# VHDL

## (VHSIC Hardware Description Language)

## *** SHORT MANUAL ***


**담당교수 : 김동욱**

Digital Design & Test Lab.
광운대학교 반도체 및 신소재공학과 참빛관 902호
e-mail : dwkim@daisy.kwangwoon.ac.kr
Phone : 02-940-5167
Fax : 02-919-3940


# 목 차

# 과 제

과제 1 : Positive Edge triggering 4bit Up/Down Counter with Reset, Enable, and Load

과제 2 : 74299, Universal Shift_Register

과제 3 : 74283, 4-Bit Binary Full Adder

과제 4 : Project

# Simulation용 VHDL Tool

Active VHDL
Altera MAX+Plus II

# 1. VHDL의 개요

## 1.1 HDL(Hardware Description Language)의 필요성

- 대형회로를 설계시 기존의 설계 방법 비효율성(시간/비용 증가)
- 복잡한 하드웨어를 쉽게 기술하기 위하여 언어를 사용하는 방법 연구
  - Programming 언어 : 회로의 지연처리, 합성 등의 문제점
  - HDL : VHDL, Verilog HDL, AHPL, IDL, ISPS, UDL/I 등이 연구, 사용

## 1.2 VHDL의 유래

- **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage.
- 미국 국방성에서 집적회로 관련 업체들 상호간에 개발을 위한 정보교환 및 문서화를 위해 개발(하드웨어의 동작을 언어로 표현 및 기술)
- 1987년 IEEE 1076-1987로 불리는 IEEE 표준 HDL 탄생.
- 1993년 IEEE 1076-1993으로 불리는 IEEE 표준 HDL new version 탄생.

## 1.3 VHDL의 응용 분야

- Documentation : 회로의 문서화
- Simulation : 시뮬레이션을 통한 검증
- Synthesis : 합성과정을 통해 실제 하드웨어로 제작

## 1.4 VHDL의 특징

- 표준 하드웨어 설계 언어
  - 설계공유
  - 설계지원 tool 개발 용이
- 우수한 하드웨어 기술 능력
- 언어로서의 유연성
- 디지털 설계
- 문서화

※ 단점

- VHDL 자체는 하드웨어가 아니므로 하드웨어 설계 시 중요한 고려사항인 임계경로, 지연시간 등의 계산이 어렵다.
- 합성 툴 부족 (모든 VHDL 구문을 H/W로 변환할 수는 없슴)

## 1.5 VHDL vs Traditional Design Methods

- 설계
  - Traditional Design Methods : Symbols, Schematics
  - VHDL : Entities, Architectures

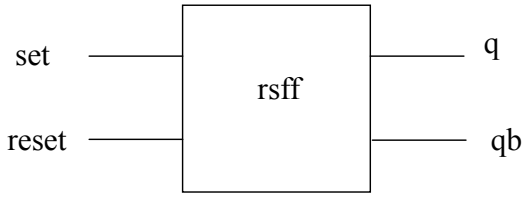
  - Entity : symbol과 같이 회로의 I/O에 대해 정의

```
entity rs_ff is
        port(set, reset : in std_logic ;
                    q, qb : out std_logic);
end rs_ff;
```

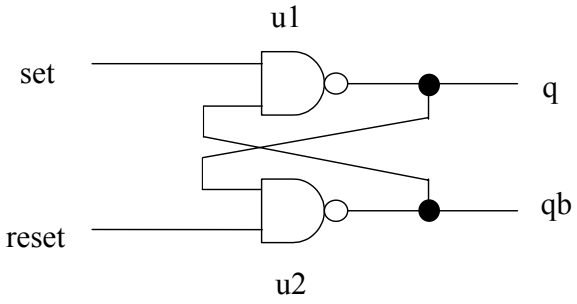
- Architecture : Schematic과 같이 회로의 구체적인 내용에 대해 기술

```
architecture logic of rs_ff is
    begin
        process(set,reset)
            begin
                if set='1' then
                  q<='1';
                  qb<='0';
                elsif reset='1' then
                  q<='0';
                  qb<='1';
                end if;
        end process;
    end logic;
```

- Symbol



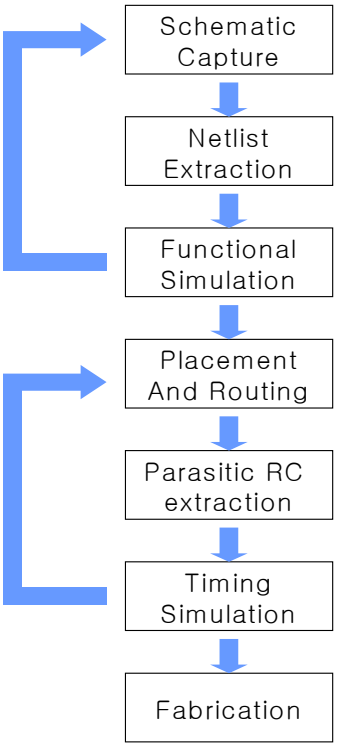

- Schematic



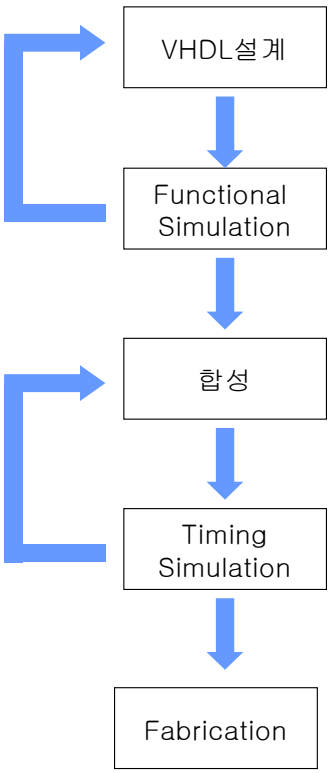

Symbol & Schematic

■ 제작

▪ Traditional method




▪ VHDL



## 1.6 VHDL 표현 방법

- **Behavioral description**
  : 가장 높은 레벨의 추상적인 표현으로써 기능적 또는 수학적인 알고리듬을 사용하여 시스템의 기능을 기술 (기존의 프로그래밍언어 스타일과 가장 유사)
- **Data flow description**
  : 하드웨어를 자료(data)의 흐름 즉 신호 및 제어의 흐름으로 나타내어 기술 (RTL과 유사)
- **Structural description**
  : 표현 방법중 하드웨어에 가장 가까운 하위 레벨의 표현으로써 컴포넌트들의 상호 연결관계를 나타내어 하드웨어를 표현.

※ Behavioral description으로 모든 회로를 설계하는 것이 효과적일 것처럼 보이나, 설계하고자 하는 회로의 크기와 복잡도가 증가할수록 회로를 여러 모듈로 나눈 후 작은 모듈은 structural 또는 behavioral level에서 설계하고, structural level 에서 설계한 회로  의 I/O들을 연결하여 회로를 설계하는 것이 보다 효율적이다.

# 2. VHDL 설계 기초

※ 본 note에서 사용되는 문자 및 특수기호 정의

- [ ] : 필요에 따라 추가/삭제 할 수 있음(optional)
- { } : 필요에 따라 반복할 수 있음(repeatable)
- | : or
- **bold face** : VHDL reserved word(형식 및 예제에만 해당)
- 굴림체 : VHDL 구문
- *italic* : VHDL 구분 중 설계자가 정의하여야 하는 이름

## 2.1 VHDL 구문

■ 주석문

- '--'로 시작.
- '--' 부터 문장 끝('Enter')까지 주석으로 처리

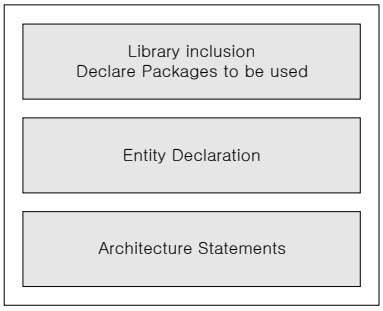
■ 식별자(Identifier)

- VHDL에서 사용되는 variable, signal, label 명
- 대/소문자 구분이 없으며 문자로 시작해야 한다.
- 중간에 특수문자를 사용할 수 없다.

예)

- 올바른 식별자
  - carry_OUT
  - Dim_Sum
  - Count7SUB_2_goX
- 잘못된 식별자
  - 7AB(start with a digit)
  - A@B(special character@)
  - SUM_(end with an underline)
  - PI__A(two underline characters)

**※ VHDL 구문의 구성**



## 2.2 Entity

■ 형식

```
entity entity_name is
        [generic(generic_list);]
        [port(port_list);]
                entity_declarative_part
[begin
        entity_statement_part]
end[entity_name];
```

■ Entity part : 회로의 I/O에 대해 기술한다.

- generic문은 generic_list를 통해서 설계 parameter를 회로에 전달함으로서, 회로 표현이 특정한 특성을 갖거나 또는 특정한 크기 등에 국한시키지 않고 보다 일반화시키기 위해서 사용한다.

