

Digital logic circuits

2015-1학기



본 실습 수업의 목적

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

Reference

- Books
 - Verilog HDL – A Guide to Digital Design and Synthesis
 - HDL Programming Fundamentals: VHDL and Verilog (Davinci Engineering)
- Online
 - Verilog 2001 Guidebook (http://www.sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf)
 - Verilog online help (<http://verilog.renerta.com/>)

목차

1. HDL 개요
2. Verilog HDL 기초
 - 2.1 Verilog HDL 문법
3. 모듈
4. GUI Verilog Simulation-Test Bench
5. 실 습

1.HDL(High Speed IC Hardware Description Language) 개요

- HDL(Hardware Description Language) 등장 배경
 - Gate Count 및 디자인의 복잡성 증가
 - 대형회로를 설계시 기존의 설계 방법 비효율성(시간/비용 증가)
- 복잡한 하드웨어를 쉽게 기술하기 위하여 언어를 사용하는 방법 연구
 - Programming 언어 : 회로의 지연처리, 합성 등의 문제점
 - HDL : VHDL, Verilog HDL, AHPL, IDL, ISPS, UDL/I 등이 연구, 사용
- HDL의 종류
 - VHDL(Very High Speed IC Hardware Description Language)
 - 미국 국방성 중심으로 1987년 표준화
 - 1993년 IEEE 1076-1993으로 불리는 IEEE 표준 HDL new version 탄생
 - 표현 형식이 엄격하여 학계에서 주로 사용
 - Verilog HDL
 - C와 비슷한 syntax
 - Gateway Design System 사에서 개발 Cadence로 흡수
 - 약 70%이상의 기업체에서 사용
 - 실습 수업에 사용할 것은 **Verilog HDL!**

2.Verilog HDL 기초

- Verilog HDL을 이용해 다음의 3가지 기술 형태를 혼용하여 회로 상태를 기술한다
 - 구조적 모델링(Structural Modeling)
 - 가장 하드웨어적 표현에 가까움
 - 하드웨어 구성 요소들간의 상호 연결관계를 나타내어 표현
 - 데이터플로우 모델링(Dataflow Modeling)
 - Behavioral Description보다 한 단계 낮은 Level
 - expression을 사용한 combinational logic 기술; Boolean Function 또는 연산자 (AND, OR) 표현
 - 하드웨어를 자료(data)의 흐름 즉 신호 및 제어의 흐름으로 나타내어 기술
 - 동작적 모델링(Behavioral Modeling)
 - 기존의 프로그래밍 언어 스타일과 유사
 - 입력상태에 대한 출력결과만을 고려한 기술방법(Functional or Algorithm Description)
 - 시스템이 내부적으로 어떠한 하드웨어적인 구조를 가지는지에 상관없고, 오로지 진리표 등으로 표현된 것을 기능적 또는 수학적인 알고리즘을 사용하여 시스템을 동작하는 기술.

2.1 Verilog HDL 문법

- Verilog 코드는 기본적으로 C/C++과 비슷하나 본 장은 작성시 필요한 기본적인 문법에 관해서 설명한 것이 때문에 좀더 자세한 문법을 공부하거나 참조하고 싶은 학생은 online help (<http://verilog.renerta.com/>)을 참고로 공부를 한다.
- 주석문
 - C언어와 동일한 주석문을 사용
 - 한 줄 주석 : //
 - 여러줄 주석 : /*..*/
- 연산자
 - C 언어와 동일한 단항, 이항, 삼항 연산자가 사용가능함
 - Arithmetic operators: +, -, *, /, %, >>, <<
 - Relational operators: <, <=, ==, !=, >=, >, ==, !=
 - Logical operators: &&, ||, !, ?:
 - Bit-wise operators: ~, &, |, ^, ~^, ^~
 - Reduction operators: &, ~&, |, ~|, ^, ~^

2.1 Verilog HDL 문법

- Verilog는 C언어와 매우 유사하며 다음과 같은 기본적인 규칙을 가지고 있다.
 1. 대소문자를 구별하는 언어이다. 예를 들어, Cout과 COUT은 같지 않다.
 2. Verilog에서 사용하는 모든 키워드는 소문자임-> 키워드는 대소문자 조합 상관 없이 변수명 등으로 절대로 사용하지 않기
 3. 모듈, 신호 등의 이름으로 사용되는 식별자(identifier)는 알파벳(a-z, A-Z), 숫자(0-9), 밑줄문자(_), \$ 문자로 구성된다. 단, 숫자와 \$로 시작할 수 없다. 그리고 길이는 1024문자까지 가능하다.
 4. Verilog의 각 문장은 대개 세미콜론으로 끝난다. 예외적으로 키워드 endmodule는 세미콜론으로 끝나지 않는다.
 5. C언어와 같이 두개의 연속된 슬래쉬 //는 줄 끝까지가 주석(comment)임을 나타내어 줄 단위의 주석에 사용되며 /*와 */는 주석의 시작과 끝을 나타내어 여러 줄의 주석을 기술하는 데에 사용될 수 있다.

2.1 Verilog HDL 문법

- 데이터 형태
 - 논리 값

논리값	의미
0	논리적 0 값 또는 조건 False
1	논리적 1 값 또는 조건 True
X	Unknown 논리 값
Z	High impedance

- 추상적 데이터

변수 선언	의미
integer Count	32 비트 integer로 Count 변수선언
Integer K[1:63]	배열 선언

2.1 Verilog HDL 문법

- 수의 표현

- 크기 지정 가능 수 (sized)

<size> ' <base format> <number 숫자>

4'b111 // 4비트 2진수, base format은 binary

12'habc // 12비트 16진수

16'd255 // 16비트 10진수

- 크기 지정 불가능 수(unsized)

- Base format 이 없는 경우는 기본적으로 십진수(decimal)를 의미

- Size가 없는 경우는 기본적으로 32bit의 수를 의미

- Base format

- Decimal : 'd 또는 'D

- Hexadecimal : 'h 또는 'H

- Binary : 'b 또는 'B

- Octal : 'o 또는 'O

2.1 Verilog HDL 문법

- 음수의 표현
 - <size> 앞에 - 부호를 위치하여 표현
 - 예 : `-8'd3` // 3의 2의 보수 형태로 저장된 8bit의 음수
 - 틀린 예 : `4'd-2`
- 수의 확장 법칙
 - MSB가 0, X, 또는 Z 인 경우
 - 각 각 MSB가 0, X, Z가 되도록 확장
 - 예 : `3'b01 = 3'b001`, `3'bx1 = 3'bxx1` , `3'bz = 3'bzzz`
- MSB가 1인 경우
 - MSB를 0으로 채우는 방향으로 확장, 예 : `3'b1 = 3'b001`
- 숫자에서 언더스코어는 가독성을 높여주는 역할로 Verilog에서 무시됨, 숫자에서 물음표는 하이임피던스
 - `12'b1111_0000_1010` ///// `12'b111100001010`과 동일
 - `4'b10??` // `4'b10zz`와 동일

2.1 Verilog HDL 문법

- 데이터 형태 (변수 선언)
 - wire : net (배선)
 - 하드웨어 요소사이에 연결을 나타낸다.
 - 값 저장이 불가능하며 assign 문에서 사용한다.
 - constant
 - 파라미터를 이용하여 변수 값을 사용할 때 사용한다.

키워드	정의
wire, tri	일반적인 net에 사용
wor, trior	OR 연산을 하는 wire
wand, triand	AND 연산을 하는 wire
triereg	값을 저장하는 net에 사용
tri1	net에 아무 것도 인가되지 않으면 1
tri0	net에 아무 것도 인가되지 않으면 0
supply1	Vdd를 나타냄
supply0	Ground를 나타냄

2.1 Verilog HDL 문법

- 데이터 형태 (변수 선언)
 - reg : register (latch 또는 flip-flop)
 - 일시적으로 데이터를 저장하는 변수를 의미한다.
 - 하드웨어 레지스터를 의미하지는 않는다
 - always, initial 문 등에서 사용한다.

키워드	정의
reg	일반적으로 사용하는 레지스터
real	실수 표현을 하는 레지스터
time	시뮬레이션 시간을 저장하는 레지스터
integer	정수 표현을 하는 레지스터, signed 형태 사용가능

2.1 Verilog HDL 문법

- 데이터 형태
 - Vectors: multiple-bit data
 - net 또는 reg 데이터 타입은 벡터(multiple bit width)로 선언할 수 있다.
 - Vector는 [MSB#, LSB#] 혹은 [LSB#, MSB#]와 같이 선언이 되며, 괄호안 왼쪽이 Vector의 최상위 비트가 된다. 그러므로 virtual_addr의 예에서 bit0가 최상위 비트가 되는 것이다. 앞의 선언중에 아래와 같이 그 일부분만 사용할 수도 있다. 만일 Bit 수가 명시 되지 않았다면, 기본으로 Scalar(1-bit)가 된다.
 - 예) wire a; //Scalar net variable, default
wire [7:0] bus; //8-bit bus
busA[7] // bit #7 of vector busA
wire [31:0] busA, busB, busC; //3 buses of 32-bit width
reg clock; //Scalar register, default
reg [0:40] virtual_addr; //Vector register, virtual address 41bts width

2.1 Verilog HDL 문법

- 데이터 형태
 - Time
 - Verilog의 Simulation은 simulation time의 관점에서 이루어진다. 특별한 Time Register Data Type이 시간을 저장하기 위해 사용된다.
 - 시간변수는 time이라는 Keyword를 통해서 선언되며, \$time은 현재의 Simulation 시간을 알아내고자 할 때 사용된다.
 - Simulation Time은 **second**의 관점에서 측정된다. 이 단위는 's'로 표기가 되며, 실제시간과 일치한다. 그러나 실제 회로에서의 시간과 Simulation상의 time 설정은 사용자에게 달려있다

2.1 Verilog HDL 문법

- Array
 - Verilog에 있어서 Array는 오직 reg, integer, time 그리고 vector에 대해서만 허용되며, 실변수(real variable)에 대해서는 사용할 수 없다.
 - <array_name> [<subscript>]와 같은 형태로 사용할 수 있으며, 다중 Array는 Verilog 2001에선 가능 하다.

예) integer counter [3:0];

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

```
time chk_point[1:100];
```

```
reg [15:0] array [0:255][0:255] ///// Verilog 2001 allows multi dimension arrays
```

```
reg [4:0] port_id[0:7]; // array of 8port_ids ; each port_id is 5 bits width
```

Verilog HDL 에서 벡터와 배열의 차이점

– vector is a **single element** with n-bits wide

– arrays are **multiple elements** that are 1-bit (default) or n-bits wide

2.1 Verilog HDL 문법

- 문자열(strings)
 - Double quote(“”)를 사용하여 문자열을 만들고 하나의 line에 표현되어야 한다.
 - 각각의 문자는 1 바이트로 표현되고 레지스터에 저장될 수 있다.

예: `reg [8*18:1] string_value; //Declare a variable that is 18 bytes wide`

`initial`

`string_value = “Hello Verilog World”; //String can be stored in variable`

2.1 Verilog HDL 문법

- 파라미터 (Parameters)
 - Verilog에서는 **parameter**라는 Keyword를 통해서 상수(constant)정의가 가능하다
 - 회로의 비트 크기 또는 지연값을 지정하기 위해 사용한다.
 - 자료형과 범위지정을 가질 수 있으며, 범위가 지정되지 않은 경우, 상수 값에 적합한 크기의 비트 폭을 default로 가진다.

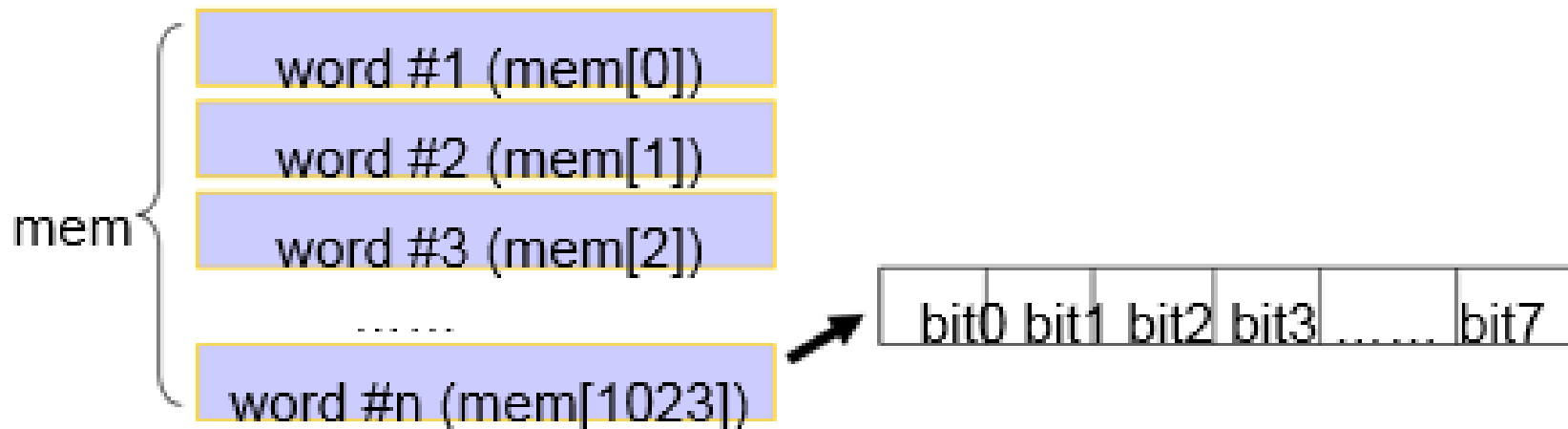
예)

```
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 ;
```

2.1 Verilog HDL 문법

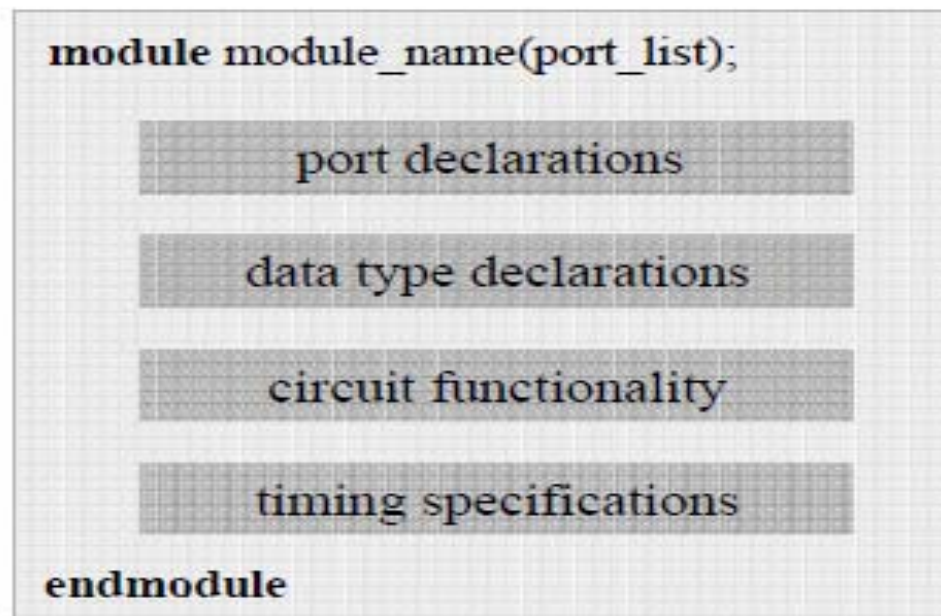
- 메모리(memory)
 - Digital Simulation에선 Register File, RAM, ROM과 같은 Modeling을 자주 하게 된다.
 - Memory는 Verilog상에서 간단히 Register Array를 사용하여 해결할 수 있다. Array에 대한 각각의 요소를 word라고 하며 각각의 word는 1-bit 또는 그 이상일 수 있다.

예) `reg [7:0] mem [0:1023] ; // 1K 8bit words`



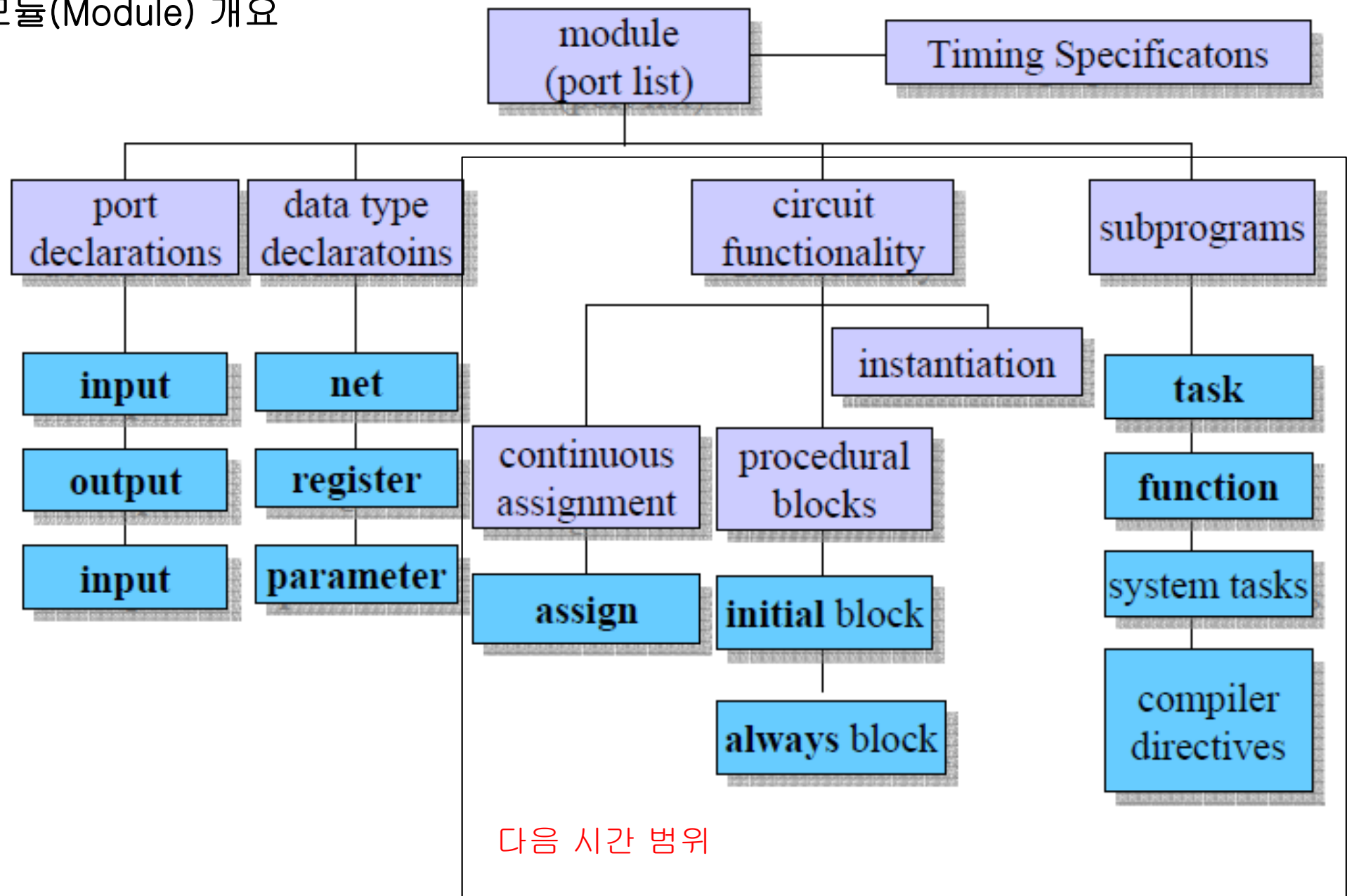
■ 모듈(Module)

- Verilog HDL 프로그램의 기본 구조이다.
- 모듈은 C의 함수(function) 단위와 비슷하며, 베릴로그 설계를 위한 기본적인 블록 단위이다.
- `module <모듈 이름> (포트 목록);` 으로 시작하며, `endmodule`로 끝난다
- 포트 인터페이스(입력, 출력)를 통해서 상위 수준의 블록에 필요한 기능 제공한다.
- Timing specification은 시뮬레이션을 위해서 사용한다.

