VHDL 기초

참고 자료

VHDL을 이용한 ASIC 설계 - IDEC

VHDL을 이용한 Microprocessor 설계 - IDEC

I8051 프로세서 설계 - IDEC

Altera MAX+PLUS II를 사용한 디지털 시스템 설계 - 북두 출판사

디지털 시스템 설계를 위한 VHDL - IDEC

하드웨어 엔지니어를 위한 VHDL - 그린 출판사

MAX+PLUS II Manual

Xilinx Foundation Manual

디지털 시스템 설계 - Mano

인터넷에서 다운 받은 문서 - 600Mega 등

목 차

VHDL의 기초적인 개요

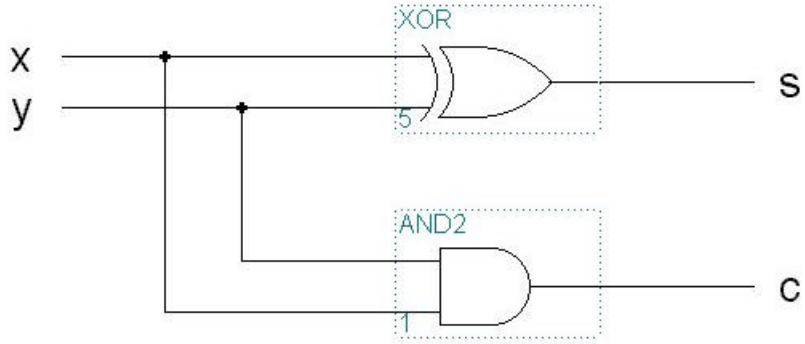
VHDL이 무엇이냐?

조금 방대한 질문이네요. 일단 예제로 시작합시다. 이것을 이해하기 위해서는 Background가 어느 정도 필요합니다. 전자과 3년 과정을 마쳤다면 충분히 이해할 수 있다고 생각되네요. 참고로 제가 사용하는 Tool은 Max Plus Ⅱ입니다.

그럼 2비트 반가산기(Adder)를 설계합시다.

$$S = x \oplus y$$
$$C = xy$$

두 개의 입력 x, y를 입력받아 Sum, Carry를 출력합니다.
이것을 먼저 Schematic으로 구현한다고 생각합시다.



이런 식으로 설계하면 되겠지요.

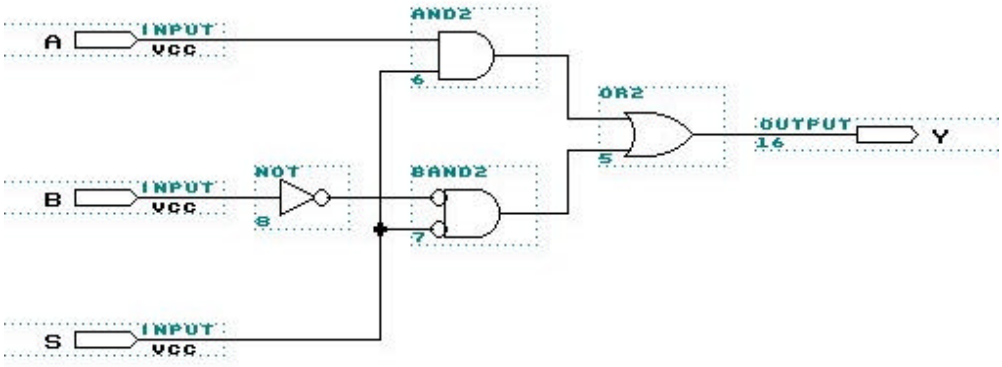
그럼 이것을 VHDL로 구현해 볼까요.

```
-- adder2.vhd

library ieee;
use ieee. std_logic_1164.all;

entity adder2 is
port ( x, y : in std_logic;
       s, c  : out std_logic);

architecture rtl of 2adder is
begin

s <= x xor y;
c <= x and y;
end rtl;
```

조금 Language와 비슷하지요. VHDL을 회로 설계하는 언어라고 생각하면 됩니다.

다른 2×1 MUX 예제를 봅시다.
이것을 한 번 구현해 봅시다.

이것을 VHDL로 구현해 봅시다.




```
-- mux21.vhd

library ieee;
use ieee. std_logic_1164.all;

entity mux21 is
port ( a, b, s : in std_logic;
           y      : out std_logic);
end mux21;

architecture rtl of mux21 is
begin
    process(a, b, s)
    begin
        if ( s = '0') then
            y <= a;
        else
            y <= b;
        end if;
    end process;
end rtl;
```

구현하는 것이 별로 어렵지는 않을 것입니다. 초보자에게는 어렵겠지만.
VHDL의 장점을 느끼지 못할 것입니다. 예제를 조금 변형해 봅시다.
입력 a, b가 4비트의 입력이라면 어떻습니까? K-map을 이용해 열심히 그려야지요.
과연 VHDL로는 어떻게 표현할 수 있을까요?

입력과 출력의 비트 수만 변형시키면 됩니다.

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

entity MUX21 is
port ( a, b     : in std_logic_vector(3 downto 0);
       s        : in std_logic;
       y        : out std_logic_vector(3 downto 0));
end MUX21;

architecture rtl of MUX21 is
begin
   process(a, b, s)
   begin
      if ( s = '0') then
         y <= a;
      else
         y <= b;
      end if;
   end process;
end rtl;
```

이것이 VHDL의 장점입니다. VHDL을 이렇게 보시면 됩니다. 회로 설계를 할 수 있는 표준 언어다. 모든 것은 장점과 단점을 가지고 있습니다. 단점에 비해 많은 장점을 가지고 있습니다. Schematic으로 구현할 수 있는 회로는 약 5만 게이트 정도입니다. 손으로 직접 그리는 것보다는 VHDL을 이용해 시간과 비용을 줄이는 것이 좋겠지요.

외국 검색 사이트(야후 또는 알타비스타 등등)를 이용해 VHDL을 입력해서 한번 찾아보세요. 얼마나 많은 사이트가 존재하는지. 국내 사이트도 많이 존재하고 있습니다.

새로운 것을 배운다는 것은 힘들지만, 시간이 있는 전자가 3, 4학년에게는 적극 권하고 싶네요. 현실적으로도 펜티엄칩도 이 VHDL을 이용해 설계하고 있습니다. 고급의 반도체 설계나 회로 설계를 원한다면 제 강의가 아니라도 다른 책을 이용해 한번 시도해 보시기를 바랍니다.

VHDL의 출현 배경과 VHDL의 구조 VHDL, ASIC, FPGA 용어 정리

“ VHDL의 출현 배경 및 변화 과정

미국 정부의 VHSIC Program의 주요 목적은 설계, 공정 및 제조 기술 분야에 있어서 미국의 기술 수준을 향상시키는 데 있었다. 또한 미국 정부는 이 Program의 일부로써 VHDL의 개발 노력을 지원하고 있었다. 그 지원 목적은 VHDL의 개발로 좀더 빠른 생산과 회사들간의 연락 기능 강화 및 개발 과정을 능률적으로 처리함으로써 비용 절감 효과를 제공하는 것이었다. 이러한 목적들을 효과적으로 달성할 수 있는 방법을 논의하기 위하여 메사추세츠의 Woods Hole에서 개최된 학술 대회를 시발점으로 1981년부터 VHDL이 개발되기 시작하였다. Technology independent(기술 독립적)하며 표준 하드웨어 기술 언어의 개발을 목표로 한 Woods Hole 학술 대회에서 정부의 공식적인 제안 요구서를 위한 초안이 제출되었으며, 이것은 전자 공학 분야에 종사하는 전문가들에 의해서 검토되는 과정을 거쳐 1983년 초에 수정, 확정되었다. 1980년대 중반 이후부터 VHDL이 문서용이 아닌 Simulation용으로 검증되어져야 한다는 여론이 강해지면서 몇몇 VHDL Simulator가 등장하였으나 업체간에 표준화가 이루어지지 않은 관계로 호환성에 문제가 있었다. 이러한 문제를 해결하고 늘어가는 VHDL관련 CAD Tool 회사간의 표준 및 호환성을 위하여 IEEE에서 1987년에 IEEE-1076이라는 표준을 만들어 공포하였다. 이 시점에서 Synthesis(회로 합성)는 아직 등장하지 않았으며 VHDL은 Simulation용으로 사용되었다. 1990년대에 들어서면서 VHDL 관련 Software 회사가 많이 등장하고 simulation뿐만 아니라 Synthesis의 기능을 갖춘 CAD Tool이 등장하면서 진정한 VHDL의 표준화가 요구되었고 1991년 IEEE-1164가 발표되면서 업체에서 공통으로 사용할 수 있는 VHDL이 탄생하였다.

1. VHDL 이란 ?

VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language) 은 상위의 동작 레벨에서부터 하위의 게이트 레벨까지 하드웨어를 기술하고 설계하도록 하는 CAD업계 및 IEEE 표준 언어이며 미국 정부가 지원을 공인한 하드웨어 설계 언어이다.

VHDL의 등장은 갈수록 복잡해지고 고집적화 되는 회로에 비례하여 어려워지는 하드웨어 설계 환경에 새로운 장을 여는 계기가 되었다. 컴퓨터 기술의 발달과 함께 VHDL을 이용한 설계 기법의 발달은 비단 이 분야에 전공하는 사람뿐만 아니라 초보자에게도 보다 쉽게 회로를 설계할 수 있는 기회를 제공하고 있다.

현재 VHDL은 세계적으로 사용되고 있으며 산업체, 대학, 연구소 등에서 그 관심이 급격히 증가하고 있다.

2. ASIC 이란

많은 사람들이 알고 있는 말 중에 하나인 ASIC이란 Application Specific IC의 약자로 우리말로 옮기게 되면 특정용도 주문형 반도체이다. 이 ASIC이란 광범위하게는 특정용도 목적으로 사용되는 모든 반도체를 가리키기도 하지만 좁게는 특히 Gate Array를 가리키는 말이다. 우리가 흔히 ASIC이라 부르며 알고 있는 상식은 Gate Array, Embedded Array, Standard Cell (또는 Cell Based IC) 세 종류를 가리키는 말로 알고 있다.

ASIC - Programmable IC Type : PAL, SOLD, COLD, PGA
- Memory IC Type: MICOM, ASIC Memory, FIFO
- Logic Based IC Type: Gate Array, Embedded Array, Standard Cell, Full-Custom IC.

하지만 ASIC이란 반도체 회사에 제품 의뢰를 하는 Gate Array 종류만을 가리키는 것이 아닌 특정 용도나 주문형 IC모두를 다 포함하고 있다. 위에서 보는 바와 같이 ASIC의 두 가지 동류로서의 Full Custom(완전 설계 방식)과 Semi Custom(반주문형 설계 방식)이라는 말은 일본에서 인위적으로 만들어 낸 것이다.

물론 ASIC에 대하여 정확하게 정의된 것이 없는 관계로 어쩌면 Gate Array 종류를 진짜 ASIC으로 부는 것도 옳다고 할 수도 있다. 어쨌거나 일반적으로 반도체 설계를 할 수 있는 방식은 일반인들이 ASIC이라 부르고 있는 Logic Based IC Type형태의 Gate Array와 Standard Cell, 그리고 Programmable IC Type형태의 EPLD, FPGA가 대표적이다.

3. FPGA

FPGA란 Field Programmable Gate Array의 준말로 Array Based와 Row Based 두 가지 방법이 있으며 구조는 Gate Array와 매우 흡사하지만 Program에 의해 내부 회로 배선이 연결되는 형식을 취하고 있다.

FPGA는 Logic Cell 위주의 설계 방식이기 때문에 SPLD Block 내부의 배선이 외부와 직접 연결될 수 있도록 고안되어 있어 일반 Gate Array와 매우 비슷하며 Timing Simulation이 반드시 필요하다.

다른 Programmable Device에 비해 속도가 월등히 뛰어나고 집적도가 좋으며 부품 단가도 훨씬 저렴하지만 이 종류는 단 한번밖에 구울 수 없기 때문이 주로 연구 개발용보다는 제품 생산용으로 많이 사용된다.

FPGA는 대개 2,000-20,000 Gates 급의 회로에 적당하며 JEDEC을 이용하여 굽도록 되어 있는 일반 PLD 종류에 비해 Programming 하는 File이 제품을 만든 회사의 고유 Programming Netlist인 ADL, QDF, XNF 등을 사용하도록 되어 있다. Array Based 형식의 FPGA는 SPLD Block들을 2차원 배열 형식으로 늘어뜨린 다음 중간에 Interconnect Channel이 서로 교차하며 연결될 수 있도록 되어 있다. 이 Array Based형식의 FPGA는 그림에서 보는 바와 같이 두 가지 종류가 있다. 대체로 MUX와 AND로 이루어진 Combinational Logic과 하나의 Flip-Flop으로 구성되어 있는 형식의 SPLD Block구조가 일반적이며 MUX들을 일렬로 배열한 다음 중간 중간에 Flip-Flip을 끼워 넣는 형식의 Logic Array형 제품도 있다. 일반적인 내부 구조는 Xilinx의 CLB, Cypress와 QuickLogic의 Logic Cell 등이 대표적인 이 형식을 취하고 있으며 Logic Array형의 내부 구조는 Altra의 LAB가 이러한 형식을 취하고 있다.

반면 Row Based FPGA의 내부 구조는 MUX 구조 형식의 Combinational Logic이 각 행마다 나열되어 있고 중간 중간에 Flip-Flop이 끼워져 있는 형식을 취하고 있다.

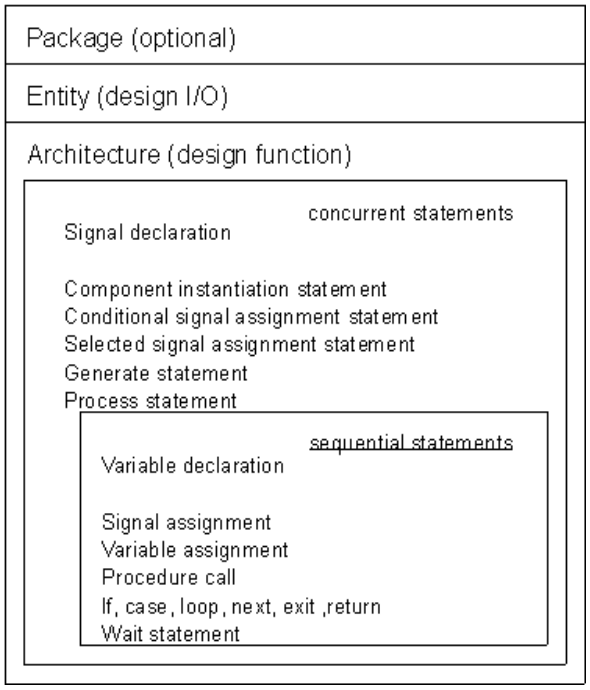
Array Based FPGA의 SPLD Block은 Logic Array 구조 형식이 가로가 아닌 세로로 된 것과 비슷하다.

다만 Array Base FPGA와의 다른 점이라면 Logic Cell끼리 연결할 수 있는 Interconnect가 나열된 Cell의 아래나 위에만 위치하여야 된다는 것이다. Actel이 이 Row Based FPGA의 대표적인 구조이다.

-- 어느 인터넷 사이트에서(어딘지는 기억이 나지 않네요.)

많은 용어들이 나오죠. 나중에 용어별로 정리해 올리겠습니다. 그리고 저에게 툴 사용법을 질문하지는 마세요. 그리고 질문은 저에게보다는 모든 사람이 볼 수 있는 곳에 질문하세요. 그리고 저는 VHDL만 알고 있습니다. 혹시 verilog을 아시는 분은 verilog로 소스를 좀 올려 주시면 감사하겠습니다. verilog 역시 하드웨어 표준 언어입니다.

제일 먼저 살펴보아야 할 것은 VHDL의 구조입니다. VHDL이 어떤 구조로 되어 있는지 밑에 있는 그림을 보면서 이해합시다.



영어로 되어 있으니 조금 복잡해 보이죠. 그러나 한정된 형식을 가지고 있으므로 그렇게 어렵지는 않습니다. 조금 하다 보면 자연스럽게 이해가 될 것입니다. 처음부터 모든 것을 다 이해하려고 하지 마세요.

Configuration
  Package
  Package Body
  Entity
Architecture

간단하게 표현하면 위의 5가지로 구성되어 있습니다.
지금 간단하게만 설명하고 될 수 있는 데로 예제를 중심으로 설명하는 방식을 사용하겠습니다.

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

entity mux21 is
port ( a, b     : in std_logic_vector(3 downto 0);
          s     : in std_logic;
          y     : out std_logic_vector(3 downto 0));
end mux21;
```

```
architecture rtl of mux21 is
begin
    process(a, b, s)
    begin
        if ( s = '0') then
            y <= a;
        else
            y <= b;
        end if;
    end process;
end rtl;
```

지난 시간 설명한 MUX21의 예제다.
여기서 진하게 마크된 것은 Reserved word(예약어)이다. 이는 VHDL 구문을 사용하면서 반드시 지켜야 할 사항이다.

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

여기가 Package 부분이다. VHDL에서는 사용할 라이브러리는 LIBRARY 키워드를 이용하여 지정해 주는데 다음과 같다.

LIBRARY ieee;

실제로 이 라이브러리가 존재하는 디렉토리는 시뮬레이션 또는 합성툴의 환경 변수로 지정되어 있게 된다. Compass Tool의 경우 Manager 윈도우를 띄우면 라이브러리 패스 (library path)를 지정하는 것을 볼 수 있을 것이다. MaxPlus 에도 User Library를 지정하는 메뉴가 있고 V-System/VHDL 에도 역시 라이브러리 패스를 지정하는 메뉴가 있습니다. 대개 라이브러리 이름과 디렉토리 패스를 연결시키도록 되어 있는데, 라이브러리 명(library name)과 패스(directory path)를 맵핑(mapping) 시킨다고 한다.

USE ieee. std_logic_1164.ALL;

이는 ieee라는 라이브러리에서 std_logic_1164 라는 이름의 패키지(package)를 가져다가 그 안에 있는 함수(function)들을 모두 사용한다는 뜻으로 ALL 이라고 한 것이다. ALL은 VHDL 의 키워드이다. 물론 ALL 대신 패키지 내에서 사용할 함수를 지정할 수도 있다.

사용툴의 Library 디렉토리에 가면 ieee. vhd 라는 파일이 있다. 이것을 직접 확인해 보는 것도 좋은 방법이다. 각각의 툴마다 조금씩 차이가 나는 것을 알 수 있을 것이다.

```
entity mux21 is
port ( a, b     : in std_logic_vector(3 downto 0);
       s        : in std_logic;
       y        : out std_logic_vector(3 downto 0));
end mux21;
```

시스템을 구성하는 부분품으로서 이들 사이의 상호 연결을 위한 통로 역할(interface)을 하는 것이 Entity이다. 말하자면 내부 설계에 대한 입출력 등을 기술하는 "포장"인 셈이다. 시스템의 입장에서 보면 내부 설계에 대한 것은 가려져 있고 다만 ENTITY에 기술된 입출력 포트만을 보게 되는데 이런 의미에서 ENTITY를 "암흑 상자"(Black Box)라고 하기도 한다.

위의 예제에서 보면 Entity name은 MUX21이고 각종 포트에 대한 설명을 한다. 몇몇의 툴에서는 Entity name과 파일의 이름이 일치해야 하는 경우도 있으니 참조하기 바란다(Max Plus Ⅱ의 경우). 입력과 출력에 대한 포트의 설명이다.

포트 name은 Reserved word와 일치해서는 안 된다.(중요)

in, out 이라고 한 것이 입력과 출력의 표시이다. **in**은 입력을 나타내고 **out**은 출력을 나타낸다. 그 외에도 **buffer** 모드와 **inout** 모드가 있다. 다음에 예제를 통해 설명하겠지만 지금 간단히 설명하면 다음과 같다. **INOUT**은 입출력 포트로 사용되므로 소스(source) 또는 목적지(destination)에 모두 쓸 수 있다. 그러나 한 개의 문장에 모두 쓸 수 없다. 입출력 모드가 정해진 경우 할당문(assign statement)에서의 엄격한 규정은 하드웨어를 다루기 때문이다. 하드웨어를 기술할 때 assign 이란 연결 관계를 표현한 것(netlist)으로서 프로그래밍 언어에서와 같은 오퍼레이션(move operation)이 아니다. **BUFFER**는 INOUT과 같은 입출력 포트로서 assign의 소스(source)측 또는 목적지(destination)측에 쓸 수 있다. INOUT 모드와 다른 점은 단일 할당문 내에서 소스와 목적지측 모두 동시에 사용할 수 있다는 것이다. 이는 BUFFER 모드에 이미 F/F를 내포하고 있다는 의미를 갖는다.

1비트의 경우는 std_logic으로 표현하고, 다중 비트의 경우는 std_logic_vector(n downto 0)라고 표현한다.

```
std_logic_vector(0 to 3)
std_logic_vector(3 downto 0)
```

위의 두 가지 모두 MSB가 왼쪽에 있는 4비트를 표현한 것이다.

std_logic 이외에도 객체형(Object Types)이 있지만 뒤에 설명하는 것이 좋겠다.

한가지 양해를 구할 것이 있다. 설계를 하다 보면 그 정식 용어를 써야 할 때가 많다. 그래서 되도록 원문을 사용하는 것이 좋다고 판단된다. 그래서 영어가 나와도 그대로 받아들이기를 원한다. 현 추세의 하드웨어 설계는 혼자서 하는 것이 아니다. 아직 Configure에 대한 설명은 하지 않았지만 Configure의 역할은 여러 사람의 설계를 조합할 때 사용하는 것이다. Pentinum 설계에 투입된 인원이 600명 이상이라고 한다. 그만큼 현 추세는 복잡하고 세분화되어 있다. 뒷부분에서 Configure의 자세한 부분들 다루기로 하겠다. 지금은 예제가 간단하므로 Configure의 설정이 필요하지 않다.

```
architecture rtl of mux21 is
begin
    process(a, b, s)
    begin
        if ( s = '0') then
            y <= a;
        else
            y <= b;
        end if;
    end process;
end rtl;
```

다음은 architecture 부분이다.
아직 버그가 잡혔는지는 모르겠지만, Lodecap에서는 architecture name과 entity name이 같아야 하는 이상한 버그가 있었다. (지금은 해결되었을 것이라 판단된다.) architecture name과 entity name을 다르게 잡아 주는 것이 좋다고 생각된다. architecture name과 entity name을 정할 때 그 파일이 어떤 파일인지 자기가 구별할 수 있도록 하는 습관은 좋은 습관이다. 모든 architecture에서는 Concurrent(동시에 수행)하게 동작된다. 모든 하드웨어 설계의 기본의 동작이 Concurrent하게 발생된다. Sequential(순차적으로)하게 표현하기 위해서 Process 문을 사용한다. Process 문안에서의 모든 동작은 순차적으로 발생된다. **하나의 Architecture 안에 다수의 Process 문을 사용할 수 있다. 그리고 적어도 하나 이상의 Entity와 Architecture 문을 구성할 수 있다.**
다음에는 많은 예제를 가지고 VHDL의 표현에 대해서 예제를 중심으로 설명하겠습니다.
많은 예제를 가지고 그것을 시뮬레이션 하다 보면 자연스럽게 VHDL의 문법을 익힐 수 있을 것입니다.

VHDL에 사용할 문법(Altera VHDL Template)

1. Overall Structure

```
-- Context Clauses
        -- Library Clause
        -- Use Clause
--      Library Units
        -- Package Declaration (optional)
        -- Package Body (optional)
        -- Entity Declaration
        -- Architecture Body
```

2. Full Design : Counter

```
-- MAX+plus II VHDL Template
-- Clearable loadable enablable counter

ENTITY __entity_name IS
        PORT
        (
                __data_input_name          : IN     INTEGER RANGE 0 TO __count_value;
                __clk_input_name           : IN     STD_LOGIC;
                __clrn_input_name          : IN     STD_LOGIC;
                __ena_input_name           : IN     STD_LOGIC;
                __ld_input_name            : IN     STD_LOGIC;
                __count_output_name        : OUT    INTEGER RANGE 0 TO __count_value
        );
END __entity_name;

ARCHITECTURE a OF __entity_name IS
        SIGNAL  __count_signal_name        : INTEGER RANGE 0 TO __count_value;
BEGIN
        PROCESS (__clk_input_name, __clrn_input_name)
        BEGIN
                IF __clrn_input_name = '0' THEN
                        __count_signal_name <= 0;
                ELSIF (__clk_input_name' EVENT AND __clk_input_name = '1') THEN
                        IF __ld_input_name = '1' THEN
                                __count_signal_name <= __data_input_name;
                        ELSE
                                IF __ena_input_name = '1' THEN
                                        ount_signal_name <= __count_signal_name + 1;
                                ELSE
                                        __count_signal_name <= __count_signal_name;
                                END IF;
                        END IF;
                END IF;
        END PROCESS;
        __count_output_name <= __count_signal_name;
END a;
```

3. Full Design : Flipflop

```
-- MAX+plus II VHDL Template
-- Clearable flipflop with enable

LIBRARY ieee;
USE ieee. std_logic_1164.all;
ENTITY __entity_name IS
        PORT
        (
                __d_input_name              : IN    STD_LOGIC;
                __clk_input_name   : IN     STD_LOGIC;
                __clrn_input_name  : IN     STD_LOGIC;
                __ena_input_name   : IN     STD_LOGIC;
                __q_output_name             : OUT   STD_LOGIC
        );
END __entity_name;

ARCHITECTURE a OF __entity_name IS
        SIGNAL __q_signal_name    : STD_LOGIC;
BEGIN
        PROCESS (__clk_input_name, __clrn_input_name)
        BEGIN
                IF __clrn_input_name = '0' THEN
                        __q_signal_name <= '0';
                ELSIF (__clk_input_name' EVENT AND __clk_input_name = '1') THEN
                        IF __ena_input_name = '1' THEN
                                __q_signal_name <= __d_input_name;
                        ELSE
                                __q_signal_name <= __q_signal_name;
                        END IF;
                END IF;
        END PROCESS;
        __q_output_name <= __q_signal_name;
END a;
```

4. Full Design : Tri-State Buffer

```
-- MAX+plus II VHDL Template
-- Tri-State Buffer

LIBRARY ieee;
USE ieee. std_logic_1164.all;

ENTITY __entity_name IS
        PORT
        (
                __oe_input_name                 : IN    STD_LOGIC;
                __data_input_name       : IN    STD_LOGIC;
                __tri_output_name       : OUT   STD_LOGIC
        );
END __entity_name;

ARCHITECTURE a OF __entity_name IS
BEGIN
        PROCESS (__oe_input_name, __data_input_name)
        BEGIN
                IF __oe_input_name = '0' THEN
                        __tri_output_name <= 'Z';
                ELSE
                        __tri_output_name <= __data_input_name;
                END IF;
        END PROCESS;
END a;
```

5. Architecture Body

```
ARCHITECTURE a OF __entity_name IS
        SIGNAL __signal_name : STD_LOGIC;
        SIGNAL __signal_name : STD_LOGIC;
BEGIN
        -- Process Statement
        -- Concurrent Procedure Call
        -- Concurrent Signal Assignment
        -- Conditional Signal Assignment
        -- Selected Signal Assignment
        -- Component Instantiation Statement
        -- Generate Statement
END a;
```

6. Case Statement

```
CASE __expression IS
        WHEN __constant_value =>
            __statement;
            __statement;
        WHEN __constant_value =>
            __statement;
            __statement;
        WHEN OTHERS =>
            __statement;
            __statement;
END CASE;
```

7. Component Declaration

```
COMPONENT __component_name
        PORT(
                __input_name, __input_name          : IN     STD_LOGIC;
                __bidir_name, __bidir_name          : INOUT  STD_LOGIC;
                __output_name, __output_name        : OUT    STD_LOGIC);
END COMPONENT;
```

8. Component Instantiation Statement

```
__instance_name: __component_name
        PORT MAP (      __formal_parameter => __actual_parameter,
                                __formal_parameter => __actual_parameter);
```

9. Concurrent Procedure Call

```
__label: __procedure_name(__actual_parameter, __actual_parameter);
```

10. Concurrent Signal Assignment Statement

```
__signal <= __expression;
```

11. Conditional Signal Assignment

```
__label:
__signal <= __expression WHEN __boolean_expression ELSE
                        __expression WHEN __boolean_expression ELSE
                        __expression;
```

12. Constant Declaration

```
CONSTANT __constant_name : __type_name := __constant_value;
```

13. Entity Declaration

```
ENTITY __entity_name IS
        PORT(
          __input_name, __input_name            : IN          STD_LOGIC;
                __input_vector_name         : IN   STD_LOGIC_VECTOR(__high downto __low);
                __bidir_name, __bidir_name      : INOUT       STD_LOGIC;
                __output_name, __output_name    : OUT         STD_LOGIC);
END __entity_name;
```

14. For Statement

```
__loop_label:
FOR __index_variable IN __range LOOP
    __statement;
    __statement;
END LOOP __loop_label;
```

15. Generate Statement (For Statement)

```
__generate_label:
FOR __index_variable IN __range GENERATE
    __statement;
    __statement;
END GENERATE;
```

16. Generate Statement (If Statement)

```
__generate_label:
IF __expression GENERATE
    __statement;
    __statement;
END GENERATE;
```

17. If Statement

```
IF __expression THEN
    __statement;
    __statement;
ELSIF __expression THEN
    __statement;
    __statement;
ELSE
    __statement;
    __statement;
END IF;
```

18. Library Clause

```
LIBRARY __library_name;
```

19. Package Declaration

```
PACKAGE __package_name IS
        -- Type Declaration
        -- Subtype Declaration
        -- Constant Declaration
        -- Signal Declaration
        -- Component Declaration
END __package_name;
```

20. Procedure Call Statement

```
__procedure_name(__actual_parameter, __actual_parameter);
```

21. Process (Combinatorial Logic)

```
__process_label:
PROCESS (__signal_name, __signal_name, __signal_name)
        VARIABLE __variable_name : STD_LOGIC;
        VARIABLE __variable_name : STD_LOGIC;
BEGIN
        -- Signal Assignment Statement
        -- Variable Assignment Statement
        -- Procedure Call Statement
        -- If Statement
        -- Case Statement
        -- Loop Statement
END PROCESS __process_label;
```

22. Process (Sequential Logic)

```
__process_label:
PROCESS
        VARIABLE __variable_name : STD_LOGIC;
        VARIABLE __variable_name : STD_LOGIC;
BEGIN
        WAIT UNTIL __clk_signal = '1';
        -- Signal Assignment Statement

        -- Variable Assignment Statement

        -- Procedure Call Statement

        -- If Statement

        -- Case Statement

        -- Loop Statement

END PROCESS __process_label;
```

23. Selected Signal Assignment Statement

```
__label:
WITH __expression SELECT
        __signal <= __expression WHEN __constant_value,
                                __expression WHEN __constant_value,
                                __expression WHEN __constant_value,
                                __expression WHEN __constant_value;
```

24. Signal Declaration

```
SIGNAL __signal_name : __type_name;
```

25. Signal Assignment Statement

```
__signal_name <= __expression;
```

26. State Machine with Async. Reset

```
ENTITY __machine_name IS
        PORT(
                clk                                     : IN    STD_LOGIC;
                reset                                   : IN    STD_LOGIC;
                __input_name, __input_name              : IN    STD_LOGIC;
                __output_name, __output_name            : OUT   STD_LOGIC);
END __machine_name;

ARCHITECTURE a OF __machine_name IS
        TYPE STATE_TYPE IS (__state_name, __state_name, __state_name);
        SIGNAL state: STATE_TYPE;
BEGIN
        PROCESS (clk)
        BEGIN
                IF reset = '1' THEN
                        state <= _state_name;
                ELSIF clk' EVENT AND clk = '1' THEN
                        CASE state IS
                                WHEN __state_name =>
                                        IF __condition THEN
                                                state <= __state_name;
                                        END IF;
                                WHEN __state_name =>
                                        IF __condition THEN
                                                state <= __state_name;
                                        END IF;
                                WHEN __state_name =>
                                        IF __condition THEN
                                                state <= __state_name;
                                        END IF;
                        END CASE;
                END IF;
        END PROCESS;

        WITH state SELECT
                __output_name   <=      __output_value  WHEN    __state_name,
                                        __output_value  WHEN    __state_name,
                                        __output_value  WHEN    __state_name;
END a;
```

27. State Machine without Async. Reset

```
ENTITY __machine_name IS
        PORT(
                clk                                     : IN    STD_LOGIC;
                __input_name, __input_name              : IN    STD_LOGIC;
                __output_name, __output_name            : OUT   STD_LOGIC);
END __machine_name;

ARCHITECTURE a OF __machine_name IS
        TYPE STATE_TYPE IS (__state_name, __state_name, __state_name);
        SIGNAL state: STATE_TYPE;
BEGIN
        PROCESS (clk)
        BEGIN
                IF clk' EVENT AND clk = '1' THEN
                        CASE state IS
                                WHEN __state_name =>
                                        IF __condition THEN
                                                state <= __state_name;
                                        END IF;
                                WHEN __state_name =>
                                        IF __condition THEN
                                                state <= __state_name;
                                        END IF;
                                WHEN __state_name =>
                                        IF __condition THEN
                                                state <= __state_name;
                                        END IF;
                        END CASE;
                END IF;
        END PROCESS;

        WITH state SELECT
                __output_name   <=      __output_value  WHEN    __state_name,
                                        __output_value  WHEN    __state_name,
                                        __output_value  WHEN    __state_name;
END a;
```

28. Subtype

```
SUBTYPE __subtype_name IS __type_name RANGE __low_value TO __high_value;
SUBTYPE __array_subtype_name IS __array_type_name(__high_index DOWNTO __low_index);
```

29. Type

```
TYPE __enumerated_type_name IS (__name, __name, __name);
TYPE __range_type_name IS RANGE __integer TO __integer;
TYPE __array_type_name IS ARRAY (INTEGER RANGE <>) OF __type_name;
TYPE __array_type_name IS ARRAY (__integer DOWNTO __integer) OF __type_name;
```

30. Use Clause

```
USE __library_name.__package_name. ALL;
```

31. Wait Statement

```
WAIT UNTIL __clk_name = '1';
```

32. Variable Declaration Statement

```
VARIABLE __variable_name : __type_name;
```

33. Variable Assignment Statement

```
__variable_name := __expression;
```

34. While Statement

```
__loop_label:
WHILE __boolean_expression LOOP
    __statement;
    __statement;
END LOOP __loop_label;
```

위에 있는 것들은 VHDL에 사용할 문법들입니다. 이것은 Altera의 문법들입니다. VHDL이 하드웨어 표준 언어이지만 각 회사들마다 조금씩 문법의 차이는 있습니다. 실 예로 Lodecap에서 모든 컴파일과 시뮬레이션을 마친 상태에서 그 소스를 Altera에서 실행한 적이 있습니다. 그런데 상태 이름이 Next였습니다. 그런데 MaxPlusⅡ에서는 Next가 Reserved word이었습니다. 그래서 몇 일을 고생한 적이 있습니다. 자기가 사용하는 툴에 대해서는 어느 정도의 지식이 있어야 합니다.
그러니 각 회사 사이트에 접속하면 Manual을 구할 수 있습니다. 그리고 공개용 버전도 어느 정도 구할 수 있을 것입니다. 직접 접속해 보시는 것도 좋은 방법입니다.

