

Verilog Syntax

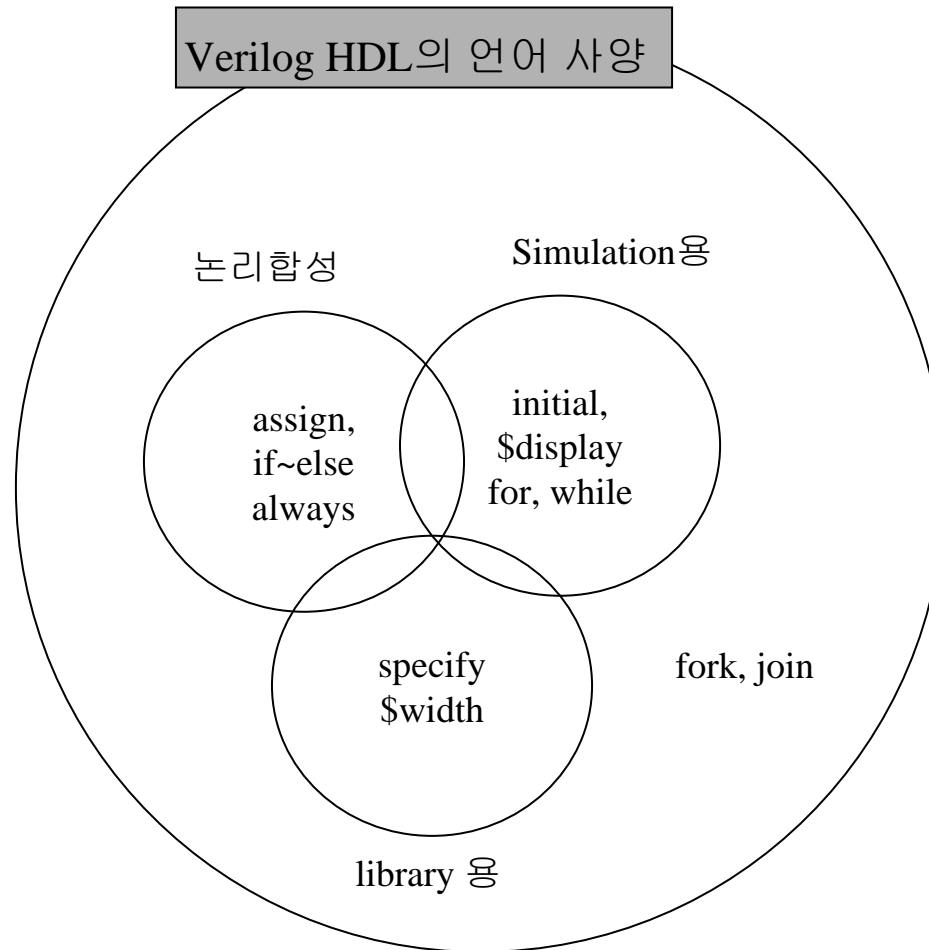
Media Processor Lab.

Sejong Univ.

- 본 ppt는 동의대학교 최병윤 교수님 자료를 이용하여 만들어 졌습니다.

Verilog HDL의 언어 사양

- Verilog HDL
 - IEEE Std 1364-1995



Verilog HDL의 기본 문법

■ module 구성

- 사용 가능 식별자 : 영문, 숫자, underscore(_)
- C언어와 같이 대 소 문자 구별

<module 구조>

```
module module명 (port-list);  
port 선언  
net 선언  
reg 선언  
parameter 선언
```

```
assign문  
function문  
task문  
initial 문  
always문  
function호출문  
task 호출문  
하위 모듈문  
endmodule
```

(예) port 선언 예
input ck, res; // 입력
input [7:0] bus1, bus2; // 입력 버스 신호
output busy; // 출력 신호
inout [15:0] dbus;

(예) net 선언 예
wire enbl;
wire [7:0] bus;

(예) register 선언 예
reg ff1, ff2;
reg [3:0] cnt4;
reg [7:0] mem[0:255];

(예) parameter 선언 예
parameter STEP=1000;
parameter HALT=2'b00, INT=2'b01,
ACTION=2'b10; // state
parameter MEMSIZE=1024;
reg [7:0] mem[0:MEMSIZE-1]; // parameter 사용 예

Verilog HDL의 기본 문법

■ module item과 statement

- module은 module_item 으로 구성
- statement문은 initial, always, function (module_item)문 내에 사용

< module_item >

Gate 호출
하위 모듈 호출
initial문
always문
assign문
function문
task 문
function 호출
task 호출

<statement>

if 문
case문
for문
while문
repeat문
begin-end block
fork-join block
등

Verilog HDL의 기본 문법

■ 논리값과 수치 표현

● Verilog-HDL에서 사용하는 논리 값

- 0, 1, x(unknown), z(high-impedance)
- 신호의 strength(supply, strong, pull, large, weak, medium, small, high-z)
(default strength = strong)

(예) strong0, strong1

- 신호의 strength는 simulation에만 사용하고, 합성 시에는 제외

● 상수의 표현

형식 : <bit width>'<base><value>

여기서 bit width : bit 폭을 나타내는 10진수(default = 32-bit)

base : b, B : binary, o, O : octal,

d, D : decimal, h, H : hexadecimal (default = decimal)

value : 기수에 대응하는 상수 값

(예: base가 octal인 경우 : 0 ~ 7, x, z)

(주) 구분을 위해 underscore(_)사용

Verilog HDL의 기본 문법

수치상수	비트 폭	기수(base,radix)	2진수 표현
10	32	10	00...01010
1'b1	1	2	1
8'haa	8	16	10101010
4'bz	4	2	zzzz
8'b0000_11xx	8	2	000011xx
'hff	32	16	00...01111111
4'd5	4	10	0101

