High-level Design Beginner Training Course
Part 1 High-level Design Modeling Exercises

Renesas Electronics Corporation
Design Automation Department

2014/2/3 Rev. 1.0

RENESAS Group CONFIDENTIAL

LLWEB-00018077

Export Control No. LLWEB-00018077

Table of Contents

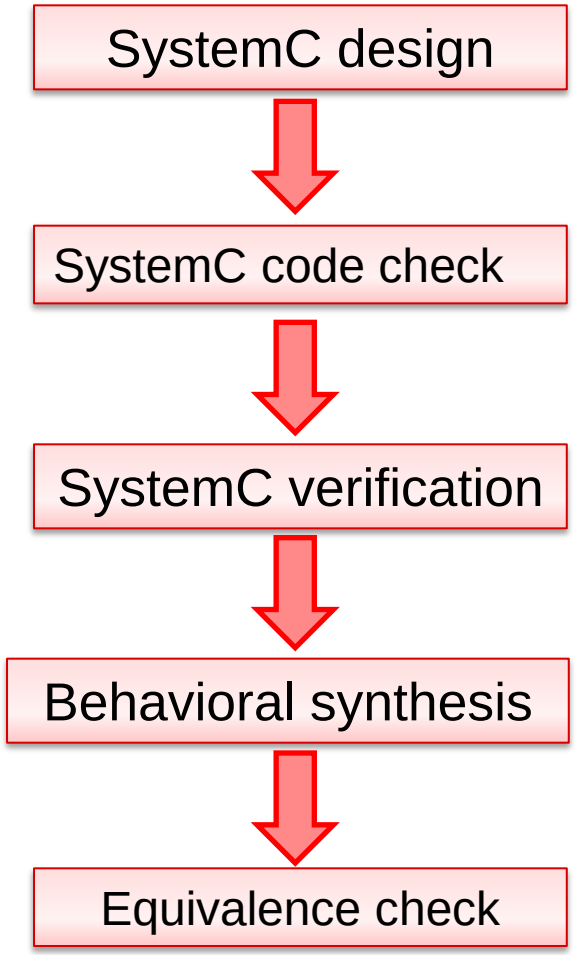
Course Prerequisites

- The high-level design beginner training course assumes basic knowledge of the topics listed below. If you are not knowledgeable about these topics, you may find this course difficult to understand. We recommend that you attend this course after obtaining necessary knowledge through the learning materials shown below.

Basic knowledge required	Learning material
Logic design methodology	•Intranet: **設計の杜** (Japanese web site) http://ppweb01.mu.renesas.com/knowledge/soc/designhome/ •Renesas technical course: RTL logic design exercises
C language programming	•Books (Japanese only) - **明解C言語入門編** (SB Creative Corp.) - C**の絵本** (SHOEISHA.Co.,Ltd.) •Web sites (Japanese web site) 42827198 -C language http://www.c-lang.org/ -Points of Learning C Language for Beginners http://www9.plala.or.jp/sgwr-t/
Overview of High-level Design	•LiveLink: “High-level Design Methodology and Application Examples” http://livelink.renesas.com/Livelink/livelink.exe/open/42827198 •【System-Level & High-Level Design/Verification】Training/Seminar Materials http://eda.develop.renesas.com/lv1ww/REL/training/en/System_High_level_Design_training.html

Purposes

- The High-level Design Beginner Training Course is divided into three parts. The purposes of each part are shown below in relation to the high-level design flow.



Part 1 Modeling Exercises (Based on This Teaching Document)

To help you learn the SystemC syntax and coding styles and create/code basic SystemC models for high-level design.
To help you detect and correct coding problems with SystemC code check tools (such as 1Team).

Part 2 Verification Exercises

To help you generate a test bench and verify the SystemC model (i.e., perform simulation-based debugging).
To help you obtain code coverage.

Part 3 Synthesis Exercises

To help you generate various RTLs from SystemC models with a behavioral synthesis tool (CtoS).
To help you check the equivalency between SystemC and RTL with an equivalence checker (SLEC).

Exercise Data

- Obtain exercise data for this training course from the following:

  Training materials for System-Level High-level Design/Verification

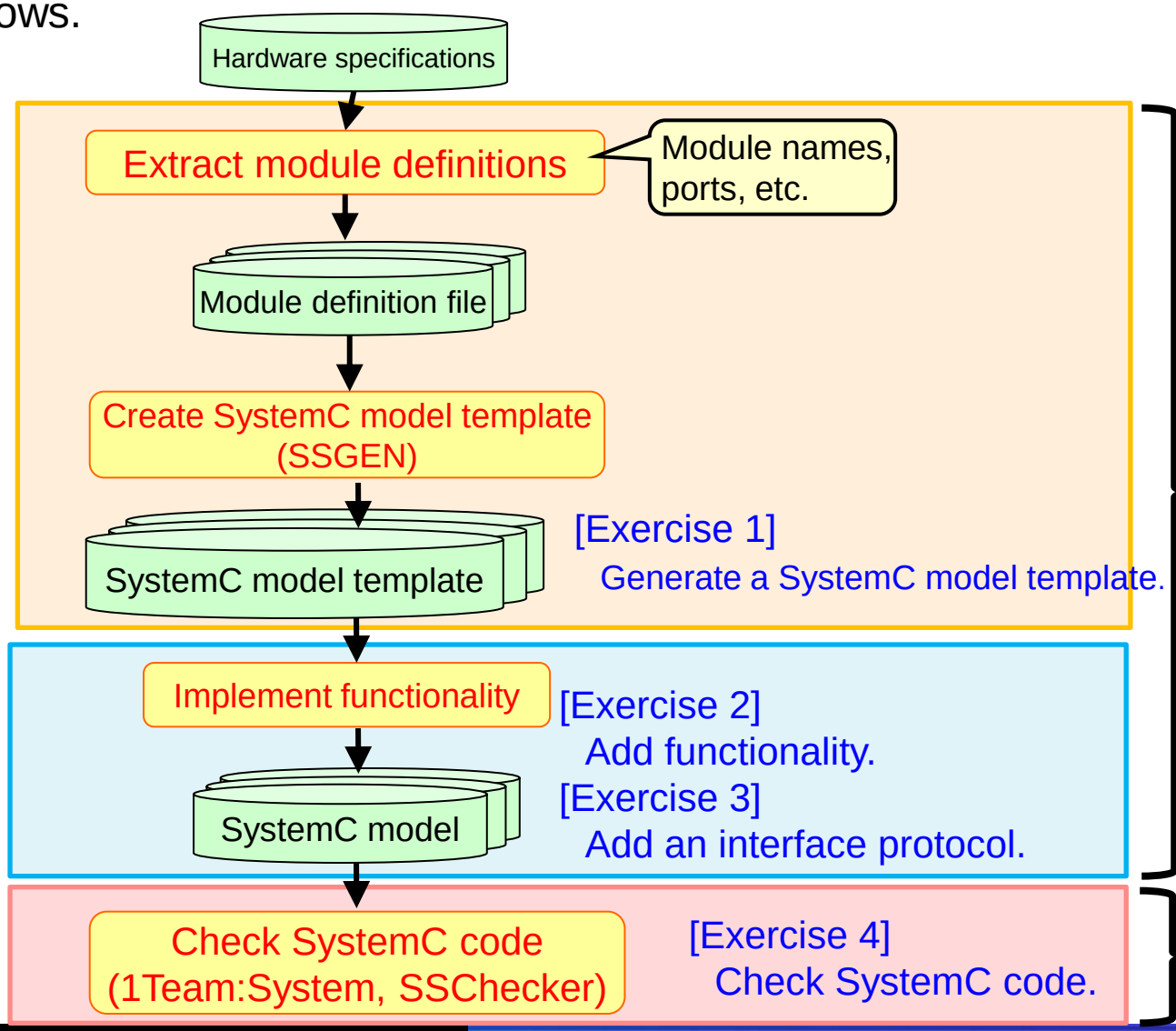
  http://eda.develop.renesas.com/lv1ww/REL/training/en/System_High_level_Design_training.html

  High-level Design Beginner Training Course

  1. High-Level Modeling Exercises (Exercise Data)

SystemC Design and SystemC Code Check Flows

- The SystemC design and SystemC code check flows are related to the exercises as follows.



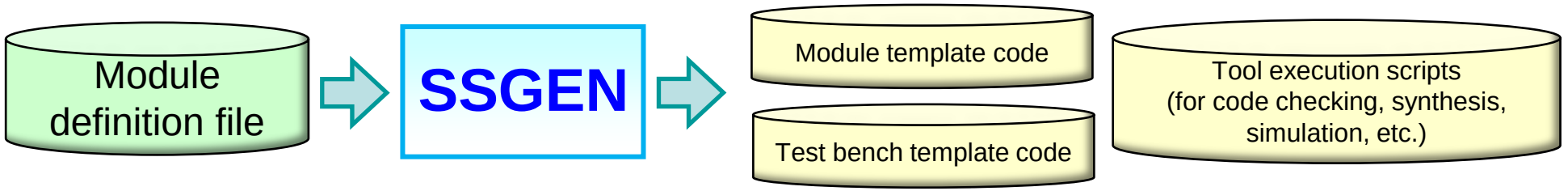
SystemC design
Create a synthesizable SystemC model by using the hardware specifications and algorithms (written in C/C++) as inputs.

SystemC code check
Check the SystemC model coding styles and problems.

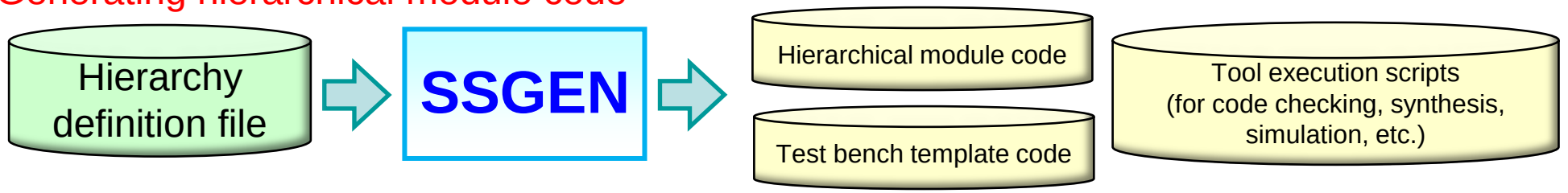
SystemC Design with SSGEN (1/5)

- The Synthesizable SystemC code generator (SSGEN), a high-level design support tool, allows you to generate module template code, hierarchical model code, test bench template code, and tool scripts simply by defining hardware interface, internal information, etc., in straightforward format.
  - SSGEN offers two features, one for generating module template code and the other for generating hierarchical module code.
  - Since the SystemC module template is automatically generated, designers can concentrate their efforts in implementing functionality. In this training course, you exercise using the module template code generation feature of SSGEN.

Generating module template code



Generating hierarchical module code



SystemC Design with SSGEN (2/5)

SSGEN commands shown in gray are not used in this exercise.