ASIC 용어 요약

1 ASIC : Application Specific Integrated Circuit
주문형 반도체
시스템 업체가 자기 시스템의 특정 회로 부분을 하나의 반도체로 집적시켜 개발하여, 반도체 제조업자에게 주문 제조한 반도체 수요 업체가 주문 제조한 특정 회로용으로만 사용되기 때문에 기존의 범용 반도체(반도체 업체가 생산하는 표준화된 반도체:Standard IC)와 상대적인 개념으로 특정용도 IC(ASIC)라 통칭함.

2 ATVG : Automatic Test Vector Generation
일반적으로 결점 적용 범위(Fault Simulation)의 레벨을 증가시키고, 기능을 검사하기 위한 테스트 패턴 (Test Pattern)들을 증가시키기 위해 이용된다.

3 Back Annotation
레이아웃(Layout)후에 R. C값을 추출하는 작업

4 Behavioral Description
알고리즘 또는 수학적인 방정식의 항으로부터 소자 또는 기능을 모델화하는것.

5 Bottom-up Design
계층적 설계 방법 (Hierarchical Design)을 이용하여 트랜지스터나 게이트 같은 기본적인 소자로부터 셀(cell),모듈(module)등 중간 레벨의 구조를 만들고 정의하여, 높은 레벨의 시스템 구조를 꾸며 나가는 설계 방식(Top-Down Design과 상대적 개념)

2.6 Cell
특정한 전기적 기능을 수행하기 위해 이미 정의된 회로 소자의 레이아웃이나 파일

7. Cell Library
특성을 가진 셀 들의 모임으로 일반적 ASIC 벤다(Vendor)회사에 특정된다.

8. CIF : Caltech Intermediate Format
표준형 기계가 읽을 수 있도록 마스크 레벨인 도형적 레이아웃을 표현하기 위한 형식. 레이아웃의 표준으로 GDSII도 있다.

9. Core
I/O 패드 링 (Pad Ring)을 제외한 영역 또는 ASIC의 능동 영역

10. Critical Path
회로망에서 가장 긴 경로. 임계 경로 전달 지연은 소자의 최대 클락 주파수를 제한한다.
11. Design Rule
도형적 레이아웃을 이루는 다각형들에 대한 최소의 너비와 간격에 대한 요구 사항들을 정의한 것. 테크놀러지별로 Metal1,Metal2, Poly등의 값들을 정의.

12. DFT : Design For Testability
테스트 용이화 설계(DFT)는 회로 설계단계시, 논리회로의 테스트를 손쉽게 할 테스트 패턴 생성을 고려하여 설계하는 것이다.

13. DIE
칩이라고도 부름. 다이는 회로나 소자의 어레이를 포함하는 웨이퍼를 스크라이브선 (Scribe Line)을 따라 잘라서 얻은 하나의 집적된 회로이다.

14. DRC : Design Rule Checker
설계 아트워크가 어떤 특정 공정에서 아무런 문제없이 제작될 수 있는가를 검토하기 위하여, 완전한 레이아웃을 공간적인 면에서 공정 설계 규칙에 맞는지 검토하는 프로그램.

15. EDA : Electronic Design Automatic
컴퓨터를 이용한 회로 설계 자동화 엔지니어링 Tools에 상응함.

16. EDIF : Electronic Data Interchange Format
어떤 설계 툴에서 만들어진 데이터를 다른 설계 Tools로 전송하기 위한 표준화된 중간 서식.

17. ERC : Electronic Rule Checker
과다한 팬 아웃, 개방(Open), 단락(Short)과 같은 전기 법칙의 위반들에 대하여 회로 레이아웃을 검토하는 프로그램.

18. Fault
어떤 회로의 개방 또는 단락으로 인하여 기능적 고장을 초래하는 제조상의 결함.

19. Floating Node
연결되지 않은 채 남겨진 게이트의 입력 또는 출력으로 놓아두면 기능적 고장을 유발하게 된다. 부동 단자는 대개 논리적으로 High 상태로 부동한다. ERC 프로그램들이 이런 레이아웃 에러를 검색하는데, 이것들은 공정 중에 불안전한 접촉에 의해 발생하기도 한다.

20. Floor Planning
최적의 레이아웃을 얻기 위하여 칩 레이아웃 영역 내의 기능 블럭들을 배치하고, 그 기능 블럭들 사이를 연결하여 할당하는 과정.

21. Foundry
ASIC 설계 업체와 고객들이 자신들이 갖고 있는 공정에 적합한 Function Block들을 사용, 완성된 설계를 제조하기 위하여 이용할 수 있는 반도체 제조 설비

22. Gate
두개 이상의 입력들과 하나의 출력을 가진 회로로, 출력은 입력 신호의 논리 함수로 표시된다. 기본적인 논리 게이트 형태로는 AND, OR, NAND, NOR와 같은 불렌함수(Boolean Function)들이 있다.

23. GDSII
마스크를 만들기 위한 레이아웃을 생성시키기 위하여 사용되는 설계 데이터 형식의 하나로 다른 형식으로는 CIF가 있다.

24. Hierarchical Design
하나의 모듈을 여러 종속 모듈들로 나누어 설계하는 방식으로 ,하나의 논리를 구조적으로 표기하는 방법. 예를 들면 하나의 마이크로 프로세서 블럭은 게이트 블럭 도로 구성된 플립플롭 블럭을 포함하고 이 블럭들로 구성된 하나의 프로그램 카운터 블럭을 포함하고 있다.

25. Load
구동 소자의 출력에 존재하는 저항이나 커패시턴스.

26. Macro Cell / Hard Macro
코아 기능이라고도 하는데 , 매크로는 본래 표준형 카탈로그 부품으로 제공되는 어떤 기능을 수행하는 복잡한 ASIC 셀이다. 하드 매크로로 불리기도 한다. (레이아웃이 설계 규칙에 맞게 고정되어 있기 때문)

27. Maintenance
소프트웨어 회사의 툴에 대해 기술적 지원, 오류 정정 소프트웨어 개선, 서비스 등을 제공받기 위해 , 사용자가 소프트웨어 회사에 지불하는 비용.

28. MPW : Multi Project Wafer
여러 다른 설계자에 의해 설계된 여러 개의 다른 프로젝트들을 한 웨이퍼 상에 제조함으로써 NRE비용을 여러 설계자들에게 분담시키는 방법.

29. Netlist
임의의 설계 구성 셀들과 이들의 연결 상태에 대한 정보 나열.

30. Net Comparison or Netcompare
Schematic Capture에서 얻어진 네트리스트와 레이아웃의 네트리스트가 같은 기능과 연결성을 갖는지를 비교 검토하는 것.

31. Node
회로 요소 또는 한 회로망의 임의 가지의 단자.

32. NRE : Non Recurring Engineering
ASIC 시제품 제작을 위한 개발 노력 행위와 그에 연관된 비용에 관한 것.

33. Pad
I/O 회로를 패키지 또는 기판에 연결하기 위해 사용되는 칩 가장자리에 위치한 금속 영역.

34. P & R
레이아웃의 배치 및 배선

35. PGA : Pin Grid Array
어떤 그리드에 나열된 패키지의 리드(Lead)를 패키지 몸체 밑의 아래 방향으로 나오게 하는 Through Hole 장착 패키지 기술.

36. Physical Design
트랜지스터, 셀, 블럭과 그것들의 배치와 배선을 포함하는 기하학적 요소들의 항으로 집적회로 레이아웃을 도형적으로 구현하는 것.

37. Placement
칩 레이아웃 내의 세 또는 블럭들을 물리적으로 위치시키는 것.

38. Primitive
게이트와 같은 낮은 레벨의 기능.

39. Prototype
어떤 특정한 응용에 대해 첫 번째 설계 또는 첫 번째 동작 모델의 형태. 정확성과 기능을 평가하기 위한 시제품 혹은 시작품.

41. Routing
Cell들 사이의 연결 통로.

42. Scribe Line
인접한 다이의 위치를 분리시키는 웨이퍼 상의 영역. 스크라이브 영역 선은 각각의 칩을 산출하기 위해 줄쳐 있거나 잘려지는 선

43. Sea of Gate
배선용 채널을 따로 가지지 않으며 트랜지스터가 연속적으로 배열된 게이트 어레이 구조의 한 형태.

44. Silicon Compiler
고수준 설계 표기가 주어졌을 때 , 도형적 설계와 시뮬레이션을 포함하는 모든 필요한 설계 관점들을 컴파일 하거나 종합하는 설계 툴.

45. Simulation or Test Vector
회로의 입력에 적용되어 연산되었을 때 , 도형적 설계와 시뮬레이션을 포함하는 모든 필요한 설계 관점들을 컴파일 하거나 종합하는 설계 툴.

46. Standard Cell
고정된 물리적, 전기적 특성들에 의해 특정 지어지는 게이트 또는 레치 같은 기본적인 기능적인 요소.

47. Symbol
셀의 경계 박스와 I/O 포트를 그림으로 나타낸 것.

48. Synthesis
상태 천이 기계, 진리표, 진리도는 불린 방정식 등의 고수준 설계 서술을 특정한 게이트 레벨 논리 구현으로 변환하는 것.

49. VHDL : VHSIC Hardware Description Language
언어적 사양에 따라 기능적 등가 칩을 생산할 수 있는 많은 ASIC회사들에 의해 이상적으로 선정된 사양을 만드는데 사용되는 기술도립적인 표준형 설계 표기 언어. (IEEE 1076)

50. Yield
웨이퍼 상의 올바로 동작하는 칩수와 전체 칩수의 비율.

VHDL의 기초적인 예제와 MUX STD_LOGIC

이번 시간부터는 많은 예제를 통하여 기초적인 회로를 구현해 보도록 하겠습니다. 많은 예제를 통해 자기 자신만의 코딩 스타일을 가지는 것도 중요한 일이지요.

ex1)

```
-- 논리식
library ieee;
use ieee. std_logic_1164.all;

entity ex1 is
port ( a, b, c : in bit;
          z    : out bit);
end ex1;

architecture rtl of ex1 is
begin
   z <= (a and b) or c;
end rtl;
```

ex2)

```
-- 논리식
library ieee;
use ieee. std_logic_1164.all;

entity ex1 is
port ( a, b, c : in std_logic;
          z    : out std_logic);
end ex1;

architecture rtl of ex1 is
begin
   z <= (a and b) or c;
end rtl;
```

세미콜론(;)을 붙이는 것도 상당히 중요합니다. 여기서는 두 가지의 중요한 사항이 있습니다. 눈썰미가 있는 분들은 한가지는 찾으셨겠죠. C 언어나 다른 언어의 기초적인 사항만 알면 별로 어려운 문법은 없다고 생각됩니다.

먼저 우리가 할 일을 먼저 생각합시다. 컴파일을 통해 일단 버그(?)를 잡아야 하겠죠. VHDL의 문법은 상당히 엄격합니다. 그래서 그 규정을 지키지 않으면 제대로 테스트하기가 어렵습니다. 컴파일이 끝나면 시뮬레이션을 통해 그 결과를 확인해야 합니다. 시뮬레이션 시 중요한 것은 Delay가 어떻게 되는지 생각해야 합니다. 결과가 제대로 나왔다고 해서 그 결과를 만족해서는 안됩니다. **중요한 것은 Real-time Simulation을 해야 한다는 것입니다.** 초기부터 항상 이것을 명심하기를 바랍니다. 시뮬레이션이 끝나면 그 결과를 구현해 보는 것도 중요합니다. 보드가 없는 사람은 시뮬레이션으로 끝나야 하겠지만 여건이 가능하다면 항상 구현해 보는 것도 즐거운 일입니다.

이 두 예제의 차이점은 **bit**와 **std_logic**의 차이입니다. bit는 전기적인 신호 '1'과 '0'을 말합니다. 전자 회로의 기초적인 사항이죠.

std_logic 이란 무엇인가요? 이것을 이해하기 전에 이것을 시뮬레이션으로 구현해 보셨나요? 결과가 어떻습니까? 같은 결과를 보여주죠.

디지털 전자 회로는 어떻습니까? '1'과 '0'으로 모든 것을 다 표현할 수 있었습니까? 그렇지는 않죠. K-map을 배웠다면 **Don't care**도 배웠을 것입니다. 그리고 결과를 표시할 수 없는 **Undefined**도 배웠을 것입니다. 그러니 '1'과 '0'이 모든 것을 표현할 수는 없습니다. 그래서 IEEE에서는 표준화되어서 나온 것이 **Std_logic**입니다. **Std_logic**은 nine values를 가지고 있습니다.

```
'U',  -- Uninitialized
'X',  -- Forcing  Unknown
'0',  -- Forcing  0
'1',  -- Forcing  1
'Z',  -- High Impedance
'W',  -- Weak     Unknown
'L',  -- Weak     0
'H',  -- Weak     1
'-'   -- Don't care
```

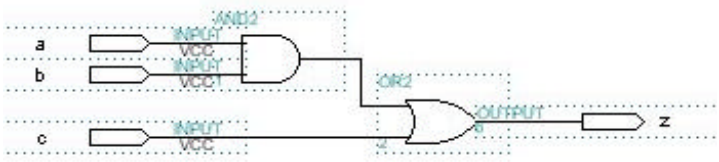
디지털 전자 회로를 표현하기 위해서는 이것이 정의되어야 합니다. 그리고 VHDL 논리 연산으로는 AND, NAND, OR, NOR, XOR, XNOR, NOT가 있습니다. 위의 예제에서 쓰인 것처럼 연산을 직접 하면 됩니다. 예제 처음에 '--'이라는 표시가 있지요. 이것은 C에서처럼 주석을 표시합니다. 한 문장의 끝까지 주석으로 처리합니다. 주석 입력이 한글로 되지 않는 툴이 많습니다. 심지어 한글로 된 디렉토리까지 읽지 못하는 경우도 있으니 되도록 디렉토리나 파일명을 영어로 입력하기를 바랍니다.

다음은 Assign에 대해서 설명하겠습니다.

```
z <= (a and b) or c;
```

입력포트로서 signal assign의 소스(source)측 (Signal일 경우 '<=', Variable일 경우 ':='의 오른쪽, R-Value)에만 쓸 수 있습니다. OUT은 출력포트로서 signal assign의 목적지(destination) ('<=' 또는 ':='의 왼쪽, L-Value)에만 쓸 수 있습니다. 위 예제에서 입력은 a, b, c로 정해져 있고 출력은 z로 정해져 있습니다. 아니 우리가 그렇게 입출력을 정했죠. 지금과 같이 입출력 모드가 정해진 경우 할당문(assign statement)에서는 엄격한 규정으로 하드웨어를 다룹니다. 하드웨어를 기술할 때 assign이란 연결 관계를 표현한 것(netlist)으로서 프로그래밍 언어에서와 같은 오퍼레이션(move operation)과는 다른 의미입니다.

두 예제를 분석하는 데에도 상당히 장난이 아니죠.



위 그림처럼 두 개의 게이트와 3개의 입력 패드 와 한 개의 출력 패드만 있으면 되는데 말이죠.

다음 예제 3가지 모두 같은 기능을 가지는 4×1 Mux입니다. 예제 3과 4는 Concurrent하게 설계되었고 나머지 하나는 Process 문안에서 Sequential하게 설계되었습니다.

ex3)

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

entity mux41 is
port ( a, b, c, d : in std_logic;
       sel        : in std_logic(1 downto 0);
       y          : out std_logic);
end mux41;

architecture rtl of mux41 is
begin
   y <= a when ( sel = "00" ) else
        b when ( sel = "01" ) else
        c when ( sel = "10" ) else
        d when ( sel = "11");
end rtl;
```

ex4)

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

entity mux41 is
port ( a, b, c, d : in std_logic;
         sel      : in std_logic(1 downto 0);
          y       : out std_logic);
end mux41;

architecture rtl of mux41 is
begin
   with sel select
    y <= a when "00",
         b when "01",
         c when "10",
         d when "11";
end rtl;
```

ex5)

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

entity mux41 is
port ( a, b, c, d : in std_logic;
          sel     : in std_logic_vector(1 downto 0);
           y      : out std_logic);
end mux41;

architecture rtl of mux41 is
begin
   process(sel, a, b, c, d)
   begin
      case sel is
         when "00" => y <= a;
         when "01" => y <= b;
         when "10" => y <= c;
         when others => y <= d;
     end case;
   end process;
end rtl;
```

예제 3, 4, 5 모두가 1-way 4 ×1 Multiplexor를 설계한 것이다. 이제부터의 예제에서는 사용되는 문법을 익히기 바란다. 예제 3, 4와 예제 5가 다른 점은 Process문이 없다는 것이다. 앞에서도 설명했듯이 Concurrent와 Sequential 문의 차이지만 기능은 같다. 여기서 자기 스타일에 맞는 예제를 선택해 자기 자신의 코딩 스타일을 갖는 것은 중요하다. 그 전에 Delay 문제와 Gate Counter를 통해 어떤 것이 가장 잘 설계되었는지 확인하는 것이 중요하다. 대부분의 Vendor Tool에는 Gate Counter가 내장되어 있다. (없는 것도 있지만)
가장 최적화 되어있는 방식을 채택하는 것이 좋겠다. 이것은 가르쳐 줄 수 있는 문제가 아니라 자기 자신이 직접 경험으로 아는 것이 중요하다.

문법(?)적인 설명을 하자면 예제3의 When-else 문은 Process 문에서는 사용될 수 있는 코딩이다. 마찬가지로 Case-when 문은 Process문 밖에서는 사용될 수 없다. 지금 현재는 모든 코딩을 Signal Assign으로 가고 있다. Signal 뿐만 아니라 Variable 이라는 것도 있다는 것을 알고 있기를 바란다. (Variable Assign은 ' := ' 이다.) Signal 과 Variable을 지금 동시에 설명한다면 현재로서는 이해하기 어려울 것이다. 그래서 그 부분은 잠시 미루기로 하겠다.
Case-when은 범용 적으로 잘 사용되는 스타일이므로 잘 기억하기 바랍니다.!!
std_logic은 1비트를 설명하는 것이며 1비트 이상에서는 **std_logic_vector(n downto 0)** 사용합니다. **std_logic_vector(n downto 0)**는 n비트를 표시하는 것이며, 여기서 n은 MSB를 나타내는 것입니다. 또한 **std_logic_vector(0 to n)**는 0이 LSB를 표시합니다.
또, 1비트에서는 **'0'** 또는 **'1'**을 표시하고 1비트 이상에서는 **"00"** 큰따옴표를 사용해요.

그럼 2-way 4×1 Multiplexor를 한 번 설계해 보세요. 그러니까 입력 4개가 각각 2비트이고, 출력이 2비트를 설계하는 것입니다. 그리고 4-way 4×1 Multiplexor를 한 번 스스로 설계해 보세요. (숙제)

그리고 참조로 말씀드리면 대부분의 고급 설계 Tool은 Unix 환경에서 작동하는 것이 많습니다. 그러니 당연히 Unix의 기본은 알고 있어야 하겠죠. Unix 환경을 구하기 힘들다면 Linux를 한 번 시도해 보는 것도 좋습니다. vi 에디터 정도는 기본적으로 알고 있어야 합니다.

조금 복잡한 MUX

그럼 2-way 4×1 Multiplexor를 한 번 설계해 보세요. 그러니까 입력 4개가 각각 2비트이고, 출력이 2비트를 설계하는 것입니다. 그리고 4-way 4×1 Multiplexor를 한 번 스스로 설계해 보세요. 이것이 숙제였죠.

```
-- 2-way 4×1 Multiplexor

library ieee;
use ieee. std_logic_1164.all;

entity mux41 is
port ( a, b, c, d : in std_logic_vector(1
                    downto 0);
     sel : in std_logic_vector(1 downto 0);
     y   : out std_logic_vector(1 downto 0));
end mux41;

architecture rtl of mux41 is
begin
   process(sel)
   begin
      case sel is
         when "00" => y <= a;
         when "01" => y <= b;
         when "10" => y <= c;
         when others => y <= d;
     end case;
   end process;
end rtl;
```

```
-- 4-way 4×1 Multiplexor

library ieee;
use ieee. std_logic_1164.all;

entity mux41 is
port (a, b, c d : in std_logic_vector(3
                  downto 0);
      sel : in std_logic_vector(1 downto 0);
      y : out std_logic_vector(3 downto 0));
end mux41;

architecture rtl of mux41 is
begin
   process(sel)
   begin
      case sel is
         when "00" => y <= a;
         when "01" => y <= b;
         when "10" => y <= c;
         when others => y <= d;
      end case;
   end process;
end rtl;
```

앞에서 std_logic은 **nine values**를 가지고 있다고 했습니다. 기억이 납니까?
그것을 이용하죠. 입력이 4비트의 6개의 입력을 가진 MUX를 한번 설계해 봅시다.
입력이 6개이면 이것을 선택하는 Select 단자는 3개가되어야 하겠죠.
그럼 입력이 6개임으로 2개의 단자는 **don't care**로 처리하는 것이 당연하겠죠. 꼭 don't care로 처리할 필요는 없습니다. 그렇지만 임의로 don't care로 처리합시다.

ex1)

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

entity mux41 is
port (a, b, c, d, e, f : in std_logic_vector(3 downto 0);
          sel          : in std_logic_vector(2 downto 0);
           y           : out std_logic_vector(3 downto 0));
end mux41;

architecture rtl of mux41 is
begin
   process(sel, a, b, c, d, e, f)
   begin
      case sel is
         when "000" => y <= a;
         when "001" => y <= b;
         when "010" => y <= c;
         when "011" => y <= d;
         when "100" => y <= e;
         when "101" => y <= f;
         when others => y <= "----";
      end case;
   end process;
end rtl;
```

그렇게 문법적인 내용은 없지요. Architecture Body의 마지막 문장에 주의해서 보십시오.

```
 when others => y <= "----";
```

y의 출력 값으로 보내는 것이 " ---- " 이지요. 이것이 don't care로 처리하는 것입니다.
std_logic의 nine values 중의 하나인 don't care입니다.

```
 'U',  -- Uninitialized          'X',  -- Forcing  Unknown
 '0',  -- Forcing  0             '1',  -- Forcing  1
 'Z',  -- High Impedance         'W',  -- Weak     Unknown
 'L',  -- Weak     0             'H',  -- Weak     1
  '-'   -- Don't care
```

이 nine values는 Standard Library에 정의되어 있습니다. 보통 라이브러리 디렉토리를 열어 보면 std1164.vhd라는 파일이 있습니다. 이 속에 정의되어 있습니다.

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

이 구문이 Standard Library를 사용하겠다는 말입니다. 앞에서 이 파일을 구경해 보라고 말씀드린 것 같은데 구경은 해 보았습니까? 해보지 않았다면 이 강좌에서는 네 가지의 다른 툴의 Library 파일을 동봉하겠습니다. 편의상 파일명을 바꾸어서 올려 드리겠습니다.

Altera. vhd - Max Plus Ⅱ의 std1164.vhd 파일
synopsis. vhd - Synopsis의 std_logic_1164.vhd 파일
lodecap. vhd - ETRI의 std_logic_1164.vhd 파일
xilinx - Xilinx의 stdlogic.vhd 파일

그냥 메모장으로 링크 하면 읽을 수 있습니다. 한 번 보세요. 회사마다 조금의 차이는 있지만 표준 규격은 같다는 정도만 아시면 됩니다. 얼마나 툴을 최적화 하느냐 하는 차이겠죠.

다시 본문으로 가서 초기화되지 않은 상태로 두고자 한다면
when others => y <= "**uuuu**";로 처리하면 되겠지요. 여기서 대문자 소문자는 별 의미가 없습니다. 그러나 Unix에서는 주의하셔야 합니다. 파일명 등에서 대소 문자를 혼동하면 안되겠지요. Reserved word에서는 별 의미가 없습니다.
또는 when others => y <= "0000";로 영으로 정하면 되겠지요.

지금 하고 있는 디지털 논리 회로는 문법적인 면에서는 별로 어렵지 않습니다. 그러나 그 기능은 반드시 알고 있어야 합니다. 예를 들어 Mux는 입력이 들어오면 그 중에 하나를 선택하는 것이라는 기능은 반드시 알아야 합니다. 이런 간단한 회로들은 Logic으로 구현되어 Symbol로 정의가 되어 있습니다. 완성된 것을 설계하는 것은 별로 의미가 없지만 VHDL의 설계에서는 가장 기초가 되는 내용이니 하찮게 여기지 맙시다.

RTL Level, synthesis이니 하는 어려운 내용은 기초적인 것이 끝나면 정리하기로 하겠습니다. 여러 가지를 설계하다 보면 그 과정이 어떻게 수행되는지 더 잘 이해가 되기 때문에 나중으로 미루고 있습니다. 제 방식이 마음에 들지 않겠지만 제 경험으로는 이 방법이 더 좋을 것이라 생각됩니다. 그리고 이 글을 보시는 분은 적어도 VHDL이 무엇인지 한번은 들어본 사람이라고 생각됩니다. IDEC에서 강의를 들을 때 이론만으로 이해하기는 힘들었습니다. 그리고 모든 강의는 원론에서 시작합니다. VHDL의 가장 핵심이 되는 내용을 이론만으로 다 이해하기는 힘들었습니다. 그래서 저는 기초 과정을 두 번 들었습니다. 그러니 어느 정도 이해가 되었습니다. **원래 엿장수 마음대로 이니까요 ! ! !**

다음 예제는 Enable을 갖는 **4-way 4×1 multiplexor**입니다. 여기에는 가장 빈도가 높은 **If-else** 구문이 사용됩니다. 잘 봅시다.

ex2)

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

entity mux41 is
port (a, b, c d : in std_logic_vector(3 downto 0);
       enable    : in std_logic;
          sel    : in std_logic_vector(1 downto 0);
           y     : out std_logic_vector(3 downto 0));
end mux41;

architecture rtl of mux41 is
begin
   process(sel)
   begin
      if (enable='0') then
      case sel is
         when "000" => y <= a;
         when "001" => y <= b;
         when "010" => y <= c;
         when others => y <= d;
     end case;
     else
         y <= "0000";
     end if;
   end process;
end rtl;
```

if-else 구문은 다음과 같은 형태를 합니다.

```
if ( 조건 )  then
    수행식;
else
    수행식;
end if;
```

if-else 구문은 순차적으로 진행되므로 **반드시 Process 문안에서 실행됩니다.** 중요한 사항입니다. 언어의 형태와 비슷하죠. 괄호는 필요한 사항은 아닙니다만 자기가 보기가 편하기 때문에 저로 주로 괄호를 이용합니다.

Enable 신호가 있으면 0000으로 Set되고 Enable 신호가 영이면 Mux의 기능을 수행합니다. 라이브러리에서는 Symbol로 구현되어 있으므로 반드시 그 형태를 따라가야 하지만 VHDL에서는 자기 자신에 맞는 코딩을 할 수 있다는 것은 굉장히 유용한 사실입니다.

다음은 Decoder와 Encoder에 대해서 알아보기로 하겠습니다. 기능은 모르고 있다면 디지털 논리 회로를 참조하기여 알아보시기를 바랍니다.

ex3) 2-to-4 decoder

```
library ieee;
use ieee. std_logic_1164.all;
entity decoder is
port ( a : in std_logic_vector(1 downto 0);
       d : out std_logic_vector(3 downto 0));
end decoder;
architecture decoder1 of decoder is
begin
   process
   begin
      case a is
         when "00" => d <= "0001";
         when "01" => d <= "0010";
         when "10" => d <= "0100";
         when others => d <= "1000";
      end case;
   end process;
end decoder1;
```

decoder라는 개념은 나중에 살펴볼 **One-hot Coding**이라는 개념과 비슷할 것입니다. 칩의 크기는 커지지만 Performance는 증가한 다는 것을 알 수 있을 것입니다.

architecture name이나 entity name이 reserved word와 이름이 달라야 한다고 말씀 드렸지요. 그럼 reserved word는 어떤 것이 있는지 살펴봅시다.

abs access after alias all and architecture array assert attribute begin block body buffer bus case component configuration constant disconnect downto else elsif end entity exit file for function generate generic guarded if in inout is label library linkage loop map mod nand new next nor not null of on open or others out package port procedure process range record register RAM report return select severity signal subtype then to transport type units until use variable wait when while with xor

조금 생소한 느낌을 주는 것들이 많지요. entity name이나 architecture name이 위의 이름을 가지면 안 된다는 뜻이죠. and는 안 되지만 and1은 다른 의미를 가지므로 가능하겠죠. 위의 내용은 그렇게 어려운 내용이 아니므로 그냥 넘어갑니다. Encoder에 대해서 알아봅시다.

ex4) 4-to-2 decoder

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

entity encoder is
port ( a : in std_logic_vector(3 downto 0);
       z : out std_logic_vector(1 downto 0));
end encoder;

architecture encoder1 of encoder is
begin
   z <= "00" when a(0) = '1'
    else "01" when a(1) = '1'
    else "10" when a(2) = '1'
    else "11";
end encoder1;
```

여기서 a(0)라고 하는 것은 입력 a가 4비트이므로, 4비트의 LSB(최하위 비트)를 말하는 것입니다. a(3)이라고 한다면 MSB(최상위 비트)를 말하는 것입니다. 이 구문은 앞에서 다른 적이 있는데 기억하고 있습니까? Concurrent 문에서 사용되는 구문이라고 앞에서 설명 드렸지요. 논리 회로의 기초적인 지식이 없으면 아무리 제가 쉽게 설명한다고 해도 따라오기가 힘들 것입니다. 부족하다고 느끼시면 다시 시작하십시오. 지금은 늦은 때가 아닙니다. 앞으로도 너무 많은 것이 남아 있습니다. 제가 언제 이것을 다할 지는 모르겠지만 하는데 까지는 해 봅시다. 자신감을 가지세요.

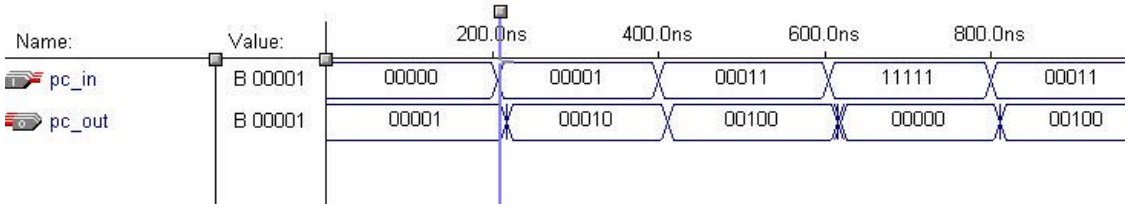
Increment, Decrement, 비교 연산 회로

오늘은 Increment, Decrement, 비교 연산 회로에 대해서 알아보기로 하겠습니다.
Increment는 입력을 받으면 +1을 해서 출력으로 내보는 것입니다. 원래 ALU의 기능에 포함되지만 독립적으로 해석해 보겠습니다. 클럭을 입력받아야 하지만 클럭의 기능은 빼고 시작하겠습니다. Decrement는 그 반대로 -1의 기능을 수행합니다. 예를 들자면 입력이 3이라고 가정하면 Increment는 4를 출력할 것이고, Decrement는 2를 출력할 것입니다. 비교기는 간단히 말해 그 입력을 비교하는 것입니다. 여기서는 a가 b보다 크면 1을 출력할 것이고 b가 크면 0을 출력할 것입니다. 간단한 논리 회로입니다. 간단하다고 생각되지만 이 기능이 나중에 큰 기능을 담당합니다.

ex1)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity inc is
port ( pc_in  : in std_logic_vector(4 downto 0);
       pc_out : out std_logic_vector(4 downto 0));
end inc;

architecture rtl of inc is
begin
  pc_out <= pc_in + 1;
end rtl;
```

여기에서는 결과 파형을 조금 살펴봅시다.



입력이 0, 1, 3, 1f, 3으로 되어 있습니다. 파형을 관찰할 때 주의 할 점은 모든 구간의 영역을 다 살펴보아야 합니다. 5비트 입력이므로 0h에서 1fh까지 모두 살펴보아야 하지만 사실 그러기는 어렵죠. 그래도 시작점과 끝 부분 그리고 입력이 바뀔 수 있는 부분은 모두 체크해야 합니다. 그리고 이 그림에서 알 수 있는데 delay라는 것을 느낄 수가 있죠. 그리고 왜 입력이 4비트가 아니라 5비트인가? 이 점에 대해서 살펴보아야 할 것입니다. Intel의 기본은 8비트입니다. 그러나 이 8비트는 4비트 두 개가 모여서 8비트를 이루는 것입니다. 4비트 연산을 위해서는 5비트의 공간이 필요합니다. 여기에서 보면 11111이 입력입니다. 출력은 00000으로 나와 있지만 값은 원래 값은 100000이 될 것입니다. 그러면 왜 출력을 6비트로

정의하지 않았느냐 하는 질문을 할 수 있을 것입니다. 그러나 상태 Flag에서 이 값을 체크해 carry로 그 값을 보낼 것입니다. 지금은 간단하게 5비트 입력, 5비트 출력 이렇게 되어 있지만, 다음에 살펴볼 ALU에서 입력이 4비트 출력이 4비트인 회로를 설계할 것입니다. 그 때는 내부 Signal이라는 것을 이용하게 될 것입니다. 내부 시그널은 다음, 다음 시간에 배우게 될 것입니다.

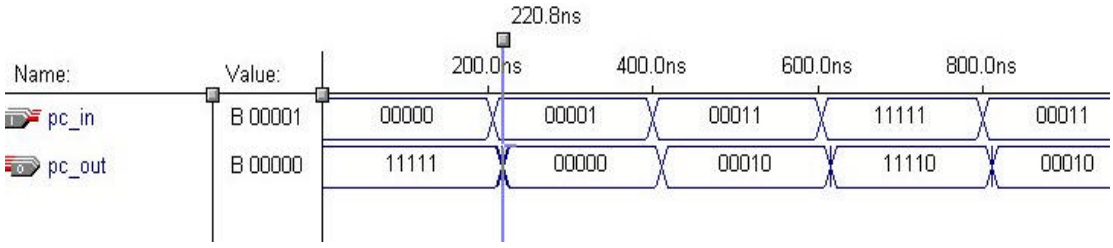
우리가 지금까지 사용하지 않은 구문이 있군요.
use ieee. std_logic_signed. all;
사용하는 Values가 signed 값을 사용한다는 의미입니다. 여기서는 그렇게 큰 의미는 없지만 unsigned 값을 사용한다면 달라지겠죠.

다음은 Decrement 회로를 정의해 볼까요.
ex2)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity dec is
port ( pc_in  : in std_logic_vector(4 downto 0);
       pc_out : out std_logic_vector(4 downto 0));
end dec;

architecture rtl of dec is
begin
  pc_out <= pc_in - 1;
end rtl;
```

+, - 기호 하나만 바꾸면 됩니다.
그리고 결과는 다음과 같습니다.