예)
```
entity example is
        generic(delay : time);
        port(input :in   bit;
             output:out  bit);
end example;

architecture logic of example is
        signal y1, y2 : bit;
begin
        y1<=input after delay;
        y2<=input after delay/2;
        output<=y1 and y2;
end logic;
```

※ 위와 같이 선언한 후 위의 예제를 component로 사용하는 다른 설계에서 delay값을 특정한 값으로 결정함으로서 회로특성을 특별한 수정 없이 쉽게 변화시킬 수 있다.

- port문에서는 회로에서 사용될 port 및 port종류에 대해 기술한다.
- I/O port 종류 : **IN, OUT, INOUT, BUFFER**
  - IN : 신호의 입력
  - OUT : 신호의 출력
  - INOUT : 신호의 입력 및 출력(출력되는 신호를 회로 내에서 사용 불가능)
  - BUFFER : 신호의 입력 및 출력(출력되는 신호를 회로 내에서 사용가능)

· LINKAGE : 동작에 영향을 주지 않고 port로서 연결된 경우

▪ entity_declarative_part에서는 architecture에서 사용할 object 및 subprogram 등을 선언한다. 여기서 선언하는 이유는 한 개의 entity가 여러 개의 architecture를 가질 수 있는데 여기서 선언된 내용들은 모든 architecture내에서 사용 가능하기 때문이다. 따라서 여러개의 architecture를 사용할 경우 architecture를 사용할 때마다 선언해야 하는 불편을 줄일 수 있다. entity_declarative_part에서 정의될 수 있는 것들은 다음과 같다.

· data type
· subtype
· constant
· signal
· subprogram
· attribute
· disconnection
· use clause
· file
· alias

※ 본 note에서는 많이 사용하지 않는 disconnection, file, alias에 대해서는 설명하지 않음

예)

```
entity example is
        port(.....);
        type three_level_logic is ('0', '1', 'X');
end example;
```

▪ entity_statement_part에서는 회로 내에서 사용되는 신호의 검사 및 감시 등을 위해서 사용한다.

예) : RS플립플롭에서 r과 s과 동시에 1이 되는 경우를 감시

```
entity RSFF is
        port(S, R       :in       bit;
             Q,Q_bar    :buffer   bit);
begin
        assert not (S='1' and R='1')
        report "S & R are '1'"
```

## 2.3 Architecture

■ 형식

**architecture** *identifier* **of** entity_name **is**
{architecture_declarative_part}
**begin**
{concurrent statement}
**end** [*identifier*];

- 회로의 내부동작 및 회로의 연결 상태를 나타낸다(한 개의 entity는 여러 개의 architecture를 가질 수 있다. 하지만 시뮬레이션이나 합성시 어떤 architecture를 사용할 것인가를 결정하여 기술해야 한다.).
- begin으로 시작, end로 끝남(begin 과 end 사이에 회로에 대해 기술)
- architecture_declarative_part에서는 architecture내에서 사용할 signal, data type, component등을 선언한다.
  - data type
  - subtype
  - constant
  - signal
  - component
  - configuration
  - subprogram
  - attribute
  - disconnection
  - use clause
  - file
  - alias

※ 본 note에서는 많이 사용하지 않는 disconnection, file, alias에 대해서는 설명하지 않음

- concurrent statement는 실제 회로를 나타내는 부분으로써, architecture 내에 기술된 내용은 모두 병렬(concurrent)적으로 수행된다. 즉 architecture내에서는 순차 처리문들이 기술될 수 없다(순차처리문은 process내에서 기술한다(뒷부분에서 다시 설명)).

## 2.4 자료형태와 객체(Types and Objects.)

- 사용 : *object_name*{, *name*} : **type**[:=*initial_value*];
- Object
  - 특정한 값을 가지고 있는 것들(변수, 상수...)
  - Class(즉 object의 종류)
    - Signal
    - Variable
    - Constant

▪ 모든 object는 어떤 값들을 보유할 것인가를 결정하기 위해 data type이 결정되어야 한다. 예를 들어, 변수 a가 bit 정보를 저장할 목적이라면 a는 bit 혹은 std_logic으로 정의되어야 하며, 정수를 저장할 목적이라면 integer로 정의되어야 한다.

■ Type

▪ VHDL에서는 여러 종류의 data type을 정의하여 사용할 수 있다.

▪ 사용할 수 있는 data type을 종류별로 분류하면 다음과 같다.

• 정수형(Integer type)

예제)

```
type byte is range -128 to 127
```

• 실수형(Real type)

예제)

```
type norm is range 0.0 to 1.0;
```

• 열거형(Enumeration type)

예제)

```
type day is (sun, mon, tue, wed, thu, fri, sat);
```

• 물리형(Physical type)

예제)

```
type length is range 0 to 1E10;
```

• 복합형(Composite type) : 여러 가지 값을 가진다.
• 참조형(Access Type) : 동적 메모리 할당을 위해 사용
• 파일형(File type) : 외부와의 입출력을 위해 사용

▪ 일반적으로 많이 사용되는 data type은 IEEE library내의 std_logic_1164 package 내에 정의되어 있으며 주로 많이 사용하는 data type은 다음과 같다.

• std_logic : 0, 1 뿐만 아니라 unknown, high-impedance등 실제 하드웨어 신호에 가까운 표현을 포함.
• std_logic_vector : multi-bit std_logic을 표현할 때 사용한다.
  · multi-bit의 크기를 표현하는 방법에는 to를 사용하여 오름(ascending)순으로 표현하는 방법과 downto를 사용하여 내림순(descending)으로 표현하는 방법이 있다.

예1) `std_logic_vector(0 to 3);`
예2) `std_logic_vector(3 downto 0);`

※ 위의 예제 모두 4 비트를 표현하지만 예1)은 번호가 낮을수록 weight가 높다. 즉, 예1)에서는 0번 bit가 MSB이며 예2)의 경우는 3번 bit가 MSB이다.

예3) a:**std_logic_vector**(0 **to** 3)
b:**std_logic_vector**(3 **downto** 0)
라고 한 후
a<="1100";
b<="1100";
을 대입하면 a, b 모두 십진수로 12를 나타내나, a(0)은 '1', b(0)은 '0'을 나타낸다.

- bit : 0, 1을 표현
- bit_vector : 멀티 bit를 표현할 때 사용

■ Subtype

▪ type으로 정의된 내용을 이용하여 새로운 date type을 선언

예)

```
type integer is range -2147483648 to 21474833647;
subtype natural is integer range 0 to integer'high
subtype positive is integer range 1 to integer'high
```

■ Attributes : 자료형의 범위, 크기 등 자료형의 추가적인 정보를 표현하기 위해 사용한다. (신호에 대한 attributes는 2.9절에서 설명)

▪ 사용 : *name of data type'attribute* (예: a'event, a'high, a'low....)

▪ 종류

- HIGH : 해당 data type의 upper bound
- LOW : 해당 data type의 lower bound
- LEFT : 해당 data type의 left bound
- RIGHT : 해당 data type의 right bound

예) 0에서 7까지의 값을 갖는 'state'라는 변수의 data type을 다음과 같이 정의하면,

```
type state is (0 to 7)
```

이 data type의 속성은,

```
state'high=7
state'low=0
state'left=0
state'right=7
```

이 된다.

만약 **type** state **is** (7 **downto** 0)로 정의하면

state'high=7
state'low=0
state'left=7
state'right=0

이 된다.

- POS(x) : x값은 data type의 몇번째 인가?
- VAL(x) : x번째 값은 무엇인가?
- LEFTOF(x) : x의 왼쪽 값은?
- RIGHTOF(x) : x의 오른쪽 값은?
- SUCC(x) : x의 다음 값은?
- PRED(x) : x의 이전 값은?

예) **type** day **is** (sun, mon, tue, wed, thu, fri, sat);

| | |
|---|---|
| day'pos(tue) | : tue는 day중에 몇번째? = 2 (position은 '0'부터 count) |
| day'val(3) | : day의 세 번째 값 = wed |
| day'leftof(tue) | : tue의 왼쪽값 = mon |
| day'rightof(tue) | : tue의 오른쪽 값 =wed |
| day'succ(thu) | : thu다음값 = fri |
| day'pred(thu) | : thu이전값 = wed |

이외에 여러 가지 attribute들이 있다.

## 2.5 Library와 Package 사용

■ Package : 자료형(data type), 함수(function), procedure, component 등을 사용할 때마다 선언하지 않고, package내에 기술한 후 package를 다시 사용함으로써 효율적인 설계를 하기 위해 사용. C-언어의 header file과 같은 역할.

▪ 형식

```
package identifier is
        package_declarative_part;
end [identifier];

package body identifier is
        package_body_declarative_part
end [identifier]
```

※ package body가 항상 필요한 것은 아니다(component문 같은 경우 package body가 필요 없음).

