NTC FPGA 강좌 0. Verilog HDL 문법

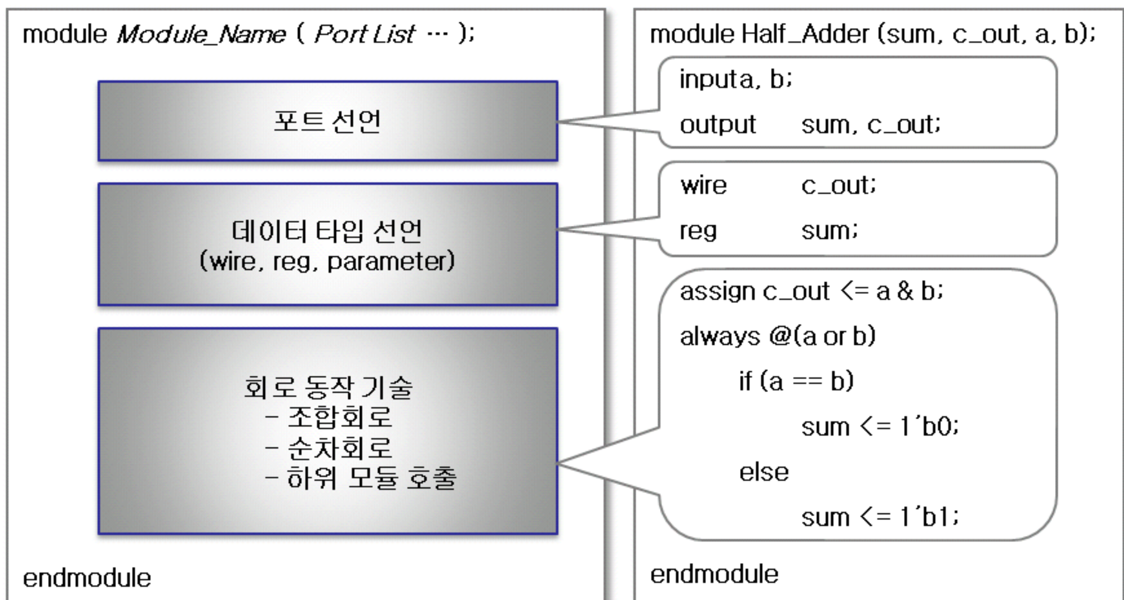
(주) 뉴티씨 (NewTC)
http://www.NewTC.co.kr

1 서론

본 강좌에서는 FPGA를 사용하는데 필요한 기본적인 문법에 관하여 설명하겠습니다. Verilog 코드 작성시 참고할 수 있도록 작성된 것이기 때문에 문법을 공부하는데는 부족할 수 있습니다. 자세한 문법은 강의 또는 참고 도서를 참고하시기 바랍니다.

2 모듈의 구조

모듈(module)은 Verilog HDL에서 시스템을 표현하는 기본 구성요소입니다. 상위 계층에서는 하위 계층의 모듈을 불러서 사용할 수 있습니다. 모듈의 구조는 아래와 같습니다.



주석문

한줄 주석문 : // comment text…

블록 주석문 : /* comment text */

모듈은 크게 회로 생성을 위한 모듈과 시뮬레이션을 위한 모듈로 구분할 수 있습니다.

회로 생성을 위한 모듈	시뮬레이션을 위한 모듈
회로 생성을 위한 구문만 사용할 수 있습니다. Ex) always, assign, function 일부 루프문 (1) Ex) for	모든 Verilog HDL 문법을 사용할 수 있습니다. (테스트 벤치를 만들 때 사용) Ex) initial, always, assign, function, task 모든 루프문 Ex) for, forever, while 등 시스템 태스크문 Ex) $monitor, $fopen 등

(1) 합성 툴에 따라 지원하는 경우가 있으며 루프 횟수에 제한이 있습니다.)

