Overtone

NSL Reference Manual

Ver. 1.4 (2012/11/1)

Overtone Corporation

Table of Contents

0. NSL outline 5
Glossary 5
Value 6
Signal line 6
Notation of numerical value. 7
Estimation of bit width in operation 8
Characters usable for module name and signal name 9
Comment out. 9
End of syntax. 9
About clock signal and reset signal. 9
Processing order of NSL processing system . 10
Operator 10
Bit operation 11
Arithmetic operation 11
Shift operation. 11
Relational operation 11
Logical operation. 11
Reduction operation†† 11
Other operations 11
Priority of operation with integer 12
Priority of the operation of only integer and integer variable 13

1. Basic structure 14

2. Declaration of I/O structure element 16
Declaration of data input terminal/data output terminal 16
Declaration of data I/O terminal 16
Declaration of control input terminal/control output terminal. 17

3. Declaration of parameter 19

4. Declaration of internal structure element . 22
Declaration of internal terminal 22
Declaration of register. 22
Declaration of control internal terminal 23
Declaration of submodule 24
Declaration of "procedure" 27
Declaration of state variable 28
Declaration of memory 29
Declaration of structure. 30

5. Atomic action / block 32
Atomic action. 32
Transfer of value 32
Increment and decrement of register 32
Example of basic operation description 34
Conditional operation 45
Block 47
Description of parallel operation block 48
”alt” block. 48
”any” block. 50
”if” block. 51

6. Construct "generate” 53
Structure syntax generate. 53
Structure syntax if 54
Integer variable Integer 55
Temporary terminal variable 56

7. Action description of control terminal 57
Control internal terminal 57
Control input terminal 58
Control output terminal 59
“seq” block. 60
Label , Goto. 61
“while” block 63
“for” block 65

8. Action description of SUBMODULE 68

9. Action description of PROCEDURE 70

10. Action description of state. 72

11. Action description of memory 74

12. Structure 75

13. Interface 77

Appendix 1. Synthesis directive 80
”include” directive 80
”define” “undef” directive 80
”ifdef” / “ifndef” / “else” / “endif” directive. 81

Appendix 2. System task 83
display and monitor 84
time 84
finish 85
readmemh, readmemb 86

Appendix 3. Reserved keyword. 88

Table and Sample contents

Description example 1. Declaration example of I/O structure element 18
Description example 2. Usage example of parameter syntax 20
Description example 3. Declaration of internal terminal/Usage of register declaration 23
Description example 4. Declaration usage example of control internal terminal 24
Description example 5. Declaration example of submodule 26
Description example 6. Parameter usage example in submodule declaration 27
Description example 7. Declaration example of "procedure" 28
Description example 8. Declaration of state variable 29
Description example 9. Declaration example of memory 30
Description example 10. Declaration example of structure 31
Description example 11. Description example of basic atomic action 34
Description example 12. Description example of bit operation 35
Description example 13. Description example of arithmetic operation 36
Description example 14. Description example of shift operation 36
Description example 15. Description example of bit coupling 37
Description example 16. Bit connecting at left-hand side 38
Description example 17. Description example of reduction operation 39
Description example 18. Description example of logical operation 40
Description example 19. Description example of repeat operation 41
Description example 20. Description example of bit clipping out 42
Description example 21. Description example of bit sequence reverse 43
Description example 22. Description example of designation of bit width 44
Description example 23. Description example of sign extension 45
Description example 24. Description example of conditional operation 47
Description example 25. Description example of "alt" block 50
Description example 26. Description example of "any" block 51
Description example 27. Description example of "if "block 52
Description example 28. Structure syntax generate 54
Description example 29. Description example of structure syntax if 55
Description example 30. Description example of partial substitution into temporary terminal 56
Description example 31. Description example of control internal terminal 58
Description example 32. Description example of control input terminal 59
Description example 33. Description example of control output terminal 60
Description example.34 Description example of “seq” block 62
Description example.35 Description example of “seq” block:label 63
Description example 36. Description example of “while” block 64
Description example 37. Detailed operation of “while” block 65
Description example 38. Description example of “for” block 66
Description example 39. Detailed operation of “for” block 67
Description example 40. Description example of submodule syntax 69
Description example 41. Description example of "procedure" 71
Description example 42. Description example of state 73
Description example 43. Description example of memory 74
Description example 44. Description example of structure 76
Description example 45. Description example of interface 78
Description example O-1. Description example of "include” 80
Description example O2-1. Example of system task of "display", "monitor", and "time" 85
Description example O2-2. Example of system task "finish" 86
Description example O2-3. Example of system task "readmemh" 87
Description example O-2. Description example of "define” 81
Description example O-3. Description example of "ifdef/ifndef/else/endif” 82
Table 0-1. Operator 11
Table 0-2. Priority of operator 12

Table 1-1. Basic structure of NSL 14
Table 2-1. Table of I/O structure element 16
Table 5-2. The block that can be used anywhere 48
Table 5-3. The block that can be used in a special condition 48
Table O2-1. Corresponding table by language of system task 83
Table O2-2. Corresponding table by language of system task 83
Table O2-3. Format specifier of system task 84

Chapter 0

0. NSL outline

Glossary

HDL (Hardware Description Language)

Computer language for hardware design that is used when ASIC and FPGA, etc. are designed

NSL (Next Synthesis Language)

Hardware Description Language based on the message driven.

Syntax

Syntax that is described in accordance with syntax of NSL language

Module

Bundle of minimum units that are described by NSL

Action

Part that operates with one clock

Action statement

Element in a module, which represents action

Common action description

Part that always operates with every clock among the action descriptions except "procedure" and "function"

Atomic action

Minimum unit of action that is described in the action description part of a module

Proc (Procedure)

Processing that consolidates actions that are used repeatedly many times

Func (Function)

Bundle of processes that can be called from outside and inside of a module via a control terminal by consolidating action descriptions in a unit

Instance

In a submodule syntax, "substance" when a module to be lower is declared in a higher module

I/O structure element

Element that combines the signal to be input into module and the signal to be output from module

Data terminal

Signal line that inputs and outputs numerical value

Register

Memory element that is used in an electronic circuit and HDL, etc. It is used to assist the operation, and store the value, etc.

Memory

It is made to save multiple data by arranging the registers.

Control terminal

Signal line to activate "function" that resides in a module

Internal structure element

Element that configures module itself such as "register" and "procedure", etc.

Block

Part that is put in {} in action description. It includes the kinds of "par", "any", "alt", "if", "seq", etc., and each block operates separately.

Hierarchic structure

Structure that overlaps in stratified like the floor in a building. Achieving a hierarchic structure makes it easy to structure a large-scale system. Submodule syntax is used for NSL to achieve a hierarchic structure.

Block top level

Independent block that is not nested by any block

MSB/LSB (Most Significant Bit/Least Significant Bit)

As for NSL language, it assumes that the leftmost is MSB (Most Significant Bit), and the rightmost is LSB (Lest Significant Bit).

Compile, Compiling operation

It is indicated to generate a lower language source from NSL language source using NSL conversion engine. Or, its operation is indicated.

Value

In NSL language, each element can take the following values.

Data terminal: "1", "0", "Z" (Hi-Z), "U" (Undriven), "X" (Indefinite)

Register and memory: "1", "0", "U" (Undriven)

Control terminal and task: "1", "0"

Signal line

Signal line is used to exchange the data between modules.

In NSL language, "1", "0", and "X" can be transmitted when one-bit "single-line signal" is prepared.

In addition, "bus" that binds up multiple single lines is prepared to send more information.

Digit number in bit of bus defines the leftmost as Most Significant Bit (MSB), and the right-most as Least Significant Bit (LSB), and it is counted from LSB as the first digit.
For example, when data of "101010" is output to a six-bit signal line, the first digit is 0, the second for 1, the third for 0, the forth for 1, fifth for 0, and the sixth for 1.

It is also possible to clip out optional bits of a bus to transfer them to other terminal, and to check only specific bits of the bus to determine the next operation.
However, it is not permitted to write in a specific bit of a bus such as preparing 4-bit register, and writing “1” in only the second bit, for example. This is due to the reasons, "the readability is remarkably decreased easily" and "it is easy to become a hotbed of defective operation".

Declaration method of bus is explained in Chapter 4, and action description of bus is done in Chapter 5.

Notation of numerical value

Notation of numerical value in NSL has two kinds of Verilog HDL type and C language type.
Notation of Verilog HDL type describes as follows:

<Bit_width>'<Radix_Character><Value>

For example, 12 in decimal numbers is expressed by the binary numbers in 4-bit width as follows:

```
4’b1100
```

Binary is abbreviated as "b".
4-bit, 5 in decimal numbers is expressed as follows:

```
4’d5
```

In this way, it describes **Binary number: b, Octal number: o, Decimal number: d, and Hexadecimal number: h** respectively in Verilog HDL type.
And, it describes in C language type notation as follows:

0<Radix_Character><Value>

For example, 8 in decimal number is expressed in binary number of 4-bit width as follows:

```
0b1000
```

In this notation of C language type, it describes **Binary number: b, and Hexadecimal number:x** respectively.
Though "value" need not be matched to the bit width in Verilog HDL type, in C language

type, the bit width is decided by the width in "value" described.
For example, it is decided to 8 bits for 0x00, and 4 bits for 0b0000.

In addition, "_" (Underscore) can be used to represent Value. Underscore is disregarded when compiling. It can be used to raise the readability of numerical representation in multiple digits.
Example:
8'b01011010 → 8'b0101_1010
32'hA9876543 → 32'hA987_6543
0x12345678 → 0x1234_5678

Estimation of bit width in operation

In NSL, only the signal value in a determined width is allowed for the operation of the signals in a transmission to a terminal and a register. It is allowed to use an integer (and an integer variable) for the 2nd term of an operation in a part of the operations in which operation operand is constrained to the numbers in the same bits as an exception. It is because NSL processing system presumed the number of bits, and changes it into a signal value in a determined width. Since the same presumption is performed also in a transmission of a value to signal and register, a setup of initial value of register and memory, and memory addressing, it realizes unambiguous description as reducing the amount of description.

An integer can also be used for a term of a formula when the number of bits can be presumed.

- As for the right-hand side (actual argument of proc or func_xxx is included) of a transmission to terminal or register, an integer or an integer variable is changed into a constant in the bit width concerned to be transferred since the bit width at the transmitting destination is determined.
- As for the operations of +, -, &, |, and ^, the bit width of 2nd term is presumed by the bit width of 1st term. So, an integer or integer variable can be used for the 2nd term. Here, in the case of 1st term and 2nd term being an integer or an integer variable, it is treated as an integer operation, and results in an integer without bit width.
- Conditional operation of if (...) ... else ... can use an integer or an integer variable for both true and false value when the bit width is determined in the description place (the right-hand side of a transmission to a terminal or a register, and the 2nd term of an operation of the same bit width each above-mentioned).

Characters usable for module name and signal name

In NSL, the following characters are usable for identifier such as module name and signal name, etc.

- English one-byte characters (A-Z/a-z)
- Figure (0-9)
- Underscore "_" (Second character or later)

However, a double mark of the underscore "__" is prohibited to prevent misreading.

Comment out

Comment out in NSL is compatible with C language, and two kinds of representations are permitted.

Single line comment describes as follows:

```
// Comment
```

Area comment:

```
/*Comment*/
```

Notes: Though single line comment can be described in an area comment, the area comment cannot be nested.

End of syntax

In NSL, an end of syntax is represented by semicolon ";".

When a syntax that is a minimum unit of the description such as declaration of element, and description of operation, etc. is ended, the end is determined using the semicolon as follows.

```
Declaration of structure element ;
Description of  action ;
```

It is also possible to do multiple descriptions by separating with a semicolon, however it is not recommended because the readability falls remarkably.

About clock signal and reset signal

Verilog HDL and VHDL are a signal-oriented language that varies the behavior with a certain signal. NSL, on the other hand is a language based on the message driven that decides the operation by describing the behavior of a module first.

NSL automatically provides a synchronization signal for module, and creates a module that operates with a single-phase or a multi-phase clock.

It also provides a reset signal for the circuit as well as the synchronization signal at the

same time.

