

## 5. 행위수준 모델링



## 5.1.1 always 구문

### □ 행위수준 모델링

- ❖ 조합논리회로와 순차논리회로의 설계, 설계된 회로의 시뮬레이션을 위한 테스트벤치의 작성에 사용
- ❖ **always** 구문, **initial** 구문, **task**, **function** 내부에 사용

### □ **always** 구문

```
always [@(sensitivity_list)] begin  
    blocking_or_nonblocking statements;  
end
```

- ❖ **@(sensitivity\_list)**는 **always** 문의 실행을 제어
  - **sensitivity\_list** (감지신호목록)에 나열된 신호들 중 하나 이상에 변화(**event**)가 발생했을 때 **always** 내부에 있는 **begin – end** 블록의 실행이 트리거됨
  - **begin – end** 블록은 절차형 문장들로 구성
  - **blocking** 할당문 또는 **nonblocking** 할당문에 따라 실행 방식이 달라짐
  - 시뮬레이션이 진행되는 동안 무한히 반복 실행됨



## 5.1.1 always 구문



예 5.1.1

```
module sample(a, b, out);
    input  a, b;
    output out;
    reg    out;

    always @(a or b) begin
        if(a==1 || b==1) out = 1;      // blocking 할당문
        else               out = 0;      // blocking 할당문
    end
endmodule
```

2입력 OR 게이트

코드 5.1



## 5.1.1 always 구문



예 5.1.2 always 구문을 이용한 순차회로 모델링

```
module dff(clk, din, qout);
    input clk, din;
    output qout;
    reg qout;

    always @(posedge clk)
        qout <= din;      // Non-blocking 할당문
endmodule
```

D 플립플롭

코드 5.2



# 5.1.1 always 구문

## □ always 구문의 sensitivity\_list (감지신호목록)

### ❖ 조합논리회로 모델링

- always 구문으로 모델링되는 회로의 입력 신호가 모두 나열되어야 함
- 일부 신호가 감지신호목록에서 빠지면, 합성 이전의 RTL 시뮬레이션 결과와 합성 후의 시뮬레이션 결과가 다를 수 있음
- 함축적 감지신호 표현 (@\*)을 사용 가능

### ❖ 순차회로 모델링

- 동기식 셋/리셋을 갖는 경우 : 클록신호만 포함
- 비동기식 셋/리셋을 갖는 경우 : 클록신호, 셋, 리셋신호를 포함

```
always @(*)      // equivalent to @(a or b or c or d or f)
  y =(a & b) | (c & d) | f;
```



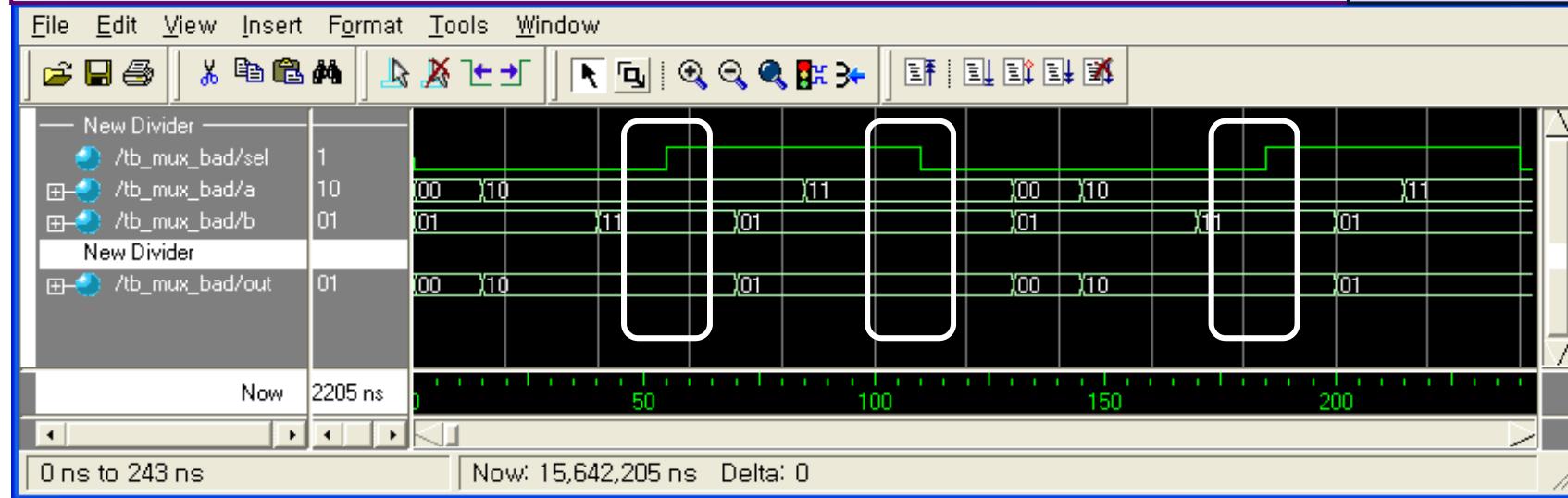
## 5.1.1 always 구문

```
module mux21_bad(a, b, sel, out);
    input [1:0] a, b;
    input      sel;
    output [1:0] out;
    reg       [1:0] out;

    always @(a or b) // sel is omitted
        if(sel ==0)
            out = a;
        else
            out = b;
endmodule
```

감지신호목록에 sel이 빠진 경우

코드 5.3



## 5.1.1 always 구문

### □ always 구문이 테스트벤치에 사용되는 경우

- ❖ 시뮬레이션 시간의 진행에 관련된 제어가 포함되어야 함
- ❖ 그렇지 않으면 **zero-delay** 무한 루프(**infinite loop**)가 발생되어 교착 상태 (**deadlock**)에 빠지게 되어 시뮬레이션이 진행되지 않음

```
always clk = ~clk; // zero-delay infinite loop
```

```
always #20 clk = ~clk; // 주기가 40ns인 신호 clk를 생성
```



## 5.1.2 initial 구문

### □ initial 구문

```
initial begin  
    blocking_or_nonblocking statements;  
end
```

- ❖ 시뮬레이션이 실행되는 동안 한번만 실행
- ❖ 절차형 문장들로 구성되며, 문장이 나열된 순서대로 실행
- ❖ 논리합성이 지원되지 않으므로 시뮬레이션을 위한 테스트벤치에 사용



예 5.1.3

```
initial begin  
    areg = 0;                                // initialize areg  
    for(index = 0; index < size; index = index + 1)  
        memory[index] = 0;                      //initialize memory word  
end
```



## 5.1.2 initial 구문



예 5.1.4

시뮬레이션 입력벡터 생성

```
initial begin
    din = 6'b000000;      // initialize at time zero
#10 din = 6'b011001;    // first pattern
#10 din = 6'b011011;    // second pattern
#10 din = 6'b011000;    // third pattern
#10 din = 6'b001000;    // last pattern
end
```

D-3



## 5.1.2 initial 구문

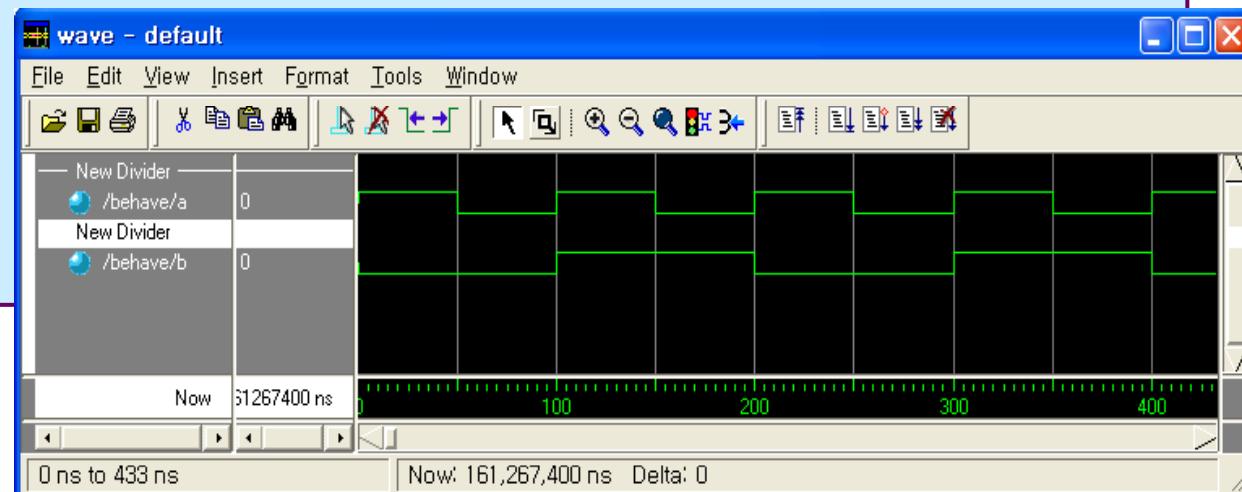


예 5.1.5

주기 신호의 생성

```
module behave;  
    reg a, b;  
  
    initial begin // 초기값 지정  
        a = 1'b1;  
        b = 1'b0;  
    end  
  
    always  
        #50 a = ~a;  
  
    always  
        #100 b = ~b;  
endmodule
```

코드 5.4



## 5.2 절차형 할당문

### □ 절차형 할당문

- ❖ **reg, integer, time, real, realtime** 자료형과 메모리 변수에 값을 갱신
- ❖ 문장이 나열된 순서대로 실행(**execute**)되어 할당문 좌변의 변수 값을 갱신하는 소프트웨어적 특성
  - 연속 할당 : 피연산자 값에 변화(**event**)가 발생할 때마다 우변의 식이 평가되고, 그 결과 값이 좌변의 **net**를 구동(**drive**)하는 하드웨어적 특성
- ❖ **Blocking** 할당문
  - 할당기호 **=** 을 사용
- ❖ **Nonblocking** 할당문
  - 할당기호 **<=** 을 사용



## 5.2.1 Blocking 할당문

### □ Blocking 할당문

- ❖ 현재 할당문의 실행이 완료된 이후에 그 다음의 할당문이 실행되는 순차적 흐름을 가짐

```
reg_lvalue = [delay_or_event_operator] expression;
```



예 5.2.1

```
initial begin
    rega = 0;                                // reg형 변수에 대한 할당
    regb[3] = 1;                               // 단일 비트에 대한 할당
    regc[3:5] = 7;                             // 부분 비트에 대한 할당
    mema[address] = 8'hff;                     // 메모리 요소에 대한 할당
    {carry, acc} = rega + regb;                // 결합에 대한 할당
end
```



## 5.2.2 Nonblocking 할당문

### □ Nonblocking 할당문

- ❖ 나열된 할당문들이 순차적 흐름에 대한 **blocking** 없이 정해진 할당 스케줄 (**assignment scheduling**)에 의해 값이 할당
- ❖ 할당문들은 우변이 동시에 평가된 후, 문장의 나열 순서 또는 지정된 자연 값에 따른 할당 스케줄에 의해 좌변의 객체에 값이 갱신
  - 동일 시점에서 변수들의 순서나 상호 의존성에 의해 할당이 이루어져야 하는 경우에 사용

```
reg_lvalue <= [delay_or_event_operator] expression;
```



## 5.2.2 Nonblocking 할당문



예 5.2.2

nonblocking 할당문과 blocking 할당문의 비교

```
module non_blk1;
    output out;
    reg a, b, clk;

    initial begin
        a = 0;
        b = 1;
        clk = 0;
    end

    always clk = #5 ~clk;

    always @(posedge clk) begin
        a <= b;
        b <= a;
    end
endmodule
```

코드 5.5-(a)

```
module blk1;
    output out;
    reg a, b, clk;

    initial begin
        a = 0;
        b = 1;
        clk = 0;
    end

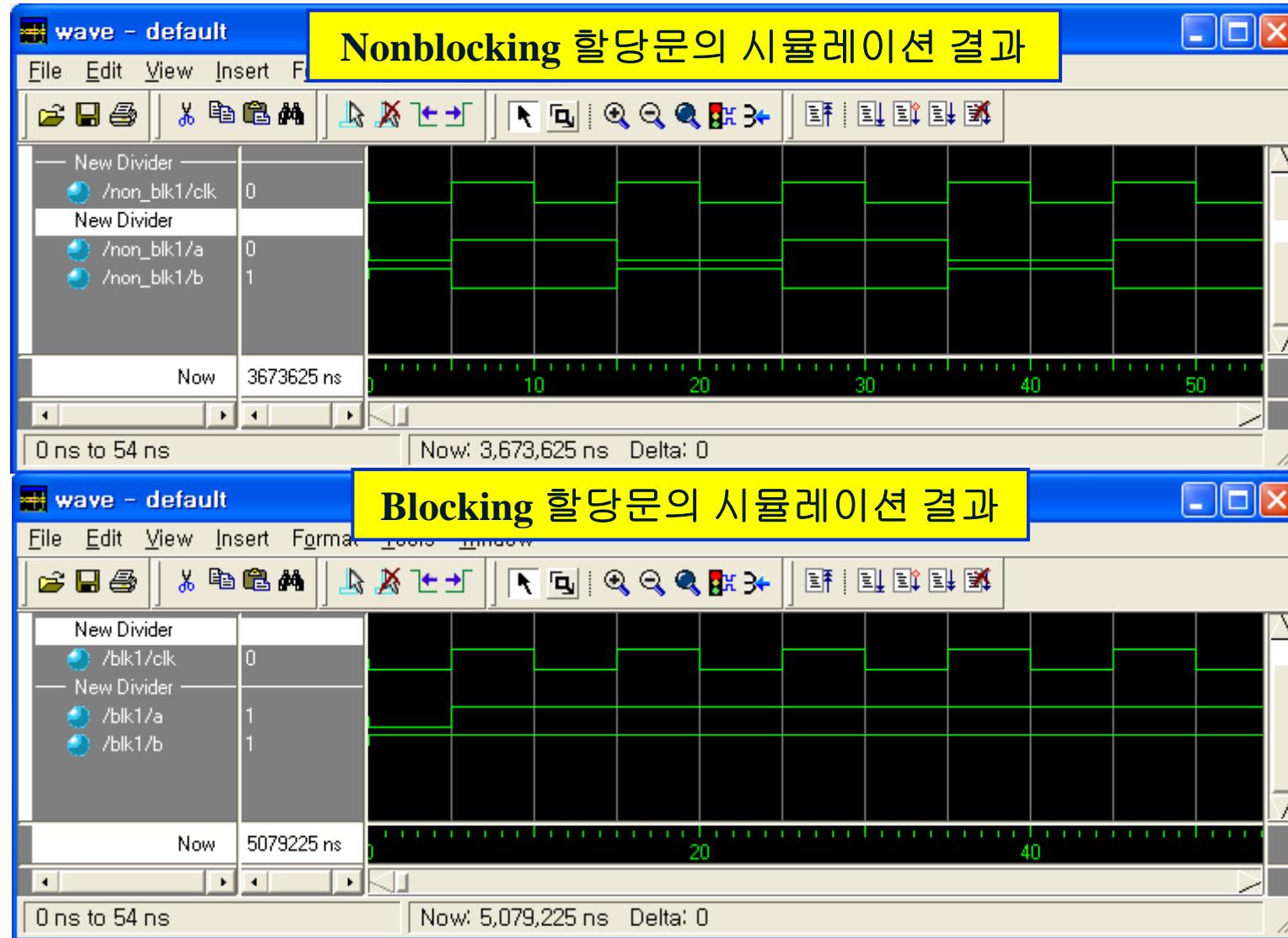
    always clk = #5 ~clk;

    always @(posedge clk) begin
        a = b;          // a=1
        b = a;          // b=a=1
    end
endmodule
```

코드 5.5-(b)



## 5.2.2 Nonblocking 할당문



## 5.2.2 Nonblocking 할당문



예 5.2.3 blocking 할당문과 nonblocking 할당문이 자연을 갖는 경우

```
module non_block2;
    reg a, b, c, d, e, f;
    //blocking assignments with intra-assignment delay
    initial begin
        a = #10 1;          // a will be assigned 1 at time 10
        b = #2 0;           // b will be assigned 0 at time 12
        c = #4 1;           // c will be assigned 1 at time 16
    end
    //non-blocking assignments with intra-assignment delay
    initial begin
        d <= #10 1;         // d will be assigned 1 at time 10
        e <= #2 0;           // e will be assigned 0 at time 2
        f <= #4 1;           // f will be assigned 1 at time 4
    end
endmodule
```

