NSL Tutorial

Written by Naohiko Shimizu

Version 2015-10-15 Issued by IP ARCH, Inc.

Introduction

NSL is a language developed for logical design. When you learn any languages, the most effective way is to learn it while actually operating the sample codes which will operate.
I hope many designers to wake up to the comfort of logical design by NSL.

Contents

In this tutorial, you will learn from the basic function of NSL to a little high-level content while actually operating the sample codes which will operate. I hope you to understand the operation by actually operating the samples which will appear in the tutorial having a processing system ready on hand.

Operation environment

The operation of this tutorial has been confirmed on NSL Tutorial LiveCygwin Version 20151006, which Overtone Corporation distributes. Although this NSL Tutorial LiveCygwin has documented NSLCORE Version 20150929, I expect it to work all right even on the version after this.
To cover all contents of the tutorial, each program below and its operating environment are necessary.

* NSLCORE
* IcarusVerilog
* GTKWave
* SystemC

Since the environment necessary for LiveCygwin is prepared completely, downloading and developing it will drastically save the time and efforts to prepare the individual program.

After starting up LiveCygwin, the sample codes of this manual are included in NSL_Tutorial directory just below home, by which directory you can actually implement the procedures of the compilation and execution introduced in this manual.

Table of contents

Chapter 1 Simple NSL circuit 5
Chapter 2 Terminal and Register (component) 8
Chapter 3 Numerical representation and integer representation 12
3.1 Floating-point number 13
Chapter 4 Conditional execution statement and comparison operation 14
Chapter 5 Transfer and operation, and compound statement 16
Chapter 6 Function (control terminal) and circuit with argument 22
Chapter 7 Order execution, repetition processing circuit by while and for ... 25
Chapter 8 Hierarchical structure 33
Chapter 9 Preprocessor 39
Chapter 10 Label and goto statement 40
Chapter 11 Procedure 45
Chapter 12 State machine and state transition 55
Chapter 13 Integer variable and temp variable, and structure development. 58
Chapter 14 Parameter 62
Chapter 15 Example of floating-point adder 65
Chapter 16 Summary of Tutorial 74

Chapter 1　Simple NSL circuit

The following example is a circuit described by NSL. Since this example uses the syntax dedicated for simulation (an element which begins with _), the part will not be an electronic circuit even if it is compiled. The description is for outputting a simulation message. Please prepare this circuit as a text file named tut0.nsl.

List 1.1:　　　tut0

```
declare tut0 simulation {}

module tut0 {
      _finish("Hello World");
}
```

The circuit of NSL is composed of 1 module or more. The module is composed of a declare statement which describes the input/output specification and a module statement which is a description of the operation body. Either of the description contents is enclosed in {}. The declare statement in this example is attached with a modifier (simulation). The modifier tells a compiler that the module is dedicated for a simulation, and cannot be a target of the logic synthesis.

A module needs a module name. The module name is made the same name for declare statement and module statement. How to use the module being seen from outside is defined by a declare statement. The module in the example doesn't have an input/output to outside, but a declare statement for a module definition cannot be omitted.

This circuit has one execution statement.

```
_finish("Hello World");
```

A semicolon ; is written after an execution statement like a description in C. (Unnecessary to be written after a compound statement like C.) When multiple execution statements are described in a module, those execution statements operate in parallel. To achieve it as hardware, an execution statement of the module starts an operation as soon as the power supply is turned on, and will continue it permanently until the power is turned off. Please note that this point is greatly different from the function of a programming language which operates only when it was called. However, since the execution statement in the module is only one

statement which stops the simulation, the simulation will stop immediately after the execution starts. (Even if the simulation stopped, the hardware will not disappear.)

_finish() is the function which outputs a message to the console, and stops the simulation. Please note that the function group which begins with _ is dedicated for a simulation, and cannot become a real circuit. The function and variable dedicated for a simulation includes the following. These functions and variables include such function and variable which correspond to VerilogHDL, and the argument is passed over just as it is when being changed into VerilogHDL.

Table 1.1 Function/variable dedicated for simulation

_finish(string, arg1, arg2,…)	Simulation finishes after message was output.
_stop(string, arg1, arg2,…)	Simulation is stopped after message was output.
_display(string, arg1, arg2,…)	Message is output.
_monitor(string, arg1,arg2,…)	Message is output when the terminal designated to an argument changed.
_readmemh(file, mem)	Content of a file is read into a memory as hexadecimal number.
_readmemb(file, mem)	Content of a file is read into a memory as decimal number.
_random	Random number is returned as a value. (32 bits)
_time	Simulation time is returned. (64 bits)
_init { }	Sequential execution block which can be used only in a simulation module. [1]
_delay(numerical value)	Specified numerical clock is delayed in _init block.

Note: Differing from VerilogHDL developed for simulation, NSL has no way to describe an internal element of other modules directly as HDL. As a complement to it, all the signals can be output to the waveform file as a result of the simulation.

When a module uses an instance of other modules hierarchically, NSL compiler reads the declare statement, and judges the input/output specification of the module. Although the declare statement and the module statement are described in

[1] _init block executes the operation in sequence after one clock from the start of a simulation. It is possible to describe in a place not inside other blocks of the simulation module.

the same file in the example, regarding the declare statement and the module statement as two separated files, and preparing the declare statements of multiple modules together as a header file are convenient to create a large-scale circuit.

To execute the simulation, this circuit is compiled with NSL CORE. Although an example using VerilogHDL compiler Icarus Verilog is shown here, a simulation by SystemC compiler is also possible performing a synthesis to SystemC.

```
> nsl2vl tut0.nsl -verisim2 -target tut0
```

This operation creates a file of VerilogHDL called as tut0.v. Next, it will be compiled with Icarus Verilog.

```
> iverilog -o tut0.vvp tut0.v
```

Then, execute it.

```
> vvp tut0.vvp

VCD info: dumpfile tut0.vcd opened for output.
Hello World
```

The simulation was finished after outputting the message described in NSL, didn't it? The line starting from VCD info is the message which Verilog compiler is providing.

Synthesis to SystemC is performed by the following command.

```
> nsl2sc tut0.nsl -scsim2 -target tut0 -split
```

Please note the last option, –split. Since SystemC compiler has the constraint condition on the declaration order of a module, NSLCORE synthesizes each module to a separate file, controls the order of include, and satisfies the constraint condition requested by SystemC. At this time, the simulation test bench will be the file name in which _sim is added to a specified module name. (tut0_sim.sc in this case) Although the compilation method of a simulation execution image may change with where to put SystemC environment, it is shown in case of LiveCygwin as an example. Since g++ compiler cannot compile a file with the extension of .sc, please

correct the file name to .cpp.

```
> cp tut0_sim.sc tut0_sim.cpp
> g++ -I/opt/systemc-2.3.1-debug/include -L/opt/systemc-2.3.1-debug/lib-cygwin -o tut0_sim.exe tut0_sim.cpp -lsystemc
```

When a compilation was completed, tut0_sim.exe is made. When this was executed, it is understood that you can get a similar result like Verilog's simulation.

```
> ./tut0_sim.exe

        SystemC 2.3.1-Accellera --- Apr 30 2014 11:54:49
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
```

```
Info: (I703) tracing timescale unit set: 1 ns (tut0.vcd)
Hello World

Info: /OSCI/SystemC: Simulation stopped by user.
```

Chapter 2　Terminal and Register (component)

The following example indicates a logic circuit to be described in NSL. This example is also dedicated for simulation, and it will not be an electronic circuit even if being compiled. Please prepare this circuit as a text file named tut1.nsl.

List 2.1:　　tut1

```
declare tut1 simulation { }
module tut1 {
    reg count[8] = 0;
    count++;
    if(count==100) _display("Hello World");
    if(count==200) _finish("bye");
}
```

In a module statement, it describes the components such as terminal and register, etc. to be used inside, and the circuit to be achieved. In the example, 8-bit register count is defined as a component, and the default value is made 0. In addition, the register is incremented by every clock, and it executes _display statement and _finish statement at the count value of 100 and 200 respectively.

When developing a large-scale circuit, each module of a logic circuit is designed as a synchronous circuit. Then, NSL does not usually write a clock signal explicitly. Because the signals of clock and reset correspond to the infrastructure in other words, water supply and sewerage, and the designer need not be conscious of it. As will be described later, you can also operate the synchronous circuits respectively by the different clock and reset signals using these signals as needed.

The hardware designer names a net (wire) in a circuit as an element for the calculation corresponding to the variable in the programming language. The named net is called a terminal, and the value is referred by describing the terminal name in the operation. In NSL, it uses an internal terminal for the arithmetic processing in a module other than the input and output terminals to connect the inside and the outside of a module.
As for the net signal, the electrical potential is kept only while being driven, however, only the output net of a storage element (register and memory) is treated as a special existence to keep the data because it continues to keep the memory contents throughout the power supply turned on.

The following 3 types are the basic components to be developed to the hardware, which declare in a module in NSL. (The components of instance, integer, variable,

and structure, etc. of other module are explained separately.) All the components are necessary to declare prior to the execution statement. In the declaration, the element name and bit width are specified following the key word which indicates the type of component.

1. Register: Register specifies reg register name [bit width]; and, name and bit width. It is possible to set a default value as = value prior to the semicolon. When the bit width is omitted, it is assumed to be 1 bit. Register is an element to memorize a data. The data memorized in a register keeps the same value until it is changed explicitly.
2. Internal terminal: Internal terminal specifies wire internal terminal name [bit width]; and name and bit width just like Register. When the bit width is omitted, it is assumed to be 1 bit. Internal terminal keeps the data only during the clock cycle in which the transfer was received.
3. Memory: Memory specifies mem memory name [number of words][bit width]; and, name and number of words, bit width. The default value can be set as mem mm[256][8] = { 3, 2, 5 }. Memory is an element which lines up the registers in an array, and reads and writes the value by its number.