When it is not specified at all, the clock signal is automatically synthesized with the name of “m_clock”, and the reset signal with “p_reset”. Both of the clock name and the signal name can be changed by a compilation option.

Processing order of NSL processing system

In NSL processing system, it is processed in the following 3 stages:

• Expansion of directive by preprocessor

• Structural expansion

• Synthesis of circuit description by structure elements

Expansion of directive by preprocessor is explained in Appendix 1.

Structural expansion is a syntax which is expanded in order of the described regardless of the element of a circuit description. Using a structural expansion makes it possible to reduce the amount of description when producing multiple, similar circuits. For details, it explains in Chapter 6. An integer (and an integer variable) determines a value by the processing called structural expansion.

Operator

Though operator of NSL is basically compatible with Verilog HDL, a part of relational operations, and division are excluded.

The unique NSL operation includes sign extended arithmetic. Operator usable for NSL is listed in Table 0-1.

(Hereafter, the semicolon used in the table is used as a sign to delimit the syntax used in NSL from its meaning.)

Table 0-1. Operator

Bit operation

& : Bit operation AND
| : Bit operation OR
^ : Bit operation EX-OR
~ : Bit operation NOT

Arithmetic operation

+ : Arithmetic addition
- : Arithmetic subtraction
* : Arithmetic multiplication
++ : increment †
-- : decrement †

Shift operation

>> : Right shift
<< : Left shift

Relational operation

== : Equal
!= : Not equal
> : Greater than
< : Less than
>= : Greater than or Equal to
<= : Less than or Equal to

Logical operation

! : Logical NOT
&& : Logical AND
|| : Logical OR

Reduction operation††

& :Reduction AND
~& : Reduction NAND
| : Reduction OR
~| : Reduction NOR
^ : Reduction EX-OR
~^ : Reduction EX-NOR

Other operations

{sigA,sigB,...,sigX} : Bit connecting
num#sig : Sign extension
num'(sig) : Specifying bit width (Extension of bit width without sign)
sig[num] : Bit clipping operation
sig[numA:numB] : Bit clipping operation
numA{numB} : Repeated operation
if(condition) sigA else sigB : Conditional operation

†The following clock reflects the result of an increment and a decrement.

††Reduction operation can be used only for the multi-bit signal. (It cannot be used for 1-bit signal.)

Moreover, the priority in Table 0-2 exists in the NSL operator.
When multiple operators are described in an expression of a sentence, the operation is sequentially executed from the operator with higher priority.
"( )" is used to raise the priority of expression in an expression.

Table 0-2. Priority of operator

Priority	Operator	Explanation	Example
1 (High)	{sigA,sigB,...,sigX}	Bit connecting	{0b0,0b1,0b01} → 0b0101
	numA{numB}	Repeated operation	4{0b0} → 0b0000
	num#sig	Sign extension	8#0b0101 → 0b00000101
	num'(sig)	Specifying bit width (Extension of bit width without sign)	8'(0b1101) → 0b00001101
	&	Reduction AND	&0b1111 → 1, &0b0111 → 0
	|	Reduction OR	|0b0000 → 0 , |0b0111 → 1
	^	Reduction EX-OR	^0b1111 → 0 , ^0b0111 → 1
	~	Bit operation NOT	~0b0101 → 0b1010
	!	Logical NOT	if (!a)
2	*	Arithmetic multiplication	q = a * b
3	+	Arithmetic addition	q = a + b
	-	Arithmetic subtraction	q = a - b
4	<<	Left shift	q = 4'b1010. (q << 1) = 4'b0100
	>>	Right shift	q = 4'b1010. (q >> 1) = 4'b0101
5	<=	Less than or Equal to	if (a <= b)
	>=	Greater than or Equal to	if (a >= b)
	<	Less than	if (a < b)
	>	Greater than	if (a > b)
6	==	Equal	if (a == b)
	!=	Not equal	if (a != b)
7	&	Bit operation AND	a = 4'b0110. b = 4'b1100. a & b = 4'b0100
	^	Bit operation EX-OR	a = 4'b0110. b = 4'b1100. a ^ b = 4'b1010
8	|	Bit operation OR	a = 4'b0110. b = 4'b1100. a | b = 4'b1110
9	&&	Logical AND	if (a && b)
10	||	Logical OR	if (a || b)
11 (Low)	if () sigA else sigB	Conditional operation	q = if (a>b) a else b

Priority of operation with integer

Although an integer bit width is determined at the time of structural expansion as above-mentioned, it is necessary to make the 1st term a signal and the 2nd term an integer for the operation (*) performed by the same bit width each in order to determine the bit width. It is judged also seeing the lead 2 terms in case of 3 terms or more, however when a formula put in parenthesis ( ) is placed, the priority is given considering it as indication.

(*) Operation by each operator of +, -, <, <=, >, >=, ==, !=, |, ^, and &

Priority of the operation of only integer and integer variable

As for the operation of only integer and integer variable, it is operated from the left regardless of the priority of the operator. Please use the parenthesis ( ) in the portion related to an operation order.

Example

```
Integer I;
Variable v[8];
```

I=2+3*4;

V=I;

In this case, not 14 but 20 goes into v.

Chapter 1

1. Basic structure

In this chapter, explains the basic system of NSL is explained.

Basic structure in the description of NSL consists of the system in Table 1-1.

(Hereafter, the syntax enclosed with <> in Reference may be omitted.)

Table 1-1. Basic structure of NSL

```
declare Modulename < interface > {
   <Parameter declaration list>
   <Declaration of I/O structure element list>
}
module Modulename {
   < Declaration of internal structure element list >
   < Action description list >
      - <Common action discription part>
      - <Function discription part>
      - <Procedure discription part>
}
```

NSL consists of the "declare" syntax and the "module" syntax.

"Declaration of I/O structure element" and "Parameter declaration" are executed in the "declare" syntax.

"Declaration of I/O structure element" is explained in Chapter 2, and "Parameter declaration" in Chapter 3.

“Declaration of internal structure element" and "Action description" are executed in the "module" syntax.

"Declaration of internal structure element" is explained in Chapter 4, and "Action description" in Chapter 5~10.

It is assumed one module in NSL combining the "declare" syntax and the "module" syntax.

The “declare” syntax and the “module” syntax need not be necessarily written sequentially. The compilation is accepted if a set of the both exists in the same file when compiling.

Therefore, it is also possible to collect only the "declare" and put them into a separate file, and to "include" it as a header file.

The modifier called interface or simulation can be attached to declare. The interface modifier is used to specify the signal name of a clock and a reset explicitly to a module calling as a submodule.

When a declare of submodule was performed without an interface modifier, the signal names of m_clock for a clock and p_reset for a reset are used. Please refer to Chapter 13. Interface for details. The simulation modifier is used when a system task is used for a simulation within the module. Please refer to Appendix 2. Task system task for details.

Chapter 2

2. Declaration of I/O structure element

In this chapter, it explains about the declaration of I/O structure element.

I/O structure element indicates the data terminal and the control terminal that is input or output to the module.

Declaration statement is needed to use the I/O structure element in NSL module.

The following Table 2-1 shows the I/O structure element in NSL.

Table 2-1. Table of I/O structure element

input	Data input terminal
output	Data output terminal
inout	Data input and output terminal
func_in	Control input terminal
func_out	Control output terminal

In the I/O structure element, it is possible to describe multiple signal names, and delimit them with a comma "," when multiple signals are declared in a declaration.

Declaration of data input terminal/data output terminal

Data terminal indicates a signal line that sends and receives a data of numerical values.

Data input terminal/data output terminal indicates a data terminal that is directed in the input direction/output direction of each module.

As for both of the data input terminal and the data output terminal, it can set "bit width" by enclosing it with **[ ]** in the back of a signal name.

At the bit width is omitted, it becomes a signal line in 1-bit width.

Data input terminal, data output terminal is declared as follows.

```
input Input_signal_name[bit_width]
output Output_signal_name[bit_width]
```

Declaration of data I/O terminal

The data I/O terminal indicates a line that can correspond to both the input and the output.

Bit width of the data I/O terminal can also omit the bit width just as the declaration of data input terminal/data output terminal.

The following shows the declaration method of the data I/O terminal.

```
inout I/O_signal_name[bit_width]
```

Declaration of control input terminal/control output terminal

In NSL, the control terminal can be described besides the data terminal.
The control terminal indicates a line to activate "function" that resides in a module from the inside and the outside of the module.
Moreover, "function" is the one in which a certain action description was consolidated.
Description method of the "function" is explained in Chapter 6.

The control input terminal is a control terminal used to activate the "function" from an external module, and the control output terminal is a control terminal used to activate an external "function". The bit width cannot be set to the control terminal.
The control input terminal/control output terminal can have a dummy argument be accompanied with one signal name. Data of an actual argument given in an action description is transmitted to the element specified for a dummy argument. That is, it becomes an undertaking spot to which the actual argument is temporarily transferred.
When multiple dummy arguments are accompanied, a comma "," is feasible to delimit the dummy arguments.
Declaration of the control input terminal/control output terminal is described respectively as follows.

```
func_in Control_input_signal_name(<dummy_argument>, <dummy_argu-
ment>, <dummy_argument>, <dummy_argument>, ...)
func_out Control_output_signal_name(<dummy_argument>, <dummy_ar-
gument>, <dummy_argument>, <dummy_argument>, ...)
```

At this time, the dummy argument accompanied with the control input signal is limited to the data input terminal, and the dummy argument accompanied with the control output signal is limited to the data output terminal.
It is recommended to write dummy argument to improve the readability, and it is also possible to omit it.

The function can have a terminal to return a return value. Since a return value terminal is a data terminal, it can also have a bit width. In the case of control input/output terminal, the return value terminal is a data input/output terminal, but note that the direction of the control terminal and the return value terminal becomes reverse.
The description method of the function with a return value terminal is as follows.

```
func_in Name of control input signal (<Dummy argument>, <Dummy
argument>, <Dummy argument>, ... ) : Return value output terminal
```

(or, input/output terminal)

func_out Name of control output signal (<Dummy argument>, <Dummy argument>, <Dummy argument>, ...) : Return value input terminal (or, input/output terminal)

Declaration example of I/O structure element is shown in the description example 1.

Description example 1. Declaration example of I/O structure element

```
declare test_inout {
  input    a ;     //Data input terminal a is declared by 1 bit.
  output   b[4] ;  //Data output terminal b is declared by 4 bits.
  inout    c[12] ; //Data I/O terminal a is declared by 12 bits.
  func_in  d ;     //Control input terminal d is declared.
  func_in  e(a) ;
         //Control input terminal e with dummy argument a is declared.
  func_out f(b) ;
         //Control input terminal f with dummy argument b is declared.

  input reti[8];
         //Declaration of data input terminal in 8 bits
           for return value terminal
  output reto[8];
         //Declaration of data output terminal in 8 bits
           for return value terminal
  func_in g ; reto;
         //Declaration of control input terminal g
           by setting return value reto
  func_out h(b) ; reti;
         //Declaration of control output terminal h
           by setting dummy argument b and return value reti
}
module test_inout {
  //Describe internal structure elements
  //Describe actions
}
```

It becomes possible to use each signal of a, b, c, d, e, and f in the action description to be described by declaring the I/O structure element like this.

Chapter 3

3. Declaration of parameter

In this chapter, it explains about the declaration method of the parameter syntax.

In NSL, it is possible to design a circuit by configuring multiple modules in a hierarchical structure. It can use not only what is described in NSL but also what is done in Verilog HDL/VHDL/SystemC for a module to be lower. The parameter syntax is provided to control a generation of the "instance" of a module that is described in a parametric syntax with Verilog HDL/VHDL/SystemC. (A module described in NSL doesn't support the parameter syntax. In this case, a parameter is given by "define" option. Please refer to the appendix for the "define" option.)

Declaration example of parameter

```
param_int Parameter_name // Integer type parameter (signed 32bit
integer)
param_str Parameter_name // Character string type parameter (No
limitation. It depends on the memory in processing environment.)
```

To create a lower module, please refer to the submodule syntax to be described in Chapter4, and Chapter 7.

Description example 2 shows a usage example of the parameter syntax.