만약 signed package가 없다면 어떻겠습니까?



-- **use** ieee. std_logic_signed. all;

주석으로 처리한 후 컴파일을 해보세요. 결과가 어떻게 나오는지?

ex3) less than 비교 연산 회로

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity less is
port ( a, b  : in std_logic_vector(1 downto 0);
          z   : out std_logic);
end less;

architecture rtl of less is
begin
   process(a, b)
   begin
      if (a < b) then
         z <= '1';
      else
         z <= '0';
      end if;
   end process;
end rtl;
```

간단한 비교기를 보았는데 less than만 될까요? 아닐 것 같지요. 그럼 변형을 해봅시다.

```
  if (a < b) then   -- b가 a보다 크다면
  if (a > b) then   -- a가 b보다 크다면
  if (a <= b) then   -- b가 a보다 크거나 같다면
  if (a >= b) then   -- a가 b보다 크거나 같다면
  if (a = b) then   -- a와 b가 같다면
  if (a /= b) then   -- a와 b가 같지 않다면
```

한 줄을 바꿈으로서 6개의 회로를 설계할 수 있습니다.
관계 연산자는 위의 6가지 형태입니다. 유용하게 쓰일 때가 있을 것입니다.

오늘은 간단하게 끝내겠습니다. 제가 좀 피곤해서요.
다음 시간에는 D Flip-flop에 대해서 설계하도록 하겠습니다. FF과 Latch의 차이점을 미리 알아두시는 것이 편할 것입니다.

DFF과 latch

우리는 제일 먼저 논리 회로에서 AND, OR, NOT 게이트를 먼저 배우고 그 다음에 NAND, NOR 게이트를 배우게 됩니다. AND, OR의 기능을 가지면 NAND와 NOR를 설명하기 쉽기 때문입니다.

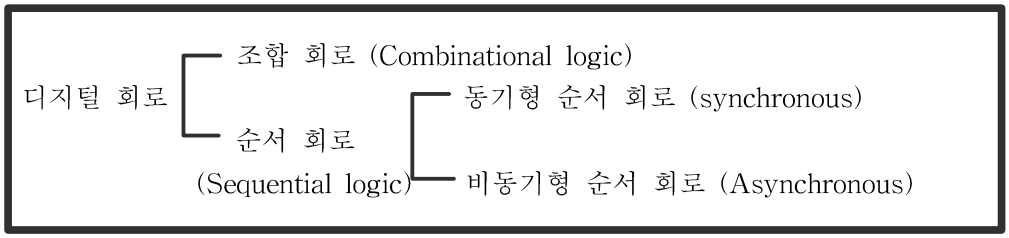
그러나 실제 회로에서는 그 개념이 아닙니다. 회로의 최적화를 이루기 위해서는 AND나 OR보다는 NAND와 NOR 게이트를 가지고 설계하는 것이 가장 최적화 되어 있는 회로입니다. 집적회로의 입장에서 보면, PNP Junction에서 NMOS와 PMOS를 가지고 설계할 수 있는 가장 기본의 형태가 **Nand, Nor** 게이트입니다.(집적회로까지 설명하면 너무 깊이 들어가야 함으로 간단하게 설명하겠습니다. 궁금한 점이 있는 집적회로를 공부하는 것이 도움이 될 것입니다. Spice Tool을 이용하여 간단한 Layout 설계를 해보시면 알 것입니다.) NAND 게이트는 and 게이트에 not 게이트를 붙여서 설계한 회로입니다. 그러니 NAND, NOR, NOT가 회로의 기본입니다. NAND만 보면 PMOS 두 개를 병렬로 연결하고 NMOS 두 개를 직렬로 연결한 것이 NAND 회로입니다. 참고로 말씀드리면 설계의 기본은 Layout 과정입니다. 배워 두시면 도움이 될 것이라 생각은 들지만 시간이 많이 걸린다는 것이 단점입니다.(?) 그럼 ASIC 설계에서 가장 기본적으로 알아야 하는 것은 무엇일까요? 아마도 DFF이라고 저는 생각하는 데요. 이제 설계자의 입장에서 말씀드리겠습니다. 기본적인 회로의 설계가 끝나면 여러분은 설계자 입장에서 모든 것을 생각하고 설계를 해야 합니다. 하드웨어 디자이너가 되는 것입니다. 기쁘겠죠.

디자이너가 가장 중요하게 생각해야 하는 Factor가 네 가지 있습니다. 아니 더 많을 수도 있지만 기본적인 4가지가 있습니다. 그 중에 한 가지인 **Area(yield and packaging cost)** 문제가 있습니다. 그 문제가 인해 **회로 면적이 적게 차지하는 DFF을 이용하는 것**입니다. 그래서 VHDL 코딩을 하면 컴파일을 하고 synthesis를 하고 나서 logic으로 구현된 설계도를 보면 메모리 소자를 DFF으로 설계된 것을 확인할 수 있을 것입니다. 지금 살펴 볼 문제는 아니지만 설계시 고려해야 할 4가지 요소에 대해서 설명 드리면 다음과 같습니다.

- **Performance :**
  - **Delay and cycle-time.**
  - **Latency.**
  - **Throughput (for pipeline application).**
- **Area (yield and packaging cost).**
- **Testability**
- **Power**

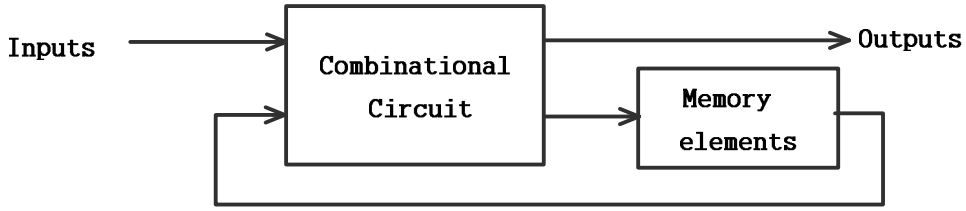
기분 나쁘게 전부 영어로 작았죠. 다른 사람과 설계에 대해서 이야기를 한다면 영어로 말을 하여 야죠. 다른 사람이 성능이 어때? 라고 묻지는 않을 것입니다. “Performance가 어때? “라고 묻지. 위의 설명은 여러분이 디자이너가 되면 시작할 것입니다. 그 때를 기대하세요. 그렇게 어려운 용어가 아닌지 대강은 감을 잡을 수 있을 것입니다.

지난 시간에 DFF과 latch에 차이에 대해서 알아보셨나요. 부지런한 디자이너는 조사해서 알아있을 것이고, 귀찮은 사람은 그냥 기다리겠죠. 그러나 이런 단순한 이론에서 오는 차이가 크다는 것을 알아야 합니다. 그만큼 회로 설계시간의 단축을 가져올 수 있습니다.



이 시간을 이해하기 위해서는 조합 회로가 무엇이고 순서 회로가 무엇인지 알아야 합니다.

조합 회로의 블럭 선도



순서 논리회로의 블럭 선도

조합 회로는 임의의 시간에서 출력이 이전의 입력에는 관계없이 현재의 입력 조합으로부터 직접 결정되는 논리회로를 말합니다. 반면 순서 회로는 출력은 외부로부터의 입력이 기억 상태를 바꾸게 된다. 즉 순서 회로의 입력, 출력 그리고 내부 상태의 시간에 따른 순서로서 규정된다. 쉽게 말하면 조합 논리의 결과가 출력이 되는 것이 아니라 메모리 요소에 저장되어 있는 변화된 값이 다시 조합 회로로 궤환되어 결과 값이 출력되는 것이다. 이것은 다시 두 가지로 나누어진다. 클럭 펄스가 들어오는 시점에서 입력 신호로부터 그 동작을 정의할 수 있는 시스템이 동기형(synchronous) 순서 논리회로이다. 비동기형(Asynchronous) 순서 회로 동작은 입력 신호가 변화하는 순서에 따라 언제라도 영향을 받을 수 있다.

Flip-flop이나 latch는 one-bit 메모리 소자입니다. 클럭이 있는 순서 논리회로에 쓰이는 기억 소자를 플립플롭이라고 한다. FF은 edge-triggered memory device라고 합니다. 이 말은 플립플롭의 상태는 입력 신호의 순간적인 변화에 따라 바뀌기 때문입니다. 순간적인 입력 변화를 트리거(trigger)라고 합니다. FF을 흔히 레지스터라고도 합니다.

비동기식 순서 회로에서는 클럭 펄스를 사용하지 않는다. 비동기식 순서 회로에서의 기억 소자로는 클럭이 없는 플립플롭(latch)이거나 시간 지연 소자(time-delay element)를 사용한다. 비동기식 순서 회로 설계는 타이밍 문제 때문에 설계가 더 어렵습니다.

latch는 level-sensitive memory device입니다. 그 말은 클럭을 사용하지 않기 때문에, 입력이 변화되는 즉시 시스템의 상태가 변화할 수 있게 되어야 한다는 뜻이죠.

클럭이 있는 동기식 시스템의 예를 많이 보았을 것이고, 비동기식 회로에서는 빠른 동작

속도가 요구될 때 사용합니다. 예를 들자면 그 자체에 독립적인 클럭을 가진 각각의 장치가 있는 2개의 시스템간의 교신은 비동기식 회로로 이루어져야 합니다.

적절한 시스템 설계를 위해서는 동기식의 일부가 비동기식 특성을 갖는 혼합된 시스템을 설계해야 합니다.

그럼 간단히 정리하죠. **동기식 순서 회로에서 사용되는 기억 소자는 플립플롭이고, 비동기식 순서 회로 사용되는 기억 소자는 레치 입니다. 플립플롭은 클럭이 있고, 레치는 클럭이 없습니다. 좀 어렵게 설명하면 펄스 지속 시간에 민감한 플립플롭 군을 레치라고 하고, 펄스에 전이에 민감한 플립플롭 군을 레지스터라고 합니다.**

이 부분은 조금 중요한 부분이라 조금 길게 설명이 됐는데 되도록 논리회로의 설명은 자제하는 방향으로 하겠습니다.

오늘은 예제가 조금 많습니다. 플립플롭과 레치에 관계된 예제입니다. 이 부분은 Actel 매뉴얼에서 인용했습니다. 인터넷 Actel 홈페이지에 가서 다운 받을 수도 있고, 사용자 등록을 하시고 매뉴얼 신청을 하시면 구할 수 있을 것입니다. http://www.actel.com

오늘 예제는 모두 DFF으로 구현된 예제입니다. 다른 플립플롭으로 구현은 아마 힘들 겁니다. 칩 설계를 위해서 그런 것이니 JKFF, RSFF, TFF 등을 구현하고자 한다면 Logic에서 설계된 것을 이용하시면 될 것입니다.

ex1) Rising Edge Flip-Flop

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

entity DFF is
port ( data, clk : in std_logic;
          q      : out std_logic);
end DFF;

architecture behav of DFF is
begin
process(clk)
begin
   if (clk='1' and clk' event) then
      q <= data;
   end if;
end process;
end behav;
```

ex2) D-Latch with Data and Enable

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

entity d_latch is
port ( data, enable : in std_logic;
             y      : out std_logic);
end d_latch;

architecture behav of d_latch is
begin
process(enable, data)
begin
   if (enable='1') then
      y <= data;
   end if;
end process;
end behav;
```

생각보다 구현은 간단하죠. process 문안에 재미있는 표현이 있군요.

if (clk='1' and clk' event) then

q <= data;

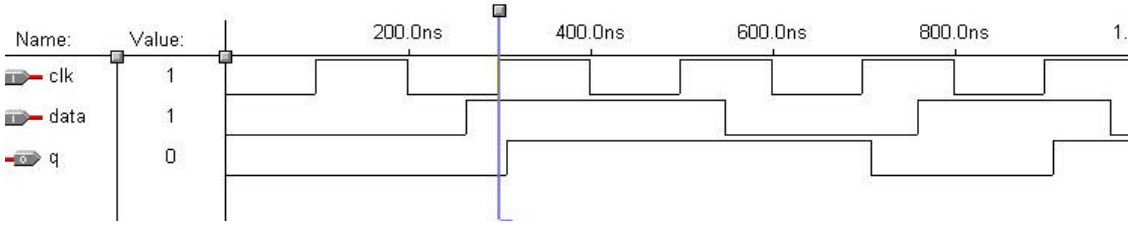
이 구문은 어떤 의미일까요? 플립플롭에서는 Trigger 방법을 이용한다고 했는데 말이죠. event를 이용해서 상승에지를 표현한 것입니다. 외우세요.

그럼 그 반대로 하강에지를 표현하면 어떻게 될까요? 1을 0으로 바꾸면 되겠죠.

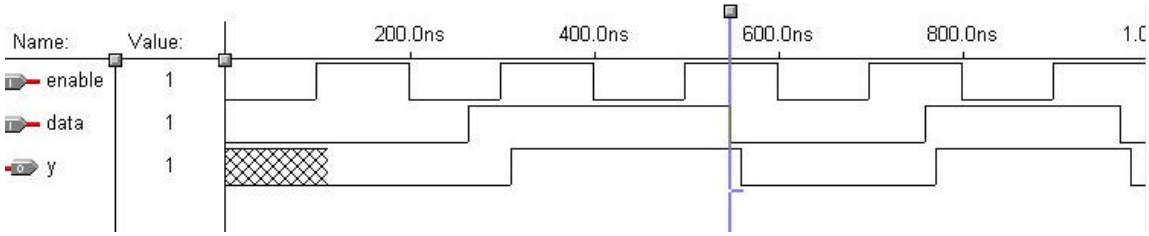
if (clk='0' and clk' event) then

q <= data;

설명할 것은 이것뿐이네요. 다음 예제로 넘어가지 전에 결과를 비교해 볼까요.



그림에서 보듯이 클럭의 입력에 따라 출력이 변한다는 것을 알 수 있습니다.



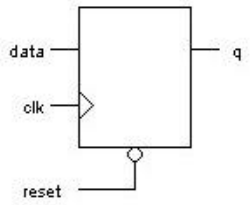
레치는 입력에 따라 값이 변한다는 것을 알 수 있을 것입니다.

그림이 조금 정확하게 보이지 않는데 주의해서 보시면 될 것입니다.

ex3) Rising Edge Flip-Flop with Asynchronous Reset

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

entity DFF is
port ( data, clk, reset : in std_logic;
          q             : out std_logic);
end DFF;

architecture behav of DFF is
begin
process(clk, reset)
begin
```

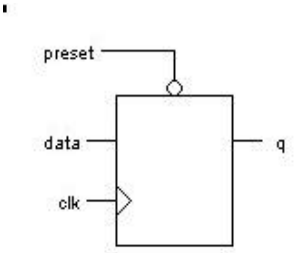
```
    if (reset='0') then
        q <= '0';
    elsif (clk='1' and clk' event) then
        q <= data;
    end if;
end process;
end behav;
```



설명할 것이 없네요. 말 그대로입니다. reset을 가진 플립플롭입니다. 다만 reset이 0일 때 플립플롭의 기능을 수행하겠죠. 코딩은 reset이 1일 때라고 의문을 가지는 분은 위의 그림을 유심히 보시면 됩니다. reset에 not가 붙어 있죠.

ex4) Rising Edge Flip-Flop with Asynchronous Preset

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

entity DFF is
port ( data, clk, preset : in std_logic;
           q             : out std_logic);
end DFF;

architecture behav of DFF is
begin
process(clk, preset)
begin
    if (preset='0') then
        q <= '0';
    elsif (clk='1' and clk' event) then
        q <= data;
    end if;
end process;
end behav;
```



C 언어에서 if-else 구문이 있죠. 이것과 사용법은 거의 비슷합니다. 다만 C 언어에서는 else if 라고 쓰지만, VHDL에서는 **elsif** 라고 씁니다. 다른 언어와 틀리니 주의하세요. 다른 언어에서처럼 if 구문 속에 if 문이 계속해서 들어갈 수 있습니다. 나중에 예제에서 보겠죠.

ex5) Rising Edge Flip-Flop with Asynchronous Reset and Preset

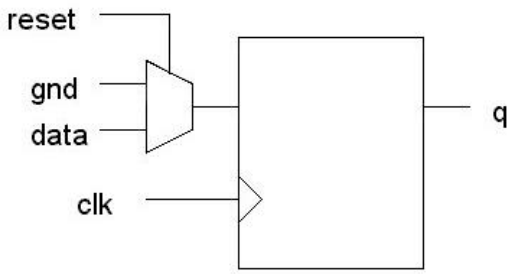
```
library ieee;
use ieee. std_logic_1164.all;

entity DFF is
port ( data, clk, reset, preset : in std_logic;
          q                     : out std_logic);
end DFF;

architecture behav of DFF is
begin
process(clk, reset, preset)
begin
   if (reset='0') then
      q <= '0';
   elsif (preset='0') then
      q <= '0';
   elsif (clk='1' and clk' event) then
      q <= data;
   end if;
end process;
end behav;
```

Asynchronous Reset and Preset은 시스템이 불안정할 수 도 있다는 것을 알고 있어야 합니다. Asynchronous Reset and Preset보다는 Synchronous Reset and Preset이 안정적이라는 것을 알고 있어야 합니다. 다시 말하면 Asynchronous Reset은 클럭에 관계없이 Reset이 1이면 모든 것을 리셋 시키는 기능을 합니다. 그렇지만 시스템 회로 설계에는 Asynchronous Reset이 더 많이 사용되고 있습니다.

ex6) Rising Edge Flip-Flop with Synchronous Reset

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

entity DFF is
port ( data, clk, reset : in std_logic;
          q             : out std_logic);
end DFF;
```

```
architecture behav of DFF is
begin
process(clk)
begin
   if (clk='1' and clk' event) then
      if (reset='0') then
         q <= '0';
      else
         q <= data;
      end if;
   end if;
end process;
end behav;
```



위 그림을 보면 어떤 의미인지 알 수 있을 것입니다.
Synchronous Reset을 사용하므로 훨씬 안정적인 회로이겠죠.

ex7) Rising Edge Flip-Flop with Synchronous Preset

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

entity DFF is
port ( data, clk, preset : in std_logic;
          q              : out std_logic);
end DFF;

architecture behav of DFF is
begin
process(clk)
begin
   if (clk='1' and clk' event) then
      if (preset='0') then
         q <= '0';
      else
         q <= data;
      end if;
   end if;
end process;
end behav;
```

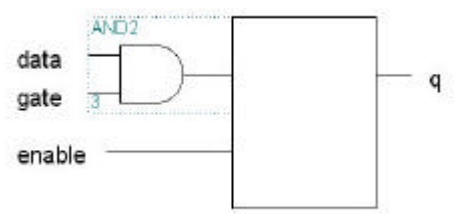
ex8) Rising Edge Flip-Flop with Asynchronous Reset and Clock Enable

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

entity DFF is
port ( data, clk, reset, en : in std_logic;
          q                 : out std_logic);
end DFF;

architecture behav of DFF is
begin
process(clk, reset)
begin
   if (reset='0') then
         q <= '0';
   elsif (clk='1' and clk' event) then
      if (en ='1') then
         q <= data;
      end if;
   end if;
end process;
end behav;
```

Asynchronous Reset과 **Clock**이 **Enable** 될 때 data에 있던 값이 q로 assign 되는 것입니다. 한 가지 예제를 알면 응용하는 범위가 VHDL은 굉장히 넓기 때문에 많은 회로를 쉽게 만들 수 있다는 이점이 있습니다. 우리가 설계한 회로를 라이브러리에 저장해 그것을 불러서 사용할 수도 있습니다. **REUSE 가능한 언어가 VHDL입니다.**

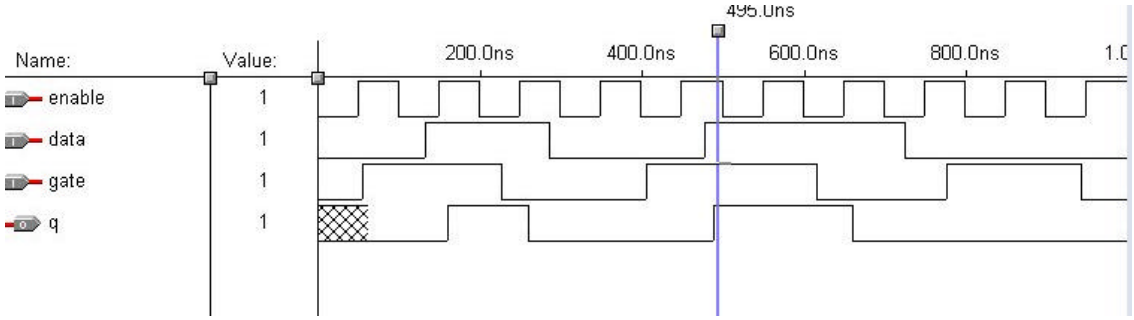
ex9) D-Latch with Gated Asynchronous Data

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

entity d_latch is
port ( data, enable, gate : in std_logic;
          q               : out std_logic);
end d_latch;

architecture behav of d_latch is
begin
process(enable, data, gate)
```



```
begin
    if (enable='1') then
        q <= data and gate;
    end if;
end process;
end behav;
```

레치의 간단한 변형입니다. 그렇게 어려운 것은 없고 출력 값이 변하는 것을 유심히 살펴보기를 바랍니다. 클럭이 시작될 때 변하는 것이 아니라 enable이 1인 영역에서 data와 gate가 둘 다 1인 시점에서 변한다는 점을 시뮬레이션 결과에서 잘 살펴보기를 바랍니다.



ex10) D-Latch with Gated Enable

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

entity d_latch is
port ( data, enable, data : in std_logic;
            q             : out std_logic);
end d_latch;

architecture behav of d_latch is
begin
process(enable, data, gate)
begin
    if ((enable and gate)='1') then
        q <= data;
    end if;
end process;
end behav;
```

if 구문 안에 어떤 것이 변하는지 잘 보면 될 것입니다. 그냥 넘어갑니다.

ex11) D-Latch with Asynchronous Reset

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

entity d_latch is
port ( data, enable, data : in std_logic;
            q             : out std_logic);
end d_latch;

architecture behav of d_latch is
begin
process(enable, data, reset)
begin
   if (reset='0') then
      q <= '0';
   elsif (enable='1') then
      q <= data;
   end if;
end process;
end behav;
```

간단한 예제들을 중심으로 플립플롭과 레치에 대해서 알아보았습니다. 8비트 레지스터를 설계하다 보면 이 부분에 대해서 다시 한번 살펴볼 기회가 있을 것이라 생각됩니다. 위 예제들을 직접보고 코딩하지 마시고, 이런 기능을 가지고 있으므로 어떤 식의 설계가 들어가야 한다라고 자꾸 생각하시면서 직접 설계해 보시는 것이 기억에 오래 남을 것입니다. 전에도 말씀드렸지만 자기 자신의 고유의 코딩 방식을 절대로 잊지 마십시오.

VHDL의 자료형과 객체형

VHDL의 자료형과 객체형, Signal과 Variable과 Constant에 대해서 배우게 될 것입니다. 오늘 배울 것은 **VHDL의 자료형과 객체형**입니다. 이제까지는 std_logic과 bit_logic 그리고 Array인 std_logic_vector 만을 배웠습니다. 다른 언어들처럼 VHDL도 많은 자료형을 사용할 수 있습니다.

VHDL에서 자료형 검사는 매우 엄격하며 정의된 자료형에 따라 사용할 수 있는 연산자 또한 각각 정의되어야 합니다. 합성기(synthesizer)에 따라 서로 지원하는 데이터형이 다를 경우 VHDL 소스 사이의 호환성에 심각한 문제를 일으키며 많은 자료형 변환 함수와 타입 캐스팅 방법이 존재하게 된다. 이러한 문제를 해결하기 위하여 87년 IEEE 1076 VHDL의 표준에 이어 1991년에 IEEE 1164로서 디지털 회로의 합성 가능한 자료형에 대한 표준이 제정되기에 이르렀다. IEEE 1164는 디지털 회로에 대하여 9 가지 값(standard 9-valued logic)을 갖는 데이터 형 "std_logic"을 정의하였고, 이는 IEEE 1076 의 "bit" 형을 확장한 것이다. IEEE 1076에는 VHDL의 언어에 대한 표준과 아울러 데이터 타입 및 각종 연산에 대한 표준이 정해져 있다. IEEE 1076에는 디지털 회로의 비트(bit) 형과 정수 및 실수형 문자형 등이 있으며 이중 bit와 정수형이 합성 가능하다.
IEEE 1164의 "std_logic"은 bit의 '1'과 '0' 이외에 Pull-up, Pull-down에 해당하는 'H'(Weak-High), 'L'(Weak-Low)과 'Z'(high-impedance), 'U'(uninitialize), 'X' (unknown), '-'(Don't care)등을 정의해 놓음으로써 디지털 회로의 합성뿐만 아니라 시스템 인터페이스에 대한 배려를 해 두었다.

우선 용어부터 정리를 할까요. VHDL에서 signal, variable, constant 등과 같이 어떠한 값을 가지고 있는 것을 객체(Object)라고 하며, 모든 object는 data type을 갖는다. 각 object가 가지는 data type은 시뮬레이션 도중에는 변하지 않는 정적(static)이다.

```
signal      a, b  :   std_logic;
객체 선언   객체      자료형(data type)
```

우선 VHDL에서 사용하는 3가지 객체 종류에 대해선 먼저 살펴봅시다.

1. Signal(신호)

```
ex) signal a, b, c : bit;
    c <= a and b;
```

2. Variable(변수)

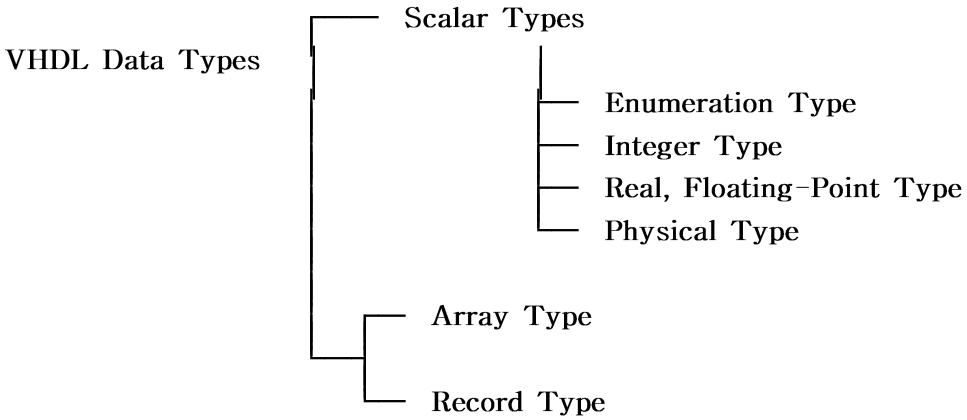
```
ex) variable temp : bit;
    temp := a and b;
```

3. Constant(상수)

```
ex) constant p1 : integer := 314;
```

위의 세 가지 객체형은 다음 시간에 자세하게 다루게 됩니다.

그럼 다음은 어떤 데이터 타입이 있는지 어디 한번 봅시다.



1. 열거형 (Enumeration type)

기본적으로 VHDL에서 사용하는 대부분의 자료형은 대부분 열거형 (enumeration type) 이다. IEEE 1076의 standard package에 정의되어 있는 열거형의 예를 들면 다음과 같다.

```
type boolean is (false, true);
type bit is ('0', '1');
```

IEEE 1164에 std_logic은 다음과 같이 열거형으로 정의되어 있다.

```
TYPE std_logic IS ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-' );
```

'U'는 초기화되지 않은 상태(Uninitialized)를 의미하며 'X'는 Unknown으로서 디지털 값의 충돌 등과 같은 에러 상태를 나타낸다. '0' 과 '1'은 디지털 값에 해당하며 'Z'는 High Impedance, 'W', 'L', 'H' 는 각각 Weak Unknown, Weak 0, Weak 1로서 Pull-up혹은 Pull-down된 디지털 값을 나타낸다. 끝으로 ' - '은 합성 시 논리 최적화에 Don't care로서 이용된다. 열거형을 이용하면 사용자는 언제든지 자료형의 정의가 가능하다. 또한 열거형으로 정의된 경우 인코딩(encoding) 방법을 지정할 수 있다. 인코딩 방법으로는 2진 코드에 의한 방법(binary)과 One-Hot Encoding이 있다. One-Hot encoding 방법은 유한 상태 머신(FSM : Finite State Machine)에서 상태를 나타내는 경우에 많이 이용되는 방법이다. 다음과 같은 예를 살펴보자.

```
type state is (idle, receive, send);
```

와 같이 정의된 자료형 "state" 의 경우 binary와 One-Hot encoding 되었을 때 차이는 다음과 같습니다.

type state is (idle, receive, send)

	Bit2	Bit1	Bit0
Idle	-	-	1
Receive	-	1	-
Send	1	-	-

One-hot Encoding

	Bit1	Bit(
Idle	0	0
Receive	0	1
Send	1	0

Binary Encoding

예제에서와 같이 열거형 "state" 는 3개의 요소를 가지므로 Binary encoding하면 2비트

가 필요한 반면 One-Hot encoding하는 경우 각 요소마다 1개의 비트를 할당하여 3비트로 표현된다. 잠시 소개가 있었지만, 회로의 Performance를 고려할 때 One-hot Coding은 아주 유용한 방식 중에 하나이다. VHDL에서 형 검사(type checking)가 치밀한 이유 중 하나가 위와 같은 열거형을 주로 다루기 때문이라고 할 수 있다. 각종 연산자 혹은 할당(assignment)의 경우 기본적으로 열거형으로 이루어진 서로 다른 두 타입은 비트 폭에서부터 다르기 때문이다. 다음과 같이 bit 형과 std_logic형의 할당문의 경우 예를 살펴보자.

```
Type bit is ( '0' , '1' );
Type std_logic is ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-' );
```

와 같이 정의된 두개의 시그널을 사용한 할당문은 하드웨어적인 연결의 의미를 고려해 볼 때 할당문은 성립할 수 없는 것이 당연하다.

```
Signal A : bit;
Signal B : std_logic;
A <= B;
```

2.. 정수형 (Integer Type)

IEEE 1076에 정의된 정수형의 기본적인 비트 폭(bit-width)은 32비트로 되어 있다.

```
type integer is range -2147483648 to 2147483647;
```

정수형은 산술 연산이 가능하며 2의 보수 형태로 이루어진다. 또한 정수형은 합성이 가능하며 범위를 지정하지 않을 경우 32비트가 된다. 따라서 과도한 비트 폭을 갖는 연산기의 합성을 피하려면 정수의 범위를 지정해서 사용하도록 한다. 다음의 예는 4비트 Signal을 선언한 예이다.

```
Signal count : integer range 0 to 15;
Signal count : std_logic_vector(3 downto 0);
```

ex1)

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

entity add is
port ( a, b : in integer range 0 to 15;
         c  : out integer range 0 to 15);
end add;

architecture rtl of add is
begin
   c <= a + b;
end rtl;
```

위의 예제는 4비트 Adder를 설계한 것이다. 다음과 비교해 보라.

ex2)

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

entity add is
port ( a, b : in std_logic_vector(3 downto 0);
          c  : out std_logic_vector(3 downto 0);
end add;

architecture rtl of add is
begin
   c <= a + b;
end rtl;
```

구현을 해보면 같은 회로라는 것을 알 수 있다. 그러나 수학적인 연산 기능을 수행할 때 비트 단위로 계산하는 것보다는 **Integer**로 변환해서 사용하는 것이 이해에 훨씬 도움을 줄 것이다.

3. 실수형 (Real, Floating-Point Type)

실수형(real type)은 합성 가능하지 않다. 시스템 모델링이나 아날로그 회로의 모델링 등에 사용될 수 있다. IEEE 1076의 실수형의 정의는 다음과 같다.

```
type real is range -1.0E38 to 1.0E38;
```

실수형의 디폴트 범위가 위의 범위이다. 그렇지만 우리가 설계하는 회로가 위의 범위를 항상 사용하는 것은 아니다. 그래서 범위를 지정해 주는 것이 효율적일 것이다.

4. Physical Type

물리량의 단위(unit)와 배수 관계를 정의한 것이며 합성 가능하지 않다. IEEE 1076에는 시간에 대한 정의가 있는데 다음과 같다.