코드 5.6



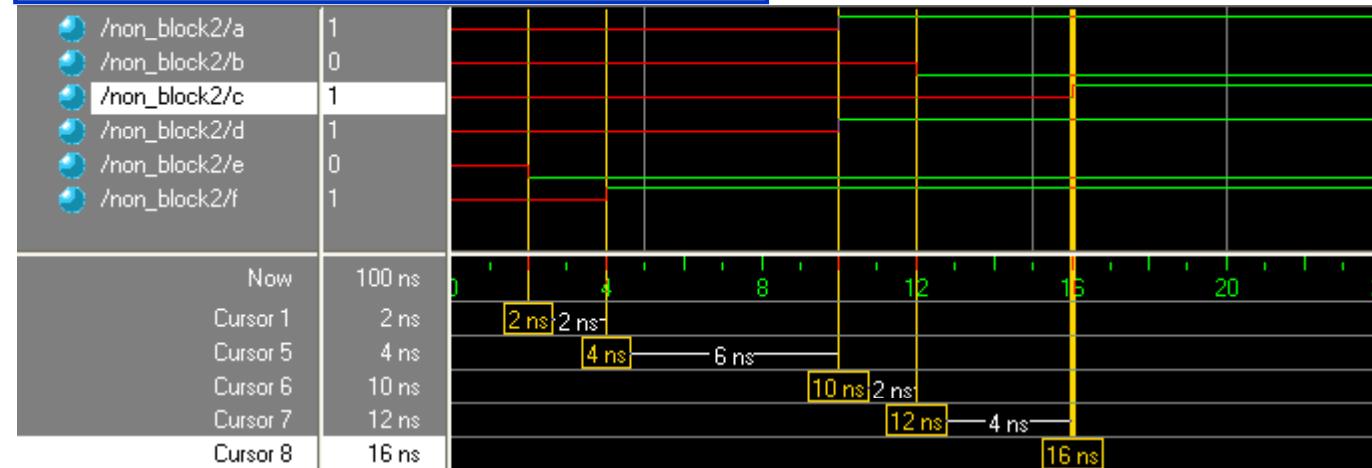
## 5.2.2 Nonblocking 할당문

```
module non_block3;
    reg a, b, c, d, e, f;
//blocking assignments with delay
    initial begin
        #10 a = 1;          // a will be assigned 1 at time 10
        #2   b = 0;          // b will be assigned 0 at time 12
        #4   c = 1;          // c will be assigned 1 at time 16
    end
//non-blocking assignments with delay
    initial begin
        #10 d <= 1;         // d will be assigned 1 at time 10
        #2   e <= 0;         // e will be assigned 0 at time 12
        #4   f <= 1;         // f will be assigned 1 at time 16
    end
endmodule
```

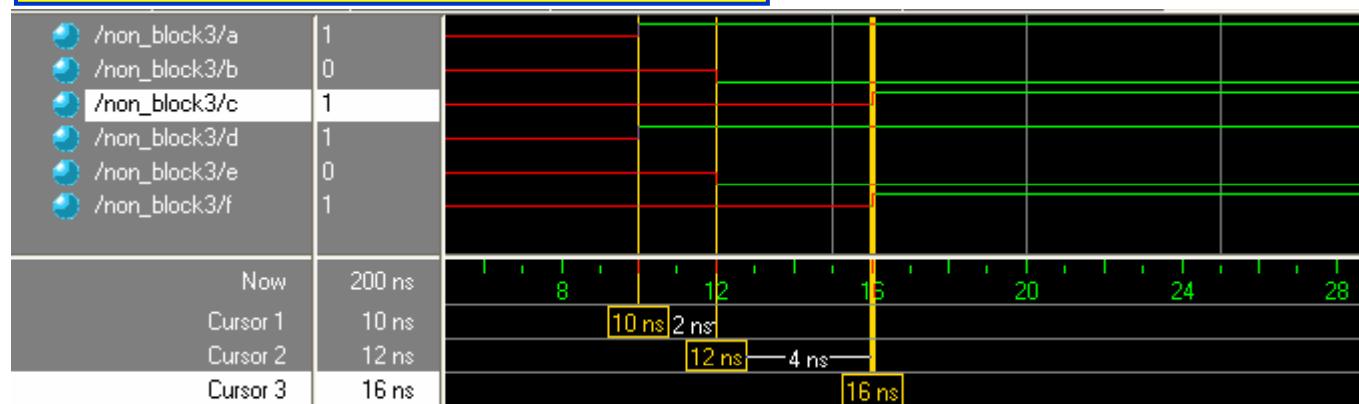


## 5.2.2 Nonblocking 할당문

non\_block2의 시뮬레이션 결과



non\_block3의 시뮬레이션 결과



## 5.2.2 Nonblocking 할당문



예 5.2.4

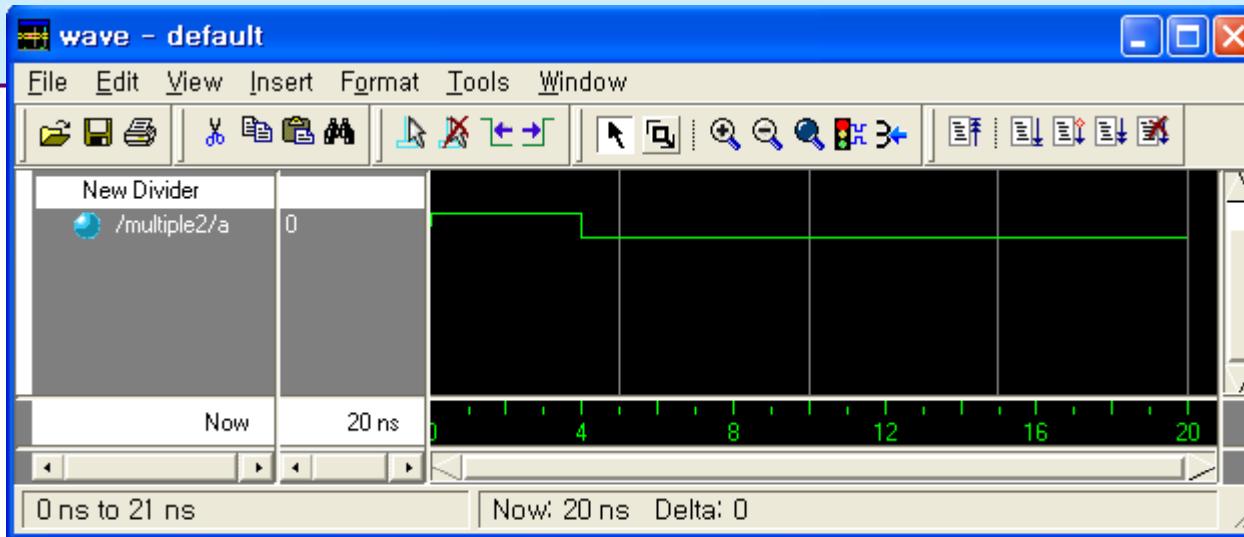
Nonblocking 할당문의 내부 지연

```
module multiple2;
    reg a;

    initial a = 1;

    initial begin
        a <= #4 1;      // schedules a = 1 at time 4
        a <= #4 0;      // schedules a = 0 at time 4
        #20 $stop;      // At time 4, a = 0
    end
endmodule
```

코드 5.7



## 5.2.2 Nonblocking 할당문



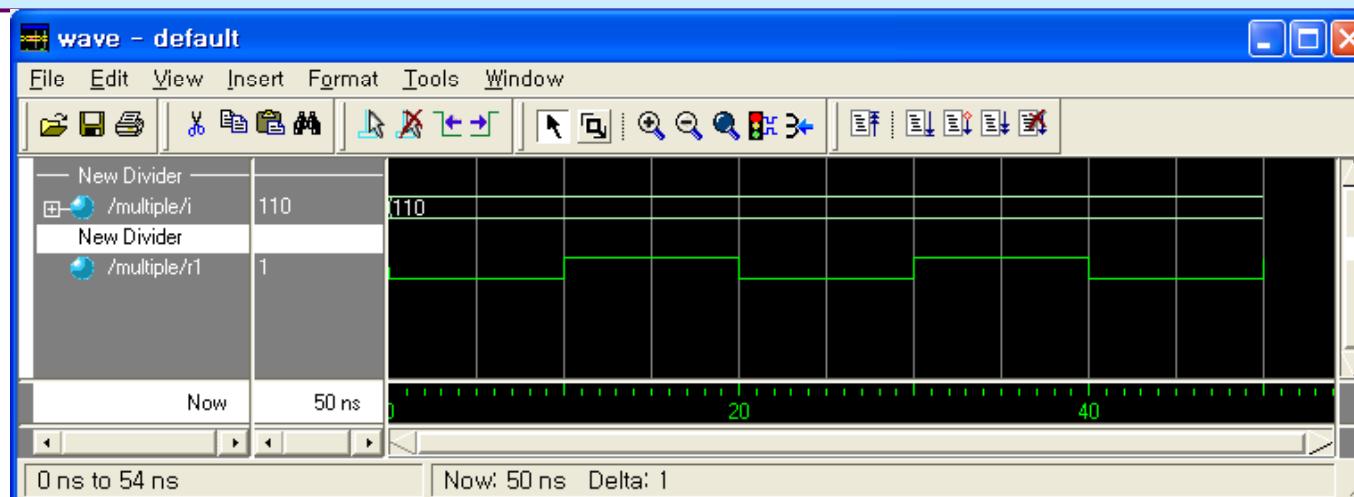
예 5.2.5

Nonblocking 할당문의 내부 지연을 이용한 주기 신호의 생성

```
module multiple;
    reg r1;
    reg [2:0] i;

    initial begin
        r1 = 0;
        for(i = 0; i <= 5; i = i+1)
            r1 <= #(i*10) i[0];
    end
endmodule
```

코드 5.8



# 5.3 if 조건문

## □ if 조건문

```
if(expression)
    statement_true;
[else
    statement_false;]
```

- ❖ 조건식이 참 (**0**이 아닌 알려진 값)이면, **statement\_true** 부분 실행
- ❖ 조건식이 거짓 (**0, x, z**)이면, **statement\_false** 부분 실행
- ❖ **else** 부분이 없으면, 변수는 이전에 할당 받은 값을 유지 (**latch** 동작)
- ❖ **if** 문의 내포(**nested**) (즉, **if** 문 내부에 또 다른 **if** 문이 있는 경우)
  - **if** 문 내부에 하나 이상의 **else** 문장이 생략되는 경우, 내부의 **else** 문은 이전의 가장 가까운 **if**와 결합됨
  - 들여쓰기 또는 **begin – end**로 명확하게 표현



## 5.3 if 조건문

```
if(index > 0)          // IF-1
    if(rega > regb)    // IF-2
        result = rega;
    else                // else of the IF-2
        result = regb;
```

```
if(index > 0) begin    // IF-1
    if(rega > regb)
        result = rega;   // IF-2
end
else result = regb;    // else of the IF-1
```



# 5.3 if 조건문

```
module mux21_if(a, b, sel, out);
    input [1:0] a, b;
    input      sel;
    output [1:0] out;
    reg      [1:0] out;

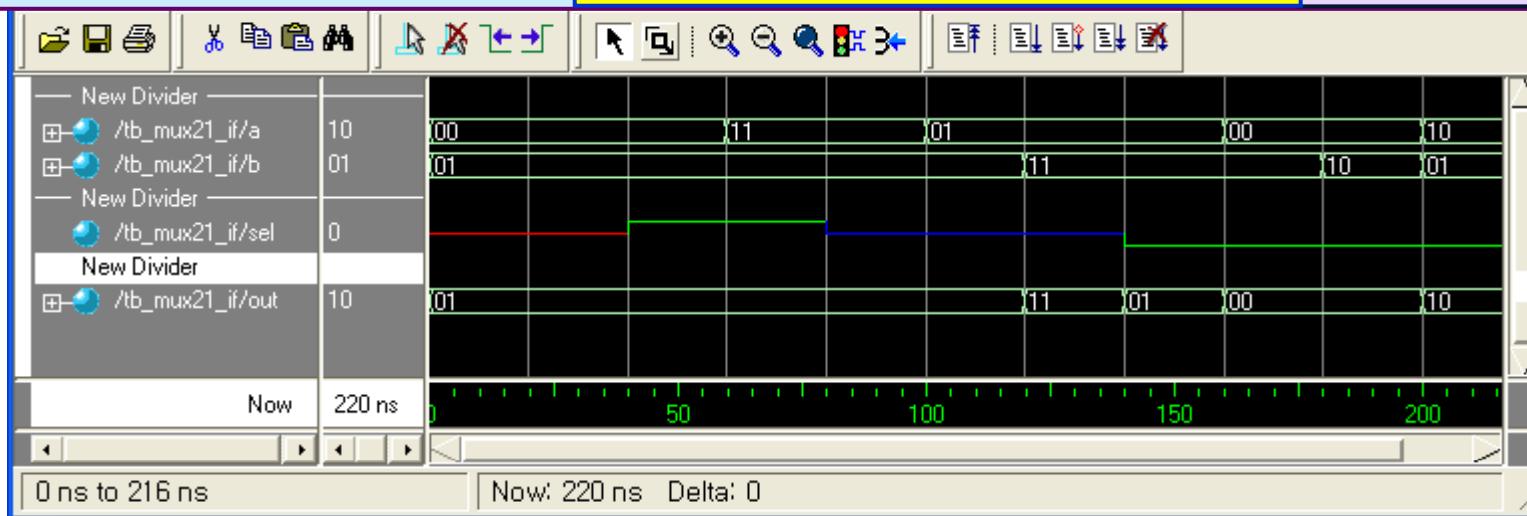
    always @(a or b or sel)
        if(sel == 1'b0)      // 또는 if(!sel)
            out = a;
        else
            out = b;
endmodule
```



예 5.3.1

if 조건문을 이용한 2 : 1 멀티플렉서

코드 5.9



# 5.3 if 조건문



예 5.3.2

비동기 set/reset을 갖는 D 플립플롭

```
module dff_sr_async(clk, d, rb, sb, q, qb);
    input clk, d, rb, sb;
    output q, qb;
    reg q;
    always @(posedge clk or negedge rb or negedge sb)
    begin
        if(rb==0)
            q <= 0;
        else if(sb==0)
            q <= 1;
        else
            q <= d;
    end

    assign qb = ~q;
endmodule
```

코드 5.10

D-3

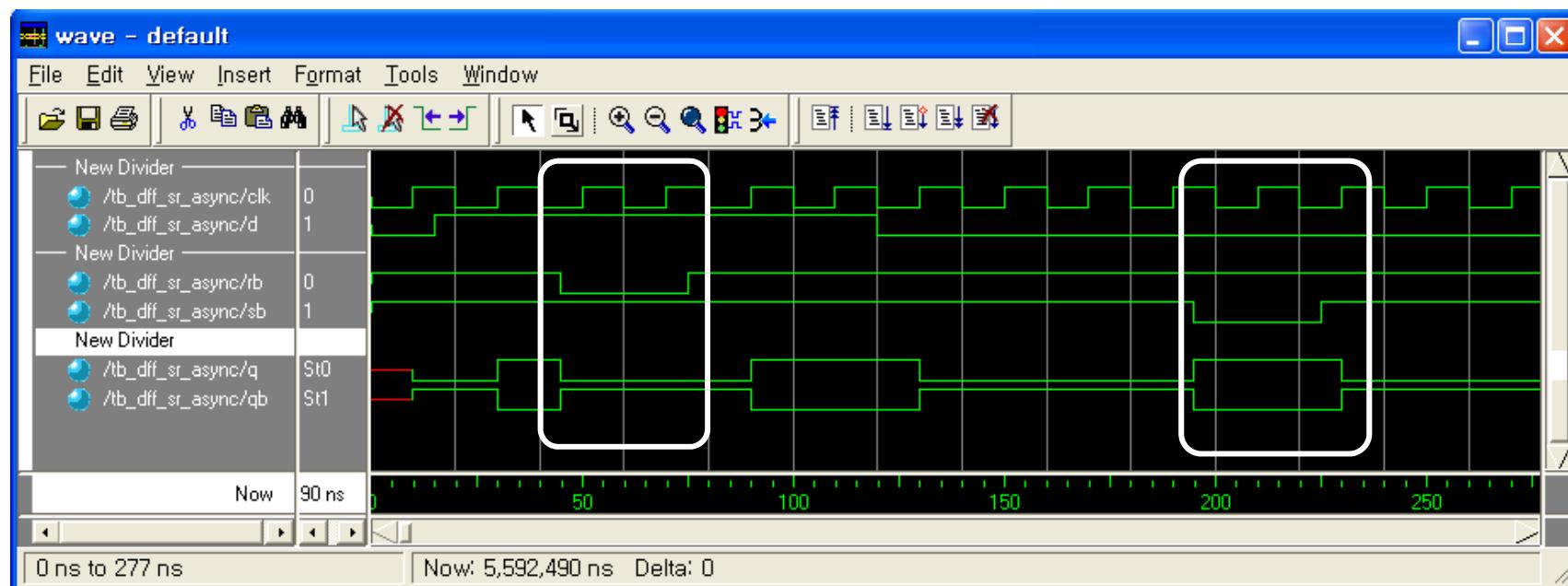


# 5.3 if 조건문



예 5.3.2

코드 5.10의 시뮬레이션 결과



# 5.4 case 문

## □ case 문

```
case(expression)
  case_item {, case_item} : statement_or_null;
  | default [:] statement_or_null;
endcase
```