Module Definition File Format (SSGEN commands for This Exercise)

module *mod_name* — Specify a module name.

clock *clock_name* — Define a clock.

sreset *reset_name* pos|neg
areset *reset_name* pos|neg — Define reset (sreset for synchronous reset or areset for asynchronous reset). pos|neg specifies the active edge.

uin*wid* *in_name* — Define an input port (for generating sc_in). *wid* specifies the bit width or 'b' (which indicates a 1 bit bool type).

uout*wid* *out_name* [-init *initval*] — Define an output port (for generating sc_out). *Initval* specifies the reset value (which is 0 by default). *wid* is similar to uin.

ureg*wid* *reg_name* [-init *initval*] — Define a signal (register) for generating sc_signal.

uvar*wid* *member_name* [-init *initval*]
uint *member_name* [-init *initval*] — Define a member variable (internal data other than signal). uvar: SystemC type (bit width), uint: Integer type

umem *wid size memory_name type latency* [*options*] — Define a memory port (details are omitted). Single ports, two ports, and dual ports are supported.

cthread *thread_name* [-reset_header] — Define a process (operates by clock synchronization). –reset_header outputs the reset function to the header file.
method *method_name sensitivity_list* — Define a process (member function which automatically operates by changes in the sensitivity list).

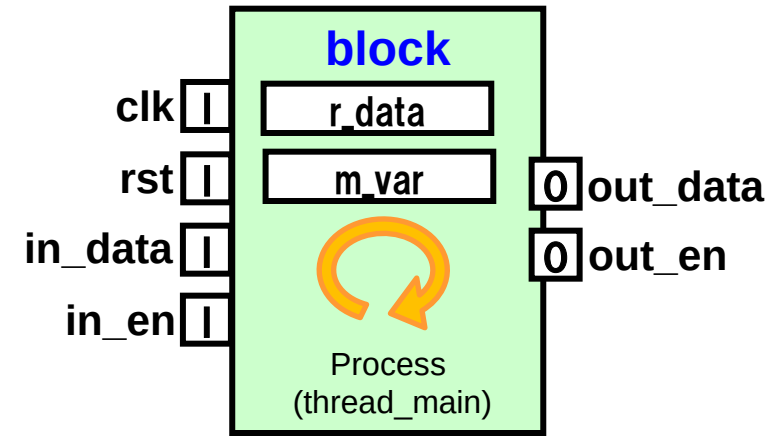
SystemC Design with SSGEN (3/5)

- Example of Generating a Module Definition File
  - Module name: block
  - Process name: thread_main
  - Interface and internal data

Name	I/O	SSGEN command	Data type	Reset value	Description
clk	IN	clock	–	–	Clock
rst	IN	sreset	–	–	Synchronous reset (polarity: pos)
in_data	IN	uin	sc_uint<16>	–	Input data
in_en	IN	uin	bool	–	Input enable
out_data	OUT	uout	sc_uint<16>	16'h0001	Output data
out_en	OUT	uout	bool	0	Output enable
r_data	–	ureg	sc_uint<8>	0	Internal data (register)
m_var	–	uvar	sc_uint<8>	0	Internal data (member variable)

block.in

```
module block

clock  clk
sreset  rst pos
uin16 in_data
uinb  in_en
uout16 out_data –init 1
uoutb out_en

ureg8 r_data
uvar8 m_var

cthread thread_main –reset_header
```



SystemC Design with SSGEN (4/5)

Reference information
(not used in the exercise)

Generating a Hierarchy Definition File

Hierarchy Definition File Format

```
top top_name
sub sub_file1.in  sub_instanace1
sub sub_file2.in  sub_instanace2

tap instance1.port  portname

bind instance1.outport  instance2.inport
```

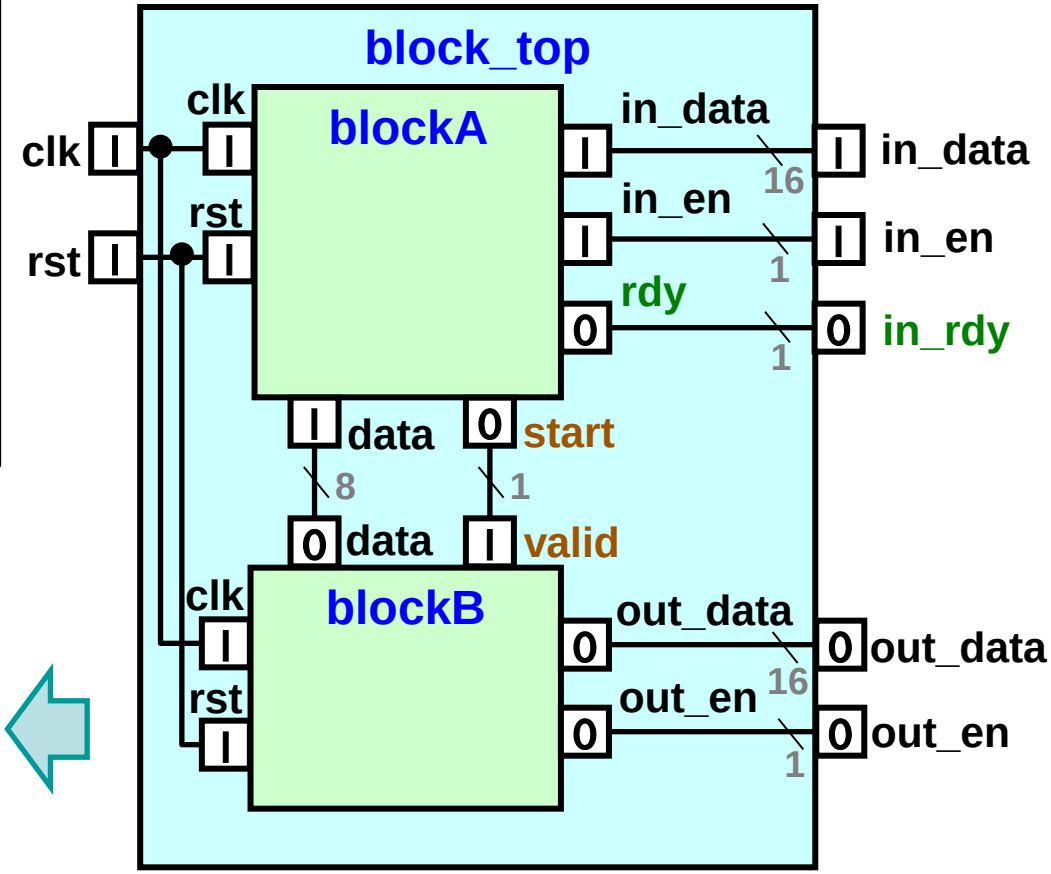
Specify a hierarchy module name.

List internal module definition files.

Generate tap ports from internal signals and ports.

Connect I/O ports of internal modules.

Example of Creating a Hierarchy Definition File



block_top.in

```
top block_top
sub blockA.in  ins_a
sub blockB.in  ins_b

tap ins_a.rdy in_rdy
bind ins_a.start ins_b.valid
```

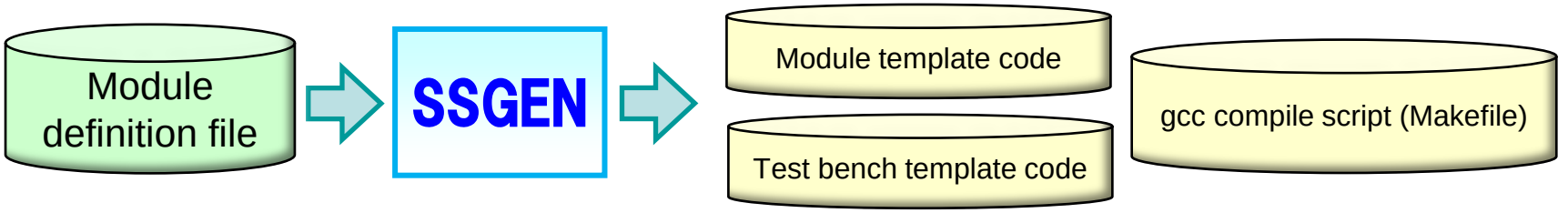
Define a hierarchy.

Request connection

If there is no tap/bind command, then internal port names are extracted and connection is provided between the ports with the same name.

SystemC Design with SSGEN (5/E)

- Example of Executing SSGEN
  - ssgen.pl -in *module definition file* -subdir -osci
    - Generate a SystemC code/test bench code template and gcc compile script.



Command Line Options (for This Exercise)

Option	**Meaning**
-in *filename*	Specifies an input file (for module or hierarchy definition). (Mandatory).
-subdir	Generates files in separate subdirectories. (By default, all files are created under the same path.)
-osci	Generates a Makefile for GCC compile and simulation.
-notb	Prevents a test bench file from being generated.
-only_script	Generates only a tool script.
-checker	Generates scripts to execute the SystemC style check (1Team:System and SSChecker).

Exercise 1: Creating a SystemC Model

- With SSGEN, create a SystemC module template to be used in this training course.

Exercise directory: modeling/ex1

Exercise 1-1. Creating an SSGEN Input File

- Generate an SSGEN input file according to the module specifications.

Exercise 1-2. Compiling the SSGEN Output File

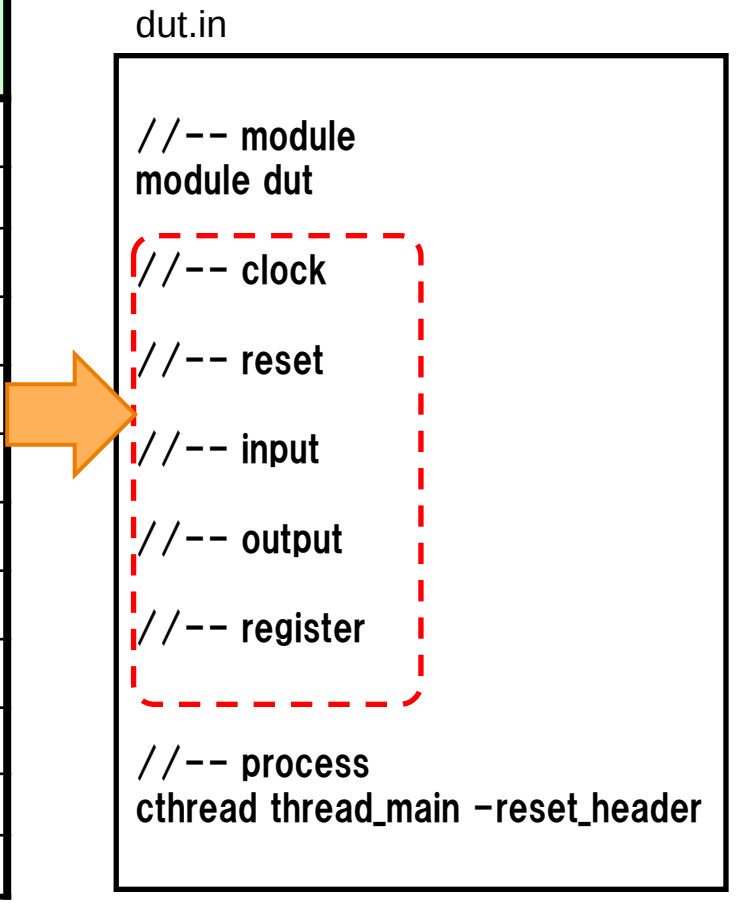
- Compile the SystemC module template.