A value transfer to an internal terminal is executed within a clock in the condition of the transfer being described, and a value transfer to a register and a memory is executed being synchronized with the rising edge of the next pulse. On the software on the premise of a static main memory area, the main purpose of the type of a variable is to specify the memory area size on a main memory. For hardware differing from software, it is necessary to specify the physical size of circuit, and the function of circuit. So, NSL provides the three circuit functions of register, internal terminal, and memory.

As the feature of hardware, the described circuits operate concurrently. So it has prepared a syntax of conditional execution to perform a circuit operation selectively depending on the condition. The conditional execution includes function, procedure, and state other than the conditional execution statement. These control mechanisms adjust the action of circuit operations, and achieve the necessary functions. Especially for the function, it has a control from the outside to the inside to cooperate with other modules in addition to the control which is completed inside the module.

For a hardware design, a dynamic data change at the moment of a transition of clock is important. The designer always has to grasp at which clock the data of an internal component becomes effective.

NSL has the syntax to support logic simulation in addition to the description to achieve as a circuit. The modification of simulation written in the declare statement

in this example indicates that this module is not a logic synthesis target but is effective only at the time of simulation. In addition, both _display and _finish are a built-in function for simulation support.

This circuit is compiled with NSL CORE to execute the simulation.

```
> nsl2vl tut1.nsl -verisim2 -target tut1
```

This operation creates a VerilogHDL file called tut1.v.

Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut1.vvp tut1.v
```

Then, execute it.

```
> vvp tut1.vvp

VCD info: dumpfile tut1.vcd opened for output.
Hello World
bye
```

What this circuit description outputs is the 2 lines of Hello World and bye.

Since NSL executes concurrently, the order of outputting messages is produced from the circuit to produce the order. In the example, the register called count is incremented by every clock, and switches the statement to be executed by the value. The programming language executes in the order as described, however such procedure is necessary because the hardware is based on concurrent operation. (There also is HDL which has a description for order execution only for a simulation like VerilogHDL.)

Chapter 3 Numerical representation and integer representation

In NSL, a bit width is defined for numerical representation used for operation. This is because the bit width of a numerical value has a big effect on circuit scale in hardware operation. The value of expression, register, and terminal of NSL is treated as unsigned binary. Since in case of performing an operation which will have a trivial bit width for the designer, the description to specify the bit width is redundant, in NSL, an integer can be specified in the part where the bit width can be estimated. The trivial case means a value transfer to a register and a terminal, and dyadic operation between the same bit widths. The notation by integer is assumed to be mainly used for specifying the number of shifts in a shift operation (since a barrel shifter is generated when all except for integer were given, the compiler gives warning).

The numerical representation which specifies a bit count includes the following 7 types.

1. Binary number: The binary number starting from 0b is treated as a numerical value with a notation bit count.
   Example: 0b100
2. Octal number: The octal number starting from 0o is treated as a numerical value in notation digit x 3 bits.
   Example: 0o13
3. Hexadecimal number: The hexadecimal number starting from 0x is treated as a numerical value in notation digit x 4 bits.
   Example: 0x123
4. Binary number: The binary number starting from the numerical value'b is treated as a numerical value with a bit width indicated by a numerical value.
   Example: 4'b1
5. Octal number: The octal number starting from the numerical value'o is treated as a numerical value with a bit width indicated by a numerical value.
   Example: 8'o25
6. Decimal number: The decimal number starting from the numerical value'd is treated as a numerical value with a bit width indicated by a numerical value.
   Example: 8'd20
7. Hexadecimal number: The hexadecimal number starting from the numerical value'h is treated as a numerical value with a bit width indicated by a numerical value.
   Example: 8'h3

An integer has the range of signed 32 bits in NSL. Also in the part where the initial value of register and memory, and bit width are presumable, when a numerical value larger than the number which can be expressed by a 32-bit signed integer is required, the numerical value expression where the bit count is specified explicitly is used not an initialization by integer.

Note: Since the integer is signed to the unsigned value of expression and terminal, please be careful when both of them are used together for calculation.

3.1 Floating-point number

Floating-point number may be used to facilitate the description of a circuit function such as the coefficient calculation of a numerical expression, etc. which are used for signal processing, etc. Although the floating-point number is not a target of the logic synthesis, it is evaluated when compiling, and the value can be forwarded to a terminal, a register and a structure as a constant. In addition, converting it into integer by _int function, the value can be used in an arithmetic expression as an integer constant.

The floating-point number is expressed in the type of the numerical value sequence including decimal point, and the floating-point number E index. The following elementary functions are available for the floating-point operation in addition to the addition, subtraction, multiplication and division.

_real, _int, _acos, _asin, _atan, _atan2, _ceil, _cos, _cosh, _exp, _fabs,
_floor, _fmod, _log, _log10, _pow, _sin, _sinh, _sqrt, _tan, _tanh

The expression and integer cannot coexist. You have to convert it into floating-point number using _real, or use only floating-point number.

When a calculation result is transferred to a circuit, it is transferred as _int (floating-point notation) when the value is treated as an integer value.
When it is transferred as an IEEE754 format bit string, the expression is described in the right side of a transfer. The destination has to be 32-bit or 64-bit wire, reg or structure. In this transfer, the floating-point number is converted into a bit string in IEEE754 floating-point single precision or double precision. It can be used also as the initial value of memory, and register, etc.

```
Example Z = _sin(45./180.);       /* To be transferred as IEEE754 format bit string */
        Y = _int(_sin(45./180.)*127.)+128; /* To transfer integer of calculation
            result */
```

Chapter 4 Conditional execution statement and comparison operation

When the value of the register count reached a certain number, it makes the corresponding action performed in the example. if statement of NSL, like C language judges an operating condition as false when the expression indicated in () is 0, and as true when all except for 0. These operators for comparison are shown. These operators return 1-bit 1 when the condition is true, and return 0 when false.

Table 4.1 Conditional operator

==	True when same value
!=	True when different value
>	True when left expression is larger than right expression
<	True when left expression is smaller than right expression
>=	True when left expression is equal to or larger than right expression
<=	True when left expression is equal to or smaller than right expression

A complicated comparison can be performed using logical operator as well as a comparison operation. The logical operator includes ==, ||, and !. NSL evaluates every value in parallel. Since this point is different from C language, please confirm the influence carefully for the function call description with side effects. In addition, differing from C language, the statement to transfer a value (assignment statement for C language, and transfer statement for NSL) does not have a value, so it cannot include a transfer in the expression. Please use the after-mentioned alt statement when it is necessary to evaluate multiple comparisons setting priorities.

else clause can be applied to if statement as well as C language.

```
if(x > 0 || x<=10)      _display("ok:%d",x);
else                    _finish("bye");
```

You can construct a circuit which has a complicated operating condition by

combining if and else. However, a complicated if statement is confusing, which will be a hotbed of bug. When you think the condition is too complicated, in most of cases, you can create a visible circuit using any and alt, which are the multidirectional branched format NSL has.

if statement is in a special format of the condition execution statement of NSL. The general format is any statement or alt statement. These statements perform a multidirectional branched processing to the multiple conditions in the type of condition: execution statement. These statements can also specify else as condition. any statement evaluates all the conditions at the same time, and executes all the statements of which condition becomes true. As for alt statement, the condition is evaluated from the higher statement, and only the statement which becomes true first is executed. It shows an example of any statement which performs the same action as two if statements in the example. Although it is similar to switch statement of C language, please note that a logical expression is described in the left of the colon :. In addition, differing from C language, when being consistent with the condition only 1 statement in the right of the colon : is executed. When it is wanted to perform multiple actions, the after-mentioned compound statement is used.

```
any {
    count==100: _display("Hello World");
    count==200: _finish("bye");
}
```

When alt statement is described, a priority encoder is produced in the hardware resulting from a compilation according to the operation principle of alt statement. Since the priority encoder is often a bottleneck of the hardware performance, please examine whether alt is a really necessary case carefully when creating a circuit using alt. In addition, a use of else is not also recommended from the same reason on a performance issue.

Chapter 5　Transfer and operation, and compound statement

A value can be transferred to the register and the terminal which are a component of NSL. When a transfer is received, the terminal and the register have a value with the bit width. While the register holds a value across a clock, the terminal has an effective value only during the clock where the transfer has occurred. So, to classify the two transfers, the transfer symbol to the register (:=) and the transfer symbol to the terminal (=) are assumed to be different symbols in NSL. The memory which is the component which stores the price has the same symbol as a register. In addition, the memory which is a component to store the value has the same symbol as the register.

As for a transfer to the register, a change in the value occurs at the beginning of the clock cycle next to the clock cycle in which the transfer was performed. That is, the register achieves it by a leading-edge triggered flip-flop.

An operation can be performed using the value of the register and the terminal. NSL has the arithmetic operation and the logical operation. The operation of NSL all produces the results in the same clock.

The following 3 arithmetic operations are prepared. The division has not prepared it as an operator because there are many achievement methods, and it seldom to use an implementation in which the value is returned in 1 clock.

- \+ Addition between the same bit widths is performed.
  Result will be a value in the same bit width.
  When integer is specified to 2nd term, it is presumed to the bit width of 1st term.
- \- Subtraction between the same bit widths is performed.
  Result will be a value in the same bit width.
  When integer is specified to 2nd term, it is presumed to the bit width of 1st term.
- * Addition is performed. No limitation on the bit width.
  Result will have the bit width of the sum of 1st term and 2nd term.

Logical operation has an operation to operate every corresponding bit, and the other reduce operation to perform an operation in the bit direction of 1 word. The reduce operation puts a prefix operator in front of the value, and the operation result will be 1 bit. The operation every bit performs an operation between the same bit widths, and the bit width as result will not change. Since it is possible to presume the bit width in the operation every bit, an integer can be used for the 2nd

term. The operations other than NOT in the next operations can be used also for reduce operator. The reduce operation cannot be applied to 1-bit signal.

- & AND
- | OR
- ^ XOR
- ~ NOT

An example of the operation is shown.

