VHDL이란?

목차

작성자 : 이종윤

1. VHDL 을 시작하기전에.

VHDL 이란? 하고 인터넷에 치면 정리가 잘 되어있는 내용들이 많이 나온다. 그 중에 대표적인게 미국방성,IEEE 관련 여러 이야기와 History가 나오는데 여기서는 그러한 이야기들을 피하기로 한다. 여기서 VHDL을 설명할려는것의 목적은 가장 쉽게 최소한의 내용만을 가지고, 내가 하고자 하는 것을 하는데 있어서 그리 어렵지 않게 하고자 함이다.
그리고, 처음 하는 사람들은 내용이 쉬어야 관심도 가지고 "아! 알고 보니 별거 아니구나" 하면서 조금씩 조금씩 실력을 쌓아가고자 하게 함에 있다.
시중에 나와있는 좋은 책들도 많고, 거기에 있는 내용들도 많다.
근데, 실제 내가 코딩을 하다 보면 엔지니어마다 습관적으로 자기가 쓰는 문법만 쓰게 되고 거기에 익숙해진다면 몇 가지 문법만 가지고 왠만한것들을 다 할 수가 있게 된다.
앞으로 쓰고자 하는 내용들은 처음 VHDL을 시작하는 사람들을 기준으로 설명하는것이기 때문에 본인 스스로 실력이 있다고 생각하는 사람들은 굳이 참고할 이유가 없다.
그리고, VHDL을 공부하기전에 기본적으로 AND,OR,XOR 게이트, 논리회로, 동기신호, 플립플롭 등의 내용들을 알고 있어야 한다.
VHDL은 언뜻 보면 C언어와 같이 생겼다. 일단 영어로 되어있으니깐..^^
이 영어로 되어있는 언어를 가지고 VHDL은 바로 디지털 논리회로를 설계하는것이다.
예전에 게이트로 되어있는 IC를 여러 개 연결해서 카운터설계하던것들을 VHDL은 영어로 된 언어로 설계를 할 수가 있다. 그렇다고 한다면 기본적으로 내 머리속에 Digital 논리회로에 대한 기본개념이 있어야 이해하기가 편하다. 그 중에서 가장 중요한 것은 동기신호라는 개념이다. 이러한것들은 한두줄의 글로서 모두 이해할 수가 없는것이기 때문에 조금씩 관심을 가지고 개념정리를 해 나가야 한다.
그럼, VHDL 에 대해서 설명을 시작하겠다.
VHDL은 위에서 말한것과 같이 디지털 논리회로를 설계하는 것이다. 그럼, 이 VHDL이 왜 생겼을까를 보면 우리들은 흔히 디지털논리회로를 구성하게 되면 이미 시중에 나와있는 IC를 이용해서 회로를 꾸미게 된다. 그런데, 만약 내가 조금만 복잡한 회로를 꾸미게 된다면 거기에 따른 IC의 수는 기하급수적으로 늘어나게 된다. 디지털이라는것은 1 과 0 으로만 이루어져있는데, 만약 10개의 디지털 입력값을 받아서 중간에 여러 연산을 거쳐 10개의 출력을 다시 내보내겠다 한다면 수백개의 게이트소자가 필요로 할것이다. 그러면 이러한 수백개의 게이트 소자를 일일이 납땜하고 연결하다보면 아마 보드도 상당히 커지게 될것이다.
그런데, 하다보면 수백개까지는 어떻게 해볼수 있다고 해도 만약 수천,수만,수십만,수백만 게이트 레벨로 올라가게 된다면 어찌될까? 그러한 보드는 생산하지도 못할것이고 일일이 설계하는 것 자체도 불가능할것이다. 그리고, 그때 그때 필요한기능을 추가하거나 빼거나 할때도 만만치 않을것이다.
그렇기 때문에 VHDL은 FPGA,CPLD 라는 칩을 이용해서 내가 원하는 기능을 그때 그때

수정해 가면서 설계를 할 수가 있다.
실제로 나는 VHDL이라는 언어로 설계를 하지만 그 설계된 VHDL을 컴파일(로직합성)하게 되면 수많은 게이트와 플립플럽들로 내가 설계한게 바뀌게 되고, 그 바뀐구조로 FPGA,CPLD 안에 집적되어 있는 게이트들이 내가 설계한데로 구성된다고 생각하면 편하다.
우리는 PIC, 8051, AVR, ARM 과 같은 MCU 와 고속데이타 처리를 위한 DSP 등을 가지고, C 라는 언어를 가지고 많은 전자제품을 개발하고 있다. 실제로 MCU와 DSP만을 가지고도 많은 일을 할 수 있는 것은 사실이다. 하지만 전자제품은 하루가 멀다하고 발전을 하기 때문에 아무래도 MCU나 DSP와 같은 것은 그때 그때 필요로 하는 부분에 모두 적용하기가 만만치 않은 부분이 있다. MCU 나 DSP 같은 경우는 아무래도 어떤 특정한 일이나 적용하고자 하는 타겟을 가지고, 많이 적용될수 있는 분야로 대체로 만들어지는것이지 나 만을 위해서 또는 몇군데 업체만을 위해서 만들어지는 것이 아니기 때문이다.
하지만 FPGA,CPLD 를 이용해서 VHDL로 구현을 한다면 간단한 몇줄의 코딩만으로 내가 원하는 나만의 IC를 만들수 있다고 생각하면 편하다.
그리고, 최대의 장점은 MCU나 DSP보다 훨씬 빠른 고속 데이터 처리가 가능하다.
일반 MCU 같은 경우는 10~20Mhz정도의 CLK을 가지고 명령어를 처리하기 때문에 외부의 컨트롤 핀이나 통신을 처리하는데 있어서 10~20Mhz보다 몇배는 느린 데이터 처리만 가능하게 된다. 이러한 부분을 보완하기 위해서 DSP,ARM 이라는 것이 나왔고, 현재는 내부 코아 속도가 1~2Gbps 단위까지도 올라갔지만 실제로 이 DSP,ARM도 결국에는 정해져 있는 타겟에 맞춰서 나온 칩이기 때문에 많은 응용분야가 있기는 하지만 이러한 칩들을 쓸려면 많은 노력이 필요하기도 하고, 내가 원하는 기능에 따라서 그때 그때 적용하는데 있어서 굉장히 비효율적인 부분이 있기 마련이다.
그렇기 때문에 MCU, ARM, DSP, FPGA 등을 이용해서 그때 그때 상황에 맞는 제품 개발이 필요하다.

2. VHDL구조

VHDL 은 library, entity, architecture, process 형태로 하나의 VHDL구문을 만들수 있다.

