# High-level Design Beginner Training Course
## Part 2 High-level Design Verification Exercises

Renesas Electronics Corporation
Design Automation Department

2014/2/3 Rev. 1.0

LLWEB-00018077

Export Control No. LLWEB-00018077

## Table of Contents

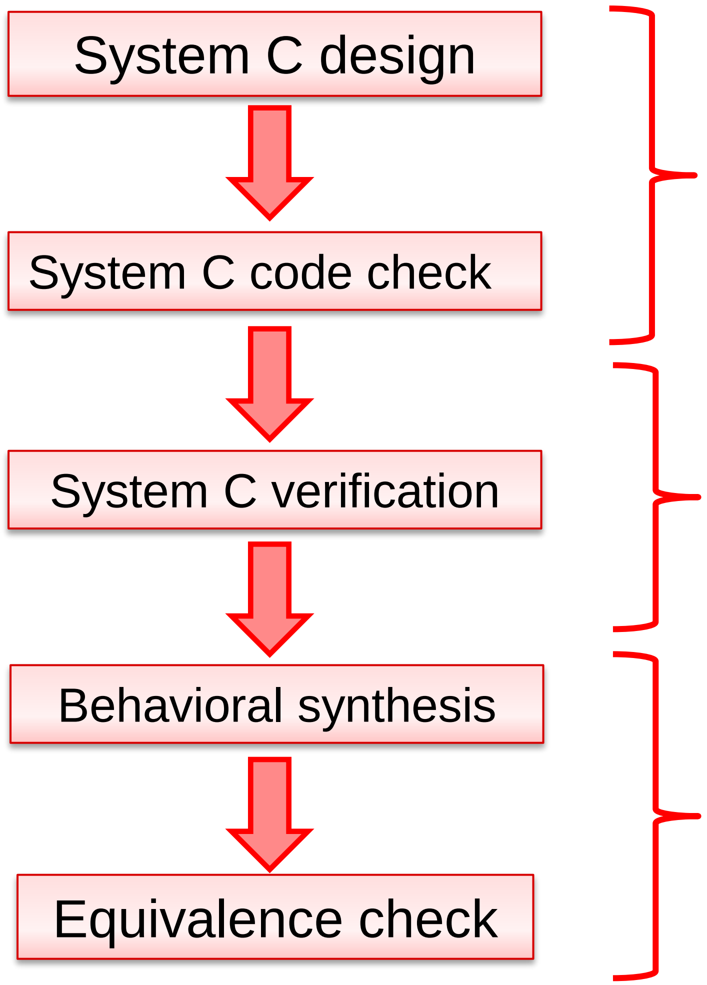
## Course Prerequisites

- 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.
  - We also recommend that before carrying out the verification exercises, you finish the “Part 1 High-level Design Modeling Exercises”.

| Basic knowledge required | Learning material |
|---|---|
| Logic design methodology | •Intranet: **設計の杜** (Japanese web site)<br>http://ppweb01.mu.renesas.com/knowledge/soc/designhome/<br>•Renesas technical course: RTL logic design exercises |
| C language programming | •Books (Japanese only)<br>- **明解C言語入門編** (SB Creative Corp.)<br>- C**の絵本** (SHOEISHA.Co.,Ltd.)<br>•Web sites (Japanese web site) 42827198<br>-C language<br>http://www.c-lang.org/<br>-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”<br>http://livelink.renesas.com/Livelink/livelink.exe/open/42827198<br>•【System-Level & High-Level Design/Verification】Training/Seminar Materials<br>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**
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 (Based on This Teaching Document)**
To help you create 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

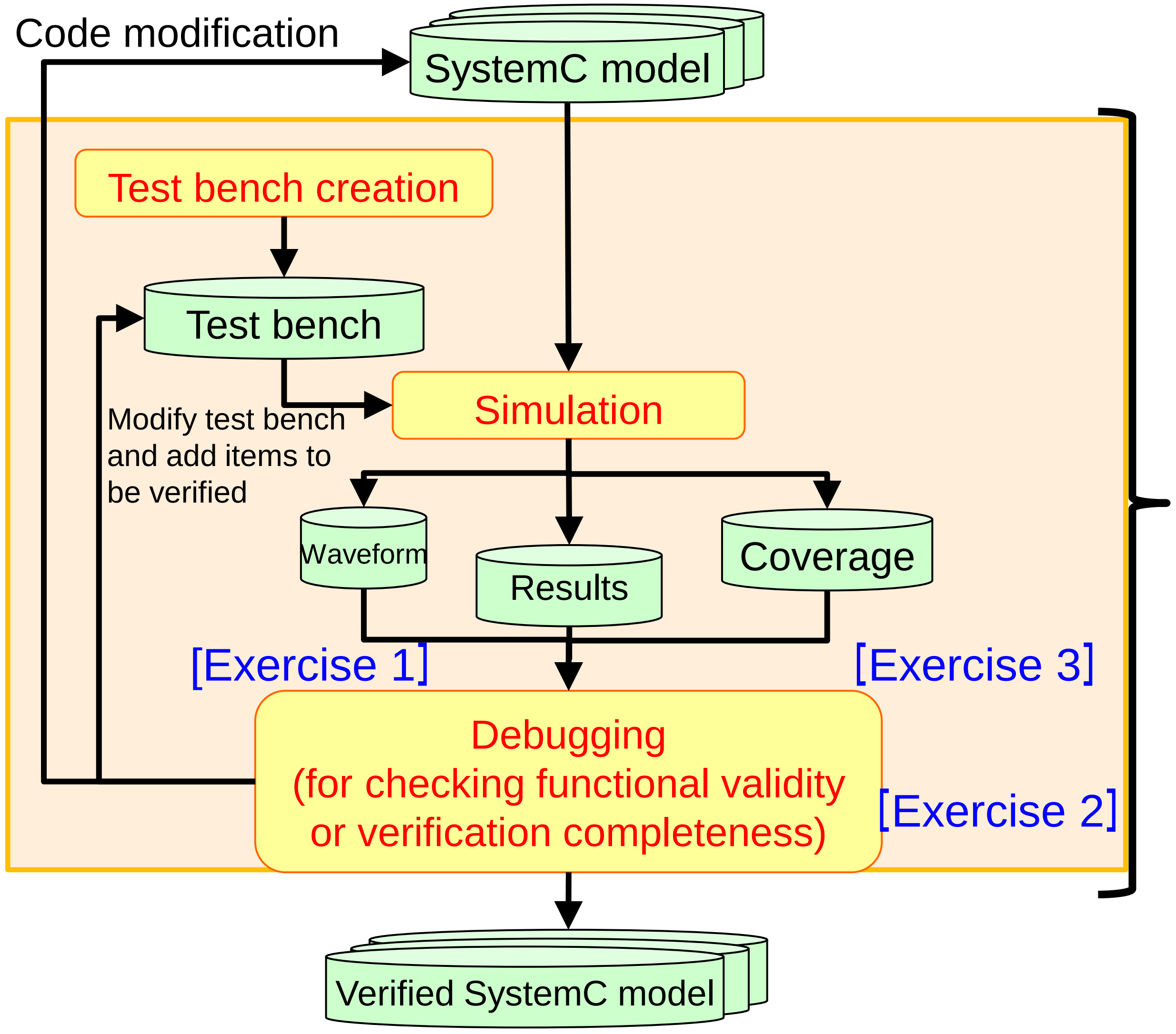
  http://eda.develop.renesas.com/lv1ww/REL/training/en/System_High_level_Design_training.html