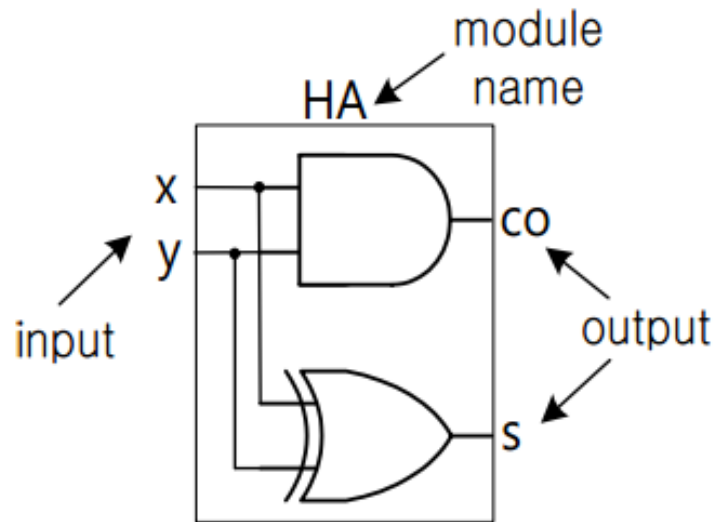


3.Verilog HDL ► 모듈

◆ 모듈(Module) 개요



- HA (Half Adder)의 모듈 설계



진리표

x y	co s
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

```
module HA (sum, co, x, y);  
    input x, y;  
    output sum, co;  
    wire c_out_bar;  
    xor (s, x, y);  
    nand (c_out_bar, x, y);  
    not (co, c_out_bar);  
endmodule
```

3.Verilog HDL ► 모듈

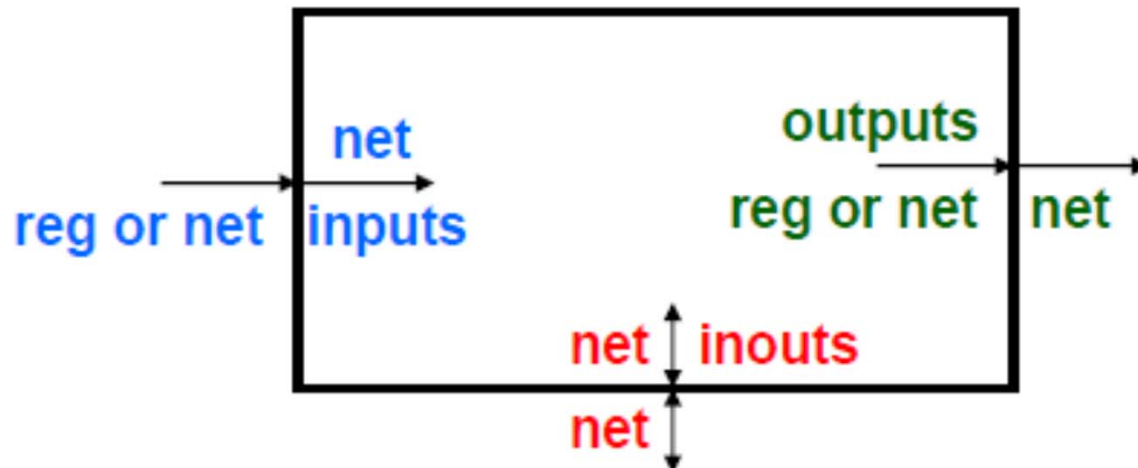
- 모듈의 포트(1)
 - 모듈과 외부와 데이터를 주고 받을 수 있도록 해 준다
 - 각 포트의 모드 세가지: input, output 또는 inout포트는 모듈과 모듈을 연결할 수 있는 인터페이스 이다.
 - <키워드> [범위] <포트 이름> 의 형태로 선언하며, 그 종류에는 input, output, inout 이 있다.
 - input은 입력포트이며, output은 출력 포트, inout은 양방향 포트이다
 - 모든 포트는 wire로 선언된다. 단, output 포트의 경우 포트의 값 변경(또는 유지)이 필요하다면 같은 이름으로 reg를 선언해야한다.

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
```

3.Verilog HDL ► 모듈

- 모듈의 포트(2)
 - 포트를 연결할 때는 다음과 같은 규칙을 따른다.
 - 내부 입력 포트는 반드시 net 형, 외부 입력 포트는 reg 또는 net 변수와 연결될 수 있다.
 - 내부 출력 포트는 reg 또는 net 형, 외부 출력 포트는 반드시 net 변수와 연결되어야 한다.
 - 내부 양방향성(입출력) 포트는 반드시 net 형, 외부 양방향성 포트는 반드시 net 변수와 연결되어야 한다.
 - 같은 비트 수끼리 연결해야 한다.



3.Verilog HDL ► 모듈

- 모듈의 포트 연결 방법

- 포트를 연결할 때는 c언어에서 함수에 인자 값 전달하듯이 인자 순서대로 연결할 수 있으며, 또는 순서에 상관없이 이름에 각각 할당할 수 있다
 - Width matching: 서로 다른 크기의 내/외부 포트를 연결하는 것이 가능
 - Unconnected ports: 연결되지 않는 포트는 “,”를 사용하여 표시

예:

```
module adder(x, y, c_out, c_in, sum);
```

```
    // ...
```

```
endmodule
```

```
// 위치에 의한 포트 연결
```

```
adder add1(inp1, inp2, carry_out, carry_in, sum_out);
```

```
//이름에 의한 포트 연결
```

```
adder
```

```
add2(.sum(sum_out), .c_out(carry_out), .c_in(carry_in), .x(inp1), .y(inp2));
```

```
adder adder1(inp1, , carry_out, carry_in, sum_out); //두번째 포트 연결하지 않음
```

4.Verilog HDL ► Test Bench

- 테스트벤치는 HDL 로 설계한 논리회로를 입력값과 출력값을 WAVE form 형태의 그래프로 출력하여 검증할 수 있다.
- 테스트 모듈에서 검증하고 싶은 모듈을 호출하여 와이어 들의 값을 확인할 수 있다.
 - 테스트 벤치용 파일에서 호출하거나 또는 모듈안에서 선언해서 사용한다
- 테스트 벤치는 사용자가 시간을 지정해 줄 수 있다.
 - ‘#숫자’로 표기된 것이 시간 단위이며 단위는’ns’로 나노초이다.
- Verilog HDL에는 default timescale 값이 지정되어 있지 않다.
 - 시뮬레이션에서 timescale 값이 명시적으로 지정되지 않으면 상대적인 지연 값으로 계산 이를 방지하기 위하여 시뮬레이션이 시작하는 첫 위치에 `timescale을 정의하는 것이 필요하다. 문장의 끝에 세미콜론이 붙지 않는다.
- 사용법
 - ‘timescale<참조시간 단위>/<시간 정밀도>
 - 참조시간 단위 : 각 단위에 대한 시간 밑수로 그 범위는 초(s)부터 femtosecond(fs)까지 다음중 하나를 사용 s ms us ns ps fs
 - 시간 정밀도 : 시뮬레이션 동안 반올림된 지연의 정확도

예) ‘timescale 1ns/10 ps 지연 시간의 1 단위는 1 ns이며 정확도는 0.01 ns(= 10 ps)

4.Verilog HDL ► Test Bench

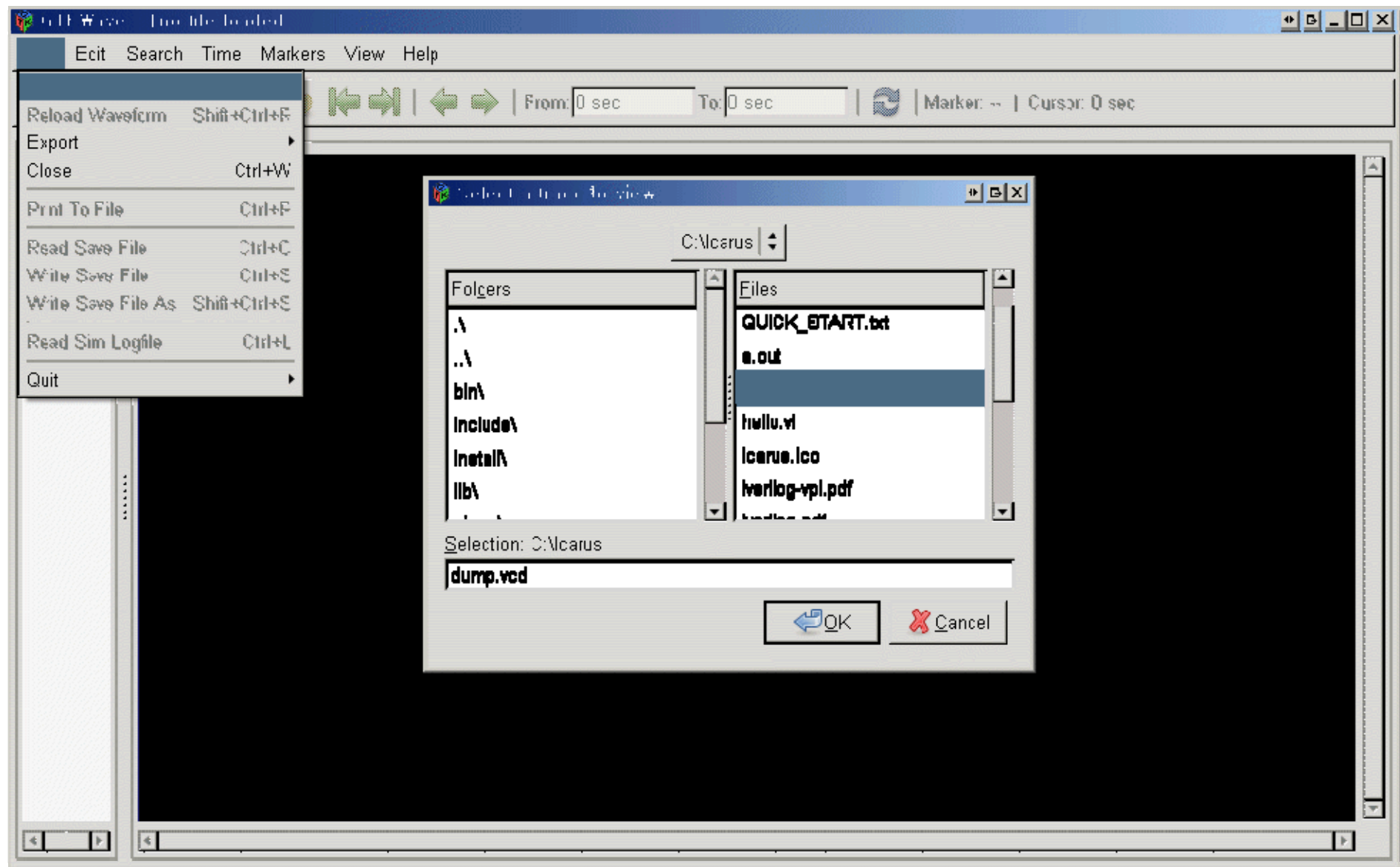
- 테스트벤치에서는 시스템 태스크 키워드(system task keyword)들이 사용된다. 그 종류는 다음과 같다.
 - \$display : 화면 출력을 위한 키워드로 변수 값, 문자열, 수식 등을 출력하는 용도로 사용한다. C언어의 printf와 용법이 비슷하다.
 - \$monitor : 모니터링 태스크이다. 연결한 레지스터나 와이어의 값이 변할 때마다 그 값을 계속해서 출력한다.
 - \$stop : 시뮬레이션을 중단하거나, 신호의 값을 알아내고자 할 때 사용한다.
 - \$finish : 시뮬레이션을 멈추기 위해 사용한다.
 - \$time : 시뮬레이션의 현재 시간을 나타낸다

4.Verilog HDL ► 실습

- 컴퓨터에 Verilog HDL 실습을 위한 Verilog Simulator – Icarus 을 설치한다.
 1. <http://bleyer.org/icarus/>에 들어 가서 iverilog-0.9.7_setup.exe (latest stable release)를 다운로드 받아서 실행한다.
 2. 설치 위치는 “C:\W\icarus”입력하여 설치한다.
 3. Command창 실행. ‘시작->실행’ 메뉴 실행후 ‘cmd’, 확인
 4. 커맨드모드 창에서 icarus가 설치된 디렉토리로 이동 (cd c:\W\icarus) 후
예제파일인 lfsr16.v실행
iverilog lfsr16.v <enter>
vvp a.out <enter>
gtkwave <enter>
 5. 화면상에서 결과값이 출력되고 완료되면 커맨드 창에서 gtkwave를 입력후 엔터

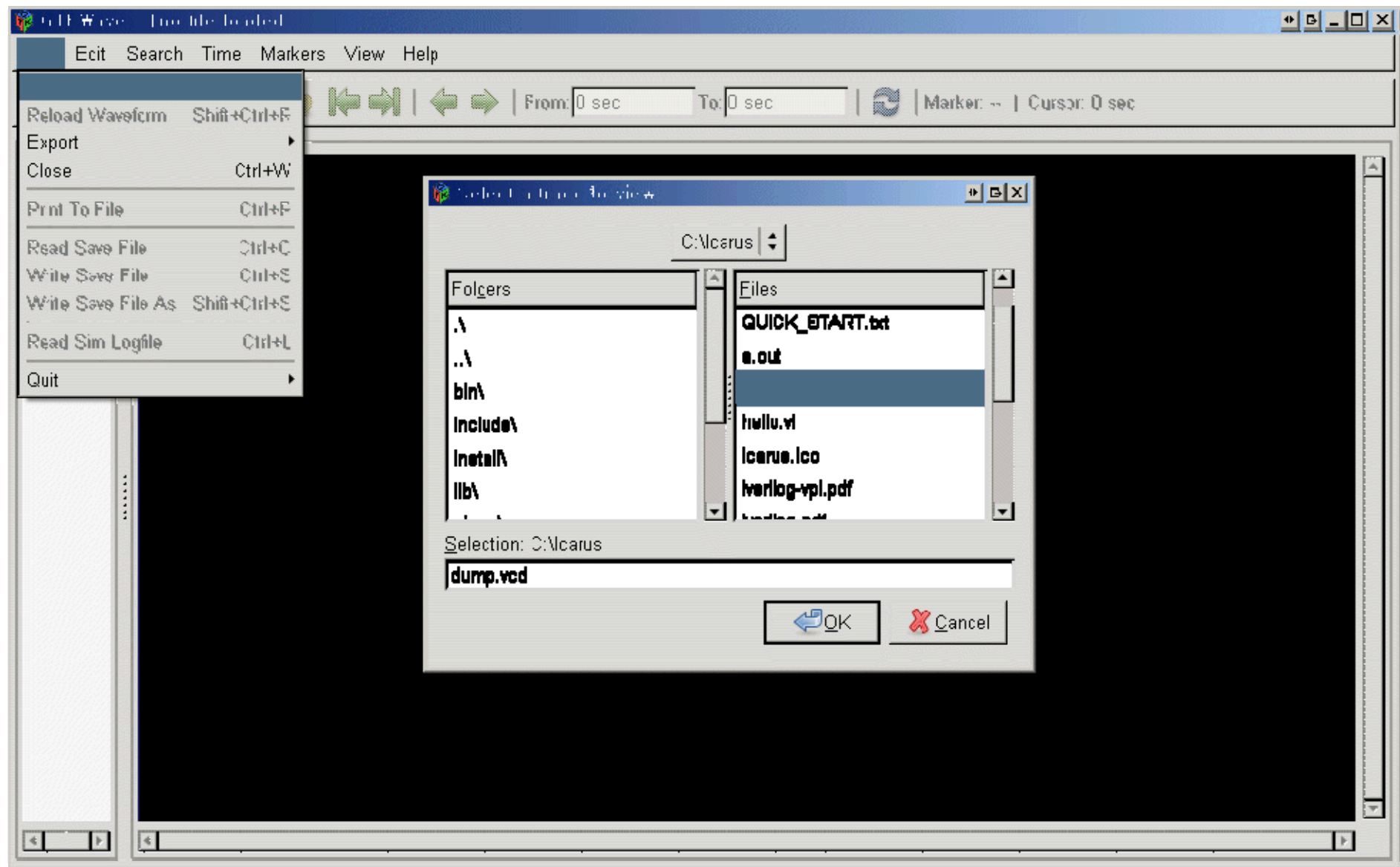
4.Verilog HDL ▶ 실습

결과파일'dump.vcd' 로드



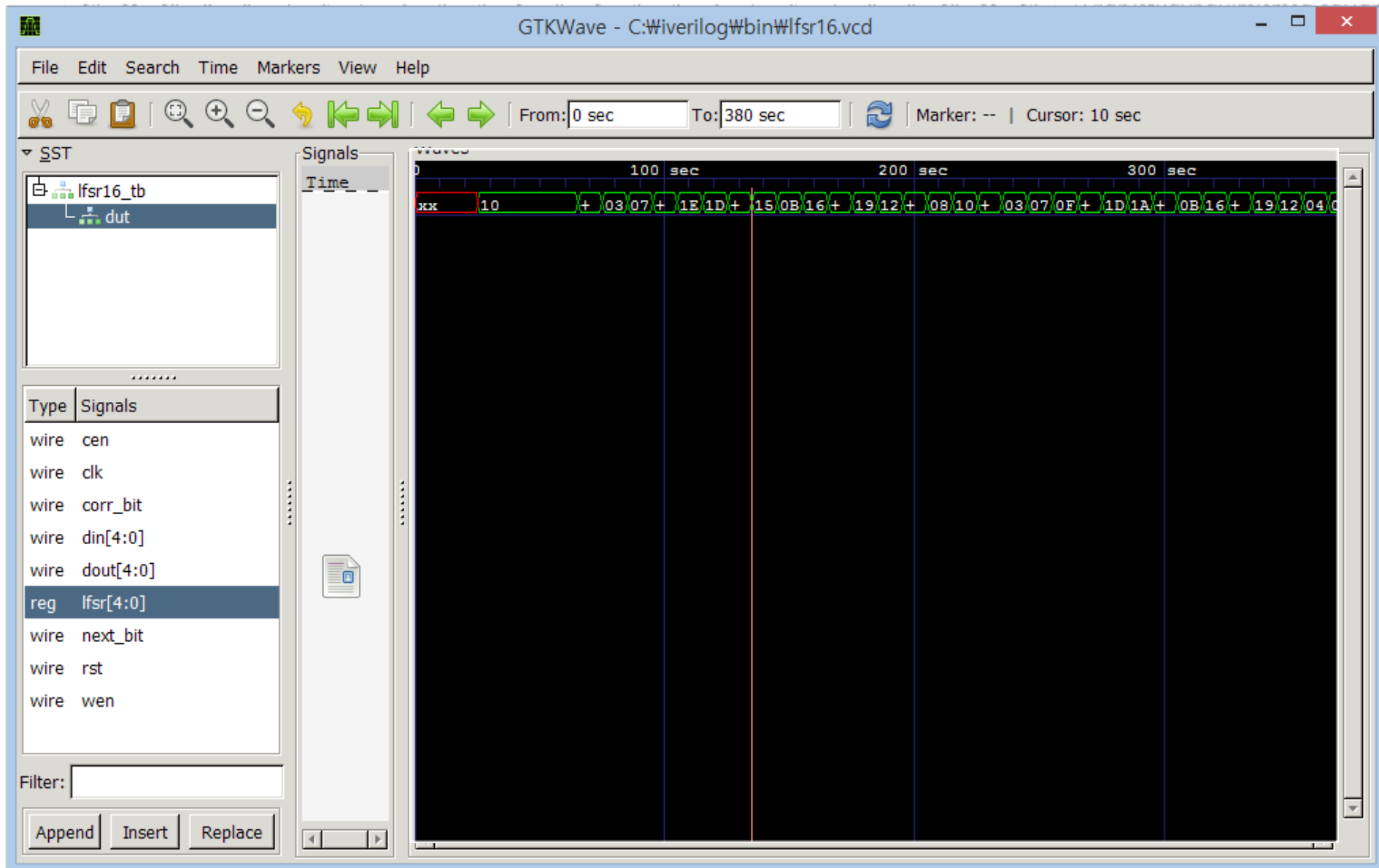
4.Verilog HDL ▶ 실습

결과파일'dump.vcd' 로드



4. Verilog HDL ▶ 실습

보기를 원하는 포트를 클릭하여 파형을 확인



Digital logic circuits

2015-1학기



목차

1. Gate Level Modeling

1. Primitive Gates

2. Gate Delay

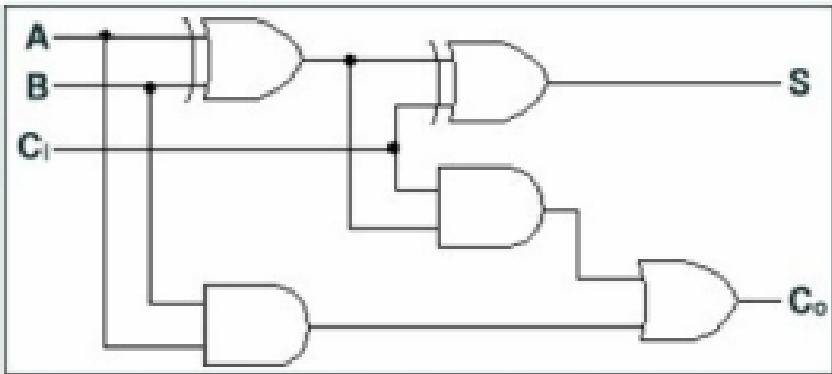
2. Dataflow Modeling

1. Gate Level Modeling 개요

- 구조적 모델링(Structural Modeling)
 - 가장 하드웨어적 표현에 가까움
 - 하드웨어 구성 요소들간의 상호 연결관계를 나타내어 표현
 - 논리 게이트, 플립플롭등을 사용한 연결 표현
- > Gate Modeling 이라고도 함

1.1 primitive gate 개요

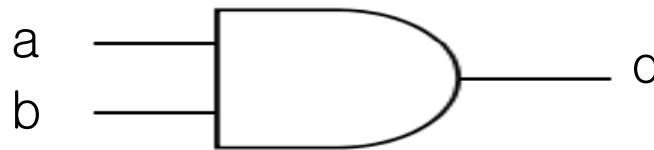
- Verilog 에서는 기본적인 논리 게이트들이 primitive 정의되어 있음
 - 모듈의 정의없이 이름 그대로 사용하면 된다
 - > and or xor nand nor xnor
 - > buf and not gates
 - Combinational logic primitive 회로의 아웃풋은 net으로 정의되어야 함
 - combinational logic primitive 회로의 인풋은 net 또는 wire 으로 정의



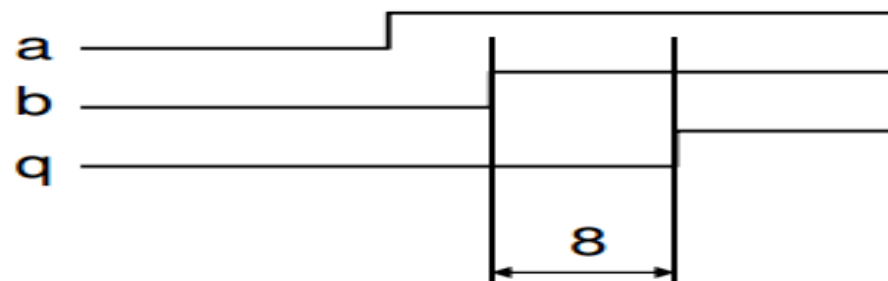
```
//1bit 전가산기의 정의
module fulladd(sum, c_out, a, b, c_in);
//I/O 포트 선언
    output sum, c_out; input a, b, c_in;
//내부 넷
    wire s1, c2, c2;
//프리미티브 논리 게이트 사용
    xor(s1, a, b);
    and(c2, a, b);
    xor(sum, s1, c_in);
    and(c2, s1, c_in);
    or(c_out, c2, c1);
endmodule
```

1.2. 게이트 지연

- Verilog-HDL에 기본 게이트는 전파 지연(propagation delay) 없음.
 - 전파지연 : 입력 신호의 변화와 이에 대한 응답 출력 변화가 나타나기 까지 걸리는 시간
- 사용자가 게이트 모델에 지연 시간을 정의 가능
- e.g.) `and #(delay_time) a1(out, a, b);`

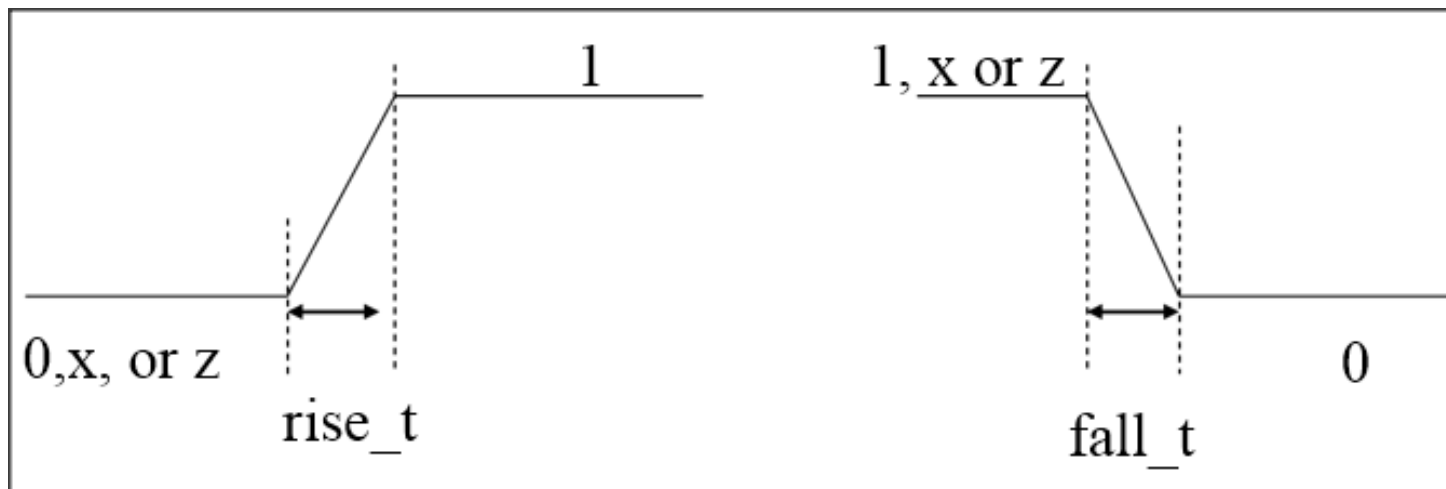