3 논리/수치 표현

3.1 데이터형

Verilog 에서 사용하는 데이터 형은 레지스터(reg)와 와이어(wire), 그리고 파라미터(parameter)가 있습니다.

3.1.1 레지스터

데이터를 저장할 수 있습니다. 레지스터에 새로운 값이 들어오기 전까지는 데이터를 유지합니다. 실제 구현은 Flip-flop 또는 Latch 로 구현됩니다.
always 구문에서 값을 인가하는 경우 레지스터로 선언해야 합니다.

```
Ex)   reg          sum;          // 1비트 레지스터
      reg   [7:0]  bus;          // 8비트 레지스터
```

3.1.2 Wire

Wire는 회로를 연결하는 모든 선입니다. 데이터를 전달할 수는 있지만 저장하지는 못합니다. assign 문을 이용하여 값을 인가할 수 있습니다.
버스 형태의 레지스터 또는 Wire 등을 와이어로 재정의 하여 사용할 수 있습니다. 1비트 Wire 는 생략 가능합니다.

```
Ex)   wire         c_out;          // 1비트 wire
      wire  [7:0]  data;           // 8비트 wire
      wire         msb = data[7];  // data[7]를 msb 로 재정의
```

3.1.3 Parameter

Parameter를 이용하여 모듈 내에서 사용하는 상수를 정의할 수 있습니다. 합성 또는 시뮬레이션 시에 값이 치환됩니다. 많이 사용되는 상수의 경우 숫자를 직접 사용하는 것보다 파라미터를 이용하는 것이 좋습니다. 하위모듈의 파라미터를 바꿀 경우 defparam 을 이용하여 바꿀 수 있습니다.

```
Ex)  parameter       SIZE = 16;              // 정수로 파라미터 정의
     reg [SIZE-1:0]  A_BUS;                  // 파라미터 값을 이용
     parameter       S_IDLE = 4'b0001;       // 4비트로 파라미터 정의
```

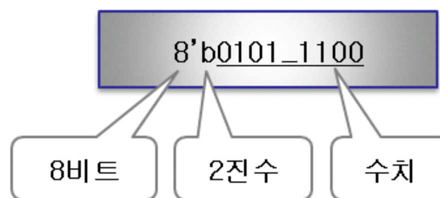
3.2 논리/수치 표현

Verilog에서는 논리 값을 0, 1, x, z 로 나타낼 수 있습니다. 0, 1은 각각 논리 0, 논리 1이며 x 는 알 수 없는 값을 나타냅니다. z 는 하이 임피던스 상태로 값이 인가되지 않는 상태입니다.

```
Ex) data = 8'b0000_zzzz      // data 변수의 하위 4비트에 하이임피던스 인가
```

수치 표현

수치 표현은 <비트폭>'<기수><수치> 와 같이 할 수 있습니다. 비트폭은 10진수로 표현하며 기수는 b (2진수), o (8진수), d (10진수), h (16진수) 가 있습니다. 마지막 수치는 기수에서 지정된 수치로 표현합니다.



아래 3개의 표현식은 data 변수에 같은 값을 저장합니다.

```
data = 8'b0011_0101;    // 53 (2진수)
data = 8'd53;           // 53 (10진수)
data = 8'h35;           // 53 (16진수)
```

비트의 구분을 위해 언더라인 "_" 문자를 추가할 수 있습니다.

4 산술/논리 연산의 종류

4.1 산술 연산

덧셈(+), 뺄셈(-), 곱셈(*), 나눗셈(/), 나머지(%) 와 같은 기본적인 산술 연산을 지원합니다. 곱셈, 나눗셈, 나머지의 경우 합성도 가능하지만 게이트를 많이 차지하는 단점이 있습니다. 따라서 꼭 필요한 경우에만 사용하는 것이 좋습니다.

음수의 경우 "-"를 이용하여 표현할 수 있습니다.

