Verilog HDL

Digital Design and Modeling

Coding Styles and Methodologies

Young-Ho Seo
Kwangwoon University
design@kw.ac.kr
http://explore.kw.ac.kr/~axl

Content

1. Digital Design Overview using Verilog HDL

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

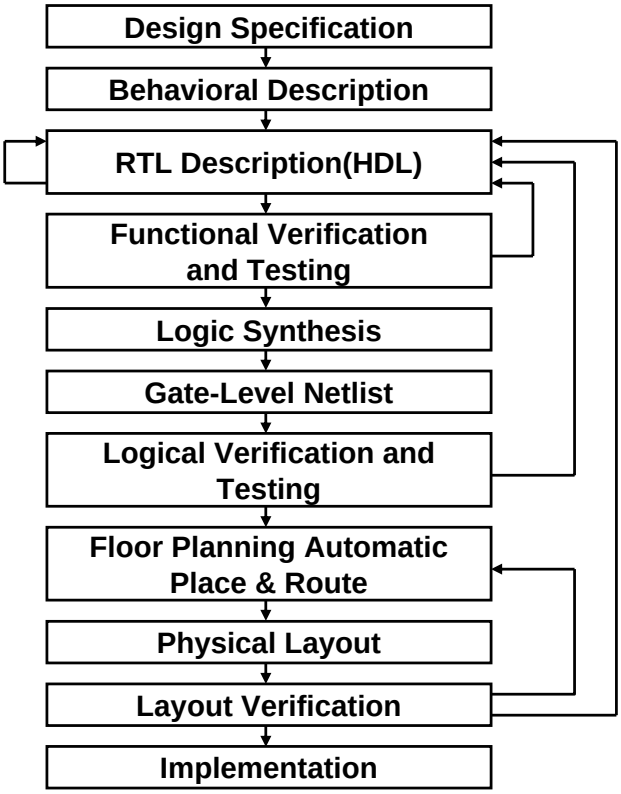
1.1 컴퓨터 지원 디지털 설계의 진전

- 회로의 집적화
  - 소규모 집적회로 – SSI : Small Scale Integration
  - 중규모 집적회로 – MSI : Medium Scale Integration
  - 대규모 집적회로 – LSI : Large Scale Integration
- 컴퓨터 지원 설계
  - CAD : Computer Aided Design
- 초대규모 집적회로
  - Very Large Scale Integration

1.2 HDL의 탄생

- **1983년 Gateway Design Automation사에서 개발**
- **DARPA와의 계약하에 VHDL 개발**
- **1980년대 후반 논리합성 등장**
  - 설계 방법론의 급진전
  - 논리합성의 발달로 HDL이 디지털 설계의 선두자리
- 시스템 수준의 설계에도 사용

1.3 일반적 설계 과정



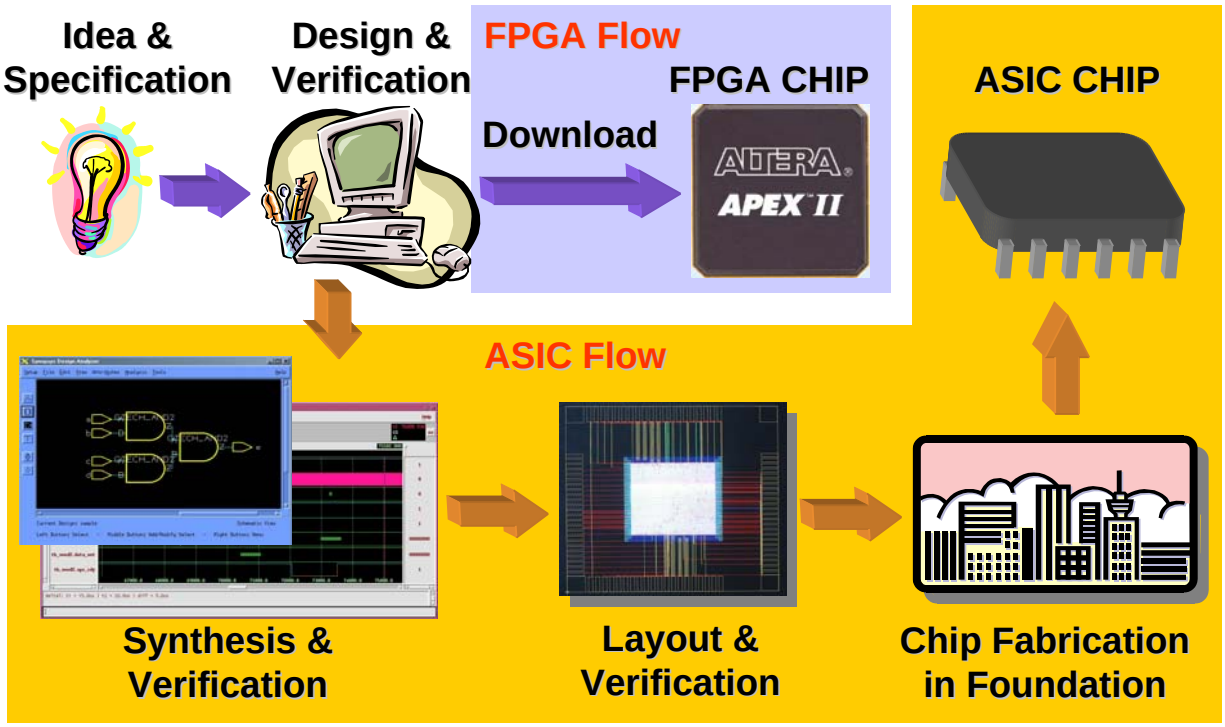
1.4 HDL의 중요성

- 추상화된 수준에서 설계 표현
- 설계 주기의 초반에 설계의 기능적 검증
- 컴퓨터 프로그래밍과 유사

1.5 Verilog HDL의 대중성

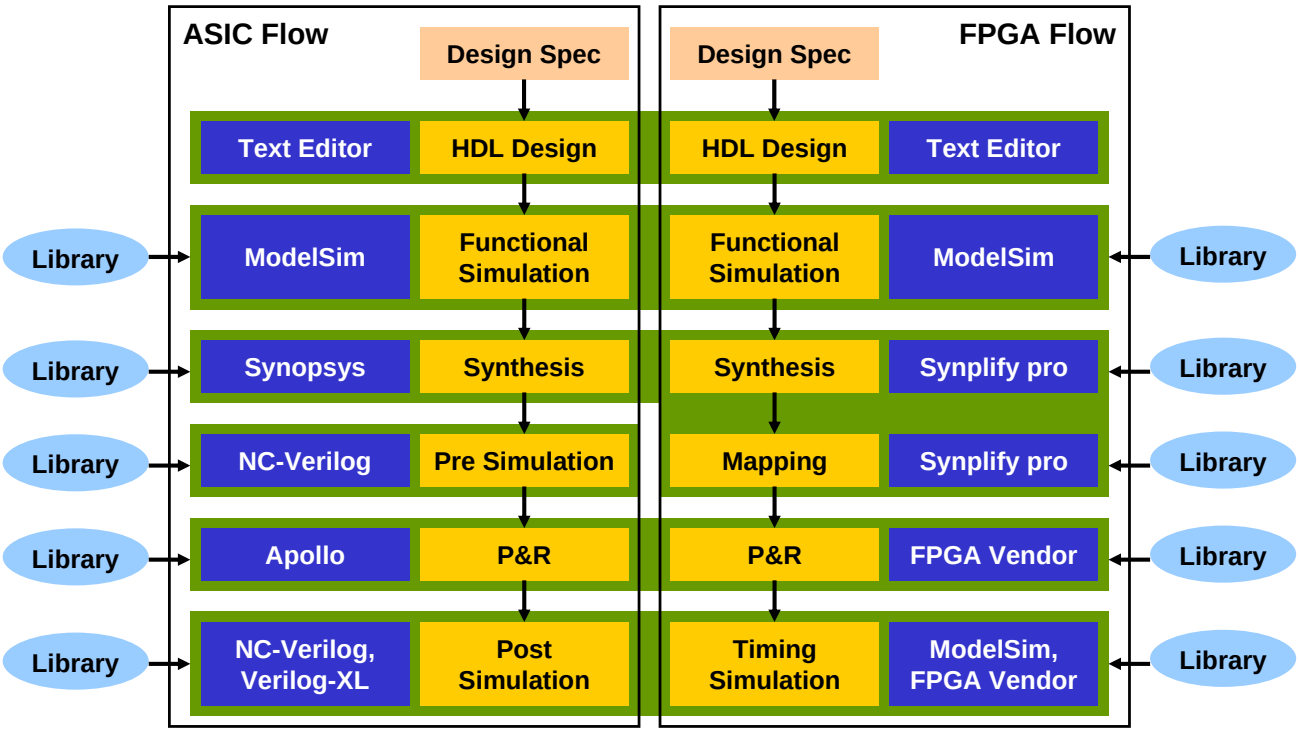
- C 언어와 유사한 형태 → 습득과 사용이 용이
- 하나의 동일한 회로 모델안에서 서로 다른 추상화 수준을 섞어서 설계 가능
- 대부분의 대중적인 논리합성 도구들이 **Verilog HDL**을 지원
- 모든 제작 업체들이 후반기 논리합성 시뮬레이션을 위한 **Verilog HDL** 라이브러리를 제공
- **Verilog HDL**의 **PLI(Programming Language Interface)**는 **Verilog** 내부 데이터 구조와 상호작용하는 사용자 **C** 코드를 사용 가능

1.6 HDL의 동향



1.6 HDL의 동향 (cont)

- Modern Design Flow using HDL



2. Structure of Hierarchical Modeling

Young-Ho Seo

http://explore.kw.ac.kr/~axl

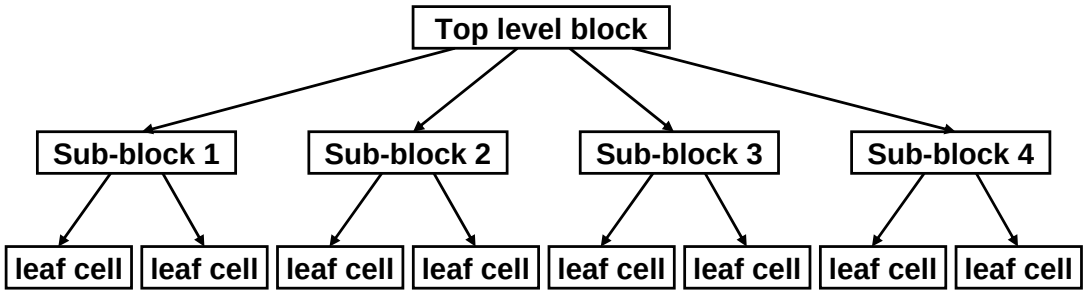
design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

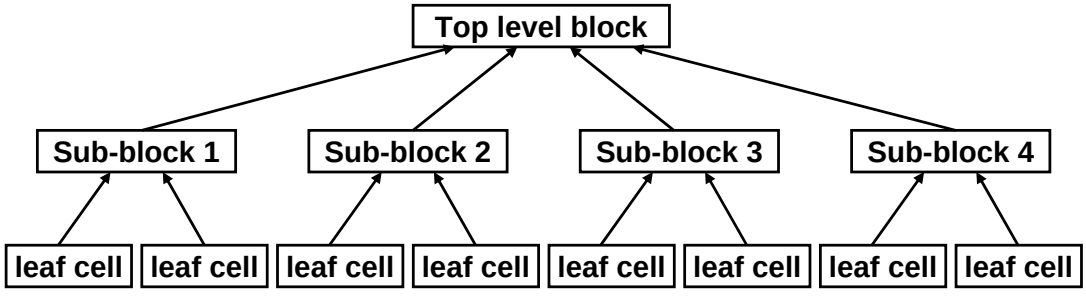
Content

2.1 설계 방법론

- **top-down** 설계 방법론



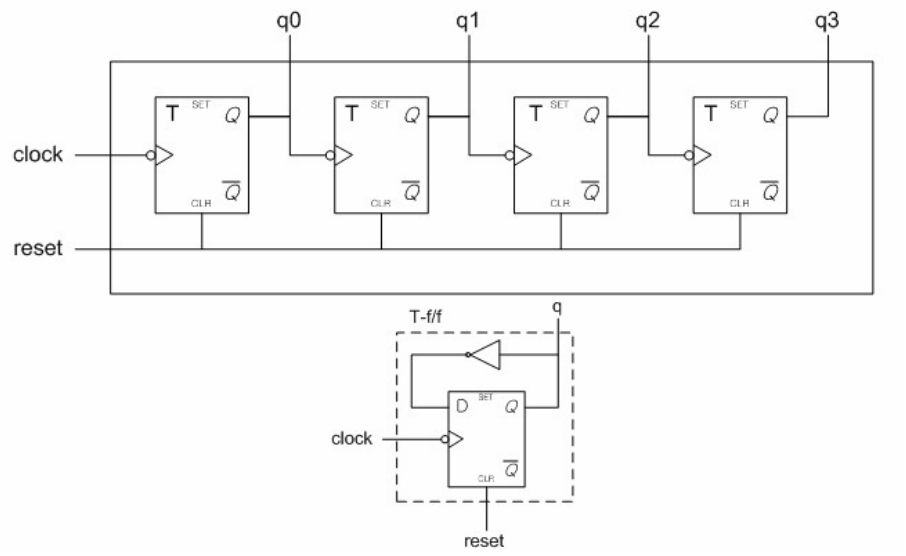
- **bottom-up** 설계 방법론



2.2 4-비트 리플 캐리 카운터

- **top-down 설계 방법**
  - ◆ 카운터 기능 정의 → T-f/f으로 카운터 구현 → D-f/f으로 T-f/f 구현

- **bottom-up 설계 방법론**
  - ◆ D-f/f으로 T-f/f 구현 → T-f/f으로 카운터 구현 → 카운터 동작



2.3 모듈(Module)

- **Module – 기본적인 설계 블록**
  - ◆ 기본적인 설계 블록
  - ◆ VHDL에서 architecture에 해당
  - ◆ 포트 인터페이스(입력, 출력)를 통해서 상위 수준의 블록에 필요한 기능 제공
- **문법**

```
module <모듈이름> (<모듈 터미널 리스트>)
...
...
<모듈 내용>
...
...
endmodule
```

```
module T_FF (Q, clk, reset)
...
...
<T-f/f의 기능>
...
...
endmodule
```

2.3 모듈(Module) (cont)

■ Verilog HDL vs. VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity ff is
port(
    reset, clk, d   :in    std_logic;
    q               :out std_logic);
end ff;

architecture logic of ff is
begin

process(reset, clk)
begin
if (reset='1') then
    q <= '0';
elsif (clk='1' and clk'event) then
    q <= d;
end if;
end process;

end logic;
```

```
module ff(reset, clk, d, q)
    input    reset, clk, d;
    output  q;

always @(posedge clk or posedge reset)
begin
    if (reset)
        q <= 1'b0;
    else
        q <= d;

end module;
```


2.3 모듈(Module) (cont)

■ 수준의 정의

- ◆ Behavioral or algorithm level
  - ■ 하드웨어 구현에 관계없이 원하는 설계 알고리듬을 바로 사용
  - ■ C 프로그래밍과 유사
- ◆ Dataflow level
  - ■ 데이터의 흐름을 명백히 기술
  - ■ 하드웨어 레지스터 사이의 데이터 흐름과 데이터 처리
- ◆ Gate level
  - ■ 논리 게이트의 연결에 의해 모듈 구현
- ◆ Switch level
  - ■ 스위치와 기억노드의 연결에 의해서 모듈 구현

2.4 인스턴스(Instance)

- **Instance – 모듈템플릿으로부터 생성된 객체**
  - 모듈은 실제 객체를 만들 수 있는 템플릿을 제공
  - 모듈의 호출 시 Verilog는 템플릿으로부터 고유한 객체 생성
  - 객체는 이름, 변수, 파라미터, 그리고 입출력 인터페이스를 가짐
  - 모듈 템플릿으로부터 객체를 생성하는 것을 파생(Instantiation)
  - 파생된 객체가 인스턴스(Instance)
- 예제 그림

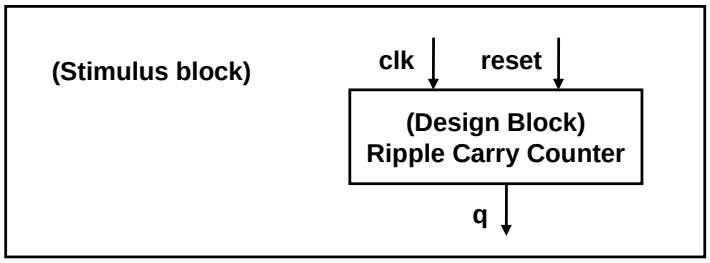
2.4 인스턴스(Instance) (cont)

- 예제

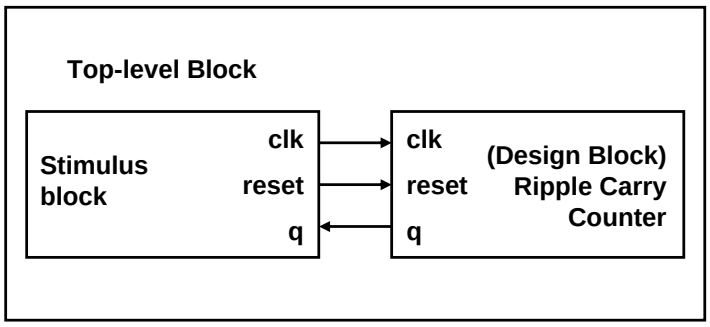
```
module ripple_carry_counter(q, clk, reset);   ← 카운터 모듈 정의
output [3:0] q;
input clk, reset;
T_FF tff0(q[0], clk, reset);      ← 네 개의 T_FF 모듈 인스턴스의 생성
T_FF tff1(q[1], q[0], reset);
T_FF tff2(q[2], q[1], reset);
T_FF tff3(q[3], q[2], reset);
end module
module T_FF(q, clk, reset);   ← T_FF 모듈의 정의
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);   ← D_FF의 파생, 인스턴스 이름은 dff0
not n1(d, q);
end module;
```

2.5 시뮬레이션 요소

- **Stimulus 블록 : test bench**
- 설계 블록을 파생하는 스티뮬러스 블록



- 스티뮬러스 블록과 설계 블록이 가상의 **Top** 모듈에서 파생



2.6 예제

- **Ripple Carry Counter – ModelSim으로 Simulation**

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

module stimulus;
reg clk;
reg reset;
wire [3:0] q;
ripple_carry_couunter r1(q, clk, reset);
```

```
initial
    clk=1'b0;
always
  #5 clk=~clk;
initial
begin
  reset=1'b1;
  #15 reset=1'b0;
  #180 reset=1'b1;
  #10 reset=1'b0;
  #20 $finish;
end

initial
  $monitor($time, "Output q=%d", q);
end module
```