```
time is range -2147483647 to 2147483647
units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  ms = 1000 us;
  sec = 1000 ms;
  min = 60 sec;
  hr = 60 min;
end units;
```

위에서 볼 수 있듯이 VHDL의 최소 시간 단위는 fs (femto second, 10E-15) 이다.

5. 배열형 (Array Type)

VHDL에서 배열형을 사용할 수 있다. 일 차원 배열형의 경우 대부분 합성기에서 지원 하지만 2차원 배열을 지원하지 않는 합성기도 많다. 디지털 회로의 기본 데이터 단위는 "bit" 이다. 이를 버스 형태로 정의하기 위해서 "bit_vector" 형을 사용한다. 즉 버스를 정의하는 것은 "bit" 의 일 차원 배열이다. 배열형을 정의 할 때 그 크기를 지정하는 경우가 있고 임의의 배열 크기를 지정할 수도 있다.

type byte is array (7 downto 0) of bit;

일차원 배열로 8-비트 크기의 데이터 타입 "byte" 을 정의한 것이다.

TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;

"std_logic" 형으로 일 차원 배열형 "std_logic_vector" 를 선언한 것이다. 이때 배열의 크기를 제한시키지 않은 것으로서 시그널을 선언할 때 그 크기를 정하여 사용할 수 있다. 배열형의 선언한 예는 다음과 같다.

```
Signal byte_a : byte;
Signal count : std_logic_vector( 7 downto 0);
```

6. 레코드 형(Record Type)

VHDL에서는 레코드형을 지원한다. 디지털 회로에서의 레코드형 사용 예는 기계어 명령의 비트맵 표현 등에 이용될 수 있다. 다음은 레코드형의 사용 예이다.

```
type month_name is (jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dev);

type date is
 record
 day : integer range 1 to 31;
 month : month_name;
 year : integer range 0 to 4000;
end record;

constant my_birthday : date := (21, nov, 1963);
SIGNAL my_birthday : date;

my_birthday. year <= 1963;
my_birthday. month <= nov;
my_birthday. day <= 21; -- 5-bit

-- my_birthday(0) : LSB
```

-- my_birthday(4) : MSB

7. 파생형 (Subtype)과 형 변환(Type Conversion)

VHDL에서의 파생형(subtype)을 정의할 수 있으며 다음의 예는 정수로부터 일정한 크기를 갖는 파생 정수형을 정의한 예이다.

```
type big_integer is range 0 to 1023;
subtype small_integer is big_integer range 0 to 7;
```

"small_integer" 는 "big_integer" 의 파생형으로 정의되었다.

이 경우 파생형 "small_integer" 의 의미는 10-비트 크기의 "big_integer" 의 하위 4-비트를 차지하도록 정렬된다는 의미로서 두 데이터형의 호환됨을 나타낸다. 다음의 예는 VHDL에서 자료형 검사(type checking)의 엄격함을 보여준다.

```
type big_integer is range 0 to 1000;
type small_integer is range 0 to 7;

signal intermediate : small_integer;
signal final : big_integer;
final <= intermediate * 5; -- type mismatch error
```

위의 예에서 "big_integer" 와 "small_integer" 형은 비록 서로 정수형이긴 하지만 서로 호환 돼지 않는다. 이러한 경우 "small_integer" 를 파생형으로 정의함으로서 형 불일치(type mismatch) 에러를 해결할 수 있다.

```
type big_integer is range 0 to 1000;
subtype small_integer is big_integer range 0 to 7;
signal intermediate : small_integer;
signal final : big_integer;
final <= intermediate * 5;
```

또 다른 방법으로 type casting을 이용할 수 있는데 이 경우는 기본적으로 두 형의 기본형이 같아야 한다.

```
type big_integer is range 0 to 1000;
type small_integer is big_integer range 0 to 7;
signal intermediate : small_integer;
signal final : big_integer;
final <= big_integer(intermediate * 5);
```

만일 정수형과 "std_logic_vector" 형으로 변환하기 위해서는 두 데이터 타이의 기본형이 전혀 다르므로 형 변환 함수(type conversion function)를 사용하여야 한다. 형 변환 함수는 사용자가 임의로 만들어 쓸 수 있으나 IEEE 1164의 산술 패키지(numeric_std, numeric_bit)에는 정수와 "std_logic_vector", "bit" 사이의 형 변환을 위한 함수를 다수 가지고 있으

므로 가급적 이를 이용하도록 권장한다. IEEE 1164 라이브러리의 산술 연산과 형 변환 함수는 합성 가능한 산술 연산을 다루면서 자세히 살펴보기로 한다. 다음의 예는 산술 연산과 형 변환 함수의 이용에 대한 것이다.

```
package my_type is
type big_integer is range 0 to 1023;
subtype small_integer is big_integer range 0 to 7;
end package;

library ieee;
 use ieee. std_logic_1164.all;
 use ieee. numeric_std. all;
 use work. my_type. all;

entity subtype_test is
port (
a : in small_integer;
 b : in small_integer;
 c : out std_logic_vector(9 downto 0) );
end subtype_test;
architecture behave of subtype_test is
 signal t_int : big_integer;
 begin
t_int <= a + b;
 c <= std_logic_vector(to_unsigned(natural(t_int), 10));
end;
```

위의 예는 정수 입력을 받아서 연산을 수행한 후 이를 10 비트 크기의 “std_logic_vector”로 출력하는 경우이다. 입력된 2개의 정수형(“small_integer”) a, b를 계산하여 “big_integer”로 할당한다.

```
t_int <= a + b;
```

사용자 정의형 “big_integer”인 “t_int”로부터 std_logic_vector”로 변환하는 과정은 약간 복잡하게 보이지만 이러한 변환 과정이 시스템 설계와 서로 다른 모듈 사이의 인터페이스 할 때 자주 직면하게 되는 문제이므로 잘 이해 해 둘 필요가 있다.

```
c <= std_logic_vector(to_unsigned(natural(t_int), 10));
```

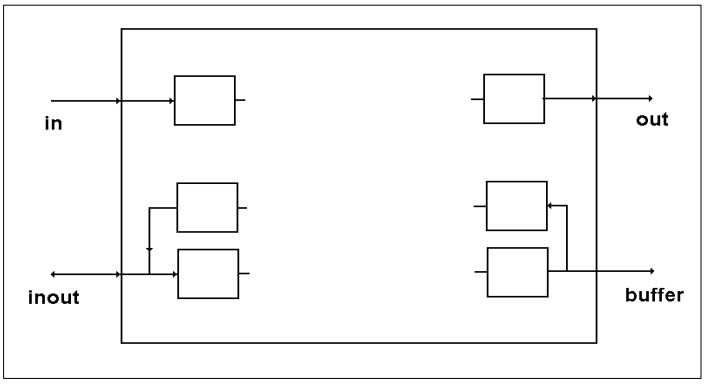
Numeric_std package에 정수-unsigned 변환 함수가 to_unsigned()로 제공되고 있다. 이 함수의 argument가 natural이므로 사용자 정의형으로부터 정수형(natural)으로 type-casting 한다. Unsigned 형으로 변환할 것이므로 integer가 아닌 natural로 type-casting 한 것이다.

정수형을 std_logic_vector로 직접 변환하는 함수를 가지고 있지 않다. 따라서 to_unsigned() 함수를 사용하여 정수를 10비트 unsigned로 변환한 후 다시 type-casting 하여 "std_logic_vector" 인 출력포트에 연결한 것이다. unsigned는 IEEE 1164의 numeric_std에 다음과 같이 정의되어 있으므로 type-casting이 가능하다.

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

이 변환 과정을 보면 알 수 있듯이 같은 기본형으로부터 파생된 자료형은 type-casting 할 수 있지만 기본형이 다른 경우 형 변환 함수를 써야 한다. 그러나 "t_int <= a + b;" 에서와 같이 기본형이 같은 파생형(subtype)의 경우에는 변환을 수행할 필요가 없다. 위의 예를 합성한 결과는 그림 4와 같다. 0~7까지의 범위를 갖는 두개의 "small-integer" 형 입력이 4비트로 연산기로 합성된 후 10비트 출력의 하위 4비트로 출력되는 것을 볼 수 있다.

다음은 입출력 타입을 다시 한번 설명하고 넘어가겠습니다. 디지털 신호가 '1' 과 '0' 만으로 회로를 다룰 것인지 아니면 하이 임피던스(Z)도 있고 풀업(H), 풀다운(L), 그리고 합성과 시뮬레이션의 상태를 나타내기 위해서 Don't care(-), Unknown(U), 버스 충돌 등에 의한 에러 상태(X) 등도 구분할 것인지 결정해야 한다. 전자의 경우 bit 타입이라고 하고 후자의 경우 std_logic 이라고 하기로 규정했다. VHDL 언어 자체적으로는 std_logic에 대한 규정은 없다. 다만 IEEE 1076을 정할 때 std_logic_1164 라는 것을 정해 놓고 이곳에 bit라는 것은 1 과 0으로 규정한다거나 std_logic은 1, 0, Z, H, L, X 등등으로 규정한다고 정했다.



위의 그림이 입출력 포트를 가장 잘 설명한 그림이라고 생각됩니다. IN과 OUT 포트는 잘 이해할 수 있을 것이라 생각됩니다. BUFFER는 INOUT과 같은 입출력 포트로서 assign의 소스(source)측 또는 목적지(destination)측에 쓸 수 있다. INOUT 모드와 다른 점은 단일 할당문 내에서 소스와 목적지측 모두 동시에 사용할 수 있다는 것이다. 이는 BUFFER 모드에 이미 F/F를 내포하고 있다는 의미를 갖는다. 그림을 자세히 보면 INOUT과 BUFFER의 차이점을 알 수 있을 것입니다.

Signal과 Variable 그리고 Constant

우선 Constant에 대해서 먼저 설명하겠습니다. Signal과 Variable의 차이점은 좀 미세한 부분이 많아서 둘을 비교해 가면서 설명하는 것이 좋을 것이라 생각됩니다.

constant는 초기에 선언한 상수의 값을 유지하는데 사용되며 VHDL의 문장 작성에 도움을 주어 쉽게 수정하거나 확장하는데 주로 이용한다. constant의 대입 기호는 ' := '를 사용하고 있다. Variable과 같은 기호를 사용합니다. constant의 초기 값은 즉시 대입되며 한 번 대입된 값은 바꿀 수 없다. 예를 들자면 Pi 값을 먼저 설정해 두고 계속 사용하면 편리할 것이다. 사용 방법은 간단하다.

```
constant p1 : integer := 314;
constant length : integer := 345;
constant pi : real := 3.1415119;
```

제가 가장 어렵다고 생각되는 부분이 Signal과 Variable입니다. 이 부분 때문에 너무 많은 고생을 한 기억이 납니다. 차이점을 잘 알고 있어야 합니다.

우선 Signal에 대해서 알아봅시다. signal은 VHDL 합성 시에 Wire(선)로 구성되며, 각 부품의 연결에 사용되는 외적 변수이다. 세 가지로 사용됩니다.

우선 포트 signal로 사용됩니다. 포트를 정의할 때 사용됩니다.

```
port ( a, b : in bit;
       z    : out bit);
```

여기에서 a, b, z는 port signal로 사용되었습니다. 입출력의 정의가 signal로 정의됩니다.

```
architecture rtl of and_1 is
begin
   z <= a and b;
end rtl;
```

위의 예제에서 보면 알 수 있듯이 포트 시그널로 정의가 되어 있기 때문에
z <= a and b; 라는 표현이 가능합니다. 만약 z := a and b; 와 같이 variable assign 한다면 에러가 날 것입니다.

두 번째로 signal을 정의하고 이것을 초기 치로 설정할 수 있습니다. 사용 방법은 열거형으로 정의하면 됩니다. 그러나 Assign의 방법이 조금 다릅니다.

```
signal temp : std_logic_vector(3 downto 0) := "1100";
```

세 번째로 일반적인 signal의 표현입니다. 저는 포트 시그널과 구별하기 위해서 이것을 내부 signal이라는 표현을 사용합니다. (그러나 포트 시그널과 내부 시그널을 구별 없이 signal이라고 하는 것이 보통입니다.)
내부 시그널이라고 표현하는 이유는 a, b, c, d라는 입력이 있고 출력이 z라는 포트 선언이 있습니다. 그러나 a와 b를 더해서 e라는 값에 저장을 하고 c와 d를 더해서 f라는 값에 저장을 합니다. e와 f를 더해서 z라는 값에 출력을 하는 회로가 있다고 합시다.

```
-- 필요 없는 부분은 삭제를 했습니다.
entity adder is
port ( a, b, c, d : in std_logic_vector(3 downto 0);
       z          : out std_logic_vector(3 downto 0));
end adder;

architecture temp of adder is

signal e : std_logic_vector(3 downto 0);
signal f : std_logic_vector(3 downto 0);

begin
   e <= a + b;
   f <= c + d;
   z <= e + f;
end temp;
```

위 부분에서 e와 f가 내부 시그널로 표현되었습니다. 그리고 Process 문에서도 사용 가능합니다. 다만 그 시점에서 값이 바로 대입되지 않고 end process를 만나면 완료되어 값이 결정됩니다. 위의 예제는 그 시점에서 바로 값이 대입된다는 것을 알 수 있습니다.
왜 내부 시그널이라는 개념을 사용하는지 알겠습니까? 그리고 항상 내부 시그널은 **architecture와 begin 사이에 들어가야 합니다. 이와 반대로 variable은 process 문안에서만 사용할 수 있고 process 문과 begin 사이에 들어가야 합니다.**

Variable은 process나 부프로그램(function과 procedure)에서만 사용되며 variable이 지니는 값도 process나 부프로그램 내에서만 유효한 내적 변수이다. variable은 signal과 같이 VHDL 합성 시에 바로 선으로 구현되는 것이 아니며, 중간 연산 단계에 주로 사용된다. variable에서 사용되는 assign 기호는 ' := '이고, 즉시(immediately)라는 의미를 가진다. signal과 비슷한 선언 방식을 사용한다.

```
-- 입력이 a와 b이고 출력이 c와 d일 때
variable tmp1, tmp2 : std_logic;
signal tmp3 : std_logic;
..
tmp1 := '1';
tmp2 := a and b;
..
c <= tmp1;
d <= tmp2;
```

그 값을 돌려줄 때 **signal <= variable;** 라는 방식을 사용한다.

다음 예제는 3입력 and 게이트의 구현이다. 좀 차이점을 실감할 수 있을 것이다.

ex1) 3입력 and gate

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

entity and_3 is
port ( a, b, c : in bit;
         z         : out bit);
end and_3;

architecture temp of and_3 is
begin
  process(a, b, c)
  variable temp : bit;
  begin
     temp := '1';
     temp := a and temp;
     temp := b and temp;
     temp := c and temp;
     z <= temp;
  end process;
end temp;
```

architecture 내에 process와 begin 사이에 variable temp를 선언하며, temp는 중간 단계의 연산 결과를 잠시 보관한다. 그러나 시그널처럼 즉시 그 값을 temp에 기록하지는 않는다. 이것이 signal과 variable의 미세한 차이이다. 다음 예제를 가지고 그 결과 값을 확인해 보기를 바란다. signal로 표현한 3입력 and 게이트이다. 분명히 에러 또는 warning을 보게 될 것이다.

ex2)

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

entity and_3_1 is
port ( a, b, c : in bit;
          z          : out bit);
end and_3_1;

architecture temp of and_3_1 is
signal temp : bit;
begin
  process(a, b, c)
  begin
     temp <= '1';
     temp <= a and temp;
     temp <= b and temp;
     temp <= c and temp;
     z <= temp;
  end process;
end temp;
```

이 두 예제의 차이점을 이해했다면 어느 정도 signal과 variable의 차이점을 이해할 수 있을 것이다.

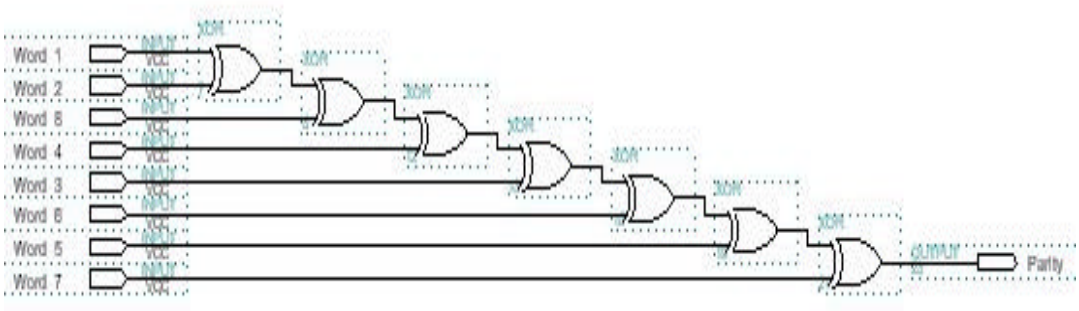
다음 예제를 구별할 수 있다면 충분히 다음으로 넘어갈 수 있을 것입니다. 이번 예제는 8비트 Parity 체크 회로입니다. 패러티를 검사하는 회로에서 signal과 variable로 정의되었을 때와 다른 회로도를 보입니다. 그 차이점을 충분히 이해할 수 있었으면 하는 바람입니다.
그리고 for loop 구문이 사용됩니다. 어떻게 사용되는지 자세히 보고 다음에 활용해 보세요.

ex3) 정상적인 Parity 체크 회로

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

entity parity is
port ( word : in bit_vector(7 downto 0);
         parity : out bit);
end parity;

architecture rtl of parity is
begin
process
variable result : bit;
begin
    result := '0';

    for i in 0 to 7 loop
        result := result xor word(i);
    end loop;

    parity <= result;
end process;
end rtl;
```

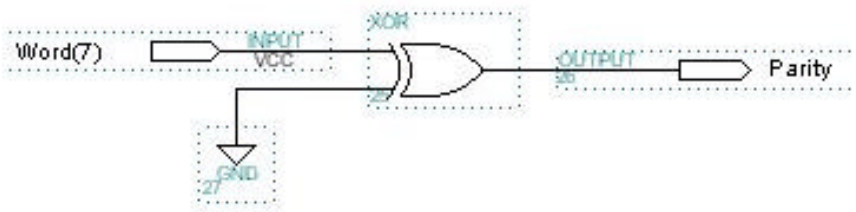
for loop 문은 별로 어려움 없이 이해할 수 있을 것입니다. 범위 시작에서 범위 끝까지 증가하면서 수행하게 된다.

```
for 변수 in 범위 시작 to 범위끝 loop
    statement;
end loop;
```



ex4)

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

entity parity is
port ( word : in bit_vector(7 downto 0);
       parity : out bit);
end parity;

architecture rtl of parity is
signal result : bit;
begin
   process
   begin
      result <= '0';

      for i in 0 to 7 loop
         result <= result xor word(i);
      end loop;

      parity <= result;
   end process;
end rtl;
```



Counter(Up/Down, Ring, Johnson, Gray Counter)

순차 회로의 대표적인 경우는 Counter가 있습니다. 계수기(Counter)가 무엇인지는 알겠지요. Clock의 입력을 받아 하나씩 증가하든지 감소하는 기능을 가진 회로를 말합니다.
간단한 기능을 가지고 있지만 대단히 활용 범위가 넓습니다. 카운터를 잘 활용하면 나중에 전자시계가 간단하게 설계할 수 있을 것입니다. 몇 비트든 상관없이 마음대로 설계할 수 있을 것입니다.

ex1)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity upcount is
port ( clk, reset : in std_logic;
         count_out  : out std_logic_vector(5 downto 0));
end upcount;

architecture rtl of upcount is
signal tmp : std_logic_vector(5 downto 0);
begin
   count_out <= tmp;
   process(clk, reset)
   begin
      if (reset='1') then
         tmp <= "000000";
      elsif (clk='1' and clk' event) then
         tmp <= tmp + '1';
      end if;
  end process;
end rtl;
```

우선 내부 시그널의 위치를 살펴봅시다. architecture와 begin 사이에 들어갑니다. 그리고 모든 회로는 Concurrent하게 동작한다고 했죠. 그러니 **count_out <= tmp;** 와 Process 문과는 두 개의 구문이 concurrent하게 동작합니다. Process 구문에서 마지막으로 tmp의 값이 결정되어 나옵니다. 그러면 count_out의 값이 결정됩니다.

process 내의 내용을 조금 살펴봅시다. reset이 1이라면 tmp의 값이 000000으로 결정됩니다. elsif 구문에서 reset이 0이고, 클럭이 상승에지에서 1 증가된 값을 가지게 됩니다.

그리고 if 구문에 대해서 알아봅시다. if 구문이 사용된 예제는 앞에서 많이 보았죠.

```
if  ...  then    ....    ;
end if;
```

위의 경우는 플립플롭의 경우라면 레치가 될 가능성이 많군요. 약간 불안정한 형태입니다.

```
if  ...    then  ....  ;
else  ....    ;
end if;
```

가장 대표적인 형태의 if 구문입니다.

```
if    ...   then   ....   ;
```

```
elsif    ....        ;
end if;
```
위의 예제의 경우가 사용된 형태입니다.
```
if    ...    then    ....    ;
elsif    ....        ;
....
else    ....        ;
end if;
```
조금 복잡한 형태의 구문입니다.

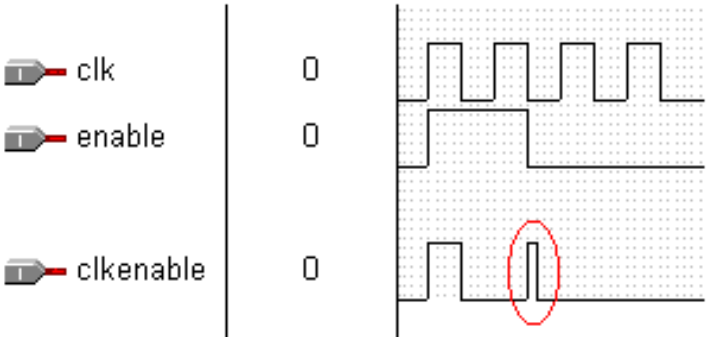
ex2)
```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity upcount2 is
port ( clk, reset, enable, load : in std_logic;
          databus     : in std_logic_vector(5 downto 0);
          count_out   : out std_logic_vector(5 downto 0));
end upcount2;

architecture rtl of upcount2 is
signal tmp : std_logic_vector(5 downto 0);
signal clkenable : std_logic;
begin
    count_out <= tmp;
    clkenable <= clk and enable;
    process(clkenable, reset, load)
    begin
        if (reset='1') then
            tmp <= "000000";
        elsif (clkenable='1' and clkenable' event) then
            if (load='1') then
                tmp <= databus;
            else
                tmp <= tmp + '1';
            end if;
        end if;
  end process;
end rtl;
```

load가 1이면 databus의 값을 로드하게 되어 있군요. 약간 복잡하게 보입니다만 그렇게 복잡한 형태는 아닙니다. 다만 clkenable라는 이상한 신호를 발견했을 것입니다.

clkenable <= clk and enable; 클럭 신호와 enable 신호의 and를 한 것이 clkenable 신호입니다. 전문적인 용어로 **gated clock**이라고 합니다. 이 신호를 사용할 때는 Timing을 고려해야 합니다. 그렇지 않으면 glitch가 발생할 가능성이 너무 많습니다.

그럼 왜 gated clock을 사용할까요. Synchronous 회로의 단점은 무엇일까요. 아마도 파워 소모가 너무 많다는 것입니다.

다음 Asynchronous 회로를 살펴봅시다.



clkenable은 전력 소모가 적지만 위의 그림과 같은 timing 문제를 반드시 고려해야 합니다. 그렇지 않으면 위의 그림과 같은 glitch가 발생하게 될 것입니다.

예제2의 경우 별로 설명할 것이 없네요. if 구문 속에 또 다른 if 구문이 들어갈 수 있다는 것에 대해서는 유심히 보세요. if 구문 속에 들어갈 수 있는 if 문의 제한은 없습니다. 그렇지만 자기가 그것을 구별하는 데는 문제가 있을 것입니다. 그러니 적당한 탭을 사용하는 것이 좋을 것이라 생각됩니다. 보기에도 깔끔하고 구분하기도 쉽게 말이죠.

ex3)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;
entity upcount3 is
port ( clk, reset, enable, load : in std_logic;
          databus     : in std_logic_vector(5 downto 0);
          count_out   : out std_logic_vector(5 downto 0));
end upcount3;

architecture rtl of upcount3 is
signal tmp : std_logic_vector(5 downto 0);
signal clkenable : std_logic;
constant lastvalue : std_logic_vector := "111011";
constant initialvalue : std_logic_vector := "000000";
begin
    count_out <= tmp;
    clkenable <= clk and enable;
    process(clkenable, reset, load)
    begin
        if (reset='1') then
            tmp <= initialvalue;
        elsif (clkenable='1' and clkenable' event) then
            if (load='1') then
                tmp <= databus;
            elsif (tmp=lastvalue) then
                tmp <= initialvalue;

            else
                tmp <= tmp + '1';
            end if;
        end if;
  end process;
end rtl;
```

초기 치와 마지막 값을 constant를 통해 설정해 두었다는 것이 틀리다는 것을 알 수 있겠죠. constant의 위치도 유심히 보시고, 초기 치가 0이고 마지막 치가 59입니다. 이 회로를 어떻게 변형하면 시계가 될 수 있는 지도 한 번 생각해 보십시오.

ex4)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity downcount is
port ( clk, reset, enable, load : in std_logic;
          databus     : in std_logic_vector(5 downto 0);
          count_out   : out std_logic_vector(5 downto 0));
end downcount;

architecture rtl of downcount is
signal tmp : std_logic_vector(5 downto 0);
signal clkenable : std_logic;
constant zero : std_logic_vector := "000000";
begin
    count_out <= tmp;
    clkenable <= clk and enable;
    process(clkenable, reset, load)
    begin
        if (reset='1') then
            tmp <= zero;
        elsif (clkenable='1' and clkenable' event) then
            if (load='1' or tmp=zero) then
                tmp <= databus;
            else
                tmp <= tmp - '1';
            end if;
        end if;
  end process;
end rtl;
```

위에서 했던 예제와는 반대로 다운 카운터입니다. '+' 대신에 '-' 가 사용되었다는 것이 다르죠. 하나의 예제를 정확하게 알 수 있다면 비슷한 응용 회로는 이해하기도 쉽고, 변형하기도 쉽죠.

ex5)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity updown_count is
port ( clk, reset, up_down, enable, load : in std_logic;
         databus     : in std_logic_vector(5 downto 0);
         count_out   : out std_logic_vector(5 downto 0));
end updown_count;

architecture rtl of updown_count is
signal tmp : std_logic_vector(5 downto 0);
signal clkenable : std_logic;
constant zero : std_logic_vector := "000000";
begin
    count_out <= tmp;
    clkenable <= clk and enable;
    process(clkenable, reset, load)
    begin
        if (reset='1') then
            tmp <= zero;
        elsif (clkenable='1' and clkenable' event) then
            if (load='1') then
                tmp <= databus;
            elsif (up_down='1') then
                tmp <= tmp + '1';
            else
                tmp <= tmp - '1';
            end if;
        end if;
  end process;
end rtl;
```

업다운 카운터는 up_down이라는 새로운 입력이 하나 있죠. up_down 입력이 이면 업카운터를 나타내고 0이면 다운 카운터의 기능을 수행합니다. 종합적인 기능을 수행하는 회로입니다. 문법적인 기능은 천천히 읽어보시면 쉽게 이해가 될 것이라 판단됩니다.

이외에도 많은 카운터의 종류가 있습니다. 그 중에 몇 가지만 더 살펴보겠습니다.

ex6)

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

entity johncnt is
port ( clk, reset : in std_logic;
          count_out  : out std_logic_vector(4 downto 0));
end johncnt;

architecture rtl of johncnt is
signal tmp : std_logic_vector(4 downto 0);
begin
    count_out <= tmp;
    process(clk, reset)
    begin
        if (reset='1') then
            tmp <= "00000";
        elsif (clk='0' and clk' event) then
            tmp <= tmp(3 downto 0) & not(tmp(4));
        end if;
  end process;
end rtl;
```

위의 예제는 Johnson Counter를 기술했습니다.

tmp <= tmp(3 downto 0) & not(tmp(4)); 여기에서는 몇 가지 설명을 드리고 갈 것이 있네요. tmp는 5비트입니다. tmp의 최상위 비트는 tmp(4)가 됩니다. 비트를 구분할 수 있다는 말입니다. not는 그 비트의 역의 값을 취합니다. 0이면 1의 값을 가진다는 말입니다. tmp가 5비트이므로 오른쪽의 tmp 비트도 당연히 5비트가 되어야 합니다. **&**라는 기호를 사용했습니다. & 표시는 비트를 합한다는 표시입니다. tmp(3 downto 0)가 4비트이고 tmp(4)가 1비트이므로 합이 5비트가 되는 것입니다. & 표시는 쉬프트 레지스터를 구현할 때 아주 유용한 표현입니다. 쉬프트 레지스터를 구현할 때 참 편리합니다. 위의 예제를 보면 알 수 있듯이 카운터를 조금만 변형하면 아주 유용한 쉬프트 레지스터를 구현할 수 있습니다.

다음은 Gray Counter입니다. gray code 식으로 구현한 카운터입니다. 예제7은 동작 적으로 기술된 Gray Counter이고 예제8은 행위 적으로 기술된 카운터입니다. 비교해 보시면 이해할 수 있을 것입니다.

ex7)

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

entity graycnt is
port ( clk, reset : in std_logic;
         count_out  : out std_logic_vector(2 downto 0));
end graycnt;

architecture rtl of graycnt is
signal tmp : std_logic_vector(2 downto 0);
constant zero : std_logic_vector := "000";
begin
    count_out <= tmp;
    process(clk, reset)
    begin
        if (reset='1') then
            tmp <= zero;
        elsif (clk='0' and clk' event) then
            case tmp is
                when "000" => tmp <= "001";
                when "001" => tmp <= "011";
                when "010" => tmp <= "110";
                when "011" => tmp <= "010";
                when "100" => tmp <= "000";
                when "101" => tmp <= "100";
                when "110" => tmp <= "111";
                when "111" => tmp <= "101";
                when others => tmp <= "000";
            end case;
        end if;
  end process;
end rtl;
```

ex8)

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

entity graycnt2 is
port ( clk, reset : in std_logic;
       count_out  : out std_logic_vector(2 downto 0));
end graycnt2;

architecture rtl of graycnt2 is
signal tmp : std_logic_vector(2 downto 0);
alias c : std_logic is tmp(2);
alias b : std_logic is tmp(1);
alias a : std_logic is tmp(0);
begin
   count_out <= tmp;
   process(clk, reset)
   begin
      if (reset='1') then
         c <= '0';
         b <= '0';
         a <= '0';
      elsif (clk='0' and clk' event) then
         c <= (not(a) and b) or (not(c) and not(b));
      end if;
  end process;
end rtl;
```

위의 예제에서는 alias 라는 재미있는 기능을 사용했네요. tmp(2) 대신에 c를 사용한다는 말입니다. Unix나 Linux에서 사용하는 alias의 기능과 같습니다. alias 기능을 사용하기 싫다면 c대신에 tmp(2)라고 입력하면 됩니다. 회로 구분을 위해 포트 이름이 길다면 alias를 사용하는 것도 좋은 것 같네요. 쓰기 싫다면 쓰지 않으면 됩니다.

회로를 설계하는데 있어 **동작 적으로** 기술하든지 아니면 **행위 적으로** 기술하든지 아니면 **구조적으로** 기술하는 것은 설계자의 마음입니다. 어떤 식으로든지 설계하면 됩니다.

좀 구체적으로 말하면 VHDL의 표현 방법은 설계의 의도와 용도에 따라 방법을 선택하거나 혼합해서 사용할 수 있습니다. 부울대수, RTL, 논리 연산자를 이용해서 자료 흐름 적으로 기술하는 방법을 자료 흐름 표현(data flow description) 방법이라 하고, 시스템의 동작을 알고리즘으로 기술하는 방법을 동작적표현 방법이라 부릅니다. 동작적표현 방법은 process 문을 이용하여 시스템의 동작을 편리하게 표현 할 수 있습니다. 구조적 표현 방법은 하드웨어의 Component의 상호 연결로 구성되며, 게이트 레벨에서 서브 시스템 레벨로 구성된 컴

포넌트의 Interface를 가능하게 한다. 하드웨어 표현에 매우 가까운 표현 방식이며 계층구조의 하드웨어 설계시에 주로 이용된다. 쉽게 말하면 port map을 이용하는 방식입니다.

ex9)

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

entity ringcnt is
port ( clk, reset : in std_logic;
         count_out  : out std_logic_vector(3 downto 0));
end ringcnt;

architecture rtl of ringcnt is
signal tmp : std_logic_vector(3 downto 0);
begin
    count_out <= tmp;
    process(clk, reset)
    begin
        if (reset='0') then
            tmp <= "0001";
        elsif (clk='0' and clk' event) then
            tmp <= tmp(2 downto 0) & tmp(3);
        end if;
  end process;
end rtl;
```

위의 예제는 간단한 기능의 4비트 Ring Counter입니다. 그냥 쉽게 읽어보세요.

다음 예제는 0에서 999까지 카운터가 가능한 회로입니다. 이 예제는 조금 중요하다고 생각됩니다. 많은 응용이 가능한 회로라고 생각이 드니까요.

ex10)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity count999 is
port ( clk, reset : in std_logic;
         out_3  : out std_logic_vector(3 downto 0);
         out_2   : out std_logic_vector(3 downto 0);
         out_1    : out std_logic_vector(3 downto 0));
end count999;
```

```
architecture rtl of count999 is
signal tmp_d100 : std_logic_vector(3 downto 0);
signal tmp_d10  : std_logic_vector(3 downto 0);
signal tmp_d1   : std_logic_vector(3 downto 0);
constant zero  : std_logic_vector := "0000";
constant nine  : std_logic_vector := "1001";
begin
    out_3  <= tmp_d100;
    out_2  <= tmp_d10;
    out_1  <= tmp_d1;
    process(clk, reset)
    begin
        if (reset='1') then
            tmp_d100 <= zero;
            tmp_d10  <= zero;
            tmp_d1   <= zero;
        elsif (clk='1' and clk' event) then
            if (tmp_d1=nine) then
                tmp_d1 <= zero;
                if (tmp_d10=nine) then
                    tmp_d10 <= zero;
                    if (tmp_d100=nine) then
                        tmp_d100 <= zero;
                    else
                        tmp_d100 <= tmp_d100 + '1';
                    end if;
                else
                    tmp_d10 <= tmp_d10 + '1';
                    tmp_d100 <= tmp_d100;
                end if;
            else
                tmp_d1 <= tmp_d1 + '1';
                tmp_d10 <= tmp_d10;
                tmp_d100 <= tmp_d100;
            end if;
        end if;
    end process;
end rtl;
```

만약 이 회로를 구현한다면 세그먼트 3개가 필요할 것입니다. 잘 생각하시면 전자 시계에 대한 답도 나올 것입니다.