4.2 논리 연산

4.2.1 논리(조건) 연산자

And (&&), Or (||), Not (!) 이 있습니다. “&&”, “||” 연산자는 이항 연산자이며 “!” 는 단항 연산자입니다.

```
Ex)  (A && B)        // A, B가 모두 1 이상일 경우 참
     (A || B)        // A또는 B가 1 이상일 경우 참
     (! A)           // A 가 0일 경우 참
```

4.2.2 비교 연산자

Verilog 에서 지원하는 연산자는 아래와 같습니다.

크다 (>), 크거나 같다 (>=)
작다 (<), 작거나 같다 (<=)
같다 (= =)
다르다 (! =)

비교 연산의 결과는 참일 경우 “1” 거짓일 경우 “0”으로 1비트 레지스터 또는 Wire에 할당하거나 if-else문 while문 등의 조건문으로 사용할 수 있습니다.

4.2.3 비트 연산자 (Bitwise Operator)

And (&), Or (|), Inv (~), Xor(^) 가 있습니다. 비트 연산자는 단일 비트 뿐만 아니라 Multi 비트에서도 사용할 수 있습니다. 이 경우 연산은 각 비트 별로 하게 됩니다.

```
Ex)  a = 4'b0101;    b=4'b0011;
     c = a & b;          // c = 4'b0001;
     c = a | b;          // c = 4'b0111;
     c = a ^ b;          // c = 4'b0110;
     c = ~ a;            // c = 4'b1010;
```

4.2.4 리덕션 연산자

리덕션 연산은 비트 연산자와 같은 동작을 하지만 피연산자를 하나만 갖습니다. 피연산자의 각 비트를 연산하여 한 비트의 값으로 만듭니다.

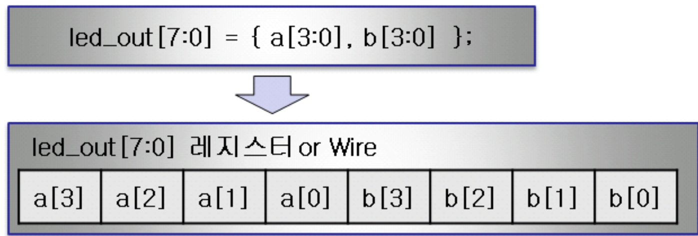
4비트의 레지스터 A (4'b1010) 가 있다면 & A 연산은 A[3] & A[2] & A[1] & A[0] 와 같은 연산을 수행하게 됩니다.

```
all_one_flag = & A;
        ||
all_one_flag = A[3] & A[2] & A[1] & A[0];
```

Ex) even_parity = ^ data[7:0]; // 짝수 패리티 검출
all_one_flag = & data[7:0]; // 8'hff 일때 1 출력
all_zero_flag = ~| data[7:0]; // 8'h00 일때 1 출력 (Zero Flag)

4.2.5 결합 연산자

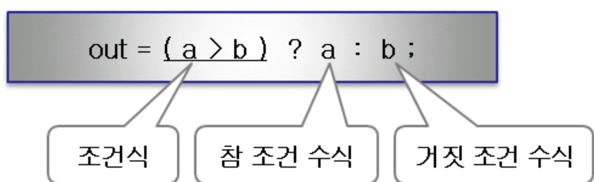
여러 개의 피연산자를 묶어서 사용할 수 있습니다. 결합 연산자를 잘 사용하면 코드 사이즈를 줄일 수 있어 편리합니다. 결합 연산자는 시작과 끝에 중괄호 ({ , })를 사용하고 각 피연산자는 콤마 ","로 구분합니다. 결합 연산자를 이용하여 값을 할당 할 수도 있습니다.



