NSL Tutorial: Basic structure of NSL

Naohiko Shimizu

Table of contents

1. Core behavior description

NSL describes the logic circuit as set of modules. The structure of a module is shown as following. Each definitions and/or descriptions can be eliminated but the order of the definitions should not be altered.

declare *module_name* ***{***
    io_facility_defenitions
}
module *module_name* **{**
    internal_facility_defenitions
    core_behavior
    control_behavior
}

Input and output of modules use external terminals:

Direction	Terminal definition command	Description
Input	**Input**	Data input terminal
Output	**Output**	Data output terminal
Bi-directional	**Inout**	Data bi-directional terminal
Input	**func_in**	Control input terminal
Output	**func_out**	Control output terminal

NSL has following facilities:

Facility definition	Description
wire	Internal data terminal
reg	Register
mem	memory
submodule_name	Submodule instance definition
func_self	Control internal terminal
proc_name	Procedure name declaration
state_name	State name declaration
func	Function behavior
proc	Procedure behavior
state	State behavior

NSL has following behavioral definitions:

Operation	Description
Unit operation	Transfer a value to terminal, write to register, write to memory, state transition, activate control terminal, procedure call
{ }	Parallel activation of unit operations.
seq	Sequential activation of unit operations.
for	Loop control. Only valid within seq block
while	Loop control. Only valid within seq block
goto	Execution control. Only valid within seq block
alt	Conditional activation of unit operations with priority.
any	Conditional activation of unit operations without priority. (All of the matched operation will be activated in parallel)
if	Conditional activation of a unit operation.

NSL operators:

Operator	Description	Operator	Description
~	Negate	**|**	OR
signal[*n*]	Bit extraction	**^**	EXOR
signal[*n:m*]	Bits extraction	**&**	AND
|	Bitwise OR	{*signal , signal*}	Bit concatenation
&	Bitwise AND	**+**	Add
^	Bitwise EXOR	**-**	Subtract
n#(*signal_name*)	Sign extension	*n*'(*signal_name*)	unsigned extension
==	Equal	**!=**	Not equal
<=	Less equal	**>=**	Greater equal
<	Less than	**>**	Greater than

NSL provide bit operators and you can make bit twisting logic as Example-NS00.

Example-NS00 Bit extraction and concatenation.

```
declare NS00 {
    input a[8],b[8];
    output f[8];
}
module NS00 {
    f={a[7],(a[2:0]&b[7:5]),b[4:2],a[6]};
}
```

The usage of sign extension operator is shown in Example-NS01.

Example-NS01 Sign extension.

```
declare NS01 {
    input a[4];
    output f[8];
}
module NS01 {
    f=8#(a);
}
```

If you need to adjust bit width without sign extension, then use concatenation as Example-NS02.

Example-NS02 Bit extension without sign extension

```
declare NS02 {
    input a[4];
    output f[8];
}
module NS02 {
    f=8'(a);
}
```

Now we will start the tutorial exercises. Open a terminal window. Then we need to change current directory before starting the tutorials:

```
# cd; cd NSL-1
```

2. Group Signals

NSL can use a set of signals as a group.

Example-NS04

```
declare NS04 {
    input a[8],b[8];
    output f[8];
}
module NS04 {
    f = a + b;
}
```

This example uses IO definitions and core behavior description. All terminals are defined as 8 bit group signals. It operates logic functions between input terminals and then transfers the result to an output terminal. The core behavior description in this example is:

```
f = a + b
```

It is an easy operation and you may image the result. We will make a simulation on this circuitry as following command:

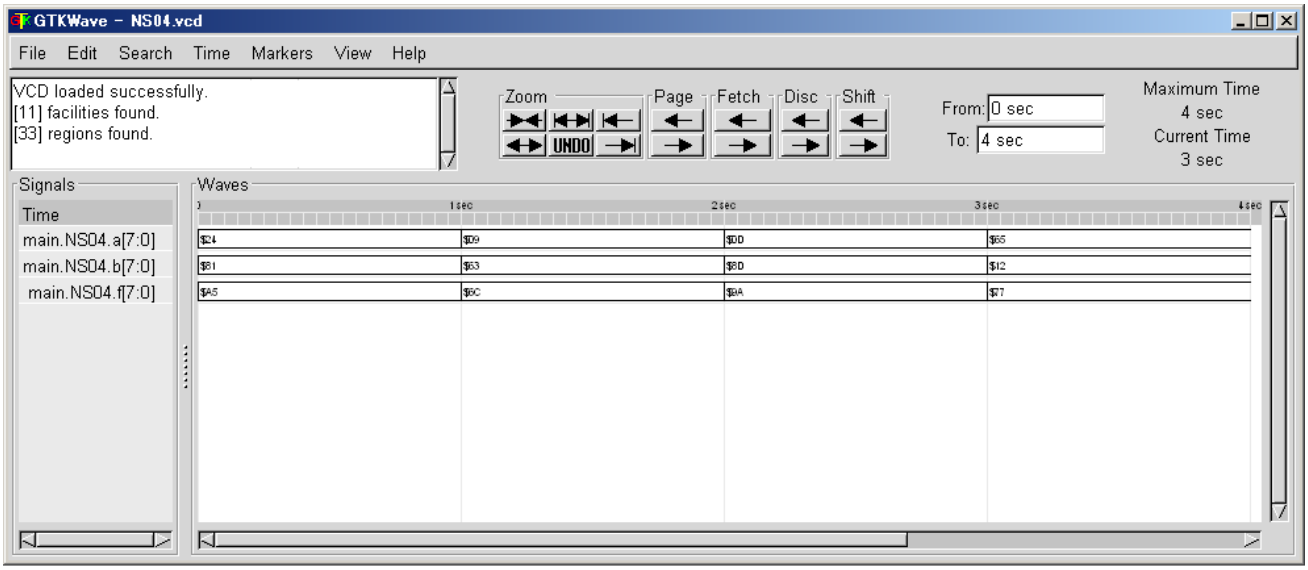
```
# ./exe NS04
```

The result will be available on your console.

```
a: 36, b:129, f:165
a:   9, b: 99, f:108
a: 13, b:141, f:154
a:101, b: 18, f:119
```

You can show the waveform style output with following command.

```
# ./exe -wave NS04
```


GTKWave – NS04.vcd
File Edit Search Time Markers View Help
VCD loaded successfully.
[11] facilities found.
[33] regions found.
Zoom
UNDO
Page
Fetch
Disc
Shift
From: 0 sec
To: 4 sec
Maximum Time
4 sec
Current Time
3 sec
Signals
Time
main.NS04.a[7:0]
main.NS04.b[7:0]
main.NS04.f[7:0]
Waves
1sec
2sec
3sec
4sec
$24
$D9
$DD
$65
$81
$63
$8D
$42
$A5
$6C
$8A
$87

3. Parallel execution

NSL basic action works in parallel.

Example-NS05

```
declare NS05 {
    input a,b;
    output f,g;
}
module NS05 {
    f = a & b;
    g = a | b;
}
```

This example uses IO definitions and core behavior description. Two behavior operation will be activated in the core behavior description.

It is an easy operation and you may image the result. We will make a simulation on this circuitry as following command:

```
# ./exe NS05
```

The result will be available on your console.

```
a:0, b:0, f:0, g:0
a:1, b:0, f:0, g:1
a:0, b:1, f:0, g:1
a:1, b:1, f:1, g:1
```

4. Alt block

NSL uses 'alt' block syntax for conditional behavior activation. The 'alt' bock has priority; higher priority for upper description. The syntax of 'alt' block is:

```
alt {
    condition1: behavior description 1
    condition2: behavior description 2
    .... as may description as you like ....
    else: behavior description for otherwise
}
```

The 'else' condition is optional.

Example-NS06

```
declare NS06 {
    input a,b;
    input c,d;
    output f;
}
module NS06 {
    alt {
      c: f = a & b;
      d: f = a ^ b;
    }
}
```

This example uses IO definitions and core behavior description. Two behavioral operations will be activated conditionally in the core behavior description.

It is an easy operation and you may image the result. We will make a simulation on this circuitry as following command:

```
# ./exe NS06
```

The result will be available on your console.

a:0, b:0, c:0, d:0, f:x
a:1, b:0, c:0, d:0, f:x
a:0, b:1, c:0, d:0, f:x
a:1, b:1, c:0, d:0, f:x
a:0, b:0, c:1, d:0, f:0
a:1, b:0, c:1, d:0, f:0
a:0, b:1, c:1, d:0, f:0
a:1, b:1, c:1, d:0, f:1
a:0, b:0, c:0, d:1, f:0
a:1, b:0, c:0, d:1, f:1
a:0, b:1, c:0, d:1, f:1
a:1, b:1, c:0, d:1, f:0
a:0, b:0, c:1, d:1, f:0
a:1, b:0, c:1, d:1, f:0
a:0, b:1, c:1, d:1, f:0
a:1, b:1, c:1, d:1, f:1

5. Any block

NSL uses 'any' block syntax for conditional behavior activation. The 'any' bock does not have priority. All the behavior which matches the condition will be activated in parallel. The syntax of 'any' block is:

any {
condition1: behavior description 1
condition2: behavior description 2
.... as may description as you like ....
else: *behavior description for otherwise*
}

The 'else' condition is optional.

