구조적 모델링(3)

Kyung-Wook Shin
kwshin@kumoh.ac.kr

School of Electronic Eng.,
Kumoh National Institute of Technology

6.4 생성문

- 생성문 (Generate statement); Verilog 2001 표준
  - `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 parameter의 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
  - genvar로 선언되는 변수는 정수형
    - 선언된 변수를 인덱스로 갖는 반복 생성문 내에서만 사용되는 지역 변수
    - genvar의 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

```
genvar_declaration ::=
   genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::=
   genvar_identifier { , genvar_identifier }
```


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 – loop에서 사용되는 인덱스 변수는 genvar로 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 6.1 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**

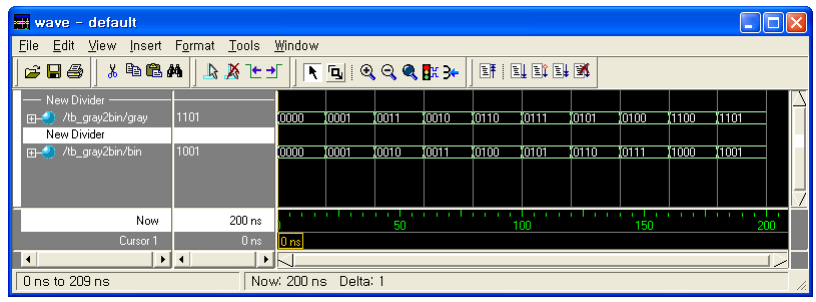
```
module gray2bin2(bin, gray);
   parameter SIZE = 4;
   output [SIZE-1:0] bin;
   input  [SIZE-1:0] gray;
   reg    [SIZE-1:0] bin;
   genvar i;

   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

참고 : 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];
// Range must be bounded by constant expressions
       end
endmodule
```

always 문을 사용하는 경우의 올바른 코드는 ?

6.4.2 반복 생성문

참고 : always 문을 사용하는 경우

```
module gray2bin_4b(bin, gray);
  parameter SIZE = 4;
        output [SIZE-1:0] bin;
        input  [SIZE-1:0] gray;
        reg    [SIZE-1:0] bin, cnt;
  integer i,j;

  always @ (gray) begin
    for(i=0;i<4;i=i+1) begin
      cnt=0;
      for(j=i;j<4;j=j+1)
        cnt[i]=gray[j]^cnt[i];
      bin[i]=cnt[i];
    end
  end
endmodule
```

6.4.2 반복 생성문

예 6.4.3 생성루프 외부에 2차원 net 선언을 갖는 파라미터화된 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

6.4.2 반복 생성문

예 6.4.4 생성루프 내부에 2차원 net 선언을 갖는 파라미터화된 ripple-carry 가산기

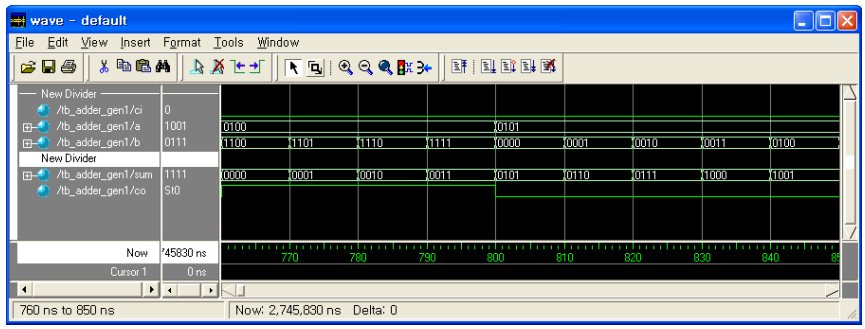
```
module adder_gen2(co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output            co;
  input  [SIZE-1:0] a, b;
  input             ci;
  wire   [SIZE  :0] c;
  genvar            i;

  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

코드 6.16, 코드 6.17의 시뮬레이션 결과



6.4.3 조건 생성문

- 조건 생성문 (`generate-if` 문)
  - `generate-endgenerate` 블록 내에 `if-else-if`의 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 조건 생성문을 이용한 파라미터화된 곱셈기 모듈

```
module multiplier(a, b, product);
   parameter a_width = 8, b_width = 8;
   localparam product_width = a_width+b_width;
// localparam can not be modified directly with defparam or
// module instance statement using #
   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

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 3비트 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                  // The generated instance name is x1
endgenerate
```

코드 6.20