- ❖ **case** 조건식의 값과 일치하는 **case\_item**의 문장이 실행
  - 각 비트가 **0, 1, x, z**를 포함하여 정확히 같은 경우에만 일치로 판단
  - **case** 문의 조건식과 모든 **case\_item**의 비트 크기는 큰 쪽에 맞추어져야 함
- ❖ 모든 **case\_item**들에 대한 비교에서 일치되는 항이 없는 경우에는 **default** 항이 실행
- ❖ **default** 항이 없으면 변수는 이전에 할당받은 값을 유지
- ❖ 하나의 **case** 문에서 여러 개의 **default**를 사용하는 것은 허용되지 않음



# 5.4 case 문



예 5.4.1

case문을 이용한 2 : 1 멀티플렉서

```
module mux21_case(a, b, sel, out);
    input [1:0] a, b;
    input      sel;
    output [1:0] out;
    reg       [1:0] out;

    always @(a or b or sel) begin
        case(sel)
            0 : out = a;
            1 : out = b;
        endcase
    end
endmodule
```

코드 5.11



## 5.4 case 문



예 5.4.2

case문을 이용한 디코더

```
reg [15:0] rega;
reg [9:0] result;

always @(rega) begin
    case(rega)
        16'd0: result = 10'b0111111111;
        16'd1: result = 10'b1011111111;
        16'd2: result = 10'b1101111111;
        16'd3: result = 10'b1110111111;
        16'd4: result = 10'b1111011111;
        16'd5: result = 10'b1111101111;
        16'd6: result = 10'b1111110111;
        16'd7: result = 10'b1111111011;
        16'd8: result = 10'b1111111101;
        16'd9: result = 10'b1111111110;
        default: result = 10'bx;
    endcase
end
```

D-3



# 5.4 case 문



예 5.4.3

case 문을 이용한 x와 z의 처리 예

```
case(select[1:2])
    2'b00      : result = 0;
    2'b01      : result = flaga;
    2'b0x, 2'b0z : result = flaga ? 'bx : 0;
    2'b10      : result = flagb;
    2'bx0, 2'bz0 : result = flagb ? 'bx : 0;
    default     : result = 'bx;
endcase
```



예 5.4.4

case 문을 이용한 x와 z의 검출 예

```
case(sig)
    1'bz: $display("signal is floating");
    1'bx: $display("signal is unknown");
    default: $display("signal is %b", sig);
endcase
```



## 5.4 case 문

### □ don't care를 갖는 case 문

- ❖ **casez** 문 : z를 don't - care로 취급하여 해당 비트를 비교에서 제외
  - don't – care 조건으로 ? 기호 사용 가능
- ❖ **casedx** 문 : x와 z를 don't - care로 취급하여 해당 비트를 비교에서 제외



예 5.4.5

```
reg [7:0] ir;  
casez(ir)  
 8'b1???????: instruction1(ir);  
 8'b01???????: instruction2(ir);  
 8'b00010????: instruction3(ir);  
 8'b000001???: instruction4(ir);  
endcase
```



## 5.4 case 문



예 5.4.6

casex 문을 이용한 3비트 우선순위 인코더(priority encoder)

```
module pri_enc_casex(encode, enc);
    input [3:0] encode;
    output [1:0] enc;
    reg      [1:0] enc;

    always @(encode) begin
        casex(encode)
            4'b1xxx : enc = 2'b11;
            4'b01xx : enc = 2'b10;
            4'b001x : enc = 2'b01;
            4'b0001 : enc = 2'b00;
        endcase
    end
endmodule
```

코드 5.12

D-3

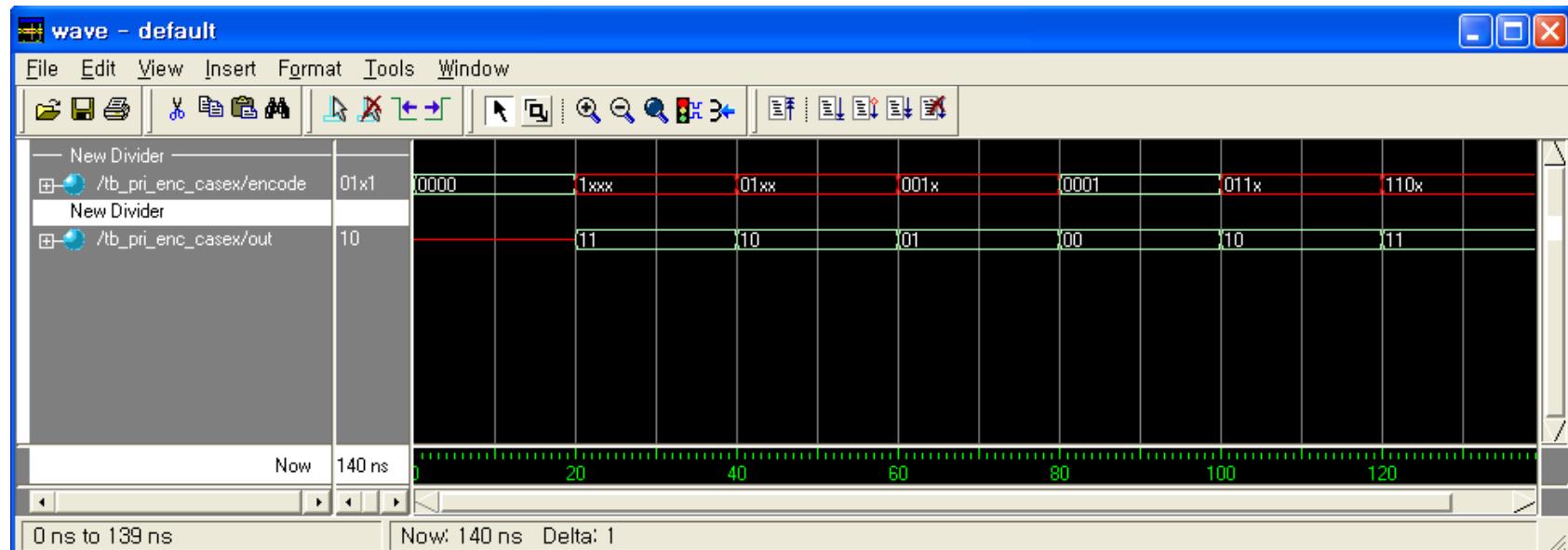


## 5.4 case 문



예 5.4.6

코드 5.12의 시뮬레이션 결과



## 5.4 case 문



예 5.4.7

case 조건식에 상수 값을 사용한 3비트 우선순위 인코더

```
module pri_enc_case(encode, enc);
    input [3:0] encode;
    output [1:0] enc;
    reg      [1:0] enc;

    always @ (encode) begin
        case (1)
            encode[3]: enc = 2'b11;
            encode[2]: enc = 2'b10;
            encode[1]: enc = 2'b01;
            encode[0]: enc = 2'b00;
            default   : $display("Error: One of the bits expected ON");
        endcase
    end
endmodule
```

코드 5.13



# 5.5 반복문

## □ 반복문

- ❖ **forever** 문 : 문장이 무한히 반복적으로 실행
- ❖ **repeat** 문 : 지정된 횟수만큼 문장이 반복 실행
  - 반복 횟수를 나타내는 수식이 **x** 또는 **z**로 평가되면 반복 횟수는 **0**이 되고, 문장은 실행되지 않음
- ❖ **while** 문 : 조건식의 값이 거짓이 될 때까지 문장이 반복 실행
  - 조건식의 초기값이 거짓이면 문장은 실행되지 않음
- ❖ **for** 문 : 반복 횟수를 제어하는 변수에 의해 문장이 반복 실행

```
forever statement;
| repeat(expression) statement;
| while(expression) statement;
| for(variable_assign); expression; variable_assign) statement;
```



## 5.5 반복문



예 5.5.1

repeat 문을 이용한 shift-add 방식의 승산기

```
module multiplier_8b(opa, opb, result);
    parameter SIZE = 8, LongSize = 2*SIZE;
    input [SIZE-1:0] opa, opb;
    output [LongSize-1:0] result;
    reg [LongSize-1:0] result, shift_opa, shift_opb;

    always @(opa or opb) begin
        shift_opa = opa;           // multiplicand
        shift_opb = opb;           // multiplier
        result = 0;
        repeat(SIZE) begin
            if(shift_opb[0])
                result = result + shift_opa;
            shift_opa = shift_opa << 1;
            shift_opb = shift_opb >> 1;
        end
    end
endmodule
```

코드 5.14

D-3

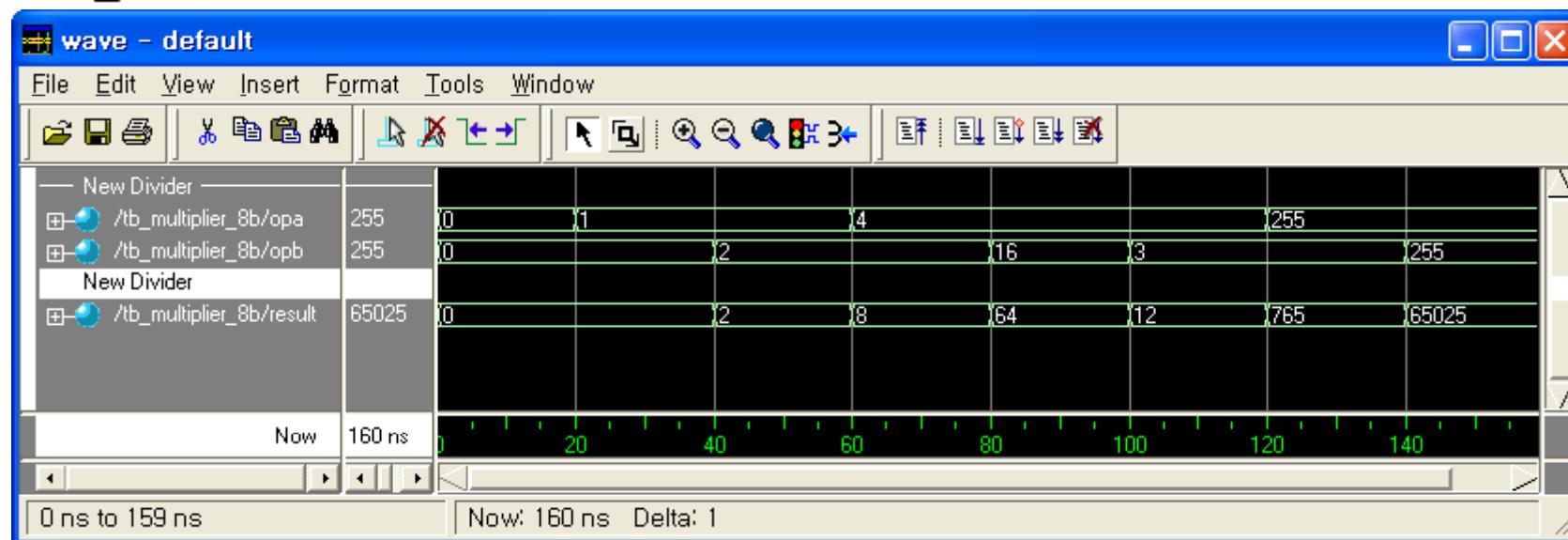


## 5.5 반복문



예 5.5.1

코드 5.14의 시뮬레이션 결과



## 5.5 반복문



예 5.5.2

8비트 입력 rega에 포함된 1을 계수하는 회로

```
module cnt_one(rega, count);
    input [7:0] rega;
    output [3:0] count;
    reg [7:0] temp_reg;
    reg [3:0] count;

    always @(rega) begin
        count = 0;
        temp_reg = rega;
        while(temp_reg) begin
            if(temp_reg[0])
                count = count + 1;
            temp_reg = temp_reg >> 1;
        end
    end

endmodule
```

코드 5.15

D-3

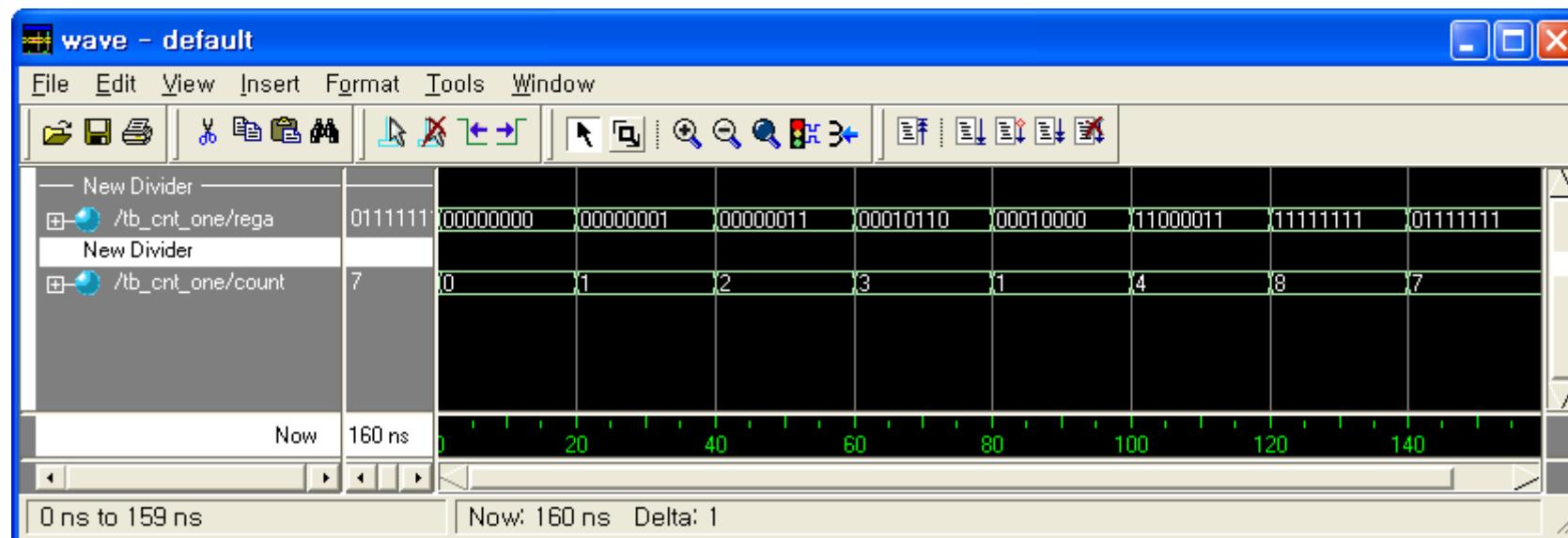


# 5.5 반복문



예 5.5.2

코드 5.15의 시뮬레이션 결과



## 5.5 반복문



예 5.5.3

for 문을 이용한 8비트 우선순위 인코더(priority encoder)

```
module enc_for(in, out);
    input [7:0] in;
    output [2:0] out;
    reg [2:0] out;
    integer i;

    always @(in) begin : LOOP
        out=0;
        for(i = 7; i >= 0; i = i-1) begin
            if(in[i]) begin
                out=i;
                disable LOOP;
            end
        end
    end
endmodule
```