▪ 사용 : use문을 사용하여 package를 사용.

※ 예제는 library예제에서 다룸.

■ Library : Package들과 자주 사용하는 회로들을 VHDL을 사용하여 설계한 후 파일로 저장하여 필요시 다시 설계하지 않고 파일에서 불러 사용하기 위해 library를 사용한다.

▪ Library 종류

• Working library

: 현재 회로를 설계하고 있는 내용들로 구성된 library. 따라서 회로설계를 진행함에 따라 working library도 증가하며 working library는 1개만 존재할 수 있다.

• Resource library

: 다른 회로에서 사용이 가능하도록 이미 분석이 끝난 library. Resource library는 여러 개 존재할 수 있다.

▪ 사용 :

• Library 사용

**library** *library_name*;

• Library에 있는 package 사용

**use** *library_name.package_name*.**all**;

▪ 예제)

```
library ieee;                    : ieee library를 사용
use ieee.std_logic_1164.all;     : ieee library의 std_logic_1164
                                   package를 사용
```

## 2.6 연산자(Operator)

| | | | | | | |
|---|---|---|---|---|---|---|
| ■ Logical | and | or | nand | nor | xor | |
| ■ Relational | = | /= | < | <= | > | >= |
| ■ Adding and Concatenation | + | - | & | | | |
| ■ Sign | + | - | | | | |
| ■ Multiplying | * | / | mod | rem | | |
| ■ Exponent, absolute, complement | ** | abs | not | | | |

## 2.7 신호전달(할당)

■ '<=' : PO와 signal에 사용

▪ 지연시간이 지난 후 신호가 전달

▪ 예제)

```
y<=a;
y<='1';
```

■ ':=' : variable에 사용

▪ 지연시간 없이 신호가 전달

▪ 예제)

```
temp:=a;
temp:='1';
```

## 2.8 Delta time

■ 어떤 신호(x)의 변화에 의해서 다른 신호(y)가 변할 때 같이 변화하는 것 같지만 엄밀히 말해서 x가 변한 후 y가 변하는 것이다.

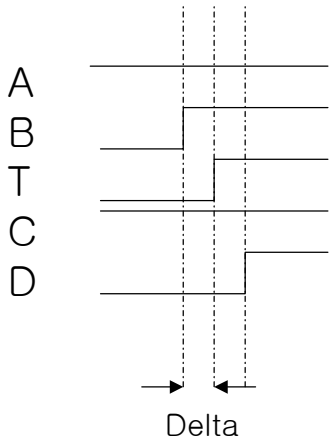
■ x에 의해서 y가 변할 때 일정한 시간(delta time)이 지연된 후 변하게 된다.

■ 예제)

▪ VHDL coding

```
architecture DELTA of NANDXOR is
        signal T : bit;
begin
        T<=A and B;
        D<=not(T xor C);
end DELTA;
```

▪ Simulation결과




▪ B가 변함에 따라 T가 변할 때 바로 변하지 않고 delta time이 지난 후 T가 변한다. T의 변화에 의해 D가 변할 때도 delta time이 지난 후 변하게 된다.

## 2.9 Concurrent and Sequential statements

■ Architecture 내의 구문(statement)들은 크게 병렬(concurrent)문과 순차(sequential)문으로 나눌 수 있다.

■ Sequential 문은 high level programming language와 같이 구문들이 순차적으로 수행된다. 따라서 구문들의 순서가 중요하다.

■ Concurrent 문은 구문들의 순서에 상관없이 특정한 신호의 변화에 의해 수행된다.

■ VHDL 코딩시 중요한 규칙들

- 병렬문만이 arhitecture내에 존재 할 수 있다.
- 순차문은 procedure, function, process문 안에만 존재할 수 있다.
- 병렬문의 정보교환은 signal로 한다(variable은 불가능).
- variable은 procedure, function, process문 안에서만 사용할 수 있다.

■ 예제1)

```
architecture logic of test is
begin
        temp1<=a and b;        -①
        temp2<=c and d;        -②
end logic;
```

예제 1)에서 ①, ②는 병렬문이다. 즉 a, b, c, d신호 변화에 따라서 ②가 ①보다 먼저 수행 될 수 있다.

■ 예제2)

```
architecture logic of test is
begin
        process(a,b,c,d)
                temp1<=a and b;        -①
                temp2<=c and d;        -②
        end process;
end logic;
```

예제 2에서 ①, ②는 순차문이다. 즉 따라서 ① 이 수행된 후 ②가 수행된다.

## 2.10 Signal-Valued Attribute와 Signal-Related Attributes

■ data type의 attribute와는 달리 signal이 실제 가지고 있는 값에 대한 정보를 표현하는데 사용한다.

- Transaction과 관련된 attributes
  - SIG'active : Transaction
  - SIG'quiet : Transaction이 일어나지 않으면 true
  - SIG'quiet(T) : T시간동안 신호에 transaction이 일어나지 않

으면 true

- SIG'transaction : Tansaction이 발생되면 이전 값의 signal은 새로운 값(이전 값의 보수)을 취한다.
- SIG'last_active : Transaction이 발생되면 참.

▪ Event와 관련된 attributes

- SIG'event : Event가 일어나면 true
- SIG'stable : Event가 일어나지 않으면 true.
- SIG'stable(T) : T시간동안 신호에 event가 일어나지 않으면 true
- SIG'last_event : 최근의 event가 일어난 후 경과된 시간을 나타낸다.
- SIG'last_value : event가 일어나기 전의 값

▪ 기타 attributes

- SIG'delayed(T) : T시간 만큼 지연시킨다.

※ 위에서 SIG는 signal, T는 시간을 각각 나타낸다.

■ 예제1)

```
if clk='1' and clk'event then
        y<=i;
end if;
```

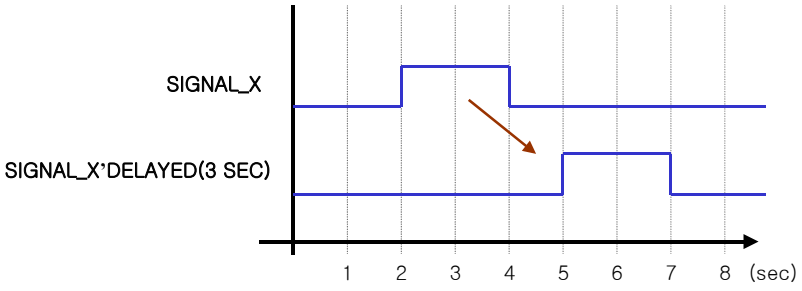
→ clock의 rising edge에서 i값이 y에 대입된다.

■ 예제2)

```
if clk='1' then
        y<=i;
end if;
```

→ clock이 1인 동안 i값이 y에 대입된다.

■ 예제3)



## 2.11 Configuration

- 1개의 entity는 여러 개의 architecture를 가질 수 있는데, 합성이나 시뮬레이션을 수행할 경우 어떤 architecture를 사용할 것인가를 결정해야 하며, 이때 configuration문을 사용한다.
- 여러 개의 architecture를 사용하는 경우를 예를 들면, 첫 번째 architecture는 시뮬레이션을 위하여 지연시간정보를 포함하고 두 번째 architecture는 합성을 위한 구문만을 사용하는 경우다.
- 형식

```
configuration identifier of entity_name is
        {use_clause|attribute_specification}
        {block_configuration}
end identifier;
```

여기에 사용되는

```
        {use_clause|attribute_specification}
        {block_configuration}
```

의 형식은 다음과 같다.

▪ block_configuration::=

```
        for block_sepcification
                {use_clause}
                {block_configuration|component_configuration)
        end for;
```

※ 본 note에서는 block_specification, component_configuration, binding_indication, component_specification에 대한 형식은 생략함.

- 예)

DFF라는 이름을 가진 entity에 3개의 architecture들(RTL1, RTL2, RTL3)이 존재할 때 RTL2를 사용하고자 하면 다음과 같은 문을 첨가한다.

```
configuration CFG_RTL_DFF of DFF is
        for RTL2
        end for;
end CFG_RTL_DFF;
```

# 3. 순차문

## 3.1 process

- Architecture내에서 순차문들은 process문 내에 존재한다.
- process간에는 서로 concurrent(병렬수행)한 성질을 유지한다.
- 형식

```
[process_label:]process[(sensitivity_list)]
                    process_declaration_part
            begin
                    sequence of sequential statements
            end process[process_label];
```

- sensitivity list
  - sensitivity list는 언제 process가 수행되는지를 결정한다. 즉 sensitivity list에 기술된 신호가 변화하면 process가 수행된다.
  - sensitivity list는 process문 바로 다음에 기술되거나 begin과 end사이에서 wait 문을 이용하여 기술할 수 있다.
  - sensitivity list의 구성 요소에 따라 합성결과가 달라질 수 있으므로 주의해야 한다. 즉, 언제 process 내의 statement가 수행되는가를 정확히 파악해야 한다.
  - 예) 다음 두 예는 동일한 효과를 나타낸다.