    High-level Design Beginner Training Course

      2. High-Level Verification Exercises (Exercise Data)

## SystemC Verification Flow

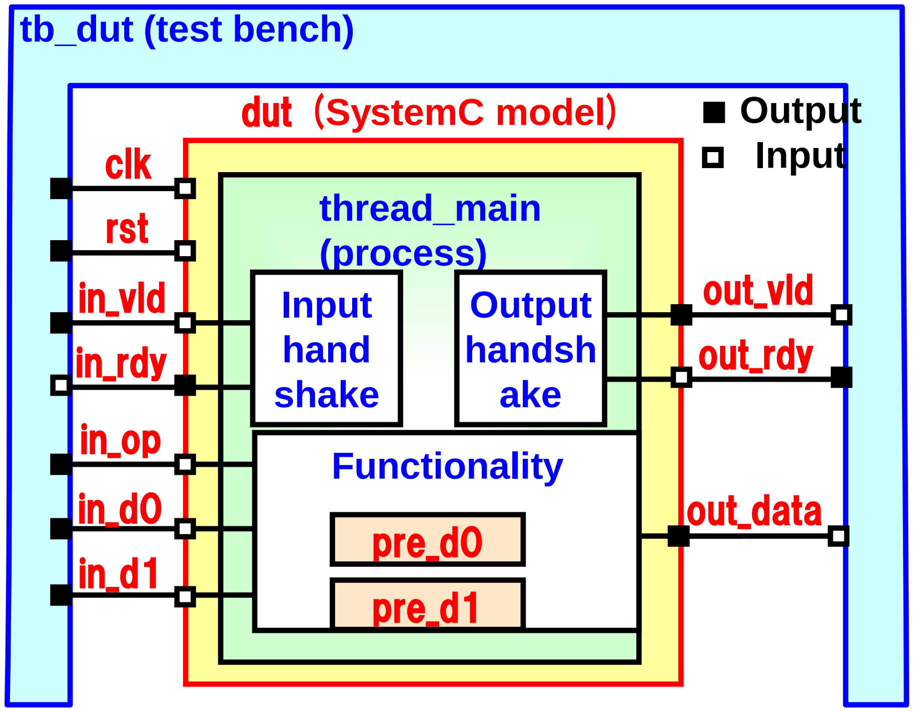
- The correspondence between the SystemC verification flow and exercises is as shown below.



**SystemC verification**
Using the test bench and simulator, verify and debug the created SystemC model.

# SystemC Model Specifications

- Below are the specifications of the SystemC model (dut) used in the verification exercises.
  - These specifications are the same as the specifications of the SystemC model used in the “Part 1 High-level Design Modeling Exercises”. For details, refer to Exercises 2 and 3 in Part 1.



- Accept input data when in_vld==1.
- in_rdy==0 during operation.
- No results can be output when out_rdy==0.
- out_vld==1 only when results are output.
- in_rdy==1 after results are output.

- Specifications
  - Hold the in_d0 and in_d1 values in registers pre_d0 and pre_d1, respectively.
  - Perform one of these operations according to the operation code (in_op):

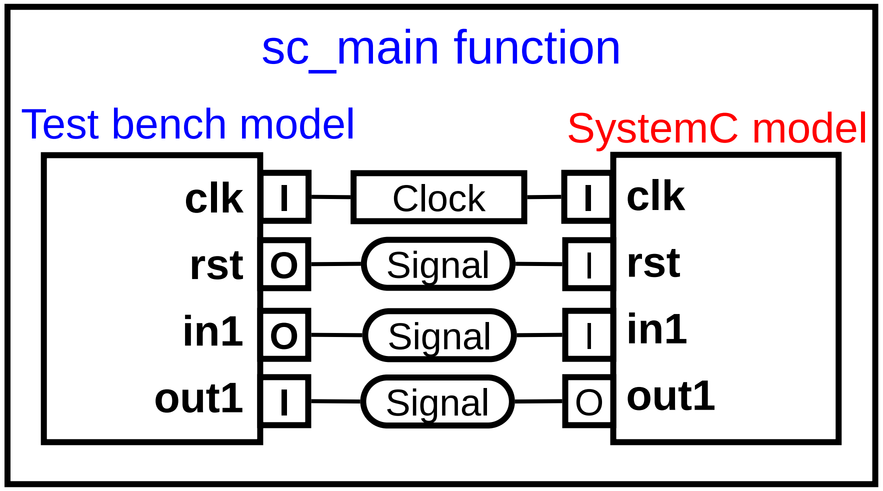
| in_op | Result |
|---|---|
| 0 | The in_d0 value or in_d1 value, whichever is greater |
| 1 | Sum of the in_d0 and in_d1 values |
| 2 | Product of the in_d0 and in_d1 values |
| 3 | Concatenation of the in_d0 and in_d1 bits |
| 4 | Value of previous-value hold register pre_d0 |
| 5 | Value of previous-value hold register pre_d1 |
| Others | Value of in_d0 |

  - Clip the operation results with maximum value 0xFF00 and output them to output port out_data.

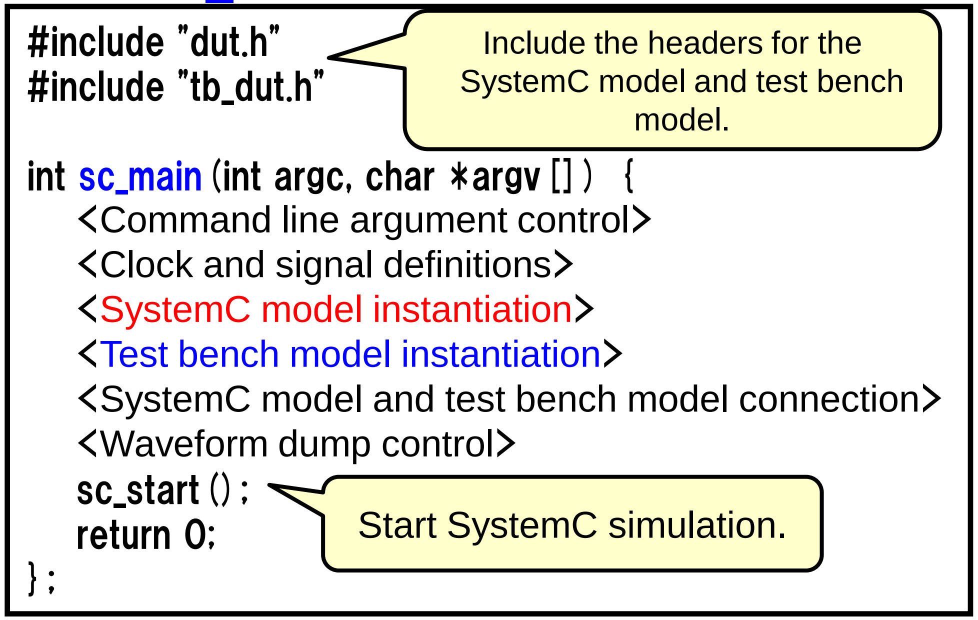
# Test Bench Specifications (1/6)

- To simulate a SystemC model, you need to prepare a test bench.
  - Test bench model: A model which controls the SystemC model.
    - Defines a port having the inputs and outputs which are opposite to those of the SystemC model's port.
    - Connected with the SystemC model. Inputs data, controls this model, and judges the output results.
  - sc_main function: Main function for SystemC simulation
    - Instantiates the SystemC model and test bench. Connects clocks and signals.
    - Controls simulation, command line arguments, waveform dump, etc.

Simplified sc_main function structure



sc_main function code structure



```
#include "dut.h"
#include "tb_dut.h"

int sc_main (int argc, char *argv []) {
    <Command line argument control>
    <Clock and signal definitions>
    <SystemC model instantiation>
    <Test bench model instantiation>
    <SystemC model and test bench model connection>
    <Waveform dump control>
    sc_start () ;
    return 0;
} ;
```

Include the headers for the SystemC model and test bench model.
Start SystemC simulation.