입력 in[7:0]	출력 out[2:0]
0000_0001	000
0000_0010	001
0000_0100	010
0000_1000	011
0001_0000	100
0010_0000	101
0100_0000	110
1000_0000	111

8비트 우선순위 인코더

코드 5.16

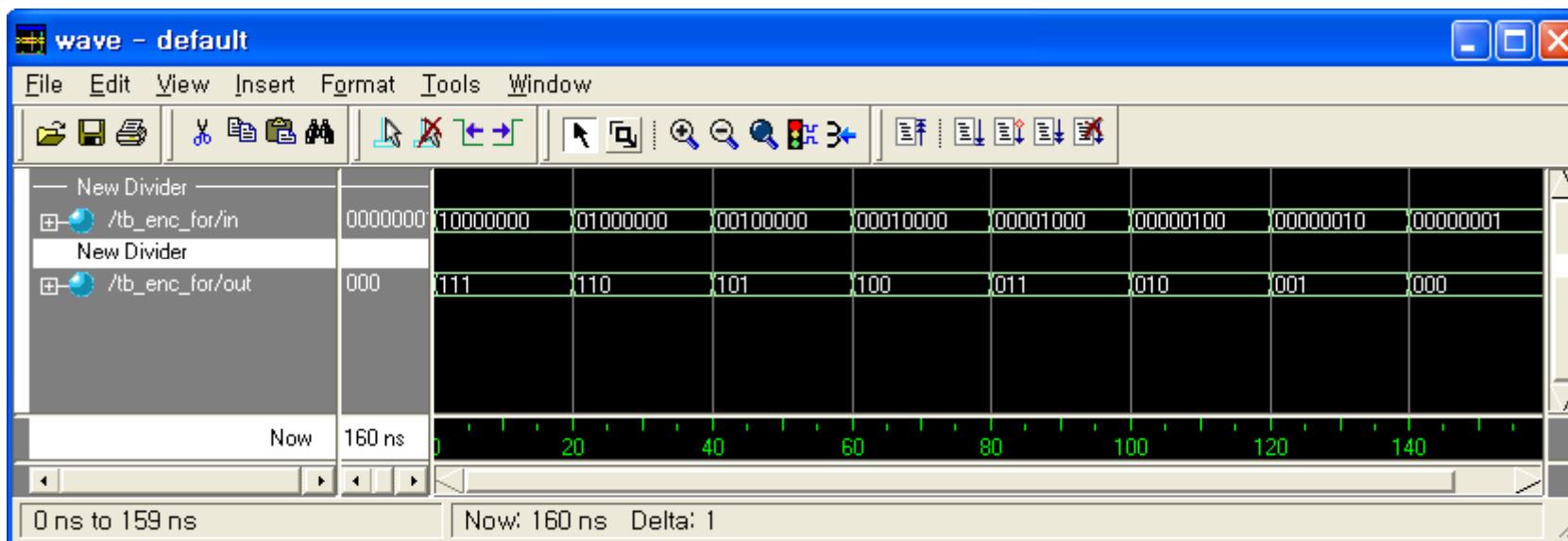


# 5.5 반복문



예 5.5.3

코드 5.16의 시뮬레이션 결과



## 5.5 반복문

```
module tb_dff ;  
    reg clk, d;
```

```
    dff U1 (clk, d, q);
```

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

```
    initial begin  
        d = 1'b0;  
        forever begin  
            #15 d = 1'b1;  
            #20 d = 1'b0;  
            #30 d = 1'b1;  
            #20 d = 1'b0;  
        end  
    end
```

```
endmodule
```

forever 문을 이용한 주기 신호 생성



# 5.6 절차형 할당의 타이밍 제어

## □ 절차형 할당의 실행 제어

### ❖ 지연 제어 :

- **delay operator : #**
- 특정 절차형 할당문의 실행 순서가 된 시점과 그 문장이 실제 실행되는 시점 사이의 시간 간격을 지정 (**문장의 실행을 지연시킴**)
- 지연 값이 **x** 또는 **z**인 경우에는 지연이 **0**으로 처리
- 음수 지연 값이 지정된 경우에는 **2**의 보수 **unsigned** 정수로 해석

```
#10 rega = regb;
#d  rega = regb;          // d is defined as a parameter
#((d+e)/2) rega = regb;  // delay is average of d and e
```

### ❖ event 제어

- **event operator : @**
- 시뮬레이션 **event**가 발생될 때까지 문장의 실행을 지연시킴
- **net**나 변수의 값 변화가 순차문의 실행을 트리거하기 위한 **event**로 사용 가능



# 5.6 절차형 할당의 타이밍 제어

## ❖ 에지 천이 검출

- **negedge** : 1에서 0, x, z로 변화, 또는 x, z에서 0으로 변화에서 **event** 발생
- **posedge** : 0에서 x, z, 1로 변화, 또는 x, z에서 1로 변화에서 **event** 발생
- **event** 발생 수식의 결과가 1비트 이상인 경우에는, 에지 천이는 결과의 LSB에서 검출

표 5.1 **posedge**와 **negedge** event의 발생

From \ To	0	1	x	z
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge



# 5.6 절차형 할당의 타이밍 제어



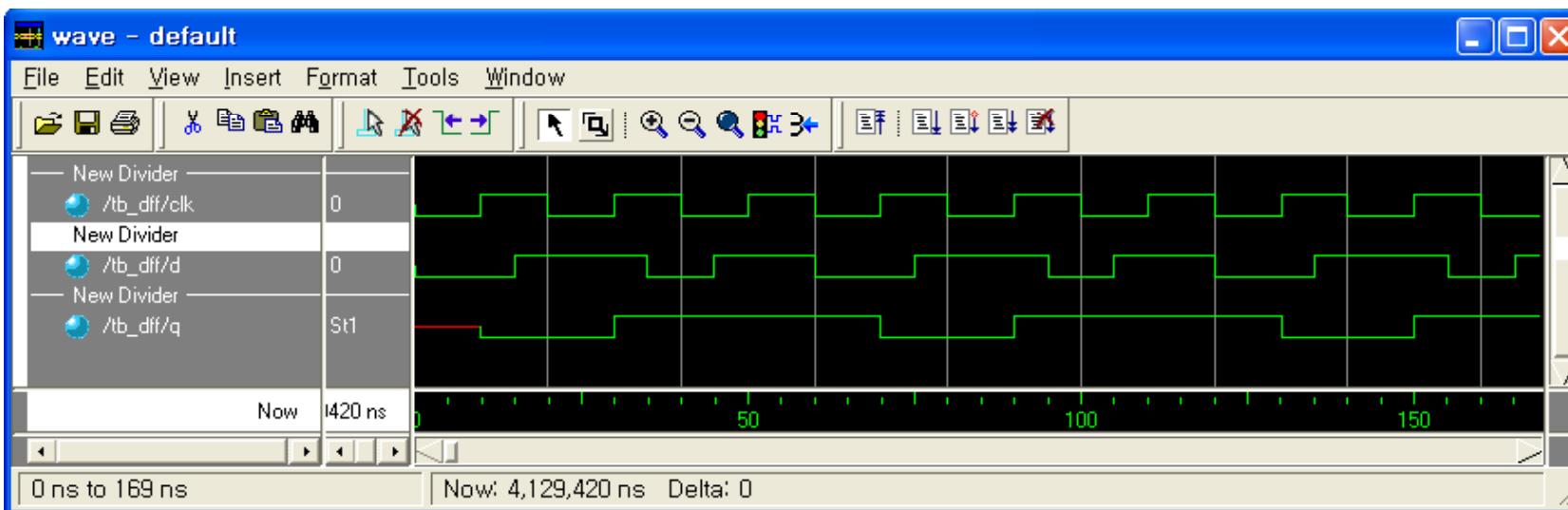
예 5.6.1

상승에지로 동작하는 D 플립플롭

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

    always @(posedge clk)
        q <= d;
endmodule
```

코드 5.17



# 5.6 절차형 할당의 타이밍 제어

## □ named event

- ❖ **event** 자료형으로 선언된 식별자
- ❖ 순차문의 실행을 제어하기 위한 **event** 표현에 사용
  - **event**가 발생될 때까지 실행이 지연됨
- ❖ **named event**는 데이터 값을 유지하지 않음
- ❖ 임의의 특정한 시간에 발생될 수 있음
- ❖ 지속 시간을 갖지 않음
- ❖ **event** 제어 구문을 이용해서 **event**의 발생을 감지할 수 있음

```
event list_of_event_identifiers; // event 선언
```

```
-> event_identifier; // event trigger
```



# 5.6 절차형 할당의 타이밍 제어



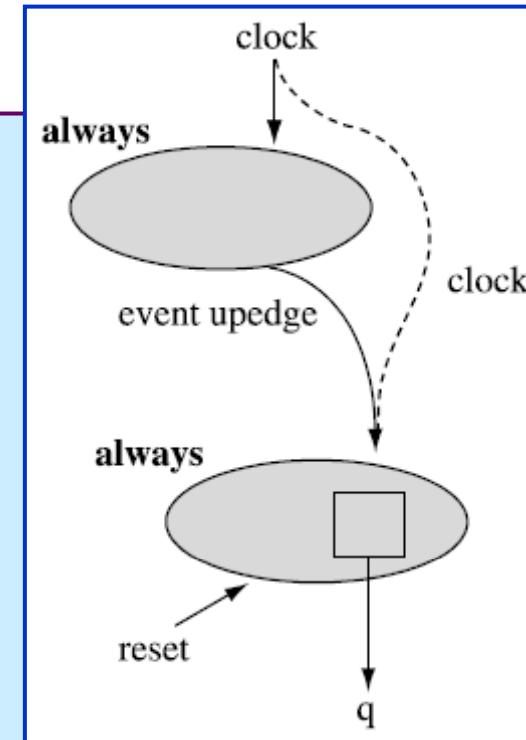
예 5.6.2

named event를 이용한 D 플립플롭

```
module ff_event(clock, reset, din, q);
    input reset, clock, din;
    output q;
    reg q;
    event upedge; // event 선언

    always @(posedge clock) -> upedge;

    always @ (upedge or negedge reset)
    begin
        if(reset==0) q = 0;
        else         q = din;
    end
endmodule
```



코드 5.18

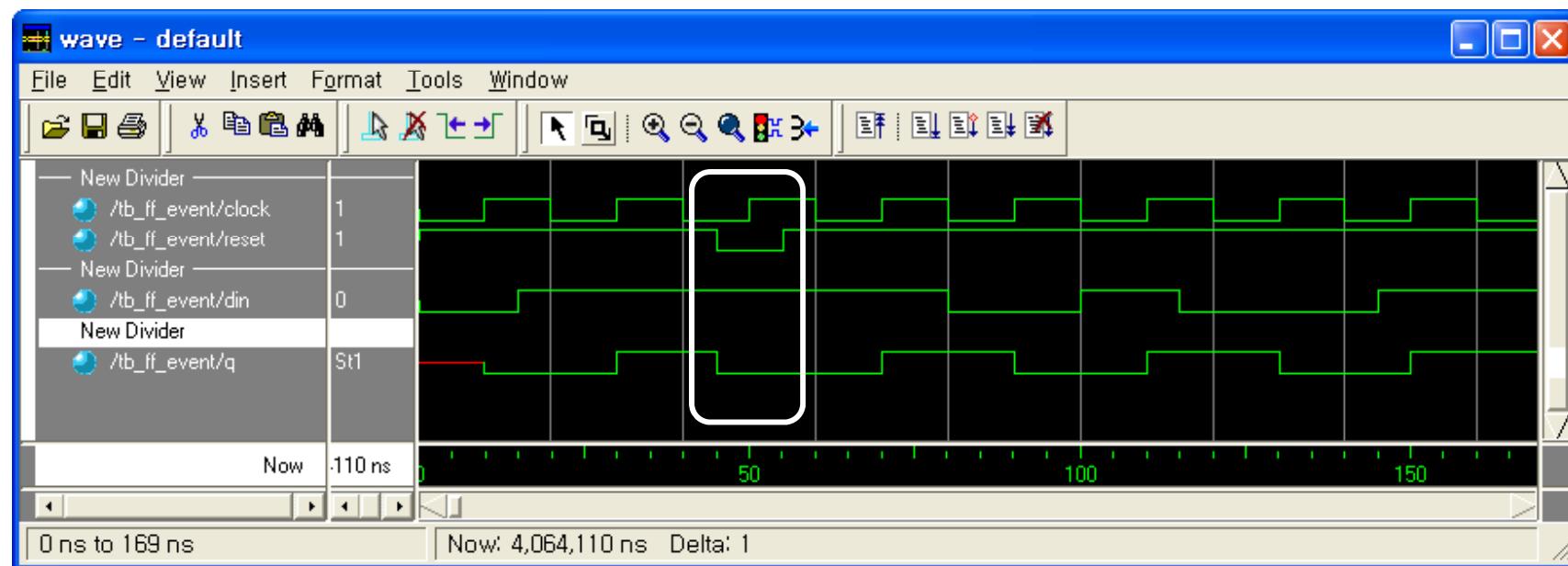


## 5.6 절차형 할당의 타이밍 제어



예 5.6.2

코드 5.18의 시뮬레이션 결과



# 5.6 절차형 할당의 타이밍 제어

## □ event or 연산자

- ❖ 다수의 **event**들은 키워드 **or** 또는 콤마(,)로 결합

```
always @(posedge clk_a or posedge clk_b or trig) rega = regb;  
always @(posedge clk_a, posedge clk_b, trig) rega = regb;
```

## □ wait 문

- ❖ 조건을 평가하여 참이면 **wait** 문에 속하는 절차형 할당문이 실행되며, 조건이 거짓이면 절차형 할당문의 실행이 중지

```
wait(expression) statement_or_null;
```

```
begin  
    wait(!enable) #10 a = b;  
                #10 c = d;  
end
```



# 5.6 절차형 할당의 타이밍 제어

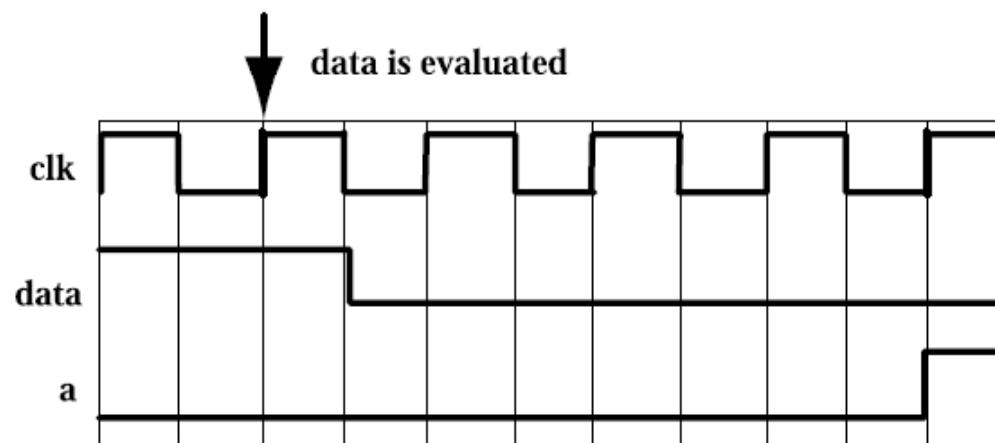
## □ intra-assignment 타이밍 제어

- ❖ 지연과 **event** 제어가 순차 할당문 안에 포함
- ❖ 우변의 수식에 대한 평가를 먼저 실행한 후, 좌변의 객체에 새로운 값이 할당되는 시점을 지정된 지연만큼 지연시킴



예 5.6.4

```
a <= repeat(5) @(posedge clk) data;
```



## 5.6 절차형 할당의 타이밍 제어



### 예 5.6.7 intra-assignment event 제어

```
module intra_delay1(clk, d, out);
    input clk, d;
    output out;
    reg out;
    always @(*)
        out = repeat(3) @(posedge clk) d;
endmodule
```

코드 5.19

등가 코드