```
and #(8) n1(q, a, b);
```



1.2. 게이트 지연

- Gate Delays의 종류
 - 상승 지연 : 0→1로 게이트의 출력이 변할 때
 - 하강 지연 : 1→0으로 게이트의 출력이 변할 때
 - e.g.) and # (4,5) a1(out, a, b); //상승 지연 =4 지연 =5단위
 - 턴-오프 지연 : 어떤 값에서 임피던스 값(z)로 출력이 변할 때
 - e.g.) and # (rise_t, fall_t, turn_off_t) a3 (out, i1, i2);



1.2. 게이트 지연

- Gate Delays: min, typical, max
 - 최소: 설계자가 예상한 게이트의 최소 지연값
 - 전형: 설계자가 예상한 보통의 지연값지연
 - 최대:설계자가 예상한 게이트의 최대 지연값
 - rise, fall, and turn-off time 값은 min, typical and max 으로 표현이 가능함

e.g.) and #(2:3:4, 3:4:5, 4:5:6) a3(out, a, b);

//2:3:4 - 상승 지연값에 대한 min, typical, max 값

//3:4:5 - 하강 지연값에 대한 min, typical, max 값

//4:5:6 - 턴-오프 지연값에 대한 min, typical, max 값

1.2. Verilog HDL ► 실습

다음 코드를 Verilog HDL로 실행하여 테스트 벤치를 통해 지연 발생 파형을 확인해 보자.

```
module multiplexor_2_to_1(out, cnt, a, b);  
    output out;  
    input  cnt, a, b;  
    wire   not_cnt, a0_out, a1_out;  
  
    not # 2  n0(not_cnt, cnt);      /* Rise=2, Fall=2, Turn-Off=2 */  
    and #(2,3) a0(a0_out, a, not_cnt); /* Rise=2, Fall=3, Turn-Off=2 */  
    and #(2,3) a1(a1_out, b, cnt);  
    or  #(3,2) o0(out, a0_out, a1_out); /* Rise=3, Fall=2, Turn-Off=2 */  
  
endmodule /* multiplexor_2_to_1 */
```

2. Data Flow Modeling

- 동작적 모델링(behavioral modeling)은 회로의 내부 구조 대신에 회로가 수행하는 기능을 기술한다.
 - 회로가 어떻게(how) 구성되는 지가 아니라 회로가 무엇을(what) 수행할 것인지를 기술한다.
- 동작적 모델링에서 회로의 동작은 부울 함수와 수식으로 기술할 수도 있고 알고리즘과 같은 추상적인 표현을 사용하여 나타낼 수도 있다. 이 중에서 부울 함수와 같은 수식으로 기술하는 모델링을 데이터흐름 모델링(data flow modeling)이라고 한다.
- 데이터 모델링에서는 게이트 보다 주로 신호와 값에 중점을 둔다.
 - 자료형으로 net과 register형을 사용한다.
 - net형의 자료형은 여러 가지가 있지만 주로 wire형을 사용하며 register형은 reg형과 integer형을 주로 사용한다.
 - wire와 reg는 기본적으로 1비트의 크기를 가지며 integer형은 기본적으로 32비트의 크기를 갖는다

2. Data Flow Modeling

- 연속 할당문(Continuous Assignments)
 - 부울함수로 표현되는 논리회로를 결과값들을 중심으로 매우 간단하게 기술할 수 있다
 - Net에 특정 논리 값을 지정하는데 사용한다.
 - 키워드 “assign”으로 시작
 - `assign <drive_strength> <delay> <list of assignments> ;`
 - C언어에서 사용하는 것과 같은 연산자와 할당문을 사용한다.
- e.g) `assign y = ~a & b | a & ~c; // 부울함수 $y = a' b + a c'$`

2. Data Flow Modeling

- 연속 할당문의 형태
 - 여러 개의 할당문을 포함한 연속 할당문
 - assign문은 하나 이상의 할당문들을 콤마로 구분하여 함께 나타낼 수 있다. 다음은 3개의 연속할당문을 하나의 assign문으로 나타낸 것이다

e.g.) assign lt = \sim a & b,

gt = a & \sim b,

eq = \sim a & \sim b | a & b;

2. Data Flow Modeling

- 연속 할당문의 형태
 - 목시적인 연속할당문
 - 연속할당문을 통해서 값이 지정되는 신호는 일반적으로 wire형으로 선언한 후에 assign문을 사용하여 값을 할당한다.
 - output 신호는 기본적으로 wire형으로 다루어지므로 별도로 wire형으로 선언할 필요가 없다.

```
wi re out = a & b | c & d;
```

위의 표현은 다음의 두 문장을 합쳐서 기술한 것이다.

```
wi re out;
```

```
assi gn out = a & b | c & d;
```

2. Data Flow Modeling

- 연속할당문의 제한점
 - 연속할당문은 간단한 부울 함수, 3상태 출력 동작과 D래치 회로와 같이 작은 회로를 모델링하는 데에는 매우 편리하다.
 - 그렇지만 입력 개수가 많고 복잡한 회로는 부울 함수로 표현하는 것이 쉽지 않고 부울 함수로 표현했더라도 설계의 내용을 이해하기 어려운 경우가 많다.
 - 큰 회로는 연속할당문을 사용한 데이터흐름 모델링 방법 보다는 완전한 동작적 모델링을 사용하는 것이 설계하기도 쉽고 설계된 내용을 이해하기도 쉽다..

2. Data Flow Modeling

- 여러가지 연산자
 - Verilog에서는 동작적 모델링을 위해서 다양한 연산자를 제공한다
 - C언어의 연산자와 같지만 C언어에서 제공하지 않는 연산자도 제공한다
 - Verilog 제공 연산자
 - 비트연산자: \sim , $\&$, $|$, \wedge , $\sim\wedge$, $\wedge\sim$
 - 축소연산자: $\&$, $\sim\&$, $|$, $\sim|$, \wedge , $\sim\wedge$
 - 결합연산자: $\{\}$
 - 산술연산자: $+$, $-$, $*$, $/$, $\%$,
 - 쉬프트 연산자 : $>>$, $<<$
 - 관계연산자: $<$, $<=$, $=$, $!=$, $>=$, $>$, $===$, $!==$
 - 논리연산자: $\&\&$, $||$, $!$, $?:$

2. Data Flow Modeling

- 비트단위 논리연산자
- 다음과 같은 비트단위 논리연산자가 있으며 XNOR 연산자를 제외하면 C언어와 같다.
 - \sim NOT 연산자
 - $\&$ AND 연산자
 - $|$ OR 연산자
 - \wedge XOR 연산자
 - $\wedge \sim$ 또는 $\sim \wedge$ XNOR 연산자
- 비트단위 논리 연산자의 우선순위는 \sim , $\&$, \wedge , $|$ 순서이다.
- XNOR연산자는 XOR연산자와 우선순위가 같다.
- 각 비트단위 논리연산자는 대응되는 게이트들이 있으므로 이 연산자들을 사용하여 표현된 식은 게이트 수준의 회로로 쉽게 합성될 수 있다

2. Data Flow Modeling

- 비트단위 논리연산자 예제

// X = 4'b1010. Y = 4'b1101

// Z = 4'b10x1

~X // 부정. 결과값 4'b0101

X & Y // 비트단위 and. 결과값 4'b1000

X | Y // 비트단위 or. 결과값 4'b1111

X ^ Y // 비트단위 xor. 결과값 4'b0111

X ^~ Y // 비트단위 xnor. 결과값 4'b1000

X & Z // 결과값은 4'b10x0

2. Data Flow Modeling

- 축소연산자
 - 축소연산자(reduction operator)는 벡터 피연산자의 모든 비트에 대해서 논리연산 수행하여 1비트의 결과를 제공하는 것으로서 다음과 같은 것들이 있다.
 - &x reduction AND
 - ~&x reduction NAND
 - |x reduction OR
 - ~| reduction NOR
 - ^x reduction XOR
 - ~^, ^~ reduction XNOR

2. Data Flow Modeling

- 축소연산자 예제

// $X = 4'b1010$

$\&X$ // $(1 \& 0 \& 1 \& 0)$ 을 계산, 결과값 $1'b0$

$|X$ // $(1 | 0 | 1 | 0)$ 을 계산, 결과값은 $1'b1$

$\^X$ // $(1 \^ 0 \^ 1 \^ 0)$ 을 계산, 결과값은 $1'b0$

// 축소 연산자 xor나 xnor는 벡터의 짝수/홀수 패리티 확인에 사용

2. Data Flow Modeling

- 결합연산자
 - 결합연산자(concatenation operator) { } 는 다음과 같이 여러 개의 피연산자를 결합하여 더 큰 크기의 벡터를 만드는 데 사용한다.
 - 결합연산자를 사용하기 위해서는 피연산자의 크기가 정해져 있어야 한다. 그리고 다음과 같이 숫자를 앞에 사용하면 반복하여 결합이 된다.

e.g) { 1'b1, 2'b01 } // 결과: 3'b101

{ 3{ A } } // { A, A, A }

{ A, 2{B} } // { A, B, B }

{ 2{3'b011} } // 6'b011011

2. Data Flow Modeling

- 등가연산자는 값의 비교 방법에 따라서 다음과 같이 두 종류가 있다.
 - 논리 등가 연산자: `==` (등가), `!=` (부등가)
- `x` 또는 `z`값과 비교할 때에는 결과가 `x` 이다.
 - 케이스(case) 등가 연산자: `===` (등가), `!==` (부등가)
- `x` 또는 `z`값도 정확하게 비교하며 결과는 0 또는 1이다.
- 다음은 여러 가지 등가연산자의 사용 예이다.
 - `4'b1010 == 4'b1xxz // x`
 - `4'b1010 != 4'b1xxz // x`
 - `4'b0010 == 4'b1xxx // 0 (거짓)`
 - `4'b1010 !== 4'b1xxz // 1 (참)`

2. Data Flow Modeling

- 다음과 같은 C 언어와 같은 논리연산자가 제공된다.
 - && 논리 AND
 - || 논리 OR
 - ! 논리 NOT
- 논리연산자의 결과는 1비트로서 참(1)과 거짓(0)을 나타낸다. 피연산자가 0이 아닌 경우에 참으로 인식하지만 피연산자가 x 또는 z값을 가질 경우에 거짓으로 인식한다.
- 이에 비해서 비트단위 논리연산자는 결과의 크기가 피연산자의 크기와 같다. 논리연산자는 논리 게이트를 사용하여 쉽게 합성된다

2. Data Flow Modeling

- 논리 연산 예시
 - `A = 3; B = 0;`
 - `A && B` // 결과값은 0, (논리-1 && 논리-0)을 계산
 - `A || B` // 결과값은 1, (논리-1 || 논리-0)을 계산
 - `!A` // 결과값은 0, not(논리-1)을 계산
 - `!B` // 결과값은 1, not(논리-0)을 계산
 - `// 알 수 없는 값`
 - `A = 2'b0x; B = 2'b10;`
 - `A && B` // 결과값은 x, (x && 논리1)을 계산
 - `// 수식`
 - `(a == 2) && (b == 3)` // (a == 2)와 (b == 3)이 모두 참이면, 결과값은 1
 - `// 둘 중의 하나만이라도 거짓이면, 결과값은 0`

2. Data Flow Modeling

- 연산자 우선 순위

연산자	연산자의 기호	우선 순위
단항 연산자. 곱셈, 나눗셈, 나머지	$+, -, !, \sim$ $*, /, \%$	가장 높은 우선 순위
덧셈, 뺄셈, 자리 이동	$+, -, <<, >>$	
관계 등가	$<, <=, >, >=$ $==, !=, ===, !==$	
축소 논리	$\&, \sim\&, \wedge, \wedge\sim, , \sim $ $\&\&, $	
조건	$?:$	가장 낮은 우선 순위



2. Data Flow Modeling

- 연속할당문과 전파지연
- 게이트 레벨에서 전파 지연을 줄 수 있는 것처럼 다음과 같이 assign 다음에 #을 사용하여 회로의 전파지연 시간을 지정할 수 있다.
 - `assign #10 out = a & b | c & d;`
 - 연속 할당문에서 전파지연 시간 보다 짧은 펄스 신호는 무시된다.
- 전파지연은 연속할당문 대신에 넷을 선언할 때에도 지정할 수 있다. 다음의 세 가지 예는 같은 효과를 나타낸다.

`wire #10 out = a & b | c & d; // 묵시적 할당문에 전파지연 지정`

`wire #10 out; // 넷을 선언할 때에 전파지연 지정`

`assign out = out = a & b | c & d;`

`wire out; // 연속할당문에서 전파지연 지정`

`assign #10 out = a & b | c & d;`

2. Data Flow Modeling

- 멀티플렉서
- 멀티플렉서(multiplexer)는 여러 개의 입력들 중에서 선택신호의 값에 따라서는 한 입력을 선택하여 출력으로 전달하는 조합회로이다. 2개의 데이터 입력 a , b 와 제어입력 s , 그리고 출력 y 를 갖는 2×1 멀티플렉서의 블록도는 그림 1과 같으며 회로의 동작은 다음 표2 와 같다.

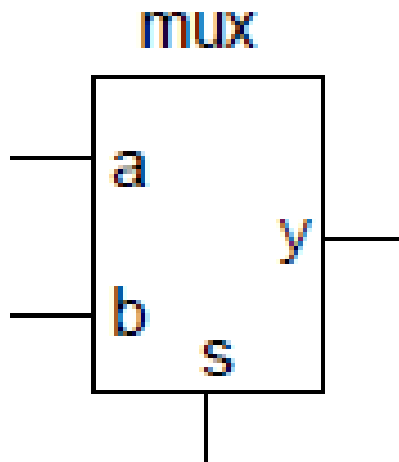


그림1. 2×1 멀티플렉서의 블록도와 동작

s	y
0	a
1	b

표 2. 2×1 멀티플렉서의 동작

2. Data Flow Modeling

- 조건 연산자를 이용한 멀티플렉서

```
module mux2_4(a, b, s, y);  
    input [3:0] a, b; // 2개의 4비트 데이터 입력  
    input s; // 선택 입력  
    output [3:0] y; // 4비트 데이터 출력  
    assign y = s ? b : a; // s=0이면 y=a, s=1이면 y=b  
endmodule.
```

2. Data Flow Modeling

- parameter를 사용한 상수 정의
 - 설계에 따라서 바뀔 수 있는 상수 값은 다음과 같이 parameter를 사용하여 기호상수로 정의를 하면 다른 모듈에서 이 모듈을 인스턴스로서 사용할 때에 기호상수의 값을 바꾸어 사용할 수 있다.
 - 따라서 이식성, 소스의 가독성, 확장성, 재구성 가능성 측면에서 장점을 가지고 있다.
- 멀티플렉서 예제를 다시 작성

```
module mux2(a, b, s, y);  
    parameter SIZE = 4; // parameter상수 정의  
    input [SIZE-1:0] a, b; // 데이터 입력  
    input s; // 선택 입력  
    output [SIZE-1:0] y; // 데이터 출력  
    assign y = s ? b : a; // s=0이면 y=a,  
    s=1이면 y=b  
endmodule
```

2. Data Flow Modeling

- parameter를 사용한 상수 정의
 - parameter를 포함한 모듈을 인스턴스로서 사용할 때에 필요에 따라서 모듈에서 정의된 parameter의 값을 다시 지정할 수 있으며, parameter값을 지정하지 않으면 모듈에서 정의된 파라미터 값이 사용된다.
 - 모듈 인스턴스에서 다음과 같이 모듈 이름 다음에 #(parameter값)을 넣어서 parameter값을 지정할 수 있으며 parameter가 여러 개인 경우에는 정의된 순서대로 값들을 콤마(,)로 구분하여 넣는다.

e.g.) mux2 #(1) u1 (d0, d1, sel, out);

- 여러 개의 parameter가 정의된 모듈의 인스턴스에서 일부 parameter의 값만 다시 지정하려면 다음의 두 가지 방법과 같이 #() 안에 또는 defparam을 사용하여 parameter이름과 값을 함께 명시하는 방법을 사용하는 것이 좋다.

e.g.) mux2 #(.SIZE(16)) u2 (a, b, sel, y); // u2의 SIZE를 16으로 지정

mux2 u3 (x, y, sel, z); // u3의 SIZE를 8로 지정

defparam u3.SIZE = 8;

3.Verilog HDL ► 실습

1. 다음과 그림 1과 같은 n-비트 비교기가 있다. 블록도로 나타낸 회로의 동작은 다음 표2와 같이 두개의 입력 A,B를 비교해서 세출력 중 하나가 1이 되고 나머지는 0이 된다.
2. 2비트 비교기를 1)비트논리연산자 2)결합 연산자와 관계연산자 조합 으로 1),2)를 두가지 Verilog 를 이용하여 구현하고 결과를 확인하라.

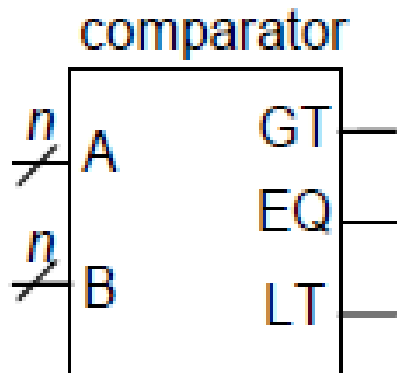


그림 1. n-비트 비교기의 블록도

입력 A, B	EQ	LT	GT
$A = B$	1	0	0
$A < B$	0	1	0
$A > B$	0	0	1

표 2. 비교기의 동작

Digital logic circuits

2015-1학기

Behavioral Level Modeling I



목차

3.1 Behavior Modeling

3.2 HDL 구문

3.3 순차 처리문

3.4 순차 할당

3.5 타이밍 제어

3.6 조건문

3.7 반복문

3.1 Behavioral Modeling개요

- 동작적 모델링(Behavioral Modeling)
 - 기존의 프로그래밍 언어 스타일과 유사
 - 입력상태에 대한 출력결과만을 고려한 기술방법(Functional or Algorithm Description)
 - 시스템이 내부적으로 어떠한 하드웨어적인 구조를 가지는지에 상관없고, 오로지 진리표 등으로 표현된 것을 기능적 또는 수학적인 알고리즘을 사용하여 시스템을 동작하는 기술

3.2 HDL 구문

- 병렬구문(Concurrent Statement)
 - 어떤 회로에 대한 동시적인 회로상태의 변화를 기술한 것.
 - 표현문장의 순서에 상관없이 똑같이 우선순위.
 - Simulation을 하면 첫 번째 줄의 쓰여진 동작 표현이나 마지막 줄에 쓰여진 동작표현이나 같은 시간에 simulation 될 수 있다.
 - 값의 전달은 signal을 통해서 전달된다.
- 순차구문(Sequential Statement)
 - 일반적인 로직의 순차적 상태를 기술할 수 있는 것
 - 프로그래밍 언어의 문장들과 같이 반드시 앞의 문장이 진행되어야 뒤에 문장이 진행된다.
 - 순차구문내의 모든 값들은 변수(variable)와 상수(constant)에 의하여 유지된다
 - 값의 전달은 variable에 의해서 전달된다.

3.3 순차처리문

- 순차 처리문
 - 논리회로는 입력으로부터 출력을 얻는 과정을 보통의 프로그램에서와 같이 순차적인 절차로 기술할 수 있으며 initial문 또는 always문과 같은 순차 처리문이 이러한 절차를 기술하기 위해서 사용된다.
 - initial문은 처음에 한 번만 수행되는 동작을, always문은 신호가 변할 때마다 반복하여 수행되는 동작을 기술한다.
 - 순차 처리문 내에서는 보통의 프로그래밍 언어에서와 같이 할당문, 조건문, 반복문등이 사용될 수 있다.
 - 순차 처리문은 다른 순차 처리문과 모듈/프리미티브 인스턴스 및 연속할당문들과는 병렬로 수행된다.

3.3 순차처리문

- Initial 문
 - 일반적으로 신호의 초기화, 모니터링을 위한 파형출력과 시뮬레이션 수행시에 한 번만 수행되어야 하는 과정을 기술하는 데 사용한다
 - 시뮬레이션 타임 0에서 시작하여 오직 1번만 실행된다
 - 여러 개의 initial문이 있으면 이들은 독립적으로 실행 한다.
 - 실행하는 문장이 여러 개가 있으면 begin과 end를 사용하여 블록으로 묶어야 하며 블록 내의 문장들은 순차적으로 실행된다.

i n i t i a l 문장; // 하나의 문장 포함

i n i t i a l b e g i n // 여러 개의 문장들을 포함
 문장;
 문장;
 ...
 end

3.3 순차처리문

- always문
 - always문은 반복조건이 만족할 때마다 문장을 반복하여 수행한다.
 - 시뮬레이션 타임 0에서 시작하여 시뮬레이션이 끝날 때 까지 반복
 - C 언어의 무한 루프와 비슷한 개념으로 \$finish 나 \$stop에 의해서만 정지한다.
 - 실행되는 문장이 여러 개이면 begin과 end를 사용하여 블록으로 설정해야 하며 이 문장들은 순차적으로 실행된다
 - Initial block 내부의 문장들은 순차적(sequential) 으로 동작한다