Be sure to log in to the RHEL5.5 login server before the exercise. If you log in to some other server, a link error may occur at compile time because the gcc version of the server differs from that is used in SystemC library compilation.

Exercise 1: Creating a SystemC Model

- Exercise 1-1. Creating an SSGEN Input File
  - Create an SSGEN input file (dut.in) according to the module specifications below.

Signal name	I/O	SSGEN command	Data type	Reset value	Description
clk	IN	clock	-	-	Clock
rst	IN	sreset	-	-	Synchronous reset (polarity: pos)
in_vld	IN	uin	bool	-	Input Valid
in_rdy	OUT	uout	bool	1	Input Ready
in_op	IN	uin	sc_uint<3>	-	Operation code
in_d0	IN	uin	sc_uint<8>	-	Input data 0
in_d1	IN	uin	sc_uint<8>	-	Input data 1
out_vld	OUT	uout	bool	0	Output Valid
out_rdy	IN	uin	bool	-	Output Ready
out_data	OUT	uout	sc_uint<16>	0	Output data
pre_d0	-	ureg	sc_uint<8>	0	Previous-value hold register 0
pre_d1	-	ureg	sc_uint<8>	0	Previous-value hold register 1



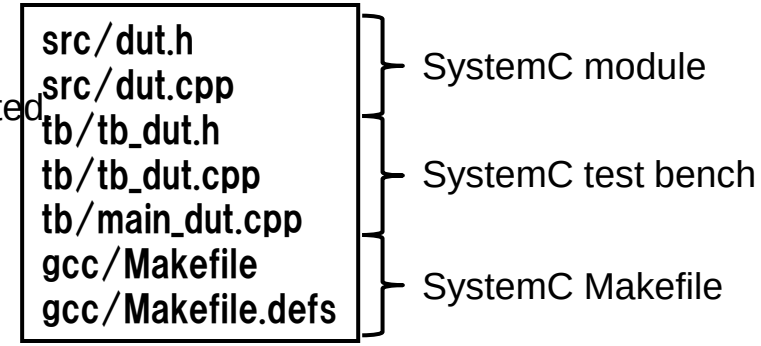
Exercise 1: Creating a SystemC Model

Exercise 1-2. Compiling the SSGEN Output File

- Compile the SystemC module template file.

1. Move to the exercise directory.
   %> cd modeling/ex1

2. Execute SSGEN.
   %> ssgen.pl -in dut.in -subdir -osci
   The file on the right will be generated.



3. Compile the file.
   %> cd gcc
   %> make
   run.exe will be generated.

4. Perform simulation.
   %> run.exe
   The following will be displayed to the standard output:

```
        SystemC 2.2.0 --- Dec  6 2010 16:13:34
    Copyright (c) 1996-2006 by all Contributors
                ALL RIGHTS RESERVED
SystemC: simulation stopped by user.
```

Checking SSGEN-Generated Code (1/3)

- Check the SystemC code generated in exercise 1.
  - Determine what information is generated from SSGEN and which file contains that information.
  - For details about the code, refer to the “Basic Knowledge of SystemC” section mentioned later.

Checking SSGEN-Generated Code (2/3)

- src/dut.h

```
#ifndef DUT_H
#define DUT_H

#include <systemc.h>
#include "ssgenlib.h"

SC_MODULE (dut) {
  sc_in < bool > clk;
  sc_in < bool > rst;
  sc_in < bool > in_vld;
  sc_in < sc_uint<3> > in_op;
  sc_in < sc_uint<8> > in_d0;
  sc_in < sc_uint<8> > in_d1;
  sc_in < bool > out_rdy;
  sc_out < bool > in_rdy;
  sc_out < bool > out_vld;
  sc_out < sc_uint<16> > out_data;

  sc_signal < sc_uint<8> > pre_d0;
  sc_signal < sc_uint<8> > pre_d1;
```

Include guard

Declare ports and signals.

Constructor

```
  SC_CTOR (dut)
    : clk ("clk")
    , rst ("rst")
    , in_vld ("in_vld")
    , in_op ("in_op")
    , in_d0 ("in_d0")
    , in_d1 ("in_d1")
    , out_rdy ("out_rdy")
    , in_rdy ("in_rdy")
    , out_vld ("out_vld")
    , out_data ("out_data")
    , pre_d0 ("pre_d0")
    , pre_d1 ("pre_d1")
  {
#ifndef _CTOS_TOP
    SC_CTHREAD (thread_main, clk.pos () );
    reset_signal_is (rst, true);
#endif
  }

  void reset_thread_main () {
    in_rdy.write (1);
    out_vld.write (0);
    out_data.write (0);
    pre_d0.write (0);
    pre_d1.write (0);
  }
```

Initialize port and signal names.

Register a process.

Function to initialize the output ports and signals

Declare an extended wait function.

```
  void my_wait ();
  void thread_main ();

#if !defined (_CTOS_) && !defined (CALYPTO_SYSC)
  void vcd_trace (ssgen_trace_file* tf, int depth = HIER_MAX) {
    if (tf != 0 && depth > 0) {
      std::string nm = std::string (name () );
      sc_trace (tf, clk, nm + ".clk");
      sc_trace (tf, rst, nm + ".rst");
      sc_trace (tf, in_vld, nm + ".in_vld");
      sc_trace (tf, in_op, nm + ".in_op");
      sc_trace (tf, in_d0, nm + ".in_d0");
      sc_trace (tf, in_d1, nm + ".in_d1");
      sc_trace (tf, out_rdy, nm + ".out_rdy");
      sc_trace (tf, in_rdy, nm + ".in_rdy");
      sc_trace (tf, out_vld, nm + ".out_vld");
      sc_trace (tf, out_data, nm + ".out_data");
      sc_trace (tf, pre_d0, nm + ".pre_d0");
      sc_trace (tf, pre_d1, nm + ".pre_d1");
    }
  }
#endif // !defined (_CTOS_) && !d
};

#ifdef _CTOS_
SC_MODULE_EXPORT (dut);
#endif

#endif // DUT_H
```

Declare a process function.

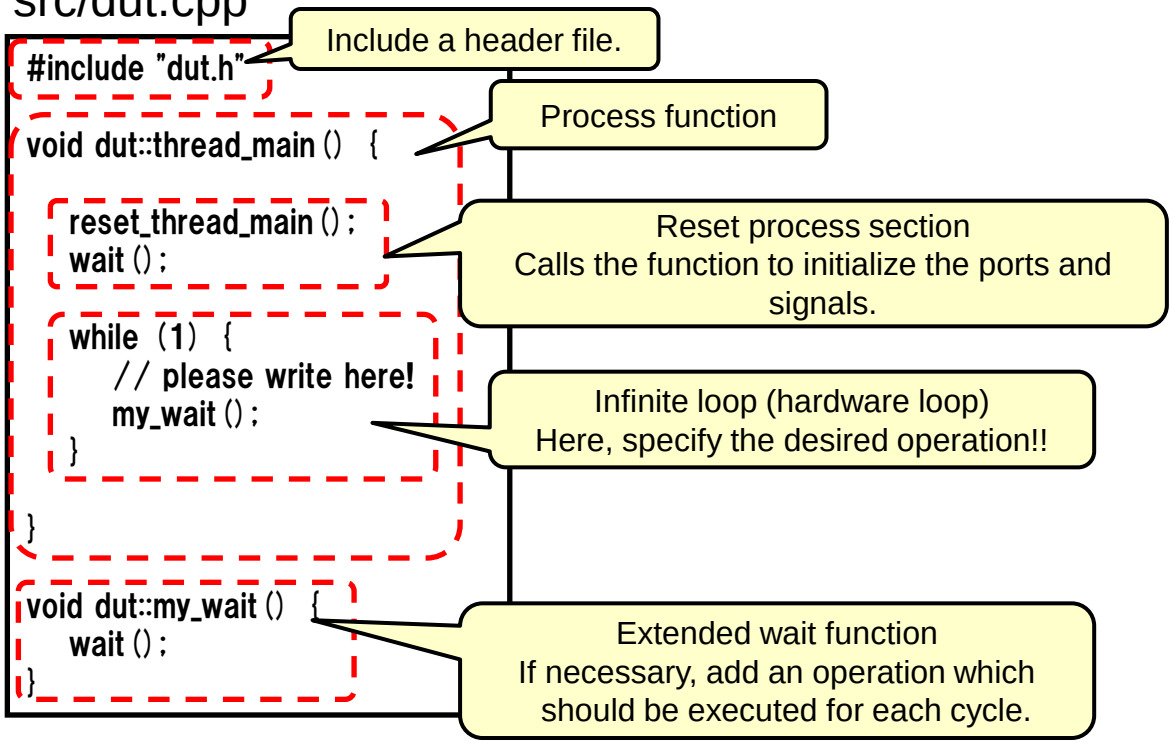
VCD trace code

Code for behavioral synthesis tool CtoS

Include guard

Checking SSGEN-Generated Code (3/E)

- src/dut.cpp



Basic Knowledge of SystemC

- In this chapter, you will learn basic knowledge of SystemC which is necessary for high-level design.
  - Modules
  - Variables
  - Input/output ports
  - Signals
  - wait()
  - Arrays
  - Branches
  - Loops
  - Functions
  - SC_CTHREAD
  - SC_METHOD
  - Partial selection
  - Concatenation
  - Assignments
  - Registers
  - Hierarchical modules
  - Waveform output
  - Log output
  - Comments
- Used in the exercise appears in the upper right-hand corner of pages. These pages provide you with knowledge necessary for the exercises in this training course.

Used in the exercise

Basic Knowledge of SystemC: Modules

- Below is an outline of the SystemC module structure (*).

module.h (header file)

```
#include <systemc.h>

SC_MODULE (module name) {

    <Port and signal declaration>
    <Member variable declaration>

    SC_CTOR(module name)
      : <Initialization list>
    {
        SC_CTHREAD (function name, clock name.pos());
        reset_signal_is(reset signal, reset polarity);
    }

    <Member function declaration>

};
```

Declare a module.

Constructor

Register a thread.

Specify the reset signal.

module.cpp (source file)

```
#include "module.h"

void module name::function name()
 {
    <Reset process>
    wait ();
    while (1) {
      <Operation description>
      wait ();
    }
}
```

Define the function registered in the thread.

Infinite loop

(*) When SSGEN is used, it generates code in the format above automatically.
The user only has to describe the desired operation in <Operation description>.

Basic Knowledge of SystemC: Variables (1/2)

Used in the exercise

- SystemC supports the following data types:

	Bit width	Sign bit (*) usage
bool	1	No
char	8	Yes
unsigned char	8	No
short	16	Yes
unsigned short	16	No
int	32	Yes
unsigned int	32	No
sc_int<N>	N (0 < N < 65)	Yes
sc_uint<N>	N (0 < N < 65)	No
sc_bigint<N>	N (N > 64)	Yes
sc_biguint<N>	N (N > 64)	No

(*)When a sign bit is used, it is MSB.