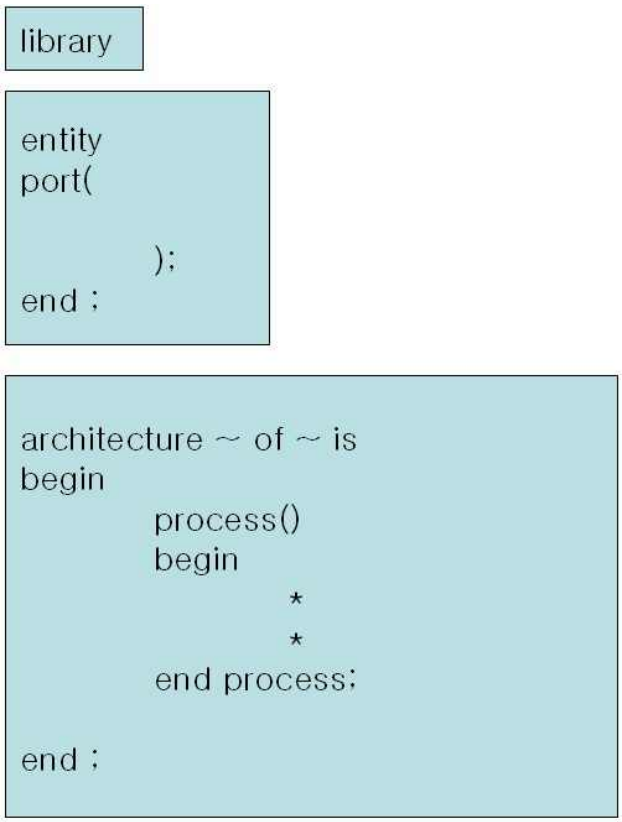


그림 2-1

위 그림2-1 의 형태와 같이 한 개의 **.vhd 구문이 되는데, 각각에 대해서 알아보기로하자.

Library : 우리가 VHDL 문법을 사용하고자 하는데 있어서 필요한 문법이 어떤형식으로 정의되어있는지 정리되어있는거라 생각하면 편하다. 즉, 우리가 VHDL 코딩을 하고나서 컴파일을 할 때 컴파일러가 코딩안에 있는 문법들을 컴파일하면서 이미 정의되어있는 library를 참고하게 된다. 만약 library에 정의되어있지 않은 문법이나 표현을 하게되면 컴파일러는 당연히 알지못하는 언어로 인식하게 된다. 즉, 이미 누군가 내가 좀더 편하게 VHDL이라는 언어를 사용할수 있게 많이 사용되는 표현들을 미리 정의해서 저장해놓은것이고, 나는 이미 정의 되어있는 것을 불러다 쓰는 형식이라 생각하면 쉽다.
이 Library에는 package라는 것이 종속되는데 이 package는 여러가지 형태의 함수나, 자료형등이 몇가지 형태로 나뉘어 구분되어있는것이라 생각하면 된다.

Library 표현법을 보면 아래와 같다.

이제 자주 사용하는 library에 대해서 알아보자.

use ieee.std_logic_1164.all;

- VHDL을 하는데 있어서 가장 기본적인 구문들이 정의되어있다고 보면 된다. std_logic, std_logic_vector 와 같이 코딩을 하는데 있어서 가장 기본이 되기 때문에 항상 사용이 된다고 보면 된다.

use ieee.std_logic_arith.all;

-VHDL 을 할 때 비트("11110000")로 나타내는 경우와 정수(240)로 나태내는 경우가 생길 수도 있고, 편의에 의해서 사용할수도 있다. 이러한 240과 같은 integer 형을 사용할 수가 있다.

use ieee.std_logic_unsigned.all;

-연산을 할 때 양수로 표현할 때 사용한다.

use ieee.std_logic_signed.all;

-연산을 할 때 음수로 표현할 때 사용한다.

일반적인 코딩을 할 때 나는

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

를 주로 사용하고, 음수연산이 있을때는 unsigned => signed로 바꾸어 사용한다.

entity : entity 안에는 port라고 정의를 내리는 부분이 있다. 이 port에는 xxx.vhd 코딩문의 입출력부분을 정의를 내리는 곳이라 생각하면 된다.

그림 2-2

위 그림 2-2와 같이 A,B 라는 Input을 받아서 그 두개의 입력 상태에 따라 변화하는 C라는 출력을 내보내주는 코딩을 하겠다고 할 때 이 블록에서는

입력 : A, B

출력 : C

가 된다. 이 A,B,C 를 바로 아래와 같이 entity문에 선언을 해준다.

```
entity test  is
port(
        A : in std_logic ;
        B : in std_logic ;
        C : out std_logic );
end  test ;
```

이 블록의 전체 이름으로 지어주면 된다.

마지막 Port에는 " ; " 을 해주지 말고, port()가 끝나고 ";" 을 붙여준다.

즉, entity문에는 블록의 입,출력 port를 미리 선언을 해준다고 생각하면 된다. 그렇게 되면 이 entity문을 보고, 현재 이 블록에 어떤한 입력과 어떤 출력포트들로 구성되어있는지 블록의 그림을 머리속에서 그릴수가 있다.

architecture : architecture안에는 실제 그림 2-2블록안에서 동작하는 부분을 코딩해주게 되고, 그 안에서 사용되는 여러가지 C에서의 변수와 같은것들도 선언해주게 된다.

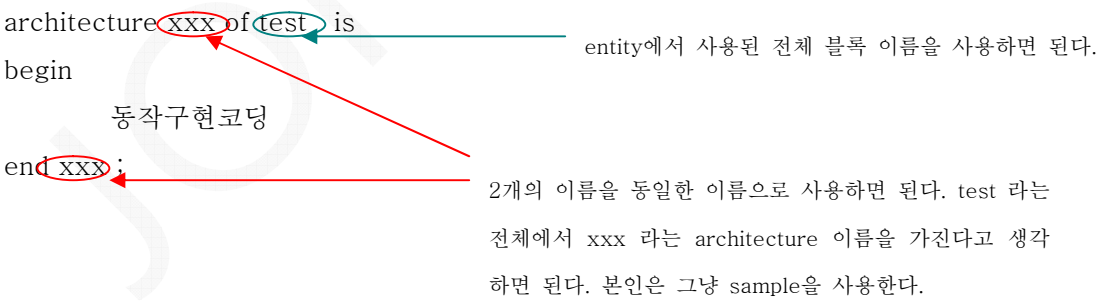
그리고, process()문이라고 해서 좀더 세부적이고, 기술적으로 표현할수 있는 문도 이 architecture 안에서 구현이 된다.

만약 위의 그림 2-2 에서의 블록안에서 C는 A xor B 의 결과값을 출력해주게 하겠다고 한다면 다음과 같이 2가지 방법으로 구현할수 있다.

architecture sample of test is
begin
C <= A xor B;
 	
end sample;
	architecture sample of test is
	begin
	process(A,B)
	begin
	if (A=B) then
	C <= '0';
	else
	C <= '1';
	end if;
	end process;
	end sample;

2가지 표현법에 대해서 어떤 차이가 있는지를 여기서 이해하고 넘어가는 것이 좋다.
언뜻 보기에는 오른쪽 process 문이 복잡해보기는 하지만 실제로 VHDL 코딩을 하는것은 오른쪽과 같이 process문을 코딩하는일이 대부분이다.
왼쪽과 같은 경우는 우리가 단순한 게이트의 조합정도를 표현할수 있지만 오른쪽은 우리가 상상하는 추상적인 개념을 실제로 하나하나 VHDL이라는 언어로 표현할 수가 있는것이기 때문이다. 즉, 우리가 말로서 표현할수 있는 것을 그대로 언어로 옮겨놓은거라 생각하면 된다. 그리고, 앞에서 이야기 했던 동기신호에 대한 구현을 하기 위해서는 이 process문을 사용해야 한다.

그리고, archtecture문을 보면 다음과 같이 표현이 된다.