```
always 반복조건
    문장;
always 반복조건 begin // 여러 개의 문장들을 포함
    문장;
    문장;
    ...
end
```

3.3 순차처리문

- 하드웨어의 동작은 전원이 공급되었을 때부터 시작하여 지속적으로 반복되는 동작으로 볼 수 있다.
- 조합회로는 입력이 변화하면 출력이 변할 수 있으며, 순차회로는 대개 클럭(clock)이 특정 조건을 만족할 때에 출력이 변한다.
- 이러한 하드웨어의 동작은 출력이 변할 수 있는 조건이 발생하였을 때에 출력 값을 다시 계산하여 지정하는 동작이 반복되는 것으로 기술할 수 있다.
- always문은 이러한 동작을 나타내는 데 적합하다.
- always문의 반복조건은 조합회로나 순차회로의 출력 값이 갱신되는 시점을 기술하며 이벤트 발생, 시간 지연 등을 표현하는 타이밍 제어문으로 나타낸다.
- always 문은 구조적인 모델링이나 수식으로 동작을 기술하기 힘든 순차회로의 동작을 기술하는 데에 특히 유용하다.

3.3 순차처리문

initial과 always의 비교

종류	형식	동작	논리합성
initial	<code>initial begin</code> . . . <code>end</code>	(1회 동작) 시뮬레이션을 시작할 때에 한 번 수행	지원되지 않음
always	<code>always 반복조건 begin</code> . . . <code>end</code>	(반복 동작) 반복조건을 만족할 때마다 반복 수행	지원됨

논리 합성(logic synthesis)이란? RTL 형태로 추상적으로 서술된 코드를 논리 회로로 이루어진 하드웨어 형태로 변환하는 과정
Resister Transistor Level (RTL) : 저항과 레지스터간에 작용을 기술한 것, 주로 클럭에 동기하여 동작하는 동기식 회로(순차회로)에 사용된다. 조합회로 모델링도 가능하다.

3.4 순차 할당문

- 기호 '='는 보통의 프로그래밍 언어에서와 같이 오른쪽 수식의 결과를 왼쪽의 변수에 저장하는 할당문에 사용된다.
- 할당문은 always와 initial과 같은 순차 처리문 안에서 사용될 수 있으며 순차 처리문 안에 여러 개의 할당문이 있는 경우에 보통의 프로그래밍 언어에서와 같이 순서대로 수행하므로 순차 처리문에서 사용되는 할당문을 순차 할당(procedural assignment)문이라고 한다
- 순차 할당문에서 값이 저장되는 왼쪽 변수는 시뮬레이션 동안 값을 저장해야 하므로 reg나 integer와 같은 register형으로 선언되어야 한다.
- 변수가 실제 신호와 관계가 있으면 reg로 선언하고, 그렇지 않으면 integer로 선언하는 것이 일반적이다.

3.4 순차 할당문

- 블록킹문장(blocking statement)
 - 할당 연산자 '=' 사용 e.g.) $x=10$
 - 보통의 programming 언어와 같음
 - begin, end 블록 내의 할당문을 순서대로 수행
 - 할당문의 순서에 영향을 받을 수 있음
 - 시뮬레이션 시간값의 누적
- 논블록킹문장(non-blocking statement)
 - 할당연산자 '<=' 사용 e.g.) $x<=10$
 - begin, end 블록 내의 할당문의 수식들을 먼저 계산한 후에 LHS변수값을 갱신함
 - 할당문의 순서에 영향을 받지 않음
 - 병렬수행 모델링

3.5 타이밍 제어

- 타이밍 제어문은 always 문의 반복 조건 뿐 만 아니라 순차 처리문의 임의의 문장과 함께 사용하여 문장의 동작에 대한 타이밍을 제어할 수 있다.
- 타이밍 제어문을 만나면 수행을 멈추었다가 타이밍 제어문에서 주어진 조건을 만족하면 다시 수행을 재개한다
- 타이밍제어문의 종류
 - Delay-based timing control
 - Event-based timing control
 - Level-based timing control

3.5 타이밍 제어

- Delay-based timing control (지연제어문)
 - 지연 제어는 다음과 같은 형식으로 지연 시간을 기술한다.
 - # 시간
 - 지연제어문을 만나면 주어진 시간만큼 지연 후 문장을 수행한다.
 - 지연 제어문에는 다음과 같이 3가지로 나눈다.
 - 정규지연제어(regular delay control)
 - 내부할당지연제어(intra-assignment delay control)
 - 제로지연제어(zero Delay Control)

3.5 타이밍 제어

- Delay-based timing control (지연제어문)
- 정규지연제어(regular delay control)
 - 언제나 시뮬레이션시간을 누적한다.
 - e.g.) #10 y = 1; // 10 단위시간 후에 y = 1 실행

3.5 타이밍 제어

- Delay-based timing control (지연제어문)
 - 내부할당지연제어(intra-assignment delay control)
 - 할당 문장 우변에 delay를 규정
 - 논블록 문장과 사용시 begin, end 사이의 시뮬레이션 시간 누적없다.
 - 정규 지연은 문장 전체를 지연하지만 내부 할당 지연은 우변의 표현을 계산하고 좌변으로 대입을 #시간 만큼 지연한다.

```
reg x, y, z;  
  initial begin  
    x = 0; z = 0;  
    y = #5 x + z; // x+z값을 지연 없이 계산하고 5 ns 만큼 후에 y에 대입  
  end
```

3.5 타이밍 제어

- Delay-based timing control (지연제어문)
 - 제로지연제어(zero delay control)
 - #0의 symbol 사용한다.
 - 동일한 시뮬레이션 타임에 서로 다른 initial/always block의 기술문장의 실행 순서는 결정되지 않는다.
 - Zero delay는 동일한 시뮬레이션 타임에 실행되는 기술문장 중에서 제일 나중에 실행되는 것을 보장
 - 동일한 시뮬레이션 타임에 여러 개의 zero delay를 사용한 경우는 역시 non-deterministic
 - race condition을 막을 수 있다.

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

3.5 타이밍 제어

- Event-based timing control (이벤트 제어문)
 - 이벤트는 신호가 변화되는 것을 의미한다.
 - 이벤트 제어는 다음과 같이 괄호 안에 변화하기를 기다리는 신호를 기술한다.
 - @ (var) // var가 변할 때에 문장 수행
 - 이벤트 제어문을 만나면 괄호 안의 신호가 변할 때까지 기다리며 해당 신호가 변하면 이어지는 문장의 수행을 계속한다. 이벤트는 신호가 0에서 1로 변하거나 1에서 0으로 변할 때에 모두 발생한다.

```
always @(a)
```

```
x = a & b; // a가 변할 때마다 x값이 다시 계산됨
```

3.5 타이밍 제어

- Event-based timing control (이벤트 제어문)
 - 이벤트 제어문은 상승 하강 신호가 번갈아 가며 나타나는 에지 트리거 회로를 기술할 때 매우 유용하다.
 - 여러 개의 신호들 중에서 하나라도 변할 때에 이벤트가 발생하도록 하려면 다음과 같이 or를 사용하여 기술한다.
 - 이벤트 제어문의 or를 사용하여 나열된 신호들의 목록을 감지 목록(sensitivity list)이라고 부른다.
 - 이러한 감지목록을 사용하는 제어문을 이벤트 OR 제어(event OR control)라고 한다.
 - 조합회로는 입력이 변할 때에 출력에 영향을 받으므로 모든 입력을 감지목록에 포함시키고 always 문 안에 입출력 관계를 나타내는 수식을 기술하여 나타낼 수 있다

D-플립플롭 에지에 따라 출력이 변하는 것을 기술

```
@(var1 or var2 or var3) // var1, var2, var3 중 하나라도 변할 때에 수행
@(posedge clock or negedge reset)
// clock의 상승에지 또는 reset의 하강에지에서 수행
```

다음 Verilog 두 구문을 작성하여 TestBench를 통해서 입력값과 출력값의 파형을 확인해보라.

```
module ex_3;  
    reg x, y, z;  
    integer counter;  
    initial  
    begin  
        x = 1'b0; y = 1'b0; z =  
        1'b0;  
        counter = 0;  
        #10 x = 1'b1;  
        #20 y = 1'b1;  
        #40 z = 1'b1;  
        counter = counter + 1;  
    end  
    initial  
        #150 $stop;  
endmodule
```

```
module ex_4;  
    reg x, y, z;  
    integer counter;  
    initial  
    begin  
        x = 1'b0; y = 1'b0; z = 1'b0;  
        counter = 0;  
        x <= #10 1'b1;  
        y <= #25 1'b1;  
        z <= #40 1'b1;  
        counter <= counter + 1;  
    end  
    initial  
        #150 $stop;  
endmodule
```

3.6 조건문

- 순차처리문 내에서는 보통의 프로그래밍 언어에서처럼 조건문과 반복문을 사용할 수 있다.
- 조건문에는 if문과 case문 등이 있다.
- If 문
 - if, else 의 중첩 사용가능하며 각 조건을 만족할 때에 수행하거나 else 에 대한 문장이 여러 개이면 begin과 end를 사용한 블록문 안에 기술한다

if (조건식) 문장 // if문: 조건식이 참일 때 문장 수행

if (조건식) 문장1 // if-else문:

else 문장2 // 조건식이 참일 때 문장1, 거짓일 때 문장2 수행

if (조건식1) 문장1 // 중첩된 if-else문:

else if (조건식2) 문장2 // 조건에 따라서 여러 문장 중 하나를 수행

else if (조건식3) 문장3

...

else 기본문장

3.6 조건문

- 순차처리문 내에서는 보통의 프로그래밍 언어에서처럼 조건문과 반복문을 사용할 수 있다.
- 조건문에는 if문과 case문 등이 있다.
- If 문
 - if, else 의 중첩 사용가능하며 각 조건을 만족할 때에 수행하거나 else 에 대한 문장이 여러 개이면 begin과 end를 사용한 블록문 안에 기술한다

if (조건식) 문장 // if문: 조건식이 참일 때 문장 수행

if (조건식) 문장1 // if-else문:

else 문장2 // 조건식이 참일 때 문장1, 거짓일 때 문장2 수행

if (조건식1) 문장1 // 중첩된 if-else문:

else if (조건식2) 문장2 // 조건에 따라서 여러 문장 중 하나를 수행

else if (조건식3) 문장3

...

else 기본문장

3.6 조건문

- If 문을 이용한 4비트 2x1 멀티플렉서 예제

```
module mux2_1(sel, a, b, y);  
    input sel;  
    input [3:0] a, b;  
    output [3:0] y;  
    reg [3:0] y;  
    always @(sel or a or b) begin  
        if (sel == 0) y = a;  
        else y = b;  
    end  
endmodule
```

3.6 조건문

- case 문
 - case문은 수식의 값에 따라서 실행할 문장을 선택하는 다중 분기문이다.
- casez문
 - ‘z’값을 don’t care 로처리
- casex문
 - ‘x’와‘z’값을 don’t care 로처리

```
case (수식)
    값1: 문장1;
    값2: 문장2;
    값3: 문장3;
    ...
    default: 기본문장; // 일치하는 값이 없으면 기본문장 수행
endcase
```

3.6 조건문

- case 문 예제 : MUX 4 to 1

```
Module SEL(X0, X1, X2, X3, SEL ,Y) /* MUX 4 to 1 */
    input X0,X1,X2,X3;
    input[1:0] SEL;
    Output Y;
Function FUNC_SEL;
input X0,X1,X2,X3;
input[1:0] SEL;
    case(SEL) // 모든 조건을 나열하였으므로 "default"는 없어도 된다.
        0 : FUNC_SEL=X0; // SEL=0이면 X0을 function 으로 return
        1 : FUNC_SEL=X1; // SEL=1이면 X1을 function 으로 return
        2 : FUNC_SEL=X2; // SEL=2이면 X2을 function 으로 return
        3 : FUNC_SEL=X3; // SEL=3이면 X3을 function 으로 return
    endcase
endfunction
assign Y=FUNC_SEL(X0,X1,X2,X3,SEL);
endmodule
```

3.7 반복문

- Verilog에는 initial 또는 always문 내에서의 동작적 모델링에 사용할 수 있는 다음과 같은 4가지의 반복문이 있다.
 - for 문
 - while 문
 - repeat 문
 - forever 문
- 이들 중에서 for와 while 문은 C언어와 형식과 기능이 똑같다. 반복할 문장이 여러 개이면 { } 대신에 begin과 end로 묶어야 한다는 점만 차이가 있다

3.7 반복문

- repeat 문
 - repeat 문은 다음과 같은 형식을 가지며 수식의 값만큼 수행을 반복한다.

repeat (수식)
문장

- repeat 문에 제공되는 수는 반복문을 시작할 때에 상수로 주어지거나 계산이 되어 있어야 한다

3.7 반복문

- repeat 문

- repeat 문은 다음과 같은 형식을 가지며 수식의 값 만큼 수행을 반복한다.

repeat (수식)

문장

- repeat 문에 제공되는 수는 반복문을 시작할 때에 상수로 주어지거나 계산이 되어 있어야 한다

- forever 문

- forever 문은 다음과 같이 조건식이 없으며 수행을 무한히 반복한다.

forever

문장

3.7 반복문

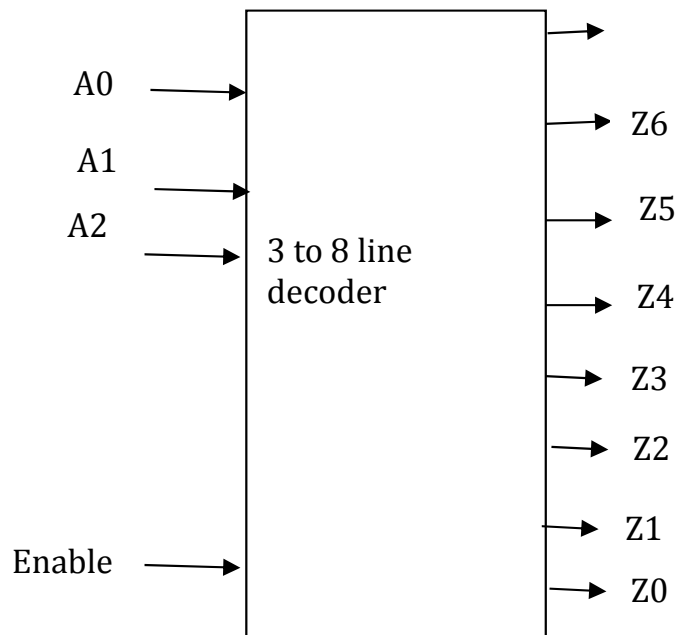
- repeat 문과 forever문 비교

```
always @(posedge clk) begin          initial begin
    x = a;                            forever @(posedge clk) begin
    y = b;                            x = a;
end                                    y = b;
                                    end
                                    end
```

forever문을 포함한 반복문의 수행은 disable 문에 의해서 종료될 수 있으며 이를 위해서는 수행을 종료할 반복문이 포함된 블록에 이름을 붙여야 한다

```
begin : break_block // break_block은 블록의 이름
i = 0;
forever begin
    if (i==a)
        disable break_block; //조건에 따라 블록 수행을 종료
    #1 i = i + 1;
end
end
```


- 다음과 그림과 같은 3-8 디코더 블록도와 상태표로 참조로 해서 Verilog 로 모듈을 정의하고 테스트 벤치를 통해 동작을 확인하라



	Input			Output							
enable	A1	A1	A0	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Digital logic circuits

2015-1학기

Behavior Modeling II
Task and Function



목차

4.1 태스크 와 함수

4.2 태스크

4.3 Function

4.4 Useful system task

4.5 Verilog HDL 실습

4.1 태스크 와 함수

- Verilog에는 공통적으로 반복하여 사용될 수 있는 코드는 함수(function)나 태스크(task)로 작성한 다음에 동작적 모델링에서 작성된 함수 또는 태스크를 호출하여 사용할 수 있다
- 태스크와 함수의 차이점

함수	태스크
하나 이상의 input 인수만을 가져야 한다.	input, output 또는 inout 인수를 가질 수 있다
타이밍 제어문을 포함 할 수 없어 항상 시뮬레이션 시간 0에 수행된다	타이밍 제어문을 사용하여 시간 지연을 포함할 수 있다
함수 이름을 통하여 결과를 전달한다	결과는 output 또는 inout 인수를 통해서 전달된다.
다른 함수를 호출해서 사용할 수 있지만 태스크는 사용할 수 없다.	다른 태스크나 함수를 사용할 수 있다.

4.1 태스크 와 함수

- 태스크와 함수의 공통점
 - 와이어(wire)를 가질 수 없다.
 - always 또는 initial 문장을 포함하는 것이 아니며 always 블록, initial 블록 또는 다른 태스크와 함수로부터 호출된다.

4.2 태스크

- 태스크는 모듈 내에서 키워드 task ~ endtask를 사용하여 다음과 같이 선언한다.

```
task gen_carry;  
    input a, b, c; // 입력 인수  
    output carry; // 출력 인수  
    begin  
        carry = a & b | a & c | b & c;  
    end  
endtask
```

- always나 initial문 내에서 다음과 같이 태스크를 호출하여 사용한다.

```
gen_carry(cout, a, b, cin);
```

- 태스크를 호출할 때에 사용하는 인수의 순서는 태스크를 선언할 때에 사용한 형식 인수의 순서와 같아야 한다.

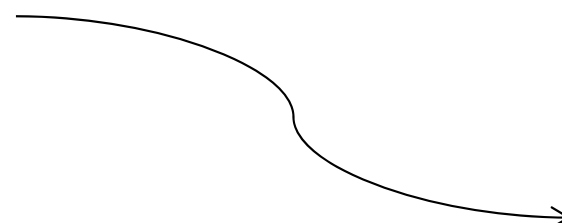
4.2 태스크

- 태스크를 이용한 4비트 리플캐리 가산기

```
module adder4_task(sum, cout, a, b, cin);
    output [3:0] sum;
    output cout;
    input [3:0] a, b;
    input cin;
    reg [4:0] carry;
    integer i;
    always @ (a or b or cin) begin
        carry[0] = cin;
        for(i = 0; i <= 3; i = i + 1)
            gen_carry(carry[i+1], a[i], b[i], carry[i]); // task 호출
    end
    assign cout = carry[4];
    assign sum = a ^ b ^ carry[3:0];
endmodule
```

task gen_carry; // task 정의

```
    output carry;
    input a, b, c;
    begin
        carry = a & b | a & c | b & c;
    end
endtask
```



endmodule

4.3 Function

- 함수는 모듈 내에서 키워드 `function ~ endfunction`을 사용하여 다음과 같이 선언한다.