① :
```
process(a,b)
begin
        ....
end
```

② :
```
process
begin
        ....
        wait on a,b;
end
```

※ wait 문에 대한 자세한 내용은 뒷부분에 다시 설명함.

- process_label
  - process를 구별하기 위해 사용하며, 선택사항임
- process_declaration_part
  - process내에서 사용될 변수(local variable)를 선언한다.

## 3.2 variable

- 형식

```
target variable:=expression;
```

▪ 예) temp:='1';
temp:=a;

- variable은 process와 subprogram(procedure, function)의 선언부에서 선언되어야 하고, process와 subprogram내에서만 사용할 수 있다.
- 지연시간 없이 새로운 값이 즉시 전달된다.
- 예제)

```
process(a,b)
        variable y1,y2:bit;
begin
        y1:=a;
        y2:=b;
end process;
```

### 3.3 Signal

- 형식

```
target variable<=value_expression;
```

- signal은 architecture의 선언부에서 선언되어야 하고 architecture내에서는 어디든지 사용 가능하다.
- 신호 할당시 after문을 사용할 수 있다. After문은 특정한 시간이 지난 후 신호가 전달(대입)되도록 한다.
  ▪ 예제) y<=a after 10ns;
  : 10ns 지난 후 y에 a가 대입됨.
  ▪ after문은 simulation tool에서는 지원되는 반면 대부분의 합성tool에서 지원되지 않는다.
- 예제)

```
architecture logic of test is
        signal A, B : bit;
begin
        y1<=a;
        y2<=b;
end logic;
```

### 3.4 IF

- 형식

```
if condition then
        sequence of sequential statements
{elsif condition then
        sequence of sequential statements }
[else
        sequence of sequential statements ]
end if;
```

■ 예제1)

```
if a='1' then
        y<=i;
end if;
```

■ 예제2)

```
if b='1' then
        y<=i1;
else
        y<=i2;
end if;
```

■ 예제3)

```
if sel="00" then
        y<=i1;
elsif sel="01" then
        y<=i2;
elsif sel="11" then
        y<=i3;
else
        y<=i4;
end if;
```

※ 상수 할당시 single bit의 경우 ‘ ’, multi-bit의 경우는 “ ” 사용

## 3.5 CASE

■ 형식

```
case expression is
        when choice(s)=>
                sequence of sequential statements
        {when choice(s)=>
                sequence of sequential statements }
end case;
```

■ 예제)

```
case sel
        when "00" =>
                y<=i1;
        when "01" =>
                y<=i2;
        when "10" =>
```

```
            y<=i3;
        when "11" =>
            y<=i4;
        when others =>
            null;
    end case;
```

- CASE문을 사용할 때 'when others'를 적절히 사용하여야 한다. 위의 예제에서 sel은 00, 01, 10, 11등 이상적인 값 뿐 만 아니라 경우에 따라서 XX, ZZ, 1X, 1Z... 등 많은 다른 값들을 가질 수 있다. 따라서 이런 값들을 가질 경우 어떻게 처리해야 하는지 정의되지 않는다면 회로가 원하지 않는 동작을 할 수도 있다. 이럴 경우 when others문을 사용해서 원치 않는 신호가 들어왔을 때 어떻게 동작해야 하는지를 기술함으로써 회로의 잘못된 동작을 예방할 수 있다.
- IF 와 CASE의 차이점

: IF는 priority를 가지고 있고 case는 가지고 있지 않다. 따라서 IF의 경우 statement문 순서에 따라 시뮬레이션 및 합성결과가 달라지는 반면 CASE는 시뮬레이션 및 합성 결과가 statement문 순서와 상관없다.

▪ 예제) 다음 두 예의 결과는 서로 다름

```
if a='1' then
        y<='1';
elsif b='1' then
        y<='0';
end if;

if b='1' then
        y<='0';
elsif a='1' then
        y<='1';
end if;
```

## 3.6 LOOP

- 형식

```
[loop_label:][while condition | for identifier in discrete_range] loop
        sequence of sequential statements
end loop [loop_label];
```

- Loop label은 선택사항
- while : 조건을 만족하는 동안 loop내의 statements 수행
- for : 일정한 횟수만큼 loop내의 statements수행

■ 예제 1 : while문

```
while (check=1) loop
        sum:=sum+1;
end loop;
```

■ 예제 2 : for문

```
for i in 1 to 10 loop
        i_squared(i) := i*i;
end loop;
```

### 3.7 NEXT

■ 형식

**next**[*loop_label*][**when** condition];

■ Loop와 함께 사용되며, 조건을 만족할 경우 해당 loop의 next 이후는 수행되지 않고 그 loop의 수행을 끝낸다.

■ 예제)

```
loop_a:for j in 1 to 10 loop
        sequential statements
        loop_b:for i in 1 to 10 loop
                sequential statements
                next when (i=j);
        end loop;
end loop;
```

※ 이 예제에서 특정 j값(ex: 3)에서 loop_b로 들어갔을 때 i값이 j값과 같아지면, next문에서 loop_b의 나머지 동작(i=5에서 10까지)을 수행하지 않고 loop_b를 빠져나와 j=4의 loop_a를 수행한다.

### 3.8 EXIT

■ 형식

**exit** [*loop_label*] [**when** condition];

or **if** condition **then exit** [loop_label];

■ Loop와 함께 사용되며 해당 loop를 강제로 끝마칠 때 사용한다.

■ 예제1)

```
test1:
loop
        a:=a+1;
        b:=b+1;
        if a>10 then
                exit test1;
        end if;
end loop;
```

■ 예제2)

```
test1:
loop
        a:=a+1;
        b:=b+1;
        exit when a>10;
end loop;
```

### 3.9 NULL

■ 아무런 동작을 하지 않음

■ 예)

```
case sel
        when "00" =>
                y<=i1;
        when "01" =>
                y<=i2;
        when "10" =>
                y<=i3;
        when "11" =>
                y<=i4;
        when others =>
                null;
end case;
```

### 3.10 ASSERTION

■ 설계자가 의도한 대로 설계가 되었는지 확인하기 위해 사용자의 message를 출력할 수 있는 기능이다. 즉 특정 variable 혹은 signal 값들을 조사하여 잘못된 값들이 인가됐을 때 화면에 특정 message를 출력한다.

■ 형식

```
assert condition [report expression] [severity expression]
```

■ condition이 거짓인 경우 report 다음에 기술된 expression과 severity 다음에 기술된 expression이 화면에 출력된다.

■ report는 화면에 특정 문자(문자열)를 출력할 때 사용하며, 생략됐을 경우 default값인 "Assertion violation"이 화면에 출력된다.

■ severity는 error의 중요도를 나타내며 severity expression(level)은 미리 정의된 (NOTE, WARNING, ERROR, FAILURE)중 한 개를 사용해야 한다. 정의하지 않을 경우 default값은 ERROR다.

- severity level 즉, NOTE, WARNING, ERROR, FAILURE에 따른 조치 및 행동은 VHDL 규정에 포함되지 않는다. 즉, 위의 level 에 따른 행동은 시뮬레이션 tool 혹은 설계자에게 달려 있다.

■ 시뮬레이션만을 위한 statement이며 합성되지 않는다

■ 예제)

```
entity rs_ff is
        port(s,r :in    bit;
             q :out   bit);
end rs_ff;
architecture logic of rs_ff is
begin
        assert not(s='1' and r='1') report  "Both s and r are '1'"
                        ...
end rs_ff;
```

※ RS F/F에서는 r과 s가 1이 되어서는 안된다. 따라서 assert 문으로 s와 r이 동시에 1인지를 감시하고 만약 그렇다면 시뮬레이터에 'Both s and r are 1'이라고 표시한다.

## 3.11 WAIT

■ 형식

- **wait on** sensitivity_list : sensitivity_list의 신호가 변할 때까지 process문 수행을 중지한다.
- **wait for** time_expression : 특정한 시간동안 process문 수행을 중지한다.
- **wait until** boolean_expression : 조건이 참이 될 때까지 process문 수행을 중지한다.

■ process문에서 sensitivity list를 process문 내에서 사용하고자 할 때 wait문을 사용한다.

■ process문 내에서 여러번 wait문이 사용될 수는 있으나, 합성 tool이 합성 할 수 없는 경우가 많다.

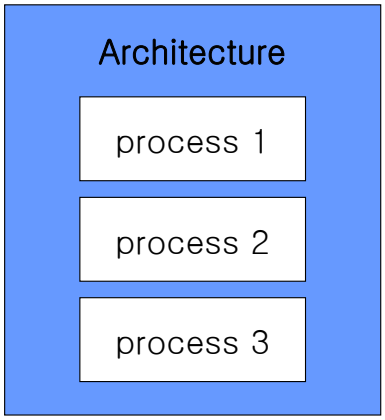
■ 예제)

```
wait on a,b,c;
wait for 10ns;
wait until enable='1';
```

# 4. 병렬문

## 4.1 process

- process문 내의 명령들은 순차적으로 수행되나, process들끼리 또는 process문 외부의 명령들은 병렬로 처리된다.
- 다음과 같이 3개의 process문이 있을 경우 process문 수행은 각 process문의 sensitivity list 변화에 의해 결정된다.