Verilog HDL의 기본 문법

■ 데이터 타입(data type)

● 신호(변수)의 선언

- 프로그램안에 사용되는 신호(변수)는 모두 형을 선언해야 함

1). Register 형(reg로 선언) : latch나 F/F(값을 보존)

- 개념상 일반 프로그래밍 언어의 변수 개념

2). Net 형 (wire로 선언한 것) : 실제적인 연결관계를 나타내는 배선

● 신호(변수) 사용상 제약 사항

- 모든 형의 참조 시에는 제약이 없음

- 모든 형은 식의 우변에 사용하거나 function의 인수로 사용 가능

- 단, 대입(식의 왼편에 놓이는 경우)시에 제약 사항

[제약 1] reg형의 신호로의 대입은 always, initial, task, function문에서만 사용

[제약 2] wire형으로의 대입은 assign문에서만 사용

● type(형) 선언의 생략

- port 신호의 경우 net형인 경우 일반적으로 생략 가능

- 1-bit wire는 생략 가능(default = 1-bit wire)

- multi-bit wire는 생략 불가능

(주) assign 문에 의해 할당되는 wire의 경우, 형 선언 생략 불가능

Verilog HDL의 기본 문법

- 데이터 형 예

```
module DEF(CK, D, Q);  
input CK, D;  
output Q;  
wire CK, D;    // 생략 가능  
reg  Q;  
  
always @(posedge CK)  
    Q = D;  
endmodule
```

```
module RSFF(SB, RB, Q);  
input SB, RB;  
output Q;  
wire temp;    // 생략 가능  
  
nand na1 (Q, SB, temp);  
nand na2 (temp, Q, RB);  
  
endmodule
```

Verilog HDL의 기본 문법

- 다 bit 신호

- 신호 선언 시에 [MSB:LSB] 형식으로 비트 폭과 범위 지정
- 다 비트 신호는 「무부호 정수」로 취급됨

(예)

```
inout [3:0] a, b; // 4 비트 입력 a, b
wire [7:0] dbus; // 8 비트의 net 신호
reg [7:0] addr_hi; // 8 비트의 register 신호
```

- 다 bit 신호의 bit 선택

(예) 다 비트 신호를 1 비트 단위로 access할 경우

```
assign MSB = dbus[7];
assign LSB = dbus[0];
assign dbus[4] = half_carry;
```

- 다 비트 신호의 부분 선택

(예)

```
wire [3:0] Hi_digit;
assign Hi_digit=addr_hi[15:12];
```

(예)

```
wire [3:0] Lo_digit;
assign dbus[3:0] =Lo_digit;
```

Verilog HDL의 기본 문법

■ register 배열(memory)

- register 형으로 선언된 배열

(예)

```
reg [7:0] mem[0:255]; // 256 words * 8-bit memory
```

(주) register 배열은 bit 선택과 부분 선택 불가

- word 단위의 access만 허용
- bit 선택과 부분 선택이 필요할 경우, 2단계 처리 필요
- programming 언어와 같은 다차원 배열은 불가

- 워드 단위의 access 예

(예) mem[0]

- 비트 선택이나 부분 선택 예

- 임시 net를 사용한 2단계 처리 필요

(예)

```
wire [7:0] temp;
```

```
assign temp = mem[100]; // 단계 1: word 단위 access
```

```
.....
```

```
temp[7] // mem[100]의 최상위 비트 : bit 단위 access
```

```
temp[3:0] // mem[100]의 하위 4 비트 : 부분 선택
```

Verilog HDL의 기본 문법

■ 연산자

연산자	기능	연산자	기능		
산술연산		논리연산			
+ - * / %	가산, plus 부호 감산, minus 부호 승산 제산 잉여	! && 	논리 부정 논리 AND 논리 OR		
		논리연산			
		== != === !==	equal not equal equal(x,z도 비교) not equal(x, z도 비교)		
				관계연산	
				< <= > >=	Less than less than or equal to larger than larger than or equal to
reduction 연산					
& ~& ~ ^ ~^	AND NAND OR NOR XOR XNOR	Shift 연산			
		<< >>	left shift right shift		
				기타 연산	
		?: { }	조건 연산 concatenation		

Verilog HDL의 기본 문법

- Equality operator


- 1). Logical equality(==)
- 2). Logical inequality(!=)
- 3). Case equality(===)
- 4). Case inequality(!==)

식 표현	동작 설명	반환 결과
a == b	Operand에 x, 또는 z가 있는 경우 결과 x	0, 1, x
a != b	Operand에 x, 또는 z가 있는 경우 결과 x	0, 1, x
a === b	Operand에 있는 x, 또는 z, 0, 1의 모든 일치 비교	0, 1
a !== b	Operand에 있는 x, 또는 z, 0, 1의 모든 불일치 비교	0, 1

(주) !=, ===은 논리 합성이 지원되지 않음

Verilog HDL의 기본 문법

■ 연산자의 우선 순위

<code>*, /, %</code>	 highest
<code>+, -</code>	
<code><<, >></code>	
<code><, <=, >, >=</code>	
<code>==, !=, ===, !==</code>	
<code>&</code>	
<code>^, ^~</code>	
<code> </code>	
<code>&&</code>	

Verilog HDL의 기본 문법

■ 연접 연산(concatenation operator)

- 연접 연산자를 대입문의 우변에 사용

```
wire [15:0] addr_bus;  
assign addr_bus = {addr_hi, addr_lo}; // addr_hi, addr_lo는 8 비트 신호
```

- 연접 연산을 대입문의 우변에 사용

```
wire [3:0] a, b, sum;  
wire carry;  
assign {carry, sum} = a + b; // 4 비트 데이터의 덧셈은 5 비트 결과
```

