Verilog HDL Summury

by 강석태

2006년 3월

Module

```
module <모듈 이름>(<포트 리스트>)
 <모듈 내용>
endmodule
```

- C 언어의 함수(Function)와 같은 개념.
- 대소문자 구분.
- 예약어는 소문자로만 쓴다.
- 이름은 영문자, 숫자, 언더 바(_)만 허용한다.
- 문장의 끝은 항상 세미콜론(;)으로 끝난다.
- end~ 로 시작하는 예약어에는 ;이 붙지 않는다.
- 하나의 File은 하나의 module만 포함시킨다.
- 파일 이름은 module의 이름과 동일하게 한다.

Module

- 모듈 선언 예제

```
module Adder (X, Y, C, S);
  input X, Y;          //입력 포트 선언
  output C, S;         //출력 포트 선언
  wire C, S;           //레지스터, 와이어 선언
  assign C = X & Y; //모듈 내용
  assign S = X ^ Y;
endmodule
```

기초 문법

- begin ~ end
  - initial, if, case, always 등을 사용할 때 블록을 지정할때 사용한다.
  - C 언어에서 { ... }와 같은 개념이다.
- 주석
  - // : 한 줄을 comment 처리할 때 사용.
  - /* ~ */ : 여러 줄을 comment 처리할 때 사용.
- 수 표현
  - <비트 폭>'<진수><값> : 8'hFB
  - <진수> : b – 2진수, d – 10진수, h – 16진수
  - 음수는 2'complememt 사용
  - 언더바(_) : 가독성을 좋게 함. (2'b1100_0101)

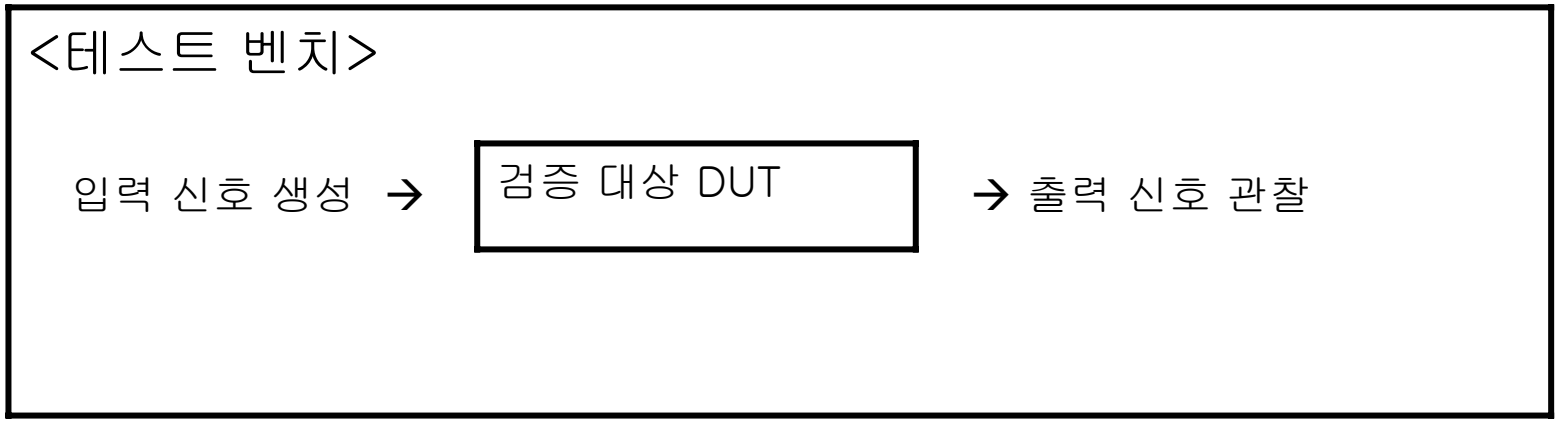
기초 문법

- 벡터(Vector)
  - 멀티비트(Multi-bits) 또는 버스(bus)라고도 한다.
  - 예약어 [MSB:LSB] 이름
  - 예약어 : input, output, reg, wire

```
input [3:0] A, B;
input [7:0] ABUS;
wire [2:0] status;
assign C = ABUS[7];
assign D = ABUS[0];
wire [3:0] H_digit;
assign H_digit = ABUS[7:4];
```

테스트 벤치 (Testbench)

- 정의
  - 시뮬레이션을 위한 모듈이다.
  - 설계한 DUT를 검증하기 위한 목적으로 사용한다.
  - DUT를 내포하고 있으며 DUT의 입력 값을 생성하고, 출력 값을 관찰한다.



