논리회로의 주요용어 개념정리(HDL, IP, FPGA) & Verilog 기본 공부하기 Verilog

2018. 8. 16. 20:03
https://blog.naver.com/hanbulkr/221340154383

https://m.blog.naver.com/PostView.nhn?blogId=interlaken89&logNo=202066596 10&proxyReferer=https%3A%2F%2Fwww.google.co.kr%2F

02. 논리회로의 주요용어 개념정리(HDL, IP, …
1. 문제 설명 A-1. Logic circuit에서 SSI, MSI, L…
m.blog.naver.com

이 곳의 자료를 정리.

IC

Integration Circuit : Integration Circuit은 반도체 물질로 (보통 실리콘) 이루어진 칩에서 전자회로들의 집합이라고 정의할 수 있으며 IC, chip, microchip이라고도 불린다. (이하 IC) 우리말로 집적회로라고 한다. 그리고 칩에 들어가는 트랜지스터의 개수에 따라 SSI, MSI, LSI, VLSI로 구분되고 있다.

SSI, MSI, LSI, VLSI

- **SSI :** small-scale intergration의 약자로, **소규모 집적회로**라 한다. 한 개의 칩에 수십개의 트랜지스터가 집적된 회로를 의미한다.

- **MSI :** medium-scale intergration의 약자로, **중규모 집적회로**라 한다. 한 개의 칩에 수백개의 트랜지스터가 집적된 회로를 의미한다. 병렬 가산회로나 쉬프트 레지스터에 쓰인다.

- **LSI :** large-scale intergration의 약자로, **대규모 집적회로**라 한다. 한 개의 칩에 수만개의 레지스터가 집적된 회로를 의미한다. 초기 컴퓨터의 메인 메모리나 CPU에 쓰인다.

- **VLSI :** very-large-scale intergration의 약자로, **초고밀도 집적회로**라 한다. 1980년대 초에 한 개의 칩에 수십만개의 트랜지스터가 집적된 회로만을 일컫었으나, 현재 수십억개의 트랜지스터가 집적된 회로까지 의미한다.

HDL

HDL : Hardware Description Language의 약자. **하드웨어 언어 기술**이라고 한다. 이것은 전자공학에서 전자회로를 정밀하게 기술하는 데 사용하는 컴퓨터 언어이다. 흔히 HDL이라고 줄여쓴다. 이 언어를 통해 사용자는 회로의 원하는 동작을 기술할 수 있고 시뮬레이션을 통해 제대로 동작하는지 검증할 수도 있다.

대표적인 종류로 VHDL과 Verilog등이 있다.

VHDL (VHSIC Hardware Description Language)은 디지털 회로의 설계 자동화에 사용하는 하드웨어 기술 언어이다. 원래 주문형 집적회로(ASIC)의 문서화에 사용하기 위해 만든 언어였다. 즉, 복잡한 매뉴얼로 회로의 동작 내용을 설명하는 대신, 회로의 동작 내용을 문서화하여 설명하기 위해 개발했다. 그러나 이런 문서를 회로 디자인 과정에서 시뮬레이션에 사용하게 되었고, VHDL 파일을 읽어들여서 논리 합성을 한 다음 실제 회로 형태를 출력하는 기능을 덧붙이게 되었다. 오늘날에는 디지털 회로의 설계, 검증, 구현 등의 모든 용도로 사용하고 있다.

Verilog는 전자 회로 및 시스템에 쓰이는 하드웨어 기술 언어 (HDL)이다. '**Verilog HDL**'이라고 부르기도 한다. 회로 설계, 검증, 구현등 여러 용도로 사용할 수 있다. **C 언어와 비슷한 문법**을 가져서 사용자들이 쉽게 접근할 수 있도록 만들어졌다. **'if'나 'while'같은 제어 구조도 동일**하며, 출력 루틴 및 연산자들도 거의 비슷하다. 다만 **C 언어와 달리, 블록의 시작과 끝을 중괄호 기호로 대신 Begin과 End**를 사용하여 구분하고, HDL의 특징인 시간에 대한 개념이 포함되었다는 것등 일반적인 프로그램과 다른 점도 많이 있다.

