



VHDL 프로그래밍

3. VHDL 문법 기초

한 동 일



학습 목표

- VHDL언어를 구성하는 문자 세트를 배운다.
- VHDL언어를 구성하는 문장 구성 요소를 배운다.
- VHDL언어의 예약어에 대해서 숙지한다.
- VHDL언어의 식별어를 파악할 줄 알고 사용할 줄 안다.
- 리터럴(literal)의 종류를 알고 구분할 수 있다.
- 객체 클래스의 종류를 알고 구분할 수 있다.
- 형(type)의 종류와 선언 방식을 알 수 있다.
- 연산자의 종류와 우선 순위를 이해한다.
- 형과 연산자의 관계를 이해한다.



목 차

- VHDL 문법 기초
 - VHDL 문장 구성 요소
 - 예약어(reserved word)
 - 식별어(identifier)
 - 리터럴(literal)
 - 주석(comment)
 - 연산자(operator)
 - 형(type)

3/73



VHDL의 문장 구성 요소

- 사용 가능한 문자 세트
 - 도형 문자(graphic code)
 - ISO/IEC 8859-1: 1987, Information Processing – 8-bit Single-Byte Coded Graphic Character Sets -- Part 1: Latin Alphabet No. 1.
 - 포맷 제어 문자(format effector)
 - 수평 탭(horizontal tab)
 - 수직 탭(vertical tab)
 - 복귀 부호(carriage return)
 - 줄먹임 문자(line feed)
 - 용지 먹임 문자(form feed)
 - 주석 부분
 - -- 기호 다음 부분
 - /* */ 기호 내부(2008 버전 이후 사용 가능)
 - 컴파일러가 무시하므로 한글도 입력 가능

4/73.

[illegible]

b_7 →					0	0	0	0	1	1	1	1	
b_6 →					0	0	1	1	0	0	1	1	
b_5 →					0	0	1	1	0	1	0	1	
Bits	b_4	b_3	b_2	b_1	<div>Column → Row ↓</div>	0	1	2	3	4	5	6	7
	↓	↓	↓	↓									



VHDL의 문장 구성 요소

■ 주 문자 세트

- 주 문자 세트만을 이용하여 VHDL 서술 가능

a) Uppercase letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ

b) Digits

0 1 2 3 4 5 6 7 8 9

c) Special characters

" # & ' () * + , - . / : ; < = > [] _ |

d) The space characters

7/73.



VHDL의 문장 구성 요소

■ 보조 문자 세트

- 보조 문자 세트를 활용하여 VHDL 서술도 가능

e) Lowercase letters

a b c d e f g h i j k l m n o p q r s t u v w x y z ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

f) Other special characters

! \$ % & @ ? \ ^ ` { } ~ ¡ ¢ £ € ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ × ÷ - (soft hyphen)

8/73.



VHDL의 문장 구성 요소

- 문장 구성 요소의 분류
 - 분리어(separator)
 - 구분어(delimiter)
 - 예약어(reserved word)
 - 식별어(identifier)
 - 리터럴(literal)
 - 주석(comment)

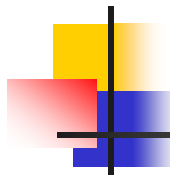
9/73.



문장 구성 요소의 분류

- 분리어(separator)
 - 문장 요소들을 서로 분리시키는 역할
 - 공백 문자(space), 포맷 제어 문자(format effector), 라인 종단 문자(End of line)로 구성됨
 - 분리어 사용 방법
 - 라인의 끝은 항상 분리어로 끝남
 - 분리어의 종류는 무관
 - 하나 이상의 분리어가 연속되어 사용되어도 무관
 - 가독성을 향상 시킬 수 있도록 적절히 사용하면 됨

10/73.



문장 구성 요소의 분류

구분어(delimiter)

- 특수 문자를 이용하여 문자열들을 서로다른 문장 요소로 분리하고 구분하는 역할
- 구분어의 분류
 - 단일 구분어(single delimiter)
 - 복합 구분어(compound delimiter)
- 단일 구분어의 종류 및 이름

종류	이름	종류	이름	종류	이름
&	Ampersand	-	Hypen, minus sign	>	Greater-than sign
'	Apostrophe, tick	.	Dot, point, period	`	Grave accent
(Left parenthesis	/	Slash, divide, solidus		Vertical line(bar)
)	Right parenthesis	:	Colon	[Left square bracket
*	Asterisk, multiply	;	Semicolon]	Right square bracket
+	Plus sign	<	Less-than sign	?	Question mark
,	Comma	=	Equals sign	@	Commencial at

11/73.



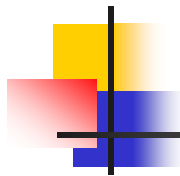
구분어

구분어(delimiter)

- 복합 구분어의 종류 및 이름

종류	이름	종류	이름
=>	Arrow	**	Double star, exponentiation
:=	Variable assignment	/=	Inequality, not equal
>=	Greater than or equal	<=	Less than or equal, signal assignment
<>	Box	??	Condition conversion
?=	Matching equality	?/=	Matching inequality
?<	Matching less than	?<=	Matching less than or equal
?>	Matching greater than	?>=	Matching greater than or equal
<<	Double less than	>>	Double greater than

12/73



문장 구성 요소의 분류

- 예약어(reserved word)
 - 총 115개(2008 버전의 경우)
 - 대소문자 구분이 없음
 - 다른 의미의 식별어로 사용하면 안됨
 - 코딩 시 속지하고 있어야 함