Ex) { START, STOP, RESET } = SW_IN [2:0]; // 스위치 이름 할당
packet [10:0] = { start_bit, data[7:0], even_parity, stop_bit }; // packet 생성

4.3 조건 연산

조건 연산자는 조건 결과에 따라 다른 값을 할당할 때 사용할 수 있습니다. 아래 예제는 a와 b 레지스터(또는 Wire)의 값 중 큰 값을 out에 할당하는 것입니다. 조건식이 참일 경우에는 a를 할당하고 거짓일 경우 b를 할당하게 됩니다.
if-else 문을 간략하게 표현할 때 사용할 수 있습니다. 조건 연산식은 보통 assign 문에서 사용되며 중첩하여 사용할 수 있습니다.



```
Ex)  assign out = sel ? a : b;                        // 2-1 Mux
     assign out =  (sel[1:0] == 2'b00) ? a :
                   (sel[1:0] == 2'b01) ? b :
                   (sel[1:0] == 2'b10) ? c : d;       // 4-1 Mux
```

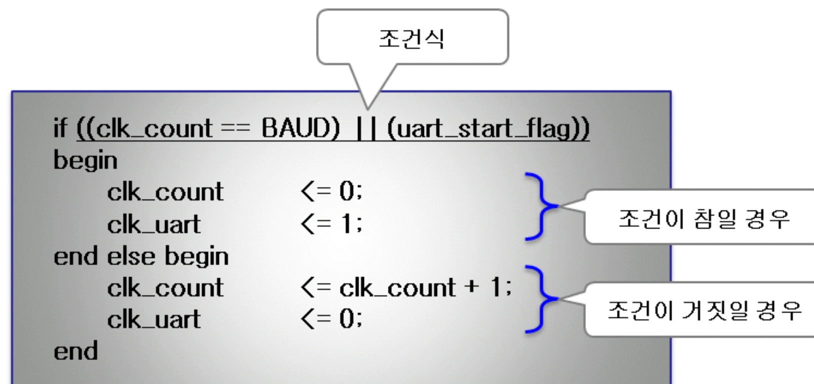
5 제어문

5.1 조건문

5.1.1 If – else 문

조건문은 기술된 조건에 따라 다른 문장을 실행할 때 사용합니다. 기본적인 형대는 If-else 형태로 C언어와 같은 방법으로 사용할 수 있습니다. If문만 단독으로 사용하거나 If - else if와 같은 형태 또는 조건문을 중복하여 사용할 수도 있습니다. 조건문을 중복하여 사용할 경우에는 실행 범위를 begin-end로 정확하게 명시하는 것이 좋습니다. 범위가 명확하지 않은 경우 오동작을 할 수 있습니다.

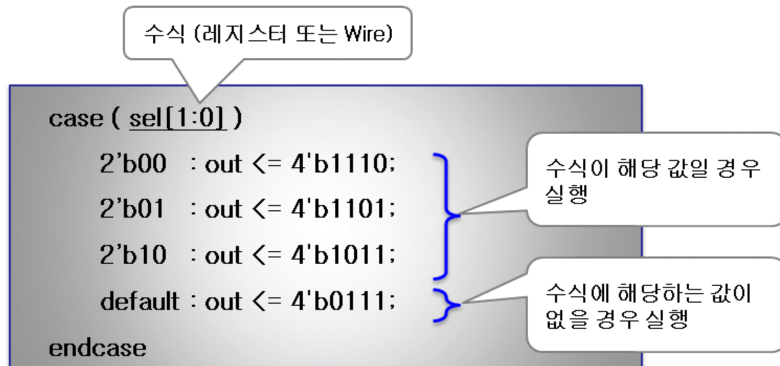
조건식 내에는 다양한 연산자를 사용할 수 있습니다.



※ **조건문을 중복하여 사용할 경우에는 실행 범위를 begin-end로 정확하게 명시해야 합니다.**

5.1.2 Case 문

조건문을 사용할 때 조건이 많은 경우에는 If-else문보다 Case문을 사용하는 것이 효율적입니다. Case문은 C언어의 switch-case 구문과 같은 동작을 하지만 정확한 키워드는 아래 그림을 참고하시기 바랍니다. Case 문의 마지막에는 "endcase"를 사용합니다.



