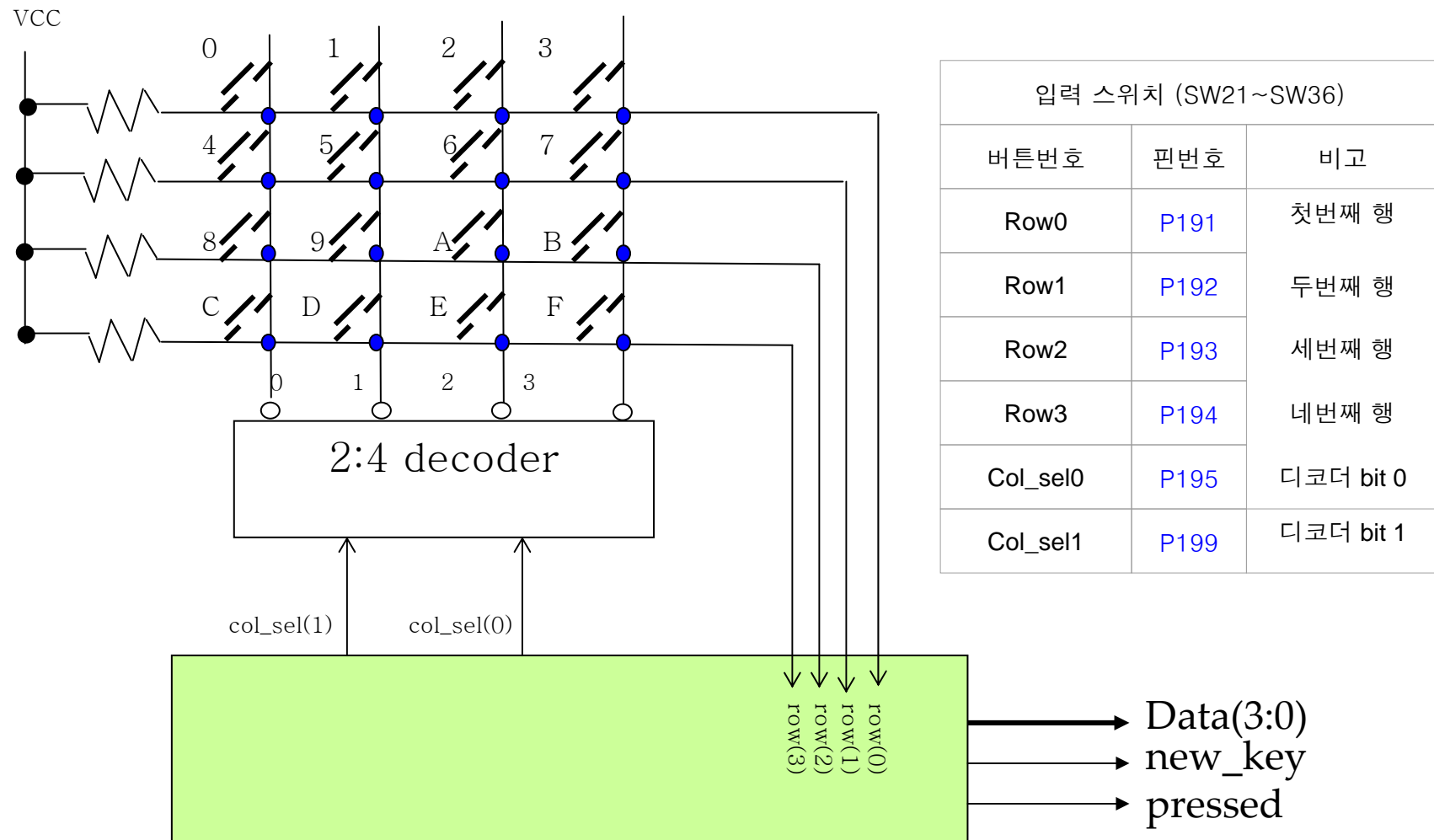
- Implement the previous design targeting at XC2S100 Xilinx FPGA. And Verify using the FPGA prototyping board. The Pin locations for the board are as follows :



Verilog-HDL : FSM Design

Exercises (3)