13/73



예약어: 1076-2008 버전

abs	access	after	alias	all	and	architecture
array	assert	assume	assume_guarantee		attribute	begin
block	body	buffer	bus	case	component	configuration
constant	context	cover	default	disconnect	downto	else
elsif	end	entity	exit	fairness	file	for
force	function	generate	generic	group	guarded	if
impure	in	inertial	inout	is	label	library
linkage	literal	loop	map	mod	nand	new
next	nor	not	null	of	on	open
or	others	out	package	parameter	port	postponed
procedure	process	property	protected	pure	range	record
register	reject	release	rem	report	restrict_guarantee	
restrict	return	rol	ror	select	sequence	severity
signal	shared	sla	sll	sra	srl	strong
subtype	then	to	transport	type	unaffected	units
until	use	variable	vmode	vprop	vunit	wait
when	while	with	xnor	xor		

14/73



문장 구성 요소의 분류

■ 식별어(identifier)

- 설계자가 명명하는 문법 구조의 이름들
 - 엔티티 이름
 - 아키텍처 이름
 - 신호, 변수 이름 등등
 - 넓은 의미로는 예약어도 식별어의 일종(식별어의 명명 규칙에 의해서 생성됨)
- 식별어의 분류
 - 기본 식별어(basic identifier)
 - 확장 식별어(extended identifier)

15/73.



식별어

■ 식별어의 BNF 명명 규칙

```
identifier ::= basic_identifier | extended_identifier
```

```
basic_identifier ::=
```

```
    letter { [ underline ] letter_or_digit }
```

```
letter_or_digit ::= letter | digit
```

```
letter ::= upper_case_letter | lower_case_letter
```

```
extended_identifier ::= \ graphic_character { graphic_character } \
```

16/73

식별어

■ 기본 식별어의 예

■ 정상적인 예

- COUNT, A, DCT, FrameCount
- X9, ADD1, Pentium4, P4C1
- RST_n, Frame_Count, ADD_A_B

■ 잘못 정의된 예

- 9X : 숫자가 제일 앞에 올 수 없음
- _carry_out : 밑줄이 제일 앞에 올 수 없음
- ADD_A_B : 밑줄이 연이어 두개 올 수 없음
- DCT_ : 밑줄이 마지막에 올 수 없음

■ 확장 식별어의 예

■ 정상적인 예

- \ELSE\, \200\$\, \ADD__A_B\, \DCT_\, \74LS32\

17/73.

문장 구성 요소의 분류

■ 리터럴(literal)

■ 상수 선언 시 사용되는 상수 값 자체

```
constant HTOTAL_WIDTH : integer := 512 ;  
constant MAX_BIN : string := B"1111_1111_1111";  
constant MAX_HEX : BIT_VECTOR := X"FFF";  
type VOLTAGE_RANGE is range -15.0 to 15.0;
```

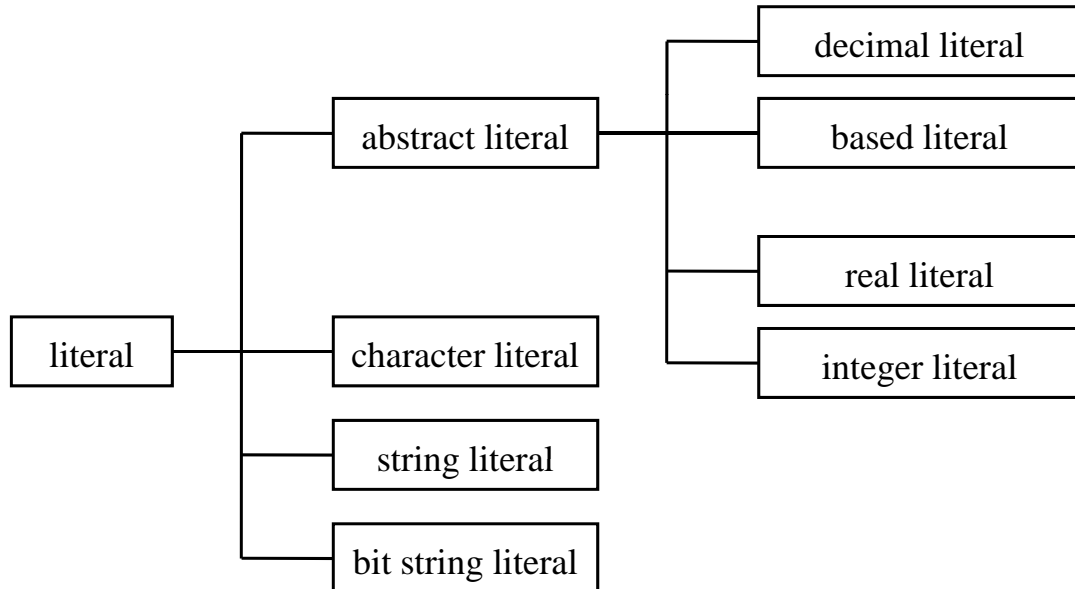
■ 리터럴

- 512
- B"1111_1111_1111"
- X"FFF"
- -15.0
- 15.0

18/73.

문장 구성 요소의 분류

- 리터럴(literal)
 - 리터럴의 분류

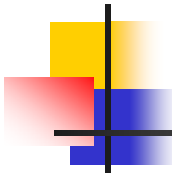


19/73

리터럴

- 추상 리터럴(abstract literal)
 - 수형에 의한 분류
 - 실수형 리터럴(real literal)
 - 정수형 리터럴(integer literal)
 - 진법에 의한 분류
 - 십진수 리터럴(decimal literal)
 - 진수 리터럴(based literal)
- 추상 리터럴의 표현
 - 숫자 앞 부분에 0을 추가 가능
 - 자리수를 일치시키거나 가독성을 위함
 - 밑줄 사용 가능
 - 두 개이상 연이어 사용 불가
 - 처음과 마지막에 사용 불가

20/73.