IP (Intellectual Property) : 지적재산. FPGA나 주문형 반도체(ASIC) 제품을 만드는 데 사용되는 일단의 로직이나 데이터. 설계 재사용의 주요 요소로서 이전 설계를 반복 사용하는 전자 설계 자동화(EDA) 산업의 일부분이다. 제조 시나 설계 시에 쉽게 삽입될 수 있도록 이식성이 있어야 하며, UART, CPU, 이더넷 제어 장치, PCI 인터페이스가 모두 IP core이다. 형태에는 hard core, firm core, soft core 3가지가 있다. -> **결론 IP는 VHDL, Verilog 등의 프로그래밍이 삽입된 하나의 소자이다.**

FPGA (Field Programmable Gate Array) : 이미 설계된 하드웨어를 반도체로 생산하기 직전 **최종적으로 하드웨어의 동작 및 성능을 검증**하기 위해 **제작하는 중간개발물 형태**의 **집적 회로(IC)**. 비메모리 반도체의 일종으로, 회로 변경이 불가능한 일반 반도체와 달리 여러 번 회로를 다시 새겨 넣을 수 있는 반도체다. **오류 발생시 수정이 가능하고 개발시간이 짧으며 초기 개발비용이 적게 든다.** 그러나 **일반적으로 속도가 느리고 복잡한 설계에 적용이 불가**하며 **소비전력이 크다**는 단점이 있다.

〈Verilog HDL 공부하기〉

1. Verilog 기본 형태

하드웨어 모델링 기법 (①Structual Modeling, ②Behavioral Modeling)

①Structual Modeling 〈 구조적으로 하드웨어를 모델링 〉

- 하드웨어를 구성하는 **부품, 소자 또는 모듈**이라고 불리는 것들을 서로 연결하여 하드웨어를 표현

(※부품, 소자, 모듈 들은 하위레벨의 하드웨어를 말한다)

②Behavioral Modeling 〈하드에워가 어떻게 행동할지를 소프트웨어로 모델링〉

- 소프트웨어로 하드웨어의 동작을 표현하면 합성툴(synthesis Tool)이 동일하게 동작하는 하드웨어를 만들어 준다.

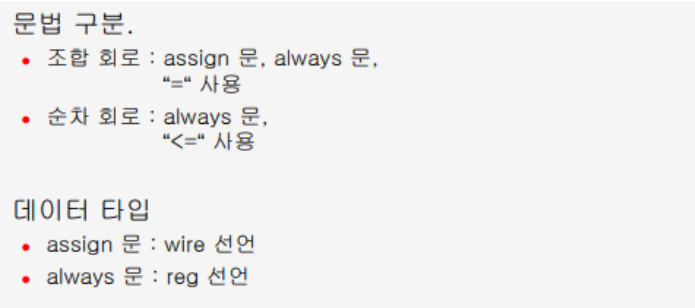
(※합성툴 - 컴파일러 같은 HDL을 실제 하드웨어로 구성할 수 있는 소자로 바꿔주는 툴이다.)

〈기본 문법〉

1. VHDL과 달리 **대소문자의 구분이 가능**하다.
2. 모든 **예약어들은 소문자로 작성**해야 한다.
3. 문장의 끝에는 ' ; ' (세미콜론) 으로 끝내고 주석 처리는 '/* */ ' , '//'
4. 이름은 영문자, 숫자, 언더 바(_)만 허용한다.
5. **파일 이름은 module의 이름과 동일**하게 한다.
6. end~ 로 시작하는 **예약어에는 ;이 붙지 않는다.**
7. **하나의 File**은 **하나의 module**만 포함시킨다.
8. **begin ~ end** 는 c언어에서 "{" "}" 중괄호 열고 닫는 녀석이랑 같다.
9. **수 표현:** 〈비트 폭〉'〈진수〉〈값〉 _ex) 8'hFB
10. **〈진수〉 :** b(2진수) , d(10진수), h(16진수)

Verilog 기본 형태

- **포트 리스트**에는 입출력에 대한 포트 리스트를 쓴다.
- **포트 선언부와 데이터 타입 선언부**는 모듈 내에서 연결될 와이어가 필요한 경우 선언 하는 곳이다.
- **회로 구현부**는 포트로 선언된 와이어와 데이터타입으로 선언된 와이어가 서로 어떻게 연결되어 있는지 또는 어떤 하위레벨의 모듈이 연결되어 있는지를 표현하여 회로를 구현하는 부분이다.