3. Design a 4 * 4 key pad input module in a dynamic fashion.



Verilog-HDL

목 차

- Basics
- Modules & Ports
- Verilog Simulation
- Gate Level Modeling
- Dataflow Modeling
- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- **Parameterized Design**
 - Parameter Value Change by defparam
 - Parameter value change at Module instance
 - Conditional Compilation
- More on Blocks
- Tasks & Functions
- Logic Synthesis

Verilog-HDL : Parameterized Design

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

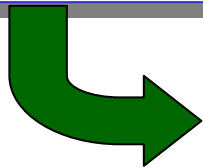
Verilog-HDL : Parameterized Design

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
```

Verilog-HDL : Parameterized Design

✚ 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
```

Verilog-HDL : Parameterized Design

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
```

Verilog-HDL : Parameterized Design

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
```

Verilog-HDL



Agenda

- Basics
- Modules & Ports
- Verilog Simulation
- Gate Level Modeling
- Dataflow Modeling
- Behavioral Modeling
- Application to Synchronous Logic
- FSM Design
- Parameterized Design
- **More on Blocks**
 - Parallel blocks
 - Named blocks
 - Nested blocks
- Tasks & Functions
- Logic Synthesis

Verilog-HDL : More on Blocks

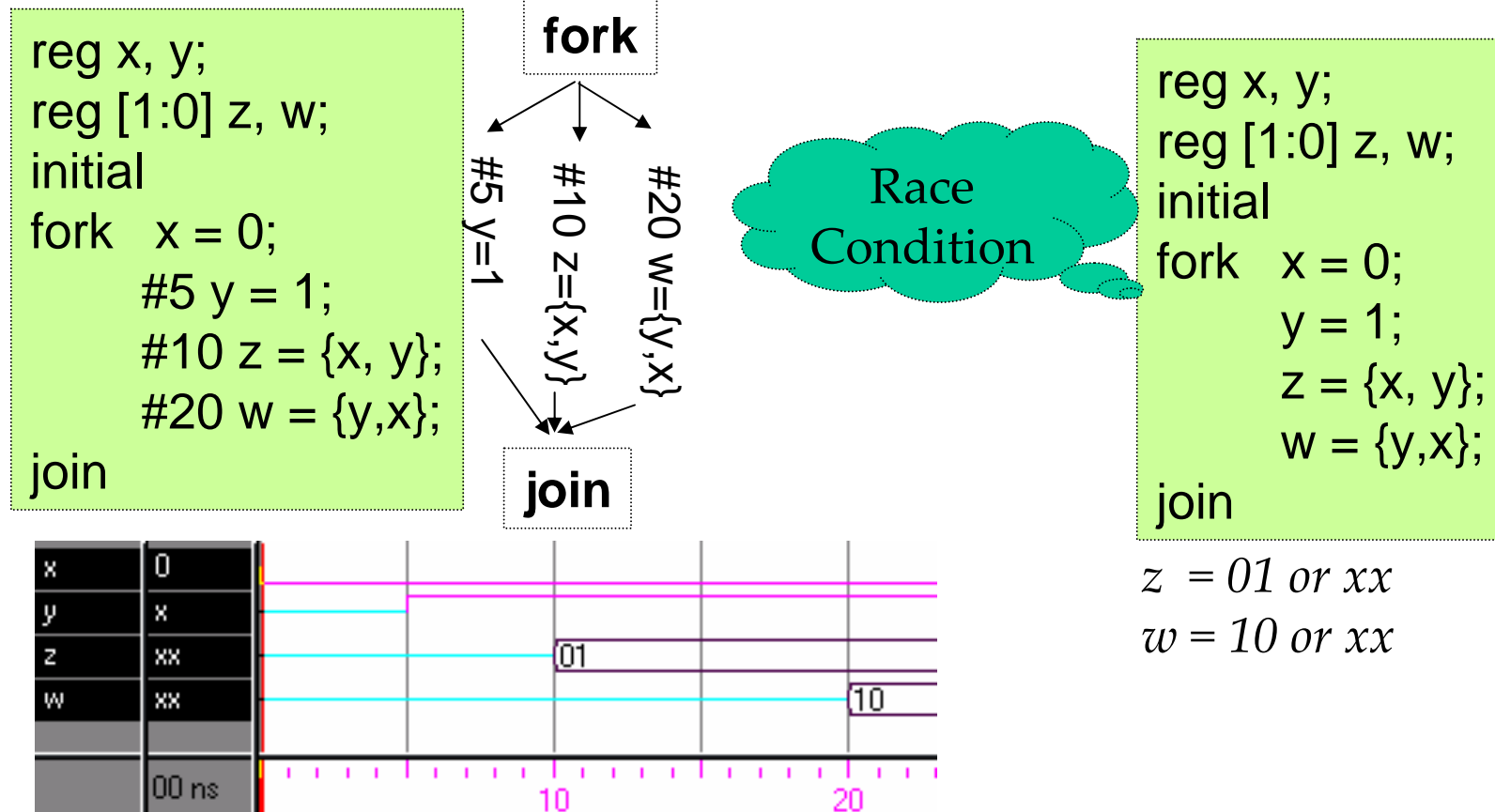
Block Types

- 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

Verilog-HDL : More on Blocks

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.

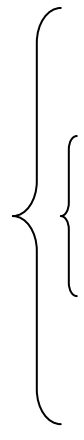


Verilog-HDL : More on Blocks

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
```



Verilog-HDL : More on Blocks

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 l ; // reg l is static & local to block2 (top.block2.i)  
    ...  
join
```

Verilog-HDL : More on Blocks

✚ 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
```

Verilog-HDL

목 차

- 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**
 - Tasks
 - Functions
 - Useful System Tasks
- Logic Synthesis

Verilog-HDL : Tasks & Functions

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

Verilog-HDL : Tasks & Functions

Function과 Task의 차이

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

Verilog-HDL : Tasks & Functions



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.

Verilog-HDL : Tasks & Functions

Task 선언 문장

```