```
a = 8'h55 & 170;
b = |a;
```

Please carefully note the difference between the calculation and the comparison operation. When describing a complicated expression using reduce operation, please use parentheses appropriately not to be confused.

Other operations include shift, bit concatenation, bit cutout, repeat, bit width change, sign extension, and conditional operation.

- >> Logical right shift. Bit width as result does not change. Please make 2nd term integer.
- << Logical left shift. Bit width as result does not change. Please make 2nd term integer.
- { expression , expression, ... } The value where multiple expressions enclosed with { } are bit-concatenated in alignment sequence is returned.
- Numerical value { expression } The value where the expression was repeated the numerical value times and concatenated is returned.
- Expression[numerical value1: numerical value2] The value where the expression was cut out from the numerical value 1st bit to the numerical value 2nd bit is return.
- Expression[expression x] The value where 1 bit of the expression x bit was cut out from the expression is returned.
- Numerical value '(expression) The value where bit width of the expression was changed into a width which is indicated by a numerical value is returned.
- Numerical value#(expression) The value where the expression was sign-extended to a numerical bit is returned.
- if (expression) expression1 else expression2 When the expression is true, the expression1, and when false, expression2 is returned as the price.

NSL checks the bit count in operation severely. Therefore, the designer may wants to change the bit count to an optional expression or numerical value on the way of the expression. Please change the bit count as follows in that case. (y is made 8 bits of terminal here.) In such case, please change the bit count as follows. (Here, y is assumed to be 8-bit terminal.)

```
y = 8'(12);
```

if operation is used when only a part of an arithmetic expression is changed according to the condition. if operation assumes the expression1 or expression2 to be the value. For example, when the smaller of a and b is transferred to the terminal x, it is described as follows.

```
x = if(a<b) a else b;
```

Additionally, there are ++ and -- operations to a register which is a memory element. ++ increments the value of a register by 1, and -- decrements the value of a register by 1. The reason these operators are limited to the register is that as for the register, since the timing of the value being reflected rises at the next clock, it can operate correctly even if the result using the value of the register is transferred to the register, however when an operation result using the value of a terminal is transferred to the same terminal, a loop circuit which consists of only combination circuits is generated, and the circuit operation is not performed correctly. When this operator is used as an expression, the two ways of prefix and postfix are available.

```
w = r++;
```

In case of the prefix, the value of the original register is assumed as the value of the expression, and the register is added or subtracted afterward.

In this example, the value of register r is transferred to terminal w.

```
y = --q;
```

In case of the prefix, the result of the addition or the subtraction is assumed as the value of the expression, the result is transferred to the register. In this example,

the value of register q-1 is transferred to terminal y.

The operation can be also described independently as an action.

```
s--;
```

In this case, the value of register s is decremented by 1.

Using the parentheses { }, it is possible to make a compound statement which consists of multiple statements. The compound statement includes the following types.

- { } : All the statements in a compound statement are executed at the same time.

```
if(x==0) {
    y=1;
    z=2;
}
```

When x is 0, it transfers 1 to terminal y, and 2 to terminal z respectively at the same time.

- seq { } : It describes a sequence block to execute 1 clock each in sequence. It can describe only as a processing of the function. Each statement constructs a pipeline. That is, when the next start-up was activated before the executed statement is finished, the subsequent processing starts an execution without waiting the precedent processing finished.

```
func foo seq {
    a=1;
    a=2;
    a=3;
}
```

When foo is called, 1, 2, and 3 are transferred to terminal a in sequence.

Please prepare the following example as a text file named tut2.nsl.

List 5.1: tut2

```
declare tut2 simulation { }

module tut2 {
    reg count[5]=0;
    wire x[5],y[5];
    any {
        count < 10 : {
                _display("x=%d,y=%d,count=%d",x,y,count);
                count := x;
                y = x + 1;
                x = count + 1;
                }
        count >= 10:     _finish("End");
    }
}
```

Example tut2 lines the descriptions up in a little bit unkind order. Before reading ahead, please think of what kind of result will come out.

It is hinted that each statement of the compound statement is all executed at the same time. The description order of the statements is not related to the execution state.

This circuit is compiled with NSL CORE to execute the simulation.

```
> nsl2vl tut2.nsl -verisim2 -target tut2
```

This operation produces a file of VerilogHDL called tut2.v.

Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut2.vvp tut2.v
```

Then, execute it.

```
> vvp tut2.vvp

VCD info: dumpfile tut2.vcd opened for output.
x= 1,y= 2,count= 0
x= 1,y= 2,count= 0
x= 2,y= 3,count= 1
x= 3,y= 4,count= 2
x= 4,y= 5,count= 3
x= 5,y= 6,count= 4
x= 6,y= 7,count= 5
x= 7,y= 8,count= 6
x= 8,y= 9,count= 7
x= 9,y=10,count= 8
x=10,y=11,count= 9
End
```

Since it is all executed at the same time in the compound statement, which is not related to the description order, the transfer to x and y, and the transfer to count are caused at the same time. However, the value of the register changes at the rising point of the next clock, so the change occurs 1 clock later than x. 2 of the line of which count is 0 appear, which is because the reset signal has got in this circuit, and count will remain 0 during the reset activated.

It is possible to transfer the values to multiple terminals and registers together by the bit-concatenated operation of the left-hand side .{}.

- { terminal1, terminal2, ... } = value ;
- { register1, register2, ... } := value ;

The right-hand side is assumed to be the value corresponding to the integer value, or the concatenated bit number of the left-hand side.

Chapter 6　Function (control terminal) and circuit with argument 6

NSL has a syntax which calls the hardware function as function and executes it. A function name can be referred as a control terminal in NSL. While hazard ('bx) is given to the normal data signal when it is not being transferred, the control terminal is 1 in the clock where the function was called and is 0 during not being called. So, as a control system signal, it can be also used as the signal which has to be 0 when it is not set. There are 3 types such as the internal function (control internal terminal) to control inside the module, the incoming function (control input terminal) which is controlled from outside of the module, and the departure function (control output terminal) which issues a control outside the module. A dummy argument and a return value can be defined to each function (control terminal). Differing from the software, the dummy argument and the return value will be a terminal of the hardware. So the terminal to become a dummy argument has declared beforehand before a declaration of the function (control terminal). In the declaration of the function (control terminal), the function (control terminal) name and the terminal name to becomes a dummy argument, and the terminal name to return the return value if necessary are shown. (The dummy argument and return value may be omitted.)

In a departure function where the self-module outputs, it does not describe action in the module because the processing is performed in an external module. However, an action of the departure function of the submodule can be described in the parent module.

- Definition of internal function

func_self **function name(argument internal terminal list)：return value terminal；**

- Definition of incoming function

func_in **function name(argument input terminal list)：return value output terminal；**

- Definition of departure function

func_out **function name(argument output terminal list)：return value input terminal；**

An action when the function was called is described in func statement.

```
func function name action
```

The syntax acceptable as an execution statement can be all described in an action of the function. A value of the function is effective only within the clock which called the function.

seq statement, which is a compound statement can be described only as an execution statement of the function, and performs a stepwise execution in sequence. When only a started clock is effective for itself, and the function (control pin) also uses a return value of the function in the case seq statement continues moving when, attention is necessary for the thing from which the value of anything but a start clock can't be received. Even when seq statement is continuing to operate, the function (control terminal) itself is effective only in the clock activated, and please note that the value other than the activated clock cannot be received when a return value of the function is used. When an operation result processed by multiple clocks is used, please store the operation result in a register, and create a function separately which reads out the value. (It is convenient to have created a function together to return the operation completion status.)

```
input a,b;
output ret;
func_in   do_calc(a,b);
func_in   get_value(): ret;
```

The following example is a logic circuit using an internal function. Please prepare this circuit as a text file of the name as tut3.nsl. Please prepare this circuit as a text file named tut3.nsl.

List 6.1:   tut3

```
declare tut3 simulation { }

module tut3 {
    wire value[8], ret[8];
    func_self   start(value) : ret;
    if(_time>=100) _finish("Value = %d",start(1));
    func start {
        return ( value + 3 );
    }
}
```

_time is a built-in variable which has a simulation time as a value. Since the simulation time is counted at both the rise and the fall of the clock, please note that it does not necessarily correspond to the time of a logical circuit which operates at the rise of the clock. (When a matching of _time is performed with the comparison operator ==, there is a possibility not to match.)

This circuit is compiled with NSL CORE to simulate.

```
> nsl2vl tut3.nsl -verisim2 -target tut3
```

This operation creates a file of VerilogHDL called tut3.v.

Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut3.vvp tut3.v
```

Then, execute it.