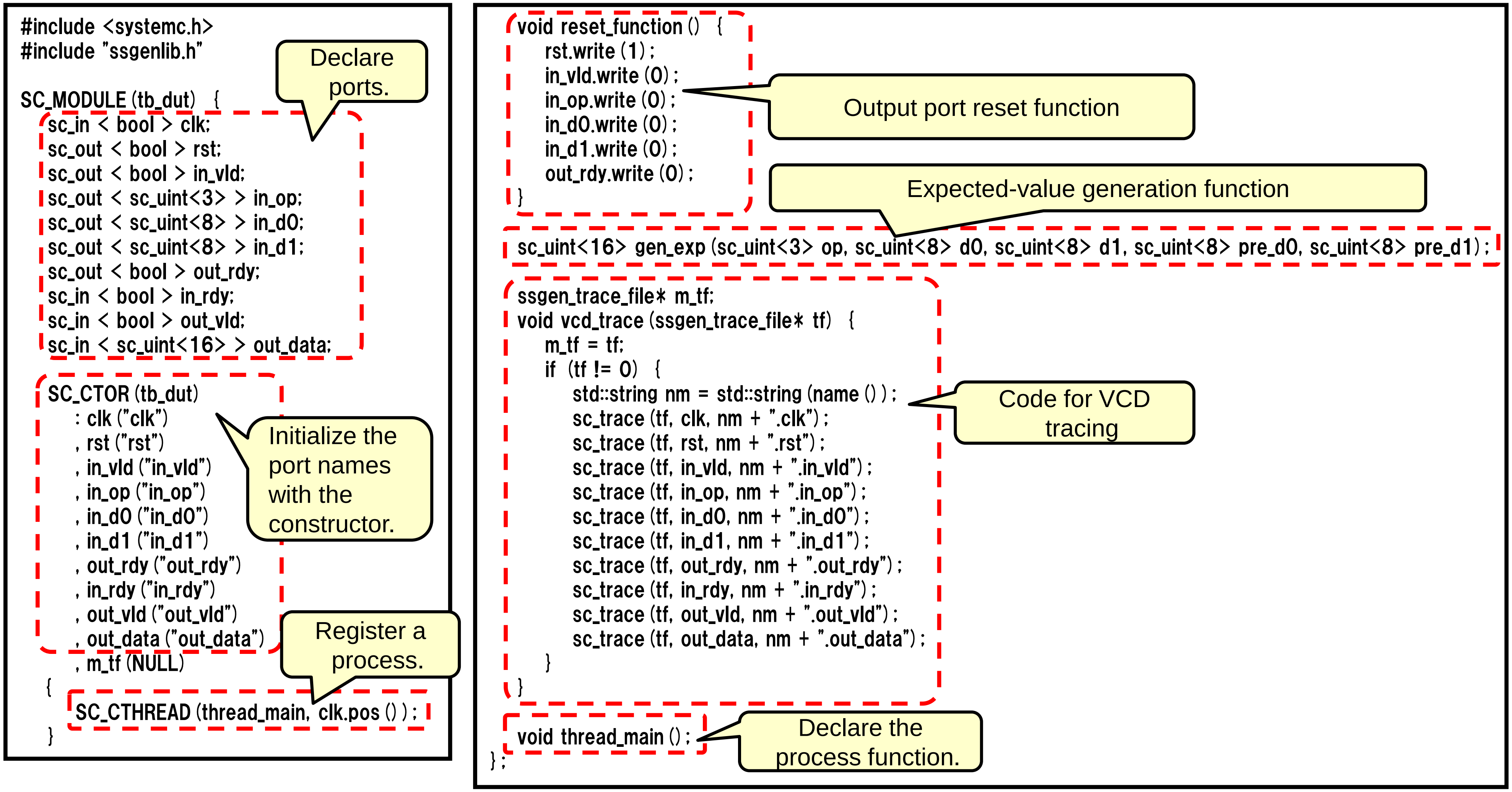
## Test Bench Specifications (2/6)

- Below are the operation specifications of the test bench (tb_dut) used in the verification exercises.
  - Create 16 input patterns and sequentially input them to the SystemC model.
  - Create in_d0 and in_d1 values randomly. Repeatedly create in_op values in order from 0 to 7.
  - Determine the in_vld output time after a randomly selected period of 0 to 4 cycles.
  - Determine the out_rdy output time after a randomly selected period of 0 to 4 cycles.
  - If in_rdy is not asserted during 100 cycles, output the “Deadlock!!” message and forcibly terminate SystemC simulation.
  - If out_vld is not asserted during 100 cycles, output the “Deadlock!!” message and forcibly terminate SystemC simulation.
  - After simulation, check whether the SystemC model output and expected value for each of the 16 input patterns match.
    - If all the values match, display “OK!!”.
    - If values for a pattern do not match, display “NG!!” in the format below.

$$\text{NG!! : }\underbrace{\text{dut=10}}_{\text{SystemC model output}}\;\underbrace{\text{exp=30}}_{\text{Expected value}}\text{ : }\underbrace{\text{op=1 d0=20 d1=10}}_{\text{Input pattern}}$$

# Test Bench Specifications (3/6)

- Below is the header file (tb_dut.h) for the test bench module.



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

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

   SC_CTOR(tb_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")
      , m_tf(NULL)
   {
      SC_CTHREAD(thread_main, clk.pos());
   }
```

```
   void reset_function()  {
      rst.write(1);
      in_vld.write(0);
      in_op.write(0);
      in_d0.write(0);
      in_d1.write(0);
      out_rdy.write(0);
   }

   sc_uint<16> gen_exp(sc_uint<3> op, sc_uint<8> d0, sc_uint<8> d1, sc_uint<8> pre_d0, sc_uint<8> pre_d1);

   ssgen_trace_file* m_tf;
   void vcd_trace(ssgen_trace_file* tf)  {
      m_tf = tf;
      if  (tf != 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");
      }
   }

   void thread_main();
};
```

Declare ports.
Initialize the port names with the constructor.
Register a process.
Output port reset function
Expected-value generation function
Code for VCD tracing
Declare the process function.

RENESAS Group CONFIDENTIAL

# Test Bench Specifications (4/6)

■ Source file (tb_dut.cpp) for the test bench module

```
#include "tb_dut.h"

void tb_dut::thread_main()  {
  reset_function();
  wait();

  sc_uint<16> exp[16];
  sc_uint<16> out[16];
  sc_uint<8> d0[16];
  sc_uint<8> d1[16];
  sc_uint<3> op;
  int j;

  rst.write(0);
  wait();

  for (int i=0; i<16; i++)  {
    j = 0;
    while ( in_rdy.read() == 0 )  {
      if ( j==100 )  {
        printf("Deadlock!! : waiting for in_rdy¥n");
        sc_stop();
        wait();
      }
      j++;
      wait();
    }

    for ( j=0; j<(rand()%5); j++ )  {
      wait();
    }
    in_vld.write( 1 );
```

Include the header file.

Call the reset function.

Release reset.

Create 16 test patterns.

Wait until in_rdy becomes 1.

Assert in_vld after waiting a randomly selected period of 0 to 4 cycles.

```
    d0[i] = rand()%256;
    d1[i] = rand()%256;
    op = i%8;

    exp[i] = gen_exp(op, d0[i], d1[i], d0[i-1], d1[i-1]);

    in_op.write( op );
    in_d0.write( d0[i] );
    in_d1.write( d1[i] );
    wait();

    in_vld.write( 0 );
    for ( j=0; j<(rand()%5); j++ )  {
      wait();
    }
    out_rdy.write( 1 );

    j = 0;
    while ( out_vld.read() == 0 )  {
      if ( j==100 )  {
        printf("Deadlock!! : waiting for out_vld¥n");
        sc_stop();
        wait();
      }
      j++;
      wait();
    }

    out[i] = out_data.read();
    out_rdy.write( 0 );
    wait();
  }
//Continued on the next slide
```

Randomly create input data.
d0/d1: 0 to 255
op: 0～7

Create an expected value.

Add input data to DUT.

Deassert in_vld, wait a randomly selected period of 0 to 4 cycles, and then assert out_rdy.

Wait until out_vld becomes 1.

Obtain output data and then deassert out_rdy.

# Test Bench Specifications (5/6)

■ Source file (tb_dut.cpp) for the test bench module

```
// Continued from the previous slide
  for (int i=0; i<16; i++)  {
      if ( out[i] != exp[i] )  {
          printf("NG!! : dut=%d exp=%d : op=%d d0=%d d1=%d¥n",
                 out[i].to_uint(), exp[i].to_uint(), i%8,
                 d0[i].to_uint(), d1[i].to_uint() );
          sc_stop();
          wait();
      }
  }
  printf("OK!!¥n");

  wait();
  sc_stop();
  wait();
}
```

Check whether all the output data and expected values match.

Call sc_stop to terminate simulation.