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
```


Verilog-HDL : Tasks & Functions

Task Invocation Syntax

<name_of_task> ;

예 는

<name_of_task> (<parameter_list>) ;

- (Example)
 1. initiate_sequences ;
 2. Compute_values (output_val, input_val1, input_val2) ;

Verilog-HDL : Tasks & Functions

Task 예제 (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  
.....  
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
```

} invocation

} definition

Verilog-HDL : Tasks & Functions

Task 예제 (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

Verilog-HDL : Tasks & Functions

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

Verilog-HDL : Tasks & Functions

Function 선언 문장

```
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)

Verilog-HDL : Tasks & Functions

Function Invocation Syntax

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

Verilog-HDL : Tasks & Functions

✚ Function 예제 (1)

```
module parity ;  
  reg [31:0] addr;  
  reg parity ;  
  ...  
  always @(addr)  
    parity = calc_parity (addr) ; // function invocation  
  .....  
  function calc_parity ; // function declaration  
    input [31:0] address;  
      calc_parity = ^ address;  
  endfunction  
  .....  
endmodule
```

invocation {

definition {

Verilog-HDL : Tasks & Functions

Function 예제 (2)

invocation {

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


Verilog-HDL : Tasks & Functions

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.

Verilog-HDL : Tasks & Functions

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
```

Verilog-HDL : Tasks & Functions

Useful System Tasks

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

Verilog-HDL : Tasks & Functions

Opening File for Output

- \$fopen
 - Usage : **\$fopen**("<filename>");
 - Usage : <file_handler> = **\$fopen**("<filename>");
 - <file_handler> : 32-bit integer value called *multichannel descriptor*
- Multichannel descriptor
 - each successive call to **\$fopen** returns *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
```

Verilog-HDL : Tasks & Functions

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
```

Verilog-HDL : Tasks & Functions

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
```

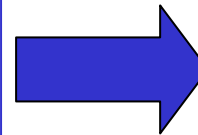
Verilog-HDL : Tasks & Functions

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
```