테스트 벤치

- 시스템 기능 연산자
  - 테스트 벤치에서만 사용한다.
  - $ 기호로 시작한다.
  - $stop 시뮬레이션 정지
  - $finish 시뮬레이션 끝
  - $time 현재 시뮬레이션 시간을 얻는다.
  - $monitor, $display 특정 값을 디스플레이 할 때 사용한다.

```
Initial begin
   #300; $finish;
end
```

테스트 벤치

- `timescale 문
  - `timescale <테스트벤치 스텝 단위>/<시뮬레이션 스텝 단위>

ex) `timescale 1ns/1ps

  - 테스트 벤치 상단에서 시간단위를 지정해준다.
  - 문장의 끝에 세미콜론이 붙지 않는다.
  - 테스트벤치 스텝 단위는 시간 지연등을 사용할때의 단위이다.
  - 예를 들어 `timescale 1ns/1ps 로 선언을 하였을 경우 테스트 벤치에서 #30의 의미는 30ns를 의미한다.
  - 시뮬레이션 스텝 단위는 시뮬레이션 후 timing diagram에서 보여지는 시간의 resolution을 결정한다.

테스트 벤치

- initial 문
  - 순차적으로 한번만 실행 시킬 때 사용된다.
  - 테스트 벤치에서만 사용이 가능하다.
- 시간 지연
  - initial 문과 함께 사용하여 DUT의 입력 값을 생성 할 수 있다.

```
initial begin
// 30[ns] 동안 X의 값을 1로 유지시킨다.
  X = 1;  #30;
end
```

데이터 형 (Data Type)

- wire
  - 논리적인 기능이나 행동 없이 단순한 선을 의미.
  - 물리적인 wire를 의미.
  - 게이트나 모듈간의 입력과 출력을 연결할 때 사용.
  - wire로 선언된 signal은 assign 문에서만 사용 가능하다.

- reg
  - 순차회로에서 사용할 때에는 플립플롭이 된다.
  - reg로 선언된 signal은 always 문에서만 사용 가능하다
  - 테스트 벤치에서는 initial 문에서 사용한다.
  - 새로운 이벤트가 발생하기 전까지 기존의 값을 유지한다.

데이터 형 (Data Type)

- parameter
  - 상수를 정의할 때 사용한다.
  - parameter WIDTH = 32;
  - parameter LENGTH = 8'h16;
  - parameter MASK = 4'b1010;

연산자

산술 연산자	+	덧셈
	–	뺄셈
	*	곱셈
관계 연산자	==	같다
	!=	같지 않다
	<	작다
	<=	작거나 같다
	>	크다
	>=	크거나 같다
논리 연산자	&&	논리 AND
	||	논리 OR
	!	논리 NOT

비트 연산자	!	비트 NOT
	&	비트 AND
	|	비트 OR
	^	비트 XOR
시프트 연산자	>>	오른쪽 shift
	<<	왼쪽 shift
그 외 연산자	? :	조건 연산자
	{ }	결합 연산자

연산자

- 조건 연산자의 예제
  - assign Y = (A>B) ? A : B;
  - A가 B보다 크면 Y에 A값을 할당, 그렇지 않으면 B를 할당.

- 결합 연산자의 예제
  - assign Y[7:0] = {A[3:0], B[3:0]};
  - 4비트짜리 A,B를 결합하여 8비트짜리 Y를 만든다.
  - Y의 MSB 자리에 A, LSB 자리에 B가 들어간다.
  - A가 4'hA 이고, B가 4'hE이라면
    Y의 값은 8'hAE 가 된다.

조건문 (if 문)

- if 문
  - 항상 always 문 안에서만 사용이 가능하다.
  - 문장이 2줄 이상이 되는 경우는 begin ~ end로 묶어준다.
  - C 언어에서의 if ~ else 문과 같다.

if(<조건식>)
  문장 1
else
  문장 2
	조건식이 참이면 문장1을 실행하고,
	거짓이면 문장2를 실행한다.
	else if를 사용하여 조건을 추가할 수도 있다.