---

2. 데이터 형 (Data Type)

wire

- 논리적인 기능이나 행동 없이 **단순한 선**을 의미.
- 물리적인 wire를 의미.
- **게이트나 모듈간**의 **입력**과 **출력**을 연결할 때 사용.
- **wire로 선언된 signal**은 **assign 문에서만 사용 가능**하다.

reg

- **순차회로**에서 사용할 때에는 **플립플롭**이 된다.
- reg로 선언된 signal은 **always 문에서만 사용 가능**하다
- **테스트 벤치**에서는 **initial 문에서 사용**한다.
- 새로운 이벤트가 발생하기 전까지 기존의 값을 유지한다.

parameter

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

---

3. 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] multibit;

wire [2:0] status;
wire [3:0] merge;

reg register1;                  // register1를 만단다.
reg [4:0] register2;            // register2를 5bit의 크기로 만든다.

assign C = multibit[7];         // 7번 bit를 연결한다.
assign D = multibit[0];         // 0번 bit를 연결한다.

assign merge = multibit[7:4];   //merge에 7번bit에서 4bit사이의 값을 넣겠다는 뜻
```

---

4. assign 문

```
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
```

- Verilog 에서 **assign 문은 와이어를 연결**하는데 사용한다.
- assign 문으로 값을 받는 signal은 **wire로 선언**되어야 한다.
- assign 문은 "=" 기호를 사용한다.

5. 조건문(if , case 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.

case 문

- 항상 **always 문 안에서만 사용**이 가능하다.
- 각 항의 **문장이 2줄 이상인 경우**는 **begin ~ end**로 묶어준다.
- C 언어에서의 **switch ~ case 문**과 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.

---

6. always 문

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 빠뜨리 지 않아야** 한다.

```
always @ (<이벤트 리스트>) begin
..
end
```

- 이벤트 리스트에는 always 문이 동작하기 위해 필요한 signal이
모두 포함되어야 한다.

- **이벤트 리스트를 쉽게 구분**하는 방법은 **"="을 중심**으로 ①**우측항에 있는 signal**과 ②**if문 또는 case문의 조건에 사용되는 signal**을 포함시키면 된다.

```
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
```

```
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
```

- **순차 회로의 구현**은 반드시 **always 문**을 사용.
- always 문에서 값을 할당 받는 signal은 **reg 로 선언**한다.
- always 문에서 **이벤트 리스트에는 clock과 reset만 사용**한다.
- **순차회로에서는 논·블락킹 대입문("<=")**을 사용한다.
- **"<="의 특징은 begin ~ end 블록의 여러 문장들이 동시에 수행**
된다는 것이다.

- **reset 동작은 가능한 negative edge**(negedge)를 사용한다.
- **always 문에서 사용하는 reg는 플립플롭**이기 때문에 **반드시 reset**
이 필요하다.
- always 문 내의 **시작은 항상 reset 동작을 수행**하도록 한다.
- **하나의 module**에서 **여러 개의 always문**이 사용 가능하다.
이 때 주의할 점은 같은 레지스터를 서로 다른 always 문에서 값
을 할당하면 안 된다는 것이다.