Example-NS07

```
declare NS07 {
    input a,b;
    input c,d;
    output f;
}
module NS07 {
    any {
            c: f = a & b;
            d: f = a ^ b;
    }
}
```

This example uses IO definitions and core behavior description. Two behavioral operations will be activated conditionally in the core behavior description.

It is an easy operation and you may image the result. We will make a simulation on this circuitry as following command:

 # ./exe NS07

The result will be available on your console.

a:0, b:0, c:0, d:0, f:x
a:1, b:0, c:0, d:0, f:x
a:0, b:1, c:0, d:0, f:x
a:1, b:1, c:0, d:0, f:x
a:0, b:0, c:1, d:0, f:0
a:1, b:0, c:1, d:0, f:0
a:0, b:1, c:1, d:0, f:0
a:1, b:1, c:1, d:0, f:1
a:0, b:0, c:0, d:1, f:0
a:1, b:0, c:0, d:1, f:1
a:0, b:1, c:0, d:1, f:1
a:1, b:1, c:0, d:1, f:0
a:0, b:0, c:1, d:1, f:x
a:1, b:0, c:1, d:1, f:x
a:0, b:1, c:1, d:1, f:x
a:1, b:1, c:1, d:1, f:x

6. Data Internal Terminals

NSL uses data internal terminals for temporary data signals. Data internal terminals are declared with 'wire'. We can refer and transfer value to the terminal as input or output terminals.

Example-NS08

```
declare NS08 {
    input a,b;
    output f;
}
module NS08 {
    wire c,d;
    c = a & ~b;
    d = ~a & b;
    f = c | d;
}
```

We will make a simulation on this circuitry as following command:

```
# ./exe NS08
```

The result will be available on your console.

```
a:0, b:0, c:0, d:0, f:0
a:1, b:0, c:1, d:0, f:1
a:0, b:1, c:0, d:1, f:1
a:1, b:1, c:0, d:0, f:0
```

7. Control Internal Terminals

NSL uses control internal terminals for sending message to a object. Control internal terminals are declared with 'func_self'. We can refer the value of the terminal as input or output terminals. To activate a control internal terminal, use a syntax 'control()' just like a function call of C language. Also control internal terminals can have arguments. A control internal terminal is a control signal of an object and the arguments are the associated data for the object which are required for the control operation. Arguments are declared as:

func_self *control_terminal_name* **(** *argument1, argument2, ...* **) ;**

Even if there is no argument at all, you cannot eliminate the parenthesis. We also declare associated behavior to the control terminal with func syntax.

func *control_terminal_name behavior_description*

Example-NS09

```
declare NS09 {
    input a,b;
    output f;
}
module NS09 {
    wire c;
    func_self do(c);
    any {
            a & ~b: do(a);
            ~a & b: do(b);
    }
    func do {
            f = c;
    }
}
```

We will make a simulation on this circuitry as following command: 15

 # ./exe NS09

The result will be available on your console.

a:0, b:0, do:0, c:x, f:x
a:1, b:0, do:1, c:1, f:1
a:0, b:1, do:1, c:1, f:1
a:1, b:1, do:0, c:x, f:x

8. Control Input Terminals

NSL uses control input terminals to recieve control message from other modules. Control input terminals are declared with 'func_in'. We can refer and invoke associate behavior with the terminal.

Example-NS10

```
declare NS10 {
    input a,b;
    output f;
    func_in exec_and();
    func_in exec_or();
    func_in exec_xor();
}
module NS10 {
    func exec_and {
            f = a & b;
    }
    func exec_or {
            f = a | b;
    }
    func exec_xor {
            f = a ^ b;
    }
}
```

We will make a simulation on this circuitry as following command:

```
# ./exe NS10
```

The result will be available on your console.

```
AND: a:0, b:0, f:0
AND: a:0, b:1, f:0
AND: a:1, b:0, f:0
AND: a:1, b:1, f:1
```

```
OR: a:0, b:0, f:0
OR: a:0, b:1, f:1
OR: a:1, b:0, f:1
OR: a:1, b:1, f:1
XOR: a:0, b:0, f:0
XOR: a:0, b:1, f:1
XOR: a:1, b:0, f:1
XOR: a:1, b:1, f:0
```

A control input terminal is a single bit signal, but associated behavior can operate on any width data as following example.