## 4.2 Assertion

- 순차문과 같이 병렬문에서도 assertion문을 사용할 수 있다.

## 4.3 Conditional Signal Assignment Statement

- 특정 조건을 만족할 때 할당문이 수행되며, 순차문의 if와 같이 priority를 가짐
- 형식

  [guarded][transport]{waveform when condition else} waveform;
- 예제)

```
architecture RTL of cnd_stm is
begin
        Y<=A when SEL="00" else
            B when SEL="01" else
            C when SEL="10" else
            D;
end RTL;
```

  - guarded : block문과 같이 사용하며, guarded를 사용한 할당문은 block문의 조건이 참인 경우에만 할당이 수행된다.

    예)

```
architecture logic of example is
```

```
begin
        b1:block(enable='1')
        begin
            c<=guarded '1' after 2 ns when a='1' or b='1' else
                        '0' after 2 ns;
        end block b1;
end logic;
```

※ 위의 예제에서 block조건이 참인 경우 즉, enable='1' 인 경우에만 a와 b의 조건에 따라 c에 '1' 혹은 '0'이 할당된다. 만약 enable='0'이라면 c에 대한 할당은 수행되지 않는다.

▪ transport : 실질적인 지연시간을 기술하는데 사용한다.

・VHDL에서는 두 종류의 지연을 제공한다.

- 관성지연(inertial delay) : 소자의 지연시간보다 작은 입력 지연폭은 무시한다. 즉 소자의 지연시간 보다 큰 입력만 전달한다. VHDL에서 기본(default)지연이다.

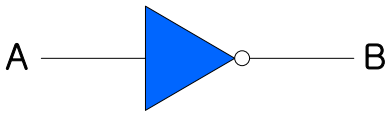
  예) a<='1' after 5ns;

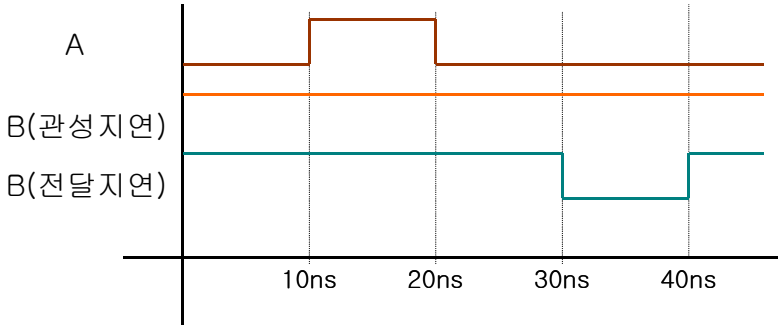
- 전달지연(transport delay) : 입력의 지속시간에 상관없이 그대로 전달한다.

  예) b<=transport '1' after 7ns;

・ 관성지연과 전달지연의 비교

: 아래의 인버터의 소자 지연시간이 20ns라고 가정할 때 입력 신호가 10ns 동안 펄스가 생기면 관성지연에서는 입력을 무시하여 출력에 변화가 없는 반면 전달지연에서는 입력을 출력으로 전달한다.



## 4.4 Selected Signal Assignment Statement

■ 특정 조건을 만족할 때 해당 할당문이 수행되며, 순차문의 case문과 유사하다.

■ 형식

```
with expression select
    target variable<=[guarded][transport]{waveform when choices,}
        waveform choices
```

■ 예제)

```
architecture RTL of sel_stm is
begin
        with SEL select
        Y<=A when "00",
            B when "01",
            C when "10",
            D when others;
end RTL;
```

## 4.5 Component

■ 형식

*instantiation_label*: *component_name* [port_map_aspect]

- instantiation_label은 선택이 아닌 필수이다.
- port_map_aspect에서 component끼리 어떻게 연결되는가를 기술한다. 이 때 연결방법을 기술하는 방식에는 2가지가 있다.
  - positional association
    component내에 기술된 I/O 순서대로 연결시키는 방법
  - name association
    component내에 기술된 I/O 순서와는 상관없이 '=>'를 사용하여 연결시키는 방법('=>'는 신호 흐름 방향을 나타내는 것이 아니라 연결상태를 나타낸다.)

예)

```
entity example is
        port(i1,i2,i3:in   std_logic;
                y :out   std_logic);
end example;

architecture logic of example is
        component and3
                port(a,b,c : in  std_logic;
```

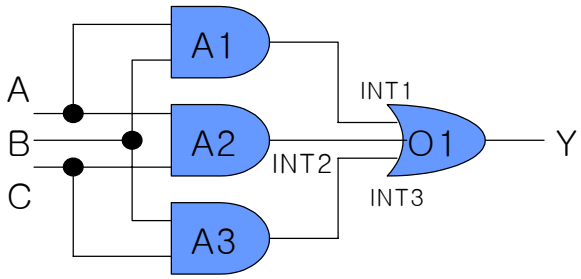
```
                                    z  : outstd_loigc);
            end component;
begin

-- ① position association을 사용하는 경우
     a1:and3 port map (i1, i2, i3, y);
-- ② name association을 사용하는 경우
     a1:and3 port map (y=>z, i1=>a, i3=>c, i2=>b);
end logic;
```

※ ①의 예제에서 순서가 틀리면 안된다. 즉 port map(y, i1, i2, i3)로 기술하면 출력 y가 a와 연결되므로 error를 유발한다.
②의 경우 '=>'는 신호의 흐름 방향을 나타내는 것이 아님에 유이해야 한다. 즉, y>=z에서 실제로 신호는 z에서 y로 흐른다.

- 이미 설계된 회로를 불러와서 입출력 신호의 연결을 정의하여 회로를 설계할 경우 사용한다. Schematic설계에서 hierarchy 구조와 유사하다.
- architecture의 선언부에서 불러서 사용하며, 사용하고자 하는 회로의 entity 이름(대부분의 경우 entity이름=화일이름)과 I/O를 선언해야 한다.
- 불러온 회로의 연결은 'port map'구문을 사용하며 형식은 다음과 같다.
  *instantiation_label* : *component_name* [port_map_aspect];

■ 예제)




위의 예제의 경우 AND2와 OR3게이트를 먼저 설계한다. 설계된 게이트들을 불러서 각 게이트들과 IN/OUT 신호의 연결을 정의해줌으로써 회로를 완성한다.

```
entity comp is
        port(a,b,c :in   bit;
             y     :out bit);
end comp;

architecture structure of comp is
```

```
        component and2
                port(a,b:in bit; y:out bit);
        end component;
        component or3
                port(a,b,c :in bit; y:out bit);
        end component;

        signal INT1, INT2, INT3 : bit;
                a1:and2 port map (A, B, INT1);
                a2:and2 port map (A, C, INT2);
                a3:and2 port map (B, C, INT3);
                o1:and2 port map (INT1, INT2, INT3, Y);
end structure;
```

■ gene

■ generic 문을 사용했을 경우는 generic map을 사용하며, 형식은 port map과 같다. 예를 들어 entity 예제에서 설명했던 아래의 예제를 참고하면,

예)
```
entity example is
        generic(delay : time);
        port(input :in   bit;
             output:out  bit);
end example;
architecture logic of example is
        signal y1, y2 : bit;
begin
        y1<=input after delay;
        y2<=input after delay/2;
        output<=y1 and y2;
end logic;
```

※ 위의 예제를 다른 설계에서 component로 불러 사용할 때 우선 아래와 같이 component 선언을 하며

```
component example
        generic(delay:time);
        port(input:in bit; output:out bit);
end component;
```

선언 후 port map 과 generic map을 사용해서 아래와 같이 설계한다.

```
u2: example generic map(delay=>2)
            port map (x=>input, y=>output);
```

즉 delay에 2ns가 대입되어 시뮬레이션 된다.

## 4.6 Generate문

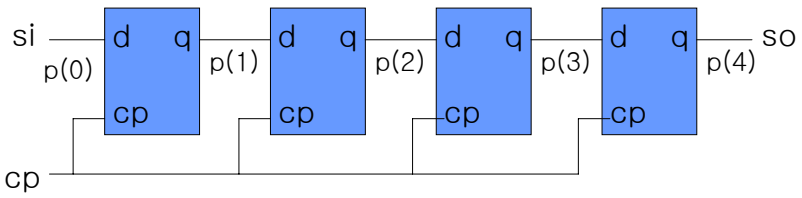
■ 형식

```
generate_label : for generate_parameter_specification generate
        concurrent statements
end generate[generate_label];

generate_label : if condition generate
        concurrent statements
end generate[generate_label];
```

■ 규칙적인 구조를 갖는 하드웨어를 설계하는 경우처럼 component를 반복적으로 연결하여 사용할 경우 generate문을 사용하면 쉽게 코딩을 할 수 있다. 즉 generate문은 component들을 자동으로 연결하여 준다.