그럼, 지금까지 설명했던 library,entity,architecture 를 하나로 합쳐보면 아래와 같은 문장을 완성할수 있다.

```
library ieee;  본인은 항상 이 library를 기본으로 사용한다.
               음수 연산을 할때만 unsigned => signed로 수정한다.
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
        A : in std_logic ;
        B : in std_logic ;
        C : out std_logic );
end test ;

architecture sample of test is
begin

        C <= A xor B;

end sample;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
        A : in std_logic ;
        B : in std_logic ;
        C : out std_logic );
end test ;

architecture sample of test is
begin

        process(A,B)
        begin
                if(A=B)then
                        C <= '0';
                else
                        C <= '1';
                end if;
        end process;
end sample;
```

여기까지 드디어 완전한 test.vhd 문장을 만드는데 성공을 하였다.
보통 처음 VHDL 구문을 코딩할 때 에러메세지가 나오고, 실수를 하는 부분은 지금 이 짧은 문장안에서 거의 다 나온다.
entity 문에서 마지막에 ‘ ; ’ 을 ()끝나고 안한다거나, entity 이름과 architecture 문 이름이 같지 않거나 process에서 begin을 놓친다거나 라이브러리와 코딩안에서의 문법이 맞지 않거나하는 부분에서 에러를 많이 발생시키고, 실수도 많이 하게 된다.
지금 이 짧은 문장을 직접 손으로 코딩하고 컴파일해보면서 손과,눈에 익숙하게 하는게 좋다.

3. signal,variable,constant 사용법.

signal	VHDL에서 로직 합성을 할 때 서로 연결해주는 선의 역할을 한다. 우리 눈으로 확인할수 있는 데이터 값의 이동을 대부분 이 signal을 통해서 한다.
variable	연산을 할 때 중간에 값을 저장하는 변수형태인데 signal하고는 확연한 차이가 있기 때문에 차이점을 분명히 알아두어야 한다.
constant	일반적인 상수값을 초기에 선언하는 용도로 사용한다. 한번 선언된 상수값은 변경할수 없다.

3-1. signal 의 특징

* 로직 합성시 선으로 나타난다.
* 대입기호 <= 를 사용한다.
* 어느 일정한 시점에서 값이 대입된다. process() 문에서는 end process 후에 대입된다.
* architecture 문에서 begin 전에 선언된다.
* entity에서의 port내에서도 선언된다.
* 초기화 선언할때는 := 를 사용한다.

Ex)

```
entity XXX is
Port ( A1, A2 : in std_logic ;
       B1     : out std_logic ;                      -- A1,A2,B1,C1 을
       C1     : out std_logic_vector(2 downto 0)  -- signal 로 선언
);
end XXX;

architecture xxx of XXX is
signal D1, D2   : std_logic ;                                  -- 1bit signal
signal D3       : std_logic_vector(1 downto 0);                -- 2bit signal
signal D4       : std_logic_vector(2 downto 0) := "110";  -- 3bit signal
                                                                초기값 셋팅
begin
```

다음은 실제 signal을 사용하는 방법과 그 signal이 로직합성시에 어떤 형태로 나오는지 알아보겠다.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
      A : in std_logic ;
      B : in std_logic ;

      C : in std_logic ;
      D : in std_logic ;

      F : out std_logic );
end test ;

architecture sample of test is

signal pin1    : std_logic ;
signal pin2    : std_logic ;

begin

      pin1 <= A and B ;

      pin2 <= C xor D ;

      F    <= pin1 nand pin2 ;

end sample;
```

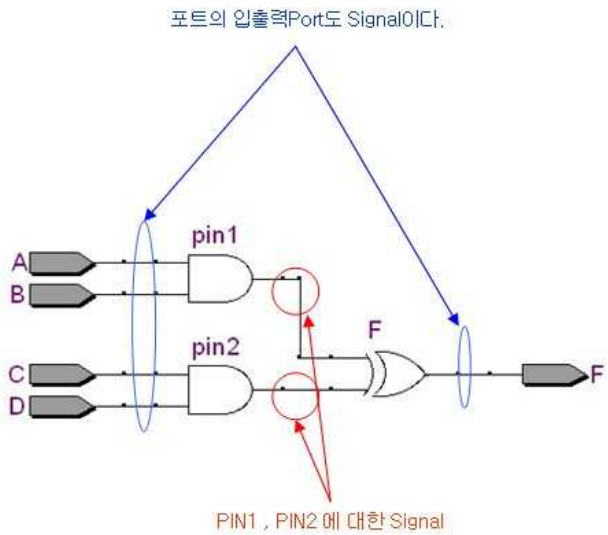


그림 3-1

왼쪽의 VHDL 구문이 실제로 컴파일되고, 로직으로 합성됐을 때 오른쪽 그림3-1처럼 나온다.

그림으로 봤을 때 port(A,B,C,D,F) , architecture(pin1,pin2) 가 실제로 어떤 형태로 서로간에 연결이 되는지 좀더 명확히 알수가 있다.

3-2. variable 의 특징

* 로직합성시 선으로 연결되는 것이 아닌 중간단계인 변수 형태로 존재한다.
* function,procedure,process 문에서만 사용이 된다.
  (function,procedure에 대해서는 여기서 설명하지 않고, process에 대해서만 설명하겠다.)
* 선언된 곳에서만 사용할수 있는 변수이다.
* 대입기호 := 를 사용한다.
* 값이 대입기호(:=)를 만나면 그 즉시 대입된다.

Ex)

```
architecture xxx of XXX is
begin
        process()
        variable D1, D2   : std_logic ;                          --1bit variable
        variable D3       : std_logic_vector(1 downto 0);        --2bit variable
        variable D4       : std_logic_vector(2 downto 0) := "110";  --3bit variable
        begin
                .
                .
        end process;
end xxx;
```

다음은 실제 variable 을 사용하는 방법과 그 variable이 로직합성시에 어떤 형태로 나오는지와 variable을 signal형태로 바꿨을때는 어떻게 변하는지를 알아보겠다. 이 두개의 차이점은 상당히 중요하니 반드시 이해하고 넘어가야 한다.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
      A : in std_logic ;
      B : in std_logic ;
      C : in std_logic ;

      D : out std_logic );
end test ;

architecture sample of test is

begin

   process(A,B,C)
      variable  pin2    : std_logic := '0';
   begin
     pin2 := '1' ;                    -->  Pin2 에 '1' 즉시 대입
     pin2 := A and pin2 ;             -->  Pin2 에 '1' and A 즉시 대입
     pin2 := B and pin2 ;             -->  Pin2 에 ('1' and A) and B 즉시 대입
     pin2 := C and pin2 ;             -->  Pin2 에 (('1' and A) and B) and C  즉시 대입
     D    <= pin2 ;                   -->  D 에 (('1' and A) and B) and C  대입
   end process;                       결론적으로  D <= A and B and C

end sample;
```

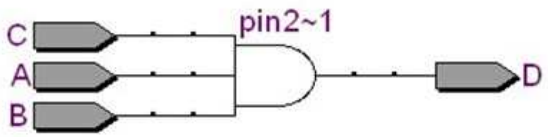


그림 3-2

왼쪽의 variable 값이 변하는 것을 쫓아가다 보면 결과적으로 A,B,C 3개의 입력을 and한값을 D에 출력해주는 결과가 된다.
그리고, 왼쪽의 구문을 로직 합성시에는 그림3-2와 같은 로직 합성결과가 나오게 된다.
이제 이 variable pin2 를 signal형태로 바꿨을때는 어떻게 변하는지 확인해보자.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
      A : in std_logic ;
      B : in std_logic ;
      C : in std_logic ;