Example-NS11

```
declare NS11 {
    input a[8],b[8];
    output f[8];
    func_in exec_add(a,b);
}
module NS11 {
    func exec_add {
            f = a + b;
    }
}
```

We will make a simulation on this circuitry as following command:

```
# ./exe NS11
```

The result will be available on your console.

```
a:129, b: 36, f:165
a: 99, b: 9, f:108
a:141, b: 13, f:154
a: 18, b:101, f:119
```

9. Submodules

NSL supports hierarchical logic design. In NSL, we call lower layer modules as submodules. The interface of a submodule is described in 'declare' sentence. It will be consist of external terminals and optional arguments for control terminals. To declare instances of a submodule, we can declare at facility_declaration part as:

submodule_name instance_name ;

Each submodule terminal can be referred or transferred a value. The terminal name is referred as:

instance_name.terminal_name

To invoke a control terminal of the submodule, we can use following syntax:

instance_name.control_input_terminal_name **()** ;

There is a special syntax that invoke a control terminal and refer a output terminal of the submodule.

*instance_name.control_input_termainal_name***(***arguments***)***.output_terminal_name*

You can use this syntax as a reference to a terminal. Now we will see an example that uses NS11.nsl as a submodule. In this example, we use three submodule instances of NS11.

Example-NS12

```
#include "../NS11/NS11.nsl"
declare NS12 {
    input a[8],b[8],c[8],d[8];
    output f[8];
    func_in do(a,b,c,d);
}
module NS12 {
```

```
    NS11 mod1,mod2,mod3;

    func do {
        f = mod1.exec_add(
            mod2.exec_add(a,b).f,
            mod3.exec_add(c,d).f).f;
    }
}
```

We will make a simulation on this circuitry as following command:

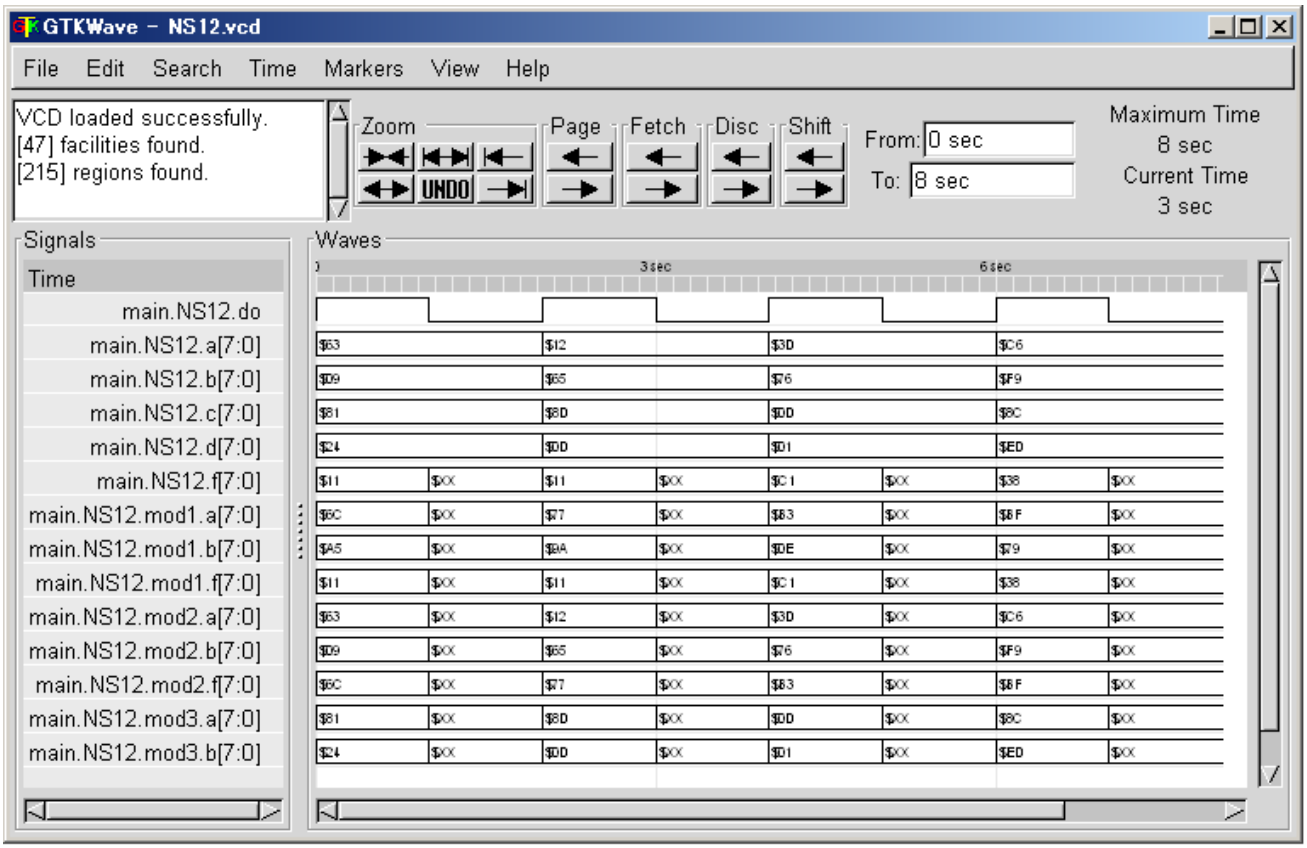
```
# ./exe NS12
```

The result will be available on your console.

```
a: 99, b: 9, c:129, d: 36, f: 17
a: 18, b:101, c:141, d: 13, f: 17
a: 61, b:118, c: 13, d: 1, f:193
a:198, b:249, c:140, d:237, f: 56
```

We will make a wave form on this circuitry as following command:

```
# ./exe -wave NS12
```


GTKWave - NS12.vcd
File Edit Search Time Markers View Help
VCD loaded successfully.
[47] facilities found.
[215] regions found.
Zoom
UNDO
Page
Fetch
Disc
Shift
From: 0 sec
To: 8 sec
Maximum Time
8 sec
Current Time
3 sec
Signals
Time
main.NS12.do
main.NS12.a[7:0]
main.NS12.b[7:0]
main.NS12.c[7:0]
main.NS12.d[7:0]
main.NS12.f[7:0]
main.NS12.mod1.a[7:0]
main.NS12.mod1.b[7:0]
main.NS12.mod1.f[7:0]
main.NS12.mod2.a[7:0]
main.NS12.mod2.b[7:0]
main.NS12.mod2.f[7:0]
main.NS12.mod3.a[7:0]
main.NS12.mod3.b[7:0]
Waves
3 sec
6 sec

10. Control Output Terminals

A module may send message to other and/or upper modules. In this case, NSL uses control output terminals. Control output terminals are declared with 'func_out'. Upper module will use 'func' to describe the associated behavior.

Example-NS13

```
declare sub {
    input a[4],b[4];
    output f[4];
    func_in do(a,b);
    func_out done(f);
}
module sub {
    func do {
    done(a&b);
    }
}
declare NS13 {
    input a[4],b[4];
    output f[4];
    func_in do(a,b);
}
module NS13 {
    sub sub1;
    func do {
            sub1.do(a,b);
    }
    func sub1.done {
            f=sub1.f;
    }
}
```

We will make a simulation on this circuitry as following command:

```
# ./exe NS13
```

The result will be available on your console.

a:0001, b:0100, f:0000
a:0011, b:1001, f:0001
a:1101, b:1101, f:1101
a:0010, b:0101, f:0000

11. Register

NSL uses registers to hold data signals. Registers will hold on the rising edge of a clock signal (m_clock). We can refer and transfer value to registers but referenced value will be the data on last clock period.

Example-NS14

```
declare NS14 {
    input a,b;
    output f;
}
module NS14 {
    reg r;
    r := a & b;
    f = r;
}
```

We will make a simulation on this circuitry as following command:

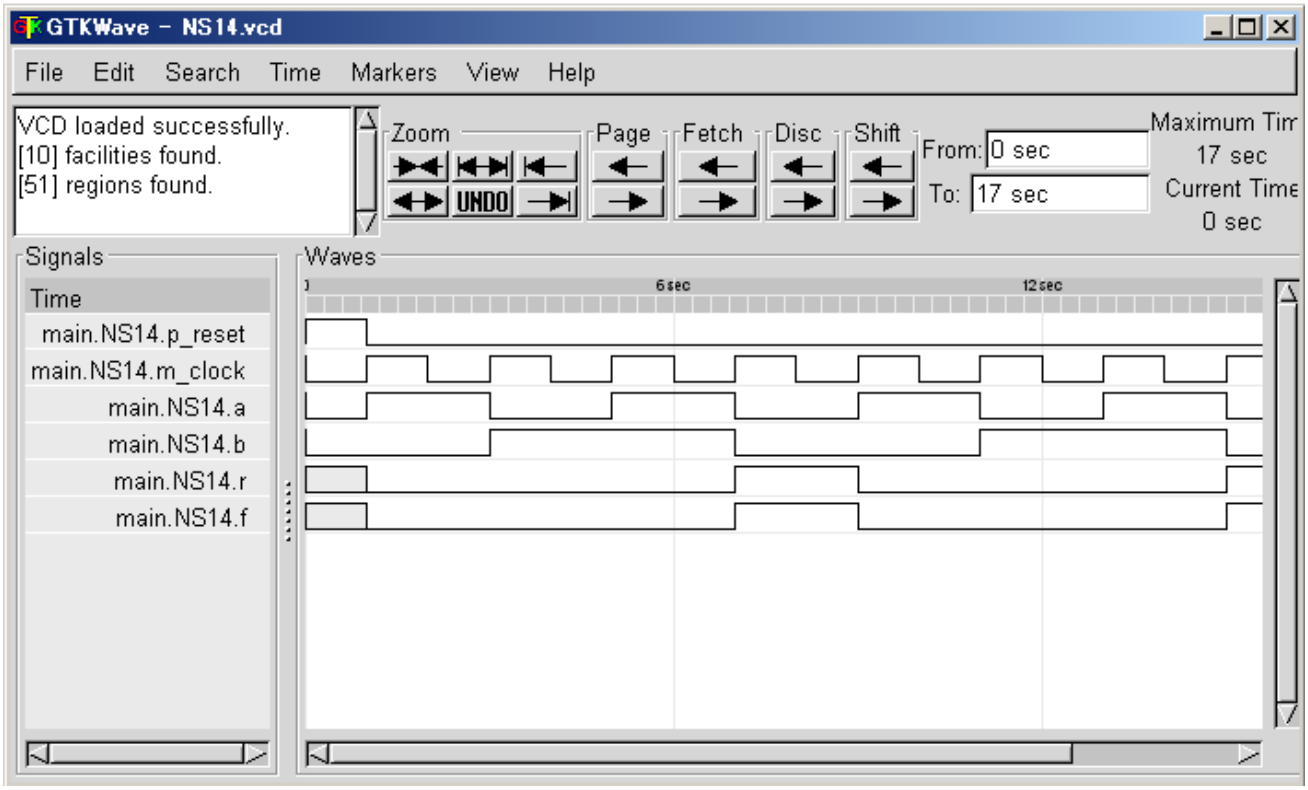
```
# ./exe NS14
```

The result will be available on your console.

```
a:0, b:0, r:x, f:x
a:1, b:0, r:0, f:0
a:0, b:1, r:0, f:0
a:1, b:1, r:0, f:0
a:0, b:0, r:1, f:1
a:1, b:0, r:0, f:0
a:0, b:1, r:0, f:0
a:1, b:1, r:0, f:0
a:0, b:0, r:1, f:1
```

We can see the wave form of this simulation as:

```
# ./exe -wave NS14
```


GTKWave - NS14.vcd
File Edit Search Time Markers View Help
VCD loaded successfully.
[10] facilities found.
[51] regions found.
Zoom
UNDO
Page
Fetch
Disc
Shift
From: 0 sec
To: 17 sec
Maximum Tim
17 sec
Current Time
0 sec
Signals
Time
main.NS14.p_reset
main.NS14.m_clock
main.NS14.a
main.NS14.b
main.NS14.r
main.NS14.f
Waves
6sec
12sec

12. Memory

Memory in NSL is a register with depth. It works on the rising edge of master clock (m_clock), but value can be referred without clock (asynchronous read).

Example-NS15

```
declare NS15 {
    input in[4],adr[8];
    output f[4];
    func_in write();
    func_in read();
}
module NS15 {
    mem m[256][4];
    func write {
            m[adr] := in;
    }
    func read {
            f=m[adr];
    }
}
```

We will make a simulation on this circuitry as following command:

```
# ./exe NS15
```

The result will be available on your console.

```
adr: 1, in: 8, f: x, read:0, write:1
adr: 2, in:11, f: x, read:1, write:0
adr: 3, in: 8, f: 9, read:0, write:1
adr: 0, in:11, f: x, read:1, write:0
adr: 1, in:14, f: 4, read:0, write:1
adr: 2, in:12, f: x, read:1, write:0
adr: 3, in: 1, f: 9, read:0, write:1
adr: 0, in: 1, f: x, read:1, write:0
```

adr: 1, in: 8, f: 4, read:1, write:0
adr: 2, in: 6, f:14, read:1, write:0
adr: 3, in: 2, f: 9, read:0, write:1
adr: 0, in: 3, f: x, read:1, write:0
adr: 1, in: 2, f: 4, read:1, write:0
adr: 2, in: 9, f:14, read:0, write:1
adr: 3, in: 1, f: x, read:1, write:0

13. Procedure

In NSL, sequential logic will be described with procedure and/or seq block. Procedure is a basic structure of a sequential logic and it is also a control signal to invoke the procedure. A procedure will be invoked from combinational logic or transferred the control from other procedure. It will be terminated with 'finish' or transfer control to other procedure. The transfer will terminate the current procedure and invoke a new one. Also '**finish()**' will terminate the current procedure. To terminate a procedure from outside of it, you must use explicit procedure name like *proc1.***finish();.** If you don't want to transfer control to the target procedure but simply want to invoke it, you can use invoke extension for it, like: *proc2*.**invoke();**. They will have arguments which consist of registers.