Description example 2. Usage example of parameter syntax

```
module lower_mdl (
   a ,
   b ,
   q ,
   Add
);

   parameter NofA = 4 ;
   parameter NofB = 6 ;

   input  [NofA-1:0]       a ;
   input  [NofB-1:0]       b ;
   output [(NofA+NofB-1):0]  q;
   input  Add;

   assign #1 q = ( Add == 1'b1 ) ? ( a + b ) : 0 ;

endmodule
```

It is assumed that there is already a following module written in Verilog HDL.

```
/* Include parameter table */
#define    Num_of_A  8           // Num_of_A is defined for submodule.
#define    Num_of_B  12          // Num_of_B is defined for submodule.
#define    Num_of_Q  (Num_of_A+Num_of_B)
                                 // Num_of_Q is defined for submodule.

/* 'Parametalized Adder' */
declare parametalized_adder interface {
                                 // Submodule is declared by interface.
   param_int NofA ;              // Parameter NofA id declared.
   param_int NofB ;              // Parameter NofB id declared.

   input     a[Num_of_A] ;
   input     b[Num_of_B] ;

   output    q[Num_of_Q] ;
   func_inAdd(a, b) ;
}
```

When this module is used as a lower module, the lower module declaration of the module to be higher is as follows.

```
/* Declare  "TOP module" */
declare   TOP_module {          // TOP_module is declared
   input     add_a[Num_of_A] ;
   input     add_b[Num_of_B] ;

   output    result_q[Num_of_Q] ;
   func_inAdd(add_a, add_b) ;
}

/* Equation of 'TOP module' */
module TOP_module {

   // Instantiate submodule and send parameter
   parametalized_adderu_adder( NofA = Num_of_A, NofB = Num_of_B ) ;

   function  Add {
      // Submodule u_adder is exceuted.
      result_q = u_adder.Add(add_a, add_b).q ;
   }
}
```

And, the lower module declared in the “declare” is called from the higher module as follows. At this time, integral value or character string can be transferred to the lower module by adding a parameter to the "instance" name that was made substantial.

Chapter 4

4. Declaration of internal structure element

In this chapter, it explains about the declaration method of internal structure element.

The internal structure element indicates the structure elements in a module such as "wiring", "register", "control internal terminal", "state variable", "procedure", and "memory", etc., and it becomes possible to use the internal structure elements in an action description to be described by declaring.

It is possible to declare multiple internal structure elements at a time by describing the multiple internal structure elements and separating them with a comma ",".

Declaration of internal terminal

An internal terminal becomes a syntax that provides "wiring" to arrange the data on multiple terminals.

Declaration of the internal terminal is “wire”, and it is defined as follows.

```
wire Internal_terminal_name[bit_width]
```

A value transferred to wire is valid within the same clock cycle.

As for the "wire”, it can set a bit width, and omit it, too.

It is treated as "X" (indefinite) while nothing has been input to the "wire".

Declaration of register

Register is a memory element that memorizes the last input value on the leading edge of clock signal. When the value is transferred to the register, the value is recorded at the next clock.

Moreover, it is also possible to prepare a register in optional bit width, and to set an initial value in NSL when declaring.

Register declaration is executed in the following method.

```
reg Register_name[bit_width] = <initial_value>
```

Here, the declaration of internal terminal and the register declaration are shown in Description example 3.

Description example 3. Declaration of internal terminal/Usage of register declaration

```
declare test_reg {
  // Describe I/O structure element
}
module test_reg {
  wire wire_a [16] ; // Internal terminal wire_a is declared by 16 bits.
  reg reg_b[4] ;     // Register reg_b is declared by 4 bits.
  reg  reg_c, reg_d, reg_e ;
    // Register reg_c, reg_d and reg_e are declared at once
  reg  reg_f[4] = 4'b1010 ;
    // Register reg_f is declared by 4bits and initialize 1010
  // Describe actions
}
```

It becomes possible to use “wire” and “reg” respectively in the action description to be described by describing the declaration of internal terminal and the register declaration like Description example 3.

It is also possible to omit setting an initial value. In that case, "X" (indefinite) is entered when resting.

Declaration of control internal terminal

Control internal terminal is a signal of control terminal in a module, and it can be called only in a described module.

Also, the control internal terminal can be related to a function. (The action description of function is explained in Chapter 6.) It can allow the control internal terminal to have multiple dummy arguments.

Declaration method of the control internal terminal is as follows.

```
func_self Control_internal_terminal
```

And, the declaration method when a dummy argument is given is described as follows.

```
func_self Control_internal_terminal(<dummy_argument>, <dummy_argument>, <dummy_argument>, ...)
```

Only "wire" defined by the declaration of internal terminal can allow a dummy argument to accompany a control internal terminal.

It is recommended to write a dummy argument to improve the readability. (It is also possible to omit it.)

It describes as follows when describing also return value terminal.

func_self Name of control internal terminal: Name of return value terminal

func_self Control internal terminal (<Dummy argument>, <Dummy argument>, < Dummy argument>, …) : Name of return value terminal

Return value terminal can be used only for wire defined by a declaration of internal terminal.

The following Description example 4 shows a declaration example of the control internal terminal.

Description example 4. Declaration usage example of control internal terminal

```
declare test_func_self {
   //Describe I/O structure element
}
module test_func_self {
   wire a [4], b[2] ;
   func_self funcC(a,b) ;
   // Describe actions
}
```

The operation of a control internal terminal can be described like this by declaring a control internal terminal.

Please refer to Chapter 6 for the action description of the control internal terminal.

Declaration of submodule

In NSL, submodule syntax is provided to describe a hierarchic structure. When the submodule syntax is used, a submodule declaration is executed in a higher module.

When the submodule syntax is used, it is indispensable to prepare a lower module to be a submodule. The module that is scheduled to be a submodule is also called "template".

The submodule declaration is made substantial by specifying a module name to be a template, and giving the name only in a higher module. This module of template that was made substantial is called "instance".

Production of instance makes the following possible.

• Inputting a value to data input terminal

• Calling control input signal
• Receiving a coming value of data output terminal
• Receiving a control output signal

The submodule syntax is declared in the following method.

```
Module_name Instance_name
```

Moreover, multiple "instances" of a submodule can be prepared describing multiple "instance" names by separating them with a comma.

```
Template name instance name 1, instance name 2, instance name 3,
…
```

Furthermore, multiple templates can be substantialized (multiplicity is given) also by the method of attaching the number of instances to the instance name with [ ].

```
Template name instance name [3]
```

Since the number of the suffix is set to the number of instances, only a natural number (1, 2, 3, …) can be set inside [ ].

Description example 5. Declaration example of submodule

```
// Submodule "test_sub" that becomes template.
declare test_sub {
   input a ;
   output f ;
}
module test_sub {
   // Describe actions
}

//"test_module" becomes top module.
declare test_module {
   input test_in ;
   output tset_out ;
}
module test_module {
   test_sub SUB ; // Template module "Test_sub" is made to instance by the
name "SUB" in test_module and defined.

test_sub SUB1,SUB2, SUB3; //Substantializing 3 modules of SUB1, SUB2, and
SUB3 at a time

      test_sub SUB_Array[3];     //3 modules of SUB_Array[0], SUB_Array[1],
SUB_Array[2] are substantialized at a time.

   // Describe actions
}
```

A submodule can be used in a higher module by being described as Description example 5. Moreover, the parameter syntax is used when a parameter of a lower module is operated from a higher module.

An example using the parameter syntax of a submodule is shown in the following Description example 6.

Description example 6. Parameter usage example in submodule declaration

```
declare test_sub {
   param_int INT ;
   param_str CHA ;
   input a ;
   output f ;
}
module test_sub {
   // Describe actions
}

declare test_module {
   input test_in ;
   output tset_out ;
}
module test_module {
   // Template module "Test_sub" is defined by the instance "SUB1"
   // in test_module.
   test_sub SUB1 ;

   //"14" is passed to the parameter "INT" of the instance "SUB2".
   test_sub SUB2(INT = 14) ;

   //String "NEKO" is passed to the parameter "CHA" of the instance "SUB3"
   test_sub SUB3(CHA = "NEKO") ;

   // Describe actions
}
```

Describing in this way, it is possible to pass a different parameter to each "instance" "SUB1", "SUB2", and "SUB3".

That is, it can start an "instance" with the same structure even giving various data and states to it when generating it. Please refer to Chapter 7 for the action description of the submodule syntax.

Declaration of "procedure"

The "procedure" is a syntax to provide the control that uses state transition, pipeline, and sequential circuit, and has an area where the operation only for the "procedure" is described excluding the common action description.

When the "procedure" was once activated, it transits to another "procedure", or continues to operate until end of the "procedure" is declared.

To declare a “procedure”, the following description method is executed. It is possible to accompany dummy argument when declaring, and multiple dummy arguments can be also given by separating them with a comma.

```
proc_name Procedure_name(<dummy_argument>, <dummy_argument>,
<dummy_argument>, ...)
```

The dummy argument that can be accompanied with a "procedure" is only register which is declared by ‘reg’ syntax.

Description example 7. Declaration example of "procedure"

```
declare test_proc {
  //Describe I/O structure element
}
module test_proc {
  reg r1, r2, r3 ;

  // Procedure proc_A is declared.
  proc_name proc_A() ;

  // Procedure proc_B with dummy argument r1 is declared.
  proc_name proc_B(r1) ;

  // Procedure proc_C with dummy arguments r2 and r3 is declared.
  proc_name proc_C(r2, r3) ;

  //  Describe actions
}
```

Declaring in this way makes it possible to use the "procedure" in a module. Please refer to Chapter 8 for the action description of the "procedure".

Declaration of state variable

State variable is the syntax to define a state transition machine, and it is called "state". The state variable can be given to an action description by declaring the state. The state is declared in the action description part though it is explained details in Chapter 9.

The declaration of a state can be executed not only the common action description part but also in the "procedure", and when it was declared in a "procedure" it can be used only in the "procedure".

Moreover, the state described at the head of the declaration is activated when the module is activated.

The state declaration method is executed as follows.

```
state_name Statemachine_name
```

Description example 8 shows a declaration example of the state variable.

Description example 8. Declaration of state variable

```
declare test_state {
   //Describe I/O structure element
}
module test_state {

   // Common action description part
   {
  //Declare states. It executes from State described at first.

   // State state1, state2 and state3 are declared.
   state_name state1, state2, state3 ;
   }
}
```

Declaring a state in this way makes it possible to use the state in the common operation part.

Please refer to Chapter 9 for the action description of a state variable.

Declaration of memory

Memory is the syntax that organizes and memorizes a large amount of information, and declares in the declaration list of internal structure elements.

The value is reflected to the relevant address at the clock next to the memory written clock.

The declaration method of the memory is as follows.

```
mem Memory_name[address_number][memory_bit_width]
```

It is also possible to initialize the memory when declaring.

The memory initialization method is as follows.

```
mem Memory_name[address_number][memory_bit_width] = {data_at_address0, data_at_address1, ... data_at_addressX}
```

In addition, the memory at an address where no initialized data exists is initialized to 0 when the number of the initialized data is less than the number of the memory addresses.

Declaration example of the memory is shown in Description example 9.

Description example 9. Declaration example of memory

```
declare test_mem {
}
module test_mem {
   // Declare a memory without initalize
   mem memory1[1024][32] ;

   // Declare a memory with initalize
   mem memory2[4][8] = { 8'hFF, 8'hAA, 8'h12, 8'h32 } ;
}
```

When being described as Description example 9, it declares memory1 (Not initialized) with the number of addresses 1024 and bit width 32, and memory2 (Initialized with 0xff, 0xaa, 0x12, and 0x32) with the number of addresses 4 and bit width 8.
Please refer to Chapter 10 for the action description using the memory.

Declaration of structure

In NSL, structure can be used so that the signals in multiple bit widths are treated collectively. Using structure makes it possible to treat the signals in various bit widths collectively.

The structure first declares. A declare is described outside module (in a portion not inside declare, nor module). The signal type is not specified at this point. In addition, please note that ";" is required at the end of struct declaration.

```
Name of structure{
   Member of structure 1;
   Member of structure 2;
   Member of structure 3;
   :
   Member of structure x;
};
```

The ones previously declared among the members of a structure are arranged at the top of the structure.