      D : out std_logic );
end test ;

architecture sample of test is

   signal  pin2      : std_logic := '0';

begin

   process(A,B,C)
   begin
    pin2 <= '1' ;          ------------>  Pin2 에 대입 대기
    pin2 <= A and pin2 ;   ------------>  Pin2 에 A and Pin2 대입 대기
    pin2 <= B and pin2 ;   ------------>  Pin2 에 B and Pin2 대입 대기
    pin2 <= C and pin2 ;   ------------>  Pin2 에 C and Pin2 대입 대기
    D    <= pin2 ;         ------------>  D 에 Pin2( C and Pin2) 대입 대기
   end process;           ------------>  D 에  C and Pin2 대입
end sample;
```

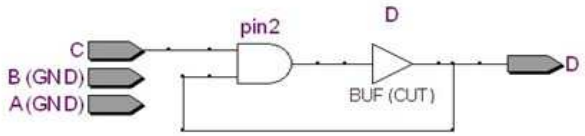


그림 3-3

왼쪽 구문을 보면 실제로 값이 즉시 대입되지 않고 대기를 하고 있다가 마지막 end process; 를 만나고 나서 값이 대입되는 것을 발견할수 있다.
그러다 보니 그전값들은 무시가 되고 마지막 pin2 <= C and pin2; 에 대한 처리만 이루어진다.
그림 3-2 와 3-3 을 비교하면서 VHDL구문을 이해하면 좀더 variable 과 signal에 대한 이해를 하기 편할것이다.

3-3. constant 의 특징

* 초기에 상수값을 선언하는데 사용한다.
* 대입기호 := 을 사용한다.
* architecture 문에서 begin 전에 선언된다.
* 한번 선언이 되면 그 값을 바꿀수가 없다.

Ex)

```
architecture xxx of XXX is
        constant X : std_logic_vector(2 downto 0) := conv_std_logic_vector(4,3);
        constant Y : std_logic_vector(1 downto 0) := conv_std_logic_vector(2,2);
        constant Z : std_logic_vector(1 downto 0) := "11";
begin
```

다음은 실제 constant 를 사용하는 방법과 로직합성시에 어떤 형태로 나오는지 보자.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
      A : in std_logic ;
      B : in std_logic ;
      C : in std_logic ;

      D : out std_logic_vector(3 downto 0);
      E : out std_logic_vector(2 downto 0);
      F : out std_logic_vector(2 downto 0)
      );
end test ;

architecture sample of test is

   constant X : std_logic_vector(2 downto 0) := conv_std_logic_vector(2,3);
   constant Y : std_logic_vector(1 downto 0) := conv_std_logic_vector(1,2);
   constant Z : std_logic_vector(1 downto 0) := "10";

begin

   D <= X + "00" & A ;
   E <= Y + "0" & B ;
   F <= Z + "0" & C ;

end sample;
```

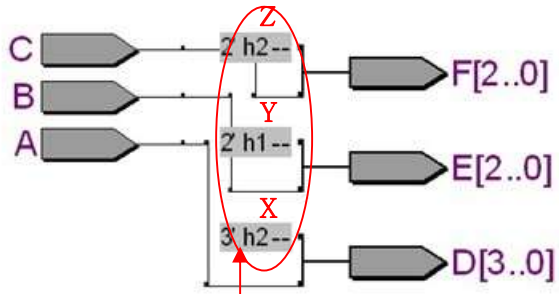


그림 3-4

로직 내에서 선언된 상수값으로 존재하는 것을 확인할수 잇다.

4. 연산자

연산자는 VHDL에서 이미 정의된 여러 개의 연산자가 있고, 우선순위가 정해져 있다. 연산자의 표는 아래와 같다.

VHDL 연산자

논리 연산자	and,or,nand,nor,xor	낮다 우선순위
관계 연산자	=, /= , < , <= , > , >=	
덧셈 연산자	+ , - , &	
단항 연산자(부호)	+ , -	
곱셈연산자	* , / , mod , rem	
기타연산자	** , abs , not	높다

표 4-1

4-1 : 논리 연산자

논리 연산자는 연산자 좌,우에 있는 두 값에 대한 논리 연산을 수행하고, 두 값의 data type은 같아야 한다.

'1' = 참 , '0' = 거짓

A	B	A and B	A or B	A nand B	A nor B	A xor B
1	1	1	1	0	0	0
1	0	0	1	1	0	1
0	1	0	1	1	0	1
0	0	0	0	1	1	0

표 4-2

논리 연산자를 여러 개 사용하게 될때는 항상 괄호를 사용하는 습관을 가지도록 해야 한다. 연산자의 우선순위를 항상 기억하고 있을수 없기 때문에 괄호를 사용해서 내가 직접 우선순위를 정해주는 것이 좋다.

Ex)

```
Z <= (((A and B) or C) nand D);
```

4-2 : 관계 연산자

관계 연산자는 연산자 좌,우에 있는 두 값에 대한 크기를 비교하는데 사용된다.
data type은 같아야 한다. 주로 if 문에서 사용된다.

Ex)

```
if(A = B)then
        C <= '0' ;        -- A = B 가 참이면 수행
else
        C <= '1' ;        -- A = B 가 참이 아니면 수행
end if;
```

4-3 : 덧셈 연산자

덧셈 연산자는 덧셈(+),뺄셈(-) 연산자와 접속 연산자(&) 를 기억해야 한다.

Ex)

```
signal A : std_logic_vector(3 downto 0) := "1001" ;
signal B : std_logic_vector(2 downto 0) := "001" ;
```

A & B : A(3) A(2) A(1) A(0) B(2) B(1) B(0)
A & B : "1001001"

4-5 : 곱셈 연산자

곱셈 연산자에서 곱셈,나눗셈 연산자의 좌,우에 있는 두 값의 data type은 같아야한다.

A rem B (나머지계산) : A 의 부호를 따른다.
결과의 절대값은 B의 절대값보다 작다.
A = (A/B) * B + (A rem B)

A mod B : B 의 부호를 따른다.
결과의 절대값이 B의 절대값보다 작다.
A = B*N + (A mod B)

A	B	A/B	A rem B	A mod B
11	4	2	3	3
-11	4	-2	-3	1
11	-4	-2	3	-1
-11	-4	2	-3	-3

표 4-3

5. process 문

process문은 architecture문안에서 실행되며, 여러 process문을 만들수가 있다.
이들 각각의 process문은 동시에 병행처리가 가능하고, process문 각각의 내부는 순차처리문으로 동작한다.
그리고, 각각의 process문에서 사용되는 signal은 다른 process문에서도 사용될수 있다. 그리고, process문 안에서는 우리가 말로 동작을 표현할수 있는것처럼 서술이 된다.