```
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
```

always 문 사용 시 주의할 점

- **클럭 검출의 에지**에 **벡터의 인덱스는 사용할 수 없다.**

```
always @ (posedge CLK[1])  //에러
```

- **리셋 조건**에는 **벡터의 비트는 사용할 수 없다.**

```
always @ (posedge CLK or negedge RESET_BUS)  //에러
  if(!RESET_BUS[1])   //에러
```

- **리셋 조건**에 **복잡한 수식을 사용할 수 없다.**

```
always @ (posedge CLK or negedge RESET)
  if(RESET == (1-1))   //에러
```

- **리셋 조건**에 **RESET 이외의 신호**를 **추가로 사용할 수 없다**.

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

- 하나의 **if문이 always 문 블록의 처음**에 있어야 한다.

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

7. 연산자

연산자 (operator)

연산자는 산술 연산자, 관계 연산자, 논리 연산자, 시프트 연산자 등이 있으며 값을 연산하는 것에 사용한다.

구분	연산자	의미
산술 연산자	+	덧셈
	-	뺄셈
	%	나머지
	*	곱셈
	/	나눗셈
관계 연산자	==	같다
	!=	같지 않다
	>	크다
	>=	크거나 같다
	<	작다
	<=	작거나 같다
논리 연산자	&&	논리적 AND
	||	논리적 OR
	!	논리적 NOT
	&	비트 AND
	|	비트 OR
	~	비트 NOT
	^	비트 XOR
	^~, ~^	비트 XNOR
시프트 연산자	>>	오른쪽 Shift
	<<	왼쪽 Shift
기타	{}	결합 연산
	?:	3항 조건 연산

http://www.rebas.kr/214 에서 퍼옴

ex)

산술연산의 경우

```
reg x, y;
wire result;
x =2'b0101;// 5
y =2'b0001;// 1
assign result = x + y;// result = 2'b0110
assign result = x - y;// result = 2'b1000
assign result = x * y;// result = 2'b0101
assign result = x / y;// result = 2'b0101
assign result = x % y;// result = 2'b0000
```

```
// 논리 연산
x =2'b00;
y =2'b01;
z =2'b0x;
(x ==1) || (y ==1);          // (거짓) || (참) 이므로 결과 값은 1
(x ==1) && (y ==1);          // (거짓) && (참) 이므로 결과 값은 0
(x ==1) || z;                // 결과 값은 x

// 비트 연산
x =4'b1101;
y =4'b1001;
z =4'b1xxz;
~x;          // 결과 값은 4'b0010
x & y;       // 결과 값은 4'b1001
x ^ y;       // 결과 값은 4'b0100
x ^~ y;      // 결과 값은 4'b1011
y | z;       // 결과 값은 4'b1xx1
~z;          // 결과 값은 4'b0xxx

// 축소 연산
x =4'b11011;
&x          // (1 & 1 & 0 & 1 & 1)  결과 값은 1'b0
~&          // (1 ~& 1 ~& 0 ~& 1 ~& 1) 결과 값은 1'b0
```

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.)

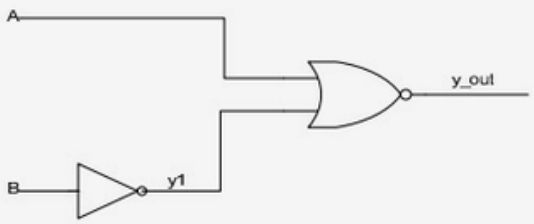
(A가 4'hA 이고, B가 4'hE이라면 Y의 값은 8'hAE 가 된다.)

넷 (net)

넷은 하드웨어 요소 사이의 연결을 나타낸다.
연결된 장치(device) 값으로 출력 값을 갖게되고, 기본적인 값(default)은 z이다.
기본적으로 <키워드> <변수 명> <변수 값> 으로 나타낸다.

키워드	정의
wire, tri	일반적인 net에 사용
wor, trior	OR 연산을 하는 wire
wand, triand	AND 연산을 하는 wire
trireg	값을 저장하는 net에 사용
tri1	net에 아무 것도 인가되지 않으면 1
tri0	net에 아무 것도 인가되지 않으면 0
supply1	Vdd를 나타냄
supply0	Ground를 나타냄

*** net 사용 예시**



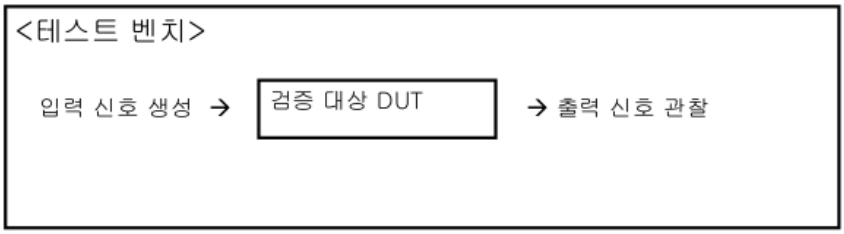
?