```
sc_uint<16> tb_dut::gen_exp(sc_uint<3> op, sc_uint<8> d0, sc_uint<8> d1,
                            sc_uint<8> pre_d0, sc_uint<8> pre_d1) {
    sc_uint<16> out;

    switch( op ) {
    case 0:
        if( d0 > d1 )
            out = d0;
        else out = d1;
        break;
    case 1:
        out = d0 + d1;
        break;
    case 2:
        out = d0 * d1;
        break;
    case 3:
        out = (d0, d1);
        break;
    case 4:
        out = pre_d0;
        break;
    case 5:
        out = pre_d1;
        break;
    default:
        out = d0;
        break;
    }

    if ( out > 0xFF00 )
        out = 0xFF00;
    return out;
}
```

Implement the expected-value generation function.

Create one of the expected values according to the operation code.

Clip the result with 0xFF00.

# Test Bench Specifications (6/E)

- sc_main function (main_dut.cpp)

Include the headers for dut and the test bench.

```
#include "dut.h"
#include "tb_dut.h"
```

Main function for SystemC simulation

```
int sc_main(int argc, char *argv[]) {
    bool vcd_dump  = 0;
    int  vcd_depth = HIER_MAX;
    for (int i = 1; i < argc; i++)
        // vcd dump control
        if (strcmp(argv[i], "-vcd") == 0) {
            vcd_dump = 1;
            if (i < argc - 1 && *argv[i+1] != '-') {
                vcd_depth = atoi(argv[i+1]);
                i++;
            }
        }
        // command line sample (you can customize)
        else if (strcmp(argv[i], "-mode") == 0) {
            i++;
            if (i < argc) {
                if (strcmp(argv[i], "0") == 0) {
                }
                else {
                }
            }
        }
    }
```

Parse the command line arguments.

```
    sc_set_time_resolution(1, SC_PS);
    sc_clock clk("clk", 10, SC_NS);
```

Set the time resolution and define the clock.

```
    dut dut0("dut0");
    tb_dut tb_dut0("tb_dut0");
```

Instantiate dut and the test bench.

Declare signals for connecting dut and the test bench.

```
    sc_signal < bool > rst;
    sc_signal < bool > in_vld;
    sc_signal < sc_uint<3> > in_op;
    sc_signal < sc_uint<8> > in_d0;
    sc_signal < sc_uint<8> > in_d1;
    sc_signal < bool > out_rdy;
    sc_signal < bool > in_rdy;
    sc_signal < bool > out_vld;
    sc_signal < sc_uint<16> > out_data;
```

Connect dut and signals.

```
    dut0.clk(clk);
    dut0.rst(rst);
    dut0.in_vld(in_vld);
    dut0.in_op(in_op);
    dut0.in_d0(in_d0);
    dut0.in_d1(in_d1);
    dut0.out_rdy(out_rdy);
    dut0.in_rdy(in_rdy);
    dut0.out_vld(out_vld);
    dut0.out_data(out_data);
```

Connect the test bench and signals.

```
    tb_dut0.clk(clk);
    tb_dut0.rst(rst);
    tb_dut0.in_vld(in_vld);
    tb_dut0.in_op(in_op);
    tb_dut0.in_d0(in_d0);
    tb_dut0.in_d1(in_d1);
    tb_dut0.out_rdy(out_rdy);
    tb_dut0.in_rdy(in_rdy);
    tb_dut0.out_vld(out_vld);
    tb_dut0.out_data(out_data);
```

Code for VCD tracing

```
    ssgen_trace_file *tf = NULL;
    if (vcd_dump == 1) tf = create_ssgen_trace_file("dut");
#ifndef _MODE_RTL
    dut0.vcd_trace(tf, vcd_depth);
#endif
    tb_dut0.vcd_trace(tf);
```

Start SystemC simulation.

```
    sc_start();
```

Code for VCD tracing

```
    if (vcd_dump == 1) {
        sc_close_vcd_trace_file(tf);
        ssgen_trace_post("dut.vcd");
    }

    return 0;
}
```

# SystemC Verification (1/3)

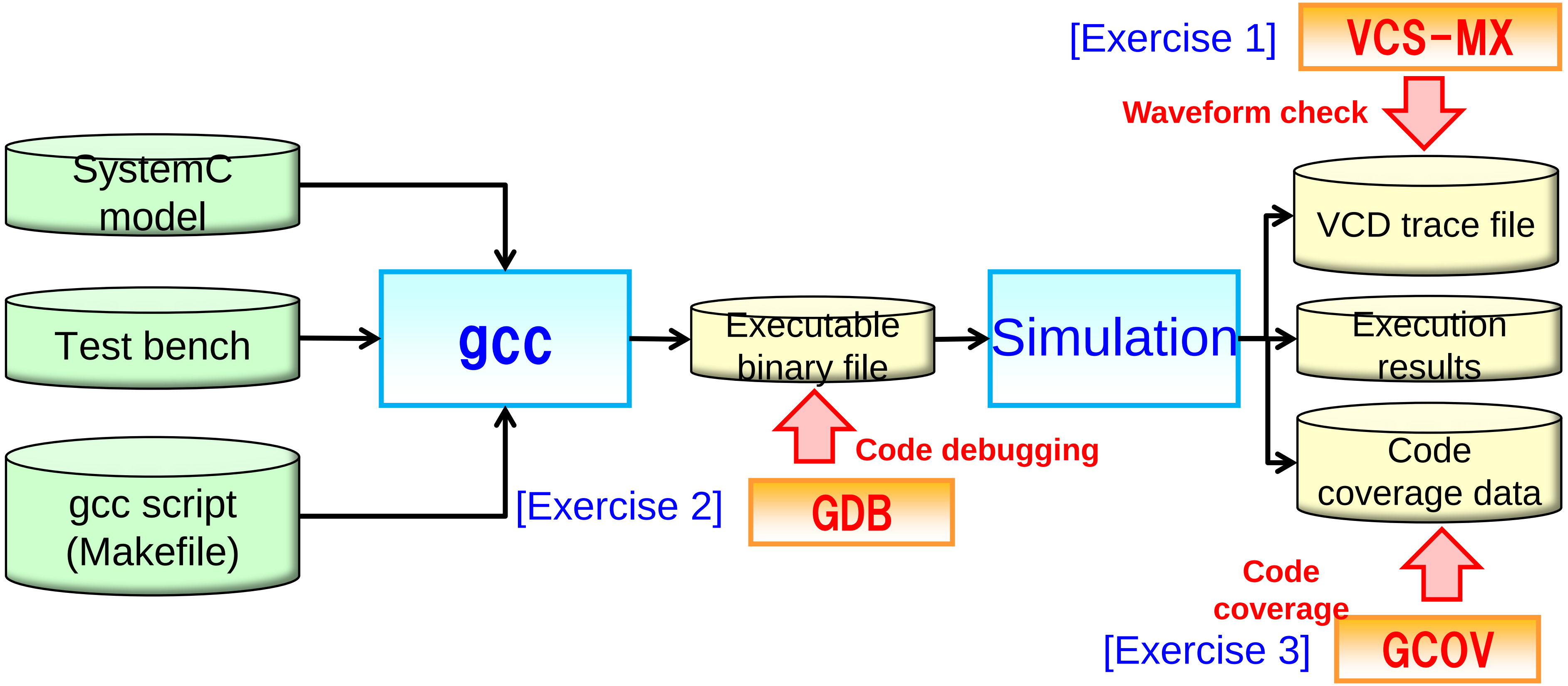
- The three simulators listed below are available for SystemC simulation under Renesas EDA design environment.
  - SSGEN allows you to generate execution scripts for these simulators.
  - Use only gcc for this exercise.

| Simulator | Overview |
|---|---|
| **gcc** | •Compiler for C/C++ programs. Available in SystemC to execute a SystemC model without using an EDA tool.<br>•GDB and DDD are available for code debugging. To check waveforms requires a waveform viewer. |
| **VCS-MX** | •Simulator by Synopsys. Equipped with a highly functional code debugger and waveform viewer (DVE). Enables simulation (CoSim) which involves connecting a SystemC model and Verilog module. |
| **IES** | •Simulator by Cadence. Equipped with a highly functional code debugger and waveform viewer (Simvision). Enables simulation (CoSim) which involves connecting a SystemC model and Verilog module. |

Simulation which uses VCS-MX and IES is not performed in this exercise.
If you want to use these tools, refer to their user’s manuals shown on the “REL EDA Tools Information” page.

## SystemC Verification (2/3)

- How to Verify the SytemC Model with gcc
  - Using a SystemC model, test bench, and gcc script (Makefile), compile the code to create an executable binary file.
  - Perform simulation to generate execution results, VCD trace file, etc.
  - Check the waveform with VCS-MX. Debug the code with GDB. Obtain code coverage with GCOV.

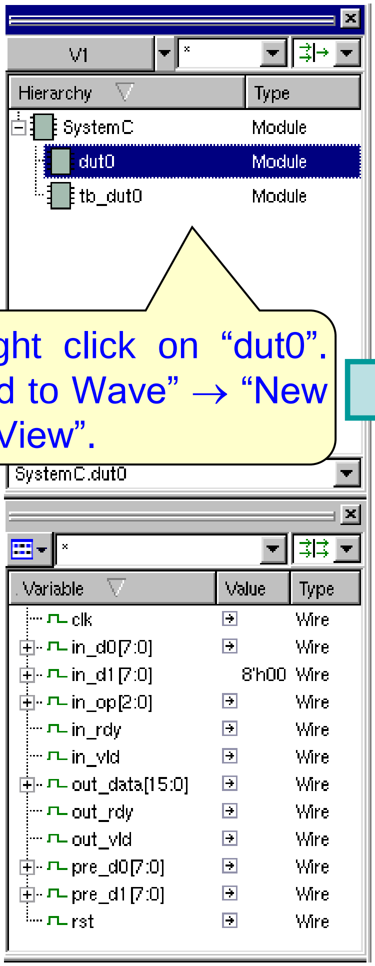


# SystemC Verification (3/E)

- How to Check the Waveform with the VCS-MX Waveform Viewer (DVE)
  - Starting and operating DVE