Then, instance of the structure is declared in the module. When instance is declared, it specifies whether the kind of signal is reg or wire. The details for the instance declaration of structure are explained in Chapter 12.

Declaration example of structure is listed in the following Description example 10.

Description example 10. Declaration example of structure

```
struct config_addr { // config_addr
   p_enable;
   p_reserve[7];
   p_bus[8];
   p_device[5];
   p_func[3];
   p_regaddr[6];
   p_zero[2];
}; // ; is required

declare {
   input p[32];
}
module test_mem {
   config_addr reg caddr_1 ;
   // config_addr Structure config_addr is declared as instance caddr_1

caddr_1 := p;
}
```

Chapter 5

5. Atomic action / block

Since NSL is a hardware description language, operation is basically executed in parallel. A minimum operation unit of the individual action executed in parallel is called “atomic action”. Moreover, the action is described by changing the behavior with a unit of "block" in NSL. The atomic action and the block are explained in this chapter.

Atomic action

An action is called atomic action in NSL.

Then, it explains details of the main atomic actions step by step.

Transfer of value

The atomic action is based on "transfer" in NSL.

"Transfer" indicates that a value is input from a terminal and a register, etc. to other terminal and register, etc.

The following Tables 5-1 shows types of the transfers.

Table 5-1. Type of transfer

wire/output/inout transfer	=
reg transfer	:=

"=" is used to transfer it to "wire”, “output”, and “inout”.

```
Transfer_destination = Transfer_source
```

In addition, ":=" is used to transfer a value to a register (reg). Transferring direction is the same as the above.

```
Register_of_transfer_destination := Transfer_source
```

Increment and decrement of register

Increment indicates that 1 is added to a variable value, and it is rewritten to the value, and decrement does that 1 is subtracted from a variable value, and it is rewritten to the value. In NSL, there is the atomic action that can increment and decrement to the register.

Notes: ++ and -- are not an operator. A varied numerical value is reflected in the register at the next clock.

Using "++" when add one to the register, and "--" when subtract one from it, it is described as follows.

Description example 11 shows an example that uses the basic atomic action that has come out so far.

```
++Register_name (pre-increment)
--Register_name (Pre-decrement)
Register_name++ (post increment)
Register_name-- (post decrement)
```

It can also use increment and decrement for the right-hand side of a formula. Even in this case, the next clock reflects the value to which increment, decrement were given.
For example,

```
i = r++
```

The above process is that the value of r is first transferred to i, and r+1 is transferred to r at the next clock.

```
i = ++r
```

The above process is that the value of r+1 is first transferred to i, and r+1 is transferred to r at the next clock.
Examples using the basic unit action so far are shown in Description example 11.

Description example 11. Description example of basic atomic action

```
declare test_par {
   input in_a[4] ;
   output out_b[4] ;
   output out_c[4] ;
   output out_d[4] ;
   output out_e[4] ;
}
module test_par {
   wire wire_i[4] ;
   reg r1[4], r2[4] = 4'd0, r3[4] = 4'd0 ;

//Common action description is begin.
   r1 := in_a ;          // In_a is transferred to r1.
   out_b = 4'b1010 ;     // 10(4'b1010) is transferred to out_b.
   out_c = r1 ;          // R1 is transferred to out_c.
   wire_i = 4'b1111 ;    // 15(4'b1111) is transferred to wire_i.
   out_d = wire_i ;      // Wire_i is transferred to out_d.
   out_e = wire_i ;      // Wire_i is transferred to out_e.
   r2++ ;                // Increment r2.
   r3-- ;                // Decrement r3.
}
```

In Description example 11, eight atomic actions are described in the common action description part, and all of them are executed at the same time.

Example of basic operation description

In the foregoing section, it explained about "transfer" that is important for the atomic action of NSL.

The following presents an example of the operation that forms the basis of action description. First of all, an example of bit operation that is the basic of HDL is shown in Description example 12.

Description example 12. Description example of bit operation

```
declare test_bit_exec {
   input inA[8] ;
   input inB[8] ;
}
module test_bit_exec {
   reg r1[8], r2[8], r3[8], r4[8] ;

   // Logical OR of each bit of inA and inB is transferred to r1.
   r1 := inA | inB ;

   // Logical AND of each bit of inA and inB is transferred to r2.
   r2 := inA & inB ;

   // Logical NOT of each bit of inA is transferred to r3.
   r3 := ~inA ;

   // Logical NOT of logical OR of inA and logical NOT of inB
   // is transferred to r4.
   r4 := ~( inA | ~inB ) ;
}
```

Description example 12 shows an example of logical OR and logical AND of the bit operation, and logical NOT.
The bit operation is an operator to which the logical operation result of each bit is output.

For example, when signals A and B in four bits are 1010 and 1001 respectively, A&B becomes 1000, and A|B does 1011.
In this way, as for the bit operation, it is operated 1 bit to 1 bit at each digit of each bit. For the bit operation, the bit width of each operation object should be the same.

Next, an example of arithmetic operation is shown in Description example 13.

Description example 13. Description example of arithmetic operation

```
declare test_math {
   input inA[16] ;
   input inB[16] ;
}
module test_math {
   reg r1[16], r2[16], r3[32] ;

   r1 := inA + inB ; // InA and inB are added and it transfers to r1.
   r2 := inA - inB ; // InB is subtracted from inA and it transfers to r2.
   r3 := inA * inB ; // InA and inB are multiplied and it transfers to r3.
}
```

Description example 13 shows an example of "addition", "subtraction", and "multiplication" of the arithmetic operation.

The sum of inA and inB is transferred to r1, the difference between inA and inB is done to r2, and the product of inA and inB is done to r3.

As for the addition and the subtraction, the bit width of each operation object should be the same.

In case of the multiplication, bit width of the output destination of an operation result should secure an adequate width beforehand because the "sum of bit width of operation objects" becomes the bit width of the operation result.

Next, an example of shift operation is shown.

The shift operation is an operation that shifts the objective signal line and register right and left by optional number of bits.

Description example 14. Description example of shift operation

```
declare test_shift {
   input inA[16] ;
}
module test_shift {
   reg r1[16], r2[16], r3[16] ;

   //InA is shifted 5 bits to the right and it transfers to r1.
   r1 := inA>>5 ;
   // InA is shifted 6 bits to the left and it transfers to r2.
   r2 := inA<<6 ;
}
```

Description example 14 shows an example of "right shift" and "left shift" of the shift operation.

The one in which inA is shifted right by five bits is transferred to r1.

The one in which inA is shifted left by six bits is transferred to r2.

In the shift operation, the bit width is the same as before the shift even if it is shifted right and left.

The bit that ran off to the right side at a right shift is discarded, and the empty left side is filled up with the value of "0".

Next, an example of the bit coupling is shown. The bit coupling is an operation that can couple individual signals.

Description example 15 is shown as follows.

Description example 15. Description example of bit coupling

```
declare test_sig {
   input inA[4] ;
   input inB[4] ;
}
module test_sig {
   reg r1[8] ;

   // InA and inB is connected and it transfers to r1.
   r1 := { inA, inB } ;
}
```

Description example 15 shows a description example of the bit coupling

The eight-bit signal that couples inA with inB is transferred to r1.

As for the bit coupling, it can couple not only two signal lines but also multiple signals like the example.

The bit width of the destination signal and the signal after coupling should be the same.

Multiple signals can also be transferred collectively by performing bit connecting at the left-hand side.

In that case, . is described before { } which indicates a connecting. The example is shown in the following Description example 16.

Description example 16. Bit connecting at left-hand side

```
declare test_sig {
   input inA[16];
   output outW[4] ;
   output outX[4] ;
   output outY[4] ;
   output outZ[4] ;
}
module test_sig {
   reg r1[4] ;
   reg r2[3] ;
   reg r3[5] ;
   reg r4[2] ;
. {outW,outX,outY,outZ } = inA ;
   // inA in 16-bit width is divided into outW, outX, outY, and outZ in
4-bit width to be transferred.
   // outW = inA[15:12] ; outX = inA[11:8] ;
   // outY = inA[7:4] ; outZ = inA[3:0] ; is equivalent
. {r1,r2,r3,r4} := 14'b0101_010_11100_11;
   //  A value in 14-bit width is transferred to the registers r1, r2,
r3, and r4.
}
```

Next, it explains a description example of the reduction operation.

An example is shown in the following Description example 17.

Description example 17. Description example of reduction operation

```
declare test_red {
}
module test_red {
   reg r1, r2, r3 ;
   wire w1[4], w2[4] ;

   w1 = 4'b1010 ;
   w2 = 4'b0000 ;

   // The operation result of reduction operation AND of w1 is
   // transferred to r1.
   r1 := &w1 ;  // Reduction AND of w1 is transferred to r1.

   // The operation result of reduction operation OR of w2 is
   // transferred to r2.
   r2 := |w2 ;  // Reduction OR of w2 is transferred to r2.

   // The operation result of reduction operation EX-OR of w3 is
   // transferred to r3.
   r3 := ^w1 ;  // Reduction EX-OR of w3 is transferred to r3.
}
```

Description example 17 shows a description example of the reduction operation.

The reduction operation is an operator to perform the logical operation of every bit digit of the bus.

For example, the reduction operator AND of 1010 in binary becomes 1 & 0 & 1 & 0, and the answer is 1 bit (false).

Next, a description example of the logical operation is shown in Description example 18.

Description example 18. Description example of logical operation

```
declare test_logic {
   input inA[4] ;
   input inB[4] ;
}
module test_logic {
   reg r1, r2, r3 ;

   // True is transferred to r1, when even one 1 exists in inA after
   // logical NOT, and false transferred it in all other cases.
   r1 := !inA ;

   // True is transferred to r2, when even one 1 exists in inA and inB
   // after logical AND, and false transferred it in all other cases.
   r2 := inA && inB ;

   // True is transferred to r3, when even one 1 exists in inA and inB
   // after logical OR, and false transferred it in all other cases.
   r3 := inA || inB ;
}
```

The logical operation has the three kinds of logical NOT, logical AND, and logical OR.

The logical operation is an operator that outputs true when even one 1 exists in an operation result, and outputs false in all other cases.

That is, the operation result of a logical operation becomes either of true or false, 1 or 0 of a bit.

Repeat operation is explained next.

Using repeat operation makes it possible to produce another bit sequence by repeating arbitrary sequence optional number of times.

Description example of repeat operation is shown in the following Description example 19.

Description example 19. Description example of repeat operation

```
declare test_repeat {
   input a[8];
   output rgb[24];
}
module test_repeat {
   reg r1[4];
   reg r2[8];

   rgb = 3{a};
   // Repeating input signal a in 8 bits 3 times to make a bit sequence
in 24 bits, and output it to rgb
   r2 := 2{r1};
   // Repeating register r1 in 4 bits 2 times to make a bit sequence in
8 bits, and output it to r2
}
```

An integer without integer and bit width can be used for the repeat count of repeat operator, and each signal of reg, wire, and variable, and an integer with bit width can be used for the bit sequence to be repeated.

Next, the description example of bit clipping out is presented.

In NSL, optional bit can be read out when a multibit signal exists. Description example 20 is shown.

Description example 20. Description example of bit clipping out

```
declare test_bit_div {
   input inA[8] ;
   input inB[8] ;
}
module test_bit_div {
   reg r1[4], r2[8], r3[14] ;

   //0~3rd digit of inA is transferred to r1.
   r1 := inA[3:0] ;

   // The one that bit-couples 0th bit of inA
   // with 0~6th digit of inB is transferred to r2.
   r2 := { inA[0], inB[6:0] } ;

   // The one that bit-couples 7th bit of inA
   //with 0~4th digit of inB and inA is transferred to r3.
   r3 := { inA[7], inB, inA[4:0] } ;
}
```

Description example 20 is an example of the bit clipping out.

0~3rd digit of inA is transferred to r1.

The one that bit-couples 0th bit of inA with 0~6th digit of inB is transferred to r2.

The one that bit-couples 7th bit of inA with 0~4th digit of inB and inA is transferred to r3.

In this way, it is possible to read it out by clipping out optional bit.