■ 예제) : D-F/F를 이용한 shift register




```
entity shifter is
        port(si, cp :in bit;
             so     :out bit);
end shifter;
```

▪ Component문만 사용할 경우

```
architecture logic of shifter is
        signal p : bit_vector(0 to 4);
        component dff
                port(d, cp:in bit; q:out bit);
        end component;
begin
        df1:dff port map (si, cp, p(1));
        df2:dff port map (p(1), cp, p(2));
        df3:dff port map (p(2), cp, p(3));
        df4:dff port map (p(3), cp, so);
end logic;
```

▪ generate문을 사용할 경우

```
architecture logic2 of shifter is
begin
        p(0)<=si;
        g1:for i in 0 to 3 generate
                dfx : dff port map(p(i), cp, p(i+1));
        end generate g1;
        so<=p(4);
end logic2;
```

- component문의 연결 횟수가 커질수록 generate 문을 사용하면 코딩이 쉽다.

## 4.7 Block

- 형식

```
block_label : block[(guard_expression)]
        block_header
        block_declarative_part
begin
        concurrent statements
end block[block_label];
```

- 특정 부분을 block화하기 위해 사용한다(block화 자체는 회로에 특별한 영향을 미치지 않는다).
- Block문 내에서 선언된 자료형(type), 신호(signal), component등은 사용 가능 범위(scope)가 block문 내로 제한된다.
- guarded문을 사용하면 특정 조건을 만족할 경우에만 block 안의 내용들이 수행되도록 할 수 있다.
- 예제)

```
architecture logic of or is
begin
b1:block(enable='1')
   begin
        S1<=guarded
                '1' after 2ns when A='1' or B='1' else
                '0' after 2ns;
    end block b1;
end logic;
```

# 5. Subprogram 및 Package

## 5.1 Subprogram

- 반복되는 동작을 기술하는 경우 subprogram으로 만든 후 불러서 사용함으로써 코딩을 보다 쉽게 할 수 있다(Structural description에서 이미 설계된 회로를 component로 불러서 회로를 설계하는 것과 유사한 개념).
- 회로 설계시 subprogrm을 먼저 기술한 후 사용하거나 미리 package내에서 기술한 후 해당 package를 불러서 사용한다(package를 사용할 경우 subprogram을 사용할 때마다 기술해야 하는 불편이 없다.).
- Subprogram은 function 혹은 procedure로 구성된다.
  - Procedure
    - 여러 개의 값을 반환할 때 사용
    - 매개변수는 in, out, inout이 가능하다.
  - Function
    - 1개의 값만을 반환할 때 사용
    - 매개변수는 in만 가능하다.
- Subprogram내의 모든 statement들은 순차적으로 처리된다.
- Subprogram은 architecture, process, block, package, subprogram등에서 선언(기술)되며 그에 따라 사용범위가 결정된다. 즉 architecture 선언부에서 선언되면 architecture내에서는 어디에서든지 사용이 가능하며, process 선언부에서 선언되면 process내에서만 사용이 가능하다.
- Subprogram 구성
  - 선언(declaration)
    - subprogram의 이름 및 매개변수, 매개변수 종류 등을 기술한다.
    - Package내에서 subprogram을 기술할 경우 사용하며 그 외에서는 기술하지 않는다.
    - 형식

```
procedure procedure_name [(formal_parameter_list)]
function function_name[(formal_parameter_list)] return type
```

  - 몸체(body)
    - subprogram내용을 기술한다.
    - 형식

```
procedure procedure_name (I/O list) is
        declarative_part
begin
        sequential statements
end [procedure_name];

function function_name (input list) return type is
```

```
                declarative_part
        begin
                sequential statements
        end [function_name];
```

▪ 사용

*subprogram_name*(*variables*);

• variables를 matching시키는 방법에는 component에서 사용하는 방법처럼 name assignment와 position assignment가 있다.

■ 예제)

▪ function

```
architecture logic of fnt is
        function rising_edge(signal S : std_logic)
                return boolean is
        begin
                if(s'event) and (s='1') and (s'last_value='0') then
                        return true;
                else
                        return false;
                end if;
        end rising_edge;
begin
        process(clock)
        begin
                if rising_edge(clock) then
                        y<=a;
                end if;
        end process;
end fnt;
```

▪ procedure

```
architecture logic of proc is
        procedure andor (
                signal A, B, C, D : in bit;
                signal Y1,Y2      : out bit) is
        begin
                Y1<=(A and B) or (C and D);
                Y2<=A and C;
        end andor;
```

```
begin
        process(i1,i2,i3,i4)
        begin
                andor(i1,i2,i3,i4,a,b)
        end process;
end andor;
```

## 5.2 Package

- package의 목적은 자주 사용하는 data type, component, subprogram등을 하나의 package화시킨 후 필요할 때마다 package를 불러서 사용함으로써 코딩을 보다 쉽게 하기 위해서이다. C언어의 header file과 비슷한 역할을 한다.
- 구성
  - 선언

```
package identifier is
        package_declarative_part
end [identifier];
```

  - 몸체

```
package body identifier is
        package_body_declarative_part
end [identifier];
```

- Package 형식은 위에서 보는바와 같이 선언부와 몸체(body)로 구성되어 있다.
- 선언부에서는 외부에서 필요로 하는 사항들을 기술한다. package를 만드는 목적이 다른 설계에서 package에 있는 내용을 사용하기 위해서이므로 선언부에서는 외부에서 직접 access가 필요한 내용에 대해 기술한다. 주로 data type, constant, subprogram 선언부 등을 이곳에서 기술한다.
- body에서는 외부에서 직접 필요로 하지 않고 단지 선언부를 보충하거나 원하는 동작 자체를 기술한다. 예를 들어 package에 있는 subprogram을 외부에서 불러서 사용하고자 한다면 우리는 subprogram 이름과 매개변수를 사용해서 subprogram을 이용한다. 따라서 설계자 입장에서는 subprogram이름과 매개변수, 그리고 결과를 이용할 수만 있으면 된다. subprogram 내부가 어떻게 기술되어 있는가는 중요하지 않으며 subprogram 내부를 외부에서 직접 access할 필요가 없다. 이와 같이 body에서는 외부에서 직접 access가 필요하지 않으며, 선언부에서 기술된 것들을 보충할 필요가 있는 내용들을 기술한다. 주로 subprogram body를 이곳에 기술한다.
- package를 compile하면 기본적으로 work library내에 자동으로 기술된다. 따라서 package 완성 후 package 내의 내용을 사용하고자 한다면 use 문을 사용한다. 즉, “use work.*package_name*.all"을 VHDL 코드 내에 첨가하면 된다.

예) 만약 package이름이 test라면 test package를 사용하기 위한 문장은 다음과 같다.

```
use work.test.all;
```

■ 예제1)

① component의 선언부만 필요한 경우

```
package logic_ops is
        component and2
                port (a, b : in bit; z : out bit);
        end component;
        component or3
                port(a, b, c : in bit; z : out bit);
        end component;
end logic_ops;
```

② 특정 설계의 선언부 중 type만 필요한 경우

```
package var_pack is
        type bus_stat_vec is array(0 to 7) of integer;
        type day is (mon, tue, thu, wed, thu, fri, sat);
end var_pack;
```

■ 예제2)

① package 선언부 및 body모두 필요한 경우

```
package basic_utilities is
        function fgl(w, x, gl : bit) return bit;
end basic_utilities;

package body basic_utilities is
begin
        function fgl(w, x, gl : bit) return bit is
        begin
                return (w and gl) or (not x and gl) or (w and not x);
        end fgl;
end basic_utilities;
```

# 6. 조합회로

## 6.1 AND2 gate

- truth table

| a | b | y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- VHDL 설계
  - Entity

```
entity and2 is
        port (a,b :in     std_logic;
              y :out   std_logic);
end and2;
```

  - Architecture 1

```
architecture logic of 2and is
begin
        process(a,b)
                if a='1' and b='1' then
                        y<='1';
                else
                        y<='0';
                end if;
        end process;
end logic2;
```

  - Architecture 2

```
architecture logic2 of and2 is
begin
        y<=a and b;
end logic2;
```

## 6.2 OR2 gate

■ truth table

| a | b | y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

■ VHDL 설계

▪ Entity

```
entity or2 is
        port(a,b:in    std_logic;
             y  :out   std_logic);
end or2;
```

▪ Architecture 1

```
architecture logic of or2 is
begin
        process(a,b)
        begin
                if a='1' or b='1' then
                        y<='1';
                else
                        y<='0';
                end if;
        end process;
end logic;
```