리터럴

■ 십진수 리터럴의 BNF 정의

```
decimal_literal ::= integer [ . integer ] [ exponent ]
integer ::= digit { [ underline ] digit }
exponent ::= E [ + ] integer | E - integer
```

■ 십진수 리터럴의 예

■ 정수형 리터럴의 예

- 15, 0, 1E6, 123_456_789

■ 실수형 리터럴의 예

- 15.0, 0.0, 1.2e+6, 3.141_592, 1.234E-15

■ 십진수 리터럴의 사용 예

```
constant HALF_PERIOD : time := 5.0E3 ns ;
```

21/73.



리터럴

■ 진수 리터럴의 BNF 정의

```
based_literal ::= base # based_integer [ . based_integer ] # [ exponent ]
base ::= integer
based_integer ::= extended_digit { [ underline ] extended_digit }
extended_digit ::= digit | letter
exponent ::= E [ + ] integer | E - integer
```

■ 2진수 ~ 16진수 사이의 진수 표현 가능

■ 진수 리터럴의 예

■ 2진수 리터럴의 예

- 2#1111_1110#, 2#1.1111_1111_111#E11

■ 16진수 리터럴의 예

- 16#FE#, 16#F.FF#E+2

22/73.



- 문자 리터럴(character literal)의 BNF 정의

```
character_literal ::= 'graphic_character'
```

- 두개의 작은 따옴표 사이에 하나의 문자만 올 수 있음
- 올 수 있는 문자는 191개의 도형 문자들

■ 문자 리터럴의 예

- 정상적인 예
 - ‘a’, ‘A’, ‘@’, ‘’, ‘ ‘, ‘1’
- 잘못 정의된 예
 - “a”, ‘ab’

23/73.



- 문자열 리터럴(string literal)의 BNF 정의

```
string_literal ::= "{ graphic_character }"
```

- 두개의 따옴표 사이에 임의의 개수의 도형문자들이 올
- 내부에 따옴표를 사용하는 경우 연속된 두 개의 따옴표를 하나의 따옴표로 계산
- 반드시 한 줄로 표기해야 함
- 두 줄 이상이 필요할 경우 & 로 두 개의 문자열을 연결

■ 문자열 리터럴의 예

- “a”, “ “, “””” -- 모두 길이가 1인 문자열
- “imagefolder/sampleimage.txt”

24/73.

리터럴

■ 비트열 리터럴(bit string literal)의 BNF 정의

```
bit_string_literal ::= [ integer ] base_specifier "[bit_value]"
```

```
bit_value ::= graphic_character { [ underline ] graphic_character }
```

```
base_specifier ::= B | O | X | UB | UO | UX | SB | SO | SX | D
```

- 2, 8, 16진수 표현의 비트값을 구성
- 두개의 따옴표 사이에 오는 문자열로 비트열 리터럴 생성
- 올 수 있는 문자열
 - 0~9, A~F, 'U', 'X', 'O', '1', 'Z', 'W', 'L', 'H', '-' (대소문자 구분 없음)
- Integer : 비트열의 길이 정보를 제공 가능
- base specifier
 - B : 이진수, O : 8진수, X : 16진수
 - U : std_ulogic, S : std_logic, D : 십진수 표기를 나타냄

25/73.

리터럴

■ 비트열 리터럴의 예

bit string literal	length	equivalent value
B""	0	empty bit string
B"1111_1111"	8	"11111111"
X"FF"	8	B"1111_1111"
O"77"	6	B"111_111"
X"77"	8	B"0111_0111"
B"XXXX_10HL"	8	"XXXX10HL"
X"F-"	8	B"1111_----"
12X"F-"	12	B"0000_1111_----"
12SX"XXA0"	12	B"XXXX_1010_0000"

- 밑줄 문자의 개수는 비트열의 길이와 무관
 - 가독성을 높이기 위한 용도

26/73

문장 구성 요소의 분류

■ 주석(comment)

- VHDL 코드의 가독성을 증가시키는 목적
 - VHDL 서술과 관련된 부가 정보를 문법 제한 없이 서술 가능
- 주석의 분류
 - 한줄 주석(single-line comment)
 - 범위 주석(delimited comment)

■ 한줄 주석

- 두 개의 연속된 하이픈 “--” 이용
 - 해당 라인 내에서 -- 기호 이후의 대부분의 기호는 주석으로 무시함
 - 주석 내부의 -- 기호도 무시
- 한줄 주석 내에 다음 문자는 사용 불가
 - 수직 탭(vertical tab), 복귀 부호(carriage return)
 - 줄먹임 문자(line feed), 용지 먹임 문자(form feed)

27/73.

주석

■ 범위 주석

- /* 와 */ 기호 사용
 - /*와 */ 내부의 모든 기호는 주석으로 간주하고 무시함
 - 범위 주석 내부의 /* 기호는 무시됨
 - 범위 주석 내부의 -- 기호도 마찬가지로 무시함
 - 한줄 주석과 마찬가지로 내부에 한글 기호 사용 가능
 - 수직 탭, 복귀 부호등도 사용가능하며 VHDL 해석기가 무시함

■ 주석의 사용 예

```
---- The first two hyphens start the comment.  
end; -- End of process  
r_image := 255; /* Comments /* do not nest */  
-- 한글 주석도 가능합니다.
```

28/73.

문장 구성 요소의 분류

■ VHDL 문장 예

```
S <= A xor B; -- calculate sum of two input
```

■ 문장 구성 요소 분류 예


- 식별어, 구분어, 주석, 예약어 사이의 공백문자는 분리어
 - S : 식별어
 - <= : 구분어
 - A : 식별어
 - xor : 예약어
 - B : 식별어
 - ; : 구분어
 - -- calcu... : 주석

29/73

연산자(operator)

■ 연산자(operator)란?