※ Case문과 유사한 조건문으로 "casex", "casez"문이 있으나 실제로 합성은 되지 않으므로 사용하지 않는 것이 좋습니다. "casex", "casez"문은 조건 수식에 "z", "x"와 같은 상태도 사용할 수 있는 구문입니다.

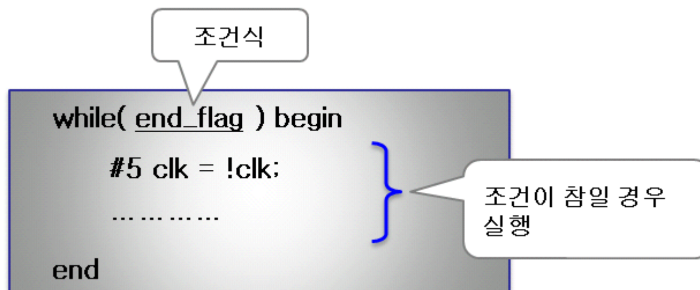
5.2 루프 문 (테스트 벤치에 사용)

루프문은 일반적으로 시뮬레이션을 위한 테스트벤치 작성 시에만 사용합니다. 제한된 형태의 for문 같은 경우 합성이 되기도 하지만 합성툴에 따라 다르기 때문에 합성 구문에는 되도록이면 사용하지 않는 것이 좋습니다.

본 강좌에서는 여러가지 형태의 루프문을 이용하여 테스트벤치 작성시 필요로 하는 10ns 클럭을 만들어 보겠습니다.

5.2.1 While 문

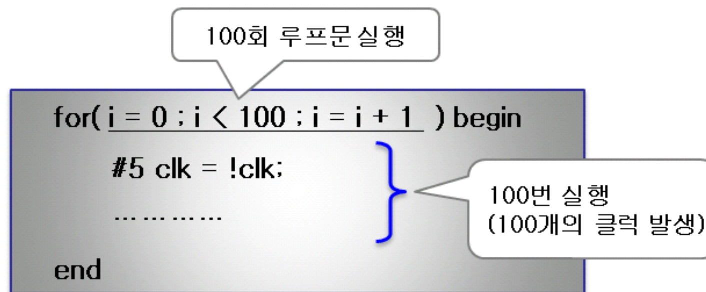
While문은 조건식이 참일 경우 실행합니다. 루프 실행 횟수가 조건에 따라 다르기 때문에 합성이 불가능합니다.



※ "#5" 구문은 timescale에서 정의된 시간의 5배 지연 후에 뒤에 구문을 실행한다는 것을 의미합니다.

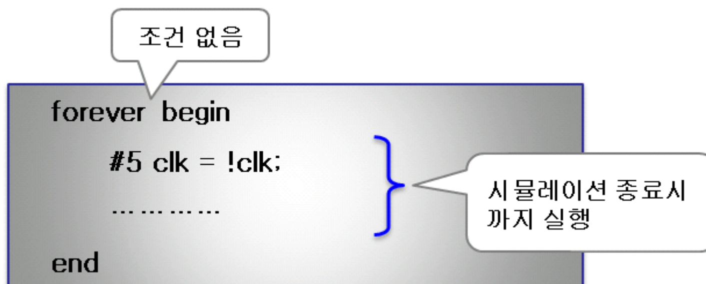
5.2.2 For 문

For 문은 루프의 실행 횟수가 정해진 경우 사용합니다. 실행 횟수가 정해지지 않은 경우 while문을 사용하는 것이 좋습니다. 합성툴에 따라 제한된 횟수의 for문은 합성 가능한 경우가 있습니다.