▪ Architecture 2

```
architecture logic2 of or2 is
begin
        y<=a or b;
end logic2;
```

## 6.3 Multiplexer

■ truth table

| sel | Y |
|---|---|
| 00 | a |
| 01 | b |
| 10 | c |
| 11 | d |

■ VHDL 설계

▪ Entity

```
entity mux4_1 is
        port(sel   :in  std_logic(1 downto 0);
             a,b,c,d:in  std_logic;
             y      :out std_logic);
end mux4_1;
```

▪ Architecture 1

```
architecture type1 of mux4_1 is
begin
        process(sel, a, b, c, d)
        begin
                if(sel="00") then
                        y<=a;
                elsif(sel="01") then
                        y<=b;
                elsif(sel="10") then
                        y<=c;
                else
                        y<=d;
                end if;
        end process;
end type1;
```

▪ Architecture 2

```
architecture type2 of mux4_1 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 "11" => y <=d;
                    when others => y <=a;
                end case;
            end process;
        end type2;
```

▪ Architecture 3

```
        architecture type3 of mux4_1 is
        begin
            y<=a when sel="00" else
                b when sel="01" else
                c when sel="10" else
                d;
        end type2;
```

## 6.4 encoder

■ truth table

| A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | Y |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 000 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 001 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 010 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 011 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 100 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 101 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 110 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 111 |

■ VHDL 설계

▪ Entity

```
entity encode is
        port(a   :in      std_logic_vector(7 downto 0);
             y   :out     std_logic_vector(2 downto 0));
end encode;
```

▪ Architecture 1

```
architecture type1 of encode is
begin
        process(a)
        begin
                if a="00000001" then y<="000";
                elsif a="00000010" then y<="001";
                elsif a="00000100" then y<="010";
                elsif a="00001000" then y<="011";
                elsif a="00010000" then y<="100";
                elsif a="00100000" then y<="101";
                elsif a="01000000" then y<="110";
                elsif a="10000000" then y<="111";
                else y<="XXX";
                end if;
        end process;
end type1;
```

▪ Architecture 2

```
architecture type2 of encode is
begin
        process(a)
        begin
```

```
            case a is
                when "00000001"=> y<="000";
                when "00000010"=> y<="001";
                when "00000100"=> y<="010";
                when "00001000"=> y<="011";
                when "00010000"=> y<="100";
                when "00100000"=> y<="101";
                when "01000000"=> y<="110";
                when "10000000"=> y<="111";
                when others => y <="XXX";
            end case;
        end process;
    end type2;
```

▪ Architecture 3

```
    architecture type3 of encode is
    begin
        y<="000" when a="00000001" else
        y<="001" when a="00000010" else
        y<="010" when a="00000100" else
        y<="011" when a="00001000" else
        y<="100" when a="00010000" else
        y<="101" when a="00100000" else
        y<="110" when a="01000000" else
        y<="111" when a="10000000" else
        "XXX";
    end type3;
```

## 6.5 decoder

■ truth table

| A | Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0 |
|---|---|---|---|---|---|---|---|---|
| 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 001 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 010 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 011 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 101 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 110 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 111 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

■ VHDL 설계

▪ Entity

```
entity decode is
        port(a   :in      std_logic_vector(2 downto 0);
             y   :out     std_logic_vector(7 downto 0));
end decode;
```

▪ Architecture 1

```
architecture type1 of decode is
begin
        process(a)
        begin
                if a="000" then y<="00000001";
                elsif a="001" then y<="00000010";
                elsif a="010" then y<="00000100";
                elsif a="011" then y<="00001000";
                elsif a="100" then y<="00010000";
                elsif a="101" then y<="00100000";
                elsif a="110" then y<="01000000";
                elsif a="111" then y<="10000000";
                else y<="100000000";
                end if;
        end process;
end type1;
```

▪ Architecture 2

```
architecture type2 of decode is
begin
        process(a)
        begin
```

```
            case a is
                when "000" => y <= "00000000";
                when "001" => y <= "00000010";
                when "010" => y <= "00000100";
                when "011" => y <= "00001000";
                when "100" => y <= "00010000";
                when "101" => y <= "00100000";
                when "110" => y <= "01000000";
                when "111" => y <= "10000000";
                when others y<="10000000";
            end case;
        end process;
    end type2;
```

▪ Architecture 3

```
    architecture type3 of decode is
    begin
        y<="00000001" when a="000" else
           "00000010" when a="001" else
           "00000100" when a="010" else
           "00001000" when a="011" else
           "00010000" when a="100" else
           "00100000" when a="101" else
           "01000000" when a="110" else
           "10000000"
    end type3;
```

# 7. 순차회로

## 7.1 D flip-flop

- Entity

```
entity dff is
        port(d, ck   :in   std_logic;
             q       :out  std_logic);
end dff;
```

- Architecture 1

```
architecture type1 of dff is
begin
        process(d, ck)
        begin
                wait until ck'event and ck='1';
                        q<=d;
        end process;
end type1;
```

- Architecture 2

```
architecture type2 of dff is
begin
        process(d, ck)
        begin
                if ck'event and ck='1' then
                        q<=d;
                end if;
        end process;
end type2;
```

## 7.2 Latch

- Entity

```
entity latch is
        port(d, en      :in     std_logic;
             q          :out    std_logic);
end latch;
```

- Architecture 1

```
architecture type2 of latch is
begin
        process(d, en)
        begin
                if en='1' then
                        q<=d;
                end if;
        end process;
end type2;
```

- Architecture 2

```
architecture type2 of latch is
        signal temp:std_logic;
begin
        process(d, en)
        begin
                temp<=(d and en) or (not(en) and temp);
        end process;

        process(temp)
        begin
                q<=temp;
        end process;
end type2;
```

## 7.3 Asynchronous Reset을 가진 16진 Binary Counter

■ Entity

```
entity counter is
        port(ck, rst    :in     std_logic;
                q       :buffer  std_logic_vector(3 downto 0));
end counter;
```

■ Architecture

```
architecture logic of counter is
begin
        process(ck, rst)
        begin
                if rst='0' then
                        q<="0000";
                else
                        if ck'event and ck='1' then
                                q<=q+1;
                        end if;
                end if;
        end process;
end logic;
```

※ 위의 예제에서처럼 만약 설계시 연산자들을 사용할 경우는 연산자들이 정의되어 있는 std_logic_arith 패키지와 std_logic_unsigned 패키지를 사용해야 한다. 즉 entity를 선언하기 전에 다음과 같이 패키지 사용을 선언해야 한다.

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

## 7.4 Synchronous Reset을 가진 16진 Binary Counter

- Entity

```
entity counter is
        port(ck, rst    :in     std_logic;
                q       :buffer  std_logic_vector(3 downto 0));
end counter;
```

- Architecture

```
architecture logic of counter is
begin
        process(ck. rst)
        begin
                if ck'event and ck='1' then
                        if rst='0' then
                                q<="0000";
                        else
                                q<=q+1;
                end if;
                end if;
        end process;
end logic;
```

# 8. Finite State Machine(FSM)

## 8.1 Background

- Finite-state machines(FSM)은 control 회로 설계에 주로 사용된다. 따라서 회로에서 중요한 부분을 차지하고 있으며 FSM의 정확한 동작은 매우 중요하다.
- 하드웨어 측면에서 FSM은 두 부분, 즉 현재 state를 기억하기 위해 flip-flop으로 구성된 순차회로와 다음 state 및 output신호 발생을 위한 조합회로로 구성되어 있다.
- VHDL을 이용해서 FSM을 설계할 경우 단순히 FSM의 동작을 기술하는 것이 아니라 실제 하드웨어를 고려하여 설계해야 정확한 동작을 수행하는 FSM을 설계할 수 있다.

## 8.2 FSM 설계를 위한 VHDL 코딩

- state 설정
  - VHDL을 이용하여 FSM 설계 시 FSM에서 사용되는 여러 state들을 쉽게 표현하기 위해서 type으로 state들을 선언하여 새로운 data type을 만든 후 state를 나타내는 signal의 date type을 미리 선언한 data type으로 정의한다.

  - 예) 만약 현재 설계하고자 하는 FSM에서 사용되는 state가 reset, ready, s1, s2 라 하면, 다음과 같이 type을 이용하여 사용되는 state들을 새로운 data type *states*로 선언한다.

    ```
    type states is (reset, ready, s1, s2);
    ```

    선언 후에는 현재 state와 next state를 나타내는 신호들의 data type을 states로 정의한다(예제에서는 현재 state를 c_state, 다음 state를 n_state라고 가정함).