```
%> source /common/appl/dotfiles/vcs_mx.CSHRC_2011.12-sp1-1
```

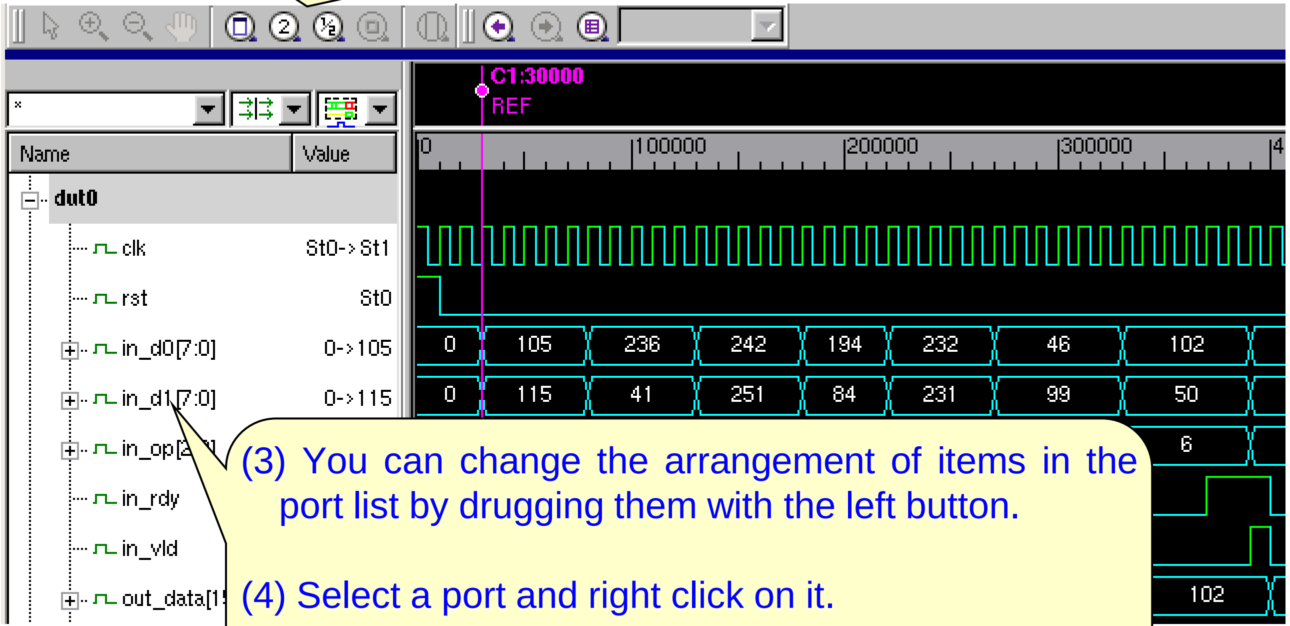
%> bs -os RHEL5 -M 500 dve -vpd *wavefile*  (the VCD trace file can be specified with *wavefile*)



| Variable | Value | Type |
|---|---|---|
| clk | | Wire |
| in_d0[7:0] | | Wire |
| in_d1[7:0] | 8'h00 | Wire |
| in_op[2:0] | | Wire |
| in_rdy | | Wire |
| in_vld | | Wire |
| out_data[15:0] | | Wire |
| out_rdy | | Wire |
| out_vld | | Wire |
| pre_d0[7:0] | | Wire |
| pre_d1[7:0] | | Wire |
| rst | | Wire |

(1) Right click on “dut0”. → “Add to Wave” → “New Wave View”.

(2) Pressing this button displays the entire waveform.



(3) You can change the arrangement of items in the port list by drugging them with the left button.

(4) Select a port and right click on it. → "Set Radix“→ "Decimal“ changes the numerical display to decimal.

## Exercise 1: Performing Simulation and Checking the Waveform

- Using the SystemC model and test bench described before, perform simulation and check the waveform as follows.

Exercise directory: verification/ex1

| | |
|---|---|
| src/dut.h<br>src/dut.cpp | SystemC module |
| tb/tb_dut.h<br>tb/tb_dut.cpp<br>tb/main_dut.cpp | SystemC test bench |
| gcc/Makefile<br>gcc/Makefile.defs | SystemC Makefile |

- Exercise 1-1. Performing Simulation
  - Compile the SystemC code and check its behavior.

- Exercise 1-2. Checking the Waveform
  - To check the waveform, use the VCD trace file which is created after the simulation.

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: Performing Simulation and Checking the Waveform

### ■ Exercise 1-1. Performing Simulation

- Compile the SystemC code.

1. Compile the code.

   ```
   %> cd gcc
   %> make
   ```

   run.exe will be generated.

2. Perform simulation.

   ```
   %> run.exe -vcd
   ```

   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
WARNING: Default time step is used for VCD tracing.
OK!!
SystemC: simulation stopped by user.
```

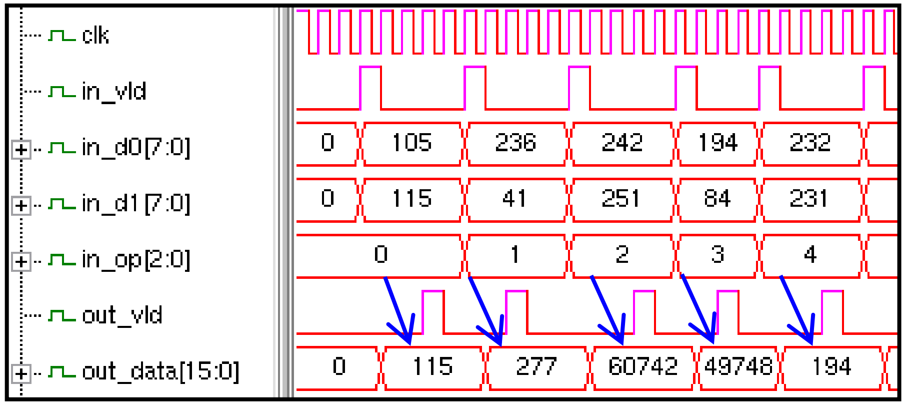
## Exercise 1: Performing Simulation and Checking the Waveform

- Exercise 1-2. Checking the Waveform
  - To check the waveform, use the VCD trace file which is generated after the simulation.