조건문 (case 문)

- case 문
  - 항상 always 문 안에서만 사용이 가능하다.
  - 각 항의 문장이 2줄 이상인 경우는 begin ~ end로 묶어준다.
  - C 언어에서의 switch ~ case 문과 같다.

case(판정식) 항1 : 문장1; 항2 : 문장2; ... default : 문장N; endcase	판정식이 항1과 같으면 문장 1을 수행, 항2와 같으면 문장 2를 수행, 모든 항과 같지 않으면 default의 문장N을 수행한다.

조합 회로

- assign 문
  - 조합회로에서 값을 할당할 때 사용한다.
  - assign 문으로 값을 받는 signal은 wire로 선언되어야 한다.
  - assign 문은 “=“ 기호를 사용한다.

ex) wire [7:0] Y;
    assign Y = A + B;

조합 회로

- 조합회로의 always 문
  - 조합회로에서 특정 조건문을 사용하고자 할 때 always 문을 쓴다.
  - always 문으로 값을 받는 signal은 reg로 선언되어야 한다.
  - 하나의 signal을 2개 이상의 always 문에서 값을 바꾸어서는 안 된다.
  - if문, case문 사용 할 때 else 또는 default 문장을 절대로 빠뜨리지 않아야 한다.

조합 회로

- 조합 회로 always 문 예제 (1)
  - always @ (<이벤트 리스트>) begin
    ..
    end
  - 이벤트 리스트에는 always 문이 동작하기 위해 필요한 signal이 모두 포함되어야 한다.
  - 이벤트 리스트를 쉽게 구분하는 방법은 “=“을 중심으로 우측항에 있는 signal과 if문 또는 case문의 조건에 사용되는 signal을 포함시키면 된다.

조합 회로

- 조합 회로 always 문 예제 (2)

```
always @ (A or B) begin
 if(A>B)
   Y = A;
 else
   Y = B;
end
```

```
always @ (alpha or beta or A or B)
begin
 if(alpha < beta)
   Y = A;
 else
   Y = B;
end
```

조합 회로

- 조합 회로 always 문 예제 (3)

```
wire  [3:0] DEC;
reg  [3:0]  Y;
always @ (DEC) begin
  case(DEC)
    4'h1 : Y = 4'b0001;
    4'h2 : Y = 4'b0010;
    4'h4 : Y = 4'b0100;
    4'h8 : Y = 4'b1000;
    default : Y = 4'b0000;
  endcase
end
```

조합 회로

- 조합회로 설계 시 주의할 점
  - Feedback 무한 루프 발생 금지.
  - assign 문, always 문 동일
  - A = A + 1; //에러
  - A[1] = A[0] & x;
    A[2] = A[1] & A[0];
    - 이 문장은 Tool에 따라서 허용이 되는 경우도 있다.
      그러나 가급적 이와 같은 문장은 사용하지 않도록 한다.
- 해결
  - 순차 회로로 변경한다. (A <= A + 1)
  - 조합 회로의 경우는 signal 이름을 분리하여 사용한다.
    Ex> A1 = A0 & x;
        A2 = A1 & A0;

순차 회로

- 순차회로의 always 문 (1)
  - 순차 회로의 구현은 **반드시 always 문**을 사용.
  - always 문에서 값을 할당 받는 signal은 reg 로 선언한다.
  - always 문에서 이벤트 리스트에는 **clock과 reset만 사용**한다.
  - 순차회로에서는 **논-블락킹 대입문(“<=”)을 사용**한다.
  - "<="의 특징은 begin ~ end 블록의 여러 문장들이 동시에 수행된다는 것이다.

순차 회로

- 순차회로의 always 문 (2)
  - reset의 동작은 가능한 negative edge(negedge)를 사용한다.
  - always 문에서 사용하는 **reg는 플립플롭이기 때문에 반드시 reset** 이 필요하다.
  - always 문 내의 시작은 항상 reset 동작을 수행하도록 한다.
  - 하나의 module에서 여러 개의 always문이 사용 가능하다.
    이 때 주의할 점은 같은 레지스터를 서로 다른 always 문에서 값을 할당하면 안 된다는 것이다.