For Verilog HDL or SystemC, if it transfers from a signal in wide bit width to another signal in narrow bit width as it is without using a bit cutout, it is transferred while the high-order bits being rounded down. Although the same description method can be used also in NSL, please use a bit cutout when a bit width is converted since there are problems, such as “becoming an error in VHDL” and “readability falling”.

In addition, the alignment sequence of bits can be reversed by describing a bit designation in [ ] at bit cutout in the order not corresponding to [large value: small value] but [small value: large value]. Only the immediate value can be used for the digit designation in [ ].

Example is shown in Description example 21.

Description example 21. Description example of bit sequence reverse

```
declare bit_field_reverse {
   input a[8];
   output b[8],c[8];
}
module bit_field_reverse {
   b = a[0:7]; // Reversing sequence of all bits
   c = {a[4:7],a[0:3]};
   // Connecting ones of which sequence was reversed by 4 bits each
}

This NSL code is synthesized into the following Verilog-HDL code.

module bit_field_reverse ( p_reset , m_clock , a , b , c );
   input p_reset, m_clock;
   input [7:0] a;
   output [7:0] b;
   output [7:0] c;

   assign b = {{{{{{{a[0],a[1]},a[2]},a[3]},a[4]},a[5]},a[6]},a[7]};
   assign c = {{{{a[4],a[5]},a[6]},a[7]},{{{a[0],a[1]},a[2]},a[3]}};
endmodule
```

Next, a description example of the designation of bit width is shown in the following Description example 22.

Description example 22. Description example of designation of bit width

```
declare test_bit_width_assign {
   input inA[8] ;
   input inB[8] ;
}
module test_bit_width_assign {
   reg r1[16] ;
   reg r2[4] ;

   r1 := 16'(inA) ;
   //  inA is extended to 16 bits, and transferred to r1
   r2 := 4'(inB) ;
   //  inB is reduced to 4 bits, and transferred to r2
}
```

When the bit width of the destination is larger than that of the source (bit extension), it is made a signal in the target bit width filling up the high-order side is with 0.
When the bit width of the destination is smaller than that of the source (bit reduction), it is transferred by cutting out the target bit width from 0th bit.
Please note that a bit width after being changed is designated for both extension and reduction.

For example, 8’(4’b1010) becomes 8’b00001010, and 4’(8’b10100101) does 4’b0101.

However, it is not permitted to write it clipping out optional bit of the register because it decreases in readability.

Next, a description example of bit width extension is shown in the following Description example 23.

Description example 23. Description example of sign extension

```
declare test_bit_ext {
   input inA[8] ;
   input inB[8] ;
}
module test_bit_ext {
   reg r1[16] ;
   reg r2[16] ;

   r1 := 16#(inA) ;
   // inA is extended to 16 bits in bit width with sign, an dtransferred
to r1
   r2 := 16'(inB) ;
   // inB is extended to 16 bits in bit width with no sign, an
dtransferred to r2
}
```

The sign extension is an operator that extends a signal to an optional bit width while assuming the first bit of the signal to be a sign bit, and maintaining the sign.
When the first bit of the signal is 0, 0 is added to the extended bit.
When the first bit of the signal is 1, 1 is added to the extended bit.

For example, when the numerical value of 4'b0101 is sign-extended to 8 bits, it becomes 8'b00000101, and when 4'b1010 is done to 8 bits, it becomes 8'b11111010.

Bit width extension without sign is the same as the bit width designation operator above mentioned, and extends a signal to in arbitrary bit width adding 0 to the head regardless of the first bit of the signal.
When the numerical value of 4’b0101 is extended to 8 bit in bit width with no sign, it becomes 8’b00000101, and when 4’b1010 is extended to 8 bit in bit width with no sign, it becomes 8’b00001010. if mark-less bit width extension is carried out at a bit.
In both cases, please note that a bit width after being extended is designated to extend the bit width.

Conditional operation

The conditional operation is used at the right-hand side of a transfer, and it is an operation to implement a case analysis to the signal and the value to be transferred.

This operation is described as follows using the operators, “if” and “else”.

```
if (Condition) <Signal_and_value> else <Signal_and_value>
```

When the conditional expression described in () just after “if” is true, the signal and the value just after “if” are used for the operation.

When the conditional expression is false, the values just after “else” are used for the operation.

The point to be noted is that “else” is indispensable.

Description example of the conditional operation is shown in the following Description example 24.

Description example 24. Description example of conditional operation

```
declare test_right_if {
   input a[3], b[3] ;
   input trigger ;
   output f[3], g[3] ;
}

module test_right_if {
     //If trigger is true “a” is transferred,
     // and if false “b” is transferred.
   f = if(trigger) a else b ;

     //If trigger is true “a+b” is transferred,
     // and if false “a+1” is transferred.
   g = a + if(trigger) b else 0b001 ;
}
```

Block

The action description indicates the area where the behavior of an atomic action is determined in NSL.

Here, it explains the block to be the basic of NSL description among the action descriptions.

The block is a syntax that defines a starting point and an ending point of the block, and changes the behavior of an atomic action in the block area.

In NSL, a system is configured using this block.

In addition, it explains about the action description of "control terminal", "submodule", "procedure", "state", and "memory", that becomes in the applied edition among the action descriptions in the following Chapters 6~10.

Types of the block are listed in the following Table 5-2 and 5-3. The block that can be used anywhere is listed in Table 5-2, and the one that can be used in a special condition is listed in Table 5-3. The block listed in Table 5-2 is explained in this chapter.

(The block listed in Table 5-3 is explained in Chapter 7.)

Table 5-2. The block that can be used anywhere

parallel operation block	{ }
alt block	alt { }
any block	any { }
if block	if (condition) else

Table 5-3. The block that can be used in a special condition

seq block	seq { } //Only in the function action
while block	while (condition) { } //Only in the "seq" block
for block	for(init variable ; condition; change variable) { } //Only in the "seq" block

Description of parallel operation block

The parallel operation block is a block that operates all the atomic actions in the block in parallel.

Declaration of internal structure elements can be described at the head of a block. The elements declared in the block can be referred to from other blocks after the declaration. However, only the state variable of a state machine cannot be referred to from other blocks.

Parallel operation block is a block which operates all unit actions in a block in parallel. The description method of a parallel operation block is shown below.

```
{
  Declaration of internal structure element
Atomic_action1
  Atomic_action2
  ...
  Atomic_actionX
}
```

The parallel operation block is used when a parallel description is written in a block where it describes the atomic action such as "alt", "any", "if", and "seq", etc.

”alt” block

The "alt" block is an abbreviation of "alternative" block, and a block where the acceptable operation starts.

Since the "alt" block is a conditional branching, relational operations of the operator are used.

The relational operation indicates the relation between the left-hand side and the right-hand side, and the condition is approved when the relation between the left-hand side and the

right-hand side is true.

When the relation between the left-hand side and the right-hand side is false, the condition ends in failure.

Description method of the conditional expression is as follows.

Left-hand_expression Relational_operator Right-hand_expression

"alt", "any", and "if" block judge the conditional expression using this relational operation.

A priority level exists in the "alt" block operation. Only the top operation in order of the description starts even when multiple acceptable operations exist.

In addition, "else" can be described as an action description when all conditions are not acceptable. "else" may be omitted. Description method of the alt block is as follows.

```
alt {
Condition1: atomic_action1   //priority level high
Condition2: atomic_action2
...
ConditionN: atomic_actionN   //priority level low
else      : atomic_actionX
}
```

The following description example of "alt" block is shown in Description example 25.

Description example 25. Description example of "alt" block

```
declare test_alt {
   input in_a[4] ;
   output out_b[4] ;
}
module test_alt {
   reg reg_c[4] ;

   // Common action description begin.
   alt{
      // If the condition is truth, 1111 is transferred to reg_c.
      in_a[3] == 1'b1 : reg_c := 4'b1111 ;
      // If the condition is truth, 1010 is transferred to reg_c.
      in_a[2] == 1'b1 : reg_c := 4'b1010 ;
      // If the condition is truth, 0101 is transferred to reg_c.
      in_a[1] == 1'b1 : reg_c := 4'b0101 ;
      // Parallel action block usage example condition branching ahead.
      in_a[0] == 1'b1 : {
         reg_c := 4'b0001 ;
         out_b =  4'b1111 ;
      }
   }
}
```

In addition, multiple atomic actions can be described after the condition transited by describing a parallel operation block in the alt block. This can be applied also to other conditional syntax.

”any” block

The "any" block is a conditional operation block where the acceptable operation starts.

Differing from the alt block, there is no priority level for the conditional operations in the "any" block, and all the acceptable operations start.

In addition, "else" can be described as an action description when all the conditions are not acceptable. "else" may be omitted.

Description method of the "any" block is as follows.

```
any {
Condition1: atomic_action1
Condition2: atomic_action2
...
ConditionN: atomic_actionN
```

```
else      : atomic_actionX
}
```

Description example of the "any" block is shown in Description example 26.

Description example 26. Description example of "any" block

```
declare test_any {
   input in_a[4] ;
}
module test_any {
   reg r1[4], r2[4], r3[4], r4[4], r5[4] ;

   // Common action description begin.
   any{
        // If the condition is truth, 1111 is transferred to r1.
     in_a[3] == 1'b1 : r1 := 4'b1111 ;
        // If the condition is truth, 1010 is transferred to r2.
     in_a[2] == 1'b1 : r2 := 4'b1010 ;
        // If the condition is truth, 0101 is transferred to r3.
     in_a[1] == 1'b1 : r3 := 4'b0101 ;
        // If the condition is truth, 0001 is transferred to r4.
     in_a[0] == 1'b1 : r4 := 4'b0001 ;
        // If all conditions are false, 0000 is transferred to r5.
     else                : r5 := 4'b0000 ;
   }
}
```

Describing in this way, it can achieve a conditional judgment circuit that uses the "any" block.

”if” block

The "if" syntax exists in a special model of the "any" block.

As for the "if" syntax, when it is the same operation as the "any" syntax with only one condition, that is when the condition is true, the operation indicated with an action description starts.

In addition, "else" is described as an action description when the condition is not acceptable. "else" may be omitted.

Description method of the "if" block is as follows.

```
if (condition) atomic_action1
else           atomic_action2
```

Description example 27. Description example of "if "block

```
declare test_if {
   input in_a[4] ;
}
module test_if {
   reg r1[4], r2[4], r3[4], r4[4], r5[4] ;

   if(in_a[3] == 1'b1)   r1 := 4'b1111 ;
   if(in_a[2] == 1'b1)   r2 := 4'b1010 ;
   if(in_a[1] == 1'b1)   r3 := 4'b0101 ;
   if(in_a[0] == 1'b1)   r4 := 4'b0001 ;
   else                  r5 := 4'b0000 ;
}
```

Describing in this way, it can use the "if" block.

In addition, Description example 26 and Description example 27 indicates an equivalent circuit.

Chapter 6

6. Construct "generate”

In a synthesis from NSL to RTL, a processing by preprocessor called structural expansion is performed between a directive expansion by preprocessor (Refer to Appendix 1) and a synthesis of circuit description by a structure element.

Structural expansion is expanded in order of the described regardless of the element of circuit description.

Using a structural expansion makes it possible to reduce the amount of description when producing the similar, multiple circuits significantly.

Structure syntax is prepared in order to realize a structural expansion.

Structure syntax generate

Structure syntax generate is a syntax which differs from for in seq block, and becomes an action at the same clock performing a structural expansion of the inside of a generate syntax when compiling to a lower language. The structure syntax generate is described as follows.

```
Initial value of loop variable; loop conditional formula; change
value of loop variable
```

Only integer variable of integer, which is described later can be used for loop variable.

The structure syntax “generate” is expanded in following procedure.

1. Initial value is set to loop variable
2. Conditional formula is judged, and it goes to 3 in case of true, and ends for false
3. Structural expansion of behavioral description
4. Change value is updated, and goes to 2

Example of structure syntax of generate is shown in the following Description example 28.

An example of the construct "generate" is shown in the following Description example 28.