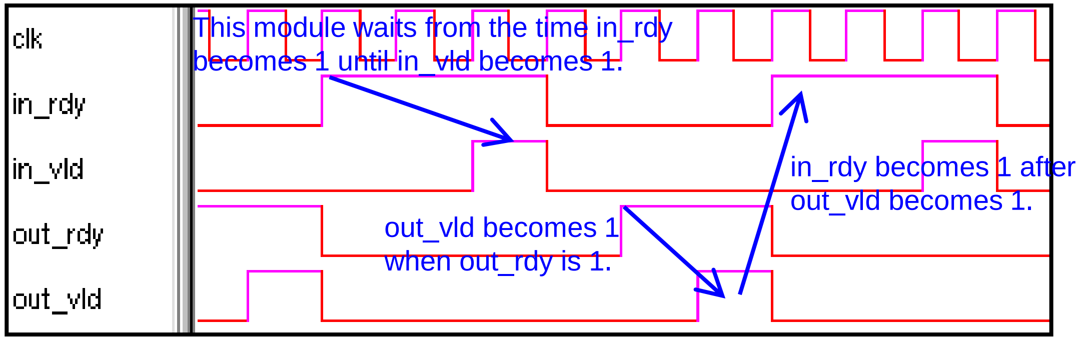
1. Display the waveform (by using VCS-MX).

```
%> source /common/appl/dotfiles/vcs_mx.CSHRC_2011.12-sp1-1
%> bs -os RHEL5 -M 500 dve -vpd dut.vcd
```

2. Check the waveform.
   By checking the SystemC model waveform, make sure that this SystemC module behaves as specified in the “SystemC model specifications”.
   - Does the behavior meet the functional specifications?
   - Does the behavior meet the interface protocol specifications?



Functionality check


clk
in_rdy
in_vld
out_rdy
out_vld
This module waits from the time in_rdy becomes 1 until in_vld becomes 1.
out_vld becomes 1 when out_rdy is 1.
in_rdy becomes 1 after out_vld becomes 1.

Interface protocol check

## Debugging Code (1/4)

- How to Debug Code with GDB
  - Changing the compile script (Makefile) settings
    1. Add “-g” to the compile options to generate debug information in the executable binary file.
    2. Link the SystemC library which includes the debug information.

Makefile settings

```
TARGET_ARCH = linux
CC     = g++
OPT    = -m32 -Wall -g
…
```

1. Add –g to the OPT variable.

Makefile.defs settings

```
SYSTEMC = /common/appl/Renesas/SystemC/SystemC-2.2

INCDIR = -I. -I$(SYSTEMC)/include
LIBDIR = -L. -L$(SYSTEMC)/lib-$(TARGET_ARCH)

LIBS   = -lm $(EXTRA_LIBS)  -lsystemc-gcc412-m32_debug
OBJDIR = obj
…
```

2. Change the SystemC library, specified with the LIBS variable, from –lsystemc to the above.

  - After that, compile the code with “make” to generate an executable binary file.

# Debugging Code (2/4)

## How to Debug Code with GDB

- Below are the GDB commands used in this exercise.

Command can be executed simply by entering this character.

| Command | Overview | Abbreviation | Execution sample |
|---|---|---|---|
| **run** [*arg1* •••] | Runs the program from the beginning. | r | run arg1 arg2 |
| **continue** [*ignorecount*] | Continues to execute the program.<br>The number of breaks to be ignored can be specified with the argument. | c | continue 10 |
| **next** [*count*] | Executes steps (without entering into the function).<br>The execution count can be specified for each line. | n | next 10 |
| **break** [[*filename:*]*linenum*] [if *cond*] | Sets a breakpoint (the current line is selected by default).<br>A breakpoint can be set by specifying if.<br>It can also be set by specifying the function name. | b | break module.cpp:100 if addr == 5 |
| **delete** [*id1*] [*id2*]••• | Deletes a breakpoint (all breakpoints are deleted by default).<br>The ID numbers are specified with id1, id2, etc. | d | delete 1 2 |
| **list** [[*filename*:]*linenum*]<br>**list** [*classname*::]*funcname* | Displays code (the line subsequent to the previously displayed line is selected by default).<br>The code can be specified with the (file name +) line number or function name. | l | list module.cpp:100<br>list module::thread_main |
| **print**[/{x\|u\|t}] *exp* | Displays data (in arbitrary format for C/C++ code)<br>x: Hexadecimal notation, u: Unsigned decimal notation, t: Binary notation | p | print reg.read().m_val |
| **quit** | Terminates GDB debugging. | q | quit |

For details about the GDB commands, refer to “SystemC Code Debugging Know-how”.
http://livelink.renesas.com/Livelink/livelink.exe/open/41175738

# Debugging Code (3/4)

## How to Debug Code with GDB

- Start GDB.

  %> /common/appl/Renesas/SystemC/debugger/bin/gdb *executable_binary*

```
GNU gdb (GDB) 7.4
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /svhome/... /run.exe...done.
(gdb)
```

The system waits for command input.

- Execute the program with GDB.

  (gdb) run

- Terminate GDB.

  (gdb) quit

# Debugging Code (4/E)

## How to Debug Code with GDB

- Examples of setting a breakpoint, checking values, and executing steps with GDB.

SystemC model（**dut.cpp**）

```
18        sc_uint<16> out;
19        sc_uint<8> d0 = in_d0.read();
20        sc_uint<8> d1 = in_d1.read();
21
22        switch ( in_op.read() ) {
23        case 0:
24            if ( d0 > d1 )
25                out = d0;
26            else out = d1;
27            break;
28
29        case 1:
30            out = d0 + d1;
31            break;
          (Omitted)
49        default:
50            out = d0;
51            break;
52        }
53
54        if ( out > 0xFF00 )
55            out = 0xFF00;
```

- Start GDB.
  (gdb) *Wait for command input.*
- Display dut.cpp source code.
  (gdb) list dut.cpp:22
- Set a breakpoint on the 22nd line of dut.cpp.
  (gdb) break 22
- Execute the program until the breakpoint on the 22nd line is reached.
  (gdb) run
- Display the value of input port in_op (sc_in< sc_uint<3> >type).
  (gdb) print in_op.read().m_val
- Execute steps (executed repeatedly until the 54th line is reached).
  (gdb) next
- Display the decimal value of variable out (sc_uint<16> type).
  (gdb) print out.m_val
- Display the hexadecimal value of variable out (sc_uint<16> type).
  (gdb) print/x out.m_val
- Continue execution until the breakpoint on the 22nd line is reached again.
  (gdb) continue
- Execute steps.
  (gdb) next (executed repeatedly until the 54th line is reached)
- Display the decimal value of variable out (sc_uint<16> type).
  (gdb) print out.m_val
- Deletes the breakpoint.
  (gdb) delete
- Run the program to the end.
  (gdb) continue
- Terminate GDB.
  (gdb) quit

Add .read().m_val to display port and signal values.
Add .m_val to display variables of SytemC data types.

## Exercise 2: Debugging Code

- How to Execute the SystemC Program with GDB.

Exercise directory: verification/ex2 (which has the same internal structure as that for exercise 1)

- Exercise 2-1. Creating an Executable Binary File for GDB
  - Using the Makefile for GDB, compile the SystemC code.

- Exercise 2-2. Executing the Program with GDB.
  - Start the program from GDB. Then, set a breakpoint, check the values of variables, and execute steps.

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: Debugging Code

- Exercise 2-1. Creating an Executable Binary File for GDB
  - Using the Makefile for GDB, compile the SystemC code.

```
%> cd gcc
     The Makefile and Makefile.defs in this directory are already modified to suit GDB.
%> make
     run.exe will be generated.
```

## Exercise 2: Debugging Code

- Exercise 2-2. Executing the Program with GDB
  - Start the program from GDB. Then, set a breakpoint, check the values of variables, and execute steps.

```
%> /common/appl/Renesas/SystemC/debugger/bin/gdb run.exe
(gdb) list dut.cpp:22
```

```
17
18              sc_uint<16> out;
19              sc_uint<8> d0 = in_d0.read();
20              sc_uint<8> d1 = in_d1.read();
21
22              switch ( in_op.read() ) {
23              case 0:
24                  if ( d0 > d1 )
25                      out = d0;
26                  else out = d1;
```

Display code around the 22nd line of dut.cpp.

```
(gdb) break 22
```

```
Breakpoint 1 at 0x804a5af: file ../src/dut.cpp, line 22.
```

Set a breakpoint on the 22nd line of dut.cpp.

```
(gdb)  run
```

```
Starting program: /svhome/.../verification/ex2/gcc/run.exe

             SystemC 2.2.0 --- Nov 28 2011 16:00:17
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

Breakpoint 1, dut::thread_main (this=0xffffc358) at ../src/dut.cpp:22
22              switch ( in_op.read() ) {
```

Execute the program until the breakpoint on the 22nd line is reached.