```
> vvp tut3.vvp

VCD info: dumpfile tut3.vcd opened for output.
Value = 4
```

Chapter 7　Order execution, repetition processing circuit by while and for

A sequence block to perform order execution can be defined to a processing of the function. In the sequence block, the described execution statement is executed in increments of 1 clock in sequence. In the sequence block, while statement and for statement are prepared to describe the processing repeatedly.

A sequence block is described in the type of seq { operation statement string }. An operation statement string is the one where multiple operation statements are lined up. In a sequence block in the function, the first operation statement is executed in the clock which called the sequence block. The subsequent operation statement is executed in increments of 1 clock in sequence basically. The sequential execution of the operation statement can perform the call of procedure (proc), the execution of for and while, and the procedural control by goto, and the execution order is changed at this time.

The following example shows a module where the internal function calls a sequence block, executes it in sequence. Using register count as a counter, when this value has become just 100, the internal function is called, and the counter value is indicated in the sequence block of the internal function.

List 7.1:　　　tut13

```
declare tut13 simulation { }

module tut13 {
    reg count[8] = 0;
    wire value[8];
    func_self   start(value);

    count++;
    if(count==100) start(count);

    func start seq {
        _display("Hello World: value = %d, count = %d", value, count);
        _display("count = %d", count);
        _display("count = %d", count);
        _finish("bye: count = %d", count);
    }
}
```

This circuit is compiled with NSL CORE to simulate it.

```
> nsl2vl tut13.nsl -verisim2 -target tut13
```

This operation creates a file of VerilogHDL called tut13.v.
Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut13.vvp tut13.v
```

Then, execute it.

```
> vvp tut13.vvp
VCD info: dumpfile tut13.vcd opened for output.
Hello World: value = 100, count = 100
count = 101
count = 102
bye: count = 103
```

Please pay attention to the fact that the argument when calling the internal function has become equal to the value of count indicated by the first operation statement indicated in the sequence block, and the value of count has increased one by one in the subsequent operation statements. Thus, the operation can be easily described to execute the processing in increments of 1 clock in sequence.

while executes an execution statement in increments of 1 clock in sequence while the expression in the parentheses is not 0. The completion judgment of loop is performed prior to an execution of the execution statement. So, an escape from loop will be performed after the completion judgment became false (0).

```
        while(count<loop) {
              _display("loop = %d, count = %d", loop, count);
        }
```

An example where while is used for tut4 is shown. The register count and the terminal value to be a dummy argument are defined, and moreover, the value of the dummy argument is specified to the control internal terminal start, which is

defined.

List 7.2: tut4

```
declare tut4 simulation { }
```

```
module tut4 {
    reg count[8] = 0;
    wire value[8];
    func_self   start(value);
    count++;
    if(count==100) start(110);

    func start seq {
        reg loop[8]=0;
        loop:=value;
        while(count<loop) {
            _display("loop = %d, count = %d", loop, count);
        }
        _finish("bye: count = %d", count);
    }
}
```

In this circuit, when the value of count has become 100, it calls the internal function start taking 110 as an actual argument. The value of count is transferred to the dummy argument value, and activates start in the same clock. The processing to be executed when start was activated is described as the executable statement of func. In the example, the sequence block is activated, and the transfer to the register of loop is executed by the start-up clock of start. Next, when the value of count is smaller than loop, it starts _display statement where while statement gives a message output. When the value of count has become equal to or larger than loop, while is finished, and _finish is executed.
The reason the dummy argument of the internal function start is retransferred to the register once is that although the value of the dummy argument which is a terminal is effective only for a clock called, while performs executions in multiple clocks, so the condition of while is necessary to be judged by the value of the register.

This circuit is compiled with NSL CORE to simulate it.

```
> nsl2vl tut4.nsl -verisim2 -target tut4
```

This operation creates a file of VerilogHDL called tut4.v.

Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut4.vvp tut4.v
```

Then, execute it.

```
> vvp tut4.vvp

VCD info: dumpfile tut4.vcd opened for output.
loop = 110, count = 101
loop = 110, count = 102
loop = 110, count = 103
loop = 110, count = 104
loop = 110, count = 105
loop = 110, count = 106
loop = 110, count = 107
loop = 110, count = 108
loop = 110, count = 109
bye: count = 111
```

Parallel processing and procedural processing are operating in parallel in this circuit. It includes the processing which increments count register, the conditional operation by if statement, and the procedural processing by the sequence block of func statement. start control of func statement is started by an execution statement of if statement, and after that, executes the operation of seq block one by one. In the hardware, all the processing usually operates at the same time in parallel, so please note that differing from a programming language, even if start control was called, the original circuit called will not be finished.

A cautious person might have a question that the last count is 111. The reason consists in the implementation method of while. The following step is to synthesize while.

1. The arithmetic to evaluate the conditional expression is synthesized. When the value of the expression is 0, goto statement to proceed on to the next processing of while block is synthesized.
2. The processing in while block is synthesized in sequence. When the first processing is not procedure, goto statement and label, it performs a synthesis to execute an evaluation of the expression and the first statement at the same time.
3. goto statement to return to the evaluation of the expression is synthesized. When the last statement in a block are not procedure, goto statement and label, a synthesis to execute the last statement and goto statement to the evaluation of the expression at the same time is performed.

In example tut4, count < loop will be a conditional expression. When the value of count has become 110, the conditional expression becomes false, goes through while, and proceeds on to the next statement. Since the execution of the next statement (_finish(), here) is started 1 clock later, the value of count has changed to 111. Waiting until the conditional expression becomes false is based on the language specification, and the timing cannot be closed up any more.

When you want to make the description compact using a loop processing, but to close up the timing strictly, please use for statement in enumeration type shown in the following example (tut5).

List 7.3: tut5

```
declare tut5 simulation { }

module tut5 {
    reg count[8] = 0;
    func_self   start();
    count++;
    if(count==100) start();

    func start seq {
        reg loop[8];
        for(loop:=0,9) {
            _display("loop = %d, count = %d", loop, count);
        }
        _finish("bye: count = %d", count);
    }
}
```

As for for statement in enumeration type, the statement in a block is executed by incrementing or decrementing by 1 by a factor of the number of the numerical value which specified the value of the register variable. Since the loop is finished when the variable has just reached the final value, the clock for judgment becomes unnecessary, and the next statement can be executed 1 clock earlier being different from while statement, etc. which finishes a loop after the condition has become false. Only an integer value can be specified to the range of the variable. Supposing the variable range is loop:=9,0, it enumerates in reverse order from 9.

```
> nsl2vl tut5.nsl -verisim2 -target tut5
> iverilog -otut5.vvp tut5.v
> vvp tut5.vvp

VCD info: dumpfile tut5.vcd opened for output.
loop =      0, count = 101
loop =      1, count = 102
loop =      2, count = 103
loop =      3, count = 104
loop =      4, count = 105
loop =      5, count = 106
loop =      6, count = 107
loop =      7, count = 108
```

```
loop =      8, count = 109
loop =      9, count = 110
bye: count = 111
```

The type similar to C is prepared for the designer accustomed to for statement of C language.

for(**expression1**; **expression2**; **expression3**) { execution statement }

The expression 1 is executed before the start of a loop, when the expression 2 is true, the execution statement is executed in sequence, and the expression 3 is executed after that. The expression 1 and the expression 3 can specify multiple

execution statements at the same time by a parallel block. As for the evaluation of the loop, while it escapes from a loop after the conditional expression becomes false like while, it can escape from a loop 1 clock earlier than while in the typical example (tut6) because only when the expression 3 is in case of the type of register ++ or register--, it has synthesized in order to evaluate a condition evaluation in the end of the loop using the value after the expression 3 was changed.

List 7.4: tut6

```
declare tut6 simulation { }

module tut6 {
    reg count[8] = 0;
    wire value[8];
    func_self   start(value);
    count++;
    if(count==100) start(5);

    func start seq {
        reg loop[8]=0,lend[8]=0;
        for({loop:=0;lend:=value;};loop<lend; loop++) {
            _display("loop = %d, count = %d", loop, count);
        }
        _finish("bye: count = %d", count);
    }
}
```

Execute the simulation.

```
> nsl2vl tut6.nsl -verisim2 -target tut6
> iverilog -otut6.vvp tut6.v
> vvp tut6.vvp

VCD info: dumpfile tut6.vcd opened for output.
loop =     0, count = 101
loop =     1, count = 102
loop =     2, count = 103
loop =     3, count = 104
loop =     4, count = 105
bye: count = 106
```

In the example tut6, the loop is controlled using the argument of the internal function start. Since the argument of the internal function is given to the terminal, it is necessary to re-receive it to the register once when being used in a statement in which an operation by multiple clocks is performed like for statement. In this example, in the head of for statement, the value of the dummy argument value is transferred to the register lend as well as the loop variable loop is initialized.

Chapter 8 Hierarchical structure

Govern by dividing. This is the beaten track for developing large-scale hardware. A hierarchical structure is used invariably for circuit development in a practical scale. The advantage of hierarchical structure is an improvement in modular performance. A design in high modular performance is not only high in development efficiency but also improves the maintainability.

A circuit of NSL is called a module, and the module can have other modules as component. The other modules read in a module are called submodule.

When a submodule is used as a component, it is necessary to have declared declare statement beforehand which indicated the input/output relation. Even if there is no module statement itself of a submodule, it is possible to compile with NSL. Of course, all the modules have to be defined when achieving it as a circuit, but NSL compiler uses declare statement as prototype statement for a division compilation.

NSL has another structure as the component by which the structure is abstracted. The structure has to have been defined beforehand using struct statement as well as declare statement.

1. declare statement: It defines the input/output of a module. And the attribute of the module has interface and simulation. The former shows that all the input/output pins of the module are defined with declare including the clock signal, and the latter means that this module is for a logic simulation, and does not develop to a real circuit.