Example-NS16

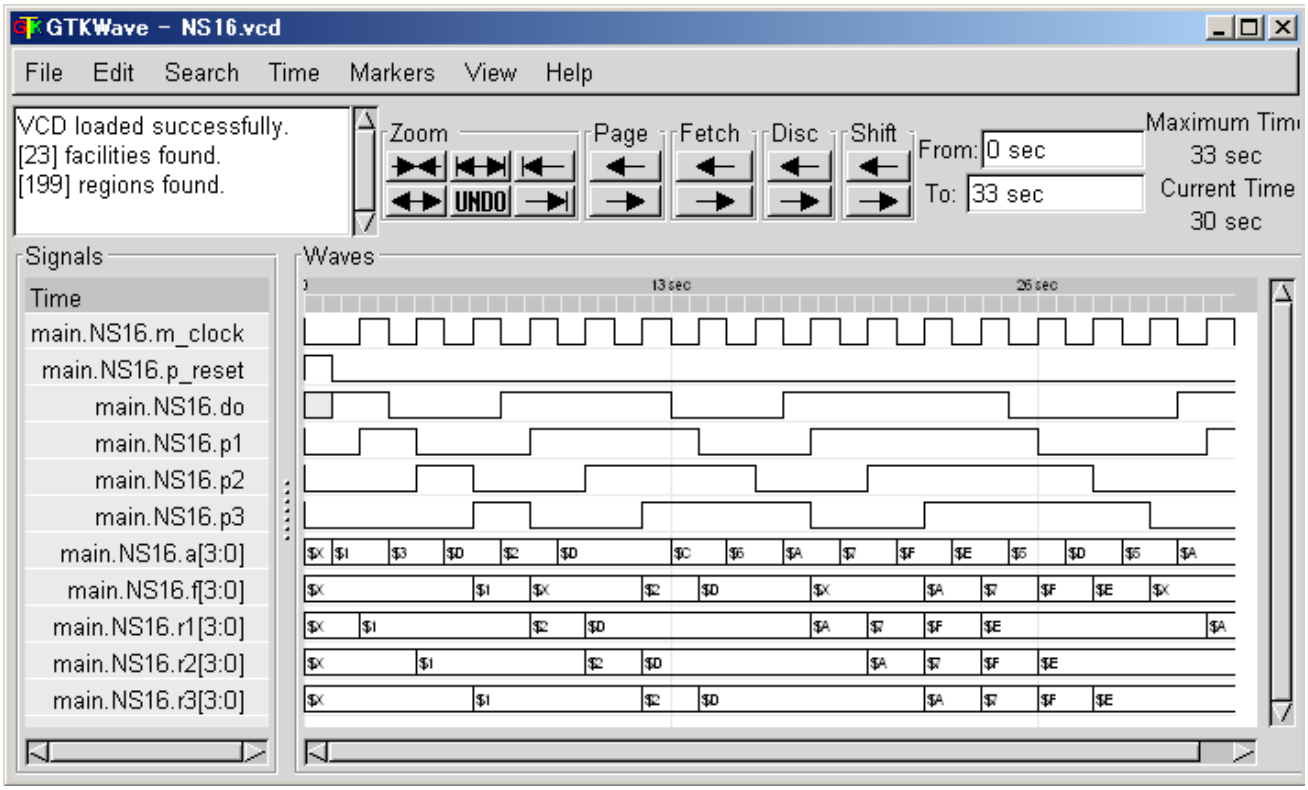
```
declare NS16 {
    input a[4];
    output f[4];
    func_in do(a);
}
module NS16 {
    reg r1[4],r2[4],r3[4];
    proc_name p1(r1);
    proc_name p2(r2);
    proc_name p3(r3);
    func do {
            p1(a);
    }

    proc p1 {
            p2(r1);
    }
    proc p2 {
            p3(r2);
    }
    proc p3 {
```

```
            f = r3;
            finish;
        }
    }
```

We will make a simulation on this circuitry as following command:

```
# ./exe -wave NS16
```

The result will be available on your console.



14. State Transition

We can describe state and state transitions in a stage with NSL. At power on reset, state will start with 'first_state'. The state will not change whenever the stage is terminated and/or invoked.