# Exercise 2: Debugging Code

## Exercise 2-2. Executing the Program with GDB

- Start the program from GDB. Then, set a breakpoint, check the values of variables, and execute steps. (Continued from the previous page)

```
(gdb) print in_op.read().m_val
$3 = 0
```

Value of input port in_op is 0.

```
(gdb) next
24                    if ( d0 > d1 )
```

Moved to the 24th line.

```
(gdb) next
26                    else out = d1;
```

Moved to the 26th line.

```
(gdb) next
27                    break;
```

Moved to the 27th line.

```
(gdb) next
54                if ( out > 0xFF00 )
```

Moved to the 54th line.

```
(gdb) print out.m_val
$4 = 115
```

Value of variable out is 115.

```
(gdb) print/x out.m_val
$2 = 0x73
```

Hexadecimal value of variable out is 0x73.

```
(gdb) continue
Breakpoint 1, dut::thread_main (this=0xffffc358) at ../src/dut.cpp:22
22                switch ( in_op.read() ) {
```

Continue execution until the breakpoint on the 22nd line is reached again.

```
18          sc_uint<16> out;
19          sc_uint<8> d0 = in_d0.read();
20          sc_uint<8> d1 = in_d1.read();
21
22          switch ( in_op.read() ) {
23          case 0:
24              if ( d0 > d1 )
25                  out = d0;
26              else out = d1;
27              break;
28
29          case 1:
30              out = d0 + d1;
31              break;
         (Omitted)
49          default:
50              out = d0;
51              break;
52          }
53
54          if ( out > 0xFF00 )
55              out = 0xFF00;
```

## Exercise 2: Debugging Code

- Exercise 2-2. Executing the Program with GDB
  - Start the program from GDB. Then, set a breakpoint, check the values of variables, and execute steps. (Continued from the previous page)

```
(gdb) next
30                  out = d0 + d1;
```

Moved to the 30th line.

```
(gdb) next
31                  break;
```

Moved to the 31st line.

```
(gdb) next
54              if ( out > 0xFF00 )
```

Moved to the 54th line.

```
(gdb) print out.m_val
$2 = 357
```

Value of variable out is 357.

```
(gdb) delete
Delete all breakpoints? (y or n) y
```

Delete the breakpoint.

```
(gdb) continue
Continuing.
OK!!
SystemC: simulation stopped by user.
 [Inferior 1 (process 22646) exited normally]
```

Run the program to the end.

```
(gdb) quit
%>
```

Terminate GDB.

```
18        sc_uint<16> out;
19        sc_uint<8> d0 = in_d0.read();
20        sc_uint<8> d1 = in_d1.read();
21
22        switch ( in_op.read() ) {
23        case 0:
24            if ( d0 > d1 )
25                out = d0;
26            else out = d1;
27            break;
28
29        case 1:
30            out = d0 + d1;
31            break;
          (Omitted)
49        default:
50            out = d0;
51            break;
52        }
53
54        if ( out > 0xFF00 )
55            out = 0xFF00;
```

## Obtaining Code Coverage (1/3)

### ■ How to Obtain Code Coverage with GCOV

●Changing the Compile Script (Makefile)

Add “-fprofile-arcs -ftest-coverage” to the compile options to obtain code coverage with GCOV.

Makefile settings

```
TARGET_ARCH = linux
CC      = g++
OPT     = -m32 -Wall
## please add your include header path to USRDIR, if any
VPATH  = ../src ../tb
USRDIR = -I../src -I../tb -I/common/appl/Renesas/SystemC/utility/ssgen
MACRO  = -D_DEBUG_SIM -D_OSCI -D_MEM_MODEL
GCOV   = -fprofile-arcs -ftest-coverage
CFLAGS = $(OPT) $(MACRO) $(GCOV)
```

Delete “#” at the beginning of this line.

●After that, compile the code with “make” to generate an executable binary file. Then, execute the program.

# Obtaining Code Coverage (2/3)

## How to Obtain Code Coverage with GCOV

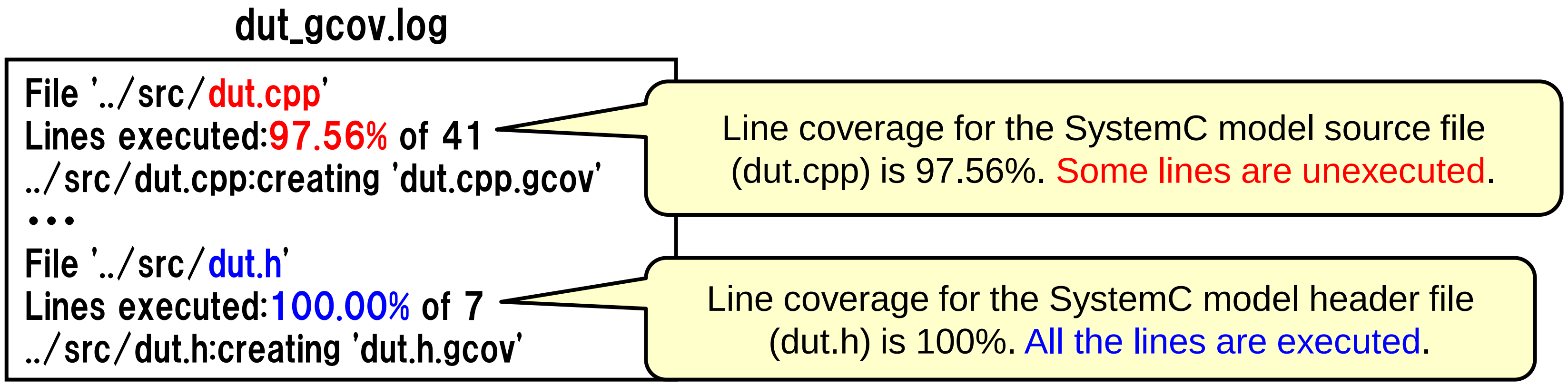
- Execute a gcov command to create a code coverage report.

```
%> gcov -o obj dut.cpp > dut_gcov.log
```

- Check the GCOV execution log.

```
%> vi dut_gcov.log
```

→ Search the log by using "dut.cpp" and "dut.h" as keywords. Check the relevant locations.



**dut_gcov.log**

```
File '../src/dut.cpp'
Lines executed:97.56% of 41
../src/dut.cpp:creating 'dut.cpp.gcov'
• • •
File '../src/dut.h'
Lines executed:100.00% of 7
../src/dut.h:creating 'dut.h.gcov'
```