3. Basic Concept

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

3.1 사전적 규약

- **3.1.1 화이트 스페이스**
- **3.1.2 주석**
  - // : 한줄 주석
  - /*....*/ : 여러줄 주석
- **3.1.3 연산자**
  - 단항, 이항, 삼항
- **3.1.4 수 표현**
  - 크기 지정 가능 수

<크기> <기본 형식> <숫자>

- 4'b111 // 4비트 2진수
- 12'habc // 12비트 16진수
- 16'd255 // 16비트 10진수

3.1 사전적 규약 (cont)

- 크기 지정 불가능수 – 환경에 따라 다름

<기본 형식>

- 23456 // 32비트 10진수
- 'hc3 // 32비트 16진수
- 'o21 // 32비트 8진수

- x, z 값
  - 12'h13x // 12비트 16진수; 마지막 4개 비트는 알수 없는 값
  - 6'hx // 6비트 16진수; 전체 벡터를 x로 채움
  - 32'bz // 32비트 2진수; 하이 임피던수 값; 전체 벡터를 z로 채움
- 음수
  - <크기> 앞에 음수 부호를 붙임
  - -6'd3 // 3의 2의 보수로써 음수

3.1 사전적 규약 (cont)

- 언더스코어 문자와 물음표
  - 숫자에서 언더스코어는 가독성을 높여주는 역활로 Verilog에서 무시됨
  - 숫자에서 물음표는 하이임피던스
  - 12'b1111_0000_1010 // 12'b111100001010과 동일
  - 4'b10?? // 4'b10zz와 동일

3.1.5 문자열

- 큰 따옴표 사이의 문자열
  - "Hello Verilog World"
  - "a/b"

3.1.6 식별자와 키워드

- 키워드 – 언어구조를 정의하기 위해 미리 예약된 특별한 식별자
  - reg value; //reg는 키워드, value는 식별자

3.2 데이터형

3.2.1 논리값 집한

- 4개의 논리값과 8개의 신호강도

논리값 수준	상태
0	논리적 0, 거짓상태
1	논리적 1, 참상태
x	알수없는 논리값
z	하이임피던스, 플로팅상태

- 3.2.2 넷(Net)
- 넷은 하드웨어 요소사이에 연결을 나타냄
- 넷은 wire에 의해 정의
- 넷의 기본값은 z

3.2.3 레지스터

- 값을 저장할 수 있는 변수
- f/f에 의한 하드웨어 레지스터와 혼동하지 말것
- 다른 논리값이 들어오기 전까지 값을 유지
- 예제 (➔)

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

3.2 데이터형 (cont)

■ 3.2.4 벡터

◆ net과 reg 데이터 형은 벡터(여러 비트폭) 사용 가능

```
wire a;
wire [7:0] bus   //8비트 버스
wire [31:0] busA, busB, busC;
reg clock;
reg [0:40]   // 41비트 버스
```

■ 3.2.5 정수, 실수, 시간 레지스터 데이터형

◆ 정수 – integer에 의해 선언

```
integer counter;
initial
   counter=-1;
```

◆ 실수 – real에 의해 선언

```
real delta;
initial
begin
   delta=4e10;   // 과학적 표시법
   delta=2.13;
end
integer i;
initial
   i=delta;   // i는 2의 값을 가짐
```


3.2 데이터형 (cont)

◆ 시간 – time 키워드에 의해 선언

- 시뮬레이션 시간을 저장하기 위해 시간 레지스터 데이터형 사용
- 시스템 함수 $time은 현재 시뮬레이션 시간을 얻을 때 사용

```
time save_sim_time;
initial
   save_sim_time=$time;
```

■ 3.2.6 배열

◆ reg, integer, time 그리고 벡터 레지스터 데이터형의 배열 가능

◆ 실수 변수를 위한 배열은 없음

```
<배열 이름> [<서브스크립트>]
```

```
integer count[0:7];      // 8 count 변수의 배열
reg bool[31:0]          // 32 1-bit bool 레지스터 변수의 배열
time chk_point[1:100]     // 100 time checkpoint 변수의 배열
reg [4:0] port_id[0:7]      // 8 port_id(5-bit 폭)의 배열
integer matrix[4:0][4:0]    // 다차원 배열은 불법
```

3.2 데이터형 (cont)

```
count[5]          // count 변수 배열의 5번째 원소
chk_point[100]    // check point 값이 100번째 시간
port_id[3]        // port_id 배열의 3번째 원소
```

■ 3.2.7 메모리

◆ 간단한 메모리 사용 시 배열을 이용

■ 3.2.8 파라미터

◆ parameter 키워드를 이용해서 모듈내부에 상수 정의

```
parameter port_id=5;
parameter cache_line_width=256;
```


3.2 데이터형 (cont)

■ 3.2.9 문자열

◆ 문자열은 reg에 저장 가능

◆ reg의 크기는 문자열보다 커야하고 작을 경우 넘는 문자열 비트를 무시

```
reg [8*18:1] string_value;
initial
   string_value="Hello Verilog World";
```

◆ 특수문자 : 특별한 목적으로 사용, 에스케이프 문자가 선행

에스케이프 문자	출력된 문자
\n	개행
\t	탭
%%	%
\\	\
\"	"
\ooo	1-3 8진수로 쓰여진 문자

3.3 시스템 태스크와 컴파일러 지시어

■ 3.3.1 시스템 태스크

◆ 루틴 연산을 하기 위한 표준 시스템 태스크(standard system task) 제공

◆ 사용법

```
$<키워드>
```

◆ 화면출력 태스크 : 변수의 값, 문자열, 수식 출력

```
$diaplay(p1, p2, p3, …., pn);
```

◆ 모니터링 태스크 : 신호의 값이 변할 때마다 그 신호를 출력

```
$monitor(p1, p2, p3, …, pn);
```

◆ 시뮬레이션 중단과 종료 태스크

```
$stop;
$finish;
```


3.3 시스템 태스크와 컴파일러 지시어 (cont)

■ 3.3.2 컴파일러 지시어

◆ ‘define : 텍스트 매크로를 정의

■ C언어의 #define과 유사

```
‘define WORD_SIZE 32
‘define S $stop
‘define WORD_REG reg [31:0]
```

◆ ‘include : 다른 verilog 파일에 있는 내용을 컴파일 하는 동안 포함

```
‘include header.v
```

◆ ‘indef

◆ ‘timescale

4. Module and Port

Young-Ho Seo

http://explore.kw.ac.kr/~axl

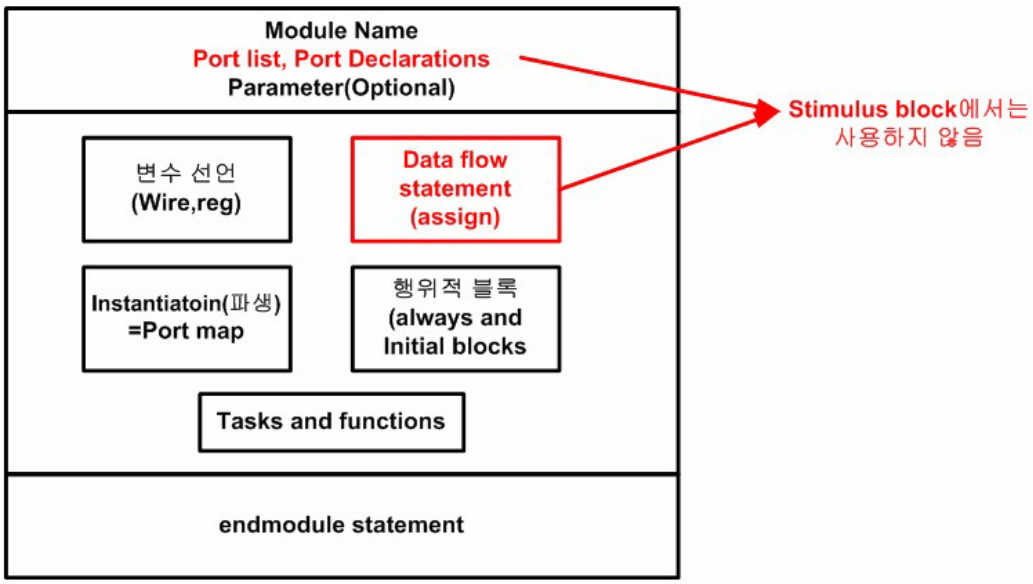
design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

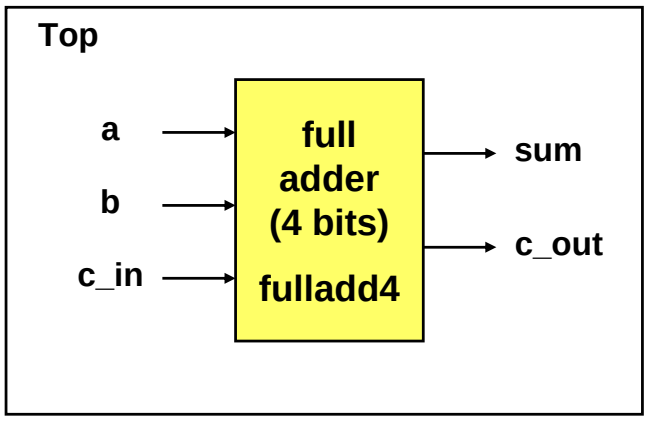
4.1 모듈

■ Module의 구성



4.2 포트

■ 4.2.1 포트 나열




```
module fulladd4(sum, c_out,a,b,c_in) ;  // 포트 리스트를 갖음
module Top ;     // Stimulus block(시뮬레이션시 최상위 블록) , 포트 리스트 없음
```

4.2 포트 (cont)

■ 4.2.2 포트 선언

◆ 포트 리스트의 모든 포트를 모듈 안에서 선언

Verilog 키워드	포트의 형
input	입력 포트
output	출력 포트
inout	양방향 포트

```
module fulladd4(sum, c_out,a,b,c_in) ;
// 포트 선언 부분
            output [3..0] sum;
            output c_out;

            input [3..0] a, b;
            input c_in;
// 포트 선언 끝 부분
…
모듈내용
…
endmodule
```


4.2 포트 (cont)

◆ 모든 포트는 기본적으로 wire로 선언

◆ 일반적으로 input 또한 inout 포트는 wire로 선언

◆ Output 포트가 값을 유지(F/F)해야 할 경우 reg로 선언

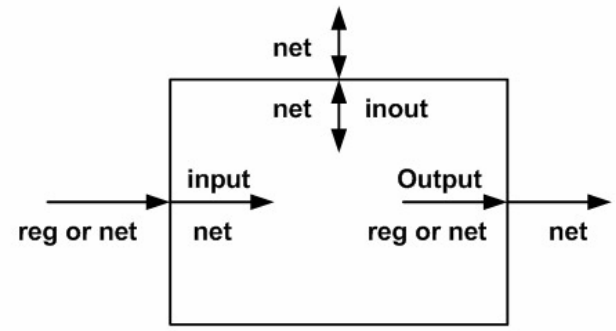
◆ input과 inout형의 포트는 reg로 선언될 수 없음

```
module DFF(q, d, clk, reset)
output q;
reg q;      // 출력 포트 q는 값을 유지 → reg로 선언
input d, clk, reset;
……
……
endmodule
```

4.2 포트 (cont)

■ 4.2.3 포트 연결 규칙

- ◆ 입력 : 내부적 – wire, 외부적 – wire, reg
- ◆ 출력 : 내부적 – reg, wire, 외부적 – wire
- ◆ 입출력 : 내부적 – wire, 외부적 – wire



- ◆ 크기 맞춤
- ◆ 연결되지 않은 포트

```
fulladd4 fa0(SUM,  , A, B, C_IN) ;  // 출력 포트 c_out이 연결되지 않았음
```


4.2 포트 (cont)

■ 4.2.4 외부 신호에 포트 연결하기

- ◆ 위치에 의한 연결

```
module Top;
// 연결 변수의 선언
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
   // fa_ordered라는 fulladd4의 파생
   // 신호가 위치에 의해 연결
   fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
   .....
   <스티뮬러스>
   .....
endmodule
```

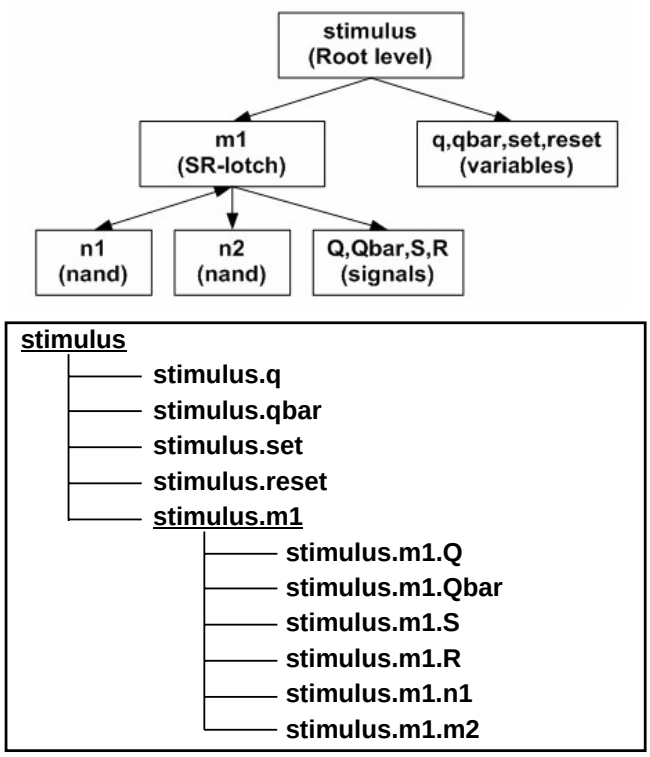
```
module fulladd4(sum, c_out, a, b, c_in);
output p3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
   .....
   <모듈 내용>
   .....
endmodule
```

- ◆ 이름에 의한 연결

```
fulladd4 fa_byneme(.c_in(C_IN), .sum(SUM), .a(A), .b(B), .c_out(C_OUT));
fulladd4 fa_byneme(.sum(SUM),   , .b(B), .c_in(C_IN), .a(A));
```

4.3 계층적 이름

■ 계층적 이름 참조



5. Gate Level Modeling

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

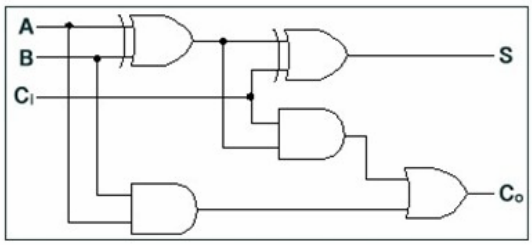
Digital Design & Test Lab.
Kwangwoon University

Content

5.1 게이트 형태

■ 게이트 형태

◆ 기본적인 논리 게이트를 primitive로 정의 → 그냥 사용하면 됨

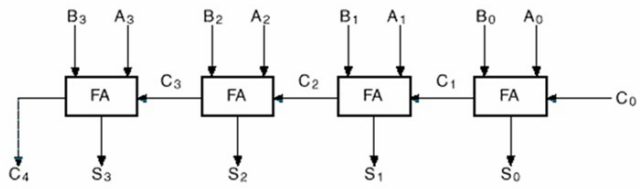



```
//1bit 전가산기의 정의
module fulladd(sum, c_out, a, b, c_in);

//I/O 포트 선언
output sum, c_out;
input a, b, c_in;
//내부 넷
wire s1, c2, c2;l

//프리미티브 논리 게이트 파생
xor (s1, a, b);
and(c2, a, b);

xor(sum, s1, c_in);
and(c2, s1, c_in);

or(c_out, c2, c1);
endmodule
```