```
function gencarry;  
input a, b, c; // 함수의 인수 - 입력만 허용  
begin  
gencarry = a & b | a & c | b & c; // 함수의 결과  
end  
endfunction
```

- 함수의 결과는 함수이름의 변수에 저장하여 돌려준다. 그리고 `always`나 `initial`문 내에서 다음과 같이 함수를 호출하여 사용한다

```
cout = gencarry(a, b, cin); // 함수의 결과를 cout에 저장
```

- 함수를 호출할 때에도 사용하는 인수의 순서는 함수를 선언할 때에 사용한 형식 인수의 순서와 같아야 한다

4.3 Function

- 함수를 이용한 4비트 리플캐리 가산기

```
module adder4_function(sum, cout, a, b, cin);
    output [3:0] sum;
    output cout;
    input [3:0] a, b;
    input cin;
    reg [4:0] carry;
    integer i;
    always @ (a or b or cin) begin
        carry[0] = cin;
        for(i = 0; i <= 3; i = i + 1) begin
            carry[i+1] = gencarry(a[i], b[i], carry[i]); // function 호출
        end
    end
    // function 정의
    function gencarry;
        input a, b, c;
        begin
            gencarry = a & b | a & c | b & c;
        end
    endfunction
endmodule
```

4.4 Useful System Task

- Verilog에서는 디버깅 이나 시뮬레이션등에서 루틴(특정한 작업을 실행하기 위한 일련의 명령) 연산을 수행하기 위해서 시스템 테스크 연산자를 제공한다.
- 사용법은 \$ 키워드 형태로 사용되며 아래와 같은 테스크 연산자를 제공한다.
 - Simulation
 - File I/O
 - Random number generation
 - Initializing Memory from File
 - Value Change Dump File

4.4 Useful System Task

- Simulation
 - 실행 제어
 - \$ stop
 - simulation을 중단하고, 대화모드(interactive mode)가 됨
 - \$finish
 - simulation의 완전한 종료

4.4 Useful System Task

- Simulation
 - 출력 제어
 - \$ display
 - 변수의 값, 문자열, 수식 등을 출력하기 위한 태스크이다.
 - 호출될 때만 실행되며, 신호의 변화가 없어도 출력한다.
 - \$display(d1, d2, ...);의 형태로 사용한다.
 - 전반적으로 C언어의 printf와 형태가 매우 유사하다.
 - \$monitor
 - 신호의 값이 변할 때마다 그 신호를 출력하기 위한 태스크
 - \$monitor(p1, p2,...); //변수 또는 파라미터에 지정된 신호의 값을 계속 모니터링,변수나 신호가 변할 때마다 모든 파라미터를 출력
 - \$monitoron; //시뮬레이션 하는 동안 모니터링을 할 수 있게 한다.
 - \$monitoroff; // 모니터링을 할 수 없게 한다
 - \$strobe
 - \$display와 거의 같지만, 같은시간에 실행될 모든 구문의 실행이 끝난 후에 \$strobe 명령을 실행하여 데이터가 출력되는 동기화 기술을 제공한다

4.4 Useful System Task

- Verilog로 coding을 하다보면 input data와 output data를 비교해야 할 경우가 생긴다.
 - 이때 사용하는 것이 바로 File I/O 이다.
 - C언어에서의 파일 처리 사용법과 매우 흡사하다.
 - 파일 모드인 w,r,a,b,+ 지원
- File I/O 관련 명령어
 - \$fopen 파일을 연다
 - 형식 \$fopen (“<파일명>”);
 - < file_handle > = \$fopen(“<파일명>”, 모드);
 - \$fclose 열린 파일을 닫는다.
 - 형식 : \$fclose(<file_handle>);
 - \$fdisplay, \$fmonitor, \$fwrite, \$fstrobe의 키워드로 file을 작성한다.
 - \$display, \$monitor, \$fwrite, \$fstrobe 와 비슷한 특성을 갖는다
 - 형식 \$명령어 < file_handle>

4.4 Useful System Task

- Verilog에서 파일 입출력
 - C언어 같은 동일한 함수들이 사용된다.
 - 문자 읽기: `$fgetc`
 - 문자열 읽기 : `$fgets`
 - `$fscanf`: 특정 내용을 파일에서 불러온다.
 - `$fread` : 바이너리 데이터 형태로 파일에서 불러온다.

4.4 Useful System Task

- File I/O 예제

```
integer file ; //integer는 32 비트 값
```

```
reg a, b, c;
```

```
initial begin
```

```
    file = $fopen("results.dat") ;
```

```
    a = b & c ;
```

```
    $fdisplay(file, "Result is: %b", a) ;
```

```
    $fclose(file) ;
```

```
end
```

4.4 Useful System Task

- Random Number Generation
 - 32bit의 random number 를 생성한다.
 - 사용법 : \$random(<seed>)
- Initializing Memory from File
 - \$readmemh, \$readmemb
 - \$readmemb(2진수 포맷), \$readmemhexa(16진수 포맷)지원
 - 형식 \$readmemb(“파일명”, 메모리명, 시작주소, 마지막 주소)
//파일명과 메모리명은 필수, 시작주소와 마지막 주소는 선택
 - \$writememh, \$writememb
 - 형식 (“filename”, memname, begin_addr, end_addr);
//begin_addr, end_addr은 생략 가능
 - 읽기 파일 안에는 공백, comment가 허용
 - 읽기 파일 안에 @hhhh... 형식으로 16진수 address 지정 가능하다.

4.4 Useful System Task

module File_Test; // 해당 코드를 실행하여 실제 데이터가 파일에 저장되는지 확인하자.

```
integer  handle, channels, index, rand;  
reg [7:0] memory [15:0];
```

```
initial begin
```

```
    handle = $fopen("mem.dat");
```

```
    channels = handle | 1;
```

```
    $display("Generating contents of file mem.dat");
```

```
    $fdisplay(channels, "@2");
```

```
        for(index = 0; index < 14; index = index + 1) begin
```

```
            rand = $random;
```

```
            $fdisplay(channels, "%b", rand[12:5]);
```

```
        end
```

```
    $fclose(handle);
```

```
    $readmemb("mem.dat", memory);
```

```
    $display("\n Contents of memory array");
```

```
        for(index = 0; index < 16; index = index + 1)
```

```
            $displayb(memory[index]);
```

```
    end
```

```
endmodule
```

4.4 Useful System Task

- Value Change Dump (VCD) File
 - VCD(Value Change Dump)는 simulation이 진행되는 동안의 simulation time, scope와 signal정의, signal value의 변화에 대한 정보를 담은 ASCII파일이다.
 - Design의 모든 신호의 변화를 저장할 수 있으며 후처리 툴을 이용하여 debug와 결과분석에 사용될 수 있다

4.4 Useful System Task

- Value Change Dump (VCD) File 관련 명령어
 - \$dumpvars
 - 덤프할 모듈 인스턴스와 모듈 인스턴스 신호를 선택
 - \$dumpfile
 - VCD 파일을 얻기 위한 태스크
 - \$dumpon
 - 덤프 과정의 시작
 - \$dumpoff
 - 덤프 과정의 멈춤
 - \$dumpall
 - 체크 포인트 생성
- \$dumpvars와 dumpfile은 시뮬레이션 초반에 지정하고, \$dumpon, dumpoff는 시뮬레이션 중에 덤프과정을 제어한다.

1. 4-bit 숫자의 팩토리얼을 계산하는 함수를 정의하여라. 해당 팩토리얼의 아웃풋은 32-bit 로 나와야 한다. 정의된 함수를 가지고 시뮬레이션을 통해서 해당 결과를 확인하라
2. 두 개 4비트 숫자인 a,b를 선택해서 다음 표와 같은 8가지 기능을 가지고 있는 ALU를 함수로 구현하라. 3비트 신호를 기반으로 함수가 동작해서 해당 결과는 5 비트로 계산되어야 한다. 해당 정의된 함수를 가지고 시뮬레이션을 통해서 해당 결과를 확인하라

Select Signal	Function Output
3'b000	a
3'b001	a + b
3'b010	a - b
3'b011	a / b
3'b100	a % b
3'b101	a << 1
3'b110	a >> 1
3'b111	a > b

3. 4-bit 숫자의 팩토리얼을 계산하는 태스크를 정의하여라. 해당 입력된 4비트 숫자의 팩토리얼의 아웃풋은 32-bit 로 나와야 한다. 해당 결과는 10 시간 만큼 지연 후에 출력이 되어야 한다.

Digital logic circuits

2015-1학기

Timing and Delay



목차

5.1 지연 경로 모델

5.2 경로 지연 모델링

5.3 타이밍 검사

5.4 Delay Back-Annotation

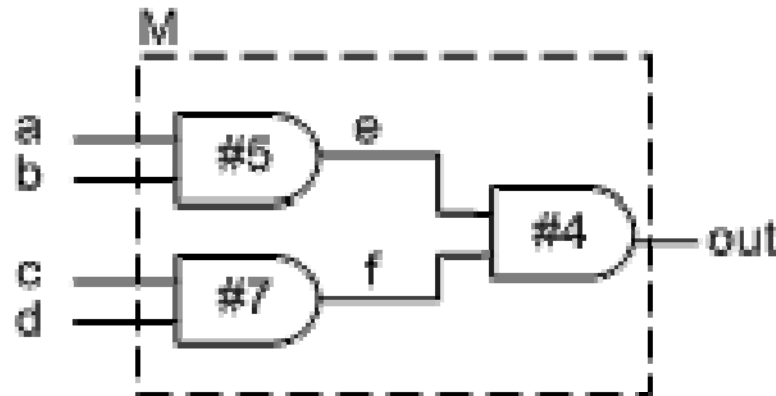
Verilog HDL ► 실습

5.1 지연 경로 모델

- 실제 하드웨어에서는 logic element들과 path들에 의해 delay들이 존재한다.
- 시뮬레이션에서는 구성된 회로가 원하는 시간 안에 결과가 나오는지 검증해야 한다.
 - 시간 지연 발생은 두 가지로 구성 요소 하드웨어 특성에 따른 회로 내부의 지연과 게이트 연결을 통해 데이터 전달에 경로에 따른 외부 지연이 생긴다.
- Verilog 시뮬레이션에서 사용되는 지연 모델은 다음 3가지가 있다.
 - Distributed delay
 - Lumped delay
 - pin-to-pin(path) delay

5.1 지연 경로 모델

- 분산 지연 (Distributed Delay)
 - 회로 안의 요소를 기반으로 따로 따로 정의할 수 있다.
 - 분산지연은 각 게이트들에게 주어진 지연값이나, assign문에 의한 지연값을 이용하여 모델링 한다.



```
// Distributed delay in gate-level
module M (out, a, b, c, d);
    output out;
    input a, b, c, d;
    wire e, f;
    and #5 a1(e, a, b);
    and #7 a2(f, c, d);
    and #4 a3(out, e, f);
```

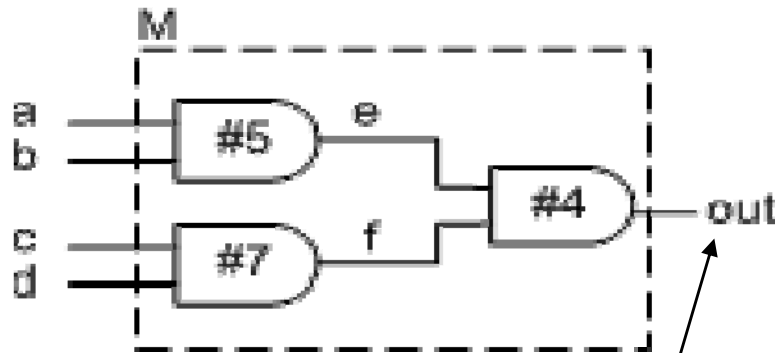
endmodule

```
// Distributed delay in data flow
module M (out, a, b, c, d);
    output out;
    input a, b, c, d;
    wire e, f;
    assign #5 e = a & b;
    assign #7 f = c & d;
    assign #4 out = e & f;
```

endmodule

5.1 지연 경로 모델

- 집중 지연(lumped Delay)
 - module 단위로 delay를 지정한다. module의 출력에 단일 delay로 지정하며 모든 path의 누적 delay를 나타낸다.

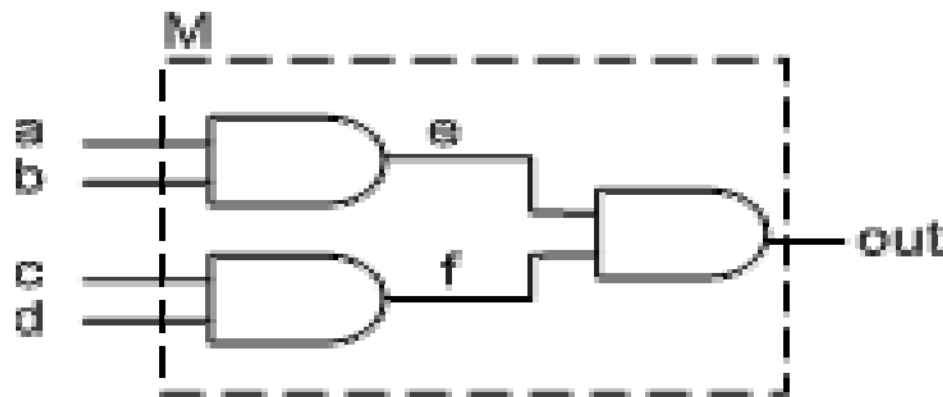


```
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
    and a1(e, a, b);
    and a2(f, c, d);
    and #11 a3(out, e, f);
endmodule
```

//입력에서 부터 출력까지 최대 지연 값을 계산(4+7)

5.1 지연 경로 모델

- Pin-to-Pin Delays (경로지연 – path delay)
 - input에서 부터 output까지의 모든 path에 대해서 개별적으로 delay를 지정한다.
 - 표준부품에 대해선 메이커가 제공하는 delay를 지정하거나 simulation을 통해 delay를 얻어 사용한다.
 - pin-to-pin delay가 세세하게 지정해줘야 하는 delay이긴 하지만 큰 사이즈의 design에서는 내부에 어떻게 구성되었는가를 일일이 고려하지 않고 입력과 출력의 관점에서 delay를 생각하기 때문에 오히려 쉬울 수 있다.



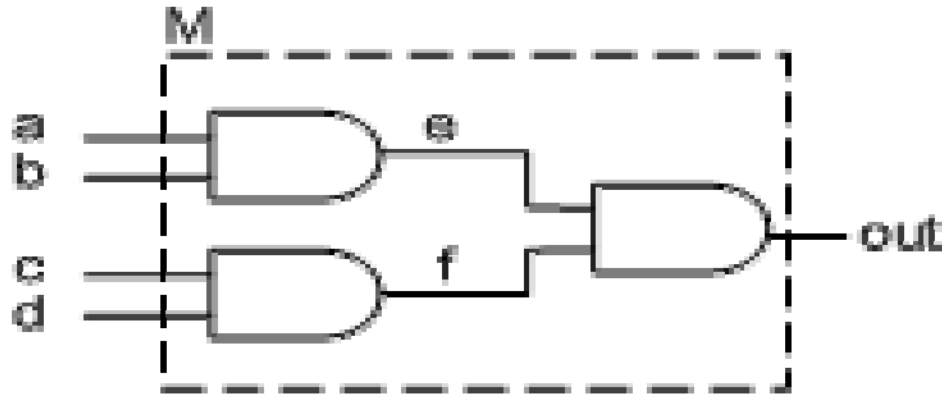
path a-e-out, delay = 9
path b-e-out, delay = 9
path c-f-out, delay = 11
path d-f-out, delay = 11

5.2 경로 지연 모델링

- Specify Blocks
 - 경로 지연(module의 source(input, inout)에서 destination(output, inout)까지의 지연)을 키워드 specify, endspecify 안에서 선언한 것을 의미한다.
 - 모듈에 존재하는 경로의 핀-대-핀 타이밍 지연을 지정하여 회로에서 확인할 타이밍을 결정한다.
 - specparam 상수를 정의한다.
 - specify block은 module안에서의 독립적인 별도의 block이다.
 - initial이나 always같은 다른 block안에 있을 수 없다.

5.2 경로 지연 모델링

- Specify Blocks을 이용한 경로 지연 예제



path a-e-out, delay = 9
path b-e-out, delay = 9
path c-f-out, delay = 11
path d-f-out, delay = 11

```
module M (out, a, b, c, d);  
  output out;  
  input a, b, c, d;  
  wire e, f;
```

```
  specify  
    (a => out) = 9;  
    (b => out) = 9;  
    (c => out) = 11;  
    (d => out) = 11;  
  endspecify
```

```
  and a1(e, a, b);  
  and a2(f, c, d);  
  and a3(out, e, f);
```

```
endmodule
```

5.2 경로 지연 모델링

- Specify Blocks의 구조
 - 병렬 연결 (parallel connection)
 - bit별 1대1의 path에 대한 delay설정하며 '='을 이용하여 할당한다.
 - 사용법 (<출발지> => <목적지>) = <지연값>;
// 출발지와 목적지가 벡터이면, 같은 비트수를 할당해야 한다.
 - 완전 연결(Full connection)
 - 모든 source와 destination을 연결하는 path의 delay를 하나의 값으로 한번에 지정하며 '*>'을 이용하여 할당한다.
 - 사용법 (<출발지> *> <목적지>) = <지연값>; //출발지의 각 비트는 목적지의 모든 비트와 연결하며 출발지와 목적지가 벡터이면, 같은 비트 수를 가질 필요는 없음

5.2 경로 지연 모델링

- Specify 블록 병렬 연결과 완전 연결 예 (page.9 회로)

```
// 병렬연결
```

```
// Single bit a, out
```

```
(a => out) = 9;
```

```
// Vector a[3:0], out[3:0]
```

```
(a => out) = 9;
```

```
// 위와 동일한 연결 방법
```

```
(a[0] => out[0]) = 9;
```

```
(a[1] => out[1]) = 9;
```

```
(a[2] => out[2]) = 9;
```

```
(a[3] => out[3]) = 9;
```

```
//완전 연결
```

```
module M (out, a, b, c, d);
```

```
    output out;
```

```
    input a, b, c, d;
```

```
    wire e, f;
```

```
    specify
```

```
        (a, b *> out) = 9;
```

```
        (c, d *> out) = 11;
```

```
    endspecify
```

```
    and a1(e, a, b);
```

```
    and a2(f, c, d);
```

```
    and a3(out, e, f);
```

```
endmodule
```

5.2 경로 지연 모델링

- Specparam Statements

- Specparam : specify block에서 사용하는 special parameter를 정의할 때 사용한다.
 - 일일이 delay 숫자를 써주는 것을 대신하여 parameter를 정의하고 specify block에서 사용한다.
 - specify parameter는 정의된 specify block안에서만 사용 가능하다.
 - e.g.) Page 9 회로도 정의

```
specify
```

```
    specparam d_to_q = 9;  
    specparam clk_to_q = 11;  
    (d => q) = d_to_q;  
    (clk => q) = clk_to_q;
```

```
endspecify
```


5.2 경로 지연 모델링

- 조건 경로 지연(Conditional Path Delays)
 - input 신호의 상태에 따라서 pin-to-pin delay가 달라질 수 있다.

```
module M (out, a, b, c, d);  
output out;  
input a, b, c, d;  
wire e, f;
```

- specify block안에 if구문을 이용해 input상태에 따라 다른 delay를 설정할 수 있으나 else 구문을 사용하는 것은 금지되어있다.

```
specify  
    if (a) (a ==> out) = 9;  
    if(~a) (a ==> out) = 10;  
//입력 신호 a의 상태에 따른 경로 지연 타이밍  
    if (b & c) (b ==> out) = 9; //신호에 따른 지연 조건문  
    if (~ (b & c)) (b ==> out) = 13;  
    if ({c, d} == 2'b01)  
        (c, d ==> out) = 11;  
    if ({c, d} != 2'b01)  
        (c, d ==> out) = 13;  
endspecify
```

5.2 경로 지연 모델링

- 상승, 하강, 턴오프 지연
 - Pin-to-pin delay는 신호의 변동에 따라 rise, fall, turn-off delay로 세분화하여 지정해 줄 수 있다
 - Rising
 - $0 \rightarrow 1$, $x \rightarrow 1$, $z \rightarrow 1$
 - Falling
 - $1 \rightarrow 0$, $x \rightarrow 0$, $z \rightarrow 0$
 - Turnoff
 - $0 \rightarrow z$, $1 \rightarrow z$, $x \rightarrow z$

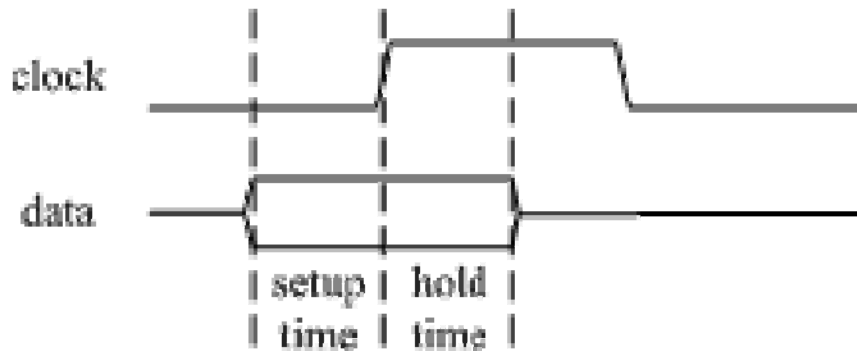
5.2 경로 지연 모델링

- 최소,전형적,최대 지연 (Min, typ and Max)
 - 경로 지연에서는 최소,전형적,최대 지연값을 지정할 수 있다.
 - min delays :최소 지연값
 - typ delays : 전형적 지연값
 - max delays : 최대 지연값

```
specparam t_rise = 8: 9 :10, t_fall = 12 : 13 : 14, t_turnoff = 10 : 11 : 12;  
(clk => q) = (t_rise, t_fall, t_turnoff);
```

5.3 타이밍 검사

- \$setup, \$hold, \$width : 가장 일반적이고 많이 사용하는 timing checking system task로 언제나 specify block안에서만 사용 가능하다.
 - \$setup and \$hold
 - 순차회로에서 clock의 전후로 클럭이 상승되기 전에 데이터가 도착하고 상승 후에 data의 최소 유효 시간을 의미한다.



`$setup(data_event, reference_event, limit);`

data_event : monitor하는 신호

reference_event : *data_event*의 기준이 되는 신호

limit : setup time에 필요한 최소 시간

– $(T_{reference_event} - T_{data_event}) < limit$ 이면 오류를 보고한다.

`$hold(reference_event, data_event, limit);`

reference_event : *data_event*의 기준이 되는 신호

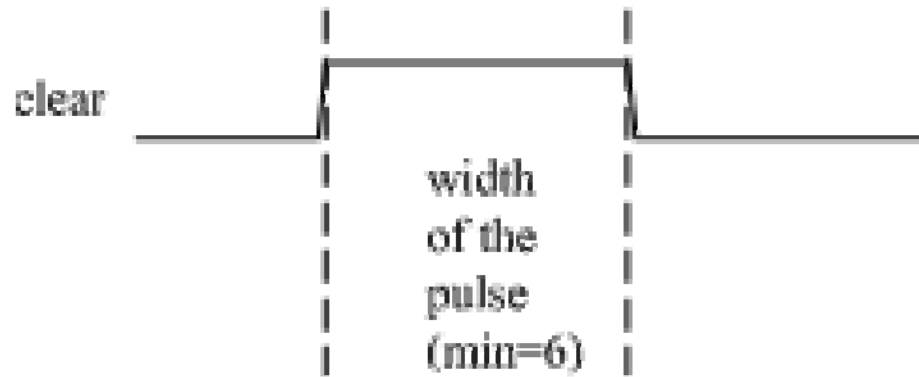
data_event : monitor하는 신호

limit : hold time에 필요한 최소 시간

– $(T_{data_even} - T_{reference_event}) < limit$ 이면 오류를 보고한다.

5.3 타이밍 검사

- \$width
 - 펄스의 폭을 check해야 할 때 사용한다.