(주) verilog HDL에서는 비트 폭이 일치하지 않는 변수의 연산이나 대입이 허용됨

- ① 좌변이 5 비트 이므로, 좌변의 a+b는 MSB에 0을 붙인 5 비트로 연산 됨
- ② 대입의 좌변이 우변 보다 비트 폭이 작을 경우, MSB는 누락되어 저장

Verilog HDL의 기본 문법

- 연접 연산을 활용한 반복 연산

- 연접 연산자를 쌍으로 사용

(예)

```
{4{2'b01}} // 8'b01010101
```

(예)

```
wire [15:0] word;  
wire [31:0] double;  
assign double = { {16{word[15]}}, word};
```

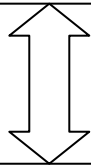

Verilog HDL의 기본 문법

■ Reduction 연산

- 항 머리에 연산자를 붙이는 단항 연산자
- 사용 연산자 : &, ~, ^, |
- 비트 폭을 가진 신호내의 모든 비트에 작용하며, 연산 결과는 1 비트 값

(예)

```
reg [7:0] cnt;  
assign all_one = &cnt;  
assign parity = ^cnt;
```



```
assign all_one = cnt[7] & cnt[6] & cnt[5] & cnt[4] & cnt[3] & cnt[2] & cnt[1] & cnt[0] ;  
assign parity = cnt[7] ^ cnt[6] ^ cnt[5] ^ cnt[4] ^ cnt[3] ^ cnt[2] ^ cnt[1] ^ cnt[0] ;
```

Verilog HDL의 기본 문법

■ Verilog HDL이 갖는 primitive function

- port내 위치가 output이 가장 앞에 위치하고, 그 뒤에 input이 나옴
- single output 형태

Gate	
and	buf
nand	not
nor	bufif0
or	bufif1
xor	notif0
xnor	notif1
	pullup
	pulldown

MOS switches and Bidirectional Transistors	
nmos	tran
pmos	tranif0
cmos	tranif1
rnmos	rtran
rpmos	rtranif0
rcmos	rtranif1

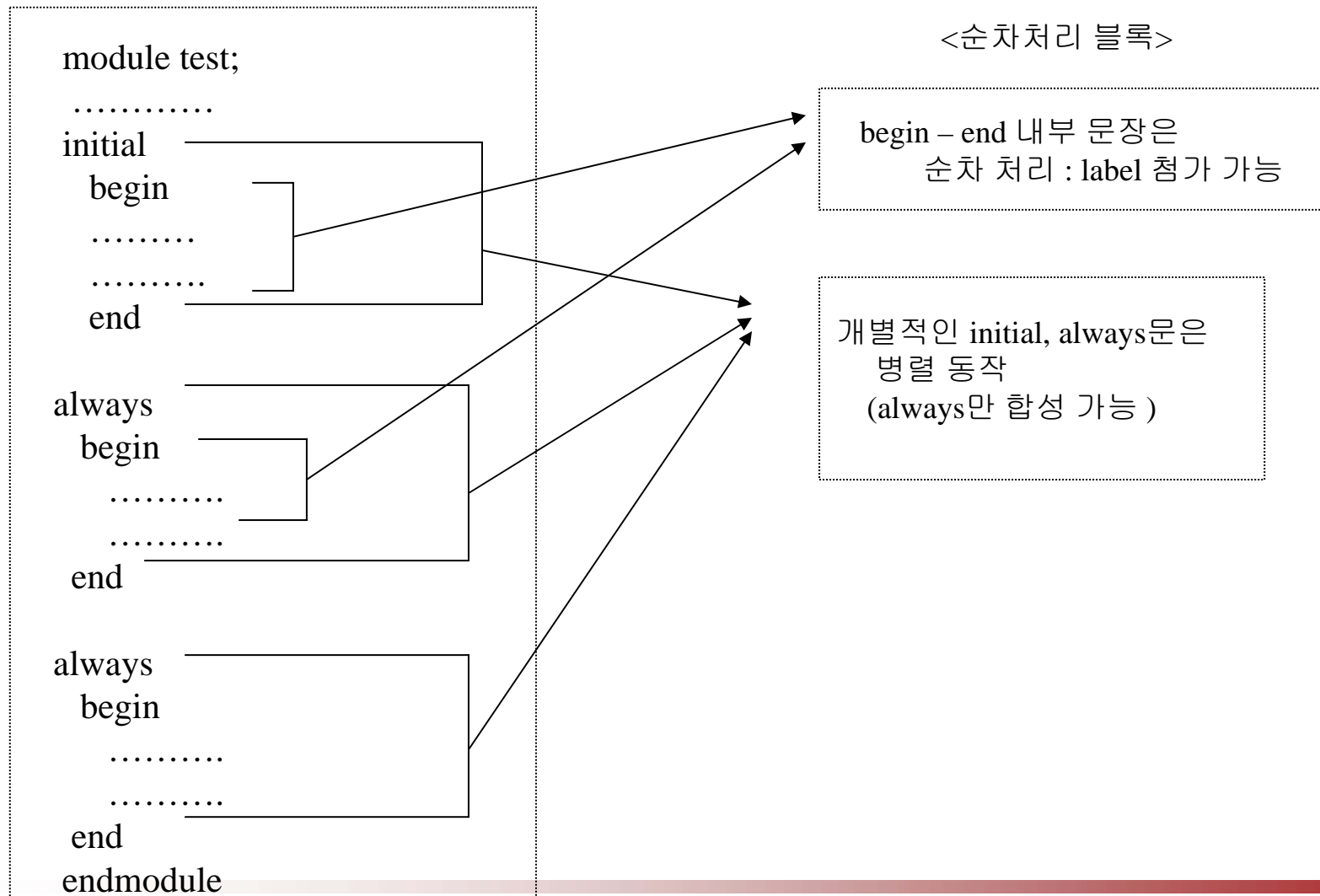
Net	
wire	
wand	wor
tri	triand
trior	supply0
tri1	supply1
tri0	