```
module intra_delay1_eq(clk, d, out);
    input clk, d;
    output out;
    reg out,temp;
    always @(*) begin
        temp = d;
        @(posedge clk);
        @(posedge clk);
        @(posedge clk) out = temp;
    end
endmodule
```

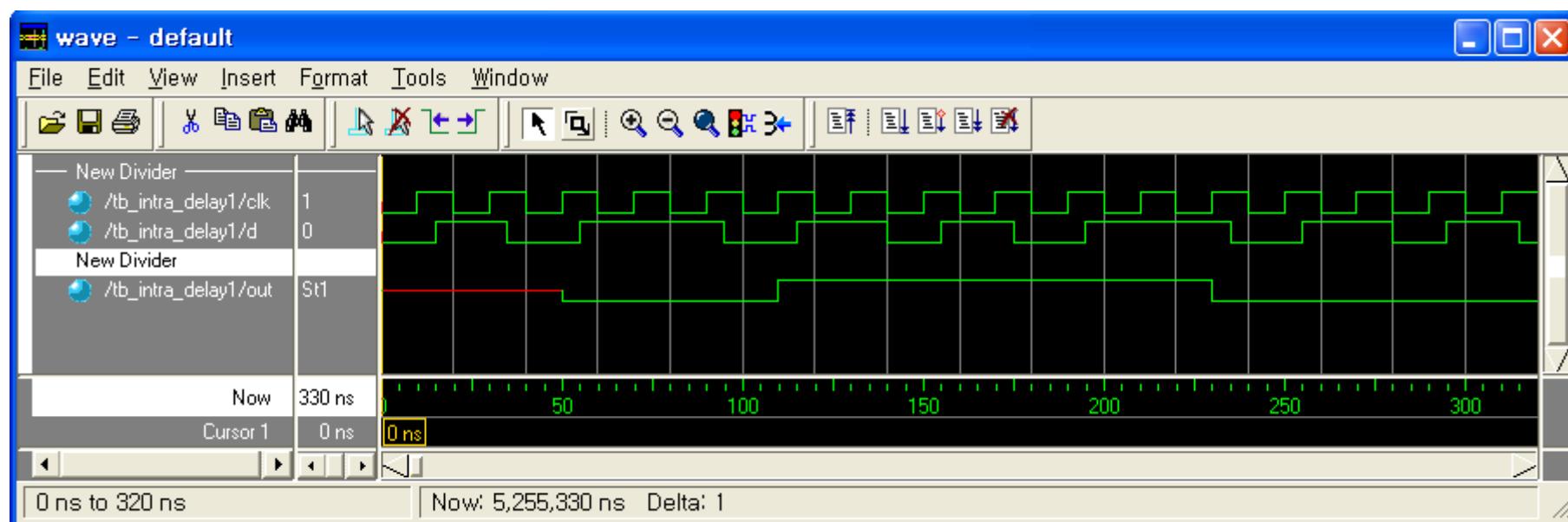
코드 5.20

D-3



# 5.6 절차형 할당의 타이밍 제어

코드 5.19의 시뮬레이션 결과



# 절차형 할당의 타이밍 제어

Equivalent model of Intra – assignment timing control	
With intra – assignment construct	Without intra – assignment construct
<pre>a = #5 b;</pre>	<pre>begin     temp = b;     #5 a = temp; end</pre>
<pre>a = @(posedge clk) b;</pre>	<pre>begin     temp = b;     @(posedge clk) a = temp; end</pre>
<pre>a = repeat(3) @(posedge clk) b;</pre>	<pre>begin     temp = b;     @(posedge clk);     @(posedge clk);     @(posedge clk) a = temp; end</pre>



# 5.7 블록문

## □ 블록문

- ❖ 두 개 이상의 문장을 그룹으로 묶어 구문적으로 하나의 문장처럼 처리
- ❖ **begin – end** 절차형 할당문 블록
  - 문장이 나열된 순서에 의해 순차적으로 실행
  - 시작시간 : 첫번째 문장이 실행될 때
  - 종료시간 : 마지막 문장이 실행될 때
  - 각 절차형 할당문의 실행은 이전 문장의 지연시간에 대해 상대적으로 결정됨
- ❖ **fork – join** 병렬문 블록
  - 문장의 나열 순서에 무관하게 동시에 실행
  - 시작시간 : 블록 내의 모든 문장의 시작 시간이 동일
  - 종료시간 : 각 문장의 지연이 고려되어 시간적으로 마지막에 실행되는 문장의 실행이 완료된 시점
  - 각 문장의 지연은 블록에 들어가는 시뮬레이션 시간을 기준으로 결정됨
  - 시뮬레이션 파형 생성에 유용함



# 5.7 블록문



예 5.7.1

```
parameter d = 50;  
reg [7:0] r;  
begin          // a waveform controlled by sequential delay  
    #d r = 8'h35;  
    #d r = 8'hE2;  
    #d r = 8'h00;  
    #d r = 8'hF7;  
    #d -> end_wave; //trigger an event called end_wave  
end
```



예 5.7.2

**fork**

```
#50  r = 8'h35;  
#100 r = 8'hE2;  
#150 r = 8'h00;  
#200 r = 8'hF7;  
#250 -> end_wave;
```

**join**

**fork**

```
#250 -> end_wave;  
#200 r = 8'hF7;  
#150 r = 8'h00;  
#100 r = 8'hE2;  
#50  r = 8'h35;
```

등가 코드

**join**

D-3



## 6. 구조적 모델링



# 6.1 모듈

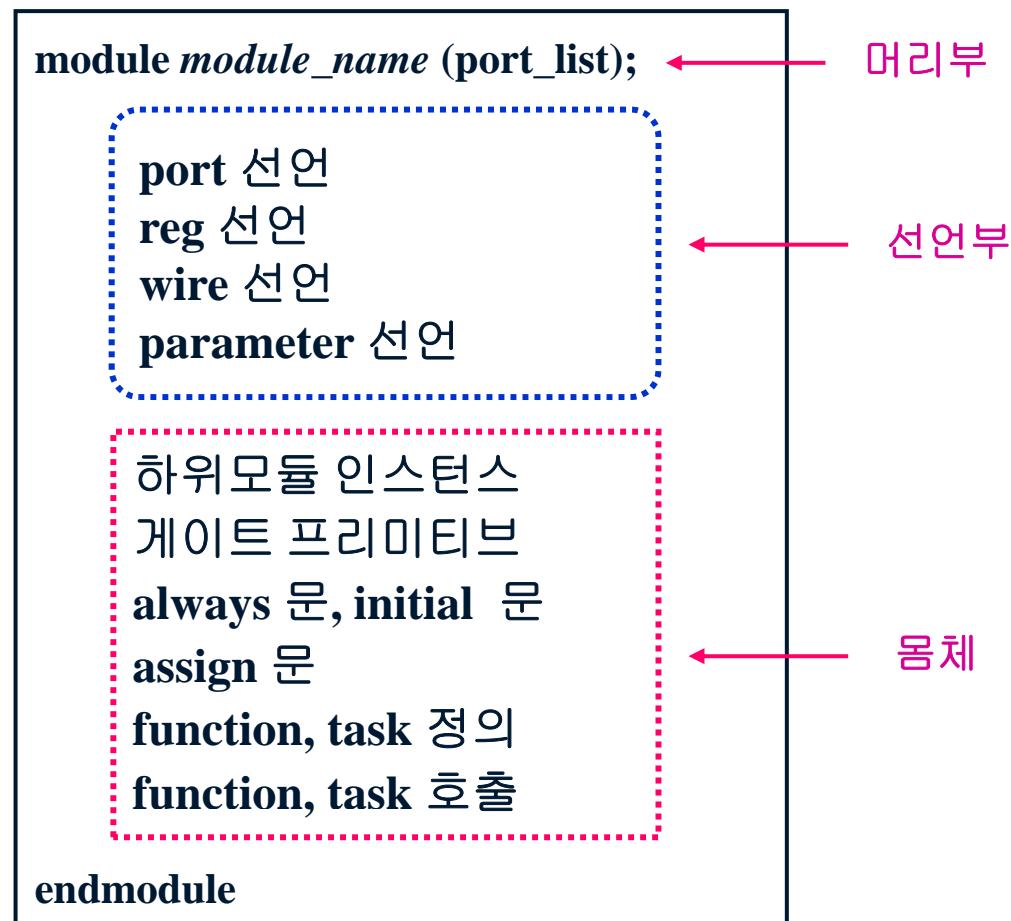


그림 6.1 Verilog 모듈의 구성



## 6.1.1 모듈 포트 선언

### □ 모듈 포트 선언

- ❖ 포트목록에 나열된 포트들은 선언부에서 포트선언을 통해 **input**, **output**, **inout(양방향)**으로 선언
- ❖ **signed**와 범위에 대한 정의를 포함한 포트에 관한 모든 다른 속성들이 포트선언에 포함될 수 있음

```
inout_declaration ::=  
    inout [ net_type ][ signed ][ range ] list_of_port_identifiers  
input_declaration ::=  
    input [ net_type ][ signed ][ range ] list_of_port_identifiers  
output_declaration ::=  
    output [ net_type ][ signed ][ range ] ist_of_port_identifiers  
    | output [ reg ][ signed ][ range ] list_of_port_identifiers  
    | output reg [ signed ][ range ] list_of_variable_port_identifiers  
    | output [ output_variable_type ] list_of_port_identifiers  
    | output output_variable_type list_of_variable_port_identifiers  
list_of_port_identifiers ::=  
    port_identifier { , port_identifier }
```



## 6.1.1 모듈 포트 선언

```
input aport;      // First declaration - okay.  
input aport;      // Error - multiple port declaration  
output aport;     // Error - multiple port declaration
```



### 예 6.1.1 기본적인 모듈 포트선언의 예

```
module test(a, b, c, d, e, f, g, h);  
    input [7:0] a, b;          // no explicit declaration - net is unsigned  
    input signed [7:0] c, d;   // no explicit net declaration - net is signed  
    output [7:0] e, f;        // no explicit declaration - net is unsigned  
    output signed [7:0] g, h; // no explicit net declaration - net is signed  
    wire signed [7:0] b;      // port b inherits signed attribute from net decl.  
    wire [7:0] c;            // net c inherits signed attribute from port  
    reg signed [7:0] f;       // port f inherits signed attribute from reg decl.  
    reg [7:0] g;             // reg g inherits signed attribute from port
```



## 6.1.1 모듈 포트 선언



### 예 6.1.2 모듈 포트를 정의하는 여러 가지 예

```
module complex_ports({c,d}, .e(f));  
// Nets {c,d} receive the first port bits.  
// Name 'f' is declared inside the module.  
// Name 'e' is defined outside the module.  
// Can't use named port connections of first port.
```

```
module split_ports(a[7:4], a[3:0]);  
// First port is upper 4 bits of 'a'.  
// Second port is lower 4 bits of 'a'.  
// Can't use named port connections because of part-select port 'a'.
```

```
module same_port(.a(i), .b(i));  
// Name 'i' is declared inside the module as a inout port.  
// Names 'a' and 'b' are defined for port connections.
```



## 6.1.1 모듈 포트 선언



예 6.1.2

모듈 포트를 정의하는 여러 가지 예

```
module renamed_concat(.a({b,c}), f, .g(h[1]));
// Names 'b', 'c', 'f', 'h' are defined inside the module.
// Names 'a', 'f', 'g' are defined for port connections.
// Can use named port connections.
```

```
module same_input(a, a);
    input a;                      // This is legal. The inputs are ored together.
```

D-3



## 6.1.2 모듈포트 선언목록

### □ 포트 선언목록

- ❖ 포트 선언목록에서 포트를 선언
- ❖ 포트 선언목록으로 선언된 포트들은 모듈의 선언부에서 재선언되지 않음

```
module test( input [7:0] a,  
             input signed [7:0] b, c, d,  
             output [7:0] e,  
             output signed reg [7:0] f, g,  
             output signed [7:0] h ) ;  
  
// illegal to redeclare any ports of the module in the body of the module.
```



## 6.2 모듈 인스턴스

### □ 구조적 모델링

- ❖ 다른 모듈의 인스턴스와 포트 매핑을 통한 모델링
- ❖ 범위 지정을 통한 인스턴스 배열의 생성 가능
- ❖ 모듈 인스턴스 이름은 생략할 수 **없음**
  - 게이트 프리미티브의 인스턴스 이름은 생략 가능

### □ 포트순서에 의한 포트 매핑

- ❖ 모듈의 포트목록에 나열된 포트 순서와 1 : 1로 대응되어 연결
- ❖ 포트에 연결되는 신호가 없는 경우에는 해당 위치를 빈칸으로 남겨 둔다

### □ 포트이름에 의한 포트 매핑

- ❖ 포트 이름과 그 포트에 연결되는 신호 이름을 명시적으로 지정
- ❖ 포트의 비트 선택, 부분 선택, 결합 등을 사용할 수 없음

```
.port_name([expression])
```



## 6.2 모듈 인스턴스



예 6.2.2

### 순서에 의한 포트 매핑

```
module topmod;
    wire [4:0] v;
    wire c, w;

    modB b1(v[0], v[3], w, v[4]);
endmodule
```

코드 6.3

```
module modB(wa, wb, c, d);
    inout wa, wb;
    input c, d;

    tranif1      g1(wa, wb, cinvert);
    not #(2, 6) n1(cinvert, int);
    and #(6, 5) g2(int, c, d);
endmodule
```



예 6.2.4

### 이름에 의한 포트 매핑

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;

    modB b1(.wb(v[3]), .wa(v[0]), .d(v[4]), .c(w));
endmodule
```

코드 6.4



## 6.2 모듈 인스턴스



### 예 6.2.1

인스턴스 배열을 이용한 구조적 모델링

```
module bus_driver(busin, bushigh, buslow, enh, enl);
    input [15:0] busin;
    input enh, enl;
    output [7:0] bushigh, buslow;

    driver busar3(busin[15:12], bushigh[7:4], enh);
    driver busar2(busin[11:8], bushigh[3:0], enh);
    driver busar1(busin[7:4], buslow[7:4], enl);
    driver busar0(busin[3:0], buslow[3:0], enl);
endmodule
```

코드 6.2-(a)

모듈 인스턴스의 배열을 이용

```
driver busar[3:0] (.in(busin),
                     .out({bushigh, buslow}),
                     .en({enh, enh, enl, enl}) );
endmodule
```

코드 6.2-(b)



## 6.2 모듈 인스턴스



예 6.2.5

동일 포트에 대한 다중 연결 오류

```
module test;
    a  U0(.i(a),
          .i(b), // illegal connection of input port twice.
          .o(c),
          .o(d), // illegal connection of output port twice.
          .e(e),
          .e(f)  // illegal connection of inout port twice.
        );
endmodule
```

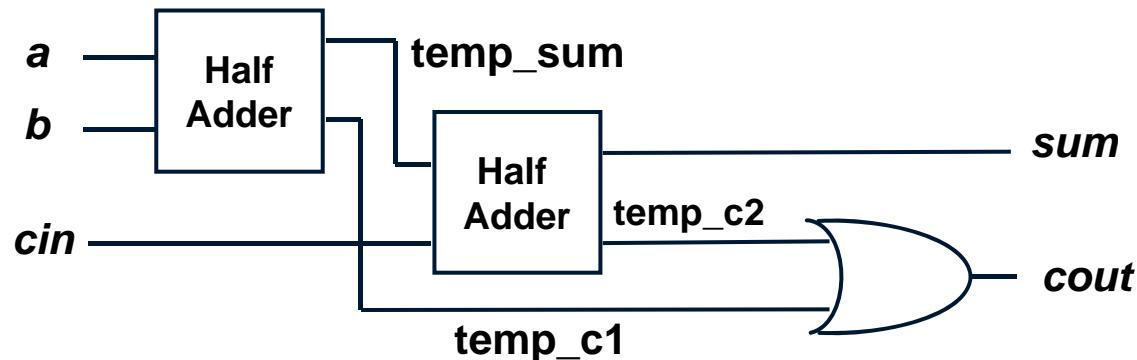
코드 6.4



## 6.2 모듈 인스턴스



예 6.2.6 1비트 전가산기 회로의 구조적 모델링



```
module half_adder(a, b, sum, cout);
    input a, b;
    output sum, cout;
    wire cout_bar; // 생략 가능 (1-bit wire)

    xor U0 (sum, a, b);
    nand (cout_bar, a, b); // 인스턴스 이름 생략 가능 (gate primitive)
    not U1 (cout, cout_bar);
endmodule
```

반가산기 모듈

코드 6.6-(a)