설계시 고려사항 4가지, 설계 계층구조(Design hierarchy) Modeling and synthesis

제 강좌를 계속 보시는 분은 적어도 한번은 컴파일 과정을 통해 에러를 체크하며 시뮬레이션 과정까지 한번은 실행해 보았을 것입니다. 그렇지 않으면 오늘 강좌를 이해하기가 좀 힘들지 않을까 생각됩니다.

오늘은 **What is VHDL?** 오늘은 조금 무거운 주제입니다. 처음에 시작하지 않은 이유는 어느 정도의 이해가 힘들기 때문에 뒤에 하는 것입니다. 이제부터는 여러분은 설계자의 입장에서 모든 것을 이해해야 합니다. 조금 복잡한 회로는 아직 시작하지 않았지만 여러분들이 잘 알고 있어야 하는 중요한 부분입니다.

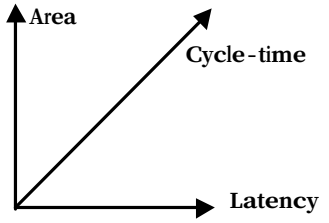
Time to Market. 이런 말을 들어본 적이 있습니까? 이 말은 설계자들이 항상 염두에 두고 있어야 하는 말입니다. 빠르게 변화하는 하드웨어 시장에서 아주 중요한 말입니다.

하드웨어 설계에서 시장까지 나오는 시간을 말합니다. 누가 먼저 얼마나 빨리 설계해서 시장까지 나오는가 이것이 중요합니다. 예를 들자면 지금의 펜티엄 프로세서가 있습니다. 그렇지만 내가 다시 설계해서 성능과 가격을 낮춘 설계를 한다고 합시다. 그렇지만 이 설계는 벌써 의미가 없다는 말입니다. 지금의 펜티엄 프로세서를 얼마나 빨리 생산하는가 하는 것이 중요하다는 말입니다. **설계에서 시장까지.** 요즘은 TR의 개수가 중요한 것이 아닙니다. 옛날에는 TR의 개당 가격이 비싸지만 요즘은 그렇게 중요한 요소가 아니라는 말입니다. 누가 빨리 시장에 선보이는가 이것이 중요한 의미입니다.

다음은 설계자가 고려해야 할 요소를 생각해 봅시다.

1. Performance :
   - Delay and cycle-time
   - Latency
   - Throughput (for pipeline application)

자기가 만든 회로의 성능이 다른 사람의 회로보다 우수하고 품질이 좋아야 한다는 것은 상식입니다. Delay적인 요소도 상당히 중요한 요소에 포함됩니다. 그리고 면적(Area)과 Delay는 반비례하는 관계가 있습니다. 면적이 커지면 Delay는 감소하고 면적이 적어지면 Delay는 증가합니다. 입력의 시작에서 출력의 끝까지의 시간(Latency), 출력 속도(Throughput) 등을 다각적인 방향에서 설계해야 합니다.



이런 비례 관계를 가지고 있습니다. 어느 하나만을 중점적으로 설계하는 것은 별로 좋은 생각이 아닙니다.

2. Area (yield and packaging cost)

면적이 커진다면 칩의 개수가 늘어나는 것은 사실입니다. 그래서 무작정 회로를 크게 설계할 수는 없습니다. 회로의 크기를 줄일 수 있다면 가능한 최대로 줄이는 것은 좋습니다. 그렇지만 면적만을 최우선으로 줄이는 것은 좋지 않습니다. 칩의 단가를 줄이는 것도 중요하지만 성능 적인 요소뿐만 아니라 다른 요소도 생각해야 합니다.

3. Testability

아주 큰 회로를 설계했다면 그것을 테스트하는 데에도 상당히 많은 비용이 든다는 사실입니다. 하나의 회로를 테스트하는 데 시간과 비용을 생각하지 않을 수 없습니다. 자기가 VHDL로 구현한 회로를 스키메틱으로 보았을 때 약간 추가된 부분을 볼 수 있을 것입니다. 그것은 Synthesis할 때 회로를 테스트하기 쉽도록 약간의 추가적인 회로가 포함됩니다. 이상한 것은 절대 아닙니다.

4. Power

회로가 복잡해지면 복잡해질수록 파워를 줄여야 합니다. 마이크로 프로세서 데이터 북을 보면 쉽게 확인할 수 있을 것입니다. 5V에서 3.5V로 이것이 다시 3.3V로 계속해서 파워를 줄이고 있습니다. 파워 소모를 줄이는 문제입니다. 점점 저 파워 회로가 주를 이루고 있습니다. 이것은 무시할 수 없는 요소입니다.

이 네 가지 요소를 복합적으로 이해하면서 네 가지 요소를 모두 충족할 수 있는 회로를 설계해야 합니다. 한 가지라도 무시하면 안 됩니다. 추가적인 요소로 생각하면 Coding style도 고려해서 설계해야 합니다. 그리고 Software적인 요소에서 소프트웨어로 처리할 수 있는 부분과 처리할 수 없는 구분을 생각해서 설계해야 합니다.

자 그럼 우리는 왜 VHDL을 사용해서 설계를 해야 할까요? 아니면 왜 VHDL을 공부하고 있습니까? **ASIC과 VHDL** 이라는 이상한 주제. 이것을 한 번 생각해 봅시다. 왜 시작해야 하는지를 역사적인 인식에서 다시 시작합시다. 이제는 설계자의 입장에서 생각해야 합니다.

VHDL(Very High Speed Integrated Circuit Hardware Description Language : VHSIC Hardware Description Language)은 공인된 표준 하드웨어 설계 언어이다.

70년대는 트랜지스터 레벨의 레이아웃(Layout) 편집기와 80년대부터 지금까지 사용하는 스키메틱(Schematic) 편집기가 등장하였다. 그런 이들 설계 기술은 대규모 회로 설계와 고도의 기능적 레벨의 회로 설계를 다루기에 적합하지 않았다. 예를 들면 Schematic으로 25,000 내지 30,000 개의 게이트를 설계하려면 250페이지 이상의 도면이 필요하다. 종이 값이 너무 많이 들겠죠. 이처럼 많은 도면을 그려야 하는 것은 물론이고 각 도면의 회로 복잡도를 조절하고 회로 도면간 배선 등을 관리하는 것은 명백히 기술적인 문제뿐만 아니라 개발비용 및 기간 등의 문제가 야기된다. 이러한 상황이 계속된다면, 대규모 회로를 개발하는데 있어 개발비용이 과대해지거나 개발 기간이 길어져 실현 불가능한 상태에 이르고 말 것이다. 스키메틱으로 구현하는데 5만 게이트가 한계라고 한다. 이러한 이유로 VHDL이 필요성이 절실하게 느껴진다.

VHDL은 초기에 하드웨어의 사양을 표준화된 방식으로 시술하는 문서화의 모델링(Modeling)을 위한 언어로 출발하였으며, 1980년대 후반에는 VHDL이 simulation에 의한 검정용 언어로 사용해야 한다는 여론에 의해 몇몇 VHDL simulator가 등장하게 되었다.

VHDL simulator가 등장한 시점에서 VHDL 관련 CAD 회사간에 VHDL의 표준화를 위해 1987년 IEEE에서 IEEE-1076이라는 표준을 만들어 공인하게 되었다. 그러나 이 시점까지도

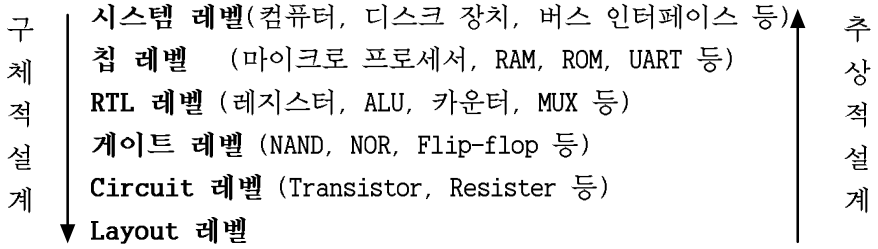
VHDL은 simulation용으로 사용되었지만, 하드웨어 합성(synthesis)을 위한 설계 언어로 사용되지는 않았다. 1990년 초에 VHDL 관련 소프트웨어 회사가 많이 등장하고 하드웨어 설계 기능을 가진 합성 Tool이 개발되었다. 그리고 1991년 IEEE에서 IEEE-1164의 표준을 만들었으며, simulation뿐만 아니라 합성을 위한 설계 기능을 갖춘 표준화된 언어로 VHDL을 인식하게 되었다. 즉 VHDL은 소규모 회로에서 대규모 시스템 설계에 이르기까지 문서화, 검정 및 설계(합성)를 위한 표준화된 설계 방식으로 사용되고 있다.

그럼 VHDL을 가지고 어떻게 설계할 것인가? 라는 질문은 합시다.

먼저 시스템 설계 요구 사양을 결정해야 한다. 무엇을 만들 것인지를 결정해야 한다. 간단한 예를 들자면 전자 시계를 만든다고 하자. 다음은 시스템 사양(System Specification)을 결정해야 한다. 시스템 사양은 시스템 모델링을 통해 이루어진다. 시스템 모델링은 하드웨어 구현에 염두를 두지 않고 시스템의 알고리즘을 표현하는 것이다. 시스템 모델링의 목적은 형식적 설계 사양으로서 사용할 수 있는 시뮬레이션 모델을 생성시키기 위한 것이다. 이 모델은 시뮬레이션 수행을 통해 시스템의 기능을 점검하고 확인하게 된다. 이러한 시스템의 사양은 설계자와 설계 의뢰 자의 요구 사항을 명확하게 하기 위해서도 필요하다. 시스템의 모델링은 최종 하드웨어 합성을 위해서 RTL(Register Transfer Logic) 설계로 변환시킨다.

이 변환의 최종 목표는 하드웨어 구현이며 RTL설계를 Netlist로 생성시키는 합성 과정을 수행하게 된다. 이 단계에서 VHDL은 특정 하드웨어 지원을 받게 되며 초기의 사양에서 제시한 칩상의 면적, 속도 및 Timing 요구에 대한 제한을 주게 되고 최초 계획된 시스템 사양의 요구에 만족하지 못하면 시스템의 모델링과 RTL변환 및 합성 과정을 다시 거치게 된다.

이제 **설계 계층구조(Design hierarchy)**에 대해서 알아봅시다.



ASIC 설계 프로세서에는 어느 레벨에서도 설계가 가능하다. 각 레벨간 표현 방식은 다르지만, 설계하려는 회로 자체에는 변함이 없어 서로 다른 레벨간 변환은 항상 가능하다.

시스템 레벨이 가장 상위 레벨이다. 가장 추상적인 레벨이기도 하지만, 설계가 가장 어렵다. 아직 VHDL Tool로 시스템 레벨을 정의하기는 어렵다. 상위 레벨의 알고리즘을 가지고 설계하기는 아직 단계적으로 어렵다는 뜻이다. 그러나 가능성은 있다.

하위 레벨로 갈수록 실제 회로 적인 문제를 다룰 수 있다. 정확한 타이밍 정보라든지 실질적인 회로 구현을 말한다. 지금 우리가 하고 있는 단계는 주로 레지스터 레벨의 설계를 주로 하고 있다. 큰 프로젝트를 설계한다면 상위 레벨부터 시작할 것이다. 이것이 Top-down 설계이다. 주로 **ASIC은 Top-down 설계 방식을 이용한다.** Layout 편집기나 Schematic으로 설계하는 방식은 Bottom-up 방식을 사용한다. 어떤 식으로 설계를 하는 것은 문제가 되지 않는다. 하나의 큰 프로젝트를 시작하면 시스템 레벨을 정의하고 그것의 Simulation이

완전하다면 다음 단계로 내려간다. 각각의 칩 레벨을 설계할 것이다. 이 단계에서는 분업을 이용할 것이다. 전문적인 분야로 나누어서 각각의 레벨을 설계하고 검증할 것이다.

그럼 간단하게 전체적인 설계 과정을 잠깐 살펴봅시다.

1. Design

1.1 Modeling
1.2 Synthesis & Optimization
1.3 Validation
Test1(Simulation) 설계에 대한 최종적인 테스트를 말합니다.

↓

2. Fabrication : 공장에서의 제조 과정을 말합니다. 줄여서 FAB.이라고 하지요.

2.1 Mask Fabrication
2.2 Wafer Fabrication
Test2 : 제조 과정에서의 테스트를 합니다.

↓

3. Packaging

3.1 Slicing
3.2 Packaging
Test3 : 칩 속에 Packaging을 해야 합니다. 그 과정에서의 테스트.

↓

4. Testing : 실제적인 칩 테스트를 Emulation이라고 합니다.(이 비용도 고려해야 함)

위 과정 중에서 1. Design의 과정이 우리의 과정입니다. FAB에 관계된 것을 알고 있다면 설계에 많은 도움을 줄 것이라 생각합니다. 시간이 가능하다면 공부해 보세요. 혼자서는 하기가 조금 힘들다는 단점이 있지만 가능하다면 집적회로 공정 기술에 대해서 공부하는 것도 도움이 되겠죠. 저도 집적회로에 많은 시간을 투자해 어느 정도의 기초적인 과정은 알고 있습니다. CMOS 제조 과정은 알고 있습니다. 그래서 layout 과정을 이해하는데 어느 정도 도움이 됩니다.

지금부터는 VHDL의 실제 설계에 관한 내용입니다. 위의 내용은 ASIC에 관계된 내용이지만 지금부터는 설계에 관계된 내용입니다. 어떻게 언어만으로 설계가 가능한지를 나타내는 과정입니다. 잘 알아두세요.

Modeling
Synthesis & Optimization
Validation

Modeling. 이것은 설계를 한다는 말입니다. 어떻게 설계할 것인지는 설계자에 달려 있죠. 그러나 어떤 방식의 설계가 있는지는 알고 있어야 하겠죠. 모든 것의 기초가 되는 것입니다. 어떻게 코딩할 것인지를 선택해야 합니다. Modeling abstraction의 종류에는 세 가지가 있습니다. 먼저 **Architectural Level**(추상적인 동작 모델링)에 대해서 먼저 알아봅시다.

이 레벨에서는 구체적인 하드웨어 시스템이 어떻게 동작해야 하는가를 기술한 최상위 모델링 기법을 제공한다. 이유는 각종 시스템 설계용 알고리즘들이 어떻게 동작하는가를 비교 평가한다든지 이들 알고리즘들이 예측 가능한 입력 조건하에서 제대로 작동하는지 알아보고 싶을 때, 그리고 구체적인 회로 설계 과정에서 하드웨어 구현 방식과 설계 사양 등을 함께 고려하고자 할 때 매우 유용하기 때문입니다. 이 레벨에서는 구체적인 정보보다는 시스템 레벨의 추상적인 기술 형태이다.

Architectural Level의 설계가 끝나면 다음으로 **Logic Level**(다시 말하면 RTL Level)의 설계를 합니다. 이 레벨은 논리 회로 설계시 상세한 블록 다이어그램과 같습니다. Block diagram에서 하드웨어 함수 블록은 그 블록에 연결된 입출력 신호 및 데이터 버스 등을 통해 알아낼 수 있다. 이 레벨에서는 Boolean 식에서 RTL에 이르기까지 다양한 방식으로 기술할 수 있습니다. 흔히 RTL 모델링을 **Data flow description**이라고 합니다.

다음 단계로는 **Geometrical Level**입니다. 이 레벨에서의 코딩 구문은 netlist 표현 방식과 유사하다. 게이트 또는 플립플롭이거나 동작적기술 레벨 내지는 RTL 코드로 기술된 Component들이 선으로 연결된 회로로 표현됩니다.

VHDL에서는 설계 과정에서 이들 세 가지 기술 형태를 임의로 혼합된 형태도 허용합니다.

다시 말하면 Behavioral View에서는 What to do?(무엇을 하는가?)의 기술이고 Structural View에서는 How to do?(어떻게 하는가?)의 기술입니다.

이 세 가지 레벨이 기술되면 이 각각의 과정을 Synthesis 과정을 수행합니다.

Architectural-level synthesis, Logic-level synthesis, Geometrical-level synthesis를 수행합니다. Architectural-level synthesis는 macroscopic structure를 결정합니다. macroscopic structure란 큰 빌딩 블록들을 말합니다. ALU나 ROM 같은 블록을 말합니다. 이 과정은 주로 Manual 설계를 할 때 주로 사용됩니다. Logic-level synthesis는 microscopic structure를 결정합니다. 로직 게이트의 상호 연결을 말합니다.

그럼 synthesis(합성)에 대해서 조금 자세히 알아봅시다. 소스를 코딩했다면 이것을 다시 게이트 레벨이나 RTL 레벨로 바꾸어 주어야 합니다. Synthesis란 번역만의 작업이 아닙니다. 소스 코드의 **Translation**(번역)뿐만 아니라 회로의 **Optimization**(최적화) 기능을 동시에 수행합니다. 합성을 하는 이유는 설계 생산성 향상을 도모할 수 있고(Schematic에 비해 10배정도), Time-to-market(시장 출하 시간)을 단축시킬 수 있고, 시스템 수준의 설계를 가능하게 해줍니다.

ASIC에서의 합성 단계를 보면 첫 번째로 상위 합성(High-level(Architectural) synthesis)을 거치고 다음으로 논리 합성(Logic synthesis)을 거칩니다. 다음 단계로는 테스트 합성(Test synthesis) 과정을 거치고, 그 다음으로 레이아웃 합성(Layout synthesis)을 거칩니다. 여기서 테스트 합성이란 회로 테스트를 위해서 넣어 주는 회로에 대한 합성을 말합니다.

위의 부분은 설계자가 반드시 알아야 하는 내용을 간략하게 정리한 내용입니다. 불필요한 부분은 많이 생략되었습니다. 위의 내용을 완전하게 이해하려면 더 많은 부분을 공부해야 할 것입니다. 제가 정리했지만 부족한 부분이 많다고 생각됩니다. 부족하다고 생각되는 부분은 자기가 스스로 찾아서 연구하십시오. 이 부분은 너무 방대한 내용이라서 정리하는데 많은 시간이 걸렸습니다. 이 부분을 이렇게 뒤에 설명하는 이유는 VHDL을 처음 시작하는 사람에게는 너무 어려운 내용이라서 이렇게 뒷부분에서 정리하고 있습니다.

Adder, 연산자, 루프문, Port map

오늘은 산술 연산 회로에 대해서 알아봅시다. 이제부터는 예제 중심으로 설명을 하고 간단한 기능을 생략하고 새로운 Syntax가 나오면 그것에 대해서 구체적으로 서술하겠습니다.

가장 보편적인 예제가 Full-Adder입니다.

전에는 예약어를 강하게 표시했지만 대부분의 툴은 syntax 칼라를 지원합니다. 그래서 예약어는 색깔이 다르게 표시될 것입니다. entity name이나 architecture name이 칼라로 표시되면 예약어를 사용하였다는 말이므로 반드시 에러가 표시될 것입니다.

ex1) 1-bit Full-Adder

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity fa is
port( a, b, cin : in std_logic;
      sum, cout : out std_logic);
end fa;

architecture rtl of fa is
signal s0, s1, s2 : std_logic;
begin
   s0 <= a xor b;
   s1 <= a and b;
   sum <= s0 xor cin;
   s2 <= s0 and cin;
   cout <= s1 or s2;
end rtl;
```

signal s0, s1, s2가 사용된 이유는 out port의 sum과 cout이 feedback을 허용하지 않기 때문입니다. 내부 시그널이라고 설명 드렸지요. architecture 구문 안에서는 어느 정도의 Delay이 가지고 있지만 병렬적으로 수행됩니다. a xor b의 결과 값이 s0인데 이 값이 sum의 입력으로 가기 위해서는 순차적으로 수행된 것이 아니냐는 질문은 하지 마십시오. 그러면 이 자체가 아무 의미가 없어집니다. 정교하게 a xor b의 결과 수행 시간이 있으므로 당연히 순차 회로라고 생각하면 안 됩니다. 그 결과 수행 시간이 아주 작은 값이라면 이것은 순차적으로 수행되는 것이 아니라 병렬적으로 수행됩니다. 논리 회로의 간단한 부분이므로 내용은 생략합니다. 모르시면 논리 회로 책을 참조하세요.

architecture 부분을 이렇게 고쳐도 시스템은 별 영향을 받지 않을 것입니다.

```
  sum <= a xor b xor cin;
  cout <= (a and b) or ((a xor b) and cin);
```

그렇지만 synthesis가 어떻게 합성이 되는가 하는 것이 문제입니다. 이런 간단한 예제 정도는 최적화 될 것입니다. 그런 간단한 예제는 문제없지만 복잡한 회로라면 최적화 되게 합성이 될지는 확신할 수 없습니다. 조금 좋은 Tool이라면 가능하겠지만 확신할 수 없는 부분입니다. 위의 예제는 비약적으로 바꾸었지만 실제로 그렇게 될 지도 모릅니다.

VHDL은 Truth table의 형태로도 코딩할 수 있습니다. 툴의 방법의 차이는 있지만 대부분의 툴이 지원하는 내용입니다. 심지어 상태도로도 코딩이 가능합니다.
위의 예제를 조금 바꾸어 봅시다. 진리표의 형태로 구현해 봅시다.

ex2)

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity fa1 is
port( a, b, cin : in std_logic;
      sum, cout : out std_logic);
end fa1;

architecture rtl of fa1 is
begin
   process(a, b, cin)
   variable temp : std_logic_vector(2 downto 0);
   begin
      temp := a & b & cin;
      case temp is
         when "000" => sum <= '0';
                       cout <= '0';
         when "001" | "010" | "100" => sum <= '1';
                       cout <= '0';
         when "011" | "101" | "110" => sum <= '0';
                       cout <= '1';
         when "111" => sum <= '1';
                       cout <= '1';
         when others => sum <= '0';
                       cout <= '0';
      end case;
   end process;
end rtl;
```

위의 예제는 상당히 재미있는 구조를 하고 있습니다. 모든 경우를 Truth table의 형태로

나타냈습니다. Adder의 모든 경우가 다 포함되어 있습니다. 그리고 ex1의 경우는 Signal을 사용한 반면 ex2는 variable을 선택해 사용했습니다. 여기서는 signal이나 variable을 써도 회로 설계에 큰 문제는 보이지 않습니다.

그럼 여기서 연산자에 대해서 살펴봅시다. 어느 것이 우선 순위가 높은지 알고 있어야 할 것입니다. 우선 순위를 잘못 정하면 자기 설계와 다른 형태로 바뀔 수 있습니다.

우선 순위가 제일 높은 것은 **기타 연산**자입니다. ****, ABS, not** 가 있습니다. ******는 지수 계산을 위한 연산자가 왼쪽 피연산자는 integer type이나 floating point type을 사용할 수 있지만 오른쪽 피연산자는 integer type만 가능합니다. 그 결과는 왼쪽 피연산자와 같은 type이 됩니다. **ABS**는 절댓값의 계산을 위하여 사용되며 모든 numeric 피연산자에 대해서 적용할 수 있습니다. **not**는 단항 연산자로서 피연산자의 논리 값을 참이면 거짓으로, 거짓이면 참으로 바꾸는 역할을 합니다.

다음은 **곱셈 연산**자입니다. ***, /, mod, rem**이 있습니다. 곱셈 연산자(*)와 나눗셈 연산자(/)는 왼쪽 및 오른쪽 피연산자는 같은 data type이어야 하며 integer type 또는 floating point type이 가능하고 그 결과도 피연산자와 같은 data type이 되어야 합니다. 나머지 계산(rem)과 모듈 계산(mod)의 경우에는 왼쪽 오른쪽 연산자 모두 integer type이어야 하고 그 결과도 integer type이고 그 결과도 integer type이 된다. A rem B는 A의 부호를 따르고 그 결과의 절댓값은 B의 절댓값을 따른다.

다음으로 우선 순위가 높은 것은 **단항 연산**자가 있다. +/-의 보호를 나타내는 연산자이다. 그렇게 설명은 필요하지 않다고 본다. 다음은 **덧셈 연산**자가 있다. 실제 연산을 수행하는 +/-와 &(접속 연산자)가 있다. 다음의 우선 순위가 있는 것은 **관계 연산**자가 있다. =, /=. >, <, <= 등이 있다. 우선 순위가 가장 낮은 것은 **논리 연산**자가 있다. or, and, nor, nand, xor, XNOR 등이 있다.

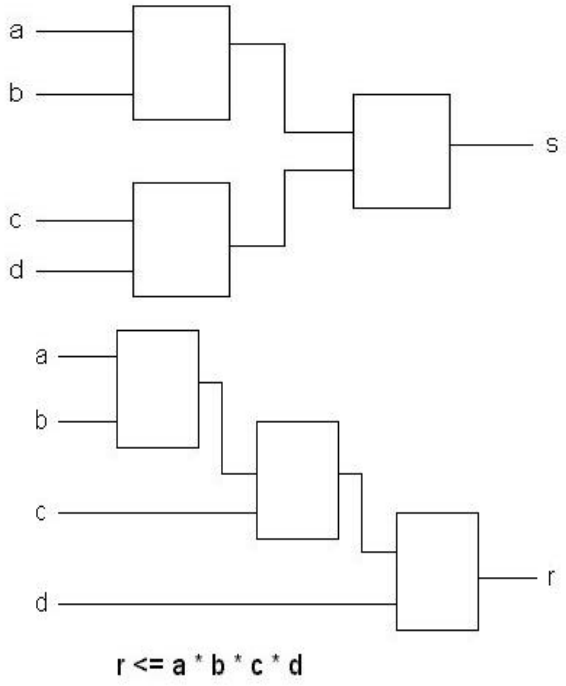
우선 순위가 높다는 말은 중요한 말이다. 다음 예제에서 잠시 비교해 보자.

옆에 있는 그림에서 알 수 있듯이
s <= (a * b) * (c * d)와
r <= a * b * c * d가 서로 다른 모양의 회로 합성을 보여 주고 있다.

이 말은 signal의 결합 방법에 따라
signal 전송 경로와 전파 속도가 다르다.

즉 s <= (a * b) * (c * d)가
r <= a * b * c * d 보다 신호의 전파 경로가 짧고 평행을 이루어 좋은 회로 합성의 형태가 된다.

우선 순위에 따라 그 회로가 자기가 원하지 않는 형태로 변화될 수 있다는 것을 알고 있어야 한다.



다음은 비트 수를 확장해서 4비트 Adder를 설계해 보자.

ex3) 4bit Adder

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

entity adder4 is
port( cin : in std_logic;
      a, b : in std_logic_vector(3 downto 0);
      sum : out std_logic_vector(3 downto 0);
      cout : out std_logic);
end adder4;

architecture rtl of adder4 is
begin
   process(cin, a, b)
   variable carry : std_logic;
   begin
      carry := cin;
      for i in 0 to 3 loop
         sum(i) <= a(i) xor b(i) xor carry;
         carry  := (carry and (a(i) xor b(i))) or (a(i) and b(i));
      end loop;
      cout <= carry;
   end process;
end rtl;
```

이번 예제에서는 signal과 variable을 동시에 사용했습니다. 이런 식의 예제 구성이 실제적으로 많이 사용되고 있습니다. 앞에서도 for loop 구문을 사용한 적이 있습니다.

그럼 Loop 문에 대해서 자세하게 알아봅시다. loop는 순차 처리문이며 element를 반복적으로 처리하기 위한 구문이다. loop 문은 for loop 형식과 while loop 형식과 단순 loop 형식이 있다.

for 루프변수 **in** 범위 시작 **to** 범위끝 **loop**
  순차처리문;
end loop;

while 조건 **loop**
  순차 처리문;
end loop;

```
loop
   순차 처리문;
end loop;
```

for loop 문은 루프변수가 1씩 증가 또는 감소하면서 최종 값에 도달할 때까지 loop문에 둘러싸인 순차 처리문을 반복 처리한다. 루프변수는 어떠한 객체로도 선언되지 않아야 되며 오직 for loop의 루프변수로만 사용되어야 한다. to가 사용되면 1씩 증가하면서 수행되고, downto가 사용되면 1씩 감소하면서 수행한다.

while (조건) loop문은 조건이 참이면 loop에 둘러싸인 순차 처리문을 반복 수행한다.while loop문의 조건은 반복 횟수가 명확히 결정되지 않으면 논리 합성 할 수 없으므로 주의를 요한다.

단순 loop 문은 무한히 반복되므로 loop를 빠져 나오기 위해서는 exit 문이 필요하며 while 문의 경우와 같이 반복 횟수가 정해지지 않을 경우에는 VHDL 논리 합성이 될 수 없다. 대부분의 VHDL 합성기는 while loop 문과 단순 loop 문을 지원하지 않는다.

```
for i in 0 to 3 loop
   sum(i) <= a(i) xor b(i) xor carry;
   carry  := (carry and (a(i) xor b(i))) or (a(i) and b(i));
end loop;
```

위의 예제를 while loop 바꾸어 보자.

```
while ( i <= 3) loop
   sum(i) <= a(i) xor b(i) xor carry;
   carry  := (carry and (a(i) xor b(i))) or (a(i) and b(i));
end loop;
```

그리고 설명한 안 한 것이 한가지 있는데 process 구문이나 for loop 문 앞에는 라벨을 붙일 수 있다는 점이다. 라벨은 나중에 혼동이 없도록 이름을 정하는 것이다. 합성 시에 아무런 문제가 되지 않는다. 간단한 회로에는 붙일 필요가 없지만 점점 회로가 복잡해지면 붙이는 것이 구분을 하기 쉬울 것이다.

예를 들자면

adder : process(cin, a, b)

이렇게 사용한다. 반복 루프문에도 사용할 수 있다.

그리고 process 괄호 안에 있는 것을 Sensitivity list라고 한다. 프로세서 문에서 입력의 변화가 있다면 반드시 괄호 안에 넣어 주어야 한다.

그럼 이것을 다르게 변형해서 설계를 해보자. 이번에는 회로 설계에 실제적으로 많이 사용되는 Port map을 사용할 것이다. component를 어떻게 이용하는지 잘 알아두세요.

ex4) 1bit 전가산기 설계

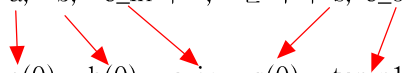
```
library ieee;
use ieee. std_logic_1164.all;

entity adder is
port ( a, b, c_in : in std_logic;
       s, c_out   : out std_logic);
end adder;

architecture rtl of adder is
begin
   s <= ((a xor b) xor c_in);
   c_out <= (a and b) or (a and c_in) or (b and c_in);
end rtl;
```

ex5)

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

entity adder4 is
port ( a, b    : in std_logic_vector(3 downto 0);
       c_in    : in std_logic;
       sum     : out std_logic_vector(3 downto 0);
       cout    : out std_logic);
end adder4;

architecture rtl of adder4 is

signal temp1, temp2, temp3 : std_logic;

component adder
port( a, b, c_in : in std_logic;
      s, c_out   : out std_logic);
end component;

begin
   u1 : adder port map(a(0), b(0), c_in, s(0), temp1);
   u2 : adder port map(a(1), b(1), temp1, s(1), temp2);
   u3 : adder port map(a(2), b(2), temp2, s(2), temp3);
   u4 : adder port map(a(3), b(3), temp3, s(3), c_out);
end rtl;
```

우선 Adder의 component를 이용하였습니다. 우선 Adder. vhd 파일이 저장되어 있어야 합니다. 같은 디렉토리에 저장되어 있거나 user Library에 저장되어 있어야 합니다. 아니면 이것을 Package화 시켜서 이용할 수 도 있습니다.

adder 회로에서 입력이 a, b, c_in이고, 출력이 s, c_out입니다.



u1 : adder port map (a(0), b(0), c_in, s(0), temp1);

순서대로 연결되어 있는 것을 알 수 있습니까? 이처럼 component의 형식 이름과 실제 이름을 결합할 때 port signal이 나열된 위치 순서대로 연결시키는 방법을 **위치 결합(positional association mapping)**이라고 합니다.

또 component의 형식 이름과 실제 이름을 결합할 때 port signal이 나열된 위치에 상관없이 각각의 형식 이름 => 실제 이름으로 결합시키는 방법을 **이름 결합(named association mapping)** 이라고 합니다. 직접 이름으로 연결되기 때문에 결합의 순서에는 상관없습니다.

Port map을 조금 쉽게 말하면 schematic의 signal을 wire로 연결시키는 것으로 보시면 됩니다.

중요한 점은 component 이름은 반드시 component 선언에서 선언된 이름이어야 합니다.