형식 1 :
primitive_gate명 (output, input1, input2,);

형식 2: // bufif0, bufif1, notif0, notif1
primitive_gate명 (output, input, control);

Verilog HDL의 기본 문법

■ Procedural Statement(I)



Verilog HDL의 기본 문법

■ Assignment statement

○ blocking assignment 형태

```
lhs-expression = expression;  
lhs-expression = #delay expression;  
lhs-expression = @event expression;
```

○ non-blocking assignment 형태

```
lhs-expression <= expression;  
lhs-expression <= #delay expression;  
lhs-expression <= @event expression;
```

Verilog HDL의 기본 문법

■ Port 선언과 연결 규칙

● Port 선언

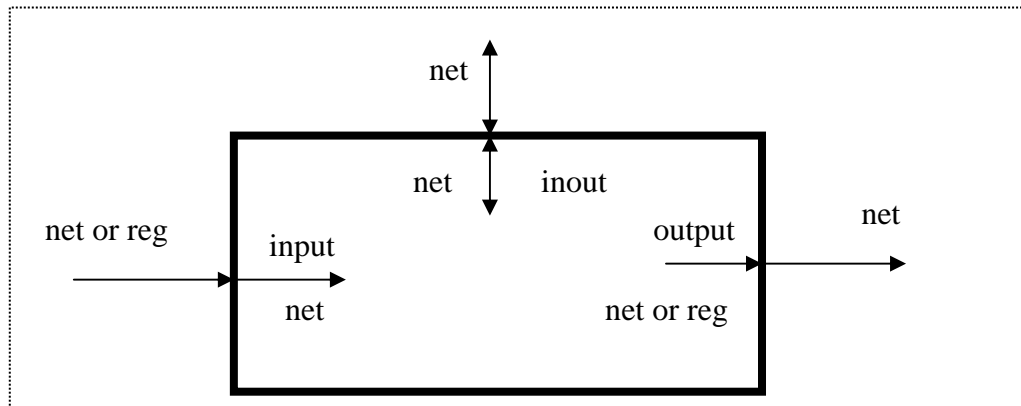
Verilog keyword	Port의 형
input	입력 port
output	출력 port
inout	양방향 port

- 모든 port는 default로 wire로 인식됨
- port가 wire인 경우는 input, output, inout 선언만으로 충분
- 단, output port의 경우 그 값이 hold되어야 하는 경우
(initial, always문 내 우변 변수)
→ 추가적으로 reg 선언이 필요
- input과 inout port는 reg 선언 불가능

Verilog HDL의 기본 문법

● Port 연결 규칙

- port는 2개의 unit측면으로 시각화 가능
 - 1). module 내부 측면
 - 2). module 외부 측면



1). input

- 내부적 측면 : 항상 net로 선언
- 외부적 측면 : reg 또는 net 형의 변수 연결

2). output

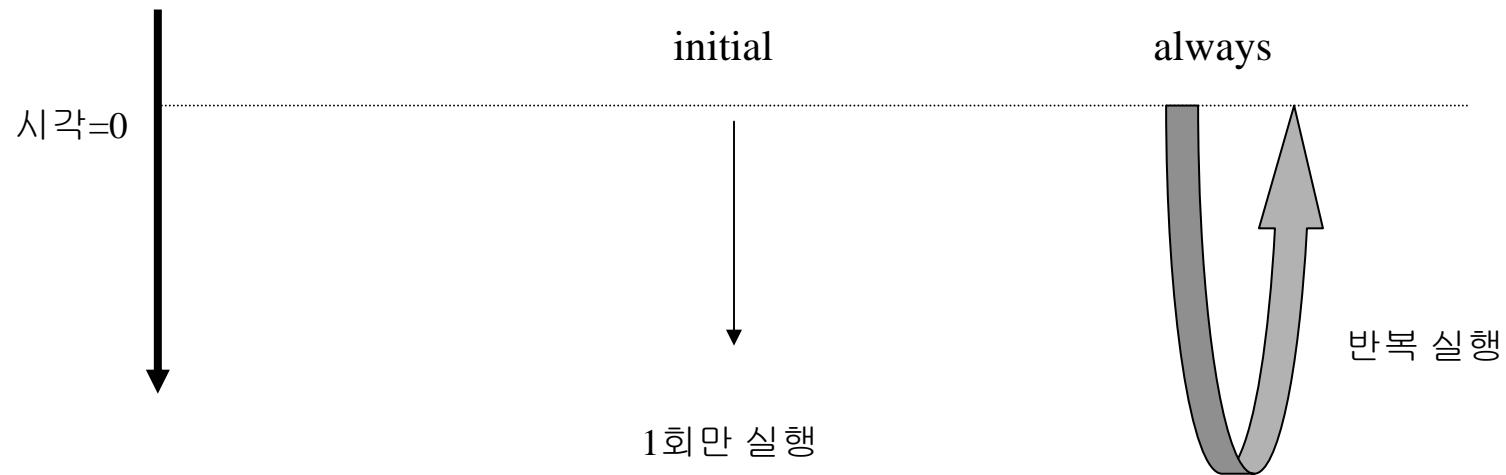
- 내부적 측면 : reg 또는 net 형의 변수 연결
- 외부적 측면 : net에 연결 (reg 연결 불가)