    ```
    signal c_state, n_state:states;
    ```

- FSM 설계
  - 현재의 상태를 기억하기 위한 순차회로와 다음상태 및 출력을 정의하기 위한 조합회로 두 부분으로 나누어 설계하며 각각 1개의 process를 사용하여 설계한다.
  - 조합회로 부분
    - 조합회로 부분에서는 현재 상태와 입력을 고려하여 다음 상태와 출력을 정의한다. 따라서 조합회로 process의 sensitivity list에는 현재상태와 입력이 포함되어야 한다.
    - 다음 상태와 출력을 정의하기 위해서는 현재 어떤 상태에 있는

지 파악을 해야하며 case 문을 사용하여 현재 상태를 파악한다.

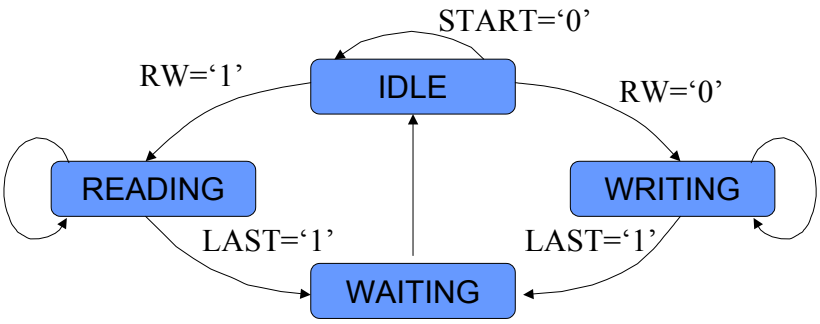
- 현재 상태가 파악되면 입력에 의해 다음상태와 출력이 정의되어야 하며 case문 내에 기술된 각 상태마다 if 문을 사용하여 기술한다.

▪ 순차회로 부분

- 순차회로 부분에서는 clock에 의해서 조합회로 부분에서 정의한 다음 상태를 현재상태로 기억(천이)한다.

■ 예) Read/Write Controller

▪ 아래의 그림은 read/write controller(RWCNTL)의 FSM state transition diagram을 나타낸다.




▪ 다음은 RWCNTL의 VHDL code를 나타낸다.

```
library ieee;
use ieee.std_logic_1164.all;
entity rwcntl is
        port(
                clock   :in     std_logic;
                start   :in     std_logic;
                rw      :in     std_logic;
                last    :in     std_logic;
                rdsig   :out    std_logic;
                wrsig   :out    std_logic;
                done    :out    std_logic);
end rwcntl;

architecture rtl of rwcntl is
        type state_type is (idle, reading, writing, waiting);
        signal current_state, next_state: state_type;
begin
```

```
        comb:process(current_state, start, rw, last)
        begin
                done<='0';
                rdsig<='0';
                wrsig<='0';
                case current_state is
                        when idle=>
                                if start='0' then
                                        next_state<=idle;
                                elsif rw='1' then
                                        next_state<=reading;
                                else
                                        next_state<=writing;
                                end if;
                        when reading=>
                                rdsig    <='1';
                                if last='0' then
                                        next_state<=reading;
                                else
                                        next_state<=waiting;
                                end if;
                        when writing=>
                                wrsig    <='1';
                                if last='0' then
                                        next_state<=writing;
                                else
                                        next_state<=waiting;
                                end if;
                        when waiting=>
                                done<='1';
                                next_state<=idle;
                end case;
        end process;

        seq:process
        begin
                wait until clock'event and clock='1';
                current_state<=next_state;
        end process;
end rtl;
```

- 조합회로 부분은 comb process에서 기술하고 있고, 순차회로 부분은 seq process에서 기술하고 있다.
- process의 첫 부분은 출력에 대한 초기값을 정의하고 있다. 만약 이곳에 초기값을 정의하지 않은 경우 case문 내에서 각 state에 따른 출력값을 모두 정의해야 하며 결국 코딩이 길어진다. 만약 각 state마다 모든 출력을 정의하지 않을 경우 합성시 불필요한 latch가 생긴다.
- 순차회로 부분은 seq process에서 기술하고 있다. clock의 상승천이에서 조합회로에서 결정한 다음 상태가 현재 상태가 된다.

## 8.3 FSM 초기화

■ VHDL 시뮬레이터는 기본적으로 object의 초기값을 지정된 data type의 left most값으로 설정한다. (위의 예제에서 current_state와 next_state는 초기값으로 idle state를 갖는다.) 그러나 실제 하드웨어에서는 어떤 state로 초기화될 지 알 수 없다. 따라서 state에 대한 초기화과정이 필요하다.

■ 초기화(reset) 방법

▪ Asynchronous reset

: Asynchronous reset을 수행하기 위해서 순차회로 부분을 수정한다. clock의 천이를 검사하기 전에 먼저 reset신호를 검사해서 reset신호가 active되었을 경우 현재 state에 원하는 초기 state로 설정한다.

▪ 예) rwcntl 예제에서 asynchronous reset을 추가하려면 seq process를 다음과 같이 수정한다.

```
seq : process (clock, reset)
begin
        if(reset='0') then
                current_state<=idle;
        elsif (clock'event and clock='1') then
                current_state<=next_state;
        end if;
end process;
```

▪ Synchronous reset

: Synchronous를 수행하기 위해 조합회로 부분을 수정한다. Reset 신호를 sensitivity list에 추가하고 reset이 active되었을 때 next_state에 초기화 하고자 하는 state를 설정한다. clock이 천이하면 seq process에서 초기화한 state가 current_state로 설정

된다.

▪ 예) rwcntl 예제에서 synchronous reset을 추가하려면 comp process를 다음과 같이 수정한다.

```
comb:process(reset, current_state, start, rw, last)
begin
        done<='0';
        rdsig<='0';
        wrsig<='0';
        if reset='0' then
                next_state<=idle;
        else
                case current_state is
                        when idel=>
                                if start='0' then
                                        next_state<=idle;
                                elsif rw='1' then
                                        next_state<=reading;
                                else
                                        next_state<=writing;
                                end if;
                        when reading=>
                                rdsig    <='1';
                                if last='0' then
                                        next_state<=reading;
                                else
                                        next_state<=waiting;
                                end if;
                        when writing=>
                                wrsig    <='1';
                                if last='0' then
                                        next_state<=writing;
                                else
                                        next_state<=waiting;
                                end if;
                        when waiting=>
                                done<='1';
                                next_state<=idle;
                end case;
        end if;
        end process;
```

# 9. 기타

## 9.1 자주 발생되는 Error들

- multiple assignment
  - 한 신호를 여러 process에서 assign하면 합성할 때 multiple assignment error가 발생한다. Architecture 내에서 process들은 서로 concurrent 하게 동작하므로 여러 process내에서 특정 신호를 assign하면 1개의 신호에 여러 신호들이 동시에 assign될 수 있다. 따라서 합성시 error가 발생한다. 그러므로 1개의 신호를 동시에 assign 시키지 않도록 coding하여야 한다.
- multiple wait
  - 경우에 따라서 process 내에서 wait문을 여러개 사용하면 회로의 동작을 보다 쉽게 기술할 수 있다. 이때 시뮬레이션에서는 문제가 없지만 실제 하드웨어로는 구현하기 어려우며 따라서 합성할 경우 error를 유발할 수 있다.
- signal'event
  - 일반적으로 if 문에서 else 혹은 elsif 문이 사용될 경우 합성하면 mux와 latch로 구현되고 signal'evnet 문은 flip-flop으로 구현된다. 하지만 signal'evnet 문 다음에 else 문이 올 경우 실제 하드웨어 구현이 어려우며 합성과정에서 error를 유발한다. 따라서 signal'event문 다음에는 else 혹은 다른 if를 기술하지 말고 end if로 끝마쳐야 한다.

  - 예)

```
if a='1' and a'evnet then
        y<='1';
else
        y<='0';
        z<='0';
end if;
```

※ 위의 예제는 a'evnet 다음에 else가 기술되었으므로 합성할 경우 error를 유발한다.

## 9.2 Reserved Worlds

- VHDL의 예약어는 다음과 같다(VHDL 87).

| | | | | |
|---|---|---|---|---|
| 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 |
| open | or | others | out | package |
| port | procedure | process | range | record |
| severity | rem | report | return | select |
| transport | type | units | until | use |
| variable | wait | when | while | with |
| xor | | | | |

VHDL 93에서 추가된 내용

| | | | | |
|---|---|---|---|---|
| group | impure | inertial | literal | postponed |
| pure | reject | rol | ror | shared |
| 님 | sll | sra | srl | unaffected |
| xnor | | | | |

## 9.3 References


[1] K. C. Chang, Digital Design and Modeling with VHDL and Synthesis, IEEE press.

[2] Douglas L. Perry, VHDL, McGraw-Hill, 1995.

[3] Allen Dewey, Analysis and Design of Digital Systems with VHDL, ITP, 1997.

[4] Zainalabedin Navabi, VHDL, McGraw-Hill, 1998.

[5] IEEE Standard VHDL Language Reference Manual, IEEE press, 1994.

[6] 이대영 외 3인, VHDL 기초와 응용, 홍릉과학출판사, 1995.