5.1 게이트 형태

- **Verilog** 에서는 기본적인 논리 게이트들이 **primitive** 정의되어 있음
  - ◆ 모듈의 정의없이 파생하여 사용
- **5.1.1 And/Or** 게이트

5.1 게이트 형태




```
//1bit 전가산기의 정의
module fulladd4(sum, c_out, a, b, c_in);

//I/O 포트 선언
output [3..0]sum;
 output c_out;
input [3..0]a, b;
input c_in;
//내부 넷
wire s1, c2, c2;

//4개의 1-bit 전가산기의 파생
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa0(sum[1], c2, a[1], b[1], c1);
fulladd fa0(sum[2], c3, a[2], b[2], c2;
fulladd fa0(sum[3], c_out, a[3], b[3], c3);
endmodule
```

5.2 게이트 지연

- 게이트 지연
  - 상승, 하강, 턴-오프 지연
    - 상승 지연 : 0 →1 로 게이트의 출력이 변할 때
    - 하강 지연 : 1 →0 으로 게이트의 출력이 변할 때
    - 턴-오프 지연 : 어떤 값에서 임피던스값(z)로 출력이 변할 때

```
and #(5) a1(out, a, b);                  // 모든 변화에 대한 5 단위 시간 지연
and #(4,6) a2(out, a, b);                // 상승=4, 하강=6
bufif0 #(3,4,5) b1(out, in, control); // 상승=3, 하강=4, 턴-오프:5
```


5.2 게이트 지연 (cont)

- 최소/전형적/최대값
  - 최소 : 설계자가 예상한 게이트의 최소 지연값
  - 전형 : 설계자가 예상한 보통의 지연값
  - 최대 : 설계자가 예상한 게이트의 최대 지연값

```
and #(2:3:4, 3:4:5, 4:5:6) a3(out, a, b);
//2:3:4 - 상승 지연값에 대한 min, typ, max 값
//3:4:5 - 하강 지연값에 대한 min, typ, max 값
//4:5:6 - 턴-오프 지연값에 대한 min, typ, max 값
```

```
최소/최대/전형적 지연값에 대한 verilog-XL 시뮬레이터 예제

verilog test.v +maxdelays  // 최대 지연값을 위한 시뮬레이션
verilog test.v +typdelays  // 전형적 지연값을 위한 시뮬레이션
verilog test.v +mindelays   // 최소 지연값을 위한 시뮬레이션

test.v 에 지연을 갖는 모듈 포함
```

6. Date Flow Modeling

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

6.1 연속 할당문

■ 용도

◆ 하나의 값을 넷에 할당하는데 사용.

```
assign out = i1 & i2; // 연속 할당문. out, i1, i2은 net
// 벡터 넷 연속 할당문. addr는 16-비트 벡터 net
// addr1과 addr2는 16-비트 벡터 레지스터
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
assign c_out, sum[3:0] = a[3:0] + b[3:0] + c_in; // 결합 왼쪽은 스칼라 net과 벡터net의 결합
```

◆ 왼쪽은 항상 스칼라나 벡터의 넷, 또는 스칼라 넷과 벡터 넷으로 합쳐진 것
◆ 오른쪽은 레지스터 또는 넷 또는 함수 호출문.
◆ 능동적 : 오른쪽 피연산자들의 값이 바뀌자마자 왼쪽의 넷에 값을 할당
◆ 지연값은 할당문 안에서 단위 시간으로 지정

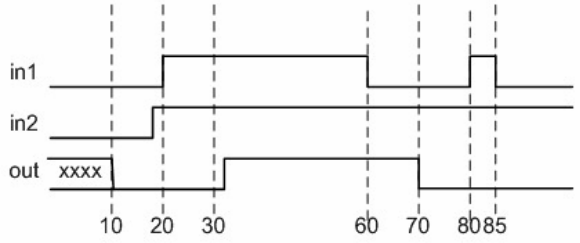
```
wire out;   // 일반적인 연속 할당문
assign out = in1 & in2;

// 위의 표현과 같은 효과가 있는 함축적인 연속 할당문
wire out = in1 & in2;
```


6.2 지연

■ 정규 할당 지연

```
assign #10 out = in1 & in2; // 연속 할당문의 지연값 지정
```

◆ in1과 in2의 값이 20 단위 시간에 상승 → out은 10만큼의 시간 후에 값이 상
◆ in1의 값이 60 단위 시간에서 하강 → out의 값은 70 단위 시간 변화
◆ in1의 80 단위 시간에서 값 변화 → 85 단위 시간에 다시 변화 → 90 단위 시간에 서는 out값은 0을 유지



■ 함축적 연속 할당 지연

```
wire #10 out = in1 & in2;     // 함축적 연속 할당 지연

wire out ;          // 같은표현
assign #10 out = in1 & in2;
```

6.2 지연 (cont)

■ 넷 선언 지연

◆ out에 일정한 지연값을 할당.

```
// 넷 지연
wire #10 out;
assign out = in1 & in2;

// 위의 문은 아래의 문과 같은 효과
wire out;
assign #10 out = in1 & in2;
```


6.3 수식, 연산자 그리고 피연산자

■ 수식

```
// 수식의 예. 연산자와 피연산자의 조합
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

■ 피 연산자

```
Integer count, final_count;
final_count = count + 1;            // count는 정수 피연산자

real a,b,c;
c=a-b;                              // a와 b는 real형 피연산자
reg[15:0] reg1, reg2;
reg[3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];  // reg1[3:0]과 reg2[3:0]는
                                    // 부분 선택 레지스터 피연산자

reg ret_value;
ret_value = calculate_parity(A, B);  // calculate_parity는 함수형 피연산자
```

6.3 수식, 연산자 그리고 피연산자 (cont)

■ 연산자

```
d1 && d2   // &&는 연산자, d1, d2는 피연산자
!a[0]      // !는 연산자, a[0]는 피연산자
B >> 1     // >>는 연산자, B와 1는 피연산자
```


6.4 연산자 형태

연사자 형태	연산자 기호	연산자 역할	피연산자 수
산술	*	곱셈	2
	/	나눗셈	2
	+	덧셈	2
	-	뺄셈	2
	%	나머지	2
논리	!	논리부정	1
	&&	논리 and	2
	||	논리 or	2
관계	>	크다	2
	<	작다	2
	>=	크거나 같다	2
	<=	작거나 같다	2
등가	==	같다	2
	!=	다르다	2
	===	케이스 등가 연산자	2
	!==	케이스 비등가 연산자	2
비트단위	~	비트단위 부정	1
	&	비트단위 and	2
	|	비트단위 or	2
	^	비트단위 xor	2
	^~ or ~^	비트단위 xnor	2

6.4 연산자 형태 (cont)

연사자 형태	연산자 기호	연산자 역할	피연산자 수
축소	& ~& | ~| ^ ^~ or ~^	축소 and 축소 nand 축소 or 축소 nor 축소 xor 축소 xnor	1 1 1 1 1 1
자리 이동	>> <<	오른쪽 자리 이동 왼쪽 자리 이동	2 2
결합 중복	{ } { { } }	결합 중복	아무 수 아무 수
조건	? :	조건	3

■ 산술 연산자

```
A = 4'b0011; B = 4'b0100;          // A, B는 레지스터 벡터
D = 6; E = 4;                      // D, E는 정수
A * B                              // A와 B의 곱셈. 결과값은 4'b1100
D / E                              // D를 E로 나눔. 결과값은 1. 나머지는 버림
A + B                              // A와 B의 덧셈. 결과값은 4'b0111
B - A                              // B에서 A를 뺌. 결과값은 4'b0001
```


6.4 연산자 형태 (cont)

■ 논리 연산자

- ◆ 논리 연산자는 항상 1-비트의 결과를 생성
  - ■ 0은 거짓, 1은 참, x는 결과가 참도 거짓도 아닌경우
- ◆ 피연산자 값이 x또는 z일 때는 x의 값으로 인식

```
// 논리 연산
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
```

6.4 연산자 형태 (cont)

■ 관계 연산자

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B      // 결과값은 거짓(0)
A > B       // 결과값은 참(1)
Y >= X      // 결과값은 참(1)
Y < Z       // 결과값은 x
```

■ 등가 연산자

수식	설명	가능한 논리결과값
a == b	a와 b가 동일, a가 b가 x또는 z값을 가지면 결과는 x	0, 1, x
a != b	a와 b가 다름, a가 b가 x또는 z값을 가지면 결과는 x	0, 1, x
a === b	a와 b가 동일, x나 z값을 포함	0, 1
a !== b	a와 b가 다름, x나 z값을 포함	0, 1

6.4 연산자 형태 (cont)

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B      // 결과값은 논리0
X != Y      // 결과값은 논리1
X == Z      // 결과값은 x
Z === M     // 결과값은 논리1(x와 z를 포함하여 모든 비트를 비교)
Z === N     // 결과값은 논리0(마지막 비트값이 틀리다.)
M !== N     // 결과값은 논리1
```

■ 비트단위 연산자

```
// 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       // 결과값는 4'b10x0
```

6.4 연산자 형태 (cont)

```
// X = 4'b1010. Y = 4'b0000

X | Y        // 비트단위 연산, 결과값 4'b1010
X || Y       // 논리연산, (1 || 0)을 계산, 결과값은 1
```

■ 축소 연산자

```
// 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
```

■ 자리 이동 연산자

```
// X = 4'b1100;
Y = X >> 1;    // Y = 4'b0110, 오른쪽으로 1-비트 자리이동, 최상위 비트부터 0으로 채워짐
Y = X << 1;    // Y = 4'b1000, 왼쪽으로 1-비트 자리이동, 최하위 비트부터 0으로 채워짐
Y = X << 2;    // Y = 4'b0000, 왼쪽으로 2-비트 자리이동
```


6.4 연산자 형태 (cont)

■ 결합 연산자

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y =  B, C                        // 결과값은 4'b0010
Y =  A, B, C, D, 3'b001          // 결과값은 11'b10010110001
Y =  A, B[0], C[1]               // 결과값은 3'b101
```

■ 반복 연산자

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y =  4A              // 결과값은 4'b1111
Y =  4A, 2B          //결과값은 8'b11110000
Y =  4A, 2B, C       // 결과값은 10'b1111000010
```

■ 조건 연산자

```
// tristare 버퍼의 기능적 모델
assign addr_bus = drive_enable ? addr_out : 36'bz;

// 2:1 mux의 기능적 모델
assign out = control ? In1 :  in0;
```

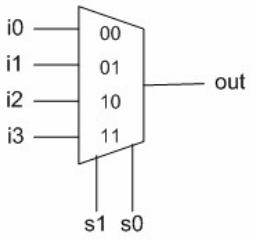
6.4 연산자 형태 (cont)

■ 연산자 우선 순위

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

6.5 예제

■ 4:1 멀티플렉서



◆ 논리식

```
// 데이터 플로우를 이용한 4:1 멀티플렉서
// 조건 연산자와 게이트-수준 모델을 비교해 보자.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0)
// I/O 다이어그램으로부터 포트 선언
output out;
input i0, i1, i2, i3;
input s1, s0;

// 조건 연산자 사용
assign out = s1 ? ( s0 ? i3 : i2 ) : ( s0 ? i1 : i0 );
endmodule
```

6.5 예제 (cont)

◆ 조건 연산자

```
// 데이터 플로우를 이용한 4:1 멀티플렉서
// 조건 연산자와 게이트-수준 모델을 비교
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0)

// I/O 다이어그램으로부터 포트 선언
output out;
input i0, i1, i2, i3;
input s1, s0;

// 조건 연산자 사용
assign out = s1 ? ( s0 ? i3 : i2 ) : ( s0 ? i1 : i0 );
endmodule
```


6.5 예제 (cont)

■ 4-비트 전가산기

◆ 데이터 플로우 연산자

```
// 데이터 플로우 문을 사용한 4-비트 전가산기의 정의
module fulladd4 ( sum, c_out, a, b, c_in );

// I/O 포트 정의
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;

// 전가산기의 기능 기술
assign  c_out, sum  = a + b + c_in ;
endmodule
```

6.5 예제 (cont)

◆ 올림수 예견 전가산기

```
module fulladd4( sum, c_out, a, b, c_in );
// 입력 및 출력
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
// 내부 와이어
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c4, c3, c2, c1;
// 각 단계를 위한 p 계산
assign p0 = a[0] ^ b[0],
p1 = a[1] ^ b[1],
p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];
// 각 단계를 위한 q 계산
assign q0 = a[0] & b[0],
q1 = a[1] & b[1],
q2 = a[2] & b[2],
q3 = a[3] & b[3];
```

```
// 각 단계를 위한 올림수 계산
// 올림수 예견 계산에서 c0은 c_in과 동일
assign c1 = g0 | ( p0 & c_in ),
c2 = g1 | ( p1 & g0 ) | ( p1 & p0 & c_in ),
c3 = g2 | ( p2 & g1 ) | ( p2 & p1 & g0 ) | ( p2 & p1 & p0 & c_in ),
c4 = g3 | ( p3 & g2 ) | ( p3 & p2 & g1 ) | ( p3 & p2 & p1 & g0 ) |
( p3 & p2 & p1 &p0 & c_in );

// sum 계산
assign sum[0] = p0 ^ c_in,
sum[1] = p1 ^ c1,
sum[2] = p2 ^ c2,
sum[3] = p2 ^ c2;

// 올림수 출력
assign c_out = c4;

endmodule
```


6.5 예제 (cont)

■ 리플 카운터

```
// 리플 카운터
module counter ( Q, clock, clear );
// I/O 포트
output [3:0] Q;
input clock, clear;
// T-플립플롭 파생
T_FF tff0( Q[0], clock, clear );
T_FF tff1( Q[1], Q[0], clear );
T_FF tff2( Q[2], Q[1], clear );
T_FF tff3( Q[3], Q[2], clear );
endmodule
```

```
// 모서리-구동 T-플립플롭
module T_FF ( q, clk, clear );
// I/O 포트
output q;
input clk, clear;
// 모서리-구동 D-FF 파생
// q는 피드백, Qbar는 필요하지 않으므로 연결하지 않음
edge_dff ff1( q,, ~q, clk, clear );
endmodule
```

6.5 예제 (cont)

```
// 모서리-구동 D-플립플롭
module edge_dff ( q, qbar, d, clk, clear );
// 입력, 출력
output q, qbar;
input d, clk, clear;
wire s, sbar, r, rbar, cbar;          // 내부 변수

// 데이터 플로우 문
assign cbar = ~clear;                 // clear 신호의 보수를 생성

// 입력 래치; 래치는 준위-구동
// 모서리-구동 플립플롭은 3개의 SR 래치로 구현
assign sbar = ~( rbar & s ),
s = ~( sbar & cbar & ~clk ),
r = ~( rbar & ~clk & s ),
rbar = ~( r & cbar & d );

// 래치 출력
assign q = ~( s & qbar ),
qbar = ~( q & r & cbar );

endmodule
```


6.5 예제 (cont)

```
module stimulus;                      // 스티뮬러스 모듈
// 시뮬레이션 입력 변수를 선언
reg CLOCK, CLEAR;
wire [3:0] Q;
initial
$monitor ($time, “Count Q = %b Clear = %b”, q[3:0], CLEAR);
counter c1( Q, CLOCK, CLEAR );        // 카운터 파생
// Clear 신호로 시뮬레이션
initial begin
CLEAR =1'b1;
#34 CLEAR = 1'b0;
#200 CLEAR = 1'b1;
#50 CLEAR = 1'b0;
end
initial begin
CLOCK = 1'b0;
forever #10 CLOCK = ~CLOCK;           // 10 단위 시간마다 클록 변화
end
initial begin
#400 $finish;                         // 400 단위 시간에 시뮬레이션 종료
end
endmodule
```

7. Behavioral Level Modeling

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

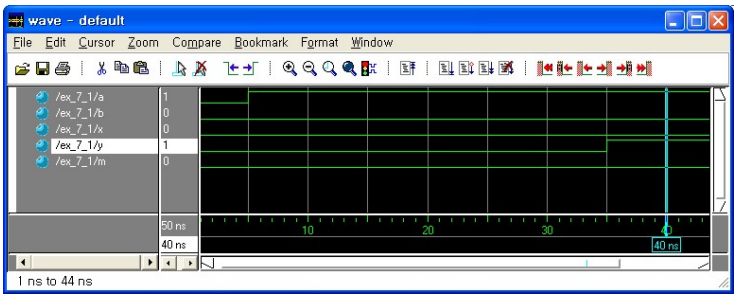
Digital Design & Test Lab.
Kwangwoon University

Contents

7.1 구조적 프로시저

■ Initial 문

- ◆ 시간 0 에서 시작
- ◆ 시뮬레이션동안 한번만 수행
- ◆ 여러 개의 문장 첨가시 begin, end 사용

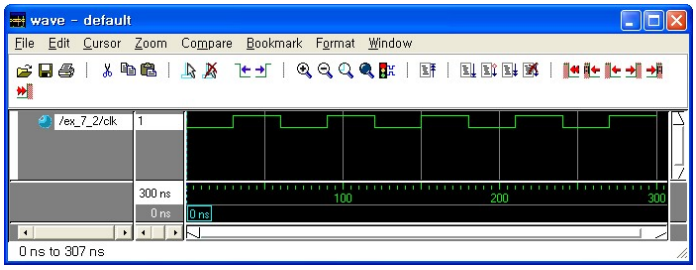