- 표현식(expression)에 사용되어 식의 값을 결정하는 용도
- 연산자의 분류
 - 8개의 클래스로 분류, 우선 순위가 높은 연산자가 먼저 계산됨
 - 같은 우선 순위에 대해서는 왼쪽부터 오른쪽 순서로 계산
 - 가급적 괄호를 이용하여 우선 순위를 지정할 것

operator class	operators	precedence
condition	??	low
logical	and or nand nor xor xnor	
relational	= /= < <= > >= ?= ?/= ?< ?<= ?> ?>=	
shift	sll srl sla sra rol ror	
adding	+ - &	
sign	+ -	
multiplying	* / mod rem	
miscellaneous	** abs not	high

30/73.

연산자(operator)

- 조건 연산자(condition operator)
 - 단항 연산자(unary operator)
 - IEEE Standard 1076-2008 버전부터 추가된 연산자
- 조건 연산자의 정의

function "??" (anonymous: BIT) **return** BOOLEAN;

- bit 형을 boolean 형으로 변환하는 용도
 - '1'인 경우 TRUE를 리턴
 - '0'인 경우 FALSE를 리턴

31/73

연산자(operator)

- 논리 연산자(logical operator)
 - bit, boolean 형에 대해서 연산이 정의(standard 패키지)
 - nand, nor 연산자가 연이어 올 때는 괄호가 필요
- 논리 연산자의 정의

<u>A</u>	<u>B</u>	<u>A and B</u>	<u>A</u>	<u>B</u>	<u>A or B</u>	<u>A</u>	<u>B</u>	<u>A xor B</u>
T	T	T	T	T	T	T	T	F
T	F	F	T	F	T	T	F	T
F	T	F	F	T	T	F	T	T
F	F	F	F	F	F	F	F	F
<u>A</u>	<u>B</u>	<u>A nand B</u>	<u>A</u>	<u>B</u>	<u>A nor B</u>	<u>A</u>	<u>B</u>	<u>A xnor B</u>
T	T	F	T	T	F	T	T	T
T	F	T	T	F	F	T	F	F
F	T	T	F	T	F	F	T	F
F	F	T	F	F	T	F	F	T

32/73

논리 연산자(logical operator)

■ 논리 연산자의 사용 예

```
signal a, b, c, b1, b2, b3, b4, b5, b6, b7, b8 : boolean;
```

```
-- 중략
```

```
b5 <= a nand b nand c; -- 에러를 유발하는 경우
```

```
b6 <= a nor b nor c; -- 에러를 유발하는 경우
```

```
b7 <= a and b or c; -- 에러를 유발하는 경우
```

```
-- 정상적인 경우들
```

```
b1 <= a and b and c and d ;
```

```
b2 <= a or b or c;
```

```
b3 <= a xor b xor c xor d;
```

```
b4 <= a xnor b xnor c;
```

```
b5 <= (a nand b) nand c;
```

```
b7 <= a and (b or c) ;
```

33/73

연산자(operator)

■ 관계 연산자(relational operator)

- 두 피연산자의 동일성 여부 검출
- 두 피연산자의 순서를 테스트

■ 관계 연산자의 분류

- 관계 연산자
 - 두 피연산자의 관계를 비교한 후 boolean 값을 리턴
- 정합 관계 연산자(matching relational operator)
 - 두 피연산자의 관계를 비교한 후 피연산자와 같은 형을 리턴
 - IEEE Standard 1076-2008 버전부터 추가

relational operator	operators
relational	= /= < <= > >=
matching relational	?= ?/= ?< ?<= ?> ?>=

34/73.

관계 연산자

■ 관계 연산자의 기능 정의

Operator	Operation	Operand type	Result type
=	Equality	Any type, other than a file type or a protected type	BOOLEAN
/=	Inequality	Any type, other than a file type or a protected type	BOOLEAN
< <= > >=	Ordering	Any scalar type or discrete array type	BOOLEAN

? =	Matching equality	BIT or STD_ULOGIC	Same type
		Any one-dimensional array type whose element type is BIT or STD_ULOGIC	The element type
? /=	Matching inequality	BIT or STD_ULOGIC	Same type
		Any one-dimensional array type whose element type is BIT or STD_ULOGIC	The element type
? < ? <= ? > ? >=	Matching ordering	BIT or STD_ULOGIC	Same type

35/73

연산자(operator)

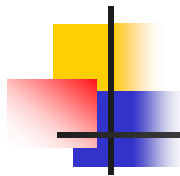
■ 자리 이동 연산자(shift operator)

- bit, boolean 형의 일차원 배열에 대해서 연산이 정의
- 피연산자의 정수 부분이 음수일 경우에는 방향이 바뀜

■ 자리 이동 연산자의 기능 정의

Operator	Operation	Left operand type	Right operand type	Result type
sll	Shift left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
srl	Shift right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sla	Shift left arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sra	Shift right arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
rol	Rotate left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
ror	Rotate right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left

36/73



자리 이동 연산자

- 자리 이동 연산자의 동작
 - **sll** : shift left logical
 - 정수값만큼 배열을 왼쪽으로 이동
 - 제일 오른쪽에는 배열 요소의 'LEFT 속성값이 입력
 - **srl** : shift right logical
 - 정수값만큼 배열을 오른쪽으로 이동
 - 제일 왼쪽에는 배열 요소의 'LEFT 속성값이 입력
 - **sla** : shift left arithmetic
 - 정수값만큼 배열을 왼쪽으로 이동
 - 제일 오른쪽에 있던 배열 값이 반복해서 입력
 - **sra** : shift right arithmetic
 - 정수값만큼 배열을 오른쪽으로 이동
 - 제일 왼쪽에 있던 배열 값이 반복해서 입력

37/73.