Verilog-HDL : Tasks & Functions

\$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

Verilog-HDL : Tasks & Functions

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
```

Verilog-HDL : Tasks & Functions

✚ Initializing Memory from File

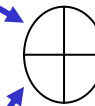
- Initializing the contents of Memory in Verilog
- 2 System Tasks : **\$readmemb** (binary) , **\$readmemh** (hexadecimal)
- \$readmemb / \$readmemh Usage :
 - **\$readmemb** (“<file_name>”, <memory_name>);
 - **\$readmemb** (“<file_name>”, <memory_name>, <start_addr>);
 - **\$readmemb** (“<file_name>”, <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
```

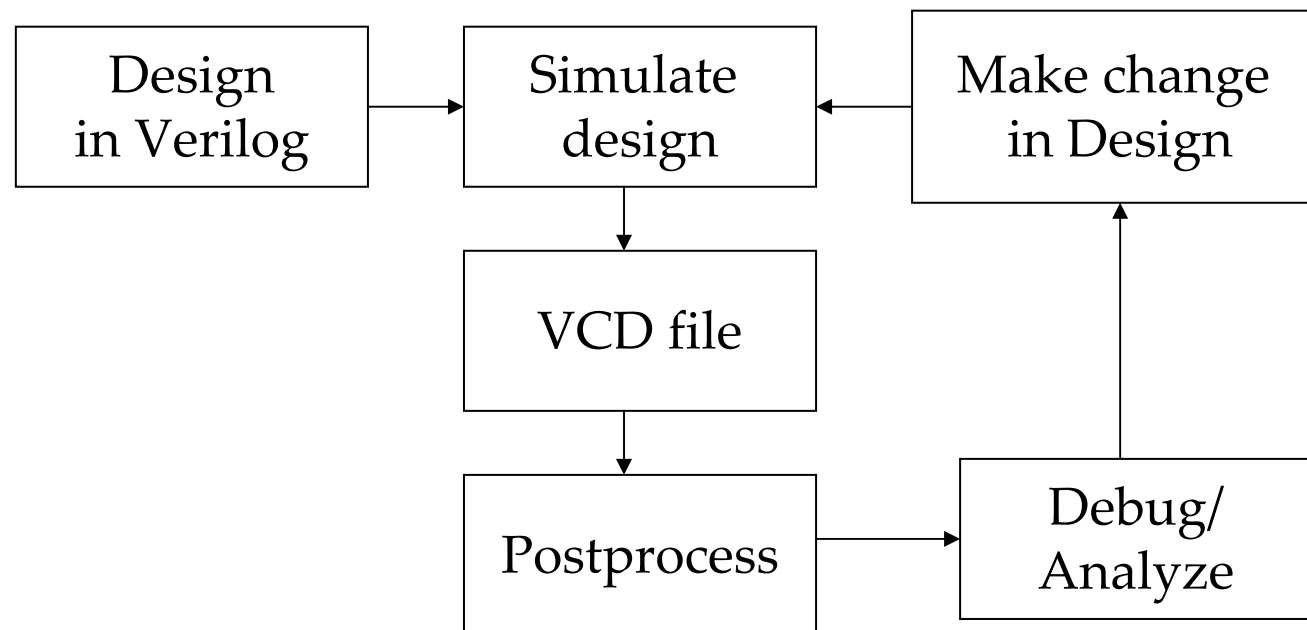
```
M[0] =xxxxxxx
M[1] = xxxxxxx
M[2] = 11111111
M[3] = 01010101
M[4] = 00000000
M[5] = xxxxxxx
M[6] = 1111zzzz
M[7] = 00001111
M[8] = xxxxxxx
M[9] = xxxxxxx
M[10] = xxxxxxx
M[11] = xxxxxxx
M[12] = xxxxxxx
M[13] = xxxxxxx
M[14] = xxxxxxx
```



Verilog-HDL : Tasks & Functions

+ 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



Verilog-HDL : Tasks & Functions

VCD와 관련된 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

Verilog-HDL : Tasks & Functions

VCD 예제

```
// 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;
```

Verilog-HDL

목 차

- 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**
 - Introduction to Logic Synthesis
 - Coding Style Guidance for Synthesis