```
specify
    $width(posedge clear, 6);
endspecify
```

`$width(reference_event, limit);`

reference_event : edge-triggered event

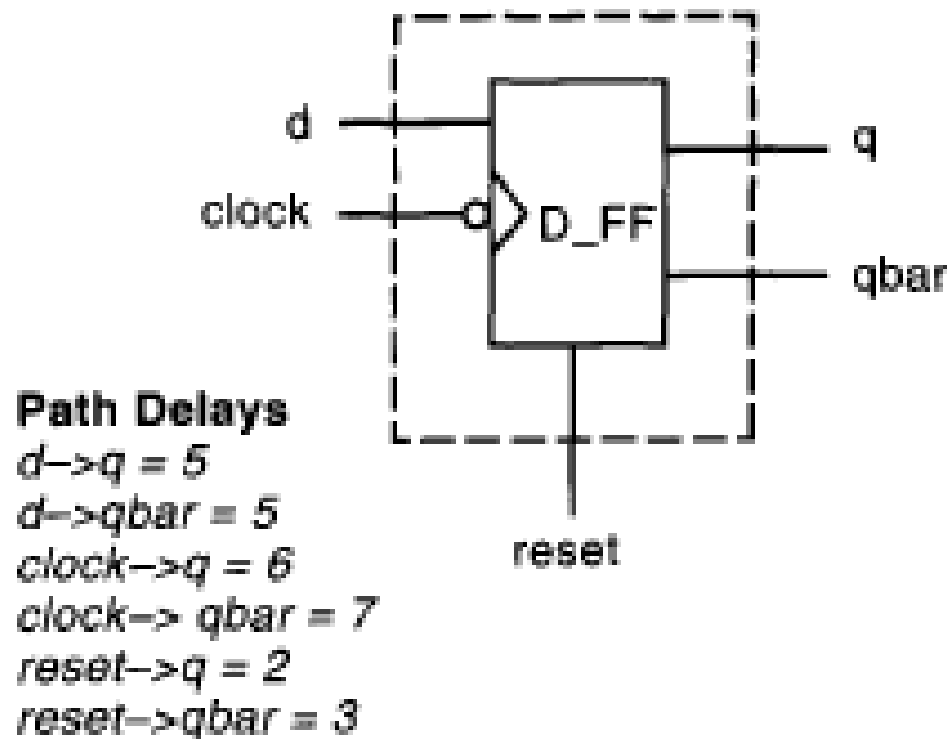
limit : 펄스 폭의 최소 시간

- *data_event*는 따로 명시하지 않는다. *reference_event*가 일어나고 난 후 다음에 오는 반대방향의 edge가 *data_event*가 된다.
- $(T_{data_event} - T_{reference_event}) < limit$ 이면 오류를 보고한다.

5.4 Delay Back-Annotation

- Delay Back-Annotation이란?
 - 구성 회로 지연을 고려하기 위해 실제 로직 및 wire 지연등의 모든 정보를 통한 예측된 지연값을 시뮬레이션하여 구성 회로가 원하는 시간안에 동작하도록 최적화를 반복하는 작업
- Delay Back-Annotation의 순서
 - 1) RTL로 기술하고 functional simulation을 실행
 - 2) RTL을 gate-level의 netlist로 변환(Logic Synthesis)
 - 3) IC공정에 관한 정보와 delay calculator등을 이용한 chip의 prelayout data를 가지고 timing simulation을 실행
 - 4) P&R(Place & Route) tool로 layout 작성. 이 layout의 R, C를 계산하여 postlayout delay값을 추출((R은 도체의 저항, C는 부도체의 커패시턴스)
 - 5) 이렇게 얻어진 delay값으로 gate-level netlist에 사용된 delay 추정값을 수정. timing simulation을 재실행. spec.을 만족하는지 확인
 - 6) timing spec을 만족하면 RTL로 돌아가 최적화를 진행하고 2단계부터 5단계의 과정을 반복.

1. 다음 그림은 negative edge-triggered with the asynchronous reset D-flipflop 이다. verilog을 통해서 해당 모델을 정의하여. I/O 포트와 경로 지연을 정의하고 , 경로 지연은 specify 블록내 병렬 연결로 정의하라.



Digital logic circuits

2015-1학기

User Defined Primitive



목차

- 6.1 UDP
- 6.2 UDP 기본 규칙
- 6.3 UDP 문법
- 6.4 UDP 예시
- 6.5 Combinational UDPs
- 6.5 Sequential UDPs
- 6.5 Level-Sensitive UDP
- 6.5 Edge-sensitive Sequential UDP (1)
- 6.5 Edge-sensitive Sequential UDP (2)
- 6.5 Sequential UDP Initialization
- 6.6 UDP 테이블 약식기호
- 6.7 UDP 설계에 대한 지침
- Verilog HDL ► 실습

6.1 UDP

- and, or, not, xor, etc.
 - Verilog에서 제공하는 System Primitives들이다.
- Verilog에서 제공하는 System Primitives이외에 사용자가 자신의 회로를 정의하여 사용할 수 있다.
 - 사용자가 정의한 회로를 User Defined Primitives (UDPs)라고 한다.
 - Truth Tables을 이용해서 다양한 UDP를 만들 수 있따.
 - UDP 는 module/endmodule 대신에 primitive/endprimitive로 정의한다.
- 다음 두가지 형태로 UDP를 정의할 수 있다.
 - 조합형(Combinational)
 - 순차형(Sequential)

6.2 UDP 기본 규칙

- 입력은 input로 크기는 1비트이며, 여러 개의 입력이 선언가능하다.
- 출력은 output, reg(순차 UDP에서만 사용)로 선언하며 그 크기는 1비트로 오직 하나의 출력만을 가져야 한다.
- 조합형 UDP가 가질 수 있는 최대 입력 포트는 10개이고 순차형 UDP가 가질 수 있는 최대 입력 포트는 내부 상태 동작을 기억하기 포트를 제외한 9개이다.
- 각각의 UDP들이 출력 가능한 값은 1,0,x이다
 - Table 이라고 선언된 부분에 진리표(True table)형식으로 사용자가 정의한 회로(UDP)의 상태를 정의한다.
 - 임피던스 z 를 가질 수 없으며 입력되는 z 는 x 로 변환하여 사용한다.
 - 상태 테이블에서 Don't care 는 ? 로 표시(? 는 1, 0, x)
 - 상태 테이블에서 상태의 유지는 - 로 표시
- 모든 UDP 포트들은 스칼라 값을 가진다.
- UDP는 양방향 포트가 불가능하다.

6.3 UDP 문법

```
primitive UDP_name {  
    <output_terminal_name>,<input_terminal_names>); //출력은  
    오직 1개  
    //terminal declarations  
    output <output_terminal_name>;  
    input <input_terminal_names>;  
    reg <output_terminal_name>; //오직 순차 UDP 일 때  
    initial <output_terminal_name> = <value>;  
    /*동작 구현(combinational과 sequential 둘다) 은 table이라고  
    선언하고 그안에서 구현한다. 구현이 완료된 후에 ' endtable ' 로  
    마무리*/  
    <table entries>  
    endtable  
  
end primitive
```

6.4 UDP 예시

```
// This code shows how UDP body looks like
```

```
primitive udp_body (
```

```
  a, // Port a
```

```
  b, // Port b
```

```
  c // Port c
```

```
);
```

```
  output a;
```

```
  input b,c;
```

```
// UDP function code here
```

```
// A = B | C;
```

```
table
```

```
  // B C : A
```

```
    ? 1 : 1;
```

```
    1 ? : 1;
```

```
    0 0 : 0;
```

```
endtable
```

```
endprimitive
```

6.5 Combinational UDPs

- 현재 출력 값에 관계없이 입력 값의 상태 테이블내의 조합에 의해서 출력 값이 결정된다.

```
primitive multiplexer(mux,control,dataA,dataB ) ;  
  output mux ;  
  input control, dataA, dataB ;  
  table  
    // control dataA dataB mux  
    0 1 ? : 1 ; // ? = 0,1,x  
    0 0 ? :0 ;  
    1 ? 1 :1 ;  
    1 ? 0 :0 ;  
    x 0 0 :0 ;  
    x 1 1 :1 ;  
  endtable  
endprimitive
```

6.5 Sequential UDPs

- 상태 테이블의 현재의 상태와 입력값에 의해서 다음상태 및 출력값이 결정한다.
 - 순차 UDP 의 출력은 항상 reg 로 선언한다.
 - 상태테이블은 입력, 현재상태, 다음상태로 표현
 - 상태테이블의 다음상태는 출력 reg 의 값이다.
 - 상태 테이블의 다음상태는 입력값과 상태테이블의 현재상태에 의해 결정된다.
- 준위-구동(Level-Sensitive) 순차형 UDP 와 모서리-구동(Edge-Sensitive) 순차형 UDP 로 구분한다.

6.5 Level-Sensitive UDP

- 아웃풋을 reg로 선언하는 것 이외에 조합 UDP와 동작은 동일하다.
- 현재 상태에 대한 필드가 추가되었다.
 - 입력과 출력을 콜론(:)으로 구분한다.
 - input field(s): current state field : output field

```
primitive latch(q, clock, data) ;                // UDP for a latch
output q; reg q ;
input clock, data;
table
    // clock data q q+
    0 1 :?: 1 ;
    0 0 :?: 0 ;
    1 ? : ? : - ; // - = no change
endtable
endprimitive
```


6.5 Edge-sensitive Sequential UDP (1)

- 모서리 변화(Edge-triggering)과 입력 값에 따라서 상태가 변화한다.
 - Edge triggering : positive edge(1) 또는 negative edge(0)에서 다른 상태 천이를 나타낸다.

모서리 변화	표현 의미
(01)	$0 \rightarrow 1$
(10)	$1 \rightarrow 0$
(1X)	$1 \rightarrow X$
(0?)	$0 \rightarrow 0, 1, X$
(??)	$0, 1, X \rightarrow 0, 1, X$

- 상태 테이블에서 한 개의 입력조건당 두 개의 모서리 변화는 불가능하다.
(01)(01)0 : 0 : 1 //불가능한 표현

6.5 Edge-sensitive Sequential UDP (2)

UDP for an edge-sensitive D-type flip-flop

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;
table
    // obtain output on rising edge of clock
    // clock data q q+
    (01) 0 : ? : 0 ;
    (01) 1 : ? : 1 ;
    (0?) 1 : 1 : 1 ;
    (0?) 0 : 0 : 0 ;
    // ignore negative edge of clock
    (?0) ? : ? : - ;
    // ignore data changes on steady clock
    ? (??) : ? : - ;
endtable
endprimitive
```

6.5 Sequential UDP Initialization

- 순차 UDP에서 initial을 사용해서 초기값을 지정해 줄 수 있다.

```
primitive udp_sequential_initial(q, clk, d);
```

```
    output q;
```

```
    input clk, d;
```

```
    reg q;
```

```
    initial begin
```

```
        q = 0;
```

```
    end
```

```
table
```

```
    // obtain output on rising edge of clk
```

```
    // clk      d      q      q+
```

```
    (01)      0 : ? : 0 ;
```

```
    (01)      1 : ? : 1 ;
```

```
    (0?)      1 : 1 : 1 ;
```

```
    (0?)      0 : 0 : 0 ;
```

```
    // ignore negative edge of clk
```

```
    (?0)      ? : ? : - ;
```

```
    // ignore d changes on steady clk
```

```
    ?      (??) : ? : - ;
```

```
endtable
```

```
endprimitive
```

6.6 UDP 테이블 약식기호

- UDP 테이블에 사용되는 ?와 같이 사용할 수 있는 약식 기호가 있다.

기호	의미	설 명
b	0 or 1	?와 같으나 x 값이 제외됨
r	(01)	입력에서 상승엣지
f	(10)	입력에서 하강엣지
p	(01) or (0x) or (x1) or (1z) or (z1)	unknown을 포함한 상승엣지들
n	(10) or (1x) or (x0) or (0z) or (z0)	unknown을 포함한 하강엣지들
*	(??)	모든 상태변이

6.7 UDP 설계에 대한 지침

- UDP 는 단지 기능적인 모델이다.
- 입력단자의 개수가 증가할 수록 상태 테이블의 조건은 지수승으로 증가한다.
 - 많은 입력단자의 개수를 쓰는 것은 바람직하지 않다.
 - UDP의 상태 테이블은 항상 모든 조건을 고려해서 완벽하게 기술해야 한다.
- 준위-구동(Level-Sensitive) 순차형 UDP 와 모서리-구동(Edge-Sensitive) 순차형 UDP를 혼합해서 사용할 수 있다.
 - e.g) 초기값과 클리어 신호를 가지고 있는 비동기 edge-triggered JK flip-flop

1. 아래 아웃풋 조건을 가지는 and-or 게이트를 UDP로 작성하고 결과를 확인해라

$$\text{out} = (\text{a1} \ \& \ \text{a2} \ \& \ \text{a3}) \ | \ (\text{b1} \ \& \ \text{b2})$$

```
답)// Description of an AND-OR gate.  
// out = (a1 & a2 & a3) | (b1 & b2).  
primitive and_or(out, a1,a2,a3, b1,b2);  
output out;  
input a1,a2,a3, b1,b2;  
table  
// a b : out ;  
111 ?? : 1 ;  
??? 11 : 1 ;  
0?? 0? : 0 ;  
0?? ?0 : 0 ;  
?0? 0? : 0 ;  
?0? ?0 : 0 ;  
??0 0? : 0 ;  
??0 ?0 : 0 ;  
endtable  
endprimitive
```

2. 두개의 입력을 가지는 SR flip-flop을 UDP로 작성하고 결과를 확인해라. 단 아웃풋의 초기값은 1'b1이다.

```
답) primitive srff_udp (q,s,r);  
output q;  
input s,r;  
  
reg q;  
  
initial q = 1'b1;  
  
table  
  // s r q q+  
  1 0 : ? : 1 ;  
  f 0 : 1 : - ;  
  0 r : ? : 0 ;  
  0 f : 0 : - ;  
  1 1 : ? : 0 ;  
endtable  
  
endprimitive
```

Digital logic circuits

2015-1학기

논리회로 모델링 실습



7.1 조합논리회로의 형태와 설계에 사용되는 Verilog 구문

조합논리회로의 형태

- 논리 게이트
- Multiplexer**
- Encoder**
- Decoder**
- Random Logic**
- Adder**
- Subtractor**
- ALU**
- Lookup Table**
- Comparator**

조합논리회로 설계에 사용되는 **Verilog** 구문

- 게이트 프리미티브
- 연속 할당문 (**assign** 문)
- 행위수준 모델링 (**if** 문, **case** 문, **for** 문)
- 함수 및 **task** (시간 또는 **event** 제어를 갖지 못한다)
- 모듈 인스턴스

논리합성이 지원되지 않는
Verilog 구문

- initial** 문
- 스위치 프리미티브 (**cmos**, **nmos**, **tran** 등)
- wait**, **event**, 지연 등 타이밍 제어 구문
- force-release**, **fork-join**
- 시스템 **task** (**\$finish**, **\$time** 등)
- forever** (**while**, **repeat**는 경우에 따라)의 반복문

7.2 조합논리회로 모델링

□ 조합논리회로 모델링 시 유의사항

❖ always 구문

- 감지신호 목록 (**sensitivity list**)에 회로 (즉, **always** 블록으로 모델링되는 회로)의 **입력신호들이 빠짐없이 모두 포함되어야 함**
- 그렇지 않은 경우 ; 합성 전과 합성 후의 시뮬레이션 결과가 다를 수 있음

❖ if 조건문과 case 문

- **모든 입력 조건들에 대한 출력 값이 명시적으로 지정되어야 함**
- 그렇지 않은 경우 ; 래치가 생성되어 순차논리회로가 될 수 있음

❖ 논리 최적화가 용이한 구조와 표현을 사용

- 최소의 게이트 수와 최소 지연경로를 갖는 회로가 합성되도록 해야 함

❖ 소스코드가 간결해지도록 모델링

- 소스코드의 가독성 (**readability**)을 좋게 하여 오류 발생 가능성을 줄여 주고, 디버깅을 용이하게 하여 설계 생산성을 높여 줌

7.3 기본 논리게이트 모델링 (I)

□ 4입력 NAND 게이트

비트 연산자, 축약연산자, 게이트 프리미티브

if 조건문

```
module nand4_if(a, y);  
    input  [3:0] a;  
    output          y;  
    reg          y;  
    always @(a) begin  
        if(a == 4'b1111) y = 1'b0;  
        else            y = 1'b1;  
    end  
endmodule
```

7.3 기본 논리게이트 모델링(II)

□ 부울함수의 모델링

$$y = \overline{(a+b) \times (c+d) \times e}$$

비트 연산자

```
module comb_gate(a, b, c, d, e, y);  
    input  a, b, c, d, e;  
    output y;  
  
    assign y = ~((a | b) & (c | d) & e);  
  
endmodule
```

7.4 4비트 4:1 멀티플렉서

❖ if 조건문, case 문, 조건연산자 등을 이용하여 모델링이 가능하다

1. if 조건문

```
module mux41_if(sel, a, b, c, d, y);
    input  [1:0] sel;
    input  [3:0] a, b, c, d;
    output [3:0] y;
    reg     [3:0] y;

    always @(*) begin
        if      (sel == 2'b00) y = a;
        else if(sel == 2'b01) y = b;
        else if(sel == 2'b10) y = c;
        else if(sel == 2'b11) y = d;
        else
            y = 4'bx;
    end
endmodule
```

2. case 문

```
module mux41_case(sel, a, b, c, d,y);

    always @(*) begin
        case(sel)
            0 : y = a;
            1 : y = b;
            2 : y = c;
            3 : y = d;
        default : y = 4'bx;
        endcase

    end
endmodule
```

7.5 2진 인코더

□ n to m 2진 인코더

- ❖ n-비트의 입력을 m-비트의 출력으로 변환(단, $n=2^m$)
- ❖ if 조건문, case 문, for 반복문 등 여러 가지 방법으로 모델링
 - 반복문 ; 입력의 비트 수가 큰 경우 또는 입/력 비트 수를 파라미터화하여 모델링하는 경우에 유용
- ❖ 진리표에 나열된 입력조건 이외의 경우에는 출력에 **don't care** 값을 할당하여 최소화된 회로가 합성되도록 함

입력 a[3:0]	출력 y[1:0]
0001	00
0010	01
0100	10
1000	11

7.6 2진 인코더

for 반복문

```
module enc_4to2_for(a, y);
    input  [3:0] a;
    output [1:0] y;
    reg     [1:0] y;
    reg     [3:0] temp;
    integer      n;

    always @(a) begin
        temp = 4'b0001;
        y = 2'bx;
        for(n = 0; n < 4; n = n + 1)
            begin
                if(a == temp)
                    y = n;
                temp = temp << 1;
            end
        end
    end
endmodule
```

7.7 우선순위 인코더

□ 4:2 우선순위 인코더

입력	출력	
a[3:0]	y[1:0]	valid
1xxx	11	1
01xx	10	1
001x	01	1
0001	00	1
0000	xx	0

// casex 문 사용

```
module pri_enc_4to2_case(a, valid, y);
    input  [3:0] a;
    output          valid;
    output [1:0] y;
    reg          valid;
    reg  [1:0] y;

    always @(a) begin
        valid = 1;
        casex(a)
            4'b1xxx : y = 3;
            4'b01xx : y = 2;
            4'b001x : y = 1;
            4'b0001 : y = 0;
            default : begin
                valid = 0;
                y = 2'bx;
            end
        endcase
    end
endmodule
```


□ 4:2 우선순위 인코더

다음 조건을 가지는 4:2 우선 순위 인코더를 for 문을 이용해서 작성해라

1. 14 페이지에 있는 cazex를 이용한 구문과 결과가 동일해야 한다.
2. 테스트 벤치를 이용해서 20ns 마다 입력 파형과 출력 파형의 변화를 확인하라

입력	출력	
a[3:0]	y[1:0]	valid
1xxx	11	1
01xx	10	1
001x	01	1
0001	00	1
0000	xx	0

7.8 디코더

□ m to n 2진 디코더

- ❖ m-비트의 입력을 n-비트의 출력으로 변환(단, $n=2^m$)
- ❖ n:m 2진 인코더로 인코딩된 데이터를 복원시킴
- ❖ if 조건문, case 문, for 반복문 등 여러 가지 방법으로 모델링

2:4 2진 디코더의 진리표

입력 a[1:0] 출력 y[3:0]

00	0001
01	0010
10	0100
11	1000

```
module dec_2to4_if(a, y);  
    input  [1:0] a;  
    output [3:0] y;  
    reg     [3:0] y;  
    always @(a) begin  
        if(a == 0)      y = 4'b0001;  
  
        else if(a ==1) y = 4'b0010;  
        else if(a ==2) y = 4'b0100;  
        else            y = 4'b1000;  
    end  
endmodule
```

□ **Active-high enable** 신호를 갖는다음 표1의 데이터 값을 가지는 **3 : 6 디코더**를 다음의 방법들로 모델링하고, 시뮬레이션을 통해 검증 하여라.