Basic Knowledge of SystemC: Variables (2/E)

- Variables are named differently depending on where they are declared.
  - Local variables
    - Local variables are declared in a function. They can be accessed (for reference or assignment) within the scope of that function only.
    - A variable which does not need to keep its value across cycles (“wait” mentioned later) should be declared as a local variable.
  - Member variables
    - Member variables are declared in the <Member variable declaration> area shown in the “Modules” section of this chapter. They can be accessed from any functions within that module. However, the access to one member variable from multiple processes might cause racing and is thus prohibited.
    - Member variables include the “input/output port” and “signal” data mentioned later.
    - A variable to be accessed from multiple functions should be declared as a member variable. When a variable is accessed from a single function, if its value must be kept across cycles, then it should be declared as a member variable.
  - Global variables
    - Global variables are declared outside of the module definition area. Do not use global variables for high-level design.

Do not use a global variable!

```
#include <systemc.h>

sc_uint<8> global_var;

SC_MODULE (module name) {

    <Port and signal declaration>
    sc_uint<8> member_var;

    …

};
```

Member variable

```
#include “module.h”

void module name::function name()
{
    wait();
    while (1) {
      sc_uint<8> local_var;
      …
      wait();
    }
}
```

Local variable

(*)When using SSGEN, you don’t have to consider the declaration position.

Used in the exercise

Basic Knowledge of SystemC: Input/Output Ports (1/2)

- Communication with the outside of a module is through its input/output ports.
  - Input ports of sc_in type
    - Declaration
      - Declare the input ports in the <Port and signal declaration> area shown in the “Modules” section of this chapter (*).

```
sc_in < bool > clk;
sc_in < bool > rst;
sc_in < sc_int<8> > data0_in;
sc_in < sc_biguint<128> > data1_in;
```

Clock reset should be of bool type.

Signed 8-bit input port

Unsigned 128-bit input port

    - Value reference
      - Use the read() method to reference the value of an input port.

```
sc_int<8> _d = data0_in.read ();
```

(*) When using SSGEN, you don’t have to consider the declaration position.

Used in the exercise

Basic Knowledge of SystemC: Input/Output Ports (2/E)

- Communication with the outside of a module is through its input/output ports.
  - Output ports of sc_out type
    - Declaration
      - Declare the output ports in the <Port and signal declaration> area shown in the “Modules” section of this chapter (*).

```
sc_out < bool > en_out;
sc_out < sc_int<8> > data0_out;
sc_out < sc_biguint<128> > data1_out;
```

Output port of bool type
Signed 8-bit output port
Unsigned 128-bit output port

    - Value reference
      - You can reference the value of an output port. Use the read() method to reference it.

```
sc_int<8> _d = data0_out.read();
```

    - Value assignment
      - Use the write() method to assign a value to an output port.

```
data0_out.write( _d );
```

(*) When using SSGEN, you don’t have to consider the declaration position.

Basic Knowledge of SystemC: Signals

Used in the exercise

- Communication between processes in a module is through signals. Also, you can use signals as registers.
  - Signals of sc_signal type
    - Declaration
      - Declare the signals in the <Port and signal declaration> area shown in the "Modules" section of this chapter (*).

```
sc_signal < bool > en_sig;
sc_signal < sc_int<8> > data0_sig;
sc_signal < sc_biguint<128> > data1_sig;
```

Signal of bool type
Signed 8-bit signal
Unsigned 128-bit signal

    - Value reference
      - Use the read() method to reference the value of a signal.

```
sc_int<8> _d = data0_sig.read ();
```

    - Value assignment
      - Use the write() method to assign a value to a signal.

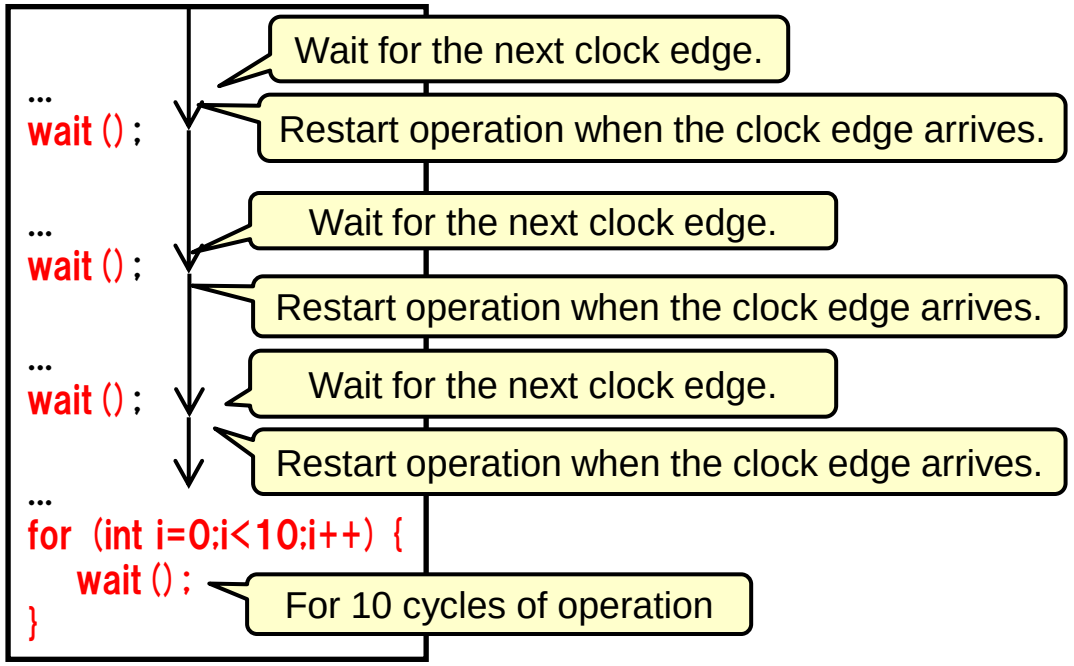
```
data0_sig.write ( _d );
```

(*) When using SSGEN, you don't have to consider the declaration position.

Basic Knowledge of SystemC: wait()

Used in the exercise

- Use wait() to express cycle operation.
  - In SC_CTHREAD for high-level design, express on cycle by using wait().
  - For express two or more cycles, use a number of wait() methods equal to the number of cycles or use a for statement.



Basic Knowledge of SystemC: Arrays (1/2)

- You can define ports, signals, and variables in arrays.
  - Arrays are useful for handling multiple items in a batch. SSGEN supports one-dimensional and two-dimensional arrays.
    - Whenever possible, the number of elements in a two-dimensional array should be a power of 2. Otherwise, the behavioral synthesis tool might generate too complex Verilog code for logic for accessing arrays.

```
sc_in < sc_uint<8> > ary0_in [4] ;
sc_out < sc_int<16> > ary0_out [2] [2] ;
sc_signal < sc_biguint<128> > ary0_sig [6] ;
sc_uint<24> ary0_var [8] [8] ;
```

Input ports defined in an unsigned 8-bit one-dimensional array
Output ports defined in a signed 16-bit two-dimensional array
Signals defined in an unsigned 128-bit one-dimensional array
Variables defined in an unsigned 24-bit two-dimensional array

  - Note 1
    - The behavioral synthesis tool creates array ports and signals and define them in Verilog as shown below.
      - One-dimensional arrays

```
sc_in < sc_uint<8> > ary1_in [4] ;
```

NAME[*i*] => NAME_*i*

```
input [7:0] ary1_in_0, ary1_in_1, ary1_in_2, ary1_in_3;
```

      - Two-dimensional arrays

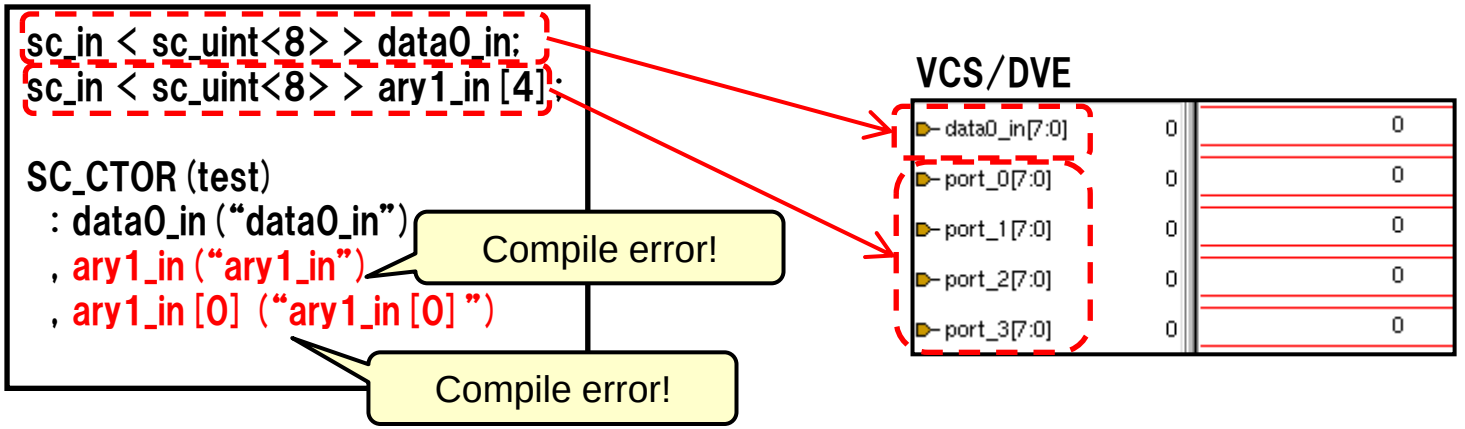
```
sc_out < sc_uint<16> > ary1_out [2] [2] ;
```

NAME[*i*][*j*] => NAME_*i*_*j*

```
output reg [15:0] ary1_out_0_0, ary1_out_0_1, ary1_out_1_0, ary1_out_1_1
```

Basic Knowledge of SystemC: Arrays (2/E)

- You can define ports, signals, and variables in arrays.
  - Note 2
    - The initialized names of ports and signals defined in arrays cannot be specified in the <Initialization list> area shown in the “Modules” section of this chapter.
      - If the names of ports and signals are not initialized with the <Initialization list>, these names are not registered for waveform dumping which involves a vendor tool (VCS/IES).
      - Use sc_trace to dump the waveforms for array ports and signals to a VCD file. Refer to the “Waveform Output” section mentioned later.



Basic Knowledge of SystemC: Branches

Used in the exercise

- Express branches in C syntax.
  - if-else

```
sc_uint<2> mode = ...;

if (mode==0) {
    out = func0 () ;
} else if (mode==1) {
    out = func1 () ;
} else if (mode==2) {
    out = func2 () ;
} else {
    out = func3 () ;
}
```

  - switch-case

```
sc_uint<2> mode = ...;

switch ( mode ) {
case 0:
    out = func0 () ; break;
case 1:
    out = func1 () ; break;
case 2:
    out = func2 () ; break;
case 3:
    out = func3 () ; break;
default:
    break;
}
```

Basic Knowledge of SystemC: Loops

Used in the exercise

- Express loops in C syntax.
  - for

```
sc_uint<8> i0;

for (i0=0; i0<10; i0++) {
    out [i0]  = func (i0) ;
}
```

  - while

```
sc_uint<8> i0;

i0 = 0;
while ( i0 < 10 ) {
  out [i0]  = func (i0) ;
  i0++;
}
```

  - do-while

```
sc_uint<8> i0;

i0 = 0;
do  {
  out [i0]  = func (i0) ;
  i0++;
} while ( i0 < 10 ) ;
```

- Combinational and Sequential Loops
  - A loop which is executed without consuming cycles is called a combinational loop. A loop which is executed while consuming cycles is called a sequential loop.