다음으로 Pipelined adder 회로에 대해서 알아봅시다. 생각보다 어려운 회로는 아닙니다.

ex6) Pipelined adder

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity pipeadder is
port ( clk    : in std_logic;
       inbus  : in std_logic_vector(7 downto 0);
       outbus : out std_logic_vector(8 downto 0));
end pipeadder;

architecture rtl of pipeadder is
signal reg_a : std_logic_vector(7 downto 0);
signal reg_b : std_logic_vector(8 downto 0);
begin
  outbus <= reg_b;
  process(clk)
  begin
    if (clk='1' and clk' event) then
      reg_a <= inbus;
      reg_b <= ('0' & reg_a) + ('0' & inbus);
    end if;
  end process;
end rtl;
```

signed package는 4칙 연산을 지원합니다. 입력이 8비트인데 비해 출력이 9비트입니다. 이유를 알겠습니까? 예를 들어 1비트 더하기 1비트를 하면 10이라는 2비트의 값을 가지는 것이 정상이죠. 이것을 다르게 표현한다면 결과 값은 8비트로 보내고 최상위 비트는 캐리로 보내는 것이 정상일 것입니다. ALU 회로 설계 때는 플래그 레지스터에 결과를 저장하는 것을 확인할 수 있을 것입니다. 그렇게 표현하는 것이 당연합니다. 지금은 간단한 연습용 회로이지 실제적인 상용화될 수 있는 회로는 아닙니다.

덧셈과 뺄셈은 종이 한 장의 차이입니다.

reg_b <= ('0' & reg_a) + ('0' & inbus);에서 ' + ' 대신에 ' - '를 사용하면 됩니다.
그럼 여기서 덧셈과 뺄셈을 동시에 수행할 수 있는 회로를 설계해 봅시다.

ex7) Pipelined adder

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity pipeadder is
port ( clk, sel : in std_logic;
        inbus  : in std_logic_vector(7 downto 0);
        outbus : out std_logic_vector(8 downto 0));
end pipeadder;

architecture rtl of pipeadder is
signal reg_a : std_logic_vector(7 downto 0);
signal reg_b : std_logic_vector(8 downto 0);
begin
  outbus <= reg_b;
  process(clk)
  begin
     if (clk='1' and clk' event) then
        if (sel='1') then
           reg_a <= inbus;
           reg_b <= ('0' & reg_a) + ('0' & inbus);
        else
           reg_a <= inbus;
           reg_b <= ('0' & reg_a) - ('0' & inbus);
        end if;
     end if;
  end process;
end rtl;
```

간단하게 sel이라는 입력 단자를 하나 추가함으로서 간단한 덧셈/뺄셈 회로를 설계할 수 있습니다. sel이 0이면 Adder의 기능을 수행하고 1이면 subtraction의 기능을 수행할 수 있습니다. VHDL을 어렵다고 생각하면 안 됩니다. 간단한 변환이 이렇게 다른 회로를 설계할 수 있습니다. 이것 말고도 다른 회로를 응용해 보시는 것은 VHDL에 좀더 가깝게 접근할 수 있을 것입니다.

다음은 4bit BCD(Binary Coded Decimal) 덧셈기를 설계합시다. 지금은 간단하게 4비트의 Adder에서 계산된 값을 BCD로 변환시키는 회로를 설계해 봅시다. 여기서 사용되는 예제는 대부분 소스가 공개되어 있는 것입니다. 제가 조금씩 변형하는 경우도 있지만 대부분은 소스를 그대로 사용하고 있습니다. 제가 새롭게 소스를 설계한다면 너무 시간이 많이 걸립니다. 그 점은 여러분도 마찬가지일 것입니다.

BCD 변환기에 대해서 한 번 생각해 봅시다. BCD 변환기의 특징은 무엇인가요?
0에서 9까지의 계산은 일반적인 adder와 같습니다. 문제는 a(10), b(11), c(12), d(13), e(14), f(15)의 계산입니다. a를 어떻게 처리하면 되겠습니까?

간단합니다. a는 10을 나타냅니다. 10은 이진수로 1010입니다. 쉽게 10은 영의 자리 숫자는 0이고 십의 자리는 1입니다. 십의 자리는 캐리를 말하면 됩니다. 그래서 a는 “0000”으로 처리하고 캐리를 1로 정의하면 됩니다. b, c, d, e, f도 이런 식으로 정의하면 됩니다.

그렇게 어렵지는 않을 것입니다. 오늘은 여기까지입니다. 소스는 밑 부분에 있습니다.

ex8)

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

entity bcdadder is
port ( s      : in std_logic_vector(3 downto 0);
       c_in   : in std_logic;
       s_out : out std_logic_vector(3 downto 0);
       c_out : out std_logic);
end bcdadder;

architecture rtl of bcdadder is
begin
  process(s, c_in)
  begin
     if (c_in='0') then
        if (s="1010") then
           s_out <= "0000";
           c_out <= '1';
        elsif (s="1011") then
           s_out <= "0001";
           c_out <= '1';
        elsif (s="1100") then
           s_out <= "0010";
           c_out <= '1';
        elsif (s="1101") then
           s_out <= "0011";
           c_out <= '1';
        elsif (s="1110") then
           s_out <= "0100";
           c_out <= '1';
        elsif (s="1111") then
           s_out <= "0101";
           c_out <= '1';
        else
           s_out <= s;
           c_out <= '0';
        end if;
     end if;
  end process;
end rtl;
```

4*4 multiplier, T Flipflop, 3-state buffer의 설계 레지스터의 설계와 Shift 레지스터

오늘 예제는 4*4 multiplier, T Flipflop, 3-state buffer의 설계 그리고 레지스터의 설계와 Shift 레지스터를 다루겠습니다. 곱셈기에 대해서는 조금 뒤에 자세하게 다루도록 하겠다.

먼저 이 예제는 가장 보편적인 예제입니다. 누구나 설계할 수 있는 형식을 취하고 있습니다. 그러나 여러분이 설계자의 입장이라면 이런 예제를 기준으로 좀더 복잡한 알고리즘을 가지고 회로의 크기가 작고 속도가 빠른 회로를 설계해야 할 것입니다. 이런 부분은 누가 가르쳐 주는 부분이 아닙니다. 자기 스스로 연구해야 하는 분야입니다.

ex1) 4×4 Multiplier

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_signed. all;

entity multi is
port ( a, b : in std_logic_vector(3 downto 0);
        prod : out std_logic_vector(7 downto 0));
end multi;

architecture rtl of multi is
signal p0,p1,p2,p3 : std_logic_vector(7 downto 0);
constant zero : std_logic_vector := "00000000";
begin
   process(a, b)
   begin
      if (b(0)='1') then p0 <= ("0000" & a);
      else p0 <= zero;
      end if;
      if (b(1)='1') then p1 <= ("000" & a & '0');
      else p1 <= zero;
      end if;
      if (b(2)='1') then p2 <= ("00" & a & "00");
      else p2 <= zero;
      end if;
      if (b(3)='1') then p3 <= ('0' & a & "000");
      else p3 <= zero;
      end if;
      prod <= (p3 + p2) + (p1 + p0);
   end process;
end rtl;
```

일단 원리를 생각해 봅시다.
예를 들어 설명하겠습니다.

```
          1 1 0 1     -- 입력 a
    ×   | 1 0 1 1     -- 입력 b
          1 1 0 1     -- b(0)가 0이 아니므로  -- p0
        1 1 0 1       -- b(1)가 0이 아니므로  -- p1
      0 0 0 0         -- b(2)가 0이므로       -- p2
    1 1 0 1           -- b(3)가 0이 아니므로  -- p3
  1 0 0 0 1 1 1 1
```

결과는 무조건 8비트를 초과할 수 없습니다. 결과의 8비트를 맞추어 주기 위해서는 비트 결합 & 연산자를 사용합니다. 비트 수는 정확히 맞추어 주어야 합니다.
이런 4단계의 연산을 이용해서 p0, p1, p2, p3을 마지막으로 더해 줍니다. 그런데 우선 순위를 생각해서 2단으로 구성되었습니다. 만약 괄호가 없다면 지연 속도가 조금 많아질 것입니다. 그런 미세한 부분까지 설계해야 좋은 시스템을 만들 수 있습니다.
어려운 예제는 아니었다고 생각되지만 이런 최적 화된 곱셈기라고 정의할 수는 없습니다.
만약 속도 문제만을 최우선으로 설계한다면 다른 모양의 설계가 가능합니다. 다만 Area가 증가하겠지요. 여러분도 어느 정도의 Know-how가 생긴다면 알파칩을 능가할 회로를 설계해 보십시오. 최적 화된 알고리즘을 이용하면 됩니다. 그 설계는 우리의 과제입니다.
제가 강의하고 있는 예제는 대부분 컴파일 과정을 거쳐 에러를 모두 수정한 예제입니다. 시뮬레이션까지 해보는 회로도 있지만 해보지 않는 회로도 많습니다. 여러분은 시뮬레이션까지 꼭 해보십시오. 모든 경우를 다해 보지는 못하겠지만 할 수 있는 정도까지는 해보아야 합니다. 시작과 끝 그리고 상태가 변하는 부분은 세밀하게 검토하고 넘어가는 습관을 가지기를 바랍니다.

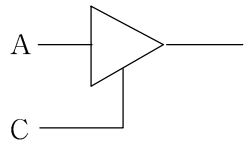
T flipflop은 입력 t가 '1'일 때 클럭의 상승에서 출력 q가 토글(toggle)되어 반전된다. 만약 이 때 입력 t가 '0'이면 출력 q는 변하지 않는다.
그렇게 많이 사용되지는 않습니다. 대부분의 회로는 면적 때문에 D flipflop을 많이 이용하고 있습니다. 기능상의 문제는 조금 떨어지는 면이 있지만 회로 최적화라는 문제 때문에 DFF을 사용합니다.

ex2) T Flipflop

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

entity t_ff is
port ( t, clk    : in std_logic;
           q     : buffer std_logic);
end t_ff;

architecture rtl of t_ff is
begin
   process(clk)
   begin
      if (clk='1' and clk' event) then
         if (t='1') then
            q <= not(q);
         else
            q <= q;
         end if;
      end if;
   end process;
end rtl;
```

포트에서 q를 buffer 모드로 설정했습니다. feedback을 허용하는 모드로 설정했습니다.
q <= q; 자기 자신으로 assign 되기 위해서는 버퍼 모드로 설정되어야 합니다.

3상태 버퍼에 대해서 알아봅시다. 대부분의 게이트는 On이 되거나 Off가 되어 회로 연결 상태를 보여줍니다. Totem-pole의 경우 아래쪽 트랜지스터가 차단되면 High 상태의 전압을 가지고 반대로 위쪽 트랜지스터가 차단될 경우 Low 상태의 전압을 가진다. 그러나 특이한 경우로 양쪽 트랜지스터가 차단될 경우 제3의 상태 즉 개방 회로를 가집니다. 즉 높은 임피던스 상태를 제공하여 공통선에 많은 출력을 직접 결선형으로 연결할 수 있게 합니다. 대부분의 3상태 게이트의 중요한 특징은 출력 이네이블 지연(output enable delay)이 출력 디스에이블 지연(output disable delay)보다 길다는 점입니다.



C가 High이면 Y = A 이고
C가 Low이면 Y = High Impedance

이론은 이 정도로 하고 어떻게 구현하는지 살펴봅시다.
예제는 2가지로 기술하겠습니다. 병렬적 처리와 순차적 처리로 정의하겠습니다.

ex3) Tri-state Buffer

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

entity tristate is
port ( e, a : in std_logic;
         y    : out std_logic);
end tristate;

architecture rtl of tristate is
begin
    process(e, a)
    begin
        if (e='1') then
            y <= a;
        else
            y <= 'Z';
        end if;
    end process;
end rtl;
```

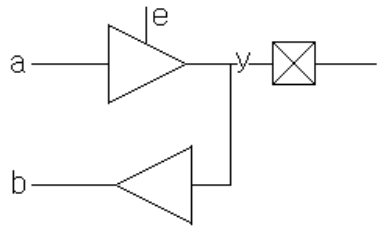
어느 정도의 기능만 알고 있다면 정의하기는 쉽습니다. 여기서 ' Z '는 nine values에서 High Impedance를 나타냅니다.

ex4) Tris-state Buffer

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

entity tristate1 is
port ( e, a : in std_logic;
         y    : out std_logic);
end tristate1;

architecture rtl of tristate1 is
begin
    y <= a when (e='1') else 'Z';
end rtl;
```

어떤 때는 when - else 구문을 사용하면 회로가 간결하게 표현되는 장점이 있습니다.

다음은 입출력 버퍼에 대해서 알아보겠습니다.



e가 1이면 y <= a가 되고
e가 0이면 y는 high Impedance
e가 1과 0이외에는 Unknown 상태
(e는 std_logic의 nine value를 가지므로)

y는 inout 모드로 설정되어야 한다. 그 이유는 회로의 원리를 자세히 보기를 바란다.

ex5) Bi-directional Buffer

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

entity bidir is
port (    y    : inout std_logic;
        e, a   : in std_logic;
          b    : out std_logic);
end bidir;

architecture rtl of bidir is
begin
   process(e, a)
   begin
      case e is
         when '1' => y <= a;
         when '0' => y <= 'Z';
         when others => y <= 'X';
      end case;
   end process;
   b <= y;
end rtl;
```

일부 툴 중에는 device의 영향으로 **inout 모드 대신에 buffer 모드로 설정해야 하는 경우**도 있습니다.

마지막으로 shift counter와 레지스터의 설계에 대해서 잠시 알아보겠습니다.
레지스터 부분은 메모리 회로 설계에서 다루어야 하지만 여기서 잠시 범용 적으로 많이 쓰이는 8비트 레지스터에 대해서 알아보겠습니다. 레지스터는 데이터의 로드가 가능해야 합니다.

ex6) 8 bit Register

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

entity register1 is
port ( enable, load, clk : in std_logic;
        reset   : in std_logic;
        regin   : in std_logic_vector(7 downto 0);
        loaddata : in std_logic_vector(7 downto 0);
        regout : out std_logic_vector(7 downto 0));
end register1;

architecture rtl of register1 is
signal gatedclk : std_logic;
begin
   gatedclk <= enable and clk;
   process(gatedclk, load, reset, enable, clk)
   begin
      if (reset='1') then
         regout <= "00000000";
      elsif (gatedclk='1' and gatedclk' event) then
         if (load='1') then
            regout <= loaddata;
         else
            regout <= regin;
         end if;
      end if;
   end process;
end rtl;
```

reset이 1이면 모든 회로는 초기 값으로 set됩니다. 당연히 두 개가 되어야 합니다. 레지스터는 자기가 가지고 있는 입력이 있고, 새로운 값이 로드될 수 있는 기능을 수행해야 하므로 입력이 두 개로 되어 있습니다. 만약 Common bus를 가지고 있다면 입력이 하나로도 충분히 가능하다고 생각됩니다. 그렇지만 여기서는 두 개의 입력을 가지고 있습니다.

로드 신호가 있느냐 없느냐에 따라서 레지스터 출력 값이 결정됩니다. 회로 원리만 이해하고 있어도 코딩은 별로 어렵지 않습니다. 조금 복잡한 시스템을 설계하다보면 가장 많이 사용되는 회로가 레지스터입니다.

ex7) Shift Register

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

entity shiftreg is
port ( enable, load, clk : in std_logic;
       din    : in std_logic;
       regin  : in std_logic_vector(7 downto 0);
       mode   : in std_logic;
       regout : out std_logic_vector(7 downto 0));
end shiftreg;

architecture rtl of shiftreg is
signal gatedclk : std_logic;
signal tmp : std_logic_vector(7 downto 0);
begin
   gatedclk <= enable and clk;
   regout <= tmp;
   process(gatedclk, din, load, mode, clk)
   begin
      if (gatedclk='1' and gatedclk' event) then
         if (load='1') then
            tmp <= regin;
         elsif (mode='0') then
            tmp <= tmp(6 downto 0) & din;
         else
            tmp <= din & tmp(7 downto 1);
         end if;
      end if;
   end process;
end rtl;
```

위의 예제와 조금 다른 부분이 있다면 mode 부분일 것이다. mode 선택에 따라서 shift left와 shift right의 기능을 수행한다. 비트 결합 연산자 &를 사용하면 간단하게 shift register를 설계할 수 있다.

```
tmp <= tmp(6 downto 0) & din;  -- shift left
tmp <= din & tmp(7 downto 1);  -- shift right
```

회로를 조금만 변형한다면 아주 재미있게 바꿀 수 있다. shift rotate의 기능을 수행하게 할 수도 있다.

```
tmp <= tmp(6 downto 0) & tmp(7);  -- shift rotate left
tmp <= tmp(0) & tmp(7 downto 1);  -- shift rotate right
```

State Machine

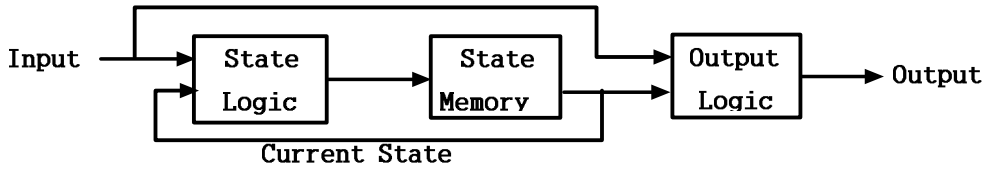
FSM(Finite State Machine)에 대해서 알아보겠습니다.

1. 디지털 하드웨어를 설계할 때 제어 신호를 생성하는 제어기는 FSM을 규정되며 FSM의 동작 표현은 state transition diagram 또는 state transition table 등으로 구현됩니다.
2. FSM은 state 변수를 기억하는 레지스터 블록과 state 변수의 천이를 표현하는 함수와 출력 값을 결정하는 조합 논리 회로로 구성된다.

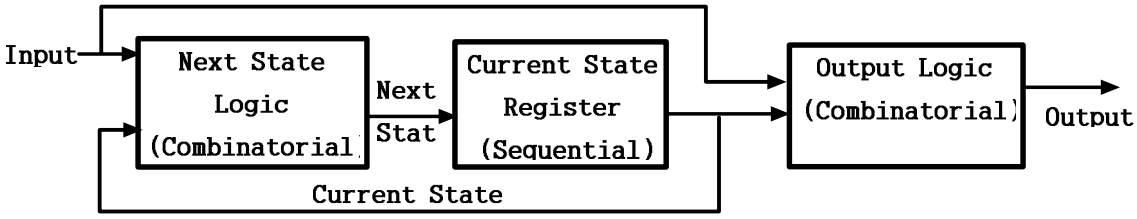
State Machine은 크게 Mealy Machine과 Moore Machine으로 구분됩니다.

Mealy machine은 출력이 현재 상태와 현재 입력에 의존하는 State Machine이고, Moore Machine은 출력이 현재 상태에만 의존하는 State Machine을 말한다.

Mealy Machine은 순차 회로의 출력이 현재의 State와 입력에 따라 결정되는 것을 말한다.



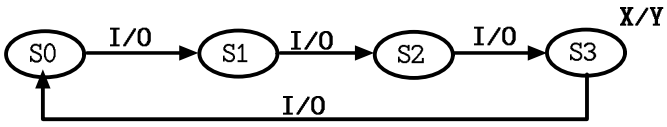
조금 다른 식으로 설명하면 다음 그림을 이해하세요.



장점은 Moore Machine에 비해 State 수가 적다는 점입니다.

개념은 이 정도로 이해하고 실제 예제로 이해해 봅시다.

State Machine의 설계는 어떤 회로의 설계를 정하고 상태도나 상태표를 먼저 구현합니다.



Input (X)	Present State	Next State	Output (Z)
0	S0	S0	0
1	S0	S1	0
0	S1	S1	0
1	S1	S2	0
0	S2	S2	0
1	S2	S3	0
0	S3	S3	0
1	S3	S0	1

State를 최소한으로 줄이는 것은 당연한 것이겠죠. state 수를 줄이는 방법은 몇 가지가 있습니다. 그 방법은 여러분이 직접.

위의 상태표를 보고 직접 Coding 작업을 해야 합니다. 어렵다고 생각하시지 마시고 천천히 시작해 보십시오. State Machine은 처음이라서 조금 어렵겠지만 자기 스스로 한 번 설계해 보시고 아래의 코딩과 비교해 보시기를 바랍니다.

ex1) Mealy Machine1

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

entity fsm1 is
port ( x, clk : in std_logic;
         z     : out std_logic);
end fsm1;

architecture rtl of fsm1 is
type state is (s0, s1, s2, s3);
signal st : state;
begin
   process(clk)
   begin
      if (clk='1' and clk' event) then
         if (x='0') then z <= '0';
         else
            case st is
               when s0 => st <= s1;
                     z <= '0';
               when s1 => st <= s2;
                     z <= '0';
               when s2 => st <= s3;
                     z <= '0';
               when s3 => st <= s0;
                     z <= '1';
            end case;
         end if;
      end if;
   end process;
end rtl;
```

자 이제 분석을 한 번 해봅시다.

type state is (s0, s1, s2, s3);

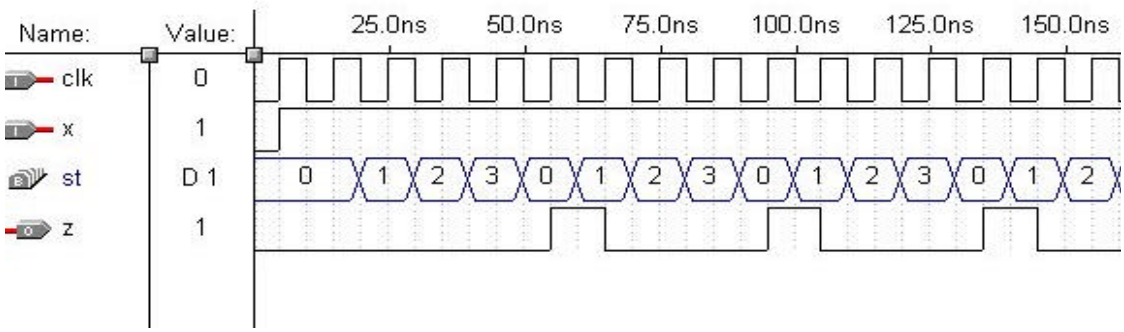
signal st : state;

state를 새로운 data type으로 정의했습니다. **s0, s1, s2, s3**을 순서대로 정의해 주어야 됩니다. 시뮬레이션 시에 초기화하기 위해서도 반드시 state 순서를 정의해 주어야 합니다. 이 부분이 state machine의 최적화에 큰 영향을 미칩니다. FF의 초기 상태를 알기 위해 reset 단자를 추가해 **initial state**를 만들어 주어야 합니다. 반드시 초기화 회로를 만들어 주는 것이 좋습니다.

when s0 => st <= s1; z <= '0';

네 가지의 정의가 있지만 한 가지만 이해하면 다른 부분은 쉽게 이해가 될 것입니다.

S0은 Current State를 나타내고 S1은 Next State를 나타냅니다. S0의 State를 S1의 State로 보내고 Z의 출력 값을 0으로 출력합니다. 여기서는 큰 변수가 없으므로 순차적인 진행을 나타냅니다. 다음의 결과를 봅시다.



when s3 => st <= s0; z <= '1';

이 상태에서 z의 값이 1로 변하는 것을 볼 수 있을 것입니다.

위의 예제보다는 밑에 조금 변형되어 있는 회로 설계 방법이 조금 좋다고 생각합니다.

FSM을 합성할 때 FSM에서의 설계의 동기 회로 부분(Synchronous)을 조합 회로 부분(Combinatorial)과 분리함으로써 순차 회로와 조합 회로의 설계를 용이하게 할 뿐 만 아니라 설계의 수정이나 오류 정정이 훨씬 간단하다는 장점을 가지고 있습니다.

ex2) Mealy Machine

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

entity fsm2 is
port ( x, clk : in std_logic;
           z     : out std_logic);
end fsm2;
```

```
architecture rtl of fsm2 is
type state is (s0, s1, s2, s3);
signal current_st, next_st : state;
begin
    comb : process(current_st, x)
    begin
        case current_st is
            when s0 =>
                if (x='0') then
                    z <= '0';
                    next_st <= s0;
                else
                    z <= '0';
                    next_st <= s1;
                end if;
            when s1 =>
                if (x='0') then
                    z <= '0';
                    next_st <= s1;
                else
                    z <= '0';
                    next_st <= s2;
                end if;
            when s2 =>
                if (x='0') then
                    z <= '0';
                    next_st <= s2;
                else
                    z <= '0';
                    next_st <= s3;
                end if;
            when s3 =>
                if (x='0') then
                    z <= '0';
                    next_st <= s3;
                else
                    z <= '0';
                    next_st <= s0;
                end if;
        end case;
    end process;
```

```
    Synch : process(clk)
    begin
        if (clk='1' and clk' event) then
            current_st <= next_st;
        end if;
    end process;
end rtl;
type state is (s0, s1, s2, s3);
signal current_st, next_st : state;
```

새로운 data type s0, s1, s2, s3을 정의하고 이것에 대한 state를 두 개로 표시했습니다. 위의 예제는 current state와 next state를 구분하기가 어려웠는데 지금은 현재의 상태를 훨씬 구분하기가 편하게 되어 있습니다. 그리고 process 구문을 한 개에서 두 개로 정의했습니다. 몇 개의 process를 쓰든 상관은 없습니다만 시스템 최적화를 위해서는 한 개의 process 보다는 두 개의 process로 state machine을 구현하는 것이 훨씬 좋다고 생각합니다.

너무 많은 프로세서 문을 사용하는 것은 좋지 않습니다. state machine의 설계에서는 2개 내지 3개 정도가 적당하다고 생각합니다.(제 경험으로는)

그리고 이번에는 process 문에 라벨을 사용했습니다. process 문의 구분을 위한 것이므로 시스템에는 아무 영향을 주지 않습니다.

```
Synch : process(clk)
        begin
            if (clk='1' and clk' event) then
                current_st <= next_st;
            end if;
        end process;
```

전에도 잠깐 설명했습니다만 process(**clk**) 여기서 clk가 sensitivity list라고 설명했습니다.

이것을 사용하지 않는 방법도 있습니다. wait until 구문을 이용해서 sensitivity list를 대치하는 방법을 소개해 드리겠습니다.

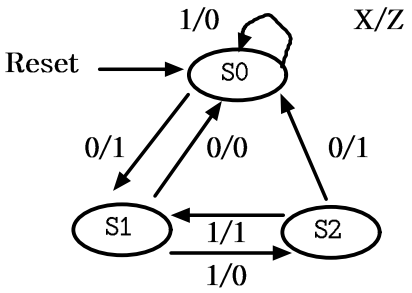
wait until 구문은 구문 다음에 오는 상황에 따라서 변합니다. 다음에 변하는 부분이 있을 때까지는 대기하고 있습니다.

```
Synch : process(clk)
        begin
            wait until (clk='1' and clk' event);
            current_st <= next_st;
        end process;
```

위에서 보면 클럭의 변화가 있기 전까지는 기다리고 있습니다. 클럭의 변화가 있으면 current_st <= next_st;을 수행합니다.

다음 예제는 reset을 가지고 있는 state machine입니다.

상태도를 먼저 봅시다.



ex3) State Machine -- Mealy Machine

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

entity fsm3 is
port ( reset, x, clk : in std_logic;
          z     : out std_logic);
end fsm3;

architecture rtl of fsm3 is
type state is (s0, s1, s2);
signal c_state, n_state : state;
begin
   Synch : process(clk, reset)
   begin
      if (reset='0') then
         c_state <= s0;
      elsif (clk='1' and clk' event) then
         c_state <= n_state;
      end if;
   end process;

   comb : process(c_state, x)
   begin
      case c_state is
         when s0 => z <= '0';
            if (x='0') then
               n_state <= s0;
            else
               n_state <= s1;
            end if;
         when s1 => z <= '0';
            if (x='0') then
```

```
                n_state <= s0;
            else
                n_state <= s2;
            end if;
        when s2 =>
            if (x='0') then
                z <= '1';
                n_state <= s0;
            else
                z <= '1';
                n_state <= s1;
            end if;
    end case;
  end process;
end rtl;
```

위의 예제와 비교해서 조금 다른 면을 볼 수 있을 것입니다. Reset 기능을 가지고 있습니다. 그리고 일방적인 state의 흐름이 아니라 양방향으로 변한다는 점입니다. 예제로서 좋다고 생각됩니다.

이 예제는 조금 자세하게 분석하겠습니다.

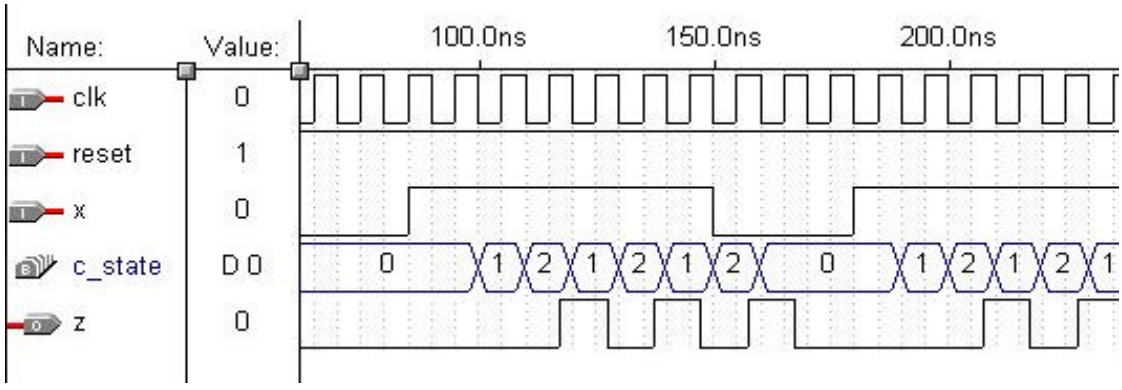
comb : process(c_state, x)

위 process 구문은 state 변수를 기억하는 레지스터에 대한 표현을 위해 사용되었다.

Synch : process(clk, reset)

위 process 구문은 state 변수의 천이와 출력 값의 결정을 위한 함수의 표현을 위해 사용되었습니다.

결과는 다음과 같습니다.




```
type state is (s0, s1, s2);
signal c_state, n_state : state;
```

우리는 State Machine의 개념을 도입하여 설계했습니다. 그냥 Binary Coding으로 설계한다면 다음과 같습니다.

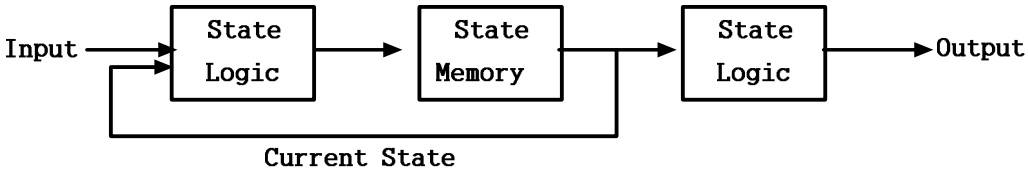
우선 type 선언문을 삭제하고 constant로 정의하면 됩니다.

```
constant s0 : std_logic_vector(1 downto 0) := "00";
constant s1 : std_logic_vector(1 downto 0) := "01";
constant s2 : std_logic_vector(1 downto 0) := "10";

signal c_state, n_state : std_logic_vector(1 downto 0);
```

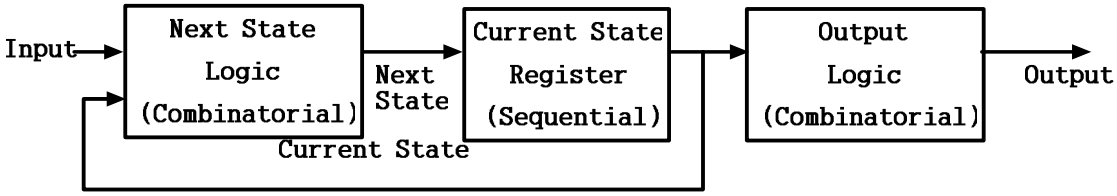
그렇지만 Binary coding으로 설계하는 것보다 state machine을 설계하는 것이 최적화의 도움을 줍니다. 시스템을 최적화 시키는 알고리즘이 Binary coding보다는 state machine 쪽이 더 잘되어 있다고 합니다.

다음은 Moore Machine에 대해서 알아보도록 하겠습니다.
Moore Machine : 순차 회로의 출력이 입력과 관계없이 현재의 상태의 함수로만 결정됩니다.

Input과 무관하므로 glitch가 덜 발생한다는 장점이 있습니다.

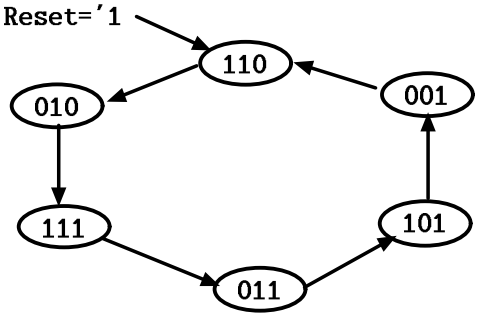


조금 다른 개념으로 알아봅시다.



조금 어렵지 않습니까? 저도 state machine을 설계하다 보면 이것이 Mealy Machine인지 아니며 Moore machine인지 알 수 없을 때가 있습니다. 너무 개념이 어렵다고 생각하시지 마시고 지금은 그냥 차이점 정도만 알아두시는 것이 좋을 것 같습니다.

다음 예제를 Moore Machine으로 설계해 봅시다.



state의 상태도의 순서가 조금 다릅니다. 이런 식의 설계도 가능합니다. 다만 순서의 영향을 조금 받는다는 점입니다.

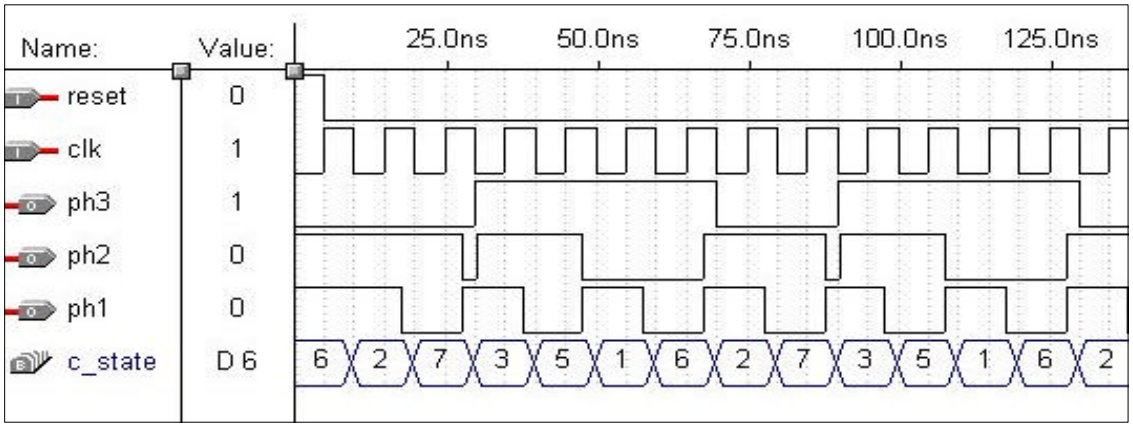
ex4) Moore Machine

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

entity fsm4 is
port ( reset, clk : in std_logic;
         ph1, ph2, ph3 : out std_logic);
end fsm4;

architecture rtl of fsm4 is
constant s0 : std_logic_vector(2 downto 0) := "110";
constant s1 : std_logic_vector(2 downto 0) := "010";
constant s2 : std_logic_vector(2 downto 0) := "111";
constant s3 : std_logic_vector(2 downto 0) := "011";
constant s4 : std_logic_vector(2 downto 0) := "101";
constant s5 : std_logic_vector(2 downto 0) := "001";
signal c_state, n_state : std_logic_vector(2 downto 0);
signal tmp_ph3 : std_logic;
signal ph : std_logic_vector(2 downto 0);
begin
    p1 : process(clk, reset)
    begin
        if (reset='1') then
            c_state <= s0;
        elsif (clk='1' and clk' event) then
            c_state <= n_state;
        end if;
    end process;

    p2 : process(c_state)
    begin
        case c_state is
            when s0 => ph <= s0;
                    n_state <= s1;
            when s1 => ph <= s1;
                    n_state <= s2;
            when s2 => ph <= s2;
                    n_state <= s3;
            when s3 => ph <= s3;
                    n_state <= s4;
            when s4 => ph <= s4;
```

```
                n_state <= s5;
            when others => ph <= s5;
                n_state <= s0;
        end case;
        ph1 <= ph(2);
        ph2 <= ph(1);
        tmp_ph3 <= ph(0);
    end process;

    p3 : process(clk)
    begin
        if (clk='0' and clk' event) then
            ph3 <= tmp_ph3;
        end if;
    end process;
end rtl;
```

여기에는 3가지의 process 구문이 사용되었습니다. 2개의 프로세서 구문을 이용하여 설계가 가능하지만 ph3은 하강에지에서 동작하도록 설계되었습니다. 이처럼 상승에지와 하강에지를 동시에 이용하는 설계도 가능합니다.