다음의 process문을 분석해보자.

```
process(A,B)            -- A,B 는 process문안에서 어떤 동작에 영향을 줄수 있는 값이 들어간다.
begin                   -- process의 시작을 알려준다.
    if(A=B)then         -- A 와 B 가 같다면
            C <= '0';   -- C 에 0 의 값을 대입하고,
    else                -- 그렇지 않다면 (A와 B가 틀리다면)
            C <= '1';   -- C 에 1 의 값을 대입해라.
    end if;             -- if 문을 마친다.
end process;            -- process 문을 마친다.
```

위 process문안에서 서술된 말은 우리가 흔히 알고 있는 C = A xor B 를 나타내고있다.
즉, A xor B 에 대해서 서술형으로 풀어놓은거라 생각하면 된다.
만약, 당신의 머리에 논리적인 어떠한 내용이 있다면 그것을 위와 같은 방법으로 process문 안에 펼쳐놓으면 그 자체가 process문이 되는것이다.
그리고, 설명을 위에서 아래로 내려가면서 하듯이 process 문 안에 있는 내용도 순차적으로 실행한다.
그리고, process(A,B) 안에 있는 A,B 는 process안에서 어떤 signal이나variable 값등에 영향을 줄수 있는 값이다. A와 B 의 값에 따라서 C의 값이 달라지듯이 C에 영향을 주는 값이다. 즉, prceess 문 안에서 어떠한 형태로든 변화를 줄수 있는 외부적인 요인이 되는값을 ()괄호 안에 넣어주는것이다.
또한 이 ()괄호 안에 들어가는 값은 process문 밖에 있는 값이어야 한다.

다음의 VHDL구문을 보자.

```
entity test is
port(
      A : in std_logic ;
      B : in std_logic ;
      C : in std_logic ;
      D : in std_logic ;
      E : in std_logic ;
      F : in std_logic ;

      AB : out std_logic ;
      CD : out std_logic ;
      EF : out std_logic
      );
end test ;

architecture sample of test is
begin

   process(A,B)
   begin
      if(A = B)then
         AB <= '0';
      else
         AB <= '1';
      end if;
   end process;

   process(C,D)
   begin
      if(C = '1' and D = '1')then
         CD <= '1';
      else
         CD <= '0';
      end if;
   end process;

   process(E,F)
   begin
      if(E = '1' or F = '1')then
         EF <= '1';
      else
         EF <= '0';
      end if;
   end process;

end sample;
```

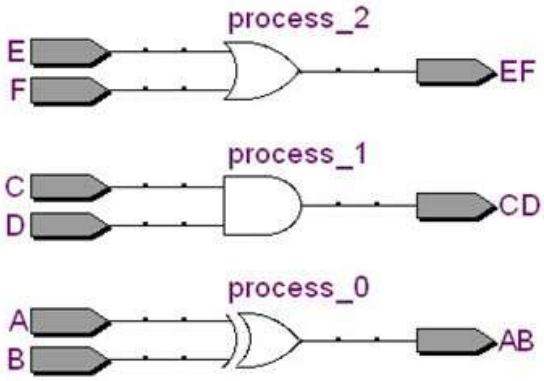


그림 5-1

-- (1) A xor B

순차적으로 처리

-- (1) C and D

순차적으로 처리

-- (1) E or F

순차적으로 처리

동시에 시작

왼쪽의 구문을 보면 총 3개의 process가 있다.

첫번째 : A xor B 를 서술.

두번째 : C and D 를 서술.

세번째 : E or F 를 서술.

이 3개의 process는 동시에 처리가 되고, 각 process안에서는 순차처리가 된다. 오른쪽의 그림5-1의 로직합성후를 보면 좀더 이해가 쉽다.

6. if 문

if문은 process내부에서 실행되는 순차처리문이다.
그리고, 어떤 조건을 기준으로 그 조건이 참이면 실행되는 형식을 가지고 있다. 조건의 경우의 수에 따라서 다음과 같은 if 문으로 구성된다.

***조건이 1개일 때**

```
if ( a = '1')then                -- a 의 값이 '1' 이라면
        수행문장1.               -- 수행문장1을 수행한다.
end if;                          -- if문이 끝남.
```

***조건이 2개일 때**

```
if ( a = '1')then                -- a 의 값이 '1' 이라면
        수행문장1.               -- 수행문장1 을 수행한다.
else                             -- a 의 값이 '1' 이 아니라면
        수행문장2.               -- 수행문장2 를 수행한다.
end if;
```

***조건이 3개이상일때**

```
if ( a = "00")then               -- a 의 값이 "00" 이라면
        수행문장1 .              -- 수행문장1 을 수행한다.
elsif( a = "01")then             -- 그렇지 않고, a 의 값이 "01" 이라면
        수행문장2.               -- 수행문장2 를 수행한다.
elsif( a = "10")then             -- 그렇지 않고, a 의 값이 "10" 이라면
        수행문장3.               -- 수행문장3 을 수행한다.
else                             -- a 의 값이 "00","01","10" 이 아니라면.
        수행문장4.               -- 수행문장4 를 수행한다.
end if;
```

그리고, if 안의 조건에는 다음과 같은 논리연산자와 관계연산자를 이용해서 여러 조건을 만들수가 있다.

```
if((a = "00") and (b = "10"))then  -- a 의 값이 "00" 이고, b 의 값이 "10" 이면 수행
if((a > 10) and (a < 20))then      -- a 의 값이 10 보다 크고 20 보다 작으면 수행.
if((a = "01") or (a = "10"))then   -- a 의 값이 "01" 이면 수행 또는 "10" 이어도 수행.
```

이제 다음의 두 문장을 비교해보자.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
      A     : in std_logic ;
      CLK   : in std_logic ;

      B     : out std_logic
      );
end test ;

architecture sample of test is
begin
      process(A)
      begin
         if(CLK'event and CLK = '1')then
            B <= A ;
         end if;
      end process;

end sample;
```

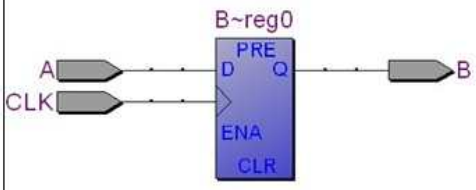


그림6-1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test is
port(
      A     : in std_logic ;
      CLK   : in std_logic ;

      B     : out std_logic
      );
end test ;

architecture sample of test is
begin
      process(A)
      begin
         if(CLK = '1')then
            B <= A ;
         end if;
      end process;

end sample;
```

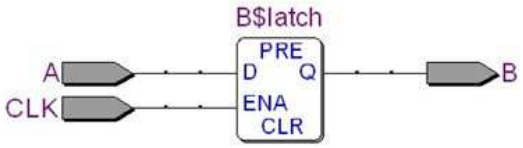
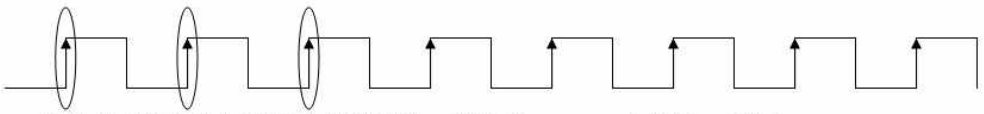


그림6-2

왼쪽과 오른쪽의 두 문장에서 차이점은 if문안에 있는 조건의 차이가 있다.