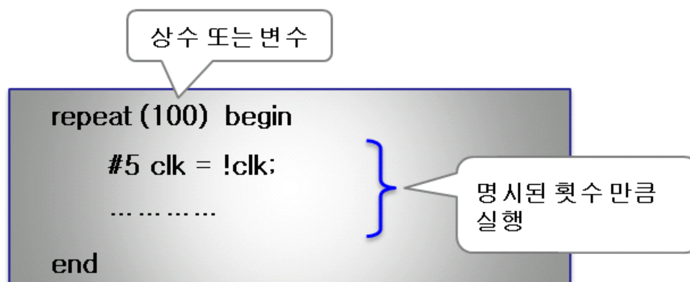
5.2.3 Forever 문

Forever문은 실행문을 무한히 반복할 때 사용합니다. disable문장을 사용하여 종료할 수 있지만 일반적으로 시뮬레이션이 종료될 때까지 동작시킬 때 사용합니다. 시뮬레이션 종료는 $finish 태스크입니다.



5.2.4 Repeat 문

Repeat 문은 특정 횟수만큼 실행할 때 사용합니다. 횟수를 나타내는 문장에는 상수 또는 변수, 파라미터가 올 수 있습니다. 레지스터와 같은 변수가 사용되더라도 Repeat문이 처음 실행할 때 값 만큼 실행됩니다. 루프 실행 중에 레지스터의 값이 변경되더라도 루프 실행 횟수에 영향을 주지 않습니다.

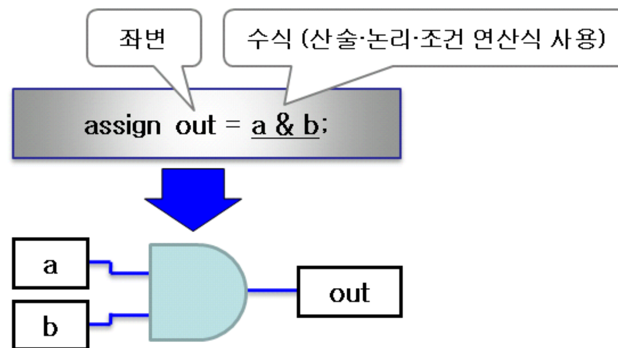


6 조합회로 (Combinational Logic)

6.1 assign 문

조합회로는 입력에 따라 출력 값이 정해져 있는 회로로 Latch 또는 Flip-Flop와 같이 기억 소자가 없는 회로입니다. 실제 구현될 때 AND, OR, NOT 게이트로 구현되는 회로입니다.

assign 문의 좌변에는 wire로 선언된 데이터 형만 올 수 있으며 선언이 생략되어 있는 경우 1비트 wire로 인식하게 됩니다. 우변에는 레지스터, Wire, Parameter 형을 모두 사용할 수 있으며 산술·논리·조건 연산식을 사용할 수 있습니다.



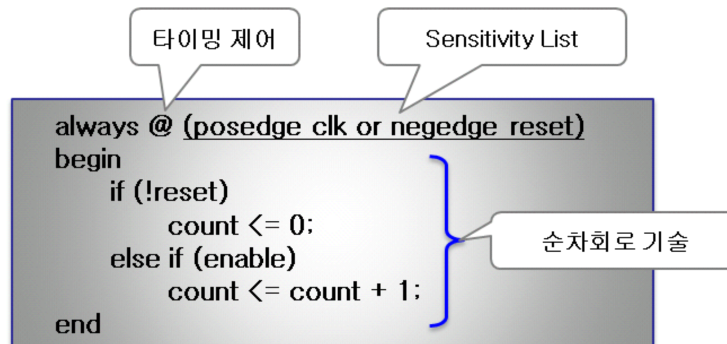

```
Ex)  assign out = a + b;                 // 산술 연산
     assign all_one_flag =  & data[7:0]; // 리덕션 연산
     assign out = sel ? a : b;           // 조건 연산
```

7 순차회로 (Sequential Logic)

7.1 Always 문

순차회로는 입력 뿐만 아니라 현재 상태에 따라 값이 다르게 나올 수 있는 회로입니다. 순차회로는 현재 상태를 기억하고 있기 때문에 메모리 소자(Latch 또는 Flip-Flop)를 가지고 있습니다.