```
declare module name module attribute {
      definition of input/output pin
      definition of incoming function
      definition of departure function
}
```

The definition other than module name may be omitted when there is no counterpart.

2. struct statement: It defines the bit allocation of a structure. Differing from the structure of C language, a type is not specified to the declaration of a structure. It is because the user is to determine to use the bit allocation of a structure for a register or a terminal, and there also are some cases where the same

structure is used for the both. In C language, the type is nothing more than the information of a size which is occupied on the memory, however the register and terminal of NSL are completely different as an entity, so it was decided not to include the type in the definition of a structure. A semicolon is necessary in the back to declare a structure name. Although primarily, there is no need for this semicolon in the present language specification, it is introduced in the language specification for later extension.

```
struct structure name {
      member name1[bit number];
      member name2[bit number];
      .....
} ;
```

When using submodule and structure as a component in the module, it is defined as follows.

1. Submodule: A submodule is described as submodule name instance name[multiplicity] ; and the multiplicity may be omitted.
2. Structure: The structure declares as structure name type structure instance name = default value ;. The type is the register type (reg) or the terminal type (wire). A default value can be set in the register type. The setting of a default value may be omitted.

The example which uses submodule and structure will be rather complicated necessarily. Please read the example carefully. tut7 is an example of an addition in 8 bits using two of the 4-bit full adder module fadd4. It uses a structure to break down 8-bit signal into two 4-bit signals.

List 8.1: tut7

```
declare tut7 simulation {}

declare fadd4 {
        input a[4];
        input b[4];
        input ci;
        output q[4];
        output c;
        output o;
        func_in ex(a,b,ci) : q;
}

struct byte_nibble {
```

```
        hi[4];
        lo[4];
} ;
module tut7 {
        byte_nibble reg count = 0 ;
        fadd4 sm[2];

        count++;
        if(count>60) {
                byte_nibble wire res;
                res.lo=sm[0].ex(count.lo,count.lo,0);
                res.hi=sm[1].ex(count.hi,count.hi,sm[0].c);
                _display("count:%d, res:%d, ovf:%d", count, res,sm[1].o);
        }
        if(count==70) _finish("bye");
}

module fadd4 {
        func ex {
                wire qs[4],qe[2];
                qs = {0b0,3'(a)} + {0b0,3'(b)} + 4'(ci);
                qe = (2'(a[3])+2'(b[3])+2'(qs[3]));
                c = qe[1];
                o = qe[1] ^ qs[3];
                return {qe[0],3'(qs)};
        }
}
```

The structure byte_nibble defines 8-bit structure which consists of two 4-bit data. The tut7 module generates two structure instances, the register type count and the terminal type res. count is initialized to 0, and incremented by 1 every clock. Please note that the addition is performed to whole structure (i.e. 8 bits). When the value of count is larger than 60, an addition is executed using two full adders. The value of count is broken down into hi and lo using the member of the structure, and used for calling a full adder. The carry from low-order 4 bits is put in the carry input of the high-order adder, and the result of the respective adders is transferred to the member of the terminal type structure res. These calculation results and the overflow output of the high-order adder are shown as calculation results. In NSL, a

numerical value is treated as an unsigned binary number, so even if an overflow was generated, the sign is not seen as minus. However, thinking it in the notation of two's complement of 8 bits, it may be understood that it has become a negative value.

For overflow calculation, the 4-bit full adder fadd4 separates low-order 3-bit calculation and MSB calculation. To get the carry, MSB is calculated with 2 bits. In addition, the carry and the exclusive OR carrying from the 3rd bit is assumed to be an overflow.

Then, execute it. First, VerilogHDL for a simulations is made using NSL CORE.

```
> nsl2vl tut7.nsl -verisim2 -target tut7
```

It is compiled with Icarus Verilog, and the simulation is performed.

```
> iverilog -otut7.vvp tut7.v
> vvp tut7.vvp

VCD info: dumpfile tut7.vcd opened for output.
count: 61, res:122, ovf:0
count: 62, res:124, ovf:0
count: 63, res:126, ovf:0
count: 64, res:128, ovf:1
count: 65, res:130, ovf:1
count: 66, res:132, ovf:1
count: 67, res:134, ovf:1
count: 68, res:136, ovf:1
count: 69, res:138, ovf:1
count: 70, res:140, ovf:1
bye
```

It is understood that the numerical value in which the value of count is doubled has returned to res. In the notation of two's complement of 8 bits, it is a positive number up to 127, and any more number will be a negative. So, the result of 128 or more has returned a result different from the original sign, and it will be an overflow. When an operation without sign is performed, overflow is ignored. When count became 70, two _display statements operate at the same time, so the order of the two changes by the simulator.

When you want to refer to a part of the expression with the same bit width as the terminal and the register declared in the structure as a structure member, you can

perform a cutout of the target bit by a cast operation.

For example,

```
struct inst {
   op[8];
   r1[4];
   r2[4];
} ;
```

ther is a 16-bit structure declared as above. When you want to take the value of op member from the 16-bit register opreg using the member name of a structure,

```
     workop = (inst)(opreg).op;
```

as above, you can refer to the member by casting the expression to the structure in the form of (structure name)(expression).member name. However, please note that a cast cannot be used for the left side of a transfer.

Chapter 9 Preprocessor

NSL has prepared the following as the preprocessor directive.

- Macro definition
  #define macro name defined value: Macro is defined. Macro name described in the source code is converted into the defined value. When a macro name is used in an identifier, it is surrounded in %%.
  #undef macro name: Macro definition is cancelled.
- Conditional compilation
  #ifdef macro name: In case of macro defined, it compiles the source code up to after #endif or #else.
  #ifndef macro name: In case of macro undefined, it compiles the source code up to after #endif or #else.
  #if numerical value: In case of all except for 0, it compiles the source code up to after #endif or #else.
  #else : In the opposite condition of #ifdef/#ifndef, it compiles the source code up to after #endif or #else.
  #endif : Stop of a conditional compilation.
- Include
  #include <file name> : It searches the file name from the path of NSL_INCLUDE environment variable, and inserts it as a source code.
  #include "file name" : It searches the file name from the current directory or the path specified at compilation, and inserts it as a source code.

The output of nslpp.exe, which is a preprocessor is compatible with the preprocessor of C language. So, when you want to use a further advanced macro, you can also use gcc, etc. as the frontend of NSL.

Chapter 10 Label and goto statement

In the software world, excluding goto statement will make a program visible. But, goto statement is needed in some cases to adjust the timing which cannot be ironed out only by a structured mechanism such as for and while in the hardware design where a space of even 1 clock is grudged. goto statement of NSL is used for controlling a sequence block. goto statement is used also for the state transition of a state machine to be mentioned later, but both are completely different things.

A label is needed for the destination to which an execution control is switched by goto statement. The label is declared in the head part of a sequence block in label_name statement, and a switching destination is defined in the form of label name:.

The following example will meet a certain condition using goto statement. Here, the statement of else is executed when the result judged by if statement is false, so it can take in the value in the same clock. The circuit in this form can be used for meeting an external block with an unclear output timing, etc.

```
func ex seq {
  label_name l1;
  reg value;
  l1: if(!condition) goto l1;
        else value:=something;
}
```

In a processing of the function ex, the processing is suspended during condition being false, and the register value takes in the value when it became true in this example. Please note that this function cannot use return statement to return the value from the function because it operates with multiple clocks using a sequence block. return statement can be described, however the means for receiving of the side which called is limited because the value returns at the timing different from the clock at which the function was called.

The following example calls a submodule of a division circuit. When the division circuit reported a result by the departure function, the value is indicated.

List 10.1: tut8

```
declare tut8 simulation {}
#define N 10
#define M 8

declare divu_%N%_%M% {
input A[N],B[M];
output Q[N],R[M];
```

```
func_in divu_do(A,B);
func_out divu_done(Q,R);
func_out divu_error;
}

module divu_%N%_%M% {
reg QB[M], QQ[N+M];
wire sub_i1[N+1], sub_i2[N], minus;
func_self sub(sub_i1,sub_i2);

func sub {
wire sub_o[N+1];
  sub_o = {sub_i1} - {0b0,sub_i2};
  minus=sub_o[N];
}

func divu_do   {
     if(B==M'b0) divu_error();
     else seq {
     reg bitcount[M];
     for( {bitcount:=0; QB:=B; QQ:={M'b0,A};} ;
                    bitcount < N ; bitcount++) {
            if(sub(QQ[N+M-1:M-1],(N'(QB)<<(N-M))).minus) {
              QQ := (QQ << 1) ;
            }
            else {
              QQ := {(sub_o << 1)[N:N-M],(QQ[N-2:0]<<1)} | (N+M)'b1;
            }
              }
     divu_done(QQ[N-1:0],QQ[(N+M-1):N]);
     }
     }
}

module tut8 {
     divu_%N%_%M%   divid;
     reg   a[N], b[M];
```

```
	func_self go();
	reg count[16]=0;
	count++;
	if(count == 10) go();
	if(divid.divu_error) _finish("divid error");

	func go seq {
		label_name wait_res;
		{
			a:=N'(_random);
			b:=M'(_random);
		}
		{
			divid.divu_do(a,b);
			_display("start %d/%d",a,b);
		}
		wait_res:
		if(!divid.divu_done) goto wait_res;
		else _display("result = %d : %d",divid.Q, divid.R);
		_finish();
	}
```

Then, execute it. It will make VerilogHDL for the simulation using NSL CORE.

```
> nsl2vl tut8.nsl -verisim2 -target tut8
```

It is compiled with Icarus Verilog to simulate it.