3). inout

- 내부적 측면 : 항상 net로 선언
- 외부적 측면 : 항상 net에 연결

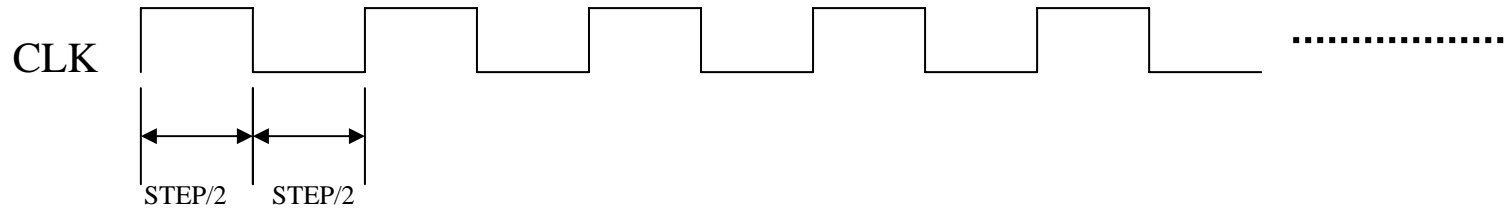
Verilog HDL의 기본 문법

■ initial 문과 always 문 비교



Verilog HDL의 기본 문법

● Clock 정의 기법



① clock정의 : initial과 always 문 사용

```
// single-phase clock
reg clk;
// 1 주기 = 100 unit
parameter STEP = 100;
always
    #(STEP/2) clk = ~clk;
initial
    clk = 1;
```

② clock정의 : always 문 사용

```
// single-phase clock
reg clk;
// 1 주기 = 100 unit
parameter STEP = 100;
always
    begin
        clk = 1;
        #(STEP/2) clk = ~clk;
        #(STEP/2);
    end
```


Verilog HDL의 기본 문법

■ timing 제어

- #<정수식> : 정수 값에 따라 지정된 시간 만큼 처리가 지연됨
- @<event식> : <EVENT식>에 의해 지정된 EVENT가 발생할 때까지 처리를 지연 시킴

<EVENT식>에 포함되는 접두어

* posedge : 상승 edge

* negedge : 하강 edge

(접두어가 생략될 경우, 상 • 하강 edge에서 event 발생)

(예) always 문에서 사용 예

```
// always문에서 사용 예
always
    #(STEP/2) CLK = ~CLK;

always @(posedge CLK)
begin
    .....
end
```

Verilog HDL의 기본 문법

■ wait문

(예) wait문 사용 예

```
// wait문 사용 예
initial
begin
    #STEP .....;
    #STEP .....;
    .....
    // BUSY가 “0”이 될 때 까지 기다림
    wait(~BUSY).....;
end
```

Verilog HDL의 기본 문법

□ Register 변수형에 대한 대입문의 2가지 형태

1). blocking 대입 :

형식 : reg 변수 = 식;

2). non-blocking 대입

- blocking 문이 사용되는 위치에 모두 사용 가능

형식 : reg 변수 <= 식;

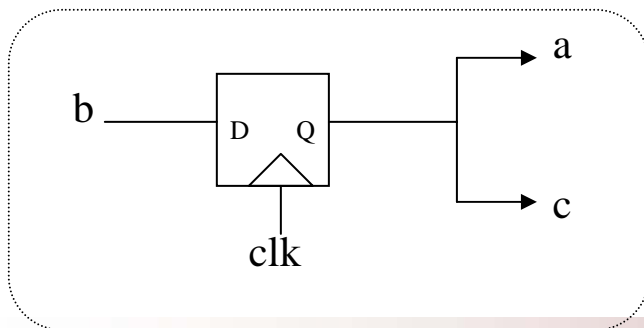
Verilog HDL의 기본 문법

■ Blocking 문과 nonblocking 문 동작 비교

- Blocking 대입(초기 a=5, b=4, c=7 가정)

```
.....  
always @(posedge clk)  
begin  
  
a = b; // b=a=4  
c = a; // c=b=a=4  
  
end  
.....
```

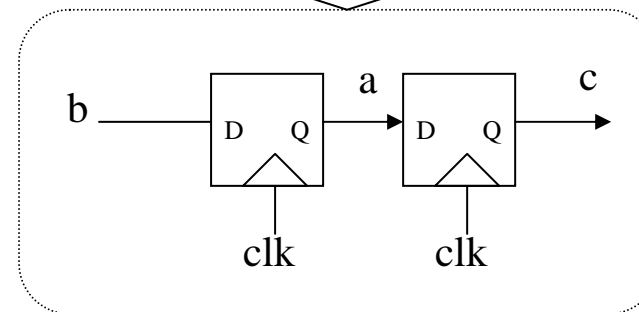
⇕ 합성시 1개의 F/F 생성



- non-blocking 대입(초기 a=5, b=4, c=7가정)

```
.....  
always @(posedge clk)  
begin  
  
a <= b; // b=a=4  
c <= a; // c=a_old=5  
  
end  
.....
```

⇕



Verilog HDL의 기본 문법

- case 문
 - multi-way branch 구현

```
module case_statement;  
  integer I;  
  initial I=0;  
  
  always  
  begin  
    $display("I = %0d", I);  
    case(I)  
      0 : I = I+2;  
      1 : I = I + 7;  
      2 : I = I -1;  
      default : $stop;  
    endcase  
  end  
endmodule
```

참고: casex, casez

Verilog HDL의 기본 문법

- if 문 형식
 - always, initial, function 문 내에 존재

< 형식 1 >

```
.....  
always @(. ....)  
begin  
.....  
if (조건)  
    statement;  
.....  
end
```