## 6.2 모듈 인스턴스



### 예 6.2.6 1비트 전가산기 회로의 구조적 모델링

```
module full_adder(a, b, cin, sum, cout);
    input a, b, cin;
    output sum, cout;
    wire temp_sum, temp_c1, temp_c2; //생략 가능

    // half_adder 모듈의 instantiation
    half_adder u0(a, b, temp_sum, temp_c1); // 순서에 의한 포트 연결
    half_adder u1(.a(temp_sum),
                  .b(cin),
                  .sum(sum),
                  .cout(temp_c2)); // 이름에 의한 포트 연결
    or u2(cout, temp_c1, temp_c2); // 게이트 프리미티브 인스턴스

endmodule
```

코드 6.6-(b)



## 6.2 모듈 인스턴스



### 예 6.2.6 1비트 전가산기 회로의 구조적 모델링

```
// 1비트 full_adder 모듈의 시뮬레이션 testbench
module tb_full_adder ;
    reg a, b, cin; // initial 블록에서 값을 받으므로 reg로 선언
    integer k;

// full_adder 모듈의 instantiation
    full_adder U0(a, b, cin, sum, cout);

// 시뮬레이션을 위한 파형 생성
initial begin
    forever
        for(k = 0; k < 8; k = k+1) begin
            cin = k/4;
            b   =(k%4)/2;
            a   = k%2;
            #10;
        end
    end
endmodule
```

코드 6.7

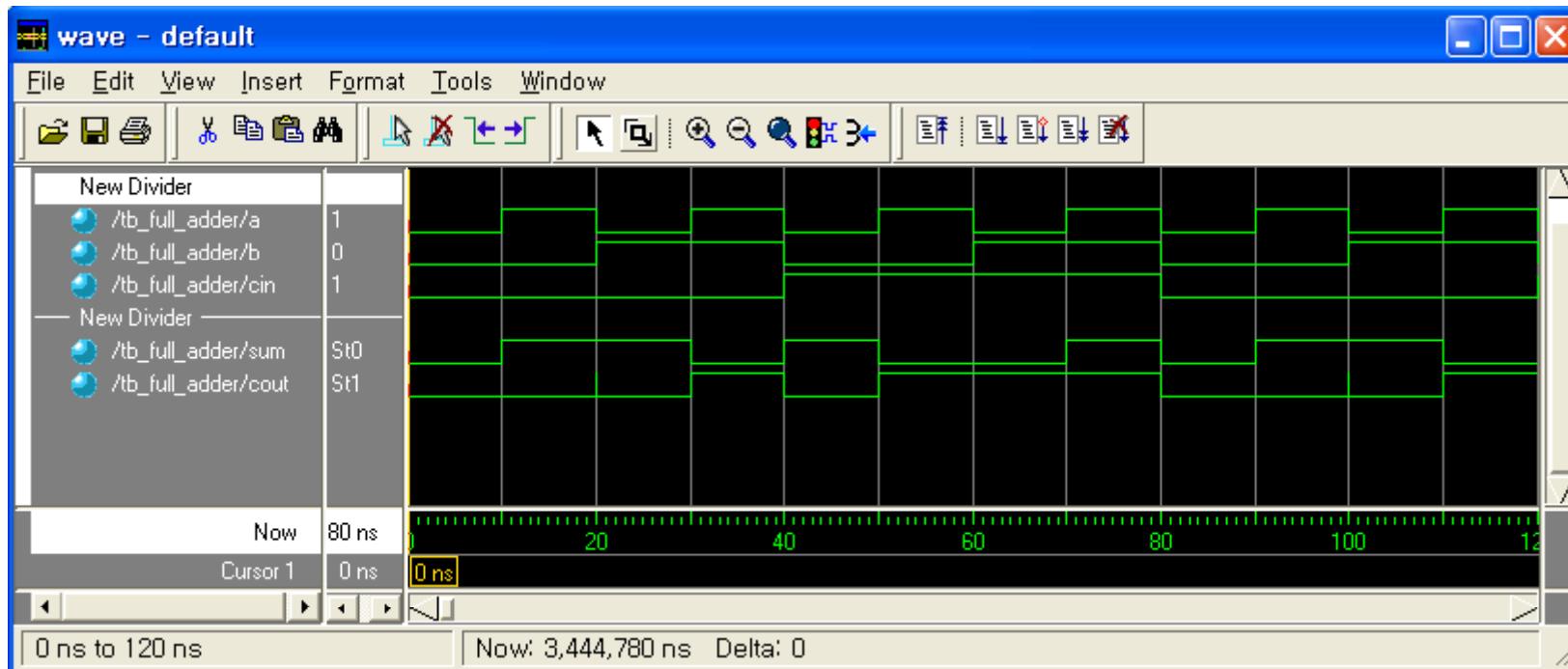


## 6.2 모듈 인스턴스



예 6.2.6

1비트 전가산기 회로의 구조적 모델링



## 6.2 모듈 인스턴스

### □ 실수형 값의 포트 연결

- ❖ **real** 자료형은 직접 포트에 연결될 수 없음
- ❖ 시스템 함수인 **\$realtobit**와 **\$bittoreal**을 통해 **real** 자료형을 비트 자료형으로 변환한 후 모듈 포트에 적용

```
module driver(net_r);
    output net_r;
    real r;
    wire [64:1] net_r = $realtobits(r);
endmodule
```

코드 6.8

```
module receiver(net_r);
    input net_r;
    wire [64:1] net_r;
    real r;

    initial
        assign r = $bitstoreal(net_r);
endmodule
```

코드 6.9



# 6.3 모듈 parameter

## □ 모듈 parameter

- ❖ **variable** 또는 **net** 범주에 속하지 않는 상수
- ❖ 모듈이 인스턴스될 때 값을 변경시킬 수 있음
  - 구조적 설계에서 모듈 인스턴스의 개별화(**customize**)를 위한 유용한 수단
  - **localparam**으로 선언된 **parameter**는 그 값을 변경할 수 없음

## □ 모듈 parameter 값의 변경

- ❖ **defparam** 문을 이용하는 방법
- ❖ 모듈 인스턴스를 통해 **parameter** 값을 변경하는 방법
- ❖ 위의 두 가지 방법에 의한 **parameter** 값 변경이 충돌되는 경우,  
**defparam** 문에 의해 설정된 값이 사용됨



# 6.3 모듈 parameter



예 6.3.1

모듈 parameter 값의 변경

```
module foo(a, b);
    real r1, r2;
    parameter [2:0] A = 3'h2;      // 자료형과 범위가 지정된 parameter 선언
    parameter          B = 3'h2;    // 자료형과 범위가 지정되지 않은 parameter 선언
    initial begin
        r1 = A;
        r2 = B;
        $display("r1 is %f, r2 is %f",r1,r2);
    end
endmodule
```

코드 6.10-(a)

```
module bar;                      // parameter overriding using defparam
    wire a, b;
    defparam U0.A = 3.1415;    // A는 3으로 변경됨
    defparam U0.B = 3.1415;    // B는 3.1415로 변경됨

    foo U0(a, b);            // module instantiation
endmodule
```

코드 6.10-(b)



# 6.3 모듈 parameter

## □ defparam 문

- ❖ **parameter**의 계층적 이름을 사용하여 전체 설계에 걸친 모듈 인스턴스의 **parameter** 값을 변경
  - 인스턴스 배열의 하부 계층 구조에 있는 **defparam** 문은 계층 구조 밖의 **parameter** 값을 변경시킬 수 없음
- ❖ 모든 **parameter** 변경 문들을 함께 묶어 독립된 모듈로 정의할 때 유용



# 모듈 parameter



예 6.3.2

defparam 문에 의한 parameter 값의 변경

```
module top;
    reg         clk;
    reg [0:4] in1;
    reg [0:9] in2;
    wire [0:4] o1;
    wire [0:9] o2;

    vdff m1(o1, in1, clk);
    vdff m2(o2, in2, clk);
endmodule
```

```
module annotate;
    defparam
        top.m1.size = 5,
        top.m1.delay = 10,
        top.m2.size = 10,
        top.m2.delay = 20;
endmodule
```

```
module vdff(out, in, clk);
    parameter size = 1, delay = 1;
    input  [0:size-1] in;
    input  clk;
    output [0:size-1] out;
    reg     [0:size-1] out;

    always @(posedge clk)
        # delay out = in;
endmodule
```

코드 6.10



# 6.3 모듈 parameter

## □ 모듈 인스턴스의 **parameter** 값 변경

- ❖ 순서화된 **parameter** 목록에 의한 방법(**ordered\_parameter\_assignment**)

```
#(expression {, expression,} )
```

- ❖ **parameter** 이름에 의한 방법 (**named\_parameter\_assignment**)

```
#(.parameter_name([expression]))
```

- ❖ 단일 모듈 인스턴스 문에서 이들 두 가지 방법을 혼용하여 사용할 수 없음
- ❖ 기호 # : **delay operator**로도 사용됨



# 6.3 모듈 parameter



예 6.3.3

모듈 parameter 값의 변경

```
module dffn(q, d, clk);
    parameter BITS = 1;
    input [BITS-1:0] d;
    input             clk;
    output [BITS-1:0] q;

    DFF dff_array [BITS-1:0] (q, d, clk); // instance array
endmodule

module MxN_pipeline(in, out, clk);
    parameter M = 3, N = 4;      // M=width,N=depth
    input [M-1:0]     in;
    output [M-1:0]    out;
    input             clk;
    wire   [M*(N-1):1] t;

    // #(M)은 모듈 dffn의 Parameter BITS 값을 재설정
    dffn #(M) p [1:N] ({out, t}, {t, in}, clk);
endmodule
```

코드 6.12

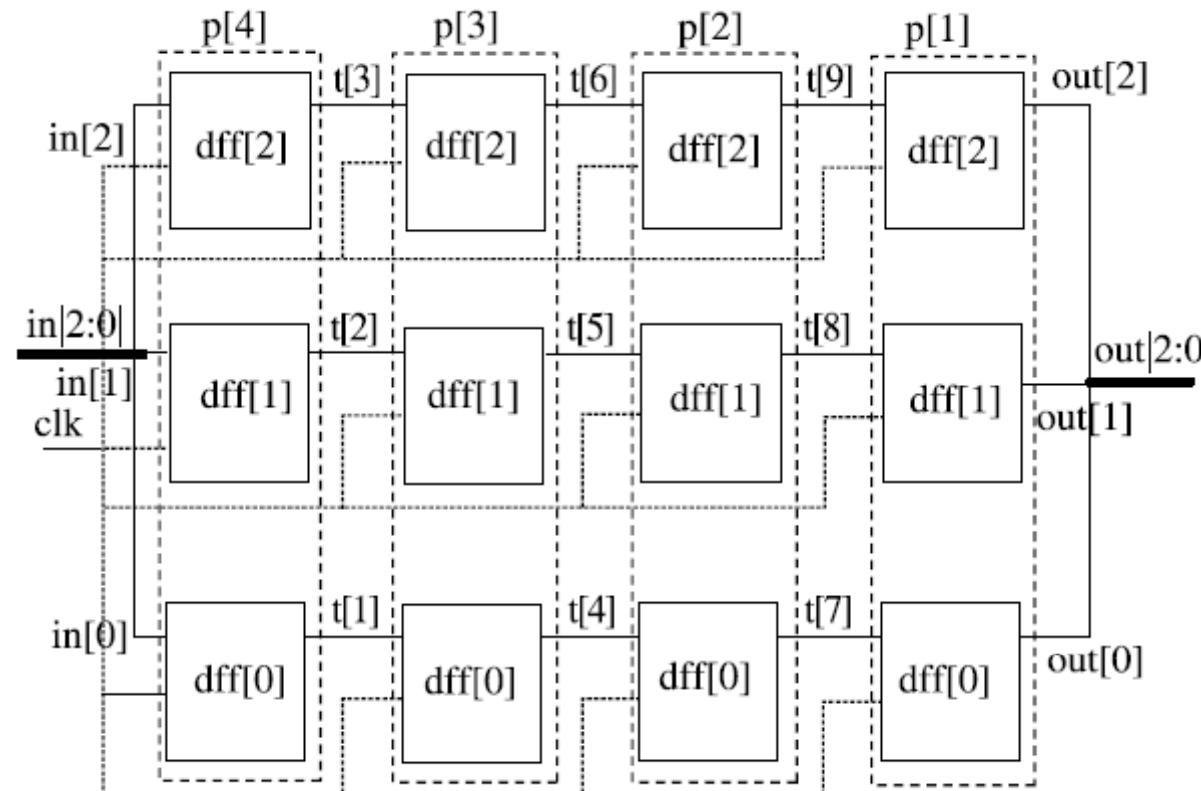


# 6.3 모듈 parameter



예 6.3.3

모듈 parameter 값의 변경



# 6.3 모듈 parameter



예 6.3.4

```
module vdff(out, in, clk);
    parameter size = 5, delay = 1;
    input [0:size-1] in;
    input                 clk;
    output [0:size-1] out;
    reg      [0:size-1] out;

    always @ (posedge clk)
        # delay out = in; // #은 delay operator 입
endmodule
```

코드 6.13-(a)

```
module m;
    reg                  clk;
    wire [0:4]  out_c, in_c;
    wire [1:10] out_a, in_a;
    wire [1:5]  out_b, in_b;

    vdff #(10,15) mod_a(out_a, in_a, clk);
    vdff mod_b(out_b, in_b, clk); // default parameter values
    vdff #(delay(12)) mod_c(out_c, in_c, clk);
endmodule
```

코드 6.13-(b)



## 6.4 생성문

### □ 생성문 (Generate statement)

- ❖ **generate – endgenerate** 블록으로 모델링
  - 반복 생성문 (**generate – for**)
  - 조건 생성문 (**generate – if**)
  - **case** 생성문 (**generate – case**)
- ❖ 모듈, 사용자 정의 프리미티브(**UDP**), 게이트 프리미티브, 연속 할당문, **initial** 블록과 **always** 블록 등의 인스턴스를 하나 또는 다수 개 생성
- ❖ **net**, **reg**, **integer**, **real**, **time**, **realtime**, **event** 등의 자료형 선언 가능
- ❖ 생성된 인스턴스와 자료형은 고유의 식별자를 가지며 계층적으로 참조 가능
- ❖ 순서 또는 이름에 의한 **parameter** 값의 변경 또는 **defparam** 문에 의한 **parameter** 재정의 가능
  - **defparam** 문은 생성 범위 내에서 선언된 **parameter**의 값에만 영향을 미침
- ❖ 생성문에서 **parameter**, **local parameter**, **input**, **output**, **inout**, **specify** **block** 등의 선언은 허용되지 않음



## 6.4.1 genvar 선언

### □ genvar 선언

- ❖ 생성문 내에서만 사용될 수 있는 인덱스 변수의 선언에 사용
- ❖ **genvar**로 선언되는 변수는 정수형
  - 선언된 변수를 인덱스로 갖는 반복 생성문 내에서만 사용되는 지역 변수
  - **genvar**의 값은 **parameter** 값이 참조될 수 있는 어떤 문장에서도 참조 가능
  - 시뮬레이터 또는 논리 합성 툴의 **elaboration** 과정 동안에만 정의되며, 시뮬레이션 또는 합성이 진행되는 동안에는 존재하지 않음
- ❖ **elaboration** 과정 : 시뮬레이션이나 합성을 위해 모듈을 분석하는 과정
  - 구문의 오류 검출, 인스턴스된 모듈의 연결(**link**), **parameter** 값의 전달, 계층적인 참조에 대한 분해 등을 수행하는 과정



## 6.4.2 반복 생성문

### □ 반복 생성문 (generate-for 문)

- ❖ generate – endgenerate 구문 내부에 **for** 문을 사용하여 특정 모듈 또는 블록을 반복적으로 인스턴스
  - **variable** 선언, 모듈, **UDP**, 게이트 프리미티브, 연속 할당문, **initial** 블록, **always** 블록 등을 인스턴스할 수 있음
- ❖ 생성문 내부의 **for – loop**에서 사용되는 인덱스 변수는 **genvar**로 선언
- ❖ **for** 문의 **begin** 뒤에 생성문 블록 식별자 (**:identifier**)를 붙여야 함



## 6.4.2 반복 생성문



예 6.4.1

gray-to-binary 변환기

표 6.2 순환 2진 부호(gray code)와 이진 부호

10진수	순환 2진 부호	이진 부호
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001



## 6.4.2 반복 생성문



예 6.4.1

generate-for 문을 이용한 gray-to-binary 변환기

```
module gray2bin1(bin, gray);
    parameter SIZE = 4;          // this module is parameterizable
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;
    genvar i;

    generate
        for(i=0; i<SIZE; i=i+1) begin :bit
            assign bin[i] = ^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

코드 6.14



## 6.4.2 반복 생성문



예 6.4.2

generate-for 문을 이용한 gray-to-binary 변환기