Combinational loop

```
sc_uint<8> i0;

for (i0=0; i0<10; i0++) {
    out [i0]  = func (i0) ;
}
```

Loop executed 10 times without consuming cycles

Sequential loop

```
sc_uint<8> i0;

for (i0=0; i0<10; i0++) {
    out [i0]  = func (i0) ;
    wait () ;
}
```

Loop executed 10 times while consuming 10 cycles (one execution per cycle)

Basic Knowledge of SystemC: Functions (1/2)

- Declare a function in the <Member function> area shown in the “Modules” section of this chapter. Enter the function in the source file (.cpp)(*).
  - This function is known as the “member function of a module” for high-level design.
  - We do not recommend you to use the global function declared in the global area.

module.h (header file)

```
#include <systemc.h>

SC_MODULE (module name) {
    ...
    SC_CTOR(module name)
    ...
    void func0 () ;
    sc_uint<16> func1 (sc_uint<8> a) ;
    ...

} ;
```

module.cpp (source file)

```
#include “module.h”

void module name::func0 ()  {
    <Operation description>
}

sc_uint<16> module name::func1 (sc_uint<8> a)  {
    sc_uint<16> rtn = 0;
    <Operation description>
    return rtn;
}
```

(*) When SSGEN is used, it generates code in the format above automatically.
The user only has to specify the desired operation in <Operation description>.

Basic Knowledge of SystemC: Functions (2/E)

- Combinational and Sequential Functions
  - A function which is executed without consuming cycles is called a combinational function. A function which is executed while consuming cycles is called a sequential function.

Combinational function

```
sc_uint<16> mod::comb_func (sc_uint<8> a) {
    sc_uint<16>rtn;

    rtn = a + 32;
    rtn = rtn << 8;
    return rtn;
}
```

Operation within a function is executed without consuming cycles.

Sequential function

```
sc_uint<16> mod::comb_func (sc_uint<8> a) {
    sc_uint<16>rtn;

    rtn = a + 32;
    wait () ;
    rtn = rtn << 8;
    wait () ;
    return rtn;
}
```

Operation within a function is executed while consuming two cycles.

Used in the exercise

Basic Knowledge of SystemC: SC_CTHREAD (1/2)

- SC_CTHREAD is a member function which runs by clock synchronization (*).
  - Register a process in the constructor when defining the member function.
  - The function registered as SC_CTHREAD should always be a void function without arguments.
  - Clock
    - For the second argument of the SC_CTHREAD macro, specify the clock name and synchronous edge (pos() or neg()).
    - Be sure to define a clock of bool type.
  - Reset
    - Synchronous reset reset_signal_is
      - For the first argument of the reset_signal_is macro, specify the reset name. For the second argument, specify the polarity (true or false).
    - Asynchronous reset async_reset_signal_is
      - Specify the same values as those for synchronous reset.
    - Be sure to define a reset of bool type

module.h (header file)

```
#include <systemc.h>

SC_MODULE (module name)  {
    sc_in < bool > clk;
    sc_in < bool > arst_n;
    sc_in < bool > rst;

   SC_CTOR(module name)
    {
        SC_CTHREAD (thread () , clk.pos () ) ;
        async_reset_signal_is (arst_n, false) ;
        reset_signal_is (rst, true) ;
    }

    void thread () ;
} ;
```

Clock reset of bool type

Synchronization with the clk rising edge

Asynchronous and synchronous resets

Declare a member function.

(*) When SSGEN is used, it generates code in the format above automatically.

Used in the exercise

Basic Knowledge of SystemC: SC_CTHREAD (2/E)

- Specify the reset section and infinite loop for the SC_CTHREAD function (*).
  - Reset section
    - Specify the output ports, signals, and member variable initialization.
    - For multiple resets, you can specify a branch and change the reset value.
    - Consume one cycle at the end of the reset section.
  - Infinite loop
    - Express an infinite loop by using while(1) or while(true).
    - Consume one or more cycles.
    - If there are any branches, consume one or more cycles for each branch.
  - A register is always added to the ports and signals written with SC_CTHREAD.

module.cpp (source file)

```
#include "module.h"

void module name::thread()
{
  if (arst_n.read () ==0)
    out.write (0) ;
  else
    out.write (0xF) ;
  wait () ;
  while (1) {
    <Operation description>
    wait () ;
  }
}
```

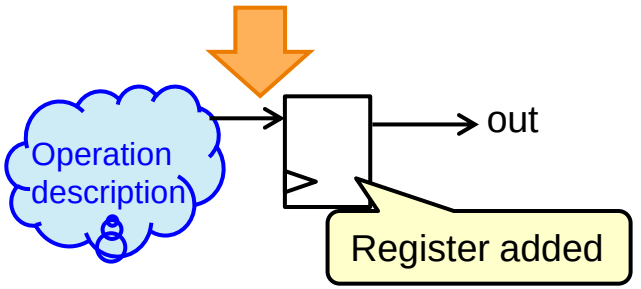
Reset section

Initialize the value to 0 for asynchronous reset and 0xF for synchronous reset.

One cycle

Infinite loop

One or more cycles



(*) When SSGEN is used, it generates code in the format above automatically.
The user only has to specify the desired operation in <Operation description>.

Basic Knowledge of SystemC: SC_METHOD (1/2)

- SC_METHOD is a member function which runs when a signal specified in the sensitivity list changes (*).
  - Register a process in the constructor when defining the member function.
  - The function registered as SC_METHOD should always be a void function without arguments.
  - Sensitivity list
    - In the sensitivity list, specify all the input/output ports and signals that should be referenced within SC_METHOD. If a port or signal is not specified in the sensitivity list, SC_METHOD does not operate when that port or signal changes.
    - SC_METHOD for synchronization with a clock edge should not be used for high-level design. Use SC_CTHREAD as the clock synchronization process.

module.h (header file)

```
#include <systemc.h>

SC_MODULE (module name) {
    sc_in < bool > in0;
    sc_in < sc_uint<2> > in1 [2];
    sc_in < sc_uint<1> > idx;

    SC_CTOR(module name)
    {
        SC_METHOD (method1);
        sensitive << in0;

        SC_METHOD (method2);
        sensitive << in1 [0] << in1 [1] << idx;
    }

    void method1 ();
    void method2 ();
};
```

Sensitivity list

For an array, specify the value for each element.

Declare a member function.

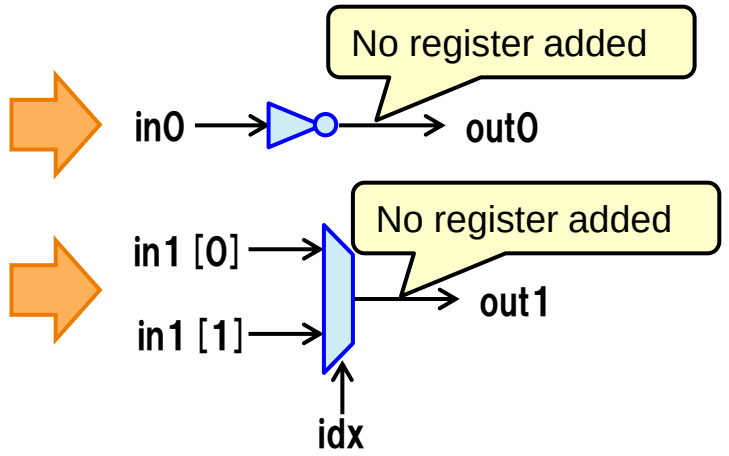
(*) When SSGEN is used, it generates code in the format above automatically.

Basic Knowledge of SystemC: SC_METHOD (2/E)

- You cannot specify cycle operation for the SC_METHOD function.
  - Do not specify cycle operation. You can specify only combinational circuit operation.
  - If any ports and signals are referenced within the SC_METHOD function, specify them in the sensitivity list.
  - Because you cannot specify cycle operation, no register is added to the ports and signals written with SC_METHOD.

module.cpp (source file)

```
#include "module.h"

void module name::method1 () {
    out0.write (! in0.read () ) ;
}

void module name::method2 () {
    out1.write ( in1 [idx.read () ] .read ()  ) ;
}
```



Basic Knowledge of SystemC: Partial Selection

- Use “range” to reference either particular bits in variables or a particular range of values. Also, use “range” to assign values to either particular bits in variables or a particular range of values.
  - You can specify “range” for variables of SystemC data types (sc_int, sc_uint, sc_bigint, and sc_biguint) only.
  - n and m for range(n, m) should satisfy n ≥ m.
  - You can specify only one variable for each range argument. With the first and second arguments, specify only the variable, (variable + constant), or (variable – constant).
  - You can combine “range” with .read() for the ports and signals. For example, you can use .read().range(0, 0).
  - You cannot combine “range” with .write() for the ports and signals.

```
sc_uint<4> data0;
sc_uint<2> data1;
sc_uint<1> data2;
sc_uint<3> data4;

data0.range (2,0)  = in0.read () ;
data0.range (3,3)  = in1.read () .range (2,2) ;

data1 = data0.range (3,2) ;
data2 = data0.range (0,0) ;

data4.range (0,0)  = data0.range (x,x) ;
data4.range (2,1)  = data0.range (x+1,x) ;
```

Assign values to bits 0 to 2 of data0.

Assign a value to bit 3 of data0 and reference bit 2 of in0.

Reference bits 2 and 3 of data0.

Reference bit 0 of data0.

Reference bit x of data0. (x>-1 && x<4)

Reference bit x – (x + 1) of data0.

Basic Knowledge of SystemC: Concatenation

Used in the exercise

- Use (,) to concatenate variables.
  - You can concatenate only variables of SystemC data types (sc_int, sc_uint, sc_bigint, and sc_biguint).

```
sc_uint<4> data0;
sc_uint<2> data1, data2;
sc_uint<1> data3, data4;

(data1, data2) = data0;

data0 = (data2, data3, data4);
```

Equivalent to data1=data0.range(3,2)and data2=data0.range(1,0).

Equivalent to data0.range(3,2)=data2, data0.range(1,1)=data3, and data0.range(0,0)=data4.

Basic Knowledge of SystemC: Assignments

Used in the exercise

- SystemC supports the concepts of non-blocking and blocking assignments.
  - Assignment to output ports and signals in SC_CTHREAD is non-blocking assignment.
  - Assignment to others is blocking assignment.

reg0_sig is of sc_signal type
(non-blocking assignments)

module.h

```
sc_signal < sc_uint<2> > reg0_sig;
```

module.cpp

```
reg0_sig.write (1);
a = reg0_sig.read ();      // a != 1
wait ();

a = reg0_sig.read ();      // a = 1
reg0_sig.write (2);
a = reg0_sig.read ();      // a = 1
wait ();

a = reg0_sig.read ();      // a = 2
```

reg0_var is of non-sc_out/sc_signal type
(blocking assignments)

module.h

```
sc_uint<2> reg0_var;
```

module.cpp

```
reg0_var = 1;
a = reg0_var;              // a = 1
wait ();

a = reg0_var;              // a = 1
reg0_var = 2;
a = reg0_var;              // a = 2
wait ();

a = reg0_var;              // a = 2
```

Basic Knowledge of SystemC: Registers (1/3)

Used in the exercise

- To express a register with SC_CTHREAD, use either of the following methods:

  Method 1: Use the sc_signal type.