< 형식 2 >

```
.....  
always @(. ....)  
begin  
    if (조건 1)  
        statement 1;  
    else if (조건 2)  
        statement 2;  
    .....  
    else if (조건 n)  
        statement n;  
    else  
        statement (n+1);  
    .....  
end
```

Verilog HDL의 기본 문법

■ repeat문

- 고정 회수의 loop를 만들기 위한 구문
 - <식>에서 주어진 회수만큼
 <statement> 반복
- 형식 : repeat (<식>) <statement>

(예)

```
// repeat 문 예
.....
reg [7:0] data;
parameter DUST =120;

initial
begin
    repeat(DUST)
        readreg(DATAREG data);
    .....
end
.....
```

■ forever 문

- 무한 loop를 만드는 문

(예)

```
// forever 문 예
.....

parameter STEP =1000;
initial
begin
    .....
    forever
    begin
        CK=0;
        #(STEP/2) CK=1;
        #(STEP/2) ;
    end
    .....
end
.....
```

Verilog HDL의 기본 문법

□ tasks와 function

■ tasks와 function 비교

function	Tasks
Function은 0 simulation time 가짐 (no delay)	Non-zero simulation time 수행 가능
어떠한 delay문, event, 또는 timing control 문 포함 불가	delay문, event, 또는 timing control 문 포함 가능
적어도 하나의 input 존재 필요 (다수 입력 가능)	0개 또는 다수개의 input, output, inout문 포함 가능
항상 하나의 값을 return - 인수로 output과 inout을 가질 수 없음 - return시 함수 명을 통해 반환	값을 return하지 않으며, output과 inout argument를 통해 다수개의 결과를 pass 가능
zero delay를 갖는 Combinational logic 구현	-Delay, timing, event을 갖고, Multiple output을 반환하는 Verilog code에 적용 가능
input port	input, inout, output port

Verilog HDL의 기본 문법

- function과 task의 일반적인 구조

```
module test (.....);  
.....  
  
function [n-1:0] func_name; // 함수 정의  
input .....; // 인수 순서 중요  
.....  
begin  
.....  
func_name = .....;  
.....  
end  
endfunction  
.....  
  
..... = func_name(...); // 함수 호출  
.....  
endmodule
```

```
module test (.....);  
.....  
  
task task_name; // 함수 정의  
input .....; // 인수 순서 중요  
output .....;  
inout .....;  
.....  
begin  
.....  
.....  
end  
endtask  
.....  
  
task_name(...); // task 호출  
.....  
endmodule
```

Verilog HDL의 기본 문법

■ `include directive 기능

- VHDL의 package와 유사하게, 기존의 verilog HDL 코드(library)를 현재 verilog HDL 코드에 포함시킴 (C언어의 #include 기능)
- 모듈의 내외 어느 쪽도 배치 가능

(예)

```
`include "header.v"  
.....  
module test;  
    .....  
endmodule
```

Verilog HDL의 기본 문법

■ `define 문

- Text 치환 용도로 사용하는 compiler 지시어

(예)

```
`define A lzc_top.izc_addr
```

→ No semicolon

```
.....  
initial $monitor("dicptr=%h", `A.dicptr);
```

Verilog HDL의 기본 문법

■ time scale command

형식 : `timescale <reference_time_unit> / <time_precision>

여기서 reference time : time과 delay 측정 단위

time_precision : round-off precision

(예)

```
`timescale 100ns / 1ns

module dumm1;
reg toggle;

initial
    toggle = 1'b0;
always #5
begin
    toggle = ~toggle;
end
endmodule
```

(주) 5 time unit=
 $5 * 100 = 500\text{ns}$

Verilog HDL의 기본 문법

■ 논리 합성 모듈(회로 프로그램)에 가장 널리 사용되는 5가지 유형

- assign문에 의한 조합 회로: 단순한 조합 회로
- always문에 의한 조합 회로 : 복잡한 조합 회로
- function문에 의한 조합 회로 : 복잡한 조합 회로
- always문에 의한 순서 회로 : Latch, F/F 소자 포함
- 하위 모듈 호출

Verilog HDL의 기본 문법

- assign 문에 의한 조합 회로

- 간단한(프로그램 특성) 조합회로는 assign문으로 프로그래밍함
- 대입되는 좌변은 반드시 wire로 선언한 net형만 가능 (reg 형 불가)

(예)

```
module and_or (in1, in2, in3, out);  
input in1, in2, in3 ;  
output out;  
  
assign out = in1 & in2 | in3;  
  
endmodule
```

Verilog HDL의 기본 문법

- always 문에 의한 한 조합 회로
 - 복잡한 조합회로는 always문으로 프로그래밍함

(예)

```
module mux_4_3 (Y, D, S);  
  input [0:3] D;  
  input [0:1] S;  
  output Y;  
  reg Y;  
  always @(S or D)  
  begin  
    case(S)  
      0 : Y = D[0];  
      1 : Y = D[1];  
      2 : Y = D[2];  
      3 : Y = D[3];  
    endcase  
  end  
endmodule
```

→ 입력 변수 모두 기술 필요

Verilog HDL의 기본 문법

- function을 사용한 조합회로

- 복잡한(회로 규모가 아닌 프로그램 기술 특성) 조합회로는 function사용

(예) 2-to-4 decoder

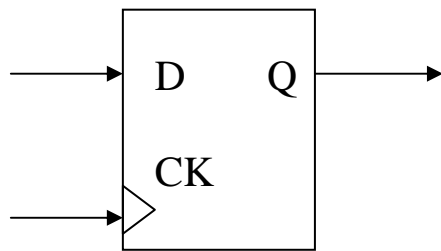
```
module dec2to4 (in1, out);
input [1:0] in1;
output [3:0] out;
// function 정의 : module내
function [3:0] dec;
input [1:0] in;
begin
    case (in)
        0 : dec=4'b0001;
        1 : dec=4'b0010;
        2 : dec=4'b0100;
        3 : dec=4'b1000;
    endcase
end
endfunction
// 함수 호출
assign out = dec(in1);
endmodule
```