always문의 타이밍 제어가 이벤트일 경우 Sensitivity List에 해당하는 이벤트가 발생할 경우 아래 순차회로가 실행되게 됩니다. (이벤트가 발생하지 않을 경우 값을 유지)

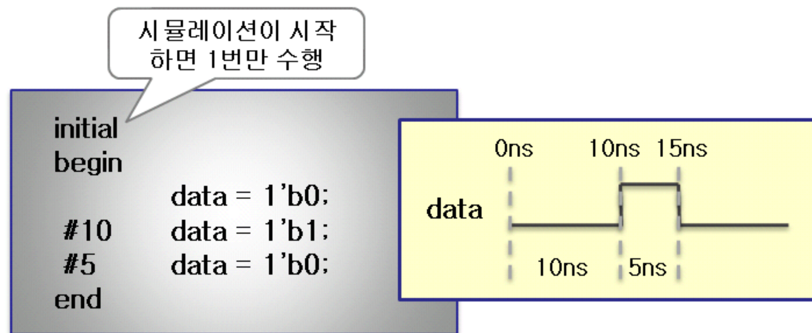


앞의 예제에서 클럭의 상승에지에서 순차회로가 동작하게 되는데 다음 상태의 count 값은 현재 상태의 count 값에 1을 더한 값이 저장됩니다. 클럭이 발생하는 입력은 같지만 현재 상태에 따라 출력 값이 다른 순차회로가 됩니다.

always 문에서 값을 할당할 수 있는 데이터 형은 레지스터 형입니다. 따라서 좌변에는 레지스터 형만 올 수 있으며 생략할 수 없습니다. 우변에는 레지스터, Wire, Parameter 형을 모두 사용할 수 있으며 산술·논리·조건 연산식을 사용할 수 있습니다.

7.2 Initial 문

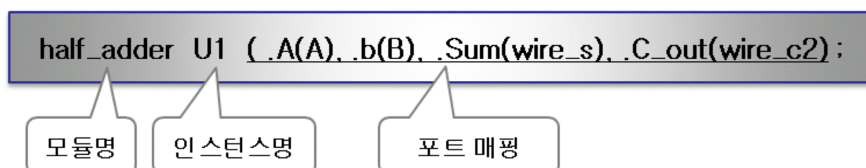
Initial 문은 시뮬레이션을 위한 구문으로 순차적으로 신호를 인가할 때 사용합니다. 시뮬레이션이 시작하면 모든 Initial 구문이 실행되어 파형을 만들어 냅니다. 시간 지연을 위해서 일반적으로 블로킹 구문을 사용하며 타이밍 제어를 합니다.



시뮬레이션이 시작하면 위 예제의 initial 문을 실행하게 됩니다. 기준시간이 1ns라고 가정 하면 시뮬레이션 시작과 동시에 data에는 1'b0이 할당됩니다. 10ns 시간 지연 후에 1'b1이 할당되며 다시 5ns 시간 지연 후에 1'b0이 할당됩니다. 마지막의 1'b0 은 15ns 시간에 할당되게 됩니다.

8 계층구조 설계 (하위 모듈 호출)

특별한 기능을 하는 블록 또는 반복해서 사용되는 블록은 모듈화 하여 사용할 수 있습니다. 블록을 모듈화 한 후 상위 계층에서 하위 모듈을 생성(Instantiation)하게 됩니다. 하위 모듈을 생성하는 방법은 아래와 같이 모듈명과 인스턴스명 그리고 포트 매핑을 하면 됩니다.



각 포트 연결은 "•포트명(Wire명)" 형식으로 합니다. "•"으로 시작하고 포트명과 Wire명이 나오게 됩니다. 괄호 안에 값이 상위 블록에서 사용하는 Wire 이름 입니다.
포트 매핑을 하는 다른 방법으로는 위치를 이용한 방법이 있습니다. 해당 위치에 Wire 이름을 쓰면 되는데 코드의 가독성이 떨어지기 때문에 일반적으로 이름을 이용한 매핑을 많이 사용합니다.