```
module ex_7_1;
reg x, y, a, b, m;
initial
Begin
 m = 1'b0;
  x = 1'b0; y = 1'b0;
  a = 1'b0; b = 1'b0;
end
initial
begin
  #5  a = 1'b1;  #25 b = 1'b0;
end
initial
begin
  #10 x = 1'b0; #25 y = 1'b1;
end
initial
  #50 $stop;
endmodule
```


7.1 구조적 프로시저

■ Always 문

- ◆ vhdl 의 process 문
- ◆ always 문간, 병렬동작
- ◆ 필요시 sensitibity 포함




```
module ex_7_2;

reg clk;

initial clk = 1'b0;
always
  #30 clk = ~clk;

initial
  #300 $stop;
endmodule
```

7.2 절차적 할당

- 할당되는 값이 변하기 전까지 계속 동일한값 유지
- 문법

<변수명> (<)= <할당변수명>

- 블록킹 문장
  - 할당연산자 '=' 사용
  - Begin, end 사이의 순차적 할당
    - 시뮬레이션 시간값의 누적
- 논 블록킹 문장
  - 할당연산자 '<=' 사용
  - Begin, end 사이의 병렬적 할당
    - Intra assingment delay control 과 사용시 시간값의 누적이 없음
  - 하나의 변수에 대한 assingment 에서는 의미 없음

7.2 절차적 할당

- 블록킹 문장과 논 블록킹 문장, 혼합된 문장의 비교

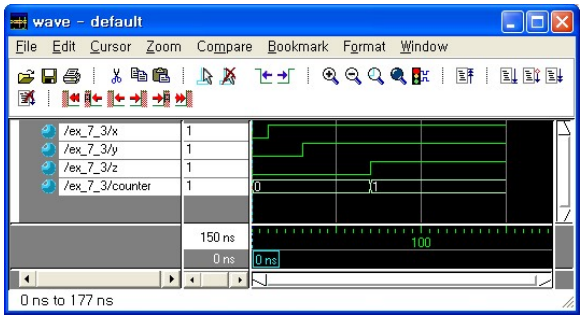
```
module ex_7_3;
reg x, y, z;
integer counter;
initial
begin
  x = 1'b0; y = 1'b0; z = 1'b0;
  counter = 0;
   #10 x = 1'b1;
   #20 y = 1'b1;
   #40 z = 1'b1;
  counter = counter + 1;
end
initial
  #150 $stop;
endmodule
```

```
module ex_7_3;
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
```

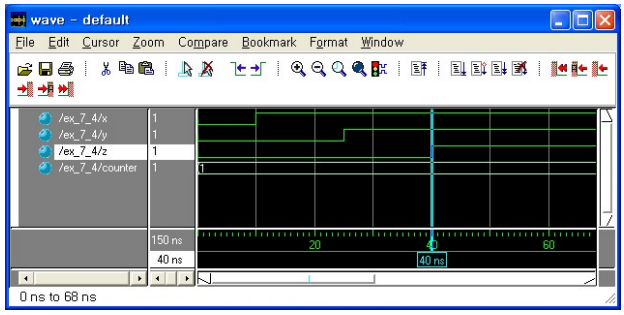
```
module ex_7_3;
reg x, y, z;
integer counter;
initial
begin
  x = 1'b0; y = 1'b0; z = 1'b0;
  counter = 0;
   x <= #10 1'b1;
  y = #25 1'b1;
  z = #40 1'b1;
  counter <= counter + 1;
end
initial
  #150 $stop;
endmodule
```

7.2 절차적 할당

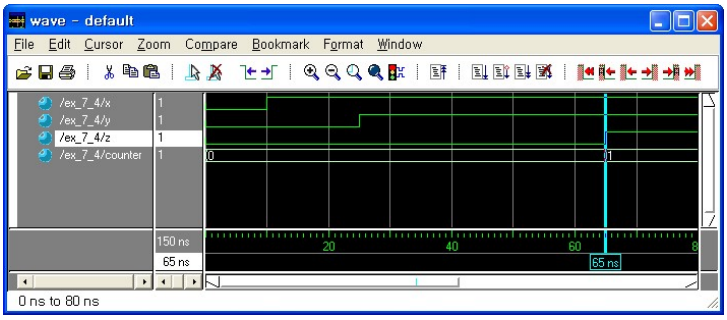
■ 시뮬레이션 결과



Block 문장



Non_blodk 문장



혼합 문장

7.3 타이밍 제어

■ 지연 기반 타이밍 제어**(Delay-based timing control)**

◆ 정규지연제어(regular delay control)

■ 언제나 시뮬레이션 시간을 누적

```
#10 x = 1'b1
```

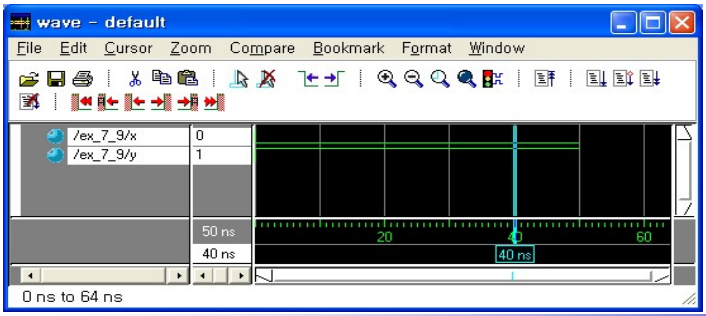
◆ 내부 할당 지연제어(intra-assignment delay control)

■ 논블럭 문장과 사용시 begin, end 사이의 시뮬레이션 시간 누적 없음

```
x <= #10 1'b1
```

◆ 제로 지연 제어(zero delay control)

■ 제일 마지막으로 시뮬레이션이 됨




```
initial
begin
  x = 1'b1; y = 1'b0;
end
initial
begin
  #0 x = 1'b0; #0 y = 1'b1;
end
```

7.3 타이밍 제어 (cont)

■ 사건 기반 타이밍 제어(Event-based timing control)

◆ 정규사건 제어(regular event contol)

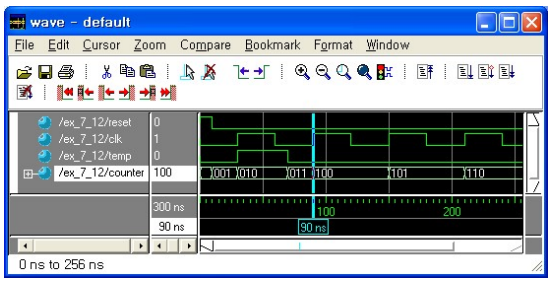
■ @( ) 안의 변수명에 의한 제어

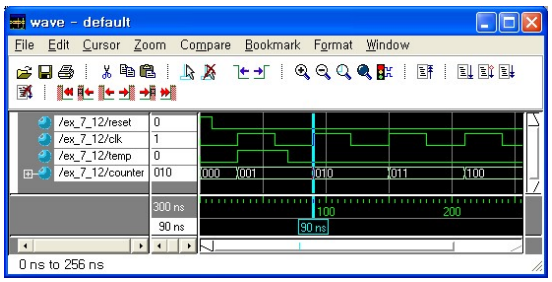
◆ 사건 OR 제어(event OR control)

■ Sensitiviti list 에 의한 제어

```
Always @(reset or posedge clk or temp)
 If(reset)
  counter <= 3'b000;
 else counter <= counter + 3'b001;
```

```
Always @(reset or posedge clk or temp)
 If(reset)
   counter <= 3'b000;
 else if(clk)
                ....
   counter <= counter + 3'b001;
```





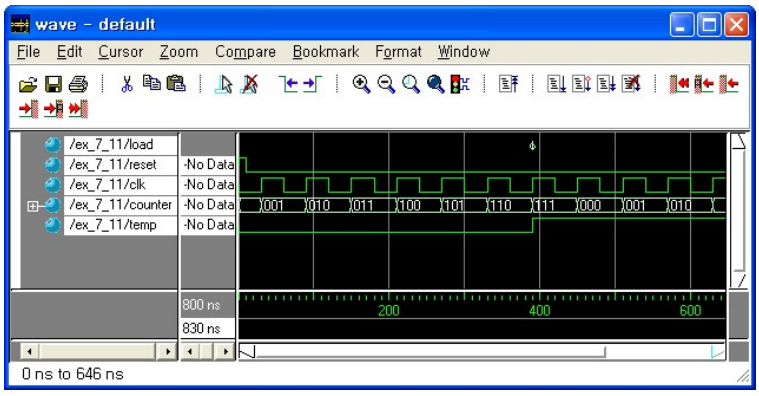
7.3 타이밍 제어 (cont)

◆ 명명된 사건 제어(named event conrtol)

■ Event 문에 의해 명명된 변수의 변화에 의한 제어

■ 준위 기반 타이밍 제어(Level-sensitive timing control)

◆ Wait 문의 사용




```
event load;
      .
initial
begin
  reset <= 1'b1;
  clk   <= 1'b0;
  temp  <= 1'b0;
end
always @(counter)
  if(counter == 3'b111)
    -> load;
always @(load)//= wait (load)
  temp <= 1'b1;
```

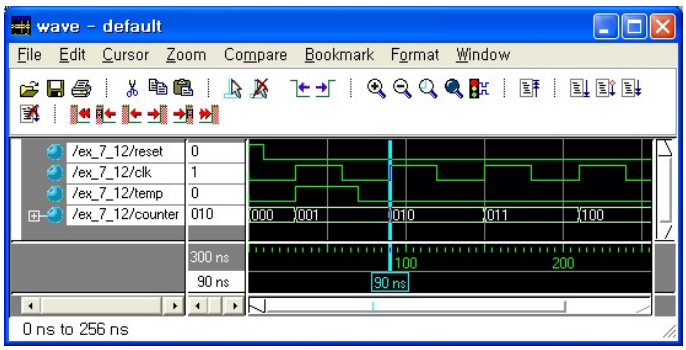
7.4 조건문

- **If, else 문 사용**
  - If, else 의 종속적 중첩사용 가능
  - Clk 를 사용하는 sequential 회로일경우, 항상 else if(clk) 다음에 조건문 기술

```
always @(reset or posedge clk or load)
 if(reset)
   counter <= 3'b000;
 else if(clk)
   if(load)
     counter <= counter + 3'b001;
   else counter <= counter – 3'b001;
```


7.5 다중 분기

- **case 문**
  - vhdl 의 when others 대신 default 문 기술
- **casez 문**
  - 'z' 값을 don't care 로 처리
- **casex 문**
  - 'x' 와 'z' 값을 don't care 로 처리



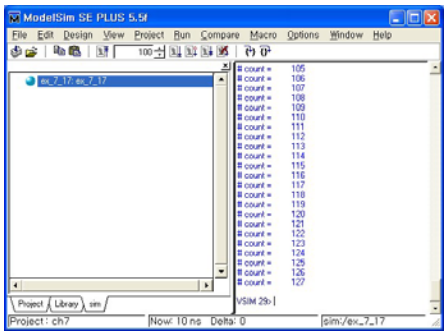

```
initial begin
  #30  d_in <= 8'b10000000;
  #30  d_in <= 8'b01000000;
  #30  d_in <= 8'b00100000;
                .
                .
end

always @(d_in)
  case(d_in)
    8'b10000000 : d_out <= 3'b111;
    8'b01000000 : d_out <= 3'b110;
    8'b00100000 : d_out <= 3'b101;
                .
                .
default      : d_out <= 3'b000;
endcase
```

7.6 루프

■ 루프

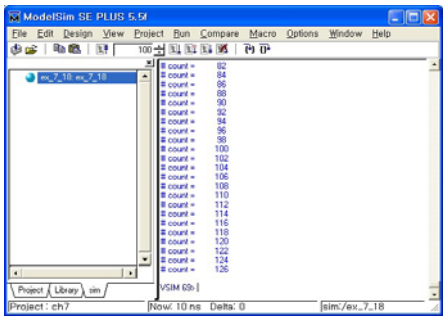
◆ While loop : while 조건이 참일때 까지 반복수행

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



◆ For loop : 정해진 조건대로 반복수행

```
integer count :=0;
initial begin
for(count=0; count<128; count =
count + 2)
  $display("Count = %d", count);
end
```



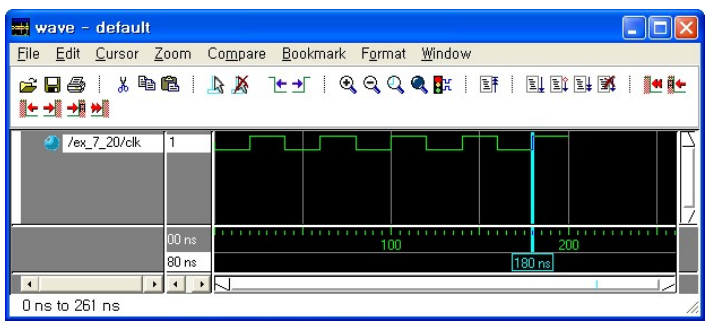
7.6 루프 (cont)

◆ Repeat loop : 상수에 의해 정해진 횟수만큼 반복 수행

```
parameter count = 128;
initial begin
repeat(count)
    $display("Count = %d", count);
    count = count + 1;
  end
end
```

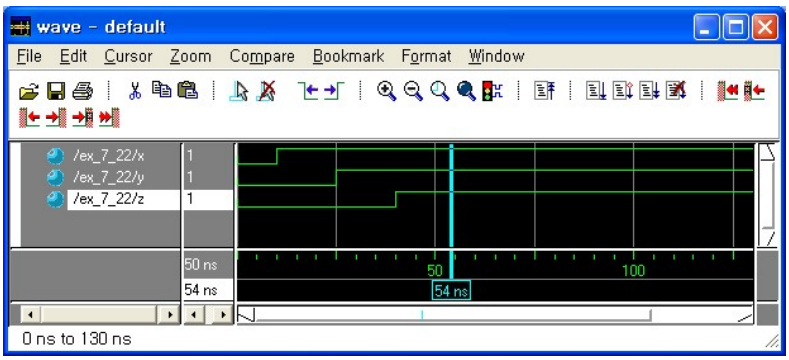
◆ Forever loop : $finish 를 만날때까지 반복수행

```
parameter clk_period = 20;
parameter end_time = 100;
Initial begin
  clk <= 1'b0;
  # clk_period clk <= ~clk;
end
initial
  # end_time $finish(stop);end
```



7.6 순차 처리와 병렬 처리 블록

- 블록형태
  - 순차처리 블록
    - Initial(begin, end) 안의 블록
  - 병렬처리 블록
    - fork, join 사이의 블록

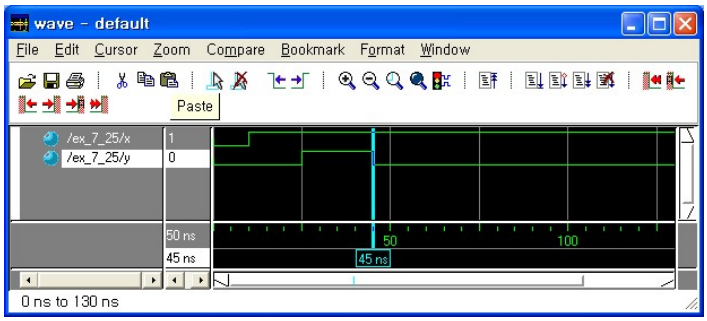



```
initial
begin
  x = 1'b0; y = 1'b0; z = 1'b0;
end
Initial begin
  fork
  x = #10 1'b1;
  y = #25 1'b1;
  z = #40 1'b1;
  join
end
initial
  #150 $stop;
endmodule
```


7.6 순차 처리와 병렬 처리 블록 (cont)

- 특별한 형태의 블록
  - 중첩된 블록 : 순차적 블록 + 병렬적 블럭

```
Initial begin
  fork
  x = #10 1'b1;  y = #25 1'b1;
Join
  y = #10 1'b0;
end
```



  - 명명된 블럭및 무효화

```
Initial begin : block1:
  fork
  x = #10 1'b1;  y = #25 1'b1;
 Join
  #30 $disable block1
  z = #40 1'b1;
end
```

8. Task and Function

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

8.1 태스크와 함수의 차이점

- **Verilog**는 **task**와 **function**을 이용해 커다란 행위 수준 설계를 작은 단위로 나눈다.

- 태스크와 함수의 차이점

Function	Task
다른 함수를 사용할 수 있지만 다른 태스크는 사용할 수 없다.	다른 태스크나 함수를 사용할 수 있다.
항상 시뮬레이션 시간 **0**에 수행된다.	**Non-zero** 시뮬레이션 시간에 수행될 수 있다.
어떤 지연, 사건, 또는 타이밍 제어문장을 포함할 수 없다.	지연, 사건, 또는 타이밍 제어 문장을 포함할 수 있다.
적어도 하나 이상의 **input** 인수를 가져야만 한다.	**Input, output** 또는 **inout**를 하나도 가지지 않거나 다수를 가질 수 있다.
항상 하나의 값을 되돌린다. **output** 또는 **inout** 인수를 가질 수 없다.	값을 되돌릴 수 없지만 **output**와 **input**을 통해서 여러개의 값을 전달할 수 있다.

8.1 태스크와 함수의 차이점

- 태스크와 함수의 공통점
  - 와이어를 가질 수 없음
  - 행위 수준 문장만 포함
  - always 또는 initial 문장을 포함하는 것이 아니며 always 블록, initial 블록 또는 다른 태스크와 함수로부터 호출

8.2 태스크

■ 8.2.1 태스크의 선언과 호출