```
> iverilog -o tut8.vvp tut8.v
> vvp tut8.vvp

VCD info: dumpfile tut8.vcd opened for output.
start   911/ 18
result =     50 :   11
```

Without using goto statement, it can perform the processing similar to tut8 using a function statement of a departure function of the submodule. In this example, this

way may be easier to understand than using goto statement because it is structural. Like this example, most of goto statements can be replaced by describing a function appropriately. The designer should consider carefully about any possible method without using goto statement, not making the circuit description poorly-visible.

List 10.2: tut8_altt

```
module tut8_alt {
    divu_%N%_%M%   divid;
    reg   a[N], b[M];
    func_self go();
    reg count[16]=0;
    count++;
    if(count == 10) go();
    if(divid.divu_error) _finish("divid error");

    func go seq {
        {
            a:=N'(_random);
            b:=M'(_random);
        }
        {
            divid.divu_do(a,b);
            _display("start %d/%d",a,b);
        }
    }

    func divid.divu_done seq {
        _display("result = %d : %d",divid.Q, divid.R);
        _finish();
    }
}
```

Chapter 11 Procedure

In many cases, a complicated logic circuit implements a pipeline control and a status control. NSL has the syntax to show these control structures openly. The one is procedure. When being called, the procedure (proc) is started up synchronously with the clock, and calls another procedure, or continues to operate until a finish is declared by itself (execution of finish()). A register type dummy argument can be specified to a declaration of the procedure. The procedure is started up after the value specified at the time of calling the procedure was transferred to the dummy argument.

When it is started up by another procedure (or, itself) in the clock in which the procedure is finished, it will be started up again at the next clock while executing a processing to finish it. With this, a pipeline operation can be described smoothly.

One of the main uses of the procedure is to describe a pipeline stage. In this case, a pipeline register is specified to the dummy argument. A circuit can be structured assuming the operating state of a procedure as the state of a state transition machine. The state can be transitioned by calling another procedure. It is used also for making it perform the processing in bulk separately in the function. When a procedure call is described at the top level (not below other blocks and if statement) of a sequence block, the sequence block suspends the execution until the procedure is finished. In the clock where the procedure is finished, the statement next to the procedure call is executed at the same time of the ending process of the procedure.

A procedure is described as a declaration of component.

```
proc_name procedure name(dummy argument1, dummy argument2,  …) ;
```

Dummy argument may be omitted. (Parenthesis is necessary also in the case.)

When calling a procedure,

```
procedure name(actual argument1, actual argument2,  …) ;
```

it is described as above. When another procedure is called during a procedure, the procedure at the calling side is finished. To finish it explicitly,

```
finish();
```

```
Procedure name.finish();
```

it is described as above. The above form is available only in the procedure, and finishes the described procedure itself. The form below can be described also from outside the procedure, and makes the specified procedure finish.

A sequence block can be utilized also in a procedure. The following example uses a procedure instead of the internal function of the above-mentioned example tut13. As for the sequence block in a procedure, please note that the first action statement is executed when an execution of the procedure was started, not the time when the procedure was called. (In NSL, while an execution of the function is started at the clock at which the function was called, the procedure starts an execution at the clock next to being called.)

The description of the example is shown.

List 11.1: tut14

```
declare tut14 simulation { }

module tut14 {
    reg count[8] = 0;
    reg value[8];
    proc_name   start(value);

    count++;
    if(count==100) start(count);
    if(count==200) _finish("countX = %d", count);

    proc start seq {
        _display("Hello World: value = %d, count1 = %d", value, count);
        _display("count2 = %d", count);
        _display("count3 = %d", count);
      finish;
    }
}
```

This circuit is compiled with NSL CORE to simulate it.

```
> nsl2vl tut14.nsl -verisim2 -target tut14
```

This operation makes a file of VerilogHDL called tut14.v.
Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut14.vvp tut14.v
```

Then, execute it.

```
> vvp tut14.vvp
VCD info: dumpfile tut14.vcd opened for output.
Hello World: value = 100, count1 = 101
count2 = 102
count3 = 103
countX = 200
```

The procedure is started from the clock next to being called, and the argument is passed by the register, so the execution result indicates that the argument becomes the value smaller than the count value by 1 as shown above.

Usually, when another procedure is executed from the procedure, the high-order procedure called is finished and the action is transferred, however, as an exception, when the top-level action statement of a sequence block (not in other blocks) calls a procedure in a sequence block in the procedure, the called procedure is treated as a subroutine, and the following action statement of the sequence block is executed after the subroutine was finished.

Look at the following example.

List 11.2: tut15

```
declare tut15 simulation { }

module tut15 {
      reg count[8] = 0;
      reg value[8];
      proc_name   start(value), subproc();

      count++;
      if(count==100) start(count);
      if(count==200) _finish("countX = %d", count);

      proc start seq {
            _display("count1 = %d", count);
            _display("count2 = %d", count);
          subproc();
            _display("count3 = %d", count);
```

```
          finish;
      }

      proc subproc seq {
            _display("sub:count1 = %d", count);
            _display("sub:count2 = %d", count);
            _display("sub:count3 = %d", count);
          finish;
      }
}
```

The two procedures of start and subproc are defined to this example. start procedure calls subproc from the sequence block.

This circuit is compiled with NSL CORE to simulate it.

```
> nsl2vl tut15.nsl -verisim2 -target tut15
```

With this operation, a file of VerilogHDL called tut15.v is made
Next, it is compiled with Icarus Verilog.

```
> iverilog -o tut15.vvp tut15.v
```

Then, execute it.

```
> vvp tut15.vvp
VCD info: dumpfile tut15.vcd opened for output.
count1 = 101
count2 = 102
sub:count1 = 104
sub:count2 = 105
sub:count3 = 106
count3 = 107
countX = 200
```

After executing up to Count2, subproc is called at the clock of count==103 as a subroutine. Since the call of proc is generated at the clock next to being called, the execution clock of the subroutine will start from count==104. The last execution statement of the subroutine procedure is executed at the same clock as the action statement just after the subroutine called in a sequence block of the call source. Here, since finish of subproc to become a subroutine is executed at the same time of _display of the call source, the value of count3 has become 107.

Next, it shows how to use the procedure without using a sequence.

The following example is a module for a small 8-bit CPU and its simulation on which 5 instructions are mounted.

List11.3: tut9

```
#define ADD 0
#define LD   1
#define ST   2
#define JMP 3
#define JZ   4

declare cpu {
    inout data[8];
    output address[8];
    func_out mread(address) : data;
    func_out mwrite(address,data);
}

module cpu {
    reg count[8]=0, pc[8], op[8], im[8], acc[8]=0;
    proc_name ift(pc), imm(op), exe(im);

    any {
        count <=20: count++;
        count == 10: ift(0);
    }

    proc ift {
        imm(mread(pc++));
    }

    proc imm {
        exe(mread(pc++));
```

```
    }

    proc exe {
        wire nextpc[8];
        any {
            op == ADD: {acc:=acc+im;   _display("ADD %d",im);}
            op == LD:   {acc:=mread(im); _display("LD %d",im);}
            op == ST:   {mwrite(im,acc); _display("ST %d",im);}
        }
        any {
            op == JMP: {nextpc=im; _display("JMP %d",im);}
            (op == JZ) && (acc == 0): {nextpc=im; _display("JZ %d",im);}
            else: nextpc=pc;
        }
        ift(nextpc);
    }
}

declare tut9 simulation {}

module tut9 {
    mem mainmem[256][8] = {ADD, 2, JZ, 10, ST, 32, ADD, -1, JMP, 2, ST, 255};
    cpu tut9cpu;

    func tut9cpu.mread {
        _display("READ: ADDRESS:%x, DATA:%x", tut9cpu.address, mainmem[tut9cpu.address]);
        return mainmem[tut9cpu.address];
    }

    func tut9cpu.mwrite {
        _display("WRITE: ADDRESS:%x, DATA:%x", tut9cpu.address, tut9cpu.data);
        mainmem[tut9cpu.address] := tut9cpu.data;
        if(tut9cpu.address == 255) _finish("SIM STOP");
    }
}
```

The module CPU will be a circuit of CPU, and tut9 will be the simulation module.

The CPU has 8-bit bidirectional data bus and 8-bit address bus, and performs input and output with the memory put outside the CPU by the departure function of read (mread) and write (mwrite) of the memory. The CPU has 3 procedure (ift, imm, exe), and specifies 8-bit dummy argument registers (pc, op, im) respectively. These procedures will be the procedures which will perform instruction read, immediate value read, and instruction execution.

The simulation module tut9 defines the CPU as instance tut9cpu, and has the main memory mainmem in which the instructions were written beforehand as a component. It performs a behavior response as a memory to two departure functions of the CPU.

The CPU begins an instruction execution from the address 0 when the internal counter register count has reached 10.

This circuit is compiled with NSL CORE to simulate it.

```
>nsl2vl tut9.nsl -verisim2 -target tut9
```

After that, it compiles and simulates with Icarus Verilog.