자리 이동 연산자

- 자리 이동 연산자의 동작
 - **rol** : rotate left logical
 - 정수값만큼 배열을 왼쪽으로 이동
 - 제일 왼쪽에 있던 값이 오른쪽으로 재입력
 - **ror** : rotate right logical
 - 정수값만큼 배열을 오른쪽으로 이동
 - 제일 오른쪽에 있던 값이 왼쪽으로 재입력

38/73



자리 이동 연산자

■ 자리 이동 연산 예

operation	result	remark
"01100011" sll 1	"11000110"	배열의 오른쪽에 BIT'LEFT 값인 '0' 이 입력됨
"01100011" sll 2	"10001100"	위의 동작을 2번 반복
"01100011" srl 1	"00110001"	배열의 왼쪽에 BIT'LEFT 값인 '0' 이 입력됨
"01100011" srl 3	"00001100"	위의 동작을 3번 반복
"01100011" srl -2	"10001100"	"01100011" sll 2 연산과 동일한 값을 출력
"01100011" sla 1	"11000111"	배열의 오른쪽 끝값 '1'이 입력됨
"01100011" sra 2	"00011000"	배열의 왼쪽 끝 값 '0'이 반복해서 입력됨
"01100011" rol 1	"11000110"	배열의 왼쪽 끝 값 '0'이 오른쪽 끝에 입력됨
"01100011" ror 2	"11011000"	배열의 오른쪽 끝 값 '1'이 연속해서 왼쪽 끝에 입력됨
"01100011" ror 0	"01100011"	변화 없음

39/73



연산자(operator)

■ 덧셈 연산자(adding operator)

- +, - : 수학적인 산술 연산 수행
- & : 배열형을 서로 접합하여 큰 배열형을 생성
- 이항 연산자

■ 덧셈 연산자의 기능 정의

Operator	Operation	Left operand type	Right operand type	Result type
+	Addition	Any numeric type	Same type	Same type
-	Subtraction	Any numeric type	Same type	Same type
&	Concatenation	Any one-dimensional array type	Same array type	Same array type
		Any one-dimensional array type	The element type	Same array type
		The element type	Any one-dimensional array type	Same array type
		The element type	The element type	Any one-dimensional array type

40/73.



덧셈 연산자(adding operator)

■ 집합 연산 예

operation	result
"0110" & "1001"	"01101001"
'0' & "1001"	"01001"
"0110" & '1'	"01101"
'0' & '1'	"01"
'0' & "1001" & '1'	"010011"

41/73



연산자(operator)

■ 부호 연산자(sign operator)

- +, - : 수학적인 산술 연산과 같은 동작
- 수치형에 대해서 정의됨
- 단항 연산자

■ 부호 연산자의 기능 정의

Operator	Operation	Operand type	Result type
+	Identity	Any numeric type	Same type
-	Negation	Any numeric type	Same type

42/73.

부호 연산자(sign operator)

■ 부호 연산자 사용 규칙

- 부호 연산보다 우선 순위가 높은 다음의 연산자들 다음에는 바로 부호 연산자가 올 수 없음

- 곱셈 연산자 : *, /
- 지수 연산자 : **
- 절대값 연산자 : abs
- not 연산자

■ 부호 연산자 사용 예

A/+B -- An illegal expression.
A**.-B -- An illegal expression.

A/(+B) -- A legal expression.
A ** (-B) -- A legal expression.

43/73.

연산자(operator)

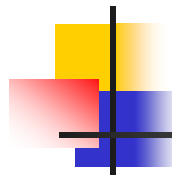
■ 곱셈 연산자(multiplying operator)

- *, / : 수학적인 정의와 같은 동작
- mod : 정수형, 물리형에 대해서 정의됨
- rem : 정수형, 물리형에 대해서 정의됨

■ 곱셈 연산자의 기능 정의

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any integer type	Same type	Same type
		Any floating-point type	Same type	Same type
/	Division	Any integer type	Same type	Same type
		Any floating-point type	Same type	Same type
mod	Modulus	Any integer type	Same type	Same type
rem	Remainder	Any integer type	Same type	Same type

44/73



곱셈 연산자

■ 나머지 연산자 **rem**의 정의

- 정수 나눗셈과 rem 연산자와의 관계식

$$A = (A/B)*B + (A \text{ rem } B)$$

- (A rem B)의 값은 피연산자 A의 부호
- (A rem B)의 절대값은 B값의 절대값보다 작은 값

■ 계수 연산자 **mod**의 정의

- mod 연산자는 임의의 정수 N에 대해 아래의 식이 성립

$$A = B*N + (A \text{ mod } B)$$

- (A mod B)의 값은 피연산자 B의 부호
- (A mod B)의 절대값은 B값의 절대값보다 작은 값

45/73.



곱셈 연산자

■ 곱셈 연산자의 연산 예

A의 부호

B의 부호

A	B	A/B	A rem B	A mod B
5	3	1	2	2
(-5)	3	-1	-2	1
(-5)	(-3)	1	-2	-2
5	(-3)	-1	2	-1
5 ns	3 ns	1	2 ns	2 ns
(-5 ns)	3 ns	-1	-2 ns	1 ns
1 ns	300 ps	3	100 ps	100 ps
11	4	2	3	3
11	(-4)	-2	3	-1
(-11)	(-4)	2	-3	-3

46/73

연산자(operator)

■ 기타 연산자(miscellaneous operator)

- ** : 지수 연산자(exponentiating operator)
 - 수학적 정의와 같은 동작
 - 정수형, 실수형에 대해서만 정의, 지수부(exponent)에는 정수형만 옴
 - 지수부가 음수일 때는 지수부가 양수일 때의 값의 역수
- abs : 절대값 연산자
 - 단항 연산자, 수치형에 대해서 정의
- not :
 - 단항 연산자, bit, boolean형에 대해서 정의

<u>A</u>	<u>not A</u>
T	F
F	T

47/73.

형(type)

■ VHDL에서 객체(object)란?

- 특정 형(type)을 가지면서 이 형에 의해서 정의될 수 있는 범위내의 값(value)를 가지는 실체
- 객체 클래스의 분류
 - 상수(constant) : 초기값을 배정하면서 이 값이 바뀌지 않음
 - 신호(signal) : 아키텍처 선언부에서 선언되며 신호 배정문을 이용
 - 변수(variable) : 프로세스 선언부에서 선언되며 변수 배정문을 이용
 - 파일(file) : 파일 형태의 데이터를 처리하는 용도로 사용
- 객체들은 4가지 클래스 중의 하나에 속함

48/73.

형(type)

■ 객체(object)의 정의

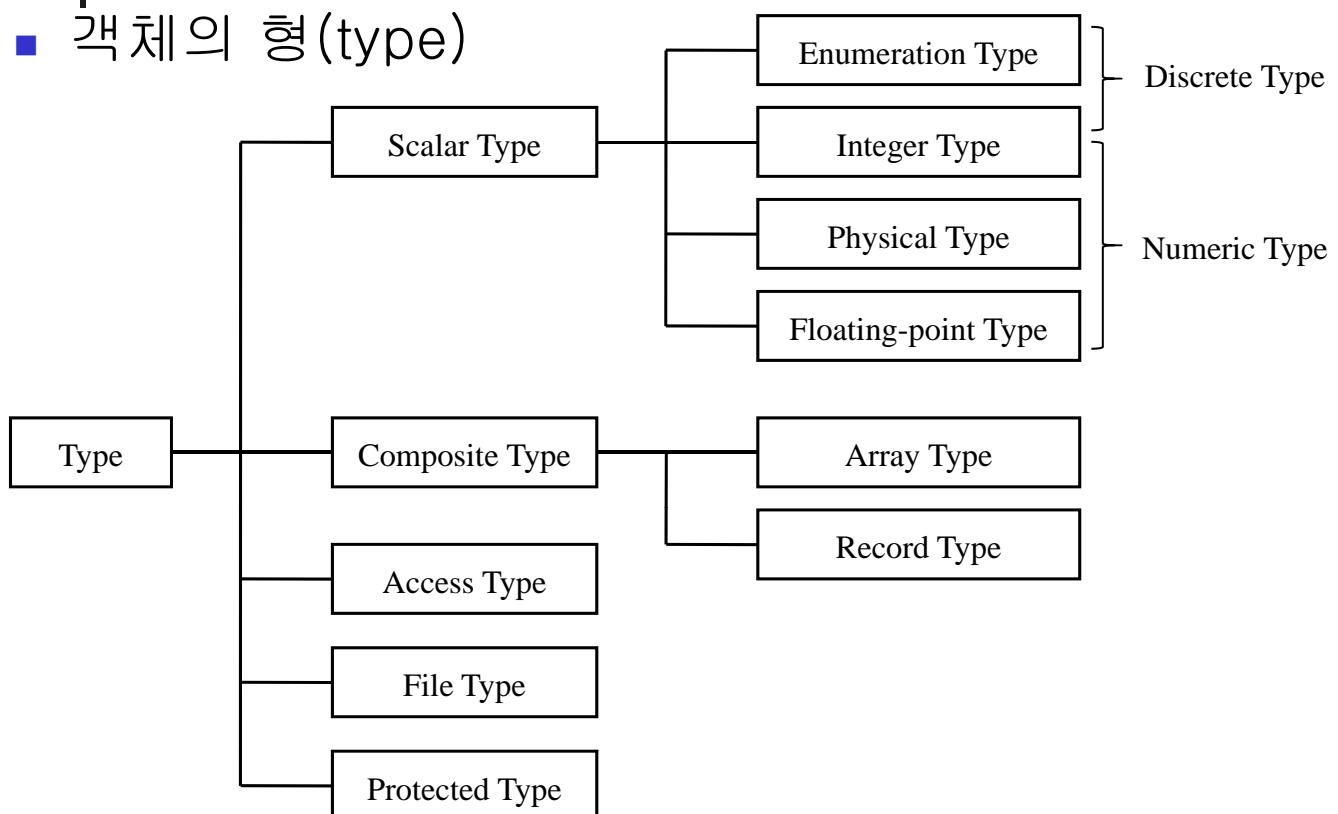
- 객체의 사용을 위해서는 먼저 객체를 정의해야 함
- 객체의 정의 시 객체 이름, 객체 클래스, 객체의 형을 동시에 정의해야 함
- 상수를 정의할 경우에는 대부분 초기값도 같이 정의

object class	Object name	type	value
constant	BUS_WIDTH	: integer := 8;	8
constant	FALL_TIME	: TIME := 20 ns;	20 ns
signal	HACTIVE	: bit ;	'0' or '1'
signal	SUM	: integer range 0 to 255;	0 ~ 255
variable	L	: line ;	Pointer 값
variable	data_bus	: bit_vector (3 downto 0);	"0000" ~ "1111"
file	IN_TXT	: text open read_mode is "filein.txt";	filein.txt
file	OUT_IMG	: char_file open write_mode is "f.bmp";	f.bmp

49/73.

형(type)

■ 객체의 형(type)



50/73

형(type)

■ 형 선언의 BNF 정의

```
type_declaration ::=  
    type identifier is type_definition ;  
type_definition ::=  
    scalar_type_definition  
    | composite_type_definition  
    | access_type_definition  
    | file_type_definition  
    | protected_type_definition
```

■ 형 선언의 예

```
type SYNC_WIDHT is range 0 to 1023;
```

51/73

형(type)

■ 스칼라형(scalar type)

■ 스칼라형의 종류

- 열거형(enumeration type)
- 정수형(integer type)
- 물리형(physical type)
- 부동소수점형(floating-point type)

■ 모든 스칼라형들은 값의 순서를 정할 수 있음

■ 관계 연산자(relational operator)를 이용한 대소 비교가 가능

52/73.

스칼라형(scalar type)

■ 열거형의 BNF 정의

```
enumeration_type_definition ::=  
    ( enumeration_literal { , enumeration_literal } )  
enumeration_literal ::= identifier | character_literal
```

- 열거형 선언에 사용된 식별어와 문자 리터럴을 열거 리터럴(enumeration literal) 이라고 함

■ 열거형 선언의 예

```
type BIT is ('0', '1');  
type MULTI_LEVEL_LOGIC is (LOW, WEAK, HIGH);
```

53/73

스칼라형(scalar type)

■ 정수형의 BNF 정의

```
integer_type_definition ::= range_constraint  
range_constraint ::= range range_info  
range_info ::=  
    range_attribute_name | simple_expression direction simple_expression  
direction ::= to | downto
```

- simple_expression 값은 컴파일 타임에 상수값이어야 함
- direction에 맞게 좌우 값의 크기가 지정되어야 함

■ 정수형 선언의 예

```
type BIT_INDEX is range 7 downto 0; -- 권장하지 않음  
type CONVERTED_CB is range -512 to 511; -- 사용 권장
```

54/73

스칼라형(scalar type)

■ 물리형의 BNF 정의

```
physical type definition ::=  
    range_constraint  
    units  
        primary_unit_declaration  
        { secondary unit declaration }  
    end units [ physical_type_simple_name ]
```

```
primary_unit_declaration ::= identifier ;  
secondary unit declaration ::= identifier = physical literal ;  
physical_literal ::= [ abstract_literal ] unit_name
```

- 시간, 거리, 전압, 전류와 같은 물리량의 표현에 사용

55/73

스칼라형(scalar type)

■ 물리형 선언의 예

```
type SIMULATION_TIME is range 0 to 1E+8  
    units  
        fs;  
        ps= 1000 fs;  
        ns= 1000 ps;  
        us= 1000 ns;  
        ms= 1000 us;  
    end units SIMULATION_TIME ;
```

- 수 표현과 단위 사이에는 반드시 공백 문자가 와야 함

56/73

스칼라형(scalar type)

■ 부동소수점형의 BNF 정의

```
floating_type_definition ::= range_constraint  
range_constraint ::= range range_info  
range_info ::=  
    range_attribute_name | simple_expression direction simple_expression  
direction ::= to | downto
```

- simple_expression 값은 컴파일 타임에 상수값이어야 함
- direction에 맞게 좌우 값의 크기가 지정되어야 함

■ 부동소수점형 선언의 예

```
type LVTTTL_VOLTAGE is range 0.0 to 3.6;  
type FILTER_RANGE is range 255.0 downto -255.0;
```

57/73

형(type)

■ 복합형(composite type)

- 값의 모임을 정의하는데 사용
- 복합형의 종류
 - 배열형(array type) : 동일한 자료형의 모임
 - 레코드형(record type) : 서로 다른 자료형의 모임을 서술 가능
- 여러 개의 내부 원소들로 이루어지는 객체의 모임을 나타냄
- 복합형의 내부 원소로 가질 수 없는 형
 - 파일형(file type)
 - 보호형(protected type)

58/73.

복합형 (composite type)

■ 배열형의 BNF 정의

```
array_type_definition ::=
    unbounded_array_definition | constrained_array_definition
constrained_array_definition ::=
    array index_constraint of element_subtype_indication
index_constraint ::= ( discrete_range { , discrete_range } )
discrete_range ::= discrete_subtype_indication | range_info
unbounded_array_definition ::=
    array ( index_subtype_definition { , index_subtype_definition } )
    of element_subtype_indication
index_subtype_definition ::= type_mark range <>
type_mark ::= type_name | subtype_name
```

- 원소의 이름은 이산형의 인덱스로 표현

59/73

복합형 (composite type)

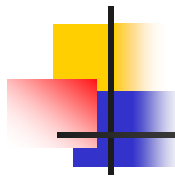
■ 제한 배열형 (constrained array) 선언의 예

```
type DATA_WORD is array (7 downto 0) of BIT;
type TWO_DIM_WORD is array (0 to 7, 0 to 255) of STD_LOGIC;
subtype MemRange is integer range 0 to 255;
type StdLogicRange is array (MemRange) of std_logic;
```

■ 무한 배열형 (unconstrained array) 선언의 예

```
type MEM is array (INTEGER range <>) of STD_LOGIC;
type MEM_MODULE is array (INTEGER range <>) of DATA_WORD;
```

60/73



복합형 (composite type)

■ 레코드형의 BNF 정의

```
record type definition ::=
    record
        element declaration
        { element_declaration }
    end record [ record type simple name ]

element declaration ::=
    identifier_list : element_subtype_definition ;
identifier list ::= identifier { , identifier }
element_subtype_definition ::= subtype_indication
```

- C 언어에서 struct 선언과 같은 역할

61/73




복합형 (composite type)

■ 레코드형 선언의 예

```
type DATE is
    record
        DAY : INTEGER range 1 to 31;
        MONTH : MONTH_NAME;
        YEAR : INTEGER range 0 to 4000;
    end record;
type OVERLAY is
    record
        ALPHA: std_logic_vector(5 downto 0);
        RED, GREEN, BLUE : std_logic_vector(7 downto 0);
    end record;
```

62/73



형(type)

- 접근형(access type)
 - 할당기(allocator)를 사용하여 생성한 객체를 접근하기 위한 용도로 사용
- 할당기(allocator)
 - 객체를 동적으로 할당하는 용도로 사용
 - new 예약어를 이용하여 객체를 할당
 - 객체 할당 시 객체의 형(type)과 초기값을 지정
 - 객체의 이름은 정의되지 않음
 - deallocate 프로시저를 이용하여 객체를 반환
 - 대부분의 논리 합성기에서 논리 합성은 불가능하며 테스트 벤치 생성용으로 대부분 사용

63/73.



접근형(access type)

- 할당기를 이용한 객체 생성의 BNF 정의

```
allocator ::=  
    new subtype_indication | new qualified_expression  
  
subtype_indication ::= type_mark [ constraint ]  
type mark ::= type name | subtype name  
qualified_expression ::= type_mark ' ( expression ) | type_mark ' aggregate
```

- 할당기의 사용 예

```
new BIT_VECTOR;  
new BIT_VECTOR("00110110"); -- qualified expression  
new STRING (1 to 10);
```

64/73

접근형(access type)

■ 접근형의 BNF 정의

```
access_type_definition ::= access subtype_indication
```

```
subtype_indication ::= type_mark [ constraint ]
```

```
type_mark ::= type_name | subtype_name
```

■ 접근형의 사용

- 비접근형의 경우 정교화 단계에서 객체 생성
- 접근형의 경우 시뮬레이션 단계에서 객체 생성
 - 시스템 자원을 효과적으로 사용하게 함
- 접근형 객체는 변수로만 선언 가능
 - 신호로 선언 불가

65/73.

접근형(access type)

■ 접근형 선언의 예

```
type ADDRESS is access MEMORY;
```

```
type IN_BUFFER is array (0 to 31) of BIT_VECTOR(7 downto 0);
```

```
type BUFFER_PTR is access IN_BUFFER ;
```

■ 접근형 사용 예

```
-- 중략
```

```
variable IB_PTR : BUFFER_PTR;
```

```
-- 중략
```

```
IB_PTR := new IN_BUFFER ;
```

- IB_PTR 변수는 IN_BUFFER형을 접근할 수 있는 포인터
 - new 예약어를 이용해서 생성된 객체를 가리킴

66/73

접근형(access type)

■ 접근형 사용의 실제 예

```
process(a)
    type memory is array (3 downto 0) of bit_vector(7 downto 0);
    type MEMORY_PTR is access memory ;
    variable PTR1: MEMORY_PTR; -- 접근형 변수만 생성

begin
    PTR1 := new memory; -- 객체 생성 및 접근형 연결
    for i in 3 downto 0 loop
        for j in 7 downto 0 loop
            PTR1 (i) (j) := '0';
        end loop;
    end loop;
end process;
```

67/73

접근형(access type)

■ 접근형을 사용하지 않은 예

```
process(a)
    type memory is array (3 downto 0) of bit_vector(7 downto 0);
    variable mem: memory; -- 정교화 단계에서 메모리 생성

begin

    for i in 3 downto 0 loop
        for j in 7 downto 0 loop
            mem (i) (j) := '0';
        end loop;
    end loop;
end process;
```

68/73



형(type)

- 파일형(file type)

- 호스트 컴퓨터 내의 파일들을 다루기 위해 사용


- 파일형의 BNF 정의

```
file_type_definition ::= file of type_mark
```

```
type_mark ::= type_name | subtype_name
```

- type_mark : 파일에 포함되어 있는 값의 형을 나타냄
 - 파일형, 접근형, 보호형이 올 수 없음
 - 복합형이 올 경우 원소로 접근형이 올 수 없음
 - 배열형이 올 경우 일차원 제한배열형이어야 함

69/73.



파일형(file type)

- 파일형 선언의 예

```
type text is file of STRING; -- TEXTIO 패키지에 정의되어 있음
```

```
type char_file is file of character;
```

```
type num_file is file of natural;
```

70/73



형(type)

- 보호형(protected type)
 - 순차문으로 이루어진 일정 영역을 여러 개의 프로세서와 공유할 때 해당 영역의 독점적인 사용을 보장하기 위해 사용
 - IEEE Standard 1076-2008 버전부터 추가된 형
 - 보호형 선언과 보호형 본체로 이루어짐
 - 보호형 선언 이후 바로 하나의 보호형 본체가 와야 함

71/73



보호형(protected type)

- 보호형의 BNF 정의

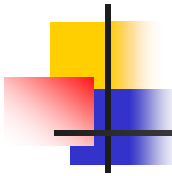
```
protected_type_definition ::=
    protected_type_declaration | protected_type_body

protected_type_declaration ::=
    protected
        protected_type_declarative_part
    end protected [ protected_type_simple_name ]

protected_type_declarative_part ::=
    { protected_type_declarative_item }

protected_type_declarative_item ::=
    subprogram_declaration | subprogram_instantiation_declaration
    | attribute_specification | use_clause
```

72/73



보호형(protected type)

■ 보호형의 BNF 정의(계속)

```
protected_type_body ::=
    protected body
        protected_type_body_declarative_part
    end protected body [ protected_type_simple name ]
protected_type_body_declarative_part ::=
    { protected_type_body_declarative_item }
protected_type_body_declarative_item ::=
    type_declaration | subprogram_body | package_declaration
    | package_body | subprogram_declaration | subtype_declaration
    | constant_declaration | variable_declaration | file_declaration
    | alias_declaration | attribute_declaration | attribute_specification
    | use_clause | group_template_declaration | group_declaration
```