  Method 2: Use a variable of non-sc_signal type. Assign a value to this variable and then reference the values before and after wait().

Method 1: reg0_sig is of sc_signal type.

module.h

```
sc_signal < sc_uint<2> > reg0_sig;
```

module.cpp

```
reg0_sig.write (1) ;

wait () ;

a = reg0_sig.read () ;
```

Hold a value.
a = 1

Method 2: reg0_var is of non-sc_signal type.

module.h

```
sc_uint<2> reg0_var;
```

module.cpp

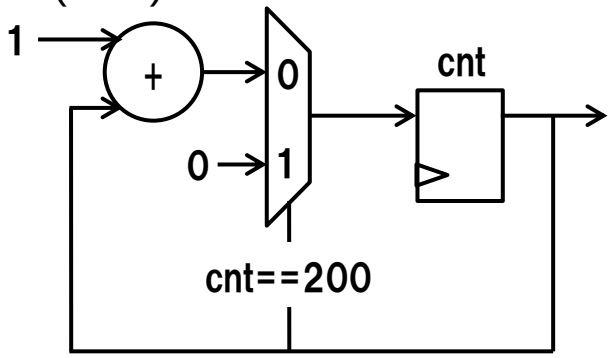
```
reg0_var = 1;

wait () ;

a = reg0_var;
```

Hold a value.
a = 1

	Merit	Demerit
Method 1	• The behavioral synthesis tool creates a register with the same name in RTL. • SystemC-RTL equivalence check can be completed soon.	• The register is not optimized by the behavioral synthesis tool.
Method 2	• The register is optimized by the behavioral synthesis tool (i.e., the register is shared). As a result, the circuit can be reduced in size.	• The register name differs between SystemC and RTL. • SystemC-RTL equivalence check might not be completed.

Basic Knowledge of SystemC: Registers (2/3)

- You can create a counter as a register.
  - Counter cnt (8-bit) which counts from 0 to 200.



Method 1: cnt is of sc_signal type.

module.h

```
sc_signal < sc_uint<8> > cnt;
```

module.cpp

```
if ( cnt.read () == 200 ) {
    cnt.write ( 0 );
} else {
    cnt.write ( cnt.read () + 1 );
}
wait ();
```

Method 2: cnt is of non-sc_signal type.

module.h

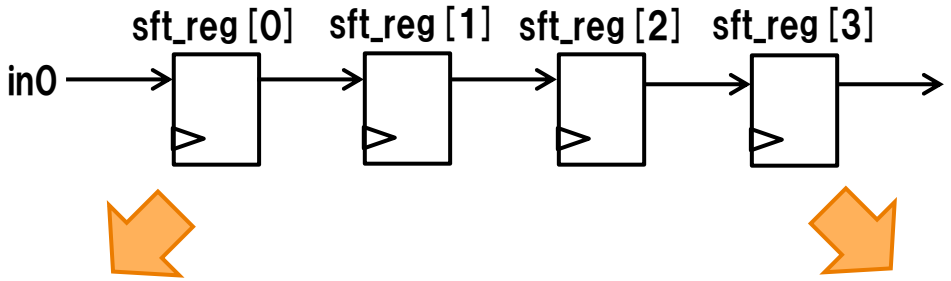
```
sc_uint<8> cnt;
```

module.cpp

```
if ( cnt == 200 ) {
    cnt = 0;
} else {
    cnt++;
}
wait ();
```

Basic Knowledge of SystemC: Registers (3/E)

- You can create a shift register by using arrays.
  - 4-stage shift register sft_reg[4]



Method 1: sft_reg0 [4] is of sc_signal type.

module.h

```
sc_signal < sc_uint<1> > sft_reg0 [4] ;
```

module.cpp

```
for (int i0=0;i<4;i++) {
    sft_reg0 [i0] .write ( sft_reg0 [i0-1] .read () ) ;
}
sft_reg0 [0] .write ( in0.read () ) ;

wait () ;
```

Method 2: sft_reg0 [4] is of non-sc_signal type.

module.h

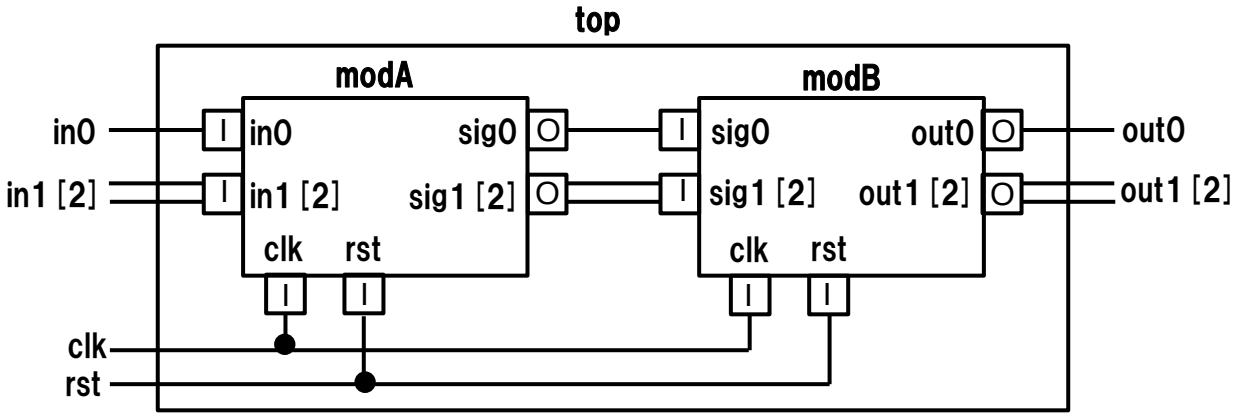
```
sc_uint<1> sft_reg0 [4] ;
```

module.cpp

```
for (int i0=0;i<4;i++) {
    sft_reg0 [i0] = sft_reg0 [i0-1] ;
}
sft_reg0 [0] = in0.read () ;

wait () ;
```

Basic Knowledge of SystemC: Hierarchical Modules

- Create instances of an internal module, define the connections between these instances, and define the port connection to the higher level (*).




```
SC_MODULE (top) {
  sc_in < bool > clk;
  sc_in < bool > rst;
  sc_in < bool > in0;
  sc_in < bool > in1 [2];
  sc_out < bool > out0;
  sc_out < bool > out1 [2];
  sc_signal < bool > sig0;
  sc_signal < bool > sig1 [2];
  modA A_ins;
  modB B_ins;
// To be continued
```

Declare ports.

Declare signals.

Create instances.

```
// Continued from previous code
SC_CTOR (top)
    ...
  , A_ins ("A_ins")
  , B_ins ("B_ins")
{
  A_ins.clk (clk);
  A_ins.rst (rst);
  A_ins.in0 (in0);
  for (int i0 = 0; i0 < 2; i0++) {
    A_ins.in1 [i0] (in1 [i0]);
  }
  A_ins.sig0 (sig0);
  for (int i0 = 0; i0 < 2; i0++) {
    A_ins.sig1 [i0] (sig1 [i0]);
  }
// To be continued
```

Initialize module names.

Connect to modA.

```
// Continued from previous code
  B_ins.clk (clk);
  B_ins.rst (rst);
  B_ins.sig0 (sig0);
  for (int i0 = 0; i0 < 2; i0++) {
    B_ins.sig1 [i0] (sig1 [i0]);
  }
  B_ins.out0 (out0);
  for (int i0 = 0; i0 < 2; i0++) {
    B_ins.out1 [i0] (out1 [i0]);
  }
}
};
```

Connect to modB.

Array ports and signals can be connected using a for statement.

(*) When SSGEN is used, it creates code in the format above automatically.

Basic Knowledge of SystemC: Waveform Output (1/3)

- There are two ways to output the waveform file:
  (1) Output a VCD file by using sc_trace (see the next page).
    - This method uses the SystemC waveform output syntax and does not require a vendor simulator.
    - The VCD file is larger than the waveform file shown in (2).
  (2) Output a waveform file for a vendor simulator (VCS or IES) (see the next page but one).
    - The waveform file is smaller than the VCD file shown in (1).
    - This method requires a vendor simulator.
    - The names of array ports and signals cannot be registered. So, they cannot be checked with a waveform viewer.

Basic Knowledge of SystemC: Waveform Output (2/3)

- You can trace the ports and signals by using the VCD format (*).
  - This waveform output does not depend on the vendor simulator.

sc_main function (test bench)

```
#include "test.h"

int sc_main (int argc, char *argv[]) {
    ...
    test test0 ("test0");
    ...
    sc_trace_file *tf = NULL;
    tf = sc_create_vcd_trace_file ("test");
    test0.vcd_trace (tf);
    sc_start ();
    sc_close_vcd_trace_file (tf);
    return 0;
}
```

Create test.vcd.

Call a module function to dump waveforms.

Complete waveform dump.

test.h (header file)

```
SC_MODULE (test) {
    sc_in < bool > clk;
    sc_in < bool > rst;
    sc_in < bool > in0;
    sc_in < bool > in1 [2];
    sc_out < bool > out0;
    ...
    void vcd_trace (sc_trace_file* tf) {
        std::string nm = std::string (name ());
        sc_trace (tf, clk, nm + ".clk");
        sc_trace (tf, rst, nm + ".rst");
        sc_trace (tf, in0, nm + ".in0");
        for (int i0 = 0; i0 < 2; i0++) {
            std::ostringstream indx;
            indx << " (" << i0 << ") ";
            sc_trace (tf, in1 [i0], nm + ".in1" + indx.str ());
        }
        sc_trace (tf, out0, nm + ".out0");
    }
};
```

Register waveforms to be dumped.

Register a port or signal as each element of the arrays by using a for statement.

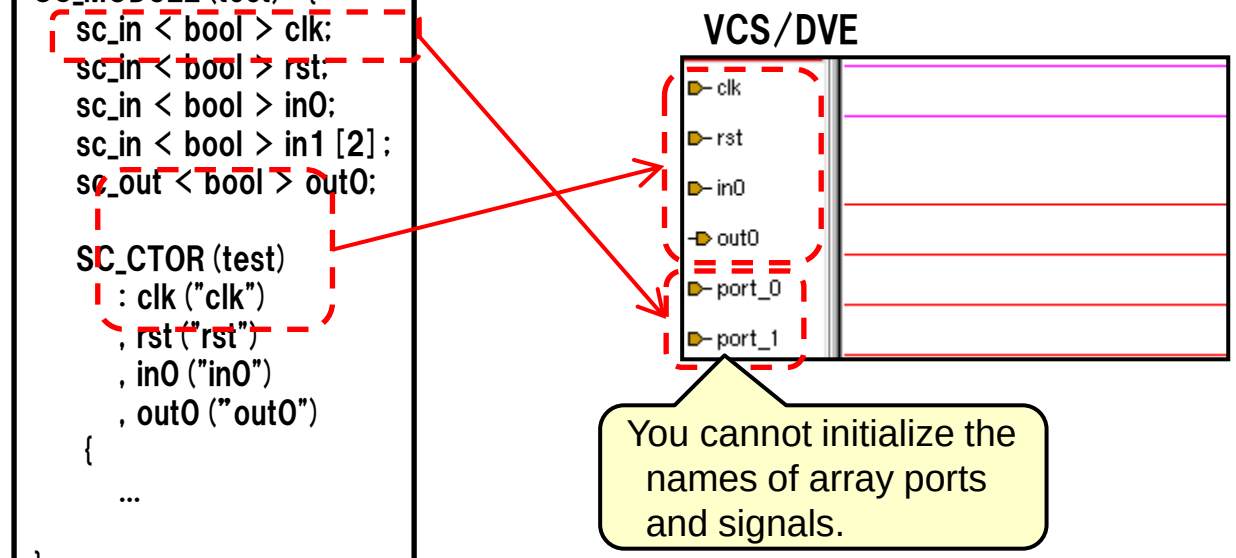
(*) When SSGEN is used, it creates code in the format above automatically.