Example-NS17

```
declare NS17 {
    func_in do();
}
module NS17 {
    proc_name p();
    func do {
            p();
    }

proc p {
    state_name st1,st2,st3;
    first_state st1;
    state st1 goto st2;
    state st2 goto st3;
    state st3 { goto st1; finish; }
    }
}
```

We will make a simulation on this circuitry as following command:

```
# ./exe NS17
```

The result will be available on your console. 29

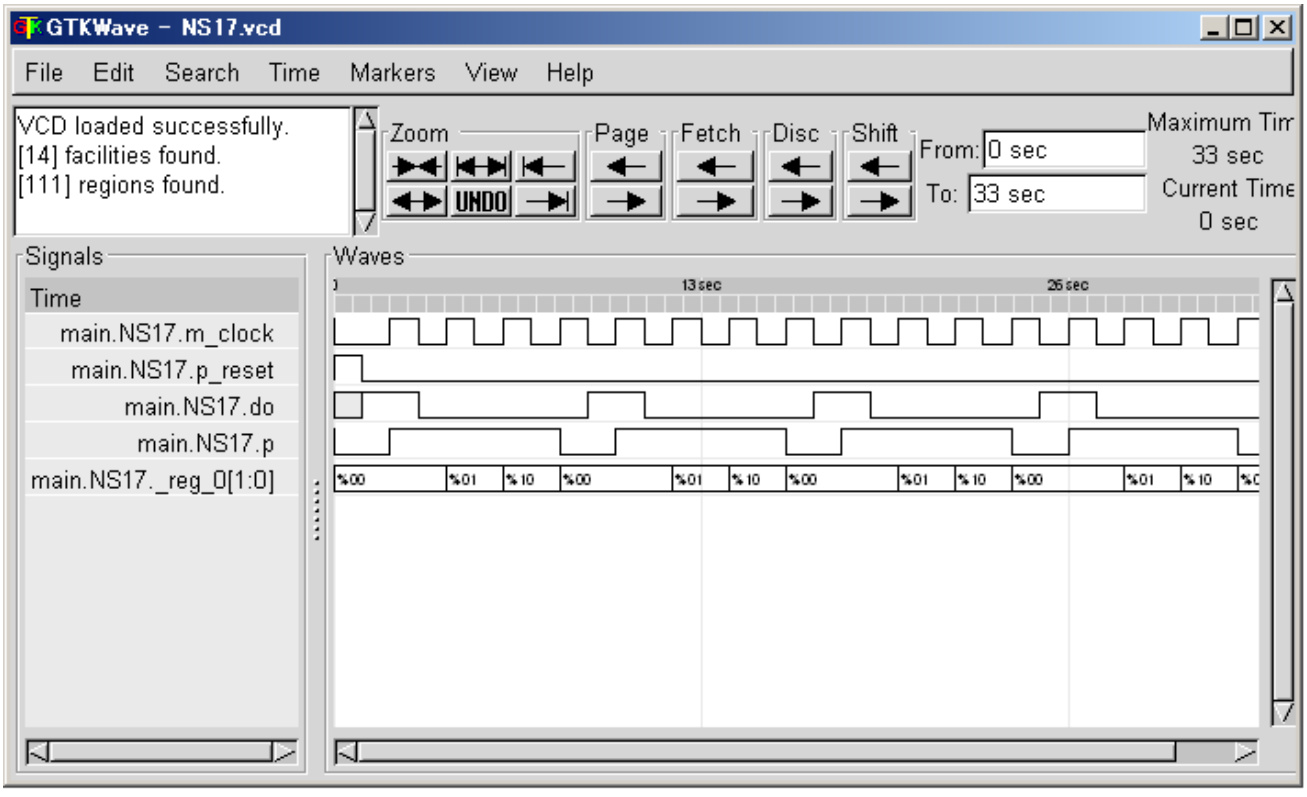
```
do:1 state:st1
do:0 state:st2
do:0 state:st3
```

do:0 state:st1
do:1 state:st1
do:0 state:st2
do:0 state:st3
do:0 state:st1
do:1 state:st1
do:0 state:st2
do:0 state:st3
do:0 state:st1
do:1 state:st1
do:0 state:st2
do:0 state:st3
do:0 state:st1

The state is hold in a state register and the value associated with the stage name is defined by compiler automatically.

Now see the wave form of the simulation.

 # ./exe -wave NS17



NSL Tutorial: Basic structure of NSL

Edition 1.0 2016/01/21

Author: Naohiko Shimizu

Editor: Akihiro Sawamura

Published: IP ARCH, Inc.

Printed: Overtone Corp.