```
// 태스크 선언 문법
<task>
        ::= task <name_of_task>;
            <tf_declaration>*
            <statement_or_null>
        endtask
<name_of_task>
        ::= <IDENTIFIER>
<tf_declaration>
        ::= <parameter_declaration>
        ||= <input[output, inout]_declaration>
        ||= <reg_declaration>
        ||= <time_declaration>
        ||= <integer[real]_declaration>
        ||= <event_declaration>
// 태스크 호출 문법
<task_enable>
        ::= <name_of_task>
        ||= <name_of_task> (<expression><,<expression>>*);
```


8.2 태스크 (cont)

■ 8.2.2 태스크 예제 – 입 출력 인수 사용

```
module operation;
……..
parameter delay=10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;
always @(A or B)
begin
    bitwise_opr(AB_AND, AB_OR, AB_XOR, A, B);
end
.....
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor;
input [15:0] a, b
begin
    #delay ab_and = a & b;
    ab_or = a | b
    ab_xor = a ^ b;
end
endtask
….
andmodule
```

태스크 bitwise_opr을 포함하는 opration이라는 모듈을 정의

A, B의 값이 바뀔때 마다

태스크 bitwise_opr을 호찰

2개의 입력 인자(A,B)를 제공하고 3개의 출력인자(AB_AND, AB_OR, AB_XOR)을 받는다

인자는 태스크 선언과 같은 순서로 지정

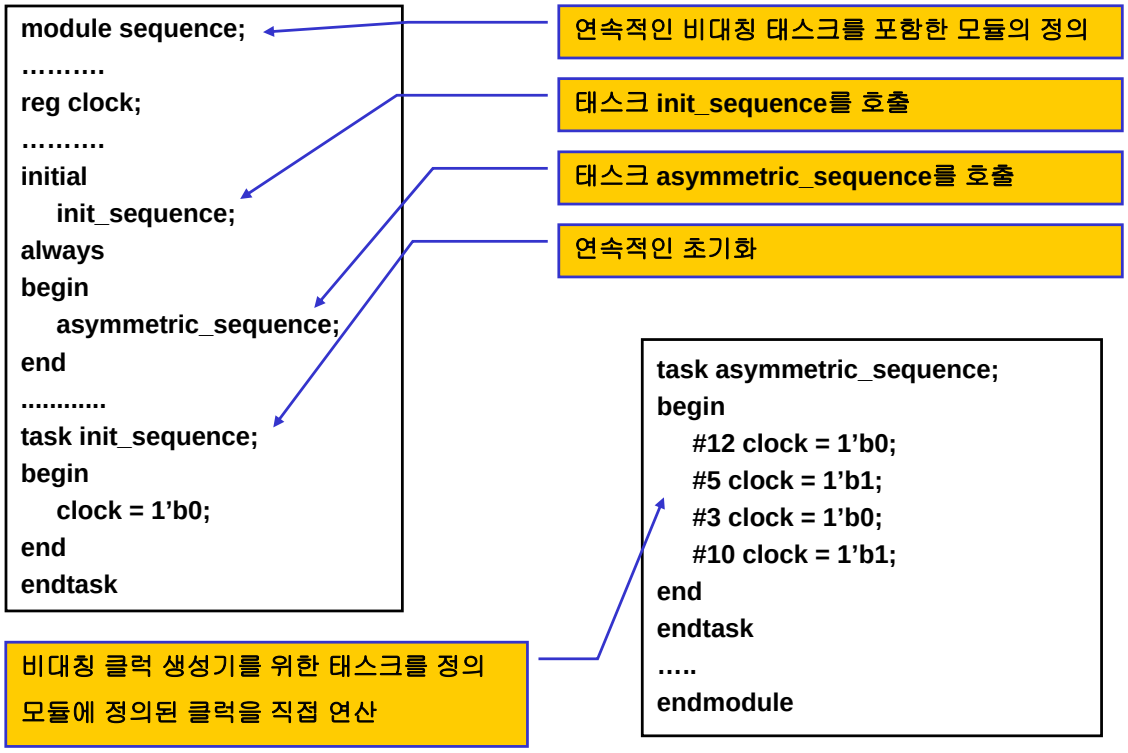
태스크 bitwise_opr의 정의

태스크 bitwise_opr의 출력

태스크 bitwise_opr의 입력

8.2 태스크 (cont)

■ 8.2.2 태스크 예제 – 비대칭 클럭 생성기



8.3 함수

■ 8.3.1 함수 선언과 호출

```
<task> // 함수 선언 문법
        ::= function <range_or_type>? <name_of_function>;
            <tf_declaration>+
            <statement>
        endfunction
<range_or_type>
        ::= <range>
        ||= <INTEGER>
        ||= <REAL>
<name_of_function>
        ::= <INDENTIFIER>
<tf_declaration>
        ::= <parameter_declaration>
        ||= <input_declaration>
        ||= <reg_declaration>
        ||= <time_declaration>
        ||= <integer_declaration>
<function_call>  // 태스크 호출 문법
        ::= <name_of_function> (<expression><,<expression>>*)
```

8.3 함수 (cont)

■ **8.3.2** 함수 예제 – 페리티 계산

```
module parity;
……..
reg [31:0] addr;
reg parity;

always @(addr)
begin
    parity = calc_parity(addr);
    $display("Parity calculated = %b", calc_parity(addr));
end
……..

function calc_parity;
input [31:0] address;
begin
    clac_parity = ^address;
end
endfunction
……..
endmodule
```

함수 calc_parity를 포함한 모듈을 정의

주소값이 변할 때마다 새로운 패리티를 계산

패리티 계산 함수 정의

알맞은 출력값을 놓는다. 암시적 내부 레지스터 calc_parity를 사용한다.

모든 주소 비트의 xor를 되돌린다.

8.3 함수 (cont)

■ **8.3.2** 함수 예제 – 좌측/우측 자리 이동

```
module shifter;
……..
‘define LEFT_SHIFT   1’b0
‘define RIGHT_SHIFT 1’b1
reg [31:0] addr, left_addr, right_addr;
reg control;
always @(addr)
begin
    left_addr = shift(addr, ‘LEFT_SHIFT);
    right_addr = shift(addr, ‘RIGHT_SHIFT);
end
……..
function shift;
input [31:0] address;
input control;
begin
   shift=(control==‘LEFT_SHIFT)?(address<<1):address>>1);
end
endfunction
……..
endmodule
```

이동함수를 포함한 모듈을 정의

좌측/우측자리 이동기

새로운 주소값이 올때마다 좌측/우측 자리 이동값이 계산

자리 이동 함수를 저으이. 출력값은 32비트값

제어 신호에 의해 출력값이 결정

9. Useful Modeling Technique

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

9.1 절차적 연속 할당

- 연속적 할당(6장)
  - Assign 문 : net의 값을 변환
- 절차적 할당(7장)
  - Register의 값을 변환
- 절차적 연속 할당
  - Register의 값을 일정시간 동안 계속 변화시킴

9.1 절차적 연속 할당 (cont)

- 9.1.1 assign과 deassign
  - 할당문의 왼쪽에는 레지스터만이 올 수 있음

```
module edge_dff(q,qbar,d,clk,reset);
  output  q, qbar;
  input d, clk, reset;
  reg q, qbar;
always@(negedge clk)
begin
  q = d;
  qbar = ~d;
end
always@(reset)
  if(reset) begin
     assign q = 1'b0;
     assign qbar = 1'b1;
  end
  else begin
     deassign q;
     deassign qbar;
  end
endmodule
```

reset이 하이상태 때마다 q와 qbar의 할당을 절차적 연속 할당을 사용하여 덮어씀

reset = 1(참)일때 절차적 연속 할당으로 기존의 할당을 새로운 값으로 덮어씀

reset = 0(거짓) 이면 deassign을 이용하여 덮어쓰인 값을 지운다. 그런 다음 q=d, qber=~d라는 기존의 할당은 다음 클럭의 하강 모서리에서 레지스터값을 바꿀 수 있다.

9.1 절차적 연속 할당 (cont)

■ 9.1.2 force와 release

- ◆ 레지스터나 넷에 새로운 할당을 덮어씌우는데 사용
- ◆ 설계블록에서는 사용하지 않고 시티뮬러스나 디버그 문에서 사용

```
module stimulus;
...
edge_dff dff(Q, Qbar, D, clk, RESET);
...
initial begin
  #50 force dff.q = q'b1;
  #50 release dff.q;
end
endmodule
```

edge_dff의 결과와 상관없이 50에서 100타임 사이에 dff.q에 1의 값을 할당한다.

-50타임에 q = 1 할당

-100타임에 q의 값을 해제

```
module top;
...
assign out = a & b & c;
...
initial begin
  #50 force out = a | b & c;
  #50 release out;
end
endmodule
```

50 ~ 100타임에는 assign의 할당과는 상관없이 out에 a | b & c 가 할당

9.2 파라미터 오버라이드

■ 9.2.1 defparam

- ◆ 설계 블록의 어떤 모듈 인스턴스 내의 파라미터 값을 변화

```
module hello_world;
// 모듈의 식별 숫자를 0으로 정의
parameter id_num = 0;
initial
  $display ("Display hello_world id number = %d, id_num);
endmodule

module top;
  defparam w1.id_num = 1, w2.id_num = 2;
// 두개의 hello_world 모듈 파생
  hello_world w1( );
  hello_world w2( );
endmodule
```

결과
Display hello_world id number = 1
Display hello_world id number = 2

defparam문을 이용하여 파생된 모듈 내의 파라미터값을 변화시킨다.

9.2 파라미터 오버라이드 (cont)

■ 9.2.2 모듈 인스턴스 파라미터 값

◆ 파라미터 값은 모듈이 파생될 때 파라미터 값의 변화 가능

```
module top;
// 두개의 hello_world 모듈 파생
  hello_world #(1)w1;   // 모듈 w1에 값 1을 넘긴다.
  hello_world #(2)w2;   // 모듈 w2에 값 2를 넘긴다.
endmodule
```

◆ 여러개의 파라미터가 정의되면 모듈 파생시 모듈내의 파라미터의 순서와 같은 순서로 새 값을 지정

```
module bus_master;
  parameter delay1 = 2;
  parameter delay1 = 3;
  parameter delay1 = 7;
endmodule

module top;  // 두개의 bus_master 모듈을 파생
bus_master#(4, 5, 6) b1( );  // b1 : delay1 = 4, delay2 = 5, delay3 = 6
bus_master#(9, 4) b2( );     // b2 : delay1 = 9, delay2 = 4, delay3 = 7
endmodule
```


9.3 조건 컴파일 실행

■ 조건 컴파일

◆ 특정 플래그가 설정되었을 경우에만 verilog 코드의 특정 부분이 컴파일되게 지정 가능

◆ 컴파일 지시자 ‘ifdef, ‘else, ‘endif 사용

```
//예제 1
‘ifdef TEST              // 텍스트 매크로 TEST가 정의되었을때만 test모듈을 컴파일
module test;
…
endmodule
‘else                    // stimulus문을 default로 컴파일
moduld stimulus;
…
endmodule
‘endif                   // ‘ifdef문 완료
//예제 2
module top;
bus_master b1( );     // 모듈을 무조건적으로 컴파일
  ‘ifdef ADD_B2
    bus_master b(2);  // 텍스트 매크로 ADD_B2가 정의되었을때만 조건부로 b2가 파생
  ‘endif
endmodule
```

9.4 시간 척도

- ‘timescale로 모듈의 참조 시간 단위를 지정
- 사용법
  - ‘timescale<참조시간 단위>/<시간 정밀도>
  - 참조시간 단위 : 시간과 지연의 측정단위
  - 시간 정밀도 : 시뮬레이션 동안 반올림된 지연의 정확도

```
‘timescale 100ns / 1ns    // 참조시간 단위는 100ns , 정말도는 1ns
…
always #5                 // 5단위 시간(500ns)마다 레지스트를 바꿈
```


9.5 유용한 시스템 태스크 (cont)

- 9.5.1 파일 출력

9.5 유용한 시스템 태스크 (cont)

■ 9.5.2 계층 구조의 표시

◆ $display, $write, $monitor, $strobe와 같은 출력 태스크의 옵션 %m으로 출력

```
module M;
…
initial
  $display("Displaying in %m);
endmodule

//모든 M 파생
module top;
…
M m1( );
M m2( );
M m3( );
endmodule
```

```
//시뮬레이션 결과
Displaying in top.m1
Displaying in top.m1
Displaying in top.m1
```


9.5 유용한 시스템 태스크 (cont)

■ 9.5.3 스트로밍

◆ 시간 단위에 데이터가 변하는 모든 할당문의 수행이 완료된 후에만 데어터가 출력되는 동기화 기술을 제공

```
always @(posedge clock)
begin
  a = b;
  c = d;
end

always @(posedge clock)
  $strobe("Displaying a = %b, c = %b", a, c);
```

a = b 와 c = d가 수행된 후 클럭의 상승 모서리에서 수행

9.5 유용한 시스템 태스크 (cont)

■ 9.5.4 난수 발생

- ◆ $random 태스크 사용
- ◆ 사용법 : $random(<seed>)

```
module test
…
Rom rom1(data, addr)
initial
  r_seed = 2 ;                    // 임의로 seed를 2로 정의
always@(posedge clock)
  addr = $random(r_seed);         //  난수 발생
…
endmodule
```


9.5 유용한 시스템 태스크 (cont)

■ 9.5.5 파일을 이용한 메모리 초기화

- ◆ $readmemb(2진수 포맷), $readmemhexa(16진수 포맷)지원
- ◆ 사용법 : $readmemb("파일명" , 메모리명, 시작주소, 마지막 주소)
  - ■ 파일명과 메모리명은 필수, 시작주소와 마지막 주소는 선택

```
module test;
reg[7:0] memory[7:0];              // 8바이트 메모리 선언
integer i;
initial
begin
  $readmemb("init.dat", memory);   // 메모리 파일 init.dat를 읽어서 초기화
  for(i=0; i<8; i=i+1)
    $display("Memory [%0d]=%d", i, memory[i]);
end
endmodule
```

9.5 유용한 시스템 태스크 (cont)

- **값 변환 덤프 파일(Value change dump : VCD)**
  - ◆ 시뮬레이션 시간, 범위와 신호 정의, 및 신호값의 변화에 대한 정보를 포함하는 ASCII 파일

```
initial
  $dumpfile("myfile.dmp");
initial
  $dumpvars; //선택 신호가 없을때는 모든 신호를 덤프
initial
  $dumpvars(1, top);
initial
  $dumpvars(2, top, m1); //top.m1 아래의 2수준 계층
initial
  $dumpvars(0, top, m1);
initial
begin
  $dumpon;              // 덤프를 시작
  #100 $dumpoff;  // 100 단위 시간 후 덤프 종료
end
initial
  $dumpall;
```

모듈 인스턴스 top내의 변수를 덤프 숫자 1은 계층 수준을 나타내며 top 아래 한계층을 덤프.(top내의 변수를 덤프, top에 의해 파생된 모듈 내의 신호는 덤프하지 않음)

0은 top.m1 아래의 전체 계층을 덤프

체크 포인트
모든 VCD 변수의 현재 값을 덤프.

10. Timing and Delay

Young-Ho Seo

http://explore.kw.ac.kr/~axl

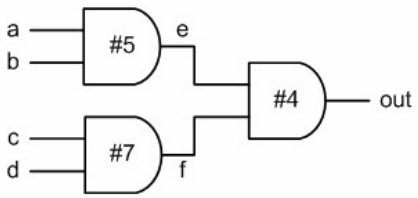
design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

10.1 지연 모델의 유형

- 분산지연
  - 회로의 게이트를 기반으로 설명
  - 분산지연은 각 게이트들에게 주어진 지연값이나, assign문에 의한 지연값을 이용하여 모델링

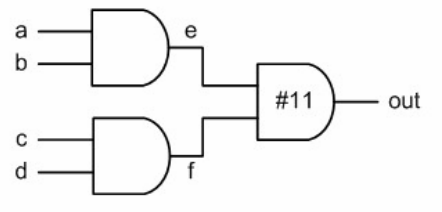



```
module M(out, a, b, c, d);
  output  out;
  input a, b, c, d;
    wire e, f;

  and #5 a1(e, a, b);
  and #7 a2(f, c, d);
  and #4 a3(out, e, f);
endmodule
```

```
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
```

10.1 지연 모델의 유형 (cont)

- 집중지연
  - 회로의 모델을 기반
  - 모듈의 출력 게이트에 지연값을 지정하여 기술
  - 모든 경로의 누적된 지연값이 한 곳에 집중

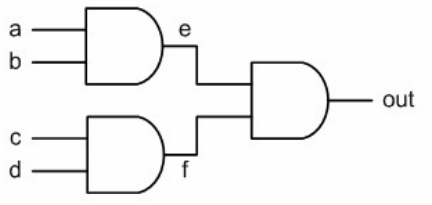