순차 회로

- 순차회로의 always 문 예제

```
always @ (posedge clock or negedge reset)
begin
  if(!reset)    //reset 시작, low active
   Y <= 0;      // "<=" 사용
  else begin
    if(A>B)
      Y <= A;
    else
      Y <= B;
  end
end
```

순차 회로

- always 문 사용 시 주의할 점.(1)
  - 클럭 검출의 에지에 벡터의 인덱스는 사용할 수 없다.
    always @ (posedge CLK[1]) //에러
  - 리셋 조건에는 벡터의 비트는 사용할 수 없다.
    always @ (posedge CLK or negedge RESET_BUS) //에러
    if(!RESET_BUS[1]) //에러
  - 리셋 조건에 복잡한 수식을 사용할 수 없다.
    always @ (posedge CLK or negedge RESET)
    if(RESET == (1-1)) //에러

순차 회로

- always 문 사용 시 주의할 점.(2)

  - 리셋 조건에 RESET 이외의 신호를 추가로 사용할 수 없다.

```
always @ (posedge CLK or negedge RESET)
  if(!RESET || zero_init)                    //에러
```

  - 하나의 if문이 always 문 블록의 처음에 있어야 한다.

```
always @ (posedge CLK or negedge RESET)
  A <= 3;                                    //에러
  if(!RESET)
```

조합 회로 & 순차 회로

- 문법 구분.
  - 조합 회로 : assign 문, always 문,
    “=“ 사용
  - 순차 회로 : always 문,
    “<=“ 사용

- 데이터 타입
  - assign 문 : wire 선언
  - always 문 : reg 선언

Module의 Port Mapping

- 모듈의 호출 (1)
  - C 언어에서 main() 함수에서 다른 함수를 호출하는 개념과 같다.
  - 테스트 벤치에 검증하기 위한 DUT를 포함시킬 때 사용된다.
  - 테스트 벤치 또는 특정 모듈에서 호출되는 모듈을 Sub module이라 한다.
  - Sub module을 호출 할 때에는 항상 instance 명을 사용한다.

Ex>

```
module tb_test();
   …
   test  test_inst(<포트 리스트>);  //DUT 호출
endmodule
```

Module의 Port Mapping

- 모듈의 호출 (2)
  - instance 명이 모듈의 이름과 동일 하여도 된다.
    (모듈을 한번만 호출 할 때 가능)
  - 하나의 모듈을 설계한 후 이 모듈을 여러 번 불러서 사용해야 할 경우에 유용하다.
  - 동일한 모듈을 여러 번 호출할 때마다 instance 명을 달리 해야 한다.

Ex>

```
module tb_test();
  …
  test test_inst0(<포트 리스트>);
  …
  test test_inst1(<포트 리스트>);
endmodule
```

Module의 Port Mapping

- 모듈의 입출력
  - 모듈의 포트리스트는 입력과 출력의 순서는 상관없다.
    (일반적으로 입력을 먼저 쓰고, 출력을 나중에 쓰도록 한다.)
  - 입출력 데이터가 멀티비트인 경우는 다음과 같이 쓴다.
    input [3:0] A;
    output [7:0] B;
  - input 신호를 모듈 내에서 값을 변경 할 수 없다.
  - output으로 정의된 신호들은 모두 reg 또는 wire로 재정의 해주어야 한다.
    - output 신호는 현재 모듈에서 값을 생성해 주어야 하기 때문이다.

Module의 Port Mapping

- 모듈의 입출력 예제

```
module test(A, B, X, Y);
 input [2:0] A;
 input B;
 output [2:0] X, Y;
 wire [2:0]  X;
 reg [2:0] Y;

assign  X = A + 3'd1;

always @ (A or B) begin
  if(B == 1)
    Y = A + 3'd2;
  else
    Y = A + 3'd3;
 end
endmodule
```

Module의 Port Mapping

- 모듈의 내부신호
  - 모듈 내에서 값을 저장할 필요가 있거나 모듈과 모듈 간에 값을 연결할 필요가 있을 때 생성한다.
  - always 문을 사용하여 값을 생성할 필요가 있을 때에는 reg 선언을 하여 사용한다.
  - assign 문을 사용하여 값을 생성하거나, 서로 다른 두 개의 sub module간에 값을 연결할 때에는 wire 선언을 하여 사용한다.