왼쪽은 if(clk'event and clk = '1')then 이라고 했는데, 상당히 중요한 내용이다. 실제로 여기서 앞서 언급했던 동기신호라는 개념도 나오고 실제 VHDL 코딩의 90%는 이 조건이 항상 들어간다고 생각하면 된다. 개념을 살펴보면



CLK 이 '1' 로 상승할때를 나타낸다. if(CLK'event and CLK = '1')then
만약 '0' 으로 내려갈때를 나타낼려면 if(CLK'event and CLK = '0')then 이다.

그리고, 왼쪽 VHDL을 로직합성했더니 그림 6-1과 같이 D flip/flop으로 로직이 생성됐다.

하지만 그림 6-2를 보면 CLK 이 '1' 로 유지되어있을때 A 의 값을 B 로 보낸다. 이는 CLK이 DATA Enable핀으로 동작한다.

두개의 차이점을 보면 그림6-1은 중간에 값이 바뀌어도 CLK이 '1' 로 올라갈때만 B도 바뀌게 된다. 하지만 그림 6-2는 CLK이 '1'일때 A의 값이 바뀌면 B의 값도 바뀐다.

두개의 차이점을 잘 숙지하고 D flip/flop 이라는거에 대해서도 한번 확인바란다.

7. case 문

if 문과 case문은 조건에 의해서 해당하는 문장을 수행하는점에서는 동일하다. 하지만 대체로 if문은 조건에 따라서 문장을 선택하지만 case문은 어떤 수식자체값에 따라서 문장을 선택하는 조건문으로 많이 사용한다. 예를 들면 진리표같은거에 사용하기 편하다.
표현법은 다음과 같다.

```
case X is                          -- 조건이 되는 x 가
        when "00" =>               -- "00" 이면
                수행문장1          -- 수행문장1을 실행
        when "01" =>               -- "01" 이면
                수행문장2          -- 수행문장2를 실행
        when "10" =>               -- "10" 이면
                수행문장3          -- 수행문장3을 실행
        when others =>             -- "00","01","10" 이 아니면
                수행문장4          -- 수행문장4를 실행
end case;                          -- case문을 마침
```

다음은 if문과 case문을 비교한것이다.

```
entity test is
port(
      A     : in std_logic_vector(1 downto 0) ;
      B     : out std_logic_vector(1 downto 0)
      );
end test ;

architecture sample of test is
begin
      process(A)
      begin
         if(A = "00")then
            B <= "01" ;
         elsif(A = "01")then
            B <= "10" ;
         elsif(A = "10")then
            B <= "11" ;
         else
            B <= "00" ;
         end if;
      end process;
end sample;
```

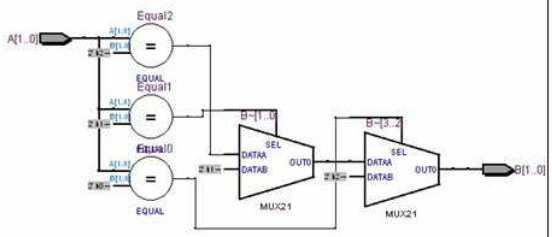


그림 7-1

```
entity test is
port(
      A     : in std_logic_vector(1 downto 0) ;
      B     : out std_logic_vector(1 downto 0)
      );
end test ;

architecture sample of test is
begin
      process(A)
      begin
         case A is
            when "00" =>
               B <= "01" ;
            when "01" =>
               B <= "10" ;
            when "10" =>
               B <= "11" ;
            when others =>
               B <= "00" ;
         end case;
      end process;
end sample;
```

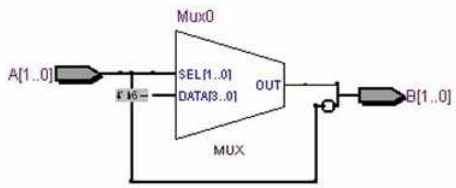


그림 7-2

동일한 기능을 하는 문장을 if 문과 case문으로 만들었을 때 실제 로직합성결과를 보여준다. 위 경우에는 case문으로 했을때가 훨씬 간결한 로직구성이 되는 것을 알수 있다. 이런 차이점을 두고 if 문과 case문을 병행해서 사용하면 된다.

8. loop 문

loop문은 반복처리문장을 보다 간결하게 만들기 위한 문장이라고 생각하면 된다. 몇가지 방식이 있기는 한데, 여기서는 for~loop 문에 대해서만 살펴보기로 하자.
표현법은 다음과 같다.

```
for 루프변수 in 변수범위 loop          -- 루프변수가 변수범위까지
        순차처리문;                     -- 순차처리문을 수행한다.
end loop;
```

다음과 같이 일반적인 문장을 loop문으로 변경할수 있다.

```
entity test is
port(
      A     : in std_logic ;
      CLK   : in std_logic ;

      B     : out std_logic
      );
end test ;

architecture sample of test is

signal A1 : std_logic_vector(5 downto 0);

begin

      process(CLK)
      begin
         if(CLK'event and CLK = '1')then

            A1(0) <= A ;

            A1(1) <= A1(0);
            A1(2) <= A1(1);
            A1(3) <= A1(2);
            A1(4) <= A1(3);
            A1(5) <= A1(4);

            B <= A1(5);

         end if;
      end process;

end sample;
```

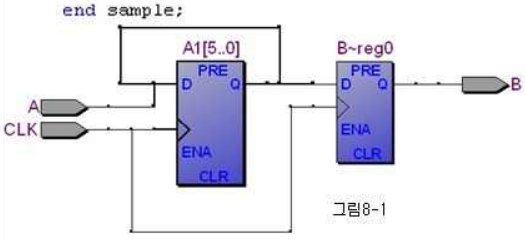


그림8-1

```
entity test is
port(
      A     : in std_logic ;
      CLK   : in std_logic ;

      B     : out std_logic
      );
end test ;

architecture sample of test is

signal A1 : std_logic_vector(5 downto 0);

begin

      process(CLK)
      begin
         if(CLK'event and CLK = '1')then

            A1(0) <= A ;

            for i in 1 to 5 loop
               A1(i) <= A1(i-1);
            end loop;

            B <= A1(5);

         end if;
      end process;

end sample;
```

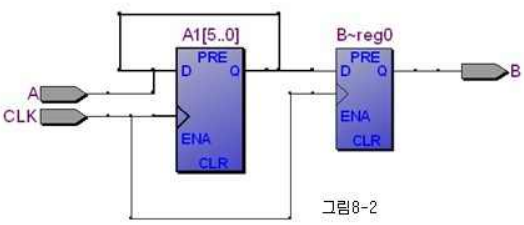


그림8-2

그림 8-1과 8-2를 비교해보면 똑 같은 로직구성을 알수 있지만 VHDL코드상에서는 좀더 효율적으로 코드를 만들수 있다는 것을 알수 있다.