Verilog-HDL : Logic Synthesis

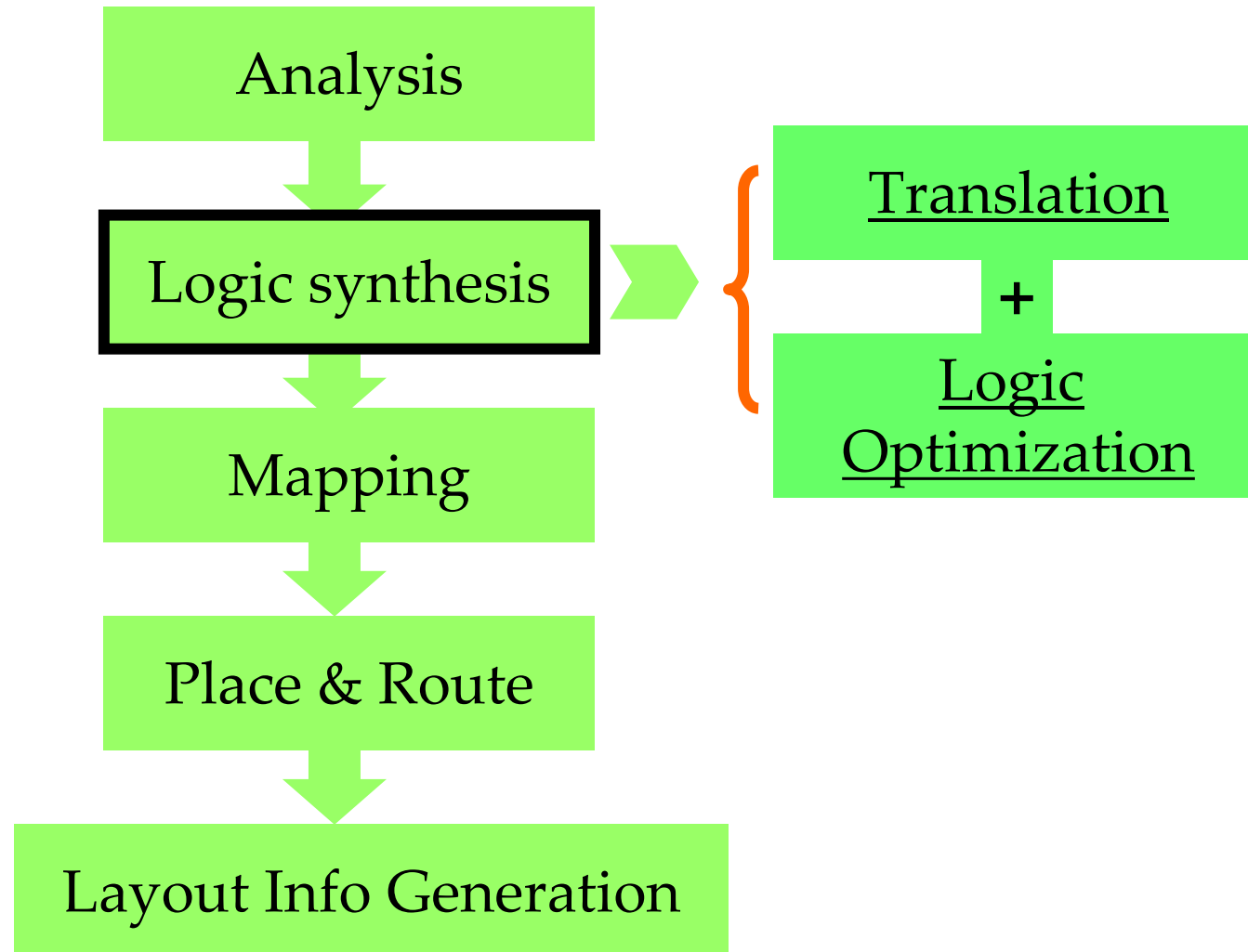


What and Why Logic Synthesis?

- 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

Verilog-HDL : Logic Synthesis

+ Typical Logic Synthesis Flow



Verilog-HDL : Logic Synthesis

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

Verilog-HDL : Logic Synthesis

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

Verilog-HDL : Logic 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
```

Verilog-HDL : Logic Synthesis

✚ 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
```

Verilog-HDL : Logic Synthesis

✚ 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
```