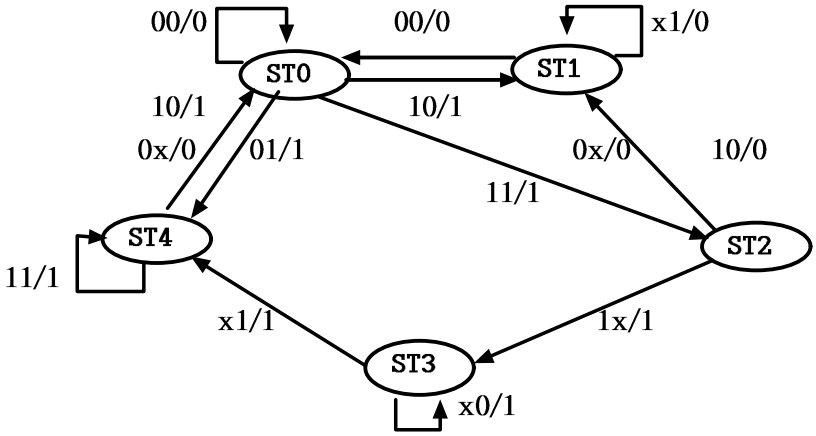
다음 결과에서 보면 ph1과 ph2는 상승에지에서 동작하는 것을 볼 수 있고, ph3은 하강에지에서 동작하는 것을 알 수 있습니다.



110 → 010 → 111 → 011 → 101 → 001

6 → 2 → 7 → 3 → 5 → 1로 변하는 것을 C_state의 변화로 알 수 있습니다. 결과는 정상적이라는 말입니다. 그러면 동작 점을 살펴봅시다. ph1과 ph2가 상승에지에서 변하는 지를 위 그림에서 확인해 보세요.

아직까지는 그렇게 차이점을 이해하기 힘들 것입니다. 그럼 같은 예제를 두 가지로 표현해 보겠습니다. 조금 복잡한 예제입니다. 그렇게 어렵지는 않습니다.



다음은 위의 상태도를 중심으로 설계한 예제를 보여줍니다.

ex5) Mealy FSM

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

entity fsm5 is
port ( reset, clk : in std_logic;
        data_in : in std_logic_vector(1 downto 0);
        data_out : out std_logic);
end fsm5;

architecture rtl of fsm5 is
type state is (st0, st1, st2, st3, st4);
signal c_state, n_state : state;
begin
    statereg: process(clk, reset)
    begin
        if (reset='1') then
            c_state <= st0;
        elsif (clk='1' and clk' event) then
            c_state <= n_state;
        end if;
    end process statereg;

    fsm : process(c_state, data_in)
    begin
        case c_state is
            when st0 =>
                case data_in is
```

```
            when "00" => n_state <= st0;
            when "01" => n_state <= st4;
            when "10" => n_state <= st1;
            when "11" => n_state <= st2;
            when others => n_state <= null;
        end case;
    when st1 =>
        case data_in is
            when "00" => n_state <= st0;
            when "10" => n_state <= st2;
            when others => n_state <= st1;
        end case;
    when st2 =>
        case data_in is
            when "00" => n_state <= st1;
            when "01" => n_state <= st1;
            when "10" => n_state <= st3;
            when "11" => n_state <= st3;
            when others => n_state <= null;
        end case;
    when st3 =>
        case data_in is
            when "01" => n_state <= st4;
            when "11" => n_state <= st4;
            when others => n_state <= st3;
        end case;
    when st4 =>
        case data_in is
            when "11" => n_state <= st4;
            when others => n_state <= st0;
        end case;
    when others => n_state <= st0;
  end case;
end process fsm;

output : process(c_state, data_in)
begin
  case c_state is
    when st0 =>
        case data_in is
            when "00" => data_out <= '0';
```

```
                when others => data_out <= '1';
            end case;
        when st1 => data_out <= '0';
        when st2 =>
            case data_in is
                when "00" => data_out <= '0';
                when "01" => data_out <= '0';
                when others => data_out <= '1';
            end case;
        when st3 => data_out <= '1';
        when st4 =>
            case data_in is
                when "10" => data_out <= '1';
                when "11" => data_out <= '1';
                when others => data_out <= '0';
            end case;
        when others => data_out <= '0';
    end case;
  end process output;
end rtl;
```

ex6) Moore FSM

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

entity fsm6 is
port ( reset, clk : in std_logic;
       data_in : in std_logic_vector(1 downto 0);
       data_out : out std_logic);
end fsm6;

architecture rtl of fsm6 is
type state is (st0, st1, st2, st3, st4);
signal c_state, n_state : state;
begin
  statereg: process(clk, reset)
  begin
    if (reset='1') then
      c_state <= st0;
    elsif (clk='1' and clk' event) then
      c_state <= n_state;
```

```
    end if;
end process statereg;

fsm : process(c_state, data_in)
begin
    case c_state is
        when st0 =>
            case data_in is
                when "00" => n_state <= st0;
                when "01" => n_state <= st4;
                when "10" => n_state <= st1;
                when "11" => n_state <= st2;
                when others => null;
            end case;
        when st1 =>
            case data_in is
                when "00" => n_state <= st0;
                when "10" => n_state <= st2;
                when others => n_state <= st1;
            end case;
        when st2 =>
            case data_in is
                when "00" => n_state <= st1;
                when "01" => n_state <= st1;
                when "10" => n_state <= st3;
                when "11" => n_state <= st3;
                when others => null;
            end case;
        when st3 =>
            case data_in is
                when "01" => n_state <= st4;
                when "11" => n_state <= st4;
                when others => n_state <= st3;
            end case;
        when st4 =>
            case data_in is
                when "11" => n_state <= st4;
                when others => n_state <= st0;
            end case;
        when others => n_state <= st0;
    end case;
```

```
    end process fsm;

    output : process(c_state)
    begin
        case c_state is
            when st0 => data_out <= '1';
            when st1 => data_out <= '0';
            when st2 => data_out <= '1';
            when st3 => data_out <= '0';
            when st4 => data_out <= '1';
            when others => data_out <= '0';
        end case;
    end process output;
end rtl;
```

두 가지의 같은 예제를 가지고 비교해 보세요. 새로운 부분은 없다고 생각되므로 각 부분에 대한 설명은 생략하겠습니다.

다음 시간에는 지금까지 설명하지 못한 문법적인 부분과 Package에 대한 부분을 잠시 다루도록 하겠습니다. 다음 시간까지가 기초적인 부분에 대한 설명이 될 것입니다. 그 다음부터는 실제 시스템에 조금 접근하도록 하겠습니다.

Package와 부프로그램

자기가 설계한 프로그램을 다시 사용할 수 있다면 설계의 시간의 단축시킬 수 있을 것입니다. 이런 개념이 VHDL에는 있습니다. 이런 개념이 Reuse라는 개념입니다. Package란 많이 사용되는 부분을 한 장소에 모아서 여러 설계에 공유할 수 있는 것을 말합니다. 한 사람만이 그 package를 이용할 수 있는 것이 아니라 설계자 모두가 사용할 수 있습니다.

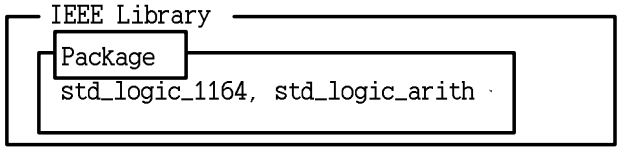
우선 library와 package에 대해서 알아보겠습니다. library는 이전에 설계한 설계 단위(design unit) 즉, entity architecture, package 등을 저장하여 필요시에 사용할 수 있도록 하는 일종의 저장 장소(directory)라고 보면 됩니다.

Package는 자료형(type), 함수(function), Procedure 등을 한 장소에 모아 선언한 것이다.

Package는 library에 종속되며 다른 VHDL 문장에서 use구문에 의해 불려진다.

먼저 library에 대해서 알아봅시다. VHDL 문장이 VHDL 해석기에 의해 해석된 문장은 지정된 library에 저장된다. 만약 특별히 저장될 library가 지정되어 있지 않으면 work library에 저장된다. work library는 현재 실행 중인 작업을 저장하는 곳(directory)으로 VHDL 사용자가 사용하고 있는 현재의 library를 말한다.

조금 쉽게 설명하면



library란 큰 테두리 속에 package란 개념이 있습니다.

```
library ieee;  -- IEEE라는 Library
use ieee.std_logic_1164.all; -- IEEE라는 Library속에 포함되어 있는 std-1164라는 package
```

Library의 종류에 대해서 잠깐 알아봅시다.

IEEE Library에 대해서는 많이 다루어 보았습니다. std_logic_1164, std_logic_arith 등의 package가 포함되어 있는 라이브러리입니다. 그리고 아직까지 다루어 보지 않은 STD Library내에 textio라는 package가 있습니다. textio는 ASCII 파일(그리고 표준 입출력)에서 읽거나 쓰기를 위해서 사용되는 subprogram을 정의하고 있습니다. 자기가 가지고 있는 Tool에서 한 번 찾아보세요. 그리고 ASIC 공급자가 제공하는 library가 있습니다. 이것은 Tool을 만드는 회사에서 제공하는 library입니다. 이것은 사용자가 많이 사용하는 부분을 라이브러리로 만든 것이므로 알고 있으면 참 편리할 것입니다. 회사에서 제공하는 매뉴얼을 보시면 알 것입니다. 그리고 사용자가 정의한 library가 있습니다. 이것은 본인이 직접 만들 수 있습니다. 자기가 많이 사용하는 부분을 라이브러리화 하면 설계의 많은 시간을 단축할 수 있을 것입니다. 그렇지만 생각해야 하는 부분이 있습니다. 자기가 사용할 라이브러리가 어디에 있는 지를 명확히 정의해야 할 필요가 있습니다. 이것을 가시성(Visibility)을 부여한다고 말합니다. 가시성이란 package나 library내에서 선언되어 있거나 정의된 것을 사용할 수 있게 경로를 설정하는 것입니다. 라이브러리는 자동으로 가시화 되지 않으므로 library 구문을 사용하여 가시화 시키고 해당 라이브러리를 가시화 시킨 다음 use 구문으로

package를 가시화 시킨다.
예를 들어서 IEEE라는 Library를 가시화 시킵니다.
library ieee;
이 가시화 된 내용을 사용한다는 의미로 다음과 같은 구문을 사용합니다.
use ieee. std_logic_1164.all;
여기서 ieee는 라이브러리 이름을 나타내고, std_logic_1164는 package의 이름을 나타냅니다. all 이란 것은 모든 것을 사용하겠다는 의미입니다.
주로 예를 IEEE 기준으로 설명을 했는데 회로 설계에 있어서 IEEE 라이브러리가 반드시 들어가야 하는 것은 아닙니다. 때에 따라서는 자기가 설계한 라이브러리만 포함되어 있는 회로도 설계가 가능합니다. 제가 IEEE 라이브러리를 자주 사용하는 이유는 가장 기본적인 라이브러리이기 때문입니다.
다음 예제를 잘 보세요.
use work. sample. all;
만약 사용하려는 package가 work library에 있을 경우에는 별도로 가시화 시킬 필요가 없이, use구문으로 해당 package를 가시화 시킵니다. work library는 현재 수행하는 디자인의 default 작업 라이브러리로서 현재 작업을 분석하고 저장하는 장소로 여러 사람이 공유할 수 있는 곳을 말합니다.

그럼 다음은 package에 대해서 알아봅시다.
package를 작성하려면 일정한 형식을 사용해야 합니다. 그 형식은 반드시 지켜야 합니다.
package는 크게 두 가지로 구성됩니다. package의 선언부와 몸체(body)로 구성됩니다. 패키지 선언의 역할은 외부에서 사용할 수 있는 인터페이스를 담당하는 것이며, 자료 타입의 선언이나 부프로그램의 이름(procedure, function)이 있습니다. 패키지 몸체는 패키지 선언에서 선언한 부프로그램의 구체적 내용이 들어 있습니다. 이것은 entity 선언과 비슷합니다.

```
-- package 선언
package package_이름 is
  자료형(type) 선언;
  부프로그램 선언;
end  package_이름;

-- package 몸체
package body package_이름 is
  부프로그램에 대한 VHDL 구문;
end package_이름;
```

entity와 architecture의 사용과 비슷하다는 생각이 들것입니다. package에 대한 예제는 아직까지 다루지 않았습니다. 그래서 조금 생소하게 느껴질 것입니다. 그렇지만 package를 잘 이용하면 상당한 시간 절약 효과를 기대할 수 있을 것입니다.
일단 두 가지의 설명이 필요합니다. 부프로그램을 사용할 때와 사용하지 않을 때 이 두 가지 경우를 나누어서 생각하겠습니다.

```
package tristate is
    type trivalue is ('0', '1', 'z', 'x');
    constant clk_h : time := 20ns;
    constant clk_l : time := 20ns;
end tristate;
```

다른 부분은 생략했습니다. 위의 경우는 package 몸체를 가지지 않습니다. 위의 경우는 자료형과 상수의 선언만으로 이루어져 있습니다. 이런 경우에는 package 몸체를 가지지 않고, 부프로그램이 package 선언에 사용된 경우에는 package 몸체를 가집니다.

그럼 부프로그램이 무엇인지 알아보아야 할 것입니다. 고급 언어에서는 함수를 사용할 수 있습니다. 사용자 편의를 위해서 VHDL에서도 function(함수)과 procedure를 사용할 수 있습니다. 자주 쓰이는 설계의 일부분을 따로 작성하거나, 기능적으로 분해 가능한 프로그램의 일부를 분리해서 작성할 수 있도록 function(함수)과 procedure를 사용할 수 있습니다.

package 내에서 function과 procedure의 예를 한 번 봅시다. 어떤 식으로 사용되는지를 한 번 살펴보아야 할 것입니다.

```
package logic is
    function f_and2 (a, b : in bit) return bit;
    procedure p_and2 (a, b : in bit ; y : out bit);
end logic;

package body logic is
    function f_and2 (a, b : in bit) return bit is
        begin
            if (a='1') and (b='1') then
                return ('1');
            else
                return ('0');
            end if;
    end f_and2;
    procedure p_and2 (a, b : in bit; y : out bit) is
        begin
            if (a='1') and (b='1') then
                y <= '1';
            else
                y <= '0';
            end if;
    end p_and2;
end logic;
```

위의 예제는 and2를 library로 만든 것이다. function과 procedure를 이용하여 패키지화 하였다. 특정 라이브러리에 저장한 다음 이것을 호출하여 사용할 수 있습니다.

이것을 사용하는 예를 간단히 설명하겠습니다. 라이브러리를 사용하기 위해서는 라이브러리의 정의가 필요합니다. 이것을 작업 디렉토리에 만들었다면

library work;
use work. my_logic. all;

이것을 작업 디렉토리에 만들지 않고 패키지를 포함한 설계를 한다면 라이브러리의 가시화는 필요하지 않을 것입니다.

이것의 function과 procedure를 이용하는 방법은 함수의 경우는
y <= f_and2 (x1, x2);
Procedure의 경우는
p_and2 (x1, x2, y); 입니다.
함수의 호출과 Procedure의 호출에서 가장 큰 차이점은 되돌려 주는 값(return value)이 하나인 경우는 함수를 사용하고 여러 개의 값을 돌려줄 경우는 Procedure를 사용합니다.

function f_and2 (a, b : **in** bit) **return** bit;
procedure p_and2 (a, b : **in** bit ; y : **out** bit);
위에서 보면 function은 return 값을 가지고 있습니다. 이 말은 되돌려 주는 값이 하나라는 소리입니다. 그 반면에 procedure의 경우는 out 모드를 가지고 있습니다. out 모드로 지정된 것은 아닙니다.

function f_and2 (a, b : **in** bit) **return** bit; -- 함수의 정의
y <= f_and2 (x1, x2); -- 함수의 사용
함수의 경우는 y의 값이 return value입니다. 되돌려 주는 값이 고정된 것은 당연한 사실입니다. bit의 경우에는 요긴하게 사용될 수 있을 것입니다. 주요 패키지에 정의된 것을 보면 주로 함수를 많이 사용된 것을 볼 수 있을 것입니다.

procedure p_and2 (a, b : **in** bit ; y : **out** bit); -- Procedure의 정의
p_and2 (x1, x2, y); -- Procedure의 사용
그리고 함수의 경우는 in 모드만을 사용할 수 있습니다. 다른 것을 사용할 수 없을 것입니다. 함수의 형식의 정의에서 알 수 있을 것입니다. Procedure의 경우는 in, out, inout 모드를 사용할 수 있습니다.

procedure의 경우 이것을 패키지 내에서 작성할 수도 있고 아니면 architecture와 begin 사이나 process와 begin 사이에 작성할 수도 있습니다. 어디에 procedure를 이용하는 것은 설계자의 마음이니 상관하지 않으셔도 됩니다.
function과 procedure에 사용되는 VHDL 문장은 모두 순차 처리문 이어야 합니다.
process 문이 있어야 반드시 순차 처리문이 되는 것은 아닙니다.
다음 예제를 한 번 봅시다. 일단 logic_pkg라는 이름의 패키지를 만듭니다.

package logic_pkg **is**

```
        procedure and_p ( a, b : in bit;
                          c    : out bit);
        function or_f ( a, b : in bit) return bit;
end logic_pkg;

package body logic_pkg is
        procedure and_p ( a, b : in bit; c : bit ) is
        begin
                if (a='1') and (b='1') then c <= '1';
                else c <= '0';
                end if;
        end and_p;
        function or_f ( a, b : in bit) return bit is
        begin
                if (a='0') and (b='0') then return '0';
                else return '1';
                end if;
        end or_f;
end logic_pkg;
```

and와 or라는 순차 처리문을 표현한 것입니다. 다만 이 경우는 process 문을 사용하지 않았다는 것입니다. 다른 차이점은 없습니다. 프로세서 문이 있어도 상관없습니다. 이것을 한 번 이용하기 위해서 간단한 예제를 and_or. vhd 파일을 만들었습니다. 다음과 같습니다.

```
entity and_or is
        port ( a1, b1, c1 : in bit;
                z      : out bit);
end and_or;

library work;
use work. logic_pkg. all;

architecture rtl of and_or is
signal temp_z : bit;
begin
    process(a1, b1, c1)
    begin
        and_p (a1, b1,temp_z);
        z <= or_f (temp_z, c1);
    end process;
end rtl;
```

function과 procedure의 내부 VHDL 구문은 반드시 순차적으로 이루어져야 한다고 말씀드렸습니다. 이것을 호출할 때 function과 procedure의 위치는 process 안이나 다른 procedure나 function 안에 둘 수 있습니다. 이런 경우를 순차 호출(Sequential Call)이라고 합니다.

```
process(a1, b1, c1)
begin
   and_p (a1, b1, temp_z);
   z <= or_f (temp_z, c1);
end process;
```

위의 예제는 순차 호출로 구성되어 있습니다.

반드시 순차 호출로 구성되어야 하는 것은 아닙니다. 호출되는 부프로그램 양단에 process()와 end process 문이 등가적으로 삽입할 수 도 있습니다. 이런 경우를 병행 호출(Concurrent Call)이라고 합니다. 쉽게 말하자면 process 문 밖에 있거나 다른 procedure나 function 밖에 있다는 말입니다. 병렬적으로 구성시킨 다는 말입니다.

```
entity and_or is
   port ( a1, b1, c1 : in bit;
               z      : out bit);
end and_or;

library work;
use work. logic_pkg. all;

architecture rtl of and_or is
signal temp_z : bit;
begin
    g1 : and_p (a1, b1, temp_z);
    g2 : z <= or_f (temp_z, c1);
end rtl;
```

부프로그램의 호출 방식은 순차 호출이든지 아니면 병행 호출의 두 가지 방법이 있습니다. 어느 방법을 사용하든지 상관은 없습니다.
function은 지연 없는 계산에 쓰이는 중간에 사용되나 procedure는 수식의 중간에 사용될 수 없습니다. 그리고 function은 wait 문을 사용할 수 없으나 procedure는 wait 문을 사용할 수 있습니다.

function의 매개변수는 signal과 constant가 될 수 있으며 procedure의 매개변수는 signal, variable, constant가 될 수 있습니다. 객체가 명시되지 않으면 function의 매개변수는 constant이며, procedure 경우에는 mode가 function과 같이 in이면 constant로 가정되고 out, inout이면 variable로 가정됩니다.

Clock의 설계

이번 장에서는 클럭에 대한 문제를 한 번 살펴보겠습니다. 회로를 설계하고 이것을 테스트할 장비가 있을 것입니다. 각 회사마다 조금씩 다른 사양을 지원하는 계측 장비가 있습니다. 이것을 사용하는 데 있어서 가장 중요한 문제가 주파수를 사용하는 것입니다. 비동기식 회로에서는 클럭이 필요하지는 않지만 대부분의 회로에서는 클럭이 중요한 문제가 됩니다.

회사에서는 다양한 종류의 주파수가 제공되지만 자기가 필요로 하는 주파수를 만드는 것도 중요한 일입니다. 간단하게는 카운터를 이용해서 업/다운 회로를 설계할 수는 있지만 플립플롭으로 설계하는 것도 대단히 귀찮은 일입니다. VHDL 코딩으로 간단하게 작업할 수 있습니다. 그리고 자기가 원하는 정확한 주파수를 사용할 수 있습니다. 예를 들어 전자 시계를 설계한다고 했을 때 지원되는 주파수가 100Khz라고 한다면 이것을 1Hz의 주파수로 변환해야 합시다. 그래야 만이 1초를 발생시킬 수 있습니다. 그러기 위해서는 주파수 분주 회로를 설계해야 합니다. 우선은 입력된 주파수를 변환하는 것을 먼저 살펴 볼 것입니다.

이번 예제는 입력된 주파수를 1/10, 1/100, 1/1000의 클럭이 발생되도록 하는 주파수 분주 회로입니다. 3개의 프로세서 문으로 구성하였습니다.

ex1)

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

entity clk is
port ( clk : in std_logic;
        clk10, clk100, clk10000 : buffer std_logic);
end clk;

architecture rtl of clk is
signal count : integer range 0 to 7;
signal count1 : integer range 0 to 49;
signal count2 : integer range 0 to 49;
begin
    process(clk)
    begin
        if (clk='1' and clk' event) then
            if (count >= 4) then
                count <= 0;
                clk10 <= not clk10;
            else
                count <= count + 1;
            end if;
        end if;
    end process;
```

```
    process(clk)
    begin
        if (clk='1' and clk' event) then
            if (count1 >= 49) then
                count1 <= 0;
                clk100 <= not clk100;
            else
                count1 <= count1 + 1;
            end if;
        end if;
    end process;

    process(clk100)
    begin
        if (clk100='1' and clk100'event) then
            if (count2 >= 49) then
                count2 <= 0;
                clk10000 <= not clk10000;
            else
                count2 <= count2 + 1;
            end if;
        end if;
    end process;
end rtl;
```

여기서는 세 개의 Integer를 사용했습니다. Integer가 계산상의 편의 등의 용이합니다. 비트로 기록하는 것보다는 Integer를 사용할 때가 편할 때가 많습니다. 세 개의 프로세서 문으로 구성되어 있지만 내용은 같습니다. 하나의 프로세서 문만 살펴보겠습니다.

```
    process(clk)
    begin
        if (clk='1' and clk' event) then
            if (count >= 4) then
                count <= 0;
                clk10 <= not clk10;
            else
                count <= count + 1;
            end if;
        end if;
    end process;
```

clk는 입력입니다. 테스트 보드에서 정의된 클럭입니다. 위의 프로세서 문은 클럭의 1/10의 클럭을 발생하게 합니다. 내부 Signal은 클럭을 입력받아 0에서 4까지는 0으로 5에서 9까지는 1로 만들어 줍니다. 이것을 clk10의 출력으로 표시됩니다. 그래서 1/10의 분주가 가능한 것입니다. 위의 프로세서 문을 잘 활용하면 여러 분주 회로를 설계 할 수 있을 것입니다.

위의 예제는 단순하게 테스트 보드의 입력을 받아서 이것을 분주 하는 것입니다. 그렇지만 대용량의 칩을 설계한다면 자체의 클럭을 가지고 있어야 합니다. 클럭의 설계를 해야 하는 것은 당연한 일이겠지요.
다음의 예제는 조금 신기하게 생각될 지도 모르겠습니다. 왜냐하면 입력이 없기 때문입니다. 클럭을 만든다면 당연히 입력이 없어야 하겠지요.

ex2)

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

entity clk_gen is
port ( clk : out std_logic);
end clk_gen;

architecture rtl of clk_gen is
begin
    process
    begin
        clk <= '1' after 25 ns,
               '0' after 50 ns;
        wait for 50 ns;
    end process;
end rtl;
```

간단하게 설계했습니다. 그렇지만 Max+Plus에서는 지원을 하지 않을 지도 모르겠습니다. 대용량의 설계에는 부족한 점이 많다고 생각됩니다. Synopsis에서는 확실하게 지원됩니다. after와 wait for 문을 사용했습니다.
위의 예제가 몇 MHz인지 모르는 사람은 없겠죠.

$$\frac{1}{50\times 10^{-9}}\text{Hz} = 20\text{MHz}$$의 설계입니다.

조금만 더 응용하겠습니다. 보통 클럭의 설계와 리셋의 설계는 같이 하는 것이 보통입니다. 개인적인 취향에 따라 다르겠지만. 그럼 클럭과 리셋을 같이 설계해 보도록 하겠습니다.

ex3)

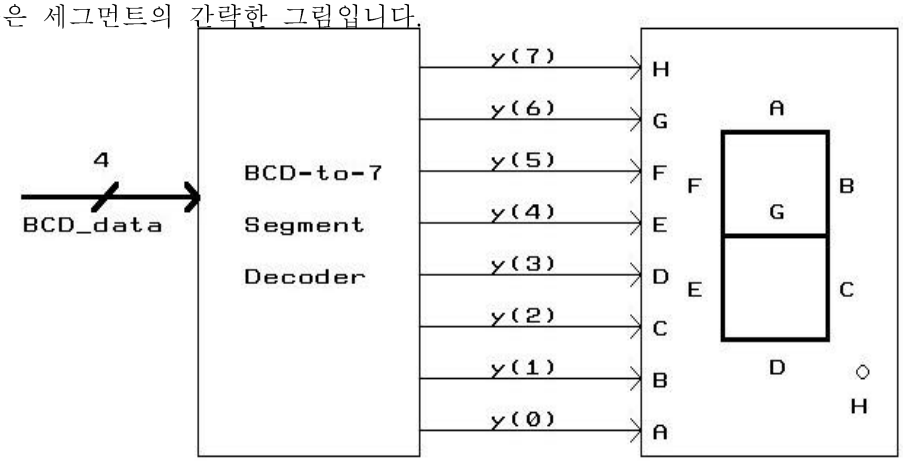
```
library ieee;
use ieee. std_logic_1164.all;
```

```
entity clk_gen is
port ( clk : out std_logic;
       reset : out std_logic);
end clk_gen;

architecture rtl of clk_gen is
constant pulse_width : time := 25 ns;
constant period : time := 50 ns;
begin
   process
   begin
      clk <= '1' after pulse_width,
             '0' after period;
      wait for period;
   end process;

   reset <= '0', '1' after 45 ns, '0' after 250 ns;
end rtl;
```

모든 회로 설계는 모방에서 시작합니다. 다른 사람의 설계를 보는 것도 중요합니다. 그렇지만 그 모방에서 벗어나는 것이 전문가가 되는 길입니다. 그래야 만이 자기 자신의 코딩을 할 수 있을 것입니다.

7 Seament

오늘은 세그먼트에 대해서 알아보도록 하겠습니다.
원리는 모두가 알고 있다고 생각됩니다. 그래서 간단하게 설계해 보도록 하겠습니다.
비트의 입력을 받아 이것을 8비트의 출력으로 사용하겠습니다. 7비트라도 상관은 없습니다.
도트는 처리하지 않아도 상관없습니다.
다음은 세그먼트의 간략한 그림입니다.



가장 간단하게 결과를 표시할 수 있는 세그먼트 입니다. 기초적인 단계에서나 세그먼트를 이용할 뿐 실제로는 LCD를 많이 이용하고 있습니다. 전자시계 정도는 세그먼트를 이용해도 충분하다고 생각됩니다. LCD의 구현은 조금 복잡한 구조를 가지고 있습니다. LCD의 구현은 조금 미루도록 하겠습니다.
원래 BCD_data의 입력이 4비트이므로 총 16개의 출력을 표시할 수 있습니다. 0에서 9까지 그리고 a에서 f까지 16개의 출력을 사용합니다. 그러나 전자 시계에서는 0에서 9까지의 출력이면 충분하다고 생각됩니다. 그리고 표시하는 방법에 따라서 AM을 표시하거나 PM을 표시한다면 조금 다릅니다. 세그먼트에서는 A와 P는 충분히 표시할 수 있습니다.
그럼 직접 구현해 보도록 하겠습니다.

ex1)

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

entity segment is
  port ( bcd : in std_logic_vector(3 downto 0);
         y : out bit_vector(7 downto 0));
end segment;

architecture rtl of segment is
begin
    process(bcd)
    begin
```

```
        case bcd is
            when  "0000" => y <= "11111100";
            when  "0001" => y <= "01100000";
            when  "0010" => y <= "11011010";
            when  "0011" => y <= "11110010";
            when  "0100" => y <= "01100110";
            when  "0101" => y <= "10110110";
            when  "0110" => y <= "10111110";
            when  "0111" => y <= "11100000";
            when  "1000" => y <= "11111110";
            when  "1001" => y <= "11100110";
            when others => y <= "00000000";
       end case;
    end process;
end rtl;
```

만약 도트는 사용하지 않는다면 출력을 7비트로 하는 것이 좋습니다. 사용하는 않는 부분은 사용하지 않는 것이 메모리 낭비를 하지 않는 것입니다.

만약 A와 P를 출력하고자 한다면

```
 when "1010" => y <= "11101110" -- A를 출력
 when "1011" => y <= "11001110" -- P를 출력
```

의 문장을 추가하면 됩니다.

다음 예제는 그냥 같은 예제를 조금 바꾸어서 표현했습니다. 다만 입력이 Integer의 형태로 되어 있고 출력이 0에서 f까지입니다.

ex2)

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

entity segment is
   port ( num : in integer range 0 to 15;
            seg : out bit_vector(6 downto 0));
end segment;

architecture rtl of segment is
signal seg_n : bit_vector( 6 downto 0);
begin
          process(num)
                    begin
        case num is
```

```
            when  0 => seg_n <= "1111110";
            when  1 => seg_n <= "0110000";
            when  2 => seg_n <= "1101101";
            when  3 => seg_n <= "1111001";
            when  4 => seg_n <= "0110011";
            when  5 => seg_n <= "1011011";
            when  6 => seg_n <= "1011111";
            when  7 => seg_n <= "1110000";
            when  8 => seg_n <= "1111111";
            when  9 => seg_n <= "1110011";
            when 10 => seg_n <= "1111101";
            when 11 => seg_n <= "0011111";
            when 12 => seg_n <= "1001110";
            when 13 => seg_n <= "0111101";
            when 14 => seg_n <= "1001111";
            when 15 => seg_n <= "1000111";
            when others => seg_n <= "1111110";
        end case;
    end process;
    seg <= seg_n;
end rtl;
```

메모리 낭비를 위해서 도트는 사용하지 않았습니다.
그리고 가장 중요한 부분을 설명하지 않았습니다.

```
when  0 => seg_n <= "1111110";
```

위에서 보면 최상위 비트가 A입니다.
그러나 제일 위에 있는 그림에서는 최하위 비트가 A라고 표시하고 있습니다.
이유는 출력은 반전되어 나오기 때문에 A를 최상위 비트로 표현해야 합니다.
A를 최하위 비트로 표현하면 결과가 이상하게 나올 것입니다.

오늘까지가 VHDL의 기초 강좌입니다. 제가 생각하는 기초적인 지식은 이 정도면 충분하다고 생각됩니다. 이제는 조금 복잡한 회로를 대하고 조금 복잡한 회로를 자기 스스로 설계한다면 많은 발전이 있을 것입니다. 다음부터는 중급 과정으로 시작하겠습니다.
첫 시간은 전자 시계에 대해서 설계하도록 하겠습니다.