Verilog HDL의 기본 문법

- always문을 쓴 순서회로

- latch나 F/F은 always문을 사용하여 프로그램함
- 순서 회로를 프로그래밍할 때, 관련된 조합회로도 동시에 프로그램 하는 것이 일반적임
- 동일 클럭으로 동작하는 순서회로는 하나의 always문 내에 모아서 프로그래밍 가능

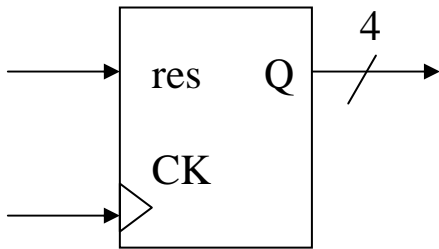
(예) always문을 쓴 D F/F



```
module DFF (ck, d, q);
input ck,d;
output q;
reg q;
always @(posedge ck)
    q <= d;
endmodule
```

Verilog HDL의 기본 문법

(예) always문을 사용한 binary counter



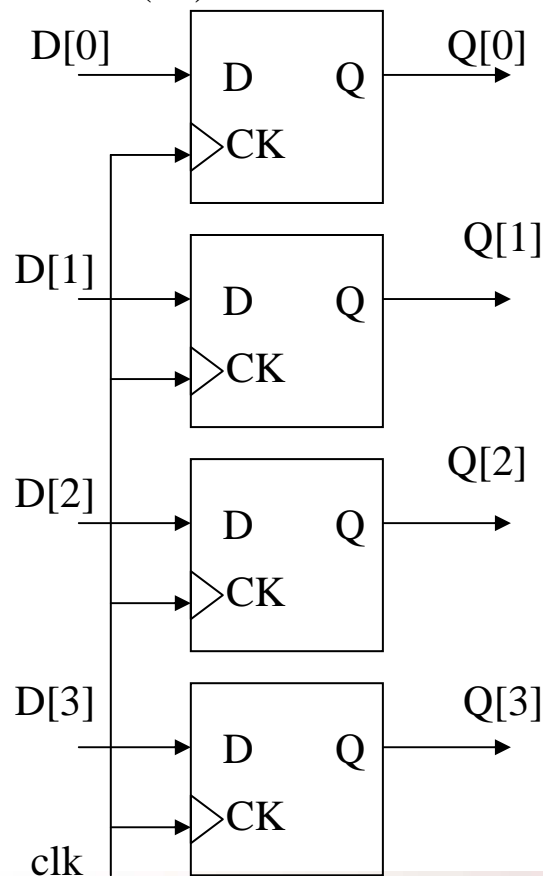
```
module counter (ck, res, q); // 동기 reset
input ck,res;
output [3:0]q;
reg [3:0] q;
always @(posedge ck) begin
    if (res)
        q <= 4'h0;
    else
        q <= q + 4'h1;
end
endmodule
```

Verilog HDL의 기본 문법

- 회로 계층의 모듈화 이유

- ① 기능적으로 정리되는 작은 블록으로 나누면 설계 검증의 효율성 증대
- ② 블록 크기를 논리 합성 툴의 실용 범위 내에서 제한할 필요성
- ③ 필요 이상 크기의 블록에 대한 논리 합성은 메모리와 시간 낭비 초래

(예) 4 비트 F/F



```
module dff4 (ck, d, q);  
  input ck;  
  input [3:0] d;  
  output [3:0] q;
```

```
// 순서에 의한 port 접속  
DFF DFF0 (ck, d[0], q[0]);  
DFF DFF1 (ck, d[1], q[1]);
```