한가지 더 일반적인 문장과 loop문을 연습해보자.

```
entity test is
port(
      A     : in std_logic_vector(5 downto 0) ;
      B     : in std_logic_vector(5 downto 0) ;

      CLK   : in std_logic ;

      C     : out std_logic_vector(5 downto 0)
      );
end test ;

architecture sample of test is
begin

      process(CLK)
      begin
         if(CLK'event and CLK = '1')then
            C(0) <= A(0) xor B(0) ;
            C(1) <= A(1) xor B(1) ;
            C(2) <= A(2) xor B(2) ;
            C(3) <= A(3) xor B(3) ;
            C(4) <= A(4) xor B(4) ;
            C(5) <= A(5) xor B(5) ;

         end if;
      end process;

end sample;
```

```
entity test is
port(
      A     : in std_logic_vector(5 downto 0) ;
      B     : in std_logic_vector(5 downto 0) ;

      CLK   : in std_logic ;

      C     : out std_logic_vector(5 downto 0)
      );
end test ;

architecture sample of test is
begin

      process(CLK)
      begin
         if(CLK'event and CLK = '1')then
            for i in 0 to 5 loop
               C(i) <= A(i) xor B(i) ;
            end loop;

         end if;
      end process;

end sample;
```

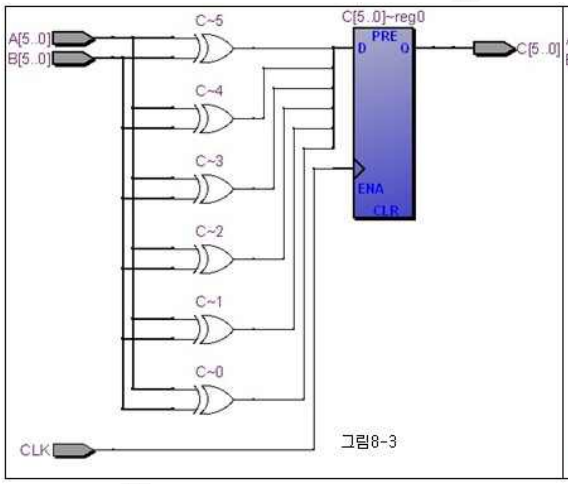


그림8-3

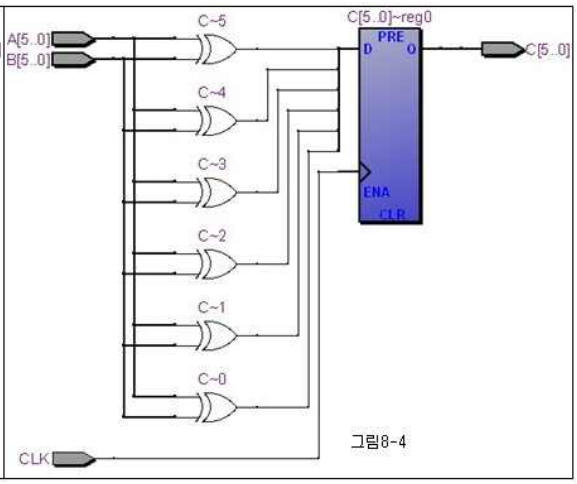


그림8-4

이 두문장을 로직합성을 했을 때도 그림8-3,그림8-4를 보면 동일하다.

이제 loop문에 대해서 어느정도 편리성과 사용법에 대해서는 숙지가 되었을거다. loop문을 잘 활용하면 상당히 코드를 간결하고, 보기 편하게 만들수 있다.

9. component 문

component라는 것은 어느 완전한 VHDL 구문을 불러다 쓰는것이다. VHDL구문을 작성하다보면 좀더 구조적으로 블록을 나누어서 VHDL구문을 작성해야 할때가 필요한데 다양하게 나누어져 있는 VHDL구문을 쉽게 불러다 쓰기 위해서 component를 이용하면 된다.

다음 그림을 살펴보자.

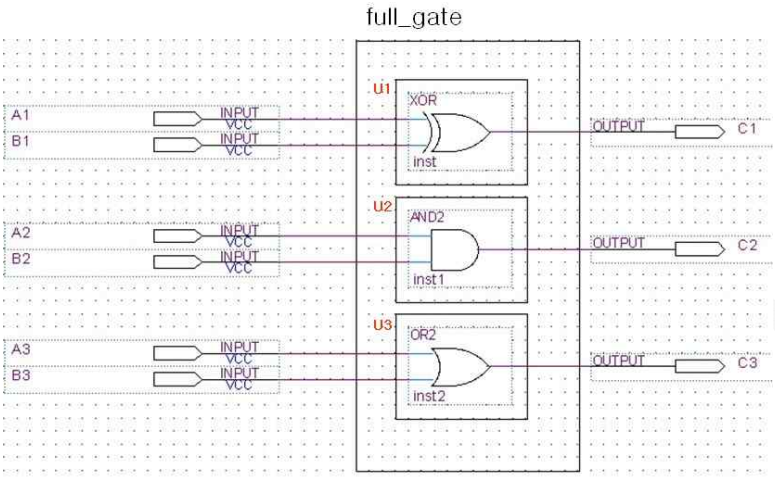


그림9-1

실제로는 full_gate라는 VHDL 구문을 만들거다. 그리고, u1, u2, u3 에 해당하는 VHDL구문이 있다고 할 때 component를 이용해서 full_gate 라는 VHDL구문을 만들어보자.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity gate_or is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end gate_or ;

architecture sample of gate_or is
begin

      process(A,B)
      begin
         if( A = '1' or B = '1')then
            C <= '1' ;
         else
            C <= '0' ;
         end if;
      end process;

end sample;
```

gate_or.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity gate_xor is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end gate_xor ;

architecture sample of gate_xor is
begin

      process(A,B)
      begin
         if( A = B )then
            C <= '0' ;
         else
            C <= '1' ;
         end if;
      end process;

end sample;
```

gate_xor.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity gate_and is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end gate_and ;

architecture sample of gate_and is
begin

      process(A,B)
      begin
         if(A = '1' and B = '1')then
            C <= '1' ;
         else
            C <= '0' ;
         end if;
      end process;

end sample;
```

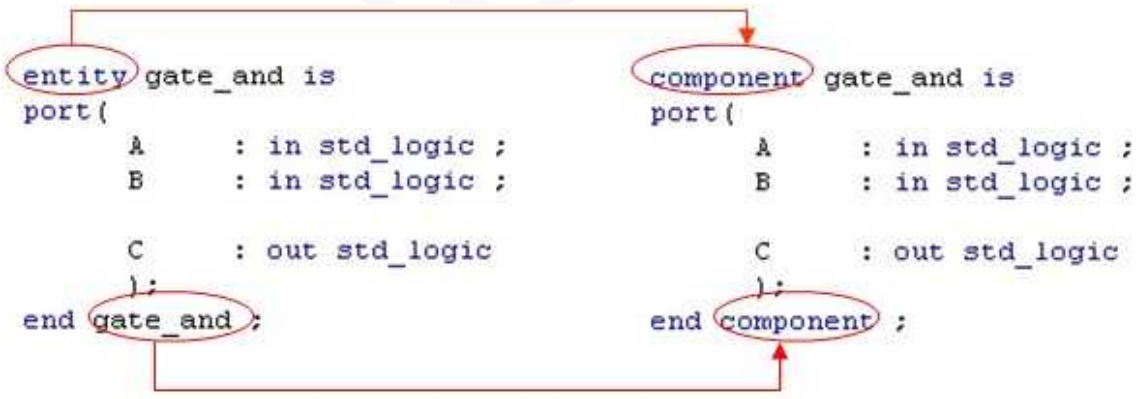
gate_and.vhd

먼저 각각의 OR,XOR,AND 게이트에 해당하는 VHDL구문을 위와 같이 만들어놔야한다.

이제 각각의 부품과 같은 gate_or.vhd, gate_xor.vhd, gate_and.vhd 구문을 component로 만들어보자. 먼저 full_gate.vhd 의 entity 부분을 만들어보자.
그림 9-1을 보면 Input/Output 을 확인하고 아래와 같이 entity 부분을 구성할수있다.