```
module M(out, a, b, c, d);
  output  out;
  input a, b, c, d;
    wire e, f;

  and a1(e, a, b);
  and a2(f, c, d);
  and #11 a3(out, e, f);
endmodule
```

입력에서부터 출력까지의 최대 지연값을 계산 한다. (7+4=11)

10.1 지연 모델의 유형 (cont)

- 핀-대-핀 지연(경로지연 – path delay)
  - 지연값은 각각의 입력에서 출력까지의 경로에 지정
  - 지연값은 각 입력/출력 경로마다 지정




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

10.2 경로 지연 모델링

- **Specify 블록**
  - 경로 지연을 키워드 specify와 endspecify 안에서 선언
  - 모듈에 존재하는 경로의 핀-대-핀 타이밍 지연을 지정
  - 회로에서 확인할 타이밍을 지연
  - specparam 상수를 정의

```
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
```


10.2 경로 지연 모델링 (cont)

- **Specify 블록의 구조**
  - 병렬연결
    - 기호 “=>”
    - (<출발지> => <목적지>) = <지연값>;
    - 출발지와 목적지가 벡터이면, 같은 비트 수
  - 완전연결
    - 기호 “*>”
    - (<출발지> *> <목적지>) = <지연값>;
    - 출발지의 각 비트는 목적지의 모든 비트와 연결
    - 출발지와 목적지가 벡터이면, 같은 비트 수를 가질 필요는 없음

```
(a => out) = 9;   //비트-대-비트
(a => out) = 9;   //벡터 연결(4 bit)

(a[0] => out) = 9;
(a[1] => out) = 9;
(a[2] => out) = 9;
(a[3] => out) = 9;
```

```
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;
  endsoecufy

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

10.2 경로 지연 모델링 (cont)

- specparam 문
  - Special 파라미터는 specify 블록 안에 사용하기 위해 선언
  - 키워드 specparam

```
specify
  specparam d_to_q =9;
  specparam cli_to_q = 11;

  (d => q) = d_to_q;
  (clk => q) = clk_to_q;
endsoecufy
```


10.2 경로 지연 모델링 (cont)

- 조건 경로 지연 (상태 의존 경로 지연 – state dependent path delay : SDPD)
  - if 문을 사용, else 문은 사용할 수 없음

```
module M(out, a, b, c, d);
  output  out;
  input a, b, c, d;
  wire e, f;
  specify
    if (a) (a => out ) =9;
    if (~a) (a => out) = 10;

    if (b & c) (b => out ) =9;
    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
  and a1(e, a, b);
  and a2(f, c, d);
  and a3(out, e, f);
endmodule
```

신호 a의 상태에 따른 다른 핀-대-핀 타이밍

b, c 신호에 따른 조건문.
b & c 가 참이면, 지연은 9, 아니면 13

연결 연산자를 사용. 완전 연결

10.2 경로 지연 모델링 (cont)

◆ 상승, 하강, 턴-오프 지연

```
specparm t_delay = 11;
  (clk => q) = t_delay;

specparm t_rise = 9, t_fall = 13;
  (clk => q) = (t_rise, t_fall);

specparm t_rise = 9, t_fall = 13, t_turnoff =11;
  (clk => q) = (t_rise, t_fall, r_turnoff);

specparm t_01 = 9, t_10 = 13, t_0z = 11;
specparm  t_z1 = 9, t_1z = 11, t_z0 =13;
  (clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);

specparm t_01 = 9, t_10 =13, t_0z = 11;
specparm t_z1 = 9, t_1z =11, t_z0 = 13;
specparm t_0x = 4, t_x1 =13, t_1z = 5;
specparm t_x0 = 9, t_xz =11, t_zx = 7;
  (clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0,
          t_0x, t_x1, t_1x, t_x0, t_xz, t_zx);
```

한 개의 지연값 표현

두개의 지연값 표현, 상승과 하강
상승은 0->1 ,0->z, z->1로의 변화
하강은 1->0, 1->z, z->0으로의 변화

상승, 하강, 턴-오프의 지연값 표현
상승은 0->1 , z->1로의 변화
하강은 1->0, z->0 로의 변화
턴-오프는 0->z, 1->z 로의 변화

6개의 지연값을 표현
0->1 , 1->0, 0->z, z->1,1->z, z->0 로의 변화

12개의 지연값을 표현
0->1 , 1->0, 0->z, z->1,1->z, z->0, 0->x, x->1, 1->x, x->0, x->z, z->x 로의 변화

10.2 경로 지연 모델링 (cont)

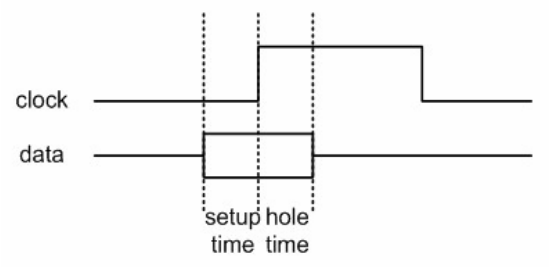
◆ 최소, 최대, 전형적 지연

- 최소 : 전형적 : 최대 (min : typ : max)
- +mindelays : 최소 지연값
- +typdelays : 전형적 지연값
- +maxdelays : 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);
```

10.3 타이밍 검사

- **$setup, $hold 검사**
  - specify블록 안에 존재
  - 클럭이 상승하기 전에 데이터가 도착
  - 클럭이 상승 후에 데이터가 변하지 않는 값




```
specicy
  $setup(data, posedge clock, 5);
endspecify
```

- **$setup 태스크**
  - 사용법 : **$setup(data_event, refrence_event, limit);**
    - **data_event :** 위배를 확인하기 위한 신호
    - **refrence_event : data_event** 신호를 확인하기 위한 레퍼런스를 만드는 신호
    - **limit :** 데이터의 **setup**을 위한 최소 시간
    - **(refernce_event – data_event) < limit** 이면, 오류발생

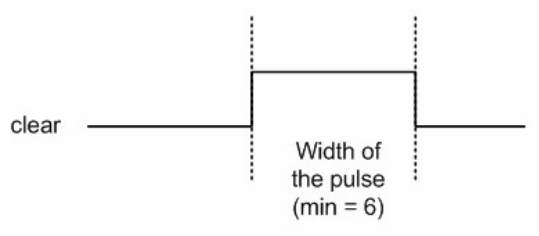
10.3 타이밍 검사 (cont)

- **$hold 태스크**
  - 사용법 : **$hold(refrence_event, data_event, limit);**
    - **refrence_event : data_event** 신호를 확인하기 위한 레퍼런스를 만드는 신호
    - **data_event :** 위배를 확인하기 위한 신호
    - **limit :** 데이터의 **hold**를 위한 최소 시간
    - **(data_event - refernce_event) < limit** 이면, 오류발생

```
specicy
  $hold(posedge clock, data, 5);
endspecify
```

10.3 타이밍 검사 (cont)

- **$width 검사**

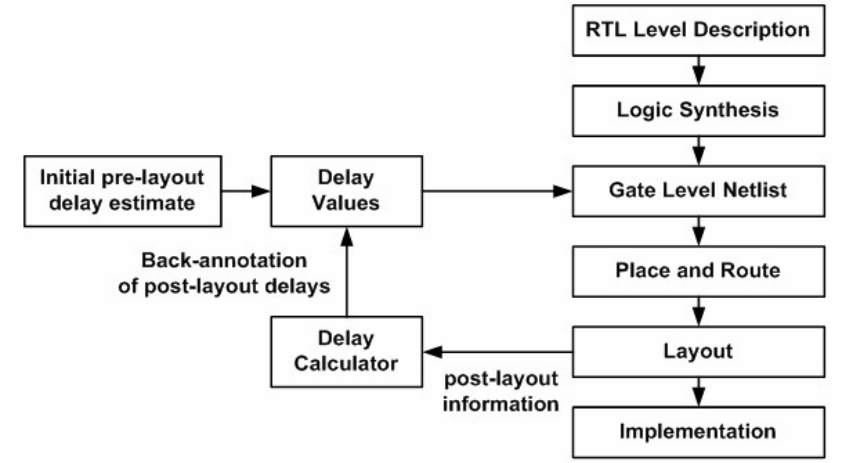



```
specicy
  $width(posedge clock, 6);
endspecify
```

- **$width 태스크**
  - 사용법 : **$width(refrence_event, limit);**
    - **refrence_event :** 모서리-구동 변화(신호의 모서리 변화)
    - **limit :** 펄스의 최소 넓이
    - **data_event** 는 **$width**에서 정확한 값으로 나타나지 않고, **refrence_event** 신호의 반대되는 신호 변화에 의해서 발생
    - **(data_event - refernce_event) < limit** 이면, 오류발생

10.4 지연 백-어노테이션

- **Back-Annotation**



- Postlayout 지연값은 게이트-수준 넷-리스트의 에측된 지연값을 수정하기 위하여 백-어노테이션을 수행

12. User Defined Primitive

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Contents

12.1 UDP 기본

- 프리미티브(Primitive)
  - 내장 (built-in)프리미티브
    - And, nand, or, nor, not..
  - 사용자 정의(user-defined) 프리미티브
    - 조합형(combination) UDP
    - 순차형(sequential) UDP

```
//UDP name and terminal list
primitive udp_name (
<output_terminal_name>,<input_terminal_names>); //출력은 오직 1개
//terminal declarations
output <output_terminal_name>;
input <input_terminal_names>;
//오직 순차 UDP 일때
reg <output_terminal_name>;
initial <output_terminal_name> = <value>;
//UDP state table
table
<table entries>
end table
//end of UDP define
endprimitive
```


12.1 UDP 기본 (cont)

- UDP 규칙
  - 입력 터미널의 크기는 1비트, 여러 개의 입력터미널 선언가능
  - 출력 터미널의 크기는 1비트, 제일 앞부분에서 선언
  - 입력 터미널은 input, 출력터미널은 output, reg(순차UDP일 경우)로 선언

```
primitive udp_example(out1, in1, in2);
Input in1, in2;
output out1;
                ……..
```

  - 상태 테이블이 가질 수 있는 값은 1, 0, x
    - UDP 는 z 를 가질 수 없으며 입력되는 z 는 x 로 변환
  - 파생시켜서 사용

```
module udp_use(c, a, b);
input a, b;
output c;
(wire c;)     //내부에서는 반드시 wire 사용
udp_example(c, a, b);
endmodule
```

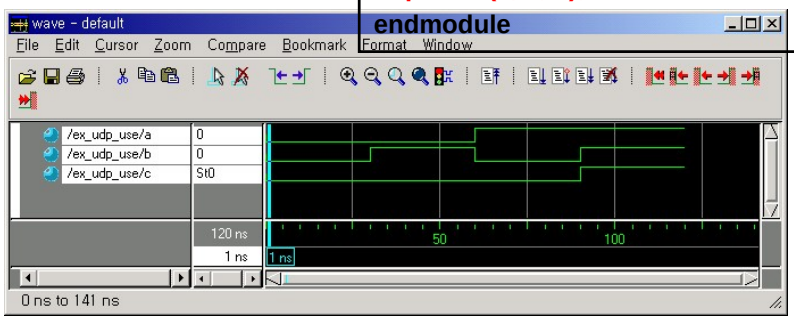
  - bi-direction port 를 지원하지 않는다.

12.1 UDP 기본 (cont)

■ UDP 사용 예제

```
primitive udp_and (out, in1, in2);
input in1, in2;
output out;
table
//in1  in2   out
  0    0    : 0;
  0    1    : 0;
  1    0    : 0;
  1    1    : 1;
endtable
endprimitive
```

```
module ex_udp_use;
reg a, b;  //input
wire c;    //ouput    //UDP out : 반드시 wire 사용
initial begin
  a <= 1'b0;    b <= 1'b0;
end
initial
  #60 a <= 1'b1;
initial begin
  #30 b <= 1'bZ; #30 b <= 1'b0;  #30 b <= 1'b1;
end
initial #120 $stop;
udp_and (c, a, b);
endmodule
```



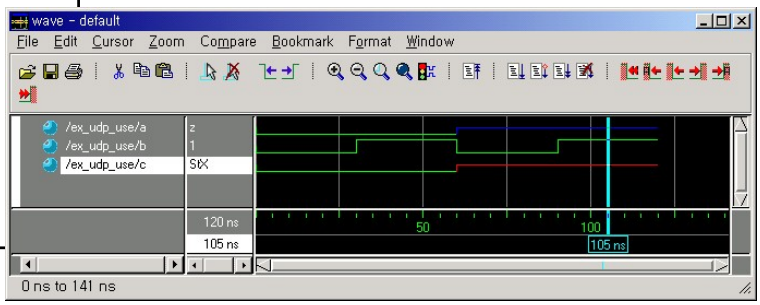
12.1 UDP 기본 (cont)

■ 상태 테이블

◆ 입력값이 상태테이블에 없으면 출력값은 X

```
module ex_udp_use;
reg a, b;  //input
wire c;    //ouput
initial begin
  a <= 1'b0;    b <= 1'b0;
end
initial
  #60 a <= 1'bZ;
initial begin
  #30 b <= 1'b1;
  #30 b <= 1'b0;
  #30 b <= 1'b1;
end
initial #120 $stop;
udp_input_Z (c, a, b);
endmodule
```

```
primitive input_Z(out, in1, in2)
 .
table
//in1  in2   out
  0    0    : 0;
  0    1    : 0;
  1    0    : 0;
  1    1    : 1;
```

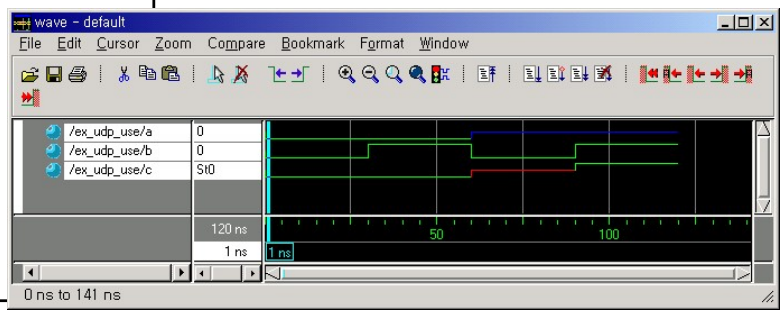


12.1 UDP 기본 (cont)

◆ 입력값이 Z 이면 상태테이블의 입력값 X 로 취급

```
module ex_udp_use;
reg a, b;  //input
wire c;    //ouput
initial begin
  a <= 1'b0;    b <= 1'b0;
end
initial
  #60 a <= 1'bZ;
initial begin
  #30 b <= 1'b1;
  #30 b <= 1'b0;
  #30 b <= 1'b1;
end
initial #120 $stop;
udp_input_Z (c, a, b);
endmodule
```

```
primitive input_Z(out, in1, in2)
 .
table
//in1  in2   out
  0    0    : 0;
  0    1    : 0;
  1    0    : 0;
  X    1    : 1;
```



◆ 상태 테이블에서 Don'care 는 ? 로 표시(? 는 1, 0, x)

◆ 상태 테이블에서 상태의 유지는 – 로 표시

12.2 조합형 UDP

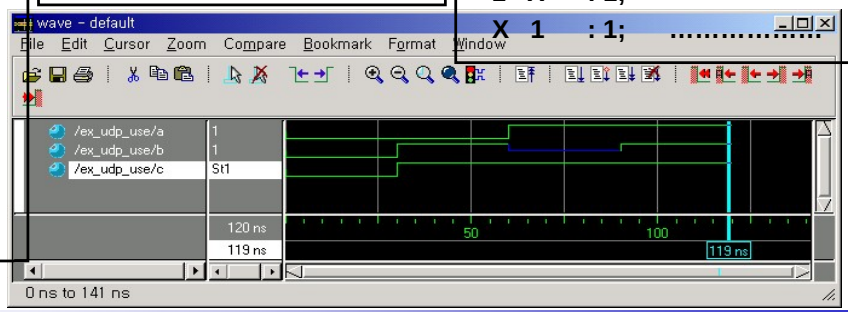
■ 조합형 UDP

◆ 현재 출력값에 관계없이 입력값의 상태테이블내의 조합에 의해서 출력값이 결정

```
module ex_udp_use;
reg a, b;  //input
wire c;    //ouput
initial begin
  a <= 1'b0;    b <= 1'b0;
end
initial
  #60 a <= 1'b1;
initial begin
  #30 b <= 1'b1;
  #30 b <= 1'bZ;
  #30 b <= 1'b1;
end
initial #120 $stop;
udp_input_or (c, a, b);
endmodule
```

```
primitive udp_or (out, in1, in2);
input in1, in2;
output out;
table
//in1  in2   out
  0    0    : 0;
  ?    1    : 1;
  1    ?    : 1;
endtable
endprimitive
```