```
module gray2bin2(bin, gray);
    parameter SIZE = 4;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;
    reg     [SIZE-1:0] bin;
    genvar i;

    generate
        for(i=0; i<SIZE; i=i+1) begin :bit
            always @(gray[SIZE-1:i])
                bin[i] = ^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

코드 6.15

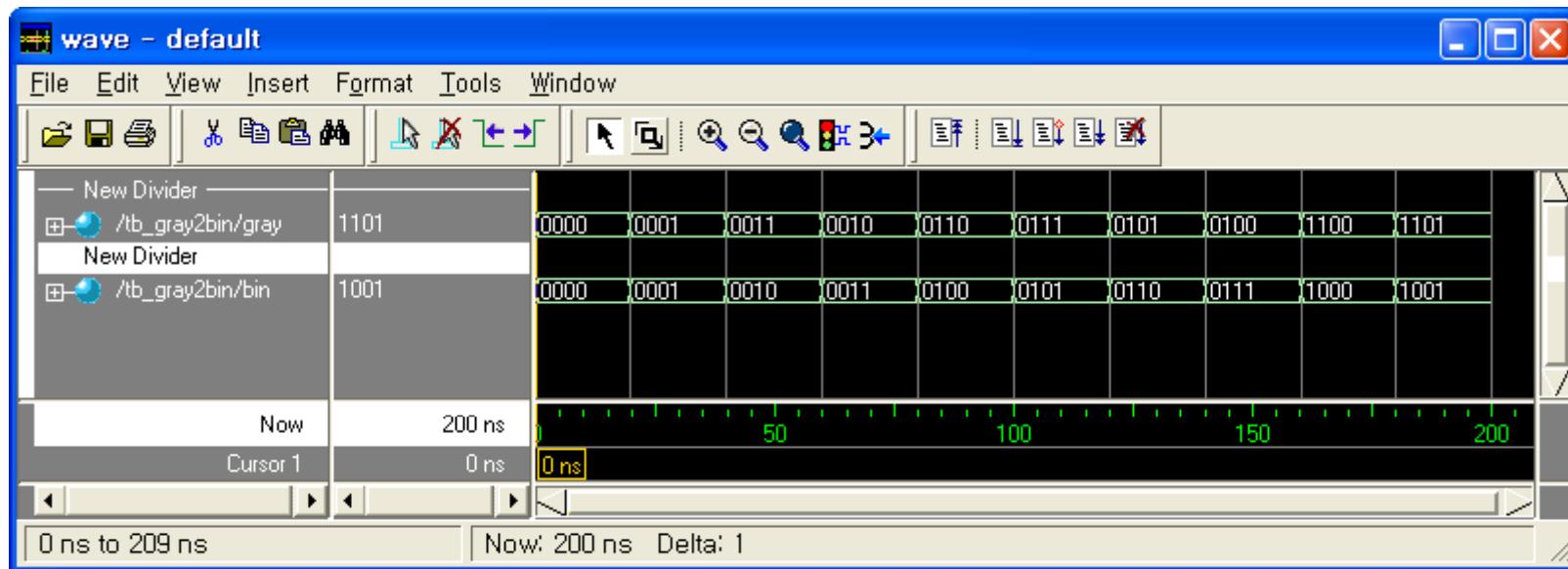


## 6.4.2 반복 생성문



예 6.4.2

generate-for 문을 이용한 gray-to-binary 변환기



## 6.4.2 반복 생성문

참고사항 : **always** 문을 사용하는 경우, **error** 발생

```
module gray2bin_error (bin, gray);
    parameter SIZE = 4;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;
    reg     [SIZE-1:0] bin, tmp;
    integer          i;

    always @(gray) begin
        tmp = gray;
        for (i=0; i<SIZE; i=i+1)
            bin[i] = ^tmp[SIZE-1:i]; // i should be constant
    end
endmodule
```

**always** 문을 사용하는 경우의 올바른 코드는 ?



## 6.4.2 반복 생성문



예 6.4.3

생성루프 외부에 2차원 net 선언을 갖는 ripple-carry 가산기

```
module adder_gen1(co, sum, a, b, ci);
parameter SIZE = 4;
output [SIZE-1:0] sum;
output co;
input [SIZE-1:0] a, b;
input ci;
wire [SIZE :0] c;
wire [SIZE-1:0] t [1:3]; // 2차원 net 선언
genvar i;

assign c[0] = ci;
generate
    for(i=0; i<SIZE; i=i+1) begin :bit
        xor g1( t[1][i], a[i], b[i]);
        xor g2( sum[i], t[1][i], c[i]);
        and g3( t[2][i], a[i], b[i]);
        and g4( t[3][i], t[1][i], c[i]);
        or  g5( c[i+1], t[2][i], t[3][i]);
    end
endgenerate
assign co = c[SIZE];
endmodule
```

코드 6.16

D-3



## 6.4.2 반복 생성문



예 6.4.4

생성루프 내부에 2차원 net 선언을 갖는 ripple-carry 가산기

```
module adder_gen2(co, sum, a, b, ci);
parameter SIZE = 4;
output [SIZE-1:0] sum;
output co;
input [SIZE-1:0] a, b;
input ci;
wire [SIZE :0] c;
genvar i;

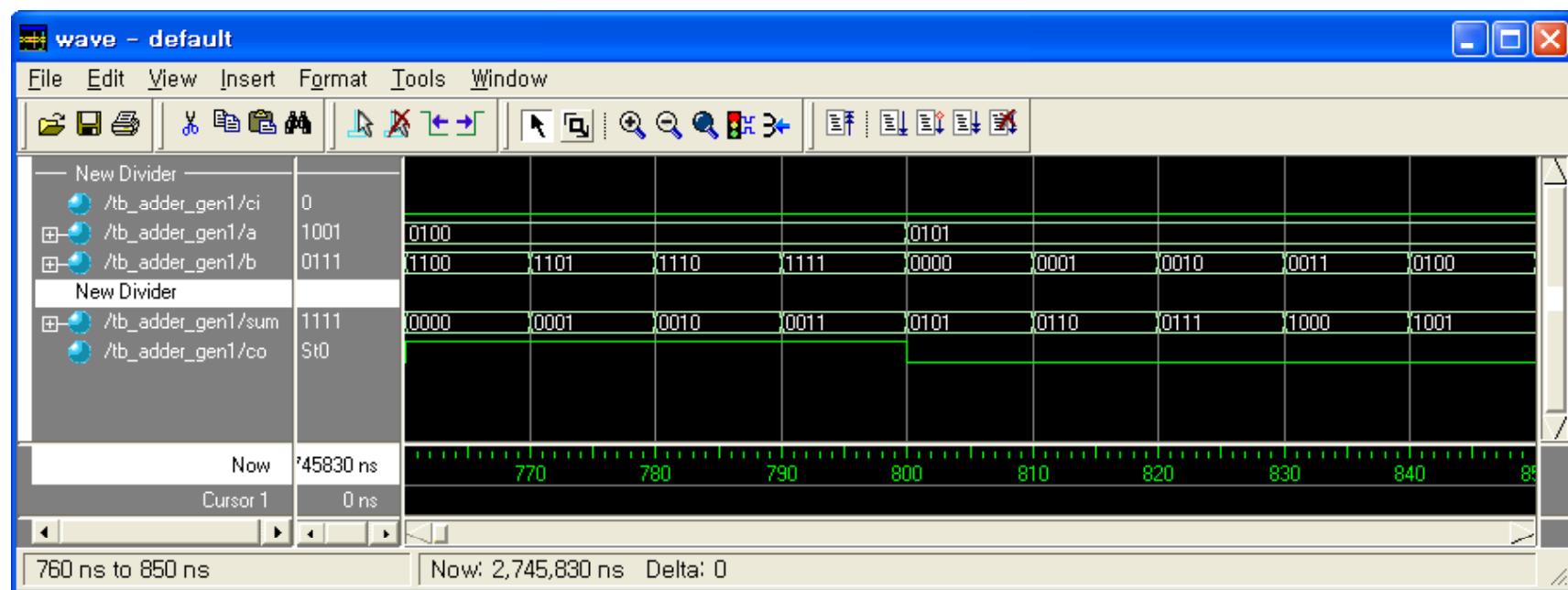
assign c[0] = ci;
generate
    for(i=0; i<SIZE; i=i+1) begin :bit
        wire t1, t2, t3;          // generated net declaration
        xor g1( t1, a[i], b[i]);
        xor g2( sum[i], t1, c[i]);
        and g3( t2, a[i], b[i]);
        and g4( t3, t1, c[i]);
        or  g5( c[i+1], t2, t3);
    end
endgenerate
assign co = c[SIZE];
endmodule
```

코드 6.17



## 6.4.2 반복 생성문

code 6.16, code 6.17의 시뮬레이션 결과



## 6.4.3 조건 생성문

### □ 조건 생성문 (generate-if 문)

- ❖ generate – endgenerate 블록 내에 if – else – if의 조건문을 사용하여 조건에 따라 특정 모듈 또는 블록을 선택적으로 인스턴스
  - 모듈, UDP, 게이트 프리미티브, 연속 할당문, initial 블록, always 블록 등을 인스턴스할 수 있음
- ❖ 인스턴스가 선택되는 조건은 elaboration되는 시점에서의 값에 의해 결정



## 6.4.3 조건 생성문



예 6.4.6

if 생성문을 이용한 파라미터화된 곱셈기

```
module multiplier(a, b, product);
    parameter a_width = 8, b_width = 8;
    localparam product_width = a_width+b_width;
    // can not be modified directly with defparam or
    // module instance statement #
    input [a_width-1:0] a;
    input [b_width-1:0] b;
    output [product_width-1:0] product;

    generate
        if((a_width < 8) ||(b_width < 8))
            CLA_mul #(a_width, b_width) u1(a, b, product);
            // instance a CLA multiplier
        else
            WALLACE_mul #(a_width, b_width) u1(a, b, product);
            // instance a Wallace-tree multiplier
    endgenerate
endmodule
```

코드 6.19

D-3



## 6.4.4 case 생성문

### □ case 생성문 (generate-case 문)

- ❖ generate – endgenerate 블록 내에 case 문을 사용하여 조건에 따라 특정 모듈 또는 블록을 선택적으로 인스턴스
  - 모듈, UDP, 게이트 프리미티브, 연속 할당문, initial 블록, always 블록 등을 인스턴스할 수 있음
- ❖ 인스턴스가 선택되는 조건은 elaboration되는 시점에서의 값에 의해 결정



## 6.4.4 case 생성문



예 6.4.7

3 이하의 비트 폭을 처리하기 위한 case 생성문

```
generate
    case(WIDTH)
        1: adder_1bit x1(co, sum, a, b, ci); // 1-bit adder
        2: adder_2bit x1(co, sum, a, b, ci); // 2-bit adder
    default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
              // carry look-ahead adder
    endcase
endgenerate
```

코드 6.20



## 7. task와 함수



# 7.1 Task와 함수

## □ Task와 함수

- ❖ 반복되는 행위수준 모델링 부분을 독립된 코드의 **task**나 **함수(function)**로 정의하고, 이들을 호출하여 사용
- ❖ 규모가 큰 행위수준 모델링을 작은 부분들로 분할하여 **task** 또는 **함수**로 표현함

## □ 장점

- ❖ 소스코드의 가독성 (**readability**)과 디버깅을 용이하게 함

## □ 종류

- ❖ 사용자 정의 **task**, **함수**
- ❖ 시스템 **task**, **함수**



# 7.1 Task와 함수

Function	Task
<ul style="list-style-type: none"><li>적어도 1개 이상의 <b>input</b> 인수가 필요</li><li><b>output, inout</b> 인수를 가질 수 없음</li><li>항상 1개의 결과값을 <b>return</b> 함</li><li>함수 명을 통해 값을 <b>return</b></li></ul>	<ul style="list-style-type: none"><li>0개 이상의 <b>input, output, inout</b> 인수 포함</li><li>값을 <b>return</b>하지 않을 수 있음</li><li><b>output, inout</b>을 통해 다수개의 값을 <b>return</b></li></ul>
<b>zero delay</b> 를 갖는 조합회로 구현	<ul style="list-style-type: none"><li><b>delay, timing, event</b>를 갖고, <b>multiple output</b>을 반환하는 source code에 적용 가능</li><li><b>task</b>의 <b>I/O</b>에 정의되지 않은 <b>module</b> 내 <b>reg</b>로 선언된 변수 access 가능</li></ul>
다른 function 호출 가능 (task는 불가)	다른 task나 function 호출 가능
<b>zero simulation delay</b>	<b>non-zero simulation time</b> 수행 가능
<b>delay, event, timing control</b> 문 포함 불가	<b>delay, event, timing control</b> 문 포함 가능
<ul style="list-style-type: none"><li><b>variable</b> 자료형만 가질 수 있으며, <b>net</b> 자료형은 가질 수 없음</li><li><b>assign</b> 문, 게이트 프리미티브와 모듈 인스턴스는 사용할 수 없음</li><li><b>initial</b> 문과 <b>always</b> 문을 포함할 수 없음</li><li><b>behavioral</b> 문만 포함 가능함</li></ul>	



# 7.1 Task와 함수

## □ function과 task의 정의 및 호출

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

.... - func_name(...); // 함수 호출
.....
endmodule
```

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

.....
task_name(...); // task 호출
.....
endmodule
```



# 7.2 Task

## □ Task의 정의

```
task task_identifier(task_port_list);
    {block_item_declaration}
    statement;
endtask
```

```
task task_identifier;
    {task_item_declaration} // task의 입력, 출력 인수를 선언
    statement;
endtask
```

## □ Task의 호출

```
task_enable ::= task_identifier [(expression {, expression} )];
```

- ❖ 인수의 순서는 **task** 정의에서 선언된 인수목록의 순서와 일치되어야 함



## 7.2 Task



예 7.2.1

```
task my_task;  
    input a, b;  
    inout c;  
    output d, e;  
begin  
    . . .          // task의 기능을 수행하는 문장들  
    c = foo1;      // 결과 값을 reg 변수에 할당하는 문장들  
    d = foo2;  
    e = foo3;  
end  
endtask
```

선언부에서 인수를 선언

```
task my_task(input a, b, inout c, output d, e);  
begin  
    . . .          // task의 기능을 수행하는 문장들
```

my\_task(v, w, x, y, z); // 인수는 순서대로 매핑

task 호출문



## 7.2 Task

```
module bit_counter (data_word, bit_count);
    input [7:0] data_word;
    output [3:0] bit_count;
    reg [3:0] bit_count;

    always @(data_word)
        count_ones(data_word, bit_count); // task 호출

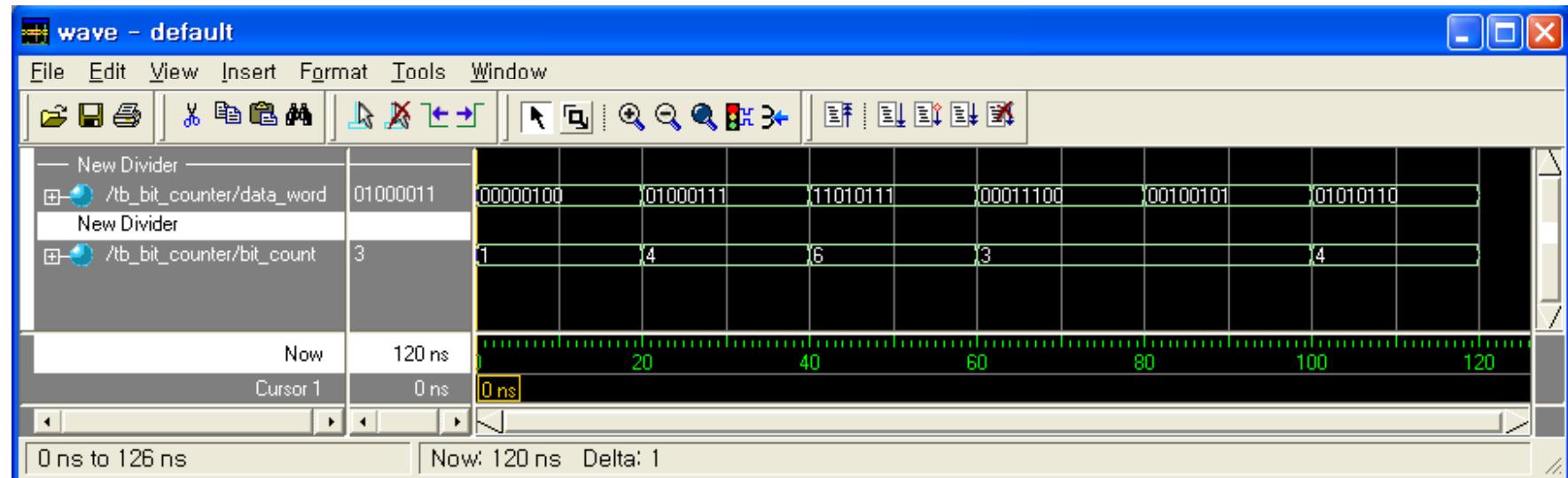
    task count_ones;
        input [7:0] reg_a;      // 인수가 여러 개인 경우, 순서가 중요함.
        output [3:0] count;
        reg [3:0] count;
        reg [7:0] temp_reg;
        begin
            count=0;
            temp_reg=reg_a;
            while (temp_reg) begin
                if (temp_reg[0])
                    count=count+ 1;
                temp_reg = temp_reg >> 1;
            end
        end
    endtask
endmodule
```