```
// 인버터(inverter)와 NOR 게이트 사이를 연결
module connection(y_out, A, B)

output y_out;

input A, B;

wire y1;

not (y1, B);
nor (y_out, A, y1);
```

그림 출처: http://www.rebas.kr/212 [PROJECT REBAS]

8. 테스트 밴치 (Testbench)

- **시뮬레이션을 위한 모듈**이다.
- **설계한 DUT를 검증하기 위한 목적**으로 사용한다.
- DUT를 내포하고 있으며 **DUT의 입력 값을 생성**하고, **출력 값을 관찰**한다.



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
```

#300은 timescale의 테스트 벤치에 따라 달라짐.

`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 30ns**를 의미한다.
-시뮬레이션 스텝 단위는 시뮬레이션 후 timing diagram에서 보여지는 시간의 resolution을 결정한다.

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
```

---

9. 모듈의 호출

- 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 module이 라 한다.
- Sub module을 호출 할 때에는 항상 instance 명을 사용한다.

```
module tb_test();
...
test test_inst(<포트 리스트>); //DUT 호출
endmodule
vivado design flow
```

- **instance 명이 모듈**의 **이름과 동일** 하여도 된다. (모듈을 **한번만 호출 할 때 가능**)
- **하나의 모듈을 설계한 후** 이 **모듈을 여러 번 불러서 사용해야 할 경우**에 유용하다.
- **동일한 모듈**을 **여러 번 호출**할 때마다 **instance 명을 달리 해야** 한다.

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

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.)

```
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;

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

assign X = A + 3'd1;
```

모듈의 내부신호

- **모듈 내에서 값을 저장할 필요**가 있거나 **모듈과 모듈 간에 값을 연결할 필요**가 있을 때 생성한다.
- **always 문을 사용하여 값을 생성**할 필요가 있을 때에는 **reg 선언**을 하여 사용한다.
- **assign 문을 사용하여 값을 생성**하거나, **서로 다른 두 개의 sub module간에 값**을 연결할 때에는 **wire 선언**을 하여 사용한다.

```
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;

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

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

- **sub_mod**라는 모듈은 **2개의 입력을 받아**서 **2 신호의 차를 출력**한다. 여기서는 sub_mod라는 모듈을 2번 호출하였다.

- **add_value** 라는 레지스터는 **test 모듈의 입력 A, B의 합을 저장**한다. **Y 라는 wire**는 **sub_mod0의 출력을 받아**서 **sub_mod1의 입력으로 연결**하는데 사용하였다.

- **Z는 output신호**이며, **sub_mod1의 출력을 받아**서 **output으로 연결**한 것이다.

모듈의 포트 Mapping 방법

<직접 Mapping>
- 모듈과 모듈간에 **포트의 위치**를 **정확하게 1대1로 연결**하는 방법.
- 각 포트들의 **순서가 매우 중요**하다.

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.

<참조 Mapping>
- 모듈과 모듈간에 **포트의 이름을 찾아서 연결**하는 방법.
- 포트의 **순서는 중요하지 않으며** **이름만 정확하게 참조**하면 된다.
- **<서브 모듈의 포트 이름>(<현재 모듈의 포트 이름>),**
- 프로그래머의 실수에 의한 신호의 오동작을 줄여준다.

```
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 순서가바뀌었지만, 포트의 연결은 A-I, B-J로 연결됨으로 좌측과 같다.
(순서는 중요하지 않고 이름만 정확히 참조 하면 된다.)

결과적으로 위에 직접 참조와 같은 형태라고 보면된다.

```
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
```