중급 과정

오늘부터는 중급 과정입니다. 기초적인 부분은 어느 정도 설명했다고 생각됩니다. 이제부터는 조금씩 큰 예제를 중심으로 설명할 계획입니다. 나의 능력이 미치는 범위까지는 계속 할 생각입니다. 그러나 개인적인 시간 관계상 빨리 업로드하지 못하는 것을 미안하게 생각합니다. 여러분에게 조금 도움이 되고자 시작했지만 그렇게 도움이 되는 지도 확신하지 못하고, 저의 개인적인 생각으로 계속 진행할 생각입니다.

여러분들이 전자시계에 대해서 질문이 많았습니다. 그래서 제 1장에서는 전자 시계를 설계할 생각입니다. 전자 시계를 구현하는 방법은 무수히 많습니다. 자기 자신의 코딩 방식으로 설계하기를 바랍니다. 제가 올리는 것은 하나의 참고적인 자료일 뿐입니다. 제 코딩이 완벽하지는 않습니다. 저보다 더 좋은 생각을 가지고 있다면 그쪽으로 설계하는 것이 좋을 것 같습니다. VHDL에는 완벽이란 존재할 수 없습니다. 상업성, 안전성 등 여러 요소들을 복합적으로 만족하는 회로를 설계하는 것이 좋습니다.

그럼 본론으로 들어가겠습니다. 우선 무엇보다 먼저 고려해야 할 것이 있습니다. 어떤 식으로 설계할 것인가? 하는 것입니다. 먼저 세그먼트를 몇 개를 사용할 것인가? 또 12시간씩 표현할 것인가? 아니면 AM과 PM을 사용해 설계할 것인가? 어떤 기능을 추가할 것인가? 예를 들자면 스톱워치의 기능을 포함할 것인지 아니면 날짜의 기능을 추가할 것인지 하는 세부적인 요소들입니다. 이런 부분들은 개인이 생각해야 하는 요소들입니다.

모듈에 대해서 먼저 설명해야 할 것입니다. 전자 시계를 구현하는 되에는 세 가지의 모듈이 필요합니다. 60초, 60분, 12시간 또는 24시간 등. FPGA로 설계한다면 세 가지의 모듈이 필요합니다. 모드 6 카운터, 모드 10 카운터, 그리고 모드 12 카운터가 필요합니다. 1분을 계산하기 위해서는 모드6과 모드 10이 필요합니다. 그래야 1분이 표현됩니다. 그리고 1시간을 표현하기 위해서는 60분이 필요합니다. 그러면 모드 6과 모드 10이 필요합니다. 그리고 마지막으로 12시간을 표현하기 위해서는 모드 12가 필요합니다. 24시간으로 표현한다면 모드 12가 두 개 필요합니다. 이것은 개념을 알기 위해서 FPGA로 설계할 때 필요한 내용입니다.
이해를 돕기 위해서 필요합니다.

시계 설계에 있어서 중요한 두 가지는 주파수 변환과 세그먼트 디스플레이가 필요합니다. 앞에서 주파수 변환회로는 설명했습니다. 테스트 보드에 테스트하기 위해서는 고정된 주파수를 1hz의 주파수로 변환해야 합니다. 그리고 스톱워치의 설계를 위해서는 100hz의 주파수가 필요합니다. 시계를 디스플레이하기 위해서는 세그먼트에 이것을 표시해야만 시계가 정상적으로 움직이는지 확인할 수 있습니다.

ex1) 기본적인 모델

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

entity clock is
  port (clk, reset : in std_logic;
        seg_s1, seg_s2 : out std_logic_vector(7 downto 0));
end clock;

architecture rtl of clock is
    signal cnt  : integer range 0 to 255;
    signal sec  : std_logic;
    signal t1sec, t10sec : integer range 0 to 255;

     function seg ( display : integer range 0 to 255)
     return std_logic_vector is
     variable seg1 :  std_logic_vector(7 downto 0);
     begin
      case display is
        when 0 => seg1 := "11111100";
        when 1 => seg1 := "01100000";
        when 2 => seg1 := "11011010";
        when 3 => seg1 := "11110010";
        when 4 => seg1 := "01100110";
        when 5 => seg1 := "10110110";
        when 6 => seg1 := "10111110";
        when 7 => seg1 := "11100000";
        when 8 => seg1 := "11111110";
        when 9 => seg1 := "11100110";
        when others => seg1 := "00000000";
      end case;
     return seg1;
     end seg;

begin
 process(clk, reset)
 begin
   if (reset = '1') then
       cnt <= 0;
       sec <= '0';
```

```
    elsif (clk'event and clk='1') then
        if (cnt>=49) then
            cnt<=0;
            sec <= not sec ;
        else
            cnt <= cnt +1;
        end if;
    end if;
 end process;

 process(reset, sec)
 begin
  if (reset='1') then
      t1sec <= 0;
      t10sec <= 0;
  elsif (sec'event and sec='1') then
      if ( t1sec=9 ) then
       t1sec<=0;
          if ( t10sec=9 ) then
           t10sec <= 0;
          else
              t10sec <= t10sec+1;
          end if;
      else
          t1sec <= t1sec + 1;
      end if;
  end if;
 end process;

 seg_s1 <= seg(t1sec);
 seg_s2 <= seg(t10sec);

end rtl;
```

위의 구조를 먼저 살펴봅시다.

```
function seg
... ...
end seg;
process( )
... ...
end process;
process( )
... ...
end process;
seg_s1 <= seg(t1sec);
seg_s2 <= seg(t10sec);
```

이런 구조를 가지고 있습니다. 먼저 function seg는 세그먼트를 LED에 구현하기 위한 설계입니다. 지금은 복잡한 구조를 가지고 있지 않지만 뒤에 점점 복잡한 구조를 가지게 됩니다. 그래서 하나의 function을 이용해 설계하는 것이 효과적입니다. 그리고 회로를 단순하게 만드는 방법입니다. 나중에 살펴볼 전자 시계와 Stop Watch의 LED 구현을 동시에 수행할 수 있게 됩니다. 그리고 첫 번째 프로세서 문은 주파수 변환을 위한 회로입니다. 앞 시간에 설명한 그대로입니다. 위의 예제는 설명을 위해서 세그먼트 두 개를 이용해서 설명했습니다. 그렇지만 세그먼트를 6개를 사용한다고 해도 똑 같은 구조를 가지고 있습니다. 그래서 설명용으로 간단하게 먼저 설계했습니다. 그리고 두 번째 프로세서 문은 00(초기치)에서 99까지 설계했습니다. 99 다음은 00으로 세팅됩니다. 시계에서 99까지는 필요 없습니다. 초 이전까지 설계한다면 필요하겠지요. 0에서 99까지 다음에 1초가 되는 원리이지요. 위의 예제를 조금만 바꾸면 재미있게 설계할 수 있습니다. 9를 5로 바뀌면 59초 또는 59분까지 설계할 수 있습니다.

그리고 signal과 variable의 쓰임새에 대해서도 자세히 살펴보세요. 이것의 용도는 직접 확인해 보시고 다른 방식의 설계도 한 번 생각해 보십시오.

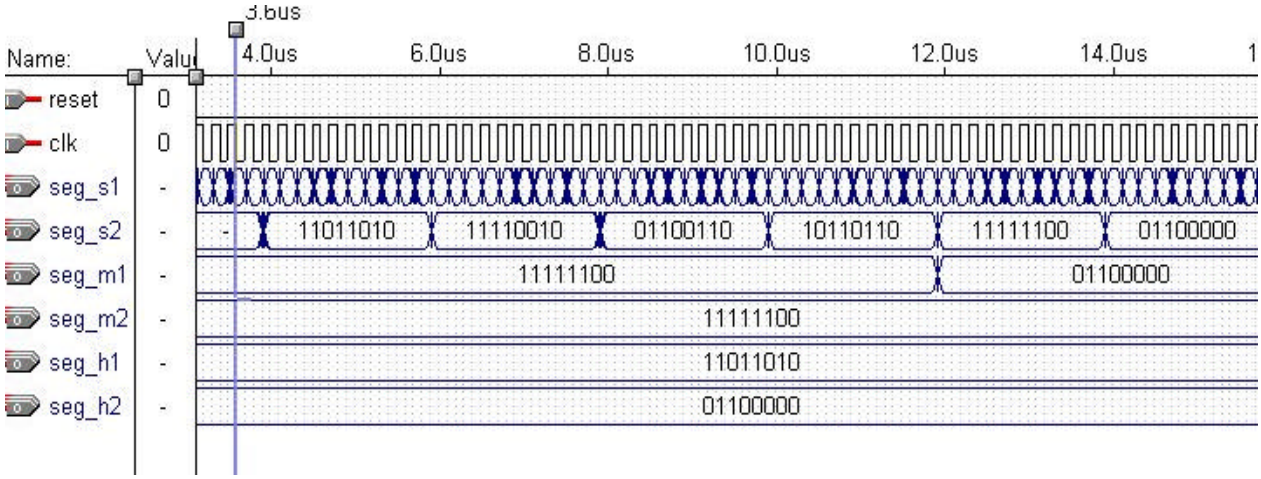
그럼 본격적으로 설계에 들어가도록 하겠습니다. 다음 예제는 세그먼트는 6개를 사용했고 12시간을 기준으로 설계에 들어갔습니다. 리셋은 12 : 00 : 00으로 되어 있습니다. 여기에 추가적으로 세그먼트 한 개를 더 사용해서 A와 P를 시간 앞에 설계한다면 더 확실히 시간을 알 수 있을 것입니다. FPGA로 설계하면 글리치의 발생이 많습니다. 그러나 VHDL로 설계하면 글리치의 발생이 없을 것입니다. 테스터 보드로 테스트할 때 글리치의 발생을 주의해서 살펴보기를 바랍니다.

ex2) 전자시계

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

entity clock1 is
  port (clk, reset : in std_logic;
        seg_s1, seg_s2 : out std_logic_vector(7 downto 0);
        seg_m1, seg_m2 : out std_logic_vector(7 downto 0);
        seg_h1, seg_h2 : out std_logic_vector(7 downto 0));
end clock1;

architecture rtl of clock1 is
    signal t1sec, t10sec : integer range 0 to 255;
    signal m1sec, m10sec : integer range 0 to 255;
    signal h1sec, h10sec : integer range 0 to 255;

     function seg ( display : integer range 0 to 255)
     return std_logic_vector is
       variable seg1 :  std_logic_vector(7 downto 0);
      begin
      case display is
        when 0 => seg1 := "11111100";
        when 1 => seg1 := "01100000";
        when 2 => seg1 := "11011010";
        when 3 => seg1 := "11110010";
        when 4 => seg1 := "01100110";
        when 5 => seg1 := "10110110";
        when 6 => seg1 := "10111110";
        when 7 => seg1 := "11100000";
        when 8 => seg1 := "11111110";
        when 9 => seg1 := "11100110";
        when others => seg1 := "00000000";
      end case;
     return seg1;
     end seg;

begin
 process(reset, clk)
  begin
  if (reset='1') then
```

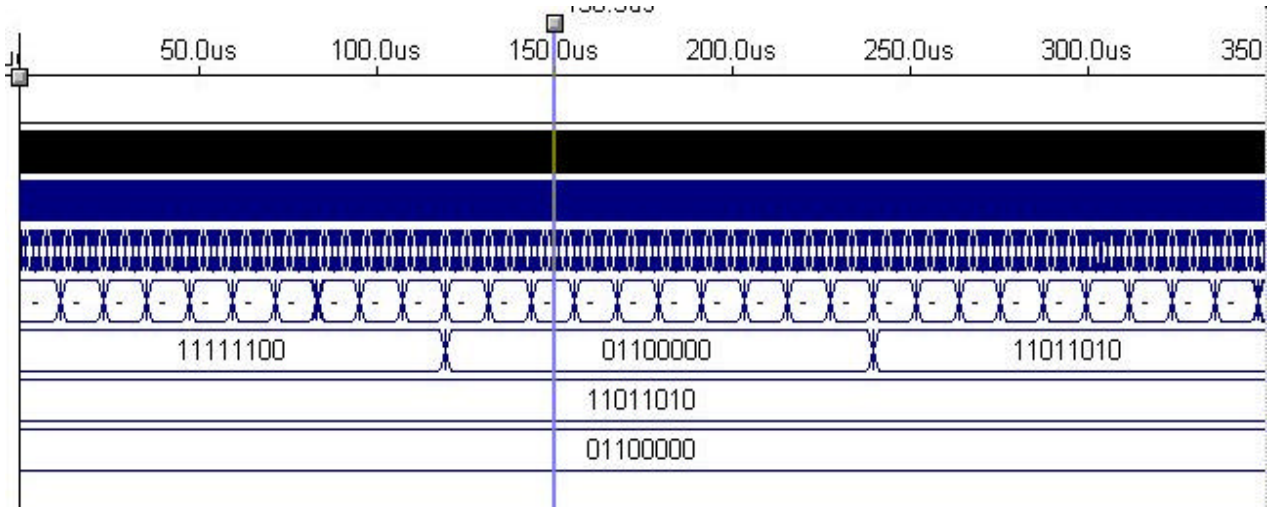
```
      t1sec <= 0;  t10sec <= 0;
      m1sec <= 0;  m10sec <= 0;
      h1sec <= 2;  h10sec <= 1;
  elsif (clk'event and clk='1') then
      if (t1sec=9) then t1sec <= 0;
        if (t10sec=5) then t10sec <= 0;
          if (m1sec=9) then m1sec <= 0;
            if (m10sec=5) then m10sec <= 0;
              if (h1sec=9) then h1sec <= 0;
                if (h10sec=1 and h1sec=2) then h10sec <= 1;
                else h10sec <= h10sec + 1;
                end if;
              else h1sec <= h1sec + 1;
              end if;
            else m10sec <= m10sec + 1;
            end if;
          else m1sec <= m1sec + 1;
          end if;
        else t10sec <= t10sec + 1;
        end if;
      else t1sec <= t1sec + 1;
      end if;
  end if;
 end process;

 seg_s1 <= seg(t1sec);
 seg_s2 <= seg(t10sec);
 seg_m1 <= seg(m1sec);
 seg_m2 <= seg(m10sec);
 seg_h1 <= seg(h1sec);
 seg_h2 <= seg(h10sec);

end rtl;
```

결과 파형을 잠시 보도록 하겠습니다. 시간의 변화 부분에 주목하시고 결과를 살펴보시면 됩니다. 테스터 보드가 없다면 쉽게 파형을 관찰하기 위해서는 세그먼트 부분을 없애고 Decimal code로 파형을 표시하면 더욱 쉽게 관찰할 수 있을 것입니다.



위 시뮬레이션 결과는 seg_s2에서 59초에서 1분으로 넘어가는 과정을 보여주고 있습니다. seg_h1과 seg_h2는 12로 표시되어 있습니다. reset에서 12시로 고정되어 있는 값입니다.

밑에 있는 결과는 조금 확대해서 본 것입니다. 10분, 20분 그리고 30분의 결과를 보여주고 있습니다. 결과를 시뮬레이션으로 모두 표시하기에는 문제가 있습니다. 시간도 많이 소모되고 그 값을 모두 확인하기에는 시간이 너무 소모된다는 단점을 가지고 있습니다.



위와 같은 요령으로 Stop Watch도 간단하게 설계할 수 있습니다. 조금 다른 것이 있다면 초기치가 00 : 00 : 00으로 셋 된다는 것입니다. 다음 Stop Watch는 1시간까지 가능하도록 설계했습니다. 세그먼트 6개를 사용했습니다. 전자 시계와 똑 같은 구조를 가지고 있습니다. 다만 클럭이 100hz라는 것이 조금 다릅니다.

ex3) Stop Watch

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

entity timer is
  port (clk, reset : in std_logic;
        seg_s1, seg_s2 : out std_logic_vector(7 downto 0);
        seg_m1, seg_m2 : out std_logic_vector(7 downto 0);
        seg_h1, seg_h2 : out std_logic_vector(7 downto 0));
end timer;

architecture rtl of timer is
    signal t1sec,t10sec : integer range 0 to 255;
    signal m1sec,m10sec : integer range 0 to 255;
    signal h1sec,h10sec : integer range 0 to 255;

     function seg ( display : integer range 0 to 255)
     return std_logic_vector is
       variable seg1 :  std_logic_vector(7 downto 0);
      begin
      case display is
        when 0 => seg1 := "11111100";
        when 1 => seg1 := "01100000";
        when 2 => seg1 := "11011010";
        when 3 => seg1 := "11110010";
        when 4 => seg1 := "01100110";
        when 5 => seg1 := "10110110";
        when 6 => seg1 := "10111110";
        when 7 => seg1 := "11100000";
        when 8 => seg1 := "11111110";
        when 9 => seg1 := "11100110";
        when others => seg1 := "00000000";
      end case;
     return seg1;
     end seg;

begin
 process(reset, clk)
  begin
  if (reset='1') then
```

```
      t1sec <= 0;  t10sec <= 0;
      m1sec <= 0;  m10sec <= 0;
      h1sec <= 0;  h10sec <= 0;
  elsif (clk'event and clk='1') then
      if (t1sec=9) then t1sec <= 0;
        if (t10sec=9) then t10sec <= 0;
          if (m1sec=9) then m1sec <= 0;
            if (m10sec=5) then m10sec <= 0;
              if (h1sec=9) then h1sec <= 0;
                if (h10sec=5) then h10sec <= 0;
                else h10sec <= h10sec + 1;
                end if;
              else h1sec <= h1sec + 1;
              end if;
            else m10sec <= m10sec + 1;
            end if;
          else m1sec <= m1sec + 1;
          end if;
        else t10sec <= t10sec + 1;
        end if;
      else t1sec <= t1sec + 1;
      end if;
  end if;
 end process;

 seg_s1 <= seg(t1sec);
 seg_s2 <= seg(t10sec);
 seg_m1 <= seg(m1sec);
 seg_m2 <= seg(m10sec);
 seg_h1 <= seg(h1sec);
 seg_h2 <= seg(h10sec);

end rtl;
```

세그먼트를 늘이는 것은 입력을 8개로 만들고 시간에 관계된 부분을 추가하면 됩니다. 어렵게 생각하지 마시고 한 번 추가해 보세요. 그럼 최종적으로 전자 시계를 설계해 보겠습니다.

ex4) 최종적인 전자 시계

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

entity watch is
  port (clk, reset, sel : in std_logic;
        seg_s1, seg_s2 : out std_logic_vector(7 downto 0);
        seg_m1, seg_m2 : out std_logic_vector(7 downto 0);
        seg_h1, seg_h2 : out std_logic_vector(7 downto 0));
end watch;

architecture rtl of watch is

   signal u1sec, u10sec : integer range 0 to 255;
   signal t1sec, t10sec : integer range 0 to 255;
   signal m1sec, m10sec : integer range 0 to 255;
   signal h1sec, h10sec : integer range 0 to 255;

    function seg ( display : integer range 0 to 255)
    return std_logic_vector is
      variable seg1 :  std_logic_vector(7 downto 0);
     begin
     case display is
       when 0 => seg1 := "11111100";
       when 1 => seg1 := "01100000";
       when 2 => seg1 := "11011010";
       when 3 => seg1 := "11110010";
       when 4 => seg1 := "01100110";
       when 5 => seg1 := "10110110";
       when 6 => seg1 := "10111110";
       when 7 => seg1 := "11100000";
       when 8 => seg1 := "11111110";
       when 9 => seg1 := "11100110";
       when others => seg1 := "00000000";
     end case;
    return seg1;
    end seg;

begin
```

```
process(reset, clk, sel)
 begin
 if (reset='1') then
    t1sec <= 0;  t10sec <= 0;
    m1sec <= 0;  m10sec <= 0;
    h1sec <= 2;  h10sec <= 1;
 elsif (clk'event and clk='1') then
   if (sel='0') then
     if (u1sec=9) then u1sec <= 0;
       if (u10sec=9) then u10sec <= 0;
         if (t1sec=9) then t1sec <= 0;
           if (t10sec=5) then t10sec <= 0;
             if (m1sec=9) then m1sec <= 0;
               if (m10sec=5) then m10sec <= 0;
                 if (h1sec=9) then h1sec <= 0;
                   if (h10sec=1 and h1sec=2) then h10sec <= 0;
                   else h10sec <= h10sec + 1;
                   end if;
                 else h1sec <= h1sec + 1;
                 end if;
               else m10sec <= m10sec + 1;
               end if;
             else m1sec <= m1sec + 1;
             end if;
           else t10sec <= t10sec + 1;
           end if;
         else t1sec <= t1sec + 1;
         end if;
       else u10sec <= u10sec + 1;
       end if;
     else u1sec <= u1sec + 1;
     end if;
   else
     if (t1sec=9) then t1sec <= 0;
       if (t10sec=9) then t10sec <= 0;
         if (m1sec=9) then m1sec <= 0;
           if (m10sec=5) then m10sec <= 0;
             if (h1sec=9) then h1sec <= 0;
               if (h10sec=5) then h10sec <= 0;
               else h10sec <= h10sec + 1;
               end if;
```

```
                else h1sec <= h1sec + 1;
                end if;
              else m10sec <= m10sec + 1;
              end if;
            else m1sec <= m1sec + 1;
            end if;
          else t10sec <= t10sec + 1;
          end if;
        else t1sec <= t1sec + 1;
        end if;
      end if;
    end if;
    end process;

  seg_s1 <= seg(t1sec);
  seg_s2 <= seg(t10sec);
  seg_m1 <= seg(m1sec);
  seg_m2 <= seg(m10sec);
  seg_h1 <= seg(h1sec);
  seg_h2 <= seg(h10sec);

end rtl;
```

하나의 구문으로 전자 시계와 Stop Watch를 구현했습니다. 약간의 문제점도 있지만 그 부분은 각자가 수정해 보시고 위의 예제는 100hz를 사용했습니다. 그래서 u1sec과 u10sec를 추가했습니다. 그렇지만 세그먼트에 표시되지는 않습니다.

제어 부분은 설계하지 않았습니다. 그래서 어느 정도의 문제점이 있습니다. 시간을 조절하는 부분도 설계되지 않았습니다. 그 부분은 한 번 생각해 보세요.

다음은 다른 사람들이 설계한 전자 시계를 한 번 살펴보겠습니다.

이 소스는 **ALTERA MAX+PLUSⅡ를 사용한 디지털 시스템 설계**에 있는 내용입니다. 소스만 발췌한 것입니다. 입력은 1MHz이고 세그먼트는 8개를 사용했습니다. 상당히 괜찮은 내용입니다. 세 부분으로 나누어져 있습니다. clock.vhd, dec7.vhd 그리고 sep.vhd로 이루어져 있습니다. dec7은 세그먼트에 디스플레이를 위한 부분이고 sep은 시, 분, 초의 두 자리 수를 분리하는데 사용합니다. clock은 전체적인 코드입니다.

이처럼 각각의 기능을 나누어서 설계하는 것도 상당히 좋은 방법입니다. 그러면 공통되는 부분은 중복해서 설계하는 단점을 피할 수 있습니다. 그리고 파워의 소비 문제도 어느 정도 해결할 수 있습니다.

```
-- dec7.vhd
library ieee;
use ieee.std_logic_1164.all;

entity dec7 is
port ( bcd : in integer range 0 to 15;
       d   : in std_logic;
       seg : out std_logic_vector(7 downto 0));
end dec7;

architecture arc of dec7 is
signal y : std_logic_vector(6 downto 0);
begin
   process(bcd)
   begin
      case bcd is
         when 0 => y <= "1111110";
         when 1 => y <= "0110000";
         when 2 => y <= "1101101";
         when 3 => y <= "1111001";
         when 4 => y <= "0110011";
         when 5 => y <= "1011011";
         when 6 => y <= "1011111";
         when 7 => y <= "1110000";
         when 8 => y <= "1111111";
         when 9 => y <= "1110011";
         when 10 => y <= "1110111";
         when 11 => y <= "0011111";
         when 12 => y <= "1001110";
         when 13 => y <= "0111101";
         when 14 => y <= "1001111";
         when 15 => y <= "1000111";
         when others => y <= "0000000";
      end case;
   end process;
   seg <= y & d;
end arc;
```

먼저 입력과 출력을 살펴봅시다. 입력은 bcd와 d, 출력은 seg으로 이루어져 있습니다. 입력 bcd는 0에서 f까지의 입력이 들어옵니다. 그러면 출력 seg은 이것을 세그먼트로 디스플레이 합니다. 또 입력 d는 세그먼트의 도트를 표시하는 것입니다. 세그먼트의 도트를 사용하면 1

초마다 시간이 흐르고 있다는 것을 표시합니다. 그리고 모든 세그먼트의 도트를 사용할 필요는 없습니다. 하나의 도트만 사용하면 됩니다. 제 생각에는 0에서 9까지만 설계해도 괜찮다고 생각합니다. 그렇게 특별한 기능을 가진 시계가 아니기 때문에 문자까지는 없어도 된다고 생각합니다.

```
-- sep.vhd
library ieee;
use ieee.std_logic_1164.all;
entity sep is
port ( a          : in integer range 0 to 59;
       ten, one : out integer range 0 to 9);
end sep;

architecture a of sep is
begin
   process(a)
   begin
      if (a<=9) then
         ten <= 0;
         one <= a;
      elsif (a<=19) then
         ten <= 1;
         one <= a - 10;
      elsif (a<=29) then
         ten <= 2;
         one <= a- 20;
      elsif (a<=39) then
         ten <= 3;
         one <= a - 30;
      elsif (a<=49) then
         ten <= 4;
         one <= a - 40;
      elsif (a<=59) then
         ten <= 5;
         one <= a - 50;
      else
         ten <= 0;
         one <= 0;
      end if;
   end process;
end a;
```

여기서도 먼저 입력과 출력을 먼저 봅시다. 입력은 a가 있고 출력은 ten과 one이 있습니다. 만약 49분이라는 입력이 들어오면 이것은 ten에서는 4를 출력하고 one에서는 9를 출력하도록 되어 있습니다. 비교회로로 설계되어 있습니다. 깔끔하고 간결하게 되어 있습니다. 그렇게 어려운 부분은 없습니다.

다음은 가장 중요한 부분은 clock 부분입니다.

```
-- clock.vhd
library ieee;
use ieee.std_logic_1164.all;

entity clock is
port (  clk   : in std_logic;
        clear : in std_logic;
        g     : out std_logic_vector(7 downto 0);
        seg7  : out std_logic_vector(7 downto 0));
end clock;

architecture arc of clock is

signal cnt       : integer range 0 to 5;
signal num       : integer range 0 to 15;
signal hour      : integer range 0 to 59;
signal min, sec  : integer range 0 to 59;
signal h10, h1   : integer range 0 to 9;
signal m10, m1   : integer range 0 to 9;
signal s10, s1   : integer range 0 to 9;
signal cnts      : integer range 0 to 499999;
signal s_clk     : std_logic;
signal m_clk     : std_logic;
signal h_clk     : std_logic;

component sep
port ( a         : in integer range 0 to 59;
       ten, one  : out integer range 0 to 9);
end component;

component dec7
port ( bcd : in integer range 0 to 15;
       d   : in std_logic;
       seg : out std_logic_vector(7 downto 0));
end component;
```

```
begin

s_s : sep port map (sec, s10, s1);
s_m : sep port map (min, m10, m1);
s_h : sep port map (hour, h10, h1);
s7  : dec7 port map (num, s_clk, seg7);

process(clk, clear)
begin
    if (clear='1') then
        cnts <= 0;
        s_clk <= '0';
    elsif (clk='1' and clk'event) then
        if (cnts >= 499999) then
            cnts <= 0;
            s_clk <= not s_clk;
        else
            cnts <= cnts + 1;
        end if;
    end if;
end process;

process(s_clk, clear)
begin
    if (clear='1') then
        sec <= 0;
    elsif (s_clk='1' and s_clk'event) then
        if (sec >= 59) then
            m_clk <= '1';
            sec <= 0;
        else
            sec <= sec + 1;
            m_clk <= '0';
        end if;
    end if;
end process;

process(m_clk, clear)
begin
    if (clear='1') then
        min <= 0;
```

```
    elsif (m_clk='1' and m_clk'event) then
        if (min >= 59) then
            h_clk <= '1';
            min <= 0;
        else
            min <= min + 1;
            h_clk <= '0';
        end if;
    end if;
end process;

process(h_clk, clear)
begin
    if (clear='1') then
        hour <= 0;
    elsif (h_clk='1' and h_clk'event) then
        if (hour >= 23) then
            hour <= 0;
        else
            hour <= hour + 1;
        end if;
    end if;
end process;

process(clk)
begin
    if (clk='1' and clk'event) then
        case cnt is
            when 0 => g <= "01111111";
                      num <= h10;
            when 1 => g <= "10111111";
                      num <= h1;
            when 2 => g <= "11011111";
                      num <= m10;
            when 3 => g <= "11101111";
                      num <= m1;
            when 4 => g <= "11110111";
                      num <= s10;
            when 5 => g <= "11111011";
                      num <= s1;
            when others => cnt <= 0;
```

```
        end case;
        cnt <= cnt + 1;
    end if;
end process;
end arc;
```

여기도 먼저 입력과 출력을 먼저 설명하겠습니다. clk는 1MHz의 입력입니다. clear는 초기치로 값을 만들어 줍니다. seg7은 세그먼트에 출력되는 값입니다. 그리고 g는 조금 설명이 필요합니다. 세그먼트의 위치를 지정하기 위한 출력입니다. 즉 세그먼트의 첫 번째 와 두 번째의 시간이, 세 번째와 네 번째에 분이, 다섯 번째와 여섯 번째에 초가 나타나며 point(.) 부분은 1초의 주기로 점등되기 위한 출력 값입니다.

그리고 여려 개의 시그널이 사용되었습니다. 그 용도는 cnt는 결과를 보드상에 표시할 때 사용하는 변수이고, num은 세그먼트의 위치를 지정하기 위해서 사용하는 변수입니다. hour, min, sec는 각각 시, 분, 초를 카운트하기 위한 변수이고 h10, h1, m10, m1, s10, s1은 시, 분, 초의 각 자리를 지정하기 위한 변수입니다. cnts는 1MHz의 입력을 1hz의 입력으로 변환하기 위해 사용된 변수이고 h_clk, m_clk, s_clk는 각각 시, 분, 초가 발생했을 때마다 해당 process 문으로 신호를 보내주기 위한 변수입니다.

```
component sep
port ( a          : in integer range 0 to 59;
       ten, one : out integer range 0 to 9);
end component;

component dec7
port ( bcd : in integer range 0 to 15;
        d    : in std_logic;
        seg : out std_logic_vector(7 downto 0));
end component;
```

먼저 sep.vhd와 dec7.vhd를 먼저 설계했습니다. 위의 component 부분은 그 부분을 사용하기 위한 입력을 적어주는 부분입니다. 그것을 사용하는 부분이 밑에 있는 부분입니다. 사용법은 예전에 설명했습니다.

```
s_s : sep port map (sec, s10, s1); -- 초의 표시
s_m : sep port map (min, m10, m1); -- 분의 표시
s_h : sep port map (hour, h10, h1); -- 시간의 표시

s7  : dec7 port map (num, s_clk, seg7);
```

```
-- 1MHz의 입력을 1Hz로 변환하는 부분
process(clk, clear)
... ...
    elsif (clk='1' and clk'event) then
        if (cnts >= 499999) then
            cnts <= 0;
            s_clk <= not s_clk;
        else
            cnts <= cnts + 1;
        end if;
    end if;
end process;
```

다음 부분은 sec이 59보다 크면 m_clk의 값이 1로 됩니다. 이것은 1분보다 작으면 seg의 값이 1초씩 증가하지만 1분보다 큰 값이면 m_clk의 값을 1로 변환시켜 1분이 증가함을 표시하는 부분입니다.

```
process(s_clk, clear)
... ...
    elsif (s_clk='1' and s_clk'event) then
        if (sec >= 59) then
            m_clk <= '1';
            sec <= 0;
        else
            sec <= sec + 1;
            m_clk <= '0';
        end if;
    end if;
end process;
```

다음 부분도 마찬가지로 59분보다 크면 1시간을 증가시키고 그렇지 않으면 1분을 증가시키는 부분입니다.

```
process(m_clk, clear)
.. ...
    elsif (m_clk='1' and m_clk'event) then
        if (min >= 59) then
            h_clk <= '1';
            min <= 0;
        else
            min <= min + 1;
            h_clk <= '0';
        end if;
```

```
    end if;
end process;
```

다음 부분은 시간을 표시하는 부분이므로 23시 이상은 없으므로 23시 보다 작으면 1시간씩 증가하게 됩니다. 이 다음 부분에 일을 표시하는 부분이 있으면 하루가 증가하게 되겠지요. 날짜와 달의 표시는 조금 생각을 해야 할 것입니다. 작게는 28일 많게는 31일이 있으니까요.

```
process(h_clk, clear)
...  ...
    elsif (h_clk='1' and h_clk'event) then
        if (hour >= 23) then
            hour <= 0;
        else
            hour <= hour + 1;
        end if;
    end if;
end process;
```

다음 부분은 세그먼트의 위치를 표시하는 부분입니다. 클럭이 있을 때 수행이 되므로 반복적으로 계속 수행되는 부분입니다. 클럭이 1Mhz입니다만 파워 소모를 작게 하기 위해서는 100Hz를 사용해도 됩니다. 타이머를 설계한다면 100Hz의 입력이 필요하므로 100Hz를 사용해도 됩니다. 이 부분은 타워 소모를 줄이는 부분도 될 수 있습니다. 잔상의 효과를 이용하는 것입니다. 세그먼트에 계속해서 파워를 공급하면 파워 소모가 심합니다. 그래서 100Hz이면 1초에 100번의 클럭의 입력이 있습니다. 이것을 각각의 세그먼트에 1번씩 주사합니다. 그러면 끊임없이 파워가 소모되는 것이 아니라 1/100번씩 세그먼트가 On됩니다. 그러면 세그먼트에 소모되는 파워를 줄일 수 있습니다. 고급 기법입니다.

```
process(clk)
begin
    if (clk='1' and clk'event) then
        case cnt is
            when 0 => g <= "01111111"; num <= h10;
            when 1 => g <= "10111111"; num <= h1;
            when 2 => g <= "11011111"; num <= m10;
            when 3 => g <= "11101111"; num <= m1;
            when 4 => g <= "11110111"; num <= s10;
            when 5 => g <= "11111011"; num <= s1;
            when others => cnt <= 0;
        end case;
        cnt <= cnt + 1;
    end if;
end process;
```