Description example 28. Structure syntax generate

```
declare x {
   output f[8];
}
module x {
   integer i;
   variable v[8];

generate(i=0;i<10;i++) {
   v=v+i;
}
   f = v;
}
```

```
generate(i=0;i<10;i++){
   v=v+i;
}
↓
v1 = v0 + 0 ;
v2 = v1 + 1 ;
v3 = v2 + 2 ;
v4 = v3 + 3 ;
v5 = v4 + 4 ;
v6 = v5 + 5 ;
v7 = v6 + 6 ;
v8 = v7 + 7 ;
v9 = v8 + 8 ;
v = v9 + 9 ;
```

In a structure syntax, the inside of a generate syntax is expanded as 1-clock action.
That is, Example 28 shows that it is expanded after structural expansion like the right list, and 45 (8'b0100_0101) is transferred to v finally.
Using this structure syntax makes it easy to expand a barrel shifter, and a multiplication, etc.

Structure syntax if

Structure syntax if can be used when wanting to change a circuit to be used according to the state of an integer variable in generate, etc.
Structural expansion of if block of which condition consists only of integer variable is performed as a structure syntax.

Structure syntax if is described as follows.

```
if (Integer variable condition) unit action 1
else unit action 2
```

When the integer variable condition is true, unit action 1 is synthesized, and unit action 2 is synthesized for false.
Example of structure syntax if is shown in Description example 29.

Description example 29. Description example of structure syntax if

```
// Randam Generator
declare glfsr {
   func_in next_rand;
   output q[16]; // Output of random numbers
}
module glfsr {
   reg r[16] = 0x39a5; // Form of random numbers
   variable v[16];
   integer i;

func next_rand {
   generate (i=0;i<15;i++) {
      if((i == 13) || (i == 12) || (i == 10)) {
         v[i] = r[i+1] ^ r[0];
         // This is selected when I is 13, 12, 10
      } else {
         v[i] = r[i+1];
         //  This is selected when i is other than 13, 12, 11
      }
   // Since partial substitution is performed, a variable terminal is
used.
   }
   v[15] = r[0];
   r:=v;
   q = r;
}
}
```

Integer variable Integer

Integer variable integer is declared as follows.

integer Name of integer variable

integer is declared in the declaration portion of an internal structure element. Integer variable integer accepts the input of 32-bit integer with sign.

Temporary terminal variable

Declaration method of a temporary terminal variable is as follows.

```
variable Name of temporary terminal [bit width]
```

variable is declared in the declaration portion of an internal structure element. The bit width of variable may be omitted. When it was omitted, it becomes in 1-bit width. Temporary terminal variable differs from internal terminal, and it can share the same terminal name.
And, variable does not an initialization, and the initial value is set to 0 at its declaration. As for a substitution into a temporary terminal, an integer can be used for the right-hand side when there is no ambiguity in syntax.
Moreover, an integer is permitted for the 2nd term when the number of bits can be decided at the time of compiling with 2-term operation. An integer is permitted to an item.

As a feature of temporary terminal, a partial substitution, which is not permitted for other terminals is possible.
Example of partial substitution is shown in Description example 30.

Description example 30. Description example of partial substitution into temporary terminal

```
declare subrange interface {
   input a[8];
   output f[8];
}

module subrange {
   variable v[8];

   v[3:0] = a[7:4];
   v[7:4] = a[3:0];
f=v;
}
```

Chapter 7

7. Action description of control terminal

Declaration method of the control input terminal and control output terminal is explained in Chapter 2, and that of the control internal terminal was done in Chapter 4.
Action description of the control input terminal, the control output terminal, and control internal terminal is explained in this chapter.

In the language specification of NSL, path of the control and flow of the data are treated separately. That is, path of the control is described besides the flow of the data such as "input", "output", and "inout". The control terminal is a line that describes a path of the control. There are three kinds of the control terminal.
That is, it involves the control input terminal that is a control signal to come in NSL module, the control output signal that is a control signal to go out from NSL module, and the control internal terminal that is a signal to describe the control inside NSL.

Control internal terminal

Since the control internal terminal is a control terminal that describes the control in the module, it can call a function only in the declared module. It describe as follows when a control internal terminal is called while describing an action.

```
Name_of_internal_terminal()
```

The function is activated by same clock when it was called.
In addition, it is possible for a control internal terminal to which a dummy argument was given to have an actual argument when it is called in the module. When an actual argument is given and a function is called, the actual arguments are listed in () after the name of the control internal terminal.

```
Name_of_control_internal_terminal(actual_argument, actual_argu-
ment, actual_argument, ...)
```

Function of the control internal terminal is described as follows.

```
func Name_of_control_internal_terminal Action_description
```

Action description of the function may be omitted.
In addition, a declared control internal terminal can also be called without function. The control internal terminal called at this time becomes 1 at the same clock when called, and becomes 0 with the next clock. Moreover, it is always 0 when it is not called.
The following Description example 31 shows a description example of the control internal terminal.

Description example 31. Description example of control internal terminal

```
declare func_test{
   input a[4] ;
   input b[4] ;
   output f[4] ;
}
module func_test{
   func_self func_do ;   // Control input terminal is declared

   // Common action description begin
   func_do() ;           // Call control input terminal

   // Control input terminal description
   func func_do {
      f = a | b ;
   }
}
```

In this way, the function is activated only after the declaration of control internal terminal, the function description, and the function call were arranged.

Control input terminal

The control input terminal is a signal of the control terminal that comes in from the outside of a module.

The control terminal coming in from the outside of a module is called a control input signal, and it can initiate a function in the module from the outside of the module. When a function is generated, it must be named as the same name of the declared control input terminal.

Description method of the function is as follows.

```
func Name_of_control_input_terminal Action_description
```

The function may be omitted.

In addition, differing from the case of the control internal terminal, since the control input terminal waits for an input of the control terminal from the outside of the module, it cannot be called from the same module where the control input terminal was declared.

And, the control input terminal called from the outside becomes 1 at the same clock when called, and becomes 0 with the next clock. Moreover, it is always 0 when it is not called.

Description example of the control input terminal is listed based on this description method and the declaration method in Chapter 2.

Description example 32. Description example of control input terminal

```
declare func_in_test{
   input a ;
   input b ;
   output f ;
   func_in func_do ;
}
module func_in_test{

   func func_do f = a | b ;
}
```

Describing as directed in Description example 31, the function "func_do” is activated when the control input terminal func_do was called.

Control output terminal

The control output terminal is a signal of the control terminal to be put outside the module.
This can give the dummy argument to the same way as other control terminals with the control terminal to control an outside module.
This is a control terminal to control the external module, and the dummy argument can be given as well as other control terminals.

When a control output terminal is called with module, it describes as follows.

Name_of_control_output_terminal()

The control output terminal is activated with the same clock when called.
In addition, when a control output terminal to which dummy argument was given is called in a module, actual argument can be given. When it is called by giving an actual argument, it is done as follows.

Name_of_control_output_terminal(*actual_argument*, *actual_argument*, *actual_argument*, ...)

There is no function description since the operation of the control output terminal is outside the module.

Description example of the control output terminal is shown in Description example 33.

Description example 33. Description example of control output terminal

```
declare func_out_test{
   input a ;
   input b ;
   output f ;

   func_out func_do(f) ;
}
module func_out_test{
   if(a & b) func_do(1'b1) ;
}
```

Describing in this way, it becomes possible to output the control output terminal toward the outside.

Here, it is output outside while the dummy argument f, and the actual argument 1 were given to the control output terminal.

Then, it can achieve an operation to output the actual argument 1 outside through the dummy argument f.

“seq” block

Next, it explains the “seq” block that can be used only in a function action.

The “seq” block is an abbreviation of “sequential” block, and in this block, the action description starts with each syntax at every clock in order of the top of being described. Moreover, the first operation is executed with the same clock when it was activated.

The “seq” block is located in the top-level block, and when it wants to call a “procedure” from the “seq” block, the next atomic action in the “seq” block is activated after waiting for the end of the “procedure” that was called.

Moreover, when the “seq” block is activated again while the “seq” block is being activated, both of the “seq” are to be activated at the same time. Therefore, a pipeline can be configured.

Description example of the “seq” block is shown as follows.

```
seq {
Atomic_action 1
Atomic_action 2
Atomic_action 3
...
Atomic_action X
```

```
}
```

Types of the syntax in “seq” block are listed in the following Table 7-1

label	Declare the label in the “seq” block
goto	Move to the label position
while	Conditional loop block
for	Conditional loop block with loop variable

Label , Goto

It can define a label in a “seq” block, and move it to the label position with “goto”.

It is indispensable to declare the label in the “seq” block to be used.

Description method of the label is as follows.

```
  label_name Label_name
```

Definition of the label in a “seq” block is described as follows.

```
  Label name :
```

And, it describes in the same “seq” block as follows when it is moved to the label position.

```
  goto Label_name ;
```

One clock is used for a transition process that uses a “goto” syntax.

Description example of the label is as follows.

```
seq {
  label_name Label_name1, label_name2
  Atomic_action1
  goto Label_name2
Label_name1 :
  Atomic_action2
  Atomic_action3
Label_name2 :
  Atomic_action4
  goto Label_name1
}
```

Description example.34 is shown as a description example of the “seq” block.

Description example.34 Description example of “seq” block

```
declare test_seq {
   input a[4], b[4] ;
   output f[4] ;
   func_in exec_add ;
}
module test_seq {
   reg opr1[4], opr2[4], result[4] ;

   function exec_add seq {
      { //Parallel operation block.  It starts at the first clock.
        opr1 := a ;
        opr2 := b ;
      }
      result := opr1 + opr2 ; // It starts at the second clock.
      f = result ;            // It starts at the third clock.
   }
}
```

Describing in this way, it can achieve a sequential execution circuit that uses the “seq” block.

As for the operation of “seq” block shown in Description example 34, calling “exec_add” starts to execute the first line of the “seq” block first at the first clock. And, it starts to execute the second line at the second clock, and the third line at the third clock.

In accordance with the above action flow, it starts to execute it in order. The execution ends in series when it started to execute the final line of the “seq” block.

In addition, it becomes a pipeline process in case of the module “test_seq” when “exec_add” is called again while the “seq” block is executing it in series by having called “exec_add” once.

An example using the label is shown in the following Description example.35.

Description example.35 Description example of “seq” block:label

```
declare test_label {
   output f[4] ;

   func_in exec_label ;
}
module test_label {
   reg r1[4]=0 ;

   function exec_label seq {
      label_name label1, label2 ; //Labels of label1, label2 are declared.

         goto label2 ;         //It executes from label2 at the next clock.

      label1 :                 //label1 is presented.
           r1++ ;

      label2 :                 //label2 is presented.
         if(r1 == 10) f = r1 ;
         goto label1 ;         // It executes from label1 at the next clock.
   }
}
```

“while” block

The “while” block exists as a syntax only in the “seq” block. The “while” block is used as a conditional loop syntax.

The “while” block ends with no activation if the condition is false before starting the operation.

The “while” block is activated in order of the top of being described if the condition is true before starting the operation.

When all the operations in the “while” block ended, it is confirmed again whether the condition is true, and if it is true, the operation starts from the beginning of the “while” block again, and if it is false, the operation of the “while” block ends.

Parentheses {} of the block is indispensable even for only one action to clarify that it is a sequential action.

Action description example of the “while” block is as follows.

```
while (Condition) {
Atomic_action1
```

```
Atomic_action2
...
Atomic_actionX
}
```

In addition, the “while” block uses one clock each for a conditional judgment and an operation transition.

Please refer to the following Description example 36 and 37 for details of the clock.

First, a description example of the “while” block is shown in the following Description example 36.

Description example 36. Description example of “while” block

```
declare test_while {
   input count_end_sig ;
   func_in exec_count ;
   func_out count_end_call ;
}
module test_while {
   reg cnt[8] = 0 ;

   function exec_count seq {

      //Starting loop of "while" syntax
      while (~count_end_sig) {
         cnt := cnt + 0x01 ;
      }

      count_end_call() ;
   }
}
```

In Description example 36, the function “exec” is called infinitely often as long as it satisfies the condition of the “while” block.

Description example 37 is the one that Description example 36 was rewritten into a form where the clock is easily seen with the “seq” block.

Description example 36 and Description example 37 indicate an equivalent circuit.