❖ **if** 조건문을 사용하는 방법

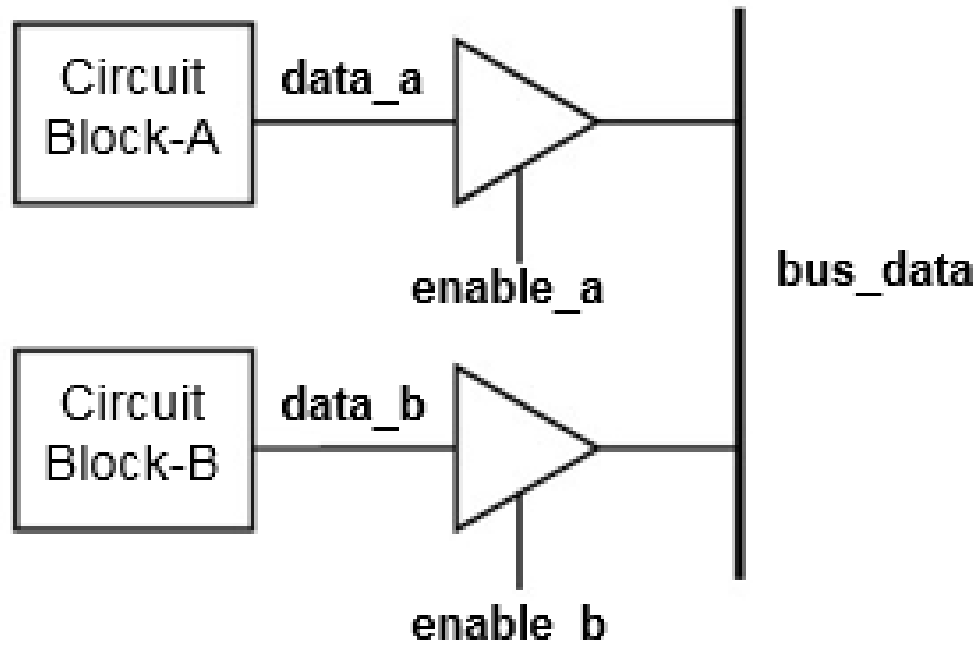
❖ **enable** 신호와 입력 **a**를 결합 연산자 **{ }**로 묶어 **case** 문의 조건으로 사용하는 방법

표 1 3:6 디코더

enable	입력 a[2:0]	출력 y[5:0]
0	xxx	000000
	000	000001
	001	000010
	010	000100
	011	001000
1	100	010000
	101	100000
	110	000000
	111	000000

7.9 3상태 (tri-state) 버스

- ❖ 회로의 여러 부분에서 생성된 신호들을 공통 버스를 통해 회로의 다른 부분으로 전송하기 위해 사용 된다.
 - 제어신호는 데이터를 버스에 보내거나 또는 데이터 소스를 버스로부터 격리시켜 **High impedance** 상태(전기적으로 절연된 상태)가 된다.
- ❖ 조건 연산자 또는 게이트 프리미티브를 사용하여 모델링한다.



7.9 Tri-state 버스

4비트 3상태 버스

```
module tri state ( q,  A,  B,  controll,  control2,  enable,  clk  );
    input [3:0] A,  B;
    input controll,  control2,  enable,  clk;
    output [3:0] q;
    tri [3:0]

    tri bus;
    tribuf driverA[3:0] (.out(tri bus),  .in(A),  .control(controll));
    tribuf driverB[3:0] (.out(tri bus),  .in(B),  .control(control2));
    dff mydff[3:0] (.q (q),  .d (tri bus),  .en(enable),  .clk(clk));

endmodule  // tri state
```

7.10 순차회로 모델링

□ 순차회로

- ❖ 현재의 입력, 과거의 입력, 회로에 기억된 상태값에 의해 출력이 결정
- ❖ 과거의 입력, 현재의 상태값을 저장하는 기억소자(래치 또는 플립플롭)와 조합논리회로로 구성
- ❖ 데이터 레지스터, 시프트 레지스터, 카운터(counter), 직렬/병렬 변환기, 유한상태머신(Finite State Machine; FSM), 주파수 분주기, 펄스 발생기 등
 - 클록신호에 의해 동작되는 래치 또는 플립플롭을 포함

□ 래치와 플립플롭

- ❖ 래치 : 클록신호의 레벨(즉, 0 또는 1)에 따라 동작하는 저장소자
- ❖ 플립플롭 : 클록신호의 상승 또는 하강에지에 동기되어 동작하는 저장소자
- ❖ **always** 구문 내부에 **if** 조건문을 이용하여 모델링

□ 순차회로의 모델링

- ❖ **always** 블록을 이용한 행위수준 모델링, 게이트 프리미티브 및 하위모듈 인스턴스, 연속 할당문 등 다양한 Verilog 구문들이 사용됨
- ❖ 할당문의 형태 (nonblocking 또는 blocking) 에 따라 회로의 동작과 구조가 달라짐

7.10 순차회로 모델링

- Flip-Flop (FF)는 Synchronous/asynchronous reset/set으로 동작을 한다.
 - Synchronous Sequential Logic Circuit는 회로의 상태가 정해진 순간의 입력 값에 따라서만 변화는 Logic Circuit
 - Asynchronous Sequential Logic Circuit는 회로의 상태가 어느 순간에서나 입력의 변화에 따라 변하는 Logic Circuit

Synchronous

```
module dff (clk, s, r, d, q);  
  input clk, s, r, d;  
  output q;  
  reg q;  
  always @(posedge clk)  
    if (r) q = 1'b0;  
    else if (s) q = 1'b1;  
    else q = d;  
  
endmodule
```

Asynchronous

```
module dff (clk, s, r, d, q);  
  input clk, s, r, d;  
  output q;  
  reg q;  
  always @(posedge r)  
    q = 1'b0;  
  always @(posedge s)  
    q = 1'b1;  
  always @(posedge clk)  
    q = d;  
  
endmodule
```

7.10 순차회로 모델링

Flip-flop을 Verilog로 잘못 작성한 예

Always 구문을 사용할 때는 어떤 클럭 엣지에 따라 변경을 할 지 명시하여야 하며 클럭 변경에 따른 데이터 변경될 시간을 기다려야 할 필요가 있다.

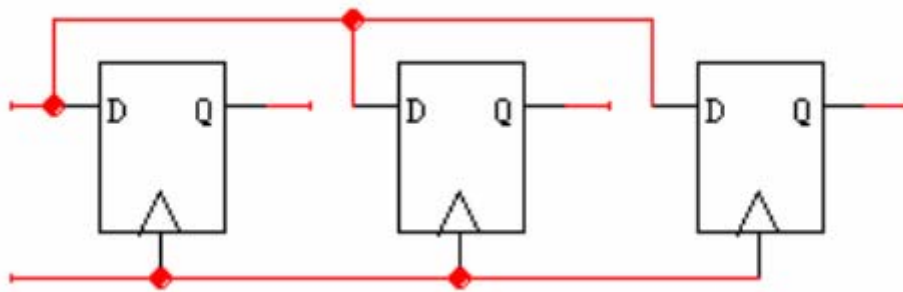
```
module dff (clk, d, q);  
  input clk, d;  
  output q;  
  reg q;  
  always @(clk)  
    q = d;  
endmodule
```

잘못된 구문 ; Q값은 엣지에 상관 없이 언제나 변한다,
posedge clk 또는 **negedge clk** 중 하나로 명시하거나
엣지마다 값을 변하시키기 위해서는 **clk**이외의 변수를
하나 더 선언하고 구문에서 지정한다.
e.g) **always @(posedge clk or negedge clk_nt)**

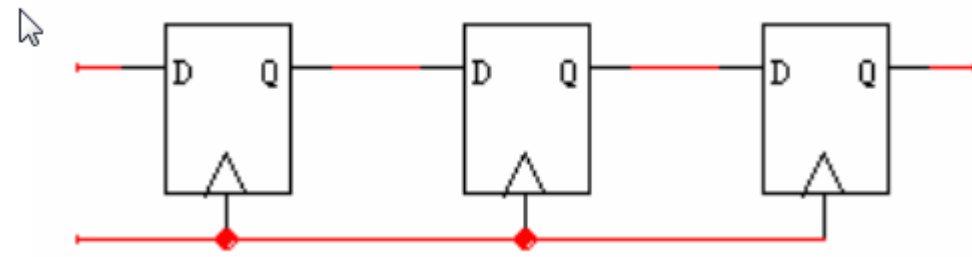
7.11 Blocking and Non-Blocking Assignments (I)

- Blocking assignments ($X=A$)
 - 계산과 동시에 저장에 이루어진다.
 - 다음 구문에 바로 계산값이 전달된다.
- Non-blocking assignments ($X<=A$)
 - 선 계산 후 저장한다.
 - 해당 구문의 계산을 금방 실행하고 계산된 값은 클럭 변경이 있을 때 까지 저장하지 않는다.
 - 동시에 여러 명령이 실행되는 경우 모든 계산이 완료된 후 저장작업이 수행된다.
 - 순차회로에서는 항상 Non-blocking assignments 를 이용해서 구현한다.

Blocking assignments



Non-blocking assignments



7.11 Blocking and Non-Blocking Assignments (II)

다음 코드는 상승 클럭에서 A 변수 값이 변화하는 순간에 모든 값이 변한다.

.

```
always @ ( posedge CLK)
begin
    B = A ;
    C = B ;
    D = C ; //A=B=C=D는 모두 같은 값
end
```

다음 코드는 상승 클럭에 각 변수의 값이 유지 된다..

```
always @ ( posedge CLK)
begin
    B <= A ;
    C <= B ;
    D <= C ; //A,B,C,D는 모두 다른 값
end
```

7.12 Edge-triggered Flip-flop

□ Edge-triggered D Flip-flop with q and q_bar outputs

```
module dff_bad1(clk, d, q, q_bar);
    input  d, clk;
    output q, q_bar;
    reg    q, q_bar;

    always @(posedge clk) begin // nonblocking assignments
        q      <= d;
        q_bar  <= ~d;
    end
endmodule
```

```
module dff_bad2(clk, d, q, q_bar);    Not Recommended!!
    input  d, clk;
    output q, q_bar;
    reg    q, q_bar;

    always @(posedge clk) begin // blocking assignments
        q      = d;
        q_bar  = ~d;
    end
endmodule
```

□ Edge-triggered D Flip-flop with q and q_bar outputs

q와 q_bar 출력을 갖는 D 플립플롭을 아래 코드와 같이 모델링하면, D 플립플롭이 아닌 다른 회로가 된다. 시뮬레이션을 통해서 해당 모델링이 어떤 회로로 동작하는지 확인하고, 그 이유를 생각해 본다.

```
module dff_bad2(clk, d, q, q_bar);    Not Recommended!!
    input  d, clk;
    output q, q_bar;
    reg    q, q_bar;

    always @(posedge clk) begin // blocking assignments
        q      = d;
        q_bar  = ~d;
    end
endmodule
```

- 순차회로에서 **blocking** 할당문 순서를 변경한 두 코드가 있다.
 - 서로 동일한 회로인가?
 - 만약 다른 회로라면 어떤 회로가 되는지 시뮬레이션 결과를 통해 확인하라.

```
module blk1(clk, d, q3);  
    input  clk;  
    output q3;  
    input  d;  
    reg    q3, q2, q1, q0;  
  
    always @(posedge clk) begin  
        q0 = d;    q1 = q0;  
        q2 = q1;    q3 = q2;  
    end  
endmodule
```

```
module blk2(clk, d, q3);  
    input  clk;  
    output q3;  
    input  d;  
    reg    q3, q2, q1, q0;  
  
    always @(posedge clk) begin  
        q3 = q2;    q2 = q1;  
        q1 = q0;    q0 = d;  
    end  
endmodule
```

7.13 Verilog HDL 코딩 가이드 라인

□ 코딩 가이드 라인 : race condition을 피하기 위해서

- ❖ 가이드라인-1 : 순차회로 또는 래치를 모델링하는 **always** 블록에서는 **nonblocking** 할당문을 사용한다.
- ❖ 가이드라인-2 : 조합논리회로를 모델링하는 **always** 블록에서는 **blocking** 할당문을 사용한다.
- ❖ 가이드라인-3 : 동일한 **always** 블록에서 순차회로와 조합논리회로를 함께 표현하는 경우에는 **nonblocking** 할당문을 사용한다.
- ❖ 가이드라인-4 : 동일한 **always** 블록 내에서 **blocking** 할당문과 **nonblocking** 할당문을 혼합해서 사용하지 않는다.
- ❖ 가이드라인-5 : 다수의 **always** 블록에서 동일한 **reg** 변수에 값을 할당하지 않는다.
- ❖ 가이드라인-6 : **nonblocking** (**<=**)으로 대입된 변수를 **simulation**에서 확인하고자 할 때에는 **\$strobe**를 이용해서 출력하자.
- ❖ 가이드라인-6 Simulation 시, **zero-delay**를 주지 말자 (같은 **time step**에서 **non-blocking** 보다 먼저 실행된다. 해당 **time step**의 최종 값을 보고싶다면 **#0 \$display** 이 아니라 **\$strobe** 명령어를 사용한다.)

- 다음 두 코드 A,B에서 각각 **y1, y2**를 확인하고, 그 결과에 대한 이유를 생각해 본다. 두 코드의 결과를 보고, 동일한 회로인지 서로 다른 회로가 되는지 확인한다.

코드 A

```
module fbosc_blk(y1,y2,clk,rst);
    output y1, y2;
    input  clk, rst;
    reg    y1, y2;

    always @(posedge clk or posedge rst)
        if(rst) y1 = 0; // reset
        else    y1 = y2;

    always @(posedge clk or posedge rst)
        if(rst) y2 = 1; // set
        else    y2 = y1;
endmodule
```

코드 B

```
module fbosc_nonblk(y1,y2,clk,rst);
    output y1, y2;
    input  clk, rst;
    reg    y1, y2;

    always @(posedge clk or posedge rst)
        if(rst) y1 <= 0; // reset
        else    y1 <= y2;

    always @(posedge clk or posedge rst)
        if(rst) y2 <= 1; // set
        else    y2 <= y1;
endmodule
```

7.14 시프트 레지스터 (Shift Register)

□ 시프트 레지스터

- ❖ 클럭신호가 인가될 때마다 데이터가 왼쪽 또는 오른쪽으로 이동되는 회로
- ❖ 여러 개의 플립플롭이 직렬로 연결된 구조
- ❖ 형태
 - 직렬입력 - 직렬출력 (**SISO**, Serial-In, Serial-Out)
 - 직렬입력 - 병렬출력 (**SIPO**, Serial-In, Parallel-Out)
 - 병렬입력 - 직렬출력 (**PISO**, Parallel-In, Serial-Out)
 - 병렬입력 - 병렬출력 (**PIPO**, Parallel-In, Parallel-Out)
 - 왼쪽 시프트, 오른쪽 시프트, 양방향 시프트
- ❖ **nonblocking** 할당문, 시프트 연산자, 결합 연산자, 반복문 등 다양한 구문으로 모델링

7.14.1 직렬입력-직렬출력 시프트레지스터

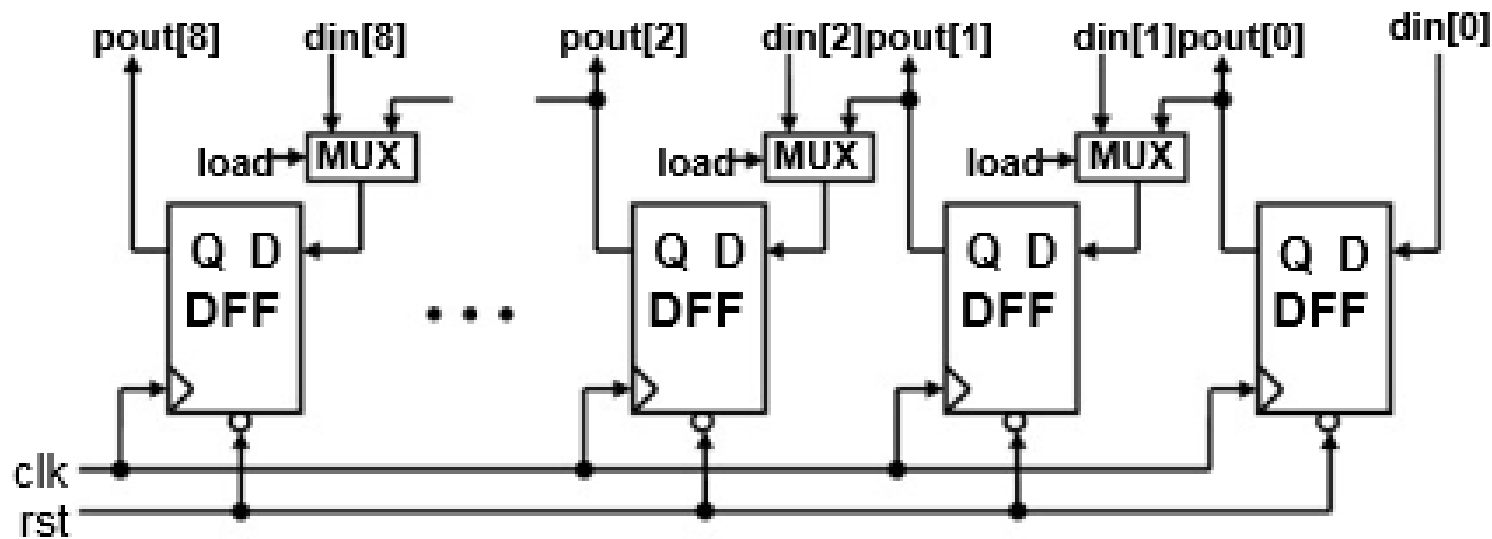
8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Clear, Serial In, and Serial Out

IO Pins	Description
clk	Positive-Edge Clock
sin	Serial In
CLR	Asynchronous Clear (active High)
sout	Serial Output

```
module shift_reg (clk, CLR, sin, sout);
    input      clk, CLR, sin;
    output     sout;
    reg [7:0] q;

    assign sout = q[7];
    always @(posedge clk) begin
        if(!CLR)
            q <= 0;
        else begin
            q[0] <= sin;
            q[7:1] <= q[6:0];
        end
    end
endmodule
```

7.14.2 병렬입력-병렬출력 시프트레지스터



```
module pld_shift_reg(clk, rst, load, din, pout);
    input      clk, rst, load;
    input  [7:0] din;
    output [7:0] pout;
    reg  [7:0] data_reg;
    assign pout = data_reg;
    always @(posedge clk) begin
        if(!rst)      data_reg <= 0;
        else
            if(load) data_reg <= din;
            else      data_reg <= data_reg << 1;
        end
    end
endmodule
```

7.14.3 병렬입력-병렬출력 시프트 레지스터

□ 양방향 병렬포트를 갖는 시프트 레지스터

- ❖ 데이터의 병렬로드(**wr=0**)와 시프팅 동작(**en=0**)이 동시에 일어나지 않으며,
- ❖ **data_io** 포트가 **inout** 이므로 **rd** 신호와 **wr** 신호가 동시에 **0**이 되지 않는다.

양방향 병렬포트를 갖는 시프트 레지스터의 신호 정의

신호 이름	기능
clk	클록
rst	리셋 (Active Low)
en	시프팅 동작을 위한 enable (Active Low)
wr	병렬로드를 위한 enable (Active Low)
rd	병렬출력을 위한 enable (Active Low)
si	직렬입력
so	직렬출력
data_io	병렬입/출력 데이터(inout)

7.14.3 병렬입력-병렬출력 시프트 레지스터

□ 양방향 병렬포트를 갖는 시프트 레지스터

```
module shifter(clk, rst, en, wr, rd, si, so, data_io);
    parameter      Len = 8;
    input          clk, rst, en, wr, rd, si;
    output         so;
    inout [Len-1:0] data_io;
    reg    [Len-1:0] shift_reg;

    assign data_io = !rd ? shift_reg : {Len{1'bz}};
    assign so = shift_reg[7];

    always @(posedge clk) begin
        if(!rst)
            shift_reg <= {Len{1'b0}};
        else begin
            if(!en) begin
                shift_reg <= shift_reg << 1;
                shift_reg[0] <= si;
            end
            else if(!wr)
                shift_reg <= data_io;
        end
    end
end
endmodule
```

7.14.3 병렬입력-병렬출력 시프트 레지스터

□ 양방향 병렬포트를 갖는 시프트 레지스터

```
module shifter(clk, rst, en, wr, rd, si, so, data_io);
    parameter      Len = 8;
    input          clk, rst, en, wr, rd, si;
    output         so;
    inout [Len-1:0] data_io;
    reg    [Len-1:0] shift_reg;

    assign data_io = !rd ? shift_reg : {Len{1'bz}};
    assign so = shift_reg[7];

    always @(posedge clk) begin
        if(!rst)
            shift_reg <= {Len{1'b0}};
        else begin
            if(!en) begin
                shift_reg <= shift_reg << 1;
                shift_reg[0] <= si;
            end
            else if(!wr)
                shift_reg <= data_io;
        end
    end
end
endmodule
```

Verilog HDL 실습 ► 시프트 레지스터

- 예제로 제시된 병렬입력 병렬 출력 코드와 신호 정의표에 다음 아래와 같은 정의를 추가해서 시프트 레지스터에 좌/우 시프팅 기능을 추가하여 설계 하고, 시뮬레이션을 통해 동작을 확인한다.