Basic Knowledge of SystemC: Waveform Output (3/E)

- You can trace the ports and signals by using the waveform dump function of a vendor simulator.
  - You can trace the ports and signals with a waveform viewer after initializing the names in the <Initialization list> area shown in the “Modules” section of this chapter (*).
  - The format depends on the vendor simulator.
  - You cannot initialize the names of array ports and signals. So, you cannot check these names with a waveform viewer. To check the waveforms of array ports and signals, use sc_trace shown on the previous page.

test.h (header file)

```
SC_MODULE (test)  {
  sc_in < bool > clk;
  sc_in < bool > rst;
  sc_in < bool > in0;
  sc_in < bool > in1 [2] ;
  sc_out < bool > out0;

  SC_CTOR (test)
    : clk ("clk")
    , rst ("rst")
    , in0 ("in0")
    , out0 ("out0")
  {
    ...

} ;
```



(*)When SSGEN is used, it generates code in the format above automatically.

Basic Knowledge of SystemC: Log Output

- You can use the C/C++ syntax.
  - printf
    - When specifying a variable of SystemC data type for the argument, cast it to a C data type.
  - cout
    - When specifying a variable of SystemC data type for the argument, there is no need to cast it to a different data type.

```
sc_uint<2> data0;
sc_int<4> data1;

printf ("%d¥n", (unsigned) data0) ;
prinrf ("%d¥n", (int) data1) ;

printf ("%d¥n", data0.to_uint () ) ;
prinrf ("%d¥n", data1.to_int () ) ;

printf ("%x¥n", data0.to_uint () ) ;
```

Cast

to_unit() and to_int() are equivalent to cast.

Hexadecimal representation

```
sc_uint<2> data0;
sc_int<4> data1;

cout << data0 << endl;
cout << data1 << endl;

cout << hex << data0 << endl;
```

Hexadecimal representation

Basic Knowledge of SystemC: Comments

- You can comment out any code in SystemC just as you would comment out code in C/C++.
  - Use “//” to comment out a single line of code.
  - Use “/* */” to comment out multiple lines of code.
  - We recommend that you enter comments in English.
    - Enter comments in English when exporting a model to an overseas office or when using an editor which cannot display test in Japanese.

```
//x = a + b;
x = a - b; // change from + to -

/*
y = x + c;
z = y * d;
*/
y = x - c;
z = y << d;
```

Exercise 2: Adding Functionality

- Add functionality to the SystemC template created in exercise 1.

Exercise directory: modeling/ex2

- Exercise 2-1. Understanding Specifications
  - Understand the specifications of the functionality.
- Exercise 2-2. Creating SystemC Code
  - Add the functionality to SystemC code.
- Exercise 2-3. Checking the Operation
  - Compile the SystemC code and check its operation.

Be sure to log in to the RHEL5.5 login server before the exercise. If you log in to some other server, a link error may occur at compile time because the gcc version of the server differs from that is used in SystemC library compilation.

Exercise 2: Adding Functionality

- Exercise 2-1. Understanding Specifications (1/2)
  - Below are the specifications of the functionality to be implemented during this exercise.

    Functionality 1. Holding previous values

    Holds the in_d0 and in_d1 values in registers pre_d0 and pre_d1, respectively. The values stored in these registers will be used for functionality 2 shown below.

    Functionality 2. Operation

    Perform one of these operations according to the operation code (in_op).

in_op	Result
0	The in_d0 value when in_d0>in_d1, the in_d1 value otherwise
1	Sum of the in_d0 and in_d1 values
2	Product of the in_d0 and in_d1 values
3	Concatenation of the in_d0 and in_d1 bits. (The high-order 8 bits make up in_d0, and the low-order 8 bits in_d1.)
4	Value of previous-value hold register pre_d0
5	Value of previous-value hold register pre_d1
Others	Value of in_d0

Exercise 2: Adding Functionality

- Exercise 2-1. Understanding Specifications (2/E)
  - Below are the specifications of the functionality to be implemented during this exercise.

Functionality 3. Clipping

If the result of executing functionality 2 is greater than 0xFF00, output 0xFF00 to output port out_data. Otherwise, output this result to out_data.

Exercise 2: Adding Functionality

Exercise 2-2. Creating SystemC Code

- Write the functionality in System C.

1. Move to the exercise directory.

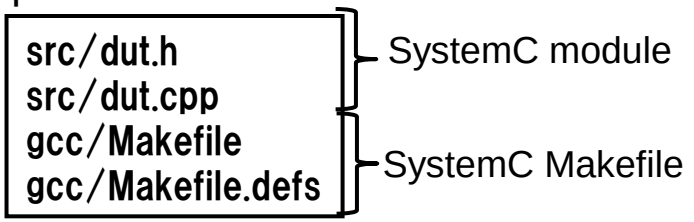
   %> cd modeling/ex2

2. Reexecute SSGEN.
   - Reexecute SSGEN by copying dut.in created in exercise 1. If necessary, modify dut.in.

     %> cp ../ex1/dut.in .
   - When reexecuting SSGEN, use the “-notb” option because, in this exercise, you should use the already created test bench to check the operation.

     %> ssgen.pl -in dut.in -subdir -osci -notb

     The file on the right will be created.



3. Create SystemC code.
   - Add the functionality, shown on the previous page, to src/dut.cpp. To change src/dut.h, change dut.in and reexecute SSGEN.
   - In this exercise, you do not need to use the input and output ports shown below.

     in_vld, in_rdy, out_vld, out_rdy

Exercise 2: Adding Functionality

Exercise 2-3. Checking the Operation

- After adding the functionality to the SystemC code, compile this SystemC code.

1. Compile the code.
   %> cd gcc
   %> make
   run.exe will be created.

2. Perform simulation.
   %> run.exe
   The following will be displayed to the standard output. If the output value and expected value match, “OK!!” is output.

```
SystemC 2.2.0 --- Dec  6 2010 16:13:34
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
OK!!
SystemC: simulation stopped by user.
```

If they do not, the pattern showing the mismatch is output.

```
NG!! : dut=103 exp=198 : op=0 d0=103 d1=198
```

Output value

Expected value

From left to right, the input values are in_op, in_d0, and in_d1.

Exercise 3: Adding an Interface Protocol

- Incorporate an input/output protocol into the SystemC code created in exercise 2.

Exercise directory: modeling/ex3

- Exercise 3-1. Understanding Specifications
  - Understand the specifications of the interface protocol.
- Exercise 3-2. Creating SystemC Code
  - Incorporate the input/output protocol into the SystemC code.
- Exercise 3-3. Checking the Operation
  - Compile the SystemC code and check its operation.

Be sure to log in to the RHEL5.5 login server before the exercise. If you log in to some other server, a link error may occur at compile time because the gcc version of the server differs from that is used in SystemC library compilation.

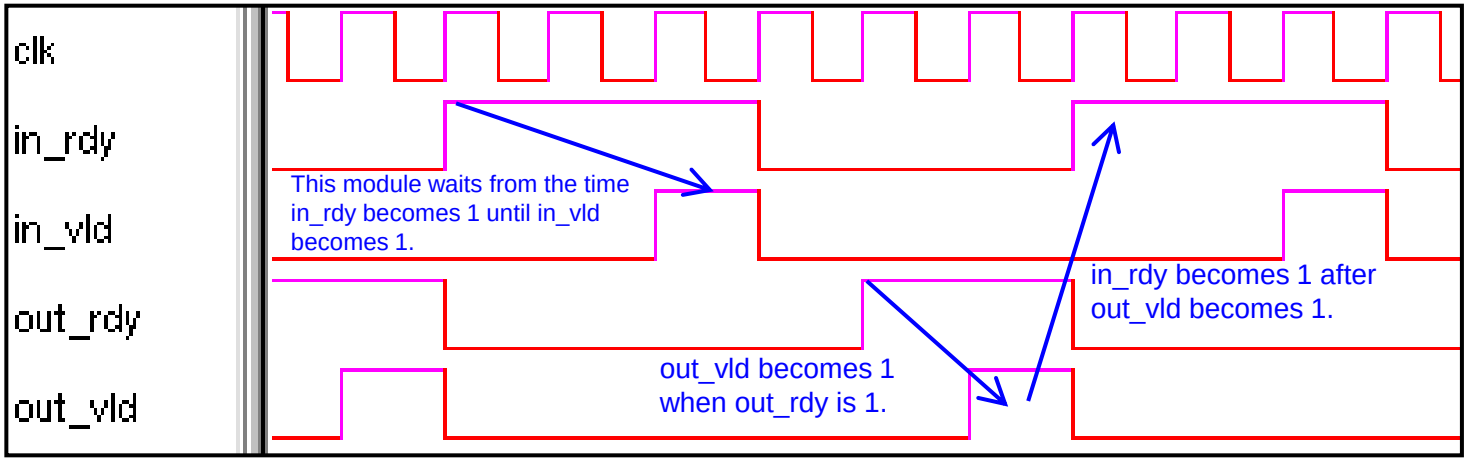
Exercise 3: Adding an Interface Protocol

- Exercise 3-1. Understanding Specifications (1/2)
  - Below are the specifications of the interface protocol implemented in this exercise.

Signal name	SSGEN command	Data type	Reset value	Description
in_vld	uin	bool	–	Input Valid • When this signal is 1, this module accepts data necessary for operation. • This module waits from the time this signal becomes 0 until it becomes 1.
in_rdy	uout	bool	1	Input Ready • This signal becomes 0 when operation begins. • After the result of operation (out_vld = 1) is output, this signal becomes 1 when the input value for the next operation becomes acceptable.
out_rdy	uin	bool	–	Output Ready • This module waits from the time this signal becomes 0 until it becomes 1. • This module outputs the result of operation (out_vld = 1) when this signal becomes 1.
out_vld	uout	bool	0	Output Valid • When out_rdy = 1 is already confirmed, this signal becomes 1 and the result of operation is output. • Otherwise, this signal becomes 0.

Exercise 3: Adding an Interface Protocol

- Exercise 3-1: Understanding Specifications (2/E)
  - Below is a waveform based on the specifications shown before.