예 7.2.2

코드 7.1



## 7.2 Task



코드 7.1의 시뮬레이션 결과



## 7.2 Task



예 7.2.2

```
module traffic_lights;
    reg clock, red, amber, green;
    parameter on = 1, off = 0, red_tics = 350,
              amber_tics = 30, green_tics = 200;
//initialize colors.
    initial red = off;
    initial amber = off;
    initial green = off;

    always begin                  // sequence to control the lights.
        red = on;                // turn red light on and wait.
        light(red, red_tics);
        green = on;              // turn green light on and wait.
        light(green, green_tics);
        amber = on;              // turn amber light on and wait.
        light(amber, amber_tics);
    end

```

코드 7.2

D-3



## 7.2 Task



예 7.2.2

```
// task to wait for 'tics' positive edge clocks before turning 'color' light off.  
task light;  
    output color;  
    input [31:0] tics;  
begin  
    repeat(tics) @(posedge clock);  
        color = off;          // turn light off.  
    end  
endtask  
  
always begin                      // waveform for the clock.  
    #100 clock = 0;  
    #100 clock = 1;  
end  
endmodule
```

코드 7.2



## 7.3 함수

### □ 함수의 정의

```
function [signed][range_type] function_identifier(function_port_list);
{block_item_declaration}
statement;
endfunction
```

```
function [signed][range_type] function_identifier ;
{function_item_declaration}
statement;
endfunction
```

- ❖ 함수 결과값의 속성 및 범위 지정 : **[signed][range\_type]**
  - 별도의 지정이 없으면, **default**로 1비트 **reg**형이 됨
- ❖ 함수가 정의되면 함수이름과 동일한 이름의 변수가 함수 내부에 선언됨
  - 함수가 정의되는 영역에서 함수이름과 동일한 이름의 다른 객체를 선언하는 것이 허용되지 않음



# 7.3 함수

## □ 함수의 호출

```
ftn_call ::= hierarchical_ftn_identifier{attribute_inst}(expr {,expr})
```

## □ 함수의 규칙

- ❖ 함수는 적어도 하나 이상의 입력인수를 포함해야 한다.
- ❖ 함수는 **output**과 **inout**으로 선언된 어떠한 인수도 가질 수 없다.
- ❖ 함수는 시간제어 문장을 가질 수 없다.
  - 함수내부의 어떠한 구문도 타이밍 제어(#, @ 또는 wait 등)를 포함할 수 없다.
- ❖ 함수는 **task**를 호출할 수 없다.
- ❖ 함수는 함수이름과 동일한 이름의 내부변수에 함수의 결과 값을 할당하는 문장을 포함한다.
- ❖ 함수는 **nonblocking** 할당을 가질 수 없다.



## 7.3 함수



예 7.3.1

```
function [7:0] getbyte;  
    input [15:0] address;  
    begin // code to extract low-order byte from addressed word  
        . . .  
        getbyte = result_expression;  
    end  
endfunction
```

선언부에서 인수를 선언

```
function [7:0] getbyte(input [15:0] address);  
    begin // code to extract low-order byte from addressed word  
        . . .  
        getbyte = result_expression;  
    end  
endfunction
```



예 7.3.2

함수 호출

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;
```



## 7.3 함수



예 7.3.3

```
module word_aligner (word_in, word_out);
    input [7:0] word_in;
    output [7:0] word_out;

    function [7:0] word_align;
        input [7:0] word;
        begin
            word_align = word;
            if (word_align != 0)
                while (word_align[7] == 0) word_align=word_align << 1;
        end
    endfunction

    assign word_out = word_align (word_in); // 함수 호출

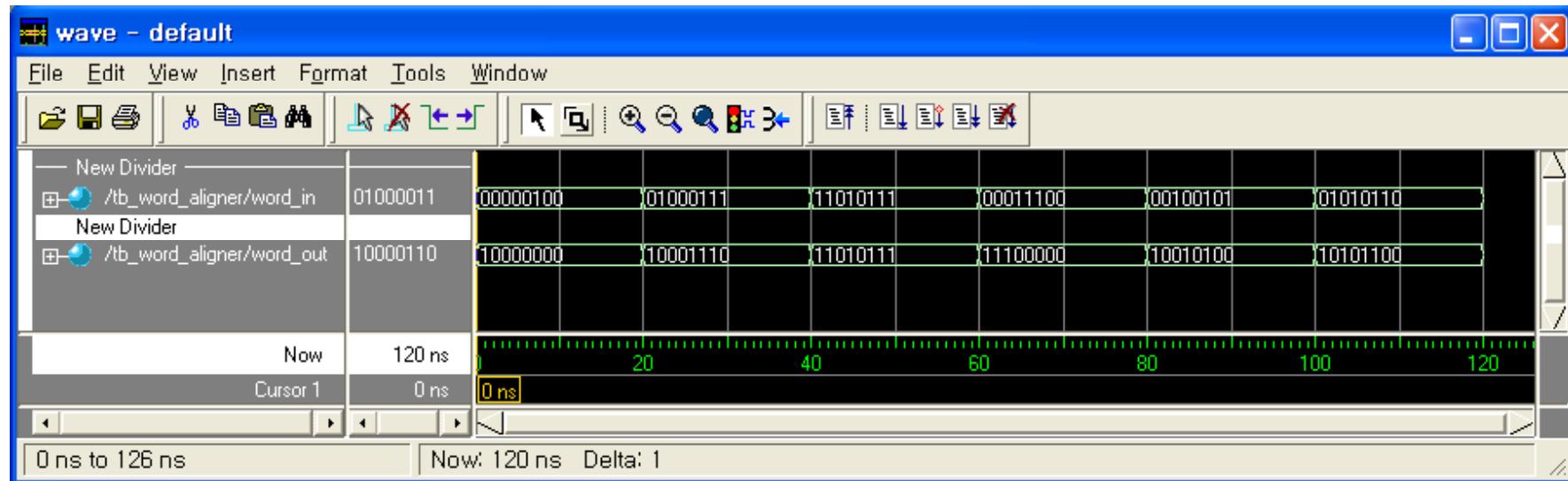
endmodule
```

코드 7.3

D-3



# 7.3 함수



코드 7.3의 시뮬레이션 결과



## 7.3 함수

```
module tryfact;
// define the function
    function automatic integer factorial;
        input [31:0] operand;
        integer i;
        if(operand >= 2)
            factorial = factorial(operand - 1) * operand;
        else
            factorial = 1;
    endfunction

// test the function
integer result;
integer n;
initial begin
    for(n = 0; n <= 7; n = n+1) begin
        result = factorial(n);
        $display("%0d factorial=%0d", n, result);
    end
end
endmodule
```



예 7.3.4

실행 결과

0 factorial=1  
1 factorial=1  
2 factorial=2  
3 factorial=6  
4 factorial=24  
5 factorial=120  
6 factorial=720  
7 factorial=5040

코드 7.4

D-3



## 8. 컴파일러 지시어



# 컴파일러 지시어

## □ 컴파일러 지시어 (Compiler directive)

- ❖ Verilog 소스코드의 컴파일 과정에 영향을 미치는 명령어
- ❖ accent grave 문자(`)로 시작된 위치에서부터 다른 컴파일러 지시어에 의해 대체되거나 또는 컴파일이 완료되기까지 전체 소스파일에 영향을 미침
  - 단일인용부호(')와 혼동하지 않도록 주의

표 8.1 Verilog 컴파일러 지시어

'celldefine	'ifndef
'default_nettype	'include
'define	'line
'else	'nounconnected_drive
'elsif	'resetall
'endcelldefine	'timescale
'endif	'unconnected_drive
'ifdef	'undef



# 8.1 `define과 `undefine

## □ `define

- ❖ 문자 치환을 위한 문자 매크로 생성에 사용
  - C 프로그래밍 언어의 `#define`과 유사
- ❖ 문자 매크로 치환
  - 자주 사용되는 문자 또는 상수를 의미 있는 이름으로 정의하고, 문자 매크로 치환을 통해 정의된 값을 사용
  - 특정 상수 (비트 폭, 지연 등)가 회로 모델링 전체에 걸쳐 반복적으로 사용되는 경우에, 문자 매크로 정의를 사용하면 소스코드의 수정 및 관리가 용이
  - `'macro_name'`을 이용하여 소스코드에 사용
- ❖ 인수를 갖는 문자 매크로를 정의할 수 있으며, 매크로 사용의 개별화가 가능

```
'define wait_state 3'b010          // No semicolon
```



# 8.1 `define과 `undefine

```
text_macro_definition ::=  
    `define text_macro_name macro_text  
text_macro_name ::=  
    text_macro_identifier [ ( list_of_formal_arguments ) ]  
list_of_formal_arguments ::=  
    formal_argument_identifier { , formal_argument_identifier }  
text_macro_identifier ::=  
    simple_identifier
```

문자 매크로 정의

```
text_macro_usage ::=  
    `text_macro_identifier [ ( list_of_actual_arguments ) ]  
list_of_actual_arguments ::=  
    actual_argument { , actual_argument }  
actual_argument ::=  
    expression
```

문자 매크로 사용



# 8.1 `define과 `undefine



예 8.1.1

```
'define wordsize 8  
reg [1:'wordsize] data;
```



예 8.1.2

```
//define a nand with variable delay  
'define var_nand(dly) nand #dly  
  
'var_nand(2) g121(q21, n10, n11);  
'var_nand(5) g122(q22, n10, n11);
```

인수를 갖는 문자 매크로



# 8.1 `define과 `undefine

## □ `undefined

- ❖ 이전에 정의된 문자 매크로를 해제
  - `define으로 정의되지 않은 문자 매크로에 대해 `undef를 정의하면 컴파일 시 경고 메시지가 출력

```
undefined_compiler_directive ::=  
    `undef text_macro_identifier
```



## 8.2 `ifdef, `else, `elsif 등

### □ `ifdef, `else, `elsif, `endif, `ifndef

❖ Verilog HDL 소스코드의 일부를 조건적으로 컴파일하기 위해 사용

- 소스의 어느 곳에서든 사용될 수 있음
- 모듈에 대한 여러 가지(행위적 모델링, 구조적 모델링, 스위치 수준 모델링 등) 모델링 중 하나를 선택하여 컴파일하는 경우
- 여러 가지의 타이밍 정보나 구조적 정보(비트 폭 등) 중 하나를 선택하여 컴파일하는 경우
- 시뮬레이션을 위해 여러 가지 입력 벡터들 중 하나를 선택하는 경우

❖ `ifdef

- 매크로 이름이 정의되어 있으면, `ifdef 이후의 소스코드가 컴파일에 포함
- 매크로 이름이 정의되어 있지 않고 `else가 있으면, `else 이후의 소스코드가 컴파일에 포함

❖ `endif

- 조건적으로 컴파일되는 소스코드의 경계를 나타냄



## 8.2 `ifdef, `else, `elsif 등

### ❖ `ifndef

- 매크로 이름이 정의되어 있지 않으면, `ifndef 이후의 코드가 컴파일에 포함
- 매크로 이름이 정의되어 있고 `else 지시어가 있으면, `else 이후의 코드가 컴파일에 포함

```
conditional_compilation_directive ::=  
    ifdef_directive | ifndef_directive  
ifdef_directive ::=  
    `ifdef text_macro_identifier  
    ifdef_group_of_lines  
    { `elsif text_macro_identifier elsif_group_of_lines }  
    [ `else else_group_of_lines ]  
    `endif  
ifndef_directive ::=  
    `ifndef text_macro_identifier  
    ifndef_group_of_lines  
    { `elsif text_macro_identifier elsif_group_of_lines }  
    [ `else else_group_of_lines ]  
    `endif
```



## 8.2 `ifdef, `else, `elsif 등



예 8.2.1

```
module and_op(a, b, c);
    output a;
    input b, c;

    `ifdef behavioral
        wire a = b & c;
    `else
        and a1(a,b,c);
    `endif
endmodule
```

코드 8.1



## 8.2 `ifdef, `else, `elsif 등

```
module test(out);
    output out;
`define wow
`define nest_one
`define second_nest
`define nest_two

`ifdef wow
    initial $display("wow is defined");
    `ifdef nest_one
        initial $display("nest_one is defined");
        `ifdef nest_two
            initial $display("nest_two is defined");
        `else
            initial $display("nest_two is not defined");
        `endif
    `else
        initial $display("nest_one is not defined");
    `endif
`else
    initial $display("wow is not defined");
    `ifdef second_nest
        initial $display("nest_two is defined");
    `else
        initial $display("nest_two is not defined");
    `endif
`endif
endmodule
```



예 8.2.2

코드 8.2



# 8.3 `include

## □ `include

- ❖ 파일 삽입 컴파일러 지시어
  - 소스파일의 내용 전체를 다른 소스파일에 포함시켜 컴파일할 때 사용
  - 자주 사용되는 정의나 task 등을 별도의 파일에 만든 후, 컴파일 과정에서 포함시킬 때 사용
  - Verilog 소스코드 내의 어느 곳에서든지 정의될 수 있음
  - 삽입되는 파일은 또 다른 `include` 컴파일러 지시어를 포함할 수 있음
  - 설계 전체에 대한 구성 관리를 통합적으로 처리할 수 있음
  - Verilog 소스코드의 효율적인 구성 및 관리 가능

```
include_compiler_directive ::=  
    'include "filename"
```

```
'include "parts/count.v"  
'include "fileB" // including fileB
```



예 8.3.1



# 8.6 `timescale

## □ 'timescale

- ❖ 시간과 지연 값의 측정단위와 정밀도를 지정
  - 다른 'timescale' 지시어가 나타나기 전까지 효력을 유지

```
timescale_compiler_directive ::=  
  'timescale time_unit / time_precision
```

표 8.2 시간정밀도 단위

Character string	Unit of measurement
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds



# 8.6 `timescale



예 8.6.1

```
'timescale 1 ns / 1 ps
```

```
'timescale 10 us / 100 ns
```



예 8.6.2

```
'timescale 10 ns / 1 ns
module test;
    reg set;
    parameter d = 1.55; //16ns의 자연으로 변환
    initial begin
        #d set = 0;          //16ns에서 set에 0이 할당
        #d set = 1;          //32ns에서 set에 1이 할당
    end
endmodule
```

코드 8.3

D-3



# 8.6 `timescale

Timescale directive (unit/precision)	Delay specification	Simulator time step	delay value used in sim.	Conversion
1 ns / ns	#4	1 ns	4 ns	4 x 1
1 ns / 100 ps	#4	100 ps	4.0 ns	4 x 1
10 ns / 100 ps	#4	100 ps	40.0 ns	4 x 10
10 ns / ns	#4	1 ns	40 ns	4 x 10
100 ns / ns	#4	1 ns	400 ns	4 x 100
10 ns / 100 ps	#4.629	100 ps	46.3 ns	round 46.29 to 46.3
10 ns / 1 ns	#4.629	1 ns	46 ns	round 46.29 to 46
10 ns / 10 ns	#4.629	10 ns	50 ns	round 46.29 to 50