```
primitive udp_or (out, in1, in2);
input in1, in2;
output out;
table
//in1  in2   out
  0    0    : 0;
  0    1    : 1;
  1    0    : 1;
  1    1    : 1;
  1    X    : 1;
  X    1    : 1;  ……
```



12.3 순차형 UDP

■ 순차형 UDP

- ◆ 상태 테이블의 현재의 상태와 입력값에 의해서 다음상태 및 출력값이 결정
- ◆ 순차 UDP 의 출력은 항상 reg 로 선언
- ◆ 상태테이블은 입력, 현재상태, 다음상태로 표현
- ◆ 상태테이블의 다음상태는 출력 reg 의 값
- ◆ 상태 테이블의 다음상태는 입력값과 상태테이블의 현재상태에 의해 결정
- ◆ 준위-구동 순차형 UDP 와 모서리-구동 순차형 UDP 로 구분

```
module ex_udp_use;
reg reset, clk, d;   //input
wire q;              //output
        .
        .
udp_latch(q, d, reset, clk);
        .
        .
endmodule
```

```
primitive udp_sequential (out, in1,    in2,    in3);
                                //   q ,  d     reset,  clk
input in1, in2, in3;
output out;
reg out;                        //reg 없으면 error
table
 //   d   reset   clk    q   q+
     ?     1      ? : ? : 0;
                          ..........
endtable
endprimitive
```

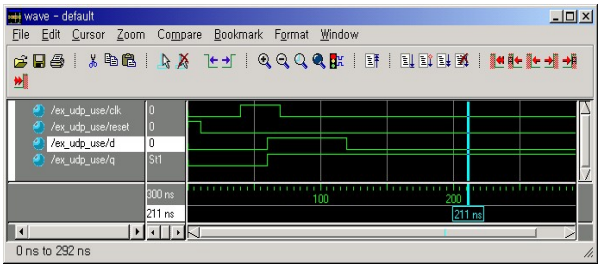

12.3 순차형 UDP (cont)

■ 준위- 구동 순차형 UDP

- ◆ 입력 수준에 따라 상태가 변화

```
module ex_udp_use;

parameter reset_p = 10;
parameter clk_p = 30;
parameter clk_2p = 2*clk_p;
reg clk, reset, d; //input
wire q;
initial begin
  clk   <= 1'b0; reset <= 1'b1; d     <= 1'b0;
  #reset_p reset <= 1'b0;
  #clk_p clk <= 1'b1;   #clk_p clk <= 1'b0;
end
initial
  begin
  #clk_2p d <= 1'b1;   #clk_2p d <= 1'b0;
end
initial #300 $stop;
udp_latch (q, d, reset, clk);
endmodule
```

```
primitive udp_latch (out, in1, in2, in3);
input in1, in2, in3;
output out;
reg out;
table
// d   reset   clk    q    q+
   ?    1       ?   : ?   : 0;
   1    0       1   : ?   : 1;
   0    0       1   : ?   : 0;
   ?    0       0   : ?   : -;
endtable
endprimitive
```



12.3 순차형 UDP (cont)

■ 모서리 – 구동 순차형 UDP

◆ 모서리 변화(Edge-triggering)과 입력 값에 따라서 상태가 변화

모서리 변화 표현	의 미
(01)	0 → 1
(10)	1 → 0
(1X)	1 → X
(0?)	0 → 0, 1, X
(??)	0, 1, X → 0, 1, X

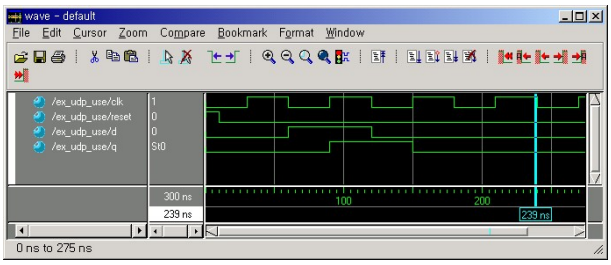
◆ 상태 테이블에서 한 개의 입력조건당 두 개의 모서리 변화는 불가능

```
table
// d   reset   clk    q    q+
  ?      1       ?   : ? : 0;
  1      0      (10): ? : 1;
 (10)   0      (10): ? : 0;
              .
              .
```


12.3 순차형 UDP (cont)

```
module ex_udp_use;

parameter reset_p = 10;
parameter clk_p = 30;
parameter clk_2p = 2*clk_p;

reg clk, reset, reg d;  //input
wire q;     //ouput
initial begin
  clk   <= 1'b0;  reset <= 1'b1;   d  <= 1'b0;
  #reset_p reset <= 1'b0;
end
always
  #clk_p clk <= ~ clk;
initial
  begin
  #clk_2p d <= 1'b1;   #clk_2p d <= 1'b0;
end
initial #300 $stop;
udp_ff (q, d, reset, clk);
endmodule
```

```
primitive udp_ff (out, in1, in2, in3);
    ……
table
// d   reset   clk    q    q+
  ?      1      ?   : ? : 0;
  ?     (10)    ?   : ? : -;
  1      0     (01) : ?: 1;
  0      0     (01) : ?: 0;
  ?      0     (1?) : ?: -;
 (??)    1      ?   : ?: -;
Endtable      ……
```



12.4 UDP 테이블 약식기호

■ UDP 테이블에서의 약식기호

약식 기호	의미	비고
?	0, 1, X	출력에는 기술불가능
b	0, 1	출력에는 기술불가능
-	상태의 변화 없음	순차 구동형에서만 사용가능
r	(01)	상승 모서리
f	(10)	하강 모서리
p	(01), (0X), (X1)	일반적인 상승모서리
n	(10), (1X), (X0)	일반적인 하강모서리
*	(??)	어떠한 변화값

```
table
// d   reset   clk   q   q+
  ?     1      ?   :  ? : 0;
  ?     f      ?   :  ? :  -;
  1     0      r   :  ?:  1;
  0     0      r   :  ?:  0;
……..
```


12.5 UDP 설계에 대한 지침

■ Item

- ◆ UDP 는 단지 기능적인 모델
- ◆ UDP 의 최대 입출력 단자의 수는 제한
  - ■ 어떠한 시뮬레이터든지 적어도 9개 이상은 지원
- ◆ 입력단자의 개수가 증가할수록 상태 테이블의 조건은 지수 승으로 증가
  - ■ 많은 입력단자의 개수를 쓰는 것은 바람직하지 않음
- ◆ UDP 의 상태 테이블은 항상 모든 조건을 고려해서 완벽하게 기술
- ◆ 상태 테이블의 준위구동 항목이 모서리 구동 항목보다 우위

13. Programmable Language Interface (PLI)

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

Content

13.1 PLI의 사용

- **PLI (Programming Language Interface)**는 사용자에게 내부 설계 표현을 억세스하는 자신만의 유틸리티를 정의하게 함으로써 **Verilog** 언어를 확장하는 강력한 기능을 제공한다.
  - 모니터링 태스크, 스티뮬러스 태스크, 디버깅 태스크
  - 번역기, 지연 계산기
  - 계층, 결합성, 팬아웃, 특정 타입의 논리 소자
  - 파형(waveforms), 논리적 연결(logic connectivity), 소스 수준 브라우저 (source level browsers), 계층(hierarchy)

13.2 PLI 태스크 링크와 동작

Hello_veirilog.c

```
#include “veriuser.h” /*include the file provided in release dir*/
Int hello.vereilog()
{
        io_printf(“Hello Verilog World\n”);
}
```

Veiusere.c

```
extern int hello_verilog();

s_tfcell veriiiiusertfs[] =
{
……
        {usertask, 0, 0 ,0, hello_verilog, 0, “$hello_verilog”, 0},
        {0}
……
};
```

첫 번째 필드는 사용자 태스크

다섯 번째 필드는 **hello_verilog** 이다.; **c** 루틴의 이름

일곱 번째 필드는 **“$hello_verilog”; Verilog** 시스템 태스크의 이름

13.2 PLI 태스크 링크와 동작 (cont)

- **PLI 태스크 링크하기(Cadence Design System의 Verilog-XL)**
  - 작업 디렉토리에 veriuser.c 파일을 복사한다.
  - veriusertfs 배열 선언부에 $hello_veilog라는 사용자-정의 시스템 테스크를 기입한다. 사용자 정의 C 루틴 hello_verilog를 외부 함수로 선언한다.
  - Verilog 릴리스 디렉토리에 있는 vconfig 유틸리티를 실행시키고 질문에 대답한다.
    - veiuser.c 파일과 hello_verilog.c 파일에 경로를 설정한다.
    - 출력 파일을 hverilog라 이름 붙인다.
    - vconfig 유틸리티는 당신의 작업 디렉토리에 cr_vlog라는 파일을 생성한다.
  - cr_vlog 파일을 실행시켜라

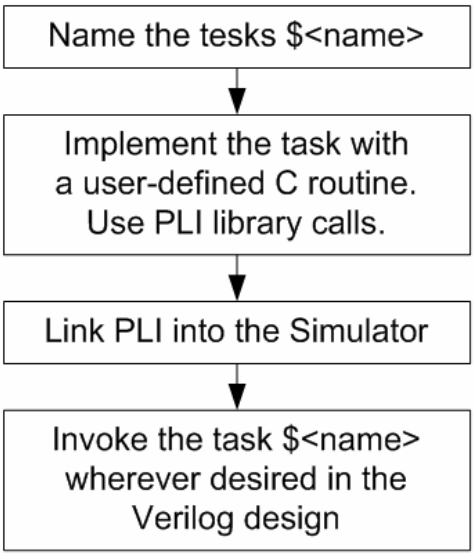
13.2 PLI 태스크 링크와 동작 (cont)

- **PLI 테스크 동작시키기**

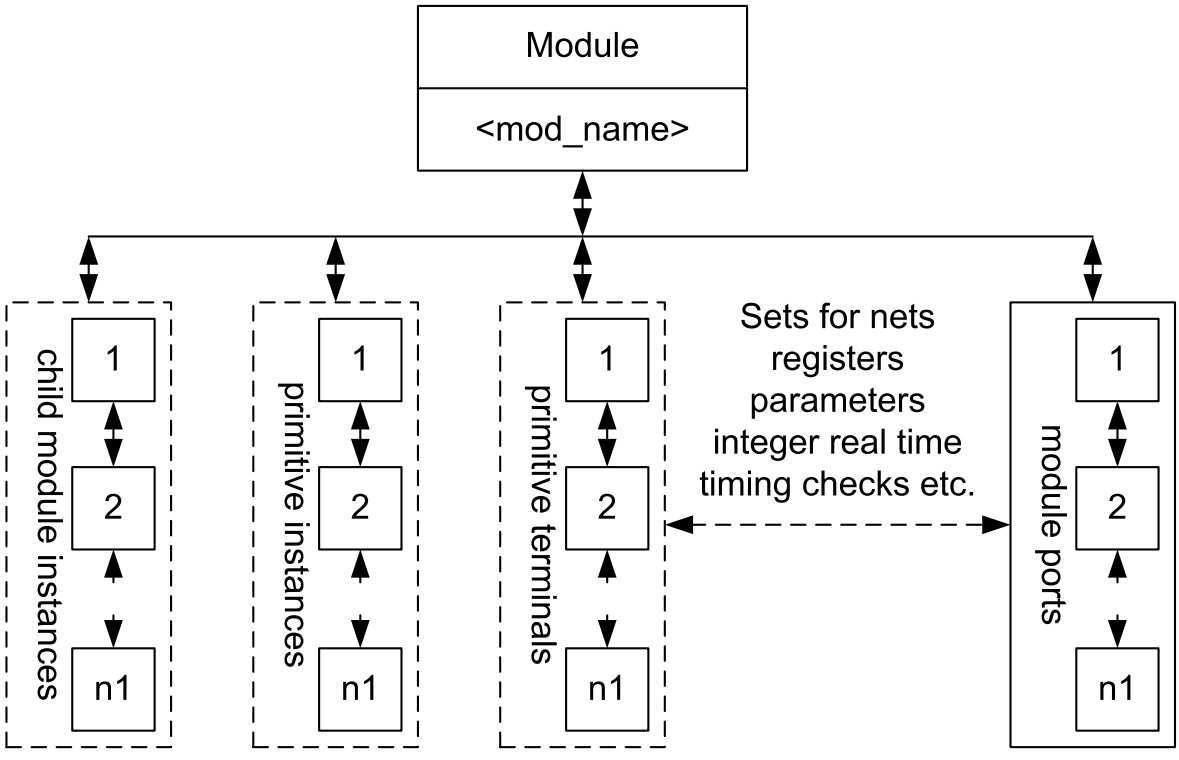
```
module hello_top;

Initial
        $hello_verilog;
endmodule
```

13.2 PLI 태스크 링크와 동작 (cont)

■ PLI 태스크 부가와 동작의 일반 과정

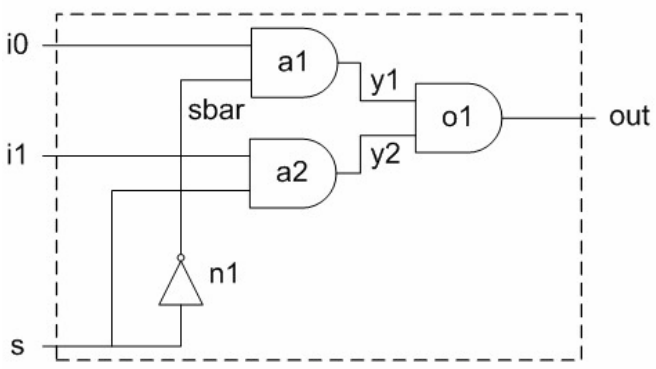


13.3 내부 데이터 표현

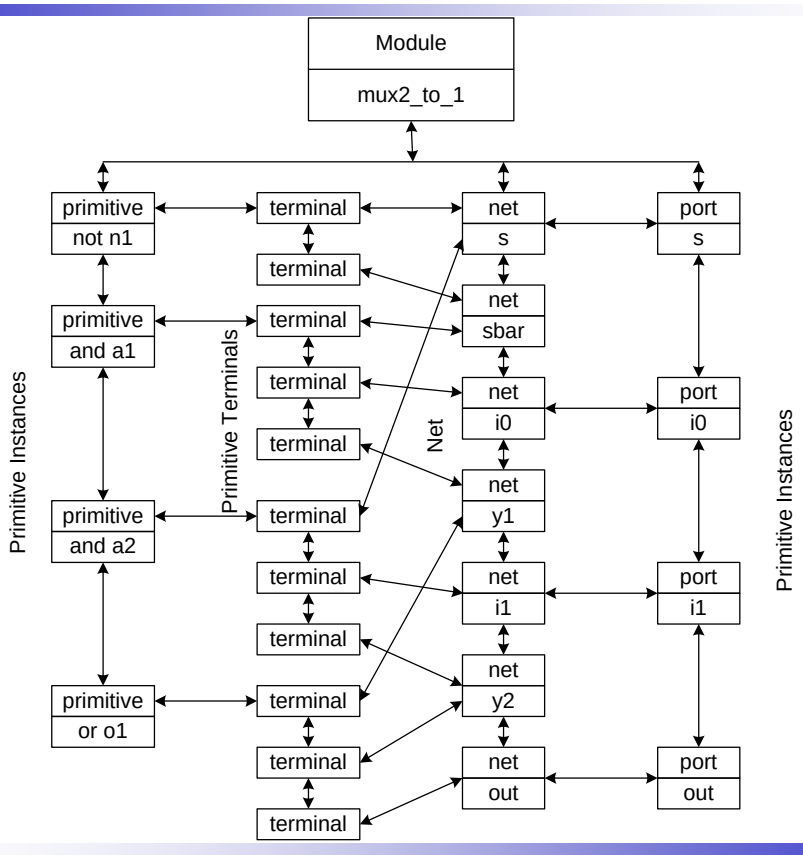


13.3 내부 데이터 표현 (cont)

```
module mux2_to_1(out, i0, i1, s);

output out;
input i0, i1;
input s;

wire sbar, y1, y2;

not n1(sbar, s);
and a1(y1, i0, sbar);
and a2(y2, i1, s);
or o1(out, y1, y2);

endmodule
```

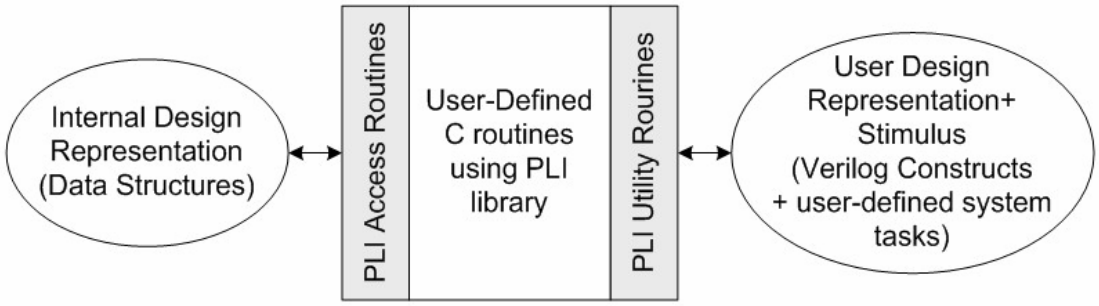


13.3 내부 데이터 표현 (cont)



13.4 PLI 라이브러리 루틴

- **$hello_verilog : 사용자 정의 시스템 태스크**
- **hello_verilog : 사용자 정의 C 루틴**
- **io_printf : PLI 라이브러리 루틴**
- **PLI 라이브러리 루틴**
  - 억세스 루틴 : 내부 데이터 표현 정보에 대한 접근을 제공
  - 유틸리티 루틴 : Verilog와 프로그래밍 언어 사이의 데이터 전달고 다방면에 걸친 보조관리 함수를 위해 사용



13.4 PLI 라이브러리 루틴 (cont)

- 억세스 루틴
  - acc 루틴
  - 내부 데이터 표현으로부터 특정 객체에 대한 정보를 읽는다.
  - 내부 데이터 표현에 특정 객체에 대한 정보를 쓴다.
  - 억세스 루틴의 원리
    - 억세스 루틴은 항상 acc로 시작한다.
    - 사용자 정의 C 루틴
      - acc_initialize() : 환경을 초기화, acc_close() : 종료
    - 파일에서 사용시
      - acc_user.h 헤더파일을 포함
  - 억세스 루틴의 형태
    - 핸들 루틴(Handle routines)
    - 다음 루틴(Next routines)
    - 값 변환 링크 루틴(Value Change Link routines)
    - 인출 루틴(Fetch routines)
    - 유틸리티 억세스 루틴(Utility access routines)
    - 수정 루틴(Modify routines)

13.4 PLI 라이브러리 루틴 (cont)