Exercise 3: Adding an Interface Protocol

- Exercise 3-2. Creating SystemC Code
  - Write the interface protocol in SystemC.

1. Move to the exercise directory.

   ```
   %> cd modeling/ex3
   ```

2. Copy the SystemC code created in exercise 2.
   - Copy the SSGEN input file and SystemC code created in exercise 2. In exercise 3, incorporate the interface protocol into the SystemC code created in exercise 2.

     ```
     %> cp ../ex2/dut.in .
     %> cp -r ../ex2/src .
     ```

3. Create SystemC code.
   - Incorporate the protocol, based on the specifications shown on the previous page, into src/dut.cpp. To change src/dut.h, change dut.in and reexecute SSGEN.

Exercise 3: Adding an Interface Protocol

Exercise 3-3. Checking the Operation

- After incorporating the interface protocol into the SystemC code, compile this SystemC code.

1. Compile the code.
   %> cd gcc
   %> make
   run.exe will be created.

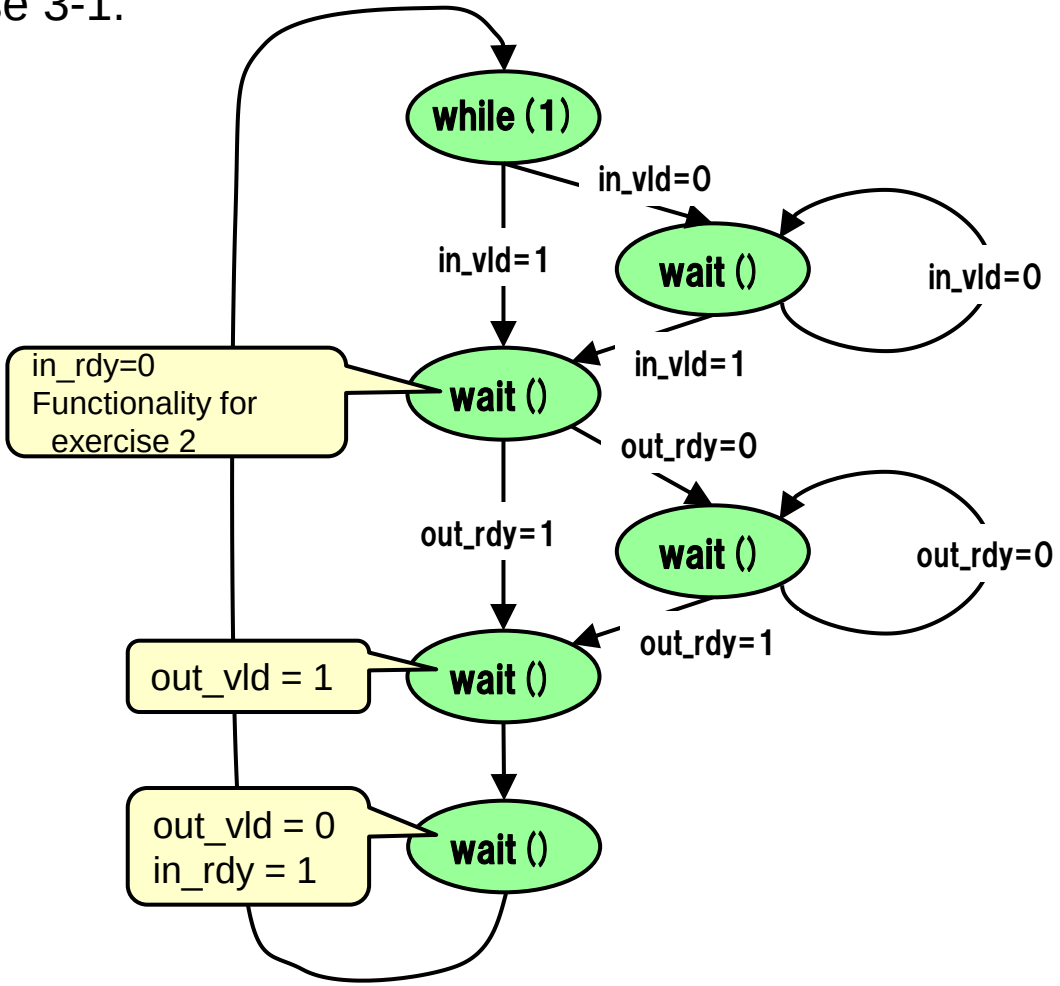
2. Perform simulation.
   %> run.exe
   The following is displayed to the standard output. If an output value and expected value match, “OK!!” is output.

```
SystemC 2.2.0 --- Dec  6 2010 16:13:34
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
OK!!
SystemC: simulation stopped by user.
```

   To output a waveform, specify –vcd in the execution command. This creates dut.vcd.
   %> run.exe -vcd

Exercise 3: Adding an Interface Protocol

- Tips for exercise 3 (1/2)
  - Below is a state transition diagram based on the specifications shown in exercise 3-1.



Exercise 3: Adding an Interface Protocol

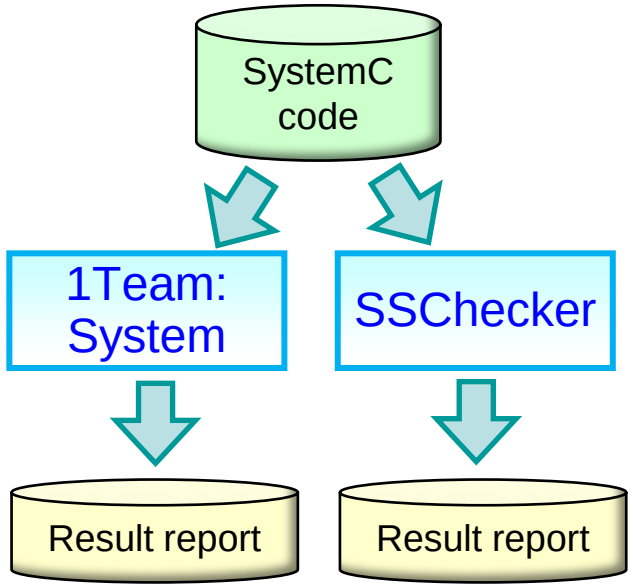
- Tips for exercise 3 (2/E)
  - The operation “Waiting until a specific condition is satisfied” shown on the previous page
    - Use “while”.

```
while ( cond ) {
  function;
}
While “cond” is satisfied, “function” is repeatedly executed.
```

    - What are “cond” and “function” for this exercise?

Checking SystemC Code

- The following two style checkers are available to check SystemC models:
  - **1Team:System** (by Atrenta)
    This tool checks the C++ and SystemC coding styles. Using a set of high-level design rules (about 250 rules), it detects any coding problems in early stages.
  - **SSChecker** (by Renesas)
    1Team:System cannot check some coding styles. SSChecker checks these coding styles (against 14 rules).
- Use both these tools to check your created SystemC model. You can use them in any order.
- Input and output



SSGEN can generate a checker execution script.

- Examples of Check Rules

Checker	Check rule
1Team	Do not access the same member variable (other than port and signal data) in multiple processes.
1Team	Use a read/write method to access the ports and signals.
1Team	Do not access areas outside of the bounds of an array.
1Team	Do not perform multi-drive operation on the same port or signal in multiple processes.
SSChecker	Do not write a comment in Japanese.
SSChecker	Do not use a tab character.
SSChecker	Do not use a global function.

Exercise 4: Checking SystemC Code

- Check the SystemC code created in exercise 3.

Exercise directory: modeling/ex4

- Exercise 4-1. Creating Checker Scripts
  - Create scripts for executing both 1Team:System and SSChecker.
- Exercise 4-2. 1Team:System
  - Execute 1Team:System.
- Exercise 4-3. SSChecker
  - Execute SSChecker.

Exercise 4: Checking SystemC Code

Exercise 4-1. Creating Checker Scripts

- Create code check scripts by using SSGEN.

1. Move to the exercise directory.

   %> cd modeling/ex4

2. Copy the SystemC code created in exercise 3.
   - Copy the SSGEN input file and SystemC code created in exercise 3.

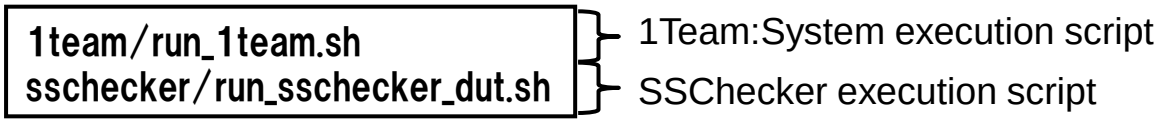
     %> cp ../ex3/dut.in .

     %> cp -r ../ex3/src .

3. Execute SSGEN.
   - Execute SSGEN as shown below. Only the code check scripts will be created.

     %> ssgen.pl -in dut.in -subdir -only_script -checker

     The files below will be created.



Exercise 4: Checking SystemC Code

```
------------------------------------------------------
Results Summary:
------------------------------------------------------
   0 FATAL Severity Messages

  Total Number of Messages     :       5
  Number of Messages Filtered  :       5  (5 waived)
  Number of Messages Displayed :       0

Note: 1. For Rule messages:              Count 'Policy'  [Severity Label]
      2. For SpyGlass messages:          Count SpyGlass Message-Type
      3. For High-Profile Rule messages: Count RuleName message(s)
------------------------------------------------------
```

- Exercise 4-2. 1Team:System
  - Execute 1Team:System.

1. Execute 1Team:System.
   %> cd 1team
   %> run_1team.sh
   spyglass.log and moresimple.rpt will be output.

2. Check the reports.
   - Make sure that “Number of Messages Displayed” at the end of spyglass.log is 0 (see the figure in the upper right corner).
   - If it is not 0, check the contents of moresimple.rpt. Then, correct the SystemC code by referring to the reports. For details about the reports, refer to the vendor manuals shown in Chapter 10 of the manual below.

User’s Manual for C/C++/SystemC Code Check Tool “1Team:System”
http://livelink.renesas.com/Livelink/livelink.exe/open/38420997

Exercise 4: Checking SystemC Code

- Exercise 4-3. SSChecker
  - Execute SSChecker.

1. Execute SSChecker.
   %> cd sschecker
   %> run_sschecker_dut.sh
   dut.rpt will be output.

2. Check the reports.
   - Make sure that no report is output to dut.rpt.
   - If any reports are output to dut.rpt, correct the SystemC code by referring to the reports.
     For details about the reports, refer to the following manual:

     User's Manual for High-Level Design Supporting Utility "SSChecker"
     http://livelink.renesas.com/Livelink/livelink.exe/open/40076956

Summary

- In the high-level design modeling exercises, you have learned these topics:
  - How to generate a SystemC model by using SSGEN
  - Basic knowledge of SystemC
  - How to implement functionality and an interface protocol
  - How to check SystemC code

Proceed to the high-level design verification exercise.

Answers to Exercises

- The exercise data file located by the path below contains the answers to the exercises.
  Refer to this file if you cannot find answers to questions.
  modeling/.answer/ex*

Renesas Electronics Corporation

Revision History

	Date of Issue	Description of Revision	Approved by	Checked by	Created by
Rev.1.0	Feb. 3, 2014	Newly created	SIDA Asano Feb.3, 2014	-	SIDA Imamura Feb. 3, 2014