신호 이름	기 능
mode	좌/우 시프팅 선택 신호 (mode=0 ; 오른쪽 시프트, mode=1 ; 왼쪽 시프트)

7.15 카운터 (Counter)

□ 카운터(counter)

- ❖ 클럭펄스가 인가될 때마다 값을 증가 또는 감소시키는 회로
- ❖ 주파수 분주기, 타이밍 제어신호 생성 등에 사용
- ❖ 동기식 카운터
 - 모든 플립플롭이 하나의 공통 클럭신호에 의해 구동되며, 모든 플립플롭의 상태변경이 동시에 일어남
 - 장점 : 설계와 검증이 용이하며, 카운팅 속도가 빠름
 - 단점 : 비동기식 카운터에 비하여 회로가 복잡함
- ❖ 비동기식 카운터
 - 첫단의 플립플롭에 클럭신호가 인가되면, 플립플롭의 출력이 다음 단의 플립플롭을 트리거시키는 방식으로 동작
 - 리플 카운터(ripple counter)라고도 함
 - 장점 : 동기식 카운터에 비해 회로가 단순해짐
 - 단점 : 각 플립플롭의 전파 지연시간이 누적되어 최종단의 출력에 나타나므로 카운팅 속도가 느림

7.15 카운터

□ 8비트 증가 카운터 (reset 기능을 가지는)

```
module up_counter      ( out, enable, clk, reset )
output [7:0] out;
input enable, clk, reset;
reg [7:0] out;

always @(posedge clk)
if (reset) begin
    out <= 8'b0 ;
end else if (enable) begin
    out <= out + 1;
end

endmodule
```

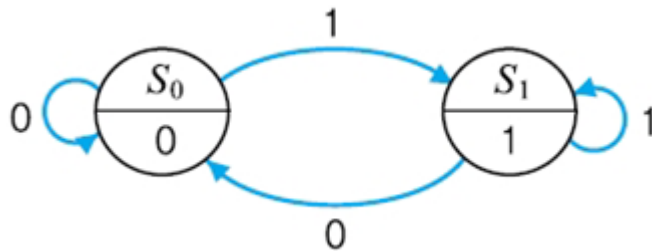

1. **Enable** 신호(**active high**)를 갖는 **8비트 감소 카운터**를 설계하고, 시뮬레이션을 통해 동작을 확인한다. **Enable** 신호 **en=1**이면 카운터가 동작하고, **en=0**이면 계수동작을 멈추는 기능을 갖는다.
2. **Mode** 신호에 따라 카운터 값이 증가(**mode=1**) 또는 감소(**mode=0**)되는 **8비트 증가/감소 카운터**를 설계하고, 시뮬레이션을 통해 동작을 확인한다.

7.16 유한상태 머신(FSM)

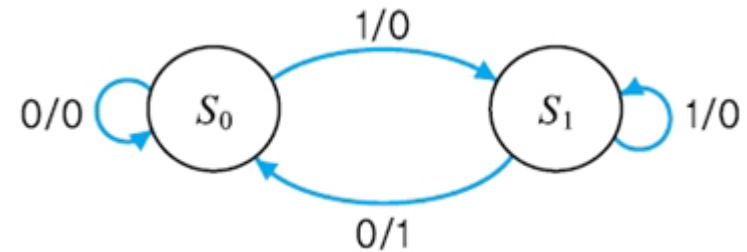
□ 유한상태 머신(Finite State Machine; FSM)

- ❖ 지정된 수의 상태를 가지고 상태들 간의 천이에 의해 출력을 생성하는 회로
- ❖ 디지털 시스템의 제어회로 구성에 사용
- ✓ **Moore** 머신 : 출력이 현재상태에 의해서만 결정, 출력이 상태내에 표시
- ✓ **Mealy** 머신 : 출력이 현재상태와 입력에 의해 결정, 출력이 상태간을 지나가는 화살표 위에 표시

Moore 머신



Mealy 머신



7.16 FSM

- ✓ 상태할당- FF은 이진수 외에 문자는 저장 불가능 하기 때문에 각각의 상태에 이진 데이터를 부여 하는 방식
 - ✓ 상태에 이진 수 인코딩 방식에 따라 상태 레지스터의 비트 수가 달라짐

상태 인코딩 예

상태번호	2진 인코딩	Gray 인코딩	Johnson 인코딩	One-hot 인코딩
0	000	000	0000	00000001
1	001	001	0001	00000010
2	010	011	0011	00000100
3	011	010	0111	00001000
4	100	110	1111	00010000
5	101	111	1110	00100000
6	110	101	1100	01000000
7	111	100	1000	10000000

7.16 FSM

□ 유한상태 머신 (FSM) 의 코딩 가이드라인

- ❖ **FSM**을 구성하는 3개의 블록(**next state logic, state register, output logic**)을 분리된 **always** 블록 또는 **assign** 문으로 구현한다.
 - **FSM** 코딩의 일관성을 좋게 하고, 수정 및 변경을 용이하게 한다.
- ❖ 상태 레지스터는 래치보다는 플립플롭을 사용하는 것이 좋다.
 - 래치를 사용하는 경우, 래치가 **transparent** 상태일 때 상태 발진(**state oscillation**)을 일으킬 수 있다.
- ❖ **Next state logic**은 **case** 문을 이용하여 모델링하는 것이 바람직하며, **FSM**에서 정의되지 않은 상태들은 **default** 문으로 표현한다.

출처 : Cummings, Clifford E. "Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements." *SNUG (Synopsys Users Group San Jose, CA 2003) Proceedings* (2003).

7.16 FSM

□ 유한상태 머신(FSM)의 코딩 가이드라인

- ❖ 각각의 상태머신을 독립된 **Verilog 모듈**로 설계한다.
 - **FSM** 모델의 유지가 용이하고, **FSM** 합성 툴의 최적화 작업에 도움이 된다.
- ❖ **FSM**의 상태 이름을 **parameter**로 정의하여 사용한다.
 - 컴파일러 지시어인 '**define**'을 이용하여 상태이름을 정의할 수도 있으나 '**define**'은 컴파일 과정에서 광역적으로 영향을 미치므로, 다수의 **FSM**에서 동일한 상태이름이 사용되는 경우에 오류가 발생된다.
 - **parameter**는 국부적인 영향을 미치므로 다른 **FSM**에 영향을 미치지 않는다.
- ❖ **FSM**이 비동기 리셋을 갖도록 설계한다.
 - 리셋을 갖지 않는 **FSM**은 초기 전원 인가 후, 상태 레지스터의 초기값이 불확정적이므로 정의되지 않은 상태에 갇혀서 빠져 나오지 못하는 상황이 발생할 수 있다.

출처 : Cummings, Clifford E. "Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements." *SNUG (Synopsys Users Group San Jose, CA 2003) Proceedings* (2003).

7.16 FSM

□ Two Always Block FSM Style (Good Style)

```
module fsm_4states (output reg gnt, input dly, done, req, clk, rst_n);
```

```
parameter [1:0] IDLE = 2'b00,  
BBUSY = 2'b01,  
BWAIT = 2'b10,  
BFREE = 2'b11;  
reg [1:0] state, next;
```

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

```
always @(state or dly or done or req) begin  
next = 2'bx;  
gnt = 1'b0;  
case (state)
```

```
    IDLE : if (req) next = BBUSY;  
           else next = IDLE;
```

```
    BBUSY: begin  
        gnt = 1'b1;  
        if (!done) next = BBUSY;  
        else if ( dly) next = BWAIT;  
        else next = BFREE;
```

```
    end  
    BWAIT: begin  
        gnt = 1'b1;  
        if (!dly) next = BFREE;  
        else next = BWAIT;  
    end
```

```
    BFREE: if (req) next = BBUSY;  
           else next = IDLE;
```

```
endcase end endmodule
```

7.16 FSM

□ 동기 순차시스템 설계 절차(synchronous sequential logic system)

단계 1: 문제에 대한 설명으로부터 메모리에 저장되어야 하는 것을 결정한다. 즉 가능한 상태가 무엇인지를 결정한다.

단계 2: 필요하다면 입,출력을 2진수로 코드화한다.

단계 3: 시스템의 동작을 설명하기 위해 상태표나 상태도를 만든다.

단계 4 : 입출력 관계는 같지만 상태의 개수가 적은 상태표를 만들기 위해 상태 축소화 기법(교재 5장 5.7절 참조)을 사용한다.

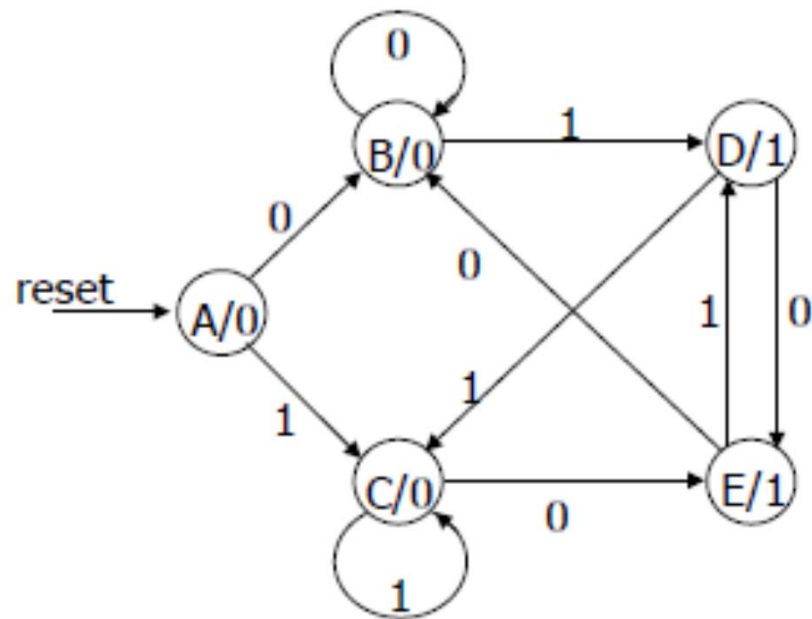
단계 5 : 상태할당을 한다. 즉, 상태를 이진수로 코딩한다.

단계 6: 플립플롭의 종류를 선택하고 플립플롭의 입력 맵 또는 표를 만든다.

단계 7: 논리식을 구하고 회로도를 그린다. (조합논리 설계 방식)

단계 8 : 단계 7에 결과를 이용해서 HDL을 통해 시뮬레이션 검증한다.

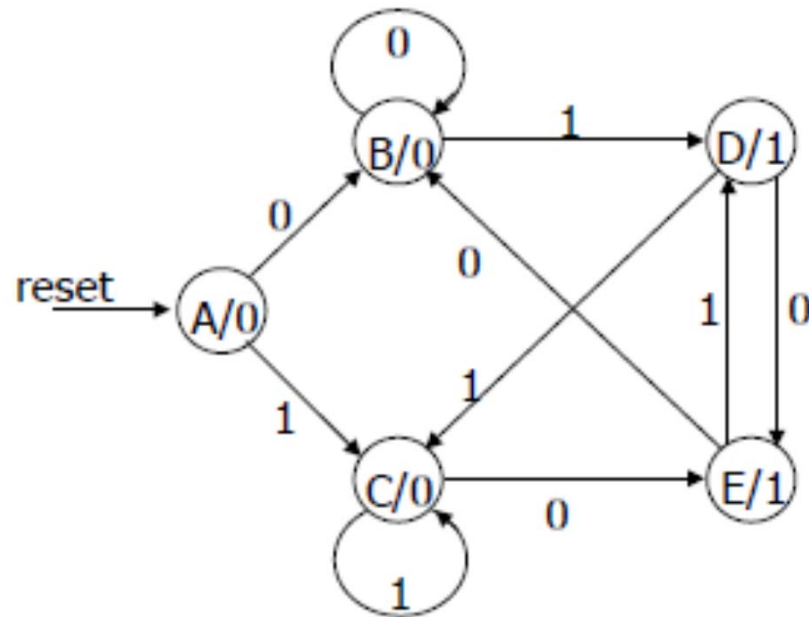
7.17 Moore FSM : Edge Detector (I)



parameter A = 2'b000, B = 2'b001,
C = 2'b010, D = 2'b011, E = 2'b100

```
reg [2:0] state, // Current state
nxtState; // Next state
always @(posedge clk) begin
    if (reset) begin
        state <= A; // Initial state
    end else begin
        state <= nxtState;
    end
end
```

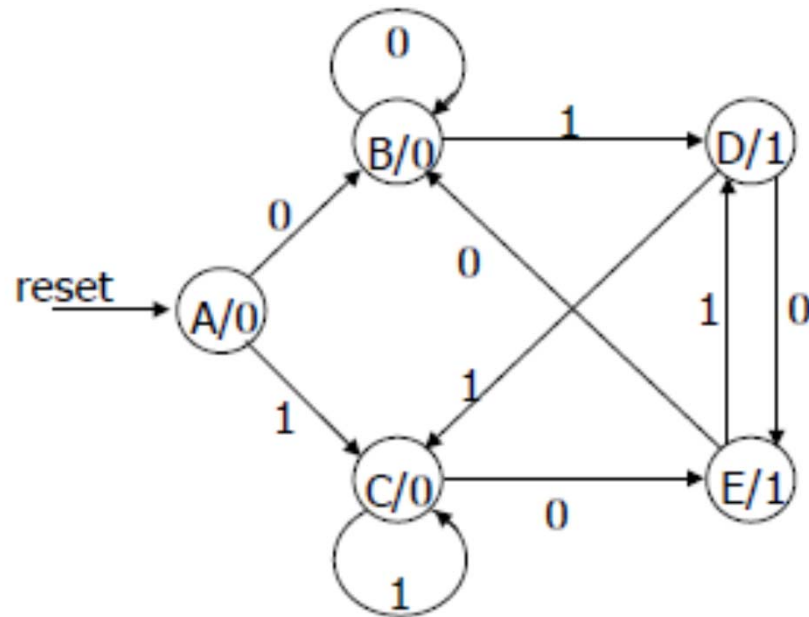

7.17 Moore FSM : Edge Detector (II)



```
always @(*) begin
    nxtState = state;
    out = 0;
    case (state)
        A : if (in) nxtState = C;
            else nxtState = B;
        B : if (in) nxtState = D;
        C : if (~in) nxtState = E;
        D : begin
            out = 1;
            if (in) nxtState = C;
            else nxtState = E;
        end
        E : begin
            out = 1;
            if (in) nxtState = D;
            else nxtState = B;
        end
        default : begin
            out = 1'bX;
            nxtState = 3'bX;
        end
    endcase
end
```

7.17 Moore FSM : Edge Detector (III)

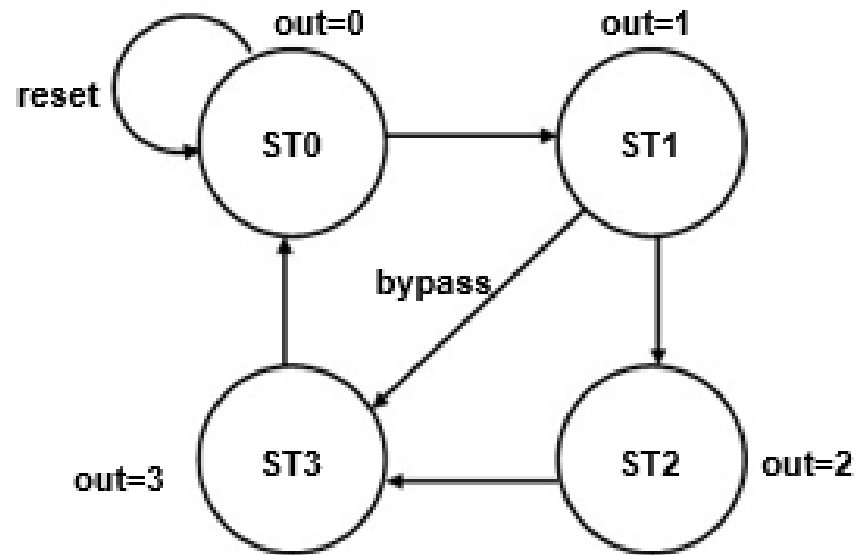
Using state assignment for output



```
always @(*) begin
    nextState = state;
    out = state[2];
    case (state)
        A : if (in) nextState = C;
            else nextState = B;
        B : if (in) nextState = D;
        C : if (~in) nextState = E;
        D : if (in) nextState = C;
            else nextState = E;
        E : if (in) nextState = D;
            else nextState = B;
        default : nextState = 3'bX;
    endcase
end
```

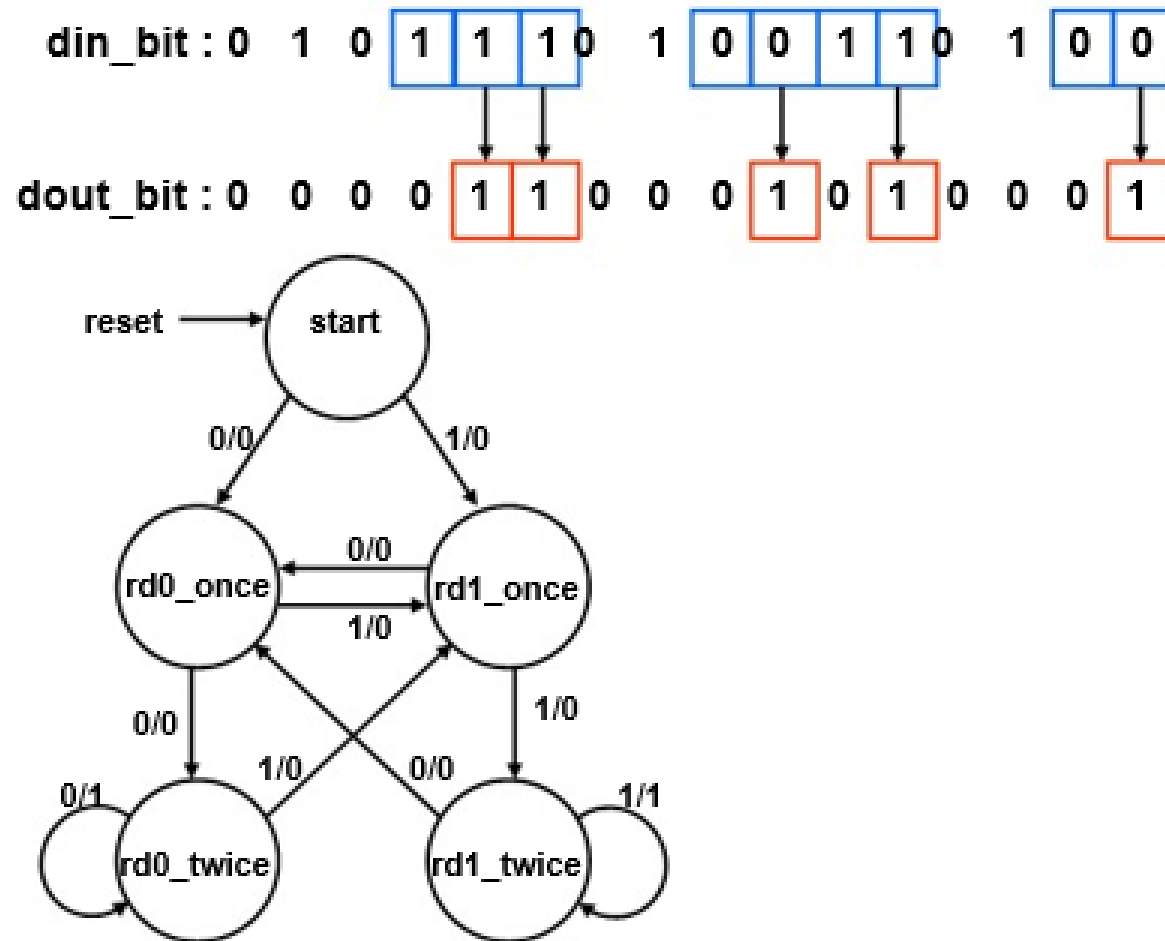
Verilog HDL 실습 ► Moore FSM

- 아래의 상태 천이도를 갖는 **Moore FSM** 회로를 설계하고, 시뮬레이션을 통해 동작을 확인한다.



7.18 Mealy FSM ; 연속비트 검출

- 입력된 비트 열중에 연속된 **0** 또는 **1** 입력을 검출하여 연속이면 1로 연속이 아니라면 0로 출력 비트를 만드는 회로의 상태도이다.



7.18 Mealy FSM ; 연속비트 검출 HDL 코드(I)

```
module seq_det_mealy(clk, rst, din_bit, dout_bit);
    input          clk, rst, din_bit;
    output dout_bit;
    reg    [2:0]    state_reg, next_state;

    // 상태 선언
    parameter      start      = 3'b000;
    parameter      rd0_once   = 3'b001;
    parameter      rd1_once   = 3'b010;
    parameter      rd0_twice  = 3'b011;
    parameter      rd1_twice  = 3'b100;

    //State Register
    always @(posedge clk or posedge rst) begin
        if(rst == 1) state_reg <= start;
        else          state_reg <= next_state;
    end

    //Output Logic
    assign dout_bit = (((state_reg == rd0_twice) &&(din_bit == 0) ||
                        (state_reg == rd1_twice) &&(din_bit == 1))) ? 1 : 0;
```

Mealy FSM ; 연속비트 검출 HDL 코드(II)

```
//Next State Logic
always @(state_reg or din_bit) begin
case(state_reg)
    start      : if      (din_bit == 0)    next_state = rd0_once;
                  else if(din_bit == 1)    next_state = rd1_once;
                  else                      next_state = start;
    rd0_once   : if(din_bit == 0)          next_state = rd0_twice;
                  else if(din_bit == 1)    next_state = rd1_once;
                  else                      next_state = start;
    rd0_twice  : if(din_bit == 0)          next_state = rd0_twice;
                  else if(din_bit == 1)    next_state = rd1_once;
                  else                      next_state = start;
    rd1_once   : if(din_bit == 0)          next_state = rd0_once;
                  else if(din_bit == 1)    next_state = rd1_twice;
                  else                      next_state = start;
    rd1_twice  : if(din_bit == 0)          next_state = rd0_once;
                  else if(din_bit == 1)    next_state = rd1_twice;
                  else                      next_state = start;
    default    :                          next_state = start;
endcase
end
endmodule
```

Verilog HDL 실습 ► Mealy FSM

다음은 입력 비트열에서 0 다음에 1이 연속으로 나오고 그 다음 0이 입력이 되면 출력 비트에서는 해당 0이 입력된 자리를 1로 표시하는 회로의 상태도이다. 해당 상태도를 참조해서 HDL을 통해 TestBench를 통한 시뮬레이션을 해보고 해당 결과를 확인해라.