```
#include “acc_user.h”
int get_ports()
{ handle mod, portl
  int input_crt = 0;
  int output_crt = 0;
  int inout_crt = 0;
  acc_initialize();
  mod = acc_handle_tfarg(1);
  port = acc_handle_port(mod, o);
  while ( port != null ){
      if ( acc_fetch_direction(port) == accInput ){
         io_printf(“Input Port %s \n”, acc_fetch_fullname(port));
         input_crt++;}
      else if ( acc_fetch_direction(port) == accOutput ){
         io_printf(“Output Port %s \n”, acc_fetch_fullname(port));
         output_crt++;}
      else if ( acc_fetch_direction(port) == accInout ){
         io_printf(“Inout Port %s \n”, acc_fetch_fullname(port));
         inout_crt++;}
      port = acc_next_port(mod, port);   }
  io_printf(“Input Ports = %d Outpit Ports =%d, Input ports= %d\n\n”, input_crt, output_crt, inout_crt);
  acc_close();
}
```


13.4 PLI 라이브러리 루틴 (cont)

```
/*Verilog_XL의 veriuser.c 파일에 아래와 같은 라인을 추가한다. */
{usertask, 0, 0, 0, get_ports, 0, “$get_ports”, 0},
```

```
moudle top;
wire OUT;
reg I0, I1, S;

mux2_to_1 my_mux(OUT, I0, I1, s);

initial
begin
  $get_ports(“top.my_mux”);
end

endmodule
```

```
Output Port top.my_mux.out
Input Port top.my_mux.i0
Input Port top.my_mux.i1
Input Port top.my_mux.s
Input Ports = 3 Output Ports = 1, Inout Ports = 0
```

13.4 PLI 라이브러리 루틴 (cont)

```
#include “acc_user.h”

char convert_to_char();
int display_net();

int my_monitor()
{
  handel net;
  char *netname;
  char *malloc();

  acc_initialize();

  net = acchandle_tfarg(1);

  netname = malloc(srtlen(acc_fetch_fullname(net)));
  strcpy(netname, acc_fetch_fullname(net));

  acc_vcl_add(net, display_net, netname, vcl_verilog_logic);

  acc_close();
}
```


13.4 PLI 라이브러리 루틴 (cont)

```
typedef struct t_vc_record{
  int vc_reason;
  int vc_hightime;
  int vc_lowtime;
  char *user_data;
  union {
    unsigned char      logic_value;
    double real_value;
    handle             vector_handle;
    s_strengths        strengths_s;
  }          out_value;
} *p_vc_record;
```

13.4 PLI 라이브러리 루틴 (cont)

```
display_net(vc_record)
p_vc_record vc_record;
{
  io_printf("%d New value of net %s is %c \n",
            vc_record->vc_lowtime,
            vc_record->user_data,
            convert_to_char(vc_record->out_value.logic_value));
}

char convert_to_char(logic_val)
char logic_val;
{
  char temp;
  swich(logic_val)
  {
    case vc10 : temp = '0';      break;
    case vc11 : temp = '1';      break;
    case vc1x : temp = 'x';      break;
    case vc1z : temp = 'z';      break;
  }
  return(temp);
}
```


13.4 PLI 라이브러리 루틴 (cont)

```
{ usertask, 0, 0, 0, my_monitor, 0, "$my_monitor", 0},
```

```
module top;
wire OUT;
reg I0, I1, S;

mux_to_1 my_mux(OUT, I0, I1, S)
initial
begin
  $my_monitor("top.my_mux.sbar");
  $my_monitor("top.my_mux.y1");
end
initial
begin
  I0 = 1'b0; i1 = 1'b1; S=1'b0;
  #5 I0 = 1'b1; I1 = 1'b1; S = 1'b1;
  #5 I0 = 1'b1; I1 = 1'b1; S = 1'bX;
  #5 I0 = 1'b1; I1 = 1'b1; S = 1'b1;
end
endmodule
```

13.4 PLI 라이브러리 루틴 (cont)

```
0 New value of net top.my_mux.y1 is 0
0 New value of net top.my_mux.sbar is 1
5 New value of net top.my_mux.y1 is 1
5 New value of net top.my_mux.sbar is 0
5 New value of net top.my_mux.y1 is 0
10 New value of net top.my_mux.sbar is X
15 New value of net top.my_mux.y1 is X
15 New value of net top.my_mux.sbar is 0
15 New value of net top.my_mux.y1 is 0
```


13.4 PLI 라이브러리 루틴 (cont)

- 유틸리티 루틴
  - tf_로 시작한다.
  - 파일내에서 veriuser.h라는 헤더 파일을 포함
  - 유틸리티 루틴의 형태
    - Vrilog 시스템 태스트 호출에 관한 정보를 얻는다.
    - 인수 목록 정보를 얻는다
    - 인수의 값을 얻는다
    - 호출한 시스템 태스크에 새로운 인수의 값을 다시 넘겨준다.
    - 인수 값의 변화를 모니터한다.
    - 시뮬레이션 시간과 예정된 이벤트에 대한 정보를 얻는다.
    - 작업 영역을 저장, 태스크를 가리키는 포인터 저장과 같은 보조관리 태스크를 수행한다.
    - 긴 연산을 수행한다.
    - 메시지를 나타낸다.
    - 시뮬레이션의 정지, 종료, 저장, 복구

13.4 PLI 라이브러리 루틴 (cont)

```
#include “veriuser.h”
int my_stop_finish()
{
  if ( tf_nump() == 1 ){
    if ( tf_getp(1) == 0 ){
      io_printf(“Message : Simulation stopped at time %d\n”, tf_gettime());
      tf_dostop();      }
    else if ( tf_getp(1) == 1 ){
      io_printf(“Message : Simulation finished at time %d\n”, tf_gettime());
      tf_dofinish();      }
    else tf_warning(“Bad argumemts to \$my_stop_finish at time %d\n”, tf_gettime());
  }
  else if ( tf_nump() == 2 ){
    if ( tf_getp(1) == 0 ){
      io_printf(“Message : Simulation stopped at time %d in instance %s \n”, tf_gettime(), tf_mipname());
      tf_dostop();      }
    else if ( tf_getp(1) == 1 ){
      io_printf(“Message : Simulation finished at time %d in instance %s \n”, tf_gettime(), tf_mipnmae());
      tf_dofinish();      }
    else tf_warning(“Bad argumemts to \$my_stop_finish at time %d\n”, tf_gettime());
  }
}
```


13.4 PLI 라이브러리 루틴 (cont)

```
{ usertask, 0, 0, 0, my_stop_finish, 0, “$my_stop_finish”, 0},
```

```
module top;
wire OUT;
reg I0, I1, S;

mux_to_1 my_mux(OUT, I0, I1, S)
initial
begin
  I0 = 1’b0; I1 = 1’b1; S = 1’b0;
  $my_stop_finish(0);
  #5 I0 = 1’b1; I1 = 1’b1; S = 1’b1;
  $my_stop_finish(0,1);
  #5 I0 = 1’b0; I1 = 1’b1; S = 1’bX;
  $my_stop_finish(2,1);
  #5 I0 = 1’b1; I1 = 1’b1; S = 1’b1;
  $my_stop_finish(1,1);
end
endmodule
```

13.4 PLI 라이브러리 루틴 (cont)

```
Message : Simulation stopped at time 0
Type ? for help
C1 > .
Message : Simulation stopped at time 5 in instance top
C1 > .
“my_stop_finish.v”, 14: warning! Bad arguments to $my_stop_finish at time 10
Message : Simulation finished at time 15 in instance top
```


14. Logic Synthesis using Verilog HDL

Young-Ho Seo

http://explore.kw.ac.kr/~axl

design@kw.ac.kr

Digital Design & Test Lab.
Kwangwoon University

contents

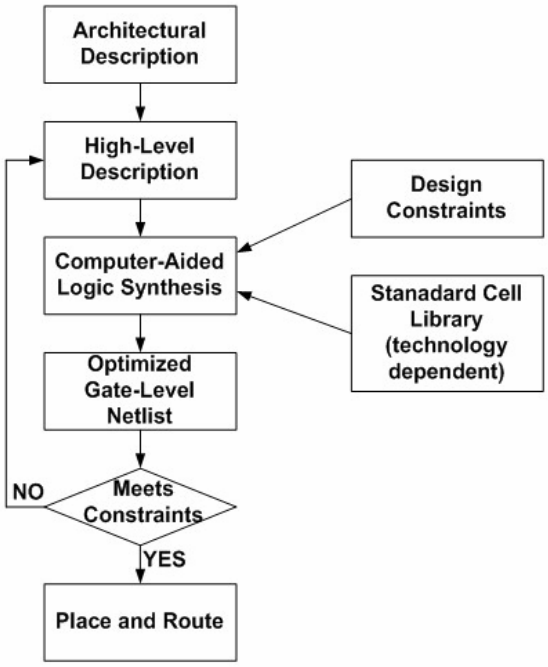
14.1 논리합성이란 무엇인가?

- 논리합성
  - 상위수준 표현의 디자인을 최적화 된 게이트 수준표현으로 바꾸는 과정

- 논리합성시 라이브러리
  - 표준 셀 라이브러리(Standard Sell Library), 테크놀로지 라이브러리 (Technology Library)
    - IC 제조회사가 제공
    - 셀의 기능성, 레이아웃정보, 타이밍정보, 전력정보등이 수록

14.1 논리합성이란 무엇인가? (cont)

- 논리합성 절차



14.2 논리합성의 영향

- 자동화된 논리합성의 도구
  - 설계자가 상위수준의 추상적인 표현(RTL 수준)만 HDL 을 표현하고 수정
    - 넷리스트(Netlist) 추출은 합성 툴에 의존
    - 공정을 고려하지 않는 HDL 기술
    - 설계수정의 간소화
  - 설계시간의 단축

14.3 Verilog HDL 합성

- 논리합성을 위한 설계
  - ◆ 레지스터 전송수준(RTL : Register Transfer Level) 으로 HDL 기술
  - ◆ 논리 합성툴은 RTL HDL 표현시 최적화된 netlist 변환
- 논리합성에 쓰이는 **Verilog HDL** 구성요소

Construct Type	Keyword or Description	Note
포트	input, output, inout	
파라미터	parameter	
모듈정의	module	
신호화 변수	wire, reg, tri	벡터허용
파생	module instances, primitive gate instances	Ex)mymux mux1(a, b, c) Ex)mux1(a, b, c)
함수와 태스크	fuction, task	타이밍 구조는 무시
순차(Procedural)	always, if, then,else, case, casex, casez	초기화는 지원되지 않음
순차적 블록	begin, end, named block, disable	명명된 블록의 무효화를 허용
데이터 플로우	assign	지연정보는 무시
루프	for, while, forever	edge triggered 표현 포함

14.3 Verilog HDL 합성 (cont)

- 논리합성을 위한 **Verilog** 연산자

연산자 타입	연산자 기호	실행되는 연산
산술	*	곱하기
	/	나누기
	+	더하기
	-	빼기
	%	나머지
	+	양수
	-	음수
논리	!	논리 not
	&&	논리 and
	||	논리 or
관계	>	보다 크다
	<	보다 작다
	>=	보다 크거나 같다
	<=	보다 작거나 같다
상등	==	같다
	!=	같지 않다

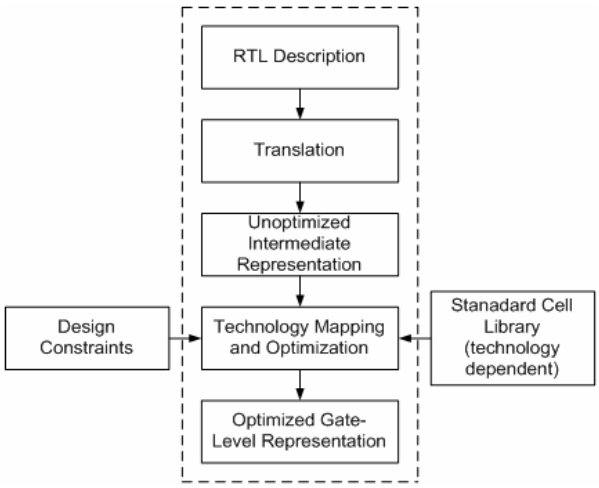
연산자 타입	연산자 기호	실행되는 연산
비트단위	~	비트단위 not
	&	비트단위 and
	|	비트단위 or
	^	비트단위 택
	^~ or ~^	비트단위 xnor
축소	&	축소 and
	~&	축소 nand
	|	축소 or
	~|	축소 nor
	^	축소 택
	^~ or ~^	축소 xnor
이동	>>	오른쪽 이동
	<<	왼쪽 이동
연결	{}	연결
조건	?:	조건

14.4 합성 설계 순서

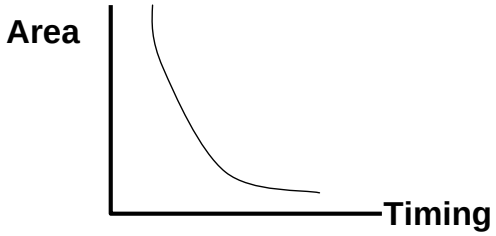
- **RTL 의 게이트로의 합성**
  - RTL 기술
    - RTL 구성요소를 이용하여 상위수준에서 설계를 기술
    - Functional Simulation
  - 번역(Translation)
    - 논리합성 도구에의해 최적화되지 않은 중간적이고 내부적인 표현으로 변환
    - 면적, 타이밍, 전력과 같은 세부제약사항은 생략
    - 최적화 되지않은 중간적 표현을 산출
    - 설계자는 알아볼수 없다
  - 논리 최적화(Logical Optimization)
    - 설계의 최저고하된 내부적 표현(Optimized internal representation) 산출
    - Netlist 산출
  - 테크놀로지 매핑과 최적화(Technology optimization)
    - 제공되는 테크놀로지 라이브러리의 셀을 이용한 최적화

14.4 합성 설계 순서 (cont)

- **Logic synthesis**



- **Area와 timing의 trade-off**



14.5 게이트 수준 넷-리스트 검증

- 기능검증 - **Functional simulation**
- 타이밍 검증
  - IC 제조회사가 시뮬레이션 라이브러리 제공

```
`timescale 1 ns / 10 ps
`celldefine
`suppress_faults
`enable_portfaults
module ad01d0(s, co, ci, b, a);
output   s, co;
xor g0 (n1, ci, b);
……..
specify
specparam a_01_LH_s =   0.511, a_10_HL_s =   0.462 ;
specparam a_10_LH_s =   0.462, a_01_HL_s =   0.462 ;
……..
endspecify
endmodule
`nosuppress_faults
`disable_portfaults
`endcelldefine
```

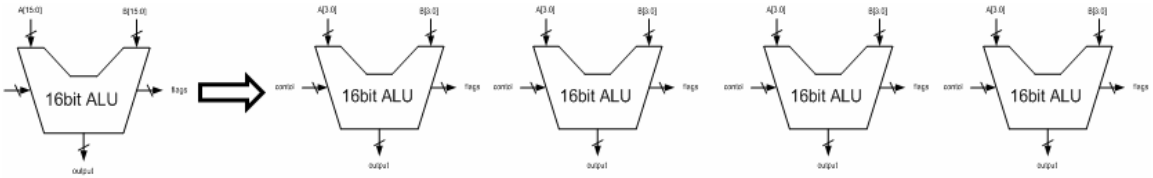

14.6 논리합성을 위한 모델링 지침

- **Verilog** 코드 작성 스타일
  - 설계 시 실제적인 하드웨어 구현에 관한 문제를 고려
  - 신호와 변수에 이름있는 이름을 사용
  - 상승/하강 모서리 구동 F/F을 혼용하지 말것
  - 기본 기능 블록을 사용하는 것과 assign 문을 사용하는 것의 구분
    - 되도록 파생시켜서 사용하지 말 것
  - 논리구조를 최적화 하기 위해서 괄호를 사용
  - 조건문(if, case..)등을 사용할 때는 default 조건 명시

14.6 논리합성을 위한 모델링 지침 (cont)

■ 설계분할

◆ 수평분할(horizontal partitioning)



◆ 수직분할(vertical partitioning)