```
entity full_gate is
port(
      A1    : in std_logic ;
      B1    : in std_logic ;

      A2    : in std_logic ;
      B2    : in std_logic ;

      A3    : in std_logic ;
      B3    : in std_logic ;

      C1    : out std_logic ;
      C2    : out std_logic ;
      C3    : out std_logic
      );
end full_gate ;
```

entity 구성이 끝났으면 이제 architecture 부분을 만들어야 하는데, component를 사용하기 위해서 아래와 같이 사용할 VHDL 구문의 entity부분을 가져와 full_gate.vhd 에서 수정하여 사용하면 된다.

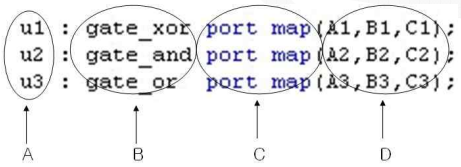


두 구문을 보면 gate_and.vhd에 있던 entity 구문을 가져와서 entity 부분과 end gate_end 부분을 component로 수정하였다.
이 수정된 구문을 component구문이라고 하는데, 이 component 구문은 architecture 와 begin 사이에 들어가게 된다.
그럼, 각각의 component 구문을 만들어서 full_gate.vhd 의 architecture와 begin사이를 완성해보자.

아래는 gate_xor , gate_or , gate_and 부분을 component 구문으로 사용할수 있게 선언해 놓은 것이다.

```
architecture sample of full_gate is

component gate_xor is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end component ;

component gate_or is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end component ;

component gate_and is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end component ;

begin
```

이제 port mapping을 해야 할 부분이 남았다. port mapping하는 방법은 다음과 같다.



A : component를 식별하는 번호정도로 생각하면 된다. 각 component앞에 하나씩 늘어날 때마다 u1,u2,u3...u10 형태로 번호를 붙여주면 된다.

B : 각 component의 VHDL 프로젝트명을 써준다. xxx.vhd 에서 xxx 를 써주면 된다.

C : port map 은 뒤에 오는 구문이 port mapping을 하는 부분이라는 것을 알려준다.

D : 실제로 port mapping 되는 port or signal 을 해당하는 위치에 써넣어준다.
full_gate.vhd 의 A1,B1 Input 을 gate_xor.vhd 의 A,B 에 연결시켜주고, C1 Output을 gate_xor.vhd 의 C에 연결시켜준다.
port mapping은 하드웨어적인 구조로 연결되기 때문에 Input/Output이 서로 틀린 것끼리는 연결시켜줄수 없다.

이제 모든 문장을 하나씩 완성했으니 한 개의 완성된 full_gate.vhd 를 만들어보자.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity full_gate is
port(
      A1    : in std_logic ;
      B1    : in std_logic ;

      A2    : in std_logic ;
      B2    : in std_logic ;

      A3    : in std_logic ;
      B3    : in std_logic ;

      C1    : out std_logic ;
      C2    : out std_logic ;
      C3    : out std_logic
      );
end full_gate ;

architecture sample of full_gate is

component gate_xor is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end component ;
component gate_or is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end component ;

component gate_and is
port(
      A     : in std_logic ;
      B     : in std_logic ;

      C     : out std_logic
      );
end component ;

begin

   u1 : gate_xor port map(A1,B1,C1);
   u2 : gate_and port map(A2,B2,C2);
   u3 : gate_or  port map(A3,B3,C3);

end sample;
```

이제 full_gate.vhd를 완성했다. VHDL구문을 만들 때 구조적으로 블록화해서 만드는 것은

추후에 디버깅을 하거나 다른 프로젝트에서 사용할때도 상당히 유용하니 이 component를 쓰는 법을 잘 이해하고 넘어가는 것이 좋다.

이제 총 4개의 VHDL구문을 가지고 logic 합성을 했을때를 보자.
아래는 full_gate를 실제로 컴파일해서 로직합성한 결과를 보여준다.

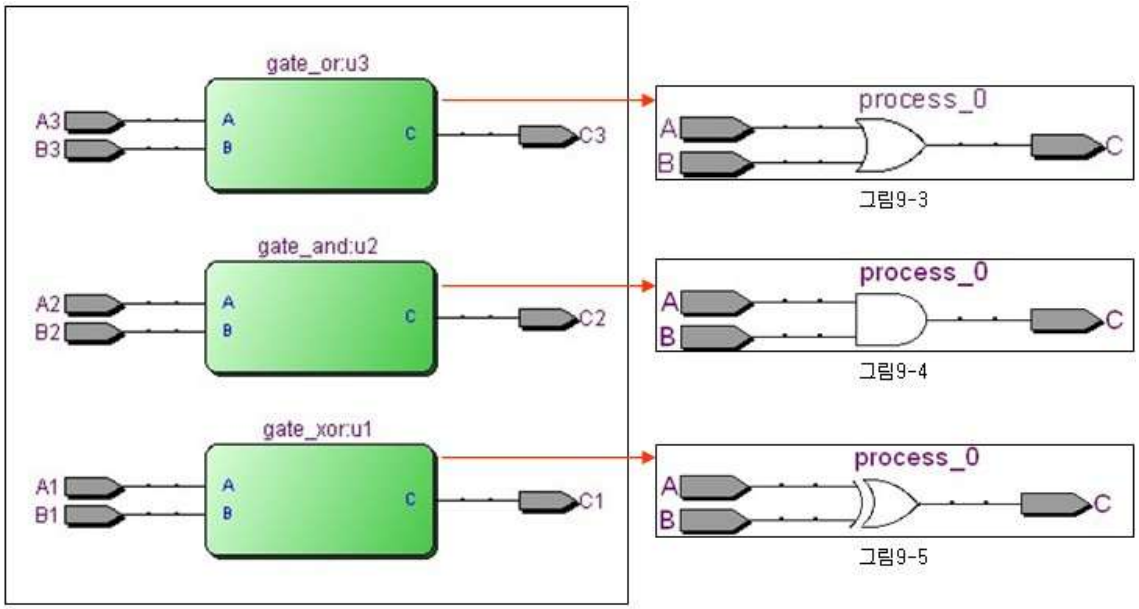


그림9-2

그림 9-2를 보면 full_gate를 합성했을 때 component문이 어떤 형태로 나타나는지를 보여준다. 하나의 component는 하나의 블록으로 보여준다.
그리고, 그 각각의 블록을 하위레벨로 더 들어가면 오른쪽과 같이 각각의 구문에 맞는 게이트나 플립플롭형태로 나오게 된다.

component구문까지 해서 VHDL에 대한 설명을 마치겠다. 몇가지 부분이 더 있기는 하지만 실제로 VHDL 구문을 만드는데 있어서 좀더 효율적이냐 아니냐의 문제지 나머지 부분을 모른다고 해서 구현을 못하고 하고의 문제는 아니다.
여기에 있는 내용은 비교적 쉽게 설명을 했고, 가장 기본적인것이기 때문에 여기에 있는 내용을 토대로 VHDL이라는거에 대해서 공부를 조금씩 하고, 다른 책들을 참고하다보면 자신도 모르게 실력이 쌓여갈거라 생각한다.