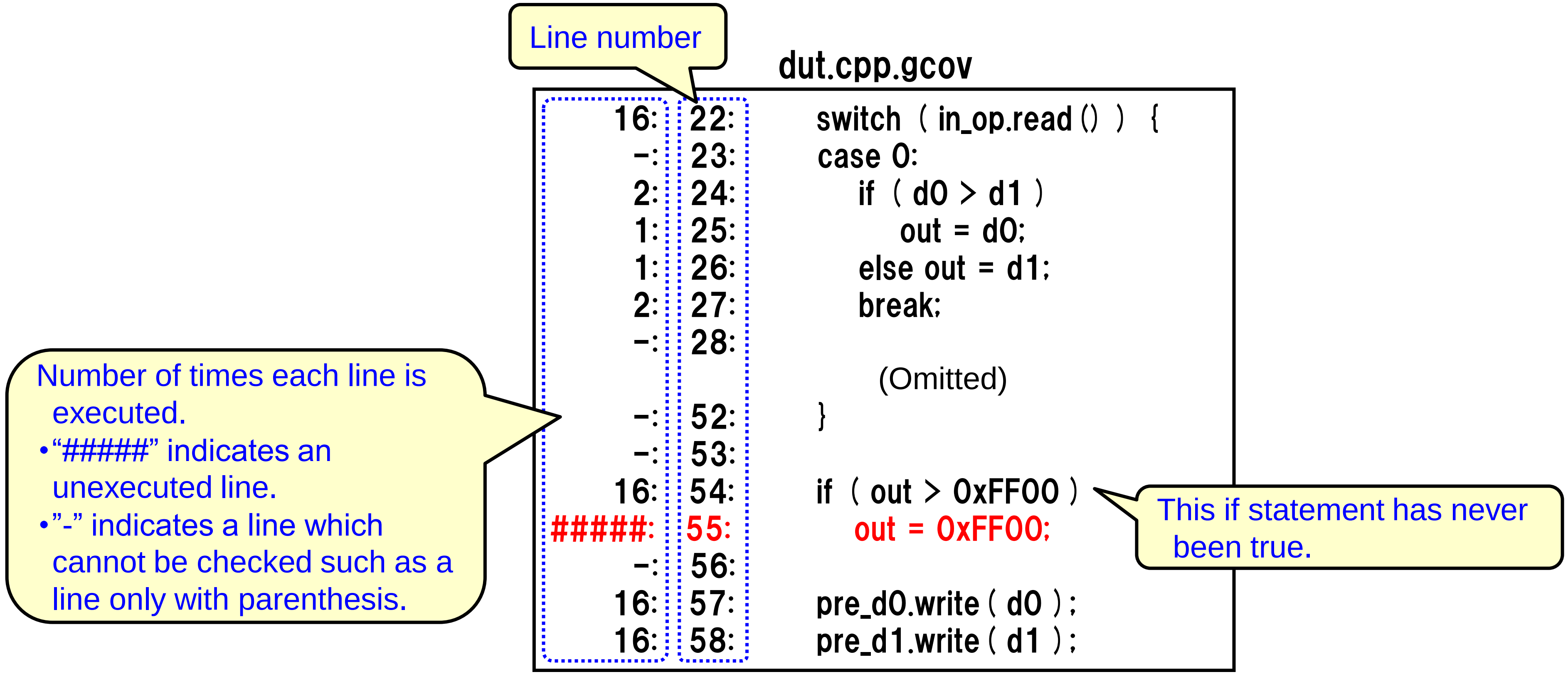
Line coverage for the SystemC model source file (dut.cpp) is 97.56%. Some lines are unexecuted.

Line coverage for the SystemC model header file (dut.h) is 100%. All the lines are executed.

## Obtaining Code Coverage (3/E)

- How to Obtain Code Coverage with GCOV
  - Check the code coverage report on the SystemC model (dut.cpp) to find unexecuted lines.

    %> vi dut.cpp.gcov



Line number

dut.cpp.gcov

```
       16:  22:      switch ( in_op.read() ) {
        -:  23:      case 0:
        2:  24:          if ( d0 > d1 )
        1:  25:              out = d0;
        1:  26:          else out = d1;
        2:  27:          break;
        -:  28:

                            (Omitted)
        -:  52:      }
        -:  53:
       16:  54:      if ( out > 0xFF00 )
    #####:  55:          out = 0xFF00;
        -:  56:
       16:  57:      pre_d0.write( d0 );
       16:  58:      pre_d1.write( d1 );
```

Number of times each line is executed.
- "#####" indicates an unexecuted line.
- "-" indicates a line which cannot be checked such as a line only with parenthesis.

This if statement has never been true.

# Exercise 3: Code Coverage

- Using GCOV, obtain code coverage.

Exercise directory: verification/ex3 (which has the same internal structure as that for exercises 1 and 2)

- Exercise 3-1. Executing GCOV
  - Using the Makefile for GCOV, compile the SystemC code and execute the program. Then, obtain code coverage with GCOV.

- Exercise 3-2. Checking the Code Coverage Results.
  - Check the GCOV execution log and code coverage report. Determine whether all the lines of code have been executed.

- Exercise 3-3. Modifying the Test Bench
  - Modify the test bench so all the lines can be executed. After that, obtain code coverage again and make sure that there are not unexecuted lines.

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: Code Coverage

## Exercise 3-1. Executing GCOV

- Using the Makefile for GCOV, compile the SystemC code and execute the program. Then, obtain code coverage with GCOV.

1. Compile and run the code.

```
%> cd gcc
```

   The Makefile and Makefile.defs in this directory are already modified to suit GCOV.

```
%> make
%> run.exe
```

2. Execute GCOV.

```
%> gcov -o obj dut.cpp > dut_gcov.log
```

# Exercise 3: Code Coverage

## ■ Exercise 3-2. Checking the Code Coverage Results

● Check the GCOV execution log and code coverage report. Determine whether all the lines of code have been executed.

1. Check the GCOV execution log.
   Check the log file by referring to the slide "Obtaining Code Coverage (2)".
   ```
   %> vi dut_gcov.log
   ```
   → Check the "dut.cpp" and "dut.h" line coverage.

2. Check the code coverage report on the SystemC model.
   ```
   %> vi dut.cpp.gcov
   %> vi dut.h.gcov
   ```
   → Check whether there are unexecuted lines (prefixed with "#####").

Not only the dut.*.gcov file but also many *.gcov files are generated under the gcc directory. These *.gcov files contain a code coverage report on the contents of the SystemC library. So, they should be ignored.

# Exercise 3: Code Coverage

## Exercise 3-3. Modifying the Test Bench

- Modify the test bench so all the lines can be executed. After that, obtain code coverage again and make sure that there are not unexecuted lines.

1. Modify the test bench.
   The test bench to be modified is tb/tb_dut.cpp. The reason that the SystemC model contains unexecuted lines is that there is a shortage of input patterns created by the test bench. Modify the test bench by considering what input patterns you need to add.
   → The next slide provides hints for test bench Modification. First, try modifying the test bench without looking at the next slide.

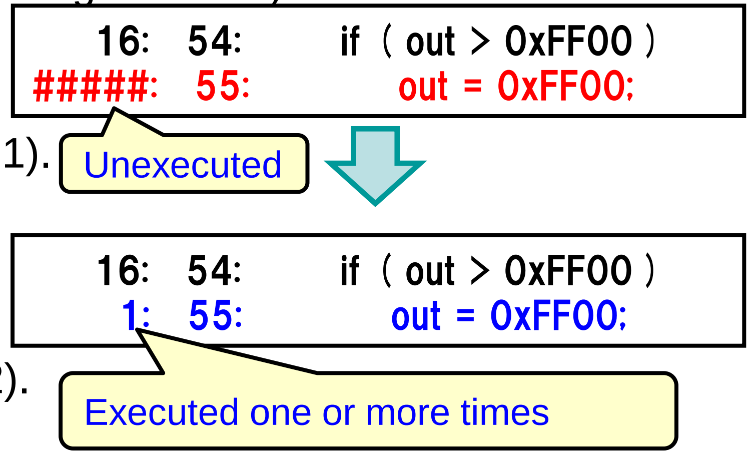
2. Delete the obj directory. (Clear the previous code coverage results.)
   %> rm –rf obj

3. Compile and reexecute the code (refer to exercise 3-1).

4. Reexecute GCOV (refer to exercise 3-1).

5. Check the GCOV execution log (refer to exercise 3-2).

6. Check the code coverage report on the SystemC model (refer to exercise 3-2).



```
   16:   54:      if ( out > 0xFF00 )
#####:   55:         out = 0xFF00;
```

```
   16:   54:      if ( out > 0xFF00 )
    1:   55:         out = 0xFF00;
```

# Exercise 3: Code Coverage

- Exercise 3-3. Hints
  - The test bench (tb_dut.cpp) creates random data and adds 16 input patterns to the SystemC model. Specify the last one input pattern directly with a numerical value and modify the test bench to achieve 100% code coverage.

**tb_dut.cpp**

```
for (int i=0; i<16; i++) {

    ...
    d0 [i] = rand () %256;
    d1 [i] = rand () %256;
    op = i%8;
    ...
    wait () ;
}
```

Create 16 random data items.

**tb_dut.cpp**

```
for (int i=0; i<16; i++) {
    ...
    if (i < 15) {
        d0 [i] = rand () %256;
        d1 [i] = rand () %256;
        op = i%8;
    }
    else {
        d0 [i] = ...;
        d1 [i] = ...;
        op = 3;
    }
...
```

The first 15 items are conventional random data.

The 16th item is direct data for operation code 3.

## Summary

- In the high-level design verification exercises, you have learned these topics:
  - Creating a test bench
  - Using a waveform viewer (DVE)
  - Using a debugger (GDB)
  - Obtaining code coverage

**Proceed to the high-level design synthesis 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.
  verification/.answer/ex3

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 |