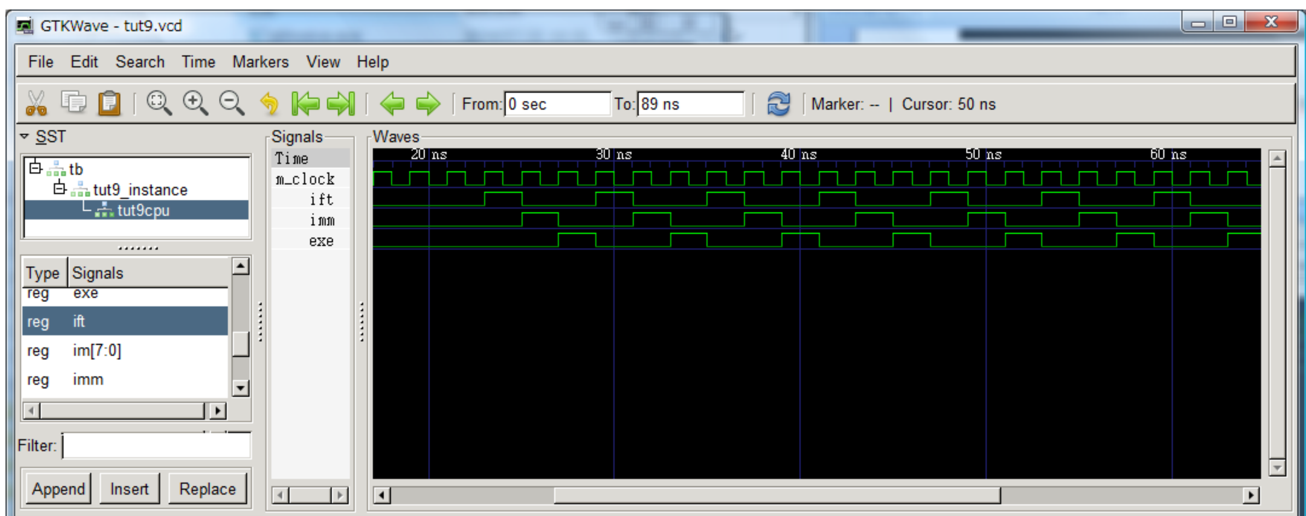
```
> iverilog -otut9.vvp tut9.v
> vvp tut9.vvp
VCD info: dumpfile tut9.vcd opened for output.
READ: ADDRESS:00, DATA:00
READ: ADDRESS:01, DATA:02
ADD     2
READ: ADDRESS:02, DATA:04
READ: ADDRESS:03, DATA:0a
READ: ADDRESS:04, DATA:02
READ: ADDRESS:05, DATA:20
WRITE: ADDRESS:20, DATA:02
ST    32
READ: ADDRESS:06, DATA:00
READ: ADDRESS:07, DATA:ff
ADD 255
READ: ADDRESS:08, DATA:03
READ: ADDRESS:09, DATA:02
JMP     2
READ: ADDRESS:02, DATA:04
READ: ADDRESS:03, DATA:0a
```

```
READ: ADDRESS:04, DATA:02
READ: ADDRESS:05, DATA:20
WRITE: ADDRESS:20, DATA:01
ST    32
READ: ADDRESS:06, DATA:00
READ: ADDRESS:07, DATA:ff
ADD 255
READ: ADDRESS:08, DATA:03
READ: ADDRESS:09, DATA:02
JMP     2
READ: ADDRESS:02, DATA:04
READ: ADDRESS:03, DATA:0a
JZ    10
READ: ADDRESS:0a, DATA:02
READ: ADDRESS:0b, DATA:ff
SIM STOP
WRITE: ADDRESS:ff, DATA:00
ST 255
```

Check the operating status with a waveform viewer.

```
> gtkwave tut9.vcd
```

As shown in the figure, 3 procedures of ift, imm, and exe are started up in sequence, and the instructions are being executed according to the procedure.



Chapter 12　State machine and state transition

In logical design, there may be the cases where it wants to make a circuit with a static state variable, not a procedure which is started when being called. NSL has the syntax to make a state machine corresponding to such case. The state variable is initialized when being reset, and rewritten into the state of the transition destination by a state transition. The state machine can be defined as a component in a module and a compound statement. Please note that the state machine defined by a compound statement will be a local state machine only in the compound statement. The definition of a state machine is given by the following syntax. The leading state will be the initial state of the state machine.

```
state_name state name1, state name2, state name3,  …  ;
```

After the state machine was defined, it describes the action corresponding to each state. The action is described using state statement.

```
state state name execution statement
```

goto statement is used when it is transitioned to other state in state statement. Please note that it is completely different from goto statement in a sequence block.

```
goto state name ;
```

Next, an example of state machine is shown

List 12.1:　　tut10

```
declare tut10 simulation {}

module tut10 {
    state_name state1, state2, state3;
    state state1 {
        reg c1[2]=0;
        if(c1++ == 3) goto state2;
```

```
        _display("in state1 %d",c1);
    }

    state state2 {
        reg c2[2]=0;
        if(c2++ == 3) goto state3;
        _display("in state2 %d",c2);
    }

    state state3 {
        reg c3[2]=0;
        if(c3++ == 3) {goto state1; _finish();}
        _display("in state3 %d",c3);
    }
}
```

Please note that the state variable is static, and a state statement performs an execution when the state variable has become in the state specified by the state variable after the conditions of execution were completed, and. For example, when a state machine was created in a function and a procedure, the execution is not performed even if the state statement was described, unless its function or procedure is operating even if the state variable is identical.

Then, execute it. It creates VerilogHDL for the simulation using NSL CORE.

```
>nsl2vl tut10.nsl -verisim2 -target tut10
```

It compiles and simulates with Icarus Verilog.

```
> iverilog -otut10.vvp tut10.v
> vvp tut10.vvp

VCD info: dumpfile tut10.vcd opened for output.
in state1 0
in state1 0
```

```
in state1 1
in state1 2
in state1 3
in state2 0
in state2 1
in state2 2
in state2 3
in state3 0
in state3 1
in state3 2
in state3 3
```

In the simulation, the value of 0 of state1 has appeared on the 2 lines first because the clock is input also when being reset and a circuit without reset signal (_display statement, etc.) is operating.

There is a point to which you should pay attention on use of a state machine. A state variable is initialized at a reset, but not done during usual operation, so it is necessary to describe goto statement to transition it to the initial state before the procedure finished when you want to return it to the initial state every call of the procedure.

Chapter 13　Integer variable and temp variable, and structure development

Many syntaxes of NSL correspond to the hardware to be generated on a one-to-one basis. This is to offer a visible language to the hardware designer. However, the syntax to develop a structure can be used conveniently for compressing the amount of a description when a number of the similar structures are generated.
NSL provides integer variable, temp variable, and structure development syntax for the structure development. These syntaxes are evaluated when being compiled, and the value is determined. Please note that the evaluation is performed in the order of description of NSL statements, and it will be unrelated to the order of operations when being executed.

An integer variable (integer) can be used for the part where an integer can be described. Also, it can use the arithmetic expression between integer variables, and of integer.

Integer **variable name**;

As a special case, when an expression consisting of integers or integer variables is specified to the condition of if statement, only one of either true or false execution statement is generated when being compiled. It makes a finer conditional control possible than a conditional compilation by the preprocessor, however please note that a decline in readability will be brought when it is used a lot.

A temp variable (variable) has a width, and is used just like a terminal. However, differing from a terminal, multiple transmissions are possible at the clock and a partial transfer can be performed. It is because a temp variable automatically creates a new terminal, and perform the necessary operation every time the transmission appears. The evaluation of the temp variable is performed in the order of description, and the transfer to a terminal after development is executed in parallel.

Variable **variable name[bit width]** ;

generate statement is available for generating a circuit using integer variable and temp variable.

generate (**integer variable=initial value ; integer variable expression ; integer variable update expression) execution statement** ;

The following example generates a pseudorandom number generation circuit by cyclic code using generate syntax. The value to each bit of temp variable v is set using generate statement, and the generated result is transferred to register r together.

List 13.1: tut11

```
declare glfsr {
    input seed[16];
    output q[16];
    func_in set_seed(seed);
    func_in next_rand : q;
}

module glfsr {
    reg r[16] = 0x39a5;
    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];
            else v[i] = r[i+1];
            }
        v[15] = r[0];
        r:=v;
        return r;
    }
    func set_seed r:=seed;
}

declare tut11 simulation {}

module tut11 {
    glfsr rmod;
    reg count[16]=0;
    count++;
    any {
        3'(count) == 7: {
            _display("set seed:%d",count+0x9876);
            rmod.set_seed(count+0x9876);
        }
        count==10:    _finish("finished");
        else:    _display("random generate %d", rmod.next_rand());
    }
}
```

Executing this example will be as follows. It creates VerilogHDL for the simulations using NSL CORE.

```
>nsl2vl tut11.nsl -verisim2 -target tut11
```

It compiles and simulates with Icarus Verilog.

```
> iverilog -otut11.vvp tut11.v
> vvp tut11.vvp

VCD info: dumpfile tut11.vcd opened for output.
random generate 14757
random generate 14757
random generate 43218
random generate 21609
random generate 40500
random generate 20250
random generate 10125
random generate 42950
set seed:39037
random generate 39037
random generate 63550
finished
```

Chapter 14 Parameter

The parameter includes integer parameter (param_int) and character string parameter (param_str). The parameter is used in the two methods in NSL. The first is how to use it as control variable when structure development is performed. In case of using it as a control variable, the value is set to the parameter. In this case, the parameter is defined outside the module, and it is treated as a global variable.

```
param_int parameter name = integer;
param_str parameter name = character string;
```

Comparison operation can be performed in a conditional expression of if statement using a parameter name. The operator identical with the integer variable is used for a comparison of the integer parameter. Match (==) and mismatch (!=) are available for comparison of character string.

```
param_str MODE = "SIM";

module cpu {
 if(MODE == "SIM") {
    _display("PC: %4X", pc);
 }
```

As other use, it includes parameter setting to the module generated by the languages (VerilogHDL and VHDL) other than NSL. When a module in other languages is used as instance, it is necessary to give a specified parameter to those modules in some cases. Therefore, it can declare a parameter in declare statement to specify the prototype of a module in other languages.

```
param_int parameter name ;
param_str parameter name ;
```

In instance declaration of a submodule, setting a value to a parameter passes the parameter to VerilogHDL or VHDL generated.

```
module name submodule instance name(parameter name=value, parameter name=value);
```

An integer parameter with the initial value set, and a character string parameter can be used for a conditional expression of structure development of if statement as integer or character string.