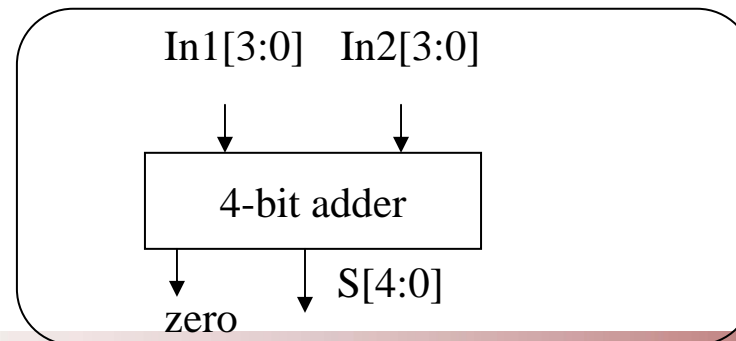
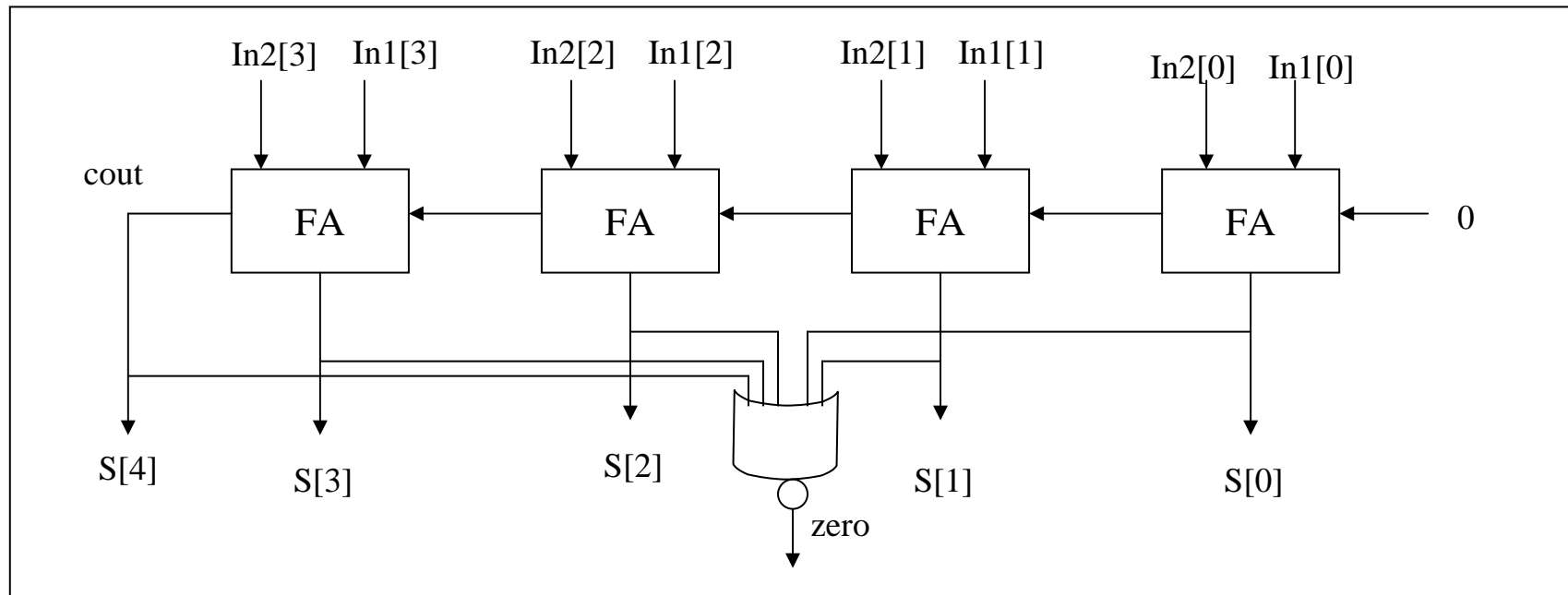
```
// 이름에 의한 접속  
DFF DFF2 (.ck(ck), .d(d[2]), .q(q[2]));  
DFF DFF3 (.ck(ck), .d(d[3]), .q(q[3]));  
  
endmodule
```

```
module DFF (ck, d, q);  
  input ck,d;  
  output q;  
  reg q;  
  always @(posedge ck)  
    q <= d;  
endmodule
```

(주) 이름에 의한 접속과 순서에 의한 접속 혼용 불가

4 -Bit Adder 에 대한 Verilog

- zero 감지 기능을 갖는 4-bit adder



4 -Bit Adder 에 대한 Verilog

● 4-bit adder에 대한 기술(I) : structural-level 기술

```
module adder4 (in1, in2, s, zero);  
    input [3:0] in1;  
    input [3:0] in2;  
    output [4:0] s;  
    output zero;  
    fulladd u1 (in1[0], in2[0], 0, s[0], c0);  
    fulladd u2 (in1[1], in2[1], c0, s[1], c1);  
    fulladd u3 (in1[2], in2[2], c1, s[2], c2);  
    fulladd u4 (in1[3], in2[3], c2, s[3], s[4]);  
    nor u5 (zero, s[0], s[1], s[2], s[3], s[4]);  
endmodule
```

< fulladd module의 기술 >

```
module fulladd (in1, in2, carryin, sum, carryout);  
    input in1, in2, carryin;  
    output sum, carryout;  
    assign {carryout, sum} = in1 + in2 + carryin;  
endmodule
```

4 -Bit Adder 에 대한 Verilog

- 4 비트 adder 기술(2) : (dataflow + behavioral) level 사용

```
module adder4 (in1, in2, sum, zero);  
    input [3:0] in1;  
    input [3:0] in2;  
    output [4:0] sum;  
    reg [4:0] sum;  
    output zero;  
    assign zero = (sum == 0) ? 1 : 0;  
    initial sum = 0;  
    always @ (in1 or in2)  
        sum = in1 + in2;  
endmodule
```

4 -Bit Adder 에 대한 Verilog

- 4-bit adder에 대한 기술(3): behavioral level기술
 - initial 문 사용으로 합성 불가능

```
module adder4 (in1, in2, sum, zero);
    input [3:0] in1;
    input [3:0] in2;
    output [4:0] sum;
    output zero;
    reg [4:0] sum;
    reg zero;
    initial begin
        sum = 0;
        zero = 1;
    end
    always @ (in1 or in2) begin
        sum = in1 + in2;
        if ( sum == 0)
            zero = 1;
        else
            zero = 0;
    end
endmodule
```

4 -Bit Adder 에 대한 Verilog

- 4-bit adder에 대한 기술 : mixed (structural + behavioral) level기술

```
module adder4 (in1, in2, s, zero);
    input [3:0] in1;
    input [3:0] in2;
    output [4:0] sum;
    output zero;
    reg zero;
    fulladd u1 (in1[0], in2[0], 0, s[0], c0);
    fulladd u2 (in1[1], in2[1], c0, s[1], c1);
    fulladd u3 (in1[2], in2[2], c1, s[2], c2);
    fulladd u4 (in1[3], in2[3], c2, s[3], s[4]);

    always @ (s )
        if ( s == 0)
            zero = 1;
        else
            zero = 0;
endmodule
```

< fulladd module의 기술>

```
module fulladd (in1, in2, carryin, sum, carryout);
    input in1, in2, carryin;
    output sum, carryout;
    assign {carryout, sum} = in1 + in2 + carryin;
endmodule
```


-
- 수고하셨습니다.