Module의 Port Mapping

- 모듈의 내부신호 예제 (1)

```
module test(reset, clock,
                A, B, Z);
 input reset, clock;
 input [2:0] A, B;
 output [3:0] Z;
 wire [3:0] Z;
 wire [2:0] Y;
 reg [3:0] add_value;

 sub_mod  sub_mod0(A, B, Y);
 sub_mod  sub_mod1(Y,add_value,Z);
```

```
always @ (posedge clock or
negedge reset) begin
  if(!reset)
   add_value <= 0;
  else
   add_value <= A +B;
 end


endmodule
```

Module의 Port Mapping

- 모듈의 내부신호 예제 (2)
  - sub_mod라는 모듈은 2개의 입력을 받아서 2 신호의 차를 출력한다. 여기서는 sub_mod라는 모듈을 2번 호출하였다.
  - add_value 라는 레지스터는 test 모듈의 입력 A, B의 합을 저장한다.
  - Y 라는 wire는 sub_mod0의 출력을 받아서 sub_mod1의 입력으로 연결하는데 사용하였다.
  - Z는 output신호이며, sub_mod1의 출력을 받아서 output으로 연결한 것이다.

Module의 Port Mapping

- 모듈의 포트 Mapping 방법
  - 직접 Mapping
    - 모듈과 모듈간에 포트의 위치를 정확하게 1대1로 연결하는 방법.
    - 각 포트들의 순서가 매우 중요하다.
  - 참조 Mapping
    - 모듈과 모듈간에 포트의 이름을 찾아서 연결하는 방법.
    - 포트의 순서는 중요하지 않으며 이름만 정확하게 참조하면 된다.
    - .<서브 모듈의 포트 이름>(<현재 모듈의 포트 이름>),
    - 프로그래머의 실수에 의한 신호의 오동작을 줄여준다.

Module의 Port Mapping

- 직접 Mapping 예제

```
module Mother(I,J,K);
 input [2:0]  I,J;
 output [2:0] K;
 wire [2:0] K;
 wire [2:0] X, Y;
 sub_mod  sub_mod0(I, J, X);
 sub_mod  sub_mod1(J, I, Y);
 assign K = X + Y;
endmodule
```

sub_mod 라는 모듈을 2번 호출.
sub_mod0의 입력은 I,J.
출력은 X.
sub_mod1에 입력은 J,I
출력은 Y.

sub_mod의 기능이 2개의 입력신호를
순서대로 뺄셈을 한다고 했을 때
수식은 다음과 같다.
sub_mod0 : X = I – J;
sub_mod1 : Y = J – I;

입·출력의 순서가 매우 중요하다.

Module의 Port Mapping

- 참조 Mapping 예제

```
module Mother(I,J,K);
 input [2:0]  I,J;
 output [2:0] K;
 wire [2:0] K;
 wire [2:0] X, Y;
 sub_mod  sub_mod0(.A(I),
                   .B(J),
                   .O(X)
                   );
 sub_mod  sub_mod1(.A(J),
                   .B(I),
                   .O(Y)
                   );
 assign K = X + Y;
endmodule
```

sub_mod의 포트는 A, B, O
sub_mod0에서는 A와 I, B와 J,
O와 X가 서로 연결 되었다.

```
sub_mod0(.B(J),
         .A(I),
         .O(X)
         );
```

좌측의 sub_mod0와 A, B의 순서가
바뀌었지만, 포트의 연결은
A-I, B-J로 연결됨으로 좌측과 같다.

Module의 Port Mapping

- 서브 모듈의 연결 예제

```
module top(I,J,c_in, carry, sum);
 input [2:0]   i, j;
 input          c_in;
 output [2:0] sum;
 output        carry;
 wire [2:0]    sum;
 wire          carry;
 wire          w1, w2, w3;
 wire          s0, s1;
 FA FA0(.x(i[0]),
        .y(j[0]),
        .c_in(c_in),
        .c_out(w1),
        .sum(s0)
         );
 FA FA1(.x(i[1]),
        .y(j[1]),
        .c_in(w1),
        .c_out(w2),
        .sum(s1)
         );
 FA FA2(.x(i[2]),
        .y(j[2]),
        .c_in(w2),
        .c_out(w3),
        .sum(s2)
         );
 assign carry = w3;
 assign sum[2:0] = {s2, s1, s0};
endmodule
```

– The End –