The following description is an example in which a clock wiring is made using the clock module of FPGA (DCM) and the global buffer (BUFG) of Xilinx. There is a parameter which must be given to DCM by the user, and it is set from NSL. This example is an example exclusively for logic synthesis, and some of the modules have only declare, no simulation execution part, so it cannot be confirmed by execution. However, please make it a reference for the description method.

List 14.1: tut12

```
declare DCM interface {
    param_int CLKDV_DIVIDE;
    param_str CLK_FEEDBACK;
    input RST, PSINCDEC, PSEN, PSCLK, CLKIN, CLKFB;
    output PSDONE, CLK0, CLK90, CLK180, CLK270,
            CLK2X, CLK2X180, CLKDV, CLKFX, CLKFX180,
            LOCKED, STATUS[8];
}
declare BUFG interface {
    input I;
    output O;
}

declare sample {
    input samplein;
    output sampleout;
}

declare tut12 {}
module tut12 {
    BUFG buff;
    DCM dcm2(CLKDV_DIVIDE=4, CLK_FEEDBACK="1X");
    sample target;
    dcm2.RST = p_reset;
    dcm2.CLKIN = m_clock;
    dcm2.CLKFB = dcm2.CLK0;
    dcm2.PSEN = 0;
    dcm2.PSCLK = 0;
```

```
    dcm2.PSINCDEC =0;
    buff.I = dcm2.CLKDV;
    target.m_clock = buff.O;
    target.p_reset = p_reset;
}
```

Chapter 15　Example of floating-point adder

It shows an example of the floating point adder assuming IEEE754 floating point form as its input. This adder has a pipeline structure, and can output an operation result every 1 clock.

List 15.1:　　fpadd

```
/*
  IEEE 754 type Single Precision Floating Point Pipeline Adder
  Copyright (c) 2011 Naohiko Shimizu, IP ARCH, Inc.

  This circuit is provided only for demonstration of NSL description.

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.


*/

/*
  Adder interface declaration.
  We will have two inputs 'a' and 'b' those must be normalized
  single precision IEEE754 numbers.
  To start the circuit, you can invoke 'exe' with two arguments.
  When the circuit finish the calculation, it will send the result
  with done signal.
  The circuit is completely pipelined.
  And you can invoke it in every cycle.
*/
```

```
declare Ieee754SpAdd {
        input a[32];
        input b[32];
        output result[32];
        func_in exe(a,b);
        func_out done(result);
}

/*
    The struct Ieee754Sp defines single precision format of IEEE754.
    It has Sign bit, 8bit Exponent, 23bit Mantissa.
*/

struct Ieee754Sp {
        Sign;
        Exponent[8];
        Mantissa[23];
};

/*
   We will use Barrel Shifter for pre-shift the mantissa.
   Though the required length is not 32bit, we will use 32bit shifter.
   Logic synthesizer will do the compaction of unused net.
*/

declare BarrelShift {
        input a[32];
        input sa[8];
        output f[32];
        func_in exe(a,sa):f;
}

/*
   After the mantissa calculation, we need suppress leading zeros
   of the result.
   LeadingZeroShift provide the function.
   It will return the result and shifted amount.
*/
```

```
declare LeadingZeroShift {
        input a[32];
        output shamt[8];
        output f[32];
        func_in exe(a):f;
}

/*
   Barrel Shifter body.
   We will decode each bit of shift amount and
   determine to shift the contents or not.
*/

module BarrelShift {
        wire t0[32],t1[32],t2[32],t3[32],t4[32];
        func exe {
                if(sa[0])         t0=a>>1;
                else              t0=a;
                if(sa[1])         t1=t0>>2;
                else              t1=t0;
                if(sa[2])         t2=t1>>4;
                else              t2=t1;
                if(sa[3])         t3=t2>>8;
                else              t3=t2;
                if(sa[4])         t4=t3>>16;
                else              t4=t3;
                if(sa[7:5]==0)    return t4;
                else              return 0;
        }
}

/*
   Leading Zero Shifter body.
   We will evaluate continuous zero from MSB to LSB.
*/

module LeadingZeroShift {
        wire t0[32],t1[32],t2[32],t3[32],t4[32];
        wire a0,a1,a2,a3,a4;
```

```
        func exe {
                if(a[31:16]==0)                 { a4 = 1; t0 = a<<16; }
                else                        { a4 = 0; t0=a; }
                if(t0[31:24]==0)            { a3 = 1; t1 = t0<<8; }
                else                        { a3 = 0; t1=t0; }
                if(t1[31:28]==0)            { a2 = 1; t2 = t1<<4; }
                else                        { a2 = 0; t2=t1; }
                if(t2[31:30]==0)            { a1 = 1; t3 = t2<<2; }
                else                        { a1 = 0; t3=t2; }
                if(t3[31]==0)               { a0 = 1; t4 = t3<<1; }
                else                        { a0 = 0; t4=t3; }
                shamt = 8'({a4,a3,a2,a1,a0});
                return t4;
        }

}

/*
  Floating Adder body.
  It has 4 stages 'stageA' through 'stageD.'
  At 'stageD' we will get the result.
  exe:     invoke the circuit and calculate exponents differences.
  stageA: pre-shift for adder operation.
  stageB: Mantissa addition.
  stageC: Leading zero shift.
  stageD: Return result.
*/

module Ieee754SpAdd {
/*
  Resources defenitions.
*/
        BarrelShift bshft;
        LeadingZeroShift lzshft;

/*
  stageA resources
*/
```

```
        reg Aexdf[8];
        Ieee754Sp reg x, y;
        proc_name stageA(Aexdf, x, y);

/*
  stageB resources
*/

        reg Bm1[32], Bm2[32], Bs1, Bs2, Bexp[8];
        proc_name stageB(Bm1, Bm2, Bs1, Bs2, Bexp);

        wire s1, s2, x1[32], x2[32], r1[32];
        func_self madd(s1, s2, x1, x2) : r1;

/*
  stageC resources
*/

        reg Cm[32], Cs, Cexp[8];
        proc_name stageC(Cm, Cs, Cexp);

/*
  stageD resources
*/

        Ieee754Sp reg z;
        proc_name stageD(z);

/*
  func exe(a,b) is the starting point of this adder.
*/

        func exe {
                wire wdiff[9];

/*
                In IEEE 754, the exponent is biased binary.
                Therefore, negative value of subtraction will
                show that 'b.Exponent > a.Exponent.'
```

```
		We will select pre-shift argument depending on the result.
*/
		wdiff = 9'((Ieee754Sp)(a).Exponent) - 9'((Ieee754Sp)(b).Exponent);
		if(wdiff[8]) {
			stageA( -wdiff[7:0], b, a );
		}
		else {
			stageA(   wdiff[7:0], a, b );
		}
	}

/*
	proc_name stageA(Aexdf, x, y);

	We will pre-shift 'y' for addition.
	Because IEEE754 suppress MSB's '1' in mantissa, we will add it here.
	But if the exponent part is zero, it may be un-normalized value.
	Therefore, we will not add '1'.
*/
	proc stageA {
		wire xmsb[3],ymsb[3];

		if(x.Exponent==0)
			xmsb=0;
		else
			xmsb=3'b1;

		if(y.Exponent==0)
			ymsb=0;
		else
			ymsb=3'b1;

		stageB( {xmsb,x.Mantissa,6'b0},
			bshft.exe({xmsb,y.Mantissa,6'b0},Aexdf),
			x.Sign,
			y.Sign,
			x.Exponent );
	}
```

```
/*
        proc_name stageB(Bm1, Bm2, Bs1, Bs2, Bexp);

        Add two mantissa values.
        In IEEE754, the mantissa does not have sign it self.
        Therefore, if the addition shows negative value,
        we will make 2's complement of it.
*/

/*
        s1,s2 is sign bit for x1,x2.
        When the value is negative, we will make 2's complement.
*/
        func madd {
                return (32#s1^x1) + 32'(s1) + (32#s2^x2) + 32'(s2);
        }

        proc stageB {
                wire m3[32];

                m3 = madd(Bs1,Bs2,Bm1,Bm2);
                if(m3[31])
                        stageC( -m3, m3[31], Bexp );
                else
                        stageC(   m3, m3[31], Bexp );
        }

/*
        proc_name stageC(Cm, Cs, Cexp);

        We now have mantissa 'Cm', sign 'Cs', exponent 'Cexp.'
        But we need to suppress leading zero of mantissa for normalize.
        If the result of mantissa calculation was 0, we will return 0.
        Or if the exponent was 0, it shows un-normalized value,
        then we will not do the leading zero shift.
*/
        proc stageC {
                if(Cm==0)
                        stageD(0);
```

```
                else if(Cexp==0)
                        stageD({Cs,Cexp,Cm[28:6]});
                else
                        stageD( {Cs,
                                8'(Cexp - lzshft.shamt + 1),
                                lzshft.exe({Cm[30:0], 1'b0})[30:8]
                                } );
        }

/*
        proc_name stageD(z);
        return ack signal and the result from adder.
*/
        proc stageD {
                done(z);
                finish;
        }
}
```

Chapter 16 Summary of Tutorial

I have explained the language specification of NSL with the examples. NSL is compact and visible, and the designer will achieve a new circuit, which is just wanted to be realized including a small CPU to be described in less than 50 lines. In addition, the existing resources of VerilogHDL and VHDL can be applied just as they are to develop the circuits, therefore the investments to date will never be thrown away.

Please design wonderfully using the next generation hardware design language NSL.

NSL Tutorial

Version v1.0.0 Issued on September 13, 2011

Version v1.0.1 Issued on October 15, 2015

Writer	Naohiko Shimizu
Editor	Naohiko Shimizu　Akihiro Sawamura
Publisher	IP ARCH, Inc.
Print	Overtone Corporation

(C) 2011- Naohiko Shimizu