Description example 37. Detailed operation of “while” block

```
declare test_while {
   input count_end_sig ;
   func_in exec_count ;
   func_out count_end_call ;
}
module test_while {
   reg cnt[8] = 0 ;

   function exec_count seq {
      label_name label1, label2 ;

      label1 :
         if(count_end_sig) goto label2 ;
         {
            cnt := cnt + 0x01 ;
            goto label1 ;
         }
      label2 :
         count_end_call() ;
   }
}
```

“for” block

The “for” block exists as a syntax limited to the use in the “seq” block.

The “for” block is used as a conditional loop syntax as well as the “while” block.

Action in the block is executed every one clock in series while changing the loop variable like the “for” syntax of C language.

Register (reg) is used for the loop variable.

Please note that it cannot be a sequential execution if other type of signal is used for the loop variable.

Action description example of the “for” block is as follows.

```
for(initial_value; condition; changed_value){
Atomic_action1
Atomic_action2
...
Atomic_actionX
}
```

The “for” block sets an initial value to the loop variable first.

And, the operation in the “for” block is activated when the conditional expression is true.

In addition, the action in the “for” block ends without being activated when the conditional expression is false.

When the “for” block was activated, the atomic action is activated one by one at every clock in order of the top of being described in the block.

When all the atomic actions in the “for” block ended, the conditional expression is compared after updating the value of change.

And, if the conditional expression is true, the operation in the “for” block is activated again, and if false the “for” block ends on the spot.

The “for” block uses one clock each for the conditional judgment and the operation transition.

In addition, the parentheses {} of the block is indispensable even for only one action to clarify that it is a sequential action as well as the “while”.

Description example of the “for” block is shown as follows.

Description example 38. Description example of “for” block

```
declare test_for {
   func_in exec_sum ;
   func_out exec_end_call ;
}
module test_for {
   reg i[4] = 0 ;
   reg sum[8] = 0 ;

   function exec_sum seq {

      // Starting loop with i of “for” syntax as a loop variable.
      for(i:=0 ; i<10 ; i++) {
         sum := sum + { 0x0, i} ;
      }

      //It calls end of “exec” with “func_out”.
      exec_end_call() ;
   }
}
```

Description example 39 is the one that the “for” block in Description example 38 was rewritten into a form where the clock is easily seen using the “seq” block.

Description example 38 and Description example 39 indicate an equivalent circuit.

Description example 39. Detailed operation of “for” block

```
declare test_for {
   func_in exec_sum ;
   func_out exec_end_call ;
}
module test_for {
   reg i[4] = 0 ;
   reg sum[8] = 0 ;

   function exec_sum seq {
      label_name label1, label2 ;

           i := 0x0 ;
      label1 :
        if(i<10) goto label2 ;
        {
           sum := sum + { 0x0, i } ;
           goto label1 ;
        }
      label2 :
        exec_end_call() ;
   }
}
```

Chapter 8

8. Action description of SUBMODULE

Declaration of the submodule is explained in Chapter 4. Action description of the submodule is explained in this chapter.

As previously mentioned, the submodule is a syntax that achieves a hierarchical structure. In NSL, each terminal of the lower module can be operated from a higher module, and data can be passed from a lower module to a higher module.

Submodule declaration specifies a template of the submodule, and describes a name to be made substantial in a higher module of the template. The one that the template was made substantial in the higher module is called "instance".
When it specifies each terminal such as data and control terminal, etc. of the submodule, it is described as follows.

```
Instance_name.Terminal_name
```

Using this description makes it possible to read, and transfer the value of the lower module.

In addition, it is also possible to call the control terminal of the lower module. When a control input terminal of the lower module is called, it describes as follows.

```
instance_name.Name_of_control_input_terminal()
```

When actual argument is given to a control input terminal of the lower module, it is described as follows.

```
instance_name.Name_of_control_input_terminal(<actual_argument>,
<actual_argument>, <actual_argument>, ...)
```

It is delimited with a comma ", " when multiple actual arguments are given.

And, when the output signal is received at the same time as calling the control input terminal, it is described as follows.

```
instance_name.Name_of_control_input_terminal(<actual_argument>).
Name_of_output_terminal
```

Description example 40, a description example of the submodule syntax is shown as follows.

Description example 40. Description example of submodule syntax

```
declare sub_test {
   input inA[16] ;
   input inB[16] ;
   input inC[16] ;
   input inD[16] ;
   output outE[16] ;
   func_in calc1(inA, inB) ;
   func_in calc2(inC, inD) ;
}
module sub_test {
   reg reg1[16] ;

   func calc1 reg1 := inA & inB ;
   func calc2 outE = inC + inD ;
}

declare main_test {
   input in_val1[16] ;
   input in_val2[16] ;
}
module main_test {
   reg result[16] ;
   sub_test SUB ;// Temprate "sub_test" of submodule is instantiated.

   // Common action description
   // In_val1 and in_val2 are passed as an argument and
   // calc1 of sub_test is called.
   SUB.calc1(in_val1, in_val2) ;

   // The value that passes in_val1 and in_val2 as an argument and
   // outputs it from outE of calc2 of sub_test to result.
   result := SUB.calc2(in_val1, in_val2).outE ;
}
```

Describing as directed in Description example 40, it is possible to achieve the submodule syntax.

The submodule syntax in Description example 40 made the "sub_test" module a template, declared it as SUB in the "main_test" module, and assumes it to be an "instance".

In addition, in the action description in "main_test", it calls the function in SUB that is an "instance", makes SUB work, and stores the returned value in the register that is named "result".

Chapter 9

9. Action description of PROCEDURE

Declaration method of the "procedure" is described in Chapter 4. Action description part of the "procedure" is explained in this chapter.

As previously mentioned, the "procedure" is a syntax that provides the control that uses the state transition, the pipeline, and the sequential circuit, and the syntax continues to operate until it transits to other "procedures", or the end is declared when it was once activated.

When the "procedure" is activated, or it transits to another "procedure" in the "procedure", it describes as follows.

The "procedure" actually starts the activation from the next clock that proposed the activation.

```
Name_of_procedure()
```

And, it describes as follows when an operation in the "procedure" is described.

```
proc Name_of_procedure {
Atomic_action1 ;
Atomic_action2 ;
...
Atomic_actionX ;
}
```

When a "procedure" is ended, it describes in the "procedure" as follows.

```
finish
```

The "procedure" ends at the clock next to the one when "finish" was declared.

A description example of the "procedure" is shown in the following Description example 41.

Description example 41. Description example of "procedure"

```
declare proc_test{
   input a[4] ;
   input b[4] ;
   output f[4] ;

   func_in start ;
}
module proc_test{
   reg r1 = 1'b0, r2 = 1'b0, r3 = 1'b0 ;
   reg result[4] = 4'b0000 ;
   proc_name idle() ;      // Procedure "idle" is declared.
   proc_name calc() ;      // Procedure "calc" is declared.
   proc_name out_data() ;  // Procedure "out_data" is declared.

   // Common action description
   r1 := r2 ;
   r2 := r3 ;
   r3 := 1'b1 ;

   if(^r1 & r2 & r3) idle() ;

   // Procedure "idol" action description
   proc idle {
      if(start) calc() ;
   }

   // Procedure "calc" action description
   proc calc {
      result := a + b ;
      out_data() ;
   }

   // Procedure "out_data" action description
   proc out_data {
      f = result ;
      idle() ;
   }
}
```

Describing as directed in Description example 41, it can achieve the "procedure".

In Description example 41, the "procedure" is used as a state transition.

A pipeline can be achieved by continuously calling the "procedure".

Chapter 10

10. Action description of state

Declaration method of the state is explained in Chapter 4. Action description of the state is explained in this chapter.

The following method is used when internal operation of the state is described.

```
state State_name Operation
```

First, the state declared first is activated immediately after the module was activated.
When the state was once activated, it continues to operate until it moves to another state.
When multiple states were declared, "goto" syntax is used when another state is activated (transits) from the state being activated. Usage of "goto" syntax is shown as follows.

```
goto State_name
```

With this "goto", another state can be activated. When it moved to another state, the original state stops.
In addition, since the state being activated is maintained, when the declaration destination is a "procedure", and the "procedure" being activated transited, and activated again, it starts from the state having been activated until just before not the state declared first.

Moreover, action description for the state can be done only in the place where it was declared. The place where the state can be declared is as follows.

- In parallel operation block
- In procedure

A description example of the state is shown in the following Description example 42.

Description example 42. Description example of state

```
declare state_test {
   input a[4] ;
   input b[4] ;
   output f[4] ;
   func_in start() ;
}
module state_test {
   reg cnt_val [4] = 4'b0000 ;

   state_name idle, count, calc ; // Declareration state.

   state idle {                    // State "idol" action description
      if(start) goto count ;
   }

   state count {                   // State "count" action description
      any{
         cnt_val == 4'b1111 : {
            cnt_val := 4'b0000 ;
            goto calc ;
         }
         else               : {
            cnt_val := cnt_val + 4'b0001 ;
         }
      }
   }

   state calc {                 // State "calc" action description
      f = a + b ;
      goto idle ;
   }
}
```

Describing in this way, the state can be achieved.

The difference from the "procedure" is that it can be used also in the common operation part and the "procedure".

In addition, when it was used in the "procedure", it is different in the part of memorizing the state when the "procedure" ended.

When a state is used in the "procedure", since the state memorizes the state condition when the original "procedure" transited even when it transited to another "procedure" and returned to the original "procedure", it is possible to start from a condition on the way.

Chapter 11

11. Action description of memory

Declaration method of the memory is explained in Chapter 4. Action description of the memory is explained in this chapter.

Transfer of ":=" is used when data is written into the memory, and the transfer of "=" is used just like the register when reading from the memory. With this, it writes in synchronization with the clock, and reads out asynchronously.

As an example, it shows a 256 by 4 bit memory that is possible to read/write.

Description example 43. Description example of memory

```
declare mem_test {
   input in_data[4] ;
   input in_addr[8] ;

   output out_data[4] ;

   func_in write() ;
   func_in read() ;
}
module mem_test {
   mem memory[256][4] = { 4'b1010, 4'b0101, 4'b0000, 4'b1100};

   func write memory[in_addr] := in_data ;    // Write memory
   func read  out_data = memory[in_addr] ;    // Read memory
}
```

When "memory" is described, please check the part where the number of the addresses is presented.

Since the number of addresses is specified by the integral value not the bit width, it is directly linked to the number of the addresses.

Chapter 12

12. Structure

The declaration method of structure was explained in Chapter 4. In this chapter, it explains about the instance declaration of structure, and the description method about structure member in an action description.

After declaring a structure outside a module, an instance declaration of the structure is executed in the module.
Type of the signal is defined at the instance declaration. The types it can specify are reg or wire. When declaring with reg, initial value also can be given.

```
Structure name reg Instance name =< Initial value>
Structure name wire Instance name
```

It is possible to have an instance have multiplicity just like declaration of a submodule by specifying a number to the instance name with [] at the time of declaration.

For example, when

```
someting reg anything[5];
```

was described, 5 instances from anything[0] to anything[4] are created.

Setup of an initial value of instance with multiplicity is described as follows.

```
someting reg anything[5] = {0,2,4};
```

In this case, although the multiplicity is 5, since the initial values are no more than 3, 0 enters the remaining 2 of (anything [3], anything [4]).

It is possible to refer and transfer to instance and each member.

To transfer to instance and member is described as follows:

Instance name := Source (In case of instance being reg)

Instance name = Source (In case of instance being wire)

Instance name. member := Source (In case of instance being reg.)

Instance name. member = Source (In case of instance being wire)

To refer to instance and member, it describes as follows:

Destination := Instance name (In case of destination being reg)

Destination = Instance name (In case of destination being wire)

Destination := Instance name. member (In case of destination being reg)

Destination = Instance name. member (In case of destination being wire)

A description example of structure is shown in Description example 44.

Description example 44. Description example of structure

```
struct strtest {
   test1[3];
   test2[4];
   test3;
};

declare st{
}

module st{
   reg r1[8],r2[3];
   strtest wire mmw ;
   strtest reg mmr ;
   r2 := mmw.test1;
   3b'100 is transferred to mmr := 8'h93;
     // mmr.test1, 4b'1001 is done to mmr.test2, and1b'1 is done to mmr.
test3.
   mmw.test2 = 0xa;
   r1 := mmw;
}
```

Chapter 13

13. Interface

Usually in NSL processing system, the clock signal and the reset signal used in a sequence circuit are concealed on language, and a clock input terminal and a reset input terminal are automatically produced at the time of producing a lower language.

Therefore, usually, NSL module becomes a circuit of single-phase clock. However, it can avoid producing the clock input terminal and the reset input terminal automatically by attaching an interface modification to a declare syntax. So, it can satisfactorily describe even the circuit which needs to use a reset clock signal explicitly when using a multi-phase clock, etc.

Note: Regardless of the existence of interface modification, when a sequence circuit was described in a module, the reset signal of the producing circuit is automatically synthesized as the name called p_reset, and the reset signal is done as the name called m_clock. (The signal name can be changed as an option at the time of synthesizing.)

The description method of interface is as follows.

```
// input/output structure element
// internal structure element
// portion of action description
```

interface syntax is explained in Description example 45.

Description example 45. Description example of interface

```
declare if_test_adder4 interface { // external module
   input m_clock ; // Clock input
   input p_reset ; // Reset input
   input add_a[4] ; // Add value A
   input add_b[4] ; // Add value B
   output result_q[4] ; // Result value Q
}
module if_test_adder4 {
   reg r1[4] = 0 ;
}
r1 := add_a + add_b ;
   result_q = r1 ;
}
declare if_test { // Declaration of main module
   input sysclk ; // Clock input
   input sysrst ; // Reset input
   input add_a[4] ; // Add value A
   input add_b[4] ; // Add value B
   output result_q[4] ; // Result value Q
}
module if_test { // Definition of main module

if_test_adder4 adder4 ;

{
// ******** Input signals ********
adder4.m_clock = sysclk ;
   // sysclk is connected to the external module of m_clock terminal.
     sysclk is connected to the external module of m_clock terminal.
adder4.p_reset = sysrst ;
   //sysrst is connected to the external module of p_reset
adder4.add_a = add_a ;
   // add_a is connected to the external module of add_a
adder4.add_b = add_b ;
   // add_b is connected to the external module of add_b
   // ******** Output signals ********
result_q = adder4.result_q ;
   // result_q terminal is connected to the external module of result_q
terminal.
}
```

Description example 45 is an example of using interface.

In the example, if_test is the top module, and if_test_adder4 is the submodule. Since inter-

face modification is attached to if_test_adder4, it does not automatically produce the reset signal “p_reset” of if_test_addr4, and the clock signal “m_clock”. Therefore, for if_test_adder4, p_reset and m_clock are declared in the declaration of data input terminal.

With this interface, it could clearly specify the reset signal and the clock signal of if_test_adder4 as a direct signal name in the module. In addition, it is possible to directly control the submodule if_test_addr4 by directly passing the rest, and the clock signal called sysclk,sysrst from the top module.

Appendix

Appendix 1. Synthesis directive

”include” directive

In NSL, an external source file can be taken by using "include" directive just like C language.

Description method of “include” directive is as follows.

```
#include "File_path_name"
```

With this "include” directive, it becomes easy to control NSL file in terms of module.

A description example of "include" is shown as follows.

Description example O-1. Description example of "include”

```
When you put the file "Sub_test.nsl" on the same directory (folder) as the
inc_test module.

#include "sub_test.nsl"
//↓When compiling, "Sub_test.nsl" is developed with here.

declare inc_test {
   // Describe I/O structure element
}
module inc_test {
   // Describe internal structure element
   // Describe actions
}
```

”define” “undef” directive

The "define” directive is prepared to give the parameter when a module described in NSL is called as a lower module. (The parameter syntax is used when a parameter is given to the module described in Verilog HDL/VHDL/SystemC.)

The "define” directive is a directive that substitutes the character string and the expression with another character string, etc. as well as C language.

For example, it becomes possible to replace "0'b0" with "ZERO". However, NSL reserved word cannot be replaced with.

The description method is as follows.

```
#define Identifier_string Replaced_expression
```

The character string distinguishes between capital letters and small letters.

The defined character string can be used in the source of NSL. To use the defined charac-

ter string in the identifier such as a module name, etc., the character string is enclosed in %%.

Moreover, it is possible to describe so that a constant is added, and subtracted to the defined character string with +/-.

In addition, the defined character string can be released by using “undef” directive. It is described as follows.

```
#undef Defined_string
```

A description example of "define" directive is shown as follows.

Description example O-2. Description example of "define”

```
#define N 8              // N is defined by 8
declare test_%N% {       // N is used for the identifier.
   // N is used as a width in bits of the data input terminal.
   input test_in[N];
   // N-1 is used as a width in bits of the data output terminal.
   output test_out[N-1];
}

module test_%N% {        // N is used for the identifier.
   // From N-2 to 0bit of the data input terminal is transferred
   // to the data output data terminal.
   test_out = test_in[N-2:0];
}
```

This NSL code is converted as follows.

```
declare test_8 {             // N is replaced for 8
   input test_in[8];         // N is replaced for 8
   output test_out[7];       // N-1 is replaced for 7
}

module test_8 {              // N is replaced for 8
   test_out = test_in[6:0];  // N-2 is replaced for 6
}
```

”ifdef” / “ifndef” / “else” / “endif” directive

In NSL, it can use the directive such as “ifdef” and “endif” the same as C language. The following directives are supported by the standard pre-processor of NSL.

ifdef

ifndef

else

endif

The usage is as follows.

```
#ifdef identifier
```

When the identifier name was defined, it is valid up to "else" or "endif" directive.

```
#ifndef identifier
```

When the identifier name was not defined, it is valid up to "else" or "endif" directive.

```
#else
```

The condition of "ifdef/ifndef" directive was not established, it is valid up to "endif" directive.

```
#endif
```

The effective area of "ifdef/ifndef/else" directive is ended.

In addition, the pre-processor of C language also can be used.

An example of "ifdef/ifndef/else/endif" directive is shown in Description example O-3.

Description example O-3. Description example of "ifdef/ifndef/else/endif"

```
#define DEBUG       // identifier DEBUG is defined

declare test {
  input a[8];
  input b[8];

  #ifdef DEBUG     // If identifier DEBUG is not defined,
    output d[8];   // This line is compiled.
  #else
    output q[8];   // If identifier DEBUG is defined, this line is compiled.
  #endif
}

module test {
  #ifndef DEBUG    // If identifier DEBUG is not defined,
    q = a & b;     // This line is compiled.
  #else
    d = a & b;     // If identifier DEBUG is defined, this line is compiled.
  #endif
}
```

Appendix 2. System task

NSL can use the Verilog-HDL/SystemC compatible system task only for the synthesis into Verilog-HDL and SystemC.

The system task is a syntax to mainly assist debugging, and it is used for the simulation.
The following Table ? shows the system tasks that can be used by NSL.
In case of NSL, the underscore "_" is applied instead of "$" due to the synthesis.

Table O2-1. Corresponding table by language of system task

System task command	Corresponding system task of Verilog-HDL	Corresponding function of SystemC	Meaning
_display	$display	printf()	Message and value are indicated on the command line.
_monitor	$monitor	printf()	Message and value are indicated on the command line only when value of the specified signal changed.
_finish	$finish	sc_stop()	End of simulation
_time	$time sc	_time_stamp()	Variable to indicate a simulation time.
_readmemb	$readmemb	-	Readout of memory file (in binary)
_readmemh	$readmemh	-	Readout of memory file (in hexadecimal)

The corresponding table by language of the system task supported by NSL is shown as follows.

Table O2-2. Corresponding table by language of system task

System task command	Verilog HDL	SystemC	VHDL
_display	Yes	Yes	No
_monitor	Yes	Yes	No
_finish	Yes	Yes	No
_time	Yes	Yes	No
_readmemb	Yes	No	No
_readmemh	Yes	No	No

Yes : Corresponding No : Not corresponding

Usage of the system task is the same as Verilog-HDL. The system task can be used just like Verilog-HDL now by applying the underscore "_" instead of "$".
In addition, since this syntax is provided for the simulation purpose, the system task part is not reflected in the real circuit when a module that includes a system task was logically synthesized.

display and monitor

The notation system of "_display" is shown as follows.

```
_display("<Format_specifier_character_string>", <Signal_name>, <
Signal_name>, ... )
```

"display" is a system task that outputs the value and the message of a specified signal to the standard output.

It automatically outputs the value of the signal, and adds a linefeed when "display" is executed in NSL syntax.

Moreover, the syntax of "_ monitor" is shown as follows.

```
_monitor("<Format_specifier_character_string>", <Signal_name>, <
Signal_name>, ... )
```

The "_monitor" is a system task that outputs the value and the message of a specified signal to the standard output.

The "_monitor" differs from "_display", and outputs the message only when the value of a specified signal changed when it was executed in NSL syntax.

time

The "time" is a system variable that indicates the simulation time. It can be used only for the argument of "display", "monitor", and "finish".

The format specifier of "display" and" monitor" is shown in the following Table O2-3, and the description example that includes the system variable "time" is shown in Description example O2-1.

Table O2-3. Format specifier of system task

%b	It outputs in binary number.
%o	It outputs in octal number.
%d	It outputs in decimal number.
%h	It outputs in hexadecimal number.
%e	It outputs a real number in exponent notation.
%f	It outputs a real number in decimal number.
%c	It outputs character.
%s	It outputs character string.

The example that uses "display", "monitor", and "time" is shown below as a usage example of the system task.

Description example O2-1. Example of system task of "display", "monitor", and "time"

```
declare test_task {
   input a[4], b[4] ;
   output f[4] ;
}
module test_task {
   reg trigger[4] = 0 ;
   reg r1[4] = 0 ;
   proc_name proc1, proc2 ;

   trigger := { trigger[3:1], 0b1 } ;
   if(trigger == 0b0111) proc1() ;

   proc proc1 {
      r1 := r1 + 0x1;
      if(r1 > 10) proc2() ;

      _display("a = % d, b = % d", a, b) ;
      _monitor("r1 = % d, T = % t", r1, _time) ;
   }
   proc proc2 {
      f = r1 ;
      finish() ;
   }
}
```

finish

The system task "finish" is a command that ends a simulation.

The notation system of "finish" is as follows.

_finish("<*Character_string*>")

When "finish" is executed in NSL syntax, the character string is output to the standard output, and the simulation ends.

A description example of the system task "finish" is presented as follows.

Description example O2-2. Example of system task "finish"

```
declare test_finish {
   func_in exec_add ;
}
module test_finish {
   reg sum[8] = 0 ;
   reg cnt[4] = 0 ;
   func exec_add seq {
      for(cnt:=0; cnt<10; cnt++) {
         sum := sum + 0x01 ;
      }
      _finish() ;
   }
}
```

readmemh, readmemb

"readmemb" and "readmemh" are the system task that loads an external file as the initial value of the memory.

Representing a sequence in the external file, it can be used by relating the file to it with this system task.

The external file writes in the sequence in ASCII file text.

It reads in with "readmemb" when the sequence is a binary number, and it reads in with "readmemh" for a hexadecimal number.

The notation system of "readmemb" and "readmemh" is as follows.

```
readmemb("File_name", memory_name)
readmemh("File_name", memory_name)
```

The usage of "readmemb" in NSL syntax is the same as "readmemh".

Next, a description example of the system task "readmemh" is presented.

Description example O2-3. Example of system task "readmemh"

```
declare test_read {
   input in_adr[8], in_data[8] ;
   output outdata[8] ;
   func_in write(in_adr, in_data) ;
   func_in read(in_adr) ;
}
module test_read {
   mem memory[256][8] ;

   func write { memory[in_adr] := in_data ; }
   func read { outdata = memory[in_adr] ; }
   _readmemh("neko.txt", memory);
}
```

Appendix 3. Reserved keyword

A:
alt
any
B:
C:
D:
declare
E:
F:
finish
for
func_in
func_out
func_self
func
G:
generate
goto
H:
I:
inout
input
integer
interface
J:
K:
L:
label
label_name
M:
m_clock
mem
module
N:
O:
output
P:
p_reset
proc_name
proc
Q:
R:
reg
S:
seq
state
state_name
T:
U:
V:
variable
W:
while
wire
X:
Y:
Z: